Rozdział 8

Aplikacja mobilna

W związku z szybkim postępem prac, postanowiono dodatkowo zaimplementować aplikację mobilną. Posiada ona funkcjonalności związane z operacjami na współlokatorach. 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 doświadczenia 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: Stuktura plików komponentów

```
components/
├── boxes
│   ├── cloud-box.component.js
│   ├── grey-box.component.js
│   ├── list-card.component.js
│   ├── ...
│   └── start-box.component.js
└── content
    ├── big-button.component.js
    ├── big-input.component.js
    ├── 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 bibliotkę react-navigation. Znaczące różnice pojawiają się przy charakterystyce podstawowych komponentów, używaniu styli i zarządzaniu stanem i cyklem życia za pomocą hooków.

8.2.1. Podstawowe komponenty

Jak zaznaczono we wstępie, 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

```
const styles = StyleSheet.create({
  container: {
    alignItems: 'center',
  },

  form_wrapper: {
    width: '80%'
  },

  button_wrapper: {
    marginVertical: 15
  },

  profile_photo: {
    width: 100,
    height: 100,
    marginVertical: 10
  },

  photo_button: {
    backgroundColor: 'lightgray',
    alignItems: "center",
    justifyContent: 'center',
    width: 100,
    height: 40,
    marginVertical: 10,
    borderRadius: 5
  },

  button_text: {
    color: 'black'
  }
})
```

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 detalach od tego znanego z css, m.in innymi wartościami domyślnymi. Sama składnia StyleSheet jest podobna do css, z wyjątkiem użycia camelCase, cudzysłowiów 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 zarządzenia 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.

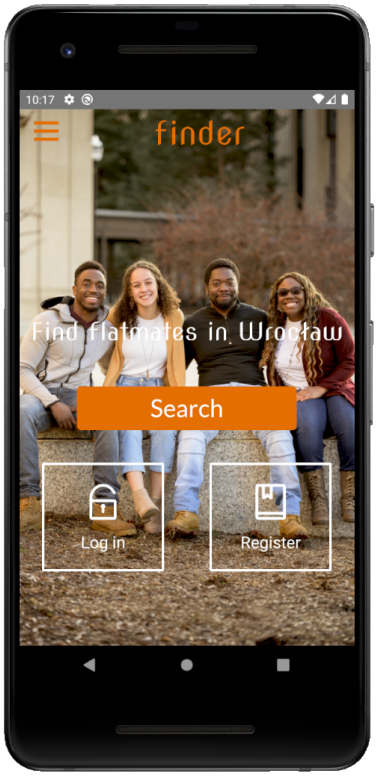
Listing 8.4: Użycie useState i useEffect

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
// ...

const styles = StyleSheet.create({
// ...
})

export const DetailScreen = (props) => {
  const [data, setData] = useState({});
  const [token, setToken] = useState('');
  const id = props.route.params.id;
  const url = `${static_host}/mate_offer_detail/${id}`;

  // getting token
  useEffect(() => {
    const fetchToken = async () => {
      try {
        const t = await getToken()
        setToken(t);
      } catch (e) {
        console.log('Błąd')
      }
    }
    fetchToken();
  }, []);

  // getting data
  useEffect(() => {
    const fetchData = async () => {
      const result = await axios.get(url);
      setData(result.data);
    }
    fetchData();
  }, [url])

  let features = <Text>{''}</Text>;
  let customs = <Text>{''}</Text>;

  if (!isEmpty(data)) {
    features = data.features.split(';').map((element, index) => (
      <Text style={styles.section_content} key={index}>{element}</Text>
    ))
    customs = data.customs.split(';').map((element, index) => (
      <Text style={styles.section_content} key={index}>{element}</Text>
    ))
  }

  return (
    <GreyBox>
    //  ...
        {features}
    //   ...
        {customs}

    // ...
    </GreyBox>
  )
}
```

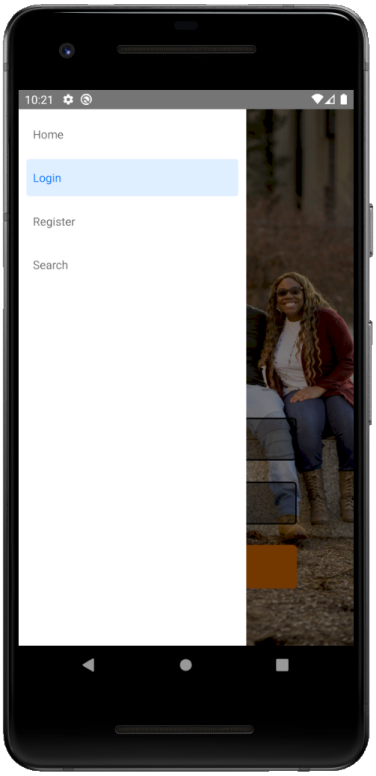
8.3. Wygląd wybranych ekranów

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 początkowej, jak i w menu typu drawer, przedstawionym na rysunku 8.2.

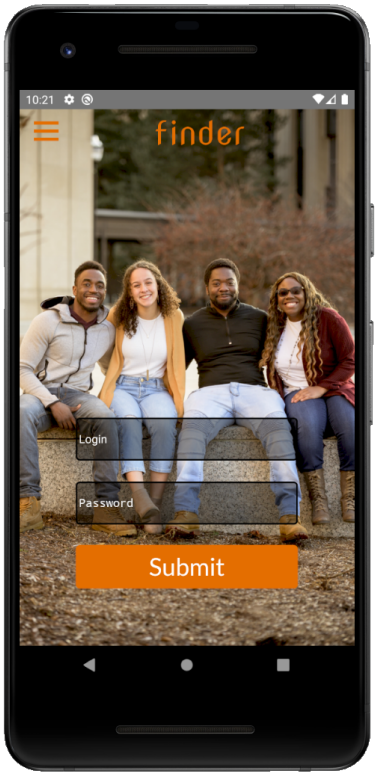


Rys. 8.1: Ekran startowy gościa (landing)

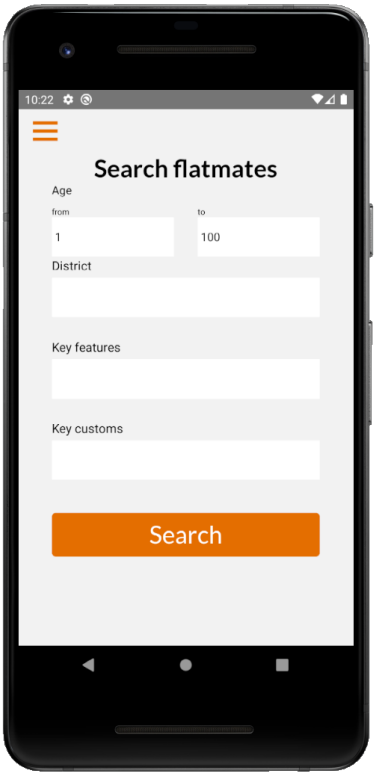


Rys. 8.2: Nawigacja typu drawer dla gościa

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.

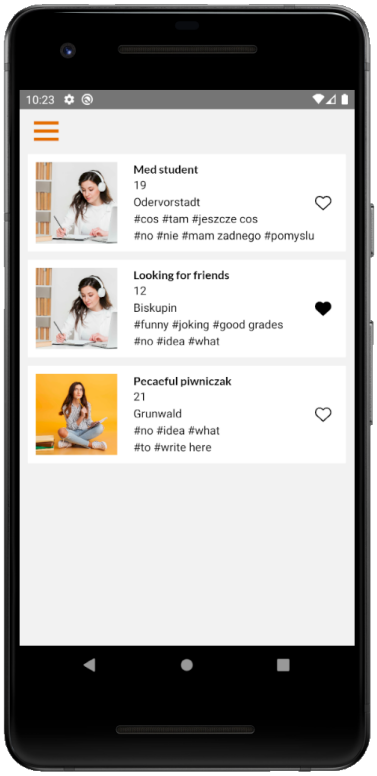


Rys. 8.3: Ekran logowania

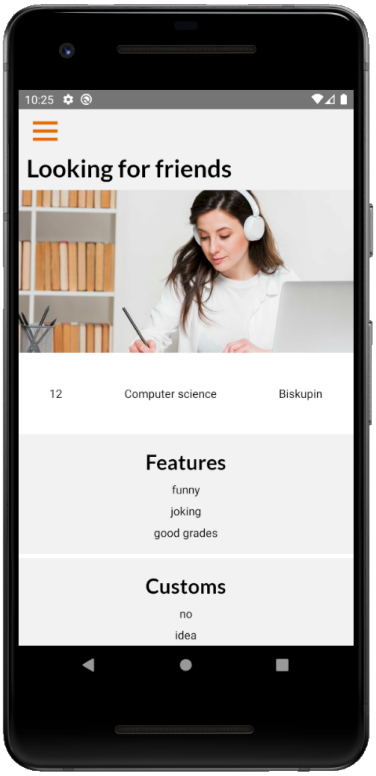


Rys. 8.4: Ekran poszukiwania

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.

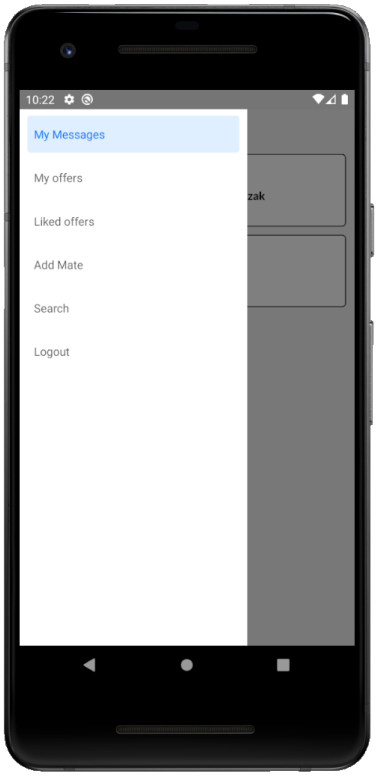


Rys. 8.5: Ekran z listą wyszukanych ofert

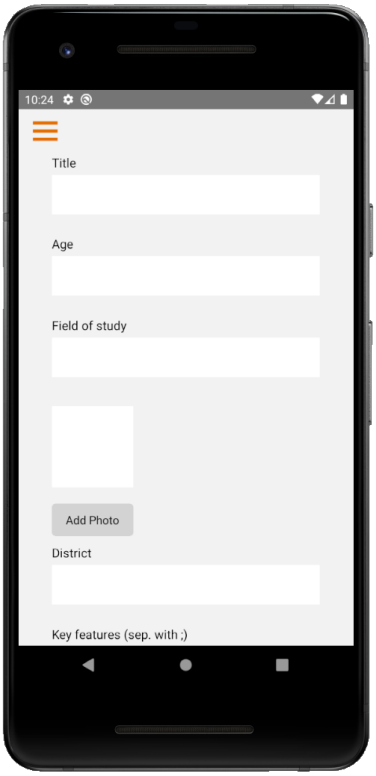


Rys. 8.6: Ekran szczegółowy oferty

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.

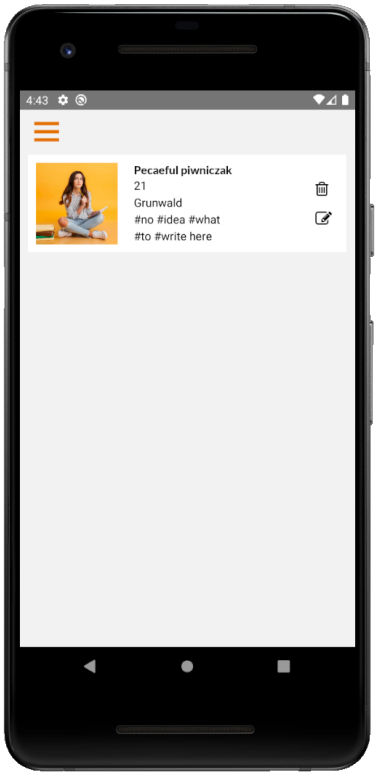


Rys. 8.7: Nawigacja typu drawer dla użytkownika



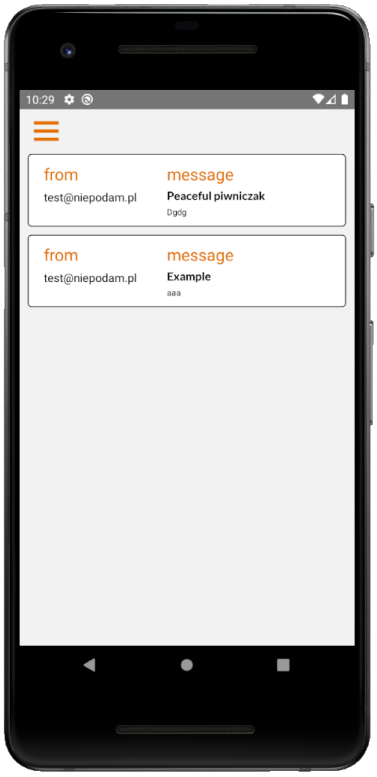
Rys. 8.8: Ekran dodawania i edycji oferty

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

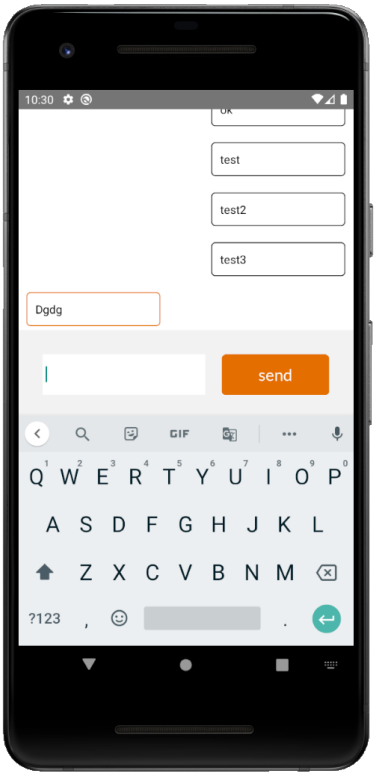


Rys. 8.9: Lista ofert użytkownika

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