



**Politechnika Wrocławska**

---

Wydział Informatyki i Telekomunikacji

---

## **Projektowanie Telemedycznych Systemów Internetowych i Mobilnych**

---

**System rejestracji pacjentów w przychodni  
specjalistycznej**

Paweł Krasicki  
Krystian Ogonowski

Prowadzący: dr inż. Mariusz Topolski  
Grupa zajęciowa: Czwartek TN 15:15

# Spis treści

1	Skład grupy .....	4
2	Opis projektu .....	4
2.1	Cel projektu .....	4
2.2	Zasoby ludzkie .....	4
2.3	System dla pacjentów .....	4
2.4	System dla lekarzy .....	4
2.5	System dla aministratora .....	4
3	Wymagania funkcjonalne .....	5
3.1	Logowanie i zakładanie konta .....	5
3.2	Zgłaszanie potrzeby wizyty przez pacjentów .....	5
3.3	Przeglądanie zaplanowanych wizyt przez pacjentów .....	5
3.4	Przeglądanie zaplanowanych wizyt przez lekarzy .....	5
3.5	Przeglądanie zgłoszeń pacjentów przez lekarzy .....	6
3.6	Przyjmowanie zgłoszeń przez lekarzy, zamieniając je na ustalone wizyty .....	6
3.7	Możliwość zarządzania całym systemem przez administratorów .....	6
4	Wykorzystane technologie i biblioteki .....	7
4.1	Aplikacja webowa .....	7
4.2	Aplikacja mobilna .....	7
4.3	REST API .....	8
4.4	Baza danych .....	8
4.5	Technologie testowania .....	8
5	Harmonogram implementacji .....	10
6	Architektura informacji .....	11
6.1	Utworzenie bazy danych .....	12
6.2	Opis encji i atrybutów w bazie danych .....	13
7	Opis interfejsu programistycznego .....	14
7.1	Modele .....	14
7.2	Endpointy .....	14
7.3	Testowanie API .....	15
7.3.1	Postman .....	15

---

7.3.2 Testy jednostkowe .....	15
8 Graficzny interfejs użytkownika .....	17
8.1 Widok mobilny aplikacji .....	17
8.2 Widok komputerowy aplikacji .....	24
8.3 Relacje pomiędzy widokami .....	24
9 Opis implementacji (Aplikacja internetowa) .....	26
9.1 Redux store .....	26
9.2 Routing .....	27
9.3 Logowanie i zakładanie konta .....	28
9.4 Wymiana danych z REST API .....	28
9.5 Przechowywanie danych .....	29
9.6 Zewnętrzne interfejsy programistyczne .....	29
9.7 Prezentowanie danych .....	29
10 Krytyczna ocena projektu .....	30
10.1 Zrealizowane założenia .....	30
10.2 Dalsze możliwości rozwoju .....	30

---

## 1 Skład grupy

- Krystian Ogonowski
- Paweł Krasicki

## 2 Opis projektu

### 2.1 Cel projektu

Celem realizowanego projektu jest implementacja systemu rejestracji pacjentów w specjalistycznej przychodni lekarskiej, składającego się z aplikacji webowej i mobilnej. Obydwie aplikacje mają być przystosowane do spełniania wymagań pracowników i potencjalnych pacjentów chcących się umówić na wizytę. Tworzony system ma zautomatyzować proces ustalenia wizyty danego pacjenta u konkretnego lekarza, specjalisty. Pozwala on również uczynić ten proces wygodnym i przyjemnym, szczególnie dla pacjenta.

### 2.2 Zasoby ludzkie

- pacjent
- lekarz
- administrator

### 2.3 System dla pacjentów

Dzięki tworzonemu systemowi, dowolna osoba zainteresowana wizytą u lekarza może zgłosić potrzebę wizyty i podać listę objawów, które zauważyła i wpłynęły na jej decyzję. Aplikacja umożliwia pacjentowi:

- przeglądanie umówionych wizyt wraz z terminami i lekarzami
- zgłaszanie potrzeby wizyty u lekarza z możliwością wpisania zaobserwowanych objawów

### 2.4 System dla lekarzy

Dzięki aplikacji, lekarze mogą przejrzeć zgłoszenia pacjentów, ich objawy i przypisać do siebie wybranego pacjenta, którego są w stanie zdiagnozować i wyleczyć. Jednocześnie widzą wszystkie swoje umówione wizyty, więc mogą łatwo wybrać termin kolejnej. Aplikacja umożliwia im:

- przeglądanie zaplanowanych spotkań z pacjentami
- przeglądanie zgłoszeń pacjentów
- obsłużenie danego zgłoszenia w postaci przypisania go sobie z ustalonym terminem
- edycję swoich zaplanowanych wizyt

### 2.5 System dla administratora

Administratorzy mogą tworzyć konta lekarzy i zarządzać nimi przy pomocy narzędzia django-admin. Widzą oni również wszystkie istniejące wizyty i mogą je kontrolować. Aplikacja umożliwia im dostęp do wszystkich funkcjonalności systemu, przeznaczonego zarówno dla pacjentów, jak i lekarzy oraz wszystkich pozostałych opcji edycji, tworzenia i usuwania oraz przeglądania odpowiednich elementów zablokowanych dla pozostałych użytkowników.

---

### 3 Wymagania funkcjonalne

- zakładanie konta
- logowanie
- zgłaszanie potrzeby wizyty przez pacjentów
- przeglądanie zaplanowanych wizyt przez pacjentów
- przeglądanie zaplanowanych wizyt przez lekarzy
- przeglądanie zgłoszeń pacjentów przez lekarzy
- przyjmowanie zgłoszeń przez lekarzy, zamieniając je na ustalone wizyty
- możliwość zarządzania całym systemem przez administratorów

#### 3.1 Logowanie i zakładanie konta

System umożliwia założenie konta, podając email i wybrane hasło, a następnie logowanie się danymi, które wprowadziliśmy podczas zakładania konta. Logowanie jest konieczne w celu przeglądania swoich wizyt i zgłoszeń, zarówno przez pacjentów, jak i lekarzy. Administrator również musi się zalogować, aby mieć pełne uprawnienia do systemu. Administrator może tworzyć konta lekarzy, lekarze sami nie są w stanie tego wykonać. Odpowiednie konta i ich uprawnienia są ważną częścią systemu.

Wymaganie zostanie spełnione, kiedy będą istniały podstrony aplikacji (widoki) służące wprowadzeniu adresu email i hasła oraz przycisk służący logowaniu i przycisk do zakładania konta oraz kiedy konta będą rzeczywiście tworzone w bazie danych.

#### 3.2 Zgłaszanie potrzeby wizyty przez pacjentów

Jednym z głównych celów systemu jest wysyłanie przez ludzi zgłoszeń potrzeby wizyty u lekarza kiedy źle się poczują. Osoby te mogą stać się pacjentami. Aby umożliwić im zgłaszanie się, potrzebne jest wprowadzenie funkcjonalności zgłaszania prośby o wizytę wraz z danymi osobowymi i krótkim opisem objawów.

Kryterium spełnienia będzie zrealizowane, kiedy w części dla pacjentów będzie istniał przycisk do zgłaszania się i widok, na którym będzie można wpisać swoje objawy, a odpowiednie zapytanie zostanie przekazane do API systemu.

#### 3.3 Przeglądanie zaplanowanych wizyt przez pacjentów

Pacjenci powinni mieć możliwość sprawdzenia wizyt i ich dat, do których lekarz ich przypisał. Kiedy lekarz przyjmie zgłoszenie i zamieni je w wizytę, powinna ona pojawić się u pacjenta. Jest to sposób informowania pacjentów o przyjęciu ich zgłoszenia i jest to kolejna fundamentalna część aplikacji.

Kryterium będzie spełnione, kiedy po stworzeniu wizyty przez lekarza wyświetli się ona w pacjenta, który wysłał zgłoszenie.

#### 3.4 Przeglądanie zaplanowanych wizyt przez lekarzy

Lekarze powinni mieć możliwość sprawdzenia wizyt i ich dat, które utworzyli. Jest to sposób wspomagania lekarzy w organizacji pracy i czasu.

Kryterium będzie spełnione, kiedy po stworzeniu wizyty przez lekarza wyświetli się ona na jego widoku wśród innych wizyt.

---

### **3.5 Przeglądanie zgłoszeń pacjentów przez lekarzy**

Lekarze powinni mieć możliwość wyświetlenia zgłoszeń od wszystkich pacjentów, aby mogli wybrać odpowiednie zgłoszenie do swoich kompetencji i się nim zająć. Powinien być do tego specjalnie wydzielony widok z listą wszystkich zgłoszeń. Jest to kolejna fundamentalna część systemu, wynikająca bezpośrednio ze zgłaszania próśb przez pacjentów.

Kryterium będzie spełnione, kiedy po stworzeniu zgłoszenia przez pacjenta wyświetli się ono w systemie lekarzy wśród innych zgłoszeń.

### **3.6 Przyjmowanie zgłoszeń przez lekarzy, zamieniając je na ustalone wizyty**

Lekarze powinni mieć możliwość przyjęcia wybranego zgłoszenia, którym chcą się zająć. Oznacza to możliwość naciśnięcia w odpowiednim miejscu na zgłoszenie, a następnie wybór przycisku służącego do utworzenia wizyty i zaplanowanie jej podając datę i ewentualnie dodatkowe uwagi. Jest to ostatnia, konieczna część systemu do jego sensownego działania.

Kryterium będzie spełnione, kiedy po stworzeniu wizyty przez lekarza wyświetli się ono w systemie lekarza wśród innych wizyt i pojawi się u odpowiedniego pacjenta w widoku wizyt.

### **3.7 Możliwość zarządzania całym systemem przez administratorów**

W razie błędów ludzkich lub awarii sprzętowych, powinien istnieć użytkownik z uprawnieniami pozwalającymi rozwiązać te problemy i edytować wszystko w systemie. Służy do tego konto administratora, z poziomu którego musi być możliwa edycja wszystkich zgłoszeń, wizyt i kont w systemie. Dodatkowo, kiedy w przychodni pojawi się nowy lekarz, administrator ma możliwość zamiany konta utworzonego przez tą osobę na konto z uprawnieniami lekarskimi.

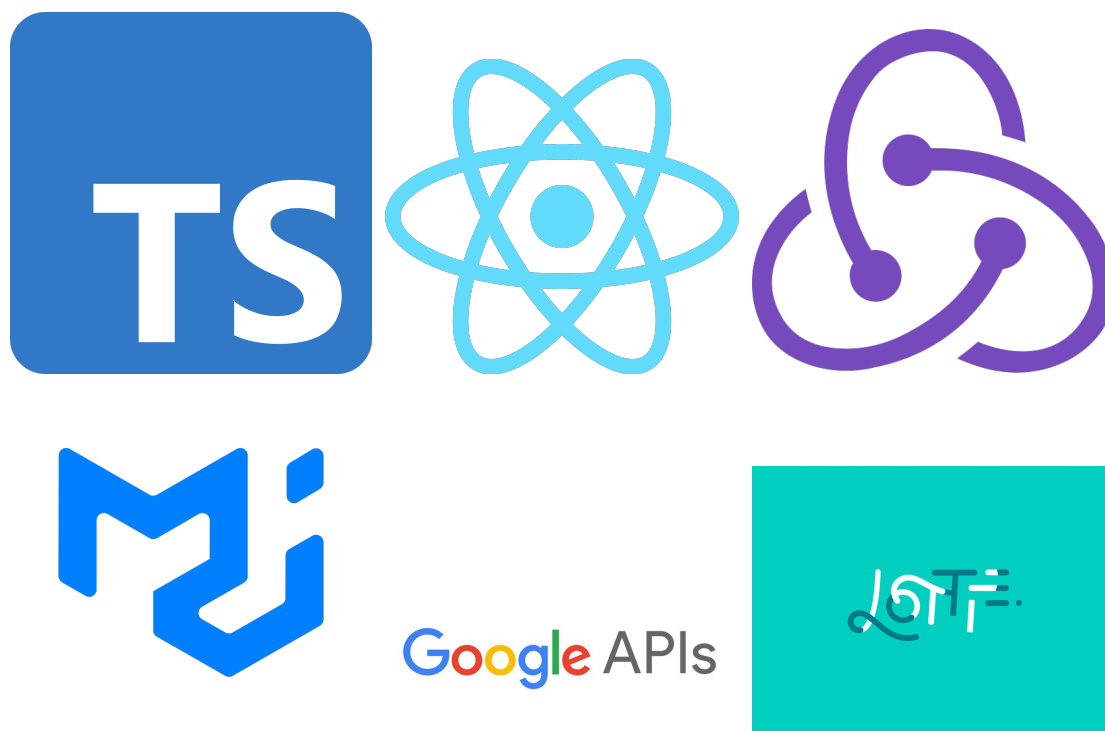
Kryterium będzie spełnione, kiedy administrator będzie mógł edytować wszystkie obiekty bazy danych i będą te zmiany trwale zapisywane.

---

## 4 Wykorzystane technologie i biblioteki

### 4.1 Aplikacja webowa

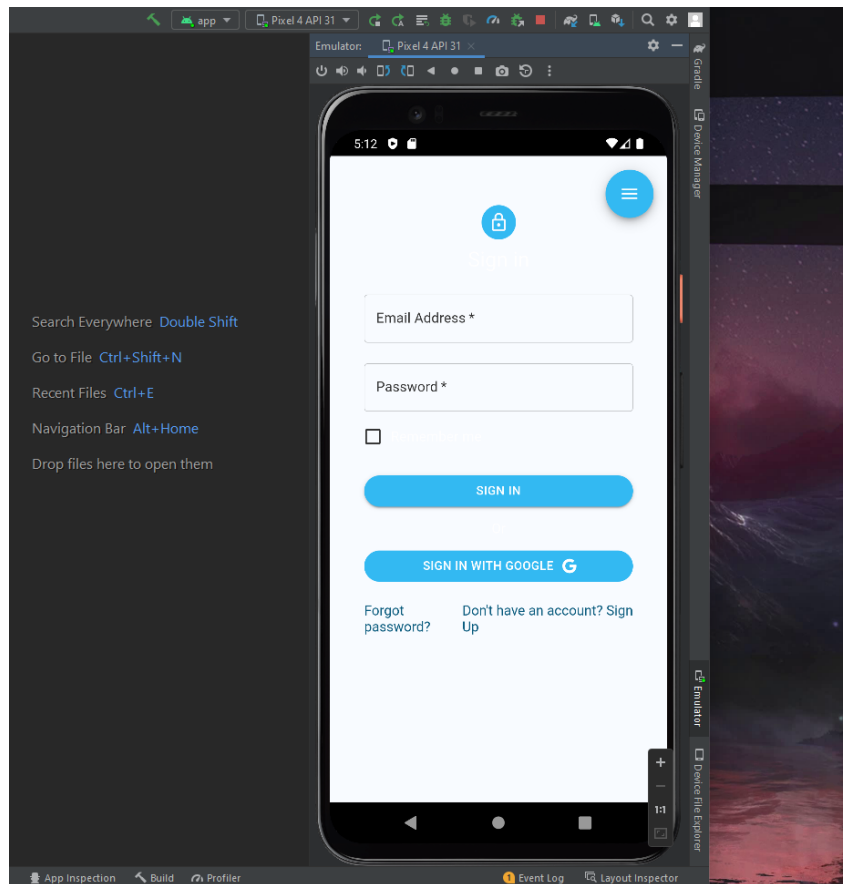
- [Typescript](#) - język programowania zbudowany na bazie [JavaScript'u](#) roszszerzający go o typy. Typescript obsługuje frontend aplikacji, interpretowany przez przeglądarkę kod, który pozwala na interakcję użytkownika z interfejsem. W jego środowisku pracują m.in. biblioteki React, Redux, MUI, gapi-script i lottie-react.
- [React](#) - biblioteka przy użyciu której jesteśmy w stanie tworzyć komponenty wielokrotnego użytku
- [Redux](#) - kontener do przechowywania stanów aplikacji (np. lista wizyt). Pozwala wyodrębnić warstwę danych od warsty interfejsu użytkownika
- [Mui](#) - biblioteka predefiniowanych komponentów, pozwala na szybsze, projektowanie i budowanie aplikacji
- [gapi-script](#) - pozwala na integrację aplikacji z interfejsem google
- [lottie-react](#) - zaawansowane animacje np. przy ładowaniu zasobów



Rysunek 1. Technologie aplikacji webowej

### 4.2 Aplikacja mobilna

Podjęliśmy próbę wygenerowania aplikacji mobilnej przy użyciu narzędzia Apache Cordova. Narzędzie pozwala na wygenerowanie aplikacji na systemy android i ios przy użyciu HTML, CSS i JS, jednak nie łączy się automatycznie z REST API, co uniemożliwia poprawne działanie aplikacji.



Rysunek 2. Aplikacja na system android

### 4.3 REST API

- [Python](#) - język programowania obsługujący backend aplikacji, czyli odpowiada na zapytania z aplikacji webowej i mobilnej oraz połączenie z bazą danych. W jego środowisku pracują biblioteki Django, django REST framework, google-api-python-client i zabezpieczenia JWT
- [Django](#) - biblioteka języka Python pozwalająca na stworzenie API przyjmującego żądania i odpowiadającego na nie oraz automatyzująca proces połączenia i obsługi bazy danych
- [django REST framework](#) - biblioteka języka Python rozszerzająca bibliotekę Django o możliwości tworzenia API w standardzie REST
- [Swagger](#) - pozwala na tworzenie dokumentacji REST API i wizualizację zasobów
- [google-api-python-client](#) - logowanie z wykorzystaniem konta Google
- [JWT](#) - sposób uwierzytelniania w postaci wymiany tokenów

### 4.4 Baza danych

- [PostgreSQL](#) - system zarządzania relacyjnymi bazami danych

### 4.5 Technologie testowania

- [Postman](#) - interakcja z API nie mając frontendu, pozwala na manualne testowanie zapytań (request) i odpowiedzi (response) oraz ich wizualizację i zapisywanie
- [PyUnit](#) - testy jednostkowe kodu *Python*, zastosowane w celu przeprowadzenia testów automatycznych API





**Rysunek 3.** Technologie backend (REST API i baza danych)



**Rysunek 4.** Technologie testowania

## 5 Harmonogram implementacji

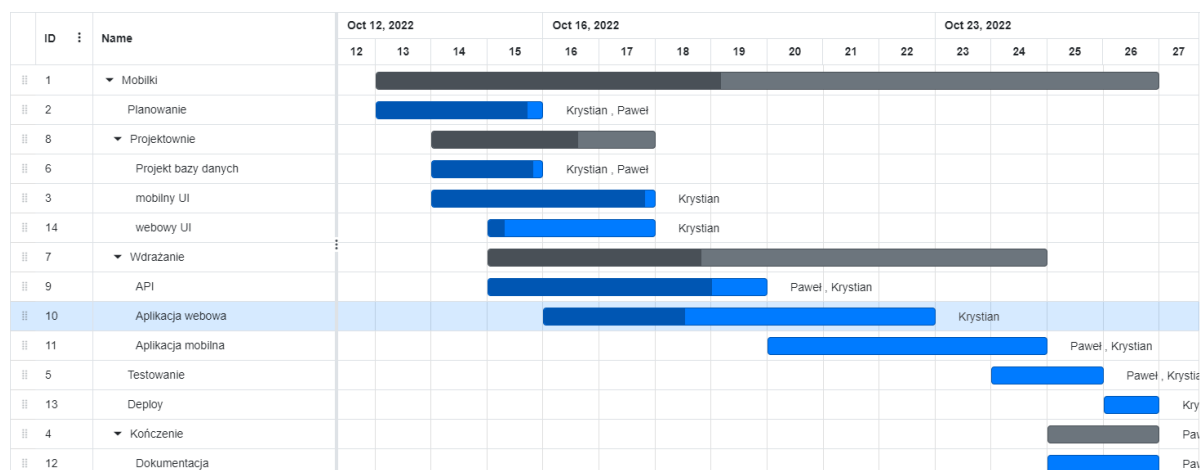
Skuteczna realizacja projektu wymaga odpowiednio przygotowanego planu działania i harmonogramu wdrażania kolejnych etapów. Jednym ze sposobów systematyzowania pracy jest Diagram Gantta, czyli graficzne przedstawienie harmonogramu prac wraz z datami i osobami odpowiedzialnymi. Zaawansowane diagramy pozwalają również na pokazywanie postępu realizowanych prac.

W przypadku naszego diagramu oś X prezentuje czas podzielony na dni, a oś Y zadania do wykonania. Dodatkowo, przy każdym zadaniu są przypisane osoby odpowiedzialne, a samo zadanie jest w postaci dwukolorowego paska oznaczającego postępy prac.

Diagram ten okazał się bardzo przydatny i pomógł nam trzymać się terminów i ważnych zadań, które w nich należało wykonać. Występowały opóźnienia w realizacji poszczególnych podpunktów w czasie, który założyliśmy, po których diagram był aktualizowany. Mimo drobnych opóźnień, diagram pozwolił wyrobić się ze znacznym zapasem czasowym względem ostatecznego terminu oddania projektu.

Nasz diagram podzielony jest na 6 głównych sekcji i są to:

1. Planowanie
2. Projektowanie
3. Tworzenie
4. Testowanie
5. Dokumentacja

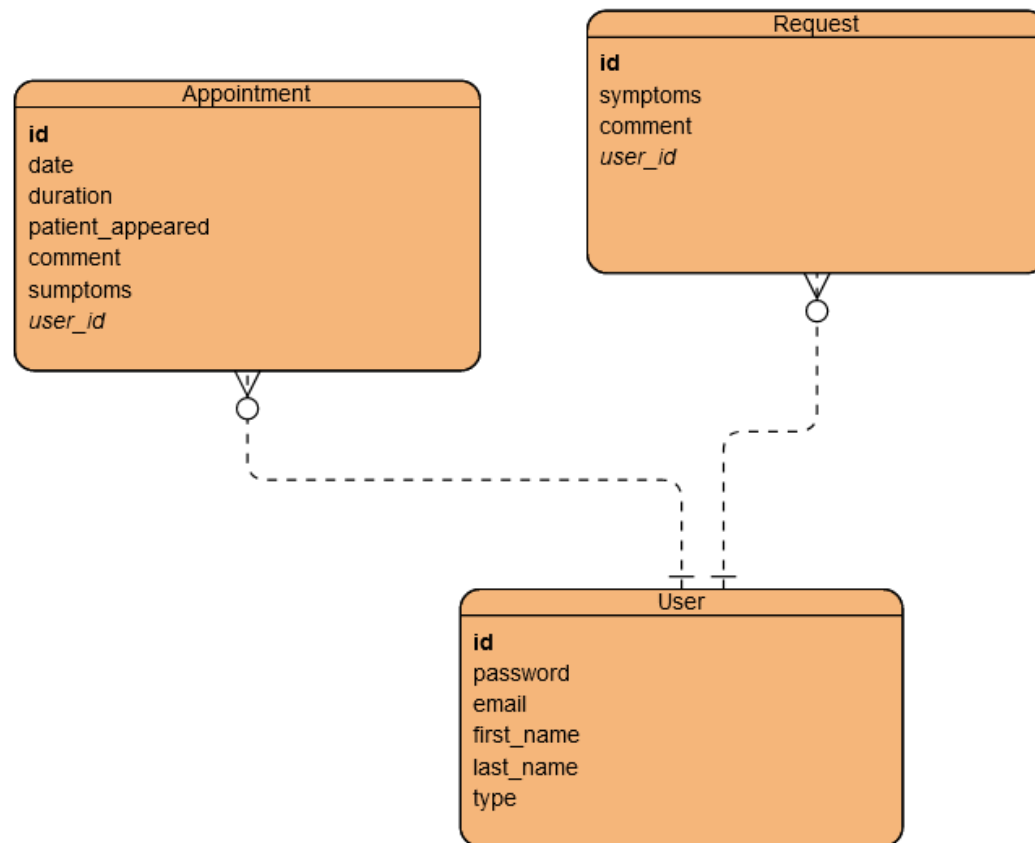


Rysunek 5. Diagram Gantta

---

## 6 Architektura informacji

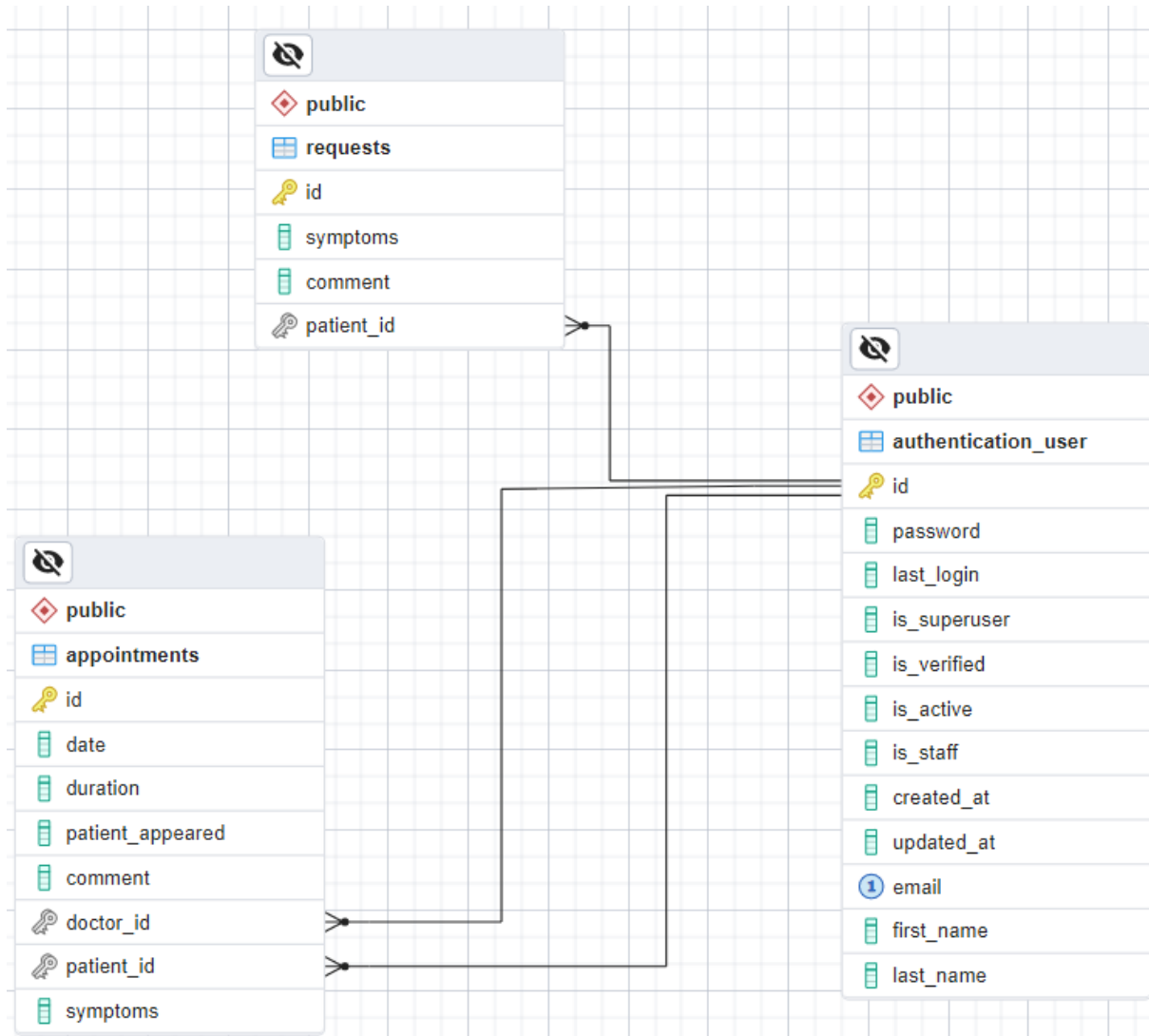
Pracę nad projektem zaczęliśmy od zaprojektowania bazy danych. Baza danych planowo powinna zawierać trzy encje, odpowiedzialne za użytkowników, żądania pacjentów i umówione wizyty, co przedstawiono na modelu logicznym.



Rysunek 6. Model logiczny bazy danych

## 6.1 Utworzenie bazy danych

Następnie baza danych została utworzona na podstawie modelu logicznego z wykorzystaniem biblioteki *Django* języka *Python*, która pozwala zautomatyzować proces tworzenia bazy danych. Do tabel zostały dodane dodatkowe atrybuty produkowane domyślnie przez *Django*.



Rysunek 7. Model fizyczny bazy danych

## 6.2 Opis encji i atrybutów w bazie danych

**Tabela 1.** Użytkownicy systemu (pacjenci, lekarze, administratorzy)

Atrybut	Klucz	Typ	Opis
id	PK	bigint	identyfikator użytkownika
password		character varying (128)	hasło użytkownika
last_login		timestamp with time zone	data ostatniego logowania
is_superuser		boolean	czy jest administratorem
is_verified		boolean	czy jest zweryfikowany
is_active		boolean	czy konto jest aktywne
is_staff		boolean	czy jest lekarzem
created_at		timestamp with time zone	data utworzenia konta
updated_at		timestamp with time zone	data ostatniej edycji konta
email		character varying (320)	email użytkownika
first_name		character varying (63)	imię użytkownika
last_name		character varying (63)	nazwisko użytkownika

**Tabela 2.** Zgłoszenia wysyłane przez pacjentów

Atrybut	Klucz	Typ	Opis
id	PK	bigint	identyfikator zgłoszenia
symptoms		character varying (255)	lista objawów podanych przez zgłaszającego
comment		text	komentarz pacjenta
patient_id	FK	bigint	id pacjenta wysyłającego zgłoszenie

**Tabela 3.** Wizyty ustalone przez lekarzy

Atrybut	Klucz	Typ	Opis
id	PK	bigint	identyfikator wizyty
date		timestamp with time zone	ustalony termin wizyty w postaci daty
duration		interval	czas trwania wizyty
patient_appeared		boolean	czy pacjent przyszedł
comment		text	komentarz lekarza
doctor_id	FK	bigint	id lekarza obsługującego wizytę
patient_id	FK	bigint	id pacjenta, który ma przyjść na wizytę
symptoms		character varying (255)	lista objawów podanych przez pacjenta

---

## 7 Opis interfejsu programistycznego

Interfejs programistyczny, czyli API, to aplikacja backendowa stworzona z użyciem *Python*, w szczególności *django-rest-framework*, dzięki któremu nasze aplikacje mogą porozumiewać się z bazą danych i ze sobą nazwzajem. Aplikacja ta może działać na serwerze i obsługiwać zapytania z wielu instancji aplikacji frontendowych jednocześnie. W API znajdują się też wszystkie zabezpieczenia i mechanizmy odpowiedzialne za logikę systemu.

### 7.1 Modele

W API zostały odzwierciedlone encje bazodanowe w postaci modeli. Modele znajdują się w module *models.py* i noszą nazwy *Appointment*, *Request* i *User*, z czego dwa pierwsze w folderze *core*, a *User* w *authentication*.

### 7.2 Endpointy

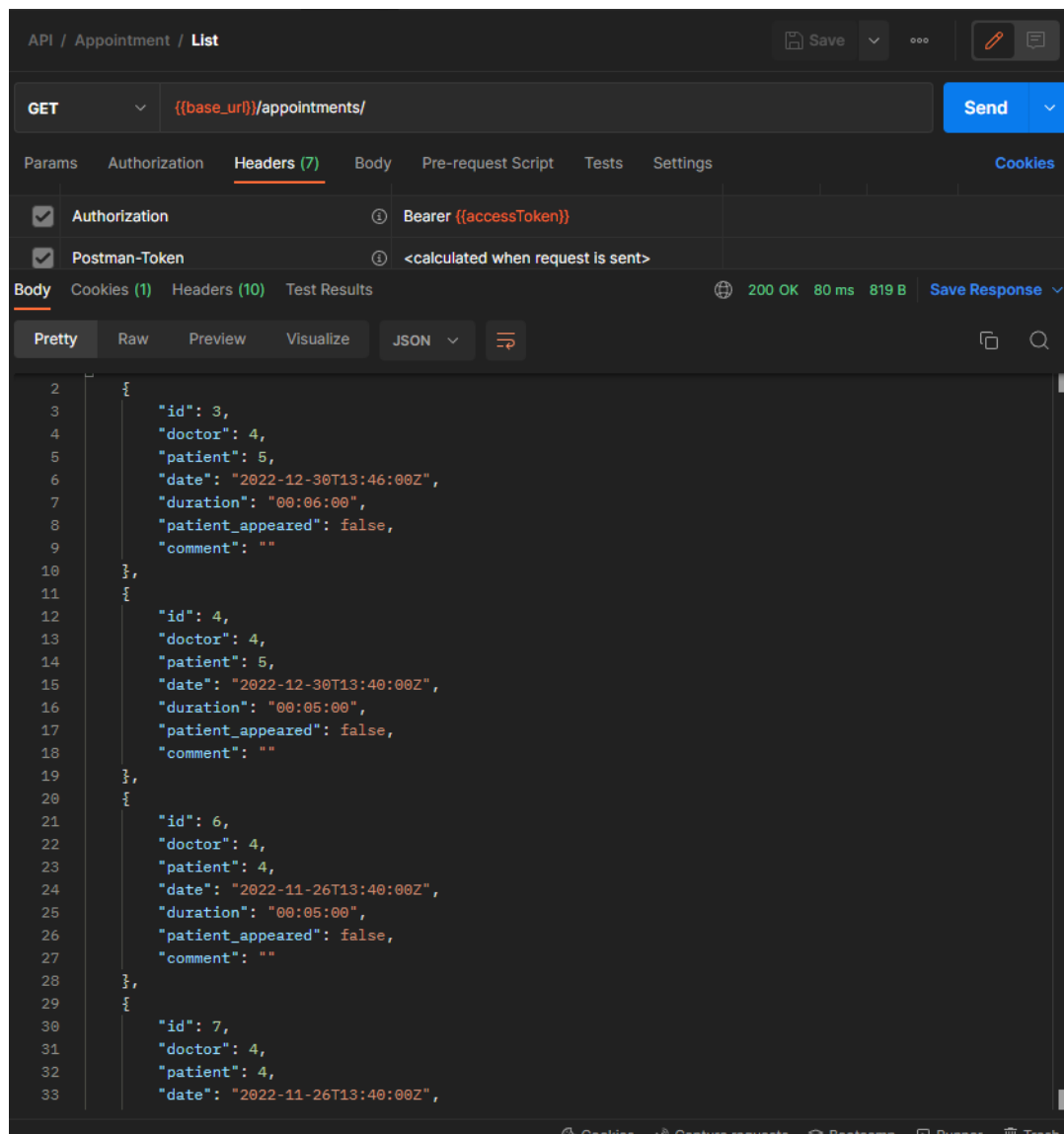
Każdy model zawiera swoje endpointy, czyli linki dostępu, przez które możemy go stworzyć, odczytać, edytować i usunąć, spełniając metodologię CRUD. Odczyt jest dodatkowo podzielony na pojedynczą instancję i wszystkie razem, co rozróżniamy przez metody *Retrieve* i *List*. Można więc powiedzieć o podejściu CRUDL, którego kolejne litery oznaczają strót od *Create*, *Retrieve*, *Update*, *Destroy*, *List*.

- `/auth/signup/` (POST) - rejestracja (zakładanie konta)
- `/auth/token/` (POST) - logowanie
- `/auth/signup/doctor/` (POST) - zakładanie konta lekarza
- `/users/:id/` (GET) - pobieranie (wyświetlanie) danych konkretnego użytkownika
- `/users/:id/` (UPDATE) - aktualizacja danych konkretnego użytkownika
- `/users/` (GET) - pobieranie danych wszystkich użytkowników
- `/appointments/` (POST) - tworzenie wizyt
- `/appointments/` (GET) - pobieranie wszystkich wizyt (w API istnieje konkretna logika powodująca, że dany lekarz może wyświetlić listę tylko swoich wizyt)
- `/appointments/:id/` (GET) - pobieranie konkretnej wizyty
- `/appointments/:id/` (UPDATE) - aktualizowanie konkretnej wizyty
- `/appointments/:id/` (DELETE) - usuwanie konkretnej wizyty
- `/requests/` (POST) - tworzenie próśb (zgłoszeń) o wizytę
- `/requests/` (GET) - pobieranie wszystkich zgłoszeń (z ograniczeniami)
- `/requests/:id/` (GET) - pobieranie konkretnego zgłoszenia
- `/requests/:id/` (UPDATE) - aktualizowanie konkretnego zgłoszenia
- `/requests/:id/` (DELETE) - usuwanie konkretnego zgłoszenia

Dodatkowo endpointy API zostały udokumentowane przy użyciu narzędzia [Swagger](#) gdzie można zobaczyć przykładową strukturę zapytań i odpowiedzi API, przykładowe obiekty i role.

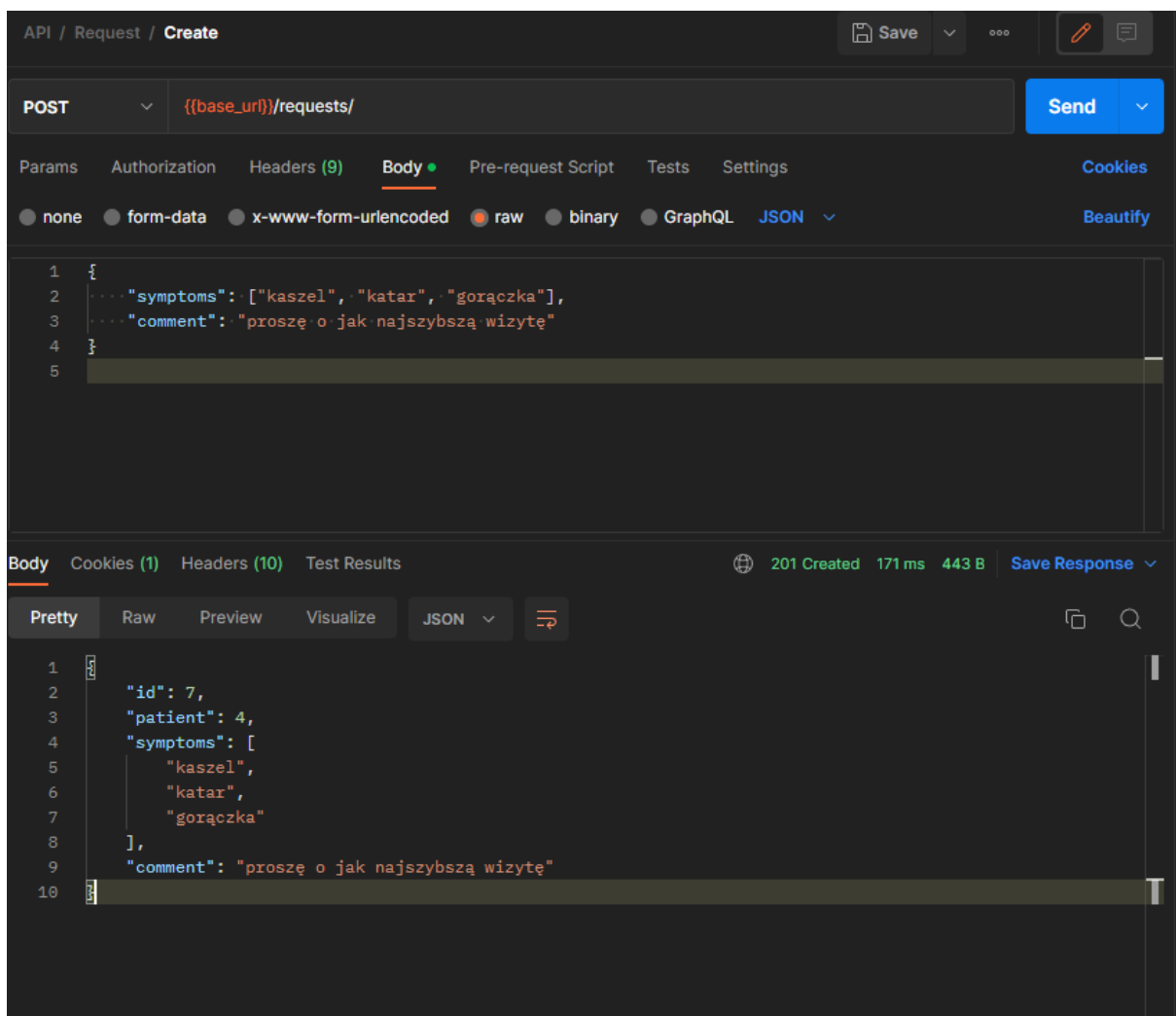
## 7.3 Testowanie API

**7.3.1 Postman** to narzędzie to wyzwalania endpointów i odczytywania ich rezultatów. Przykładowe zrzuty ekranu poniżej przedstawiają pobieranie informacji o wszystkich wizytach [25](#), które dotyczą lekarza wykonującego zapytanie oraz tworzenie zgłoszenia potrzeby wizyty przez pacjenta [9](#) z podaniem objawów. W wiadomości zwrotnej otrzymujemy status *201 Created* i dokładne dane stworzonego obiektu w bazie danych.



Rysunek 8. Odczytanie wszystkich wizyt danego lekarza (o id 4)

**7.3.2 Testy jednostkowe** pozwoliły sprawdzić działanie wszystkich endpointów, również w skrajnych przypadkach. Testy te są napisane raz i mogą być uruchamiane dowolną ilość razy, najlepiej po każdych zmianach w kodzie API. Testy czasem powstawały najpierw, przed daną funkcjonalnością, przez co nie były spełniane i staraliśmy się stworzyć daną funkcjonalność tak, aby spełniała test i przechodził on pozytywnie. Takie podejście nazywa się *Test-driven development*. Na zrzucie [10](#) widać wyniki uruchomienia testów.



Rysunek 9. Stworzenie zgłoszenia przez pacjenta

```
Found 18 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
-----
Ran 18 tests in 13.482s

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

Rysunek 10. Testy jednostkowe API



---

## 8 Graficzny interfejs użytkownika

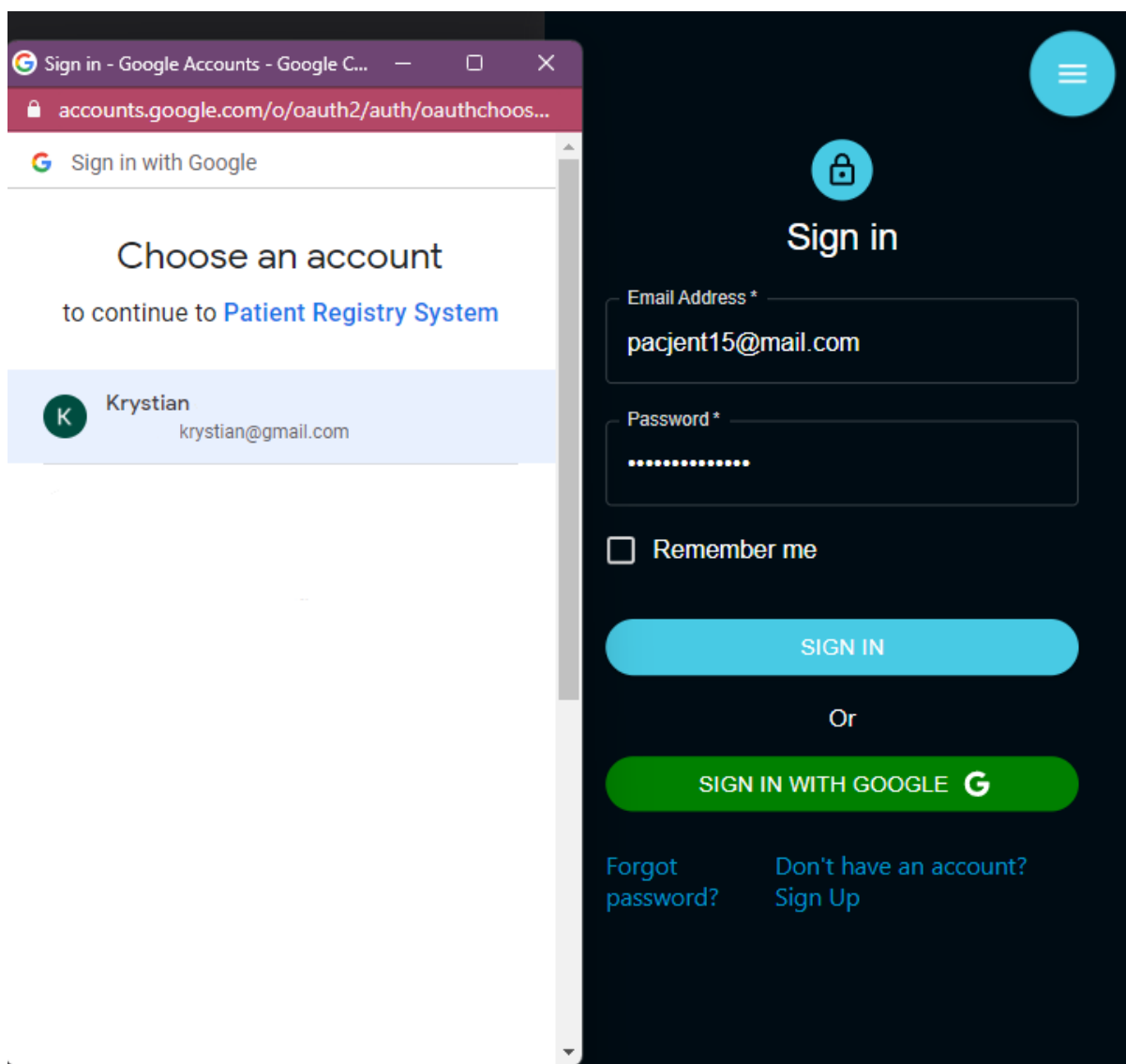
### 8.1 Widok mobilny aplikacji

The image displays two side-by-side mobile application screens with a dark blue background and light blue accents. Both screens feature a hamburger menu icon in the top right corner.

**Sign in screen (left):** It has a light blue lock icon at the top center. Below it is the text "Sign in". There are two input fields: "Email Address \*" and "Password \*". Below the password field is a checkbox labeled "Remember me". A large light blue button labeled "SIGN IN" is centered below the inputs. Underneath the button is the word "Or". Below "Or" is another large light blue button labeled "SIGN IN WITH GOOGLE" with the Google logo. At the bottom, there are two links: "Forgot password?" on the left and "Don't have an account? Sign Up" on the right.

**Sign Up screen (right):** It has a light blue lock icon at the top center. Below it is the text "Sign Up". There are three input fields: "Email Address \*", "Password \*", and "Repeat Password \*". A large light blue button labeled "CREATE ACCOUNT" is centered below the inputs. At the bottom, there is a link: "Already a member? Sign in".

Rysunek 11. Widok logowania i rejestracji



Rysunek 12. Widok logowania przy użyciu Google

Patient: John Doe

e-mail: johndoe@gmail.com

Symptoms:

zatoki    zatłany nos

Date and time  
11/24/2022 04:50

Duration  
00:30

Notes

CREATE APPOINTMENT

Rysunek 13. Widok tworzenia wizyty

Patient: John Doe

e-mail: johndoe@gmail.com

Symptoms:

DATE AND TIME

2022

Nov 10    04:50

CANCEL    OK

Rysunek 14. Widok wybierania czasu wizyty

**Appointments**

**A** Patient: tailowskikrystian@gmail.co  
Doctor: doctor13@mail.com

'Krystianowski Ogonowski'

Comment: Notatki

Date and time 08/18/2014 21:11 Duration 12:00

Doctor:doctor13@mail.com

**A** Patient: doctor13@mail.com  
Doctor: doctor13@mail.com

Comment: fggggggg

Date and time 08/18/2014 21:11 Duration 12:00

Doctor:doctor13@mail.com

**A** Patient: doctor12@mail.com  
Doctor: doctor13@mail.com

Comment: sdfsadfsaf

Date and time 08/18/2014 21:11 Duration 12:00

Doctor:doctor13@mail.com

Appointments

Rysunek 15. Widok umówionych wizyt

**Describe your symptoms**

zatkany nos zatoki cos tam

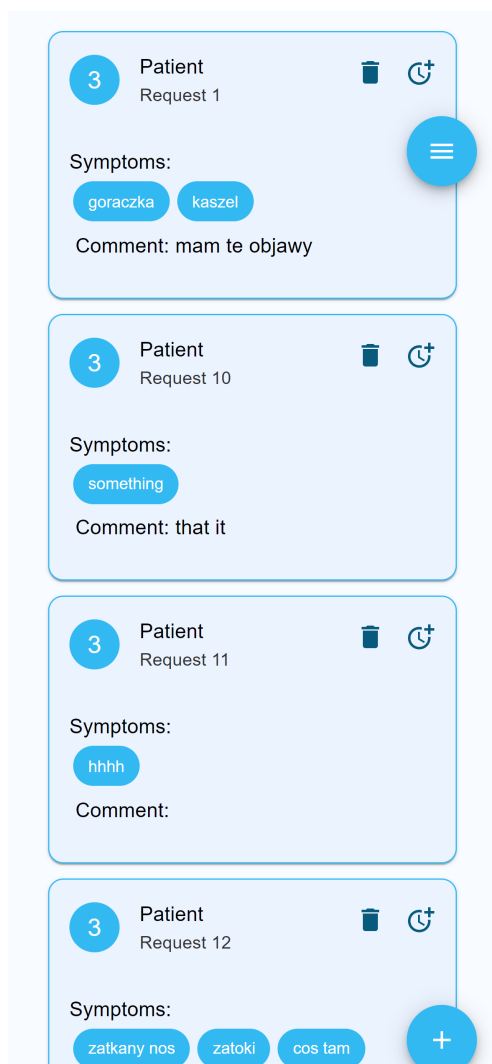
Symptom +

Description

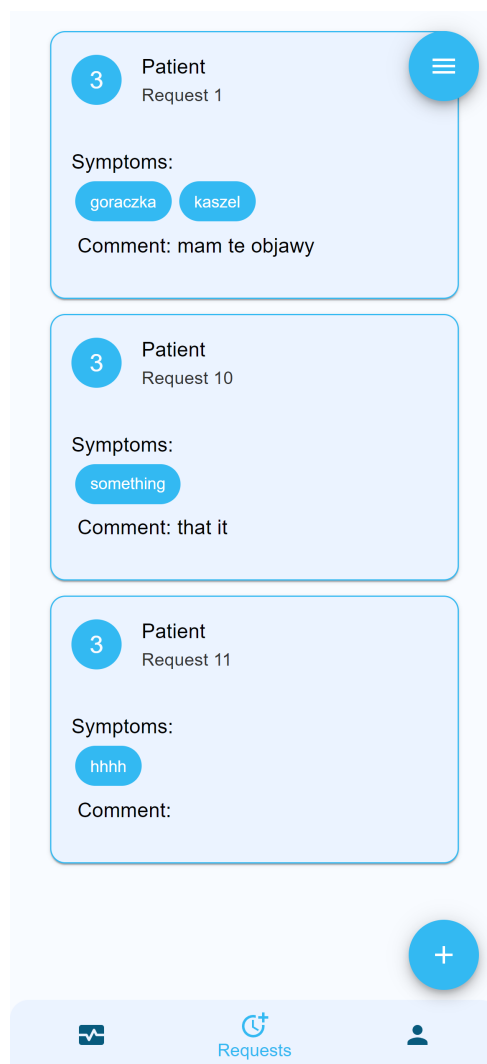
Mam takie objawy

REQUEST APPOINTMENT

Rysunek 16. Widok tworzenia zgłoszenia



**Rysunek 17.** Widok wizyt dla lekarza




**Rysunek 18.** Widok wizyt dla pacjenta


---

Personal information


Email

 tailowskikrystian@gmail.com

First name

 Krystianowski

Last name

 Ogonowski

Email notifications

☒ Doctor changes appointment



☐ Doctor deletes appointment


☒ Doctor accepts appointment

Extra features

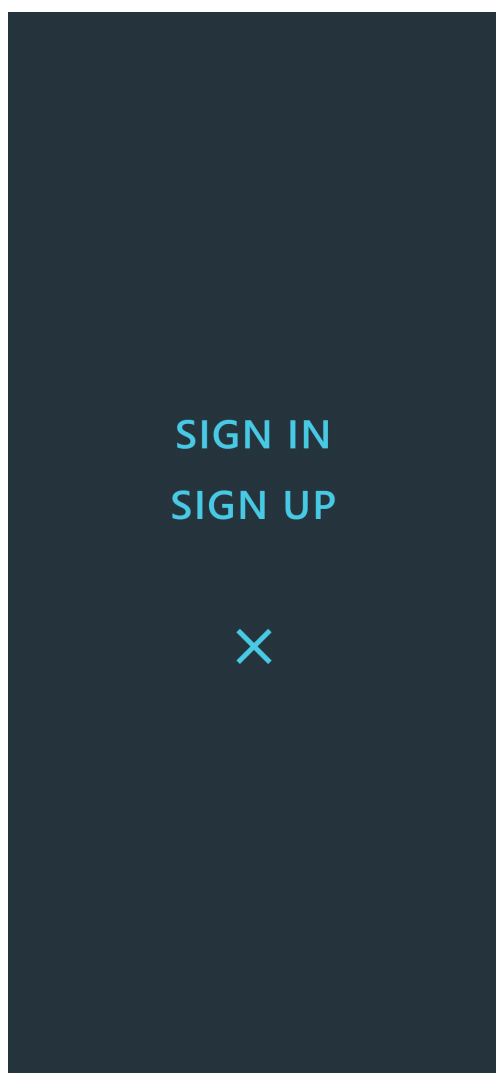
☐ Dark mode

☐ Vacation mode

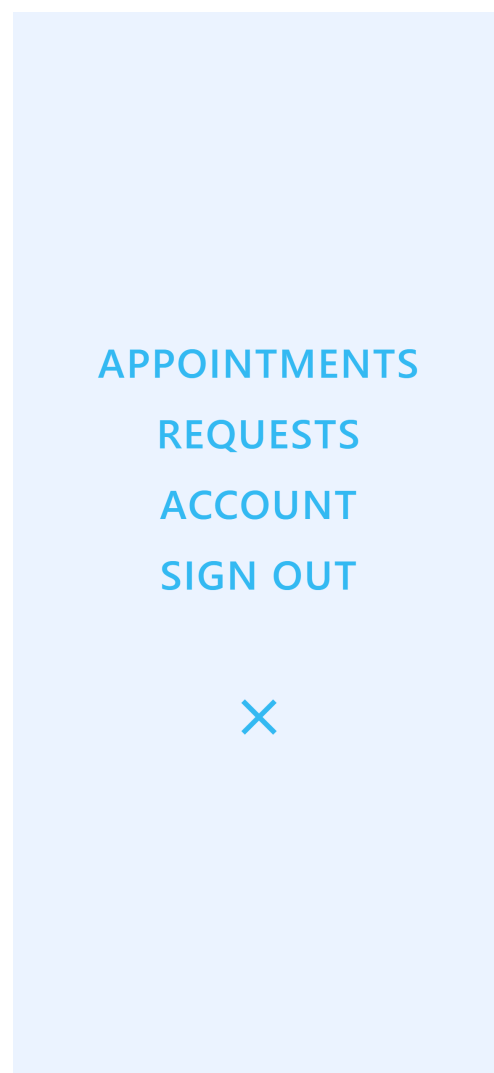


  
Account

Rysunek 19. Widok profilu użytkownika



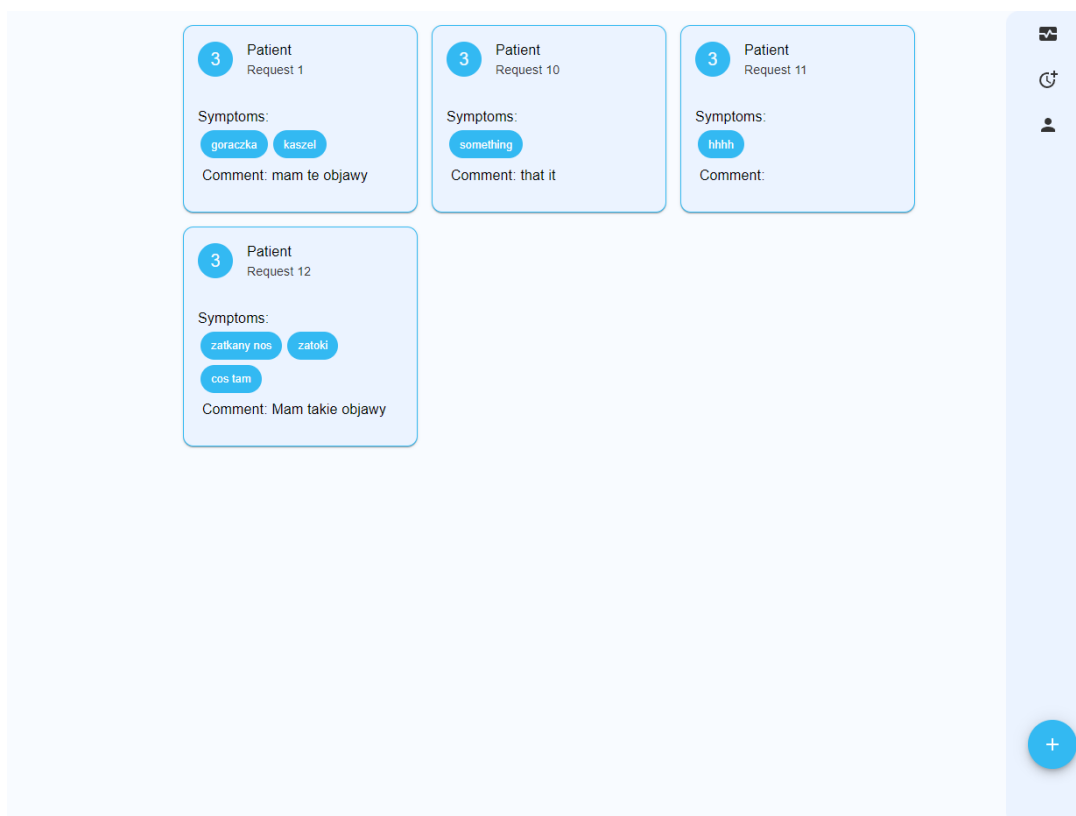
**Rysunek 20.** Widok nawigacji dla użytkownika niezalogowanego



**Rysunek 21.** Widok nawigacji dla użytkownika zalogowanego

## 8.2 Widok komputerowy aplikacji

Przykładowy widok dla aplikacji desktopowej. Aplikacja jest stworzona głównie pod widok mobilny. Widoki są niemal identyczne jak dla widoku mobilnego poza paskiem po prawej stronie okna i ilością kolumn.

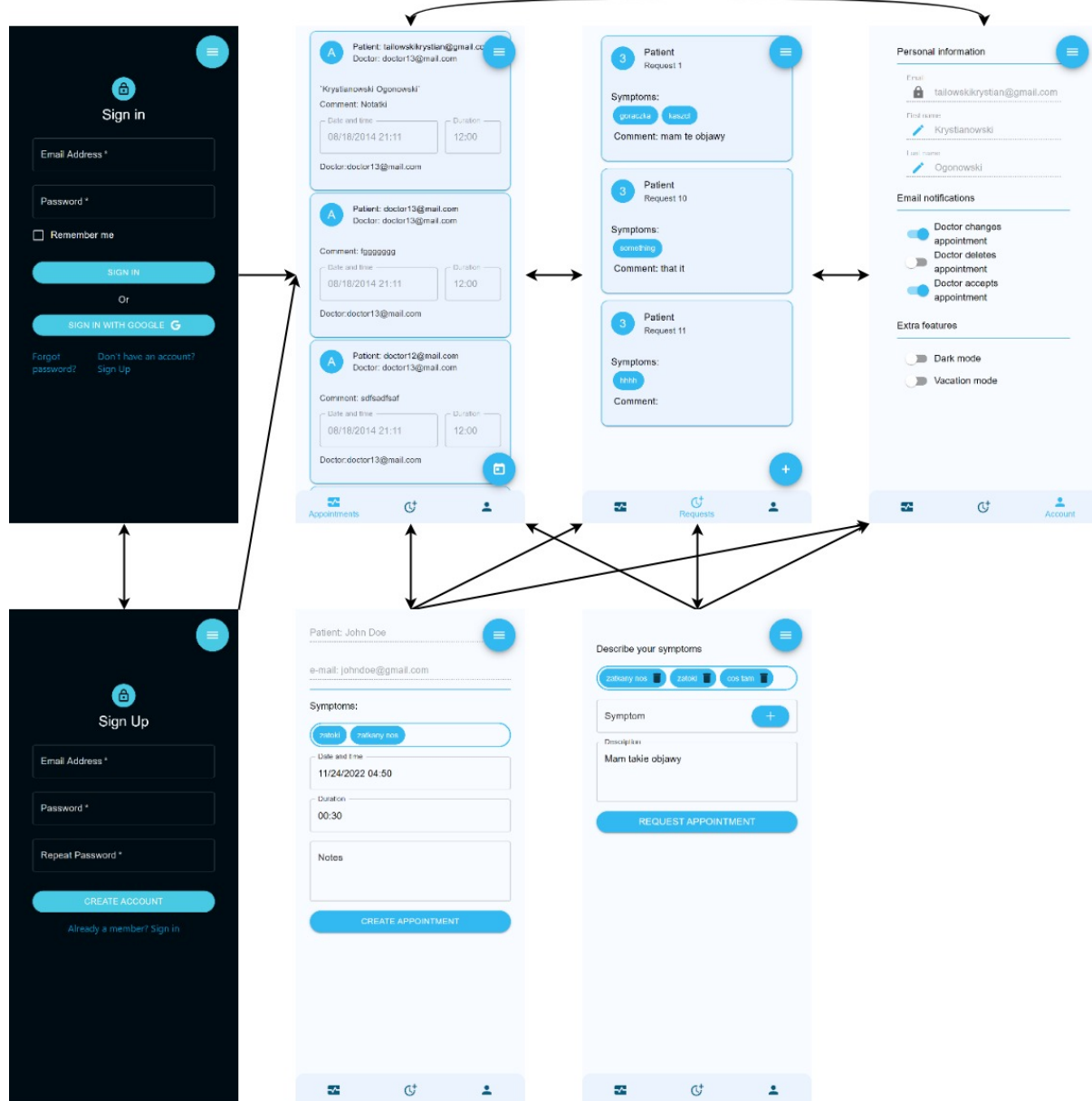


Rysunek 22. Widok profilu użytkownika

## 8.3 Relacje pomiędzy widokami

W każdym widoku mamy do dyspozycji przycisk otwierający szufladę z nawigacją (przykłady [Rys 20.](#), [Rys 21.](#)). Zalogowany użytkownik w wersji mobilnej aplikacji desktopowej ma dodatkowo do dyspozycji panel nawigacji ("*na dole*" *wyświetlanego okna*) pomiędzy ustalonymi wizytami, prośbami o wizytę i swoim profilem. W wersji desktopowej ten panel pojawia się po prawej stronie okna ([Rys 22.](#)). Pacjent z widoku "*Requests*" może przejść do widoku tworzenia prośby o wizytę. Lekarz dodatkowo może z tego poziomu przejść do widoku ustalania wizyty poprzez wybranie przycisku na karcie.



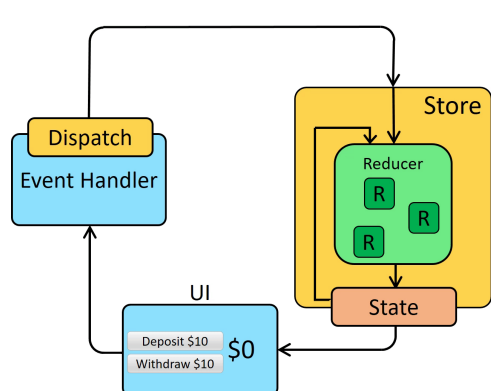


Rysunek 23. Relacje pomiędzy widokami

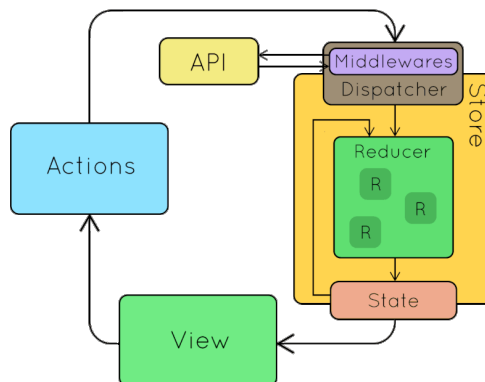
## 9 Opis implementacji (Aplikacja internetowa)

### 9.1 Redux store

Dzięki odrębnemu magazynowi na dane który zapewnia Redux, możemy w każdym miejscu aplikacji wywoływać endpointy i pobierać dane. Dodatkowo dodaje on system automatycznego odświeżania danych poprzez Tagi (linie 17,28,38 [Listingu 1](#). np. przy dodawaniu nowego elementu, podawany jest dodatkowy tag który np. dodatkowo wywołuje endpoint ponownie pobierający listę elementów (aby widoczny był nowy element).



Rysunek 24. Przepływ danych z użyciem kontenera Redux



Rysunek 25. Przepływ danych z dodatkowym middleware

```
1 import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';
2
3 const baseQuery = fetchBaseQuery({
4   baseUrl: config.API_SERVER,
5   prepareHeaders: (headers, { getState }) => {
6     const token = (getState() as RootState).auth.token
7     if (token) {
8       headers.set('Authorization', `Bearer ${token.access}`)
9     }
10    return headers
11  }
12 })
13
14 export const api = createApi({
15   reducerPath: 'api',
16   baseQuery: baseQuery,
17   tagTypes: ['Appointment', 'Request'],
18   ...
19 })
20
21 export const appointmentsApi = api.injectEndpoints({
22   endpoints: (build) => ({
23     getAppointments: build.query<Appointments, void>({
24       query: () => ({
25         url: '/appointments/',
26         method: 'GET'
27       }),
28       providesTags: ['Appointment']
29     }),
30     addAppointment: build.mutation<Appointment, Partial<CreateAppointment>
31     >>({
32       query: (body: CreateAppointment) => {
```

```

32         return {
33             url: '/appointments/',
34             method: 'POST',
35             body
36         }
37     },
38     invalidatesTags: ['Appointment', 'Request']
39 },
40 ...
41 })
42
43 export const {
44     useGetAppointmentsQuery,
45     useAddAppointmentMutation,
46     ...
47 } = appointmentsApi

```

Listing 1. Główna część pliku z routingiem

## 9.2 Routing

W zależności od aktualnej ścieżki wyświetlamy różne strony, przy użyciu biblioteki `react-router-dom` to zadanie jest znacznie ułatwione dzięki zagnieżdżaniu komponentów jak np. w liniach 12-34 Listingu 2. W dalszej części dzięki hook'i z tej biblioteki umożliwiają nawigację pomiędzy stronami w kodzie (`useNavigate`), zapewniają parametry query (`useParams`) i przekazują informacje pomiędzy stronami (`useLocation`).

```

1  <Suspense fallback={<Loading />}>
2    <Routes>
3      <Route path="/" element={<WithNav />}>
4        {/* public routes */}
5        <Route index element={<Home />} />
6        <Route path="signin" element={<SignIn />} />
7        <Route path="signup" element={<SignUp />} />
8        {/* <Route path="restore" element={<RestorePassword />} /> */}
9        <Route path="Unauthorized" element={<Unauthorized />} />
10
11      {/* private routes */}
12      <Route
13        element={
14          <RequireAuth
15            allowedUserType={[UserType.PATIENT, UserType.DOCTOR]}
16          />
17        }
18      >
19        <Route path="account" element={<Account />} />
20
21        <Route path="appointments">
22          <Route index element={<Appointments />} />
23          <Route
24            element={<RequireAuth allowedUserType={[UserType.DOCTOR
25              ]} />}
26          >
27            <Route path="create" element={<AddAppointment />} />
28          </Route>
29          <Route path="requests">
30            <Route index element={<Requests />} />
31            <Route path=":id" element={<Request />} />
32            <Route path="create" element={<AddRequest />} />

```

```

33         </Route>
34     </Route>
35
36     { /* catch all */ }
37     <Route path="*" element={<NotFound />} />
38 </Route>
39 </Routes>
40 </Suspense>

```

**Listing 2.** Główna część pliku z routingiem

### 9.3 Logowanie i zakładanie konta

Przy logowaniu użytkownik dostaje informację jeśli email lub hasło nie są poprawne:

- email nie jest zarejestrowany
- hasło nie pasuje do podanego emaila

Przy czym jest to ten sam komunikat, w inny przypadku możnaby określić czy email istnieje w bazie danych. Przy rejestracji sprawdzana jest poprawność wprowadzonych danych poprzez regexy:

- email: `/^((([<>() []\.,;:\s@"]+(\. [^<>() []\.,;:\s@"]+)*)(".+"))@((([0-9]1,3[0-9]1,3[0-9]1,3[0-9]1,3))—((([a-zA-Z0-9]+)+[a-zA-Z]2,))$)/—`
- hasło: `/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$`

```

1 const slice = createSlice({
2   name: 'theme',
3   initialState: isDarkTheme,
4   reducers: {
5     toggle: (state: boolean) => {
6       setDarkTheme(prev => !prev)
7       state = !state
8     }
9   }
10 })
11 ...

```

**Listing 3.** Informacje o preferencji kolorystycznej aplikacji

### 9.4 Wymiana danych z REST API

Do tego zadania ponownie wykorzystano magazyn jakim jest *Redux*, pozwala on przechowywać i buforować dane takie jak: tokeny, informacje o użytkowniku czy dane o wizytach. [Listing 4.](#) pokazuje przykładowo można zdefiniować zapytania GET i POST

```

1 export const requestsApi = api.injectEndpoints({
2   endpoints: (build) => ({
3     getRequests: build.query<Requests, void>({
4       query: () => ({
5         url: '/requests/',
6         method: 'GET'
7       }),
8     providesTags: ['Request']
9   }),

```

```

10      addRequest: build.mutation<Request, Partial<Request>>({
11          query: (body: Request) => {
12              return {
13                  url: '/requests/',
14                  method: 'POST',
15                  body
16              }
17          },
18          invalidatesTags: ['Request']
19      }),
20      ...
21  })
22  })

```

**Listing 4.** Przykładowe endpointy z wykorzystaniem Redux

## 9.5 Przechowywanie danych

Do przechowywania danych takich jak *refresh token* wykorzystano pamięć lokalną (*localStorage*), a cały proces usprawnia hook *usehooks-ts*. Przykład zastosowanie pokazuje Listing 5.

```

1  const [isDarkTheme, setDarkTheme] = useLocalStorage<boolean>('darkTheme', false
2  )
3  ...
4  toggle: (state: boolean) => {
5      setDarkTheme(prev => !prev)
6      ...
7  }

```

**Listing 5.** Przykładowe endpointy z wykorzystaniem Redux

## 9.6 Zewnętrzne interfejsy programistyczne

Do aplikacji dodany został interfejs *google api (gapi)*, który dodaje możliwość utworzenia konta lub zalogowania do aktualnie istniejącego przy użyciu istniejącego konta Google. Aplikacja jest w fazie testowej dlatego przed zalogowaniem należy dodać konto do zaufanych w aplikacji *Google Cloud*. Po udanym zalogowaniu backend pobiera podstawowe informacje o użytkowniku Google takie jak: imię, nazwisko czy email.

## 9.7 Prezentowanie danych

Do prezentowania danych, wykorzystano spersonalizowane *karty (Card)* z biblioteki *MUI*. Do prezentacji informacji o użytkowniku (profil) użyto *stosu (Stack)* i pól tekstowych czy tym podobnych komponentów.

---

## 10 Krytyczna ocena projektu

System spełnia swoje założenia i działa w sposób bardzo kontrolowany. Jedyne utrudnienia wynikały z braków czasowych, przez co niektóre funkcjonalności wdrażaliśmy z opóźnieniem względem nałożonych surowych ram czasowych. Więcej uwagi przyłożono do aplikacji webowej niż mobilnej. Niemniej jednak wszystkie przypadki użycia działają w każdym środowisku. Projekt uważamy za udany.

### 10.1 Zrealizowane założenia

Zrealizowano wszystkie wymagania funkcjonalne, co było celem projektu. Wymyślono dodatkowo funkcjonalności systemu, które można wprowadzić w przyszłości. Ponadto, wprowadzony został system logowania przez konto Google, co wykracza poza początkowe założenia projektu.

### 10.2 Dalsze możliwości rozwoju

System można rozwinąć o wiele funkcjonalności, a przykładowe z nich to:

- kod weryfikacyjny wysyłany na maila po założeniu konta
- wyszukiwanie wizyt po dacie
- powiadomienia mailowe dla pacjenta o utworzeniu i ewentualnej zmianie (edycji) wizyty przez lekarza

## Podsumowanie

Projekt ten okazał się bardzo przyjemny i rozwojowy. Praca nad nim była zebraniem dotychczasowych umiejętności w całość i zrobieniem kompletnego systemu, który ma podstawowe funkcjonalności, ale każda funkcjonalność jest dopracowana. System ten można rozwijać o dalsze funkcjonalności w przyszłości. Dodatkowo pozytywne było zaznajomienie się z aplikacjami mobilnymi, co było dla nas nowością.