Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

Piotr Zieliński
Nr albumu: 29979

Kierunek studiów: Informatyka
Forma studiów: studia stacjonarne

System płatności w strefie płatnego parkowania z wykorzystaniem urządzeń mobilnych oraz kodów QR

A payment system in a paid parking zone with mobile devices and QR codes

Praca dyplomowa inżynierska
napisana pod kierunkiem:
dr inż. Edwarda Półrolniczaka
Katedra Systemów Multimedialnych

Data wydania tematu pracy:
Data złożenia pracy:

Szczecin, 2016

Spis treści

Wstęp

W największych polskich metropoliach codziennością są różnego rodzaju systemy informatyczne, wspierające miejską infrastrukturę. Ich zastosowanie obejmuje najczęściej transport publiczny, czy dotyczące tej pracy, strefy płatnego parkowania, gdzie bilet może zostać kupiony przy użyciu aplikacji na urządzenie mobilne. Popularne stało się ostatnio wprowadzanie kart miejskich, takich jak Szczecińska Karta Aglomeracyjna, integrujących wybrane systemy miejskie. Pełnią one rolę wirtualnych portmonetek, przy użyciu których można zapłacić zarówno za bilet komunikacji, jak i wypożyczyć rower miejski.

Wdrażanie podobnych rozwiązań z pewnością nie byłoby możliwe, gdyby nie znaczący w ostatnich latach rozwój płatności elektronicznych. Dotyczą one realizowania transakcji finansowych przy użyciu elektronicznych instrumentów płatniczych i znacząco przyczyniły się do powstania alternatywnych form prowadzenia biznesu. E-commerce (ang. handel elektroniczny) rozumiany jest jako całokształt prowadzenia działalności gospodarczej, głównie za pośrednictwem internetu, i stał się jedną z ważniejszych gałęzi gospodarki [21]. Dynamiczny rozwój e-płatności w ostatnich latach, wzajemnie napędzany był przez postęp technologiczny. Główny udział w tym miały zyskujące coraz większą popularność urządzenia mobilne, których poziom nasycenia w krajach wysoko rozwiniętych często przekracza już 100% [8]. Szeroki zakres oferowanych w nich funkcjonalności dotyczy także płatności (tzw. m-płatności), a same smartfony bywają już nazywane instrumentami płatnicznymi. Najczęściej wykorzystywanych do niewielkich transakcji finansowych, tzw. mikropłatności, jak chociażby zakup biletu postojowego.

Temat pracy został wybrany, ze względu na chęć rozwinięcia już istniejących systemów płatności w strefie płatnego parkowania o kody QR, które mają posłużyć do identyfikacji pojazdów. Realizacja zadań praktycznych wymaga zapoznania się z nowymi narzędziami oraz technologiami, takimi jak: tworzenie aplikacji internetowych oraz mobilnych i ich integracji z systemami płatności. Związana z tym możliwość zyskania nowych umiejętności też miała wpływ na podjęcie wyboru.

Celem tej pracy jest stworzenie systemu dla strefy płatnego parkowania, który umożliwia zarówno kupno, jak i kontrolę biletu postojowego, z wykorzystaniem urządzeń mobilnych oraz kodów QR.

Utworzony w ramach tej pracy system dla strefy płatnego parkowania, nazwany został ParQ. Zadania, które należało wykonać w jego zakresie, obejmowały napisanie aplikacji internetowej oraz osobnych aplikacji mobilnych dla kierowcy oraz kontrolera. Cała interakcja użytkownika z systemem odbywa się za pośrednictwem urządzenia mobilnego. Tam ma możliwość zarejestrowania, dodania pojazdu, czy zakupu biletu. Przed tym musi jednak zostać doładowane konto, z którym powiązana jest wirtualna portmonetka, co także realizowane jest z poziomu aplikacji mobilnej. Z każdym pojazdem powiązany jest numer identyfikacyjny UUID, który przedsta-

wiony w postaci kodu QR zostaje następnie umieszczany za przednią szybą pojazdu. Kontroler skanując taką plakietkę aparatem urządzenia mobilnego z zainstalowaną aplikacją, zyskuje informację o ważności biletu.

W rozdziale 1 został poruszony temat płatności elektronicznych. Celem tej części jest wprowadzenie czytelnika do omawianego zjawiska. Pokrótce przedstawiono historię oraz etapy rozwoju, poddano analizie przyczyny coraz większej popularności e-płatności, a także ich wpływ na gospodarkę, czy modyfikację obecnych modeli biznesowych. Duży nacisk został położony na zaprezentowanie różnych form płatności internetowych, razem z przedstawieniem ich wad oraz zalet. Na końcu opisane są bramki płatności online.

Rozdział 2 zawiera informacje o technologiach i narzędziach, które zostały użyte do realizacji celu pracy. Pierwszy podrozdział poświęcony jest kodom graficznym QR, używanych przy identyfikacji pojazdów w systemie. Tam dokonano ich podziału, ze względu na wersje, typ przechowywanych danych oraz poziom korekcji błędów. Następna część rozdziału opisuje architekturę REST, której zalecenia zostały wykorzystane do komunikacji urządzeń mobilnych z serwerem w systemie. Dalszy fragment poświęcony jest systemowi Android. To właśnie dla niego wykonana została część mobilna pracy. Natomiast do implementacji aplikacji internetowej użyto frameworka Django, który opisany jest na końcu rozdziału.

W rozdziale 3 znajdują się informacje na temat Strefy Płatnego Parkowania w Szczecinie, z którą system ParQ jest zgodny. Tutaj opisano sposób w jaki naliczane są opłaty za postój. Przedstawione zostały także podobne systemy do tworzonego w ramach tej pracy, które pozwalają na kupienie biletu za pośrednictwem urządzeń mobilnych.

W rozdziale 4 zawarta została cała dokumentacja techniczna. Na początku przedstawiony został sposób, w jaki system działa. Następnie wymienione są wymagania funkcjonalne oraz niefunkcjonalne. Kolejny podrozdział to diagramy UML, w tym diagramy przypadków użycia, pakietów, klas, aktywności i sekwencji. Ostatnia część opisuje api, za pomocą którego aplikacja mobilna komunikuje się z serwerem. Przedstawione zostały tam przykładowe zapytania oraz odpowiedzi.

Ostatni rozdział 5 opisuje szczegóły dotyczące implementacji systemu, z podziałem na część mobilną oraz serwerową. Zaprezentowano tam też fragmenty kodu, realizujące wybrane funkcjonalność. Dalsza część zawiera wyniki działania systemu, czyli zrzuty ekranu działających aplikacji mobilnych oraz internetowej. Tutaj opisane są też sposoby, jakimi testowano oprogramowanie oraz środowiska i edytory użyte do jego pisania.

1 Płatności elektroniczne

Poniższy rozdział jest poświęcony płatnościom elektronicznym. W pierwszej części przedstawiona została historia ich rozwoju. Dalej dokonano analizy różnych metod płatności, ze względu na takie kryteria jak wielkość pojedynczej transakcji. Na końcu omówione zostały bramki płatności online.

1.1 Wprowadzenie

E-commerce niepoprawnie utożsamiany jest tylko z dokonywaniem zakupów przez internet, mimo że może odbywać się także z wykorzystaniem m.in. telefonu, faksu, czy telewizji. Odnosi się ogólnie do stosowania urządzeń elektronicznych w zakupie oraz sprzedaży. Jednak to właśnie internet jest dominującą obecnie formą e-handlu, będąc medium informacyjnym łączącym niejako wszystkie poprzednio używane. Tradycyjne formy przeprowadzania transakcji nie pasują do specyfiki biznesu internetowego. Opłata za pobraniem związana jest z wyższą prowizją, a przelewy wiążą się z długim czasem oczekiwania za zrealizowanie operacji finansowej. Nie są to efektywne metody płatności w internecie, gdzie niebagatelne znaczenie ma właśnie szybkość. Alternatywą, a także odpowiedzą na oczekiwania e-biznesu, są należące do bezgotówkowych form transakcji - płatności elektroniczne. Są to wszelkiego rodzaju opłaty, zawierane za pośrednictwem internetu. Nazywane także e-płatnościami [4], przeprowadzane są różnymi kanałami elektronicznymi, do których należą: karty płatnicze, czy przelewy bankowe. Bardzo popularne są ostatnio także usługi oferowane przez dostawców płatności elektronicznych, takich jak PayPal.

Wspólny rozwój e-handlu z internetem umożliwił powstanie nowych metod zawierania transakcji. Dzięki temu są one dobrze przystosowane do wymagań stawianych w rozwiązaniach z dziedziny e-commerce. Oprócz szybkości, oferują one także bezpieczeństwo oraz wygodę w zawieraniu transakcji. Dzięki coraz większej konkurencji pomiędzy dostawcami usług płatniczych - także korzystniejsze prowizje. Ich zastosowanie rośnie, wraz z rosnącą liczbą usług oferowanych w internecie. Szczególnie zaznaczyły swoja obecność w aplikacjach mobilnych.

1.2 Ewolucja systemów płatniczych

Płatności elektroniczne mają swój początek w e-bankowości. Wprowadzanie przez banki, a później instytucje pozabankowe, nowe udogodnienia technologiczne, spowodowały radykalną zmianę w sposobie przeprowadzania operacji finansowych. Przykładem tego mogą być karty płatnicze, zaprezentowane po raz pierwszy w latach pięćdziesiątych. Innym znaczącym osiągnięciem są pieniądze elektroniczne, także będące formą bezgotówkowych transakcji.

Bankowość elektroniczna

Bankowość elektroniczna kryje się pod wieloma nazwami: Internet banking, e-banking, on-line banking. Według J. Masiota jest to "każda usługa bankowa, która umożliwia klientowi wzajemny kontakt z instytucją bankową z oddalonego miejsca poprzez: telefon, terminal, komputer osobisty, odbiornik telewizyjny z dekoderem" [3]. Dodatkowo użytkownikowi oferowany jest podobny zakres usług jak w placówce fizycznej. Ogólnie dotyczy ona zdalnej obsługi konta bankowego. Pierwsze zastosowanie e-bankingu nastąpiło w USA, gdzie Diners Club wprowadził kartę płatniczą [3]. Nikt wtedy nie mógł zdawać sobie sprawy, jaką wielką popularność zyska ten instrument płatniczy. Następnie w 1970 r. powstał system kart debetowych, a w latach osiemdziesiątych pojawiły się karty zawierające mikrochip. W Polsce pierwsze bankomaty powstały w 1990 r. za sprawą banku Pekao S.A.

Home banking był jedną z form bankowości elektronicznej, który powstał z myślą o klientach indywidualnych i małych przedsiębiorcach. Po zainstalowaniu specjalnego oprogramowania, bądź kupienia odpowiedniej przystawki, klient mógł wykonywać operacje na swoim koncie. Ta i podobne odmiany e-bankingu nie zdążyły na dobre zaznaczyć swojej obecności. Wprowadzenie internetu do powszechnego użytku, przemodelowało dotychczas stosowane rozwiązania.

Bankowość internetowa

Początki sieci globalnej sięgają lat sześćdziesiątych XX stulecia, kiedy to na zlecenie Departamentu Obrony USA opracowany został ARPA-Net [3]. Od tego momentu Internet ewoluował, modyfikując stopniowo naszą rzeczywistość. Duże ułatwienia w komunikowaniu się wpłynęły na przemiany społeczne, oprócz tego powstanie oraz rozwój sieci odbił się szczególnie mocno na dziedzinach związanych z przetwarzaniem informacji [3], czyli m.in. na sektor bankowy. Jego podatność na innowacje technologiczne pozwoliła na zupełnie nowy sposób dostępu do usług bankowych.

W bankowości internetowej oraz wirtualnej komunikacja odbywa się za pośrednictwem przeglądarki internetowej. Klient ma dostęp do większości usług oferowanych przez bank w placówce. Użytkownik może kontrolować stan konta, zaciągać kredyty lub wykonywać przelewy. Dostępność do takiej usługi jest niezależna od miejsca, 24 godziny na dobę i posiada wszystkie zalety, jakie niesie ze sobą korzystanie z internetu. La Jolla Bank FSB w 1994 r. był pierwszym bankiem, który umożliwił wykonywanie podstawowych operacji za pośrednictwem sieci. Ciekawą i dość popularną także w Polsce odmianą bankowości, jest bankowość wirtualna. Polega ona na obsłudze klienta tylko internetowo, a banki takie często nie posiadają nawet swoich placówek. Przykładem takich banków jest chociażby mBank.

Sieć, by móc się rozprzestrzeniać, musiała przez lata wykształcić takie właściwości, jak: bezpieczeństwo, uniwersalność, interaktywność. Chcąc dokonać płatności, czy sprawdzić konto w

banku chcemy mieć pewność, że nasze dane są bezpieczne. Istotny jest także sposób dostępu, coraz mniej zależny od używanego systemu operacyjnego. Na przestrzeni lat najważniejszy okazał się jednak stały rozwój.

Pierwsza generacja płatności internetowych

Rozpoczęta w latach dziewięćdziesiątych pierwsza generacja płatności, próbowała wprowadzić alternatywną gotówkę, np.: e-monety, czy tokeny [4]. E-gotówka miała zachować wszystkie cechy tradycyjnego pieniądza, oferując m.in. brak opłat transakcyjnych, czy anonimowość. Pierwszym systemem był wydany w 1994 r. e-cash, założony przez amerykańską firmę DigiCash. Podobnie jak w większości wprowadzanych w tamtym czasie rozwiązań, tak samo e-cash oznaczał elektroniczną walutę indywidualnym numerem seryjnym. Takie podejście miało chronić pieniądze przed fałszerstwem, a dostarczało tylko dodatkowych trudności. Cechą wspólną pierwszej generacji jest dość problematyczna obsługa oraz wymaganie dodatkowego oprogramowania, bądź nawet czytników kart. Sama firma DigiCash zakończyła swoją działalność w 1998 r.

Druga generacja płatności internetowych

Trwająca do dziś i charakteryzująca się znacznie większą prostotą druga generacja, została zapoczątkowana na przełomie XX i XXI wieku. Jej powstanie i odmienność od wcześniej stosowanych rozwiązań, wynika z możliwości i ułatwień jakie niesie ze sobą internet. Po kilkunastu latach dynamicznego rozwoju, zdążył się lepiej dostosować do stawianych mu wymagań. Szczególnie postęp do obszarze zabezpieczeń, wiążący się z powstaniem szyfrowanych protokołów przesyłania danych (np.: HTTPS), jest znaczący w systemach płatności. Nie są już potrzebne specjalne czytniki, wszystko może odbywać się przez przeglądarkę internetową. Sprawia to, że korzystanie z takiej formy płatności jest znaczenie wygodniejsze i szybsze, a także prostsze, gdyż zmniejsza się ilość kroków, jakie trzeba wykonać, aby dokonać zakupu.

1.3 Analiza metod płatności

Różnorodność dostępnych metod płatności internetowych sprawia, że mogą być one bardzo dobrze wpasowane w każdy model biznesowy. Ważnym kryterium przy wyborze płatności jest wielkość pojedynczej transakcji w systemie, co wiąże się bezpośrednio z poziomem zabezpieczeń jaki należy zapewnić. Na decyzję powinny także wpływać indywidualne preferencje użytkowników. Szczególnie istotny jest poziom zaufania, z jakim spotyka się dane rozwiązanie. Duża część użytkowników internetu przyzwyczajona jest do tradycyjnych płatności, szczególnie do gotówki i takiej formy zapłaty będą oczekiwać. Jest to ważny wybór, wpływający na

odczucia płynące z korzystania z serwisu.

Wysokość transakcji

Przedstawiony poniżej podział dokonany został ze względu na wielkość pojedynczej transakcji. Obok każdej z kategorii zostały podane wartości, z którymi można się w ich przypadku najczęściej spotkać. Niema w biznesie jednej, ścisłej definicji wymienionych tutaj typów płatności. Wartości mogą się też różnić, w zależności od dostawcy usług płatniczych. To rozróżnienie sugeruje przede wszystkim poziom zabezpieczeń, jaki należy zapewnić podczas przeprowadzania transakcji.

- Milipłatności - płatność do kilkudziesięciu groszy,
- Mikropłatności - 1 zł do 80 zł,
- Minipłatności - 80 zł do 800 zł,
- Makropłatności - wszystko powyżej 800 zł.

Milipłatności z mikropłatnościami dotyczą niewielkich, bardzo często kilkugroszowych operacji finansowych. Ich definicja różni się co do górnej granicy transakcji - najczęściej wynosi ok. 80 zł [2][4]. Oprócz kwoty, dodatkowym wyróżnikiem jest krótki czas oraz łatwość w przeprowadzaniu płatności. Użytkownik spodziewa się, że taki proces nie będzie wymagał podjęcia przez niego wielu kroków - bardzo często opłaty w sklepach internetowych mogą być zrealizowane bez konieczności opuszczania strony sprzedającego. Ze względu na małe kwoty zabezpieczenia nie muszą być bardzo restrykcyjne, co pozwala na wprowadzenie wymienionych udogodnień. Większość transakcji zawieranych w internecie zamyka się w granicach mikropłatności. To właśnie ich dalszy rozwój, poprzez powstawanie nowych kanałów realizacji opłat, jest najistotniejszy zarówno dla sprzedawców jak i kupujących.

Podejście do zabezpieczeń w przypadku minipłatności musi być zdecydowanie bardziej rygorystyczne, a dla makropłatności stanowi to już priorytet. Tego typu transakcje związane są z większą liczbą kroków, jaką konsument musi wykonać, aby mogła być zrealizowana. Najczęściej będzie się odbywała za pośrednictwem strony banku lub dostawcy usług płatności. Powoduje to, że zakupy są znacznie wolniejsze, jednak w tym przypadku nie jest to wadą. Dzięki temu ryzyko niechcianych zakupów lub dokonania transakcji przez osobę trzecią jest mniejsze. Płatności mogą dotyczyć np.: zakupu sprzętu RTV lub AGD.

Moment pobrania

W przeciwieństwie do tradycyjnych metod płatności, te elektroniczne mogą zostać w znaczenie większym stopniu dopasowane do potrzeb przedsiębiorstwa. To właśnie elastyczność przyczy-

nia się w znacznym stopniu do ich rozpowszechniania. Jednym z takich czynników jest właśnie moment pobrania. Dokonanie zakupu towaru bądź usługi nie zawsze musi się wiązać z natychmiastowym pobraniem pieniędzy z konta. W zależności od rodzaju płatności elektronicznej, przelanie pieniędzy będzie odbywać na różnych etapach. Niektóre serwisy mogą na przykład wymagać zakupu wirtualnej waluty, obowiązującej tylko w jego ramach. Taka sytuacja często spotykana jest w grach sieciowych.

Poniżej przedstawione zostały różne momenty pobrania, z jakimi można się spotkać w płatnościach elektronicznych. Wybór konkretnej kategorii będzie wiązał się z szybkością transakcji oraz poziomem zabezpieczeń, niezbędnym podczas ich realizacji.

- System przedpłat (ang. pay before) - użytkownik najpierw musi zasilić swoje wirtualne konto, aby później mieć możliwość dokonywania zakupów.
- System natychmiastowych płatności (ang. pay now) - występuje w przypadku kart debetowych. Obciążenie rachunku następuje w momencie dokonania płatności. Nie ma możliwości uzyskania kredytu.
- System z odroczoną płatnością (ang. pay later) - do tego typu płatności należą karty kredytowe i obciążeniowe. Rachunek posiadacza takich instrumentów płatniczych zostanie obciążony dopiero w momencie spłaty zaciągniętego kredytu.

Dużą popularność zyskują rozwiązania typu pay-before. Funkcjonują one pod postacią wirtualnych portmonetek, gdzie użytkownicy trzymają elektroniczne pieniądze. Najpierw muszą zostać zasilone przez właściciela odpowiednią kwotą, aby później mogły być wykorzystywane do najróżniejszych transakcji finansowych. Są opcją szczególnie korzystną w przypadku częstych opłat, ponieważ nie są z nimi związane prowizje od każdej operacji, jak w przypadku kart. Także są względnie bezpieczniejsze - użytkownik nie łączy się za każdym razem z bankiem, a w przypadku oszustwa - straci jedynie środki wpłacone na konto.

Metody płatności w internecie

Zakupy w internecie nie muszą dotyczyć tylko usług cyfrowych, niedostępnych w sklepach stacjonarnych. Często stanowią konkurencję dla tradycyjnych form sprzedaży, posiadając nad nimi wiele zalet (chociażby możliwość zwrotu towaru do 14 dni od zakupu [1]). Zapłatę w sklepach internetowych bardzo często można dokonać na wiele różnych sposobów, dostosowanych do prywatnych preferencji kupującego. Znajdują się wśród nich także metody nie należące do płatności elektronicznych, jak przelewy tradycyjne.

Wybór odpowiednich metod realizacji transakcji udostępnianych dla klientów w serwisie, powinien być poprzedzony dokładną analizą grupy docelowych użytkowników. Niektórzy mogą nie mieć dużego zaufania do tego typu rozwiązań i będą woleli zapłacić gotówką. Próba stwo-

rzenia serwisu przyjmującego większość dostępnych metod płatności, może okazać się z kolei zbyt kosztowym i czasochłonnym zadaniem. Do najpopularniejszych metod płatności zaliczyć można: przelewy tradycyjne i internetowe, płatności komórką, portfelem internetowym oraz płatności kartami płatniczymi.

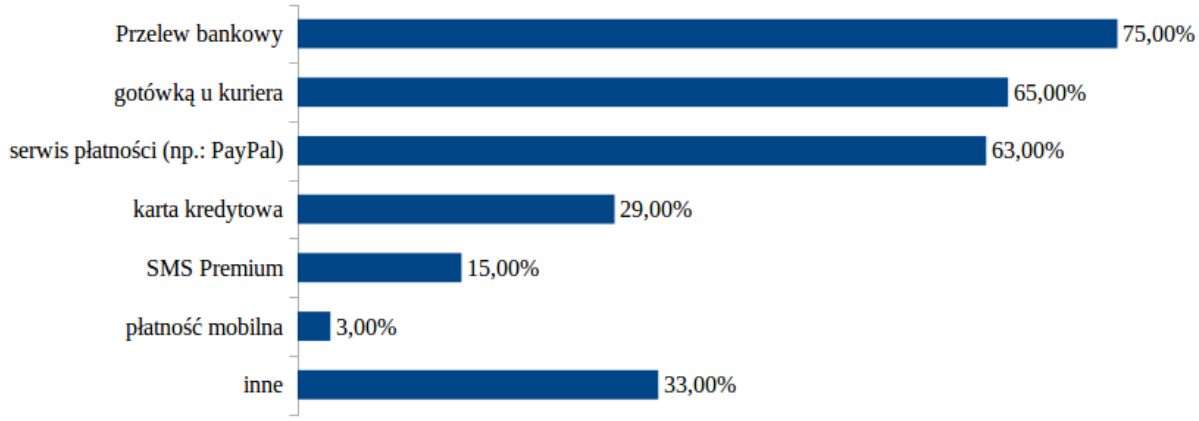
Przelewy internetowe są wciąż najchętniej wybieraną metodą płatności w Polsce. Realizacja odbywa się za pośrednictwem strony internetowej wybranego przez klienta banku, w którym trzyma pieniądze. Po zalogowaniu, należy wpisać dane odbiorcy oraz przelewaną kwotę. W przeciwieństwie do tradycyjnych przelewów, sprzedawca otrzymuje pieniądze szybciej, najczęściej następnego dnia. Dodatkowo klient nie musi odwiedzać placówki banku lub poczty. Taka forma zapłaty jest bardzo bezpieczna i wygodna dla klienta. Wadą jest na pewno ilość kroków, jakie należy wykonać - zawsze trzeba obowiązkowo odwiedzić stronę banku, co przy częstych i niskich transakcjach jest dość niewygodne. Tego typu operacje obciążone są pewną prowizją, a dzień oczekiwania na realizację to wciąż długo, szczególnie w porównaniu z możliwościami innych metod płatności.

Częściowym rozwiązaniem na problemy związane z przelewami, jest stosowanie kart płatniczych w transakcjach internetowych. Proces płacenia jest dużo łatwiejszy i szybszy. Użytkownik zobowiązany jest do podania danych karty, takich jak: numer, data ważności, kod zabezpieczający oraz imię i nazwisko posiadacza. Cała operacja płacenia przeprowadzana jest bez potrzeby opuszczania sklepu. Ponadto, podanie danych może być wymagane tylko raz. To niestety może prowadzić też do nadużyć, czy niechcianych subskrypcji. Kolejną wadą są stosunkowo wysokie prowizje, wynoszące ok. 2 - 3%. Mimo, że pozwalają na dokonywanie płatności niezależnie od waluty (w przypadku przelewów nie jest to możliwe), to przewalutowanie jest niekorzystne przy większych zakupach. Wbrew tym niedogodnością, karty zyskują coraz większą popularność w Polsce, będąc już jedną z najczęściej wybieranych metod.

SMS Premium to specjalna usługa telefoniczna, pozwalająca na przeprowadzanie opłat używając do tego telefonu komórkowego. Wiadomość tekstowa wysłana na specjalny numer, obciążona jest dodatkową opłatą. W ten sposób sprzedawca mający podpisaną umowę z operatorem telefonicznym, może otrzymywać od niego pieniądze, za wysłany SMS. Ogromnymi zaletami tej metody jest duża szybkość realizacji, a także prostota obsługi. Niestety, wysokość prowizji przekraczająca 50%, sprawia że jest często nieopłacalna. Także jej ograniczenie do niewielkich kwot, od 1 zł do 19 zł, decyduje o jej coraz mniejszym wykorzystaniu.

Coraz częściej spotykane są, opisywane wcześniej, płatności portfelami elektronicznymi. Po utworzeniu oraz zasileniu konta, użytkownik może dysponować swoimi środkami w postaci pieniędzy elektronicznych. Ta forma cechuje się bardzo dużą szybkością realizacji transakcji, zwłaszcza jeśli odbywa się w ramach tego samego systemu. Dodatkowo w takiej sytuacji, przekazywanie pieniędzy najczęściej pozbawione jest jakiejkolwiek prowizji. Niestety płatności ograniczona jest tylko do jednego systemu elektronicznych portmonetek. Klient nie będzie

miał możliwości jej wykorzystania, jeśli taka możliwość nie jest udostępniana przez sklep internetowy.



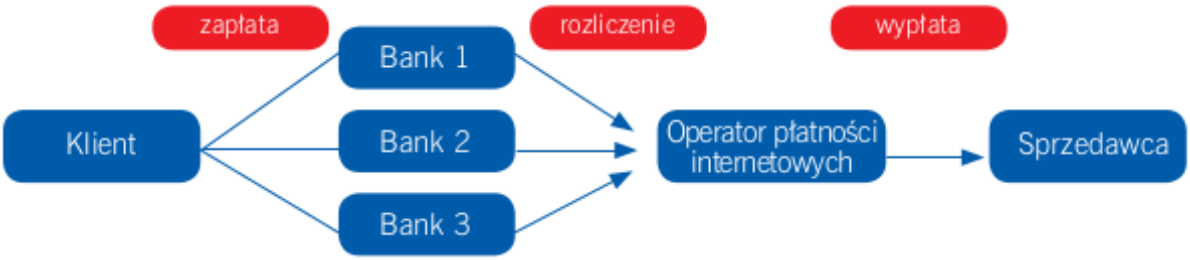
Rys. 1.1: Płatności z których skorzystali internauci
Źródło: Opracowanie własne. Dane z raportu Gemius. [22]

1.4 Bramki płatności

Wdrożenie jednej lub kilku z wymienionych metod we własnym zakresie nie jest prostym rozwiązaniem. Implementacja z pominięciem wszelkich pośredników może okazać się zajęciem zbyt kosztownym i czasochłonnym, szczególnie dla firm rozpoczynających swoją działalność. Im więcej metod sprzedawca chce udostępnić, tym proces ten jest dłuższy i bardziej skomplikowany, angażując do tego kolejne podmioty. Wymaga to podpisania wielu dodatkowych umów z bankiem, czy posiadania specjalnego w nim konta.

Bardzo dobrą alternatywą jest skorzystanie z usług operatorów płatności elektronicznych. Są oni pośrednikami transakcji przeprowadzanych w internecie, przyjmując opłaty od klientów i przekazując następnie na rachunek sprzedawców. Ich usługi polegają na integrowaniu wielu metod płatności, dzięki czemu przedsiębiorca nie musi ręcznie wdrażać każdej z nich. Wystarczy, że podpisze umowę z jednym z dostawców systemów płatniczych. Jest to rozwiązanie korzystne także ze strony osoby dokonującej zakupów. Oferowany jest jej szeroki zakres metod, w jakich może dokonać płatności, np.: karty płatnicze, e-przelewy, portmonetki elektroniczne. Korzystanie z usług operatorów płatności jest bardzo częste. W popularnym polskim serwisie aukcyjnym Allegro, wszystkie opłaty realizowane są z pośrednictwem systemu PayU.

Przechodząc do finalizacji transakcji, klient proszony jest na stronie sprzedawcy o wybranie metody płatności. W przypadku przelewu internetowego, kierowany jest jeszcze na stronę swojego banku, skąd po podaniu danych, wraca na witrynę sprzedawcy. Po zaakceptowaniu zapłaty, operator usług dostaje przelew z banku klienta, skąd dalej w różny sposób pieniądze zostają przekazywane na rachunek sprzedawcy.



Rys. 1.2: Schemat działania bramek płatności
Źródło: Elektroniczne metody płatności. [4]

Na rynku istnieje wielu dostawców takich usług, np.: PayU, Dotpay, Przelewy24, czy PayPal. Różnią się oni między sobą przede wszystkim prowizjami za wypłatę pieniędzy, czy za skorzystanie z jakiejś metody płatności. Większość z nich pełni głównie rolę pośrednią, znajdując się między bankiem kupującego, a sprzedającego.

1.5 PayPal

Użytkownik kupując bilet postojowy w pewnym momencie będzie musiał za niego zapłacić, dlatego tworzony system musi akceptować płatności. Do tego zadania wykorzystany został PayPal. Jego sposób działania jest nico odmienny, od pozostałych bramek płatności jakie zostały wymienione, gdyż pełni on także funkcję wirtualnej portmonetki. Oznacza to, że pieniądze nie trafiają od razu na konto sprzedającego, jak to się dzieje w przypadku PayU, czy Dotpay. Zapisywane są najpierw w postaci elektronicznej i dopiero na żądanie mogą zostać wypłacone. Wiąże się to niestety z dłuższym czasem oczekiwania i pewną prowizją.

Integracja

Po założeniu konta w PayPalu integracja z tworzonym systemem może odbywać się na kilka sposobów. W przypadku sklepów, gdzie płatności realizowane są na stronach internetowych, popularnym i najszybszym rozwiązaniem jest PayPal Standard Payments. Jest to usługa, która pozwala na wygenerowanie gotowych przycisków, w postaci fragmentów kodu HTML. Jego kliknięcie przenosi na stronę z wyborem metody płatności, a pieniądze przelewane są na powiązane konto sprzedającego. Mogą one dotyczyć jednej transakcji, ale możliwa jest także obsługa całego asortymentu sklepu. O dokonaniu transakcji i produkcie lub usłudze jaka została kupiona, sprzedawca informowany jest za pośrednictwem swojego konta na PayPalu.

Powyższy sposób sprawdza się jedynie w przypadku sklepów internetowych. Do tworzonego systemu płatnego parkowania, gdzie płatność jest realizowana poprzez aplikację na urządzeniu mobilnym, niezbędne jest inne podejście. W tym przypadku komunikacja z PayPalem i realizacja opłat odbywa się za pośrednictwem udostępnianego API. Polega ona na wymianie żądań

i odpowiedzi z serwerem, poprzez protokół HTTP. Może to się odbywać zarówno w architekturze REST, jaki i SOAP. Zapytania wysyłane są na domenę api.paypal.com, a komunikacja zabezpieczona jest standardem OAuth2.

Pierwszą rzeczą jaką należy wykonać to zarejestrowanie klienta, który będzie miał prawo obsługiwać płatności w imieniu posiadacza konta PayPal. Dzieje się to poprzez utworzenie tzw. aplikacji, razem z którą zostanie wygenerowany jego identyfikator oraz sekret, w postaci ciągu znaków. Obie te wartości wysłane w odpowiednim żądaniu HTTP, pozwolą na uzyskanie tokenu autoryzacyjnego. Na listingu 1.1 pokazany został format odpowiedzi z takim tokenem. Obowiązuje on przez określony czas, a po jego wygaśnięciu należy ponowić żądanie.

Listing 1.1: JSON z tokenem autoryzacyjnym
Źródło: dokumentacja PayPala [4]

```
{
    "scope":"https://api.paypal.com/v1/payments/.*
        https://api.paypal.com/v1/vault/credit-card
        https://api.paypal.com/v1/vault/credit-card/.*",
    "access_token":"Access-Token",
    "token_type":"Bearer",
    "app_id":"APP-6XR95014SS315863X",
    "expires_in":28800
}
```

Taki token będzie następnie umieszczany w nagłówku każdego zapytania, które wymaga autoryzacji. Dzięki temu będzie możliwe realizowanie pozostałych usług oferowanych przez API PayPala. Może to być chociażby pozwolenie na realizowanie tzw. Future payments, czyli cykliczne pobieranie opłat od klienta. Dzięki temu nie będzie on musiał podawać danych uwierzytelniających przy każdej płatności. W realizowanym systemie wykorzystywane będą jednorazowe opłaty natychmiastowe, gdzie każda transakcja wymaga uwierzytelnienia przez użytkownika. Przykład takiego żądania zaprezentowany został na listingu 1.2. W nim podawana jest metoda płatności, oraz kwota jaka ma zostać przelana na konto sprzedającego. Dodatkowo, w bardziej rozbudowanych systemach, możliwe jest także wysłanie informacji o produkcie jakie jest kupowany.

Listing 1.2: Dane żądania utworzenia płatności
Źródło: dokumentacja PayPala [4]

```
{
  "intent":"sale",
  "redirect_urls":{
    "return_url":"http://example.com/your_redirect_url.html",
    "cancel_url":"http://example.com/your_cancel_url.html"
  },
  "payer":{
    "payment_method":"paypal"
  },
  "transactions":[
    {
      "amount":{
```

```
        "total":"7.47",
        "currency":"USD"
      }
    }
  ]
}
```

W stworzonym systemie komunikacja z PayPalem odbywa się w podobny sposób. Wykorzystana została jednak do tego specjalna biblioteka, dzięki czemu nie ma potrzeby ręcznego budowania takich żądań.

2 Opis wykorzystanych technologii

W tym rozdziale znajdują się informacje o najważniejszych technologiach, wykorzystanych podczas realizacji zadań pracy. W kolejnych podrozdziałach opisane zostały kody graficzne QR, system mobilny Android oraz framework aplikacji serwerowych - Django.

2.1 Kody graficzne QR

Przedstawianie danych w postaci kodów graficznych nie jest niczym innowacyjnym - w sklepach towary oznaczane są za pomocą jednowymiarowego kodu kreskowego. Kombinacja jasnych oraz ciemnych linii umożliwia przechowywanie danych, które odczytywane są za pomocą skanera z laserem. Tego typu metody stosuje się głównie w celach identyfikacji. Do przechowywania większej ilości danych wykorzystuje się częściej tzw. kody 2D.

Kody QR (ang. Quick Respone - szybka odpowiedź) to dwuwymiarowe, kwadratowe kody graficzne. Składają się z modułów, czyli kombinacji ciemnych oraz jasnych kwadratów, które są nośnikami danych. Zostały stworzone przez japońską firmę Denso-Wave w 1994 r [24]. Według postanowień licencyjnych mogą być wykorzystywane bez żadnych opłat, a sam standard jest opisany w normie ISO/IEC 18004:2015 [15]. Dzięki dodatkowemu wymiarowi, pozwalają na przechowywanie większej ilości informacji (do ok. 7000 liczb lub 4000 znaków alfanumerycznych) niż kody kreskowe, posiadające tylko jeden wymiar. Ponadto, zapewniają zdecydowanie lepszą korekcję błędów. Nawet częściowo uszkodzony kod może zostać poprawnie odczytany. Posiadają kilka miejsc szczególnych do ułatwienia orientacji podczas odkodowywania. Ich liczba zależy od rozmiaru kodu.

Rys. 2.1: Tytuł pracy przedstawiony w postaci kodu QR

Pierwotnie bardzo duże zastosowanie kody QR znalazły w logistyce, gdzie zawierały informacje o przesyłanych paczkach. Współcześnie kojarzone są przeważnie z urządzeniami mobilnymi. Spotykane na przystankach, w sklepach lub magazynach służą do komunikacji z użytkownikami smartfonów, przełamując niejako barierę między światem wirtualnym, a rzeczywistym. Kod zawiera informacje jedynie w postaci liczb, liter i symboli. Jednak odpowiednie formatowanie informacji, pozwala na dodatkowe interpretowanie ich przez urządzenie przenośne. I tak po zeskanowaniu może zostać wysłana wiadomość e-mail, albo dodany numer do kontaktów.

Najczęściej jednak zawierają adresy URL, które powodują wyświetlenie odpowiedniej strony w przeglądarce internetowej telefonu.

Sposoby kodowania

Przed zakodowaniem informacji do postaci kodu QR, należy określić trzy główne parametry. Są to: wersja, typ danych oraz poziom korekcji błędów.

Najważniejszym parametrem, wpływającym bezpośrednio na ilość danych jakie kod będzie w stanie przechować, to jego wersja. Numerowana jest od 1 do 40 i każda z nich ma przypisany do siebie rozmiar. Wersja pierwsza posiada 21 na 21 modułów, druga 24 na 24, a ostatnia, czyli czterdziesta - 177 na 177. Każda kolejna jest większa od poprzedniej o trzy moduły na bok. Oczywiście im numer wersji, czyli też rozmiar, jest większy, tym więcej danych będzie można zakodować.

Równie ważne jak wybór wersji jest określenie typu danych, jaki ma być zakodowany. Informacja o typie zapisywana jest w kodzie QR, dzięki czemu podczas odczytywania czytnik wie jak ma interpretować dane. Dodatkowo, ta informacja decyduje też o maksymalnej pojemności. Typ numeryczny przechowuje informacje jedynie o liczbach, dlatego będzie potrzebował mniej bitów na znak, niż w przypadku danych alfanumerycznych. Dostępne są cztery typy:

- Numeryczny – ten tryb pozwala na zakodowanie tylko cyfr od 0 do 9, co umożliwia maksymalnie na przechowywanie 7089 znaków.
- Alfanumeryczny – oprócz cyfr, także wielkie litery oraz znaki ’$’, ’%’, ’*’, ’+’, ’-’, ’.’, ’/’, ’:’ i spacja. Można zakodować do 4296 znaków.
- Binarny – domyślnie dla zestawu znaków z ISO-8859-1, ale także UTF-8. Maksymalnie 2953 znaków.
- Kanji – znaki z systemu kodowania Shift JIS. Pomieści nie więcej niż 1817 znaków.

Poziom korekcji błędów służy do określenia, czy dane zostały odczytane poprawnie. Pozwala także na odzyskanie części z nich, nawet jeśli kod został uszkodzony (dzięki algorytmowi Reeda-Solomona). Specyfikacja wyróżnia cztery poziomy korekcji. Obok każdego z nich podany został procent danych, jakie można odzyskać:

- L (Low) - 7% danych,
- M (Medium) - 15% danych,
- Q (Quartile) - 25% danych,
- H (High) - 30% danych.

Tab. 2.1: Pojemność kodów QR dla różnych ustawień

Wersja	Moduły	Korekcja	Numeryczny	Alfanumeryczny	Binarny	Kanji
1	21x21	L	41	25	17	10
		M	34	20	14	8
		Q	27	16	11	7
		H	17	10	7	4
40	177x177	L	7089	4296	2953	1817
		M	5596	3391	2331	1435
		Q	3993	2420	1663	1024
		H	3057	1852	1273	784

Tworzenie kodu graficznego QR jest procesem dość złożonym. Po analizie danych określającej ich typ, konieczne jest ich odpowiednie zakodowanie. Informacje dzielone są na bloki, do których dodawane są kolejne bity związane z korekcją błędów. Przed przedstawieniem danych w postaci modułów QR, ważne jest również odpowiednie ich maskowanie. Zbyt duża ilość kwadratów o tym samym kolorze w jednym miejscu, może spowodować błędne odczytanie. Dopiero po tych kilku etapach, może zostać wygenerowany kod. Praktycznie dla każdego języka istnieją biblioteki, które wykonują te wszystkie procesy. Wystarczy podać jedynie parametry kodu z danymi. Na przykład w Pythonie takim modułem jest qrcode.

Odczytanie, czyli odkodowanie informacji możliwe jest na wiele sposobów. Razem z systemami mobilnymi często dostarczane są specjalne aplikacje, które wykorzystując wbudowaną kamerę, pozwalają na odczytanie zakodowanych danych. Podobnie, takie rozwiązanie możliwe jest dzięki odpowiednim biblioteką. W Androidzie jest to dostępna za darmo Zebra Crossing.

2.2 Architektura REST

Porozumiewanie się serwera z urządzeniami mobilnymi w opracowywanym systemie zostało wykonane w oparciu o Representational State Transfer, czyli REST. Jest to wzorzec oprogramowania, opisujący oraz zawierający zalecenia co do tworzenia API w protokole HTTP. Ma w założeniu ułatwić obsługę żądań i odpowiedzi, dzięki czemu nie trzeba zawsze odwoływać do dokumentacji. W jego ramach wykorzystuje się bezstanową komunikację, zasoby, czy hipermedia. Przesyłane dane mogą być w dowolnym formacie, jednak w ostatnim czasie najpopularniejszy jest JSON. Sam REST często jest określany jako następca innego standardu komunikacji – SOAP.

Adresy URL w przypadku REST pełnią rolę pewnego rodzaju identyfikatora, pod którym kryje się konkretny zasób. Wysyłanie określonych żądań na ten adres, będzie wiązało się z przeprowadzeniem związanych z nim operacji. To jaka operacja będzie wykonywana, zależne jest

od metody HTTP określonej w żądaniu, np.: GET – pobranie danych, PUT – edycja, POST – przesłanie nowych danych, DELETE - usunięcie. W REST panuje pewna konwencja co do nazywania adresów URL. Wyróżnia się ich dwa rodzaje, np.: adres /tickets/ będzie się wiązał z dostępem do kolekcji danych, a /tickets/1/ to konkretny element, w tym przypadku z identyfikatorem równym jeden.

2.3 System Android

Systemy na urządzenia mobilne z czasem stawały się coraz bardziej zaawansowane, przypominając swoją funkcjonalnością te przeznaczone na komputery. Dzisiaj oprócz obsługi podstawowych zadań telefonu jak dzwonienie, pozwalają na przeglądanie internetu, czy instalowanie dodatkowych aplikacji. Do najpopularniejszych systemów w Polsce należą: Android z 65% udziałem w rynku, Windows Phone - 16% i iOS - 4% [18].

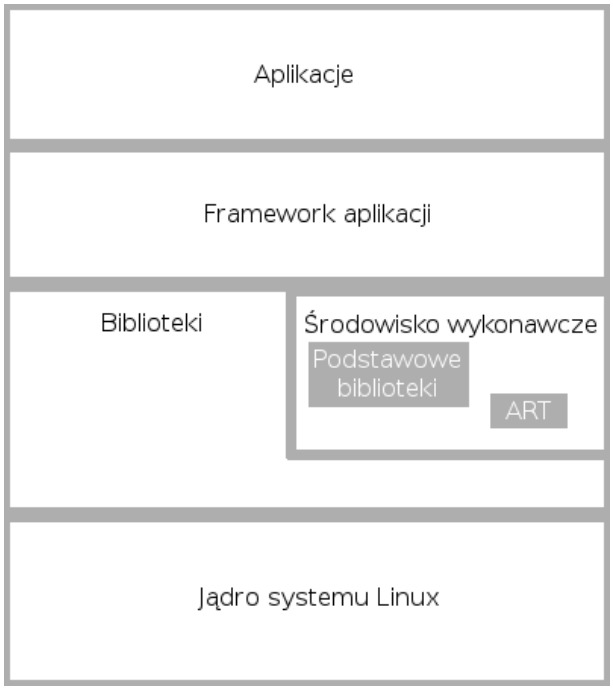
O Androidzie

Android to mobilna platforma systemowa, stworzona w 2003 r, a następnie wykupiona przez Google'a w 2005 r. z rąk Android Inc. Od 2007 r. rozwijany jest w ramach sojuszu kilkudziesięciu firm - Open Handset Alliance. Android został zbudowany na bazie jądra Linuksa i podobnie jak on rozpowszechniany jest za darmo w ramach Open Source. To właśnie dostępność oraz możliwość dowolnego modyfikowania spowodowała, że zdołał w tak niedługim czasie zawładnąć rynkiem, stając się najpopularniejszym systemem mobilnym. Można go spotkać na większości popularnych urządzeń przenośnych, jak: telefony komórkowe, smartfony, tablety, netbooki. Jest stosowany także w e-bookach niektórych firm, czy innych sprzętach domowego użytku.

Architektura systemu

Ze względu na architekturę systemu, można wyróżnić w Androidzie kilka abstrakcyjnych warstw: aplikacji, frameworku aplikacji, bibliotek, środowiska wykonawczego i jądra Linux, na którym bazuje cały system. Cała funkcjonalność systemu, niezbędna podczas działania aplikacji, dostępna jest poprzez framework aplikacji, czyli systemowe API napisane w Javie. Używając go, programista może kontrolować sposób działania oraz wygląd programu. Tutaj znajduje się menedżer aktywności (ang. Activity Manager), odpowiedzialny za cykl życia aplikacji, czy menedżer powiadomień (ang. Notification Manager) obsługujący wyświetlanie wszelkich notyfikacji. Także udostępnianie przez aplikacje swoich funkcjonalności innym programom (z wykorzystaniem intencji) możliwe jest dzięki tej warstwie. Poniżej niej znajdują się natywne biblioteki napisane w C i C++. Dzięki systemowemu API najczęściej nie ma konieczności z

nich korzystać, a do tworzenia aplikacji można używać Javy. Najgłębiej w systemie znajduje się jego jądro, czyli Linux. Wykonuje ono najbardziej podstawowe funkcje, będąc odpowiedzialnym m.in. za zarządzanie baterią. Posiada też sterowniki systemowe: ekranu, aparatu, czy audio.



Rys. 2.2: Schemat architektury systemu Android

Środowisko wykonawcze

Uruchamianiem programów napisanych w Javie zajmuje się wirtualna maszyna Javy (ang. Java Virtual Machine). Po skompilowaniu tworzony jest kod bajtowy, plik class, który następnie po załadowaniu interpretowany jest przez JVM (możliwa też kompilacja JIT). Takie podejście pozwala na przenośność programów, czyli niezależność od platformy, kosztem pewnego spadku wydajności.

Mimo, że programy na Androida pisane są w Javie, to nie JVM używany jest do ich późniejszego wykonywana. Głównie jest to spowodowane chęcią stworzenia środowiska uruchomieniowego, które będzie lepiej przystosowane do słabszych wydajnościowo od komputerów maszyn, jakimi są urządzenia mobilne. Proces budowania aplikacji rozpoczyna się tak samo. Kompilator przekształca pliki zawierające kod Javy, do kodu bajtowego. W takiej postaci program nie mógłby zostać jeszcze uruchomiony. Najpierw specjalne narzędzie, pod nazwą dx, modyfikuje wynik działania kompilatora. Jego zadaniem jest połączenie wszystkich tych kodów bajtowych w jeden plik dex, z usunięciem powtarzających się symboli oraz zmianą znajdujących się tam instrukcji, na odpowiednie dla Androida. Dzięki temu skompilowany program będzie mniejszy oraz powinien działać szybciej. Ostatnim etapem budowania aplikacji jest stworzenie pliku

apk, odpowiednika jar, w którym oprócz kodu bajtowego, znajdą się takie zasoby jak zdjęcia wykorzystywane w aplikacji.

Rys. 2.3: Proces budowy aplikacji

Do wersji 4.4 Androida (KitKat) aplikacje uruchamiane były w wirtualnej maszynie Dalvik. Sposób działania jest dość zbliżony do JVM. Kod bajtowy w postaci plików dex jest interpretowany, z możliwością kompilacji do kodu natywnego (JIT). Jedną z różnic jest sposób działania wirtualnego procesora, który w Dalviku opartu został na rejestrach, a nie na stosie. Takie podejście wpływa na mniejsze zużycie pamięci, jednak programy są większe, gdyż instrukcje muszą zawierać dodatkowe informacje co do rejestrów, z których korzystają.

W wersji 5.0 Androida (Lolipop) zastąpiono dotychczasową maszynę wirtualną Dalvik, środowiskiem uruchomieniowym Android runtime (ART). Również przyjmuje pliki dex, jednak nie interpretuje ich, a w momencie instalacji aplikacji - kompiluje. Taka kompilacja kodu pośredniego, języka wysokiego poziomu, do kodu natywnego, nosi nazwę Ahead-of-time (AOT). Przy każdym uruchomieniu aplikacji, dzięki ART, wykorzystywany jest jej natywny kod. Ta zmiana powoduje szybsze działanie i uruchamianie się aplikacji. Wadą jest znacznie dłuższy czas instalacji, który teraz obejmuje także dodatkową kompilację.

Programowanie aplikacji

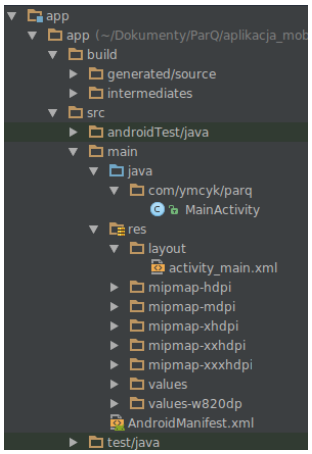
Aby móc tworzyć aplikacje na Androida z wykorzystaniem Javy, konieczne jest posiadanie:

- Java z JDK i JRE,
- Android SDK.

W ten sposób aplikacja do funkcji systemowych odwoływać się będzie za pomocą udostępnionego przez system API, przeznaczonego dla Javy. Dzięki temu, że nie jest to kod natywny, nie jest konieczna osobna kompilacja dla każdej dostępnej architektury. Programy są uniwersalne i dopiero po zainstalowaniu na konkretnym urządzeniu interpretowany jest kod bajtowy, bądź przeprowadzana kompilacja AOT. Przez twórców systemu udostępniane jest NDK, czyli Native Development Kit. Z jego pomocą aplikacje mogą być tworzone w C lub C++, odwołując się bezpośrednio do bibliotek systemowych. Niestety, mimo możliwego zysku na wydajności, stworzony kod jest zależny od architektury. Dodatkowo większość zewnętrznych bibliotek jest tworzonych w Javie.

Bardzo zalecane jest używanie zintegrowanego środowiska programistycznego, które automatyzuje niektóre czynności. Potrafi stworzyć za użytkownika projekt z wymaganą strukturą plików,

wykonać wszystkie etapy kompilacji, wgrać program na urządzenie i wiele innych. Dedykowanym IDE jest Android Studio, do niedawna wykorzystywany był domyślnie Eclipse.



Rys. 2.4: Pliki projektu utworzonego w Android Studio

Podczas działania aplikacja prezentuje użytkownikowi tzw. ekrany. Są to odpowiedniki okien systemowych, gdzie umieszczane są elementy graficznego interfejsu użytkownika, czyli widoki (ang. views), jak np.: przyciski, rozwijane listy, czy pola tekstowe. Wchodząc z nimi w interakcję, możliwa jest komunikacja między użytkownikiem, a urządzeniem.

Wygląd ekranów definiowany jest w plikach XML, nazywanych układami (ang. layout). Każdy z widoków jest osobnym znacznikiem, a za pomocą argumentów można modyfikować wybrane parametry jak rozmiar, czy kolor. Wszystkie widoki w pliku XML muszą posiadać swojego rodzica, który definiuje jak mają one być traktowane w tym układzie. W Androidzie dostępnych jest ich kilka, a do najpopularniejszych należą: RelativeLayout (położenie widoków określane jest względem siebie), LinearLayout (układ liniowy, gdzie elementy GUI wyświetlane są jeden koło drugiego) i GridLayout (dzieli ekran na siatkę, składającą się z wierszy oraz kolumn, i pozwala umieścić widoki we wskazanych komórkach).

Układy definiują jak dany ekran ma wyglądać, natomiast aktywności (ang. activity), czyli klasy Javy, określają w jaki sposób mają one reagować na interakcje użytkownika. Przechodząc do danego ekranu, tworzona jest najpierw aktywność. W metodzie onCreate, wywoływanej przed wyświetleniem ekranu, wybierany jest układ, który ma zostać użyty przez system do stworzenia interfejsu. Widoki z tego układu mogą mieć powiązane ze sobą jakieś operacje, które będą przeprowadzane np.: po naciśnięciu przycisku. To także odbywa się w aktywności, która może posiadać metody wywoływane po zakończeniu danej interakcji.

Rozwiązania mobilne cieszą się coraz większym zainteresowaniem. Tylko w sklepie z aplikacjami Androida, Google Play, liczba programów przekroczyła w 2015 r. 1,6 mln [7]. Powstające coraz to nowe urządzenia, na których zainstalowany jest Android sprawia, że umiejętność programowania na tą platformę będzie coraz bardziej doceniana.

2.4 Framework Django

Frameworki aplikacji internetowych powstały z myślą o zwolnieniu programisty z obowiązku pisania części kodu, który jest wspólny dla większości serwisów. Może to być dostęp do bazy danych, obsługa linków (ang. routing), czy zarządzanie sesjami. Dodatkowo, pisanie aplikacji internetowej od podstaw, jest zadaniem czasochłonnym oraz dość trudnym. Frameworki są odpowiedzią na te problemy, dostarczając zestaw gotowych oraz przetestowanych rozwiązań, które należy dostosować do własnych potrzeb. Umożliwiają utworzenie struktury plików projektu, na bazie którego dalej będzie rozwijana aplikacja. Każdy z najpopularniejszych języków programowania oferuje duży wybór silników, przeznaczonych do tworzenia usług internetowych i np.: w C# napisane zostały ASP.NET i MonoRail, w Javie - Spring i JavaServer Faces, a w Python - Django i Flask. Mogą się różnić przede wszystkim stopniem złożoności, dzięki czemu nadają się do różnych zastosowań - prezentują odmienne sposoby realizacji podobnych zadań. Wybór programisty będzie głównie zależny od jego prywatnych preferencji.

Początki Django

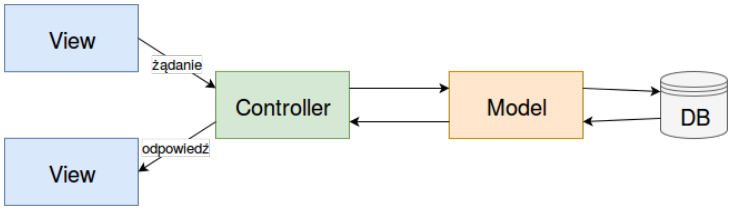
Django rozwijany jest jako wolne oprogramowanie na GitHub'ie, w ramach fundacji Django Software Foundation, ale skupiaja wokół siebie także wielu niezależnych twórców. Został napisany w Pythonie, stając się z czasem najpopularniejszym frameworkiem dla tego języka. Jego historia rozpoczęła się w 2003 r., kiedy to dwóch programistów Adrian Holovaty i Simon Willison zaczęli używać Pythona do tworzenia aplikacji webowych dla kilku serwisów informacyjnych, m.in. Lawrence.com. Praca w środowisku dziennikarskim, cechująca się napiętym grafikiem, wymagała niezwykle szybkiej realizacji zadań. Z tej konieczności opracowali własny silnik, który w 2005 r., już pod nazwą Django, został udostępniony publicznie. To właśnie szybkość oraz względna łatwość tworzenia aplikacji są jego głównymi zaletami.

Charakterystyka

Aplikacje Django tworzone są w interpretowanym języku Python, przeznaczonym głównie do pisania skryptów. Jako że jego interpretery dostępne są na wielu platformach, jest on niezależny od systemu operacyjnego. Dodatkowo wsparcie do wielu paradygmatów (imperatywnego, funkcyjnego, obiektowego), przekłada się na jego szerokie zastosowanie. Oprócz programowania serwerów, jest używany do pisania testów, aplikacji z graficznym interfejsem, a także gier 3D. Wyróżnia się głównie charakterystyczną składnią, w której poszczególne bloki kodu odseparowane są wcięciami (spacje lub tabulator). Wpływa to na zwiększoną czytelność kodu.

Architektura aplikacji sieciowych typu klient-serwer, opiera się często na wzorcu projektowym Model-View-Controller (pol. Model-Widok-Kontroler), w skrócie MVC. Jego ideą jest odsepa-

rowanie części prezentacji danych, od kodu odpowiedzialnego za ich przetwarzanie. Aplikacja dzielona jest w nim na trzy główne części. Model jest reprezentacją danych oraz logiki problemu. Widok odpowiedzialny jest za prezentację - określa w jaki sposób informacje mają zostać przedstawione. Kontroler przyjmuje żądania i wykonuje związane z nimi akcje. Głównie aktualizuje widoki oraz modele.



Rys. 2.5: Wzorzec projektowy MVC

Wzorzec MVC jest stosowany w Django, jednak został zrealizowany w sposób odmienny od najczęściej spotykanych jego implementacji. Z tego względu często nazywa się go wzorcem MTV (Model-Template-View), będącego pewną wariacją MVC. Także wyróżnia się w nim trzy główne części aplikacji, a są to:

- Model – podobnie jak w MVC, zapewnia dostęp do danych. Opisane są tutaj relacje między danymi oraz odbywa się ich walidacja.
- Template (pol. szablon) – warstwa prezentacji, czyli jak ma zostać wygenerowana odpowiedź (w postaci dokumentu HTML lub innym formacie).
- View (pol. widok) – zawiera logikę biznesową. Pobiera dane z modeli i łączy je z szablonami, tworząc w ten sposób odpowiedź. Stanowi pomost między dwoma wcześniej wymienionymi elementami MTV.

Rolę kontrolera MVC pełni w Django sam framework. Otrzymane zapytanie wysyłane jest do odpowiedniego widoku, w zależności od konfiguracji URL.

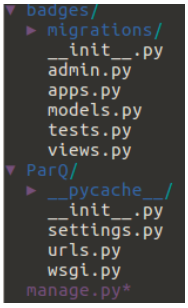
Listing 2.3: Przykładowa konfiguracja URL w Django

```
urlpatterns = [
    url(r'index/^$', views.index, name='index'),
    url(r'^admin/', admin.site.urls),
]
```

W przeciwieństwie do mikro frameworków, takich jak Flask, Django dostarcza programiście wielu gotowych rozwiązań, jak: dostęp do baz danych, klasy ORM, obsługa linków URL, uprawnienia użytkowników, automatycznie generowana strona administratora, szablony HTML i wiele innych. Dodatkowo posiada wbudowaną ochronę przed typowymi atakami, takimi jak: cross-site scripting (XSS), cross-site request forgery (CSRF), a także wstrzykiwanie SQL'a.

Programowanie aplikacji

Tak jak większość frameworków, także i Django tworzy za użytkownika gotową strukturę projektu. Poza konfiguracją, nie znajduje się w nim jednak żadna logika. Wszystkie modele oraz widoki w danym projekcie tworzone są w ramach tzw. aplikacji, czyli pakietów Pythona (których struktura także generowana jest przez framework), odpowiednio w plikach models.py oraz views.py. Dzięki temu mogą zostać one ponownie wykorzystane. W hierarchii plików znajdują się na tym samym poziomie, co projekt.



Rys. 2.6: Struktura plików projektu (ParQ) z aplikacją (badges)

W pliku settings.py projektu znajdują się wszystkie ustawienia serwisu, takie jak konfiguracja bazy danych, strefy czasowej, silnika szablonów oraz pakietów pośredniczących (middleware) wykorzystywanych np.: do autoryzacji. Tam także powinny zostać zarejestrowane wszystkie aplikacje używane w projekcie.

W models.py aplikacji znajdują się modele danych - są to klasy dziedziczące po Model. Posiadają zmienne klasowe, które reprezentują kolumny w tabelach baz danych. Nazwa takiego pola jest później używana jako nazwa kolumny, natomiast jej typ definiowany jest przez instancję jednej z klas pochodnych od Field. I tak dla przykładu IntegerField będzie typem całkowitoliczbowym, a CharField znakowym. Parametry podane w konstruktorze pozwalają zdefiniować rozmiar, czy unikalność krotki w kolumnie. Relacje również tworzone są za pomocą pól. Model posiadający pole relacji, może odwoływać się za jego pomocą do powiązanych danych. W Django znajdują się trzy takie klasy:

- ForeignKey – pole z kluczem obcym, używane w relacjach jeden do wielu. Tworzona jest kolumna w tabeli.
- ManyToManyField – używane w relacjach wiele do wielu. W bazie danych zostanie automatycznie utworzona tabela pośrednia, z kluczami powiązanych tabel.
- OneToOneField – do relacji jeden do jeden. Także wykorzystywana jest tabela pośrednia, jednakże oba klucze są unikalne - mogą wystąpić tylko w jednej relacji w ramach tej tabeli.

W modelach dozwolone jest także definiowanie własnych metod.

Listing 2.4: Przykładowy model danych z modułu models.py

```
from django.db import models
from charges.models import ScheduleLot

class Parking(models.Model):
    name = models.CharField(
            _('name'),
            max_length=50,
            )
    description = models.CharField(
            _('description'),
            max_length=150,
            blank=True,
            null=True,
            )
    schedule_lot = models.OneToOne(ScheduleLot)

    def __str__(self):
        return self.name
```

Kolejnym ważnym plikiem w aplikacji tworzonej przez użytkownika jest views.py. To tutaj znajdują się widoki ze wzorca MTV i scalają one szablony z modelami. Jako parametr przyjmują obiekt HttpRequest, zawierający dane zawarte w żądaniu HTTP. To właśnie te widoki podawane są podczas konfiguracji linków w funkcji url, pliku urls.py w projekcie. Zostało to zaprezentowane na listingu 2.3.

Listing 2.5: Widok

```
from django.shortcuts import render
from badges.models import Parking

def index(request):
    parking = Parking.objects.get()
    return render(request, 'badges/index.html', {'p_name': parking.name})
```

Aplikacje jako dodatki

Napisane aplikacje Django, mogą być wielokrotnie wykorzystywane w innych projektach. Na tej zasadzie funkcjonują dodatki pisane do tego frameworku. Jednymi z nich są: Django REST Framework używany do tworzenia API w architekturze REST, czy django-annoying modyfikujący działanie niektórych elementów silnika.

3 Opłaty w strefie płatnego parkowania

W tym rozdziale znajdują się informacje dotyczące szczecińskiej Strefy Płatnego Parkowania. Przedstawione są tutaj m.in. sposoby naliczania opłat, cennik, czy taryfikator. Informacje i wymagania związane z SPP są istotne dla tworzonego systemu, ponieważ to na ich bazie została zaimplementowana w nim logika biznesowa. W drugiej części tego rozdziału przedstawione są podobne rozwiązania do tworzonego systemu, które aktualnie funkcjonują w Szczecinie.

3.1 Strefa Płatnego Parkowania w Szczecinie

Zgodnie z obwieszczeniem Rady Miasta Szczecin z dnia 26 maja 2014 r. [23] strefa płatnego parkowania w Szczecinie podzielona jest na dwie podstrefy: A i B. W każdej z nich obowiązuje inny cennik, który został przedstawiony w tabeli 3.1. W niej znajdują się także różne stawki, a to które z nich będą brane pod uwagę podczas naliczania opłaty, zależny od czasu postoju. Parkowanie w strefie jest płatne w dni robocze, w godzinach od 8:00 do 17:00 z wyjątkiem dni wolnych od pracy.

Tab. 3.1: Stawki w strefie płatnego parkowania w Szczecinie

L.p.	Opłaty jednorazowe	Kwota w zł.	
		Podstrefa A	Podstrefa B
1.	do 15 minut	0,70	0,40
2.	pierwsza godzina	2,80	1,60
3.	druga godzina	3,20	1,80
4.	trzecia godzina	3,60	2,00
5.	każda kolejna godzina	2,80	1,60

Ważna jest także informacja odnośnie czasu w którym bilety obwiązują, a mianowicie moment zakupu jest także chwilą, w której zaczynają być ważne. Nie ma możliwości zakupu biletów tzw. “do przodu”, których czas rozpoczęcia jest późniejszy od daty zakupu [11].

3.2 Zakup i kontrola biletu

Bilety postojowe są dostępne do kupienia w parkomatach, rozmieszczonych w miejscach gdzie obowiązuje strefa płatnego parkowania. Długość postoju ustalana jest na bazie kwoty, jaka została wpłacona oraz podstrefy, w której bilet został kupiony. Płatność może być realizowana gotówką, ale także przy użyciu karty SKA (Szczecińska Karta Aglomeracyjna), która w tym przypadku działa jak wirtualna portmonetka. Otrzymany wydruk potwierdzający opłacenie miejsca, należy umieścić za przednią szybą pojazdu. Kontroler sprawdza czy bilet został kupiony na odpowiednią podstrefę i czy nie przekroczono czasu postoju.

Oprócz parkomatów, od pewnego czasu w Szczecinie bilet może być kupiony także poprzez aplikację na urządzenia mobilne, a jednym z takich rozwiązań jest system moBiLET. Dostępny w wielu miastach Polski, pozwala na zakup nie tylko biletów postojowych, ale także komunikacji miejskiej, czy kolejowych. Po pobraniu aplikacji i zarejestrowaniu, niezbędne jest jeszcze doładowanie wirtualnej portmonetki, powiązanej z kontem użytkownika. W przypadku korzystania z strefy płatnego parkowania, konieczne jest jeszcze dodanie samochodu oraz umieszczenie w nim informacji, że bilet dla danego pojazdu został opłacony za pomocą tej właśnie aplikacji. Dzięki tej informacji, kontroler po wprowadzeniu numeru rejestracyjnego pojazdu, będzie mógł sprawdzić ważność biletu.

System ParQ

Tworzony w ramach pracy system ParQ różni się od istniejących rozwiązań wykorzystaniem kodów QR. Każdy z pojazdów będzie musiał posiadać plakietkę z tym kodem, gdyż to za ich pomocą odbywa się identyfikacja pojazdów w systemie. Z perspektywy kierowcy, działanie tego systemu jest zbliżone do istniejących już rozwiązań. Po założeniu oraz doładowaniu konta odpowiednią kwotą, konieczne jest jeszcze dodanie swojego pojazdu. Po tym kroku kierowca otrzyma wiadomość e-mail z kodem QR, który powinien zostać umieszczony pod przednią szybą pojazdu. Od tego momentu wystarczy jedynie kupić bilet.

Istotna różni się natomiast sposób przeprowadzania kontroli. Dzięki zastosowaniu kodów QR, nie ma potrzeby ręcznego wprowadzania numeru tablicy rejestracyjnej. Osoba sprawdzająca bilet będzie musiała jedynie zeskanować za pomocą aparatu wbudowanego w telefon plakietkę, znajdująca się pod przednią szybą pojazdu. Następnie informacja zwrócona z serwera tego systemu powiadomi go o tym, czy dany samochód ma opłacony bilet postojowy.

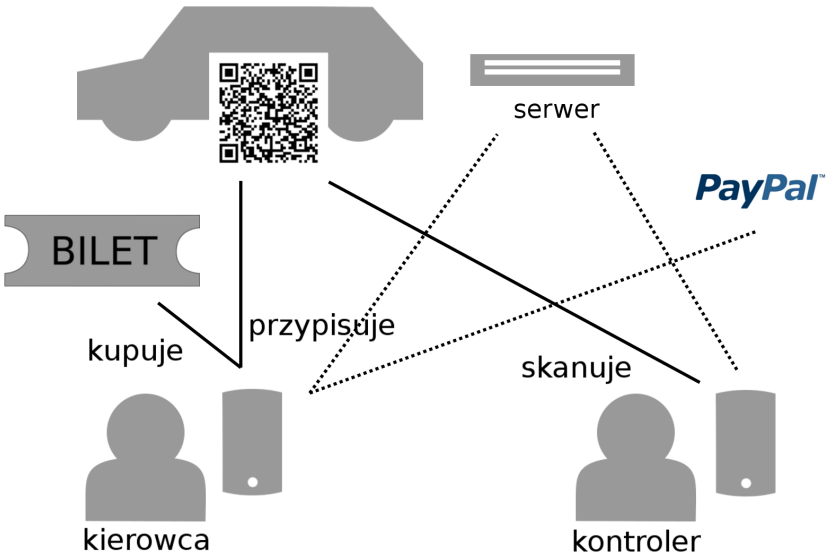
4 Projekt systemu

W tym rozdziale zawarta została dokumentacja techniczna systemu strefy parkowania. Na samym początku przedstawiony jest schemat, opisujący sposób jego działania. W dalszej części opisana została specyfikacja wymagań, z definicją systemu oraz wymaganiami funkcjonalnymi i niefunkcjonalnymi. Pozostała część rozdziału została poświęcona diagramom UML i opisowi API, wykorzystywanego do komunikacji serwera z aplikacją mobilną.

4.1 Schemat działania systemu ParQ

Stworzony w ramach tej pracy system płatności dostosowany jest do potrzeb Strefy Płatnego Parkowania w Szczecinie. Umożliwia on utworzenie taryfikatora wraz z opłatami, które zmieniają się wraz z czasem postoju. System nazwany został ParQ, umożliwia dokonanie zakupu biletu postojowego, oraz jego późniejszą kontrolę. Obie te czynności wykonywane są w oparciu o generowany dla każdego pojazdu, unikalny kod UUID. To właśnie on znajduje się na plakietkach z kodem QR. Po jego zeskanowaniu, kontroler uzyska informacje o danym pojeździe.

Cała funkcjonalność systemu dostępna jest dla jego użytkowników za pośrednictwem aplikacji na urządzeniach mobilnych. Wykonane one zostały w formie tzw. “cienkiego klienta”, co oznacza, że nie są przeprowadzane na nich żadne skomplikowane operacje. Komunikują się z serwerem wysyłając do niego żądania, a użytkownikowi prezentowana jest odebrana odpowiedź. Do skorzystania z systemu niezbędne jest posiadanie wcześniej założonego konta.



Rys. 4.1: Schemat działania systemu

Z każdym kontem kierowcy w systemie powiązana jest jego wirtualna portmonetka. Są to elektroniczne pieniądze (w postaci krotki z ilością pieniędzy w bazie danych), jakimi dokonywać

będzie on opłat za bilety postojowe. Po zarejestrowaniu użytkownik musi doładować swoje konto. Wykonuje to także za pośrednictwem aplikacji mobilnej, z wykorzystaniem bramki płatności, a wielkość doładowania zależy od kwoty, jaka została wpłacona. Po tej operacji, a także zarejestrowaniu swojego pojazdu w systemie, możliwy staje się zakup biletu postojowego.

Transakcja związana z dodaniem środków na konto w systemie odbywa się z wykorzystaniem usług PayPal'a, który udostępnia na Androida specjalną bibliotekę. To ona odpowiedzialna jest za komunikację z serwerami tego usługodawcy płatności. Kierowcy, po podaniu kwoty jaką chce zasilić konto, prezentowany jest ekran (także udostępniany w ramach tej biblioteki), gdzie dokonywany jest wybór metody płatności i autoryzacja. Po tych krokach, a także transakcji przeprowadzonej z sukcesem, zwrócony zostaje identyfikator płatności. Ten unikalny ciąg znaków aplikacja wysyła już do serwera systemu ParQ. On, komunikując się z PayPal'em, uzyskuje informacje o kwocie, jak została wpłacona. Po otrzymaniu odpowiedzi, konto kierowcy zostaje zasilone. Co istotne, identyfikator transakcji zapisywany jest w bazie danych, dzięki czemu ta sama wpłata nie zostanie naliczona więcej niż jeden raz.

Osoba przeprowadzająca kontrolę potrzebuje do swojej pracy urządzenia mobilnego ze sprawnym aparatem. Sprawdzenie ważności biletu odbywa się poprzez zeskanowanie plakietki z kodem identyfikacyjnym pojazdu. Odpowiedź zwrócona przez serwer zawiera informacje o danym pojeździe, a także o braku lub posiadaniu przypisanego biletu postojowego.

Moduły

Aplikacja internetowa został podzielona na pięć modułów:

- badges – klasy pojazdu i kodu identyfikacyjnego. W tym miejscu generowany jest także kod QR.
- charges – tworzenie taryfikatora oraz grafiku. Tutaj zostaje naliczona opłata.
- parkings – klasa parkingu oraz biletu.
- paypal – komunikacja z PayPal'em. W tym miejscu zostaje zapisany numer identyfikacyjny transakcji.
- users – klasy kierowcy oraz kontrolera.

Aplikacje mobilne zawierają klasy aktywności, które są powiązane z klasami obsługującymi komunikację z serwerem.

4.2 Specyfikacja wymagań

Definicja systemu

System przeznaczony jest dla miejskich stref płatnego parkowania, które chcą wykorzystać w swojej pracy urządzenia mobilne. Klienci zyskują możliwość bardziej elastycznego dokonywania opłat. Kontrolerzy, dzięki zastosowaniu kodów graficznych QR, mogą znacznie szybciej przeprowadzać kontrolę postoju pojazdu. Wystarczy zeskanowanie aparatem urządzenia mobilnego plakietki z kodem identyfikacyjnym.

Technologie

Aplikacja internetowa została napisana w języku Python 3, z wykorzystaniem frameworka Django 1.10. Część mobilna wykonana jest w systemie Android i języku Java. Płatności odbywają się za pośrednictwem systemu PayPal. Do realizacji zadań konieczne było wykorzystanie zewnętrznych bibliotek oraz rozszerzeń.

Wymagania funkcjonalne

Wymagania funkcjonalne stanowią zbiór wszystkich funkcjonalności, udostępnianych przez system. Dotyczą one zarówno aplikacji mobilnej, jak i serwerowej. Wszelka interakcja kierowcy oraz kontrolera z systemem, odbywa się jedynie za pośrednictwem aplikacji mobilnej.

1. Zarządzanie kontem użytkownika
    - System posiada dwie role użytkowników, którzy mogą się zalogować w aplikacji mobilnej: kierowcę oraz kontrolera.
    - Kierowca może się zarejestrować w systemie.
    - Kierowca może zalogować się do systemu.
    - Podczas zakładania konta kierowcy, wymagane są: nazwa użytkownika, e-mail oraz hasło.
    - Nazwa użytkownika jest unikalna w całym systemie, niezależnie od jego roli.
    - Kierowca może przeglądać informacje podane podczas rejestracji.
    - Konto kontrolera tworzone jest przez administratora.
2. Zarządzanie pojazdami
    - Kierowca może zarejestrować swoje pojazdy w systemie.

- Kierowca podczas rejestracji pojazdu musi podać nazwę, kraj i numer rejestracji.
- Kierowca może usunąć pojazd przypisany do swojego konta.
- Kierowca może przeglądać wszystkie dodane przez siebie pojazdy.
- Kierowca po zarejestrowaniu pojazdu otrzymuje na maila wiadomość z kodem QR, używanego do identyfikacji pojazdu.
- Pojazd może być dodawany jedynie przez użytkownika, który ma przypisaną rolę kierowcy.

3. Zakup biletu postojowego
    - Kierowca kupując bilet określa czas postoju oraz parking w jakim chce dany bilet kupić.
    - Kierowca może zakupić bilet, jeżeli posiada środki w wirtualnej portmonetce oraz przynajmniej jeden zarejestrowany pojazd w systemie.
    - Bilet obowiązuje od chwili zakupu.
    - Nie ma możliwości zakupu biletu na datę inną, niż moment zakupu.
    - Kierowca może kupić bilet jedynie dla pojazdu przypisanego do jego konta.
    - Kierowca nie może zakupić biletu, jeżeli nie posiada wystarczającej ilości środków.
    - Bilet może obowiązywać tylko na jeden taryfikator.
    - Bilet nie może zostać kupiony, jeżeli nie obowiązuje żaden taryfikator.
    - Zakupu biletu postojowego może dokonać jedynie użytkownik, który ma przypisaną rolę kierowcy.
4. Płatności
    - W systemie jedynie kierowca posiada wirtualną portmonetkę.
    - Kierowca ma wgląd w stan konta wirtualnej portmonetki.
    - Aplikacja dla kierowcy jest zintegrowana z bramką płatności.
    - Kierowca może dokonać płatności w aplikacji mobilnej, celem doładowania konta.
5. Kontrola biletu postojowego
    - Kontrolę biletu może przeprowadzać jedynie zalogowany użytkownik, który ma przypisaną rolę kontrolera.
    - Kontrola odbywa się poprzez zeskanowanie kodu QR z identyfikatorem pojazdu.

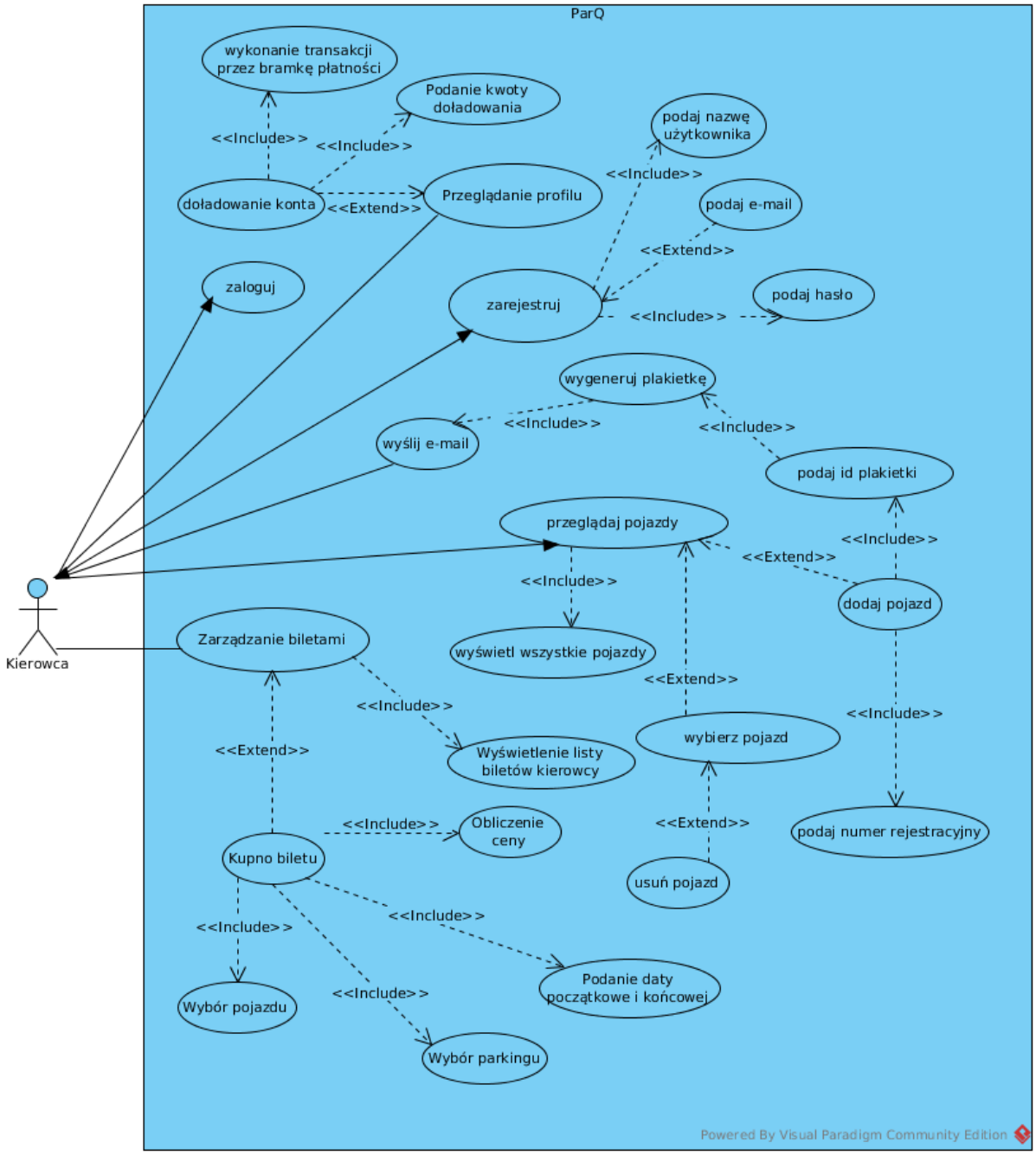
- Kontroler po zeskanowaniu otrzymuje informacje o numerze rejestracyjnym pojazdu, celem sprawdzenia, czy dany identyfikator jest przypisany do kontrolowanego pojazdu.
- Kontroler otrzymuje informację o ważności postoju – czy bilet został zakupiony i czy jest ważny oraz na jaki parking został zakupiony.

6. Zarządzanie parkingami
    - System pozwala na dodawanie i usuwanie parkingów.
    - System pozwala na przeglądanie dodanych pojazdów.
    - System pozwala na tworzenie i usuwanie taryfikatorów opłat.
    - System pozwala na przeglądanie istniejących taryfikatorów.
    - System pozwala na określenie czasu, w jakim dany taryfikator będzie obowiązywał.
    - System pozwala na przypisanie taryfikatora do parkingu.

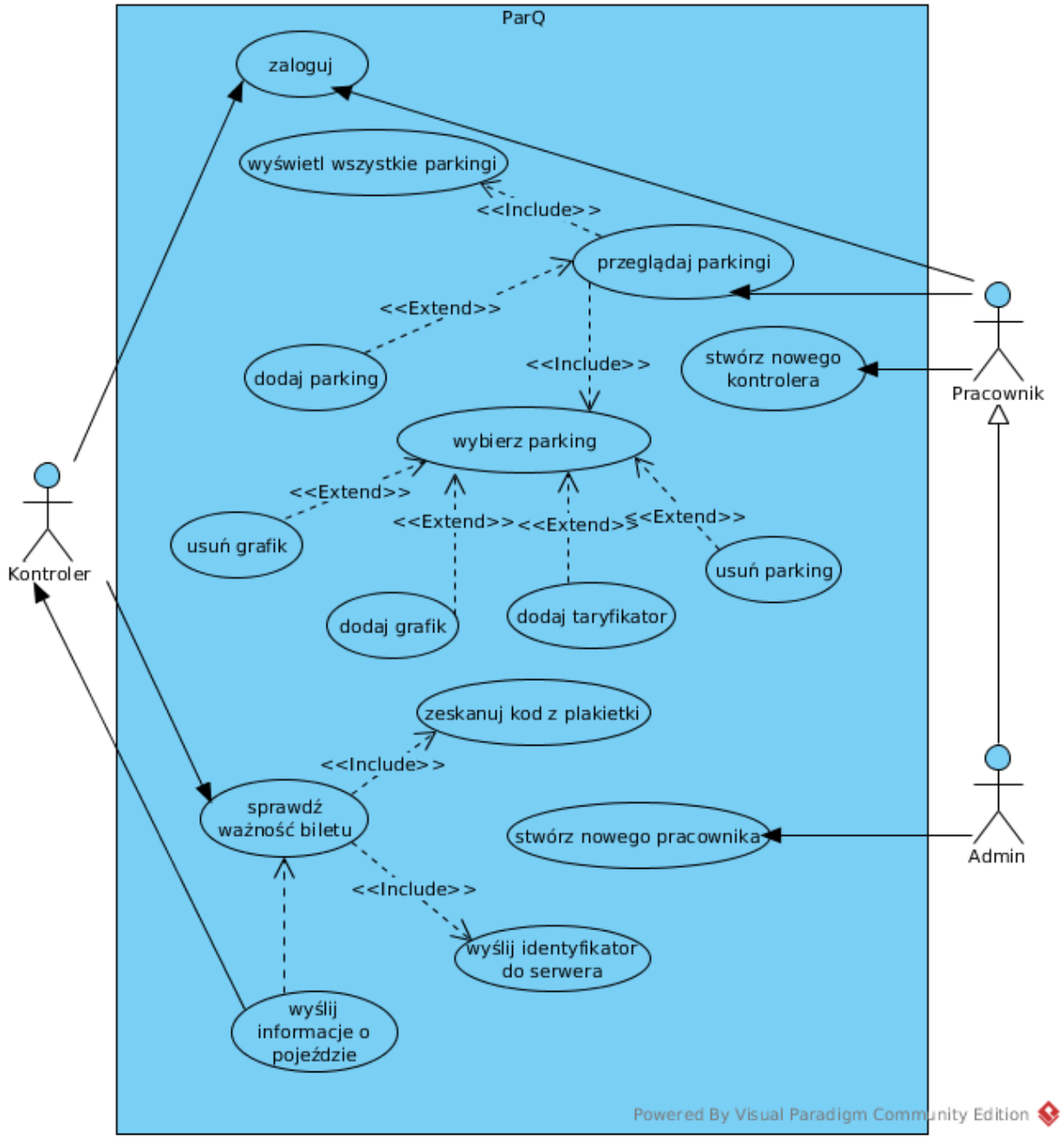
Wymagania niefunkcjonalne

- Odpowiedź z serwera powinna być wysłana w czasie nie dłuższym niż 5 sekund.
- Aplikacja mobilna powinna działać na systemie Android, wersja 4.1 (API 16) i wyższych.
- Aplikacja mobilna powinna posiadać prosty interfejs graficzny.
- Zakup biletu w aplikacji mobilnej powinien trwać nie dłużej niż 5 sekund.
- Uzupełnienie konta w systemie możliwe jest z wykorzystaniem kart płatniczych.
- System powinien być w stanie obsłużyć przynajmniej 400 tyś. użytkowników.
- Opłata za bilet naliczana jest za każdą minutę postoju.

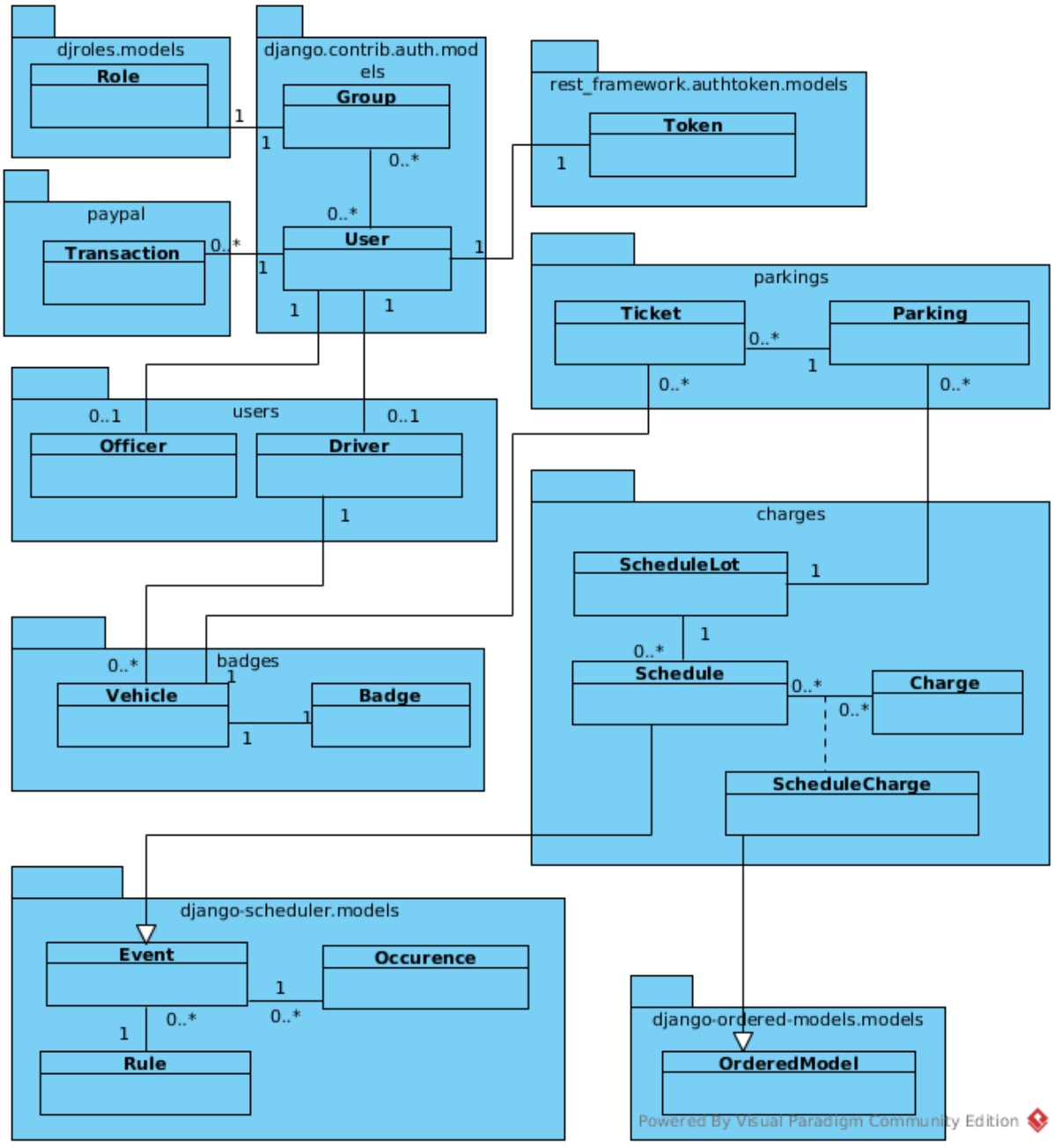
4.3 Diagramy UML



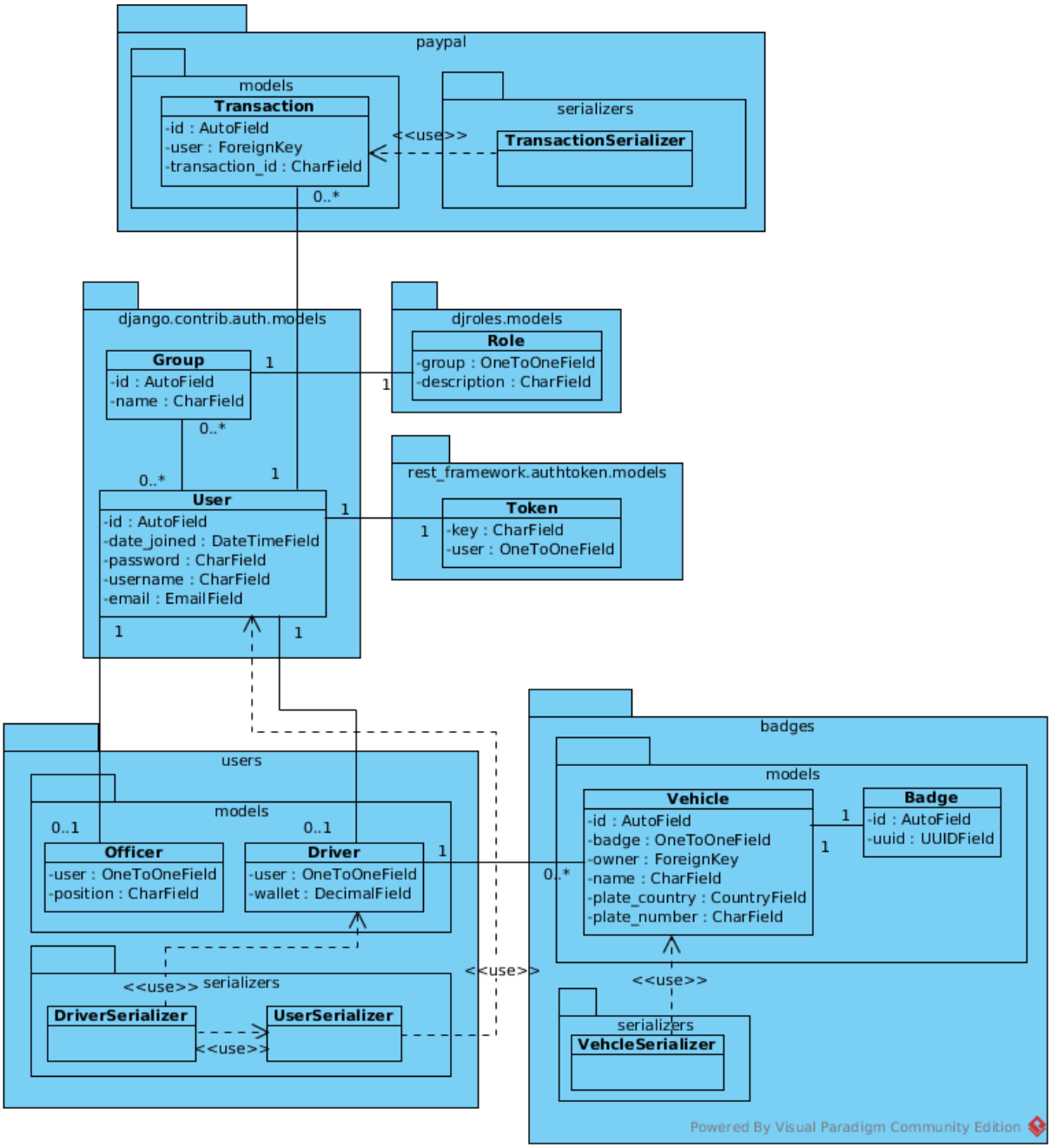
Rys. 4.2: Przypadki użycia dla kierowcy



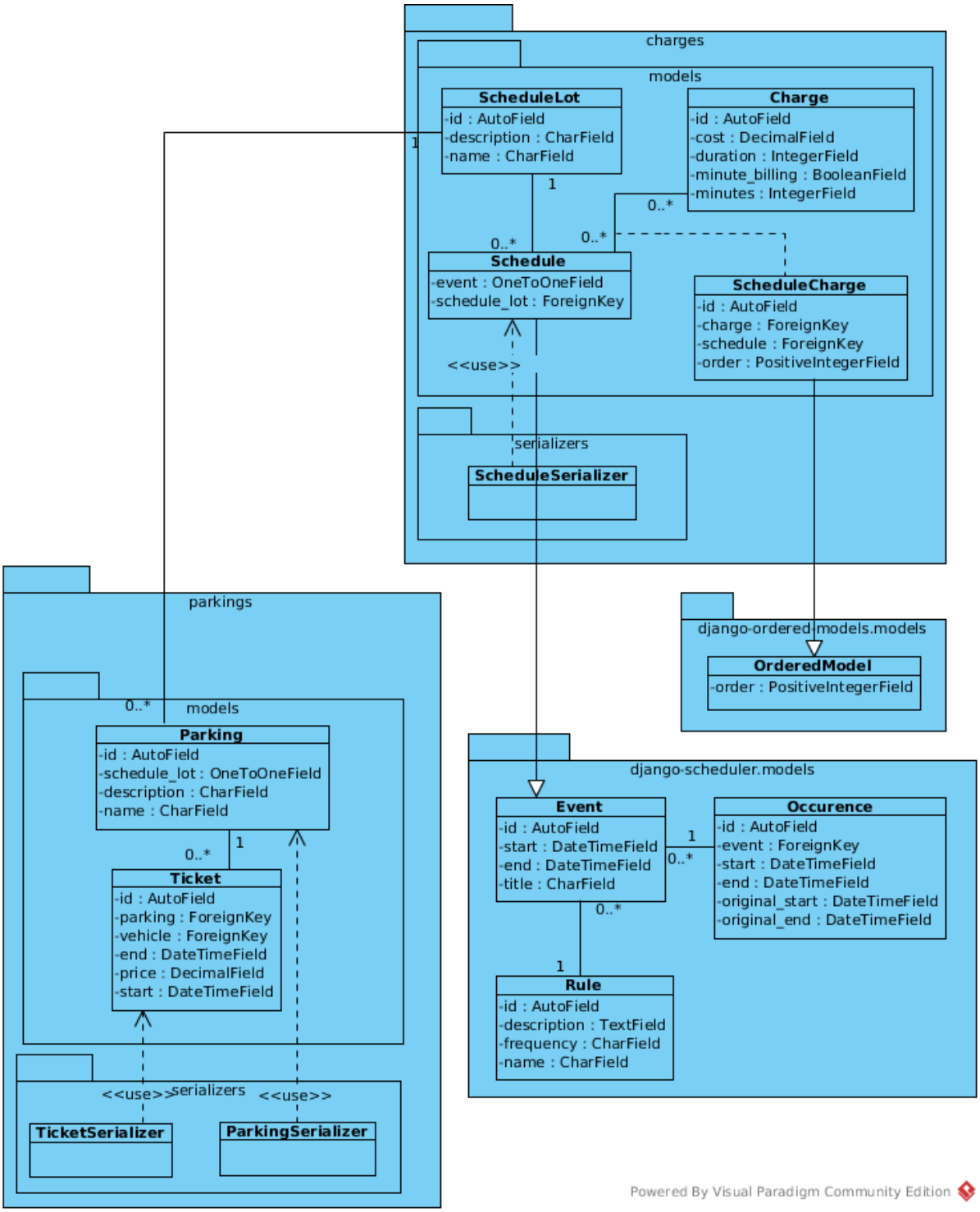
Rys. 4.3: Przypadki użycia dla kontrolera, pracownika i administratora



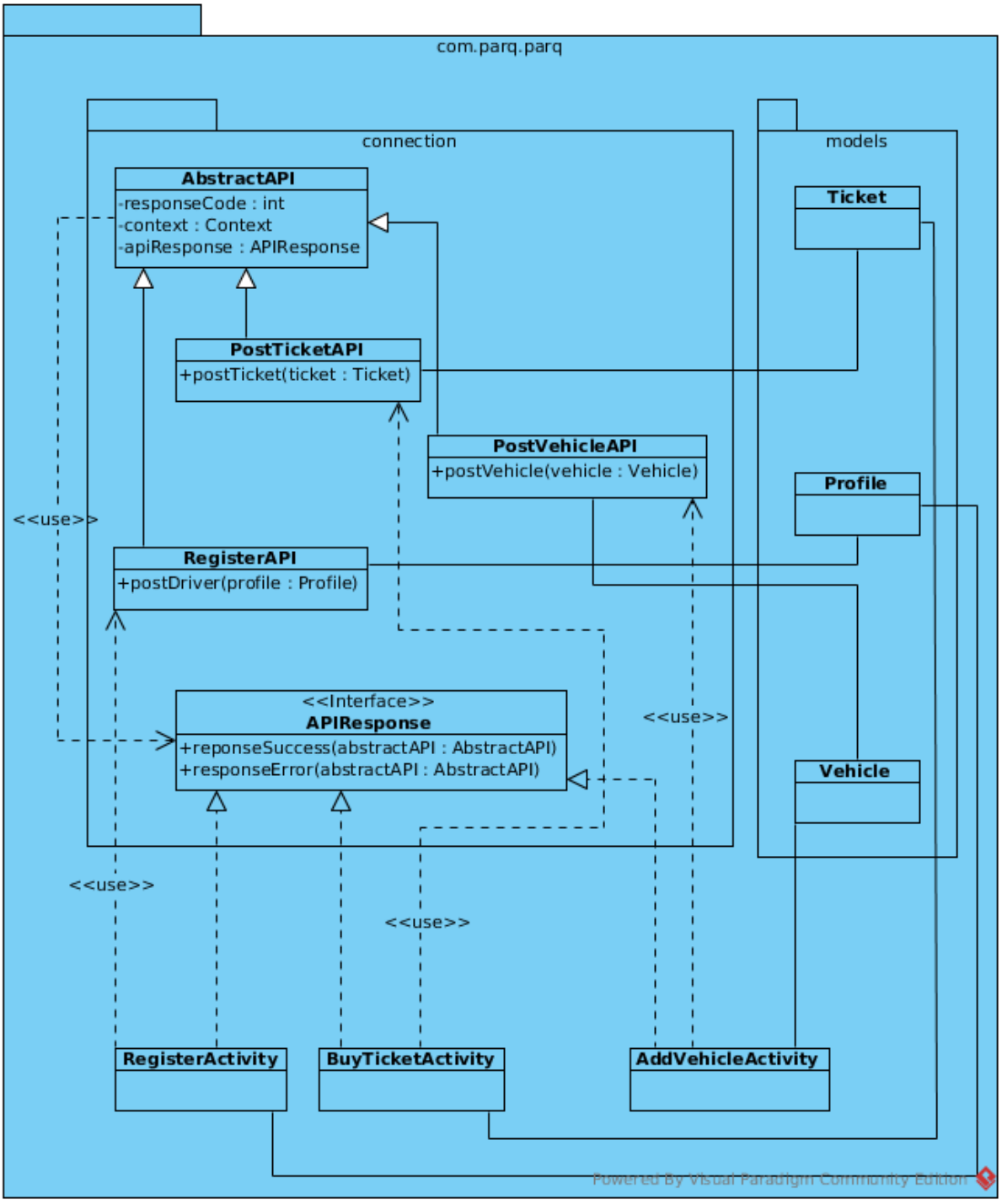
Rys. 4.4: Diagram pakietów



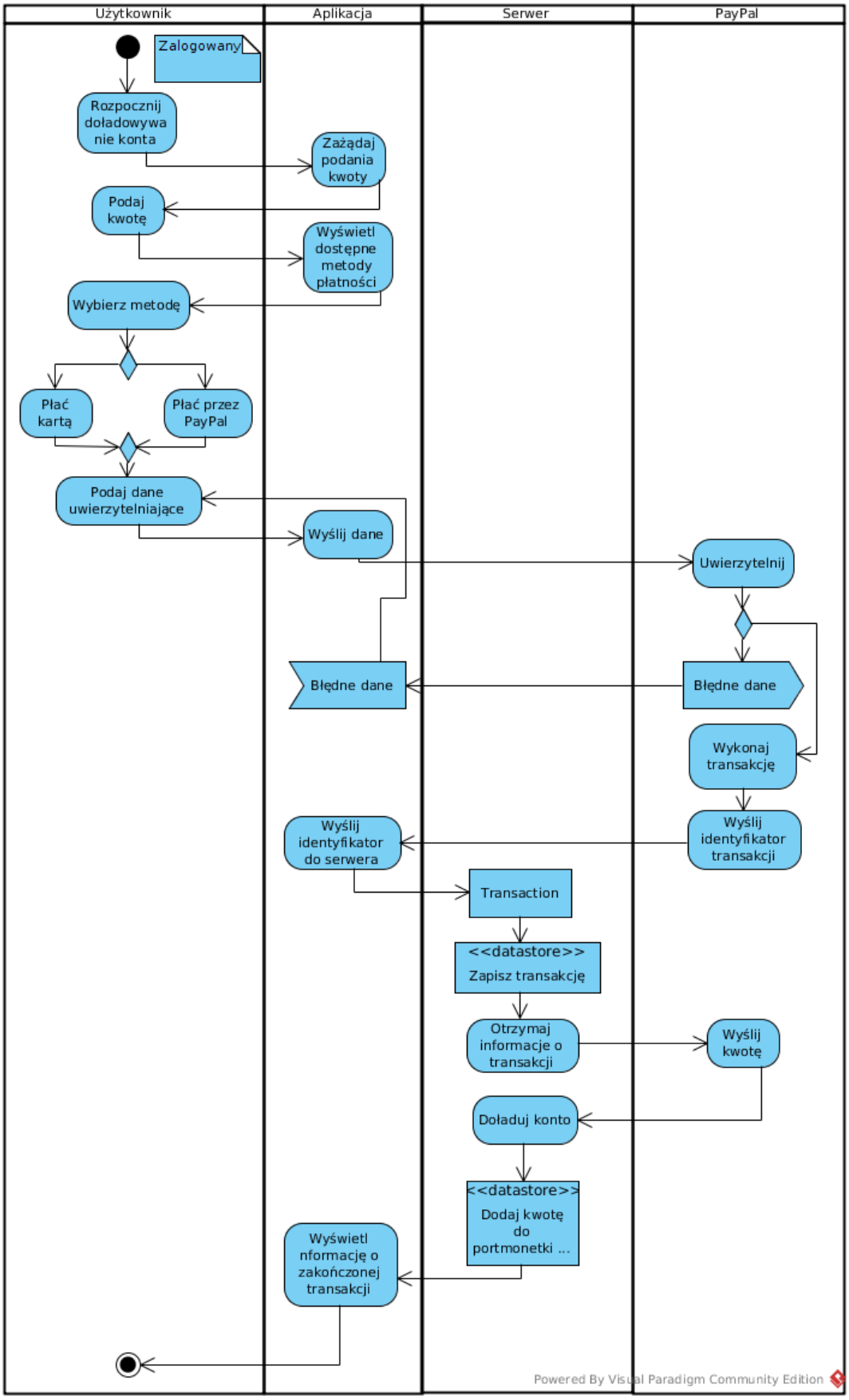
Rys. 4.5: Diagram klas dla pakietów paypal, users, badges oraz powiązanych



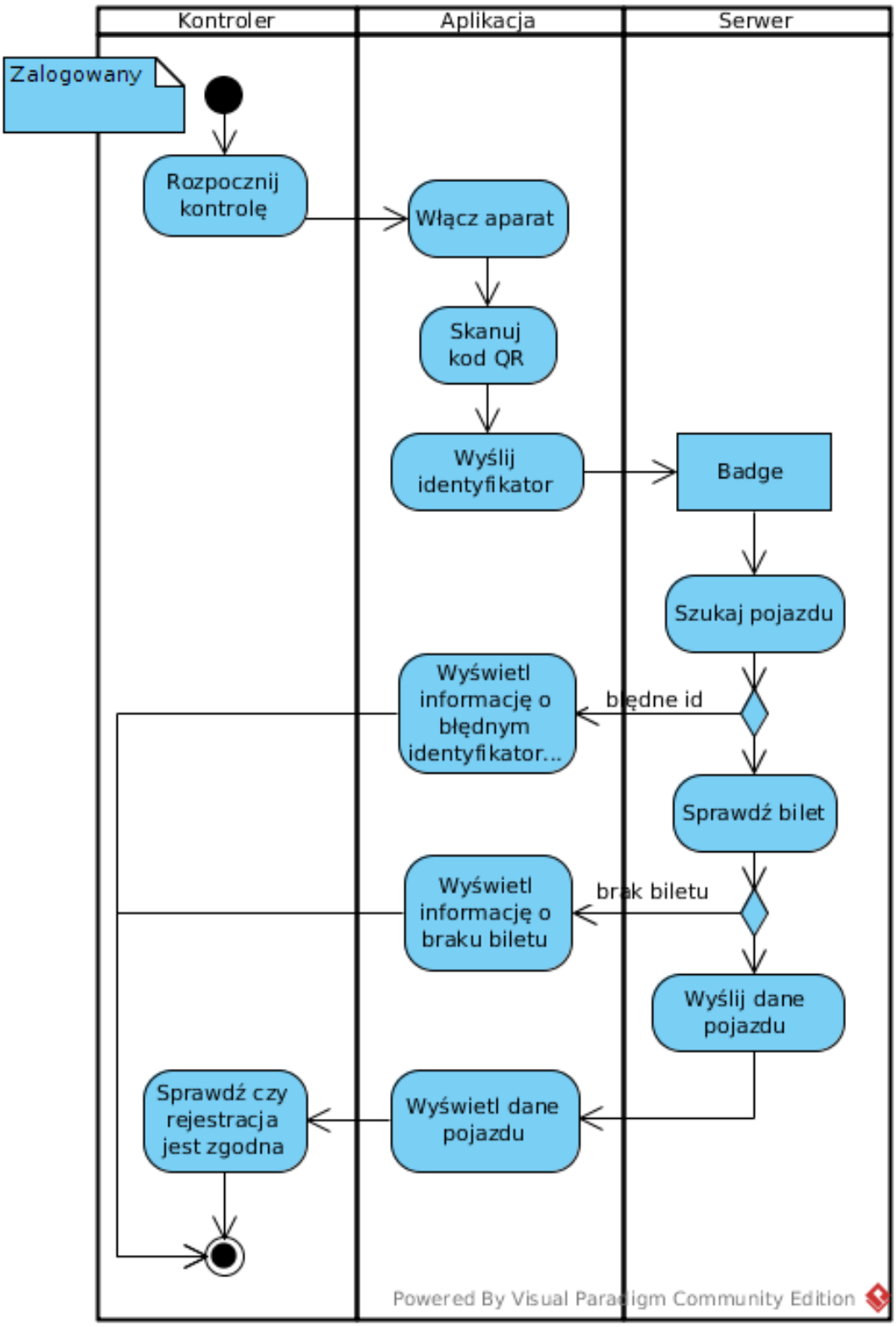
Rys. 4.6: Diagram klas dla pakietów parkings, charges oraz powiązanych



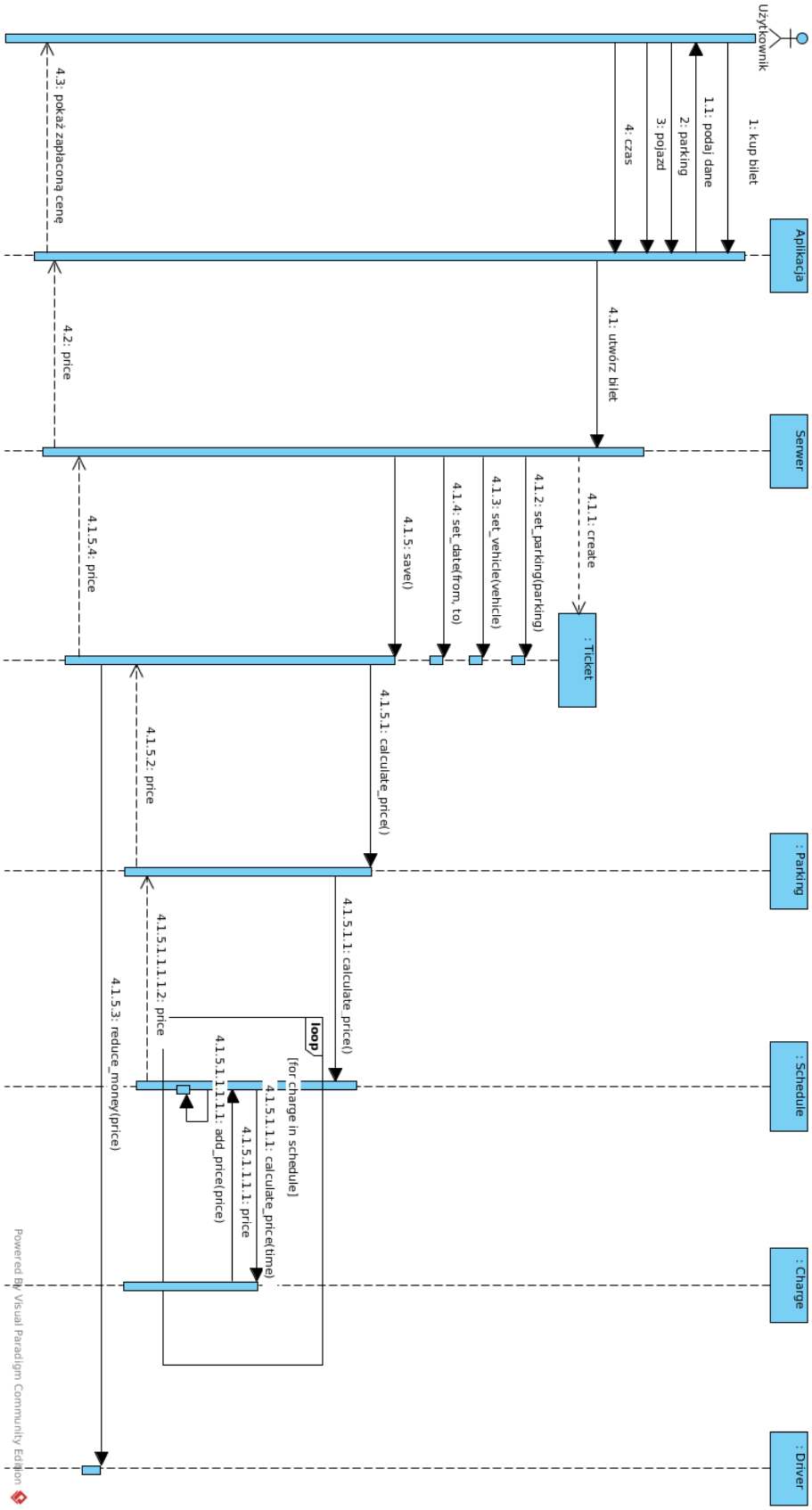
Rys. 4.7: Diagram klas rejestracji, dodawania pojazdu i zakupu biletu aplikacji mobilnej



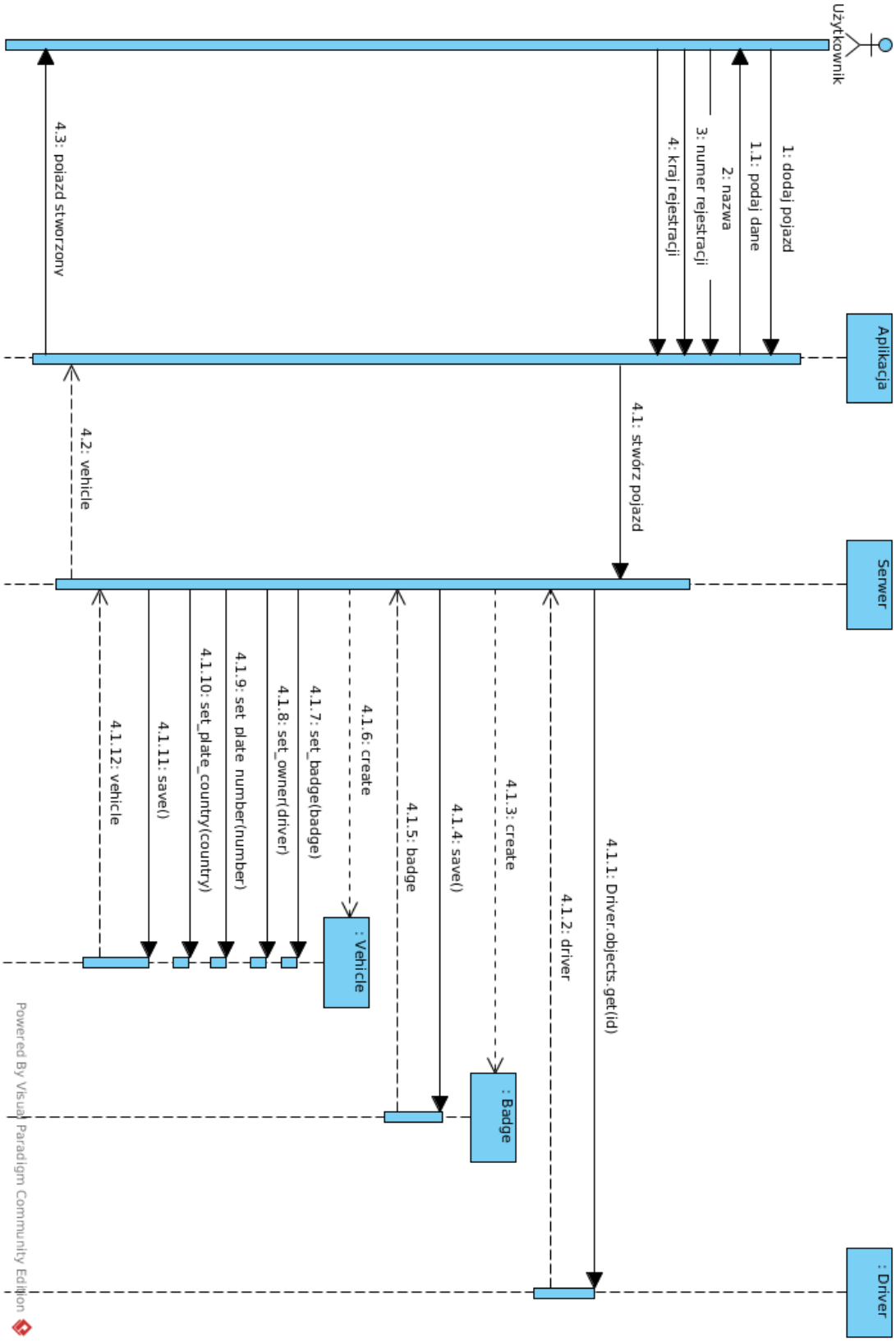
Rys. 4.8: Diagram aktywności doładowywania portmonetki



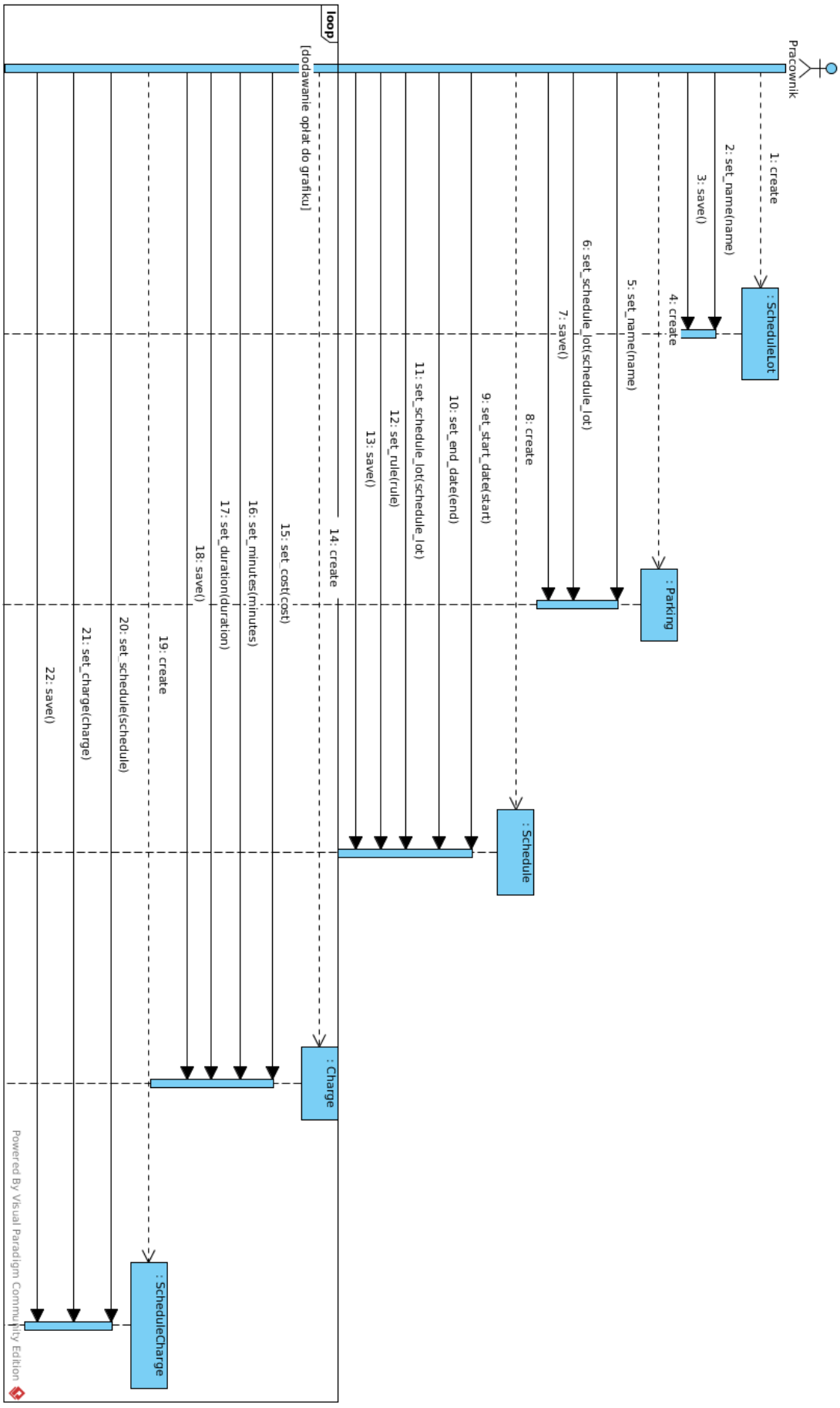
Rys. 4.9: Diagram aktywności kontrolowania biletu



Rys. 4.10: Diagram sekwencji zakupu biletu postojowego



Rys. 4.11: Diagram sekwencji dodawania nowego pojazdu



Rys. 4.12: Diagram sekwencji dodawania nowego parkingu oraz opłat

4.4 Opis REST API

Autoryzacja użytkowników w systemie opera się na tokenie, generowanym podczas zakładania konta. Jest on niezbędny do komunikacji serwera z aplikacją mobilną, która uzyskuje go podczas logowania. Dołączany jest on później do nagłówka każdego żądania (oprócz rejestracji oraz logowania).

Listing 4.1: Dane umieszczane w nagłówkach żądań

```
Content-Type:    application/json/
Authorization:   Token 84f41d74832b9a7c95e8120ca856f1f29cdaa7cf
```

Poniżej znajdują się żądania wysyłane przez aplikację mobilną oraz odpowiedzi, jakie otrzyma w przypadku poprawnych zapytań.

1. Logowanie do systemu
   - Adres: /api/login/
   - Metoda: POST
   - Dane żądania:

```
{
    "username": "admin",
    "password": "12345",
    "role": "driver"
}
```

   - Odpowiedź:

```
{
    "token":"a059e251e2bd105b1987c99ade84d320eac747d1"
}
```

2. Zakładanie konta
   - Adres: /api/register/
   - Metoda: POST
   - Dane żądania:

```
{
    "username": "admin",
    "email": "admin@o2.pl",
    "password": "12345"
}
```

   - Odpowiedź:

```
{
    "id": 1,
    "username": "admin",
    "email": "admin@o2.pl"
}
```

3. Lista pojazdów
   - Adres: /api/vehicles/

- Metoda: GET
- Odpowiedź:

```
[{
    "id":5,
    "owner":5,
    "badge":"41203830-4e4d-49d5-aac0-f9cb8a65d7ed",
    "name":"Maluch",
    "plate_country":"PL",
    "plate_number":"ZS11111"
}]
```

4. Dodawanie pojazdu
   - Adres: /api/vehicles/id
   - Metoda: POST
   - Dane żądania:

```
{
    "name": "Golf",
    "plate_number": "ZS12ER3",
    "plate_country": "PL"
}
```

   - Odpowiedź:

```
{
    "id": 2,
    "name": "Golf",
    "plate_number": "ZS12ER3",
    "plate_country": "PL"
}
```

5. Usuwanie pojazdu
   - Adres: /api/vehicles/id
   - Metoda: DELETE
6. Lista parkingów z godzinami otwarcia
   - Adres: /api/parkings/
   - Metoda: GET
   - Odpowiedź:

```
[{
    "id":1,
    "name":"Strefa A",
    "description":"",
    "open":{
        "start":"2017-01-16T08:00:00Z",
        "end":"2017-01-16T17:00:00Z"
    }
}]
```

7. Zakup biletu
   - Adres: /api/tickets/

- Metoda: POST
- Dane żądania:

```
{
    "vehicle": "1",
    "parking": "1",
    "minutes": "15"
}
```

- Odpowiedź:

```
[{
    "start":"2017-01-02T07:00:00Z",
    "end":"2017-01-02T08:00:00Z",
    "vehicle":{
        "id":1,"owner":1,
        "badge":"18a131b1-7ff0-412b-8d6a-65b30c8e3ede",
        "name":"Golf",
        "plate_country":"PL",
        "plate_number":"ZS11111"
    },
    "parking":{
        "id":1,
        "name":"Strefa A",
        "description":""
    },
    "price":"1.00"
}]
```

8. Profil zalogowanego użytkownika
   - Adres: /api/parkings/
   - Metoda: GET
   - Odpowiedź:

```
{
    "user":{
        "id":5,
        "username":"admin",
        "email":"admin@gmail.com"},
    "wallet":"27.00"
}
```

9. Potwierdzenie transakcji
   - Adres: /api/payments/
   - Metoda: POST
   - Dane żądania:

```
{
    "transaction_id":"PAY-48T58255YC665612DLBSPNFI"
}
```

5 Implementacja systemu

W tym rozdziale opisana została implementacja systemu. Na początku omówiono sposób działania aplikacji internetowej oraz mobilnej na przykładowych fragmentach kodu. Dalsza część zawiera wyniki działania systemu, w tym zrzuty ekranu aplikacji mobilnej. Na końcu znajdują się metody testowania systemu oraz środowiska programistyczne i edytory wykorzystane do realizacji zadań pracy.

5.1 Aplikacja internetowa

Główne zadania jakie były do zrealizowania podczas implementacji serwera, to: autoryzacja, zarządzanie i identyfikowanie pojazdów, kupno oraz kontrola biletu postojowego. Aby te funkcjonalności mogły być osiągalne dla aplikacji mobilnej, konieczne jest udostępnienie zasobów poprzez adresy URL. W Django razem z projektem tworzony jest plik urls.py, gdzie są one umieszczane w liście urlpatterns. Każda pozycja to wywołanie funkcji ulr(), która jako pierwszy argument przyjmuje wyrażenie regularne z adresem, a jako drugi widok, który obsługuje żądanie.

Na listingu 5.1 zaprezentowany jest fragment kodu z pliku urls.py, ze wszystkimi adresami na jakie swoje żądania wysyła aplikacja mobilna. W zależności od realizowanych zadań, przyjmują wybrane metody protokołu HTTP.

Listing 5.1: Mapowane adresy URL z pliku urls.py

```
1  urlpatterns = [
2      url(r'^login/', ParQAuthToken.as_view()),
3      url(r'^register/$', register_driver),
4      url(r'^vehicles/$', vehicle_list),
5      url(r'^vehicles/(?P<pk>[0-9]+)$', vehicle_detail),
6      url(r'^parkings/$', parking_list),
7      url(r'^tickets/$', ticket_list),
8      url(r'^current/$', current_user),
9      url(r'^payments/$', payment_list),
10 ]
```

Tworzenie konta i autoryzacja

Z procesem utworzenia konta w systemie przez kierowcę lub kontrolera związanych jest kilka dodatkowych czynności. Są to przypisanie roli oraz generowanie tokenu autoryzacyjnego. Wszyscy użytkownicy tworzeni są w oparciu o istniejący już w Django model użytkownika – User, z modułu django.contrib.auth.models. Niezbędny jest jednak sposób, który pozwoli na odróżnienie kierowcy od kontrolera, aby można było nadać im odmienne uprawnienia. Do tego celu wykorzystany został dodatek djroles, napisany specjalnie na potrzeby tego systemu. Swoje działanie opiera na istniejących w Django grupach, z którymi domyślnie użytkownicy są w relacji wiele-do-wielu. Dodatek ten tworzy dodatkową tabelę – Role, w której umieszczane są

wybrane grupy. Spośród nich, użytkownik będzie mógł należeć tylko do jednej, w tym samym czasie. Na listingu 5.2 pokazany został sposób, w jaki są deklarowane grupy, które będą używane do tworzenia ról. Jest to robione poprzez zadeklarowanie klasy Pythona, która dziedziczy po BaseRole - jej nazwa zostanie użyta do utworzenia grupy. Jako że ten język dopuszcza wielodziedziczenie, wykorzystany został model Driver oraz Officer, dzięki czemu grupa posiada tę samą nazwy co tabele z dodatkowymi informacjami kierowcy i kontrolera.

Listing 5.2: Fragment modelu Driver

```
from decimal import Decimal
from django.db import models
from django.contrib.auth.models import User
from djroles.models import Role
from djroles.roles import BaseRole

class Driver(models.Model, BaseRole):
    user = models.OneToOneField(
            User,
            on_delete=models.CASCADE,
            primary_key=True
            )
    wallet = models.DecimalField(
            _('wallet'),
            max_digits=8,
            decimal_places=2,
            default=Decimal()
            )
```

Poza przypisaniem grupy, każdy użytkownik w systemie, niezależnie już od pełnionej roli, musi mieć wygenerowany token autoryzacyjny. Obie te czynności realizowane są przez sygnały (ang. signals), dostępne w Django. Są to funkcje, które zostaną wykonane w odpowiedzi na jakieś zdarzenie związane z ustaloną klasą w projekcie. Na listingu 5.3 przedstawione zostały sygnały powiązane z domyślną klasą użytkownika User (tworzenie tokenu) oraz klasami Driver i Officer – przypisywanie do ról. Wykonywane są w odpowiedzi na zapisanie modelu w bazie danych, czyli sygnał post_save.

Listing 5.3: Sygnały związane z tworzeniem konta w systemie

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from djroles.models import Role
from django.contrib.auth.models import User
from rest_framework.authtoken.models import Token

from .models import Driver, Officer

def assign_to_role(role_class, user):
    role = Role.objects.get_for_class(role_class)
    role.give_role(user)

@receiver(post_save, sender=Driver)
def assign_driver_to_group(instance, created, **kwargs):
    if created:
```

```
16          assign_to_role(Driver, instance.user)
17
18  @receiver(post_save, sender=Officer)
19  def assign_officer_to_group(instance, created, **kwargs):
20      if created:
21          assign_to_role(Officer, instance.user)
22
23  @receiver(post_save, sender=User)
24  def create_auth_token(sender, instance, created, **kwargs):
25      if created:
26          Token.objects.create(user=instance)
```

Identyfikacja pojazdów

Z każdym pojazdem kierowcy w systemie powiązany jest identyfikator UUID (ang. Universally unique identifier), czyli 128-bitowa losowa wartość. Przechowywana jest ona w modelu Badge (listing 5.4), powiązanym relacją jeden-do-jeden z pojazdem (Vehicle). W modelu znajduje się także metoda generate_image(). W niej właśnie utworzony zostanie kod QR, w którym zakodowany będzie identyfikator. Do generowania QR w postaci pliku png, użyta została biblioteka Pythona qrcode. Oprócz danych, podawany jest także poziom korekcji błędów i wersja kodu.

Listing 5.4: Badge - model identyfikatora

```
1   class Badge(models.Model):
2       uuid = models.UUIDField(default=uuid.uuid4, editable=False)
3
4       @property
5       def is_assigned(self):
6           return hasattr(self, 'vehicle')
7
8       def path_to_file(self):
9           path = os.path.join(settings.BASE_DIR, 'badges', 'images')
10          return '{0}/{1}.png'.format(path, self.uuid)
11
12      def generate_image(self):
13          qr = qrcode.QRCode(
14              version=4,
15              error_correction=qrcode.constants.ERROR_CORRECT_H,
16              box_size=20,
17              border=4
18              )
19          qr.add_data(self.uuid)
20          path = os.path.join(settings.BASE_DIR, 'badges', 'images')
21          return qr.make_image().save(self.path_to_file())
```

Utworzony w ten sposób kod QR jest następnie wysyłany na e-maila, podanego przez kierowcę podczas rejestracji. Umieszczony w widocznym miejscu pojazdu, będzie używany przez kontrolera podczas sprawdzania biletu.

Taryfikator

Taryfikator oprócz powiązanych ze sobą opłat, musi zawierać także informację o czasie w którym obowiązuje, zarówno godzin jak i konkretnej daty w kalendarzu. Dobrym rozwiązaniem jest umożliwienie ustawienia także takiej daty jako cyklicznej, dzięki czemu dany taryfikator mógłby obowiązywać np.: co tydzień w sobotę. Taką możliwość daje klasa Event, z dodatku django-scheduler. Pozwala ona na stworzenie wydarzenia z datą początkową oraz końcową, która będzie przechowywana w bazie danych. Dodatkowo takie wydarzenie może być cykliczne, a podane daty wyznaczać będą wtedy dzień tygodnia, czy miesiąca. Model Schedule rozszerza Event, dzięki czemu taryfikator posiada zarówno opłaty jak i czas obowiązywania. Na listingu 5.5 przedstawiono przykład jego tworzenia.

Listing 5.5: Tworzenie cotygodniowego taryfikatora

```
lot = ScheduleLot.objects.create(name='Strefa A')
start = timezone.make_aware(datetime(2016, 12, 25, 8))
end = timezone.make_aware(datetime(2016, 12, 25, 17))
rule = Rule.objects.create(frequency='WEEKLY', name='weekly')
sch = Schedule.objects.create(start=start, end=end,
rule=rule, schedule_lot=lot)
cha = Charge.objects.create(cost=1, minutes=60, duration=60)
ScheduleCharge.objects.create(schedule=sch, charge=cha)
```

Kupno biletu

Oprócz pojazdu na jaki ma zostać zakupiony bilet, użytkownik podaje także parking, który razem z datą rozpoczęcia postoju wyznacza obowiązujący taryfikator. Podobnie jak w Strefie Płatnego Parkowania w Szczecinie, cena może się zmieniać w zależności od długości parkowania. Na listingach 5.6 i 5.7 przedstawione zostały algorytmy naliczające opłatę. W klasie Schedule najpierw wyznaczany jest efektywny czas postoju, w przypadku gdyby podany bilet wykraczał poza datę obowiązywania taryfikatora. Po przedstawieniu czasu trwania postoju w postaci minut, pobierane są wszystkie opłaty powiązane z danym taryfikatorem. Następnie w pętli, aż do wyczerpania ilości minut postoju, każda z opłat nalicza swoją część ceny (model Charge), zgodnie z jej czasem obowiązywania. W przypadku wyczerpania listy opłat przed zakończeniem postoju, ostatnia z nich (tak jak w SPP w Szczecinie) naliczy cenę dla pozostałych minut postoju. W pętli wszystkie opłaty cząstkowe są sumowane i ta właśnie suma zostanie naliczona jako opłata kierowcy.

Listing 5.6: Obliczanie łącznej kwoty w klasie Schedule

```
def calculate_price(self, ticket):
    effective_dates = self._get_effective_dates(ticket)
    time = self._to_minutes(effective_dates[1] - effective_dates[0])
    charges = list(self.charges.all().order_by('-schedulecharge__order'))

    if not charges:
        raise Exception('Schedule without charges')

```

```
9       price = Decimal()
10      while time > 0:
11          charge = charges.pop()
12          if len(charges) == 0 or time <= charge.duration:
13              price += charge.calculate_price(time)
14              break
15          else:
16              price += charge.calculate_price(charge.duration)
17              time -= charge.duration
18      return price
```

Listing 5.7: Obliczanie częstki ceny w pojedynczej opłacie - model Charge

```
1  def calculate_price(self, time):
2      price = Decimal()
3      price += Decimal(time // self.minutes) * self.cost
4      rest = (Decimal(time % self.minutes) / self.minutes) * self.cost
5      if not self.minute_billing and rest:
6          price += self.cost
7      else:
8          price += rest
9      return price
```

Doładowanie konta

Doładowywanie konta odbywa się dwustopniowo. Najpierw w aplikacji mobilnej użytkownik przelewa swoje pieniądze (np.: z wykorzystaniem karty płatniczej) na konto PayPal'a powiązane z systemem. Zwrócony identyfikator transakcji wysyłany jest metodą POST do serwera systemu ParQ, na adres payments/. W widoku przedstawionym na listingu 5.8, obsługującym ten adres, przedstawiony jest fragment kodu odpowiedzialny za to żądanie. W linii 12 wywoływana jest metoda, w której otrzymany identyfikator przesyłany zostaje do serwera PayPal'a, celem uzyskania informacji o kwocie jaka została przelana. Następnie, jeśli wszystko się zgadza, numer transakcji zapisywany jest w bazie danych (linia 13). Gdy transakcja o takim identyfikatorze już istnieje, zwrócony zostanie w odpowiedzi błąd. Tylko w razie powodzenia wykonana będzie następna linia, gdzie powiązana portmonetka użytkownika jest uzupełniana o wpłaconą kwotę.

Listing 5.8: payment_list - widok doładowania konta użytkownika

```
1  @api_view(['POST'])
2  @permission_classes((IsAuthenticated,))
3  def payment_list(request):
4      serializer = TransactionSerializer(data=request.data)
5      if serializer.is_valid():
6          try:
7              driver = Driver.objects.get(pk=request.user.id)
8          except Driver.DoesNotExist:
9              return Response('User is not driver',
10                             status=status.HTTP_403_FORBIDDEN)
11
12         money = get_payment_money(request.data['transaction_id'])
13         serializer.save(user=request.user)
```

```
14          driver.add_money(Decimal(money))
15          driver.save()
16          driver_ser = DriverSerializer(driver)
17          return Response(driver_ser.data, status=status.HTTP_201_CREATED)
18      return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Do realizacji tej części pracy wykorzystano kilka dodatków do Django oraz jeden dodatkowy pakiet Pythona, a są to:

- Django REST Framework – na jego podstawie tworzone są widoki, które obsługują żądania w architekturze REST. Ten dodatek umożliwia także autoryzacje żądań, opartą na generowanym wcześniej tokenie.
- django-scheduler – tworzenie wydarzeń, które mogą się powtarzać cyklicznie.
- django-ordered-model – numerowane relacje w tabelach.
- django-countries – państwa i ich kody ze standardu ISO 3166.
- djroles – tworzenie ról w Django.
- qrcode – pakiet Pythona umożliwiający generowanie kodów QR.

5.2 Aplikacje mobilne

W ramach pracy zostały wykonane dwie oddzielne aplikacje dla kierowcy oraz kontrolera, ze względu na odmienny sposób w jaki korzystają oni z systemu. Kierowca oprócz logowania, powinien móc m.in. założyć konto, czy dokonać płatności przy doładowywaniu konta. Kontroler w swojej aplikacji powinien mieć możliwość przeprowadzenia kontroli biletów postojowych, co odbywa się poprzez odczytanie plakietki z kodem QR. Do każdej z tych aplikacji zalogować może się jedynie użytkownik posiadający konto, do którego została przypisana odpowiednia rola. Obie wymagają ciągłej komunikacji z serwerem.

Parsowanie danych

Format JSON składa się z par danych - z każdą wartością związany jest klucz, który służy do jej identyfikacji. Przechowywana wartość może być ciągiem znaków, liczą całkowitą i zmiennopozycyjną, ale także obiektem zawierającym dalszy zestaw par (otoczonym nawiasami klamrowymi) lub tablicą takich obiektów (nawiasy klamrowe). JSON przesyłany jest jako tekst w części danych żądania lub odpowiedzi HTTP. W celu wydobycia informacji musi być poddany odpowiedniej analizie, zarówno na serwerze jak i aplikacji mobilnej. W Androidzie parsowanie można wykonać za pomocą klas JSONObject oraz JSONArray z pakietu org.json

JSONObject umożliwia parsowanie pojedynczego jsona, a dane do analizy podawane są jako ciąg znaków (String) w konstruktorze. Metody instancji tej klasy służące do wydobywania wartości, jako parametr przyjmują nazwę klucza z jakim ta wartość jest związana. Są to m.in: get(), getInt(), czy getString(), a ich użycie zależy od spodziewanego typu wartości przechowywanego w dokumencie. Istnieje także metoda getJSONObject(), która zwraca

zagnieżdżony obiekt jsona w postaci instancji klasy JSONObject. JSONArray instancjonowana jest w podobny sposób, a używana jest w przypadku gdy json zawiera tablicę danych. Po wywołaniu metody getJSONObject() z indeksem elementu w parametrze, zwracany jest obiekt pojedynczego jsona, czyli klasy JSONObject. Na listingu 5.9 znajduje się przykład z wykorzystaniem kolekcji danych.

Listing 5.9: Parsowanie jsona dla kolekcji pojazdów kierowcy

```
1  JSONArray array = new JSONArray(response);
2
3  LinkedList<Vehicle> vehicleList = new LinkedList<>();
4
5  for(int i = 0; i < array.length(); i++) {
6      JSONObject json = array.getJSONObject(i);
7      Vehicle vehicle = new Vehicle();
8
9      vehicle.setId(json.getInt("id"));
10     vehicle.setBadge(json.getString("badge"));
11     vehicle.setName(json.getString("name"));
12     vehicle.setPlateCountry(json.getString("plate_country"));
13     vehicle.setPlateNumber(json.getString("plate_number"));
14
15     vehicleList.add(vehicle);
16 }
```

Ten fragment kodu zostaje wykonany, w momencie otrzymania z serwera pojazdów powiązanych z danym kierowcą. Dla każdego elementu tablicy json wydobywany jest obiekt klasy JSONObject, z którego pobierane są dane pojazdu. Zostaną one później zaprezentowane użytkownikowi w odpowiednim widoku.

Komunikacja z serwerem

Komunikacja w systemie polega na wysyłaniu żądań przez aplikacje do serwera i oczekiwaniu na rezultat, który zostanie przesłany w odpowiedzi. Tym zajmuje się biblioteka HTTP – Volley. Stanowi alternatywę dla wykorzystywanych wcześniej klas Javy, jak HttpURLConnection, będąc rozwiązaniem dedykowanym dla Androida, cechującym się prostotą i szybkością działania. Świetnie nadaje się do prostych API, w których wymiana informacji polega na przesyłaniu list oraz pojedynczych danych w formacie json. Jedną z jej głównych zalet jest zdolność do buforowania odpowiedzi. Jeśli zapytanie może zostać obsłużone dzięki danych znajdującym się w pamięci podręcznej, nie będzie ono musiało zostać ponownie wysłane.

Listing 5.10 przedstawia sposób, w jaki konstruowane są zapytania w tej bibliotece. Najpierw w konstruktorze podawany jest rodzaj metody HTTP oraz adres, na jaki żądanie będzie miało zostać wysłane. Dwa następne parametry to obiekty klas anonimowych. Jeśli odpowiedz z serwera będzie zwrócona z kodem 2xx, to wykonana zostanie metoda onResponse() pierwszego obiektu. Parametr response zawiera dane odpowiedzi i to właśnie on będzie poddawaniu parsowaniu. Jeśli odpowiedz jest błędna, czyli wysłana została z kodem 4xx lub 5xx, wykonana

zostanie onErrorResponse. Tak utworzone zapytanie dodawane jest do kolejki zapytań, gdzie kolejność wysłania może być zależna od ustawionego priorytetu.

Listing 5.10: Wysłanie żądania

```
StringRequest ticketRequest = new StringRequest(
        Request.Method.GET,
        url.getTicketByBadgeURL(badge),
        new Response.Listener<String>() {
            @Override
            public void onResponse(String response) {
                // w response znajduje się ciało odpowiedzi
            }
        },
        new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                // w przypadku błędu. Kody 4xx i 5xx
                error.printStackTrace();
            }
        }
) { };

Volley.newRequestQueue(scanActivity).add(ticketRequest);
```

Realizacja płatności

Do integracji PayPal'a z aplikacją mobilną została wykorzystana biblioteka PayPal Android SDK, która jest dostępna w ramach open source. Razem z nią, oprócz możliwości realizacji opłat, udostępniane są także gotowe ekrany, gdzie użytkownik może podać swoje dane uwierzytelniające. Biblioteka ta pozwala na realizowanie płatności pojedynczych (tzw. Single Payment, użytkownik za każdym razem musi podawać dane uwierzytelniające) oraz automatycznych (Future Payment, dane podawane tylko raz, a zwrócony token OAuth pozwala na dokonywanie płatności w imieniu użytkownika). W tym systemie oferowana jest tylko pierwsza opcja.

Na listingu 5.11 przedstawiony został fragment, w którym za pomocą intencji uruchomiona zostaje aktywność uwierzytelnienia płatności. W klasie PayPalPayment podawane zostają informacje odnośnie wysokości płatności, waluty oraz typu transakcji. Obiekt tej klasy, razem z informacjami konfiguracyjnymi, zostaje umieszczony w intencji. Wywołanie metody aktywności Androida startActivityForResult() spowoduje wyświetlenie nowego ekranu.

Listing 5.11: Intencja rozpoczynająca aktywność PayPal'a

```
private void getPayment() {
    paymentAmount = amountText.getText().toString();

    PayPalPayment payment = new PayPalPayment(new BigDecimal(
            String.valueOf(paymentAmount)),
            "PLN", "Ticket Fee", PayPalPayment.PAYMENT_INTENT_SALE);

    Intent intent = new Intent(this, PaymentActivity.class);
```

```
9       intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
10      intent.putExtra(PaymentActivity.EXTRA_PAYMENT, payment);
11
12      startActivityForResult(intent, PAYPAL_REQUEST_CODE);
13  }
```

Metoda startAcitvityForResult() uruchamiająca nową aktywność różni się od startActivity() tym, że od docelowej aktywności oczekiwane jest otrzymanie jakiegoś wyniku. Po zakończeniu utworzonego ekranu uruchomiona zastanie metoda onActivityResult(). Do niej właśnie przesłana zostanie odpowiedź z serwera PayPal'a o statusie przeprowadzonej płatności, a także w razie sukcesu, identyfikator płatności. W tym miejscu właśnie będzie on wysłany do serwera systemu, celem jego dalszego uwierzytelnienia.

Kontrola biletu

Kontrola biletu możliwa jest w aplikacji przeznaczonej dla kontrolera. Odbywa się poprzez skanowanie obrazu z kamery wbudowanej w urządzenie mobilne. Do tego celu użyta została biblioteka ZXing ("Zebra Crossing"), która służy do przetwarzania obrazu. Oprócz QR, umożliwia także skanowanie kodów kreskowych. Domyślnie nie są dołączone do niej żadne gotowe aktywności, jak w przypadku PayPal Android SDK. Istnieje jednak dodatek, ZXing Android Embedded, którego sposób działania jest zbliżony do przeprowadzania transakcji. Zawiera on gotową aktywność, w której zostaje uruchomiona kamera urządzenia mobilnego. Na listingu 5.12 w klasie IntentIntegrator konfigurowany jest najpierw skaner. Podawany jest tam rodzaj kodów graficznych, czy orientacja ekranu. Po natrafieniu przez skaner na kod, skanowanie zostaje zakończone. Informacja jaką udało się odkodować, zwracana jest w metodzie onActivityResult().

Listing 5.12: Intencja rozpoczynająca aktywność skanowania

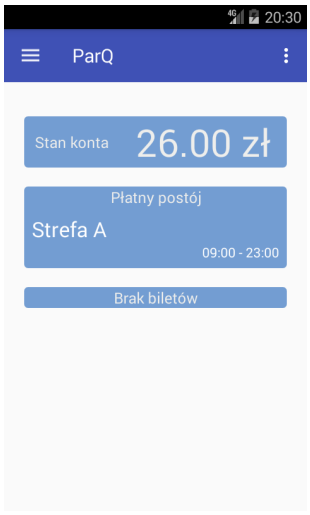
```
1  public void startScanOnClick(View view) {
2      IntentIntegrator integrator = new IntentIntegrator(this)
3              .setDesiredBarcodeFormats(IntentIntegrator.QR_CODE_TYPES)
4              .setPrompt("Scan")
5              .setOrientationLocked(true);
6      integrator.initiateScan();
7  }
```

Po zeskanowaniu kodu, znajdująca się w nim informacja trafia do serwera. Tam sprawdzane jest najpierw, czy jest to poprawny identyfikator UUID. Wykorzystane do tego zostało odpowiednie wyrażenie regularne. Następnie, jeśli format jest poprawny, wykonywane są zapytania do bazy danych, mające na celu znalezienie tego kodu, powiązanego z nim pojazdu oraz biletu, który jest ważny. Jeśli wszystkie te operacje zakończą się sukcesem, kontroler otrzymuje informację o ważnym bilecie postojowym. W przypadku gdyby któryś z kroków zakończył się niepowodzeniem, konieczne będzie wystawienie mandatu.

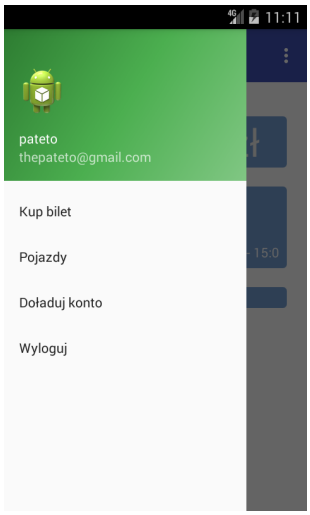
5.3 Wyniki działania systemu

Kierowca korzystający z systemu ParQ, oprócz logowania i rejestracji, może także doładować konto, dodawać nowe pojazdy oraz kupować bilety postojowe w przeznaczonej dla niego aplikacji. Kontroler natomiast ma możliwość przeprowadzenia kontroli zaparkowanego pojazdu z wykorzystaniem wbudowanego w telefon aparatu. Poniżej znajduje się opis funkcjonującego systemu, z zrzutami ekranów stworzonych na jego potrzeby aplikacji mobilnych oraz wynikami działania serwera.

W pierwszej kolejności opisana została aplikacja przeznaczona dla kierowcy, korzystającego ze strefy płatnego parkowania. Po zalogowaniu do niej, wyświetlany jest ekran główny, znajdujący się na rys. 5.1. To tutaj użytkownik uzyskuje podstawowe dane powiązane ze swoim kontem, takie jak ilość pieniędzy znajdujących się w portmonetce, czy godziny płatnego postoju w dniu sprawdzania. Tutaj też są wyświetlane informacje o wszystkich aktywnych biletach postojowych, wraz z informacjami o pojeździe na który zostały zakupione i godziną zakończenia. Z ekranu głównego możliwe jest przejście do pozostałych ekranów aplikacji. Po naciśnięciu ikonki w lewym górnym rogu, wysuwa się menu boczne przedstawione na rys. 5.2. Na górze wyświetlana jest nazwa oraz e-mail zalogowanego użytkownika. Poniżej znajdują się opcje przenoszące do ekranów zakupu biletu, zarządzania pojazdami lub doładowywania konta.



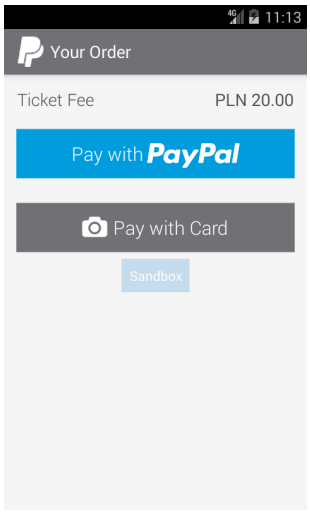
Rys. 5.1: Informacje o koncie



Rys. 5.2: Menu boczne aplikacji

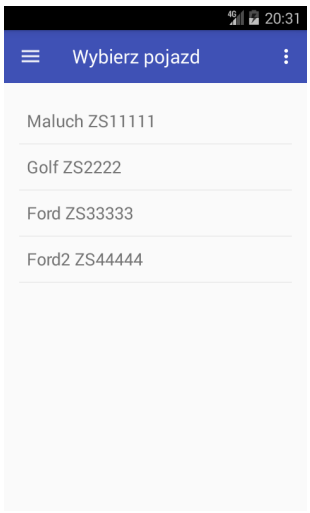
Doładowywanie konta (rys. 5.3 i 5.4) rozpoczyna się od podania kwoty, jaką konto użytkownika w systemie ma zostać doładowane. Zatwierdzenie jej, powoduje pokazanie ekranów pochodzących z wykorzystywanej do realizacji płatności biblioteki – PayPal Android SDK. To w nich możliwe jest wybranie metody płatności, uwierzytelnienie oraz zatwierdzenie transakcji. Po zakończeniu, użytkownik przenoszony jest na ekran główny z zaktualizowanym stanem konta.

Rys. 5.3: Pole z kwotą doładowania

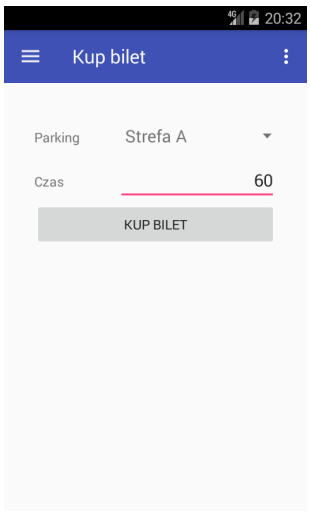


Rys. 5.4: Realizacja płatności w PayPal

Do zakupu biletu w systemie niezbędna jest informacja o pojeździe oraz parkingu, w którym będzie odbywał się postój. Te czynności wykonywane są na dwóch ekranach zaprezentowanych poniżej. Wybranie samochodu na rys. 5.5 spowoduje wyświetlenie ekranu z rys. 5.6, gdzie należy wskazać parking oraz czas postoju w minutach. Gdy użytkownik posiada odpowiednią ilość pieniędzy, nastąpi zakupienie biletu, czego potwierdzenie znajdzie się na ekranie głównym aplikacji.

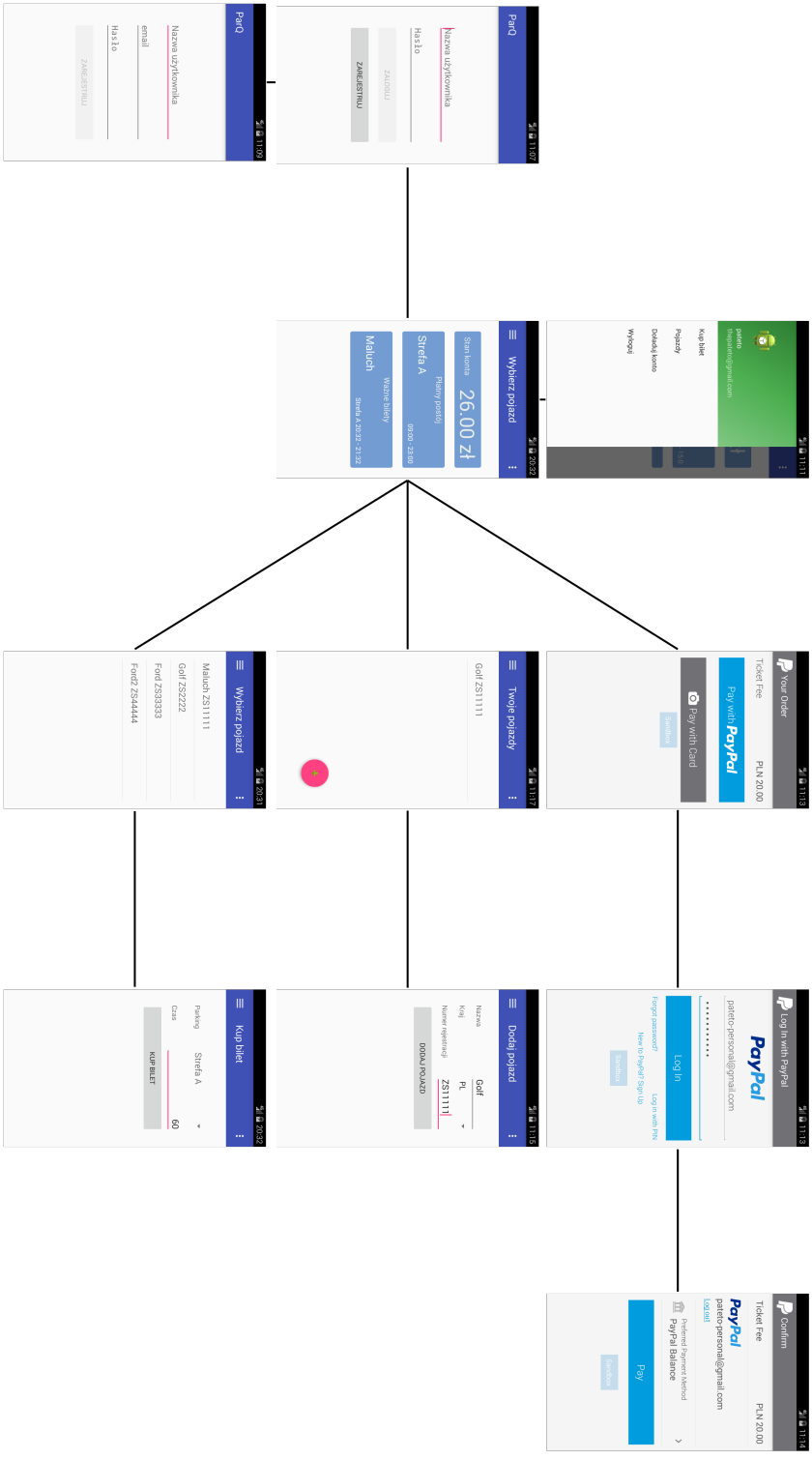


Rys. 5.5: Zakup biletu - wybór pojazdu



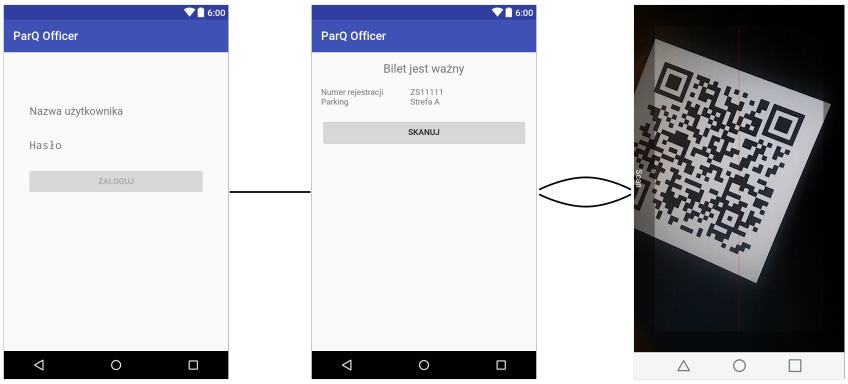
Rys. 5.6: Zakup biletu - parking i czas

Na rys. 5.7 znajduje się schemat, na którym przedstawione zostały wszystkie ekrany oraz możliwe przejścia między nimi w aplikacji przeznaczonej dla kierowców.



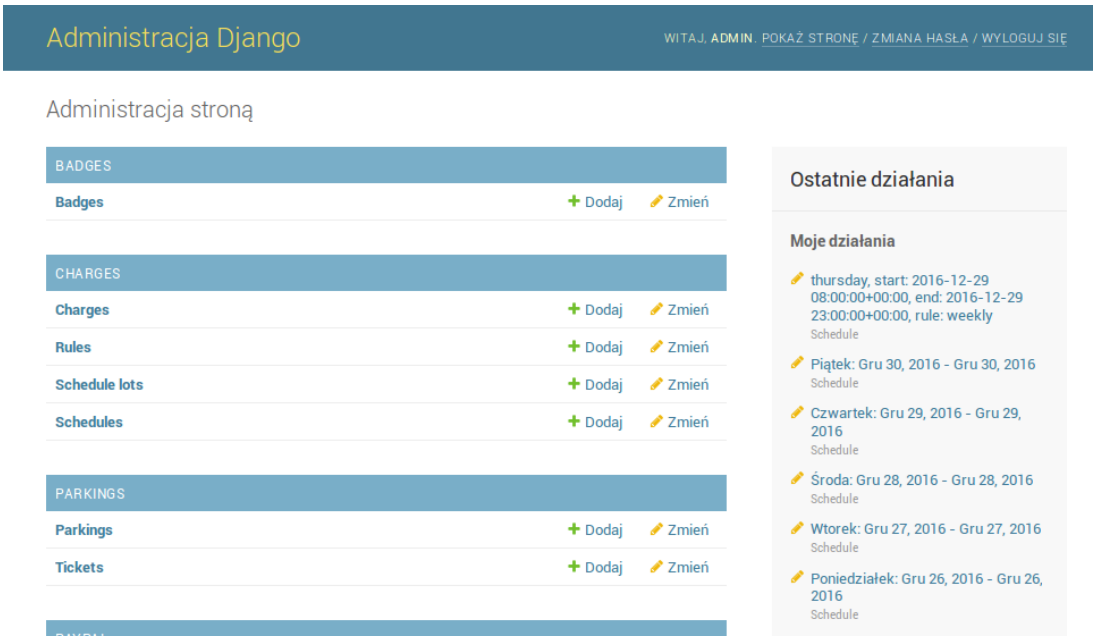
Rys. 5.7: Ekrany w aplikacji kierowcy

Druga z aplikacji umożliwiająca przeprowadzenie kontroli, została zbudowana z trzech ekranów. Po zalogowaniu, użytkownikowi prezentowany jest widok, na którym będą wyświetlane informacje o kontrolowanym pojeździe. Włączenie aparatu telefonu i rozpoczęcie przeprowadzania kontroli, zaczyna się w momencie dotknięcia przycisku “Skanuj”. W chwili napotkania dowolnego kodu QR, skanowanie zostaje zakończone, a dane znajdujące się w kodzie są wysyłane do serwera systemu. Zwrócona odpowiedź jest prezentowana na ekranie, który był widoczny zaraz po zalogowaniu. To tutaj kontroler zostanie poinformowany o tym, czy kod jest poprawny i czy jest z nim powiązany obowiązujący bilet. W celu dalszej weryfikacji, prezentowana jest także informacja o numerze rejestracyjnym, aby można było sprawdzić czy plakietka z kodem QR jest powiązana z zaparkowanym pojazdem. Na rys. 5.8 znajdują się ekrany aplikacji kontrolera.

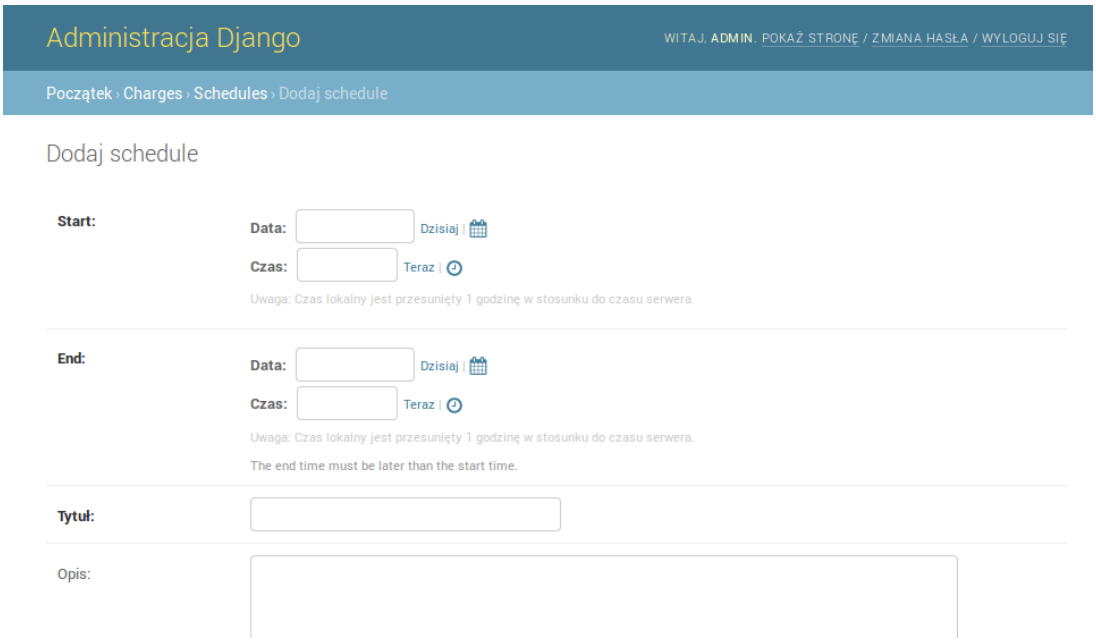


Rys. 5.8: Ekrany w aplikacji kontrolera

Cała funkcjonalność systemu dostępna z poziomu aplikacji mobilnych oparta jest na wymianie danych z serwerem. To właśnie udostępnianie odpowiedniego API jest głównym zadaniem aplikacji internetowej. W ten sposób tworzone są konta użytkowników, dodawane nowe pojazdy, kupowane oraz sprawdzane bilety. Oprócz tego system umożliwia także zarządzanie parkingiem, w tym dodawanie nowych grafików, taryfikatorów, czy tworzenie kont kontrolerów. To realizowane jest za pomocą strony administracyjnej, do której dostęp będą mieć pracownicy administracyjni. Strona ta została wygenerowana automatycznie przez framework Django, na bazie utworzonych modeli. Każdy z nich może być z jej poziomu tworzony, aktualizowany czy usuwany. Na rys. 5.9 została przedstawiona strona główna. Rys. 5.10 przedstawia jedną z podstron, w której tworzony jest nowy grafik.



Rys. 5.9: Główny panel administratora



Rys. 5.10: Dodawanie nowego grafiku

5.4 Testy

Do weryfikowania poprawności tworzonego oprogramowania napisane zostały zestawy testów. Testy jednostkowe sprawdzają odpowiednie funkcjonowanie pojedynczych elementów systemu, takich jak metody klas. Poprawność interakcji zachodzących między modułami sprawdzana jest za pomocą testów integracyjnych. W aplikacji internetowej napisanej w Django, używana do tego celu jest klasa TestCase z pakietu django.test. Każdy z modułów tej aplikacji zawiera plik tests.py, w którym umieszczane są testy. Każdy z nich zawiera przynajmniej jedną asercję, która sprawdza poprawność otrzymanych danych i decyduje o sukcesie lub porażce przeprowadzonego testu. Podobnie sytuacja wygląda w Androidzie, gdzie używana jest biblioteka JUnit. Dodatkowo aplikacja mobilna była testowana zarówno na emulatorze, jak i prawdziwym urządzeniu.

Akceptowanie transakcji przychodzących od klientów systemu wymaga posiadania specjalnego konta sprzedawcy. Dzięki PayPal Sandbox możliwe jest stworzenie środowiska testowego dla aplikacji, w którym mogą być utworzone fikcyjne konta klientów oraz sprzedawców. W ten sposób przeprowadzany jest cały proces płatności, który z perspektywy serwera i aplikacji mobilnej, niczym nie różni się od prawdziwych transakcji.

5.5 Środowiska programistyczne i edytory

Część mobilna systemu została wykonana w Android Studio, będącym dedykowanym środowiskiem programistycznym dla systemu Android. Zostało stworzone przez Google i bazuje na IntelliJ. Jest to rozbudowane IDE, które oprócz tworzenia gotowego szablonu aplikacji, posiada wszystkie funkcje spotykane w nowoczesnych środowiskach, takie jak refaktoryzacja kodu, podpowiedzi, czy poprawianie składni. Razem z nim instalowany jest także emulator, dzięki czemu możliwe jest testowanie aplikacji bez potrzeby posiadania prawdziwego urządzenia z systemem Android.

Do pisania aplikacji serwerowej wykorzystywany był, dostępny z poziomu wiersza poleceń, edytor Vim. Jego standardowa funkcjonalność została rozszerzona o zewnętrzne dodatki, dzięki stworzonemu przez społeczność systemowi zarządzania dodatkami – Vundle. Dodatek YouCompleteMe wprowadził okna z podpowiedziami i uzupełniania do wpisywanego kodu. Drugi zainstalowanym dodatkiem był NERD Tree, dzięki któremu możliwe stało się wyświetlanie struktury plików katalogu, w którym został uruchomiony edytor.

Podsumowanie

Słownik pojęć

E-commerce – (e-handel, ang. handel elektroniczny)
Płatności elektroniczne
Dostawca usług płatniczych
System płatniczy
Instrument płatniczy
Bankowość elektroniczna
Bankowość internetowa
Bankowość wirtualna

Literatura

[1] Jerzy Andrzejewski. *Wszystko o płatnościach elektronicznych*. http://www.komputerswiat.pl/centrum-wiedzy-konsumenta/uslugi-online/wszystko-o-platnosciach-elektronicznych/podstawowe-formy-platnosci-w-sieci.aspx. (dostęp: 15.12.2016).

[2] Janina Banasikowska. *Rodzaje płatności i systemy płatności na rynku elektronicznym*. http://www.swo.ae.katowice.pl/_pdf/127.pdf. (dostęp: 6.12.2016).

[3] Artur Borcuch. *Pieniądz elektroniczny pieniądz przyszłości – analiza ekonomiczno-prawna*. CeDeWu, 2007.

[4] Bartłomiej Chinowski. *Elektroniczne metody płatności. Istota, rozwój, prognoza*. CEDUR, 2013.

[5] Django REST framework. *Dokumentacja Django REST framework*.

[6] Django Software Foundation. *Dokumentacja Django*.

[7] Maciej Dudko (red.). *Biblia e-biznesu: Nowy Testament*. Helion, 2016.

[8] Maciej Dutko (red.). *Biblia e-biznesu*. Helion, 2013.

[9] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. *Head First Design Patterns*. Helion, 2005.

[10] Nigel George (red.). *The Django Book*. http://djangobook.com, 2016.

[11] Marcin Gigiel. *Wartość biletów parkingowych można sumować?* http://wszczecinie.pl/aktualnosci,wartosc_biletow_parkingowych_mozna_sumowac_nasz_czytelnik_zaskarzyl_decyzje_gminy_szczecin_dotyczaca_spp,id-27585.html. (dostęp: 22.01.2017).

[12] Pete Goodliffe. *Jak stać się lepszym programistą*. Helion, 2015.

[13] Dawn Griffiths, David Griffiths. *Rusz głową! Android*. Helion, 2016.

[14] Cay S. Horstmann. *JAVA Podstawy. Wydanie IX*. Helion, 2013.

[15] iso.org. *ISO/IEC 18004:2015*. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=62021. (dostęp: 17.12.2016).

[16] Karol Kunat. *W 2017 roku smartfony i tablety będą odpowiedzialne za 75% ruchu w internecie*. https://www.tabletowo.pl/2016/10/31/w-2017-roku-smartfony-i-tablety-beda-odpowiedzialne-za-75-ruchu-w-internecie/. (dostęp: 21.11.2016).

[17] Mark Lutz. *Python. Wprowadzenie. Wydanie IV*. Helion, 2011.

[18] Monika Mikowska. *POLSKA.JEST.MOBI 2015*. Jestem, 2015.

[19] paypal.com. *Dokumentacja PayPala*. https://developer.paypal.com. (dostęp: 16.01.2017).

[20] Michał Pisarski. *Android - historia najpopularniejszego mobilnego systemu operacyjnego*. http://www.komputerswiat.pl/artykuly/redakcyjne/2014/06/android-historia-najpopularniejszego-mobilnego-systemu-operacyjnego.aspx.

(dostęp: 19.12.2016).

[21] polskieradio.pl. *Barometr e-commerce 2016*. http://www.polskieradio.pl/42/273/Artykul/1571779,Barometr-ecommerce-2016.
(dostęp: 21.11.2016).

[22] Patrycja Sass-Staniszewska, Mateusz Gordon. *E-commerce w Polsce 2014. Gemius dla e-Commerce Polska*. Gemius Polska, 2014.

[23] Rada Miasta Szczecin. *OBWIESZCZENIE NR 13/14 z dnia 26 maja 2014 r.* http://bip.um.szczecin.pl/UMSzczecinFiles/file/Obwieszczenie_13(2).pdf.
(dostęp: 22.01.2017).

[24] thonky.com. *QR Code Tutorial*. http://www.thonky.com/qr-code-tutorial/.
(dostęp: 17.12.2016).