AKADEMIA TARNOWSKA
Wydział Nauk Technicznych

Kierunek: ***Informatyka***

2025/2026

Alicja Borgula

PRACA INŻYNIERSKA

Projekt i implementacja aplikacji mobilnej do wymiany żywności

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2026*

Spis treści

1. Wstęp

Coraz więcej ludzi jest świadomych, jak ważna jest ekologia i dostrzega wielkość problemu marnowania żywności. Aby zmniejszyć skalę tego kłopotu, stajemy przed wyzwaniem opracowania narzędzi wspierających zarządzanie zasobami żywnościowymi, co może przynieść korzyści ekologiczne i ekonomiczne.

1.1 Motywacja

Obecnie mamy szeroki wybór produktów spożywczych, co sprzyja nadmiernemu gromadzeniu oraz wyrzucaniu pokarmu. Brak planowania posiłków prowadzi do mało efektywnych zakupów, co przynosi niepotrzebne wydatki i niszczenie środowiska.

Problem ten w jeszcze większym stopniu dotyczy branży gastronomicznej - restauracji, piekarni czy kawiarni, które z obawy przed niedoborem produktów dla klientów utrzymują duże zapasy żywności. W efekcie znaczna część jedzenia, mimo że nadaje się do spożycia, trafia na śmietnik.

Zarówno w Polsce, jak i na świecie, dostępne są już aplikacje wspierające ograniczanie marnowania żywności. Umożliwiają one właścicielom lokali gastronomicznych wystawianie paczek z nadmiarem jedzenia w znacznie niższych cenach. Dzięki temu przedsiębiorcy mogą ograniczyć straty, a klienci zyskują możliwość zakupu pełnowartościowych produktów w atrakcyjnych cenach.

Istnieje jednak luka rynkowa - większość tego typu rozwiązań skierowana jest do firm, a nie do osób prywatnych. Tymczasem wielu z nas również boryka się z problemem nadmiaru jedzenia, szczególnie po spotkaniach rodzinnych, imprezach czy innych wydarzeniach, kiedy przygotowane potrawy i wypieki często ulegają zmarnowaniu. Coraz więcej ludzi dostrzega wagę tego problemu i chce aktywnie mu przeciwdziałać.

1.2 Cel pracy

Celem pracy jest wykonanie mobilnej aplikacji umożliwiającej ograniczenie marnowania żywności poprzez ułatwienie wymiany produktów spożywczych pomiędzy użytkownikami. Aplikacja ma pozwalać osobom prywatnym na bezpłatne lub symboliczne przekazywanie nadmiaru jedzenia, zarówno gotowych potraw, jak i produktów spożywczych, innym użytkownikom w ich okolicy. Użytkownicy poszukujący jedzenia będą mogli z kolei odebrać produkty od osób, które przygotowały ich zbyt dużo.

1.3 Zakres pracy

Zakres pracy obejmuje zaprojektowanie, implementację oraz testowanie mobilnej aplikacji, której celem jest ograniczanie marnowania żywności poprzez umożliwienie użytkownikom dzielenia się nadmiarem jedzenia z innymi osobami.

W ramach projektu opracowano system składający się z dwóch głównych elementów:

- mobilnej aplikacji *Replate*, zaimplementowanej w języku Java w środowisku Android Studio,
- bazy danych *Firebase Firestore*, wykorzystywanej do przechowywania informacji o paczkach, użytkownikach oraz rezerwacjach.

Pracę podzielono na kolejne etapy umożliwiające uporządkowany rozwój. Zrealizowano następujące kroki:

- **Analiza wymagań użytkownika -** obejmowała rozpoznanie potrzeb przyszłych użytkowników, określenie dostępnych funkcji systemu oraz wymagań dotyczących interfejsu i prezentacji danych.
- **Dobór odpowiednich technologii -** polegał na wyborze narzędzi odpowiednich do opracowania nowoczesnej aplikacji mobilnej. Projekt zrealizowano w środowisku Android Studio z wykorzystaniem języka Java, bazy danych Cloud Firestore oraz usług Firebase. Funkcje mapowe zaimplementowano przy użyciu Google Maps, a obsługę płatności zapewniono używając Stripe.

- **Implementacja systemu -** w części praktycznej zrealizowano aplikację mobilną, obejmującą logikę działania, interfejs użytkownika i integrację z bazą danych. Całość połączono w spójnie działający system.
- **Testowanie -** na końcowym etapie przeprowadzono testy poszczególnych modułów oraz testy integracyjne sprawdzające współdziałanie elementów aplikacji. Pozwoliło to na wykrycie i usunięcie błędów.

2. Analiza biznesowa

W niniejszym rozdziale zostaną omówione podstawowe funkcjonalności, sposób interakcji użytkownika z systemem oraz cechy, które powinna zachować aplikacja. Celem etapu jest zrozumienie dlaczego i dla kogo została zaprojektowana.

2.1 Analiza rynku

W ostatnich latach obserwuje się wzrost zainteresowania tematem ekologii oraz ograniczania marnowania żywności. Coraz więcej osób poszukuje rozwiązań umożliwiających odpowiedzialne gospodarowanie produktami spożywczymi, co sprzyja rozwojowi aplikacji wspierających tego typu działania.

Na rynku działają już aplikacje o podobnym charakterze, takie jak *TooGoodToGo* czy *Foodsi*, które zyskują na popularności. Według dostępnych analiz, dzięki partnerom pierwszej z wymienionych w ciągu 6 lat działalności udało się uratować 15 milionów paczek, co daje wartość około 610 mln złotych. Aplikacja szczególnie się rozrosła w ostatnich kilku latach, co pokazuje zapotrzebowanie na narzędzia tego typu.

Analiza dostępnych rozwiązań wskazuje, że aplikacje skupiają się jednak na współpracy z lokalami gastronomicznymi, takimi jak sklepy czy restauracje, aby pozbyły się nadwyżek żywności. Na rynku jest więc luka, dająca możliwość rozwoju narzędzia, które będzie działać na podobnych zasadach, ale dedykowanemu innym odbiorcom. Aplikacja będąca tematem pracy odróżnia się tym, że koncentruje się na użytkownikach indywidualnych. Grupą docelową są studenci, rodziny oraz osoby samotne, które wolą podzielić się nadmiarem jedzenia, zamiast je wyrzucać. Jest to więc odpowiedź na potrzebę ograniczenia marnowania żywności w skali lokalnej.

2.2 Wymagania funkcjonalne

System powinien dostarczać użytkownikom funkcji, dzięki którym będą mogli w łatwy i szybki sposób pozbywać się nadmiaru żywności oraz odbierać paczki z ogłoszeń. Użytkownicy powinni mieć możliwość:

- **Rejestracji i logowania do konta** używając adresu e-mail i hasła, poprzez konto Google lub używając numeru telefonu.
- **Przeglądania ogłoszeń** w okolicy używając opcji z mapą, lub wyświetlając je w liście.
- **Filtrowania ogłoszeń** aby usprawnić proces wyszukiwania interesujących nas paczek, poprzez wybór kategorii oraz dnia i godziny odbioru.
- **Wyboru lokalizacji** poprzez wpisanie wybranego adresu lub nazwy miejscowości i wybierając zakres odległości od danego miejsca.
- **Dodawania do ulubionych** oznaczając konkretnych użytkowników, co ułatwiłoby wyszukiwanie dodanych przez nich ogłoszeń.
- **Wystawiania ogłoszeń** wpisując tytuł, opis, wybierając tagi, czas i lokalizację odbioru, podając numer kontaktowy oraz opcjonalnie cenę i zdjęcie.
- **Wystawiania opinii i oceny** osobie wystawiającej po zakończonym procesie odbioru.
- **Ustawienia** w swoim profilu adresu oraz numeru telefonu, aby były uzupełniane automatycznie przy dodawaniu ogłoszenia, oraz możliwość wyłączenia powiadomień.
- **Odbierania powiadomień** gdy ulubiony użytkownik doda paczkę, gdy jest czas na odbiór zamówienia, a także gdy ktoś zarezerwuje produkty lub anuluje rezerwację.
- **Dokonania płatności** za paczkę poprzez integrację *Stripe* (platforma do obsługi płatności online), wybierając jedną z kilku dostępnych opcji, np. karta płatnicza.

2.3 Wymagania niefunkcjonalne

W tym podrozdziale zostaną zaprezentowane kluczowe cechy systemu decydujące o jego jakości funkcjonowania i doświadczeniu użytkownika.

- **Wydajność:** aplikacja mobilna powinna się uruchamiać w czasie nie dłuższym niż 1,5 sekundy przy zimnym starcie, ponieważ badania dotyczące UX (*ang. user experience*) wskazują, że 53% użytkowników opuszcza aplikację lub stronę, jeśli ładowanie trwa dłużej niż 3 sekundy, co czyni 1,5 sekundy bezpieczną wartością docelową. Wszystkie akcje użytkownika powinny być przetwarzane w czasie poniżej 200 ms, ponieważ latencje wyższe niż ta wartość mogą być interpretowane jako wolne działanie systemu, co zmniejsza zadowolenie użytkownika [1].

- **Bezpieczeństwo:** hasła użytkowników powinny być przechowywane i zaszyfrowane w bazie danych, a dostęp do konta chroniony poprzez uwierzytelnienie i autoryzację. Danych płatniczych nie należy przechowywać lokalnie w aplikacji, a do komunikacji z serwerem powinien zostać użyty protokół *HTTPS*.

- **Niezawodność**: aplikacja wymaga stabilnego działania, bez nagłych przerw i awarii, a ważne operacje jak usuwanie ogłoszenia powinno się dodatkowo potwierdzać.

- **Dostępność i intuicyjny interfejs:** dla zapewnienia międzynarodowej dostępności interfejs użytkownika powinien być w języku angielskim, a jego wygląd warto utrzymać w sposób prosty i przejrzysty, aby ułatwić szybkie działania. Najważniejsze funkcje umieszczone w dolnym menu byłyby łatwo dostępne.

3. Stos technologiczny

Właściwy dobór technologii ma istotne znaczenie dla realizacji aplikacji mobilnej, która stanowi główny cel pracy. W tym rozdziale zostaną omówione rozwiązania użyte do opracowania projektu, aby spełniał wymagania współczesnych aplikacji.

3.1 Android Studio

Aplikacja została opracowana w oficjalnym środowisku do programowania aplikacji mobilnych - *Android Studio*, które oferuje narzędzia do testowania, pisania kodu i projektowania interfejsu użytkownika.

Android Studio to oficjalne zintegrowane środowisko programistyczne od Google, przeznaczone do projektowania i rozwijania aplikacji na system Android. Oparte na platformie *IntelliJ IDEA*, oferuje wiele narzędzi i funkcji, które ułatwiają cały proces wytwarzania oprogramowania - od pisania kodu po testowanie i debugowanie [6]. Jest darmowym narzędziem, z którego korzystają zarówno profesjonalni programiści, jak i osoby amatorsko projektujące aplikacje w językach *Java*, *Kotlin* lub *C++*. Oprogramowanie działa na systemach typu: *Windows*, *Linux* i *macOS* i jest oparte na środowisku *IntelliJ IDEA*. Zostało wprowadzone przez firmę *Google* jako alternatywę dla środowiska *Eclipse* [5].

W *Android Studio* wbudowane jest narzędzie do projektowania widoków aplikacji jako tryb *WYSIWYG* (ang. *What You See Is What You Get*, czyli „To, co widzisz, jest tym, co otrzymasz") [11], który pozwala programiście widzieć podgląd ekranu końcowego. Emulator Androida symuluje urządzenie na komputerze, dzięki czemu można testować aplikację na różnych urządzeniach i poziomach API Androida (interfejsu programowania aplikacji, określającego wersję systemu i dostępne funkcjonalności) bez potrzeby posiadania fizycznego urządzenia. Emulator oferuje następujące zalety:

- **Elastyczność:** oprócz możliwości symulowania różnych urządzeń i poziomów API, emulator posiada gotowe konfiguracje dla telefonów, tabletów, urządzeń z *Wear OS*, *Android Automotive OS* oraz *Android TV*.

- **Wysoka wierność:** emulator zapewnia niemal wszystkie funkcje prawdziwego urządzenia z Androidem. Można symulować przychodzące połączenia telefoniczne, wiadomości tekstowe, określać lokalizację urządzenia, symulować prędkości sieci, obrót i inne czujniki, uzyskiwać dostęp do sklepu *Google Play* i wiele więcej.
- **Szybkość:** testowanie aplikacji na emulatorze jest po części szybsze i łatwiejsze niż na urządzeniu fizycznym. Na przykład transfer danych do emulatora jest szybszy niż do urządzenia podłączonego przez USB [15].

3.2 Java

Aplikacja została zrealizowana w języku programowania *Java*, dlatego w tej części przedstawiono jego krótką charakterystykę oraz najważniejsze cechy.

W 1995 roku firma *Sun Microsystems* wprowadziła na rynek nowy język programowania - *Javę*. Był to język składniowo oparty na znanym i sprawdzonym języku *C++*. W przeciwieństwie jednak do swego pierwowzoru język *Java* nie musiał zachowywać kompatybilności wstecznej z wcześniejszymi językami. Dzięki temu jest on bardziej spójny i „lżejszy” od *C++*. Dziś to nie tylko język programowania, ale cały olbrzymi zbiór technologii i produktów z nim powiązanych. Programy napisane w *Javie* działają na przeróżnych maszynach i systemach poczynając od telefonów komórkowych poprzez komputery PC, a kończąc na wysokowydajnych serwerach [4].

Język miał być odpowiedzią na potrzeby programistów, którzy szukali narzędzia przyjaznego użytkownikowi, o prostej składni i bogatej bibliotece. Mając to na uwadze autorzy opracowali listę 11 kluczowych elementów, które powinien zawierać [2]. Poniżej przedstawiono wybrane z nich:

- prostota,
- obiektowość,
- niezawodność,
- przenośność,
- wielowątkowość.

Początkowo język miał być wykorzystany do rozwoju oprogramowania urządzeń domowego użytku, natomiast wraz z rozwojem komputerów oraz Internetu uznano, że bardziej przydatny będzie w obszarze aplikacji desktopowych oraz internetowych [3]. Od samej chwili powstania do dzisiaj, *Java* zajmuje wysokie miejsca w rankingu popularności. Według indeksu społeczności programistów *TIOBE Java* znajduje się obecnie na trzecim miejscu na liście [13].

3.3 Cloud Firestore

Do przechowywania danych w aplikacji wykorzystano *Cloud Firestore*, nierelacyjną bazę danych w chmurze oferowaną przez *Google* w ramach platformy *Firebase,* która udostępnia narzędzia do rozwoju aplikacji.

Cloud Firestore to elastyczna i skalowalna baza danych przeznaczona do projektowania aplikacji mobilnych, webowych i serwerowych, oferowana w ramach platform *Firebase* oraz *Google Cloud*. Podobnie *jak Firebase Realtime Database*, czyli usługa do przechowywania i synchronizacji danych pomiędzy klientami, *Firestore* umożliwia aktualizację danych na bieżąco, a także oferuje wsparcie trybu *offline* dla urządzeń mobilnych i aplikacji webowych, co pozwala projektować responsywne aplikacje działające niezależnie od opóźnień sieciowych czy dostępności Internetu. *Cloud Firestore* umożliwia również płynną integrację z innymi produktami *Firebase* i *Google Cloud*, w tym z *Cloud Functions*. Kluczowe możliwości:

- **Elastyczność:** model danych w *Cloud Firestore* obsługuje hierarchiczne struktury danych. Dane przechowuje się w dokumentach, zorganizowanych w kolekcje i mogą zawierać złożone, zagnieżdżone obiekty oraz podkolekcje.
- **Wyszukane zapytania:** w *Cloud Firestore* można formułować zapytania, aby pobierać pojedyncze, konkretne dokumenty lub wszystkie dokumenty w kolekcji spełniające określone kryteria. Zapytania mogą obejmować wiele filtrów łańcuchowych oraz łączyć filtrowanie z sortowaniem.

- **Aktualizacje w czasie rzeczywistym:** *cloud Firestore* synchronizuje dane, aby aktualizować je na każdym podłączonym urządzeniu. Jednocześnie został zaprojektowany tak, aby efektywnie obsługiwać proste, jednorazowe zapytania [9].

3.4 Google Maps API

Do obsługi map i lokalizacji w aplikacji wykorzystano *Google Maps API*. Umożliwiło to dynamiczne prezentowanie danych geograficznych oraz dodanie funkcji opartych na lokalizacji, takich jak wyszukiwanie paczek czy wyznaczanie tras.

Google Maps API to narzędzie analizy lokalizacji, które umożliwia firmom osadzanie map, obliczanie tras, wyświetlanie punktów zainteresowania oraz integrację funkcji geolokalizacji w aplikacjach internetowych i mobilnych. Dzięki wykorzystaniu *Google Maps API* firmy mogą optymalizować doświadczenie klientów poprzez lokalizatory sklepów, śledzenie dostaw i płynną nawigację [19]. Mapa dodana przy użyciu tego *API* może zostać modyfikowana. Przykładowe elementy, które możemy dodać na mapie osadzonej poprzez *Google Map API*:

- kolory mapy,
- znacznik,
- dodać “dymek” do pineski,
- jakie informacje będą wyświetlane na mapie (np. nazwy ulic).

3.5 Google Places API

Do obsługi wyszukiwania i autouzupełniania lokalizacji w aplikacji wykorzystano Google Places API. Dzięki temu użytkownik może szybko wyszukać interesujące go miejsca, a aplikacja może je poprawnie wyświetlić na mapie lub użyć do wyznaczania tras. Interfejs Places API to usługa, która akceptuje żądania HTTP dotyczące danych o lokalizacji za pomocą różnych metod. Zwraca sformatowane dane o lokalizacji oraz obrazy obiektów, miejsc geograficznych lub ważnych punktów zainteresowania [14].

3.6 Firebase Authentication

Do obsługi logowania i autoryzacji użytkowników w aplikacji wykorzystano *Firebase Authentication*, która oferuje backend, zestawy SDK (ang. *Software Development Kit,* czyli pakiety narzędzi programistycznych) i gotowe biblioteki interfejsu użytkownika do uwierzytelniania użytkowników w aplikacji. Obsługuje logowanie za pomocą haseł, numerów telefonów oraz popularnych zewnętrznych dostawców tożsamości, takich jak *Google*, *Facebook* i inne [12].

3.7 Material Design

W projektowaniu interfejsu użytkownika w aplikacji kierowano się *Material Design* - zestawem wytycznych i komponentów dostarczonych przez *Google*, których celem jest powstawanie spójnych i intuicyjnych interfejsów.

Jeszcze w 2014 roku *Google* podczas corocznej konferencji I/O ogłosił wprowadzenie *Material Design*. Była to odpowiedź na często spotykane w tamtym okresie nieprzyjazne dla użytkowników i chaotyczne style projektowania stron internetowych. Choć od tego czasu minęło już ponad 10 lat, *Material Design* ma wciąż ogromne znaczenie. Umożliwia on projektantom opracowywać witryny skoncentrowane na rezultacie. To zestaw wytycznych, który definiuje zasady projektowania dotyczące między innymi: siatek, koloru, obrazów[10].

3.8 Stripe

Do obsługi płatności w aplikacji wykorzystano *Stripe.* Jest to międzynarodowa firma świadcząca usługi finansowe, oprogramowanie do zarządzania płatnościami oraz *API* dla firm internetowych i *e-commerce*. Do dyspozycji przedsiębiorców *Stripe* udostępnia także terminal płatniczy do obsługi sprzedaży stacjonarnej [16]. *Stripe Android SDK* umożliwia używanie gotowych elementów interfejsu użytkownika, które mogą być wykorzystane od razu do

gromadzenia danych płatniczych użytkowników. Udostępnione niskopoziomowe *API* pozwala budować całkowicie własne, spersonalizowane interfejsy płatności.

SDK umożliwia wprowadzanie numerów kart kredytowych przy zachowaniu zgodności z normami *PCI (*ang. *Payment Card Industry),* co oznacza, że wrażliwe dane są przesyłane bezpośrednio do *Stripe*, zamiast przechodzić przez serwer aplikacji. W ramach *SDK* dostępne są natywne ekrany i elementy do obsługi płatności. Przykładem jest *PaymentSheet* – gotowy interfejs łączący wszystkie kroki płatności (przekazanie danych i potwierdzenie płatności) w jednym oknie aplikacji [17].

4. Implementacja systemu

W tym rozdziale zostanie opisana techniczna strona działania aplikacji *Replate*, obejmująca jej budowę, strukturę, zastosowane moduły, sposób komunikacji z zewnętrznymi usługami oraz mechanizmy zapewniające poprawną integrację wszystkich komponentów.

4.1 Architektura systemu

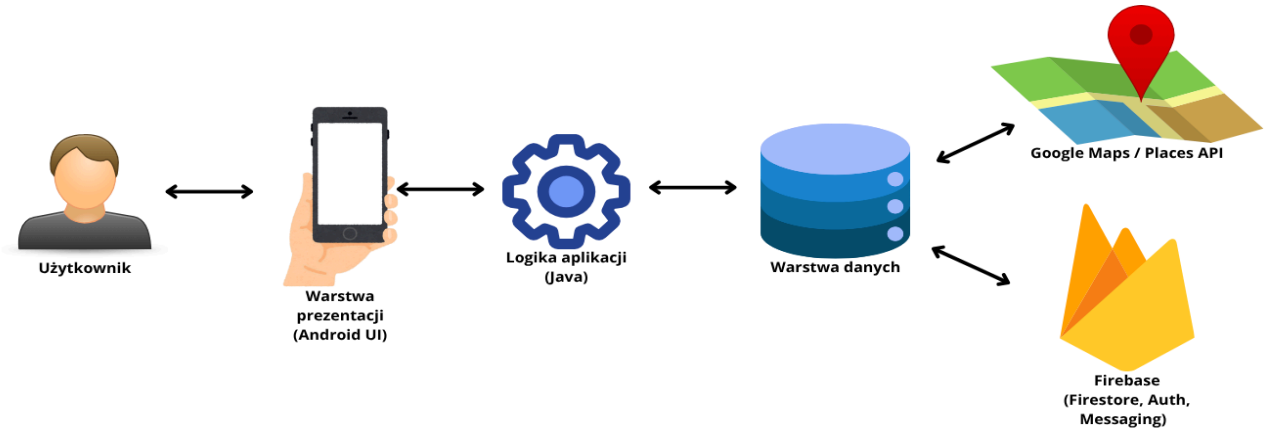
Aplikację zbudowano w oparciu o architekturę klient-serwer. Część kliencka została zaimplementowana w środowisku *Android Studio* z wykorzystaniem języka *Java*. Logikę serwerową zrealizowano używając platformy *Firebase*, która pełni funkcję *"Backend as a Service - BaaS"*. Jest to model liczbowy, który umożliwia deweloperom korzystanie z gotowych serwerów backendowych na zasadzie usług. Koncentruje się na automatyzacji infrastruktury backendowej, umożliwiając programistom skupienie się głównie na aspektach frontendowych aplikacji. *BaaS* dostarcza gotowe usługi i rozwiązania, takie jak zarządzanie danymi, powiadomienia push czy autentykację użytkowników [7]. Dzięki temu nie było konieczności implementacji interfejsu *REST API*, ani opracowywania własnego serwera.

4.1.1 Warstwy aplikacji

Aplikacja składa się z trzech głównych warstw: prezentacji, logiki oraz danych. Warstwa aplikacji odpowiada za interfejs użytkownika i interakcję, jej przykładowe elementy to *MainActivity*, *ParcelActivity* (ekrany aplikacji), *FragmentMap* (fragment interfejsu wyświetlany wewnątrz ekranu) oraz *Layouty XML* (pliki definiujące układ elementów interfejsu). W warstwie logiki aplikacji przetwarzane są dane, obsługiwane operacje i następuje komunikacja z *Firestore*. Są to na przykład metody obsługujące dodawanie paczek, filtrowanie czy rezerwacje. Z kolei warstwa danych zarządza zapisem i odczytem danych z *Firestore* i *SharedPreferences* - jej elementy to *FirebaseFirestore*, *FirebaseAuth* oraz modele danych (*Parcel, User*).

4.1.2 Diagram architektury i komunikacja z zewnętrznymi usługami

Do komunikacji z zewnętrznymi usługami użyto oficjalnych bibliotek SDK. Dostęp do bazy danych, systemu logowania oraz powiadomień (*Firestore*, *Authentication* i *Cloud Messaging*) jest zapewniony przez *Firebase*. Wykorzystano także usługi *Google Maps API* do wyświetlania lokalizacji paczek oraz *Google Places API* do wyszukiwania miejsc i adresów.



Rys. 4.1 Architektura systemu [Opracowanie własne]

4.2 Zakres funkcji

System składa się z dwóch podstawowych aktorów:

- Użytkownik niezalogowany,
- Użytkownik zalogowany.

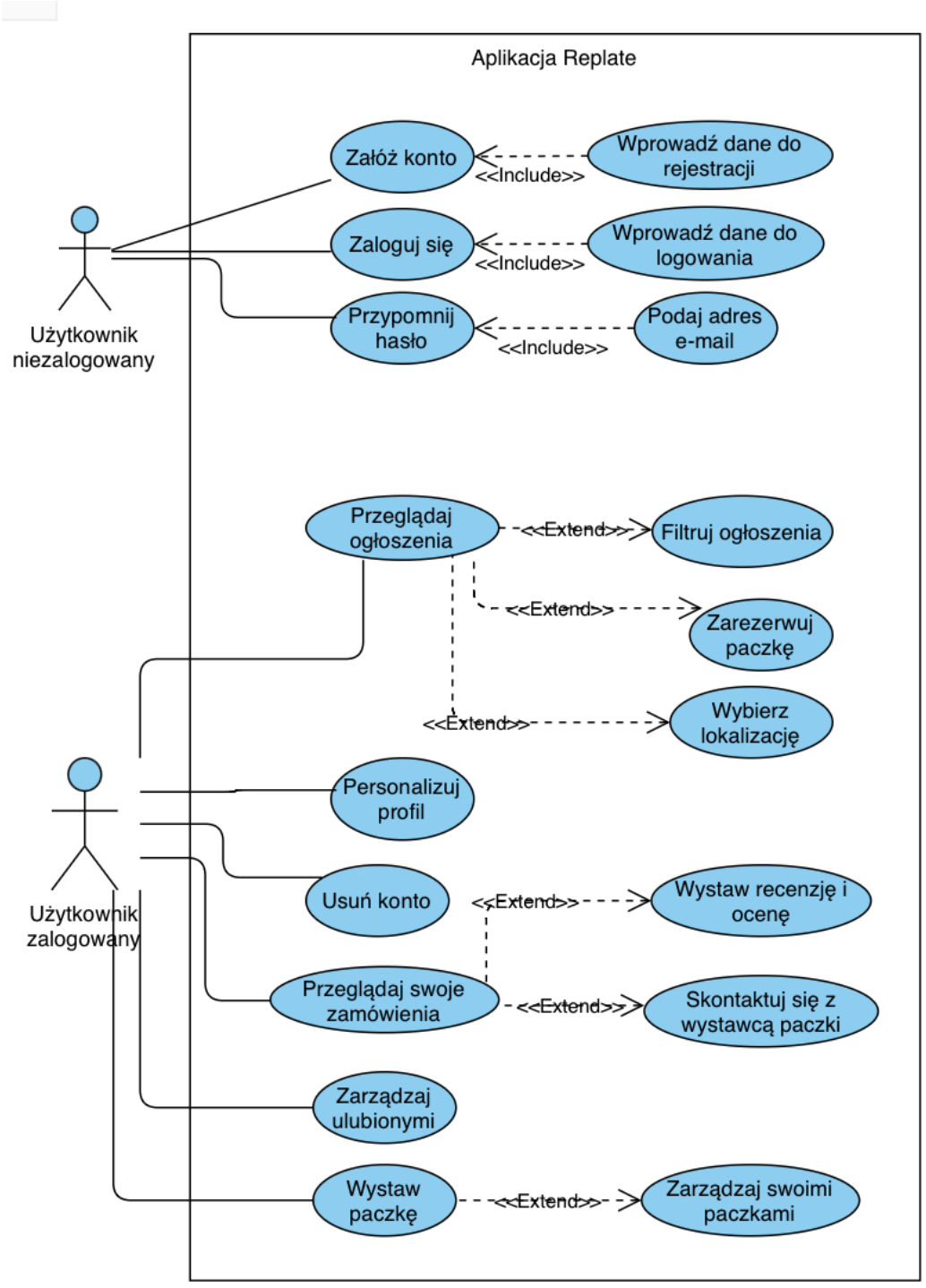
Użytkownik niezalogowany nie ma aktywnej sesji w systemie i jego dostęp do funkcji aplikacji jest ograniczony do procesów uwierzytelniania. Po uruchomieniu widzi ekrany wprowadzające, a następnie może wybrać logowanie lub rejestrację. Może to zrobić różnymi metodami:

- za pomocą konta Google,
- przy użyciu numeru telefonu, na który wysyłany jest kod weryfikacyjny,
- poprzez adres e-mail i hasło.

Użytkownik zalogowany ma pełny dostęp do aplikacji. Może pełnić funkcję zarówno jako odbiorca, jak i nadawca paczek. Jego funkcje obejmują:

- **Przeglądanie ogłoszeń** - użytkownik może przeglądać aktywne paczki filtrując je zgodnie ze swoimi preferencjami oraz wybierając lokalizację.
- **Zarządzanie ulubionymi użytkownikami** - może dodawać innych użytkowników do ulubionych, dzięki czemu będzie otrzymywać powiadomienia, kiedy zostaną przez nich dodane nowe paczki.
- **Personalizacja profilu** - użytkownik może dodać zdjęcie profilowe, ustawić domyślne informacje (numer telefonu, nazwa, adres) oraz włączyć lub wyłączyć opcję powiadomień.
- **Wystawianie ogłoszeń** - użytkownik może wystawić paczkę, decydując gdzie i kiedy może zostać odebrana, oraz za jaką kwotę.
- **Usuwanie konta** - użytkownik może usunąć swoje konto z bazy.

W aplikacji nie ma osobnego panelu administratora jako użytkownika końcowego. Występuje on jako rola techniczna. Ma dostęp do danych przechowywanych w bazie poprzez środowisko *Firebase*, gdzie może modyfikować i usuwać ogłoszenia, oraz monitorować aktywność w systemie. Rysunek 4.2 przedstawia diagram przypadków użycia ilustrujący funkcjonalności dostępne w systemie.



Rys. 4.2 Diagram przypadków użycia [Opracowanie własne]

4.3 Charakterystyka bazy danych

Żeby aplikacja prawidłowo funkcjonowała, wszyscy użytkownicy muszą widzieć to samo, a także konieczne jest zapisywanie dużej ilości informacji. W projekcie została użyta baza danych *Firestore*, w której dane są przechowywane jako kolekcje i dokumenty. Jest to baza nierelacyjna, dlatego poszczególne dokumenty mogą zawierać różne zestawy pól.

Pierwszą kolekcją w bazie danych jest kolekcja *parcels*, w której każdy dokument reprezentuje pojedynczą ofertę przekazania paczki (ogłoszenie dodane przez użytkownika). Dokument może zawierać następujące pola:

- **address** - miejsce odbioru paczki.
- **title** - tytuł paczki.
- **date** - data, do której paczka powinna zostać odebrana.
- **description** - opis zawartości paczki.
- **startTime** - godzina rozpoczęcia przedziału czasowego, w którym paczka może zostać odebrana.
- **endTime** - godzina zakończenia przedziału czasowego odbioru paczki.
- **finished** - etykieta wskazująca, czy proces oddania paczki został zakończony.
- **imageUrl** - adres URL obrazka powiązanego z konkretnym ogłoszeniem, przechowywanego w Firebase Storage.
- **lat** - szerokość geograficzna miejsca odbioru paczki, wykorzystywana do obliczenia odległości od użytkownika.
- **lng** - długość geograficzna miejsca odbioru paczki, wykorzystywana do obliczenia odległości od użytkownika.
- **phone** - numer telefonu kontaktowego do wystawcy paczki.
- **placeId** - unikalny identyfikator miejsca w *Google Places API*, który pozwala dokładnie wskazać lokalizację na mapie i korzystać z usług *Google Maps*.
- **status** - etykieta wskazująca dostępność paczki: *active* oznacza, że paczka jest dostępna, *inactive* - że została zarezerwowana.
- **tags** - tablica tagów opisujących zawartość paczki i umożliwiająca jej filtrowanie.
- **userId** - identyfikator użytkownika wystawiającego paczkę.

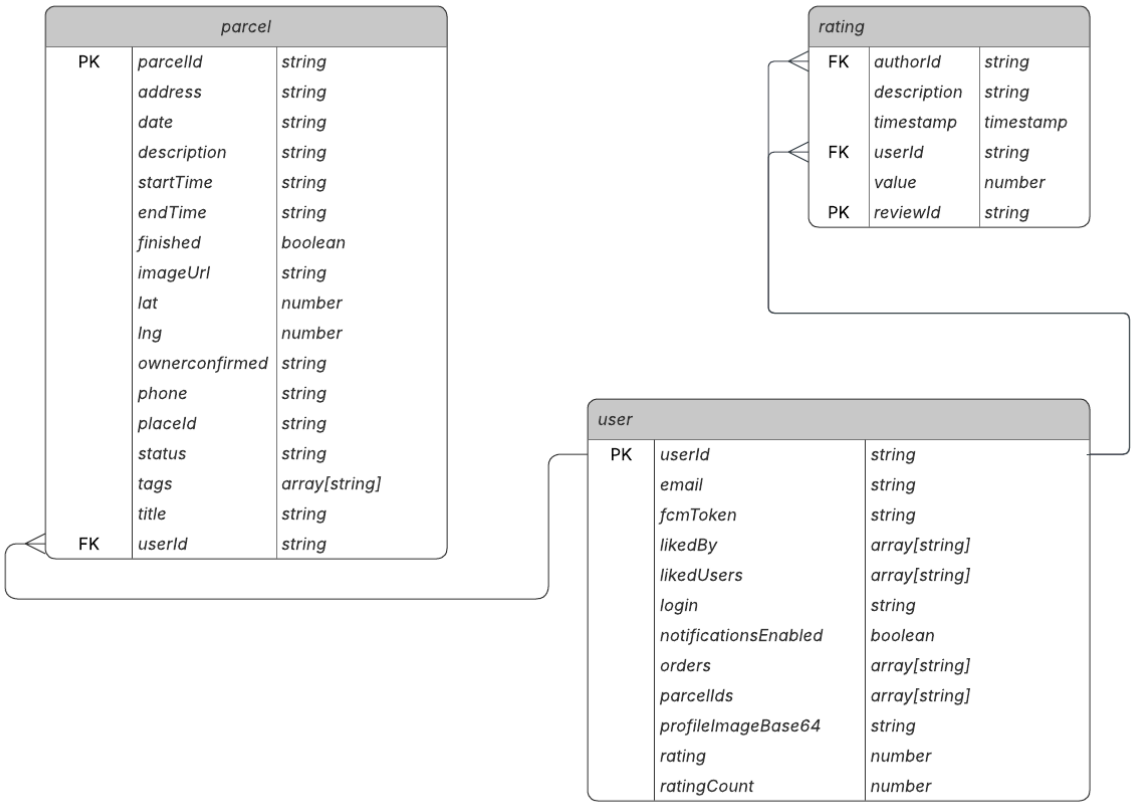
Kolekcja *users* przechowuje dane użytkowników aplikacji *Replate*. Pojedynczy dokument zawiera zwykle::

- **address** - opcjonalny adres użytkownika, który jest automatycznie dodawany do paczki przy jej wystawianiu.
- **email** - pole przechowujące adres e-mail użytkownika; występuje, jeżeli użytkownik loguje się poprzez adres e-mail i hasło lub konto *Google*.
- **fcmToken** - unikalny token urządzenia użytkownika, wygenerowany przez *Firebase Cloud Messaging,* służący do wysyłania powiadomień push.
- **likedBy** - tablica identyfikatorów użytkowników, którzy dodali danego użytkownika do ulubionych.
- **likedUsers** - tablica identyfikatorów użytkowników dodanych do ulubionych przez danego użytkownika.
- **login** - nazwa użytkownika wyświetlana w jego profilu.
- **notificationsEnabled** - etykieta wskazująca zgodę użytkownika na otrzymywanie powiadomień.
- **orders** - tablica identyfikatorów paczek zarezerwowanych przez danego użytkownika.
- **parcelIds** - tablica identyfikatorów paczek dodanych przez danego użytkownika.
- **phone** - opcjonalny numer telefonu, który jest automatycznie dodawany do paczki przy jej wystawianiu; w przypadku logowania przez telefon jest dodawany automatycznie.
- **rating** - ocena danego użytkownika, obliczana jako średnia wszystkich wystawionych ocen.
- **ratingCount** - liczba ocen wystawionych użytkownikowi, wykorzystywana do obliczenia wartości średniej w polu rating.
- **profileImageBase64** - zakodowany w formacie Base64 ciąg znaków reprezentujący zdjęcie profilowe użytkownika.
- **stripeAccountId** - identyfikator konta *Stripe* użytkownika, obecny tylko wtedy, gdy użytkownik chce umożliwić odbieranie płatności online za swoje paczki.

Ostatnią z kolekcji w bazie *Firestore* jest kolekcja *ratings*, która przechowuje wystawione recenzje użytkowników. Dokument w tej kolekcji posiada następujące pola:

- **authorId** - identyfikator użytkownika, który wystawił recenzję.
- **description** - treść recenzji.
- **timestamp** - data i czas dodania recenzji.
- **userId** - identyfikator użytkownika, który został oceniony.
- **value** - ocena wystawiona użytkownikowi, wyrażona w skali od 1 do 5.

Dane użytkowników, paczek i ocen są przechowywane w chmurze *Firebase Firestore*, która jest nierelacyjną bazą danych. Zamiast organizacji w klasyczne tabele, dane są przechowywane w kolekcjach i dokumentach. Poniższy diagram ERD pokazuje w formie przybliżonej struktury danych i powiązania logiczne.



Rys. 4.3. Diagram ERD [opracowanie własne]

4.4 Wybrane elementy implementacyjne

4.4.1 Logowanie użytkownika za pomocą numeru telefonu

By korzystać z aplikacji użytkownik musi się zalogować. Jednym ze sposobów jest użycie numeru telefonu dzięki integracji z usługą *Firebase Authentication*. Umożliwia to szybkie logowanie bez konieczności zapamiętywania hasła. Poniższy fragment przedstawia odpowiedzialną za to funkcję *startPhoneLogin()*:

Listing 4.1. Logowanie z użyciem numeru telefonu [Opracowanie własne]

```
private void startPhoneLogin() {
     AlertDialog dialog = new AlertDialog.Builder(this)
                 .setTitle("Login with your phone number")
                 .setView(LayoutInflater.from(this).inflate(R.layout.dialog_phone_input, null))
                 .setPositiveButton("OK", null)
                 .create();

     dialog.setOnShowListener(d -> {
          dialog.getButton(AlertDialog.BUTTON_POSITIVE).setOnClickListener(v -> {
                 String phone = ((EditText)
dialog.findViewById(R.id.inputEditText)).getText().toString().trim();
                 if (!phone.startsWith("+")) return;

                 PhoneAuthOptions options = phoneAuthOptions.newBuilder(FirebaseAuth.getInstance())
                             .setPhoneNumber(phone)
                             .setTimeout(68L, TimeUnit.SECONDS)
                             .setActivity(this)
                             .setCallbacks(new
PhoneAuthProvider.OnVerificationStateChangedCallbacks() {
                                  @Override
                                  public void onVerificationCompleted(@NonNull PhoneAuthCredential
c) {
                                        FirebaseAuth.getInstance().signInWithCredential(c)
                                                  .addOnCompleteListener(task -> {
                                                        if  (task.isSuccessful()) {
                                                            startActivity(new
Intent(LoginActivity.this, HomeActivity.class));
                                                            finish(); }
                                                  }); }
                                  @Override
                                  public void onVerificationFailed(@NonNull FirebaseException e)
{}
                             }).build();

                 PhoneAuthProvider.verifyPhoneNumber(options);
                 dialog.dismiss();
            });
        });
        dialog.show(); }
```

Po wywołaniu funkcji otwierane jest okno dialogowe z polem do wprowadzenia numeru telefonu. Aplikacja sprawdza jego poprawność - czy zaczyna się od „+” i zawiera kod kraju. *Firebase* wysyła sms z kodem weryfikacyjnym, a w momencie pomyślnej weryfikacji aplikacja loguje użytkownika używając metody *signInWithCredential()*. Następuje przekierowanie do ekranu głównego.

4.4.2 Wybór i aktualizacja zdjęcia profilowego użytkownika

Użytkownik ma możliwość dostosowania swojego profilu poprzez dodanie zdjęcia profilowego. Jest ono wyświetlane w ogłoszeniu paczki, oraz w liście dodanych do ulubionych.

Listing 4.2. Wybór i aktualizacja zdjęcia profilowego użytkownika [Opracowanie własne]

```
profileImage.setOnClickListener(v -> {
    String[] options = {"Choose from gallery", "Take a photo"};
    new AlertDialog.Builder(getContext())
        .setTitle("Change profile picture")
        .setItems(options, (dialog, which) -> {
            if (which == 0) {
                pickImageFromGallery();
            } else {
                takePhotoWithCamera();
            }
        }).show();
});

private void pickImageFromGallery() {
    Intent intent = new Intent(Intent.ACTION_PICK);
    intent.setType("image/*");
    startActivityForResult(intent, REQUEST_IMAGE_PICK);
}

private void takePhotoWithCamera() {
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    if (intent.resolveActivity(requireActivity().getPackageManager()) != null) {
        startActivityForResult(intent, REQUEST_IMAGE_CAPTURE);
    } else {
        Toast.makeText(getContext(),
            "No camera app found", Toast.LENGTH_SHORT).show();
    }
}
```

Po kliknięciu na aktualne zdjęcie użytkownik otrzymuje możliwość wyboru jednej z dwóch opcji: wybrania zdjęcia z galerii lub zrobienia nowego zdjęcia aparatem.
W zależności od wyboru, aplikacja uruchamia odpowiedni intent systemowy:

- **Intent.ACTION_PICK** umożliwia wybór pliku graficznego z pamięci urządzenia,
- **MediaStore.ACTION_IMAGE_CAPTURE** otwiera aplikację aparatu w celu wykonania nowego zdjęcia.

Po zakończeniu akcji wynik jest przekazywany do metody *onActivityResult*, gdzie zdjęcie może zostać zapisane w bazie *Firebase Storage* oraz powiązane z kontem użytkownika w *Firestore*.

4.4.3 Pobieranie aktywnych paczek i filtrowanie

W fragmencie *FragmentMap* użytkownik może przeglądać dostępne paczki, które są wyświetlane na mapie jako znaczniki zawierające miniatury zdjęć dodanych do paczek. Wyświetlane paczki są filtrowane zgodnie z preferencjami użytkownika, np. po dacie, tagach.

Metoda *loadParcelsAndShowMarkers* przedstawiona na listingu 4.3 pobiera aktywne paczki z kolekcji *Firestore* i filtruje je po kilku kryteriach: użytkownik, data, tagi i przedział czasowy. Pomyślnie przefiltrowane paczki są następnie wyświetlane na mapie Google przy użyciu metody *fetchPlaceAndAddMarker*. Dzięki temu użytkownik widzi tylko te paczki, które są dla niego dostępne w wybranym dniu i pasują do jego preferencji.

Listing 4.3. Pobieranie aktywnych paczek i filtrowanie ich [Opracowanie własne]

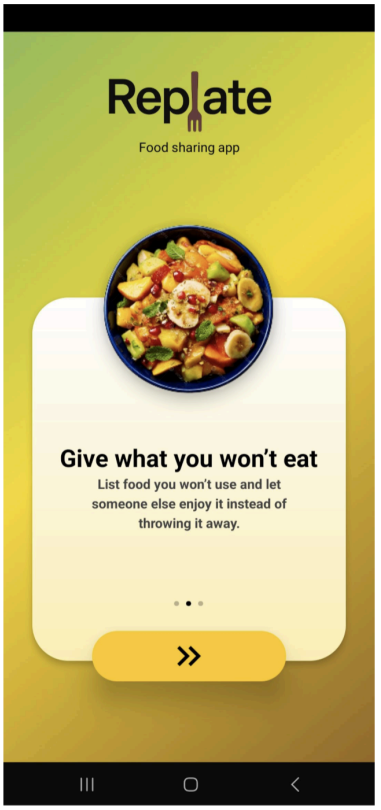
```
private void loadParcelsAndShowMarkers() {
    String currentUserId = FirebaseAuth.getInstance().getCurrentUser().getUid();
    SharedPreferences prefs = requireContext().getSharedPreferences("MyPrefs", Context.MODE_PRIVATE);
    String selectedDate = prefs.getString("selectedDate", getTodayDate());

    db.collection("parcels")
        .whereEqualTo("status", "active")
        .get()
        .addOnSuccessListener(querySnapshot -> {
            for (QueryDocumentSnapshot doc : querySnapshot) {
                String parcelId = doc.getId();
                String creatorId = doc.getString("userId");

                if (creatorId.equals(currentUserId)) continue;

                String parcelDate = doc.getString("date");
                if (!selectedDate.equals(parcelDate)) continue;
                List<String> tags = (List<String>) doc.get("tags");
                if (!tagsMatchFilter(tags)) continue;

                String placeId = doc.getString("placeId");
                String title = doc.getString("title");
                if (placeId != null) fetchPlaceAndAddMarker(placeId, title, parcelId);
            }
        })
        .addOnFailureListener(e -> {
            Toast.makeText(requireContext(), "Failed to load parcels", Toast.LENGTH_SHORT).show();
        });
}

private String getTodayDate() {
    return new
java.text.SimpleDateFormat("yyyy-MM-dd").format(java.util.Calendar.getInstance().getTime());
}

private boolean tagsMatchFilter(List<String> tags) {
    if (currentSelectedChips == null || currentSelectedChips.isEmpty())
        return true;

    for (String t : tags) {
        if (currentSelectedChips.contains(t))
            return true;
    }
    return false;
}
```

5. Interfejsy użytkownika

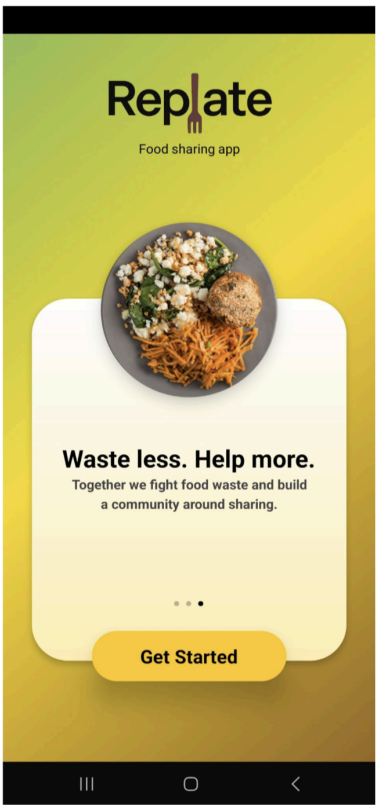
W tym rozdziale omówione zostaną interfejsy użytkownika, pozwalające na korzystanie z kluczowych funkcji systemu. Załączone rysunki wraz z opisami umożliwią lepsze zrozumienie działania aplikacji.

5.1 SplashScreen i ekrany powitalne

Rys. 5.1. SplashScreen [Opracowanie własne]



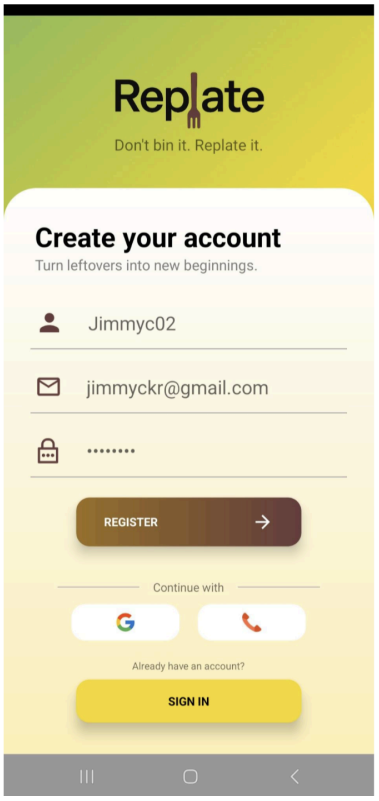
Rys. 5.2. Ekran powitalny cz.1 [Opracowanie własne]



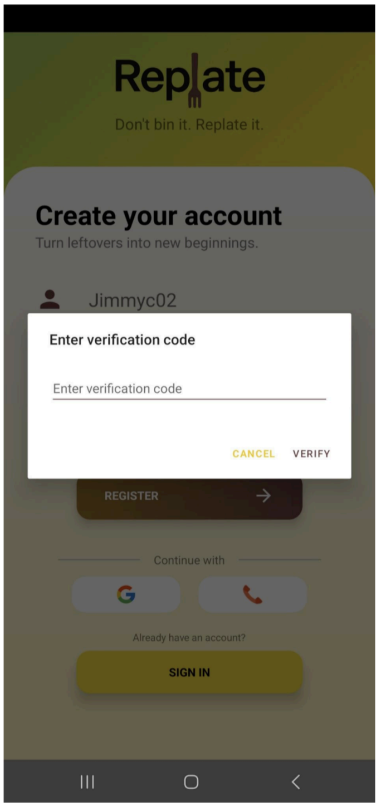
Rys. 5.3. Ekran powitalny cz. 2 [Opracowanie własne]

Po uruchomieniu wyświetlany jest *SplashScreen* (ekran pojawiający się podczas startu programu) z animacją zwiększania i zmniejszania logo. Jeżeli użytkownik otwiera aplikację po raz pierwszy, pokazywane są ekrany powitalne, wyjaśniające jej ideę.

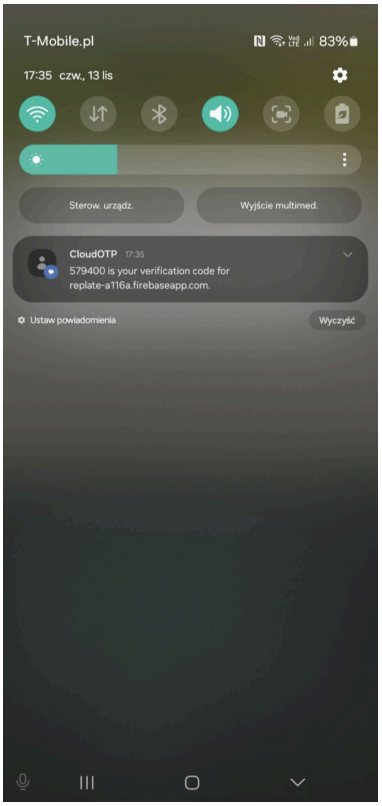
5.2 Rejestracja



Rys. 5.4. Ekran rejestracji [Opracowanie własne]



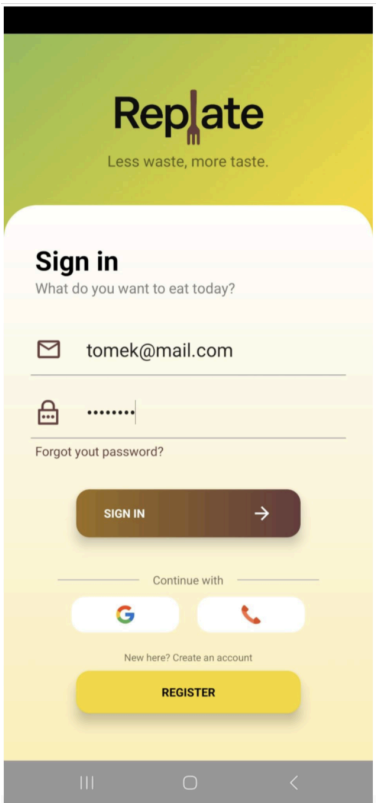
Rys. 5.5. Okno dialogowe do wpisania kodu weryfikacyjnego [Opracowanie własne]



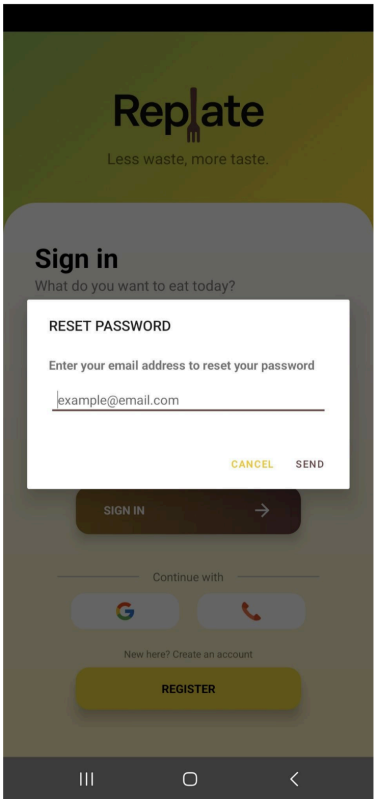
Rys. 5.6. Wiadomość z kodem weryfikacyjnym [Opracowanie własne]

Użytkownik ma możliwość zarejestrowania się w systemie poprzez wpisanie adresu e-mail, nazwy i hasła, łącząc swoje konto Google lub używając numeru telefonu. Na rysunku 5.5 widoczne jest okno do wpisania kodu weryfikacyjnego wysłanego na podany numer.

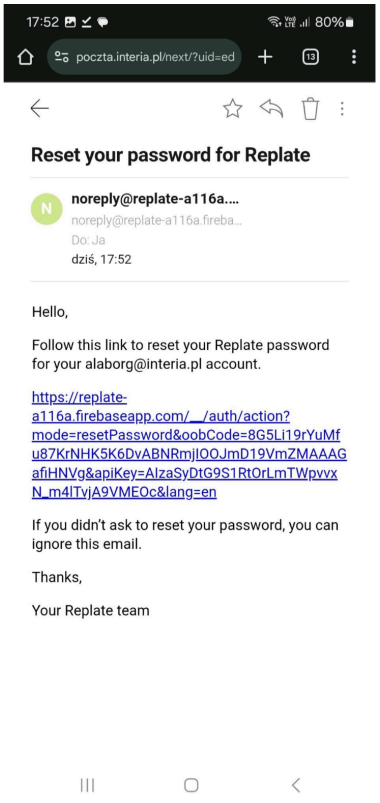
5.3 Logowanie i resetowanie hasła



Rys. 5.7. Ekran logowania [Opracowanie własne]



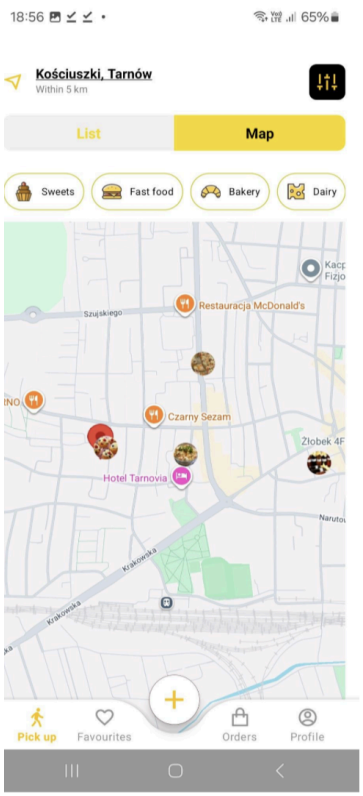
Rys. 5.8. Okno dialogowe do resetowania hasła [Opracowanie własne]



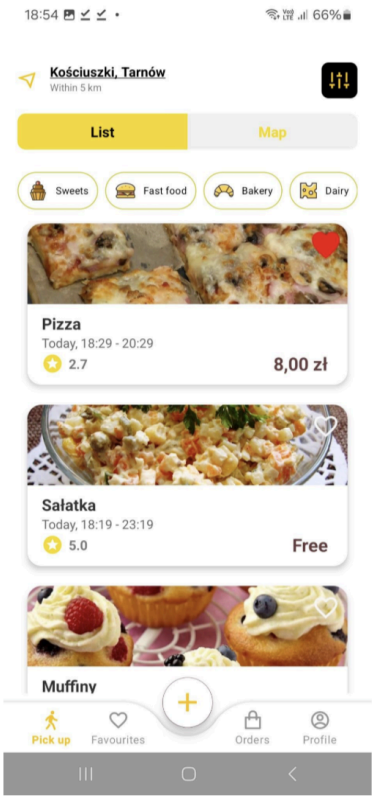
Rys. 5.9. Wiadomość z linkiem do resetowania hasła [Opracowanie własne]

Użytkownik może zalogować się do aplikacji, korzystając z adresu e-mail i hasła lub wybierając logowanie przez konto Google bądź numer telefonu. W przypadku logowania Google lub telefonicznego system najpierw weryfikuje, czy w bazie istnieje już konto powiązane z podanym adresem e-mail lub numerem. Jeśli nie, automatycznie zakładane jest nowe konto użytkownika. Funkcja resetowania hasła pozwala na podanie adresu e-mail. Jeżeli w bazie znajduje się konto przypisane do tego adresu, system wysyła na niego wiadomość z linkiem umożliwiającym ustawienie nowego hasła.

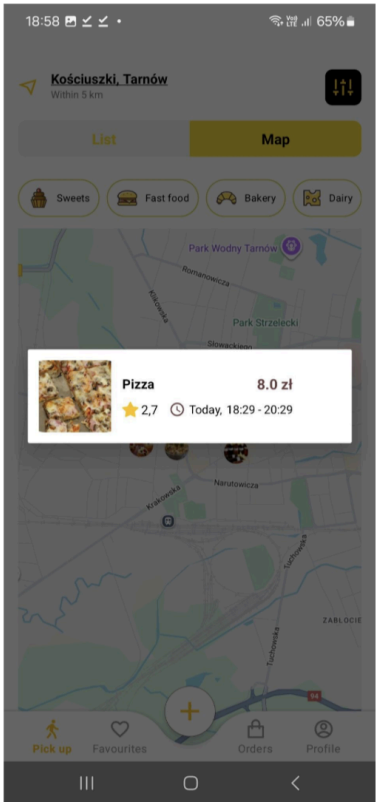
5.4 Wyświetlanie paczek



Rys. 5.10. Wyświetlanie paczek na mapie [Opracowanie własne]



Rys. 5.11. Ogłoszenia w liście [Opracowanie własne]



Rys. 5.12. Okno pokazujące szczegóły paczki [Opracowanie własne]

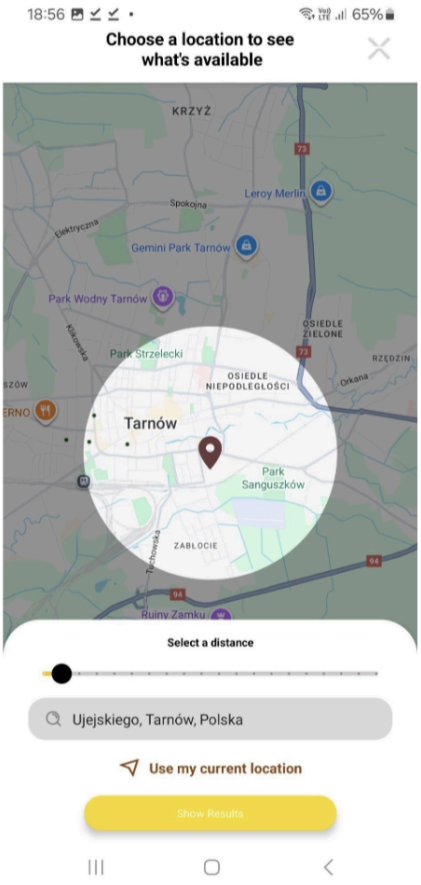
Na rysunku 5.11 widać ekran wyświetlany użytkownikowi po udanym logowaniu. Wyświetlane są w liście aktualnie dostępne ogłoszenia. Z ekranu głównego można wybrać tagi filtrujące wyświetlane paczki. Użytkownik ma opcję kliknięcia ikony serca na elemencie listy, co dodaje właściciela do ulubionych.

Na górze ekranu możliwe jest przełączenie w tryb mapy (rysunek 5.10), gdzie widać aktualne ogłoszenia jako małe kółka zawierające zdjęcia przypisane do danego ogłoszenia. Po kliknięciu na obrazek, wyświetlane jest okno pokazujące najważniejsze informacje o paczce, co widać na rysunku 5.12.

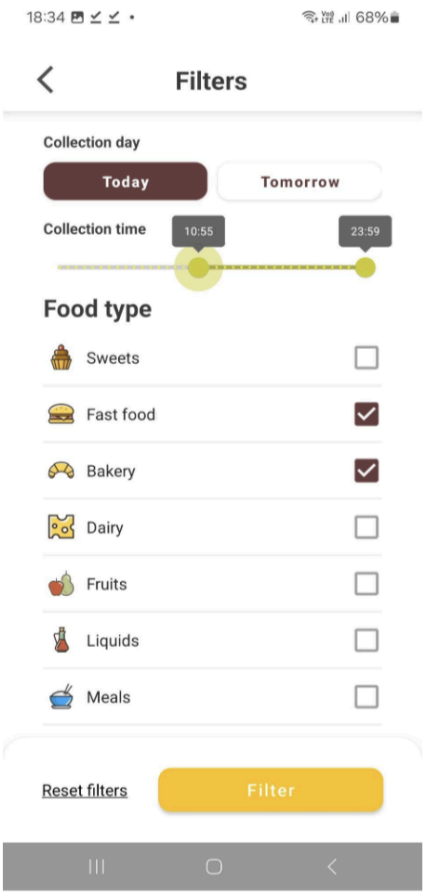
Z poziomu dolnej nawigacji użytkownik może przejść do listy ulubionych użytkowników, do widoku zarezerwowanych paczek oraz do swojego profilu. Przycisk z ikoną „plusa” umożliwia szybkie uruchomienie procesu dodawania nowego ogłoszenia.

5.5 Wybór lokalizacji i filtry

Na rysunku 5.13 przedstawiono widok wyboru lokalizacji, w obrębie której użytkownik chce wyświetlać dostępne paczki. W polu wyszukiwania można wpisać nazwę miejsca, a następnie wybrać jedną z sugerowanych pozycji udostępnianych przez *Google Places API*. Za pomocą suwaka można regulować promień wyszukiwania. Rysunek 5.14 pokazuje ekran filtrów. Możemy wybrać dzień odbioru, przedział czasowy i tagi przypisane do paczek.



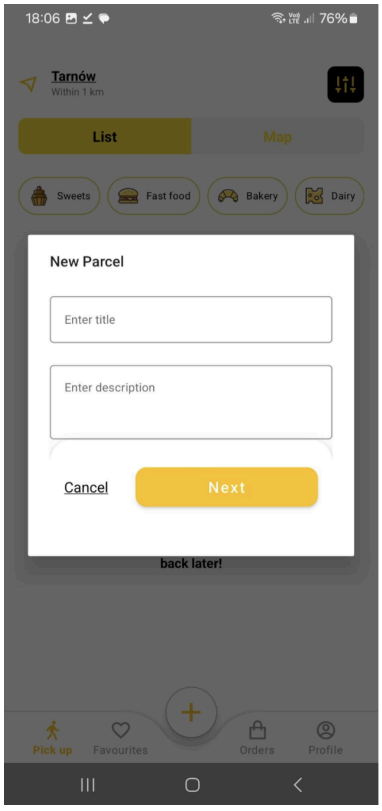
Rys. 5.13. Wybór lokalizacji [Opracowanie własne]



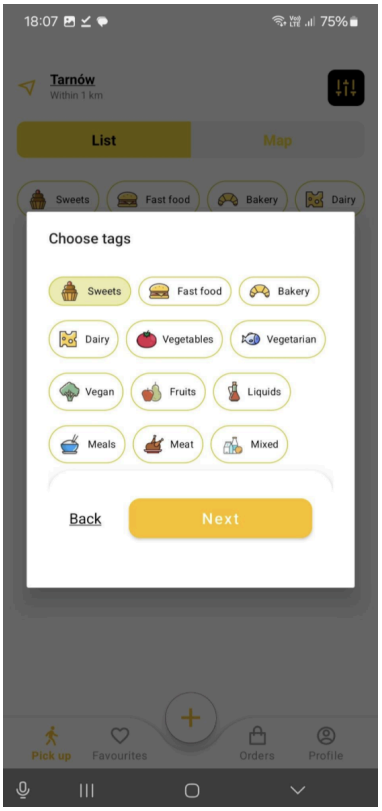
Rys. 5.14. Ekran filtrów [Opracowanie własne]

5.6 Proces dodawania ogłoszenia

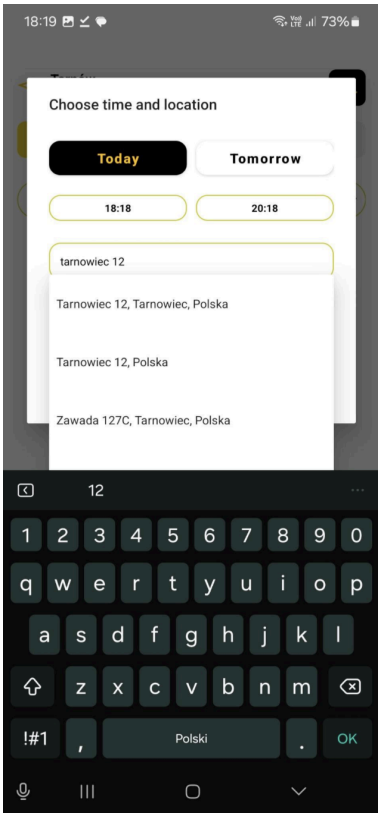
Na rysunkach 5.15, 5.16, 5.17, 5.18, 5.19 i 5.20 przedstawiono kolejne etapy procesu dodawania ogłoszenia. Użytkownik podaje tytuł i opis paczki, następnie wybiera tagi odpowiadające jej zawartości. Kolejnym krokiem jest określenie dnia oraz przedziału czasowego odbioru przy użyciu selektora czasu, a także ustalenie ceny. Należy wprowadzić numer telefonu, aby odbiorca mógł skontaktować się z autorem ogłoszenia. Opcjonalnie można dodać zdjęcie paczki, jeśli użytkownik tego nie zrobi, aplikacja automatycznie ustawi obraz domyślny na podstawie wybranych tagów.



Rys. 5.15. Okno dodawania ogłoszenia I [Opracowanie własne]

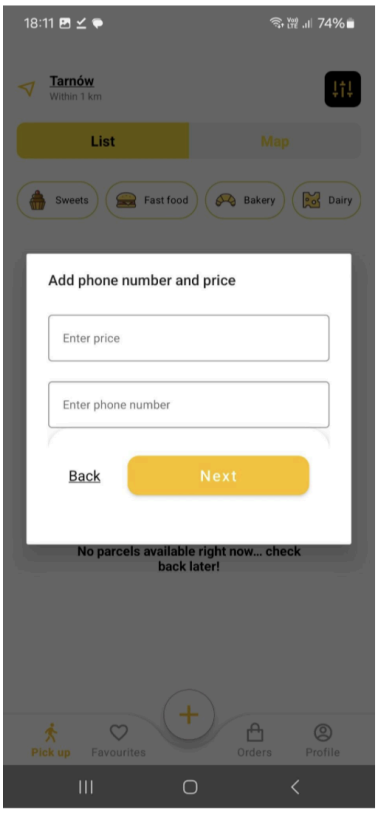


Rys. 5.16. Okno dodawania ogłoszenia II [Opracowanie własne]

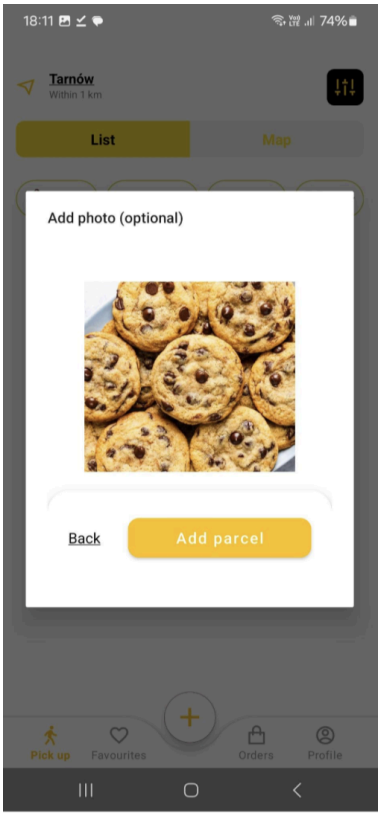


Rys. 5.17. Okno dodawania ogłoszenia III [Opracowanie własne]

Rys. 5.18. Okno dodawania ogłoszenia - TimePicker [Opracowanie własne]

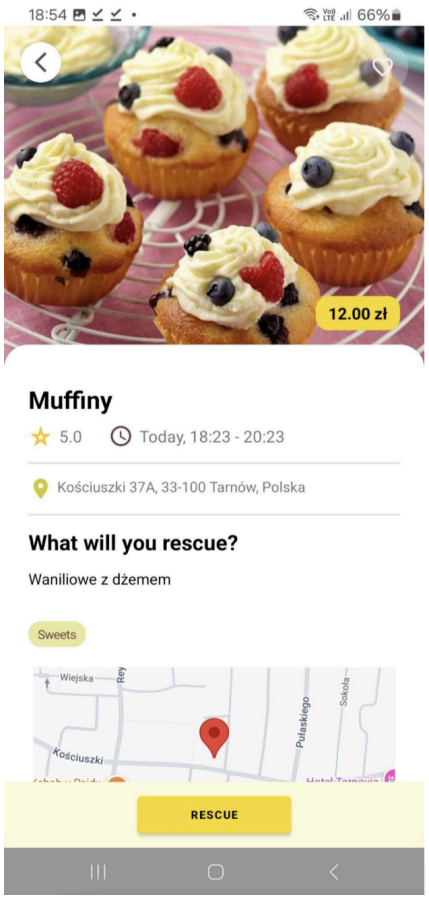


Rys. 5.19. Okno dodawania ogłoszenia IV [Opracowanie własne]

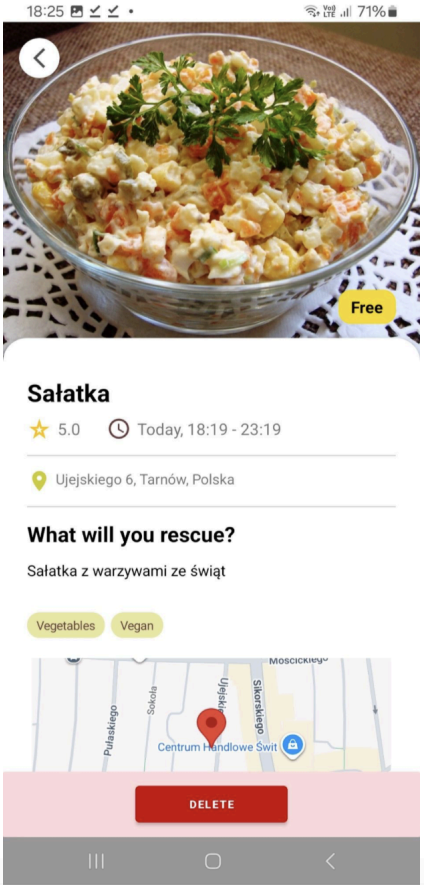


Rys. 5.20. Okno dodawania ogłoszenia V [Opracowanie własne]

5.7 Ekran ogłoszenia



Rys. 5.21. Ekran ogłoszenia I [Opracowanie własne]

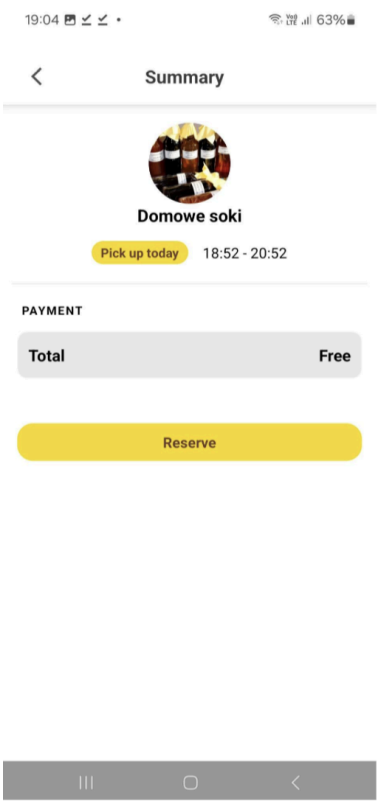


Rys. 5.22. Ekran ogłoszenia II [Opracowanie własne]

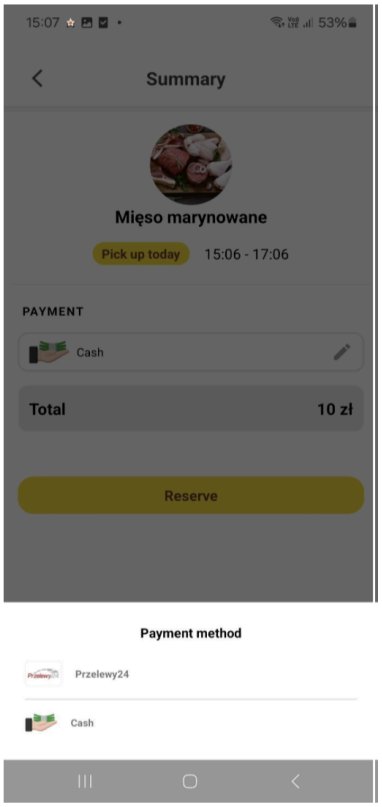
Rysunki 5.21 oraz 5.22 przedstawiają ekran ogłoszenia. Pokazany jest opis, średnia ocena autora, czas odbioru i adres, a także dodane tagi i fragment mapy pokazujący miejsce odbioru. Jeżeli otwarte zostało przez użytkownika niebędącego autorem, przycisk na dole umożliwia zarezerwowanie paczki. Jeżeli widzi ten ekran ogłoszeniodawca, może użyć przycisku do usunięcia paczki.

5.8 Rezerwacja paczki

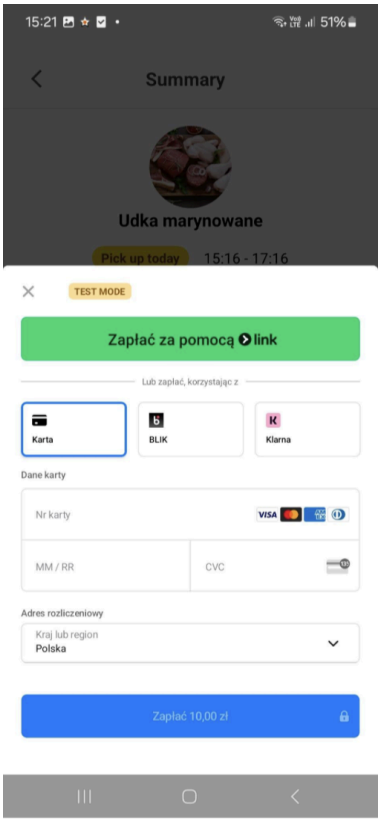
Rysunek 5.23. pokazuje podsumowanie rezerwacji. Jeżeli paczka nie jest darmowa, możemy wybrać sposób płatności (Rysunek 5.24) - gotówką przy odbiorze lub płatność online (Rysunek 5.25), jeżeli autor ma przypisane do swojego profilu konto bankowe.



Rys. 5.23. Podsumowanie rezerwacji [Opracowanie własne]



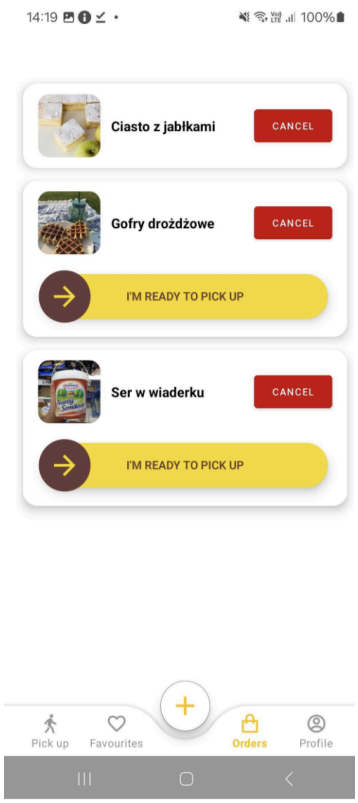
Rys. 5.24. Wybór sposobu płatności [Opracowanie własne]



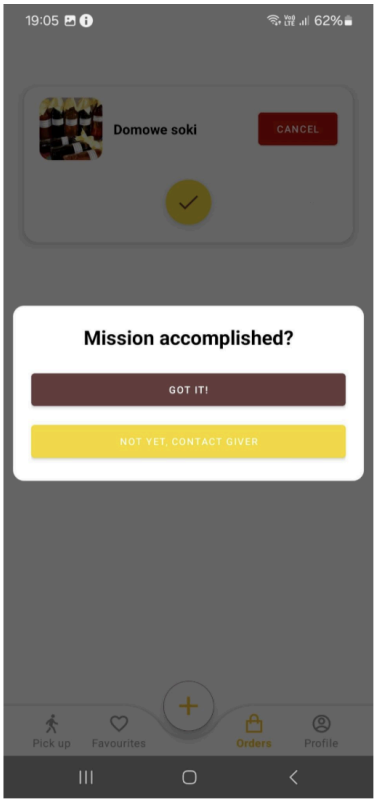
Rys. 5.25. Płatność online [Opracowanie własne]

5.9 Odbiór paczki

Rysunek 5.26 przedstawia okno *Orders*. W tym miejscu wyświetlane są paczki, które użytkownik zarezerwował. Jest możliwość rezygnacji z odbioru klikając przycisk „cancel", co powoduje wysłanie powiadomienia do autora, informujące o ponownej dostępności paczki. Jeżeli aktualna data i godzina mieszczą się w przedziale czasowym wyznaczonym na odbiór paczki, przy danym elemencie listy wyświetlany jest suwak umożliwiający potwierdzenie odbioru. Odbiorca może zaznaczyć odebranie paczki, lub kliknąć przycisk kontaktu z ogłoszeniodawcą, co powoduje wybranie numeru podanego w ogłoszeniu (Rysunek 5.27).



Rys. 5.26. Lista zarezerwowanych paczek [Opracowanie własne]

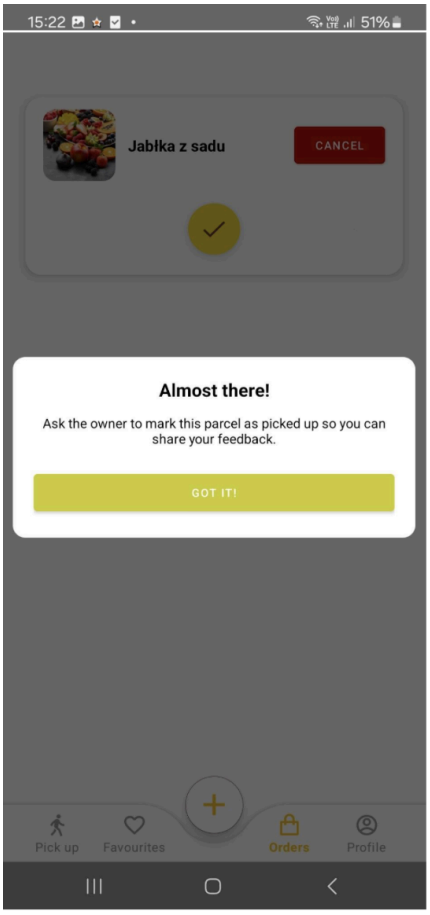


Rys. 5.27. Okno potwierdzające odbiór paczki [Opracowanie własne]

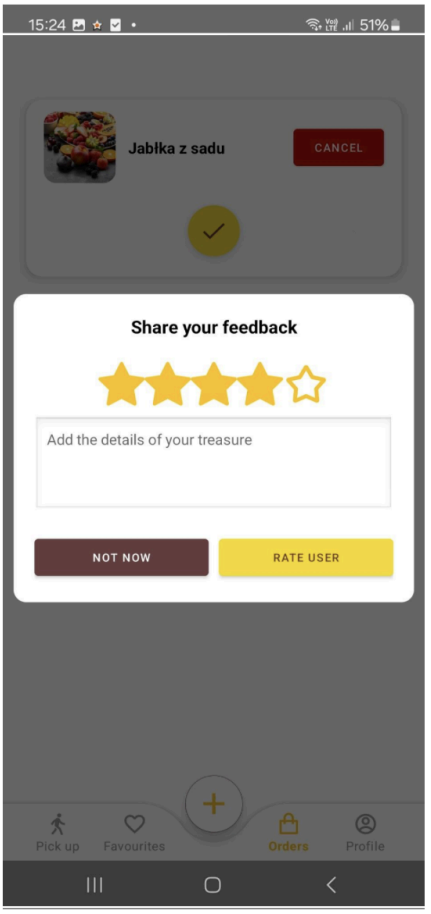
Rys. 5.28. Brak zarezerwowanych paczek [Opracowanie własne]

5.10 Wystawianie opinii

Każdy dokument *parcel* w bazie *Firestore* posiada pole *ownerconfirmed*, domyślnie ustawione na „no”. Dopiero gdy ogłoszeniodawca potwierdzi odbiór, możliwe jest wystawienie opinii. W przeciwnym razie, po potwierdzeniu odbioru przez odbiorcę, wyświetlane jest okno dialogowe wyjaśniające, że recenzję można dodać dopiero gdy odbiór zostanie zatwierdzony (Rysunek 5.29). Na rysunku 5.30 widać okno dodawania opinii - wybieramy ocenę, można także napisać recenzję.



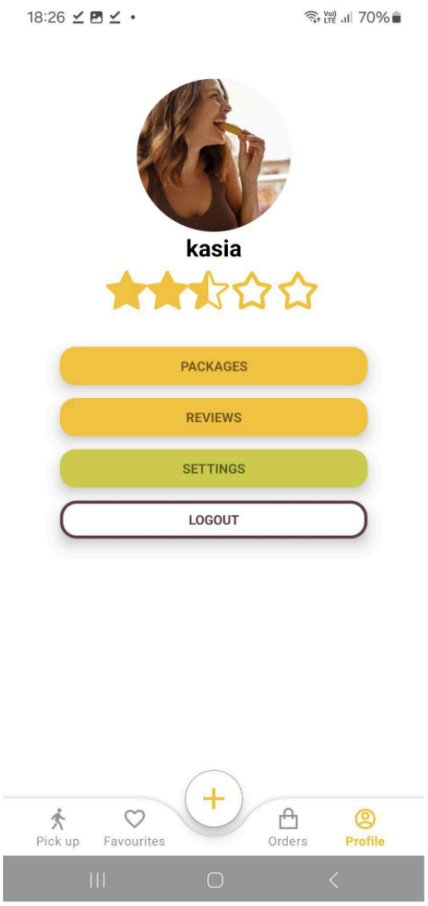
Rys. 5.29. Okno przypominające o potwierdzeniu odbioru [Opracowanie własne]



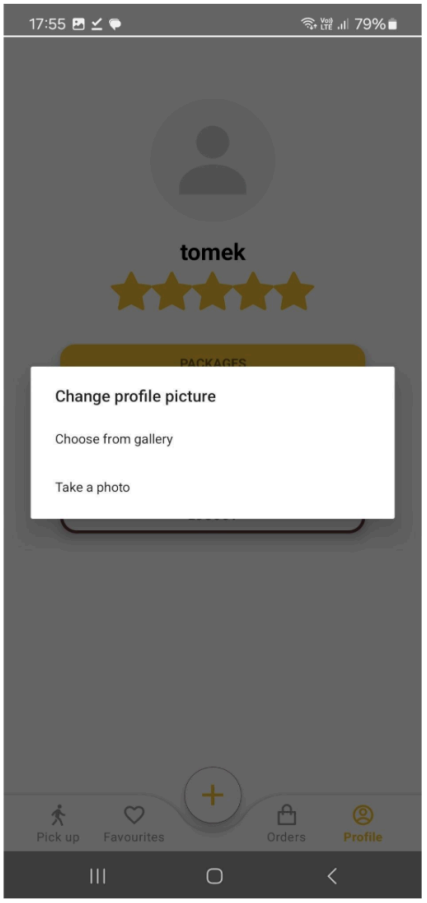
Rys. 5.30. Okno wystawiania recenzji [Opracowanie własne]

5.11 Profil użytkownika

Rysunek 5.31 pokazuje profil użytkownika. Możemy dodać zdjęcie wybierając z galerii, lub otwierając aparat (Rysunek 5.32). Ze swojego profilu jest możliwość przejścia do wystawionych przez siebie paczek, otrzymanych opinii, ustawień lub wylogowania się.



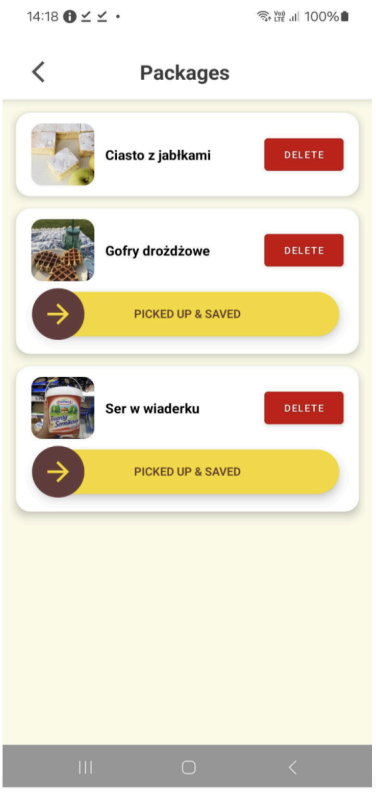
Rys. 5.31. Profil użytkownika



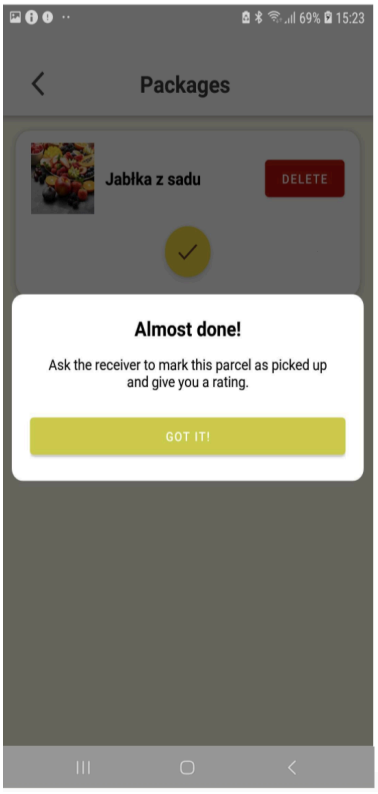
Rys. 5.32. Zmiana zdjęcia profilowego [Opracowanie własne]

5.12 Ekrany ogłoszeń i opinii

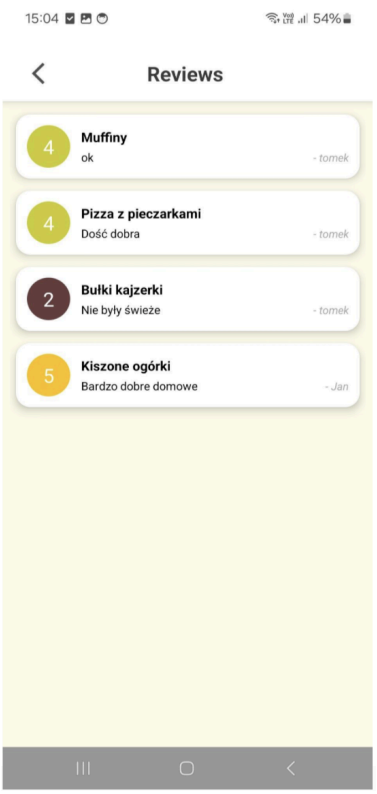
Rysunek 5.33 przedstawia ekran prezentujący paczki dodane przez zalogowanego użytkownika. Jeżeli dana paczka została zarezerwowana, użytkownik może oznaczyć ją jako odebraną, co umożliwia odbiorcy wystawienie opinii. Na rysunku 5.35 zaprezentowano przykłady już wystawionych opinii.



Rys. 5.33. Okno dodanych ogłoszeń [Opracowanie własne]



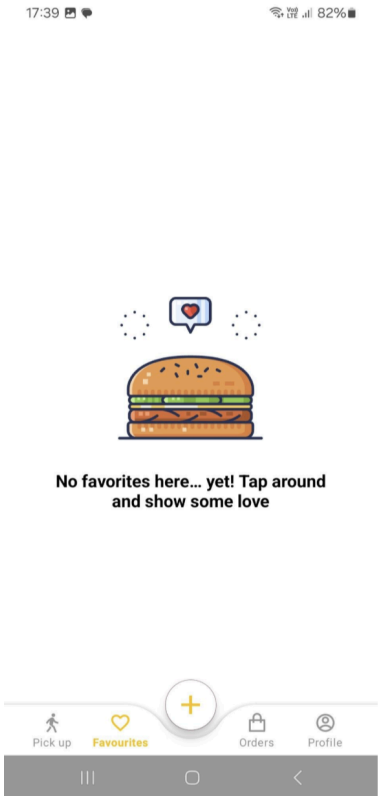
Rys. 5.34. Okno potwierdzenia odbioru [Opracowanie własne]



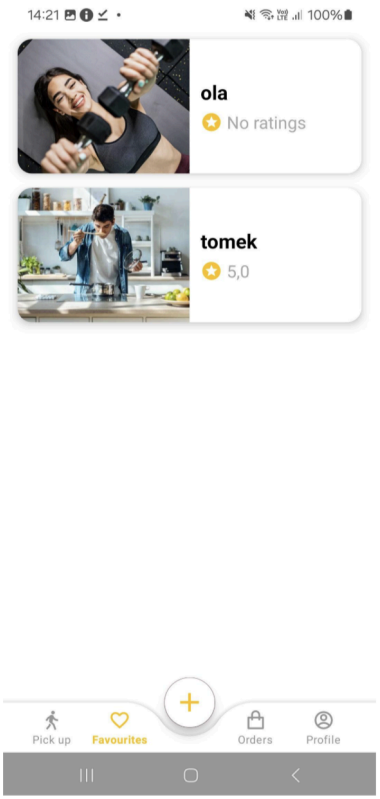
Rys. 5.35. Okno recenzji [Opracowanie własne]

5.13 Ekran ulubionych użytkowników

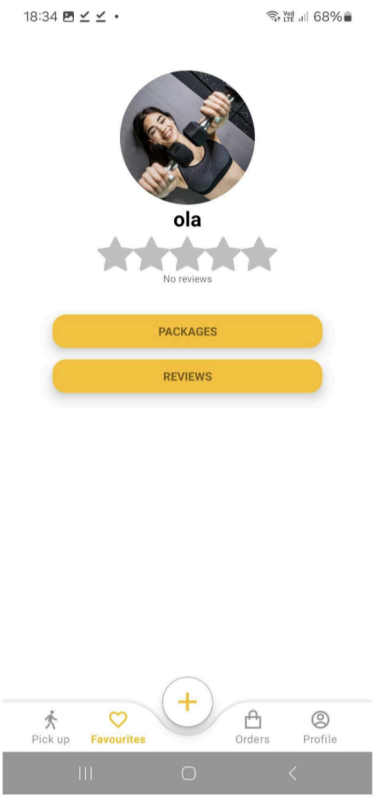
Po wybraniu w dolnym menu zakładki *Favourites* wyświetlani są dodani do ulubionych ogłoszeniodawcy (Rysunki 5.36 i 5.37). Można usunąć z listy pole poprzez przeciągnięcie go (*swipe*) w lewo. Kliknięcie jednego z użytkowników przenosi nas do podglądu jego profilu, skąd można przejrzeć dodane ogłoszenia i otrzymane recenzje (Rysunek 5.38). W przypadku, gdy lista ulubionych użytkowników jest pusta, aplikacja wyświetla animację informującą o braku elementów. Animacja została zaimplementowana z wykorzystaniem biblioteki *Lottie*.



Rys. 5.36. Brak ulubionych użytkowników [Opracowanie własne]



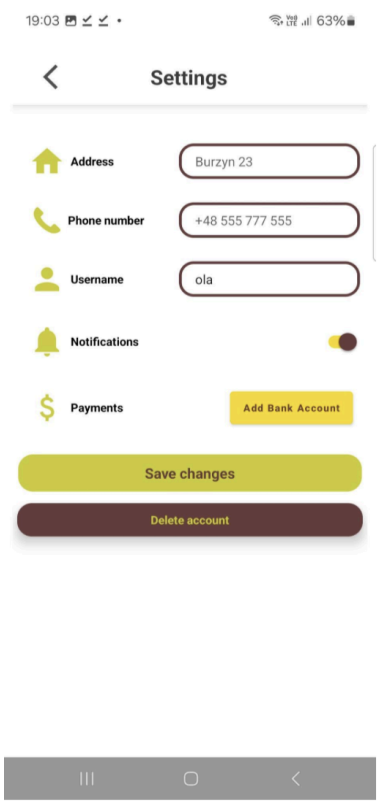
Rys. 5.37. Ekran ulubionych użytkowników [Opracowanie własne]



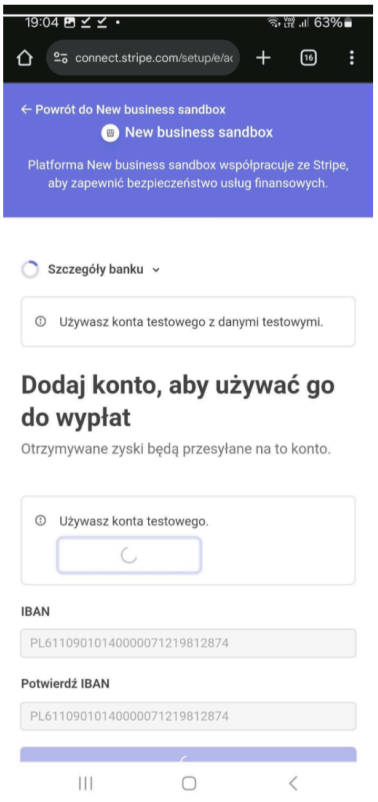
Rys. 5.38. Profil innego użytkownika [Opracowanie własne]

5.14 Ekran ustawień

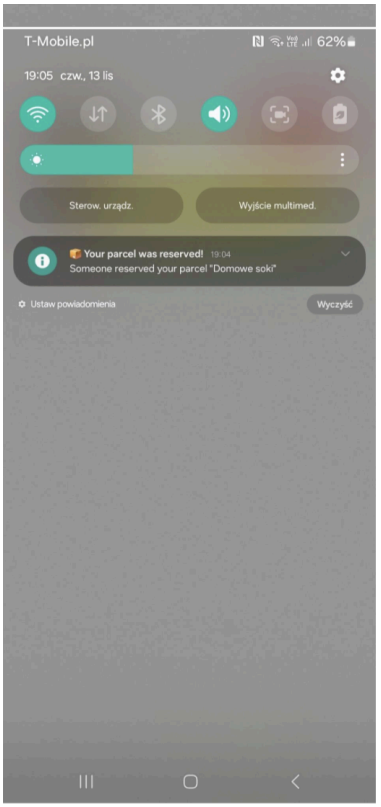
Rysunek 5.39 prezentuje ekran ustawień profilu. Podanie adresu i numeru telefonu przyspiesza proces dodawania ogłoszenia, uzupełniając te pola automatycznie. Użytkownik może włączyć opcję powiadomień, przykładowe powiadomienie pokazano na rysunku 5.41. W celu umożliwienia odbiorcom paczek płatności online, musimy dodać do profilu konto bankowe (Rysunek 5.40). Przycisk na dole służy do usuwania konta z aplikacji.



Rys. 5.39. Ustawienia profilu [Opracowanie własne]



Rys. 5.40. Dodawanie konta do wypłat [Opracowanie własne]



Rys. 5.41. Otrzymane powiadomienie [Opracowanie własne]

6. Testy

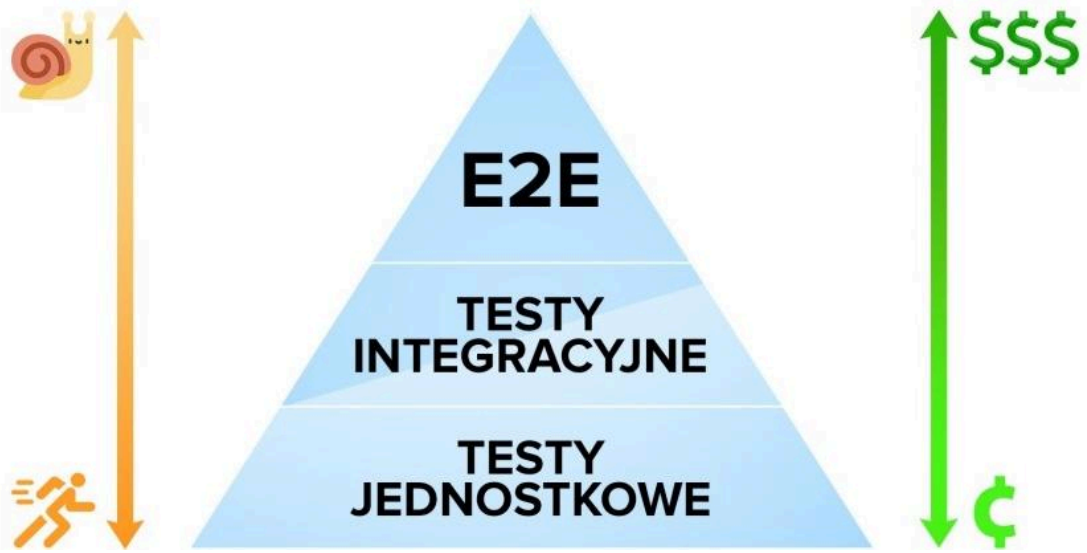
Jednym z ważnych etapów rozwoju aplikacji jest proces testowania oprogramowania, który sprawdza czy wymagania są spełnione, a także zapewnia gwarancję wysokiej jakości ostatecznego produktu. Przeprowadzanie testów w miarę rozwoju aplikacji pozwala szybko wykryć potencjalne błędy i zmniejszyć koszty ich naprawy. Należy pamiętać, że nie jest to jednorazowe działanie, ale proces trwający równolegle do rozwoju aplikacji.

Testy można podzielić na manualne i automatyczne. Pierwsze z nich są wykonywane przez testera, czyli osobę odpowiedzialną za sprawdzenie i zachowanie poprawnego działania systemu. Wciela się on w potencjalnego użytkownika aplikacji, a swoją pracę rozpoczyna od przygotowania wstępnego środowiska do testów. Następnie weryfikuje działanie aplikacji, notując wszelkie problemy, które będą musiały zostać naprawione przez programistów.

Testowanie automatyczne ma z kolei zweryfikować rzeczywiste działanie aplikacji z oczekiwaniami. Te testy są bardziej niezawodne, reużywalne i szybsze, ponieważ nie są spowalniane ludzkimi ograniczeniami. Wykorzystuje się do nich specjalne narzędzia oraz przygotowane skrypty, a ich celem jest określenie czy zaimplementowane funkcje i algorytmy działają zgodnie z przeznaczeniem.

Testy można podzielić także ze względu na poziomy: jednostkowe, integracyjne oraz E2E. Pierwsze z nich mają weryfikować poprawność działania poszczególnych, najmniejszych części, jak na przykład klasy czy metody. Testy integracyjne mają sprawdzać czy serwisy lub systemy poprawnie się ze sobą komunikują, natomiast testy E2E (ang. *End-to-End*, od początku do końca) stanowią metodę do symulowania realnych scenariuszy działania użytkownika i sprawdzają poprawność działania systemu od początku do końca.

Na rysunku 6.1 przedstawiono piramidę testów, będąc graficznym modelem prezentującym zależności między poziomami testów, a ich ilością i szybkością. Testów jednostkowych, które są najbardziej zautomatyzowane i najszybsze, powinno być najwięcej. Z kolei im wyżej w piramidzie, tym koszt i czas rosną, więc ilość testów musi maleć.



Rys. 6.1. Piramida testów [*Źródło*: *https://bykowski.pl/piramida-diament-i-trofeum-jak-rozplanowac-testy-automatyczne-w-aplikacji/*]

Popularnym wzorcem do pisania testów jest wzorzec AAA (Arrange-Act-Assert), który pomaga je zrozumieć i uporządkować. Można go podzielić na części:

- **Arrange** (tłum. Przygotowanie) to faza, w której przygotowujemy środowisko testowe, czyli ustawiamy warunki startowe testu. Często obejmuje dodanie zależności, inicjalizację obiektów i zasobów.
- **Act** (tłum. Działanie) na tym etapie wykonujemy konkretne działanie, które chcemy przetestować, czyli na przykład wywołujemy daną funkcję i sprawdzamy jej wynik.
- **Assert** (tłum. Asercja) to część gdzie sprawdzamy czy rezultat testu jest zgodny z tym czego oczekiwaliśmy.

6.1 Testy jednostkowe

Testy jednostkowe to metoda testowania oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów programu - np. metod lub obiektów w programowaniu obiektowym lub procedur
w oprogramowaniu proceduralnym. Testowany fragment programu poddawany jest testowi, który go wykonuje i porównuje wynik z wynikami, których oczekujemy, zarówno pozytywnymi, jak i negatywnymi, ponieważ niepowodzenie działania kodu również może podlegać testowaniu.

Testy są pisane głównie przez programistów i stanowią jeden z elementów wytwarzania oprogramowania. Dopisując nową funkcjonalność w kodzie, programista pokrywa ją testami jednostkowymi, aby sprawdzić, czy ten pojedynczy element zachowuje się tak, jak powinien dla przekazanych danych [8].

W projekcie do przeprowadzenia testów użyłam framework JUnit 5, czyli standardowe narzędzie do testowania aplikacji w języku Java. Oferuje liczne funkcjonalności ułatwiające proces testów:

- **silnik uruchamiania testów** - framework zapewnia uruchamianie testów, ich analizę i zgłaszanie błędów,
- **asercje** - jest to zestaw metod, pozwalających porównać wynik działania kodu z wartościami oczekiwanymi, na przykład:
    - **assertEquals(expected, actual)** - sprawdza czy to co jest wynikiem kodu zgadza się z oczekiwaniami,
    - **assertTrue(condition)** - sprawdza czy wynikiem jest *true*.
- **adnotacje** - metody są identyfikowane dzięki specjalnym adnotacjom pozwalającym na przykład określić, która część kodu ma być wykonana jako test:
    - **@Test** - wskazuje że jest to metoda testowa.
    - **@BeforeEach** - oznacza metody, które należy wykonać przed metodami oznaczonymi @Test, służące do przygotowania danych, inicjalizacji obiektów czy ustawienia wstępnych warunków.
    - **@AfterEach -** metody uruchamiane po metodach oznaczonych @Test, mają czyścić pamięć albo zwalniać zasoby.

6.1.1 Test poprawności filtrowania paczek na podstawie tagów

Po zalogowaniu użytkownik ma możliwość przefiltrowania paczek na podstawie przypisanych do niej tagów. Dodając paczkę, wymagane jest, by określić ją przynajmniej jednym, co umożliwi przypisanie jej do danej kategorii przy wyświetlaniu.

Na listingu 6.1 został przedstawiony test filtrowania paczek na podstawie tagów. Pozostałe warunki (date, time range, userId) zostały ustawione tak, żeby nie wpływać na wynik filtrowania. Jest to fragment klasy *ParcelFilterServiceTest*, zajmującej się przetwarzaniem listy dostępnych ogłoszeń i użyto konwencji *Arrange-Act-Assert*, porządkującej logikę. Na początku inicjalizowany jest obiekt *FilterCriteria*, zawierający listę tagów, po których ma nastąpić filtrowanie. W tym przykładzie wybrano filtr „bakery". Wcześniej w metodzie oznaczonej adnotacją *@Before* przygotowano dwie testowe paczki.

W części *Act* wywoływana jest faktyczna metoda serwisu. Oczekuje się, że zostaną zwrócone tylko paczki posiadające tag „bakery".

Ostatnia część - *Assert*, wykonuje sprawdzenie, czy zwrócono to, czego się spodziewano, czyli że otrzymano tylko jedną paczkę i czy zawiera ona właściwy tag.

Listing 6.1 Test filtrowania paczek na podstawie tagów [Opracowanie własne]

```
@Test
public void testFilterByChips() {
    // Arrange
    ParcelFilterService.FilterCriteria criteria = new ParcelFilterService.FilterCriteria(
            "someone", Arrays.asList("bakery"), "2025-12-05", 0, 1440
    );
    // Act
    List<Parcel> filtered = service.filterParcels(Arrays.asList(parcel1, parcel2), criteria);
    // Assert
    assertEquals(1, filtered.size());
    assertTrue(filtered.get(0).getTags().contains("bakery"));
}
```

6.1.2 Test poprawności pobierania aktywnych paczek

Czasem w testach jednostkowych konieczne jest przetestowanie logiki w izolacji, bez angażowania prawdziwych zależności. Wykorzystuje się wtedy *mocki*, czyli obiekty, które imitują zachowanie prawdziwych komponentów. *Mock* pozwala symulować odpowiedzi i kontrolować scenariusze testów, a także obserwować wywoływania metod.

W Javie wykorzystuje się narzędzie *Mockito*, czyli framework, który umożliwia:

- inicjalizację atrap klas,
- definiowanie zachowania (np. zwróć a, kiedy wywołane będzie b),
- testowanie fragmentów systemu bez uruchamiania jego zależności.

Na listingu 6.2 pokazano test weryfikujący poprawne działanie metody *getActiveParcels()* z klasy *ParcelRepository*, zajmującej się pobieraniem aktywnych ogłoszeń z bazy danych *Firebase Firestore*. Wykorzystany został framework *Mockito*, więc dostęp do bazy zastąpiono mockami.

Test sprawdza poprawne wywołania *Firestore* - powinno odczytać dokumenty z kolekcji *parcels*, stosując filtr „active”, czyli te paczki, które nie zostały przez nikogo zarezerwowane. Następnie symulowane jest działanie *Firebase* poprzez inicjalizację atrap zależności, np. Task albo Query. Test przygotowuje dwa dokumenty i ustawia ich zachowanie tak, by zwracały obiekty *Parcel*. Dzięki *assertEquals()* sprawdzana jest poprawność rozmiaru listy oraz czy identyfikatory i tytuły paczek są zgodne z tymi ustalonymi w mockach.

Listing 6.2 Test pobierania aktywnych paczek [Opracowanie własne]

```
@Test
public void getActiveParcels_Success_ReturnsList() {

    // Arrange
    FirebaseFirestore db = mock(FirebaseFirestore.class);
    CollectionReference col = mock(CollectionReference.class);
    Query query = mock(Query.class);
    Task<QuerySnapshot> task = mock(Task.class);
    QuerySnapshot snapshot = mock(QuerySnapshot.class);

    when(db.collection("parcels")).thenReturn(col);
    when(col.whereEqualTo("status", "active")).thenReturn(query);
    when(query.get()).thenReturn(task);

    QueryDocumentSnapshot d1 = mock(QueryDocumentSnapshot.class);
    QueryDocumentSnapshot d2 = mock(QueryDocumentSnapshot.class);

    Parcel p1 = new Parcel(); p1.setTitle("Parcel 1");
    Parcel p2 = new Parcel(); p2.setTitle("Parcel 2");

    when(d1.toObject(Parcel.class)).thenReturn(p1);
    when(d1.getId()).thenReturn("id1");
    when(d2.toObject(Parcel.class)).thenReturn(p2);
    when(d2.getId()).thenReturn("id2");

    when(snapshot.iterator()).thenReturn(Arrays.asList(d1, d2).iterator());

    ArgumentCaptor<OnSuccessListener<QuerySnapshot>> captor =
            ArgumentCaptor.forClass(OnSuccessListener.class);

    when(task.addOnSuccessListener(captor.capture())).thenReturn(task);

    ParcelRepository.OnParcelsLoadedListener listener =
            mock(ParcelRepository.OnParcelsLoadedListener.class);

    ParcelRepository repo = new ParcelRepository(db);


    // Act
    repo.getActiveParcels(listener);
    captor.getValue().onSuccess(snapshot);


    // Assert
    ArgumentCaptor<List<Parcel>> parcels = ArgumentCaptor.forClass(List.class);
    verify(listener).onSuccess(parcels.capture());

    List<Parcel> result = parcels.getValue();

    assertEquals(2, result.size());
    assertEquals("id1", result.get(0).getId());
    assertEquals("Parcel 1", result.get(0).getTitle());
}
```

6.2 Testy integracyjne

Kolejnym etapem testowania oprogramowania są testy integracyjne, mające na celu wykrycie defektów w interfejsach i interakcjach pomiędzy modułami i systemami. Przeprowadza się je, żeby ocenić zgodność systemu z określonymi wymaganiami funkcjonalnymi. Często jest tak, że funkcje napisane przez różne osoby działają osobno bez błędów, ale po połączeniu pojawiają się problemy we współpracy między nimi [18].

Testy integracyjne można podzielić ze względu na poziom podejścia:

- **Top-Down** - najpierw testuje się moduły na najwyższym poziomie, a te znajdujące się w hierarchii niżej są symulowane. Można na przykład przetestować warstwę UI lub kontrolery, mimo że repozytoria czy serwisy nie są jeszcze gotowe.
- **Bottom-Up** - pierwsze testowane są komponenty położone najniżej, a proces jest kontynuowany do momentu przetestowania tych z najwyższego poziomu. Można na przykład testować bazę danych, mimo że UI nie jest jeszcze gotowe.
- **Big Bang** - jest to sposób testowania, gdzie wszystkie moduły są integrowane jednocześnie. Sprawdza się wówczas, czy cały system działa tak jak oczekiwano. Jednak jeśli pojawi się jakiś błąd, ciężko wykryć jego powód.

6.2.1 Test rejestracji użytkownika i kompletności danych

Pierwszą z funkcjonalności koniecznych do używania aplikacji jest sprawnie działająca rejestracja, bez której użytkownik nie może utworzyć konta, a więc korzystać z żadnych funkcji aplikacji. Proces ten został podzielony na dwa komponenty:

- **AuthService** - odpowiedzialny za komunikację z *Firebase Authentication* i zakładanie konta.
- **UserRepository** - zapisujący dane użytkownika do bazy danych *Firestore.*

Dzięki rozdzieleniu łatwiej przetestować kod, ponieważ wymaga sprawdzenia, czy komponenty poprawnie ze sobą współpracują. Test pokazany listingu 6.3 ma na celu weryfikację poprawności współpracy między AuthService a UserRepository w przypadku udanej rejestracji przez email i hasło.

Do przeprowadzenia testu wykorzystano mocki zewnętrznych zależności (FirebaseAuth, FirebaseFirestore, FCMTokenService), które wymagają połączenia internetowego, generują koszty i są niestabilne. Dzięki temu można symulować różne scenariusze i kontrolować odpowiedzi. Kluczowe dla przeprowadzenia testu było użycie prawdziwych instancji klas AuthService i UserRepository, co pozwala na testowanie ich logiki biznesowej i komunikacji.

Najpierw ustawiane są mocki, aby symulowały udaną rejestrację - FirebaseAuth zwróci użytkownika z ID „user123”. W części *Act* wywoływana jest prawdziwa metoda z AuthService *authService.registerWithEmail(TEST_EMAIL, TEST_PASSWORD, callback)*; która po pomyślnej komunikacji z Firebase wywołuje *UserRepository.createUser()*. Repozytorium pobiera token FCM (identyfikator urządzenia używany do wysyłania powiadomień push) i zapisuje dane do Firestore. Następnie wykonywana jest weryfikacja przepływu oraz danych. Za pomocą metody *assertEquals()* sprawdzane jest, czy dane zapisane do bazy zawierają wymagane pola z ustawionymi wartościami. Test wykrywa błędy:

- **przepływu danych** - np. userId nie jest przekazywane między komponentami.
- **niekompletności danych** - np. brak wymaganego pola w obiekcie userData.
- **sekwencji wywołań** - np. zapis do bazy następuje przed pobraniem tokenu.

Operacje w Firebase są asynchroniczne - nie czekają na wynik funkcji, co może stanowić problem w teście, ponieważ jeśli zakończy się on zbyt szybko, to weryfikacja *assertEquals* nie będzie miała danych do sprawdzenia. Dlatego wykorzystane zostało narzędzie synchronizacyjne CountDownLatch, które blokuje wątek testowy dopóki nie zakończą się operacje asynchroniczne.

Listing 6.3 Test rejestracji użytkownika [Opracowanie własne]

```
@Test
public void testSuccessfulEmailRegistration_AuthAndRepositoryIntegration()
        throws InterruptedException {

    when(mockAuthResult.getUser()).thenReturn(mockUser);
    when(mockUser.getUid()).thenReturn(TEST_USER_ID);
    CountDownLatch latch = new CountDownLatch(1);

    // Mock Auth success
    doAnswer(invocation -> {
        OnCompleteListener<AuthResult> listener = invocation.getArgument(0);
        when(mockAuthTask.isSuccessful()).thenReturn(true);
        when(mockAuthTask.getResult()).thenReturn(mockAuthResult);
        listener.onComplete(mockAuthTask);
        return mockAuthTask;
    }).when(mockAuthTask).addOnCompleteListener(any());

    // Mock Firestore success
    doAnswer(invocation -> {
        OnSuccessListener<Void> listener = invocation.getArgument(0);
        listener.onSuccess(null);
        return mockFirestoreTask;
    }).when(mockFirestoreTask).addOnSuccessListener(any());

    // Act
    authService.registerWithEmail(TEST_EMAIL, TEST_PASSWORD,
            userId -> userRepository.createUser(userId, TEST_LOGIN, TEST_EMAIL,
                    new UserRepository.UserCreationCallback() {
                        @Override
                        public void onSuccess() {
                            latch.countDown();
                        }
                        @Override
                        public void onFailure(Exception e) {
                            latch.countDown();
                        }
                    }),
            e -> latch.countDown()
    );

    // Assert
    assertTrue(latch.await(5, TimeUnit.SECONDS));
    verify(mockAuth).createUserWithEmailAndPassword(TEST_EMAIL, TEST_PASSWORD);
    verify(mockFcmTokenService).getToken(any());

    ArgumentCaptor<Map> captor = ArgumentCaptor.forClass(Map.class);
    verify(mockUserDocument).set(captor.capture());

    Map<String, Object> data = captor.getValue();
    assertEquals(TEST_LOGIN, data.get("login"));
    assertEquals(TEST_EMAIL, data.get("email"));
    assertEquals(TEST_FCM_TOKEN, data.get("fcmToken"));
    assertNotNull(data.get("likedUsers"));}
```

6.2.2 Test pobierania danych w celu dodania oceny

Kiedy użytkownik odbierze paczkę, a jej autor potwierdzi odbiór, możliwe jest wystawienie opinii - napisanie recenzji i zaznaczenie oceny. Aby było to możliwe, konieczna jest współpraca dwóch komponentów:

- **RatingService** - odpowiedzialny za pobieranie właściciela paczki,
- **UserRepository** - odpowiedzialny za zarządzanie danymi użytkowników.

Test pokazany na listingu 6.4 symuluje sytuację, gdy paczka zostanie odebrana i odbiorca chce wystawić ocenę właścicielowi. System musi pobrać informację o tym, kto jest autorem paczki, a później załadować dane, żeby wyświetlić je w interfejsie.

W teście mockowane jest *FirebaseFirestore*, żeby uniknąć połączeń z bazą danych, użyto natomiast prawdziwego serwisu, aby przetestować jego logikę pobierania właściciela i parsowania odpowiedzi oraz repozytorium, żeby sprawdzić poprawność ładowania danych. Dane przepływają więc między zintegrowanymi obiektami. W teście użyto *CountDownLatch* z licznikiem 2, ponieważ scenariusz obejmuje dwie operacje asynchroniczne, które muszą zakończyć się przed weryfikacją wyników: *RatingService.getParcelOwner(parcelId)* oraz *UserRepository.getUserData(authorId)*.

W fazie Arrange przygotowywane są odpowiedzi z *Firebase Firestore* oraz mechanizm do synchronizacji operacji. Następnie w części Act wywoływana jest metoda serwisu RatingService, która pobiera identyfikator właściciela paczki. Po tym wywołaniu metoda *getUserData()* z UserRepository pobiera dane użytkownika na podstawie otrzymanego identyfikatora. W części Assert sprawdzane jest czy zwrócony identyfikator jest poprawny, a obiekt danych został prawidłowo załadowany przez repozytorium, oraz czy otrzymany login użytkownika odpowiada oczekiwanej wartości.

Listing 6.4 Test przepływu danych w celu wystawienia oceny [Opracowanie własne]

```
@Test
   public void testGetParcelOwnerAndLoadUserData_Integration() throws
InterruptedException {
       CountDownLatch latch = new CountDownLatch(2);
       final String[] ownerId = {null};
       final UserRepository.UserData[] userData = {null};

       mockDocument(mockParcelsCollection, "parcel_789", "userId", "author_123");
       mockDocument(mockUsersCollection, "author_123", "login", "TestAuthor");

       // Act
       ratingService.getParcelOwner("parcel_789", new RatingService.ParcelOwnerCallback()
{
           @Override
           public void onSuccess(String authorId, String parcelTitle) {
               ownerId[0] = authorId;
               latch.countDown();

               userRepository.getUserData(authorId, new
UserRepository.OnUserDataLoadedListener() {
                   @Override
                   public void onSuccess(UserRepository.UserData data) {
                       userData[0] = data;
                       latch.countDown();
                   }

                   @Override
                   public void onFailure(Exception e) {
                       latch.countDown();
                   }
               });
           }
       });

       // Assert
       assertEquals("author_123", ownerId[0]);
       assertNotNull(userData[0]);
       assertEquals("TestAuthor", userData[0].login);
   }
```

7. Podsumowanie i wnioski

Celem pracy dyplomowej była implementacja aplikacji mobilnej, która pozwala użytkownikom otrzymywać żywność za darmo lub w niskich kosztach, oraz oddawanie nadmiaru produktów. Wymaganie funkcjonalne i niefunkcjonalne przedstawione we wcześniejszych podrozdziałach zostały w pełni spełnione.

Użytkownicy mają możliwość logowania i rejestracji różnymi metodami. Mogą przeglądać ogłoszenia w formie listy lub na mapie, filtrując je według kategorii, czasu odbioru czy ceny, a także ograniczając wyniki dostosowując obszar odbioru. Zaimplementowano również możliwość sprawnego dodawania ogłoszeń, a także dodawania innych użytkowników do ulubionych oraz oceniania ich aktywności. Można dokonywać płatności online oraz otrzymywać powiadomienia z aplikacji.

Spełniono także założone wymagania niefunkcjonalne. Aplikacja działa płynnie i szybko, a interfejs jest przejrzysty i prosty w obsłudze, co wpływa na komfort użytkowania. Dzięki zastosowaniu Firebase, wszystkie funkcjonalności z bazą danych, uwierzytelnianiem i synchronizacją działają sprawnie i w czasie rzeczywistym.

Dalszy rozwój aplikacji może obejmować dodanie funkcji chatu pomiędzy użytkownikami, co usprawniłoby proces odbioru paczki. Kolejnym możliwym kierunkiem jest wprowadzenie raportów, prezentujących liczbę uratowanych paczek oraz oszczędności finansowe. Możliwe jest dodanie także opcji dostawy dla większych zamówień, co zwiększyłoby funkcjonalność aplikacji.

Podczas realizacji projektu zauważono, że projektowana aplikacja może cieszyć się realnym zainteresowaniem na rynku. Coraz więcej osób chce działać dla ekologii, a jednocześnie poszukuje rozwiązań umożliwiających oszczędzanie środków finansowych.

Implementacja systemu umożliwiła praktyczne wykorzystanie umiejętności nabytych w czasie trwania studiów oraz poznanie nowych technologii zastosowanych w aplikacji. Problemy napotkane podczas rozwoju aplikacji stanowiły cenne doświadczenie, które ma wpływ na dalszy rozwój zawodowy.

8. Bibliografia

[1] Akshat, P. React Native for Mobile Development: Harness the Power of React Native to Create Stunning iOS and Android Applications. 2019

[2] Cay S. Horstmann. Java. Podstawy. Helion wydanie X 2016

[3] Herbert Schildt. Java. Kompendium programisty. Wydanie XI. Helion. 2005

[4] Zoła, A. Programowanie w języku Java. 2004

[5] *Android Studio* [Online]
https://uci.usz.edu.pl/slownik-it/android-studio/

[6] *Android Studio. Jak zainstalować i skonfigurować środowisko programistyczne* [Online]
https://nofluffjobs.com/pl/log/praca-w-it/poczatki-w-it/android-studio-jak-zainstalowac-i-skonfigurowac-srodowisko-programistyczne/

[7] *Backend-as-a-Service (BaaS): Co to jest i jakie są główne zalety korzystania z tego modelu usługowego?* [Online]
https://boringowl.io/blog/backend-as-a-service-baas-co-to-jest-i-jakie-sa-glowne-zalety-korzystania-z-tego-modelu-uslugowego

[8] *CZYM SĄ TESTY JEDNOSTKOWE?* [Online]
https://www.wyszkolewas.com.pl/czym-sa-testy-jednostkowe/

[9] *Cloud Firestore* [Online]
https://firebase.google.com/docs/firestore

[10] *Co to jest Material Design i jak z niego korzystać?* [Online]
https://semcore.pl/co-to-jest-material-design-i-jak-z-niego-korzystac/

[11] *Develop a UI with Views* [Online]
https://developer.android.com/studio/write/layout-editor

[12] *Firebase Authentication* [Online]
https://firebase.google.com/docs/auth

[13] *Index TIOBE* [Online]
https://www.tiobe.com/tiobe-index/

[14] *Overview* [Online]
https://developers.google.com/maps/documentation/places/web-service/overview

[15] *Run apps on the Android Emulator* [Online]
https://developer.android.com/studio/run/emulator
[16] *Stripe Polska – przewodnik po płatnościach, prowizjach i integracji* [Online]
https://wise.com/pl/blog/stripe-polska
[17] *stripe-android* [Online]
https://github.com/stripe/stripe-android
[18] *Testy integracyjne* [Online]
https://pl.wikipedia.org/wiki/Testy_integracyjne
[19] *What is Google Maps API? Benefits, Use Cases, and Implementation* [Online]
https://www.tatvic.com/blog/what-is-google-maps-api/

9. Spis rysunków

10. Spis listingów