

# Spis treści

<b>1. Wstęp . . . . .</b>	<b>4</b>
1.1. Cel i zakres pracy . . . . .	4
1.2. Układ pracy . . . . .	4
<b>2. Wymagania aplikacji . . . . .</b>	<b>5</b>
2.1. Wymagania funkcjonalne . . . . .	5
2.1.1. Wymagania funkcjonalne gościa . . . . .	5
2.1.2. Wymagania funkcjonalne użytkownika . . . . .	5
2.1.3. Wymagania niefunkcjonalne . . . . .	6
<b>3. Architektura i zastosowane technologie . . . . .</b>	<b>8</b>
3.1. Architektura . . . . .	8
3.2. Baza danych . . . . .	9
3.3. Część serwerowa aplikacji . . . . .	9
3.4. Część kliencka aplikacji . . . . .	10
3.5. Narzędzia . . . . .	10
3.5.1. Rozwijanie . . . . .	10
3.5.2. Uruchamianie . . . . .	10
3.5.3. Testowanie . . . . .	10
3.5.4. Wdrażanie . . . . .	10
<b>4. Część serwerowa aplikacji . . . . .</b>	<b>11</b>
4.1. Projekt i integracja bazy danych . . . . .	11
4.1.1. Diagram encji bazy danych . . . . .	11
4.1.2. Implementacja bazy danych . . . . .	12
4.1.3. Kontrola stanu struktury bazy danych . . . . .	12
4.2. Widoki i serializery . . . . .	13
4.2.1. Widok - informacje podstawowe . . . . .	13
4.2.2. Zastosowanie widoku . . . . .	13
4.2.3. Serializer - informacje podstawowe . . . . .	14
4.2.4. Zastosowanie Serializerów . . . . .	14
4.3. Bezpieczeństwo . . . . .	15
4.3.1. Haszowanie haseł . . . . .	15
4.3.2. Potwierdzanie e-mail . . . . .	15
4.3.3. JWT . . . . .	15
4.3.4. CORS . . . . .	16
<b>5. Część kliencka aplikacji . . . . .</b>	<b>17</b>
5.1. Projekt graficzny . . . . .	17
5.2. Architektura i implementacja . . . . .	18

---

5.2.1.	Style . . . . .	18
5.2.2.	Komunikacja klient-serwer . . . . .	19
5.2.3.	Formularze . . . . .	19
5.2.4.	Routing . . . . .	20
5.3.	Wygląd wybranych podstron . . . . .	21
<b>6.</b>	<b>Testy aplikacji . . . . .</b>	<b>27</b>
6.1.	Testy części serwerowej aplikacji . . . . .	27
6.1.1.	Testy widoków gościa . . . . .	27
6.1.2.	Testy widoków użytkownika . . . . .	28
6.2.	Testy części klienckiej aplikacji . . . . .	28
<b>7.</b>	<b>Wdrożenie projektu . . . . .</b>	<b>29</b>
7.1.	Plan wdrożenia . . . . .	29
7.1.1.	Kupno serwera . . . . .	29
7.1.2.	Konfiguracja i instalacja niezbędnych programów . . . . .	29
7.1.3.	Uruchomienie aplikacji w sposób produkcyjny . . . . .	29
<b>8.</b>	<b>Aplikacja mobilna . . . . .</b>	<b>31</b>
8.1.	Architektura . . . . .	31
8.2.	Implementacja . . . . .	32
8.2.1.	Podstawowe komponenty . . . . .	32
8.2.2.	Style w React-Native . . . . .	32
8.2.3.	Zarządzanie stanem i cyklem życia . . . . .	33
8.3.	Wygląd wybranych ekranów . . . . .	34
<b>9.</b>	<b>Podsumowanie . . . . .</b>	<b>39</b>
<b>Bibliografia . . . . .</b>		<b>40</b>

# **Rozdział 1**

## **Wstęp**

### **1.1. Cel i zakres pracy**

Celem pracy jest zaprojektowanie, implementacja i wdrożenie aplikacji webowej służącej do wyszukiwania współlokatorów i kwater studenckich. Aplikacja ma za zadanie ułatwiać studentom znalezienie lokum i współlokatora na podstawie zadanych parametrów wyszukiwania jak np. część miasta lub cena. W zakres pracy wchodzi:

- zapoznanie się z technologiami pozwalającymi na wykonanie strony serwera aplikacji webowej,
- zapoznanie się i wybranie odpowiedniego DBMS (Database Management System)[1],
- zapoznanie się z technologiami pozwalającymi na wykonanie strony klienta aplikacji webowej,
- wykonanie diagramu encji dla bazy danych,
- implementacja bazy danych wykorzystując technologie ORM,
- zaprojektowanie wyglądu aplikacji,
- implementacja części serwerowej aplikacji webowej,
- implementacja części klienckiej aplikacji webowej,
- testy aplikacji.

### **1.2. Układ pracy**

Praca jest podzielona w następujący sposób:

- rozdział 2 - opis wymagań funkcjonalnych i niefunkcjonalnych aplikacji,
- rozdział 3 - opis architektury, technologii i uzasadnienie dokonanych wyborów,
- rozdział 4 - opis implementacji części serwerowej aplikacji,
- rozdział 5 - opis implementacji części klienckiej aplikacji,
- rozdział 6 - przedstawienie metodologii testów,
- rozdział 7 - opis wdrożenia aplikacji,
- rozdział 8 - przedstawienie aplikacji mobilnej,
- rozdział 9 - podsumowanie pracy.

## Rozdział 2

# Wymagania aplikacji

### 2.1. Wymagania funkcjonalne

W aplikacji zostały wyróżnione dwa poziomy dostępu: gość i użytkownik. Gość to internauta, odwiedzający stronę. Jeśli posiada konto w serwisie, to jest on niezalogowany. Użytkownik to internauta, posiadający konto w serwisie i będący aktualnie zalogowany.

#### 2.1.1. Wymagania funkcjonalne gościa

- może ustawić parametry wyszukiwania pokojów na wynajem,
- może ustawić parametry wyszukiwania współlokatorów,
- może przeglądać oferty współlokatorów,
- może przeglądać oferty pokojów na wynajem,
- może filtrować wyniki wyszukiwania pokojów na wynajem,
- może filtrować wyniki wyszukiwania współlokatorów,
- może się zarejestrować w serwisie.

#### 2.1.2. Wymagania funkcjonalne użytkownika

- może ustawić parametry wyszukiwania pokojów na wynajem,
- może ustawić parametry wyszukiwania współlokatorów,
- może przeglądać oferty współlokatorów,
- może przeglądać oferty pokojów na wynajem,
- może filtrować wyniki wyszukiwania pokojów na wynajem,
- może filtrować wyniki wyszukiwania współlokatorów,
- może się logować,
- może przypomnieć hasło,
- będąc zalogowanym, może dodawać oferty pokojów na wynajem do 'ulubionych',
- będąc zalogowanym, może dodawać oferty współlokatorów do 'ulubionych',
- będąc zalogowanym, może usuwać oferty pokojów na wynajem z 'ulubionych',
- będąc zalogowanym, może usuwać współlokatorów z 'ulubionych',
- będąc zalogowanym, może przeglądać oferty pokojów na wynajem dodane do 'ulubionych',
- będąc zalogowanym, może przeglądać oferty współlokatorów dodane do 'ulubionych',
- będąc zalogowanym, może prowadzić konwersacje z oferentami za pomocą wbudowanego komunikatora,
- będąc zalogowanym, może dodać własną ofertę pokoju na wynajem,
- będąc zalogowanym, może dodać własną ofertę współlokatorską,

- będąc zalogowanym, może edytować własne oferty pokojów na wynajem,
- będąc zalogowanym, może edytować własne oferty współlokatorskie,
- będąc zalogowanym, może usunąć własne oferty pokojów na wynajem,
- będąc zalogowanym, może usunąć własne oferty współlokatorskie.

Na rysunku 2.1 przedstawiono diagram przypadków użycia wynikający z powyższych wymagań funkcjonalnych.

### 2.1.3. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne zostały określone w oparciu o szablon FURPS[2]

1. obszar funkcjonalności:

- forma: aplikacja webowa,
- aplikacja działa w najnowszych wersjach przeglądarek Chromium i Mozilla Firefox,
- hasła użytkowników są hashowane.

2. obszar użyteczności:

- aplikacja może korzystać z JavaScript, zatem skrypty JavaScript nie mogą być blokowane,
- aplikacja jest utrzymywana w kolorystyce pomarańczowo - szarej,
- aplikacja nie powinna wymagać przeładowania całej strony w celu załadowania danych,
- użytkownik powinien mieć uniemożliwione założenie konta z hasłem zbyt krótkim, lub zbyt podobnym do nazwy użytkownika,
- użytkownik powinien potwierdzić rejestracje e-mailowo,
- użytkownik powinien móc zmienić hasło,
- aplikacja obsługuje kodowanie polskich znaków, czytelnych na różnych platformach.

3. obszar niezawodności:

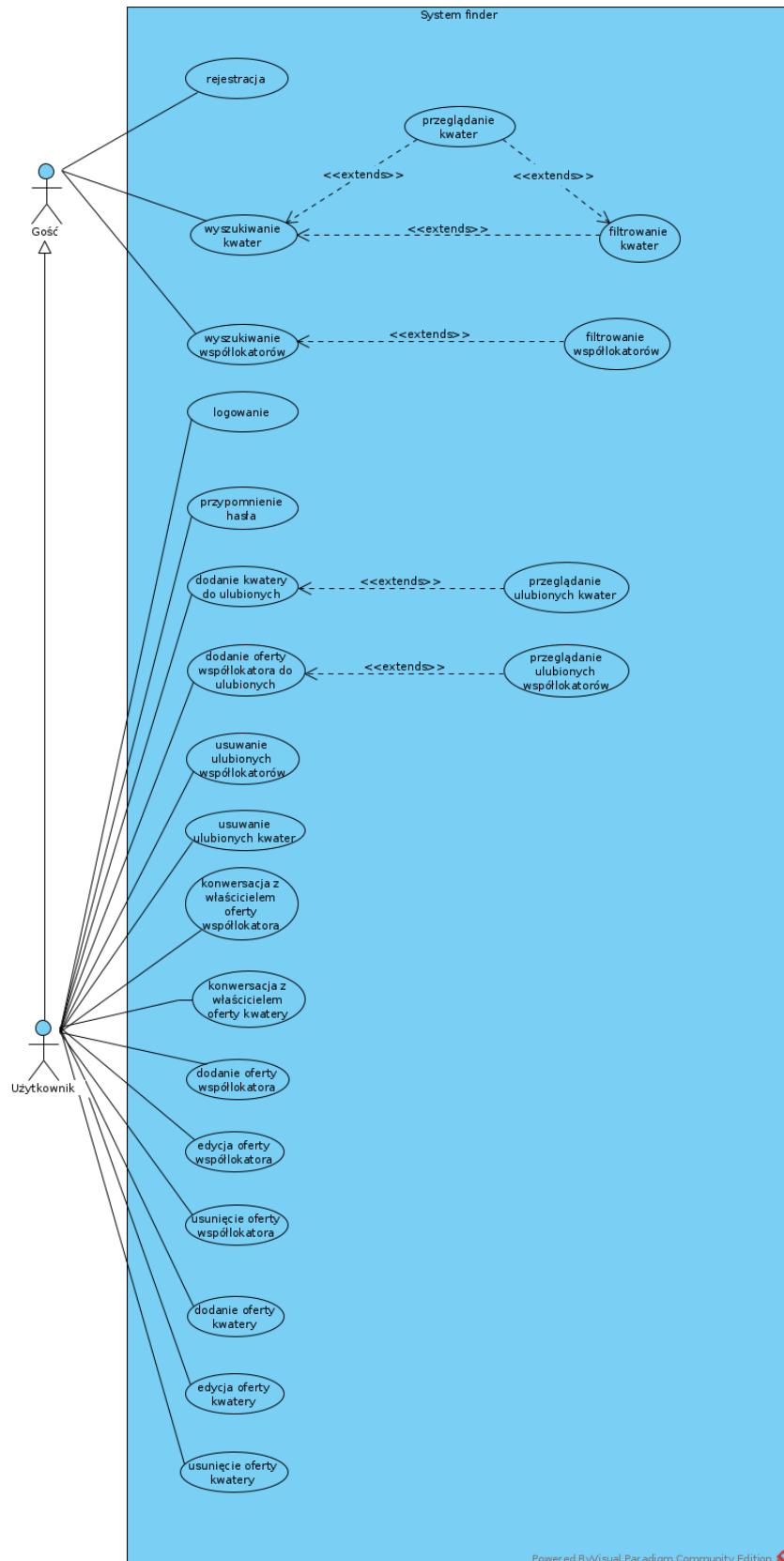
- ewentualne okna serwisowe mogą odbywać się między godziną 22.00, a 6.00.

4. obszar wydajności:

- aplikacja umożliwia korzystanie z niej przez 1 osobę w danym momencie.

5. obszar wsparcia:

- błędy krytyczne będą naprawiane w ciągu 480h od wystąpienia awarii.



Powered By Visual Paradigm Community Edition

Rys. 2.1: Diagram przypadków użycia

## Rozdział 3

# Architektura i zastosowane technologie

### 3.1. Architektura

Pod uwagę zostały wzięte dwa typy architektur: MVT[3] (Model View Template) i REST[4] (REpresentational State Transfer).

Architektura MVT oznacza, że aplikacja jest monolitem, nie ma rozdziału na część kliencką i serwerową. Logika zawarta w widokach (View), wykonuje operacje na tabelach bazy danych (Model), i rezultaty przekazuje do szablonu (Template). Cały proces dzieje się w jednej aplikacji. Frameworkami wykorzystującymi architekturę MVT są m.in. Django i Flask.

Architektura MVT pozwala na szybkie prototypowanie i tworzenie aplikacji. Składa się na to między innymi brak potrzeby implementacji komunikacji między klientem a serwerem i pełna integracja wszystkich części aplikacji.

Wadą takiego rozwiązania jest mała elastyczność. Niemożliwe jest, aby aplikacja działała w formie SPA (Single Page Application)[5]. Jest to spora wada, biorąc pod uwagę aktualne trendy w dziedzinie User Experience[6].

Druga z możliwości to aplikacja w architekturze REST. Głównym założeniem tej architektury jest wykorzystanie architektury klient-serwer. Oznacza to oddzielenie części odpowiedzialnej za interfejs użytkownika i części odpowiedzialnej za przechowywanie i operacje na danych.

Oznacza to także, że należy samodzielnie obsługiwać komunikacje między klientem a serwerem, co przekłada się na zwiększoną ilość kodu. Co więcej, architektura REST pozwala na zrealizowanie wielu aplikacji klienckich, korzystających z jednej aplikacji serwerowej. Przykładowo może być to aplikacja webowa zaimplementowana w Vue.js i aplikacja mobilna zaimplementowana w React Native.

Dla projektu została wybrana architektura REST. Przyczyny tego wyboru są następujące:

1. konieczność zapewnienia użytkownikom jak najlepszego UX podczas korzystania z aplikacji (przez implementację aplikacji klienckiej w technologii umożliwiającej tworzenie SPA),
2. zapewnienie możliwości rozbudowy aplikacji np. kliencką aplikację mobilną lub o publiczne API dla deweloperów,
3. poznanie wcześniej nieznanych technologii.

## 3.2. Baza danych

Wybór DBMS (Database Management System) sprowadzał się do dwóch decyzji:

1. wybranie bazy relacyjnej lub nierelacyjnej,
2. wybranie DBMS.

Wybrana została baza relacyjna, z następujących względów:

- z wstępniego szkicu diagramu encji wynikało, że mogą powstać skomplikowane relacje, i będzie potrzeba wykonania operacji 'join' na tabelach,
- gwarancja prawidłowego przetwarzania danych w bazie - respektowanie ACID[7],
- lepsze wsparcie ze strony narzędzi wykorzystywanych po stronie serwerowej aplikacji,
- większe dotychczasowe doświadczenie w projektowaniu relacyjnych baz danych.

Spośród silników relacyjnych, wybrano PostgreSQL. Decyzja padła ze względu na:

- otwartoźródłowość kodu DBMS[8],
- bardzo dobrą wydajność[9],
- dobrą integrację z częścią serwerową aplikacji[10].

Dobrym wyborem byłoby również MySQL, i w przypadku projektu z punktu widzenia wydajności było bez znaczenia, który z tych dwóch DBMS zostanie wybrany. Jednak dobra integracja i nieco większe możliwości PostgreSQL w połączeniu z częścią serwerową zdecydowały o wyborze właśnie tej technologii.

## 3.3. Część serwerowa aplikacji

Jako że REST jest obecnie wiodącą architekturą dla implementacji aplikacji internetowych, wybór narzędzi ułatwiających zrealizowanie części serwerowej jest bardzo bogaty. Ze względu na znajomość technologii, pod uwagę zostały wzięte frameworki języka Python:

- flask (wraz z flask-restful),
- django (wraz z Django REST Framework).

Flask to mikroframework, którego głównym założeniem jest wysoka konfigurowalność[11]. Jest to niewątpliwą zaletą w niektórych zastosowaniach, jednak w przypadku dużych aplikacji ilość konfiguracji urasta do sporych rozmiarów i zajmuje dużo czasu.

Django to framework, który przychodzi z wbudowanymi mechanizmami mapowania obiektowo relacyjnego, panelu administratora, uwierzytelniania i autoryzacji[3]. W przypadku mniejszych aplikacji jest to zdecydowanie duży narzut, jednak w przypadku bardziej rozbudowanych programów, dostarczone wraz z django rozwiązania zdecydowanie przyśpieszają pracę nad projektem.

Ze względu na spodziewany rozmiar projektu, zdecydowano się na oparcie części serwerowej aplikacji o framework django, wraz z bibliotekami wspierającymi rozwiązania RESTowe

Wraz z wyborem frameworka została podjęta decyzja o użyciu wbudowanego mechanizmu mapowania obiektowo-relacyjnego (ang object-relational-mapping, ORM). Mapowanie obiektowo relacyjne polega na odwzorowaniu relacyjnej bazy danych przez obiektową architekturę[12]. Pozwala to na projektowanie bazy danych niezależenie od używanego silnika DBMS, a także eliminuje potrzebę tworzenia kodu SQL wewnątrz aplikacji.

## 3.4. Część kliencka aplikacji

Ze względu na chęć zapewnienia użytkownikom aplikacji jak najlepszego UX, zdecydowano się na implementację aplikacji webowej w myśl architektury SPA. Oznacza to że aplikacja dynamicznie doładowuje zawartość, zamiast przeładowywać całą stronę. Celem takiego zachowania jest zapewnienie doświadczenia takiego, jak w aplikacji natywnej[5].

Najpopularniejszymi rozwiązaniami z obszaru front-endu są frameworki:

- React
- Angular
- Vue

Z powyższych do projektu został wybrany React ze względu na duży udział w rynku[13], wsparcie firmy Facebook i dużą swobodę w wyborach architektonicznych.

## 3.5. Narzędzia

Częścią projektu są także narzędzia, które pozwalają na wygodne pisanie, uruchamianie i testowanie napisanego kodu.

### 3.5.1. Rozwijanie

Rzeczą niewątpliwie potrzebną jest zapisywanie postępów pracy i możliwość cofnięcia się do nich po pewnym czasie. Te, i wiele więcej funkcjonalności oferuje system kontroli wersji git. Dla bezpieczeństwa i zminimalizowania ryzyka utraty danych, zostało wykorzystane także zdalne repozytorium na stronie [github.com](https://github.com).

### 3.5.2. Uruchamianie

Do uruchamiania projekt korzysta z dwóch powiązanych ze sobą narzędzi: Docker i docker-compose.

Docker pozwala na konteneryzację aplikacji, czyli izolację aplikacji od reszty środowiska. Rozwiązuje to problem pracy na różnych systemach i różnych wersjach oprogramowania[14].

Docker-compose to narzędzie pozwalające działać na wielu kontenerach jednocześnie[15]. W przypadku projektu, dzięki docker-compose można jednocześnie uruchomić kontener dla części serwerowej, klienckiej i bazy danych.

### 3.5.3. Testowanie

Do testowania części serwerowej wykorzystano program curl. Pozwala on na tworzenie zapytań http. Jest to program działający z linii komend, a więc możliwe jest zastosowanie daleko idącej optymalizacji testowania w postaci opracowania skryptu powłoki.

### 3.5.4. Wdrażanie

W celu zapewnienia wysokiej wydajności i bezpieczeństwa koniecznym jest skorzystanie z produkcyjnego serwera wsgi i http[16]. W tym celu zastosowane zostały gunicorn i nginx. Do zarządzania stanem aplikacji wykorzystano usługę systemd[17].

# Rozdział 4

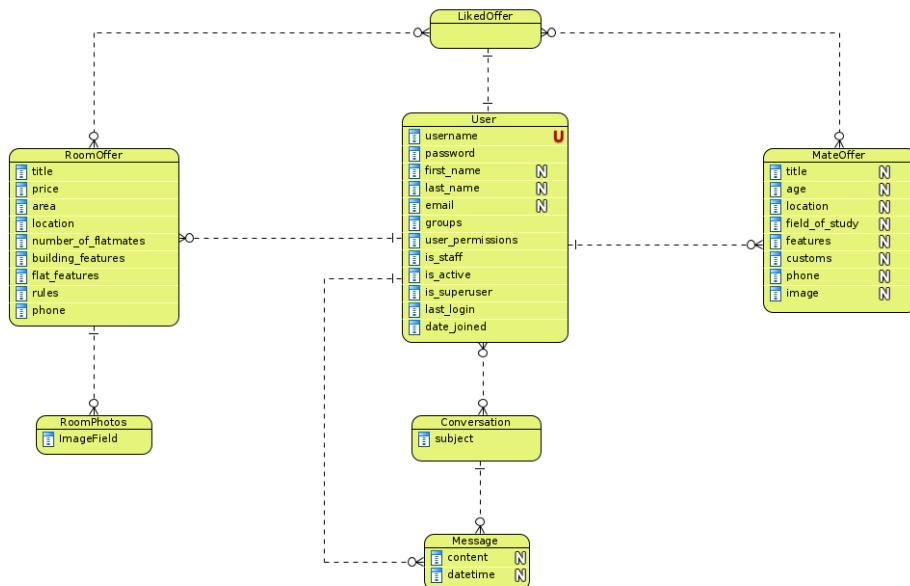
## Część serwerowa aplikacji

Część serwerowa aplikacji została wykonana w frameworku Django wraz z Django REST framework. Odpowiada ona za generowanie i wysyłanie odpowiedzi na zapytania części klienckiej. Aplikacja serwerowa jest także bezpośrednio odpowiedzialna za implementacje bazy danych. Poprzez mapowanie obiektowo-relacyjne dostarczone przez Django, możliwe jest zdefiniowanie encji bazy danych za pomocą klas Pythona.

### 4.1. Projekt i integracja bazy danych

#### 4.1.1. Diagram encji bazy danych

Jako że aplikacja działa na trwałych danych, projektowanie należało zacząć od koncepcji bazy danych. Po analizie wymagań, zrealizowany został ERD (Entity-Relationship Diagram) wykorzystujący koncepcyjny model danych (CDM, Conceptual Data Model).



Rys. 4.1: Koncepcyjny diagram encji

## 4.1.2. Implementacja bazy danych

Na podstawie diagramu 4.1 zaimplementowano modele baz danych. Przykładowa implementacja została pokazana w listingu 4.1

Listing 4.1: Implementacja encji na przykładzie RoomOffer

---

```

1 class RoomOffer(models.Model):
2     owner = models.ForeignKey(User, related_name='room_offer', on_delete=models.CASCADE, null=
3         ↪ False, blank=False)
4     title = models.CharField(max_length=100)
5     price = models.IntegerField(default=0)
6     area = models.IntegerField(default=0)
7     location = models.CharField(max_length=100)
8     number_of_flatmates = models.IntegerField(default=0)
9     building_features = models.TextField()
10    flat_features = models.TextField()
11    flatmates_features = models.TextField()
12    rules = models.TextField()
13    phone = models.CharField(max_length=20)
14
15    def __str__(self):
16        return f'{self.area}: {self.title}'

```

---

Jak widać, jako typy danych stosowane są pola dostarczone przez mechanizm ORM Django. Buduje to warstwę abstrakcji i pozwala na implementacje bazy danych niezależnie od używanego systemu zarządzania bazą danych.

Mechanizm ORM dostarczony przez framework Django pozwala także na implementacje triggerów. Przykładowy trigger pokazany w listingu 4.2 tworzy rekord w tabeli LikedOffer, będącej w relacji z tabelą User, kiedy to nowy User został utworzony.

Listing 4.2: Przykładowy trigger na przykładzie modelu LikedOffer

---

```

1 class LikedOffer(models.Model):
2     user = models.OneToOneField(User, on_delete=models.CASCADE, related_name='liked_offer')
3     room_offers = models.ManyToManyField(RoomOffer)
4     mate_offers = models.ManyToManyField(MateOffer)
5
6 @receiver(post_save, sender=User)
7 def create_user_profile(sender, instance, created, **kwargs):
8     if created:
9         LikedOffer.objects.create(user=instance)

```

---

## 4.1.3. Kontrola stanu struktury bazy danych

Mechanizm ORM dostarczony przez Django umożliwia zapisywanie stanu struktury bazy danych (tabel i kolumn). Umożliwia to odtworzenie struktury bazy danych z dowolnego wcześniejszego zapisu. Zapisy stanu struktury bazy danych nazywane są migracjami[18]. Migracja jest wywoływana komendą i jest automatycznie generowana. Plik migracji można edytować, aby ewentualnie skorygować niespodziewane zachowania, lub dostosować ją do indywidualnych potrzeb. Przykładowa migracja zawierająca usunięcie i dodanie kolumny w encji MateOffer została przedstawiona w listingu 4.3.

Listing 4.3: Przykładowa migracja na przykładzie modelu MateOffer

---

```

1 # Generated by Django 3.0.7 on 2020-07-09 12:27
2
3 from django.db import migrations, models
4
5
6 class Migration(migrations.Migration):
7
8     dependencies = [
9         ('rest_api', '0003_likedoffer'),
10    ]
11
12    operations = [

```

---

```

13     migrations.RemoveField(
14         model_name='mateoffer',
15         name='area',
16     ),
17     migrations.AddField(
18         model_name='mateoffer',
19         name='location',
20         field=models.CharField(default='Grunwaldzki Square', max_length=100),
21         preserve_default=False,
22     ),
23 ]

```

---

## 4.2. Widoki i serializery

Widoki i serializery to zasoby odpowiedzialne za logikę i kodowanie informacji uzyskiwanych i wysyłanych do części klienckiej aplikacji.

### 4.2.1. Widok - informacje podstawowe

Widoki to zasoby API, które można osiągnąć pod określonym URL. W widoku znajduje się logika działania wydzielonej części aplikacji. Mogą tam zachodzić operacje na bazie danych, operacje arytmetyczne, czy processing tekstu. Mogą zwracać dowolny rodzaj danych (tekst, szablony HTML, dane binarne, etc)[19]. W przypadku aplikacji dane będą zwracane w formie JSONa.

Django REST Framework dodaje nowe rodzaje widoków. Są to widoki oparte o klasy. Są one bardzo generyczne i w typowych przypadkach potrafią w znaczny sposób skrócić kod[20]. W projekcie jednak w niewielu miejscach była możliwość użycia generycznego widoku, więc zdecydowano korzystać z widoków w formie funkcji.

### 4.2.2. Zastosowanie widoku

Dodanie widoku do API można opisać w następujących krokach:

1. implementacja logiki w funkcji; pobranie niezbędnych danych i ich processing. Przykład widoku znajduje się w listingu 4.4,

Listing 4.4: Przykładowy widok - dodanie współlokatora do ulubionych

```

1  @api_view(['POST'])
2  def add_mate_offer_to_liked(request):
3      data = request.data
4      data['owner'] = request.user.id
5
6      try:
7          user_liked_offers = get_object_or_404(LikedOffer, user=data['owner'])
8          liked_offer = get_object_or_404(MateOffer, id=data['id'])
9          user_liked_offers.mate_offers.add(liked_offer)
10         return Response(status=200)
11     except:
12         import traceback
13         for line in traceback.format_exc().splitlines():
14             print(line)
15         return Response(status=400)

```

---

2. zdefiniowanie metod HTTP, jakie ma akceptować widok, klas parserów oraz praw dostępu do widoku. Określa się to za pomocą dekoratorów. W przypadku braku dekoratora, stosowane są ustawienia domyślne. W listingu 4.4 określono tylko metodę HTTP, zaś resztę ustawień pozostawiono domyślnie,

3. zmapowanie zasobu na url w pliku urls.py. Przykładowe mapowanie znajduje się w listingu 4.5.

Listing 4.5: Przykładowe mapowanie url na widok

---

```

1 urlpatterns = [
2     # pozostałe sciezki
3     path('add_mate_offer_to_liked/', views.add_mate_offer_to_liked),
4 ]

```

---

### 4.2.3. Serializer - informacje podstawowe

Serializery to klasy mające za zadanie zarówno serializować odpowiedzi API na żądania klienta, jak i deserializować dane wysyłane do serwera. Polega to na zamianie złożonych typów obiektowych na tekst, i tekstu na złożone typy[21]. Serializery są częścią Django REST Framework

### 4.2.4. Zastosowanie Serializerów

Serializer tworzony jest na podstawie modelu. Należy określić jakie pola danego modelu mają być kodowane i dekodowane. Serializery mogą być zagnieżdżone. Obrazuje to listing 4.6. Serializer konwersacji zwraca nie tylko pola obiektu modelu bazowego (subject), ale także zagnieżdża inne serializery (message i members).

Listing 4.6: Przykładowy serializer

---

```

1 class ConversationSerializer(serializers.ModelSerializer):
2     message = MessageSerializer(many=True, read_only=True)
3     members = UserSerializer(many=True, read_only=True)
4
5     class Meta:
6         model = Conversation
7         fields = ['id', 'members', 'subject', 'message']

```

---

Przykładowa odpowiedź wygenerowana przez serializer z listingu 4.6 została zamieszczona w listingu 4.7

Listing 4.7: Odpowiedź wygenerowana przez serializer 4.6

---

```

1 {
2     "id": 2,
3     "members": [
4         {
5             "username": "admin",
6             "email": "admin@admin.pl"
7         },
8         {
9             "username": "a",
10            "email": "a@a.pl"
11        }
12    ],
13    "subject": "Peaceful IT student",
14    "message": [
15        {
16            "owner": 3,
17            "content": "Dobry",
18            "datetime": "2020-08-14T17:06:30.173766Z"
19        },
20        {
21            "owner": 1,
22            "content": "Co tam?",
23            "datetime": "2020-08-14T17:32:20.514572Z"
24        }
25    ]
26 }

```

---

Z perspektywy zwracania odpowiedzi, zastosowanie serializera przedstawiono w listingu 4.8. Dane są pobierane z bazy danych, a następnie serializowane do tekstu w formie JSON.

Listing 4.8: Zastosowanie serializatora w widoku

---

```

1 @api_view(['GET'])
2 def get_conversation(request, conversation_id):
3     try:
4         owner = request.user
5         data = Conversation.objects.get(pk=conversation_id)
6         serializer = ConversationSerializer(data)
7         return Response(serializer.data, status=200)
8     except:
9         import traceback
10        for line in traceback.format_exc().splitlines():
11            print(line)
12        return Response(status=400)

```

---

Z perspektywy otrzymywania danych, zastosowanie serializera przedstawiono w listingu 4.9. Sprawdzane są kompletność i poprawność danych, a następnie, w przypadku powodzenia, dane zapisywane są do bazy.

Listing 4.9: Zastosowanie serializatora w widoku

---

```

1 @api_view(['POST'])
2 @permission_classes([AllowAny])
3 def register_user(request):
4     data = request.data
5     serializer = UserSerializer(data=data)
6
7     if serializer.is_valid():
8         serializer.save()
9     return JsonResponse(serializer.data, status=status.HTTP_201_CREATED)
10    return JsonResponse(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

```

---

## 4.3. Bezpieczeństwo

### 4.3.1. Haszowanie haseł

Aby zapobiec poznaniu haseł przy ewentualnym włamaniu lub wycieku danych, postanowiono zastosować haszowanie haseł. Polega ono na zakodowaniu hasła w ciąg znaków, według zadanego algorytmu (w przypadku Django PBKDF2 z haszem SHA-256[22]). Mechanizm ten dostarczany jest przez framework Django (metoda set\_password())[23]).

### 4.3.2. Potwierdzanie e-mail

Aby uniknąć masowego zakładania kont przez roboty, i umożliwić reset hasła, zastosowano weryfikację konta przez adres e-mail. W celu wysyłania wiadomości e-mail zastosowano skrzynkę pocztową Gmail.

Do obsługi mechaniki wysyłania została zastosowana biblioteka django-rest-registration. Jej użycie sprawdza się do dodania w kodzie dodatkowego wpisu do pliku urls.py. Na adres zmapowany pod właśnie ten url, przychodzą klienckie zapytania o zmianę hasła czy potwierdzenie adresu e-mail[24].

### 4.3.3. JWT

JWT - Json Web Token - to sposób bezpiecznej wymiany danych. Jest używany do autoryzacji użytkowników[25]. Użytkownik logując się, dostaje token. Token przechowywany jest w localStorage. Na podstawie tokenu przyznawane są uprawnienia do

interakcji z częścią aplikacji, do której dostęp ma tylko zalogowany użytkownik.

Podobnie jak w przypadku potwierdzenia e-mail, została użyta zewnętrzna biblioteka (`djangorestframework-simplejwt`). Dostarcza ona dwa urle: pod którym można uzyskać token podczas logowania, i pod którym można odświeżyć token.

#### 4.3.4. CORS

CORS (Cross-Origin Resource Sharing) - to mechanizm bazujący na nagłówkach HTTP, który umożliwia wskazanie części serwerowej wszystkich źródeł, z których przeglądarka powinna pozwolić załadować zasoby[26]. Brak odpowiednich nagłówków HTTP w odpowiedzi serwera, uniemożliwi odebranie danych przez przeglądarkę.

W praktyce CORS pozwala ograniczyć zbiór klientów, którzy mogą się kontaktować z API. Kontrolę nad CORS umożliwia biblioteka `django-cors-headers`[27]. Udostępnia ona możliwość zapełnienia listy z dozwolonymi adresami klientów (`CORS_ALLOWED_ORIGINS`), lub pozwolenie wszystkim klientom na połączenia (`CORS_ORIGIN_ALLOW_ALL`). Ze względu na ciągły rozwój projektu, obecnie każdy klient może się połączyć z aplikacją serwerową.

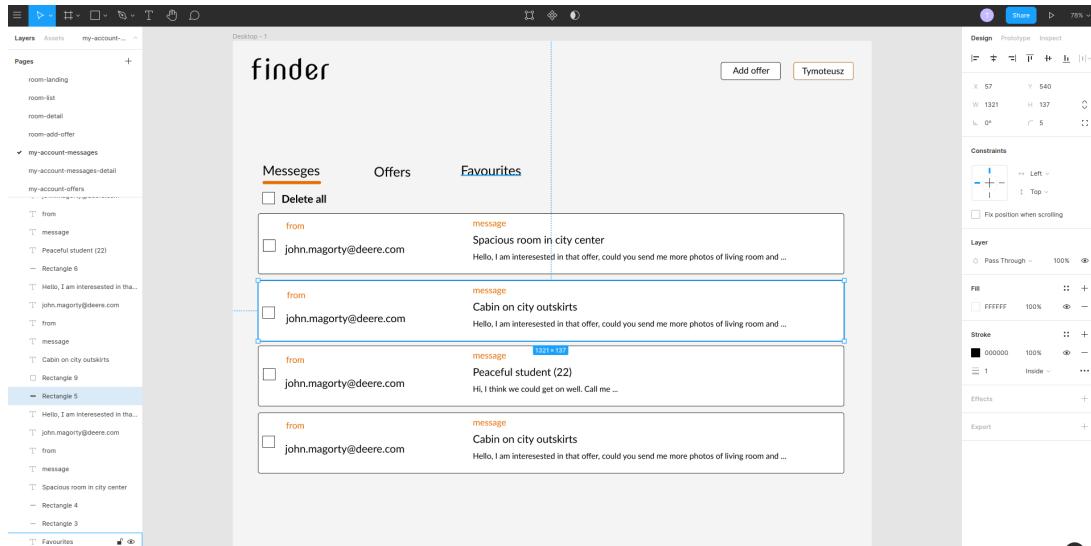
# Rozdział 5

## Część kliencka aplikacji

Część kliencka aplikacji, zgodnie z założeniami, została zrealizowana w architekturze SPA, za pomocą języka Javascript i frameworka React. Aplikacja kliencka odpowiada za generowanie podstron, tworzenie adresów url, i bierze udział w autoryzacji użytkowników serwisu.

### 5.1. Projekt graficzny

Przed implementacją każdej podstrony i komponentu powstał projekt graficzny. Do tego celu zostało użyte narzędzie Figma. Narzędzie to jest przystosowane do projektowania interfejsów webowych. Oferuje nawet wygenerowany kod css, jest jednak nieoptymalny i nadmiarowy, i nie został on wykorzystany przy implementacji części klienckiej. Rysunek 5.1 przedstawia interfejs programu Figma.



Rys. 5.1: Interfejs programu Figma

Po analizie wymagań stwierdzono że należy zaprojektować poniżej wymienione podstrony:

- strona startowa dla części aplikacji do wyszukiwania kwater studenckich,
- strona startowa dla części aplikacji do wyszukiwania współlokatorów,
- strona wyliczająca znalezione oferty kwater studenckich,
- strona wyliczająca znalezione oferty kwater współlokatorów,
- strona wyświetlająca szczegółowe informacje o kwaterze studenckiej,

- strona wyświetlająca szczegółowe informacje o współlokatorze,
- panel użytkownika,
- widok listy wszystkich konwersacji użytkownika,
- widok szczegółowy konwersacji użytkownika,
- widok listy wszystkich ofert użytkownika,
- widok szczegółowy ofert użytkownika,
- widok listy wszystkich ulubionych ofert użytkownika,
- strona do dodania nowej oferty kwatera studenckiej
- strona do dodania nowej oferty współlokatora,
- strona do edycji oferty kwatera studenckiej,
- strona do edycji oferty współlokatora,
- strona umożliwiająca potwierdzenie rejestracji użytkownika,
- strona do wysłania e-maila umożliwiającego zmianę hasła,
- strona umożliwiająca reset hasła.

## 5.2. Architektura i implementacja

Filozofia tworzenia nowoczesnych aplikacji front-endowych zakłada tworzenie komponentów wielokrotnego użytku[28]. W związku z tym, został zastosowany podział na możliwe do wielokrotnego użytku komponenty i podstrony, które się z nich składają.

### 5.2.1. Style

Na przykładzie listingu 5.1 został pokazany sposób stylowania komponentów. Została wykorzystana biblioteka styled-components. Pozwala ona na stylowanie elementów bez konieczności wymyślania globalnie unikalnych nazw klas, upraszcza dynamiczne stylowanie na podstawie stanu lub właściwości, a co najważniejsze, dzięki dzieleniu kodu (ang. code splitting) wczytywanie css jest szybsze niż w przypadku wczytywania całego arkusza jednocześnie[29]. Możliwe jest też używanie zmiennych[30] oraz zagnieżdżeń[31], dostępnych w preprocesorach css.

Listing 5.1: Stylowanie za pomocą styled-components

```

1 import React from 'react';
2 import styled from 'styled-components';
3 // ...
4
5 const Container = styled.div `
6   width: 100%;
7   display: flex;
8   flex-direction: column;
9 `
10
11 const Title = styled.h3 `
12   text-align: left;
13   font-weight: 700;
14   font-size: 30px;
15   padding: 30px;
16 `
17
18 const PhotoWrapper = styled.div `
19   width: 100%;
20   background-color: var(--color-dark-grey);
21   height: 400px;
22   display: flex;
23 `
24
25 const Photo = styled.img `
26   margin: 0 auto;
27 
```

```

28     height: 400px;
29     object-fit: cover;
30   '
31   // ...

```

---

## 5.2.2. Komunikacja klient-serwer

Do komunikacji z aplikacją serwerową wykorzystano bibliotekę axios. Ze względu na wysoko poziomowość (automatyczna konwersja z tekstu do JSON, łatwość ustawienia timeoutu) i wsparcie starszych przeglądarek[32], axios zdaje się być najlepszym z dostępnych rozwiązań.

Komunikacja z aplikacją serwerową odbywała się zwykle zaraz po zamontowaniu komponentu. Przykład komunikacji z API serwera znajduje się na listingu 5.2

Listing 5.2: Pobieranie danych za pomocą biblioteki axios

```

1 class MateDetailPage extends React.Component {
2   constructor(props) {
3     this.state = {data: {}}
4   }
5
6   componentDidMount() {
7
8     axios.get(`http://localhost:8000/mate_offer_detail/${this.props.match.params.offerId
9       }`)
10      .then(res => {
11        if (res.status === 200) {
12          this.setState({ data: res.data });
13        }
14      })
15
16    render() {
17      // ...
18    }
19 }

```

---

## 5.2.3. Formularze

Do pozyskiwania danych od użytkownika wykorzystano formularze HTML zintegrowane z biblioteką Formik. Ułatwia ona dostęp do stanu formularza, walidację i wyświetlanie wiadomości o błędzie przy wprowadzaniu danych oraz obsługę zatwierdzania wypełnionego formularza[33]. Obsługę formularza za pomocą biblioteki formik przedstawia listing 5.3.

Listing 5.3: Obsługa formularza z pomocą biblioteki Formik

```

1 <Formik
2   initialValues={{
3     username: '',
4     email: '',
5     password: '',
6     repeatedPassword: ''
7   }}
8
9   validationSchema={Yup.object({
10     username: Yup.string()
11       .required('required'),
12     email: Yup.string()
13       .email('invalid email address')
14       .required('required'),
15     password: Yup.string()
16       .required('password is required'),
17     repeatedPassword: Yup.string()
18       .oneOf([Yup.ref('password'), null], 'passwords must match')
19   })}
20

```

```

21  onSubmit={ values => {
22    axios . post(`http://127.0.0.1:8000/accounts/register/`,
23      // password_confirm required by Django REST Registration
24      { username: values.username , email: values.email , password: values.password ,
25       ↪ password_confirm: values.password })
26      .then(res => {
27        // ...
28      }).catch(error => {
29        // ...
30      });
31  }
32 >
33
34 {props => (
35   <FormWrapper onSubmit={props.handleSubmit}>
36     <Title>Register </Title>
37     <InputAndLabel label="username" name="username" id="username" onChange={props.
38       ↪ handleChange} value={props.values.username} />
39     {props.errors.username && <Feedback>{props.errors.username}</Feedback>}
40     // ...
41     <ButtonWrapper>
42       <PopupButton content="Register" />
43     </ButtonWrapper>
44   </FormWrapper>
45 )
46 )
47 </Formik>

```

Warto przyjrzeć się walidacji formularzy. Jest ona wykonywana za pomocą biblioteki Yup. Pozwala ona na walidację pól w oparciu o predefiniowane schematy[34]. Pozwala to na pominięcie etapu tworzenia wyrażeń regularnych np. aby sprawdzić poprawność wprowadzonego adresu e-mail.

## 5.2.4. Routing

Do zrealizowania routingu wykorzystano bibliotekę react-router. Pozwala ona w ramach aplikacji reaktowej tworzyć nawigowalne adresy URL[35]. Ustawienie routingu polega na zmapowaniu komponentów pod ścieżki, pod którymi będą one renderowane. Konfigurację routera w aplikacji przedstawia listing 5.4

Listing 5.4: Konfiguracja routingu

```

1 <Router>
2   <Switch>
3     <PrivateRoute component={ConversationPage} path="/conversations/:conversationId" />
4     <PublicRoute component={RoomDetailPage} path="/rooms/:offerId" />
5     <PublicRoute component={MateDetailPage} path="/mates/:offerId" />
6     <PrivateRoute component={AddMatePage} path="/edit/mates/:offerId" />
7     <PrivateRoute component={AddRoomPage} path="/edit/rooms/:offerId" />
8     <PublicRoute component={AccountConfirmedPage} path="/verify-user" />
9     <PublicRoute component={ResetPasswordPage} path="/reset-password" />
10    <PublicRoute component={MateListPage} path="/mate/list/" />
11    <PublicRoute component={RoomListPage} path="/room/list/" />
12    <PrivateRoute component={AddMatePage} path="/add/mates" />
13    <PrivateRoute component={AddRoomPage} path="/add/rooms" />
14    <PrivateRoute component={AccountPage} path="/account" />
15    <PublicRoute component={ForgotPasswordPage} path="/forgot-password" />
16    <Route path="/mates">
17      <LandingPage
18        title={'Find your mate in Wrocław'}
19        image={mate_landing}
20        renderLoginPopup={this.state.displayLoginPopup}
21        renderRegisterPopup={this.state.displayRegisterPopup}
22        handleLoginClosing={this.handleLoginClosing}
23        handleRegisterClosing={this.handleRegisterClosing}
24        handleSwitchVisibility={this.handleSwitchVisibility}
25        handleLoginButtonChange={this.handleLoginButtonChange} />
26    </Route>
27    <Route path="/rooms">

```

```

28 <LandingPage
29   title={'Find your room in Wrocław'}
30   image={room_landing}
31   renderLoginPopup={this.state.displayLoginPopup}
32   renderRegisterPopup={this.state.displayRegisterPopup}
33   handleLoginClosing={this.handleLoginClosing}
34   handleRegisterClosing={this.handleRegisterClosing}
35   handleSwitchVisibility={this.handleSwitchVisibility}
36   handleLoginButtonChange={this.handleLoginButtonChange} />
37 </Route>
38 </Switch>
39 </Router>

```

W tym miejscu odbywa się także autoryzacja. Goście serwisu, nie posiadający tokenu, nie będą mieli dostępu do prywatnych ścieżek (PrivateRoute).

## 5.3. Wygląd wybranych podstron

Na poniższych rysunkach przedstawiony jest wygląd wybranych podstron aplikacji Finder. Pierwszą stroną, na jaką napotka każdy z gości, jest strona startowa. Widoczna jest na rysunku 5.2. Posiada ona menu, które dynamicznie dostosowuje opcje w zależności, od tego czy użytkownik jest zalogowany, czy nie.



Rys. 5.2: Strona startowa

Dodatkowo, jeśli użytkownik wybierze z menu opcję logowania szukania, na środku strony pojawi się okno dialogowe, zaprezentowane odpowiednio na rysunkach 5.4 i 5.3. Dodatkowo, z okna logowania można przejść do okna rejestracji, widocznego na rysunku 5.5.



Rys. 5.3: Okienko dialogowe do szukania współlokatorów



Rys. 5.4: Okienko dialogowe do logowania



Rys. 5.5: Okienko dialogowe do rejestracji

Po wskazaniu parametrów szukania, użytkownik zostanie przeniesiony do widoku listy ofert. Tam, zarejestrowany użytkownik może dodać ofertę do ulubionych, klikając na serce z prawej strony oferty. Widok listy widoczny jest na rysunku 5.6. W tym miejscu, u góry listy, jest także możliwość dalszego filtrowania rezultatów.

Rys. 5.6: Strona z listą wyników szukania i filtrowania

Po wybraniu którejś z ofert użytkownik jest przekierowany do widoku detalicznego. Na rysunku 5.7 przedstawiony jest widok detaliczny współlokatora, zaś na rysunku 5.8 przedstawiony jest widok detaliczny kwater studenckiej. Z poziomu widoku detalicznego, użytkownicy mogą także wysyłać do siebie wiadomości.

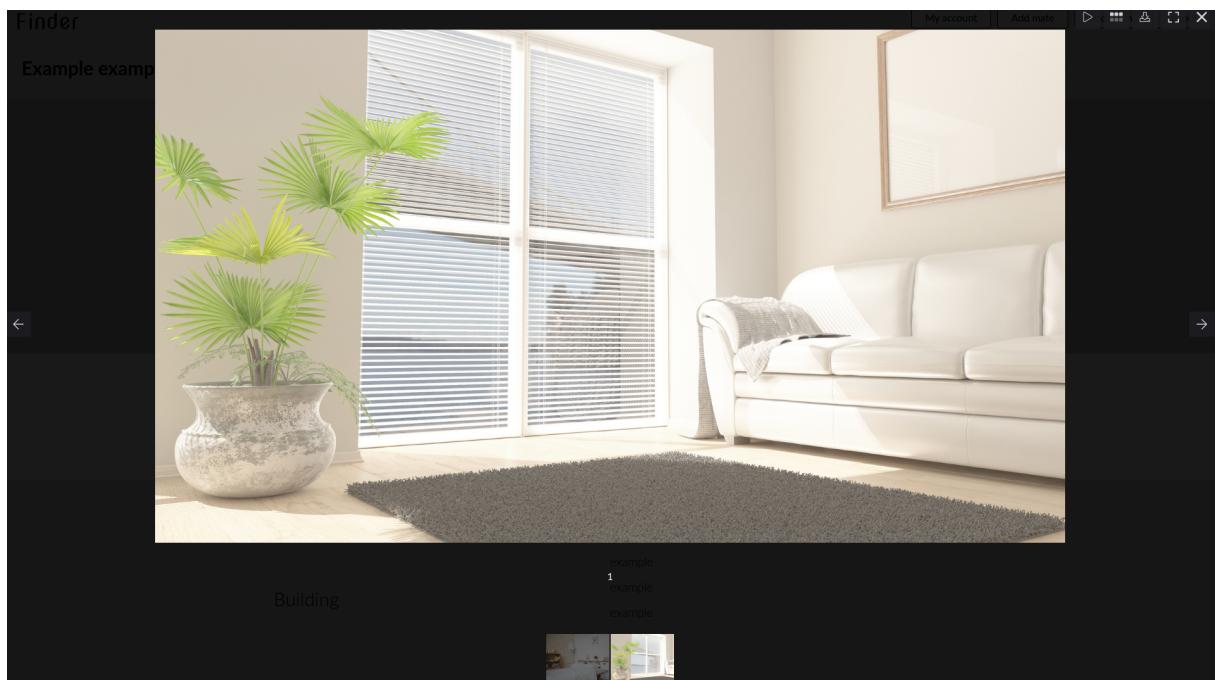
The screenshot shows a user profile for "Dent Student" at the top. Below it, a search result for "Krzyki" is displayed, featuring a yellow background image of a person sitting cross-legged with books, a title "Features", and a list of tags: "busy", "nice", "quiet". Another section for "Customs" lists "get up early" and "hard studying". A "Contact" section includes a phone number "123123123" and a large empty input field with a "Send" button. At the bottom, there's a navigation bar with links: "My account", "Add mate", "Add room", and "Logout".

Rys. 5.7: Strona z detalicznym opisem współlokatora

The screenshot shows a room listing for "Example example". It features two images of a bedroom: one showing a double bed and a brick wall, and another showing a white sofa and a window with blinds. The top navigation bar includes "Finder", "My account", "Add mate", "Add room", and "Logout". Below the images, the price is listed as "1800PLN/month" and the area as "21 m<sup>2</sup>". A "Building" section contains a list of "example" entries. At the bottom, there's a navigation bar with links: "My account", "Add mate", "Add room", and "Logout".

Rys. 5.8: Strona z detalicznym opisem kwater studenckiej

Rysunek 5.9 przedstawia galerię, która pokazuje się przy kliknięciu na zdjęcia w widoku detalicznym kwater studenckiej.



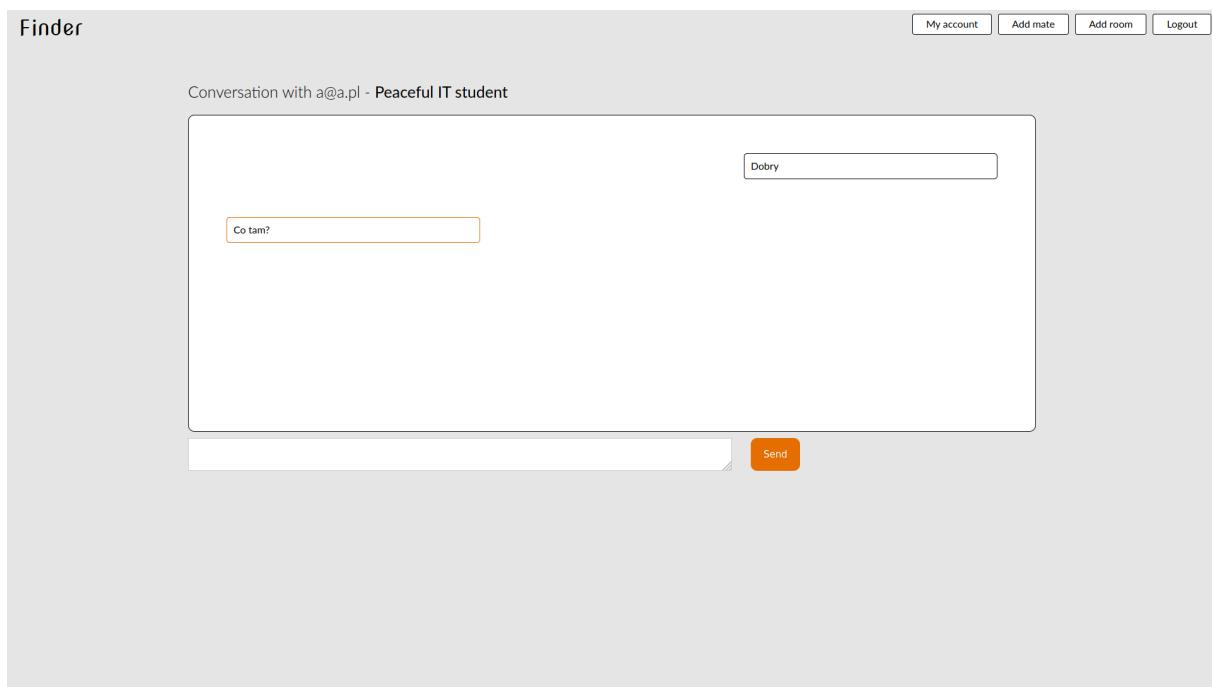
Rys. 5.9: Galeria na stronie z detalicznym opisem kwatery studenckiej

Rysunki 5.10 i 5.11 pokazują odpowiednio widok listy wszystkich konwersacji i widok detaliczny wybranej konwersacji.

The screenshot shows a user interface for messaging. At the top, there are buttons for "My account", "Add mate", "Add room", and "Logout". Below this, there are tabs for "Messages", "Offers", and "Favourites". The "Messages" tab is selected. There are two messages listed:

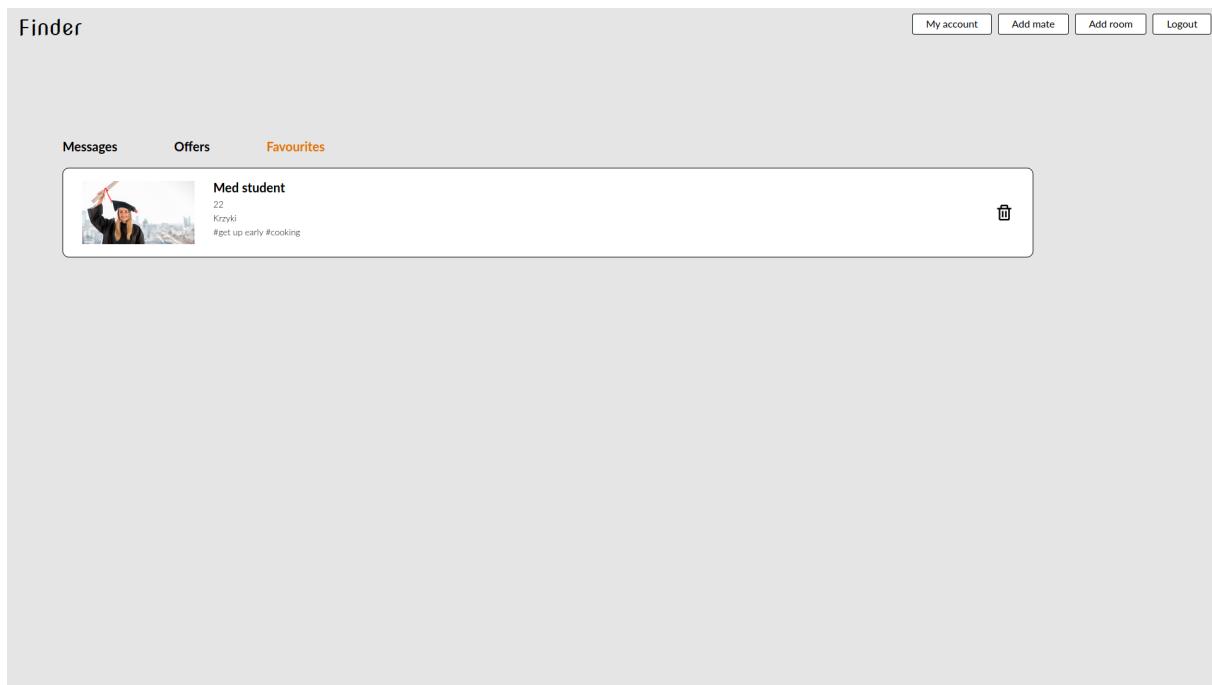
- From: a@a.pl (message) Room in city center  
Wiadomość od A
- From: a@a.pl (message) Peaceful IT student  
Co tam?

Rys. 5.10: Panel użytkownika - widok na wiadomości



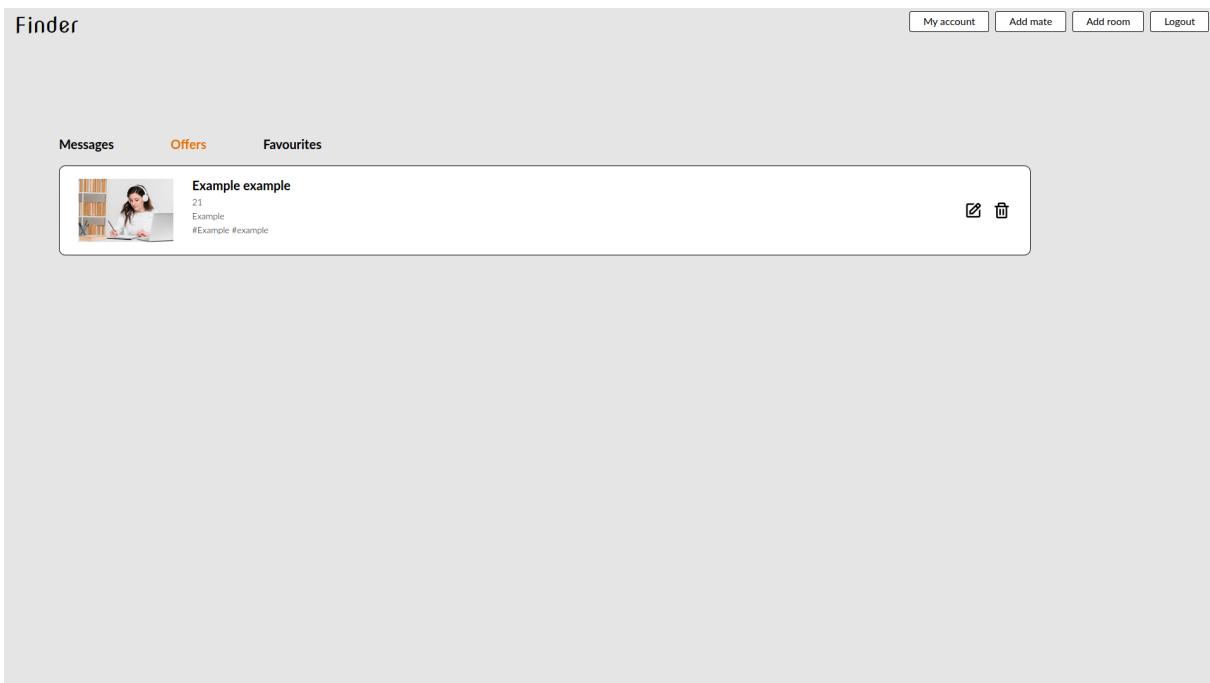
Rys. 5.11: Panel użytkownika - konwersacja

Rysunek 5.12 pokazuje wszystkie ulubione oferty użytkownika. Oferty można usunąć z ulubionych przez wybranie śmiertnika po prawej stronie, lub przejść do widoku detalicznego, klikając na zdjęcie na ofercie.



Rys. 5.12: Panel użytkownika - widok na ulubione oferty

Na stronie zaprezentowanej na rysunku 5.13 podsumowane są wszystkie oferty użytkownika. Można je edytować lub usunąć, korzystając z przycisków po prawej stronie.



Rys. 5.13: Panel użytkownika - widok na własne oferty

Strona edycji i dodawania, zaprezentowana na rysunku 5.14 pozwala na edytowanie oferty (wtedy zapełniana jest aktualnymi danymi) lub dodanie nowej (wtedy wszystkie pola są puste).

The screenshot shows a form for managing an offer. At the top right, there are links for 'My account', 'Add mate', 'Add room', and 'Logout'. The form fields include:

- Title:** Example example
- Photo:** A file input field with '3875528.jpg' selected.
- Age:** Input field containing '21'.
- Field of study:** Input field containing 'Example'.
- Phone:** Input field containing 'Example'.
- Location:** Input field containing 'Example'.
- Personal features, Separable with semicolon:** Input field containing 'Example:example'.
- Customs, Separate with semicolon:** Input field containing 'Example:example:example'.

A red 'Submit' button is located at the bottom right of the form area.

Rys. 5.14: Strona służąca do dodawania i edycji własnej oferty współlokatora

# Rozdział 6

## Testy aplikacji

### 6.1. Testy części serwerowej aplikacji

Część serwerową aplikacji przetestowano za pomocą testów jednostkowych[36]. Testy jednostkowe sprawdzają najmniejszy możliwy do przetestowania wycinek kodu. Podczas testowania przyjmowana jest zasada, że sposób działania funkcji jest znany (white-box). Testy jednostkowe są szybkie w realizacji i pozwalają wykryć błędy, zanim będzie to bardziej czasochłonne i kosztowne.

Do testów części serwerowej wykorzystano mechanizmy dostarczane przez Django i django REST framework. Mechanizmy te charakteryzują się tym, że testy są odizolowane od bazy danych aplikacji (za każdym uruchomieniem testów tworzona jest tymczasowa baza). Dodatkowo, testy nie zachowują stałej kolejności.

#### 6.1.1. Testy widoków gościa

Testy widoków gościa są w swojej budowie nieskomplikowane. Sprawdzane jest jedynie czy do danych widoków dostęp ma każdy. Osiągnięto to przez sprawdzenie kodu statusu HTTP[37]. Zwrócenie kodu 200 oznacza, że o widok może zapytać każdy i dostanie odpowiedź. Przykład takiego testu przedstawiono w listingu 6.1.

Listing 6.1: Test widoku gościa - zapytanie o listę pokojów

---

```
1 def test_get_rooms(self):
2     """
3     Ensure that it is publically available to get room offers.
4     """
5     url = reverse('get_all_rooms')
6     response = self.client.get(url, format='json')
7     self.assertEqual(response.status_code, status.HTTP_200_OK)
```

---

Nieco bardziej złożonym testem dla gościa jest test rejestracji. Oprócz statusu, należy sprawdzić czy pojawił się odpowiedni rekord w bazie danych. Test rejestracji widoczny jest w listingu 6.2.

Listing 6.2: Test rejestracji

---

```
1 def test_user_registration(self):
2     url = reverse('register')
3     data = {"username": "example", "password": "gjkfnjkgfnjkgnfjk",
4             "password_confirm": "gjkfnjkgfnjkgnfjk", "email": "a@niepodam.pl"}
5     response = self.client.post(url, data, format='json')
6     self.assertEqual(response.status_code, status.HTTP_201_CREATED)
7     self.assertEqual(User.objects.get().username, 'example')
```

---

## 6.1.2. Testy widoków użytkownika

Testy widoków użytkownika są bardziej skomplikowane. Do prawidłowego działania niezbędne było stworzenie funkcji pomocniczych dla testów. Były one odpowiedzialne za logowanie i dodawanie testowych danych do pustej bazy. W testach sprawdzano zwrócony kod statusu HTTP i stan bazy danych. Przykładowy test widoku przeznaczonego dla użytkownika został przedstawiony w listingu 6.3.

Listing 6.3: Test widoku użytkownika (dodanie współlokatora)

---

```

1 def test_add_mate(self):
2     self._get_token()
3     self._add_mate()
4     self.assertEqual(self.response.status_code, status.HTTP_201_CREATED)
5     self.assertEqual(MateOffer.objects.count(), 1)
6     self.assertEqual(MateOffer.objects.get().phone, '123456789')

```

---

## 6.2. Testy części klienckiej aplikacji

Aplikację kliencką przetestowano używając biblioteki enzyme[38]. Zawiera ona niezbędne narzędzia potrzebne do jednostkowego tesowania komponentów. Przykładem może być umożliwienie płatkiego renderowania, renderującego tylko dany komponent, bez komponentów w nim zagnieźdzonych.

Bazę stanowią proste testy, sprawdzające czy komponent się wyrenderuje, lub czy zostanie w nim zawarte przekazane property. Przykład takich testów znajduje się w listingu 6.4.

Listing 6.4: Podstawowe testy frontendowe

---

```

1 it('renders without crashing', () => {
2     shallow(<ChatMessage/>);
3 });
4
5 it('renders message content', () => {
6     const component = shallow(<ChatMessage content="example" />);
7     expect(component.text()).toContain('example');
8 });
9
10 it('has paragraph with text', () => {
11     const component = shallow(<ChatMessage content="example" />);
12     expect(component.containsMatchingElement(<p>example </p>)).toEqual(true);
13 });

```

---

Bardziej skomplikowane okazały się testy komponentów, które pobierają dane z API. W tym przypadku została wykorzystana metoda enzyme dla płatkiego komponentu: setState. Pozwala ona sztucznie ustawić stan komponentu. W listingu 6.5 ustawiana jest przykładowa odpowiedź API jako stan, a następnie sprawdzane jest, czy stan został prawidłowo przekazany do zagnieżdzonego komponentu.

Listing 6.5: Test emulujący zapytanie do API

---

```

1 it('checks if it passes props correctly', () => {
2     const component = shallow(<MateDetailPage />);
3     component.setState({
4         data: {
5             id: 9, title: "Peaceful IT student", age: 21,
6             location: "Grunwald", field_of_study: "Computer Science",
7             features: "peaceful;quiet;gaming;cycling",
8             customs: "no smoking;no partying;wakes up at 11–12;goes to bed 23–24",
9             phone: "123123123", owner: 5
10        }
11    });
12    expect(component.find('ContactBox').prop('phone')).toEqual('123123123');
13 })

```

---

## Rozdział 7

# Wdrożenie projektu

Proces wdrożenia aplikacji można obecnie przeprowadzić na wiele sposobów. Istnieje wiele platform[39], które znacznie ułatwiają proces wdrażania, jednak takie narzędzia jednocześnie dają mniej kontroli nad technologiami i konfiguracją. Z tego powodu, zdecydowano na samodzielne wdrożenie na wirtualnym serwerze prywatnym.

### 7.1. Plan wdrożenia

Wdrożenie produkcyjne można podzielić na etapy widoczne w podsekcjach.

#### 7.1.1. Kupno serwera

Serwer musiał spełniać poniższe parametry:

- co najmniej 10GB dostępnego miejsca na dysku - ze względu na użycie Dockera, rozmiar aplikacji sięga 8.8GB,
- możliwość zainstalowania systemu Debian lub Ubuntu,
- 2GB pamięci RAM - dla komfortowej pracy,
- niezawodność na poziomie 99% lub lepszą, gwarantowaną przez SLA(Service Level Agreement)[40].

Poniższe wymagania spełnia m.in. oferta 'Starter' firmy OVH. Ze względu na wcześniejsze pozytywne doświadczenia, zakupiony został serwer tej firmy.

#### 7.1.2. Konfiguracja i instalacja niezbędnych programów

Konfiguracja nowego serwera sprowadziła się do dwóch czynności:

- przypisanie domeny do adresu IP, poprzez dodanie wpisu w DNS,
- dodanie klucza ssh, dla wygodnego i bezpiecznego logowania.

Niezbędne do wdrożenia programy:

- vim - edytor tekstu,
- nginx - serwer HTTP,
- docker-compose.

#### 7.1.3. Uruchomienie aplikacji w sposób produkcyjny

Uruchomienie aplikacji na serwerze produkcyjnym można podzielić na dwie części. Pierwszą rzeczą, którą należało zrobić, było uruchomienie aplikacji serwerowej w sposób wydajny i

bezpieczny. Wykorzystany został gunicorn, z konfiguracją widoczną na listingu 7.1. Skonfigurowane zostały ścieżki logowania, przypisanie do portu i maksymalna ilość workerów.

Listing 7.1: Konfiguracja gunicorna

---

```

1 from multiprocessing import cpu_count
2
3
4 def max_workers():
5     return cpu_count()
6
7
8 bind = '0.0.0.0:8000'
9 workers = max_workers()
10 access_logfile = '/tmp/app_gunicorn_access.log'
11 error_logfile = '/tmp/app_gunicorn_errors.log'
12 max_requests = 1000

```

---

Drugą częścią zadania jest serwowanie aplikacji klienckiej w sposób produkcyjny. W tym celu, zalecane jest użycie komendy, która całą aplikację umieszcza w kilku zminifikowanych plikach statycznych[41]. Tak wyeksportowane pliki można serwować za pomocą serwera http, czyli w przypadku projektu nginx. Konfiguracja nginx została przedstawiona w listingu 7.2.

Listing 7.2: Konfiguracja nginxa

---

```

1 server {
2     listen 80;
3     server_name 51.83.130.30 finder.tscode.net www.finder.tscode.net;
4     root /home/ubuntu/apps/finder/frontend/build;
5     index index.html index.htm;
6
7     location / {
8         try_files $uri /index.html =404;
9     }
10 }

```

---

Ostatnim krokiem, spinającym obie części jest implementacja usługi, która będzie nadzorowała działanie aplikacji. W razie błędu aplikacji, lub restartu serwera, aplikacja zostanie zresetowana. Usługa systemd jest dobrym rozwiązaniem, także dlatego, że znając nazwę serwisu można go uruchomić lub zamknąć, nie zagłębiając się w szczegóły implementacyjne. Usługa dla aplikacji została przedstawiona w listingu 7.3.

Listing 7.3: Konfiguracja usługi systemd

---

```

1 Description=app for finding flatmates and student quarters
2 Wants=network-online.target
3 After=network-online.target
4
5 [Service]
6 Type=simple
7 WorkingDirectory=/home/ubuntu/apps/finder
8 ExecStart=/usr/bin/docker-compose up
9 Restart=always
10
11 [Install]
12 WantedBy=multi-user.target

```

---

W przypadku usługi komendami sterującymi będą polecenia widoczne w listingu 7.4

Listing 7.4: Komendy sterujące

---

```

1 # status aplikacji
2 sudo systemctl status finder.service
3
4 # start aplikacji
5 sudo systemctl start finder.service
6
7 # zatrzymanie aplikacji
8 sudo systemctl stop finder.service

```

---

# Rozdział 8

## Aplikacja mobilna

W związku z szybkim postępem prac, postanowiono dodatkowo zaimplementować aplikację mobilną. Posiada ona funkcjonalności związane z operacjami na współlokalatorach. Aplikacja przeznaczona jest na urządzenia mobilne działające pod kontrolą systemu operacyjnego Android.

### 8.1. Architektura

Aplikacja została zaimplementowana przy pomocy narzędzia React Native. Pozwala on na użycie języka Javascript i mechanizmów Reacta, przy jednoczesnym wykorzystaniu natywnych komponentów jak inputy czy przyciski[42]. Ze względu na doświadczenie z pracy nad aplikacją przeglądarkową, zdecydowano o przestrzeganiu poniższych założeń architektonicznych:

1. wykorzystanie komponentów funkcyjnych i hooków zamiast komponentów klasowych.  
Powodem takiego wyboru jest zwięzła struktura kodu i fakt, iż zewnętrzne biblioteki często oferują potrzebne hooki, których nie można użyć w komponentach klasowych. Doświadczenie pokazało także, że używanie hooka useEffect jest także lepszą formą kontroli cyklu życia, gdyż w czasie procesu implementacji, niektóre metody cyklu życia zdążyły otrzymać miano "UNSAFE"[43],
2. szczegółowy podział komponentów, w celu ułatwienia nawigacji w projekcie. Komponenty wygodnie jest podzielić według spełnianych funkcji. W przypadku aplikacji mobilnej nastąpił podział na kontenery, opakowujące jedynie treść, i komponenty stanowiące treść, jak inputy, przyciski, etc. Taki podział ułatwia orientację w projekcie i wymusza większe rozbicie na komponenty. Struktura komponentów została przedstawiona w listingu 8.1.

Listing 8.1: Struktura plików komponentów

```
1 components /  
2   └── boxes  
3     ├── cloud-box.component.js  
4     ├── grey-box.component.js  
5     ├── list-card.component.js  
6     ├── ...  
7     └── start-box.component.js  
8   └── content  
9     ├── big-button.component.js  
10    ├── big-input.component.js  
11    └── burger.component.js
```

---

12     └ ...  
13     └ small-button.component.js

---

## 8.2. Implementacja

W ogólnych założeniach implementacja nie odbiega znacząco od części webowej. Tworzone są komponenty wielokrotnego użytku, w komponentach możliwe jest zarządzanie stanem, a dane pobierane są z części serwerowej projektu za pomocą biblioteki axios. Stosowany jest podobny mechanizm routingu, tym razem dostarczany przez bibliotekę react-navigation. Znaczące różnice pojawiają się przy charakterystyce podstawowych komponentów, używaniu stylów i zarządzaniu stanem i cyklem życia za pomocą hooków.

### 8.2.1. Podstawowe komponenty

Jak zaznaczono we wstępnie, React Native korzysta z natywnych komponentów. Komponenty te działają w inny sposób niż komponenty webowe (np. dane tekstowe mogą znajdować się jedynie wewnątrz komponentu Text). Można jednak powiązać np. działanie komponentu natywnego View z komponentem webowym div, czy np. Image z img[44]. Przykładowy komponent został przedstawiony w listingu 8.2.

Listing 8.2: Przykładowy komponent

---

```

1 import React from 'react';
2 import { View, StyleSheet, Text, TouchableOpacity } from 'react-native';
3 // ...
4
5 const styles = StyleSheet.create({
6   // ...
7 })
8
9 export const MessageCard = (props) => {
10   const navigation = useNavigation();
11
12   return (
13     <TouchableOpacity
14       activeOpacity={0.5}
15       style={styles.container}
16       onPress={() => {
17         navigation.navigate('MessageDetail', { id: props.id });
18       }}>
19       <View>
20         <Text style={styles.head}>from</Text>
21         <Text>{props.email}</Text>
22       </View>
23
24       <View>
25         <Text style={styles.head}>message </Text>
26         <Text style={styles.subject}>{props.subject}</Text>
27         <Text style={styles.content}>{props.last_message}</Text>
28       </View>
29     </TouchableOpacity>
30   )
31 }
```

---

### 8.2.2. Style w React-Native

Chociaż wcześniej używana biblioteka styled-components oferuje wsparcie dla React Native, tak dokumentacja w tym temacie jest skromna[45], i poświęcony zagadnieniu paragraf nie był pomocny przy problemach napotkanych podczas próby integracji biblioteki z aplikacją.

W związku z powyższym wykorzystano wbudowane narzędzie React Native - StyleSheet. Przykładowe style zostały pokazane w listingu 2

Listing 8.3: Przykładowy StyleSheet

```

1 const styles = StyleSheet.create({
2   container: {
3     alignItems: 'center',
4   },
5
6   form_wrapper: {
7     width: '80%'
8   },
9
10  button_wrapper: {
11    marginVertical: 15
12  },
13
14  profile_photo: {
15    width: 100,
16    height: 100,
17    marginVertical: 10
18  },
19
20  photo_button: {
21    backgroundColor: 'lightgray',
22    alignItems: "center",
23    justifyContent: 'center',
24    width: 100,
25    height: 40,
26    marginVertical: 10,
27    borderRadius: 5
28  },
29
30  button_text: {
31    color: 'black'
32  }
33})

```

Co warto zaznaczyć, style działają w sposób podobny, aczkolwiek nie identyczny jak te stosowane w stronach internetowych. Domyślną formą porządkowania elementów jest flexbox[46]. Flexbox oferowany w React Native też różni się w detaluach od tego znanego z css, m.in innymi wartościami domyślnymi. Sama składnia StyleSheet jest podobna do css, z wyjątkiem użycia camelCase, cudzysłów i przecinków.

### 8.2.3. Zarządzanie stanem i cyklem życia

Zarządzanie stanem i cyklem życia w komponentach funkcyjnych odbiega od wzorców stosowanych w komponentach klasowych. Jeśli chodzi o zarządzania stanem, użyty został hook useState. Zwraca on zmienną przechowującą lokalny stan i funkcję do jego aktualizacji[47]. Użytkowanie jest zatem podobne do obiektu this.state i metody setState z komponentu klasowego.

Do zarządzania cyklem życia komponentu użyty został hook useEffect. Hook efektów pozwala na przeprowadzanie efektów ubocznych w komponentach funkcyjnych[48]. Domyślnie uruchamiany jest przy każdym renderowaniu strony, ale jest możliwość ograniczenia renderowania w zależności od zmiany parametrów. Zachowanie takie można otrzymać poprzez dodanie tablicy ze zmiennymi, od których hook będzie zależny. Wtedy, hook się uaktywni tylko gdy zmieni się któraś ze zmiennych w tablicy[49].

Przykład zastosowania wymienionych wyżej hooków został przedstawiony w listingu 8.4. Za pomocą useEffect pobierane są token i dane do wyrenderowania na stronie, zaś za pomocą useState zmieniany jest stan strony.

Listing 8.4: Użycie useState i useEffect

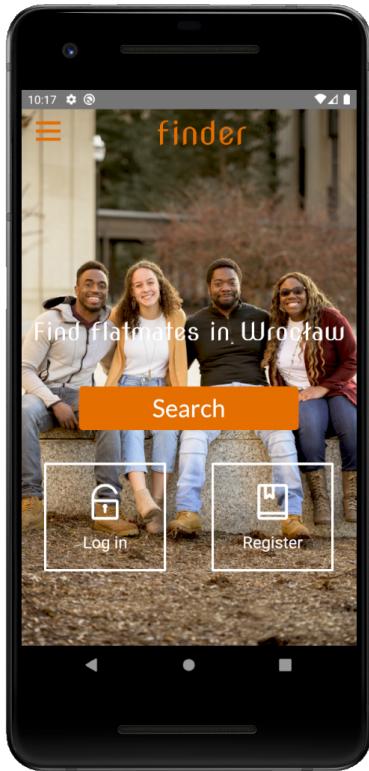
```

1 import React, { useState, useEffect } from 'react';
2 import axios from 'axios';
3 // ...
4
5 const styles = StyleSheet.create({
6 // ...
7 })
8
9 export const DetailScreen = (props) => {
10   const [data, setData] = useState({});
11   const [token, setToken] = useState('');
12   const id = props.route.params.id;
13   const url = `${static_host}/mate_offer_detail/${id}`;
14
15   // getting token
16   useEffect(() => {
17     const fetchToken = async () => {
18       try {
19         const t = await getToken()
20         setToken(t);
21       } catch (e) {
22         console.log('Błąd')
23       }
24     }
25     fetchToken();
26   }, []);
27
28   // getting data
29   useEffect(() => {
30     const fetchData = async () => {
31       const result = await axios.get(url);
32       setData(result.data);
33     }
34     fetchData();
35   }, [url]);
36
37   let features = <Text>{ '' }</Text>;
38   let customs = <Text>{ '' }</Text>;
39
40   if (!isEmpty(data)) {
41     features = data.features.split(';').map((element, index) => (
42       <Text style={styles.section_content} key={index}>{element}</Text>
43     ))
44     customs = data.customs.split(';').map((element, index) => (
45       <Text style={styles.section_content} key={index}>{element}</Text>
46     ))
47   }
48
49   return (
50     <GreyBox>
51     // ...
52     {features}
53     // ...
54     {customs}
55
56     // ...
57   </GreyBox>
58 )
59 }
```

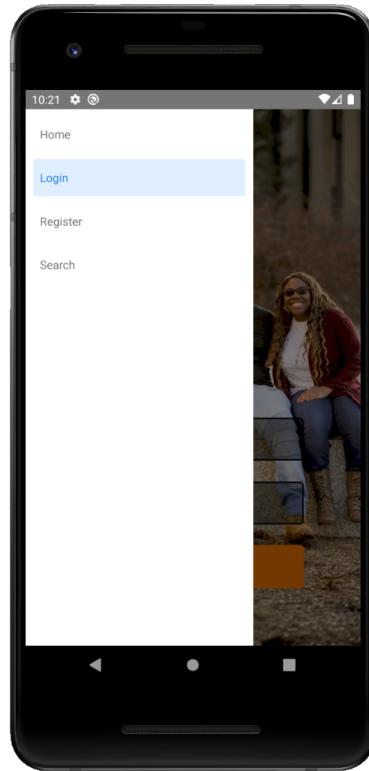
## 8.3. Wygląd wybranych ekranów

Na poniższych rysunkach przedstawiony jest wygląd wybranych ekranów aplikacji mobilnej Finder.

Rysunek 8.1 to strona początkowa dla gości. Wszystkie możliwe opcje nawigacji, czyli przejście do poszukiwania, loginu lub rejestracji dostępne są zarówno na stronie głównej, jak i w menu typu drawer, przedstawionym na rysunku 8.2.

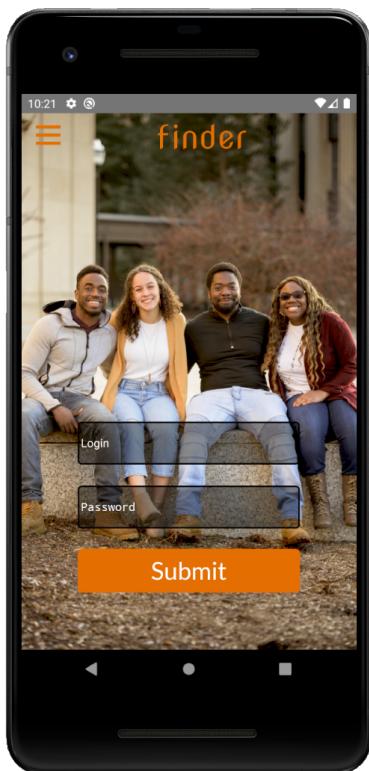


Rys. 8.1: Ekran startowy gościa (landing)

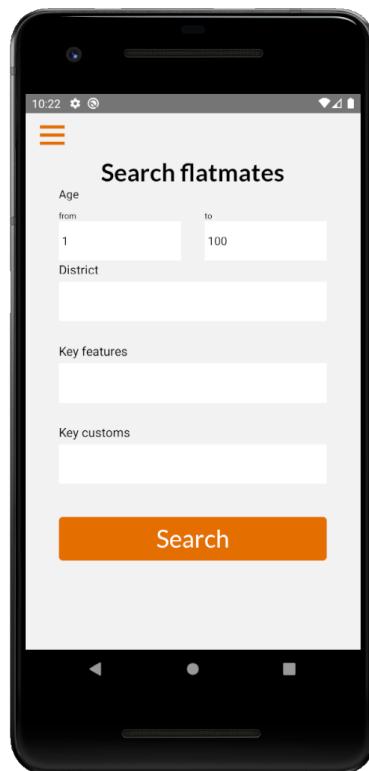


Rys. 8.2: Nawigacja typu drawer dla gościa

Na rysunku 8.3 widoczny jest ekran logowania. Bliźniaczy ekran, z odpowiednimi polami służy do rejestracji. Na rysunku 8.4 widoczny jest ekran służący do zadawania parametrów przy szukaniu współlokatorów.

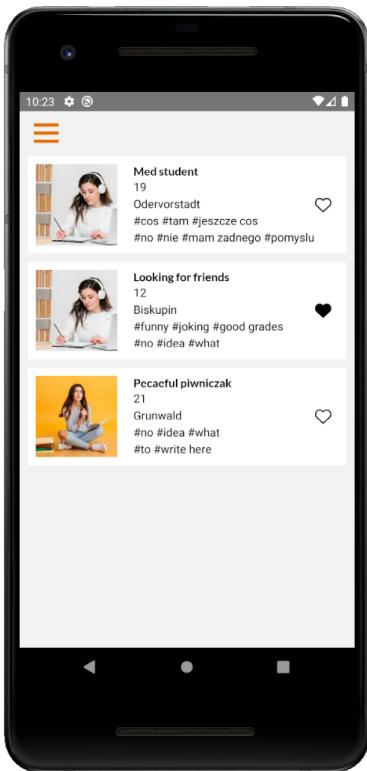


Rys. 8.3: Ekran logowania

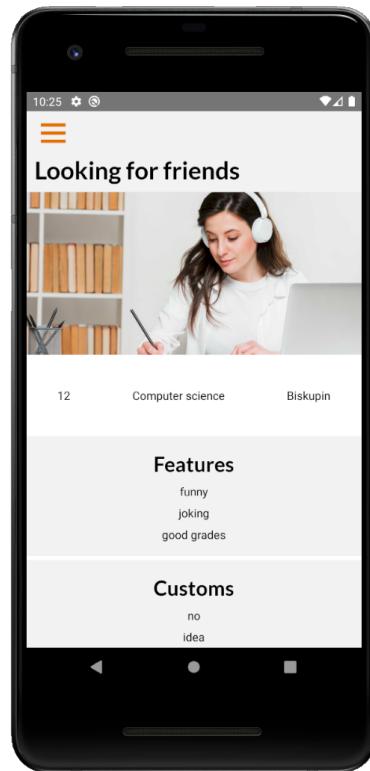


Rys. 8.4: Ekran poszukiwania

Ekran 8.5 przedstawia widok listy ofert. Każda z ofert posiada możliwość dodania do ulubionych, jeśli użytkownik jest zalogowany. Ekran 8.6 przedstawia widok detaliczny oferty, do którego można przejść poprzez naciśnięcie na któryś z elementów na liście.

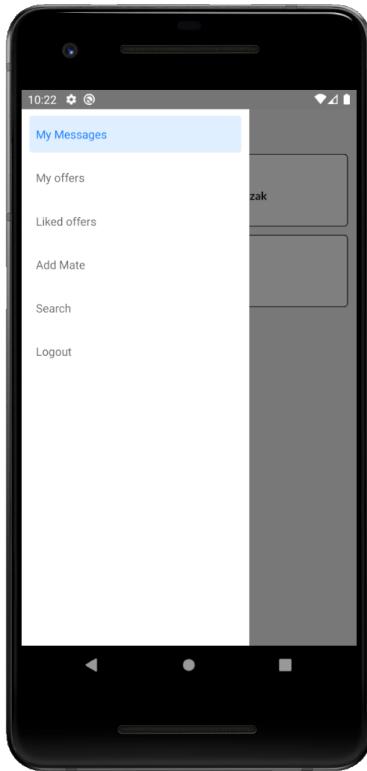


Rys. 8.5: Ekran z listą wyszukanych ofert

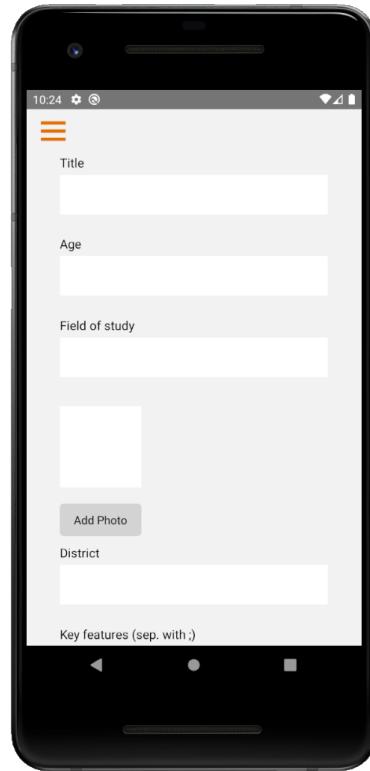


Rys. 8.6: Ekran szczegółowy oferty

Ekran 8.7 pokazuje możliwości nawigacji, jakie posiada użytkownik. Zaraz po zalogowaniu zachodzi w menu zmiana i np. znika możliwość dostania się do widoku loginu lub rejestracji. Ekran 8.8 przedstawia część interfejsu umożliwiającą dodanie lub edycję oferty. W zależności o naciśniętego przycisku w menu użytkownika, ekran jest pusty, i możliwe jest dodanie nowej oferty, lub w przypadku naciśnięcia przycisku edycji oferty, zapełniany jest dotychczasowymi danymi.

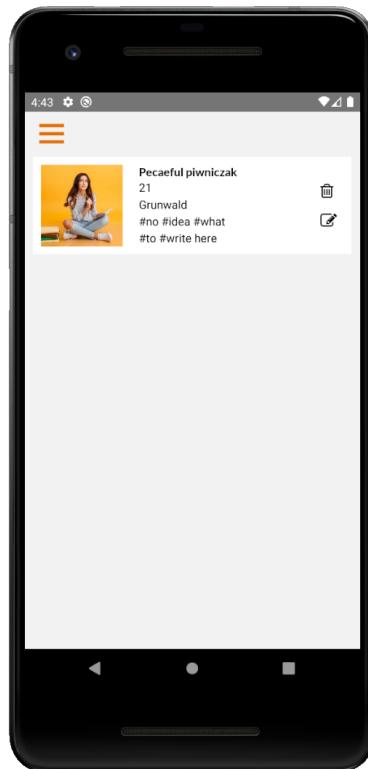


Rys. 8.7: Nawigacja typu drawer dla użytkownika



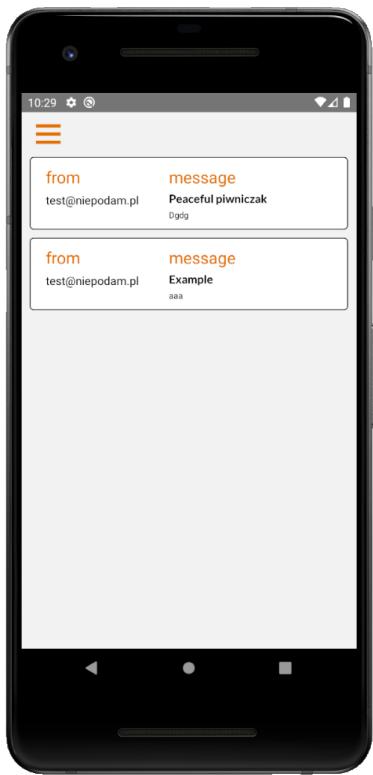
Rys. 8.8: Ekran dodawania i edycji oferty

Ekran 8.9 przedstawia oferty użytkownika. Ikonki z prawej strony umożliwiają edycję lub usunięcie oferty.

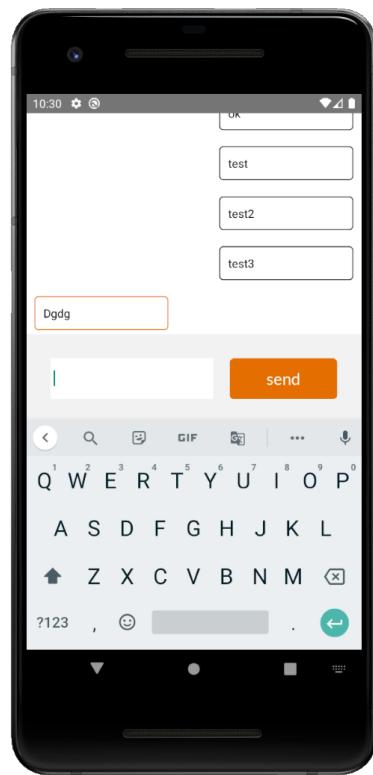


Rys. 8.9: Lista ofert użytkownika

Ekrany 8.10 i 8.11 to odpowiednio lista konwersacji i konwersacja. Po kliknięciu odpowiedniej konwersacji, następuje przekierowanie do widoku detalicznego.



Rys. 8.10: Ekran z listą konwersacji



Rys. 8.11: Ekran szczegółowy konwersacji

# Rozdział 9

## Podsumowanie

Podsumowując projekt, można stwierdzić że spełnił on postawione przed nim wymagania. Aplikacja jest w pełni funkcjonalna, oraz działa w środowisku produkcyjnym.

Podczas procesu implementacji aplikacji zostało zebranych wiele doświadczeń. Pewne wzorce architektoniczne okazały się w trakcie realizacji mało elastyczne, a ich zastąpienie czasochłonne. Potencjalnymi ulepszeniami mogą być:

- podział kodu widoków części serwerowej. Obecnie wszystkie widoki znajdują się w jednym pliku liczącym blisko 400 linii,
- Skorzystanie z komponentów funkcyjnych w kliencie przeglądarkowym. W trakcie realizacji część metod cyklu życia stała się przestarzała. React zdaje się preferować użycie komponentów funkcyjnych i hooków,
- skorzystanie z ContextApi w kliencie przeglądarkowym (przeglądarkowej). Drzewo komponentów renderujące okienka do rejestracji i logowania jest bardzo głębokie, i stan jest przekazywany przez wiele komponentów. Z perspektywy dnia dzisiejszego lepszym pomysłem byłoby skorzystanie z ContextApi lub Reduxa,
- przetestowanie aplikacji mobilnej na urządzeniu pracującym pod kontrolą systemu iOS. Uzyskanie działania na drugim najpopularniejszym systemie nie powinno sprawić dużych problemów. Z dużą dozą prawdopodobieństwa wystarczy jedynie zastąpić kilka elementów specyficznych dla platformy Android innymi rozwiązaniami. Obecnie jedyną przeszkodą w realizacji jest nieposiadanie odpowiedniego urządzenia (komputera lub smartfona firmy Apple),
- rozważenie możliwości uzyskania połączenia z częścią serwerową tylko za pośrednictwem serwera http (nginxa). Wraz z dodaniem aplikacji mobilnej, API zostało upublicznione i każdy może wysłać do niego żądanie. Ze względów wydajnościowych, wraz ze wzrostem obciążenia konieczne może być wykorzystanie nginxa.

Dodatkowym celem projektu była nauka technologii pozwalających na realizację Single Page Application, aplikacji natywnych i zgłębienie architektury REST. Dla autora był to pierwszy, w pełni samodzielnie zrealizowany projekt aplikacji w architekturze klient-serwer. Pozwolił wywieść cenne doświadczenia, które przydadzą się przy projektowaniu kolejnych aplikacji serwerowych i klienckich.

# Bibliografia

- [1] *What is DBMS? Definition and FAQs.* <https://www.omnisci.com/technical-glossary/dbms> [dostęp dnia 3 grudnia 2020].
- [2] *Analiza wymagań.* [https://www.is.umk.pl/~grochu/wiki/doku.php?id=zajecia:ppz2:analiza\\_wymagan](https://www.is.umk.pl/~grochu/wiki/doku.php?id=zajecia:ppz2:analiza_wymagan) [dostęp dnia 18 listopada 2020].
- [3] Arun Ravindran Samuel Dauzon Aidas Bendoraitis. *Django: Web Development with Python.* Packt Publishing Ltd., 2016. ISBN: 978-1-78712-138-6.
- [4] Michael Stowe. *Undisturbed REST: A guide to designing the perfect API.* MuleSoft, 2015. ISBN: 978-1-329-11656-6.
- [5] *SPA (Single-page application).* <https://developer.mozilla.org/en-US/docs/Glossary/SPA> [dostęp dnia 18 listopada 2020].
- [6] Surjith S M. *How single page applications enhance the User experience.* <https://web3canvas.com/how-single-page-applications-enhance-the-user-experience/> [dostęp dnia 1 września 2020].
- [7] *ACID Compliance: What It Means and Why You Should Care.* <https://mariadb.com/resources/blog/acid-compliance-what-it-means-and-why-you-should-care/> [dostęp dnia 18 listopada 2020].
- [8] <https://github.com/postgres/postgres> [dostęp dnia 1 września 2020].
- [9] Sveta Smirnova Anastasia Raspopina. *PostgreSQL and MySQL: Millions of Queries per Second.* <https://www.percona.com/blog/2017/01/06/millions-queries-per-second-postgresql-and-mysql-peaceful-battle-at-modern-demanding-workloads/> [dostęp dnia 1 września 2020].
- [10] Navnath Gadakh. *Why Django is so impressive for developing with PostgreSQL and Python.* <https://restfulapi.net/> [dostęp dnia 1 września 2020].
- [11] *What does “micro” mean?* <https://flask.palletsprojects.com/en/1.1.x/foreword/#what-does-micro-mean> [dostęp dnia 1 września 2020].
- [12] *Systemy ORM.* <https://python101.readthedocs.io/pl/latest/bazy/orm/> [dostęp dnia 30 września 2020].
- [13] *Front-end frameworks popularity.* <https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190> [dostęp dnia 1 września 2020].
- [14] *Why Docker?* <https://www.docker.com/why-docker> [dostęp dnia 1 września 2020].
- [15] *Overview of Docker Compose.* <https://docs.docker.com/compose/> [dostęp dnia 1 września 2020].
- [16] *WSGI Servers.* <https://www.fullstackpython.com/wsgi-servers.html> [dostęp dnia 1 września 2020].

- [17] *systemd.service*. <https://www.freedesktop.org/software/systemd/man/systemd.service.html> [dostęp dnia 1 września 2020].
- [18] *Migrations*. <https://docs.djangoproject.com/en/3.1/topics/migrations/> [dostęp dnia 1 września 2020].
- [19] *Writing views*. <https://docs.djangoproject.com/en/3.1/topics/http/views/> [dostęp dnia 1 września 2020].
- [20] *Class based views - Django REST Framework*. <https://www.djangoproject-rest-framework.org/tutorial/3-class-based-views/> [dostęp dnia 3 grudnia 2020].
- [21] *Serializers*. <https://www.django-rest-framework.org/api-guide/serializers/#serializers> [dostęp dnia 30 września 2020].
- [22] *Password management in Django*. <https://docs.djangoproject.com/en/3.1/topics/auth/passwords/#how-django-stores-passwords> [dostęp dnia 30 września 2020].
- [23] *django.contrib.auth*. <https://docs.djangoproject.com/en/3.1/topics/auth/passwords/#how-django-stores-passwords> [dostęp dnia 30 września 2020].
- [24] *Quickstart*. <https://django-rest-registration.readthedocs.io/en/latest/quickstart.html> [dostęp dnia 30 września 2020].
- [25] *Introduction to JSON Web Tokens*. <https://django-rest-registration.readthedocs.io/en/latest/quickstart.html> [dostęp dnia 7 października 2020].
- [26] *Cross-Origin Resource Sharing (CORS)*. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> [dostęp dnia 7 października 2020].
- [27] *django-cors-headers*. <https://github.com/adamchainz/django-cors-headers> [dostęp dnia 3 grudnia 2020].
- [28] *Komponenty i właściwości*. <https://pl.reactjs.org/docs/components-and-props.html> [dostęp dnia 7 października 2020].
- [29] *Basics*. <https://styled-components.com/docs/basics#motivation> [dostęp dnia 7 października 2020].
- [30] *CSS Variables for React Devs*. <https://joshwcomeau.com/css/css-variables-for-react-devs/#quick-intro> [dostęp dnia 7 października 2020].
- [31] *Basics*. <https://styled-components.com/docs/basics#pseudoelements-pseudoselectors-and-nesting> [dostęp dnia 7 października 2020].
- [32] Faraz Kelhini. *Axios or fetch(): Which should you use?* <https://joshwcomeau.com/css/css-variables-for-react-devs/#quick-intro> [dostęp dnia 8 października 2020].
- [33] *Tutorial. Overview: What is Formik?* <https://formik.org/docs/tutorial#overview-what-is-formik> [dostęp dnia 8 października 2020].
- [34] *Yup*. <https://github.com/jquense/yup#yup> [dostęp dnia 8 października 2020].
- [35] *React Router: Declarative Routing for React.js*. <https://reactrouter.com/> [dostęp dnia 8 października 2020].
- [36] Rodney Parkin. "Software Unit Testing". W: (1997), s. 1–1.
- [37] "HTTP response status codes". W: (). <https://developer.mozilla.org/en-US/docs/Web/HTTP>Status> [dostęp dnia 14 października 2020].

- [38] *Introduction: Enzyme.* <https://enzymejs.github.io/enzyme/> [dostęp dnia 15 października 2020].
- [39] Ashutosh Singh. *8 ways to deploy a React app for free.* <https://blog.logrocket.com/8-ways-to-deploy-a-react-app-for-free/> [dostęp dnia 20 października 2020].
- [40] *Service level agreement (SLA).* <https://searchitchannel.techtarget.com/definition/service-level-agreement> [dostęp dnia 20 października 2020].
- [41] *Optymalizacja wydajności.* <https://pl.reactjs.org/docs/optimizing-performance.html#create-react-app> [dostęp dnia 20 października 2020].
- [42] *React Native. Learn once, write anywhere.* <https://reactnative.dev/> [dostęp dnia 17 listopada 2020].
- [43] *Przestarzałe metody cyklu życia.* <https://pl.reactjs.org/docs/react-component.html#legacy-lifecycle-methods> [dostęp dnia 17 listopada 2020].
- [44] *Core components and Native components.* <https://reactnative.dev/docs/intro-react-native-components> [dostęp dnia 17 listopada 2020].
- [45] *Basics.* <https://styled-components.com/docs/basics#react-native> [dostęp dnia 17 listopada 2020].
- [46] *Layout with Flexbox.* <https://reactnative.dev/docs/flexbox> [dostęp dnia 17 listopada 2020].
- [47] *Hooki - interfejs API.* <https://pl.reactjs.org/docs/hooks-reference.html#usestate> [dostęp dnia 18 listopada 2020].
- [48] *Hooki - interfejs API.* <https://pl.reactjs.org/docs/hooks-reference.html#useeffect> [dostęp dnia 18 listopada 2020].
- [49] *Hooki - interfejs API.* <https://pl.reactjs.org/docs/hooks-reference.html#conditionally-firing-an-effect> [dostęp dnia 18 listopada 2020].