

1. Stworzenie projektu za pomocą SpringBoot'a oraz wstępna konfiguracja:

1.1. Wchodzimy na [Spring Initializr \(start.spring.io\)](https://start.spring.io) oraz konfigurujemy nasz projekt w następujący sposób:

- Project - zakładka w której wybieramy narzędzie, za pomocą którego będziemy budowali aplikację.
 - **Maven Project**
- Language - język w którym tworzymy aplikację
 - **Java**
- Spring boot - wersja Spring'a (zalecane wybranie najnowszej i stabilnej wersji (bez nawiasów przy numerem wersji - nie "snapshot", "m2(milestone 2)" etc.)
 - **Na czas tworzenia instrukcji 2.7.5**
- Project Metadata - metadane naszego projektu
 - **Przykładowe wypełnienie na obrazie poniżej**
- Packages
 - **Jar**
- Java - wersja Javy
 - wybrana została 17 (można wybrać inną np. najnowszą)**
- Dependencies - bardzo ważna sekcja, gdzie dodamy zależności potrzebne do projektu
 - **Spring Web,**
 - **JDBC API,**
 - **MySQL Driver,**
 - **w celu usprawnienia tworzenia aplikacji dodano również Lombok - [Project Lombok](#), [Introduction to Project Lombok](#)**

1.2. Poprawna konfiguracja:

The screenshot displays the Spring Initializr web form with the following configuration:

- Project:** ☒ Gradle Project, ☒ **Maven Project**
- Language:** ☒ **Java**, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☐ 3.0.0 (SNAPSHOT), ☐ 3.0.0 (RC2), ☐ 2.7.6 (SNAPSHOT), ☒ **2.7.5**, ☐ 2.6.14 (SNAPSHOT), ☐ 2.6.13
- Project Metadata:**
 - Group:
 - Artifact:
 - Name:
 - Description:
 - Package name:
 - Packaging: ☒ **Jar**, ☐ War
 - Java: ☐ 19, ☒ **17**, ☐ 11, ☐ 8
- Dependencies:**
 - Spring Web** ☒ **WEB**: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Lombok** ☒ **DEVELOPER TOOLS**: Java annotation library which helps to reduce boilerplate code.
 - MySQL Driver** ☒ **SQL**: MySQL JDBC and R2DBC driver.
 - JDBC API** ☒ **SQL**: Database Connectivity API that defines how a client may connect and query a database.

At the bottom, there are three buttons: , , and

1.3. Generowanie i otwieranie projektu:

Na dole naszej strony klikamy przycisk *Generate*, a następnie wypakowujemy nasz projekt z zip'a. Nasze archiwum powinno zawierać następujące pliki:

Nazwa	Rozmiar	Skompres...	Typ	Zmodyfiko...	CRC32
..	Folder plików				
.mvn	58 960	52 127	Folder plików	18.11.2022 ...	
src	562	341	Folder plików	18.11.2022 ...	
.gitignore	395	233	Dokument teks...	18.11.2022 ...	C3F389...
HELP.md	1 420	584	Plik MD	18.11.2022 ...	BA288...
mvnw	10 284	3 287	Plik	18.11.2022 ...	A9285...
mvnw.cmd	6 734	2 550	Skrypt poleceń...	18.11.2022 ...	24AD8...
pom.xml	1 843	599	Dokument XML	18.11.2022 ...	29B197...

Projekt otwieramy w IntelliJ przykładowo otwierając plik pom.xml za pomocą naszego IDE. Bądź przy otwartym środowisku wybierając *Open* z zakładki *File* z paska menu.

1.4. Konfiguracja bazy

Do pierwszego uruchomienia naszego projektu potrzebujemy konfiguracji naszej bazy danych. Składa się na nią url do naszej bazy danych, nazwa użytkownika oraz hasło. Wartości tych właściwości zostaną wykorzystane przy tworzeniu bazy (w dalszej części instrukcji).

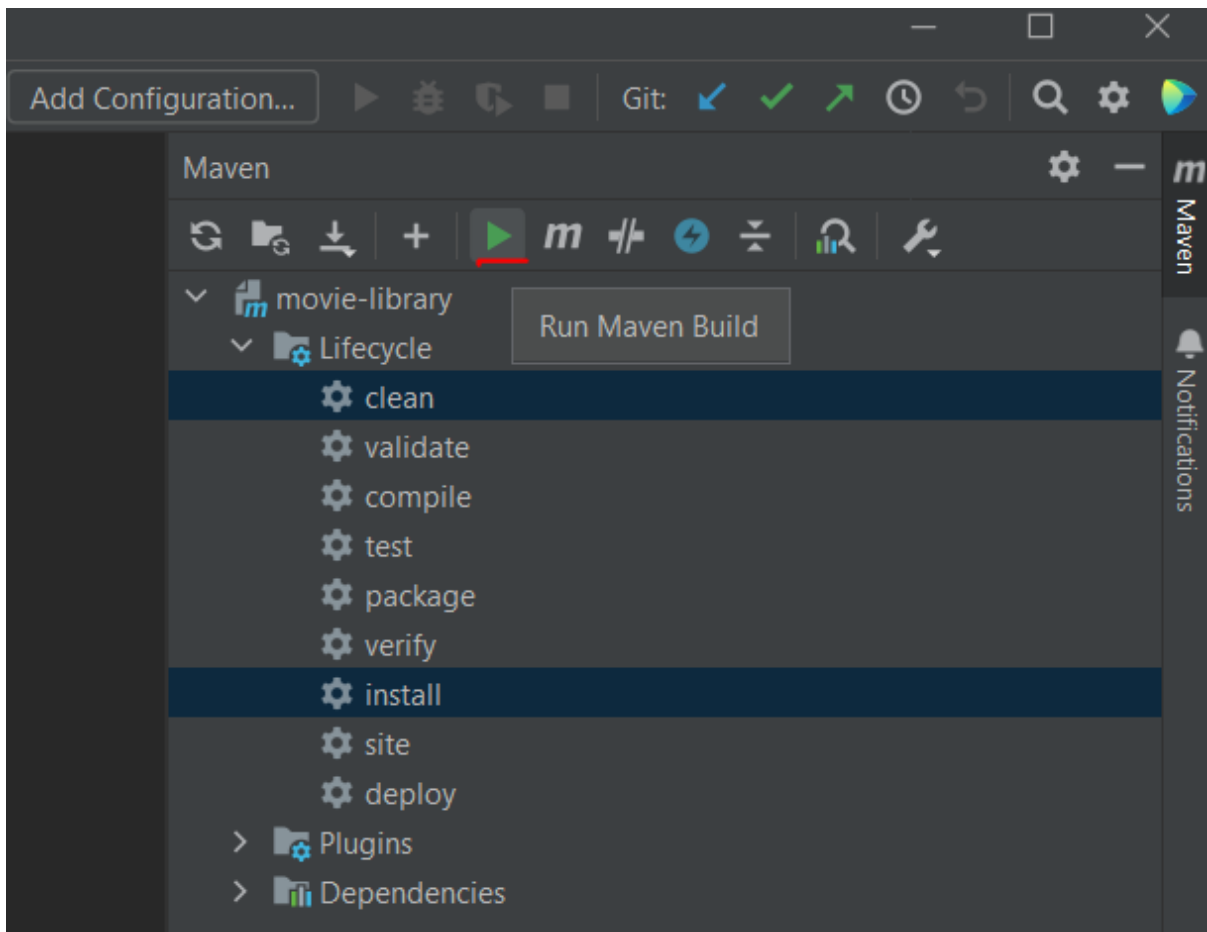
```
application.properties
1 spring.datasource.url=jdbc:mysql://localhost:3306/movie_library?useSSL=false
2 spring.datasource.username=root
3 spring.datasource.password=movieolib
4
```

1.5. Uruchamianie projektu.

Przy pierwszym uruchomieniu budujemy projekt, pozwalamy Mavenowi pociągnąć zależności, które znajdują się w wygenerowanym przez SpringBoot'a pom.xml. W tym celu robimy clean install. W instrukcji podane na to zostaną dwa sposoby

Pierwszy sposób, to uruchomienie terminal'a w IntelliJ Idea oraz wpisanie polecenia *mvn clean install*.

Drugą metodą jest otwarcie zakładki Maven'a (domyślnie znajduje się w prawym górnym rogu naszego IDE razem z zakładką *Notifications*), zaznaczenie dwóch pożądaných opcji (*clean* oraz *install* z *Lifecycle*) oraz uruchomienie builda za pomocą zielonego trójkąta.



Jeśli budowa nie będzie zakończona sukcesem, najprawdopodobniej jest to wina braku konfiguracji bazy danych (screen pierwszy) lub wina błędnej konfiguracji (screen drugi). Błędy te zlokalizować scrollując wyżej w logach budowy projektu w IntelliJ.

Przykładowe błędy:

```
APPLICATION FAILED TO START
*****

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class
```

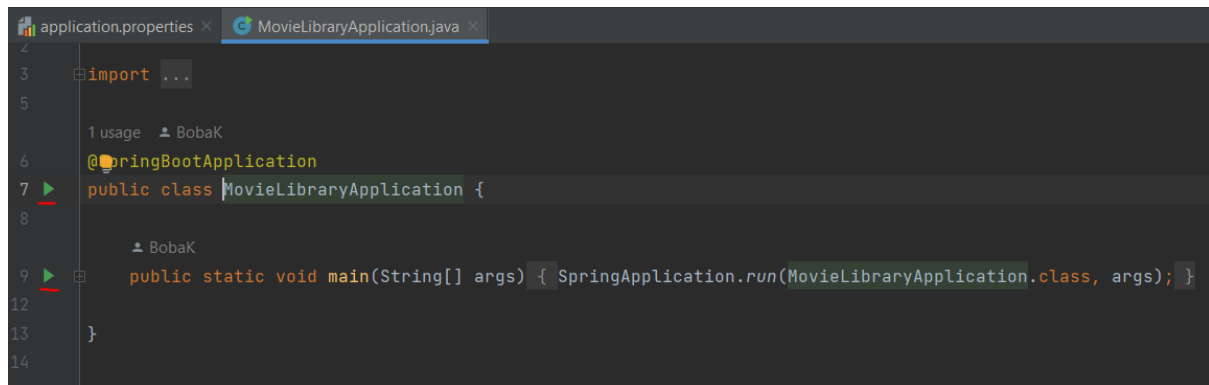
```
*****
APPLICATION FAILED TO START
*****

Description:

Failed to configure a DataSource: no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class
```

Po budowie czas uruchomić aplikację klikając klasę wygenerowaną przez SpringBoota. Jedną z opcji jest otwarcie klasy i kliknięcie jednego z zielonych trójkątów.



```
2
3 import ...
4
5
6 @SpringBootApplication
7 public class MovieLibraryApplication {
8
9     @BobaK
10    public static void main(String[] args) { SpringApplication.run(MovieLibraryApplication.class, args); }
11
12 }
13
14
```

Po uruchomieniu aplikacji widzimy w logach, że działa ona na porcie 8080, wchodzimy zatem na localhost:8080 w przeglądarce i oczom powinien nam się ukazać poniższy komunikat. Jeśli go widzimy - wszystko jest w porządku, tak wygląda pusta aplikacja.



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

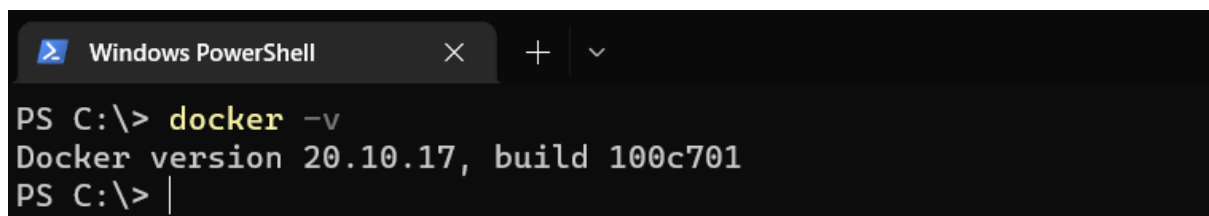
Sun Nov 20 13:06:25 CET 2022

There was an unexpected error (type=Not Found, status=404).

2. Konfiguracja bazy danych oraz dockera

2.1. Instalacja

Na początku warto sprawdzić, czy docker nie jest już na naszym komputerze zainstalowany. Można to sprawdzić za pomocą polecenia `docker -v` (sprawdzenie wersji dockera).

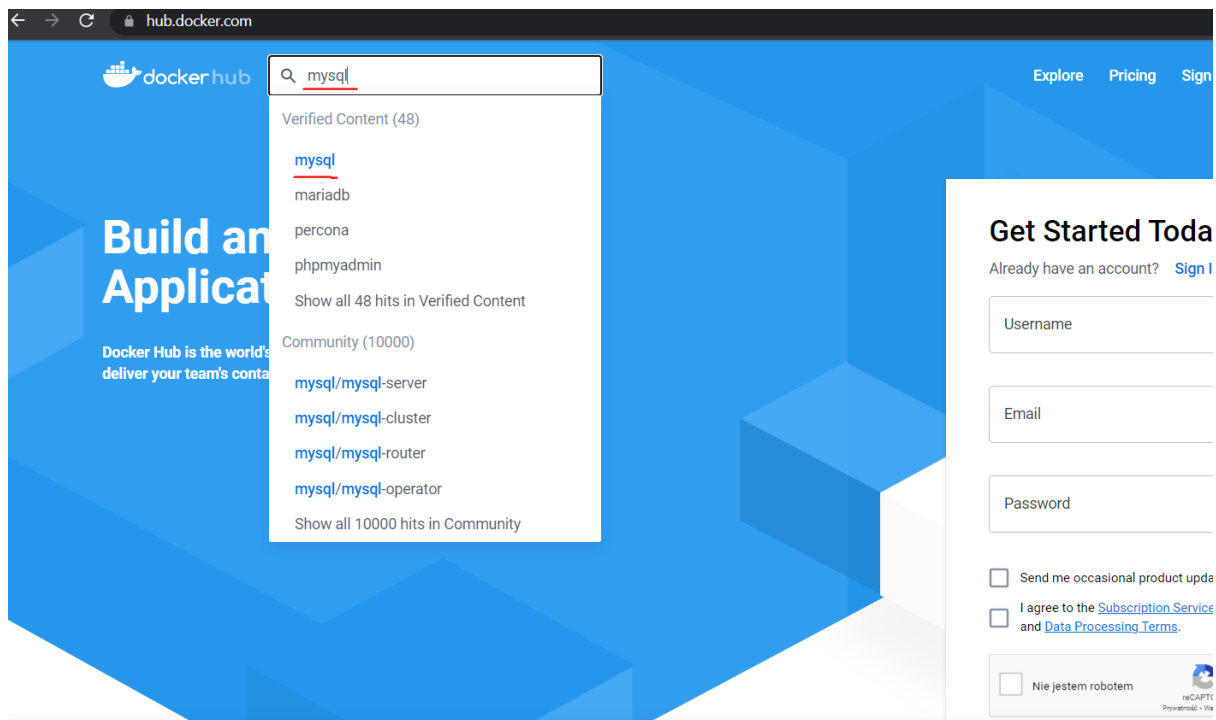


```
Windows PowerShell
PS C:\> docker -v
Docker version 20.10.17, build 100c701
PS C:\> |
```

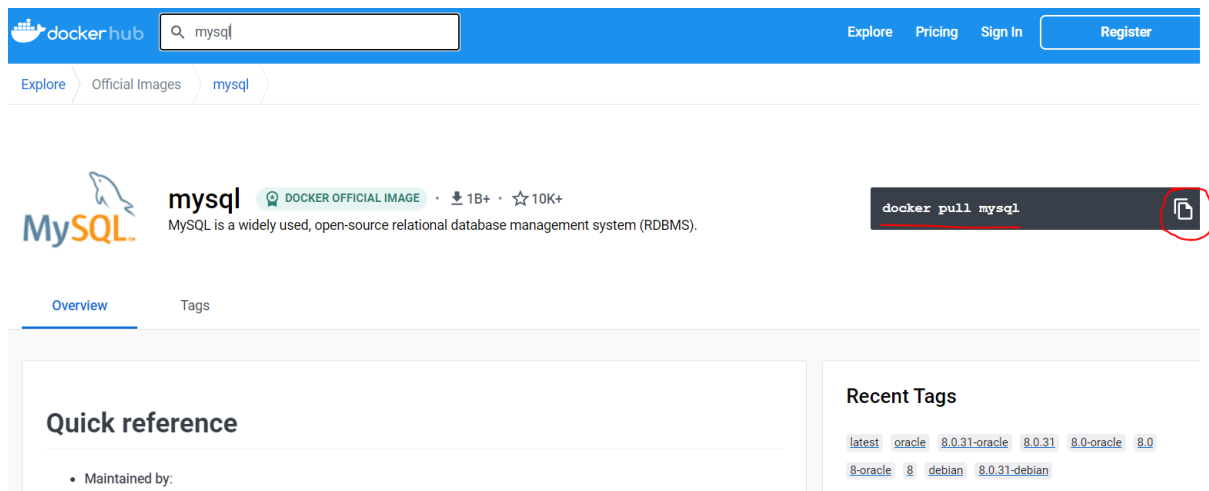
Jeśli nie posiadamy dockera, instaujemy go przykładowo z <https://docs.docker.com/engine/install/>.

2.2. MySQL Docker Hub

Wchodzimy na docker hub oraz odszukujemy obraz mysql



Kopiuujemy komendę i pobieramy obraz wklejając skopiowaną komendę do terminala.



```
PS C:\> docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql
0bb5c0c24818: Pull complete
cbb3106fbb5a: Pull complete
550536ae1d5e: Pull complete
33f98928796e: Pull complete
a341087cfff11: Pull complete
0e26ac5b33f6: Pull complete
c883b83a7112: Pull complete
873af5c876c6: Pull complete
8fe8ebd061d5: Pull complete
7ac2553cf6b4: Pull complete
ad655e218e12: Pull complete
Digest: sha256:96439dd0d8d085cd90c8001be2c9dde07b8a68b472bd20efcbe3df78cff66492
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest
PS C:\> |
```

Jeśli ktoś posiada architekturę arm64/v8 i nie może pobrać obrazu

```
Using default tag: latest
latest: Pulling from library/mysql
no matching manifest for linux/arm64/v8 in the manifest list entries
```

Może pobrać obraz w architekturze x84_64 (zalecane) za pomocą polecenia:
`docker pull --platform=linux/x86_64 mysql`

Może też pobrać obraz specjalny dla tej architektury scrollując w dół i odszukując sekcję **Quick reference**, lub klikając bezpośrednio w link: <https://hub.docker.com/r/arm64v8/mysql>.

Quick reference (cont.)

- Where to file issues:
<https://github.com/docker-library/mysql/issues>
- Supported architectures: ([more info](#))
`amd64` , `arm64v8`

2.3. Uruchomienie obrazu MySQL

Na hub.docker, z którego korzystaliśmy(hub.docker.com/_/mysql) zjeżdżając w dół mamy sekcję, która mówi nam, jak możemy używać tego obrazu.

How to use this image

Start a `mysql` server instance

Starting a MySQL instance is simple:

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

... where `some-mysql` is the name you want to assign to your container, `my-secret-pw` is the password to be set for the MySQL root user and `tag` is the tag specifying the MySQL version you want. See the list above for relevant tags.

Skorzystamy tutaj z polecenia:

```
$ docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
```

w zmienionej wersji, gdzie:

- po `--name` wprowadzamy nazwę kontenera,
- jako hasło wprowadzamy hasło podane przez nas w `application.properties`, oraz
- dodajemy dodatkowo przełącznik `-p 3306:3306`, który przekieruje port 3306 komputera na port 3306 hosta
- usuwając `:tag`

```
docker run --name mysql-movie-library -e MYSQL_ROOT_PASSWORD=movielib -d -p 3306:3306 mysql
```

* dla arm64 ponownie trzeba dodać przełącznik `--platform=linux/x86_64`, jeśli skorzystano z zalecanej metody pobierania obrazu (jeśli uruchomiono wcześniej polecenie bez tego przełącznika, trzeba usunąć powstały kontener -> `docker rm <nazwa>`):

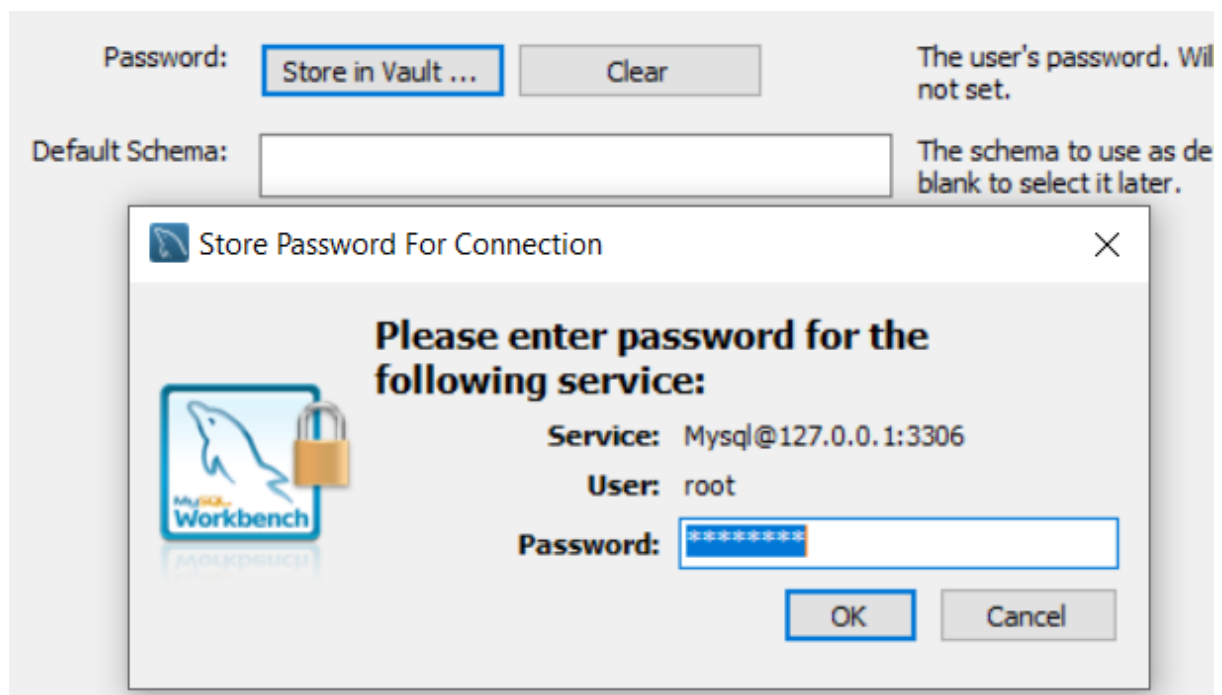
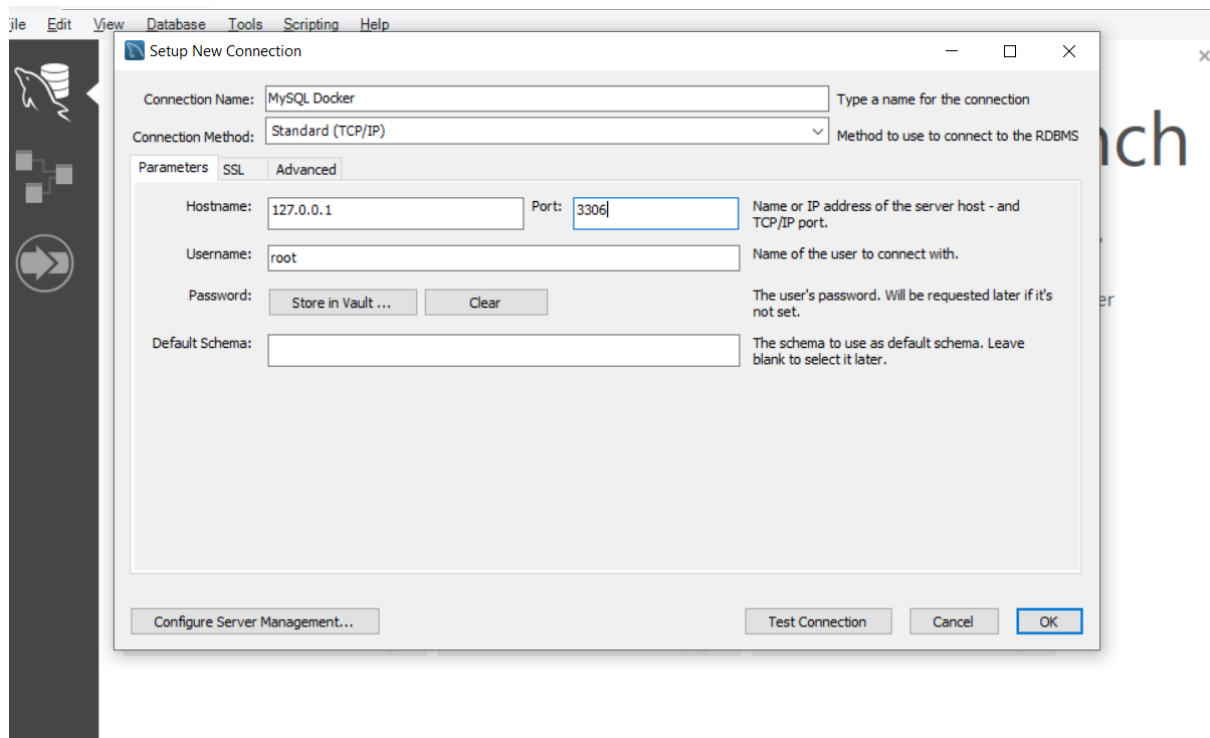
```
docker run --name mysql-movie-library -e MYSQL_ROOT_PASSWORD=movielib -d -p 3306:3306 --platform=linux/x86_64 mysql
```

** w przypadku problemu z portem 3306 można zmodyfikować fragment `-p 3306:3306`, na np. `-p 3307:3306`. Wtedy aplikacja będzie działać u nas na porcie 3307. W tym momencie musimy również zmienić port naszej bazy w `application.properties` z 3306 na 3307.

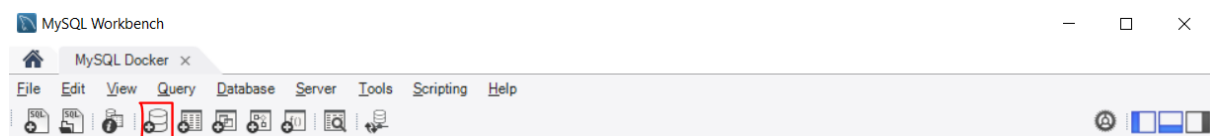
2.4. MySQL Workbench

Do pracy z bazą będzie nam potrzebne narzędzie do administrowania bazą MySQL. W tym celu idealnym narzędziem będzie MySQL Workbench, którego musimy zainstalować.

Po zainstalowaniu narzędzia uruchamiamy je oraz dodajemy nowe połączenie o konfiguracji adekwatnej do tej przedstawionej poniżej.




Teraz tworzymy nową schemę, której nazwa musi być zgodna z tą, którą podaliśmy w *application.properties*.



Query 1 movie_library - Schema x

Name: Specify the name of the schema here. You can use any combination of ANSI letters, numbers and the underscore character for names that

 Rename References Refactor model, changing all references found in view, triggers, stored procedures and functions from the old schema name to the new one.

Charset/Collation: The character set and its collation selected here will be used when no other charset/collation is set for a database object (it uses the DEFAULT)

Schema

Apply Revert

Apply SQL Script to Database

Review SQL Script

Apply SQL Script

Review the SQL Script to be Applied on the Database

Online DDL

Algorithm: Lock Type:

```

1 CREATE SCHEMA `movie_library` ;
2

```

< >

Back Apply Cancel

Następnie utworzymy nową tabelę.

Navigator

SCHEMAS


Filter objects

▼ movie_library

- Tables
- Views
- Stored Proc
- Functions

Query 1 movie_library - Schema x

Name: Specify the

 Rename References Refactor mod

Charset/Collation: The charact

Create Table... Create Table Like...

Query 1 movie_library - Schema movie - Table

Table Name: Schema: **movie_library**

Charset/Collation: Engine:

Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
title	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
director	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
rating	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Column Name: Data Type:

Charset/Collation: Default:

Comments:

Storage: ☐ Virtual ☐ Stored

☐ Primary Key ☐ Not Null ☐ Unique

☐ Binary ☐ Unsigned ☐ Zero Fill

☐ Auto Increment ☐ Generated

Columns Indexes Foreign Keys Triggers Partitioning Options

Apply SQL Script to Database

Review SQL Script

Apply SQL Script

Review the SQL Script to be Applied on the Database

Online DDL

Algorithm: Lock Type:

```

1 CREATE TABLE `movie_library`.`movie` (
2   `id` INT NOT NULL AUTO_INCREMENT,
3   `title` VARCHAR(100) NULL,
4   `director` VARCHAR(45) NULL,
5   `rating` INT NULL,
6   PRIMARY KEY (`id`));
7

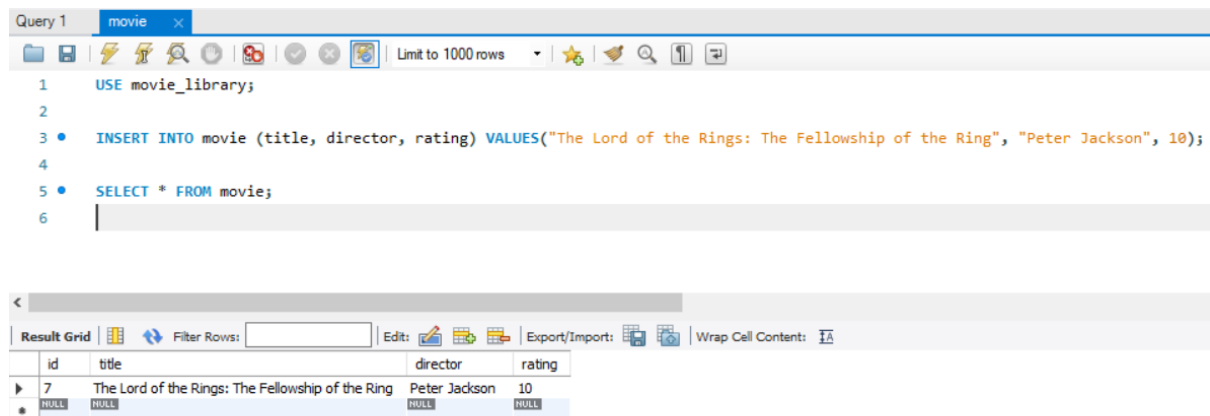
```

Następnie możemy dodać do bazy testowo jakiś rekord i sprawdzić, czy wszystko działa poprawnie.

USE movie_library;

INSERT INTO movie (title, director, rating) **VALUES**("The Lord of the Rings: The Fellowship of the Ring", "Peter Jackson", 10);

SELECT * FROM movie;



Jak widzimy - nasz MySQL Workbench działa poprawnie z naszą dockerową bazą danych.

3. CRUD z wykorzystaniem Springa

3.1. Model

3.1.1. Encja

Zanim stworzymy nasze endpointy, stworzymy sobie encję, która będzie odpowiadać naszym filmom z bazy danych. W celu zmniejszenia ilości kodu skorzystamy z Lombok'a.

```
application.properties × MovieLibraryApplication.java × Movie.java ×
1 package com.github.kboba.movieslibrary;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
8 @AllArgsConstructor
9 @NoArgsConstructor
10 public class Movie {
11     private int id;
12     private String title;
13     private String director;
14     private int rating;
15 }
```

3.1.2. Repository

Następnie tworzymy klasę, która będzie odpowiadała za komunikację z bazą danych. W klasie potrzebujemy wstrzyknąć zależność *JdbcTemplate*, z której będziemy korzystali do tworzenia zapytań.

```
application.properties × MovieLibraryApplication.java × Movie.java × MovieRepository.java ×
1 package com.github.kboba.movieslibrary;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.jdbc.core.JdbcTemplate;
5 import org.springframework.stereotype.Repository;
6
7 @Repository
8 public class MovieRepository {
9
10     @Autowired
11     JdbcTemplate jdbcTemplate;
12
13 }
```

3.2. Read

3.2.1. getAll()

Na początku zajmiemy się endpoint'em, który będzie wyświetlał wszystkie filmy, które zostały dodane do bazy. W tym celu, w naszym repozytorium odpowiedzialnym za komunikację z bazą danych, tworzymy metodę, która pobiera filmy z bazy danych, mapuje je do obiektów i zwraca w liście.

```
@Repository
public class MovieRepository {

    1 usage
    @Autowired
    JdbcTemplate jdbcTemplate;

    public List<Movie> getAll(){
        return jdbcTemplate.query( sql: "SELECT id, title, director, rating FROM movie",
            BeanPropertyRowMapper.newInstance(Movie.class));
    }
}
```

Następnie musimy stworzyć endpoint, pod którym będziemy mieli nasze filmy. Endpointy tworzymy w Controlle'rze z adnotacją `@GetMapping()`, w której jako parametr podajemy adres endpoint'a. Nasz *MovieController* będzie korzystał z naszego napisanego wcześniej *MovieRepository*, a w wypadku zwracania filmów konkretnie z metody *getAll()*. Aby z niej skorzystać wstrzykujemy zależność - nasze repozytorium.

```
@RestController
public class MovieController {

    1 usage
    @Autowired
    MovieRepository movieRepository;

    @GetMapping("/movies")
    public List<Movie> getAll() {
        return movieRepository.getAll();
    }
}
```

Jak widzimy dany endpoint zwraca nam listę filmów (w tym wypadku jest to lista jednoelementowa ze względu na naszą bazę danych).

```
← → ↻ ⓘ localhost:8080/movies
[{"id":1,"title":"The Lord of the Rings: The Fellowship of the Ring","director":"Peter Jackson","rating":10}]
```

3.2.2. getById()

Następnym endpointem będzie endpoint bardzo podobny, z tą małą różnicą, że będzie zwracał jeden film. W związku z tym analogicznie tworzymy metodę w naszym *MovieRepository*, które tym razem wyciągnie nam jeden film o konkretnym *id* z bazy danych. W tym celu tworzymy metodę *getById(int id)* przyjmującą jako argument *id* szukanego filmu. Do wyciągnięcia filmu wykorzystamy metodę *queryForObject()*, która oprócz zapytania i Mappera jako argument przyjmuje *id*, które jest wstawiane w miejsce “?” z naszego zapytania.

```
public Movie getById(int id){
    return jdbcTemplate.queryForObject(
        sql: "SELECT id, title, director, rating FROM movie WHERE id = ?",
        BeanPropertyRowMapper.newInstance(Movie.class), id);
}
```

Również analogicznie do poprzedniego przykładu tworzymy endpoint, który znajdować się będzie w *MovieController*. Aby przekazywać *id* z naszego endpoint'a skorzystamy tutaj z adnotacji *@PathVariable* przy naszym argumencie. W ciele metody ponownie mamy proste wywołanie metody z naszego repozytorium.

```
@GetMapping("/movies/{id}")
public Movie getById(@PathVariable("id") int id) {
    return movieRepository.getById(id);
}
```

Jak można zauważyć po ponownym uruchomieniu aplikacji, metoda działa prawidłowo. Mogłoby się wydawać, że w obecnym wypadku nie ma różnicy przy obu odpowiedziach, natomiast różnica ta mieści się w nawiasach kwadratowych, których w obecnej odpowiedzi nie ma. Jest tak ze względu na to, że nie mamy zwracanej listy obiektów, a jeden obiekt.

```
← → ↻ ⓘ localhost:8080/movies/1
{"id":1,"title":"The Lord of the Rings: The Fellowship of the Ring","director":"Peter Jackson","rating":10}
```

3.3. Create

Następną funkcjonalnością będzie dodawanie filmów do naszej bazy danych. W tym celu musimy ponownie stworzyć metodę w *MovieRepository*, oraz *MovieController*. Metoda w repository będzie działała bardzo prosto, będzie przyjmowała jako argument listę filmów, którą doda do naszej bazy danych. Metoda ta będzie również informowała nas o powodzeniu w bardzo prosty sposób - jeśli operacja się powiedzie i nie zostanie zwrócony żaden błąd - metoda zwróci "1".

```
public int add(List<Movie> movies) {  
    movies.forEach(movie -> jdbcTemplate.update(  
        sql: "INSERT INTO movie(title, director, rating) VALUES(?, ?, ?) ",  
        movie.getTitle(), movie.getDirector(), movie.getRating()  
    ));  
  
    return 1;  
}
```

Tym razem zostanie wykorzystane metoda POST z racji, że dodajemy coś do naszej bazy danych. Metoda posiada adnotację *@RequestBody*, ze względu na to, że obiekty muszą skądś przyjść - z ciała zapytania HTTP, które będzie w formacie JSON. Obiekty są automatycznie mapowane z na listę filmów.

```
@PostMapping("/movies")  
public int add (@RequestBody List<Movie> movies) {  
    return movieRepository.add(movies);  
}
```

Do testowania metody POST nie wystarczy przeglądarka. Potrzebne jest tutaj narzędzie to wysyłania zapytań. Idealnym narzędziem będzie tutaj Postman. Jak wspomniano wcześniej, musimy za pomocą metody POST - na odpowiedni endpoint - wysłać zapytanie z ciałem formatu JSON. Przykładowe zapytanie znajduje się poniżej. Jeśli w odpowiedzi dostaniemy cyfrę "1". Oznacza to, że dodawanie do bazy danych się powiodło, co można łatwo sprawdzić za pomocą przeglądarki, lub MySQL Workbench

POST localhost:8080/movies Send

Body JSON Beautify

```

1 [
2   {
3     "title": "The Lord of the Rings: The Two
4       Towers",
5     "director": "Peter Jackson",
6     "rating": 10
7   },
8   {
9     "title": "The Lord of the Rings: Return of
10      the King",
11     "director": "Peter Jackson",
12     "rating": 10
13   }
14 ]

```

Body 200 OK 39 ms 165 B Save Response

Pretty Raw Preview Visualize JSON

Limit to 1000 rows

```

1 • USE movie_library;
2
3 • SELECT * FROM movie;
4

```

Result Grid Filter Rows: Edit Export/Import: Wrap Cell

	id	title	director	rating
▶	1	The Lord of the Rings: The Fellowship of the Ring	Peter Jackson	10
	4	The Lord of the Rings: The Two Towers	Peter Jackson	10
	5	The Lord of the Rings: Return of the King	Peter Jackson	10
*	NULL	NULL	NULL	NULL

3.4. Update

Przedostatnią z funkcjonalności będzie możliwość aktualizacji naszych filmów. W tym celu po raz kolejny tworzymy metodę w *MovieRepository*, oraz dwie metody w *MovieController* (jedna do całkowitej aktualizacji - metoda PUT, oraz druga do aktualizacji częściowej - metoda PATCH). Obie metody z *MovieController* będą wykorzystywać tę samą metodę z *MovieRepository*, ze względu na to, że filmy są w ten sam sposób aktualizowane w bazie danych.

Ponownie metody w Repository będą uproszczone i będą z tego powodu zwracać "1" jeśli operacja aktualizacji danych przejdzie pomyślnie. Informacja znowu będzie przekazywane przez metodę z naszego Controller'a.


```
public int update(Movie movie) {
    return jdbcTemplate.update( sql: "UPDATE movie SET title=?, director=? rating=?, WHERE id=?",
        movie.getTitle(), movie.getDirector(), movie.getRating(), movie.getId());
}
```

Do testowania tej funkcjonalności ponownie nie wystarczy nasza przeglądarka. Będzie nam potrzebny przykładowo Postman, który pozwala nam wysyłać zapytania PUT oraz PATCH, wykorzystane do stworzenia możliwości aktualizacji naszych filmów w bazie danych.

3.4.1. update()

Zwykła aktualizacja wykorzysta wspomnianą powyżej metodę PUT, stąd skorzystamy z adnotacji *@PutMapping()*. Metoda ta działa w sposób aktualizowania kompletnego. Jeżeli nie podamy jakiejś wartości w ciele naszego zapytania, przykładowo *rating* (int), to wartość tego pola zostanie ustawiona na domyślną, czyli "0".

Metoda ta również posiada argument *@RequestBody*, które posiadało dodawanie filmów do bazy, ze względu na to, że aby film zaktualizować, musimy przesłać dane do aktualizacji. Oprócz wymaganego ciała - ponownie występuje ścieżka zakończona "{id}". Związane jest to z tym, że tak jak przy pobieraniu konkretnego filmu będziemy w tym momencie chcieli zaktualizować konkretny film, do którego odwołujemy się za pomocą id.

Do pobrania filmu, który zostanie zaktualizowany oraz później uaktualniony w bazie danych wykorzystamy napisaną wcześniej metodę w naszym repozytorium (*getById(int id)*).

```
@PutMapping("/movies/{id}")
public int update(@PathVariable("id") int id, @RequestBody Movie updatedMovie) {
    Movie movie = movieRepository.getById(id);

    if(movie != null) {
        movie.setTitle(updatedMovie.getTitle());
        movie.setDirector(updatedMovie.getDirector());
        movie.setRating(updatedMovie.getRating());
        return movieRepository.update(movie);
    } else
        return -1;
}
```

3.4.2. partialUpdate()

Do aktualizacji częściowej naszego filmu w bazie danych wykorzystana zostanie metoda PATCH, z której skorzystać możemy dzięki adnotacji *@PatchMapping()*. W przypadku tej metody wartości pól, których nie ma w ciele zapytania http nie są ustawiane na domyślne, nie są w ogóle ustawiane.

Ponownie jako argumenty przyjmowane jest id szukanego filmu oraz ciało zapytania. W ciele napisanej metody widzimy, że pola naszego filmu, który będzie aktualizowany będą zmieniane tylko w wypadku, gdy nie jako String nie mamy null (domyślna wartość obiektów,

to "null"), oraz gdzie mamy wartość naszego ratingu większą od 0 (domyślna wartość int, to "0")

```
@PatchMapping("/movies/{id}")
public int partiallyUpdate(@PathVariable("id") int id, @RequestBody Movie updatedMovie) {
    Movie movie = movieRepository.getById(id);

    if(movie != null) {
        if(updatedMovie.getTitle() != null) movie.setTitle(updatedMovie.getTitle());
        if(updatedMovie.getDirector() != null) movie.setDirector(updatedMovie.getDirector());
        if(updatedMovie.getRating() > 0) movie.setRating(updatedMovie.getRating());
        return movieRepository.update(movie);
    } else
        return -1;
}
```

3.5. Delete

Ostatnią funkcjonalnością jest usuwanie filmu z bazy danych. W tym celu jak zawsze napiszemy metodę w *MovieRepository* oraz metodę w *MovieController*. Metoda ta będzie jedynie wywoływała prosty skrypt SQL za pomocą naszego obiektu *JdbcTemplate*, z którego korzystamy przy komunikowaniu się z bazą danych. Po raz kolejny przy powodzeniu wykonania operacji zwrócimy wartość "1".

```
public int delete(int id) {
    return jdbcTemplate.update("DELETE FROM movie WHERE id=?", id);
}
```

Jako metoda http posłuży nam tym razem DELETE. Do stworzenia endpoint'u z tą metodą wykorzystamy adnotację *@DeleteMapping()*. Ponownie ścieżka będzie zakończona "{id}" ze względu na chęć usunięcia konkretnego filmu. Tym razem nie potrzebujemy żadnego ciała metody. Potrzebujemy jedynie odpowiedniego "id".

```
@DeleteMapping("/movies/{id}")
public int delete(@PathVariable("id") int id) {
    return movieRepository.delete(id);
}
```

4. Angular - frontend

4.1. Konfiguracja

Pracę z frameworkiem Angular należy poprzedzić instalacją Node.js wraz z node package manager - dostępny do pobrania pod [Downloading and installing Node.js and npm | npm Docs \(npmjs.com\)](https://docs.npmjs.com).

Po poprawnej instalacji Instalujemy Angular CLI z użyciem Node Package Managera:

```
npm install -g @angular/cli
```

Instrukcja została przygotowana w oparciu o Angular CLI w wersji 15.0.0.

4.2. Tworzenie projektu

Do stworzenia nowego projektu należy wykorzystać komendę

```
ng new nazwa-projektu
```

W razie błędu z poleceniem ng, wynikającego z ustawień skryptów w systemie Windows, można zmienić te ustawienia poleceniem

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

W trakcie tworzenia projektu zostaniemy zapytani o przesyłanie danych statystycznych do Google (dowolna odpowiedź), włączenie Angular routingu (wybieramy tak) oraz format arkuszy stylu (wybieramy CSS).

Po poprawnym wygenerowaniu możemy zobaczyć domyślną zawartość projektu pod adresem <http://localhost:4200> - najpierw musimy jednak uruchomić projekt poleceniem

```
ng serve
```

wykonanym w nowo wygenerowanym katalogu (o nazwie zgodnej z nazwą projektu - dla powyższego przykładowego polecenia będzie to nazwa-projektu). Polecenie ng serve (lub ng s) jest przydatne przy developmencie - poza początkową kompilacją projektu dokonuje ono również automatycznej rekompilacji po zapisaniu zmian w plikach projektu.

Zainstalujemy również framework bootstrap, zawierający gotowe do użycia style CSS, używając polecenia

```
npm install bootstrap
```

Żeby zacząć korzystać ze stylów bootstrapa, konieczne jest zaimportowanie ich - można to zrobić m. in. dyrektywą import w pliku src/styles.css:

```
@import "~bootstrap/dist/css/bootstrap.css"
```

4.3. Stworzenie nowej strony

W celu stworzenia strony głównej (czyli tak naprawdę komponentu `HomePageComponent`) w katalogu `pages`, na której wyświetlana będzie lista filmów, należy wykonać polecenie

```
ng generate component pages/home-page
```

W efekcie w katalogu `src/app/pages` pojawi się folder `home-page`, zawierający 4 pliki z rozszerzeniami `html`, `css`, `ts` oraz `spec.ts`. W tej instrukcji pominięty zostanie plik `spec.ts`, służący do testów komponentu z wykorzystaniem frameworka `Jasmine` i narzędzia `Karma`.

Polecenie `ng generate` (lub `ng g`), poza stworzeniem potrzebnych plików, doda również nowo utworzony komponent `HomePageComponent` do sekcji `declarations[]` pliku `app.module.ts`.

Po wygenerowaniu nowej strony należy również podmienić zawartość domyślną projektu. W tym celu należy usunąć całą zawartość `app.component.html`, a w jego miejsce wstawić znacznik `<app-home-page></app-home-page>` - osadzi on nasz nowy komponent na stronie głównej.

Teraz strona <http://localhost:4200> powinna wyświetlić tekst "home-page works!".

4.4. Stworzenie nowego serwisu

W celu wysyłania zapytań do backendu stworzymy również serwis `movies`, ponownie wykorzystując polecenie `ng generate`:

```
ng generate service services/movies
```

Jako pierwszy zaimplementujemy funkcję korzystającą z endpointu `GET /movies`. Podstawowy adres URL naszego API będziemy przechowywać w zmiennej środowiskowej `baseUrl` - w tym celu należy dodać do projektu plik `src/environments/environment.ts`:

```
export const environment = {  
  production: false,  
  baseUrl: 'http://localhost:8080/'  
};
```

Zdefiniujemy również model filmu w klasie `Movie`, zgodny z modelem zwracanym przez backend - nie jest to konieczne, gdyż moduł klienta `http` może również odbierać obiekty dowolnego typu poprzez wykorzystanie słowa kluczowego *any*, ale ułatwia pracę na odebranych danych. Model:

```
export class Movie{  
  id?: number | null;  
  title?: string;  
  director?: string;  
  rating?: number;  
}
```

Musimy również uzupełnić `app.module.ts` o import modułu klienta `http`, a następnie wstrzyknąć go do naszego serwisu. Moduł importujemy poprzez dodanie w `app.module.ts`

linii `import { HttpClientModule } from '@angular/common/http';` w sekcji importów oraz `HttpClientModule` do tablicy `imports[]`.

Wstrzykiwanie zależności w Angularze można wykonać przez konstruktor - w naszym wypadku w liście parametrów konstruktora serwisu wystarczy dodać prywatnego klienta http:

```
constructor(private http: HttpClient) { }
```

Następnie, wykorzystując wstrzykniętego klienta, pobierzemy listę filmów zwracaną przez API:

```
public getMovies(): Observable<Movie[]> {  
    return this.http.get<Movie[]>(this.moviesUrl);  
}
```

W tym wypadku `this.moviesUrl` to adres składający się z `environment.baseUrl` oraz sufiksu ścieżki endpointa `/movies`.

Powyższe polecenie zwróci obiekt typu `Observable<Movie[]>` (wymaga importu pakietu `rxjs`). Obiekt typu `Observable<T>` pozwala w tym wypadku na obsługę asynchronicznego zapytania do backendu naszej aplikacji. Wykonanie na nim metody `subscribe()` pozwoli na oczekiwanie na odpowiedź ze strony backendu, oraz obsłużenie potencjalnego błędu.

4.5. Pobranie i wyświetlenie listy filmów

W celu wykorzystania stworzonego serwisu do pobrania listy filmów należy go wstrzyknąć do komponentu `home-page` z wykorzystaniem konstruktora:

```
public constructor(movieService: MoviesService) {}
```

Kolejnym krokiem będzie pobranie listy filmów:

```
private getMovies() {  
    this.movieService.getMovies().subscribe({  
        next: (res: any) => {  
            this.movies = res;  
        },  
        error: (error: any) => {  
            console.log(error);  
            alert("Error - could not GET movies");  
        }  
    })  
}
```

Pierwsza "arrow function" obsługuje rezultat zapytania zakończony sukcesem - przypisuje otrzymaną listę obiektów typu `Movie` do pola komponentu. Druga opisuje reakcję serwisu na zapytanie kończące się niepowodzeniem - zapisanie błędu do konsoli oraz wyświetlenie alertu.

Aby przy wczytaniu komponentu home-page została również wykonana powyższa metoda, klasa HomeComponent musi implementować OnInit - metodę getMovies() należy następnie wywołać w metodzie ngOnInit, wykonywanej przy inicjalizacji komponentu.

Do wyświetlenia danych wykorzystamy jeden z gotowych stylów tabel Bootstrapa - [Tables · Bootstrap \(getbootstrap.com\)](https://getbootstrap.com/docs/4.0/components/tables/). Aby wyświetlić w kolejnych rzędach pobrane filmy, wykorzystamy konstrukcję ngFor, iterującą po elementach kolekcji:

```
<tr *ngFor="let movie of movies">
  <td>{{movie.id}}</td>
  <td>{{movie.title}}</td>
  <td>{{movie.director}}</td>
  <td>{{movie.rating}}</td>
</tr>
```

Po dodaniu tytułu (wycentrowanie oraz marginesy można uzyskać z wykorzystaniem `class="d-flex justify-content-center m-3"` dzięki bootstrapowi) podstawowa strona powinna prezentować się w następujący sposób:

Movie list			
Id	Title	Director	Rating
1	The Lord of the Rings: The Fellowship of the Ring	Peter Jackson	10
2	The Lord of the Rings: The Two Towers	Peter Jackson	10
3	The Lord of the Rings: Return of the King	Peter Jackson	10
4	Batman	Tim Burton	8
5	Superman	Richard Donner	7
6	Last Samurai	Edward Zwick	8

4.6. Dodawanie filmów do bazy

W celu dodania filmu do bazy konieczne będzie stworzenie kolejnej metody w naszym serwisie.

Na początek przygotujemy metodę, która przyjmuje jako parametr listę obiektów typu Movie[] i wykonuje akcję POST:

```
public addMovie(movies: Movie[]): Observable<any> {
  let headers = { 'content-type': 'application/json' };
  return this.http.post(this.moviesUrl, JSON.stringify(movies), {
    headers: headers,
  });
}
```

Metoda JSON.stringify() konwertuje obiekt do jego opisu w formacie JSON.

Do tworzenia filmów wykorzystamy okno typu modal. W tym celu należy wygenerować nowy komponent z wykorzystaniem polecenia `ng g component components/movie-modal`. Będziemy również potrzebować pakietu ngx-bootstrap, instalowanego poleceniem `npm i ngx-bootstrap`.

W następnej kolejności dodamy do app.module.ts potrzebne komponenty: do tablicy `imports[]` należy dodać `FormsModule`, a do `providers[]` - `BsModalService`.

W nowo wygenerowanym komponencie jako pierwszy dodamy element @Input, pozwalający na przepływ danych od rodzica (w naszym przypadku home page):

```
@Input() modalRef!: BsModalRef;
```

Przygotujmy również pole przechowujące model obiektu typu movie oraz (póki co) pustą metodę addMovie:

```
public movie: Movie = {title: '', director: '', rating: 0};
```

Następnie należy zdefiniować strukturę HTML naszego okna tworzenia filmów:

```
<div class="modal-header">
  <h4 class="modal-title pull-left theme-text-color">Add movie</h4>
  <button type="button" class="btn-close close pull-right"
aria-label="Close" (click)="modalRef.hide()">
    <span aria-hidden="true" class="visually-hidden">&times;</span>
  </button>
</div>
<div class="modal-body">
  <div class="form-floating mb-3 mt-3">
    <input value='{{movie.title}}' type="text"
[(ngModel)]='movie.title' class="form-control">
    <label for="title">Title</label>
  </div>
  <div class="form-floating mb-3 mt-3">
    <input value='{{movie.director}}' type="text"
[(ngModel)]='movie.director' class="form-control">
    <label for="director">Director</label>
  </div>
  <div class="form-floating mb-3 mt-3">
    <input value='{{movie.rating}}' type="number"
[(ngModel)]='movie.rating' class="form-control">
    <label for="rating">Rating</label>
  </div>
  <button type="submit" class="btn btn-dark float-end"
(click)="addMovie()">Add</button>
</div>
```

Obsługa przycisku zamykania okna jest wykonywana przez instrukcję (click)="modalRef.hide()". Przycisk Add wywołuje metodę addMovie(), której implementacją zajmiemy się w następnej kolejności. Do tego celu musimy jednak wstrzyknąć, analogicznie jak dla HomePageComponent, serwis MoviesService.

Implementacja metody AddMovie będzie wymagała przekazania do serwisu tablicy jednoelementowej. Analogicznie jak wcześniej w metodzie GET, obsłużymy 2 ścieżki - film dodany poprawnie skutkuje zamknięciem okna, w razie błędu logujemy jego treść w konsoli oraz wyświetlamy alert.

```
public addMovie()
{
  let movies = [this.movie];
```

```

    this.moviesService.addMovie(movies).subscribe({
      next: (res: any) => {
        this.modalRef.hide();
      },
      error: (error: any) => {
        console.log(error);
        alert("Movie creation failed");
      }
    })
  }
}

```

Musimy teraz obsłużyć nowy komponent w 'rodzicu' - HomePageComponent. W tym celu należy dodać pole typu BsModalRef w rodzicu, a następnie osadzić nowy komponent w HTML:

```

<ng-template #template>
  <app-movie-modal class="theme-bg-1 "
    [modalRef]="modalRef!"></app-movie-modal>
</ng-template>

```

Konieczne będzie również wstrzyknięcie serwisu BsModalService - z jego wykorzystaniem otworzymy okno tworzenia filmu w nowej metodzie.

```

public openMovieModal(template: TemplateRef<any>) {
  this.modalRef = this.modalService.show(template);
}

```

Teraz wystarczy dodać w HTMLu przycisk tworzenia filmu:

```

<button type="button" class="btn btn-primary"
  (click)="openMovieModal(template)">Add movie</button>

```

Po dodaniu testowego filmu można zauważyć, że lista nie zostaje odświeżona - konieczne jest odświeżenie strony. Można temu zaradzić wykorzystując konstrukcję @Output(), pozwalającą na przepływ informacji od dziecka do rodzica. W komponencie MovieModal należy dodać linijkę:

```

@Output() reload = new EventEmitter<boolean>();

```

I w razie sukcesu, po dodaniu filmu wywołać `this.reload.emit(true);`. W komponencie HomePage należy natomiast w pliku .html zaktualizować linię:

```

<app-movie-modal class="theme-bg-1 " [modalRef]="modalRef!"
  (reload)="getMovies()"></app-movie-modal>

```

oraz zmienić modyfikator dostępu funkcji getMovies() na public. Teraz już po dodaniu nowego filmu, lista powinna odświeżać się automatycznie.

4.7. Usuwanie

Tak, jak wcześniej, zaczniemy od dodania metody w serwisie. URL skonstruujemy, wykorzystując operatory interpolacji stringów:

```
public deleteMovie(id: number): Observable<any> {  
  return this.http.delete(`${this.moviesUrl}/${id}`)  
}
```

Następnie przygotujemy metodę w HomeComponent, wywołującą powyższą metodę serwisu:

```
public deleteMovie(id: number) {  
  this.movieService.deleteMovie(id).subscribe({  
    next: (res: any) => {  
      this.getMovies();  
    },  
    error: (error: any) => {  
      console.log(error);  
      alert("Error while deleting the entry");  
    }  
  })  
}
```

Pozostało jedynie wywołanie tej funkcji spod przycisku w HTML. W tym celu dodamy dodatkową kolumnę, bez nazwy, w której dla każdego rekordu znajdzie się przycisk "Delete". Ponownie wykorzystamy dla przycisku gotowe style bootstrapa:

```
<table class="table table-hover">  
  <thead>  
    <tr>  
      <th scope="col">Id</th>  
      <th scope="col">Title</th>  
      <th scope="col">Director</th>  
      <th scope="col">Rating</th>  
      <th scope="col"></th>  
    </tr>  
  </thead>  
  <tbody>  
    <tr *ngFor="let movie of movies">  
      <td>{{movie.id}}</td>  
      <td>{{movie.title}}</td>  
      <td>{{movie.director}}</td>  
      <td>{{movie.rating}}</td>  
      <td><button type="button" class="btn btn-danger"  
(click)="deleteMovie(movie.id!)">Delete</button></td>  
    </tr>  
  </tbody>
```

</table>

Po kliknięciu przycisku Delete rekord powinien zostać usunięty z bazy, a lista zostanie automatycznie odświeżona.

4.8. Edycja

Analogicznie jak w poprzednich punktach, zaczniemy od dodania metody w serwisie:

```
public updateMovie(movie: Movie): Observable<any> {  
  let headers = { 'content-type': 'application/json' };  
  return this.http.patch(`${this.moviesUrl}/${movie.id}`,  
    JSON.stringify(movie), {  
      headers: headers,  
    });  
}
```

Do edycji rekordów wykorzystamy stworzony wcześniej komponent MovieModal. W tym celu będziemy musieli uzupełnić go o dodatkowy @Input(), pozwalający na przekazywanie danych o edytowanym filmie z rodzica:

```
@Input() movie!: Movie;
```

Inicjalizację pola movie przeniesiemy do HomePageComponent:

```
public movie: Movie = {id: null, title: '', director: '', rating: 0};
```

Zaktualizujemy również tytuł modala oraz tekst na przycisku - zamiast “Add” będzie to “Add/Update”. Ustawienie wartości początkowych pól formularza zostało już obsłużone przy tworzeniu nowego rekordu przez value="{{movie.xyz}}", gdzie xyz to nazwa danego pola.

Przycisk Add/Update będzie teraz wywoływał nową metodę addOrUpdateMovie(), która w zależności od wartości movie.id wywoła odpowiednią metodę serwisu:

```
public addOrUpdateMovie() {  
  if (this.movie.id != null && this.movie.id != undefined) {  
    this.updateMovie();  
  }  
  else {  
    this.addMovie();  
  }  
}
```

przy czym musimy zaimplementować również nową metodę updateMovie(), analogiczną do addMovie():

```
public updateMovie() {  
  this.moviesService.updateMovie(this.movie).subscribe({  
    next: (res: any) => {  
      this.modalRef.hide();  
      this.reload.emit(true);  
    },  
  },
```

```

    error: (error: any) => {
      console.log(error);
      alert('Movie update failed');
    },
  });
}

```

Pozostało obsłużenie zmian w komponencie rodzica - w pierwszej kolejności konieczne jest przekazanie wartości do pola Input:

```

<app-movie-modal class="theme-bg-1 " [modalRef]="modalRef!"
[movie]="movie" (reload)="getMovies()"></app-movie-modal>

```

Nie należy zapominać o reinicjalizacji wartości pola movie w rodzicu na początku metody openMovieModal() - w przeciwnym razie po kliknięciu przycisku Add będziemy edytować ostatnio edytowany element.

Do wywołania okna edycji stworzymy nową metodę:

```

public updateMovie(template: TemplateRef<any>, movie: Movie) {
  this.movie = movie;
  this.modalRef = this.modalService.show(template);
}

```

Pozostało jedynie dodanie nowego przycisku pozwalającego na edycję we wcześniej stworzonej kolumnie - ponownie wykorzystamy style bootstrapa:

```

<td>
  <button type="button" class="btn btn-warning mx-1"
(click)="updateMovie(template, movie)">Edit</button>
  <button type="button" class="btn btn-danger mx-1"
(click)="deleteMovie(movie.id!)">Delete</button>
</td>

```

Również po edycji filmu lista zostanie odświeżona.