



Zaawansowane technologie internetowe  
Dokumentacja projektu

System rekomendacji restauracji

Patryk Śledź

303806

Informatyka Stosowana

Wydział Fizyki i Informatyki Stosowanej

Akademia Górniczo-Hutnicza w Krakowie

## Spis treści

1. Cel projektu .....	3
2. Ogólne założenia funkcjonalne.....	3
3. Założenia dotyczące części serwerowej i klienta .....	3
3.1. Część serwerowa .....	3
3.2. Część klienta .....	3
4. Projekt bazy danych .....	4
Zdefiniowane typy wierzchołków: .....	4
Zdefiniowane relacje między wierzchołkami: .....	4
Pola zawarte w wierzchołkach i relacjach: .....	5
5. Aplikacja serwerowa .....	6
Przykładowy klasa kontrolera.....	7
Przykładowa klasa serwisu .....	8
Przykładowa klasa modelu .....	9
Przykładowy interfejs dostępu do bazy danych.....	10
Dokumentacja REST API .....	11
6. Aplikacja klienta .....	12
Zarządzanie stanem aplikacji.....	13
Omówienie udostępnionej funkcjonalności .....	14
7. Przewodnik uruchomienia aplikacji .....	21
Szybka metoda uruchomienia aplikacji .....	21
Dłuższa metoda uruchomienia aplikacji .....	22
8. Podsumowanie .....	23

## 1. Cel projektu

Celem projektu jest stworzenie serwera aplikacyjnego w technologii Java Spring wraz z warstwą dostępu do danych w postaci Spring Data JPA oraz interfejsu użytkownika będącego stroną internetową typu SPA. Zadaniem zaprojektowanego serwisu jest rekomendacja restauracji na podstawie ich podobieństwa oraz wykorzystując preferencje innych użytkowników.

## 2. Ogólne założenia funkcjonalne

- Dostęp do serwisu będzie zabezpieczony (autoryzacja tokenowa).
- Użytkownik powinien mieć możliwość rejestracji konta oraz edycji swoich danych.
- Użytkownik powinien mieć możliwość przeglądania proponowanych restauracji:
  - w pobliżu wybranej miejscowości,
  - w obrębie całego kraju,
  - na podstawie dotychczasowych polubień oraz preferencji innych użytkowników.
- Oprócz zdefiniowanych wcześniej restauracji, powinna być możliwość dodawania nowych.
- Użytkownik powinien mieć możliwość oceniania, komentowania oraz obserwowania wybranych restauracji.
- Użytkownicy powinni widzieć komentarze, oceny innych użytkowników.

## 3. Założenia dotyczące części serwerowej i klienta

### 3.1. Część serwerowa

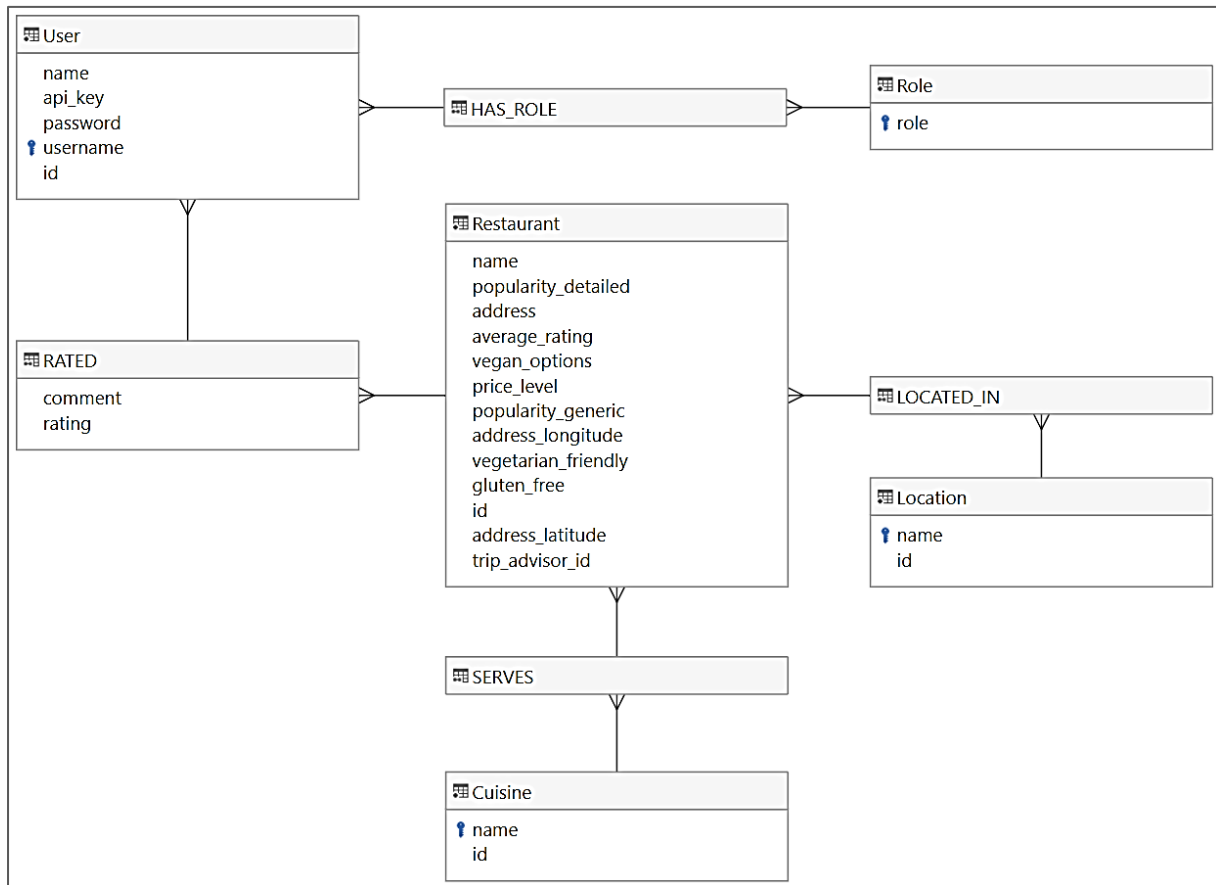
- **Wybrana technologia serwera:** biblioteka Spring wraz z narzędziem Spring-Boot.
- **Proponowana baza danych:** nierelacyjna baza grafowa – Neo4j.
- **Wybrana technologia klient-serwer:** Rest API.

### 3.2. Część klienta

- **Wybrana technologia:** aplikacja typu SPA, biblioteka React.js.

## 4. Projekt bazy danych

Główną funkcjonalnością projektu jest system rekomendacji. W związku z tym jako bazę danych wybrano bazę grafową Neo4j. Diagram UML przygotowanego schematu został przedstawiony na ?.



Zdefiniowane typy wierzchołków:

- Restaurant – reprezentuje restauracje
- Location – reprezentuje lokalizacje restauracji
- Cuisine – reprezentuje typy kulinarne restauracji
- User – reprezentuje użytkowników
- Role – reprezentuje role użytkowników

Zdefiniowane relacje między wierzchołkami:

- SERVES – krawędź skierowana, łącząca restaurację (Restaurant) i typy kulinarne (Cuisine)
- LOCATED\_IN – krawędź skierowana, łącząca restaurację (Restaurant) i lokalizację (Location)
- RATED – krawędź skierowana, łącząca restaurację (Restaurant) i użytkownika (User)
- HAS\_ROLE – krawędź skierowana, łącząca użytkownika (User) z przydzielonymi rolami (Role)

Pola zawarte w wierzchołkach i relacjach:

- **Restaurant**

Nazwa	Typ	Opis
id	integer	Domyślny identyfikator restauracji
name	string	Nazwa restauracji
address	string	Szczegółowy adres restauracji
address_latitude	double	Szerokość geograficzna
address_longitude	double	Długość geograficzna
vegan_options	boolean	Pole sygnalizujące posiadanie pozycji przeznaczonych dla wegan
vegetarian_friendly	boolean	Pole sygnalizujące posiadanie pozycji przeznaczonych dla vegetarian
gluten_free	boolean	Pole sygnalizujące posiadanie pozycji przeznaczonych dla osób nietolerujących gluten
popularity_detailed	string	Szczegółowe informacje na temat restauracji
popularity_generic	string	Skrócone informacje na temat restauracji
trip_advisor_id	string	Unikalny identyfikator, wykorzystywany podczas importu danych

- **Location**

Nazwa	Typ	Opis
id	integer	Domyślny identyfikator lokalizacji
name	string	Unikalna nazwa miasta

- **Cuisine**

Nazwa	Typ	Opis
id	integer	Domyślny identyfikator typu kulinarnego
name	string	Nazwa typu kulinarnego

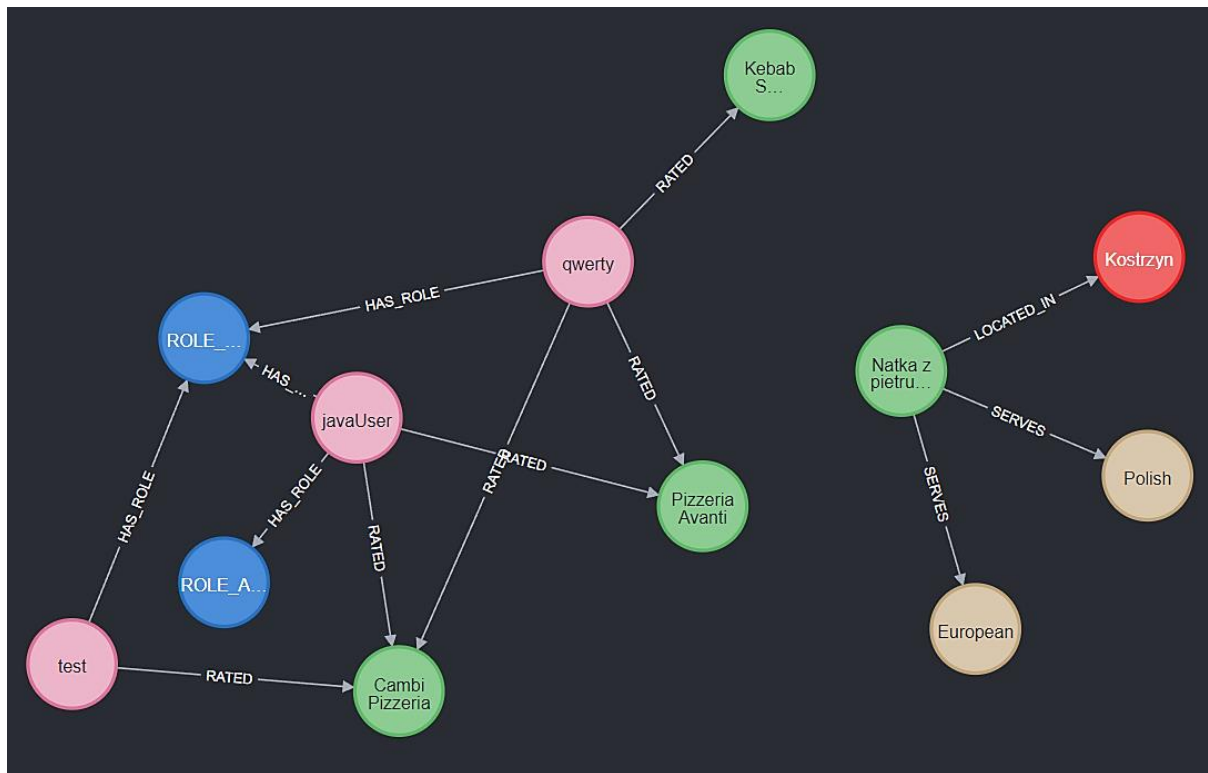
- **User**

Nazwa	Typ	Opis
id	integer	Domyślny identyfikator użytkownika
password	string	Hasło użytkownika
username	string	Login użytkownika
name	string	Opcjonalna nazwa użytkownika

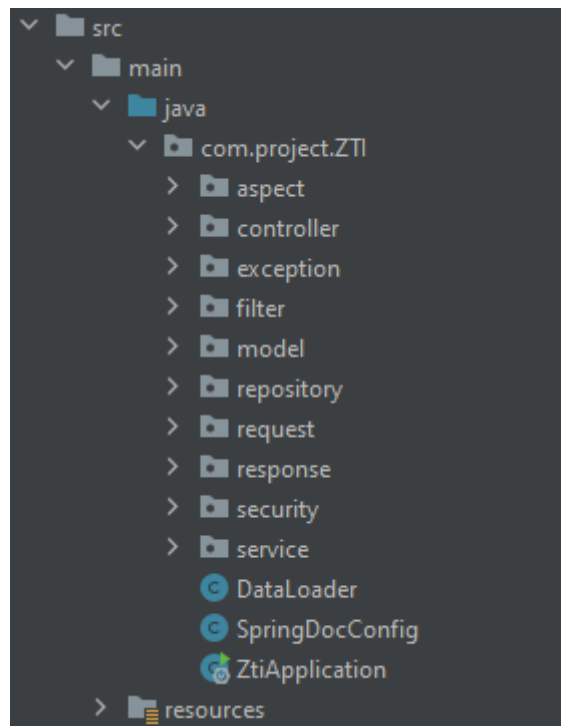
- **Role**

Nazwa	Typ	Opis
id	integer	Domyślny identyfikator użytkownika
role	string	Nazwa roli

Na poniższym rysunku przedstawiono wizualizację diagramu zdefiniowanego modelu.



## 5. Aplikacja serwerowa



Zgodnie z zdefiniowanymi wymaganiami aplikacji serwerowej, wykorzystano technologie takie jak:

- **Spring oraz Spring Boot**
- **Spring Data**
- **Spring AOP**
- **Springdoc-OpenAPI (Swagger-ui)**

Aplikacja jest zarządzana przez narzędzie Maven i została podzielona na 9 głównych katalogów:

- **aspect** – zawiera klasy związane z obsługą aspektów z biblioteki Spring AOP
- **controller** – zawiera klasy obsługujące żądania, które trafiają do serwera
- **exception** – zawiera klasy wyjątków stworzonych przez programistę
- **filter** – zawiera filtry Spring Security definiujące logikę odpowiedzialną za autentykację oraz w dalszej kolejności za autoryzację
- **model** – zawiera klasy odpowiedzialne za mapowanie odpowiednich struktur bazy grafowej tj. wierzchołków i krawędzi
- **repository** – pakiet zawierający interfejsy dostępu do bazy danych
- **request i response** – pakiety zawierające klasy pomocnicze do obsługi żądań HTTP
- **security** – pakiet zawierający klasy odpowiedzialne za konfigurację zabezpieczeń oraz generację i walidację tokenów JWT
- **service** – pakiet zawierający klasy serwisowe, pełniące rolę pośrednika pomiędzy kontrolerami oraz bazą danych

Przykładowy klasa kontrolera

```
@RestController
@RequestMapping("/api")
public class RestaurantPropertiesController {

    private final RestaurantPropertiesService restaurantPropertiesService;

    @Autowired
    public RestaurantPropertiesController(RestaurantPropertiesService restaurantPropertiesService){
        this.restaurantPropertiesService = restaurantPropertiesService;
    }

    @Operation(summary = "Get all locations")
    @GetMapping("/locations")
    public ResponseEntity<List<Location>> getAllLocations(){
        return new ResponseEntity<>(restaurantPropertiesService.getAllLocations(), HttpStatus.OK);
    }

    @Operation(summary = "Get all cuisines")
    @GetMapping("/cuisines")
    public ResponseEntity<List<Cuisine>> getAllCuisines(){
        return new ResponseEntity<>(restaurantPropertiesService.getAllCuisines(), HttpStatus.OK);
    }
}
```

Wyżej przedstawiony kontroler obsługuje żądania związane z właściwościami restauracji: lokalizacja oraz typy kulinarne. Kontroler nie jest bezpośrednio związany z interfejsem dostępu do bazy danych, ale przekazuje przepływ działania do odpowiedniej metody serwisu, która manipuluje danymi i zwraca kontrolerowi odpowiednią porcję danych.

### Przykładowa klasa serwisu

```
@Service
@Slf4j
public class RestaurantPropertiesService {
    private final LocationRepository locationRepository;
    private final CuisineRepository cuisineRepository;

    @Autowired
    public RestaurantPropertiesService(LocationRepository locationRepository, CuisineRepository cuisineRepository){
        this.locationRepository = locationRepository;
        this.cuisineRepository = cuisineRepository;
    }

    public Location getLocationById(Long locationId){
        return locationRepository.findById(locationId).orElseThrow(() -> new LocationNotFoundException(locationId));
    }

    public Cuisine getCuisineById(Long cuisineId){
        return cuisineRepository.findById(cuisineId).orElseThrow(() -> new CuisineNotFoundException(cuisineId));
    }

    public List<Location> getAllLocations(){
        log.info("Get all locations");
        Sort sort = Sort.by("name").ascending();
        return locationRepository.findAll(sort);
    }

    public List<Cuisine> getAllCuisines(){
        log.info("Get all locations");
        Sort sort = Sort.by("name").ascending();
        return cuisineRepository.findAll(sort);
    }
}
```

Przedstawiona powyżej klasa odpowiada za manipulację danymi pobranymi z bazy danych oraz zwróceniem ich do kontrolera. Wykorzystywane są również customowe klasy wyjątków, które odpowiadają za zwrócenie użytkownikowi odpowiedniej informacji błędu.



## Przykładowa klasa modelu

```
@Node
@Getter
@Setter
@NoArgsConstructor
public class Restaurant {

    @Id @GeneratedValue private long id;

    private String name;
    private String address;
    private double address_latitude;
    private double address_longitude;
    private boolean vegan_options;
    private boolean vegetarian_friendly;
    private boolean gluten_free;
    private String popularity_detailed;
    private String popularity_generic;
    private String trip_advisor_id;

    @Relationship(type = "SERVES", direction = Relationship.Direction.OUTGOING)
    private Set<Cuisine> cuisines;

    @Relationship(type = "LOCATED_IN", direction = Relationship.Direction.OUTGOING)
    private Location location;

    @Relationship(type = "RATED", direction = Relationship.Direction.INCOMING)
    private List<Rates> rates;
}
```

```
@RelationshipProperties
@Getter
@Setter
public class Rates {
    @RelationshipId
    private Long id;
    private int rating;
    private String comment;

    //points towards user -rated-> restaurant
    @TargetNode
    @JsonIgnore
    private final User user;

    public Rates(User user, int rating, String comment){
        this.user = user;
        this.rating = rating;
        this.comment = comment;
    }
}
```

Wyżej przedstawiony model odwzorowuje wierzchołki typu Restaurant z bazy danych Neo4j.

Jako, że restauracja jest powiązana z innymi typami wierzchołków, należało te relacje zdefiniować. W omawianym przykładzie istnieją 2 relacje wychodzące (OUTGOING) od wierzchołków restauracji do odpowiednio typów kuchni oraz lokalizacji. Wyróżniamy również jedną relację przychodzącą (INCOMING), jest to relacja wiążąca użytkownika, który ocenił/skomentował restaurację.

### Przykładowy interfejs dostępu do bazy danych

```
public interface RestaurantRepository extends Neo4jRepository<Restaurant, Long> {
    List<Restaurant> findRestaurantByLocationId(Long id);

    List<Restaurant> findTopBy(Pageable pageable);

    Optional<Restaurant> findRestaurantById(Long id);

    List<Restaurant> findRestaurantByIdIn(List<Long> id);

    @Query("MATCH (m:Restaurant)-[:LOCATED_IN]->(location:Location)<-[:LOCATED_IN]-(other:Restaurant)\n" +
        "where id(m) = $restaurantId\n" +
        "MATCH (m)-[:SERVES]-(t)-[:SERVES]-(other)\n" +
        "WITH m, other, COUNT(t) AS intersection, COLLECT(t.name) AS i\n" +
        "MATCH (m)-[:SERVES|LOCATED_IN]-(mt)\n" +
        "WITH m, other, intersection, i, COLLECT(mt.name) AS s1\n" +
        "MATCH (other)-[:SERVES|LOCATED_IN]-(ot)\n" +
        "WITH m, other, intersection, i, s1, COLLECT(ot.name) AS s2\n" +
        "WITH m, other, intersection, s1, s2\n" +
        "WITH m, other, intersection, s1+[x IN s2 WHERE NOT x IN s1] AS union, s1, s2\n" +
        "RETURN other as restaurant, union as params, round(((1.0*intersection)/SIZE(union)), 2) " +
        "AS jaccard ORDER BY jaccard DESC LIMIT 100")
    List<RecommendationsByRestaurantProjection> findRestaurantRecommendationsByCity(@Param("restaurantId") Long restaurantId);

    @Query("MATCH (m:Restaurant)-[:SERVES|LOCATED_IN]-(t)-[:SERVES|LOCATED_IN]-(other:Restaurant)\n" +
        "where id(m) = $restaurantId\n" +
```

Zdefiniowano interfejs rozszerzający Neo4jRepository<> w którym zapewniono deklaracje własnych metod zwracających pożądane dane oraz 3 metody obsługujące polecenia języka zapytań Cypher.

### Spring AOP

Zgodnie z wymaganiami projektu, wykorzystano programowanie aspektowe do obsługi zdarzeń. W omawianej aplikacji serwerowej aspekty odpowiadają za zwracanie komunikatów, które ułatwiają analizę uruchomionej aplikacji.

```
: CONTROLLER INFO ==> path(s): [/user/recommendations], method(s): GET, arguments: ["javax.servlet.http.HttpServletRequest"]
: Fetching user qwerty from the database
: SERVICE INFO <== method: UserServiceImplementation.getUser(..), retuning: class com.project.ZTI.model.user.User
: SERVICE INFO <== method: UserRecommendationService.getUsers(..), retuning: interface java.util.List
: CONTROLLER INFO <== path(s): [/user/recommendations], method(s): GET, retuning: "OK"
```

Na powyższym rysunku, widoczny jest wycinek konsoli programu. Po wywołaniu przykładowego punktu końcowego o adresie: **/api/user/recommendations**, zostały zarejestrowane następujące zdarzenia:

- wywołanie metody kontrolera,
- wywołanie metody serwisu,
- zwrócenie danych przez metodę serwisu,
- zwrócenie danych przez metodę kontrolera.

## Dokumentacja REST API

Za opis REST API odpowiada narzędzie SWAGGER, które wspomaga proces dostarczania dokumentacji punktów końcowych. Poniżej przedstawiono pewien wycinek wygenerowanej dokumentacji.

<b>restaurant-controller</b>		▼
<b>user-administration-controller</b>		
PUT	/api/admin/role/cancel Remove admin role from user	▼ 🔒
PUT	/api/admin/role/assign Assign role to user	▼ 🔒
POST	/api/admin/role/save Add new role	▼ 🔒
GET	/api/admin/users Get all users	▼ 🔒
<b>auth-controller</b>		^
POST	/api/user/save Register new user with basic roles	▼ 🔒
POST	/api/login Login and retrieve new authentication token	▼ 🔒
GET	/api/token/refresh Refresh authentication token using old token	▼ 🔒
<b>restaurant-administration-controller</b>		^
POST	/api/admin/restaurant Add new restaurant to database	▼ 🔒
DELETE	/api/admin/restaurant/{restaurantId} Delete restaurant by id	▼ 🔒
<b>user-recommendation-controller</b>		▼
<b>restaurant-recommendations-controller</b>		▼
<b>restaurant-properties-controller</b>		▼

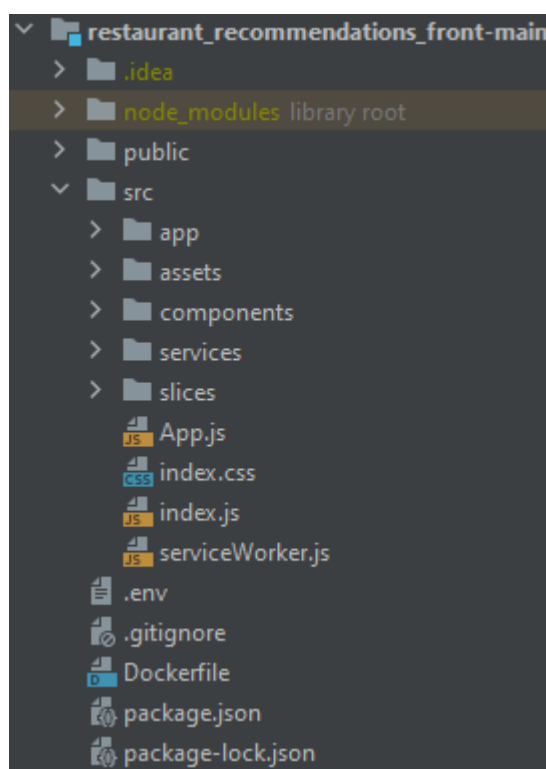
## 6. Aplikacja klienta

W ramach interfejsu użytkownika stworzono prostą aplikację internetową typu SPA. Do jej utworzenia wykorzystano biblioteka React.js oraz język JavaScript.

Aplikacja wykorzystuje dodatkowe biblioteki:

- **React-Redux**
- **Material UI**
- **React-Bootstrap**
- **Leaflet** oraz **OpenStreetMap**

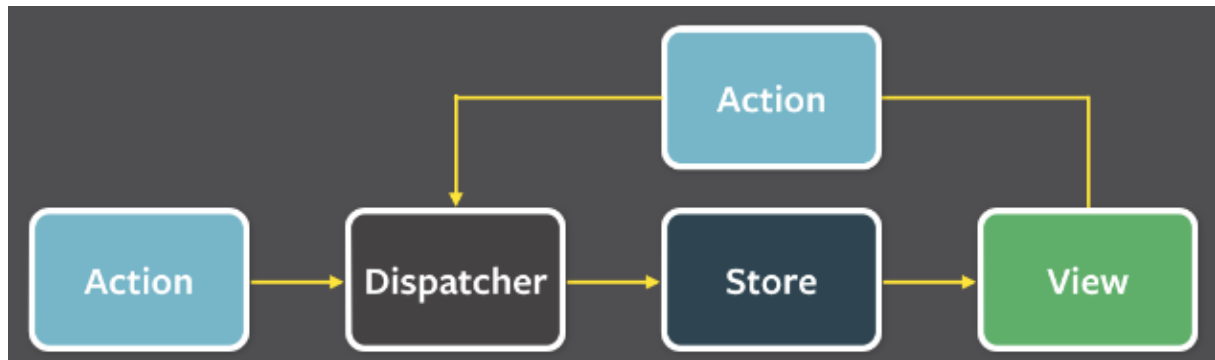
Aplikacja została podzielona na niezależne komponenty, gdzie każdy odpowiada za określoną funkcjonalność serwisu. Do inicjalizacji projektu użyto narzędzia Create React App. Struktura plików aplikacji klienta została przedstawiona na poniższym rysunku.



- **public** – zawiera podstawowe elementy aplikacji tj. plik index.html w którym renderowane są elementy interfejsu użytkownika,
- **src** – katalog zbiorczy, zawierający kod źródłowy aplikacji,
- **assets** – katalog zawierające elementy graficzne aplikacji,
- **components** – katalog komponentów – fragmentów interfejsu użytkownika,
- **services** – zbiór funkcji odpowiedzialnych za realizację zapytań do aplikacji serwerowej,
- **slices** – zbiór akcji i reduktorów z biblioteki React-Redux, umożliwiających zarządzanie centralnym stanem komponentów,
- **App.js** – główny komponent aplikacji,
- **index.js** – punkt startowy aplikacji.

## Zarządzanie stanem aplikacji

W celu ułatwienia przepływu stanu pomiędzy komponentami wykorzystano centralny magazyn stanu obsługiwany przez bibliotekę Redux.



Interfejs użytkownika wykorzystuje bibliotekę Axios oraz magazyn stanu Redux do wysyłania oraz przechowywania odpowiedzi serwera. Logikę komunikacji z serwerem podzielono na 2 warstwy:

- pierwszą jest warstwa serwisów, komunikujących się bezpośrednio z API,
- drugą warstwą jest zbiór akcji oraz reduktorów.

Przykładowa metoda serwisu została przedstawiona poniżej:

```
const login = (username, password) => {
  return axios
    .post( url: API_URL + "login",
      data: {
        username,
        password
      })
    .then((response : AxiosResponse<any> ) => {
      if (response.data.access_token) {
        localStorage.setItem("user", JSON.stringify(response.data));
      }
      if (response.data.status === 'fail') {
        throw new Error(response.data.message);
      }
      return response.data;
    });
}
```

Metoda ta wykorzystując bibliotekę Axios wysyła żądanie HTTP typu POST do API zlokalizowanego pod adresem API\_URL.

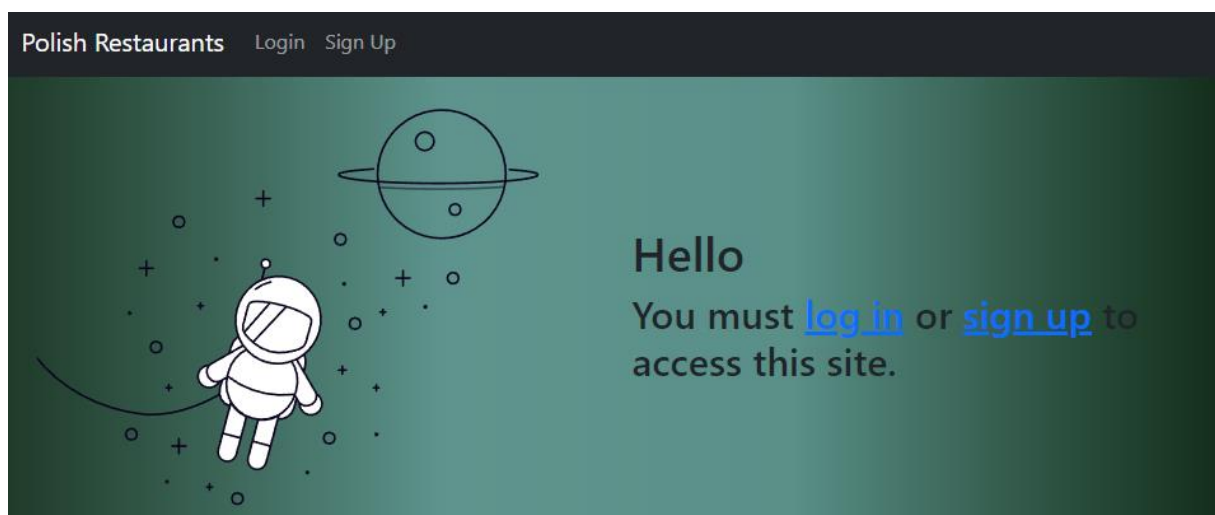
Przykładowa budowa funkcji akcji została przedstawiona poniżej:

```
export const login = createAsyncThunk(
  typePrefix: "auth/login",
  payloadCreator: async ({username, password}, thunkAPI : GetThunkAPI<{}> ) => {
    try {
      const data = await AuthService.login(username, password);
      return {user: data};
    } catch (error) {
      const message =
        (error.response &&
          error.response.data &&
          error.response.data.message) ||
        error.message ||
        error.toString();
      thunkAPI.dispatch(setMessage(message));
      return thunkAPI.rejectWithValue();
    }
  }
);
```

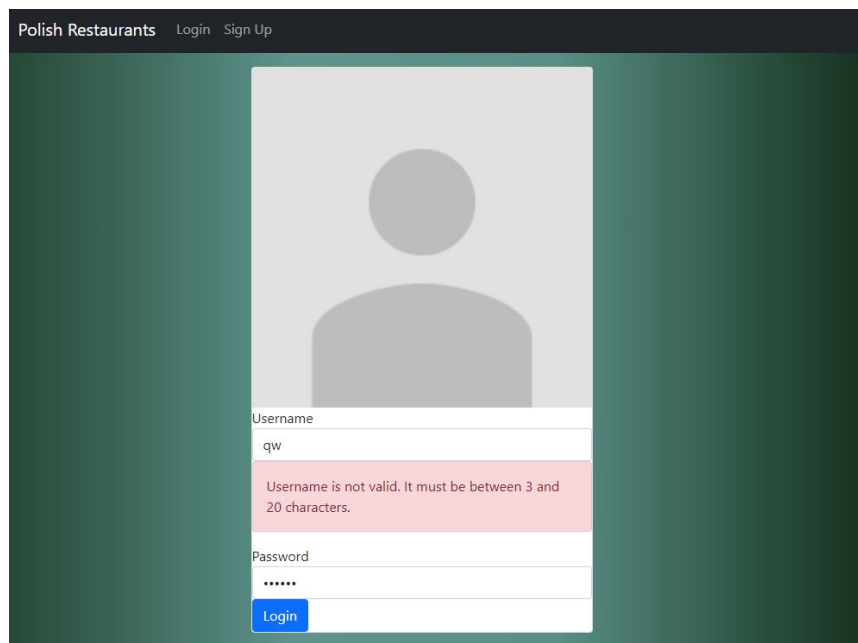
```
[login.fulfilled]: (state, action) => {
  state.isLoggedIn = true;
  state.user = action.payload.user;
},
[login.rejected]: (state, action) => {
  state.isLoggedIn = false;
  state.user = null;
},
```

Omówienie udostępnionej funkcjonalności

- Strona domowa aplikacji



- Strona logowania/rejestracji



Polish Restaurants Login Sign Up

Username

qw

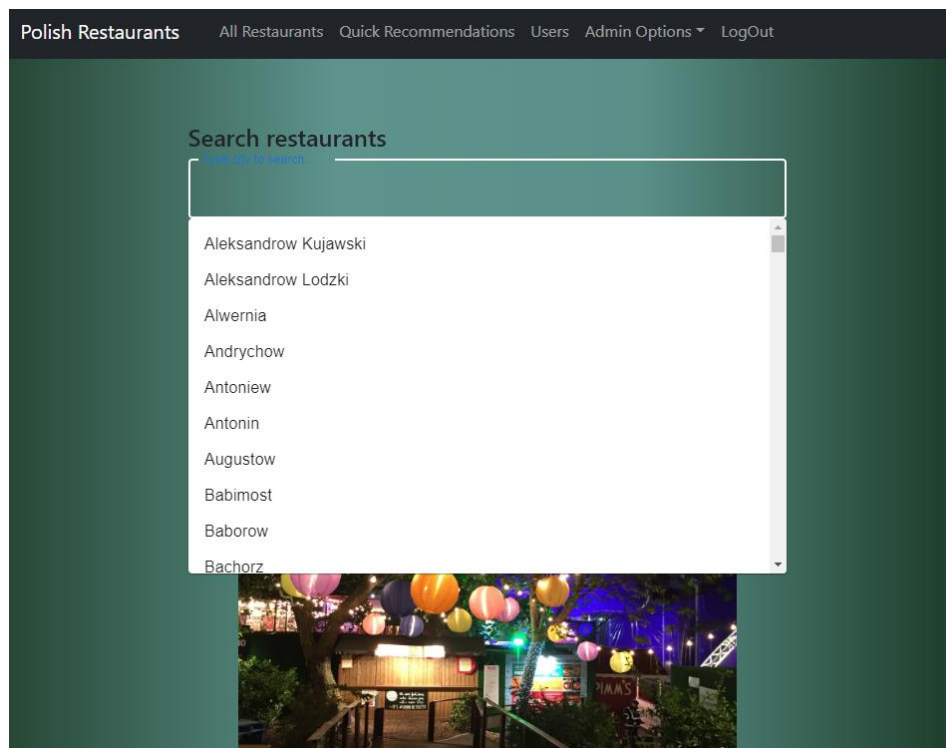
Username is not valid. It must be between 3 and 20 characters.

Password

\*\*\*\*\*

Login

- Wyszukiwanie restauracji w wybranym mieście



Polish Restaurants All Restaurants Quick Recommendations Users Admin Options LogOut

Search restaurants

Type city to search...

Aleksandrow Kujawski

Aleksandrow Lodzki

Alwernia

Andrychów

Antoniew

Antonin

Augustów

Babimost

Baborów

Baborz

- Wyszukiwanie restauracji w wybranym mieście - wyniki

### Search restaurants

Type city to search...  
Krakow

Search in all: Search 1687 records...

Hide columns

Properties		
Name	Cuisines	Location
<div>All</div>		
<div>Smakolyki</div> <div>Straszewskiego 28, Krakow 31-113 Poland</div>	<ul style="list-style-type: none"> <li>Polish</li> <li>EasternEuropean</li> <li>CentralEuropean</li> <li>European</li> </ul>	Krakow
<div>Pronto Pizza &amp; Grill</div> <div>Ulica Plac Dominikanski 1 Plac Dominikanski 1, Krakow 31-043 Poland</div>	<ul style="list-style-type: none"> <li>Polish</li> <li>Italian</li> <li>Pizza</li> <li>Grill</li> </ul>	Krakow
<div>Karczma Rzym</div> <div>Tyniecka 118h, 30-376, Krakow Poland</div>	<ul style="list-style-type: none"> <li>Pub</li> <li>Polish</li> <li>European</li> </ul>	Krakow

«

<

>

»

1 of 338

1

Show 5



- Okno wyświetlające szczegóły wybranej restauracji

S

Smakolyki  
Straszewskiego 28, Krakow 31-113 Poland

More info: [Trip Advisor](#)

100% NATURAL

VEGAN

100% VEGETARIAN

GLUTEN FREE

#63 of 1498 Restaurants in Krakow  
#80 of 2056 places to eat in Krakow

Rate: ☆☆☆☆☆

Leave a comment

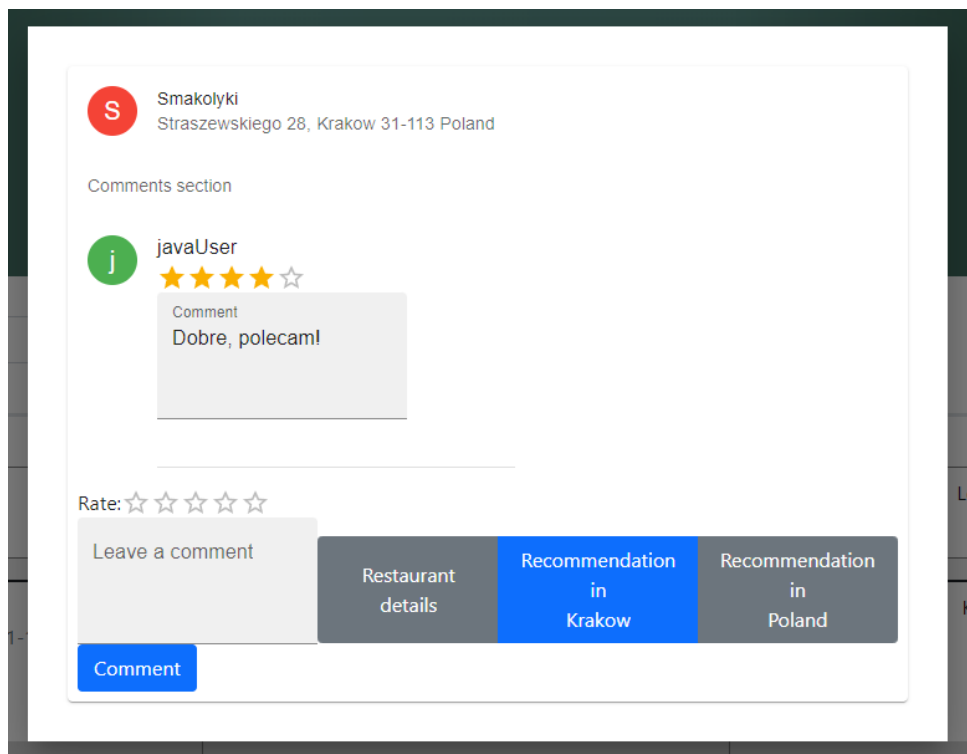
All comments

Recommendation in Krakow

Recommendation in Poland

Comment

- Komentarze innych użytkowników



- Funkcjonalność rekomendacji w wybranym mieście lub na terenie całego kraju

Search in all:

Hide columns

Properties		
Name	Matched Parameters	Recommendation accuracy
<input type="text" value="Search 100 records..."/>		
Marmolada ul. Grodzka 5 31-008 Krakow, Krakow 31-006 Poland	<ul style="list-style-type: none"> <li>European</li> <li>EasternEuropean</li> <li>Polish</li> <li>CentralEuropean</li> <li>Krakow</li> </ul>	<ul style="list-style-type: none"> <li>80 %</li> </ul>
Restauracja Max 18 Rynek Główny, Krakow Poland	<ul style="list-style-type: none"> <li>European</li> <li>EasternEuropean</li> <li>Polish</li> <li>CentralEuropean</li> <li>Krakow</li> </ul>	<ul style="list-style-type: none"> <li>80 %</li> </ul>
Restauracja Polska	<ul style="list-style-type: none"> <li>European</li> </ul>	<ul style="list-style-type: none"> <li>80 %</li> </ul>

1 of 20
1

Show 5

- Funkcjonalność rekomendacji restauracji w oparciu o interakcje innych użytkowników

Search in all:

Hide columns ▾


Properties		
Name	Users who likes it	Average rating
<input type="text" value="Search 3 records..."/>		
Smakolyki Straszewskiego 28, Krakow 31-113 Poland	<ul style="list-style-type: none"> <li>javaUser</li> </ul>	★★★★☆
Cambi Pizzeria ul. Lipowa 4, Aleksandrow Kujawski 87-700 Poland	<ul style="list-style-type: none"> <li>javaUser</li> </ul>	★★★★☆☆
Pizzeria Avanti ul. Fryderyka Chopina 7, Aleksandrow Kujawski 87-700 Poland	<ul style="list-style-type: none"> <li>javaUser</li> </ul>	★★★☆☆

« < > »
1 of 1
1
Show 5 ▾

- Funkcjonalność wyszukiwanie użytkowników o zbliżonych preferencjach


Polish Restaurants
All Restaurants
Quick Recommendations
Users
Admin Options ▾
LogOut

## Similar Users


test
Similarity: 100%

Similar Restaurants:

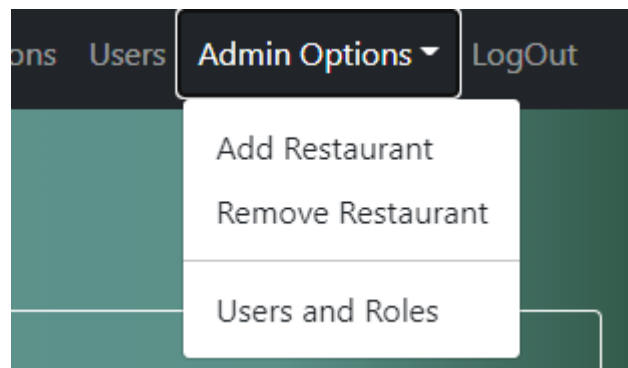
- Cambi Pizzeria


javaUser
Similarity: 98%

Similar Restaurants:

- Pizzeria Avanti
- Cambi Pizzeria

- Opcje administracyjne dla użytkowników posiadających uprawnienia administratora



- Opcja dodania nowej restauracji

### Add new restaurant

Restaurant Name

Type city to search...

Address

NEXT

### Properties

Type cuisine to search...

☐ Vegetarian

☐ Vegan

☐ Gluten Free

PREVIOUS SAVE

### Location details

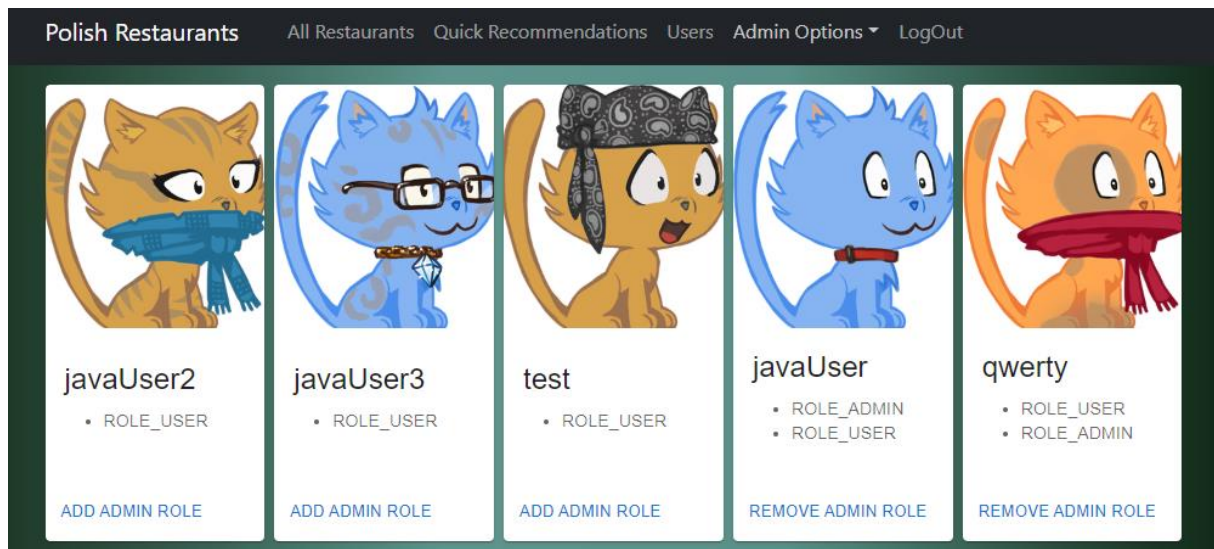
Latitude

Longitude

PREVIOUS NEXT

 A screenshot of the 'Location details' form. It features a map of Warsaw with a blue pin indicating the restaurant's location. Below the map are input fields for 'Latitude' and 'Longitude', and 'PREVIOUS' and 'NEXT' buttons.

- Opcja zarządzania użytkownikami serwisu



## 7. Przewodnik uruchomienia aplikacji

Uruchomienie aplikacji wymaga od użytkownika posiadania narzędzi: **Docker**, **Docker Compose** oraz **Maven**. Kod źródłowy został umieszczony w repozytorium GitHub:

[https://github.com/patryk0504/restaurant\\_recommendation](https://github.com/patryk0504/restaurant_recommendation)

### Szybka metoda uruchomienia aplikacji

- Należy pobrać plik **remote-docker-compose.yml** ([LINK](#)) i umieścić go w dowolnym folderze
- Z poziomu katalogu z plikiem remote-docker-compose.yml należy wywołać następujące polecenie:
  - **`docker-compose -f ./remote-docker-compose.yml up -d`**
  - pobrane zostaną obrazy z zewnętrznego repozytorium Docker Hub, a następnie aplikacja zostanie uruchomiona,
  - operacja może chwilę potrwać.
- Interfejs użytkownika dostępny jest pod adresem:
  - <http://localhost:3000>
- Aplikacja serwera dostępna jest pod adresem:
  - <http://localhost:7777>
- Swagger UI - dokumentacja
  - <http://localhost:7777/swagger-ui/index.html>
  - aby skorzystać z udostępnionych punktów końcowych należy:
    - uruchomić endpoint **/api/login** podając dane logowania,
    - skopiować uzyskany token autoryzacyjny,

- przejść na samą górę strony Swagger UI i wybrać opcję **Authorize**, która jest widoczna w prawym górnym rogu ekranu,
- po wklejeniu i uzyskaniu autoryzacji, wszystkie punkty końcowe są dostępne do testowania.
- W celu zakończenia pracy aplikacji wykonujemy polecenie:
  - **`docker-compose -f ./remote-docker-compose.yml down`**

Testowa dane logowania:

- Login: qwerty
- Hasło: qwerty

## Dłuższa metoda uruchomienia aplikacji

Metoda ta wymaga od użytkownika pobrania kodu źródłowego oraz samodzielnej kompilacji kodu serwera do pliku .jar oraz zbudowania aplikacji React wykorzystując npm.

- Należy sklonować pliki z repozytorium:
  - **`git clone --recurse-submodules`**  
[https://github.com/patryk0504/restaurant\\_recommendation](https://github.com/patryk0504/restaurant_recommendation)
- Należy przejść do katalogu **restaurant\_recommendation/ZTI** i wywołać polecenie:
  - **`mvn clean package`**
  - zbudowany zostanie plik .jar
- Należy przejść do katalogu **restaurant\_recommendation** i wywołać polecenie:
  - **`docker-compose build`**
  - zostaną zbudowane obrazy Dockera, budowa aplikacji React w zależności od wydajności komputera może chwilę potrwać ~5min
- Następnie aby uruchomić aplikację wywołujemy polecenie:
  - **`docker-compose up -d`**
  - zostaną uruchomione serwisy aplikacji klienta oraz aplikacji serwerowej, dodatkowo zostaną wczytane dane do bazy Neo4j – wykonanie może chwilę potrwać
    - wykorzystywane porty to: 7777 – backend, 3000 – frontend, 7474 oraz 7687 – baza Neo4j
- Interfejs użytkownika dostępny jest pod adresem:
  - <http://localhost:3000>
- Aplikacja serwera dostępna jest pod adresem:
  - <http://localhost:7777>
- Swagger UI - dokumentacja
  - <http://localhost:7777/swagger-ui/index.html>
- Aby wstrzymać działanie aplikacji wykonujemy polecenie:
  - **`docker-compose down`**

Testowa dane logowania:

- Login: qwerty
- Hasło: qwerty

## 8. Podsumowanie

Podsumowując, wymagania funkcjonalne/niefunkcjonalne zdefiniowane w rozdziale 2 oraz założenia projektowe zostały spełnione:

- aplikacja serwerowa, aplikacja klienta oraz baza danych wykorzystują konteneryzację,
- serwer wykorzystuje technologię Spring oraz narzędzie Spring Boot,
- dostęp do danych realizowany w oparciu o Spring Data,
- zaimplementowana obsługa zdarzeń z wykorzystaniem programowania aspektowego Spring AOP,
- aplikacja klienta typu SPA, opracowana z wykorzystaniem biblioteki React.js.