POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Inżynieria systemów informatycznych (INS)

PRACA DYPLOMOWA INŻYNIERSKA

System informatyczny wspomagający organizację wspólnych przejazdów samochodami w mieście

The information system supporting organization of joint car journeys in the city

AUTOR:
Mikołaj Grygiel

PROWADZĄCY PRACĘ:
dr inż. Marek Piasecki

OCENA PRACY:

WROCŁAW 2015

Spis treści

Rozdział 1

Wprowadzenie

1.1 Cel pracy

Celem niniejszego projektu jest zaprojektowanie systemu informatycznego, który będzie wspomagał użytkowników przy organizacji wspólnych przejazdów samochodami w mieście oraz zaimplementowanie wybranych funkcjonalności tego systemu.

Głównymi funkcjonalnościami systemu będą:

- Oferowanie wspólnych przejazdów samochodem innym użytkownikom
- Wyszukiwanie przejazdów jako pasażer

System będzie składał się z 4 komponentów:

- bazy danych
- web serwisu
- aplikacji moblinej
- aplikacji internetowej

1.2 Przegląd wybranych rozwiązań dostępnych na rynku

Na rynku dostępne są aplikacje oferujące zbliżone funkcjonalności do projektowanego systemu:

- **Uber** – zamawianie przejazdu na żądanie. Są dwa typy użytkowników z oddzielnymi procesami rejestracji. Użytkownicy, którzy szukają środka transportu, w aplikacji widzą aktywnych kierowców oraz ich położenie na mapie miasta. Użytkownik po wyborze kierowcy, kontaktuje się z nim za pomocą wiadomości w aplikacji dokąd chce pojechać oraz uzgadnia cenę przejazdu. Uber zapewnia system płatności, część zapłaty za przejazd jest potrącana przez aplikacje. Dostępna jest aplikacja mobilna dla Androida, iOSa oraz Windows Phone.

- **Jakdojade** – aplikacja, służąca do wyszukiwania połączeń publicznymi środkami transportu między interesującymi nas miejscami w obrębie maista. Możemy wybrać godzinę, o której chcemy wyjechać lub na którą godzinę chcemy dotrzeć na miejsce docelowe. Wyszukiwarka wyszukuje pełne połączenie między wybranymi miejscami, uwzględniając złożenie pełnej trasy z kilku różnych przejazdów oraz drogi pieszo. Są do wyboru 3 tryby wyszukiwania połaczenia:

  - Wygodne – szukana jest trasa z jak najmniejszą ilością przesiadek oraz z jak najkrótszą odległością do przejścia pieszo
  - Optymalna – wyszukiwarka szuka połączeń o zrównoważonym czasie podróży do jej wygody
  - Ekspresowa – wyszukiwana jest jak najszybsza trasa

  Aplikacja ma swoją wersje na przeglądarkę oraz na systemy mobilne Android, iOS i Windows Phone.

- **Blablacar** – aplikacja do organizowania wspólnych przejazdów między miastami. Można dodawać przejazdy jednorazowe lub cykliczne, których trasa jest nie dłuższa niż 75km. Użytkownik, który chce oferować przejazdy, musi najpierw dodać swój samochód (markę, wyposażenie), oraz określić reguły obowiązujące w jego samochodzie (np. czy zgadza się na przewóz zwierząt). Miejsca w samochodzie można rezerwować automatycznie online lub manualnie, poprzez rozmowę telefoniczną z kierowcą, który następnie aktualizuje ilość dostępnych miejsc w samochodzie. Użytkownicy mogę nawzajem siebie oceniać, nie jest do tego wymagany wspólnie odbyty przejazd. Jedyne zabezpieczenie przed nieprawdziwymi komentarzami to możliwość odpowiedzenia na nie. Z innymi użytkownikami można komunikować się poprzez wewnętrzny system wiadomości. Użytkowników można wyszukać jedynie po ich numerze telefonu. Podczas dodawania nowej oferty przejazdu można określić pośrednie miasta podróży, w których kierowca zgadza się wysadzić pasażerów. Korzystanie z aplikacji jest w pełni darmowe, jednak za przejazdy kierowcy mogą żądać opłaty. Aplikacja nie posiada żadnego systemu płatności, rozliczenie między kierowcą, a pasażerami odbywa się podczas podróży poza systemem. Blablacar oprócz wersji przeglądarkowej posiada wersje na urządzenia mobilne z system iOS lub Android.

- **Otodojazd** – Aplikacja oferuje organizacje wspólnych dojazdów na uczelnie oraz do pracy. Kierowca dodając przejazd musi podać szereg parametrów:
  - punkt początkowy
  - punkt końcowy
  - stacje pośrednie
  - w jakie dni oferuje wspólną podróż
  - czy jego przejazdy są cykliczne
  - o której godzinie wyjeżdża z miejsca początkowego
  - ile ma wolnych miejsc w samochodzie
  - opłatę za dojazd

  Trasy można wyszukiwać podając samo miasto lub punkt początkowy i końcowy podróży. Wyszukiwarka wyszukuje trasy analizując cały ich przebieg, nie tylko punkty krańcowe.

Tablica 1.1 Porównanie funkcjonalności istniejących aplikacji z projektowanym systemem

	Uber	Jakdojade	Blablacar	Otodojazd	Proj. system
Dodawanie przejazdów	-	-	+	+	+
Zamawianie przejazdów	+	-	-	-	-
Określenie warunków przejazdu	+	-	+	+	+
Wyszukiwanie przejazdów	-	+	+	+	+
Łączenie przejazdów	-	+	-	-	+
Podróże między miastami	-	-	+	-	-
Podróże w obrębie miasta	+	+	-	+	+
Rezerwacja online przejazdu	+	-	+	-	+
Informacja o położeniu kierowcy	+	-	-	-	+
Informacja o położeniu pasażera	+	-	-	-	+
Nawigacja do przystanku	-	+	-	-	+
Wersja mobilna aplikacji	+	+	+	+	+
Wersja internetowa aplikacji	+	+	+	+	+
Komentowanie kierowców	+	-	+	+	+

1.3 Możliwe warianty systemu *Jedźmy razem*

Główne funkcjonalności systemu *Jedźmy razem* można zrealizować na kilka sposobów. Poniżej przedstawiono trzy warianty systemu, w każdym z nich inaczej zrealizowane jest wyszukiwanie przejazdów, co powoduje inną reprezentacje połączeń w systemie.

Wariant A. W systemie *Jedźmy razem* miasto zapisane jest jako graf skierowany, ważony. Wierzchołkami grafu są lokalizacje w mieście gdzie możliwe jest zatrzymanie się samochodem i zabranie/zostawienie pasażerów. Wagą każdej krawędzi jest średni czas potrzebny na przejazd samochodem między lokalizacjami zapisanymi w wierzchołkach. Kierowca podczas tworzenia przejazdu podaje punkt początkowy, punkt końcowy, proponowane punkty pośrednie oraz określa maksymalny czas jaki jest gotowy poświęcić na zjechanie ze swojej głównej trasy. Trasa na stałe zapisana jest jedynie jako punkt początkowy i końcowy. Gdy pasażerowie wyszukują przejazdu, modyfikowane są trasy kierowców, nie wydłużając ich ponad maksymalny czas podróży podany przez kierowcę. Trasa zapisana jest jako ścieżka w grafie.

Zalety:

- Elastyczne dopasowanie trasy kierowcy do miejsc gdzie znajdują się pasażerowie.
- Reprezentacja tras w formie grafu, może ułatwić wyszukiwanie przejazdów dzięki wykorzystaniu algorytmów wyszukujących ścieżki w grafie.
- Przy wyszukiwaniu złożonych tras, połączenia między pośrednimi przejazdami są zdefiniowane z góry za pomocą grafu miasta.

Wady:

- W systemie trzeba zdefiniować graf dla całego miasta.
- Przed wdrożeniem systemu w innych miastach, konieczne jest definiowanie grafów dla nowych miast.

Wariant B. W systemie nie jest zapisana siatka połączeń dla miasta. Trasy planowane są przez zewnętrzny serwis, który przetłumaczy nazwy miejsc podanych przez kierowce na współrzędne geograficzne. Kierowca jest odpowiedzialny za wprowadzenie miejsc gdzie może zatrzymać się samochodem. Kierowca podaje punkt początkowy, punkty pośrednie oraz punkt końcowy. Przejazdy wybierane są na podstawie odległości od punktów pasażera.

Zalety:

- System nie musi zajmować się planowaniem trasy kierowcy.
- Zastosowanie systemu dla innych miast nie wymaga zbyt dużo dodatkowej pracy.

Wady:

- Przy wyszukiwaniu złożonych tras, konieczne jest za każdym razem sprawdzanie, które przejazdy pośrednie znajdują się blisko siebie.

Wariant C. Trasa jest zapisana w systemie jako lista nazw miejsc wraz z godziną, które podał kierowca. Pasażer wprowadza ramy czasowe, w których zamierza podróżować. Zwrócone zostają wszystkie przejazdy znajdujące się w podanym przedziale czasowym.

Zalety:

- Przejazdów nie trzeba wyszukiwać względem lokalizacji co ułatwi implementacje.
- System nie musi zajmować się planowaniem trasy kierowcy.
- Zastosowanie systemu dla innych miast nie wymaga zbyt dużo dodatkowej pracy.

Wady:

- System bez wyszukiwania połączeń względem lokalizacji może okazać się mało funkcjonalny dla użytkownika.

Najbardziej funkcjonalnym wydaje się być wariant A. Jednak zdefiniowanie miasta w formie grafu i zapisanie go w bazie danych jest bardzo czasochłonnym przedsięwzięciem. Z tego powodu do wykonania podczas projektu inżynierskiego został wybrany wariant B., który posiada równie dużo funkcjonalności dla użytkownika co wariant A., ale dzięki pominięciu zapamiętania mapy miasta po stronie systemu jest możliwy do zrealizowania w czasie przeznaczonym na wykonanie projektu. Wariant C. został odrzucony z powodu zbyt małej złożoności.

Rozdział 2

Opis projektu realizowanego systemu

2.1 Struktura systemu

System zgodnie za schematem na rysunku [2.1] został podzielony na 3 warstwy:

- Warstwa danych (ang. data access layer) – to najniższy poziom systemu, w której znajduje się baza danych. W tej warstwie wyłącznie przechowywane są dane i udostępniane za pomocą mapowania relacji na obiekty. Wybrana baza danych to Postgresql.
- Warstwa logiki biznesowej (ang. buisness logic layer) – na tym poziomie dane są przetwarzane. W tej warstwie znajduje się web serwis wykonany w Ruby on Rails. Komunikacja z wyższą warstwą odbywać się będzie za pomocą protokołu http, a dane będą w formacie JSON.
- Warstwa prezentacji danych (ang. presentation layer) – poziom, w którym odbywa się komunikacja między użytkownikiem, a systemem. Aplikacje klienckie zawarte są w tej warstwie. Podczas projektu wykonane zostaną dwie aplikacje klienckie:
  - aplikacja internetowa utworzona za pomocą AngularJS
  - aplikacja mobilna na platformę Android

Rysunek 2.1 Struktura systemu

2.2 Funkcjonalność systemu

Na rysunkach [2.2] i [2.3] przedstawiono możliwe przypadki użycia systemu przez użytkowników, którzy mogą być podzieleni na dwie grupy według dwóch głównych funkcjonalności systemu:

- kierowcy - użytkownicy, którzy oferują przejazdy
- pasażerowie - użytkownicy, którzy wyszukują przejazdy

Podział ten ma charakter wyłącznie abstrakcyjny, dla zwiększenia czytelności diagramów. W systemie każdy użytkownik ma dostęp do tego samego zestawu funkcjonalności.

Rysunek 2.2 Diagram przypadków użycia systemu jako kierowca

Rysunek 2.3 Diagram przypadków użycia systemu jako pasażer

Poniżej przestawiono przykładowy scenariusz dla przypadku użycia. Scenariusze dla wszystkich przypadków użycia dostępne są w dodatku A.

Uruchom menadżer kierowcy

Aktorzy	Kierowca
Cel	Nawigowanie kierowcy wzdłuż trasy przejazdu oraz informowanie o pozycji pasażerów.
Warunki wstępne	Kierowca posiada przejazd w aktualnym czasie.
Warunki końcowe	Kierowca zakończył swój przejazd.
Scenariusz główny	1. Pobranie trasy przejazdu z bazy danych. 2. Zapisanie pozycji kierowcy w bazie danych. 3. Zaznaczenie pozycji kierowcy względem trasy przejazdu. 4. Pobranie pozycji pasażerów. 5. Zaznaczenie pozycji pasażerów względem trasy przejazdu. 6. Przejście do kroku drugiego.
Wyjątki	2.A. Nie można pobrać położenia kierowcy. 1.A.1. Wyświetlenie widoku z możliwością uruchomienia modułu gps 1.A.2. Przejście do kroku pierwszego. 4.A. Pozycja, któregoś z pasażerów jest nieznana. 4.A.1. Wyświetlenie informacji o nieokreślonej lokalizacji danego pasażera. 4.A.2. Przejście do kolejnego kroku. 4.B. Pozycja, wszystkich pasażerów jest nieznana. 4.B.1. Wyświetlenie informacji o nieokreślonej lokalizacji pasażerów. 4.B.2. Przejście do kroku pierwszego. 6.A. Zakończenie trasy przejazdu. 6.A.1. Zakończenie przypadku użycia.
Rozszerzenia	

2.3 Struktura bazy danych

Baza danych została zaprojektowana dla pełnego systemu. Typy kolumn, zgadzają się z typami dostępnymi w Postgresql, z wyjątkiem typu "point", który jest dostarczony przez bibliotekę Postgis.

Rysunek 2.4 Diagram ERD

Poszczególne tabele w bazie danych przedstawionej na rysunku [2.4]:

- **Users** – dane o użytkownikach:
  - username – nazwa użytkownika, pomocna przy kontaktach między użytkownikami. Nie musi być ona unikalna, ponieważ nie jest używana podczas logowania.
  - email – adres email, musi być unikalny, ponieważ jest używany podczas logowania.
  - phone – numer telefonu użytkownika, musi być unikalny, ponieważ każdy użytkownik powinien mieć tylko jedno konto.
  - password – zakodowane hasło użytkownika.
  - current_sign_in_at – data i godzina utworzenie aktualnej sesji.
  - current_sign_in_ip – adres ip, z którego utworzono aktualną sesje.
  - last_sign_in_at – data i godzina utworzenie poprzedniej sesji.
  - last_sign_in_ip – adres ip, z którego utworzono poprzednią sesje.
  - reset_password_sent_at – data i godzina wysłania tokenu do resetowania hasła, to pole może być puste.
  - reset_password_token – token do resetowania hasła, to pole może być puste.
  - sign_in_count – liczba utworzonych sesji przez użytkownika.
  - last_known_location – ostatnia znana lokalizacja użytkownika, to pole może być puste.
  - location_recorded_at – data i godzina zapisania ostatniej znanej lokalizacji, to pole może być puste.
  - vehicle_model_id – klucz obcy do tabeli "VehicleModels", określa markę i model samochodu przypisanego do użytkownika.
  - vehicle_production_year – rok produkcji samochodu.
  - vehicle_description – opis pojazdu, wprowadzony przez użytkownika, np. kolor, funkcjonalności.
- **VehicleModels** – dostępne modele samochodów:
  - model – model samochodu.
  - maker – marka pojazdu.
- **Journeys** – dane o przejazdach:
  - date – data przejazdu.
  - spaces – ilość oferowanych miejsc.
  - driver – klucz obcy do tabeli "Users", który określa kierowce przejazdu.
  - created_at – data utworzenia przejazdu.
  - updated_at – data ostatniej modyfikacji przejazdu.
- **Waypoints** – punkty pośrednie przejazdu:
  - time – godzina, o której kierowca znajdzie się w danym punkcie.
  - journeys_id – klucz obcy do tabeli "Journeys", określa do którego przejazdu należy punkt.
  - localizations_id – klucz obcy do tabeli Łocalizations", określa w jakiej lokalizacji geograficznym znajduje się punkt przejazdu.
- **Localizations** – lokalizacje geograficzne:
  - geographic_point – współrzędne geograficzne.
  - name – nazwa lokalizacji, np. ulica, nr domu(Grunwaldzka 18).
- **Passangers** – tabela łącząca użytkowników z przejazdami, start_waypoint i finish_waypoint muszą należeć do tego samego przejazdu:
  - user_id – klucz obcy do tabeli "Users", definiuje pasażera.
  - start_point – klucz obcy do tabeli "Waypoints", punkt początkowy podróży pasażera.

  - finish_point – klucz obcy do tabeli "Waypoints", punkt końcowy podróży pasażera.

- **Comments** – komentarze dotyczące konkretnych przejazdów:
  - mark – ocena podróży.
  - comment – treść komentarza na temat przejazdu.
  - passenger_id – klucz obcy do tabeli "Passengers", definiuje przejazd i pasażera, którego dotyczy komentarz.
  - is_from_driver – jeśli wartość jest prawdą, to oznacza, że komentarz wystawił kierowca, w przeciwny razie komentarz jest od pasażera.
  - created_at – data i godzina utworzenia komentarza.
- **Messages** – wiadomości użytkowników:
  - from – klucz obcy do tabeli "Users", oznacza nadawce wiadomości.
  - message – treść wiadomości.
  - created_at – data i godzina utworzenia wiadomości.
- **Recivers** – odbiorcy wiadomości:
  - to – klucz obcy do tabeli "Users", oznacza odbiorce wiadomości.
  - read_at – data i godzina otworzenia wiadomości przez odbiorce, początkowo nie ma wartości.
  - message_id – klucz obcy do tabeli "Messages".

2.4 Zakres projektu

Z powodu ograniczenia czasowego, do zaimplementowania w ramach projektu wybrano kilka przypadków użycia przedstawionych na rys. [2.5].

Rysunek 2.5 Diagram przypadków użycia do zaimplementowania w projekcie

2.5 Algorytm wyszukiwania przejazdów

Na wejściu algorytmu do wyszukiwania przejazdów mamy następujące dane:

- p_p – lokalizacja początku podróży pasażera
- p_k – lokalizacja końca podróży pasażera
- t_p – godzina rozpoczęcia podróży

Parametrami algorytmu są:

- Δ_t – maksymalne opóźnienie rozpoczęcia podróży
- k_{max} – maksymalna ilość przejazdów pośrednich
- d_{max} – maksymalna odległość między dwoma punktami, przy której stwierdzamy, że dwa przejazdy z tymi punktami są ze sobą połączone

Pomocnicze funkcje użyte na schematach blokowych:

- size – zwraca rozmiar listy
- push(element) – wstawia podany element na koniec listy
- last – zwraca ostatni element listy
- connect(przejazd, lokalizacja) – zwraca wartość null, jeśli przejazd nie zawiera punktu p, takiego, że odległość między punktem p, a podaną lokalizacją jest mniejsza od d_{max}, w przeciwnym razie zwraca punkt p
- find_connection(kurs, lista_przejazdow) – zwraca przejazd z podanej listy przejazdów, między którym istnieje połączenie z podanym kursem. Jeśli taki przejazd nie istnieje, zwraca wartość null.

Rysunek 2.6 Schemat blokowy algorytmu do wyszukiwania przejazdów

Ogólny schemat blokowy algorytmu został przedstawiony na rysunku [2.6]. Pierwszym krokiem jest wczytanie z bazy danych przejazdów występujących w zadanym czasie. Następnie można wydzielić dwa główne etapy algorytmu:

ETAP I – podzielenie wczytanych przejazdów na 4 listy:

- g_1 - przejazdy bezpośrednie
- g_2 - przejazdy znajdujące się w otoczeniu punktu początkowego p_p
- g_3 - przejazdy znajdujące się w otoczeniu punktu końcowego p_k
- g_4 - reszta przejazdów

Rysunek 2.7 Schemat blokowy etapu pierwszego algorytmu do wyszukiwania przejazdów

ETAP II – wyznaczenie tras pośrednich złożonych z przejazdów z list g_2, g_3, g_4.

Rysunek 2.8 Schemat blokowy etapu drugiego algorytmu do wyszukiwania przejazdów

Podczas implementacji algorytmu w projekcie przyjęto następujące parametry algorytmu:

- $\Delta_t = 2h$
- $k_{max} = 4$
- $d_{max} = 1000m$

Złożoność obliczeniowa algorytmu w notacji $O[1]$ wynosi $O(n^{k_max})$, gdzie ń"to liczba przejazdów w danym przedziale czasowym. Aby zmniejszyć złożoność obliczeniową algorytmu można by zastosować heurystyki, ograniczające zbiór przeglądanych rozwiązań, np.:

- jeśli przejazd nie jest przejazdem bezpośrednim, punktów przesiadkowych szukać wyłącznie w m najdłuższych przejazdów, gdzie prawdopodobieństwo znalezienia punktu przesiadkowego jest większe z powodu większej ilości punktów przejazdu
- na podstawie przejazdów zapisanych w bazie danych typować obszary o największej ilości przejazdów. Trasy z przesiadkami wyszukiwać wyłącznie wśród przejazdów przebiegających przez te obszary

Rozdział 3

Wykorzystane technologie

3.1 Baza danych

W projekcie wykorzystano relacyjną bazę danych PostgreSQL z dodatkiem Postgis, który pozwala operować na danych przestrzennych takich jak np.:

- Lokalizacja geograficzna - Point
- Trasa - Polyline
- Obszar geograficzny - Polygon

Dane tych typów są przechowywane w formie binarnej. Oprócz zapisania informacji w oszczędnym formacie, możliwe jest sprawne wykonywanie zapytań z warunkami przestrzennymi np. wczytać wszystkie punkty odległe od zadanego punktu nie więcej niż zadana wartość. Przykład takiego zapytania widoczny jest w fragmencie kodu [3.1, którego wynikiem są wszystkich rekordy z tabeli "waypoints", gdzie odległość punktu z kolumny "point"do punkt (51.12312, 17.223104) jest mniejsza niż 1000m.

Fragment kodu 3.1 Przykład zapytania SQL z użyciem operacji na danych przestrzennych

```
SELECT * FROM waypoints WHERE
ST_Distance(waypoints.point, POINT(51.12312 17.223104)) < 1000;
```

3.2 Web serwis

Do komunikacji między web serwisem, a aplikacjami klienckimi zostanie wykorzystana architektura RESTful API, która musi trzymać się przynajmniej 3 prostych reguł[3]:

- Bazować na adresach URI
- Reprezentować dane z użyciem internetowego nośnika danych, najczęściej jest to JSON
- Używać podstawowych metod HTTP:
  - GET - pobieranie danych
  - POST - tworzenie danych
  - PUT - modyfikowanie danych
  - DELETE - usuwanie danych

Do zaimplementowania aplikacji serwerowej wykorzystany zostanie framework Ruby on Rails, który jest oparty o wzorzec projektowy Model-Widok-Kontroler[6]. Aplikacja oparta na tym wzorcu jest podzielona na 3 części:

- Modele - reprezentują logikę biznesową. Tutaj znajdują się wszelkie obiekty, które służą do wykonywania wszelkich operacji związanych z implementacją funkcjonalności aplikacji.
- Widoki - służą do prezentowania danych. W projektowanej aplikacji, ta część nie zostanie wykorzystana, ponieważ rolę widoków przejmą aplikacje klienckie.
- Kontrolery - obsługują zapytania użytkownika. Nie przetwarzają żadnych danych, jedynie odbierają parametry, przekazują je do odpowiedniego modelu i zwracają odpowiedź modelu.

W projekcie wykorzystana zostanie ostatnia stabilna wersja rubiego 2.2.2 z najnowszą stabilną wersją frameworka rails 4.2.3.

3.3 Aplikacja internetowa

Aplikacja internetowa zostanie stworzona we frameworku języka JavaScript AngularJS. AngularJS korzysta z wzorca projektowego Model-Widok-Kontroler, oddzielając warstwę prezentacji od logiki biznesowej. Framework rozszerza standardowe możliwości HTMLa m. in. dzięki wiązaniom dwukierunkowym(ang. two-way-binding). Kompletną wiedzę na temat zasad działania i zastosowania AngularJS można zdobyć z książki *ng-book*[4].

W projekcie AngularJS zostanie użyty z językiem CoffesScript, który jest kompilowany do JavaScriptu i w pełni kompatybilny z tym językiem. Widoki zostaną napisane za pomocą języka Slim, który jest kompilowany do HTMLa, a style widoków zostaną napisane z wykorzystaniem języka SASS, który jest interpretowany do CSSa.

3.4 Aplikacja mobilna

Aplikacja mobilna zostanie stworzona na platformę Android. W tym celu zostanie wykorzystana paczka narzędzi programistycznych Android SDK. Minimalną wersją systemu wymaganą do poprawnego działania aplikacji będzie Android 4.0.3 (API wersja 15). Dzięki temu aplikacja będzie wspierana dla około 96%[1] telefonów komórkowych spośród wszystkich z systemem Android.

[1] Dane z dnia 2 listopada 2015 roku, dostępne na stronie http://developer.android.com/about/dashboards/index.html

Rozdział 4

Implementacja

4.1 Web serwis

4.1.1 Tworzenie przejazdu

Tworzenie przejazdu rozpoczyna się od wysłania zapytania POST na adres */journeys*. Adres ten jest obsługiwany przez metodę *create* z kontrolera *JourneysController* widoczną w fragmencie kodu [4.1]

Fragment kodu 4.1 Obsługa zapytania POST na adres /journeys

```
def create
  @journey = Journey.create_with_path(journey_params, current_user)
  render json: { status: :created, journey: @journey }
rescue StandardError => e
  render json: { status: :unprocessable_entity, error: e.to_s }
end
```

Podczas tworzenia przejazdu, parametry przekazane w zapytaniu, przepuszczane są przez metodę *journey_params* widoczną w poniższym kodzie źródłowym. Metoda ta ma na celu odfiltrowanie wymaganych parametrów od pozostałych, aby uniknąć przetwarzania nie pożądanych danych.

Fragment kodu 4.2 Odfiltrowanie wymaganych parametrów

```
def journey_params
  params.require(:journey)
    .permit(:date, :spaces, path: [:time, :name, point: []])
end
```

Dane do bazy danych zapisywane są poprzez wywołanie metody *Journey.create_with_path(journey_params, current_user)*, która jest zamieszczona w fragmencie kodu [4.3]. Wszystkie zapytania do bazy danych w bloku *Journey.transaction do .. end* zostaną wykonane w jednym połączeniu z bazą danych. Zwiększa to wydajność aplikacji oraz chroni przed utratą integralności danych w bazie. W przypadku niepowodzenia jednego z zapytań, cała transakcja jest wycofywana.

Fragment kodu 4.3 Zapisanie przejazdu w bazie danych

```
def self.create_with_path(journey_with_path, user)
  Journey.transaction do
    journey = Journey.create!(date: journey_with_path[:date],
      spaces: journey_with_path[:spaces],
      driver: user)
    Waypoint.create_from_array(journey_with_path[:path], journey)
    journey
  end
end
```

4.1.2 Wyszukiwanie przejazdów

Zgodnie z algorytmem opisanym w rozdziale 2.5 główna funkcja do wyszukiwania tras *get_journeys_in_period*, wywołuje kolejno 3 funkcje:

- *get_journeys_in_period* - wczytanie z bazy danych przejazdów odbywających się w ciągu 2 godzin od planowanego przez pasażera rozpoczęcia podróży
- *sort_journeys* - podzielenie przejazdów na 4 grupy, ze względu na ich relacje z punktem początkowym i punktem końcowym podróży pasażera
- *get_matched_journeys_from_sorted_journeys* - wyznaczenie tras, łączących lokalizacje początkową i cel podróży

Fragment kodu 4.4 Wyszukanie przejazdów o zadanych parametrach

```
def self.search_journeys(parameters)
  candidates = Journey.get_journeys_in_period(parameters[:start_time],
    parameters[:date])
   sorted_js = Journey.sort_journeys(candidates, parameters)
   journeys = Journey.get_matched_journeys_from_sorted_journeys(sorted_js)
   Journey.sort_and_format_response(journeys)
end
```

Na końcu znalezione przejazdy są sortowane według godziny rozpoczęcia i przetwarzane do formatu, który może być zwrócony w zapytaniu http.

4.2 Aplikacja internetowa

4.2.1 Zapytania do API

W AngularJS wszystkie zapytania http wykonywane są w tle, dzięki temu strona jest bardziej dynamiczna, wiele akcji można wykonywać równolegle. Przykład takich zapytań przedstawiono w fragmencie kodu [4.5]. Składnia jest bardzo intuicyjna, obiekt *$http* posiada metody odpowiadające konkretnym typom zapytań. Pierwszym argumentem każdej z tych metod jest adres, na który chcemy wysłać zapytanie, a drugim jest hash z parametrami do przesłania.

Fragment kodu 4.5 Zapytania do API

```
angular.module('JedzmyrazemApp').factory 'Journey', ($http) ->
  createJourney: (journey) ->
    $http.post('/journeys.json', {journey: journey})
  searchJourney: (params) ->
    $http.get('/journeys.json', {params: params})
```

Przy wywołaniu funkcji, która zawiera takie zapytanie można zdefiniować osobne akcje w przypadku sukcesu(funkcja ŚuccessÇ) lub porażki(funkcja ęrrorÇ) zapytania. Jedna z tych akcji zostanie wywołana dopiero po zakończeniu zapytania, które wykonywane jest w tle. Przykład jest zamieszczony w fragmencie kodu 4.6

Fragment kodu 4.6 Obsługa zapytania asynchronicznego

```
$scope.search = () ->
  if checkParameters()
    params = {date: moment($scope.dt).format("YYYY-MM-DD"),
      start_time: moment($scope.time).format("HH:mm"),
      start_lat: $scope.startPlace.geometry.location.lat()
      start_lng: $scope.startPlace.geometry.location.lng(),
      finish_lat: $scope.finishPlace.geometry.location.lat(),
      finish_lng: $scope.finishPlace.geometry.location.lng()}
    Journey.searchJourney(params).success (data) ->
      $scope.journeys = data.journey
      if data.journey.length < 1
        toastr.warning('Nie ma żadnych przejazdów o tych parametrach.',
        'Przykro nam')
    .error (data) ->
      toastr.error('Spróbuj ponownie za chwilę.', 'Wystąpił błąd')
```

4.2.2 Walidacja formularzy

Dzięki wiązaniom dwukierunkowym, dane można walidować już w trakcie ich wpisywanie, zamiast dopiero przy próbie ich wysłania. W fragmencie kodu [4.7] pokazana jest dyrektywa *$watch*, która jest wywoływana za każdym razem gdy zmieni się jej argument. Do dyrektywy przekazywana jest poprzednia i nowa wartość argumentu.

Fragment kodu 4.7 Walidowanie formatu email

```
$scope.$watch 'user.email', (newValue, oldValue) ->
  re =
    /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})
    \.([a-z]{2,6}(?:\.[a-z]{2})?)$/i
  if !re.test(newValue) && typeof(newValue) != 'undefined' && newValue !=''
    $scope.validate.email_format = true
  else
    $scope.validate.email_format = false
```

W przypadku, kiedy wprowadzony email jest niezgodny z wyrażeniem regularnym opisującym format adresu, zmienna *$scope.validate.email_format* ustawiona jest na true. Każda zmienna zdefiniowana w obrębie dyrektywy *$scope* współdzielona jest między widokiem i kontrolerem. Dzięki temu w przypadku wykrycia niepoprawnego adresu email, pokazywane jest stosowne powiadomienie w formularzu za pomocą dyrektywy *ng-show*, która pokazuje przypisany do niej frament widoku tylko, jeśli, jej argument jest prawdą, co można zobaczyć w poniższym kodzie źródłowym.

Fragment kodu 4.8 Walidowanie foramtu email

```
#password.input-group
  span.input-group-addon
    i.icon_key_alt
  input.form-control ng-model="user.password" placeholder="Hasło" type="password"
  div ng-show="validate.password_require"
    label.error for="password" ng-show="validate.password_require"
    | Pole hasło jest wymagane
    br
  label.error for="password" ng-show="validate.password_format"
    | Hasło musi mieć conajmniej 8 znaków
```

4.2.3 Stworzone widoki

Pierwszą funkcjonalnością po wejściu na stronę aplikacji jest wyszukiwanie przejazdów. Aby wyszukać daną trasę należy wypełnić wyświetlone pola. Dane, które użytkownik musi wprowadzić to lokalizacja początkowa, cel podróżować oraz data i godzina rozpoczęcia podróży.

Rysunek 4.1 Wyszukiwanie połączeń

Aby użytkownik zaistniał w systemie, należy dokonać poprawnej rejestracji, podać w formularzu adres e-mail, nazwę użytkownika, numer telefonu oraz login i hasło. Email oraz numer telefonu muszą być unikalne, a hasło musi mieć co najmniej 8 znaków.

Rysunek 4.2 Tworzenie konta w systemie

Po poprawnym podaniu danych, konto zostaje utworzone, a użytkownik jest automatycznie zalogowany na stronę internetową.

Rysunek 4.3 Widok systemu po zalogowaniu

Po wykonaniu czynności wyloguj, klient może ponownie się zalogować do systemu.

Rysunek 4.4 Widok logowania do systemu

W przypadku kliknięcia na odnośnik Ńie pamiętasz hasła?", użytkownik zostaje przekierowany na widok, z którego może wysłać sobie instrukcje z resetowaniem hasła.

Rysunek 4.5 Wysłanie maila z instrukcją do zresetowania hasła

Po chwili czasu, na ustawionego maila powinna przyjść wiadomość o zmianie hasła.

Rysunek 4.6 Email z instrukcją resetowania hasła

Po kliknięciu na link dot. zmiany hasła użytkownik zostanie przekierowany do formularza, w którym może wprowadzić nowe hasło.

Rysunek 4.7 Widok do wprowadzenia nowego hasła

Po zalogowaniu użytkownik może edytować dane swojego konta.

Rysunek 4.8 Edytowanie danych własnego konta

Zalogowany użytkownik może również dodawać przejazdy.

Rysunek 4.9 Dodawanie nowego przejazdu

4.3 Aplikacja mobilna

4.3.1 Utrzymywanie sesji użytkownika

W systemie sesja użytkownika utrzymywana jest za pomocą unikalnych tokenów dla użytkowników przechowywanych w ciasteczkach, jest to domyślny mechanizm używany w przeglądarkach internetowych. Aby ten mechanizm zadziałał w aplikacji mobilnej, należy zapisywać otrzymane ciasteczka wysyłane razem z odpowiedzią web serwisu oraz załączać je w kolejnych zapytaniach do web serwisu. Tą funkcjonalność udostępnia biblioteka *loopj*. Należy utworzyć obiekt *PersistentCookieStore* i wstrzyknąć go do obiektu klienta http co przedstawia fragment kodu 4.9. Dla usprawnienia zapamiętania sesji w aplikacji mobilnej, w pamięci podręcznej telefonu zapamiętywany jest stan sesji użytkownika. Dzięki temu, użytkownik posiada ważną sesje do czasu wyczyszczenia pamięci podręcznej aplikacji i nie jest konieczne łączenie z serwerem w celu sprawdzenia stanu sesji.

Fragment kodu 4.9 Automatyczna obsługa ciasteczek

```
cookieStore = new PersistentCookieStore(LoginActivity.this);
httpClient.setCookieStore(cookieStore);
```

Fragment kodu 4.10 Sprawdzenie czy użytkownik posiada aktywną sesje poprzez odczyt zmiennej łogged"przechowywanej w pamięci telefonu

```
SharedPreferences pref;
SharedPreferences.Editor editor;

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  pref = getSharedPreferences("userdetails", MODE_PRIVATE);

  String getStatus = pref.getString("logged", "");
  if(getStatus.equals("true")){
    Intent intent = new Intent(this, SearchActivity.class);
    startActivity(intent);
  }
```

Fragment kodu 4.11 Ustawienie zmiennej łoggedńa true po udanym logowaniu lub rejestracji

```
editor = pref.edit();
editor.putString("logged", "true").apply();
```

4.3.2 Komunikacja z web serwisem

Wszystkie dostępne adresy końcowe API zapisane są w zmiennej typu enum *RestApiUrl*, która dla danego adresu końcowego dodaje adres serwera. Dzięki takiemu rozwiązaniu zmiana adresu serwera lub adresu końcowego dla danego zapytania, wymaga zmiany wyłącznie w jednym miejscu w aplikacji mobilnej.

Fragment kodu 4.12 Zmienna enum, zwracająca pełny adres dla zapytania do API

```
public enum RestApiUrl {
  SIGN_IN("users/sign_in.json"),
  SIGN_UP("users.json"),
  SEARCH("journeys.json");

  private final String url;
  private final String BASE_URL = "http://www.jedzmyrazem.pl/";

  RestApiUrl(String url) {
    this.url = url;
  }

  public final String getUrl() { return  BASE_URL + url;}
}
```

Każde zapytanie odbywa się za pomocą klasy dziedziczącej po klasie *AsyncTask*, aby zadania z wysyłaniem zapytania http, które trwają kilka sekund były wykonywane w tle. Wykorzystane metody z klasy *AsyncTask* to *doInBackground*, w której umieszcza się kod do wykonania w tle oraz metoda *onPostExecute*, która wykonywana jest w głównym wątku aplikacji po zakończeniu metody *doInBackground*. Przykład tych metod dla wyszukiwania przejazdów przedstawiono w fragmentach kodu 4.13 i 4.14.

Fragment kodu 4.13 Metoda "doInBackground"dla wyszukiwania połączeń

```
protected String doInBackground(String... params) {
  String parameters = prepareStringParameters(params);
  if (parameters == null) return null;
  ServiceHandler sh = new ServiceHandler();
  return sh.makeServiceCall(RestApiUrl.SEARCH.getUrl(), ServiceHandler.GET, null, parameters);
}
```

Fragment kodu 4.14 Metoda ónPostExecute"dla wyszukiwania połączeń

```
protected void onPostExecute(String result) {
  parents.clear();
  ((MyExpandableListAdapter)expListView.getExpandableListAdapter()).notifyDataSetChanged();
  if (result == null) return;
  try {
    if (parseResponse(result)) return;
      ((MyExpandableListAdapter)expListView.getExpandableListAdapter()).notifyDataSetChanged();
  } catch (JSONException e) {
    e.printStackTrace();
  }
}
```

4.3.3 Google Places API

Podstawowe informacje o tym jak zastosować bibliotekę Google Places API zarówno w aplikacji mobilnej jak i internetowej można znaleźć w jej dokumentacji[2].

Do pokazywania podpowiedzi z proponowanymi lokalizacjami została napisana klasa *PlaceArrayAdapter* rozszerzająca standardową klasę *ArrayAdapter*. Metoda pobierająca podpowiedzi jest przedstawia fragment kodu 4.15.

Fragment kodu 4.15 Pobieranie podpowiedyi z Google Places API

```
private ArrayList<PlaceAutocomplete> getPredictions(CharSequence constraint) {
  if (mGoogleApiClient != null) {
    PendingResult<AutocompletePredictionBuffer> results =
      Places.GeoDataApi.getAutocompletePredictions(mGoogleApiClient, constraint.toString(), mBounds,
          mPlaceFilter);
    //ustawienie timeout'u na 60 sekund
    AutocompletePredictionBuffer autocompletePredictions = results.await(60, TimeUnit.SECONDS);
    final Status status = autocompletePredictions.getStatus();
    //jeśli zapytanie się nie powiedzie,
    //zostanie wyświetlony toast z~błędem
    if (!status.isSuccess()) {
      Toast.makeText(getContext(), "Error:_" + status.toString(), Toast.LENGTH_SHORT).show();
      autocompletePredictions.release();
      return null;
    }
    ArrayList resultList = formatArrayList(autocompletePredictions);
    autocompletePredictions.release();
    return resultList;
  }
  return null;
}
```

4.3.4 Odczyt lokalizacji GPS telefonu

Do pobierania lokalizacji została zaimplementowana osobna klasa *GPSTracker* rozszerzająca klasę *Service*. System Android nie pozwala na bezpośrednie pobranie aktualnej lokalizacji z modułu GPS telefonu. Można jedynie pobrać ostatnio zarejestrowaną lokalizacje. W wykonywanej aplikacji do wyszukiwania przejazdów chcemy użyć aktualnej pozycji użytkownika jako punktu startowego podróży. Zgodnie z przykładem z książki *Android. Poradnik programisty*[7] aby to osiągnąć należy wykonać dwa kroki:

1. Wysłać żądanie o zarejestrowanie lokalizacji

2. Pobrać ostatnią znaną lokalizacje

Kod realizujący te dwa kroki został przedstawiony poniżej.

Fragment kodu 4.16 Pobranie lokalizacji

```
public Location getLocation() {
  try {
    locationManager = (LocationManager) mContext.getSystemService(LOCATION_SERVICE);
    if (isGPSEnabled()) {
      //aktualizacja lokalizacji
      locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 60000, 10, this);
      if (locationManager != null) {
        //pobranie zaaktualizowanej lokalizacji
        location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
  return location;
}
```

4.3.5 Widoki aplikacji

Aplikacja mobilna posiada 3 proste widoki:

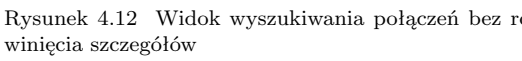
- Widok logowania
- Widok tworzenia konta
- Widok wyszukiwania przejazdów

Na ekranach łogowaniaóraz "tworzenia konta"wszystkie pola są wymagane i jest walidowana ich zawartość.

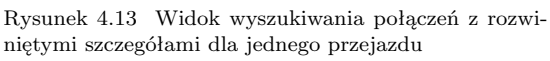
Rysunek 4.10 Widok logowania

Rysunek 4.11 Widok tworzenia konta

Na ekranie "wyszukiwania przejazdów", po kliknięciu na jednym ze znalezionych połączeń, są rozwijane szczegóły pośrednich przejazdów. Wszystkie przejazdy znajdują się na przewijanej liście.



Rysunek 4.12 Widok wyszukiwania połączeń bez rozwinięcia szczegółów



Rysunek 4.13 Widok wyszukiwania połączeń z rozwiniętymi szczegółami dla jednego przejazdu

W aplikacji mobilnej również zaimplementowanu użycie Google Places Api z podpowiadaniem wprowadzanej lokalizacji.

Rysunek 4.14 Podpowiadanie lokalizacji

Rozdział 5

Testowanie systemu

5.1 Statyczna analiza kodu

Podczas trwania projektu, została wykonywana statyczna analiza kodu. Dla każdej z części projektu użyto dedykowanych narzędzi:

a) Web serwis

- **Rubocop** - statyczna analiza kodu w języku Ruby.
- **Rcov** - mierzy pokrycie kodu testami jednostkowymi.

b) Aplikacja internetowa

- **Coffeelint** - analizuje kod w języku coffeescript.

c) Aplikacja mobilna

- **Findbug** - służy do analizy aplikacji napisanych w Javie.
- **Androidlint** - sprawdza stosowanie złych praktyk w aplikacjach na platformę Android.

Według Roberta Martina[5] dobre oprogramowanie cechuje się przejrzystym kodem, w którym łatwiej znaleźć potencjalne miejsca błędów. Dzięki zastosowaniu powyższych narzędzi, poprzez praktykę można nauczyć się zasad, wzorców i heurestyk używanych w danym język programowania i sprawdzać czy zdobyta wiedza jest wykorzystywana, co przekłada się na jakość i przejrzystość stworzonego kodu źródłowego.

5.2 Testy jednostkowe

5.2.1 Web serwis

Do testów jednostkowych aplikacji stworzonej w Ruby on Rails wykorzytsano bibliotekę RSpec. Testować można zarówno metody kontrolerów, jak i modeli. RSpec do testów wykorzystuje osobną, testową bazę danych, która jest tworzona przed wykonywaniem testów i czyszczona po ich zakończeniu(można ją również czyścić po każdym teście). Składnia testów jest bardzo prosta i intuicyjna, testy podzielone są na 3 części:

1. „describe” – tu definiujemy jaką funkcje/klasę/moduł ma sprawdzać dany zbiór testów
2. „context” – to miejsce służy do określenia warunków testu, ta część jest opcjonalna
3. „it” – mówi nam jak powinien się zachować testowany moduł

Fragment kodu 5.1 Przykład testów dla metody kontrolera

```
RSpec.describe JourneysController, type: :controller do
  before(:each) do
    user = FactoryGirl.build(:user)
    allow_any_instance_of(JourneysController).to receive(:current_user)
      .and_return(user)
  end
  describe 'POST_#create' do
    context 'with_valid_attributes' do
      it 'creates_the_journey' do
        post :create, journey: FactoryGirl.attributes_for(:journey_with_path),
          format: :json
        expect(Journey.count).to eq(1)
      end
    end
    context 'with_failed_waypoints_saving' do
      it 'won\'t_save_trip' do
        allow(Waypoint).to receive(:create_from_array).and_raise('Error')
        post :create, journey: FactoryGirl.attributes_for(:journey_with_path),
          format: :json
        expect(Journey.count).to eq(0)
      end
    end
end
```

W fragmencie kodu [5.1] przed każdym testem jest tworzony mock funkcji *current_user*, która zwraca aktualnie zalogowanego użytkownika. Do stworzenia instancji modelu *User* została użyta biblioteka *FactoryGirl*, która implementuje wzorzec "fabryki", ułatwiający tworzenie obiektów w testach o określonych parametrach. Przykład definicji fabryki został zamieszczony w poniższym kodzie źródłowym.

Fragment kodu 5.2 Fabryka do tworzenia obiektu typu "User"

```
FactoryGirl.define do
  factory :user do
    username 'Test_user'
    password '12345678'
    email 't@t.pl'
    phone 123_456_789
  end
end
```

Pokrycie kodu testami było mierzone za pomocą narzędzia RCov podczas każdej zmiany na repozytorium kodu źródłowego, ostatecznie wyniosło ono 100%, czyli każda funkcja jest wywoływana podczas testów.

Rysunek 5.1 Historia wyników analizy

5.2.2 Aplikacja internetowa

Do przetestowania aplikacji wykonanej we frameworku AngularJS wymagane jest kilka dodatkowych narzędzi:

- "Teaspoon- tworzy testowy serwer z środowiskiem uruchomieniowym dla JavaScriptu
- "Jasmine- biblioteka do tworzenia testów jednostkowych dla aplikacji napisanych w JavaScript'cie
- Angular Mocks- biblioteka do symulowania właściwości i działania aplikacji stworzonej przy pomocy AngularJS

Test składa się z takich samych części jak w przypadku RSpeca:

1. „describe"
2. „context"
3. „it"

Przed każdym testem dla kontrolera, należy go zainicjować, aby mieć dostęp do jego zmiennych, funkcji i dyrektyw. Przykład inicjalizacji kontrolera w teście przedstawiono w fragmencie kodu [5.3].

Fragment kodu 5.3 Inicjalizacja testów aplikacji internetowej

```
describe 'HomeCtrl', ( $scope)->
  $scope = null
  $controller = null
  beforeEach module('JedzmyrazemApp')

  beforeEach inject ($injector, _$q_) ->
    $scope = $injector.get('$rootScope').$new()
    $controller = $injector.get('$controller')
    $controller('HomeCtrl', {$scope: $scope })
```

W fragmencie kodu źródłowego [5.4] znajduje się prosty test, który sprawdza, czy po wywołaniu funkcji *$scope.clear()* zmienna *$scope.dt* jest równa *null*.

Fragment kodu 5.4 Test jednostkowy dla metody z aplikacji internetowej

```
describe 'clear', ->
  it 'clear $scope.dt var', ->
    $scope.dt = new Date(2014, 11, 15)
    $scope.clear()
    expect($scope.dt).toBeNull
```

5.2.3 Aplikacja mobilna

W aplikacji mobilnej do testów jednostkowych użyto dodatkowych bibliotek:

- JUnit - biblioteka do pisania testów jednostkowych dla aplikacji wykonanych w języku Java.
- Robolectric - zawiera narzędzia do symulowania platformy Android, dzięki temu w testach nie potrzeba używać emulatora lub rzeczywistego urządzenia.
- Mockito - moduł do tworzenia mocków części aplikacji androidowej.

Za pomocą adnotacji *@Before* można oznaczyć funkcje, które powinny być wykonane przed rozpoczęciem testów. W fragmencie kodu 5.5 przedstawiono czynności wykonywane przed wykonaniem testów dla aktywności *SearchActivity*.

Fragment kodu 5.5 Inicjalizowanie testów

```
@Before
public void setup() {

  //Stworzenie mocka dla Google Play Services, aby można było uruchomić
  //testy bez korzystania z zewnętrznego narzędzia
  ShadowApplication shadowApplication = Shadows.shadowOf(RuntimeEnvironment.application);
  shadowApplication.declareActionUnbindable("com.google.android.gms.analytics.service.START");
  ShadowGooglePlayServicesUtil.setIsGooglePlayServicesAvailable(ConnectionResult.API_UNAVAILABLE);

  activity = Robolectric.setupActivity(SearchActivity.class);
  //Stworzenie mocka dla klasy GPSTracker
  gpsTrackerMock = Mockito.mock(GPSTracker.class);
  activity.gps = gpsTrackerMock;
}
```

Przykładem testów dla aktywności wyszukiwania przejazdów, może być sprawdzenie, czy próbuje ona pobrać lokalizacje gps, jeśli użytkownik nie poda swojego położenia. Takie testy przedstawia poniższy fragment kodu.

Fragment kodu 5.6 Testy wywołania funkcji pobierającej lokalizacje

```
//test sprawdzający, czy lokalizacja jest pobierana z modułu gps,
//jeśli użytkownik jej nie wprowadził
@Test
public void searchCallGetLocationIfStarLocationIsNotSetted()
{
  activity.search(null);
  Mockito.verify(gpsTrackerMock, Mockito.times(1)).getLocation();
}
//test sprawdzający, czy lokalizacja nie jest pobierana z modułu gps,
//jeśli użytkownik ją wprowadził
@Test
public void searchNotCallGetLocationIfStartLocationIsSetted()
{
  activity.mAutocompleteTextViewSrc.setText("start");
  activity.start_location = new LatLng(52.00, 17.00);
  activity.search(null);
  Mockito.verify(gpsTrackerMock, Mockito.times(0)).getLocation();
}
```

Rozdział 6

Opis wdrożenia web serwisu i aplikacji internetowej

6.1 Uruchomienie aplikacji

Aby uruchomić aplikacje opartą o Ruby on Rails oraz AngularJS z bazą danych Postgresql + Postgis na serwerze z systemem Linux w środowkisku produkcyjnym należy wykonać następującą listę kroków:

1. Zainstalować narzędzia do obsługi bazy danych Postgresql:
   $ apt-get install postgresql-9.4 postgresql-client-9.4 postgresql-contrib-9.4 libpq-dev
   postgis2_93

2. Stworzyć super rolę w bazie danych, o nazwie takiej samej jak konto użytkownika, na którym zostanie uruchomiony serwer:
   $ psql
   $ createuser –superuser *USERNAME*

3. Zainstalować ruby i bibliotekę rails - np. przy pomocy
   RVM (Ruby Version Manager):
   $ gpg –keyserver\
   $ hkp://keys.gnupg.net –recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3 $ curl -sSL https://get.rvm.io — bash #instalacja rvm
   $ rvm install 2.2.2 #instalacja ruby w wersji 2.2.2
   $ rvm gemset create rails-4.2.3 #instalacja railsów w wersji 4.2.3
   $ rvm use 2.2.2@rails-4.2.3 #ustawienie zainstalowanych narzędzi

4. Pobrać wszystkie potrzebne biblioteki
   $ cd *PROJECT_DIR*
   $ bundle install

5. Stworzyć bazę danych i wprowadzić migracje
   $ RAILS_ENV=production rake db:create db:migrate

6. Skompilować pliki dodane do projektu jako assets (np. pliki .coffeescript, .slim). Ten krok nie jest konieczny do poprawnego działania aplikacji, ale zwiększa jej wydajność, ponieważ pliki assets nie będą kompilowane na żądanie podczas działania aplikacji.
   $ RAILS_ENV=production rake assets:precompile

7. Wygenerować i wyeksportować secret key:
   $ export SECRET_KEY_BASE=$(RAILS_ENV=production rake secret)

8. Uruchomić serwer:
   $ rails server -e production -p 80 -b 0.0.0.0

6.2 Automatyzacja aktualizacji oprogramowania

System od początku został uruchomiony na zewnętrznym serwerze i był automatycznie aktualizowany po każdej przetestowanej zmianie. Do tego celu został wykorzystany *Jenkins*, czyli narzędzie do automatyzacji integracji oprogramowania. Jenkins wykonywał dwa zadania:

1. „web_jedzmyrazem_tests” - zadanie budowane po każdej zmianie wprowadzonej na repozytorium. Podczas tego zadania wykonywane były testy i satatyczna analiza kodu, jeśli nie znaleziono, żadnych błędów, zadanie wyzwalało kolejne zadanie „web_jedzmyrazem_update”. Głównym krokiem tego zadania jest wykonanie skryptu widocznego na fragmencie kodu [6.1]

Fragment kodu 6.1 Skrypt do wykonywania testów aplikacji

```
#!/bin/bash -e
source "/usr/local/rvm/scripts/rvm"
rvm use 2.2.2@rails-4.2.3

set -x
sudo bundle install --without production
#usunięcie i~utworzenie nowej bazy danych
RAILS_ENV=test rake db:drop db:create db:migrate

#wykonanie testów jednostkowych railsów
RAILS_ENV=test rake ci:setup:rspec spec || true

#wykonanie statycznej analizy kodu railsów
bundle exec rubocop \
  --require rubocop/formatter/checkstyle_formatter \
  --format RuboCop::Formatter::CheckstyleFormatter -o\
    reports/xml/checkstyle-result.xml \
  --format html -o reports/html/index.html

#wykonanie statycznej analizy kodu aplikacji napisanej za pomocą AngularJS
coffeelint spec/ app/assets/javascripts/ --reporter checkstyle\
  > coffe_check.xml || true
#wykonanie testów aplikacji napisanej za pomocą AngularJS
RAILS_ENV=test teaspoon --format junit | tail -n+3\
  > spec/reports/teaspoon.xml || true
```

2. „web_jedzmyrazem_update” - aktualizacja wersji aplikacji na serwerze. Skrypt wykonywany przez to zadanie znajduje się w fragmencie kodu [6.2]

Fragment kodu 6.2 Skrypt do aktualizacji aplikacji na serwerze

```
#!/bin/bash -e
source "/usr/local/rvm/scripts/rvm"
rvm use 2.2.2@rails-4.2.3
export SECRET_KEY_BASE=$(RAILS_ENV=production rake secret)
set -x
cd /var/jedzmyrazem
#pobranie najnowszej wersji aplikacji
git pull origin master
#zainstalowanie brakujących bibliotek
RAILS_ENV=production bundle install
RAILS_ENV=production rake assets:precompile || true
RAILS_ENV=production rake db:migrate
#wyłączenie serwera, który już działa
if [ -e /var/jedzmyrazem/tmp/pids/server.pid ]; then
  kill -9 `cat /var/jedzmyrazem/tmp/pids/server.pid` || true
fi
#włączenie serwera
BUILD_ID=dontKillMe rails server -e production -p 80 -b 0.0.0.0 -d
```

Rozdział 7

Podsumowanie

7.1 Realizacja celu projektu

Cel projektu opisany w rozdziale 1.1 został w pełni zrealizowany. System jest w pełni funkcjonalny i działający. Można oferować przejazdy innym użytkownikom jako kierowca oraz wyszukiwać połączenia dodane przez innych użytkowników. Cały system składa się z zaplanowanych 4 części. Ponadto zakres projektu podany w rozdziale 2.4 został w pełni zaimplementowany.

7.2 Ocena jakości systemu

System został przetestowany przy pomocy testów jednostkowych sprawdzających poprawność działania pojedynczych elementów każdej z części projektu. Za pomocą tych testów stwierdzono, że każdy element działa poprawnie. Dodatkowo wszystkie błędy wskazane przez wykorzystywane narzędzia do statycznej analizy kodu zostały poprawione.

7.3 Kierunki rozwoju

W przyszłości w projekcie zostały do zaimplementowania pozostałe przypadki użycia systemu. Dodatkowo dzięki modułowej budowie systemu z osobną częścią realizującą REST API, można tworzyć aplikacje na inne platformy mobilne, np. iOS lub Windows Phone. Aby system można w pełni udostępnić dla użytkowników należałoby zwiększyć bezpieczeństwo za pomocą protokołu ssl, do szyfrowania przesyłanych danych między aplikacjami klienckimi, a web serwisem.

Bibliografia

[1] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., *Wprowadzenie do algorytmów*, The Massachusetts Institute of Technology, 1990

[2] *Google Places API*, dostępne pod adresem:
`https://developers.google.com/places`, aktualne na dzień 03.12.2015r.

[3] Kuri Abraham, *APIs on Rails*, 2014

[4] Lerner Ari, *ng-book. The Complete Book on AngularJS*, 2013

[5] Martin Robert C., *Czysty kod*, Pearson Education, 2009

[6] Ruby Sam, Thomas Dave, Hansson Heinemeier David, *Agile Web Development with Rails 4*, Pragmatic Programmers, 2013

[7] Wei-Meng Lee, *Android. Poradnik programisty*, Helion, 2013

Dodatek A

Scenariusze przypadków użycia

Utwórz konto	
Aktorzy	Pasażer, Kierowca
Cel	Utworzenie nowego użytkownika w bazie danych.
Warunki wstępne	Gość systemu nie posiada konta.
Warunki końcowe	w systemie zostało utworzone konto dla nowego użytkownika.
Scenariusz główny	1. Pobranie danych użytkownika z formularza rejestracji. 2. Walidacja wprowadzonych danych. 3. Utworzenie nowego użytkownika w bazie danych. 4. Wykonanie PU "Utwórz sesje".
Wyjątki	2.A. Dane wprowadzone przez użytkownika są niepoprawne. 2.A.1. Przekierowanie użytkownika na stronę rejestracji. 2.A.2. Wyświetlenie informacji o błędnych danych. 2.A.3. Przejście do kroku 1.
Rozszerzenia	

Zaloguj się	
Aktorzy	Pasażer, Kierowca
Cel	Autentykacja użytkownika.
Warunki wstępne	Użytkownik nie jest zalogowany do systemu.
Warunki końcowe	Użytkownik posiada aktywną sesje.
Scenariusz główny	1. Pobranie danych użytkownika z formularza logowania. 2. Wyszukanie użytkownika w bazie danych. 3. Wykonanie PU "Utwórz sesje".
Wyjątki	2.A. Użytkownik z podanymi danymi nie istnieje. 2.A.1. Przekierowanie użytkownika na stronę logowania. 2.A.2. Wyświetlenie informacji o błędnych danych. 2.A.3. Przejście do kroku 1.
Rozszerzenia	

Utwórz sesje

Aktorzy	Pasażer, Kierowca
Cel	Utworzenie sesji użytkownika.
Warunki wstępne	Użytkownik posiada konto.
Warunki końcowe	Sesja użytkownika jest aktywna.
Scenariusz główny	1. Wygenerowanie unikalnego tokenu dla użytkownika. 2. Zapisanie tokenu w ciasteczku.
Wyjątki	
Rozszerzenia	

Zakończ sesje

Aktorzy	Pasażer, Kierowca
Cel	Zakończenie sesji użytkownika.
Warunki wstępne	Użytkownik jest zalogowany do systemu.
Warunki końcowe	Sesja użytkownika jest nieaktywna.
Scenariusz główny	1. Usunięcia ciasteczka z tokenem sesji. 2. Przekierowanie użytkownika na główną stronę aplikacji.
Wyjątki	
Rozszerzenia	

Edytuj konto

Aktorzy	Pasażer, Kierowca
Cel	Edycja danych użytkownika.
Warunki wstępne	Użytkownik posiada aktywną sesje.
Warunki końcowe	Zmodyfikowane dane użytkownika są zapisane w bazie danych.
Scenariusz główny	1. Pobranie danych użytkownika z bazy danych. 2. Pobranie zmodyfikowanych danych użytkownika z formularza wraz z aktualnym hasłem. 3. Walidacja danych. 4. Zaktualizowanie danych użytkownika w bazie danych.
Wyjątki	3.A. Aktualne hasło podane przez użytkownika jest nieprawidłowe. 3.A.1. Przekierowanie użytkownika na stronę edycji danych. 3.A.2. Wyświetlenie informacji o błędnym haśle. 3.A.3. Przejście do kroku 1. 3.B. Wprowadzone dane nie przeszły walidacji. 3.B.1. Przekierowanie użytkownika na stronę edycji danych. 3.B.2. Wyświetlenie informacji o błędnych danych. 3.B.3. Przejście do kroku 1.
Rozszerzenia	

Zresetuj hasło

Aktorzy	Pasażer, Kierowca
Cel	Zmiana hasła użytkownika.
Warunki wstępne	Użytkownik nie jest zalogowany do systemu.
Warunki końcowe	Zapisanie nowego hasła w bazie danych.
Scenariusz główny	1. Pobranie wprowadzonego przez użytkownika adresu email. 2. Wyszukanie użytkownika o podanym adresie email. 3. Wygenerowanie bezpiecznego tokenu do zmiany hasła i zapisanie go w bazie danych w rekordzie konkretnego użytkownika. 4. Wysłanie emaila z wygenerowanym tokenem. 5. Pobranie z formularza tokenu, nowego hasła oraz potwierdzenia nowego hasła. 6. Walidacja danych. 7. Zapisanie nowego hasła. 8. Usunięcie tokenu do zmiany hasła z bazy danych dla podanego użytkownika.
Wyjątki	2.A. Użytkownik z podanym adresem email nie istnieje. 2.A.1. Wyświetlenie informacji o błędnym adresie email. 2.A.2. Przejście do kroku 1. 6.A. Wprowadzony token do zmiany hasła jest nieprawidłowy. 6.A.1. Wyświetlenie informacji o błędnym tokenie. 6.A.2. Przejście do kroku 5. 6.B. Wprowadzone hasło oraz powtórzenie hasła są od siebie różne. 6.B.1. Wyświetlenie informacji o niezgodnych hasłach. 6.B.2. Przejście do kroku 5.
Rozszerzenia	

Wyślij wiadomość do innego użytkownika.

Aktorzy	Kierowca, Pasażer
Cel	Wysłanie wiadomości.
Warunki wstępne	Użytkownik posiada aktywną sesje.
Warunki końcowe	Wiadomość jest zapisana w bazie danych.
Scenariusz główny	1. Pobranie wiadomości oraz identyfikatorów użytkowników wprowadzonych przez użytkownika. 2. Zapisanie danych.
Wyjątki i rozszerzenia	

Przejrzyj wiadomości.

Aktorzy	Kierowca, Pasażer
Cel	Wyświetlenie wiadomości danego użytkownika.
Warunki wstępne	Użytkownik posiada aktywną sesje.
Warunki końcowe	Wiadomości danego użytkownika są wyświetlone.
Scenariusz główny	1. Pobranie wiadomości danego użytkownika. 2. Wyświetlenie pobranych wiadomości. 3. Zapisanie aktualnej daty jako czas odczytu w wcześniej nieodczytanych wiadomościach.
Wyjątki	2.A. Użytkownik nie ma żadnych wiadomości. 2.A.1. Wyświetlenie informacji o braku wiadomości. 2.A.2. Zakończenie przypadku użycia.
Rozszerzenia	

Dodaj samochód

Aktorzy	Kierowca
Cel	Zapisanie danych na temat samochodu użytkownika w bazie danych.
Warunki wstępne	Użytkownik posiada aktywną sesje
Warunki końcowe	Użytkownik posiada przypisany samochód.
Scenariusz główny	1. Pobranie danych samochodu wprowadzonych przez użytkownika. 2. Walidacja danych. 3. Zapisanie danych.
Wyjątki	2.A. Wprowadzone dane są niepoprawne. 2.A.1. Wyświetlenie informacji o błędnych danych. 2.A.2. Przejście do kroku 1.
Rozszerzenia	

Edytuj samochód

Aktorzy	Kierowca
Cel	Zmodyfikowanie danych samochodu użytkownika.
Warunki wstępne	Użytkownik posiada aktywną sesje i ma przypisany samochód.
Warunki końcowe	Zmienione dane samochodu są zapisane w bazie danych.
Scenariusz główny	1. Wpisanie danych samochodu z bazy danych do formularza. 2. Pobranie danych samochodu wprowadzonych przez użytkownika. 3. Walidacja danych. 4. Zapisanie zmodyfikowanych danych.
Wyjątki	3.A. Wprowadzone dane są niepoprawne. 3.A.1. Wyświetlenie informacji o błędnych danych. 3.A.2. Przejście do kroku 1.
Rozszerzenia	

Utwórz nowy przejazd

Aktorzy	Kierowca
Cel	Dodanie nowego przejazdu do bazy danych.
Warunki wstępne	Użytkownik posiada aktywną sesje
Warunki końcowe	Nowy przejazd jest zapisany w bazie danych.
Scenariusz główny	1. Pobranie danych przejazdu wprowadzonych przez użytkownika. 2. Walidacja danych. 3. Utworzenie nowego przejazdu.
Wyjątki	2.A. Wprowadzone dane są niepoprawne. 2.A.1. Wyświetlenie informacji o błędnych danych. 2.A.2. Przejście do kroku 1.
Rozszerzenia	

Przejrzyj własne przejazdy

Aktorzy	Kierowca
Cel	Wyświetlenie przejazdów danego użytkownika.
Warunki wstępne	Użytkownik posiada aktywną sesje.
Warunki końcowe	Użytkownik może przejrzeć stan wszystkich swoich przejazdów.
Scenariusz główny	1. Pobranie przejazdów danego użytkownika. 2. Wyświetlenie pobranych przejazdów.
Wyjątki	
Rozszerzenia	• Anuluj swój przejazd • Edytuj swój przejazd • Oceń pasażera

Anuluj swój przejazd

Aktorzy	Kierowca
Cel	Anulowanie przejazdu.
Warunki wstępne	Użytkownik posiada aktywną sesje i wybrał przejazd z listy własnych przejazdów.
Warunki końcowe	Przejazd nie jest dostępny dla innych użytkowników.
Scenariusz główny	1. Ustawienie flagi „nieaktywny” w bazie danych w wybranym przejeździe. 2. Wysłanie powiadomienia do zapisanych pasażerów.
Wyjątki	
Rozszerzenia	

Edytuj swój przejazd

Aktorzy	Kierowca
Cel	Zmiana szczegółów przejazdu.
Warunki wstępne	Użytkownik posiada aktywną sesje i wybrał przejazd z listy własnych przejazdów.
Warunki końcowe	Zmodyfikowany przejazd jest zapisany w bazie danych.
Scenariusz główny	1. Ustawienie flagi „nieaktywny” w bazie danych w wybranym przejeździe. 2. Wysłanie powiadomienia do zapisanych pasażerów.
Wyjątki	2.A. Wprowadzone dane są niepoprawne. 2.A.1. Wyświetlenie informacji o błędnych danych. 2.A.2. Przejście do kroku 2.
Rozszerzenia	

Oceń pasażera

Aktorzy	Kierowca
Cel	Ocenienie pasażera
Warunki wstępne	Użytkownik zakończył przejazd jako kierowca.
Warunki końcowe	Ocena pasażera jest zapisana w bazie danych.
Scenariusz główny	1. Pobranie oceny oraz komentarza wprowadzonego przez użytkownika. 2. Zapisanie danych.
Wyjątki	
Rozszerzenia	

Uruchom menadżer kierowcy

Aktorzy	Kierowca
Cel	Nawigowanie kierowcy wzdłuż trasy przejazdu oraz informowanie o pozycji pasażerów.
Warunki wstępne	Kierowca posiada przejazd w aktualnym czasie.
Warunki końcowe	Kierowca zakończył swój przejazd.
Scenariusz główny	1. Pobranie trasy przejazdu z bazy danych. 2. Zapisanie pozycji kierowcy w bazie danych. 3. Zaznaczenie pozycji kierowcy względem trasy przejazdu. 4. Pobranie pozycji pasażerów. 5. Zaznaczenie pozycji pasażerów względem trasy przejazdu. 6. Przejście do kroku drugiego.
Wyjątki	2.A. Nie można pobrać położenia kierowcy. 1.A.1. Wyświetlenie widoku z możliwością uruchomienia modułu gps 1.A.2. Przejście do kroku pierwszego. 4.A. Pozycja, któregoś z pasażerów jest nieznana. 4.A.1. Wyświetlenie informacji o nieokreślonej lokalizacji danego pasażera. 4.A.2. Przejście do kolejnego kroku. 4.B. Pozycja, wszystkich pasażerów jest nieznana. 4.B.1. Wyświetlenie informacji o nieokreślonej lokalizacji pasażerów. 4.B.2. Przejście do kroku pierwszego. 6.A. Zakończenie trasy przejazdu. 6.A.1. Zakończenie przypadku użycia.
Rozszerzenia	

Wyszukaj przejazd

Aktorzy	Pasażer
Cel	Wyszukanie przejazdów o zadanych parametrach.
Warunki wstępne	Użytkownik posiada dostęp do systemu. i wybrał przejazd z listy własnych przejazdów.
Warunki końcowe	Wyświetlenie przejazdów o podanych parametrach.
Scenariusz główny	1. Pobranie z formularza podanych parametrów. 2. Wyszukanie w bazie danych przejazdów o zadanych kryteriach. 3. Wyświetlenie znalezionych połączeń wraz z przejazdami pośrednimi.
Wyjątki	2.A. Nie znaleziono przejazdów o zadanych kryteriach. 2.A.1. Wyświetlenie informacji o braku przejazdów o określonych parametrach.
Rozszerzenia	• Zapisz się na przejazd(y)

Zapisz się na przejazd(y)

Aktorzy	Pasażer
Cel	Zajęcie miejsca w wybranym przejeździe.
Warunki wstępne	Użytkownik wyszukał interesujący go przejazd.
Warunki końcowe	Dany użytkownik został przypisany do podanego przejazdu jako pasażer.
Scenariusz główny	1. Sprawdzenie czy użytkownik jest zalogowany. 2. Powiązanie użytkownika z przejazdem/przejazdami w bazie danych. 3. Wyświetlenie znalezionych połączeń wraz z przejazdami pośrednimi.
Wyjątki	1.A. Użytkownik nie jest zalogowany. 1.A.1. Wykonanie PU „Zaloguj się”. 1.A.2. Przejście do kroku 2. 1.B. Użytkownik nie posiada konta. 2.A.1. Wykonanie PU „Utwórz konto”. 2.A.2. Przejście do kroku 2. 2.A. Jeden z przejazdów nie ma już wolnych miejsc. 2.A.1. Wyświetlenie informacji o braku miejsc. 2.A.2. Zakończenie PU.
Rozszerzenia	

Przejrzyj przejazdy, do których jesteś zapisany

Aktorzy	Pasażer
Cel	Wyświetlenie przejazdów danego pasażera.
Warunki wstępne	Użytkownik posiada aktywną sesje.
Warunki końcowe	Użytkownik może przejrzeć przejazdy, do których jest przypisany jako pasażer.
Scenariusz główny	1. Pobranie przejazdów, gdzie pasażerem jest dany użytkownik. 2. Wyświetlenie pobranych przejazdów.
Wyjątki	2.A. Użytkownik nie jest przypisany do żadnego przejazdu. 2.A.1. Wyświetlenie informacji o braku przejazdu.
Rozszerzenia	• Oceń kierowce • Wypisz się z przejazdu

Oceń kierowce

Aktorzy	Pasażer
Cel	Ocenienie kierowcy
Warunki wstępne	Użytkownik zakończył przejazd jako pasażer.
Warunki końcowe	Ocena kierowcy jest zapisana w bazie danych.
Scenariusz główny	1. Pobranie oceny oraz komentarza wprowadzonego przez użytkownika. 2. Zapisanie danych.
Wyjątki	
Rozszerzenia	

Wypisz się z przejazdu

Aktorzy	Pasażer
Cel	Wypisanie się z przejazdu.
Warunki wstępne	Użytkownik jest przypisany do nadchodzącego przejazdu.
Warunki końcowe	Użytkownik nie jest powiązany z nadchodzącym przejazdem.
Scenariusz główny	1. Usunięcie powiązania między pasażerem, a podanym przejazdem. 2. Wysłanie powiadomienia do kierowcy.
Wyjątki	
Rozszerzenia	

Uruchom menadżer pasażera

Aktorzy	Pasażer
Cel	Informowanie pasażera o pozycji kierowcy i nawigowanie do przystanku.
Warunki wstępne	Pasażer posiada przejazd w bliskim czasie.
Warunki końcowe	Pasażer rozpoczął przejazd.
Scenariusz główny	1. Zapisanie pozycji pasażera w bazie danych. 2. Pobranie początku przejazdu z bazy danych. 3. Zaznaczenie pozycji pasażera względem trasy przejazdu. 4. Pobranie pozycji kierowcy. 5. Zaznaczenie pozycji kierowcy względem trasy przejazdu. 6. Przejście do kroku pierwszego.
Wyjątki	1.A. Nie można pobrać położenia pasażera. 1.A.1. Wyświetlenie widoku z możliwością uruchomienia modułu gps. 1.A.2. Przejście do kroku pierwszego. 4.A. Pozycja, kierowcy jest nieznana. 4.A.1. Wyświetlenie informacji o nieokreślonej lokalizacji kierowcy. 4.A.2. Przejście do pierwszego kroku. 6.A. Dotarcie do początku przejazdu. 6.A.1. Zakończenie przypadku użycia.
Rozszerzenia	

Dodatek B

Opis zawartości płyty CD

W głównym katalogu załączonej do pracy płyty CD znajdują się:

- jedzmyrazem_android - kod źródłowy projektu aplikacji mobilnej
- jedzmyrazem_rails - kod źródłowy web serwisu i aplikacji internetowej
- dokumentacja.pdf - niniejszy dokument
- streszczenie.pdf - streszczenie niniejszego dokumentu