Aplikacja webowa do planowania podróży

(Web application for travel planning)

Marcin Sarnecki

Praca inżynierska

Promotor: prof. Małgorzata Biernacka

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

3 czerwca 2024

Streszczenie

Aplikacja "Travel Planner" jest projektem umożliwiającym użytkownikom planowanie wspólnych podróży z użyciem interaktywnych map Google. Główne funkcjonalności aplikacji obejmują możliwość tworzenia kont użytkowników, zalogowanie się poprzez zewnętrzną aplikacje, zarządzanie znajomościami, organizowanie podróży, planowanie wydarzeń oraz zapraszanie znajomych do wspólnych wyjazdów. Dodatkowo, konteneryzacja z użyciem Dockera ułatwiaja proces wdrażania. Aplikacja korzysta z Spring Framework, Thymeleaf, Google Maps API oraz PostgreSQL, zapewniając solidne i efektywne środowisko dla użytkowników oraz deweloperów.

The "Travel Planner" application is a project that enables users to plan group trips using interactive Google Maps. The main functionalities of the application include the ability to create user accounts, log in through external application, manage friendships, organize trips, plan events and invite friends to join the trips. Additionally, containerization using Docker simplifies the deployment process. The application utilizes the Spring Framework, Thymeleaf, Google Maps API, and PostgreSQL, providing a solid and efficient environment for both users and developers.

Spis treści

Rozdział 1.

Wprowadzenie

Pomysł na projekt opiera się na aplikacji webowej do planowania podróży z innymi użytkownikami przy wykorzystaniu interfejsu map Google. Na główne funkcjonalności składają się: tworzenie konta, możliwość zalogowania się poprzez zewnętrzną aplikację, dodawanie do znajomych innych użytkowników, tworzenie podróży i zapraszanie do niej znajomych, planowanie wydarzeń oraz system do rozliczeń. Celem jest ułatwienie organizacji podróży poprzez dostarczenie intuicyjnych narzędzi do planowania i koordynacji wspólnego wyjazdu.

Rozdział 2.

Technologie

Lista technologii użytych w projekcie:

- **Java**: Język programowania, użyty do głównej logiki aplikacji, wersja 17.
- **Spring Framework**: Framework Java, użyty do budowy backendu aplikacji, wersja 3.2.0.
- **Thymeleaf**: Silnik szablonów, użyty przy generowaniu plików HTML.
- **Javascript**: Język programowania, użyty przy dynamicznych zmianach wyglądu na stronach internetowych.
- **Google Maps API**: Interfejs programistyczny od firmy Google, użyty do integracji map na stronach służących do wyświetlania zaplanowanych miejsc i tras w trakcie trwania podróży.
- **Docker**: Narzędzie do konteneryzacji aplikacji
- **Test Containers**: Narzędzie do uruchamiania kontenerów Docker w testach integracyjnych, użyte przy testowaniu klas modyfikujących bazę danych.
- **PostgreSQL**: Relacyjna baza danych.
- **Git**: Repozytorium kodu, użyte do śledzenia i zarządzania zmianami w kodzie źródłowym.
- **Github**: Platforma hostingowa do przechowywania repozytoriów kodu. Pozwala również łatwą integrację z zewnętrznymi aplikacjami poprzez możliwość logowania z protokołem uwierzytelniania OAuth.
- **Maven**: Narzędzie do zarządzania zależnościami i budowania aplikacji, wersja 3.8.6.

Rozdział 3.

Funkcjonalności aplikacji

Poniżej znajduje się szczegółowy opis głównych funkcjonalności aplikacji.

3.1. Zarządzanie użytkownikami

Klasa `AppUserService` odpowiada za zarządzanie danymi użytkowników. Oto główne funkcje tej klasy:

3.1.1. Rejestracja i logowanie

- **Rejestracja:** Użytkownicy mogą rejestrować się w aplikacji za pomocą tradycyjnego formularza rejestracyjnego, podając nazwę użytkownika, email i hasło. Dane są weryfikowane (aplikacja sprawdza czy w bazie danych nie ma już użytkownika z tą samą nazwą, czy email jest poprawny oraz czy hasło jest wystarczająco silne)
- **Logowanie:** Autentykacja użytkowników jest obsługiwana przez standardowe logowanie (nazwa użytkownika i hasło dla użytkowników, którzy zarejestrowali się poprzez aplikację) lub przez OAuth2. W przypadku zewnętrznej autentykacji skorzystałem z Githuba, w tym przypadku po pomyślnym pierwszym zalogowaniu aplikacja utworzy konto dla użytkownika. W bazie danych przechowywana jest informacja, czy dany użytkownik zarejestrował się w standardowy sposób, czy skorzystał z zewnętrznej autentykacji (users.provider przyjmujące wartości 'APP' lub 'GITHUB').

3.1.2. Profil użytkownika

- **Aktualizacja profilu:** Użytkownicy mogą aktualizować swoje dane profilowe, takie jak imię (domyślnie przy rejestracji jest ono takie samo jak nazwa użytkownika), email czy adres url do zdjęcia profilowego.

- **Zmiana hasła:** Umożliwia zmianę hasła przez zalogowanego użytkownika. Dostęp do tej strony mają tylko użytkownicy zarejestrowani w standardowy sposób - w przypadku osób używających logowania poprzez Github ta funkcjonalność nie miałaby sensu.

3.2. Znajomości pomiędzy użytkownikami

Klasa `FriendshipService` odpowiada za zarządzanie relacjami pomiędzy użytkownikami. Funkcjonalności obejmują:

3.2.1. Zarządzanie znajomościami

- **Wysyłanie zaproszeń do znajomych:** Użytkownicy mogą wysyłać zaproszenia do znajomych, podając ich nazwę użytkownika.
- **Akceptacja i odrzucanie zaproszeń:** Użytkownicy mogą akceptować lub odrzucać otrzymane zaproszenia do znajomych.
- **Blokowanie użytkowników:** Umożliwia blokowanie innych użytkowników, co zapobiega otrzymywaniu od nich zaproszeń. Istnieje możliwość odblokowania.
- **Usuwanie znajomych:** Pozwala na usunięcie użytkownika z listy znajomych.
- **Wyświetlanie listy znajomych:** Użytkownicy mogą przeglądać listę swoich znajomych.
- **Wyświetlanie listy otrzymanych zaproszeń:** Umożliwia przeglądanie zaproszeń do znajomych, które nie zostały jeszcze obsłużone.
- **Wyświetlanie listy zablokowanych użytkowników:** Użytkownicy mogą zobaczyć, kogo zablokowali.

3.3. Zarządzanie podróżami

Klasa `TripService` umożliwia tworzenie i zarządzanie podróżami. Funkcjonalności obejmują:

3.3.1. Tworzenie i edycja podróży

- **Tworzenie nowej podróży:** Użytkownicy mogą tworzyć nowe podróże, określając ich nazwę, datę rozpoczęcia, lokalizację, walutę do rozliczeń oraz opis. Wszystkie akcje związane z zarządzaniem podróżą wykonać może tylko jej założyciel.

- **Edycja podróży:** Pozwala na zmianę szczegółów istniejącej podróży.
- **Dodawanie wydarzeń do podróży:** Umożliwia dodawanie wydarzeń do podróży opisanych za pomocą nazwy, daty i czasu oraz typu (pojedyncze miejsce lub przemieszczenie się pomiędzy dwoma lokalizacjami). Strona do tej funkcjonalności wykorzystuje interfejs map Google w celu pokazania lokalizacji planowanych miejsc. Użytkownik może edytować przybliżenie na mapie.
- **Usuwanie wydarzeń:** Umożliwia usuwanie wydarzeń z podróży.
- **Usuwanie podróży:** Założyciel podróży może ją usunąć.

3.3.2. Zarządzanie uczestnikami

- **Dodawanie uczestników do podróży:** Właściciel podróży może zapraszać swoich znajomych do dołączenia.
- **Usunięcie uczestników:** Właściciel podróży może usuwać uczestników.
- **Akceptacja i odrzucanie zaproszeń do podróży:** Użytkownicy mogą akceptować, bądź odrzucać zaproszenia do uczestnictwa w podróży.
- **Opuszczanie podróży:** Zwykły uczestnik może opuścić podróż. Założyciel nie może tego zrobić.

3.3.3. Wyświetlanie planowanej podróży

- **Przeglądanie listy podróży:** Użytkownicy mogą przeglądać listę wszystkich swoich podróży.
- **Dostęp do szczegółów podróży:** Użytkownicy mają dostęp do pełnych detali każdej z podróży, do której zostali zaproszeni.
- **Wyświetlanie wydarzeń podróży:** Użytkownicy mogą przejrzeć wszystkie zaplanowane wydarzenia na stronie, posortowane po dacie i czasie. Miejsca są wyświetlane z użyciem interfejsu map Google. Istnieje możliwość odtworzenia ich w postaci pokazu slajdów, użytkownik może pauzować, restartować, dostosowywać prędkość pokazu i zmieniać aktualnie wyświetlane wydarzenie.

3.4. Rozliczenia

Klasa `ExpensesService` umożliwia zarządzanie wydatkami w podróży. Funkcjonalności obejmują:

- **Dodawanie wydatków:** Użytkownicy mogą zapisywać wydatki podczas podróży. Na każdy wydatek składa się nazwa, data, kwota, nazwa użytkownika, który zapłacił oraz lista uczestników wraz z ich udziałami. Domyślnie udziały uczestników dzielone są po równo, ale można je edytować - w formularzu nowego wydatku dostępny jest tryb zaawansowany. Idealne podzielenie wydatku często nie jest możliwe - przykładowo przy kwocie 100 podzielonej na 3 uczestników nie da się tego rozdzielić po równo. Zakładam, że uczestnicy podróży nie będą się rozliczać co do ostatniego grosza. Od strony technicznej w takich sytuacjach aplikacja zapisuje minimalnie niedokładną kwotę wydatku w bazie danych, tak aby zawsze suma należności równała się kwocie wydatku. Dane odnośnie kwot zdecydowałem się zapisywać w bazie danych jako liczby całkowite odpowiadające wartościom kwot przemnożonym przez 100.

- **Przegląd wydatków:** Użytkownicy mogą przeglądać wydatki w danej podróży, z możliwością filtrowania swoich wydatków

- **Usuwanie wydatków:** Użytkownicy mogą usuwać wydatki.

- **Wykres rozliczeń:** Użytkownicy mają dostęp do wykresu analizującego finansowy balans dla każdego użytkownika. Aplikacja pokazuje również listę przelewów, które należy wykonać pomiędzy użytkownikami, tak aby wszyscy byli rozliczeni. Do stworzenia listy przelewów stosuję prosty algorytm zachłanny. Najpierw dzielę nierozliczonych użytkowników tych, którzy mają coś do oddania oraz tych, którym należy się zwrot. Następnie dopóki wszyscy nie są rozliczeni, wybieram największego pod względem kwoty użytkownika z każdej grupy i wykonuję największy możliwy przelew. Przykładowo, jeśli użytkownik `A` ma do oddania kwotę 60, a użytkownikowi `B` należy się zwrot 80, wtedy użytkownik `A` powinien oddać użytkownikowi `B` kwotę 60. Po tym przelewie usuwamy użytkownika `A` z grupy dłużników i aktualizujemy balans użytkownika `B` na kwotę 20. Za każdym przelewem usuwamy przynajmniej jedną osobę. Zakładając, że sumy kwot z obu grup są sobie równe, ostatni przelew usunie dwie osoby, zatem wykonamy co najwyżej `n-1` przelewów. Jeśli jakiś użytkownik chciałby się rozliczyć ze swoich długów, może to łatwo zrobić - wystarczy wybrać odpowiedni przelew pod wykresem rozliczeń i kliknąć 'Mark as Paid'. Użytkownik zostanie wtedy przekierowany na automatycznie uzupełnioną stronę z dodaniem nowego wydatku rekompensującego należytą wartość. Przykładowo, jeśli użytkownik `C` ma do oddania użytkownikowi `D` kwotę 50, po kliknięcu 'Mark as Paid' nastąpi przekierowanie na uzupełniony formularz z wydatkiem o kwocie 50 dla użytkownika `D` i płatnikiem ustawionym na użytkownika `C`.

Rozdział 4.

Struktura plików

W głównym folderze projektu `travel-planner` znajdziemy następujące elementy:

- `src/`: Główny folder kodu źródłowego aplikacji
- `.git/`: Repozytorium Git
- `.gitignore`: Pliki i foldery pomijane przez Gita
- `pom.xml`: Plik konfiguracyjny Mavena
- `.env`: Plik zmiennymi środowiskowymi używany przez Dockera
- `Dockerfile, docker-compose.yml`: Pliki konfiguracyjne Dockera

4.1. Struktura folderu `/src`

W folderze `/src` znajdują się wszystkie pliki związane z kodem źródłowym oraz testami aplikacji. Struktura ta jest podzielona dalej na dwie części: `main` oraz `test`.

4.1.1. main

Folder main zawiera kod źródłowy aplikacji. Zawiera on następujące elementy:

- `/java`: Główny folder dla klas Javy.
- `/resources`: Folder dla zasobów aplikacji, takich jak pliki konfiguracyjne (np. application.properties), czy pliki z rozszerzeniami .html, .css, .js, .png.

4.1.2. test

Folder test zawiera wszystkie pliki związane z testami aplikacji. Oto główne elementy w tej sekcji:

- `/java`: Główny folder dla klas testowych Javy.
- `/resources`: Folder dla zasobów testowych, takich jak pliki konfiguracyjne.

4.2. Struktura klas Javy w folderze `/src/main/java`

Klasy w projekcie języka Java podzielone są na pakiety. W tym projekcie zdecydowałem się rozpocząć od `uwr/ms`, zatem to jest główny, startowy pakiet. Tutaj znajduje się główna klasa całego projektu oznaczona adnotacją `@SpringBootApplication` oraz reszta pakietów, które są uporządkowane następująco:

- `constant`: Klasy typu enum reprezentujące stałe wartości, takie jak `FriendshipStatus` czy `EventType`
- `controller`: Springowe kontrolery obsługujące żądania HTTP
- `exception`: Klasy wyjątków używane w aplikacji
- `model`: Klasy modeli (Entity) reprezentujące tablice w bazie danych oraz ich repozytoria odpowiadające za komunikację z bazą danych
- `security`: Konfiguracja zabezpieczeń oraz usługi związane z autoryzacją
- `service`: Klasy serwisów zawierające logikę biznesową
- `util`: Narzędzia pomocnicze, np. walidacja haseł i emaili

Rozdział 5.

Model MVC

Model MVC (Model-View-Controller) jest jednym z najbardziej znanych wzorców projektowych w rozwoju aplikacji webowych. Jego głównym celem jest separacja logiki biznesowej (modelu) od interfejsu użytkownika (widoku), co ułatwia zarządzanie kodem, jego testowanie i rozwijanie. Kontroler jest pośrednikiem zarządzającym interakcjami między modelem a widokiem.

5.1. Model

Model odpowiada za strukturę danych aplikacji oraz logikę biznesową. Reprezentuje dane, na których operuje aplikacja, oraz reguły, według których te dane są przetwarzane. W aplikacji Travel Planner:

5.1.1. Klasy domenowe:

Tabele w bazie danych są reprezentowane przez klasy z adnotacją `@Entity`. Każda z tych klas odwzorowuje tabelę w bazie danych.

5.1.2. Repozytoria:

Interakcja z bazą danych odbywa się za pomocą klas typu `Repository`, które są interfejsami rozszerzającymi `JpaRepository` lub `CrudRepository` z Spring Data. Te interfejsy dostarczają metody do zarządzania danymi bez konieczności pisania podstawowych zapytań w `SQL`, co pozwala na uniknięcie problemu 'boilerplate code'. Poza podstawowymi operacjami zwanymi potocznie CRUD (z ang. create, read, update, delete) możemy w tych klasach implementować własne, bardziej złożone zapytania z użyciem JPQL. Przykładowo, jeśli chcemy znaleźć wszystkie podróże podanego użytkownika, możemy napisać następującą metodę z własnym zapytaniem:

```
@Query("SELECT DISTINCT t
FROM TripParticipantEntity tp
JOIN tp.trip t
WHERE tp.user.username = :username")
List<TripEntity> findDistinctTripsByUserUsername(
@Param("username") String username);
```

Klasa `TripParticipantEntity` posiada odwołanie do klasy `TripEntity` pod polem 'trip' oraz do klasy `UserEntity` pod polem 'user'.

5.1.3. Serwisy

Logika biznesowa jest zaimplementowana w klasach serwisowych oznaczonych adnotacją `@Service`. Serwisy te operują na modelach i repozytoriach, realizując konkretną logikę biznesową aplikacji.

5.2. Widok

Odpowiada za prezentację danych użytkownikowi. W tej aplikacji za widok odpowiada silnik szablonów `Thymeleaf`, który dynamicznie generuje pliki HTML używając danych z modelu przekazanych przez kontroler. Ponadto, pliki HTML wzbogacone są o style CSS i kod w jeżyku Javascript. Widok wyświetla dane użytkownikowi, ale nie wykonuje operacji biznesowych ani nie manipuluje danymi bezpośrednio.

5.3. Kontroler

Odpowiada za odbieranie interakcji użytkownika, ich przetworzenie (zwykle z użyciem Modelu) i zwrócenie odpowiedniego widoku jako reakcji. W tej aplikacji elementem Kontrolera są klasy z adnotacją `@Controller`, które nasłuchują zapytań HTTP typu `GET` i `POST`. Każda metoda w kontrolerze mapuje określone żądanie HTTP na logikę biznesową modelu i zwraca widok. Na przykład metoda

```
@GetMapping("/my-trips")
public String getMyTrips(Model model) {
    String username = SecurityContextHolder.getContext()
        .getAuthentication().getName();
    List<TripDTO> trips = tripService.findAllTripsByUser(username);
    model.addAttribute("trips", trips);
    return "trips/my_trips";
}
```

zdefiniowana w kontrolerze

```
@Controller
@RequestMapping("/trips")
public class TripController {...}
```

będzie oczekiwała na żądanie typu `GET` pod adresem **/trips/my-trips**, znajdzie wszystkie podróże zalogowanego użytkownika, przekaże je do widoku i zwróci stronę zdefiniowaną w pliku `my_trips.html` znajdującym się w katalogu **trips**, który znajduje się w katalogu z szablonami stron.

Rozdział 6.

Schemat bazy danych

Schemat bazy danych zaprojektowałem tak, aby wspierać zarządzanie użytkownikami, ich podróżami oraz interakcjami pomiędzy nimi, takimi jak znajomości czy zaproszenia.

6.1. Przegląd tabel

- **users** - Tabela zawiera informacje o użytkownikach. Kluczem unikalnym jest `username`
- **friendships** - Tabela ze znajomościami pomiędzy użytkownikami. Poza `id`, zawiera trzy kolumny: `requester_username`, `addressee_username`, oraz `status`. Status znajomości definiuje relację pomiędzy użytkownikami:
  - `ACCEPTED` - Użytkownicy są znajomymi;
  - `REQUESTED` - Zaproszenie do grona znajomych zostało wysłane, adresat może je zaakceptować, odrzucić lub zablokować użytkownika, który je wysłał
  - `BLOCKED` - Adresat zaproszenia zablokował osobę, która wysyłała zaproszenie
- **trip_participants** - Tabela służąca do połączenia użytkowników z podróżami. Poza `id`, zawiera trzy kolumny:
  - `username` - Nazwa użytkownika, tworzy relację wiele do jednego z `users.username`
  - `trip_id` - Identyfikator podróży; tworzy relację wiele do jednego z `trips.id`
  - `role` - Określa rolę użytkownika w podróży. Przyjmuje wartości:
    - `OWNER` - Właściciel podróży, tylko on może nią zarządzać

    * `MEMBER` - Uczestnik podróży

- **trips** - Tabela z podróżami. Poza `id`, zawiera kolumny takie, jak: `name`, `location`, `description` oraz `start_date`.
- **trip_invitations** - Tabela służąca do zarządzania zaproszeniami do podróży. Poza `id`, zawiera kolumnę `trip_id` tworzącą relację wiele do jednego z `trips.id` oraz kolumny `sender_username` i `receiver_username`
- **events** - Tabela do przechowywania informacji o wydarzeniach w trakcie podróży Zawiera `trip_id` tworzącą relację wiele do jednego z `trips.id` oraz inne informacje definiujące wydarzenie.
- **expenses** - Tabela do przechowywania informacji o wydatkach. Poza `id`, zawiera kolumny takie, jak `amount`, `date`, `title` oraz `payer_username` i `trip_id` będące w relacjach wiele do jednego z `users.username` i `trips.id`
- **expenses_participants** - Tabela do przechowywania informacji o udziałach uczestników w danym wydatku. Poza `id`, zawiera kolumny takie, jak `amount`, `expense_id` i `participant`. Kolumna `expense_id` jest w relacji wiele do jednego z `expenses.id` oraz kolumna `participant` jest w relacji wiele do jednego z `users.username`.

Rozdział 7.

Profile i pliki konfiguracyjne

W projekcie opartym na frameworku Spring możemy wykorzystać różne profile oraz pliki konfiguracyjne, aby dynamicznie dostosować zachowanie aplikacji w różnych środowiskach. Zdecydowałem się na użycie dwóch profili - `dev` oraz `docker`. Profilu `dev` używałem w trakcie rozwoju aplikacji w środowisku programistycznym Intellij Idea. Profilu `docker` używam przy konteneryzacji aplikacji.

7.0.1. application.properties

Plik ten zawiera ogólne ustawienia aplikacji, które są ładowane zawsze, niezależnie od tego, czy jakikolwiek profil jest aktywny. Ustawiam tutaj pola `client-id` oraz `client-secret`, których wartości wygenerowałem z użyciem mojego konta w serwisie Github. Dzięki temu można zalogować się do aplikacji, używając swojego konta na Githubie. Zarówno w profilu `dev`, jak i `docker` wartości czerpię ze zmiennych środowiskowych, więc te linijki umieściłem w ogólnym pliku `application.properties`. W przypadku deweloperskim w środowisku Intellij Idea zmienne środowiskowe możemy skonfigurować w opcjach uruchamiania projektu, a w przypadku z kontenerami ich wartości brane są z pliku `.env` i używane dalej w pliku `docker.compose`.

```
spring.security.oauth2.client.registration.github.client-id=
${GITHUB_CLIENT_ID}
spring.security.oauth2.client.registration.github.client-secret=
${GITHUB_CLIENT_ID}
```

7.0.2. application-dev.properties

Plik `application-dev.properties` zawiera ustawienia specyficzne dla środowiska deweloperskiego. Tutaj konfiguruję lokalne połączenie z bazą danych. Opcja `spring.jpa.hibernate.ddl-auto=update` pozwala na automatyczną aktualizację schematu bazy danych, co jest szczególnie przydatne w środowisku deweloperskim.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/mydatabase
spring.datasource.username=myuser
spring.datasource.password=mypassword

spring.jpa.hibernate.ddl-auto=update
```

7.0.3. application-docker.properties

Plik `application-docker.properties` zawiera ustawienia dla środowiska konteneryzacji. Tutaj konfiguruję połączenie z bazą danych w kontenerze, wykorzystując zmienne środowiskowe.

```
spring.datasource.url=${SPRING_DATASOURCE_URL}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD}
```

Rozdział 8.

Konteneryzacja

Zdecydowałem się na konteneryzację projektu (zarówno aplikacji, jak i bazy danych) z użyciem Dockera. Poniżej przedstawiam pliki używane do zbudowania i uruchomienia kontenerów:

8.1. Dockerfile

Plik `Dockerfile` definiuje kroki potrzebne do zbudowania obrazu aplikacji

```
FROM maven:3.8.4-openjdk-17-slim AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package -DskipTests

FROM openjdk:17-slim
COPY --from=build /home/app/target/*.jar /usr/local/lib/app.jar
COPY wait-for-it.sh /usr/local/bin/wait-for-it.sh
RUN chmod +x /usr/local/bin/wait-for-it.sh
EXPOSE 8080
ENTRYPOINT ["sh", "-c", "/usr/local/bin/wait-for-it.sh db:5432 -- java -jar /usr/loc
```

Pierwszy etap (stage) korzysta z obrazu Mavena w wersji 3.8.4 z OpenJDK 17, aby zbudować aplikację. Pomijam tutaj testy ze względu na konfigurację w projekcie wykorzystującą testowe kontenery do testów integracyjnych - ciężko je uruchomić w kontenerze. Nie oznacza to jednak, że nie przejmuję się testowaniem. Aby uruchomić wszystkie testy, wystarczy polecenie `mvn test` w katalogu głównym. Drugi etap kopiuje skompilowane pliki JAR do obrazu opartego na OpenJDK 17, aby uruchomić aplikację, która będzie nasłuchiwać na porcie 8080.

Dodatkowo, skrypt `wait-for-it.sh` [1] jest kopiowany do obrazu i używany w poleceniu `ENTRYPOINT` do upewnienia się, że baza danych jest gotowa.

8.2. docker-compose.yml

Plik docker-compose.yml definiuje usługi aplikacji i bazy danych oraz ich konfiguracje:

```
version: '3.8'
services:
    app:
        build: .
            ports:
                - "8080:8080"
            environment:
                SPRING_PROFILES_ACTIVE: docker
                SPRING_DATASOURCE_URL: ${SPRING_DATASOURCE_URL}
                SPRING_DATASOURCE_USERNAME: ${SPRING_DATASOURCE_USERNAME}
                SPRING_DATASOURCE_PASSWORD: ${SPRING_DATASOURCE_PASSWORD}
                GITHUB_CLIENT_ID: ${GITHUB_CLIENT_ID}
                GITHUB_CLIENT_SECRET: ${GITHUB_CLIENT_SECRET}
                GOOGLE_MAPS_API_KEY: ${GOOGLE_MAPS_API_KEY}
depends_on:
    - db
db:
    image: postgres:13
    environment:
        POSTGRES_DB: ${POSTGRES_DB}
        POSTGRES_USER: ${POSTGRES_USER}
        POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
        - "5432:5432"
    volumes:
        - db-data:/var/lib/postgresql/data

volumes:
  db-data:
```

Definiujemy dwie usługi: „app” dla aplikacji i „db” dla bazy danych PostgreSQL. Usługa „app” korzysta z obrazu zdefiniowanego w `Dockerfile`, przekierowuje ruch na port 8080 i konfiguruje zmienne środowiskowe, które są wykorzystywane do

konfiguracji połączenia z bazą danych, integracji z Google Maps i uwierzytelniania przez GitHub. Usługa „app” jest zależna od „db” - oczywiście najpierw chcemy mieć bazę danych, a potem dopiero uruchamiać aplikację z niej korzystającą. Usługa „db” korzysta z obrazu PostgreSQL 13, konfigurując bazę danych, użytkownika i hasło, oraz przekierowuje port 5432.

Całość można uruchomić poleceniem

```
docker-compose up --build
```

Rozdział 9.

Instalacja

9.1. Wymagania

- **Docker:** Narzędzie do tworzenia, wdrażania i uruchamiania aplikacji przy użyciu kontenerów.
- **Docker Compose:** Narzędzie do definiowania i uruchamiania aplikacji wielokontenerowych z wykorzystaniem Dockera

9.2. Budowanie i uruchamianie projektu

W głównym katalogu należy wykonać polecenie, które zbuduje wszystkie serwisy i uruchomi je w izolowanych kontenerach

```
docker-compose up --build
```

Po uruchomieniu aplikacja powinna być dostępna pod adresem `http://localhost:8080`.

9.3. Uruchamianie testów

Do uruchomienia testów potrzebne jest lokalne środowisko z zainstalowanymi:

- **Java Development Kit (JDK):** Wersja 17 lub nowsza.
- **Maven:** Wersja 3.8.6 lub nowsza.

Testy można uruchomić, wykonując w głównym katalogu następujące polecenie:

```
mvn test
```

9.4. Zakończenie pracy z aplikacją

Po zakończeniu pracy z aplikacją, można zatrzymać uruchomione kontenery, używając polecenia:

```
docker-compose stop
```

Rozdział 10.

Praca nad projektem

W tym rozdziale opiszę pracę nad całym projektem z perspektywy dewelopera.

10.1. Początek

Pracę rozpocząłem od strony `https://start.spring.io/` i wybrania interesujących mnie modułów. Dzięki temu na starcie otrzymałem poprawnie skonfigurowany plik **pom.xml** ze wszystkimi zależnościami w projekcie wykorzystującym `Spring`. Następnie przeszedłem do podstaw całego projektu, takich jak: strona do rejestracji i logowania; klasa `UserEntity` reprezentująca tabelę z użytkownikami; konfiguracja połączenia z bazą danych; konifguracja Dockera z dwoma kontenerami - jednym dla bazy danych i drugim dla aplikacji; pasek nawigacji. Gdy wszystko było poprawnie skonfigurowane, stworzyłem prywatne repozytorium na Githubie i pierwszy commit.

10.2. Szablon dla stron webowych

Aby uniknąć powtarzania kodu html w wielu plikach, przed tworzeniem właściwych stron webowych zdecydowałem się na zaprojektowanie szablonu `Thymeleaf` w pliku **main_template.html**. Poza kilkoma ogólnymi ustawieniami, jego kod sprowadza się do przechwycenia 4 argumentów (content, title, script, css) oraz wyświetlenia fragmentu z paskiem nawigacyjnym oraz fragmentu z zawartością strony:

```
<!DOCTYPE html>
    <html lang="en" xmlns:th="http://www.thymeleaf.org"
        data-th-fragment="layout(content, title, script, css)">
    <head>
    ...
    </head>
    <body>
        <div class="left-div">
            <div data-th-replace=
                "~{fragments/sidebar_fragment::content}">
            </div>
        </div>
        <div class="right-div">
            <main role="main">
                <div th:replace="${content}"></div>
            </main>
        </div>
    </body>
</html>
```

Plik **sidebar_fragment.html** definiuje fragment zawierający pasek nawigacyjny. Dzięki temu, aby stworzyć nową stronę, wystarczy zdefiniować krótki plik html odwołujący się do głównego szablonu z odpowiednimi argumentami oraz fragment z zawartością strony (opcjonalnie możemy również zdefiniować pliki css i js). Przykład ze stroną do wyświetlania podróży:

- **view_trip.html:**

```
<!DOCTYPE html>
<html lang="en" data-th-replace="~{main_template::layout(
    ~{fragments/view_trip_fragment::content},
    'View Trip',
    '/js/view_trip.js',
    '/css/view_trip.css')}">
</html>
```

- **view_trip_fragment.html:**

```
<div data-th-fragment="content">
    ...
</div>
```

Kontroler zwracający widok tej strony odwoła się do pliku **view_trip.html**.

10.3. Dodawanie nowych funkcjonalności

Dzięki solidnej podstawie całego projektu oraz szablonie stron, przy dodawaniu kolejnych funkcjonalności zachowywałem porządek w strukturze plików.

- `Nowa tabela w bazie danych:` Stworzenie nowej tabeli sprowadzało się do definicji klasy typu `Entity` reprezentującej pojedynczy wiersz w tabeli oraz klasy typu `Repository` do komunikacji z bazą danych.

- `Nowa strona:` Dodanie nowej strony wymagało zdefiniowania metody w kontrolerze nasłuchującej adres url (oraz oczywiście kodu samej strony, patrz 10.2). Przykładowo, gdy chciałem, aby pod adresem `/friends/my-friends` dostępna była strona wyświetlająca naszych znajomych, definiowałem metodę w kontrolerze w następujący sposób:

```
@Controller
@RequestMapping("/friends")
public class FriendshipController {
    ...
    @GetMapping("/my-friends")
    public String getMyFriends(Model model) {
        String username = SecurityContextHolder.getContext()
            .getAuthentication().getName();
        List<UserEntity> friends =
            friendshipService.getAllFriends(username);
        model.addAttribute("friends", friends);
        return "friends/my_friends";
    }
}
```

- `Komunikacja z backendem:` Aby przesłać dane z formularza `html` do części backendowej aplikacji, definiowałem odpowiednią metodę w kontrolerze. Przykładowo, jeśli formularz wysyłał obiekt o nazwie 'trip' pod adres `/trips/update/4` (liczba 4 jest tutaj przykładowym identyfikatorem podróży), definiowałem metodę w następujący sposób:

```
@Controller
@RequestMapping("/trips")
public class TripController {
    ...
    @PostMapping("/update/{id}")
    public String updateTrip(@PathVariable("id") Long tripId,
        @ModelAttribute("trip") TripEntity trip,
        RedirectAttributes redirectAttributes) {
        ...
    }
}
```

- `Logika biznesowa`: Aby dodać coś nowego do warstwy logiki biznesowej (przykład: zwróć listę wszystkich uczestników podróży, usuń znajomego, dodaj nowy wydatek), definiowałem nowe metody w klasach typu `Service`. Korzystały one z klas typu `Entity` i `Repository` przy komunikacji z bazą danych. Przykładowy kod tworzący nową podróż:

```
@Service
public class TripService {
    ...
    @Transactional
    public void createTrip(TripEntity trip, String ownerUsername) {
        UserEntity owner = userRepository
            .findByUsername(ownerUsername)
            .orElseThrow(() -> new IllegalArgumentException(
                String.format(Message.USER_NOT_FOUND.toString(),
                ownerUsername)));
        if (tripRepository.existsByNameAndOwnerUsername(
            trip.getName(),
            ownerUsername))
            throw new IllegalStateException(
            Message.TRIP_NAME_EXISTS.toString());
        if(trip.getName().length() > 30)
            throw new IllegalArgumentException
            (Message.TRIP_NAME_TOO_LONG.toString());
        TripParticipantEntity ownerParticipant =
            new TripParticipantEntity();
        ownerParticipant.setUser(owner);
        ownerParticipant.setRole(TripParticipantRole.OWNER);
        trip.addParticipant(ownerParticipant);
        tripRepository.save(trip);
    }
}
```

Metoda najpierw sprawdza, czy użytkownik istnieje, następnie sprawdza, czy ten użytkownik nie stworzył wcześniej podróży z taką samą nazwą oraz czy nazwa nie jest zbyt długa. Następnie tworzy się obiekt klasy `TripParticipantEntity` (`TripParticipantEntity` jest pomiędzy UserEntity oraz TripEntity, które są w relacji wiele do wielu). Na końcu zapisuję obiekty klas typu `Entity`, co skutkuje utworzeniem nowych rekordów w bazie danych. Metody serwisów są wywoływane przez kontrolery.

10.4. Testowanie

Po implementacji nowej funkcjonalności pisałem testy sprawdzające metody w klasach serwisowych. Przykładowo, aby sprawdzić, czy metoda usuwająca znajomego poprawnie modyfikuje bazę danych, napisałem następującą metodę w klasie testującej:

```
@SpringBootTest
@Transactional
@ActiveProfiles("postgres")
public class FriendshipServiceTest {
    ...
    @BeforeEach
    void setup() {
        // tutaj tworzę użytkowików user1, user2, user3 oraz znajomości
        // pomiędzy user1 i user2 oraz pomiędzy user1 i user3
    }

    @Test
    void removeOneFriendUpdatesFriendList() {
    friendshipService.deleteFriend(user1.getUsername(),
        user3.getUsername());
    List<UserEntity> updatedFriends = friendshipService
        .getAllFriends(user1.getUsername());

    assertThat(updatedFriends)
        .hasSize(1)
        .extracting(UserEntity::getUsername)
        .containsExactly(user2.getUsername());
    }
```

Metoda `removeOneFriendUpdatesFriendList()` usuwa znajomość z użytkownikiem user3 z perspektywy user1, następnie pobiera listę wszystkich znajomych

użytkownika user1. Jeśli user3 został poprawnie usunięty ze znajomych, lista ta powinna zawierać jedynie użytkownika user2.

10.5. Konfiguracja interfejsu map Google

Integrację z mapami rozpocząłem od stworzenia nowego projektu w Google Cloud i wygenerowaniu klucza. Następnie korzystając z dokumentacji [6] przeszedłem do implementacji. Klucz w części backendowej pobieram ze zmiennej środowiskowej. Oto fragment kodu kontrolera podróży:

```
@Controller
@RequestMapping("/trips")
public class TripController {
    @Value("${GOOGLE_MAPS_API_KEY}")
    private String googleMapsApiKey;
    ...
}
```

Zmienną `googleMapsApiKey` umieszczam w widoku strony zawierającej mapy. W kodzie `html` definiuję diva, który będzie zawierał mapę:

```
    <div id="mapContainer" th:data-api-key="${googleMapsApiKey}"
        class="map-container"></div>
```

Do poprawnego działania potrzebowałem jeszcze skryptu od map Google używającego mojego klucza:

```
function loadGoogleMapsScript(callback) {
    var map = document.getElementById('mapContainer');
    var apiKey = map.getAttribute('data-api-key');
    const script = document.createElement('script');
    script.src = 'https://maps.googleapis.com/maps/api/js?key=' + apiKey +
        '&loading=async' + '&libraries=places' +
        '&callback=initializeFirstEvent';
    script.id = 'googleMapsScript';
    document.head.appendChild(script);
}

document.addEventListener('DOMContentLoaded', function() {
    loadGoogleMapsScript();
    ...
}
```

Dzięki temu z poziomu Javasciptu mogłem łatwo ładować mapę z odpowiednimi argumentami używając funkcji ze skryptu map Google. Przykładowy kod ładujący mapę można znaleźć w pliku `view_trip.js`, linia 75.

Rozdział 11.

Podręcznik użytkownika

11.1. Strona startowa dla niezalogowanych użytkowników

- **Adres:** `localhost:8080/login`
- **Logowanie:** Wprowadź swoją nazwę użytkownika i hasło, aby się zalogować.
- **Rejestracja:** Kliknij przycisk *Sign up*, aby przejść do formularza tworzenia nowego konta.
- **Logowanie przez Githuba:** Kliknij przycisk *Continue with GitHub*, aby uwierzytelnić się przy użyciu konta GitHub.

11.2. Strona startowa dla zalogowanych użytkowników

Tutaj (localhost:8080) następuje przeniesienie po pomyślnym zalogowaniu. Po lewej stronie znajduje się pasek nawigacyjny zawierający linki do reszty stron. W dalszej części podręcznika zakładam, że użytkownik jest zalogowany.

11.3. Zarządzanie profilem użytkownika

- **Edycja profilu** Account → Edit Profile
- **Zmiana hasła** Account → Change Password, hasło musi mieć długość przynajmniej 8 znaków, zawierać dużą literę, małą literę oraz cyfrę
- **Usuwanie konta** Account → Edit Profile → Delete Account, następnie potwierdź swoje hasło
- **Wylogowanie** Account → Sign out

11.4. Zarządzanie znajomościami

- **Dodawanie znajomych** Friends → Add friend
- **Zarządzanie zaproszeniami** Friends → Requests, zaproszenie do znajomych można zaakceptować albo odrzucić. Istnieje możliwość zablokowania, co zablokuje wysyłającemu zaproszenie możliwość ponownego wysłania.
- **Lista znajomych** Friends → My friends

11.5. Podróże

- **Tworzenie nowej podróży** Trips → New trip
- **Lista podróży** Trips → My trips
- **Prezentacja podróży** Trips → My trips, następnie znajdź podróż, którą chcesz obejrzeć i kliknij View. Na wyświetlonej stronie w zakładkach 'Trip Info' i 'Participants' znajdziesz podstawowe informacje o podróży i jej uczestnikach. W zakładce 'Events' znajdziesz uporządkowaną listę wszystkich zaplanowanych wydarzeń z interfejsem map Google. Na dole pojawi się możliwość przeglądu wszystkich wydarzeń w postaci pokazu slajdów.
- **Edycja podróży** Trips → My trips, następnie znajdź podróż, którą chcesz edytować i kliknij Edit. Edycja jest dostępna tylko dla założyciela podróży.
  - **Edycja podstawowych informacji** W zakładce 'Edit Trip' możesz edytować podstawowe informacje na temat podróży, takie jak nazwa, data, lokalizacja, waluta do rozliczeń i opis.
  - **Zarządzanie uczesntikami** W zakładce 'Current Participants' znajdziesz aktualną listę uczestników podróży z możliwością usunięcia niechcianych osób. W zakładce 'Add Participant' znajdziesz listę swoich znajomych, których nie ma w aktualnie edytowanej podróży. Pole do wyszukiwania po nazwie jest interaktywne - na bieżąco filtruje listę. Aby wysłać znajomemu zaproszenie do podróży, zaznacz go na liście (jego wiersz podświetli się na zielono oraz jego), a następnie wciśnij przycisk 'Invite to the trip!'.
  - **Edycja wydarzeń** W zakładce 'Edit Events' znajdziesz aktualną listę wydarzeń z możliwością usuwania. Zakładka 'Add New Event' zawiera formularz z interfejsem map Google. Użytkownik może zobaczyć na mapie podgląd planowanego miejsca lub trasy i wybrać przybliżenie, z jakim zobaczą je inni uczestnicy podróży.
- **Opuszczenie podróży** Trips → My trips, następnie znajdź podróż, którą chcesz opuścić i kliknij 'Exit'. Założyciel podróży nie może jej opuścić.

- **Usuwanie podróży** Trips → My trips, następnie znajdź podróż, którą chcesz usunąć i kliknij 'Edit'. Następnie kliknij 'Delete trip'.
- **Rozliczenia** Trips → My trips, następnie znajdź podróż, dla której chcesz edytować wydatki i kliknij 'Expenses'.
  - **Przegląd wydatków** Lista wydatków dostępna jest w zakładce 'Expenses'. Po kliknięciu na konkretny wydatek rozwinie się lista uczestników wraz z ich udziałami w danym wydatku oraz przycisk do usunięcia wydatku.
  - **Rozliczenia** Wykres rozliczeń i lista przelewów potrzebna do wyrównania wszystkich należności dostępne są w zakładce 'Balances'. Jeśli chcemy zapisać oddanie konkretnej należności, wybieramy dany przelew - pojawi się opcja 'Mark as Paid', która przekieruje nas na wypełniony formularz z wydatkiem rekompensującym należność.
  - **Dodanie wydatku** Aby dodać wydatek, przechodzimy do zakładki 'Add Expense'. Uzupełniamy tytuł, kwotę, datę oraz płatnika. Z listy osób wybieramy uczestników wydatku. Domyślnie udziały dzielą się po równo, jednak możemy wybrać tryb zaawansowany, w którym można edytować udziały.

Bibliografia

[1] Vishnu Bob, *wait-for-it*, 2020. https://github.com/vishnubob/wait-for-it/blob/master/wait-for-it.sh.

[2] Spring Framework Documentation, *Spring Framework Reference Documentation*, 2024. https://docs.spring.io/spring-framework/docs/current/reference/html/.

[3] Java SE Documentation, *Java Platform, Standard Edition Documentation*, 2024. https://docs.oracle.com/en/java/javase/17/docs/.

[4] Docker Documentation, *Docker Documentation*, 2024. https://docs.docker.com/.

[5] Spring Boot Documentation, *Spring Boot Reference Documentation*, 2024. https://docs.spring.io/spring-boot/docs/current/reference/html/.

[6] Google Maps API Documentation, *Google Maps Platform Documentation*, 2024. https://developers.google.com/maps/documentation.