WYŻSZA SZKOŁA BANKOWA
Warszawa

Programowanie aplikacji w Java

Maciej Gowin

Zjazd 6 - dzień 1

Linki

Opis

https://maciejgowin.github.io/wsb-java/

Kod źródłowy przykładów oraz zadań

https://github.com/MaciejGowin/wsb-programowanie-aplikacji-java

Łączenie z bazą danych

Do tej pory łączyliśmy się z bazą danych MySQL za pomocą wbudowanego klienta `mysql`. Jest to przydatne do zarządzania bazą oraz danymi. Jednak z perspektywy programu - to on sam powinien być w stanie takie połączenie nawiązać, aby zarządzać danymi.

Połączenie z bazą danych będzie zastępowało operacje pisania oraz czytania z pliku, o których mówiliśmy poprzednio.

Łączenie z bazą danych

Java dostarcza szereg klas umożliwiających komunikacje z relacyjnymi bazami danych. Dostępne są one w pakiecie `java.sql`.

Przepływ danych będzie odbywał się podobnie jak w przypadku wbudowanego klienta:

- na początku programu będziemy tworzyli połączenie,
- po uzyskaniu połączenia będziemy wykonywali zapytania na bazie oraz odczytywali wyniki,
- w momencie zakończenia pracy będziemy zamykali połączenie.

Jak widać, przepływ ten jest już nam znany z operacji na plikach oraz operacji wejścia/wyjścia.

Programowanie: przykład 52

```
package pl.wsb.programowaniejava.maciejgowin.przyklad52;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Main {

    public static void main(String[] args) {
        try {
            // Class.forName("com.mysql.jdbc.Driver");
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            System.out.printf("Failed to load connector: %s%n", e.getMessage());
        }

        try (Connection connection = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/booking_system", "root", "root")) {

            try (Statement statement = connection.createStatement()) {
                ResultSet resultSet = statement.executeQuery("SELECT * FROM airports");
                while (resultSet.next()) {
                    System.out.printf("Airport: [%s, %s, %s, %s]%n",
                            resultSet.getString(1),
                            resultSet.getString(2),
                            resultSet.getString(3),
                            resultSet.getString(4));
                }
            }
        } catch(Exception e) {
            System.out.printf("Failed to load: %s%n", e.getMessage());
        }
    }
}
```

Język Java: połączenie z bazą danych

Łączenie z bazą danych odbywa się przy pomocy łańcucha połączenia (ang. connection string). Składa się on z kilku elementów opisujących sposób połączenia oraz lokalizację bazy.

```
jdbc:mysql://localhost:3306/booking_system
```

`jdbc` - nazwa interfejsu, przy którego udziale będziemy komunikować się z bazą.
`mysql` - nazwa sterownika bazy danych, który będzie realizował połączenie.
`localhost` - adres bazy danych (nazwa lub IP).
`3306` - port, na którym dostępne jest połączenie do bazy danych.
`booking_system` - nazwa bazy danych.

JDBC

Java Database Connectivity (JDBC) jest interfejsem aplikacyjnym (API) dla języka Java, który definiuje sposób dostępu do bazy danych przez klienta.

Rozwiązanie to jest częścią standardowej dystrybucji Javy. Pozwala na wykonywanie zapytań obsługujących odczytywanie oraz uaktualnianie danych.

Rozwiązanie JDBC korzysta ze sterowników JDBC, które są implementacją JDBC API. Sterowniki są dostarczane dla danego typu bazy danych. Dla przykładu, aby połączyć się z bazą MySQL, do działającego programu należy dołączyć sterownik MySQL, który realizuje JDBC API.

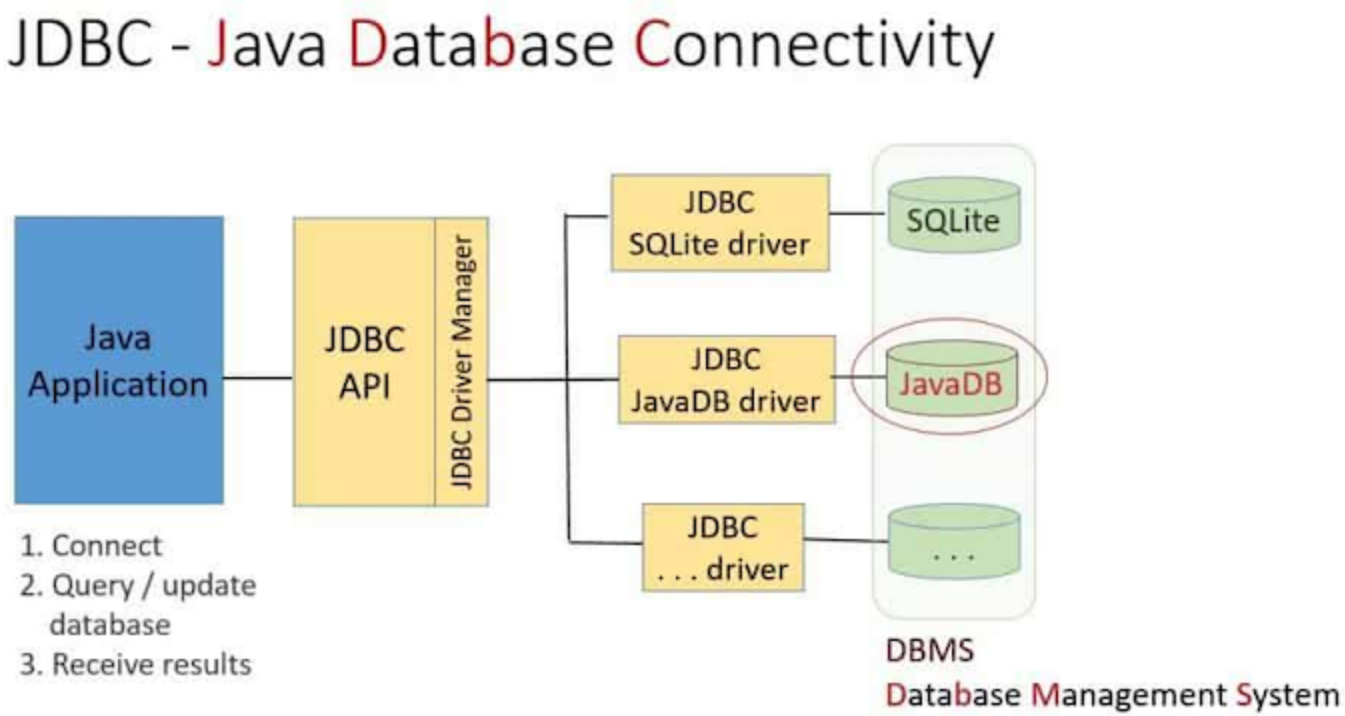
Na samo JDBC możemy patrzeć jak na fasadę. Izolacja pozwala na spójny sposób wykorzystywania JDBC oraz zmieniania sterowników w miarę potrzeb.

JDBC

Istnieje wiele typów sterowników JDBC, jak i poziomów komunikacji. Będziemy się koncentrować na sterownikach, które łączą się z bazą danych poprzez konwersję wywołań JDBC na zapytania specyficzne dla bazy danych.

Sterowniki te są niezależne od platformy. Jego przykładem jest sterownik do obsługi MySQL.

JDBC



Źródło: https://networkencyclopedia.com/wp-content/uploads/2019/08/java-database-connectivity-jdbc-1024x576.jpg

Język Java: ładowanie sterowników

Sterowniki ładowane są dynamicznie przez mechanizm refleksji. Implementacja sterownika musi być dostarczona w czasie jego użycia i dostępna w ścieżce przeszukiwań (ang. classpath).

W naszym przykładzie używaliśmy sterownika dla bazy MySQL, którego dostarczyliśmy podczas uruchomienia.

Język Java: ładowanie sterowników

Sterownik został załadowany na początku programu:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Taka inicjalizacja sterownika jest znana ze starszych wersji JDBC. W nowoczesnych środowiskach, zaczynając od JDBC 4.0, wszystkie sterowniki, które zostaną odnalezione w ścieżce wyszukiwań (classpath), są automatycznie ładowane.

> Sterownik nie jest potrzebny na poziomie kompilacji, ponieważ nie odwołujemy się do niego bezpośrednio. Wszystkie wywołania odnoszą się do JDBC API.

Język Java: połączenie z bazą danych

W najprostszym przykładzie do uzyskania połączenia będziemy używać klasy `DriverManager`. Jest ona odpowiedzialna za nawiązanie połączenia.

```
Connection connection = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/booking_system", "root", "root")
```

Do nawiązania połączenia konieczny jest jego definicja oraz użytkownik wraz z hasłem.

Język Java: operacje na połączeniu

Do wykonania zapytania na bazie danych używamy klasy `Statement`, która pełni rolę interfejsu do komunikacji z bazą:

```
Statement statement = connection.createStatement();
```

Po stworzeniu instancji możemy wykonać konkretne zapytanie, którego rezultat prezentowany będzie przez klasę `ResultSet`:

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM airports")
```

Język Java: klasa Statement

Warto zaważyć, że `ResultSet` może zwrócić zbiór elementów (z zależności od zapytania).

Operacje na nim traktujemy w sposób znany z `Iteratora` przechodząc po kolejnych elementach przy użyciu metody `next()` aż do momentu napotkania wartości `false`.

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM airports");
while(resultSet.next()) {
    // (...)
}
```

Język Java: klasa Statement

Domyślnie w danym momencie tylko jeden obiekt typu `ResultSet` może być otwarty dla konkretnego `Statement`. Dla kolejnych zapytań będziemy tworzyli nowe instancje `Statement` lub też używali operacji na pakietach (ang. batch).

Język Java: zapytania na Statement

Klasa `Statement` dostarcza szereg metod pozwalających na wykonanie zapytania na bazie.

execute()

Używana głównie do wykonywanie operacji typu DDL (ang. data definition language) takich jak CREATE, ALTER czy DROP. Zwraca `true`, jeżeli dodatkowo został zwrócony `ResultSet`, który może zostać pobrany za pomocą `getResultSet` lub `false` w przeciwnym wypadku.

Język Java: zapytania na Statement

Klasa `Statement` dostarcza szereg metod pozwalających na wykonanie zapytania na bazie.

executeUpdate()

Używana w przypadku zapytań modyfikujących dane, takich jak INSERT, UPDATE czy DELETE. Zwraca `int` reprezentujący ilość wierszy, które zostały zmienione. Zapytanie nie może zwracać rezultatów w postacie `ResultSet`.

Język Java: zapytania na Statement

Klasa `Statement` dostarcza szereg metod pozwalających na wykonanie zapytania na bazie.

executeQuery()

Używana w przypadku zapytań wyszukujących dane, takich jak SELECT oraz zwracających tabularyczne rezultaty. Zwraca `ResultSet` dostarczający wyszukane dane. Zapytanie musi zwracać rezultaty w postacie `ResultSet`.

Programowanie: przykład 53

Użycie metod `Statement`.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad53;

import java.sql.Connection;

import static pl.wsb.programowaniejava.maciejgowin.przyklad53.Configuration.createSchema;
import static pl.wsb.programowaniejava.maciejgowin.przyklad53.Configuration.getConnection;

public class Main {

    public static void main(String[] args) {
        try (Connection connection = getConnection(); AirportManager airportManager = new AirportManager(connection)) {

            createSchema(connection);

            // List airports
            System.out.printf("Check (1): %s%n", airportManager.getAirports());

            // Delete airports
            airportManager.deleteAirports();

            // List airports
            System.out.printf("Check (2): %s%n", airportManager.getAirports());

            // Add airports
            airportManager.addAirport(new Airport("WRO", "Wroclaw", 1.1, 11.11));
            airportManager.addAirport(new Airport("DUB", "Dublin", 2.2, 22.22));
            airportManager.addAirport(new Airport("PRG", "Prague", 3.3, 33.33));

            // List airports
            System.out.printf("Check (3): %s%n", airportManager.getAirports());

            // Get airport by code
            System.out.printf("Check (4): %s%n", airportManager.getAirport("WRO"));
            System.out.printf("Check (5): %s%n", airportManager.getAirport("BCN"));

            // Update airport
            airportManager.updateAirport(new Airport("WRO", "Wroclaw Miasto Spotkan", 1.1, 11.11));

            // List airports
            System.out.printf("Check (6): %s%n", airportManager.getAirports());
        } catch(Exception e) {
            System.out.printf("Failed: %s%n", e.getMessage());
        }
    }
}
```

Maven: resources

Maven pozwala na automatyczne dołączenie dowolnych plików do wynikowego archiwum (np. pliku JAR). Domyślnie wszystkie pliki umieszczone w katalogu `src/main/resources` zostają dodane do pliku wynikowego. Zachowana jest struktura katalogów, w których znajdują się pliki.

Częstą praktyką jest dostarczanie plików konfiguracyjnych aplikacji poprzez mechanizm zasobów.

Maven: resources

Oczywiście istnieje możliwość zmiany domyślnej lokalizacji, w której Maven będzie dodawał zasoby do naszego archiwum (np. pliku JAR).

```
<project>
  <!-- ... -->
  <build>
    <!-- ... -->
    <resources>
      <resource>
        <directory>[RESOURCES-DIRECTORY></directory>
      </resource>
    </resources>
  <!-- ... -->
  </build>
  <!-- ... -->
</project>
```

Gdzie RESOURCES-DIRECTORY to np.: `src/my-new-resources`

Język Java: ładowanie zasobów

Java pozwala nam na załadowanie dowolnego pliku z domyślnej ścieżki przeszukiwań. Podobnie jest w przypadku wyszukiwania plików z archiwum JAR, które te pliki zawiera.

Pliki określane są mianem zasobów (ang. resources).

Jednym ze sposobów pogrania zawartości pliku jest odwołanie się do jego zawartości poprzez strumień.

```
InputStream input = Main.class.getClassLoader().getResourceAsStream("myFile.txt");
```

Język Java: klasa PreparedStatement

Jak widać w przykładach, w przypadku zapytań, istotna jest ich parametryzacja. Do tej pory parametryzowaliśmy zapytania za pomocą formatowanych ciągów znaków. Java wychodzi nam naprzeciw i pozwala na parametryzowanie poprzez `PreparedStatement`.

```
public void update(Customer customer) throws SQLException {
    try (PreparedStatement preparedStatement = connection.prepareStatement("UPDATE customers SET name = ? WHERE id = ?")) {
        preparedStatement.setString(1, customer.getName());
        preparedStatement.setInt(2, customer.getId());
        preparedStatement.executeUpdate();
    }
}
```

Każdy z parametrów zostaje zdefiniowany znakiem `?`. Wartości przyjmują numery sekwencyjne (licząc od 1).

Programowanie: zadanie 26

Dla przykładu 53 przepisz operacje dodawania, uaktualniania oraz wyszukiwania lotnisk po kodzie za pomocą `PreparedStatement`.

Język Java: wartość AUTO_INCREMENT

Do tej pory, dodawaliśmy wiersze, których klucz główny był znany. W przypadku wierszy z kluczem głównym ustawionym na `AUTO_INCREMENT` może on zostać automatycznie wygenerowany.

Po dodaniu elementu chcielibyśmy poznać wartość wygenerowanego klucza głównego.

Programowanie: przykład 54

Pobieranie automatycznie wygenerowanego klucza głównego.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad54;

import java.sql.Connection;
import java.time.LocalDate;
import java.util.Optional;

import static pl.wsb.programowaniejava.maciejgowin.przyklad54.Configuration.createSchema;
import static pl.wsb.programowaniejava.maciejgowin.przyklad54.Configuration.getConnection;

public class Main {

    public static void main(String[] args) {
        try (Connection connection = getConnection()) {

            createSchema(connection);

            CustomerManager customerManager = new CustomerManager(connection);

            // Delete customers
            customerManager.deleteCustomers();

            // List customers
            System.out.printf("Check (1): %s%n", customerManager.getCustomers());

            // Add customer
            Optional<Integer> newId = customerManager.addCustomer("Maciej", "Gowin", LocalDate.parse("1986-11-21"));

            System.out.printf("Check (2): %s%n", newId);

            // List customers
            System.out.printf("Check (3): %s%n", customerManager.getCustomers());
        } catch(Exception e) {
            System.out.printf("Failed: %s%n", e.getMessage());
        }
    }
}
```

Programowanie: zadanie 27

Stwórz aplikację pozwalającą na dodawanie tytułów książek oraz wyszukiwanie książek po tytule.

SQL a programowanie obiektowe

Używanie niskopoziomowego dostępu do bazy danych przy użyciu pakietu `java.sql` może być uciążliwe. Wiąże się ono z ciągłym mapowaniem struktury tabel bazy danych na strukturę klas.

Pojawia się wiele miejsc, w których wykonujemy powtarzalne czynności, co jest zaprzeczeniem podejścia DRY (ang. don't repeat yourself).

Automatyzacja procesu mapowania pozwoliłaby na zaoszczędzenie czasu oraz ilości niepotrzebnego kodu, który potrzebny jest do wykonania tych operacji explicite.

Mapowanie obiektowo-relacyjne (ORM)

Problem odwzorowania struktury obiektowej na strukturę relacyjną jest opisany przez mapowanie obiektowo-relacyjne (ang. object-relational mapping, ORM).

Przykładem jest wspomniana już konwersja podejścia obiektowego użytego w kodzie Java na strukturę relacyjną użytą w bazie MySQL.

Java Persistence API

Java Persistence API (JPA) jest oficjalnym standardem opisującym mapowanie obiektowo-relacyjny dla języka Java.

JPA nie jest implementacją, jest jedynie standardem. Wprowadzą ono kolejny poziom abstrakcji w porównaniu do JDBC.

Java Persistence API

Składa się z 3 podstawowych elementów:

- samego API
- języka zapytań JPQL (The Jakarta Persistence Query Language, kiedyś Java Persistence Query Language)
- opisu mapowania obiektowo-relacyjnego

Java Persistence API

Historycznie standard JPA jest elementem standardu EJB 3.0, a zatem również Java Enterprise Edition (J2EE). Najczęściej środowisko to związane jest z sewerami aplikacyjnymi pozwalającymi na tworzenie aplikacji webowych przy użyciu języka Java.

Implementacje tego standardu nie ograniczają się tylko do serwerów aplikacyjnych i mogą byż z powodzeniem używane w aplikacjach Java Standard Edition (Java SE).

Java Persistence API/Jakarta Persistence

Istnieje szereg wersji, które prowadzają nowe funkcjonalności. Ostatnio standard ten został przemianowany na Jakarta Persistence.

Co za tym idzie, zmienione zostały również pakiety, pod którym zdefiniowany jest standard z `javax.persistance` na `jakarta.persistence`.

Główne wersje:

- JPA 2.0 (2009)
- JPA 2.1 (2013)
- JPA 2.2 (2017)
- Jakarta Persistence 3.0 (2020)

W dalszej części będziemy dalej używać skrótu JPA w odniesieniu do Java Persistence API/Jakarta Persistence.

Java Persistence API/Jakarta Persistence

Należy podkreślić, że JPA jest jedynie specyfikacją. Istnieje szereg implementacji tego standardu, do których należą:

- Hibernate (od wersji 5.5)
- EclipseLink (od wersji 3.0)
- DataNucleus (od wersji 6.0)

W dalszej części przyjrzymy się i będziemy korzystać z implementacji Hibernate.

Hibernate

Hibernate jest narzędziem dostarczającym funkcjonalność ORM dla języka Java. Jego głównym zadaniem jest łatwe mapowanie obiektów klas na wiersze tabel baz relacyjnych oraz funkcjonalności pozwalających na łatwe operacje na danych.

U swoich podstaw Hibernate był narzędziem ORM, dopiero od wersji 3.2 wprowadzona została implementacja dla JPA. Istnieje wiele spekulacji dotyczących tego, czy standard JPA nie powstał na podstawie implementacji Hibernate.

Hibernate może być używany w aplikacjach Java SE oraz J2EE poprzez bezpośrednie użycie natywnych klas Hibernate lub też implementacji JPA.

Hibernate - zalety

Do głównych zalet Hibernate należą:

- zarządzanie mapowaniem poprzez konfigurację XML bez konieczności pisania kodu.
- prosty interfejs dostępu do danych
- łatwe w zarządzaniu zmiany modelu przy pomocy plików konfiguracyjnych
- abstrakcja dla SQL
- dostęp do bazy poprzez zoptymalizowane strategie
- prosty sposób zapytań o dane

Hibernate - wsparcie dla baz

Hibernate wspiera większość ze znanych relacyjnych baz danych, w tym:

- MySQL
- PostgreSQL
- Oracle
- Microsoft SQL Server
- DB2/NT

Hibernate w praktyce: klasy a tabele

W najprostszym przykładzie jedna klasa odpowiada dokładnie jednej tabeli. Do tej pory konwersja danych odbywała się programistycznie - przy każdym odwołaniu do zbioru danych.

W Hibernate mechanizm ten jest zautomatyzowany. Wymagana jest jedynie konfiguracja zależności i sposobu konwersji danych - sam proces odbywa się już automatycznie.

Hibernate w praktyce: klasy a tabele

```
public class Airport {
    private String code;
    private String name;
    private double latitude;
    private double longitude;
}
```

```
CREATE TABLE airports
(
    code VARCHAR(3) NOT NULL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    latitude DOUBLE NOT NULL,
    longitude DOUBLE NOT NULL
);
```

Programowanie: przykład 55

Konfiguracja projektu i mapowań przy użyciu Hibernate.

```
package pl.wsb.programowaniejava.maciejgowin.przyklad55;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class Main {

    public static void main(String[] args) {
        try (SessionFactory factory = new Configuration().configure().buildSessionFactory();
             AirportManager airportManager = new AirportManager(factory)) {

            // List airports
            System.out.printf("Check (1): %s%n", airportManager.getAirports());

            // Delete airports
            airportManager.deleteAirports();

            // List airports
            System.out.printf("Check (2): %s%n", airportManager.getAirports());

            // Add airports
            airportManager.addAirport(new Airport("WRO", "Wroclaw", 1.1, 11.11));
            airportManager.addAirport(new Airport("DUB", "Dublin", 2.2, 22.22));
            airportManager.addAirport(new Airport("PRG", "Prague", 3.3, 33.33));

            // List airports
            System.out.printf("Check (3): %s%n", airportManager.getAirports());

            // Get airport by code
            System.out.printf("Check (4): %s%n", airportManager.getAirport("WRO"));
            System.out.printf("Check (5): %s%n", airportManager.getAirport("BCN"));

            // Update airport
            airportManager.updateAirport(new Airport("WRO", "Wroclaw Miasto Spotkan", 1.1, 11.11));

            // List airports
            System.out.printf("Check (6): %s%n", airportManager.getAirports());
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```