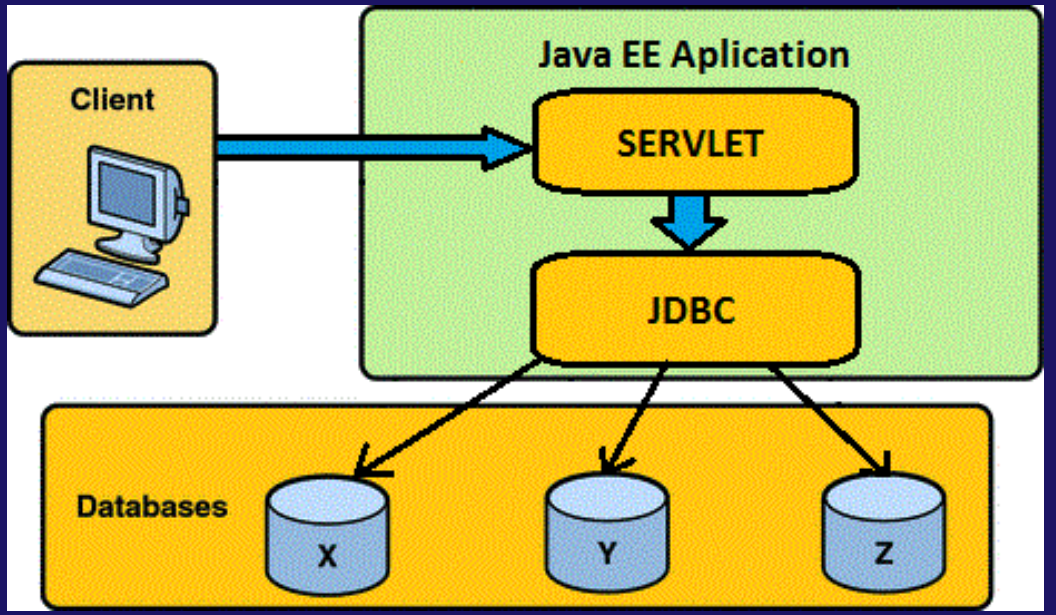
```
import java.sql.*;
```

JDBC

Java Database Connectivity

JDBC

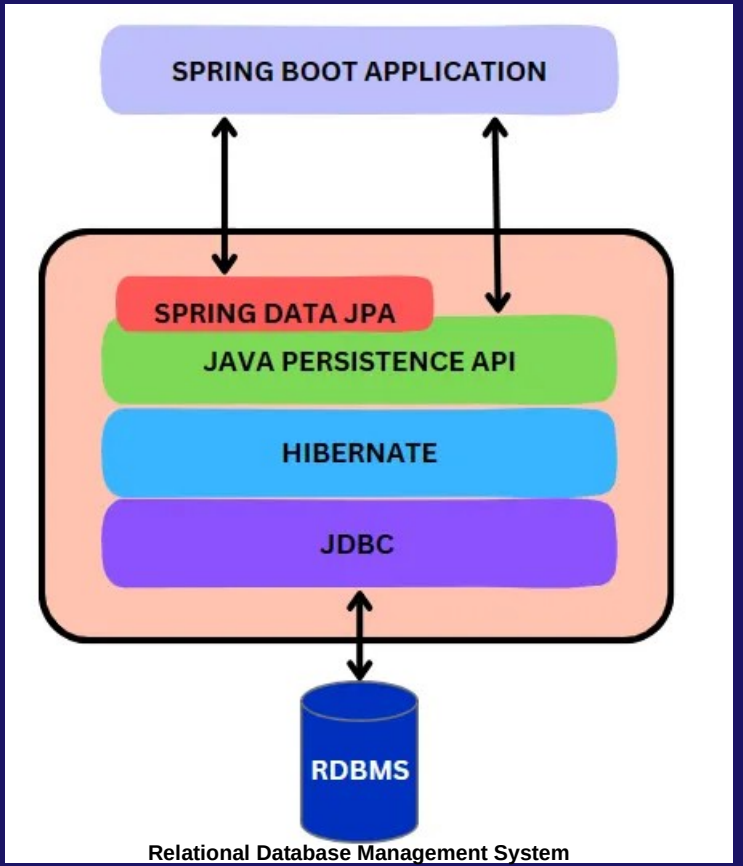
Java Database Connectivity jest to interfejs pozwalający na ustanowienie połączenia do relacyjnej bazy danych z poziomu aplikacji napisanej w Javie.



https://javastart.pl/baza-wiedzy/java-ee/jdbc-podstawy-pracy-z-baza-danych

JDBC

W aplikacjach pisanych w Spring Framework i Hibernate JDBC jest jedną z warstw.



https://medium.com/@aprayush20/spring-data-jpa-with-spring-boot-9e3b261e4cc7

Komponenty

JDBC

DriverManager

Klasa, która zarządza listą sterowników baz danych. Umożliwia aplikacji uzyskanie połączenia z bazą danych poprzez wywołanie metody getConnection.

Connection

Interfejs reprezentujący połączenie z bazą danych. Umożliwia wykonywanie poleceń SQL, zarządzanie transakcjami i uzyskiwanie metadanych bazy danych.

Statement

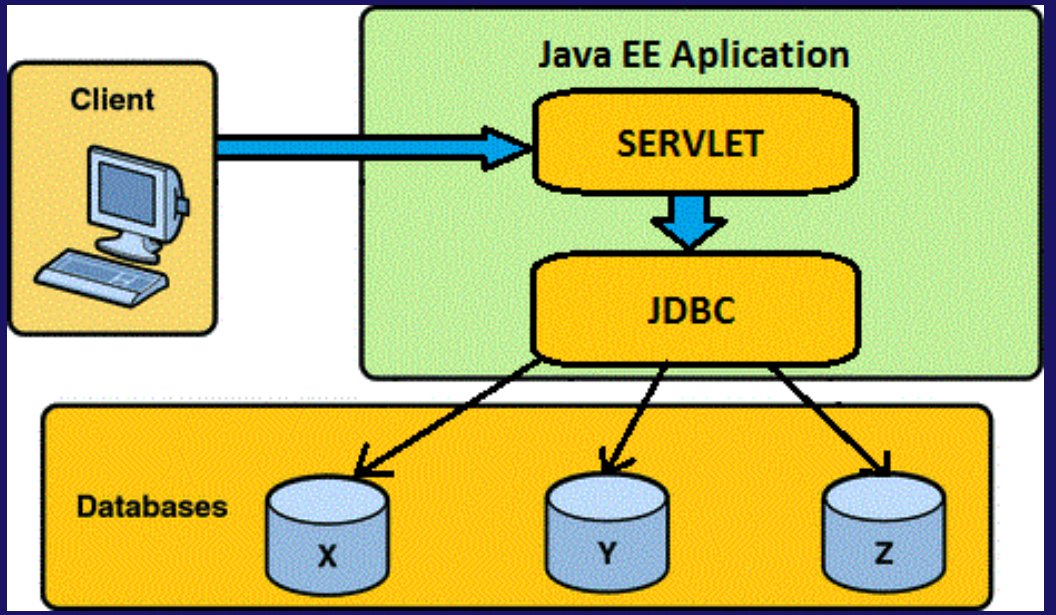
Interfejs służący do wykonywania zapytań SQL. Istnieją różne typy obiektów Statement, takie jak Statement, PreparedStatement i CallableStatement, które odpowiadają za wykonanie zapytań, przygotowane zapytania i wywołania procedur składowanych.

ResultSet

Interfejs reprezentujący zbiór wyników zapytania SQL. Umożliwia iterowanie po wynikach i odczytywanie wartości poszczególnych kolumn.

Baza danych

Przed prześledzeniem działania komponentów JDBC należy wybrać bazę i sterownik.



https://javastart.pl/baza-wiedzy/java-ee/jdbc-podstawy-pracy-z-baza-danych

Oto darmowy serwis:
https://neon.tech/

Baza danych

Konektory MySQL oraz PostgreSQL:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.3</version>
</dependency>
```

Baza danych

Przykładowa baza danych z zajęć wypełniona danymi:

Baza danych

```sql
CREATE TABLE users (
    id TEXT PRIMARY KEY,
    login TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    role TEXT NOT NULL
);
```

```sql
INSERT INTO users (id, login, password, role) VALUES
('fc6e600b-e89e-42e2-b2b7-ca18edb68924', 'lukasz',
'$2a$10$7MUYZR8FP8ebLzfqwMK85ejBke/TNp/Yme9NrC0XxRy.NAFde2vEm', 'ADMIN'),
('f4b5b79b-df2c-4e0a-89fc-464908d2e6dc', 'kamil',
'$2a$10$BwY3UtVLQTSUjX3zXjCeJ.h4Gwq4MiLhhTaUC.Yesce7rBDRkdU/i', 'USER');
```

Baza danych

zapis danych w postaci jsonb trwa dłużej, operacje wykonywane na typie jsonb trwają krócej, gdyż nie trzeba ich dodatkowo parsować.

```
CREATE TABLE vehicle (
    id TEXT PRIMARY KEY,
    category TEXT,
    brand TEXT,
    model TEXT,
    year INT,
    plate TEXT,
    price NUMERIC,
    attributes JSONB
);
```

```
INSERT INTO vehicle (id, category, brand, model, year, plate, price, attributes) VALUES
('1', 'Bus', 'Volkswagen', 'T2', 1985, 'LU123', 100.0, '{"seats": 20}'),
('2', 'Motorcycle', 'Honda', 'CBR600', 2016, 'LU456', 200.0, '{"licence_category": "A", "drive": "chain"}'),
('3', 'Car', 'Toyota', 'Corolla', 2024, 'LU789', 300.0, '{}');
```

Typ jsonb przechowywany w postaci binarnej.
Dostęp do operatorów, np.:
→>,czy ?

Baza danych

```sql
CREATE TABLE rental (
    id TEXT PRIMARY KEY,
    vehicle_id TEXT NOT NULL,
    user_id TEXT NOT NULL,
    rent_date TEXT NOT NULL,
    return_date TEXT,
    FOREIGN KEY (vehicle_id) REFERENCES vehicle(id),
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

```sql
INSERT INTO rental (id, vehicle_id, user_id, rent_date, return_date) VALUES
('884646be-d46a-4cfb-bda2-12c5ffff87f0', '2', 'f4b5b79b-df2c-4e0a-89fc-464908d2e6dc',
'2025-03-25T14:25:49.982037500', '2025-03-26T13:26:07.052319200'),
('3500e6c9-9878-41ac-9b65-ee5af8a5fc41', '1', 'fc6e600b-e89e-42e2-b2b7-ca18edb68924',
'2025-03-25T08:59:12.121505', '2025-03-26T07:59:20.468257700');
```

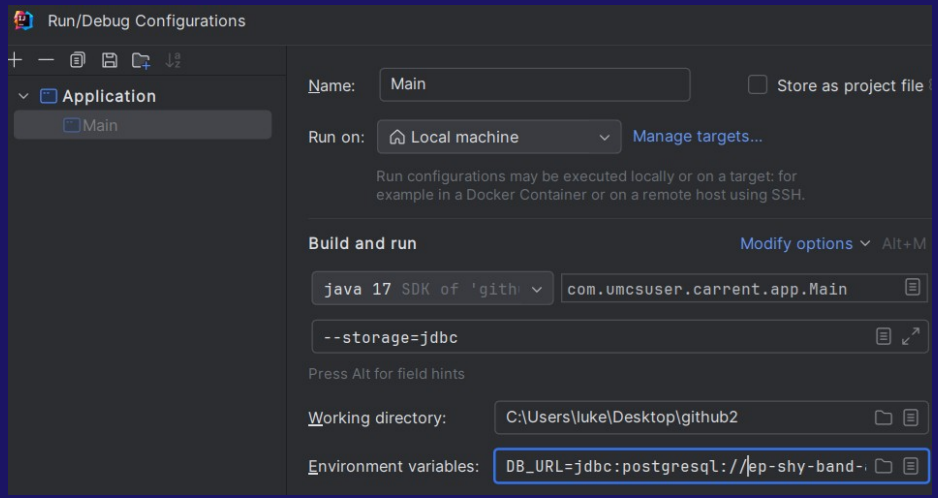
DriverManager

1. Rejestracja sterowników
2. Nawiązywanie połączeń
3. Zarządzanie sterownikami

Nawiązywanie połączenia wygląda bardzo prosto:

```
DriverManager.getConnection(url);
```

DriverManager



Połączenie może nam zwracać własna klasa singleton:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JdbcConnectionManager {
    private static JdbcConnectionManager instance;
    private final String url;

    public static JdbcConnectionManager getInstance() {
        if (instance == null) {
            instance = new JdbcConnectionManager();
        }
        return instance;
    }

    private JdbcConnectionManager() {
        url = System.getenv("DB_URL");
        if (url == null) {
            throw new RuntimeException("DATABASE_URL not set!");
        }
    }
    public Connection getConnection() {
        try {
            return DriverManager.getConnection(url);
        } catch (SQLException e) {
            throw new RuntimeException("Connection Failed!", e);
        }
    }
}
```

```
DriverManager.getConnection(url)
```

Connection

1.Wykonywanie Zapytań SQL:

Metody do tworzenia poleceń:
Statement,
PreparedStatement,
CallableStatement,
do wykonywania zapytań SQL, takich jak SELECT, INSERT, UPDATE, DELETE oraz wywoływanie procedur składowanych.

2.Zarządzanie Transakcjami

Manualnie:
connection.setAutoCommit(false)
Zatwiedzanie:
connection.commit()
Odrzucanie:
connection.rollback().

3.Zamykanie połączenia

connection.close().

PreparedStatement

Bezpieczny przed SQL Injection do głównego zapytania.

Parametry (?) ustawiane za pomocą metoda setInt(), setString() itp.

Wyższa wydajność, przy wielokrotnym użyciu zapytań w tym samym połączeniu.

```
public interface PreparedStatement
extends Statement
```

An object that represents a precompiled SQL statement.

A SQL statement is precompiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times.

Note: The setter methods (`setShort`, `setString`, and so on) for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type `INTEGER`, then the method `setInt` should be used.

If arbitrary parameter type conversions are required, the method `setObject` should be used with a target SQL type.

In the following example of setting a parameter, `con` represents an active connection:

```
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
                              SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00)
pstmt.setInt(2, 110592)
```

PreparedStatement

```java
@Override
public Optional<Vehicle> findById(String id) {
    String sql = "SELECT * FROM vehicle WHERE id = ?";
    try (Connection connection = JdbcConnectionManager.getInstance().getConnection();
         PreparedStatement stmt = connection.prepareStatement(sql)) {

        stmt.setString(1, id);
        try (ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                String attrJson = rs.getString("attributes");
                Map<String, Object> attributes = gson.fromJson(attrJson,
 new TypeToken<Map<String, Object>>(){}.getType());

                Vehicle vehicle = Vehicle.builder()
                        .id(rs.getString("id"))
                        .category(rs.getString("category"))
                        .brand(rs.getString("brand"))
                        .model(rs.getString("model"))
                        .year(rs.getInt("year"))
                        .plate(rs.getString("plate"))
                        .price(rs.getDouble("price"))
                        .attributes(attributes != null ? attributes : new HashMap<>())
                        .build();
                return Optional.of(vehicle);
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException("Error occurred while reading vehicle", e);
    }
    return Optional.empty();
}
```

PreparedStatement

```java
@Override
public List<Vehicle> findAll() {
    List<Vehicle> list = new ArrayList<>();
    String sql = "SELECT * FROM vehicle";
    try (Connection connection = JdbcConnectionManager.getInstance().getConnection();
         PreparedStatement stmt = connection.prepareStatement(sql);
         ResultSet rs = stmt.executeQuery()) {

        while (rs.next()) {
            String attrJson = rs.getString("attributes");
            Map<String, Object> attributes = gson.fromJson(attrJson,
 new TypeToken<Map<String, Object>>(){}.getType());

            Vehicle vehicle = Vehicle.builder()
                    .id(rs.getString("id"))
                    .category(rs.getString("category"))
                    .brand(rs.getString("brand"))
                    .model(rs.getString("model"))
                    .year(rs.getInt("year"))
                    .plate(rs.getString("plate"))
                    .price(rs.getDouble("price"))
                    .attributes(attributes != null ? attributes : new HashMap<>())
                    .build();
            list.add(vehicle);
        }
    } catch (SQLException e) {
        throw new RuntimeException("Error occurred while reading vehicles", e);
    }
    return list;
}
```

```java
@Override
public Vehicle save(Vehicle vehicle) {
    if (vehicle.getId() == null || vehicle.getId().isBlank()) {
        vehicle.setId(UUID.randomUUID().toString());
    } else {
        deleteById(vehicle.getId());
    }
```

PreparedStatement

```java
    String sql = "INSERT INTO vehicle (id, category, brand, model, year, plate, price, attributes) VALUES
(?, ?, ?, ?, ?, ?, ?, ?::jsonb)";
    try (Connection connection = JdbcConnectionManager.getInstance().getConnection();
         PreparedStatement stmt = connection.prepareStatement(sql)) {

        stmt.setString(1, vehicle.getId());
        stmt.setString(2, vehicle.getCategory());
        stmt.setString(3, vehicle.getBrand());
        stmt.setString(4, vehicle.getModel());
        stmt.setInt(5, vehicle.getYear());
        stmt.setString(6, vehicle.getPlate());
        stmt.setDouble(7, vehicle.getPrice());
        stmt.setString(8, gson.toJson(vehicle.getAttributes()));

        stmt.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException("Error occurred while saving vehicle", e);
    }
    return vehicle;
}
```

PreparedStatement

```java
@Override
public void deleteById(String id) {
    String sql = "DELETE FROM vehicle WHERE id = ?";
    try (Connection connection = JdbcConnectionManager.getInstance().getConnection();
         PreparedStatement stmt = connection.prepareStatement(sql)) {

        stmt.setString(1, id);
        stmt.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException("Error occurred while deleting vehicle", e);
    }
}
```

ResultSet

ResultSet jest obiektem, który umożliwia dostęp do wyników zapytań - tabela(wiersze i kolumny)

next() - przesuwa kursor do następnego wiersza.

Metody getter’y - getString(),getInt() zwracają wartości kolumn z aktualnego wiersza .

Wartości kolumn można pobierać po numerze indeksu, aliasie lub nazwie kolumny

A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A `ResultSet` object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row, and because it returns `false` when there are no more rows in the `ResultSet` object, it can be used in a `while` loop to iterate through the result set.

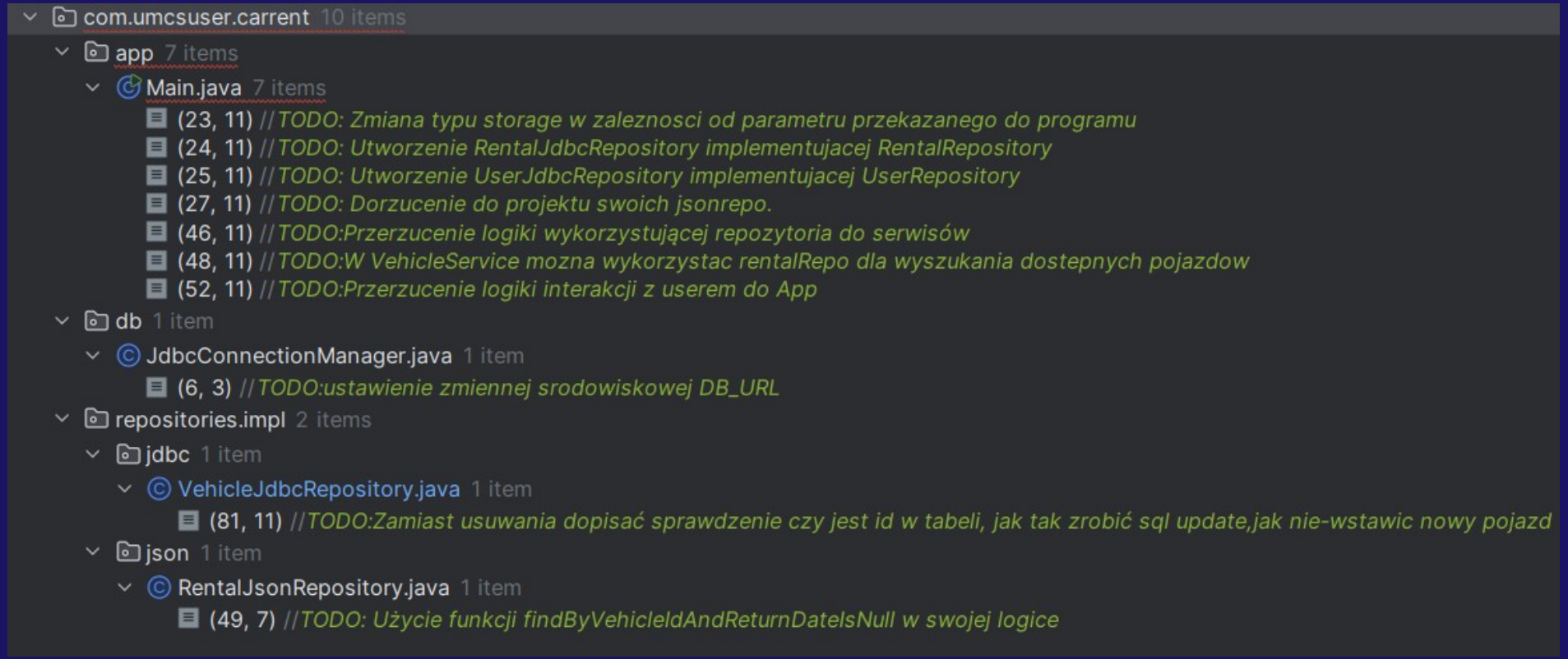
A default `ResultSet` object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable. See `ResultSet` fields for other options.

Zamykanie zasobow!!!!

Należy pamiętać o tym, że
ResultSet,
PreparedStatement
Connection trzeba zamknąć!!!

W naszym przykładzie robi się to automatycznie przez try-with-resources!!

Zadania


com.umcsuser.carrent 10 items
app 7 items
Main.java 7 items
(23, 11) //TODO: Zmiana typu storage w zaleznosci od parametru przekazanego do programu
(24, 11) //TODO: Utworzenie RentalJdbcRepository implementujacej RentalRepository
(25, 11) //TODO: Utworzenie UserJdbcRepository implementujacej UserRepository
(27, 11) //TODO: Dorzucenie do projektu swoich jsonrepo.
(46, 11) //TODO:Przerzucenie logiki wykorzystującej repozytoria do serwisów
(48, 11) //TODO:W VehicleService mozna wykorzystac rentalRepo dla wyszukania dostepnych pojazdow
(52, 11) //TODO:Przerzucenie logiki interakcji z userem do App
db 1 item
JdbcConnectionManager.java 1 item
(6, 3) //TODO:ustawienie zmiennej srodowiskowej DB_URL
repositories.impl 2 items
jdbc 1 item
VehicleJdbcRepository.java 1 item
(81, 11) //TODO:Zamiast usuwania dopisać sprawdzenie czy jest id w tabeli, jak tak zrobić sql update,jak nie-wstawic nowy pojazd
json 1 item
RentalJsonRepository.java 1 item
(49, 7) //TODO: Użycie funkcji findByVehicleIdAndReturnDateIsNull w swojej logice

Dziękuję za uwagę!