



POLITECHNIKA
LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI

Projekt zespołowy inżynierski

na kierunku Informatyka
na bloku dyplomowania Aplikacje Internetowe

Aplikacja mobilna wspomagająca turystykę górską.

Magdalena Kamińska
97628

Katarzyna Jakubowska
97620

Promotor Dr inż. Piotr Kopniak

Lublin 2025

Spis treści

1	Streszczenie	4
2	Wstęp	5
3	Cel i zakres pracy	7
3.1	Podział pracy	8
4	Analiza porównawcza cech wybranych aplikacji mobilnych umożliwiających korzystanie z map	10
4.1	Platforma Mapa Turystyczna	10
4.2	Platforma Mapy.cz	11
5	Przegląd narzędzi implementacyjnych	14
6	Projekt aplikacji	16
6.1	Wymagania funkcjonalne	16
6.2	Wymagania niefunkcjonalne	17
6.3	Diagramy przypadków użycia	18
6.4	Scenariusze	22
6.5	Diagram klas	27
6.6	Diagram ERD	31
6.7	Projekt interfejsu użytkownika	34
7	Implementacja	42
7.1	Moduł map	42
7.2	Moduł zgłoszeń	48
7.3	Moduł logowania i rejestracji	50
7.4	Widoki - użytkownik	52
7.5	Widoki - administrator	65
8	Testy	70
8.1	Testy jednostkowe	70
8.2	Testy manualne	73
9	Wnioski	84

1 Streszczenie

Niniejsza praca jest opisem procesu projektowania i implementacji aplikacji mobilnej usprawniającej nawigowanie po szlakach górskich. Wykorzystuje do tego mapy udostępnione do użytku przez Google Maps Platform, a cała aplikacja napisana została w języku Java. Dostęp do funkcji aplikacji ograniczony jest odpowiednimi rolami: administrator oraz użytkownik. Administrator jako kluczowa jednostka w działaniu systemu zarządza szlakami, wysyła ogólnosystemowe powiadomienia, zatwierdza zgłoszenia wysłane przez użytkowników. Użytkownik natomiast może cieszyć się dostępem do nawigacji, przeglądania map celem wybrania trasy, dostępem do swojej historii wędrówek oraz skrótem do telefonów alarmowych przydatnych w razie wypadku na szlaku.

Słowa klucze: Java, aplikacja mobilna, nawigacja, mapy, wyznaczanie trasy, Android

Abstract

This thesis describes the design and implementation of a mobile application that improves navigation on mountain trails. For this purpose, it uses maps made available for use by Google Maps Platform, and the entire application was written in Java. Access to application functions is limited by appropriate roles: administrator and user. The administrator, as a key unit in the operation of the system, manages trails and attractions, sends system-wide notifications, and approves applications sent by users. The user can enjoy access to navigation, browsing maps to choose a route, access to their hiking history and easier access to emergency phones useful in the event of an accident on the trail.

Keywords: Java, mobile application, navigation, maps, route mapping, Android

2 Wstęp

Polskie góry od zawsze cieszyły się niezwykłą popularnością wśród turystów zarówno z kraju, jak i z zagranicy. Nie jest to nic dziwnego, polskie szczyty są dosyć niskie i możliwe do zdobycia dla większości ludzi, gdyż nie wymagają ani profesjonalnego sprzętu, ani kondycji fizycznej wieloletniego sportowca. W erze mobilności i posiadania wszystkiego na wyciągnięcie ręki coraz mniej zwiedzających decyduje się na korzystanie z tradycyjnych map papierowych z wyznaczonymi szlakami górkimi. Mapy papierowe oczywiście są bardzo dobrym, sprawdzonym rozwiązaniem, lecz są podatne na zniszczenie przez zamoczenie czy podarcie. Ponadto zajmują miejsce w plecaku turysty i często sprawiają problem z ich po-prawnym złożeniem, zwłaszcza będąc na szlaku. Posługiwanie się nimi wymaga także przy-najmniej podstawowej umiejętności odczytywania danych z mapy oraz dobrej orientacji w terenie i obsługi kompasu. Mapy mobilne niwelują wiele z tych problemów, a przy okazji ułatwiają poruszanie się po szlakach, ponieważ wykorzystując moduł GPS w urządzeniu mo-bilnym, pokazują na bieżąco położenie wędrowca.

Już w 2. połowie XIX wieku turystyka górska zyskała znaczną popularność, w szcze-gólności w Tatrach. Niezbadane i nieoznakowane dotąd tereny budziły zainteresowanie po-dróżników, którzy wyrażali chęć powołania stowarzyszenia łączącego pasjonatów wypraw górkich. Dlatego już w 1873 roku powstało Towarzystwo Tatrzańskie, pierwsza taka organi-zacja na ziemiach polskich i szósta na świecie [ptt]. Członkom Towarzystwa zależało przede wszystkim na zbadaniu Karpat, ochronie przyrody, w szczególności zwierząt alpejskich oraz rozpowszechnianiu zebranych informacji o tym paśmie górkim celem jeszcze większego spopularyzowania turystyki górkiej na terenie Polski i ułatwieniu zainteresowanym zwie-dzania, a także wsparciu szeroko pojętego przemysłu górkiego. Za poboczny cel utworzenia Towarzystwa można również uznać chęć zjednoczenia narodu polskiego, żyjącego w tam-tych latach pod zaborami, jako próba wspólnego wystąpienia przeciwko wrogom. Towarzy-stwo w 1920 roku zmieniło swoją nazwę na Polskie Towarzystwo Tatrzańskie, lecz jego cele pozostały bez zmian. Dzięki jego działalności rejony Tatr cieszyły się tak wielkim zaintere-sowaniem, że już w 1889 roku zauważono potrzebę podjęcia odpowiednich kroków w celu ochrony przyrody, jednak Tatrzański Park Narodowy utworzony został dopiero po wojnie w 1954 roku [tpn].

Fakt, iż polskie Tatry cieszą się ogromną popularnością potwierdzają poniższe wykresy stworzone na podstawie statystyk prowadzonych przez Tatrzański Park Narodowy [tpnstat]. Turyści najczęściej wędrują po Tatrach w lecie, gdy warunki atmosferyczne pozwalają na ko-rzystanie z wyznaczonych szlaków bez konieczności zaopatrzenia się w profesjonalny sprzęt wspinaczkowy, lecz zimą to pasmo górkie także cieszy się popularnością podróżników 2.1. Wykres 2.2 pokazuje stale rosnące zainteresowanie turystyką górką w tym regionie.

`includegraphics[scale=1]img/wykresy/sprzedaż 2023.jpg`

Rysunek 2.1: Wykres przedstawiający liczbę sprzedanych biletów wstępu na teren Tatrzańskiego Parku Narodowego w poszczególnych miesiącach 2023 roku.

`includegraphics[scale=1]img/wykresy/sprzedaż ogółem.jpg`

Rysunek 2.2: Wykres przedstawiający liczbę biletów wstępu na teren Tatrzańskiego Parku Narodowego w latach 2013 - 2023 sprzedanych łącznie stacjonarnie i internetowo.

Niniejsza praca opisuje rozwiązywanie problemu nawigowania po szlakach górskich w polskiej części Tatr. Harnasie to aplikacja mobilna, która ułatwia poruszanie się po górach, z gwarancją bezpieczeństwa podczas wędrówki. Aplikacja oprócz nawigowania po szlakach posiada funkcję zgłoszenia zagrożenia występującego na danym szlaku. Użytkownik może powiadomić innych turystów o niebezpieczeństwie, jak również otrzymać informację o zagrożeniu na trasie, na której aktualnie się znajduje. Baza danych wykorzystywana w aplikacji zawiera informacje o użytkownikach korzystających z tej aplikacji, szlakach oraz powiadomieniach opisujących zagrożenie na trasie. Potrzeba stworzenia takiej aplikacji jest duża. Harnasie dają gwarancję bezpieczeństwa oraz pewność, iż każdy, kto będzie z niej korzystał, nie zabłędzi i zostanie powiadomiony o wszystkich występujących zagrożeniach.

W kolejnych rozdziałach opisane zostały najważniejsze elementy mające wpływ na jakość tworzonej aplikacji. Praca zawiera również opis użytych technologii oraz wszystkich funkcji, które posiada aplikacja. Dzięki temu czytelnik może zapoznać się z budową aplikacji oraz zasadami jej działania. Niektóre funkcje aplikacji zostały zaprogramowane na podstawie znajomości działania popularnych aplikacji, obsługujących śledzenie trasy przy wykorzystaniu modułu GPS. Wiedza ta znacznie ułatwiła rozwiązywanie pewnych problemów dotyczących wyznaczania trasy oraz modyfikowania jej, gdy system znajdzie lepszą drogę.

Zwieńczeniem pracy stanowi rozdział 9. Prezentuje on wnioski końcowe na temat stworzonej aplikacji.

3 Cel i zakres pracy

Celem pracy jest wykonanie projektu i implementacja w języku Java aplikacji mobilnej na urządzenia z systemem Android wspomagającej jednodniowe górskie wędrówki w Tatrach Polskich. Aplikacja wykorzystuje moduł GPS telefonu, mapy Google oraz usługi oferowane przez Firebase.

Zakres działań obejmuje:

- przegląd rynku w poszukiwaniu aplikacji oferujących mobilne mapy rejonu Karpat
- sformułowanie wymagań funkcjonalnych oraz niefunkcjonalnych,
- stworzenie diagramów przypadków użycia oraz scenariuszy,
- utworzenie konceptualnego diagramu klas,
- wykonanie projektu modelu danych,
- wybór narzędzi projektowych i programistycznych,
- projekt interfejsu użytkownika i administratora,
- implementację aplikacji,
- testy automatyczne i manualne

Zakres implementacji obejmuje:

- zaprojektowanie i implementacja bazy danych,
- umożliwienie planowania trasy,
- nawigację po wybranej trasie,
- stworzenie formularza do zgłoszenia zagrożeń i zapisywanie wysłanych zagrożeń,
- wysyłanie powiadomień ogólnosystemowych do wszystkich użytkowników odnośnie zagrożeń bądź wyłączenia szlaków z użycia i remontów,
- autoryzację i logowanie użytkowników,
- zapisywanie historii wędrówek i przedstawianie jej w formie wykresów lub raportów,
- dostęp z poziomu aplikacji do numerów alarmowych.

3.1 Podział pracy

- Streszczenie i wstęp - Katarzyna Jakubowska
- Cel i zakres pracy - wspólnie
- Analiza rynku - wspólnie
- Przegląd technologii - Katarzyna Jakubowska
- Projekt aplikacji - wspólnie
- Implementacja
 - Mapa - Magdalena Kamińska
 - * wyznaczanie trasy z punktami postojowymi (tworzenie url)
 - * obsługa markerów oznaczających schroniska i szczyty (widoczność na mapie)
 - * pobieranie punktów współrzędnych z plików .kml i rysowanie szlaków na mapie
 - * obsługa nawigacji po trasie za pomocą aplikacji Google Maps
 - * wybór szlaku z listy i pokazanie go na mapie
 - Firebase
 - * integracja funkcji oferowanych przez Firebase z aplikacją: bazy danych Firebase Firestore, Firebase Authentication - Katarzyna Jakubowska,
 - * zapis i odczyt zagrożeń z Firebase Firestore - Magdalena Kamińska,
 - * obsługa akceptacji zgłoszeń i ustawienie markera z zagrożeniem na mapie - Magdalena Kamińska,
 - * utworzenie wykresu aktywności użytkownika na podstawie danych pobranych z Firebase - Magdalena Kamińska,
 - * pobieranie z Firebase Storage i zapis lokalny na urządzeniu plików .kml ze szlakami i szczytami - Magdalena Kamińska,
 - * logowanie i autoryzacja użytkowników z podziałem na role - Katarzyna Jakubowska,
 - * zapis aktywności użytkownika do bazy danych Firestore - Katarzyna Jakubowska
 - Powiadomienia o zagrożeniach
 - * wysyłanie powiadomień o zagrożeniach lokalnych do użytkowników znajdujących się w pobliżu - Katarzyna Jakubowska

- * wysyłanie powiadomień ogólnosystemowych do wszystkich użytkowników z treścią komunikatów TOPR, TPN oraz GOPR o zagrożeniach występujących w Tatrach - Katarzyna Jakubowska
- Interfejsy aplikacji
 - interfejs administratora - Katarzyna Jakubowska,
 - interfejs użytkownika - Magdalena Kamińska
- Testy manualne i jednostkowe - Katarzyna Jakubowska
- Wnioski - wspólnie

Implementacja - opisy

- Moduł map - Magdalena Kamińska
- Moduł powiadomień - Katarzyna Jakubowska
- Moduł zgłaszania zagrożeń
- Interfejs użytkownika - Magdalena Kamińska
- Interfejs administratora - Katarzyna Jakubowska

4 Analiza porównawcza cech wybranych aplikacji mobilnych umożliwiających korzystanie z map

Dogłębna analiza rynku pozwoliła nam poznać istniejące już aplikacje mobilne umożliwiające nawigowanie po szlakach górskich oraz różne ich cechy, zarówno w wersji darmowej, jak i płatnej, premium. Poniższe porównanie dwóch aplikacji pozwoliło na podjęcie decyzji, jak opisywana w tej pracy aplikacja powinna wyglądać.

4.1 Platforma Mapa Turystyczna

Aplikacja Mapa Turystyczna [**mapatu**] jest to polska nakładka na Mapy Google. Wyznacza trasy tylko w ramach przebiegu szlaku wprowadzonego na mapy. Oprócz szlaków w Polsce, Czechach i Słowacji zaznaczone schroniska i niektóre atrakcje (np. jaskinie).

Rysunki 4.1 - 4.4 przedstawiają zrzuty ekranu z aplikacji mobilnej Mapa Turystyczna.

includegraphics[scale=0.15]img/rynek/mapatu-mapa.jpg

Rysunek 4.1: Mapa Turystyczna - widok po uruchomieniu aplikacji.

Aplikacja mobilna po uruchomieniu przenosi od razu użytkownika do widoku prezentującego mapę. Domyślnie mapa ma wybraną warstwę zawierającą kolorowe szlaki turystyczne, jak i ścieżki rowerowe, zaznaczone na mapie bary, jadłodajnie, wodopoje oraz zabytki i inne miejsca warte zwiedzenia.

includegraphics[scale=0.15]img/rynek/mapatu-menu.jpg

Rysunek 4.2: Mapa Turystyczna - menu z dostępnymi opcjami (użytkownik niezalogowany).

Dostępne rozwijane menu z dodatkowymi opcjami. Opcja planowania trasy dostępna jest z wersji desktopowej aplikacji, a tak stworzoną mapę można zapisać na profilu jako użytkownik zalogowany i korzystać z niej na urządzeniu mobilnym. Przebyte trasy użytkownik może także nagrywać lub importować do aplikacji jako plik GPX w aplikacji webowej. Konieczne do tej funkcji jest założenie konta i zalogowanie użytkownika.

includegraphics[scale=0.15]img/rynek/mapatu-warstwy.jpg

Rysunek 4.3: Mapa Turystyczna - dostępne warstwy mapy do wyboru.

Na powyższym widoku przedstawione są dostępne warstwy mapy: satelita, mapa terenowa z zaznaczonymi terenami zielonymi oraz szlakami i żłobieniami w terenie oraz zwykła mapa. Pozwala to użytkownikowi na kontrolę tego, jakie informacje wyświetla jego aplikacja.

includegraphics[scale=0.15]img/rynek/mapatu-premium.jpg

Rysunek 4.4: Mapa Turystyczna - dostępne funkcje po zakupie wersji premium.

W wersji premium aplikacja mobilna oferuje nam wersję aplikacji bez uciążliwych reklam, nawigację po trasie, nakładanie własnych tras na mapę a także pobieranie map i korzystanie z nich w trybie offline.

Wady

- Pełna wersja aplikacji płatna (miesiąc 8zł, rok – 35zł).
- W darmowej wersji mapy nie działają bez sieci komórkowej.
- Nie można dodać własnych map.
- Gotowe mapy nie obejmują terytorium Ukrainy.
- Wersja darmowa zawiera dużo reklam.
- Brak powiadomień o zagrożeniach.
- Nie tworzy trasy, użytkownik idzie po mapie, bez nawigacji.

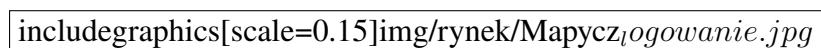
Zalety

- Dostępny darmowy tydzień testowy, dzięki któremu można pobrać mapy offline.
- Prosta obsługa.
- Oprócz trasy w wyniku planowania otrzymujemy: profil, czas przejścia, informację o punktach GOT.
- Aplikacja mobilna i wersja desktopowa.
- Historia tras oraz możliwość planowania podróży z wyprzedzeniem.

4.2 Platforma Mapy.cz

Aplikacja Mapy.cz [[mapycz](#)] rozwijana w wielu językach, darmowa, bez konieczności logowania. Po ówczesnej rejestracji użytkownik ma możliwość zapisywania tras i dodawania zdjęć obiektów spotkanych na swojej drodze. Oprócz wyszukiwania tras aplikacja oferuje także wyszukiwanie atrakcji, nawigację i mapy offline. Podczas planowania trasy istnieje możliwość jej zapisania i udostępniania, a także dodanie do niej interesujących obiektów. Bez aplikacji mobilnej istnieje możliwość odczytania współrzędnych z mapy webowej.

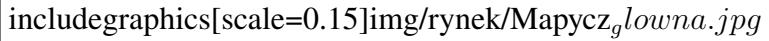
Rysunki 4.5 - 4.8 przedstawiają zrzuty ekranu z aplikacji mobilnej Mapy.cz.



Rysunek 4.5: Mapy.cz - funkcje dostępne po założeniu konta.

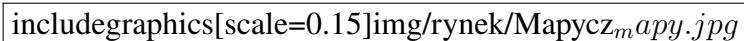
Aplikacja po uruchomieniu po raz pierwszy wita użytkownika powyższym widokiem. Przedstawione są na nim dodatkowe funkcje dostępne dopiero po zalogowaniu do aplikacji.

Te funkcje to m.in. planowanie i zapisywanie trasy oraz ciekawych miejsc, dodawanie zdjęć ze szlaku, ocenianie odwiedzonych miejsc oraz import tras i miejsc w formacie GPX.



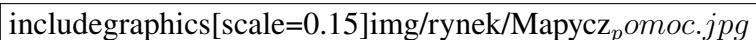
Rysunek 4.6: Mapy.cz - widok główny.

Kolejnym widokiem jest ekran główny, którego dominującym elementem jest skalowalna mapa, na której oprócz szlaków zaznaczone są także atrakcje turystyczne. Na górze ekranu ukazana jest wyszukiwarka miejsc na mapie. W lewym dolnym rogu widoczne są trzy przyciski: zmiana typu mapy (rys. 4.7), lokalizacja oraz rozwijane menu. W dostępnym po kliknięciu w ikonę menu znajdują się takie funkcje, jak: dostęp do konta, moje mapy, planuj trasę, wycieczka po okolicy, mapy offline, udostępnij lokalizację, uruchom rejestrator i ustawienia, a także skróty do mediów społecznościowych twórców i pierwsza pomoc (rys. 4.8).



Rysunek 4.7: Mapy.cz - dostępne typy map.

Na rysunku 4.7 widać dostępne typy map oferowane przez aplikację. Domyślnie typ mapy ustawiony jest na "turystyczna" z zaznaczonymi szlakami górskimi, lecz do wyboru użytkownik ma także: lotniczą, zimową, podstawową oraz warunki ruchu. W zależności od potrzeb użytkownika może on swobodnie wybierać różne tryby mapy.



Rysunek 4.8: Mapy.cz - widok zawierający wskazówki dotyczące pierwszej pomocy.

Powyższy widok zawiera wiele przydatnych opcji w razie wypadku na szlaku. Na samej górze opcja szukania najbliższego defibrylatora AED, możliwość nawigacji do punktu, w którym się znajduje. Poniżej znajdują się przyciski do bezpośredniego dzwonienia na numer 112 i przycisk do szukania pomocy lokalnie podpowiadający najbliższe szpitale i centra pomocy z opcją naprowadzania do nich. Większą część tego widoku zajmują przyciski z opisanymi możliwymi urazami, po kliknięciu których użytkownik zostaje przeniesiony do widoku odpowiadającemu danemu urazowi z instrukcjami, również graficznymi, jak postąpić w danej sytuacji.

Wady

- Wyszukiwanie w polu tekstowym nie do końca dopracowane.
- Słaba nawigacja po aplikacji - wybór jednej opcji niweluje możliwość powrotu do ostatniego miejsca; jeżeli użytkownik po wybraniu którejś opcji przejdzie do ekranu głównego, to musi on ponownie szukać poprzednio przeglądanego ekranu.

Zalety

- Łatwa w obsłudze, przypomina Mapy Google.
- Pobieranie map offline (region lub cały kraj).
- Plynność działania.
- Łatwe planowanie trasy z dużą liczbą punktów pośrednich.
- Możliwość rejestrowania trasy.
- W pełni darmowa aplikacja mobilna i webowa.

5 Przegląd narzędzi implementacyjnych

Dla prawidłowej i jak najskuteczniejszej pracy przygotowywanej aplikacji najważniejsze jest w procesie jej projektowania, aby dobrać optymalne narzędzia i technologie. W tym rozdziale skupiono się na krótkim omówieniu popularnych technologii programistycznych, które posiadają najbardziej potrzebne funkcje do prawidłowego działania systemu.

Usługi Firebase

Dla prostej skalowalności całego projektu, a także usprawnienie połączeń między bazą danych a aplikacją oraz względne pozbycie się serwera aplikacji wielu programistów decyduje się na korzystanie z usług oferowanych przez Firebase [[firebase](#)]. Do obsługi bazy danych najczęściej wybieraną usługą jest Firebase Firestore [[firebase-book](#)]. Jest to szybka i elastyczna baza danych NoSQL, która działa w infrastrukturze Google Cloud i synchronizuje dane pomiędzy różnymi platformami.

Do usprawnienia logowania i autoryzacji w aplikacji często wykorzystuje się usługę Firebase Authentication [[fireauth](#)] pozwalającą na sprawdzanie tożsamości użytkowników i personalizowanie treści wyświetlanego na urządzeniach klientów. Authentication obsługuje uwierzytelnianie przy użyciu haseł, numerów telefonów, a także kont na platformach mediów społecznościowych, takich jak Google, Facebook, Twitter i innych kanałów.

Przechowywanie plików w chmurze często jest potrzebne do sprawnej synchronizacji danych pomiędzy urządzeniami, kontami, jak i platformami. Firebase również tym razem wychodzi programistom na przeciwnie, udostępniając usługę Storage. Jest to wydajna, prosta w obsłudze i niedroga usługa pamięci masowej opracowana z myślą o skali Google.

Google Maps Platform

Po dogłębnym przeszukaniu rynku oferującego wiele API obsługujących mapy na platformę Android, Google Maps Platform okazała się najbardziej odpowiednia. Jest to jedno z wielu usprawnień oferowanych przez Google przez dostęp do ich Google Clouds Platform. Dzięki temu wszelkie inne API z kategorii takich jak: obliczenia, przechowywanie i bazy danych, analiza danych, sztuczna inteligencja, sieci itp. są ze sobą zsynchronizowane, a do obsługiwanego ich wszystkich wystarczy jedno konto, które przez pierwsze 90 dni jest darmowe i posiada 300USD przypisanych do portfela usługi. Po tym czasie użytkownik może zmienić typ swojego konta na płatne, wtedy środki pobierane są bezpośrednio z karty użytkownika za każde ponadplanowe użycie zasobów. Aspekt płatności, choć może się wydawać odstraszający dla wielu, nie jest skomplikowany i pozwala na pełny darmowy rozwój aplikacji, jednak przy komercjalizacji projektu darmowe zasoby mogą okazać się niewystarczające.

Android Studio

Najpopularniejsze IDE pozwalające na tworzenie aplikacji mobilnych na urządzenia z systemem Android [[androidstudio](#)] [[android-studio-book](#)]. Obsługuje język Java oraz Kotlin i zawiera wiele wbudowanych narzędzi ułatwiających tworzenie aplikacji na urządzenia mobilne. Posiada m.in. emulatory urządzeń, integrację z systemem kontroli wersji Git oraz narzędzia do profilowania wydajności aplikacji.

Język Java

Java[[java](#)] jest to wysokopoziomowy, obiektowy język programowania. Jego szeroki zakres użytku pozwala na tworzenie różnych aplikacji w zależności od potrzeby, m.in. aplikacji webowych, systemów biznesowych, aplikacji mobilnych (Android), a także można go wykorzystać w systemach wbudowanych i przetwarzaniu danych [[firebase-book](#)]. Dlaczego Java?

- Obiektywość. Napisany kod można wykorzystywać wielokrotnie i dzielić go na moduły.
- Ogromny zasób gotowych bibliotek, które obsługują wszelakie funkcje.
- Aplikacje tworzone w Javie można uruchamiać na wszelkiego typu urządzeniach dzięki maszynie wirtualnej (JVM).

[[gmapsand](#)] [[gmapsogol](#)] [[java-book](#)]

6 Projekt aplikacji

6.1 Wymagania funkcjonalne

Administrator

1. Zarządzanie trasami dostępnymi w systemie
 - (a) Dodawanie nowych tras (jeśli nowe zostały wyznaczone).
 - (b) Rozbudowa bazy tras o nowe regiony.
 - (c) Aktualizacja istniejących w systemie szlaków i ich opisów.
 - (d) Usuwanie szlaków, które są tymczasowo wyłączone z użycia przez TPN (np. w celu modernizacji).
2. Obsługa powiadomień
 - (a) Wysyłanie powiadomień ogólnosystemowych do wszystkich użytkowników.
 - (b) Przekazywanie komunikatów wychodzących odgórnie o możliwym zagrożeniu (np. zagrożenie lawinowe, burze ogłoszane przez TPN/GOPR).
 - (c) Wysyłanie powiadomień do użytkowników w obrębie konkretnego regionu o miejscowościowym zagrożeniu bądź utrudnieniach.
 - (d) Akceptacja powiadomień przesyłanych przez użytkowników o zgłoszonych incydentach na danej trasie.

Wędrowkowicz

1. Planowanie trasy
 - (a) Zaznaczanie na mapie wybranego szlaku.
 - (b) Pobieranie map na urządzenie celem korzystania z aplikacji w trybie offline.
 - (c) Dodawanie pobocznych, wyznaczonych w aplikacji szlaków do głównej trasy.
 - (d) Edytowanie trasy w dowolnym momencie.
 - (e) Dodawanie punktów postojowych
2. Pomoc
 - (a) Dostęp do numerów alarmowych i lokalnych numerów ratunkowych (GOPR itp.)
 - (b) Wybieranie numeru z poziomu aplikacji i dzwonienie bez konieczności kopiowania numeru.
3. Zgłaszanie zagrożeń
 - (a) Wybieranie zagrożeń z dostępnych domyślnych opcji.

- (b) Zgłaszczenie innego typu zagrożenia z pełnym jego opisem.
 - (c) Wybranie możliwego regionu, na którym dane zagrożenie może występować.
 - (d) Wypełnienie zgłoszenia możliwe w trybie offline, wysłanie zgłoszenia do administratora tylko po połączeniu z internetem.
4. Przegląd historii wędrówek
 - (a) Generowanie raportu z wybranego okresu.
 - (b) Filtrowanie przebytych tras.
 5. Nawigacja
 - (a) Korzystanie z map w trybie offline.
 - (b) Naprowadzanie na szlak w wypadku zejścia z niego.
 - (c) Otrzymywanie powiadomień o zagrożeniach na podstawie obecnej lokalizacji (tryb online)
 6. Krokomierz
 - (a) Obliczanie spalonych kalorii na podstawie aktywności zarejestrowanej przez krokomierz.
 - (b) Generowanie podsumowań jednodniowych, tygodniowych lub z wybranego okresu.
 7. Logowanie
 - (a) Zakładanie konta.
 - (b) Usuwanie konta.
 - (c) Edytowanie danych.

6.2 Wymagania niefunkcjonalne

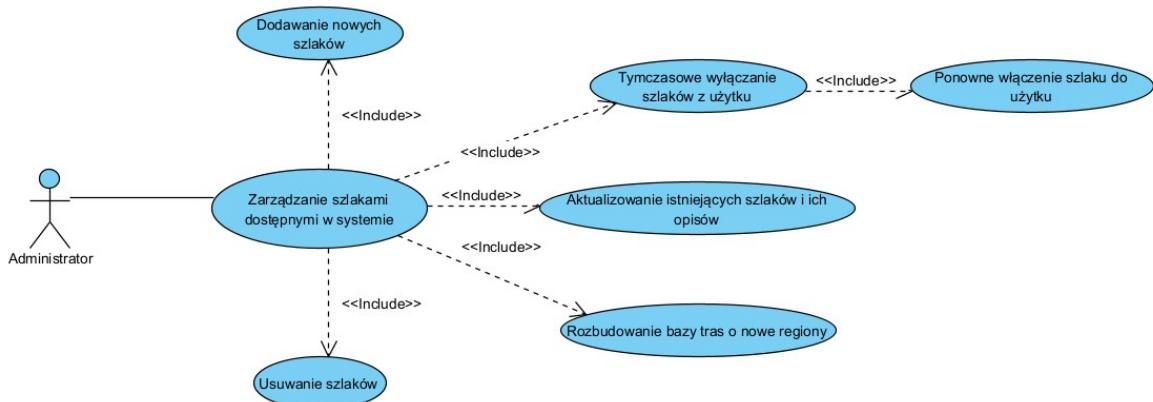
1. System zbudowany na urządzenia mobilne z systemem Android w wersji 7.0 lub wyższej.
2. System wykorzystuje moduł GPS celem nawigacji po wyznaczonych szlakach.
3. System wykorzystuje krokomierz (jeśli urządzenie takowy posiada) celem liczenia kroków i spalonych przez użytkownika kalorii.
4. System powinien zapewniać bezpieczeństwo danych użytkowników, zwłaszcza szczególnie wrażliwych informacji takich jak dane osobowe.
5. System powinien posiadać zrozumiały interfejs użytkownika w celu łatwego korzystania z niego.

6. System powinien umożliwiać monitorowanie jego działania w celu ułatwienia rozwiązywania problemów.
7. Hasła użytkowników powinny być szyfrowane.

6.3 Diagramy przypadków użycia

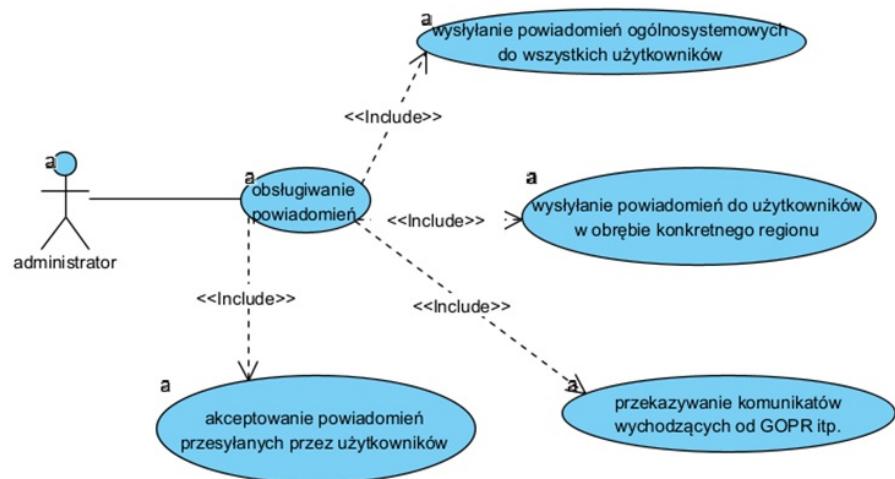
Diagramy przypadków użycia pozwalają w prosty sposób zwizualizować dostępne dla poszczególnych ról funkcje oraz ich przebiegi. Pozwalają także na uwzględnienie wyjątków i ewentualnych problemów, jakie użytkownik może napotkać podczas korzystania z systemu.

Poniższe diagramy prezentują poszczególne funkcje aplikacji oferowane poszczególnym rolem użytkowników opracowane na podstawie określonych wcześniej wymagań funkcjonalnych.



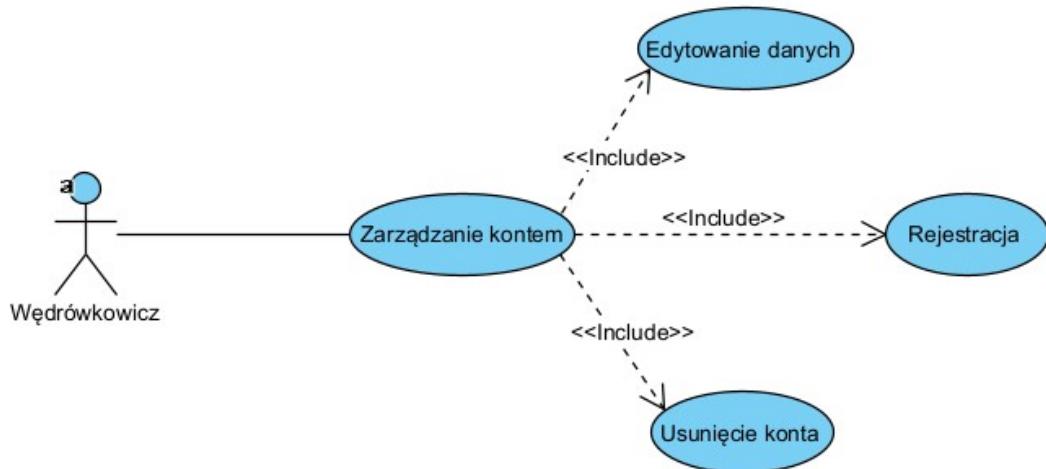
Rysunek 6.1: Diagram przypadku użycia: zarządzanie szlakami przez administratora.

Funkcja zarządzania szlakami przez administratora (rys. 6.1) składa się z kilku pomocniczych funkcji. Są to między innymi: tymczasowe wyłączenie z użytku (np. na potrzebę modernizacji szlaku bądź usunięcia z niego powalonego drzewa), ponowne jego włączenie do użytku, dodawanie nowych szlaków, usuwanie szlaków, aktualizacja przebiegu istniejących szlaków oraz rozbudowanie bazy tras o kolejne pasma górskie.



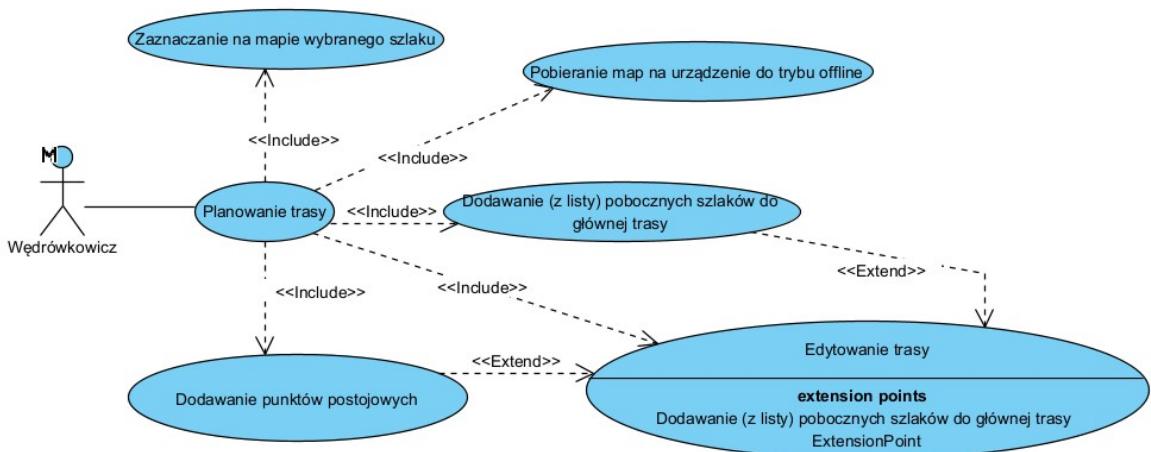
Rysunek 6.2: Diagram przypadku użycia: obsługa powiadomień po stronie administratora.

Funkcja obsługi powiadomień (rys. 6.2) dostępna dla administratora pozwala na wysyłanie powiadomień od użytkowników w obrębie konkretnego regionu, wysyłanie powiadomień ogólnosystemowych do wszystkich użytkowników, przekazywanie komunikatów wychodzących od GOPR oraz akceptowanie zgłoszonych zagrożeń przesyłanych przez użytkowników.



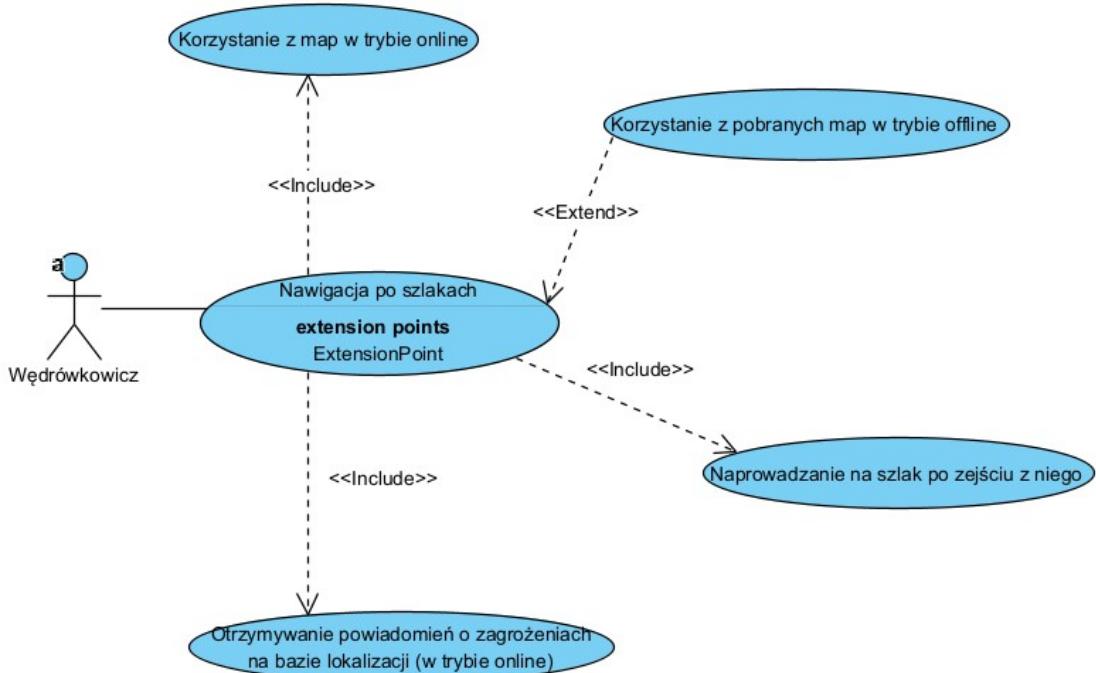
Rysunek 6.3: Diagram przypadku użycia: zarządzanie kontem.

Zarządzanie kontem (rys. 6.3) leży w całości po stronie użytkownika - wędrówkowicza. Powierzamy mu takie funkcje jak: rejestracja, edytowanie danych przypisanych do konta oraz usunięcie konta.



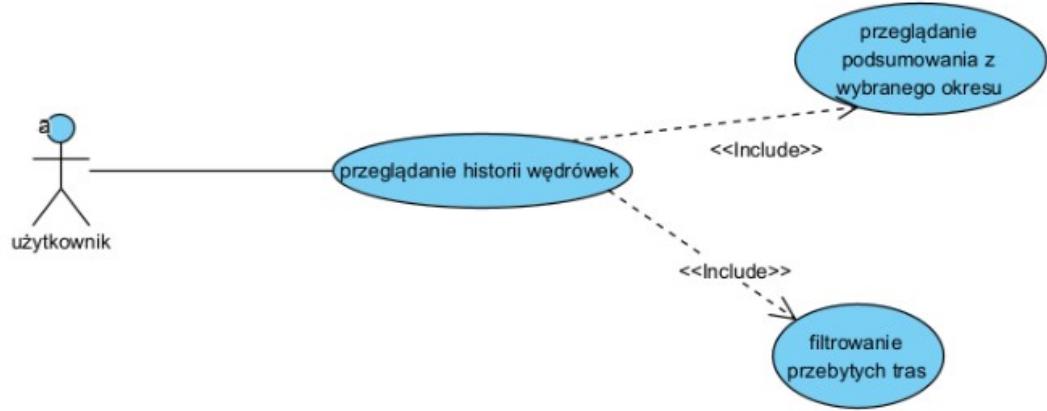
Rysunek 6.4: Diagram przypadku użycia: planowanie trasy przez użytkownika - wędrowkowicza

Planowanie trasy (rys. 6.4) jest procesem umożliwiającym podróżującemu na wybranie proponowanych mu szlaków, aby przejść z jednego miejsca w drugie. Na ten proces składają się funkcje takie jak: zaznaczanie na mapie wybranego szlaku, dodawanie z listy przylegających szlaków do głównej trasy, dodawanie punktów postojowych,



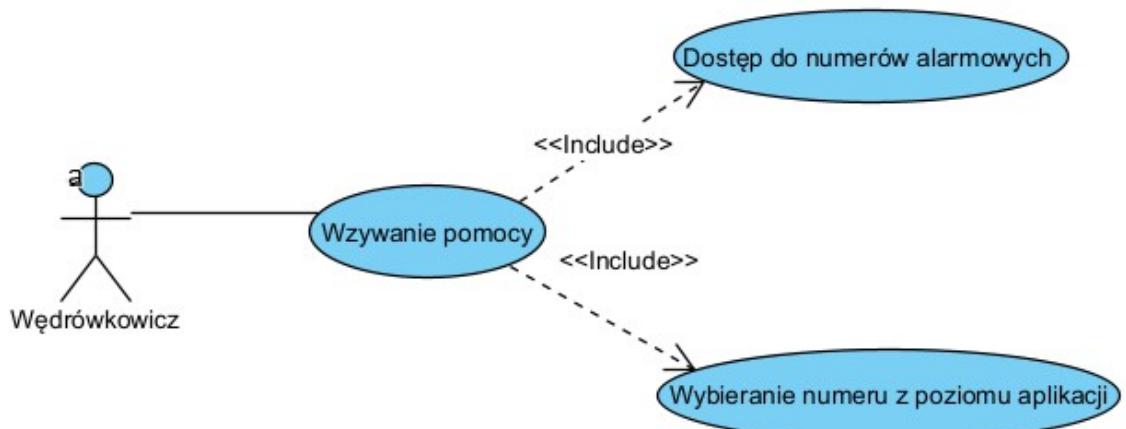
Rysunek 6.5: Diagram przypadku użycia: nawigacja

Nawigacja po szlakach górskich (rys. 6.5) jest głównym zadaniem aplikacji. Poprawne jej funkcjonowanie gwarantują takie elementy, jak: korzystanie z map w trybie online, otrzymywanie powiadomień o zagrożeniach bądź utrudnieniach na trasie, pobieranie map i korzystanie z nich w trybie offline oraz naprowadzanie na szlak po zejściu z niego.



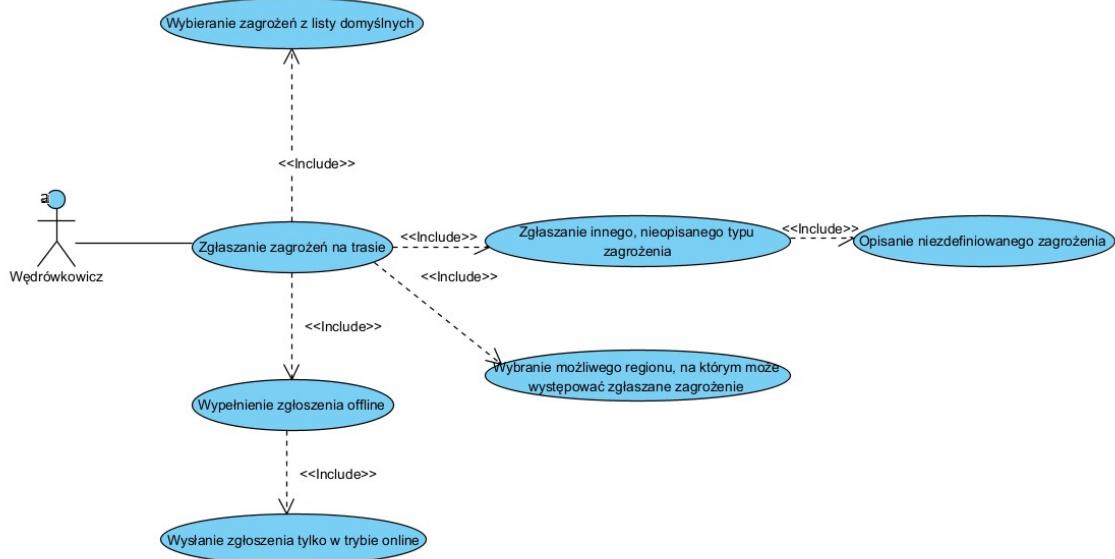
Rysunek 6.6: Diagram przypadku użycia: historia przebytych tras.

Przeglądanie historii przebytych tras przez danego użytkownika (rys. 6.6) pozwala na kontrolowanie przebiegu swoich wędrówek. Dodatkową opcją tej funkcji jest generowanie raportów za ustalony okres, co ułatwia szybki powrót do swoich wypraw.



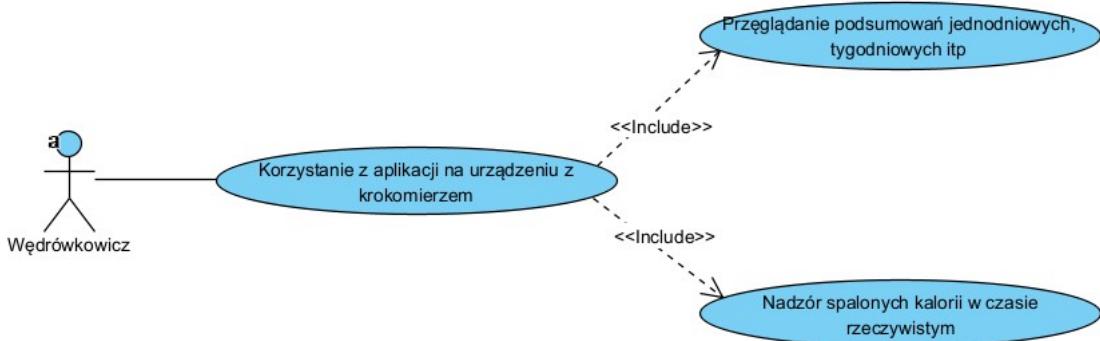
Rysunek 6.7: Diagram przypadku użycia: wzywanie pomocy.

Wzywanie pomocy na szlaku (rys. 6.7) jest niezwykle istotną funkcją. Nie wszyscy turyści znają numery alarmowe, w tym numer GOPRu, często przed wyjściem na szlak zapominają ich zapisać. W związku z tym udostępniamy im opcję skopiowania numerów oraz bezpośredniego wybierania ich z poziomu aplikacji.



Rysunek 6.8: Diagram przypadku użycia: zgłasza zagrożeń.

Zgłasza zagrożeń oraz utrudnień występujących na szlaku (rys. 6.8) pomaga wszystkim turystom bezpiecznie przejść całą zaplanowaną przez nich trasę. Występujące zagrożenie może zgłosić każdy z dostępem do internetu, lecz formularz zgłoszeniowy można wypełnić również w trybie offline. Zapisany i wypełniony formularz zostanie wysłany do zatwierdzenia, gdy urządzenie użytkownika uzyska dostęp do internetu. Zgłoszone zagrożenie posiada także przypisany region obowiązywania, tj. okolice lokalizacji użytkownika w momencie wypełnienia zgłoszenia. Użytkownik może wybrać typ zagrożenia z gotowej listy lub własnoręcznie opisać zdarzenie.



Rysunek 6.9: Diagram przypadku użycia: użycie krokomierza

6.4 Scenariusze

Scenariusze pozwalają rozpatrzyć dokładnie konkretne użycia danych funkcji oraz ich przebiegi alternatywne i sytuacje wyjątkowe. Poniższe wybrane scenariusze opisują kolejno:

zgłoszenie zagrożenia przez użytkownika, wyznaczanie trasy, wysyłanie przez administratora powiadomienia o zagrożeniu do użytkowników oraz wyłączanie szlaku z użycia przez administratora.

Scenariusz 1: Zgłaszcenie zagrożeń

S1.1 Opis

Scenariusz opisujący zgłaszczenie przez użytkownika zagrożeń występujących na danej trasie w celu poinformowania innych o utrudnieniach przez aplikację mobilną.

S1.2. Aktorzy

Klient/użytkownik

S1.3. Warunki początkowe

Użytkownik jest zalogowany na aplikację i ma wyznaczoną trasę do przejścia.

S1.4. Warunki końcowe

System przesyła ogłoszenie o zagrożeniu do weryfikacji administratorowi.

S1.5. Przebieg główny

1. Użytkownik wybiera opcję zgłoszenia zagrożenia.
2. System wyświetla listę z przykładowymi zagrożeniami.
3. Użytkownik wybiera rodzaj zagrożenia z listy.
4. System wyświetla zgłoszenie z obecną lokalizacją użytkownika.
5. Użytkownik zatwierdza zagrożenie.

S1.6. Przebieg alternatywny

PA 3.1 Użytkownik dodaje własny opis kategorii zagrożenia.

- (a) Użytkownik odpowiada na pytania w celu zidentyfikowania zagrożenia.
- (b) Powrót do punktu 4 przebiegu głównego.

PA 5.1 Użytkownik edytuje zgłoszenie.

- (a) Użytkownik odpowiada na pytania w celu zidentyfikowania zagrożenia.
- (b) Powrót do punktu 4 przebiegu głównego.

S1.7. Sytuacje wyjątkowe

SW 1. System nie może zlokalizować użytkownika.

- (a) System wyświetla komunikat o błędzie.

S1.8. Wymagania niefunkcjonalne

Takie same jak do systemu ułatwiającego poruszanie się po górach

S1.9. Uwagi i pytania otwarte

Brak

Scenariusz 2: Wyznaczanie trasy

S2.1 Opis

Scenariusz opisujący wyznaczenie najlepszej trasy dla użytkownika.

S2.2 Aktorzy

Klient/użytkownik

S2.3 Warunki początkowe

Użytkownik jest zalogowany do aplikacji.

S2.4 Warunki końcowe

System wyświetla wybraną przez użytkownika trasę zaznaczoną na mapie.

S2.5 Przebieg główny

1. Użytkownik wybiera opcję wyznaczania trasy.
2. Użytkownik podaje cel wędrówki.
3. System pokazuje dostępne szlaki.
4. Użytkownik wybiera najlepszą dla niego trasę.
5. System wyświetla informacje o dystansie oraz planowanym czasie wędrówki.
6. Użytkownik zatwierdza wybraną trasę.

S2.6 Przebieg alternatywny

PA 6.1 Użytkownik modyfikuje trasę, dodając przystanki na trasie.

- (a) Użytkownik dodaje nowy cel na trasie.
- (b) Powrót do punktu 5 przebiegu głównego.

S2.7 Sytuacje wyjątkowe

SW 1. System nie może znaleźć dostępnej trasy dla użytkownika.

- (a) System wyświetla komunikat o błędzie.

S2.8. Wymagania niefunkcjonalne

Takie same jak do systemu ułatwiającego poruszanie się po górach.

S2.9. Uwagi i pytania otwarte

Brak

Scenariusz 3: Wysyłanie powiadomień do użytkowników w obrębie konkretnego regionu o miejscowym zagrożeniu bądź utrudnieniach.

S3.1. Opis

Scenariusz opisujący wysyłanie przez administratora powiadomień do użytkowników znajdujących się w obrębie danego regionu o miejscowym zagrożeniu.

S3.2. Aktorzy

Administrator

S3.3. Warunki początkowe

Administrator jest zalogowany na aplikację i jest połączony z internetem.

S3.4. Warunki końcowe

System wysyła powiadomienie o zagrożeniu do wszystkich użytkowników, którzy znajdują się w podanym przez administratora regionie.

S3.5. Przebieg główny

1. Administrator wybiera opcję wysłania powiadomienia do konkretnego regionu.
2. System wyświetla listę z gotowymi, przykładowymi powiadomieniami.
3. Administrator wybiera szablon i uzupełnia go o niezbędne informacje.
4. Administrator wybiera region, w którym może występować zagrożenie.
5. System wyświetla pełne powiadomienie ze wszystkimi szczegółami i czeka na potwierdzenie.
6. Administrator zatwierdza powiadomienie.

S3.6. Przebieg alternatywny

PA 3.1 Administrator pisze ręcznie całe powiadomienie.

1. System wyświetla pusty formularz.
2. Administrator wpisuje treść powiadomienia ręcznie uwzględniając w nim wszystkie szczegóły.
3. Powrót do punktu 4 przebiegu głównego.

S3.7. Sytuacje wyjątkowe

SW 1. Administrator traci połączenie z internetem w trakcie tworzenia treści powiadomienia.

1. System wyświetla informację o przełączeniu w tryb offline.
2. System zapisuje wprowadzone do tej pory zmiany tak, aby po ponownym połączeniu z internetem powiadomienie było gotowe do wysłania.

S3.8. Wymagania niefunkcjonalne

Takie same jak do systemu ułatwiającego poruszanie się po górach.

S3.9. Uwagi i pytania otwarte

Brak

Scenariusz 4: Tymczasowe wyłączanie szlaków z użycia

S4.1.Opis

Scenariusz opisujący tymczasowe wyłączanie szlaków z użycia przez administratora ze względu na prowadzone na nim prace konserwacyjne bądź inne, długotrwałe trudnienia.

S4.2. Aktorzy

Administrator

S4.3. Warunki początkowe

Administrator jest zalogowany na aplikację i jest połączony z internetem.

S4.4. Warunki końcowe

System wyszarza dany szlak na wszystkich mapach online i wyświetla komunikat o jego zamknięciu, gdy użytkownik chce go wybrać.

S4.5. Przebieg główny

1. Administrator wybiera szlak, który chce wyłączyć z użycia.
2. System wyświetla krótki formularz.
3. Administrator podaje do formularza komunikat, dlaczego szlak jest zamknięty i przewidywaną datę jego ponownego otwarcia.
4. System wyświetla gotowy do zatwierdzenia komunikat o zamknięciu szlaku.
5. Administrator zatwierdza komunikat.

S4.6. Przebieg alternatywny

PA 3.1 Administrator wyłącza jedynie część szlaku z użycia.

1. Administrator wybiera punkt, od którego szlak ma być wyłączony.
2. System wyświetla inne, niezamknięte szlaki niedaleko tego, który obecnie wyłącza administrator.
3. Administrator ustala obejście zamkniętej części szlaku.
4. Powrót do punktu 2 przebiegu głównego.

S4.7. Sytuacje wyjątkowe

SW 1. Administrator traci połączenie z internetem w trakcie tworzenia treści powiadomienia.

1. System wyświetla informację o przełączeniu w tryb offline.
2. System zapisuje wprowadzone do tej pory zmiany tak, aby po ponownym połączeniu z internetem powiadomienie było gotowe do wysłania.

S4.8. Wymagania niefunkcjonalne

Takie same jak do systemu ułatwiającego poruszanie się po górach.

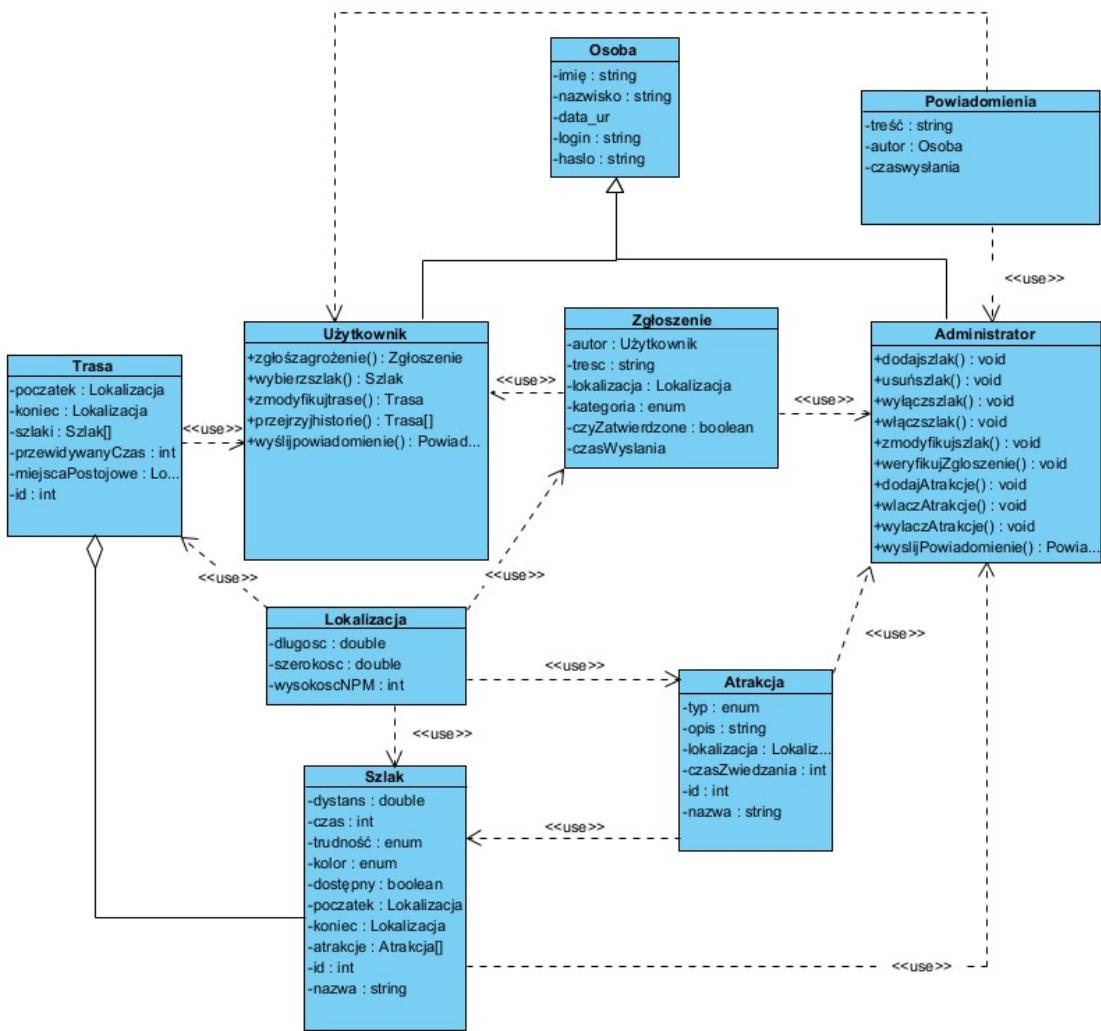
S4.9. Uwagi i pytania otwarte

Brak

6.5 Diagram klas

Diagram klas jest świetnym sposobem na przedstawienie struktury systemu w modelach obiektowych przez ilustrację struktury klas i zależności między nimi. Ukazuje on klasy (typy) obiektów w programie, w odróżnieniu od diagramu obiektów, który pokazuje jedynie egzemplarze (instancje) obiektów i ich zależności istniejące w konkretnym momencie.

Poniższy diagram klas (rys. 6.10) opisuje strukturę systemu aplikacji wspomagającej turystykę górską. Składa się z następujących klas: Osoba, Użytkownik, Administrator, Zgłoszenie, Powiadomienie, Lokalizacja, Atrakcja, Szlak. Klasy te są ze sobą powiązane asocjacjami i agregacjami oraz korzystają z siebie nawzajem.



Rysunek 6.10: Diagram klas

Opis klas

Klasa Osoba:

Atrybuty:

- *imie*: string - przechowuje imię danej osoby,
- *nazwisko*: string - przechowuje nazwisko danej osoby,
- *data_ur* - przechowuje datę urodzenia danej osoby,
- *login*: string - przechowuje login użytkownika będący adresem e-mail danej osoby,
- *haslo*: string - przechowuje szyfrowane hasło przypisane do konta danej osoby.

Klasa Uzytkownik

Klasa ta dziedziczy po klasie Osoba przyjmując wszystkie jej atrybuty.

Funkcje:

- *zgłośzagrożenie()* - zwraca nowo utworzony obiekt typu Zagrożenie z jego wszystkimi atrybutami,
- *wybierzszlak()* - zwraca wybrany obiekt typu Szlak,
- *zmodyfikujtrase()* - zwraca zmodyfikowany przez użytkownika obiekt Trasa,
- *przejrzyjhistorie()* - zwraca listę Tras przebytych przez użytkownika,
- *wyślijpowiadomienie()* - po zgłoszeniu Zagrożenia i zaakceptowaniu go przez Administratora wysyła Powiadomienie.

Klasa Administrator

Klasa ta dziedziczy po klasie Osoba przyjmując wszystkie jej atrybuty.

Funkcje:

- *dodajszlak()* - funkcja void, tworzy nowy szlak i dodaje go do bazy danych,
- *usuńszlak()* - funkcja void, usuwa wybrany szlak z bazy danych,
- *wyłączszlak()* - funkcja void, wyłącza tymczasowo wybrany Szlak z użycia,
- *włączszlak()* - funkcja void, włącza ponownie wybrany, uprzednio wyłączony z użycia Szlak,
- *zmodyfikujszlak()* - funkcja void, modyfikuje istniejący już Szlak według nowych wytycznych,
- *weryfikujZgłoszenie()* - funkcja void, pozwala na zweryfikowanie poprawności Zgłoszenia; po weryfikacji zamienia Zgłoszenie w Powiadomienie,
- *dodajAtrakcje()* - funkcja void, dodaje do bazy danych nową Atrakcję do bazy Atrakcji dostępnych dla turystów,
- *wylaczAtrakcje()* - funkcja void, wyłącza tymczasowo wybraną Atrakcję z użycia,
- *wlaczAtrakcje()* - funkcja void, włącza ponownie wybraną, uprzednio wyłączoną z użycia Atrakcję,
- *wyślijPowiadomienie()* - zwraca nowo utworzone Powiadomienie, które nie musi przechodzić procesu weryfikacji.

Klasa Zgłoszenie

Atrybuty:

- *autor*: Użytkownik - przechowuje autora zgłoszenia, którym jest Użytkownik,
- *tresc*: string - przechowuje treść opisu zgłoszenia,
- *lokalizacja*: Lokalizacja - przechowuje Lokalizację pobraną z modułu GPS użytkownika w momencie wypełniania Zgłoszenia lub Lokalizację wybraną z mapy,
- *kategoria*: enum - przechowuje wybraną z listy kategorię Zgłoszenia,

- *czyZatwierdzone*: boolean - przechowuje informację, czy Zgłoszenie zostało zatwierdzone przez Administratora,
- *czasWyslania* - przechowuje informację o czasie wysłania zgłoszenia celem sortowania ich przy wyświetaniu Administratorowi do zatwierdzenia (od najwcześniej zgłoszonego).

Klasa Powiadomienie

Atrybuty:

- *treść*: string - przechowuje treść zgłoszenia automatycznie wygenerowaną bądź zmodyfikowaną przez Administratora,
- *autor*: Osoba - przechowuje dane autora Powiadomienia, którym może być zarówno Użytkownik jak i Administrator,
- *czaswysłania* - przechowuje datę i czas wysłania Powiadomienia.

Klasa Lokalizacja

Atrybuty:

- *dlugosc*: double - przechowuje długość geograficzną,
- *szerokosc*: double - przechowuje szerokość geograficzną,
- *wysokoscNPM*: int - przechowuje wysokość punktu nad poziomem morza.

Klasa Szlak

Atrybuty:

- *dystans*: double - przechowuje długość danego Szlaku od jego początku do końca,
- *czas*: int - przechowuje przybliżony czas potrzebny na przejście danego Szlaku (w minutach),
- *trudność*: enum - przechowuje poziom trudności danego Szlaku,
- *kolor*: enum - przechowuje kolor danego Szlaku według wytycznych kartograficznych,
- *dostępny*: boolean - przechowuje informację o dostępności Szlaku (czy jest wyłączony z użycia lub czy turyści mogą się po nim poruszać),
- *początek*: Lokalizacja - przechowuje dane lokalizacyjne o początku Szlaku,
- *koniec*: Lokalizacja - przechowuje dane lokalizacyjne o końcu Szlaku,
- *atrakcje*: Atrakcja[] - przechowuje listę Atrakcji dostępnych na wybranym Szlaku,
- *id*: int - przechowuje identyfikator Szlaku,

- *nazwa*: string - przechowuje nazwę Szlaku.

Klasa Atrakcja

Atrybuty:

- *typ*: enum - przechowuje typ Atrakcji,
- *opis*: string - przechowuje krótki opis Atrakcji,
- *lokalizacja*: Lokalizacja - przechowuje Lokalizację danej Atrakcji,
- *czasZwiedzania*: int - przechowuje przybliżony czas zwiedzania Atrakcji (w minutach),
- *id*: int - przechowuje identyfikator Atrakcji,
- *nazwa*: string - przechowuje nazwę Atrakcji.

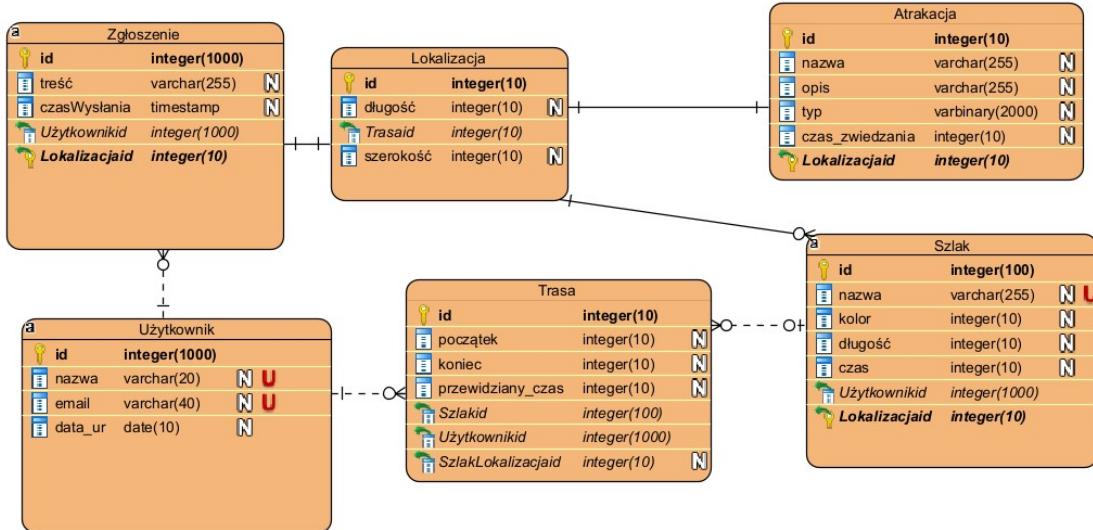
Klasa Trasa

Atrybuty:

- *początek*: Lokalizacja - przechowuje początek Trasy,
- *koniec*: Lokalizacja - przechowuje koniec Trasy,
- *szlaki*: Szlak[] - przechowuje listę Szlaków, którymi trzeba przejść, aby zrealizować daną Trasę,
- *przewidywanyCzas*: int - przechowuje przewidywany czas potrzebny na przejście danej Trasy od początku do końca,
- *miejscaPostojowe*: Lokalizacja[] - przechowuje listę zaplanowanych miejsc postojowych,
- *id*: int - przechowuje identyfikator Trasy.

6.6 Diagram ERD

Diagram ERD (Entity-Relationship Diagram) to graficzne przedstawienie modelu danych, które pokazuje związki między różnymi elementami systemu. ERD składa się z trzech głównych komponentów: encje - reprezentują obiekty lub pojęcia, które mają znaczenie w kontekście bazy danych, atrybuty - określają właściwości lub cechy encji oraz związki - pokazują, jak encje są ze sobą powiązane.



Rysunek 6.11: Diagram ERD.

Powyższy diagram prezentuje encje, atrybuty oraz powiązania istniejące w opisywanej aplikacji.

Encje i ich atrybuty

Użytkownik

- *id* typu integer jako klucz główny,
- *nazwa* typu varchar (unikalna, niepusta),
- *email* typu varchar (unikalny, niepusty),
- *data_ur* typu date (niepusta).

Trasa

- *id* typu integer jako klucz główny,
- *początek* typu integer (niepusty),
- *koniec* typu integer (niepusty),
- *przewidziany_czas* typu integer (niepusty),
- *Szlakid* typu integer,
- *Użytkownikid* typu integer,
- *SzlakLokalizacjaid* typu integer (niepusta).

Szlak

- *id* typu integer (klucz główny),
- *nazwa* typu varchar (unikalna, niepusta),

- *kolor* typu integer (niepusty),
- *długość* typu integer (niepusta),
- *czas* typu integer (niepusty),
- *Uzytkownikid* typu integer,
- *Lokalizacjaid* typu integer (klucz obcy).

Zgłoszenie

- *id* typu integer jako klucz główny,
- *treść* typu varchar (niepusta),
- *czasWysłania* typu timestamp (niepusty),
- *Uzytkownikid* typu integer,
- *Lokalizacjaid* typu integer (klucz obcy).

Lokalizacja

- *id* typu integer jako klucz główny,
- *długość* typu integer (niepusta),
- *Trasaid* typu integer,
- *szerokość* typu integer (niepusta).

Atrakcja

- *id* typu integer jako klucz główny,
- *nazwa* typu varchar (niepusta),
- *opis* typu varchar (niepusty),
- *typ* typu varbinary (niepusty),
- *czas_zwiedzania* typu integer (niepusty),
- *Lokalizacjaid* typu integer (klucz obcy),

Relacje

- Zgłoszenie - Lokalizacja: jedno zgłoszenie musi mieć jedną lokalizację, ale z jednej lokalizacji można wysłać kilka zgłoszeń
- Użytkownik - Zgłoszenie: użytkownik może, ale nie musi wysłać wielu zgłoszeń, ale jedno zgłoszenie może być wysłane tylko przez jednego użytkownika,
- Użytkownik - Trasa: użytkownik może, ale nie musi stworzyć wiele tras, ale jedna trasa może być stworzona tylko przez jednego użytkownika,

- Trasa - Szlak: jeden szlak może, ale nie musi być zawarty w wielu trasach, a jedna trasa może przebiegać przez tylko jeden szlak,
- Lokalizacja - Szlak: jeden szlak zbudowany jest z wielu lokalizacji, ale jedna lokalizacja może występować tylko na jednym szlaku,
- Lokalizacja - Atrakcja: w jednej lokalizacji może występować tylko jedna atrakcja, a jedna atrakcja ma jedną lokalizację.

6.7 Projekt interfejsu użytkownika

Interfejs użytkownika jest kluczowym elementem doświadczenia użytkownika, wpływającym na to, jak użytkownicy postrzegają i korzystają z aplikacji. Dobrze zaprojektowany interfejs sprawia, że aplikacja jest intuicyjna, łatwa w użyciu i atrakcyjna wizualnie, co zwiększa satysfakcję użytkowników i ich efektywność.

Poniżej ukazane są pogłądowe interfejsy użytkownika zarówno dla zwykłego podróżnika, jak i administratora.

Użytkownik

Użytkownik aplikacji ma dostęp do takich widoków jak: logowanie, rejestracja, widok główny zawierający mapę, profil użytkownika, historię przebytych tras, wybór trasy, telefony alarmowe oraz zgłoszenie zagrożenia.

Rejestracja



Rysunek 6.12: Interfejs użytkownika: rejestracja.

Widok na rys.6.12 zawiera formularz rejestracyjny, który pobiera od użytkownika następujące dane: imię, nazwisko, adres e-mail używany później jako login, hasło oraz datę urodzenia. Poniżej formularza znajduje się przycisk "Zarejestruj się", który dodaje użytkownika

do bazy danych. Kliknięcie przycisku powoduje wyświetlenie powiadomienia o utworzeniu konta oraz przejście na ekran główny aplikacji(rys. 6.14).

Logowanie



Rysunek 6.13: Interfejs użytkownika: logowanie.

Widok na rys.6.13 zawiera formularz logowania, który pobiera od użytkownika jego login oraz hasło. Poniżej pól do pobierania danych znajduje się przycisk "Zaloguj się", który przenosi użytkownika na ekran główny (rys. 6.14). Oprócz tego widoczny jest odnośnik do formularza rejestracji (rys. 6.12), jeśli użytkownik nie założył jeszcze konta w aplikacji.

Widok główny



Rysunek 6.14: Interfejs użytkownika: widok główny.

Widok na rys.6.14 zawiera interaktywną mapę, którą możemy przesuwać, przybliżać, oddalać. Na dole ekranu widoczny jest przycisk "Wyznacz trasę", przenoszący użytkownika

do widoku z rys. 6.18. W lewym górnym rogu widoczne jest zwijane menu aplikacji, które pozwala użytkownikowi przejście na wybrany widok aplikacji.

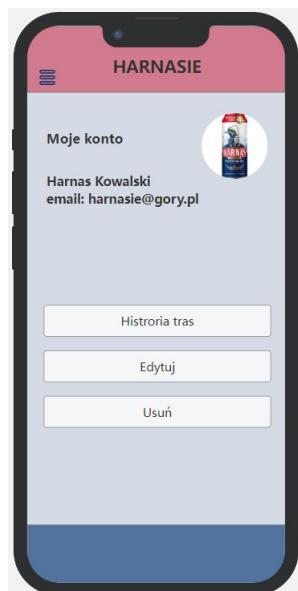
Menu rozwijane



Rysunek 6.15: Interfejs użytkownika: menu rozwijane.

Widok na rys.6.15 zawiera przyciski prowadzące do poszczególnych widoków aplikacji. Przycisk "Moje konto" przenosi użytkownika do widoku z rys. 6.16, przycisk "Telefony alarmowe" do widoku z rys. 6.20, "Zgłoś zagrożenie" do widoku z rys. 6.19, a "Wyloguj się" do ekranu logowania (rys. 6.13). Menu rozwijane widoczne jest we wszystkich widokach aplikacji z wyłączeniem ekranu logowania (rys. 6.13) oraz rejestracji (rys. 6.12).

Moje konto

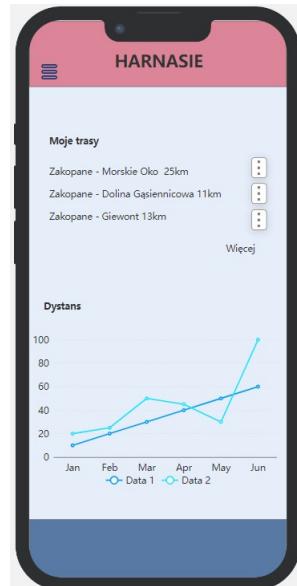


Rysunek 6.16: Interfejs użytkownika: moje konto.

Widok na rys 6.16 zawiera wypisane dane użytkownika, jego zdjęcie profilowe oraz przy-

ciski pozwalające na przejrzenie historii wędrówek, edycję danych oraz usunięcie konta. W przypadku wybrania przycisku "Historia tras" użytkownik przenoszony jest do kolejnego, osobnego widoku (rys. 6.17).

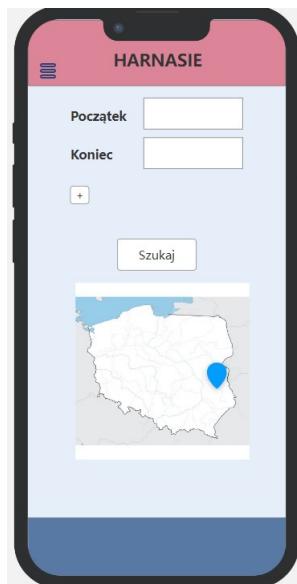
Historia tras



Rysunek 6.17: Interfejs użytkownika: historia przebytych tras.

Widok na rys. 6.17 zawiera historię przebytych przez użytkownika tras. Szczegóły każdej trasy użytkownik może zobaczyć klikając przycisk z trzema kropkami obok wybranej trasy. Poniżej wypisanych tras znajduje się także wykres, na którym użytkownik może zobaczyć wstępne, graficzne podsumowanie ostatnich wycieczek.

Wyznaczanie trasy

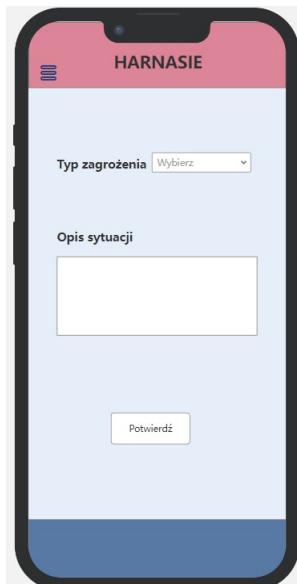


Rysunek 6.18: Interfejs użytkownika: wyznaczanie trasy.

Widok na rys. 6.18 zawiera pola do wpisania początku i końca trasy, jaką użytkownik

chce przejść. Pod nimi znajduje się przycisk "+", który pozwala na dodanie punktu postoju-wego i doprecyzowanie najbardziej pożądanej trasy. Wprowadzone dane po kliknięciu "Szukaj" wprowadzane są na mapę jako punkty i na ich podstawie wyznaczana jest najlepsza trasa, którą widać pod wyżej wymienionymi kontrolkami.

Zgłaszanie zagrożenia



Rysunek 6.19: Interfejs użytkownika: zgłoszenie zagrożenia.

Widok na rys. 6.19 zawiera listę rozwijaną typów zagrożeń, w tym możliwą opcję "inne", jeżeli dane zagrożenie nie zdarzyło się do tej pory. Każde zagrożenie można opisać w polu tekstowym poniżej. Po kliknięciu przycisku "Potwierdź" pojawia się powiadomienie o potwierdzeniu przesłania zgłoszenia do administratora, który zgłoszenie musi zatwierdzić (widok na rys. 6.22)

Telefony alarmowe



Rysunek 6.20: Interfejs użytkownika: telefony alarmowe.

Widok na rys. 6.20 zawiera wypisane ogólnopolskie telefony alarmowe, jak i numer na Górskie Ochotnicze Pogotowie Ratunkowe. Każdy numer można skopiować lub wybrać i zadzwonić na niego bezpośrednio z poziomu aplikacji.

Administrator

Administrator aplikacji jest jednostką bardzo istotną dla poprawnego działania całej aplikacji. Po zalogowaniu na konto administratora (rys. 6.13) dostępne są następujące widoki: ekran główny, zatwierdzanie zgłoszeń przesyłanych przez użytkowników, zarządzanie szlakami, zarządzanie atrakcjami, wysyłanie powiadomień oraz sekcja poświęcona GOPR.

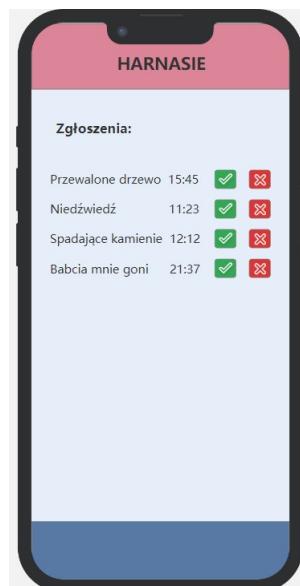
Ekran główny



Rysunek 6.21: Interfejs użytkownika: ekran główny administratora.

Widok na rys. 6.21 zawiera kilka przycisków wspomagających nawigację po aplikacji. Dzięki tym przyciskom może przenieść się do ekranów: wyślij powiadomienie, zgłoszenia do zatwierdzenia, zarządzaj szlakami, zarządzaj atrakcjami oraz GOPR. Przyciski są odpowiednio nazwane, aby nie było wątpliwości, na jaki ekran użytkownik za chwilę zostanie przeniesiony.

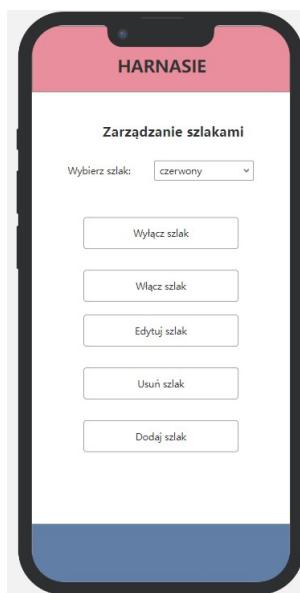
Zatwierdź zgłoszenia



Rysunek 6.22: Interfejs użytkownika: zatwierdzanie zgłoszeń przesłanych przez użytkowników.

Widok na rys. 6.22 zawiera listę zgłoszonych, nierożpatrzonych jeszcze zgłoszeń użytkowników. Każdy element listy zawiera typ zgłoszenia, datę wysłania zgłoszenia oraz przyciski do akceptacji bądź odrzucenia konkretnego zgłoszenia.

Zarządzanie szlakami



Rysunek 6.23: Interfejs użytkownika: zarządzanie szlakami.

Widok na rys. 6.23 zawiera listę rozwijaną szlaków dostępnych w bazie aplikacji. Administrator może z niej wybrać szlak, który wymaga modyfikacji a następnie wybrać jedną z dostępnych opcji edycji szlaku spośród: wyłącz szlak, włącz szlak, edytuj szlak oraz usuń

szlak. Opcje włączenia i wyłączenia szlaku odnoszą się do tymczasowego blokowania szlaków z użycia ze względu na długotrwałe występujące na nim utrudnienia, np. zasypany szlak po przejściu lawiny. Poniżej wszystkich przycisków związanych z modyfikacją istniejących już szlaków znajduje się jeszcze jeden, odpowiedzialny za przejście do nowego widoku (rys. 6.24), gdzie możliwe jest dodanie nowego szlaku do bazy.

Dodaj szlak



Rysunek 6.24: Interfejs użytkownika: dodawanie nowego szlaku do bazy dostępnych szlaków.

Widok na rys. 6.24 zawiera kontrolki wspomagające wprowadzenie wszystkich niezbędnych informacji na temat nowego szlaku, który administrator chce utworzyć. Aby dodać nowy szlak należy podać jego nazwę, zaznaczyć jego przebieg na mapie oraz utworzyć krótki opis szlaku. Po kliknięciu przycisku "Potwierdź" wyświetla się powiadomienie informujące administratora o pomyślnym dodaniu szlaku do bazy.

7 Implementacja

Implementacją projektu autorki podzieliły się według ich możliwości oraz mocnych stron. Moduł map, dopracowany i doszlifowany z wieloma funkcjami personalizującymi wygląd map, moduł powiadomień odpowiedzialny za komunikowanie użytkowników o czyniących na szlakach niebezpieczeństwach oraz moduł zgłaszania zagrożeń usprawniający przesył informacji pomiędzy wędrowcami. W tym rozdziale opisane zostały najistotniejsze elementy aplikacji: listingi wybranych funkcji, ich opisy oraz wszystkie dostępne widoki.

7.1 Moduł map

Podrozdział został poświęcony na dokładne przyjrzenie w jaki sposób działają funkcje, które umożliwiają poruszanie się po górach oraz wpływają na wygląd mapy. Każda z nich sprawia, iż użytkownik ma możliwość pokonania wybranej trasy, a naniesione na mapę szlaki pozwalają mu zaplanować wędrówkę w każdym momencie. Poniżej przedstawione zostały wybrane funkcje składające się na implementację rysowania szlaków, wyznaczenia trasy, zapisu lokalnego plików oraz obsługi ikon ze schroniskami, szczytami i stawami.

Implementacja rysowania szlaków na mapie

Fragment kodu, zaprezentowany na listingu 7.1 odczytuje dane z pliku KML i na ich podstawie rysuje szlaki na mapie. Za pomocą FileInputStream zostaje stworzony strumień wejściowy do odczytania pliku, który potrzebny jest do utworzenia warstwy KML. Następnie jest ona dodawana do mapy i rozpoczyna się iterowanie po kontenerach. Pętla odczytująca obiekty KmlPlacemark, pobiera ich nazwę i na jej podstawie określa kolor szlaku. W kolejnym kroku zostają zdefiniowane opcje linii, czyli wybrany kolor oraz szerokość. Jeżeli geometria danego znacznika jest linią (KmlLineString), tworzona jest lista, przechowująca współrzędne szlaku, które następnie dodane są do wcześniej zdefiniowanej linii. Na koniec, utworzona trasa zostanie narysowana na mapie, dzięki czemu użytkownik widzi wszystkie szlaki Tatr. Efekt ten widoczny jest na rys. 7.6.

Listing 7.1: Funkcja odpowiedzialna za rysowanie szlaków na mapie

```
private void readAndProcessKMLFile(File file) { 1 usage
    try {
        FileInputStream fileInputStream = new FileInputStream(file);
        KmlLayer kmlLayer = new KmlLayer(mMap, fileInputStream, context);
        kmlLayer.addLayerToMap();
        for (KmlContainer container : kmlLayer.getContainers()) {
            for (KmlPlacemark placemark : container.getPlacemarks()) {
                String name = placemark.getProperty("name");

                int color = Color.GRAY;
                if (name != null) {
                    if (name.contains("czarny")) {
                        color = Color.RED;
                    } else if (name.contains("niebieski")) {
                        color = Color.BLUE;
                    } else if (name.contains("żółty")) {
                        color = Color.YELLOW;
                    } else if (name.contains("zielony")) {
                        color = Color.GREEN;
                    } else if (name.contains("czarny")) {
                        color = Color.BLACK;
                    }
                }

                PolylineOptions polylineOptions = new PolylineOptions()
                    .color(color)
                    .width(5f);

                if (placemark.getGeometry() instanceof KmlLineString) {
                    KmlLineString lineString = (KmlLineString) placemark.getGeometry();
                    List<LatLng> coordinates = lineString.getGeometryObject();
                    polylineOptions.addAll(coordinates);
                    mMap.addPolyline(polylineOptions);
                }
            }
        }
    } catch (IOException | XmlPullParserException e) {
        Log.e(TAG, msg: "Błąd podczas odczytu pliku: " + e.getMessage());
    }
}
```

Implementacja lokalnego zapisu plików

Funkcja przedstawiona na rys. 7.2 odpowiada za kontrolę pobierania plików z serwera (Firebase Storage) i jest wywołana po każdym uruchomieniu aplikacji. Pliki zawierają informacje o szlakach oraz punktach, w których znajdują się schroniska, szczyty oraz stawy. Na początku funkcja tworzy obiekt reprezentujący ścieżkę do katalogu w pamięci i sprawdza, czy istnieje on fizycznie, jeśli nie, tworzy go. Następnie ustala pełną ścieżkę do lokalnego pliku i również sprawdza czy istnieje. Dalszy fragment kodu porównuje jego rozmiar z rozmiarem wersji na serwerze, aby określić, czy pobranie nowej wersji jest konieczne. Jeśli lokalny plik różni się zostaje on pobierany ponownie. Działanie funkcji sprawia, iż użytkownik ma zawsze aktualne dane zapisane na swoim urządzeniu.

Listing 7.2: Funkcja odpowiedzialna za lokalny zapis plików.

```
private void checkAndDownloadFile(StorageReference fileRef, String filename) {
    File localDir = new File(this.getExternalFilesDir(Environment.DIRECTORY_DOCUMENTS), filename);
    if (!localDir.exists()) {
        localDir.mkdirs();
    }

    File localFile = new File(localDir, fileRef.getName());

    if (localFile.exists()) {
        fileRef.getMetadata().addOnSuccessListener(metadata -> {
            long serverFileSize = metadata.getSizeBytes();
            long localFileSize = localFile.length();

            if (serverFileSize == localFileSize) {
                Log.d(TAG, msg: "Plik jest już aktualny, nie trzeba pobierać: " + fileRef.getName());
            } else {
                Log.d(TAG, msg: "Plik różni się od lokalnej wersji, pobieram nową wersję: " + fileRef.getName());
                downloadFile(fileRef, localFile);
            }
        }).addOnFailureListener(exception -> {
            Log.e(TAG, msg: "Błąd pobierania metadanych dla pliku: " + fileRef.getName(), exception);
            downloadFile(fileRef, localFile);
        });
    } else {
        fileRef.getMetadata().addOnSuccessListener(metadata -> {
            Log.d(TAG, msg: "Plik nie istnieje lokalnie, ale jest na serwerze. Pobieram: " + fileRef.getName());
            downloadFile(fileRef, localFile);
        }).addOnFailureListener(exception -> {
            Log.e(TAG, msg: "Błąd pobierania metadanych dla pliku: " + fileRef.getName(), exception);
        });
    }
}
```

Implementacja ikon ze schroniskami, szczytami oraz stawami

Fragment kodu przedstawiony na listingu 7.3 odpowiada za odczyt danych z pliku i utworzenie listy markerów. Użycie XmlPullParser umożliwia analizę pliku i pobranie danych potrzebnych do stworzenia markera. Po jego konfiguracji, następuje iterowanie po pliku, aż do osiągnięcia końca dokumentu. Podczas iteracji funkcja odczytuje dane z konkretnych tagów, kolejno nazwę z <name>, opis <description> oraz współrzędne <coordinates>. Po napotkaniu na tag zamykający <Placemark> i sprawdzeniu czy lokalizacja jest poprawna, zostaje utworzony marker z ikoną pokazaną jako parametr funkcji. Następnie jest on dodany do listy, która umożliwia kontrolę widoczności markerów.

Listing 7.3: Funkcja odpowiedzialna za wyświetlanie na mapie ikon schronisk, szczytów oraz stawów.

```
private List<Marker> addMarkersFromKML(String absolutePath, BitmapDescriptor icon) { 3 usages
    ... //inicjalizacja zmiennych name, description, coordinates, markerList

    try {
        InputStream kmlInputStream = new FileInputStreamAbsolutePath);
        XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
        factory.setNamespaceAware(true);
        XmlPullParser parser = factory.newPullParser();
        parser.setInput(kmlInputStream, "UTF-8");

        int eventType = parser.getEventType();
        while (eventType != XmlPullParser.END_DOCUMENT) {
            String tagName = parser.getName();

            switch (eventType) {
                case XmlPullParser.START_TAG:
                    if ("name".equals(tagName)) { name = parser.nextText(); }
                    else if ("description".equals(tagName)) { description = parser.nextText(); }
                    else if ("coordinates".equals(tagName)) {
                        ...
                        //rozdzielenie współrzędnych
                        coordinates = new LatLng(lat, lng);
                    } break;

                case XmlPullParser.END_TAG:
                    if ("Placemark".equals(tagName)) {
                        if (coordinates != null) {
                            Marker marker = mMap.addMarker(new MarkerOptions() ... //dodanie markera na mapę
                            coordinates = null;
                            if (marker != null) { markersList.add(marker); }
                        }
                    } break;
                } eventType = parser.next();
            }
        } catch (Exception e) { e.printStackTrace(); }
        return markersList;
    }
}
```

Implementacja wyznaczenia trasy

Za wyznaczanie trasy odpowiedzialnych jest kilka funkcji. Jedna z nich została przedstawiona na listingu 7.4 i służy do utworzenia adresu URL, który definiuje trasę. Początek kodu sprawdza, czy dane trasy są puste, jeśli tak, zostanie rzucony wyjątek z informacją o tym, jeśli nie, zostaje wykonany dalszy kod. Określone zostają informacje o wyznaczonej trasie, t.j. początek i koniec oraz tryb pieszy, a także klucz API Google Maps, potrzebny do poprawnego stworzenia adresu URL. Funkcja encodeLatLng() konwertuje współrzędne, tak aby pasowały do jego formatu. Kolejnym krokiem jest sprawdzenie, czy trasa posiada przystanki, jeśli tak, ich współrzędne są pobierane i dodane do ciągu znaków. Na koniec, wszystkie parametry połączone zostają w jeden ciąg tekstowy, którym jest gotowy adres URL. Link zawiera wynik trasy w formacie JSON, który jest odbierany i przetwarzany przez kolejne funkcje, które nie zostały opisane ze względu na ich ilość. Po wykonaniu ich wszystkich, można z odczytać szczegółowe dane drogi, wybranej przez użytkownika.

Listing 7.4: Funkcja odpowiedzialna za wyznaczanie trasy

```
private String getDirectionsUrlWITH(LatLng origin, LatLng dest, List<LatLng> waypointsList) { 1 usage
    if (origin == null || dest == null) {
        throw new IllegalArgumentException("Dane trasy są nieuzupełnione");
    }

    String str_origin = "origin=" + encodeLatLng(origin);
    String str_dest = "destination=" + encodeLatLng(dest);
    String mode = "mode=walking";
    String key = "key=AIZaSyC4KaiLnKSYLuF9xCyzudGh8DMCB-6HefJA";

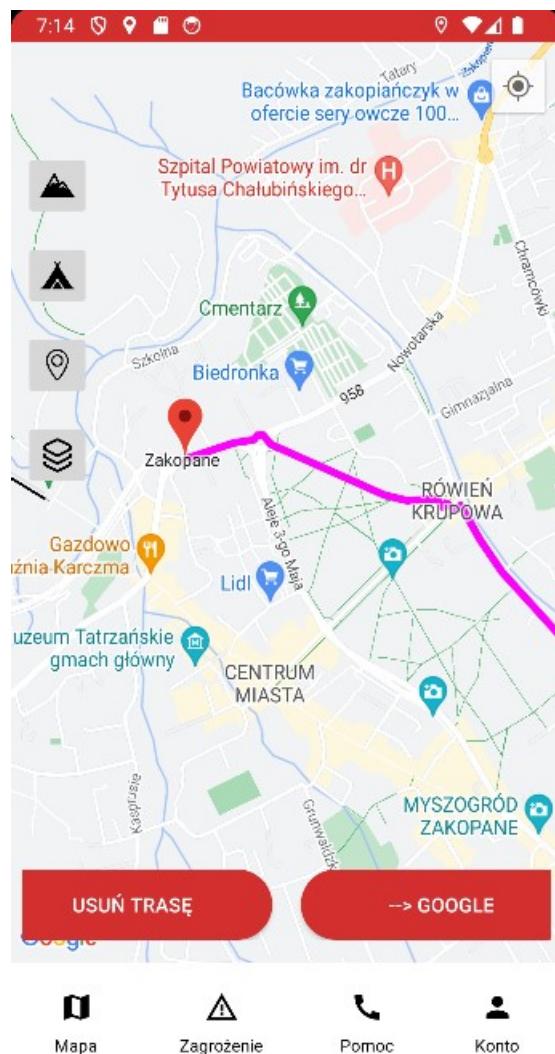
    String waypoints = "";
    if (waypointsList != null && !waypointsList.isEmpty()) {
        StringBuilder waypointsBuilder = new StringBuilder("waypoints=");
        for (int i = 0; i < waypointsList.size(); i++) {
            LatLng point = waypointsList.get(i);
            waypointsBuilder.append(encodeLatLng(point));
            if (i < waypointsList.size() - 1) {
                waypointsBuilder.append("|");
            }
        }
        waypoints = waypointsBuilder.toString();
    }

    StringBuilder parametersBuilder = new StringBuilder(str_origin);
    parametersBuilder.append("&").append(str_dest);
    parametersBuilder.append("&").append(mode);
    parametersBuilder.append("&").append(key);

    if (!waypoints.isEmpty()) {
        parametersBuilder.append("&").append(waypoints);
    }

    return "https://maps.googleapis.com/maps/api/directions/json?" + parametersBuilder.toString();
}
```

Funkcja, przedstawiona na listingu 7.5 rysuje wyznaczoną trasę z przetworzonego wyniku, uzyskanego z adresu URL. Przed pobraniem współrzędnych linii, mapa jest czyszczona z innych tras. Następnie tworzonych jest kilka obiektów: lista punktów nowej ścieżki, zbiór z unikalnymi punktami, który zapobiega powielaniu takich samych współrzędnych oraz linia, reprezentująca nową trasę. Podczas iteracji, pobierane są współrzędne każdego punktu na ścieżce i na ich podstawie utworzony zostaje obiekt LatLng. Jeśli dodanie jego do zbioru uniquePoints przejdzie pomyślnie, zostanie on dodany do listy z punktami trasy. W kolejnym kroku wyznaczona droga, zostanie zdefiniowana o te współrzędne, kolor oraz szerokość. Następnie funkcja rysuje ją na mapie i aktualizuje listę jej punktów. Na koniec, kamera zostanie ustawiona na pierwszy punkt trasy z określonym przybliżeniem, dzięki czemu użytkownik może rozpocząć wędrówkę po swojej ścieżce.



Rysunek 7.1: Pokazanie efektu po zatwierdzeniu wybranej przez użytkownika trasy.

Listing 7.5: Funkcja odpowiedzialna za rysowanie szlaków na mapie

```
@Override 2 usages
protected void onPostExecute(List<List<HashMap<String, String>>> result) {
    routePoints.clear();
    ArrayList<LatLng> points = new ArrayList<>();
    HashSet<LatLng> uniquePoints = new HashSet<>();
    PolylineOptions lineOptions = new PolylineOptions();
    routePolylines.clear();

    for (int i = 0; i < result.size(); i++) {
        List<HashMap<String, String>> path = result.get(i);
        for (HashMap<String, String> point : path) {
            double lat = Double.parseDouble(point.get("lat"));
            double lng = Double.parseDouble(point.get("lng"));
            LatLng position = new LatLng(lat, lng);

            if (uniquePoints.add(position)) {
                points.add(position);
            }
        }
    }

    lineOptions.addAll(points);
    lineOptions.width(15);
    lineOptions.color(Color.MAGENTA);

    if (mMap != null) {
        clearPreviousRoute();
        currentRoutePolyline = mMap.addPolyline(lineOptions);
        routePoints.clear();
        routePoints.addAll(points);

        if (!routePoints.isEmpty()) {
            LatLng startPoint = routePoints.get(0);
            mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(startPoint, zoom: 15));
        }
    }
}
```

7.2 Moduł zgłoszeń

Przy procesie obsługi zgłoszeń uczestniczy zarówno użytkownik jak i administrator. Pierwszy z nich zgłasza zagrożenie, podając wszystkie niezbędne informacje o bezpieczeństwie. Drugi natomiast decyduje, czy dane zgłoszenie jest pomocne dla innych użytkowników. Poniżej zostały przedstawione najważniejsze funkcje składające się na implementację zapisu zgłoszeń do bazy Firebase oraz akceptacji zgłoszeń.

Implementacja zapisu zgłoszeń do bazy (Firebase)

Fragment kodu, pokazany na rys. 7.6 odpowiada za zapis zgłoszenia do bazy. Na początku sprawdzone zostaje połączenie z internetem, jeśli go nie ma, dane zostają zapisane lokalnie na urządzeniu, za co odpowiedzialna jest funkcja `saveDangerLocally()`. W przypadku, gdy użytkownik ma włączony internet i GPS, zostaje zapisana jego lokalizacja oraz możliwe jest pobranie jego nazwy oraz email. Na podstawie danych o zalogowanym zostaje utworzony obiekt `Map<String, Object>`, który jest potrzebny do poprawnego zapisu zgłoszenia. Następnie dane przekazane w parametrze funkcji zostają dodane do obiektu “danger”, który zawiera informacje o zagrożeniu. W kolejnym kroku zostają one zapisane do bazy. W przypadku niepowodzenia zostaje wyświetlony komunikat o błędzie.

Listing 7.6: Funkcja odpowiedzialna za zapis zgłoszeń do bazy.

```

private void addDanger(String description, String type, String uid) {
    if (!isInternetAvailable()) { saveDangerLocally(description, type, uid); }
    else {
        checkLocationPermission();
        ... //sprawdzenie dostępu do lokalizacji
        mFusedLocationClient.getLastLocation().addOnSuccessListener(activity: this, location -> {
            if (location != null) {
                currentLocation = new LatLng(location.getLatitude(), location.getLongitude());
            }
        });
        db.collection(collectionPath: "users").document(uid).get().addOnSuccessListener(userDoc -> {
            if (userDoc.exists()) {
                Map<String, Object> userMap = new HashMap<>();
                userMap.put(k: "email", userDoc.getString(field: "email"));
                ...
                //pozostałe dane użytkownika
                Map<String, Object> danger = new HashMap<>();
                danger.put(k: "description", description);
                ...
                //pozostałe dane zgłoszenia
            }
            db.collection(collectionPath: "dangers").add(danger)
                .addOnSuccessListener(documentReference -> {
                    Toast.makeText(context: DangerActivity.this, text: "Dodano zgłoszenie z ID: " + documentReference.getId(), Toast.LENGTH_SHORT).show();
                })
                .addOnFailureListener(e -> {
                    Log.e(tag: "FirestoreError", msg: "Błąd przy dodawaniu zagrożenia: " + e.getMessage());
                    Toast.makeText(context: DangerActivity.this, text: "Błąd przy dodawaniu zagłoszenia: " + e.getMessage(), Toast.LENGTH_SHORT).show();
                });
            } else {
                Toast.makeText(context: this, text: "Nie znaleziono użytkownika.", Toast.LENGTH_SHORT).show();
            }
        }).addOnFailureListener(e -> {Toast.makeText(context: this, text: "Błąd przy pobieraniu danych użytkownika: " + e.getMessage(), Toast.LENGTH_SHORT).show();});
    } else {Toast.makeText(context: DangerActivity.this, text: "Nie można uzyskać bieżącej lokalizacji.", Toast.LENGTH_SHORT).show();};
});;
}

```

Implementacja akceptacji zgłoszeń

Funkcja, pokazana na rys. 7.7 odpowiada za aktualizację danych zgłoszenia, a dokładniej ustawienie wartości pola “accepted” na “true”. Na podstawie identyfikatora dokumentu, przekazanego w parametrze funkcji, wiadome jest, które zgłoszenie należy zaktualizować. Skutki działania tego fragmentu kodu są wykorzystane w innych funkcjach aplikacji, np. getAccpetDangersLoc opisanej poniżej.

Listing 7.7: Funkcja odpowiedzialna za akceptacje zgłoszeń.

```

private void acceptDanger(String dangerId) { 1 usage
    db.collection(collectionPath: "dangers").document(dangerId).DocumentReference
        .update(field: "accepted", value: true) Task<Void>
        .addOnSuccessListener(aVoid -> {
            Toast.makeText(context, text: "Zagłoszenie zaakceptowane.", Toast.LENGTH_SHORT).show();
        })
        .addOnFailureListener(e -> {
            Toast.makeText(context, text: "Błąd przy akceptacji zgłoszenia: " + e.getMessage(), Toast.LENGTH_SHORT).show();
        });
}

```

Funkcja przedstawiona na rys. 7.8 służy do pobierania lokalizacji zgłoszeń zagrożeń, które zostały zaakceptowane. Na początku dane są filtrowane, tzn. sprawdzane jest czy dane zgłoszenie w polu dokumentu “accepted” ma wartość “true”. W przypadku niepowodzenia podczas pobierania danych z bazy, zostaje wyświetlony komunikat, informujący o takiej sytuacji. Jeśli pobieranie danych zakończy się powodzeniem i wyniki filtrowania nie są puste, następuje iteracja przez każdy dokument. W kolejnym kroku jest sprawdzane czy posiada on pole “location”, jeśli tak, dane zostają pobrane i zapisane do Map<String, Object>. Kluczem stworzonej mapy jest “latitude” lub “longitude”, a wartością współrzędne. Następnie, dane te są łączone w jeden obiekt LatLng, reprezentujący miejsce, w którym zostało wykonane zgłoszenie. Na koniec, współrzędne są dodawane do listy, przechowującej lokalizacje zaakceptowanych zgłoszeń. Po sprawdzeniu czy nie jest ona pusta zostaje wywołana funkcja updateMapWithAcceptedDangers(), przedstawiona na rys. 7.9. Odpowiada ona za ustawienie markerów na mapie w miejscu, wystąpienia zagrożenia. Mapa po aktualizacji została przedstawiona na rys XX.

Listing 7.8: Funkcja odpowiedzialna za pobranie lokalizacji zgłoszeń.

```

private void getAcceptedDangersLocations() { // usage
    db.collection(collectionPath: "dangers").CollectionReference
        .whereEqualTo(field: "accepted", value: true).Query
        .get().Task<QuerySnapshot>
        .addOnSuccessListener(queryDocumentSnapshots -> {
            if (queryDocumentSnapshots != null && !queryDocumentSnapshots.isEmpty()) {
                List<Latlng> locations = new ArrayList<>();
                for (DocumentSnapshot documentSnapshot : queryDocumentSnapshots) {
                    if (documentSnapshot.contains("location")) {
                        Map<String, Object> locationMap = (Map<String, Object>) documentSnapshot.get("location");

                        if (locationMap != null) {
                            double latitude = (double) locationMap.get("latitude");
                            double longitude = (double) locationMap.get("longitude");

                            Latlng location = new Latlng(latitude, longitude);
                            locations.add(location);

                            Log.d(tag: "Accepted Danger", msg: "Location: " + latitude + ", " + longitude);
                        }
                    }
                }
                if (!locations.isEmpty()) {
                    updateMapWithAcceptedDangers(locations);
                } else {Toast.makeText(context: MapActivity.this, text: "Brak zaakceptowanych zgłoszeń.", Toast.LENGTH_SHORT).show();}
            } else {Toast.makeText(context: MapActivity.this, text: "Brak zgłoszeń w bazie.", Toast.LENGTH_SHORT).show();}
        })
        .addOnFailureListener(e -> {
            Toast.makeText(context: MapActivity.this, text: "Błąd przy pobieraniu zgłoszeń: " + e.getMessage(), Toast.LENGTH_SHORT).show();
            Log.e(tag: "FirestoreError", msg: "Błąd przy pobieraniu zaakceptowanych zgłoszeń", e);
        });
}
}

```

Listing 7.9: Funkcja odpowiedzialna za ustawienie markerów z zagrożeniami na mapie.

```

private void updateMapWithAcceptedDangers(List<Latlng> locations) { // usage
    Bitmap originalBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.warning_sign);
    Bitmap scaledBitmap = Bitmap.createScaledBitmap(originalBitmap, dstWidth: 85, dstHeight: 85, filter: false);
    BitmapDescriptor icon = BitmapDescriptorFactory.fromBitmap(scaledBitmap);
    for (Latlng location : locations) {
        mMap.addMarker(new MarkerOptions().position(location).icon(icon).title("Zagrożenie"));
    }
}

```

7.3 Moduł logowania i rejestracji

Logowanie i rejestracja pozwala na łatwe zidentyfikowanie użytkownika aplikacji, a także pozwala na śledzenie jego aktywności celem stworzenia raportów czy wykresów. Pozwala też na ustalenie, który użytkownik wysłał zgłoszenie o zagrożeniu. Poniżej przedstawione są funkcje niezbędne do implementacji logowania oraz rejestracji nowych użytkowników w systemie.

Implementacja rejestracji

Rejestracja w tej aplikacji opiera się na adresie email oraz haśle i wybraniu nazwy użytkownika. Wszystkie te dane muszą być podane, aby rejestracja przebiegła pomyślnie. Do zbiegu rejestracji wykorzystywany jest moduł Firebase Authentication ułatwiający autoryzację użytkowników. Cała rejestracja kryje się pod przyciskiem "Zarejestruj" na widoku rejestracji użytkownika.

Listing 7.10: Rejestracja użytkownika w "słuchaczu" przycisku "Zarejestruj".

```
btnSignUp.setOnClickListener(v -> {
    String email = etEmail.getText().toString().trim();
    String password = etPassword.getText().toString().trim();
    String username = etUsername.getText().toString().trim();

    if (email.isEmpty() || password.isEmpty() || username.isEmpty()) {
        Toast.makeText(context: SignUpActivity.this, text: "Wypełnij wszystkie pola.", Toast.LENGTH_SHORT).show();
        return;
    }

    auth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener(activity: this, task -> {
            if (task.isSuccessful()) {
                FirebaseUser user = auth.getCurrentUser();
                if (user != null) {
                    addUserToFirestore(user.getUid(), email, username);
                }
            } else {
                String errorMessage = task.getException() != null ? task.getException().getMessage() : "Unknown error";
                Log.e(tag: "SignUpError", msg: "Sign-up failed: " + errorMessage);
                Toast.makeText(context: SignUpActivity.this, text: "Rejestracja nie powiodła się: " + errorMessage, Toast.LENGTH_SHORT).show();
            }
        });
});
```

Chcemy także, aby nowi użytkownicy byli zapisywani do bazy danych Firebase Firestore celem przypisania im ich aktywności na mapie. Odpowiedzialna jest za to funkcja addUserToFirestore(), która na podstawie przekazanego uid użytkownika z Firebase Authentication tworzy takiego samego użytkownika w bazie danych Firebase Firestore ze wszystkimi wprowadzonymi przez użytkownika danymi. Cała mapa z danymi zapisywana jest jako nowy dokument w bazie NoSQL, do którego można się odwołać za pomocą unikalnego uid użytkownika.

Listing 7.11: Dodawanie użytkownika do Firebase Firestore.

```
protected void addUserToFirestore(String uid, String email, String username) {
    if (uid == null || uid.isEmpty()) {
        Toast.makeText(context: SignUpActivity.this, text: "Błąd: UID jest null lub pusty", Toast.LENGTH_SHORT).show();
        return;
    }

    Map<String, Object> userMap = new HashMap<>();
    userMap.put("email", email);
    userMap.put("username", username);
    userMap.put("uid", uid);
    userMap.put("role", "user");

    db.collection(collectionPath: "users").document(uid).set(userMap)
        .addOnSuccessListener(avoid ->
            Toast.makeText(context: SignUpActivity.this, text: "Użytkownik dodany do bazy.", Toast.LENGTH_SHORT).show()
        )
        .addOnFailureListener(e -> {
            Log.e(tag: "FirestoreError", msg: "Error adding user", e);
            Toast.makeText(context: SignUpActivity.this, text: "Błąd przy dodawaniu nowego użytkownika: " + e.getMessage(), Toast.LENGTH_SHORT).show();
        });
}
```

Implementacja logowania.

Każdy użytkownik chce mieć dostęp do swoich danych, a także utworzonego konta. Logowanie pomaga nam zidentyfikować konkretnego użytkownika oraz przypisać mu jego aktywność. W tym module aplikacja również posiłkuje się Firebase Authentication. Pozwala to na szybkie logowanie użytkownika i udostępnienie mu wszystkich funkcjonalności aplikacji zgodnych z jego rolą.

Listing 7.12: Logowanie użytkownika w ”słuchaczu” przycisku ”Zaloguj”.

```
auth.signInWithEmailAndPassword(email, password)
    .addOnCompleteListener( activity: this, task -> {
        if (task.isSuccessful()) {
            firebaseUser user = auth.getCurrentUser();
            String userId = auth.getCurrentUser().getUid();
            db.collection( collectionPath: "users" ).document(userId).get()
                .addOnSuccessListener( documentSnapshot -> {
                    if (documentSnapshot.exists()) {
                        String role = documentSnapshot.getString( field: "role" );
                        String username = documentSnapshot.getString( field: "username" );
                        Log.d( tag: "DEBUG", msg: "Rola użytkownika: " + role );
                        setLoggedInState( state: true, userId, username );
                        if ("admin".equalsIgnoreCase(role)) {
                            Log.d( tag: "DEBUG", msg: "POSZLO" );
                            Intent intent = new Intent( packageContext: SignInActivity.this, AdminMenuActivity.class );
                            startActivity(intent);
                            finish();
                        } else {
                            Intent intent = new Intent( packageContext: SignInActivity.this, UserActivity.class );
                            startActivity(intent);
                            finish();
                        }
                    }
                });
        if (user != null) {
            fetchUsernameFromFirestore(user.getUid());
        }
    }
}
```

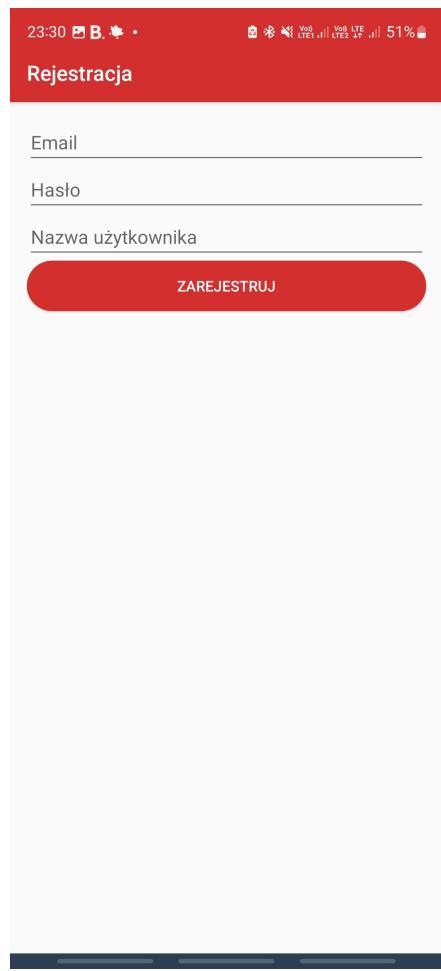
7.4 Widoki - użytkownik

Widok główny

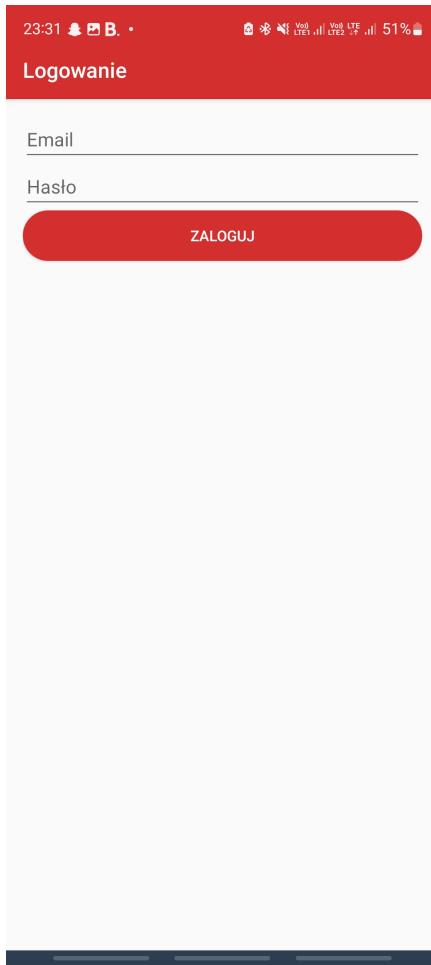
Po uruchomieniu aplikacji użytkownik ma do wyboru dwie opcje ”Zaloguj” oraz ”Zarejestruj”, co zostało przedstawione na rys. 7.2. Po kliknięciu danej opcji, wyświetla się formularz, który pobiera od użytkownika potrzebne dane i zapisuje je w bazie. Rejestrację przedstawia rys. 7.3, natomiast logowanie rys. 7.4. Jeżeli użytkownik logował się już wcześniej do aplikacji, widok początkowy jest pomijany i od razu przenosi użytkownika do następnego ekranu, pokazanego na rys. 7.5. W sytuacji, gdy zalogowanym jest administrator, wyświetlony zostanie widok aplikacji, przedstawiony na rys. 7.15. Tło na początkowym ekranie reprezentuje dwie najważniejsze funkcjonalności aplikacji t.j. wędrówki po górach oraz ostrzeganie innych o zagrożeniu. Rejestracja jest ważna, gdyż dzięki niej wiadomo kto zgłosił dane zagrożenie oraz pozwala na zbieranie danych o aktywności użytkownika.



Rysunek 7.2: Widok początkowy po uruchomieniu aplikacji.



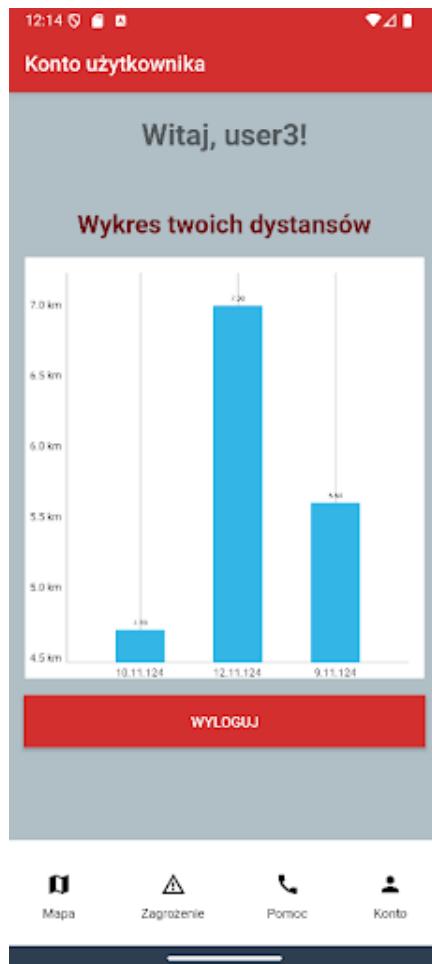
Rysunek 7.3: Widok formularza rejestracji do aplikacji.



Rysunek 7.4: Widok formularza logowania do aplikacji.

Konto użytkownika

Na ekranie zaprezentowanym na rys.7.5 użytkownik może zobaczyć swoją nazwę oraz wykaz pokonanych dystansów, przedstawionych w formie wykresu słupkowego. Pod nim znajduje się przycisk, dzięki któremu użytkownik może się wylogować z aplikacji. Na samym dole ekranu widoczne jest menu z czterema ikonami. Po naciśnięciu na wybrany przycisk, użytkownik zostanie przeniesiony do odpowiedniej aktywności. Pasek menu znajduje się na każdym ekranie aplikacji i umożliwia szybką nawigację między jej widokami.



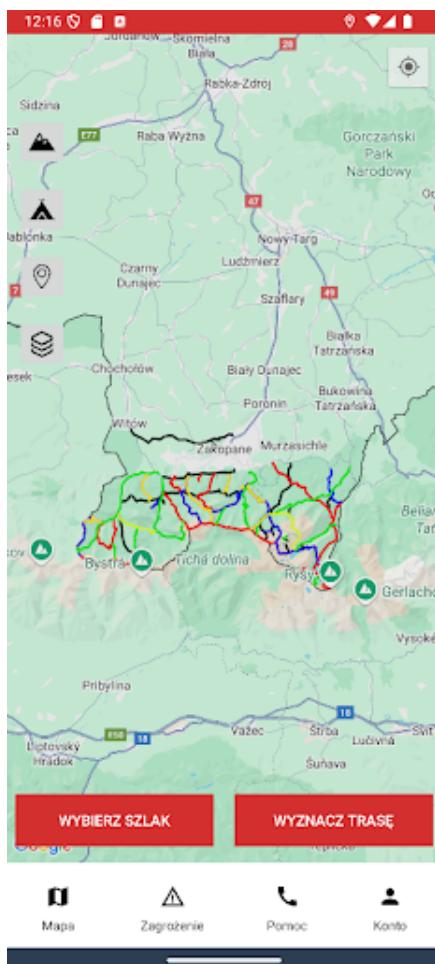
Rysunek 7.5: Widok danych użytkownika z wykresem.

Widok mapy

Ekran mapy, przedstawiony na rys.7.6 wyświetla szlaki Tatr oraz pozwala użytkownikowi wykonać wiele operacji m.in. zaplanować trasę oraz przejrzeć same szlaki. Wszystkie funkcjonalności będą opisane poniżej. Ekran mapy zawiera następujące elementy:

1. Ikonki ze schroniskami, szczytami oraz stawami - odpowiadają za widoczność danych markerów na mapie
2. Ikona z warstwą - zmienia wygląd mapy z podstawowego na widoki z satelity
3. Ikona z lokalizacją - nakierowuje mapę na obecną lokalizację użytkownika
4. Wybierz szlak - pokazuje listę szlaków do wyboru
5. Wyznacz trasę - umożliwia użytkownikowi podanie szczegółów trasy (początek, koniec oraz przystanki).

Szczegółowy opis ostatnich dwóch opcji będzie przedstawiony w dalszej pracy.

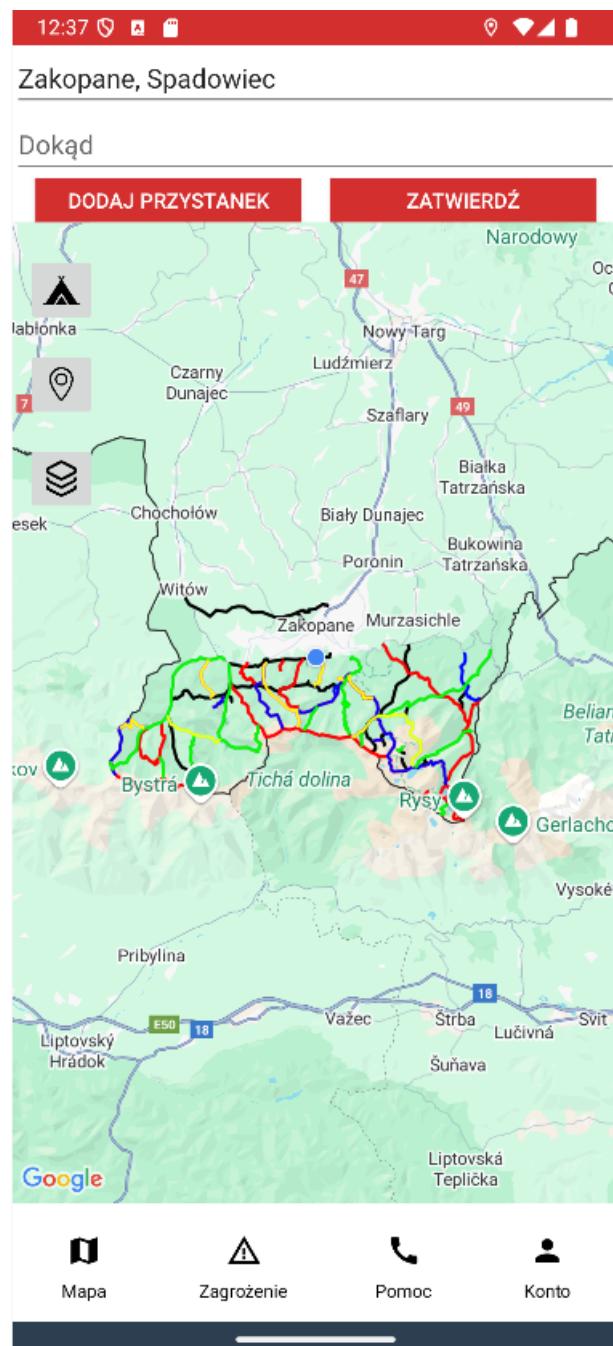


Rysunek 7.6: Widoku mapy z narysowanymi szlakami.

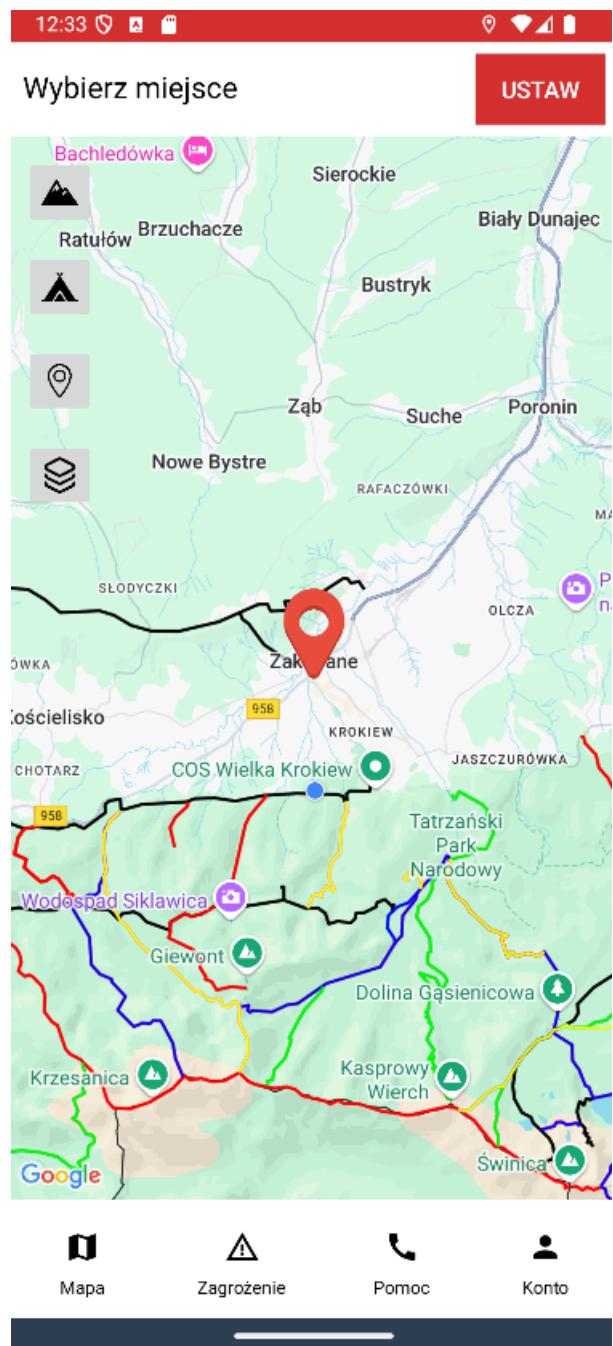
Wyznaczanie trasę

Wybranie opcji “wyznacz trasę” z widoku mapy (rys. 7.7) spowoduje wyświetlenie okna, które umożliwia użytkownikowi podanie szczegółów trasy. Po kliknięciu pola z punktem trasy, na środku mapy pokazuje się marker oraz przycisk do zatwierdzenia lokalizacji, co zostało przedstawione na rys. 7.8 Potwierdzenie wybranej lokalizacji spowoduje, iż w miejscu pola z punktem trasy pojawia się jej nazwa (rys. 7.9). Użytkownik w każdej chwili może zmienić ustawione punkty trasy, ponownie klikając w wybrane pole. W przypadku włączonej lokalizacji, pole z początkiem trasy jest uzupełnione już o współrzędne wędrówkowicza. Po ustaleniu wszystkich pól z punktami trasy, użytkownik może wybrać następujące opcje:

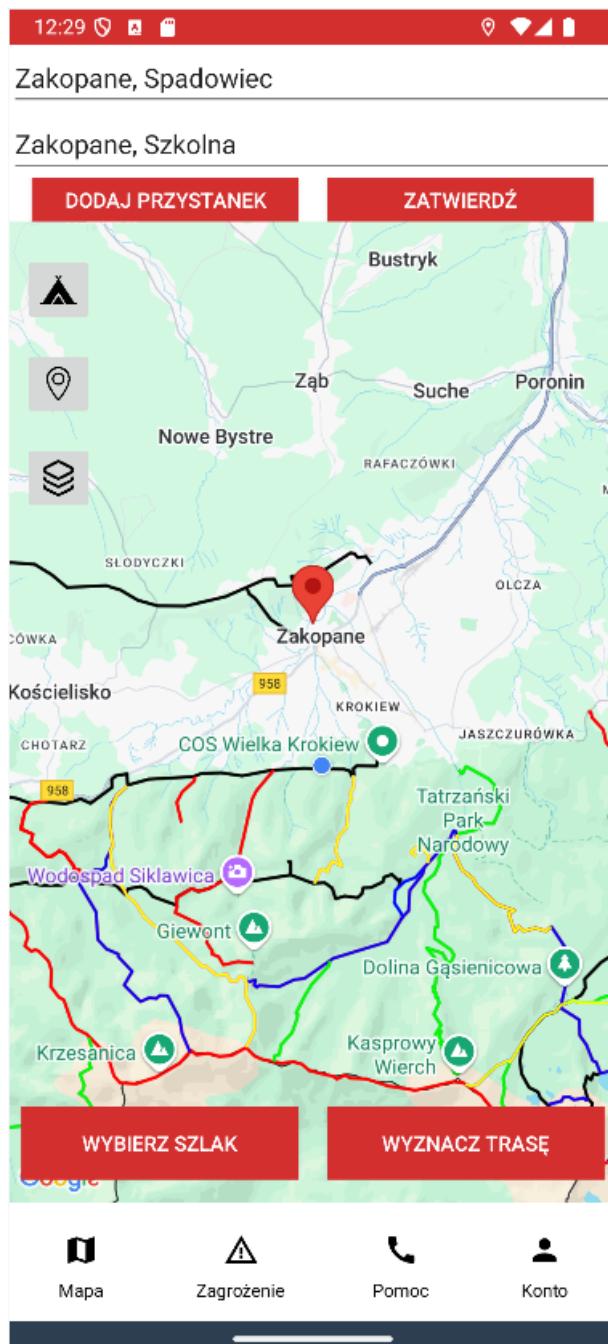
- Dodaj przystanek - pokazuje pole do podania jego lokalizacji,
- Zatwierdź - wyznacza na mapie trasę, którą stworzył użytkownik po podaniu wszystkich potrzebnych punktów,
- Usuń - usuwa przystanek z trasy



Rysunek 7.7: Opcje widoczne po kliknięciu przycisku "Wyznacz trasę".



Rysunek 7.8: Wybór punktów trasy za pomocą markera.

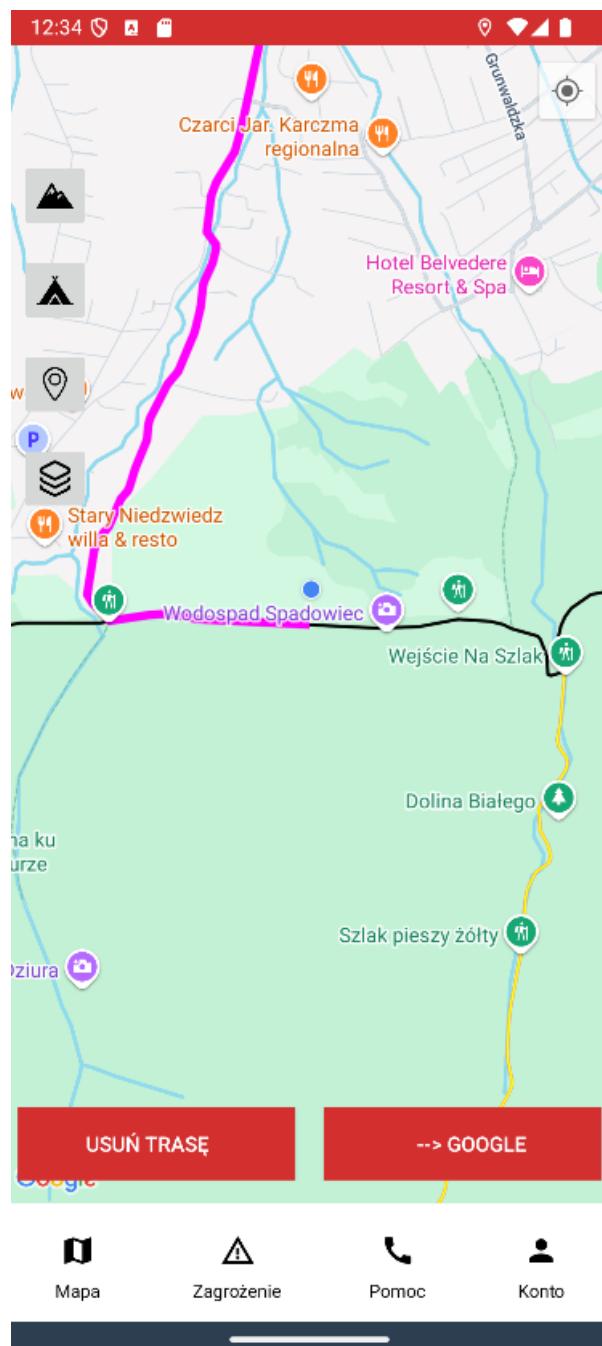


Rysunek 7.9: Rezultat wyboru lokalizacji punktu trasy.

Zatwierdzanie trasy

Po zatwierdzeniu trasy, pokazuje się ona na mapie wraz z kolejnych opcjami, które zostały przedstawione na rys. 7.10:

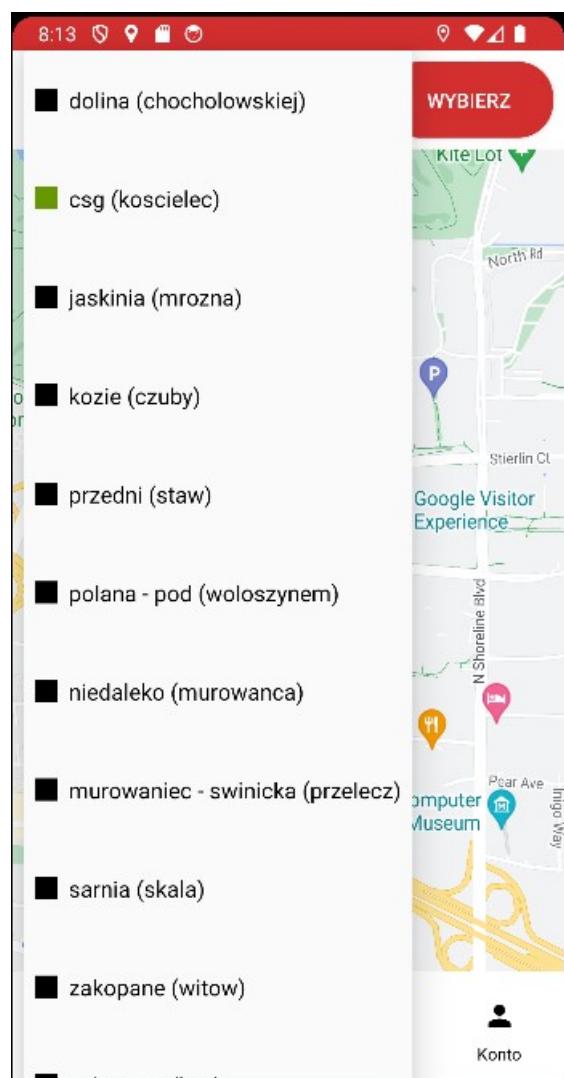
- Usuń trasę - powoduje usunięcie trasy oraz markerów oznaczających kolejne jej punkty,
- Google - umożliwia użytkownikowi nawigację po szlaku za pomocą aplikacji Mapy Google



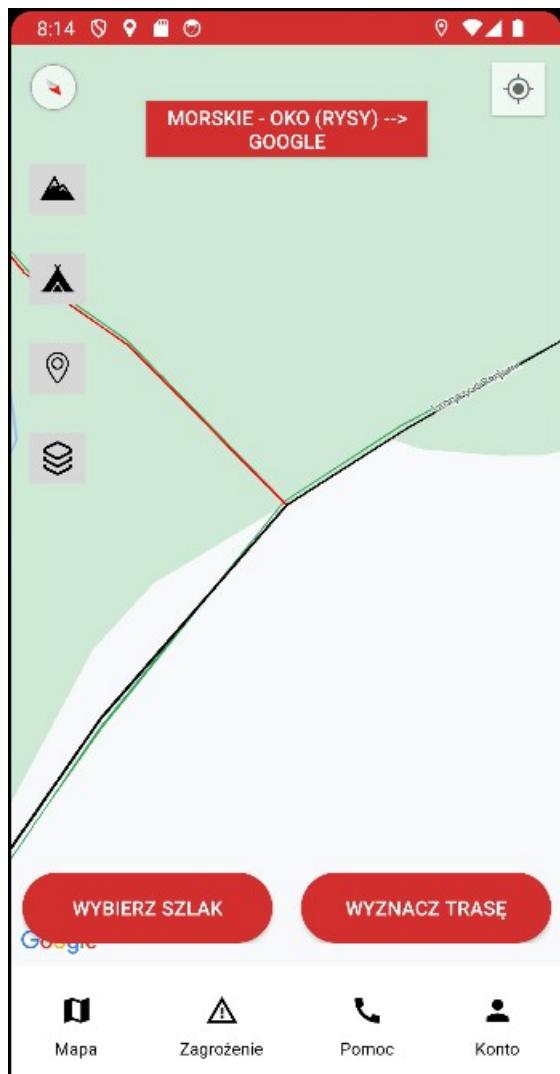
Rysunek 7.10: Widok aplikacji po zatwierdzeniu wybranej przez użytkownika trasy.

Wybieranie szlaku

Po wybraniu opcji “Wybierz szlak” z widoku mapy (rys. 7.6) pokaże się lista szlaków z oznaczeniem koloru, widoczna na rys. 7.11. Użytkownik może przeszukać wszystkie szlaki górskie Tatr, a po kliknięciu przycisku “wybierz”, zostaje nakierowany na początek wybranej trasy, jak zostało przedstawione na rys. 7.12. Na ekranie widoczna jest także kolejna opcja, która umożliwia użytkownikowi nawigację po trasie za pomocą aplikacji Mapy Google.



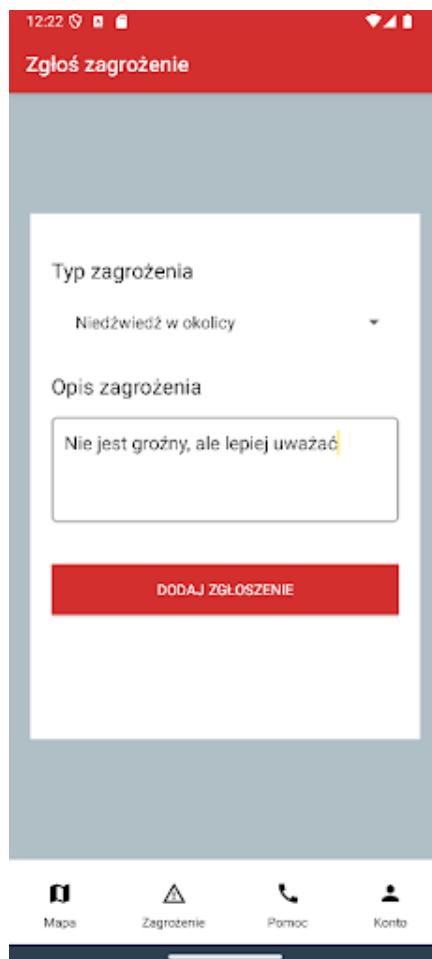
Rysunek 7.11: Lista dostępnych szlaków.



Rysunek 7.12: Widok z początkiem wybranego z listy szlaku.

Widok formularza zgłoszeń zagrożeń

Ekran zgłoszenia pokazany na rys. 7.13 spełnia jedno z podstawowych funkcjonalności aplikacji. To tutaj użytkownik, mając włączoną lokalizację i internet, może poinformować innych o danym zagrożeniu na szlaku, wybierając jego typ z listy i opisując szczegóły. Po naciśnięciu “Zgłoś zagrożenie”, zgłoszenie zostaje od razu wysłane lub w przypadku problemów z internetem, zapisane i wysłane po ponownym połączeniu.



Rysunek 7.13: Widok z formularzem zgłoszeniowym dla zagrożeń.

Widok ekranu z numerami alarmowymi

Ekran aplikacji, pokazany na rys. 7.14 umożliwia użytkownikowi wezwanie pomocy w razie wypadku. Kliknięcie przycisków po lewej stronie spowoduje zadzwonienie na dany numer, natomiast przyciski po prawej pozwalają na skopiowanie numeru telefonu. Jest to bardzo wygodna opcja, która oszczędza użytkownikowi dużo czasu, gdyż nie musi go szukać i ma zawsze pod ręką.



Rysunek 7.14: Widok ekranu z numerami alarmowymi.

7.5 Widoki - administrator

Dodając do dowolnego zarejestrowanego użytkownika rolę administratora, zmieni się zupełnie wygląd jego aplikacji po zalogowaniu. Administrator ma dostęp do kompletnie innych funkcjonalności niż zwykły użytkownik, takich jak dodawanie nowych szlaków do Firebase Storage w formacie .kml, jak i akceptowanie zgłoszeń o zagrożeniach przesyłanych przez użytkowników. Poniżej znajduje się pełny opis wyglądu i funkcjonalności aplikacji użytkownika z rolą "admin".

Widok główny administratora

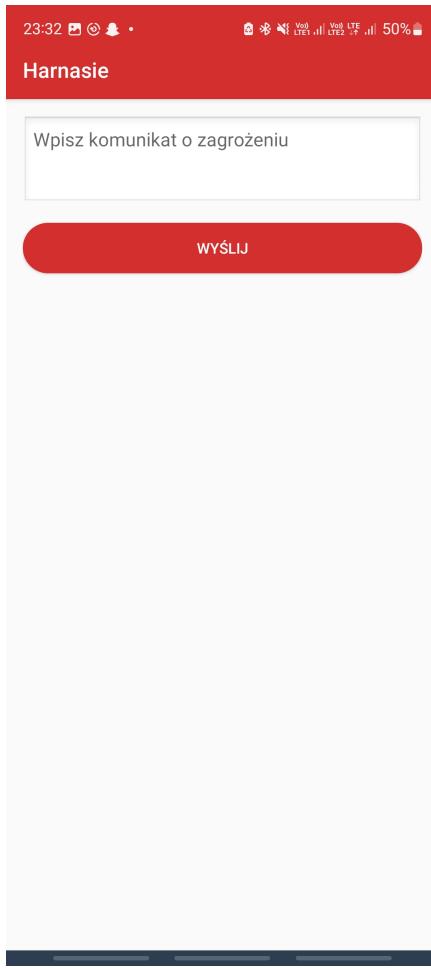
Administrator posiada swoją własną wersję aplikacji. Jego ekran główny 7.15 posiada zestaw przycisków odpowiedzialnych za główne elementy, którymi zarządza. Kolejno od góry widoczne są przyciski "Wyślij powiadomienie ogólne", przenoszący go do widoku krótkiego formularza (rys. 7.16), gdzie może wkleić treść komunikatu od służb tatrzańskich lub napisać własną treść powiadomienia wysyłanego do wszystkich użytkowników, "Pokaż wszystkie zgłoszenia" przenoszący do widoku listy wszystkich zgłoszonych zagrożeń (rys. 7.17) czekających na akceptację, "Dodaj trasę" przenoszący do widoku umożliwiającego dodanie nowego szlaku do bazy (rys. 7.18) oraz "Wyloguj" zamykający bieżącą sesję użytkownika.



Rysunek 7.15: Widok główny po zalogowaniu na konto z uprawnieniami administratora.

Widok formularza komunikatów

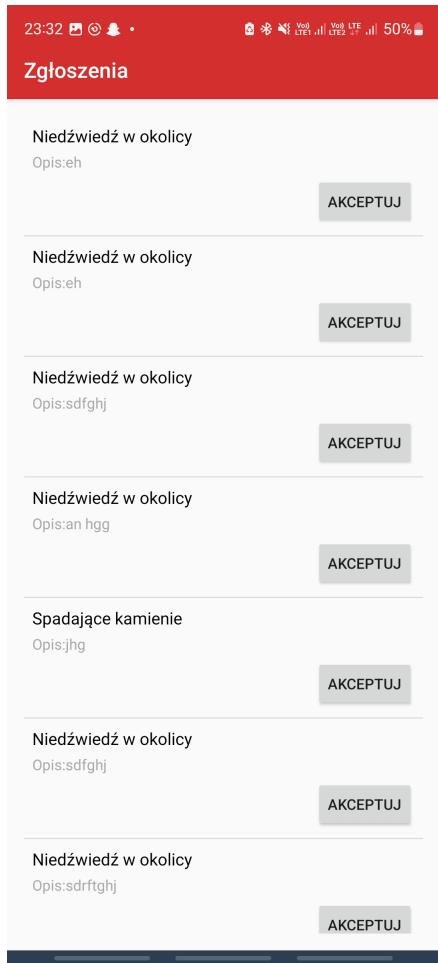
Formularz przedstawiony na poniższym widoku (rys. 7.16) umożliwia administratorowi wysłanie powiadomień do wszystkich użytkowników jednocześnie z komunikatami bezpieczeństwa wydawanymi przez TOPR, GOPR lub TPN najczęściej dotyczących stanu meteorologicznego panującego w Tatrach.



Rysunek 7.16: Widok formularza administratora do przekazywania komunikatów TOPR, GOPR, TPN w formie powiadomienia dla wszystkich użytkowników.

Widok listy zgłoszeń

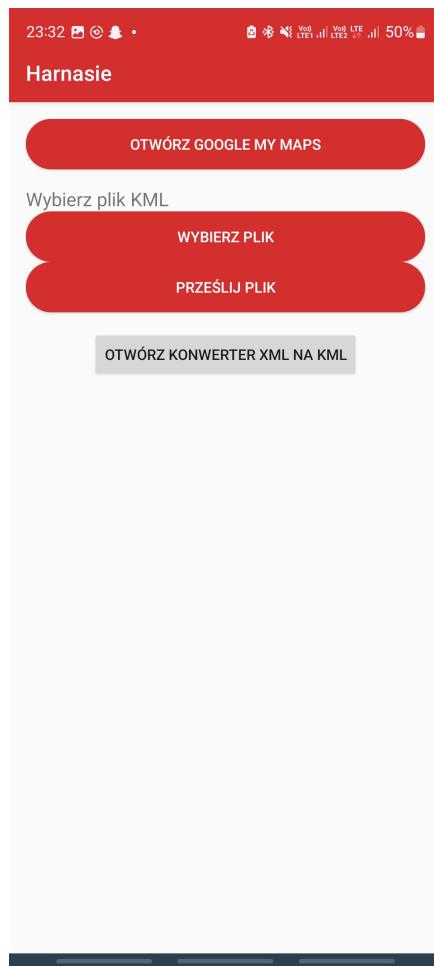
Poniższy widok (rys. 7.17) przedstawia listę zgłoszonych przez użytkowników zagrożeń na trasie, które czekają na akceptację po stronie administratora. Każde zgłoszenie musi mieć swój typ oraz krótki opis, a aplikacja zbiera dodatkowo informacje o aktualnym miejscu użytkownika w momencie wysłania zgłoszenia (nawet offline) oraz jego lokalizację.



Rysunek 7.17: Widok administratora z listą zagrożeń do zaakceptowania.

Widok wspomagający dodanie szlaków do bazy

Poniższy widok (rys. 7.18) przedstawia prosty proces dodawania plików .kml z nowymi szlakami. Najpierw administrator wyznacza przebieg szlaku za pomocą strony Google My Maps, gdzie ma możliwość eksportowania pliku z nowo utworzonym szlakiem do formatu .kml. Następnie może wybrać świeżo przygotowany plik i spróbować go do usługi Firebase Storage. Pojawia się jednak jeden problem - w procesie implementacji tej funkcji oraz testowania jej poprawnego działania okazało się, że urządzenia mobilne mogą zapisywać pliki .kml jako pliki.xml. Dlatego zdecydowałyśmy się dodać kolejny przycisk "Otwórz konwerter xml na kml", który przenosi administratora na przeglądarkowy konwerter plików. Wyeksportowany, przekonwertowany plik administrator wysyła na serwer za pomocą przycisku "Prześlij plik".



Rysunek 7.18: Widok administratora z przeprowadzeniem krok po kroku po procedurze dodania nowych szlaków do bazy dostępnych szlaków.

8 Testy

Przeprowadzenie testów aplikacji zapewnia programistom pewność, że użytkownicy ich systemu nie napotkają żadnych błędów lub niedociągnięć podczas codziennego korzystania z usługi. Testy systemów informatycznych dzielimy na manualne i automatyczne, a te drugie na kolejne podtypy: jednostkowe, integracyjne, wydajnościowe, systemowe, bezpieczeństwa itp. Zdecydowałyśmy przeprowadzić testy jednostkowe wybranych funkcji oraz testy manualne najdłuższych sekwencji, jakie użytkownik jest w stanie wykonać w naszej aplikacji.

8.1 Testy jednostkowe

Testy jednostkowe pozwalają na sprawdzenie działania poszczególnych, pojedynczych funkcji systemu celem weryfikacji wyniku ich użycia.

Ze względu na to, że funkcje w naszej aplikacji skupione są głównie na wyświetlaniu poszczególnych elementów i pobieraniu danych z innych klas, funkcji czy widoków, niewiele testów jednostkowych było możliwych do zrealizowania. Przetestowane zostały jednak funkcje logowania, rejestracji oraz tworzenia linku używanego do wyznaczania trasy w Mapach Google.

Test jednostkowy - logowanie użytkownika

Test ma na celu sprawdzenie poprawności działania funkcji logowania. Do tego przygotowano próbную instancję Firebase Authentication oraz przykładowego użytkownika o danych logowania: user@example.com z hasłem password123. Taki użytkownik nie istnieje w prawdziwej bazie danych systemu, został stworzony w klasie testowej na potrzeby przeprowadzenia tego testu. Treść testu sprawdzającego poprawność logowania przedstawiona jest na listingu 8.1.

Listing 8.1: Test jednostkowy - logowanie

```
@Test
public void testSignIn_Success() {
    try (ActivityScenario<SignInActivity> scenario = ActivityScenario.launch(SignInActivity.class)) {
        scenario.onActivity(activity -> {
            activity.setAuth(mockAuth);

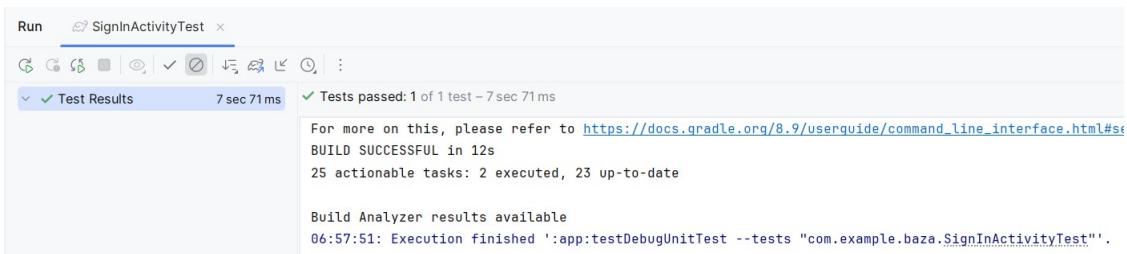
            EditText etEmail = activity.findViewById(R.id.et_email);
            EditText etPassword = activity.findViewById(R.id.et_password);
            Button btnSignIn = activity.findViewById(R.id.btn_login);

            etEmail.setText("user@example.com");
            etPassword.setText("password123");

            when(mockAuth.signInWithEmailAndPassword(s: "user@example.com", s1: "password123"))
                .thenReturn(mockAuthTask);

            btnSignIn.performClick();
            verify(mockAuth).signInWithEmailAndPassword(s: "user@example.com", s1: "password123");
        });
    }
}
```

Zgodnie z założeniami test przebiegł pomyślnie, a więc funkcja działa poprawnie i pozwala użytkownikom na logowanie do aplikacji za pomocą ich danych. Poprawny przebieg testu potwierdza rys. 8.2.



Rysunek 8.1: Wynik testu jednostkowego sprawdzającego poprawność logowania.

Test jednostkowy - rejestracja użytkownika

Test ma na celu sprawdzenie poprawności działania funkcji rejestracji nowego użytkownika w systemie. Do tego przygotowano próbную instancję Firebase Authentication oraz przykładowego użytkownika o danych logowania: test@example.com o nazwie użytkownika TestUser z hasłem password123. Taki użytkownik nie istnieje w prawdziwej bazie danych systemu, został stworzony w klasie testowej na potrzeby przeprowadzenia tego testu. Treść testu sprawdzającego poprawność rejestracji przedstawiona jest na listingu 8.2.

Listing 8.2: Test jednostkowy - rejestracja

```
@Test
public void testSignUp_Success() {
    try (ActivityScenario<SignUpActivity> scenario = ActivityScenario.launch(SignUpActivity.class)) {
        scenario.onActivity(activity -> {
            activity.setAuth(mockAuth);
            activity.setDb(mockFirestore);
            Log.d( tag: "Test", msg: "Firestore instance set: " + mockFirestore);

            EditText etEmail = activity.findViewById(R.id.et_email);
            EditText etPassword = activity.findViewById(R.id.et_password);
            EditText etUsername = activity.findViewById(R.id.et_username);
            Button btnSignUp = activity.findViewById(R.id.btn_register);

            etEmail.setText("test@example.com");
            etPassword.setText("password123");
            etUsername.setText("TestUser");

            when(mockAuth.createUserWithEmailAndPassword(s: "test@example.com", s1: "password123"))
                .thenReturn(mockAuthTask);
            when(mockAuthTask.isSuccessful()).thenReturn( value: true);
            when(mockAuth.getCurrentUser()).thenReturn(mockUser);
            when(mockUser.getUid()).thenReturn( value: "testUid");
            when(mockFirestoreTask.isSuccessful()).thenReturn( value: true);

            doAnswer(invocation -> {
                OnCompleteListener<AuthResult> listener = invocation.getArgument( index: 0);
                listener.onComplete(mockAuthTask);
                return null;
            }).when(mockAuthTask).addOnCompleteListener(any());

            doAnswer(invocation -> {
                OnCompleteListener<Void> listener = invocation.getArgument( index: 0);
                listener.onComplete(mockFirestoreTask);
                return null;
            }).when(mockFirestoreTask).addOnCompleteListener(any());

            btnSignUp.performClick();

            verify(mockFirestore).collection( collectionPath: "users");
            verify(mockCollectionReference).document( documentPath: "testUid");

            ArgumentCaptor<Map<String, Object>> userMapCaptor = ArgumentCaptor.forClass(Map.class);
            verify(mockDocumentReference).set(userMapCaptor.capture());

            Map<String, Object> capturedUserMap = userMapCaptor.getValue();
            assertEquals( expected: "test@example.com", capturedUserMap.get("email"));
            assertEquals( expected: "TestUser", capturedUserMap.get("username"));
            assertEquals( expected: "testUid", capturedUserMap.get("uid"));
            assertEquals( expected: "user", capturedUserMap.get("role"));
        });
    }
}
```

Zgodnie z założeniami test przebiegł pomyślnie, a więc funkcja działa poprawnie i pozwala użytkownikom na zakładanie konta w aplikacji i zapis ich danych. Poprawny przebieg testu potwierdza rys. 8.2.



Rysunek 8.2: Wynik testu jednostkowego sprawdzającego poprawność rejestracji nowego użytkownika.

Test jednostkowy - generowanie URL trasy

Test ma na celu sprawdzenie poprawności generowania URL z trasą o podanym jej początku i końcu. Sprawdzono to za pomocą asercji.

Listing 8.3: Test jednostkowy - generowanie URL

```
@Test
@DisplayName("Url")
public void testGetDirectionsUrl() {
    LatLng origin = new LatLng( latitude: 51.250945, longitude: 22.5747267);
    String expectedUrl = "https://maps.googleapis.com/maps/api/directions/json?origin=" +
        "51.250945,22.5747267&destination=49.47356220575305,20.178220197558403&mode=walking&key=AIzaSyC4KaLnKSYLuF9xCyzudGh8DMCB-6HefJA";
    LatLng destination= new LatLng( latitude: 49.47356220575305, longitude: 20.1782201975584030);
    String actualUrl = MapActivity.getDirectionsUrl(origin, destination);

    assertEquals(expectedUrl, actualUrl);
}
```

Zgodnie z założeniami test przebiegł pomyślnie, a więc funkcja działa poprawnie i generuje poprawne linki do wyznaczonej trasy na bazie jej punktu początkowego i końcowego. Poprawny przebieg testu potwierdza rys. 8.3.



Rysunek 8.3: Wynik testu jednostkowego sprawdzającego poprawność generowania URL z trasą o podanym punkcie początkowym i końcowym.

Powyższe testy przebiegły pomyślnie, funkcje zwróciły oczekiwane wartości i spełniły nasze wymagania.

8.2 Testy manualne

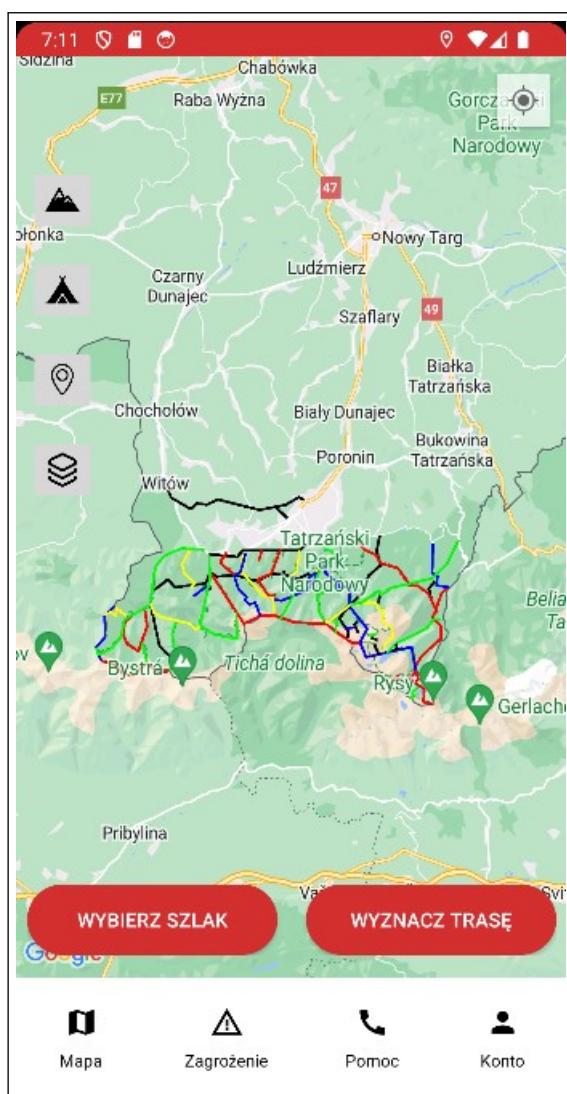
Testy manualne są ręcznym sposobem na sprawdzenie zachowania aplikacji. Tester sprawdza wtedy wszystkie możliwe dla użytkownika opcje i kombinacje kroków, które klient może wykonać choćby przez przypadek. Stosowanie testów manualnych pozwala na wykrycie luk w kodzie i funkcjonalnościach, których programista mógł nie przewidzieć lub przeoczyć. Niewykrycie niektórych niedopatrzeń może nieść za sobą fatalne skutki, jak na przykład

usunięcie całej bazy danych systemu informatycznego, jeśli programista nie zabezpieczy wystarczająco dostępu do treści nieupoważnionym użytkownikom.

W naszej aplikacji głównymi funkcjonalnościami jest możliwość wyznaczenia trasy na mapie oraz zgłaszanie zagrożeń występujących na trasie.

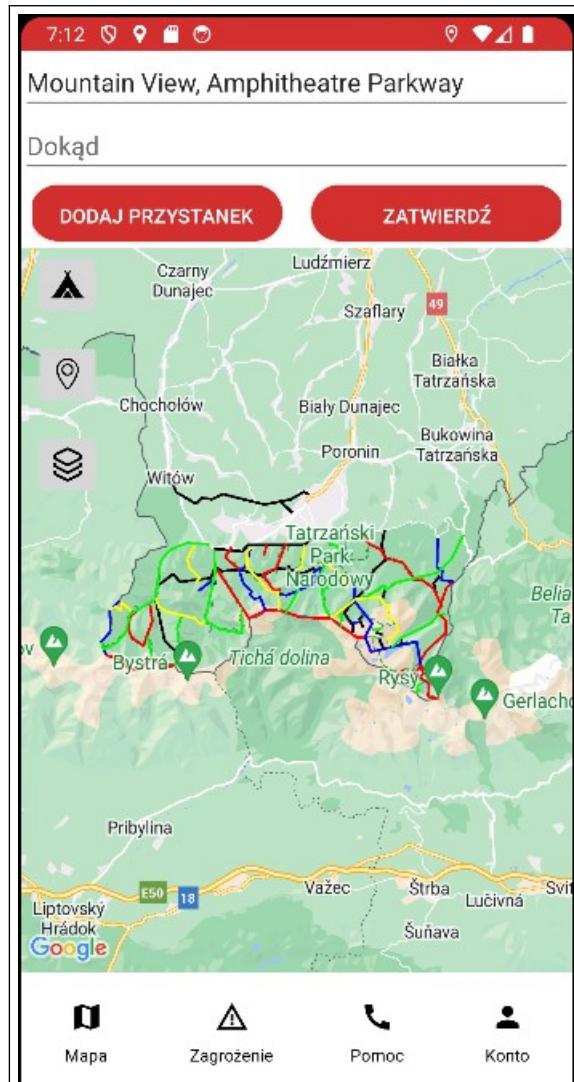
Test manualny - wyznaczanie trasy.

Główną funkcjonalnością opisywanej aplikacji są mapy i usługi z nimi związane. Najważniejszą dostępną opcją jest wyznaczanie trasy z jednego punktu do drugiego i rysowanie jej na mapie tak, aby użytkownik widział dokładnie, gdzie się znajduje. Dlatego niniejszy test sprawdzi poprawność wykonywania tej funkcji.



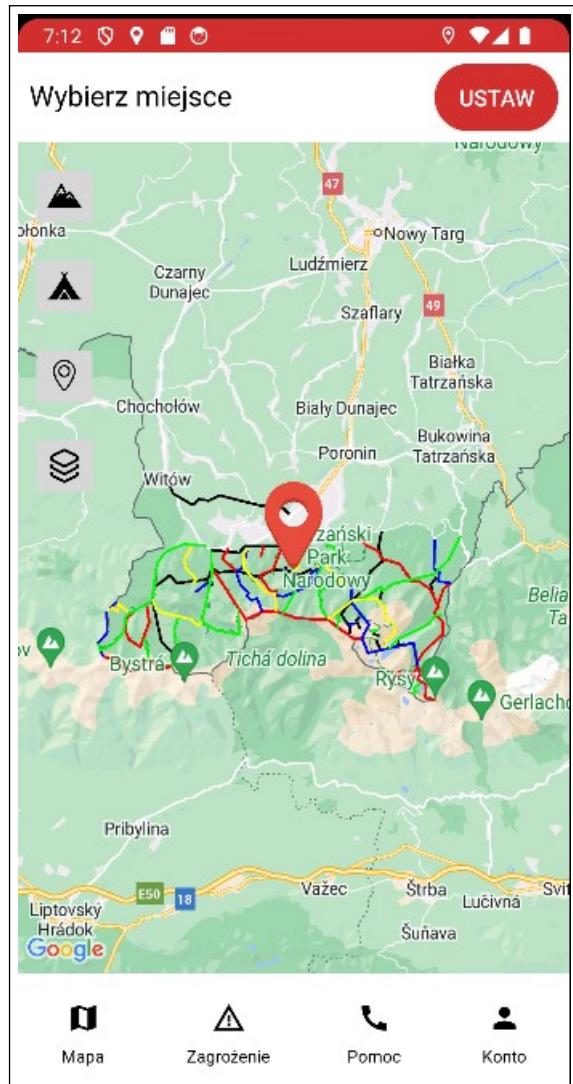
Rysunek 8.4: Widok początkowy map.

Aby przejść do trybu wyznaczania trasy należy kliknąć przycisk "Wyznacz trasę" widoczny na dole ekranu na rys. 8.4.



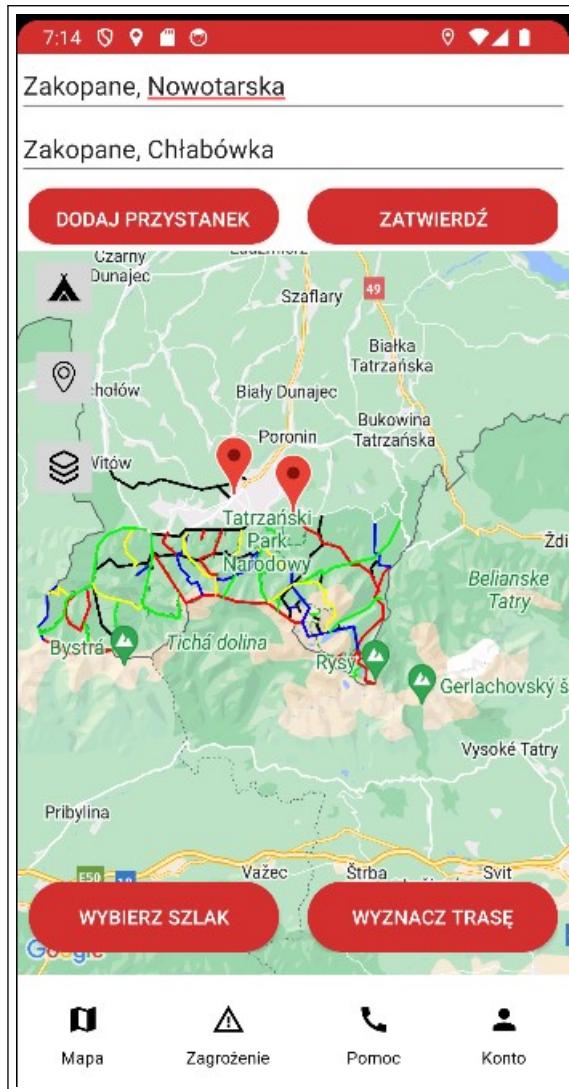
Rysunek 8.5: Wybranie funkcji wyznaczania trasy.

Tryb wyznaczania trasy posiada pola wyświetlające początek i koniec trasy (rys.8.5), które ustawia się za pomocą markera wyświetlanego na mapie, co jest widoczne na rys. 8.6. Ponadto jest opcja dodania przystanku, która wyznacza trasę z uwzględnieniem wybranego punktu na mapie. Na potrzeby testów ten krok został pominięty.



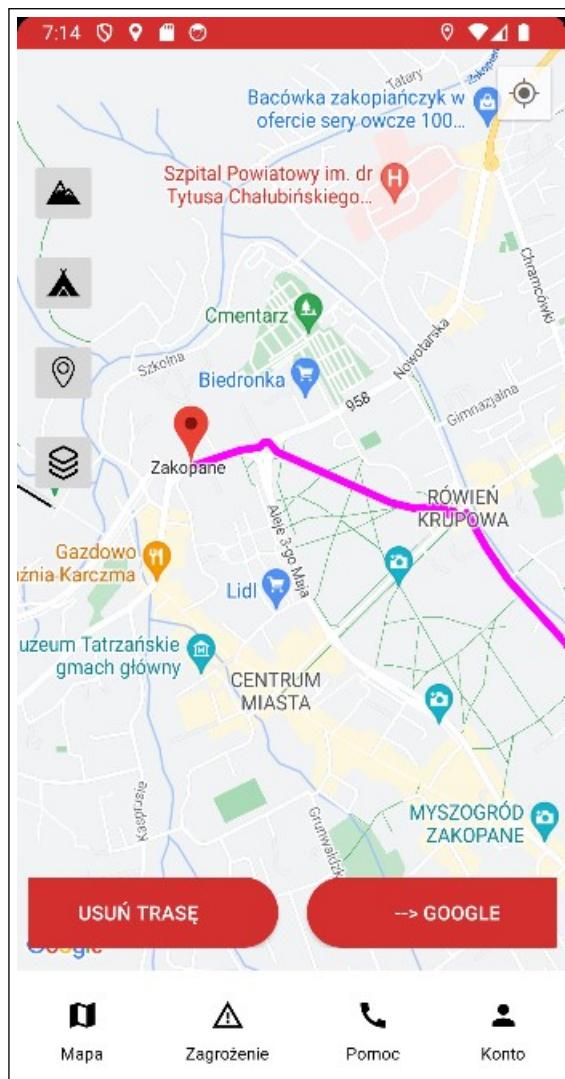
Rysunek 8.6: Wybieranie punktu początkowego i końcowego trasy za pomocą markera.

Na rys. 8.7 ukazane są markery początkowe i końcowe wybranej przez użytkownika trasy oraz słowne określenie ich położenia w górnej części ekranu. Aby trasa została wyznaczona, użytkownik musi potwierdzić swój wybór klikając "Zatwierdź". Aplikacja wtedy wyznacza trasę i rysuje ją na mapie.



Rysunek 8.7: Widoczne markery na początku i końcu trasy.

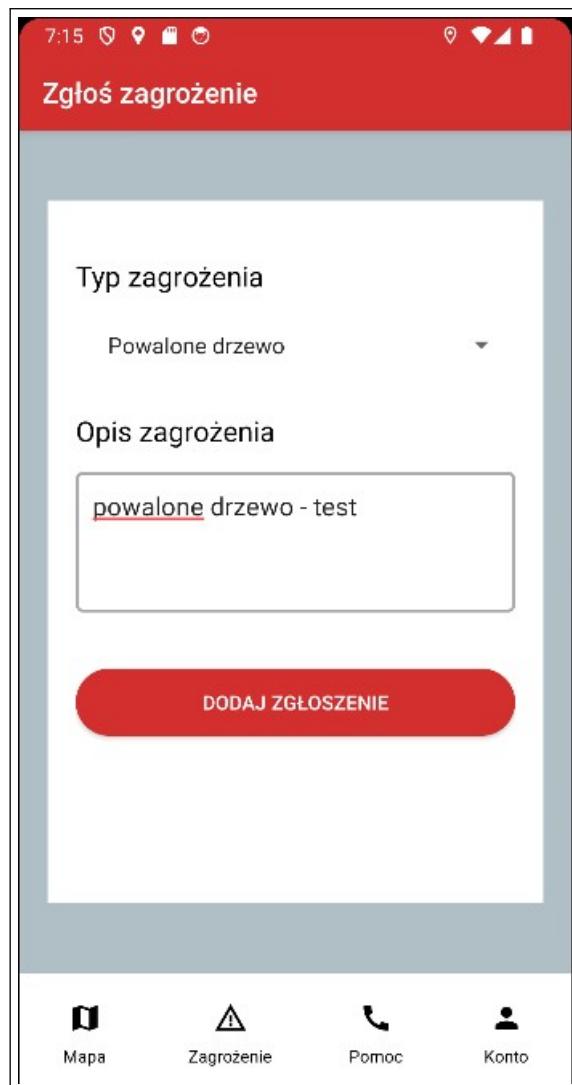
Po zatwierdzeniu wybranej trasy jest ona rysowana na mapie z pomocą koloru różowego, aby była łatwa do odróżnienia od szlaków rysowanych na mapie. Aplikacja od razu nakieruje kamerę na początek trasy (rys. 8.8) i jest gotowa do pokazywania użytkownikowi jego lokalizacji względem wyznaczonej trasy.



Rysunek 8.8: Wyznaczona trasa i ukazany jej początek.

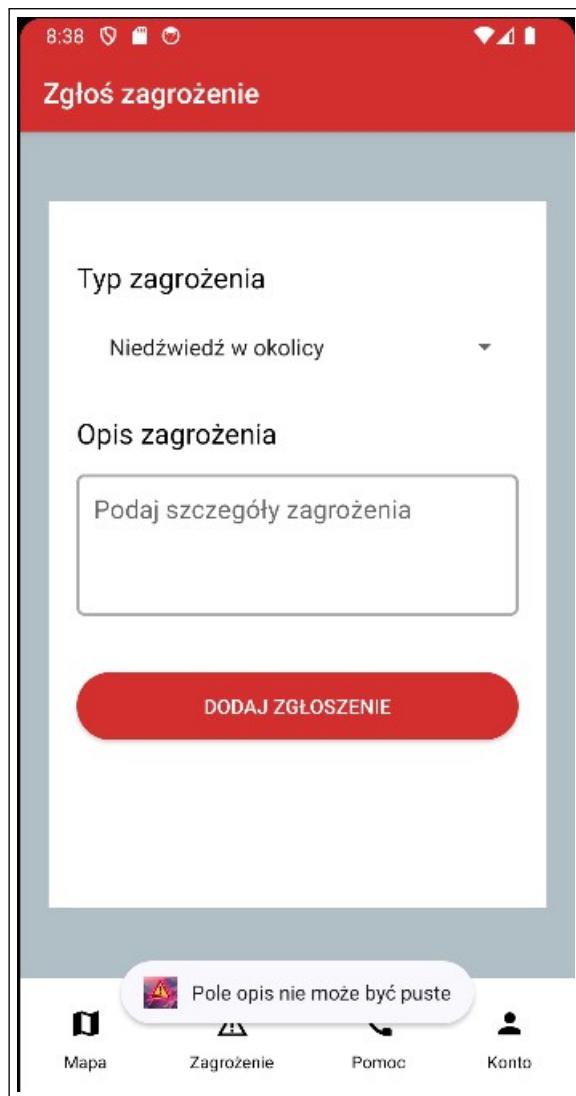
Test manualny - zgłaszanie zagrożeń.

Projektując aplikację autorki pracy miały na uwadze omylność ludzką, dlatego starały się zabezpieczyć system na różnego rodzaju błędy ludzkie i niedopatrzenia. Poniższy test sprawdzi, czy na pewno wszystkie możliwe opcje błędów zostały uwzględnione i zabezpieczone.



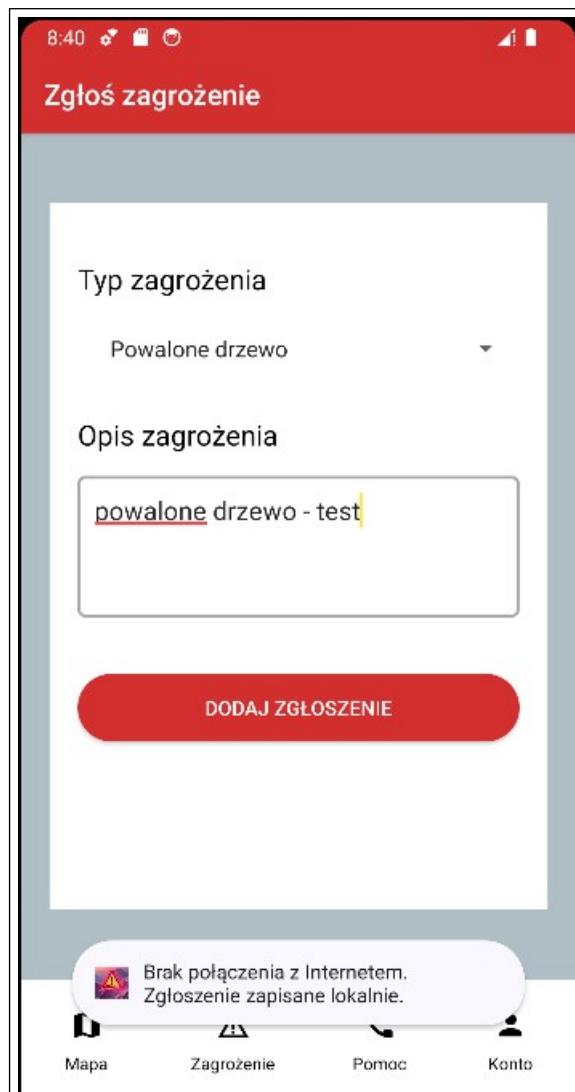
Rysunek 8.9: Formularz zgłaszanego zagrożenia na trasie.

Formularz przedstawiony na rys. 8.9 wymaga od użytkownika podania typu oraz opisu zagrożenia występującego na szlaku. W przypadku braku opisu, na ekranie pojawia się komunikat, widoczny na rys. 8.10, informujący o konieczności jego uzupełnienia.



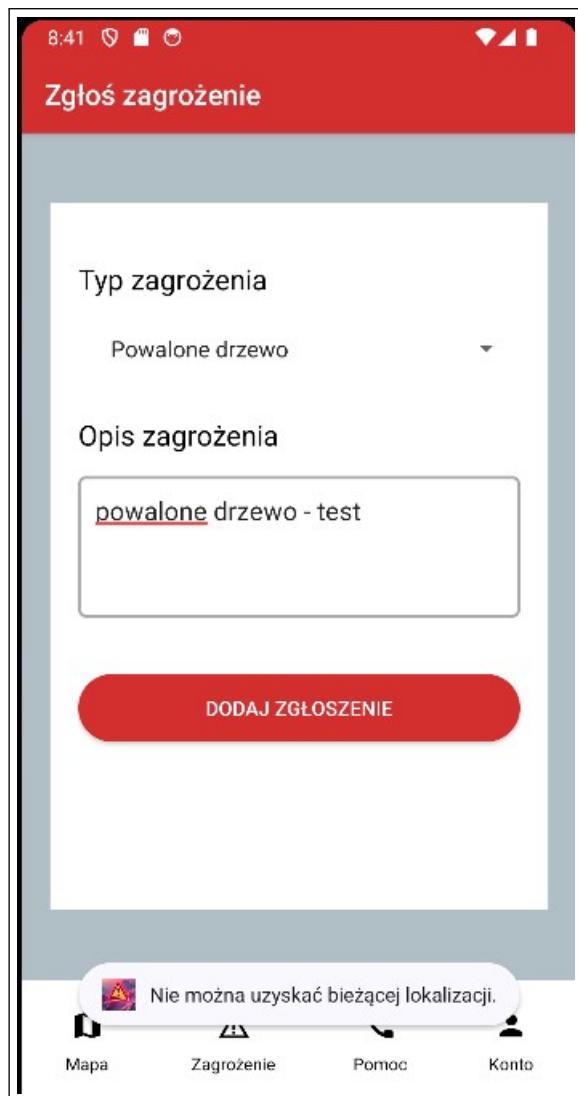
Rysunek 8.10: Komunikat informujący o braku opisu zagrożenia.

W sytuacji, gdy użytkownik chce wysłać zgłoszenie, lecz w danym momencie nie ma połączenia z internetem, aplikacja wyświetla powiadomienie o braku połączenia, tak jak na rys. 8.11.



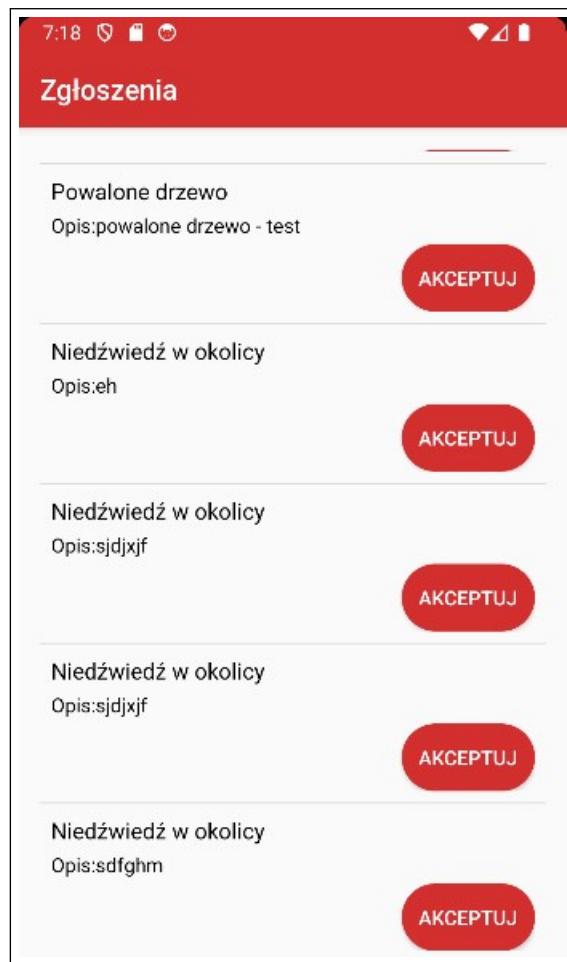
Rysunek 8.11: Komunikat informujący o braku internetu.

Jednak, aby zgłoszenie zostało wysłane, użytkownik musi mieć jeszcze włączoną lokalizację, o czym informuje komunikat przedstawiony na rys. 8.12.



Rysunek 8.12: Komunikat informujący o braku lokalizacji.

Po kliknięciu “Zgłoś zagrożenie” i spełnieniu wszystkich wymagań, formularz zostanie pomyślnie wysłany i wyświetlony na liście zgłoszeń do zaakceptowania w interfejsie administratora (rys. 8.13) .



Rysunek 8.13: Pomyślnie wysłane zgłoszenie i wyświetlenie go w interfejsie administratora.

Opisany test dowódł, iż zgłoszenie zostanie wysłane tylko wtedy, gdy użytkownik poda wszystkie wymagane informacje dotyczące zagrożenia oraz będzie miał włączony internet, a także udostępnioną lokalizację.

9 Wnioski

Główym celem pracy było stworzenie projektu aplikacji, która umożliwi bezpieczne poruszanie się po polskich górach Tatrach. Cel ten został osiągnięty po uprzednim zapoznaniu się z problematyką wyznaczania tras na podstawie obecnej lokalizacji użytkownika oraz powiadomienia go o zagrożeniach występujących w okolicy. Wszystkie założenia zostały dokładnie przeanalizowane, tak aby aplikacja mogła działać, a użytkownik bezpiecznie zdobywał polskie Tatry.

Użyte technologie spełniły nasze oczekiwania i pozwoliły na stworzenie sprawnie działającej aplikacji. Wykorzystanie usług oferowanych przez Firebase, znacznie usprawniło za-programowanie aplikacji, gdyż dzięki nim zyskała ona serwer, a także bazę danych przechowującą m.in. informacje o użytkownikach oraz zgłoszeniach. Podczas tworzenia aplikacji napotkałyśmy kilka problemów, które po wielu próbach zostały rozwiązane. Największym z nich było zmodyfikowanie szlaków na mapie, aby po kliknięciu na nie system wyświetlał o nich informacje. Funkcja ta została zastąpiona listą szlaków, z której użytkownik może wybrać ten, który go interesuje.

Aplikacja “Harnasie” posiada potencjał na dalszy rozwój funkcji przez nią oferowanych. Obszar szlaków może być poszerzony o inne rejony gór, dzięki czemu użytkownik będzie mógł korzystać z aplikacji nie tylko w Tatrach. Baza danych może być także uzupełniona o nową tabelę szlaki. Administrator mógłby wtedy nimi zarządzać, np. włączać i wyłączać z użycia.