

Uniwersytet Mikołaja Kopernika w Toruniu

Wydział Matematyki i Informatyki

Mateusz Mariusz Szymczak

nr albumu: 308216

Informatyka, studia inżynierskie

Praca inżynierska

Aplikacja webowa do zarządzania przychodnią lekarską

Promotor

dr Jakub Narębski

Toruń 2025



# Spis treści

Spis treści .....	3
Wstęp.....	1
1. Technologie i środowisko pracy.....	2
1.1. Visual Studio Code .....	2
1.2. System kontroli wersji Git oraz GitHub .....	2
1.3. HTML .....	2
1.4. SCSS (Sass) .....	3
1.5. TypeScript.....	3
1.6. Angular .....	3
1.7. Bootstrap.....	3
1.8. RxJS .....	3
1.9. Angular Material .....	4
1.10. Python .....	4
1.11. Django Rest Framework (DRF).....	4
1.12. Pytest.....	4
1.13. Swagger / OpenAPI .....	4
1.14. PostgreSQL.....	5
1.15. Docker Compose.....	5
1.16. GitHub Actions .....	5
2. Architektura aplikacji .....	6
2.1. Baza danych.....	6
2.1.1. Schemat bazy danych .....	6
2.1.2. Sekwencje w systemie identyfikatorów tabel.....	9
2.2. Backend.....	10
2.2.1. Struktura katalogów projektu .....	10
2.2.2. Zmienne środowiskowe .....	11
2.2.3. Dane inicjalne .....	13
2.2.4. Autoryzacja i JWT .....	14
2.2.5. System uprawnień (permissions).....	16
2.2.6. reCAPTCHA .....	16
2.2.7. Wysyłka wiadomości e-mail.....	17
2.2.8. Throttling.....	18
2.2.9. Architektura API i ViewSety.....	19
2.2.10. Dokumentacja API (Swagger / OpenAPI) .....	22
2.2.11. Testy w Pytest .....	23

2.3.	Frontend .....	24
2.3.1.	Struktura katalogów projektu .....	24
2.3.2.	Architektura aplikacji i podejście standalone .....	26
2.3.3.	Routing i ochrona tras.....	26
2.3.4.	Autoryzacja i zarządzanie stanem użytkownika .....	27
2.3.5.	Komunikacja z backendem.....	29
2.3.6.	Dyrektywy uprawnień i widoczności elementów .....	31
2.3.7.	Komponenty tabel i obsługa danych.....	33
2.3.8.	Formularze i walidacja .....	36
2.3.9.	Powiadomienia i komunikaty użytkownika (ToastService) .....	38
2.4.	Docker.....	39
2.5.	CI / CD (Continuous Integration / Continuous Deployment) .....	40
3.	Przegląd funkcjonalności aplikacji.....	42
3.1.	Ekran powitalny aplikacji (dla niezalogowanego użytkownika) .....	42
3.2.	Moduł uwierzytelniania i zarządzania kontem .....	43
3.2.1.	Rejestracja użytkownika.....	43
3.2.2.	Logowanie .....	46
3.2.3.	Resetowanie hasła.....	46
3.2.4.	Zmiana hasła i wylogowywanie użytkownika.....	48
3.3.	Uprawnienia użytkowników w aplikacji.....	49
3.3.1.	Pacjent .....	49
3.3.2.	Pielęgniarka .....	50
3.3.3.	Lekarz .....	50
3.3.4.	Administrator.....	51
3.4.	Dostęp do słowników.....	52
3.5.	Wizyty.....	54
3.6.	Recepty .....	57
	Podsumowanie i plany rozwoju.....	60
	Bibliografia.....	62
	Spis fragmentów kodu.....	63
	Spis rysunków .....	64
	Spis tabel .....	65

## Wstęp

Wraz z postępującą cyfryzacją sektora ochrony zdrowia rośnie zapotrzebowanie na systemy informatyczne wspierające zarządzanie placówkami medycznymi. Przychodnie lekarskie, zarówno publiczne, jak i prywatne, korzystają z tzw. systemów HIS (ang. *Healthcare Information Systems*), które umożliwiają elektroniczną obsługę pacjentów, prowadzenie dokumentacji medycznej oraz zarządzanie pracą personelu.

Na rynku dostępnych jest wiele rozwiązań tego typu – od darmowych, oferowanych przez państwowe instytucje (np. [gabinet.gov.pl](http://gabinet.gov.pl)), po rozbudowane komercyjne produkty (np. mMedica, Optimed NXT, Kamsoft KS-SOMED). Systemy te różnią się zakresem funkcjonalności, stopniem integracji z usługami NFZ oraz poziomem dostosowania do konkretnych potrzeb placówek.

Coraz większą popularność zyskują rozwiązania oparte o **architekturę webową**, które w przeciwieństwie do tradycyjnych aplikacji desktopowych nie wymagają instalacji na stanowiskach roboczych, a do działania potrzebują jedynie przeglądarki internetowej. Tego typu systemy oferują łatwiejszą aktualizację, centralne zarządzanie, lepszą skalowalność oraz dostępność z różnych lokalizacji – co jest szczególnie istotne w kontekście pracy zdalnej, wizyt domowych lub rozproszonej struktury placówek. Dodatkowo aplikacje webowe mogą być łatwo integrowane z zewnętrznymi usługami, takimi jak system e-Recepta, P1 czy eWUŚ, co ułatwia wdrażanie obowiązujących standardów teleinformatycznych w ochronie zdrowia.

W niniejszej pracy zaprojektowano i zaimplementowano autorski system wspierający wybrane procesy w przychodni – w szczególności obsługę użytkowników (pacjentów i personelu), rejestrację wizyt oraz dostęp do informacji o dostępnych gabinetach. System został zbudowany w architekturze klient-serwer z wykorzystaniem frameworków Angular (frontend) i Django REST Framework (backend), a jego rozwój uwzględniał dobre praktyki w zakresie testowania i dokumentacji backendu oraz automatyzacji wdrożeń (CI/CD).

# **1. Technologie i środowisko pracy**

W niniejszym rozdziale przedstawiono zestaw technologii i narzędzi wykorzystanych podczas realizacji projektu. Każde z opisanych rozwiązań odgrywa istotną rolę w architekturze systemu – od warstwy frontendowej, przez backend i bazę danych, aż po narzędzia wspomagające testowanie oraz uruchamianie środowisk deweloperskich. Wybór technologii został podyktowany ich stabilnością, popularnością w branży, a także dopasowaniem do potrzeb projektu i zespołu programistycznego.

## **1.1. Visual Studio Code**

Visual Studio Code (VS Code) to lekki, wieloplatformowy edytor kodu źródłowego stworzony przez Microsoft. Oferuje rozbudowane wsparcie dla języków programowania, inteligentne podpowiedzi składni (IntelliSense), wbudowany terminal, debugger oraz integrację z systemami kontroli wersji. Dzięki bogatemu ekosystemowi rozszerzeń i dużej społeczności, VS Code umożliwia efektywną pracę zarówno nad frontendem (Angular), jak i backendem (Python/Django) [1].

## **1.2. System kontroli wersji Git oraz GitHub**

Git to rozproszony system kontroli wersji, który umożliwia śledzenie zmian w kodzie źródłowym, pracę w zespołach oraz zarządzanie historią projektu [2]. GitHub to platforma hostingowa dla repozytoriów Git, która oferuje dodatkowe funkcje, takie jak pull requesty, system zgłoszeń (issues), wiki oraz integrację z narzędziami CI/CD [3]. W projekcie Git był wykorzystywany do zarządzania wersjami kodu, a GitHub pełnił funkcję zdalnego repozytorium.

## **1.3. HTML**

HTML (HyperText Markup Language) to podstawowy język znaczników wykorzystywany do tworzenia struktury dokumentów internetowych. Umożliwia definiowanie elementów strony, takich jak nagłówki, paragrafy, formularze, tabele czy przyciski. W projekcie HTML pełnił rolę szkieletu aplikacji frontendowej, definiując strukturę komponentów Angulara oraz interfejsów użytkownika, które następnie stylowano za pomocą CSS/SCSS i ożywiano z wykorzystaniem TypeScript [4].

## **1.4. SCSS (Sass)**

SCSS (Sassy CSS) to składnia preprocesora CSS o nazwie Sass, która rozszerza możliwości tradycyjnego CSS o zmienne, zagnieżdżenia, dziedziczenie, mixiny i funkcje. Dzięki temu możliwe jest tworzenie bardziej modularnych, skalowalnych i czytelnych arkuszy stylów. W projekcie SCSS był wykorzystywany do stylowania komponentów Angulara zgodnie z podejściem scoped CSS, co umożliwiało zachowanie izolacji stylów oraz ich wielokrotne wykorzystanie [5].

## **1.5. TypeScript**

TypeScript to nadzbiór języka JavaScript, który wprowadza statyczne typowanie oraz dodatkowe konstrukcje obiektowe. Dzięki typom możliwe jest wcześniejsze wykrywanie błędów, lepsze wsparcie przez IDE oraz większa czytelność kodu. W projekcie TypeScript był językiem wykorzystywanym do pisania całej logiki frontendowej w Angularze – zarówno komponentów, jak i usług, modeli danych oraz obsługi komunikacji z API [6].

## **1.6. Angular**

Angular to framework do tworzenia aplikacji frontendowych w języku TypeScript, rozwijany i wspierany przez Google. Umożliwia budowanie dynamicznych aplikacji jednostronicowych (SPA) z wykorzystaniem komponentów, usług i reaktywnego programowania. Angular zapewnia także wbudowane mechanizmy routingu, formularzy, komunikacji z API oraz modularności aplikacji [7].

## **1.7. Bootstrap**

Bootstrap to popularny framework CSS służący do tworzenia responsywnych i estetycznych interfejsów użytkownika. Ułatwia projektowanie układów stron, formularzy, przycisków i innych elementów UI z użyciem gotowych komponentów oraz systemu siatki (grid system). W projekcie został użyty do zapewnienia spójnego i responsywnego wyglądu aplikacji frontendowej [8].

## **1.8. RxJS**

RxJS (Reactive Extensions for JavaScript) to biblioteka służąca do reaktywnego programowania z użyciem obserwowalnych strumieni danych. Umożliwia asynchroniczne przetwarzanie zdarzeń oraz zarządzanie stanem aplikacji w sposób deklaratywny. W

połączeniu z Angularem RxJS odgrywa kluczową rolę w komunikacji z backendem, obsłudze formularzy oraz zarządzaniu przepływem danych [9].

## **1.9. Angular Material**

Angular Material to biblioteka komponentów UI zgodna z wytycznymi Material Design, rozwijana przez zespół Angulara. Umożliwia szybkie tworzenie estetycznych, nowoczesnych i dostępnych interfejsów użytkownika w aplikacjach Angularowych. W projekcie wykorzystano ją m.in. do tworzenia formularzy, dialogów oraz elementów nawigacyjnych [10].

## **1.10. Python**

Python to wszechstronny język programowania o czytelnej składni i bogatym ekosystemie bibliotek. Jest szeroko wykorzystywany w aplikacjach webowych (m.in. z Django), analizie danych, automatyzacji oraz uczeniu maszynowym. Jego prostota i ekspresyjność czynią go idealnym wyborem zarówno dla początkujących, jak i profesjonalnych programistów [11].

## **1.11. Django Rest Framework (DRF)**

Django Rest Framework to rozszerzenie dla frameworka Django, które umożliwia tworzenie nowoczesnych i elastycznych interfejsów API w stylu REST. DRF oferuje bogaty zestaw narzędzi do serializacji danych, autoryzacji, paginacji i filtrowania, co przyspiesza budowę backendu aplikacji webowych i mobilnych [12].

## **1.12. Pytest**

Pytest to popularne narzędzie do testowania w Pythonie, pozwalające pisać czytelne i rozbudowane testy z użyciem prostego API. Obsługuje testy jednostkowe, funkcjonalne oraz integracyjne, a dzięki wsparciu dla fixture'ów i pluginów, umożliwia łatwe rozszerzanie funkcjonalności testów [13].

## **1.13. Swagger / OpenAPI**

Swagger (obecnie rozwijany jako OpenAPI) to zestaw narzędzi służących do dokumentowania i testowania REST API. Specyfikacja OpenAPI pozwala opisać strukturę endpointów, formaty danych, typy odpowiedzi oraz metody HTTP. W projekcie wykorzystano bibliotekę drf-spectacular, która automatycznie generuje dokumentację Swaggera na podstawie kodu źródłowego Django REST Framework. Dokumentacja ta



stanowi wygodne źródło wiedzy dla frontendowców, testerów oraz przyszłych integratorów systemu [14].

### **1.14. PostgreSQL**

PostgreSQL to zaawansowany, obiektowo-relacyjny system zarządzania bazami danych o otwartym kodzie źródłowym. Jest ceniony za stabilność, zgodność ze standardem SQL, rozszerzalność i wsparcie dla skomplikowanych operacji (takich jak zapytania zagnieżdżone, indeksy pełnotekstowe czy JSON). Idealnie nadaje się do aplikacji wymagających silnej integralności danych [15].

### **1.15. Docker Compose**

Docker Compose to narzędzie służące do definiowania i uruchamiania wielokontenerowych aplikacji Dockera. Pozwala opisać całą infrastrukturę aplikacji w jednym pliku YAML, co upraszcza konfigurację i ułatwia uruchamianie środowisk developerskich oraz produkcyjnych [16].

### **1.16. GitHub Actions**

GitHub Actions to wbudowane w GitHub narzędzie do automatyzacji procesów związanych z integracją i wdrażaniem (CI/CD – Continuous Integration / Continuous Deployment). Umożliwia definiowanie workflowów w plikach YAML, które automatycznie uruchamiają się w odpowiedzi na zdarzenia w repozytorium, takie jak push, pull request czy publikacja wersji. W projekcie wykorzystano GitHub Actions do automatycznego uruchamiania testów (Pytest), sprawdzania poprawności kodu (lint) oraz budowania i uruchamiania kontenerów Dockerowych. Takie podejście zwiększa niezawodność wdrożeń oraz ułatwia utrzymanie wysokiej jakości kodu [17].

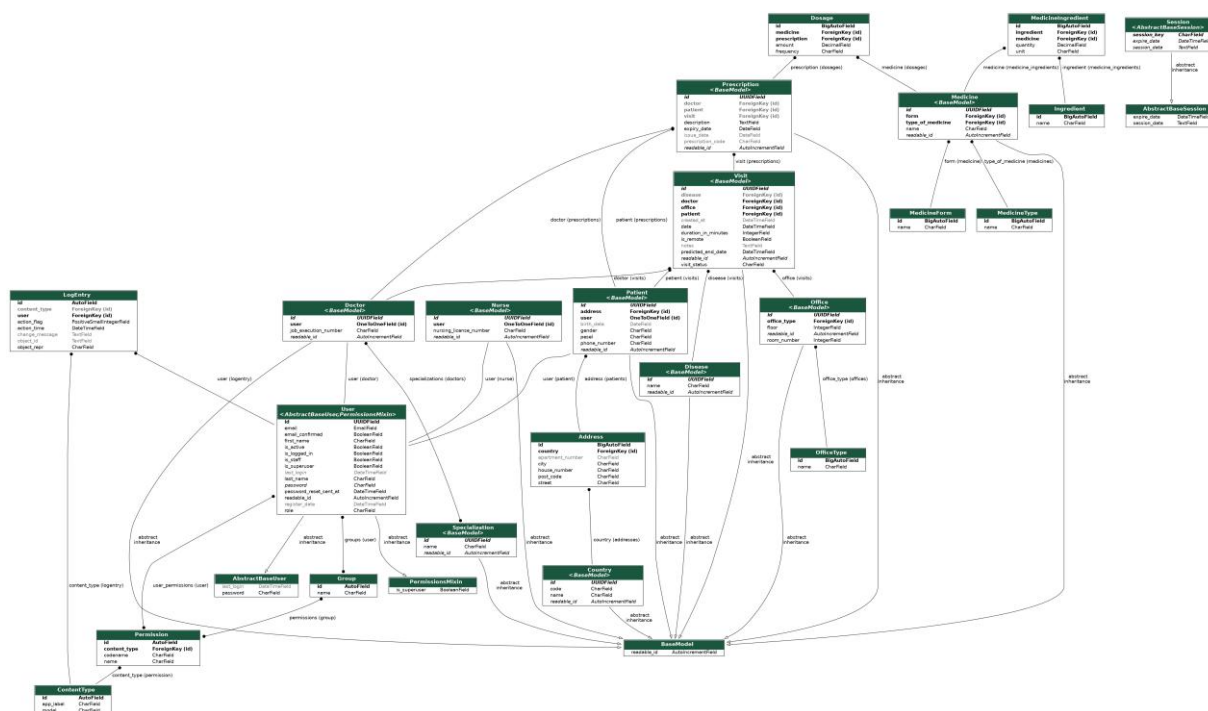
## 2. Architektura aplikacji

### 2.1. Baza danych

#### 2.1.1. Schemat bazy danych

Schemat bazy danych stanowi logiczny fundament systemu informatycznego, determinując sposób przechowywania, organizacji oraz powiązania danych w obrębie aplikacji. Poprawnie zaprojektowana struktura bazy danych pozwala nie tylko na efektywne zarządzanie informacją, ale także na bezpieczne, szybkie i spójne wykonywanie operacji związanych z przetwarzaniem danych.

Poniższy rysunek przedstawia pełną strukturę logiczną bazy danych stworzonej na potrzeby aplikacji. Graficzna reprezentacja modelu danych została przedstawiona na rysunku 1.



Rysunek 1: Schemat bazy danych

Rysunek ilustruje diagram encji i relacji (ERD), wygenerowany automatycznie na podstawie modeli aplikacji zbudowanej w środowisku Django. Diagram ukazuje zarówno encje reprezentujące konkretne obiekty w systemie (takie jak pacjenci, lekarze, wizyty czy leki), jak i relacje zachodzące między nimi.

Model danych został zaprojektowany w sposób modularny, z wyraźnym podziałem na domeny funkcjonalne. Wśród najważniejszych obszarów systemu można wyróżnić:

<b>Encja</b>	<b>Powiązania i relacje</b>
User	Dziedziczy po <code>AbstractBaseUser</code> i <code>PermissionsMixin</code> . Posiada relacje wiele-do-wielu z <code>Group</code> i <code>Permission</code> , co umożliwia elastyczne zarządzanie rolami i uprawnieniami. Może być powiązany z dokładnie jednym <code>Patient</code> , <code>Doctor</code> lub <code>Nurse</code> .
Patient	Powiązany relacją jeden-do-jednego z <code>User</code> . Każdy pacjent posiada przypisane konto użytkownika. Dodatkowo związany relacją wiele-do-jednego z <code>Address</code> . Może być powiązany z wieloma <code>Doctor</code> poprzez <code>Visit</code> .
Address	Przechowuje dane adresowe pacjenta. Może być współdzielony przez wielu pacjentów, choć w większości przypadków występuje relacja 1:1. Powiązany relacją wiele-do-jednego z <code>Country</code> .

Tabela 1: Struktura encji i relacje w module użytkownika i pacjenta

<b>Encja</b>	<b>Powiązania i relacje</b>
Doctor	Powiązany relacją jeden-do-jednego z <code>User</code> . Posiada relację wiele-do-wielu z <code>Specialization</code> . Może mieć wiele <code>Visit</code> oraz <code>Prescription</code> .
Nurse	Powiązany relacją jeden-do-jednego z <code>User</code> . W aktualnym modelu nie posiada bezpośrednich relacji z <code>Visit</code> , ale może być potencjalnie zaangażowany w realizację wizyt (np. w przyszłości).
Visit	Reprezentuje wizytę lekarską. Powiązana relacjami wiele-do-jednego z <code>Patient</code> , <code>Doctor</code> , <code>Office</code> oraz opcjonalnie z <code>Disease</code> . Może mieć przypisane <code>Prescription</code> .
Prescription	Reprezentuje receptę. Powiązana relacją wiele-do-jednego z <code>Visit</code> , <code>Patient</code> i <code>Doctor</code> . Może zawierać

	wiele leków (Medicine) poprzez model pośredni Dosage.
Medicine	Lek, który zawiera wiele Ingredient (substancji czynnych) poprzez model pośredni MedicineIngredient. Może występować w wielu Prescription.
Ingredient	Składnik aktywny leków. Powiązany relacją wiele-do-wielu z Medicine.
Dosage	Łączy Prescription z Medicine. Opisuje szczegóły dawkowania danego leku w ramach recepty.

Tabela 2: Struktura encji i relacje w module medycznym

Encja	Powiązania i relacje
Office	Reprezentuje gabinet lekarski. Powiązany relacją wiele-do-jednego z OfficeType. Wykorzystywany w Visit jako miejsce wizyty.

Tabela 3: Struktura encji i relacje w module lokalizacji

Encja	Powiązania i relacje
Group	Grupa użytkowników – relacja wiele-do-wielu z User.
Permission	Uprawnienia – relacja wiele-do-wielu z Group i User.
PermissionsMixin	Klasa bazowa rozszerzająca User o system uprawnień i grup Django.

Tabela 4: Struktura encji i relacje w module uprawnień i ról

Typ relacji	Przykład	Opis
Jeden-do-wielu	Doctor → Visit	Jeden lekarz może odbyć wiele wizyt.
Wiele-do-jednego	Visit → Patient, Doctor, Office	Jedna wizyta dotyczy jednego pacjenta, lekarza i odbywa się w jednym

gabinecie.

Jeden-do-jednego	Patient → User, Doctor → User, Nurse → User	Każda z tych ról przypisana jest do dokładnie jednego konta użytkownika.
Wiele-do-wielu	Doctor ↔ Specialization	Lekarz może mieć wiele specjalizacji i odwrotnie.
Wiele-do-wielu	Medicine ↔ Ingredient	Lek może zawierać wiele składników, a składnik może występować w wielu lekach.

Tabela 5: Podsumowanie relacji między encjami

Wspólną bazą dla wielu encji jest klasa „BaseModel”, która zawiera pola identyfikacyjne: numeryczny klucz główny id oraz pole „readable\_id”, będące czytelnym identyfikatorem generowanym na podstawie sekwencji. Taka konstrukcja ułatwia jednoznaczną identyfikację rekordów zarówno wewnątrz, jak i w kontekście komunikacji z użytkownikami.

Wszystkie relacje między encjami zostały odwzorowane przy pomocy kluczy obcych (ang. *foreign keys*), co zapewnia spójność referencyjną danych oraz umożliwia wykonywanie złożonych zapytań łączących dane z różnych obszarów systemu.

Schemat bazy danych stanowi zatem nie tylko odwzorowanie struktury danych, ale także logiczne odzwierciedlenie procesów biznesowych zachodzących w systemie. Jego przejrzystość i modularność umożliwiają dalszy rozwój aplikacji bez ryzyka naruszenia integralności danych.

### 2.1.2. Sekwencje w systemie identyfikatorów tabel

W celu zapewnienia czytelności identyfikatorów wykorzystywanych w systemie, obok tradycyjnego, wewnętrznego pola „id” zastosowano również dodatkowe pole „readable\_id”. Jest to alternatywny identyfikator, generowany automatycznie przy użyciu dedykowanych sekwencji bazodanowych.

Sekwencje zostały utworzone oddzielnie dla każdego modelu dziedziczącego po klasie „BaseModel” i służą do nadawania unikalnych, kolejnych numerów, które są łatwe do odczytu i komunikacji z użytkownikiem. W przeciwieństwie do klasycznego identyfikatora

„id”, który pełni funkcję techniczną (klucz główny w tabeli), pole „readable\_id” może być wykorzystywane m.in. w interfejsach użytkownika, raportach, numeracji dokumentów lub systemach odniesienia między modułami.

Poniżej przedstawiono przykładowy fragment kodu SQL tworzący sekwencje dla wybranych modeli systemu:

```
DO $$
DECLARE
    seq_name TEXT;
    seq_names TEXT[] := ARRAY [
        'clinic_user_readable_id_seq',
        'clinic_doctor_readable_id_seq',
        # Inne sekwencje
    ];
BEGIN
    FOR i IN 1..array_length(seq_names, 1) LOOP
        seq_name := seq_names[i];
        EXECUTE format('CREATE SEQUENCE IF NOT EXISTS %I START WITH 1
INCREMENT BY 1;', seq_name);
    END LOOP;
END $$;
```

Listing 1: Skrypt w PostgreSQL generujący sekwencje przyjaznych identyfikatorów

Dzięki takiemu rozwiązaniu każdy rekord w tabeli otrzymuje nie tylko wewnętrzny identyfikator „id”, ale również bardziej „przyjazny” identyfikator „readable\_id”, który może być bezpiecznie prezentowany użytkownikom bez obaw o ujawnienie struktury wewnętrznej bazy danych.

## 2.2. Backend

### 2.2.1. Struktura katalogów projektu

Kod źródłowy backendu został uporządkowany w przejrzystą strukturę katalogów, zgodną z dobrymi praktykami Django oraz zasadami modularności. Poniżej przedstawiono ogólny opis głównych katalogów projektu:

#### 1. src/

Główny katalog z kodem aplikacji Django. Zawiera dwa podmoduły:

- core/ – konfiguracja globalna projektu:
  - ustawienia (settings.py),
  - routing (urls.py),
  - pliki uruchomieniowe (asgi.py, wsgi.py).
- clinic/ – główny moduł domenowy, podzielony na kilka logicznych obszarów:
  - auth/ – obsługa logowania, rejestracji, tokenów i resetu hasła.

- `roles/` – logika ról użytkowników (lekarz, pielęgniarka, pacjent).
- `treatment/` – moduł odpowiedzialny za leczenie (wizyty, recepty).
- `dictionaries/` – słowniki danych: choroby, leki, specjalizacje.
- `management/` – własne komendy Django (`load_data.py`, `create_sequences.py`).
- `templates/` – szablony HTML e-maili (weryfikacja, reset hasła).
- `migrations/` – migracje modeli Django.
- pliki wspólne, jak `models.py`, `serializers.py`, `filters.py`, `validators.py`.

## 2. tests/

Zestaw testów jednostkowych i integracyjnych, odzwierciedlający strukturę katalogu `clinic`. Testy są podzielone na moduły:

- `tests/clinic/auth/` – testy widoków logowania, rejestracji itd.
- `tests/clinic/roles/` – testy API dla ról użytkowników.
- `tests/clinic/treatment/` – testy dla wizyt, recept.
- `tests/clinic/dictionaries/` – testy słowników.
- `conftest.py` – plik z fixture'ami Pytest.

## 3. Katalog główny projektu

Zawiera pliki konfiguracyjne i wspomagające uruchamianie aplikacji:

- `manage.py` – plik główny do zarządzania Django.
- `Dockerfile`, `entrypoint.sh` – pliki konteneryzacji aplikacji.
- `requirements.txt`, `pyproject.toml` – zależności projektu.
- `coverage.xml`, `pytest.ini` – konfiguracja testów i pokrycia kodu.

### 2.2.2. Zmienne środowiskowe

W celu zapewnienia elastycznej konfiguracji aplikacji backendowej oraz oddzielenia danych wrażliwych i środowiskowych od kodu źródłowego, w projekcie zastosowano mechanizm **zmiennych środowiskowych**. Pozwalają one na definiowanie ustawień takich jak dane dostępowe do bazy danych, konfiguracja poczty, klucze rejestracyjne czy ścieżki plików, bez konieczności modyfikowania kodu źródłowego.

Zmienne środowiskowe wczytywane są automatycznie z pliku `.env`, który jest przekazywany do kontenera backendu za pomocą opcji `env_file` w pliku `docker-compose.yml`. Dzięki temu rozwiązaniu możliwe jest łatwe dostosowanie konfiguracji aplikacji do różnych środowisk (np. lokalne, testowe, produkcyjne).

Nazwa zmiennej	Opis
DB_NAME	Nazwa bazy danych PostgreSQL.
DB_USER	Nazwa użytkownika bazy danych.
DB_PASSWORD	Hasło użytkownika bazy danych.
DB_HOST	Adres hosta bazy danych (dla Dockera zazwyczaj db lub database).
DB_PORT	Port bazy danych (domyślnie 5432 dla PostgreSQL).
DJANGO_SECRET_KEY	Sekretny klucz Django – używany do podpisywania danych (np. JWT).
DJANGO_SUPERUSER_FIRST_NAME	Imię superużytkownika Django tworzonego przy starcie kontenera.
DJANGO_SUPERUSER_LAST_NAME	Nazwisko superużytkownika Django.
DJANGO_SUPERUSER_EMAIL	Adres e-mail superużytkownika.
DJANGO_SUPERUSER_PASSWORD	Hasło superużytkownika Django.
EMAIL_HOST	Adres serwera SMTP (np. smtp.gmail.com).
EMAIL_PORT	Port serwera SMTP (domyślnie 587).
EMAIL_USE_TLS	Czy używać TLS do wysyłki wiadomości (True / False).
EMAIL_HOST_USER	Użytkownik serwera e-mail (np. adres Gmail).
EMAIL_HOST_PASSWORD	Hasło do konta e-mailowego.
DEFAULT_FROM_EMAIL	Domyślny adres nadawcy e-mail.
EMAIL_FILE_PATH	Ścieżka zapisu wiadomości e-mail w trybie deweloperskim.
RECAPTCHA_SECRET_KEY	Sekretny klucz reCAPTCHA v2 używany do weryfikacji użytkowników.
RECAPTCHA_VERIFY_URL	Adres endpointa API Google do weryfikacji tokenu reCAPTCHA.
TEST_DB_NAME	Nazwa testowej bazy danych wykorzystywanej w czasie testów Pytest.



Nazwa zmiennej	Opis
FRONTEND_URL	Adres frontendowej części aplikacji (np. <code>http://localhost:4200</code> ).

Tabela 6: Wykorzystywane zmienne środowiskowe

Dzięki wykorzystaniu zmiennych środowiskowych możliwe było:

- oddzielenie danych konfiguracyjnych od kodu źródłowego,
- poprawienie bezpieczeństwa aplikacji (brak wrażliwych danych w repozytorium),
- łatwe wdrażanie aplikacji w różnych środowiskach,
- centralne zarządzanie konfiguracją bez modyfikowania aplikacji.

Zmienna `DJANGO_SECRET_KEY` oraz hasła i tokeny (np. `EMAIL_HOST_PASSWORD`, `RECAPTCHA_SECRET_KEY`) powinny być szczególnie chronione i **nie powinny być nigdy wersjonowane**. W przypadku wdrażania aplikacji produkcyjnej zaleca się ich przechowywanie w bezpiecznym menedżerze tajemnic (np. Docker secrets, Vault, AWS Secrets Manager).

### 2.2.3. Dane inicjalne

W celu usprawnienia pierwszego uruchomienia systemu oraz zapewnienia spójnej bazy danych w środowisku deweloperskim, testowym i produkcyjnym, zaimplementowano mechanizm ładowania danych inicjalnych z plików JSON.

Dane te obejmują między innymi:

- listę chorób,
- wykaz leków,
- dostępne specjalizacje lekarskie,
- gabinety.

Pliki z danymi znajdują się w katalogu `src/clinic/dictionaries/data/`, a ich struktura została dostosowana do modeli Django wykorzystywanych w systemie. Do ich załadowania służy dedykowana komenda zarządzania Django: `python manage.py load_data`.

Podczas działania komendy parser wczytuje dane z plików JSON, waliduje je i zapisuje do bazy danych przy pomocy odpowiednich serializerów. Mechanizm ten pozwala łatwo inicjalizować bazę danych bez konieczności ręcznego wprowadzania informacji przez panel administracyjny.

Dla potrzeb projektu dane słownikowe zostały wygenerowane przy pomocy sztucznej inteligencji, a następnie ręcznie dostosowane do potrzeb aplikacji oraz zweryfikowane pod względem struktury i spójności. Takie podejście pozwoliło szybko przygotować realistyczne dane testowe bez konieczności korzystania z rzeczywistych rejestrów lub danych wrażliwych.

Dodatkowo, lista krajów została automatycznie zaimportowana przy użyciu biblioteki `django-countries` [18], która zapewnia zestandaryzowany zestaw nazw państw zgodny z normą ISO 3166-1. Pozwoliło to uniknąć ręcznego definiowania danych geograficznych i zapewniło wysoką jakość oraz kompatybilność informacji krajowych w aplikacji.

Zastosowanie tego rozwiązania zapewnia:

- spójność danych bazowych między środowiskami,
- szybkie przygotowanie systemu do testowania,
- łatwość aktualizacji słowników przy pomocy edycji plików JSON,
- zgodność z międzynarodowymi standardami dotyczącymi nazw krajów.

W przyszłości planowana jest również możliwość pobierania danych słownikowych z zewnętrznych źródeł (np. API GUS, URPL), co zwiększy aktualność oraz automatyzację procesu.

#### 2.2.4. Autoryzacja i JWT

W aplikacji zastosowano mechanizm autoryzacji oparty o tokeny JWT (JSON Web Token), co umożliwia bezpieczną i skalowalną obsługę logowania użytkowników oraz autoryzacji żądań. JWT jest popularnym rozwiązaniem w architekturach opartych na SPA (Single Page Application), takich jak Angular + Django REST API, ze względu na brak konieczności przechowywania sesji po stronie serwera. Do implementacji mechanizmu autoryzacji wykorzystano bibliotekę `django-rest-framework-simplejwt` [19], będącą rozszerzeniem Django REST Framework. Biblioteka ta pozwala na generowanie oraz weryfikację dwóch rodzajów tokenów:

- **Access token** – krótkożyjący token (np. 5–15 minut), przesyłany w nagłówku `Authorization: Bearer`, służący do autoryzacji bieżących żądań. W aplikacji czas życia access tokena został ustawiony na 15 minut.
- **Refresh token** – token o dłuższym czasie ważności (np. kilka dni), wykorzystywany do odświeżenia access tokena bez konieczności ponownego logowania użytkownika.

Proces logowania realizowany jest za pośrednictwem dedykowanego endpointa, który przyjmuje adres e-mail oraz hasło użytkownika. W przypadku poprawnych danych

uwierzytelniających, system generuje parę tokenów JWT oraz zwraca podstawowe informacje o użytkowniku. Dla zwiększenia bezpieczeństwa wprowadzono następujące zabezpieczenia:

- Weryfikacja statusu konta – użytkownik nie może się zalogować, jeśli jego adres e-mail nie został wcześniej potwierdzony lub konto jest nieaktywne.
- Obsługa flagi `is_logged_in` – system zapobiega wielokrotnemu logowaniu tego samego użytkownika jednocześnie, blokując próbę zalogowania, jeśli flaga ta jest już ustawiona.
- Obsługa tokenów JWT – access token musi być dołączony do każdego zapytania do endpointów chronionych, a w przypadku jego wygaśnięcia klient frontendowy może wysłać refresh token, aby uzyskać nowy access token bez konieczności ponownego logowania.
- Mechanizm force logout – w przypadku wygaszenia sesji po stronie frontendowej przez przekroczenie czasu życia tokena, system umożliwia wymuszone wylogowanie użytkownika poprzez odpowiedni endpoint, który czyści flagę `is_logged_in`.

Dodatkowo użytkownicy mogą zresetować hasło, korzystając z mechanizmu generowania jednorazowego linku wysyłanego e-mailem. Link ten zawiera zaszyfrowany identyfikator użytkownika oraz token weryfikacyjny, który pozwala bezpiecznie zidentyfikować użytkownika i umożliwić zmianę hasła. Weryfikacja tokenu odbywa się przy pomocy wbudowanego generatora `default_token_generator` z Django.

Dzięki zastosowaniu JWT możliwe było pełne oddzielenie warstwy frontendowej od backendu, bez konieczności wykorzystywania ciasteczek ani serwerowych sesji. Tokeny są generowane po stronie backendu i przechowywane w przeglądarce użytkownika, np. w mechanizmie `localStorage` [20]. Choć rozwiązanie to pozwala frontendowi utrzymywać stan zalogowania użytkownika, nie wymaga ono przechowywania jakichkolwiek danych sesyjnych po stronie serwera – backend pozostaje bezstanowy (*ang. stateless*).

Takie podejście zapewnia zarówno bezpieczeństwo transmisji danych, jak i wysoką elastyczność, pozwalającą na łatwą integrację z różnymi klientami (np. aplikacjami mobilnymi). Jednocześnie aplikacja obsługuje pełen cykl życia użytkownika – od rejestracji, przez logowanie, potwierdzanie adresu e-mail, po resetowanie i zmianę hasła.

Wszystkie hasła użytkowników są automatycznie hashowane przy użyciu bezpiecznego algorytmu (domyślnie **PBKDF2** z **SHA256**) wbudowanego w Django. Dane te nigdy nie są przechowywane w postaci jawnej (plaintext), co zapewnia zgodność z najlepszymi praktykami w zakresie bezpieczeństwa i ochrony danych osobowych.

### 2.2.5. System uprawnień (permissions)

W celu zapewnienia kontroli dostępu do zasobów API, w systemie zastosowano mechanizm uprawnień (permissions) oparty na wbudowanych i rozszerzalnych klasach Django REST Framework. Permisje określają, którzy użytkownicy mają prawo do wykonania żądania na danym widoku – niezależnie od poprawnej autoryzacji tokenem JWT.

W aplikacji wykorzystano dwie podstawowe klasy wbudowane w DRF:

- `IsAuthenticated` – wymaga, aby użytkownik był zalogowany (posiadał poprawny token JWT),
- `AllowAny` – umożliwia dostęp wszystkim użytkownikom, również anonimowym (np. do rejestracji, logowania, resetowania hasła).

Dodatkowo, ze względu na obecność różnych typów użytkowników (admin, lekarz, pielęgniarka, pacjent), zdefiniowano własne klasy uprawnień, które ograniczają dostęp do wybranych widoków na podstawie roli użytkownika. Każda z tych klas dziedziczy po `BasePermission` i nadpisuje metodę `has_permission`, porównując rolę zalogowanego użytkownika z wymaganą.

Przykładowo, dostęp do widoków związanych z zarządzaniem użytkownikami może być ograniczony tylko dla administratora, natomiast dostęp do danych medycznych – wyłącznie dla lekarza lub pielęgniarki. Poniżej przedstawiono klasę uprawnień pacjenta:

```
class IsPatient(permissions.BasePermission):
    def has_permission(self, request, view):
        return request.user.is_authenticated and request.user.role == Role.PATIENT
```

Listing 2: Przykładowa klasa uprawnień dla roli administratora

### 2.2.6. reCAPTCHA

W celu ochrony systemu przed nadużyciami ze strony botów i zautomatyzowanych skryptów, w wybranych punktach aplikacji zastosowano mechanizm weryfikacji typu reCAPTCHA. Rozwiązanie to pochodzi od firmy Google i polega na analizie zachowania użytkownika w celu odróżnienia człowieka od programu komputerowego.

W projekcie wykorzystano reCAPTCHA v2 typu „I'm not a robot”, która oferuje prosty interfejs weryfikacyjny i może być zintegrowana zarówno po stronie frontendowej (Angular), jak i backendowej (Django).

Weryfikacja odbywa się dwustopniowo:

1. Po stronie frontendowej reCAPTCHA generuje token, który potwierdza pozytywne przejście testu przez użytkownika.
2. Token ten przesyłany jest do backendu, gdzie przy pomocy zapytania do API Google reCAPTCHA następuje walidacja jego poprawności.

W systemie reCAPTCHA została zaimplementowana w dwóch krytycznych punktach:

- **Formularz rejestracji użytkownika** – ochrona przed masowym zakładaniem kont przy użyciu botów.
- **Formularz resetowania hasła** – ochrona przed zautomatyzowanym wysyłaniem żądań resetu hasła, które mogłyby prowadzić do ataków typu spam lub denial of service.

Weryfikacja odbywa się w metodzie `validate_recaptcha_response`, której implementacja znajduje się w serializerach rejestracji (`UserRegisterSerializer`) oraz resetowania hasła (`ResetPasswordSerializer`). W przypadku niepowodzenia walidacji reCAPTCHA, użytkownik otrzymuje komunikat informujący o błędnej próbie weryfikacji.

Dzięki zastosowaniu tego mechanizmu możliwe było zwiększenie poziomu bezpieczeństwa aplikacji oraz ograniczenie ryzyka nadużyć bez negatywnego wpływu na komfort użytkowników.

Poniżej znajduje się metoda walidująca pole z frontendu oraz metoda do walidacji odpowiedzi recaptcha.

```
def validate_recaptcha_response(self, value):
    if not verify_recaptcha(value):
        raise serializers.ValidationError(
            _("Nieprawidłowa reCAPTCHA. Spróbuj ponownie.")
        )
    return value
```

Listing 3: Walidacja pola `recaptcha_response` w serializerze

```
def verify_recaptcha(response):
    data = { "secret": settings.RECAPTCHA_SECRET_KEY, "response": response }
    response = requests.post(settings.RECAPTCHA_VERIFY_URL, data=data)
    result = response.json()
    return result.get("success")
```

Listing 4: Metoda sprawdzająca odpowiedź reCAPTCHA

### 2.2.7. Wysyłka wiadomości e-mail

W projekcie zastosowano mechanizm wysyłania wiadomości e-mail w celu obsługi dwóch kluczowych procesów: **potwierdzania rejestracji użytkownika** oraz **resetowania hasła**.

Wiadomości generowane są dynamicznie na podstawie dedykowanych szablonów HTML i zawierają unikalne linki umożliwiające wykonanie odpowiednich akcji.

W przypadku rejestracji system automatycznie wysyła e-mail zawierający link weryfikacyjny. Link ten zawiera zakodowany identyfikator użytkownika (uidb64) oraz token, który umożliwia bezpieczne potwierdzenie adresu e-mail. Po kliknięciu w link użytkownik zostaje oznaczony jako aktywny i może zalogować się do systemu.

Podobnie, w procesie resetowania hasła użytkownik otrzymuje e-mail zawierający link do formularza ustawienia nowego hasła. Link jest ważny czasowo, a jego poprawność weryfikowana jest przy pomocy wbudowanego mechanizmu `default_token_generator`.

Za wysyłkę wiadomości odpowiada specjalny **mixin** – `MailSendingMixin`, który udostępnia metodę `send_email`. Metoda ta odpowiada za renderowanie wiadomości z użyciem szablonu i kontekstu, a następnie wysłanie jej na wskazany adres e-mail. W przypadku błędu, np. problemów z serwerem SMTP, błąd zostaje zalogowany i zwrócony jako wyjątek walidacyjny.

Poniżej przedstawiono implementację klasy `MailSendingMixin`:

```
class MailSendingMixin:
    def send_email(self, subject, template_name, context, to_email):
        try:
            message = render_to_string(template_name, context)
            email_message = EmailMessage(subject, message, to=[to_email])
            email_message.content_subtype = "html"
            email_message.send()
        except Exception as e:
            logger.error(
                f"Error sending email to {to_email}: {str(e)}", exc_info=True
            )
            raise serializers.ValidationError({
                "non_field_errors": [_("Wystąpił błąd podczas wysyłania e-maila.")]}
            )
```

Listing 5: Klasa odpowiedzialna za wysyłkę wiadomości e-mail

### 2.2.8. Throttling

Throttling to mechanizm ograniczający liczbę żądań, jakie mogą zostać wysłane do API w określonym czasie. Jego głównym celem jest ochrona systemu przed przeciążeniem, nadużyciami oraz potencjalnymi atakami typu brute-force lub denial of service (DoS). Mechanizm ten działa na poziomie Django REST Framework i może być stosowany globalnie lub indywidualnie dla poszczególnych widoków.

W aplikacji throttling został zastosowany w sposób zróżnicowany – w zależności od roli użytkownika lub jego statusu (anonimowy/zalogowany). Dzięki temu możliwe było dostosowanie poziomu ograniczeń do charakteru aktywności danego typu użytkownika:

- **Użytkownik anonimowy** – może wykonać maksymalnie 100 żądań na godzinę.
- **Użytkownik zalogowany (bez określonej roli)** – limit wynosi 1000 żądań na godzinę.
- **Lekarz (rola: doctor)** – limit ustawiono na 1000 żądań na godzinę, ze względu na potencjalnie intensywną pracę z systemem.
- **Pielęgniarka (rola: nurse)** – może wysłać do 750 żądań na godzinę.
- **Pacjent (rola: patient)** – limit został ustalony na 500 żądań na godzinę.

Takie podejście umożliwia optymalną ochronę zasobów serwera przy jednoczesnym zachowaniu komfortu użytkowników końcowych. Wysokie limity dla personelu medycznego odzwierciedlają ich większą aktywność operacyjną w systemie, natomiast niższe limity dla pacjentów i użytkowników anonimowych ograniczają ryzyko przeciążenia aplikacji przez niewłaściwe użycie. Mechanizm throttlingu działa automatycznie i zwraca odpowiedź HTTP 429 („Too Many Requests”), gdy użytkownik przekroczy dozwolony limit żądań.

### 2.2.9. Architektura API i ViewSety

Backend aplikacji został zbudowany w oparciu o architekturę REST i framework Django REST Framework (DRF), co pozwoliło na szybkie tworzenie rozbudowanych, ale czytelnych i modularnych interfejsów API. Kluczowe komponenty DRF, na których oparto architekturę systemu, to:

- **Modele (models)** – klasy Pythona odwzorowujące strukturę bazy danych. Przechowują dane oraz podstawową logikę biznesową, np. `Visit`, `Prescription`, `Patient`.
- **Serializery (serializers)** – odpowiedzialne za przekształcanie obiektów Pythona (modeli) do formatu JSON i odwrotnie. Umożliwiają także walidację danych wprowadzanych przez użytkownika, np. `VisitWriteSerializer`.
- **ViewSety (viewsets.ModelViewSet)** – klasy obsługujące logikę HTTP (GET, POST, PATCH, DELETE). Automatyzują operacje CRUD i pozwalają łatwo kontrolować uprawnienia, paginację i filtrowanie.
- **Querysety** – zapytania do bazy danych reprezentowane jako obiekty Pythona. Pozwalają w elastyczny sposób filtrować, sortować i przetwarzać dane np. zależnie od roli użytkownika (`get_visit_queryset`).

Każdy ViewSet został dostosowany do specyfiki modelu oraz logiki biznesowej poprzez:

- **dynamiczne dobieranie serializerów** w zależności od rodzaju akcji (list, retrieve, create, itp.) – np. `VisitWriteSerializer` do zapisu, `VisitReadSerializer` do odczytu,
- **zastosowanie klas uprawnień (permissions)** – m.in. `IsDoctor`, `IsPatient`, `IsAdmin`, które ograniczają dostęp do wybranych operacji,
- **dynamiczne filtrowanie querysetu** w metodzie `get_queryset`, w zależności od roli użytkownika,
- **ręczne przypisywanie throttlingu** (`get_throttles`) – np. ograniczenie liczby wystawionych recept przez lekarza,
- **obsługę paginacji** przy pomocy klasy `StandardResultsSetPagination`,
- **możliwość filtrowania i sortowania wyników** za pomocą `DjangoFilterBackend` i `OrderingFilter`.

Przykład pełnego przepływu danych modelu: `Visit`:

Aby zobrazować sposób działania komponentów Django REST Framework, poniżej przedstawiono przykładową implementację obsługi modelu `Visit` – od modelu, przez serializer i queryset, aż po filtr i viewset.

```
class Visit(BaseModel):
    date = models.DateTimeField(_("visit date"))
    duration_in_minutes = models.IntegerField(_("duration [min]"))
    visit_status = models.CharField(
        _("visit status"),
        max_length=1,
        choices=VisitStatus.choices,
        default=VisitStatus.SCHEDULED,
    )
    # Inne pola

    def save(self, *args, **kwargs):
        # Ustawienie daty końca wizyty oraz statusu na podstawie czasu trwania i daty wizyty
        super().save(*args, **kwargs)

    def __str__(self):
        patient_str = f"Patient: {self.patient}"
        doctor_str = f"Doctor: {self.doctor}"
        date_str = self.date.strftime("%d.%m.%Y, %I:%M")
        return f"Visit ({date_str}) - {patient_str}, {doctor_str}"

    class Meta:
        verbose_name = _("visit")
        verbose_name_plural = _("visits")
```

Listing 6: Model wizyty

```
class VisitWriteSerializer(serializers.ModelSerializer):
    def validate_duration_in_minutes(self, value):
        # Walidacja czasu trwania wizyty
        return value
```



```

def validate(self, data):
    # Logika walidacji wizyty
    return super().validate(data)
# Inne metody

class Meta:
    model = Visit
    fields = "__all__"
    read_only_fields = ("id",) # Inne pola

```

Listing 7: Serializer do zapisu wizyty

```

def get_visit_queryset(user):
    if user.role in (Role.ADMIN, Role.NURSE):
        return Visit.objects.all()
    # Inne warunki
    return Visit.objects.none()

```

Listing 8: Queryset wizyty

```

class VisitFilter(BaseTreatmentFilterSet):
    date = filters.DateTimeFromToRangeFilter()
    duration_in_minutes = filters.RangeFilter()
    visit_status = filters.ChoiceFilter(choices=VisitStatus.choices)
    office__office_type__name = filters.CharFilter(lookup_expr="icontains")
    # Inne pola

class Meta:
    model = Visit
    fields = BaseTreatmentFilterSet.Meta.fields + ("date",) # Inne pola

```

Listing 9: FilterSet wizyty

```

class VisitViewSet(viewsets.ModelViewSet):
    permission_classes = (IsNurse | IsDoctor | IsAdmin | IsPatient,)
    queryset = Visit.objects.all()
    filter_backends = (DjangoFilterBackend, OrderingFilter)
    filterset_class = VisitFilter
    ordering = ("readable_id",)
    ordering_fields = (
        "date", # Inne pola
    )
    pagination_class = StandardResultsSetPagination
    http_method_names = ("get", "post", "delete", "patch", "head", "options")

    def get_permissions(self):
        if self.action in ("create", "destroy", "partial_update"):
            self.permission_classes = (IsNurse | IsAdmin,)
        return super().get_permissions()

    def get_serializer_class(self):
        if self.action in ("list", "retrieve"):
            return VisitReadSerializer
        return VisitWriteSerializer

    def get_queryset(self):
        return get_visit_queryset(self.request.user)

    def get_throttles(self):
        if self.action in ("create", "destroy", "partial_update"):
            self.throttle_classes = (NurseRateThrottle,)
        return super().get_throttles()

```

Listing 10: ViewSet wizyty

### 2.2.10. Dokumentacja API (Swagger / OpenAPI)

W celu zapewnienia przejrzystej i interaktywnej dokumentacji interfejsu API, w projekcie wykorzystano bibliotekę `drf-spectacular` [21] – rozszerzenie Django REST Framework umożliwiające generowanie specyfikacji **OpenAPI 3.0**. Na podstawie zdefiniowanych serializerów, widoków, metod oraz dekoratorów `@extend_schema`, tworzona jest dokumentacja API dostępna w formacie Swagger UI.

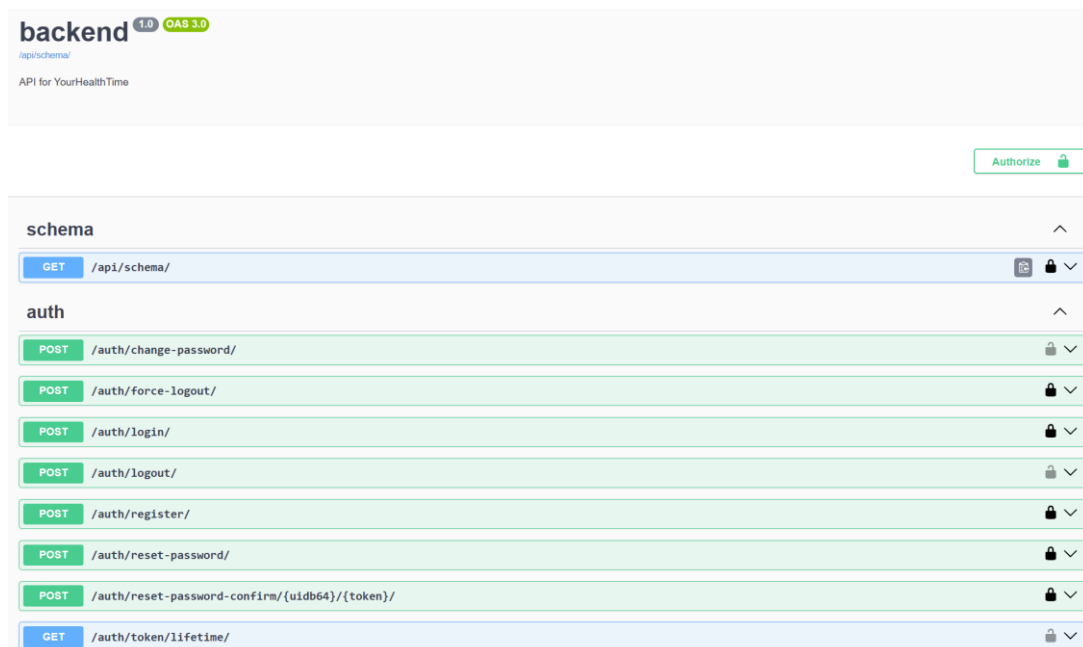
Dzięki tej integracji możliwe jest:

- przeglądanie wszystkich endpointów systemu,
- analizowanie struktury danych wejściowych i wyjściowych,
- testowanie zapytań HTTP bezpośrednio z poziomu przeglądarki,
- ułatwienie współpracy z frontendem i innymi klientami API.

Interfejs Swaggera generowany jest automatycznie i aktualizuje się wraz ze zmianami w kodzie backendu, co pozwala utrzymać spójną i zawsze aktualną dokumentację systemu.

```
@extend_schema(  
    methods=("post", "patch"),  
    request=VisitWriteSerializer,  
    responses={200: VisitReadSerializer},  
)  
class VisitViewSet(viewsets.ModelViewSet):  
    ...
```

Listing 11: Przykład zastosowania dekoratora `@extend_schema` w widoku



Rysunek 2: Interfejs Swagger UI

### 2.2.11. Testy w Pytest

W projekcie zastosowano framework **Pytest** jako główne narzędzie do testowania backendu opartego na Django. Dzięki integracji z biblioteką `pytest-django` [25], możliwe było tworzenie czytelnych, parametryzowanych i wydajnych testów jednostkowych oraz integracyjnych.

Testy weryfikują m.in.:

- poprawność działania endpointów API,
- reakcję systemu na nieprawidłowe dane lub błędne zapytania,
- ograniczenia wynikające z autoryzacji i ról użytkowników,
- poprawność walidacji i komunikatów o błędach,
- obsługę sytuacji brzegowych, np. nieistniejących zasobów czy błędnych tokenów.

Poniżej przedstawiono przykład trzech testów do różnych przypadków testowych:

```
@pytest.mark.django_db
def test_verify_email_successful(api_client, user_not_confirmed,
email_verification_url):
    response = api_client.get(email_verification_url)
    user_not_confirmed.refresh_from_db()
    assert (
        response.status_code,
        user_not_confirmed.email_confirmed,
        user_not_confirmed.is_active,
    ) == (status.HTTP_200_OK, True, True)
```

Listing 12: Test weryfikujący poprawną aktywację konta użytkownika

```
@pytest.mark.django_db
def test_verify_email_invalid_link(api_client):
    bad_uidb64 = "invalidUID"
    bad_token = "invalidToken"
    url = reverse("verify-email", args=(bad_uidb64, bad_token))
    response = api_client.get(url)
    expected_error = ErrorDetail(
        string="Nieprawidłowy identyfikator użytkownika.", code="invalid_uid"
    )
    expected_response = {"non_field_errors": [expected_error]}
    assert (response.status_code, response.data) == (status.HTTP_400_BAD_REQUEST,
expected_response)
```

Listing 13: Test weryfikujący walidację identyfikatora użytkownika

```
@pytest.mark.django_db
@pytest.mark.parametrize(
    "user_fixture",
    (("authenticated_doctor"), ("authenticated_nurse"), ("authenticated_patient")),
)
def test_user_role_can_list_offices(request, user_fixture, office_instances):
    api_client, _ = request.getfixturevalue(user_fixture)
    url = reverse("office-list")
    ordering = "readable_id"
    response = api_client.get(f"{url}?ordering={ordering}")
    assert (response.status_code, len(response.data)) == (status.HTTP_200_OK, 6)
```

Listing 14: Test sparametryzowany

## 2.3. Frontend

### 2.3.1. Struktura katalogów projektu

Kod źródłowy Kod źródłowy frontendowej części aplikacji został uporządkowany zgodnie z dobrymi praktykami Angulara oraz zasadą podziału na warstwy funkcjonalne i domenowe.

Poniżej przedstawiono ogólną strukturę katalogów projektu:

1. `src/app/`

Główny katalog aplikacji Angular, zawierający wszystkie moduły funkcjonalne i warstwę wspólną:

- `auth/` – moduł odpowiedzialny za autoryzację użytkownika: logowanie, rejestrację, reset hasła, weryfikację e-maila.
  - `components/` – formularze i dialogi związane z logowaniem i rejestracją.
  - `services/` – logika biznesowa (np. `auth.facade.ts`).
  - `interceptors/` – przechwytywanie żądań HTTP i dodawanie tokenów.
  - `misc/` – komunikaty, stałe i konfiguracje autoryzacji.
  - `types/` – interfejsy danych używanych w module.
- `core/` – warstwa infrastrukturalna wspólna dla całej aplikacji.
  - `components/` – wspólne komponenty (np. `spinner`).
  - `directives/` – dyrektywy strukturalne i atrybutowe (np. `permission.directive.ts`).
  - `guards/` – strażnicy tras (np. `role.guard.ts`).
  - `interceptors/` – ogólne przechwytywacze żądań HTTP.
  - `initializers/` – funkcje inicjalizujące aplikację.
  - `misc/` – konfiguracja endpointów, obsługa błędów, pomocnicze funkcje.
  - `pipes/` – niestandardowe filtry Angulara.
  - `services/` – usługi globalne (np. `api.service.ts`, `toast.service.ts`).
  - `types/` – typy danych wykorzystywane globalnie.
  - `validators/` – walidatory formularzy, podzielone na podkatalogi według pola (np. `pesel.validator.ts`, `post-code.validator.ts`).
- `dictionaries/` – moduł słowników danych (kraje, choroby, leki, specjalizacje, gabinety).

Każdy słownik ma własną strukturę katalogów:

- `pages/` – komponenty listujące elementy słownika.
- `services/` – usługi pobierające dane.

- `types/` – interfejsy danych.
  - `misc/` – komunikaty, tłumaczenia, stałe.
- `layout/` – moduł odpowiedzialny za układ i interfejs użytkownika.
  - `components/` – komponenty interfejsu użytkownika: menu, dashboardy, zegar sesji, informacje o użytkowniku.
  - `pages/` – komponenty układu głównego.
  - `services/` – logika layoutu, np. obsługa menu i timera sesji.
  - `types/`, `misc/` – definicje struktur danych i komunikaty.
- `prescriptions/`, `visits/`, `roles/` – moduły domenowe odpowiedzialne kolejno za recepty, wizyty i role użytkowników (lekarze, pielęgniarki, pacjenci).  
Struktura każdego z modułów obejmuje:
  - `components/` – formularze i elementy formularzowe.
  - `pages/` – komponenty stron (np. lista wizyt).
  - `services/` – fasady i serwisy do komunikacji z backendem.
  - `misc/`, `types/` – komunikaty, tłumaczenia, interfejsy.
- `shared/` – komponenty współdzielone, podzielone na podmoduły:
  - `shared-form/` – komponenty formularzy (np. `input-field`, `autocomplete-field`), wspólne typy, klasy bazowe (`field.component-base.ts`), dyrektywy formularzowe.
  - `shared-table/` – komponenty tabel z filtrowaniem, sortowaniem, paginacją.
  - `shared-confirm/` – dialog potwierdzenia akcji.
- 2. `src/environments/` - środowiska konfiguracyjne Angulara (`environment.ts`, `environment.prod.ts`), wykorzystywane do przełączania między wersją deweloperską a produkcyjną.
- 3. `src/assets/` - zasoby statyczne, takie jak obrazy i logotypy (`logo.png`, `undraw_medicine_hqgg.png`).
- 4. `src/styles/` - zestaw globalnych stylów aplikacji:
  - `theme/`, `utils/`, `vendors/`, `widgets/` – style Material, zmienne SCSS, style zewnętrzne (np. bootstrap), widżety formularzy i tabel.

Dzięki modularnej organizacji kodu, projekt frontendowy zachowuje wysoką czytelność, łatwość w testowaniu, możliwość ponownego użycia komponentów oraz prostotę w dalszym rozwoju aplikacji.

### 2.3.2. Architektura aplikacji i podejście standalone

W aplikacji wykorzystano nowoczesne podejście do struktury projektu Angularowego oparte na komponentach typu standalone, dostępnych od wersji Angular 14+. Komponenty te nie wymagają deklarowania w modułach (NgModule), co upraszcza architekturę i zwiększa czytelność projektu. Każdy komponent samodzielnie importuje potrzebne zależności (np. CommonModule, ReactiveFormsModule, MatButtonModule), co zmniejsza ryzyko błędów konfiguracji i poprawia rozdzielność kodu.

Taka architektura sprzyja lepszej organizacji i łatwemu skalowaniu aplikacji, ponieważ każdy komponent stanowi niezależny blok funkcjonalny. Podejście standalone ułatwia także testowanie oraz potencjalne przenoszenie komponentów między projektami.

### 2.3.3. Routing i ochrona tras

W aplikacji frontendowej opartej na Angularze, routing stanowi kluczowy element nawigacji pomiędzy komponentami. W projekcie zastosowano nowoczesny sposób deklarowania tras z wykorzystaniem loadComponent, co umożliwia lazy loading poszczególnych widoków, poprawiając wydajność ładowania aplikacji.

Przykładowa konfiguracja trasy wygląda następująco:

```
export const routes: Routes = [
  {
    path: 'uzytkownicy',
    children: [
      {
        path: 'lekarze',
        loadComponent: () => import('@roles/doctors/pages/doctors-list/doctors-
list.component').then(m => m.DoctorsListComponent),
        canActivate: [(route, state) => inject(ROLE_GUARD)(route, state)],
        data: { roles: [UserRole.NURSE, UserRole.DOCTOR, UserRole.ADMIN] }
      },
      // Inne trasy
    ]
  }
]
```

Listing 15: Przykładowa konfiguracja trasy

W powyższym przykładzie zastosowano mechanizm ochrony tras (canActivate), który sprawdza, czy aktualny użytkownik posiada odpowiednią rolę przed uzyskaniem dostępu do danej trasy. Do tego celu wykorzystano własny InjectionToken o nazwie ROLE\_GUARD.

Definicja strażnika ról (ROLE\_GUARD) wygląda następująco:

```
export const ROLE_GUARD = new InjectionToken<CanActivateFn>('roleGuard');

export const roleGuard: CanActivateFn = (route: ActivatedRouteSnapshot, _state:
RouterStateSnapshot) => {
  const authFacade = inject(AuthFacade);
  const router = inject(Router);

  const allowedRoles = route.data?.['roles'] as UserRole[] | undefined;
```

```

    if (!allowedRoles) {
        return true;
    }
    const userRole = authFacade.getUserRole();

    if (!userRole || !allowedRoles.includes(userRole)) {
        router.navigate(['/no-permissions']);
        return false;
    }
    return true;
};

```

Listing 16: Definicja roleGuard

Dzięki zastosowaniu InjectionToken możliwe było łatwe przypisanie wspólnej logiki strażnika do wielu tras, z jednoczesnym wsparciem dla dependency injection bez konieczności tworzenia klasy serwisu. Rozwiązanie to cechuje się przejrzystością i elastycznością — każda trasa może wskazać wymagane role poprzez wpis w `data.roles`.

W przypadku braku uprawnień użytkownik zostaje przekierowany na specjalnie przygotowaną trasę `/no-permissions`. Zastosowanie takiego mechanizmu ochrony zapewnia kontrolę dostępu w oparciu o rolę użytkownika już na poziomie frontendowym, zanim komponent zostanie załadowany.

#### 2.3.4. Autoryzacja i zarządzanie stanem użytkownika

System autoryzacji użytkownika oparty jest na tokenach JWT (JSON Web Token), które są przechowywane lokalnie w przeglądarce (`localStorage`). Odpowiadają one za identyfikację i uwierzytelnienie użytkownika podczas korzystania z aplikacji. Obsługa logowania, wylogowania, odświeżania tokenów oraz zarządzania sesją została rozdzielona pomiędzy kilka dedykowanych serwisów: `AuthStorageService`, `AuthFacade` oraz `SessionTimerService`.

#### Przechowywanie tokenów JWT w `localStorage`

Dane autoryzacyjne są przechowywane przy pomocy serwisu `AuthStorageService`, który pełni rolę warstwy dostępu do `localStorage`. Obsługuje on zapis i odczyt:

- tokenów (`access_token`, `refresh_token`),
- danych użytkownika (`UserAuthData`),
- roli użytkownika (`UserRole`).

Dzięki temu stan użytkownika może być łatwo przywracany po odświeżeniu strony.

## **Zarządzanie stanem użytkownika – AuthFacade**

Serwis AuthFacade pełni funkcję pośrednika między komponentami aplikacji a warstwą sesji i autoryzacji. Odpowiada za:

- inicjalizację sesji użytkownika po odświeżeniu strony,
- ustawianie i czyszczenie stanu zalogowania (`setAuthState()`, `clearAuthState()`),
- logowanie, wylogowanie oraz rejestrowanie użytkownika,
- odświeżanie tokena w tle, z obsługą kolejkowania zapytań podczas jednoczesnych prób (`refreshTokens()`),
- udostępnianie obserwowalnego strumienia użytkownika (`user$`) oraz stanu inicjalizacji (`initialized$`) i błędu (`errorOccurred$`).

Wszystkie dane sesyjne są synchronizowane z `localStorage`, dzięki czemu aplikacja działa w sposób przewidywalny nawet po odświeżeniu strony.

## **Obsługa czasu życia sesji – SessionTimerService**

W celu automatycznego zarządzania czasem życia sesji, zastosowano dedykowany serwis `SessionTimerService`. Działa on w oparciu o:

- domyślny czas życia tokena (`TokenLifetime`), pobierany z backendu po zalogowaniu,
- timestamp rozpoczęcia sesji (`token_start_timestamp`), który jest zapisywany w `localStorage`,
- timer (`setTimeout`) uruchamiany po zalogowaniu lub przywracany po odświeżeniu strony (`restoreTimer()`).

Po przekroczeniu dopuszczalnego czasu życia sesji (w przypadku tej aplikacji 15 minut), aplikacja automatycznie wylogowuje użytkownika. Serwis oferuje również pomocnicze metody do sprawdzania pozostałego czasu (`getSecondsRemaining()`) oraz jego formatowania (`formatTime()`).

## **Inicjalizacja stanu aplikacji**

W trakcie uruchamiania aplikacji stan autoryzacji jest automatycznie przywracany, o ile w `localStorage` znajdują się dane sesji. Proces ten odbywa się w metodzie `initializeAuthState()` dostępnej w serwisie `AuthFacade`, która jest wywoływana podczas uruchamiania aplikacji Angular przy pomocy mechanizmu `APP_INITIALIZER`.

W ramach inicjalizacji:



1. Sprawdzana jest obecność tokenu `access_token` w `localStorage`.
2. Jeżeli token istnieje, następuje pobranie danych użytkownika z backendu (`getUserInfo()`).
3. Dane sesji są przywracane (`setAuthState()`), a następnie uruchamiany jest `SessionTimerService` z odtworzeniem czasu rozpoczęcia sesji.
4. W przypadku błędu (np. niepoprawny lub wygasły token) następuje automatyczne wylogowanie (`forceLogout()`).

Dzięki temu użytkownik nie musi ponownie się logować po odświeżeniu strony, a aplikacja zachowuje stan sesji w sposób spójny i bezpieczny.

### 2.3.5. Komunikacja z backendem

W aplikacji zastosowano nowoczesne i czytelne podejście do komunikacji z backendem, oparte na kilku warstwach odpowiedzialności. Struktura ta umożliwia łatwą rozbudowę, testowanie i utrzymanie kodu. Komunikacja z backendem opiera się na:

- generycznych serwisach danych (`ApiService`)
- fasadach pośredniczących (`Facade`)
- centralnym repozytorium endpointów (`Endpoints`)
- interceptorach HTTP obsługujących uwierzytelnianie i błędy

#### Serwisy danych (`ApiService`)

Bazą dla wszystkich zapytań HTTP jest generyczna klasa `ApiService<T>`, która implementuje zestaw metod do wykonywania standardowych operacji CRUD (**C**reate, **R**ead, **U**ppdate, **D**ele~~t~~e). Wśród dostępnych metod znajdują się m.in.:

- `getList` – pobieranie listy elementów z uwzględnieniem paginacji
- `getListAll` – pobieranie wszystkich rekordów bez paginacji
- `get` – pobieranie pojedynczego obiektu na podstawie identyfikatora
- `post` – wysyłanie nowych danych do serwera
- `put` i `patch` – aktualizacja danych (całościowa lub częściowa)
- `delete` – usuwanie danych

Serwisy danych dziedziczą po `ApiService` i ustawiają odpowiedni endpoint. Przykład:

```
@Injectable()
export class OfficesDataService extends ApiService<Office> {
  protected url = Endpoints.urls.dictionaries.offices;
}
```

Listing 17: Przykładowy serwis danych

## Fasady (Facade)

Warstwa fasad stanowi pośrednika pomiędzy komponentami a serwisami danych. Fasady odpowiadają za:

- uproszczenie logiki dostępnej dla komponentów
- hermetyzację złożonych operacji na danych
- ewentualne przekształcenia danych (np. mapowanie, agregacja)
- ułatwienie testowania logiki aplikacji

Przykład użycia fasady:

```
@Injectable()
export class OfficesFacade {
  constructor(private officesDataService: OfficesDataService) {}

  getOffices(params: ListParams): Observable<ListResponse<Office>> {
    return this.officesDataService.getList(params);
  }
}
```

Listing 18: Przykładowa fasada

## Centralne zarządzanie endpointami (Endpoints)

Wszystkie adresy URL do backendu są przechowywane w jednej klasie Endpoints, co znacząco poprawia czytelność kodu oraz ułatwia zarządzanie zmianami. Przykład:

```
export class Endpoints {
  static readonly baseUrl = environment.apiUrl;
  static readonly urls = Object.freeze({
    auth: {
      changePassword: '/auth/change-password/', //oraz inne endpointy logowania
    },
    dictionaries: {
      countries: '/dictionaries/countries/', // oraz inne endpointy słowników
    },
    roles: {
      doctors: '/roles/doctors/', // oraz inne endpointy ról użytkowników
    },
    treatment: {
      prescriptions: '/treatment/prescriptions/',
      visits: '/treatment/visits/'
    }
  });
}
```

Listing 19: Klasa Endpoints

## Interceptory HTTP

Interceptory HTTP to specjalne mechanizmy dostępne w Angularze, które pozwalają przechwytywać i modyfikować wszystkie wychodzące żądania HTTP oraz przychodzące odpowiedzi. Umożliwiają one m.in.:

- dodawanie nagłówków do żądań (np. tokenu autoryzacyjnego),
- modyfikowanie treści żądań lub odpowiedzi,
- globalną obsługę błędów HTTP,
- logowanie żądań i odpowiedzi,
- wykonywanie dodatkowych działań przed lub po przesłaniu żądania.

Interceptory są konfigurowane globalnie w aplikacji i działają transparentnie, dzięki czemu logika uwierzytelniania i obsługi błędów nie musi być powtarzana w każdym serwisie lub komponencie.

W aplikacji zdefiniowano dwa interceptory HTTP:

- **AuthInterceptor** – odpowiada za:
  - automatyczne dodawanie nagłówka Authorization z tokenem JWT
  - odświeżanie tokena, gdy jego czas do wygaśnięcia jest krótki
  - ponowne wykonanie żądania z nowym tokenem
- **HttpErrorInterceptor** – odpowiada za:
  - globalną obsługę błędów HTTP (np. 400, 404, 500)
  - wyświetlanie komunikatów o błędach za pomocą `ToastService`
  - automatyczne przekierowanie na stronę błędu w przypadku kodu 404

### 2.3.6. Dyrektywy uprawnień i widoczności elementów

W celu zapewnienia odpowiedniego poziomu kontroli dostępu do poszczególnych elementów interfejsu użytkownika, w aplikacji zaimplementowano dedykowane dyrektywy do zarządzania widocznością oraz stanem aktywności komponentów na podstawie przypisanej roli użytkownika.

Dzięki zastosowaniu dyrektyw `yhtPermission` oraz `yhtPermissionDisabled`, możliwe jest centralne i deklaratywne zarządzanie uprawnieniami bez konieczności implementowania tej logiki bezpośrednio w komponentach. Rozwiązanie to znacząco poprawia czytelność szablonów, a jednocześnie zwiększa bezpieczeństwo aplikacji poprzez ograniczanie akcji dostępnych tylko dla użytkowników z odpowiednimi rolami.

## Dyrektywa `yhtPermissionDisabled`

Dyrektywa `yhtPermissionDisabled` umożliwia blokowanie interakcji z elementem (np. przyciskiem), jeżeli użytkownik nie posiada odpowiednich uprawnień. Elementy takie są wizualnie oznaczane jako nieaktywne (np. przez nałożenie półprzezroczystego stylu lub nadanie klasy „disabled”), a także mogą wyświetlać dodatkową informację o braku uprawnień za pomocą podpowiedzi (tooltipa).

Dyrektywa działa zarówno na standardowych przyciskach, jak i na elementach menu (`mat-mdc-menu-item`). W przypadku braku dostępu, element zostaje zablokowany poprzez manipulację jego atrybutami DOM (np. `disabled`, `tabindex`), a jego działanie zostaje całkowicie wyłączone.

Zastosowanie tej dyrektywy pozwala na:

- dynamiczne blokowanie elementów w zależności od roli użytkownika,
- ujednolicenie sposobu prezentacji niedostępnych funkcji w UI,
- uniknięcie powielania logiki warunkowej w komponentach.

Poniżej pokazano przykład zastosowania dyrektywy:

```
<button
  [yhtPermissionDisabled]="[userRole.NURSE, userRole.ADMIN]"
  color="accent"
  mat-raised-button
  (click)="openFormDialog()"
>
  {{ strings.actions.create.label }}
</button>
```

Listing 20: Przykład zastosowania dyrektywy atrybutowej

## Dyrektywa `yhtPermission`

Druga dyrektywa – `yhtPermission` – pełni rolę strukturalną, umożliwiając warunkowe renderowanie fragmentów szablonu tylko wtedy, gdy użytkownik posiada jedną z wymaganych ról. Działa analogicznie do natywnej dyrektywy Angulara `*ngIf`, lecz jej warunek oparty jest na bieżącej roli zalogowanego użytkownika.

W przypadku braku wymaganych uprawnień, zawartość objęta dyrektywą nie jest w ogóle renderowana w DOM, co dodatkowo zwiększa bezpieczeństwo aplikacji oraz zmniejsza ryzyko przypadkowego ujawnienia ukrytych funkcji.

Zalety tego podejścia to m.in.:

- całkowite ukrycie niedozwolonych elementów,
- proste i przejrzyste szablony,
- pełna integracja z architekturą uprawnień aplikacji.

Poniżej pokazano przykład zastosowania dyrektywy:

```
@for (menuItem of userMenuConfig; track menuItem) {  
  <button *yhtPermission="menuItem.roles" mat-menu-item (click)="menuItem.action()">  
    {{ menuItem.label }}  
  </button>  
}
```

Listing 21: Przykład zastosowania dyrektywy strukturalnej

## Integracja z systemem ról

Obie dyrektywy współpracują z AuthFacade, który udostępnia aktualnego użytkownika w postaci obserwowalnego strumienia (user\$). Dzięki temu dyrektywy automatycznie reagują na zmiany stanu logowania lub modyfikacje ról użytkownika. W przypadku odświeżenia strony, stan użytkownika jest przywracany z localStorage, co zapewnia ciągłość działania dyrektyw bez konieczności ręcznej inicjalizacji.

### 2.3.7. Komponenty tabel i obsługa danych

W aplikacji zaimplementowano zestaw komponentów opartych o Angular Material oraz podejście reaktywne (Observable), które wspierają zaawansowaną obsługę tabeli danych. Podczas ładowania danych komponent wyświetla animację postępu (spinner), co poprawia czytelność interfejsu użytkownika. Komponenty te bazują na wspólnej klasie abstrakcyjnej TableComponentBase<T>, która udostępnia funkcjonalności takie jak:

- sortowanie kolumn (MatSort),
- paginacja wyników (MatPaginator),
- dynamiczne filtrowanie danych (formularze nad kolumnami),
- możliwość przeszukiwania listy,
- dynamiczne pobieranie danych z backendu (wspierające Observable),
- elastyczna konfiguracja kolumn z możliwością przypisania typów i filtrów.

## Integracja z Angular Material

Tabela oparta jest na komponentach Angular Material, w szczególności:

- mat-paginator – komponent odpowiedzialny za obsługę paginacji (zmiana strony, liczby elementów na stronie),
- mat-sort-header – dyrektywa umożliwiająca sortowanie kolumn po kliknięciu nagłówka.

Wartości aktualnej strony, limitu oraz kolejności sortowania są automatycznie synchronizowane z obiektem `ListParams`, który odpowiada za generowanie zapytań do backendu.

Poniższy fragment przedstawia strukturę klasy `ListParams`, która przechowuje dane dotyczące filtrowania, sortowania i paginacji:

```
export class ListParams {
  filters: CustomObject;
  ordering: OrderingType[];
  pagination: ListPagination;

  constructor(options?: { filters?: CustomObject; ordering?: OrderingType[];
    pagination?: ListPagination }) {
    this.filters = options?.filters || {};
    this.ordering = options?.ordering || [environmentBase.list.ordering as
    OrderingType];
    this.pagination = new ListPagination(options ? options.pagination : null);
  }

  getParams(): CustomObject {
    return Object.assign({}, this.filters, this.getOrdering(), this.getPagination() ||
    {});
  }

  private getPagination(): { limit: number; offset: number } {
    return this.pagination ? { limit: this.pagination.limit, offset:
    this.pagination.offset } : undefined;
  }

  setOrdering(sortEvent: Partial<Sort>): void {
    this.ordering = [{ column: sortEvent.active, direction: sortEvent.direction }];
  }

  private getOrdering(): { ordering?: string } {
    const orderingStr = this.ordering.map(orderingItem =>
    this.transformOrderingToString(orderingItem)).join(',');
    return orderingStr ? { ordering: orderingStr } : {};
  }

  private transformOrderingToString(ordering: OrderingType): string {
    return ordering && ordering.column ? `${ordering.direction} == 'desc' ? '-' :
    ''}${ordering.column}` : '';
  }
}
```

Listing 22: Klasa `ListParams` - budowanie zapytania do backendu

## Obsługa danych

Dane do tabeli są ładowane dynamicznie przy użyciu metody `getData()`, która korzysta z `Observable`. W zależności od komponentu, dane mogą być pobierane z backendu przez fasadę lub bezpośrednio z serwisu danych (`ApiService`). Komponent wykrywa zmiany w parametrach filtrowania, sortowania i paginacji, a następnie odświeża dane.

W przypadku błędów podczas pobierania danych (np. błąd sieci, nieautoryzowany dostęp), komponent rejestruje błąd `HttpErrorResponse` i wyświetla odpowiedni komunikat. Obsługa błędów jest ujednolicona i realizowana za pomocą dedykowanej metody `handleError()`.

### Filtrowanie danych

Filtrowanie realizowane jest przez komponent `TableFiltersComponent`, który generuje dynamiczny formularz filtrów na podstawie definicji kolumn. Obsługuje różne typy pól (tekstowe, logiczne, zakres dat, listy wyboru) oraz dodatkowe filtry „rozszerzone”. Dane z formularza są emitowane do komponentu tabeli, który odpowiednio aktualizuje parametry zapytania.

### Generowanie kolumn – `ListHelper`

W celu ułatwienia konfigurowania kolumn tabeli wykorzystano pomocniczą klasę `ListHelper`, która na podstawie typu danych generuje odpowiednie ustawienia kolumny – w tym typ filtra, tłumaczenia, styl wyrównania i opcje formatowania. Poniżej przedstawiono przykład generowania kolumny za pomocą `ListHelper`:

```
ListHelper.getColumnDef('visit_status', TableColumnTypes.SELECT, this.visitStrings, {
  data: {
    options: [VisitStatus.COMPLETED, VisitStatus.IN_PROGRESS,
VisitStatus.SCHEDULED].map(status => ({
      label: this.visitStrings.visit_status.options[status],
      value: status
    }))
  } as TableSelectCellData<{ label: string; value: string }>,
  styles: {
    cssClass: 'text-nowrap',
    isNarrow: true
  }
})
```

Listing 23: Przykład generowania kolumny typu SELECT

Takie podejście pozwala na szybkie przygotowanie konfiguracji tabel w sposób zwięzły i spójny.

### Rozszerzenia – rozwijane wiersze

Dla tabel wymagających wyświetlania dodatkowych informacji o rekordzie, zastosowano klasę bazową `TableRowExpanderComponentBase<T>`, która dodaje kolumnę „rozszerzającą” (expander) oraz obsługuje przełączanie widoczności szczegółów. Każdy wiersz posiada flagę `isExpanded`, która decyduje o jego rozwinięciu.

### 2.3.8. Formularze i walidacja

W aplikacji zaimplementowano mechanizm obsługi formularzy oparty o podejście reaktywne (ReactiveFormsModule) dostępne w Angularze. Formularze są dynamicznie konfigurowane na podstawie mapy pól (FormFields), co umożliwia wielokrotne wykorzystanie wspólnych komponentów z zachowaniem pełnej kontroli nad walidacją i prezentacją formularza.

#### Formularze reaktywne

Reaktywne formularze pozwalają na definiowanie struktur UntypedFormGroup, UntypedFormControl i UntypedFormArray w logice komponentu. Każde pole formularza opisane jest przy pomocy struktury FormField, która zawiera m.in. typ pola, etykietę, wymagania walidacyjne, tryb tylko do odczytu (readonly) oraz dane do pól wyboru.

Do dynamicznego tworzenia formularzy służy metoda prepareForm() z klasy FormComponentBase, która przekształca mapę pól w strukturę UntypedFormGroup z przypisanymi walidatorami:

```
protected prepareForm(fields: FormFields = this.fields, returnForm = false):
UntypedFormGroup | void {
  const formControls: { [key: string]: UntypedFormControl } = {};
  fields.forEach(field => {
    const validators = field.validators ? [...field.validators] : [];

    if (field.required) {
      validators.push(Validators.required);
    }

    if (!field.notEditable) {
      formControls[field.name] = new UntypedFormControl({ value: field.value,
disabled: field.disabled }, validators);
    }
  });

  const form = new UntypedFormGroup(formControls);
  if (returnForm) {
    return form;
  } else {
    this.form = form;
  }
}
```

Listing 24: Tworzenie dynamicznego formularza w FormComponentBase

#### Obsługa walidacji i komunikatów błędów

W celu ujednolicenia obsługi błędów formularzy stworzono klasę bazową FieldComponentBase, z której dziedziczą wszystkie komponenty renderujące konkretne pola. Klasa ta subskrybuje zmiany statusu kontrolki (statusChanges) i przypisuje komunikaty błędów do pola field.errors, wykorzystywanego w szablonie.



Poniżej przedstawiono fragment kodu odpowiadający za ustawianie komunikatów błędów.

```
protected initListenerOnStatusChanged(): void {
    if (this.control && this.field) {
        this.setFieldErrors();
        this.controlStatusSubscription = this.control.statusChanges.subscribe(() => {
            this.setFieldErrors();
        });
    }
}

private setFieldErrors(): void {
    this.field.errors = this.getErrorMessage(this.field.name);
    this.cdr.markForCheck();
}
```

Listing 25: Przykład ustawiania komunikatu błędu w FieldComponentBase

Metoda `getErrorMessageFromControl` tłumaczy typy błędów (np. `required`, `email`, `min`) na tekstowe komunikaty dostosowane do użytkownika. Dzięki temu użytkownik otrzymuje jasną informację, które pole zostało błędnie wypełnione i dlaczego.

### Komponent bazowy dla pól formularza

Każde pole formularza renderowane jest przy użyciu dedykowanego komponentu, dziedziczącego po klasie bazowej `FieldComponentBase`. Komponent ten zapewnia:

- przypisywanie placeholderów i ID pól,
- dynamiczne wyświetlanie błędów,
- śledzenie zmian walidacyjnych,
- obsługę formularzy tablicowych (`UntypedFormArray`) poprzez `arrayName` i `index`.

### Dynamiczne renderowanie pól – `FormFieldSwitcherComponent`

W zależności od typu pola (`FormFieldTypes`) komponent `FormFieldSwitcherComponent` wybiera odpowiedni komponent renderujący: pole tekstowe, autouzupełnianie, wybór z listy, pola daty, pola logiczne, zakresy liczbowe itd. Przykład użycia znajduje się w komponencie `FormListComponent`, który renderuje tabelę pól na podstawie mapy definicji.

### Komponent bazowy dla formularzy

Większość formularzy w aplikacji dziedziczy po abstrakcyjnej klasie `FormComponentBase`, która centralizuje takie operacje jak:

- budowanie formularzy (`prepareForm`),
- przypisywanie danych (`setFieldsValues`),
- ustawianie trybu tylko do odczytu (`setFieldsReadonly`),

- obsługa błędów backendu (handleError).

W przypadku błędu zapisu (np. błędna walidacja po stronie serwera), komunikaty z pola `non_field_errors` są rozprowadzane do odpowiednich kontrolek.

```
protected handleError(error: HttpResponse): void {
  this.saving = false;
  handleRequestErrors.call(this, error, this.form);
  this.cdr.markForCheck();
}
```

Listing 26: Obsługa błędów z backendu

### Formularze z wieloma wierszami – `FormListComponent`

Dla bardziej złożonych formularzy wykorzystuje się komponent `FormListComponent`, który obsługuje dynamiczną listę elementów (np. dawkovanie leków na recepcie). Komponent ten umożliwia:

- dodawanie i usuwanie wierszy formularza,
- dynamiczne renderowanie pól w tabeli,
- walidację na poziomie listy (np. wymagane minimum jednego wpisu).

### Ochrona formularzy – `FormCanDeactivate`

Aby zapobiec przypadkowemu opuszczeniu formularza z niezapisanymi zmianami, w aplikacji zastosowano mechanizm `FormCanDeactivate`. Klasa ta definiuje metodę `canDeactivate()`, która zwraca `false`, jeśli formularz zawiera niezapisane zmiany. Mechanizm ten obsługuje również zamykanie modali (`DialogFormCanDeactivate`) i komunikaty przy próbie zamknięcia przeglądarki.

```
@HostListener('window:beforeunload', ['$event'])
onBeforeUnload(event: BeforeUnloadEvent): BeforeUnloadEvent {
  if (!this.canDeactivate()) {
    event.returnValue = 'Unsaved';
  }
  return event;
}
```

Listing 27: Przykład ochrony formularza przed zamknięciem

### 2.3.9. Powiadomienia i komunikaty użytkownika (`ToastService`)

W aplikacji zaimplementowano system powiadomień oparty na bibliotece `ngx-toastr`, który umożliwia prezentowanie użytkownikowi krótkich komunikatów w formie tzw. *toastów*. Komunikaty te służą do informowania o powodzeniu operacji, wystąpieniu błędów lub innych istotnych zdarzeniach w aplikacji.

Obsługa toastów została ujednolicona za pomocą dedykowanego serwisu `ToastService`. Serwis ten pełni rolę warstwy pośredniej między logiką aplikacji a biblioteką `ngx-toastr` [22] i udostępnia zunifikowane metody do wyświetlania komunikatów typu: sukces, błąd oraz ostrzeżenie. Zastosowanie takiego podejścia umożliwia centralne zarządzanie powiadomieniami oraz ewentualne ich modyfikowanie bez konieczności zmiany kodu w wielu komponentach.

W zależności od sytuacji, toast może zawierać komunikat zdefiniowany bezpośrednio lub odwołanie do tłumaczeń aplikacji. Serwis wykorzystywany jest m.in. w komponentach formularzy, przy operacjach zapisu i usuwania danych, a także w interceptorach obsługujących błędy HTTP.

Dzięki zastosowaniu `ToastService` uzyskano:

- spójny sposób wyświetlania komunikatów w całej aplikacji,
- lepszą separację odpowiedzialności (komponenty nie korzystają bezpośrednio z `ngx-toastr`),
- możliwość łatwego testowania i ewentualnej podmiany mechanizmu wyświetlania powiadomień w przyszłości,
- zwiększenie komfortu użytkownika dzięki natychmiastowej informacji zwrotnej po wykonaniu akcji.

Rozwiązanie to wpisuje się w nowoczesne podejście do projektowania frontendowych aplikacji webowych i stanowi jeden z elementów poprawiających jakość interfejsu użytkownika.

## 2.4. Docker

W celu uproszczenia procesu uruchamiania oraz zwiększenia przenośności systemu, w projekcie zastosowano Dockera – platformę do konteneryzacji aplikacji. Dzięki temu każda część systemu (baza danych, backend, frontend) działa w izolowanym środowisku, co eliminuje problemy związane z różnicami w konfiguracji lokalnej i produkcyjnej.

Konfiguracja kontenerów została opisana w pliku `docker-compose.yml`, który definiuje trzy główne usługi:

- **database** – kontener z bazą danych PostgreSQL 16, uruchamiany z odpowiednimi zmiennymi środowiskowymi i skryptami inicjalizacyjnymi,

- **backend** – aplikacja Django uruchamiana w kontenerze na bazie obrazu python:3.12, z automatycznym wykrywaniem migracji, tworzeniem konta administratora oraz ładowaniem danych słownikowych,
- **frontend** – aplikacja Angular uruchamiana w kontenerze z wykorzystaniem node:23 oraz Angular CLI.

Dzięki zastosowaniu Dockera:

- cała aplikacja może być uruchomiona jednym poleceniem `docker-compose up`,
- środowisko deweloperskie i testowe jest spójne i łatwe do odtworzenia na dowolnym komputerze lub w chmurze,
- uruchamianie testów backendu odbywa się również w środowisku kontenerowym (z osobną instancją bazy danych),
- poszczególne komponenty są odseparowane, co ułatwia debugowanie i rozwój systemu.

Na backendzie zastosowano również dedykowany skrypt `entrypoint.sh`, który przed uruchomieniem serwera wykonuje:

- oczekiwanie na gotowość bazy danych,
- tworzenie i stosowanie migracji,
- tworzenie sekwencji bazodanowych,
- ładowanie danych słownikowych,
- tworzenie konta administratora (jeśli nie istnieje).

Dzięki temu uruchomienie projektu na nowym środowisku jest w pełni zautomatyzowane i nie wymaga ręcznej konfiguracji.

## 2.5. CI / CD (Continuous Integration / Continuous Deployment)

W projekcie został zaimplementowany proces CI/CD (ang. *Continuous Integration / Continuous Deployment*) przy użyciu GitHub Actions. Dzięki temu każda zmiana wprowadzana do głównej gałęzi (**master**) repozytorium jest automatycznie:

1. Pobierana i analizowana – kod źródłowy jest klonowany do środowiska CI po każdym pushu lub pull requeście.
2. Weryfikowana pod kątem stylu i jakości kodu:
  - Backend (Django) jest sprawdzany narzędziem Ruff [22] pod kątem zgodności ze standardami Pythona i błędów statycznych.

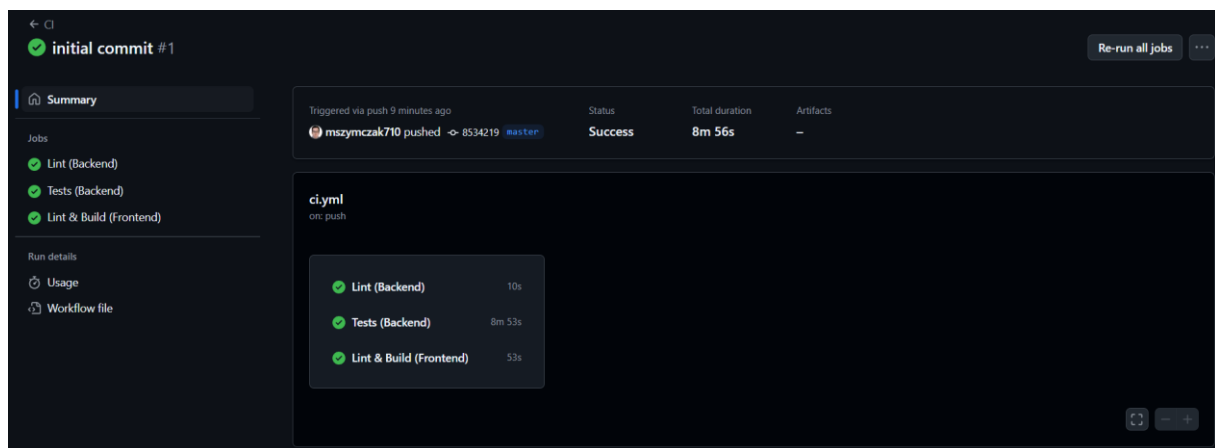
- Frontend (Angular) jest sprawdzany z użyciem Prettier [23] i lintera Angulara (eslint) [24] w celu zachowania spójności kodu i dobrych praktyk.

### 3. Testowana automatycznie:

- Testy jednostkowe i integracyjne backendu są uruchamiane z użyciem pytest oraz coverage do mierzenia pokrycia kodu testami. W ramach środowiska testowego uruchamiany jest również kontener z bazą danych PostgreSQL.
- Budowana aplikacja frontendowa – jeżeli formatowanie i linting zakończą się powodzeniem, projekt Angulara jest budowany w trybie produkcyjnym (`--configuration production`).

Takie podejście pozwala:

- szybciej wykrywać błędy,
- utrzymywać wysoką jakość kodu,
- przyspieszyć cykl wdrażania,
- zapewnić spójność i niezawodność w procesie rozwoju aplikacji.



Rysunek 3: Widok zakończonego sukcesem workflow GitHub Actions

### 3. Przegląd funkcjonalności aplikacji

#### 3.1. Ekran powitalny aplikacji (dla niezalogowanego użytkownika)

Po wejściu na stronę główną aplikacji niezalogowany użytkownik zostaje przywitany przejrzystym ekranem powitalnym. Celem tego widoku jest szybkie przekazanie podstawowych informacji o funkcjonalnościach aplikacji oraz umożliwienie przejścia do rejestracji lub logowania.

Centralnym elementem interfejsu jest logo aplikacji oraz komunikat powitalny:

*Witaj w aplikacji Your Health Time*

*Jeśli nie masz jeszcze konta, naciśnij przycisk Zarejestruj się.*

*Jeśli już posiadasz konto – Zaloguj się, aby kontynuować.*

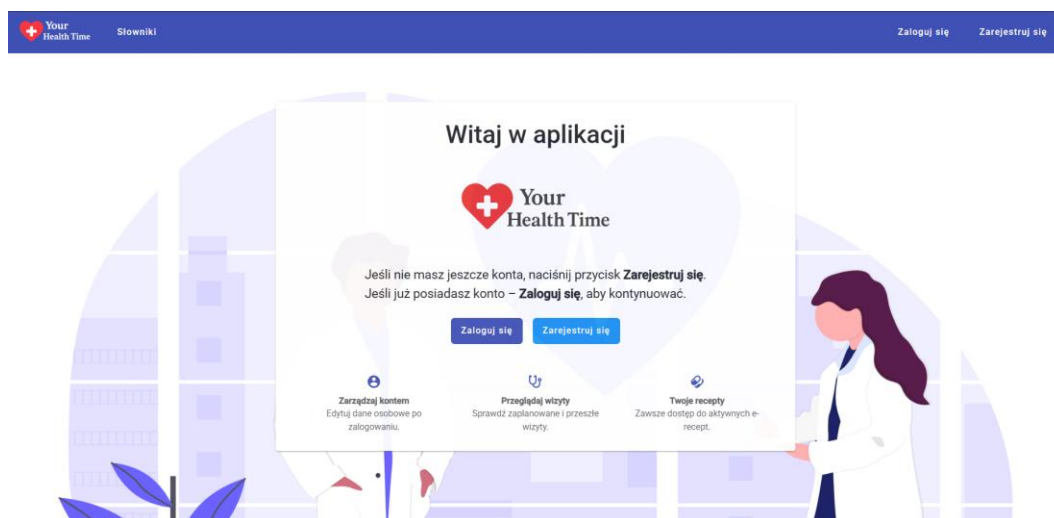
Użytkownik ma do wyboru dwa główne przyciski:

- **Zaloguj się** – przekierowuje do formularza logowania,
- **Zarejestruj się** – rozpoczyna proces rejestracji.

Dodatkowo poniżej znajdują się krótkie opisy głównych funkcjonalności aplikacji, które staną się dostępne po zalogowaniu:

- **Zarządzaj kontem** – edytuj dane osobowe,
- **Przeglądaj wizyty** – sprawdzaj planowane i przeszłe wizyty,
- **Twoje recepty** – zawsze miej dostęp do aktywnych e-recept.

Dzięki tej konstrukcji aplikacja jasno komunikuje cel i korzyści, a także zachęca do rejestracji, jeśli użytkownik odwiedza ją po raz pierwszy.



Rysunek 4: Dashboard niezalogowanego użytkownika

## 3.2. Moduł uwierzytelniania i zarządzania kontem

### 3.2.1. Rejestracja użytkownika

Proces rejestracji stanowi pierwszy etap autoryzacji użytkownika w systemie. Został podzielony na trzy kroki, co zapewnia przejrzystość i uporządkowanie wprowadzanych danych:

#### Dane podstawowe

W pierwszym kroku formularza użytkownik wprowadza podstawowe informacje identyfikacyjne i dane logowania. Wymagane pola to:

- **Imię**
- **Nazwisko**
- **Adres e-mail** – służy zarówno do logowania, jak i do przesyłania wiadomości aktywacyjnych.
- **Hasło** – użytkownik musi dwukrotnie wpisać hasło w celu weryfikacji poprawności.
- **PESEL** – wykorzystywany do identyfikacji użytkownika w systemie, np. w kontekście danych medycznych.
- **Numer telefonu** – umożliwi późniejszy kontakt i może być wykorzystywany np. do przypomnień o wizytach.

Wszystkie pola w tej sekcji są obowiązkowe. Formularz waliduje poprawność danych (np. długość hasła, zgodność PESEL-u z formatem) przed przejściem do kolejnego kroku.

Rejestracja

1 Dane podstawowe 2 Adres 3 ReCAPTCHA

Imię \*  
Wpisz imię

Nazwisko \*  
Wpisz nazwisko

Adres e-mail \*  
Wpisz adres e-mail

Hasło \*  
Wpisz hasło

Powtórz hasło \*  
Wpisz ponownie hasło

PESEL \*  
Wpisz PESEL

Numer telefonu \*  
Wpisz numer telefonu

Anuluj Dalej

Rysunek 5: Formularz rejestracji - dane podstawowe

#### Adres

W drugim kroku użytkownik podaje dane adresowe:

- **Ulica**
- **Numer domu**
- **Numer lokalu** (pole opcjonalne)
- **Kod pocztowy**
- **Miasto**
- **Kraj** – możliwe jest użycie pola z autouzupełnianiem.

Również tutaj wymagane jest wypełnienie wszystkich pól (z wyjątkiem numeru lokalu), a formularz weryfikuje ich poprawność. Podział na sekcje umożliwia intuicyjne uzupełnienie danych kontaktowych.

Rysunek 6: Formularz rejestracji - adres

## Weryfikacja ReCAPTCHA

Ostatnim krokiem formularza jest zabezpieczenie za pomocą mechanizmu Google ReCAPTCHA. Użytkownik musi zaznaczyć opcję „Nie jestem robotem”. Jest to obowiązkowy etap, chroniący system przed automatycznym zakładaniem kont.

Rysunek 7: Formularz rejestracji - weryfikacja ReCAPTCHA

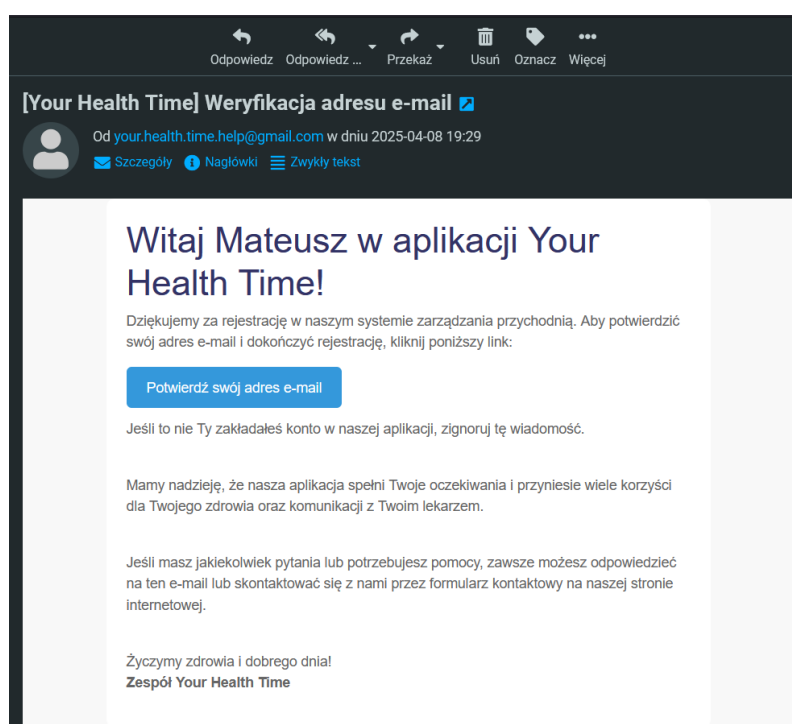


Po przesłaniu formularza rejestracyjnego system generuje wiadomość e-mail weryfikacyjną, która zostaje wysłana na podany przez użytkownika adres e-mail. Wiadomość ta zawiera personalizowaną treść powitalną oraz przycisk prowadzący do linku aktywacyjnego.

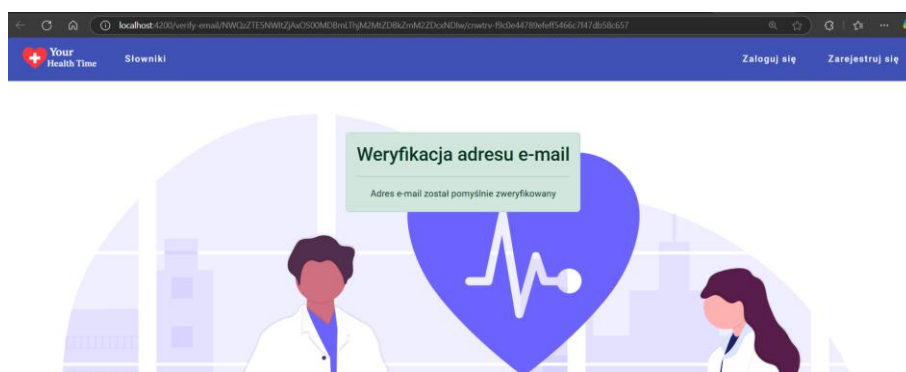
Wiadomość ma za zadanie:

- potwierdzić tożsamość użytkownika,
- upewnić się, że podany adres e-mail należy do osoby rejestrującej się,
- aktywować konto i umożliwić dostęp do aplikacji.

Kliknięcie w przycisk „**Potwierdź swój adres e-mail**” przekierowuje użytkownika na stronę aplikacji, gdzie wyświetlany jest komunikat informujący o pomyślnej weryfikacji adresu.



Rysunek 8: E-mail weryfikacyjny



Rysunek 9: Przekierowanie z linku aktywacyjnego

### 3.2.2. Logowanie

Po aktywacji konta użytkownik może zalogować się do systemu za pomocą formularza logowania. Formularz zawiera dwa podstawowe pola:

- **Adres e-mail** – wymagany do identyfikacji konta,
- **Hasło** – wprowadzone podczas rejestracji.

Po przesłaniu formularza system:

- weryfikuje poprawność danych logowania,
- sprawdza, czy konto zostało wcześniej aktywowane za pomocą linku weryfikacyjnego,
- generuje tokeny JWT (access token i refresh token) w przypadku poprawnej autoryzacji,
- zapisuje dane użytkownika i tokeny w pamięci przeglądarki (localStorage).

W przypadku błędnych danych logowania lub nieaktywnego konta system wyświetla stosowny komunikat błędu. Po poprawnym zalogowaniu użytkownik zostaje przekierowany do głównego widoku aplikacji, a jego uprawnienia (rola) determinują dostęp do poszczególnych modułów systemu.

#### Logowanie

Adres e-mail \*

Wpisz adres e-mail



Hasło \*

Wpisz hasło



[Nie pamiętasz hasła?](#)

Anuluj

Zaloguj

Rysunek 10: Formularz logowania

### 3.2.3. Resetowanie hasła

Jeśli użytkownik zapomni hasła, może skorzystać z funkcji resetowania hasła, dostępnej z poziomu formularza logowania.

Na ekranie logowania znajduje się przycisk „**Nie pamiętasz hasła?**”, który otwiera formularz resetowania hasła. Formularz zawiera dwa obowiązkowe elementy:

- **Pole na adres e-mail** – użytkownik podaje adres przypisany do swojego konta,

- **Google reCAPTCHA** – zabezpieczenie przed nadużyciami ze strony botów.

Po poprawnym wypełnieniu formularza i jego zatwierdzeniu, system:

- sprawdza, czy podany adres istnieje w bazie danych,
- generuje unikalny token zabezpieczający,
- wysyła wiadomość e-mail z linkiem umożliwiającym zmianę hasła.

Link do resetowania hasła jest ważny przez **1 godzinę** od momentu jego wygenerowania. Po tym czasie staje się nieaktywny i użytkownik musi ponownie rozpocząć procedurę resetu.

### Resetowanie hasła

Adres e-mail \*

Wpisz adres e-mail



Nie jestem robotem



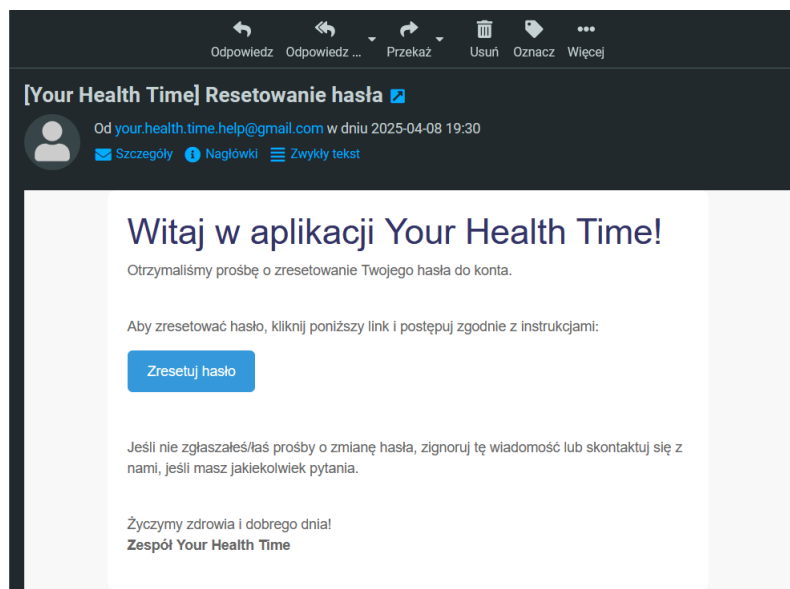
reCAPTCHA  
Prywatność - Warunki

To pole jest wymagane.

Anuluj

Resetuj hasło

Rysunek 11: Formularz resetowania hasła



Rysunek 12: Wiadomość e-mail z linkiem resetowania hasła

Kliknięcie w link przekierowuje użytkownika do osobnego formularza ustawiania nowego hasła, w którym należy dwukrotnie wprowadzić nowe hasło.

Po jego zatwierdzeniu użytkownik może zalogować się z użyciem nowego hasła.

Rysunek 13: Formularz ustawiania nowego hasła

W przypadku nieprawidłowego lub wygasłego tokena aplikacja informuje użytkownika o błędzie i konieczności ponownego rozpoczęcia procesu.

#### 3.2.4. Zmiana hasła i wylogowywanie użytkownika

Po zalogowaniu do systemu użytkownik ma możliwość zmiany swojego hasła oraz wylogowania się z aplikacji. Obie te funkcjonalności są dostępne z poziomu interfejsu użytkownika i mają na celu zapewnienie większej kontroli nad bezpieczeństwem konta.

##### Zmiana hasła

Zmiana hasła możliwa jest wyłącznie po zalogowaniu do systemu. W tym celu użytkownik przechodzi do formularza zmiany hasła, w którym musi podać:

- **obecne hasło** – w celu potwierdzenia tożsamości,
- **nowe hasło** – które ma zostać ustawione,
- **powtórzenie nowego hasła** – w celu wyeliminowania błędów przy wpisywaniu.

Po zatwierdzeniu formularza aplikacja:

- weryfikuje poprawność obecnego hasła,
- sprawdza zgodność nowego hasła z wymaganiami (np. długość, złożoność),
- aktualizuje hasło w bazie danych.

##### Wylogowywanie

System umożliwia użytkownikowi ręczne wylogowanie się w dowolnym momencie poprzez interfejs aplikacji. Proces ten:

- usuwa tokeny JWT zapisane w przeglądarce z `localStorage`,
- kończy aktywną sesję użytkownika,
- przekierowuje użytkownika do strony głównej.

Wylogowanie jest niezbędnym elementem zachowania bezpieczeństwa w aplikacjach opartych na tokenach, szczególnie w środowiskach współdzielonych lub publicznych.

### 3.3. Uprawnienia użytkowników w aplikacji

Aplikacja Your Health Time wykorzystuje mechanizm ról użytkowników, aby odpowiednio ograniczyć dostęp do poszczególnych funkcjonalności. Dzięki temu zapewnione jest bezpieczeństwo danych oraz logiczny podział obowiązków pomiędzy użytkowników.

W systemie wyróżniono cztery podstawowe role: **Pacjent**, **Pielęgniarka**, **Lekarz** oraz **Administrator**. Poniżej przedstawiono szczegółowy zakres uprawnień każdej z ról.

#### 3.3.1. Pacjent

##### Widok po zalogowaniu:

Pacjent po zalogowaniu zostaje przekierowany do formularza zawierającego jego dane osobowe. W formularzu tym pacjent może:

- Edytować swoje dane adresowe (ulica, numer domu, miasto, kraj itd.),
- Zmienić numer telefonu kontaktowego.

The screenshot shows the 'Your Health Time' application interface. At the top, there is a blue header bar with a logo on the left and navigation links 'Wizyty', 'Recepty', and 'Słownik' in the center. On the right side of the header, there is a user profile icon and a session timer showing 'Czas trwania sesji: 14:45'. Below the header, the main content area features a large, light-colored card titled 'Dane pacjenta'. This card contains two sections: 'Dane osobowe' and 'Adres'. The 'Dane osobowe' section includes fields for 'Imię' (Mateusz), 'Nazwisko' (Szymczak), 'Adres e-mail' (mszymczak710@mat.umk.pl), 'PESEL' (01210307910), 'Data urodzenia' (03.01.2001), and 'Płeć' (Mężczyzna). The 'Adres' section includes fields for 'Ulica' (Targowa), 'Numer domu' (16A), 'Numer lokalu' (37), 'Kod pocztowy' (87-100), 'Miasto' (Toruń), and 'Kraj' (Polska). At the bottom left of the card, there is a blue button labeled 'Edytuj'. The background of the interface features a stylized illustration of a city skyline and a person in a white lab coat.

Rysunek 14: Dasboard pacjenta

##### Dostępne funkcje:

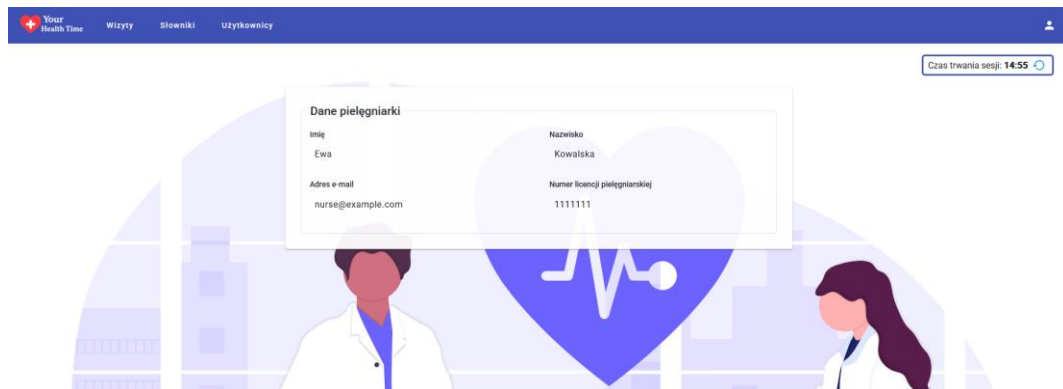
- Przegląd własnych wizyt,

- Podgląd historii e-recept,
- Dostęp do listy krajów (np. do aktualizacji danych adresowych).

### 3.3.2. Pielęgniarka

#### Widok po zalogowaniu:

Pielęgniarka widzi formularz ze swoimi danymi osobowymi, jednak wszystkie pola są wyłącznie do odczytu – nie ma możliwości ich edycji.



Rysunek 15: Dashboard pielęgniarki

#### Dostępne funkcje:

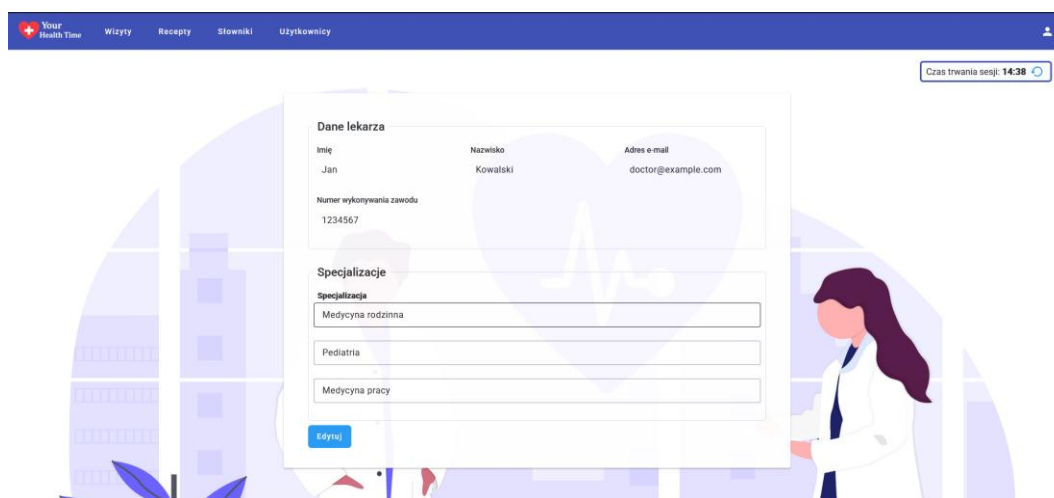
- Przeglądanie, dodawanie, edytowanie oraz usuwanie wizyt,
- Dostęp do listy pacjentów, lekarzy oraz innych pielęgniarek,
- Przegląd słowników: lista krajów, chorób i gabinetów lekarskich.

### 3.3.3. Lekarz

#### Widok po zalogowaniu:

Lekarz zostaje przekierowany do formularza ze swoimi danymi osobowymi. W tym formularzu może edytować listę swoich specjalizacji.

Na następnej stronie przedstawiono stronę główną lekarza.



Rysunek 16: Dashboard lekarza

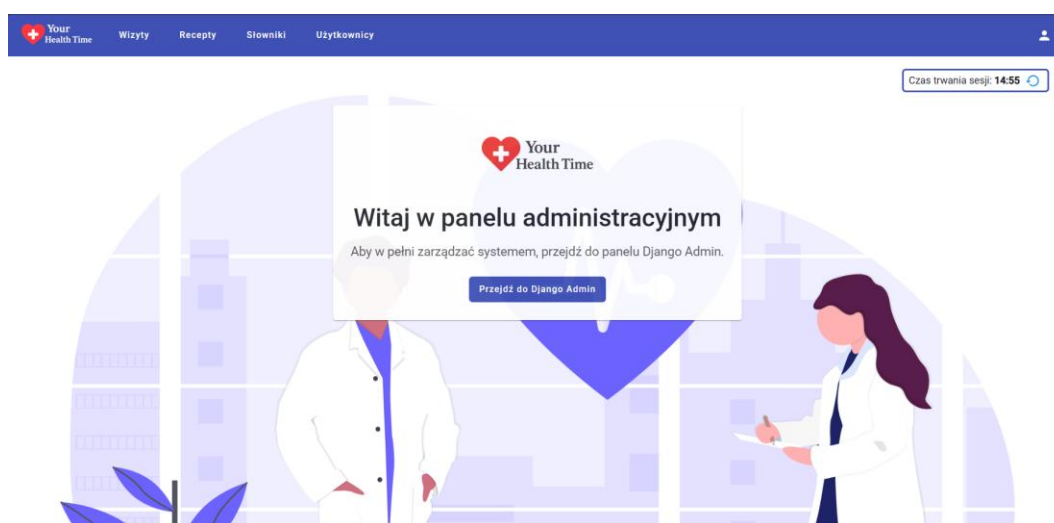
### Dostępne funkcje:

- Przegląd przypisanych do siebie wizyt,
- Wystawianie e-recept dla pacjentów,
- Dostęp do danych słownikowych (kraje, choroby, gabinety, leki i specjalizacje).

### 3.3.4. Administrator

#### Widok po zalogowaniu

Administrator po zalogowaniu widzi ekran powitalny zawierający przycisk, który przekierowuje bezpośrednio do **panelu administracyjnego Django (Django Admin)**. Panel ten stanowi główne narzędzie do zarządzania systemem i dostępny jest wyłącznie dla użytkowników z przypisaną rolą administratora.



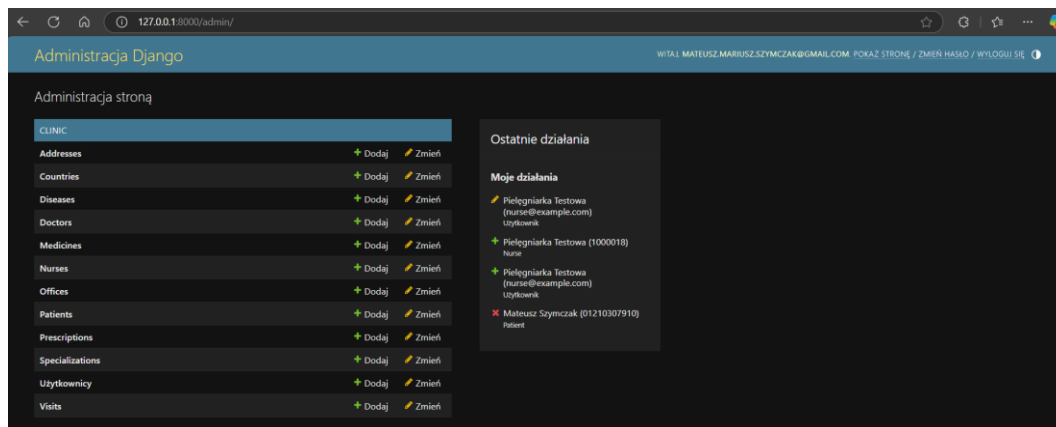
Rysunek 17: Dashboard admina

### Dostępne funkcje:

- Zarządzanie kontami użytkowników (dodawanie, edytowanie, nadawanie ról),
- Pełen dostęp do danych:
  - wizyt,
  - recept,
  - słowników (kraje, gabinety, specjalizacje itd.),
- Możliwość usuwania oraz przeglądania wszystkich danych w systemie.

Wszystkie powyższe funkcjonalności są dostarczane przez **standardowy panel administracyjny Django**, który umożliwia intuicyjne zarządzanie bazą danych z poziomu przeglądarki. Panel ten oferuje mechanizmy filtrowania, wyszukiwania, sortowania oraz formularze do edycji obiektów w systemie, co znacznie ułatwia codzienną administrację aplikacją.

Dodatkowo, w panelu administracyjnym dostępna jest sekcja pokazująca **ostatnie działania administratora** – np. ostatnio edytowane obiekty. Ułatwia to bieżący nadzór nad zmianami wprowadzanymi w systemie oraz pozwala szybko powrócić do wcześniej przeglądanych danych.



Rysunek 18: Panel administracyjny Django

### 3.4. Dostęp do słowników

W aplikacji **Your Health Time** zastosowano zestaw słowników (tzw. danych referencyjnych), które służą do zapewnienia spójności oraz ułatwienia wprowadzania danych w różnych częściach systemu. Słowniki te wykorzystywane są m.in. podczas uzupełniania formularzy rejestracyjnych, edycji profilu użytkownika, zarządzania wizytami czy wystawiania e-recept.



## Charakterystyka słowników

Wszystkie słowniki dostępne w systemie mają charakter **tylko do odczytu** z perspektywy zwykłych użytkowników systemu (Pacjent, Pielęgniarka, Lekarz). Oznacza to, że nie mają oni możliwości samodzielnego dodawania, edytowania ani usuwania wpisów słownikowych. Taka konstrukcja zapewnia bezpieczeństwo oraz jednolitość danych wykorzystywanych w całej aplikacji.

Zarządzanie treścią słowników możliwe jest wyłącznie z poziomu **panelu administracyjnego Django**, który dostępny jest dla użytkowników z rolą **Administratora**.

## Dostępne słowniki w systemie

W systemie zaimplementowano następujące słowniki:

- **Lista krajów** – wykorzystywana głównie w formularzach adresowych (np. podczas rejestracji użytkownika).
- **Lista gabinetów lekarskich** – niezbędna przy planowaniu i edytowaniu wizyt.
- **Lista chorób** – używana formularzu dodawania i edycji wizyty
- **Lista leków** – udostępniana podczas wystawiania recept.
- **Lista specjalizacji lekarskich** – przypisywana lekarzom w celu określenia ich kompetencji.

Każdy ze słowników jest prezentowany w aplikacji w postaci listy rozwijanej lub komponentu z funkcją autouzupełniania, co znacząco ułatwia wybór odpowiedniej wartości i minimalizuje ryzyko błędów podczas wprowadzania danych.

## Przykładowe użycie słowników

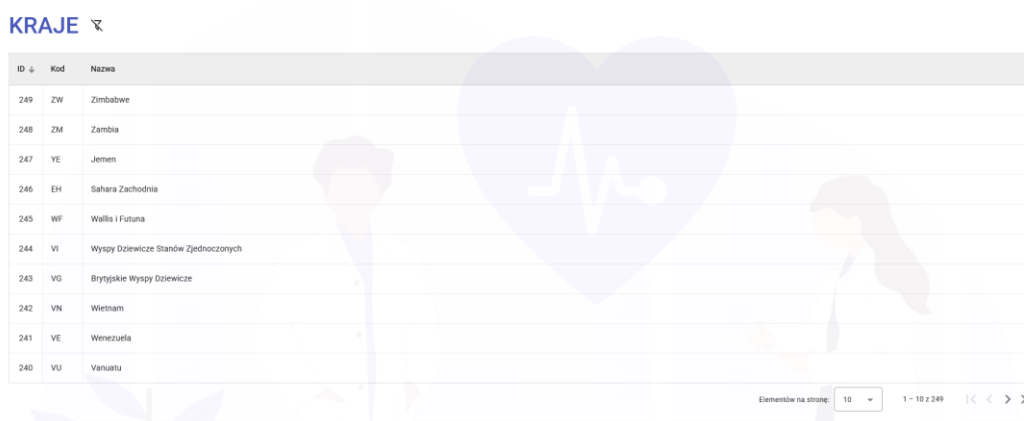
- Podczas rejestracji nowego użytkownika konieczne jest wskazanie kraju zamieszkania z listy krajów.
- W przypadku lekarza, w jego profilu możliwe jest przypisanie jednej lub kilku specjalizacji z listy dostępnych specjalizacji.
- Pielęgniarki przy planowaniu wizyty przypisują ją do konkretnego gabinetu z listy gabinetów.
- Lekarz, wystawiając receptę, wybiera z listy nazwę leku

## Utrzymanie i administracja słownikami

Administrator systemu może zarządzać zawartością słowników z poziomu interfejsu panelu **Django Admin**. Ma możliwość:

- dodawania nowych wpisów (np. nowego kraju, leku czy choroby),
- edycji istniejących rekordów,
- usuwania danych, które są nieaktualne lub błędne.

Dzięki centralnemu zarządzaniu słownikami aplikacja zachowuje integralność danych, a użytkownicy korzystają zawsze z aktualnych i jednolitych zestawów wartości.



ID	Kod	Nazwa
249	ZW	Zimbabwe
248	ZM	Zambia
247	YE	Jemen
246	EH	Sahara Zachodnia
245	WF	Wallis i Futuna
244	VI	Wyspy Dziewicze Stanów Zjednoczonych
243	VG	Brytyjskie Wyspy Dziewicze
242	VN	Wietnam
241	VE	Wenezuela
240	VU	Vanuatu

Rysunek 19: Widok listy krajów

## 3.5. Wizyty

Moduł obsługi wizyt stanowi kluczowy element aplikacji. Umożliwia on zarządzanie spotkaniami pacjentów z personelem medycznym, zapewniając przejrzysty interfejs do przeglądania, planowania i edycji wizyt. W zależności od przypisanej roli użytkownika, dostęp do funkcjonalności może być ograniczony.

### Przeglądanie wizyt

System umożliwia przeglądanie wizyt medycznych w formie tabeli zawierającej najważniejsze informacje: datę i godzinę wizyty, lekarza, pacjenta, lokalizację (piętro, numer pokoju), czas trwania, formę wizyty (stacjonarna/zdalna) oraz jej status.

- **Pacjent** ma dostęp wyłącznie do listy swoich wizyt.
- **Lekarz** widzi wyłącznie wizyty, które zostały do niego przypisane.
- **Pielęgniarka** i **administrator** mają dostęp do pełnej listy wszystkich wizyt w systemie.

Tabela wizyt zawiera również możliwość rozwinięcia szczegółów – np. wyświetlenia notatek powiązanych z daną wizytą.

The screenshot shows a web interface titled 'WIZYTY'. At the top right is a 'Dodaj wizytę' button. Below it is a search form with fields for ID, Date (from/to), Doctor's name, Doctor's surname, Patient's name, Patient's surname, Duration (min/max), and Status. There are 'Zastosuj' and 'Wyczyść' buttons. Below the form is a table of visits with columns: ID, Data wizyty, Imię lekarza, Nazwisko lekarza, Imię pacjenta, Nazwisko pacjenta, Piętro, Nr pokoju, Czas trwania, Czy zdalna?, and Status. The first row shows a visit on 12.04.2025 at 12:00 by Jan Kowalski for Mateusz Szymczak, 3rd floor, room 123, 20 minutes, planned status. At the bottom right, there is a pagination control showing '1 - 1 z 1'.

Rysunek 20: Widok listy wizyt

### Dodawanie wizyt

Tworzenie nowej wizyty jest możliwe wyłącznie dla użytkowników z rolą **pielęgniarki**.

Formularz dodawania wizyty zawiera następujące pola:

- **Pacjent** – wybierany z listy istniejących użytkowników,
- **Lekarz** – specjalista, do którego przypisywana jest wizyta,
- **Data i godzina wizyty** – określenie terminu spotkania,
- **Czas trwania wizyty** – podawany w minutach,
- **Czy zdalna?** – wybór między wizytą zdalną a stacjonarną,
- **Gabinet** – miejsce wizyty w przypadku formy stacjonarnej (piętro, pokój),
- **Choroba** – opcjonalne pole słownikowe pozwalające powiązać wizytę z rozpoznaniem,
- **Notatki** – miejsce na dodatkowe informacje, np. zgłaszane przez pacjenta objawy.

Formularz waliduje poprawność danych i nie pozwala na zapisanie niekompletnej lub błędnej wizyty.

### Edycja i usuwanie wizyt

Pielęgniarka może również **edytować** istniejące wizyty lub **usuwać** je z systemu. Funkcje te dostępne są z poziomu widoku listy wizyt za pomocą odpowiednich ikon (ołówki – edycja, kosz – usunięcie).

Podczas edycji użytkownik ma dostęp do pełnego formularza wizyty, umożliwiającego aktualizację takich danych jak godzina, lekarz, gabinet czy notatki.

Usunięcie wizyty jest operacją trwałą i wymaga potwierdzenia, aby zapobiec przypadkowym modyfikacjom.

### Szczegóły wizyty

Każda wizyta zawiera komplet informacji niezbędnych do jej przeprowadzenia. Poza danymi kontaktowymi pacjenta i lekarza oraz miejscem wizyty, przechowywane są także:

- **Notatki medyczne** – wprowadzane przez pielęgniarkę, mogą zawierać opis objawów zgłaszanych przez pacjenta,
- **Informacje pomocnicze** – np. przypisana choroba.

Dodatkowo, **status wizyty** ustawiany jest automatycznie na podstawie bieżącej daty oraz informacji o dacie, godzinie rozpoczęcia i czasie trwania wizyty. Aplikacja może przypisać jeden z trzech statusów:

- **Zaplanowana** – jeśli czas rozpoczęcia wizyty jest w przyszłości,
- **W toku** – jeśli bieżąca data i godzina mieszczą się w przedziale czasowym trwania wizyty (czyli między godziną rozpoczęcia a zakończenia),
- **Zakończona** – jeśli wizyta już się odbyła i jej czas zakończenia minął.

Dzięki automatycznej aktualizacji statusów użytkownicy mają bieżący wgląd w to, które wizyty aktualnie trwają, które są zaplanowane, a które zostały już zakończone. Informacja ta widoczna jest w kolumnie „Status” na liście wizyt oraz może być wykorzystana do filtrowania danych w interfejsie użytkownika.

Wszystkie dane prezentowane są w sposób czytelny zarówno w widoku edycji, jak i w podglądzie listy.

#### Edycja wizyty

<b>Pacjent *</b>		<b>Lekarz *</b>	
<input type="text" value="Mateusz Szymczak (PESEL: 01210307910)"/>		<input type="text" value="Jan Kowalski (nr wykonywania zawodu: 1234567)"/>	
<b>Data wizyty *</b>	<b>Godzina wizyty *</b>	<b>Czas trwania wizyty [min] *</b>	<b>Czy zdalna? *</b>
<input type="text" value="12.04.2025"/>	<input type="text" value="12:00"/>	<input type="text" value="20"/>	<input type="button" value="Tak"/> <input checked="" type="button" value="Nie"/>
<b>Gabinet *</b>	<b>Choroba</b>		
<input type="text" value="Chirurgiczny (piętro: 3, pokój: 123)"/>	<input type="text" value="Wybierz chorobę"/>		
<b>Notatki</b>			
<input type="text" value="Pacjent zgłasza ból brzucha"/>			

Rysunek 21: Formularz dodawania/edycji wizyty

### 3.6. Recepty

Moduł zarządzania receptami w aplikacji umożliwia tworzenie, przeglądanie i analizowanie e-recept. Jest to kluczowa funkcjonalność systemu wspierająca proces leczenia i umożliwiającą lekarzowi szybkie wystawianie zaleceń farmakologicznych dla pacjenta.

#### Przeglądanie recept

Widok recept dostępny jest dla różnych ról w ograniczonym zakresie:

- **Pacjent** ma dostęp wyłącznie do listy własnych recept.
- **Lekarz** widzi wszystkie recepty, które sam wystawił.
- **Administrator** może przeglądać wszystkie recepty w systemie.

Interfejs prezentuje podstawowe informacje o każdej receptce, takie jak: ID, kod recepty, imię i nazwisko lekarza oraz pacjenta, data wystawienia, data ważności i opis. Istnieje możliwość rozwinięcia pozycji w tabeli, aby zobaczyć szczegóły dawkowania.

Na górze widoku dostępny jest panel filtrowania, umożliwiający wyszukiwanie recept według różnych kryteriów, np. nazwiska pacjenta, daty wystawienia czy opisu.

ID	Kod recepty	Imię lekarza	Nazwisko lekarza	Imię pacjenta	Nazwisko pacjenta	Data wystawienia	Data ważności	Opis
1	0693	Jan	Kowalski	Mateusz	Szymczak	08.04.2025	08.05.2025	Pacjent zmaga się z silnym przeziębieniem, dlatego został mu przepisany APAP oraz Ibuprofen

Dawkowanie

- APAP, 1x Tabletki, rano
- Ibuprofen, 2x Tabletki, rano i wieczorem

Rysunek 22: Widok listy recept

#### Tworzenie recept

Nowa recepta tworzona jest w dwuetapowym formularzu:

##### Krok 1: Podstawowe dane

- **Pacjent** – wybierany z listy pacjentów.
- **Opis** – opis objawów lub powód wystawienia recepty.

## Dodawanie recepty

1 Podstawowe dane 2 Dawkowanie

Pacjent \*

Wybierz pacjenta

Opis \*

Wpisz opis

Anuluj Dalej

Rysunek 23: Formularz dodawania recepty - podstawowe dane

### Krok 2: Dawkowanie

- **Lek** – wybierany z listy słownikowej.
- **Ilość** – liczba jednostek leku.
- **Częstotliwość** – opis sposobu dawkowania (np. „2x dziennie, rano i wieczorem”).

Formularz umożliwia dodanie wielu pozycji dawkowania. Wszystkie pola są wymagane i podlegają walidacji.

## Dodawanie recepty

1 Podstawowe dane 2 Dawkowanie

Dawkowanie \*

Lek \* Ilość \* Częstotliwość \*

Wybierz lek Wpisz ilość Wpisz częstotliwość

Dodaj pozycję

Wstecz Anuluj Zapisz

Rysunek 24: Formularz dodawania recepty - dawkowanie

Recepty mogą być tworzone wyłącznie przez użytkowników z rolą **lekarza** lub **administratora**. W przypadku, gdy zalogowany użytkownik posiada rolę lekarza, pole lekarza w formularzu jest automatycznie wypełniane jego danymi. Dla administratora pole to pozostaje puste i wymaga ręcznego wyboru lekarza z listy.

## Ustawianie dat

System automatycznie ustawia dwie daty:

- **Data wystawienia** – jest ustawiana przez backend na aktualną datę w momencie zapisu recepty,
- **Data ważności** – obliczana jako +30 dni od daty wystawienia.

Dzięki automatyzacji tych pól formularz jest prostszy w obsłudze, a spójność danych jest zachowana niezależnie od roli użytkownika.

## Podsumowanie i plany rozwoju

Celem niniejszej pracy było zaprojektowanie oraz implementacja aplikacji wspomagającej wybrane procesy zarządzania przychodnią lekarską. System został oparty na architekturze klient-serwer, z wykorzystaniem technologii Angular po stronie frontendowej oraz Django REST Framework po stronie backendu. Wdrożono mechanizmy takie jak: rejestracja i logowanie użytkowników z obsługą JWT, zarządzanie rolami, obsługa wizyt, panel administracyjny, formularze oraz system autoryzacji i uprawnień.

Zaimplementowane funkcje pozwalają na sprawną obsługę pacjentów, personelu medycznego oraz częściowo administratora. Aplikacja została przygotowana w sposób modułowy i rozszerzalny, z uwzględnieniem dobrych praktyk programistycznych (testy, CI/CD, dokumentacja Swagger, reCAPTCHA, throttling).

W dalszych etapach planowany jest rozwój aplikacji o następujące funkcjonalności:

- **Generowanie recept w formacie PDF** – umożliwienie lekarzom wystawiania recept, które będzie można pobierać lub drukować w standardowym formacie.
- **Automatyczne pobieranie danych adresowych z bazy TERYT** – ułatwienie wypełniania formularzy adresu dzięki integracji z rejestrem GUS.
- **Rozszerzenie panelu administracyjnego** – przeniesienie większej części funkcjonalności z domyślnego panelu Django Admin do interfejsu aplikacji frontendowej, z uwzględnieniem bardziej przyjaznego UI.
- **Responsywność interfejsu użytkownika** – dostosowanie aplikacji do urządzeń mobilnych (smartfony, tablety), co zwiększy jej dostępność w codziennej pracy personelu.
- **Historia leczenia pacjenta** – dodanie funkcjonalności umożliwiającej lekarzowi przegląd wszystkich wizyt, recept, notatek oraz skierowań przypisanych do danego pacjenta, co ułatwi kompleksową opiekę medyczną.
- **Możliwość wydruku badań, skierowań i innych dokumentów** – wprowadzenie możliwości generowania dokumentów w formacie PDF (np. potwierdzenia wizyt, wyników badań laboratoryjnych, skierowań), które będzie można zapisać lokalnie lub wydrukować.
- **Dodanie modułu skierowań** – umożliwienie lekarzom wystawiania i zarządzania skierowaniami, z możliwością śledzenia ich statusu.
- **Wdrożenie standardów dostępności WCAG** – poprawa dostępności aplikacji dla osób z niepełnosprawnościami zgodnie z wytycznymi WCAG 2.1.



- **Samodzielna rejestracja na wizyty przez pacjentów** – obecnie zapisu dokonuje pielęgniarka, jednak w przyszłości planowane jest umożliwienie pacjentom samodzielnego umawiania wizyt poprzez interfejs aplikacji, z uwzględnieniem dostępnych terminów i lekarzy.
- **Wdrożenie testów frontendowych (E2E) z wykorzystaniem Playwright** – w celu zwiększenia niezawodności aplikacji planowane jest dodanie automatycznych testów end-to-end, które pozwolą na symulację rzeczywistych zachowań użytkownika i szybsze wykrywanie potencjalnych błędów w interfejsie graficznym.
- **Integracja z zewnętrznym API leków** – obecnie informacje o lekach pobierane są z wewnętrznego źródła (plik JSON) generowanym za pomocą sztucznej inteligencji. W przyszłości planowana jest integracja z publicznie dostępnym, zewnętrznym rejestrem leków (np. baza URPL), co pozwoli na uzyskiwanie zawsze aktualnych danych o dostępnych preparatach i ich właściwościach.
- **Automatyczna wysyłka powiadomień e-mail** – rozszerzenie obecnego systemu mailowego o dodatkowe powiadomienia, takie jak:
  - potwierdzenie umówionej wizyty – po zapisaniu pacjenta na wizytę,
  - informacja o wystawieniu recepty – gdy lekarz zakończy proces wypisywania recepty.

Tego typu automatyczne wiadomości zwiększą komfort użytkowników i poprawią komunikację między personelem a pacjentami.

- **Dodanie tłumaczeń w aplikacji** – planowane jest wdrożenie mechanizmu tłumaczeń interfejsu użytkownika (i18n), co pozwoli na obsługę wielu języków (np. polski, angielski) i zwiększy dostępność aplikacji dla szerszego grona odbiorców.
- **Dodanie tłumaczeń do panelu Django Admin** – panel administracyjny zostanie rozszerzony o pełne wsparcie języka polskiego, dzięki czemu zarządzanie systemem będzie bardziej intuicyjne dla administratorów niemających doświadczenia z językiem angielskim.

Projekt dostarczył solidnych podstaw do dalszego rozwoju aplikacji oraz potwierdził skuteczność przyjętych rozwiązań technologicznych. Wdrożenie opisanych usprawnień pozwoli w przyszłości zwiększyć funkcjonalność systemu oraz lepiej dostosować go do rzeczywistych potrzeb placówek medycznych.

## Bibliografia

- [1] Visual Studio Code – <https://code.visualstudio.com/> [dostęp: 30.03.2025]
- [2] Git – <https://git-scm.com/> [dostęp: 30.03.2025]
- [3] GitHub – <https://github.com/> [dostęp: 30.03.2025]
- [4] HTML - <https://developer.mozilla.org/pl/docs/Web/HTML/> [dostęp: 30.03.2025]
- [5] SCSS - <https://sass-lang.com/guide/> [dostęp: 30.03.2025]
- [6] TypeScript - <https://www.typescriptlang.org/> [dostęp: 30.03.2025]
- [7] Angular – <https://angular.io/> [dostęp: 30.03.2025]
- [8] Bootstrap – <https://getbootstrap.com/> [dostęp: 30.03.2025]
- [9] RxJS - <https://rxjs.dev> [dostęp: 30.03.2025]
- [10] Angular Material – <https://material.angular.io/> [dostęp: 30.03.2025]
- [11] Python – <https://www.python.org/about/> [dostęp: 30.03.2025]
- [12] Django Rest Framework – <https://www.django-rest-framework.org/> [dostęp: 30.03.2025]
- [13] Pytest – <https://docs.pytest.org/en/stable/> [dostęp: 30.03.2025]
- [14] Swagger / OpenAPI - <https://swagger.io/specification/> [dostęp: 30.03.2025]
- [15] PostgreSQL – <https://www.postgresql.org/about/> [dostęp: 30.03.2025]
- [16] Docker Compose – <https://docs.docker.com/compose/> [dostęp: 30.03.2025]
- [17] GitHub Actions - <https://github.com/features/actions/> [dostęp: 30.03.2025]
- [18] django-countries - <https://pypi.org/project/django-countries/> [dostęp: 30.03.2025]
- [19] django-rest-framework-simplejwt - <https://django-rest-framework-simplejwt.readthedocs.io/en/latest/> [dostęp: 31.03.2025]
- [20] localStorage - <https://www.frontstack.pl/blog/czym-jest-local-storage-i-jak-uzywac/> [dostęp: 31.03.2025]
- [21] drf-spectacular - <https://drf-spectacular.readthedocs.io/en/latest/> [dostęp: 01.04.2025]
- [22] ngx-toastr - <https://www.npmjs.com/package/ngx-toastr/> [dostęp: 03.04.2025]
- [22] Ruff - <https://docs.astral.sh/ruff/> [dostęp: 03.04.2025]
- [23] Prettier - <https://prettier.io/> [dostęp: 03.04.2025]
- [24] eslint - <https://eslint.org/> [dostęp: 03.04.2025]
- [25] pytest-django - <https://pytest-django.readthedocs.io/en/latest/> [dostęp: 03.04.2025]

## Spis fragmentów kodu

Skrypt w PostgreSQL generujący sekwencje przyjaznych identyfikatorów .....	10
Przykładowa klasa uprawnień dla roli administratora.....	16
Walidacja pola recaptcha_response w serializerze.....	17
Metoda sprawdzająca odpowiedź reCAPTCHA .....	17
Klasa odpowiedzialna za wysyłkę wiadomości e-mail .....	18
Model wizyty.....	20
Serializer do zapisu wizyty.....	21
Queryset wizyty.....	21
FilterSet wizyty .....	21
ViewSet wizyty .....	21
Przykład zastosowania dekoratora @extend_schema w widoku .....	22
Test weryfikujący poprawną aktywację konta użytkownika.....	23
Test weryfikujący walidację identyfikatora użytkownika.....	23
Test sparametryzowany.....	23
Przykładowa konfiguracja trasy .....	26
Definicja roleGuard .....	27
Przykładowy serwis danych .....	29
Przykładowa fasada .....	30
Klasa Endpoints.....	30
Przykład zastosowania dyrektywy atrybutowej .....	32
Przykład zastosowania dyrektywy strukturalnej .....	33
Klasa ListParams - budowanie zapytania do backendu.....	34
Przykład generowania kolumny typu SELECT .....	35
Tworzenie dynamicznego formularza w FormComponentBase .....	36
Przykład ustawiania komunikatu błędu w FieldComponentBase .....	37
Obsługa błędów z backendu .....	38
Przykład ochrony formularza przed zamknięciem.....	38

## Spis rysunków

Schemat bazy danych .....	6
Interfejs Swagger UI .....	22
Widok zakończonego sukcesem workflow GitHub Actions .....	41
Dashboard niezalogowanego użytkownika .....	42
Formularz rejestracji - dane podstawowe .....	43
Formularz rejestracji - adres .....	44
Formularz rejestracji - weryfikacja ReCAPTCHA .....	44
E-mail weryfikacyjny .....	45
Przekierowanie z linku aktywacyjnego .....	45
Formularz logowania .....	46
Formularz resetowania hasła .....	47
Wiadomość e-mail z linkiem resetowania hasła .....	47
Formularz ustawiania nowego hasła .....	48
Dasboard pacjenta .....	49
Dashboard pielęgniarki .....	50
Dashboard lekarza .....	51
Dashboard admina .....	51
Panel administracyjny Django .....	52
Widok listy krajów .....	54
Widok listy wizyt .....	55
Formularz dodawania/edycji wizyty .....	56
Widok listy recept .....	57
Formularz dodawania recepty - podstawowe dane .....	58
Formularz dodawania recepty - dawkowanie .....	58

## Spis tabel

Struktura encji i relacje w module użytkownika i pacjenta.....	7
Struktura encji i relacje w module medycznym .....	8
Struktura encji i relacje w module lokalizacji .....	8
Struktura encji i relacje w module uprawnień i ról .....	8
Podsumowanie relacji między encjami .....	9
Wykorzystywane zmienne środowiskowe .....	13