



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

KATEDRA INFORMATYKI STOSOWANEJ I FIZYKI KOMPUTEROWEJ

Projekt dyplomowy

Aplikacja mobilna wspomagająca wyszukiwanie wolnych miejsc na
parkingach wielopoziomowych

Mobile Application Supporting Finding Available Spaces in
Multi-Level Parking Lots

Autor:	Karol Hoerner de Roithberg
Kierunek studiów:	Informatyka Stosowana
Opiekun pracy:	dr inż. Ewa Olejarz-Mieszaniec

Kraków, 2024

Spis treści

1	Wstęp	4
1.1	Wprowadzenie	4
1.2	Cel projektu	5
1.3	Podobne rozwiązania	6
1.3.1	Model inteligentnego parkingu IoT	6
1.3.2	MRS-Trafic system	6
1.3.3	Mobiparking	6
2	Technologie	7
2.1	Aplikacja mobilna	7
2.1.1	Java Script	7
2.1.2	React	7
2.1.3	React Native	8
2.1.4	Dodatkowe biblioteki	8
2.2	Serwer	9
2.2.1	Python	9
2.2.2	Django	9
2.2.3	Dodatkowe biblioteki	9
2.3	Baza danych	10
2.3.1	MySQL	10
3	Implementacja	11
3.1	Aplikacja mobilna	11
3.1.1	Struktura kodu i funkcjonalności	11
3.1.2	Nawigacja	14
3.2	Serwer	15
3.2.1	Struktura kodu i funkcjonalności	15
3.3	Baza danych	20
3.4	Przepływ danych	21
4	Testowanie	23
4.1	Testy aplikacji	23
4.2	Testy serwera	23
4.2.1	Parking	23
4.2.2	Użytkownicy	24
4.2.3	Rezerwacje	25
4.2.4	Sklepy	26

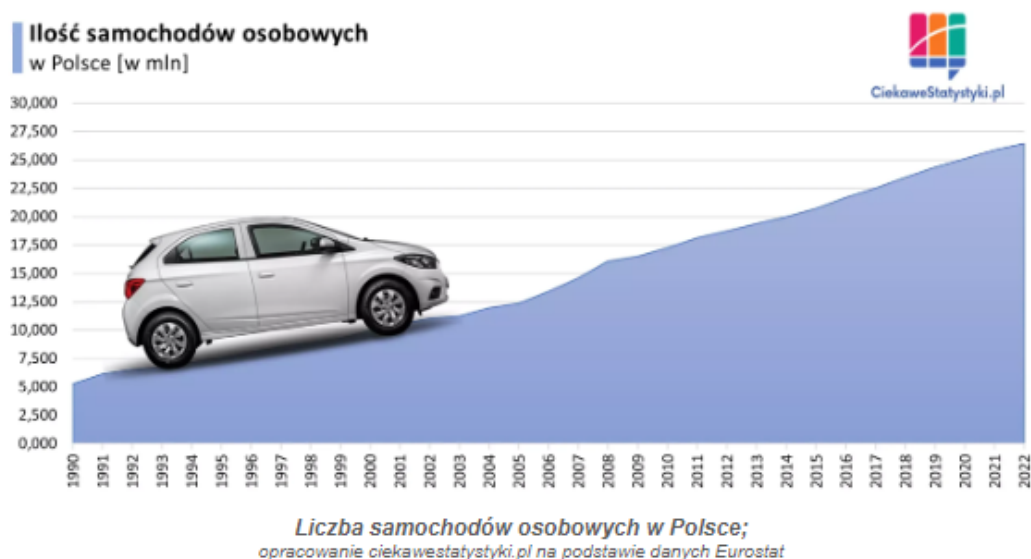
5	Dokumentacja użytkownika	28
5.1	Rejestracja i logowanie	28
5.2	Wybór parkingu oraz panel administratora	30
5.2.1	Ekran wyboru parkingu	30
5.2.2	Panel administratora	32
5.3	Ekran parkingu oraz płatności	33
5.3.1	Ekran parkingu	34
5.3.2	Ekran płatności	36
6	Podsumowanie	38
6.1	Wnioski	38
6.2	Perspektywy dalszego rozwoju	38
7	Bibliografia	40

Rozdział 1

Wstęp

1.1 Wprowadzenie

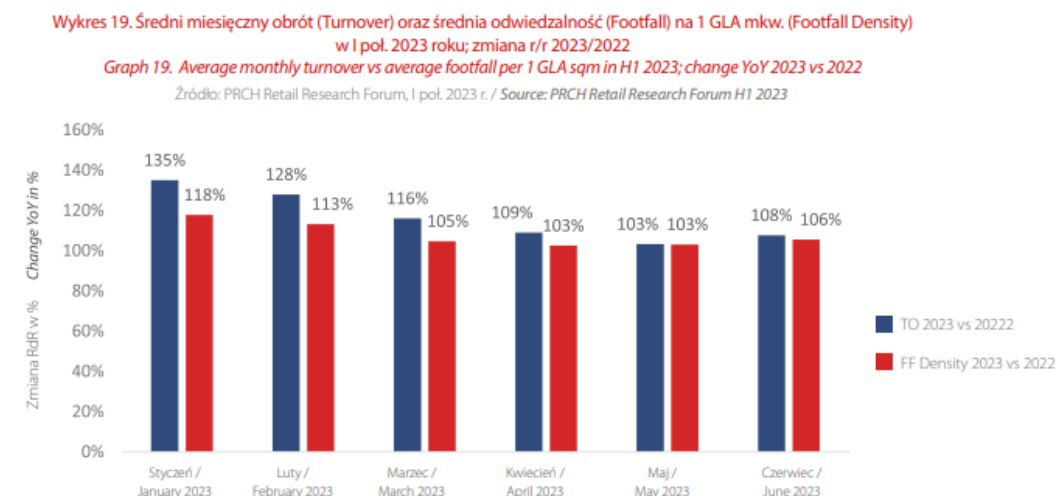
Wskaźnik motoryzacji to ilość aut podzielona przez liczbę mieszkańców kraju. Według danych Eurostat w Polsce w latach 2001 a 2021 wskaźnik ten wzrósł o 412 (rysunek 1.1). Oznacza to że w tym okresie w Polsce przybyło ponad 15 milionów nowych samochodów osobowych. W roku 2021 liczba aut w Polsce przypadająca na 1000 obywateli wynosi 687 samochodów, co plasuje nasz kraj na pierwszym miejscu w Unii Europejskiej. Kolejnymi krajami w tym rankingu są Luksemburg ze wskaźnikiem 682 oraz Włochy z liczbą 675 pojazdów na 1000 osób [1].



Rysunek 1.1: Liczba samochodów osobowych w Polsce w latach 1990-2022 [w mln].
Źródło: [1]

Galerie handlowe są jednymi z najpopularniejszych miejsc spotkań społecznych, odwiedzane są przez miliony Polaków każdego roku. Od okresu pandemii koronawirusa, odwiedzalność galerii, czyli ilości odwiedzających galerie osób w przeliczeniu na wielkość obiektu, rośnie z roku na rok (rysunek 1.2). Pierwszym miesiącem bez obostrzeń pandemicznych był kwiecień 2022, porównując go z kwietniem 2023, odwiedzalność galerii

wzrosła o 2,6% [2], a różnica ta między czerwcem 2022 a czerwcem 2023 wynosiła już 5,8% [3].



Rysunek 1.2: Średni miesięczny obrót (Turnover) oraz średnia odwiedzalność (Footfall) na 1 GLA mkw. (Footfall Density) w I poł. 2023 roku; zmiana r/r 2023/2022. Źródło: [4]

Biorąc pod uwagę zwiększającą się liczbę samochodów osobowych na drogach oraz przyrost odwiedzalności galerii, parkowanie w popularnie odwiedzanych przez ludzi miejscach może być wymagające i czasochłonne, dlatego też notowane jest zapotrzebowanie na różne rozwiązania informatyczne niwelujące te problemy.

1.2 Cel projektu

Głównym celem projektu jest stworzenie aplikacji mobilnej pozwalającej kierowcą na proste znalezienie wolnego miejsca na parkingach, przy założeniu, że parking ma wbudowany system detekcji wolnych oraz zajętych miejsc postojowych, oraz API przetwarzające i udostępniające te dane. Aplikacja ma zapewnić możliwość wyboru strefy oraz piętra miejsca parkingowego, ma ona pozwolić również na wybór miejsca w pobliżu wybranego przez użytkownika sklepu, co zaoszczędzi czas klientom, którzy mają w planie odwiedzić określone miejsce. W celu ułatwienia dojazdu do przypisanego miejsca w aplikacji ważne jest zilustrowanie schematu parkingu z odpowiednio pokolorowanymi miejscami postojowymi. Plany parkingu ułatwią również powrót do auta, który po długim czasie spędzonym na zakupach może być problematyczny.

W przypadku parkingów płatnych można opłacić postój na wybraną ilość czasu lub przedłużyć czas parkowania, gdy zajdzie taka potrzeba. Jeśli miejsca mają bezpłatny okres początkowy lub opłacony czas się kończy, system powinien wysłać na telefon powiadomienie o tym informujące.

1.3 Podobne rozwiązania

W związku z popularnością i możliwościami technologicznymi obecnych urządzeń mobilnych inteligentne systemy parkingowe są w ostatnich latach popularnym tematem. Poniżej opisano kilka takich rozwiązań.

1.3.1 Model inteligentnego parkingu IoT

Model oparty o IoT (Internet of Things) polega na ciągłym monitorowaniu przestrzeni parkingowej za pomocą sensorów takich jak kamery, czujniki ultradźwiękowe czy czujniki podczerwieni. System w czasie rzeczywistym przekazuje dane o wolnych oraz zajętych miejscach użytkownikowi przez aplikację mobilną. System ten jest potencjalnym rozwiązaniem i nie znaleziono informacji o jakichkolwiek jego wdrożeniach [5].

1.3.2 MRS-Trafic system

MRS-Trafic system korzysta z radiowych czujników U-Spot zamontowanych na każdym miejscu parkingowym. Czujniki przekazują informacje o stanie miejsca parkingowego do głównego kontrolera U-Box. Przekazanie informacji użytkownikowi polega na wypisaniu ilości wolnych miejsc w strefie lub alejce za pomocą słupów z wyświetlaczami. Plusem tego sposobu jest zwrócenie uwagi na strefy specjalne takie jak strefy dostaw lub strefa wyłączona z ruchu, po ustawieniu sensorów w takich miejscach mogą one przesyłać do systemu komunikaty o błędnym parkowaniu. Minusem tego układu jest brak wskazania klientowi dokładnego miejsca parkingowego, jest pokazana tylko ilość miejsc wolnych w strefie [6].

1.3.3 Mobiparking

Mobiparking jest aplikacją dostępną w 247 miejscach w całej Polsce. Polega ona na opłaceniu parkowania w strefach płatnych, wystarczy w aplikacji podać numery rejestracyjne swojego auta oraz miasto, w którym parkujemy. Bardzo dużym plusem jest możliwość zapłacenia za postój bez potrzeby szukania parkomatu. Kolejnym wielkim plusem jest możliwość opłacenia rzeczywistego czasu parkowania, co jest bardzo oszczędne. Jedynym defektem aplikacji jest brak dostępnych informacji o miejscach parkingowych, użytkownik musi je znaleźć samodzielnie [7].

Rozdział 2

Technologie

System został zbudowany w oparciu o trójwarstwowy model architektury, jest podzielony na trzy połączone ze sobą moduły: serwer, baza danych oraz interfejs aplikacji mobilnej. Serwer napisany jest w frameworku Django oraz języku Python, baza danych stworzona została w systemie MySQL, natomiast aplikacja mobilna została napisana w języku JavaScript udostępniającym frameworki React oraz React Native.

2.1 Aplikacja mobilna

2.1.1 Java Script

JavaScript jest językiem skryptowym oraz jest obecnie jednym z najpopularniejszych języków programowania. Cechują go wysokopoziomowość, jednowątkowość oraz dynamiczne dobieranie typów. Pomimo jednowątkowości możliwe jest pisanie funkcji asynchronicznych przy użyciu Promise API. Głównym zamysłem języka jest budowanie aplikacji programowaniem funkcjonalnym, ale JavaScript wspiera również możliwość tworzenia obiektów oraz dziedziczenia. Główną zaletą JavaScript jest to, że jest on jednym z niewielu języków natywnie obsługiwanych przez obecne przeglądarki internetowe, dlatego służy on głównie do budowania interfejsu aplikacji internetowych. Ponadto obecnie jest również używany do pisania serwerów za pomocą środowiska Node.js, aplikacji mobilnych z użyciem React Native lub Ionic, oraz aplikacji desktopowych za pomocą frameworku Electron [8].

2.1.2 React

React jest środowiskiem języka JavaScript do budowania interfejsów użytkownika, jest aktualnie najpopularniejszym narzędziem do budowania frontendu aplikacji internetowych. React bazuje na budowaniu komponentów interfejsu w języku HTML, które można używać wielokrotnie, zaletą jest uproszczenie budowania komponentów do minimum, czyli funkcji języka JavaScript. Komponentom można przekazywać również dane, które można później użyć w jego funkcjonalności lub interfejsie użytkownika, co daje możliwość zmiany stanu aplikacji w czasie rzeczywistym. React umożliwia również wbudowane funkcje zwane Hooks, które zezwalają na zmianę stanu wewnątrz zbudowanych komponentów, do najpopularniejszych należą `useEffect`, które aktualizuje stan aplikacji podczas zmian w interfejsie oraz `useState`, które zmienia wewnętrzny stan komponentu.

tu. Głównym powodem tak dużej popularności Reacta jest to, że jego kod źródłowy jest publicznie dostępny. Dzięki temu społeczność może sama rozwijać jego możliwości i publikować nowe biblioteki, oraz komponenty [9].

2.1.3 React Native

React Native jest biblioteką pozwalającą programować aplikacje na urządzenia Android, iOS oraz przeglądarki internetowe z jednego kodu źródłowego, co jest wielką zaletą tego języka. React Native działa na podobnej zasadzie jak framework React, jednak zamiast języka HTML, używa natywnych komponentów, które można znaleźć w systemach operacyjnych, do najczęściej używanych należą View, Button, Text, Switch oraz ScrollView. Podczas używania aplikacji elementy te zostaną podmienione na natywne komponenty odpowiedniej platformy. Dodatkowo React Native posiada moduł sprawdzający, w jakim systemie operacyjnym aplikacja jest uruchamiana, dzięki czemu możemy renderować różne komponenty na różnych platformach. Technologia React Native pozwala również podłączyć się pod wbudowane możliwości telefonu takich jak aparat, powiadomienia lub geolokalizacja. Tak samo jak w technologii React, dzięki udostępnieniu kodu źródłowego, społeczność może sama tworzyć komponenty oraz biblioteki [10].

2.1.4 Dodatkowe biblioteki

Ponadto w aplikacji mobilnej zostały użyte dodatkowe biblioteki, jak:

- *expo* - to platforma z publicznym kodem źródłowym do budowania natywnych aplikacji na systemy Android, iOS oraz przeglądarki internetowe, udostępnia platformę wykonawczą i dodatkowe biblioteki [11].
- *react-navigation* - jest to biblioteka tworząca strukturę aplikacji za pomocą nawigatorów, które kierują użytkownika na poszczególne ekrany. Posiada ona różne sposoby nawigacji, np. nawigator stosu, nawigator dolnej karty i nawigator szufladowy [12].
- *react-native-screens* - biblioteka zintegrowana z powyższą biblioteką *react-navigation*, pozwalająca dodawać ekrany do nawigacji [13].
- *react-native-safe-area-context* - biblioteka udostępniająca API do odczytywania bezpiecznego obszaru urządzenia [14].
- *react-native-async-storage* - biblioteka dająca dostęp do asynchronicznego systemu przechowywującego dane w schemacie klucz, wartość [15].
- *react-native-swiper* - biblioteka udostępniająca komponent do przewijania między wyświetlanymi elementami listy [16].
- *react-native-dropdown-select-list* - biblioteka zapewniająca odpowiednik komponentu *select* w języku HTML dla React Native [17].
- *expo-checkbox* - biblioteka zapewniająca komponent pole wyboru dla środowiska React Native [18].
- *validator* - biblioteka udostępniająca funkcje, które sprawdzają poprawność łańcuchów znaków [19].

2.2 Serwer

2.2.1 Python

Python to obecnie najpopularniejszy skryptowy język programowania ze względu na prostotę nauki oraz praktyczność w dużych projektach. Jest on wykorzystywany do pisania serwerów aplikacji internetowych w środowiskach Django oraz Flask, jest również głównym językiem używanym do przetwarzania danych dzięki bibliotekom *pandas* i *NumPy*, oraz sztucznej inteligencji dzięki bibliotekom *TensorFlow* i *PyTorch*, bardzo przydatne do operacji na danych są arkusze Jupyter notebook, w których kod można pisać w osobnych komórkach, a dane wyjściowe są wyświetlane bezpośrednio pod kodem. Python wyróżnia się spośród innych języków strukturą kodu, zamiast nawiasów klamrowych do definiowania funkcji, pętli lub wyrażeń warunkowych używane są dwukropki oraz wcięcia, lecz tak samo jak inne języki wspiera programowanie obiektowe oraz funkcjonalne [20].

2.2.2 Django

Django jest wysokopoziomowym środowiskiem do pisania aplikacji internetowych w prosty i przejrzysty sposób, które posiada wiele wbudowanych bibliotek i funkcji, co ułatwia rozwijanie aplikacji bez potrzeby dodatkowych bibliotek. Jednym z głównych dodatków Django jest mapowanie obiektowo-relacyjne (ORM), które zezwala na tworzenie baz relacyjnych za pomocą obiektów tworzonych w języku Python. Django pozwala stworzyć kilka osobnych aplikacji w jednym projekcie, co zwiększa przejrzystość kodu [21]. Głównymi plikami w aplikacjach Django są:

- `settings.py` - plik definiujący ustawienia całego projektu.
- `apps.py` - plik, który inicjalizuje moduł aplikacji
- `views.py` - plik, w którym tworzy się funkcje wykonywane podczas zapytań do aplikacji
- `urls.py` - plik tworzący ruting aplikacji oraz przywiązujący je do funkcji z pliku `views.py`
- `models.py` - plik, w którym tworzy się modele bazy danych
- `tests.py` - plik, w którym tworzy się testy aplikacji

2.2.3 Dodatkowe biblioteki

Po stronie serwera zostały użyte dodatkowe biblioteki, jak:

- *django-rest-framework* - biblioteka udostępniająca wiele funkcji dla tworzenia aplikacji internetowych takich jak: autoryzacja, serializacja, itp. [22].
- *rest-framework-simplejwt* - biblioteka pozwalająca uwierzytelnianie użytkowników za pomocą *JSON Web Token*, korzysta również z funkcji biblioteki *Django Rest Framework* [23].
- *networkx* - pakiet służący do tworzenia oraz operacji na złożonych sieciach. Posiada wiele funkcji do operacji na grafach [24].

- *background_task* - biblioteka Django pozwalająca na wykonywanie zadań w osobnym wątku, daje również możliwość wykonywania zadań cyklicznych [25].
- *mysqlclient* - pakiet pozwalający na połączenie aplikacji napisanej w języku Python z bazą danych MySQL [26].

2.3 Baza danych

2.3.1 MySQL

MySQL to relacyjna baza danych używająca język SQL, czyli język zapytań strukturalnych. Relacyjne bazy danych organizują dane za pomocą tabel, gdzie kolumny przechowują atrybuty lub typy danych, a wiersze przechowują osobne rekordy z unikatowymi id zwanymi głównymi kluczami. Między tabelami możemy tworzyć relacje, przechowując główny klucz w innej tabeli, nazywając go kluczem zewnętrznym. Relacyjne bazy danych mają trzy typy relacji: jeden do jednego (1:1), jeden do wielu (1:N) oraz wiele do wielu (N:N). Język SQL polega na wykonywaniu zapytań, które mogą tworzyć tabele, zmieniać, wpisywać oraz odczytywać dane z tabel [27].

Rozdział 3

Implementacja

W poniższym rozdziale zostaną opisane najważniejsze punkty implementacji projektu oraz przepływu danych między jego modułami.

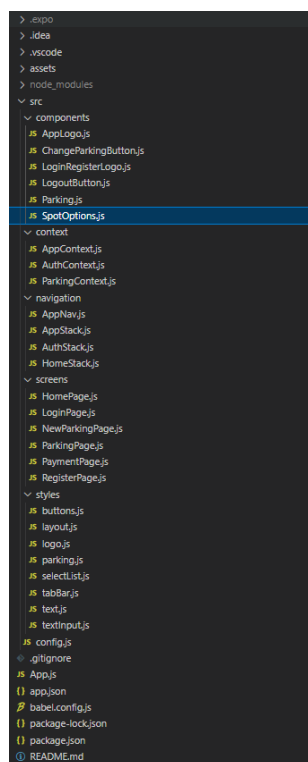
3.1 Aplikacja mobilna

Aplikacja mobilna została napisana w języku JavaScript z użyciem środowisk React oraz React Native. Do projektu zostały użyte również dodatkowe biblioteki dla przejrzystości kodu oraz ich funkcjonalności.

3.1.1 Struktura kodu i funkcjonalności

Na rysunku 3.1 widać strukturę plików aplikacji mobilnej. Głównym plikiem wykonywalnym aplikacji jest plik `App.js`, którego kod jest przedstawiony na rysunku 3.2. Aplikację mobilną uruchamia się komendą `npm start`. Wewnątrz tego pliku znajdują się komponenty `AppNav`, `AuthProvider` oraz `.`. `AppNav` zajmuje się nawigacją całej aplikacji, będzie ona opisana szczegółowo w kolejnym punkcie.

Komponenty `AuthProvider` oraz `AuthProvider` są komponentami dostawcy, z których można przekazywać dane oraz funkcje do głębiej zagnieżdżonych komponentów aplikacji (rysunek 3.3d), pochodzą one z komponentów kontekstu `AppContext` (rysunek 3.3b) oraz `AuthContext` (rysunek 3.3a). Funkcjami komponentu `AuthContext` są autoryzacja oraz przechowywanie danych o zalogowanym użytkowniku, zapisuje on również *JSON Web Token* w *AsyncStorage*, który używany jest do automatycznego zalogowania użytkownika po zamknięciu aplikacji. `AppContext` zapewnia po zalogowaniu listę parkingów dostępnych w systemie, oraz przekazuje do komponentu `ParkingContext` (rysunek 3.3c) informację o wybranym przez użytkownika parkingu. Komponent `ParkingContext` zapewnia informacje o całym parkingu razem z jego ilością pięter, stref oraz zajętych lub wolnych miejscami, sklepach przypisanych do tego parkingu oraz informację czy parking jest płatny.



Rysunek 3.1: Struktura plików w aplikacji mobilnej

```
export default function App() {
  return (
    <AuthProvider>
      <AppProvider>
        <AppNav />
      </AppProvider>
    </AuthProvider>
  );
}
```

Rysunek 3.2: Kod źródłowy pliku App.js

W folderze screens znajdują się ekrany aplikacji, między którymi porusza się użytkownik, opis ekranów znajduje się w następnym punkcie nawigacja. Ekrany zawierają w sobie stworzone komponenty znajdujące się w folderze components. Komponenty te zostały stworzone dla przejrzystości kodu, Parking (komponent rysujący schemat parkingu) oraz SpotOptions (komponent wyświetlający wybór opcji parkowania), oraz z uwagi na to, że powtarzają się na kilku ekranach: LogoutButton, ChangeParkingButton, AppLogo oraz LoginRegisterLogo.

```
export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {

  const [userToken, setUserToken] = useState(null);
  const [id, setId] = useState(null);
```

(a) Stany komponentu AuthContext

```
export const AppContext = createContext();

export const AppProvider = ({ children }) => {

  const [parkingNames, setParkingNames] = useState([]);
  const [parkingName, setParkingName] = useState(null);
```

(b) Stany komponentu AppContext

```
export const ParkingContext = createContext();

export const ParkingProvider = ({ children }) => {

  const [parking, setParking] = useState([]);
  const [isPaid, setIsPaid] = useState(null);
  const [shops, setShops] = useState([]);
  const [levels, setLevels] = useState([]);
  const [zones, setZones] = useState([]);
  const [loading, setLoading] = useState(true);
```

(c) Stany komponentu ParkingContext

```
return (
  <AuthContext.Provider value={{ login, logout, userToken, id, checkToken }}>
    {children}
  </AuthContext.Provider>
)
```

(d) Przykład przekazania parametrów do komponentów wewnętrznych

Rysunek 3.3: Fragmenty kodu z komponentów kontekstowych

```
export const loginRegisterLogo = {
  flex: 1,
  backgroundColor: '#0092ca',
  textAlign: 'center',
  flexDirection: 'column',
  justifyContent: 'center',
  alignItems: 'center',
}
```

(a) Styl logo ze strony logowania oraz rejestracji

```
export const loginRegisterTextInput = {
  borderColor: '#0092ca',
  width: '80%',
  padding: 5,
  borderWidth: 2,
  borderRadius: 10,
}
```

(b) Styl pola tekstowego na stronie logowania oraz rejestracji

```
export const tabBar = {
  right: 20,
  left: 20,
  position: 'absolute',
  borderColor: '#393e46',
  borderWidth: 1,
  borderRadius: 10,
  bottom: 20,
  backgroundColor: '#393e46'
}
```

(c) Styl dolnego panelu do zmieniania ekranu

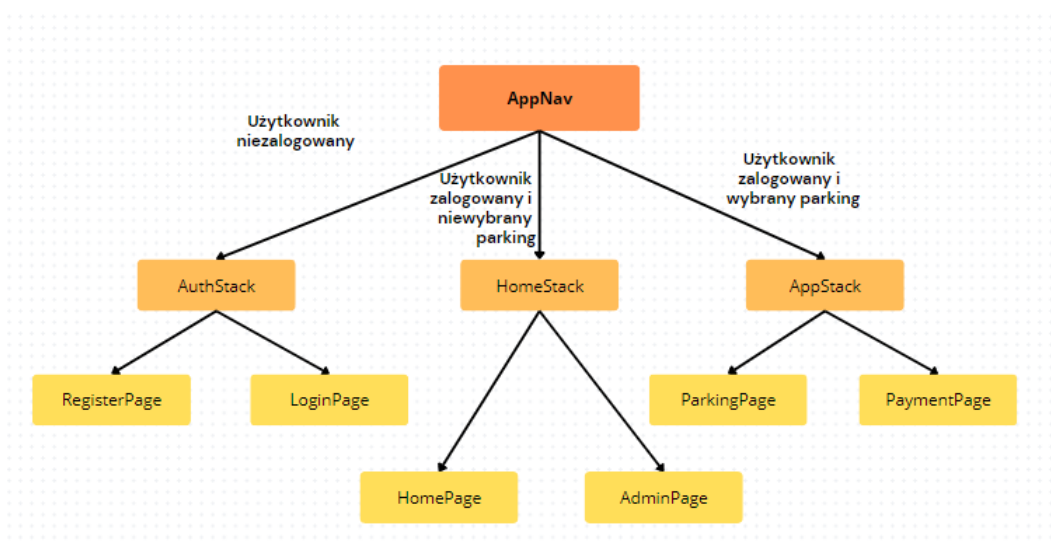
```
export const logoutButton = {
  height: '60%',
  width: '15%',
  backgroundColor: '#eeeeee',
  borderRadius: 10,
  justifyContent: 'center',
  position: 'absolute',
  right: 15
}
```

(d) Styl przycisku wylogowania

Rysunek 3.4: Przykładowe fragmenty kodu stylów komponentów

W aplikacji zostały użyte style które są obiektami języka JavaScript, które posiadają jako atrybuty cechy komponentów (rysunek 3.4). Style eksportowane są z plików zawartych w folderze `styles`, tworzone były w pliku z nazwą odpowiadającą ich przeznaczeniu, np. `layout` posiada style zajmujące się odpowiednim ułożeniem obiektów na ekranie, `parking` zawiera obiekty stylujące oraz układające schemat parkingu, a `buttons` zawiera style przycisków.

3.1.2 Nawigacja



Rysunek 3.5: Schemat nawigacji aplikacji

Głównym komponentem nawigacyjnym jest `AppNav`, uruchamia on jeden z 3 nawigatorów aplikacji w zależności od danych znajdujących się w `AsyncStorage` (rysunek 3.5). Jeśli `JSON Web Token` nie jest zapisany w `AsyncStorage` aplikacja uruchomi `AuthStack` (rysunek 3.6c), który jest nawigatorem stosu, składa się on ze strony do rejestracji `RegisterPage` oraz strony logowania `LoginPage`. Po zalogowaniu `JSON Web Token` zostaje zapisany w `AsyncStorage` i uruchamiany zostaje `HomeStack` (rysunek 3.6b), użyty w nim został nawigator dolnej karty, w którym są dwa ekrany `HomePage`, na którym można wybrać parking z którego chcemy skorzystać oraz `AdminPage`, tymczasowy panel administratora, na którym można dodać nowy parking. Po wybraniu parkingu jego nazwa również zostaje zapisana do `AsyncStorage`, aby po kolejnym uruchomieniu aplikacji `AppStack` uruchamiał się automatycznie. `AppStack` (rysunek 3.6a) również używa nawigatora dolnej karty do poruszania się między ekranami, którymi są `ParkingPage`, gdzie wyświetlony jest schemat parkingu oraz jest opcja wyboru swojego miejsca parkingowego, oraz jeśli parking jest płatny `PaymentPage`, gdzie można zarezerwować i zapłacić za miejsce na określoną ilość czasu. Aby móc zmienić stos komponentów zostały wprowadzone przyciski wylogowania oraz zmiany parkingu, które odpowiednio usuwają z `AsyncStorage` `JSON Web Token` oraz nazwę parkingu.

```
return (
  <Tab.Navigator screenOptions={({ route }) => ({
    tabBarIcon: ({ color }) => screenOptions(route, color),
    headerShown: false,
    tabBarStyle: tabBar,
    tabBarInactiveTintColor: 'eeeeee'
  }) >
    <Tab.Screen name="Parking" component={ParkingPage} />
    {isPaid ? <Tab.Screen name="Payment" component={PaymentPage} /> : null}
  </Tab.Navigator>
);
```

(a) Komponent nawigacyjny AppStack

```
return (
  <Tab.Navigator screenOptions={({ route }) => ({
    tabBarIcon: ({ color }) => screenOptions(route, color),
    headerShown: false,
    tabBarStyle: tabBar,
    tabBarInactiveTintColor: 'eeeeee'
  }) >
    <Tab.Screen name="Home" component={HomePage} />
    <Tab.Screen name="Add parking" component={AdminPage} />
  </Tab.Navigator>
);
```

(b) Komponent nawigacyjny HomeStack

```
return (
  <Stack.Navigator screenOptions={{ headerShown: false }}>
    <Stack.Screen name="Login" component={LoginPage} />
    <Stack.Screen name="Register" component={RegisterPage} />
  </Stack.Navigator>
);
```

(c) Komponent nawigacyjny AuthStack

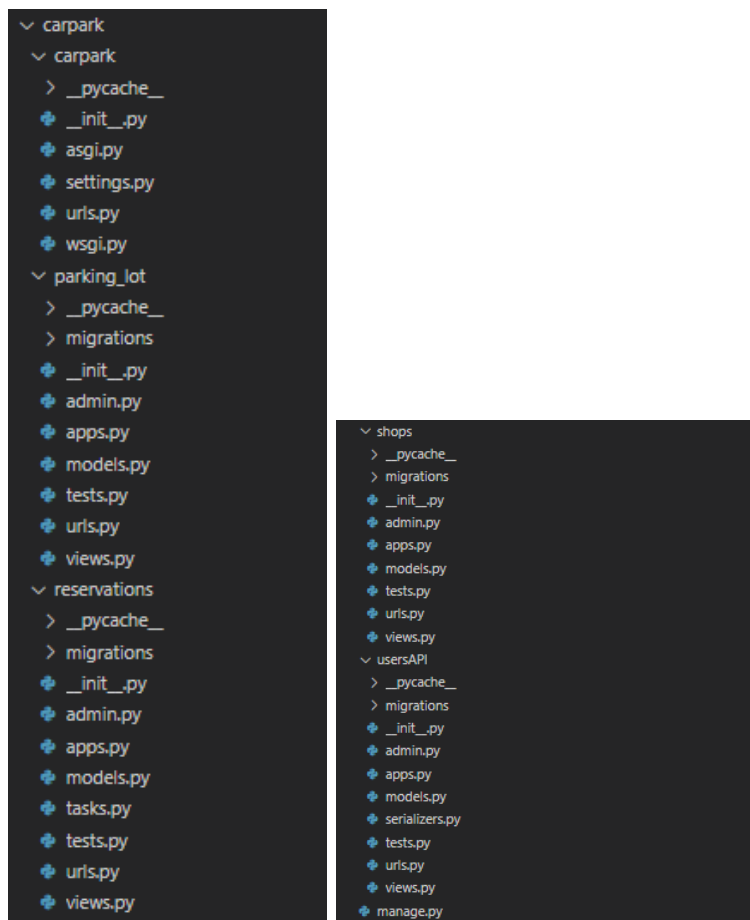
Rysunek 3.6: Kod komponentów nawigacyjnych

3.2 Serwer

Strona serwerowa aplikacji napisana jest w języku Python oraz frameworku Django, zostały również użyte dodatkowe biblioteki dla funkcjonalności niedostarczonych przez standard wymienionych technologii.

3.2.1 Struktura kodu i funkcjonalności

Rysunek 3.7 przedstawia strukturę plików strony serwerowej. Głównym plikiem uruchamiającym naszą aplikację Django jest plik `manage.py`, serwer uruchamiamy komendą `python manage.py runserver`, importowane są wtedy również ustawienia z pliku `settings.py` oraz uruchamiane są podrzędne moduły. Każdy folder wewnątrz zewnętrznego folderu `carpark` to osobny moduł. Wewnętrzny folder **carpark** jest głównym modułem projektu, jego zadaniem jest tworzenie architektury ustawień, importowanie pozostałych modułów oraz przydzielenie modułom odpowiednich endpointów (rysunek 3.8).



(a) pierwsza część

(b) druga część

Rysunek 3.7: Struktura plików serwera

```
from django.urls import path, include

urlpatterns = [
    path('api/users/', include('usersAPI.urls')),
    path('api/parking/', include('parking_lot.urls')),
    path('api/reservations/', include('reservations.urls')),
    path('api/shops/', include('shops.urls'))
]
```

Rysunek 3.8: Rutiny endpointów modułów serwera

Użytkownicy

Moduł **użytkownicy** definiuje model użytkownika w bazie danych (rysunek 3.9) oraz odpowiada wszystkim operacjom związanym z kontem użytkownika, obsługuje jedno z najważniejszych funkcji aplikacji, czyli rejestrację i logowanie, przy którym użytkownik uzyskuje parę tokenów, autoryzacji i odświeżenia. Zajmuje się on również weryfikacją wcześniej wspomnianych tokenów. Powyższe funkcjonalności z wyłączeniem rejestracji są zaprogramowane w bibliotece *rest_framework_simplejwt*, na serwerze wystarczy umieścić odniesienie do odpowiedniej klasy. Dodatkową funkcjonalnością modułu użytkownicy jest

możliwość uzyskania ID użytkownika. Poniższy fragment pliku `urls.py` (rysunek 3.10) tego modułu pokazuje URI węzłów końcowych tych funkcji.

```
class User(AbstractUser, PermissionsMixin):
    password = models.CharField(max_length=255)
    username = models.EmailField(max_length=255, unique=True)
```

Rysunek 3.9: Model użytkownika

```
urlpatterns = [
    path('register/', views.RegisterView.as_view(), name="user_register"),
    path('login/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('token/verify/', TokenVerifyView.as_view(), name='token_verify'),
    path('get_id/', views.get_id, name="user_get_id")
]
```

Rysunek 3.10: Nawigacja węzłów końcowych modułu użytkownicy

Parking

Głównym zadaniem modułu **parking** jest symulacja systemu inteligentnego parkingu, administrator może taki parking stworzyć (rysunek 3.12), podając parametry takie jak: nazwa, zajętość (dla sprawdzenia wszystkich możliwości testowych), ilość pięter oraz informację czy parking jest płatny (rysunek 3.11).

```
class Parking(models.Model):
    name = models.CharField(max_length=255, unique=True)
    is_paid = models.BooleanField()

class Level(models.Model):
    level_number = models.IntegerField()
    parking = models.ForeignKey(to=Parking, to_field='id', on_delete=models.CASCADE, default=None)

class Zone(models.Model):
    name = models.CharField(max_length=1)
    level = models.ForeignKey(to=Level, to_field='id', on_delete=models.CASCADE, default=None)

class Spot(models.Model):
    spot_number = models.IntegerField()
    is_taken = models.BooleanField(max_length=50)
    user_id = models.IntegerField(default=None, null=True)
    distance = models.FloatField(null=False)
    zone = models.ForeignKey(to=Zone, to_field='id', on_delete=models.CASCADE, default=None)
```

Rysunek 3.11: Model parkingu

```

try:
    parking = Parking.objects.create(name=name, is_paid=is_paid)
except:
    return JsonResponse({"err": "Parking with this name already exists"}, status=400)
for l in levels:
    level = Level.objects.create(level_number=l, parking=parking)
    for z in zones:
        zone = Zone.objects.create(name=z, level=level)
        spots = [{ 'spot_number': spot + 1, 'user_id': None, 'is_taken': True if random.random() <= occupancy else False } for spot in range
                    (number_of_spots_in_zone)]
        for s in spots:
            dist = distance[f'{l}{z}{s["spot_number"]}']
            spot = Spot.objects.create(spot_number=s['spot_number'], user_id=s['user_id'], is_taken=s['is_taken'], distance=dist, zone=zone)

```

Rysunek 3.12: Fragment funkcji tworzącej parking

Do obliczenia dystansu między wjazdem a miejscami został użyty algorytm Dijkstry (rysunek 3.13), wierzchołkami grafu były miejsca parkingowe, a do krawędzi były przypisane odległości (rysunek 3.14) między sąsiednimi miejscami, wjazdem na parking i wyjazdem na następne piętro. Do stworzenia grafu oraz algorytmu Dijkstry została użyta biblioteka *networkx*, zapewniająca różnorodne funkcje grafowe.

```
distances = nx.dijkstra_predecessor_and_distance(G, f'Entr1')[1]
```

Rysunek 3.13: Algorytm Dijkstry biblioteki networkx

```

def distance(G, first_node, second_node):
    first_node_pos = G.nodes[first_node]['pos']
    second_node_pos = G.nodes[second_node]['pos']
    dist = math.sqrt((first_node_pos[0]-second_node_pos[0])**2 + (first_node_pos[1]-second_node_pos[1])**2)
    return round(dist,2)

```

Rysunek 3.14: Funkcja obliczająca dystans między miejscami parkingowymi

Serwer udostępnia informacje o nazwach wszystkich parkingów, wybranym parkingu oraz parametrach wyboru miejsca parkingowego. Dzięki nim użytkownik może znaleźć miejsce parkingowe z wybranymi przez niego parametrami oraz zwolnić to miejsce (na potrzeby symulacji odjazdu). Na rysunku 3.15 przedstawiono punkty końcowe wszystkich powyższych funkcjonalności

```

urlpatterns = [
    path('create/', views.create, name="parking_create"),
    path('get_parking/', views.get_parking, name="get_parking"),
    path('get_names/', views.get_names, name="parking_get_names"),
    path('find_spot/', views.find_spot, name="parking_find_spot"),
    path('delete_spot/', views.delete_spot, name="parking_delete_spot"),
    path('get_parking_options/', views.get_parking_options, name="get_parking_options"),
]

```

Rysunek 3.15: Nawigacja endpointów modułu parking

Rezerwacje

Jest to moduł obsługujący rezerwacje użytkowników, definiuje on model rezerwacji oraz tworzy dwie tabele rezerwacje i zarchiwizowane rezerwacje (rysunek 3.16).

```

class Reservations(models.Model):
    start_date = models.DateTimeField()
    end_date = models.DateTimeField()
    user = models.ForeignKey(to=User, to_field='id', on_delete=models.CASCADE, default=None)
    parking = models.ForeignKey(to=Parking, to_field='id', on_delete=models.CASCADE, default=None)

class ArchivedReservations(models.Model):
    start_date = models.DateTimeField()
    end_date = models.DateTimeField()
    user = models.ForeignKey(to=User, to_field='id', on_delete=models.CASCADE, default=None)
    parking = models.ForeignKey(to=Parking, to_field='id', on_delete=models.CASCADE, default=None)

```

Rysunek 3.16: Modele rezerwacji

Funkcjonalnościami tego modułu są: stworzenie lub przedłużenie rezerwacji (rysunek 3.17), usunięcie rezerwacji oraz pobranie daty końcowej rezerwacji. Na rysunku 3.18 widać węzły końcowe tego modułu.

```

reservation = Reservations.objects.filter(user_id=user.id, parking_id=parking.id)
if reservation.exists():
    reservation = reservation.first()
    reservation.end_date += timedelta(hours=hours, minutes=minutes)
    reservation.save()
else:
    reservation = Reservations.objects.create(start_date=start_date, end_date=end_date, user=user, parking=parking)

```

Rysunek 3.17: Fragment funkcji tworzącej lub przedłużającej rezerwację

```

urlpatterns = [
    path('create/', views.create, name="reservation_create" ),
    path('cancel_reservation/', views.cancel_reservation, name="reservation_cancel"),
    path('get_reservation/', views.get_reservation, name="reservation_get"),
]

```

Rysunek 3.18: Nawigacja endpointów modułu rezerwacji

Dodatkowo w tym module znajduje się plik `tasks.py` z zadaniami działającymi w tle, który co sześćdziesiąt sekund przeszukuje bazę danych w celu znalezienia przedawnionych rezerwacji oraz je archiwizuje (rysunek 3.19).

```

@background(schedule=60)
def check_for_old_reservations():
    date = datetime.now()
    old_reservations = Reservations.objects.filter(end_date__lt=date)

    if old_reservations.exists():
        for old_reservation in old_reservations:
            user = User.objects.get(id=old_reservation.user_id)
            parking = Parking.objects.get(id=old_reservation.parking_id)
            ArchivedReservations.objects.create(start_date = old_reservation.start_date, end_date = old_reservation.end_date, user=user, parking=parking)
            old_reservation.delete()

```

Rysunek 3.19: Zadanie wykonujące się cyklicznie po uruchomieniu serwera

Sklepy

Jedynymi zadaniami modułu **sklepy** jest definiowanie modelu sklepu (rysunek 3.20) oraz możliwość tworzenia sklepów (rysunek 3.21).

```
class Shops(models.Model):
    name = models.CharField(max_length=255)
    zone = models.CharField(max_length=1)
```

Rysunek 3.20: Model sklepu

```
def create(request):
    if request.method == "POST":
        flag = False
        shops = json.loads(request.body.decode('utf-8'))['shops']
        for shop in shops:
            shop_exists = Shops.objects.filter(name=shop['name'])
            if not shop_exists:
                Shops.objects.create(name = shop['name'], zone = shop['zone'])
            else:
                flag = True

        if not flag:
            return JsonResponse({})
        else:
            return JsonResponse({"err": "one or more shops already exist"}, status = 400)
```

Rysunek 3.21: Funkcja tworząca sklepy

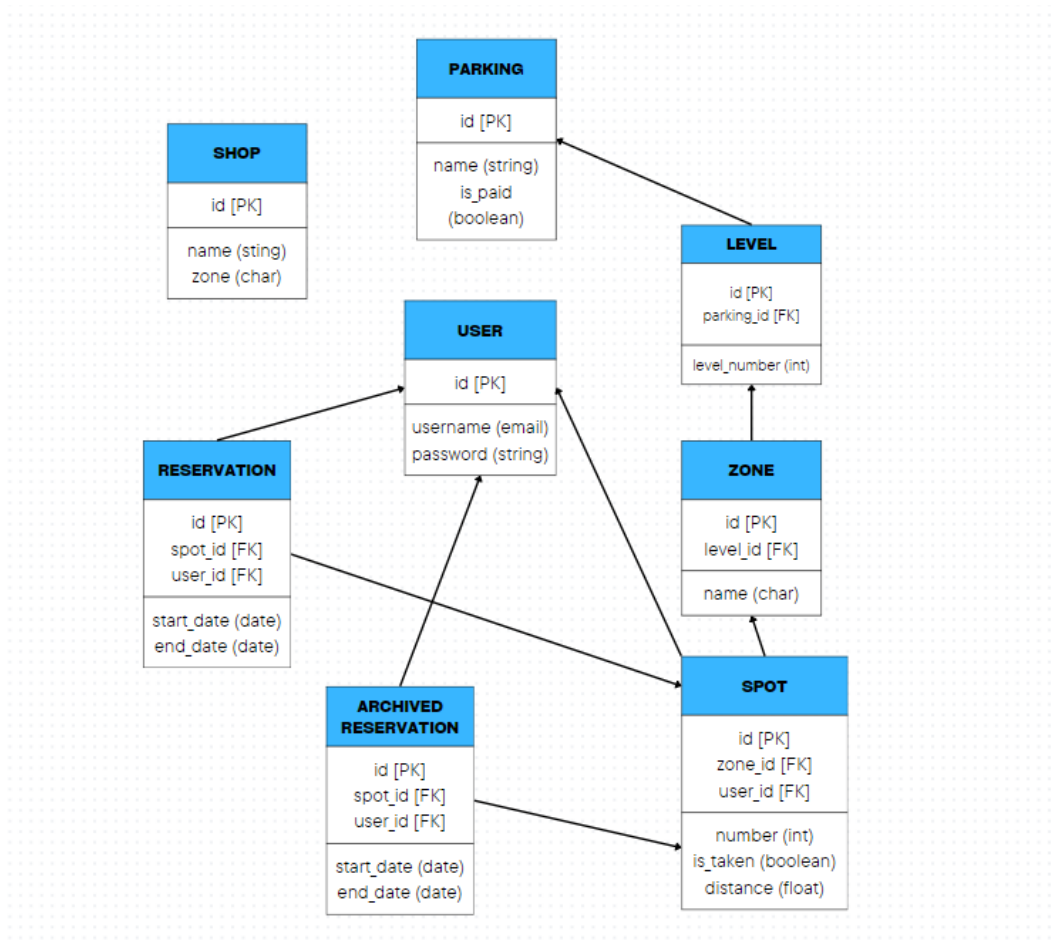
3.3 Baza danych

Baza danych została stworzona na serwerze MySQL za pomocą mapowania obiektowo-relacyjnego, modele zostały zdefiniowane w modułach po stronie serwera i później zmapowane do tabel w MySQL. Połączenie między serwerem a bazą danych wykonuje biblioteka *mysqlclient* oraz parametry połączenia zdefiniowane są w pliku *settings.py* (rysunek 3.22).

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'CarPark',
        'USER': 'root',
        'PASSWORD': 'admin',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

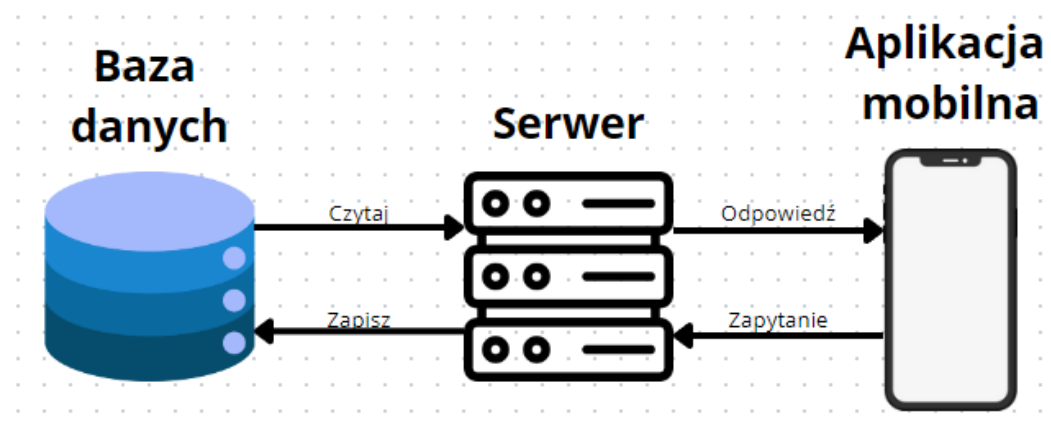
Rysunek 3.22: Parametry połączenia między serwerem a bazą danych

Jedynym rodzajem relacji występującym w bazie danych są połączenia jeden do wielu. Występują one między np. parkingiem a piętrami, piętrami a strefami oraz strefami a miejscami parkingowymi. Schemat bazy danych został przedstawiony na poniższym diagramie ERD (rysunek 3.23).



Rysunek 3.23: Diagram ERD bazy danych

3.4 Przepływ danych



Rysunek 3.24: Schemat przepływu danych w projekcie

Przepływ danych w projekcie jest trójwarstwowy, dane przechowywane są w bazie danych. Serwer może zapisywać lub odczytywać informację z bazy danych. Dane przechodzące między bazą danych a serwerem są przesyłane za pomocą obiektów `QuerrySet` dostępnych w środowisku Django.

Połączenie między serwerem a aplikacją mobilną polega na wysyłaniu zapytań oraz odpowiedzi, są one wysyłane protokołem HTTP. Obiekty wewnątrz zapytań oraz odpowiedzi są formatu JSON. Na rysunku 3.24 przedstawiony jest schemat przepływu danych w projekcie.

Wszystkie zapytania aplikacji wysyłane na serwer, poprzedzone są zapytaniem na węzeł końcowy sprawdzający ważność *JSON Web Tokenu*, jeśli ten token nie zostanie poprawnie autoryzowany, zostaje wysłane kolejne zapytanie na punkt końcowy zajmujący się odświeżaniem tokenów. W sytuacji, kiedy token odświeżający również stracił ważność, system wylogowuje użytkownika z aplikacji (rysunek 3.25).

```
const checkToken = async () => {  
  accessToken = await AsyncStorage.getItem("userToken");  
  
  const params = {  
    method: 'POST',  
    headers: { 'Content-Type': 'application/json' },  
    body: JSON.stringify({  
      token: accessToken  
    })  
  };  
  
  await fetch(`${BASE_URL}/api/users/token/verify/`, params)  
    .then(response => {  
      if (response.ok) {  
        return;  
      }  
      else {  
        refreshToken = AsyncStorage.getItem("refreshToken")  
        .then(refreshToken) => {  
          params['body'] = JSON.stringify({  
            refresh: refreshToken  
          });  
          fetch(`${BASE_URL}/api/users/token/refresh/`, params)  
            .then(response => {  
              if (response.ok) {  
                response.json()  
                  .then(data => {  
                    setUserToken(data["access"])  
                    AsyncStorage.setItem("userToken", data["access"]);  
                    return;  
                  })  
              }  
              else {  
                response.json()  
                  .then(data => {  
                    console.log(data);  
                    logout();  
                    return false;  
                  })  
              }  
            })  
          }  
        })  
      }  
    })  
  ).catch(error => console.error(error));  
}
```

Rysunek 3.25: Funkcja wysyłająca zapytania sprawdzające JSON Web Token

Rozdział 4

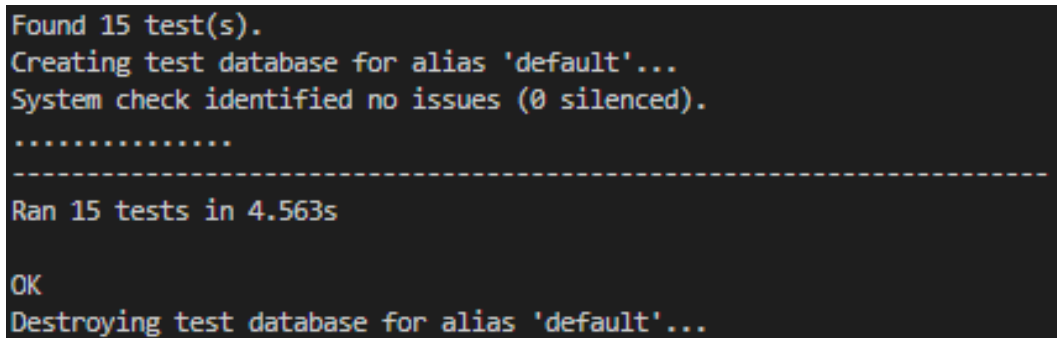
Testowanie

4.1 Testy aplikacji

Po stronie aplikacji zostały przeprowadzone testy manualne, pod uwagę zostały wzięte wszystkie scenariusze główne oraz alternatywne.

4.2 Testy serwera

Po stronie serwera zostały przeprowadzone testy integracyjne (rysunek 4.1), czyli został przetestowany każdy węzeł końcowy serwera pod względem zwracanego statusu oraz zwracanych danych. W przypadkach gdy dane zwracane przez endpoint są generowane z serwera np. zajęcie miejsca parkingowego, sprawdzane są klucze zawarte w zwracanej wartości. Do przeprowadzenia testów została użyta biblioteka Django test, wbudowana w standard frameworku.



```
Found 15 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
-----
Ran 15 tests in 4.563s

OK
Destroying test database for alias 'default'...
```

Rysunek 4.1: Wynik przeprowadzonych testów

4.2.1 Parking

- Test tworzenia parkingu - test został podzielony na dwa przypadki, kiedy parking o takiej nazwie nie istniał oraz kiedy parking o takiej nazwie już istniał (rysunek 4.2).
- Test zwracania parkingu - przetestowane zostały klucze wewnętrznych elementów parkingu, takie jak: piętra, strefy i miejsca parkingowe, oraz wartości atrybutów parkingu (rysunek 4.3).

- Test zwracania nazw parkingów - przetestowane zostały wartości odpowiadające nazwą parkingu.
- Test szukania miejsca - przetestowane zostały cztery przypadki, dwa z nich dla parkingu, który nie posiada wolnych miejsc parkingowych oraz dwa dla parkingu z wolnymi miejscami. W obu przypadkach dostarczane były informacje o wybranej strefie lub sklepie.
- Test odstąpienia miejsca - przetestowane zostały dwa przypadki, jeden, w którym użytkownik nie miał zajętego miejsca oraz drugi, w którym użytkownik miał zajęte miejsce.
- Test zwracania opcji wybierania miejsca parkingowego - przetestowane zostały klucze zawarte w zwracanych danych.

```
def test_create(self):
    data = json.dumps({"levels": 3, "name": "create_test", "occupacy": "25%", "isPaid": True})

    # test poprawnej nazwy parkingu
    response = self.client.post(reverse('parking_create'), data, content_type=self.content_type)
    self.assertEqual(response.status_code, 200)

    # test zajętej nazwy parkingu
    response = self.client.post(reverse('parking_create'), data, content_type=self.content_type)
    self.assertEqual(response.status_code, 400)
    self.assertEqual(response.json()['err'], "Parking with this name already exists")
```

Rysunek 4.2: Kod testu tworzenia parkingu

```
def test_get_parking(self):
    data = json.dumps({"name": "test"})

    # sprawdzenie poprawności wartości lub kluczy całego parkingu
    response = self.client.post(reverse('get_parking'), data, content_type=self.content_type)

    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.json()['name'], 'test')
    self.assertEqual(response.json()['is_paid'], True)
    self.assertEqual(len(response.json()['levels']), 3)
    for l in range(len(response.json()['levels'])):
        self.assertTrue('level' in response.json()['levels'][l])
        self.assertTrue('zones' in response.json()['levels'][l])
        for z in range(len(response.json()['levels'][l]['zones'])):
            self.assertTrue('zone' in response.json()['levels'][l]['zones'][z])
            self.assertTrue('spots' in response.json()['levels'][l]['zones'][z])
            for s in range(len(response.json()['levels'][l]['zones'][z]['spots'])):
                self.assertTrue('is_taken' in response.json()['levels'][l]['zones'][z]['spots'][s])
                self.assertTrue('number' in response.json()['levels'][l]['zones'][z]['spots'][s])
                self.assertTrue('user_id' in response.json()['levels'][l]['zones'][z]['spots'][s])
```

Rysunek 4.3: Kod testu zwracania parkingu

4.2.2 Użytkownicy

- Test rejestracji - test podzielono na dwa przypadki, kiedy rejestruje się użytkownik z email'em użytym poprzednio w aplikacji oraz z nowym email'em (rysunek 4.4).

- Test logowania - przetestowane zostały przypadki, kiedy użytkownik podaje poprawne hasło i login, oraz kiedy dane te są niepoprawne (rysunek 4.5).
- Test odświeżenia tokenu - test został podzielony na dwa przypadki, kiedy token odświeżający jest poprawny lub nie.
- Test weryfikacji tokenu - test został podzielony na dwa przypadki, kiedy token autoryzacji jest poprawny lub nie.
- Test zwracania id - przetestowano poprawność zwracanego id użytkownika

```
def test_register(self):
    data = json.dumps({"username": "test@test.com", "password": "test_password"})

    # test rejestracji
    response = self.client.post(reverse('user_register'), data, content_type=self.content_type)
    self.assertEqual(response.status_code, 200)

    # test ponownej rejestracji
    response = self.client.post(reverse('user_register'), data, content_type=self.content_type)
    self.assertEqual(response.status_code, 400)
    self.assertEqual(response.json()['username'], 'user with this username already exists.')
```

Rysunek 4.4: Kod testu rejestracji

```
def test_login(self):
    # test logowania ze złymi danymi
    wrong_data = json.dumps({"username": "wrong@test.com", "password": "test"})

    response = self.client.post(reverse('token_obtain_pair'), wrong_data, content_type=self.content_type)

    self.assertEqual(response.status_code, 401)
    self.assertEqual(response.json()['detail'], 'No active account found with the given credentials')

    # test poprawnego logowania
    response = self.client.post(reverse('token_obtain_pair'), self.data, content_type=self.content_type)

    self.assertTrue('access' in response.json())
    self.assertTrue('refresh' in response.json())
```

Rysunek 4.5: Kod testu logowania

4.2.3 Rezerwacje

Testowane czasy rezerwacji były zaokrąglone do sekund ze względu na czas wykonywania operacji.

- Test tworzenia lub przedłużenia rezerwacji - przetestowano przypadki kiedy rezerwacja istnieje i zostaje przedłużona oraz kiedy rezerwacja nie istnieje (rysunek 4.6).
- Test usuwania rezerwacji - przetestowano poprawność usuwania rezerwacji.
- Test zwracania daty końcowej rezerwacji - przetestowano poprawność zwracanego czasu oraz przypadek kiedy rezerwacja nie istnieje (rysunek 4.7).

```

def test_create(self):
    data = json.dumps({"hours" : "1", "minutes" : "1", "parkingName": self.parking.name, "userId": self.user.id})

    response = self.client.post(reverse('reservation_create'), data, content_type=self.content_type)
    end_date = (datetime.now() + timedelta(hours=1, minutes=1)).strftime('%Y-%m-%dT%H:%M:%S.%f')

    # sprawdzanie poprawności tworzenia rezerwacji
    self.assertEqual(response.status_code, 200)
    self.assertEqual(re.sub(r'\.*', '', response.json()['end_date']), re.sub(r'\.*', '', end_date))

    reservation = Reservations.objects.get(parking_id=self.parking.id, user_id=self.user.id)

    # sprawdzanie poprawności przedłużenia rezerwacji
    response = self.client.post(reverse('reservation_create'), data, content_type=self.content_type)
    end_date = (reservation.end_date + timedelta(hours=1, minutes=1)).strftime('%Y-%m-%dT%H:%M:%S.%f')

    self.assertEqual(response.status_code, 200)
    self.assertEqual(re.sub(r'\.*', '', response.json()['end_date']), re.sub(r'\.*', '', end_date))

```

Rysunek 4.6: Kod testu tworzenia lub przedłużenia rezerwacji

```

def test_get_reservation(self):
    data = json.dumps({"parkingName": self.parking.name, "userId": self.user.id})

    # test bez rezerwacji
    response = self.client.post(reverse('reservation_get'), data, content_type=self.content_type)

    self.assertEqual(response.status_code, 400)
    self.assertEqual(response.json()['end_date'], "doesn't exist")

    start_date = datetime.now()
    end_date = start_date + timedelta(hours=1)
    reservation = Reservations.objects.create(user=self.user, parking=self.parking, end_date=end_date, start_date=start_date)

    # test posiadanej rezerwacji
    end_date = reservation.end_date.strftime('%Y-%m-%dT%H:%M:%S.%f')
    response = self.client.post(reverse('reservation_get'), data, content_type=self.content_type)

    self.assertEqual(response.status_code, 200)
    self.assertEqual(re.sub(r'\.*', '', response.json()['end_date']), re.sub(r'\.*', '', end_date))

```

Rysunek 4.7: Kod testu zwracania daty końcowej rezerwacji

4.2.4 Sklepy

W module sklepów został stworzony test tworzenia, został podzielony na dwa przypadki. W jednym sklepy nie były tworzone na serwerze, a w drugim jeden lub więcej sklepów z listy były już dodane (rysunek 4.8).

```

class TestShop(TestCase):
    def setUp(self):
        self.shops = {"shops": [{"name": "Adidas", "zone": "A"}, {"name": "Nike", "zone": "B"}]}
        self.content_type = 'application/json'

    def test_create(self):
        data = json.dumps(self.shops)

        # test dodawania nie istniejących sklepów
        response = self.client.post(reverse('shop_create'), data, content_type=self.content_type)
        self.assertEqual(response.status_code, 200)

        # test dodawania istniejących sklepów
        response = self.client.post(reverse('shop_create'), data, content_type=self.content_type)
        self.assertEqual(response.status_code, 400)
        self.assertEqual(response.json()['err'], 'one or more shops already exist')

```

Rysunek 4.8: Kod testu tworzenia sklepów

Rozdział 5

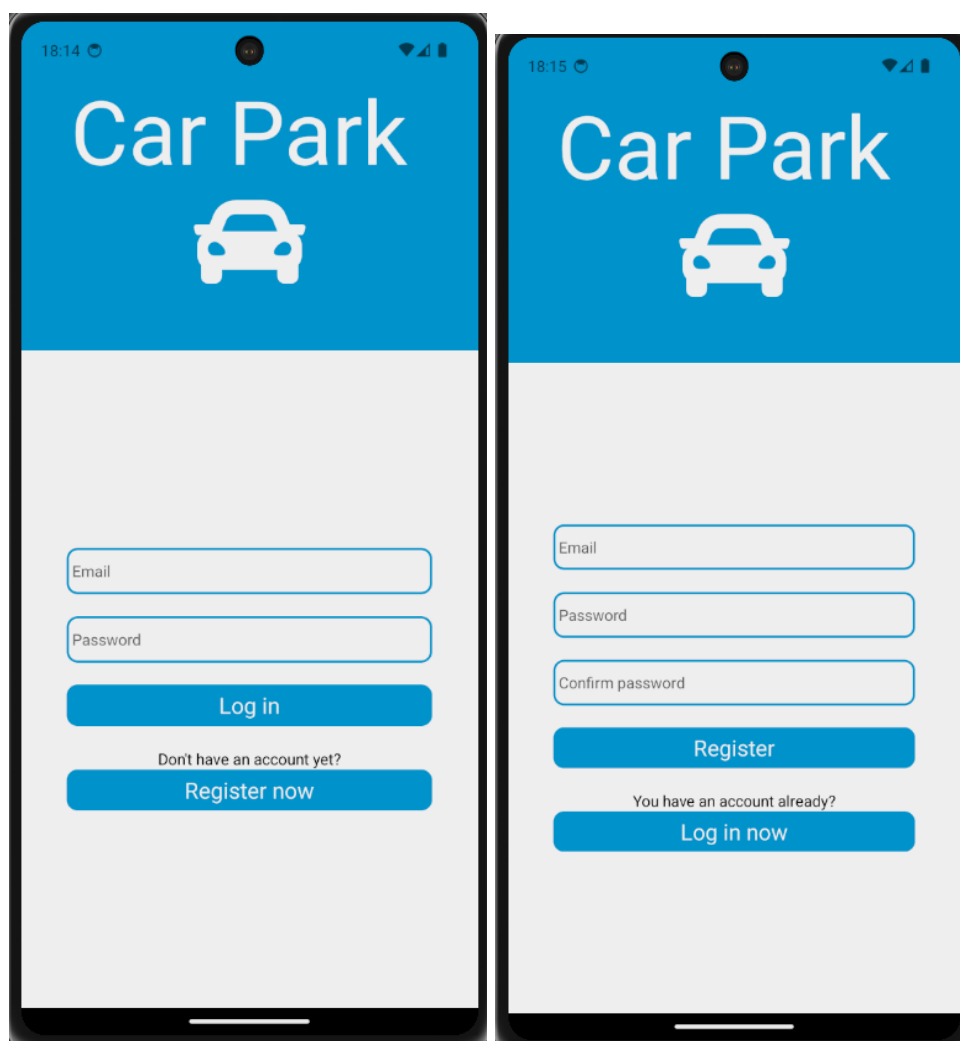
Dokumentacja użytkownika

Dokumentację użytkownika podzielono na trzy części:

- Rejestracja i logowanie.
- Wybór parkingu oraz panel administratora.
- Ekran parkingu oraz płatności.

5.1 Rejestracja i logowanie

Pierwszym ekranem, który po uruchomieniu aplikacji widzi użytkownik, jest ekran logowania (rysunek 5.1a), na tym ekranie użytkownik może się zalogować, wpisując email oraz hasło założonego konta. Na ekranie logowania widnieje również przycisk pozwalający na przejście do ekranu rejestracji (rysunek 5.1b). Użytkownik może się zarejestrować, używając adresu email oraz wpisując swoje hasło. Przejście między ekranami polega na kliknięciu przycisku **Register now** na stronie logowania lub **Log in now** na stronie rejestracji.

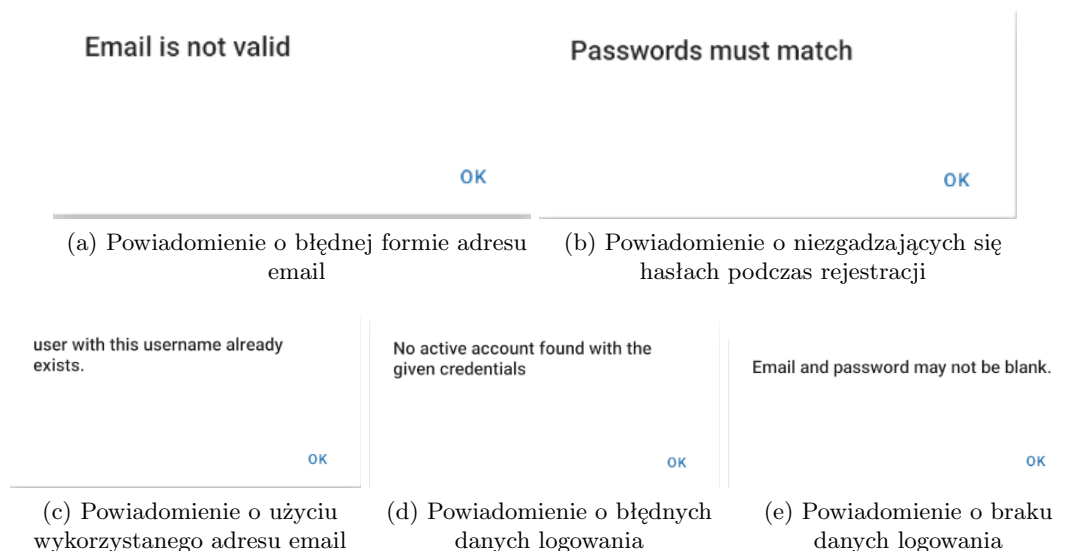


(a) Ekran logowania

(b) Ekran rejestracji

Rysunek 5.1: Ekrany logowania i rejestracji

Jeśli użytkownik popełni jakiś błąd, aplikacja wyświetla mu stosowny komunikat opisujący problem. Kiedy użytkownik rejestruje się, podając nieprawidłową formę adresu email, wyświetla się komunikat z rysunku 5.2a. Jeśli hasła użytkownika podczas rejestracji nie zgadzają się ze sobą, wyświetla się komunikat przedstawiony na rysunku 5.2b. W wypadku kiedy użytkownik rejestruje się adresem email użytym już w aplikacji, pojawia się komunikat z rysunku 5.2c. Jeśli użytkownik próbuje się zalogować, podając niepoprawne dane, pojawia się komunikat z rysunku 5.2d oraz kiedy użytkownik nie wpisze danych logowania, wyświetla się komunikat pokazany na rysunku 5.2e.



Rysunek 5.2: Komunikaty o błędzie wyświetlane przez aplikację na ekranach logowania i rejestracji

5.2 Wybór parkingu oraz panel administratora

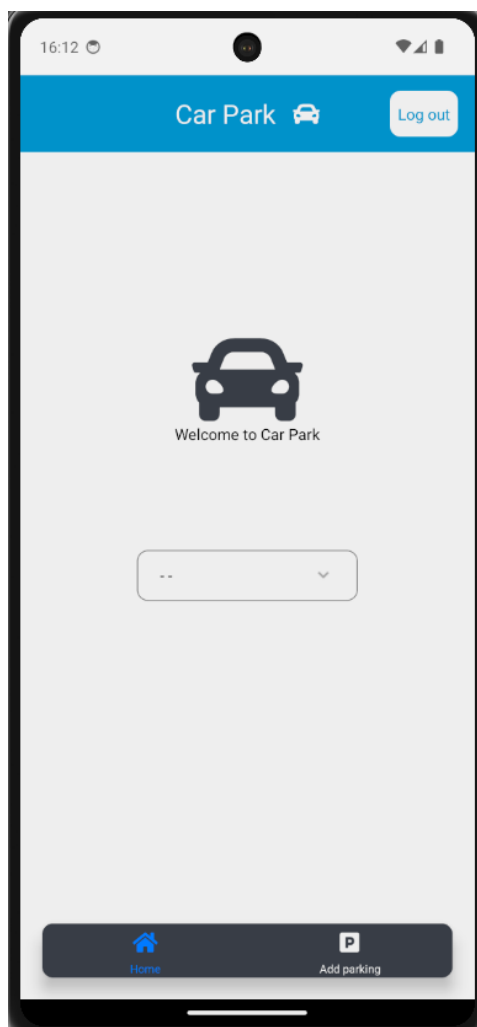
Pierwszym ekranem wyświetlającym się po zalogowaniu jest ekran wyboru parkingu (jeśli parking nie został wybrany podczas ostatniego uruchomienia aplikacji). Pomiędzy ekranami wyboru parkingu oraz panelu administratora poruszamy się przy pomocy nawigatora dolnej karty (rysunek 5.3).



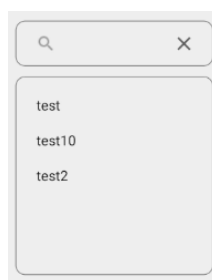
Rysunek 5.3: Nawigacja między ekranami wyboru parkingu oraz panelu administratora

5.2.1 Ekran wyboru parkingu

Ten ekran (rysunek 5.4) służy klientowi jako miejsce wyboru parkingu, na którym chce zaparkować. Ekran prezentuje listę wyboru parkingów dostępnych w systemie (rysunek 5.5), po wyborze jednego z nich aplikacja przenosi użytkownika na stronę parkingu.



Rysunek 5.4: Ekran wyboru parkingu

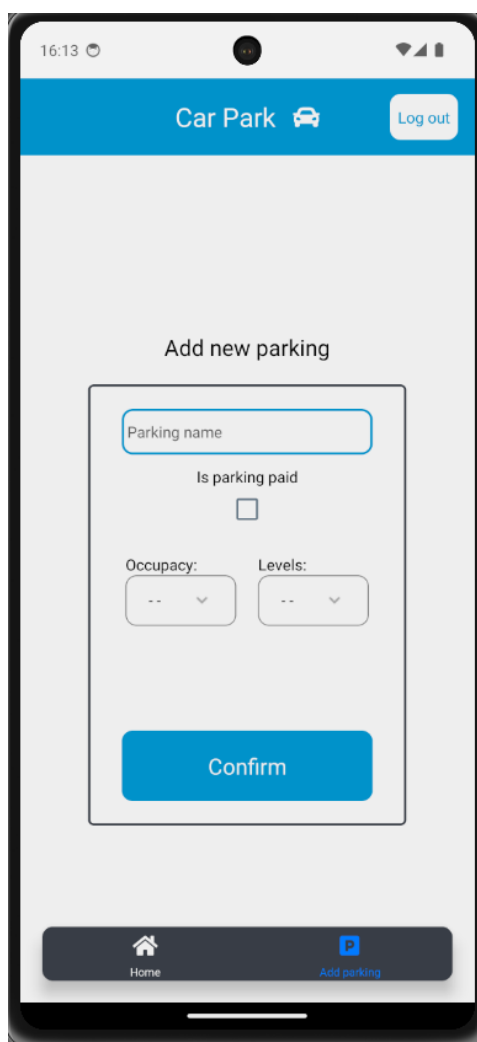


Rysunek 5.5: Lista wyboru parkingów w aplikacji

Na tej stronie można również wylogować się z aplikacji, używając przycisku **Log out** na górnym panelu lub zmienić go na ekran dodawania parkingu używając nawigacji na dole ekranu.

5.2.2 Panel administratora

Tymczasowy panel administratora (rysunek 5.6) został umieszczony w aplikacji, na tym panelu można dodać do aplikacji nowy parking. Dodanie nowego parkingu wymaga ustawienia jego nazwy, ilości pięter, informacji czy parking jest płatny oraz procenta zajętości miejsc parkingowych (potrzebne do przetestowania wszystkich alternatywnych scenariuszy).

The image shows a mobile application interface for managing parking spaces. At the top, there is a blue header bar with the text "Car Park" and a car icon, and a "Log out" button. Below the header, the main content area is titled "Add new parking". Inside this area, there is a form with several fields: a text input for "Parking name", a checkbox for "Is parking paid", two dropdown menus for "Occupacy:" and "Levels:", and a blue "Confirm" button at the bottom. The bottom of the screen features a dark navigation bar with two icons: a home icon labeled "Home" and a parking 'P' icon labeled "Add parking".

Rysunek 5.6: Ekran panelu administratora

Jeśli tworzonemu parkingowi nadamy nazwę, która już istnieje w systemie, aplikacja wyświetli komunikat z rysunku 5.7a lub jeśli nie wypełnimy wszystkich potrzebnych informacji, wystąpi powiadomienie z rysunku 5.7b.

Parking with this name already exists

Values can't be empty

OK

OK

(a) Powiadomienie o użyciu zajętej
nazwy parkingu

(b) Powiadomienie o niewypełnieniu
formularza

Rysunek 5.7: Komunikaty o błędzie wyświetlane przez aplikację na panelu administratora

5.3 Ekran parkingu oraz płatności

Poruszanie się między ekranem parkingu oraz płatności obsługuje nawigатор dołu karty przedstawiony na rysunku 5.8, jeśli wybrany przez użytkownika parking jest bezpłatny, pole to wygląda jak na rysunku 5.9.

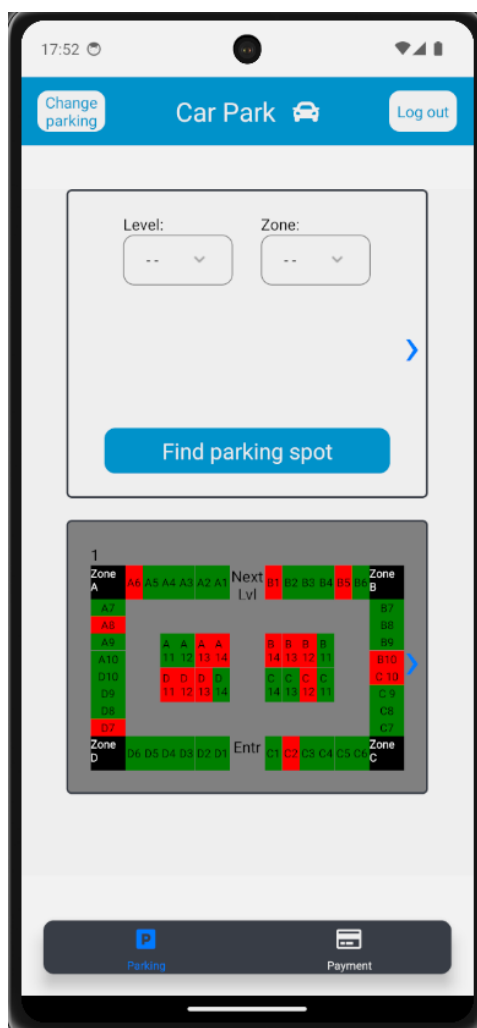


Rysunek 5.8: Nawigacja między ekranami parkingu oraz płatności



Rysunek 5.9: Nawigacja jeśli parking jest bezpłatny

5.3.1 Ekran parkingu



Rysunek 5.10: Ekran parkingu

Na ekranie parkingu (rysunek 5.10) wyświetlony jest schemat parkingu oraz pole z wyborem parametrów parkowania. Na schemacie parkingu miejsca zaznaczone kolorem zielonym są miejscami wolnymi oraz miejsca koloru czerwonego są miejscami zajęтыми, miejsca przypisane użytkownikowi jest koloru złotego (rysunek 5.11). Po kliknięciu strzałki na schemacie parkingu zostaje pokazany schemat następnego piętra parkingu (rysunek 5.11). Piętro, na którym się znajdujemy, pokazane jest w lewym górnym rogu parkingu.



Rysunek 5.11: Schemat parkingu na drugim piętrze, miejsce użytkownika A14

Panel wyboru miejsca parkowania również udostępnia wybór różnych parametrów, użytkownik może wybrać piętro oraz strefę lub piętro i sklep, w pobliżu którego chce zaparkować (rysunek 5.12).

Panel wyboru miejsca parkowania. (a) Pierwsza strona wyboru miejsca: Level: --, Zone: --, Find parking spot. (b) Druga strona wyboru miejsca: Level: --, Shop: --, Find parking spot.

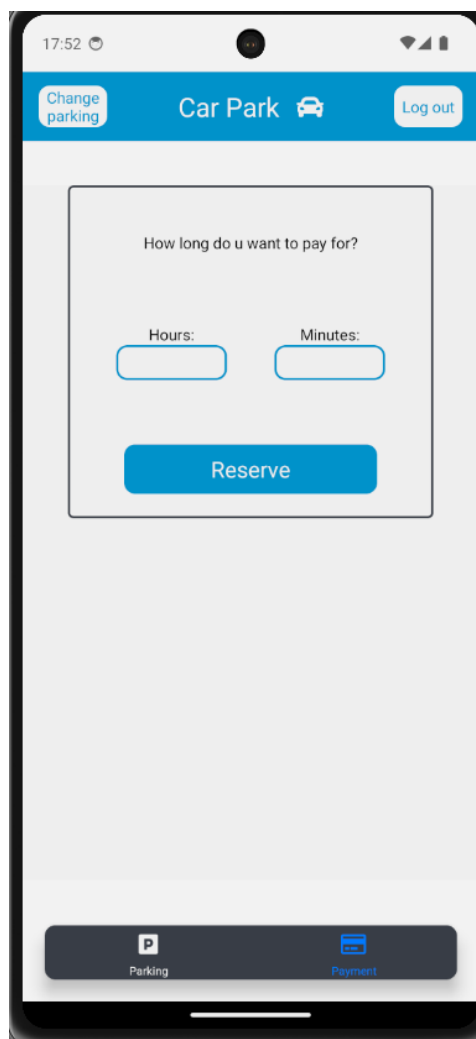
Rysunek 5.12: Panel wyboru opcji miejsca

Jeśli użytkownik nie wybierze żadnej z opcji, szukane jest miejsce najbliższe do wjazdu. Możliwe jest również wybranie jednego z parametrów, przy wyborze piętra system przeszuka to piętro, znajdując najbliższe miejsce od wjazdu na to piętro. W wypadku kiedy niewybrane jest piętro, ale wybrana jest strefa lub sklep, system przeszuka wybrany obszar na wszystkich piętrach oraz znajdzie najbliższe miejsce. Przy wyborze strefy lub sklepu oraz piętra, system wyszuka miejsce w wybranej strefie oraz na wybranym obszarze. W wypadku kiedy na parkingu nie znaleziono wolnego miejsca o podanych parametrach, aplikacja wyświetli komunikat z rysunku 5.13.



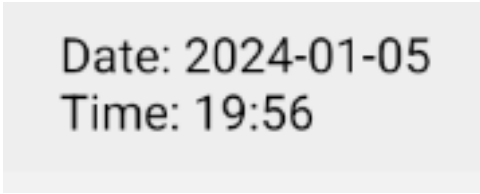
Rysunek 5.13: Powiadomienie o braku wolnych miejsc o wybranych parametrach

5.3.2 Ekran płatności



Rysunek 5.14: Ekran płatności

Ekran płatności (rysunek 5.14) dostępny jest jedynie w momencie kiedy parking wybrany przez użytkownika jest parkingiem płatnym. W panelu rezerwacji miejsca możemy wpisać czas, na jaki chcemy zarezerwować parking. Musi być wypełnione co najmniej jedno z pól, godziny lub minuty. Po zarezerwowaniu miejsca użytkownik może przedłużyć okres rezerwacji, powtarzając operację płatności na wybraną ilość czasu. Po opłaceniu miejsca pod tym panelem wyświetla się czas końca rezerwacji (rysunek 5.15).



Date: 2024-01-05
Time: 19:56

Rysunek 5.15: Wyświetlany czas daty końcowej płatności

Przy zmianie parkingu dostępnej w lewym górnym rogu ekranów płatności i parkingu, wybrane miejsce parkingowe oraz rezerwacja miejsca zostają usunięte.

Rozdział 6

Podsumowanie

6.1 Wnioski

Głównym celem aplikacji było stworzenie przejrzystego oraz intuicyjnego interfejsu aplikacji parkowania. Dzięki prostocie aplikacji użytkownik w szybki sposób może znaleźć odpowiadające mu miejsce parkingowe w pobliżu miejsca, które go interesuje. Dodatkowo, w przypadku płatnych parkingów, moduł rezerwacji upraszcza płacenie do wybrania czasu parkowania, oraz oszczędza czas potrzebny na znalezienie automatu.

Jednym z celów aplikacji było stworzenie powiadomień o kończącym się czasie parkowania, jednakże z powodów problemów połączenia między komputerem a urządzeniem mobilnym oraz ograniczoną ilość czasu ten cel nie został zrealizowany.

Aplikacja jest wersją demo, która powinna zostać połączona z osobnymi systemami automatyzacji parkingów obsługującymi czujniki i inne urządzenia monitorujące stan miejsc postojowych. Dzięki olbrzymim możliwościom technologicznym środowisk React, React Native oraz Django aplikacja może zostać wzbogacona o dodatkowe funkcjonalności.

6.2 Perspektywy dalszego rozwoju

Głównym założeniem dalszego rozwoju aplikacji jest zastąpienie strony serwerowej odpowiedzialnej za symulację parkingu, realnym systemem inteligentnego parkingu. Aby było to możliwe, trzeba również zamienić statyczne rysowanie przykładowego schematu parkingu, do rysowania parkingu można by wykorzystać bibliotekę **Canvas** języka JavaScript, używając planów technicznych do znalezienia współrzędnych miejsc parkingowych oraz przejazdów.

Jedną z ważnych funkcji aplikacji jest płatność za rezerwację miejsca, w przyszłości w aplikacji dodane zostałyby możliwości płatności elektronicznych, takie jak podłączenie karty płatniczej, PayPal lub Blik.

Kolejnym punktem rozwoju aplikacji jest stworzenie konta administratora. Panel administratora zostałby usunięty z aplikacji użytkownika oraz stworzony zostałby niezależny interfejs stworzony na przeglądarki internetowe. Do panelu administratora zostałyby dodana funkcjonalność przypisywania sklepów do parkingu oraz strefy.

W aplikacji można dodać również profil użytkownika, w którym byłoby możliwe dodanie swoich danych, informacji o używanych pojazdach lub numerze rejestracyjnym

oraz wybranie metody płatności, ponadto przy rejestracji można dodać wysyłanie maila potwierdzającego na pocztę.

W ramach rozwoju pracy dodane zostałyby również powiadomienia push wysyłające informacje o kończącym się czasie parkowania, użytą do tego byłaby biblioteka **expo-notifications**.

Rozdział 7

Bibliografia

1. Ile pojazdów (samochodów osobowych, ciężarowych, motocykli) jest w Polsce? <https://www.ciekawestatystyki.pl/2023/09/ile-pojazdow-jest-w-polsce.html>, dostęp w dniu 22.12.2023.
2. Galerie handlowe. Obroty i liczba odwiedzających w górę. Dawid Kuciński https://biznes.lovekrakow.pl/aktualnosci/galerie-handlowe-obroty-i-liczba-odwiedzajacych-w-gore_51679.html, dostęp w dniu 22.12.2023.
3. PRCH: Obroty najemców w centrach handlowych w czerwcu 2023 r. były o 7,7% wyższe niż w czerwcu 2022 r. <https://retailnet.pl/2023/08/25/prch-obroty-najemcow-w-centrach-handlowych-w-czerwcu-2023-r-byly-o-77-wyzsze-niz-w-czerwcu-2022-r/>, dostęp w dniu 22.12.2023.
4. Rynek obiektów handlowych w Polsce - raport za I półrocze 2023 r. <https://prch.org.pl/bazawiedzy/raporty/?start=1>, dostęp w dniu 22.12.2023.
5. Inteligentne systemy parkingowe – strumień przestrzenny czujników IoT <https://mikrokontroler.pl/2021/06/07/inteligentne-systemy-parkingowe-strumien-przestrzenny-czujnikow-iot/>, dostęp w dniu 27.12.2023.
6. MSR-Traffic system zajętości miejsc dla parkingów zewnętrznych <https://detektory.pl/system-zajetosci-miejsc-parking-zewnetrzny>, dostęp w dniu 27.12.2023.
7. Wygodne parkowanie za pomocą telefonu. Wypróbuj mobiParking. <https://www.skycash.com/parkowanie-przez-telefon/>, dostęp w dniu 27.12.2023.
8. Co to jest JavaScript? <https://widoczni.com/blog/najwiekszy-slownik-marketingu-internetowego-w-polsce/javascript/>, dostęp w dniu 31.12.2023
9. React <https://react.dev/>, dostęp w dniu 28.12.2023.
10. React Native <https://reactnative.dev/>, dostęp w dniu 28.12.2023.
11. Expo <https://expo.dev/>, dostęp w dniu 28.12.2023.
12. React Navigation <https://reactnavigation.org/docs>, dostęp w dniu 28.12.2023.
13. React Native Screens <https://www.npmjs.com/package/react-native-screens?activeTab=readme>, dostęp w dniu 28.12.2023.

14. React Native Safe Area Context <https://docs.expo.dev/versions/latest/sdk/safe-area-context/>
15. AsyncStorage <https://reactnative.dev/docs/asyncstorage>, dostęp w dniu 28.12.2023.
16. React Native Swiper <https://www.npmjs.com/package/react-native-swiper>, dostęp w dniu 28.12.2023.
17. React Native Dropdown Select List <https://www.npmjs.com/package/react-native-dropdown-select-list>, dostęp w dniu 28.12.2023.
18. Expo Checkbox <https://docs.expo.dev/versions/latest/sdk/checkbox/>, dostęp w dniu 28.12.2023.
19. Validator <https://www.npmjs.com/package/validator>, dostęp w dniu 28.12.2023.
20. Python <https://www.python.org/>, dostęp w dniu 28.12.2023.
21. Django <https://www.djangoproject.com/>, dostęp w dniu 28.12.2023.
22. Django REST framework <https://www.django-rest-framework.org/>, dostęp w dniu 28.12.2023.
23. Simple JWT <https://django-rest-framework-simplejwt.readthedocs.io/en/latest/>, dostęp w dniu 28.12.2023.
24. NetworkX <https://networkx.org/>, dostęp w dniu 28.12.2023.
25. Background task <https://pypi.org/project/django-background-task/>, dostęp w dniu 28.12.2023.
26. MySQLclient <https://readthedocs.org/projects/mysqlclient/>, dostęp w dniu 28.12.2023.
27. MySQL <https://www.mysql.com/>, dostęp w dniu 28.12.2023.

Spis rysunków

1.1	Liczba samochodów osobowych w Polsce w latach 1990-2022 [w mln]. Źródło: [1]	4
1.2	Średni miesięczny obrót (Turnover) oraz średnia odwiedzalność (Footfall) na 1 GLA mkw. (Footfall Density) w I poł. 2023 roku; zmiana r/r 2023/2022. Źródło: [4]	5
3.1	Struktura plików w aplikacji mobilnej	12
3.2	Kod źródłowy pliku App.js	12
3.3	Fragmenty kodu z komponentów kontekstowych	13
3.4	Przykładowe fragmenty kodu stylów komponentów	13
3.5	Schemat nawigacji aplikacji	14
3.6	Kod komponentów nawigacyjnych	15
3.7	Struktura plików serwera	16
3.8	Ruting endpointów modułów serwera	16
3.9	Model użytkownika	17
3.10	Nawigacja węzłów końcowych modułu użytkownicy	17
3.11	Model parkingu	17
3.12	Fragment funkcji tworzącej parking	18
3.13	Algorytm Dijkstry biblioteki networkx	18
3.14	Funkcja obliczająca dystans między miejscami parkingowymi	18
3.15	Nawigacja endpointów modułu parking	18
3.16	Modele rezerwacji	19
3.17	Fragment funkcji tworzącej lub przedłużającej rezerwację	19
3.18	Nawigacja endpointów modułu rezerwacji	19
3.19	Zadanie wykonujące się cyklicznie po uruchomieniu serwera	19
3.20	Model sklepu	20
3.21	Funkcja tworząca sklepy	20
3.22	Parametry połączenia między serwerem a bazą danych	20
3.23	Diagram ERD bazy danych	21
3.24	Schemat przepływu danych w projekcie	21
3.25	Funkcja wysyłająca zapytania sprawdzające JSON Web Token	22
4.1	Wynik przeprowadzonych testów	23
4.2	Kod testu tworzenia parkingu	24
4.3	Kod testu zwracania parkingu	24
4.4	Kod testu rejestracji	25
4.5	Kod testu logowania	25
4.6	Kod testu tworzenia lub przedłużenia rezerwacji	26

4.7	Kod testu zwracania daty końcowej rezerwacji	26
4.8	Kod testu tworzenia sklepów	27
5.1	Ekran logowania i rejestracji	29
5.2	Komunikaty o błędzie wyświetlane przez aplikacje na ekranach logowania i rejestracji	30
5.3	Nawigacja między ekranami wyboru parkingu oraz panelu administratora	30
5.4	Ekran wyboru parkingu	31
5.5	Lista wyboru parkingów w aplikacji	31
5.6	Ekran panelu administratora	32
5.7	Komunikaty o błędzie wyświetlane przez aplikacje na panelu administratora	33
5.8	Nawigacja między ekranami parkingu oraz płatności	33
5.9	Nawigacja jeśli parking jest bezpłatny	33
5.10	Ekran parkingu	34
5.11	Schemat parkingu na drugim piętrze, miejsce użytkownika A14	35
5.12	Panel wyboru opcji miejsca	35
5.13	Powiadomienie o braku wolnych miejsc o wybranych parametrach	36
5.14	Ekran płatności	36
5.15	Wyświetlany czas daty końcowej płatności	37