Zaawansowane techniki internetowe

Dokumentacja projektu

Aplikacja ułatwiająca śledzenie i rozliczanie wydatków wśród grupy znajomych

Aleksandra Rolka

Informatyka Stosowana
Wydział Fizyki i Informatyki Stosowanej
Akademia Górniczo-Hutnicza w Krakowie

1. Temat i cel projektu

Aplikacja dotyczy rozliczania wspólnych wydatków, rachunków pośród grup znajomych. Często problematyczne jest rozliczanie się np. ze wspólnych wyjazdów, gdzie są wydatki wspólne płacone przez jedną osobę, czasem jedna osoba płaci za drugą, innym razem za siebie i część znajomych. Dla tego celu stworzona została ta aplikacja (obecnie tylko część serwerowa), aby śledzić rozliczenia, ale przede wszystkim by długi pomiędzy członkami grupy były uproszczone, by ilość zwrotów pieniędzy była jak najmniejsza.
Dodatkowo osobistym celem było poznanie nowych technologii backendowych, w tym głównie technologii Spring Boot.

3. Założenia funkcjonalne

Użytkownik powinien mieć możliwość:

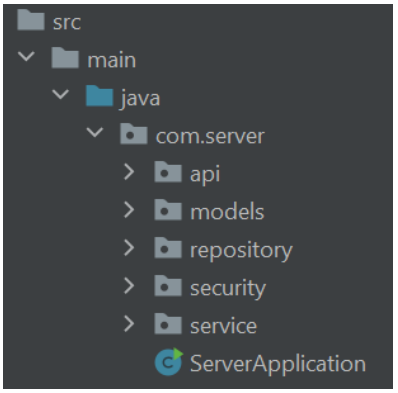
- założenia konta
- tworzenia grup znajomych
- dodawania kolejnych znajomych do grupy
- tworzenia wydatków w danej grupie ze szczegółami tj.:
    - tytuł w którym może zawrzeć informacje o celu wydatku
    - opis - w razie potrzeby jako poszerzenie informacji z tytułu
    - informacje o tym kto zapłacił za dany wydatek (domyślnie osoba, która dodaje wydatek do listy)
    - lista osób, pośród których ma być rozliczony rachunek
        - możliwość podzielenia rachunku po równo jak i indywidualnie dla kazdej osoby osobno ze wskazaniem odpowiedich kwot
- rozliczania się pomiędzy znajomymi w grupie tworząc symboliczną płatność, aby móc odnotować w systemie zwrot i tym samym zaktualizowć bilans wybranych użytkowników

4. Backend

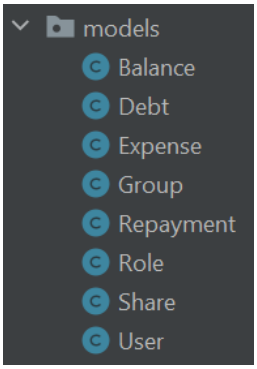
Część backendowa została utworzona z wykorzystaniem technologii Spring Boot. Aplikacja korzysta z bazy danych postgreSQL hostowanej w chmurze - ElephantSQL. Przygotowane interfejs API serwer-klient wykonany został w architekturze REST.

4.1 Struktura projektu

Poniżej znajduje się główny zarys struktury projektu:



- *models/* - znajdują się tutaj zdefiniowane klasy z adnotacjami *@Entity*, które mapowane są na tabele w bazie danych



Przykładowy fragment klasy:

```
@Slf4j
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "users",
        uniqueConstraints = {@UniqueConstraint(columnNames = "email")})
public class User {
    @Id @GeneratedValue(strategy = AUTO)
    private Long id;
    @NotBlank
    @Size(max=50)
    private String firstName;
    @NotBlank
    @Size(max=50)
    private String lastName;
    @NotBlank
    @Size(max=50)
    private String email;
    @NotBlank
    private String password;
    @ManyToMany(fetch = FetchType.EAGER)
    private Collection<Role> roles;
    private String joinDate;
```

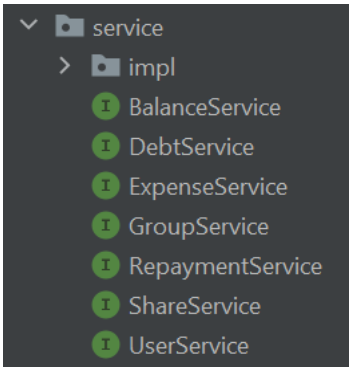
Dzięki dodaniu kilku adnotacji, możliwe jest zaoszczędzenie pisania linijek kodu. Aadnotacja *@Data* tworzy wszystkie settery i gettery, kolejne dwie widzone na zdjęciu tworzą konstruktory bezargumentowe i argumentowe. W niektórych klasach zostały one nadpisane własnymi zdefioniowanymi konstruktrorami, w przypadku gdzie celem było przypisanie innych wartości niż w konstruktorach stworzonych automatycznie ub w przypadku potrzeby utwrzenia konstruktorów np z częścią argumentów. Możliwe również jest dodanie resrykcji tak jak w tym przypadku, że pole

‘email’ ma być unikalne w tabeli. Dodatkowo możliwe jest dodanie również wymagań co do wartości poszczególnych pól.

- *repository/* - zawiera interfejcy do komunikacji bezposredniej z bazą danych, znajdują się w niej zapytania do bazy, które można utworzyć automatycznie przestrzegając składni wraz z wykorzystaniem nazw pól zdefiniowanych z klasach w folderze models/ lub możliwe jest tez utworzenie SQLowych zapytań dzięki adnotacji *@Query,* w której możemy umieścić treś zapytania :

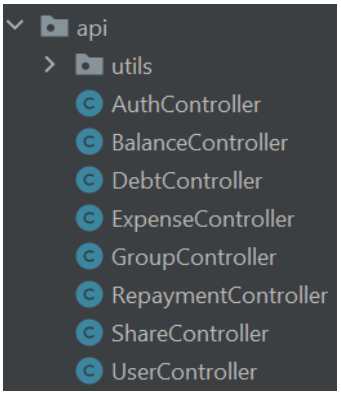
```
public interface GroupRepository extends JpaRepository<Group, Long> {

    Group findByName(String name);
    Optional<Group> findById(Long id);
    List<Group> findAll();
    Collection<User> findMembersById(Long id);
}
```

- *service/* - tutaj znajduje się główna logika biznesowa,



metody klas z adnotacjami @Service są można powiedzieć wykonawcami tego co zleci, zarządzi kontroller, który znajduje w folderze api/. Natomiast klasy umieszczone w folderze service/, a dokładniej implementacje poszczególnych interfejsów odpowiadają za dostęp do bazy danych i wykonanie odpowiednich zapytań pośrednio przed metody obiektów ‘**Repository’*, następnie manipulacje, przetwarzanie, filtrowanie tych danych i ich błędów, a następnie zwrócenie wyniku z powrotem do klas odpowiadających za API

- *api/* - znajdują się tam klasy z adnotacjami *@RestCotroller* mające pełnić API do komunikacji i zarządzania danymi bazy danych. Klasy podzielone są ze względu na to jakich entity głównie dotyczą:



Wewnątrz klas można znaleźć odpowiednie metody odpowiadające endpointom wraz z typem requestu http, np:

```
@GetMapping(path = "/user")
public ResponseEntity<?> getUserByEmail(@RequestBody Email obj) {
    log.info("Fetch user by email: {}", obj.getEmail());
    User user = userService.getUser(obj.getEmail());
    if(user == null)
        return ResponseEntity.badRequest().body( body: new CustomErrorMessage("User not exist!"));
    return ResponseEntity.ok().body( body: user);
}
```

Widoczny mamy tutaj typ requestu jakim jest *GET*, czyli zasoby są pobierane i wysyłane w odpowiedzi. Na obiekcie *userService* wywołana jest metoda *getUser*. Zapewnia to wydzielenie w projekcie API od logiki biznesowej i dostępu do danych.

- security/ - dodatkowo wydzielona została część odpowiadająca za kontrole dostępu do pozostałych części aplikacji. Znajduje się tu filtrowanie dostępu do odpowiednich endpointów API. Filtrowanie odbywa się tu z wykorzystaniem autoryzacji poprzez role użytkowników oraz autentykacji z wykorzystaniem tokenów JWT: access i refresh token.

4.2 Baza danych

W bazie danych dostępnych jest kilka modeli utworzonych z wykorzystaniem bibliioteki Spring Data JPA, która mapuje zdefiniowane w projekcie klasy na table w bazie danych. Oczywiście nie mapuje wszystkich, a te oznaczone adnotacją *@Entity*. Poniżej znajdującą się klasy zdefiniowane w projekcie, które zostały zmapowane do bazy danych::

- *Role* - z nazwą roli użytkownika, by w przyszłości móc rozdzielić funkcjonalność pomiędzy zwyklego uzytkownika i administratora, na ten moment to pole nie wpływa na autoryzację dostępu do danych
- *User* - tabela zawierająca takie informacje jakk imie, nazwisko, email uzytkownika, hasło oraz listę ról typu *Role* jakie ma przypisane
- *Group* - zawiera podstawowe informacje o grupach (nazwa, timestamp utworzenia groupy oraz jej aktualizacji) oraz listę użytkowników typu *User*
- *Debt* - odpowiada długowi pomiędzy dwoma użytkownikami z grupy, zawiera id obydwu użytkowników, id grupy z której dany dług pochodzi, timestap utworzenia długu oraz kwotę (dodatnia wartość oznacza, że drugi uzytkownik jest winny pierwszemu wskazaną kwotę)
- *Balance* - id użytkownika, iformacje o wszystkich długach wobec innych, o dlugach które inni są danemu uzytkownikowi dłużni oraz ostateczny bilans
- *Repayment* - logicznie prezentuje spłatę należności pomiędzy dwoma użytkownikami, zawiera id użytkownika, który wykonał płatność, id użytkownika, który tą płatność otrzymał, id grupyużytkowników, kwotę, oraz timestamp daty utworzenai płatności
- *Share* - prezentuje element podziału płatności za wydatek, zawiera informacje o tym który użytkownika płacił , który jest winny, id grupy, kwotę, oraz timestamp utworzenia

- *Expense* - model ten jest ogólnym obiektem który może prezentować wydatek lub płatność, został stworzony dla ułatwienia grupowania wszystkich transakcji, zawiera pola takie jak tytuł, opis, id grupy w której dokonana była tranzakcja, informacje czy jest to spłata czy wydatek, pole typu Repayment, wypełniane w przypadłu spłaty, listę typu Share, która jest wypełniana w przypadku wydatku i zawiera informacje o podziale płatności

5. Frontend

W założeniu miała być to aplikacji SPA z wykorzystniem biblioteki React, jednak ta część nie powstała..

6. Uruchomienie aplikacji

Aplikacja docelowo miała być uruchamiana z wykorzystaniem konteneryzacji. Odpowiednie pliki Dockerfile i docker-compose.yaml zostały przygotowane i przetestowane manualnie, aplikacja buduje i uruchamia się poprawnie. Uruchomić ją można wykonując w terminalu poniższą kommendę:

docker-compose up --build

oraz aby zatrzymać i usunąć utworzony kontener:

docker-compose down

Jednak ze względu na brak frontendu proszę o przeczytanie informacji w kolejnym punkcie.

6. Informacje dodatkowe

Aby pokazać, że część serwerowa działa. że brakuje aplikacji tylko opakowania i dobrego interfesu dostępu do danych jakim byłby frontend to załączam export z aplikacji Postman, w której testowałam część przygotowanych endpointów. Po uruchomieniu projektu np w IDE Intellij (lub zbudowaniu dockera) i załadowaniu załączonego pliku z exportem do Postmana możliwe jest przetestowanie funkcjonalności backendu. Poniżej załączam screenshot z przygotowanymi testami wybranych endpointów:

POST Login | localhost:8080/api/auth/login

GET Get all users | localhost:8080/api/users

GET Refresh access token | localhost:8080/api/au...

POST Add role to user | localhost:8080/api/role/ad...

GET Get user by Id | localhost:8080/api/user/1

GET Get user by email | localhost:8080/api/user

POST Signup | localhost:8080/api/auth/signup

POST Add group | localhost:8080/api/group/save

GET Get all groups | localhost:8080/api/groups

GET Get group by id | localhost:8080/api/group/7

POST Add members to group | localhost:8080/api/...

GET Get members by group id | localhost:8080/a...

GET Get all user groups | localhost:8080/api/user/...

GET Get all expenses | localhost:8080/api/expens...

GET Add repayment | localhost:8080/api/expense...

POST Add payment| localhost:8080/api/expense/a...