Politechnika Wrocławska

Zaawansowane zagadnienia projektowania obiektowego

Informatyczny system obsługi przychodni

Łukasz Gawron 264475,

Filip Murawski 263894,

Maciej Padula 263919,

Maksymilian Tara 264000

Spis treści

1. Wprowadzenie 3
1.1 Cel projektu 3
1.2 Streszczenie 3
1.3 Planowane funkcjonalności 3
1.4 Wybór podejścia projektowego 4
2. Opis słowny 4
2.1. Założenia 4
2.2 Wymagania 4
2.3 Ograniczenia 5
2.4 Słownik Pojęć 5
2.5. Aktorzy 5
2.6. Przypadki użycia 5
3. Słownik pojęć 5
4. Analiza wymagań użytkownika 7
5. Modele systemu z różnych perspektyw 9
5.1 Struktura systemu 9
5.2 Zachowanie systemu 9
5.3 Rozważania na temat alternatywnych rozwiązań 10
6. Rozważania implementacyjne 11
7. Praktyczna realizacja wybranego fragmentu systemu 11
7.1. Realizacja struktury bazy danych 11
7.2. Struktura fragmentu systemu 13
7.3 Testy jednostkowe 17
7.4 Przykład działania fragmentu systemu 18
8. Podsumowanie i dyskusja krytyczna 20
9. Wykaz materiałów źródłowych 21

1. Wprowadzenie

1.1 Cel projektu

Celem projektu jest stworzenie aplikacji webowej umożliwiającej zarządzanie kalendarzem pacjentów dla wielu lekarzy. System będzie współpracował z bazą danych.

System realizowany będzie przy pomocy języka C# w technologii ASP.NET Core MVC przy współpracy z bazą danych MS SQL Server.

1.2 Streszczenie

- Typ systemu: Aplikacja typu klient-serwer, gdzie interfejs użytkownika dostępny jest przez przeglądarkę internetową, a serwer zarządza danymi i logiką aplikacji.
- Baza danych: MS SQL Server będzie przechowywał dane pacjentów, lekarzy oraz ich wizyt.
- Architektura aplikacji: System będzie składał się z warstwy prezentacji (frontend), warstwy logiki biznesowej z dostępem do danych (backend).
- Wzorce projektowe: Zastosowane wzorce projektowe to
    - Model-View-Controller (MVC),
    - Dependency Injection (wstrzykiwanie zależności), aby nie uzależniać klas nadrzędnych od konkretnych implementacji, tylko od interfejsów,
    - Command Query Separation (CQS),
    - Repository do zarządzania operacjami na danych
- Interfejs użytkownika: Responsywny interfejs użytkownika zaprojektowany z użyciem technologii HTML, CSS, JavaScript oraz bibliotek takich jak Bootstrap.
- Funkcje aplikacji:
    - Zarządzanie kalendarzem wizyt pacjentów.
    - Możliwość dodawania wizyt.
    - Powiadomienia e-mail o nadchodzących wizytach.
- Środowisko: Visual Studio jako główne środowisko programistyczne oraz Git do zarządzania wersjami kodu.
- Techniki programistyczne:
    - Dziedziczenie do tworzenia wspólnych klas bazowych dla modeli danych.
    - Kompozycja do tworzenia skomplikowanych obiektów z prostszych komponentów.
- Testowanie: Wykorzystanie narzędzi takich jak NUnit do testów jednostkowych.
- Rozszerzalność: System zaprojektowany w sposób umożliwiający łatwe dodawanie nowych funkcji i modułów (modularny monolit).

1.3 Planowane funkcjonalności

Zakładamy istnienie dwóch ról użytkowników: **Pacjentów** oraz **Lekarzy**.

Pacjenci będą posiadali możliwość:

- rejestracji na wizytę w dogodnym, wolnym terminie u wybranego specjalisty.

- możliwość przeglądania diagnoz wystawionych przez lekarzy po odbytej wizycie oraz przepisanych leków wraz z dawkowaniem.

Lekarze będą posiadali możliwość zarządzania wizytami w ramach czego wchodzą:

- możliwość wpisania diagnozy

- możliwość wpisania dawkowania leków

- możliwość wystawiania elektronicznej recepty

- anulowanie wizyty

1.4 Wybór podejścia projektowego

- Scrum (sprinty 3 tygodniowe)

- Komunikacja: Discord

- Zarządzanie projektem: Azure Devops

2. Opis słowny

2.1. Założenia

Projekt zakłada stworzenie kompleksowego systemu zarządzania przychodnią, który będzie wspierał pracę personelu medycznego oraz ułatwiał pacjentom umawianie wizyt. System ma na celu optymalizację obecnych procesów pracy, poprawę komunikacji między użytkownikami systemu oraz zapewnienie łatwiejszego i szybszego dostępu do informacji o pacjentach i usługach medycznych. Przyjęto założenie, że system będzie funkcjonował jako aplikacja webowa, dostępna z poziomu przeglądarki internetowej bez konieczności instalacji dodatkowego oprogramowania.

Aplikacja ma na celu ułatwiać rejestrowanie się na wizyty pacjentom poprzez intuicyjny interfejs. W podobny sposób, aplikacja ma na celu ułatwić pracę lekarzom przychodni, poprzez udostępniając im widok kalendarza z planowanymi wizytami.

2.2 Wymagania

Wymagania funkcjonalne:

- Zarządzanie kalendarzem: możliwość planowania wizyt, zarządzania harmonogramem lekarzy.

- Ewidencja pacjentów: przechowywanie i dostęp do wizyt i danych osobowych.

- Moduł komunikacji: system powiadomień mailowych dla pacjentów o nadchodzących wizytach i zmianach w harmonogramie.

Wymagania niefunkcjonalne:

- Skalowalność: system powinien być zdolny do obsługi rosnącej liczby użytkowników i danych bez utraty wydajności.

- Intuicyjność: interfejs użytkownika musi być przejrzysty i łatwy w użyciu dla wszystkich grup użytkowników.

2.3 Ograniczenia

- Budżet: dostępne środki finansowe na rozwój i utrzymanie systemu są ograniczone.

- Technologia: wybór technologii jest ograniczony do tych, które są szeroko wspierane i łatwo dostępne na rynku.

2.4 Słownik Pojęć

- Aktor: użytkownik systemu (lekarz, pacjent).

- Przypadek użycia: opis interakcji użytkownika z systemem w celu wykonania określonej funkcji (np. umówienie wizyty).

- Ewidencja medyczna: zbiór danych i informacji dotyczących historii medycznej pacjenta.

- Moduł: funkcjonalnie wydzielona część systemu realizująca określone zadania.

2.5. Aktorzy

- Pacjent: osoba korzystająca z usług medycznych przychodni. Posiada możliwość rejestracji na wizytę u wybranego specjalisty w dostępnym terminie.

- Lekarz: profesjonalista medyczny odpowiedzialny za diagnozowanie i leczenie pacjentów. Posiada podstawową możliwość zarządzania terminami wizyt.

2.6. Przypadki użycia

- Umówienie wizyty przez pacjenta: Pacjent loguje się do systemu, wybiera lekarza, specjalizację, i dostępny termin, następnie potwierdza wizytę.

- Przeglądanie harmonogramu przez lekarza: Lekarz loguje się do systemu, aby przeglądać swój harmonogram wizyt, zarządzać swoim czasem pracy i dostępnością.

3. Słownik pojęć

- Pacjent
    - Opis: Osoba, która korzysta z usług medycznych i jest zarejestrowana w systemie.
    - Klasy: User, UserRepository, GetUsersHandler
    - Atrybuty: identyfikator, imię, nazwisko, data urodzenia, adres email, numer telefonu.

- Lekarz
    - Opis: Profesjonalista medyczny, który świadczy usługi zdrowotne dla pacjentów.
    - Klasy: User, UserRepository, GetUsersHandler
    - Atrybuty: identyfikator, imię, nazwisko, data urodzenia, adres email, numer telefonu

- Wizyta
    - Opis: Zaplanowane spotkanie pomiędzy pacjentem a lekarzem w celu udzielenia porady medycznej lub przeprowadzenia badania.
    - Klasy: Appointment, AppointmentRepository, GetAppointmentsHandler, CreateAppointmentsHandler
    - Atrybuty: identyfikator, data, godzina, powód wizyty, pacjent, lekarz.

- Kalendarz
    - Opis: Narzędzie do organizacji i zarządzania wizytami pacjentów, dostępne zarówno dla pacjenta (lista wizyt pacjenta), jak i dla lekarzy (lista zaplanowanych wizyt).
    - Klasy: UserCalendarItem, GetUserCalendarHandler, CreatePatientCalendarItemHandler, CreatePatientCalendarItemCommand
    - Atrybuty: lista wizyt, Wizyta: identyfikator, data, godzina, powód wizyty, pacjent, lekarz.

- Użytkownik
    - Opis: Osoba korzystająca z systemu, mogąca być pacjentem lub lekarzem.
    - Klasy: User, UserType
    - Atrybuty: login, hasło, rola (pacjent, lekarz).

- Powiadomienie
    - Opis: Informacja wysyłana do użytkownika w celu przypomnienia o nadchodzącej wizycie lub zmianie statusu wizyty.
    - Atrybuty: treść, data wysłania, typ powiadomienia (e-mail).

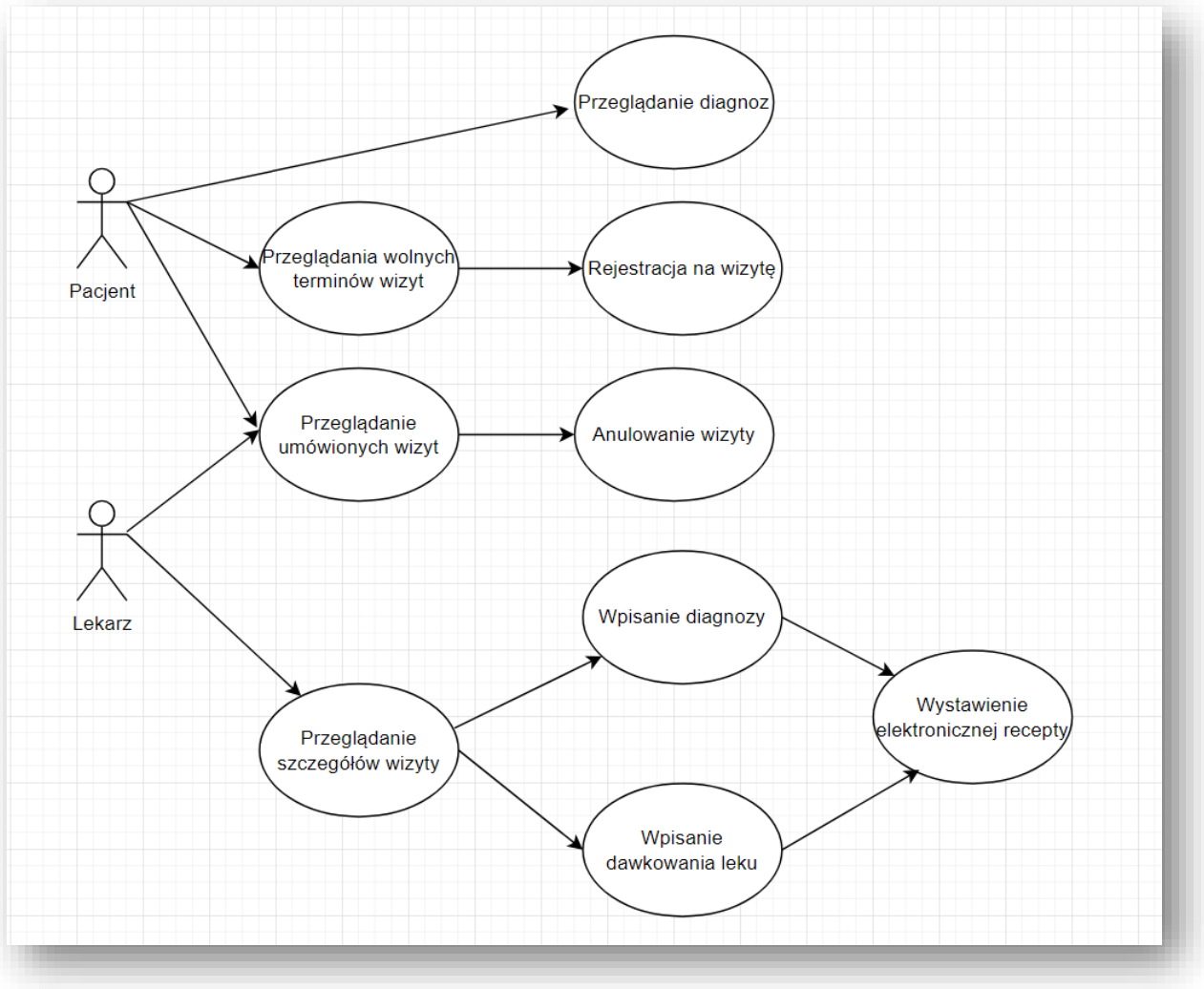
- Infrastruktura
    - Opis: Moduł odpowiedzialny za wystawienie konkretnych implementacji na podstawie kontraktu wystawionego przez moduły: Użytkowników, Wizyt oraz Kalendarza. Dla przykładu w tym module znajduje się implementacja repozytorium AppointmentRepository – implementuje on interfejs IAppointmentRepository, dostarczając wyżej wymienionemu modułowi konkretną klasę umożliwiającą dostęp do bazy danych, dzięki wykorzystaniu mechanizmu wstrzykiwania zależności (Dependency Injection). Zaletą takiego rozwiązania jest możliwość zmiany tej implementacji w infrastrukturze.
    - Klasy: AppointmentRepository, UserRepository

- Historia medyczna
    - Opis: Zbiór danych medycznych pacjenta, obejmujący wcześniejsze wizyty wraz z diagnozami, wypisane recepty oraz dawkowania leków.

- REST API (Interfejs Programowania Aplikacji)
    - Zestaw narzędzi i protokołów umożliwiających komunikację między różnymi częściami aplikacji oraz integrację z innymi systemami.
    - Klasy: CallendarController, CreateAppointmentRequest, CalendarIndexViewModel
    - Atrybuty: punkty końcowe, metody, zasoby.

4. Analiza wymagań użytkownika

Poniżej widoczny jest diagram przypadków użycia aplikacji:



Rysunek 1. Diagram przypadków użycia systemu.

Główne scenariusze działania:

- Rejestracja na wizytę przez pacjenta
  - Pacjent loguje się do systemu.
  - Przechodzi do kalendarza i wybiera opcję dodania nowej wizyty.
  - Wybiera lekarza z listy.
  - Wprowadza szczegóły wizyty: powód wizyty. Podczas tworzenia wizyty pacjent wybiera z dostępnych terminów (data oraz godzina)
  - Zatwierdza wprowadzone dane.
  - Lekarz otrzymuje powiadomienie o dodaniu wizyty.

- Przeglądanie umówionych wizyt
  - Użytkownik loguje się do systemu.
  - Przechodzi do kalendarza.

    - Przegląda zaplanowane wizyty.
    - Użytkownik może wejść w szczegóły wizyty, gdzie znajdują się informacje o diagnozie oraz opcjonalnie recepta oraz dawkowanie leków ustalone przez lekarza po wizycie.
    - Lekarz posiada możliwość wprowadzania/edycji szczegółów wizyty, takich jak informacje o diagnozie, recepta, dawkowanie leków. Lekarz posiada również możliwość anulowania wizyty, w przypadku, gdy zauważy nieprawidłowości.

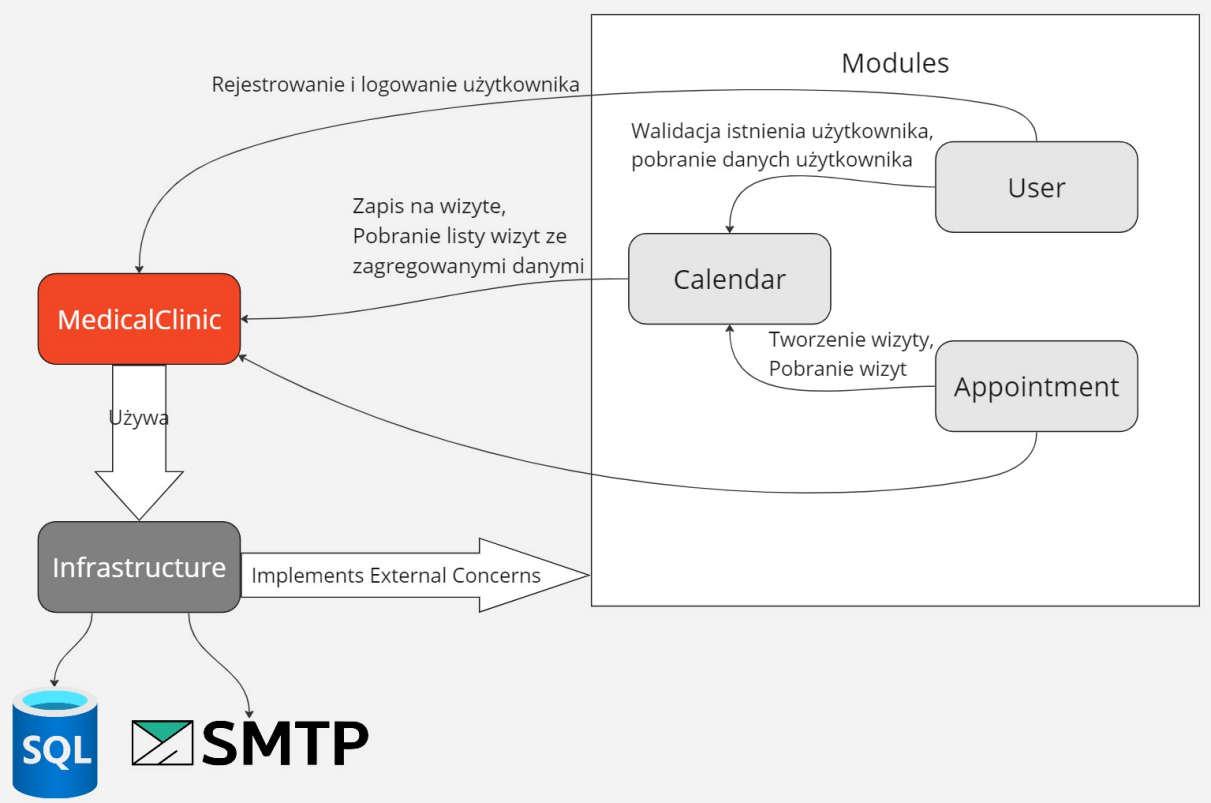
- Wyprowadzanie danych przez lekarza po odbyciu wizyty
    - Lekarz loguje się do systemu.
    - Przechodzi do sekcji zarządzania wizytami.
    - Wybiera konkretną wizytę.
    - Aktualizuje dane wizyty, dodając nowe informacje o diagnozie.
    - Zatwierdza zmiany, które są automatycznie zapisane w systemie.

Scenariusze nietypowe:

- Problemy z rejestracją użytkownika
    - Nowy użytkownik wprowadza swoje dane rejestracyjne.
    - System wykrywa, że podany adres e-mail jest już używany.
    - Użytkownik otrzymuje informację o konieczności wyboru innego adresu e-mail lub odzyskania hasła.

- Awaria bazy danych podczas aktualizacji
    - Lekarz próbuje zaktualizować dane pacjenta.
    - System napotyka problem z zapisem danych z powodu awarii bazy danych.
    - Lekarz otrzymuje komunikat o błędzie i próbuje ponownie zapisać dane po naprawieniu problemu.

5. Modele systemu z różnych perspektyw



Rysunek 2. Diagram przedstawiający moduły oraz przepływ kontroli pomiędzy nimi.

5.1 Struktura systemu

Projekt opiera się na strukturze modularnego monolitu i „Clean Architecture". Oznacza to, że przypadki użycia są organizowane w moduły, w których zarówno proste, jak i złożone przypadki są grupowane. Każdy moduł odpowiedzialny jest za realizację logiki biznesowej i warstwę prezentacji, a także udostępnia publiczny interfejs do wywoływania tych przypadków. Dodatkowo, moduły te oferują interfejsy umożliwiające integrację z zewnętrznymi serwisami. Wszystkie te implementacje są centralizowane w wspólnym module o nazwie „Infrastructure".

5.2 Zachowanie systemu

System składa się z różnorodnych modułów. Moduły proste, takie jak „User" i „Appointment", obsługują podstawowe przypadki użycia, takie jak rejestracja nowego użytkownika czy tworzenie wpisu wizyty w systemie.

Z kolei moduł „Calendar" zajmuje się bardziej złożonymi operacjami, oferując funkcje takie jak pobieranie informacji o zaplanowanych wizytach oraz danych lekarzy przypisanych do tych wizyt.

5.3 Rozważania na temat alternatywnych rozwiązań

- MVC: Aplikacja w strukturze pojedynczego monolitu, gdzie każdej encji w bazie danych odpowiadał będzie własny kontroler i serwisy. Nie występuje podział na moduły, a podział logiki jest warstwowy, na przykład warstwa logiki biznesowej dla wszystkich przypadków użycia znajdowałaby się w jednym projekcie.

- Mikro serwisy: System w takiej strukturze podzielony jest w sposób analogiczny, jak w przypadku modularnego monolitu. Każdy moduł mógłby zostać wydzielony jako osobna aplikacja mikro serwisowa, która posiadałby wydzieloną, własną infrastrukturę. Aplikacje te komunikowałyby się między sobą np. poprzez REST API bądź gRPC.

Aspekt	Monolit	Modularny monolit	Mikro serwisy
Skalowalność			
Łatwość współpracy			
Poziom skomplikowania infrastruktury			
Zasoby			
Zarządzanie zależnościami			
Różnorodność technologiczna			

Legenda:

Kolor	Poziom
	Wysoka
	Umiarkowana
	Niska

Do realizacji projektu wybrano architekturę opartą o modularny monolit, ponieważ jego poziom skomplikowania infrastruktury oraz wymaganych zasobów (mierzonych w godzinach pracy oraz wymaganego sprzętu do hostowania aplikacji). Modularny monolit pozwala na stosunkowo dobre zarządzanie zależnościami, jednocześnie nie będąc przesadnie skomplikowanym. Sprawia to, że jest idealną architekturą dla małych oraz średniej wielkości projektów.

6. Rozważania implementacyjne

Wybrano następujący stos technologiczny:

- .NET Core 8.0 (C#)
- ASP.NET Core MVC
- Razor Pages
- Entity Framework
- Microsoft SQL Server
- Microsoft Azure

Wymienione technologie należą do platformy .NET, wybrane ze względu na ich znajomość przez członków zespołu oraz ich popularność w środowiskach produkcyjnych wielu firm.

Do infrastruktury wybrano Microsoft SQL Server oraz chmurę Microsoft Azure, ze względu na fakt, iż obie technologie są dobrze zintegrowane z platformą .NET.

Narzędzia z jakich skorzystamy podczas implementacji:

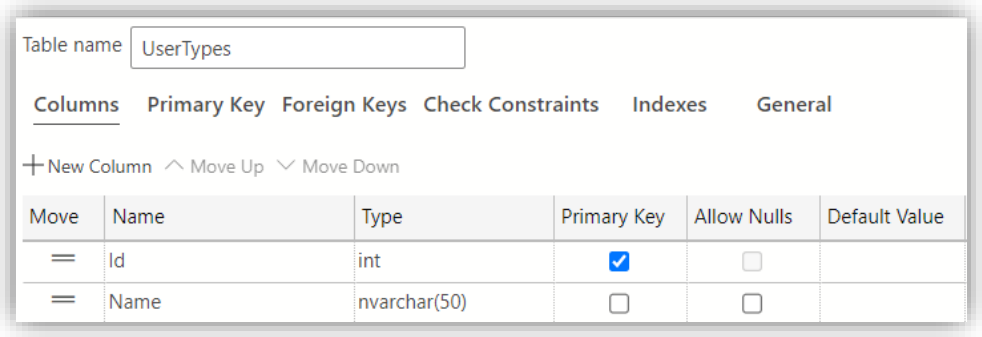
- Microsoft Visual Studio
- Microsoft Azure Data Studio
- Microk8s, Docker, DockerDesktop
- Github
- Draw.io

7. Praktyczna realizacja wybranego fragmentu systemu

Po dokonaniu analizy przystąpiono do realizacji fragmentu systemu. Zaimplementowano wybrane przypadki użycia, jakimi są wyświetlanie wizyt lekarskich oraz dodawanie nowych wizyt.

7.1. Realizacja struktury bazy danych

W pierwszej kolejności należało zaprojektować oraz zaimplementować strukturę bazy danych. W celu realizacji wybranych przypadków użycia powstały trzy encje, odpowiadające za typ użytkownika, dane użytkownika oraz wizyty.



Rysunek 3. Tabela „UserTypes”.

Tabela „UserTypes” odpowiada za przechowywanie typów użytkowników systemu. W implementowanym fragmencie systemy przewiduje się dwa typy użytkowników – Pacjentów oraz Doktorów.

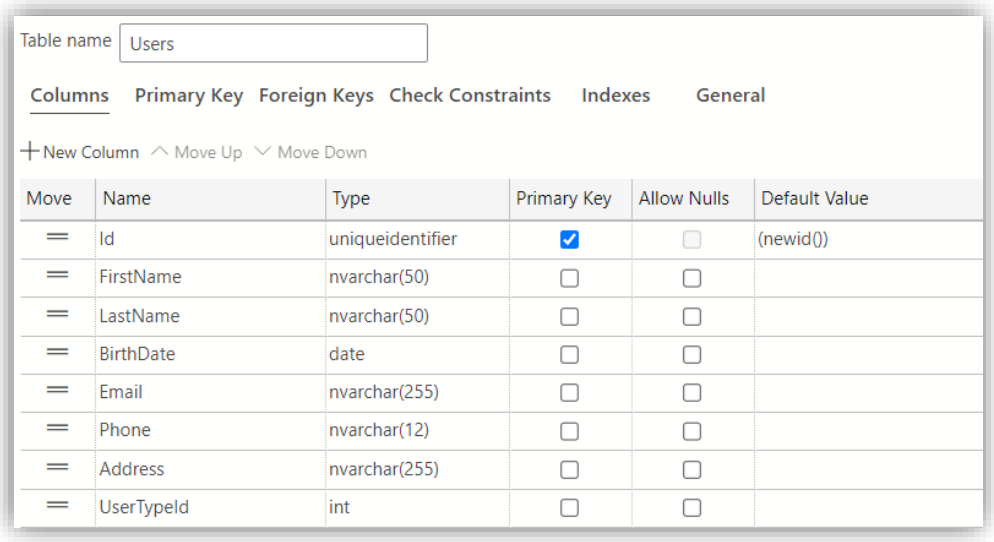


Table name: Users

Columns | Primary Key | Foreign Keys | Check Constraints | Indexes | General

New Column | Move Up | Move Down

Move	Name	Type	Primary Key	Allow Nulls	Default Value
=	Id	uniqueidentifier	☑	☐	(newid())
=	FirstName	nvarchar(50)	☐	☐	
=	LastName	nvarchar(50)	☐	☐	
=	BirthDate	date	☐	☐	
=	Email	nvarchar(255)	☐	☐	
=	Phone	nvarchar(12)	☐	☐	
=	Address	nvarchar(255)	☐	☐	
=	UserTypeId	int	☐	☐	

Rysunek 4. Tabela „Users”.

Tabela „Users” służy przechowywaniu danych użytkowników systemu. System zbiera takie dane osobowe jak Imię, Nazwisko, Data urodzenia, E-mail, Nr. Telefonu oraz Adres. Tabela zawiera także wcześniej zdefiniowany Typ użytkownika – klucz obcy „UserTypeId”.

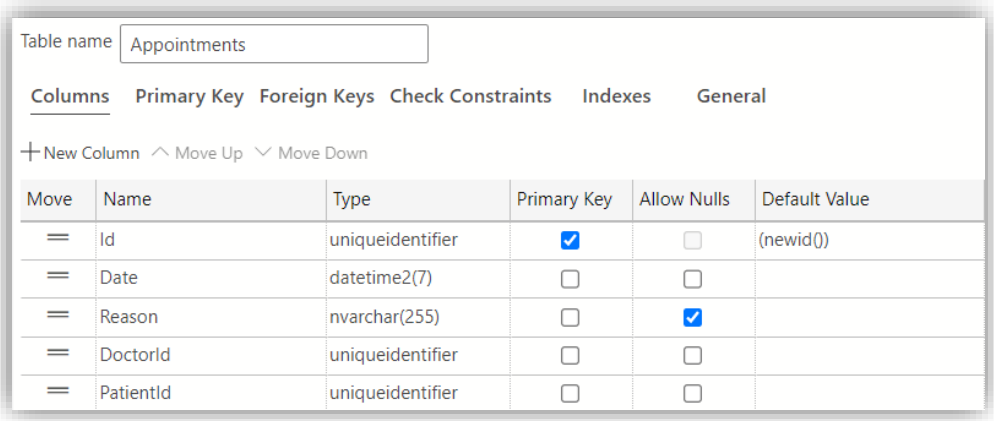


Table name: Appointments

Columns | Primary Key | Foreign Keys | Check Constraints | Indexes | General

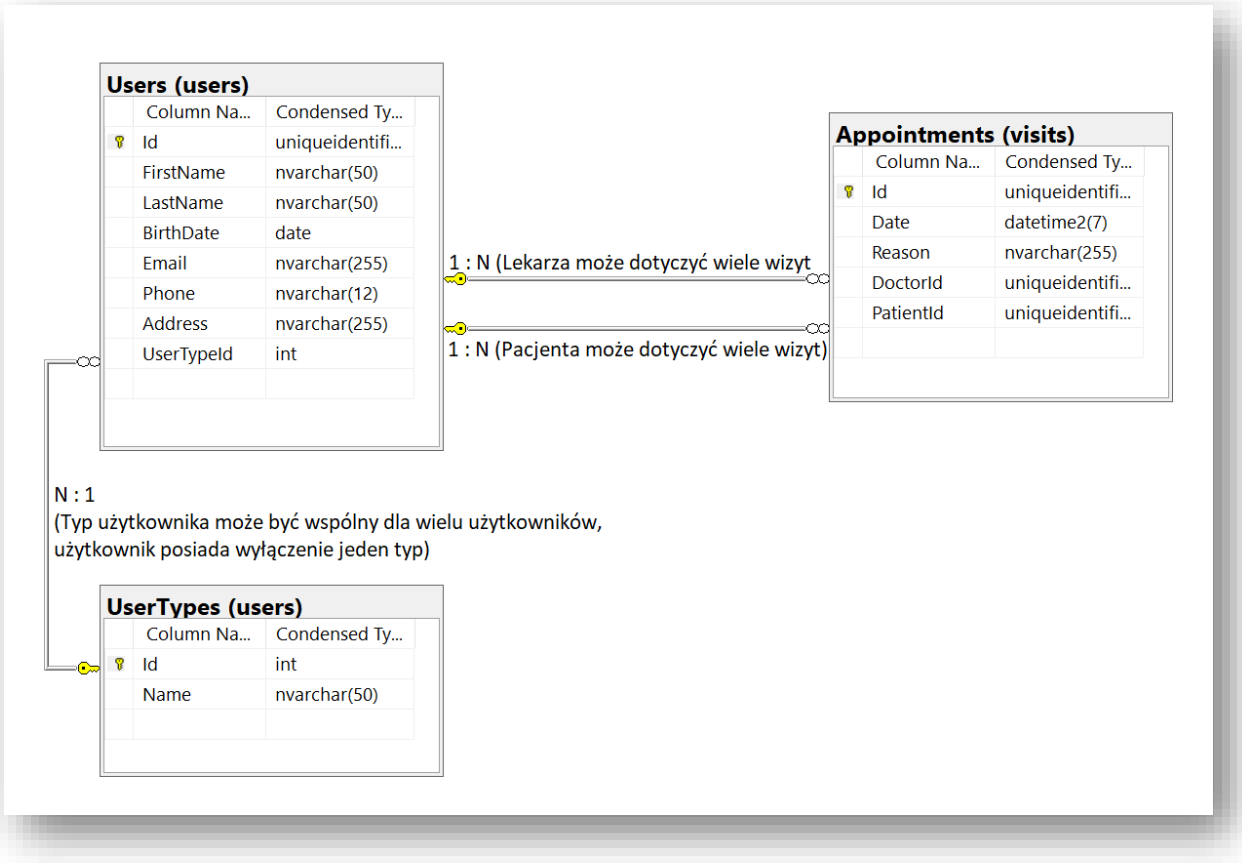
New Column | Move Up | Move Down

Move	Name	Type	Primary Key	Allow Nulls	Default Value
=	Id	uniqueidentifier	☑	☐	(newid())
=	Date	datetime2(7)	☐	☐	
=	Reason	nvarchar(255)	☐	☑	
=	DoctorId	uniqueidentifier	☐	☐	
=	PatientId	uniqueidentifier	☐	☐	

Rysunek 5. Tabela „Appointments”

Tabela „Appointments” odpowiada przechowywaniu informacji o wizytach lekarskich. Encja zawiera takie dane jak data wizyty, powód wizyty, lekarza oraz pacjenta (odwołania do wcześniej zdefiniowanej tabeli Użytkowników – klucze obce).

Poniżej przedstawiony został diagram relacji encji (ERD):



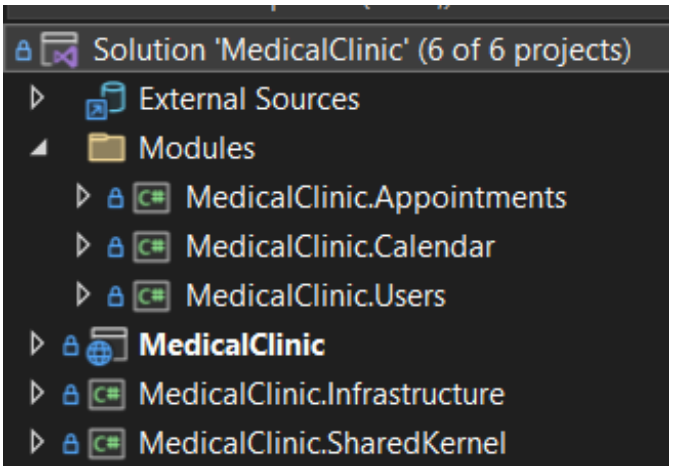
Rysunek 6. Diagram relacji encji wykorzystywanych w celu zaimplementowania funkcjonalności rezerwacji wizyt.

7.2. Struktura fragmentu systemu

W strukturze systemu wyróżniamy moduły niższego poziomu, odpowiedzialne za obsługę komunikacji z bazą danych – „MedicalClinic.Appointments" oraz „MedicalClinic. W modułach tych zaimplementowano m.in. metody odpowiedzialne za pobieranie encji z bazy danych oraz ich mapowanie na pożądane typy, a także dodawanie nowych wpisów w bazie danych.

Moduł o nazwie „MedicalClinic.Calendar" jest wyżej w hierarchii od dwóch pozostałych modułów – jest to moduł agregujący. Moduł ten posiada odwołania do wcześniej wymienionych modułów podrzędnych, z których wykorzystuje zależności m.in jako warstwa dostępu do danych. Moduł kalendarza zwiera w sobie logikę służącą przetworzeniu danych pobranych z bazy, a następnie zwraca je w pożądanej formie do ostatniej warstwy kontrolerów odpowiadającej za wystawienie endpointów.

Struktura projektu prezentuje się następująco (jest to modularny monolit):

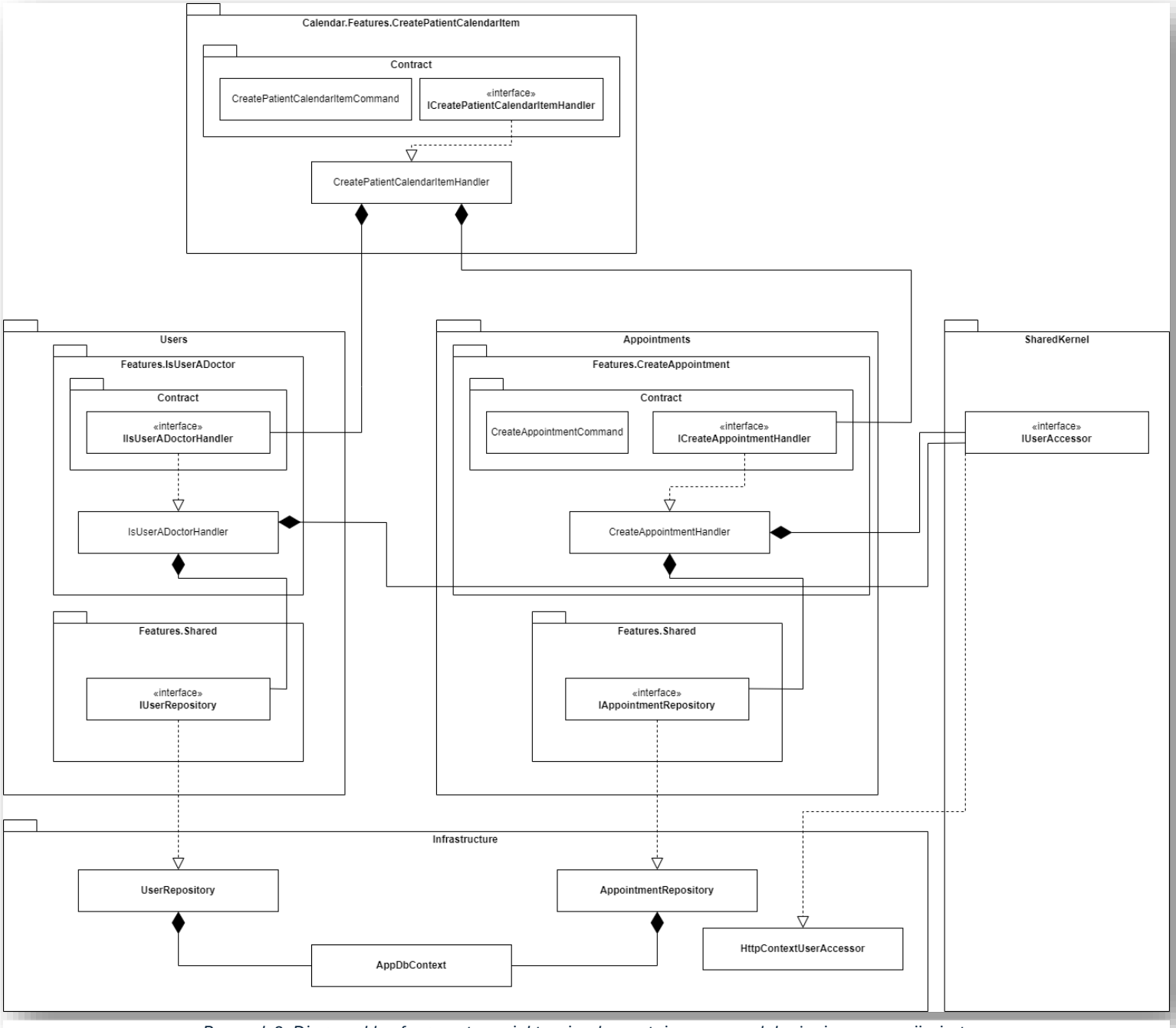


Rysunek 7. Zrzut ekranu z IDE Visual Studio przedstawiający strukturę projektu.

Przykładowa klasa zajmująca się pobraniem kalendarza użytkownika:

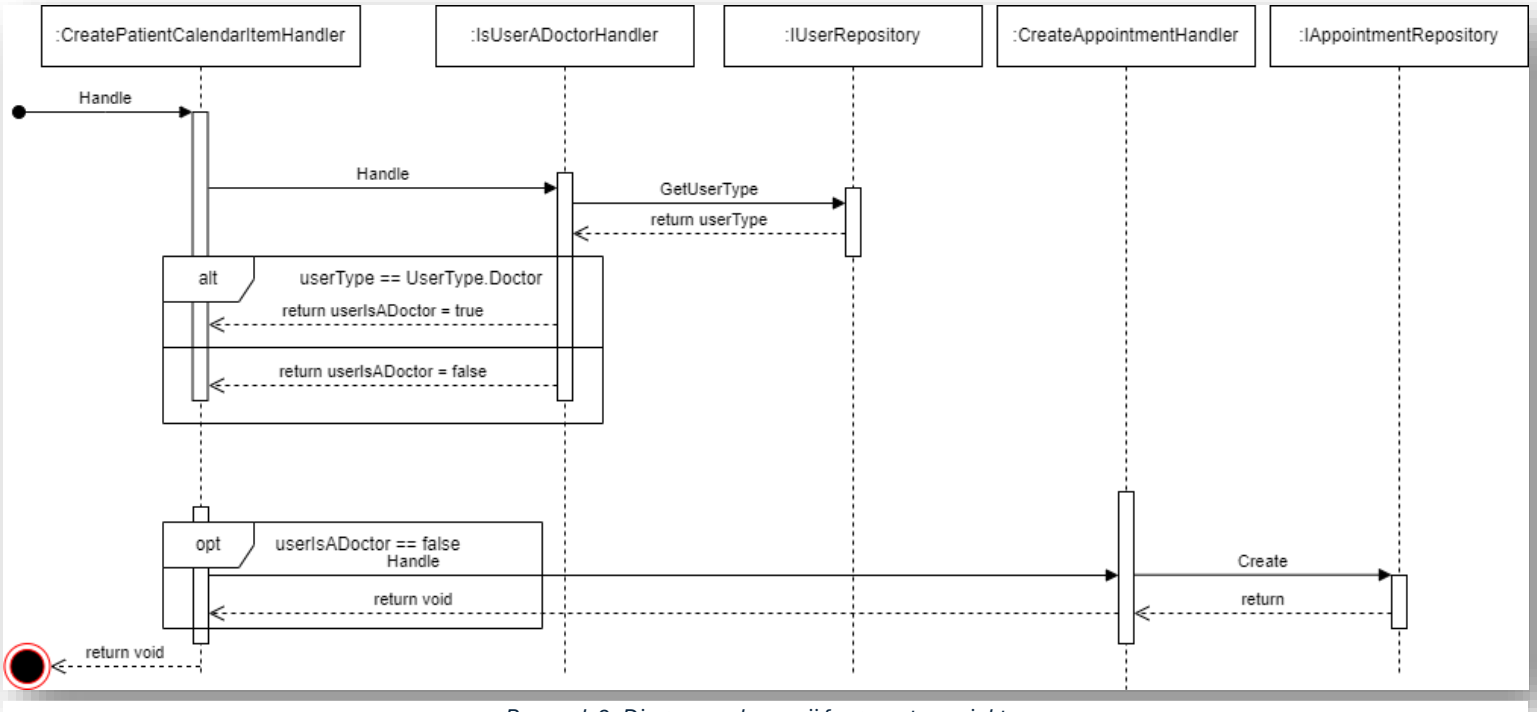
```
internal class GetUserCalendarHandler : IGetUserCalendarHandler
{
    private readonly IGetAppointmentsHandler _getAppointmentsHandler;
    private readonly IGetUsersHandler _getUsersHandler;

    public GetUserCalendarHandler(
        IGetAppointmentsHandler getAppointmentsHandler,
        IGetUsersHandler getUsersHandler)
    {
        _getAppointmentsHandler = getAppointmentsHandler;
        _getUsersHandler = getUsersHandler;
    }

    public async Task<List<UserCalendarItem>> Handle()
    {
        var appointments = await _getAppointmentsHandler.Handle();
        var users = await
_getUsersHandler.Handle(new(GetUserIds(appointments)));

        return appointments
            .Select(x => new UserCalendarItem(
                x.Id,
                x.Date,
                x.Reason,
                users.GetValueOrDefault(x.DoctorId),
                users.GetValueOrDefault(x.PatientId)))
            .ToList();
    }

    private IEnumerable<Guid> GetUserIds(IEnumerable<Appointment>
appointments) =>
        appointments
            .SelectMany(x => new Guid[] { x.DoctorId, x.PatientId })
            .ToHashSet();
}
```

Poniżej znajduje się diagram klas wybranego przypadku użycia, a mianowicie tworzenia kalendarza pacjenta:



Rysunek 8. Diagram klas fragmentu projektu – implementujące przypadek użycia rezerwacji wizyt.

Diagram sekwencji tworzenia wpisu do kalendarza:



Rysunek 9. Diagram sekwencji fragmentu projektu.

7.3 Testy jednostkowe

Napisano testy jednostkowe modułów „Appointments”, „Calendar” oraz „Users” przy pomocy biblioteki NUnit. Dodatkowo wykorzystana została biblioteka NSubstitute w celu zasymulowania zależności testowanych klas (poprzez wstrzykiwanie zależności). Wykorzystano również bibliotekę FluentAssertions.

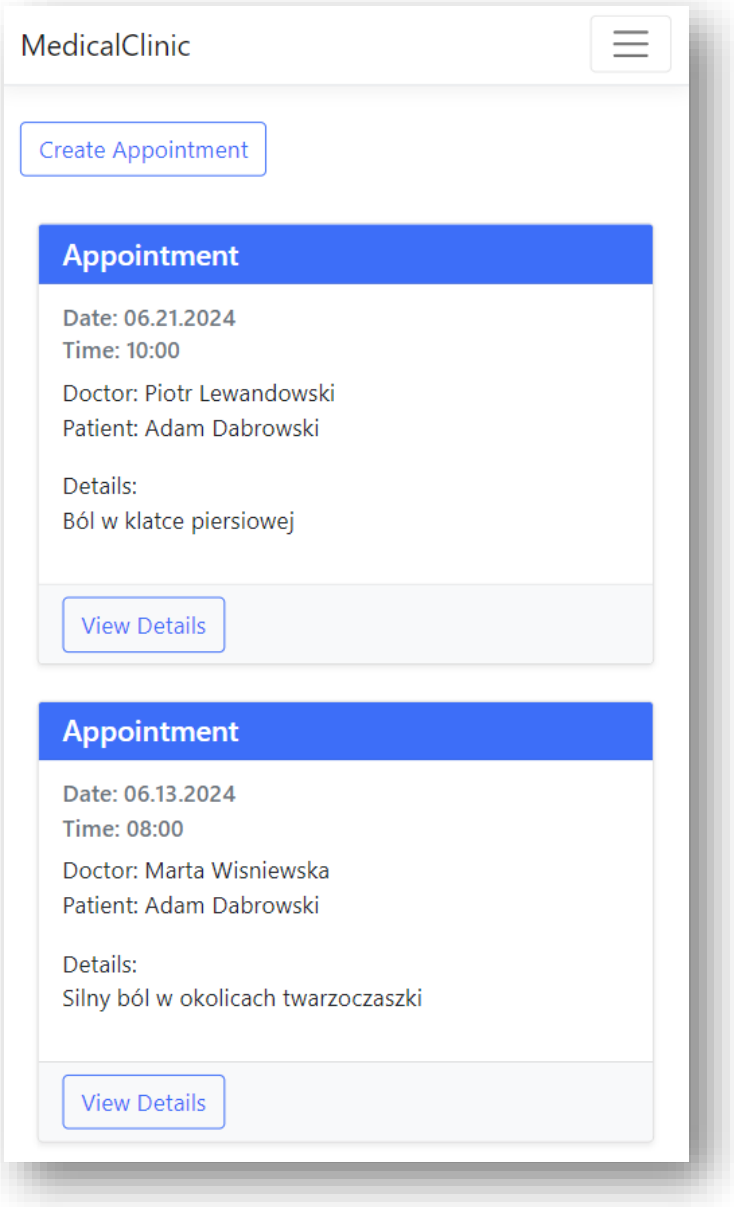
Poniżej znajduje się przykładowy test jednostkowy klasy odpowiedzialnej za zwracanie zaplanowanych wizyt dla lekarza o zadanym identyfikatorze:

```
9  public class GetAppointmentsHandlerTest
10 {
11     private GetAppointmentsHandler _sut;
12     private IAppointmentRepository _appointmentRepository;
13     private IUserAccessor _userAccessor;
14
15     [SetUp]
16     public void Setup()
17     {
18         _appointmentRepository = Substitute.For<IAppointmentRepository>();
19         _userAccessor = Substitute.For<IUserAccessor>();
20         _sut = new GetAppointmentsHandler(_appointmentRepository, _userAccessor);
21     }
22
23     [Test]
24     public async Task Handle_WhenUserTypeIsDoctor_ShouldReturnAppointmentsByDoctorId()
25     {
26         // Arrange
27         var doctorId = Guid.NewGuid();
28         _userAccessor.UserType.Returns(returnThis: UserType.Doctor);
29         _userAccessor.UserId.Returns(doctorId);
30
31         var appointments = new List<Appointment>
32         {
33             new() { Id = Guid.NewGuid(), DoctorId = doctorId },
34             new() { Id = Guid.NewGuid(), DoctorId = doctorId }
35         };
36
37         _appointmentRepository.GetByDoctorId(doctorId).Returns(appointments);
38
39         // Act
40         var result :List<Appointment> = await _sut.Handle();
41
42         // Assert
43         result.Should().NotBeNull();
44         result.Should().BeEquivalentTo(appointments);
45     }
```

Rysunek 10. Test jednostkowy klasy „GetAppointmentHandler”.

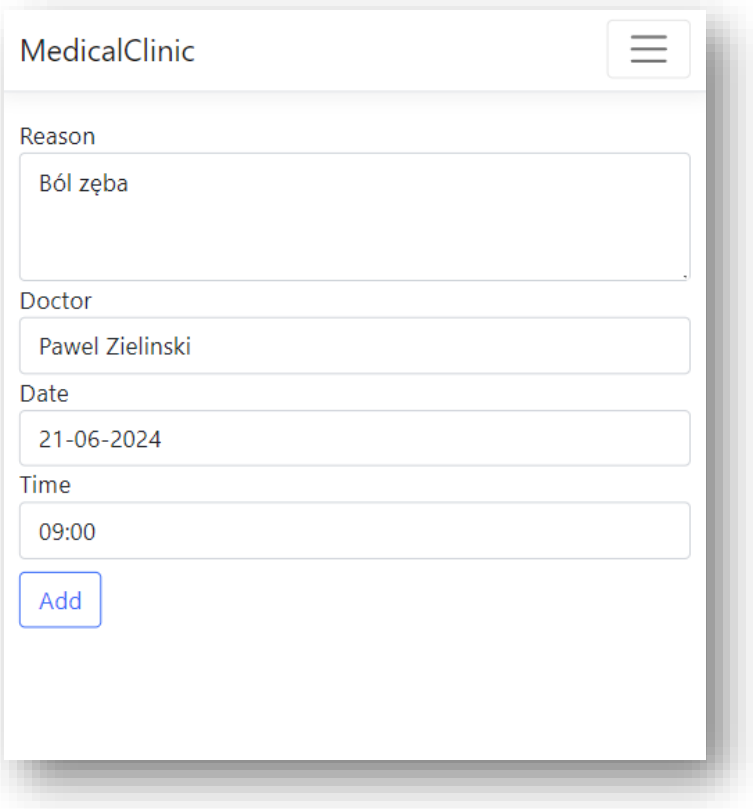
7.4 Przykład działania fragmentu systemu

Zaimplementowano wybrane przypadki użycia, jakimi są wyświetlanie wizyt lekarskich oraz dodawanie nowych wizyt. Akcje dostępne są w widoku aplikacji internetowej, widocznym poniżej.



Rysunek 11. Widok listy wizyt użytkownika.

Dodawanie wizyt (formularz) widoczny poniżej:



Rysunek 12. Formularz tworzenia nowej wizyty.

8. Podsumowanie i dyskusja krytyczna

Przedstawione cele i założenia projektu zostały zrealizowane.

Podsumowując, projekt wykazał, że wybór architektury modularnego monolitu i technologii ASP.NET w połączeniu z SQL Server jest solidnym rozwiązaniem dla aplikacji wymagających wysokiej niezawodności, wydajności i skalowalności. Osiągnięte cele projektowe potwierdzają, że zastosowane technologie i podejścia architektoniczne sprawdziły się w praktyce. Zastosowanie modularnego monolitu uprościło równoległą pracę w zespole, poprzez między innymi zminimalizowanie ryzyka powstania konfliktów między zmianami wprowadzonymi przez programistów w trakcie równoległej pracy. Zaleca się kontynuację monitorowania wydajności i bezpieczeństwa systemu, regularne aktualizacje komponentów oraz ewentualne rozważenie stopniowej ewolucji systemu w kierunku mikroserwisów, jeśli wzrost i wymagania biznesowe będą tego wymagać.

9. Wykaz materiałów źródłowych

- https://code-maze.com/using-dapper-with-asp-net-core-web-api/
- https://www.tutorialspoint.com/asp.net_core/asp.net_core_setup_mvc.htm
- https://bulldogjob.pl/readme/modular-monolith-modularnosc-droga-do-mikroserwisow
- https://www.c-sharpcorner.com/article/jwt-json-web-token-authentication-in-asp-net-core/
- https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16