
POLITECHNIKA ŚLĄSKA

Wydział Inżynierii Materiałowej

TAIiB projekt

TEMAT:

# Przychodnia

Uwagi prowadzącego:

Data przyjęcia:

Podpis prowadzącego:

Imię i nazwisko:
Piotr Bistyga
Jakub Czulak
Maciej Broś
Kacper Kania
Marek Wójcik

## Założenia funkcjonalne projektu:

- zarządzanie personelem
- obsługa wizyt i badań
- generowanie recepty
- historia pacjenta i tworzenie raportów
- logowanie w zależności od funkcji

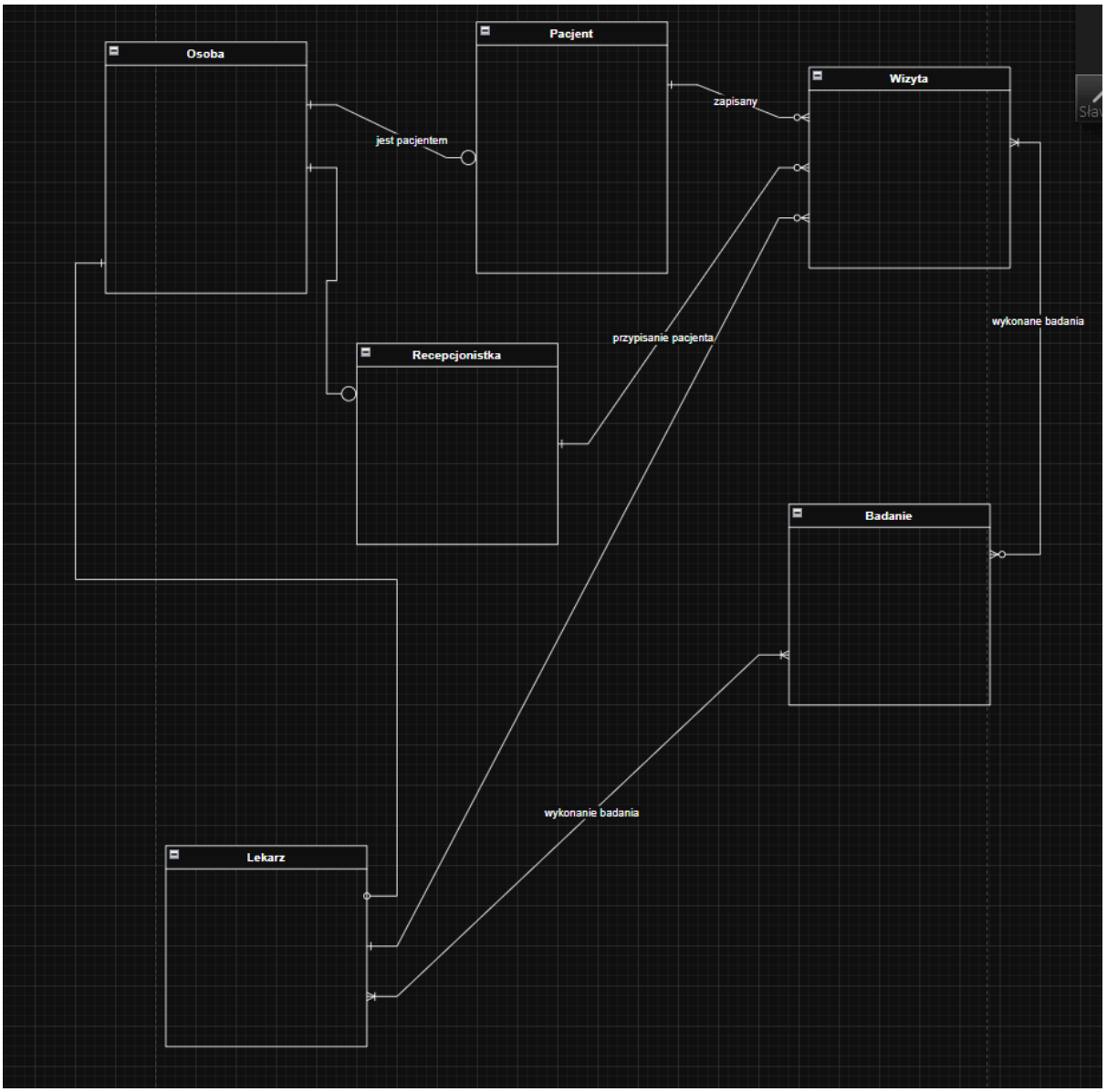
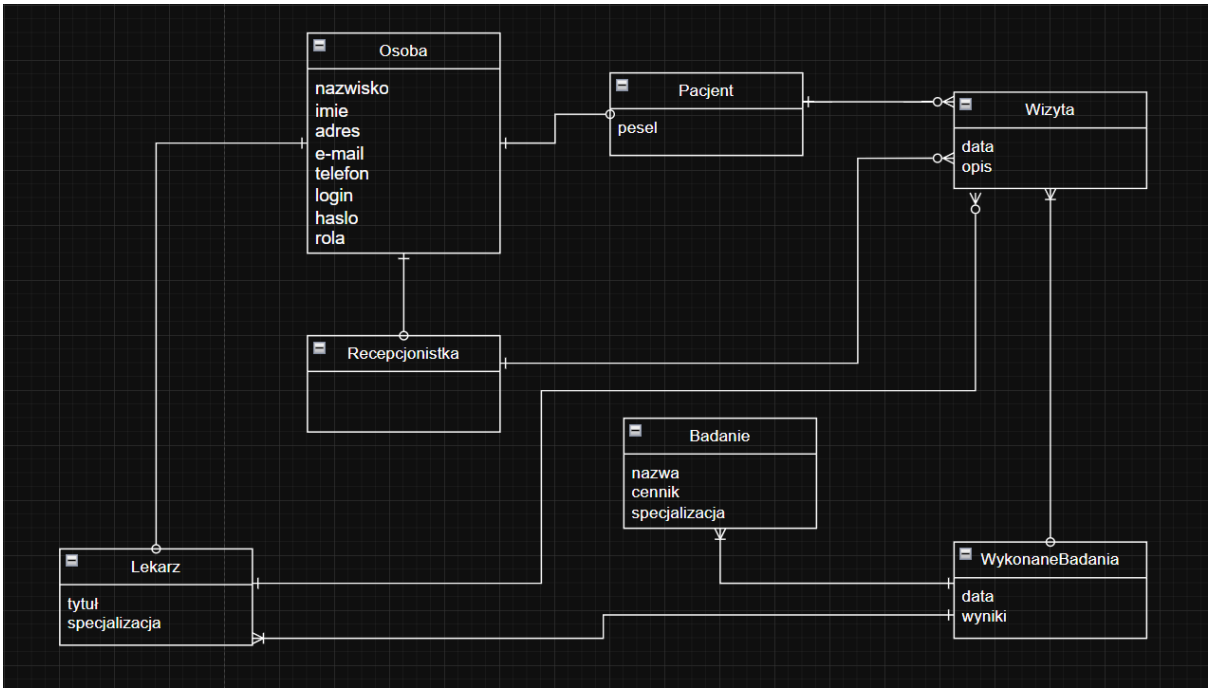


Diagram CDM:




Poprawiony :

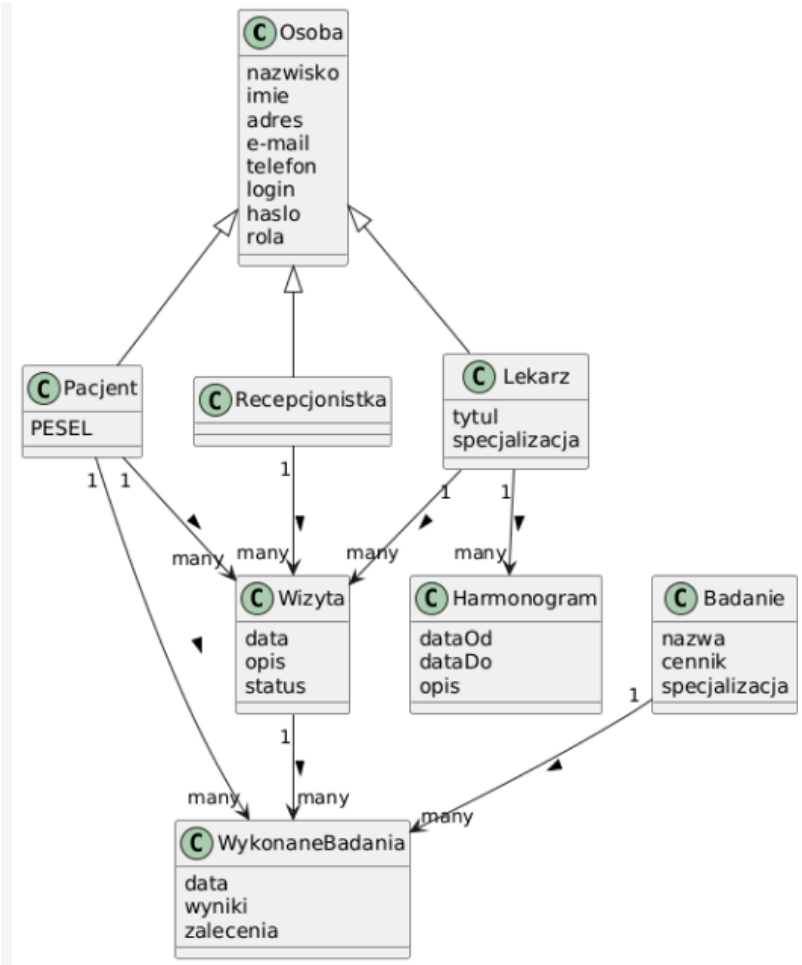
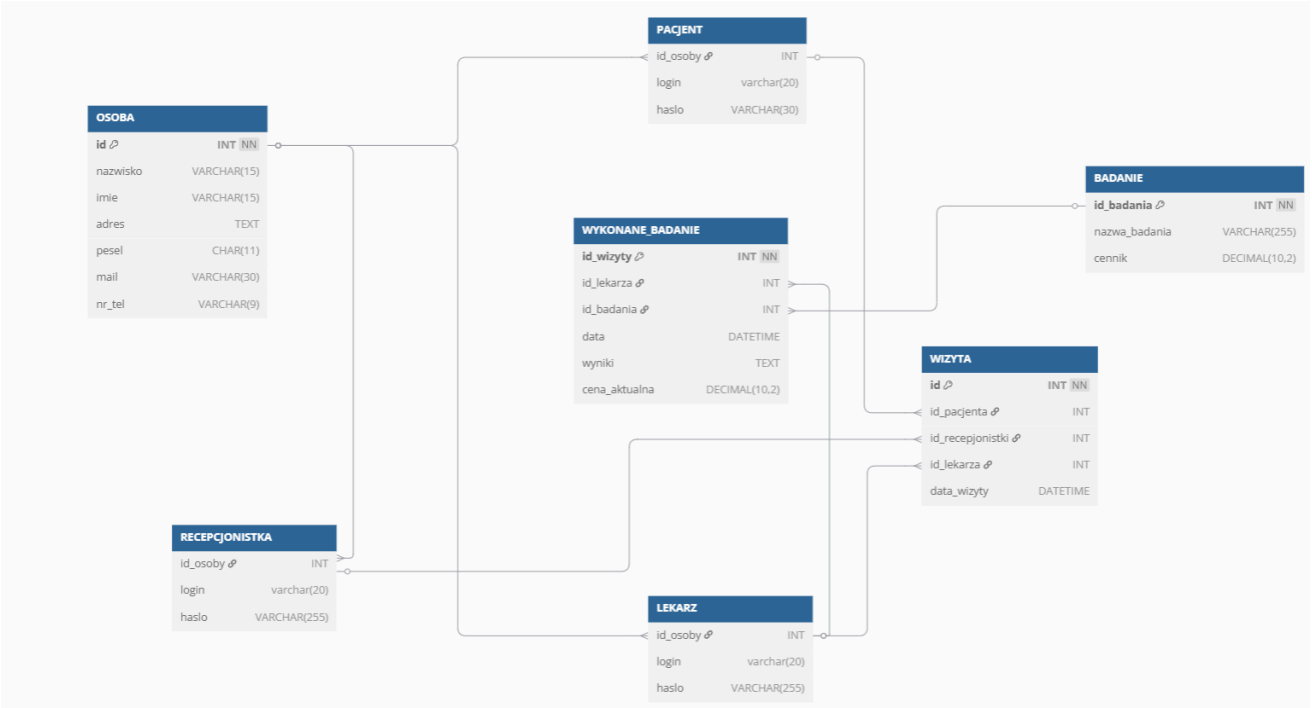



Diagram PDM:


PACJENT
id_osoby INT
login varchar(20)
haslo VARCHAR(30)
OSOBA
id INT NN
nazwisko VARCHAR(15)
imie VARCHAR(15)
adres TEXT
pesel CHAR(11)
mail VARCHAR(30)
nr_tel VARCHAR(9)
BADANIE
id_badania INT NN
nazwa_badania VARCHAR(255)
cennik DECIMAL(10,2)
WYKONANE_BADANIE
id_wizyty INT NN
id_lekarza INT
id_badania INT
data DATETIME
wyniki TEXT
cena_aktualna DECIMAL(10,2)
WIZYTA
id INT NN
id_pacjenta INT
id_recepjonistki INT
id_lekarza INT
data_wizyty DATETIME
RECEPCJONISTKA
id_osoby INT
login varchar(20)
haslo VARCHAR(255)
LEKARZ
id_osoby INT
login varchar(20)
haslo VARCHAR(255)

Poprawiony:

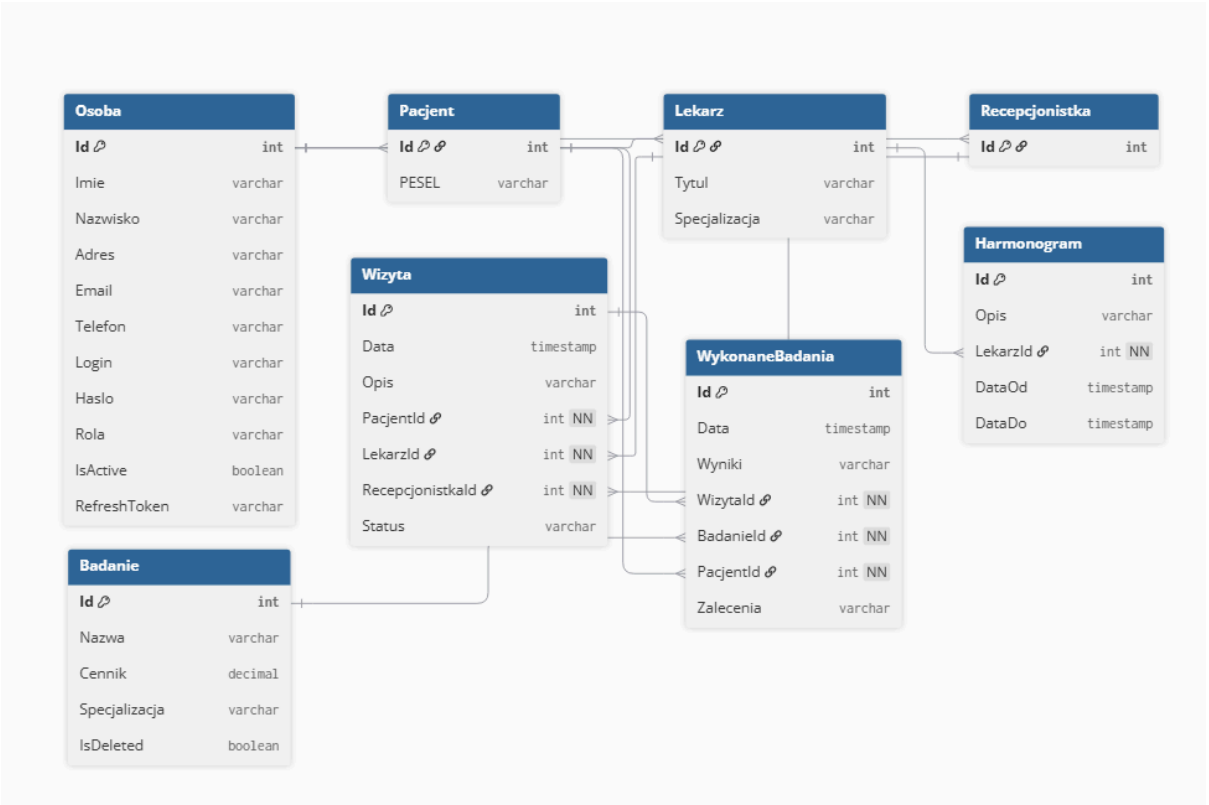

Osoba
Id int
Imie varchar
Nazwisko varchar
Adres varchar
Email varchar
Telefon varchar
Login varchar
Haslo varchar
Rola varchar
IsActive boolean
RefreshToken varchar
Pacjent
Id int
PESEL varchar
Lekarz
Id int
Tytul varchar
Specjalizacja varchar
Recepcjonistka
Id int
Harmonogram
Id int
Opis varchar
LekarzId int NN
DataOd timestamp
DataDo timestamp
Wizyta
Id int
Data timestamp
Opis varchar
PacjentId int NN
LekarzId int NN
RecepcjonistkaId int NN
Status varchar
WykonaneBadania
Id int
Data timestamp
Wyniki varchar
WizytaId int NN
BadanieId int NN
PacjentId int NN
Zalecenia varchar
Badanie
Id int
Nazwa varchar
Cennik decimal
Specjalizacja varchar
IsDeleted boolean

Diagram przypadków użycia:

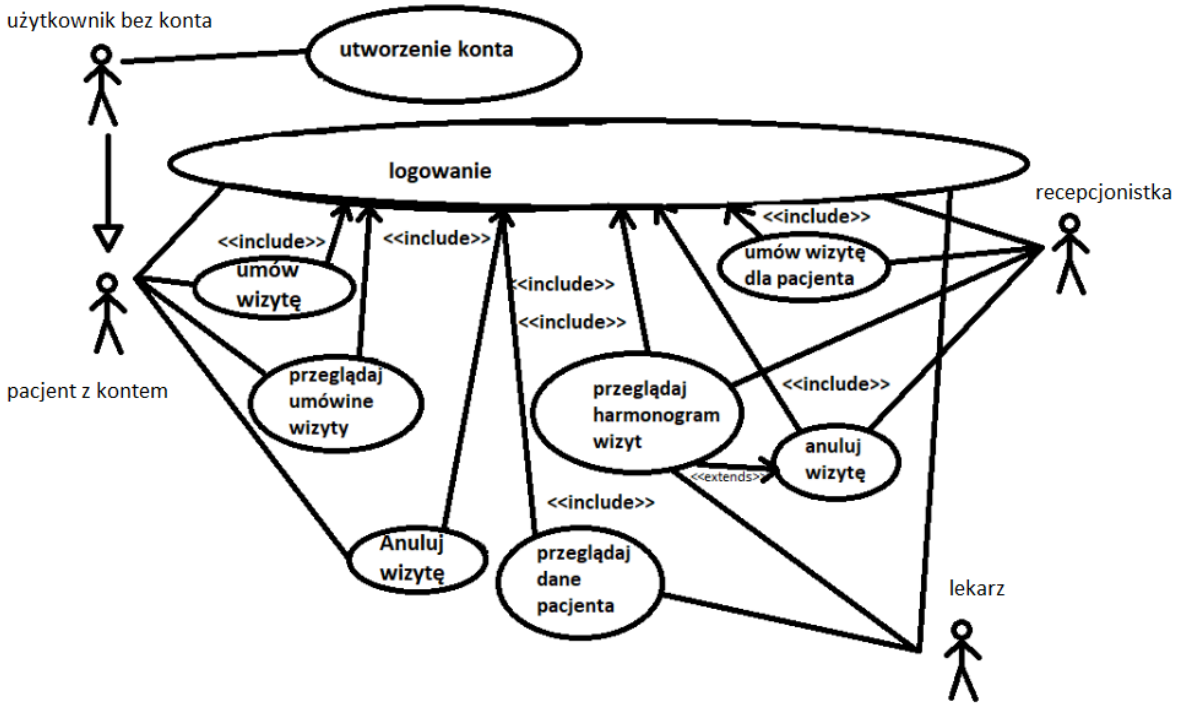
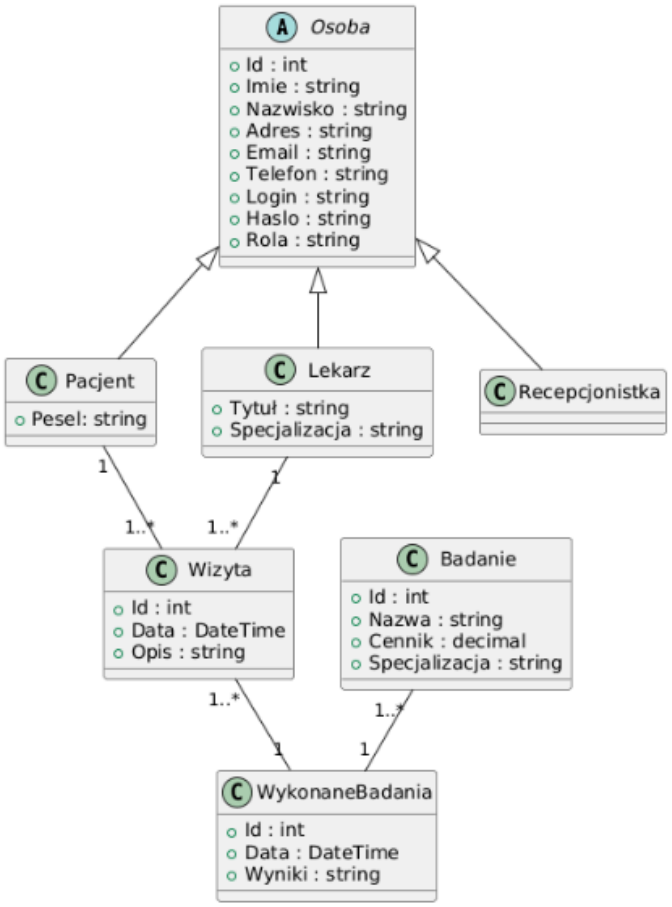
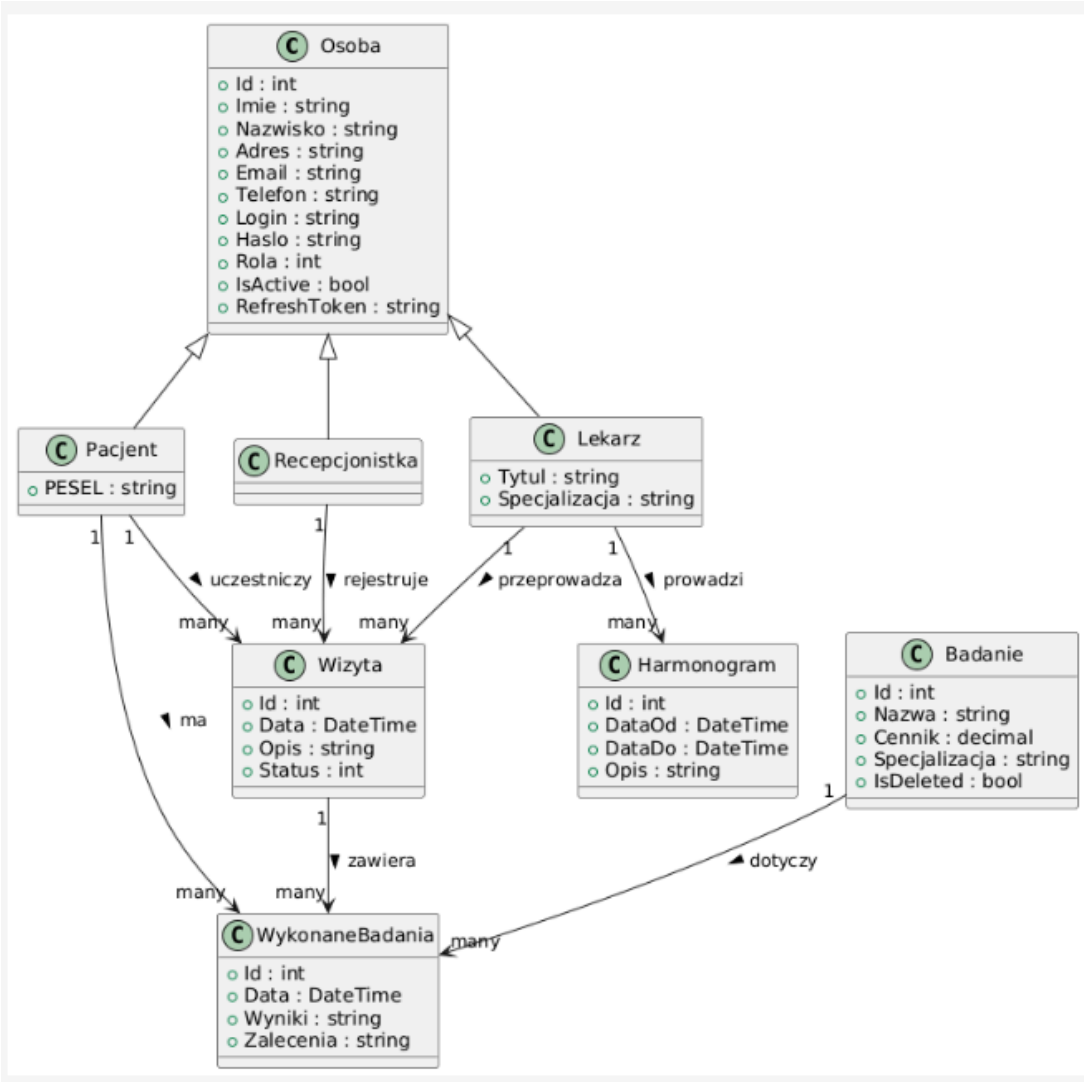



Diagram klas modelu:




Poprawiony:




## Opis działania aplikacji:

Aplikacja Przychodni to zestaw mikroserwisów współpracujących ze sobą przez HTTP/REST. Każdy mikroserwis realizuje jedną domenę odpowiedzialności. Wewnątrz każdego serwisu zachowana jest warstwowa separacja: Controller → BLL (Business Logic Layer) → DAL (Data Access Layer). Główne funkcje aplikacji:

1. Autoryzacja i logowanie:

    - Użytkownik loguje się. SerwisAutoryzacji weryfikuje dane i zwraca token JWT z rolami/zakresem uprawnień.

    - Token przesyłany jest w nagłówku Authorization: Bearer przy każdym żądaniu. Kontrolery w serwisach walidują token i autoryzują dostęp do endpointów.

2. Zarządzanie personelem:
    - CRUD pracowników.
    - Przypisanie lekarza do badania.
3. Obsługa wizyt i badań:
    - Rezerwacja wizyt na dane badania u lekarzy.
    - Generowanie recepty.
4. Generowanie recepty:
    - System generuje plik PDF, na którym wypisana jest recepta pacjenta.
5. Logowanie w zależności od funkcji:
    - Każda osoba ma role na stronie i role dają różne możliwości. Wyróżniamy takie role jak: Pacjent, Recepcjonistka i Lekarz.

# Technologie użyte w Backendzie:

### .NET 8 — jądro i punkt wejścia

Projekt działa na **.NET 8**, co daje nam nowoczesny runtime, najnowsze funkcje języka C# i minimal hosting model. Program.cs jest głównym punktem startowym aplikacji — tutaj jest konfiguracja DI, middleware, rejestracja serwisów BLL/DAL i uruchamianie aplikacji. Jest to główny program całego projektu.

**Struktura projektów (Controller / BLL / DAL / interfejsy / Models):**

- **Controller (API)** — kontrolery ASP.NET Core przyjmują żądania HTTP i zwracają odpowiedzi. Zawierają minimalną logikę (mapowanie, walidacja wejścia), delegując rzeczywistą logikę do BLL.

- **BLL (Business Logic Layer)** — tu siedzi logika biznesowa: reguły walidacji, orchestracja operacji (np. tworzenie wizyty + powiadomień), transformacje DTO ↔ model domenowy.

- **DAL (Data Access Layer)** — implementacja dostępu do bazy (EF Core), repozytoria, migracje, seedy.

- **Interfejsy (IDAL, IBLL)** — kontrakty, które umożliwiają łatwe mockowanie w testach i separację zależności.

- **Models** — wspólne modele/domenowe klasy i DTO używane w całym projekcie.

  Taka separacja ułatwia testowanie, czytelność kodu i wieloosobową pracę nad systemem.

**Autoryzacja i uwierzytelnianie — JWT**

Używamy **JWT** z pakietami Microsoft.AspNetCore.Authentication.JwtBearer i System.IdentityModel.Tokens.Jwt. Mechanizm: użytkownik loguje się → serwis autoryzacji wydaje token JWT zawierający rolę i claims → klient przesyła token w nagłówku Authorization: Bearer. Kontrolery i middleware walidują token.

**Dokumentacja API — Swashbuckle/Swagger (OpenAPI)**

Swashbuckle generuje **specyfikację OpenAPI** i **Swagger UI**, czyli interaktywną dokumentację. Dzięki temu każdy endpoint jest automatycznie opisany i możemy wywołać taki endpoint z poziomu przeglądarki.

**Testy — xUnit, Moq, Microsoft Fakes**

Testy jednostkowe i integracyjne napisaliśmy przy pomocy **xUnit** i xunit.runner.visualstudio. Logikę BLL testujemy jednostkowo, **mockując DAL** przy pomocy **Moq** . Gdy trzeba podmienić trudne do mockowania elementy , używamy Microsoft.QualityTools.Testing.Fakes .

**DAL — Entity Framework Core i migracje**

Warstwa DAL korzysta z **Entity Framework Core** do mapowania klas na tabele i zarządzania zapytaniami. Schemat bazy wersjonujemy poprzez **EF Migrations**: tworzymy migrację (dotnet ef migrations add), commitujemy ją i stosujemy (dotnet ef database update) w środowisku. Dodatkowo przygotowujemy **seedy** (dane początkowe: role, konta demo), które uruchamiają się przy starcie lub w osobnym skrypcie migracyjnym.

**Komunikacja — REST + HTTPS**

Komunikacja z klientem odbywa się przez **REST** z użyciem **HTTPS**. Endpointy zwracają JSON. kontrolery mapują żądania do DTO i przekazują je do BLL. HTTPS zapewnia poufność i integralność danych

.
**Przepływ żądania: Controller → BLL → DAL → Client**

Typowy scenariusz:

1. Klient wysyła żądanie HTTP do endpointu (np. POST).

2. **Controller** przyjmuje żądanie, mapuje payload do DTO i wykonuje wstępną walidację.

3. Controller wywołuje metodę z **BLL**, która zawiera logikę biznesową.

4. BLL używa **repozytorium z DAL** do zapisania danych w DB w jednej transakcji.

5. Po zakończeniu operacji BLL zwraca wynik do kontrolera, który mapuje go do response i odsyła klientowi.

# Technologie użyte we Frontendzie:

## Framework i język

Angular 16 — framework do budowy aplikacji jednostronicowych (SPA).
TypeScript — główny język programowania zapewniający statyczne typowanie i lepszą czytelność kodu.
HTML + CSS — do definiowania struktury oraz stylów interfejsu użytkownika.

## Struktura projektu po stronie frontendu:

Struktura projektu frontendowego została podzielona w sposób zapewniający czytelność i łatwą rozbudowę aplikacji. Najważniejszą część stanowią komponenty, które znajdują się w katalogu src/app/components. Każdy komponent odpowiada za

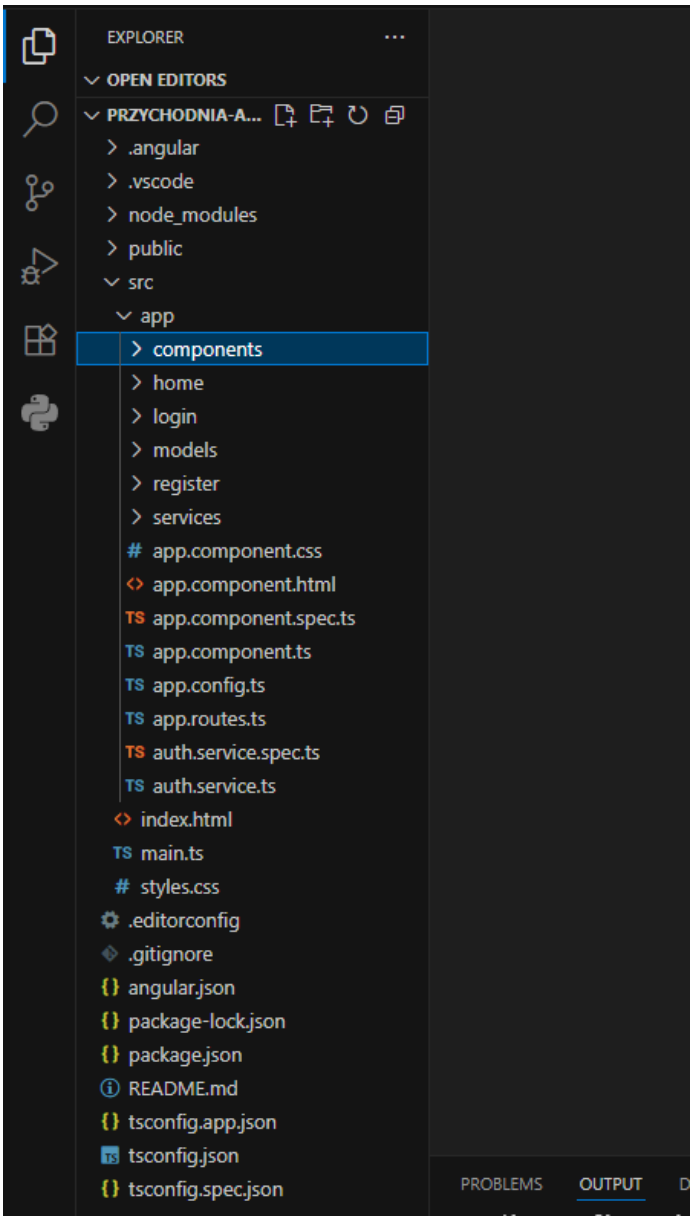
wyświetlanie określonego widoku oraz obsługę logiki z nim związanej, np. zarządzanie wizytami, harmonogramem lekarza, badaniami czy obsługą recepcji. Każdy z nich składa się z trzech podstawowych plików: TypeScript (.ts) z logiką, HTML (.html) z widokiem oraz CSS (.css) ze stylami.

Drugim ważnym elementem są serwisy, umieszczone w katalogu src/app/services. To one odpowiadają za komunikację z backendem przy pomocy wywołań REST API. Dzięki temu komponenty mogą korzystać z gotowych metod do pobierania czy wysyłania danych (np. serwisy wizyt, badań czy logowania).

Dane przesyłane pomiędzy frontendem a backendem zostały opisane w katalogu models (src/app/models). Znajdują się tam definicje typów i struktur obiektów, takich jak wizyta czy badanie, co pozwala zachować spójność danych w całej aplikacji.

Nawigacja pomiędzy poszczególnymi ekranami realizowana jest poprzez routing, którego konfiguracja została zapisana w pliku app.routes.ts. Z kolei pliki app.config.ts i main.ts pełnią rolę głównych elementów startowych aplikacji – odpowiadają za jej inicjalizację i podstawową konfigurację.

Zrzut ekranu - **struktura katalogów oraz plików projektu frontendowego**:




## Formularze i dane:

- FormsModule — do obsługi formularzy (np. logowanie, rejestracja, umawianie wizyty).
- NgModel — dwukierunkowe wiązanie danych pomiędzy modelem a widokiem.
- Walidacja formularzy wykonywana po stronie frontendu i backendu.

### Autoryzacja i bezpieczeństwo:

- AuthService — serwis obsługujący logowanie i rejestrację użytkowników.
- Tokeny JWT otrzymywane z backendu są wykorzystywane do autentykacji i autoryzacji użytkowników.

### Stylizacja:

- CSS dla komponentu — każdy komponent posiada własny plik ze stylami (`.component.css`).
- Spójny layout aplikacji oparty na responsywnych tabelach i formularzach.

## Baza danych:

W projekcie używam relacyjnej bazy danych zarządzanej za pomocą **Entity Framework Core**. Pliki migracji i snapshot modelu wygenerowane przez EF Core definiują schemat bazy oraz reguły integralności.

#### Technologia i sposób pracy z migracjami

Do wersjonowania schematu wykorzystuję mechanizm migracji EF Core. Główne kroki:

- modyfikuję modele klas w projekcie (Models),
- generuję migrację poleceniem dotnet ef migrations add <Nazwa>,
- stosuję migrację do bazy dotnet ef database update.
  EF Core tworzy pliki migracji oraz snapshot modelu, które przechowuję w repozytorium.

#### Główne tabele i relacje

Schemat bazy zawiera następujące tabele i zależności:

- **Osoby** Tabela Osoby jest użyta jako hierarchia typu table-per-hierarchy dla podtypów Pacjent, Recepcjonistka i Lekarz. W tabeli znajdują się pola wspólne (Id, Imie, Nazwisko, Adres, Email, Telefon, Login, Haslo, Rola, IsActive, RefreshToken) oraz kolumny specyficzne dla podtypów (Tytul, Specjalizacja, PESEL)

- **Badania**
  Tabela Badania przechowuje listę dostępnych badań: Id, Nazwa, Cennik , Specjalizacja, IsDeleted. Z Badania jest relacja do wykonanych badań.

- **Harmonogramy**
  Tabela Harmonogramy powiązana jest z lekarzem i przechowuje zakres dat oraz opis.

- **Wizyty**
  Tabela Wizyty zawiera informacje o wizytach: Data, Opis, PacjentId, LekarzId, RecepcjonistkaId, Status. Wszystkie klucze obce do Osoby mają ograniczenie usuwania ustawione na Restrict, co zapobiega przypadkowemu skasowaniu powiązanych wizyt.

- **WykonaneBadania**
  Tabela łączy Wizyty i Badania, przechowuje datę, wyniki i zalecenia. FK do Wizyty ma OnDelete: Cascade (usunięcie wizyty usuwa powiązane wykonane badania), FK do Badania ma OnDelete: Restrict.

**Główne decyzje projektowe widoczne w migracji**

- Identyfikatory są kolumnami autoinkrementującym.

- Typy kolumn dostosowane są do SQL Server.

- Zastosowanie TPH dla klasy Osoba upraszcza modelowanie dziedziczenia, ale powoduje, że wszystkie pola podtypów są w jednej tabeli.

- Indeks unikalny na PESEL z filtrem IS NOT NULL — PESEL może być pusty, ale jeśli jest podany, musi być unikalny.

- Sprawdzenia integralności przenoszą część walidacji z logiki aplikacji do bazy danych, co zwiększa odporność na błędy.

**Jak baza współpracuje z aplikacją**

W aplikacji posługuję się klasą DbPrzychodnia : DbContext z DbSetami odpowiadającymi tabelom. Operacje wykonywane są za pomocą zapytań LINQ; EF Core tłumaczy je na SQL i komunikuje się z serwerem SQL. W warstwie dostępu do danych stosuję repozytoria i unit of work, dzięki czemu logika biznesowa pozostaje

odseparowana od implementacji DB. Migracje są uruchamiane w środowisku deweloperskim lub w pipeline CI/CD przed wdrożeniem.

## Hosting:

Microsoft Azure to chmura obliczeniowa, czyli zbiór usług dostępnych przez internet, które pozwalają uruchamiać aplikacje, przechowywać dane i korzystać z gotowych rozwiązań bez konieczności kupowania własnych serwerów. Azure można traktować jako ogromne centrum danych, do którego mamy dostęp na żądanie.

Platforma udostępnia wiele kategorii usług, takich jak uruchamianie maszyn wirtualnych i aplikacji, przechowywanie plików, zarządzane bazy danych SQL i NoSQL, narzędzia sieciowe do łączenia systemów, mechanizmy bezpieczeństwa, a także monitoring i analitykę.

Największą zaletą korzystania z Azure jest możliwość szybkiego uruchamiania środowisk, skalowania aplikacji w górę i w dół w zależności od potrzeb, a także model płatności, w którym opłaca się tylko faktyczne zużycie zasobów. Dodatkowo Azure zapewnia wysoki poziom bezpieczeństwa, mechanizmy tworzenia kopii zapasowych oraz zgodność z międzynarodowymi standardami.

Podsumowując, Microsoft Azure to platforma chmurowa, która umożliwia szybkie i elastyczne tworzenie oraz rozwijanie nowoczesnych aplikacji i systemów informatycznych.

## Swagger:

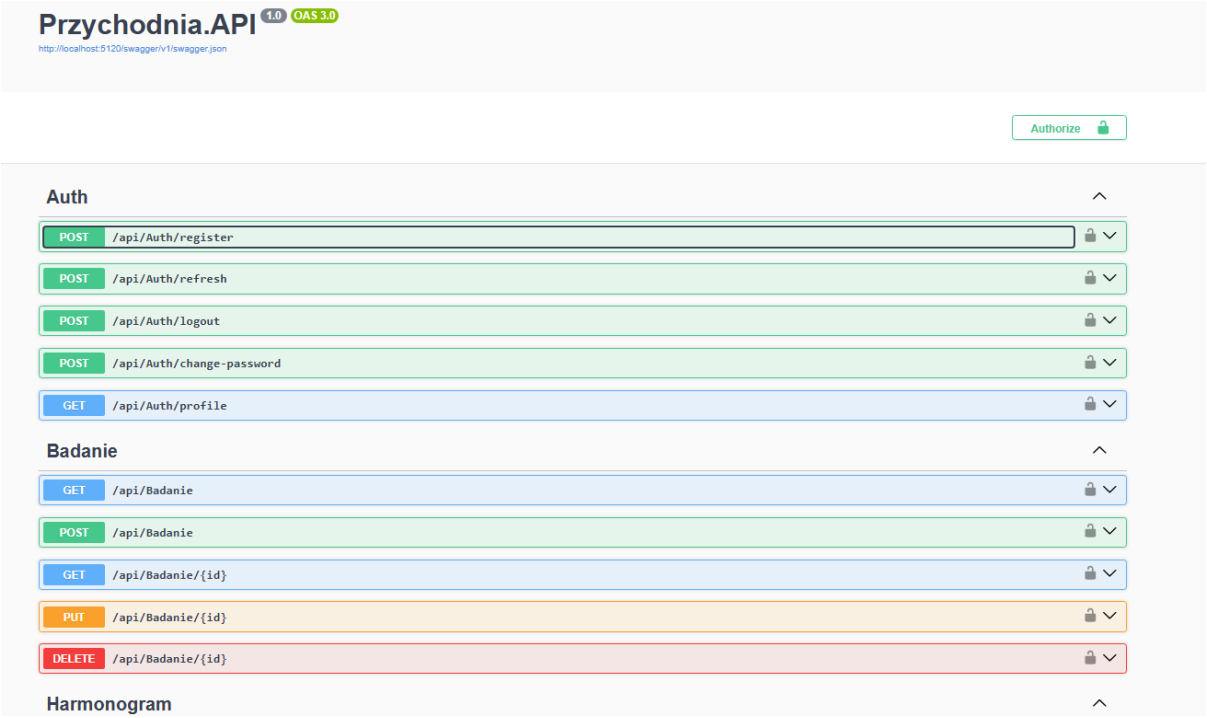
**Swagger** - to narzędzie i standardowa specyfikacja (OpenAPI), która służy do opisywania, dokumentowania i testowania REST API. Umożliwia automatyczne generowanie interaktywnej dokumentacji, zrozumiałej zarówno dla programistów, jak i maszyn, bez potrzeby zaglądania do kodu źródłowego. Pozwala to uprościć proces tworzenia, udostępniania i integracji usług sieciowych, a także oferuje możliwość interaktywnego testowania API bezpośrednio z poziomu przeglądarki.

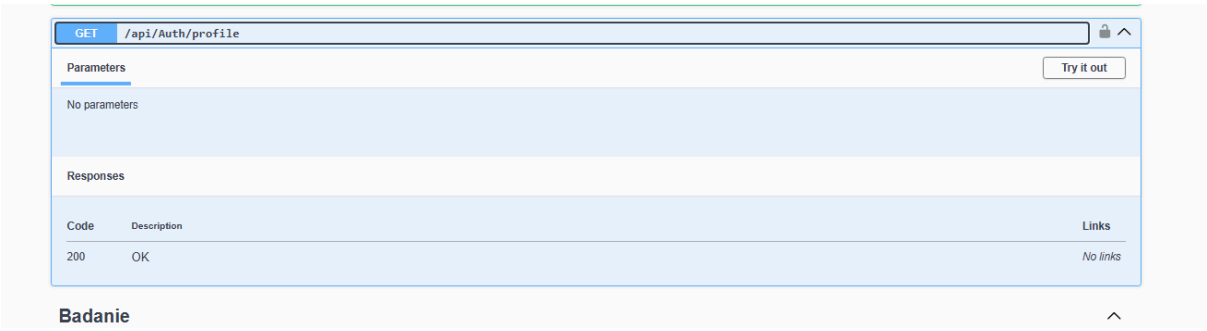
- **Interaktywnego testowania endpointów** — w Swagger UI klikasz „Try it out”, wysyłasz żądania i od razu widzisz odpowiedzi (bez Postmana).

- **Automatycznej dokumentacji API** — opis endpointów, modeli request/response i kodów odpowiedzi generuje się z kodu i jest zawsze aktualny.

- **Szybszego debugowania** — mamy od razu wszystkie endpointy wygenerowane przez niego, następnie dajemy dane i sprawdzamy jakie zwrócił nam respony.

Screeny:




# Screeny kodu:

Screeny przykładowego kodu backend:

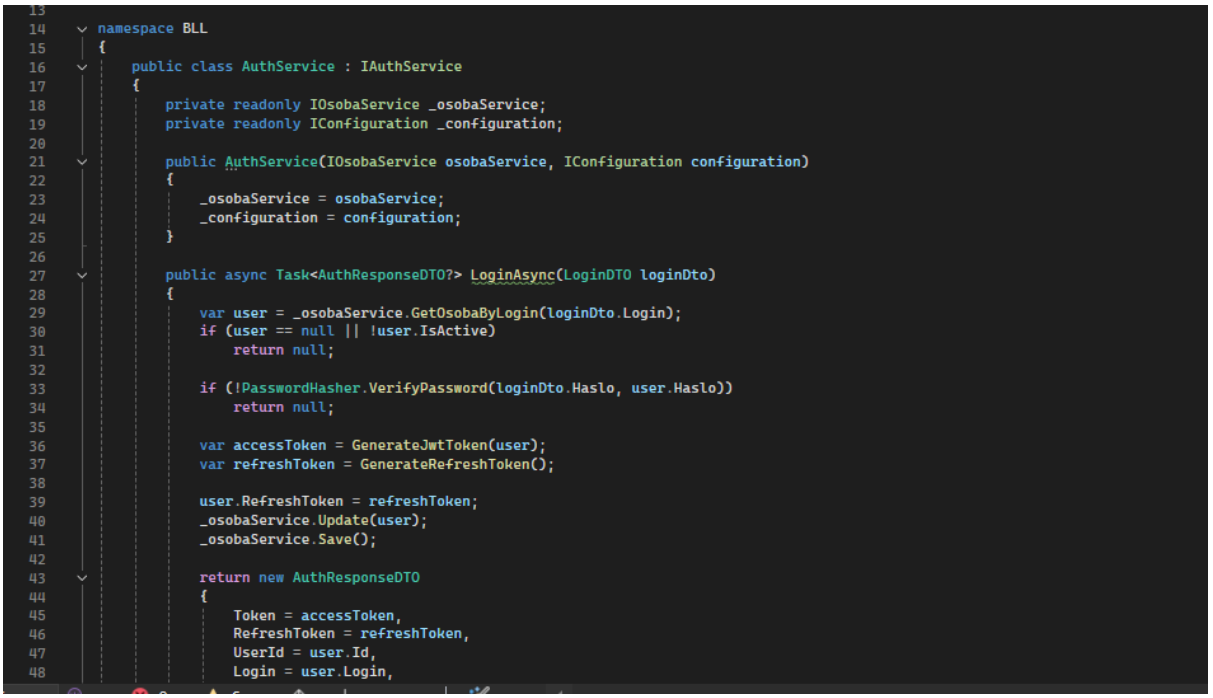
Controller

```
{
    [ApiController]
    [Route("api/[controller]")]
    public class AuthController : ControllerBase
    {
        private readonly IAuthService _authService;

        public AuthController(IAuthService authService)
        {
            _authService = authService;
        }

        [HttpPost("register")]
        public async Task<IActionResult> Register([FromBody] RegisterDTO registerDto)
        {
            if (!ModelState.IsValid)
                return BadRequest(ModelState);

            var result = await _authService.RegisterAsync(registerDto);
            if (result == null)
                return BadRequest("Rejestracja nie powiodła się. Login lub email mogą być zajęte.");

            return Ok(result);
        }

        [HttpPost("refresh")]
        public async Task<IActionResult> RefreshToken([FromBody] string refreshToken)
        {
            if (string.IsNullOrEmpty(refreshToken))
                return BadRequest("Refresh token jest wymagany");

            var result = await _authService.RefreshTokenAsync(refreshToken);
            if (result == null)
                return Unauthorized("Nieprawidłowy refresh token");
```

IBLL (interface serwisu):

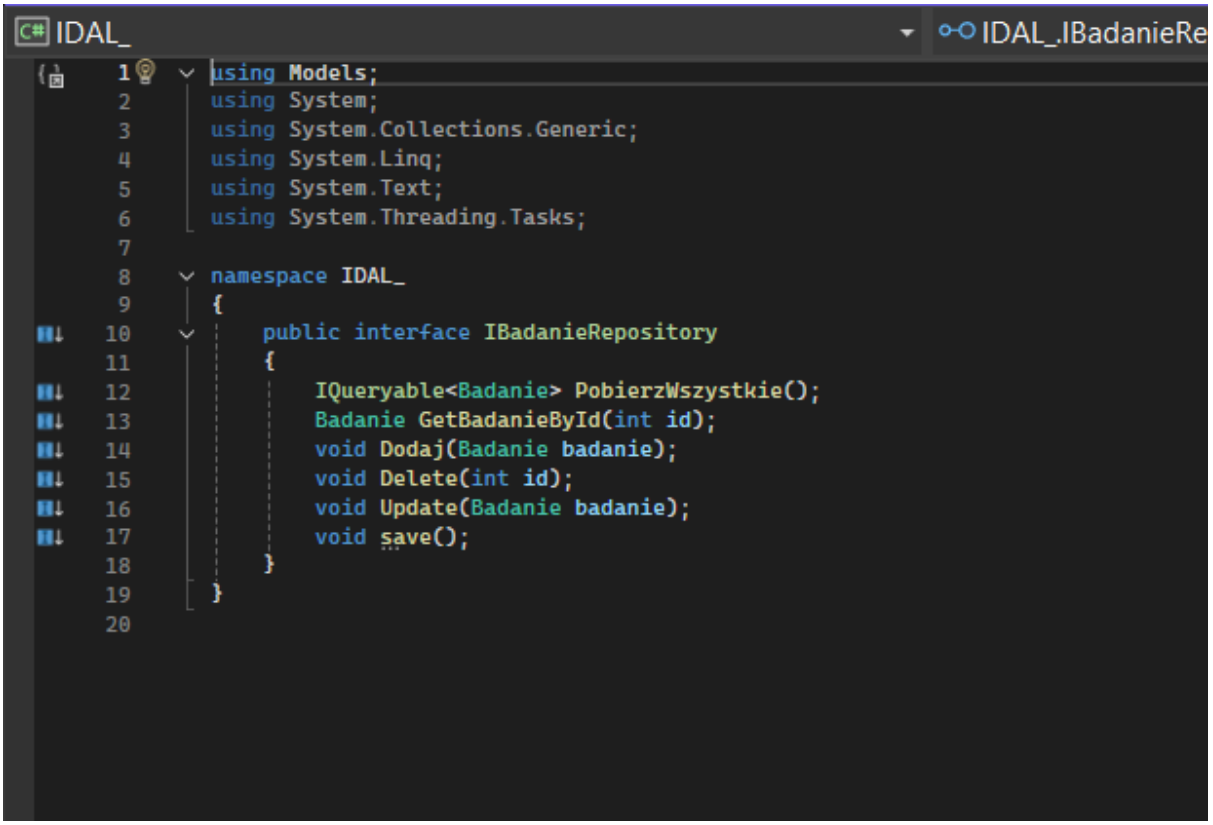
```
using DTOs;
using Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace IBLL
{
    public interface IAuthService
    {
        Task<AuthResponseDTO?> LoginAsync(LoginDTO loginDto);
        Task<AuthResponseDTO?> RegisterAsync(RegisterDTO registerDto);
        Task<AuthResponseDTO?> RefreshTokenAsync(string refreshToken);
        Task<bool> LogoutAsync(int userId);
        Task<bool> ChangePasswordAsync(int userId, ChangePasswordDTO changePasswordDto);
        Task<bool> IsUserActiveAsync(int userId);
        string GenerateJwtToken(Osoba user);
        string GenerateRefreshToken();
        Task<Osoba?> ValidateRefreshTokenAsync(string refreshToken);
    }
}
```

BLL (implementacja serwisu):

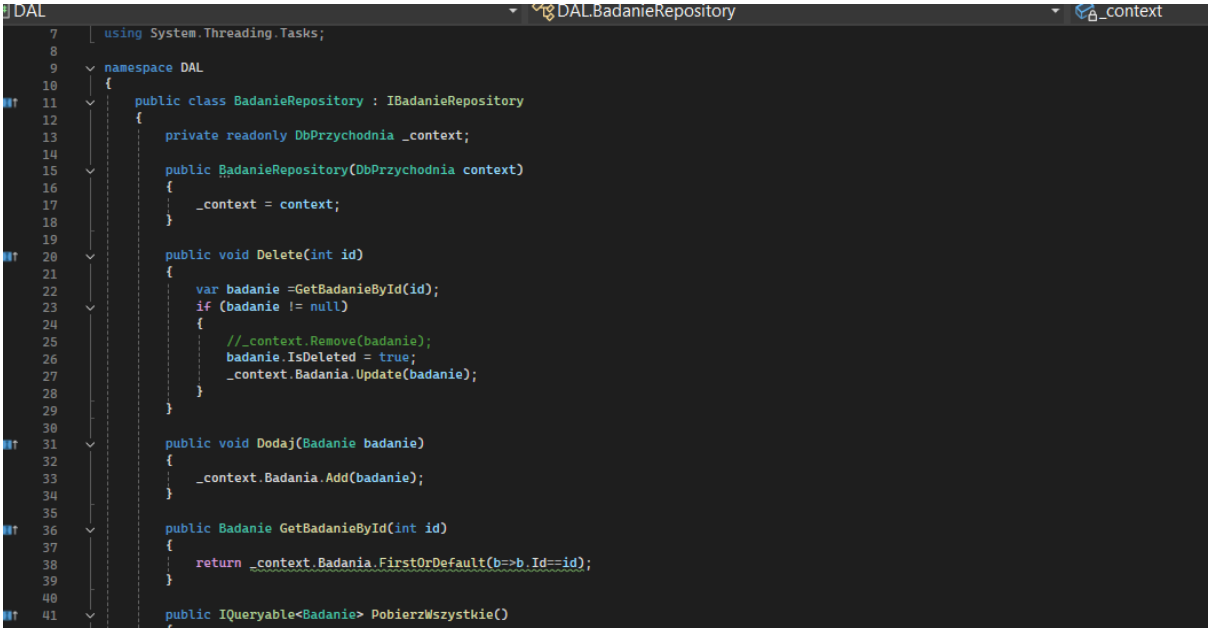


```
namespace BLL
{
    public class AuthService : IAuthService
    {
        private readonly IOsobaService _osobaService;
        private readonly IConfiguration _configuration;

        public AuthService(IOsobaService osobaService, IConfiguration configuration)
        {
            _osobaService = osobaService;
            _configuration = configuration;
        }

        public async Task<AuthResponseDTO?> LoginAsync(LoginDTO loginDto)
        {
            var user = _osobaService.GetOsobaByLogin(loginDto.Login);
            if (user == null || !user.IsActive)
                return null;

            if (!PasswordHasher.VerifyPassword(loginDto.Haslo, user.Haslo))
                return null;

            var accessToken = GenerateJwtToken(user);
            var refreshToken = GenerateRefreshToken();

            user.RefreshToken = refreshToken;
            _osobaService.Update(user);
            _osobaService.Save();

            return new AuthResponseDTO
            {
                Token = accessToken,
                RefreshToken = refreshToken,
                UserId = user.Id,
                Login = user.Login,
```

IDAL (interface repozytorium):




DAL (implementacja repozytorium):




## Wnioski:

Realizacja projektu pozwoliła nam nie tylko na stworzenie prawie kompletnej aplikacji, ale przede wszystkim na zdobycie cennych doświadczeń związanych z pracą zespołową, wykorzystywaniem nowoczesnych technologii oraz stosowaniem dobrych praktyk wytwarzania oprogramowania.

Praca w pięcioosobowym zespole wymagała od nas bieżącej komunikacji i koordynacji działań, co okazało się trudniejsze niż myśleliśmy.
Jednym z głównych wyzwań okazało się zarządzanie repozytorium kodu oraz integracja zmian wprowadzanych równolegle przez różnych członków zespołu. Zdarzały się konflikty przy łączeniu gałęzi (branchy), co wymagało dodatkowej pracy przy rozwiązywaniu niezgodności oraz ustalenia jasnych reguł commitowania i pracy z systemem kontroli wersji Git.
Dzięki temu nauczyliśmy się większej dyscypliny w zakresie organizacji kodu i komunikacji w zespole programistycznym.

Projekt był również okazją do pogłębienia znajomości technologii, które wcześniej znaliśmy jedynie w podstawowym zakresie. Backend został zrealizowany w oparciu o platformę .NET 8, a frontend w Angularze — to zestaw technologii, który umożliwił nam poznanie zarówno nowoczesnych narzędzi Microsoftu, jak i frameworków frontendowych opartych na TypeScript. Dużo uwagi poświęciliśmy również komunikacji poprzez REST API, co pozwoliło nam lepiej zrozumieć proces integracji systemów klienckich i serwerowych.

W trakcie realizacji projektu nauczyliśmy się też praktycznego wykorzystania wzorców architektonicznych, takich jak warstwowe podejście (separacja warstwy prezentacji, logiki biznesowej i dostępu do danych). Wdrożenie Entity Framework Core, JWT do autoryzacji czy Swaggera do dokumentacji API umożliwiło nam lepsze poznanie technologii stosowanych w rzeczywistych projektach komercyjnych.

**Podsumowując**, realizacja projektu przyniosła nam:

- praktyczne doświadczenie w pracy zespołowej przy większym projekcie,
- umiejętność radzenia sobie z konfliktami w repozytorium i wdrażania dobrych praktyk Git,
- pogłębioną znajomość Angulara i .NET 8,
- wiedzę z zakresu autoryzacji, bezpieczeństwa i dokumentacji API,
- rozwój umiejętności organizacyjnych oraz komunikacyjnych.

Zdobyte doświadczenia z pewnością ułatwią nam udział w kolejnych, bardziej złożonych projektach, zarówno na studiach, jak i w przyszłej pracy zawodowej.