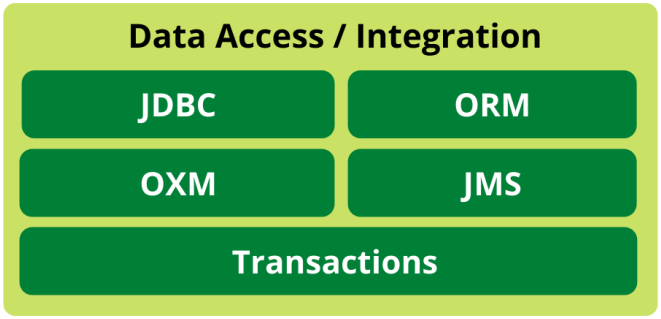
Spring Data access - JDBC

Spis treści

Spring Data Access

Patrząc na ogólny zarys modułów Spring Framework, można zauważyć wydzielony moduł *Data Access / Integration*, który dotyczy interakcji pomiędzy warstwą danych a warstwą logiki biznesowej.



Obraz 1. Moduły Data Access / Integration

W tej części przyjrzymy się bliżej modułom *JDBC* i *Transactions*.

Spring JDBC

Zależność do biblioteki Spring JDBC:

```
// https://mvnrepository.com/artifact/org.springframework/spring-jdbc
implementation "org.springframework:spring-jdbc:$springVersion"
```

Spring ułatwia korzystanie z tradycyjnego JDBC, biorąc na siebie odpowiedzialność za niskopoziomowe operacje, co ładnie obrazuje tabelka z dokumentacji Spring'a:

Tabela 1. Podział obowiązków Spring|Deweloper:

Akcja	Spring	Deweloper
Definicja parametrów połączenia	—	X
Otwarcie połączenia	X	—
Definicja zapytania SQL	—	X

Akcja	Spring	Deweloper
Deklaracja parametrów i ich wartości	—	X
Przygotowanie i uruchomienie zapytania	X	—
Obsłużenie wyjątków	X	—
Zarządzanie transakcjami	X	—
Zamknięcie połączenia, zapytania i wyników	X	—

Sposoby korzystania ze Spring JDBC

- *JdbcTemplate*
- *SimpleJdbcInsert* i *SimpleJdbcCall*
- *NamedParameterJdbcTemplate*

JdbcTemplate (*org.springframework.jdbc.core*)

Obiekt *JdbcTemplate* jest punktem startowym, poprzez który mamy dostęp do głównych funkcjonalności API tj. zarządzanie połączeniami, wywoływanie zapytań i obsługa ResultSet. Użycie jego jest podstawowym i najpopularniejszym podejściem korzystania ze Spring JDBC, a zarazem jest trzonem działania dla kolejnych klas: *SimpleJdbcInsert*, *SimpleJdbcCall* i *NamedParameterJdbcTemplate*.

Przejdźmy do przykładów jak z tego korzystać, ale najpierw trzeba przygotować połączenie do bazy. Wystarczy nam kilka parametrów połączenia, a resztą zajmie się Spring.

Przykład definiowania **dataSource***:*

```
package pl.zajavka;

import org.postgresql.Driver; ①
import org.springframework.context.annotation.*;
import org.springframework.jdbc.datasource.SimpleDriverDataSource;

@Configuration
@ComponentScan(basePackageClasses = Runner.class)
public class DataSourceConfiguration {

    @Bean
    public SimpleDriverDataSource databaseDataSource() {
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource(); ②
        dataSource.setDriver(new Driver()); ③
        dataSource.setUrl("jdbc:postgresql://localhost:5432/zajavka");
        dataSource.setUsername("postgres");
        dataSource.setPassword("postgres");
        return dataSource;
    }
}
```

① Gradle: "org.postgresql:postgresql:$postgresqlVersion",

② *org.springframework.jdbc.datasource.* `SimpleDriverDataSource` jest klasą odpowiedzialną za przygotowanie połączenia do bazy danych,

③ *new Driver()* jest sterownikiem do bazy *PostgreSQL*.

Zależność do biblioteki PostgreSQL JDBC Driver:

```
// https://mvnrepository.com/artifact/org.postgresql/postgresql
implementation "org.postgresql:postgresql:$postgreSQLVersion"
```

Bean *databaseDataSource()* będzie używany w kolejnych przykładach. Załóżmy też, że w naszej bazie mamy tabelę *PERSON*, na bazie której będą opierać się kolejne przykłady.

Tabela 2. Tabela **Person** *z kolumnami* **ID**, **NAME** *i* **AGE**:

ID	NAME	AGE
1	Stefan	23
2	Agnieszka	46
3	Tomasz	52

Klasa **Person** *z polami* **id**, **name** *i* **age**:

```
package pl.zajavka.springjdbc;

import lombok.*;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Person {
    private Long id;
    private String name;
    private Integer age;
}
```

DDL tworzący wspomnianą tabelę:

```
DROP TABLE IF EXISTS person CASCADE;

CREATE TABLE person
(
    id SERIAL NOT NULL,
    age       INT,
    name      VARCHAR(255),
    PRIMARY KEY (id)
);
```

Zastosowany tutaj został typ danych `SERIAL`. Cytując źródło:

> The SERIAL data type stores a sequential integer, of the INT data type, that is automatically assigned by the database server when a new row is inserted.

Czyli można to rozumieć jak typ danych `INT`, tylko, że to baza danych będzie odpowiedzialna za automatyczne zwiększanie wartości. Jest to bardzo przydatne w przypadku kluczy głównych, bo

programista nie musi się martwić wtedy strategią zwiększania ich wartości.

Klasa Runner

```
public class Runner {

    public static void main(String[] args) {
        ApplicationContext context
            = new AnnotationConfigApplicationContext(DataSourceConfiguration.class);

        JdbcTemplateExamples jdbcTemplateExamples = context.getBean(JdbcTemplateExamples.class);
        jdbcTemplateExamples.insertExample();
        jdbcTemplateExamples.updateExample();
        jdbcTemplateExamples.selectExample();
        jdbcTemplateExamples.deleteExample();
    }
}
```

Przykład użycia **JdbcTemplate***:*

```
@Repository
@RequiredArgsConstructor
public class JdbcTemplateExamples {

    private final SimpleDriverDataSource simpleDriverDataSource;

    public void insertExample() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(simpleDriverDataSource); ①
        String sqlInsert = "INSERT INTO PERSON (NAME, AGE) VALUES (?, ?)";
        jdbcTemplate.update(sqlInsert, "Roman", 25); ②
    }

    public void updateExample() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(simpleDriverDataSource);
        String sqlUpdate = "UPDATE PERSON SET AGE = ? where NAME = ?";
        jdbcTemplate.update(sqlUpdate, 29, "Roman"); ③
    }

    public void selectExample() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(simpleDriverDataSource);
        String sqlSelect = "SELECT * FROM PERSON";
        RowMapper<Person> personRowMapper = (resultSet, rowNum) -> Person.builder()
            .id(resultSet.getLong("ID"))
            .name(resultSet.getString("NAME"))
            .age(resultSet.getInt("AGE"))
            .build();
        List<Person> result = jdbcTemplate.query(sqlSelect, personRowMapper); ④
        System.out.println(result);

        BeanPropertyRowMapper<Person> personBeanPropertyRowMapper
            = BeanPropertyRowMapper.newInstance(Person.class); ⑤
        List<Person> result2 = jdbcTemplate.query(sqlSelect, personBeanPropertyRowMapper);
        System.out.println(result2);
    }

    public void deleteExample() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(simpleDriverDataSource);
        String sqlDelete = "DELETE FROM PERSON where NAME = ?";
        jdbcTemplate.update(sqlDelete, "Roman"); ⑥
```

```
    }

}
```

1. Utworzenie obiektu `JdbcTemplate`, który w konstruktorze przyjmuje wcześniej przygotowany obiekt `simpleDriverDataSource`.
2. Przykład dodania nowego wiersza. Zmienna `sqlInsert` zawiera query z insertem do tabeli `person`. Zawarte w nim znaki zapytania **?** to placeholdery (po polsku to będzie chyba *symbol zastępczy*) na wartości kolumn tworzonego wiersza. Te wartości podaje się jako kolejne (zaraz po query) argumenty metody `JdbcTemplate#update()`. W naszym przypadku to NAME = *"Roman"* i AGE = *25*. Po wywołaniu tego inserta, w bazie danych pojawi się wpis:

ID	NAME	AGE
1	Roman	25

3. Przykład aktualizacji istniejącego wiersza. Zmienna `sqlUpdate` zawiera query z aktualizacją tabeli `person` i jej wywołanie podlega tym samym zasadom co insert.

ID	NAME	AGE
1	Roman	29

4. Przykład odczytu danych z tabeli. Jeżeli chodzi o selecty, czyli odczyt danych, to używamy metody `JdbcTemplate#query()`. Otrzymane, w wyniku zapytania, dane przydałoby się jakoś przygotować do późniejszego użycia. Do tego służy interfejs `RowMapper` z metodą `mapRow()`. To, co widać w przykładzie to specjalna implementacja `RowMapper` utworzona na potrzeby tabeli `person`. Dzięki niej `jdbcTemplate` wie, w jaki sposób ma przekształcić otrzymane wyniki zapytania do obiektu javowego `Person`. Jako, że odczytanych wierszy może być wiele, metoda `JdbcTemplate#query()` zwraca listę obiektów `Person`. Zwróć uwagę, że sposób przemapowania odczytanych danych na obiekty typu `Person` został tutaj napisany ręcznie.
5. Wykorzystano tutaj `BeanPropertyRowMapper`, jeżeli nazwy kolumn są identyczne jak nazwy pól w klasie, to zamiast pisać kod odpowiedzialny za mapowanie ręcznie, możemy wykorzystać `BeanPropertyRowMapper`. Spring ogarnie to wtedy za nas. `BeanPropertyRowMapper`, jest implementacją `RowMapper`.
6. Przykład usunięcia wiersza. Ostatnie co można zrobić z wierszem w bazie to go usunąć. I tutaj wraca metoda `JdbcTemplate#update()`, ale tym razem z query zawierającym **delete**. Sposób użycia pozostaje identyczny. Wpis z imieniem "Roman" zostanie usunięty.

SimpleJdbcInsert i SimpleJdbcCall (*org.springframework.jdbc.core.simple*)

Klasy *SimpleJdbcInsert* i *SimpleJdbcCall* upraszczają tworzenie zapytań dzięki użyciu dostępnych metadanych bazy.

Metadane to dane o danych / informacje o informacjach. Załóżmy, że masz plik Excel z jakimiś danymi, to informacja o dacie utworzenia tego pliku jest jego metadaną.

Serwery baz danych przechowują bazy danych, ale też informacje o tych bazach. Czyli np. nazwy i typy kolumn oraz tabel itp. Dzięki temu chcąc utworzyć nowy wiersz z zastosowaniem `SimpleJdbcInsert`,

wystarczy podać nazwę tabeli i wartości do wstawienia do bazy w postaci mapy zawierającą nazwy kolumn i ich wartości. `SimpleJdbcCall` pozwala na szybkie wywołanie procedury lub funkcji przechowywanej w bazie podając jej nazwę i wymagane parametry.

Z funkcji bazodanowych korzystaliśmy już wcześniej przy pracy z PostgreSQL.

Procedura składowana (*stored procedure*) jest pewnego rodzaju programem, który może zostać napisany na bazie danych. Procedura może realizować logikę, która inaczej mogłaby być zaimplementowana po stronie aplikacji. Procedury można stosować jeżeli chcemy zaoszczędzić czas i pamięć przy rozległych i złożonych procesach, które wymagałyby wielu instrukcji SQL. Logikę taką możemy zapisać w procedurze, a następnie aplikacja może taką procedurę wielokrotnie wywołać. W takiej procedurze możemy przykładowo zapisywać instrukcje warunkowe - stąd nazywamy to stworzeniem logiki po stronie bazy danych.

SimpleJdbcInsert

Przykład użycia **SimpleJdbcInsert***:*

```
@Repository
@AllArgsConstructor
public class SimpleJdbcInsertExamples {

    private final SimpleDriverDataSource simpleDriverDataSource;

    public void simpleJdbcInsertExample() {
        SimpleJdbcInsert simpleJdbcInsert = new SimpleJdbcInsert(simpleDriverDataSource); ①
        simpleJdbcInsert.setTableName("PERSON"); ②

        Map<String, Object> params = new HashMap<>(); ③
        params.put("NAME", "Karol");
        params.put("AGE", 15);

        int result = simpleJdbcInsert.execute(params); ④
        System.out.println("Rows affected: " + result);
    }

    // ...

}
```

① Utworzenie obiektu `SimpleJdbcInsert`, który w konstruktorze przyjmuje wcześniej przygotowany obiekt `simpleDriverDataSource`.

② Określenie, do jakiej tabeli chcemy dodać wiersz. W przykładzie jest to tabela `person`.

③ Kolejnym krokiem jest przygotowanie danych do wstawienia, czyli *"Karol"* i *15*.

④ Przykład dodania nowego wiersza z wykorzystaniem metody `SimpleJdbcInsert#execute()` i z użyciem mapy z wartościami do wstawienia.

ID	NAME	AGE
3	Karol	15

Nie umieszczam przykładu klasy `Runner` ponownie, gdyż jest on analogiczny do poprzednich przykładów.

Uproszczeniem tutaj jest pozbycie się potrzeby tworzenia query z insertem, tak jak to było w przypadku `JdbcTemplate`:

```
String sqlInsert = "insert into PERSON (NAME, AGE) values (?, ?);";
```

Ten sam efekt można osiągnąć, ale z użyciem *BeanPropertySqlParameterSource*:

Przykład użycia **BeanPropertySqlParameterSource***:*

```
@Repository
@AllArgsConstructor
public class SimpleJdbcInsertExamples {

    private final SimpleDriverDataSource simpleDriverDataSource;

    // ...

    public void simpleJdbcInsertWithBeanPropertySqlParameterSourceExample() {
        SimpleJdbcInsert simpleJdbcInsert = new SimpleJdbcInsert(simpleDriverDataSource);
        simpleJdbcInsert.setTableName("PERSON");

        Person person = Person.builder() ①
            .name("Stefan")
            .age(55)
            .build();

        BeanPropertySqlParameterSource paramSource = new BeanPropertySqlParameterSource(person); ②
        int result = simpleJdbcInsert.execute(paramSource); ③
        System.out.println("Rows affected: " + result);
    }

}
```

① Utworzenie obiektu `Person`, który później będzie dodany do bazy.

② Przygotowanie obiektu `BeanPropertySqlParameterSource`.

③ Przykład dodania nowego wiersza z użyciem obiektu `BeanPropertySqlParameterSource`.

Użycie `BeanPropertySqlParameterSource` uprasza nam prace, bo nie ma potrzeby tworzenia specjalnej mapy z nazwami kolumn i jej wartościami. Za to można operować na gotowym obiekcie javowowym, który najprawdopodobniej będziemy już mieli na tym etapie gotowy.

SimpleJdbcCall

Zacznijmy od dodania swojej własnej funkcji do bazy danych:

Przykładowa funkcja licząca sumę dwóch wartości:

```
create function calc_sum(value1 int, value2 int)
    returns int
    language plpgsql
as
$$
begin
    return value1 + value2;
```

```
end;
$$;
```

Przykład użycia **SimpleJdbcCall** *do wywołania funkcji zapisanej w bazie danych:*

```
@Repository
@AllArgsConstructor
public class SimpleJdbcCallExamples {

    private final SimpleDriverDataSource simpleDriverDataSource;

    public void simpleJdbcCallExample() {
        SimpleJdbcCall simpleJdbcCall = new SimpleJdbcCall(simpleDriverDataSource); ①
        simpleJdbcCall.withFunctionName("calc_sum"); ②

        SqlParameterSource sqlParameterSource = new MapSqlParameterSource() ③
            .addValue("value1", 2)
            .addValue("value2", 3);
        Integer result = simpleJdbcCall.executeFunction(Integer.class, sqlParameterSource); ④
        System.out.println("Result: " + result);
    }

}
```

① Utworzenie obiektu `SimpleJdbcCall`, który w konstruktorze przyjmuje wcześniej przygotowany obiekt `simpleDriverDataSource`.

② Określenie, jaką funkcję chcemy wywołać, u nas jest to `calc_sum`.

③ Przygotowanie argumentów wymaganych przez funkcję.

④ Przykład wywołania funkcji z wykorzystaniem metody `SimpleJdbcCall#executeFunction()` z argumentami w obiekcie typu `MapSqlParameterSource`.

NamedParameterJdbcTemplate (*org.springframework.jdbc.core.namedparam*)

NamedParameterJdbcTemplate działa w oparciu o *JdbcTemplate*. Wprowadza nazwy parametrów do zapytań SQL, zamiast standardowego znaku zapytania `"?"`. Jest to szczególnie przydane, kiedy mamy do czynienia ze złożonymi zapytaniami z wieloma parametrami. Co więcej, jak zawsze, lepsza czytelność przekłada się na jakość kodu.

Przykład użycia **NamedParameterJdbcTemplate***:*

```
@Service
@RequiredArgsConstructor
public class NamedParameterJdbcTemplateExamples {

    private final SimpleDriverDataSource simpleDriverDataSource;

    public void namedParameterJdbcTemplateExample() {
        NamedParameterJdbcTemplate namedParameterJdbcTemplate
            = new NamedParameterJdbcTemplate(simpleDriverDataSource); ①

        String sqlInsert = "INSERT INTO PERSON (NAME, AGE) VALUES (:name, :age)"; ②

        Map<String, Object> params1 = new HashMap<>();
```

```
        params1.put("name", "Karol");
        params1.put("age", 56);
        namedParameterJdbcTemplate.update(sqlInsert, params1); ③

        MapSqlParameterSource params2 = new MapSqlParameterSource()
            .addValue("name", "Karol")
            .addValue("age", 73);
        namedParameterJdbcTemplate.update(sqlInsert, params2); ④

        Person person = Person.builder()
            .name("Karol")
            .age(12)
            .build();
        BeanPropertySqlParameterSource params3 = new BeanPropertySqlParameterSource(person);
        namedParameterJdbcTemplate.update(sqlInsert, params3); ⑤
    }

}
```

① Utworzenie obiektu `NamedParameterJdbcTemplate`, który w konstruktorze przyjmuje wcześniej przygotowany obiekt `simpleDriverDataSource`.

② Definicja zapytania SQL, ale tym razem, dzięki zastosowaniu `NamedParameterJdbcTemplate`, zamiast znaków zapytania mamy placeholdery w postaci nazw parametrów potrzebnych do wykonania zapytania: *:name* oraz *:age*.

③ Przykład wykonania zapytania z użyciem mapy.

④ Przykład wykonania zapytania z użyciem obiektu `MapSqlParameterSource`.

⑤ Przykład wykonania zapytania z użyciem obiektu `BeanPropertySqlParameterSource`.