Spring jdbc Template

Raul Estrada

Octubre 2020

All the classes in Spring JDBC are divided into four separate packages:

- ***core*** – the core functionality of JDBC. Some of the important classes under this package include *JdbcTemplate*, *SimpleJdbcInsert*, *SimpleJdbcCall* and *NamedParameterJdbcTemplate*.
- ***datasource*** – utility classes to access a datasource. It also has various datasource implementations for testing JDBC code outside the Jakarta EE container.
- ***object*** – DB access in an object-oriented manner. It allows executing queries and returning the results as a business object. It also maps the query results between the columns and properties of business objects.
- ***support*** – support classes for classes under *core* and *object* packages. E.g. provides the *SQLException* translation functionality.

2. Configuration

To begin with, let's start with some simple configuration of the data source (we'll use a MySQL database for this example):

```java
@Configuration
@ComponentScan("com.baeldung.jdbc")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUsername("guest_user");
        dataSource.setPassword("guest_password");

        return dataSource;
    }
}
```

Alternatively, we can also make good use of an embedded database for development or testing – here is a quick configuration that creates an instance of H2 embedded database and pre-populates it with simple SQL scripts:

```java
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
      .setType(EmbeddedDatabaseType.H2)
      .addScript("classpath:jdbc/schema.sql")
      .addScript("classpath:jdbc/test-data.sql").build();
}
```

Finally – the same can, of course, be done using XML configuring for the *datasource*:

```xml
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/springjdbc"/>
    <property name="username" value="guest_user"/>
    <property name="password" value="guest_password"/>
</bean>
```

3. The *JdbcTemplate* and Running Queries

3.1. Basic Queries

The JDBC template is the main API through which we'll access most of the functionality that we're interested in:

- creation and closing of connections
- executing statements and stored procedure calls
- iterating over the *ResultSet* and returning results

Firstly, let's start with a simple example to see what the *JdbcTemplate* can do:

```java
int result = jdbcTemplate.queryForObject(
    "SELECT COUNT(*) FROM EMPLOYEE", Integer.class);
```

and also here's a simple INSERT:

```
public int addEmplyee(int id) {
    return jdbcTemplate.update(
      "INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)", id, "Bill", "Gates", "USA");
}
```

Notice the standard syntax of providing parameters – using the `?` character. Next – let's look at an alternative to this syntax.

3.2. Queries With Named Parameters

To get **support for named parameters**, we'll use the other JDBC template provided by the framework – the *NamedParameterJdbcTemplate*.

Additionally, this wraps the *JbdcTemplate* and provides an alternative to the traditional syntax using "*?*" to specify parameters. Under the hood, it substitutes the named parameters to JDBC "?" placeholder and delegates to the wrapped *JDCTemplate* to execute the queries:

```
SqlParameterSource namedParameters = new MapSqlParameterSource().addValue("id", 1);
return namedParameterJdbcTemplate.queryForObject(
  "SELECT FIRST_NAME FROM EMPLOYEE WHERE ID = :id", namedParameters, String.class);
```

Notice how we are using the *MapSqlParameterSource* to provide the values for the named parameters.

For instance, let's look at below example that uses properties from a bean to determine the named parameters:

```
Employee employee = new Employee();
employee.setFirstName("James");

String SELECT_BY_ID = "SELECT COUNT(*) FROM EMPLOYEE WHERE FIRST_NAME = :firstName";

SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(employee);
return namedParameterJdbcTemplate.queryForObject(
  SELECT_BY_ID, namedParameters, Integer.class);
```

Note how we're now making use of the *BeanPropertySqlParameterSource* implementations instead of specifying the named parameters manually like before.

3.3. Mapping Query Results to Java Object

Another very useful feature is the ability to map query results to Java objects – by implementing *the RowMapper* interface.

For example – for every row returned by the query, Spring uses the row mapper to populate the java bean:

```java
public class EmployeeRowMapper implements RowMapper<Employee> {
    @Override
    public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
        Employee employee = new Employee();

        employee.setId(rs.getInt("ID"));
        employee.setFirstName(rs.getString("FIRST_NAME"));
        employee.setLastName(rs.getString("LAST_NAME"));
        employee.setAddress(rs.getString("ADDRESS"));

        return employee;
    }
}
```

Subsequently, we can now pass the row mapper to the query API and get fully populated Java objects:

```java
String query = "SELECT * FROM EMPLOYEE WHERE ID = ?";
Employee employee = jdbcTemplate.queryForObject(
  query, new Object[] { id }, new EmployeeRowMapper());
```

4. Exception Translation

Spring comes with its own data exception hierarchy out of the box – with *DataAccessException* as the root exception – and it translates all underlying raw exceptions to it.

And so we keep our sanity by not having to handle low-level persistence exceptions and benefit from the fact that Spring wraps the low-level exceptions in *DataAccessException* or one of its sub-classes.

Also, this keeps the exception handling mechanism independent of the underlying database we are using.

Besides, the default *SQLErrorCodeSQLExceptionTranslator*, we can also provide our own implementation of *SQLExceptionTranslator*.

Here's a quick example of a custom implementation, customizing the error message when there is a duplicate key violation, which results in error code 23505 when using H2:

```java
public class CustomSQLErrorCodeTranslator extends SQLErrorCodeSQLExceptionTranslator {
    @Override
    protected DataAccessException
      customTranslate(String task, String sql, SQLException sqlException) {
        if (sqlException.getErrorCode() == 23505) {
          return new DuplicateKeyException(
            "Custom Exception translator - Integrity constraint violation.", sqlException);
        }
        return null;
    }
}
```

To use this custom exception translator, we need to pass it to the *JdbcTemplate* by calling *setExceptionTranslator()* method:

```java
CustomSQLErrorCodeTranslator customSQLErrorCodeTranslator =
  new CustomSQLErrorCodeTranslator();
jdbcTemplate.setExceptionTranslator(customSQLErrorCodeTranslator);
```

5. JDBC Operations Using SimpleJdbc Classes

SimpleJdbc classes provide an easy way to configure and execute SQL statements. These classes use database metadata to build basic queries. *SimpleJdbcInsert* and *SimpleJdbcCall* classes provide an easier way to execute insert and stored procedure calls.

5.1. *SimpleJdbcInsert*

Let's take a look at executing simple insert statements with minimal configuration.

The INSERT statement is generated based on the configuration of *SimpleJdbcInsert* and all we need is to provide the Table name, Column names and values.

First, let's create a *SimpleJdbcInsert*:

```
SimpleJdbcInsert simpleJdbcInsert =
  new SimpleJdbcInsert(dataSource).withTableName("EMPLOYEE");
```

Next, let's provide the Column names and values, and execute the operation:

```java
public int addEmplyee(Employee emp) {
    Map<String, Object> parameters = new HashMap<String, Object>();
    parameters.put("ID", emp.getId());
    parameters.put("FIRST_NAME", emp.getFirstName());
    parameters.put("LAST_NAME", emp.getLastName());
    parameters.put("ADDRESS", emp.getAddress());

    return simpleJdbcInsert.execute(parameters);
}
```

Further, to allow the **database to generate the primary key**, we can make use of the *executeAndReturnKey()* API; we'll also need to configure the actual column that is auto-generated:

```
SimpleJdbcInsert simpleJdbcInsert = new SimpleJdbcInsert(dataSource)
                                        .withTableName("EMPLOYEE")
                                        .usingGeneratedKeyColumns("ID");

Number id = simpleJdbcInsert.executeAndReturnKey(parameters);
System.out.println("Generated id - " + id.longValue());
```

Finally – we can also pass in this data by using the *BeanPropertySqlParameterSource* and *MapSqlParameterSource*.

5.2. Stored Procedures With *SimpleJdbcCall*

Also, let's take a look at executing stored procedures – we'll make use of the *SimpleJdbcCall* abstraction:

```java
SimpleJdbcCall simpleJdbcCall = new SimpleJdbcCall(dataSource)
                                     .withProcedureName("READ_EMPLOYEE");
```

```java
public Employee getEmployeeUsingSimpleJdbcCall(int id) {
    SqlParameterSource in = new MapSqlParameterSource().addValue("in_id", id);
    Map<String, Object> out = simpleJdbcCall.execute(in);

    Employee emp = new Employee();
    emp.setFirstName((String) out.get("FIRST_NAME"));
    emp.setLastName((String) out.get("LAST_NAME"));

    return emp;
}
```

6. Batch Operations

Another simple use case – batching multiple operations together.

6.1. Basic Batch Operations Using *JdbcTemplate*

Using *JdbcTemplate, Batch Operations* can be executed via the *batchUpdate()* API.

The interesting part here is the concise but highly useful *BatchPreparedStatementSetter* implementation:

```java
public int[] batchUpdateUsingJdbcTemplate(List<Employee> employees) {
    return jdbcTemplate.batchUpdate("INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)",
        new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws SQLException {
                ps.setInt(1, employees.get(i).getId());
                ps.setString(2, employees.get(i).getFirstName());
                ps.setString(3, employees.get(i).getLastName());
                ps.setString(4, employees.get(i).getAddress();
            }
            @Override
            public int getBatchSize() {
                return 50;
            }
        });
}
```

6.2. Batch Operations Using *NamedParameterJdbcTemplate*

We also have the option of batching operations with the *NamedParameterJdbcTemplate – batchUpdate()* API.

This API is simpler than the previous one – no need to implement any extra interfaces to set the parameters, as it has an internal prepared statement setter to set the parameter values.

Instead, the parameter values can be passed to the *batchUpdate()* method as an array of *SqlParameterSource*.

```
SqlParameterSource[] batch = SqlParameterSourceUtils.createBatch(employees.toArray());
int[] updateCounts = namedParameterJdbcTemplate.batchUpdate(
    "INSERT INTO EMPLOYEE VALUES (:id, :firstName, :lastName, :address)", batch);
return updateCounts;
```

7. Spring JDBC With Spring Boot

Spring Boot provides a starter *spring-boot-starter-jdbc* for using JDBC with relational databases.

As with every Spring Boot starter, this one also helps us in getting our application up and running quickly.

7.1. Maven Dependency

We'll need the *spring-boot-starter-jdbc* dependency as the primary one as well as a dependency for the database that we'll be using. In our case, this is *MySQL*:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

7.2. Configuration

Spring Boot configures the data source automatically for us. We just need to provide the properties in a *properties* file:

```
spring.datasource.url=jdbc:mysql://localhost:3306/springjdbc
spring.datasource.username=guest_user
spring.datasource.password=guest_password
```

That's it, just by doing these configurations only, our application is up and running and we can use it for other database operations.

The explicit configuration we saw in the previous section for a standard Spring application is now included as part of Spring Boot auto-configuration.

8. Conclusion

In this article, we looked at the JDBC abstraction in the Spring Framework, covering the various capabilities provided by Spring JDBC with practical examples.

Also, we looked into how we can quickly get started with Spring JDBC using a Spring Boot JDBC starter.