Wyższa Szkoła Bankowa
Gdańsk Gdynia

Analiza danych w języku Java cz. II

Praca z relacyjnymi bazami danych

Maciej Radzikowski <maciej@radzikowski.com.pl>

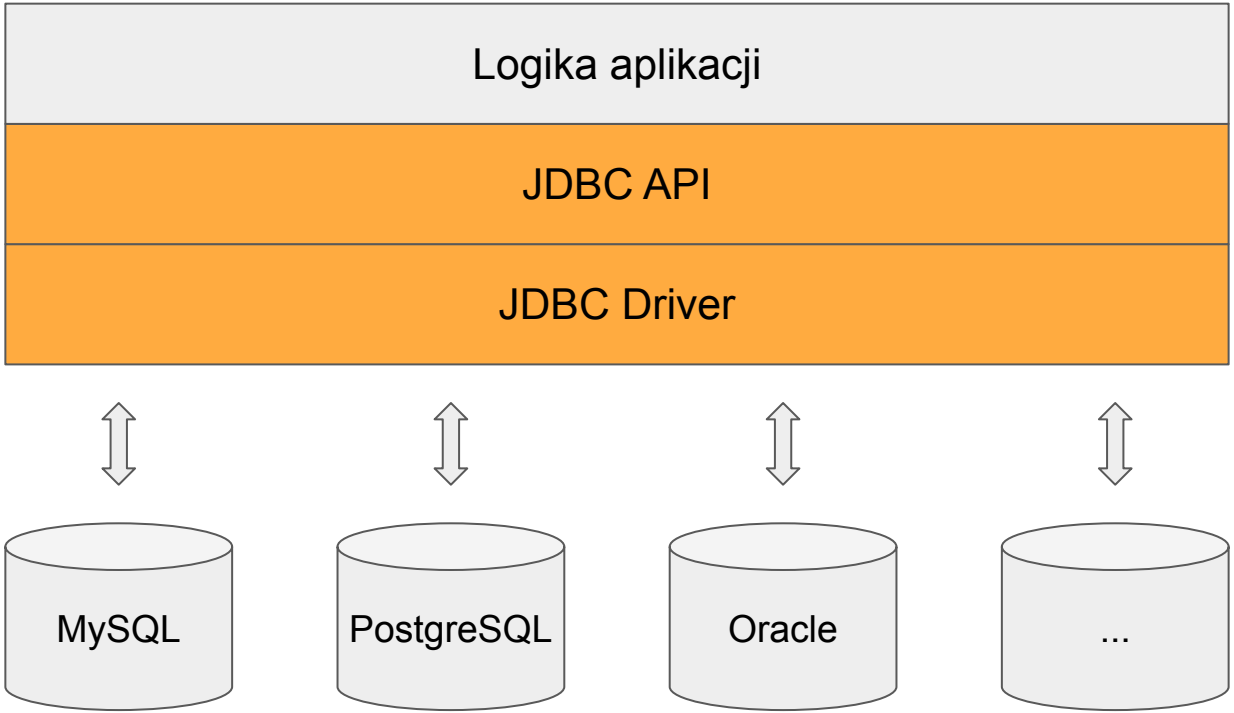
Połączenie z bazą danych

Praca z relacyjnymi bazami danych odbywa się przy użyciu standardu JDBC (Java Database Connectivity).

Połączenie wymaga biblioteki-sterownika (JDBC Driver), umożliwiającej komunikację z bazą danych za pomocą standardowego API.

Biblioteki te są osobne dla każdej z baz danych (MySQL, Oracle, PostgreSQL, ...).

Architektura JDBC



Dodawanie bibliotek

Zewnętrzne biblioteki, jak np. Driver JDBC dla danej bazy,
można dodać na dwa sposoby.

- dodanie paczki .jar w ścieżce ładowanej przez Javę
- pobranie poprzez managera zależności, np. Maven (plik pom.xml)

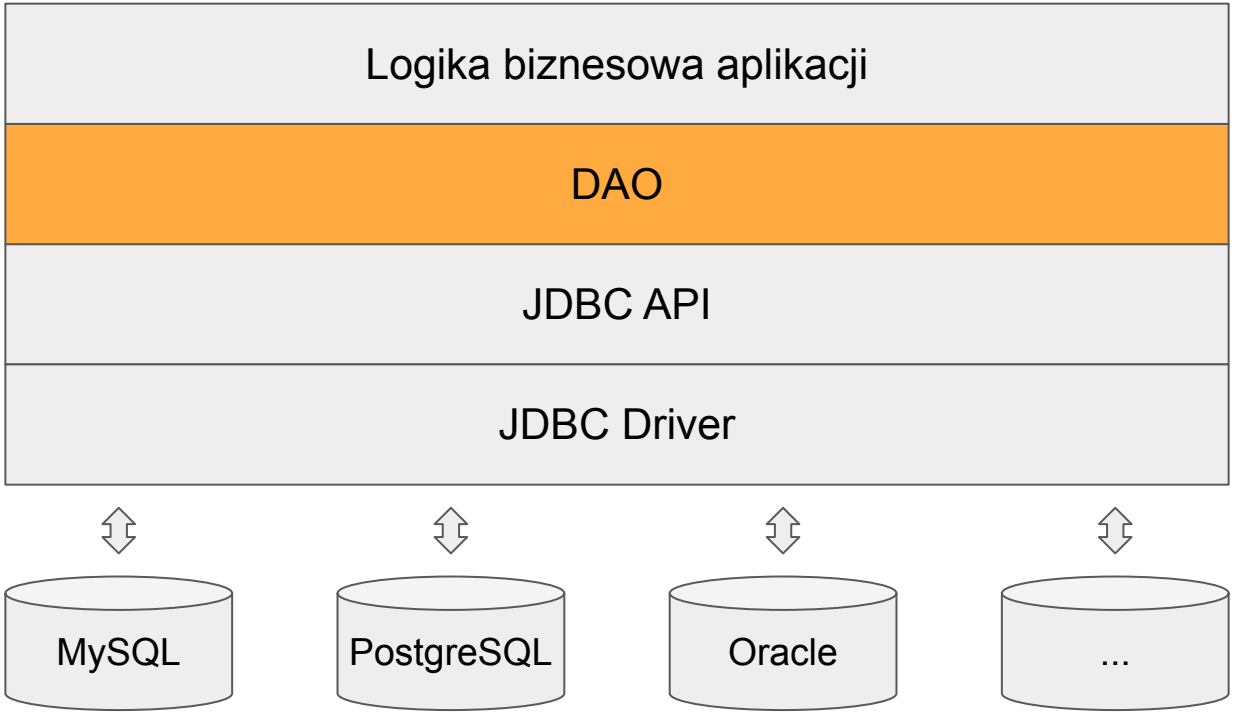
Użycie managera zależności jest znacznie lepszym podejściem.

Data Access Object Pattern

DAO to wzorzec architektoniczny, w ramach którego komunikacja między aplikacją a bazą danych wydzielona jest osobnej warstwy abstrakcji.

Obiekty DAO zapewniają metody do wykonywania operacji na bazie danych bez konieczności znajomości szczegółów implementacyjnych komunikacji z bazą danych przez inne fragmenty aplikacji.

Warstwa Data Access Object



Praca z bazami danych

Podejście niskopoziomowe - JDBC

Komunikacja z bazą danych przez “czyste” zapytania SQL, wykorzystując bezpośrednio JDBC, wyniki zwracane jako tabela danych.

Podejście wysokopoziomowe - JPA i ORM

Biblioteki typu ORM (Object-Relational Mapping) pośredniczące w komunikacji, automatycznie mapujące rekordy na obiekty i udostępniające metody do pracy na tych obiektach w kontekście ich utrwalania i wyszukiwania w bazie danych. Ich generalne API w Javie określa JPA.

JDBC

Cechy podejścia niskopoziomowego (JDBC)

- pełna kontrola nad wykonywanymi zapytaniami SQL
- możliwość używania niestandardowych (specyficznych dla danej bazy) poleceń SQL
- mało “obiektowe” podejście, należy samodzielnie napisać kod pobierający poszczególne wartości z wyników zapytania i tworzący obiekty, jeśli tego potrzebujemy

Otwieranie połączenia z bazą danych

Połączenie reprezentuje obiekt `Connection`.

Do stworzenia połączenia służy metoda `DriverManager.getConnection()`.

Obowiązkowym parametrem tej metody jest “connection string” - specjalny adres URL pozwalający określić typ oraz lokalizację bazy danych.

Schemat URL połączenia

```
jdbc:mysql://localhost:3306/test
```

```
jdbc:sqlserver://localhost:1433;databaseName=AdventureWorks
```

Identyfikator drivera Adres bazy Numer portu Nazwa bazy

Obiekty Statement

Obiekty implementujące interfejs `Statement` służą bezpośredniej komunikacji z bazą danych - wykonywaniu zapytań i odczytywaniu wyników.

Obiekty te zwracają metody `Connection`, np. `createStatement()`.

Ten sam obiekt `Statement` może być wykorzystany wielokrotnie, do wykonywania kolejnych zapytań.

Wykonywanie zapytań SELECT

Do wykonywania zapytań SQL typu SELECT służy metoda `executeQuery()`, przyjmująca jako tekst zapytanie SQL.

Zwracanym obiektem jest `ResultSet`, który pozwala iterować przez zwrócone wyniki i pobierać z nich wartości.

```
ResultSet result = statement.executeQuery(sql);

result.first();
int intValue =     result.getInt(intColumnName);
String stringValue =   result.getString(stringColumnName);
```

Obiekt wynikowy ResultSet

id	title	year

wskaźnik

1	Dziesięciu Murzynków	1939
2	Mały Książę	1943
3	Władca Pierścieni	1954

Pobieranie pojedynczego rekordu

Na wyniku zapytania zwracającego jeden lub zero rekordów możemy wykonać metodę `first()`, która:

- zwraca wartość `boolean`, czy rekord istnieje,
- jeśli istnieje, ustawia iterator na pierwszym rekordzie.

```
boolean itemExists = result.first();
```

Zadanie

JdbcFilmDaoTest

findById

Zadanie

JdbcFilmDaoTest

findById_nonExistent

Pobieranie wielu rekordów

Wskaźnik przesuwamy do następnych wyników metodą `next()`.

Podobnie jak `first()`, ta metoda także zwraca informację czy następna wartość istnieje.

```
while (result.next()) {
    // read record
}
```

Zadania

JdbcStoreDaoTest

Prepared Statements

Metoda `Connection.prepareStatement()` pozwala stworzyć zapytanie, które zostanie raz skompilowane przez bazę danych i może być potem wielokrotnie użyte z różnymi parametrami (np. warunków `WHERE`).

```
PreparedStatement stmt = connection.prepareStatement(
    "SELECT title FROM book WHERE id = ?");

stmt.setInt(1, 42);
ResultSet result = stmt.executeQuery();
```

Zalety Prepared Statements

- baza danych tylko raz kompiluje zapytanie, przygotowując plan wykonania zapytania (execution plan), co redukuje czas wykonania

- podstawiane parametry są zabezpieczone przed atakiem SQL Injection, co zapewnia większe bezpieczeństwo niż w przypadku samodzielnej budowy zapytania przez sklejanie zmiennych

Zadanie

JdbcFilmDaoTest

findByIdWithPreparedStatement

Zadania

JdbcRentalDaoTest

Operacje DML

Do wykonywania operacji z grupy DML (Data Manipulation Language), takich jak `INSERT`, `UPDATE` i `DELETE`, służy metoda `executeUpdate()`.

```
statement.executeUpdate(updateSQL);
```

Zadanie

JdbcActorDaoTest

JPA i ORM

Zalety ORM

- w pełni obiektowe podejście, praca wyłącznie na obiektach odwzorowujących rekordy tabel
- optymalizacja - zarządzanie połączeniami, cache
- model tabeli zdefiniowany w jednym miejscu, łatwy do modyfikacji
- większe bezpieczeństwo (zabezpieczenie przed atakiem SQL Injection)
- łatwa przenoszalność pomiędzy bazami danych

Wady ORM

- narzut wydajnościowy (tworzenie obiektów, nieoptymalne zapytania)
- trudność tworzenia skomplikowanych zapytań
- trudność używania niestandardowych poleceń SQL
- ryzyko błędów wydajnościowych związanych z ilością wykonywanych zapytań

Warstwa Object-Relational Mapping

Encje

Pracując z ORM operacje wykonujemy względem klas i obiektów będących reprezentacjami rekordów w tabelach, a nie samych rekordów i tabel.

Obiekty reprezentujące rekordy nazywamy encjami.

Klasę encji tworzymy przez nadanie jej adnotacji `@Entity`.

```
@Entity
public class Book {
    @Id
    private Integer id;
}
```

Konfiguracja JPA

Frameworki ORM w Javie są zgodne ze specyfikacją Java Persistence API, niekiedy dodając swoje dodatkowe funkcje.

Konfiguracja JPA ustawiana jest w pliku `resources/META-INF/persistence.xml`, gdzie dla każdej bazy z którą chcemy się łączyć definiujemy Persistence Unit.

Obiekt EntityManager

Podstawowym obiektem pośredniczącym w komunikacji pomiędzy aplikacją a ORM jest `EntityManager`.

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("myPU");

EntityManager em = entityManagerFactory.createEntityManager();
```

Język JPQL

Ponieważ operacje nie odbywają się względem tabel a obiektów encji, do tworzenia zapytań służy specjalny język Java Persistence Query Language.

JPQL jest bardzo podobny do SQL, lecz używamy w nim nazw obiektów i ich pól, a nie tabel i ich kolumn.

Tworzenie zapytań JPQL

Do tworzenia zapytań JPQL służy metoda `createQuery()` przyjmująca zapytanie oraz klasę oczekiwanego obiektu wynikowego.

Na powstałym zapytaniu możemy wywołać metodę `getSingleResult()`, która zwraca pojedynczy obiekt.

```
Book book = em
    .createQuery(
        “SELECT b FROM Book b WHERE b.id=1”, Book.class
    )
    .getSingleResult();
```

Zadanie

FilmDaoTest

findWithId1

Parametry w zapytaniach JPQL

Podobnie jak PreparedStatement, zapytania JPQL przyjmują parametry.

Mogą być one nazwane, zamiast numerowane, co zwiększa czytelność i pomaga unikać błędów.

```
Book book = em
    .createQuery("FROM Book b WHERE b.id = :id", Book.class)
    .setParameter("id", 1)
    .getSingleResult();
```

Zadanie

FilmDaoTest

findByIdWithParameter

Wyjątki

Metoda `getSingleResult()` oczekuje, że wynikiem zapytania będzie dokładnie jeden rekord. W przeciwnym wypadku rzuci wyjątek:

- `NoResultException`, jeśli w wyniku nie będzie żadnego rekordu,
- `NonUniqueResultException`, jeśli w wyniku będzie więcej niż jeden rekord.

Zadanie

FilmDaoTest

findByIdWithParameter_nonExistent

Pobieranie encji wg klucza głównego

Do pobrania pojedynczego obiektu wg jego klucza głównego (id) służy metoda `find()`.

```
Book book = em.find(Book.class, 1);
```

Zadanie

FilmDaoTest

findByIdWithEM

Zadanie

FilmDaoTest

findByIdWithEM_nonExistent

Zadanie

StoreDaoTest

findById

Relacje

Relacje określane są przez adnotacje na polach encji:

- @OneToOne
- @OneToMany / @ManyToOne
- @ManyToMany

```
@Entity
public class Book {
    @ManyToOne
    private Author author;
}
```

```
@Entity
public class Author {
    @OneToMany
    private List<Book> books;
}
```

Relacje - klucz obcy

Dodatkowo adnotacją `@JoinColumn` możemy zdefiniować nazwę kolumny zawierającej klucz obcy (foreign key).

```
@Entity
public class Book {
    @ManyToOne
    @JoinColumn(name = “author_id”)
    private Author author;
}
```

Zadanie

StoreDaoTest

findById_withAddress

Zwracanie wielu obiektów

Wykonując zapytanie zwracające wiele wyników należy użyć metody getResultList(), aby otrzymać listę encji.

```
List<Book> books = em
    .createQuery("SELECT b FROM Book b", Book.class)
    .getResultList();
```

Zadanie

StoreDaoTest

findAll_withAddresses

Ilość wykonywanych zapytań

Pobierając encje przez ORM należy zważać na faktyczną liczbę wykonywanych zapytań.

Jeżeli pobierzemy z bazy encję posiadającą relację do innej encji,
a następnie odwołamy się do pola tej relacji,
ORM automatycznie pobierze tą encję.

Prowadzi to do problemu “n+1”, gdzie jednym zapytaniem pobieramy listę encji, a następnie ORM wykonuje n zapytań by dociągnąć wszystkie relacje.

Operacja JOIN FETCH

Do odwołania do powiązanych relacją encji w JPQL służy, podobnie jak w SQL, operacja `JOIN`. Może to służyć np. filtrowania wg tej relacji.

By pobrać także te encje należy użyć operacji `JOIN FETCH`.

```
SELECT b FROM Book b LEFT JOIN FETCH b.author
```

Zadanie

StoreDaoTest

findAll_withAddresses_singleQuery

Zadanie

StoreDaoTest

findAll_withAddressesAndCities_singleQuery

Operator WHERE

Operator `WHERE`, podobnie jak w SQL, pozwala filtrować wyniki zapytania.

```
SELECT b FROM Book b WHERE b.year = 2005
```

Zadanie

CustomerDaoTest

findByLastName

Zadanie

CustomerDaoTest

findByLastName_singleQuery

Operator LIKE i CONCAT

Operator `LIKE`, podobnie jak w SQL, pozwala filtrować tekst na podstawie jego fragmentu. Znakiem % określamy dowolny fragment tekstu.

Operator `CONCAT` pozwala scalić fragmenty tekstu w jeden parametr, np. nazwany parametr i znak %.

```
SELECT b FROM Book b WHERE b.title LIKE CONCAT(:title, “%”)
```

Zadanie

CustomerDaoTest

findByLastNameStartinWith

Funkcje agregujące

JPQL wspiera operatory `GROUP BY` i `HAVING` oraz podstawowe funkcje agregujące: `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`.

```
SELECT COUNT(b) FROM Book b
```

Zadanie

CustomerDaoTest

findByCity

Zadanie

CustomerDaoTest

countByCity

Dalsze zagadnienia

Dalsze zagadnienia

- operacje na encjach: `persist()`, `remove()`, `detach()`, `merge()`
- relacje EAGER i LAZY
- generowanie bazy danych z klas encji
- biblioteka Spring Data JPA (https://spring.io/projects/spring-data-jpa)
- biblioteka JOOQ (https://www.jooq.org/)