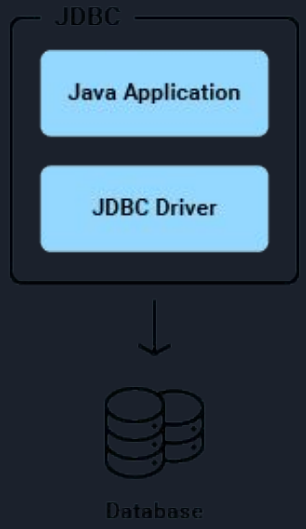
Spring Data Access with JdbcTemplate

Julia Pabiańczyk
Michał Czarnik
Mikołaj Telec
Norbert Faron
Damian Skorupa
Kamil Rzeszutek

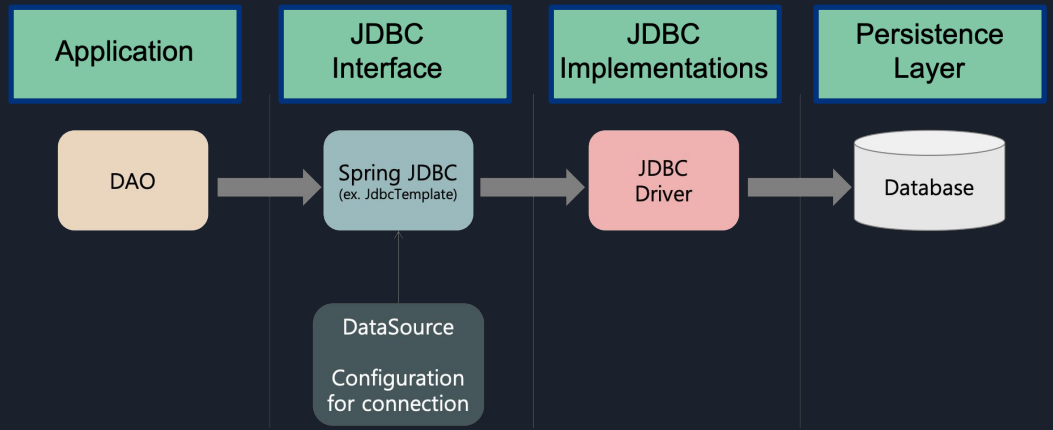
Jdbc - Java DataBase Conectivity

- Jdbc to standardowy interfejs API umożliwiający łączenie się z relacyjnymi bazami danych
- Jdbc oferuje realizację operacji CRUD (Create, Read, Update, Delete) w łatwy sposób
- Jest to podstawowe narzędzie używane w różnych frameworkach
- Otrzymane dane trzeba zmapować na oczekiwany obiekt



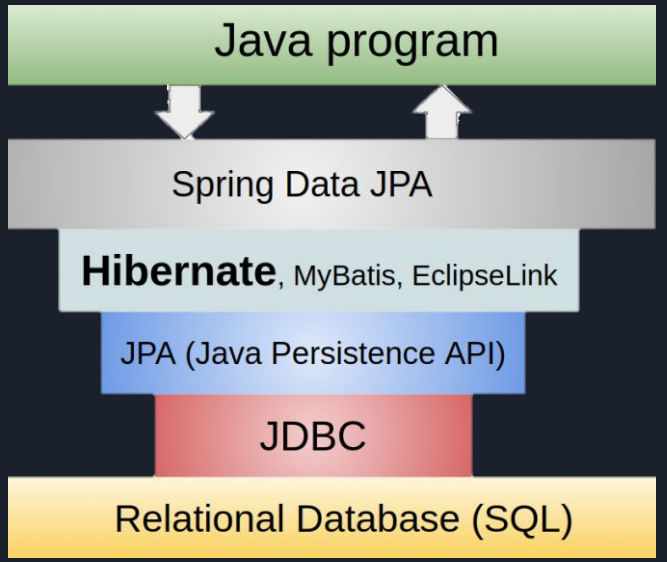
JdbcTemplate

- JdbcTemplate jest interfejsem upraszczającym wykonywanie zapytań za pomocą interfejsu JDBC.
- Pozwala wyeliminować podstawowe błędy przy wykorzystaniu czystego JDBC.
- Dzięki użyciu JdbcTemplate kod jest bardziej zwięzły i przejrzysty.



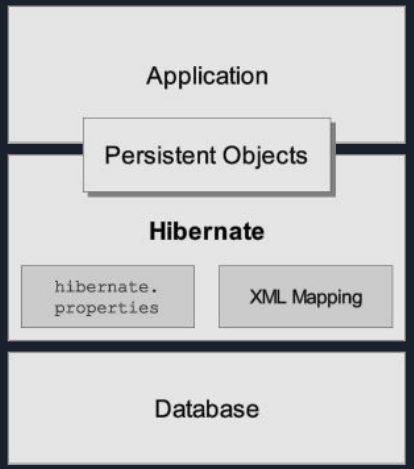
ORM - Object-Relational Mapping

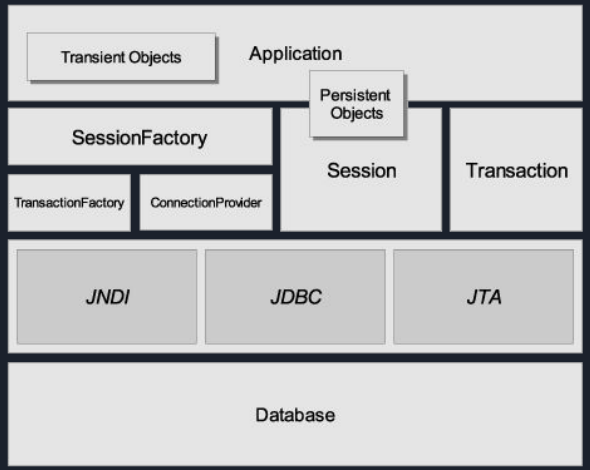
- ORM to technika służąca do odwzorowania obiektów domenowych na tabele relacyjnej bazy danych.
- Najpopularniejszą biblioteką ORM w Javie jest Hibernate.
- ORM dostarcza narzędzia, dzięki którym komunikacja z bazą nie wymaga dużej ilości kodu.
- Ma dobrą obsługę wyjątków i jest łatwa do testowania.
- Zapewnia zarządzanie transakcjami.
- Biblioteka Spring Data JPA jest dodatkową nakładką na technikę ORM, która ułatwia dostęp do danych. Dostarcza podstawowe operacje do wyciągania danych za pomocą interfejsów, które wystarczy rozszerzyć. Największą wadą jest spowolnienie procesu



Hibernate

- Aplikacja sama zarządza transakcjami, połączeniami, etc. Używany jest tylko minimalny podzbiór funkcjonalności Hibernate.





Wybór odpowiedniego narzędzia

- Jeśli zamierzamy manipulować na danych w reprezentacji klas-relacja to zdecydowanie wygodniejszym sposobem jest wykorzystanie ORM.
- Wykorzystanie biblioteki Spring Data JPA przyśpiesza nam proces budowania warstwy dostępu do danych.
- W sytuacji, w której potrzebujemy stworzyć skomplikowane zapytanie i niekoniecznie potrzebujemy potem uzyskać reprezentację obiektową, JdbcTemplate może okazać się lepszym i wygodniejszym rozwiązaniem

Lombok - wstęp

“Lombok is a ***compilation*** dependency, not a runtime dependency. You don’t need to have Lombok in your classpath at runtime at all.”

Po co jest Lombok?

```
1  package pl.jdbcqueryexample.jdbcquery.dto;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Getter;
5  import lombok.NoArgsConstructor;
6  import lombok.Setter;
7
8  @AllArgsConstructor
9  @Setter
10 @Getter
11 @NoArgsConstructor
12 public class SongCreateDTO {
13
14     private String name;
15     private Long artistId;
16
17 }
```

Lombok - najważniejsze adnotacje

```
@ToString

@EqualsAndHashCode

@NoArgsConstructor

@RequiredArgsConstructor

@AllArgsConstructor

@Data
```

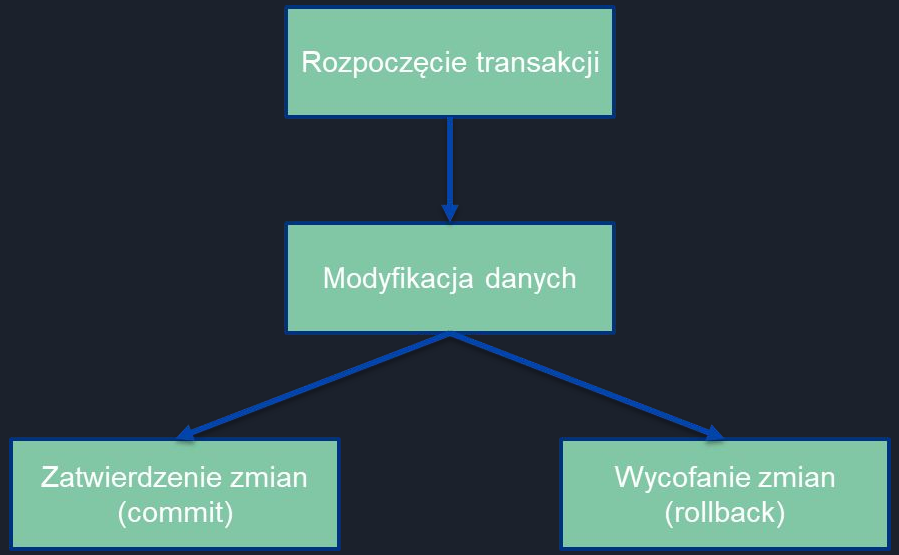
Lombok - wady i zalety

Zalety	***Wady***
Zmniejsza ilość kodu w projekcie	Niektóre adnotacje nie współpracują ze sobą
Przyspiesza pisanie kodu	Programista nie widzi kodu, który został wygenerowany przez Lombok
Jest używany przez duże projekty np. Spring Data	Korzystanie z różnych bibliotek (takich jak np. Hibernate) może powodować konieczność automatycznego dodawania niektórych adnotacji

Transakcje - wstęp

Transakcje to mechanizm, który zapewnia prawidłowy przebieg operacji z wykorzystaniem bazy danych

Transakcje - koncepcja



Transakcje – gwarancje

ACID, czyli:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

@Transactional

- Włącza obsługę transakcji
- Dodawana do klas oraz ich metod
- Dostarcza dane oraz API dla ORM, który odpowiada za wymianę danych z bazą
- Umożliwia dwukierunkową integrację Springa z ORM
- Uruchamia mechanizmy monitorujące rzucane wyjątki czy też czas wykonania operacji
- Zatwierdza lub wycofuje zmiany na bazie po wyjściu z metody

@Transactional - przykład

```
@AllArgsConstructor
@Service
class SongService implements ISongService
```

```
@Override
@Transactional
public SongDTO create(SongCreateDTO dto) {
    var artist : Artist = artistQueryService.getById(dto.getArtistId());
    var created : Song = songCudService.create(SongCreator.songBySongCreateDTOAndArtist(dto , artist));
    return SongCreator.songDTOBySong(created);
}
```

@Transactional - konfiguracja

- Manager
- Rodzaj propagacji
- Poziom izolacji
- Maksymalny czas życia transakcji
- Flaga read-only
- Sterowanie wyjątkami (rollback)

@Transactional - propagacja

- REQUIRED
- SUPPORTS
- REQUIRES_NEW
- MANDATORY
- NOT_SUPPORTED
- NEVER
- NESTED

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
```

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
```

@Transactional - rollback

- rollbackFor

```
@Transactional(rollbackFor = { SQLException.class })
```

- rollbackForClassName
- noRollbackFor

```
@Transactional(noRollbackFor = { TimeoutException.class })
```

- noRollbackForClassName

Transakcje - kosztowność

- Wymagana większa ilość zasobów po stronie aplikacji i serwera baz danych
- Dodatkowy czas poświęcany na:
  - Zatwierdzanie lub wycofanie operacji
  - Monitoring zmian oraz wyjątków
  - Uruchamianie dodatkowych zasobów (np. procesy i połączenia sieciowe)

Endpointy

Endpointy to punkty końcowe, które służą do wymiany danych między różnymi systemami lub komponentami

Endpointy - tworzenie

Aby stworzyć endpoint, należy najpierw zdefiniować metodę w kontrolerze, która będzie obsługiwać żądanie. Następnie należy oznaczyć tę metodę adnotacją odpowiedniego rodzaju, w zależności od tego, jakiego typu żądanie ma obsługiwać (GET, POST, PUT, itd.)

Endpointy - przykład

```java
@AllArgsConstructor
@RequestMapping("/artist")
@Controller
public class ArtistController {

    private final IArtistService artistService;
    private final ArtistValidator validator;


    @GetMapping
    public ResponseEntity <Set <ArtistBasicDTO>> getAll() {
        return new ResponseEntity <>(artistService.getAllBasicArtistDTO() , HttpStatus.OK);
    }

    @GetMapping("/{id}")
    public ResponseEntity <ArtistBasicDTO> getById(@PathVariable Long id) {
        return new ResponseEntity <>(artistService.getBasicArtistDTOById(id) , HttpStatus.OK);
    }
```

Operacje crudowe

Operacje CRUD to cztery podstawowe operacje, które mogą być wykonywane na danych w aplikacjach internetowych. CRUD to skrót od Create, Read, Update, Delete, co oznacza tworzenie, odczytywanie, aktualizowanie i usuwanie danych.

Operacje CRUD są bardzo często używane w aplikacjach internetowych do zarządzania danymi, na przykład do zarządzania bazami danych lub zasobami w chmurze.

CRUD	HTTP	REST
Create	POST	/api/movie
Read	GET	/api/movie/{id}
Update	PUT	/api/movie
Delete	DELETE	/api/movie/{id}

READ & CREATE

```java
@Transactional(readOnly = true)
@Override
public Set <Song> findAll() {
    String sql = "SELECT * FROM song s INNER JOIN artist a ON s.artist_id = a.id";
    List <Song> songs = jdbcTemplate.query(sql , rowMapper: this);
    return new HashSet <>(songs);
}
```

```java
@Transactional
@Override
public Song save(Song entity) {
    String sql = String.format("INSERT INTO song (name, artist_id) VALUES('%s',%d)" , entity.getName() , entity.getArtist().getId());
    KeyHolder keyHolder = new GeneratedKeyHolder();
    jdbcTemplate.update(connection -> connection.prepareStatement(sql , Statement.RETURN_GENERATED_KEYS) , keyHolder);
    if ( Objects.isNull(keyHolder.getKey()) ) {
        throw new SQLModifyException("Cannot create Song: " + entity.getName());
    }
    return findById(keyHolder.getKey().longValue());
}
```

UPDATE & DELETE

```java
@Transactional
@Override
public Song update(Song entity) {
    String sql = String.format("UPDATE song SET name='%s' WHERE id=%d" , entity.getName(), entity.getId());
    jdbcTemplate.update(sql);
    return findById(entity.getId());
}
```

```java
@Transactional
@Override
public void deleteById(Long id) {
    String sql = "DELETE FROM song WHERE id = ?";
    jdbcTemplate.update(sql , id);
}
```

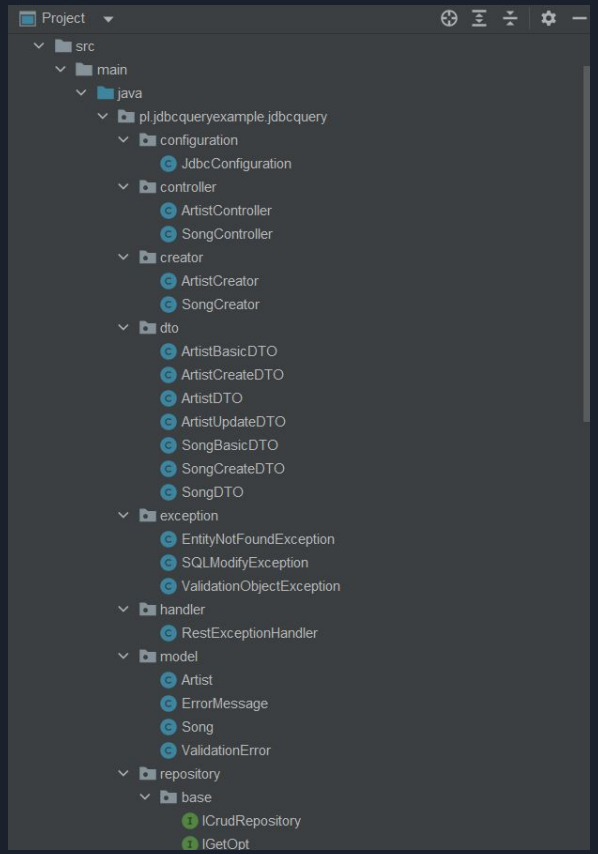
JOINY

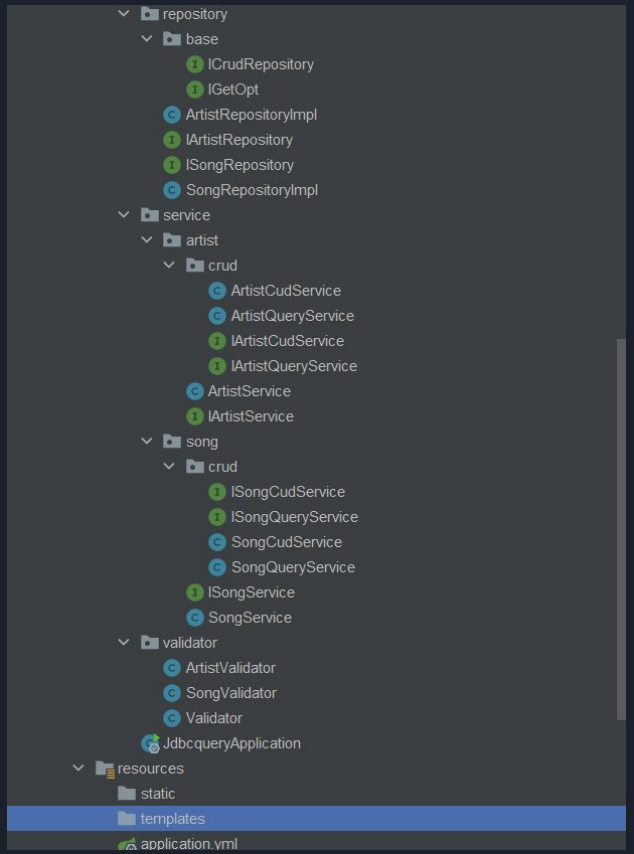
```java
@Transactional(readOnly = true)
@Override
public Song findById(Long id) {
    String sql = "SELECT * FROM song s INNER JOIN artist a ON s.artist_id = a.id WHERE s.id = ?";
    List <Song> songs = jdbcTemplate.query(sql , rowMapper: this , id);
    switch (songs.size()) {
        case 0:
            return null;
        case 1:
            return songs.get(0);
        default:
            throw new IncorrectResultSizeDataAccessException(songs.size());
    }
}
```

```java
@Override
public Song mapRow(ResultSet rs , int rowNum) throws SQLException {
    return Song.builder()
            .id(rs.getLong( columnLabel: "s.id"))
            .name(rs.getString( columnLabel: "s.name"))
            .artist(Artist.builder()
                    .id(rs.getLong( columnLabel: "a.id"))
                    .name(rs.getString( columnLabel: "a.name"))
                    .lastName(rs.getString( columnLabel: "a.last_name"))
                    .nickname(rs.getString( columnLabel: "a.nickname"))
                    .build()
            )
            .build();
}
```

Aplikacja

https://github.com/norbert15174/jdbcquery





Aplikacja

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/dbquery?serverTimezone=UTC
    username: root2
    password: ${DB_PASSWORD}
    driver-class-name: com.mysql.cj.jdbc.Driver
  liquibase:
    change-log: classpath:changelog.xml
```

DZIĘKUJEMY ZA UWAGĘ!