Notatki - JDBC - cz.1

Spis treści

JDBC

To skoro już potrafimy poruszać się w świecie baz danych, to możemy zacząć rozmawiać o tym jak z taką bazą danych może się dogadać Java.

Zanim natomiast przejdziemy do konkretów, chciałbym opisać dosyć ciekawą kwestię, której jeszcze nie widzieliśmy w całym procesie nauki. Tak jak wspominałem, różnych rodzajów baz danych jest dużo. Java nie wie konkretnie jak do każdej z tych baz danych ma się podłączyć. To znaczy, nie ma w JRE zapisanego potężnego ifa, który wyglądałby jakoś tak:

```
if (naszaBazaDanych = "PostgreSQL") {
    polacz_sie_w_ten_sposob();
}
if (naszaBazaDanych = "MySQL") {
    polacz_sie_w_inny_sposob();
}
```

Określone są natomiast pewne interfejsy (jak już wiemy, będące kontraktem), mówiące, jakie zachowania "coś" ma implementować, aby móc podłączyć się do konkretnej bazy danych. Wiemy też już, że aby interfejs był w ogóle użyteczny to musimy dostarczyć konkretną implementację metod z interfejsu.

Aby dostarczyć takie konkretne implementacje interfejsów, na których będziemy operować należy dostarczyć sterownik (driver) do konkretnej implementacji bazy danych. Sterownik taki będzie dostarczony w postaci pliku `.jar` (skrót od **Java Archive**) i będzie on zawierał konkretne implementacje wymaganych interfejsów, aby móc podłączyć się do konkretnej bazy danych.

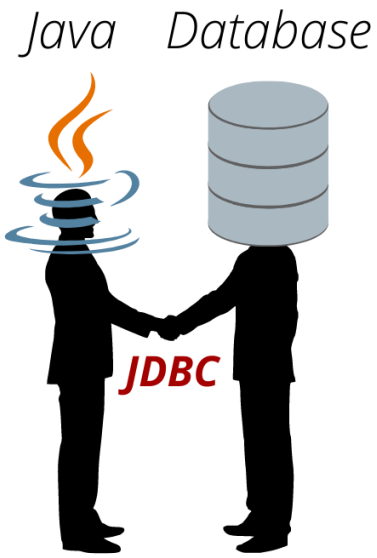
Możemy taki sterownik znaleźć w internecie, wpisując przykładowo **postgresql jdbc driver**. Przy pobieraniu Chrome nas zapyta, czy jesteśmy pewni, że chcemy pobrać ten plik bo może on być potencjalnie niebezpieczny. Prawda, może być on potencjalnie niebezpieczny, jeżeli nie znamy źródła pliku, jeżeli natomiast pobierzemy go ze strony Link to powinien on być bezpieczny ☺. Plik taki może się

nazywać przykładowo `postgresql-42.2.23.jar`. Najważniejsze jest to, że klasy, które są zawarte w pliku `.jar` wiedzą jak podłączyć się konkretnie do tej bazy danych, która nas interesuje.

Następnie mając taki plik `.jar` należy dodać go do naszego projektu i zaznaczyć, że jest on 'biblioteką'. Możemy to ustawić w ustawieniach naszego projektu **CTRL** + **ALT** + **SHIFT** + **S** w zakładce **libraries**.

Dzięki temu, możemy używać interfejsów w naszym projekcie podczas pisania kodu, natomiast aby faktycznie ten kod zadziałał, to musimy dostarczyć implementację naszych interfejsów. Dostarczamy ją dołączając sterownik dedykowany do konkretnej bazy danych do naszego projektu.

Przy następnych przykładach będziemy posługiwać się interfejsami `Driver`, `Connection`, `Statement` i `ResultSet`. Aby program mógł poprawnie działać należy dostarczyć sterownik, który zawiera implementację tych interfejsów. Nie interesuje nas natomiast jak konkretnie nazywają się klasy implementujące te interfejsy.



Obraz 1. JDBC

Podłączenie do bazy danych

Spróbujemy przejść przez przykłady w logicznej kolejności, to znaczy, że najpierw spróbujemy podłączyć się do bazy danych, aby później otrzymać wynik zapytania, które będziemy na takiej bazie wykonywać.

Zanim natomiast nastąpi połączenie do takiej bazy, musimy podać w jakiś sposób gdzie taka baza się znajduje, czyli podać jej adres.

JDBC URL

Jeżeli funkcjonujesz już trochę w Internecie to wydaje się naturalnym taki zapis https://www.youtube.com/. Jest to adres strony internetowej, na której możemy oglądać filmy o śmiesznych kotkach.

Bazy danych również mają swoje adresy i musimy podać taki adres, aby móc do takiej bazy danych się podłączyć. W przypadku baz danych, adres taki wygląda w ten sposób:

```
<protokol>:<rodzaj_bazy_danych>://<adres_bazy_danych>/<nazwa_bazy_danych>
```

Jeżeli teraz zmienne w nawiasach trójkątnych zastąpimy wartościami, to taki adres mógłby wyglądać w taki sposób:

```
jdbc:postgresql://localhost:5432/zajavka
```

Znaczenie kolejnych fragmentów:

- **jdbc** - nazwa protokołu jaki wykorzystujemy do połączenia
- **postgres** - nazwa bazy danych do jakiej chcemy się podłączyć
- **localhost** - nazwa hosta (inaczej strony internetowej), podając ten adres możemy znaleźć w sieci gdzie fizycznie znajduje się nasza baza danych. **Localhost** oznacza, że jest ona na naszej maszynie. Możemy ewentualnie spotkać zapis **127.0.0.1**, również oznacza on naszą maszynę
- **5432** - port, na którym baza danych jest dostępna. Możemy to sobie wyobrazić w taki sposób, że localhost (albo inny adres) jest miastem i aby wpłynąć do danego miasta statkiem potrzebujemy portu. Ale w jednym mieście może być wiele portów. Tak samo uruchamiając jakiś serwer na komputerze, jeżeli chcemy dać możliwość, aby taki serwer przyjmował jakikolwiek ruch to musi wystawiać otwarty port. Portów mamy wiele, bo gdyby serwer miał tylko jeden port, to na jednym komputerze moglibyśmy uruchomić tylko jedną aplikację. Jeżeli każda aplikacja/baza danych/inne rzeczy, których jeszcze nie poznaliśmy mogą być uruchamiane na różnych portach, to na jednym komputerze możemy uruchomić wiele aplikacji, które pozwalają się komunikować ze sobą przez sieć
- **zajavka** - nazwa bazy danych

Spróbujmy połączyć się do bazy danych

Teraz gdy już wiemy jak skonstruować adres, możemy spróbować faktycznie połączyć się do tej bazy danych.

```
public class JdbcConnectionExample {

    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection(
                    "jdbc:postgresql://localhost:5432/zajavka",
                    "postgres",
                    "password"
            );
            System.out.println(conn);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Jesteś już w stanie rozpoznać w powyższym kodzie JDBC URL, który musimy podać aby podłączyć się do

bazy danych. `postgres` to nazwa użytkownika bazy danych, natomiast `password` to przykładowe hasło dla tego użytkownika. Jeżeli nie udaje Ci się uzyskać połączenia, może to być kwestia niezgodności podanego adresu/użytkownika/hasła. Pamiętaj, że hasło było ustalane na etapie instalacji. Jeżeli coś nie działa na tym etapie, polecam googlować, musimy się uczyć rozwiązywać takie problemy ☺.

Gdy uruchomimy ten kod, na ekranie zostanie wydrukowane coś w tym stylu (no chyba, że wcześniej dodaliśmy sterownik do naszego projektu):

```
java.sql.SQLException: No suitable driver found for jdbc:postgresql://localhost:5432/zajavka
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:702)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:251)
    at pl.zajavka.JdbcConnectionExample.main(JdbcConnectionExample.java:10)
```

Dodajemy teraz sterownik do naszego projektu w oknie z ustawieniami naszego projektu **CTRL** + **ALT** + **SHIFT** + **S** w zakładce "libraries". Spróbujmy uruchomić teraz ten kod ponownie, na ekranie wydrukuje się coś podobnego do:

```
org.postgresql.jdbc.PgConnection@b59d31
```

Czyli konkretna klasa implementująca nasz interfejs `Connection`. W przypadku podłączenia do innej bazy danych, klasa ta nazywałaby się inaczej.

Trochę rzeczy stało się tutaj jakby magicznie. Wszystko dzięki metodzie `DriverManager.getConnection()`, która jest odpowiedzialna za znalezienie odpowiedniej klasy implementującej nasz interfejs (nie musimy tego podawać sami).

Od razu też uprzedzę, że to co zrobiliśmy tutaj zostało przedstawione w formie edukacyjnej. W praktyce prawdopodobnie nawet nie będziesz znać hasła do bazy danych na serwerze produkcyjnym, zostaną do tego wykorzystane inne mechanizmy.

Druga kwestia do wyjaśnienia - prawdopodobnie nie będziesz też ręcznie pisać/używać klas `Connection`, `Statement` itp. w praktyce. W praktyce stosowane są biblioteki, które robią to za Ciebie. Musimy natomiast wiedzieć jak to się zachowuje pod spodem, żeby nie była to dla nas jeszcze większa magia.

Statement

Następnym interfejsem, który powinniśmy omówić, jest `Statement`. Reprezentuje on `SQL Statement`, o których to SQLach uczyliśmy się wcześniej.

Najprostszym sposobem na uzyskanie `Statement` jest `connection.createStatement()`:

```
public class JdbcConnectionExample {

    public static void main(String[] args) {
        Optional<Connection> connection = getConnection();
        Optional<Statement> statement = connection.flatMap(JdbcConnectionExample::createStatement);
    }

    private static Optional<Connection> getConnection() {
        try {
```

```
            Optional<Connection> connection = Optional.of(DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/zajavka",
                "postgres",
                "password"
            ));
            System.out.println(connection);
            return connection;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return Optional.empty();
    }

    private static Optional<Statement> createStatement(final Connection connection) {
        try {
            return Optional.of(connection.createStatement());
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return Optional.empty();
    }
}
```

Są jednak dostępne przeładowane wersje metody `createStatement()`, które mogą przyjmować parametry. Nie chcę natomiast mocno się rozpisywać na ich temat, gdyż w większości przypadków będziesz stosować podstawowy wariant metody `createStatement()`.

Oba te parametry dotyczą `ResultSet`, czyli interfejsu, który będzie reprezentował wynik otrzymany po poprawnym wykonaniu zapytania na bazie danych. Parametry te określają, co możemy z takim `ResultSet` po jego otrzymaniu zrobić innego niż zachowanie domyślne.

ResultSet Type

Pierwszym parametrem, który możemy określić jest `ResultSet Type`. Określa on w jakim kierunku możemy czytać dane ze zbioru, który otrzymaliśmy. W większości przypadków będziemy czytać je dokładnie w takim, w jakim je otrzymaliśmy, bez żadnego kombinowania. Zapewni nam to parametr `ResultSet.TYPE_FORWARD_ONLY`, który jest wartością domyślną tego parametru w metodzie `createStatement()`. Jeżeli ktoś będzie potrzebował tutaj coś kombinować to odsyłam do dokumentacji ☺.

ResultSet Concurrency Mode

Drugi parametr jaki możemy przekazać do metody `createStatement()` dotyczy tego, czy z otrzymanego `ResultSet` możemy tylko czytać, czy może też próbować przez niego zapisywać dane do bazy danych. W większości przypadków z `ResultSet` będziemy tylko czytać i zapewnia nam to wartość domyślna tego parametru `ResultSet.CONCUR_READ_ONLY`, która jest używana w metodzie `createStatement()`. Jeżeli chcemy dokonać aktualizacji danych w bazie danych, raczej robi się to przy wykorzystaniu `UPDATE QUERY`, niż przez `ResultSet`. Jeżeli ktoś będzie potrzebował tutaj coś kombinować to odsyłam do dokumentacji ☺.

Zamykanie otwartych zasobów

Przejdźmy jeszcze do tematu zamykania otwartych połączeń/zasobów. Rzecz polega na tym, że na bazach danych ustawia się limity otwartych połączeń, więc jeżeli nie będziemy ich zamykać możemy

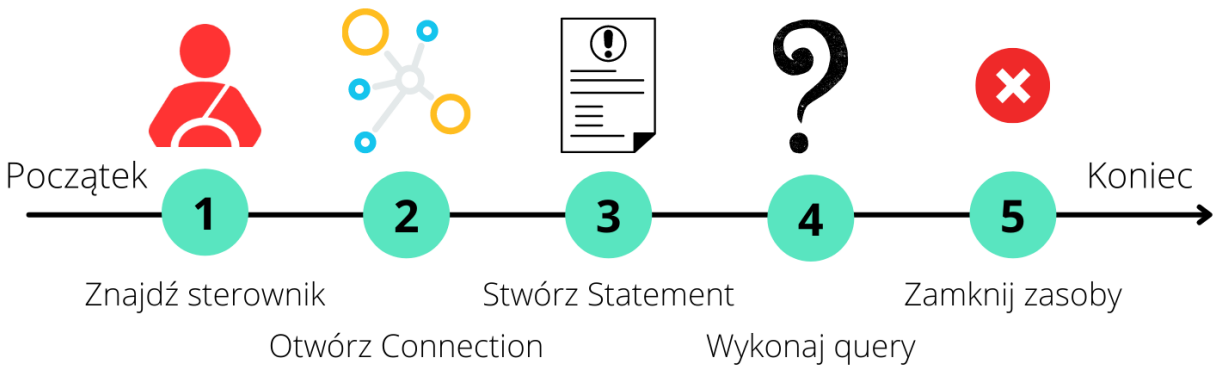
doprowadzić do takiej sytuacji, że nie będziemy w stanie podłączyć się do bazy danych, bo próbujemy ustanowić nowe połączenie, ale nie możemy tego zrobić bo nie zamknęliśmy starych i limit się wyczerpał. W praktyce stosowane są takie praktyki jak pule połączeń, aby móc reużywać otwarte już połączenia, ale nie jest to temat na teraz.

Interface `ResultSet` będzie omawiany w następnej kolejności. Tutaj jest on przywołany aby pokazać mechanizm otwierania i zamykania wszystkich zasobów, na których będziemy pracować.

Jak zamknąć połączenie?

```
private static void oldWay() throws SQLException {
    Connection connection = null;
    Statement statement = null;
    ResultSet resultSet = null;
    try {
        connection = DriverManager.getConnection(
            "jdbc:postgresql://localhost:5432/zajavka",
            "postgres",
            "password"
        );
        statement = connection.createStatement();
        resultSet = statement.executeQuery("SELECT * FROM CUSTOMER;");
        while (resultSet.next()) {
            System.out.println(resultSet.getString("name"));
        }
    } finally {
        try {
            if (resultSet != null)
                resultSet.close();
        } catch (SQLException e) {
            System.err.println("Failed to close ResultSet, " + e.getMessage());
        }
        try {
            if (statement != null)
                statement.close();
        } catch (SQLException e) {
            System.err.println("Failed to close Statement, " + e.getMessage());
        }
        try {
            if (connection != null)
                connection.close();
        } catch (SQLException e) {
            System.err.println("Failed to close Connection, " + e.getMessage());
        }
    }
}
```

Zwróć uwagę, że zamykamy nie tylko połączenie, ale również `ResultSet` i `Statement`. Generalnie jest dobrym nawykiem zamykać wszystkie 3 zasoby ręcznie (`Connection`, `Statement` i `ResultSet`), ale nie jest to konieczne. Teoretycznie jeżeli zamkniemy `Connection`, powinno ono zamknąć `Statement` i `ResultSet`. Jeżeli zamkniemy sam `Statement`, to powinien on zamknąć również `ResultSet`. Ważne jest natomiast aby zamykać te zasoby w odwrotnej kolejności niż zostały utworzone.



Obraz 2. Kroki połączenia JDBC

A da się prościej?

Pamiętasz `try-with-resources`? Kod pokazany powyżej da się mocno uprościć o ile zastosujemy konstrukcję `try-with-resources`.

```
private static void newWay() throws SQLException {
    try (Connection connection = DriverManager
            .getConnection("jdbc:postgresql://localhost:5432/zajavka", "postgres", "password");
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery("select name from animal")
    ) {
        while (resultSet.next()) {
            System.out.println(resultSet.getString("name"));
        }
    }
}
```

Konstrukcja `try-with-resources` zamykała zasoby w odwrotnej kolejności niż zostały one podane w zapisie. Czyli najpierw zostanie zamknięty `ResultSet`, potem `Statement`, a potem `Connection`. W przykładzie, w którym nie używaliśmy tej konstrukcji, trzeba było dodać samemu logowanie ewentualnych błędów, które mogły wystąpić na etapie zamykania każdego z zasobów. Tutaj jeżeli takie błędy wystąpią, będą one dostępne jako `Suppressed Exception`.