Pusta strona 1

Pusta strona 2

Streszczenie

(maksymalnie 1 strona)

(strona nr 3 – numer widoczny)

Xxxxxxxx xxxxxxx xxxxxxxx xxxxxxxxxxxx xxxxxxxx x xxxxxxxxx. Xxxxx xxxxxxxxx xxxx xxxxxx xxxxx. Xxxxxxxx xxxxxxx xxxxxxxx xxxxxxxxxxxx xxxxxxxx x xxxxxxxxx. Xxxxxxxx xxxxxxx xxxxxxxx xxxxxxxxxxxx xxxxxxxx x xxxxxxxxx.

Słowa kluczowe: xxxxxxx, xxxxxxxx

Dziedzina nauki i techniki, zgodnie z wymogami OECD: dziedzina, technika, …

ABSTRACT (maksymalnie 1 strona)

Xxxxxxxx xxxxxxx xxxxxxxx xxxxxxxxxxxx xxxxxxxx x xxxxxxxxx. Xxxxx xxxxxxxxx xxxx xxxxxx xxxxx. Xxxxxxxx xxxxxxx xxxxxxxx xxxxxxxxxxxx xxxxxxxx x xxxxxxxxx. Xxxxxxxx xxxxxxx xxxxxxxx xxxxxxxxxxxx xxxxxxxx x xxxxxxxxx.

Keywords: xxxxxxx, xxxxxxxx

Spis treści

Wykaz najważniejszych oznaczeń i skrótów

REST (Representational State Transfer) - transfer stanu przez reprezentację

JSON (JavaScript Object Notation) - obiektowy sposób zapisu danych

BJSON (binary JSON) - JSON kodowany binarnie

JWT (JSON Web Token) - token w formacie JSON używany w celu zabezpieczenia dostępu aplikacji

API (Application Programming Interface) - interfejs aplikacji dostępny dla programisty

MVP (Model View Presenter) - wzorzec architektoniczny

IDE (Integrated Development Environment) - zintegrowane środowisko programistyczne

DTO (Data State Object) - wzorzec projektowy, rodzaj kontenera na dane

CRUD – (create, read, update, delete) - zestaw czterech podstawowych funkcji dostępu do bazy danych

DAO – (Data access object) - klasa będąca odzwierciedleniem tabeli bazodanowej SQL – (Standard Query Language) - język zapytań do niektórych baz danych (w tym PostgreSQL) JDBC - (Java Database Connectivity) - zestaw funkcji umożliwiających połączenie z bazą danych PostgreSQL

Rozdział 1

Wstęp i cel pracy

xxxxx

Rozdział 2

Przegląd podobnych rozwiązań

Aplikacje do planowania podróży to często spotykane rozwiązanie ułatwiające organizowanie wyjazdów. Oferowane są dla systemów operacyjnych Android oraz iOS odpowiednio w serwisach Google Play i Apple Store. Przetestowano 4 przykładowe aplikacje, których podstawowe dane zawiera tabela 2.1.

Tabela 2.1: Dane testowanych aplikacji

Nazwa aplikacji	**Google Play**		**Apple Store**
	Średnia ocena	Ilość pobrań	Średnia ocena
TripIt: Travel Planner	4.4/5 (53217 ocen)	ponad 1 mln	4.8/5 (122.9K ocen)
Google Trips - Travel Planner	4.1/5 (30473 ocen)	ponad 5 mln	4.4/5 (3100 ocen)
Sygic Travel: Planuj Podróż	4.2/5 (10637 ocen)	ponad 1 mln	4.6/5 (1600 ocen)
Expedia	4.2/5 (172035 ocen)	ponad 10 mln	4.8/5 (551.9K ocen)

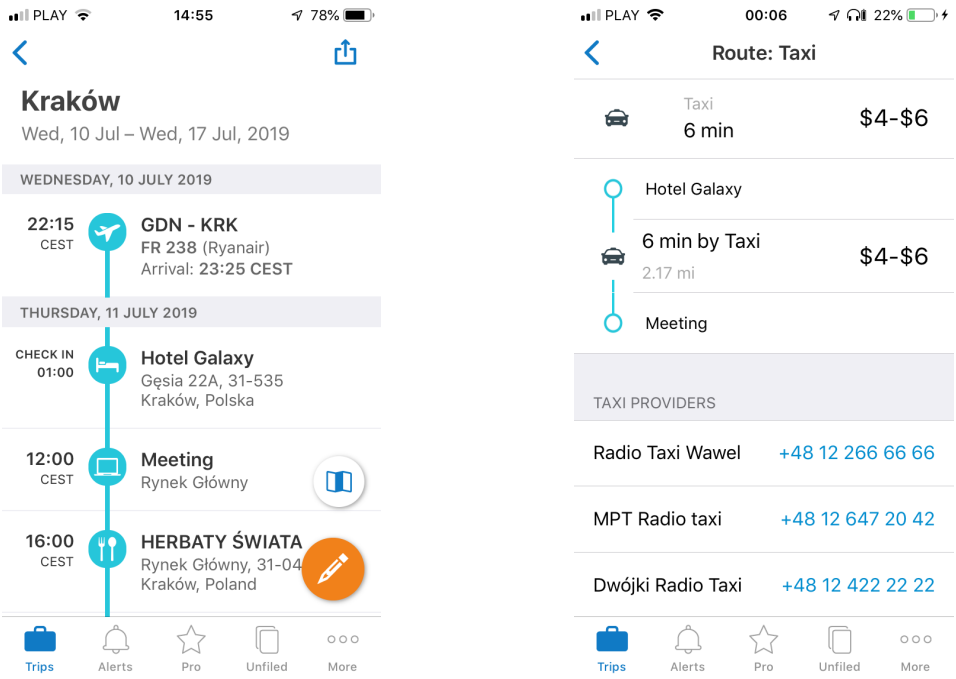
2.1 Aplikacja TripIt: Travel Planner (Dorota Tomczak)

TripIt to aplikacja do planowania podróży, którą przetestowano na urządzeniu z systemem operacyjnym iOS. Dostępna była jedynie w języku angielskim. Podstawowa wersja aplikacji była darmowa, natomiast wersję rozszerzoną (TripIt Pro) oferowano w formie subskrypcji, która kosztowała 49$ rocznie. Oprócz tego istniała też opcja TripIt for Teams, która pozwalała na planowanie podróży grupy osób.

Aby móc korzystać z aplikacji trzeba było najpierw założyć konto w serwisie TripIt, co jednak pozwalało na dostęp do wszystkich swoich planów podróży na różnych urządzeniach. Opcjonalne było natomiast zezwolenie na automatyczne przesyłanie wiadomości e-mail ze swojej skrzynki pocztowej na adres plans@tripit.com. Po umożliwieniu tej opcji, TripIt kilka razy dziennie sprawdzało skrzynkę użyt-

kownika, by następnie pobrać z niej informacje o rezerwacjach lotów, hoteli, samochodów i dodać je do planu podróży.

Stworzenie nowego planu podróży było bardzo proste. Wymagało podania miasta docelowego wyjazdu oraz zakresu dat, w których podróż miała się odbyć. Dodatkowo można było dodać ogólną nazwę wyprawy oraz jej opis. Po zalogowaniu do aplikacji na ekranie widoczne były nadchodzące podróże, a po przełączeniu zakładki również te już odbyte. Obok informacji o nazwie bądź celu podróży, jej terminie i czasu trwania wyświetlało się zdjęcie przedstawiające miasto docelowe podróży, albo zdjęcie samolotu dla miast, których aplikacja zdjęć nie posiadała. Każdą podróż można było usunąć, edytować, scalić z inną oraz dodać do niej plan.



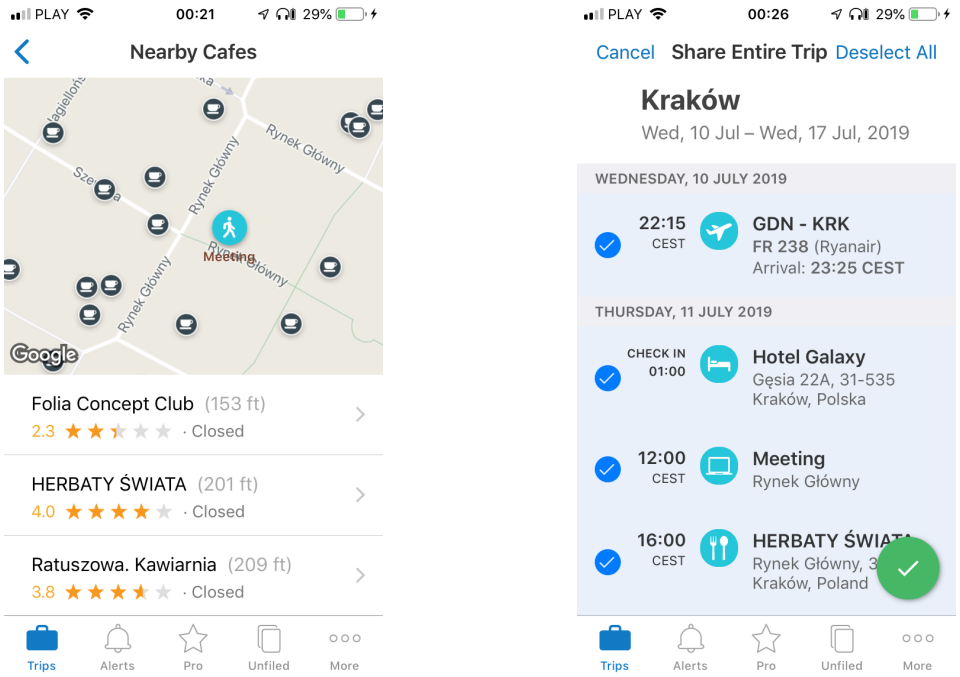
(a) Plan podróży. *(b) Opcje transportu taksówką.*

Rysunek 2.1: TripIt: Travel Planner.

Oprócz wspomnianego wcześniej automatycznego dodawania rezerwacji i planów przez aplikację po pobraniu tych informacji ze skrzynki pocztowej, można to też było zrobić manualnie. TripIt oferowało 18 różnych elementów, które można było dodać do planu, między innymi: lot (ang. flight), kwatera (ang. lodging), wynajem samochodu (ang. car rental), teatr (ang. theater), rejs (ang. cruise), transport (ang. transportation) i ogólna aktywność (ang. activity). Poszczególne elementy umożliwiały dodanie różnych informacji o planowanej podróży, jednak każdy z nich wymagał dodania daty i godziny do danej aktywności, aby aplikacja mogła je później zorganizować chronologicznie. Przy wpisywaniu niektórych informacji aplikacja podpowiadała jakie słowa mógł mieć na myśli użytkownik, co bywało pomocne. Było tak na przykład w przypadku wpisywania nazwy linii lotniczych i nazwy restauracji, a także w przypadku wybrania pola przeznaczonego na podanie adresu. Była to usługa dostarczana przez firmę Google. Po wybraniu takiego zasugerowanego słowa TripIt często samodzielnie uzupełniał niektóre informacje, takie jak np. adres strony internetowej danego obiektu, czy godzina odprawy samolotowej.

Po dodaniu nowego elementu do planu podróży można go było edytować, usunąć, przenieść do innego planu lub uzupełnić o szczegóły, które nie były dostępne przy tworzeniu np. informacje o rezerwacji (ang. booking info), zdjęcia (ang. photos), uczestnicy (ang. attendees). Dla każdego elementu dostępny był również nawigator (ang. navigator), który po wybraniu miejsca startowego i docelowego wyszukuje możliwe opcje transportu z szacowanym czasem ich trwania i kosztem (w dolarach). Po wybraniu transportu samochodem lub pieszo, aplikacja TripIt przekierowywała do aplikacji Google Maps, gdzie można było zobaczyć wyznaczoną trasę.

Ciekawą funkcją było pozyskiwanie informacji o okolicy (ang. neighborhood info), w której znajdował się dodany obiekt. Usługa ta była dostarczana przez GeoSure i udostępniała wskaźniki między innymi z takich kategorii jak: ogólne bezpieczeństwo (ang. overall safety), ryzyko doznania krzywdy fizycznej (ang. physical harm), ryzyko kradzieży (ang. theft). Poza tym TripIt oferowała też wyświetlenie miejsc, będących w pobliżu obiektu z kilku kategorii: restauracje, kawiarnie, bary, bankomaty, parkingi. Wyszukane miejsca pokazywały się na mapie, a po wybraniu jednego z nich można było między innymi zobaczyć godziny otwarcia oraz dodać je do swojego planu podróży.



(a) Wyszukanie pobliskich kawiarni. *(b) Udostępnianie planu podróży.*

Rysunek 2.2: TripIt: Travel Planner.

Każdy plan lub element podróży można było udostępnić lub zapisać w formacie txt. Wybierając opcję Invite Others to View Trip, plan podróży wysyłano na wybrany adres mailowy (który może zostać wyszukany poprzez kontakty) osoby, której można nadać jedno z trzech uprawnień: can view, can edit, can edit and is traveling. W tak wysłanej wiadomości znajdował się link do planu podróży wraz z mapą, które można było wyświetlić w przeglądarce lub aplikacji. W przeglądarce wyświetlała się dodatkowo informacja o pogodzie na każdy dzień podróży. Osoby, którym nadano odpowiednie uprawnienia i posiadały one konto w serwisie TripIt, mogły edytować udostępniony im plan podróży.

W aplikacji można było przechowywać informacje o swoich dokumentach i kontaktach, które mogły się przydać w razie nieprzewidzianych sytuacji. Dane te były dostępne po wprowadzeniu 4-cyfrowego pinu, który należało uprzednio ustawić. W ustawieniach można było włączyć synchronizację kalendarza na naszym urządzeniu z aplikacją TripIt, co powodowało automatyczne dodawanie informacji o planie podróży z aplikacji do kalendarza. Dodatkowo w aplikacji można było obejrzeć statystyki ze wszystkich swoich podróży, takie jak liczba przebytych kilometrów, liczba odwiedzonych krajów i inne.

Opisane wyżej funkcje dotyczyły darmowej wersji aplikacji TripIt. Wersja Pro rozszerzała tą wersję przede wszystkim o rozbudowany system powiadomień oraz mechanizmów śledzących (ang. trackers). Niektóre z tych udogodnień to: powiadomienia o dostępności lepszych miejsc w samolocie, przypomnienia o konieczności odprawy 24 godziny przed wylotem, wyszukiwanie alternatywnych połączeń, wysyłanie informacji o locie w czasie rzeczywistym podczas całej podróży, udostępnianie interaktywnych map lotniska. Wersję TripIt Pro można było przetestować za darmo przez 30 dni.

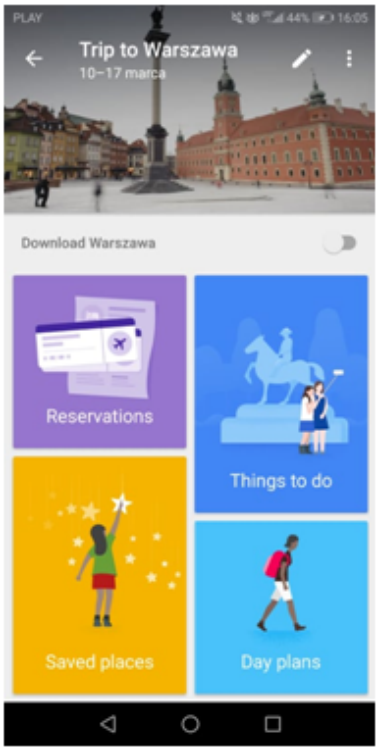
Aplikacja TripIt oferowała wiele możliwości w planowaniu swoich podróży, co pozwalało na stworzenie bardzo szczegółowych planów. Wiele informacji trzeba było wprowadzać samodzielnie, jednak aplikacja często podpowiadała jakie słowa mógł mieć na myśli użytkownik, co znacznie ułatwiało i przyspieszało ten proces. Dodatkowym ułatwieniem była funkcja automatycznego dodawania danych o rezerwacjach, które mogły być pobierane ze skrzynki pocztowej. Dużą zaletą aplikacji była również opcja wyznaczania trasy między dwoma punktami, jednak koszt takiej wyprawy podawany był tylko w dolarach amerykańskich bez względu na kraj, w którym odbywała się podróż i miejsce zamieszkania użytkownika. TripIt pozwalała w łatwy sposób wyszukać sąsiednie obiekty, takie jak restauracje, czy parkingi, jednak brakowało tu atrakcji turystycznych, a odległość od tych obiektów podawana była w stopach (ft) i nie dało się tego zmienić. Z pewnością przydatną funkcją była możliwość udostępnienia stworzonego przez siebie planu innym osobom, jednak by wysłać link ze sformatowanym planem można było to zrobić jedynie na adres mailowy, a nie przez inne środki komunikacji, gdzie wysłany plan był dostępny wyłącznie w formie tekstowej. Wadą aplikacji była jej dostępność jedynie w języku angielskim. Ponadto cena wersji PRO, która wprowadzała szereg dodatkowych udogodnień, była wysoka i mogła odstraszać potencjalnego użytkownika.

2.2 Aplikacja Google Trips - Travel Planner (Karolina Makuch)

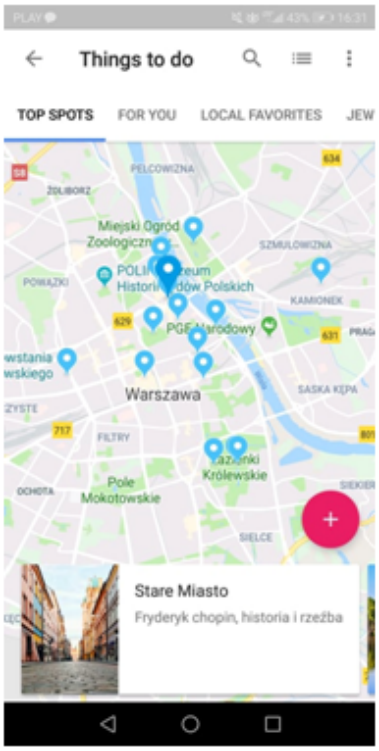
Aplikacja Google Trips – Travel Planner była dostępna na urządzenia mobilne z systemem iOS oraz Android. Była jedną z popularniejszych aplikacji w Sklepie PLAY. Google Trips była także dostępne w Apple Store.

W celu korzystania z Google Trips należało posiadać konto Google. Umożliwiło to dostęp do planu podróży z różnych urządzeń. Niestety aplikacja miała jeden wariant językowy (język angielski).

Na samym początku należało wybrać miasto docelowe. Po wyszukaniu miasta, użytkownik miał możliwość nazwania swojej wycieczki oraz dodawania poszczególnych etapów podróży, uwzględniając przedział czasu poświęcony na każdy z nich.



(a) Tworzenie nowej podróży



(b) Przeglądanie mapy z polecanymi obiektami

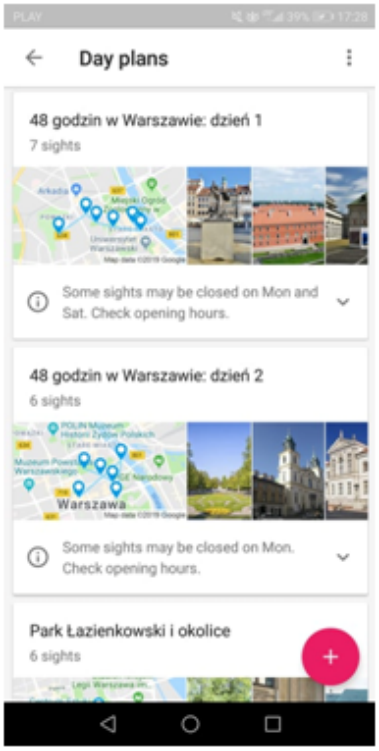
Rysunek 2.3: Google Trips - Travel Planner.

Aplikacja umożliwiała zapisywanie rezerwacji (ang. Reservations) lotu (ang. Flight), hotelu (ang. Hotel), pociągu (ang. Train), autobusu (ang. Bus), samochodu (ang. Car rental) oraz restauracji (ang. Restaurant). W zależności od rodzaju transportu użytkownik uzupełniał odpowiednie informacje takie jak: skąd chce się wybrać, dokąd, kiedy, o której godzinie, numer telefonu, numer potwierdzający,miejsca pasażerów oraz rodzaj transportu. Google Trips posiadał także Notatnik posiadający ograniczenie do 10 000 słów.

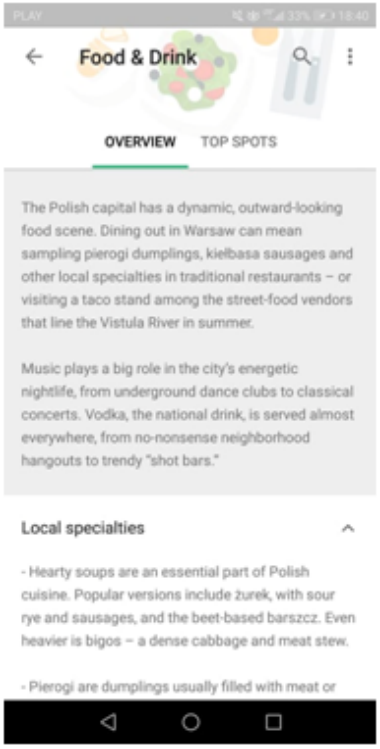
Kolejną opcją była możliwość przejrzenia oraz zapisania dostępnych atrakcji (ang. Things to do). Zostały one podzielone na kilka podkategorii na przykład : polecane (ang. Top Spots), preferowane (ang. For you), parki (ang. Parks & gardens), muzea (ang. Museums), w pomieszczeniu (ang. Indoors), na świeżym powietrzu (ang.outdoor), przyjazne dzieciom (ang. Kids friendly) oraz okoliczne atrakcje (ang. Farther away). Po wybraniu interesującej użytkownika atrakcji użytkownik mógł zobaczyć fotografię, która została wykonana atrakcji,a także miał możliwość przeczytania opisu oraz recenzji użytkowników. Umieszczony był także adres, numer telefonu oraz adres strony internetowej. Aplikacja umożliwiała także nawigację do danej atrakcji. Po naciśnięciu mapy(górny prawy róg) użytkownik mógł zobaczyć lokalizację atrakcji na planie miasta. Daną atrakcję można było zapisać na jednej z domyślnych list: ulubione(ang.favorites), planowane do odwiedzenia (ang. Want to go) oraz ocenione (ang.starred place). Istniała także możliwość stworzenia własnej listy poprzez podanie nazwy.

Następny kafelek umożliwiał przejrzenie zapisanych wcześniej atrakcji oraz wyświetlenie ich w wspomnianych wcześniej listach.

Aplikacja układała także sugerowane plany dnia (ang. Day plans). Każdy z nich otrzymał własną nazwę. Składał się z mapy,a także ilości oraz odnośników do poszczególnych atrakcji. Aplikacja udzielała użytkownikowi wskazówek odnośnie godzin i dni otwarcia danych atrakcji, ile czasu zazwyczaj spędzają tam zwiedzający, a także jak dużo czasu zajmie przejście do kolejnego punktu z planu.



(a) Przeglądanie planu dnia



(b) Przeglądanie informacji na temat regionalnej kuchni

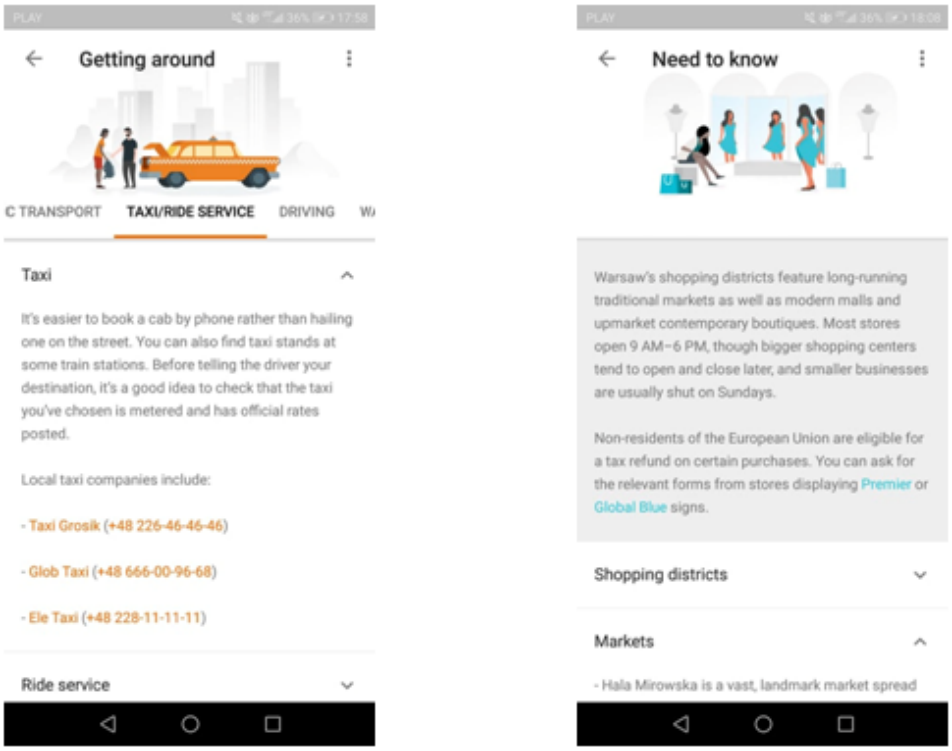
Rysunek 2.4: Google Trips - Travel Planner.

Korzystanie z aplikacji umożliwiało użytkownikom otrzymanie zniżek na poszczególne usługi na przykład: taksówki (pre-booked taxis) czy wypożyczenie samochodu (car hire). Aplikacja posiadała także informacje na temat lokalnej kuchni (kafelek Food & Drink). Zawierał on krótki opis charakterystycznej dla danego rejonu kuchni, wypunktowane lokalne specjalności kuchni(ang.local specialities), polecane miejsca do jedzenia „na mieście" (ang. Dining out) , a także wskazówki na temat '"nocnego życia" (ang. Nightlife). Zakładka „Top Spots" zawierała najbardziej polecane miejsca w poszczególnych kategoriach: ekskluzywne posiłki (ang.high-end dining), budżetowe posiłki (ang. on a budget), miejsca przyjazne rodzinom (ang.family-friendly), śniadania i przekąski (ang. breakfast & brunch), obiady(ang.lunch),posiłki wegetariańskie (ang.vegetarian-friendly),bary (ang.vodka bars), piwowarnie kraftowe (ang.craft beer) oraz kawiarnie tematyczne(ang.themed cafes). Po wybraniu danej kategorii istniała opcja sortowania propozycji według odległości. Można było także przefiltrować wyniki ze względu na godziny otwarcia (ang. Open now) a także wcześniej zapisanych pozycji (ang. only saved places).

Kafelek Zwiedzaj (ang. getting around) umożliwiał sprawdzenie poszczególnych kategorii transportu: on arrival (taxi, ride service, train, bus), public transport (ticket, metro, bus, train, tram), taxi (ang. taxi/ride service), driving, walking & biking. Każda kategoria była wystarczająco szczegółowo opisana.

Zawierała informacje na temat cen, możliwości dojazdu, czasu trwania, możliwych korkach oraz na przykład numerów telefonów do taksówek.

Aplikacja zawierała także wskazówki związane z zakupami (ang. Need to know). Informowała użytkownika o najpopularniejszych godzinach otwarcia sklepów, gdzie znajduje się najwięcej sklepów (ang. Shopping districts). Można było tam także znaleźć przykładowe sklepy (ang. Markets) oraz galerie handlowe (ang. Malls).



(a) Przeglądanie możliwości transportu

(b) Przeglądanie przydatnych informacji

Rysunek 2.5: Google Trips - Travel Planner.

Zaplanowana przez użytkownika wycieczka została automatycznie przypisana do konta Google. Istniała możliwość udostępnienia jej znajomym. Można było połączyć kilka wycieczek. Po pobraniu informacji na temat danej wycieczki, istniała możliwość powrotu do planu, gdy telefon jest offline.

Niestety aplikacja była dostępna tylko w języku angielskim. Posiadała także kilka innych wad. Podczas wyszukiwania odpowiedniego hotelu aplikacja wyświetla listę miejsc. Przykładowo przy wyszukiwaniu hotelu można było wybrać sklep meblowy. Brakowało także podpowiedzi przy niektórych polach np. cartype. Aplikacja nie zawsze dodawała informacje z serwisu gmail. Nie można było samemu dodać atrakcji za pomocą informacji z Google Maps,a także dodać ręcznie maili z rezerwacjami.

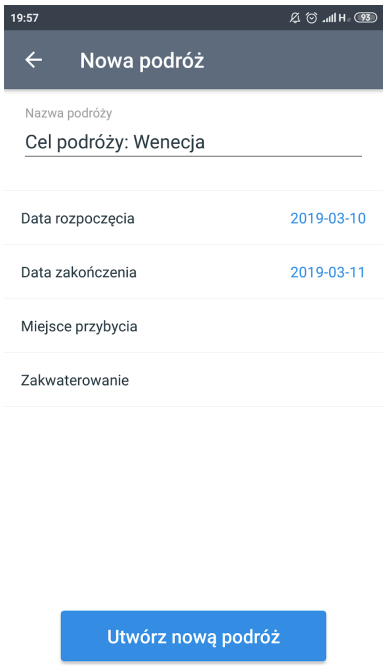
2.3 Sygic Travel - Planuj podróż (Anna Malizjusz)

Planowanie podróży umożliwia również aplikacja Sygic Travel: Planuj Podróż, która została pobrana z serwisu Google Play. Jej duża zaleta to możliwość korzystania z większości funkcji w języku polskim

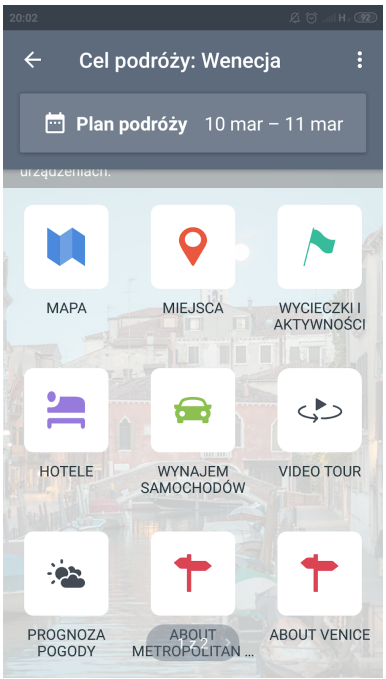
oraz brak konieczności tworzenia konta. Jedyną opcją niedostępną dla niezalogowanego użytkownika była synchronizacja z innymi urządzeniami.

Podstawowymi informacjami potrzebnymi do rozpoczęcia planowania podróży były cel, data rozpoczęcia oraz data zakończenia (rys. 2.6a). Opcjonalnie można było podać miejsce przybycia oraz zakwaterowania. Wygodny wybór daty umożliwił wyświetlany kalendarz, a miejsca należało wybrać z podanej listy. Istniała również możliwość filtrowania podanych miejsc po nazwie.

Po stworzeniu podróży został wyświetlony interfejs oferujący różne opcje (rys. 2.6b). Po wybraniu mapy można zobaczyć mapę miejsca docelowego z zaznaczonymi atrakcjami. Listę atrakcji można też zobaczyć wybierając opcję miejsca. W obu przypadkach można wybrać kategorie, np. zwiedzanie, zakupy, relaks oraz tagi, np. zwierzęta mile widziane (ang. pets allowed), a także możliwe było dodanie miejsca do odwiedzenia w danym dniu. Dodatkowo zaprezentowano szacunkowy czas dotarcia do celu piechotą z miejsca zakwaterowania. Ta część aplikacji nie została przetłumaczona i była dostępna tylko w języku angielskim. Analogicznie można było przeglądać dostępne kwatery po wybraniu opcji hotele. Dostępne były typowe opcje pomocne w poszukiwaniu zakwaterowania, takie jak wybór średniej oceny, przyznanych gwiazdek, typu zakwaterowania czy udogodnień, np. darmowe Wi-Fi (ang. free hotel Wi-Fi) lub klimatyzacja (ang. air conditioning).



(a) Tworzenie nowej podróży.



(b) Menu główne.

Rysunek 2.6: Sygic Travel - Planuj podróż.

Innego typu funkcjami aplikacji były wycieczki i aktywności oraz wynajem samochodów. Pierwsza opcja umożliwiała wyszukiwanie wycieczek, a także ich rezerwację oraz płatność. Druga wyszukiwała oferty wynajmu samochodów, wyświetlała wszystkie dostępne informacje oraz umożliwiała rezerwację. W obu funkcjach udostępniono możliwość filtrowania wyników. Poważną wadą wyszukiwarki samocho-

dów do wynajęcia stanowił niejasny komunikat o błędzie w przypadku podania dat z przeszłości, który sugerował sprawdzenie zaznaczonych pól nie wskazując możliwego rozwiązania problemu.

Kolejna ciekawa opcja to wycieczka wideo (ang. video tour), która była dostępna tylko dla bardziej popularnych celów podróży. Po wybraniu tej opcji użytkownikowi wyświetlany był film wraz z angielskim komentarzem z jednodniowej wycieczki po wybranym mieście. Prezentowano charakterystyczne i warte odwiedzenia zabytki i atrakcje turystyczne, a użytkownikowi umożliwiono ingerowanie w pokazywany obraz. Filmy oferowały widok 360° po obróceniu telefonu.

3 kolejne funkcje były czysto informacyjne, lecz przydatne. Pokazywały prognozę pogody na najbliższe 14 dni, a także informacje o mieście i okolicy. Obszerność tych danych różniła się w zależności od popularności celu podróży, ale mogła stanowić dobrą podstawę dla nieobeznanych z miejscem użytkowników. Informacje w języku angielskim zostały zaczerpnięte z serwisu Wikivoyage, który można otworzyć w aplikacji oraz przeglądarce internetowej.

Po zakupie Sygic Travel Premium za 88,99 zł pojawiały się dodatkowe funkcjonalności, takie jak zapisanie mapy miejsca docelowego do użytku offline. Kolejną opcją było wyświetlenie informacji o zabytkach, hotelach i atrakcjach, których odwiedzenie proponowała aplikacja. Darmowa wersja oferowała wszystkie najbardziej potrzebne opcje, więc zakup rozszerzonej wersji nie był koniecznością, a jedynie nieznacznym zwiększeniem możliwości aplikacji.

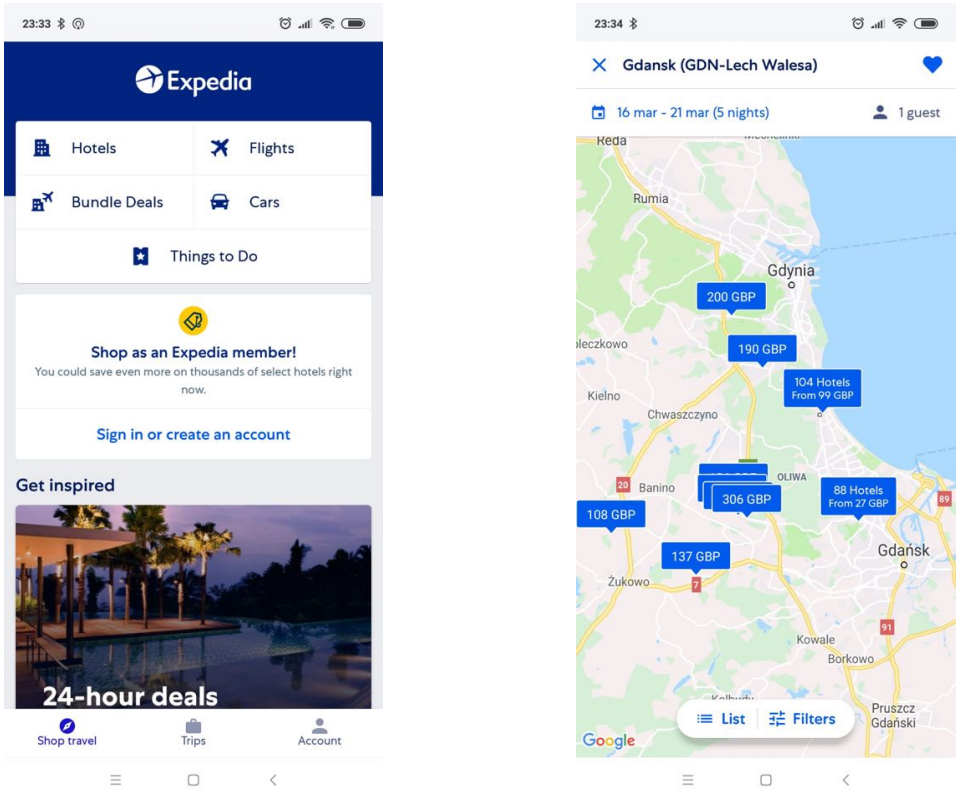
Aplikacja została zintegrowana z innymi produktami firmy Sygic. Przy każdym miejscu oferowała możliwość nawigacji przy pomocy Sygic GPS Navigation & Maps, a rezerwacja wycieczek i samochodów była wspierana przez witrynę Sygic Travel, która była dostępna również z komputera z przeglądarki internetowej.

Sygic Travel to spełniające podstawowe funkcje narzędzie ułatwiające planowanie podróży. Nie oferowała wielu możliwości, jednak warto zauważyć, że większość z nich była dostępna w wersji podstawowej i nie zmuszała użytkownika do zakupu stosunkowo drogiej wersji premium. Udostępniała bogaty wybór możliwych do odwiedzenia miejsc w wybranym celu podróży, a sam cel mógł należeć do egzotycznych, np. Udaipur w Indiach. Podstawowe informacje, takie jak mapa, atrakcje turystyczne czy prognoza pogody były zawsze dostępne.

2.4 Expedia (Magdalena Solecka)

Kolejna aplikacja do planowania podróży to Expedia, dostępna na urządzeniach z systemem iOS oraz Android. Korzystanie z niej było możliwe po założeniu konta użytkownika. Wśród dostępnych tłumaczeń nie znajdował się język polski. Aplikacja składała się z czterech głównych funkcjonalności: rezerwacja noclegu (ang. Hotels), rezerwacja lotu (ang. Flights), rezerwacja samochodu (ang. Cars) oraz wyszukiwanie ciekawych miejsc w okolicy (ang. Thing to Do). Nie było możliwości wyszukania restauracji w rejonie ani transportu naziemnego, koleją lub autobusem.

W celu wyszukania noclegu należało podać następujące dane: datę zameldowania i wymeldowania, miasto docelowe podróży, liczbę pokoi oraz osób do zakwaterowania. Zaprezentowane przez aplikację wyniki mogły zostać posortowane po cenie lub recenzjach użytkowników, a także przefiltrowane wzglę-



(a) Menu główne. *(b) Wyszukiwanie noclegu.*

Rysunek 2.7: Expedia.

dem darmowego anulowania rezerwacji, śniadaniu wliczonym w cenę pobytu, klimatyzacją czy Wi-Fi. Wygodnym rozwiązaniem była mapka z naniesionymi na nią punktami z ceną w miejscu, w którym znajdował się oferowany pokój, co umożliwiało łatwiejszą ocenę odległości od centrum.

W przypadku rezerwacji miejsca w samolocie można było wyszukać lot w jedną lub obie strony. Należało podać miejsce wylotu i przylotu oraz daty. Przedstawione wyniki można było porządkować według rosnącego kosztu. Aplikacja nie dawała możliwości porównania cen w różnych dniach np. w formie kalendarza uzupełnionego najniższymi cenami w danym dniu, co przy podróżowaniu z niskim budżetem byłoby pomocne. Możliwe było równoczesne wyszukanie lotów oraz noclegu (ang. Bundle Deals).

Aplikacja oferowała również możliwość wypożyczenia samochodu. Tak jak w poprzednich dwóch przypadkach należało podać przedział czasu korzystania z pojazdu, miejsce odbioru i pozostawienia. Wyświetlone informacje o pojazdach były przydatne przy dokonywaniu wyboru, automatyczna czy manualna skrzynia biegów, ilość pasażerów.Dostępne sortowanie po cenie.

Ostatnia funkcjonalność, wyszukiwanie atrakcji, zabytków, ciekawych wydarzeń. Wymagane ramy czasowe oraz lokalizacja. Rezultaty mogły być przeglądane po cenie lub popularności. Nie umożliwiono użytkownikowi oceny odległości między zaproponowanymi miejscami tak jak w przypadku noclegu, ale pomocny był przybliżony czas, który należy przeznaczyć na każdą z atrakcji.

Wszystkie rezerwacje wykonane za pomocą aplikacji Expedia, można było przeglądać w zakładce Wycieczki (ang. Trips) i takim zestawem dzielić się ze współtowarzyszami podróży również korzystających z aplikacji.

Rozdział 3

Specyfikacja wymagań systemowych - ekstrakt (Anna Malizjusz)

W celu lepszego zrozumienia wymagań projektu inżynierskiego przygotowano dokument SWS (Specyfikacji Wymagań Systemowych) w całości umieszczony na załączonej do pracy płycie CD. Dokonano identyfikacji udziałowców projektu i otoczenia systemu z uwzględnieniem użytkowników oraz systemów zewnętrznych. Wyróżniono cele projektu oraz wymagania, których spełnienie będzie kluczowe dla końcowej akceptacji systemu. Poniżej przedstawiono najistotniejsze elementy SWS, które miały największy wpływ na projekt i implementacje systemu.

Wyróżniono udziałowców, których wpływ na system powinien być największy:

- programiści,
- promotor,
- użytkownik.

A także dokumenty, które należało uwzględnić w pracy:

- regulamin aplikacji,
- rozporządzenie o ochronie danych osobowych (RODO).

Docelowego użytkownika zidentyfikowano jako młodego człowieka, najczęściej studenta o ograniczonym budżecie i czasie, który może poświęcić na planowanie podróży. Posiada on smartfona z dostępem do internetu oraz używa systemu operacyjnego Android. Chce podróżować i kontaktować się z przyjaciółmi poprzez aplikację. Często zapomina o terminach i koniecznych dokumentach, więc potrzebuje przypomnień oraz dostępu do skanów przy pomocy telefonu.

Wyróżniono systemy zewnętrzne, z którymi zintegrowano aplikację. Skupiono się na dostępie do zewnętrznego API udostępnionego przez serwis Here [1], które umożliwiało dostęp do map, nawigacji i wyszukiwania obiektów takich jak hotele, restauracje, zabytki, itp. Zaletą serwisu był darmowy dostęp do danych. Here umożliwiało wykonanie 250 tys. zapytań miesięcznie bez dodatkowych opłat, a każdy kolejny tysiąc kosztował 1$, co zostało uznane za wystarczające dla testowania aplikacji. Uwzględniono również system GPS, który był niezbędny do zrealizowania podstawowych funkcjonalności, np. wyszu-

kiwania obiektów w pobliżu aktualnej lokalizacji. W tym celu zdecydowano skorzystać z możliwości oferowanych przez serwis Google Play w paczce *com.google.android.gms.location*[2].

Określono najważniejsze cele projektu:

- zwiększenie zadowolenia z podróży,
- zaspokajanie potrzeb informacyjnych użytkowników,
- zoptymalizowanie trasy,
- zmniejszenie ilości spóźnień,
- ułatwienie komunikacji pomiędzy użytkownikami,
- ułatwienie możliwości koordynacji planu dnia przez użytkownika,
- skrócenie czasu oczekiwania na dany środek transportu,
- zmniejszenie czasu przeznaczonego na planowanie podróży,
- zmniejszenie poziomu stresu użytkowników podczas planowania podróży.

Zidentyfikowano wymagania funkcjonalne, które jednocześnie stanowiły kryteria akceptacyjne projektu:

- rejestracja i logowanie użytkownika,
- dodanie, przeglądanie i edycja planu podróży i planu dnia,
- wyszukanie elementu w pobliżu danej lokalizacji,
- wyszukanie zakwaterowania,
- wyszukanie i zaproszenie innego użytkownika do wyświetlania lub edycji podróży,
- dodanie oceny do planu dnia, podróży lub odwiedzonego miejsca,
- otrzymanie propozycji na podstawie ocen,
- wyszukanie transportu między lokalizacjami,
- wyszukanie najkrótszej trasy,
- skanowanie biletów i innych dokumentów potrzebnych w trakcie podróży.

Wyróżniono dodatkowe wymagania, które nie były niezbędne do realizacji projektu, ale znacznie zwiększały możliwości aplikacji:

- oznaczenie elementu z planu dnia jako wykonany,
- udostępnianie zrealizowanego punktu planu dnia w mediach społecznościowych,
- wygenerowanie planu dnia/podróży,
- zapisywanie podróży i dokumentów na urządzeniu, aby możliwe było korzystanie z nich bez dostępu do internetu,
- powiadomienie o obiekcie w okolicy,
- powiadomienie o opóźnieniu,
- powiadomienia dotyczące lotów,
- przeglądanie statystyk zrealizowanych podróży.

Dodatkowo zostały określone wymagania jakościowe dotyczące aplikacji. Ze względu na RODO przetwarzanie danych użytkowników ograniczono do minimum i zdecydowano o wyświetlaniu użytkow-

nikom informacji o sposobie używania danych. Postanowiono skorzystać z bezpiecznych algorytmów szyfrowania i uwierzytelniania, aby zapewnić danym bezpieczeństwo. Zobowiązano się do zapewnienia autentyczności proponowanych podróży, tj. sugerowany czas spędzony w danym obiekcie jest zbliżony do rzeczywistego i trafności polecanych obiektów. Zapytania użytkowników mają być obsługiwane nie dłużej niż 5 s., a układanie planu dnia będzie trwać maksymalnie 10 s.

Za docelowe urządzenie przyjęto smartfon z systemem operacyjnym Android, którego minimalna wersja to 5.0 (Lollipop), co miało zapewnić obsługę ponad 94% urządzeń z systemem Android (dane aktualne na dzień 05.05.2019[3]). Spodziewane wymiary urządzeń to od 115.20 mm x 58.60 mm do 242,8 mm x 189,7 mm. Zaplanowano rozszerzenie działania aplikacji na telefony z systemem operacyjnym iOS.

Zwrócono uwagę na zagwarantowanie czytelności interfejsu użytkownika. Zaplanowano użycie stonowanych, niejaskrawych kolorów i ograniczenie dostępnych informacji na jednym ekranie z możliwością przejścia do kolejnych stron lub filtrowania wyników. Dostępne ma być powiększenie ekranu.

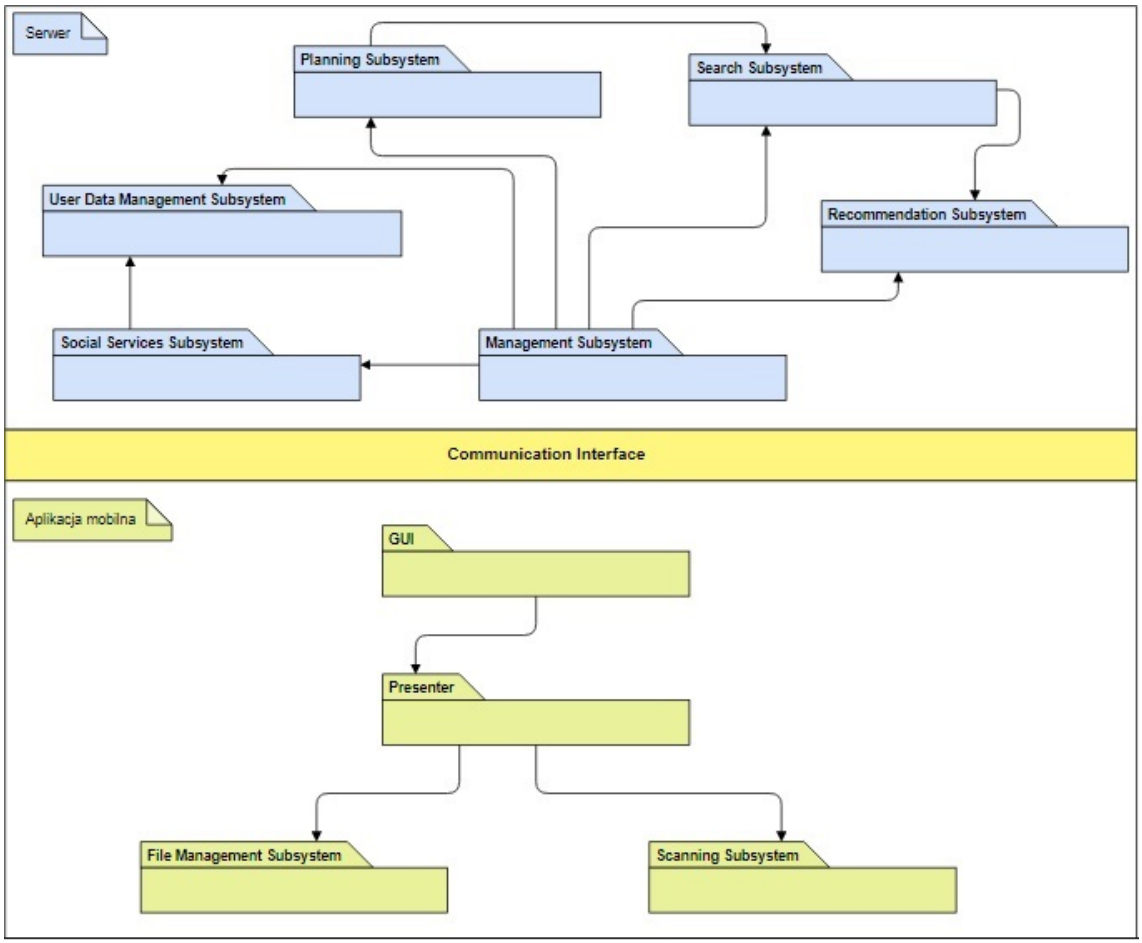
Rozdział 4

Projekt systemu (Magdalena Solecka)

System postanowiono podzielić na następujące komponenty:

- Po stronie serwera:
  - Podsystem planowania (ang. Planning subsystem)
  - Podsystem wyszukiwania (ang. Search subsystem)
  - Podsystem zarządzania danymi użytkownika (ang. User Data Management Subsystem)
  - Podsystem polecający (ang. Recommendation Subsystem)
  - Podsystem usług społecznościowych (ang. Social Services Subsystem)
  - Podsystem zarządzania (ang. Management Subsystem)
- Po stronie aplikacji mobilnej:
  - GUI
  - Prezenter (ang. Presenter)
  - Podsystem zarządzania plikami (ang. File Management Subsystem)
  - Podsystem skanowania (ang. Scanning Subsystem)

Kontakt klienta mobilnego z aplikacją serwerową zaplanowano poprzez interfejs komunikacji (ang. communication interface). Zależności pomiędzy podsystemami zostały przedstawione na diagramie (rys. 4.1).



Rysunek 4.1: Diagram komponentow systemu.

4.1 Scenariusze użycia

Zaprojektowano scenariusze użycia w celu dokładnego określenia sposobów, w które użytkownicy będą korzystać z aplikacji. Wyróżniono najważniejsze:

1. Stworzenie planu dnia.
2. Wygenerowanie planu dnia.
3. Stworzenie planu podróży.
4. Interakcje z innymi użytkownikami aplikacji.
5. Interakcje serwera z aplikacją w czasie trwania podróży.

4.1.1 Stworzenie planu dnia (Magdalena Solecka)

Martyna widzi pusty ekran planu dnia. Wpisuje miasto docelowe wyjazdu - Paryż. Wybiera przycisk "+" i wybiera atrakcje. Ukazuje się przed ną ekran wyszukiwania ze znakiem wyszukiwania,a po chwili z wynikami wyszukiwania w formie listy atrakcji. Wybiera jeden z elementów listy – katedra Notre Dame i czyta jej opis. Wybiera przycisk "Dodaj". Widzi ponownie ekran planu Dnia z dodaną przez siebie atrakcją. Wybiera w ten sposób kilka kolejnych atrakcji. Ponownie wybiera przycisk "+" i wybiera restauracje. Ponownie widzi ekran wyszukiwania z listą restauracji. Wybiera pasujący jej obiekt i przy użyciu przy-

cisku "Dodaj" zostaje on dodany do planu dnia który ponownie wyświetla się przed nią. Użytkownik wybiera przycisk "Ułóż" i czeka aż aplikacja zakończy obliczanie najbardziej optymalnej trasy. Po kilku sekundach plan dnia jest już gotowy. Martyna stwierdza jednak że potrzebuje w ciągu dnia odpoczynku dlatego postanawia przesunąć zwiedzanie katedry Notre Dame na następny dzień. Wybiera element z katedrą i przycisk "Przenieś", a następnie numer dnia podróży. Następnie stwierdza, że właściwie nie interesuje jej sztuka sakralna, więc usuwa element z planu przesuwając go w prawo.

Warunki początkowe:

1. Użytkownik jest zalogowany.
2. Aplikacja ma dostęp do internetu.
3. Użytkownik wybrał opcję stwórz plan dnia.

Warunki końcowe:

1. Użytkownik stworzył plan dnia.

4.1.2 Wygenerowanie planu dnia (Anna Malizjusz)

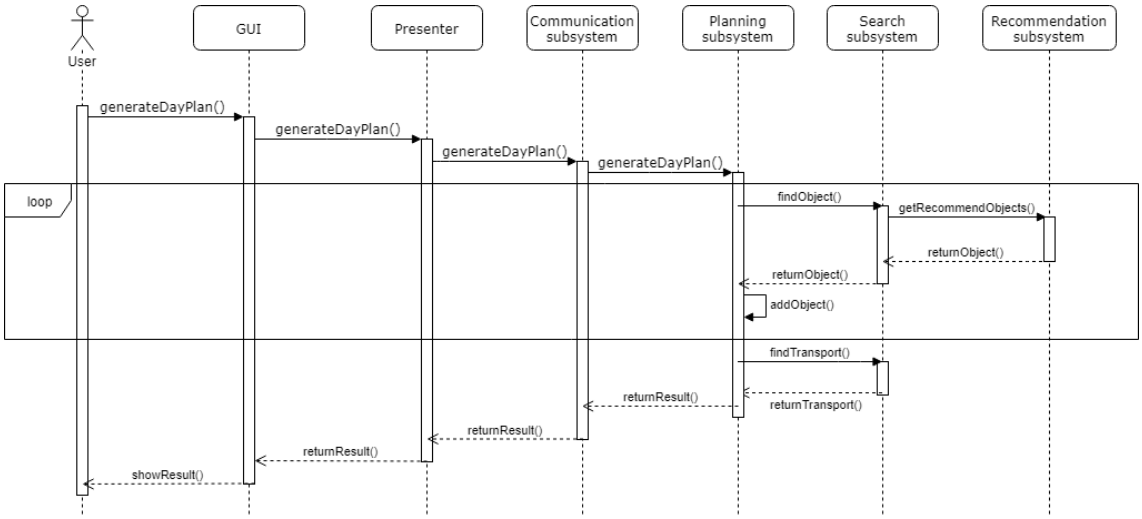
Martyna stworzyła wyjazd, lecz nie ma pomysłu na spędzenie dnia. Wybiera w aplikacji swoją podróż, po czym generuje przyciskiem plan dnia. Zostaje wyświetlony rezultat, który może dowolnie modyfikować. (rys. 4.2).

Warunki początkowe:

1. Użytkownik jest zalogowany.
2. Aplikacja ma dostęp do internetu.
3. Użytkownik wybrał podróż.

Warunki końcowe:

1. Wyświetlono proponowany plan podróży.



Rysunek 4.2: Generowanie planu dnia.

4.1.3 Stworzenie planu podróży (Karolina Makuch)

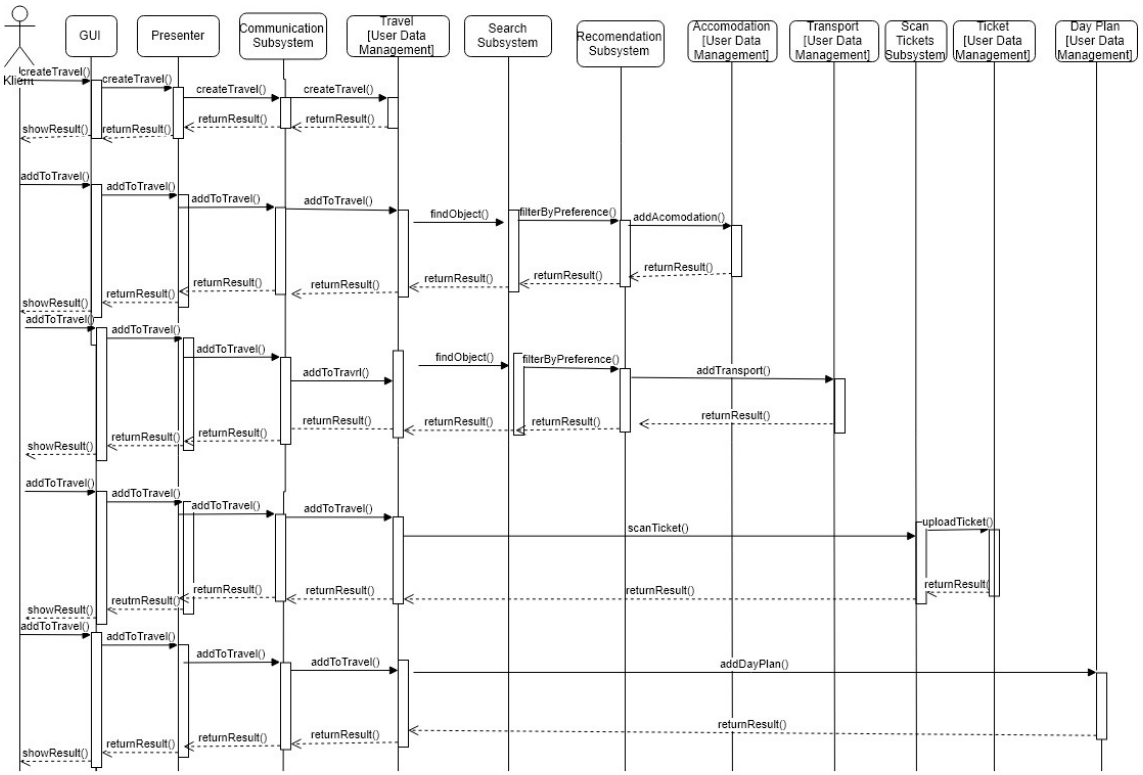
Martyna pragnie zaplanować podróż za pomocą aplikacji poprzez aplikację podróż. (rys. 4.3).

Warunki początkowe:

1. Użytkownik jest zalogowany.
2. Aplikacja ma dostęp do internetu.
3. Użytkownik wybrał opcję stwórz plan podróży.

Warunki końcowe:

1. Użytkownik stworzył plan podróży.



Rysunek 4.3: Tworzenie planu podróży.

4.1.4 Interakcje z innymi użytkownikami aplikacji (Anna Malizjusz)

Martyna wyjeżdża na wakacje razem z koleżanką, więc chce udostępnić jej swój plan podróży. Zaprasza nowego użytkownika do skorzystania z aplikacji (rys. 4.4), po czym dodaje znajomego (rys. 4.5) i zaprasza koleżankę do edytowania podróży (rys. 4.6).

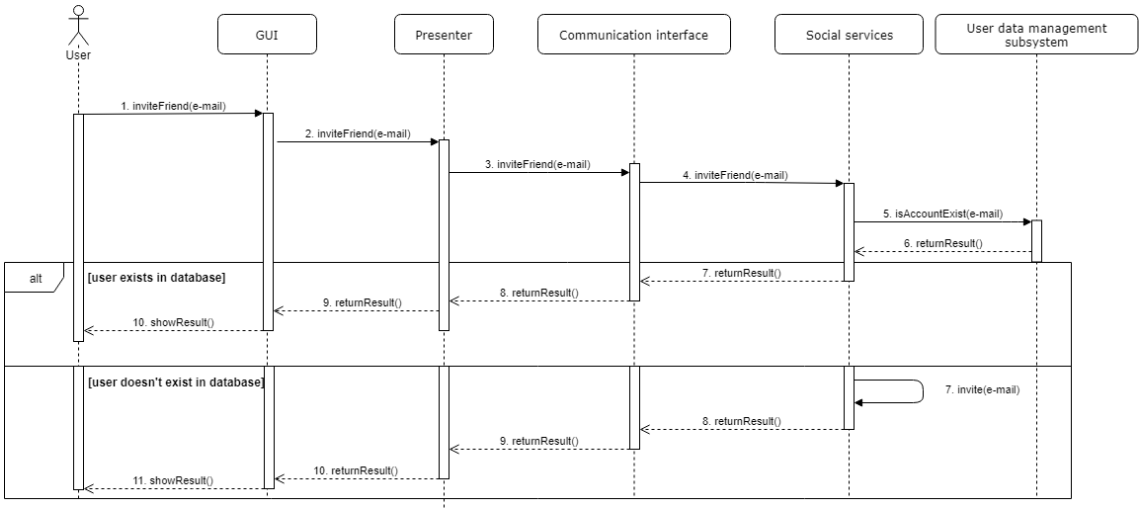
Zapraszanie użytkownika do aplikacji

Warunki początkowe:

1. Użytkownik jest zalogowany.
2. Aplikacja ma dostęp do internetu.
3. Użytkownik wybrał z menu opcję "Znajomi".

Warunki końcowe:

1. Wiadomość e-mail została wysłana lub napotkano błąd.
2. Użytkownik otrzymał informację o pomyślnym wysłaniu zaproszenia lub błędzie.



Rysunek 4.4: Zapraszanie użytkownika do skorzystania z aplikacji.

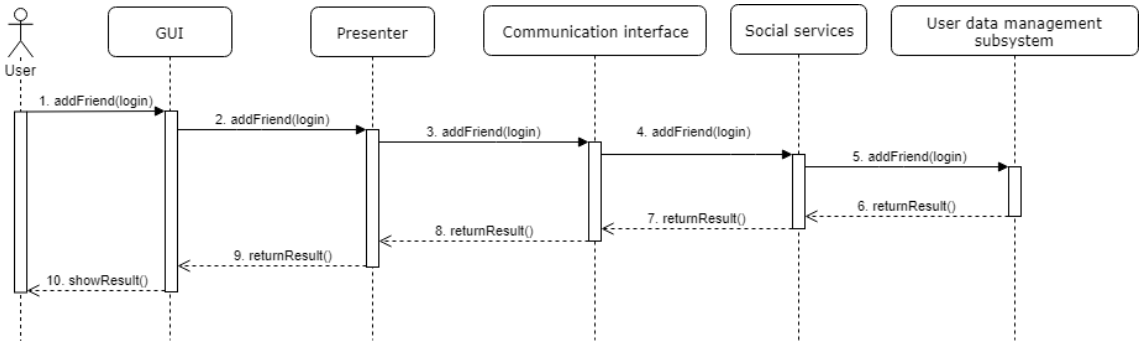
Dodawanie znajomego

Warunki początkowe:

1. Użytkownik jest zalogowany.
2. Aplikacja ma dostęp do internetu.
3. Użytkownik wybrał z menu opcję "Znajomi".

Warunki końcowe:

1. Użytkownik otrzymał informację o rezultacie.
2. Drugi użytkownik otrzymał informację zaproszenie.



Rysunek 4.5: Dodanie użytkownika do znajomych.

Udostępnienie podróży

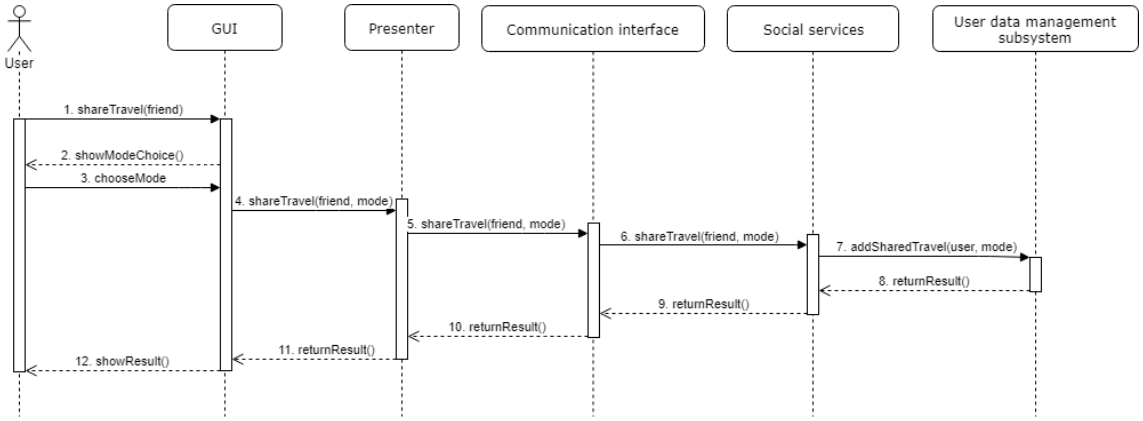
Warunki początkowe:

1. Użytkownik jest zalogowany.
2. Aplikacja ma dostęp do internetu.

3. Użytkownik wybrał podróż do udostępnienia.
4. Użytkownik docelowy jest znajomym użytkownika udostępniającego.

Warunki końcowe:

1. Użytkownik otrzymał informację o rezultacie.
2. Drugi użytkownik otrzymał informację o udostępnionej podróży.



Rysunek 4.6: Udostępnienie podróży.

4.1.5 Interakcje serwera z aplikacją w czasie trwania podróży (Dorota Tomczak)

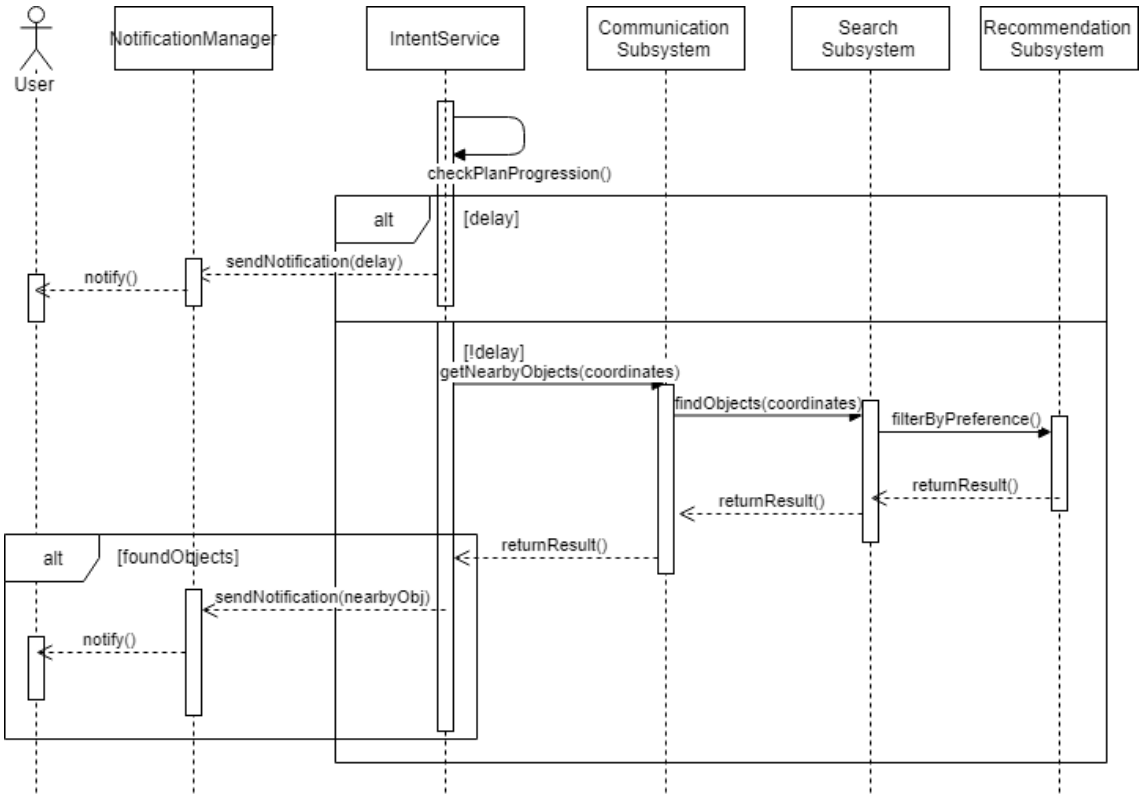
Martyna wraz z koleżanką odbywają zaplanowaną poprzez aplikację podróż. W drodze do jednej z atrakcji aplikacja powiadamia Martynę o interesującym obiekcie znajdującym się niedaleko miejsca, w którym przebywają. Dziewczyny ruszają do wskazanej lokalizacji. Tracą poczucie czasu, więc aplikacja informuje je, że jeśli się nie pośpieszą, nie zdążą dojść do następnego punktu planu dnia przed jego zamknięciem (rys. 4.7).

Warunki początkowe:

1. Użytkownik jest zalogowany.
2. Aplikacja ma dostęp do internetu.
3. Aplikacja ma dostęp do lokalizacji użytkownika.
4. Użytkownik nie zablokował powiadomień od aplikacji.
5. Użytkownik utworzył plan podróży i jest w określonym miejscu i czasie.

Warunki końcowe:

1. Użytkownik otrzymał odpowiednie powiadomienie.



Rysunek 4.7: Otrzymywanie powiadomień podczas trwania podróży.

Rozdział 5

Decyzje projektowe

5.1 Wybór języka Kotlin

Przed rozpoczęciem prac implementacyjnych nad projektem należało podjąć decyzję, co do wyboru języka programowania. W przypadku aplikacji mobilnej na system Android, przy założeniu że miałaby to być aplikacja natywna, wybór ten ograniczał się do dwóch opcji – Java lub Kotlin. Ostatecznie zdecydowano o napisaniu aplikacji w Kotlinie, głównie ze względu na fakt, że 7 maja 2019 firma Google na konferencji Google I/O 2019 ogłosiła, że ten język jest obecnie preferowanym językiem do tworzenia aplikacji mobilnych na Androida[32]. Aby ułatwić jednoczesną pracę nad aplikacją serwerową podjęto decyzję o wykorzystaniu języka Kotlin również i w tym projekcie.

Podczas pracy nad projektem oraz z perspektywy czasu po zakończeniu działań implementacyjnych, jednomyślnie stwierdzono, że wybór Kotlina był słuszny. Język ten zapewnia wiele przydatnych udogodnień i ulepszeń w stosunku do języka Java, które wpływają na jakość kodu i satysfakcję z pracy nad nim. Jedną z jego niezaprzeczalnych zalet jest minimalizacja tzw. kodu boilerplate, co przejawia się między innymi w takich jego cechach jak:

- utworzenie klasy typu Singleton jest jednoznaczne z dodaniem klasy oznaczonej słowem kluczowym *object*,
- klasy oznaczone słowem kluczowym *data* służące głównie jako obiekty DTO, do których to kompilator sam generuje konstruktory oraz metody dostępu do wszystkich pól,
- wyrażenia lambda i funkcje inline.

```
data class Travel (
        val id: Int,
        var name: String,
        var imageUrl: String? = null
) : Serializable
```

Rysunek 5.1: Klasa DTO w Kotlinie - 5 linii kodu.

Rozdział 6

Aplikacja mobilna

6.1 Projekt i implementacja oparte na wzorcu MVP (Dorota Tomczak)

Pracę nad aplikacją mobilną rozpoczęto od stworzenia bazowego projektu składającego się na kilka pustych widoków, do których to następnie można było dodawać kolejne funkcjonalności. W celu zachowania najlepszych praktyk programistycznych zdecydowano się na oparciu projektu o wzorzec architektoniczny MVP (ang. model-view-presenter). Jest to wzorzec szczególnie nadający się do implementacji w aplikacjach mobilnych na systemy Android ze względu na aktywności (ang. activity), które pełnią funkcję środkowej warstwy – widoku (ang. view). W implementacji wzorca w tym projekcie widok jest pasywny (ang. passive view) tzn. widok powinien odpowiadać jedynie za wyświetlanie interfejsu i użycie bibliotek związanych z Androidem. Cała logika aplikacji ma być zawarta w prezenterze (ang. presenter), który pełni funkcję kontrolera.

Zastosowanie wzorca MVP pozwoliło na zachowanie porządku w strukturze projektu i przejrzysty podział na warstwy. Do każdej aktywności, czyli nowego ekranu w aplikacji, utworzono interfejs zwany kontraktem (ang. contract) zawierający opis interakcji jakie mogą zajść pomiędzy prezenterem a widokiem, a w szczególnych przypadkach również między prezenterem a adapterem, który odpowiada za wyświetlanie listy obiektów. Klasy aktywności i prezentera implementują interfejsy zawarte w kontrakcie, prezenter jest wstrzykiwany do widoku a referencja widoku jest przekazywana do prezentera w konstruktorze.

Do wstrzykiwania zależności, w tym prezenterów do aktywności, wykorzystano popularny framework *Dagger 2*[12], co ostatecznie okazało się być nie najlepszym wyborem – należało dodać dwie dodatkowe klasy do każdej aktywności, co przy dużej ich liczbie wygenerowało wiele plików o bardzo podobnej strukturze. Ponadto niemalże jedynymi wstrzykiwanymi obiektami były obiekty klas prezenterów.

```
interface LauncherContract {
    interface View {
        fun showSignIn()
        fun showTravels()
    }
    interface Presenter {
        fun redirect(credentials: SharedPreferencesUtils.Credentials)
        fun unsubscribe()
    }
}
```

Rysunek 6.1: Prosty kontrakt widoku odpowiadający za przekierowanie do widoku logowania lub listy podróży.

6.2 Komunikacja z aplikacją serwerową (Anna Malizjusz)

Aplikacja mobilna musi komunikować się z RESTowym API udostępnianym przez serwer. W tym celu wykorzystano klienta o nazwie Retrofit 2 [10]. Dzięki niemu w łatwy sposób można zaimplementować interfejs odpowiedzialny za wysyłanie zapytań i odbieranie odpowiedzi.

Napisano kilka klas, aby umożliwić prostą komunikację w aplikacji. Interfejs *ServerApi* zawierał zbiór metod z odpowiednimi adnotacjami @PUT, @POST, @GET, @DELETE i nazwami punktów końcowych. Metody nie wymagały implementacji przez programistów, ponieważ należało to do odpowiedzialności klienta Retrofit 2.

```
interface ServerApi {

    // users - travels
    @GET( value: "users/{userId}/travels")
    fun getTravels(@Path( value: "userId") userId: Int): Single<Response<List<Travel>>>

    @POST( value: "users/{userId}/travels")
    fun addTravel(@Path( value: "userId") userId: Int, @Body travelName: String): Single<Response<Travel>>

    @PUT( value: "users/{userId}/travels")
    fun changeTravelName(@Path( value: "userId") userId: Int, @Body travel: Travel): Single<Response<Travel>>
```

Rysunek 6.2: Przykładowa zawartość interfejsu używanego przez Retrofit 2.

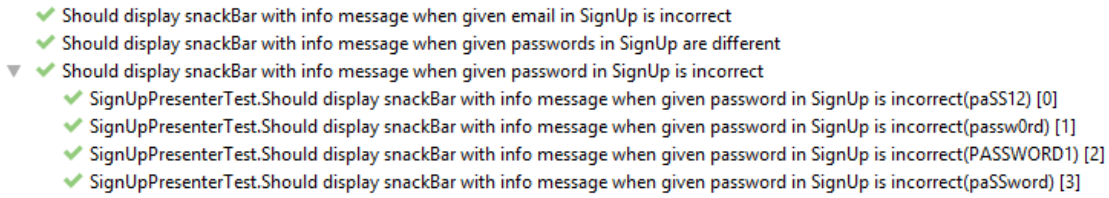
Dodatkowo zaimplementowano klasę pomocniczą konfigurującą klienta. Udostępniała ona zmienną (ang. property), która była przygotowanym interfejsem do komunikacji. Ustawiono w niej adres serwera REST, a także konwertery przeprowadzające serializację obiektów do formatu JSON oraz deserializację z otrzymanego ciągu znaków w formacie JSON do obiektu będącego instancją danej klasy. Skorzystano z klasy *GsonConverter*, która jest oferowana przez Retrofit API. Wykorzystano wzorzec projektowy interceptor implementując klasę *AuthTokenInterceptor*, którego rolą było dodanie do każdego zapytania nagłówka (ang. header) z lokalnie zapisanym tokenem identyfikującym użytkownika. Interceptor aplikacji jest wywoływany zawsze i tylko jeden raz, nie wpływają na to przekierowania ani ponawianie zapytań. Aby zintegrować interfejs używanego klienta z interceptorem, należało dodać dodatkowego klienta - *OkHttpClient*[11]. Pochodzi z biblioteki OkHTTP, którą można było wykorzystać do komuni-

kacji aplikacji mobilnej z serwerem, jednak postawiono wybrać Retrofit 2 z uwagi na mniejszy poziom skomplikowania oferowanego API.

Rezultatem każdego zapytania jest struktura *Single<Response<T> >*, gdzie T jest oczekiwanym typem zwracanego obiektu. Obiekt *Single* informuje o tym, że jest spodziewana pojedyncza odpowiedź. *Response* jest strukturą zdefiniowaną w projekcie inżynierskim. Zawiera kod odpowiedzi opisywany szerzej przy sposobie implementacji błędów, a także także pole *data* z przesłanymi przez serwer danymi.

6.3 Logika rejestracji i logowania (Anna Malizjusz)

Rejestracja użytkownika polega na podaniu trzech informacji: adresu email oraz dwukrotnym podaniu hasła. Każde z tych pól jest walidowane. Adres email powinien mieć formę xxx@yyy.zzz, gdzie xxx, yyy i zzz są dowolnymi ciągami znaków. Hasło nie powinno być trywialne. Musi zawierać co najmniej jedną cyfrę, jedną wielką i jedną małą literę, a także mieć przynajmniej 8 znaków długości. Poprawność walidacji została sprawdzona testami jednostkowymi.



Rysunek 6.3: Testy jednostkowe sprawdzające poprawność sposobu walidacji formularza rejestracji

6.4 Wyszukiwanie atrakcji turystycznych i zakwaterowania (Anna Malizjusz)

Dodanie konkretnego miejsca do planu dnia jest możliwe po jego wyszukaniu. W tym celu stworzono ekran umożliwiający wybranie obiektu z mapy. Na wyświetlanym obszarze są widoczne pinezki odpowiadające obiektom w podanej okolicy.

Głównym komponentem widoku wyszukiwania jest *SearchView*[4]. Dzięki temu na ekranie widać pole, w które można wpisać nazwę lub początek nazwy miasta i wybrać je z listy. Mapa automatycznie przeniesie się do wskazanego miejsca. Implementacja mechanizmu podpowiedzi znajduje się w klasie *CitySuggestionProvider*, dziedziczącej po *ContentProvider*[5]. Kluczowym było nadpisanie metody *query(Uri, String[], Bundle, CancellationSignal)*, która zwracała wynik do widoku. Używając interfejsu komunikacji z serwerem aplikacja pobierała listę proponowanych miast, a *CitySuggestionProvider* zwracał je w formie kursorów.

Po wybraniu pinezki jednego z wyświetlanych obiektów na ekranie pojawia się nazwa miejsca, a także możliwość pokazania jego szczegółów. Zostało to osiągnięte dzięki użyciu układowi o nazwie *SlidingUpPanelLayout*[6]. Pozwala on na rozwinięcie i późniejsze ukrycie widocznego na dole obszaru ze

szczegółowymi informacjami. Te same dane są w późniejszym etapie dostępne po wybraniu elementu z planu dnia. Z tego powodu postanowiono wyodrębnić układ pól z informacjami, zdefiniować go w osobnym pliku i wykorzystać go zarówno w widoku wyszukiwania, jak i widoku detali obiektu.

Uwzględniono szczególny przypadek, jakim jest wyszukanie zakwaterowania. Jeżeli aplikacja wykryje, że dany obiekt jest miejscem noclegu, zaproponuje dodatkowo dodanie daty wykwaterowania. Końcowym efektem będzie pojawienie się dwóch elementów w planach dni. Pierwszy z nich jest podpisany jako zakwaterowanie (ang. check in), a drugi jako wykwaterowanie (ang. check out). Wystawienie oceny jest możliwe tylko dla elementu reprezentującego drugie z tych wydarzeń.

6.5 Tworzenie planów podróży (Dorota Tomczak)

Tworzenie planów podróży jest kluczową funkcją aplikacji. Stworzenie takiego planu jest możliwe, gdy użytkownik ma przypisaną do swojego konta dowolną podróż. Dodawanie do niej kolejnego elementu planu zostało zrealizowane poprzez prosty formularz, który zawiera:

- rozwijaną listę z kategoriami miejsc odpowiadającą wybranym kategoriom z *Here API*[1],
- pole z nazwą miejsca, które po dotknięciu przekierowuje do ekranu wyszukiwania, a po wybraniu obiektu wypełnia się automatycznie,
- pola od dnia i od godziny, po których dotknięciu otwierają się dialogi odpowiednio z kalendarzem i zegarem,
- pola analogiczne do dwóch powyższych, jeśli wybrano kategorię Zakwaterowanie, oznaczające dzień i godzinę wymeldowania,
- pole z adresem, które wypełnia się automatycznie po wyszukaniu obiektu,
- pole na notatki, które są potem możliwe do edycji.

Po zatwierdzeniu formularza zostaje on poddany walidacji – wszystkie pola oprócz notatek muszą być wypełnione, a dla kategorii *Zakwaterowanie* (ang. *Accommodation*) czas zakwaterowania nie może być przed czasem wykwaterowania. Jeśli formularz pomyślnie przejdzie próbę walidacji, nowy element planu podróży zostaje utworzony. Elementy planu dnia wyświetlają się w formie chronologicznej listy rozdzielonej separatorami z datami. Do implementacji tego rozwiązania posłużył adapter, który na podstawie typu elementu określa jaki układ xml (ang. layout xml) wyświetlić – czy ten dla elementu planu czy daty. Aby oba typy mogły być używane przez adapter, muszą dziedziczyć po wspólnym interfejsie, którym jest *DayPlanItem* z jedną metodą, która zwraca typ obiektu.

Jednak zanim nowy element planu dnia może zostać przekazany do adaptera i wyświetlony na ekranie jest najpierw dodawany do kolekcji typu *TreeSet*, która sortuje znajdujące się w niej elementy po wyniku zwracanym przez metodę *compareTo(other: PlanElement): Int*, zaimplementowaną w klasie dodawanych elementów. Następnie tworzona jest lista składająca się z separatorów oraz wspomnianych elementów – po sprawdzeniu pola *fromDate* elementu, następuje decyzja, czy rozdzielić elementy nowym separatorem z dniem czy nie. Tak przygotowana lista trafia do adaptera i efektem jest widok posortowanych chronologicznie elementów planów dni w podróży.

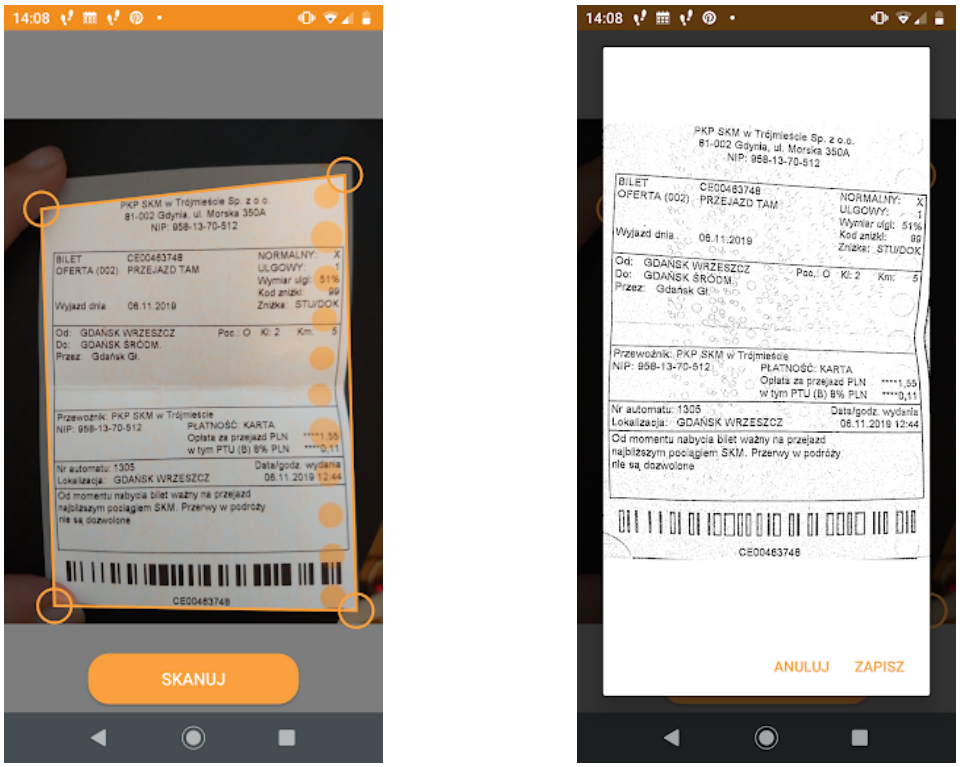
6.6 Implementacja skanowania dokumentów z użyciem OpenCV (Dorota Tomczak)

Aplikacja umożliwia zrobienie zdjęcia, wybranie czworokątnego obszaru oraz takie przycięcie i modyfikację kolorów fotografii aby uzyskać efekt skanu. W początkowych założeniach zakładano skorzystanie z gotowej biblioteki, która poza wymienionymi wyżej funkcjonalnościami dokonywałaby automatycznej detekcji krawędzi, jednak ze względu na brak darmowych i dostosowanych do potrzeb rozwiązań zdecydowano stworzyć skaner od podstaw bazując na projekcie open-source *SimpleDocumentScanner-Android*[13].

Wykonanie fotografii jest możliwe jeśli użytkownik nadał aplikacji odpowiednie uprawnienia. Są one weryfikowane przy każdej próbie dodania skanu. Jeśli aplikacja została do tego uprawniona, tworzony jest plik tymczasowy w pamięci urządzenia, a następnie otwierany jest widok kamery za pomocą intencji (ang. intent) utworzonej poprzez podanie do konstruktora nazwy odpowiedniej akcji. Zdjęcie wykonane przez użytkownika jest zapisywane w utworzonym wcześniej pliku tymczasowym a jego ścieżka przekazywana do aktywności (ang. activity) skanera, aby umożliwić w niej jego wyświetlenie i modyfikację.

W celu zaimplementowania możliwości wyboru obszaru zdjęcia poprzez gesty użytkownika dodano specjalną klasę dziedziczącą po *ImageView*. Taka klasa mogła zostać następnie dodana do layoutu xml skanera, a zaimplementowane w niej metody pozwoliły na rysowanie i przeciąganie czterech linii złączonych końcami i tworzących czworokąt. Końce czworokątu są reprezentowane przez cztery zmienne typu *PointF* zawierające informacje o ich położeniu na ekranie urządzenia. Po każdym wykryciu gestu przeciągnięcia linii łączącej punkty, obliczane jest ich nowe położenie, a następnie ponownie złączane są linią. Gdy użytkownik zatwierdzi zaznaczony obszar punkty transformowane są z ich współrzędnych na urządzeniu na współrzędne na zdjęciu.

Kolejnym etapem jest uzyskanie efektu zeskanowanego dokumentu. Aby móc to osiągnąć zastosowano bibliotekę *OpenCV*[14], której pliki źródłowe musiały zostać dołączone do projektu aplikacji w postaci modułu. Aby ograniczyć rozmiar aplikacji wynikowej, który znacznie wzrósł po dodaniu biblioteki, usunięto jej klasy, które nie były potrzebne do realizacji funkcjonalności. Najważniejszymi operacjami, które dostarczyła OpenCV jest transformacja perspektywiczna (ang. perspective transform), która w ogólności zajmuje się przekształceniem trójwymiarowego świata do dwuwymiarowego obrazka, oraz progowanie obrazu (ang. thresholding), czyli metoda segmentacji obrazu, pozwalająca uzyskać obraz binarny z obrazu kolorowego. Otrzymany skan po zastosowaniu tych dwóch operacji i innych pomocniczych może być przez użytkownika zachowany lub odrzucony. W tym pierwszym przypadku zostaje on przesłany na serwer.



(a) Zaznaczanie obszaru do skanowania. *(b) Skan wynikowy po transformacjach zdjęcia.*

Rysunek 6.4: Skanowane biletu.

6.7 Obsługa przesyłania i pobierania plików (Dorota Tomczak)

Aby umożliwić użytkownikom korzystanie ze swoich kont na różnych urządzeniach oraz udostępnianie stworzonych przez siebie podróży, wynikła konieczność przesyłania dodanych zdjęć na serwer, zamiast przechowywania ich lokalnie na urządzeniach mobilnych. Zdjęciami w aplikacji mogą być wykonane skany lub fotografie dodane do poszczególnych podróży, a ich przesyłanie odbywa się przy pomocy RESTowego API udostępnianego przez serwer, jak zostało to opisane w podrozdziale 5.2. Zapytania za to odpowiadające są dodatkowo opatrzone adnotacją *@Multipart*, aby umożliwić przesyłanie całych plików w ciele (ang. body) żądania, natomiast pliki przekształcane są na typ *MultiPartBody.Part*. Gdy plik zostanie odebrany po stronie serwera następuje próba jego zapisu w ustawionym w konfiguracji folderze przez *FileStorageService*, czyli serwis do przechowywania plików. Następnie ścieżka do pliku jest dodawana do bazy wraz z innymi niezbędnymi informacjami umożliwiającymi jego identyfikację.

Ładowanie plików w aplikacji zostało zrealizowane dzięki bibliotece *Glide*[17], która pozwala na zwiększenie efektywności tego procesu poprzez automatyczne cachowanie i optymalizację rozmiaru zdjęcia. Znając adres, pod którym znajduje się plik, użycie wspomnianej biblioteki jest bardzo proste, dlatego na większą uwagę zasługuje proces wysyłania fotografii przez serwer. Plik o wskazanej nazwie zostaje załadowany jako typ *Resource* przez *FileStorageService*, a następnie konstruowana jest odpowiedź serwera, która w ciele zawiera otrzymany zasób. Odpowiedź jest dodatkowo opatrzona nagłówkiem *Content-Disposition*, który zawiera informację o tym, że plik ma zostać potraktowany jako załącznik.

6.8 Dodawanie użytkowników do znajomych oraz wyświetlanie listy znajomych (Karolina Makuch)

W celu umożliwienia użytkownikowi udostępniania planu podróży znajomym dodano do aplikacji funkcjonalność odpowiadającą za wyszukiwanie konkretnych użytkowników. Została ona zrealizowana poprzez wykorzystanie komponentu *SearchView* (*appcompat.widget.SearchView*) zawierającego pole umożliwiające wpisanie ciągu znaków. Podczas wprowadzania kolejnych liter zostają wyświetlane podpowiedzi w czasie rzeczywistym. Aktualizują się przy każdej zmianie wyrazu. Uwzględniają one wszystkie adresy mailowe zarejestrowanych w aplikacji użytkowników zawierające wpisane litery.

W tym celu została stworzona klasa implementująca abstrakcyjną klasę *ContentProvider* (*android.content.ContentProvider*). Po wysłaniu odpowiedniego zapytania zostaje utworzony *MatrixCursor* zawierający dwie kolumny: identyfikator użytkownika oraz jego adres mailowy. Po otrzymaniu odpowiedzi od serwera dla każdego użytkownika zostaje utworzony wiersz, który następnie jest dodawany do kursora. Wyświetlana jest tylko kolumna zawierająca adres mailowy.

Kursor został również wykorzystany w celu umożliwienia dodania wybranego użytkownika do znajomych. Podczas kliknięcia na wybrany wiersz zostaje wywoływana nadpisana metoda *onSuggestionClick* obiektu klasy *SearchManager* (*android.app.SearchManager*). Pobiera ona potrzebne dane wybranej pozycji z tablicy *MatrixCursor'a*.

Do eliminacji przypadkowych działań zostało wykorzystane okno dialogowe wymagające potwierdzenia zamiaru dodania wybranego użytkownika do znajomych. W tym celu została wykorzystana klasa *AlertDialog* (*androidx.appcompat.app.AlertDialog*). Po wyrażeniu zgody oraz w przypadku poprawnego wykonania metody użytkownik otrzymuje powiadomienie stworzone poprzez wywołanie konstruktora klasy *Snackbar* (*com.google.android.material.snackbar.Snackbar*).

W tym samym oknie pod polem do wyszukiwania znajduje się lista znajomych użytkownika. Została ona dodana jako liniowa warstwa (ang. *LinearLayout*) umożliwiająca odświeżanie widoku poprzez szybkie przesunięcie palcem po ekranie (ang. *SwipeRefreshLayout*). Implementuje ona tryb usuwania.

Powiązanie dwóch użytkowników poprzez oznaczenie ich jako znajomych zostało zrealizowane poprzez dołączenie do bazy danych osobnej tabeli przechowującej potrzebne informacje. Dodawanie jest jednostronne. Znajomy danego użytkownika nie jest zobowiązany do posiadania go na swojej liście znajomych.

Po stronie serwera zostały dodane pliki umożliwiające komunikację z wspomnianą powyżej tabelą oraz wykonywaniu potrzebnych zapytań. Oprócz bazowych, niezbędnych do komunikacji zostało również zaimplementowane w jednym zapytanie zwracające wszystkie adresy mailowe nieznajomych użytkowników zawierających wpisany ciąg znaków.

6.9 Udostępnianie podróży znajomym (Karolina Makuch)

Udostępnianie podróży jest możliwe po dodaniu użytkownika do listy znajomych. W momencie wybrania ikony udostępnienia pojawia się okno dialogowe (*androidx.fragment.app.DialogFragment*)

zawierające znajomych użytkownika w formie listy wyboru (ang. *Checkbox*) nie mających dostępu do danego planu podróży. Podczas implementacji została nadpisana metoda *setMultiChoiceItems* w celu umożliwienia udostępnienia planu wielu znajomym podczas jednej czynności.

Po wybraniu osób, zatwierdzeniu decyzji oraz poprawnym wykonaniu metody użytkownik otrzymuje powiadomienie stworzone poprzez wywołanie konstruktora klasy *Snackbar* (*com.google.android.material.snackbar.Snackbar*) Odpowiednie powiadomienie zostaje również wyświetlone w sytuacji niewybrania ani jednego znajomego oraz potwierdzeniu wykonania czynności.

Każdy użytkownik mający dostęp do podróży posiada prawa do edycji jej planu. Podczas wykonywania czynności udostępniania wybranemu znajomemu zostaje przypisana dana podróż poprzez dodanie powiązania w odpowiedniej tabeli.

```
"SELECT DISTINCT $tableName.* FROM $tableName " +
        "INNER JOIN ${UserFriendRepository.tableName} " +
        "on $tableName.$columnId = ${UserFriendRepository.tableName}.${UserFriendRepository.columnFriendId} " +
        "WHERE ${UserFriendRepository.tableName}.${UserFriendRepository.columnUserId}=? " +
        "AND $tableName.$columnId " + negation + " IN " +
        "(SELECT ${UserTravelRepository.tableName}.${UserTravelRepository.columnUserId} " +
        "FROM ${UserTravelRepository.tableName} " +
        "WHERE ${UserTravelRepository.tableName}.${UserTravelRepository.columnTravelId}=?) "
```

Rysunek 6.5: Zapytanie zwracające listę adresów mailowych znajomych danego użytkownika nieposiadających dostępu do wybranej podróży

6.10 Udostępnianie planu dnia w medium społecznościowym (Karolina Makuch)

Po dłuższym przytrzymaniu poszczególnego elementu planu pojawia się „dymek" (*android.widget.PopupMenu*) zawierający trzy czynności do wyboru. Jedną z nich jest możliwość udostępnienia wybranego punktu w medium społecznościowym. Postanowiono wybrać jedno z najpopularniejszych – *Facebook'a*. W tym celu wykorzystano akcję wysyłającą (ang. *ACTION.SEND*) za pomocą intencji (*android.content.Intent*).

W przypadku znalezienia aplikacji Facebook na telefonie użytkownika, zostanie on automatycznie przekierowany do publikowania post'a na własnej „tablicy". Domyślną treścią jest link do map HERE z wyszukanym miejscem z elementu planu dnia. W celu pobrania właściwego odnośnika zostało zaimplementowane odczytywanie wartości parametru widoku (ang. *view*) znajdującego się w odpowiedzi w formacie JSON (z wykorzystaniem *com.google.gson.JsonParser*). Odsyłacz do wyników zapytania został pobrany z bazy danych. Z racji niemożliwości wykonania analizy (ang. *parse*) wyniku zapytania w głównym wątku aplikacji utworzono osobny wątek.

Początkowo planowano inną domyślną treść. Miała ona zawierać najważniejsze informacje na temat wybranego elementu planu. Niestety zarówno wbudowany w Androidzie moduł udostępniania, a także *API Facebook'a* nie udostępnia tej możliwości. Problem ten został zgłoszony ponad 7 lat temu, lecz nie został rozwiązany[15]. Pola *"EXTRA SUBJECT"* oraz *"EXTRA TEXT"* są pomijane w momencie próby udostępniania domyślnej zawartości w aplikacji Facebook. Z racji możliwości załączania zdjęć, linków

oraz plików multimedialnych (np. video) rozważane było zamienienie tekstu na plik obrazu oraz dodanie go do akcji. Brane również pod uwagę było przeniesienie domyślnej wiadomości do schowka oraz wyświetlenie komunikatu informującego użytkownika z prośbą o wklejenie wiadomości.

Jednakże stwierdzono, że takich akcji nie powinno się wymagać od użytkownika. Ostatecznie postanowiono, że najlepszym rozwiązaniem będzie zdecydowanie się na wspomniany wcześniej odsyłacz do wyszukanego miejsca w mapy HERE[16]. Dzięki czemu treść post'a zostanie utworzona przez użytkownika umożliwiając mu zachowanie własnego charakteru pisania.

6.11 Manualna realizacja planu (Karolina Makuch)

Kolejną akcją dostępną w „dymku" pojawiającym się po przytrzymaniu poszczególnego elementu jest manualna realizacja planu dnia. Początkowa treść akcji dla nowopowstałego nieukończonego jeszcze elementu planu dnia brzmi *„Oznacz jako ukończony"* (ang. *„Mark as completed"*). W przypadku oznaczenia części składowej planu jako zrealizowany tekst automatycznie zmienia się na *„Oznacz jako nieukończony"* (ang. *„Mark as incompleted"*) oraz zwiększa się stopień przeźroczystości odpowiedniego elementu listy. Postanowiono oznaczać zrealizowane plany poprzez wygaszenie kolorów elementów danego planu zmniejszając wartość kanału *alfa* (ang. *Alpha*).

Zdecydowano się umożliwić użytkownikowi cofnięcie wprowadzonych zmian w przypadku wykonania niechcianego kliknięcia, świadomie rezygnując z wymagającego potwierdzenia okna dialogowego. Stwierdzono duże prawdopodobieństwo wykonywania tej akcji w pośpiechu. Dodatkowe kliknięcie wydłużyłoby czas wykonywania akcji oraz mogłoby to irytować użytkownika.

Oznaczenie planu jako wykonany zostało umożliwione dzięki dodaniu odpowiedniej kolumny do tabeli zawierającej plany dni. Dzięki temu każdy użytkownik mający dostęp do podróży ma możliwość wykonania tej czynności i jej efekty są widoczne dla wszystkich uczestników wyprawy.

Rozdział 7

Serwer REST

7.1 Implementacja serwera (Anna Malizjusz)

Serwer dla aplikacji mobilnej został zaimplementowany w języku Kotlin. Celem było stworzenie bezstanowego API zgodnego z tzw. RESTful Web Service. Oznacza to, że na serwerze nie jest utrzymywana sesja użykownika, a każde zapytanie jest niezależne od poprzednich. Każdorazowo należy podać wszystkie niezbędne informacje niezbędne do realizacji żądania, m. in. token, który potwierdza tożsamość użytkownika i uprawnia go do określonych akcji.

Wykorzystano framework Spring Boot[7]. Jest on oparty na platformie Spring, która dostarcza mechanizmów wstrzykiwania zależności, możliwości użycia wzorca MVC (ang. model-view-controller), a także modułów do implementacji testów jednostkowych. Celem obu frameworków jest ułatwienie implementacji serwera. Jedną z zalet użytego niniejszej pracy frameworka jest prosta konfiguracja, która nie wymaga tworzenia plików w formacie xml. Wynika to z zastosowania reguły *konwencja ponad konfigurację* (ang. convention over configuration). Programista nie musi definiować wszystkich ustawień, jeśli stosuje się do przyjętych konwencji. W przypadku technologii Spring Boot kluczowe są adnotacje nad klasami pełniącymi określone role. Konfiguracja serwera zachodzi automatycznie na podstawie zależności, jeśli dodano adnotację "@EnableAutoConfiguration" do klasy uruchamiającej serwer. Framework dostarcza również narzędzi do tworzenia punktów końcowych (ang. endpoints).

Skorzystano z mechanizmu wstrzykiwania zależności. Klasy mogą być oznaczone jako komponenty (@Component), serwisy (@Service). Serwis jest szczególnym typem komponentu. Komponenty są zarządzalnymi obiektami w aplikacji oraz mogą być wstrzykiwane do pól odpowiedniego typu, które są oznaczone adnotacją @Autowired. Serwisy to elementy, które należą do logiki biznesowej.

Definicję punktów końcowych umieszczono w klasach kontrolerów oznaczonych adnotacją "@RestController". Zdefiniowano kilka rodzajów kontrolerów, aby rozdzielić odpowiedzialność za poszczególne zadania:

- kontroler użytkowników jako *ServerUserController*
- kontroler podróży jako *ServerTravelController*
- kontroler odpowiedzialny za skany jako *ServerScanController*

- kontroler pośredniczący w komunikacji z zewnętrznym API dostarczanym przez firmy Google oraz Here: *ServerHereGoogleApiController*
- kontroler funkcji rekomendujących: *ServerRecommendationController*

W każdym z kontrolerów wyróżniono punkty końcowe (ang. endpoints). Zastosowano powszechną konwencję nazewnictwa oraz znaczenie czasowników protokołu HTTP. Przykładowo dodawanie (POST), usuwanie (DELETE), aktualizacja (PUT) oraz odczytanie (GET) podróży obsługiwane w kontrolerze *ServerTravelController* odbywa się w następujący sposób:

- *@GetMapping("users/userId/travels")* zwraca podróże należące do użytkownika o podanym numerze ID;
- *@PostMapping("users/userId/travels")* dodaje podróż podaną w ciele zapytania (ang. body) do podróży użytkownika o podanym numerze ID;
- *@PutMapping("users/userId/travels")* aktualizuje podróże zawarte w ciele zapytania (ang. body);
- *@DeleteMapping("users/userId/travels")* umożliwia usunięcie listy podróży, która została wysłana w ciele zapytania.

Każde zapytanie powinno zawierać w nagłówku token użytkownika, który jest sprawdzany w celu weryfikacji źródła zapytania. Parametry ścieżki (ang. path parameters) punktów końcowych uszczegóławiają zasób, np. "userId" w powyższym przykładzie. Dodatkowe parametry zapytania (ang. query parameters) i ciało zapytania(ang. body) pozwala na sprecyzowanie żądania.

7.2 Obsługa sytuacji wyjątkowych (Dorota Tomczak)

Aplikacja mobilna komunikuje się z serwerem poprzez REST'owe API, jeśli więc po stronie serwera dojdzie do sytuacji wyjątkowej, aplikacja powinna otrzymać wiadomość o tym, co poszło nie tak i odpowiednio ją obsłużyć. W tym celu zdefiniowano kilkanaście własnych wyjątków, czyli klas dziedziczących po klasie java.lang.Exception oraz implementujących własny interfejs ApiException, który zawiera kod błędu wraz z wiadomością opisującą błąd. W celu uniknięcia tworzenia wielu bloków try-catch oraz powielania bloków kodu zaimplementowano globalny moduł obsługi wyjątków, czyli klasę opatrzoną adnotacją *@RestControllerAdvice*, zawierającą dwie metody z adnotacjami *@ExceptionHandler* – jedna służąca do obsługi nowo zdefiniowanych wyjątków, a druga do pozostałych. W obu tych metodach złapany wyjątek jest dodawany do logów serwera, a następnie zwracana jest odpowiedź z odpowiednim kodem błędu. Tak zdefiniowany moduł pozwala na obsługę wszystkich wyjątków występujących na serwerze po dowolnym żądaniu obsłużonym przez każdy z kontrolerów.

7.3 Uwierzytelnienie i autoryzacja użytkownika (Anna Malizjusz)

Podstawowy mechanizm uwierzytelnienia i autoryzacji opiera się na standardzie opisanym po raz pierwszy w 2010 roku jako JSON Web Token. Jest on "kompaktowym i bezpiecznym sposobem przesyłania informacji między dwiema stronami" (RFC 7519[8]). Użyto biblioteki *io.jsonwebtoken.jjwt*[9], która dostarcza interfejs do generowania i odczytywania tokenów w przystępny sposób.

Uwierzytelnienie użytkownika odbywa się podczas logowania. Serwer porównuje podany adres email oraz zmodyfikowane funkcją mieszającą hasło z danymi zapisanymi w bazie danych. Jeżeli informacje są zgodne, na ich podstawie jest generowany token. JWT pozwala na dodanie twierdzeń (ang. claims) i ustawienie czasu ważności, które zostaną zaszyfrowane przy pomocy algorytmu HS256 i sekretnego klucza. Używane są następujące twierdzenia określające podane cechy tokenu:

- *iss* - wydawca tokenu (ang. issuer),
- *sub* - podmiot (ang. subject),
- *email* - adres email użytkownika,
- *id* - id użytkownika,
- *generatedTimestamp* - czas wygenerowania tokenu.

Przykładowy token zapisany składający się z 3 części XXX.YYY.ZZZ zaprezentowano poniżej:

eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBY2Nlc3NUb2tlbiIsImdlbmVyYXRlZFRpbWVzdGFtcCI6eyJ5Z
WFyIjoyMDE5LCJtb250aCI6Ik5PVkVNQkVSIiwibW9udGhWYWx1ZSI6MTEsImRheU9mTW9udG
giOjE3LCJjaHJvbm9sb2d5Ijp7ImlkIjoiSVNPIiwiY2FsZW5kYXJUeXBlIjoiaXNvODYwMSJ9LCJkYXl
PZldlZWsiOiJTVU5EQVkiLCJsZWFwWWVhciI6ZmFsc2UsImRheU9mWWVhciI6MzIxLCJlcmEiOi
JDRSJ9LCJpc3MiOiJUcmF2ZWxBcHBfU2VydmVyIiwiaWQiOjEwLCJleHAiOjE1NzQxMDc4NTksI
mVtYWlsIjoicXFxMSJ9.vs9PVbgkzxsLpRqxXY0Jaey6fmMXdOLwQW_dUe9Xxcw

Pierwsza część to zakodowany nagłówek, który zawiera algorytm szyfrujący oraz typ tokenu. Kolejna jest zawartość, którą tworzą określone przy generacji tokenu twierdzenia. Ostatnia część to podpis. Użyto w nim znanego tylko serwerowi sekretnego klucza, niezbędnego do odszyfrowania otrzymanego od klienta tokenu.

Użytkownik aplikacji mobilnej przechowuje swój token i każdorazowo dołącza go do wysyłanych zapytań. Aplikacja serwerowa przy użyciu klucza odczytuje go i weryfikuje otrzymane dane, w szczególności datę ważności tokenu oraz id użytkownika.

Rozszyfrowany token jest zapisywany w formacie JSON w postaci, która jest czytelna i zrozumiała dla człowieka. Dzięki użytej bibliotece *io.jsonwebtoken.jjwt* z tokenu w formacie JSON można odczytywać dane jak ze słownika, np. *Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).body["id"].toString()*, aby odczytać wartość twierdzenia o nazwie "id".

PAYLOAD: DATA


```
{
  "sub": "AccessToken",
  "generatedTimestamp": {
    "year": 2019,
    "month": "NOVEMBER",
    "monthValue": 11,
    "dayOfMonth": 17,
    "chronology": {
      "id": "ISO",
      "calendarType": "iso8601"
    },
    "dayOfWeek": "SUNDAY",
    "leapYear": false,
    "dayOfYear": 321,
    "era": "CE"
  },
  "iss": "TravelApp_Server",
  "id": 10,
  "exp": 1574107859,
  "email": "qqq1"
}
```


Rysunek 7.1: Rozszyfrowana zawartość podanego powyżej tokenu.

7.4 Komunikacja z zewnętrznym API (Anna Malizjusz)

Potrzeba wyszukiwania infromacji o miejscach na świecie wymusiła korzystanie z zewnętrznych dostawców danych. Po analizie przeprowadzonej na etapie tworzenia specyfikacji wymagań systemowych zdecydowano o użyciu Here API[21]. Próba implementacji rozwiązania zmusiła programistów do skorzystania zarówno z wybranego dostawcy, jak i z początkowo odrzuconego Google API[22], które oferowało dużo bogatszą bazę sposobów transportu.

Podstawowym problemem było odczytanie danych. Wynikiem wysłania zapytania do dostawcy informacji był ciąg znaków zawierający znaczą ilość danych nie tylko o temacie zapytania. W przypadku każdego typu żądania przeanalizowano wynik i zdefiniowane wspólną metodę wyszukującą oraz wybierającą niezbędne informacje. Następnie należało je dopasować do potrzeb - nie wszystkie informacje były potrzebne aplikacji. Napisano funkcję zmieniającą ciąg znaków na instancję klasy. Skorzystano z możliwości klasy *JsonParser*[23] oraz *GsonBuilder*[24].

Należało też uwzględnić sposób zapisanych informacji. Wiele znaków, np. "'" było zakodowanych, aby uniemożliwić wstrzykiwanie złośliwyego kodu. W celu rozwiązania tego problemu skorzystano z funkcji *StringEscapeUtils.unescapeHtml3()*[25], która zmieniła formę znaków z kodowej na graficzną.

7.5 Wspólne klasy serwera i aplikacji

Podczas implementacji napotkano na problem powtarzających się klas w identycznej formie po stronie aplikacji i serwera. Stwarzało to kłopot przy edycji - obie wersje pliku musiały być zawsze identyczne. Spróbowano rozwiązać to na następujące sposoby.

7.5.1 Oddzielny projekt dodany jako moduł (Dorota Tomczak)

Pierwszym pomysłem na wydzielenie wspólnych klas z projektu aplikacji i serwera było utworzenie osobnego projektu, który by te klasy zawierał. Następnie nowy projekt mógłby być dołączony jako moduł do obu projektów, co jednak okazało się być nietrywialnym rozwiązaniem między innymi ze względu na różnice w plikach gradle - plik build.gradle w aplikacji mobilnej został napisany w języku Groovy, natomiast ten na serwerze w Kotlinie. Obie próby stworzenia takiego pliku dla wspólnego modułu w każdym ze wspomnianych języków nie powiodły się. Moduł mógł działać z określonym plikiem gradle tylko dla jednego z projektów jednocześnie. IDE na którym otwarty był serwer podpowiadało by usunąć wersję Kotlina w build.gradle, natomiast IDE aplikacji mobilnej podpowiadało wykonanie przeciwnej akcji. Po wielu próbach, ostatecznie porzucono zaproponowany pomysł na stworzenie osobnego projektu ze współdzielonymi klasami.

7.5.2 Submoduły w repozytorium (Anna Malizjusz)

Podjęto próbę dodania repozytorium zawierającego tylko wspólne pliki, a następnie wstawienia go do głównego repozytorium jako submoduły. Stworzyło to kolejne problemy w postaci konieczności częstych aktualizacji submodułów.

Zrezygnowano z submodułów i wykorzystano symboliczne powiązania. Git traktował powiązany folder jako istniejący, więc nie powodowało to dodatkowych problemów. Plik zmieniony i zapisany w jednej z lokalizacji był automatycznie odwzorowywany w drugiej. Rozwiązanie to, choć z pozoru skuteczne nie mogło być wykorzystane przez wszystkich członków zespołu ze względu na różnice w systemie operacyjnym. Używane były zarówno komputery z system Window, oparte na Linuxie oraz MacOS. Ostatecznie próby rozwiązania zarzucono.

7.6 Rekomendacja miejsc z wykorzystaniem Collaborative Filtering (Dorota Tomczak)

Po dodaniu elementu planu podróży można wejść w jego szczegóły i dokonać oceny miejsca, które znajduje się w planie. Oceny miejsc w skali od jednego do pięciu są przechowywane wraz z użytkow-

nikami, którzy wystawili daną ocenę w osobnej tabeli bazy danych. Na podstawie tych ocen mogą być następnie obliczane rekomendacje przy użyciu techniki zwanej *Collaborative-Filtering*, a w szczególności jej odmianą opartą na użytkowniku (ang. user-based).

Do implementacji wspomnianej metody wyznaczania poleceń wykorzystano bibliotekę *Apache Mahout*[18], która oferuje wiele implementacji algorytmów opartych o uczenie maszynowe. Po wskazaniu źródła danych, czyli w tym przypadku tabeli w bazie, utworzono model danych, który posłużył do wykonania niezbędnych obliczeń. Wyznaczenie polecanych miejsc przebiega w następujący sposób: wyliczane jest podobieństwo między użytkownikami algorytmem współczynnika korelacji Pearsona, algorytm k – najbliższych sąsiadów dla k równego 2 wyznacza najbardziej polecane miejsca, a na koniec na podstawie otrzymanych wyników tworzony jest obiekt klasy *GenericUserBasedRecommender*. Wywołanie metody *recommend* na obiekcie z podanym identyfikatorem użytkownika i liczbą rekomendacji, które ma zwrócić, skutkuje otrzymaniem listy identyfikatorów polecanych miejsc, jeśli zostały jakieś znalezione.

Testy przedstawionego rozwiązania aplikacją *Postman*[19] wykazały, że zwrócenie odpowiedzi trwa bardzo długo (nawet 20 s), a im większa liczba danych w bazie tym czas oczekiwania się wydłużał. Aby uniknąć konieczności oczekiwania na wynik po stronie aplikacji mobilnej, zdecydowano na prezentację polecanych miejsc w postaci powiadomień, czyli w momencie gdy serwer jest gotowy na wysłanie rekomendacji wysyła odpowiednie powiadomienie, a do tego czasu użytkownik może dalej swobodnie nawigować po aplikacji.

W celu wysyłania powiadomień od serwera do aplikacji mobilnej skorzystano z usługi *Firebase Cloud Messaging*[20]. Serwer po obliczeniu polecanych miejsc buduje wiadomość w postaci mapy, która ma zostać wysłana na urządzenie o określonym unikalnym tokenie. Za wysłanie odpowiada instancja klasy *FirebaseMessaging*. W konsoli usługi można zobaczyć statystyki wysłanych wiadomości, które zawierają między innymi informacje o tym ile z nich zostało otwartych przez użytkowników. Do odbierania wiadomości po stronie aplikacji mobilnej zaimplementowano serwis, który dzięki temu, że dziedziczy po *FirebaseMessagingService* może reagować na zdarzenia takie jak nadejście wiadomości oraz zmiana tokena.

Gdy serwis odbierze wiadomość, odczytuje ją, a następnie tworzy powiadomienie, ustawiając jego parametry takie jak m.in. jego tytuł, priorytet, dźwięk. Powiadomienie jest następnie obsługiwane przez *NotificationManager* i wyświetla się na ekranie urządzenia mobilnego. Użytkownik może powiadomienie od razu usunąć lub otworzyć. W drugim przypadku zostaje on przekierowany do aplikacji mobilnej, gdzie zostaje mu zaprezentowana lista polecanych miejsc wraz z nazwą i adresem.

7.7 Dostęp do bazy danych (Magdalena Solecka)

Do implementacji połączenia z bazą został wykorzystany PostgreSQL JDBC Driver. Zastosowano wzorzec Singleton, który w języku kotlin implementuje się używając słowa kluczowego [**?**] zamiast [**?**].

Zaimplementowano również wzorzec repozytorium (ang. repository pattern). Stworzono bazowy interfejs generyczny IRepository oraz jego implementację - Repository zawierajace operacje typu CRUD. Repozytoria dla poszczególnych modeli DAO dziedziczą po generycznej klasie bazowej Repository. Im-

plementują również własny interfejs z funkcajmi wymaganymi dla danej tabeli. Problemem przy wykorzystaniu typu generycznego okazało się tworzenie nowego obiektu, dlatego postanowiono wymusić na programistach implementację dodatkowej funkcji T tworzący ten obiekt dodając ją do interfejsu IRepository. W celu zmniejszenia ilości powtarzającego się kodu wyodrębniono również stałe fragmenty tekstu z zapytań SQL takie jak na przykład [**?**] w postaci stałej selectStatement. Stałe zostały dodane do klasy bazowej Repository jako abstrakcyjne. W podobny sposób rozwiązano problem powtarzających się nazw tabel i kolumn. Nie udało się jednak utworzyć stałej abstrakcyjnej, wynikiem prób była pusta wartość w zapytaniach w miejscu gdzie powinna była znaleźć się na przykład nazwa tabeli.

Rozdział 8

Baza danych

8.1 Wybór (Magdalena Solecka)

Przy wyborze systemu do zarządzania bazą danych były brane pod uwagę Neo4j oraz PostgreSQL. Pierwsza z wymienionych wzbudziła zainteresowanie ze względu na planowaną implementację algorytmu rekomendującego collaborative filtering. Jej zaletą, jako grafowej bazy danych, byłoby szybkie wydobycie podobieństw pomiędzy użytkownikami na podstawie ocenionych przez nich miejsc. Uznano jednak, że zespół pracował do tej pory z relacyjnymi bazami danych i językiem sql a dłuższe wykonanie algorytmu nie spowoduje problemów z wydajnym działaniem pozostałych funkcjonalności aplikacji, dlatego ostatecznie zdecydowano się na PostgreSQL chwalonego za wysoką wydajność, oferujacego wsparcie typu BJSON, który mógłby być przydatny przy przechowywaniu większej ilości informacji o obiektach, a także znajomą zespołowi składnię zapytań. Bazę danych wdrożono w serwisie ElephantSQL z instancją [**?**] która pozwalała na zapis 20 MB danych, 5 równoległych połączeń z bazą co było w zupełności wystarczające dla środowiska testowego.

8.2 placeholder - Schemat ERD (Magdalena Solecka)

8.3 Praca z bazą danych (Magdalena Solecka)

Warstwę danych tworzono podejściem [**?**] (ang. code first), głównie z powodu trudności z przewidzeniem, które struktury dostarczane przez zewnętrzne API uda się otrzymać i które informacje są niezbędne, aby znacząco nie zmniejszyć szybkości aplikacji, a o które można zapytać źródło podczas ładowania elementów ekranu. Stwarzało to problemy z kompatybilnością wsteczną podczas rozwijania aplikacji, dodawania nowych kolumn, zmiany typów. Początkowe rozwiązanie wymagało powiadomienia wszystkich programistów w zespole o godzinach, w których baza danych może przestać działać poprawnie, a przed jakąkolwiek dalsza pracą konieczne było ściągnięcie najnowszych zmian z repozytorium github. Drugi w kolejności wprowadzonym sposobem było utworzenie tabeli tymczasowej - wykonanie kopii tabeli oraz wprowadzenie zmian w kopii. Podmiany dokonywano przy dołączaniu zmian z gałęzi podrzędnej do gałęzi develop w serwisie github. Nadal jednak wymagane było pobranie najnow-

szej wersji kodu. Z początku wymagało to zaplanowania pracy w ten sposób, aby lokalny kod nadawał się do zapisu (ang.commit) w chwili zmian w bazie, ale problem zakończył się odkryciem funkcji [**?**] oferowanej przez gita.

Rozdział 9

Testy

9.1 Testy jednostkowe i instrumentalne aplikacji mobilnej (Dorota Tomczak)

Dla aplikacji mobilnej napisano 23 testy jednostkowe oraz 9 instrumentalnych, które obejmują klasy i metody komponentów służących do logowania, rejestracji oraz przekierowania po starcie aplikacji. Testy jednostkowe, czyli takie które weryfikują poprawność realizacji pojedynczych funkcjonalności aplikacji, zostały zbudowane w oparciu o następujące zależności: *JUnit 4*[26], *JUnitParams*[**?**] do tworzenia testów parametryzowanych oraz bibliotekę *MockK*[28] służącą do mockowania obiektów w języku Kotlin. Testy te służą do weryfikacji działania metod zdefiniowanych w prezenterach, z tego względu przed startem każdego z nich odpowiedni prezenter nie jest mockowany, ale inicjalizowany w sposób, który umożliwia jego „szpiegowanie". Aby zasymulować asynchroniczną komunikację sieciową pomiędzy aplikacją a serwerem, która odbywa się poprzez zastosowanie biblioteki *RxJava*[29], stworzono klasę pomocniczą *RxImmediateJavaSchedulerRule*, która ustawia wszystkie metody, rozpoczynające nowe wątki, aby korzystały zamiast tego z wątku obecnego.

Testy instrumentalne, czyli takie które działają na emulatorze lub na fizycznym urządzeniu mobilnym, zostały zrealizowane dla dwóch aktywności – *SignInActivity* oraz *SignUpActivity* i testują, czy na ekranie wyświetlają się prawidłowe informacje, czy akcje użytkownika wywołują odpowiednie metody oraz czy aplikacja reaguje na nie w odpowiedni sposób. Wykorzystują wiele zależności, w tym między innymi *Espresso*[30] i *MockK-Android*[31]. W klasach testowych tworzone są role aktywności (*activityScenarioRole*), na których następnie można wywoływać określone akcje np. wciśnięcie przycisku, czy wpisanie tekstu w polu do tego przeznaczonym.

Wszystkie testy, zarówno instrumentalne jak i jednostkowe zostały podzielone na trzy segmenty:

- *given* - definicja danych wejściowych oraz mockowanie wywołań metod i obiektów,
- *when* - wywołanie testowanej metody lub wykonanie akcji,
- *then* - weryfikacja otrzymanych rezultatów po wywołaniu metody lub wykonaniu akcji.

Do nazewnictwa testów przyjęto następującą konwencję: co powinno się stać, gdy zostaną spełnione określone warunki. Przykładowo "Should display snackBar with info message when given password in

SignUp is incorrect", co w tłumaczeniu oznacza "Powinien się wyświetlić snackBar z komunikatem informacyjnym, gdy hasło podane w SignUp jest nieprawidłowe".

```
@Test
fun should_DisplaySnackBar_When_EmailIsAlreadyTaken() {

    // given
    val email = "email"
    val password = "password"
    val confirmPassword = "password"
    val hashedPassword = "hashed password"

    mockkObject(PasswordUtils)
    every { PasswordUtils.hashPassword(password) } returns hashedPassword

    val signUpRequest = SignUpRequest(email, hashedPassword)
    mockkObject(CommunicationService)
    every { CommunicationService.serverApi.register(signUpRequest) } returns Single.just(
            Response(ResponseCode.EMAIL_TAKEN_ERROR, Unit))

    // when
    onView(withId(R.id.editTextEmail)).perform(typeText(email))
    onView(withId(R.id.editTextPassword)).perform(typeText(password))
    onView(withId(R.id.editTextConfirmPassword)).perform(typeText(confirmPassword), closeSoftKeyboard())
    onView(withId(R.id.buttonSignUp)).perform(click())

    // then
    onView(withId(com.google.android.material.R.id.snackbar_text))
            .check(matches(withText("Account exists")))
}
```

Rysunek 9.1: Przykładowy test instrumentalny.

Rozdział 10

Placeholder - Podręcznik użytkownika

Rozdział 11

Podsumowanie (Anna Malizjusz)

Wynikiem pracy inżynierskiej jest zaimplementowana, działająca w środowisku testowym aplikacja mobilna oraz serwer REST. Zrealizowane zostały wszystkie wymagania o wysokim priorytecie niezbędne do akceptacji projektu. Najważniejsze z nich to:

- rejestracja i logowanie użytkownika,
- dodanie, przeglądanie i edycja planu podróży i planu dnia,
- wyszukanie elementu w pobliżu danej lokalizacji, w szczególności zakwaterowania,
- wyszukanie innego użytkownika i udostępnienie mu podróży,
- dodanie oceny do odwiedzonego miejsca,
- otrzymanie propozycji na podstawie ocen,
- wyszukanie transportu między lokalizacjami,
- skanowanie biletów.

Zaimplementowano też kilka dodatkowych funkcji, takich jak udostępnienie odwiedzonego miejsca w serwisie Facebook oraz oznaczenie planu dnia jako wykonany. Uwzględniano ostrzeżenia o użyciach przestarzałych funkcji, aby w aplikacji korzystano z najnowszych i najbezpieczniejszych praktyk.

Aplikację mobilną przetestował zespół programistów oraz kilku testerów. Nie stwierdzono rażących błędów, które uniemożliwiałyby korzystanie z zaimplementowanego rozwiązania. Interfejs okazał się łatwy w obsłudze dla młodych ludzi.

W trakcie projektu inżynierskiego zespół doświadczył trudności w implementacji rozwiązania "od zera", bez bazowego kodu ani dokładnych. Brakowało również praktyki w prowadzeniu względnie obszernego i skomplikowanego przedsięwzięcia, jakim jest stworzenie działającej aplikacji w pół roku. Wielokrotnie wybrane rozwiązania były uznawane za niewystarczające i zmieniane na lepsze. Początkowo tryb pracy i stosowane konwencje często się zmieniały, jednak w pierwszym etapie projektu udało się ustabilizować wewnętrzne wymagania. Dzięki temu recenzje kodu przebiegały sprawniej.

Spotykano się z problemem braku pomocnych informacji w internecie. Część używanych rozwiązań zostało oznaczonych jako przestarzałe i należało samodzielnie znaleźć aktualniejsze. Czasem część znalezionych informacji była specyficzna dla języka Java i nie istniała banalna konwersja do używanego języka Kotlin.

Podczas pracy nad aplikacją mobilną i serwerową zdefiniowano kolejne przydatne funkcjonalności lub ulepszenia istniejących. Postanowiono umożliwić użytkownikowi otrzymywanie powiadomień na adres email, a także umożliwić edycję planu dnia. Dodatkowo należy rozszerzyć aplikację o możliwość zmienienia oraz przypomnienia hasła. W serwisie GitHub stworzono nowe zadania odnośnie poprawienia jakości kodu obsługującego zapytania na serwerze, w bazie danych oraz w aplikacji mobilnej. Zdecydowano o przyszłej zmianie niektórych z elementów aplikacji, np. sposobu wyświetlania obiektów na mapie. Jednocześnie planowane jest zaimplementowanie kolejnych funkcjonalności, które zostały oznaczone jako mniej ważne.

Zdecydowano o późniejszym dokładniejszym pokryciu kodu testami jednostkowymi, a także o dodaniu większej liczby testów integracyjnych. W celu łatwiejszego i bezpieczniejszego rozwoju aplikacji zostaną zaimplementowane mechanizmy ciągłej integracji (ang. continous integration), które zapobiegną problemom z kompilacją oraz działaniem aplikacji na głównej gałęzi (ang. branch) repozytorium (master).

Zaimplementowane rozwiązanie, dzięki zachowaniu dobrych praktyk programistycznych podczas pisania kodu, może być dalej rozwijane w ramach pracy magisterskiej. Dotychczasowa praca zespołu zwiększyła jego umiejętności, zarówno programistyczne, jak i zdolność do współpracy z innymi członkami grupy.

Podział zadań implementacyjnych

Tabela 11.1: Autorzy zaimplementowanych funkcji

Dorota Tomczak	**Magdalena Solecka**	**Anna Malizjusz**	**Karolina Makuch**
Zaprojektowanie i implementacja bazowej aplikacji mobilnej (wzorzec MVP, wstrzykiwanie zależności Dagger2)	Wybór systemu zarządzania bazą danych Postgresql	Projekt i implementacja bazowej postaci serwera oparta o framework Spring Boot	Implementacja wyświetlania znajomych oraz wyszukiwania użytkowników w celu dodania ich do znajomych
Obsługa sytuacji wyjątkowych na serwerze	Implementacja podstawowego dostępu do bazy wyznaczającego schemat dla kolejnych modeli przy użyciu repository pattern	Implementacja mechanizmów uwierzytelniania i autoryzacji na serwerze	Implementacja manualnej realizacji planu
Implementacja skanera z użyciem OpenCV	Pierwsza implementacja ekranu logowania i rejestracji	Wyszukiwanie obiektów w aplikacji - implementacja wyboru miejsca z mapy	Implementacja udostępniania planu podróży wybranym znajomym
Tworzenie planów podróży po stronie aplikacji (formularz dodawania i chronologiczne wyświetlanie listy)	Dodawanie planów podróży po stronie serwera	Wyświetlanie szczegółowych informacji o miejscu w planie dnia	Implementacja udostępniania planu dnia w medium społecznościowym (Facebook)
Obsługa pobierania i przesyłania plików (skany i zdjęcia podróży)	Implementacja testów automatycznych operacji dostępu do bazy danych	Odczytywanie i korzystanie z informacji otrzymanych z Here API i Google API	

Integracja w aplikacji komponentu *Navigation drawer*	Implementacja trybu usuwania w aplikacji mobilnej	Komunikacja z serwerem w aplikacji mobilnej	
Implementacja rekomendacji miejsc i wysyłanie ich w formie powiadomień	Implementacja dodawania podróży oraz zmiany nazwy podróży	Implementacja logiki rejestracji i logowania w aplikacji	
Testy jednostkowe i instrumentalne dla logowania, rejestracji i komponentu *launcher*			

Bibliografia

[1] developer.here.com

[2] developers.google.com/android/reference/com/google/android/gms/location/package-summary

[3] gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide

[4] https://developer.android.com/reference/android/widget/SearchView

[5] https://developer.android.com/reference/android/content/ContentProvider

[6] https://github.com/umano/AndroidSlidingUpPanel

[7] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[8] https://tools.ietf.org/html/rfc7519

[9] https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt

[10] https://square.github.io/retrofit/

[11] https://square.github.io/okhttp/

[12] https://github.com/google/dagger

[13] https://github.com/jbttn/SimpleDocumentScanner-Android

[14] https://opencv.org/

[15] https://developers.facebook.com/support/bugs/332619626816423/

[16] https://mobile.here.com/

[17] https://bumptech.github.io/glide/

[18] https://mahout.apache.org/

[19] https://www.getpostman.com/

[20] https://firebase.google.com/docs/cloud-messaging

[21] https://developer.here.com/

[22] https://cloud.google.com/maps-platform/routes/

[23] https://static.javadoc.io/com.google.code.gson/gson/2.8.5/com/google/gson/JsonParser.html

[24] https://static.javadoc.io/com.google.code.gson/gson/2.8.0/com/google/gson/GsonBuilder.html

[25] https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringEscapeUtils.htm

[26] https://junit.org/junit4/

[27] https://github.com/Pragmatists/JUnitParams

[28] https://mockk.io/

[29] https://github.com/ReactiveX/RxJava

[30] https://developer.android.com/training/testing/espresso

[31] https://mockk.io/ANDROID.html

[32] https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/

Spis rysunków

Spis tabel