Spring Data Access with Jdbc Template

Dawid Mazurkiewicz, Grzegorz Pryjma, Kamil Maksymowicz,
Maksymilian Siembab, Aleksandra Radziak,
Andrzej Sobierajski, Krzysztof Rokosz

Agenda

1. Project setup
2. Our entity and data models
3. Accessing Data
4. Adding the REST Endpoints

Project setup

Do stworzenia funkcjonalnego projektu należy:

- stworzyć odpowiednią strukturę katalogów
- przygotować `build.gradle` z dołączeniem:
  - `spring-boot-starter-jdbc` - wprowadza wszystko czego potrzebuje Spring do zapewnienia wsparcia i ekosystemu Spring JDBC
  - `spring-boot-starter-web` - zapewnienie działania backend'u w pełni na potrzeby opisanego projektu
  - Lombok - procesor adnotacji dla Javy, przydatny w generowaniu kodu wzorcowego.

```
plugins {
    id 'org.springframework.boot' version '2.1.4.RELEASE'
}

apply plugin: 'io.spring.dependency-management'

dependencies {
    compile "com.h2database:h2:1.4.199"
    compile "org.springframework.boot:spring-boot-starter-web"
    compile "org.springframework.boot:spring-boot-starter-jdbc"

    compileOnly "org.projectlombok:lombok"
    // we want the most recent release of lombok, so "1.+"
    annotationProcessor "org.projectlombok:lombok:1.+"

    testCompile "org.springframework.boot:spring-boot-starter-test"
}
```

Lombok

Procesor adnotacji dla języka Java co oznacza, że korzysta z jej mechanizmu, który pozwala przed kompilacją wygenerować kod.
Uogólniając Lombok pozwala zaoszczędzić czas poprzez generowanie powtarzalnego kodu (np. funkcje @Getter, @Setter).
Lombok posiada wiele funkcji m. in.:

- @ToString - dodaje metodę ToString()
- @EqualsAndHashCode - generują metody equals() i hashCode()
- @NoArgsConstructor - generowanie konstruktorów
- @RequiredArgsConstructor
- @AllArgsConstructor
- @Data - generuje wszystko to co adnotacje @ToString, @EqualsAndHashCode, @RequiredArgsConstructor, @Getter i @Setter

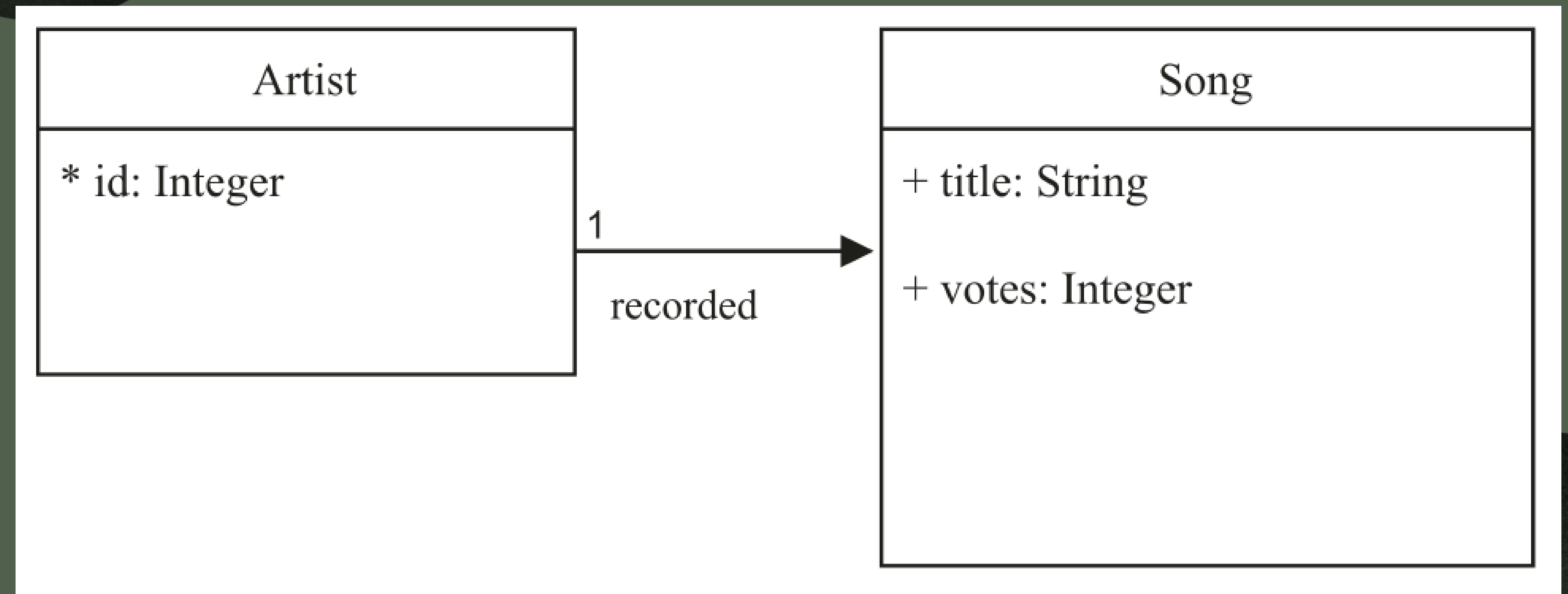
```
package com.bsg5.chapter8;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;
import org.springframework.lang.NonNull;

@Data
@AllArgsConstructor
@RequiredArgsConstructor
@NoArgsConstructor
public class Artist {
    Integer id;
    @NonNull
    String name;
}
```

Our Entity and Data Model

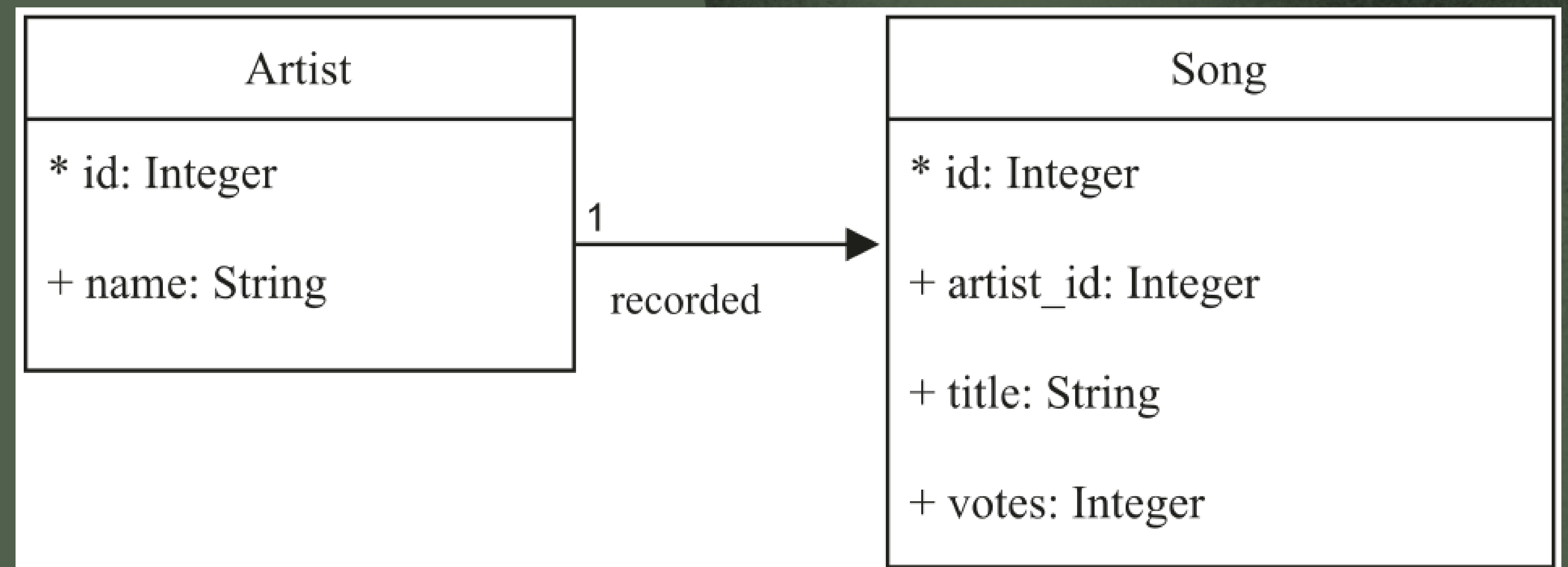
Czym się różnią?

Terminy entity model i data model są często używane zamiennie jako iż nie mają formalnej definicji. W tym przypadku jednak mianem entity model będziemy określać relacje między jednostkami (entity) a data model oznaczać będzie szczegóły opis używanych jednostek. Na przykład: entity model opisuje, że jednostka Artist istnieje i posiada imie. Data model natomiast opisuje rzeczy przydatne z programistycznego punktu widzenia, jak klucz główny, co nie jest uwzględnione w definicji Artist.

Entity model



Data model



Manualne tworzenie bazy danych

W celu utworzenia bazy np. do testwania aplikacji można użyć bazy h2. Aby to zrobić należy najpierw załączyć ją do projektu i wskazać na wykorzystanie takiego serwera w odpowiednim pliku konfiguracyjnym. Kiedy baza jest już skonfigurowana można utworzyć schemat bazy danych w SQLu.

```
spring.datasource.platform=h2
```

```
CREATE TABLE IF NOT EXISTS artists
(
  id   IDENTITY,
  name VARCHAR(64) NOT NULL
);
CREATE UNIQUE INDEX IF NOT EXISTS artist_name
  ON artists(name);

CREATE TABLE IF NOT EXISTS songs
(
  id        IDENTITY,
  artist_id INT,
  name      VARCHAR(64) NOT NULL,
  votes     INT DEFAULT 0,
  FOREIGN KEY (artist_id) REFERENCES artists (id)
    ON UPDATE CASCADE
);
CREATE UNIQUE INDEX IF NOT EXISTS song_artist
  ON SONGS (artist_id, name);
```

Accessing Data

JdbcTemplate

JdbcTemplate jest interfejsem ułatwiającym wykonywanie zapytań za pomocą interfejsu JDBC, w celu pobrania rekordu z bazy danych i zmapowania go do odpowiedniego obiektu.

Interfejs ten umożliwia pracę bez konieczności otwierania i zamykania połączeń czy obsługi wyjątków.

Interfejs ten posiada 19 wariantów metody *query()* takich jak *queryForObject()*, *queryForList()* służących do egzekwowania zapytań SQL, iteracji po danych wejściowych i utworzenia nowego obiektu.

JdbcTemplate

```
public <T> List<T> query(
  String sql,
  @Nullable Object[] args,
  RowMapper<T> rowMapper
) throws DataAccessException
```

Korzystając z metod *query()* interfejsu JdbcTemplate należy stworzyć polę przechowujące zapytanie SQL.
Kolejnym argumentem jest lista typu *Object* służąca do przechowywania parametrów użytych w zapytaniu.
Trzeci argument to *RowMapper* - interface wykorzystywany do mapowania danych i zwrócenia zbioru wynikowego.

RowMapper

RowMapper to interfejs mapujący każdy wiersz zapytania na model obiektowy. Odpowiedzialny jest za akceptację zbioru wynikowego i liczby wierszy w postaci liczby całkowitej.

Posiada tylko jedną metodę *mapRow*, umożliwiającej jej reprezentowanie wykorzystując lambdę.

```
new RowMapper<String>() {
 @Override
    String mapRow(ResultSet rs, int rowNum) throws SQLException {
  return rs.getString("name");
 }
}
```

@Transactional

Transakcje to mechanizm, który umożliwia zachowanie prawidłowego przebiegu operacji z wykorzystaniem bazy danych

```
@Transactional
public List<Song> getSongsForArtist(String artistName) {
    String selectSQL = "SELECT id, artist_id, name, votes " +
            "FROM songs WHERE artist_id=? " +
            "order by votes desc, name asc";
    Artist artist = internalFindArtistByName(artistName, true);
    return jdbcTemplate.query(
            selectSQL, new Object[]{artist.getId()},
            songRowMapper);
}
```

Adnotacja @Transactional posiada kilka atrybutów, służących do konfiguracji transakcji:

Rodzaj propagacji transakcji:

- REQUIRED
- SUPPORTS
- NEVER
- NOT_SUPPORTED
- REQUIRES_NEW

Poziom izolacji transakcji:

- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

x	Dirty Read	Non Repeatable Read	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Commited	Solved	Possible	Possible
Repeatable Read	Solved	Solved	Possible
Serializable	Solved	Solved	Solved

- Dopuszczalny maksymalny czas życia transakcji

```
@Transactional(timeout=5))
```

- Sterowanie wyjątkami dla wycofywania transkacji

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

- Flaga tylko do odczytu

```
@Transactional(readOnly = true)
public List<String> getMatchingArtistNames(String artistName) {
    return artistRepository
        .findAllByNameIsLikeIgnoreCaseOrderByName(
            converter.convertToWildCard(artistName))
        .stream()
        .map(A::getName)
        .collect(Collectors.toList());
}
```

Adding the REST Endpoints

ArtistController

Porównanie dwóch wersji kodu mających tą samą logikę. Po lewej widzimy tradycyjną wersją kodu. Po prawej wersję z użyciem strumieniowania.

```
@RestController
public class ArtistController {
    private MusicRepository service;

    ArtistController(MusicRepository service) {
        this.service = service;
    }

    @GetMapping(value = {"/artists/search/{name}", "/artist/search/"},
        produces = MediaType.APPLICATION_JSON_VALUE)
    Artist findArtistByName(
        @PathVariable(required = false) String name
    ) {
        if (name != null) {
            Artist artist = service.findArtistByNameNoUpdate(decode(name));
            if (artist != null) {
                return artist;
            } else {
                throw new ArtistNotFoundException();
            }
        } else {
            throw new IllegalArgumentException("No artist name submitted");
        }
    }
```

```
@RestController
public class ArtistController {
    private MusicRepository service;

    ArtistController(MusicRepository service) {
        this.service = service;
    }

    Optional<Artist> findArtistByName(Optional<String> name, boolean update) {
      return Optional.of(service.findArtistByName(
        decode(
          name.orElseThrow(
            () -> new IllegalArgumentException("No artist name supplied")
          )
        )
      ));
    }
    @GetMapping({"/artist/search/{name}", "/artist/search/"})
    Artist findArtistByName(
      @PathVariable(required = false) Optional<String> name
    ) {
      Optional<Artist> artistOptional = findArtistByName(name, false);
      return artistOptional.orElseThrow(
        ArtistNotFoundException::new
      );
    }
```

SongController

Kolejnym krokiem jest utworzenie SongController'a. Zawiera on następujące metodę decode oraz następujące metody GET.

- **Song voteForSong(String name, String title)**
- **Song getSong(String name, String title)**
- **List<Song> getSongsForArtist(String name)**
- **List<String> findSongsForArtist(String name, String title)**

SongControllerTest

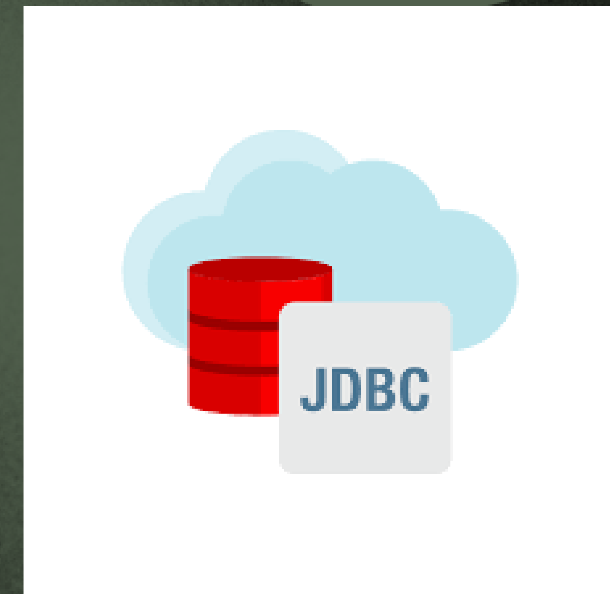
Zawarte tutaj testy, pokrywają niemal wszystko, co zostało utworzone w projekcie.

```java
@Test
void testSongsForArtist() {
    ParameterizedTypeReference<List<Song>> type =
        new ParameterizedTypeReference<>() {
    };
    String url = "http://localhost:"
        + port
        + "/artists/"
        + encode("Threadbare Loaf")
        + "/songs";
    ResponseEntity<List<Song>> response = restTemplate.exchange(
            url,
            HttpMethod.GET,
            null,
            type
    );
    assertEquals(response.getStatusCode(), HttpStatus.OK);
    List<Song> songs =response.getBody();

    assertEquals(songs.size(), 2);
    assertEquals(songs.get(0).getName(), "What Happened To Our
    First CD?");
    assertEquals(songs.get(0).getVotes(), 17);
    assertEquals(songs.get(1).getName(), "Someone Stole the Flour");
    assertEquals(songs.get(1).getVotes(), 4);
}
```

Podsumowanie

W prezentacji omówiono:

- Project setup;
- Our entity and data models;
- Accessing Data;
- Adding the REST Endpoints;



JdbcTemplate

Dziękujemy za uwagę