Notatki - JDBC i ResultSet

Spis treści

JDBC i ResultSet

ResultSet

Otrzymaliśmy już wcześniej `ResultSet`, ale ani razu jeszcze nie wykorzystaliśmy go do faktycznego odczytania wartości.

W kodzie poniżej staramy się pobrać wszystkich klientów naszego sklepu, którzy mają imię zawierające litery `me`. W tym celu konstruujemy zapytanie:

```
SELECT * FROM CUSTOMER WHERE NAME LIKE '%me%';
```

Pamiętając, że w przypadku wykorzystania klauzuli `LIKE` należy zwrócić uwagę na procenty i miejsca, w których je umieszczamy. Procent oznacza brak znaku, jeden lub więcej dowolnych znaków.

Wykonujemy takie zapytanie, pobieramy `ResultSet` i wykorzystując pętle `while`, staramy się pobrać wartości z kolejnych kolumn i na tej podstawie zwrócić obiekt `Customer`. Następnie obiekty te dodajemy do listy.

Wykorzystujemy metodę `next()` gdyż `ResultSet` posiada kursor, który określa obecną pozycję do odczytu danych w zbiorze danych będącym rezultatem zapytania. W momencie, gdy wywołujemy metodę `next()`, przesuwamy ten kursor o jedną pozycję do przodu. Gdy metoda `next()` zwróci `true`, oznacza to, że możemy odczytać dane. Gdy zwróci `false`, mówi to nam, że nie mamy w tym `ResultSet` już dalej danych. Dlatego można w prosty sposób wykorzystać tę metodę w pętli `while()`.

Klasa JdbcResultSetExample

```
public class JdbcResultSetExample {

    public static void main(String[] args) {
        String query = "SELECT * FROM CUSTOMER WHERE NAME LIKE ?;";
        String parameter = "%me%";

        String datbaseURL = "jdbc:postgresql://localhost:5432/zajavka";
        String user = "postgres";
        String password = "password";
        try (
            Connection connection = DriverManager.getConnection(datbaseURL, user, password);
            PreparedStatement statement = connection.prepareStatement(query)
        ) {
            statement.setString(1, parameter);
            try (
                ResultSet resultSet = statement.executeQuery();
            ) {
                List<Customer> customers = CustomerMapper.mapToCustomers(resultSet);
                customers.forEach(c -> System.out.println("Customer: " + c));
            }
        } catch (Exception e) {
            System.err.printf("Error while working on database: %s%n", e.getMessage());
        }
    }
}
```

Klasa CustomerMapper

```
public class CustomerMapper {

    static List<Customer> mapToCustomers(final ResultSet rs) {
        List<Customer> customers = new ArrayList<>();
        try {
            while (rs.next()) {
                long id = rs.getLong("id");
                String userName = rs.getString("user_name");
                String email = rs.getString("email");
                String name = rs.getString("name");
                String surname = rs.getString("surname");
                String dateOfBirth = rs.getString("date_of_birth");
                String telephoneNumber = rs.getString("telephone_number");
                customers.add(
                    new Customer(
                        id,
                        userName,
                        email,
                        name,
                        surname,
                        LocalDate.parse(dateOfBirth),
                        telephoneNumber));
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
        return customers;
    }
```

```
}
```

Zwróć uwagę na metody w stylu `getLong()`, `getString()` i nazwę kolumny z bazy danych. Metod tego typu jest o wiele więcej: `getInt()`, `getDouble()`, `getBoolean()` itp. Ważne jest tutaj, aby `String` podany jako argument tych metod był dokładnie taki jak nazwa kolumny. Wielkość liter nie ma natomiast znaczenia.

Oprócz metod odwołujących się po nazwie kolumny możemy również odwołać się na podstawie indeksów kolejnych kolumn. W takim przypadku metoda `mapToCustomers()` wyglądałaby w ten sposób:

```
public class CustomerMapper {

    static List<Customer> mapToCustomersByIndex(final ResultSet rs) {
        List<Customer> customers = new ArrayList<>();
        try {
            while (rs.next()) {
                long id = rs.getLong(1);
                String userName = rs.getString(2);
                String email = rs.getString(3);
                String name = rs.getString(4);
                String surname = rs.getString(5);
                String dateOfBirth = rs.getString(6);
                String telephoneNumber = rs.getString(7);
                customers.add(
                    new Customer(
                        id,
                        userName,
                        email,
                        name,
                        surname,
                        LocalDate.parse(dateOfBirth),
                        telephoneNumber));
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
        return customers;
    }
}
```

Pierwsza kwestia to zapamiętania przy stosowaniu metod odwołujących się po indeksach kolumn to to, że numeracja rozpoczyna się o `1`, a nie od `0`. Odwoływanie się po nazwach kolumn jest też czytelniejsze, bo określa w jawny sposób z perspektywy czytającego kod, która konkretnie kolumna nas interesuje.

Skoro poruszyliśmy już jak działa metoda `next()`, to jednocześnie widzimy, że nawet jeżeli `ResultSet` będzie zawierał przykładowo `1000` rezultatów, to możemy w taki sposób napisać kod, aby odczytać tylko jeden, albo tylko `10`. Kwestia tego jak napiszemy pętlę. W przypadku próby odczytu tylko pierwszego rekordu z `ResultSet`, możemy wykorzystać `if` zamiast pętli `while`.

Ważne też jest aby pamiętać, że należy sprawdzić, poprzez metodę `next()` czy kursor w `ResultSet` wskazuje obecnie na wiersz, który posiada dane. Jeżeli tego nie zrobimy i postaramy się wywołać metodę np. `rs.getString("id")`, rezultatem będzie `SQLException`, gdyż w `ResultSet` może nie być danych.

Co w przypadku dat i czasów?

Wspomniałem, że mamy dostępne metody `get···()`, które pozwalają nam na odczyt typów takich jak `Boolean`, `Byte`, `Short`, `Integer`, `Double`, `Float` oraz `String`. Co ciekawe, nie ma metody `getChar()`, taka ciekawostka. W tym przypadku będziemy używać metody `getString()`. Co natomiast w przypadku dat? Mamy dostępne następujące metody:

Metoda	Typ zwracany w Javie	Typ bazodanowy	Odpowiedni typ w Java 8
getDate()	java.sql.Date	DATE	java.time.LocalDate
getTime()	java.sql.Time	TIME	java.time.LocalTime
getTimestamp()	java.sql.Timestamp	TIMESTAMP	java.time.LocalDateTime

Jeżeli chcielibyśmy odczytywać daty z `ResultSet` to moglibyśmy to zrobić w następujący sposób:

```
public class DateMapper {

    static LocalDate getDate(final ResultSet rs) {
        try {
            if (rs.next()) {
                Date date = rs.getDate(1);
                return date.toLocalDate();
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
        return null;
    }
}
```

Zwróć uwagę, że mamy dostępną metodę `toLocalDate()` w klasie `java.sql.Date`. Analogicznie, będziemy mieli dostępne metody `toLocalTime()` oraz `toLocalDateTime()` w klasach `java.sql.Time` oraz `java.sql.Timestamp`.

getObject()

Dodam również, że mamy dostępną taką metodę jak `getObject()`, którą możemy wywołać na `ResultSet`. Wymaga ona od nas później ręcznego sprawdzenia typu, który się kryje w danej kolumnie. Pamiętasz, że w Javie wszystko (oprócz prymitywów) dziedziczy z klasy `Object`? Zatem wszystko może się kryć pod klasą `Object`. Dlatego kod korzystający z metody `getObject()` mógłby wyglądać w ten sposób:

```
public class ObjectMapper {

    static void mapObject(final ResultSet rs) {
        try {
            while (rs.next()) {
                Object id = rs.getObject("id");
                Object name = rs.getObject("name");
                if (id instanceof Integer) {
                    int idAsInt = (Integer) id;
                    System.out.println(idAsInt);
                }
                if (name instanceof String) {
                    String nameAsString = (String) name;
                    System.out.println(nameAsString);
                }
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
```

```
    }
}
```

Jak widzisz, stosowanie tej metody jest o tyle problematyczne, że musimy sami zdecydować na jaki typ dany obiekt ma zostać rzutowany. Aczkolwiek mogą być w praktyce przypadki, kiedy będzie to potrzebne.

Co daje nam wyjątek SQLException

Tak jak tytuł sekcji wskazuje, co takiego dodaje nam SQLException? Posiada on pewne metody, które pomogą nam namierzyć powód wystąpienia błędu. Aby to zobrazować wywołajmy taki fragment kodu:

```
public class JdbcSqlExceptionExample {

    public static void main(String[] args) {
        getConnection1();
        getConnection2();
    }

    private static Optional<Connection> getConnection1() {
        try {
            Optional<Connection> connection = Optional.of(DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/nonExistingDatabase",
                "wrongUsername",
                "wrongPassword"
            ));
            System.out.println(connection);
            return connection;
        } catch (SQLException e) {
            System.err.println("Failed to create connection: " + e.getMessage());
        }
        return Optional.empty();
    }

    private static Optional<Connection> getConnection2() {
        try {
            Optional<Connection> connection = Optional.of(DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/nonExistingDatabase",
                "wrongUsername",
                "wrongPassword"
            ));
            System.out.println(connection);
            return connection;
        } catch (SQLException e) {
            System.err.printf(
                "Failed to create connection, message: [%s], sqlState: [%s], errorCode: [%s]%n",
                e.getMessage(), e.getSQLState(), e.getErrorCode());
        }
        return Optional.empty();
    }
}
```

W przypadku złapania błędu Exception, na ekranie zostanie wydrukowana taka wiadomość:

```
Failed to create connection: FATAL: password authentication failed for user "wrongUsername"
```

Jeżeli natomiast złapiemy wyjątek SQLException, oprócz metody getMessage() mamy jeszcze dostępne metody getSQLState() oraz getErrorCode().Dzięki czemu możemy pobrać kody dające nam informacje o szczegółach błędu. Na podstawie tych kodów możemy poszukać dodatkowych informacji i dowiedzieć się czegoś więcej niż z samej wiadomości błędu.