

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH

PRACA DYPLOMOWA
INŻYNIERSKA

System wspomagający organizację gry miejskiej
wykorzystującej urządzenia z systemem Android

Management system for location-based game for
Android devices

AUTOR:

Adam Zimny, 209787

PROWADZĄCY PRACĘ:

dr inż. Paweł Rogaliński, W-4, K-9

OCENA PRACY:

WROCŁAW, 2016

Spis treści

Spis tabel	5
Spis listingów	6
1. Wstęp	9
1.1. Cel projektu	9
1.2. Zakres projektu	9
2. Zasady gry	10
2.1. Zlecenia	10
2.2. Lokalizacja gracza	11
2.3. Punktacja	11
2.4. Rejestracja do gry	11
2.5. Słownik pojęć	12
3. Analiza wymagań systemu	13
3.1. Wymagania funkcjonalne	13
3.1.1. Aplikacja kliencka	13
3.1.2. Aplikacja serwerowa	15
3.1.3. Panel administracyjny	17
3.2. Konfigurowalne parametry systemu	17
4. Projekt systemu	19
4.1. Architektura systemu	19
4.2. Przypadki użycia	19
4.3. Projekt interfejsu	26
4.3.1. Aplikacja kliencka	26
4.4. Projekt bazy danych	30
4.4.1. Model konceptualny	31
4.4.2. Model fizyczny	32
4.5. Protokół komunikacji w systemie	32
4.5.1. Ścieżki dostępu API	33
4.5.2. Struktury danych	43
5. Implementacja systemu	45
5.1. Aplikacja serwerowa	45
5.1.1. Wykorzystywane technologie i narzędzia	45
5.1.2. Struktura plików projektu	45
5.1.3. Zabezpieczenie dostępu do API	46

5.1.4.	Baza danych	48
5.1.5.	Komunikacja i przepływ danych	50
5.1.6.	Implementacja obsługi wybranych zapytań	51
5.1.7.	Automatyzacja działania systemu	54
5.2.	Aplikacja kliencka	57
5.2.1.	Struktura projektu	57
5.2.2.	Biblioteki wykorzystane w projekcie	58
5.2.3.	Tworzenie interfejsu aplikacji	59
5.2.4.	Komunikacja z serwerem	65
5.2.5.	Rejestracja konta	67
5.2.6.	Logowanie do konta	73
5.2.7.	Lokalizacja urządzenia	75
5.2.8.	Lista zleceń	78
5.2.9.	Szczegóły zlecenia	83
5.2.10.	Mapa znanych lokalizacji użytkownika	85
5.2.11.	Generowanie kodów potwierdzających	86
6.	Testy systemu	90
7.	Podsumowanie	91
Literatura	92	
A. Opis załączonej płyty CD/DVD	93	

Spis rysunków

4.1.	Diagram przypadków użycia po stronie aplikacji mobilnej	20
4.2.	Diagram przypadków użycia po stronie aplikacji serwerowej	21
4.3.	Ekran startowy aplikacji	26
4.4.	Ekran rejestracji	27
4.5.	Ekran logowania	27
4.6.	Ekran zapisu lokalizacji po logowaniu	28
4.7.	Lista zleceń	28
4.8.	Ekran rejestracji	29
4.9.	Mapa znanych lokalizacji użytkownika	29
4.10.	Panel nawigacyjny aplikacji	30
4.11.	Ekran Generowanie kodu	30
4.12.	Model koncepcyjny bazy danych	31
4.13.	Model fizyczny bazy danych	32
5.1.	Panel nawigacyjny w aplikacji - implementacja	65
5.2.	Przykład przebiegu procesu wyboru zdjęcia	70
5.3.	Ekran zapisu lokalizacji - implementacja	77
5.4.	Widok szczegółów zlecenia - implementacja	84
5.5.	Mapa lokalizacji użytkownika po dodaniu znaczników	86
5.6.	Widok generowania kodu podczas ładowania przycisku	87

Spis tabel

5.1. Parametry wyrażeń cron	55
---------------------------------------	----

Spis listingów

5.1.	Konfiguracja Spring Security	46
5.2.	Konfiguracja bazy danych	48
5.3.	Przykładowa klasa modelu	49
5.4.	Przykładowy interfejs DAO	50
5.5.	Zapytanie SQL zawierające łączenie tabel	50
5.6.	Przykładowy kontroler	50
5.7.	Przykładowy serwis aplikacji	51
5.8.	Kontroler rejestracji	51
5.9.	Metoda rejestracji użytkownika w klasie <code>UserServiceImpl</code>	52
5.10.	Kontroler lokalizacji użytkownika	52
5.11.	Metody pobierania lokalizacji w <code>LocationService</code>	53
5.12.	Fragment kontrolera zleceń	53
5.13.	Metoda serwisu <code>TargetService</code> potwierdzająca wykonanie zleceń	54
5.14.	Konfiguracja planera zadań	54
5.15.	Zadanie generujące nowe zlecenia	56
5.16.	Pobieranie listy użytkowników z uwzględnieniem odległości	56
5.17.	Zadanie zmieniające widoczność zleceń	57
5.18.	Metoda serwisu <code>TargetService</code> pobierająca zlecenia wymagające zmian.	57
5.19.	Przykładowy plik widoku aplikacji	59
5.20.	Tworzenie powiązań interfejsu przy pomocy biblioteki <code>ButterKnife</code>	60
5.21.	Klasa <code>IntentHelper</code>	61
5.22.	Wywołanie metody klasy <code>IntentHelper</code>	61
5.23.	Plik <code>AndroidManifest.xml</code>	62
5.24.	Panel nawygacyjny aplikacji	63
5.25.	Lista opcji dostępna na panelu nawigacyjnym	64
5.26.	Lista opcji dostępna na panelu nawigacyjnym	65
5.27.	Przykładowe interfejsy zapytań <code>Retrofit</code>	65
5.28.	Klasa wspomagająca generowanie implementacji interfejsów biblioteki <code>Retrofit</code>	66
5.29.	Użycie biblioteki <code>Retrofit</code>	67
5.30.	Wywołanie okna wyboru obrazu	68
5.31.	Obsługa wyników aktywności wyboru zdjęcia	68
5.32.	Wyświetlanie okna dialogowego kalendarza	70
5.33.	Interfejs zapytań serwisu <code>Imgur</code>	71
5.34.	Wyświetlanie okna dialogowego kalendarza	72
5.35.	Logowanie do konta	73
5.36.	Sprawdzenie wymagań użycia modułu GPS	75
5.37.	Rozpoczęcie aktualizacji GPS	77
5.38.	Obsługa aktualizacji i zapis do bazy	77
5.39.	Adapter fragmentów widoku zleceń	78

5.40. Adapter fragmentów widoku zleceń	79
5.41. Adapter fragmentów widoku zleceń	79
5.42. Metoda odświeżania listy zleceń	81
5.43. Konfiguracja słuchaczy klasy TargetsPageFragment	82
5.44. Metoda uruchamiająca aktywność szczegółów zlecenia	82
5.45. Pobranie danych inicjalizujących z intencji	83
5.46. Pobranie danych inicjalizujących z intencji	83
5.47. Obsługa wywołania aktywności mapy	84
5.48. Fragment pliku interfejsu aktywności MapActivity	85
5.49. Obsługa wywołania aktywności mapy	85
5.50. Obsługa zdarzeń przycisku wykorzystującego HoldToLoadLayout	87
5.51. Generowanie kodu	87
5.52. Generowanie losowego ciągu znaków	88

Skróty

API (ang. *Application Programming Interface*)

DAO (ang. *Database Access Object*)

GPS (ang. *Global Positioning System*)

GSM (ang. *Global System for Mobile Communications, pierwotnie Groupe Spécial Mobile*)

HTTP (ang. *Hypertext Transfer Protocol*)

JSON (ang. *JavaScript Object Notation*)

ORM (ang. *Object-Relational Mapping*)

URI (ang. *Uniform Resource Identifier*)

URL (ang. *Uniform Resource Locator*)

Rozdział 1

Wstęp

Niniejszy dokument opisuje przebieg realizacji projektu dyplomowego o temacie *System wspomagający organizację gry miejskiej wykorzystującej urządzenia z systemem Android* na kierunku Informatyka, na wydziale Elektroniki Politechniki Wrocławskiej.

Grąorską nazywana jest gra, która wykorzystuje przestrzeń świata rzeczywistego jako element swojej mechaniki. Uczestnicy gry wykonują zlecone przez organizatorów zadania na podstawie wskazówek lub instrukcji. Gry miejskie często posiadają element fabularny, nadający kontekstu wykonywanym zadaniom i pozwalający uczestnikom wcielać się w postaci opowiadanej historii. Tematyka gier może również zawierać motyw historyczne, promujące miasto i zachęcające do poznawania jego kultury, szczególnie jeśli prowadzona jest na terenie jednego miasta.

1.1. Cel projektu

Celem projektu jest stworzenie systemu pozwalającego na zorganizowanie i zarządzanie grąorską oraz branie w niej udziału na terenie wybranego obszaru. System powinien prowadzić grę bez konieczności nadzoru ze strony osób zarządzających.

1.2. Zakres projektu

Projekt systemu obejmuje następujące zagadnienia:

- opracowanie zasad gry
- analizę wymagań systemu
- projekt interfejsu użytkownika
- wybór technologii wykonania systemu
- projekt i implementację mobilnej aplikacji klienckiej
- projekt i implementację serwera prowadzącego grę
- projekt i implementację panelu administracyjnego
- przeprowadzenie testów systemu
- stworzenie dokumentacji projektu
- wdrożenie systemu z wykorzystaniem sklepu Google Play

Rozdział 2

Zasady gry

W projektowanej w tej pracy grze zadaniem uczestnika jest fizyczne odnalezienie w wyznaczonym czasie innych uczestników. Po odnalezieniu określonej osoby należy uzyskać od niej hasło, pozwalające na potwierdzenie wykonania zadania. Za wykonywanie zadań uczestnicy są nagradzani punktami, na podstawie których tworzony jest ranking.

Obszar gry nie jest z góry ograniczony przez system. Uczestnicy gry korzystając z aplikacji udostępniają swoją lokalizację, na podstawie której tworzone są dla nich zadania.

2.1. Zlecenia

Zleceniami nazywane są zadania, które wykonują uczestnicy gry. Celem zadania jest odnalezienie wyznaczonego gracza w określonym czasie na podstawie informacji dostępnych w aplikacji. Wyznaczony gracz nazywany jest **szukanym**, zaś gracz, któremu przydzielone zostaje zlecenie **szukającym**.

Na informacje udostępniane o szukanym składają się: imię, nazwisko, wiek i zdjęcie, a także mapa zawierająca znane lokalizacje szukanego zarejestrowane podczas udziału w grze. Szczegóły dotyczące rejestracji lokalizacji opisane zostały w punkcie 2.2.

Zlecenia generowane są automatycznie w procesie zwanym **losowaniem**. Losowania odbywają się w stałych odstępach czasu, ustalanych przez administratorów gry. Każdy gracz może posiadać maksymalnie 5 zleceń prywatnych w danym momencie. Jeśli gracz posiada mniej niż 5 zleceń, mogą one zostać uzupełnione do 5 podczas losowania. Gracze, którzy najdłużej oczekiwali na nowe zlecenie mają pierwszeństwo w losowaniu.

Szukani są przydzielani szukającym na podstawie ich lokalizacji zarejestrowanej w trakcie uczestnictwa w grze. Gracz może ustalić maksymalną odległość, w jakiej mają mu być przydzielane cele. Jeżeli to możliwe, gracz otrzyma zlecenia natychmiast po rejestracji. W przypadku, gdy po rejestracji gracza nie ma możliwości utworzenia nowych zleceń, gracz zostaje dodany do na początek kolejki graczy oczekujących na losowanie. W przypadku gdy gracz powraca do gry po nieaktywności, zostaje on dodany na koniec kolejki. W sytuacji, gdy szukany staje się nieaktywny, szukający zostaje dodany na początek Kolejki.

Wykonanie zlecenia potwierdza się poprzez wygenerowanie kodu w aplikacji na urządzeniu szukanego i wprowadzenie go w aplikacji na urządzeniu szukającego. Kod domyślnie jest aktywny przez 2 minuty. W momencie generowania i wpisania kodu oba urządzenia muszą znajdować się w pobliżu siebie. Wylogowanie z aplikacji powoduje wygaśnięcie wszystkich kodów przypisanych do danego użytkownika.

2.2. Lokalizacja gracza

Aplikacja mobilna korzysta z danych lokalizacyjnych telefonu. Z tego względu wymagane jest włączenie ich w telefonie podczas użytkowania aplikacji i nadanie aplikacji odpowiednich uprawnień. Jeśli w momencie logowania nie jest możliwe uzyskanie tych danych korzystanie z aplikacji zostanie wstrzymane.

Aplikacja rejestruje pozycje w momencie uruchomienia (logowania) oraz podczas generowania i wprowadzania kodu.

Zarejestrowane przez aplikacje lokalizacje będą udostępniane innym graczom wśród danych na temat celu lub listu gończego oraz będą wykorzystywane przy losowaniach celów do ograniczenia obszaru gry. Gracz może ustalić maksymalną odległość, w jakiej mają mu być przydzielane Zlecenia.

2.3. Punktacja

Punkty w grze przydzielane są za:

- wykonanie zlecenia,
- zrealizowanie listu gończego,
- uniknięcie odnalezienia w trakcie trwania zlecenia,
- uniknięcie odnalezienia w trakcie trwania listu gończego.
- codzienne logowanie - gracz otrzymuje punkty za logowanie do aplikacji. Kolejne logowania pod rząd przez kilka dni zwiększą punkty za logowanie.

Punkty uzyskane w grze tracą ważność po upływie 1 roku od ich uzyskania. Gracze są klasyfikowani w rankingu na podstawie sumarycznej wartości punktów uzyskanych w ciągu ostatniego roku. Graczowi przynajmniej się jeden z 5 poziomów rankingowych zależnie od jego miejsca w rankingu. Liczba graczy na każdym z poziomów określona jest procentowo w następujący sposób, zaczynając od graczy z największą liczbą punktów:

- 1% użytkowników
- kolejne 10% użytkowników
- kolejne 20% użytkowników
- kolejne 30% użytkowników
- pozostała grupa użytkownicy

Podczas losowania gracz ma zwiększoną szansę przydzielenia graczy z tego samego poziomu rankingowego.

2.4. Rejestracja do gry

W celu wzięcia udziału w grze wymagane jest stworzenie konta. Rejestracja do gry odbywa się za pomocą aplikacji mobilnej. Podczas rejestracji wymagane jest podanie nazwy użytkownika, hasła, imienia, nazwiska, daty urodzenia, adresu email, krótkiego opisu oraz przesyłania swojego zdjęcia. Informacje podane przy rejestracji mogą zostać zmienione w późniejszym czasie z poziomu edycji profilu. Gracze, którzy nie będą aktywni przez 28 dni zostaną tymczasowo wyłączeni z losowań celów i zostaną o tym powiadomieni mailowo. Gracz może powrócić do normalnego trybu rozgrywki z poziomu ustawień aplikacji. Gracze łamiący zasady gry mogą zostać zgłoszeni przez innych graczy z poziomu aplikacji. Po otrzymaniu zgłoszeń od więcej niż

X użytkowników, gracz otrzyma powiadomienie mailowe. Po otrzymaniu kolejnych Y zgłoszeń, konto gracza zostanie zablokowane w celu kontroli przez administratora.

2.5. Słownik pojęć

W tym punkcie przedstawiona została lista pojęć używanych w pracy do określania istotnych elementów lub aspektów rozgrywki.

Gracz - zarejestrowany użytkownik systemu nie posiadający uprawnień administracyjnych, biorący udział w rozgrywce

Administrator - zarejestrowany użytkownik systemu posiadający uprawnienia do zmiany parametrów gry

Zlecenie - zadanie znalezienia innego gracza przydzielane zarejestrowanym użytkownikom, może być prywatne lub publiczne. Zlecenia prywatne widoczne są tylko dla użytkownika, któremu są przydzielone, zaś publiczne dla wszystkich użytkowników

Szukany - gracz, którego odnalezienie zostało nakazane w zleceniu lub w Liście gończym

Szukający - gracz, któremu przydzielone zostało zlecenie

Kod - hasło o krótkim okresie ważności używane do potwierdzania znalezienia celu, generowane w aplikacji

Aplikacja lub **Klient** - aplikacja kliencka gry na telefony z systemem Android, przeznaczona do rejestracji i odczytywania zleceń

Serwer - aplikacja serwerowa zapewniająca komunikację między klientem a bazą danych, a także zajmująca się prowadzeniem gry

Losowanie zleceń - procedura dobierania w pary szukanych i szukających, której wynikiem jest utworzenie nowego zlecenia

Wykonanie zleceń - znalezienie szukanego gracza i wprowadzenie wygenerowanego przez niego kodu do aplikacji

Kolejka - lista graczy oczekujących na losowanie, posiadających mniej niż maksimum zleceń prywatnych.

Ranking - określana na podstawie punktacji klasyfikacja graczy

Poziomy rankingowe - wyróżnione w rankingu stopie osiągane przez zdobywanie punktów

Rozdział 3

Analiza wymagań systemu

3.1. Wymagania funkcjonalne

Po przeprowadzeniu analizy opisu zasad gry ustalone wymagania funkcjonalne. Poniższe podrozdziały opisują opracowane wymagania z wyróżnieniem części systemu, do której przynależą.

3.1.1. Aplikacja kliencka

Wymagania funkcjonalne aplikacji klienckiej skupią się wokół zapewnienia użytkownikom wglądu w ich konto.

1. Logowanie - aplikacja powinna pozwalać na zalogowanie istniejącego użytkownika przy pomocy nazwy użytkownika i hasła. W aplikacji należy umożliwić zapamiętanie danych logowania. Po zapamiętaniu danych logowanie do konta użytkownika powinno nastąpić automatycznie po uruchomieniu aplikacji. Po wprowadzeniu nieprawidłowych danych logowania aplikacja powinna wyświetlić stosowny komunikat. Po poprawnym logowaniu aplikacja przechodzi do zapisu lokalizacji.
2. Rejestracja - aplikacja powinna umożliwiać rejestrację nowego konta. W procesie rejestracji należy podać dane osobowe oraz dane logowania, a także wybrać zdjęcie. Po wyborze zdjęcia aplikacja powinna umożliwić przycięcie go. Po kliknięciu w pole Data urodzenia powinien wyświetlić się kalendarz. Typ pola do wprowadzenia danych powinien odpowiadać wprowadzanym danym (email, hasło, imię itd), tak aby na urządzeniu wyświetlała się właściwa klawiatura. Przed zarejestrowaniem użytkownika należy zweryfikować poprawność wprowadzonych danych oraz zdjęcia. W przypadku niewprowadzenia któregoś z elementów lub braku weryfikacji należy wyświetlić komunikat i zaznaczyć brakujący element. Po poprawnym zakończeniu rejestracji aplikacja powinna powrócić do ekranu startowego.
3. Wykrycie twarzy na zdjęciu - aplikacja przed przesłaniem zdjęcia na serwer powinna dokonać próby wykrycia twarzy na zdjęciu. W przypadku niewykrycia twarzy, wykrycia więcej niż jednej twarzy, lub za małej powierzchni zdjęcia zajętej przez twarz należy wyświetlić komunikat z prośbą o wybór innego zdjęcia lub inne jego przycięcie. Po poprawnym wykryciu twarzy na zdjęciu należy wyświetlić informację o poprawności zdjęcia na ekranie.
4. Przesłanie zdjęcia - aplikacja powinna umożliwić przesyłanie zdjęcia na serwer. W przypadku błędu podczas przesyłania należy wyświetlić komunikat o błędzie. Po poprawnym przesłaniu zdjęcia należy przypisać jego identyfikator użytkownikowi.

5. Zapis lokalizacji przy logowaniu - po zalogowaniu użytkownika należy zapisać lokalizację do bazy danych. Jeśli aplikacja nie posiada uprawnień do modułu GPS urządzenia, należy o nie poprosić. Jeśli użytkownik odmówi uprawnień lub moduł GPS jest wyłączony należy wyświetlić ekran z informacją o konieczności jego włączenia. Po uzyskaniu dostępu do modułu aplikacja powinna poczekać na ustalenie lokalizacji z dobrą dokładnością (kilka metrów) a następnie wykonać zapis do bazy. Po poprawnym wykonaniu zapisu aplikacja przechodzi do widoku Zleceń.
6. Wyświetlenie Zleceń - aplikacja powinna umożliwiać wyświetlenie Zleceń użytkownika z podziałem na publiczne i prywatne. Zlecenia powinny być pobrane automatycznie po wyświetleniu ekranu. Należy umożliwić odświeżenie listy poprzez gest pociągnięcia w dół, gdy lista jest przewinięta do początku. Elementy listy powinny wyświetlać dane osobowe celu, zdjęcie, odległość od użytkownika oraz pozostały czas. Pozostały czas zlecenia powinien być aktualizowany w czasie rzeczywistym. Zlecenia zakończone powinny zamiast czasu wyświetlać napis z ich stanem. W przypadku błędu podczas pobierania listy celów należy wyświetlić komunikat o błędzie. Kliknięcie na element listy powinno otworzyć ekran szczegółów zlecenia.
7. Szczegóły zlecenia - aplikacja powinna wyświetlić szczegóły zlecenia po wyborze z listy Zleceń. Zdjęcie celu powinno zajmować możliwie dużą powierzchnię ekranu. Na ekranie należy wyświetlić informacje o celu, odległość od użytkownika i pozostały czas. Pozostały czas powinien być aktualizowany w czasie rzeczywistym. Ekran Szczegóły powinien zawierać pole tekstowe oraz przycisk służące do wprowadzenia kodu potwierdzającego. Gest pociągnięcia w dół na ekranie powinien pokazać mapę zawierającą lokalizacje Szukanego. Po kliknięciu na marker lokalizacji na mapie powinno zostać wyświetlone okno z datą dodania znacznika. Ekran Szczegółów powinien umożliwiać raportowanie użytkowników łamiących zasady.
8. Potwierdzenie wykonania Zlecenia - użytkownik powinien mieć możliwość potwierdzenia wykonania Zlecenia poprzez wprowadzenie kodu generowanego w aplikacji szukanego użytkownika. Po wprowadzeniu kodu aplikacja powinna przesłać kod na serwer w celu sprawdzenia jego poprawności w bazie danych. W przypadku kodu niepoprawnego należy wyświetlić komunikat oraz zablokować możliwość ponownego wprowadzenia kodu na 10 sekund. Po wprowadzeniu poprawnego kodu Zlecenia zmienia status na wykonane.
9. Generowanie kodu - aplikacja umożliwia wygenerowanie losowego kodu służącego do potwierdzania zleceń. W momencie generowania aplikacja powinna zapamiętać czas generowania i zapisać go w bazie danych. Na ekranie powinien zostać wyświetlony czas ważności kodu. Czas ważności powinien być aktualizowany w czasie rzeczywistym.
10. Zgłoszenie gracza - aplikacja powinna umożliwiać zgłoszenie gracza łamiącego zasadę gry. Po wyborze opcji Zgłoś gracza powinno pokazać się okno dialogowe z możliwością wyboru jednego powodów zgłoszenia, polem tekstowym do opisania sytuacji i przyciskami potwierdzającym i zamkającym okno dialogowe. Po potwierdzeniu zgłoszenie zostanie przesłane na serwer, a okno dialogowe powinno zostać zamknięte.
11. Nawigacja w aplikacji - aplikacja powinna umożliwiać nawigowanie pomiędzy dostępnymi ekranami przy pomocy menu bocznego wysuwanego z lewej strony. Kliknięcie elementu listy powinno zmienić ekran i zamknąć menu. Kliknięcie poza wysuniętym menu lub gest przeciągnięcia w lewo powinny zamknąć menu.
12. Profil użytkownika - aplikacja powinna udostępnić ekran wyświetlający profil zalogowanego użytkownika w ten sam sposób, w jaki wyświetlane są szczegóły zlecenia, z pominięciem elementów interfejsu dotyczących czasu zlecenia i kodu potwierdzającego. Widok profilu użytkownika powinien pozwolić na edycję danych użytkownika oraz zmianę

zdjęcia. Przy edycji profilu powinna być możliwość zmiany maksymalnej odległości zleceń. 16. Strona startowa - aplikacja powinna zawierać ekran startowy zawierający grafikę promocyjną i przyciski Zarejestruj oraz Zaloguj. Strona startowa powinna być wyświetlana przy pierwszym uruchomieniu aplikacji, oraz przy kolejnych jeśli użytkownik nie skorzystał z opcji automatycznego logowania.

13. Wylogowanie - aplikacja powinna umożliwiać wylogowanie z konta użytkownika. Po wylogowaniu dotychczas zapamiętana nazwa użytkownika i hasło powinny zostać zapomniane. Przy kolejnym uruchomieniu aplikacja powinna wyświetlić ekran startowy.
14. Szkolenie - przy pierwszym uruchomieniu aplikacji powinno zostać wyświetlane krótkie szkolenie w postaci slajdów opisujące krótko najważniejsze funkcje aplikacji oraz zasady gry.
15. Ustawienia aplikacji - w aplikacji powinna być możliwość edycji preferencji dotyczących otrzymywanych powiadomień.
16. Ocena - aplikacja powinna posiadać skrót do strony aplikacji w sklepie Google Play.
17. Strona "O aplikacji" - aplikacja powinna umożliwiać wyświetlenie strony informacyjnej o aplikacji zawierającej dane na temat obecnej wersji, autora, wykorzystanych bibliotek i ich licencji.
18. Kontakt z twórcą - aplikacja powinna umożliwiać przesyłanie informacji o błędach lub opinii do twórcy aplikacji. Przesyłanie może być obsłużone za pomocą zewnętrznej aplikacji pocztowej lub wbudowane w aplikację.

3.1.2. Aplikacja serwerowa

1. Połączenie z bazą danych - system w ramach swojego działania powinien mieć możliwość nawiązania połączenia i wykonywania operacji na bazie danych.
2. Rejestracja - system pozwala na zarejestrowanie nowego konta. Podczas rejestracji należy nadać kontu odpowiednie uprawnienia.
3. Uwierzytelnienie użytkownika - system powinien być zabezpieczony przed dostępem osób nieuwierzytelnionych. W celu uzyskania dostępu użytkownik musi podać login i hasło wybrane przy rejestracji. System powinien działać w trybie bezstanowym tj. nie przechowywać informacji o zalogowanym użytkowniku lub sesji.
4. Sprawdzenie dostępności nazwy użytkownika - system powinien umożliwiać sprawdzenie zadanej nazwy użytkownika pod kątem występowania konta o tej nazwie w bazie danych.
5. Sprawdzenie adresu email - system powinien umożliwiać sprawdzenie zadanego adresu email pod kątem występowania konta z tym adresem w bazie danych.
6. Pobranie konta użytkownika - system powinien umożliwić pobranie danych o koncie użytkownika na podstawie jego numeru id lub nazwy użytkownika.
7. Pobranie zleceń użytkownika - system powinien umożliwić pobranie zleceń dostępnych dla użytkownika na podstawie jego numeru id.
8. Pobranie lokalizacji użytkownika - system powinien umożliwić pobranie znanych lokalizacji użytkownika nowszych niż zadana data.
9. Zapis lokalizacji - system powinien umożliwić zapis nowej lokalizacji do bazy danych. We wpisie należy zatrzymać datę jej utworzenia, użytkownika, któremu jest przypisana, a także informację o typie wpisu (logowanie, wykonanie zlecenia, wygenerowanie kodu).
10. Zapis kodu - system powinien umożliwić zapis nowo wygenerowanego kodu potwierdzającego wykonanie zlecenia do bazy danych. We wpisie należy zatrzymać informację o dacie wykonania wpisu oraz o użytkowniku, który wygenerował kod.
11. Potwierdzenie wykonania zlecenia - system powinien umożliwić potwierdzenia wykonania zlecenia za pomocą kodu wygenerowanego w aplikacji Szukanej.

12. Pobranie punktacji użytkownika - system powinien umożliwić pobranie informacji o sumarycznej liczbie punktów uzyskanych przez użytkownika oraz miejscu w rankingu na podstawie tej punktacji.
13. Pobranie przedziałów punktowych - system powinien umożliwić wyznaczenie przedziałów punktowych na podstawie ustalonych progów. Użytkownik powinien mieć możliwość sprawdzenia liczby punktów potrzebnych do zmiany obecnie zajmowanego przez niego przedziału na wyższy lub niższy.
14. Pobranie rankingu najlepszych graczy - system powinien umożliwić pobranie informacji o najlepszych graczach w rankingu.
15. Pobranie uśrednionej lokalizacji - system powinien umożliwić pobranie uśrednionej lokalizacji użytkownika na podstawie jego numeru id. Lokalizację uśredzoną należy wyznaczyć poprzez znalezienie takiej lokalizacji użytkownika, która w swoim otoczeniu ma maksymalną liczbę innych lokalizacji. W przypadku, gdy liczba ta jest równa dla dwóch lub więcej wpisów, należy wybrać najbardziej aktualną.
16. Pobranie najnowszej lokalizacji - system powinien umożliwić pobranie najbardziej aktualnej lokalizacji użytkownika na podstawie jego numeru id.
17. Edycja profilu - system powinien umożliwiać edycję danych na koncie użytkownika
18. Zgłoszenie łamania zasad - system powinien umożliwiać dodanie nowego zgłoszenia o złamaniu zasad przez użytkownika. Danymi wymaganymi do dodania zgłoszenia jest id użytkownika zgłaszającego oraz zgłoszonego i powód zgłoszenia. Opcjonalne jest dodanie opisu słownego. System powinien zarejestrować datę przyjęcia zgłoszenia.
19. Zamrożenie i odmrożenie konta - w przypadku wydłużonej nieaktywności Gracz powinien mieć możliwość wstrzymania działalności na swoim koncie poprzez jego zamrożenie. Po ponownym rozpoczęciu aktywności system powinien umożliwić odmrożenie konta. Konta zamrożone nie biorą udziału w rozgrywce.
20. Usunięcie konta - system powinien umożliwić użytkownikowi usunięcie swojego konta. Po usunięciu konta powinny zostać także usunięte wszystkie inne wpisy w bazie danych powiązane z usuwanym kontem.
21. Zmiana hasła - system powinien umożliwić zmianę hasła użytkownikowi. Po prośbie o zmianę hasła wysłanej z poziomu aplikacji mobilnej system powinien wysłać email z linkiem potwierdzającym zmianę. Po wejściu w link należy wygenerować losowo nowe hasło i wysłać je mailowo użytkownikowi.
22. Utworzenie nowych zleceń - system powinien automatycznie co określony czas przeprowadzić losowanie nowych zleceń dla użytkowników. Czas, co jaki tworzone są zlecenia jest parametrem systemu i należy go pobrać z bazy danych po każdym losowaniu.
23. Wygaśnięcie zleceń - system powinien automatycznie zmieniać status aktywnych zleceń na niewykonane, w przypadku gdy zlecenie pozostaje niewykonane po przekroczeniu czasu ważności. Czas ważności zlecenia publicznego to parametr systemu, który należy pobrać z bazy danych.
24. Zmiana widoczności zleceń - system powinien automatycznie zmieniać widoczność zleceń z prywatnych na publiczne, w przypadku gdy zlecenie prywatne pozostaje niewykonane po przekroczeniu czasu ważności. Czas ważności zlecenia prywatnego to parametr systemu, który należy pobrać z bazy danych.
25. Wygasanie kodów - system powinien automatycznie zmienić status kodów w bazie danych na nieaktualne po upływie ich czasu ważności. Czas ważności kodu to parametr systemu, który należy pobrać z bazy danych.
26. Wykrywanie nieaktywnych użytkowników - system powinien automatycznie wykrywać długie okresy nieaktywności użytkowników. Użytkownik po przekroczeniu określonego

czasu nieaktywności powinien otrzymać mailową informację o nieaktywności. W przypadku braku aktywności przez kolejny okres czasu, konto użytkownika należy zamrozić. Użytkownik powinien zostać poinformowany o zamrożeniu konta drogą mailową. Czas, po którym wysyłane jest przypomnienie i czas zamrożenia konta to parametry systemu, które należy pobrać z bazy danych.

27. Blokada użytkowników - system powinien automatycznie blokować konta użytkowników, którzy przekroczą określona liczbę zgłoszeń o złamaniu regulaminu. Wielokrotne zgłoszenia od tego samego użytkownika należy liczyć pojedynczo. Użytkownik, którego konto zostało zablokowane powinien otrzymać powiadomienie mailowe.
28. Wysyłanie powiadomień push - system powinien mieć możliwość wysyłania powiadomień push na telefony z systemem Android.
29. Zmiana parametrów systemu - system powinien udostępniać API pozwalające na zmianę zadanego parametru systemu przez podanie jego nazwy oraz nowej wartości. Parametry zostały opisane w punkcie 3.2.
30. Odczyt parametrów systemu - system powinien pozwalać na odczyt ustawionej wartości parametru na podstawie jego nazwy.

3.1.3. Panel administracyjny

1. Kontrola dostępu - panel administracyjny powinien pozwalać na dostęp jedynie użytkownikom posiadającym odpowiednie uprawnienia.
2. Logowanie - panel powinien pozwalać na logowanie użytkownika
3. Wyświetlenie listy użytkowników - z poziomu panelu administrator powinien mieć możliwość pobrania i wyświetlenia listy użytkowników. Należy zapewnić możliwość filtrowania listy na podstawie kryteriów: statusu konta, daty rejestracji, daty ostatniej aktywności.
4. Wyświetlenie listy zgłoszeń - administrator powinien mieć możliwość wyświetlenia listy zgłoszeń o złamaniu zasad. Należy zapewnić możliwość filtrowania listy na podstawie kryteriów: zgłoszony użytkownik, zgłaszający użytkownik, data zgłoszenia, rodzaj zgłoszenia.
5. Zmiana statusu konta - administrator powinien mieć możliwość zablokowania lub odblokowania konta wybranego użytkownika
6. Zmiana uprawnień administracyjnych - administrator powinien mieć możliwość nadania lub cofnięcia uprawnień administracyjnych wybranego użytkownika poza wyznaczonym kontem nadzorującym
7. Zmiana parametrów gry - panel administracyjny powinien pozwalać na odczyt i zmianę parametrów systemu. Lista parametrów systemu przedstawiona została w punkcie 3.2.

3.2. Konfigurowalne parametry systemu

W celu zapewnienia maksymalnej elastyczności działania systemu wprowadzone zostały parametry zarządzające przebiegiem gry. Parametry te mogą zostać zmienione w dowolnym momencie z poziomu panelu administracyjnego, bez konieczności wstrzymywania gry. Parametry są odczytywane z bazy danych za każdym razem, gdy ich wartość jest potrzebna. Dzięki takiemu rozwiązaniu po zmianie wartości parametru przy kolejnym jego użyciu przez aplikację, odczytana zostanie zawsze aktualna wartość. Do konfigurowalnych parametrów systemu należą:

- czas trwania zlecenia prywatnego
- czas trwania zlecenia publicznego
- liczba punktów przydzielana za wykonanie zlecenia prywatnego

- liczba punktów za wykonanie zlecenia publicznego
- czas ważności uzyskanych punktów
- czas ważności wygenerowanego kodu
- czas, po którym użytkownik otrzyma przypomnienie o nieaktywności
- czas, po którym konto użytkownika zostanie zamrożone
- maksymalna liczba zleceń prywatnych, jaką może posiadać gracz
- czas ważności lokalizacji użytkownika
- minimalna powierzchnia zdjęcia zajęta przez twarz w procentach

Rozdział 4

Projekt systemu

4.1. Architektura systemu

System zostanie podzielony na cztery części:

- aplikację mobilną na telefony z systemem Android,
- aplikację internetową realizującą funkcje konsoli administracyjnej
- serwer prowadzący grę
- bazę danych

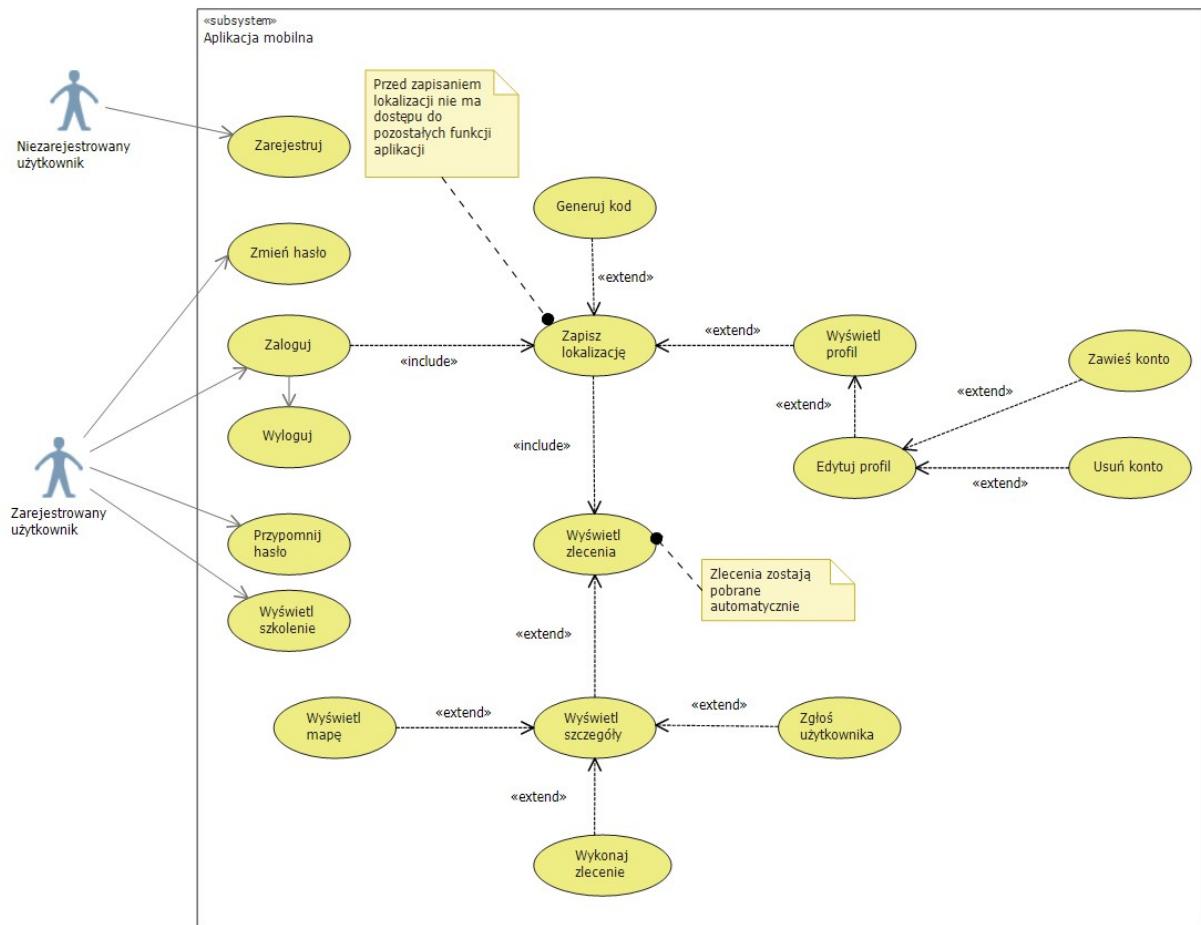
Aplikacja mobilna będzie pełnić rolę klienta gry i przeznaczona będzie dla graczy. Z jej pomocą użytkownicy będą mogli odczytywać i wykonywać przydzielone im zlecenia, edytować swój profil, przeglądać ranking i zarządzać swoim kontem. Urządzenie mobilne zostanie wykorzystane do określenia lokalizacji użytkowników na potrzeby gry.

Konsola administracyjna służyć będzie administratorom gry do obserwowania jej przebiegu i modyfikowania parametrów, a także zarządzania kontami użytkowników.

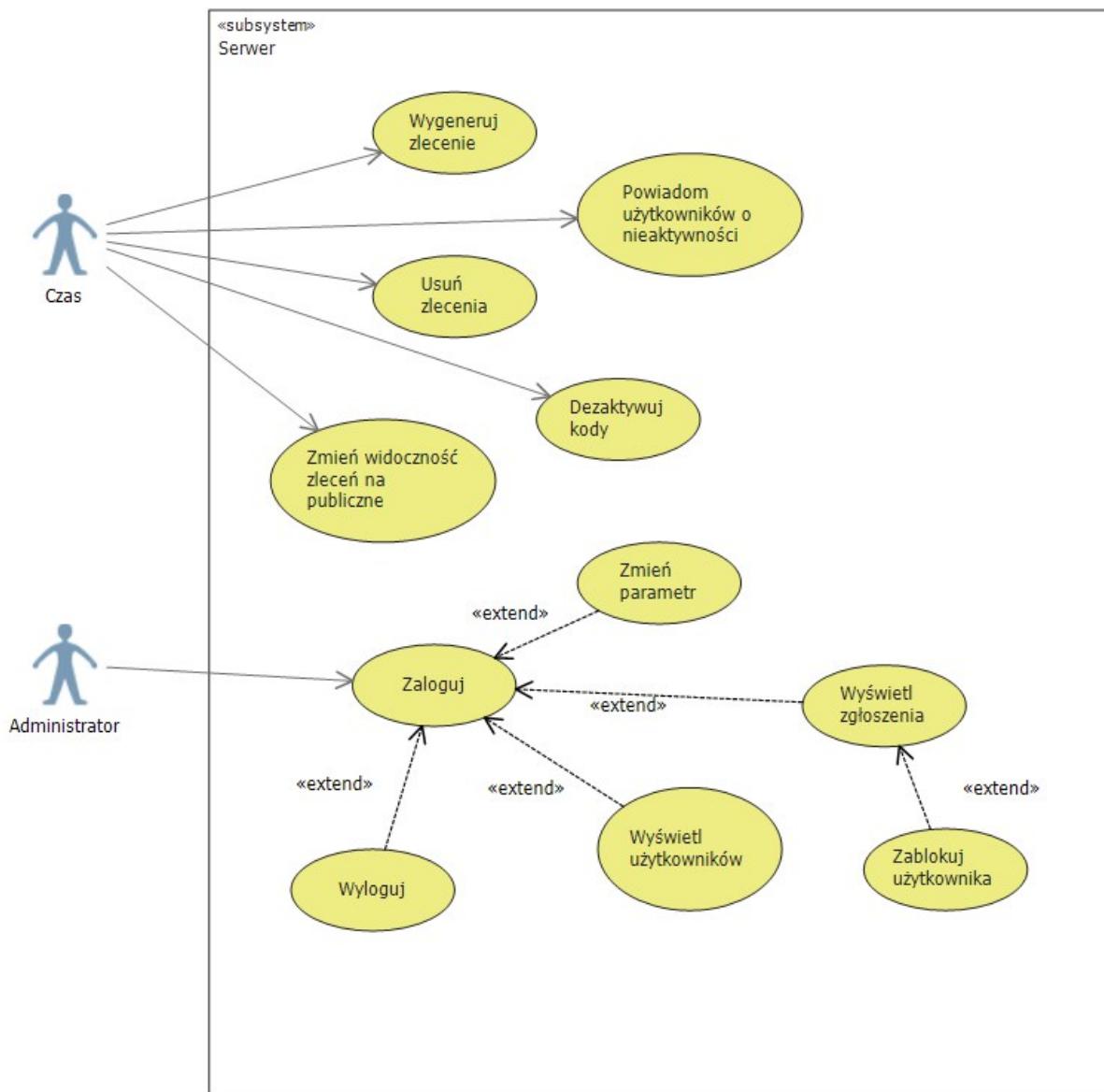
Zadaniem serwera jest automatyczne prowadzenie gry bez konieczności udziału administratorów systemu. W ramach pracy serwera powinny być generowane lub usuwane zlecenia prywatne i publiczne, wysyłane powiadomienia do graczy, a także monitorowana aktywność graczy. Serwer wykorzystywany będzie także do stworzenia interfejsu komunikacyjnego pomiędzy aplikacjami klienckimi lub konsolą administracyjną a bazą danych.

4.2. Przypadki użycia

Na podstawie sprecyzowanych powyżej zasad gry i opisu działania systemu opracowano przypadki użycia systemu. Diagramy przypadków użycia przedstawiają rysunki 4.1 oraz 4.2.



Rys. 4.1: Diagram przypadków użycia po stronie aplikacji mobilnej



Rys. 4.2: Diagram przypadków użycia po stronie aplikacji serwerowej

Poniżej znajdują się przykładowe scenariusze opisujące działania użytkownika, jakie mogą zajść podczas korzystania systemu. Ze względu na zautomatyzowaną naturę części serwerowej systemu, opis scenariuszy skupia się głównie na czynnościach wykonywanych przez użytkownika aplikacji mobilnej. W scenariuszach zostały wyszczególnione sytuacje wyjątkowe oznaczające błąd w podstawowym przebiegu interakcji prowadzący do jej niepoprawnego zakończenia oraz przebiegi alternatywne opisujące inny zestaw kroków prowadzący do tego samego wyniku.

1. Rejestracja

Aktor

Niezarejestrowany użytkownik aplikacji

Zdarzenie inicjujące

Użytkownik będąc na ekranie startowym aplikacji kliknął przycisk „Rejestracja”.

Warunki początkowe

brak

Opis przebiegu interakcji

1. W aplikacji wyświetlany jest ekran Rejestracja.
2. Użytkownik kliką przycisk „Wybór zdjęcia”.
 - 2.1. Na telefonie zostaje otwarta aplikacja do wyboru obrazu
 - 2.2. Użytkownik wybiera obraz
 - 2.3. Aplikacja wyświetla widok pozwalający na przycięcie zdjęcia
 - 2.4. Użytkownik ustawia przycięcie i potwierdza
 - 2.5. Przyjęte zdjęcie wyświetla się w miejscu przycisku na ekranie Rejestracja
 - 2.6. Na zdjęciu wykrywana jest twarz i wyświetlana jest informacja potwierdzająca wykrycie twarzy
3. Użytkownik wypełnia pola tekstowe szablonu rejestracyjnego
 - 3.1. Po kliknięciu w pole „Data urodzenia” zamiast klawiatury wyświetla się kalendarz.
 - 3.2. Użytkownik wybiera datę z kalendarza i potwierdza przyciskiem Ok.
 - 3.3. Data zostaje wpisana w polu „Data urodzenia”.
4. Użytkownik zaznacza pole wyboru „Akceptuję zasady gry”.
5. Użytkownik kliką przycisk „Rejestruj”.
6. Zdjęcie użytkownika jest przesyłane na serwer.
7. W miejscu przycisku „Rejestruj” wyświetlony zostaje pasek postępu przesyłania.
8. Po przesłaniu zdjęcia otrzymywany jest jego identyfikator.
9. Dane z szablonu wraz z identyfikatorem zdjęcia przesyłane są na serwer.
10. Okno rejestracji zamyka się automatycznie po 4 sekundach i aplikacja powraca do ekranu startowego.

Sytuacje wyjątkowe

Ad. 3.3. Wybrana data urodzenia jest w przyszłości. Wyświetlany jest komunikat o niepoprawności daty.

Ad. 8. Przesłanie zdjęcia nie powiodło się. Wyświetlany jest komunikat o błędzie przesyłania zdjęcia. Następuje powrót do kroku 5.

Przebiegi alternatywne

1. Punkty 2., 3. oraz 4. mogą zostać wykonane w dowolnej kolejności.
2. W dowolnym momencie rejestracji użytkownik może wyświetlić zasady gry. Wyświetlenie zasad nie ma wpływu na resztę przebiegu rejestracji. Scenariusz zostaje rozszerzony o kroki:
 1. Użytkownik kliką w przycisk „Zasady gry”.
 2. Wyświetlone zostaje okno z zasadami.
 3. Użytkownik czyta zasady i kliką przycisk „Ok”.
 4. Okno z zasadami zamyka się.

Warunki końcowe

Zarejestrowane zostało nowe konto użytkownika.

2. Logowanie

Aktor

Użytkownik

Zdarzenie inicjujące

Użytkownik będąc na ekranie startowym aplikacji kliknął przycisk „Logowanie”.

Warunki początkowe

Użytkownik posiada konto w aplikacji (jest zarejestrowany).

Opis przebiegu interakcji

1. W aplikacji wyświetlany jest ekran Logowanie.
2. Użytkownik uzupełnia pola „nazwa użytkownika” i „hasło”.
3. Użytkownik kliką przycisk „Zaloguj”.
4. Pola tekstowe i przycisk zastępowane są przez wskaźnik postępu.
5. Dane logowania przesyłane są przez aplikację na serwer.
6. Serwer sprawdza poprawność wprowadzonych danych.
7. Aplikacja otrzymuje token uwierzytelniający użytkownika.
8. Aplikacja przechodzi do widoku Zapis lokalizacji.

Sytuacje wyjątkowe

Ad. 4. Wprowadzone dane są niepoprawne. Wyświetlony zostaje komunikat „Niepoprawne dane logowania”. Następuje powrót do kroku 2.

Przebiegi alternatywne

brak

Warunki końcowe

Użytkownik został zalogowany.

3. Zapis lokalizacji użytkownika

Aktor

Aplikacja

Zdarzenie inicjujące

Użytkownik został zalogowany (przypadek Logowanie do aplikacji mobilnej)

Warunki początkowe

brak

Opis przebiegu interakcji

1. W aplikacji wyświetlony zostaje ekran Zapis lokalizacji
2. Wyświetlony zostaje komunikat „Sprawdzanie uprawnień”
3. Aplikacja sprawdza uprawnienia dostępu do modułu GPS telefonu.
4. Wyświetlony zostaje komunikat „Sprawdzanie ustawień”
5. Aplikacja sprawdza, czy w telefonie włączone są usługi lokalizacyjne.
6. Wyświetlony zostaje komunikat „Oczekiwanie na lokalizację”
7. Aplikacja rozpoczyna nasłuchiwanie aktualizacji lokalizacji.
8. Gdy dokładność lokalizacji osiągnie 25 m informacje o lokalizacji są przesyłane na serwer.
9. Serwer zapisuje dane do bazy.
10. Aplikacja przechodzi do widoku Zlecenia.

Sytuacje wyjątkowe

Ad. 8. Nie udało się ustalić dokładnej lokalizacji przez okres 30 sekund. Wyświetlany jest komunikat „Nie udało się ustalić lokalizacji. Spróbuj ponownie później”.

Przebiegi alternatywne

Ad. 3. Aplikacja nie ma uprawnień do modułu GPS telefonu.

Wyświetlane jest okno dialogowe z prośbą o uprawnienia. Następuje powrót do kroku 3.

Ad. 5. W telefonie wyłączone są usługi lokalizacyjne. Wyświetlane jest okno dialogowe z prośbą o zmianę ustawień. Otwarte zostają ustawienia telefonu. Następuje powrót do kroku 5.

Warunki końcowe

Lokalizacja została zapisana do bazy danych.

Uwagi

Aktualizacje lokalizacji nie są zatrzymywane po wykonaniu zapisu.

4. Potwierdzenie wykonania zlecenia

Aktor

Użytkownik

Zdarzenie inicjujące

Wyświetlenie szczegółów zlecenia

Warunki początkowe

Zlecenie ma status Aktywny

Opis przebiegu interakcji

1. Użytkownik wprowadza kod uzyskany od znalezionejgo użytkownika w odpowiednie pole w widoku Szczegóły zlecenia
2. Użytkownik zatwierdza kod przyciskiem Wyślij
3. Aplikacja pobiera lokalizację użytkownika i razem z kodem przesyła dane na serwer
4. Na serwerze następuje sprawdzenie, czy wprowadzony kod jest poprawny oraz czy lokalizacje generowania i wprowadzenia kodu znajdują się w pobliżu
5. Jeśli kod jest poprawny, zlecenie zostaje zatwierdzone. Aplikacja ponownie pobiera zmienione już dane o zleceniu
6. Widok Szczegółów zlecenia jest aktualizowany nowymi danymi

Sytuacje wyjątkowe

Ad. 3. Aktualna lokalizacja użytkownika nie jest dostępna – przesyłana jest ostatnia lokalizacja logowania użytkownika.

Ad 5. Kod jest niepoprawny – przycisk Wyślij zostaje zablokowany na 10 sekund. W miejscu przycisku pokazuje się odliczanie czasu. W Zleceniu nie następują żadne zmiany.

Warunki końcowe

Zlecenie zostało oznaczone jako wykonane.

5. Generowanie kodu

Aktor

Użytkownik

Zdarzenie inicjujące

Użytkownik wybrał pozycję „Generuj kod” z menu nawigacyjnego aplikacji

Warunki początkowe

brak

Opis przebiegu interakcji

1. W aplikacji wyświetlony zostaje widok Generator kodów.
2. Użytkownik przytrzymuje przycisk z ikoną kostki do gry.
3. W czasie trzymania przycisku wyświetlana jest animacja lądowania wokół przycisku
4. Po zakończeniu lądowania generowany jest losowy 6-cyfrowy ciąg znaków złożony z dużych liter i cyfr.
5. Wygenerowany kod wyświetlany jest w odpowiednim polu w aplikacji.
6. Pobierana jest lokalizacja użytkownika i wraz z wygenerowanym kodem przesyłana jest na serwer i zapisywana w bazie danych.
7. Uruchamiane jest odliczanie czasu ważności kodu.

Sytuacje wyjątkowe

Ad 3. W przypadku puszczenia przycisku przed zakończeniem animacji kod nie jest generowany, animacja jest resetowana do początku. Po kolejnym wcisnięciu przycisku

ładowanie rozpoczyna się od początku.

Warunki końcowe

Na ekranie aplikacji wyświetlony został wygenerowany kod.

6. Edycja profilu

Aktor

Użytkownik

Zdarzenie inicjujące

Użytkownik kliknął przycisk Edycja profilu w widoku Profil użytkownika

Warunki początkowe

brak

Opis przebiegu interakcji

1. Pola z informacjami o użytkowniku w aplikacji stają się edytowalne
2. Przycisk Edycja profilu zamienia się w „Zapisz”.
3. Użytkownik zmienia interesujące go pola
4. Użytkownik zatwierdza zmiany wciskając przycisk Zapisz
5. Zmiany są weryfikowane pod kątem poprawności podobnie jak przy rejestracji.
6. Zmiany zostają przesłane na serwer i zapisane do bazy danych.
7. Pola z informacjami ponownie stają się nieedytowalne

Sytuacje wyjątkowe

Ad 5. Weryfikacja poprawności danych nie powiodła się – błędnie wprowadzone pola zostają zaznaczone kolorem czerwonym. Następuje powrót do kroku 3.

Ad. 6. Nie udało się nawiązać połączenia z serwerem – wyświetlony jest komunikat o błędzie połączenia. Następuje powrót do kroku 4.

Użytkownik opuścił widok Profil użytkownika przed zapisaniem zmian – Wprowadzone zmiany zostają utracone.

Warunki końcowe

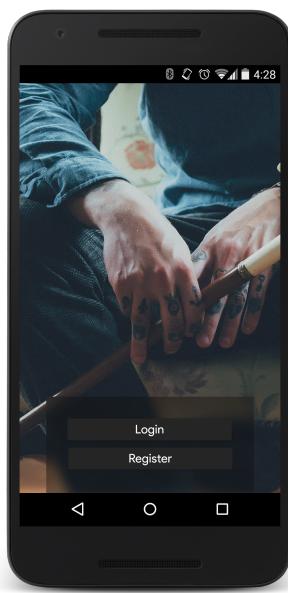
Zmiany w profilu użytkownika zostały zapisane.

4.3. Projekt interfejsu

4.3.1. Aplikacja kliencka

Projekt interfejsu aplikacji mobilnej został wykonany w oparciu o wskazówki *Material design??* opracowane przez Google. Po implementacji niektóre elementy interfejsu mogą różnić się nieznacznie od zaprojektowanych, zależnie od dostępnych

Po uruchomieniu aplikacji wyświetlony zostaje ekran powitalny, z którego użytkownik może przejść do logowania lub rejestracji do gry. Z poziomu ekranu powitalnego jest również możliwość wyświetlenia zasad gry. Po kliknięciu przycisku Login, wyświetlony zostaje ekran logowania widoczny na rysunku 4.5, a po kliknięciu przycisku Register ekran Rejestracja (rysunek 4.4).



Rys. 4.3: Ekran startowy aplikacji

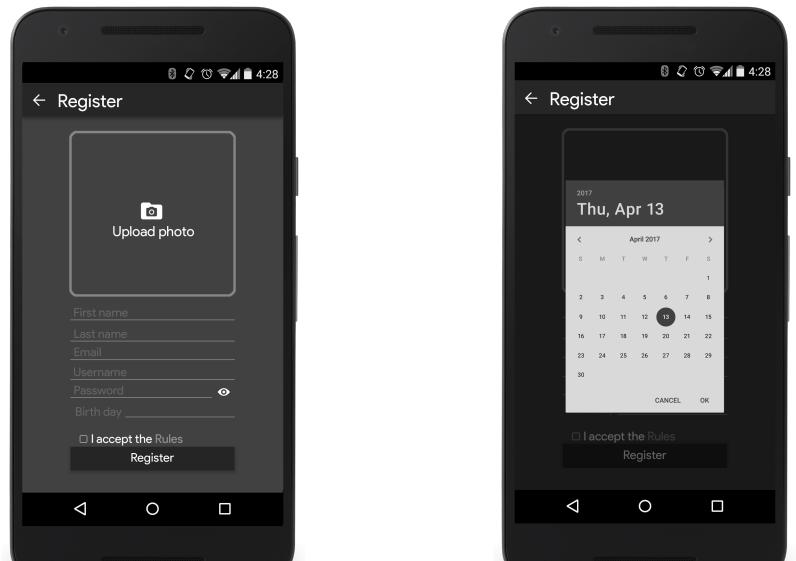
Na ekranie rejestracji wyświetlane są w kolumnie pola tekstowe na informacje o użytkowniku. Pole do wpisania hasła domyślnie nie pokazuje jawnie wpisywanego tekstu, lecz zastępuje go kropkami. Na prawo od pola hasła znajduje się przycisk z ikoną oka, po jego naciśnięciu znaki hasła zostaną pokazane.

U góry ekranu znajduje się miejsce na zdjęcie profilowe użytkownika. Po kliknięciu wewnątrz ramki użytkownik zostanie przekierowany do odpowiedniej dla jego urządzenia aplikacji galerii w celu wyboru zdjęcia. Po wyborze aplikacja poprosi użytkownika o przyjęcie wybranego zdjęcia do interesujących rozmiarów. Zdjęcia profilowe muszą być kwadratowe. Po zakończeniu wybrane zdjęcie zostanie wyświetlone wewnątrz ramki.

Po kliknięciu przez użytkownika w pole *data urodzenia* zamiast standardowej klawiatury wyświetlone zostanie okno dialogowe z kalendarzem widoczne na rysunku 4.4b.

Na górnej krawędzi ekranu znajduje się pasek nawigacyjny pozwalający na powrót do ekranu startowego aplikacji.

Ekran logowania zawiera dwa podstawowe pola tekstowe, nazwa użytkownika oraz hasło oraz przycisk logowania. Pod przyciskiem znajduje się pole wyboru pozwalające zapamiętać wpisane dane. Po naciśnięciu przycisku formularz logowania znika, a w jego miejscu pojawia się wskaźnik ładowania. W przypadku błędnych danych wyświetlony zostaje komunikat o błędzie.



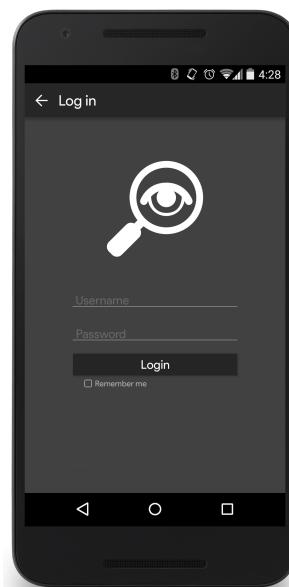
(a) Formularz rejestracji

(b) Okno dialogowe kalendarza

Rys. 4.4: Ekran rejestracji

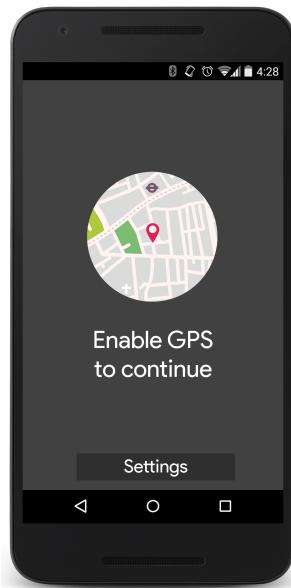
Po poprawnym logowaniu w aplikacji zostaje wyświetlony ekran Zapis lokalizacji widoczny na rysunku 4.6.

Ponownie umieszczono na górnej krawędzi ekranu pasek nawigacyjny pozwalający na powrót do ekranu startowego aplikacji.



Rys. 4.5: Ekran logowania

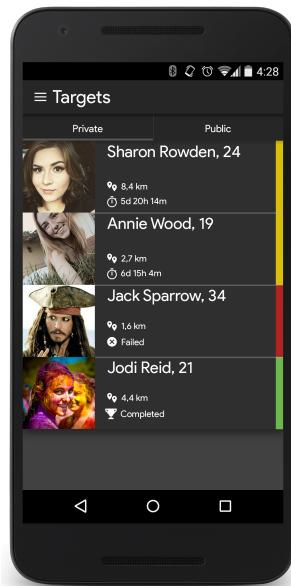
Na ekranie Zapis lokalizacji pod umieszczoną grafiką wyświetlany jest komunikat o aktualnie wykonywanym zadaniu lub informacja dla użytkownika o dodatkowych czynnościach, które należy wykonać np. włączenie GPS. Pod komunikatem znajduje się przycisk pozwalający na otwarcie ustawień systemowych telefonu, w przypadku gdy użytkownik zamknął okno dialogowe informujące o konieczności uaktywnienia modułu GPS telefonu.



Rys. 4.6: Ekran zapisu lokalizacji po logowaniu

Widok listy zleceń użytkownika wyświetlany po zakończeniu zapisu lokalizacji podzielony jest na dwie jednakowe sekcje: prywatne i publiczne. W każdej z sekcji znajduje się lista zleceń danego typu. Elementy listy zawierają informacje o użytkowniku będącym celem zlecenia oraz jego status. Po kliknięciu elementu listy otwarty zostaje widok Szczegóły zlecenia widoczny na rysku 4.8.

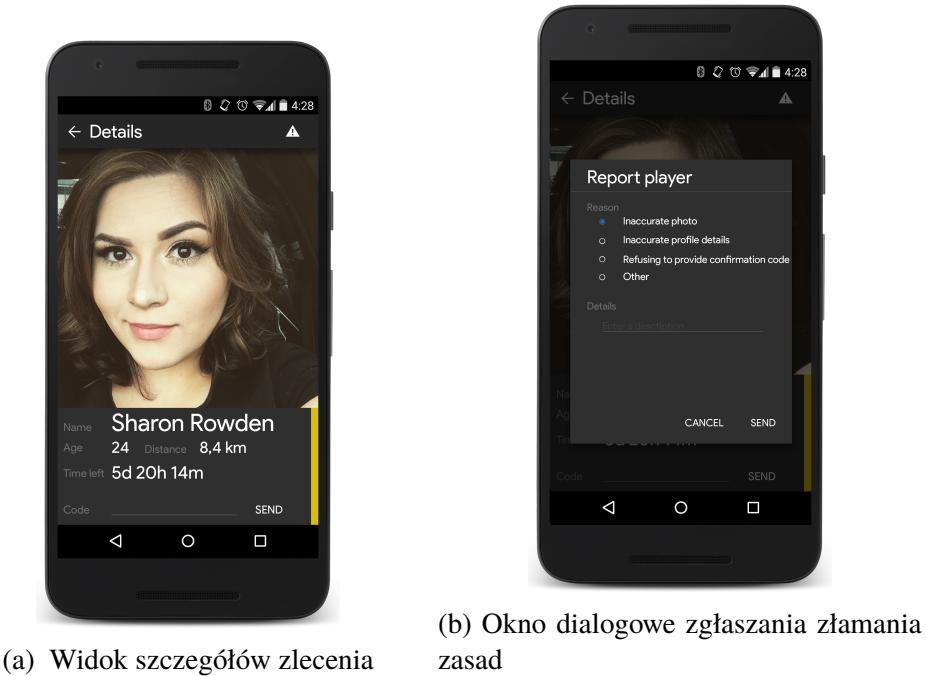
Na górnym pasku nawigacyjnym znajduje się przycisk pozwalający na pokazanie bocznego panelu nawigacyjnego. Panel nawigacyjny może też zostać przywołany poprzez przeciągnięcie w prawo po ekranie. Wysunięty panel nawigacyjny widoczny jest na rysunku 4.10.



Rys. 4.7: Lista zleceń

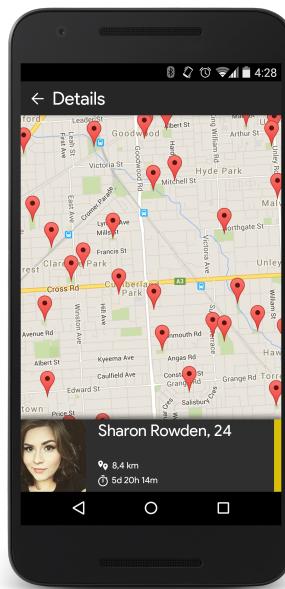
Główym elementem widoku szczegółów zlecenia zajmującym znaczącą część ekranu jest zdjęcie profilowe użytkownika. Pod nim znajdują się dane osobowe szukanego gracza, a także informacje o statusie zlecenia. Na dole ekranu znajduje się formularz do wprowadzenia kodu

potwierdzającego wykonanie zlecenia. Górný pasek aplikacji zawiera przycisk powroto do listy zleceń oraz przycisk do zgłoszania złamania przez gracza zasad gry. Po naciśnięciu tego przycisku w aplikacji wyświetcone zostanie okno dialogowe widoczne na rysunku 4.8 b).



Rys. 4.8: Ekran rejestracji

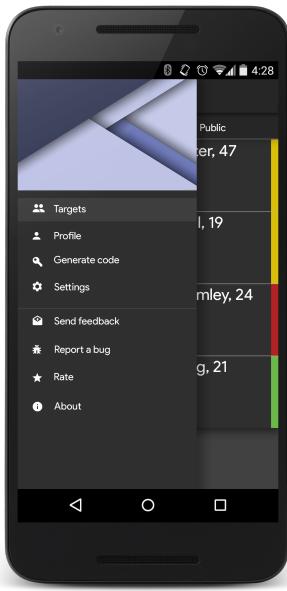
Po wykonaniu gestu przeciągnięcia w dół na ekranie szczegółów zlecenia wyświetlona zostaje mapa z naniesonymi znacznikami oznaczającymi znane lokalizacje szukanego gracza. Po kliknięciu na znacznik pokazana zostanie data i godzina zarejestrowania tej lokalizacji.



Rys. 4.9: Mapa znanych lokalizacji użytkownika

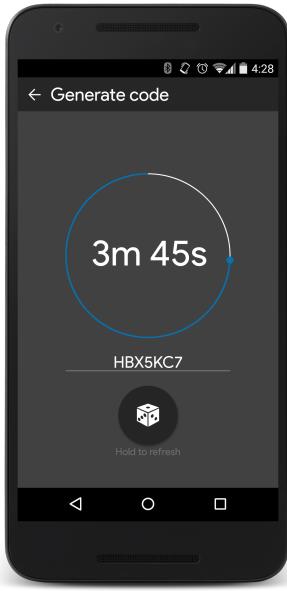
Boczne menu nawygacyjne przywoływanie poprzez naciśnięcie przycisku nawygacyjnego lub przeciągnięcie w prawo na ekranie listy zleceń pozwala na przełączanie pomiędzy kolejnymi

ekranami aplikacji. U dołu menu nawigacyjnego znajdują się również przyciski pozwalające na kontakt z twórcą aplikacji oraz otwarcie strony aplikacji w sklepie Google Play.



Rys. 4.10: Panel nawigacyjny aplikacji

Na ekran generowania kodu potwierdzającego składa się zegar wraz z okrągłym paskiem postępu, pole tekstowe oraz przycisk. W celu wygenerowania kodu użytkownik musi przytrzymać przycisk aż do naładowania do pełna wskaźnika, który pojawi się wokół przycisku. Po naładowaniu generowany kod pojawi się w polu tekstowym, a zegar zacznie odliczać czas do końca ważności kodu.



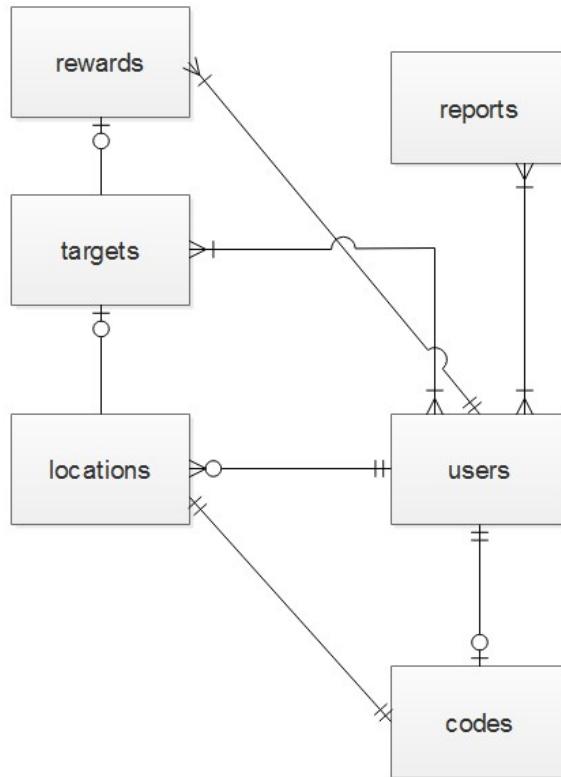
Rys. 4.11: Ekran Generowanie kodu

4.4. Projekt bazy danych

W tym rozdziale przedstawiony został model konceptualny oraz fizyczny bazy danych wykorzystywanej w systemie.

4.4.1. Model konceptualny

Poniżej zaprezentowano diagram związków encji w bazie danych prezentujący relacje pomiędzy poszczególnymi tabelami.



Rys. 4.12: Model konceptualny bazy danych

Podstawową tabelą w bazie jest tabela **users** przechowującą konta użytkowników. Każdy z użytkowników może posiadać zero lub więcej zleceń (**targets**). Zlecenie może być przypisane jednemu użytkownikowi w przypadku zleceń prywatnych, lub nieprzypisane w przypadku zleceń publicznych. Każde zlecenie musi posiadać dokładnie jednego użytkownika będącego Szukanym. Zlecenia mogą mieć przypisaną lokalizację, w której zostały wykonane.

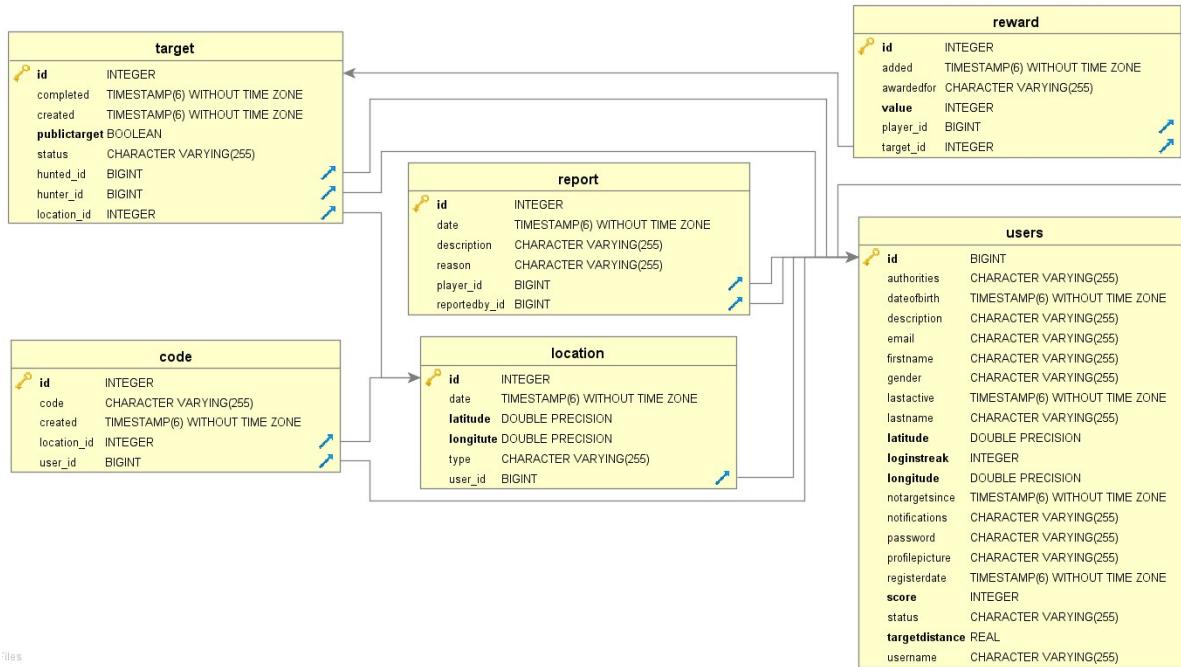
Kod generowany w aplikacji musi być przypisany do użytkownika generującego go. Użytkownik może nie posiadać wygenerowanych kodów, lub posiadać dokładnie jeden, jednak to ograniczenie nie zostało uwzględnione w relacjach encji z założeniem, że zostanie wprowadzone na etapie implementacji.

Kod musi posiadać lokalizację, w której został wygenerowany. Lokalizacja musi być przypisana do dokładnie jednego użytkownika. Użytkownik może posiadać wiele lokalizacji.

Zgłoszenia (**reports**) o złamaniu zasad muszą należeć do jednego użytkownika zgłoszonego, oraz mieć jednego użytkownika zgłaszającego.

Punkty (**rewards**) otrzymywane są za wykonywanie konkrentnego zlecenia i przyznawane jednemu użytkownikowi. Użytkownik może posiadać zero lub wiele przyznanych punktów.

4.4.2. Model fizyczny



Rys. 4.13: Model fizyczny bazy danych

Tabela users zawiera istotne informacje personalne użytkownika, takie jak imię, nazwisko, datę urodzenia, a także wybrane przez użytkownika dane logowania. Tabela została rozszerzona o dane statystyczne o użytkowniku, m.in. datę ostatniego logowania, datę rejestracji. Pole authorities określa rolę użytkownika (gracz, administrator) i jest wykorzystywane w metodach autoryzacji, co zostanie omówione w późniejszych rozdziałach.

Tabela location przechowuje poza kluczami głównym i pochodnym użytkownika informacje lokalizacyjne: długość i szerokość geograficzną oraz datę utworzenia i typ wpisu (np. logowanie).

Tabela target łączy dwóch użytkowników w parę. Jeden z nich staje się szukającym, a drugi szukanym. W tabeli umieszczone zostały informacje o dacie utworzenia i ukończenia zlecenia, widoczności dla użytkowników systemu oraz statusie zlecenia.

Tabela code zawiera ciąg znaków będący losowo generowanym hasłem potwierdzającym zlecenie, datę utworzenia oraz referencje do użytkownika będącego właścicielem kodu i lokalizacji wygenerowania.

W tabeli report przechowywane są zgłoszenia o złamaniu zasad składające się z powodu zgłoszenia (wybieranego w aplikacji z zamkniętej listy dostępnych możliwości), daty i nieobowiązkowego opisu dodatkowego. Zgłoszenie przypisywane jest dwóm użytkownikom, zgłoszonemu i zgłaszającemu.

Tabela reward przechowuje informacje o zdobywanych przez użytkowników punktach. Wpis składa się z wartości punktowej, daty dodania, opisu słownego mówiącego o tym, za co punkty zostały przyznane oraz informacji o użytkowniku otrzymującym je.

4.5. Protokół komunikacji w systemie

Korzystając z przygotowanych wymagań funkcjonalnych oraz przypadków użycia aplikacji opracowano protokół komunikacyjny w systemie. Komunikacja pomiędzy serwerem a aplikacjami dostępnymi dla użytkowników systemu będzie odbywać się poprzez zapytania HTTP. Do wymiany danych zastosowany zostanie format JSON (ang. JavaScript Object Notation).

W planowanej koncepcji protokołu komunikacyjnego adresy zostały podzielone na trzy grupy różniące się uprawnieniami dostępu:

- /api/* - adresy dostępne dla graczy
- /admin/* - adresy dostępne dla administratorów
- pozostałe - adresy dostępne dla wszystkich

W celu uzyskania dostępu do /api/* lub /admin/* wymagane jest przesłanie wraz z zapytaniem nagłówka X-Auth-Token zawierającego informacje o uprawnieniach. Szczegóły dotyczące tego zabezpieczenia opisane zostały w sekcji ??.

4.5.1. Ścieżki dostępu API

Wybrane adresy obsługi zapytań prezentuje poniższa lista. Struktury danych wymienione w sekcji Parametry danych lub Zwaracane wartości zostały przedstawione w kolejnym punkcie.

1. Rejestracja konta

POST	/register/						
Rejestruje nowe konto. Dane w szablonie wypełnionym przez użytkownika są uzupełniane o uprawnienia i datę rejestracji, a hasło jest szyfrowane, po czym następuje zapis do bazy.							
PARAMETRY DANYCH							
<table> <thead> <tr> <th>parametr</th> <th>typ</th> <th>opis</th> </tr> </thead> <tbody> <tr> <td>template</td> <td>UserTemplate</td> <td>szablon danych użytkownika</td> </tr> </tbody> </table>		parametr	typ	opis	template	UserTemplate	szablon danych użytkownika
parametr	typ	opis					
template	UserTemplate	szablon danych użytkownika					
ZWRACANE WARTOŚCI:							
<table> <thead> <tr> <th>Code</th> <th>Message</th> <th>Content</th> </tr> </thead> <tbody> <tr> <td>200</td> <td>Ok</td> <td>User</td> </tr> </tbody> </table>		Code	Message	Content	200	Ok	User
Code	Message	Content					
200	Ok	User					

2. Logowanie

POST

/auth/

Autentykuje użytkownika na podstawie podanych danych logowania i zwraca token dostępu do API systemu.

PARAMETRY DANYCH

parametr	typ	opis
name	AuthenticationRequest	

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	AuthenticationResponse

3. Sprawdzenie dostępności nazwy użytkownika lub adresu email

GET

/register/check?username={name}

Zwraca informacje o istnieniu konta użytkownika o zadanej nazwie.

PARAMETRY ADRESU

parametr	typ	opis
name	String	nazwa użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	boolean
true	nazwa użytkownika jest wolna	
false	nazwa użytkownika jest zajęta	

4. Pobranie konta użytkownika

GET /api/user?user={id}

Zwraca dane o koncie użytkownika o zadanych numerze id.

PARAMETRY ADRESU

parametr	typ	opis
id	long	numer identyfikacyjny użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	User
404		Not found

5. Pobranie zleceń użytkownika

GET /api/target?user={id}

Zwraca zlecenia użytkownika o zadanych numerze id.

PARAMETRY ADRESU

parametr	typ	opis
id	long	numer identyfikacyjny użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	List<Target>
404		Not found

6. Pobranie lokalizacji użytkownika

GET /api/location?user={id}&date={date}

Zwraca zlecenia użytkownika o zadanym numerze id.

PARAMETRY ADRESU

parametr	typ	opis
id	long	numer identyfikacyjny użytkownika
date	timestamp	data utworzenia, od której mają zostać pobrane rekordy

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	List<Location>
400	Bad request	"User not found!"

7. Zapis lokalizacji użytkownika

POST /api/location

Zapis lokalizacji użytkownika do bazy.

PARAMETRY DANYCH

parametr	typ	opis
location	Location	lokalizacja użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	„Location save success”
400	Bad request	Exception stack trace

8. Zapis kodu

POST /api/code

Zapis kodu do bazy danych.

PARAMETRY DANYCH

parametr	typ	opis
code	Code	kod potwierdzający zlecenie

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	

9. Potwierdzenie wykonania zlecenia

PATCH /api/target/confirm?target={id}&code={code}

Potwierdza wykonanie zlecenia. Kod przesłany jako parametr jest porównywany z wyszukiwanym w bazie na podstawie użytkownika będącego celem zlecenia.

PARAMETRY DANYCH

parametr	typ	opis
id	long	numer identyfikacyjny zlecenia
code	String	sześcioliterowy kod potwierdzający wygenerowany w aplikacji

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	Target
400	Bad request	

10. Pobranie najbardziej aktualnej lokalizacji

GET**/api/location/last?user={id}**

Zwraca najbardziej aktualną znaną lokalizację użytkownika o zadanym numerze id.

PARAMETRY ADRESU

parametr	typ	opis
user	long	numer identyfikacyjny użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	Location
400	Bad request	"User not found!"

11. Edycja profilu użytkownika

PATCH**/api/user**

Aktualizuje informacje zawarte w koncie użytkownika na podstawie przesłanego szablonu.

PARAMETRY DANYCH

parametr	typ	opis
template	UserTemplate	szablon z danymi użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	User
400	Bad request	

12. Zgłoszenie złamania zasad

POST /api/report

Zapis zgłoszenia o łamaniu zasad do bazy.

PARAMETRY DANYCH:

parametr	typ	opis
code	Report	

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	

13. Odmrożenie lub zamrożenie konta

PATCH /api/user/status?user={id}&status={status}

Zmiana statusu konta na aktywne lub zawieszone.

PARAMETRY ADRESU:

parametr	typ	opis
user	long	numer identyfikacyjny użytkownika
status	String	active lub suspended

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	User
400	Bad request	

14. Usunięcie konta użytkownika

DELETE /api/user?user={id}

Usuwa konto użytkownika i powiązane z nim informacje. Konto może zostać usunięte tylko przez jego właściciela lub administratora.

PARAMETRY ADRESU:

parametr	typ	opis
user	long	numer identyfikacyjny użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	

15. Zmiana hasła

GET /auth/password?username=username

Wysyła prośbę o przesłanie linku do zmiany hasła na adres email użytkownika o podanej nazwie.

PARAMETRY ADRESU

parametr	typ	opis
username	String	nazwa użytkownika

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	

16. Zmiana parametru działania systemu

PATCH /admin/param?key={key}&value={value}

Aktualizuje wybrany parametr zadaną wartością.

PARAMETRY ADRESU

parametr	typ	opis
key	String	nazwa parametru
value	String	nowa wartość

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	

17. Odczyt wartości parametru działania systemu

GET /admin/param?key={key}

Odczytuje wartość zadanego parametru.

PARAMETRY ADRESU

parametr	typ	opis
key	String	nazwa parametru

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	String
400	Bad request	

18. Odblokowanie lub zablokowanie konta

PATCH /admin/user?user={id}&status={status}

Zmienia status konta użytkownika na zablokowany lub aktywny. Dostępne tylko dla administracji.

PARAMETRY ADRESU

parametr	typ	opis
user	long	numer identyfikacyjny użytkownika
status	String	active lub banned

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	User
400	Bad request	

19. Pobranie listy użytkowników

GET /admin/user?status={status}&role={role}

Pobiera listę użytkowników nakładając zdefiniowane filtry.

PARAMETRY ADRESU

parametr	typ	opis
status	String	active, suspended lub banned
role	String	user lub admin

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	List<User>
400	Bad request	

20. Pobranie listy zgłoszeń

GET /admin/report?user={id}&role={role}

Pobiera listę zgłoszeń o łamaniu zasad nakładając zdefiniowane filtry.

PARAMETRY ADRESU

parametr	typ	opis
user	long	numer identyfikacyjny użytkownika
role	String	user lub admin

ZWRACANE WARTOŚCI:

Code	Message	Content
200	Ok	List<Report>
400	Bad request	

21. Nadanie lub odebranie uprawnień administracyjnych

PATCH	<code>/admin/report?by={id}&reported={id2}&reason={reason}</code>			
Nadaje lub obiera uprawnienia administracyjne zadanemu kontu.				
PARAMETRY ADRESU				
parametr	typ	opis		
reported	long	numer identyfikacyjny zgłoszonego użytkownika		
by	long	numer identyfikacyjny zgłaszającego użytkownika		
reason	String	powód zgłoszenia		
ZWRACANE WARTOŚCI:				
Code	Message	Content		
200	Ok	User		
400	Bad request			

4.5.2. Struktury danych

Poniższy spis przedstawia struktury danych w formacie JSON przesyłane podczas komunikacji pomiędzy serwerem a aplikacjami klienckimi. W celu uproszczenia opisu struktury zagnieżdzone zostały zastąpione ich nazwą, np. `<User>`.

1. AuthenticationRequest

```
{
    "username": "string",
    "password": "string"
}
```

2. AuthentiactionResponse

```
{
    "token": "string",
    "user": <User>
}
```

3. Location

```
{
    "date": "2016-11-20T22:03:02.155Z",
    "id": 0,
    "latitude": 0,
    "longitude": 0,
    "type": "string",
    "user": <User>
}
```

4. Target

```
{
    "completed": "2016-11-20T22:03:02.153Z",
    "created": "2016-11-20T22:03:02.153Z",
    "hunted": <User>,
    "hunter": <User>,
    "id": 0,
    "location": <Location>,
```

```

    "publicTarget": true,
    "status": "string"
}
```

5. Report

```

{
    "date": "2016-11-20T22:03:02.153Z",
    "reported_by": <User>,
    "id": 0,
    "player_id": <User>,
    "description": "string",
    "reason": "string"
}
```

6. Reward

```

{
    "added": "2016-11-20T22:03:02.153Z",
    "awardedfor": "string",
    "value": 0,
    "player_id": <User>,
    "id": 0,
    "taget_id": <Target>
}
```

7. User

```

{
    "authorities": "string",
    "dateOfBirth": "2016-11-19T16:48:15.682Z",
    "description": "string",
    "email": "string",
    "firstName": "string",
    "gender": "string",
    "id": 0,
    "lastActive": "2016-11-19T16:48:15.682Z",
    "lastName": "string",
    "latitude": 0,
    "loginstreak": 0,
    "longitude": 0,
    "noTargetSince": "2016-11-19T16:48:15.683Z",
    "notifications": "string",
    "password": "string",
    "profilePicture": "string",
    "registerDate": "2016-11-19T16:48:15.683Z",
    "score": 0,
    "status": "string",
    "targetDistance": 0,
    "username": "string"
}
```

8. UserTemplate

```

{
    "dateOfBirth": "2016-11-20T21:10:21.927Z",
    "description": "string",
    "email": "string",
    "firstName": "string",
    "gender": "string",
    "lastName": "string",
    "notifications": "string",
    "password": "string",
    "profilePicture": "string",
    "targetDistance": 0,
    "username": "string"
}
```

Rozdział 5

Implementacja systemu

W tym rozdziale opisana została implementacja wybranych elementów zaprojektowanego systemu.

5.1. Aplikacja serwerowa

5.1.1. Wykorzystywane technologie i narzędzia

Podczas realizacji systemu wykorzystano następujące technologie:

1. Java 1.8
2. Spring Framework?? - jest to szkielet tworzenia aplikacji internetowych w języku Javy dla platformy Enterprise Edition. Obecnie jest jedną z najbardziej popularnych technologii na rynku ze względu na wsparcie twórców i społeczności oraz duże możliwości konfiguracyjne, pozwala łatwo dostosować go do własnych potrzeb
3. Spring Boot - rozszerzenie framework'a Spring pozwalające na uruchomienie aplikacji jak wykonywalnego programu Javy
4. Spring Security - rozszerzenie skupiające się na dostarczeniu funkcjonalności autentykowania i autoryzacji użytkowników w aplikacjach Spring
5. Hibernate - narzędzie do realizacji warstwy dostępu do danych, pozwala na przenoszenie danych pomiędzy relacyjną bazą danych a obiektami Javy
6. Swagger UI - narzędzie generujące dokumentację API dostępną pod wybranym adresem URL wewnętrz aplikacji internetowej
7. JSON Web Token - metoda reprezentowania deklaracji uprawnień użytkowników pozwalająca na bezpieczne przesyłanie informacji pomiędzy dwoma stronami
8. Maven
9. projekt Cerberus - w ramach tego projektu zaprezentowano implementację systemu bezstanowej autoryzacji użytkowników z wykorzystaniem Spring Security oraz JSON Web Token.

5.1.2. Struktura plików projektu

Projekt został wykonany w środowisku IntelliJ IDEA w wersji 2016.2 Ultimate. Pliki klas projektu podzielone zostały na paczki zgodnie z ich zastosowaniem:

- config - zawiera pliki konfiguracyjne frameworka Spring, oznaczone adnotacją `@Configuration`
- constant - zawiera klasy definiujące stałe używane w projekcie
- controller - zawiera klasy kontrolerów, które obsługują zapytania http wysyłane na przypisane im adresy URL i poprzez klasy service zwracają użytkownikowi dane z bazy danych
- mapper - zawiera klasy pozwalające na konwersje pomiędzy klasami modelu
- model - zawiera definicje obiektów reprezentujących rzeczywiste dane, na których operuje system
- repository - zawiera interfejsy dostępu do bazy danych (ang. DAO - database access object), w których zdefiniowane są zapytania wywoływanie przez aplikacje
- security - pochodzące z projektu Cerberus klasy wspomagające obsługę autoryzacji przy pomocy tokena
- service - zawiera klasy przetwarzające dane pobrane z bazy danych przez DAO przed przekazaniem ich do kontrolera
- task - zawiera klasy automatyzujące zadania wykonywane przez aplikacje poprzez okresowe wywoływanie metod

Oprócz plików z kodem źródłowym programu ważnym plikiem jest `pom.xml` pozwalający na dołączanie do projektu zewnętrznych bibliotek pobieranych z odpowiednich serwisów, np. Maven Repository??.

5.1.3. Zabezpieczenie dostępu do API

Jako podstawę implementacji serwera wykorzystano projekt demonstracyjny Cerberus?? dostępny na repozytorium GitHub na licencji MIT?? . Konfigurację zabezpieczeń tworzy się poprzez utworzenie klasy rozszerzającej `WebSecurityConfigurerAdapter`. W tej klasie nadpisuje się metodę `configure()`, a w niej tworzy politykę zabezpieczeń dostępu do API. Na listingu widocznym poniżej przedstawiono fragment konfiguracji, która określa trzy poziomy dostępu dla różnych adresów URL w zależności od zadeklarowanej grupy użytkownika:

- USER
- ADMIN
- pozostałe

Nad deklaracją klasy widnieje adnotacja `@Configuration` wskazująca na to, że metody klasy mogą być wykorzystane przez Spring do tworzenia obiektów podczas uruchomienia programu.

Listing 5.1: Konfiguracja Spring Security

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
            .csrf()
            .disable()
            .exceptionHandling()
            .authenticationEntryPoint(this.unauthorizedHandler)
            .and()
```

```

        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .authorizeRequests()
        .antMatchers(HttpMethod.OPTIONS, "/**").permitAll()
        .antMatchers("/api/**").hasRole("USER")
        .antMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().permitAll();

    // Custom JWT based authentication
    httpSecurity
        .addFilterBefore(authenticationTokenFilterBean(),
                         UsernamePasswordAuthenticationFilter.class);
}

```

W celu uaktywnienia filtrowania zapytań, przed każdym zapytaniem wywoływany jest filtr sprawdzający obecność tokena w nagłówku X-Auth-Token zapytania. Z uzyskanego tokena odczytywane są dane użytkownika, które następnie przekazywane są do filtra autoryzującego zapytania. W przypadku braku obecności poprawnego tokena użytkownik zostanie przekierowany na adres /error/.

```

public class AuthenticationTokenFilter
    extends UsernamePasswordAuthenticationFilter {

    @Value("X-Auth-Token")
    private String tokenHeader;

    @Autowired
    private TokenUtils tokenUtils;

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {

        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String authToken = httpRequest.getHeader(this.tokenHeader);
        String username = this.tokenUtils.getUsernameFromToken(authToken);

        if (username != null && SecurityContextHolder
            .getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService
                .loadUserByUsername(username);
            if (this.tokenUtils.validateToken(authToken, userDetails)) {
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(userDetails, null,
                        userDetails.getAuthorities());
                authentication.setDetails(new WebAuthenticationDetailsSource()
                    .buildDetails(httpRequest));
                SecurityContextHolder.getContext()
                    .setAuthentication(authentication);
            }
        }
        chain.doFilter(request, response);
    }
}

```

```
}
```

5.1.4. Baza danych

Jako technologię wykonania bazy danych wybrano PostgreSQL. Połączenie z bazą danych realizowane jest przy pomocy sterownika JDBC. Parametry połączenia i konfiguracja bazy danych określa klasa DatabaseConfig przedstawiona na listingu 5.2. W konfiguracji definiowane jest obiekt dataSource, w którym zawarty jest adres bazy danych oraz dane logowania, a także dodatkowe ustawienia powiązane z frameworkm Hibernate, w szczególności określana jest platforma bazy danych. W konfiguracji obiektu EntityManagerFactory odpowiedzialnego za tworzenie połączeń z bazą danych przekazywane są ustalone wcześniej parametry połączenia, a także określana jest ścieżka do modelu danych.

Listing 5.2: Konfiguracja bazy danych

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories("adamzimny.repository")
public class DatabaseConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.postgresql.Driver");
        dataSource.setUrl(
            "jdbc:postgresql://<database-url>");
        dataSource.setUsername("xxx");
        dataSource.setPassword("xxx");
        return dataSource;
    }

    @Bean
    public Properties jpaProperties() {
        Properties properties = new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "update");
        properties.setProperty("hibernate.enable_lazy_load_no_trans", "true");
        return properties;
    }

    @Bean
    public HibernateJpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter hibernateJpaVendorAdapter
            = new HibernateJpaVendorAdapter();
        hibernateJpaVendorAdapter.setShowSql(false);
        hibernateJpaVendorAdapter
            .setDatabasePlatform("org.hibernate.dialect.PostgreSQLDialect");
        return hibernateJpaVendorAdapter;
    }

    @Bean
    public EntityManagerFactory entityManagerFactory() {
        HibernateJpaVendorAdapter vendorAdapter
            = new HibernateJpaVendorAdapter();
        vendorAdapter.setGenerateDdl(true);
```

```

        LocalContainerEntityManagerFactoryBean factory
            = new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("adamzimny.model");
        factory.setJpaVendorAdapter(jpaVendorAdapter());
        factory.setDataSource(dataSource());
        factory.setJpaProperties(jpaProperites());
        factory.afterPropertiesSet();
        return factory.getObject();
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory());
        return txManager;
    }
}

```

Model fizyczny bazy danych generowany jest na podstawie klas znajdujących się w pakiecie `model`. Podczas uruchomienia aplikacji baza danych obecna na serwerze jest skanowana pod kątem różnic z modelem. Jeżeli zostaną one wykryte, wykonywane są modyfikacje prowadzące do przeniesienia zmian z modelu na bazę danych. Dużą zaletą tego rozwiązania jest brak konieczności prowadzenia ręcznych modyfikacji w bazie danych.

Tworzenie tabel w bazie danych zarządzane jest przez dodanie odpowiednich adnotacji w klasach modelu. Przykładowa klasa przedstawiona została na listingu 5.3. Adnotacja `@Entity` mówi o tym, że należy utworzyć z tej klasy tabelę w bazie danych. Pole adnotowane `@Id` staje się kluczem głównym tabeli, którego wartości będą generowane automatycznie dzięki adnotacji `@GeneratedValue`. Klucze obce tabeli oznaczane są jedną z adnotacji `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`, zależnie od typu relacji pomiędzy encjami.

Listing 5.3: Przykładowa klasa modelu

```

@Entity
public class Target {

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    Integer id;
    @OneToOne
    User hunted;
    @OneToOne
    User hunter;
    String status;

    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss Z")
    Date created;

    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss Z")
    Date completed;
    boolean publicTarget = false;

    @OneToOne
    Location location;
}

```

Interfejsy dostępu do bazy danych znajdujące się w pakiecie `repository` rozszerzają klasę `JpaRepository`. Klasa ta dostarcza podstawowych metod operacji na tabelach, takich jak

`save()` czy `findAll()`. Tworzenie dodatkowych zapytań jest możliwe poprzez deklaracje metod interfejsu. Nie ma konieczności pisania zapytań w języku SQL, framework Hibernate dostarcza mechanizm pozwalający na generowanie ich na podstawie nazwy metody. Nazwa powinna rozpoczynać się od słów `findBy`, po których znajdują się nazwy kolumn w tabeli. Możliwe jest również grupowanie, zliczanie i sortowanie wyników według tej samej konwencji nazewnictwa

Listing 5.4: Przykładowy interfejs DAO

```
@Repository
@Transactional
public interface LocationDAO extends JpaRepository<Location, Integer> {

    List<Location> findByUserAndDateAfter(User user, Date date);
    List<Location> findByUserAndType(User user, String type);
    List<Location> findByUser(User user);
    List<Location> findByUserOrderByDateDesc(User user);
}
```

Mechanizm tworzenia zapytań jest jednak ograniczony do jednej tabeli. Jeżeli zapytanie wymaga połączenia rekordów wielu tabel, wymagane jest użycie adnotacji `@Query` nad metodą interfejsu oraz podanie w niej pełnego zapytania w składni SQL. Parametry zapytania należy opisać adnotacją `@Param`.

Listing 5.5: Zapytanie SQL zawierające łączenie tabel

```
@Query("select u from Target t join t.hunter u group
        by u.id having count(t) < :count")
List<User> findByTargetCountLessThan(@Param("count") long count);
```

5.1.5. Komunikacja i przepływ danych

Komunikacja pomiędzy serwerem a aplikacjami mobilnymi realizowana jest przy pomocy zapytań HTTP. Po poprawnym przetworzeniu zapytania wysłanego przez aplikację kliencką, serwer odpowiada danymi zapisywanyimi w formacie JSON.

Aplikacje tworzone przy pomocy framework'a Spring cechują się wspólną architekturą warstwową definiującą przepływ informacji w systemie. Po otrzymaniu zapytania HTTP przez aplikację przekazywane jest ono do odpowiedniego kontrolera. Adresy obsługiwane przez kontroler lub jego metody określa się przy pomocy adnotacji `@RequestMapping`, co pokazane na listingu 5.6.

Listing 5.6: Przykładowy kontroler

```
@EnableAutoConfiguration
@RestController
public class UserController {

    @Autowired
    UserService userService;

    @RequestMapping(value = "api/user/status", method = RequestMethod.PATCH)
    public ResponseEntity<?> changeStatus(@RequestParam("user") long id,
                                             @RequestParam("status") String status) {
        if ("active".equals(status) || "suspended".equals(status)) {
            Optional<User> userOptional = userService.get(id);
            if (userOptional.isPresent()) {
                User user = userOptional.get();
```

```

        user.setStatus(status);
        userService.save(user);
        return ResponseEntity.ok(userOptional.get());
    }
}
return ResponseEntity.badRequest().body(null);
}
}

```

Po otrzymaniu zapytania kontroler może wywołać metody klas typu `service`, które z kolei pobierają dane z bazy danych poprzez metody klas typu `DAO` z pakietu `repository`.

Listing 5.7: Przykładowy serwis aplikacji

```

@Service
public class UserService {
    @Autowired
    UserDAO userDAO;

    public Optional<User> get(long id) {
        return userDAO.findById(id);
    }
}

```

Po zwróceniu danych z bazy w kontrolerze tworzona jest odpowiedź na zapytanie. Zależnie od przebiegu obsługi zapytania, ustawiany jest status odpowiedzi, a w przypadku zapytań poprawnych (ze statusem `200 Ok`) w ciało odpowiedzi wpisywane są pobrane z bazy danych wyniki.

5.1.6. Implementacja obsługi wybranych zapytań

Obsługa zapytań przez serwer jest w wielu przypadkach bardzo zbliżona. W tym dziale przedstawione zostały przykładowe realizacje wybranych wymagań funkcjonalnych.

Rejestracja

Tworzenie nowych obiektów w konwencji REST realizowane jest przez zapytania typu `POST`. Rejestracja nowego konta użytkownika rozpoczyna się od przesłania zapytania na adres `/register/`. Zapytania wysyłane na ten adres należą do trzeciej grupy opisanej w punkcie 4.5 i nie muszą posiadać tokenu.

Listing 5.8: Kontroler rejestracji

```

@RestController
public class RegisterController {

    @Autowired
    UserService userService;

    @RequestMapping(value = "register", method = RequestMethod.POST)
    public ResponseEntity<?> register(@RequestBody UserTemplate userTemplate) {
        User user = UserMapper.map(userTemplate);
        BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        User u = userService.register(user);
        return ResponseEntity.ok().body(u);
    }
}

```

W kontrolerze pobierany jest przesłany parametr `userTemplate`. Przy wykorzystaniu klasy konwertującej z pakietu `mapper` tworzony jest obiekt klasy `User`, która jest docelową klasą zapisywana w bazie danych.

Hasło przesłane przez użytkownika jest szyfrowane, aby nie zostało zapisane jawnym tekstem w bazie danych. Po zaszyfrowaniu hasła obiekt przekazywany jest do serwisu.

Listing 5.9: Metoda rejestracji użytkownika w klasie `UserService`

```
public User register(User user) {
    user.setRegisterDate(new Date());
    user.setStatus("ACTIVE");
    user.setAuthorities("ROLE_USER");

    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
    Date d;
    try {
        d = sdf.parse("01/01/1980");
    } catch (ParseException e) {
        d = new Date();
    }
    user.setNoTargetSince(d);

    userDAO.save(user);
    return user;
}
```

Serwis zajmuje się przygotowaniem obiektu do zapisu w bazie danych. Ustawiana jest data rejestracji i nadawane są uprawnienia użytkownika. Dodatkowo użytkownikowi przypisuje się datę, od której nie posiada on zleceń jako rok 1980. Pozwoli to odnaleźć nowych użytkowników w systemie sortując po dacie i nadać im priorytet w przydzielaniu zleceń.

Po wykonaniu tych operacji obiekt jest zapisywany w bazie danych metodą `save()` klasy `UserDAO` i zwracany do kontrolera. W kontrolerze tworzona jest odpowiedź na zapytanie o statusie **200 Ok** informującym o poprawnym wykonaniu rejestracji, a nowo utworzony obiekt użytkownika jest zwracany jako w odpowiedzi.

Pobranie lokalizacji użytkownika

Jako zapytania zwracające dane użytkownikowi przyjęto zapytania GET. Parametrami zapytania są id użytkownika oraz data, od której należy wyszukać lokalizacje. W sygnaturze metody przed drugim parametrem znajduje się dopisek `required = false`. Oznacza to, że parametr ten jest opcjonalny, czyli zapytanie wysłane na adres `/api/location?user=1` również zostanie obsłużone przez tę metodę.

Rozpoczynając obsługę zapytania kontroler wyszukuje użytkownika o zadanym numerze id w bazie danych. Jeżeli użytkownik nie zostanie znaleziony, tworzona jest odpowiedź o statusie **400 Bad request**. W przypadku znalezienia użytkownika, sprawdzana jest obecność opcjonalnego parametru daty i jeżeli został on podany, uwzględniany jest on w dalszej obsłudze zapytania przez serwis.

Listing 5.10: Kontroler lokalizacji użytkownika

```
@RequestMapping(value = "api/location")
public class LocationController {

    @RequestMapping(value = "api/location", method = RequestMethod.GET)
    public ResponseEntity<?> findByUserAndDateAfter(
```

```

    @RequestParam("user") long id,
    @RequestParam(value = "date",
                  required = false) String dateString)
    throws ParseException {

    Optional<User> user = userService.get(id);
    if (user.isPresent()) {
        if (dateString != null) {
            DateFormat df = new SimpleDateFormat(
                "yyyy-MM-dd HH:mm:ss zzz");
            Date date = df.parse(dateString);

            return ResponseEntity.ok(locationService
                .findByUserAndDateAfter(user.get(), date));
        } else {
            return ResponseEntity.ok(locationService
                .findByUser(user.get()));
        }
    }
    return ResponseEntity.badRequest().body("User not found!");
}
}

```

Kontroler wywołuje jedną z dwóch metod serwisu zależnie od wartości parametru date. W obsłudze tego zapytania nie ma żadnej dodatkowej logiki, która musi zostać wykonana przez serwis, więc jego działanie sprowadza się do pobrania rekordów z bazy danych i ich zwrócenia.

Listing 5.11: Metody pobierania lokalizacji w LocationService

```

public List<Location> findByUserAndDateAfter(User user, Date date) {
    List<Location> list = locationDAO.findByUserAndDateAfter(user, date);
    return list;
}

public List<Location> findByUser(User user) {
    return locationDAO.findByUser(user);
}
}

```

Po wykonaniu opracji na bazie danych tworzona jest odpowiedź na zapytanie o statusie 200 Ok.

Potwierdzenie wykonania zlecenia

Do wprowadzania modyfikacji używa się metody PATCH zapytań HTTP. Parametrami zapytania są numer id zlecenia oraz kod potwierdzający. Po otrzymaniu zapytania na adres /api/target/confirm kontroler wyszukuje zlecenie o zadanym numerze. Jeżeli zlecenie nie zostanie odnalezione kontroler zwraca odpowiedź o statusie 400 Bad request.

Listing 5.12: Fragment kontrolera zleceń

```

@RestController
@RequestMapping(value = "api")
public class TargetController {

    @Autowired
    TargetService targetService;
}

```

```

    @Autowired
    UserService userService;

    @Autowired
    CodeService codeService;

    @RequestMapping(value = "target/confirm", method = RequestMethod.PATCH)
    public ResponseEntity<Target> confirm(
        @RequestParam("target") long target_id,
        @RequestParam("code") String codeString) {
        Optional<Target> target = targetService.findById(target_id);
        if(target.isPresent()){
            Optional<Code> code = codeService.getCodeForUser(
                target.get().getHunted());
            if(code.isPresent() && code.get().getCode().equals(codeString)){
                return ResponseEntity.ok(targetService.confirm(
                    target.get(), code.get()));
            }
        }
        return ResponseEntity.badRequest().body(null);
    }
}

```

Po pobraniu zlecenia z bazy odczytywany jest ostatni kod wygenerowany przez użytkownika będącego Szukanym w zleceniu. Kod przesłany jako parametr zapytania jest porównywany z pobranym kodem. Jeżeli są one jednakowe, kontroler przekazuje obiekt zlecenia do serwisu w celu potwierdzenia.

Serwis ustawia status i datę wykonania zlecenia, a także przypisuje lokalizację, w której zlecenie zostało wykonane na podstawie użytego kodu. Po wykonaniu zapisu do bazy zmodyfikowany obiekt jest zwracany do kontrolera.

Listing 5.13: Metoda serwisu TargetService potwierdzająca wykonanie zlecenia

```

public Target confirm(Target target, Code code) {
    target.setCompleted(new Date());
    target.setStatus("COMPLETED");
    target.setLocation(code.getLocation());
    targetDao.save(target);
    return target;
}

```

Kontroler po otrzymaniu zmodyfikowanego obiektu przesyła go w odpowiedź na zapytanie http ze statusem 200 Ok.

5.1.7. Automatyzacja działania systemu

W celu umożliwienia wykonywania części zadań serwera automatycznie w określonych odstępach czasu wykorzystano funkcjonalność planowania zadań framework Spring. Uaktywnienie funkcjonalności wymaga utworzenia konfiguracji z użyciem anotacji `@EnableScheduling`. W konfiguracji przedstawionej na listingu 5.14 wyspecyfikowano także liczbę wątków zadań, jakie mogą zostać przydzielone. Jeżeli wątków byłoby zbyt mało, zaplanowane zadania byłyby opóźniane aż do momentu ukończenia aktywnych zadań i zwolnienia wątku.

Listing 5.14: Konfiguracja planera zadań

```

@Configuration
@EnableScheduling
public class SchedulerConfig {

    @Bean
    public Executor taskScheduler() {
        return Executors.newScheduledThreadPool(10);
    }
}

```

Do tworzenia zadań służy adnotacja `@Scheduled`. Klasa, w której tworzone są zadania musi posiadać adnotację `@Component`, mówiącą o tym, że jest zarządzana przez Spring i może dzięki temu zostać odnaleziona podczas skanowania klas przy uruchamianiu aplikacji.

Adnotacja `@Scheduled` przyjmuje jeden z dostępnych parametrów ustalających sposób powtarzania zadań:

- `fixedRate` - opóźnienie kolejnego wykonania liczone jest od rozpoczęcia poprzedniego
- `fixedDelay` - opóźnienie kolejnego wykonania liczone jest od zakończenia poprzedniego
- `cron` - czas wykonywania zadania definiowany jest przy pomocy wyrażenia cron

Sposób tworzenia wyrażeń cron przedstawia poniższa tabela.

jednostka czasu	wymagane	dozwolone wartości	znaki specjalne
sekundy	Y	0-59	, - * /
minuty	Y	0-59	, - * /
godziny	Y	0-23	, - * /
dzień miesiąca	Y	1-31	, - * ? / L W
miesiąc	Y	0-11 lub JAN-DEC	, - * /
dzień tygodnia	Y	1-7 or SUN-SAT	, - * ? / L #
rok	N	puste lub 1970-2099	, - * /

Tab. 5.1: Parametry wyrażeń cron

* - każda wartość

? - wartość dowolna

- - zakres

, - wiele wartości

/ - wartość inkrementowana, np. */5 - co 5 minut

L - ostatnia wartość, np. 5L - ostatni piątek miesiąca

W - najbliższy dzień roboczy

- kolejny numer wystąpienia, np. 6#3 - trzeci piątek miesiąca

Na przykład:

0 15 10 * * ? 2005 - uruchom o 10:15 każdego dnia w roku 2005

0 0/5 14 * * ? - uruchom co 5 minut, zaczynając od 14:00, do 14:55 włącznie, każdego dnia

0 15 10 ? * 6L 2002-2005 - uruchom o 10:15, w każdy ostatni piątek miesiąca w latach od 2002 do 2005 włącznie

Dodatkowym parametrem, który można przekazać w adnotacji `@Scheduled` jest `initialDelay` pozwalający określić opóźnienie rozpoczęcia wykonywania zadania po raz pierwszy po uruchomieniu aplikacji.

Przykłady implementacji zadań w projekcie

1. **Generowanie nowych zleceń.** Schemat powtarzania tego zadania określa wyrażenie `cron = "0 0 0/12 * * ?"`, które interpretowane jest jako „co 12 godzin, każdego

dnia, zaczynając o północy". Generowanie zleceń rozpoczyna się od pobrania z bazy danych listy użytkowników, którzy posiadają mniej niż maksymalna dozwolona liczba zleceń prywatnych. Ze względu na złożoność zapytania, konieczne było jego zdefiniowanie ręczne przy pomocy adnotacji @Query, co zaprezentowano na listingu 5.5 w dziale 5.1.4 Baza danych.

Następnie, dla każdego użytkownika tworzona jest lista użytkowników znajdujących się w pobliżu na podstawie zapamiętanej ostatniej ich lokalizacji. Z utworzonej listy losowanych jest tyle użytkowników, aby uzupełnić zlecenia do maksymalnej wartości.

Listing 5.15: Zadanie generujące nowe zlecenia

```
@Scheduled(cron = "0 0 0/12 * * ?")
public void createNewTargets() {
    List<User> hunters = targetService
        .findByTargetCountLessThan(Preferences.MAX_PRIVATE_TARGETS);
    for (User u : hunters) {
        List<User> potentialTargets = userService
            .findNearbyPlayers(u);
        for (int i = 0; i < Preferences.MAX_PRIVATE_TARGETS
            - targetService.countTargetsOf(u); i++) {
            Random r = new Random();
            int id = r.nextInt() % potentialTargets.size();
            targetService.createTarget(u.getId(), id, false);
            potentialTargets.remove(id);
        }
    }
}
```

Sposób implementacji funkcji wyszukiwania graczy został zaprezentowany na poniższym listingu.

Listing 5.16: Pobieranie listy użytkowników z uwzględnieniem odległości

```
public List<User> findNearbyPlayers(User u) {
    List<User> results = new ArrayList<>();
    List<User> activeUsers = userDAO.findByStatus(Users.ACTIVE);
    for (User a : activeUsers) {
        if (a != u && getDistance(a, u)) {
            results.add(a);
        }
    }
    return results;
}

private boolean getDistance(User a, User u) {
    return distance(a.getLatitude(), u.getLatitude(),
        a.getLongitude(), u.getLongitude(), 0, 0)
        <= u.getTargetDistance();
}

public static double distance(double lat1, double lat2, double lon1,
    double lon2, double el1, double el2) {

    final int R = 6371; // Radius of the earth

    Double latDistance = Math.toRadians(lat2 - lat1);
    Double lonDistance = Math.toRadians(lon2 - lon1);
    Double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
```

```

+ Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2))
* Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
Double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
double distance = R * c * 1000; // convert to meters

double height = el1 - el2;

distance = Math.pow(distance, 2) + Math.pow(height, 2);

return Math.sqrt(distance);
}

```

Funkcja obliczająca odległość dwóch punktów na podstawie współrzędnych została za-czerpnięta z serwisu StackOverflow??.

- Zmiana widoczności zleceń z prywatnych na publiczne. Aby możliwe było utrzymanie spójności danych, konieczne jest częste wykonywanie aktualizacji. Z tego powodu to zadanie wywoływane jest co 10 sekund. Podczas wykonywania pobierana jest lista zleceń prywatnych, które przekroczyły czas przydzielone na ich potwierdzenie. Odpowiednie zapytanie zaprezentowano na listingu 5.18. Następnie każde zlecenie jest modyfikowane przez zmianę widoczności na publiczne, nadanie nowej daty utworzenia w celu zrestartowania ważności i usunięcie przypisania do użytkownika Szukającego.

Listing 5.17: Zadanie zmieniające widoczność zleceń

```

@Scheduled(fixedRate = 10 * 1000)
public void makePublicTargets() {
    List<Target> targetList = targetService.findExpired(false);
    for (Target t : targetList) {
        t.setPublicTarget(true);
        t.setCreated(new Date());
        t.setHunter(null);
        targetService.save(t);
    }
}

```

Listing 5.18: Metoda serwisu TargetService pobierająca zlecenia wymagające zmian.

```

public List<Target> findExpired(boolean publi) {
    LocalDateTime daysAgo = LocalDateTime.now()
        .minusDays(Preferences.PRIVATE_TARGET_DAYS);
    return targetDao.findByPublicTargetAndStatusAndCreatedBefore(
        publi, Targets.ACTIVE, Date.from(daysAgo.atZone(
            ZoneId.systemDefault()).toInstant()));
}

```

5.2. Aplikacja kliencka

Ta sekcja opisuje proces implementacji aplikacji mobilnej przeznaczonej dla graczy.

5.2.1. Struktura projektu

Projekt został wykonany w programie Android Studio w wersji 2.1.3. W plikach projektu wyróżnić można dwie zasadnicze grupy: klasy w języku Java oraz pliki zasobów. Pliki klas podzielone zostały w pakiety zgodnie z ich przeznaczeniem:

- activity - zawiera klasy **Activity** widoków dostępnych w aplikacji
- api - klasy dostarczające funkcjonalności komunikacji z serwerem poprzez zapytania http
- constant - wartości stałych wykorzystywanych w projekcie
- fragment - zawiera klasy fragmentów będących zawartością klas **Activity**
- model - zawiera definicje obiektów reprezentujących rzeczywiste dane, na których operuje system
- util - zawiera klasy wspomagające dostarczające funkcjonalności przetwarzania danych poprzez statyczne metody
- view - zawiera niestandardowe klasy komponentów widoku

Struktura plików zasobów jest określona z góry dla wszystkich projektów. Wyróżnić można następujące foldery:

- drawable - obrazy oraz pliki xml elementów widoku
- layout - pliki interfejsu graficznego aplikacji
- menu - lista opcji dostępnych na różnych menu aplikacji
- mipmap - ikony aplikacji
- values - pliki xml określające wartości takie jak kolory, wymiary, style używane w aplikacji

Oprócz opisanych powyżej plików istotną rolę w projekcie pełnią dwa pliki `build.gradle` pozwalające na konfiguracje projektu oraz dołączanie bibliotek.

5.2.2. Biblioteki wykorzystane w projekcie

W celu poszerzenia funkcjonalności dostępnych w projektach Androida i zaoszczędzenia czasu programowania stosuje się biblioteki dołączane poprzez plik `gradle.build`. Biblioteki dołącza się poprzez podanie zależności w sekcji `dependencies{}`.

Poniższa lista przedstawia wszystkie biblioteki wykorzystywane w projekcie. Wybrane z nich zostaną szerzej opisane w dalszej części pracy.

- ButterKnife - biblioteka pozwala na wstrzykiwanie elementów widoków z pliku xml do klas Javy
- FastAdapter - biblioteka ułatwiająca tworzenie list z wykorzystaniem komponentu `RecyclerView`
- AppCompat - biblioteka dostarczana przez Google, zapewniająca kompatybilność ze starszymi wersjami systemu Android
- Design - biblioteka Google dostarczająca komponentów widoków
- Circular Image View - element interfejsu pozwalający na utworzenie okrągłebo obrazu
- Gson - biblioteka pozwalająca na konwersję pomiędzy obiektami Javy a ich reprezentacją w formacie JSON
- Retrofit - klient HTTP dla Androida pozwalający na wygodne tworzenie zapytań HTTP
- Joda - dostarcza funkcjonalności pozwalających na wygodne przetwarzanie dat i czasu z uwzględnieniem stref czasowych
- Universal Image Loader - pozwala na ładowanie obrazów z różnych źródeł w tle, pozwalając na płynną pracę aplikacji bez obciążania interfejsu
- Android Image Cropper - dostarcza funkcjonalność przycinania zdjęć wykorzystywaną podczas rejestracji konta
- Hold To Load Layout - widok w kształcie koła wyświetlający pasek ładowania wokół siebie podczas długiego wcisnięcia
- Okhttp3 Logging Interceptor - rozszerzenie dla biblioteki Retrofit opartej na Okhttp3 pozwalające na wyświetlanie zapytań HTTP w logach aplikacji i na konsoli

- Number Progress Bar - pasek postępu z numeryczną reprezentacją poziomu ukończenia w procentach
- Play Services - usługi dostarczane przez Google, niezbędne do wykorzystania API Google Maps
- Google Maps - obsługa Google Maps
- Circular Progress Bar - pasek postępu w kształcie okręgu, wykorzystywany jako odliczanie czasu

5.2.3. Tworzenie interfejsu aplikacji

Interfejs użytkownika w aplikacjach na systemy Android tworzony jest przy pomocy plików XML znajdujących się w katalogu `res/layout` projektu. Do tworzenia interfejsu wykorzystuje się standardowe komponenty dostarczane przez środowisko Android, jak np. Button, ImageView, EditText itp. Komponenty osadza się kontenerach określających ich układ na ekranie, np. LinearLayout, w którym elementy układane są jeden obok drugiego w określonym kierunku(pionowo lub poziomo). Każdy element widoku posiada właściwości, którymi można zmieniać jego wygląd i zachowanie. Możliwe jest wykorzystanie elementów spoza domyślnego zestawu dostępnego w środowisku, na przykład poprzez dołączenie zewnętrznej biblioteki lub zdefiniowanie własnej klasy. W takim wypadku przy deklaracji elementu w pliku XML należy podać pełną ścieżkę do klasy z uwzględnieniem pakietu w jakim się znajduje. Przykładowy plik definicji widoku wykorzystujący zewnętrzną klasę przedstawiono na listingu 5.19. Elementom interfejsu można nadawać identyfikatory korzystając z parametru `android:id="@+id/<nazwa>"`. Po nadaniu identyfikatora możliwe jest odnalezienie elementu w klasie aktywności.

Listing 5.19: Przykładowy plik widoku aplikacji

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/gray_background">

    <ProgressBar
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="10dp"
        android:id="@+id/progressBar"
        android:layout_gravity="bottom|center_horizontal"
        android:indeterminate="true"/>

    <com.mikhaellopez.circularimageview.CircularImageView
        android:layout_marginTop="60dp"
        android:layout_gravity="center"
        app:civ_border="false"
        app:civ_shadow="false"
        android:layout_width="180dp"
        android:layout_height="180dp"
        android:src="@drawable/no_gps_logo"/>

    <TextView android:layout_width="wrap_content"
        android:layout_gravity="center"
        android:layout_height="wrap_content"
```

```

        android:textSize="38sp"
        android:text="Enable GPS\nto continue"
        android:layout_margin="40dp"
        android:id="@+id/status_text"
        android:gravity="center_horizontal"/>

    <Button android:layout_width="match_parent"
            android:layout_height="@dimen/button_height"
            android:layout_marginLeft="@dimen/button_width_margin"
            android:id="@+id/settings"
            android:layout_marginRight="@dimen/button_width_margin"
            android:background="@color/gray_button"
            android:text="Settings"
            android:textSize="14sp"
            android:textAllCaps="false"/>
</LinearLayout>
```

Widoki w aplikacjach Android posiadają kontrolery, które pozwalają na określanie ich zachowania. Klasami kontrolerów są przeważnie klasy **Activity** lub **Fragment**. W klasach tych następuje powiązanie pomiędzy zdefiniowanymi elementami pliku XML a zmiennymi w kodzie programu, co pozwala na oprogramowanie ich działania.

Biblioteka **ButterKnife** pozwala na wykonywanie opisanych powiązań przy pomocy adnotacji. W tym projekcie wykorzystano dwa rodzaje adnotacji:

- **@BindView** - stosowana do zmiennych klasy, wyszukuje element o zadanym id w pliku XML i przypisuje go do adnotowanej zmiennej
- **@OnClick** - stosowana do metod klasy, powoduje wywołanie adnotowanej metody po kliknięciu w element widoku o określonym id.

W celu uaktywnienia rozszerzenia **ButterKnife** w metodzie **onCreate** kontrolera po wywołaniu metody **setContentView** przypisującej plik z definicją interfejsu do kontrolera konieczne jest utworzenie powiązań metodą **ButterKnife.bind(this);**. Fragment kodu kontrolera prezentujący opisane czynności przedstawia poniższy listing.

Listing 5.20: Tworzenie powiązań interfejsu przy pomocy biblioteki **ButterKnife**

```

public class GpsActivity extends AppCompatActivity
    implements LocationListener {

    android.location.Location lastKnown;
    boolean exitLoop = false;

    @BindView(R.id.settings)
    Button settingsButton;

    @BindView(R.id.status_text)
    TextView statusText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_gps);
        ButterKnife.bind(this);
        AppVariable.locationManager = (LocationManager) getSystemService(
                Context.LOCATION_SERVICE);
        AppVariable.locationListener = this;
        getLocation();
    }
}
```

```

}
```

W danym momencie, w aplikacji aktywny może być tylko jeden widok z jednym kontrolerem. Przełączanie pomiędzy widokami realizowane jest przez intencje (ang. intent). Kontrolery widoków, które były uruchomione poprzednio w aplikacji pozostają uruchomione w tle, dzięki czemu możliwy jest powrót do nich. Możliwe jest zakończenie pracy kontrolera poprzez wywołanie metody `finish()`.

W projekcie aby zredukować ilość powielonego kodu potrzebnego do przełączania aktywności zaimplementowana została klasa `IntentHelper` dostarczająca statycznych metod pozwalających na utworzenie intencji zaledwie jedną linią kodu. Kod klasy zaprezentowano na listingu 5.21.

Listing 5.21: Klasa IntentHelper

```

public class IntentHelper {
    public final static int FILE_PICK = 1001;
    public final static int GPS_SETTINGS = 1002;
    public static final int PERMISSION_GPS = 1003;

    public static void chooseFileIntent(Activity activity){
        Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
        intent.setType("image/*");
        activity.startActivityForResult(intent, FILE_PICK);
    }

    public static void startActivityIntent(Activity source,
                                           Class<? extends Activity> nextActivity){
        Intent i = new Intent(source,nextActivity);
        source.startActivity(i);
    }

    public static void startActivityIntent(Activity source,
                                           Class<? extends Activity> nextActivity, Bundle bundle){
        Intent i = new Intent(source,nextActivity);
        i.putExtras(bundle);
        source.startActivity(i);
    }
}

```

Jako parametry metod zwartych w klasie należy podać kontekst źródłowy, z którego wywoływana jest intencja oraz klasę aktywności, która ma zostać uruchomiona. Możliwe jest również podanie dodatkowych danych do uruchamianej aktywności za pomocą klasy `Bundle`. Wywołanie przejścia przedstawia listing 5.22.

Listing 5.22: Wywołanie metody klasy IntentHelper

```

public void goToTargets() {
    IntentHelper.startActivityIntent(this, TargetActivity.class);
    finish();
}

```

Jeżeli po zakończeniu pracy kontrolera powinny zostać zwrócone jakieś dane, aktywność należy wywołać przy pomocy metody `startActivityForResult`. Wówczas w aktywności źródłowej należy przeciążyć metodę `onActivityResult` pozwalającą na odebranie i przetworzenie zwracanych danych. Przykłady zastosowania tych funkcjonalności zostaną zaprezentowane przy opisie implementacji modułu rejestracji.

W aplikacji zaimplementowane zostały następujące aktywności:

- CodeActivity - widok pozwalający na wygenerowanie kodu potwierdzającego
- DetailsActivity - widok szczegółów zlecenia
- GpsActivity - widok wyświetlany po zalogowaniu użytkownika wykoujący zapis lokalizacji do bazy, drugą jego funkcją jest zablokowanie dostępu do pozostałych funkcji aplikacji
- LoginActivity - widok logowania do konta
- MapActivity - widok mapy z naniesionymi znanimi lokalizacjami użytkownika, wywoływany z poziomu szczegółów zlecenia
- TargetActivity - widok listy zleceń
- RegisterActivity - widok rejestracji konta
- WelcomeActivity - widok startowy aplikacji

Szczegóły dotyczące implementacji kolejnych aktywności opisują kolejne sekcje tego rozdziału.

Aktywności zadeklarowane w aplikacji muszą zostać wpisane do pliku `AndroidManifest.xml`. Plik ten dostarcza niezbędnych informacji o zawartości aplikacji do systemu operacyjnego, takich jak wymagane uprawnienia aplikacji czy wersja systemu operacyjnego. Zwartość tego pliku w projekcie pokazano na listingu 5.23. Oprócz wymienionych powyżej aktywności zadeklarowana została jedna dodatkowa `CropImageActivity` pochodząca z biblioteki ImageCropper.

Listing 5.23: Plik `AndroidManifest.xml`

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

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".activity.LoginActivity"
            android:label="Login"
            android:theme="@style/AppTheme.NoActionBar">
        </activity>
        <activity
            android:name=".activity.TargetActivity"
            android:label="Targets">
        </activity>
        <activity
            android:name=".activity.WelcomeActivity"
            android:label="Welcome"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity android:name=".activity.UglyRegisterActivity">
    
```

```

</activity>
<activity android:name=".activity.DetailsActivity">
</activity>

<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="@string/google_maps_key"/>

<activity
    android:name=".activity.MapActivity"
    android:label="@string/title_activity_map">
</activity>
<activity android:name=".activity.GpsActivity">
</activity>
<activity android:name=".activity.CodeActivity">
</activity>
<activity android:name="com.theartofdev.edmodo.cropper.CropImageActivity"
    android:theme="@style/Base.Theme.AppCompat"/>
</application>

</manifest>

```

W pliku manifest zostały zadeklarowane uprawnienia aplikacji:

- INTERNET - pozwala na dostęp do internetu
- ACCESS_FINE_LOCATION - pozwala na pozyskanie lokalizacji przy pomocy modułu GPS
- ACCESS_COARSE_LOCATION - pozwala na pozyskanie lokalizacji przy pomocy modułu GSM lub Wifi
- READ_EXTERNAL_STORAGE - pozwala na odczyt plików z zewnętrznej karty pamięci

W przypadku braku wymienionych uprawnień przy próbie dostępu do zabezpieczonych zasobów aplikacja zgłosiłaby wyjątek `SecurityException`.

Przełączanie pomiędzy wybranymi widokami aplikacji odbywa się za pomocą bocznego panelu nawigacyjnego. Za jego utworzenie odpowiedzialny jest plik interfejsu `global.xml`, którego zawartość pokazano na listingu 5.25. W komponencie `NavigationView` umieszczono parametr `app:menu` definiujący zawartość panelu.

Listing 5.24: Panel nawygacyjny aplikacji

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include
        layout="@layout/activity_target"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"

```

```

        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="false"
        app:headerLayout="@layout/nav_drawer_banner"
        app:menu="@menu/activity_target_drawer"
        android:background="@color/gray_item"
        app:itemTextColor="@color/white" app:itemIconTint="@color/white"/>

    </android.support.v4.widget.DrawerLayout>

```

Listing 5.25: Lista opcji dostępna na panelu nawigacyjnym

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_targets"
            android:icon="@drawable/ic_menu_camera"
            android:title="Targets" />
        <item
            android:id="@+id/nav_profile"
            android:icon="@drawable/ic_menu_gallery"
            android:title="Profile" />
        <item
            android:id="@+id/nav_code"
            android:icon="@drawable/ic_menu_slideshow"
            android:title="Generate code" />
        <item
            android:id="@+id/nav_settings"
            android:icon="@drawable/ic_menu_manage"
            android:title="Settings" />
    </group>

    <item android:title="Developer">
        <menu>
            <item
                android:id="@+id/nav_feedback"
                android:icon="@drawable/ic_menu_share"
                android:title="Send feedback" />
            <item
                android:id="@+id/nav_bug"
                android:icon="@drawable/ic_menu_send"
                android:title="Report a bug" />
            <item
                android:id="@+id/nav_rate"
                android:icon="@drawable/ic_menu_send"
                android:title="Rate" />
            <item
                android:id="@+id/nav_about"
                android:icon="@android:drawable/ic_dialog_info"
                android:title="About" />
        </menu>
    </item>
</menu>

```

Panel nawigacyjny dostępny jest tylko z aktywności `TargetActivity`. Obsługą kliknąć w panelu zajmuje się metoda `onNavigationItemSelected`. Fragment tej metody pozwalający na uruchomienie aktywności `CodeActivity` przedstawia poniższy listing.

Listing 5.26: Lista opcji dostępna na panelu nawigacyjnym

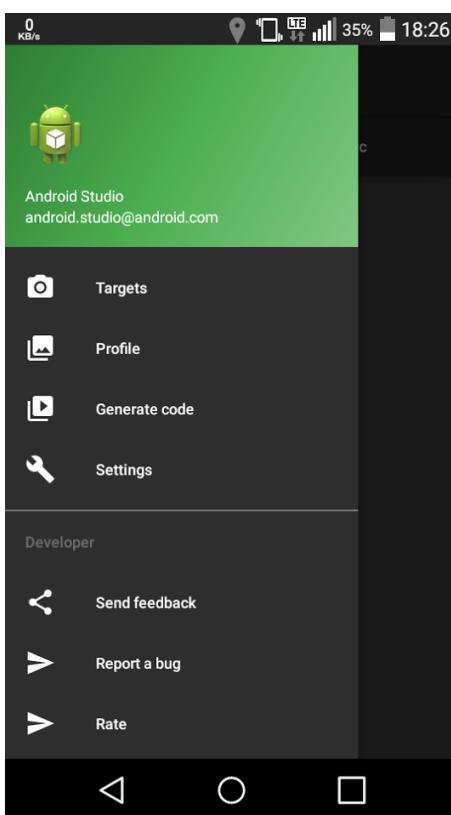
```

@Override
public boolean onNavigationItemSelected(MenuItem item) {
    // Handle navigation view item clicks here.
    int id = item.getItemId();

    if (id == R.id.nav_code) {
        IntentHelper.startActivityIntent(this, CodeActivity.class);
    } else if (id == R.id.nav_settings) {
        //...
    }

    drawer.closeDrawer(GravityCompat.START);
    return true;
}

```



Rys. 5.1: Panel nawigacyjny w aplikacji - implementacja

5.2.4. Komunikacja z serwerem

Przesyłanie danych pomiędzy zaimplementowanym serwerem, a opisywaną aplikacją mobilną odbywa się przy pomocy zapytań HTTP. Do tworzenia zapytań wykorzystana została biblioteka Retrofit rozpowszechniana na licencji Apache 2.0^{??}. Retrofit pozwala na przekształcenie API HTTP w interfejs Javy, a następnie przy pomocy metod i adnotacji zdefiniowanie dostępnych adresów URL. Następnie możliwe jest wygenerowanie implementacji oraz obsługi zapytań wysyłanych z jej pomocą.

Listing 5.27: Przykładowe interfejsy zapytań Retrofit

```

public interface TargetService {

    @GET("api/target")
    Call<List<Target>> getTargetsForUser(@Query("user") long id,
                                              @Header("X-Auth-Token") String token);

    @PATCH("api/target/confirm")
    Call<Target> confirm(@Query("target") Integer target,
                         @Query("code") String code, @Header("X-Auth-Token") String token);

    @GET("api/target")
    Call<Target> getTargetById(@Query("id") long id,
                               @Header("X-Auth-Token") String token);
}

public interface LocationService {

    @GET("/api/location")
    Call<List<Location>> findByUser(@Query("user") long id,
                                         @Header("X-Auth-Token") String token);

    @GET("/api/location")
    Call<List<Location>> findByUserAndDateAfter(@Query("user") long id,
                                                 @Query("date") RetroDate date, @Header("X-Auth-Token") String token);

    @POST("/api/location")
    Call<Void> save(@Body Location location,
                    @Header("X-Auth-Token") String token);
}

```

Adnotacje @GET, @POST oraz @PATCH widoczne na powyższym listingu określają metodę i adres URL wywołania. Metody interfejsu zwracają typ `Call<?>`, którego parametrem jest typ zwracany przez zapytanie. Adnotacje `@Query` i `@Body` pozwalają na dodanie parametrów do zapytania, a `@Header` służy do przesyłania tokena autoryzującego w nagłówku.

Sposób generowania implementacji interfejsów przedstawiono na poniższym listingu. Podobnie jak w przypadku klasy `IntentHelper` w celu redukcji powtarzanego kodu stworzono klasę ze statycznymi metodami pozwalającą w prosty sposób wygenerować implementację poprzedzającą podanie klasy interfejsu oraz bazowego adresu URL.

Listing 5.28: Klasa wspomagająca generowanie implementacji interfejsów biblioteki Retrofit

```

public class RetrofitBuilder {
    private static boolean uselocalhost = false;
    public static final String BASE_URL = "http://[...].com/";
    public static final String IMGUR_URL = "https://api.imgur.com/";
    public static final String LOCALHOST = "http://192.168.0.103:8080";

    public static Retrofit build(String url) {
        GsonBuilder builder = new GsonBuilder();
        HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
        interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
        OkHttpClient client = new OkHttpClient.Builder()
            .addInterceptor(interceptor).build();

        Gson gson = builder.setDateFormat("yyyy-MM-dd HH:mm:ss Z").
            create();
    }
}

```

```

        return new Retrofit.Builder()
            .client(client)
            .baseUrl(url)
            .addConverterFactory(GsonConverterFactory.create(gson))
            .build();
    }

    public static Object getService(Class<?> clas, String url) {
        if(clas == ImgurService.class)
            return build(url).create(clas);
        return build(useLocalhost ? LOCALHOST : url).create(clas);
    }
}

```

Przykład wykorzystania opisanego mechanizmu w praktyce przedstawia listing 5.29. Po wygenerowaniu implementacji serwisu tworzona jest instancja zapytania Call<Void> saveCall. Następnie przy pomocy metody enqueue dodawana jest ona do kolejki i oczekuje na obsłużenie. Odpowiedzi na zapytania tworzone w ten sposób obsługiwane są w osobnym wątku aplikacji, dlatego konieczne jest opisanie zachowania aplikacji po otrzymaniu odpowiedzi za pomocą klasy Callback. Metody tej klasy wywoływane są w głównym wątku aplikacji.

Listing 5.29: Użycie biblioteki Retrofit

```

private void saveLoginLocation() {
    Log.d("location", "Saving location");
    LocationService service = (LocationService) RetrofitBuilder
        .getService(LocationService.class, RetrofitBuilder.BASE_URL);
    Location loc = LocationHelper.map(lastKnown, Locations.LOGIN);
    Call<Void> saveCall = service.save(loc, AppVariable.token);
    saveCall.enqueue(new Callback<Void>() {
        @Override
        public void onResponse(Call<Void> call, Response<Void> response) {
            Log.d("locations", "Response!" + response.code());

            if (response.code() == 200) {
                Log.d("locations", "Saved!");
                goToTargets();
            }
        }

        @Override
        public void onFailure(Call<Void> call, Throwable t) {
            Log.d(getClass().getSimpleName(),
                  "Location saving error! " + t.getLocalizedMessage());
        }
    });
}

```

5.2.5. Rejestracja konta

Kontrolerem widoku jest klasa RegisterActivity. Rejestracja użytkownika odbywa się w czterech etapach:

1. Wybór zdjęcia
2. Uzupełnienie formularza
3. Przesłanie zdjęcia na serwer hostingowy
4. Utworzenie konta

Na szczytce widoku znajduje się komponent `ImageButton`. Po jego kliknięciu uruchomiona zostanie aktywność wyboru pliku z galerii. Metoda odpowiedzialna za to wywołanie przedstawiona została na listingu 5.30. Wykorzystuje ona opisaną wcześniej klasę `IntentHelper` do utworzenia intencji. Widoczna adnotacja `@OnClick` z biblioteki ButterKnife przypisuje metodę do akcji przycisku.

Listing 5.30: Wywołanie okna wyboru obrazu

```
@OnClick(R.id.image_button)
public void pickPhoto() {
    IntentHelper.chooseFileIntent(this);
}
```

Wywoływana aktywność zwraca dane w postaci adresu Uri wybranego obrazu, dlatego musi zostać obsłużona w metodzie `onActivityResult`. Po zakończeniu wyboru obrazu jest on przekazywany do biblioteki Image Cropper, która pozwala na przycięcie zdjęcia. Wynik działania aktywności przycięcia ponownie musi zostać obsłużony.

Listing 5.31: Obsługa wyników aktywności wyboru zdjęcia

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                               Intent data) {
    if (requestCode == IntentHelper.FILE_PICK
        && resultCode == RESULT_OK) {
        imageUri = data.getData();
        CropImage.activity(imageUri).setFixAspectRatio(true)
            .setGuidelines(CropImageView.Guidelines.ON)
            .start(this);
    }

    if (requestCode == CropImage.CROP_IMAGE_ACTIVITY_REQUEST_CODE) {
        CropImage.ActivityResult result = CropImage
            .getActivityResult(data);
        if (resultCode == RESULT_OK) {
            Uri resultUri = result.getUri();
            ImageLoader imageLoader = ImageLoader.getInstance();
            imageLoader.displayImage(
                "file://" + resultUri.getPath(), imageButton);

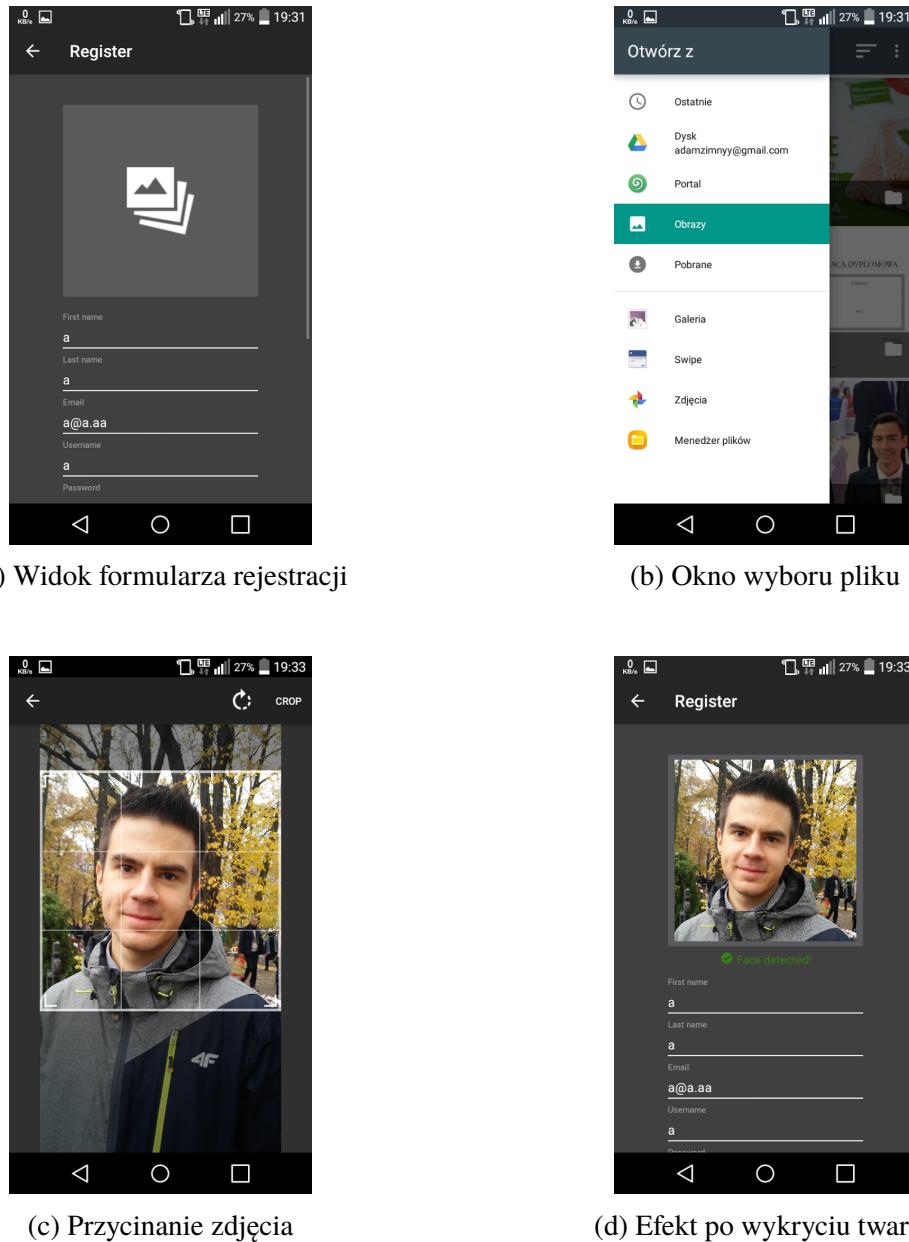
            try {
                croppedImage = MediaStore.Images.Media
                    .getBitmap(this.getContentResolver(), resultUri);
            } catch (IOException e) {
                e.printStackTrace();
            }
            FaceDetector faceDetector = new FaceDetector
                .Builder(this).build();
            Frame frame = new Frame.Builder()
                .setBitmap(croppedImage).build();
            SparseArray<Face> faces = faceDetector.detect(frame);
            faceDetector.release();

            if (faces.size() == 0) {
                noFaceDetected.setVisibility(View.VISIBLE);
            } else if (faces.size() > 1) {
                multipleFacesDetected.setVisibility(View.VISIBLE);
            }
        }
    }
}
```

```
        } else if (FaceHelper.getFaceSizePercent(faces.get(0),  
croppedImage) < Params.FACE_PERCENTAGE) {  
            faceTooSmall.setVisibility(View.VISIBLE);  
        } else {  
            faceDetected.setVisibility(View.VISIBLE);}  
    } else if (resultCode ==  
        CropImage.CROP_IMAGE_ACTIVITY_RESULT_ERROR_CODE) {  
        Exception error = result.getError();  
    }  
}
```

W celu zabezpieczenia przez wykorzystywaniem przez użytkowników zdjęć nieprzedstawiających osób, wprowadzony został mechanizm wykrywania twarzy przy pomocy narzędzia Mobile Vision?. Po zakończeniu przycinania zdjęcie jest one przekazywane do obiektu FaceDetector, a następnie poddawane analizie. W zależności od liczby wykrytych twarzy, pod zdjęciem wyświetlany jest odpowiedni komunikat.

Przykład działania opisanego procesu zaprezentowano na poniższych rysunkach.



Rys. 5.2: Przykład przebiegu procesu wyboru zdjęcia

Formularz danych składa się z szeregu pól tekstowych. Szczególnym przypadkiem jest pole wyboru daty urodzenia, które wyświetla okno dialogowe kalendarza po kliknięciu na nie. W celu wyświetlenia okna tworzony jest obiekt `DatePickerDialog`, a do pola tekstowego dodawany jest `OnFocusChangeListener`, pozwalający na detekcję zmiany aktywności pola. Wynik wyboru date z kalendarza obsługiwany jest przez interfejs `DatePickerDialog.OnDateSetListener`.

Listing 5.32: Wyświetlanie okna dialogowego kalendarza

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_register);
    ButterKnife.bind(this);
    datePickerDialog = createDialog();
    setTitle("Register");
    ImageLoaderHelper.initialize(this);
```

```

        setSupportActionBar(toolbar);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        birthdayField.setOnFocusChangeListener(
                new View.OnFocusChangeListener() {
                    @Override
                    public void onFocusChange(View view, boolean b) {
                        if (b) {
                            datePickerDialog.show();
                        }
                    }
                });
        imageView.requestFocus();
    }

    public DatePickerDialog createDialog() {
        // Use the current date as the default date in the picker
        final Calendar c = Calendar.getInstance();
        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);

        // Create a new instance of DatePickerDialog and return it
        return new DatePickerDialog(this, this, year, month, day);
    }

    @Override
    public void onDateSet(DatePicker datePicker, int i, int i1, int i2) {
        birthdayField.setText(i + "/" + i1 + "/" + i2);
    }
}

```

Po wypełnieniu przez użytkownika pól formularza i wcisnięciu przycisku Register u doły strony, następuje walidacja wprowadzonych danych. W przypadku wykrycia niepoprawnie uzupełnionych pól zostają one podświetlone, a na ekranie pokazuje się komunikat o błędzie.

Jeżeli walidacja została wykonana poprawnie, konto może zostać utworzone. Wybrany obraz przesyłany jest na zewnętrzny serwis hostingowy Imgur. Serwis ten został wybrany do przechowywania zdjęć ze względu na łatwość wykorzystania:

- zdjęcia mogą być przesyłane w dowolnym formacie
- nie ma konieczności implementacji mechanizmów obsługi plików po stronie serwerowej systemu
- plikom automatycznie nadawane są unikatowe identyfikatory, które łatwo można zapisać w bazie danych
- w łatwy sposób można odtworzyć adres przesłanego pliku na podstawie identyfikatora, bez znajomości rozszerzenia pliku

Korzystanie z tego serwisu wymaga rejestracji aplikacji w celu uzyskania klucza dostępu do API, który należy przesłać jako parametr zapytania. Serwis Retrofit zapytań Imgur przedstawia listing 5.33.

Listing 5.33: Interfejs zapytań serwisu Imgur

```

public interface ImgurService {

    @Multipart
    @POST("3/image")
    @Headers("Authorization: Client-ID 9df1a76680eff2a")
}

```

```

    Call<ImgurData> uploadImage(@Part("image") ProgressRequestBody image);
}

```

Do utworzenia konta wymagane jest wykonanie trzech zapytań http:

- sprawdzającego dostępność nazwy użytkownika
- do serwisu Imgur w celu przesłania zdjęcia
- przesłanie danych konta na adres /register/

. Ze względu na to, że kolejne zapytanie może zostać wykonane dopiero po uzyskaniu odpowiedzi z poprzedniego, nie można zastosować metody enqueue do wykonania zapytań. Aby zapewnić właściwą kolejność wykonania zapytania zostają wykonane w zdarzeniu asynchronicznym AsyncTask.

Korzystając z narzędzia Retrofit tworzone są serwisy, z których następnie pobierane są odpowiednie metody. Zapytania wykonywane są metodą execute(), która gwarantuje wykonanie ich w wątku, w którym zostaną wywołane. Po uzyskaniu odpowiedzi z identyfikatorem przesłanego zdjęcia jest on odzyskiwany przy pomocy metody imgurResponse.body().getData().getId() i przekazywany do obiektu UserTemplate. Po poprawnym wykonaniu zapytania tworzącego konto aktywność jest zamknięta w metodzie onPostExecute.

Listing 5.34: Wyświetlanie okna dialogowego kalendarza

```

public class RegisterTask extends AsyncTask<Void, Void, User> {

    @Override
    protected void onPostExecute(User user) {
        progressBar.setVisibility(View.GONE);
        if (user != null) {
            Log.d("validator", user.toString());
            Handler handler = new Handler();

            handler.postDelayed(new Runnable() {
                public void run() {
                    finish();
                }
            }, 3000);
        }
    }

    @SuppressLint("ConstantConditions")
    @Override
    protected User doInBackground(Void... voids) {
        ImgurService imgurService = (ImgurService) RetrofitBuilder
            .getService(ImgurService.class, RetrofitBuilder.IMGUR_URL);
        UserService userService = (UserService) RetrofitBuilder
            .getService(UserService.class, RetrofitBuilder.BASE_URL);
        Call<Boolean> usernameCall = userService
            .checkUsername(template.getUsername());
        Response<Boolean> usernameResponse;
        try {
            usernameResponse = usernameCall.execute();
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }

        File file = null;
        try {

```

```
file = new File(RegisterActivity.this.getCacheDir(),
                "temp_profile");
file.createNewFile();

//Convert bitmap to byte array

        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        croppedImage.compress(Bitmap.CompressFormat.PNG,
                0 /*ignored for PNG*/, bos);
        byte[] bitmapdata = bos.toByteArray();

//write the bytes in file
        FileOutputStream fos = new FileOutputStream(file);
        fos.write(bitmapdata);
        fos.flush();
        fos.close();
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    ProgressRequestBody pBody = new ProgressRequestBody(file,
            RegisterActivity.this);
    uploadImageCall = imgurService.uploadImage(pBody);
    Response<ImgurData> imgurResponse;
    try {
        imgurResponse = uploadImageCall.execute();
    } catch (IOException e) {
        return null;
    }
    template.setProfilePicture(imgurResponse.body()
                                .getData().getId());
    Call<User> userCall = userService.register(template);
    Response<User> registeredUser;
    try {
        registeredUser = userCall.execute();
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    return registeredUser.body();
}
```

5.2.6. Logowanie do konta

Logowanie do konta wykonywane jest przy pomocy adresu email oraz hasła podanego przy rejestracji. Obsługą widoku logowania zajmuje się klasa `LoginActivity`. Po wprowadzeniu danych i naciśnięciu przycisku „Log in” walidowane są pola danych, a następnie przesyłane na serwer z użyciem serwisu `Retrofit`. W odpowiedzi na zapytanie przesyłany jest token dostępu do API serwera, który zapamiętywany jest w statycznym polu klasy `AppVariable`, aby mógł zostać wykorzystany w kolejnych zapytaniach. Po poprawnym zalogowaniu aplikacji przechodzi do aktywności `GpsActivity`.

Listing 5.35: Logowanie do konta

```
private void attemptLogin() {
```

```

// Reset errors.
usernameField.setError(null);
passwordField.setError(null);

// Store values at the time of the login attempt.
String username = usernameField.getText().toString();
String password = passwordField.getText().toString();

boolean cancel = false;
View focusView = null;

// Check for a valid password, if the user entered one.
if (TextUtils.isEmpty(password)) {
    passwordField.setError(getString(R.string.error_invalid_password));
    focusView = passwordField;
    cancel = true;
}

// Check for a valid email address.
if (TextUtils.isEmpty(username)) {
    usernameField.setError(getString(R.string.error_field_required));
    focusView = usernameField;
    cancel = true;
}

if (cancel) {
    focusView.requestFocus();
} else {
    showProgress(true);
    final AuthService authService = (AuthService) RetrofitBuilder
        .getService(AuthService.class, RetrofitBuilder.BASE_URL);
    Call<AuthenticationResponse> authCall;
    authCall = authService.authenticate(
        new AuthenticationRequest(username, password));
    authCall.enqueue(new Callback<AuthenticationResponse>() {
        @Override
        public void onResponse(Call<AuthenticationResponse> call,
                               Response<AuthenticationResponse> response) {

            if (response.code() == 200) {

                AppVariable.token = response.body().getToken();
                AppVariable.loggedUser = response.body().getUser();
                Toast.makeText(LoginActivity.this,
                               "Login success", Toast.LENGTH_LONG).show();
                IntentHelper.startActivityIntent(LoginActivity.this,
                                                GpsActivity.class);
                finish();
            }
        }
    } else {
        AppVariable.token = "";
        AppVariable.loggedUser = null;
        passwordField.setError(getString(
            R.string.error_incorrect_password));
        passwordField.requestFocus();
        showProgress(false);
    }
}
}

```

5.2.7. Lokalizacja urządzeń

Po zalogowaniu aplikacja przechodzi do zapisu lokalizacji. Na ekranie wyświetlony zostaje widok `activity_gps.xml` obsługiwany przez `GpsActivity`.

Po utworzeniu widoku w metodzie `onCreate` pobierana jest instancja klasy `LocationManager` pozwalająca na nasłuchiwanie aktualizacji lokalizacyjnych telefonu. Przed rozpoczęciem aktualizacji należy sprawdzić czy moduł GPS został włączony w ustawieniach oraz czy aplikacji ma uprawnienia dostępu. Jeżeli wymagane jest włączenie którejś z tych opcji w aplikacji pokazane zostanie stosowne okno dialogowe. Implementacja opisanego procesu zaprezentowana została na poniższym listingu.

Listing 5.36: Sprawdzenie wymagań użycia modułu GPS

```
public class GpsActivity extends AppCompatActivity
    implements LocationListener {

    android.location.Location lastKnown;

    @BindView(R.id.settings)
    Button settingsButton;

    @BindView(R.id.status_text)
    TextView statusText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_gps);
        ButterKnife.bind(this);
        AppVariable.locationManager = (LocationManager)
            getSystemService(Context.LOCATION_SERVICE);
        AppVariable.locationListener = this;
        getLocation();
    }

    private void getLocation() {
        //TODO zmienić UI odpowiednio do wykonywanej części

        boolean gpsPermission = checkPermissions();
        if (!gpsPermission) {
            Log.d(getClass().getSimpleName(),
                  "No GPS permissions!");
            requestPemissions();
            return;
        }
        boolean gpsEnabled = checkSettings();
        if (!gpsEnabled) {
            Log.d(getClass().getSimpleName(),
                  "GPS not enabled!");
            buildAlertMessageNoGps();
            return;
        }
        // subscribeForLocation(25f, this);
        getLocationUpdates();
    }

    private void requestPemissions() {
```

```

        Log.d(getClass().getSimpleName(), "Requesting permissions...");
        statusText.setText("Requesting permissions...");
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            requestPermissions(new String[]{Manifest.permission.ACCESS_FINE_LOCATION,
                Manifest.permission.ACCESS_COARSE_LOCATION}, IntentHelper.PERMISSION_GPS);
        }
        Log.d(getClass().getSimpleName(), "Done.");
    }

    private boolean checkSettings() {
        statusText.setText("Checking settings...");
        Log.d(getClass().getSimpleName(), "Checking settings...");
        return AppVariable.locationManager
            .isProviderEnabled(LocationManager.GPS_PROVIDER);
    }

    private boolean checkPermissions() {
        statusText.setText("Checking permissions...");
        Log.d(getClass().getSimpleName(), "Checking permissions...");

        return ActivityCompat.checkSelfPermission(this,
            Manifest.permission.ACCESS_FINE_LOCATION) ==
            PackageManager.PERMISSION_GRANTED &&
            ActivityCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_COARSE_LOCATION) ==
            PackageManager.PERMISSION_GRANTED;
    }

    private void buildAlertMessageNoGps() {
        Log.d(getClass().getSimpleName(), "Showing GPS settings alert...");
        statusText.setText("Enable GPS to continue.");

        final AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setMessage("Your GPS seems to be disabled,\n" +
            "do you want to enable it?")
            ..setCancelable(false)
            .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    startActivityForResult(new Intent(
                        android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS),
                        IntentHelper.GPS_SETTINGS);
                }
            })
        [...]
    });
    final AlertDialog alert = builder.create();
    alert.show();
}

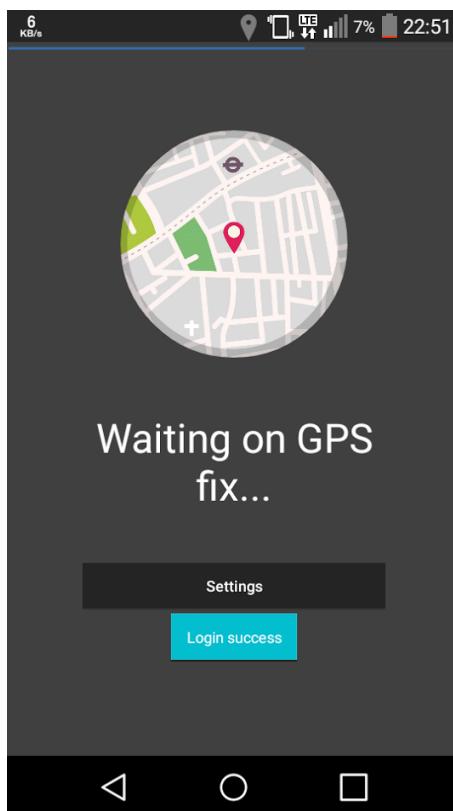
```

Jeśli moduł GPS jest gotowy do użycia, możliwe jest rozpoczęcie nasłuchiwanego. Android dostarcza dwóch metod lokalizacyjnych, za pomocą GPS oraz na podstawie nadajników GSM. W aplikacji wykorzystywana jest głównie ta druga ze względu na szybsze aktualizacje, jednak ujętywione zostają obie. Przed wykonaniem dowolnej operacji korzystającej z obiektów

LocationManager, konieczne jest wykonanie sprawdzenia uprawnień, w sposób pokazany metodzie checkPermissions na listingu 5.36.

Listing 5.37: Rozpoczęcie aktualizacji GPS

```
public void getLocationUpdates() {
    statusText.setText("Waiting on GPS fix...");
    [...]
    // sprawdzenie uprawnień
    AppVariable.locationManager.requestLocationUpdates(
        LocationManager.NETWORK_PROVIDER, 0, 0, this);
    AppVariable.locationManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER, 0, 0, this);
}
```



Rys. 5.3: Ekran zapisu lokalizacji - implementacja

Aktualizacje lokalizacji otrzymywane są przed zaimplementowany interfejs LocationListener. W trakcie oczekiwania na ekranie wyświetlany jest stosowny komunikat, co pokazano na rysunku 5.3. Jeżeli po aktualizacji dokładność pozycji osiągnie wymagany poziom, wykonywany jest zapis do bazy z użyciem serwisu Retrofit, a lokalizacja zapamiętywana jest w klasie AppVariable, aby przyspieszyć jej pobieranie w kolejnych zapytaniach. Po wykonaniu zapisu wywoływana jest aktywność TargetActivity.

Listing 5.38: Obsługa aktualizacji i zapis do bazy

```
private void saveLoginLocation() {
    Log.d("location", "Saving location");
    LocationService service = (LocationService) RetrofitBuilder
        .getService(LocationService.class, RetrofitBuilder.BASE_URL);
    Location loc = LocationHelper.map(lastKnown, Locations.LOGIN);
```

```

        Call<Void> saveCall = service.save(loc, AppVariable.token);
        saveCall.enqueue(new Callback<Void>() {
            @Override
            public void onResponse(Call<Void> call,
                                  Response<Void> response) {
                if (response.code() == 200) {
                    goToTargets();
                }
            }
        });

@Override
public void onLocationChanged(android.location.Location location) {
    if (location.getAccuracy() < Params.LOCATION_ACCURACY) {
        AppVariable.lastKnownLocation = new Location(
            location.getLatitude(), location.getLongitude());
        AppVariable.lastKnownLocationDate = new Date();
        lastKnown = location;
        if (!AppVariable.loginLocationSaved) {
            saveLoginLocation();
            AppVariable.loginLocationSaved = true;
        }
    }
}
}

```

5.2.8. Lista zleceń

Organizacja widoki listy zleceń jest inna od pozostałych aktywności w aplikacji. Aby umożliwić podział na dwie zakładki zastosowane zostały komponenty TabLayout oraz ViewPager pozwalające na zmianę widoku przeciągnięciem w poziomie po ekranie. Zawartość poszczególnych widoków przeniesiona została do fragmentów. W przeciwnieństwie do aktywności, fragmenty nie mogą występować samodzielnie w aplikacji, muszą być częścią jakiejś aktywności. Możliwe jest jednak posiadanie wielu fragmentów jednocześnie w ramach jednego widoku.

Zasób widoków, jakie mogą zostać wyświetlane na kartach definiuje się w adapterze komponentu ViewPager. Zastosowaną klasę adaptera przedstawia listing 5.39. Zdefiniowane zostały dwie karty o tytułach „Private” oraz „Public”.

Listing 5.39: Adapter fragmentów widoku zleceń

```

public class TargetPageAdapter extends FragmentPagerAdapter {

    final int PAGE_COUNT = 2;
    private String tabTitles[] = new String[]{"Private", "Public"};
    private Context context;

    public TargetPageAdapter(FragmentManager fm, Context context) {
        super(fm);
        this.context = context;
    }

    @Override
    public int getCount() {
        return PAGE_COUNT;
    }

    @Override
    public Fragment getItem(int position) {

```

```

        return TargetsPageFragment.newInstance(position, position != 0);
    }
    @Override
    public CharSequence getPageTitle(int position) {
        // Generate title based on item position
        return tabTitles[position];
    }
}

```

Fragmenty umieszczane na kartach składają się z komponentu RecyclerView wyświetlającego listę zleceń zawartego w widoku SwipeRefreshLayout. Widok ten pozwala na wykonanie wybranej akcji gestem pociągnięcia w dół.

Lista RecyclerView korzysta z kontenerów nazywanych ViewHolder do wyświetlania elementów. Konieczne jest zdefiniowanie pliku interfejsu określającego wygląd elementu, a następnie stworzenie zmiennych w klasie ViewHolder, które zostaną później wykorzystane ustawienia wyświetlanych danych, i powiązanie ich z widokiem. W implementowanej liście pojedynczy element składa się ze zdjęcia, kilku elementów tekstowych oraz kolorowego paska statusu.

Listing 5.40: Adapter fragmentów widoku zleceń

```

protected static class ViewHolder extends RecyclerView.ViewHolder {

    @BindView(R.id.name)
    protected TextView name;

    @BindView(R.id.profile_image)
    ImageView profileImage;

    @BindView(R.id.text_distance)
    TextView distance;

    @BindView(R.id.text_time_left)
    TextView timeLeft;

    @BindView(R.id.status_view)
    View statusBarView;

    public ViewHolder(View view) {
        super(view);
        ButterKnife.bind(this, view);
    }
}

```

Listy RecyclerView podobnie jak ViewPager korzystają z adaptera do zarządzania wyświetlonymi danymi. Ze względu na to, że proces jego implementacji może być skomplikowany, stworzone zostały liczne biblioteki mające na celu uproszczenie tego procesu. Biblioteka wykorzystana w tym projekcie to FastAdapter stworzona przez użytkownika mikepenz serwisu GitHub, rozpowszechniana na licencji Apache 2.0. W tej implementacji wymagane jest jedynie stworzenie klasy modelu danych rozszerzającej dostarczoną w bibliotece klasę AbstractItem.

W klasie należy określić wykorzystywany plik interfejsu oraz przekazać dane do wyświetlenia na dostarczonym obiekcie ViewHolder w metodzie onBindViewHolder.

Listing 5.41: Adapter fragmentów widoku zleceń

```

public class TargetItem extends AbstractItem<TargetItem,
    TargetItem.ViewHolder> {
    Target target;
    Context context;
}

```

```

//The unique ID for this type of item
CountDownTimer timer;

public TargetItem withTarget(Target target) {
    this.target = target;
    return this;
}

public TargetItem withContext(Context context) {
    this.context = context;
    return this;
}

@Override
public int getType() {
    return R.id.target_list_item;
}

//The layout to be used for this type of item
@Override
public int getLayoutRes() {
    return R.layout.list_element;
}

public Target getTarget() {
    return target;
}

public void stopTimer() {
    Log.d("timer", "stopTimer");
    if (timer != null)
        timer.cancel();
}

//The logic to bind your data to the view
@Override
public void bindView(final ViewHolder viewHolder) {
    //call super so the selection is already handled for you
    super.bindView(viewHolder);

    viewHolder.name.setText(target.getHunted().getFirstName() + " " +
        target.getHunted().getLastName() + ", " +
        DateUtils.getAgeFromBirthDate(target.getHunted().getDateOfBirth()));
    ImageLoader imageLoader = ImageLoader.getInstance();
    imageLoader.displayImage(ImgurHelper.compileUrl(
        target.getHunted().getProfilePicture()), viewHolder.profileImage);
    viewHolder.distance.setText(String.valueOf(
        LocationHelper.distance(target)));
    switch (target.getStatus()) {
        case Targets.ACTIVE:
            timer = new CountDownTimer(
                DateUtils.timeLeftOnTargetFromDateAsSeconds(
                    public void onTick(long millisUntilFinished) {
                        viewHolder.timeLeft.setText(
                            DateUtils.timeLeftOnTargetFromDate(
                                target.getCreated())));
                    });
            timer.start();
    }
}

```

```

        viewHolder.statusBarView.setBackgroundColor(
            ContextCompat.getColor(context, R.color.target_yellow));
        break;
    case Targets.FAILED:
        viewHolder.timeLeft.setText(Targets.FAILED);
        viewHolder.statusBarView.setBackgroundColor(
            ContextCompat.getColor(context, R.color.target_red));
        if (timer != null) timer.cancel();
        break;
    case Targets.COMPLETED:
        viewHolder.timeLeft.setText(Targets.COMPLETED);
        viewHolder.statusBarView.setBackgroundColor(
            ContextCompat.getColor(context, R.color.target_green));
        if (timer != null) timer.cancel();
        break;
    }
}
}

```

Do tworzonego obiektu `TargetItem` przekazywany jest również pobrany z bazy danych obiekt `Target` zawierający dane do wyświetlenia. W metodzie `onBindViewHolder` ustawiane jest wyświetlane zdjęcie, oraz w zależności od statusu tekst oraz kolor bocznego paska elementu. Jeżeli zlecenie pozostaje aktywne uruchamiane jest odliczanie, które co sekundę aktualizuje czas pozostały jego wykonanie.

Pobieranie danych o zleceniach realizowane jest w klasie `TargetsPageFragment`. Metodą odpowiedzialną za tą operację jest `refresh`. Na początku zatrzymywane są wszystkie zegary jakie mogą znajdować się na istniejących elementach listy, a następnie listy są czyszczone. Przy użyciu Retrofit wykonywane jest zapytanie do serwera, a po otrzymaniu wyników, dla każdego rekordu tworzony jest obiekt opisanej wcześniej klasy `TargetItem`. Stworzone obiekty są dodawane do adaptera w celu wyświetlenia.

Listing 5.42: Metoda odświeżania listy zleceń

```

private void refresh() {
    for (TargetItem t : recyclerItems) {
        t.stopTimer();
    }
    recyclerItems.clear();
    fastAdapter.clear();
    targetCall = targetService.getTargetsForUser(
        AppVariable.loggedUser.getId(), AppVariable.token);
    targetCall.enqueue(new Callback<List<Target>>() {
        @Override
        public void onResponse(Call<List<Target>> call,
            Response<List<Target>> response) {
            if (response.code() == 200) {
                List<Target> targets = response.body();
                Collections.sort(targets);
                List<TargetItem> items = new ArrayList<>();
                for (Target t : targets) {
                    if(t.isPublicTarget() == isPublic)
                        items.add(new TargetItem()
                            .withTarget(t)
                            .withContext(getActivity())));
                }
                refreshLayout.setRefreshing(false);
                recyclerItems.addAll(items);
                fastAdapter.add(items);
            }
        }
    });
}
}

```

```

        fastAdapter.notifyDataSetChanged();
    }
}
);
}
}

```

Aby umożliwić odświeżanie listy z wykorzystaniem widoku SwipeRefreshLayout wymagane jest dodanie słuchacza OnRefreshLayout. Po kliknięciu elementu listy powinna zostać uruchomiona aktywność szczegółów zlecenia. Przy użyciu biblioteki FastAdapter możliwe jest dodanie słuchacza OnClickListener do każdego z elementów listy. Operacje te wykonywane są w metodzie onCreateView.

Listing 5.43: Konfiguracja słuchaczy klasy TargetsPageFragment

```

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {

    View view = inflater.inflate(R.layout.content_targets_recycler,
        container, false);
    ButterKnife.bind(this, view);
    fastAdapter = new FastItemAdapter();
    recyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));
    recyclerView.setAdapter(fastAdapter);
    fastAdapter.withSelectable(true);
    fastAdapter.withOnClickListener(new FastAdapter
        .OnClickListener<TargetItem>() {
        @Override
        public boolean onClick(View v, IAdapter<TargetItem> adapter,
            TargetItem item, int position) {
            openDetails(item.getTarget());
            return true;
        }
    });
    refreshLayout.setOnRefreshListener(
        new SwipeRefreshLayout.OnRefreshListener() {
            @Override
            public void onRefresh() {
                refresh();
            }
        });
    refresh();
    return view;
}

```

W metodzie openDetails dane modelu z wybranego elementu listy przekazywane są przez klasę Bundle do aktywności DetailsActivity.

Listing 5.44: Metoda uruchamiająca aktywność szczegółów zlecenia

```

private void openDetails(Target target) {
    Bundle bundle = new Bundle();
    bundle.putSerializable("target", target);
    IntentHelper.startActivityIntent(getActivity(),
        DetailsActivity.class, bundle);
}

```

5.2.9. Szczegóły zlecenia

Aktywnością obsługującą ekran szczegółów zlecenia jest `DetailsActivity`. Dane o wyświetlanym zleceniu przekazywane są w intencji przez klasę `Bundle`. Na widoku znajduje się ten sam zestaw danych, co na elemencie listy opisany w punkcie 5.2.8, zatem metoda inicjalizująca komponenty widoku jest zbliżona do tej przedstawionej na listingu 5.39. Zasadniczą różnicą jest sposób pobrania danych do inicjalizacji. Nie są one przekazywane w konstruktorze, lecz muszą zostać pobrane z intencji. Widok inicjalizowany jest na podstawie otrzymanej klasy modelu `Target`.

Listing 5.45: Pobranie danych inicjalizujących z intencji

```
private void initView(Target t) {
    target = t == null ? (Target) getIntent()
        .getSerializableExtra("target") : t;

    [...] // ustawienie wartości pól

}
```

Z poziomu widoku szczegółów zlecenia możliwe jest wprowadzenie kodu potwierdzającego. U dołu ekranu znajduje się pole tekstowe `codeField` oraz przycisk zatwierdzający `sendButton`, co pokazano na rysunku 5.4. Metoda wysyłająca dane przypisana została za pomocą adnotacji `ButterKnife`. Po naciśnięciu przycisku tworzone jest zapytanie z użyciem narzędzia `Retrofit`, a następnie przesyłane na serwer. Po otrzymaniu odpowiedzi z serwera, widok jest ponownie inicjalizowany nowymi danymi. Jeżeli wprowadzony kod był niepoprawny, przycisk wysyłający kod zostanie zablokowany na 10 sekund. Tworzony jest zegar, który odlicza czas i aktualizuje tekst wyświetlany na przycisku co sekundę.

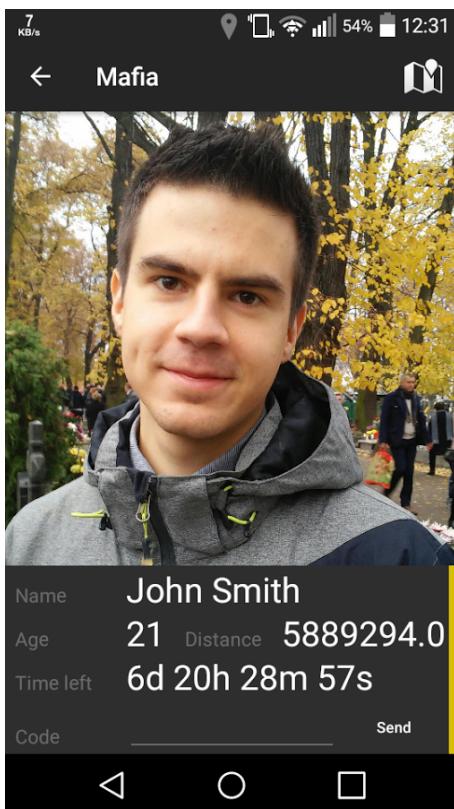
Listing 5.46: Pobranie danych inicjalizujących z intencji

```
@onClick(R.id.send_button)
public void confirmTarget() {
    TargetService service = (TargetService) RetrofitBuilder
        .getService(TargetService.class, RetrofitBuilder.BASE_URL);
    Call<Target> confirmCall = service.confirm(target.getId(),
        codeField.getText().toString().trim(), AppVariable.token);
    confirmCall.enqueue(new Callback<Target>() {
        @Override
        public void onResponse(Call<Target> call, Response<Target> response) {
            if (response.code() == 200 &&
                response.body().getCompleted() != null) {
                target = response.body();
                initView(target);
            } else {
                sendButton.setEnabled(false);
                sendButton.setTextColor(ContextCompat.getColor(
                    DetailsActivity.this, R.color.placeholder_text));
                CountDownTimer timer = new CountDownTimer(
                    Params.WRONG_CODE_RETRY_TIME_SECONDS * 1000, 1000) {
                    @Override
                    public void onTick(long l) {
                        sendButton.setText(DateUtils.formatMillisAsYMDHMS(l));
                        sendButton.invalidate();
                    }
                }

                @Override
                public void onFinish() {
```

```
        sendButton.setText(R.string.button_code_send);
        sendButton.setEnabled(true);
        sendButton.setTextColor(ContextCompat.getColor(
            DetailsActivity.this, R.color.white));
        sendButton.invalidate();
    }
};

timer.start();
}
});
```



Rys. 5.4: Widok szczegółów zlecenia - implementacja

Na górnym pasku aplikacji znajduje się przycisk z ikoną mapy. Po jego kliknięciu otwarta zostaje aktywność MapActivity wyświetlająca znane lokalizacje gracza. Ponownie dane zlecenia przekazywane są przez obiekt Bundle. Ze względu na to, że przycisk ten nie należy do klasy Button, lecz jest elementem menu kontekstowego, za jego obsługę odpowiedzialna jest metoda `onOptionsItemSelected`.

Listing 5.47: Obsługa wywołania aktywności mapy

```
@Override
public boolean onOptionsItemSelected(MenuItem menuItem) {
    if (menuItem.getItemId() == android.R.id.home) {
        finish();
    }
    if (menuItem.getItemId() == R.id.action_map) {
        Bundle bundle = new Bundle();
        bundle.putSerializable("target", target);
        IntentHelper.startActivityIntent(this, MapActivity.class, bundle);
    }
}
```

```

    }
    return super.onOptionsItemSelected(menuItem);
}

```

5.2.10. Mapa znanych lokalizacji użytkownika

Umieszczenie mapy Google wewnątrz widoku odbywa się poprzez dodanie do pliku interfejsu fragmentu z ustawionym parametrem `android:name="com.google.android.gms.maps.MapFragment"`. Pod zadany widok zostanie podpięta instancja klasy `MapFragment`. Kontroler aktywności musi implementować interfejs `OnMapReadyCallback`. Po utworzeniu widoku mapa jest pobierana metodą `getMapAsync`. Następnie z serwera pobierane są punkty lokalizacyjne za pomocą interfejsu Retrofit.

Listing 5.48: Fragment pliku interfejsu aktywności `MapActivity`

```

<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/map"
    tools:context="mafia.adamzimny.mafia.activity.MapActivity"
    android:name="com.google.android.gms.maps.SupportMapFragment"/>

```

Narzędzie Google Maps oferuje możliwość grupowania znaczników znajdujących się blisko siebie w celu zwiększenia przejrzystości mapy. Grupowanie wymaga utworzenia obiektu `ClusterManager`, który zarządza wyświetlonymi znacznikami. W przypadku wyświetlania bez grupowania znaczniki dodawać można bezpośrednio do obiektu mapy. Po pobraniu danych o lokalizacji znaczniki dodawane są w jeden z opisanych wyżej sposobów zależnie od ustawionych preferencji. Znaczników przypisywane są dodatkowe informacje `title` oraz `snippet` wyświetlane po kliknięciu na znacznik. Dla lokalizacji zapisanych mniej niż tydzień temu w tytule zamiast pełnej daty wyświetlony zostanie względny czas utworzenia. Na rysunku 5.5 przedstawiono efekt dodania znaczników dla przykładowych danych.

Listing 5.49: Obsługa wywołania aktywności mapy

```

@Override
public void onMapReady(GoogleMap googleMap) {
    map = googleMap;
    setUpClusterer();
}

private void setUpClusterer() {
    mClusterManager = new ClusterManager<>(this, map);
    map.setOnCameraIdleListener(mClusterManager);
    map.setOnMarkerClickListener(mClusterManager);
    mClusterManager.setRenderer(new OwnRendering(
        getApplicationContext(), map, mClusterManager));
}

public void getHuntedLocations() {
    [...] // Retrofit call
}

```

```
for (Location l : list) {
    if (DateUtils.daysSince(l.getDate().getTime()) > 6) {
        if (AppVariable.useMarkerClusters) {
            mClusterManager.addItem(new MapMarker(l.getLatitude(),
                l.getLongitude(), DateUtils.date(l.getDate().getTime()),
                DateUtils.hour(l.getDate().getTime())));
        } else {
            map.addMarker(new MarkerOptions()
                .position(new LatLng(l.getLatitude(), l.getLongitude()))
                .title(DateUtils.date(l.getDate().getTime()))
                .snippet(DateUtils.hour(l.getDate().getTime())));
        }
    } else {
        if (AppVariable.useMarkerClusters) {
            mClusterManager.addItem(new MapMarker(l.getLatitude(),
                l.getLongitude(), android.text.format.DateUtils
                    .getRelativeTimeSpanString(
                l.getDate().getTime() + "",
                DateUtils.format(l.getDate().getTime())));
        } else {
            map.addMarker(new MarkerOptions()
                .position(new LatLng(l.getLatitude(), l.getLongitude()))
                .title(android.text.format.DateUtils.getRelativeTimeSpanString(
                    l.getDate().getTime() + ""))
                .snippet( DateUtils.format(l.getDate().getTime())));
        }
    }
}
```



(a) Znaczniki lokalizacji



(b) Grupa znaczników



(c) Okno z danymi o znaczniku

Rys. 5.5: Mapa lokalizacji użytkownika po dodaniu znaczników

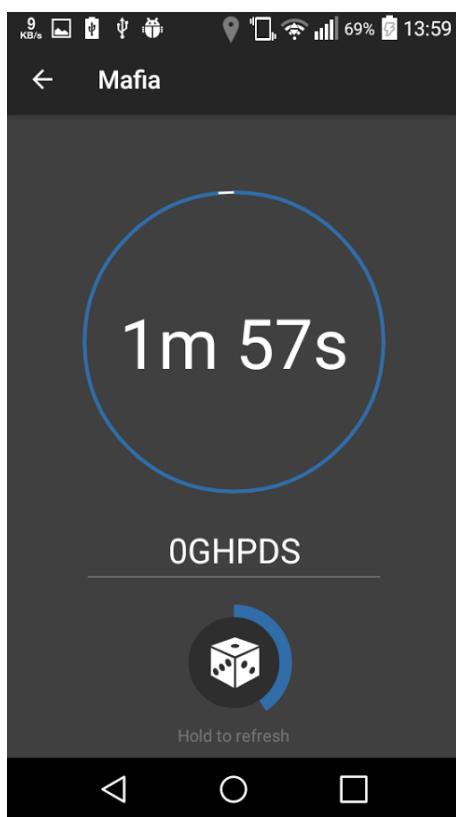
5.2.11. Generowanie kodów potwierdzających

Do aktywności generowania kodu przejść można z bocznego panelu nawigacyjnego. Kontrolerem aktywności jest klasa `CodeActivity`. Do generowania kodu wykorzystywany jest przycisk utworzony przy pomocy biblioteki `HoldToLoadLayout`. Podczas przytrzymania przycisku

wyświetlany jest pasek ładowania. Biblioteka wymaga zaimplementowania metod wywoływanych po ukończeniu ładowania. Po utworzeniu widoku, w metodzie `onCreateView` przypisywana jest instancja słuchacza wyzwalającego zdarzenia po osiągnięciu stanu pełnego i pustego paska ładowania. Wygląd interfejsu podczas ładowania przedstawiono na rysunku 5.6.

Listing 5.50: Obsługa zdarzeń przycisku wykorzystującego `HoldToLoadLayout`

```
button.setFillListener(new HoldToLoadLayout.FillListener() {
    @Override
    public void onFull() {
        if (canSendRequests) {
            generateCode();
            codeField.setText(codeString);
            progressBar.setProgress(0);
            canSendRequests = false;
        }
        button.setStrokeColor(R.color.target_green);
        button.setDuration(750);
    }
    [...]
}); // onEmpty(), onAngleChanged()
```



Rys. 5.6: Widok generowania kodu podczas ładowania przycisku

Po ukończeniu ładowania przycisku wywoływana jest metoda `generateCode` tworząca losowy ciąg znaków i wykonująca zapis do bazy przez zapytanie http. Tworzony jest również zegar aktualizujący interfejs co sekundę w celu wizualizacji pozostałego czasu aktywności kodu. Metoda `setDuration` pozwala zmienić czas potrzebny na wypełnienie ładowania.

Listing 5.51: Generowanie kodu

```

private void generateCode() {
    codeString = nextString(Params.CODE_LENGTH);
    AppVariable.code = codeString;
    Log.d("code", "Code generated! " + codeString);
    if (timer != null) timer.cancel();
    timer = new CountDownTimer(TWO_MINUTES, 1000) {
        public void onTick(long millisUntilFinished) {
            AppVariable.codeTimeLeft = millisUntilFinished;
            time.setText(DateUtils.formatMillisAsYMDHMS(millisUntilFinished));
            progressBar.setProgressWithAnimation(
                millisUntilFinished * 100f / TWO_MINUTES, 2000);
        }

        public void onFinish() {
            codeField.setText("Code expired!");
            progressBar.setProgress(0);
            time.setText("0m 00s");
        }
    };
    timer.start();

    Location codeLocation = AppVariable.lastKnownLocation;
    codeLocation.setType(Locations.CODE);
    Code code = Code.Builder
        .create()
        .withCode(codeString)
        .withCreated(new Date())
        .withLocation(codeLocation)
        .withUser(AppVariable.loggedUser)
        .build();

    CodeService service = (CodeService) RetrofitBuilder
        .getService(CodeService.class, RetrofitBuilder.BASE_URL);
    Call<String> call = service.createNewCode(AppVariable.token, code);
    call.enqueue( [...] );
}

```

Do generowania kodów wykorzystana została metoda `nextString`, której parametrem jest długość ciągu znaków. Kody składają się z dużych liter i cyfr.

Listing 5.52: Generowanie losowego ciągu znaków

```

private final Random random = new Random();

public String nextString(int len) {
    StringBuilder tmp;

    tmp = new StringBuilder();
    for (char ch = '0'; ch <= '9'; ++ch)
        tmp.append(ch);
    for (char ch = 'A'; ch <= 'Z'; ++ch)
        tmp.append(ch);
    char[] symbols = tmp.toString().toCharArray();
    tmp.setLength(0);

    for (int i = 0; i < len; i++)
        tmp.append(symbols[random.nextInt(symbols.length)]);
}

```

```
    return tmp.toString();  
}
```

Rozdział 6

Testy systemu

Rozdział 7

Podsumowanie

Celem pracy było zaprojektowanie oraz zaimplementowanie systemu do organizacji gry miejscowości oraz brania w niej udziału. W ramach realizacji projektu wykonana została aplikacja serwerowa pozwalająca na prowadzenie gry, baza danych przechowująca dane użytkowników oraz aplikacja mobilna na systemy Android pełniąca rolę klienta gry.

Podstawowym celem gry jest odnajdywanie innych graczy w rzezywistym świecie na podstawie informacji udzielanych przez nich przy rejestracji oraz danych lokalizacyjnych rejestrowanych przez aplikację mobilną.

Aplikacja serwerowa opracowana została w języku Java 8 z wykorzystaniem technologii Spring Framework, w oparciu o styl architektury REST pozwalający na komunikację z pozostałymi uczestnikami systemu poprzez zapytania HTTP. Baza danych została wykonana w systemie PostgreSQL. Dzięki wykorzystaniu narzędzia do mapowania obiektowo-relacyjnego Hibernate możliwe było utworzenie schematu bazy danych na podstawie zaimplementowanych klas w języku Java. Stworzona aplikacja serwerowa pozwala na tworzenie kont użytkowników i logowanie do nich, pobieranie danych użytkowników, zleceń oraz lokalizacji, potwierdzanie wykonywania zadań gry, modyfikację parametrów gry oraz zarządzanie kontami użytkowników. Określone zostały dwie grupy użytkowników systemu posiadająca różne poziomy uprawnień: gracz i administrator. Zaimplementowany został moduł automatycznego prowadzenia gry bez konieczności nadzoru przez administrację.

Klient gry na systemy Android wykonany również wykonany został z wykorzystaniem języka Java, będącego podstawowym sposobem implementacji tego typu aplikacji. Do komunikacji z serwerem wykorzystana została biblioteka Retrofit, pozwalająca na generowanie implementacji serwisów zapytań HTTP na podstawie zdefiniowanych interfejsów. Aplikacja pozwala na rejestrację konta z wyborem i przycięciem zdjęcia z galerii telefonu, logowanie, zapis lokalizacji urządzenia, wyświetlenie zleceń, szczegółów zlecenia i mapy lokalizacji, potwierdzanie zleceń i generowanie kodów.

Ze względu na szeroki zakres projektu fazą implementacji zostały objęte jedynie wybrane wymagania funkcjonalne. Poza zaimplementowanymi modułami w projekcie systemu opisana została konsola administracyjna w formie aplikacji internetowej, a także dodatkowe funkcjonalności aplikacji mobilnej.

Literatura

Dodatek A

Opis załączonej płyty CD/DVD

Tutaj jest miejsce na zamieszczenie opisu zawartości załączonej płyty. Należy wymienić, co zawiera.