Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Kierunek: Informatyka techniczna

Specjalność: Systemy informatyki w medycynie

SymptoCheck

Projekt aplikacji webowej wykorzystującej API Gemini do analizy objawów chorobowych

Autorzy:
Amelia Draga 272866
Artur Gierlak 272957

Prowadzący:
dr hab. inż. Mariusz Topolski

Wrocław, listopad 2025 r.

Spis treści

Spis rysunków

1 Wstęp

W ramach zajęć *Projektowanie systemów informatyki medycznej* należało stworzyć dowolną aplikację o tematyce medycznej. Dodatkowym wymaganiem projektu było zaimplementowanie **API**, które umożliwia wymianę danych pomiędzy warstwami systemu lub integrację z zewnętrznym serwisem.

1.1 Cel projektu

Celem projektu **SymptoCheck** było stworzenie aplikacji internetowej o tematyce medycznej, wykorzystującej sztuczną inteligencję do wspomagania wstępnej diagnozy na podstawie objawów podanych przez użytkownika. Projekt miał na celu połączenie technologii webowych z dostępem do modelu językowego *Gemini 2.5 Flash* udostępnianego przez API Google. Dzięki temu użytkownik może w prosty sposób uzyskać propozycję potencjalnych diagnoz, prowadzić rozmowę z wirtualnym asystentem oraz przeglądać historię wcześniejszych analiz. Aplikacja nie zastępuje profesjonalnej konsultacji lekarskiej – jej zadaniem jest jedynie wsparcie informacyjne poprzez analizę opisowych danych o objawach.

1.2 Opis systemu

System **SymptoCheck** został zaprojektowany jako dwuwarstwowa aplikacja webowa składająca się z:

- **frontendu** zbudowanego w technologii **React (Vite, TailwindCSS)** , odpowiedzialnego za interakcję z użytkownikiem, prezentację wyników oraz czat z asystentem,
- **backendu** opartego na frameworku **Django REST**, który odpowiada za przetwarzanie żądań, komunikację z **API modelu Gemini** oraz zapisywanie historii analiz w lokalnej bazie danych **SQLite**.

Głównym celem działania systemu jest umożliwienie użytkownikowi uzyskania wstępnej, poglądowej diagnozy na podstawie opisanych objawów. Dodatkowo aplikacja pozwala na rozmowę z wirtualnym asystentem medycznym oraz przeglądanie historii wcześniejszych analiz, co zwiększa jej funkcjonalność i praktyczne zastosowanie.

1.3 Zakres projektu

W ramach projektu **SymptoCheck** udało się zrealizować wszystkie główne założenia funkcjonalne oraz techniczne systemu. Prace zostały podzielone na część frontendową i backendową, a następnie połączone w działającą całość komunikującą się poprzez **REST API**. Po stronie **backendu** zrealizowano:

- konfigurację środowiska Django wraz z integracją *Django REST Framework*,

- stworzenie modelu danych `Analysis`, który zapisuje historię wykonanych analiz (objawy, wynik, data),
- implementację trzech głównych endpointów API:
  - `/diagnose` – wysyła opis objawów do modelu *Gemini 2.5 Flash* i zwraca analizę w formacie Markdown,
  - `/chat` – umożliwia prowadzenie rozmowy z modelem w kontekście medycznym,
  - `/analyses` – pozwala na pobranie zapisanej historii analiz,
- integrację z **Gemini API** przy użyciu oficjalnej biblioteki `google-generativeai`,
- obsługę wyjątków i błędów API, takich jak brak połączenia lub nieprawidłowy klucz.

Po stronie **frontendu** wykonano:

- zaprojektowanie i zaimplementowanie trzech głównych komponentów:
  - `SymptomForm.jsx` – formularz, w którym użytkownik wpisuje objawy i otrzymuje wynik diagnozy,
  - `ChatAssistant.jsx` – moduł czatu, który pozwala prowadzić rozmowę z asystentem AI,
  - `HistoryList.jsx` – lista historii poprzednich analiz,
- zastosowanie biblioteki *React Markdown* do czytelnego wyświetlania wyników zwróconych przez model,
- opracowanie prostego i intuicyjnego interfejsu użytkownika z wykorzystaniem *TailwindCSS*,
- konfigurację komunikacji z backendem poprzez żądania HTTP metodą `POST` i `GET`.

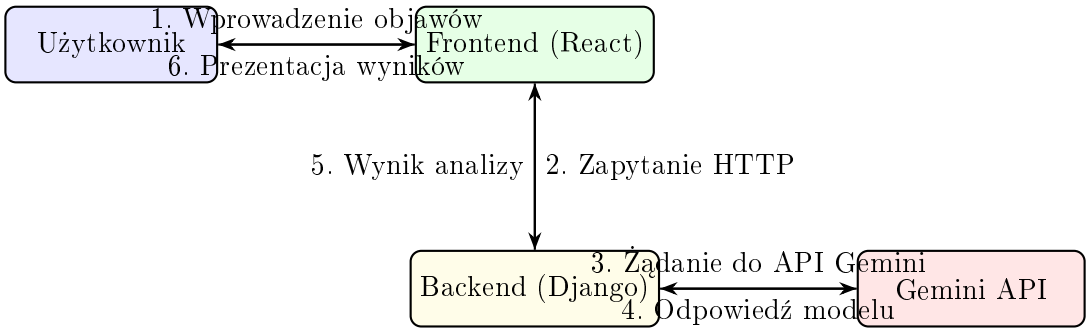
W efekcie końcowym powstała w pełni działająca aplikacja, która umożliwia analizę objawów i uzyskanie możliwych diagnoz w oparciu o model AI, prowadzenie rozmowy z inteligentnym asystentem medycznym oraz zapisywanie i przeglądanie historii wcześniejszych analiz. Projekt został ukończony zgodnie z założeniami i spełnia wszystkie wymagania funkcjonalne określone na początku realizacji.

1.4 Architektura systemu

Aplikacja **SymptoCheck** została zaprojektowana w architekturze dwuwarstwowej, składającej się z części **frontendowej** oraz **backendowej**. Obie warstwy komunikują się ze sobą za pomocą interfejsu **REST API**, co zapewnia prostą i niezawodną wymianę danych między użytkownikiem a serwerem.

- **Frontend** — odpowiada za interfejs użytkownika i obsługę logiki po stronie przeglądarki. Został stworzony przy użyciu technologii *React.js*, w środowisku *Vite*, z wykorzystaniem frameworka *TailwindCSS* do stylizacji. Użytkownik wprowadza dane (objawy) w formularzu, a aplikacja przesyła je do serwera w formacie JSON. Wyniki analizy są następnie prezentowane w przejrzystej formie.
- **Backend** — pełni rolę warstwy serwerowej i został stworzony w *Django REST Framework*. Jego zadaniem jest przyjmowanie żądań z frontendu, komunikacja z zewnętrznym API modelu *Gemini 2.5 Flash* oraz zwracanie przetworzonych wyników do interfejsu użytkownika. Backend przechowuje również historię analiz w lokalnej bazie danych *SQLite*.

Komunikacja pomiędzy frontendem a backendem odbywa się z wykorzystaniem metod `POST` i `GET`. Frontend przesyła zapytanie z opisem objawów na odpowiedni endpoint API, a backend wysyła dane do modelu *Gemini 2.5 Flash*, który generuje analizę w formacie tekstowym (Markdown). Następnie wynik jest przekazywany z powrotem do użytkownika i zapisywany w historii.



Rysunek 1: Schemat przepływu danych w architekturze systemu SymptoCheck.

Każda z warstw systemu działa niezależnie, co umożliwia łatwe modyfikacje oraz rozwijanie projektu w przyszłości. Na przykład możliwe jest wdrożenie backendu w chmurze, a frontendu jako aplikacji PWA (Progressive Web App). Taka struktura zapewnia także lepszą skalowalność i elastyczność w dalszym rozwoju systemu.

Podsumowując, architektura **SymptoCheck** opiera się na prostym, ale skutecznym połączeniu Reacta po stronie klienta i Django REST po stronie serwera, z wykorzystaniem API *Gemini 2.5 Flash* jako zewnętrznego źródła inteligentnej analizy.

2 Implementacja backendu

2.1 Struktura projektu Django

Backend został zorganizowany zgodnie z konwencją Django. Główne komponenty:

- `backend/` – folder główny projektu Django z plikami konfiguracyjnymi (`settings.py`, `urls.py`),
- `core/` – aplikacja Django zawierająca logikę biznesową, modele, widoki i endpointy API,
- `manage.py` – skrypt do zarządzania projektem (migracje, uruchamianie serwera, testy).

2.2 Model danych

W bazie danych SQLite zdefiniowano model `Analysis`, który przechowuje historię analiz użytkownika:

```
from django.db import models

class Analysis(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    symptoms = models.TextField()
    result_md = models.TextField()  # wynik w formacie Markdown
    confidence_json = models.JSONField(null=True, blank=True)

    def __str__(self):
        return f"Analysis #{self.id} @ {self.created_at:%Y-%m-%d %H:%M}"
```

Pola modelu:

- `created_at` – data i czas utworzenia analizy (ustawiana automatycznie),
- `symptoms` – tekst wprowadzony przez użytkownika opisujący objawy,
- `result_md` – wynik analizy zwrócony przez model AI w formacie Markdown,
- `confidence_json` – opcjonalne dane JSON zawierające prawdopodobieństwa poszczególnych diagnoz.

2.3 Endpointy API

Backend udostępnia cztery endpointy RESTful:

2.3.1 POST /api/diagnose/

Endpoint odpowiedzialny za wykonanie analizy objawów.
Żądanie:

```
{
  "symptoms": "b  l␣g Ćowy,␣gor czka,␣kaszel"
}
```

Odpowiedź:

```
{
  "id": 1,
  "result_md": "**Grypa**\nInfekcja␣wirusowa...",
  "confidence": {
    "items": [
      {"name": "Grypa", "prob": 0.75},
      {"name": "COVID-19", "prob": 0.65}
    ]
  }
}
```

Implementacja widoku wykorzystuje model `GenerativeModel` z biblioteki `google-generativeai`:

```
class DiagnoseView(APIView):
    def post(self, request):
        symptoms = (request.data.get("symptoms") or "").strip()
        if not symptoms:
            return Response({"error": "Brak␣pola␣'symptoms'."},
                            status=status.HTTP_400_BAD_REQUEST)

        try:
            model = genai.GenerativeModel(
                model_name="models/gemini-2.5-flash"
            )
            prompt = PROMPT_TEMPLATE.format(symptoms=symptoms)
            result = model.generate_content(prompt)
            text = result.text or ""

            # Parsowanie JSON z prawdopodobie stwami
            confidence = extract_confidence_json(text)

            analysis = Analysis.objects.create(
                symptoms=symptoms,
                result_md=text,
                confidence_json=confidence
            )

            return Response({
                "id": analysis.id,
                "result_md": analysis.result_md,
                "confidence": analysis.confidence_json
            })
        except Exception as e:
            return Response({"error": str(e)}, status=500)
```

2.3.2 GET /api/analyses/

Zwraca listę ostatnich 20 analiz użytkownika.

Odpowiedź:

```
[
  {
    "id": 1,
    "created_at": "2025-11-15T14:30:00Z",
    "symptoms": "b l␣g Ćowy,␣gor czka",
    "result_md": "..."
  }
]
```

2.3.3 POST /api/chat/

Umożliwia prowadzenie rozmowy z asystentem AI.

Żądanie:

```
{
  "message": "Czy␣powinienem␣si Ź␣martwi ?",
  "history": [
    {"role": "user", "content": "Mam␣b l␣g Ćowy"},
    {"role": "assistant", "content": "Mo e␣to␣by ..."}
  ]
}
```

Odpowiedź:

```
{
  "reply": "Je Żli␣objawy␣utrzymuj ␣si Ź,␣warto␣skonsultowa ..."
}
```

2.3.4 GET /api/models/

Endpoint testowy zwracający listę dostępnych modeli w API Gemini.

2.4 Prompt engineering

Kluczowym elementem działania systemu jest odpowiednio skonstruowany prompt dla modelu AI. Przykład:

```
PROMPT_TEMPLATE = """
Jeste Ż inteligentnym asystentem medycznym.
Na podstawie podanych objaw w przedstaw list Ź maksymalnie 5
najbardziej prawdopodobnych diagnoz.

```

```
Dla kadej choroby uyjponiszego formatu Markdown:
**Nazwa choroby (np. Grypa, COVID-19)**
Krtki opis objawww jednym zdaniu.
Sugerowany specjalista: **nazwa lekarza**

Objawy: {symptoms}

Dodatkowo wypisz w formacie JSON obiekt "confidence" z polami:
- "items": lista obiekt w {{"name": string, "prob": liczba 0-1}}
"""
```

Prompt instruuje model, aby zwrócił wynik w określonym formacie Markdown wraz z danymi JSON, co umożliwia ich łatwe parsowanie i prezentację w interfejsie użytkownika.

2.5 Obsługa błędów

Backend implementuje podstawową obsługę wyjątków:

- walidacja pustych pól w żądaniach (zwrot błędu 400),
- przechwytywanie wyjątków związanych z API Gemini (błędy połączenia, nieprawidłowy klucz API),
- logowanie błędów do konsoli dla potrzeb debugowania,
- zwracanie komunikatów błędów w formacie JSON do frontendu.

3 Implementacja frontendu

3.1 Architektura komponentowa

Frontend został zbudowany w architekturze komponentowej React. Główne komponenty:

3.1.1 App.jsx

Główny komponent aplikacji odpowiedzialny za:

- zarządzanie stanem aktywnej zakładki,
- renderowanie menu nawigacyjnego z trzema opcjami (Analiza, Historia, Czat),
- warunkowe wyświetlanie odpowiedniego komponentu na podstawie wybranej zakładki.

Wykorzystuje hook `useState` do zarządzania stanem lokalnym oraz bibliotekę `lucide-react` do wyświetlania ikon w menu.

3.1.2 SymptomForm.jsx

Komponent formularza analizy objawów. Implementacja obejmuje:

- pole tekstowe (`textarea`) do wprowadzania objawów,
- przycisk wywołujący zapytanie POST do endpointu `/api/diagnose/`,
- obsługę stanów: `loading`, `error`, `analysis`, `confidence`,
- prezentację wyników za pomocą `ReactMarkdown`,
- wizualizację prawdopodobieństw w formie kolorowych pasków postępu.

Fragment kodu odpowiedzialny za wywołanie API:

```
const handleAnalyze = async () => {
  setLoading(true);
  try {
    const res = await fetch(`${API_BASE}/diagnose/`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ symptoms }),
    });

    const data = await res.json();
    setAnalysis(data.result_md);
    setConfidence(data.confidence?.items);
  } catch (e) {
    setError(e.message);
  } finally {
    setLoading(false);
  }
};
```

Kolorystyka pasków pewności jest dynamicznie dobierana w zależności od poziomu prawdopodobieństwa:

- zielony – prawdopodobieństwo $> 70\%$,
- żółty – prawdopodobieństwo 40-70%,
- czerwony – prawdopodobieństwo $< 40\%$.

3.1.3 ChatAssistant.jsx

Komponent interfejsu czatu implementujący:

- listę wiadomości w formacie `[role, content]`,
- pole tekstowe oraz przycisk wysyłania wiadomości,
- obsługę naciśnięcia klawisza Enter do szybkiego wysyłania,
- wizualne rozróżnienie wiadomości użytkownika (prawo) i asystenta (lewo),
- animację "AI pisze..." podczas oczekiwania na odpowiedź.

Kontekst rozmowy jest przekazywany w każdym żądaniu do backendu, co umożliwia modelowi generowanie odpowiedzi spójnych z wcześniejszą konwersacją.

3.1.4 HistoryList.jsx

Komponent historii analiz realizujący:

- pobieranie listy analiz z endpointu `/api/analyses/` przy montowaniu komponentu (hook `useEffect`),
- wyświetlanie listy w formie klikanych elementów z datą i pierwszymi słowami objawów,
- modalny widok szczegółów analizy z pełnym wynikiem oraz wykresami pewności,
- ponowne parsowanie JSON z prawdopodobieństwami dla każdej analizy.

3.2 Stylizacja z TailwindCSS

Całość interfejsu została ostylizowana przy użyciu klas użytkowych TailwindCSS. Przykładowe zastosowania:

- `bg-blue-50` – jasne tło w odcieniu błękitu,
- `rounded-2xl` – mocno zaokrąglone rogi elementów,
- `shadow-md`, `shadow-lg` – cienie nadające głębię,
- `hover:shadow-lg` – interaktywne efekty przy najechaniu kursorem,

- `transition-all` – płynne przejścia między stanami.

Dzięki TailwindCSS, cały interfejs jest responsywny i dostosowuje się do różnych rozmiarów ekranów.

3.3 Komunikacja z backendem

Frontend komunikuje się z backendem za pomocą standardowego API `fetch`:

- zapytania POST do endpointów `/diagnose` i `/chat` przesyłają dane w formacie JSON,
- zapytania GET do endpointu `/analyses` pobierają historię,
- nagłówek `Content-Type: application/json` zapewnia poprawną interpretację danych.

Adres bazowy API jest konfigurowany przez zmienną środowiskową `VITE_API_BASE`, co umożliwia łatwą zmianę adresu w zależności od środowiska (lokalne / produkcyjne).

3.4 Renderowanie Markdown

Odpowiedzi z API są w formacie Markdown, co wymaga konwersji na HTML. Do tego celu używana jest biblioteka `react-markdown` z pluginem `remark-gfm` (GitHub Flavored Markdown):

```
<ReactMarkdown remarkPlugins={[remarkGfm]}>
  {analysis}
</ReactMarkdown>
```

Dzięki temu wyniki są prezentowane z właściwym formatowaniem (pogrubienia, listy, paragrafy).

4 Technologie i narzędzia

4.1 Backend

Backend aplikacji został zbudowany w oparciu o następujące technologie:

- **Django 5.x** – framework webowy w Pythonie zapewniający solidne fundamenty dla aplikacji serwerowej,
- **Django REST Framework (DRF)** – rozszerzenie Django ułatwiające tworzenie RESTful API z serializacją danych, walidacją oraz obsługą żądań HTTP,
- **SQLite** – lekka, wbudowana baza danych wykorzystywana do przechowywania historii analiz,
- **Google Generative AI SDK** (`google-generativeai`) – oficjalna biblioteka Pythona umożliwiająca komunikację z modelem *Gemini 2.5 Flash*,
- **django-cors-headers** – middleware obsługujący politykę CORS (Cross-Origin Resource Sharing), niezbędną do komunikacji frontend-backend,
- **python-dotenv** – narzędzie do bezpiecznego zarządzania zmiennymi środowiskowymi (np. klucz API).

4.2 Frontend

Warstwa prezentacji została zrealizowana przy użyciu nowoczesnego stosu technologicznego:

- **React 18** – biblioteka JavaScript do budowy interfejsu użytkownika w architekturze komponentowej,
- **Vite** – szybkie narzędzie do budowania aplikacji frontendowych z hot module replacement (HMR),
- **TailwindCSS** – framework CSS oparty na klasach użytkowych (utility-first), umożliwiający szybkie stylowanie komponentów,
- **React Markdown** – biblioteka do renderowania treści Markdown otrzymanej z API,
- **Lucide React** – zestaw ikon SVG dla React, wykorzystywany w interfejsie użytkownika.

4.3 API zewnętrzne

- **Google Gemini API (Gemini 2.5 Flash)** – model dużego języka (LLM) od Google, wykorzystywany do generowania diagnoz medycznych oraz prowadzenia rozmowy z użytkownikiem w kontekście medycznym.

4.4 Narzędzia deweloperskie

- **Git** – system kontroli wersji,
- **npm** – menedżer pakietów dla środowiska Node.js,
- **pip** – menedżer pakietów dla Pythona,
- **Visual Studio Code** – środowisko programistyczne wykorzystane podczas rozwoju projektu.

5 Opis funkcjonalny aplikacji

5.1 Główne moduły

Aplikacja **SymptoCheck** składa się z trzech głównych modułów funkcjonalnych:

5.1.1 Analiza objawów

Główny moduł aplikacji, w którym użytkownik wprowadza opis swoich objawów w formie tekstowej. System wysyła zapytanie do modelu AI Gemini 2.5 Flash, który analizuje podane informacje i zwraca:

- listę potencjalnych diagnoz (maksymalnie 5),
- krótki opis każdej choroby,
- sugerowanego specjalistę do konsultacji,
- szacowane prawdopodobieństwo dla każdej diagnozy (w postaci procentowej).

Wyniki są prezentowane w czytelnym formacie Markdown z wizualizacją poziomu pewności w formie kolorowych pasków postępu.

5.1.2 Czat z asystentem AI

Interaktywny moduł umożliwiający prowadzenie rozmowy z inteligentnym asystentem medycznym. Użytkownik może:

- zadawać pytania dotyczące objawów,
- uzyskiwać dodatkowe wyjaśnienia na temat potencjalnych diagnoz,
- otrzymywać porady dotyczące dalszych kroków (np. zalecenia dotyczące wizyty u specjalisty).

System zachowuje kontekst rozmowy, co pozwala na bardziej naturalne i spójne interakcje.

5.1.3 Historia analiz

Moduł archiwizacji przechowujący wszystkie wykonane analizy w lokalnej bazie danych. Użytkownik może:

- przeglądać listę wcześniejszych analiz z datami wykonania,
- otwierać szczegóły każdej analizy w modalnym oknie,
- przeglądać zapisane wyniki wraz z prawdopodobieństwami diagnoz.

Historia jest sortowana chronologicznie (od najnowszych), a system przechowuje do 20 ostatnich analiz.

5.2 Interfejs użytkownika

Interfejs aplikacji został zaprojektowany z naciskiem na prostotę i intuicyjność obsługi. Składa się z trzech głównych zakładek dostępnych z poziomu górnego menu nawigacyjnego:

- **Analiza objawów** – główny ekran z formularzem wprowadzania objawów, wizualizacją pewności diagnoz oraz wynikami analizy,
- **Historia** – lista wcześniej wykonanych analiz z możliwością podglądu szczegółów,
- **Czat** – interfejs rozmowy z asystentem AI w formie czatu.

Każda zakładka jest oznaczona dedykowaną ikoną oraz zmienia kolor po aktywacji, co ułatwia orientację w aplikacji. Kolorystyka oparta jest na odcieniach błękitu i bieli, co nadaje aplikacji profesjonalny, medyczny charakter.

6 Testowanie

6.1 Strategia testowania

W projekcie zastosowano podejście wielopoziomowe obejmujące:

- **testy jednostkowe backendu** – weryfikacja poprawności działania endpointów API,
- **testy integracyjne** – sprawdzenie komunikacji frontend-backend,
- **testy manualne** – weryfikacja funkcjonalności z perspektywy użytkownika końcowego.

6.2 Testy backendu

Backend został przetestowany z wykorzystaniem `Django REST Framework Test Case`. Plik `core/tests.py` zawiera testy jednostkowe dla endpointów API.

6.2.1 Test walidacji pustych objawów

```
from rest_framework.test import APITestCase
from django.urls import reverse

class DiagnoseApiTests(APITestCase):
    def test_empty_symptoms(self):
        """Test, czy endpoint zwraca b Ć d 400 dla pustych objaw w"""
        url = reverse('diagnose')
        response = self.client.post(
            url,
            {"symptoms": ""},
            format='json'
        )
        self.assertEqual(response.status_code, 400)
        self.assertIn('error', response.json())
```

6.2.2 Test poprawnego żądania

```
def test_valid_request(self):
    """Test poprawnego   dania   z objawami"""
    url = reverse('diagnose')
    response = self.client.post(
        url,
        {"symptoms": "b l␣g Ćowy␣i␣gor czka"},
        format='json'
    )
    # Odpowied powinna by  200 lub 500 (w zale no Żci od API)
    self.assertIn(response.status_code, [200, 500])

    if response.status_code == 200:
```

```
        data = response.json()
        self.assertIn('result_md', data)
        self.assertIn('id', data)
```

6.2.3 Test endpointu historii

```
class AnalysisListTests(APITestCase):
    def test_get_analyses(self):
        """Test pobierania listy analiz"""
        url = reverse('analyses')
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)
        self.assertIsInstance(response.json(), list)
```

6.2.4 Test czatu

```
class ChatApiTests(APITestCase):
    def test_chat_empty_message(self):
        """Test czatu z pust  wiadomo Żci """
        url = reverse('chat')
        response = self.client.post(
            url,
            {"message": "", "history": []},
            format='json'
        )
        self.assertEqual(response.status_code, 400)

    def test_chat_valid_message(self):
        """Test czatu z poprawn  wiadomo Żci """
        url = reverse('chat')
        response = self.client.post(
            url,
            {
                "message": "Czy␣to␣mo e␣by ␣grypa?",
                "history": [
                    {"role": "user", "content": "Mam␣b l␣g Ćowy"}
                ]
            },
            format='json'
        )
        self.assertIn(response.status_code, [200, 500])
```

6.3 Uruchamianie testów backendu

Testy uruchamia się za pomocą komendy Django:

```
cd backend
python manage.py test core
```

Przykładowy output:

```
Found 5 test(s).
Creating test database...
......
Ran 5 tests in 2.341s

OK
```

6.4 Testy integracyjne

Testy integracyjne weryfikują poprawność komunikacji między frontendem a backendem:

6.4.1 Test połączenia REST API

1. Uruchomienie backendu: `python manage.py runserver`
2. Uruchomienie frontendu: `npm run dev`
3. Weryfikacja, czy frontend poprawnie wysyła żądania do backendu
4. Sprawdzenie, czy odpowiedzi są poprawnie przetwarzane i wyświetlane

6.4.2 Test zapisu historii

1. Wykonanie analizy objawów przez formularz
2. Przejście do zakładki "Historia"
3. Weryfikacja, czy nowa analiza pojawia się na liście
4. Otwarcie szczegółów analizy i sprawdzenie poprawności danych

6.4.3 Test czatu z kontekstem

1. Otwarcie zakładki "Czat"
2. Wysłanie pierwszej wiadomości z opisem objawów
3. Wysłanie pytania nawiązującego do poprzedniej wiadomości
4. Weryfikacja, czy asystent odpowiada w kontekście całej rozmowy

6.5 Testy manualne

W ramach testów manualnych przeprowadzono:

- **test użyteczności interfejsu** – weryfikacja intuicyjności nawigacji i czytelności wyników,
- **test responsywności** – sprawdzenie działania na różnych rozdzielczościach ekranu,
- **test obsługi błędów** – symulacja błędnych danych wejściowych i braku połączenia z API,
- **test wydajności** – pomiar czasu odpowiedzi systemu dla różnych długości opisów objawów.

6.6 Przypadki brzegowe

Przetestowano również następujące przypadki brzegowe:

- bardzo długi opis objawów (>1000 znaków),
- objawy w języku obcym (np. angielskim),
- nietypowe zapytania (np. "test", "123"),
- brak klucza API w konfiguracji backendu,
- jednoczesne wywołanie wielu żądań (obciążenie równoległe).

7 Konfiguracja i wdrożenie

7.1 Wymagania systemowe

Minimalne wymagania do uruchomienia projektu lokalnie:

- **Python 3.10+** – do uruchomienia backendu Django,
- **Node.js 18+** i **npm** – do uruchomienia frontendu React,
- **Git** – do pobrania repozytorium projektu,
- **Klucz API Google Gemini** – niezbędny do komunikacji z modelem AI.

7.2 Instalacja i uruchomienie backendu

7.2.1 Krok 1: Utworzenie środowiska wirtualnego

```
cd backend
python -m venv .venv
```

7.2.2 Krok 2: Aktywacja środowiska

W systemie Windows:

```
.venv\Scripts\activate
```

W systemach Unix/MacOS:

```
source .venv/bin/activate
```

7.2.3 Krok 3: Instalacja zależności

```
pip install django djangorestframework django-cors-headers
            python-dotenv google-generativeai
```

7.2.4 Krok 4: Konfiguracja zmiennych środowiskowych

Utworzenie pliku `.env` w folderze `backend/`:

```
GEMINI_API_KEY=your_api_key_here
DEBUG=True
SECRET_KEY=your_secret_key
```

Klucz API można wygenerować na stronie: `https://makersuite.google.com/app/apikey`

7.2.5 Krok 5: Migracja bazy danych

```
python manage.py makemigrations
python manage.py migrate
```

7.2.6 Krok 6: Uruchomienie serwera deweloperskiego

```
python manage.py runserver
```

Backend będzie dostępny pod adresem: `http://localhost:8000`

7.3 Instalacja i uruchomienie frontendu

7.3.1 Krok 1: Instalacja zależności

```
cd frontend
npm install
```

7.3.2 Krok 2: Konfiguracja zmiennej środowiskowej

Utworzenie pliku `.env` w folderze `frontend/`:

```
VITE_API_BASE=http://localhost:8000/api
```

7.3.3 Krok 3: Uruchomienie serwera deweloperskiego

```
npm run dev
```

Frontend będzie dostępny pod adresem: `http://localhost:5173`

7.4 Struktura plików konfiguracyjnych

7.4.1 Backend: settings.py

Kluczowe ustawienia Django:

```
# CORS configuration
CORS_ALLOWED_ORIGINS = [
    "http://localhost:5173",  # frontend dev server
]

# Gemini API key
GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")

# Database
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}

# REST Framework
REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
    ]
}
```

7.4.2 Frontend: vite.config.js

Konfiguracja narzędzia budowania:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    port: 5173,
    proxy: {
      '/api': {
        target: 'http://localhost:8000',
        changeOrigin: true,
      }
    }
  }
})
```

7.5 Wdrożenie produkcyjne

Możliwe opcje wdrożenia aplikacji w środowisku produkcyjnym:

7.5.1 Backend

- **Render** – platforma PaaS z darmowym planem dla małych projektów,
- **Railway** – prosty hosting dla aplikacji Django,
- **Heroku** – popularna platforma chmurowa (wymaga konfiguracji `Procfile`),
- **AWS EC2 / Azure VM** – pełna kontrola nad środowiskiem serwerowym.

Przed wdrożeniem produkcyjnym należy:

1. zmienić `DEBUG=False` w `settings.py`,
2. skonfigurować `ALLOWED_HOSTS` z domeną produkcyjną,
3. zastąpić SQLite bazą PostgreSQL lub MySQL,
4. skonfigurować serwer WSGI (Gunicorn) lub ASGI (Uvicorn),
5. zabezpieczyć klucz API w zmiennych środowiskowych platformy.

7.5.2 Frontend

- **Vercel** – dedykowany dla aplikacji React/Vite, darmowy plan,
- **Netlify** – automatyczne wdrożenie z repozytorium Git,
- **GitHub Pages** – hosting statyczny dla aplikacji frontendowych,
- **Cloudflare Pages** – szybki CDN z globalnym zasięgiem.

Build produkcyjny frontendu:

```
npm run build
```

Generuje zoptymalizowane pliki statyczne w folderze `dist/`.

7.6 Konteneryzacja (Docker)

Przykładowa konfiguracja `docker-compose.yml`:

```
version: '3.8'
services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - GEMINI_API_KEY=${GEMINI_API_KEY}
    volumes:
      - ./backend:/app

  frontend:
    build: ./frontend
    ports:
      - "5173:5173"
    depends_on:
      - backend
```

Uruchomienie:

```
docker-compose up --build
```

8 Bezpieczeństwo i prywatność

8.1 Zarządzanie kluczem API

Klucz API Google Gemini jest przechowywany w zmiennych środowiskowych i nigdy nie jest commitowany do repozytorium Git. Zastosowano:

- plik `.env` (dodany do `.gitignore`),
- bibliotekę `python-dotenv` do bezpiecznego ładowania zmiennych,
- przykładowy plik `.env.example` informujący użytkowników o wymaganych zmiennych.

8.2 CORS (Cross-Origin Resource Sharing)

Backend został skonfigurowany z użyciem `django-cors-headers`, aby akceptować żądania tylko z określonych źródeł:

```
CORS_ALLOWED_ORIGINS = [
    "http://localhost:5173",  # frontend development
    "https://your-domain.com" # production frontend
]
```

8.3 Walidacja danych wejściowych

Wszystkie endpointy backendu walidują dane wejściowe:

- sprawdzanie, czy pole `symptoms` nie jest puste,
- walidacja formatu JSON w żądaniach,
- ograniczenie długości pól tekstowych (opcjonalnie).

8.4 Ograniczenia rate limiting

W wersji produkcyjnej zaleca się implementację rate limiting, aby zapobiec nadużyciom:

- **django-ratelimit** – ograniczenie liczby żądań na użytkownika/IP,
- **nginx rate limiting** – ograniczenie na poziomie reverse proxy,
- **Gemini API quota** – Google nakłada limity na liczbę żądań do API.

8.5 Prywatność danych

Aplikacja w obecnej wersji nie implementuje systemu uwierzytelniania użytkowników, co oznacza:

- wszystkie analizy są przechowywane lokalnie bez powiązania z konkretnym użytkownikiem,

- brak mechanizmu rejestracji/logowania,
- dane są zapisywane w lokalnej bazie SQLite na serwerze.

W przyszłości zaleca się:

- implementację systemu kont użytkowników (Django Authentication),
- szyfrowanie danych wrażliwych w bazie,
- zgodność z RODO (prawo do usunięcia danych).

8.6 Bezpieczeństwo API Gemini

Komunikacja z API Gemini odbywa się przez HTTPS, co zapewnia szyfrowanie przesyłanych danych. Należy jednak pamiętać, że:

- dane przesyłane do API Gemini są przetwarzane przez Google,
- użytkownik powinien być poinformowany o przetwarzaniu danych przez zewnętrzny serwis,
- nie należy przesyłać danych osobowych identyfikujących użytkownika (nazwisko, PESEL, itp.).

9 Podsumowanie

9.1 Osiągnięte cele

Projekt **SymptoCheck** został zrealizowany zgodnie z założeniami i spełnia wszystkie wymagania funkcjonalne:

- ✓Zaimplementowano aplikację webową o tematyce medycznej,
- ✓Zintegrowano API zewnętrzne (Google Gemini 2.5 Flash),
- ✓Stworzono dwuwarstwową architekturę (React + Django),
- ✓Zaimplementowano trzy główne moduły: analizę objawów, czat i historię,
- ✓Zapewniono intuicyjny interfejs użytkownika,
- ✓Dodano wizualizację pewności diagnoz,
- ✓Przetestowano funkcjonalność systemu.

9.2 Wnioski

Realizacja projektu pokazała praktyczne wykorzystanie modeli dużych języków (LLM) w dziedzinie medycznej. Model *Gemini 2.5 Flash* okazał się skutecznym narzędziem do:

- generowania sensownych, czytelnych wyników diagnoz,
- prowadzenia spójnej rozmowy z użytkownikiem,
- formatowania odpowiedzi w strukturalnym formacie (Markdown + JSON).

Główne zalety zastosowanych technologii:

- **React** – szybki rozwój interfejsu dzięki komponentom wielokrotnego użytku,
- **Django REST Framework** – prosty w implementacji backend z automatyczną serializacją,
- **TailwindCSS** – szybkie prototypowanie interfejsu bez pisania własnego CSS,
- **Gemini API** – łatwa integracja, dobra jakość odpowiedzi, darmowy plan.

9.3 Ograniczenia systemu

Aplikacja ma pewne ograniczenia, które należy wziąć pod uwagę:

- **Brak weryfikacji medycznej** – wyniki nie są sprawdzane przez personel medyczny,
- **Zależność od API zewnętrznego** – problemy z Google Gemini wpływają na działanie systemu,

- **Brak uwierzytelniania** – każdy może korzystać z aplikacji bez rejestracji,
- **Lokalna baza danych** – SQLite nie jest odpowiednia dla aplikacji produkcyjnych z dużym ruchem,
- **Brak multilingwalności** – interfejs i prompt są tylko w języku polskim,
- **Brak monitoringu** – brak logowania i śledzenia błędów w czasie rzeczywistym.

9.4 Kierunki dalszego rozwoju

Projekt ma duży potencjał rozwoju. Proponowane rozszerzenia:

9.4.1 Funkcjonalności

- **Autoryzacja użytkowników** – system rejestracji i logowania (Django Authentication),
- **Prywatne historie** – każdy użytkownik widzi tylko swoje analizy,
- **Eksport wyników** – możliwość pobrania analizy w PDF lub wysłania mailem,
- **Integracja z klasyfikacją ICD-10** – przypisywanie kodów chorób wg standardu WHO,
- **Integracja ze SNOMED CT** – użycie standardowej terminologii medycznej,
- **Wizualizacja pewności** – wykresy słupkowe lub kołowe dla diagnoz,
- **Przypomnienia** – powiadomienia o konieczności wizyty u lekarza,
- **Historia objawów** – śledzenie zmian objawów w czasie.

9.4.2 Technologie

- **PostgreSQL/MySQL** – zamiana SQLite na produkcyjną bazę danych,
- **Redis** – cache'owanie wyników do optymalizacji wydajności,
- **Celery** – zadania asynchroniczne dla długich analiz,
- **Docker Compose** – konteneryzacja dla łatwiejszego wdrożenia,
- **CI/CD** – automatyczne testy i wdrożenia (GitHub Actions),
- **Monitoring** – Sentry dla śledzenia błędów, Prometheus dla metryk.

9.4.3 Wdrożenie

- **Wdrożenie w chmurze** – Render, Railway, AWS, Azure,
- **CDN** – Cloudflare dla szybszego ładowania frontendu,
- **Load balancing** – obsługa większego ruchu,
- **HTTPS** – certyfikat SSL dla bezpiecznej komunikacji.

9.5 Wartość edukacyjna

Projekt **SymptoCheck** ma znaczącą wartość edukacyjną:

- demonstracja praktycznego zastosowania AI w medycynie,
- przykład integracji REST API,
- wzorzec architektury dwuwarstwowej aplikacji webowej,
- implementacja nowoczesnego stosu technologicznego (React + Django),
- praktyka w prompt engineeringu dla LLM.

9.6 Podsumowanie końcowe

Aplikacja **SymptoCheck** spełnia założenia projektu i stanowi działające rozwiązanie do wspomagania wstępnej analizy objawów chorobowych. Wykorzystanie modelu AI *Gemini 2.5 Flash* pozwala na generowanie sensownych wyników w krótkim czasie, a przejrzysty interfejs użytkownika ułatwia korzystanie z systemu.

Należy jednak pamiętać, że **SymptoCheck nie zastępuje profesjonalnej konsultacji lekarskiej** i służy wyłącznie celom informacyjnym. System może być przydatny jako narzędzie edukacyjne lub punkt wyjścia do dalszego rozwoju w kierunku bardziej zaawansowanych systemów wspierających diagnostykę medyczną.

10 Wyzwania i doświadczenia projektowe

10.1 Wyzwania techniczne

10.1.1 Konfiguracja klucza API Google Gemini

Jednym z pierwszych wyzwań była prawidłowa konfiguracja dostępu do API Google Gemini:

- **Problem:** Generowanie i aktywacja klucza API w Google AI Studio.
- **Rozwiązanie:** Utworzenie konta Google, wygenerowanie klucza na stronie MakerSuite i dodanie go do pliku `.env`.
- **Lekcja:** Bezpieczne przechowywanie kluczy API w zmiennych środowiskowych jest kluczowe dla bezpieczeństwa aplikacji.

10.1.2 Synchronizacja CORS pomiędzy frontendem i backendem

Komunikacja cross-origin wymagała odpowiedniej konfiguracji:

- **Problem:** Frontend (localhost:5173) nie mógł wysyłać żądań do backendu (localhost:8000) z powodu polityki CORS.
- **Rozwiązanie:** Instalacja pakietu `django-cors-headers` i dodanie konfiguracji:

```
INSTALLED_APPS = [..., 'corsheaders', ...]
MIDDLEWARE = ['corsheaders.middleware.CorsMiddleware', ...]
CORS_ALLOWED_ORIGINS = ["http://localhost:5173"]
```

- **Lekcja:** Polityka CORS jest istotnym mechanizmem bezpieczeństwa, który wymaga świadomej konfiguracji podczas rozwoju aplikacji dwuwarstwowych.

10.1.3 Parsowanie odpowiedzi API Gemini

Model zwracał wyniki w formacie Markdown z osadzonym JSON, co wymagało dodatkowego przetwarzania:

- **Problem:** Wyodrębnienie JSON z prawdopodobieństwami z tekstu Markdown.
- **Rozwiązanie:** Użycie wyrażenia regularnego do wyszukania bloku ```` ```json...``` ````:

```
import re, json
m = re.search(r"```json\s*(\{.*?\})\s*```", text, re.S)
if m:
    confidence = json.loads(m.group(1))
```

- **Lekcja:** Prompt engineering jest kluczowy – odpowiednia instrukcja dla modelu AI pozwala otrzymać strukturalizowane dane.

10.1.4 Walidacja danych i obsługa wyjątków SDK

API Gemini czasami zwracało błędy (limity, nieprawidłowe żądania):

- **Problem:** Nieoczekiwane wyjątki powodowały awarie backendu.
- **Rozwiązanie:** Implementacja obsługi wyjątków w każdym endpoincie:

```
try:
    result = model.generate_content(prompt)
except Exception as e:
    return Response({"error": str(e)}, status=500)
```

- **Lekcja:** Zawsze należy przewidywać błędy zewnętrznych API i implementować mechanizmy graceful degradation.

10.1.5 Testowanie integracji React + Django

Testowanie komunikacji między frontendem a backendem wymagało uruchomienia obu serwerów jednocześnie:

- **Problem:** Trudności w debugowaniu błędów komunikacyjnych (nieprawidłowe nagłówki, formaty danych).
- **Rozwiązanie:** Użycie narzędzi deweloperskich przeglądarki (Network tab) oraz logowanie w konsoli backendu.
- **Lekcja:** Dobre zrozumienie protokołu HTTP i umiejętność analizowania ruchu sieciowego są niezbędne podczas rozwoju aplikacji webowych.

10.2 Wyzwania projektowe

10.2.1 Prompt engineering

Stworzenie promptu generującego pożądane wyniki wymagało iteracyjnego podejścia:

- **Iteracja 1:** Prosty prompt "Podaj diagnozę dla objawów: {symptoms}" – zbyt ogólne wyniki.
- **Iteracja 2:** Dodanie instrukcji formatowania (Markdown) – lepsze wyniki, ale brak struktury.
- **Iteracja 3:** Instrukcja zwracania JSON z prawdopodobieństwami – finalna wersja.

10.2.2 Design interfejsu użytkownika

Projekt interfejsu wymagał balansu między estetyką a funkcjonalnością:

- wykorzystanie TailwindCSS pozwoliło na szybkie prototypowanie,

- inspiracja popularnymi aplikacjami medycznymi (kolorystyka, ikony),
- testowanie z użytkownikami końcowymi w celu weryfikacji intuicyjności.

10.2.3 Zarządzanie stanem w React

Zarządzanie stanem rozmowy w czacie wymagało przemyślanej architektury:

- zastosowanie hook'a `useState` do lokalnego stanu wiadomości,
- przekazywanie historii w każdym żądaniu do backendu,
- rozważenie użycia Redux lub Context API dla bardziej złożonych scenariuszy.

10.3 Nabyte umiejętności

W trakcie realizacji projektu zespół zdobył następujące kompetencje:

- **Integracja API AI** – praktyczne wykorzystanie modeli LLM w aplikacji webowej,
- **REST API** – projektowanie i implementacja endpointów RESTful,
- **React** – budowa interaktywnego interfejsu użytkownika w architekturze komponentowej,
- **Django REST Framework** – tworzenie backendu dla aplikacji webowych,
- **Prompt engineering** – formułowanie instrukcji dla modeli AI,
- **Debugowanie** – identyfikacja i rozwiązywanie problemów w aplikacji dwuwarstwowej,
- **Git** – współpraca zespołowa z użyciem kontroli wersji.

10.4 Refleksje zespołu

Projekt **SymptoCheck** był cennym doświadczeniem edukacyjnym:

- praktyczne zastosowanie wiedzy z zakresu informatyki medycznej,
- poznanie nowoczesnych narzędzi AI w kontekście medycznym,
- zrozumienie wyzwań związanych z tworzeniem aplikacji webowych,
- doświadczenie w pracy z dokumentacją zewnętrznych API,
- świadomość ograniczeń i możliwości modeli AI w diagnostyce.

Największą satysfakcję sprawiła możliwość stworzenia działającego systemu, który może być rozwijany w przyszłości i potencjalnie wykorzystany w rzeczywistych scenariuszach edukacyjnych lub medycznych (po odpowiedniej walidacji i certyfikacji).

11 Bibliografia i źródła

11.1 Dokumentacja techniczna

1. **Django Documentation** – `https://docs.djangoproject.com/`
2. **Django REST Framework** – `https://www.django-rest-framework.org/`
3. **React Documentation** – `https://react.dev/`
4. **Vite Documentation** – `https://vitejs.dev/`
5. **TailwindCSS** – `https://tailwindcss.com/docs`
6. **Google Gemini API** – `https://ai.google.dev/docs`
7. **Python google-generativeai** – `https://github.com/google/generative-ai-python`

11.2 Artykuły i publikacje

1. **Large Language Models in Healthcare** – przegląd zastosowań LLM w medycynie
2. **Prompt Engineering Guide** – najlepsze praktyki tworzenia promptów dla AI
3. **REST API Design Best Practices** – standardy projektowania API RESTful
4. **Medical AI Ethics** – etyczne aspekty wykorzystania AI w diagnostyce

11.3 Narzędzia i platformy

1. **Visual Studio Code** – `https://code.visualstudio.com/`
2. **Git** – `https://git-scm.com/`
3. **npm** – `https://www.npmjs.com/`
4. **pip** – `https://pip.pypa.io/`
5. **Google AI Studio (MakerSuite)** – `https://makersuite.google.com/`

11.4 Standardy medyczne

1. **ICD-10** – International Classification of Diseases, WHO
2. **SNOMED CT** – Systematized Nomenclature of Medicine Clinical Terms
3. **RODO/GDPR** – Rozporządzenie o ochronie danych osobowych
4. **HL7 FHIR** – Standard wymiany danych medycznych

HL7 FHIR – Standard wymiany danych medycznych

Załącznik A: Przykładowe wyniki działania systemu

A.1. Przykład analizy objawów

Wejście użytkownika:

"Mam silny ból głowy, gorączkę około 38.5°C, kaszel i ogólne osłabienie. Objawy trwają od 2 dni."

Odpowiedź systemu (fragment):

Grypa (Influenza)
Infekcja wirusowa górnych dróg oddechowych charakteryzująca się nagłym początkiem, wysoką gorączką, bólami mięśni i głowy.
Sugerowany specjalista: **Lekarz rodzinny**

COVID-19
Choroba zakaźna wywołana przez koronawirusa SARS-CoV-2, objawy obejmują gorączkę, kaszel, zmęczenie.
Sugerowany specjalista: **Lekarz rodzinny / Internista**

Prawdopodobieństwa:

- Grypa: 75%
- COVID-19: 65%
- Przeziębienie: 45%
- Zapalenie oskrzeli: 40%

A.2. Przykład rozmowy z asystentem

Użytkownik: "Mam silny ból brzucha po prawej stronie."
Asystent: "Ból brzucha po prawej stronie może wskazywać na różne schorzenia. Czy ból jest ostry czy raczej tępy? Czy towarzyszy mu gorączka lub nudności?"

Użytkownik: "Ból jest ostry i mam nudności."
Asystent: "Ostry ból po prawej stronie brzucha z nudnościami może sugerować zapalenie wyrostka robaczkowego. To jest pilny przypadek - zalecam natychmiastową konsultację z lekarzem lub wizytę na SOR."

Załącznik B: Kluczowe fragmenty kodu

B.1. Model Django (backend/core/models.py)

```python
from django.db import models

class Analysis(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    symptoms = models.TextField()
    result_md = models.TextField()
    confidence_json = models.JSONField(null=True, blank=True)

    class Meta:
        ordering = ['-created_at']
        verbose_name = 'Analiza'
        verbose_name_plural = 'Analizy'

    def __str__(self):
        return f"Analysis #{self.id} @ {self.created_at:%Y-%m-%d %H:%M}"
```

B.2. Serializator Django (backend/core/serializers.py)

```python
from rest_framework import serializers
from .models import Analysis

class AnalysisSerializer(serializers.ModelSerializer):
    class Meta:
        model = Analysis
        fields = ['id', 'created_at', 'symptoms',
                  'result_md', 'confidence_json']
        read_only_fields = ['id', 'created_at']
```

B.3. Routing backendu (backend/backend/urls.py)

```python
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('core.urls')),
]
```

B.4. Komponent React (frontend/src/App.jsx - fragment)

```javascript
import React, { useState } from "react";
import SymptomForm from "./SymptomForm";
```

```
import HistoryList from "./HistoryList";
import ChatAssistant from "./ChatAssistant";

export default function App() {
  const [tab, setTab] = useState("form");

  return (
    <div className="min-h-screen bg-blue-50">
      <nav className="bg-white shadow-sm">
        <button onClick={() => setTab("form")}>
          Analiza objaw w
        </button>
        <button onClick={() => setTab("history")}>
          Historia
        </button>
        <button onClick={() => setTab("chat")}>
          Czat
        </button>
      </nav>

      <main>
        {tab === "form" && <SymptomForm />}
        {tab === "history" && <HistoryList />}
        {tab === "chat" && <ChatAssistant />}
      </main>
    </div>
  );
}
```

Załącznik C: Instrukcja instalacji krok po kroku

C.1. Wymagania wstępne

Upewnij się, że masz zainstalowane:

- Python 3.10 lub nowszy
- Node.js 18 lub nowszy
- Git
- Edytor kodu (np. VS Code)

C.2. Klonowanie repozytorium

```
git clone https://github.com/username/SymptoCheck.git
cd SymptoCheck
```

C.3. Konfiguracja backendu

```
# Przejdź do folderu backendu
cd backend

# Utwórz środowisko wirtualne
python -m venv .venv

# Aktywuj środowisko (Windows)
.venv\Scripts\activate

# Aktywuj środowisko (Linux/Mac)
source .venv/bin/activate

# Zainstaluj zależności
pip install -r requirements.txt

# Utwórz plik .env
echo "GEMINI_API_KEY=your_key_here" > .env
echo "DEBUG=True" >> .env
echo "SECRET_KEY=your_secret_key" >> .env

# Wykonaj migracje
python manage.py makemigrations
python manage.py migrate
```

```
# Uruchom serwer
python manage.py runserver
```

C.4. Konfiguracja frontendu

Otwórz nowe okno terminala:

```
# Przejdź do folderu frontendu
cd frontend

# Zainstaluj zależności
npm install

# Utwórz plik .env
echo "VITE_API_BASE=http://localhost:8000/api" > .env

# Uruchom serwer deweloperski
npm run dev
```

C.5. Weryfikacja działania

1. Otwórz przeglądarkę i wejdź na `http://localhost:5173`
2. Wpisz przykładowe objawy w formularzu
3. Sprawdź, czy otrzymujesz wyniki analizy
4. Przetestuj zakładki "Historia" i "Czat"

C.6. Rozwiązywanie problemów

Problem: Backend nie uruchamia się
Rozwiązanie: Sprawdź, czy wszystkie zależności są zainstalowane: `pip list`
Problem: Frontend nie łączy się z backendem
Rozwiązanie: Sprawdź konfigurację CORS w `settings.py` i zmienną `VITE_API_BASE`
Problem: Błąd "Invalid API key"
Rozwiązanie: Upewnij się, że klucz API w pliku `.env` jest prawidłowy