Wydział
Informatyki

Piotr Zieliński

Nr albumu: 29979

Kierunek studiów: Informatyka

Specjalność: Techniki programowania

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, 2017

OŚWIADCZENIE
AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa inżynierska pn.

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

napisana pod kierunkiem:

dr inż. Edwarda Półrolniczaka

jest w całości moim samodzielnym autorskim opracowaniem sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych.

Złożona w dziekanacie

Wydziału Informatyki

treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej i graficznej.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

.....
podpis dyplomanta

Szczecin, dn. .....

ABSTRACT

Spis treści

Wstęp

W największych polskich metropoliach powoli codziennością staje się korzystanie z systemów informatycznych, które swoim działaniem wspierają miejską infrastrukturę. Ich zastosowanie obejmuje najczęściej transport publiczny, czy dotyczące tej pracy, strefy płatnego parkowania, w których 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 nie byłoby możliwe, gdyby nie znaczący rozwój płatności elektronicznych. Dotyczą one realizowania transakcji finansowych przy użyciu elektronicznych instrumentów płatniczych. Przyczyniły się do powstania nowych form prowadzenia biznesu, takich jak e-commerce (ang. handel elektroniczny), który wykorzystuje różne kanały elektroniczne (internet, faks, czy telewizję), do zawarcia transakcji handlowej. W ostatnich latach stał się jedną z ważniejszych gałęzi gospodarki [21]. Dynamiczny rozwój e-płatności wzajemnie napędzany był przez postęp technologiczny, a główny udział w tym miały zyskujące coraz większą popularność urządzenia mobilne. Szeroki zakres oferowanych w nich funkcjonalności dotyczy także płatności (tzw. m-płatności), a same smartfony bywają już zaliczane do instrumentów płatniczych. Najczęściej są wykorzystywane 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. 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 wybór tematu.

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

Systemowi informatycznemu, który powstał w ramach tej pracy, została nadana nazwa ParQ. Najważniejsze zadania jakie należało zrealizować, dotyczyły implementacji części serwerowej oraz dwóch aplikacji mobilnych – dla kontrolerów oraz kierowców. Interakcja użytkownika z systemem odbywa się jedynie za pośrednictwem urządzenia mobilnego, w którym ma on możliwość utworzenia konta, dodania pojazdu, czy zakupu biletu. Z każdym pojazdem powiązany jest specjalny numer identyfikacyjny, który następnie zostaje przedstawiony w postaci kodu QR. Do przeprowadzenia kontroli wystarczy zeskanowanie go za pomocą wbudowanego w telefon aparatu.

Cała praca została podzielona na pięć rozdziałów. W rozdziale 1. został poruszony temat płat-

ności elektronicznych. Celem tej części jest wprowadzenie czytelnika do omawianego tematu. 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. Duża część rozdziału została poświęcona na zaprezentowanie różnych form płatności internetowych, razem z przedstawieniem ich wad oraz zalet. W końcowej części znajduje się opis zasad działania Strefy Płatnego Parkowania w Szczecinie, według których został stworzony system ParQ. Przedstawiono m.in. sposób w jaki naliczane są opłaty za postój. Omówione zostały także podobne systemy wspierające korzystanie ze stref parkowania, wykorzystujące aplikacje mobilne.

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. W tej części 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, dla którego wykonana została część mobilna. Opisany tu został także framework Django, wykorzystany do implementacji aplikacji internetowej. Na końcu przedstawiony został PayPal, czyli usługodawca płatności online.

W kolejnym 3. rozdziale zawarta została dokumentacja techniczna. Na początku przedstawiony został szczegółowy sposób działania systemu, a następnie wymieniono jego wymagania funkcjonalne i niefunkcjonalne. Kolejny podrozdział zawiera diagramy UML, w tym diagramy przypadków użycia, pakietów, klas, aktywności i sekwencji. Ostatnia część opisuje API (ang. Application Programming Interface), za pomocą którego aplikacja mobilna komunikuje się z serwerem, w tym zostały przedstawione także przykładowe zapytania oraz odpowiedzi.

Ostatni rozdział opisuje szczegóły implementacji systemu, z podziałem na część mobilną oraz serwerową. Zaprezentowano w nim fragmenty kodu, realizujące wybrane funkcjonalności. Dalsza część zawiera wyniki działania systemu oraz sposoby testowania, środowiska programistyczne i edytory tekstowe użyte do pisania kodu.

1 Płatności elektroniczne i strefa płatnego parkowania

Poniższy rozdział w większości został poświęcony płatnościom elektronicznym. W pierwszej części przedstawiona została historia oraz poszczególne etapy ich rozwoju. Dalej dokonano analizy różnych metod płatności, ze względu na takie kryteria jak wielkość pojedynczej transakcji. Na końcu opisano zasady funkcjonowania szczecińskiej Strefy Płatnego Parkowania, razem z akceptowanymi w niej formami płatności.

1.1 Wprowadzenie

Mimo, że e-commerce odnosi się ogólnie do stosowania urządzeń elektronicznych w zakupie oraz sprzedaży, to utożsamiany jest głównie z internetem. Będąc medium łączącym niejako wszystkie poprzednio używane (faks, telefon, telewizja), sieć pełni dominującą rolę w e-handlu. 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 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, które są 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 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, ale także na dziedziny związane 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, które są oferowane przez bank w placówce. Użytkownik może kontrolować stan konta, zaciągać kredyty lub wykonywać przelewy. Taka usługa jest niezależna od miejsca, dostępna 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

Podczas rozpoczętej w latach dziewięćdziesiątych pierwszej generacja płatności próbowano 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 podczas użytkowania. Cechą wspólną pierwszej generacji jest dość problematyczna obsługa oraz wymaganie dodatkowego oprogramowania, bądź nawet specjalistycznych urządzeń. 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. Szczególnie postęp w obszarze zabezpieczeń, wiążący się z powstaniem szyfrowanych protokołów przesyłania danych (np.: HTTPS), jest znaczący dla systemów 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. Zmniejszyła się ilość kroków jakie trzeba wykonać, aby dokonać zakupu.

1.3 Analiza metod płatności

Duża różnorodność wśród metod płatności pozwala na dostosowanie ich do modelu biznesowego. Ważnym kryterium przy wyborze płatności jest wielkość pojedynczej transakcji w systemie. Wiąże się ona bezpośrednio z poziomem zabezpieczeń jaki należy zapewnić. Na decyzję powinny także wpływać indywidualne preferencje użytkowników, gdyż szczególnie istotny jest poziom zaufania, z jakim spotyka się dane rozwiązanie. Duża część użytkowników internetu przyzwyczajona jest do tradycyjnych płatności, zwłaszcza 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. Można wyróżnić:

- 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 mogły być zrealizowane. Najczęściej będą się odbywały 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. Ten rodzaj płatności może dotyczyć np.: zakupu sprzętu RTV lub AGD.

Moment pobrania

W przeciwieństwie do tradycyjnych metod płatności, te elektroniczne są zdecydowanie bardziej elastyczne. Jednym z czynników który na to wpływa jest 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 odbywa się na różnych etapach. Niektóre serwisy mogą na przykład wymagać zakupu wirtualnej waluty,

obowiązującej tylko w ich ramach. Taka sytuacja często spotykana jest na przykład 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. Wyróżnia się trzy typy, a są to:

- 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, w których użytkownicy trzymają elektroniczne pieniądze. Najpierw muszą zostać zasilone przez właściciela odpowiednią kwotą, aby później mogły być wykorzystywane do różnych transakcji finansowych. Są opcją szczególnie korzystną w przypadku częstych opłat, ponieważ nie wiążą się z nimi żadne prowizje pobierane od dokonanej transakcji. 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.

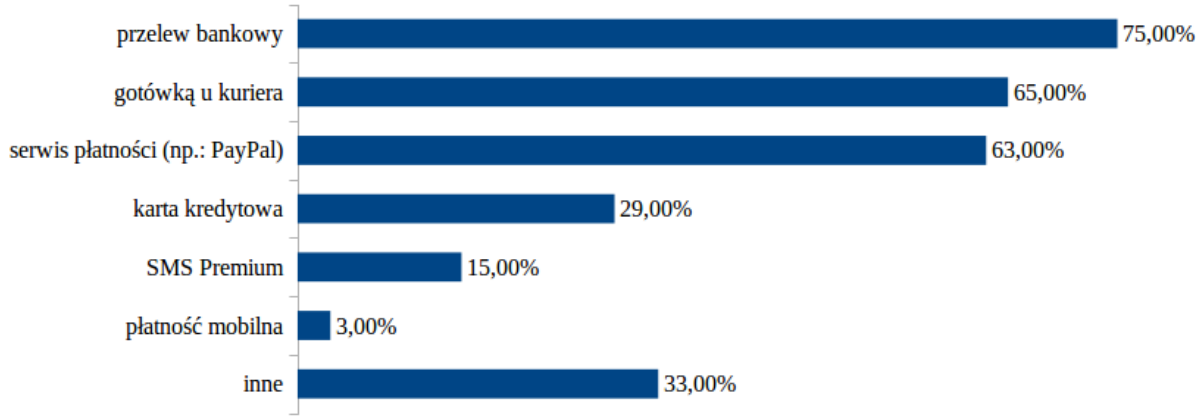
Wybór odpowiednich metod realizacji transakcji udostępnianych dla klientów w serwisie, powinien być poprzedzony dokładną analizą grupy docelowych użytkowników. Niektórzy mogą nie mieć dużego zaufania do tego typu rozwiązań i będą woleli zapłacić gotówką. Próba stworzenia serwisu przyjmującego większość 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 za pomocą 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 (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].

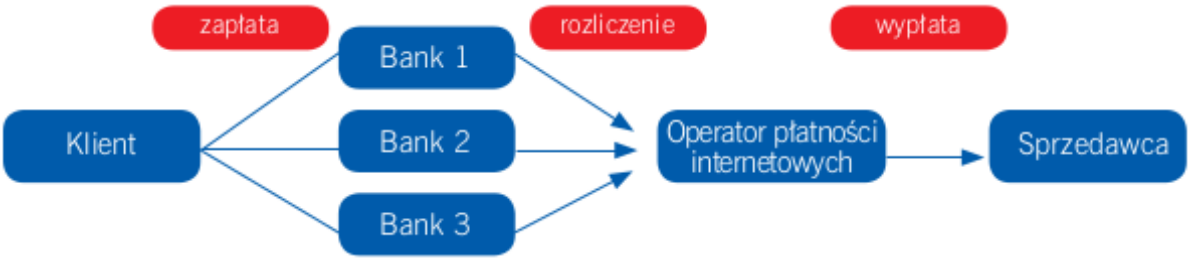
1.4 Bramki płatności

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 wdrażania będzie 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 implementować 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. 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.

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.



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

1.5 Opłaty w strefie płatnego parkowania

Poniżej 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.

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 1.1. 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.

Tab. 1.1: 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].

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 zo-

stała 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. Tutaj użytkownik ma do wyboru kilka metod płatności, w tym SMS Premium, czy płatność kartą. Po tym kroku do płatności będą wykorzystywane pieniądze elektroniczne, zgromadzone w portmonetce. Proces ten będzie wyglądał podobnie w tworzonym w ramach tej pracy systemie.

2 Opis wykorzystanych technologii

W tym rozdziale znajdują się informacje dotyczące najważniejszych technologii, które wykorzystano podczas realizacji zadań pracy. W kolejnych podrozdziałach opisane zostały m.in. kody graficzne QR, system mobilny Android oraz Django – framework do tworzenia aplikacji mobilnych. Na końcu znajduje się opis PayPala, czyli usługi wykorzystywanej do realizacji płatności w systemie.

2.1 Kody graficzne QR

Z przedstawianiem danych w postaci kodów graficznych, można się spotkać na przykład w marketach, gdzie towary oznaczane są za pomocą jednowymiarowego kodu kreskowego. Kombinacja jasnych oraz ciemnych linii służy do kodowania informacji, 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, do których zaliczy jest kod QR.

Kody QR (ang. Quick Respone - szybka odpowiedź) to dwuwymiarowe, kwadratowe kody graficzne. Składają się z modułów, czyli ciemnych oraz jasnych kwadratów, które są nośnikami danych. Zostały stworzone przez japońską firmę Denso-Wave w 1994 r [24] i według postanowień licencyjnych, mogą być wykorzystywane bez żadnych opłat. Standard został 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 jednowymiarowe kody kreskowe. Ponadto, zapewniają zdecydowanie lepszą korekcję błędów, ponieważ nawet częściowo uszkodzony kod może zostać poprawnie odczytany. W samym kodzie oprócz danych umieszczane są także miejsca szczególne, ułatwiające orientację podczas odkodowywania. Ich liczba zależy od rozmiaru kodu.

Rys. 2.1: Tytuł pracy przedstawiony w postaci kodu QR.

Na początku kody QR były głównie wykorzystywane w logistyce, przechowując na przykład informacje o adresie docelowym przesyłki. Współcześnie kojarzone są przeważnie ze smartfonami i można je spotkać niemal wszędzie. Służą do komunikacji z użytkownikami urządzeń mobilnych, przełamując niejako barierę między światem wirtualnym, a rzeczywistym. Mimo,

że kod zawiera informacje jedynie w postaci liczb, liter i symboli, to odpowiednie formatowanie informacji pozwala na dodatkowe ich interpretowanie przez urządzenie przenośne. I tak po zeskanowaniu może zostać wysłana wiadomość e-mail, dodany numer do kontaktów, czy wyświetlona strona w przeglądarce internetowej.

Sposoby kodowania

Kodowanie informacji odbywa się w zależności od trzech parametrów, a są to: wersja, typ danych oraz poziom korekcji błędów. Liczba modułów w kodzie ustalana jest przez numer wersji, która jest numerowana od 1 do 40. Determinuje ona ilość informacji, jakie mogą się w kodzie znaleźć. Wersja pierwsza posiada 441 modułów (21 na bok), a czterdziesta składa się z 31329 modułów (177 na bok). Każda kolejna jest większa od poprzedniej o trzy moduły. Wyższy numer wersji przekłada się na większą pojemność kodu, jednak wpływ na to mają także inne parametry. Dla wersji czterdziestej, w zależności od ustawień pozostałych parametrów, liczba znaków mieści się w przedziale od 7089 do 784 znaków.

Typ ustalany jest w zależności od rodzaju danych, jakie mają się znaleźć w kodzie. W przypadku gdy będą to tylko liczby, wystarczający będzie typ numeryczny, jeśli natomiast informacja zbudowana jest także z liter, to użyty może być na przykład typ alfanumeryczny. Wybór wpływa na ilość bitów (czyli modułów), które będą przechowywać informację o pojedynczym znaku, co przekłada się na maksymalną pojemność. Kod QR pozwala na przechowywanie czterech różnych typów danych, a są to:

- 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.

Korekcja 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 dzięki nim 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 złożonym. Po analizie danych określającej 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 kodu. Dopiero po tych kilku etapach, może zostać wygenerowany kod. Dla wszystkich najpopularniejszych języków programowania istnieją odpowiednie do tego zadania biblioteki. Na przykład w Pythonie taką biblioteką jest qrcode.

Do odczytywania kodu mogą zostać wykorzystane urządzenia mobilne. Często razem z nimi dostarczane są specjalne aplikacje, które to umożliwiają. Podobnie jak w przypadku kodowania, do odczytywania także można wykorzystać gotowe rozwiązania, w postaci bibliotek. Na urządzenia z systemem Android jest to np.: bilioteka ZXing (“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. Wiąże się to bezpośrednio z operacjami, jakie na takich adresach mogą być wykonane. Dodawanie pojedynczego elementu, np.: biletu, będzie wykonywane jako żądanie POST na kolekcji danych. Jednak to są jedynie zalecenia, do których programista nie jest zobowiązany się stosować. Wykorzystanie ich ułatwia jednak interakcję z takim serwisem.

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].

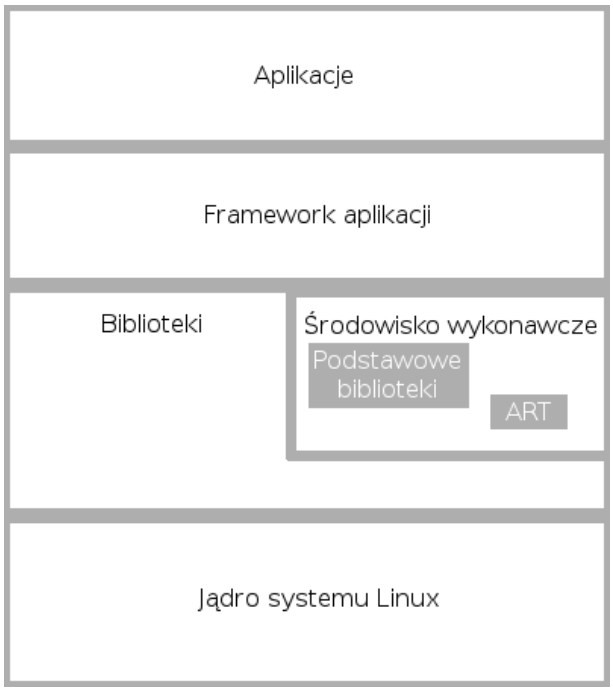
O Androidzie

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. Takie podejście pozwala na przenośność programów, czyli niezależność od platformy, kosztem pewnego spadku wydajności.

Mimo, że programy na Androida pisane są w Javie, nie jest wykorzystywany w nim JVM. 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, usunięcie powtarzających się symboli oraz zamiana znajdujących się tam instrukcji, na odpowiednie dla Androida. Dzięki temu skompilowany program będzie mniejszy oraz z założenia powinien działać szybciej. Ostatnim etapem budowania aplikacji jest stworzenie pliku ze skompilowaną aplikacją oraz jej zasobami.

Rys. 2.3: Proces budowy aplikacji.

Do wersji 4.4 Androida (KitKat) aplikacje uruchamiane były w wirtualnej maszynie Dalvik. Sposób działania był dość zbliżony do JVM, opierając się na interpretacji kodu bajtowego. Jedną z różnic był sposób działania wirtualnego procesora, który w Dalviku opartu został na rejestrach, a nie na stosie. Takie podejście wpływało na mniejsze zużycie pamięci, jednak programy były większe, gdyż instrukcje musiały zawierać dodatkowe informacje co do rejestrów, z których korzystały.

W wersji 5.0 Androida (Lolipop) zastąpiono dotychczasową maszynę wirtualną Dalvik, środowiskiem uruchomieniowym Android runtime (ART). Podobnie jak poprzednik 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 obowiązkową kompilację.

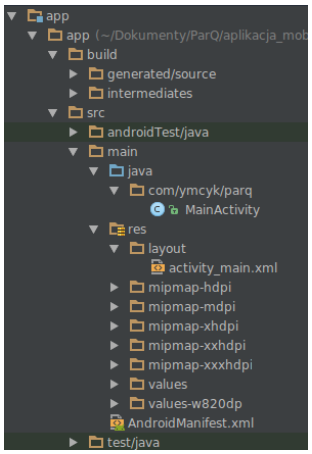
Programowanie aplikacji

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

- Java z JDK i JRE,
- Android SDK.

Wykorzystując Javę, aplikacja do funkcji systemowych odwoływać się będzie za pomocą udostępnionego przez system API. 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 także 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. Metoda onCreate znajdująca się w aktywności, określa jaki układ ma zostać użyty do zbudowania graficznego interfejsu. W odpowiedzi na interakcję z jakimś widokiem, mp.: przyciskiem, może zostać wywołana metoda znajdująca się w powiązanej z układem aktywności.

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].

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.

Początki Django

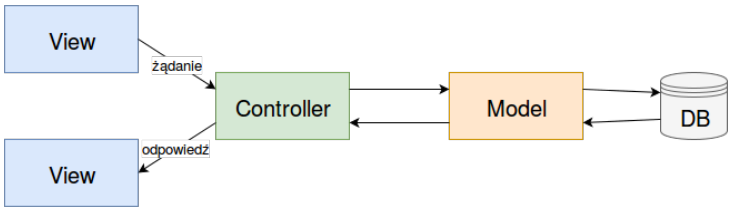
Django rozwijany jest jako wolne oprogramowanie na GitHub'ie, w ramach fundacji Django Software Foundation, ale skupia wokół siebie także wielu niezależnych twórców. Został napisany w Pythonie, stając się z czasem najpopularniejszym frameworkiem dla tego języka. Jego historia rozpoczęła się w 2003 r., 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 dla 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), co wpływa na zwiększoną czytelność.

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.

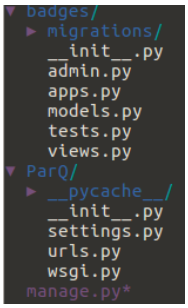
Listing 2.1: Przykładowa konfiguracja URL w Django.

```
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 moduł z konfiguracją projektu.



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 2.2: 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 2.1.

Listing 2.3: 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.

2.5 PayPal

Do realizowania płatności w systemie wykorzystywany jest system PayPal. Jego sposób działania jest nico odmienny od pozostałych bramek płatności jakie zostały wymienione w tej pracy, 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 (np.: aplikacji mobilnej), który będzie miał prawo obsługiwać płatności w imieniu posiadacza konta PayPal. Dzieje się to poprzez utworzenie tzw. aplikacji (ang. application) w systemie, 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 2.4 pokazany został format odpowiedzi z takim tokenem. Obowiązuje on przez określony czas, a po jego wygaśnięciu należy ponowić żądanie.

Listing 2.4: 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
}
```

Token będzie następnie umieszczany w nagłówku każdego zapytania, które wymaga autoryzacji, jak na przykład żądanie utworzenia nowej płatności. W tworzonym systemie wykorzystywa-

ne będą jednorazowe opłaty natychmiastowe, gdzie każda transakcja wymaga uwierzytelnienia przez użytkownika. Przykład takiego żądania zaprezentowany został na listingu 2.5. W nim podawana jest metoda płatności jaką klient chce zapłacić, oraz kwota jaka ma zostać przelana na konto sprzedającego. Dodatkowo, w bardziej rozbudowanych systemach, możliwe jest także wysłanie informacji o produkcie jaki jest kupowany.

Listing 2.5: 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.

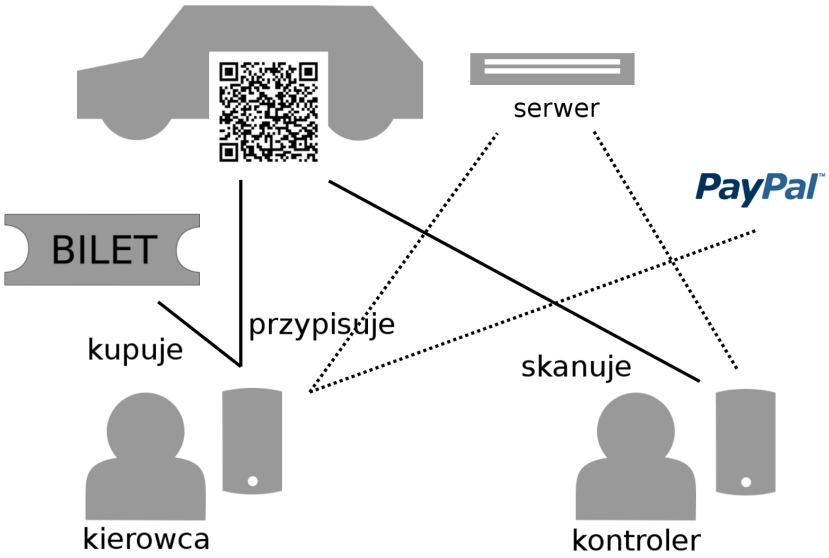
3 Projekt systemu

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.

3.1 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 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, który znajduje się na plakietkach w postaci kodu 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. 3.1: 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 będzie unikalny identyfikator płatności, który zostaje wysłany przez aplikację 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.

3.2 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.

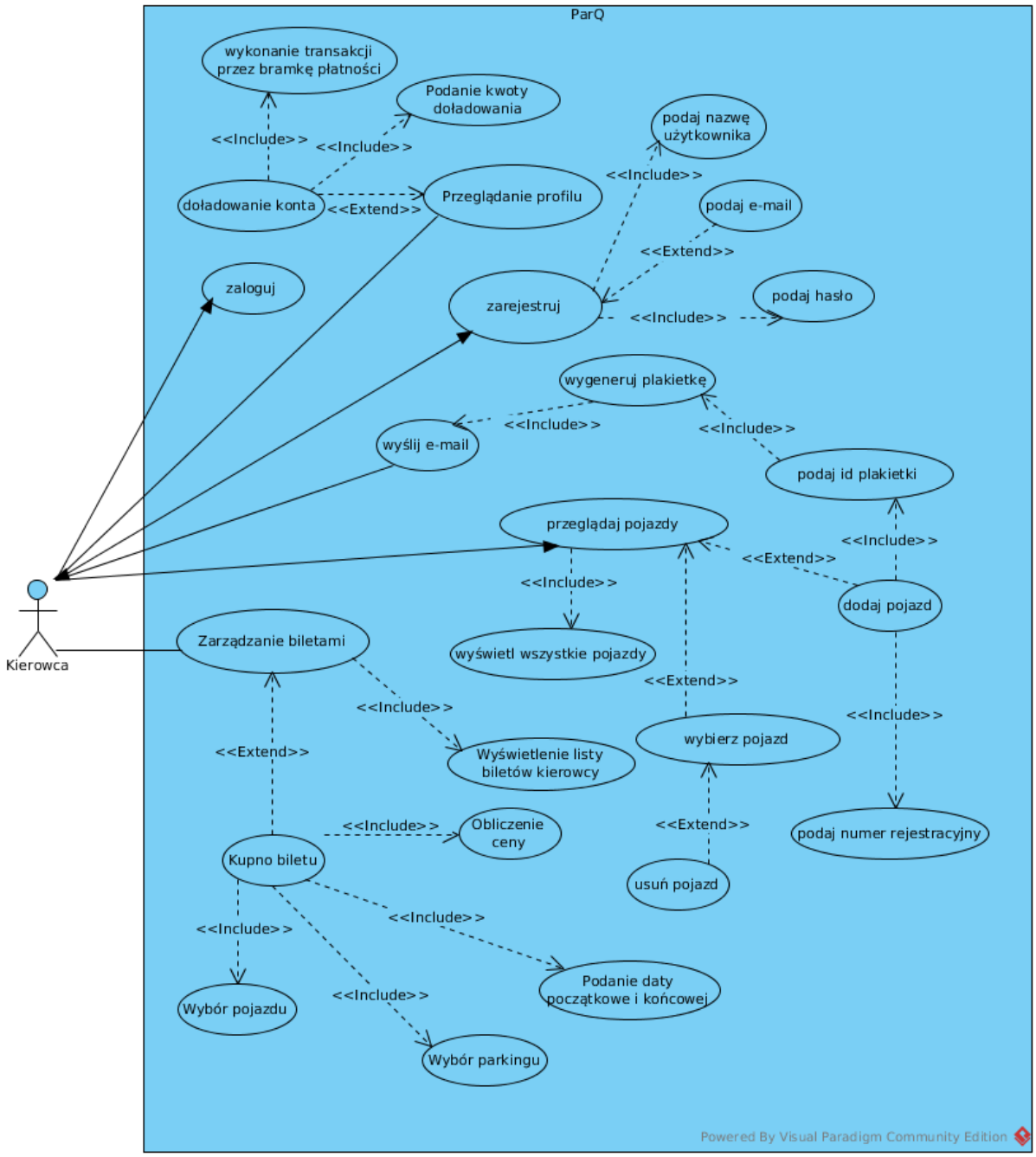
- Kontroler po zeskanowaniu otrzymuje informacje o numerze rejestracyjnym pojazdu, celem sprawdzenia, czy dany identyfikator jest przypisany do kontrolowanego pojazdu.
- Kontroler otrzymuje informację o ważności postoju – czy bilet został zakupiony i czy jest ważny oraz na jaki parking został zakupiony.

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.

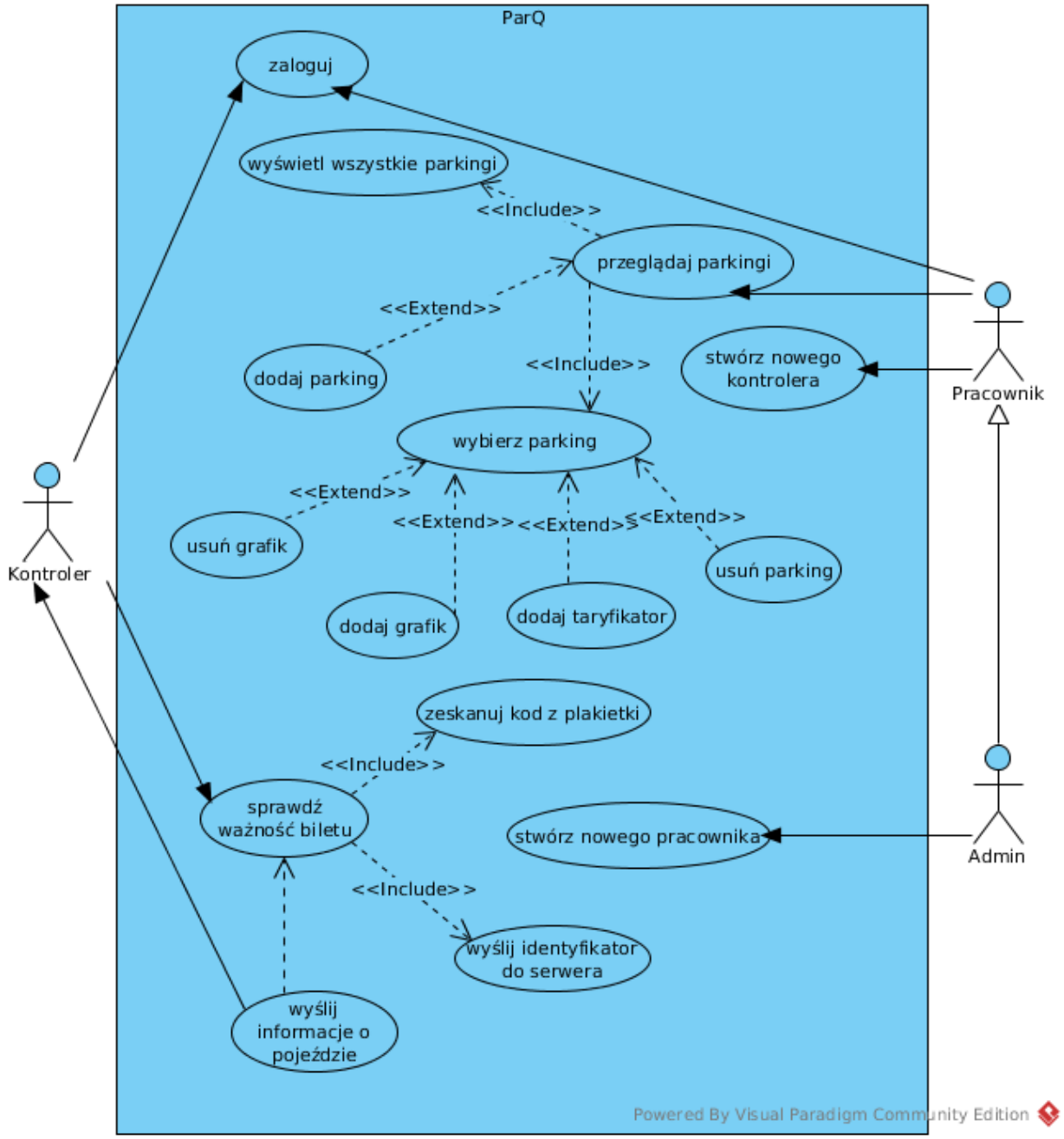
Wymagania niefunkcjonalne

- 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.

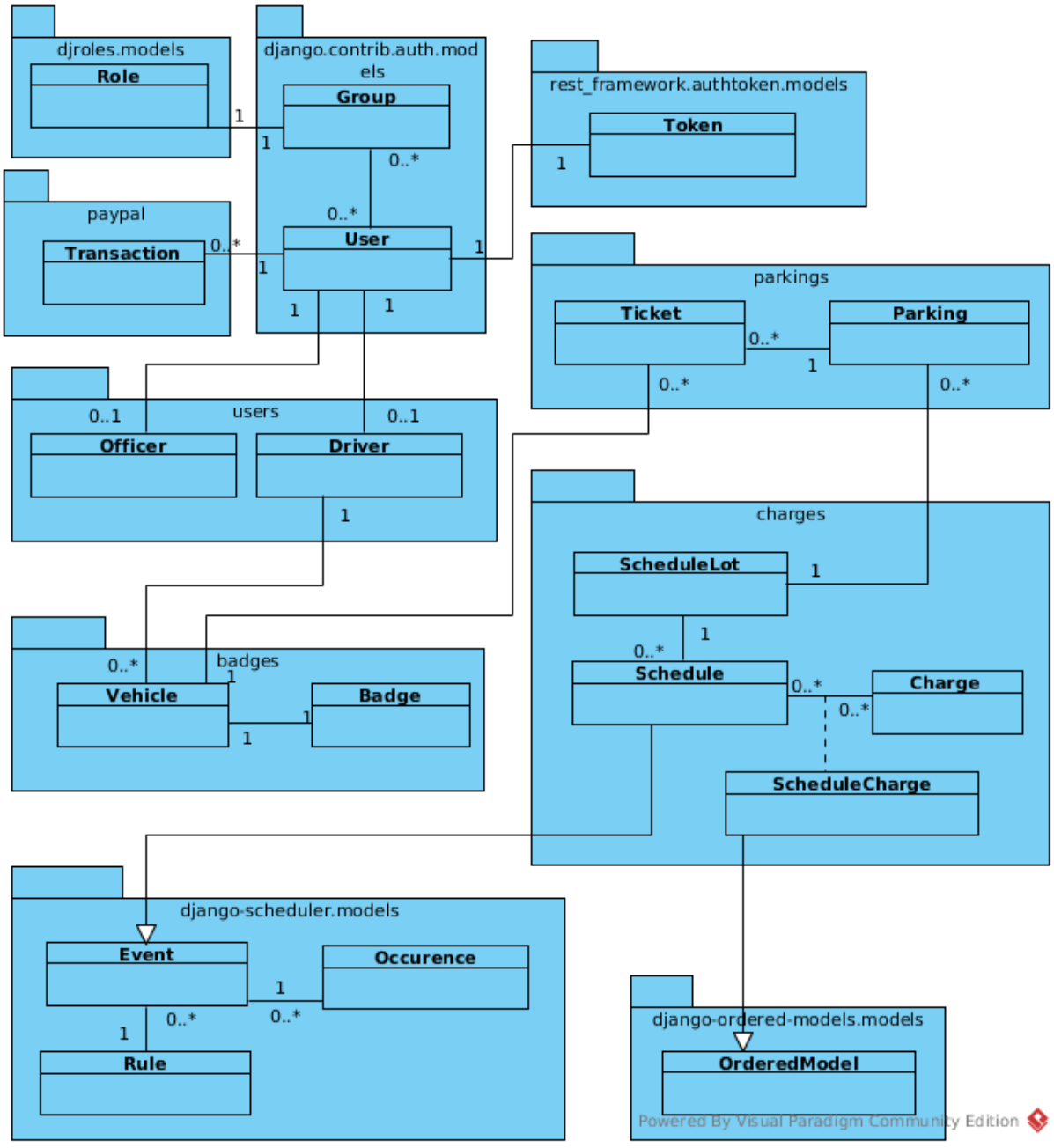
3.3 Diagramy UML



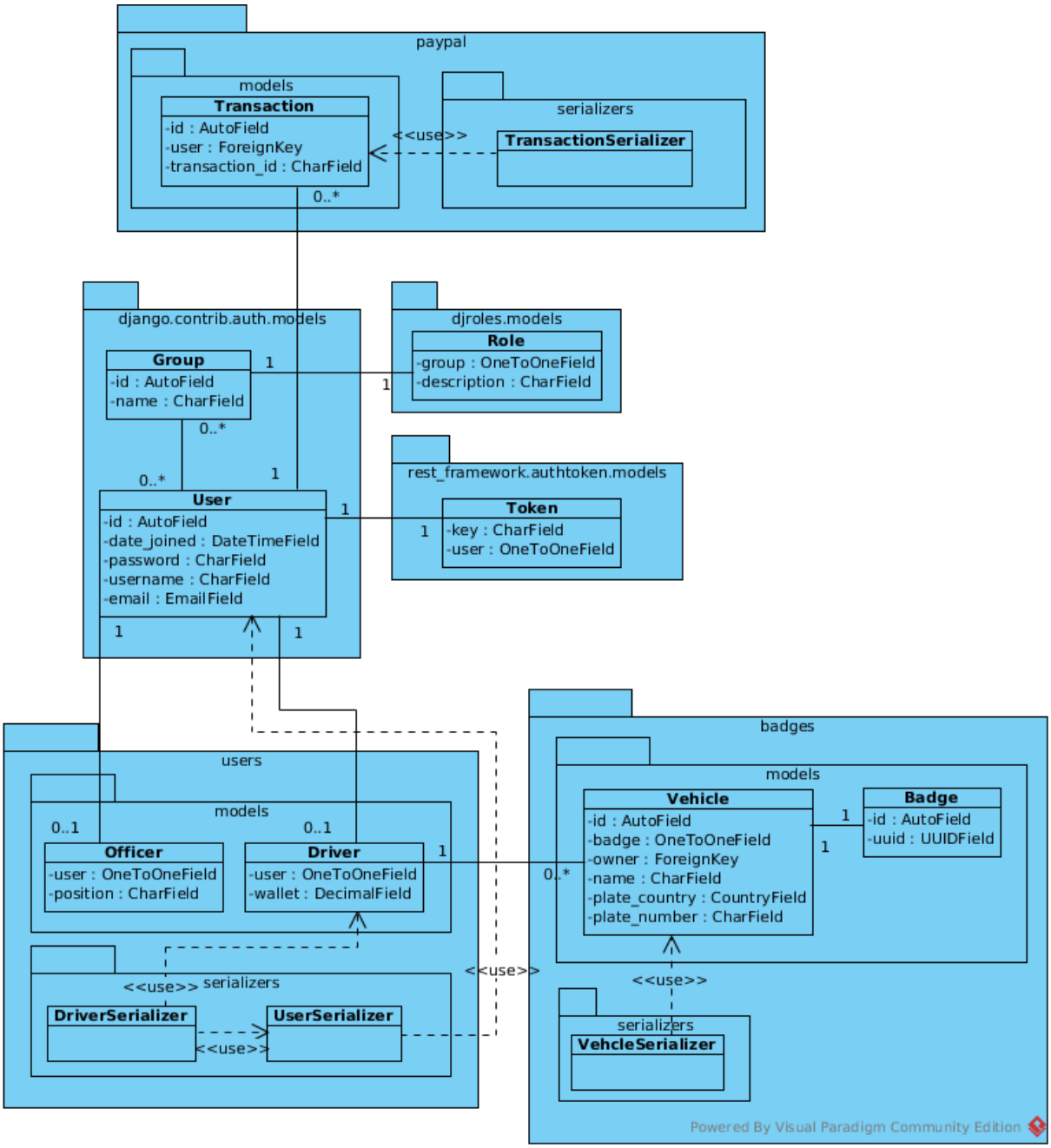
Rys. 3.2: Przypadki użycia dla kierowcy.



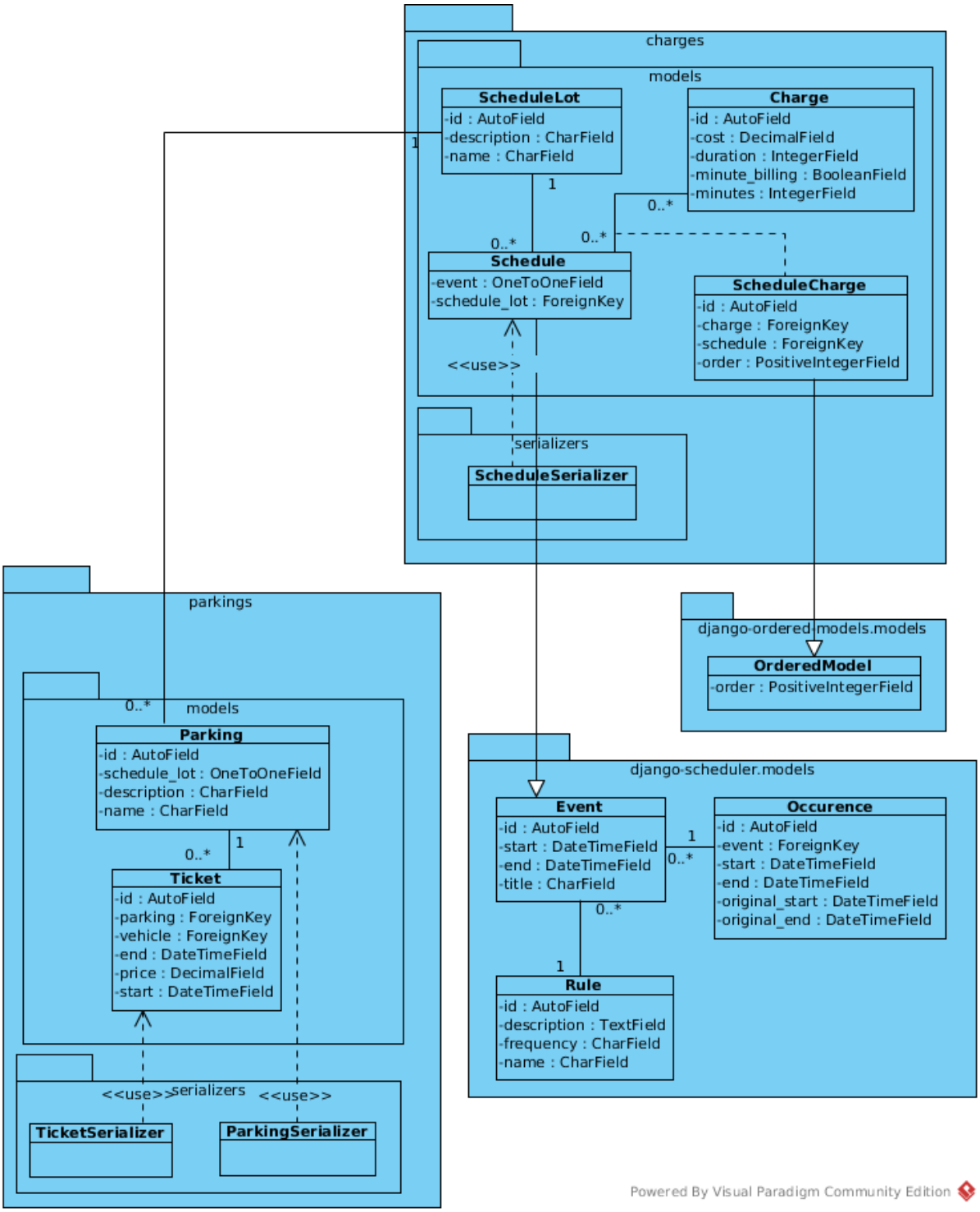
Rys. 3.3: Przypadki użycia dla kontrolera, pracownika i administratora.



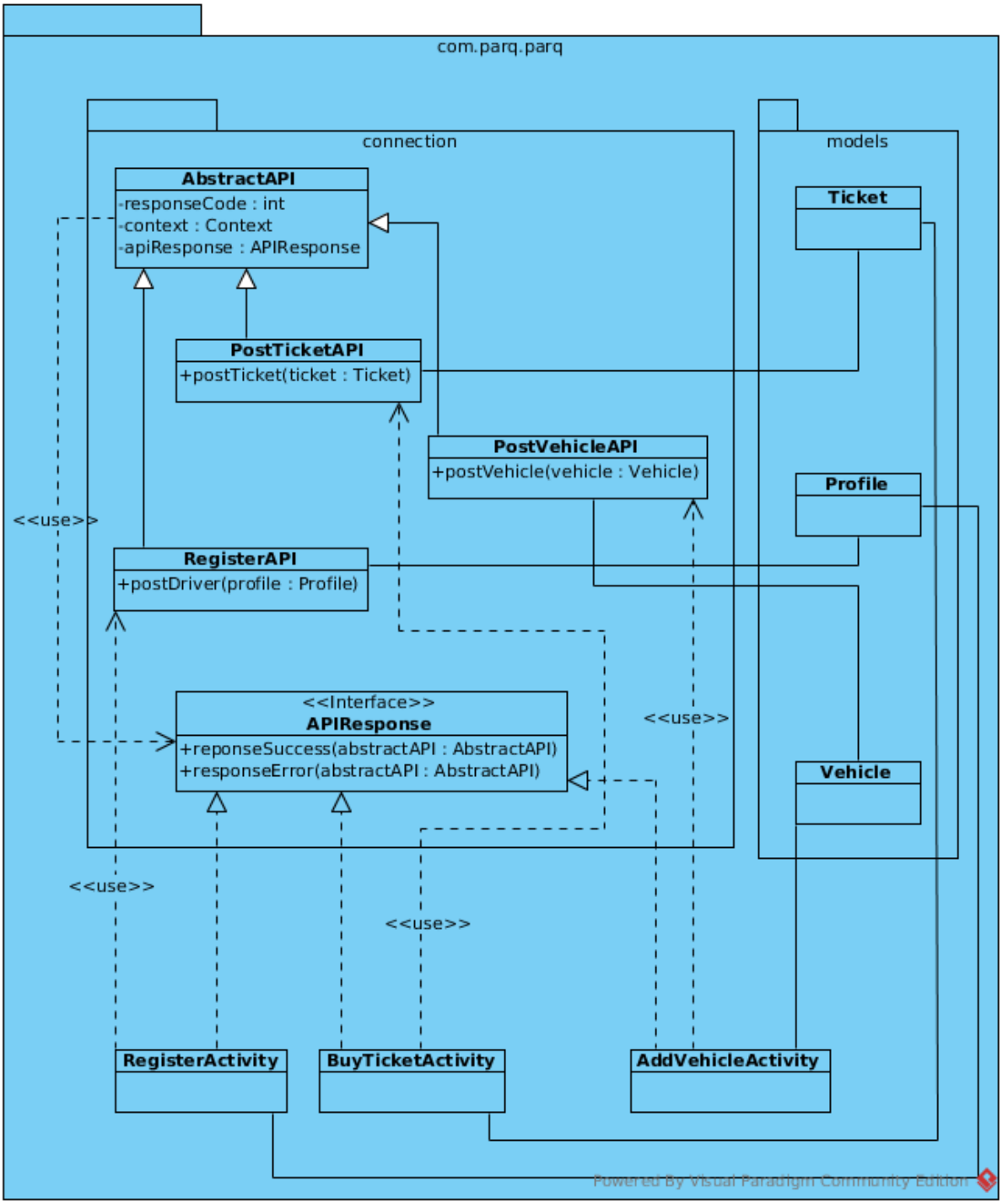
Rys. 3.4: Diagram pakietów.



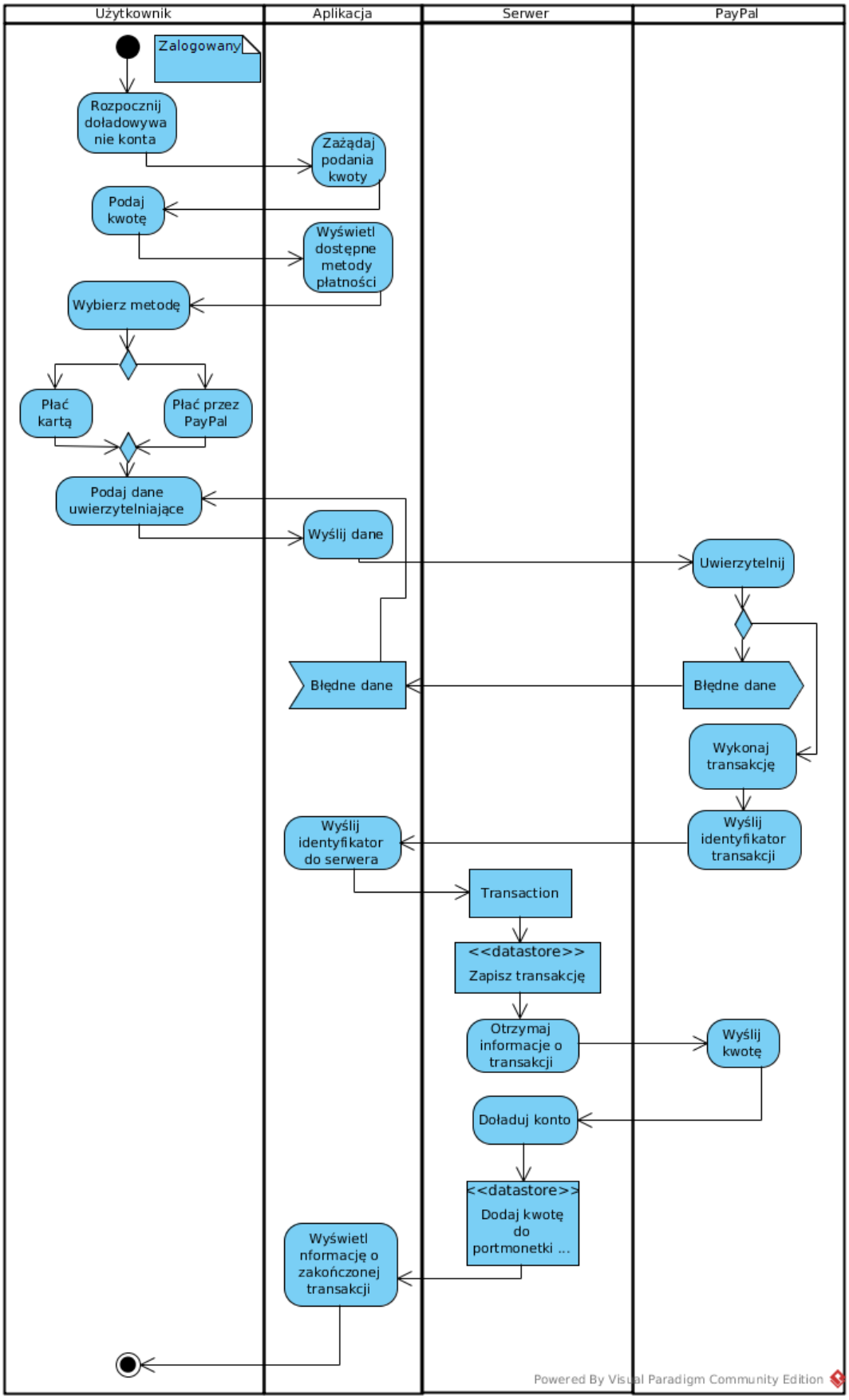
Rys. 3.5: Diagram klas dla pakietów paypal, users, badges oraz powiązanych.



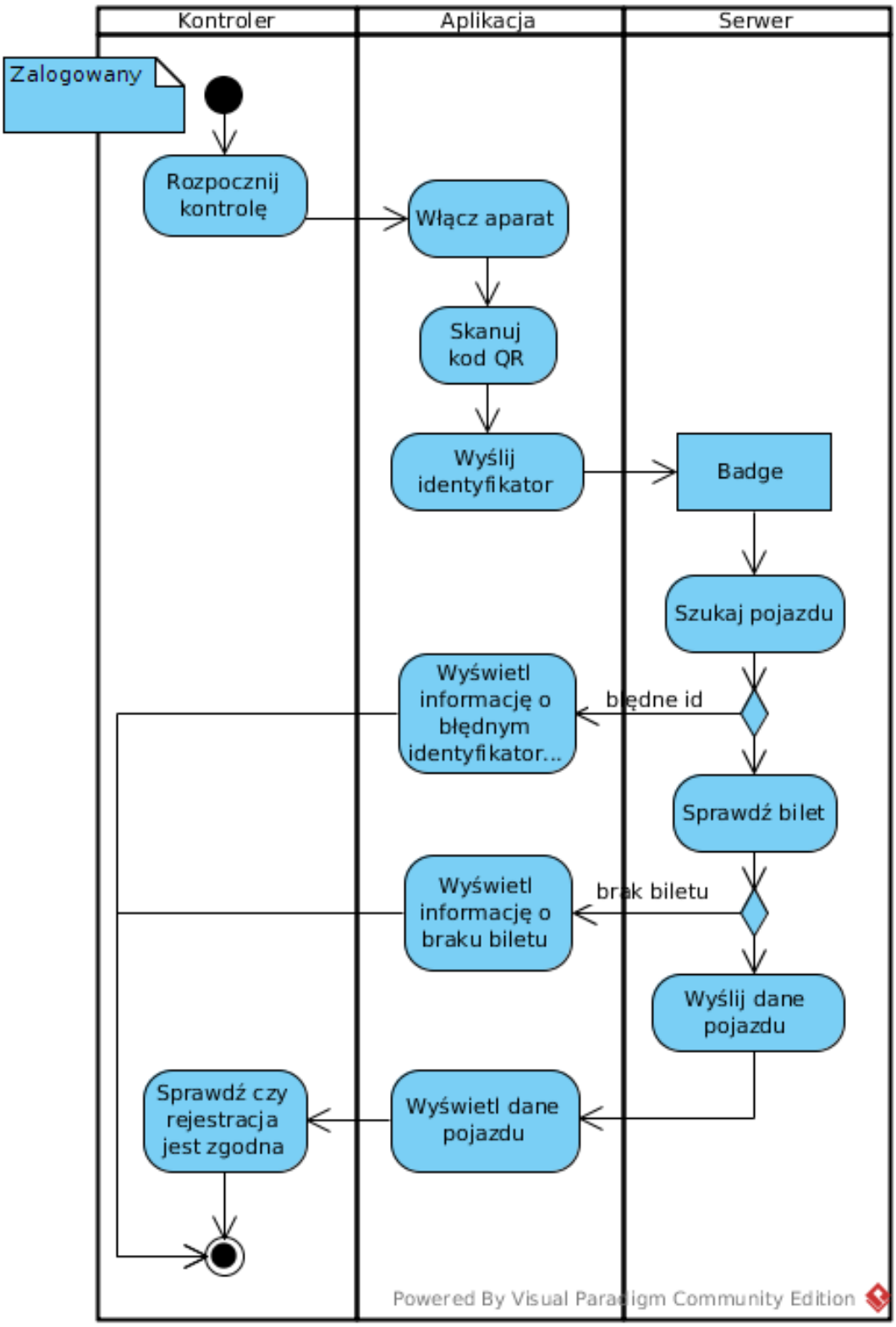
Rys. 3.6: Diagram klas dla pakietów parkings, charges oraz powiązanych.



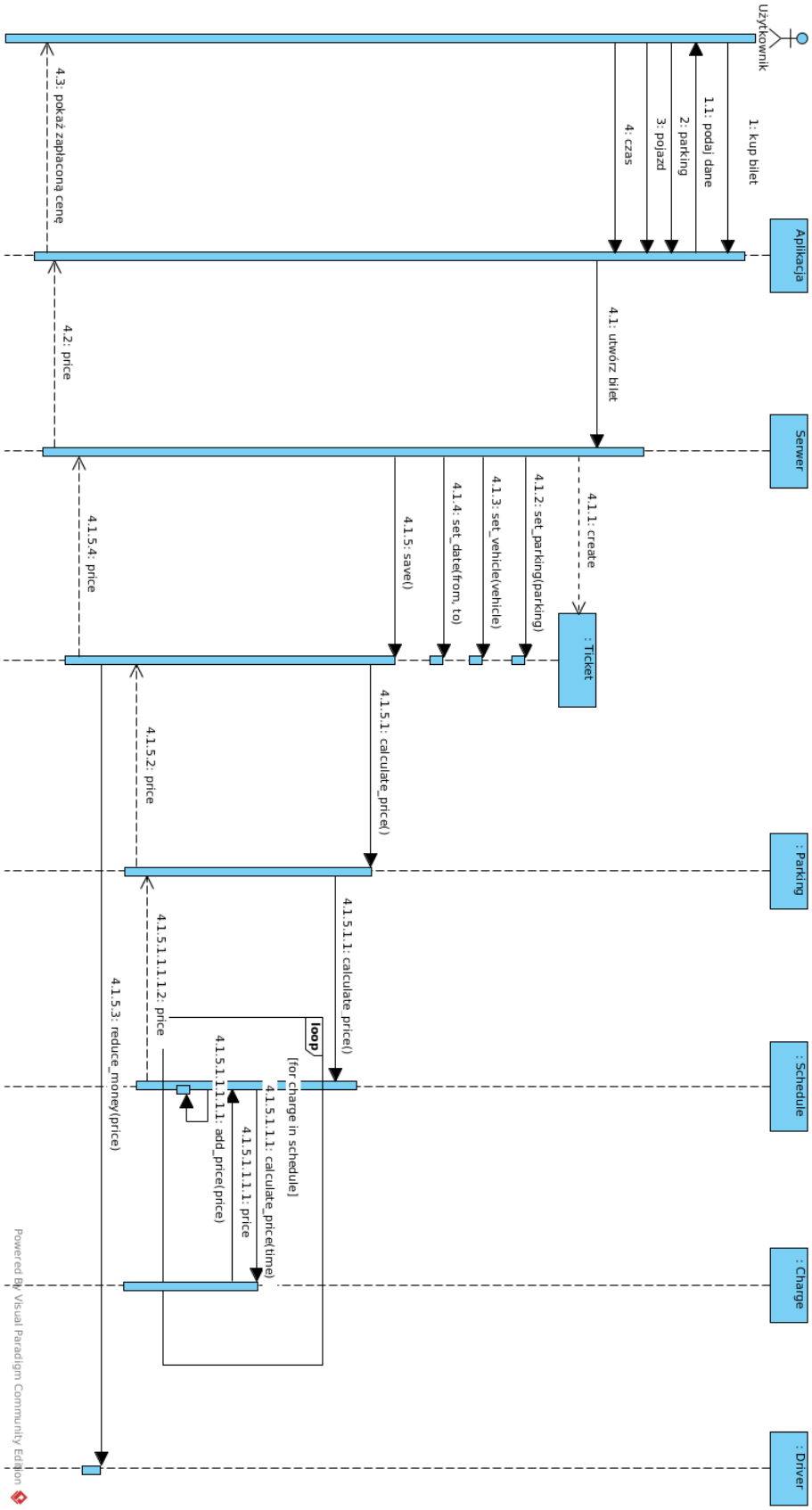
Rys. 3.7: Diagram klas rejestracji, dodawania pojazdu i zakupu biletu aplikacji mobilnej.



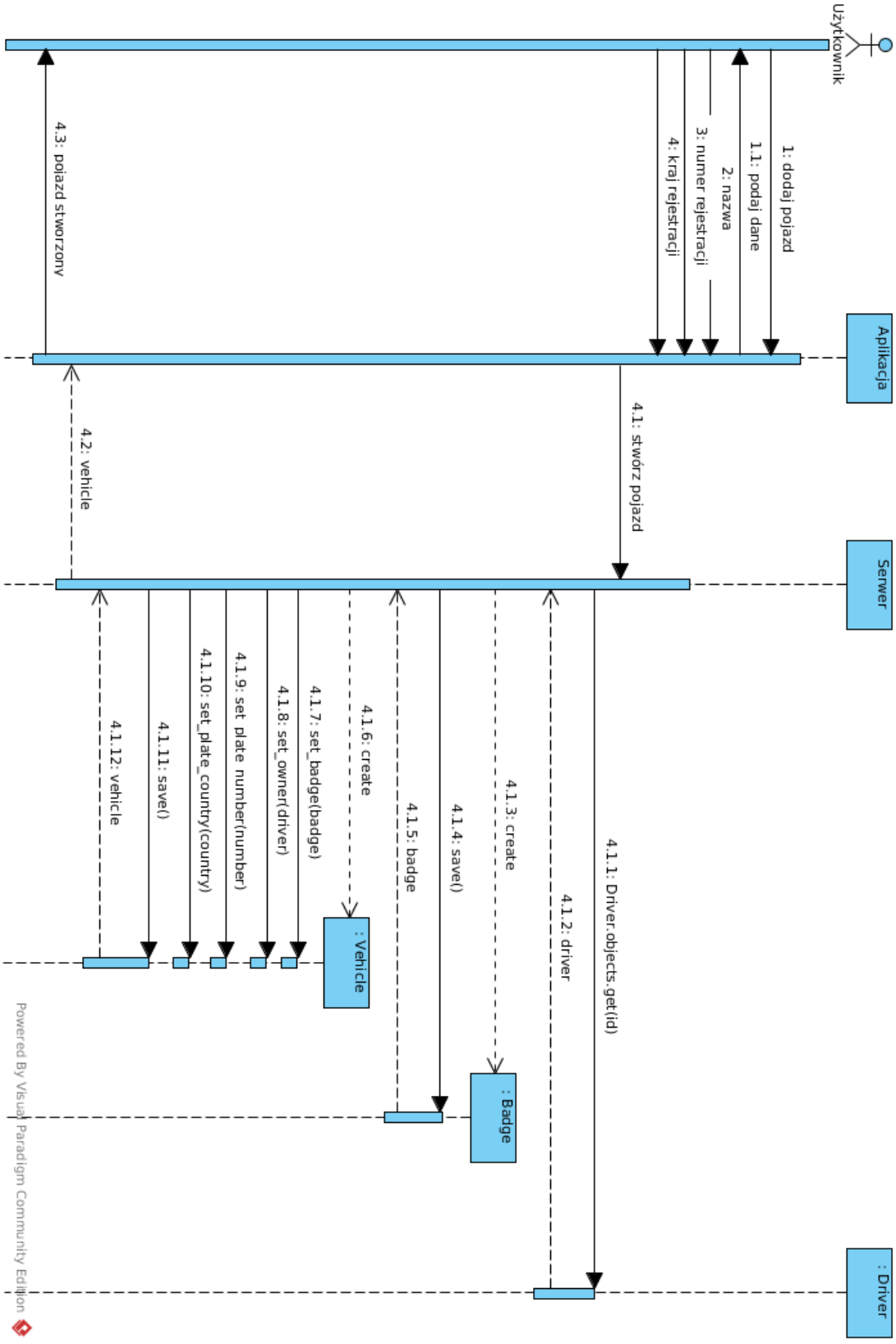
Rys. 3.8: Diagram aktywności doładowywania portmonetki.



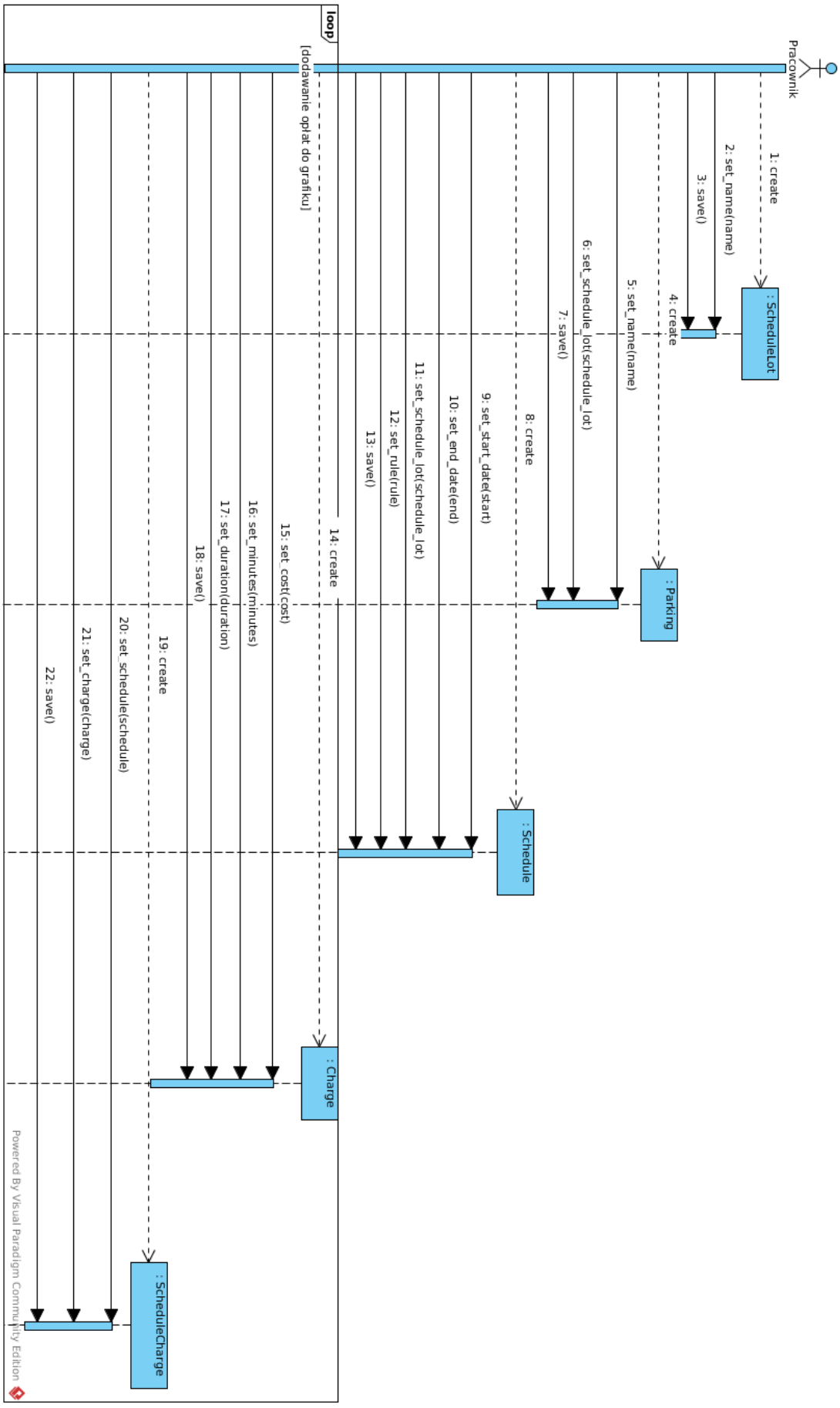
Rys. 3.9: Diagram aktywności kontrolowania biletu.



Rys. 3.10: Diagram sekwencji zakupu biletu postojowego.



Rys. 3.11: Diagram sekwencji dodawania nowego pojazdu.



Rys. 3.12: Diagram sekwencji dodawania nowego parkingu oraz opłat.

3.4 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 3.1: 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"
}
```

4 Implementacja systemu

W tym rozdziale zawarte są szczegóły implementacji systemu dla wybranych funkcjonalności, do przedstawienia których posłużono się fragmentami kodów źródłowych. Następnie na zrzutach ekranu zaprezentowane zostały wyniki działania systemu. Koniec rozdziału poświęcony został sposobom testowania, a także opisowi wykorzystanych środowisk programistycznych i edytorów kodu.

4.1 Aplikacja internetowa

Do głównych zadań serwera należą: autoryzacja, zarządzanie i identyfikowanie pojazdów, kupno oraz kontrola biletu postojowego. Wszystkie funkcjonalności udostępniane są aplikacji mobilnej poprzez API w postaci adresów URL. Wysyłając żądanie na jeden z nich, aplikacja otrzyma w odpowiedzi dokument JSON. W Django, czyli frameworku użytym do implementacji części serwerowej systemu, miejscem przeznaczonym do mapowania adresu na widok jest plik urls.py. Jego fragment został zaprezentowany na listingu 4.1. W tym miejscu znajdują się także wszystkie adresy w systemie, na jakie aplikacja mobilna będzie wysyłać swoje żądania.

Listing 4.1: Mapowane adresy URL z pliku urls.py.

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

Tworzenie konta i autoryzacja

Z procesem utworzenia konta w systemie przez kierowcę i kontrolera związanych jest kilka dodatkowych czynności, którymi są 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 kont kierowcy i 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, czyli mogą należeć do kilku grup jednocześnie. Jego zadaniem jest umożliwienie wybrania zestawu grup, spośród których użytkownik może należeć tylko do jednej w tym samym czasie. Do tego celu tworzona jest tabela pomocnicza – Role, w której umieszczane będą grupy, nazywane dalej rolami. Na listingu 4.2 pokazany został sposób, w jaki deklarowane są grupy, które będą używane do tworzenia ról. Jest to robione poprzez napisanie

klasy Pythona, która dziedziczy po BaseRole - nazwa utworzonej roli będzie pokrywać się z nazwą klasy. Dzięki możliwemu wielodziedziczeniu w tym języku, do tego celu mogą zostać wykorzystane zdefiniowane wcześniej modele.

Listing 4.2: 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 posiadać wygenerowany token autoryzacyjny. Obie te czynności muszą zostać wykonane w momencie utworzenia konta. Zostało to zrealizowane poprzez 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 4.3 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 4.3: 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:
        assign_to_role(Driver, instance.user)

@receiver(post_save, sender=Officer)
```

```
def assign_officer_to_group(instance, created, **kwargs):
    if created:
        assign_to_role(Officer, instance.user)

@receiver(post_save, sender=User)
def create_auth_token(sender, instance, created, **kwargs):
    if created:
        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 4.4), powiązanym relacją jeden-do-jeden z pojazdem (Vehicle). W modelu znajduje się także metoda generate_image(), która odpowiedzialna jest za generowanie kodu QR, w którym zakodowany będzie identyfikator. W niej wykorzystywane są funkcje pochodzące z zewnętrznej biblioteki Pythona – qrcode. Oprócz danych, podawany jest także poziom korekcji błędów i wersja kodu.

Listing 4.4: Badge - model identyfikatora.

```
class Badge(models.Model):
    uuid = models.UUIDField(default=uuid.uuid4, editable=False)

    @property
    def is_assigned(self):
        return hasattr(self, 'vehicle')

    def path_to_file(self):
        path = os.path.join(settings.BASE_DIR, 'badges', 'images')
        return '{0}/{1}.png'.format(path, self.uuid)

    def generate_image(self):
        qr = qrcode.QRCode(
            version=4,
            error_correction=qrcode.constants.ERROR_CORRECT_H,
            box_size=20,
            border=4
            )
        qr.add_data(self.uuid)
        path = os.path.join(settings.BASE_DIR, 'badges', 'images')
        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 skanowany przez kontrolera podczas sprawdzania biletu.

Taryfikator

Taryfikator oprócz powiązanych ze sobą opłat, musi zawierać także informację o czasie w którym obowiązuje. Taką możliwość daje klasa Event, pochodząca z dodatku django-scheduler.

Pozwala ona na stworzenie wydarzenia z datą początkową oraz końcową, a dzięki klasie Rule, istnieje możliwość jego powtarzania np.: co tydzień. Model Schedule rozszerza Event, dzięki czemu przetrzymuje informacje zarówno o opłatach, jak i czasie swojego obowiązywania. Dzięki możliwości tworzenia wydarzeń cyklicznych, może zostać ustawiony na wybrany dzień tygodnia. Na listingu 4.5 przedstawiono przykład jego tworzenia.

Listing 4.5: 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

Za obliczenie kosztu biletu odpowiedzialne są fragmenty kodu przedstawione na listingach 4.6 i 4.7. Bazują na zasadzie naliczania opłaty stosowanej w szczecińskiej Strefie Płatnego Parkowania, w której stawki zmieniają się w zależności od długości postoju. W klasie Schedule obliczany jest najpierw czas trwania parkowania w minutach. Następnie po pobieraniu wszystkich opłat, w pętli każda z nich 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, wyznaczając w ten sposób koszt biletu.

Listing 4.6: 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')

    price = Decimal()
    while time > 0:
        charge = charges.pop()
        if len(charges) == 0 or time <= charge.duration:
            price += charge.calculate_price(time)
            break
        else:
            price += charge.calculate_price(charge.duration)
            time -= charge.duration
    return price
```

Listing 4.7: Obliczanie częstki ceny w pojedynczej opłacie - model Charge.

```
def calculate_price(self, time):
    price = Decimal()
    price += Decimal(time // self.minutes) * self.cost
    rest = (Decimal(time % self.minutes) / self.minutes) * self.cost
    if not self.minute_billing and rest:
        price += self.cost
    else:
        price += rest
    return price
```

Obliczanie opłaty za postój następuje w momencie kupowania biletu, po czym portmonetka użytkownika zostanie obciążona odpowiednią kwotą. Jeśli nie posiada on wystarczającej ilości pieniędzy, bilet nie może zostać kupiony.

Doładowanie konta

Doładowywanie konta odbywa się dwuetapowo. Najpierw w aplikacji mobilnej użytkownik przelewa pieniądze za pomocą bramki płatności PayPal. W przypadku powodzenia transakcji zwrócony zostanie jej identyfikator, który wysyłany jest do serwera systemu ParQ. Na listingu 4.8 przedstawiony został widok odpowiedzialny za to żądanie. W linii 12 wywoływana jest metoda, w której otrzymany od aplikacji mobilnej 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). Gdyby transakcja o takim identyfikatorze już istniała, doładowywanie zostanie przerwane. Tylko w razie powodzenia wykonany będzie dalszy fragment kodu, gdzie powiązana portmonetka użytkownika jest uzupełniana o wpłaconą kwotę.

Listing 4.8: payment_list - widok doładowania konta użytkownika.

```
@api_view(['POST'])
@permission_classes((IsAuthenticated,))
def payment_list(request):
    serializer = TransactionSerializer(data=request.data)
    if serializer.is_valid():
        try:
            driver = Driver.objects.get(pk=request.user.id)
        except Driver.DoesNotExist:
            return Response('User is not driver',
                    status=status.HTTP_403_FORBIDDEN)

        money = get_payment_money(request.data['transaction_id'])
        serializer.save(user=request.user)
        driver.add_money(Decimal(money))
        driver.save()
        driver_ser = DriverSerializer(driver)
        return Response(driver_ser.data, status=status.HTTP_201_CREATED)
    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 żą-

dania 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.

4.2 Aplikacje mobilne

W ramach pracy zostały wykonane dwie oddzielne aplikacje mobilne – dla kierowcy oraz kontrolera. Kierowca oprócz logowania, może założyć konto, czy dokonać płatności celem doładowywania portmonetki. Kontroler w swojej aplikacji może skanować plakietki z kodem QR, aby przeprowadzić kontrolę biletu. 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 jest obecnie najpopularniejszym formatem, wykorzystywanym do wymiany danych w architekturze REST. Wszystkie dane to zmienne, a ich nazwy są otoczone cudzysłowami. Wartości mogą być typu string (ciąg znaków), liczbami całkowitymi i zmiennopozycyjnymi, a także tablicami w których skład wchodzą wymienione wyżej zmienne, bądź obiektem JSON. Obiekty i tablice nie mają żadnych ograniczeń, jeśli chodzi o ich zagnieżdżanie. W celu wydobycia informacji, JSON 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. Ta druga służy do analizy tablic danych.

JSONObject umożliwia parsowanie pojedynczego JSONa, a dane do analizy podawane są jako ciąg znaków (String) w konstruktorze. Metody tej klasy służące do wydobywania wartości, jako parametr przyjmują nazwę klucza z którym 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. Po wywołaniu metody getJSONObject() z indeksem elementu w parametrze, zwracany jest obiekt pojedynczego JSONa, czyli klasy JSONObject. Na listingu 4.9 znajduje się przykład z wykorzystaniem kolekcji danych.

Listing 4.9: Parsowanie jsona dla kolekcji pojazdów kierowcy.

```
JSONArray array = new JSONArray(response);

LinkedList<Vehicle> vehicleList = new LinkedList<>();

for(int i = 0; i < array.length(); i++) {
    JSONObject json = array.getJSONObject(i);
    Vehicle vehicle = new Vehicle();

    vehicle.setId(json.getInt("id"));
    vehicle.setBadge(json.getString("badge"));
    vehicle.setName(json.getString("name"));
    vehicle.setPlateCountry(json.getString("plate_country"));
    vehicle.setPlateNumber(json.getString("plate_number"));

    vehicleList.add(vehicle);
}
```

Ten fragment kodu zostaje wykonany w momencie otrzymania z serwera listy pojazdów powiązanych z danym kierowcą. Dla każdego elementu tablicy JSON wydobywany jest obiekt klasy JSONObject, z którego pobierane są dane pojazdu. Zostają 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 do komunikacji 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 danym znajdującym się w pamięci podręcznej, nie będzie ono musiało zostać ponownie wysłane.

Listing 4.10 przedstawia sposób, w jaki konstruowane są zapytania w tej bibliotece. Najpierw w konstruktorze podawany jest rodzaj metody HTTP oraz adres, na jaki żądanie ma zostać wysłane. Dwa następne parametry to obiekty klas anonimowych. Jeśli odpowiedz z serwera będzie zwrócona z kodem HTTP oznaczającym powodzenie operacji (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 błędu (4xx lub 5xx), wywołana będzie onErrorResponse(). Tak utworzone zapytanie dodawane jest do kolejki zapytań, w której moment wysłania zależy od priorytetu jaki został nadany.

Listing 4.10: 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 na licencji open source. Razem z nią, oprócz możliwości realizacji opłat, udostępniane są także gotowe ekrany dla aplikacji mobilnej, na których 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 tworzonym systemie oferowana jest tylko pierwsza opcja.

Na listingu 4.11 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 4.11: 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);
    intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
    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 czego użyta została biblioteka ZXing ("Zebra Crossing"). Służy ona do przetwarzania obrazu, w celu odkodowania kodów graficznych QR i kreskowych. Domyślnie nie są dołączone do niej żadne gotowe aktywności jak w przypadku PayPal Android SDK, istnieje jednak dodatek – ZXing Android Embedded, który takie dostarcza. Sposób działania jest dzięki niemu zbliżony do kroków, jakie należało wykonać podczas przeprowadzania transakcji. Zawiera on gotową aktywność, w której zostaje uruchomiona kamera urządzenia mobilnego. Na listingu 4.12 w klasie IntentIntegrator konfigurowany jest najpierw skaner, gdzie podawany jest rodzaj kodów graficznych jakich ma poszukiwać oraz orientacja ekranu. Po natrafieniu przez skaner na kod, skanowanie zostaje zakończone, a informacja jaką udało się odkodować zwracana jest w metodzie onActivityResult().

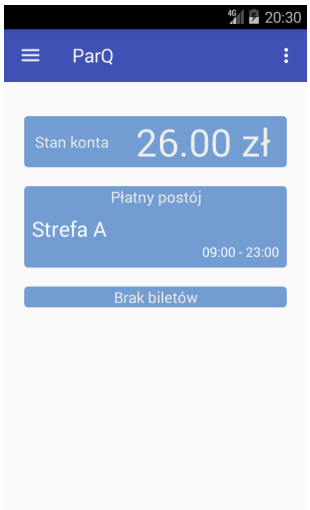
Listing 4.12: Intencja rozpoczynająca aktywność skanowania.

```
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  }
```

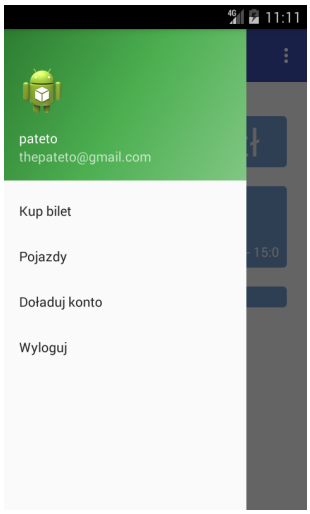
4.3 Wyniki działania systemu

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. Poniżej znajduje się opis funkcjonującego systemu, z zrzutami ekranów z 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 obowiązywania płatnego postoju. Tutaj też są wyświetlane informacje o wszystkich aktywnych biletach postojowych, wraz z informacjami o pojeździe na który zostały zakupione i godziną ich 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.



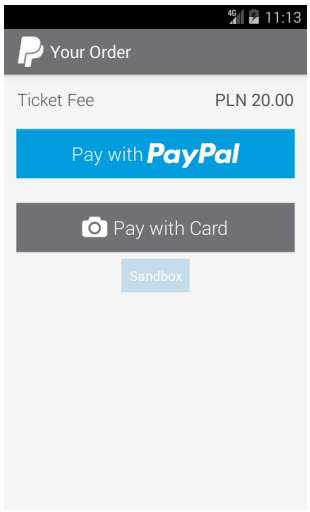
Rys. 4.1: Informacje o koncie.



Rys. 4.2: Menu boczne aplikacji.

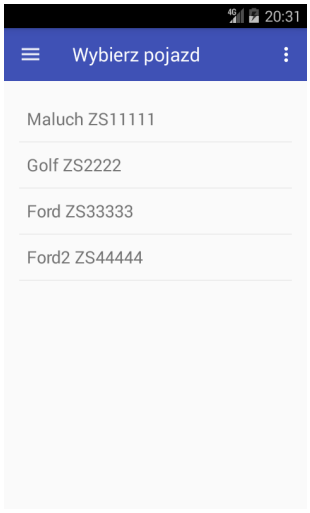
Doładowywanie konta (rys. 5.3 i 5.4) rozpoczyna się od podania kwoty, jaką konto użytkownika w systemie ma zostać doładowane. Zatwierdzenie jej, powoduje pokazanie ekranów pochodzących z wykorzystywanej do realizacji płatności biblioteki – PayPal Android SDK. Umożliwia ona wybranie metody płatności,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

Rys. 4.3: Pole z kwotą doładowania.

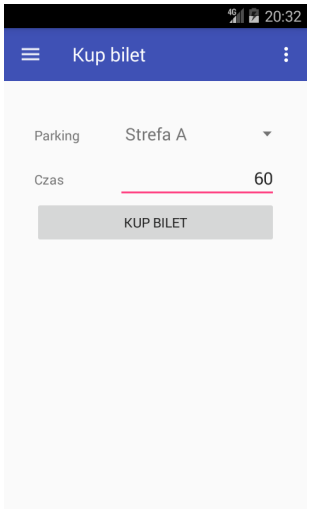


Rys. 4.4: Realizacja płatności w PayPal.

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.

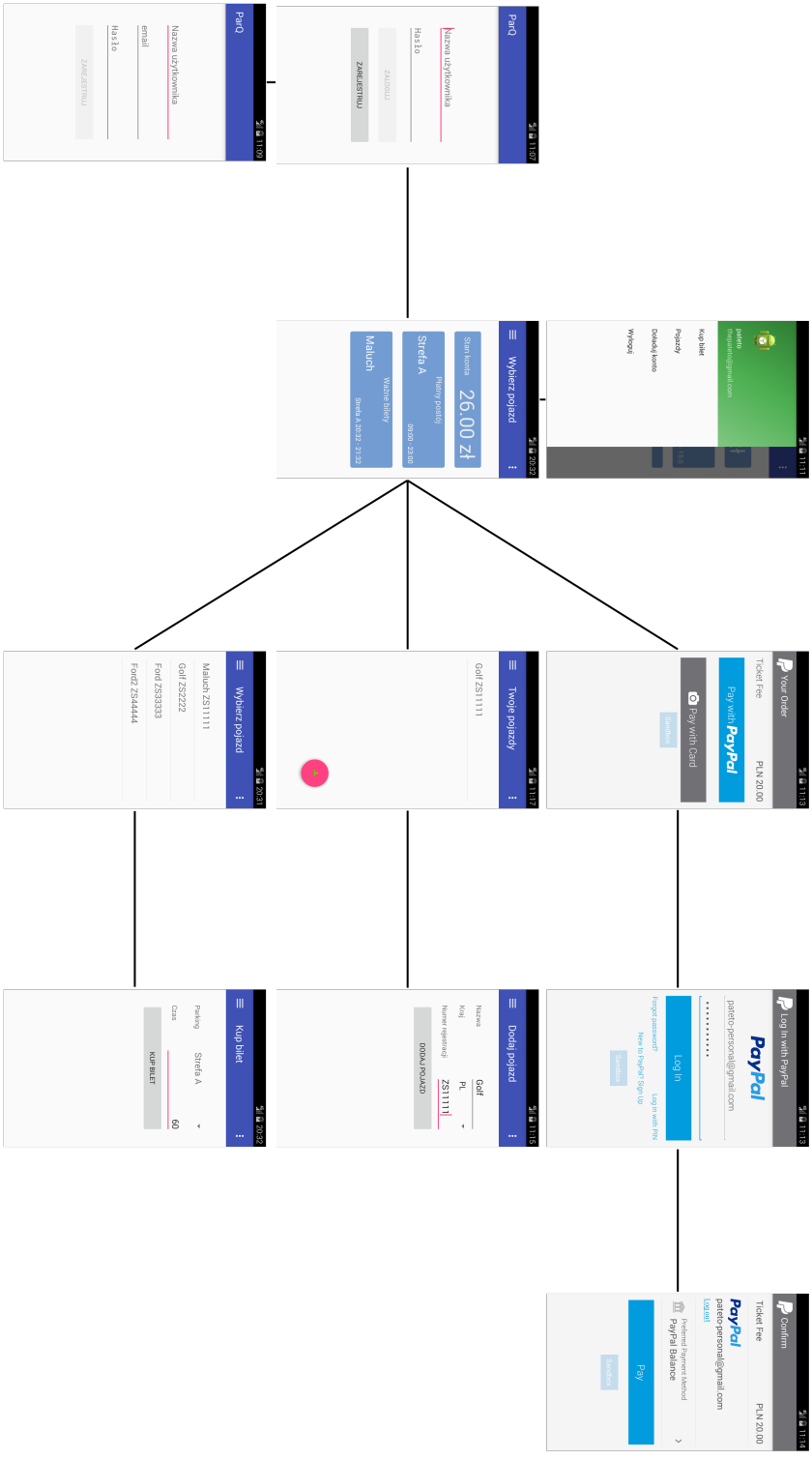


Rys. 4.5: Zakup biletu - wybór pojazdu.



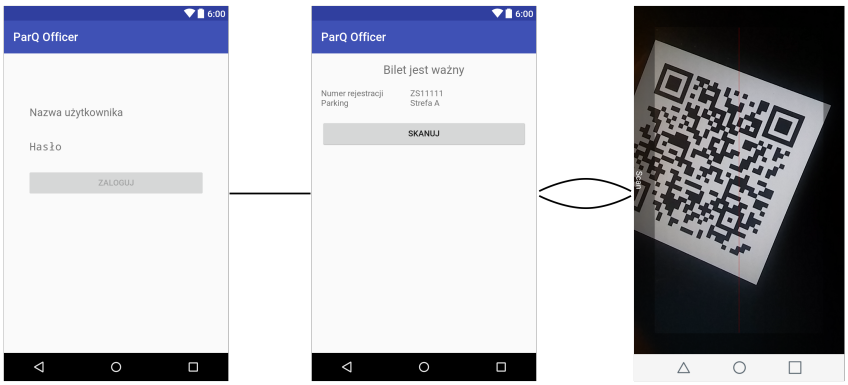
Rys. 4.6: Zakup biletu - parking i czas.

Na rys. 4.7 znajduje się schemat, na którym przedstawione zostały wszystkie ekrany oraz możliwe przejścia między nimi w aplikacji przeznaczonej dla kierowców.



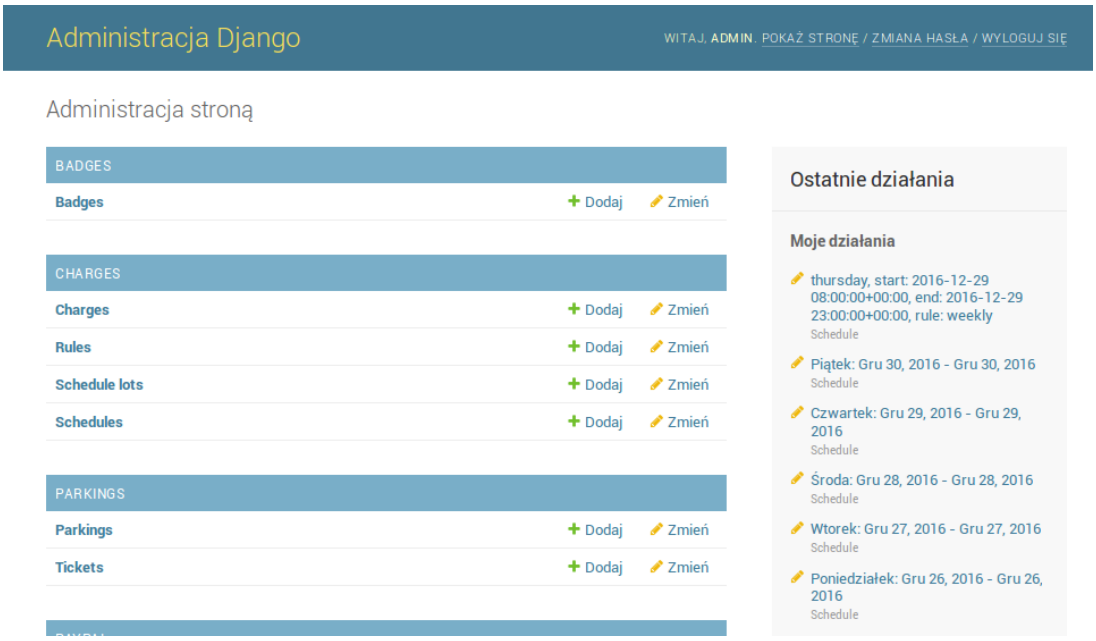
Rys. 4.7: Ekrany w aplikacji kierowcy.

Druga z aplikacji umożliwiająca przeprowadzenie kontroli, a w jej skład wchodzą trzy ekrany. 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. 4.8 znajdują się ekrany aplikacji kontrolera.

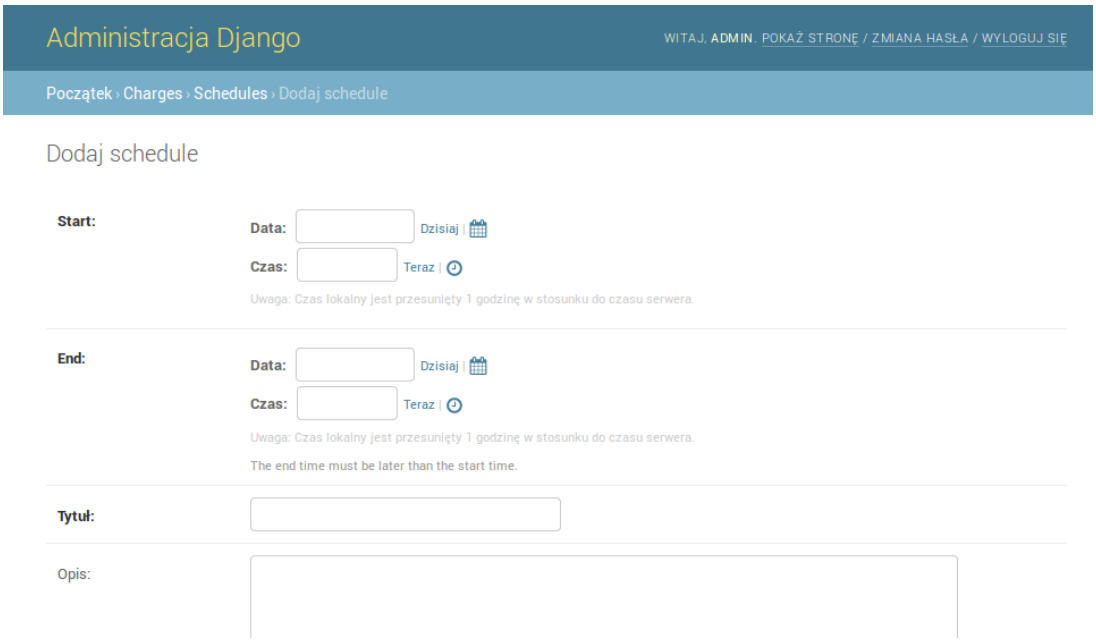


Rys. 4.8: Ekrany w aplikacji kontrolera.

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. 4.9 została przedstawiona strona główna panelu administratora. Rys. 4.10 przedstawia jedną z podstron, w której tworzony jest nowy grafik.



Rys. 4.9: Główny panel administratora.



Rys. 4.10: Dodawanie nowego grafiku.

4.4 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 musi posiadać 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 w celu dokładnego przetestowania, aplikacja mobilna była uruchamiana na emulatorze oraz rzeczywistym urządzeniu z systemem Android.

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.

4.5 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 programistycznych, 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 rozszerzeniami – Vundle. Do pracy wykorzystane zostały dodatki YouCompleteMe (okna z podpowiedziami i uzupełnianie kodu) i NERD Tree (wyświetlanie struktury plików).

Podsumowanie

Celem niniejszej pracy było stworzenie systemu płatności dla strefy płatnego parkowania. Do jego głównych zadań należało określenie wymagań systemu, zaimplementowanie serwera oraz aplikacji mobilnych. Użytkownicy korzystający z tego systemu mogą doładowywać swoje konta, rejestrować pojazdy oraz kupować bilety postojowe. Osoby kontrolujące mają możliwość sprawdzania ważności biletów postojowych. Główną cechą, która wyróżnia ten system spośród większości istniejących na rynku rozwiązań, jest zastosowanie kodów QR, za pomocą których identyfikowane są zaparkowane samochody. Postawiony w tej pracy cel został zrealizowany.

Pierwszy rozdział w całości poświęcony był płatnościom elektronicznym. Przedstawione w nim informacje pozwoliły na dokonanie analizy różnych metod płatności, pod kątem ich przydatności w systemach internetowych. Na bazie tych rozważań wybrana została forma zawierania transakcji, która najlepiej spełnia wymagania tworzonego systemu. W drugim rozdziale czytelnik został zapoznany z technologiami wykorzystywanymi do realizacji zadań pracy, a także pojęciami z nimi związanymi. Były one istotne dla pełnego zrozumienia dalszej części pracy. W rozdziale 3. omówiona została Strefa Parkowania w Szczecinie oraz dokonano porównania systemu ParQ z już istniejącymi rozwiązaniami, które spełniają podobne zadania. Kolejny rozdział zawiera dokumentację techniczną, natomiast ostatni, poświęcony został szczegółom implementacji.

Najbardziej pomocnymi pozycjami z bibliografii, były publikacje o tematyce dotyczącej płatności elektronicznych. Szczególnie przydatna okazała się książka autorstwa Artura Borcucha, pt. *Pieniądz elektroniczny pieniądz przyszłości – analiza ekonomiczno-prawna*. W niezwykle wyczerpujący sposób opisane zostały w niej poszczególne etapy rozwoju pieniądza elektronicznego, które omówiono w rozdziale pierwszym. Szczególnie ważna na etapie implementacji systemu była dokumentacja Django. Głównie dzięki przejrzystości oraz licznym przykładom w postaci fragmentów kodu, którymi opatrzone zostało prawie każde poruszane tam zagadnienie.

Mam nadzieję, że zebrane materiały okazały się wystarczające do zapoznania z zastosowanymi narzędziami oraz technologiami, użytymi do realizacji zadań pracy. Wybór tematu został podyktowany także rosnącą popularnością elektronicznych metod w internecie i aplikacjach mobilnych. Liczę, że podjęta analiza różnych form płatności, wykazała słuszność zastosowanej w tym systemie metody.

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).