**Database Access Approaches in Spring**

Spring framework provides several distinct approaches to database access. The two popular approaches are:

* Unified APIs for directly executing the SQL statements, for example, *JdbcTemplate*
* ORM framework support, for example, [Hibernate](https://howtodoinjava.com/series/hibernate-tutorials/)

The unified APIs provide a straightforward and efficient approach, allowing developers to work with SQL queries in a more direct manner. Key components of this approach include *JdbcTemplate*, *NamedParameterJdbcTemplate* and *JdbcClient*.

Object-relational mapping (ORM) frameworks, such as Hibernate, provide an abstraction layer over relational databases, allowing developers to interact with the database using object-oriented paradigms. This enables developers to map Java objects to database tables, abstracting away the details of SQL queries.

The choice between direct SQL execution and ORM framework support depends on various factors, including the complexity of the application, developer preferences, and specific project requirements. **Direct SQL execution is favored for its simplicity and control, while ORM frameworks excel in scenarios where object-oriented design and abstraction are priorities.**

**2. Difference between JdbcClient Vs. JdbcTemplate**

The *JdbcTemplate* is a core class in Spring Data that simplifies the use of JDBC and eliminates much of the boilerplate code associated with traditional JDBC usage. It provides methods for executing SQL queries, updates, and stored procedures.

It has the following features:

* Executes SQL queries, updates, and stored procedures.
* Parameterized queries using ‘?’ placeholders.
* Row mapping through *RowMapper* or *ResultSetExtractor*.

Consider the following code for example.

**public** **int** getCountOfUsers(String name) {

String sql = "SELECT COUNT(\*) FROM users WHERE name = ?";

**return** jdbcTemplate.queryForObject(sql, Integer.**class**, new Object[]{name});

}

The *JdbcClient* is an enhanced and unified JDBC client API introduced in Spring 6.1, providing a fluent interaction model for both named and positional parameter statements. It aims to streamline JDBC operations further.

It has the following features:

* Unified support for both named and positional parameters.
* Aims to simplify JDBC operations further.
* Introduced as part of the evolving Spring framework.

Consider the following code for example.

**public** **int** getCountOfUsers(String name) {

**return** jdbcClient.sql("SELECT COUNT(\*) FROM users WHERE name = ?")

.param(name)

.query(Integer.**class**)

.single();

}

**The choice between *JdbcTemplate* and *JdbcClient* depends on the project’s context, version of Spring, and developer preferences.** Both tools serve the purpose of simplifying database interactions, each with its own set of advantages and use cases.

**3. Getting Started with *JdbcClient***

**3.1. Maven**

To use *JdbcClient* in a Spring application, we must ensure that the project is using the minimum Spring 6.1 or Spring Boot 3.2 version.

**<parent>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-parent**</artifactId>**

**<version>**3.2.0-M2**</version>**

**<relativePath/>** *<!-- lookup parent from repository -->*

**</parent>**

**<dependencies>**

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-data-jpa**</artifactId>**

**</dependency>**

*<!-- other dependencies -->*

**</dependencies>**

**3.2. Configuration and Initialization**

To configure the *JdbcClient*, ensure that we have a *[DataSource](https://howtodoinjava.com/spring-boot2/datasource-configuration/)*[bean](https://howtodoinjava.com/spring-boot2/datasource-configuration/) in the configuration either using properties file or java bean configuration.

**Create *JdbcClient***

Spring Boot framework auto-discovers the DB connection properties in the *application.properties* and creates the *JdbcClient* bean during the application startup. After this, the *JdbcClient* bean can be autowired in any class.

Instances of the *JdbcClient* class are thread-safe, once configured. A popular approach is to dependency-inject the *DataSource* bean into the Repository/Dao classes. The *JdbcClient* is created in the repository constructor using the statement ***JdbcClient.create()***. The *DataSource* bean is injected automatically by the Spring framework using [constructor injection](https://howtodoinjava.com/spring-core/spring-ioc-vs-di/#6-2-constructor-injection).

@Repository

**public** **class** PersonRepository {

**private** **final** JdbcClient jdbcClient;

**public** PersonRepository(DataSource dataSource) {

**this**.jdbcClient = JdbcClient.create(dataSource);

}

*//...*

}

**3.3. A Simple Example to Use *JdbcClient***

Once the *JdbcClient* instance has been constructed, we can use it for executing the SQL queries using its convenient methods.

**public** Optional<Person> findById(Long id) {

String sql = "select id, first\_name, last\_name, created\_at from person where id = :id";

**return** jdbcClient.sql(sql)

.param("id", id)

.query((rs, rowNum) -> **new** Person(

rs.getInt("id"),

rs.getString("first\_name"),

rs.getString("last\_name"),

rs.getTimestamp("created\_at").toInstant()))

.optional();

}

**4. Passing Parameters to SQL Statements**

The *JdbcCleint* API is quite flexible in accepting the SQL parameters. Let us see a few ways.

**4.1. Positional Parameters**

Positional parameters are placeholders in a query or statement that are identified by their position or order within the statement. These parameters are represented by placeholders like *‘?’* in the SQL statement.

In the following example, the query parameters for *first\_name*, *last\_name* and *created\_at* are registered implicitly in the order in which they are assigned to the method ***StatementSpec.param()***.

String sql = "insert into person(first\_name, last\_name, created\_at) values (?, ?, ?)";

jdbcClient.sql(sql)

.param("Alex")

.param("Dave")

.param(Timestamp.from(Instant.now()))

.update();

We can also pass the parameters in as var-args using the ***StatementSpec.params()*** method as follows:

jdbcClient.sql(sql)

.params("Alex", "Dave", Timestamp.from(Instant.now()))

.update(keyHolder);

Further, it is also possible to pass the parameters as a *List*.

jdbcClient.sql(sql)

.params(List.of("Alex", "Dave", Timestamp.from(Instant.now())))

.update(keyHolder);

If we want to further ensure that we are binding the parameters in the correct order, we can even pass the parameter index to be double sure.

jdbcClient.sql(sql)

.param(1, "Alex")

.param(2, "Dave")

.param(3, Timestamp.from(Instant.now()))

.update()

**4.2. Named Parameters**

Similar to *NamedParameterJdbcTemplate*, the *JdbcClient* also supports the named SQL statement parameters with the placeholder ':paramName' format.

String sql = "insert into person(first\_name, last\_name, created\_at) values (:firstName, :lastName, :createdAt)";

jdbcClient.sql(sql)

.param("firstName, "Alex")

.param("lastName", "Dave")

.param("createdAt", Timestamp.from(Instant.now()))

.update();

It is also possible to pass the named parameters in the form of a *Map* where keys represent the named parameters and values are passed to the query in runtime.

Map<String, ?> paramMap = Map.of(

"firstName", "Alex",

"lastName", "Dave",

"createdAt", Timestamp.from(Instant.now())

);

jdbcClient.sql(sql)

.params(paramMap)

.update();

**4.3. Parameter Source**

To make things even simpler, it is also possible to pass an object (a [record](https://howtodoinjava.com/java/java-record-type/) class, a class with bean properties, or a plain field holder) that has field names matching the named parameters.

In the following example, we perform an INSERT operation into a database using values from the person object.

Person person = **new** Person(**null**, "Clark", "Kent", Instant.now());

jdbcClient.sql(sql)

.paramSource(person)

.update();

The *Person* class is a record type that has matching field names with named parameters.

**public** **record** Person(Long id, String firstName, String lastName, Instant createdAt) {

}

Similarly, we can use the *SimplePropertySqlParameterSource* and *BeanPropertySqlParameterSource* strategies as well.

SqlParameterSource namedParameters = **new** BeanPropertySqlParameterSource(person);

jdbcClient.sql(sql)

.paramSource(namedParameters)

.update();

**5. Mapping ResultSet to Objects**

**5.1. Using Custom RowMapper**

When querying the rows from the database, we can retrieve the values of the columns using the *ResultSet* as shown in *section 2.3*. But if we want to add more flexibility and clean code then we can consider using the RowMapper.

The following *PersonRowMapper* class implements the *RowMapper* interface and overrides the mapRow() method which contains the logic to map a database row to a *Person* instance.

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Timestamp;

**import** java.time.Instant;

**import** com.howtodoinjava.model.Person;

**import** org.springframework.jdbc.core.RowMapper;

**public** **class** PersonRowMapper **implements** RowMapper<Person> {

**private** PersonRowMapper() {}

**private** **static** **final** PersonRowMapper INSTANCE = **new** PersonRowMapper();

**public** **static** PersonRowMapper getInstance() {

**return** INSTANCE;

}

@Override

**public** Person mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

**return** **new** Person(

rs.getLong("id"),

rs.getString("first\_name"),

rs.getString("last\_name"),

getInstantFromTimestamp(rs.getTimestamp("created\_at"))

);

}

**private** Instant getInstantFromTimestamp(Timestamp timestamp) {

**return** (timestamp != **null**) ? timestamp.toInstant() : **null**;

}

}

Now, we can use the *PersonRowMapper* in the *query()* method and Spring takes care of using the mapper internally and we get the *Person* instance directly.

String querySql = "select id, first\_name, last\_name, created\_at from person where id = :id";

Optional<Person> personOptional = jdbcClient.sql(querySql)

.param("id", 1)

.query(PersonRowMapper.getInstance())

.optional();

**5.2. Using Class Mapping**

If creating a *PersonRowMapper* seems like extra effort because you have a direct field mapping between the class fields and database columns, you can directly pass the class type of the *query()* method and it will also work.

String querySql = "select id, first\_name, last\_name, created\_at from person where id = :id";

Optional<Person> personOptional = jdbcClient.sql(querySql)

.param("id", 1)

.query(Person.**class**)

.optional();

**6. SQL Query and Update Operations**

JdbcClient supports all types of DB manipulations like selecting, creating, updating, and deleting records.

* The **query()** operation executes a given SQL query, with several result options available in the returned query specification such as mapped class, *RowMapper*, *RowCallbackHandler* and *ResultSetExtractor*.
* The ***update()*** operation executes the provided SQL statement as an update. It always returns an int value indicating the number of rows affected.

We have also seen the example of these methods in previous sections so let’s not repeat again.

**7. Consideration for Batch Inserts and Stored procedures**

The *JdbcClient* is a flexible but very simplified facade for only JDBC query/update statements. If you need to do more complex things, like performing [batch operations](https://howtodoinjava.com/series/spring-batch-tutorials/) or [calling stored procedures](https://howtodoinjava.com/hibernate/jpa-21-namedstoredprocedurequery-annotation-example/), *JdbcClient* might not have all the features you need.

In those cases, you may want to use other tools provided by Spring, such as *SimpleJdbcInsert* or *SimpleJdbcCall*.

Alternatively, you can go back to using the more basic *JdbcTemplate* directly for tasks that *JdbcClient* doesn’t cover as thoroughly. Think of it like having different tools in a toolbox – you pick the one that’s best for the job you’re doing