ABSTRACT

The topic *Mobile application for user location and tracking to improve his security* was developed as part of the engineering project. The project aim is to contribute to the sense of security and greater freedom of movement for users using it. The main goal of the engineering project is to design and implement a mobile application for the Android platform that can offer to send information to friends about the situation and location, where the user is. Additionally, a friend should easily be informed on how to get to the destination. The scope of the project includes the analysis of knowledge on the design of similar applications, the creation of a design analysis and the determination of functional, non-functional requirements and use cases. The next step is to complete the system design and define the architecture and schemas of classes and databases. In addition, the scope of the project includes the implementation of the application and its testing. The result of the engineering project was checked by using test scenarios, which were prepared as a part of functional testing. Most of the assumed functional requirements, including all of the highest priority, have been achieved. Users can easily share the location and their friends are quickly informed about it. Mobile applications can contribute to the improvement of many aspects of life, including increasing user safety.

Keywords: Informatics, Android, Mobile application, Location sharing, Maps, Security.

SPIS TREŚCI

1. WSTĘP I CEL PRACY

Sytuacje niebezpieczne powodujące uszczerbek na zdrowiu, a także zagrażające życiu w dzisiejszym świecie występują coraz częściej. Dzieje się to mimo coraz lepszej wykrywalności przestępców. Wieczorne wyjście, szczególnie samotne jest bardzo ryzykowne i może skutkować nieprzyjemnymi sytuacjami. W takim momencie należy jak najszybciej powiadomić o tym znajomych i poinformować ich o swojej lokalizacji, aby wiedzieli, gdzie taką osobę szukać. Nowoczesna technologia może nam bardzo ułatwić to zadanie. Wystarczy do tego urządzenie mobilne oraz dostęp do internetu. Są to rzeczy powszechnie dostępne w społeczności. Istnieją na rynku aplikacje zawierające takie funkcje czego przykładem jest bardzo popularna aplikacja *Google Maps* od firmy *Alphabet*. Jednak zazwyczaj ta funkcjonalność jest jedną z wielu, których skonfigurowanie i do których dotarcie w interfejsie zajmuje zbyt wiele czasu, a jest on niezbędny w awaryjnych sytuacjach.

Bezpieczeństwo człowieka jest dla nas, czyli twórców projektu najważniejszym priorytetem, a przy okazji również dla dużej części społeczeństwa. Chcemy, aby projekt był przydatny nie tylko dla małej, ściśle określonej grupy użytkowników.

Dlatego celem projektu inżynierskiego jest zaprojektowanie i implementacja aplikacji mobilnej na platformę Android, która pomoże jej użytkownikowi zwiększyć poczucie bezpieczeństwa. Aplikacja powinna oferować szybkie informowanie znajomych o sytuacji i lokalizacji. Realizowane może być to przy pomocy odpowiedniego przycisku w aplikacji. Ponadto znajomy w łatwy sposób powinien być poinformowany jak dotrzeć do otrzymanego miejsca. Projekt skupia się na systemie Android ze względu na jego popularność w urządzeniach mobilnych, szczególnie na terenie Polski. Aplikacja powinna być skierowana do polskojęzycznych użytkowników.

Projekt inżynierski tworzony jest przez trzy osoby tj. Kseniya Shastak, Andrei Shastak i Maciej Teclaf. Praca nad aplikacją rozpoczęła się już na początku marca 2020 roku, a prace nad projektem inżynierski zakończono pod koniec listopada tego samego roku. Na początku przystąpiono do analizy i projektowania działania aplikacji i jej architektury. Następną fazą była implementacja, którą rozpoczęto w sierpniu tegoż roku, a do testowania przystąpiliśmy po zakończeniu prac implementacyjnych w listopadzie 2020 roku.

Praca inżynierska charakteryzuje się następującą strukturą: w drugim rozdziale opisana jest wiedza jaką pozyskano i wykorzystano w realizacji projektu. Zawarta została tu główna część teoretyczna pracy inżynierskiej. Trzeci rozdział zawiera informacje na temat analizy projektowej. Zdefiniowano tu wymagania funkcjonalne i pozafunkcjonalne aplikacji oraz przypadki użycia. Rozdział czwarty omawia wykonany projekt aplikacji. Omówiona została tu architektura systemu, logika biznesowa aplikacji klienckiej oraz serwerowej wraz z diagramami ich klas oraz struktura baz danych wykorzystanych do przetrzymywania informacji użytkowników. Piąty rozdział opisuje przebieg implementacji projektu. Zawarto w nim informacje na temat narzędzi wykorzystanych do napisania aplikacji oraz przedstawiono jak zaimplementowano najważniejsze funkcjonalności. Przy okazji ukazane zostały tu zrzuty ekranu z działającej aplikacji. Szósty rozdział opisuje

sposób testowania projektu i wykorzystane przypadki testowe. Następnie podsumowano pracę inżynierską i podano jej bibliografię oraz spisy tabel i rysunków.

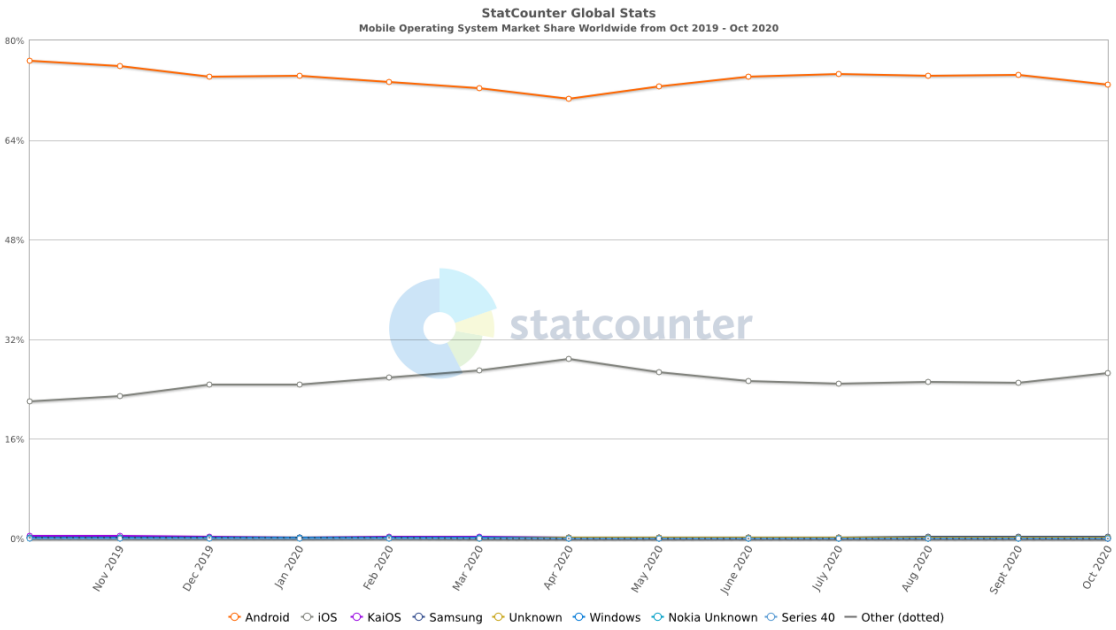
2. ANALIZA STANU WIEDZY

W tym rozdziale przystąpiono do analizy wiedzy potrzebnej w realizacji planowanego projektu. Opisano tu platformę Android na której realizowana będzie aplikacja mobilna oraz technologie wykorzystywane do tworzenia aplikacji serwerowej. Ponadto przedstawiono możliwości komunikacji między tymi elementami projektu.

2.1. Android

Android to system operacyjny oparty na jądrze Linuksa zaprojektowany z myślą o urządzeniach mobilnych [1]. Aktualnie jednak nie jest on przeznaczony tylko na telefony i tablety. Swoje zastosowanie znalazł również w samochodach, telewizorach i inteligentnych zegarkach, które z angielskiego nazywane są jako smartwatch.

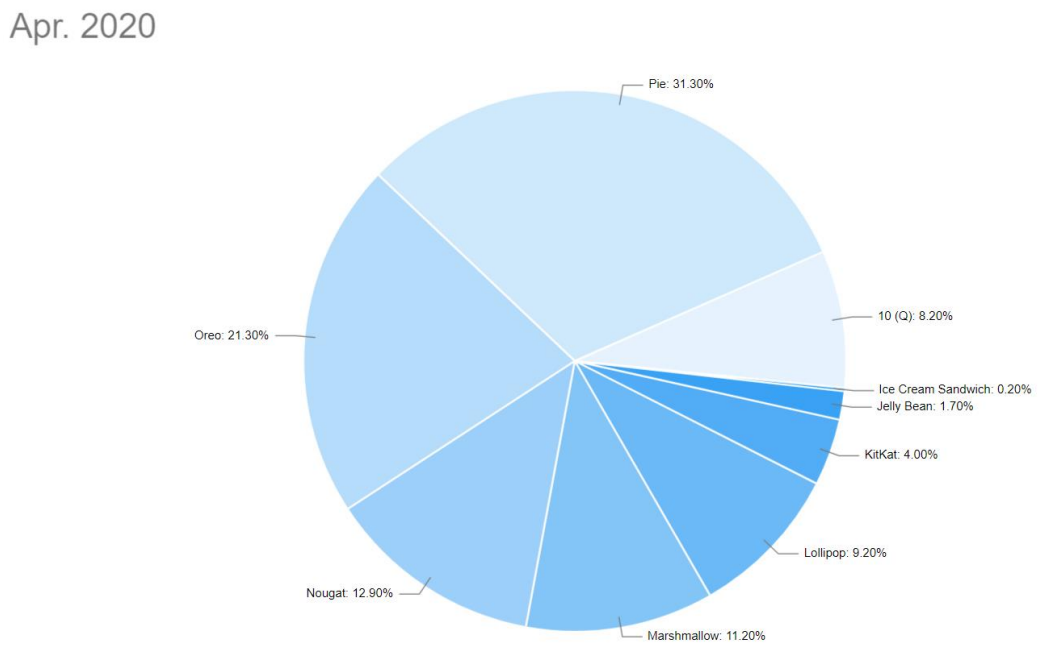
Wśród systemów mobilnych Android nie ma sobie równych, bazując na statystykach od *statcounter* w październiku 2020 roku 73% urządzeń mobilnych pracowało pod jego kontrolą. Przez ostatni rok stale utrzymywał najwyższą pozycję nie spadając poniżej 70% udziału w rynku systemów mobilnych. Na drugim miejscu jest system iOS pod kontrolą którego pracowało około 25% urządzeń mobilnych, a kolejne systemy takie jak np. KaiOS, Windows nie uzyskały nawet 0,25% udziału w tym rynku. Na rysunku 2.1 przedstawiono wykres popularności mobilnych systemów operacyjnych na świecie.



Rysunek 2.1 Popularność mobilnych systemów operacyjnych na świecie
Źródło: statcounter [2]

System Android na rynku jest od już 2008 roku i przez ten okres pojawiło się jego wiele wersji, które aktualnie wydawane są raz na rok. Dostępność platformy na wiele urządzeń od wielu producentów spowodowało, że wiele telefonów nadal pracuje na wersjach wydanych kilka lat temu. Projektując więc aplikacje należy przejrzeć najpopularniejsze wersje i zastanowić się jakie warto wspierać, gdyż pisanie programu pod bardzo stare wydania z niewielkim udziałem może

być nieopłacalne. Bazując na wykresie od *androiddistribution* w kwietniu 2020 roku najpopularniejszą wersją była *Pie* z udziałem 31,3%, wydana w 2018 roku, na drugim miejscu jest *Oreo*, a na trzecim *Nougat*. Najnowsze wydanie numer 10 jest dopiero na 6 miejscu jednak wraz z upływem czasu zapewne jego popularność będzie wzrastać. Najstarsze wersje ukazane na tym wykresie takie jak: *Ice Cream Sandwich, Jelly Bean* i *KitKat* mają razem około 6% udziału co jest już niewielką liczbą, która na dodatek cały czas się zmniejsza. Na rysunku 2.2 przedstawiono wykres kołowy na którym oznaczono popularność poszczególnych wersji systemu Android w kwietniu 2020 roku.



Rysunek 2.2 Popularność poszczególnych wersji systemu Android
Źródło: androiddistribution [4]

Aplikacje na system Android mogą być napisane w trzech językach: Kotlin, Java oraz C++, które są kompilowane przez Android SDK do plików w formacie *apk*, które z łatwością mogą być zainstalowane na urządzeniu mobilnym. Taka aplikacja może składać się z czterech głównych komponentów takich jak Aktywność (ang. *Activity*), Serwis (ang. *Service*), Odbiorcy transmisji (ang. *Broadcast receivers*), Dostawcy danych (ang. *Content providers*). Pierwszy z wymienionych komponentów jest punktem wejściowym do interakcji z użytkownikiem, reprezentuje on pojedynczy ekran aplikacji. Przykładem może być pojedynczy ekran logowania lub rejestracji wypełniony polami do wypełnienia przez użytkownika i przyciskami. Aktywności mogą ze sobą współpracować jednak każde z nich wykonuje własne niezależne działanie. Serwisy to komponenty, których zadaniem jest wykonywanie długotrwałych operacji nawet, gdy aplikacja działa w tle. Nie zapewniają one interfejsu dla użytkownika. Przykładowym ich użyciem może być odtwarzanie muzyki w tle czy pobieranie dużej ilości danych z sieci. Odbiorcy transmisji umożliwiają dostarczanie zdarzeń do aplikacji przez system, które mogą być wywoływane nawet jeśli nie jest ona aktywna. Dostawca danych to komponent pozwalający na zarządzanie danymi zgromadzonymi przez aplikację m.in. w systemie plików i bazie danych SQLite. Warto przy

tworzeniu aplikacji pamiętać o tym, że system Android realizuje zasadę najmniejszego uprzywilejowania (ang. *principle of least privilege*). Znaczy to, że każda aplikacja ma dostęp tylko do tych składników systemu których faktycznie potrzebuje i do których uzyskała pozwolenie od systemu lub użytkownika [3].

System Android dostarcza natywnych funkcji umożliwiających uzyskanie aktualnej pozycji urządzenia mobilnego. Pozwala nam na to zarówno klasa *FusedLocationProviderClient* jak i *LocationManager* wraz z *LocationListener*. Zgodnie z zasadą najmniejszego uprzywilejowania, aby chronić prywatność użytkownika dostęp do lokalizacji jest możliwy po uzyskaniu jego zgody. Wyróżniane są dwie kategorie dostępu do lokalizacji, czyli pierwszoplanową (ang. *foreground*) i drugoplanową (ang. *background*). Pierwsza kategoria pozwala na otrzymanie lokalizacji tylko raz lub przez ściśle określony okres czasu. Działanie to musi być bezpośrednio inicjowane przez użytkownika. Korzystamy tutaj z pozwoleń *ACCESS_COARSE_LOCATION* i *ACCESS_FINE_LOCATION*. Druga kategoria natomiast umożliwia ciągłe pobieranie lokalizacji urządzenia w tle. Tutaj należy skorzystać z pozwolenia *ACCESS_BACKGROUND_LOCATION* [5].

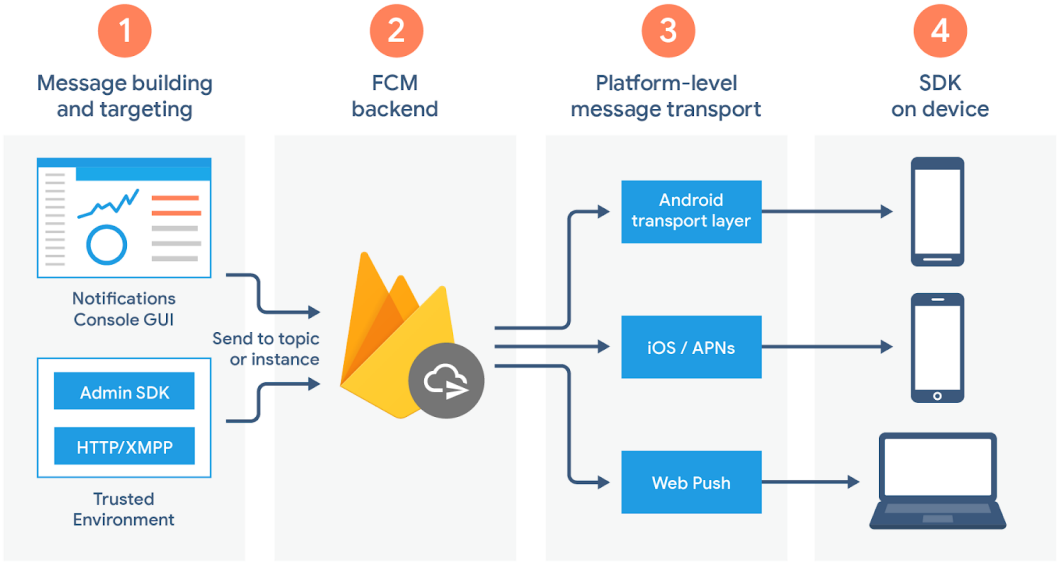
Android zapewnia możliwość tworzenia lokalnej instancji relacyjnej bazy danych SQLite. Utworzyć ją można poprzez bezpośrednie komendy wykorzystywane w SQLite i dzięki nim również możliwy jest dostęp do danych w tej bazie. Nie jest to jednak już rekomendowane rozwiązanie. Zalecane jest stosowanie biblioteki *Room*, która jest abstrakcyjną warstwą dla SQLite zapewniającą płynny i łatwy dostęp do bazy danych. Wyróżnić można tu trzy główne komponenty oznaczone przy pomocy adnotacji, czyli baza danych, encja i *dao* będące obiektem zawierającym metody zapewniające dostęp do bazy danych. Pierwszy z wymienionych komponentów natomiast jest abstrakcyjną klasą będącą podstawowym punktem dostępu do relacyjnie zgromadzonych danych. Encja jest klasą zawierającą pola występujące w danej tabeli w bazie danych i oferującą metody dostępowe do nich [6].

2.2. Firebase Cloud Messaging

Firebase Cloud Messaging to darmowe rozwiązanie do przesyłania wiadomości między platformami [8]. Wyróżnić można dwa typy komunikatów tak przesyłanych, czyli powiadomienie, które automatycznie po otrzymaniu przez urządzenie wyświetlane jest w pasku powiadomień. Zawiera ono pewną informację skierowana do klienta. Drugim typem jest wiadomość z danymi, która przetwarzana jest przez aplikację kliencką nawet gdy użytkownik z niej nie korzysta. Takie dane można zapisać do bazy danych, pliku, a także możliwe jest utworzenie z nich własnego powiadomienia. Wiadomość zapisana jest w postaci par klucz - wartość tak jak to występuje w formacie *JSON* [9].

Architekturę tego rozwiązania można podzielić na cztery komponenty. Pierwszy jest narzędziem, w którym tworzona jest wiadomość. Może być to własna aplikacja lub oprogramowanie dostarczane przez *Firebase*. Drugi komponent zajmuje się przyjmowaniem żądań wysłania wiadomości, nadaje im odpowiednie identyfikatory i generuje metadane. Kolejny komponent kieruje wiadomości do docelowego urządzenia. Nie musi być to tylko urządzenie

mobilne z systemem Android. Możliwe jest wysyłanie komunikatów również na system iOS oraz do aplikacji internetowych. Ostatni komponent odpowiedzialny jest za odbiór i przetwarzanie wiadomości [10]. Na rysunku 2.3 przedstawiono architekturę Firebase Cloud Messaging.



Rysunek 2.3 Architektura Firebase Cloud Messaging
Źródło: firebase.google.com [10]

2.3. REST API

REST API jest stylem w architekturze oprogramowania wykorzystywanym do komunikacji opartej o protokół *HTTP* między aplikacjami. Protokół *HTTP* wykorzystuje łącznie 9 metod do komunikacji takich jak *GET, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH, HEAD*. Jednak, aby zbudować podstawowy sposób wymiany danych wystarczą 4 pierwsze metody. GET służy do pobierania danych, POST wykorzystywany jest do tworzenia, przesyłania nowych danych, PUT do aktualizowania zawartych informacji, a DELETE służy do usuwania danych. Taka komunikacja odbywa najczęściej się przy pomocy formatu JSON jednak nie jest to regułą i wykorzystywany jest jeszcze format *XML*. Wymiana danych przy pomocy REST API musi spełniać kilka założeń wśród których najważniejsze to bezstanowość, czyli każde zapytanie musi posiadać komplet informacji koniecznych do jego zrealizowania. Serwer nie przechowuje stanu sesji. Takie pojęcie w tym przypadku nawet nie istnieje. Drugim założeniem jest odseparowanie interfejsu użytkownika od operacji na serwerze, czyli aplikacja kliencka nie może mieć wpływu na to co się dzieje na serwerze i odwrotnie. Aplikacje te muszą działać niezależnie. Kolejnym założeniem jest separacja warstw, czyli należy podzielić aplikację na warstwy takie jak dostępu do danych, logiki biznesowa i prezentacji [11].

2.4. Spring Boot

Spring Boot to biblioteka (*framewok*) oparta na języku programowania Java umożliwiający tworzenia aplikacji opartych o mikroserwisy, które mogą działać niezależnie od

siebie jako osobne procesy. Framework ten oferuje szybkie i szeroko dostępne środowisko oraz niewielką ilość konfiguracji. Znajdzie zastosowanie w wielu aplikacjach m.in. wymagających połączeń z baza danych, a także w programach serwerowych oferujących komunikację w standardzie REST. Aplikacje napisane przy pomocy Spring Boot można z łatwością uruchamiać z konsoli przy pomocy polecenia *java -jar* [12].

2.5. Java Persistence API

Java Persistence API to standard, który dostarcza dla deweloperów rozwijających swoje aplikacje w języku programowania Java łatwy dostęp do relacyjnej bazy danych. Jego zadaniem jest mapowanie modelu relacyjnego do obiektowego i odwrotnie. Automatyzuje znaczną część działań na bazie danych. Najczęściej stosowanym dostawca tego standardu jest *Hibernate*. Wyróżniane są tu klasy takie jak encje będące obiektowym odwzorowaniem tabel i zapewniające podstawowy dostęp do pól oraz interfejsy dao oferujące metody wykonujące komendy w bazie danych zazwyczaj definiowane przy pomocy adnotacji *Query* [13].

2.6. Podsumowanie

Android jest obecnie najpopularniejszym mobilnym systemem operacyjnym na świecie, a jego konkurencja ma znacznie mniejszy udział w rynku. Pisanie aplikacji na ten system jest mocno zautomatyzowane i pobieranie lokalizacji urządzenia czy dostęp do bazy danych odbywa się przy pomocy gotowych funkcji i bibliotek. Komunikacja między aplikacją mobilną, a serwerową może odbywać się przy pomocy Firebase Cloud Messaging, jeśli chcemy szybko poinformować użytkownika o zdarzeniach i REST API, jeśli potrzebne jest pobranie większej ilości danych lub wysłanie nowych informacji na serwer. Aplikację wspierającą takie formy komunikacji można napisać w języku Java przy wykorzystaniu frameworku Spring Boot, a dostęp do bazy danych może być w niej realizowana za pomocą standardu Java Persistence *API*.

3. ANALIZA PROJEKTOWA

W trzecim rozdziale przedstawiono składniki zrealizowane w pierwszej fazie tworzenia projektu inżynierskiego, czyli analizy projektowej. Wykonano w ramach tego rozdziału dokument specyfikacji wymagań, gdzie określono wymagania funkcjonalne i pozafunkcjonalne, które przedstawiono w tej pracy. Ponadto wykonano i zaprezentowano tu schemat przypadków użycia wraz z jego opisem.

3.1. Wymagania funkcjonalne

Wymagania funkcjonalne przedstawiają zarys funkcji jakie powinien posiadać realizowany projekt. Podzielono je według priorytetu.

Priorytet wysoki:

- rejestracja nowego konta w systemie,
- logowanie się do aplikacji na utworzone wcześniej konto,
- możliwość wylogowywania się z systemu,
- dodanie nowego kontaktu w liście zaufanych użytkowników,
- akceptacja zaproszenia na dodanie kontaktu do znajomych,
- wysłanie powiadomienia do listy kontaktów zawierającą lokalizację wraz z prośbą o pomoc,
- otrzymanie powiadomienia z lokalizacją i prośbą o pomoc,
- udostępnienie lokalizacji swoim znajomym za pomocą przycisku,
- możliwość wizualizacji przesłanej lokalizacji znajomego na mapie.

Priorytet średni:

- usuwanie kontaktu z listy znajomych oraz danych o kontakcie,
- możliwość wysyłania wiadomości do znajomych,
- możliwość otrzymania wiadomości od innych użytkowników,
- dostęp do części funkcji aplikacji przy braku połączenia z siecią. W tym m.in. przegląd listy kontaktów, przesyłanych wiadomości i tras,
- odrzucenie zaproszenia na dodawanie kontaktu do znajomych.

Priorytet niski:

- edycja danych użytkownika.

3.2. Wymagania pozafunkcjonalne

Wymagania pozafunkcjonalne dotyczą jakości tworzonego projektu i podzielono je na wymagania na dane, jakościowe, użyteczności oraz w zakresie wydajności i elastyczności.

Wymagania na dane:

- dostęp do API które oferuje dostęp do informacji na temat różnych sposobów dojazdu do określonej lokalizacji;
- dane o użytkownikach w liście kontaktów (imię, nazwisko, adres email);

- dane o przesyłanych wiadomościach między użytkownikami (treść, data, adresat, odbiorca);
- dane o udostępnionych lokalizacjach (udostępniający, długość i szerokość geograficzna, data).

Wymagania jakościowe:

- przetwarzania danych osobowych użytkownika zgodne z regulacjami o ochronie danych osobowych (*RODO*);
- szyfrowanie wrażliwych danych, użycie bezpiecznych algorytmów uwierzytelniania.

Wymagania w zakresie wydajności:

- aplikacja powinna być w stanie obsłużyć 100 użytkowników jednocześnie przy założeniu, że średnia liczba zapytań od użytkownika wynosi 1-2 na 10 sekund;
- odpowiedź na zapytanie użytkownika w czasie nie dłuższym niż 5 s.

Wymagania w zakresie elastyczności:

- dostępność opcji uruchamiania aplikacji na urządzeniach z ekranami o różnych wymiarach;
- możliwość uruchomienia aplikacji na systemie operacyjnym Android.

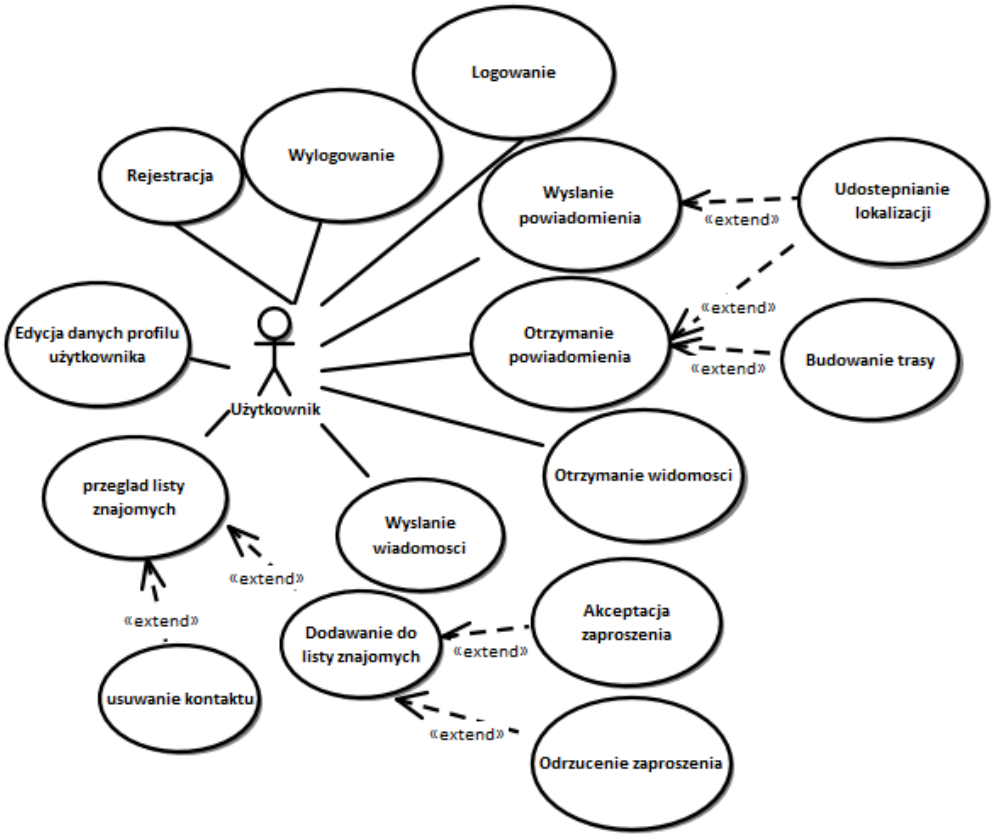
Wymagania użyteczności:

- brak jaskrawych kolorów męczących wzrok użytkownika, nazwy krótkie, ale jednoznacznie określające funkcje. Dobrze widoczne miejsce, w którym znajduje się użytkownik;
- aplikacja poprawnie uruchamia się w systemie Android w wersji 5.0 lub nowszej.

3.3. Przypadki użycia

Przypadki użycia przedstawiają możliwości interakcji użytkowników z systemem. Pojedynczy przypadek użycia obejmuje scenariusz sposobu korzystania systemu, zdarzenia inicjujące, nietypowe zdarzenia, sposoby ich rozwiązania i warunek końcowy. Na rysunku 3.1 przedstawiono schemat przypadków użycia dla projektu inżynierskiego.

W tabelach od 3.1 do 3.15 przedstawiono opisy przypadków użycia przedstawionych na powyższym schemacie. Zawarto w nich streszczenie przypadku, zdarzenie inicjujące, opis, sytuacje wyjątkowe oraz warunki początkowe i końcowe.



Rysunek 3.1 Schemat przypadków użycia

Tabela 3.1 Przypadek użycia *Rejestracja*

Streszczenie	Funkcja aplikacji pozwalająca na rejestrację w systemie.
Zdarzenie inicjujące	Aktor wybiera przycisk rejestracji.
Warunki początkowe	Aktor ma status użytkownika.
Pełny opis	1. Aktor wybiera przycisk rejestracji w uruchomionej aplikacji, 2. Aktor wypełnienia wyświetlone pola zgodnie z zaleceniami (login, imię, nazwisko, e-mail, ustawienie hasła (hasło powinno zawierać 8 znaków, w tym co najmniej jedna duża litera, jeden znak specjalny i liczba)), 3. Aktor wybiera przycisk rejestracji.
Sytuacje wyjątkowe	Błąd łączenia się z bazą danych: ponowna próba. Źle wprowadzone hasło, brak któregoś z wymagań: korekta zgodnie z wymaganiem. Źle wprowadzony adres mailowy: korekta nazwy adresu. Puste pole: uzupełnienie wymaganymi danymi.
Warunki końcowe	Rejestracja jest zakończona, dane są zapisane w bazie danych, aktor zostaje przekierowany na stronę logowania.

Tabela 3.2 Przypadek użycia *Logowanie*

Streszczenie	Funkcja aplikacji pozwalająca na logowanie się w systemie.
Zdarzenie inicjujące	Aktor wybiera przycisk logowania.
Warunki początkowe	Aktor ma status zarejestrowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk logowania w uruchomionej aplikacji,

Pełny opis	2. Aktor widzi pola do wypełnienia (login, hasło), 3. Po wypełnieniu aktor wybiera przycisk logowania się, 4. Przejście do następnego widoku.
Sytuacje wyjątkowe	Błąd łączenia się z bazą danych: ponowna próba. Błędnie wprowadzony login: wprowadzenie poprawnego loginu. Błędnie wprowadzone hasło: wprowadzenie poprawnego hasła. Puste pole login lub hasło: wprowadzenie danych.
Warunki końcowe	Logowanie powiodło się, użytkownik przechodzi na następny widok.

Tabela 3.3 Przypadek użycia *Wylogowanie*

Streszczenie	Funkcja aplikacji pozwalająca na wylogowanie się z systemu.
Zdarzenie inicjujące	Aktor wybiera przycisk służący do wylogowania.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk służący do wylogowania się w uruchomionej aplikacji, 2. Aktor zostaje przekierowany na widok początkowy aplikacji.
Sytuacje wyjątkowe	Brak połączenia z serwerem: ponowna próba wylogowania się.
Warunki końcowe	Wylogowanie powiodło się użytkownik przechodzi na widok początkowy aplikacji.

Tabela 3.4 Przypadek użycia *Przegląd listy znajomych*

Streszczenie	Funkcja aplikacji wyświetlająca listy znajomych użytkownika.
Zdarzenie inicjujące	Aktor wybiera przycisk, który wyświetla listę znajomych.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk, który wyświetla listę znajomych, 2. Zostaje wyświetlona lista znajomych, 3. Dla wybranego znajomego wyświetlane są dane szczegółowe.
Sytuacje wyjątkowe	Zawieszenie systemu: ponowna próba wyświetlania listy.
Warunki końcowe	Wyświetlona lista wszystkich znajomych wraz z możliwością dodawania nowych, edycji i usuwania.

Tabela 3.5 Przypadek użycia *Usuwanie Kontaktu*

Streszczenie	Funkcja aplikacji, która pozwala usuwać kontakty z listy.
Zdarzenie inicjujące	Aktor wybiera kontakt, który chciałby usunąć.
Warunki początkowe	1. Aktor ma status zalogowanego użytkownika, 2. Aktor wybrał kontakt z listy (PU przegląd listy kontaktów), 3. Aktor wybrał funkcję usunięcia kontaktu z listy znajomych.
Pełny opis	1. Na ekranie wyświetlona jest lista kontaktów, 2. Aktor wybiera kontakt, 3. Dla wybranego kontaktu jest możliwość usunięcia. Aktor wybiera usunięcie, 4. Aplikacja prosi o potwierdzenie danej akcji. Aktor wybiera przycisk, zatwierdzający akcję, 5. Kontakt został usunięty z listy.
Sytuacje wyjątkowe	Brak kontaktów: nie zostaje wyświetlona lista kontaktów. Zawieszenie systemu: ponowna próba wywołania funkcji.

Warunki końcowe	Kontakt zostanie usunięty z listy znajomych w aplikacji.

Tabela 3.6 Przypadek użycia *Edycja danych profilu użytkownika*

Streszczenie	Funkcja aplikacji, która pozwala na edycję danych profilu użytkownika.
Zdarzenie inicjujące	Aktor wybiera przycisk, który przekieruje go na ekran z danymi o użytkowniku.
Warunki początkowe	1. Aktor ma status zalogowanego użytkownika, 2. Aktor wybrał przycisk na ekranie przekierowujący do ekranu z edycją profilu.
Pełny opis	1. Na ekranie wyświetlone są pola z informacjami na temat aktualnie zalogowanego użytkownika, 2. Aktor wybiera pole, które chce edytować, 3. Aktor zmienia dane, 4. Aplikacja prosi o potwierdzenie danej akcji. Aktor wybiera przycisk potwierdzający działanie, 5. Wyświetlane dane zostały zmienione.
Sytuacje wyjątkowe	Aktor nie zatwierdził zmiany: zmiany nie zostały wprowadzone, rozpocząć ponowną próbę edycji. Zawieszenie systemu: ponowna próba wywołania funkcji.
Warunki końcowe	Informacje o użytkowniku zostały zmienione.

Tabela 3.7 Przypadek użycia *Dodawanie do listy znajomych*

Streszczenie	Funkcja aplikacji, która pozwala dodawać znajomych do listy kontaktów.
Zdarzenie inicjujące	Aktor wybiera przycisk umożliwiający dodanie znajomych na swoją listę kontaktów.
Warunki początkowe	1. Aktor ma status zalogowanego użytkownika, 2. Aktor wybrał opcję wyświetlenia listy kontaktów.
Pełny opis	1. Na ekranie wyświetlona jest lista kontaktów. W przypadku, gdy ich nie ma na ekranie jest tylko przycisk dodawania znajomych, 2. Aktor wybiera przycisk dodawania znajomych, 3. W polu login aktor wpisuje login osoby, do której będzie wysyłać zaproszenie, 4. Po wprowadzeniu loginu aktor wybiera przycisk pozwalający na wysłanie zaproszenia, 5. Pojawia się komunikat informujący o tym, że zaproszenie zostało wysłane.
Sytuacje wyjątkowe	Nieprawidłowo podany login: należy wpisać prawidłowy login. Puste pole loginu: wypełnić potrzebne dane w odpowiednim polu.
Warunki końcowe	Zaproszenie zostanie wysłane i po akceptacji kontakt pojawi się na liście.

Tabela 3.8 Przypadek użycia *Akceptacja zaproszenia*

Streszczenie	Funkcja aplikacji, która pozwala akceptować zaproszenia na dodanie do listy znajomych.
Zdarzenie inicjujące	Aktor wybiera przycisk umożliwiający przegląd listy zaproszeń.
Warunki początkowe	1. Aktor ma status zalogowanego użytkownika, 2. Aktor wybrał zakładkę z listą zaproszeń.
Pełny opis	1. Na ekranie wyświetlona jest lista zaproszeń do akceptacji,

Pełny opis	2. Aktor wybiera zaproszenie od danego użytkownika, 3. Aktor wybiera przycisk, który pozwala zaakceptować zaproszenie, 4. Zaakceptowany znajomy jest przeniesiony do listy znajomych.
Sytuacje wyjątkowe	Brak połączenia z serwerem: ponowna próba połączenia się.
Warunki końcowe	Znajomy zostaje dodany na listę kontaktów.

Tabela 3.9 Przypadek użycia *Odrzucenie zaproszenia*

Streszczenie	Funkcja aplikacji, która pozwala odrzucić zaproszenia na dodanie do listy znajomych.
Zdarzenie inicjujące	Aktor wybiera przycisk umożliwiający przegląd listy zaproszeń.
Warunki początkowe	1. Aktor ma status zalogowanego użytkownika, 2. Aktor wybrał zakładkę z listą zaproszeń.
Pełny opis	1. Na ekranie jest wyświetlana lista zaproszeń do akceptacji, 2. Aktor wybiera zaproszenie od danego użytkownika, 3. Aktor wybiera przycisk, który pozwala odrzucić zaproszenie, 4. Odrzucone zaproszenie znika z listy.
Sytuacje wyjątkowe	Odrzucenie nie powiodło się i zaproszenie nadal jest widoczne: ponowna próba.
Warunki końcowe	Zaproszenie zostaje odrzucone i znajomy nie pojawia się na liście kontaktów.

Tabela 3.10 Przypadek użycia *Wysłanie wiadomości*

Streszczenie	Funkcja aplikacji pozwalająca na wysyłanie wiadomości swoim znajomym z listy kontaktów.
Zdarzenie inicjujące	Aktor wybiera przycisk, pozwalający na wysyłanie wiadomości.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk, który otworzy widok z listą znajomych, 2. Aktor wybiera kontakt, do którego chce wysłać wiadomość, 3. Aktor widzi pole do wpisywania wiadomości, 4. Aktor wpisuje swoją wiadomość w polu, 4. Aktor wybiera przycisk, który wysyła wiadomość wybranemu znajomemu.
Sytuacje wyjątkowe	Wiadomość nie została wysłana: sprawdzenie łączności z siecią i ponowne przejście do punktu 4.
Warunki końcowe	Wysłanie wiadomości powiodło się. Wiadomość zostanie wysłana znajomym z listy.

Tabela 3.11 Przypadek użycia *Otrzymanie wiadomości*

Streszczenie	Funkcja aplikacji pozwalająca na otrzymanie wiadomości od swoich znajomych.
Zdarzenie inicjujące	Aktor otrzymuje powiadomienie o nowej wiadomości.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk, który przekieruje go na ekran z listą wiadomości od znajomych, 2. Aktor wybiera wiadomość, którą chce otworzyć, 3. Otwiera się widok, który wyświetla treść wiadomości.
Warunki końcowe	Wiadomość została otrzymana i przeczytana przez użytkownika.

Tabela 3.12 Przypadek użycia *Wysłanie powiadomienia*

Streszczenie	Funkcja aplikacji pozwalająca na wysyłanie powiadomienia swoim znajomym.
Zdarzenie inicjujące	Aktor wybiera przycisk, pozwalający wysyłanie powiadomienia z prośbą o pomoc.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk, który wyśle wiadomość z prośbą o pomoc w uruchomionej aplikacji.
Sytuacje wyjątkowe	Błąd łączenia się z siecią: znalezienie sieci. Nie nadano uprawnień na udostępnianie lokalizacji: nadać aplikacji zezwolenie na udostępnianie lokalizacji i ponownie przejść do punktu 1.
Warunki końcowe	Wysłanie powiadomienia powiodło się. Powiadomienie zostanie wyświetlone na urządzeniu znajomych.

Tabela 3.13 Przypadek użycia *Otrzymanie powiadomienia*

Streszczenie	Funkcja aplikacji pozwalająca na otrzymanie powiadomienia od swojego znajomego.
Zdarzenie inicjujące	Aktor otrzymuje powiadomienie z prośbą o pomoc.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor przyciska na otrzymane powiadomienie, 2. Aktor został przekierowany na ekran z treścią powiadomienia od kontaktu z listy.
Warunki końcowe	Powiadomienie zostało otrzymane.

Tabela 3.14 Przypadek użycia *Udostępnianie lokalizacji*

Streszczenie	Funkcja aplikacji pozwalająca na udostępnianie swojej lokalizacji znajomym z listy.
Zdarzenie inicjujące	Aktor wybiera przycisk, pozwalający wysłać prośbę o pomoc wraz z danymi o swojej lokalizacji.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk udostępnienia lokalizacji w uruchomionej aplikacji, 2. Użytkownik informowany jest, że trwa udostępnianie lokalizacji.
Sytuacje wyjątkowe	Błąd połączenia z siecią: próba kolejnego połączenia się i przejście do punktu 1. Nie nadano uprawnień aplikacji do śledzenia pozycji: nadać zezwolenie w ustawieniach urządzenia dla tej aplikacji i przejść do punktu 1.
Warunki końcowe	Znajomi zostali poinformowani o lokalizacji użytkownika.

Tabela 3.15 Przypadek użycia *Budowanie trasy*

Streszczenie	Funkcja aplikacji pozwalająca na budowanie trasy dojazdu do znajomego, potrzebującego pomocy.
Zdarzenie inicjujące	Aktor otrzymuje powiadomienie z prośbą o pomoc.
Warunki początkowe	Aktor ma status zalogowanego użytkownika.
Pełny opis	1. Aktor wybiera przycisk przekierowujący go do okna udostępnianych mu tras, 2. Aktor wybiera trasę znajomego, 3. Aktor zostaje przekierowany do widoku z mapą,

Pełny opis	4. Aktor wybiera swoje położenie i położenie osoby, potrzebującej pomocy, 5. Aktor wybiera optymalną trasę dojazdu z przedstawionych.
Sytuacje wyjątkowe	Błąd łączenia się z siecią i brak możliwości budowania trasy: znalezienie sieci. Nie nadano uprawnień na udostępnianie lokalizacji: nadać aplikacji zezwolenie na udostępnianie lokalizacji i przejście do punktu 5.
Warunki końcowe	Aktorowi wyświetlana jest trasa do znajomego.

3.4. Podsumowanie

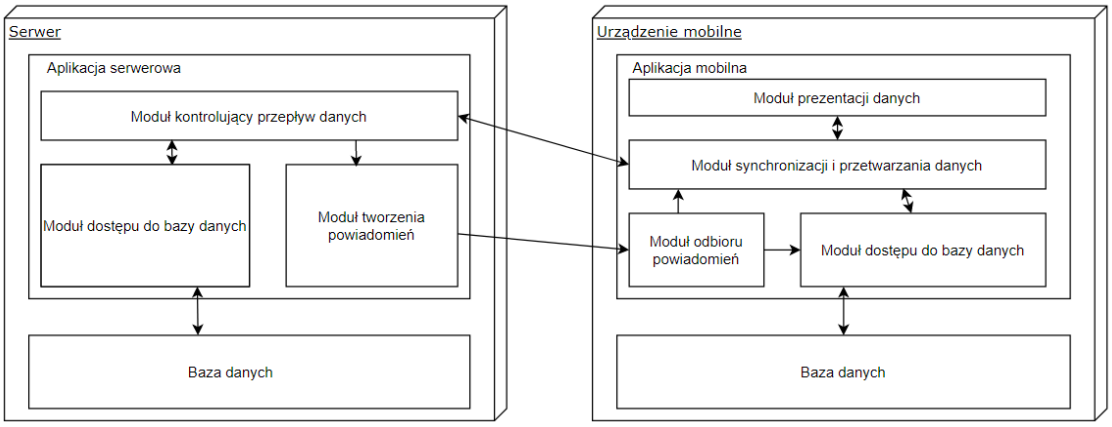
Analiza projektowa jest istotnym elementem procesu tworzenia aplikacji. W ramach niej określono wymagania funkcjonalne, które posortowano według priorytetu na wysoki, średni i niski. Zdecydowano, że najważniejszymi wymaganiami są m.in. możliwość rejestracji i logowania użytkownika oraz dodawanie znajomych i wysyłanie im danych na temat lokalizacji. Wymagania pozafunkcjonalne zostały podzielone na wymagania jakościowe, użyteczności, w zakresie elastyczności i wydajności oraz na dane. Następnie przygotowano schemat przypadków użycia opierający się na wcześniej przygotowanych wymaganiach.

4. PROJEKT

W ramach czwartego rozdziału przygotowano projekt planowanego systemu. Określono tu jaką architekturę powinien on posiadać i z jakich modułów składać będą się odpowiednie aplikacje. Następnie przedstawiono projekty baz danych przechowujących informacje na temat zarejestrowanych użytkowników. Na końcu zaprezentowano projekt klas aplikacji mobilnej oraz serwerowej wraz z krótkim opisem ich funkcji.

4.1. Architektura systemu

Projekt inżynierski został oparty o architekturę klient-serwer. Na urządzeniu klienta znajduje się aplikacja mobilna oraz lokalna instancja bazy danych przechowująca dane na temat użytkowników zalogowanych na tym urządzeniu. Na serwerze znajduje się aplikacja serwerowa oraz centralna baza danych przechowująca informacje o wszystkich użytkownikach mających konto w aplikacji. Na rysunku 4.1 przedstawiono schemat architektury projektu.



Rysunek 4.*1* Architektura systemu

Aplikację mobilną oraz serwerową można podzielić na moduły realizujące osobne działania i komunikujące się ze sobą. W pierwszej wymienionej aplikacji możemy wyróżnić cztery podstawowe moduły, czyli prezentacji danych, synchronizacji i przetwarzania danych, odbioru powiadomień i dostępu do bazy danych. Moduł prezentacji danych odpowiedzialny jest za utworzenie interfejsu użytkownika, dane do wyświetlenia pobiera poprzez komunikację z modułem synchronizacji i przetwarzania danych, który jest odpowiedzialny za większość logiki biznesowej aplikacji i synchronizację danych zawartych w centralnej i lokalnej bazie danych. Moduł odbioru powiadomień analizuje przesłane komunikaty do użytkownika i wyświetla je w postaci powiadomienia oraz może je zapisać do bazy danych. Ostatni z modułów oferuje operacje bazodanowe realizowane na lokalnej instancji. W aplikacji serwerowej można wyróżnić trzy moduły, czyli kontrolujący przepływ danych, tworzenia powiadomień i dostępu do bazy danych. Pierwszy z nich odbiera, przetwarza i przesyła dane użytkowników poprzez protokół HTTP. Realizowana jest tu główna logika biznesowa. Jeśli otrzymuje informacje, które muszą być szybko przesyłane klientom aplikacji to przekazuje je do modułu tworzenia powiadomień odsyłający je użytkownikom w formie notyfikacji. Ostatni z komponentów odpowiedzialny jest za operacje bazodanowe na centralnej instancji bazy danych.

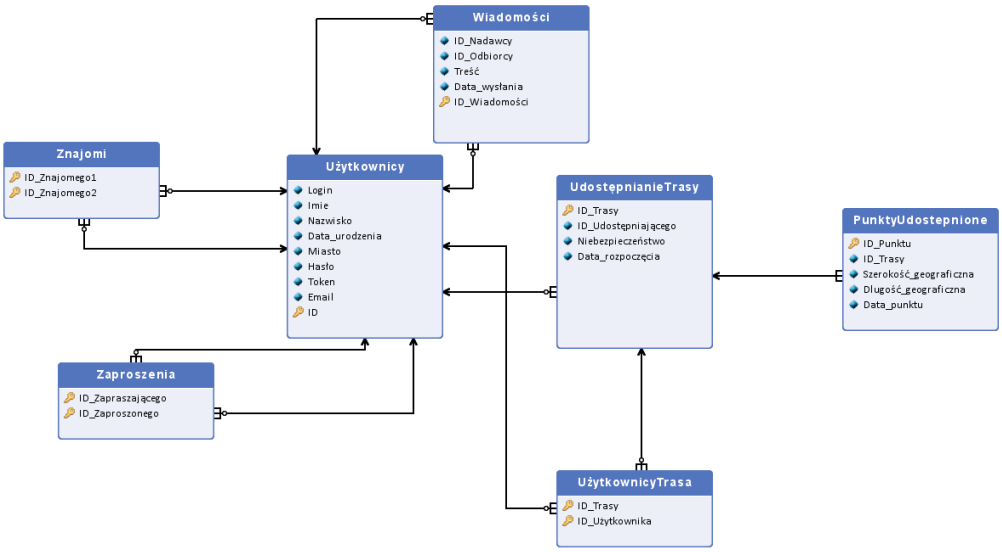
Komunikacja między aplikacją serwerową, a kliencką odbywa się dwustronnie pomiędzy modułem kontrolującym przepływ danych, a synchronizacji i przetwarzania danych przy pomocy *REST API* oraz jednostronna od modułu tworzenia powiadomień do odbioru powiadomień poprzez Firebase Cloud Messaging.

4.2. Baza danych

W tym rozdziale zostały przedstawione schematy i opisy baz danych znajdujących się w środowisku serwerowym oraz na urządzeniu mobilnym.

4.2.1. Baza danych w środowisku serwerowym

Baza danych w środowisku serwerowym przechowuje dane na temat wszystkich użytkowników systemu i ich wiadomościach oraz udostępnionych trasach. Każda z encji identyfikowana jest na podstawie sztucznego klucza głównego. Na rysunku 4.2 przedstawiono schemat bazy danych w środowisku serwerowym.



Rysunek 4.*2* Schemat bazy danych w środowisku serwerowym

Przedstawiona powyżej baza danych składa się z czterech głównych encji, czyli *Uzytkownicy, Wiadomości, UdostępnioneTrasy*, *PunktyUdostepnione* oraz trzech będących realizacją relacji wiele do wielu, czyli *Znajomi, Zaproszenia* i *UzytkownicyTrasa*.

Encja *Użytkownicy* zawiera podstawowe informacje na temat osób mających konta w aplikacji. Składa się z pól:

- *id*, typ numeryczny, klucz główny,
- *login*, typ tekstowy,
- *imię*, typ tekstowy,
- *nazwisko*, typ tekstowy,
- *email*, typ tekstowy,
- *data urodzenia*, data,

- *miasto*, typ tekstowy,
- *hasło*, typ tekstowy,
- *token*, typ tekstowy.

Encja *Wiadomości* zawiera informacje na temat przesyłanych komunikatów między użytkownikami. Składa się z pól:

- *id wiadomości*, typ numeryczny, klucz główny,
- *id nadawcy*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*,
- *id odbiorcy*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*,
- *treść*, typ tekstowy,
- *data wysłania*, data.

Encja *UdostępnioneTrasy* zawiera informacje na temat udostępniania lokalizacji przez użytkownika. Składa się z pól:

- *id trasy*, typ numeryczny, klucz główny,
- *id udostępniającego*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*,
- *niebezpieczeństwo*, typ logiczny,
- *data rozpoczęcia*, data.

Encja *PunktyUdostepnione* zawiera informacje na temat pojedynczego punktu o danej szerokości i długości geograficznego udostępnionego w ramach trasy innym znajomym. Składa się z pól:

- *id punktu*, typ numeryczny, klucz główny,
- *id trasy*, typ numeryczny, klucz obcy do tabeli *UdostępnioneTrasy*,
- *szerokość geograficzna*, typ zmiennoprzecinkowy,
- *długość geograficzna*, typ zmiennoprzecinkowy,
- *data punktu*, data.

Encja *Znajomi* realizuje relację wiele do wielu między tabelą *Użytkownicy* i wszystkie atrybuty tworzą klucz złożony. Informuje jakie osoby są ze sobą znajomymi. Składa się z pól:

- *id znajomego1*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*,
- *id znajomego2*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*.

Encja *Zaproszenia* realizuje relację wiele do wielu między tabelą *Użytkownicy* i wszystkie atrybuty tworzą klucz złożony. Informuje jaka osoba i do kogo wysłała zaproszenie do bycia znajomymi w aplikacji. Składa się z pól:

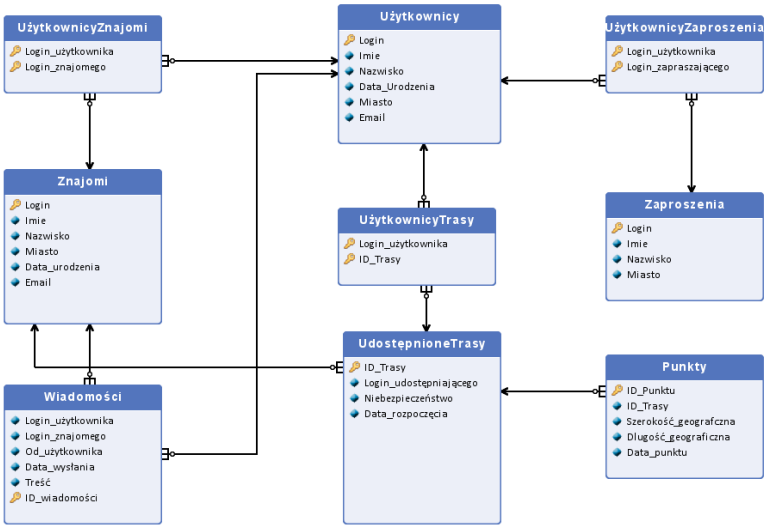
- *id zapraszającego*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*,
- *id zaproszonego*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*.

Encja *UżytkownicyTrasa* realizuje relację wiele do wielu między tabelą *UdostępnioneTrasy* , a *Użytkownicy* i wszystkie atrybuty tworzą klucz złożony. Informuje komu została udostępniona dana trasa. Składa się z pól:

- *id trasy*, typ numeryczny, klucz obcy do tabeli *UdostępnioneTrasy*,
- *id użytkownika*, typ numeryczny, klucz obcy do tabeli *Użytkownicy*.

4.2.2. Baza danych na urządzeniu klienta

Baza danych na urządzeniu klienta zawiera informacje tylko na temat użytkowników zalogowanych na danym urządzeniu mobilnym. Przechowywane dodatkowo są tu informacje o znajomych, udostępnionych trasach oraz wiadomościach. Składa się z sześciu głównych encji, czyli *Użytkownicy, Znajomi, Zaproszenia, Wiadomości, UdostępnioneTrasy* i *Punkty* oraz trzech będących realizacją relacji wiele do wielu, czyli *UżytkownicyZnajomi, UżytkownicyTrasa, UżytkownicyZaproszenia*. Wielkość tej bazy danych będzie znacznie mniejsza niż na serwerze oraz schemat również się trochę różni, dlatego encje identyfikowane głównie będą przy pomocy loginu użytkownika. Jedynie encje *Wiadomości, UdostępnioneTrasy* i *Punkty* identyfikowane są przy pomocy klucza sztucznego zsynchronizowanego z tym w bazie danych na serwerze. Na rysunku 4.3 przedstawiono schemat bazy danych na urządzeniu mobilnym.



Rysunek 4.3 Schemat bazy danych umieszczonej na urządzeniu klienta

Encja *Użytkownicy* zawiera podstawowe informacje na temat osób, które były zalogowane na danym urządzeniu mobilnym. Możliwe jest logowanie wielu osób do aplikacji na jednym urządzeniu. Składa się z pól:

- *login*, typ tekstowy, klucz główny,
- *imię*, typ tekstowy,
- *nazwisko*, typ tekstowy,
- *email*, typ tekstowy,
- *data urodzenia*, data,
- *miasto*, typ tekstowy.

Encja *Znajomi* zawiera podstawowe informacje na temat znajomych użytkowników, którzy byli zalogowani na danym urządzeniu mobilnym Składa się z pól:

- *login*, typ tekstowy, klucz główny,
- *imię*, typ tekstowy,
- *nazwisko*, typ tekstowy,

- *email*, typ tekstowy,
- *data urodzenia*, data,
- *miasto*, typ tekstowy.

Encja *Zaproszenia* zawiera podstawowe informacje na temat osoby wysyłającej zaproszenie dla użytkownika do bycia znajomym. Składa się z pól:

- *login*, typ tekstowy, klucz główny,
- *imię*, typ tekstowy,
- *nazwisko*, typ tekstowy,
- *miasto*, typ tekstowy.

Encja *Wiadomości* zawiera informacje na temat przesyłanych komunikatów zarówno utworzonych przez użytkownika jak i wysłanych przez niego do znajomych. Składa się z pól:

- *id wiadomości*, typ numeryczny, klucz główny,
- *login użytkownika*, typ tekstowy, klucz obcy do tabeli *Użytkownicy*,
- *login znajomego*, typ tekstowy, klucz obcy do tabeli *Znajomi*,
- *od użytkownika*, typ logiczny, zawiera informację czy nadawcą wiadomości był zalogowany użytkownik,
- *treść*, typ tekstowy,
- *data wysłania*, data.

Encja *UdostepnioneTrasy* zawiera informacje na temat udostępnionych tras przez użytkownika jak i jego znajomych. Składa się z pól:

- *id trasy*, typ numeryczny, klucz główny,
- *login udostępniającego*, typ tekstowy, klucz obcy do tabeli *Znajomi*,
- *niebezpieczeństwo*, typ logiczny,
- *data rozpoczęcia*, data.

Encja *Punkty* zawiera informacje na temat pojedynczego punktu o danej szerokości i długości geograficznego udostępnionego w ramach danej trasy. Składa się z pól:

- *id punktu*, typ numeryczny, klucz główny,
- *id trasy*, typ numeryczny, klucz obcy do tabeli *UdostępnioneTrasy*,
- *szerokość geograficzna*, typ zmiennoprzecinkowy,
- *długość geograficzna*, typ zmiennoprzecinkowy,
- *data punktu*, data.

Encja *UżytkownicyZnajomi* realizuje relację wiele do wielu między tabelą *Użytkownicy*, a *Znajomi* i wszystkie atrybuty tworzą klucz złożony. Informuje jakich znajomych ma dany użytkownik. Składa się z pól:

- *login użytkownika*, typ tekstowy, klucz obcy do tabeli *Użytkownicy*,
- *login znajomego*, typ tekstowy, klucz obcy do tabeli *Znajomi*.

Encja *UżytkownicyTrasy* realizuje relację wiele do wielu między tabelą *Użytkownicy*, a *UdostępnioneTrasy* i wszystkie atrybuty tworzą klucz złożony. Informuje do jakich tras ma dostęp dany użytkownik wcześniej zalogowany na urządzeniu. Składa się z pól:

- *login użytkownika*, typ tekstowy, klucz obcy do tabeli *Użytkownicy*,
- *id trasy*, typ numeryczny, klucz obcy do tabeli *UdostępnioneTrasy*.

Encja *UżytkownicyZaproszenia* realizuje relację wiele do wielu między tabelą *Użytkownicy*, a *Zaproszenia* i wszystkie atrybuty tworzą klucz złożony. Informuje jakie zaproszenia do bycia znajomymi zostały wysłane danemu użytkownikowi. Składa się z pól:

- *login użytkownika*, typ tekstowy, klucz obcy do tabeli *Użytkownicy*,
- *login zapraszającego*, typ tekstowy, klucz obcy do tabeli *Zaproszenia*.

Encje *Użytkownicy* i *Znajomi* zawierają te same pola jednak ze względu na możliwość późniejszego rozwoju aplikacji nie łączyliśmy tych danych do jednej encji. W przyszłości dzięki temu będzie możliwe w łatwiejszy sposób powiększanie ilości prywatnych informacji na temat użytkownika nie udostępnianych znajomym.

4.3. Klasy aplikacji mobilnej

W niniejszym podrozdziale omówione zostaną klasy aplikacji mobilnej. Przedstawione zostały tutaj najważniejsze klasy jakie należy zaimplementować w projekcie. Można je podzielić na te, które odzwierciedlają encję w bazach danych w celu łatwiejszego operowania na danych, które oferują podstawowe operacje bazodanowe oraz na takie które odpowiedzialne są za komunikację z serwerem i wykonujące logikę biznesowa aplikacji.

Wyróżniliśmy sześć klas odzwierciedlających encje w bazie danych na urządzeniu klienckim przedstawionej w poprzednim rozdziale. Zawierają one wszystkie takie same atrybuty jak dana tabela wraz z identycznymi ich typami oraz dodatkowo zapewniają dostęp do zmiennych przy pomocy odpowiednich metod *get* oraz *set*. Te klasy to:

- *Friends*, która odwzorowuje encje *Znajomi*, zawiera zmienne i metody oferujące dostęp do informacji na temat znajomych danego użytkownika takie jak login, imię, nazwisko e-mail, data urodzenia i miejsce zamieszkania,
- *Invitations*, która odwzorowuje encje *Zaproszenia*, zawiera zmienne i metody oferujące dostęp do informacji na temat osoby wysyłającej zaproszenie dla użytkownika do bycia znajomym takie jak login, imię, nazwisko i miejsce zamieszkania,
- *Messages*, która odwzorowuje encje *Wiadomości*, zawiera zmienne i metody oferujące dostęp do informacji przesyłanych komunikatów zarówno utworzonych przez użytkownika jak i wysłanych przez niego do znajomych takie jak identyfikator, login adresata i odbiorcy, datę wysłania i treść wiadomości,
- *Points*, która odwzorowuje encje *Punkty*, zawiera zmienne i metody oferujące dostęp do informacji na temat pojedynczego punktu przesłanego w ramach całej trasy, czyli identyfikatora punktu jak i trasy, do której się odnosi, daty oraz szerokości i długości geograficznej,
- *Routes*, która odwzorowuje encje *UdostępnioneTrasy*, zawiera zmienne i metody oferujące dostęp do informacji udostępnionych tras przez użytkownika jak i jego znajomych takich jak identyfikator, login osoby udostępniającej i daty,

- *Users*, która odwzorowuje encje *Użytkownicy*, zawiera zmienne i metody oferujące dostęp do podstawowej informacji na temat osób, które były zalogowane na danym urządzeniu mobilnym, czyli login, imię, nazwisko e-mail, data urodzenia i miejsce zamieszkania.

Dla każdej z encji w bazie danych która posiada swoje odwzorowanie w postaci obiektowej stworzono klasę oferującą podstawowe operacje bazodanowe takie jak aktualizacja, usunięcie, dodanie oraz pobranie danych. Jest to:

- klasa *FriendsDao*, udostępniająca metody operujące na encji *Znajomi*,
- klasa *InvitationsDao*, udostępniająca metody operujące na encji *Zaproszenia*,
- klasa *MessagesDao*, udostępniająca metody operujące na encji *Wiadomości*,
- klasa *PointsDao*, udostępniająca metody operujące na encji *Punkty*,
- klasa *RoutesDao*, udostępniająca metody operujące na encji *UdostępnioneTrasy*,
- klasa *UserDao*, udostępniająca metody operujące na encji *Użytkownicy*.

Do komunikacji z serwerem zaprojektowano klasę *ServerApi*, która wysyła i odbiera dane zgodnie ze stylem w architekturze oprogramowania REST API. Zawiera takie metody jak:

- *register*, która przesyła dane wprowadzone przez użytkownika w procesie rejestracji,
- *login*, która przesyła dane wprowadzone przez użytkownika w procesie logowania,
- *logout*, która przesyła informacje o wylogowaniu się danego użytkownika,
- *getUserInvitations*, która pobiera informacje o zaproszeniach użytkownika,
- *sendInvitations*, pobiera wszystkie zaproszenia skierowane do danego użytkownika,
- *dismissInvitation* zawiadamia o odrzuceniu danego zaproszenia,
- *acceptInvitation* informuje o zaakceptowaniu danego zaproszenia,
- *getUserFriends*, która pobiera informacje o wszystkich znajomych użytkownika,
- *startDanger* zawiadamia, że dana osoba znajduje się w niebezpieczeństwie,
- *sendPoint* przesyła aktualną lokalizację użytkownika,
- *getMyRoutes* pobiera wszystkie trasy, które udostępnił dany użytkownik,
- *getFriendsRoutes* pobiera wszystkie trasy udostępnione przez znajomych danego użytkownika,
- *getMessages* pobiera wiadomości danej osoby,
- *sendMessage* wysyła wiadomość do znajomego,
- *deleteFriend* usuwa danego użytkownika z listy znajomych,
- *sendToken* przesyła nowy token czyli ciąg znaków identyfikujący urządzenie mobilne.

Klasa *DangerForegroundService* odpowiada za przesyłanie pozycji użytkownika na serwer i do jego kontaktów w tle. Występuje tu funkcja tworząca powiadomienie informujące, że lokalizacja jest udostępniana oraz metoda ustalająca szerokość i długość geograficzną na której znajduje się użytkownik i przesyłająca te dane regularnie do aplikacji serwerowej.

Klasa *FirebaseService* odpowiedzialna jest za odbiór komunikatów przesyłanych przez Firebase Cloud Messaging, utworzenie powiadomień i zapisanie przesyłanych notyfikacji do bazy danych. Wyróżniliśmy tu następujące metody:

- *onMessageReceived*, która wywoływana jest podczas otrzymania notyfikacji poprzez Firebase Cloud Messaging,
- *onNewToken* przesyła zaktualizowany token na serwer przy pomocy klasy *ServerApi*,
- *startDanger* powiadamia użytkownika, że jego znajomy jest w niebezpieczeństwie,
- *addPoint* informuje użytkownika o aktualnej lokalizacji znajomego i zapisują ją do bazy danych,
- *newInvitation* informuje użytkownika o nowym zaproszeniu do grona znajomych i zapisuje je do bazy danych,
- *newMessage* informuje o nowej wiadomości dla użytkownika od znajomego,
- *newFriend* informuje, że dany użytkownik zaakceptował zaproszenie do grona znajomych.

Klasa *SynchronizeDataService* odpowiedzialna jest za zsynchronizowanie lokalnej instancji bazy danych z centralną znajdującą się na serwerze. Wyróżniliśmy tu następujące metody, które współpracują z funkcjami zawartymi w klasie *ServerApi*:

- *synchronizeUserData* odpowiada za synchronizację danych użytkownika w przypadku pierwszego logowania na danym urządzeniu,
- *SynchronizeInvitations* synchronizuje informacje na temat zaproszeń do grona znajomych użytkownika,
- *SynchronizeFriends* synchronizuje informacje na temat znajomych danego użytkownika,
- *synchronizeRoutes* synchronizuje informacje na temat tras zarówno użytkownika jak i jego znajomych,
- *synchronizeMessage* synchronizuje informacje na temat wiadomości utworzonych przez użytkownika przesyłanych do niego.

Klasa *ValidForms* odpowiedzialna jest za walidację danych wprowadzonych przez użytkownika zarówno w procesie rejestracji jak i logowania. Zawiera następujące metody:

- *validLogin*, która sprawdza, czy pole do wpisywania loginu jest poprawnie wypełnione, nie jest za krótkie i nie jest puste,
- *validName*, która sprawdza, czy pole do wpisywania imienia nie jest puste i jest wypełnione zgodnie z wymaganiami,
- *validSurname*, która sprawdza, czy pole do wpisywania nazwiska jest wypełniane zgodnie z wymaganiami i nie jest puste,
- *validEmail*, która sprawdza czy zawartość pola do wpisywania adresu mailowego nie jest pusta i czy jest zgodne z wymaganiami,
- *validPassword*, która sprawdza, czy pole do wpisywania hasła wypełnione jest prawidłowo i nie jest puste.

4.4. Klasy aplikacji serwerowej

W niniejszym podrozdziale przedstawione zostaną klasy aplikacji serwerowej. Omówione zostały tutaj klasy, które według nas są mają największe znaczenie podczas implementacji. Są

to klasy, które obsługują żądania od klientów zgodnie ze stylem w architekturze oprogramowania REST API oraz oferujące podstawowe operacje bazodanowe i serwisy wykonujące logikę biznesową aplikacji.

Zaprojektowano sześć klas odzwierciedlających encje w bazie danych na środowisku serwerowym przedstawionej w podrozdziale 4.2.1. Encje zawierają takie same atrybuty jak dana tabela wraz z identycznymi ich typami oraz dodatkowo zapewniają dostęp do zmiennych przy pomocy odpowiednich metod *get* oraz *set*. Te klasy to:

- *Friend*, która odwzorowuje encje *Znajomi*, zawiera zmienne i metody, które oferują dostęp do identyfikatorów użytkowników, którzy są ze sobą znajomymi,
- *Invitations*, która odwzorowuje encje *Zaproszenia*, zawiera zmienne i metody, które oferują dostęp do identyfikatora osoby wysyłającej zaproszenie i użytkownika, do którego zostało ono wysłane,
- *Message* odwzorowująca encje *Wiadomości*, zawiera zmienne i metody oferujące dostęp do informacji na temat przesyłanych komunikatów między użytkownikami takie jak identyfikator adresata, odbiorcy i wiadomości, oraz treść i datę wysłania,
- *Points* która odwzorowuje encje *PunktyUdostępnione*, zawiera zmienne i metody oferujące dostęp do informacji na temat pojedynczego punktu takie jak identyfikator punktu i trasy, szerokość oraz długość geograficzną i datę,
- *User*, która odwzorowuje encje *Użytkownicy*, zawiera zmienne i metody oferujące dostęp do informacji na temat osób mających konta w aplikacji, czyli login, hasło, imię, nazwisko, e-mail, data urodzenia, miejsce zamieszkania, token, który jest ciągiem znaków identyfikującym urządzenie, na którym ostatnio zalogował się użytkownik i identyfikator encji,
- *SharedRoute*, która odwzorowuje encje *UdostępnioneTrasy*, zawiera zmienne i metody oferujące dostęp do informacji na temat udostępnianych lokalizacji przez użytkownika takie jak identyfikator trasy i udostępniającego oraz data rozpoczęcia przesyłania danych.

Dla każdej z encji w bazie danych która posiada swoje odwzorowanie w postaci obiektowej stworzono klasę oferującą podobnie jak w aplikacji klienckiej podstawowe operacje bazodanowe. Jest to:

- klasa *FriendDao*, udostępniająca metody operujące na encji *Znajomi*,
- klasa *InvitationsDao*, udostępniająca metody operujące na encji *Zaproszenia*,
- klasa *MessageDao*, udostępniająca metody operujące na encji *Wiadomości*,
- klasa *PointDao*, udostępniająca metody operujące na encji *PunktyUdostepnione*,
- Klasa *SharedRouteDao*, udostępniająca metody operujące na encji *UdostepnioneTrasy*,
- klasa *UserDao*, udostępniająca metody operujące na encji *Użytkownicy*.

Ponadto utworzono klasy kontrolujące dostęp do informacji zgromadzonych na serwerze. Są one pierwszym punktem dostępowym do aplikacji serwerowej, które przekształcają informacje przesłane przy pomocy żądań HTTP na zestaw określonych danych. Są to klasy:

- *UserController*, która przyjmuje zapytania o przesłanie informacji na temat użytkownika oraz o jego rejestrację, logowanie i wylogowanie,

- *FriendsController*, która przyjmuje zapytania o przesłanie informacji na temat znajomych użytkownika oraz o usunięcie danej osoby z listy kontaktowej;
- *RoutesController* przyjmująca zapytania odnośnie rozpoczęcia udostępniania lokalizacji oraz o przesłanie informacji na temat tras użytkownika i znajomego;
- *InvitationsController*, która przyjmuje zapytania o przesłanie informacji na temat zaproszeń do danego użytkownika, a także o utworzenie nowego i ich akceptację lub odrzucenie;
- *MessagesController* przyjmująca zapytania odnośnie przesłania nowych wiadomości oraz pobrania danych na temat wszystkich skierowanych do danego użytkownika i przez niego przesłanych.

Ostatnim zestawem są serwisy, czyli klasy wykonujące logikę biznesową aplikacji i pośredniczące między warstwą dostępu do danych, a prezentacji realizowanej przy pomocy kontrolerów. Występują tu następujące klasy:

- *UserService* oferująca metody pozwalające na realizację rejestracji, logowania i wylogowania użytkownika oraz pobranie pełnej informacji o danym użytkowniku,
- *FriendsService* oferująca metody pobierania danych o znajomych określonego użytkownika i służące do usunięcia osoby z list kontaktów,
- *InviationsService* odpowiedzialna za wysyłanie powiadomienia o zaproszeniu jego akceptacji dla danego użytkownika do grona znajomych oraz oferuje metody odrzucające zaproszenie i pobierające je wszystkie dla danej osoby,
- *RoutesService* oferuje metody zwracające listę tras danego użytkownika oraz jego znajomych. Ponadto dysponuje funkcjami, które obsługują tworzenie nowych tras i dodawanie do nich punktów oraz powiadomienie o tym użytkowników,
- *MessageService* odpowiedzialna jest za obsługę nowych wiadomości i pobranie całej ich listy dla danego użytkownika,
- *FirebaseService* oferuje metody przesyłające komunikaty do danych urządzeń poprzez Firebase Cloud Messaging.

4.5. Podsumowanie

W fazie projektowania systemu określono, że planowaną architekturą będzie klient-serwer, gdzie aplikacja kliencka znajdować będzie się na urządzeniu mobilnym użytkownika wraz z lokalną instancją bazy danych przechowującą informacje tylko o osobach logujących się na danym urządzeniu. Aplikacja serwerowa znajduje się na urządzeniu serwerowym wraz z centralną bazą danych przechowującą informacje o wszystkich zarejestrowanych użytkownikach. Szczegółowe schematy tych baz danych wraz z opisem przedstawiono w podrozdziale 4.2. Główne ich encje przechowują informacje o użytkowniku, wiadomościach przez niego wysłanych, znajomych i udostępnionych lokalizacjach. W podrozdziałach 4.3 i 4.4 przedstawiono klasy jakie powinny być zaimplementowane w gotowym projekcie. Odpowiedzialne są one m.in. za wykonywanie operacji bazodanowych, odzwierciedlenie encji

przy pomocy obiektów, a także przesyłanie lokalizacji, odbieranie i tworzenie powiadomień oraz komunikację między aplikacjami.

5. IMPLEMENTACJA

Piąty rozdział pracy inżynierskiej zawiera informacje na temat przebiegu implementacji projektu stworzonego w poprzednim rozdziale. Omówiono najpierw krótko środowiska programowe wykorzystane podczas tworzenia aplikacji. Następnie przedstawiono zaimplementowane najważniejsze klasy i funkcje realizowane w ramach aplikacji serwerowej i mobilnej. Realizują one m.in. komunikację między aplikacjami, dostęp do bazy danych i logikę biznesową obu programów.

5.1. Środowiska programowe

Zgodnie z założeniami architektury zrealizowano dwie aplikacje, czyli kliencką i serwerową. Aplikacja serwerowa została napisana w środowisku *Intellij Idea* [14]. Wybrano to środowisko dlatego, że jest ono nam bardzo dobrze znane. Wykorzystywaliśmy je przy wielu projektach w języku Java. Dodatkowo umożliwia integrację z systemami kontroli wersji umożliwiającymi lepszą pracę w grupie. Ponadto oferuje liczne inteligentne podpowiedzi do wpisywanych klas, metod czy zmiennych oraz analizę kodu, która wskazuje podejrzane problemy i opcjonalnie oferuje listę prawdopodobnych szybkich poprawek wykrytych problemów, co znacznie ułatwia pisanie programów.

Aplikacja kliencka została zrealizowana w środowisku *Android Studio* [15], które wspiera tworzenie aplikacji mobilnych na system Android. Oferuje prace z kilkoma językami programowania takimi jak Kotlin, C++ oraz Java, która wykorzystywana była w projekcie. Program umożliwia tworzenie projektów dla różnych wersji systemu oraz posiada wbudowany emulator, na którym można testować działanie aplikacji na różnych wersjach i wielkościach oraz proporcjach ekranu. Dodatkowo jest możliwość instalacji i debugowania aplikacji na własnym urządzeniu mobilnym. Tworzenie interfejsu użytkownika ułatwia edytor graficzny w którym to dodawanie podstawowych elementów można realizować poprzez przeciągnięcia na ekranie. Android Studio udostępnia narzędzia do śledzenia zużycia pamięci, procesora, a nawet baterii przez napisaną aplikację umożliwiając optymalizację programu. Posiada również funkcjonalności zawarte w Intellij Idea takie jak stosowanie podpowiedzi, wskazywanie i analiza błędów oraz sugerowanie poprawek w kodzie.

5.2. Aplikacja serwerowa

Aplikacja serwerowa została napisana przy pomocy bibliotek Spring Boot. Łączy się ona z bazą danych utworzoną w języku MySQL zgodnie ze schematem przedstawionym dla środowiska serwerowego. Występują tu klasy obsługujące zapytania HTTP, logikę biznesową i dostęp do bazy danych.

5.2.1. Realizacja dostępu do bazy danych

Dostęp do bazy danych w aplikacji serwerowej został zrealizowany przy pomocy standardu Java Persistence *API*, którego dostawcą jest *Hibernate*. Dostęp do bazy danych uzyskano konfigurując plik *aplication.properties* w którym to podano adres, pod którym ona się

znajduje, jej typ oraz dane niezbędne do uzyskania dostępu takie jak login i hasło. Do operowania na bazie stworzono klasy, które odzwierciedlają encje oraz wykonują podstawowe operacje bazodanowe.

Klasy encyjne powstały na podstawie każdej z tabel bazy danych. To m.in. *Użytkownicy, Znajomi, Zaproszenia, UdostępnioneTrasy*. Dla każdej z nich została utworzona odpowiednia klasa, a ich implementacja jest bardzo podobna. Na rysunku 5.1 przedstawiono część klasy *User* odwzorowującej encję *UżytkownicyTrasa*.

```
@Entity
@Table(name = "Users")
public class User implements IEntity{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String login;

    @Column(name = "user_name")
    private String name;
    private String surname;
    private String city;
    private String email;

    @Column(name = "user_password")
    private String password;

    @Column(name = "date_of_birth")
    private LocalDate birth;
    private String token;

    public User(){}
```

Rysunek 5.1 Implemetacja klasy *User*

Taka klasa powinna być oznaczona odpowiednimi adnotacjami. Pierwszą jest *@Entity*, która wskazuje, że dana klasa odwzorowuje encję, następnie, jeśli nazwa klasy odbiega od tej w bazie danych dodano adnotację *@Table* z atrybutem *name*, którego wartość wskazuje poprawną nazwę tabeli w bazie danych. Identyfikator oznaczamy przy pomocy *@Id po* czym określamy jak będzie on generowany przy pomocy *@GeneratedValue* i atrybutu *strategy.* W tym przypadku ustalono, że jego wartość będzie automatycznie przypisywana podczas tworzenia obiektu na podstawie poprzednich wartości. Ostatnią używana adnotacją jest *@Column* działająca podobnie jak *@Table*, gdzie określamy nazwę atrybutu encji, do której dana zmienna się odnosi. Dostęp do wartości realizowany jest przy pomocy odpowiednich metod *get* i *set*. Przykładowo dla pola login mają one nazwę *getLogin,* która zwraca zmienną i *setLogin,* która ustawia pole określonym obiektem. Ponadto możliwe jest automatyczne tworzenie relacji między klasami, dzięki czemu pobierając przykładowo jeden punkt mamy dostęp do pełnej informacji, o trasie której dotyczy. Zrealizowane zostało to przy pomocy adnotacji *@ManyToOne.*

Kolejne interfejsy wykorzystywane w warstwie dostępu do danych oferują podstawowe operacje bazodanowe. Utworzone zostały one dla każdej z encji. Na rysunku 5.2 i 5.3 zostały przedstawione ich implementacje odpowiednio dla encji *Użytkownicy* i *UdostępnioneTrasy.*

```
public interface IUserDao extends CrudRepository<User,Long> {

    User findByLogin(String login);
    User findByToken(String token);
    User findByEmail(String email);

    @Transactional
    @Modifying
    @Query("Update User u SET u.token=?1 WHERE u.id=?2")
    void updateToken(String token,Long id);

}
```

Rysunek 5.2 Interfejs udostępniający metody bazodanowe dla encji *Użytkownicy*

```
public interface IUserRouteDao extends CrudRepository<UserRoute, UserRouteId> {

    @Query("SELECT r.id.route FROM UserRoute r WHERE r.id.user = ?1")
    Iterable<Long> getFriendsRoutesId(Long userId);

    @Query("SELECT r.id.user FROM UserRoute r WHERE r.id.route = ?1")
    Iterable<Long> getRoutesFriendId(Long routeId);
}
```

Rysunek 5.3 Interfejs udostępniający metody bazodanowe dla encji *UżytkownicyTrasa*

Interfejsy te oferują zdefiniowane przez użytkownika metody, które wykonują zapytania na bazie danych określone w adnotacji *@Query.* Dodatkowo rozszerzają interfejs *CrudRepository,* który definiuje metody do zapisu danych, szukania ich po identyfikatorze oraz odpowiedzialne za usunięcie pewnych informacji w bazie. Jeśli funkcja dodatkowo modyfikuje dane to oznaczono ją przy pomocy adnotacji, że informacje w bazie są aktualizowane i wykonywana jest transakcja dzięki czemu, jeśli metoda nie powiedzie się to dane przywracane są do stanu sprzed wywołania funkcji.

5.2.2. Realizacja logiki biznesowej aplikacji

Logika biznesowa aplikacji serwerowej została zrealizowana przy pomocy klas serwisów, które pośredniczą między warstwami dostępu do danych, a ich prezentacji. Oznaczane są one przy pomocy adnotacji *@Service.* W zaimplementowano serwisy, które korzystają z bazy danych takie jak:

- *FriendsService* oferujący dostęp do danych na temat znajomych określonego użytkownika,
- *RoutesService* obsługujący tworzenie i aktualizowanie tras użytkowników,

- *UserService* odpowiedzialny jest za rejestrację, logowanie i wylogowywanie użytkownika,
- *InviationsService* obsługujący dodawanie znajomych i przesyłanie zaproszeń oraz odpowiedzi na nie.

Na rysunku 5.4 przedstawiono implementację klasy *FriendsService.* Przedstawia ona przykład implementacji klasy serwisowej. Można zauważyć adnotacje *@Service* oraz *@Autowired*, która to automatycznie powiązuje w tym przypadku interfejsy oferujące operacje bazodanowe z tą klasą podczas uruchamiania aplikacji. Schemat każdego z wyżej wymienionych serwisów jest taki sam jedynie oferują inne metody.

```
@Service
public class FriendsService implements IFriendsService{

    @Autowired
    private IFriendDao friendDao;

    @Autowired
    private IUserDao userDao;

    @Override
    public Iterable<User> getFriends(String userLogin) {
        User user = userDao.findByLogin(userLogin);
        if(user == null)
            return null;

        Iterable<FriendId> ids = friendDao.getFriendsId(user.Id());
        List<Long> friendsId = new LinkedList<>();
        for (FriendId id : ids) {
            if (id.getUser1().equals(user.Id()))
                friendsId.add(id.getUser2());
            else
                friendsId.add(id.getUser1());

        }
        return userDao.findAllById(friendsId);

    }
}
```

Rysunek 5.4 Implementacja klasy *FriendService*

Poza wyżej wymienionymi serwisami jeszcze wyróżnić należy *FirebaseService*, który nie korzysta bezpośrednio z bazy danych jednak współpracuje z innymi klasami wykonującymi logikę biznesową, a jego zadaniem jest przesyłanie powiadomień do ściśle określonych użytkowników. Korzystamy tutaj z metod i bibliotek dostarczonych przez Firebase Cloud Messaging dla języka Java. Aby powiązać instancję z aplikacją został wygenerowany plik w formacie JSON, w którym zawarte są informacje m.in. na temat identyfikatora projektu, adresu chmury obliczeniowej i klucza prywatnego w celu bezpiecznego przesyłania wiadomości. Na rysunku 5.5 przedstawiona została implementacja klasy *FirebaseService*. Wiadomość przesyłana jest klasy *Message*, gdzie umieszczony jest najpierw token, który identyfikuje urządzenie klienta, następnie typ komunikatu, czyli czy dotyczy nowego znajomego, zaproszenia lub udostępniania trasy, aby aplikacja kliencka mogła poprawnie zinterpretować dane, a następnie reszta przesyłanych informacji na przykład

dane lokalizacyjne. Wiadomość przesyłana jest asynchronicznie, aby nie blokować dalszego działania metod.

```
@Service
public class FirebaseService {

    public void sendDataAsync(String token, IEntity entity,int messageType) throws FirebaseMessagingException{
        Message message = Message.builder()
                .setToken(token)
                .putData("type",String.valueOf(messageType))
                .putAllData(entity.ObjectMap())
                .build();

        FirebaseMessaging.getInstance().sendAsync(message);
    }

    public void sendDataAsync(List<String> token, IEntity entity, int messageType) throws FirebaseMessagingException{
        MulticastMessage message = MulticastMessage.builder()
                .putData("type",String.valueOf(messageType))
                .putAllData(entity.ObjectMap())
                .addAllTokens(token)
                .build();

        FirebaseMessaging.getInstance().sendMulticastAsync(message);
    }
```

Rysunek 5.5 Implementacja klasy FirebaseService

5.2.3. Obsługa zapytań HTTP

Zapytania HTTP zostały obsłużone przez klasy kontrolery, które przekształcają dane zawarte w formacie JSON na odpowiednie obiekty i przekazują do określonych metod zawartych w serwisach. Po wykonaniu funkcji zwracają wynik klientowi. Przy pomocy platformy Spring Boot cały ten proces jest mocno zautomatyzowany i uproszczony. Kontrolery oznaczono przy pomocy adnotacji *@RestController.* Zaimplementowano tu klasy:

- *FriendsController* obsługująca zapytania odnośnie znajomych użytkownika,
- *InvitationsController,* która odpowiada za żądania związane z zaproszeniami, takie jak wysłanie, odrzucenie, zaakceptowanie i wyświetlenie,
- *RoutesController*, która obsługuje zapytania na temat udostępniania lokalizacji
- *UserController,* która przyjmuje żądanie o zalogowanie, wylogowanie i rejestrację użytkownika

Każda z nich połączona jest z odpowiadającym jej serwisem. Każda z metod posiada adnotacje *@PostMapping,* jeśli obsługuje zapytania typu HTTP POST lub *@GetMapping*, jeśli obsługuje zapytania typu HTTP GET wraz z adresem, pod którym wykonywana jest funkcja. Spring Boot umożliwia automatyczne przekonwertowania treści zapytania na odpowiednie obiekty.

5.3. Aplikacja mobilna

Aplikacja mobilna została napisana w języku Java na system Android. Wykorzystywana jest tu domyślna baza danych SQLite z którą aplikacją łączy się przy pomocy biblioteki Room. Występuje tu wiele klas, które realizują m.in. dostęp do bazy danych, komunikację z serwerem i logikę biznesową. Dodatkowo występują tu klasy obsługujące zachowanie użytkownika i pliki

w formacie XML w których opisany jest interfejs stworzony przy pomocy edytora zawartego w środowisku programowym Android Studio.

5.3.1. Realizacja dostępu do bazy danych

Dostęp do bazy danych zrealizowany został zgodnie z najnowszymi zaleceniami, czyli wykorzystując bibliotekę Room. Wykorzystywane są tu klasy encyjne, interfejsy oferujące operacje bazodanowe oraz klasy repozytoria przetwarzające logikę biznesową związana z operacjami na bazie danych. Dodatkowo do wyświetlenia informacji wykorzystano klasy *ViewModel* przechowujące dane, które są zrozumiałe dla widoku.

Pierwszym działaniem jakie zrealizowano było utworzenie klas encyjnych na podstawie utworzonego projektu bazy danych na urządzeniu klienckim. Każda tabela odwzorowana została przy pomocy klasy. Ich implementacja wygląda podobnie jak w aplikacji serwerowej. Do ich oznaczenia wykorzystano adnotacji *@Entity*, a do identyfikatora *@PrimaryKey.* Posiada wszystkie takie same zmienne jak atrybuty w encjach, a dostęp do nich zapewnia przy pomocy odpowiednich metod *get* i *set.* Realizacja relacji wiele do wielu została zrealizowana przy pomocy dodatkowych klas w których określamy pomiędzy jakimi klasami i atrybutami ona zachodzi w adnotacji *@Relation*. Na rysunku 5.6 przedstawiono klasę *UserWithFriends* realizującą tą relacje między tabelą *Użytkownicy*, a *Znajomi.* Klasa ta zawiera informacje na temat jednej osoby i wszystkich jego znajomych. Odwrotna sytuacja również jest możliwa jednak akurat w tym przypadku nie była konieczna.

```
public class UserWithFriends {
    @Embedded
    private Users user;

    public void setUser(Users user) { this.user = user; }

    public void setFriends(List<Friends> friends) { this.friends = friends; }

    @Relation(
            associateBy = @Junction(UserFriends.class),
            parentColumn = "user_login",
            entityColumn = "friend_login"

    )
    private List<Friends> friends;

    public Users getUser() { return user; }

    public List<Friends> getFriends() { return friends; }
}
```

Rysunek 5.6 Implementacja relacji wiele do wielu w systemie Android

Operacje bazodanowe definiowane są w interfejsach oznaczonych adnotacją *@Dao*. Dostępne jest zdefiniowanie metod do dodawania, aktualizacji i usuwania danych przy pomocy odpowiednich adnotacji, a także utworzenie funkcji wykonujących bardziej zaawansowane zapytania przy pomocy *@Query*. Każda z tabel posiada taki interfejs. Na rysunku 5.7

przedstawiono interfejs, w którym zostały zaprezentowane metody wywoływane na encji *Użytkownicy*. Można zauważyć, że niektóre z metod zwracają obiekty *LiveData*, które to nasłuchują zmian zachodzących w bazie i zawierają najbardziej aktualne dane.

```
@Dao
public interface UserDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insert(Users user);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insert(List<Users> users);

    @Update
    void update(Users user);

    @Delete
    void delete(Users user);

    @Transaction
    @Query("SELECT * FROM Users WHERE user_login = :login")
    LiveData<UserWithFriends> getFriends(String login);

    @Query("SELECT * FROM Users WHERE user_login = :login")
    LiveData<Users> getUser(String login);

    @Query("SELECT * FROM Users")
    LiveData<List<Users>> getAllUsers();

    @Query("SELECT * FROM Users WHERE user_login = :login")
    Users getUserSync(String login);

    @Transaction
    @Query("SELECT * FROM Users WHERE user_login = :login")
    UserWithFriends getFriendsSync(String login);
}
```

Rysunek 5.7 Implementacja interfejsu *UserDao*

Cała baza danych zdefiniowana została przy pomocy klasy z adnotacją *@Database* w której to przy pomocy atrybutu *entities* przedstawiamy wszystkie klasy encyjne. Zapewnia ona metodę zwracającą obiekt bazy oraz interfejsy *@Dao,* których metody są automatyczne zaimplementowane przez bibliotekę Room. Na rysunku 5.8 przedstawiono klasę tworzącą lokalną instancje bazy danych w projekcie.

```
@Database(entities = {Friends.class, Invitations.class, Messages.class, Points.class, Routes.class,
        UserFriends.class, UserInvitations.class, UserRoutes.class, Users.class}, version = 3)
@TypeConverters({Converters.class})
public abstract class ProjectDatabase extends RoomDatabase {

    private static ProjectDatabase database;

    public static synchronized ProjectDatabase getDatabase(Context context){
        if(database == null){
            database = Room.databaseBuilder(context.getApplicationContext(),
                    ProjectDatabase.class, name: "Logged users database")
                    .fallbackToDestructiveMigration()
                    .build();
        }
        return database;
    }
```

Rysunek 5.8 Utworzenie lokalnej instancji bazy danych na urządzeniu mobilnym

Następnie utworzono klasy repozytoria, które odpowiedzialne są za wykonywanie metod określonych w interfejsach *Dao*, których wyniki zwracają. W związku z tym, że operacje te nie mogą być wykonywane w głównym wątku aplikacji to zleca je klasom *AsyncTask*, które wykonują je na osobnych wątkach. Utworzono sześć klas repozytoriów - *FriendsRepository, InvitationsRepository, MessagesRepository, PointsRepository, RoutesRepository, UserRepository* umożliwiających usuwanie, dodawanie, aktualizowanie oraz pobieranie informacji kolejno o znajomych, zaproszeniach, wiadomościach, udostępnianych lokalizacjach i trasach danego użytkownika oraz o wszystkich osobach zalogowanych na danym urządzeniu mobilnym.

W celu wyświetlenia danych w określonym widoku stworzono dodatkowo klasy *ViewModel*, które przechowują zmienne czytelne dla okna interfejsu pobrane z bazy danych przy pomocy repozytoriów.

5.3.2. Realizacja komunikacji z serwerem

Komunikacja z serwerem przebiega na dwa sposoby. W pierwszym poprzez protokoły HTTP, a w drugim przy pomocy Firebase Cloud Messaging. Pierwszy z kanałów jest dwustronny urządzenie mobilne zarówno przesyła dane na serwer oraz je odbiera. Drugi jest jednostronny, czyli aplikacja tylko odbiera informacje przesyłane z serwera. Wykorzystywany jest on głównie to otrzymywania powiadomień nawet gdy aplikacja nie jest otwarta.

Do implementacji pierwszego sposobu komunikacji wykorzystano bibliotekę *Retrofit2* umożliwiającą tworzenie klientów dla aplikacji serwerowej komunikujących się zgodnie ze stylem w architekturze oprogramowania REST API [16]. Utworzono w tym celu interfejs *ServerApi* w którym zdefiniowano metody przesyłające dane podane jako argumenty na serwer i odbierające odpowiedzi w formie obiektu *Call*. Funkcje zawarte w tym interfejsie zostały opisane w projekcie klas, nie trzeba ich implementować. Zajmuje się tym biblioteka Retrofit2. Należy dodatkowo określić przy pomocy adnotacji jaki rodzaj zapytania dana metoda realizuje. Wykorzystano *@GET* i *@POST* a po nich podano adres, na który ma być ta funkcja kierowana.

```
ServerApi api = ServerClient.getClient();
Call<ResponseModel> call =api.sendInvitation(user,login);
call.enqueue(new Callback<ResponseModel>() {
    @Override
    public void onResponse(@NotNull Call<ResponseModel> call, @NotNull Response<ResponseModel> response) {
        if(!response.isSuccessful())
            AlertDialogs.serverError(getContext());
        else if(response.body().getCode() == MessageCodes.OK.getCode())
            Toast.makeText(getContext(), text: "Zaproszenie zostało wysłane",Toast.LENGTH_LONG).show();
        else if(response.body().getCode() == MessageCodes.INVALIDLOGIN.getCode())
            Toast.makeText(getContext(), text: "Nieprawidłowy login",Toast.LENGTH_LONG).show();
        else
            Toast.makeText(getContext(), text: "Zaproszenie zostało już wcześniej wysłane",Toast.LENGTH_LONG).show();
    }

    @Override
    public void onFailure(@NotNull Call<ResponseModel> call, @NotNull Throwable t) {
        AlertDialogs.networkError(getContext());
    }
});
```

Rysunek 5.9 Komunikacja aplikacji mobilnej z serwerem

Wywołanie metody zawartej w interfejsie *ServerApi* przedstawiono na rysunku 5.9. Realizowane jest tu wysłanie zaproszenia do grona znajomych, gdzie podawane jako argumenty są login osoby zapraszającej i zaproszonej. Aby nie blokować głównego wątku aplikacji w oczekiwaniu na odpowiedź wykorzystano interfejs *Callback*, który wywoływany jest dopiero po otrzymaniu odpowiedzi. W zależności od odpowiedzi wykonywana jest stosowna akcja.

```
@Override
public void onMessageReceived(RemoteMessage remoteMessage) {

    Log.d(TAG,  msg: "From: " + remoteMessage.getFrom());

    if (remoteMessage.getData().size() > 0) {
        Log.d(TAG,  msg: "Message data payload: " + remoteMessage.getData());
        String stringType = remoteMessage.getData().get("type");
        if(stringType != null ) {
            int type = Integer.parseInt(stringType);
            switch(type) {
                case startDangerRoute:
                    startDanger(remoteMessage);
                    break;
                case newPoint:
                    addPoint(remoteMessage);
                    break;
                case newFriend:
                    newFriend(remoteMessage);
                    break;
                case newInvitation:
                    newInvitation(remoteMessage);
                    break;
                case newMessage:
                    newMessage(remoteMessage);
                    break;
                default:
                    break;
            }

        }
```

Rysunek 5.10 Obsługa wiadomości przesłanych przez Firebase Cloud Messaging

Do implementacji drugiego sposobu komunikacji stworzono nowy serwis *FirebaseService*, rozszerzający klasę *FirebaseMessagingService,* który posiada metodę odbierania wiadomości przedstawioną na rysunku 5.10. Jej parametrem jest klasa *RemoteMessage* posiadająca dane przesyłane z serwera w postaci mapy klucz - wartość. Wszystkie dane są typu tekstowego, dlatego przekształcenie ich do innego typu wymaga odpowiedniego parsowania. Każda wiadomość niezależnie od przeznaczenia trafia do tej metody. Dlatego ważne było na początku jej treści określić typ, który odczytujemy i w zależności od niego przekierowujemy do odpowiedniej funkcji zajmującej się jej obsługą. W nich najczęściej tworzymy nowe powiadomienia i zapisujemy przesłane informacje do bazy danych przy pomocy repozytoriów. Rozwiązanie to można w łatwy sposób rozszerzać o nowe komunikaty.

5.3.3. Udostępnianie lokalizacji użytkownika

Przesyłanie lokalizacji w sytuacji niebezpiecznej zostało zaimplementowane przy pomocy *foreground service*, dzięki któremu może być ono realizowane w tle i przez tak długi okres czasu

jaki będzie konieczny. Użytkownik wciska przycisk na ekranie telefonu w głównym widoku uruchamiając przy tym serwis. Dodatkowo informowany jest o możliwości zadzwonienia na służby ratownicze, który przekierowuje do dialera telefonu z wybranym numerem 112. Aplikacja powinna być kompatybilna z nowymi wersjami Androida. Zgodnie z nimi, jeśli serwis chce działać w tle to należy informować o tym użytkownika poprzez notyfikację. Rysunek 5.11 przedstawia pierwsze działanie podejmowane przy uruchomieniu serwisu, czyli utworzenie powiadomienia.

```
@Override
public int onStartCommand(Intent intent,int flags, int startId){
    Intent notificationIntent = new Intent( packageContext: this, HomeFragment.class);
    PendingIntent pendingIntent = PendingIntent.getActivity( context: this, requestCode: 0,notificationIntent, flags: 0);

    Notification notification =new NotificationCompat.Builder( context: this, FOREGROUND_SERVICE_CHANNEL)
            .setContentTitle("Udostępnianie lokalizacji")
            .setContentText("Niebezpieczeństwo! Twoja lokalizacja udostępniana jest znajomym")
            .setSmallIcon(R.drawable.ic_baseline_location_on_black)
            .setContentIntent(pendingIntent)
            .build();

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
            startForeground( id: 1,notification, ServiceInfo.FOREGROUND_SERVICE_TYPE_LOCATION);
        }
        else
            startForeground( id: 1,notification);
```

Rysunek 5.11 Utworzenie *foreground service* na urządzeniu z systemem Android

Ustalanie lokalizacji urządzenie zrealizowane zostało przy pomocy klasy *FusedLocationProviderClient.* Dostęp do niej wykonywany jest za uprzednim zezwoleniem użytkownika. Lokalizacja pobierana jest z wysokim priorytetem, ale w celu mniejszego poboru energii ustalono, że będzie ona aktualizowana co dwie minuty. Dodano metodę *locationCallback* wywoływaną, gdy nastąpi zmiana lokalizacji. Przesyłana jest ona regularnie na serwer co 4 minuty Wykorzystano do tego obiekt typu *Timer* wywołujący metodę w osobnym wątku co określony czas przesyłającą dane na serwer. Na rysunku 5.12 przedstawiono powyżej opisane działania inicjalizacji obiektu ustalającego lokalizację użytkownika i obiektu wywołującego cyklicznie metodę do jej przesyłania.

```
location = LocationServices.getFusedLocationProviderClient( context: this);
LocationRequest locationRequest = new LocationRequest();
locationRequest.setInterval(1000*60*2);
locationRequest.setFastestInterval(1000*60);
locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
location.requestLocationUpdates(locationRequest, locationCallback,Looper.getMainLooper());

runnable =new Runnable() {
    @Override
    public void run() { doInBackground(); }
};

new Thread(runnable).start();
timer = new Timer();
timerTask = (TimerTask) () -> { new Thread(runnable).start(); };
timer.schedule(timerTask, delay: 1000*60*4, period: 1000*60*4);

return START_STICKY;
```

Rysunek 5.12 Cykliczne ustalanie lokalizacji użytkownika

Użytkownik otrzymawszy powiadomienie, że jego znajomy jest w niebezpieczeństwie może przejść w sekcję udostępnianych mu tras. Po wybraniu odpowiedniej wyświetlone jest mu jak najbardziej aktualne położenie znajomego na mapie oraz możliwość nawigacji do tej lokalizacji, która przekierowuje użytkownika do aplikacji *Google Maps* gdzie ukazane są możliwości tras dla różnych środków transportu. Na rysunku 5.13 ukazano, jak zostało zaimplementowane otworzenie aplikacji *Google Maps*. Wykorzystano tu odpowiednio sporządzony adres, w którym zawarta jest aktualna lokalizacja użytkownika oraz znajomego. Lokalizacja użytkownika została ustalona również przy pomocy obiektu *FusedLocationProviderClient.*

```
public void onClick(View v) {
    if(isFriendRoute) {
        if (mDestination != null && mOrigin != null) {
            String uri = String.format(Locale.ENGLISH,  format: "http://maps.google.com/maps?saddr=%f,%f&daddr=%f,%f",
                    mOrigin.latitude, mOrigin.longitude, mDestination.latitude, mDestination.longitude);
            Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(uri));
            intent.setPackage("com.google.android.apps.maps");
            startActivity(intent);
        }
    }
```

Rysunek 5.13 Przekierowanie użytkownika do nawigacji w aplikacji Google Maps

5.4. Podsumowanie

W fazie implementacji projektu udało się zrealizować większość wymagań funkcjonalnych określonych podczas analizy projektu oraz klas i ich metod przedstawionych w poprzednim rozdziale. Aplikację serwerową i mobilną napisano przy pomocy języka programowania Java. Dodatkowo w pierwszej z nich wykorzystano biblioteki Spring Boot oraz standard Java Persistence Api, którego dostawcą jest *Hibernate*. W drugiej natomiast wykorzystano Room oraz Retrofit2. Dodatkowo komunikuje się ona z aplikacją *Google Maps* w celu łatwej nawigacji użytkownika do znajomego. Baza danych na serwerze wykorzystuje język

MySQL, a na urządzeniu mobilnym SQLite. Do implementacji aplikacji serwerowej wykorzystano środowisko programowe Intellij Idea, a klienckiej Android Studio, które ułatwiły proces implementacji programów.

6. TESTOWANIE

Szósty rozdział pracy inżynierskiej poświęcony jest informacji na temat testowania funkcjonalnego aplikacji, stworzonej w ramach projektu inżynierskiego. Omówione zostaną scenariusze testowe, gdzie określone zostały kroki do wykonania i oczekiwane rezultaty działania aplikacji. Pod każdym testem zawarta jest informacja na temat wyniku przeprowadzonego testu.

6.1. Funkcjonalności aplikacji - scenariusze testowe

Przeprowadzono dwanaście testów funkcjonalnych. W tabelach od 6.1 do 6.12 przedstawiono scenariusze testowe wraz z ich wynikiem. Do pierwszych dwóch testów nie należy być uprzednio zalogowanym, ale w następnych jest to wymagane.

Tabela 6.1 Scenariusz testowy *Rejestracja*

Scenariusz	Oczekiwany Rezultat
1. Klikamy na etykietę aplikacji. 2. Wybieramy przycisk rejestracji. 3. W polu login wpisujemy login, który będziemy dalej wykorzystywali przy wejściu do systemu. 4. W polu imię wpisujemy imię. 5. W polu nazwiska wpisujemy nazwisko. 6. W polu email podajemy adres poczty. 7. W polu hasło ustawiamy hasło, które jest nie krótsze niż 8 znaków w tym jeden znak specjalny, jedna duża litera i jedna cyfra. 8. Wybieramy przycisk, zapisujący dane rejestracji.	1. Uruchomienie aplikacji, otwiera się, zawierający pola dla loginu i hasła, oraz dwa przyciski: logowania i rejestracji. 2. Otwiera się widok, która zawiera pola do wypełnienia danymi, potrzebnymi do rejestracji. 3. Login zostaje wpisany w polu login. 4. Imię zostało zapisane w polu imię. 5. Nazwisko zostało zapisane w polu nazwisko. 6. W polu adresu email został zapisany wpisany adres mailowy. 7. Zostało wpisane hasło, zawierające 8 znaków, w tym duża litera, znak specjalny i cyfra. 8. Po wciśnięciu przycisku otworzył się widok początkowy.
Wynik: Rejestracja powiodła się, dane zostały zapisane w bazie danych na serwerze.	

Tabela 6.2 Scenariusz testowy *Logowanie*

Scenariusz	Oczekiwany rezultat
1. Kliknij na przycisk, uruchamiający aplikację. 2. W polu login wpisujemy login. 3. W polu hasło wpisujemy hasło. 4. Wybieramy przycisk, pozwalający na wejście do systemu aplikacji.	1. Uruchomienie aplikacji, pojawienie się widoku z możliwością logowania się i rejestracji. 2. W polu login został zapisany login użytkownika. 3. W polu hasło wprowadzono hasło. 4. Wciśnięty został przycisk logowania. Otwiera się widok główny aplikacji.
Wynik: Logowanie się powiodło, dane o loginie i haśle zostały poprawnie odebrane z bazy danych.	

Tabela 6.3 Scenariusz testowy *Wylogowanie*

Scenariusz	Oczekiwany Rezultat
1. Klikamy na etykietę aplikacji. 2. Na głównym widoku wybieramy przycisk menu bocznego. 3. Wybieramy przycisk wyloguj, żeby opuścić system.	1. Uruchomienie aplikacji. 2. Otworzyło się menu boczne z listą przycisków. 3. Wciśnięty został przycisk wylogowania się. Przekierowanie na widok początkowy, który zawiera pola i przyciski do logowania i rejestracji.
Wynik: Poprawne wylogowanie się z systemu.	

Tabela 6.4 Scenariusz testowy *Dodawanie nowego kontaktu*

Scenariusz	Oczekiwany rezultat
1. Klikamy na przycisk z etykietą aplikacji. 2. Wybieramy menu boczne. 3. Wśród przycisków wybieramy przycisk o nazwie znajomi. 4. W polu wpisujemy login znajomego. 5. Wybieramy przycisk, która wysyła zaproszenie.	1. Aplikacja uruchamia się. 2. Otwiera się menu boczne. 3. Otwiera się widok, w którym trzeba wpisać login. 4. Login znajomego został wpisany. 5. Po wciśnięciu przycisku pojawia się komunikat, informujący o tym, że zaproszenie zostało wysłane.
Wynik: Zaproszenie zostało wysłane, kontakt otrzymał zaproszenie i może go zaakceptować lub odrzucić.	

Tabela 6.5 Scenariusz testowy *Akceptacja zaproszenia*

Scenariusz	Oczekiwany rezultat
1. Klikamy na etykietę aplikacji. 2. Wybieramy przycisk menu bocznego. 3. Wybieramy przycisk, który przekieruje nas na listę zaproszeń. 4. Wybieramy zaproszenie, które chcemy zaakceptować. 5. Wybieramy przycisk, który akceptuje zaproszenie.	1. Aplikacja się uruchamia. 2. Otwiera się menu boczne z listą dostępnych przycisków. 3. Otwiera się lista zaproszeń użytkownika. 4. Wybrane zaproszenie jest oznaczone jak wybrane (inny kolor). 5. Zaproszenie zostało zaakceptowane i znika z listy.
Wynik: Kontakt został zaakceptowany i pojawił się na liście znajomych.	

Tabela 6.6 Scenariusz testowy *Odrzucenie zaproszenia*

Scenariusz	Oczekiwany rezultat
1. Klikamy na etykietę aplikacji. 2. Wybieramy przycisk menu bocznego. 3. Wybieramy przycisk, który przekieruje nas na listę zaproszeń. 4. Wyodrębniamy zaproszenie, które chcemy odrzucić. 5. Wybieramy przycisk, który odrzuca zaproszenie.	1. Aplikacja się uruchamia. 2. Otwiera się menu boczne z listą dostępnych przycisków. 3. Otwiera się lista zaproszeń użytkownika. 4. Wybrane zaproszenie jest oznaczone jak wybrane (inny kolor). 5. Zaproszenie zostało odrzucone.

Wynik: Zaproszenie zostało odrzucone i znika z listy aktualnych zaproszeń.

Tabela 6.7 Scenariusz testowy *Udostępnienie lokalizacji swoim znajomym za pomocą przycisku, wysyłanie i otrzymanie powiadomienia*

Scenariusz	Oczekiwany rezultat
1. Uruchamiamy aplikację, klikając na etykietę. 2. Otwiera się widok, zawierający przycisk, który udostępnia lokalizację i menu boczne. 3. Wybieramy przycisk, który udostępnia lokalizację i wysyła powiadomienie kontaktom z listy.	1. Aplikacja uruchamia się. 2. Otworzył się widok z przyciskiem, który wysyła dane lokalizacji i powiadomienie kontaktom z listy i menu bocznego. 3. Przycisk został wciśnięty, a powiadomienie zostało wysłane i wyświetla się na ekranie odbiorcy.
Wynik: Użytkowniku udało się wysłać dane o swojej lokalizacji znajomym za pomocą przycisku, a odbiorca otrzymał powiadomienie, które zawiera dane lokalizacji i krótką treść z prośbą o pomoc.	

Tabela 6.8 Scenariusz testowy *Możliwość wizualizacji przesłanej lokalizacji znajomego na mapie*

Scenariusz	Oczekiwany rezultat
1. Klikamy na etykietę aplikacji. 2. Na głównym widoku wybieramy przycisk menu bocznego. 3. Wybieramy przycisk, który wyświetla udostępnione trasy. 4. Wybieramy jedną z udostępnionych tras. 5. Następnie klikamy na przycisk, który znajdzie trasę.	1. Aplikacja się uruchamia. 2. W wybranym menu bocznym wybieramy przycisk, która otwiera listę udostępnionych tras. 3. Zostaliśmy przekierowani na widok z listą udostępnionych tras. 4. Zostaliśmy przekierowani na widok z mapą, która wyświetla aktualne położenie użytkownika. 5. Otwiera się widok ze zbudowaną trasą.
Wynik: Trasa została zbudowana i wizualizowana na mapie, pokazując możliwe sposoby dojazdu do punktu końcowego od aktualnego punktu lokalizacji użytkownika.	

Tabela 6.9 Scenariusz testowy *Wysyłanie wiadomości do kontaktu z listy*

Scenariusz	Oczekiwany rezultat
1. Klikamy na etykietę aplikacji. 2. Z ekranu głównego wybieramy przycisk menu bocznego. 3. W wyświetlonym menu wybieramy przycisk, który przekierowuje do widoku z listą znajomych. 4. Wśród listy wybieramy kontakt, do którego chcemy napisać. 5. Wybieramy przycisk, który pozwoli napisać wiadomość. 6. W polu tekstowym piszemy wiadomość; 7. Wybieramy przycisk, wysyłający wiadomość.	1. Uruchamia się aplikacja. 2. Wyświetla się lista przycisków z menu bocznego. 3. Wyświetla się lista znajomych. 4. Wyświetla się informacja o kontakcie. 5. Otwiera się widok, w którym można wysłać wiadomość. 6. Treść wiadomości jest widoczna w polu tekstowym. 7. Wiadomość została wysłana do wybranego kontaktu.

Wynik: Funkcjonalność nie została zrealizowana w projekcie.

Tabela 6.10 Scenariusz testowy *Możliwość otrzymania wiadomości*

Scenariusz	Oczekiwany rezultat
1. Klikamy na etykietę aplikacji. 2. W otwartym widoku głównym wybieramy przycisk menu bocznego. 3. W otwartym menu wybieramy przycisk, który wyświetli listę wiadomości. 4. Wybieramy wiadomość, która oznaczona jest, jako nieprzeczytana (inny kolor).	1. Uruchamiamy aplikację. 2. Otwiera się menu boczne. 3. Obok przycisku wiadomości znajduje się etykieta, która informuje o nowej wiadomości. 4. Wyświetla się treść wiadomości, została ona otrzymana i przeczytana.
Wynik: Funkcjonalność nie została zrealizowana w projekcie.	

Tabela 6.11 Scenariusz testowy *Usuwanie kontaktu z listy*

Scenariusz	Oczekiwany rezultat
1. Klikamy na etykietę aplikacji. 2. W widoku głównym wybieramy przycisk menu bocznego. 3. W menu bocznym wybieramy przycisk, który przekieruje na listę kontaktów. 4. W liście kontaktów wybieramy ten, który chcemy usunąć. 5. W zakładce z informacją o kontakcie, wybieramy przycisk, który usunie kontakt z listy. 6. Potwierdzamy przyciskiem akceptującym akcję.	1. Uruchamia się aplikacja. 2. Otwiera się menu boczne. 3. Otwiera się lista kontaktów. 4. Wyświetla się informacja o kontakcie. 5. Wybieramy przycisk, która usuwa kontakt z listy. 6. Po zatwierdzeniu akcji przyciskiem, kontakt znika z listy.
Wynik: Funkcjonalność nie została zrealizowana w projekcie.	

Tabela 6.12 Scenariusz testowy *Edycja danych profilu*

Scenariusz	Oczekiwany rezultat
1. Klikamy na etykietę aplikacji. 2. Wybieramy przycisk menu bocznego. 3. Wybieramy przycisk, które przekieruje do widoku z danymi profilu użytkownika. 4. W widoku z danymi profilu wybieramy przycisk, który pozwoli na edycję danych. 5. W widoku edycji wpisujemy dane, które chcemy poprawić. 6. Zatwierdzamy akcje przyciskiem zapisu zmian.	1. Uruchamia się aplikacja. 2. Otwiera się lista dostępnych przycisków. 3. Otwiera się widok z informacją o profilu użytkownika. 4. Otwiera się widok, który zawiera pola z loginem, imieniem, nazwiskiem, pocztą i hasłem. 5. W potrzebnym nam polu wpisujemy nowe dane. 6. Po wciśnięciu przycisku dane użytkownika zostaną zmienione.
Wynik: Funkcjonalność nie została zrealizowana w projekcie.	

6.2. Podsumowanie

Testowanie jest jedną z ważniejszych części tworzenia oprogramowania, która zapewnia jakość produktu i pomaga programistom wykrywać błędy i usterki. Testowanie pomaga poprawić ogólną wydajność i dokładność systemu. Testując aplikację, którą tworzyliśmy w ramach projektu inżynierskiego, sprawdziliśmy poprawność działania realizowanych funkcjonalności. Zespół dążył do tego, by założenia i główny cel projektu zostały zrealizowane. Wymagania o priorytecie wysokim były spełnione i wszystkie funkcjonalności działają zgodnie z założeniami. Niektóre wymagania o priorytecie średnim i niskim zespół pozostawił do możliwego dalszego rozwoju aplikacji.

7. PODSUMOWANIE

Za cel pracy inżynierskiej przyjęto implementację aplikacji mobilnej na system Android, która zaoferuje większe bezpieczeństwo jej użytkowników poprzez łatwe udostępnianie lokalizacji znajomym. Na początku przeanalizowano wiedzę jaka jest przydatna przy tego typu projekcie, czyli o tworzeniu aplikacji mobilnych oraz serwerowych, a także o sposobach komunikacji między nimi oraz z bazami danych. Następnie utworzono analizę projektu, gdzie określono wymagania funkcjonalne podzielone według priorytetu na wysoki średni i niski. Priorytet wysoki otrzymały takie wymagania jak m.in. możliwość rejestracji, logowania i zapraszania znajomych oraz szybkie i łatwe udostępnianie lokalizacji. Wymagania pozafunkcjonalne dotyczyły danych, jakości, wydajności, elastyczności oraz użyteczności systemu. Ponadto utworzono w ramach tej fazy schemat przypadków użycia, gdzie określono działania jakie powinien móc podejmować w aplikacji użytkownik. Po tym etapie przystąpiono do zdefiniowania architektury systemu, gdzie zdefiniowano, że system składać będzie się z aplikacji serwerowej i klienckiej znajdującej się na urządzeniu mobilnym. Dla każdej z nich stworzono i opisano schemat baz danych, a następnie określono ich diagramy klasy. Po zrealizowaniu tych etapów przystąpiono do implementacji, w której zrealizowano dwie aplikacje, czyli serwerową i kliencką, które współpracują i komunikują się ze sobą oraz bazy danych oparte o MySQL i SQLite. Na końcu przetestowano ich działanie przy pomocy testowania funkcjonalnego na podstawie przygotowanych scenariuszy testowych.

Udało się zrealizować wszystkie wymagania funkcjonalne o najwyższym priorytecie. Użytkownik ma możliwość rejestracji do systemu oraz logowania i wylogowywania na dowolnym urządzeniu z systemem Android z wersją minimum 5.0 niezależnie od proporcji i wielkości ekranu. Interfejs jest czytelny i prosty w obsłudze. Dodatkowo każda osoba zalogowana posiada możliwość dodania znajomych poprzez wysłanie zaproszeń, które można zaakceptować. Zrealizowano również udostępnianie lokalizacji, które najbardziej może przyczynić się do zwiększenia poczucia bezpieczeństwa. Realizowane jest to przy pomocy łatwo dostępnego przycisku w interfejsie lub poprzez potrząśnięcie telefonem. Po czym istnieje jeszcze możliwość szybkiego połączenia ze służbami ratunkowymi. Znajomy automatycznie informowany jest o niebezpieczeństwie przy pomocy notyfikacji, a udostępnioną lokalizacja wyświetla się na mapie. Znajomy może w każdej chwili przejść do nawigacji ukazującej mu trasę do danej lokalizacji. Z wymagań funkcjonalnych o średnim priorytecie zrealizowano możliwość odrzucenia zaproszenia i dostęp do części funkcji systemu bez połączenia z internetem. Wszystkie te opisane funkcje uzyskały pozytywne wyniki podczas testowania.

Nie zrealizowano natomiast możliwości przesyłania wiadomości między użytkownikami, możliwości usuwania kontaktu z listy znajomych oraz edycji danych użytkownika. Funkcje te miały priorytet średni lub niski. Architektura i bazy danych jednak już są przystosowane pod implementację tej funkcjonalności. Dodatkowo bezpieczeństwo przesyłanych danych jest na niskim poziomie. Pracowano również nad własnym algorytmem budowania tras, aby uniezależnić się od zewnętrznych aplikacji jednak efekty nie były zadowalające i ostatecznie zarzucono jego implementację a skorzystano z *Google Maps API*.

Prace nad aplikacjami mogą być kontynuowane. Istnieje wiele możliwości ich rozwoju. Przede wszystkim głównymi aspektami nad którymi najpierw należałoby się pochylić jest implementacja niezrealizowanych rozwiązań takich jak pisanie wiadomości między użytkownikami, usuwanie znajomych oraz edycja danych użytkownika. Ponadto kontynuować należy prace nad zwiększonym bezpieczeństwem uwierzytelniania i przesyłania danych pomiędzy klientem, a serwerem. Jeśli chodzi o nowe funkcjonalności to aplikacja może być rozszerzona w kolejnych wersjach systemu o przesyłanie lokalizacji innymi środkami komunikacji, nie tylko poprzez połączenie internetowe. Dodatkowo funkcję udostępniania lokalizacji można rozszerzyć o przesyłanie lokalizacji znajomym, jeśli osoba wyjdzie poza określony teren. W dalszym rozwoju warto również uniezależnić się od zewnętrznych aplikacji i map od Google poprzez stworzenie własnych.

WYKAZ LITERATURY

[1] *What is Android? Here's everything you need to know,* https://www.androidauthority.com/what-is-android-328076/ (dostęp: 4.11.2020 r.)

[2] *Mobile Operating System Market Share Worldwide,* https://gs.statcounter.com/os-market-share/mobile/worldwide (dostęp: 9.11.2020 r.)

[3] *Application Fundamentals,* https://developer.android.com/guide/components/fundamentals (dostęp: 9.11.2020 r.)

[4] *Distribution data for Android*, https://androiddistribution.io/#/ (dostęp: 9.11.2020 r.)

[5] *Request location permissions,* https://developer.android.com/training/location/permissions (dostęp: 9.11.2020 r.)

[6] *Get the last known location,* https://developer.android.com/training/location/retrieve-current (dostęp: 9.11.2020 r.)

[7] *Save data in a local database using Room*, https://developer.android.com/training/data-storage/room (dostęp: 9.11.2020 r.)

[8] Firebase *Cloud Messaging,* https://firebase.google.com/docs/cloud-messaging (dostęp: 10.11.2020 r.)

[9] About *FCM messages,* https://firebase.google.com/docs/cloud-messaging/concept-options (dostęp: 12.11.2020 r.)

[10] FCM *Architectural Overview,* https://firebase.google.com/docs/cloud-messaging/fcm-architecture (dostęp: 12.11.2020 r.)

[11] *Wstęp do REST API,* https://devszczepaniak.pl/wstep-do-rest-api/ (dostęp: 12.11.2020 r.)

[12] Spring *Boot Reference Documentation,* https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/ (dostęp: 12.11.2020 r.)

[13] The *Java EE 6 Tutorial, Chapter 32 Introduction to the Java Persistence API,* https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html (dostęp: 12.11.2020 r.)

[14] IntelliJ *IDEA: The Java IDE for Professional Developers by JetBrains,* https://www.jetbrains.com/idea/ (dostęp: 25.11.2020 r.)

[15] Download *Android Studio and SDK tools* | *Android Developers*, https://developer.android.com/studio (dostęp: 25.11.2020 r.)

[16] *Retrofit*, https://square.github.io/retrofit/ (dostęp: 25.11.2020 r.)

Spis rysunków:

Spis tabel:

Dodatek A: Diagram klas aplikacji mobilnej i serwerowej

Do pracy inżynierskiej dodane zostały diagramy klas aplikacji mobilnej i serwerowej, które to zostały częściowo opisane w podrozdziale 4.3, 4.4 oraz 5.2 i 5.3