POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH

PRACA DYPLOMOWA INŻYNIERSKA

Aplikacja mobilna ułatwiająca podróżowanie samochodami elektrycznymi

Mobile application that facilitates traveling by electric cars

AUTOR:
Nikita Stepanenko

PROWADZĄCY PRACĘ:
dr inż. Tomasz Kubik, K30W04D03

OCENA PRACY:

WROCŁAW, 2020

Spis treści

Spis rysunków

Spis listingów

Spis tabel

Skróty i definicje

API (ang. *Application Programming Interface*) jest to interfejs programowania.

JSON (ang. *JavaScript Object Notation*) jest tekstowym formatem do wymiany danymi, którego podstawą jest Java Scrypt.

BSON (ang. *Binary JavaScript Object Notation*) Binarny zapis formata JSON.

JWT (ang. *JSON Web Token*) jest standardem tworzenia tokenów dostępu.

SQL (ang. *Structured Query Language*) Jest językiem programowania strukturalnych zapytań. Najczęściej używa się do skutecznego zapisywania danych, wyshukiwania, zmiany, pobieania oraz usuwania danych z bazy.

NoSQL (ang. *Not only SQL*) Szereg podejść, których celem jest tworzenie systemów zarządzania bazami dabych, któte mają dużą różnicę w porównaniu do modeli tradycyjnych relacyjnych baz dabych.

SDK (ang. *software development kit*) Są narzędziami programistycznymi, umozliwiającymi programistom tworzenie aplikacji do określianego pakietu oprogramowania. Przyznaczony do użatwiania i przespiesznia pracy z systemem.

Rest (ang. *Representational state transfer*) To styl achitektury oprogramowania przyznaczony do systemów rozproszonych. Zwykle używany do budowy usług internetowych.

URL (ang. *Uniform Resource Locator*) Jest standardem zapisu linków do obiektów w internecie.

YAML (ang. *Yet Another Markup Language*) Język znaczników.

UUID (ang. *universally unique identifier*) Standatd identyfikacji stosowany w tworzeniu oprogramowania.

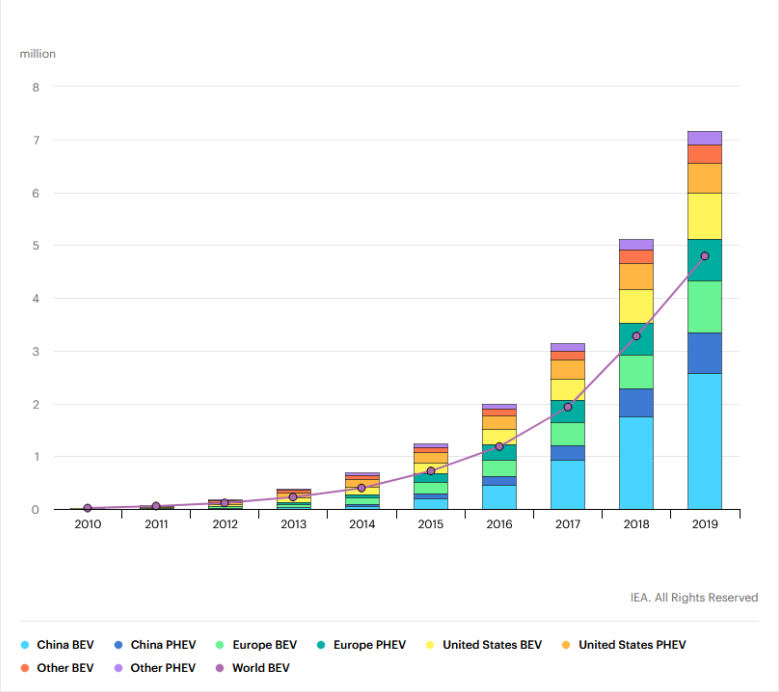
HTTP (ang. *HyperText Transfer Protocol*) Protoków wymiany danymi (w pierwszej kolejności hipertekstu).

Rozdział 1

Wstęp

1.1. Wprowadzenie

Ochrona Ziemi przed zgubnymi skutkami zanieczyszczeń, właściwa gospodarka odpadami, racjonalne zużycie wody – to wyzwania, którym ludzka cywilizacja musi czym prędzej stawić czoło. Od wielu już lat podkreślają to różne organizacje oraz społeczności. Za tymi wyzwaniami ukryte są jednak liczne problemy, które spowalniają czy też utrudniają podejmowanie pro-ekologicznych działań. Dotyczą one technologii, kultury, polityki i innych obszarów. Wśród nich szczególne miejsce zajmują problemy motoryzacji. Od dawna podkreśla się negatywny wpływ emisji gazów cieplarnianych oraz wyników spalania paliw na ekologię Ziemi. Zahamowanie trwającej od kilku dziesięcioleci tendencji powinno nastąpić w wyniku zastosowania w samochodach napędów hybrydowych lub czysto elektrycznych. Według różnych źródeł liczba samochodów elektrycznych rośnie z każdym rokiem i ten wzrost powinien utrzymać się przez długi czas (rys. 1.1) [14, 9]. Wraz z nimi rośnie potrzeba rozbudowy infrastruktury pozwalającej na szybkie uzupełnianie w tych samochodach energii.



Rys. 1.1: Przyrost liczby samochodów elektrycznych w różnych krajach w osi czasu [10]

W krajach WNP (Wspólnoty Niepodległych Państw), i być może nie tylko w nich, sporym problemem okazuje się znalezienie stacji do ładowania samochodów elektrycznych. Zdarzają się sytuacje, w których po przybyciu do stacji ładowania stacja ta okazuje się nieczynna lub położona w niedostępnym miejscu (na przykład na prywatnym terenie), a przewody umożliwiające podłączenie się są za krótkie. Dlatego nasuwa się pytanie, czy nie dałoby się jakoś poinformować właścicieli elektrycznych samochodów o tym gdzie i na jakich warunkach będą oni mogli doładować swoje pojazdy.

Niniejsza praca jest próbą udzielenia odpowiedzi na takie pytanie. Jej temat zredagowano z myślą o stworzeniu aplikacji, która pomogłaby znaleźć działające i dostępne stacje ładowania uwzględniając bieżące położenie elektrycznego pojazdu. Ponieważ zwykle kierujący pojazdem człowiek najczęściej posługuje się telefonem (jest on znacznie poręczniejszy niż komputer, laptop czy tablet) skupiono się na aplikacji mobilnej. Biorąc pod uwagę rankingi popularności systemów na urządzenia mobilne zdecydowano, że aplikacja ta przeznaczona będzie dla urządzeń z systemem Android [25].

1.2. Cel i zakres pracy

Celem pracy jest zaprojektowanie i zaimplementowanie aplikacji mobilnej z połączeniem internetowym, pozwalającej ułatwić wyszukiwanie miejsc do ładowania samochodów elektrycznych. Aplikacja ta powinna umożliwiać gromadzenie danych o stacjach ładowania, wystawianie o nich opinii, jak również powinna pozwalać na przeglądanie zgromadzonych informacji.

W ramach pracy powstać ma projekt aplikacji webowej, z wyróżnionymi częściami: aplikacją mobilną, usługą sieciową oraz bazą danych. Aplikacja mobilna ma służyć do komunikacji użytkownika z usługą sieciową oraz do wizualizacji treści. Aplikacja sieciowa zaś ma przetwarzać gromadzone informacje i zarządzać bazą danych, w której informacje te będą składowane.

Dla implementacji aplikacji mobilne zaplanowano wykorzystać narzędzie Android Studio oraz język Java, dla usługi sieciowej — język Go. Dla przechowywania danych została wybrana MongoDB.

1.2.1. Układ pracy

Praca składa się z pięciu rozdziałów. W pierwszym znajduje się ogólna informacja o pracy. W drugim skupiono się na wymaganiach dotyczących mającej powstać aplikacji oraz na wykorzystanych technologiach. Trzeci rozdział przeznaczono na opis szczegółów implementacji poszczególnych części aplikacji. W czwartym rozdziale opisano zagadnienie testowania stworzonej aplikacji. W ostatnim, piątym rozdziale zamieszczono podsumowanie zawierające wnioski oraz plany dalszego rozwoju aplikacji.

Rozdział 2

Założenia projektowe

W rozdziale opisano główny problem oraz potencjalne ścieżki prowadzące do jego rozwiązania. Zamieszczono w nim również wynik przeprowadzonej analizy wymagań funkcjonalnych i niefunkcjonalnych dla mającej powstać aplikacji. Zaproponowano ponadto makiety interfejsu użytkownika oraz przedstawiono narzędzia i technologie wybrane do realizacji celu pracy.

2.1. Analiza biznesowa

Podczas podróży pojazdami elektrycznymi sporym problemem jest znalezienie stacji ładowania, w której można byłoby doładować akumulatory. Znając nawet położenie takich stacji nie ma żadnej pewności, że stacje te aktualnie działają, że zapewniony jest do nich dostęp czy też posiadają odpowiednie oprzyrządowanie. Większość istniejących aplikacji mobilnych wspierających użytkowników w takich kwestiach najczęściej gromadzi dane dotyczące stacji konkretnych firm, bez żadnych wskazówek co do lokalizacji stacji ładowania konkurentów. Co więcej, nawet te dane, które są oferowane, bywają nieaktualne, nie mówiąc już o braku ocen wystawianych przez użytkowników. Problemy te można byłoby rozwiązać za pomocą aplikacji mobilnej. Wystarczyłoby, by aplikacja ta wyświetliła bieżącą pozycję użytkownika na mapie oraz pokazała stacje znajdujące się w pobliżu wraz z możliwością podglądu opinii na ich temat wystawionych przez innych użytkowników.

Aplikacja ta w szczególności mogłaby wspierać kierowców pojazdów elektrycznych w odnajdowaniu najbliższych stacji ładowania, pomagając w łatwy i przyjazny sposób wybrać stację najlepszą. Właściciele mogliby dostarczać informacji o swoich stacjach, co może robiliby z chęcią z uwagi na ich dodatkową reklamę.

Aby stworzyć jakiś ranking stacji czy też obiektywnie oceniać ich jakość należałoby wprowadzić możliwość tworzenia aktualnych komentarzy wraz z wystawianiem oceny przez zarejestrowanych użytkowników. Każdy zarejestrowany użytkownik musiałby wtedy posiadać konto w systemie, by można było autoryzować jego działania.

Wdrożenie takiego systemu mogłoby przyczynić się do poprawy jakości usług świadczonych na stacjach ładowania pojazdów elektrycznych. Jednak aby ten cel osiągnąć w pełni, potrzebne byłoby uzyskanie wsparcia od społeczności użytkowników.

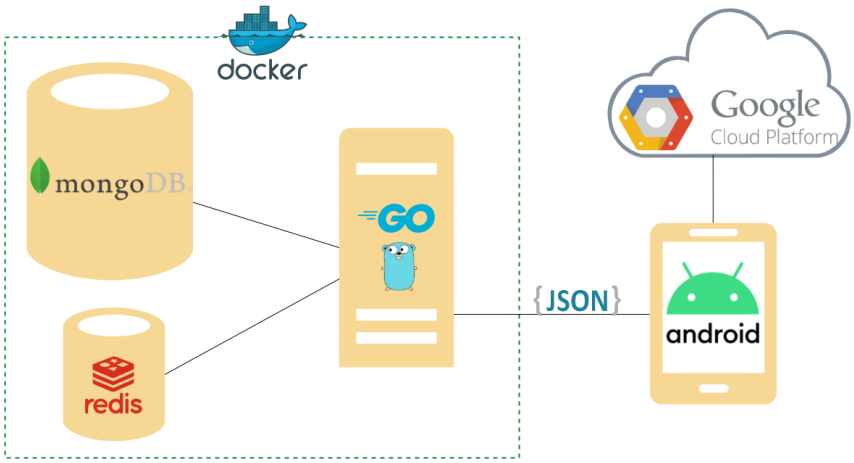
2.2. Zarys architektury systemu

Aplikacja mobilna powinna oferować możliwość korzystania z serwisu mapowego ułatwiającego wyszukiwanie oraz lokalizowanie wskazanych miejsc. Należy wdrożyć system rejestracji i logowania. Oprócz interfejsu użytkownika musi być zaimplementowana część serwerowa do obsługi działań użytkowników. A także baza danych do przechowywania stacji i komentarzy.

Różni użytkownicy muszą mieć dostęp do tych samych danych. Oznacza to, że powinna istnieć jedna wspólna baza danych, do którego użytkownicy będą uzyskiwać dostęp przez Internet. Użytkownicy nie mogą mieć bezpośredniego dostępu do bazy danych, jedynie poprzez część serwerową.

Przedstawioną koncepcję można zrealizować w architekturze składającej się z następujących elementów (rys. 2.1):

- MongoDB – baza danych do ciągłego przechowywania danych systemu;
- Redis – baza danych, która jest wykorzystywana w jakości pamięci podręcznej do przechowywania JWT tokenów po wylogowaniu;
- Go – część serwerowa napisana w języku Go. Umożliwia komunikację między interfejsem użytkownika a baza danych. Pozwala na autentykację użytkowników oraz obliczeń biznesowych;
- Android – aplikacja mobilna. Ta część przeznaczona do ułatwiana zdalnego dostępu (przez sieć Internet do endpointów Rest API serwera) użytkownikowi do systemu. Komunikacja zachodzi przez internet do interfejsów programistycznych części serwerowej oraz serwisów Google Cloud Platform;
- Docker – kontenery dla wdrożenia części ukrytych od użytkownika na zdalnym urządzeniu: baz danych oraz części serwerowej. Trzy kontenery powiązane między sobą;
- Google Cloud Platform – zdalna platforma serwisów, stworzona przez Google, która umożliwi dostęp do pracy z mapą za pomocą wbudowanego API.

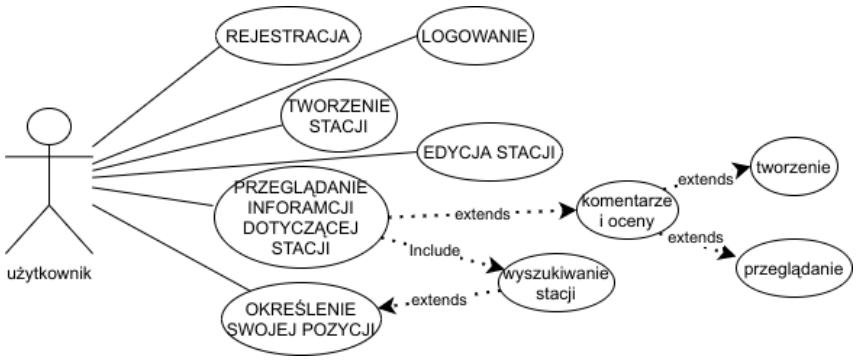


Rys. 2.1: Zarys architektury systemu

2.3. Analiza wymagań

2.3.1. Wymagania funkcjonalne

Wymagania funkcjonale definiują oczekiwany zakres funkcji dostarczanych przez tworzony system oraz jego zachowania. Wymagania te można definiować w języku naturalnym bądź też w języku formalnym. W niniejszej pracy wymagania przedstawiono na diagramie przypadków użycia (patrz rysunek 2.2) oraz opisano tabelarycznie cel ich wdrożenia (patrz tabela 2.1). Na diagramie przypadków użycia zwykle przedstawia się użytkowników w powiązaniu z dostępnymi dla nich funkcjami. W budowanej aplikacji istnieje tylko jeden typ użytkownika.



Rys. 2.2: dagram przypadków uzycia

Tab. 2.1: Wymagania funkcjonalne

№	Przypadek	Cel
1	Logowanie	Pozwala użytkownikowi zalogować się do własnego konta
2	Rejestracja	Pozwala na stworzenie konta (w bazie danych pojawia się nowy wpis) i otwiera możliwość zalogowania się
3	Tworzenie stacji ładowania	Pozwala na dodanie nowej stacji do serwisu (pojawia się nowy wpis w bazie danych)
4	Edycja stacji ładowania	Właściciel stacji (twórca) ma możliwość edycji wpisu w bazie danych
5	Przeglądanie informacji dotyczącej stacji ładowania	Pozwala na wczytanie danych wybranej stacji ładowania z bazy danych oraz wyświetlenie ich użytkownikowi
6	Określenie swojej pozycji	Pozwala użytkownikowi pobrać współrzędne swojej pozycji na mapie Ziemi dla wykorzystania podczas wyszukiwania oraz możliwość wyświetlania tej pozycji na mapie
7	Wyszukiwanie stacji ładowania	Pozwala użytkownikowi na przeglądanie listy stacji ładowniczych (pobranej z bazy danych) znalezionych według parametrów wyszukiwania
8	Komentarze i oceny: tworzenie	Pozwala na dodanie komentarza i oceny dotyczącej pewnej stacji do bazy danych
9	Komentarze i oceny: przeglądanie	Pozwala na wczytanie danych komentarzy o stacji z bazy danych oraz wyświetlenie ich użytkowników

2.3.2. Wymagania niefunkcjonalne

W tabeli 2.2 przedstawiono wymagania niefunkcjonalne: krótki opis wymagania, typ (jakiej dziedziny to dotyczy) oraz uwagi do tego wymagania.

Tab. 2.2: Wymagania niefunkcjonalne

№	typ	Opis	Uwagi
1	Bezpieczeństwo	Hasła muszą być przechowywane w bezpieczniej formie	Hasła nie przechowają się w pierwotnej formie, tylko w formie zaszyfrowanej
2	Bezpieczeństwo	Małe ryzyka przechwytywania hasła przez trzecich osób podczas komunikacji interfejsu użytkownika i części serwerowej	Autentykacja użytkownika na serwerze powinna zachodzić za pomocą tokenów
3	Przenoszalność	wdrożenie systemu powinno być szybkie i łatwe	
4	Konfigurowalność	Zmiana ustaleń bez konieczności rekompilacji części serwerowej	Zarządzaniem portu na którym działa część serwerowa, wymiana adresu oraz nazwy kolekcji bazy danych, musi być możliwa za pomocą pliku konfiguracyjnego
5	Estetyczne	Przyjazny interfejs użytkownika	Intuicyjny i zrozumiały interfejs
6	Ergonomia	Interfejs w języku angielskim	Internacjonalizacja aplikacji
7	Przenoszalność	Aplikacja mobilna musi działać na 90% lub więcej telefonów pracujących na systemie Android	Możliwość korzystania z aplikacji przez dużą ilość ludzi
8	Estetyczne	Kolory muszą być odpowiednio dopasowane	Wykorzystanie palety z małą liczbą odpowiednich kolorów
9	Wydajność	Płaynna reakcja systemu	Użytkownik kontynuuje współpracę z systemem podczas ładowania danych z serwera
10	Informatywność	System powinien powiadomiać o błędach i sukcesach	Podczas zarządzania systemem użytkownik powinien zawsze wiedzieć jaki wynik jego działania
11	Ergonomia	Wykorzystanie z aplikacji mobilnej za pomocą jednej ręki	Przyciski muszą znajdować się w dolnej części ekranu lub znajdować się w dostępnym miejscu
12	Estetyczne	Wszystkie elementy muszą być w jednym stylu	Niedopuszczalne jest użycie różnych czcionek oraz ciągłej zmiany kolorów
13	Dostęp	Dane o stacjach ładowniczych oraz komentarzy przechowywane zdalnie	Różni użytkownicy miały do nich dostęp do tej samej bazy danych
14	Dostęp	Użytkownicy nie mają bezpośredniego dostępu do bazy danych	Wszystkie zapytania muszą być opracowanie przez część serwerową
15	Dostęp	Część serwerowa zrobiona zgodnie z regułami Rest (ang. *Representational State Transfer*) API (ang. *Application Programming Interface*)	Przestrzeganie się Best Prakties Rest API [23]]
16	Ergonomia	Dla realizacji mapy wykorzystuję się Google Maps	Ułatwia rozumienie interfejsu dla użytkowników

2.3.3. Narzędzia i technologie

Do stworzenia aplikacji mobilnej, części serwerowej oraz bazy danych wybrano następujące narzędzia i technologie:

- Visual Studio Code,
- Go,
- REST API,
- JWT,
- MongoDB,
- Redis,
- Android Studio,
- Gradle,
- Android SDK,
- Java JDK 8,
- RxJava 2,
- Google Cloud Platform,
- GMS,
- Docker.

Visual Studio Code – to darmowy edytor kodu źródłowego wyprodukowany przez firmę Microsoft. Działa na systemach Windows, Linux, macOS. Wspiera on różne języki programowania oraz posiada możliwość rozszerzeń dzięki mechanizmowi plug-inów. Wśród jego funkcji można wymienić: podświetlanie składni, IntelliSense, refaktoryzacja, debugowanie i inne. [12]

Go (golang) – jest językiem programowania stworzonym przez firmę Google. Pozwala tworzyć aplikacje wielowątkowe kompilowane dla systemów Linux, Windows, macOS, FreeBSD, Android i innych. Język jest przeznaczony do budowania serwisów działających efektywnie pod wysokim obciążeniem w środowisku rozproszonym i z wielowątkowymi procesorami. [7, 15, 26, 24]

REST API (ang. *Representational State Transfer Application Programming Interface*) – jest podejściem stosowanym w tworzenia interfejsów programistycznych aplikacji webowych. Obowiązują w nim następujące zasady: komunikacja w architekturze klient-serwer (serwer nasłuchuje na żądania wysyłane przez klienta oraz udziela na nie odpowiedzi), brak stanu (serwer nie przechowuje stanu klienta, wszystkie potrzebne informacje są wysyłane wraz z żądaniem), buforowanie (buforowanie danych jest dozwolone, jeśli w odpowiedzi jest na to zgoda), jednorodność interfejsu (ograniczenia stylu pisania), wielopoziomowość systemu (komponenty systemu mają bezpośredni dostęp tylko do sąsiednich warstw), łatwość rozszerzenia funkcjonalności (opcjonalnie). [29, 23]

JWT (ang. *JSON Web Token*) – jest standardem wykorzystywanym do tworzenia tokenów dostępu. Za pomocą JWT sprawdza się, czy wchodzące dane zostały wysłane przez autoryzowane źródło. Wykorzystywany model informacyjny składa się z trzech części: nagłówka (informacja o tym, w jaki sposób odszyfrować token), danych (zaszyfrowane dane) oraz sygnatury. Każdy token ma określony okres ważności. Nie może być oznaczony jako nieważny, dopóki ten okres się nie zakończy. [18]

MongoDB – jest dokumentową bazą danych zaliczaną do rozwiązań typu NoSQL (ang. *Not only Structured Query Language*). Formatem przechowywania danych jest BSON (ang. *Binary JavaScript Object Notation*). Działa na licencji SSPL (ang. *Server Side Public License*). Wykorzystuje technikę segmentacji obiektów bazy danych, co pozwala na bilansowanie obciążenia. [28, 11, 4]

Redis – ten system zarządzania bazami danych zaliczany do rozwiązań typu NoSQL, operujący na strukturach typu „klucz-wartość”. Najczęściej używa się do implementacji baz danych w pamięci podręcznej, głównie brokerów wiadomości. [8]

Android Studio – to zintegrowane środowisko programistyczne stworzone przez firmę JetBrains na podstawie IntelliJ IDEA, służące do budowania aplikacji na platformie Android. Środowisko to jest dostępne do systemów Windows, Linux, macOS na bezpłatnej licencji Apache 2.0. [3, 2]

Gradle – jest systemem automatycznego budowania aplikacji oraz zarządzania zależnościami w aplikacjach rozwijanych w języku Java. [1, 5]

Android SDK – jest pakietem narzędzi dostarczonym przez firmę Google. Służy do tworzenia aplikacji mobilnych działających pod kontrolą systemu Android. W skład SDK (ang. *Software Development Kit*) wchodzą: zestaw bibliotek Android i Java, emulator telefonu, debugger, dokumentacja, szablony prostych aplikacji. [2]

Java JDK 8 – jest zestawem narzędzi programistycznych stosowanym do tworzenia oprogramowania na platformie Java. JDK (ang. *Java Development Kit*) zawiera: kompilator języka Java, przykłady, dokumentację, środowisko uruchomieniowe JRE (ang. *Java Runtime Environment*). Java jest obiektowym językiem ogólnego zastosowania, kompilowanym do niezależnego od systemu operacyjnego kodu bajtowego uruchamianego na maszynie wirtualnej. Powstał w firmie Sun Microsystems, którą przejęła firma Oracle. [19] Do budowy aplikacji wykorzystano JDK w wersji 1.8 (nie jest to najnowsza wersja, ale z uwagi na duże wsparcie i liczne zastosowania pozostaje wciąż popularna).

RxJava 2 – jest biblioteką pozwalającą na stosowanie reaktywnego programowania w języku Java. Programowanie reaktywne jest paradygmatem programowania pozwalającym na posługiwanie się asynchronicznymi strumieniami danych (sekwencjami zdarzeń uporządkowanymi według czasu). Biblioteka ta przydaje się do zapewnienia ciągłości funkcjonowania interfejsu użytkownika nawet w podczas realizacji złożonej logiki obliczeniowej, na przykład podczas oczekiwania na odpowiedzi serwera.

Google Cloud – jest pakietem usług udostępnianych w chmurze Google. Do tego pakietu należą znane usługi, jak: Google Search, Gmail, Google Drive, Youtube. Ponadto w ramach tego pakietu możliwe są obliczenie i przechowywanie danych, uczenie maszynowe i inne usługi, które Google wykorzystuje w swoich projektach. [16] Usługa ta jest odpłatna, a wysokość opłat zależy od wybranego planu biznesowego oraz stopnia wykorzystania chmury. Jeśli koszty użycia serwisów kształtują się na poziomie niższym niż 200 dolarów miesięcznie, wtedy opłaty nie są egzekwowane przez Google. W tej pracy inżynierskiej wykorzystano serwis Static Maps [22]. Jest on darmowy przy tworzeniu oprogramowania do telefonów z wykorzystaniem Android Maps SDK for Android [20].

GMS (ang. *Google Mobile Services*) – to zestaw najpopularniejszych aplikacji i API, jak Goole Maps, Google Paly Store, Gmail Google Drive Google Duo, Google Chrome, Google Photos, Google TV, Youtube, Youtube Music adresowanych przez Google na urządzenia mobilne.

Docker – jest oprogramowaniem przeznaczonym do automatyzacji wdrażania i zarządzania aplikacjami. Aplikacje uruchamiają się w kontenerach. Docker pozwala na upakowanie aplikacji oraz wszystkich zależności w niezależny od podstawowego systemu kontener, który może być łatwo przeniesiony na dowolny *nix system. [27, 21]

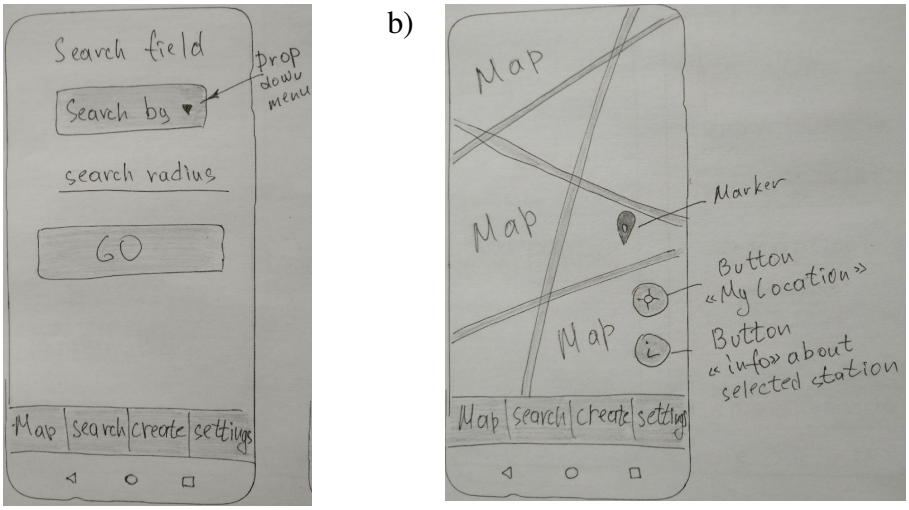
diagrams.net – Jest owartą platformą do tworzenia diagramów do różnego rodzaju aplikacji [6]. Została wykorzystana dla tworzenia dyagramów w tej pracy.

2.3.4. Makiety interfejsu użytkownika

Po analizie wymagań zaprojektowano makiety interfejsu użytkownika aplikacji. Aplikacja będzie posiadać na dole ekranu główne menu zawierające 4 pozycje: mapę, wyszukiwanie stacji, tworzenie stacji i parametry. Wybierając jedną z tych pozycji będzie można przełączać się między różnymi widokami.

Na rysunku 2.3a przedstawiono widok wyszukiwania stacji ładowania. Pod polem wyszukiwania znajduje się rozwijane menu, które pozwala wybierać typ wyszukiwania, w tym wyszukiwanie na podstawie pozycji użytkownika oraz nazwy i opisu stacji. Poniżej znajduje się możliwość do zmiany dystansu wyszukiwania oraz przycisk wyszukiwania.

a) b)



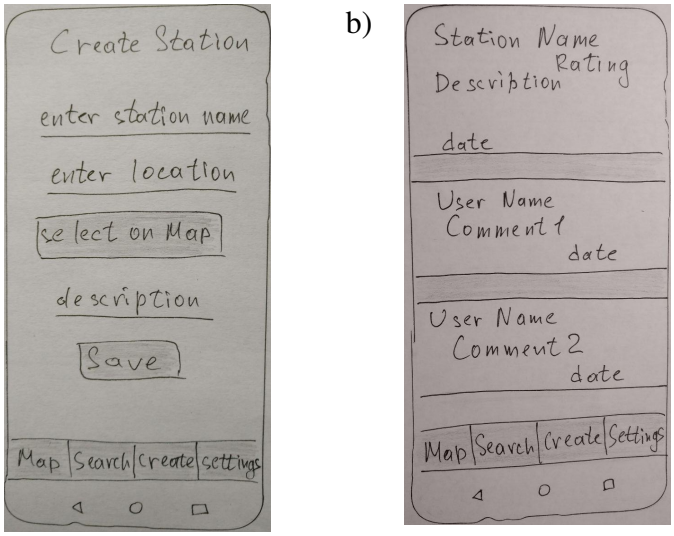
Rys. 2.3: Makiety interfejsów użytkownika: a) strona wyszukiwania stacji ładowania, b) strona główna.

Na rysunku 2.3b przedstawiono makietę pozwalającą dokonać wyboru stacji ładowania. Położeniom stacji odpowiada odpowiedni znacznik na mapie. W celu otrzymania informacji o wybranej stacji (dla danego markera na mapie) oraz wyświetlania pozycji użytkownika wykorzystują się okrągłe przyciski z lewej strony ekranu.

Podczas tworzenia stacji ładowania dla wygody użytkowania można wybrać miejsce na mapie poprzez odpowiedni przycisk (rys. 2.4a). Na stronie informacji o stacji oprócz nazwy stacji, opisu oraz oceny, wyliczonej na podstawie komentarzy oceny, znajdują się komentarzy użytkowników uporządkowane od najnowszego (rys.2.4b) .

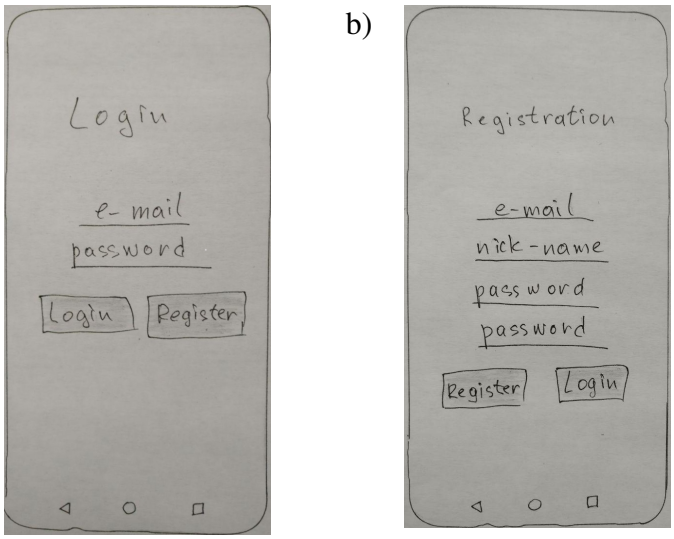
Do rejestracji i logowania wykorzystują się dwa różne ekrany pokazane, odpowiednio, na rysunku 2.5a oraz rysunku 2.5b)

a) b)



Rys. 2.4: Makiety interfejsów użytkownika: a) strona tworzenia stacji ładowania, b) strona informacji o stacji ładowania.

a) b)



Rys. 2.5: Makiety interfejsów użytkownika: a) strona logowania, b) strona rejestracji.

Rozdział 3

Implementacja aplikacji

W niniejszym rozdziale znajduje się opis struktur danych znajdujących się w bazach danych. Opisane interfejsy programistyczne, modeli danych oraz zasady działania części serwerowej. Przedstawione widoki interfejsu użytkownika. Zostały opisane zasady współpracy serwera i aplikacji mobilnej.

3.1. Model bazy danych

W systemie wykorzystano dwie nierelacyjne bazy danych: MongoDB i Redis. MongoDB jest przeznaczona do przechowywania danych aplikacji. Redis wykorzystuje się tylko do czasowego przechowywania tokenów.

MongoDB

Baza danych składa się z dwóch kolekcji: `station` i `user`. Kolekcja `user` składa się z następujących pól:

- `_id : ObjectId(String)`
- `user_name : String`
- `email : String`
- `password : String`
- `Model : Object :`
  - `create_at : ISODate`
  - `update_at : ISODate`
  - `delete_at : ISODate`

Przykład encji:

```
{
    "_id":ObjectId("5fb93e4b721953b7c60983c6"),
    "user_name":"newtest",
    "email":"newtest@test.test",
    "password":"$2a$04$o0WfZvdk4WUpsct.BH3zw.3MFJFmUuLe8VjJx2OeyxtZuBliMOrl.",
    "model":{
        "create_at":ISODate("2020-11-21T16:20:27.044Z"),
        "update_at":ISODate("2020-11-21T16:20:27.044Z"),
        "delete_at":ISODate("0001-00-00T00:00:00Z")
    }
}
```

Kolekcja `station` składa się z następujących pól:

- `_id : String`
- `station_name : String`
- `owner_id : String`

- rating : Double
- latitude : Double
- longitude : Double
- description : String
- comments : Array :
  - _id : String
  - user_id : String
  - user_name : String
  - text : String
  - rating : Double
  - model : Object :
    - create_at : ISODate
    - update_at : ISODate
    - delete_at : ISODate
- model : Object :
  - create_at : ISODate
  - update_at : ISODate
  - delete_at : ISODate

Przykład encji:

```
{
    "_id":ObjectId("5fca6f81bb37f04ad438c1a5"),
    "station_name":"Station Name",
    "owner_id":"5fb828babe10c57ba70d49cd",
    "rating":3.6666666666666665,
    "description":"description",
    "latitude":57.12662933894774,
    "longitude":14.208925142884254,
    "model":{
        "create_at":ISODate("2020-12-04T17:18:57Z"),
        "update_at":ISODate("2020-12-04T17:20:55Z"),
        "delete_at":ISODate("0001-00-00T00:00:00Z")
    },
    "comments":[
        {
            "_id":"5fca6ff7bb37f04ad438c1a8",
            "user_id":"5fb828babe10c57ba70d49cd",
            "user_name":"test",
            "text":"  Comment 3",
            "rating":5,
            "model":{
                "create_at":ISODate("2020-12-04T17:20:55Z"),
                "update_at":ISODate("2020-12-04T17:20:55Z"),
                "delete_at":ISODate("0001-00-00T00:00:00Z")
            }
        },
        {
            "_id":"5fca6fe5bb37f04ad438c1a7",
            "user_id":"5fb828babe10c57ba70d49cd",
            "user_name":"test",
            "text":" Comment 2",
            "rating":3,
            "model":{
                "create_at":ISODate("2020-12-04T17:20:37Z"),
                "update_at":ISODate("2020-12-04T17:20:37Z"),
                "delete_at":ISODate("0001-00-00T00:00:00Z")
            }
        },
        {
            "_id":"5fca6fb8bb37f04ad438c1a6",
            "user_id":"5fb828babe10c57ba70d49cd",
            "user_name":"test",
            "text":"Comment 1",
            "rating":3,
            "model":{
```

```
                "create_at":ISODate("2020-12-04T17:19:52Z"),
                "update_at":ISODate("2020-12-04T17:19:52Z"),
                "delete_at":ISODate("0001-00-00T00:00:00Z")
            }
        }
    ]
}
```

Redis

Baza danych Redis wykorzystana tylko dla przechowywania tokenów użytkowników, które już wylogowane, ponieważ jedną z wad JWT tokenów jest to, że wygenerowany token nie można określić jako niedziałający, dopóki nie skończy się określony czas jego działania. Redis częściowo eliminuje ten problem.

W nim przechowuje się para „klucz - wartość", pewny czas. Po upływie tego czasu zapis automatycznie jest usuwany. Dla szybkiego wyszukiwania encja wygląda w następujący sposób: `token`, `token`. To pozwala często wylogować się użytkownikowi, ale zajmuje więcej miejsca niż encja typu: `user_id`, `token`.

3.2. Implementacja części serwerowej

3.2.1. Struktura RestApi

Narzędzia, technologie, biblioteki

Do stworzenia serwerowej części aplikacji użyto następujących technologii:

- Visual Studio Code - środowisko programistyczne;
- Go - język programowania;
- Go Modules - system zarządzania zależnościami;
- gorilla/mux - Router mapuje przychodzące żądania na listę zarejestrowanych tras i wywołuje moduł obsługi tego żądania, który odpowiada URL (ang. *Uniform Resource Locator*) adresowi;
- sirupsen/logrus - rejestrator strukturalny;
- mongo-driver - sterowanie MongoDB z języka Go;
- go-redis/redis - sterowanie Redis z języka Go;
- dgrijalva/jwt-go - realizacja JWT w języku Go;
- crypto/bcrypt - realizuje algorytm haszowania bcrypt;
- go-ozzo/ozzo-validation - pakiet wspomagający na walidację danych;
- yaml.v2 - implementuje obsługę YAML (ang. *Yet Another Markup Language*);
- google/uuid - sprawdza i generuje UUID (ang. *universally unique identifier*);

Struktura plików RestApi

Na rysunku 3.1 została przedstawiona struktura plików części serwerowej. Obok plików, niezbędnych do działania aplikacji, znajdują się pliki pozwalające na prowadzenie testów jednostkowych. Te pliki mają nazwę w postaci `*_test.go`.

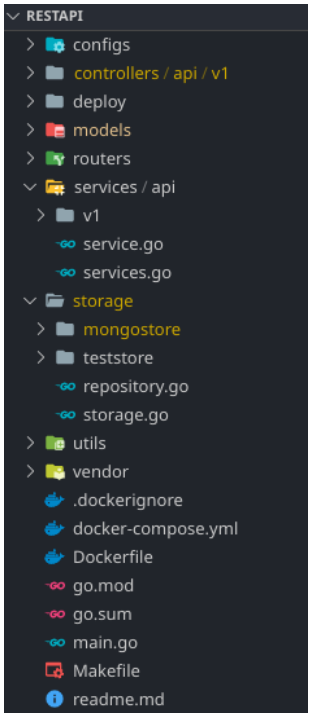
Katalog `config` zawiera pliki konfiguracyjne.

W katalogu `controllers` (rys. 3.2a) znajdują się kontrolery służące do kontrolowania zapytania, opracowania i zwrotu danych. Warstwa kontrolerów komunikuje z warstwą serwisów.

Do katalogu `deploy` jest kompilowana aplikacja przy uruchomieniu `Makefile`.

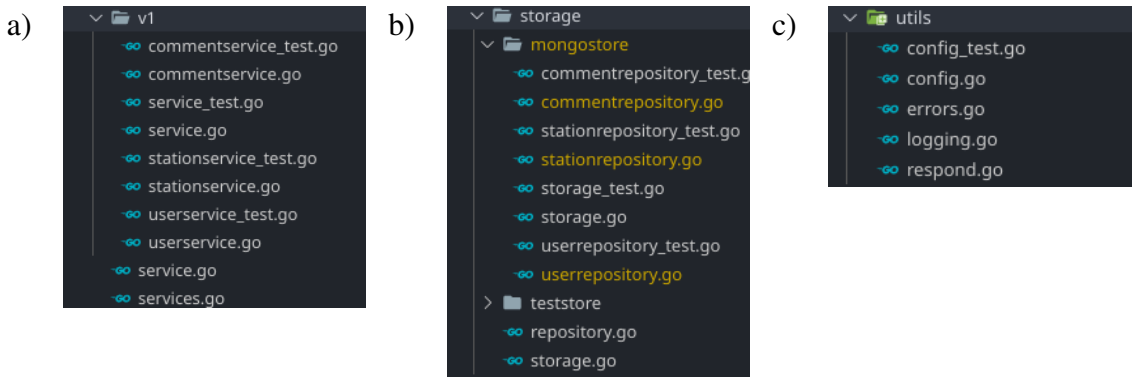
Katalog `models`(rys. 3.2b) zawiera modele danych.

Katalog `routers` (rys. 3.2c) zawiera plik, w którym są definiowane punkty końcowe (ang. *endpoints*) części serwerowej oraz zachodzi ich mapowanie na kontrolery.



Rys. 3.1: Struktura plików

Rys. 3.2: Struktura plików: a) kontrolery, b) modele danych, c) routery.



Rys. 3.3: Struktura plików: a) serwisy, b) współdziałanie z bazą danych, c) narzędzia.

Funkcje lub metody przechowywane w katalogu `services` (rys. 3.3a) prowadzą biznes logikę (obróbkę danych i podejmują decyzję co z nimi trzeba zrobić). Warstwa serwisów komunikuje z warstwą `DAL` (ang. *data access layer*). Część tych metod nie mają logiki biznesowej, natomiast one zaimplementowane w celu ułatwiania rozszerzenia funkcjalności, do podtrzymywania porządku w aplikacji oraz zgodności z Rest API architekturą.

Współpraca z bazą danych zachodzi w katalogu `storage` (rys. 3.3b). Jest to warstwa `DAL` która przeznaczona do komunikacji z bazą danych. Katalog `mongostore` współdziała z

MongoDB, natomiast `teststore` wykorzystuje się do testowania, które będzie omówione w rozdziale 4.

W katalogu `utils` (rys. 3.3c) znajdują się rzeczy wspomagające, na przykład lista błędów lub rejestracja działania serwera.

Konfiguracja serwera

Do zapobiegania ponownie kompilacji w przypadku zmiany portu, na którym działa serwer lub adresów baz danych, została utworzona struktura `Config` 3.1 (plik utils/config.go), która przy uruchomieniu aplikacji pobiera dane z pliku, który zostanie podany jako parametr wejściowy. Plik musi być typu `yaml`. Za pomocą biblioteki `yaml.v2` ten plik jest parsowany do obiektu struktury `Config` 3.2. Przykład pliku:

```
bind_addr: :8081
database_url: mongodb://127.0.0.1:27017
db_name: elCharge
db_user_collection: user
db_station_collection: station
db_redis: 127.0.0.1:6379
jwtKey: 21d5680b6a
```

Listing 3.1: Klasa konfiguracyjna części serwerowej.

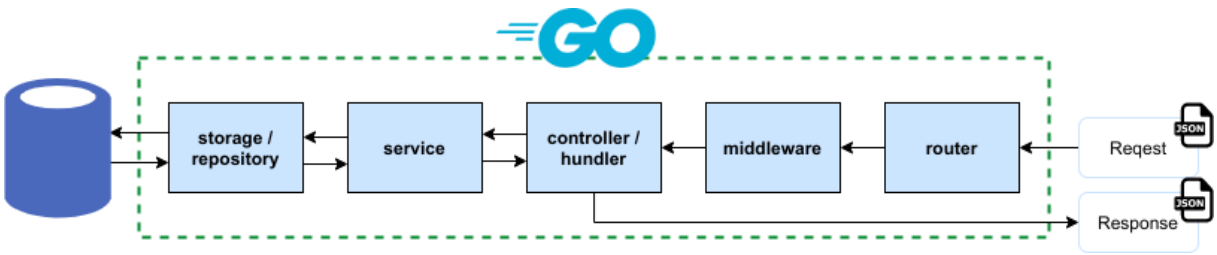
```
type Config struct {
    BindAddr            string `yaml:"bind_addr"`
    DatabaseURL         string `yaml:"database_url"`
    DbName              string `yaml:"db_name"`
    DbUserCollection    string `yaml:"db_user_collection"`
    DbStationCollection string `yaml:"db_station_collection"`
    RedisDB             string `yaml:"db_redis"`
    JWTKey              string `yaml:"jwtKey"`
}
```

Listing 3.2: Wczytanie pliku konfiguracyjnego części serwerowej.

```
func NewConfig(path string) (*Config, error) {
    configFile, err := ioutil.ReadFile(path)
    if err != nil {
        return nil, err
    }
    config := &Config{}
    err = yaml.Unmarshal(configFile, config)
    if err != nil {
        return nil, err
    }
    return config, nil
}
```

Przepływ danych

Na rysunku 3.4 został przedstawiony schemat przetwarzania i przepływu danych przy wysyłaniu zapytania do części serwerowej niniejszej aplikacji.



Rys. 3.4: Przepływ danych

Punkty końcowe

Część serwerowa jest napisana zgodnie z modelem Rest API. Wymiana danymi zachodzi za pomocą standardów: HTTP (ang. *HyperText Transfer Protocol*), URL, JSON. W tabeli 3.1 przedstawiony spis endpointów razem z metodą ich wysłania oraz krótkim opisem.

Tab. 3.1: Lista Enpointów części serwerowej

№	Metoda	Endpoint	Opis
1	GET	/api/v1	Endpoint do testowania działania serwera.
2	POST	/api/v1/login	Zalogowanie się użytkownika.
3	GET	/api/v1/logout	Wylogowanie się użytkownika.
4	POST	/api/v1/users	Tworzenie użytkownika / rejestracja
5	GET	/api/v1/users/id	Wczytywanie danych jednego użytkownika.
6	PUT	/api/v1/users/id	Edycja użytkownika.
7	DELETE	/api/v1/users/id	Usuwanie użytkownika.
8	GET	/api/v1/users/read?skip=&limit=	Wczytywanie danych limitowanej listy użytkowników użytkownika.
9	POST	/api/v1/stations	Tworzenie stacji ładowania.
10	GET	/api/v1/stations/id	Wczytywanie danych jednej stacji ładowania.
11	PUT	/api/v1/stations/id?ownid=	Edycja stacji.
12	DELETE	/api/v1/stations/id?ownid=	Usuwanie stacji ładowania.
13	GET	/api/v1/stations/read?skip=&limit= &lat=&lng=&dist=&descr=&name=	Wyszukiwanie stacji ładowania w zależności od parametrów.
14	POST	/api/v1/stations/{sid}/comments	Tworzenie komentarza.
15	GET	/api/v1/stations/{sid}/commentsid	Wczytywanie danych jednego komentarza.
16	PUT	/api/v1/stations/{sid}/commentsid	Edycja komentarza.
17	DELETE	/api/v1/stations/{sid}/commentsid	Usuwanie komentarza.
18	GET	/api/v1/stations/{sid}/read?skip= &limit=	Wczytywanie danych limitowanej listy komentarzy należących do pewnej stacji.

Kodem 3.3 została przedstawiona implementacja endpointów. Dla rejestracji połączeń wchodzących zostały użyte metody medialne: `s.logger.SetRequestID`, do przeznaczenia id każdemu połączeniu ,oraz `s.logger.LogRequest`, do wypisywania tego do konsoli. Dla większości przypadków też jest sprawdzano czy jest użytkownik zalogowany do systemu `s.authController.CheckToken`. To będzie wyjaśnione później (sekcja 3.2.2).

Listing 3.3: Implementacja punktów końcowych.

```
func (s *Server) SetupRouters() *mux.Router {
    v1 := "/api/v1"
    s.router.Schemes("http")
    s.router.Use(s.logger.SetRequestID) // middleware
    s.router.Use(s.logger.LogRequest)   // middleware
    s.router.HandleFunc(v1, s.testController.TestAPIV1()).Methods("GET")
    s.router.HandleFunc(v1+"/users", s.authController.CreateUser()).Methods("POST")
    s.router.HandleFunc(v1+"/login", s.authController.Login()).Methods("POST")
    s.router.HandleFunc(v1+"/logout/{id}", s.authController.Logout()).Methods("GET")

    user := s.router.PathPrefix(v1 + "/users").Subrouter()
    user.Use(s.authController.CheckToken)
    user.HandleFunc("/read", s.userController.Read()).Methods("GET")
    user.HandleFunc("/{id}", s.userController.FindByID()).Methods("GET")
    user.HandleFunc("/{id}", s.userController.DeleteByID()).Methods("DELETE")
```

```
    user.HandleFunc("/{id}", s.userController.UpdateByID()).Methods("PUT")

    stat := s.router.PathPrefix(v1 + "/stations").Subrouter()
    stat.Use(s.authController.CheckToken)
    stat.HandleFunc("", s.statController.CreateStation()).Methods("POST")
    stat.HandleFunc("/read", s.statController.Read()).Methods("GET")
    stat.HandleFunc("/{id}", s.statController.FindByID()).Methods("GET")
    stat.HandleFunc("/{id}", s.statController.DeleteByID()).Methods("DELETE")
    stat.HandleFunc("/{id}", s.statController.UpdateByID()).Methods("PUT")

    comm := stat.PathPrefix("/\{sid\}/comments").Subrouter()
    comm.Use(s.authController.CheckToken)
    comm.HandleFunc("/read", s.commController.Read()).Methods("GET")
    comm.HandleFunc("", s.commController.CreateComment()).Methods("POST")
    comm.HandleFunc("/{id}", s.commController.FindByID()).Methods("GET")
    comm.HandleFunc("/{id}", s.commController.DeleteByID()).Methods("DELETE")
    comm.HandleFunc("/{id}", s.commController.UpdateByID()).Methods("PUT")
    return s.router
}
```

3.2.2. Funkcje części serwerowej

W tej sekcji są opisane implementacje endpointów części serwerowej.

Użytkownik

Użytkownik jest niezbędny w pierwszej kolejności do uwierzytelniania. Niektóre dane użytkownika też są używane do rozumienia do kogo należy stacja ładownicza lub komentarz.

W listingu 3.4 jest przedstawiona struktura użytkownika, wraz ze sposobem konwersji do JSON i BSON, która znajduje się w pliku `models/user`.

Listing 3.4: Model danych użytkownika.

```
type User struct {
    ID       string `bson:"_id,omitempty" json:"_id,omitempty"`
    UserName string `bson:"user_name,omitempty" json:"user_name,omitempty"`
    Email    string `bson:"email,omitempty" json:"email,omitempty"`
    Password string `bson:"password,omitempty" json:"password,omitempty"`
    Model
}
```

Wczytywanie

Do wczytywania konkretnego użytkownika został zaimplementowany endpoint `/api/v1/users/{id}`. Metoda zapytania `GET`. W adresie URL musi być podany id użytkownika. Dla korzystania z danego endpointu użytkownik musi być zalogowany. Zwraca JSON obiekt znalezionego użytkownika.

Za pomocą routera jest realizowane przekierowanie z URL do wywołania metody kontrolera użytkownika `FindByID()` (listing 3.5), która zaczyna obróbkę biznesową zapytania. W tej metodzie jest wywoływana metoda serwisu `FindByID()` (listing 3.6). Później jest wysyłana odpowiedź.

Listing 3.5: Kontroler wczytywania użytkownika.

```
func (c *UserController) FindByID() http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        params := mux.Vars(r)
        id, ok := params["id"]
        if !ok {
            utils.Error(w, r, http.StatusBadRequest, utils.ErrWrongRequest)
            return
        }
        u, err := c.service.User().FindByID(id)
        if err != nil {
            utils.Error(w, r, http.StatusNoContent, err)
            return
```

```
            }
            utils.Respond(w, r, http.StatusFound, u)
        })
    }
```

Metoda `FindByID()` (listing 3.6) z pliku `services/v1/userservice.go`:

Listing 3.6: Serwis wczytywania użytkownika.

```
func (s *UserService) FindByID(id string) (*models.User, error) {
    u, err := s.storage.User().FindByID(id)
    if err != nil {
        return nil, err
    }
    u.Sanitize()
    return u, nil
}
```

Metoda `FindByID()` (listing 3.7) wyszukuje użytkownika z pewnym ID w bazie danych. Ona znajduję się w pliku `storage/mongostore/userrepository.go`.

Listing 3.7: Wczytywanie uzytkownika z bazy danych.

```
func (r *UserRepository) FindByID(id string) (*models.User, error) {
    idi, err := primitive.ObjectIDFromHex(id)
    if err != nil {
        return nil, err
    }
    filter := bson.M{"_id": idi}
    res := r.col.FindOne(context.TODO(), filter)
    u := &models.User{}
    err = res.Decode(u)
    if err != nil {
        return nil, utils.ErrRecordNotFound
    }
    return u, nil
}
```

Edycja

Do edycji użytkownika został zaimplementowany endpoint `/api/v1/users/{id}`. Metoda zapytania `PUT`. W adresie URL musi być podany id użytkownika. Jako ciało zapytania należy wysyłać obiekt w formacie JSON. Przykład:

```
{
"user_name": "test_user_2",
"update_at": "2020-11-01T13:27:31.105Z"
}
```

By móc korzystać z danego endpointu użytkownik musi być zalogowany.

Za pomocą routera jest realizowane przekierowanie z URL do wywołania metody warstwy kontrolerów użytkownika `UpdateByID()`, w której jest dekodowane ciało zapytania do obiektu `User`. Dalej dane przekazywane do metody warstwy serwisów `UpdateByID()`, która prowadzi obróbkę biznesową zapytania: haszowanie hasła metodą `bcrypt`, jeśli musi być zmienione. Zachowanie w zmiany danych w bazie danych zachodzi w metodzie `UpdateByID()` w pliku `storage/mongostore/userrepository.go` (warstwa bazy danych). Jako odpowiedź, przy udanej edycji zwraca się już edytowany obiekt (wyszukany w bazie danych za pomocą metody `FindByID()`) w formacie JSON.

Usunięcie

Do Usunięcia użytkownika został zaimplementowany endpoint `/api/v1/users/id`. Metoda zapytania `DELETE`. W adresie URL musi być podany id użytkownika. Dla korzystania z danego endpointu użytkownik musi być zalogowany.

Za pomocą routera jest realizowane przekierowanie z URL do wywołania metody kontrolera użytkownika `DeleteByID()`, która zaczyna obróbkę zapytania. W tej metodzie jest wywoływana metoda serwisu `DeleteByID()`. Ten serwis nie posiada logiki, oprócz

przekazania danych do warstwy bazy danych, ale może być przydatny w przyszłości. Metoda `DeleteByID()` (warstwa bazy danych) usuwa użytkownika z pewnym ID z bazy danych. Jej kod zawarty jest w `/storage/mongostore/userrepository.go`.

Autentykacja / logowanie i rejestracja

Dla autentykacji został wykorzystany JWT token, który jest generowany na serwerze, przy znalezieniu użytkownika o podanym adresie mailowym i haśle w bazie danych, i za tym jest wysłany w nagłówku odpowiedzi razem z danymi tego użytkownika. Ten token jest ważny jeden tydzień, za tym traci ważność i należy zalogować się ponownie. Wewnątrz tokena znajduje się nagłówek, sygnatura, czas ważności oraz id użytkownika. Większość endpointów dostępne tylko dla uwierzytelnionych użytkowników. W nagłówku zapytania musi znajdować się ważny token oraz ten token nie znajduje się w czarnej liście tokenów przechowywanych w bazie danych Redis.

Logowanie

Dla zalogowania należy wysłać metodą POST zapytanie na endpoint `api/v1/login` z ciałem zawierającym `email` i `password` w formacie JSON:

```
{
    "email": "test@test.test",
    "password": "password"
}
```

Mapowanie routera przekieruje to zapytanie do metody `Login()`, która znajduje się w `controllers/api/authcontroller.go` (listing 3.8). W tej metodzie, po znalezieniu użytkownika o takim adresie mailowym i haśle, jest generowany token JWT (listing 3.9) i wysyła się odpowiedź zawierająca nagłówek z tokenem oraz ciało z danymi użytkownika.

Listing 3.8: Kontroller logowania użytkownika.

```
func (c *AuthController) Login() http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        u := &models.User{}
        err := json.NewDecoder(r.Body).Decode(u)
        if err != nil {
            utils.Error(w, r, http.StatusBadRequest, err)
            return
        }
        u, err = c.service.User().Login(u)
        if err != nil {
            utils.Error(w, r, http.StatusUnauthorized, utils.ErrIncorrectEmailOrPassword)
            return
        }
        token, err := c.createTokenString(u.ID)
        if err != nil {
            utils.Error(w, r, http.StatusInternalServerError, err)
            return
        }
        w.Header().Set("Authorization", "Bearer "+token)
        utils.Respond(w, r, http.StatusOK, u)
    })
}
```

Listing 3.9: Generacja JWT tokena.

```
func (c *AuthController) createTokenString(uid string) (string, error) {
    expirationTime := time.Now().Add(168 * time.Hour)
    claims := &Claims{
        UID: uid, // user id
        StandardClaims: jwt.StandardClaims{
            ExpiresAt: expirationTime.Unix(), //lifetime
        },
    }
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    tokenString, err := token.SignedString([]byte(c.jwtKey))
```

```
    if err != nil {
        return "", err
    }
    return tokenString, nil
}
```

Dla wyszukiwania użytkownika o podanym adresie mailowym i haśle jest używana metoda `Login` (listing 3.10), znajdująca się w `serwices/api/v1/userservice.go`, która wyszukuje w bazie danych użytkownika o podanym adresie (adresy mailowe są unikalne) za pomocą metody `FindByEmal()` (listing 3.11) z pliku `storage/mongostore/userrepository.go` i porównuje hashowane hasło, pobrane z bazy danych, z nie haszowanym, z zapytania (listing 3.12).

Listing 3.10: Serwis logowania użytkownika.

```
func (s *UserService) Login(u *models.User) (*models.User, error) {
    if err := u.Validate(); err != nil {
        return nil, err
    }
    u2, err := s.storage.User().FindByEmail(u.Email)
    if err != nil {
        return nil, err
    }
    if !u2.VerifyPassword(u.Password) {
        return nil, utils.ErrIncorrectEmailOrPassword
    }
    u2.Sanitize()
    return u2, nil
}
```

Listing 3.11: Wysukiwanie użytkownika w bazie po adresie mailowym.

```
func (r *UserRepository) FindByEmail(email string) (*models.User, error) {
    filter := bson.M{"email": email}
    res := r.col.FindOne(context.TODO(), filter)
    u := &models.User{}
    err := res.Decode(u)
    if err != nil {
        return nil, utils.ErrRecordNotFound
    }
    return u, nil
}
```

Listing 3.12: Porównywanie hasła.

```
func (u *User) VerifyPassword(p string) bool {
    return bcrypt.CompareHashAndPassword([]byte(u.Password), []byte(p)) == nil
}
```

Rejestracja

Aby zarejestrować nowego użytkownika, należy wysłać zapytanie POST na adres `api/v1/users` z odpowiednim ciałem JSON zawierającym `email`, `user_name`, `password`:

```
{
    "email": "email@test3.com",
    "user_name": "test_user_1",
    "password": "password"
}
```

Dalej zostanie wykonana metoda `CreateUser()` z pliku `controllers/api/v1/authcontroller.go` (listing 3.13). W tej metodzie są dekodowane dane z ciała zapytania, które są w formacie JSON, i przekazane do metody `CreateUser()` (listing 3.15), która znajduje się w pliku `services/api/v1/userservice.go`. Po otrzymaniu danych z tego serwisu tworzy się token oraz wysyłana odpowiedź, która zawiera wygenerowany token i dane utworzonego użytkownika.

Listing 3.13: Kontroler tworzenia użytkownika.

```
func (c *UserController) CreateUser() http.HandlerFunc {
```

```
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        u := &models.User{}
        err := json.NewDecoder(r.Body).Decode(u)
        if err != nil {
            utils.Error(w, r, http.StatusBadRequest, err)
            return
        }
        u, err = c.service.User().CreateUser(u)
        if err != nil {
            utils.Error(w, r, http.StatusBadRequest, err)
            return
        }
        utils.Respond(w, r, http.StatusCreated, u)
    })
}
```

W metodzie `CreateUser()` warstwy serwisów (listing 3.15) zachodzi:

- walidacja danych (listing 3.14);
- sprawdzanie, czy istnieje użytkownik z takim adresem mailowym;
- jeśli nie istnieje, to zachodzi haszowanie hasła (listing 3.16), ustalenie czasu tworzenia oraz czasu ostatniej edycji obiektu. Jeśli już istnieje, to zwraca błąd;
- tworzenie nowego użytkownika w bazie danych (listing 3.17);
- przygotowanie obiektu użytkownika do transmisji: usuwanie hasła i daty tworzenia dla bezpieczeństwa oraz szybkości transmisji danych.

Ta metoda znajduje się w pliku `services/api/v1/userservice.go`.

Listing 3.14: Walidacja danych użytkownika.

```
func (u *User) Validate() error {
    return validation.ValidateStruct(
        u,
        validation.Field(&u.UserName, validation.By(isRequired(u.UserName != "")),
            ↪ validation.Length(2, 100)),
        validation.Field(&u.Email, validation.Required, is.Email),
        validation.Field(&u.Password, validation.By(isRequired(u.Password == "")),
            ↪ validation.Length(8, 50)),
    )
}
```

Listing 3.15: Serwis tworzenia użytkownika.

```
func (s *UserService) CreateUser(u *models.User) (*models.User, error) {
    if err := u.Validate(); err != nil {
        return u, err
    }
    _, err := s.storage.User().FindByEmail(u.Email)
    if err != utils.ErrRecordNotFound {
        if err != nil {
            return nil, err
        }
        return u, utils.ErrRecordAlreadyExists
    }
    if err := u.BeforeCreate(); err != nil {
        return u, err
    }
    id, err := s.storage.User().Create(u)
    if err != nil {
        return nil, err
    }
    u, err = s.storage.User().FindByID(id)
    if err != nil {
        return nil, err
    }
    u.Sanitize()
    return u, nil
}
```

Listing 3.16: Haszowanie hasła.

```
func EncryptString(str string) (string, error) {
    b, err := bcrypt.GenerateFromPassword([]byte(str), bcrypt.MinCost)
```

```
    if err != nil {
        return "", err
    }
    return string(b), nil
}
```

Metoda `Create()` (listing 3.17) tworzy zapis nowego użytkownika w bazie danych i zwraca jego ID. Ona znajduję się w `/storage/mongostore/userrepository.go`.

Listing 3.17: Zachowanie użytkownika do bazy danych.

```
func (r *UserRepository) Create(u *models.User) (string, error) {
    res, err := r.col.InsertOne(context.TODO(), u)
    if err != nil {
        return "", err
    }
    id := res.InsertedID.(primitive.ObjectID).Hex()
    return id, nil
}
```

Sprawdzenie tokenu

Większość, jak juz zostało powiedziano, endpointów pracują tylko z uwierzytelnionymi użytkownikami. Dla realizacji tej funkcji należy sprawdzić token, który zawiera się w nagłówku zapytania. Token musi nie tylko nie stracić ważności, ale i nie znajdować się w czarnej liście tokenów, która znajduje się Redis, bazie danych wykonującej rolę pamięci podręcznej. Metoda `CheckToken()` sprawdza te rzeczy (listing 3.18). Ta metoda jest metodą medialną, więc jest wywołana przed przekierowaniem poprzez router do warstwy kontrolerów. Dla komunikacji z bazą danych Redis została wykorzystana metoda `redis.Client.Get(sting)`, która pobiera dane.

Listing 3.18: Walidacja JWT tokena.

```
func (c *AuthController) CheckToken(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        tokenString := r.Header.Get("Authorization")
        if len(tokenString) == 0 {
            utils.Error(w, r, http.StatusUnauthorized, errors.New("Missing Authorization
                ↪ Header"))
            return
        }
        tokenString = strings.Replace(tokenString, "Bearer ", "", 1)
        claims := &Claims{}
        tkn, err := jwt.ParseWithClaims(tokenString, claims, func(token *jwt.Token) (
            ↪ interface{}, error) {
            return []byte(c.jwtKey), nil
        })
        if err != nil {
            if err == jwt.ErrSignatureInvalid {
                utils.Error(w, r, http.StatusUnauthorized, err)
                return
            }
            utils.Error(w, r, http.StatusBadRequest, err)
            return
        }
        if !tkn.Valid {
            utils.Error(w, r, http.StatusUnauthorized, err)
            return
        }
        _, err = c.rClient.Get(tokenString[37:]).Result() // find tocken in Redis
        if err != redis.Nil {
            utils.Error(w, r, http.StatusUnauthorized, errors.New("Invalid token"))
            return
        }
        next.ServeHTTP(w, r)
    })
}
```

Wylogowanie

Po wysłaniu zapytania GET na adres `api/v1/logout` zostanie wywoływana metoda

`Logout()` (listing 3.19) podczas działania której zostanie sprawdzony token (listing 3.18), za tym dodany do bazy danych Redis (pamięci podręcznej) oraz usunięty z nagłówka odpowiedzi.

Listing 3.19: Wylogowanie.

```
func (c *AuthController) Logout() http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        tokenString := r.Header.Get("Authorization")
        if len(tokenString) == 0 {
            utils.Error(w, r, http.StatusUnauthorized, errors.New("Missing Authorization
                ↪ Header"))
            return
        }
        tokenString = strings.Replace(tokenString, "Bearer ", "", 1)
        claims := &Claims{}
        tkn, err := jwt.ParseWithClaims(tokenString, claims, func(token *jwt.Token) (
            ↪ interface{}, error) {
            return []byte(c.jwtKey), nil
        })
        if err != nil {
            if err == jwt.ErrSignatureInvalid {
                utils.Error(w, r, http.StatusUnauthorized, err)
                return
            }
            utils.Error(w, r, http.StatusBadRequest, err)
            return
        }
        if !tkn.Valid {
            utils.Error(w, r, http.StatusUnauthorized, err)
            return
        }
        _, err = c.rClient.Get(tokenString[37:]).Result() // find in Redis db
        if err != redis.Nil {
            utils.Error(w, r, http.StatusUnauthorized, errors.New("Invalid token"))
            return
        }
        params := mux.Vars(r)
        uid, ok := params["id"]
        if !ok {
            utils.Error(w, r, http.StatusBadRequest, utils.ErrWrongRequest)
            return
        }
        if uid != claims.UID {
            utils.Error(w, r, http.StatusBadRequest, utils.ErrWrongRequest)
            return
        }
        c.rClient.Set(tokenString[37:], tokenString, 168*time.Hour) // save to Redis db
        utils.Respond(w, r, http.StatusOK, nil)
    })
}
```

Stacja

Głównym modelem w aplikacji jest stacja ładująca pojazdy elektryczne. Aplikacja została stworzona, aby uzyskiwać informacje o nich. Stacje można tworzyć, przeglądać, a także, jeśli użytkownik jest właścicielem, może też modyfikować.

W listingu 3.20 jest przedstawiona struktura encji stacji ładowania, wraz ze sposobem konwersji do JSON i BSON, która znajdująca się w pliku `models/user.go`.

Listing 3.20: Model danych stacji ładowania.

```
type Station struct {
    ID          string    `bson:"_id,omitempty" json:"_id,omitempty"`
    StationName string    `bson:"station_name,omitempty" json:"station_name,omitempty"`
    OwnerID     string    `bson:"owner_id,omitempty" json:"owner_id,omitempty"`
    Rating      float32   `bson:"rating,truncate" json:"rating,truncate"`
    Description string    `bson:"description,omitempty" json:"description,omitempty"`
    Comments    []Comment `bson:"comments,omitempty" json:"comments,omitempty"`
    Latitude    float64   `bson:"latitude" json:"latitude"`
    Longitude   float64   `bson:"longitude" json:"longitude"`
    Model
}
```

Dodawanie

Żeby dodać nową stację do systemu, należy wysłać zapytanie `POST` do części serwerówej na endpoint `api/v1/stations`. Zapytanie musi posiadać działający token JWT w nagłówku. Ciało zapytania musi posiadać następujące pola: `description`, `station_name`, `owner_id`, `longitude`, `latitude`. Przykład ciała zapytania:

```
{
            "description": "testText",
            "station_name": "station name",
            "owner_id": "5fbe4b1a187e72c56a5e4f70",
            "longitude":    15.162,
            "latitude": 12.46
}
```

Router przekieruje, po przechodzeniu przez funkcje medialne (w tym sprawdzenie JWT tokena), dane do kontrolera `CreateStation()` (plik `controllers/api/v1/stationcontoller.go`) który dekoduje wejściowy JSON w obiekt struktury `Station`. Ta metoda wywołuje serwis `CreateStation()`, znajdujący się w pliku `services/api/v1/stationservice.go`, w którym sprawdza się, czy istnieje stacja w tej pozycji, tworzy stację w bazie danych (metoda `Create()` z pliku `storage/mongostore/stationrepository.go`) oraz, przy udanym wpisie, pobiera się z bazy danych ten dokument i wysyła się jako ciało odpowiedź. Przed tworzeniem wpisu w bazie danych zachodzi walidacja danych (listing 3.21) oraz uzupełnienie niektórych niezbędnych pól (listing 3.22), na przykład czasu tworzenia i ostatniej modyfikacji.

Listing 3.21: Walidacja danych stacji ładowania.

```
func (s *Station) Validate() error {
    return validation.ValidateStruct(
        s,
        validation.Field(&s.StationName, validation.Required, validation.Length(2, 100)),
        validation.Field(&s.Description, validation.Required, validation.Length(5, 512)),
        validation.Field(&s.OwnerID, validation.Required, validation.Length(20, 30)),
        validation.Field(&s.Latitude, validation.Required, validation.Max(float64(90))),
        validation.Field(&s.Latitude, validation.Required, validation.Min(float64(-90))),
        validation.Field(&s.Longitude, validation.Required, validation.Max(float64(180))),
        validation.Field(&s.Longitude, validation.Required, validation.Min(float64(-180)))
            ↪ ,
    )
}
```

Listing 3.22: Uzupełnienie danych systemowych dotyczących.

```
func (s *Station) BeforeCreate() error {
    s.Model.BeforeCreate()
    s.Rating = 0
    return nil
}
```

Wczytywanie

Do otrzymania danych konkretnej stacji należy wysłać zapytanie `GET` na URL `api/v1/stations/{id}`. Na podstawie `id`, podanego w adresie, będzie znaleziony odpowiedni wpis w bazie danych lub, jeśli takiego nie istnieje, zwróci się pusty komunikat z nagłówkiem `204 Status No Content`.

Po przekierowaniu przez router z URL do metody kontrolera `FindByID()` (plik `controllers/api/v1/stationcontroller.go`), jest wycięty z adresu URL `id` stacji ładowania oraz wywołana metoda `FindByID()` (plik `services/api/v1/stationservice.go`). Ten serwis tylko wywołuje metodę (`FindByID`, która znajduje się w `/storage/mongostore/stationrepository.go`), która już bezpośrednio zwraca się do sterownika bazą danych MongoDB w języku Go.

Wyszukiwanie

W zależności od parametrów w adresie URL, przy wysłaniu zapytania metodą `GET` na adres `api/v1/stations/read`, będzie zrobione wyszukiwanie według różnych parametrów. Dostęp mają tylko zalogowani użytkownicy.

Lista parametrów w adresie URL (kolejność nie ma znaczenia):

- `skip` – pominięcie jakiejś liczby elementów (niezbędny parametr);
- `limit` – ograniczenie liczby zwracanych elementów (niezbędny parametr);
- `name` – wyszukiwanie według nazwy;
- `descr` (ang. *description*) – wyszukiwanie według opisu;
- `lat` (ang. *latitude*) – szerokość geograficzna. Wyszukiwanie według dokładnych współrzędnych geograficznych. Należy używać razem z `lng`;
- `lng` (ang. *longitude*) – długość geograficzna. Wyszukiwanie według dokładnych współrzędnych geograficznych. Należy używać razem z `lat`;
- `dist` (ang. *distance*) – promień wyszukiwania wokół współrzędnych geograficznych ustalonych za pomocą `lat`, `lng`. Wyszukiwanie zachodzi nie w promieniu od współrzędnych, lecz w kwadracie (dystans obliczany na wschód, na zachód, na północ oraz na południe).Jest opcjonalny przy użyciu `lat` oraz `lng`. Nie używa się osobno.

Przykład użycia: wyszukiwanie dwustu stacji ładowania w promieniu 100 kilometrów wokół pewnych współrzędnych geograficznych `http://127.0.0.1:8081/api/v1/stations/read?lat=57&lng=15&skip=0&limit=200&dist=100`.

Router przekieruje do metody `Read()` (listing 3.23) (plik `controllers/api/v1/stationcontroller.go`) dalej w zależności od wprowadzonych parametrów będzie wywołana ta czy inna metoda serwisu, które są podobne i najczęściej tylko przekazuja dane do następnej warstwy (warstwy bazy danych). Przy wprowadzaniu dwóch parametrów, na przykład `name` razem z `desc` będzie zrobione wyszukiwanie tylko według jednego parametru, w tym przypadku `name`. Jeśli wyszukiwanie zachodzi według nazwy lub opisu, zostaną wykorzystane wyrażenia regularne, co pozwala wyszukiwać, wiedząc tylko część nazwy lub opisu. Dla selekcji wyników na poziomie bazy danych często wykorzystuje się agregacja (metoda `Aggregare`), która po kolej prowadzi różne przekształcenia danych według sekwencji przenośnika (metoda `Pipeline`). Przykładami pracy z bazą danych są listingi 3.25 oraz 3.7.

Lista przypadków wywołań metod serwisu w zależności od parametru (wszystkie te metody znajdują się w pliku `services/api/v1/stationservice.go`):

- `name` – wykonuje się metoda `FindByName()` (listing 3.24), która wywołuje metodę współpracy z bazą danych `FinDByName()`;
- `descr` – wykonuje się metoda `FindByDescription()`, która wywołuje metodę współpracy z bazą danych `FindByDescription()`;
- `dist` razem z `lat` oraz `lng` – wykonuje się metoda `FindInRadius()`, która wywołuje metodę współpracy z bazą danych `FindInRadius()` (listing 3.25);
- `lat` oraz `lng` – wykonuje się metoda `FindByLocation()`, która wywołuje metodę współpracy z bazą danych `FindByLocation()`;
- `skip` razem z `limit` – bedzie pobrana lista kolejnych dokumentów z bazy danych, która będzie ograniczona przez te parametry (metoda `Read()`, która wywołuje metodę współpracy z bazą danych `Read()`.

Listing 3.23: Kontroler wyszukiwania stacji ładowania.

```
func (c *StationController) Read() http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        params := r.URL.Query()
        skipINT, err := strconv.Atoi(params.Get("skip"))
        if err != nil {
```

```
            utils.Error(w, r, http.StatusNoContent, err)
            return
        }
        limitINT, err := strconv.Atoi(params.Get("limit"))
        if err != nil {
            utils.Error(w, r, http.StatusNoContent, err)
            return
        }
        name := params.Get("name")
        if name != "" {
            stations, err := c.service.Station().FindByName(name)
            if err != nil {
                utils.Error(w, r, http.StatusNoContent, err)
                return
            }
            utils.Respond(w, r, http.StatusOK, stations)
            return
        }
        descr := params.Get("descr")
        if name != "" {
            stations, err := c.service.Station().FindByDescription(descr)
            if err != nil {
                utils.Error(w, r, http.StatusNoContent, err)
                return
            }
            utils.Respond(w, r, http.StatusOK, stations)
            return
        }
        latitude, err := strconv.ParseFloat(params.Get("lat"), 64)
        if err == nil {
            longitude, err := strconv.ParseFloat(params.Get("lng"), 64)
            if err == nil {
                // if we want to find station in radius around the coordinates
                distance, err := strconv.Atoi(params.Get("dist"))
                if err == nil && distance != 0 {
                    stations, err := c.service.Station().FindInRadius(latitude, longitude,
                        ↪ distance, skipINT, limitINT)
                    if err != nil {
                        utils.Error(w, r, http.StatusNoContent, err)
                        return
                    }
                    utils.Respond(w, r, http.StatusOK, stations)
                    return
                }
                // if we want get station in coordinates
                station, err := c.service.Station().FindByLocation(latitude, longitude)
                if err != nil {
                    utils.Error(w, r, http.StatusNoContent, err)
                    return
                }
                utils.Respond(w, r, http.StatusOK, station)
                return
            }
        }
        stations, err := c.service.Station().Read(skipINT, limitINT)
        if err != nil {
            utils.Error(w, r, http.StatusNoContent, err)
            return
        }
        utils.Respond(w, r, http.StatusOK, stations)
    })
}
```

Listing 3.24: Serwis wyszukiwania stacji ładowania według nazwy.

```
func (s *StationService) FindByName(name string) ([]models.Station, error) {
    st, err := s.storage.Station().FindByName(name)
    if err != nil {
        return nil, err
    }
    return st, nil
}
```

Listing 3.25: Wyszukiwanie stacji ładowania w bazie danych w pobliżu podanych współrzędnych na mapie Ziemi.

```
func (r *StationRepository) FindInRadius(latitude float64, longitude float64, radius
    ↪ float64, skip int, limit int) ([]models.Station, error) {
    matchInRadius := bson.D{{"$match", bson.M{
        "$and": []interface{}{
            bson.M{"latitude": bson.M{
                "$gte": latitude - radius}},
            bson.M{"latitude": bson.M{
                "$lte": latitude + radius}},
            bson.M{"longitude": bson.M{
                "$gte": longitude - radius}},
            bson.M{"longitude": bson.M{
                "$lte": longitude + radius}},
        }},
    }}
    cursor, err := r.col.Aggregate(
        context.TODO(),
        mongo.Pipeline{
            matchInRadius,
            bson.D{{"$skip", skip}},
            bson.D{{"$limit", limit}},
        })
    if err != nil {
        return nil, err
    }
    stations := []models.Station{}
    err = cursor.All(context.TODO(), &stations)
    if err != nil {
        return nil, err
    }
    return stations, nil
}
```

Dla wyszukiwania listy stacji ładowania w pobliżu podanych współrzędnych na mapie Ziemi należy wykorzystać wzór obliczający długość łuku ($L = \alpha R$, gdzie L - długość łuku w kilometrach, α - kąt, R - promień Ziemi (6378 km.)) do przekształcenia kilometrów w stopni kąta ($\alpha = L/R * 180/\pi$). To przekształcenie zachodzi w metodzie `getRadius()` (listing 3.26).

Listing 3.26: Obliczenie dystansu przeszukiwania.

```
func getRadius(rad int) float64 {
    if rad > 40074 {
        rad = 40074
    }
    radius := float64(float64(float64(rad)/float64(6378)) * float64(180) / math.Pi)
    return radius
}
```

Edycja

Do edycji stacji ładowania został zaimplementowany endpoint `/api/v1/stations/id?ownid=`. Metoda zapytania `PUT`. W adresie URL musi być podany id użytkownika oraz, jako parametr, id użytkownika, który probuje edytować. Edytować może tylko właściciel stacji oraz, dla zapobiegania jednoczesnej edycji z kilku urządzeń, musi zgadzać się pole `update_at` z wartością w bazie danych, które przechowuje datę ostatniej modyfikacji. Dostęp mają tylko zalogowani użytkownicy. Jako ciało zapytania należy wysyłać obiekt w formacie JSON. Do edycji dozwolone następujące pola: `description`, `station_name`. Przykład ciała zapytania:

```
{
            "description": "testText",
            "station_name": "station name"
}
```

Router przekieruje z adresu URL do metody kontrolera `UpdateByID()` z pliku `controllers/api/v1/stationcontroller.go`. W tej metodzie zostaje pobrany, z URL adresu, `id` stacji oraz `ownid` (id Użytkownika), dla

sprawdzenia właściciela. Dalej jest wywoływania metoda serwisu UpdateByID() (plik services/api/v1/stationservice.go), w której wywołują się metody współpracy z bazą danych UpdateByID() oraz FidByID() (plik storage/mongostore/stationrepository.go) do zmiany dokumentu we wpisie oraz pobraniu wynikowego dokumentu, odpowiednio.

Usunięcie

W celu usunięcia stacji ładowania z systemu został stworzony endpoint api/v1/stations/{id}?ownid="", metoda do wysyłania DELETE, id to jest id stacji która podlega usunięciu. Jako parametr ownid należy podać id użytkownika, który stworzył tą stację.

Router przekieruje dane do metody DeleteByID() kontrolera z pliku controllers/api/v1/stationcontroller.go, gdzie są wczytywane dane z adresu URL oraz dekodowane ciało zapytania. Dalej dane przekierowane do metody serwisu (DeleteByID() znajdującej się w pliku services/api/v1/stationservice.go). W serwisie jest wywołana metoda współpracy z bazą danych (DeleteByID) z pliku stoarage/mongostore/tationrepository.go, gdzie zachodzi sterowanie bazą danych do usunięcia wpisu.

Komentarz

Stacja ładownicza przechowuje listę komentarzy. Komentarz zawiera tekst, id, imę użytkownika, który go stworzył, jego id, oraz ocenę, na podstawie której jest oceniana cała stacja. Strukturę komentarza w języku Go, sposób przekazania do obiektu JSON oraz BSON pokazano na listingu (listing 3.27).

Listing 3.27: Model danych komentarza.

```
type Comment struct {
    ID       string  `bson:"_id,omitempty" json:"_id,omitempty"`
    UserID   string  `bson:"user_id,omitempty" json:"user_id,omitempty"`
    UserName string  `bson:"user_name,omitempty" json:"user_name,omitempty"`
    Text     string  `bson:"text,omitempty" json:"text,omitempty"`
    Rating   float32 `bson:"rating,omitempty,truncate" json:"rating,omitempty,truncate"`
    Model
}
```

Dodawanie

W celu umożliwiana tworzenia komentarza został zaimplementowany endpotint /api/v1/stations/{sid}/comments. Zapytanie wysyła się metodą POST, ciało zawiera następujące pola: user_id, rating, text, user_name. Dla tworzenia komentarza do stacji ładowania zapytanie musi posiadać działający token JWT oraz w adresie URL musi być podany id (sid -to id stacji) tej stacji. Przykład ciała zapytania:

```
{
            "user_id":   "5fb828babe10c57ba70d49cd",
            "rating":  3,
            "text":     "some text",
            "user_name": "test username"
}
```

Router przekieruje zapytanie, po przechodzeniu autentykacji, do metody kontrolera CreateComment() (plik controllers/api/v1/commentcontroller.go), w której zachodzi otrzymanie niezbędnych danych z adresu URL oraz zaczyna się część biznesowa w metodzie CreateComment() (listing 3.28) (plik services/api/v1/commentservice.go). W tym serwisie zachodzi walidacja, dodawanie niektórych pól (na przykład czas tworzenia i ostatniej modyfikacji), dodanie nowego komentarza na początek listy komentarzy stacji ładowania (jest to edycja dokumentu stacji ładowania w

bazie danych o podanym numerze `id`) (metoda `Create`) oraz obliczenie oceny wynikowej stacji ładowania z uwzględnieniem nowego komentarza za pomocą metody `UpdateRaitingByID` (listing 3.29) z pliku `storage/mongostore/stationrepository.go`. Do aktualizacji oceny została wykorzystana agregacja postępów: za pomocą parametru `$reduce` została obliczona suma ocen wszystkich komentarzy, podzielona (`$divide`) przez liczbę komentarzy (`$cond`), i zachowana w pole `rating` za pomocą parametru `$set` (listing 3.29). Aktualizacja zachodzi w dodatkowej goroutinie (gorutiny są jak lekkie wątki, o rozmiarze 2Kb) żeby zmniejszyć czas oczekiwania użytkownika.

Listing 3.28: Serwis tworzenia komentarza.

```
func (s *CommentService) CreateComment(sid string, c *models.Comment) (*models.Comment,
    ↪ error) {
    if err := c.Validate(); err != nil {
        return c, err
    }
    if err := c.BeforeCreate(); err != nil {
        return c, err
    }
    id, err := s.storage.Comment().Create(sid, c)
    if err != nil {
        return nil, err
    }
    go func() {
        err := s.storage.Station().UpdateRaitingByID(sid)
        if err != nil {
            log.Println(err.Error())
        }
    }()
    c, err = s.storage.Comment().FindByID(sid, id)
    if err != nil {
        return nil, err
    }
    return c, nil
}
```

Listing 3.29: Aktualizacja oceny stacji.

```
func (r *StationRepository) UpdateRaitingByID(id string) error {
    idi, err := primitive.ObjectIDFromHex(id)
    filter := bson.M{"_id": idi}
    avg := bson.D{{
        "$set", bson.M{
            "rating": bson.M{
                "$divide": []interface{}{
                    bson.M{
                        "$reduce": bson.M{
                            "input":        "$comments",
                            "initialValue": 0,
                            "in":           bson.M{"$add": []interface{}{"$$value", "
                                ↪ $$this.rating"}},
                        },
                    },
                    bson.M{
                        "$cond": []interface{}{
                            bson.M{"$ne": []interface{}{bson.M{"$size": "$comments"}, 0}},
                            bson.M{"$size": "$comments"},
                            1,
                        },
                    },
                },
            },
        },
    }}
    _, err = r.col.UpdateOne(
        context.TODO(),
        filter,
        mongo.Pipeline{avg})
    return err
}
```

Wczytywanie

Do wczytywania danych konkretnego komentarza został zaimplementowany endpoint `api/v1/stations/{sid}/comments/{id}`. Jako `sid` używa się `id` stacji, do której należy komentarz, jako `id` używa się `id` komentarza. Zapytanie mysi być wysłano za pomocą metody `GET`. Użytkownik musi być uwierzytelniony.

Router przekazuje zapytanie dane do warstwy kontrolerów, do metody `DeleteByID()`. Kontroler pobiera dane z URL adresu i wywołuje metodę warstwy serwisów (metoda `DeleteByID()`), która wywołuje metodę do współpracy z bazą danych `DeleteByID()` (z pliku `storage/mongostore/commentrepository.go`). W bazie danych zachodzi znalezienie dokumentu stacji ładowania o podanym `id` w cztery kroki: znalezienie stacji z podanym `id` (`$match`), dekonstrukcja dokumentu stacji tak, aby wyświetlić dokument ko każdego elementu listy komentarzy (`$uwnind`), wymiana wejściowego dokumentu zadanym (`$replaceRoot`), znalezienie komentarza z pewnym `id` (`$match`).

Edycja

W celu edycji komentarza został zaimplementowany endpoint `api/v1/stations/\{sid\}/comments/{id}`. Jako `sid` używa się `id` stacji, do której należy komentarz, jako `id` używa się `id` komentarza. Zapytanie mysi być wysłano za pomocą metody `PUT`. Użytkownik musi być uwierzytelniony.

Router przekieruje adres URL do metody `UpdateByID()` warstwy kontrolerów. W tej metodzie zachodzi otrzymanie danych (parametrów oraz zmiennych) z adresu URL oraz dekodowanie ciała z formatu JSON do obiektu `Comment`. Dalej te dane przekazują się do metody `UpdateByID()` warstwy serwisów, gzie zachodzi edycja listy komentarzy należących do stacji ładowania, oraz jednocześnie z wyszukiwaniem i wysyłaniem klientowi edytowanego komentarza, zachodzi aktualizacja oceny stacji ładowania (listing 3.29).

Usunięcie

`api/v1/stations/{sid}/comments/{id}`. Jako `sid` używa się `id` stacji, do której należy komentarz, jako `id` używa się `id` komentarza. Zapytanie mysi być wysłano za pomocą metody `DELETE`. Użytkownik musi być uwierzytelniony.

Router przekazuje zapytanie dane do warstwy kontrolerów, do metody `DeleteByID()`. Kontroler pobiera dane z URL adresu i wywołuje metodę warstwy serwisów (`DeleteByID()`, która wywoła metodę do współpracy z bazą danych `DeleteByID()` (z pliku `storage/mongostore/commentrepository.go`). W bazie danych zachodzi modyfikacja dokumentu stacji ładowania o podanym `id`: usunięcie komentarza z listy. Następnym krokiem jednocześnie zachodzi wysłanie potwierdzenie usunięcia oraz aktualizacja oceny (listing 3.29).

3.3. Implementacja Interfejsu użytkownika

Interfejsem użytkownika jest aplikacją mobilną dla telefonów z systemem operacyjnym Android.

3.3.1. Struktura AndroidUI

Narzędzia, technologie, biblioteki

Do stworzenia aplikacji mobilnej użyto następujących technologii:

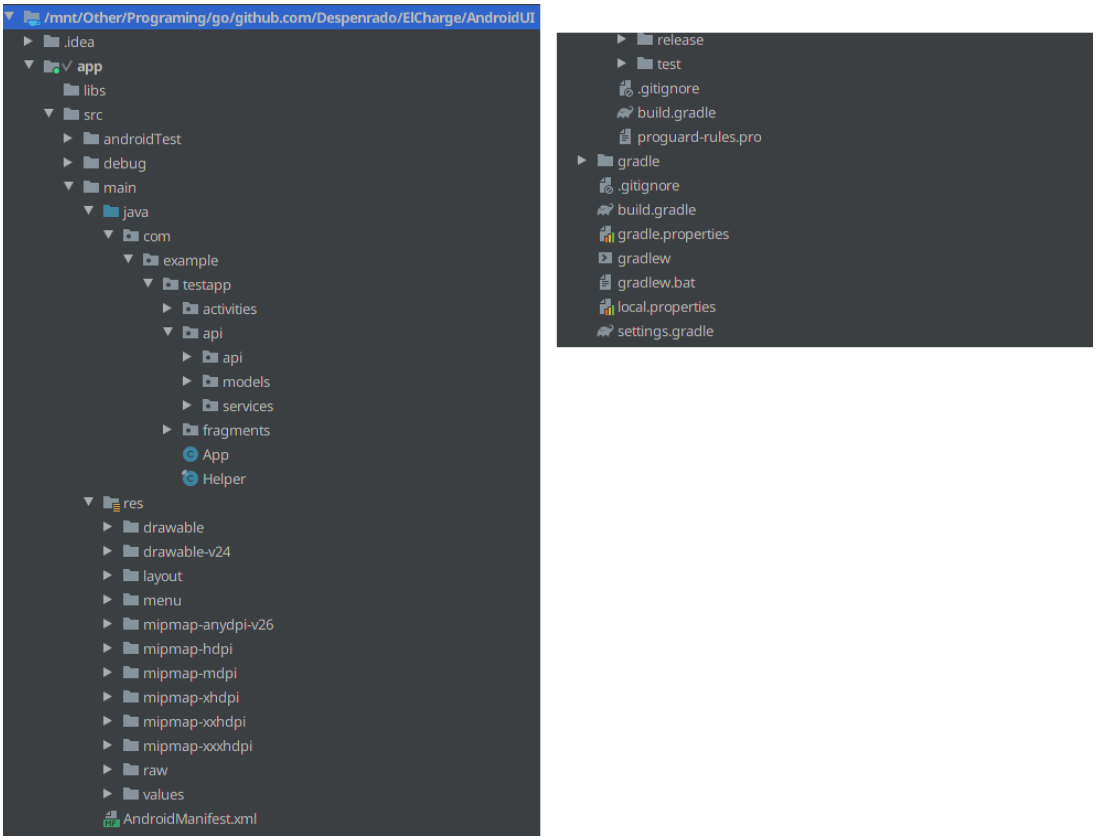
- Android Studio - srodowisko programistyczne;
- Java SDK 8 - narzędzia programistyczne języka Java wersji 8;

- Grable - system zarządzania zależnościami oraz budowaniem projektu;
- OkHttp3 - HTTP klient;
- Retrofit - służące do komunikacji przez interfejsy API opakowanie dla OkHttp3;
- RxJava2 - biblioteka umożliwiająca programowanie reaktywne w języku Java;
- gms:play-services-maps - narzędzia programistyczne do pracy z mapą Google. Jest częścią Google Maps Android API;
- gms:play-services-location - narzędzia programistyczne do pracy z położeniem urządzenia użytkownika. Jest częścią Google Maps Android API;
- android-maps-utils - narzędzia programistyczne do pracy z elementami dodatkowymi mapy (na przykład markery). Jest częścią Google Maps Android API;
- gson - służy do konwersji obiektów do Formaty JSON;
- yaml.v2 - implementuje obsługę YAML (ang. *Yet Another Markup Language*);

Do pracy ze Static Maps (dostęp do API Google Maps) od firmy Google, należy otrzymać i ustalić w aplikacji token identyfikujący aplikację. Przez to można przeglądać statystykę zapytań do serwisów Google za pomocą `console google cloud`.

Struktura plików AndroidUI

Na rysunku 3.5 została przedstawiona struktura plików aplikacji mobilnej. Ta struktura ma dość głębokie drzewo, więc dalej będzie wykorzystany jako katalog korzeniowy `app/src/main`.



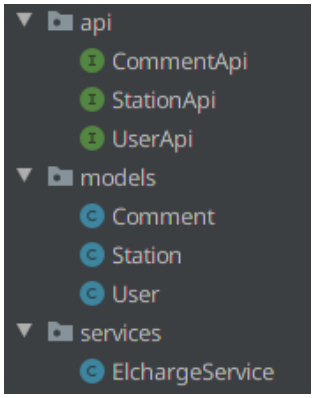
Rys. 3.5: Struktura plików aplikacji mobilnej

Zawartość znaczących katalogów i plików:

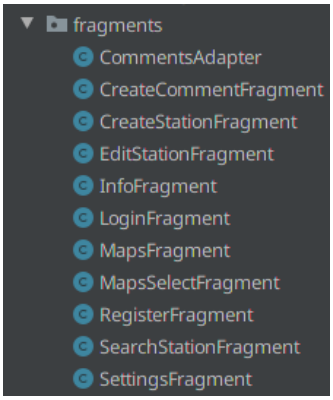
- katalog `main/java/com/example/testapp/activities` zawiera klasy `activity`, są ekranami aplikacji, na których umieszczone wszystkie komponenty interfejsu użytkownika.

- katalog `main/java/com/example/testapp/api` zawiera modeli danych (`katalog models`), sposób komunikacji (katalog `services`) oraz endpointy (katalog `api` na rysunku 3.6a) do komunikacji z częścią serwerową .
- katalog `main/java/com/example/testapp/fragments` zawiera klasy fragmentów (rys. 3.6b), które umieszczają się w kontenerze na `activity` i zawierają elementy interfejsu użytkownika (przyciski, pola tekstowe i inne).
- w katalogu `res` znajdują się zasoby aplikacji, na przykład: plik konfiguracyjny (`res/raw/config.properties`), opisy komponentów oraz ich dyslokacja na fragmentach i oknach aplikacji (`res/layout/` na rysunku 3.6c) i inne.
- plik `main/androidManifest.xml` (listing 3.30) zawiera informację niezbędną do budowy aplikacji: definiowanie `activity`, uprawnień, klasy podstawowej.
- plik `main/java/com/example/testapp/App.java` jest klasą podstawową, który jest niezbędny do podtrzymywania globalnego stanu aplikacji (`ApplicatioContext`).

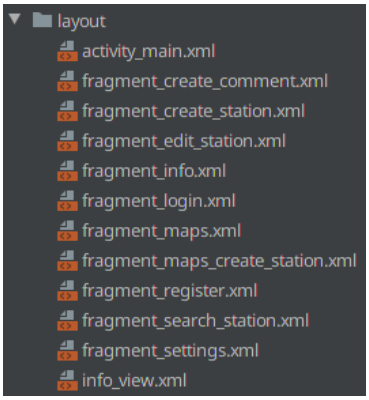
a)



b)



c)



Rys. 3.6: Struktura plików: a) API do połączenia z serwerem, b) klasy fragmentów, c) opis fragmentów.

Listing 3.30: androidManifest.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.testapp">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

    <application
        android:name=".App"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme"
        android:usesCleartextTraffic="true">
        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="@string/google_maps_key" />

        <activity android:name=".activities.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
```

```
</manifest>
```

Modele danych

W prazy zrealizowane modeli danych: `User` (listing 3.31), `Station`, `Comment`. W listingu komentarz `// GETTERS and SETTERS` pokazuje, że w kodzie programu tym miejscu znajdują się proste metody do zapisu i wczytania pól klasy. Adnotacja `@SerializedName` ustala nazwę do dekodowania do formatu JSON i na odwrót. Klasy `Station` oraz `Comment` zrealizowane podobno do klasy `User`.

Listing 3.31: Plik `main/java/com/example/testapp/api/api/UserApi.java`.

```
public class User {
    @SerializedName("_id")
    private String id;
    @SerializedName("user_name")
    private String userName;
    @SerializedName("email")
    private String email;
    @SerializedName("password")
    private String password;
    @SerializedName("update_at")
    private String updateAt;

    public User() {
    }

    public User(String id, String userName, String email, String password, String updateAt
        ↪ ) {
        this.id = id;
        this.userName = userName;
        this.email = email;
        this.password = password;
        this.updateAt = updateAt;
    }
    // GETTERS and SETTERS
}
```

API: Zaimplementowane API podzielono na trzy części: `UserApi`, `StationApi`, `CommentApi`. Metody API służą do komunikacji z częścią serwerową.

Zastosowany framework pozwala na zadeklarowanie metod protokołu HTTP `POST`, `PUT`, `GET`, `DELETE` obsługiwanych w danej metodzie interfejsu na danym adresie URL poprzez zastosowanie adnotacji. Adnotacja `Body` pozwala na zadeklarowanie zamieszczenia danych w ciele zapytania. Adnotacja `Path` mówi, że dane ukryte są w adresie URL. Adnotacja `Query` mówi, że dane przekazane są jako parametry w adresie URL. Na listingu 3.32 pokazano przykład implementacji `UserAPI`.

Listing 3.32: Plik `main/java/com/example/testapp/api/api/UserApi.java`.

```
public interface UserApi {
    @POST("users")
    public Single<Response<User>> createUser(@Body User body);

    @POST("login")
    public Single<Response<User>> login(@Body User body);

    @GET("logout/{id}")
    public Single<ResponseBody> logout(@Path("id") String id);
}
```

W tabeli 3.2 przedstawione API do połączenia z częścią serwerową (niektóre endpointy zaimplementowane z perspektywą na przyszłość)

Tab. 3.2: API do połączenia z częścią serwerową

№	Metoda	url	Opis
1	POST	/api/v1/login	Zalogowanie się użytkownika.
2	GET	/api/v1/logout	Wylogowanie się użytkownika.
3	POST	/api/v1/users	Tworzenie użytkownika / rejestracja
9	POST	/api/v1/stations	Tworzenie stacji ładowania.
10	GET	/api/v1/stations/id	Wczytywanie danych jednej stacji ładowania.
11	PUT	/api/v1/stations/id?ownid=	Edycja stacji.
12	DELETE	/api/v1/stations/id?ownid=	Usuwanie stacji ładowania.
13	GET	/api/v1/stations/read?skip=&limit= &lat=&lng=&dist=	Wyszukiwanie stacji ładowania w promieniu od pozycji na mapie (wszpórzędnych).
13	GET	/api/v1/stations/read?skip=&limit= &lat=&lng=	Wyszukiwanie stacji ładowania według pozycji na mapie (wszpórzędnych).
13	GET	/api/v1/stations/read?skip=&limit= &descr=	Wyszukiwanie stacji ładowania według opisu.
13	GET	/api/v1/stations/read?skip=&limit= &name=	Wyszukiwanie stacji ładowania według nazwy.
13	GET	/api/v1/stations/read?skip=&limit=	Pobranie pewnej liczby stacji zaczynając od pewnej pozycji.
14	POST	/api/v1/stations/{sid}/comments	Tworzenie komentarza.
15	GET	/api/v1/stations/{sid}/commentsid	Wczytywanie danych jednego komentarza.
16	PUT	/api/v1/stations/{sid}/commentsid	Edycja komentarza.
17	DELETE	/api/v1/stations/{sid}/commentsid	Usuwanie komentarza.
18	GET	/api/v1/stations/{sid}/read?skip= &limit=	Wczytywanie danych limitowanej listy komentarzy należących do pewnej stacji.

Komunikacja z częścią serwerową

Sposób komunikacji z częścią serwerową oraz etapy pośrednicze (zostały zdefiniowane w metodzie `createOkHttpClient()`), w których zachodzi dodawanie nagłówka oraz pisanie logów zapytań i odpowiedzi, został opisany w klasie `ElchargeService` (listing 3.33). Został użyty HTTP klient `OkHttp3` oraz jego opakowanie `Retrofit2`.

Listing 3.33: obsługa komunikacji z częścią serwerową.

```
public class ElchargeService {
String apiAddr;
String token;
UserApi userApi;
StationApi stationApi;
CommentApi commentApi;
User user;

public ElchargeService(){
    apiAddr = Helper.getConfigValue(App.getAppContext(),"apiserver_addr");
    token = Helper.getConfigValue(App.getAppContext(),"apiserver_token");
    Retrofit retrofit = createRetrofit();
    userApi = retrofit.create(UserApi.class);
    stationApi = retrofit.create(StationApi.class);
    commentApi = retrofit.create(CommentApi.class);
}

private OkHttpClient createOkHttpClient() {
```

```
        OkHttpClient.Builder httpClient = new OkHttpClient.Builder();
        httpClient.addInterceptor(new Interceptor() {
            @Override
            public Response intercept( Chain chain) throws IOException {
                Request request = chain.request().newBuilder()
                        .addHeader("Authorization", token)
                        .build();
                return chain.proceed(request);
            }
        });
        HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
        logging.setLevel(HttpLoggingInterceptor.Level.BODY);
        httpClient.addInterceptor(logging);
        return httpClient.build();
    }

    private Retrofit createRetrofit() {
        return new Retrofit.Builder()
                .baseUrl(apiAddr)
                .client(createOkHttpClient())
                .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
                .addConverterFactory(GsonConverterFactory.create())
                .build();
    }

    public void setToken(String token) {
        Helper.setConfigValue(App.getAppContext(),"apiserver_token", token);
        this.token = token;
    }

    // GETTERS and SETTERS
}
```

Komunikacja z serwerem w celu zwiększenia pozytywnego doświadczenia pracy użytkownika z aplikacją zrobiona w sposób asynchroniczny za pomocą biblioteki `RxJava2`. Do serwera wysyła się zapytanie na adres URL za pomocą odpowiedniego interfejsu API, indujących się w katalogu `main/java/com/example/testapp/api/api` (kod klasy `User`: 3.32. Dałej zachodzi oczekiwanie na odpowiedź (`subscribeOn(Shedulers.io())`), po otrzymaniu odpowiedzi w głównym wątku aplikacji (`observeON(AndroidShedulers.mainThread)`) wykonuje się kod opisany poprzez realizację klasy abstrakcyjnej `DisposableSingleObserver<T>`. Podczas realizacji tej klasy należy przedefiniować metodę `onSucces(Object)` (opis działań przy otrzymaniu odpowiedzi od serwera) oraz `onError(Throwable)` (opis działań przy braku odpowiedzi od serwera). Przykład kody na podstawie logowania znajduje się w listingu 3.37. Przy zamknięciu fragmentu lub aplikacji przez użytkownika, kanał oczekujący na odpowiedź z serwera będzie również zamknięty.

Układ interfejsu graficznego

Ekran aplikacji składa się z kilku warstw:

- `Activity` to samodzielny ekran, na którym umieszczają się inne elementy (warstwy, przyciski, teksty i inne).
- `Fragment` to ekran, na którym umieszczają się inne elementy (warstwy, przyciski, teksty i inne). Nie jest samodzielnym ekranem. Zwykle umieszcza się w kontenerze.

Dla umieszczania oraz definiowania elementów na ekranie wykorzystane opisy w postaci plików `xml` (ang. *eXtensible Markup Language*), które przechowywane w katalogu `main/res`. Na listingu 3.34 przedstawiony plik `main/res/layout/fragment_create_comment.xml`. Jest to opis fragmentu tworzenia nowego komentarza (rys. 3.11). Aplikacja korzysta z jednego ekranu `Activity` na którym na dole znajduje się menu. Natomiast pozostałą, większą, przestrzeń ekranu zajmuje kontener do fragmentu, na fragmentach umieszczają się różne elementy aplikacji (przyciski, teksty i inne). Takie podejście pozwala nakładać fragmenty jeden

na jednego, dla uniknięcia usunięcia już wprowadzonej informacji przez użytkownika, oraz usuwać już niepotrzebne. Opisy zachowania elementów ekranu znajdują się w odpowiednich klasach opisujących zachowanie całego ekranu i elementów na nim. Te klasy napisane w języku Java (pliki w katalogu `main/java/com/example/testapp/fragments`) (przykład w listingu 3.35).

Listing 3.34: Opis fragmentu tworzenia komentarza.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.
    ↪ com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/cardview_light_background"
    tools:context=".fragments.CreateCommentFragment">

    <EditText
        android:id="@+id/editTextText"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="16dp"
        android:layout_marginBottom="505dp"
        android:ems="10"
        android:hint="Text"
        android:maxLength="512"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/ratingBar" />

    <RatingBar
        android:id="@+id/ratingBar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="72dp"
        android:layout_marginTop="26dp"
        android:layout_marginEnd="73dp"
        android:layout_marginBottom="22dp"
        android:backgroundTint="#2ED573"
        android:numStars="5"
        android:stepSize=".5"
        app:layout_constraintBottom_toTopOf="@+id/editTextText"
        app:layout_constraintEnd_toEndOf="@+id/editTextText"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/buttonSave"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="136dp"
        android:layout_marginEnd="76dp"
        android:text="Save"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editTextText" />

    <Button
        android:id="@+id/buttonCancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="76dp"
        android:layout_marginTop="136dp"
        android:text="cancel"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editTextText" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 3.35: Klasa opisująca zachowanie elementów na fragemencie tworzenia komentarza.

```
public class CreateCommentFragment extends Fragment {
```

```
    private CompositeDisposable disposable = new CompositeDisposable();
    private App app;
    private String stationID;

    public CreateCommentFragment(String stationID) {
        this.stationID = stationID;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_create_comment, container, false);
    }

    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        this.app = (App) getActivity().getApplication();
        Button btnSave = (Button)getView().findViewById(R.id.buttonSave);
        btnSave.setOnClickListener(this::onButtonSave);
        Button btnCancel = (Button)getView().findViewById(R.id.buttonCancel);
        btnCancel.setOnClickListener(this::onButtonCancel);
    }

    private void onButtonSave(View v){
        Comment comment = new Comment();
        comment.setRating(((RatingBar)getView().findViewById(R.id.ratingBar)).getRating())
            ↪ ;
        comment.setText(((EditText)getView().findViewById(R.id.editTextText)).getText().
            ↪ toString());
        comment.setUserID(app.getElchargeService().getUser().getId());
        comment.setUserName(app.getElchargeService().getUser().getUserName());
        createComment(comment);
    }

    private void onButtonCancel(View v){
        getFragmentManager().beginTransaction().remove(this).commit();
    }
}
```

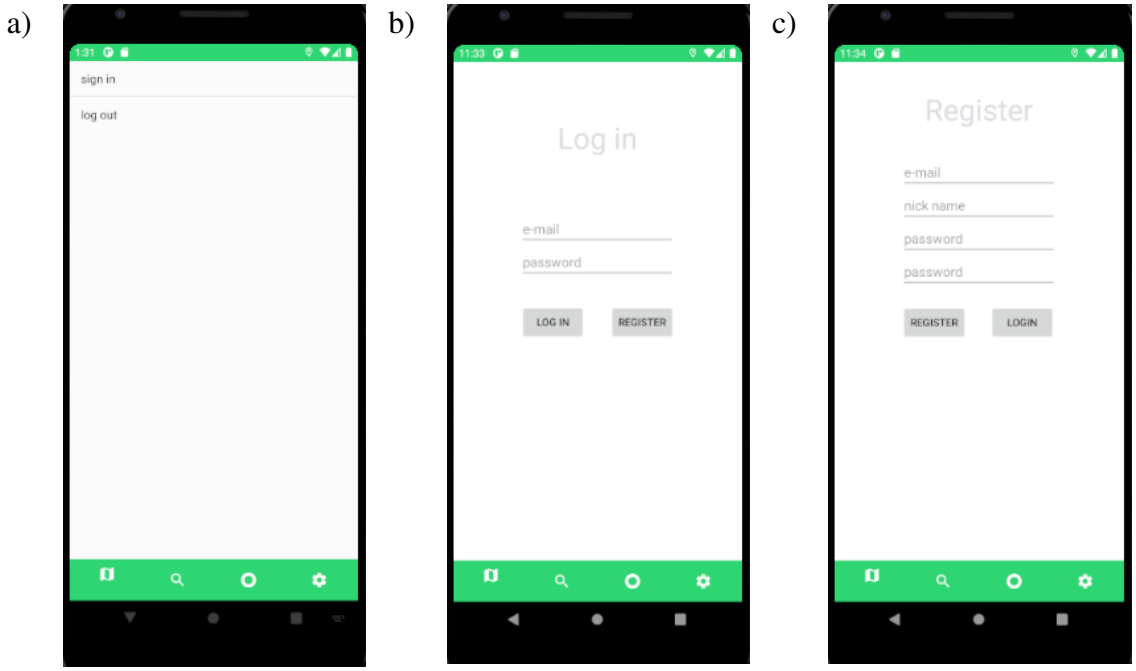
3.3.2. Funkcje aplikacji mobilnej

W tej sekcji opisano współdziałanie użytkownika i systemu oraz implementacja tych części.

Logowanie i rejestracja

Dla otwarcia strony logowania lub rejestracji można nacisnąć odpowiedni przycisk, który znajduje się w prawym dolnym rogu (koło zębate). Otworzą się ustalenia jak na rysunku 3.7a. Na tym fragmencie można zalogować się (przycisk `Log in`) lub wylogować się (przycisk `Log out`). Przy próbie wykorzystania elementów aplikacji, które potrzebują autentykacji, ale użytkownik jeszcze nie będzie zalogowany do systemu, logowanie (rys. 3.7b) będzie proponowane automatycznie. Ze strony logowania można trafić do strony rejestracji (rys. 3.7c) za pomocą odpowiedniego przycisku. Jeśli logowanie zostało proponowane automatycznie, dane poprzedniego ekranu nie będą usuwane, więc po zalogowaniu lub rejestracji można kontynuować pracę.

Po wprowadzeniu danych i wciśnięciu przycisku `Log in` przez użytkownika na stronie logowania, wszystkie pola muszą być niepuste, wysyła się zapytanie do serwera (opis API połączenia z serwerem 3.3.1) z danymi do logowania (listing 3.37). Obsługa przycisku zachodzi w metodzie `onButtonLoginClick` (listing 3.36). Przy pomyślnym zalogowaniu JTW token zachowuje się w pliku konfiguracyjnym, w celu automatycznego logowania



Rys. 3.7: Interfejs użytkownika: a) ustawienia, b) logowanie, c) rejestracja.

przy następnym wykorzystaniu aplikacji, oraz zamyka się fragment logowania, za pomocą klasy sterującej fragmentami (`FragmenManager`). Przy otwarciu fragmentu logowania lub rejestracji wcześniejszy fragment pozostaje pracować na dolnej warstwie fragmentów, co pozwala zachowywać wprowadzone dane. Cała klasa obsługująca znajduje się w pliku `main/java/com/example/testapp/fragments/LoginFragment.java`.

Listing 3.36: Obsługa przycisku login.

```
public void onButtonLoginClick(View v) {
    EditText email = (EditText) getView().findViewById(R.id.editTextEmail);
    EditText pass2 = (EditText) getView().findViewById(R.id.editTextPassword);
    if (email.getText().toString().equals("") || pass2.getText().toString().equals("")) {
        Helper.messageLogger(App.getAppContext(), Helper.LogType.NONE, "login", "Login or
            ↪ Password is incorrect");
    } else {
        User u = new User();
        u.setEmail(email.getText().toString());
        u.setPassword(pass2.getText().toString());
        login(u);
    }
}
```

Listing 3.37: Logowanie: wysłanie zapytania i obsługa odpowiedzi.

```
private void login(User user) {
    final LoginFragment tmpcls = this;
    disposable.add(app.getElchargeService().getUserApi().login(user)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribeWith(new DisposableSingleObserver<Response<User>>() {
                @Override
                public void onSuccess(Response<User> response) {
                    try {
                        if (response.code() == 200) {
                            User tmp = response.body();
                            String token = response.headers().get("Authorization");
                            app.getElchargeService().setToken(token);
                            app.getElchargeService().setUser(tmp);
                            Helper.messageLogger(App.getAppContext(), Helper.LogType.INFO,
                                ↪  "login", "LOGGED IN");
                            getFragmentManager().beginTransaction().remove(tmpcls).commit
                                ↪ ();
                        } else {
```

```
                        Helper.messageLogger(App.getAppContext(), Helper.LogType.INFO,
                            ↪ "login", response.message());
                    }

                } catch (Exception e) {
                    Helper.messageLogger(App.getAppContext(), Helper.LogType.ERR, "
                        ↪ login", e.getMessage());
                }
            }

            @Override
            public void onError(Throwable e) {
                Helper.messageLogger(App.getAppContext(), Helper.LogType.ERR, "login",
                    ↪ e.getMessage());
            }
        }));
    }
```

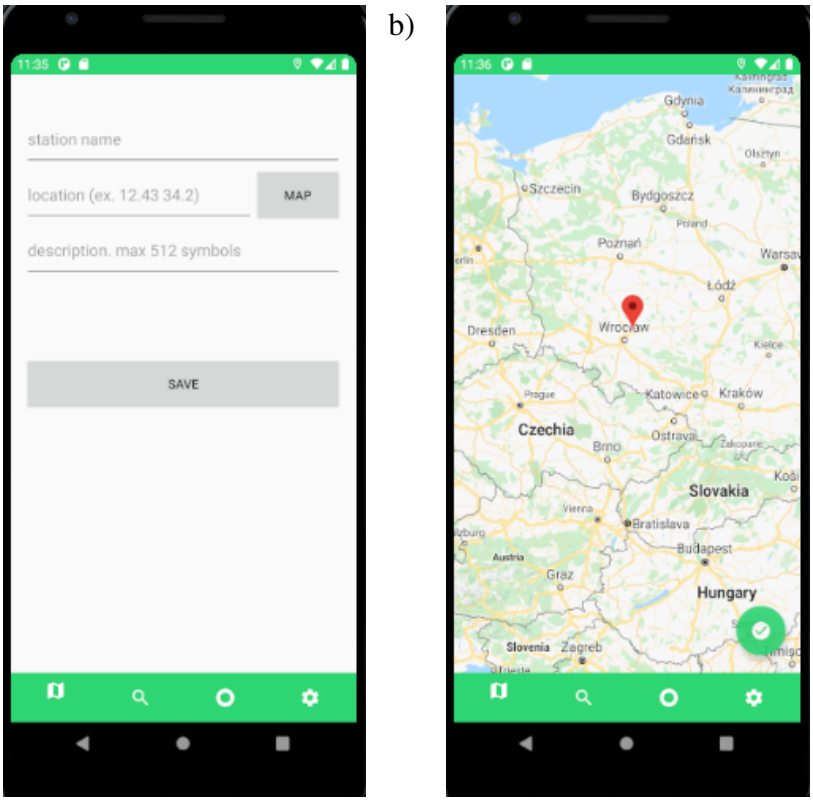
Rejestracja działa w podobny sposób do logowania (opis: 3.3.2): sprawdzanie wprowadzonych przez użytkownika danych oraz komunikacja z częścią serwerową. Cała klasa obsługująca rejestrację znajduje się w pliku `main/java/com/example/testapp/fragments/RegisterFragment.java`.

Stacja

Dodawanie

Jedną z najważniejszych funkcji jest tworzenie stacji ładowania. Dla jej tworzenia stworzony fragment aplikacji dostępny pod przyciskiem kółka (trzeci znaczek w menu pod spodem) (rys. 3.8a). Za pomocą przycisku `MAP` można nie wprowadzać współrzędne ręcznie, a ustalić miejsce na za pomocą mapy (rys. 3.8b). Za pracę ze stroną zawierającą interaktywną mapą odpowiada fragment `fragment_maps_create_station.xml` oraz kalsa `MapsSelectFragment`.

a) b)



Rys. 3.8: Interfejs użytkownika: a) tworzenie stacji, b) ustalenie markera na mapie.

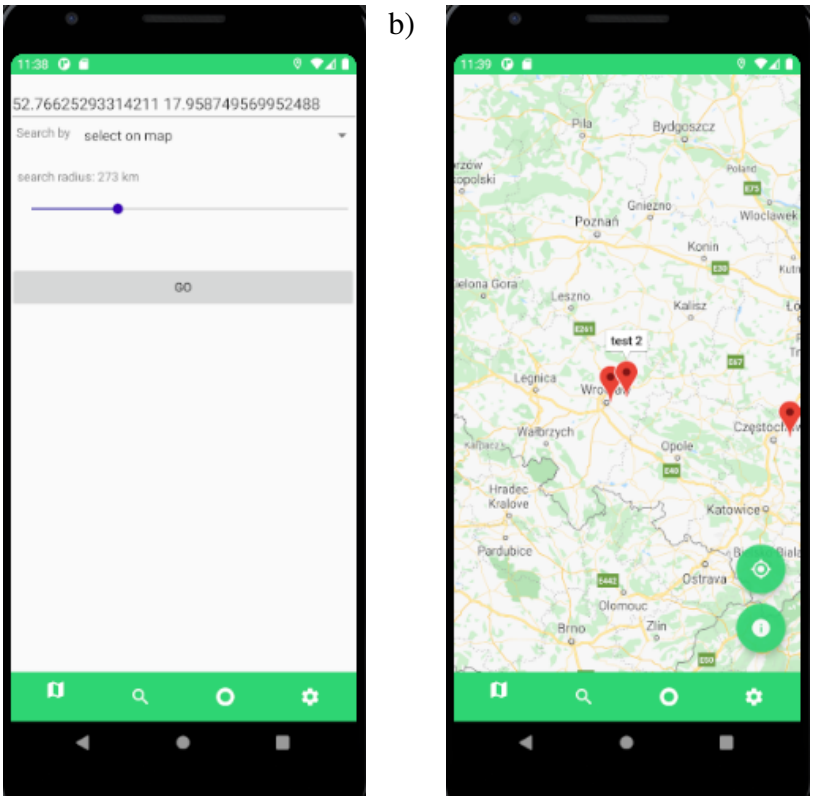
Naciśnięcie przycisku `SAVE` powoduje sprawdzenie wprowadzonych danych oraz wysłanie odpowiedniego zapytania do części serwerowej w celu tworzenia nowego wpisu w bazie

danych. Jeśli użytkownik nie został zalogowany lub zarejestrowany, wyświetla się okno logowania lub rejestracji (rys. 3.7b, 3.7c). Klasa obsługująca ten fragment znajduje się w pliku `main/java/com/example/testapp/fragments/CreateStationFragment.java`.

Wyszukiwanie

Najważniejszą funkcją aplikacji jest wyszukiwanie stacji ładowania do pojazdów elektrycznych. W tym celu został stworzony przycisk w dolnym menu, który wygląda jak lupa (rys. 3.9a). Na tym ekranie można wyszukać stacje ładowania na podstawie: wprowadzonych współrzędnych, ustaleniu miejsca na mapie (rys. 3.8b), niniejszej pozycji użytkownika, według części nazwy lub opisu stacji ładowania. Przy wyszukiwaniu według różnego rodzaju kordynat, istnieje możliwość ustalenia dystansu wyszukiwania za pomocą suwaka. Naciskając na przycisk `GO` zalogowany użytkownik trafi na stronę z mapą, gdzie będą oznaczone wyszukane stacje (rys. 3.9b).

a) b)



Rys. 3.9: Interfejs użytkownika: a) wyszukiwanie stacji, b) lista znalezionych stacji.

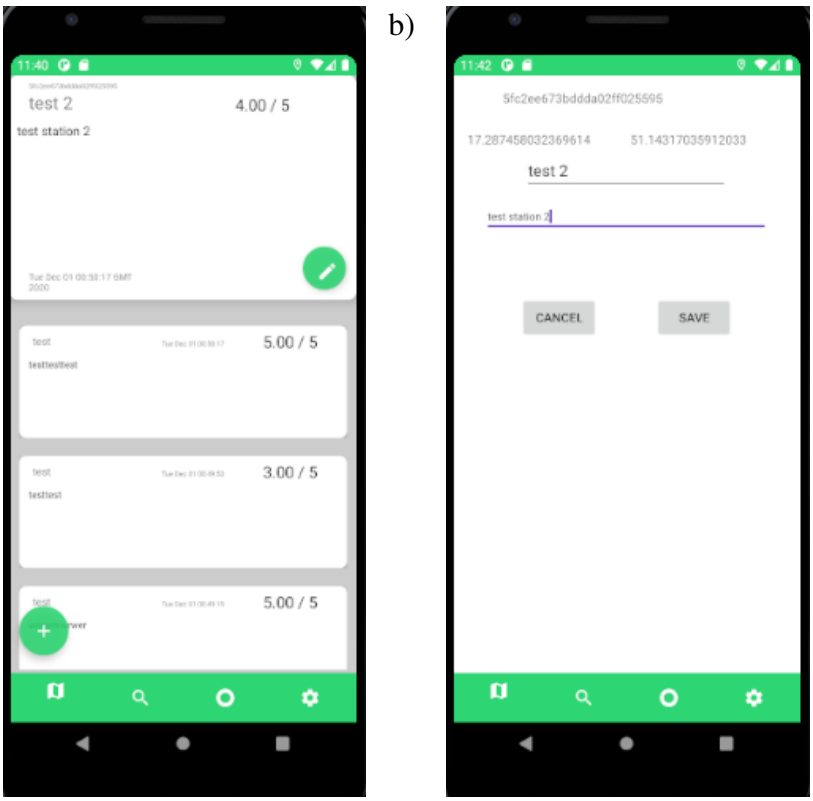
W zależności od wybranego, za pomocą spadającego menu, trybu wyszukiwania, będzie wykorzystany odpowiedni API. W przypadku wybory automatycznego wprowadzenia pozycji użytkownika sprawdza się czy aplikacja ma odpowiednie uprawnienia (`ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION`). Jeśli brak tych uprawnień, odpytuje użytkownika o zezwoleniu na korzystanie z nich. `FusedLocationProviderClient` (część SDK Google Mobile Services) pozwala na określenie pozycji nie tylko za pomocą modułu GPS, ale i IP adresie. W przypadku nieudanego otrzymania niniejszej pozycji zostają użyte ostatnie wiadome współrzędne. Użytkownik też może ustalić marker do wyszukiwania na mapie, wtedy jest wywołany fragment do rysowania mapy (rys. 3.8b) i po ustaleniu pozycji, współrzędne będą automatycznie wprowadzone.

Wczytywanie

Otrzymanie informacji dotyczącej stacji ładowania jest dostępne po wyborze markera na

mapie i naciśnięciu przycisku `info` na tym ekranie (rys. 3.9b). Zostanie wysłano zapytanie do serwera z prośbą o zwróceniu informacji dotyczącej wybranej stacji ładowania. Po otrzymaniu odpowiedzi zostanie wyświetlony fragment z nazwą stacji, jej `id`, opisem, datą ostatniej modyfikacji, oceną, wyliczoną na podstawie komentarzy, i lista komentarzy napisanych do tej stacji w kolejności: najpierw najnowsze. Komentarze zawierają `user_name`, tekst, ocenę oraz datę. Ten fragment jest pokazany na rysunku 3.10a.

a) b)



Rys. 3.10: Interfejs użytkownika: a) informacja o stacji, b) edycja stacji.

Do otrzymania aktualnej informacji została zrealizowana metoda `updateInfo()` w której wysyła się zapytanie do części serwerowej. Klasa wewnętrzna `StationDisposableSingleObserver` (listing 3.38), która dziedziczy po klasie abstrakcyjnej `DisposableSingleObserver`, służy do opracowania i wyświetlania informacji (metoda `updateInfoOnView()`) na ekranie (listing 3.39). Oczekiwanie na odpowiedź z serwera zachodzi asynchronicznie.

Listing 3.38: Wczytanie danych stacji z części serwerowej.

```
public void updateInfo(){
    disposable.add(app.getElchargeService().getStationApi().readStationsByLatAndLng(0, 0,
        ↪ currentStation.getLatitude(), currentStation.getLongitude())
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribeWith(new StationDisposableSingleObserver(this)));
}

private class StationDisposableSingleObserver extends DisposableSingleObserver<Response<
    ↪ Station>> {
    private InfoFragment parent;

    public StationDisposableSingleObserver(InfoFragment parent) {
        this.parent = parent;
    }

    @Override
    public void onSuccess(Response<Station> response) {
        try {
```

```
            if (response.code() == 200) {
                Station station = response.body();
                Helper.messageLogger(App.getAppContext(), Helper.LogType.INFO, "station",
                    ↪ Integer.toString(response.code()));
                currentStation = station;
                updateInfoOnView();
            } else {
                Helper.messageLogger(App.getAppContext(), Helper.LogType.INFO, "station",
                    ↪ Integer.toString(response.code()));
                if (response.code() == 401) {
                    LoginFragment lf = new LoginFragment();
                    getFragmentManager().beginTransaction().add(R.id.container, lf).commit
                        ↪ ();
                    getFragmentManager().beginTransaction().show(lf).commit();
                }else{
                    getFragmentManager().beginTransaction().remove(parent).commit();
                }
            }

        } catch (Exception e) {
            Helper.messageLogger(App.getAppContext(), Helper.LogType.ERR, "station", e.
                ↪ getMessage());
        }
    }

    @Override
    public void onError(Throwable e) {
        Helper.messageLogger(App.getAppContext(), Helper.LogType.ERR, "station", e.
            ↪ getMessage());
    }
}
```

Listing 3.39: Odnowienie informacji o stacji na ekranie.

```
private void updateInfoOnView(){
    if (app.getElchargeService().getUser().getId().equals(currentStation.getOwnerID())){
        buttonEditStation.show();
    }else {
        buttonEditStation.hide();
    }
    recyclerView = getView().findViewById(R.id.recyclerViewComments);
    recyclerView.setLayoutManager((new LinearLayoutManager(App.getAppContext())));

    commentsAdapter = new CommentsAdapter(currentStation.getComments());
    recyclerView.setAdapter(commentsAdapter);

    ((TextView) getView().findViewById(R.id.stationId)).setText(currentStation.getId());
    ((TextView)getView().findViewById(R.id.stationName)).setText(currentStation.
        ↪ getStationName());
    ((TextView)getView().findViewById(R.id.rating)).setText(String.format("%.2f",
        ↪ currentStation.getRating()) + " / 5");
    ((TextView)getView().findViewById(R.id.description)).setText(currentStation.
        ↪ getDescription());
    ((TextView)getView().findViewById(R.id.date)).setText(Helper.getDateFromISO8601(
        ↪ currentStation.getUpdateAt()));
}
```

Edycja

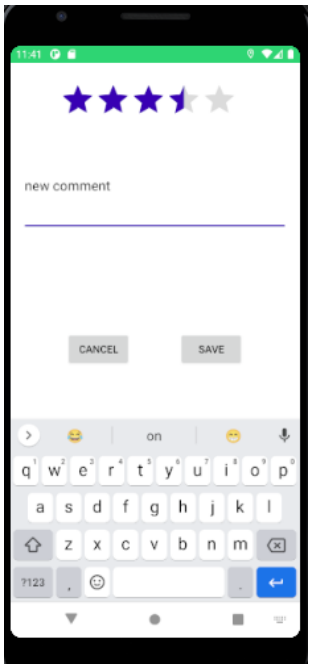
Tylko użytkownik, który stworzył stację ładowania do pojazdów elektrycznych, ma możliwość modyfikacji stacji ładowania. Jeśli twórca przegląda swoją stację, obok opisu tej stacji istnieje przycisk (rys. 3.10a), za pomocą którego można wejść do strony modyfikacji (rys. 3.10b). Modyfikacji podlegają tylko nazwa i opis. Przycisk `CANCEL` przewróci na stronę informacji o stacji ładowania. Komunikacja z serwerem zachodzi na tej samej zasadzie, co już opisana wyżej, w części logowania i rejestracji.

Komentarz

Komentarzy mogą pisać i przeglądać zalogowane użytkowniki podczas przeglądania informacji o stacji ładowania. Oni są niezbędne do wystawiania oceny stacji.

Dodawanie

Dodać komentarz można na stronie informacyjnej stacji ładowania (rys. 3.10a) za pomocą odpowiedniego przycisku (plus w kółku). Na rysunku 3.11 przedstawiony ekran tworzenia komentarza. Zasada działania jest prosta: walidacja wpisanej informacji, wysłanie zapytania



Rys. 3.11: Tworzenie komentarza

do serwera, asynchroniczne oczekiwanie na odpowiedź, przy udanym tworzeniu — powrót do strony zawierającej informację o stacji ładowania pojazdów elektrycznych. Użytkownik widzi juz zmieniona informację.

Rozdział 4

Testy

W niniejszym rozdziale zostały przedstawione listy przeprowadzonych testów oraz kilka ich przykładów.

4.1. Testy jednostkowe

Aby przeprowadzić testy po stronie serwera, wystarczy uruchomić polecenie `make test` w folderze 'RestAPI'. Do testowania użyto biblioteki „stretchr/testify". Pliki przeznaczone do testowania znajdują się obok testowanych plików. Różnią się one od plików aplikacji nazwą: *_test.go.

Kod 4.1 pokazuje przykład testowania walidacji danych dla jednostki użytkownika, która jest używana na przykład przed dodaniem użytkownika do bazy danych.

Listing 4.1: Kod testowania walidacji danych użytkownika

```
func TestValidate(t *testing.T) {
    testCase := []struct {
        name    string
        user    *User
        isValid bool
    }{
        {
            name: "simple",
            user: &User{
                UserName: "valid",
                Email:    "valid@email.com",
                Password: "validpass",
            },
            isValid: true,
        },
        {
            name: "without username",
            user: &User{
                Email:    "valid@email.com",
                Password: "validpass",
            },
            isValid: true,
        },
        {
            name: "without email",
            user: &User{
                UserName: "valid",
                Password: "validpass",
            },
            isValid: false,
        },
        {
            name: "without password",
            user: &User{
                UserName: "valid",
                Email:    "valid@email.com",
```

```
            },
            isValid: false,
        },
        {
            name: "wrong email",
            user: &User{
                UserName: "valid",
                Email:    "wrong",
                Password: "validpass",
            },
            isValid: false,
        },
        {
            name: "wrong password",
            user: &User{
                UserName: "valid",
                Email:    "valid@email.com",
                Password: "wrong",
            },
            isValid: false,
        },
        {
            name: "wrong username",
            user: &User{
                UserName: "n",
                Email:    "valid@email.com",
                Password: "validpass",
            },
            isValid: false,
        },
    }

    for _, item := range testCase {
        t.Run(item.name, func(t *testing.T) {
            if item.isValid {
                assert.NoError(t, item.user.Validate())
            } else {
                assert.Error(t, item.user.Validate())
            }
        })
    }
}
```

Do testowania funkcji, które nie wymagają bezpośredniego połączenia z MongoDB, używany jest pakiet „storage/teststorage” z bazą danych zaimplementowanej jako mapa klucz-wartość (na przykład warstwie serwisów nie ma różnicy, z jaką bazą danych pracować). Przeprowadzono również testy interakcji z rzeczywistą bazą danych.

Przykładem testowania tworzenia użytkownika w bazie danych jest kod 4.2:

Listing 4.2: Kod testowania tworzenia użytkownika w MongoDB

```
func TestCreate(t *testing.T) {
    ur := testHelperUser()
    ti := models.GetTimeNow()
    user := &models.User{
        UserName: "username_1",
        Email:    "1@email.com",
        Password: "passwoed_1",
        Model: models.Model{
            UpdateAt: ti,
            CreateAt: ti,
        },
    }
    id, err := ur.Create(user)
    assert.Nil(t, err)
    assert.NotEqual(t, id, "")
    ur.DeleteByID(id)
}
```

Przykład testowania tworzenia użytkownika w fikcyjnej bazie danych 4.3.

Listing 4.3: Kod testowania tworzenia użytkownika w fikcyjnej bazie danych

```
func TestCreate(t *testing.T) {
    ur := NewUserRepository()
```

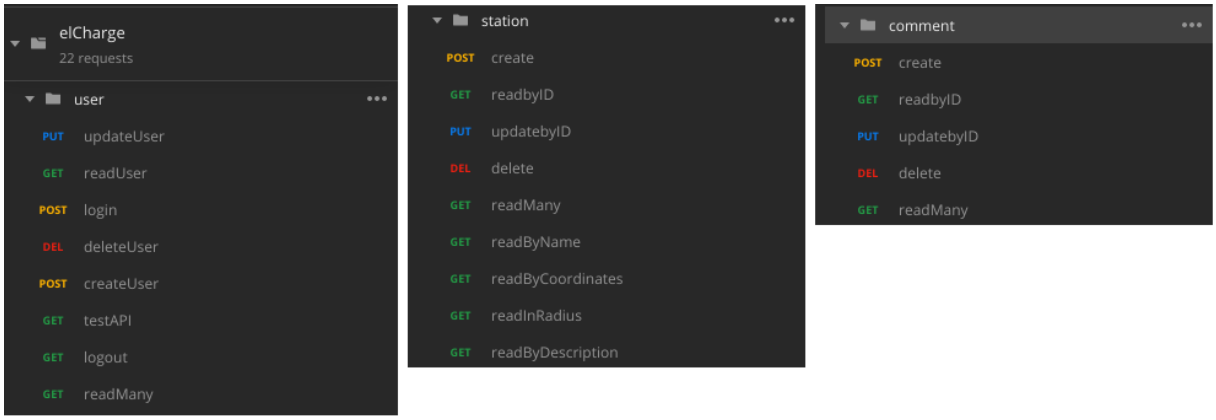
```
        ti := models.GetTimeNow()
        user := &models.User{
            UserName: "username_1",
            Email:    "1@email.com",
            Password: "passwoed_1",
            Model: models.Model{
                UpdateAt: ti,
                CreateAt: ti,
            },
        }
        id, err := ur.Create(user)
        assert.Nil(t, err)
        assert.NotEqual(t, id, "")
        user.ID = id
        assert.Equal(t, ur.db[id], user)
    }
```

Lista wszystkich testów jednostkowych znajduje się na rysunku 4.1.

Rys. 4.1: Lista testów jednostkowych

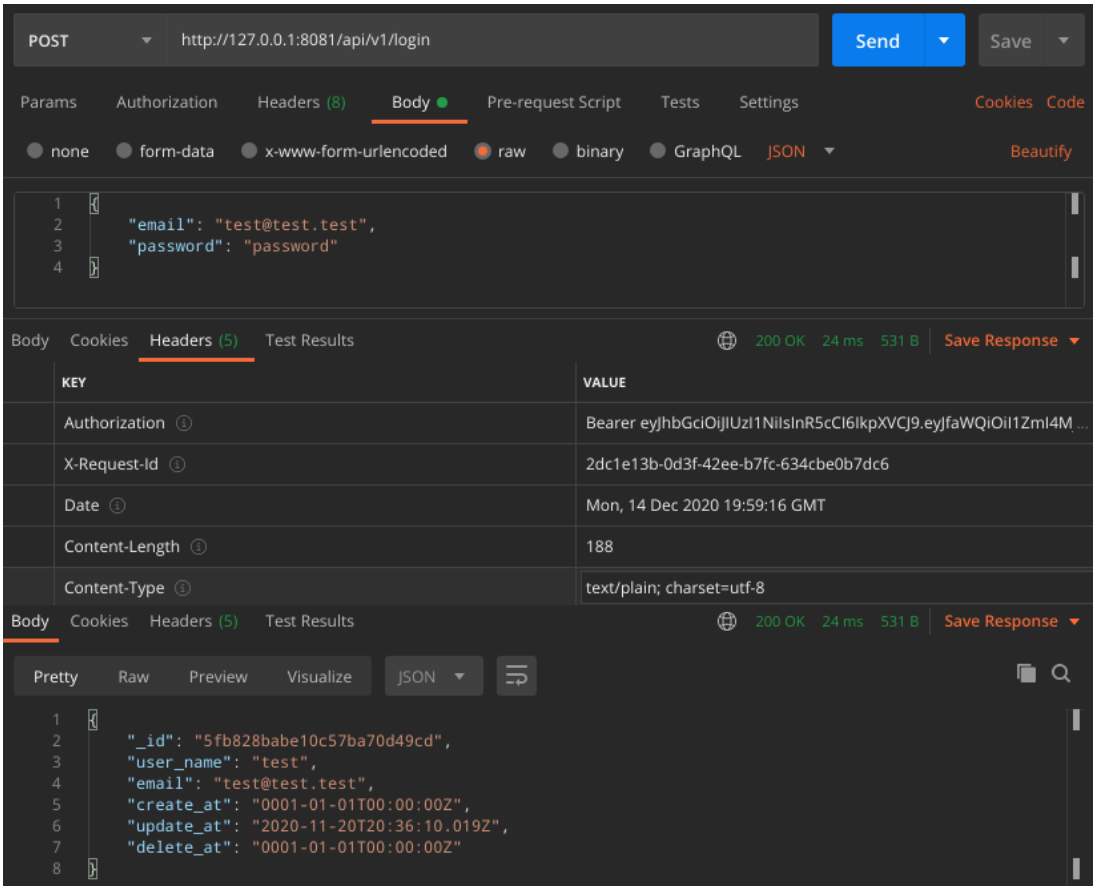
4.2. Testy API

Do testowania interfejsu API wykorzystano narzędzie do automatycznego testowania Postman. Przeprowadzono testy wszystkich endpointów (rys. 4.2).



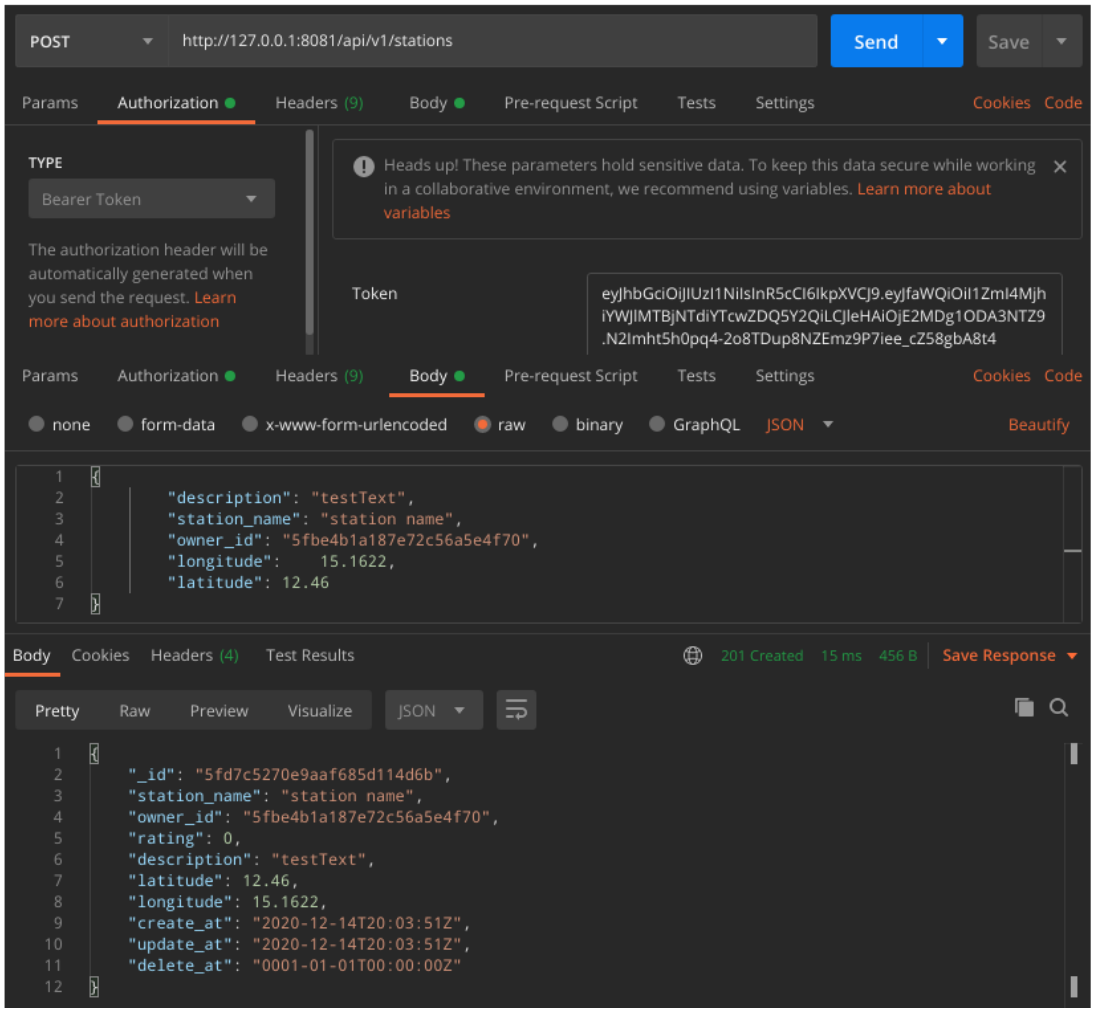
Rys. 4.2: Lista testów API

Przykład testowania logowania (rys. 4.3):



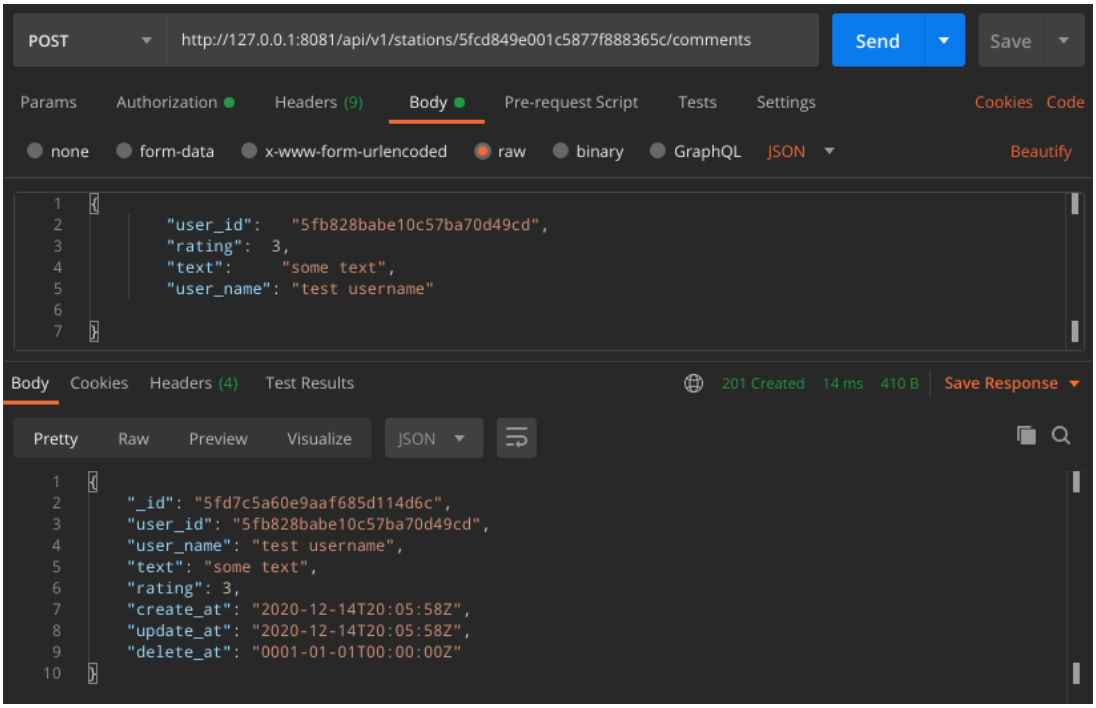
Rys. 4.3: Testowanie logowania za pomocą Postman

Przykład testowania tworzenia stacji ładującej samochody elektryczne (rys. 4.4).
Przykład tworzenia komentarza (rys. 4.5) i jego modyfikacji (rys. 4.6).

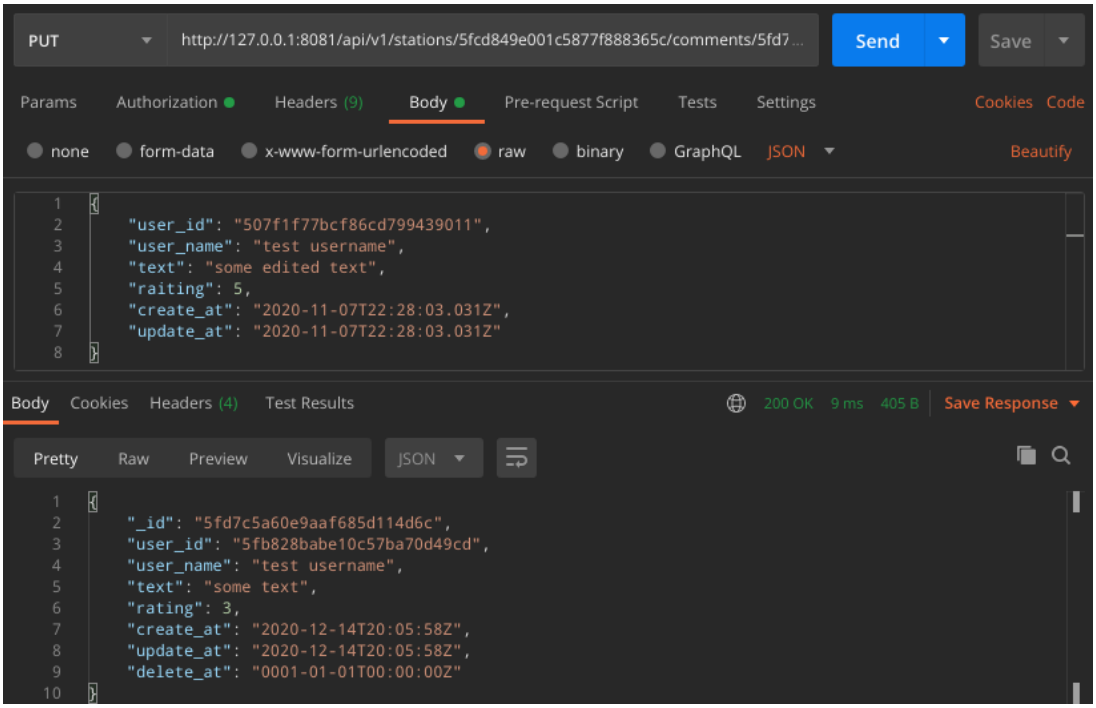


Rys. 4.4: Testowanie tworzenia stacji ładowanicej za pomocą Postman

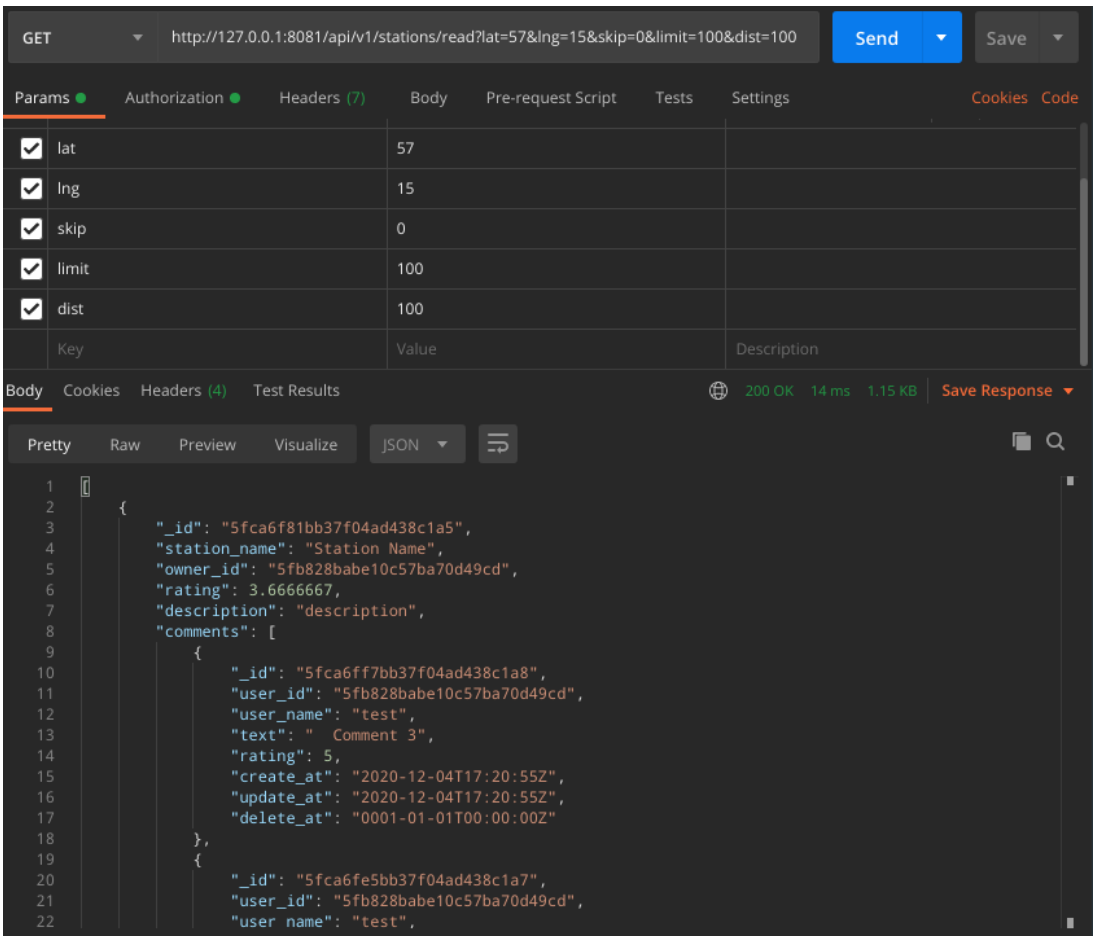
Przykład wyszukiwania stacji ładującej (rys. 4.7) w dystansie 100 km od ustalonych współrzędnych, a także ograniczenia listy:



Rys. 4.5: Testowanie tworzenia komentarza za pomocą Postman



Rys. 4.6: Testowanie edycji komentarza za pomocą Postman



Rys. 4.7: Testowanie złożonego wyszukiwania stacji ładowniczej za pomocą Postman

Rozdział 5

Podsumowanie

W tym rozdziale znajdują się wnioski i plany na rozwój aplikacji.

5.1. Wnioski

W tym projekcie planowano rozwiązać problem ze znalezieniem stacji ładowania pojazdów elektrycznych. Miała zostać stworzona aplikacja mobilna, w której użytkownik może nie tylko szybko i wygodnie wyszukiwać stację ładownicze, które mu odpowiadają, ale także dodawać nowe, których jeszcze nie ma w systemie. Umożliwiłoby to stworzenie samowystarczalnej aplikacji, która nie wymagałaby dużych inwestycji w celu utrzymania infrastruktury. Aplikacja miała umożliwić tworzenie komentarzy i ocenianie stacji ładowania pojazdów elektrycznych. Dla ułatwienia wyszukiwania miała być użyta mapa. Na mapie oprócz stacji ładowania miała być zaznaczona m.in. pozycja użytkownika.

W aplikacji zostały zaimplementowane wszystkie powyższe funkcje, ale w procesie rozwoju zidentyfikowano pewne niezbędne funkcjonalności, które zostaną określone w planach 5.2 na przyszłość.

Dzięki wybranej bazie danych MongoDB aplikacja może być łatwo rozbudowywana o nowe kolekcje i zmienione już istniejące. Wszystkie wybrane technologie okazały się bardzo elastyczne i odpowiednie przystosowane do tego zadania. Najbardziej użytecznym źródłem informacji była oficjalna dokumentacja wybranych narzędzi [3, 24, 15, 11]. Też była użyteczna książka o języku Go [13] oraz książka o języku Java [17]

5.2. Plany

Bezpieczeństwo:

- Tworzenie panelu admina do moderacji komentarzy;
- Zamknięcie dostępu dowolnego użytkownika do danych, które nie są dozwolone za pośrednictwem aplikacji mobilnej, to znaczy zamknięcie API;
- Ulepszenie niektórych endpointów, na przykład sposobu wyszukiwania według parametrów w endpoincie `/api/v1/stations/read?skip=&limit=&lat=&lng=&dist=&descr=&nam=`.

Wygoda użytkowania aplikacją:

- Dodawanie możliwości usuwania komentarzy za pośrednictwem aplikacji mobilnej;
- Rozszerzenie możliwości pracy z mapą za pomocą usług google cloud. Na przykład wyznaczanie trasy;

Literatura

[1] Accelerate developer productivity. https://gradle.org [Online; accessed 03-December-2020].

[2] Android Studio provides the fastest tools for building apps on every type of Android device. https://developer.android.com/studio [Online; accessed 03-December-2020].

[3] Build anything on Android. https://developer.android.com [Online; accessed 03-December-2020].

[4] O. Bunin. MongoDB survival guide, Czerw. 2019. https://habr.com/ru/company/oleg-bunin/blog/454748/ [Online; accessed 05-December-2020].

[5] Configure your build, Paz. 2020. https://developer.android.com/studio/build/index.html [Online; accessed 03-December-2020].

[6] diagrams.net. https://www.diagrams.net/about.html [Online; accessed 30-November-2020].

[7] Documentation. https://golang.org/doc/ [Online; accessed 03-December-2020].

[8] Documentation. https://redis.io/documentation [Online; accessed 03-December-2020].

[9] Electric Vehicle Market by Vehicle (Passenger Cars & Commercial Vehicles), Vehicle Class (Mid-priced & Luxury), Propulsion (BEV, PHEV & FCEV), EV Sales (OEMs/Models) Charging Station (Normal & Super) & Region - Global Forecast to 2030, Czerw. 2019. https://www.marketsandmarkets.com/Market-Reports/electric-vehicle-market-209371461.html#:~:text=The%20Electric%20Vehicles%20Market%20is,is%20from%202019%20to%202030. [Online; accessed 30-November-2020].

[10] Electric Vehicle Market by Vehicle (Passenger Cars & Commercial Vehicles), Vehicle Class (Mid-priced & Luxury), Propulsion (BEV, PHEV & FCEV), EV Sales (OEMs/Models) Charging Station (Normal & Super) & Region - Global Forecast to 2030, Paz. 2020. https://www.iea.org/data-and-statistics/charts/global-electric-car-stock-2010-2019 [Online; accessed 10-Decemberer-2020].

[11] Get started with MongoDB. https://docs.mongodb.com [Online; accessed 03-December-2020].

[12] Getting Started. https://code.visualstudio.com/docs [Online; accessed 03-December-2020].

[13] M. Gieben. Learning Go, 2018. https://www.miek.nl/go/ [Online; accessed 05-December-2020].

[14] Global EV Outlook 2020, Czerw. 2020. https://www.iea.org/reports/global-ev-outlook-2020 [Online; accessed 30-November-2020].

[15] Go by Example. https://gobyexample.com [Online; accessed 03-December-2020].

[16] Google Cloud. https://cloud.google.com [Online; accessed 03-December-2020].

[17] C. S. Horstmann. *Core Java Volume I–Fundamentals (10th Edition)*. Prentice Hall, 2016.

[18] Introduction to JSON Web Tokens. https://jwt.io/introduction/ [Online; accessed 03-December-2020].

[19] javadoc.io. https://www.javadoc.io [Online; accessed 03-December-2020].

[20] Maps SDK for Android, List. 2020. https://developers.google.com/maps/documentation/android-sdk/overview [Online; accessed 03-December-2020].

[21] Orientation and setup. https://docs.docker.com/get-started/ [Online; accessed 03-December-2020].

[22] Pricing that scales to fit your needs. https://cloud.google.com/maps-platform/pricing [Online; accessed 08-December-2020].

[23] RESTful API Design: 13 Best Practices to Make Your Users Happy, Sier. 2018. https://florimond.dev/blog/articles/2018/08/restful-api-design-13-best-practices-to-make-your-users-happy/ [Online; accessed 03-Decemberer-2020].

[24] Search for Go Packages. https://godoc.org [Online; accessed 03-December-2020].

[25] The global electric vehicle market in 2020: statistics & forecasts, 2019. https://leftronic.com/android-vs-ios-market-share/#Top_Mobile_OS [Online; accessed 30-November-2020].

[26] The Go Project. https://golang.org/project/ [Online; accessed 03-December-2020].

[27] What is a Container? https://www.docker.com/resources/what-container [Online; accessed 03-December-2020].

[28] What Is MongoDB? https://www.mongodb.com/what-is-mongodb [Online; accessed 03-December-2020].

[29] What is REST. https://restfulapi.net [Online; accessed 03-Decemberer-2020].

Dodatek A

Opis załączonej płyty CD/DVD

W katalogu `ElCharge` znajduje się kod źródłowy trzech części projektu: część serwerowa (katalog `RestAPI`), mobilna aplikacja (katalog `AndroidUI`), praca dyplomowa (katalog `Documentation`).

Dla wdrożenia części serwerowej w katalogu `ElCharge/RestAPI` znajduje się plik `docker-compose.yaml`. Istnieje też wersja skompilowana w katalogu `ElCharge/RestAPI/deploy`. Skompilowana wersja aplikacji mobilnej znajduje się w katalogu `ElCharge/AndroidUI/app/build/outputs/apk/debug/` i jest plikiem `elCharge.apk`. Wersja elektroniczna pracy inżynierskiej: `ElCharge/Documentation/W04_245816_2020_praca inżynierska.pdf`

Dodatek B

Wdrożenie aplikacji

B.1. Wdrożenie części serwerowej

RestAPI Do wdrożenia części serwerowej wraz z bazami danych MongoDB i Redis wystarczy wykonać dwa polecenia. Musi być zainstalowany Docker i Docker-compose.

Pierwsze polecenie tworzy 3 obrazy: obraz bazy danych dunnh MongoDB, obraz bazy danych Redis i obraz serwerowej części aplikacji napisanej w języku Go:

`$ docker-compose build`

Drugie polecenie uruchamia wcześniej utworzone obrazy w określonej kolejności, a także tworzy kanały do interakcji części aplikacji między kontenerami oraz środowyskiem wewnętrznym.

`$ docker-compose up`

Do zarządzania kontenerami służy plik `docker-compose.yml` B.1:

Listing B.1: docker-compose.yml

```
version: "3.5" # Use version 3.5 syntax
services: # Here we define our service(s)
    db:
        container_name: mongoDB-elcharge # Container name
        image: mongo # image name to start/build
        ports: # Port mapping
            - "2717:27017"
        volumes: # Volume binding
            - "~/example:/data/db"
    cache-db:
        container_name: redis-elcharge # Container name
        image: redis
        ports:
            - "63799:6379"
        volumes: # Volume binding
            - "/opt/redis/data:/data"
    golang-restapi: # The name of the service
        build:
            context: .
            dockerfile: Dockerfile # Location of our Dockerfile
        image: despenrado/golang-restapi-elcharge:prod.0.1
        container_name: restapi-elcharge # Container name
        depends_on: # start after
            - cache-db
            - db
        ports:
            - "8081:8081"
        links: # list mapping: service_name:name_how_will_see_your_program
            - "db:mymongo"
            - "cache-db:myredis"
```

Plik Dockerfile B.2 jest używany do etapów instalacji zależności, kompilacji i samego tworzenia obrazu zaplecza aplikacji:

Listing B.2: Dockerfile

```
FROM golang:1.15 AS builder
WORKDIR /go/src/github.com/Ddespenrado/ElCharge/RestAPI/
COPY . .
RUN go mod tidy
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -a -tags netgo -ldflags '-w' -o restapi
    ↪ .

FROM scratch
EXPOSE 8081
WORKDIR /root/
COPY deploy/apiserver.yaml .
COPY --from=builder /go/src/github.com/Ddespenrado/ElCharge/RestAPI/restapi .
ENTRYPOINT ["./restapi","-config=apiserver.yaml"]
```

Plikiem konfiguracyjnym aplikacji, w przypadku używania dockera jest plik `RestAPI/deploy/apiserver.yaml`.

B.2. Instalacja i uruchomienie aplikacji mobilnej

Najpierw trzeba skopiować plik `elCharge.apk` do urządzenia z systemem Android, znaleźć ten plik, otworzyć (kliknąć na niego) oraz zgodzić się na instalację. Uruchomić aplikację `TestApp`.

W przypadku uruchomiania za pomocą emulatora, wbudowanego w Android Studio, należy mieć zainstalowany Android Studio oraz Java SDK 8. Należy otworzyć katalog `AndroidUI` za pomocą Android Studio, wybrać emulator z systemem Android powyżej wersji 4.1 i uruchomić za pomocą odpowiedniego przycisku.