UNIWERSYTET RZESZOWSKI
WYDZIAŁ NAUK ŚCISŁYCH I TECHNICZNYCH
INSTYTUT INFORMATYKI

MediCare

Projekt Inżynierski

Sebastian Mikoś 131472
Michał Kalisiak 131448
Amadeusz Nowak 131477

Praca wykonana pod kierunkiem
mgr inż. Aleskander Wojtowicz

Spis treści

Wprowadzenie

1.1 Cele projektu

Celem projektu **MediCare** jest stworzenie kompletnego systemu informatycznego wspierającego pracę placówki medycznej. System umożliwia zarządzanie pacjentami, lekarzami, pokojami oraz wizytami, a także zapewnia panel administracyjny, który pozawala na zarządzanie wszystkimi tabelami z bazy danych oraz podgląd statystyk z wizyt pacjentów.

1.2 Zakres funkcjonalny

System obejmuje następujące moduły:

- **Autoryzacja i role** – obsługa użytkowników z rolami *Admin*, *Doctor*, *Patient*.
- **Pacjenci i lekarze** – rejestracja, logowanie, edycja danych, reset haseł.
- **Wizyty** – planowanie, historia, edycja, statystyki dzienne i zbiorcze.
- **Pokoje i specjalizacje** – przypisywanie specjalizacji do pokoi, zarządzanie zasobami.
- **Panel administracyjny** – zarządzanie danymi, dashboardem oraz podgląd statystyk.

1.3 Zawartość pracy

Praca została podzielona na następujące rozdziały:

- Rozdział poświęcony strukturze i kompilacji projektu – opis organizacji kodu źródłowego oraz procesu kompilacji,
- Rozdział poświęcony technologiom – charakterystyka środowiska uruchomieniowego oraz technologii wykorzystanych w implementacji,
- Rozdział poświęcony architekturze systemu – przedstawienie warstw backendu, frontendu oraz bazy danych,
- Rozdział poświęcony implementacji backendu – szczegółowy opis klas serwera i ich funkcjonalności,
- Rozdział poświęcony interfejsowi użytkownika (UI) – prezentacja wyglądu aplikacji.

- Rozdział poświęcony bezpieczeństwu – omówienie mechanizmów autoryzacji, uwierzytelniania oraz walidacji danych,
- Rozdział poświęcony testom i walidacji – opis testów jednostkowych oraz integracyjnych,
- Rozdział końcowy – podsumowanie pracy oraz wskazanie możliwych kierunków dalszego rozwoju systemu.

Struktura i kompilacja projektu MediCare

2.1 Podział na projekty

Projekt **MediCare** został zorganizowany jako rozwiązanie w środowisku *Visual Studio*, zawierające trzy główne projekty:

- **medicare.client** – aplikacja frontendowa napisana w React z TypeScript, odpowiadająca za interfejs użytkownika.
- **MediCare.Server** – aplikacja backendowa w ASP.NET Core, zawierająca kontrolery API, logikę biznesową, konfigurację JWT oraz dostęp do bazy danych.
- **MediCare.ServerTests** – projekt testowy zawierający testy jednostkowe i integracyjne dla warstwy serwerowej.

2.2 Struktura projektu klienta

Projekt `medicare.client` zawiera następujące foldery:

- `src/components` – komponenty interfejsu strony głównej.
- `src/pages` – widoki odpowiadające konkretnym trasom (np. edycja pacjenta, lista wizyt).
- `src/context` – kontekst autoryzacji i zarządzania sesją użytkownika.
- `src/services` – funkcje komunikujące się z API
- `src/interfaces` – definicje typów danych (DTO) używanych w aplikacji.
- `layout` – komponenty układu strony (np. nawigacja, stopka).

Dodatkowo projekt zawiera pliki konfiguracyjne: `vite.config.ts`, `tsconfig.json`, `package.json`, `README.md`.

2.3 Struktura projektu serwera

Projekt `MediCare.Server` zawiera następujące foldery:

- `Controllers` – kontrolery API dla encji takich jak Patients, Doctors, Visits, Rooms.
- `Data` – konfiguracja kontekstu bazy danych (`MediCareDbContext.cs`) oraz seedy.
- `Entities` – definicje encji bazodanowych.
- `Helpers`, `Extensions` – klasy pomocnicze i rozszerzenia.

Główne pliki startowe to `Program.cs`, która konfiguruje aplikację, middleware, autoryzację oraz połączenie z bazą danych.

2.4 Struktura testów

Projekt `MediCare.ServerTests` zawiera następujące foldery:

- `IntegrationTests` – testy integracyjne sprawdzające poprawność działania endpointów kontrolerów w komunikacji z bazą danych.
- `UnitTests` – testy jednostkowe weryfikujące logikę biznesową poszczególnych komponentów systemu.
- `TestInfrastructure` – infrastruktura wspierająca uruchamianie testów integracyjnych, w tym konfiguracja bazy testowej i mockowanie zależności.
- `Helpers` – klasy pomocnicze wykorzystywane do generowania tokenów JWT na potrzeby testów integracyjnych.

2.5 Kompilacja projektu MediCare

Projekt **MediCare** składa się z dwóch głównych części: aplikacji serwerowej (ASP.NET Core) oraz aplikacji klienckiej (React + TypeScript). Kompilacja i uruchomienie projektu odbywa się lokalnie przy użyciu poniższych poleceń:

Backend – MediCare.Server

- Środowisko: .NET 8.0
- Kompilacja: `dotnet build`
- Uruchomienie: `dotnet run`
- Testy: `dotnet test MediCare.ServerTests`

Frontend – medicare.client

- Środowisko: Node.js + Vite
- Instalacja zależności: `npm install`
- Kompilacja i uruchomienie: `npm run dev`
- Budowanie wersji produkcyjnej: `npm run build`

Projekt można uruchomić lokalnie, pod warunkiem poprawnej konfiguracji plików środowiskowych (`.env`) oraz połączenia z bazą danych PostgreSQL.

2.6 Uruchomienie w Dockerze

[3] Alternatywnie frontend (oraz cały projekt) można uruchomić w kontenerach Docker:

1. Pobierz repozytorium:

```
git clone https://github.com/Codyy03/Medicare.git
cd Medicare
```

2. Uruchom kontenery:

```
docker compose up -d
```

3. Dostęp do aplikacji po uruchomieniu:
   - Frontend: `http://localhost:3000`
   - Backend API (Swagger): `https://localhost:7014/swagger`
   - pgAdmin: `http://localhost:8080`
4. Zatrzymanie projektu:

```
docker compose down -v
```

Technologie i narzędzia użyte w projekcie

Projekt **MediCare** został zbudowany z wykorzystaniem następujących technologii:

3.1 Backend

- **ASP.NET Core** – framework do tworzenia API REST.
- **Entity Framework Core** – ORM do komunikacji z bazą danych.
- **JWT (JSON Web Token)** – mechanizm autoryzacji i zarządzania rolami.
- **PostgreSQL** – relacyjna baza danych.
- **xUnit** – framework do testów jednostkowych i integracyjnych.

3.2 Frontend

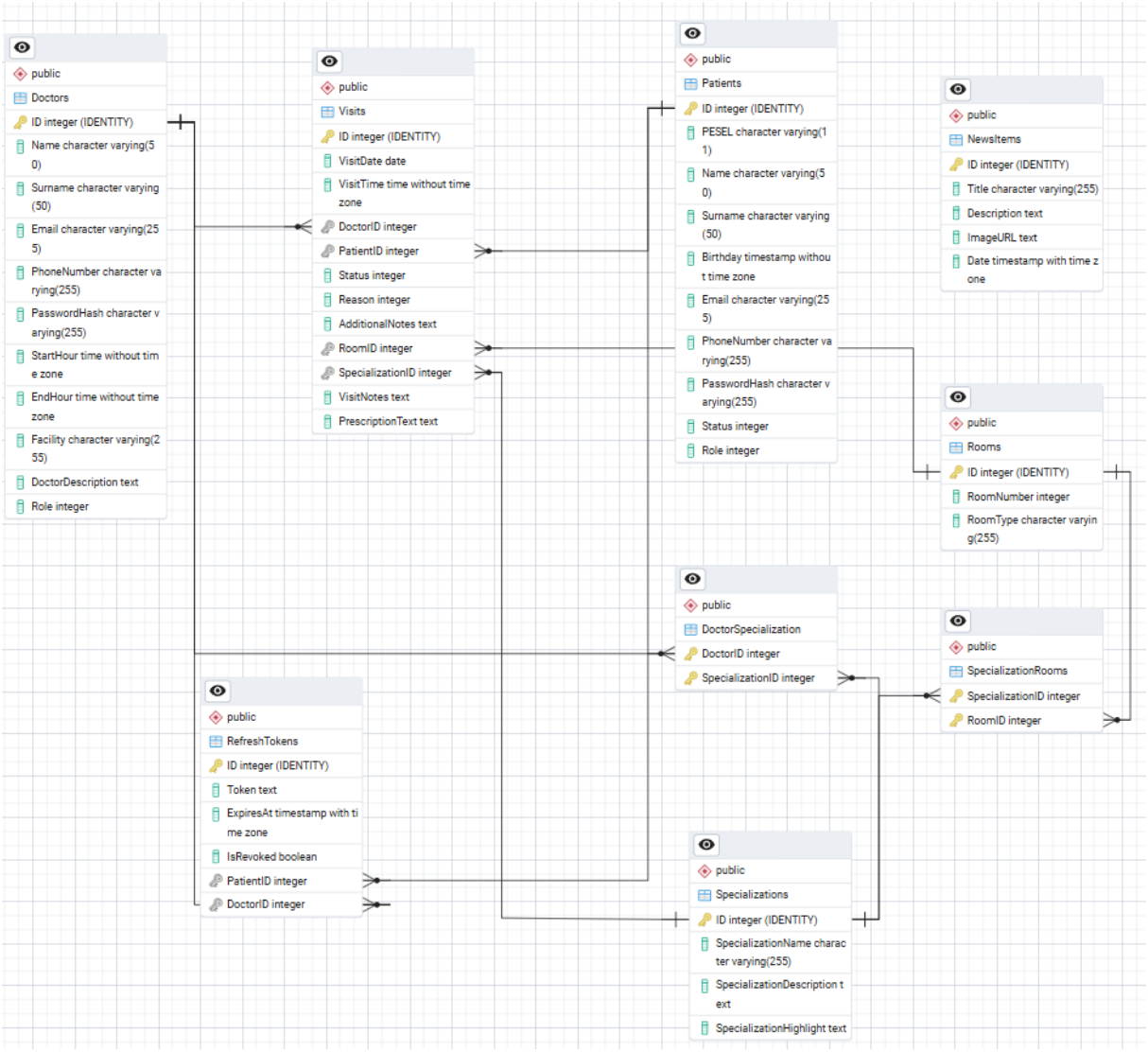
- **React** – biblioteka do budowy interfejsu użytkownika.
- **TypeScript** – typowany język nad JavaScriptem.
- **Bootstrap** – framework CSS do stylowania i responsywności.
- **Axios** – biblioteka do komunikacji z API.
- **Vite** – narzędzie do szybkiego budowania aplikacji frontendowej.

3.3 Narzędzia wspierające

- **V**isual Studio – środowiska programistyczne.
- **Git + GitHub** – kontrola wersji i współpraca zespołowa.
- **GitHub Actions (CI/CD)** – automatyzacja procesu budowania, testowania i wdrażania aplikacji.
- **S**wagger– dokumentacja interfejsu API.
- **D**ocker – konteneryzacja aplikacji.

3.4 Diagram związków encji

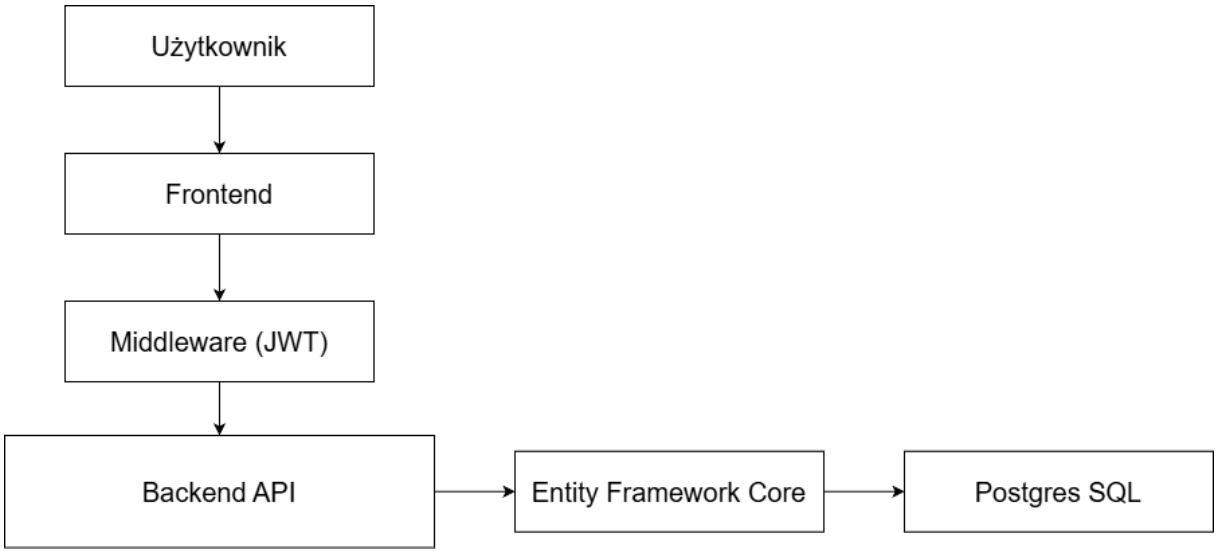
Baza danych powstała w oparciu o relacyjny model PostgreSQL. [6] Na Rys. 3.1 przedstawiono schemat związków encji.



Rys. 3.1: Diagram związków encji w bazie danych MediCare (źródło: własne opracowanie na podstawie PostgreSQL).

3.5 Przepływ danych

Na Rys. 3.2 przedstawiono przepływ danych pomiędzy frontendem, backendem i bazą danych.



Rys. 3.2: Schemat przepływu danych w systemie MediCare(źródło: https://www.drawio.com/).

3.6 Mechanizmy bezpieczeństwa

System wykorzystuje mechanizm **JWT (JSON Web Token)** do autoryzacji użytkowników i zarządzania dostępem do zasobów. Tokeny są generowane podczas logowania i weryfikowane przez middleware przed przekazaniem żądania do kontrolera.

3.7 Dokumentacja API

Do dokumentowania interfejsu API wykorzystano narzędzie **Swagger**, które automatycznie generuje opis dostępnych endpointów, metod HTTP, parametrów oraz odpowiedzi. Umożliwia również testowanie żądań bezpośrednio z poziomu przeglądarki.

Implementacja wybranych metod oraz komponentów

[2], [4] Rozdział ten przedstawia szczegółową implementację wybranych elementów systemu **MediCare**, obejmującą zarówno warstwę backendową, jak i frontendową (komponenty React). Opisano kluczowe metody interfejsu API oraz odpowiadające im komponenty użytkownika, które realizują funkcjonalność systemu.

4.1 MediCareDbContext

Klasa `MediCareDbContext` jest głównym kontekstem bazy danych w systemie MediCare, odpowiedzialnym za mapowanie encji na tabele oraz konfigurację relacji między nimi. Została zaimplementowana przy użyciu `Entity Framework Core`, co umożliwia korzystanie z mechanizmu ORM (Object-Relational Mapping) i ułatwia komunikację z bazą danych PostgreSQL.

Definicja DbSet Kontekst definiuje kolekcje `DbSet` dla wszystkich encji systemu, m.in.: `Patients`, `Doctors`, `Visits`, `Rooms`, `Specializations`, `NewsItems` oraz `RefreshTokens`. Każda z nich odpowiada tabeli w bazie danych i umożliwia wykonywanie operacji CRUD.

Konfiguracja relacji W metodzie `OnModelCreating` skonfigurowano relacje między encjami: - relacje **jeden-do-wielu** (np. lekarz–wizyty, pacjent–wizyty), - relacje **wiele-do-wielu** (np. lekarze–specjalizacje, pokoje–specjalizacje).

Dodatkowo zdefiniowano unikalne indeksy dla kluczowych pól (np. PESEL pacjenta, adres e-mail lekarza, numer pokoju), co zapewnia integralność danych.

Seedowanie danych Kontekst zawiera metody seedujące, które wypełniają bazę przykładowymi danymi początkowymi. Dodano m.in. przykładowe specjalizacje, pokoje, lekarzy, pacjentów, wizyty oraz wiadomości. Dzięki temu system posiada dane testowe umożliwiające szybkie uruchomienie i weryfikację funkcjonalności.

Znaczenie w systemie `MediCareDbContext` stanowi centralny element warstwy danych, zapewniając spójne odwzorowanie modelu domenowego w bazie oraz kontrolę nad relacjami i integralnością danych. Jego konfiguracja umożliwia łatwe rozszerzanie systemu o nowe encje i relacje, a mechanizm seedowania wspiera proces testowania i rozwoju aplikacji.

4.1.1 MediCareDbContextFactory

Klasa `MediCareDbContextFactory` implementuje interfejs `IDesignTimeDbContextFactory<MediCareDbContext>` i służy do tworzenia instancji kontekstu bazy danych w czasie projektowania. Jest wykorzystywana głównie przez narzędzia `Entity Framework Core` podczas generowania i stosowania migracji.

Znaczenie Dzięki tej klasie możliwe jest poprawne odczytanie konfiguracji z pliku `appsettings.json` oraz zainicjalizowanie kontekstu `MediCareDbContext` z odpowiednim connection stringiem. W implementacji użyto dostawcy `Npgsql`, co oznacza, że system korzysta z bazy danych PostgreSQL.

Praktyczne zastosowanie `MediCareDbContextFactory` nie jest wykorzystywana bezpośrednio w kodzie aplikacji, ale stanowi niezbędny element w procesie zarządzania schematem bazy danych. Umożliwia uruchamianie poleceń takich jak `dotnet ef migrations add` czy `dotnet ef database update`, zapewniając spójność konfiguracji pomiędzy środowiskiem uruchomieniowym a projektowym.

4.2 Kontrolery

4.2.1 AdminDoctorsController

Kontroler `AdminDoctorsController` odpowiada za zarządzanie encją `Doctors` przez administratora systemu. Udostępnia metody umożliwiające tworzenie nowych kont lekarzy, aktualizację istniejących rekordów oraz ich pobieranie. Poniżej przedstawiono wybrane operacje:

POST /api/AdminDoctorsRegister Metoda umożliwia utworzenie nowego konta lekarza. Dostępna wyłącznie dla użytkowników z rolą `Admin`. Dane wejściowe są walidowane, a następnie zapisywane w bazie danych przy użyciu `Entity Framework Core`.

PUT /api/AdminDoctors/{id} Metoda pozwala na aktualizację danych lekarza o podanym identyfikatorze. Wymaga autoryzacji użytkownika z rolą `Admin`. Po walidacji przekazanych danych rekord lekarza zostaje zaktualizowany w bazie danych.

4.2.2 AdminNewsController

Kontroler `AdminNewsController` odpowiada za zarządzanie encją `NewsItems` w systemie MediCare. Dostępny jest wyłącznie dla administratora i umożliwia tworzenie oraz aktualizację rekordów wiadomości. W implementacji kontroler korzysta bezpośrednio z kontekstu `MediCareDbContext`, co pozwala na zapis i modyfikację danych w bazie PostgreSQL. Do komunikacji z warstwą prezentacji wykorzystywany jest obiekt transferowy `NewsDto`, który zapewnia separację między encjami a danymi przesyłanymi w żądaniach HTTP.

POST /api/AdminNews/createNews Metoda umożliwia utworzenie nowego wpisu w sekcji wiadomości. Przyjmuje obiekt `NewsDto` zawierający tytuł, opis, adres URL obrazu oraz datę publikacji. Po walidacji i zapisaniu danych w bazie zwracana jest odpowiedź `204 No Content`, co oznacza poprawne utworzenie rekordu.

PUT /api/AdminNews/update/{id} Metoda pozwala na aktualizację istniejącego wpisu wiadomości o podanym identyfikatorze. Jeśli rekord istnieje, jego pola zostają nadpisane wartościami z obiektu `NewsDto`, a następnie zapisane w bazie danych. W przypadku powodzenia zwracana jest odpowiedź `200 OK` wraz z zaktualizowanymi danymi. Jeżeli rekord nie zostanie odnaleziony, metoda zwraca `404 Not Found`.

4.2.3 AdminPatientsController

Kontroler `AdminPatientsController` odpowiada za zarządzanie encją `Patients` w systemie MediCare. Dostęp do metod kontrolera jest ograniczony wyłącznie do użytkowników z rolą `Admin`. Kontroler umożliwia tworzenie nowych kont pacjentów, aktualizację istniejących rekordów oraz ich usuwanie. W implementacji wykorzystano kontekst `MediCareDbContext`, a walidacja haseł została zaimplementowana bezpośrednio w kontrolerze.

POST /api/AdminPatients Metoda umożliwia utworzenie nowego konta pacjenta. Przyjmuje model `AdminPatientRegisterModel`, który zawiera dane osobowe oraz hasło. Przed zapisaniem do bazy wykonywana jest walidacja unikalności numeru PESEL i adresu e-mail, a także kontrola siły hasła. W przypadku powodzenia rekord pacjenta zostaje zapisany w bazie, a odpowiedź zwraca kod `204 No Content`.

PUT /api/AdminPatients/{id} Metoda pozwala na aktualizację danych pacjenta o podanym identyfikatorze. Przyjmuje model `AdminPatientUpdateModel`, który zawiera nowe dane osobowe i status pacjenta. Po walidacji rekord zostaje zaktualizowany w bazie, a odpowiedź zwraca kod `200 OK` wraz z danymi pacjenta.

DELETE /api/AdminPatients/{id} Metoda umożliwia usunięcie pacjenta z systemu. Jeśli rekord o podanym identyfikatorze istnieje, zostaje usunięty z bazy danych, a odpowiedź zwraca kod `204 No Content`. W przypadku braku pacjenta metoda zwraca `404 Not Found`.

4.2.4 AdminRoomsController

Kontroler `AdminRoomsController` odpowiada za zarządzanie encją `Rooms` w systemie MediCare. Dostęp do metod kontrolera jest ograniczony wyłącznie do użytkowników z rolą `Admin`. Kontroler umożliwia tworzenie nowych pokoi, ich aktualizację, usuwanie oraz pobieranie szczegółowych informacji wraz z przypisanymi specjalizacjami. W implementacji wykorzystano kontekst `MediCareDbContext`, a dane przesyłane w żądaniach są reprezentowane przez obiekty transferowe `RoomDto` oraz `SpecializationDto`.

POST /api/AdminRooms Metoda umożliwia utworzenie nowego pokoju wraz z przypisanymi specjalizacjami. Przyjmuje obiekt `RoomDto`, który zawiera numer pokoju, typ

oraz listę specjalizacji. Przed zapisaniem do bazy wykonywana jest walidacja unikalności numeru pokoju oraz sprawdzenie, czy numer jest dodatni. W przypadku powodzenia rekord zostaje zapisany w bazie, a odpowiedź zwraca kod `204 No Content`.

PUT /api/AdminRooms Metoda pozwala na aktualizację istniejącego pokoju. Przyjmuje obiekt `RoomDto` zawierający nowe dane pokoju oraz listę specjalizacji. Po walidacji numeru pokoju i sprawdzeniu jego unikalności rekord zostaje zaktualizowany w bazie, a przypisane specjalizacje są odświeżane. W przypadku powodzenia metoda zwraca kod `204 No Content`, natomiast brak pokoju skutkuje odpowiedzią `404 Not Found`.

GET /api/AdminRooms/{id} Metoda umożliwia pobranie szczegółowych informacji o pokoju na podstawie identyfikatora. Zwraca obiekt `RoomDto` zawierający numer pokoju, typ oraz listę przypisanych specjalizacji. W przypadku braku pokoju metoda zwraca odpowiedź `404 Not Found`.

4.2.5 SpecializationsController

Kontroler `SpecializationsController` odpowiada za zarządzanie encją `Specializations` w systemie MediCare. Dostęp do metod kontrolera jest ograniczony wyłącznie do użytkowników z rolą `Admin`. Kontroler umożliwia tworzenie nowych specjalizacji, ich aktualizację, usuwanie oraz pobieranie szczegółowych informacji. Dane przesyłane w żądaniach są reprezentowane przez obiekty transferowe `SpecializationDto`, `SpecializationCreateDto` oraz `SpecializationUpdateDto`.

POST /api/admin/Specializations Metoda umożliwia utworzenie nowej specjalizacji. Przyjmuje obiekt `SpecializationCreateDto`, który zawiera nazwę, wyróżnik oraz opis specjalizacji. Przed zapisaniem do bazy wykonywana jest walidacja unikalności nazwy. W przypadku powodzenia rekord zostaje zapisany w bazie, a odpowiedź zwraca kod `201 Created` wraz z utworzoną specjalizacją.

PUT /api/admin/Specializations/{id} Metoda pozwala na aktualizację istniejącej specjalizacji o podanym identyfikatorze. Przyjmuje obiekt `SpecializationUpdateDto`, który zawiera nowe dane. Po walidacji rekord zostaje zaktualizowany w bazie, a odpowiedź zwraca kod `204 No Content`. W przypadku braku specjalizacji metoda zwraca `404 Not Found`.

GET /api/admin/Specializations/{id} Metoda umożliwia pobranie szczegółowych informacji o specjalizacji na podstawie identyfikatora. Zwraca obiekt `SpecializationDto` zawierający nazwę, wyróżnik oraz opis specjalizacji. W przypadku braku rekordu metoda zwraca odpowiedź `404 Not Found`.

4.2.6 AuthController

Kontroler `AuthController` odpowiada za proces uwierzytelniania oraz zarządzanie tokenami w systemie MediCare. Jego głównym zadaniem jest odświeżanie tokenów JWT na podstawie poprawnych i nieprzeterminowanych tokenów odświeżających. Dzięki temu

użytkownicy mogą utrzymać aktywną sesję bez konieczności ponownego logowania. Kontroler korzysta z kontekstu `MediCareDbContext` oraz pomocniczej klasy `JwtTokenHelper`, która generuje nowe tokeny.

POST /api/Auth/refresh Metoda umożliwia odświeżenie pary tokenów (access token oraz refresh token). Przyjmuje obiekt `RefreshRequestDto`, zawierający token odświeżający. Jeśli token jest poprawny, nieprzeterminowany i należy do istniejącego użytkownika (pacjenta lub lekarza), generowana jest nowa para tokenów. W przypadku powodzenia metoda zwraca odpowiedź `200 OK` wraz z nowym access tokenem i refresh tokenem. Jeżeli token jest niepoprawny, wygasł lub nie należy do żadnego użytkownika, metoda zwraca `401 Unauthorized`.

4.2.7 DoctorsController

Kontroler `DoctorsController` odpowiada za zarządzanie encją `Doctors` w systemie MediCare. Udostępnia metody umożliwiające rejestrację nowych lekarzy, logowanie, pobieranie profilu zalogowanego użytkownika oraz aktualizację danych. Dodatkowo kontroler obsługuje operacje filtrowania lekarzy według specjalizacji, nazwiska czy godzin dostępności. W implementacji wykorzystano kontekst `MediCareDbContext` oraz klasę pomocniczą `JwtTokenHelper` do generowania tokenów JWT.

POST /api/Doctors/register Metoda umożliwia rejestrację nowego lekarza. Przyjmuje obiekt `DoctorRegisterDto`, który zawiera dane osobowe, hasło oraz listę identyfikatorów specjalizacji. Przed zapisaniem do bazy wykonywana jest walidacja unikalności adresu e-mail, sprawdzenie poprawności hasła oraz weryfikacja przypisanych specjalizacji. W przypadku powodzenia rekord zostaje zapisany w bazie, a odpowiedź zwraca kod `201 Created` wraz z podstawowymi danymi lekarza.

POST /api/Doctors/login Metoda obsługuje proces logowania lekarza. Przyjmuje obiekt `LoginDto` zawierający adres e-mail i hasło. Po poprawnej weryfikacji danych generowany jest token JWT oraz token odświeżający, które umożliwiają dalszą autoryzację w systemie. W przypadku niepoprawnych danych metoda zwraca odpowiedź `401 Unauthorized`.

GET /api/Doctors/me Metoda umożliwia pobranie profilu zalogowanego lekarza na podstawie identyfikatora użytkownika z tokena JWT. Zwraca obiekt `DoctorDto` zawierający dane osobowe oraz listę przypisanych specjalizacji. W przypadku braku autoryzacji metoda zwraca `401 Unauthorized`, a w przypadku braku rekordu `404 Not Found`.

PUT /api/Doctors/update Metoda pozwala na aktualizację danych zalogowanego lekarza. Przyjmuje obiekt `DoctorUpdateDto`, który zawiera nowe dane osobowe, godziny pracy oraz opcjonalnie informacje o placówce i opisie lekarza. Przed zapisaniem wykonywana jest walidacja numeru telefonu, imienia, nazwiska oraz godzin pracy. W przypadku powodzenia rekord zostaje zaktualizowany w bazie, a odpowiedź zwraca kod `200 OK` wraz z zaktualizowanymi danymi.

PUT /api/Doctors/password-reset Metoda umożliwia zmianę hasła zalogowanego lekarza. Przyjmuje obiekt `PasswordResetDto`, który zawiera stare i nowe hasło. Przed zapisaniem wykonywana jest weryfikacja starego hasła oraz walidacja nowego. W przypadku powodzenia metoda zwraca kod `204 No Content`, natomiast w przypadku błędów odpowiednio `400 Bad Request`, `401 Unauthorized` lub `404 Not Found`.

4.2.8 PatientsController

Kontroler `PatientsController` odpowiada za zarządzanie encją `Patients` w systemie MediCare. Udostępnia metody umożliwiające rejestrację nowych pacjentów, logowanie, pobieranie profilu zalogowanego użytkownika, aktualizację danych oraz operacje związane z bezpieczeństwem konta (reset hasła, dezaktywacja). W implementacji wykorzystano kontekst `MediCareDbContext` oraz klasę pomocniczą `JwtTokenHelper` do generowania tokenów JWT.

POST /api/Patients/register Metoda umożliwia rejestrację nowego pacjenta. Przyjmuje obiekt `PatientRegisterDto`, który zawiera dane osobowe, numer PESEL, adres e-mail, numer telefonu oraz hasło. Przed zapisaniem do bazy wykonywana jest walidacja unikalności numeru PESEL i adresu e-mail. W przypadku powodzenia rekord zostaje zapisany w bazie, a odpowiedź zwraca kod `201 Created` wraz z podstawowymi danymi pacjenta.

POST /api/Patients/login Metoda obsługuje proces logowania pacjenta. Przyjmuje obiekt `LoginDto` zawierający adres e-mail i hasło. Po poprawnej weryfikacji danych generowany jest token JWT oraz token odświeżający, które umożliwiają dalszą autoryzację w systemie. W przypadku niepoprawnych danych lub nieaktywnego konta metoda zwraca odpowiedź `401 Unauthorized`.

GET /api/Patients/me Metoda umożliwia pobranie profilu zalogowanego pacjenta na podstawie identyfikatora użytkownika z tokena JWT. Zwraca obiekt `PatientDto` zawierający dane osobowe, numer PESEL oraz status konta. W przypadku braku autoryzacji metoda zwraca `401 Unauthorized`, a w przypadku braku rekordu `404 Not Found`.

PUT /api/Patients/update Metoda pozwala na aktualizację danych zalogowanego pacjenta. Przyjmuje obiekt `PatientUpdateDto`, który zawiera nowe dane osobowe, numer PESEL, datę urodzenia oraz numer telefonu. Przed zapisaniem wykonywana jest walidacja imienia, nazwiska, numeru PESEL oraz daty urodzenia. W przypadku powodzenia rekord zostaje zaktualizowany w bazie, a odpowiedź zwraca kod `200 OK` wraz z zaktualizowanymi danymi.

PUT /api/Patients/password-reset Metoda umożliwia zmianę hasła zalogowanego pacjenta. Przyjmuje obiekt `PasswordResetDto`, który zawiera stare i nowe hasło. Przed zapisaniem wykonywana jest weryfikacja starego hasła oraz walidacja nowego. W przypadku powodzenia metoda zwraca kod `204 No Content`, natomiast w przypadku błędów odpowiednio `400 Bad Request`, `401 Unauthorized` lub `404 Not Found`.

PUT /api/Patients/deactivate Metoda umożliwia dezaktywację konta pacjenta. Przyjmuje obiekt `PasswordDto` zawierający aktualne hasło pacjenta. Po poprawnej weryfikacji hasła status konta zostaje ustawiony na `Inactive`, a odpowiedź zwraca kod `200 OK` wraz z komunikatem o dezaktywacji.

DELETE /api/Patients/{id} Metoda umożliwia usunięcie pacjenta z systemu na podstawie identyfikatora. W przypadku powodzenia rekord zostaje usunięty z bazy, a odpowiedź zwraca kod `204 No Content`. Jeżeli pacjent nie istnieje, metoda zwraca `404 Not Found`.

4.2.9 NewsController

Kontroler `NewsController` odpowiada za zarządzanie Newsami w systemie MediCare. Udostępnia metody umożliwiające pobieranie wszystkich wiadomości, filtrowanie ich według daty, tworzenie nowych rekordów, aktualizację istniejących oraz usuwanie. W implementacji wykorzystano kontekst `MediCareDbContext`, który zapewnia bezpośrednią komunikację z bazą danych.

GET /api/News Metoda umożliwia pobranie wszystkich wiadomości z systemu. Zwraca listę obiektów `NewsItem` zawierających tytuł, opis, datę oraz opcjonalny adres URL obrazu.

GET /api/News/by-date Metoda pozwala na filtrowanie wiadomości według miesiąca i roku oraz sortowanie wyników w kolejności rosnącej lub malejącej. Dzięki temu użytkownik może uzyskać listę wiadomości dopasowaną do określonych kryteriów czasowych. W przypadku powodzenia metoda zwraca kod `200 OK` wraz z listą wiadomości.

GET /api/News/latest/{count} Metoda umożliwia pobranie najnowszych wiadomości w systemie. Parametr `count` określa maksymalną liczbę rekordów do zwrócenia. Wyniki są sortowane malejąco według daty publikacji, a odpowiedź zawiera listę obiektów `NewsItem`.

POST /api/News Metoda umożliwia utworzenie nowego wpisu w sekcji wiadomości. Przyjmuje obiekt `NewsItem` zawierający tytuł, opis, datę oraz opcjonalny adres URL obrazu. Po poprawnej walidacji rekord zostaje zapisany w bazie, a odpowiedź zwraca kod `201 Created` wraz z utworzonym wpisem.

PUT /api/News/{id} Metoda pozwala na aktualizację istniejącego wpisu wiadomości o podanym identyfikatorze. Jeśli rekord istnieje, jego pola zostają nadpisane wartościami z obiektu `NewsItem`. W przypadku powodzenia metoda zwraca kod `204 No Content`, natomiast brak rekordu skutkuje odpowiedzią `404 Not Found`.

DELETE /api/News/{id} Metoda umożliwia usunięcie wiadomości z systemu na podstawie identyfikatora. W przypadku powodzenia rekord zostaje usunięty z bazy, a odpowiedź zwraca kod `204 No Content`. Jeżeli wiadomość nie istnieje, metoda zwraca `404 Not Found`.

4.2.10 SpecializationsController

Kontroler `SpecializationsController` odpowiada za zarządzanie encją `Specializations` w systemie MediCare. Udostępnia metody umożliwiające pobieranie wszystkich specjalizacji, pobieranie uproszczonych list nazw i wyróżników, tworzenie nowych rekordów, aktualizację istniejących oraz usuwanie. W implementacji wykorzystano kontekst `MediCareDbContext`, który zapewnia bezpośrednią komunikację z bazą danych.

GET /api/Specializations Metoda umożliwia pobranie wszystkich specjalizacji dostępnych w systemie. Zwraca listę obiektów `Specialization` zawierających pełne informacje o specjalizacji.

GET /api/Specializations/specializationsNames Metoda zwraca uproszczoną listę specjalizacji w postaci identyfikatora i nazwy. Zwraca listę obiektów `SpecializationsNamesID`, co pozwala na szybkie wyświetlanie nazw w interfejsie użytkownika.

GET /api/Specializations/highlights Metoda umożliwia pobranie wyróżników specjalizacji, tj. nazwy oraz krótkiego opisu. Zwraca listę obiektów `SpecializationHighlightDto`, które mogą być wykorzystane np. w sekcji podsumowań na stronie głównej.

POST /api/Specializations Metoda umożliwia utworzenie nowej specjalizacji. Przyjmuje obiekt `Specialization` zawierający nazwę, wyróżnik oraz opis. Po poprawnej walidacji rekord zostaje zapisany w bazie, a odpowiedź zwraca kod `201 Created` wraz z utworzoną specjalizacją.

PUT /api/Specializations/{id} Metoda pozwala na aktualizację istniejącej specjalizacji o podanym identyfikatorze. Jeśli rekord istnieje, jego pola zostają nadpisane wartościami z obiektu `Specialization`. W przypadku powodzenia metoda zwraca kod `204 No Content`, natomiast brak rekordu skutkuje odpowiedzią `404 Not Found`.

DELETE /api/Specializations/{id} Metoda umożliwia usunięcie specjalizacji z systemu na podstawie identyfikatora. W przypadku powodzenia rekord zostaje usunięty z bazy, a odpowiedź zwraca kod `204 No Content`. Jeżeli specjalizacja nie istnieje, metoda zwraca `404 Not Found`.

4.2.11 VisitsController

Kontroler `VisitsController` odpowiada za zarządzanie encją `Visits` w systemie MediCare. Udostępnia metody umożliwiające planowanie nowych wizyt, pobieranie szczegółowych informacji o wizytach, aktualizację ich statusu oraz anulowanie. Dodatkowo kontroler obsługuje logikę sprawdzania dostępności pokoi dla wybranych specjalizacji i lekarzy. W implementacji wykorzystano kontekst `MediCareDbContext` oraz klasę pomocniczą `JwtTokenHelper` do obsługi autoryzacji.

GET /api/Visits Metoda umożliwia pobranie wszystkich wizyt z systemu. Zwraca listę obiektów `VisitResponseDto`, które zawierają szczegółowe informacje o dacie, godzinie, lekarzu, pacjencie, specjalizacji, pokoju oraz statusie wizyty.

GET /api/Visits/{id} Metoda pozwala na pobranie szczegółowych informacji o pojedynczej wizycie na podstawie identyfikatora. Zwraca obiekt `VisitResponseDto` zawierający dane lekarza, pacjenta, pokoju, specjalizacji oraz dodatkowe notatki i receptę. W przypadku braku rekordu metoda zwraca `404 Not Found`.

POST /api/Visits Metoda umożliwia utworzenie nowej wizyty pacjenta. Przyjmuje obiekt `VisitCreateDto`, który zawiera identyfikatory lekarza, pacjenta, pokoju i specjalizacji, datę oraz godzinę wizyty. Przed zapisaniem wykonywana jest walidacja: sprawdzenie konfliktów w harmonogramie lekarza i pokoju, poprawności przypisania pokoju do specjalizacji oraz poprawności powodu wizyty. W przypadku powodzenia rekord zostaje zapisany w bazie, a odpowiedź zwraca kod `201 Created` wraz z utworzoną wizytą.

PUT /api/Visits/update/{id} Metoda pozwala na aktualizację istniejącej wizyty. Przyjmuje obiekt `VisitEditDto`, który zawiera nowe dane dotyczące daty, godziny, statusu, powodu oraz dodatkowych notatek. W przypadku powodzenia metoda zwraca kod `204 No Content`, natomiast brak wizyty skutkuje odpowiedzią `404 Not Found`.

PUT /api/Visits/startVisit/{id} Metoda umożliwia oznaczenie wizyty jako zakończonej. Przyjmuje obiekt `StartVisitDto`, który zawiera notatki oraz treść recepty. Po zapisaniu danych status wizyty zmienia się na `Completed`, a odpowiedź zwraca kod `200 OK`.

POST /api/Visits/canceledVisit/{id} Metoda umożliwia anulowanie zaplanowanej wizyty pacjenta. Jeśli wizyta istnieje i nie została wcześniej zakończona ani anulowana, jej status zostaje zmieniony na `Cancelled`. W przypadku powodzenia metoda zwraca kod `200 OK` wraz z zaktualizowanymi danymi wizyty.

GET /api/Visits/freeRoomsForDay/{specId} Metoda pozwala na sprawdzenie dostępnych pokoi dla wybranej specjalizacji i lekarza w danym dniu. Zwraca słownik, w którym kluczem jest przedział czasowy, a wartością lista wolnych pokoi (`RoomDto`). Dzięki temu możliwe jest planowanie wizyt z uwzględnieniem dostępności infrastruktury.

4.2.12 VisitStatisticsController

Kontroler `VisitStatisticsController` odpowiada za generowanie danych statystycznych dotyczących wizyt w systemie MediCare.

GET /api/VisitStatistics/stats/summary Metoda zwraca zagregowane statystyki wszystkich wizyt w systemie. Zwraca całkowitą liczbę wizyt oraz podział według statusów: `Scheduled`, `Completed`, `Cancelled`.

GET /api/VisitStatistics/stats/daily Metoda umożliwia pobranie dziennych statystyk wizyt z ostatnich N dni (domyślnie 30). Dane są grupowane według daty i zawierają liczbę wizyt w podziale na statusy.

GET /api/VisitStatistics/stats/top-doctors Metoda zwraca ranking lekarzy według liczby przeprowadzonych wizyt (z wyłączeniem anulowanych). Dla każdego lekarza podawane są: imię i nazwisko, całkowita liczba wizyt oraz podział na statusy `Completed` i `Scheduled`.

GET /api/VisitStatistics/stats/specializations Metoda zwraca statystyki wizyt w podziale na specjalizacje (z wyłączeniem anulowanych). Dla każdej specjalizacji podawane są: nazwa, całkowita liczba wizyt oraz podział na statusy `Completed` i `Scheduled`.

4.2.13 JwtTokenHelper

Klasa pomocnicza `JwtTokenHelper` odpowiada za generowanie tokenów JWT oraz tokenów odświeżających w systemie MediCare. Jej głównym zadaniem jest zapewnienie mechanizmu uwierzytelniania i autoryzacji użytkowników (pacjentów i lekarzy) poprzez bezpieczne przekazywanie informacji o tożsamości i roli użytkownika w postaci zaszyfrowanego tokena.

Metoda GenerateJwtToken Metoda `GenerateJwtToken` tworzy podpisany token JWT zawierający zestaw `claims`, czyli informacji o użytkowniku: identyfikator, adres e-mail, imię oraz rolę (np. `Patient`, `Doctor`). Token jest podpisywany algorytmem `HmacSha256` przy użyciu klucza symetrycznego zdefiniowanego w konfiguracji aplikacji. Czas życia tokena został ustawiony na jedną godzinę, co ogranicza ryzyko nadużyć i wymusza cykliczne odświeżanie sesji.

Metoda GenerateRefreshToken Metoda `GenerateRefreshToken` generuje losowy ciąg znaków o długości 64 bajtów, zakodowany w formacie Base64. Token odświeżający służy do uzyskania nowej pary tokenów (JWT + refresh token) bez konieczności ponownego logowania. Dzięki temu użytkownik może utrzymać aktywną sesję w systemie, a jednocześnie zachowana jest wysoka odporność na ataki typu brute-force.

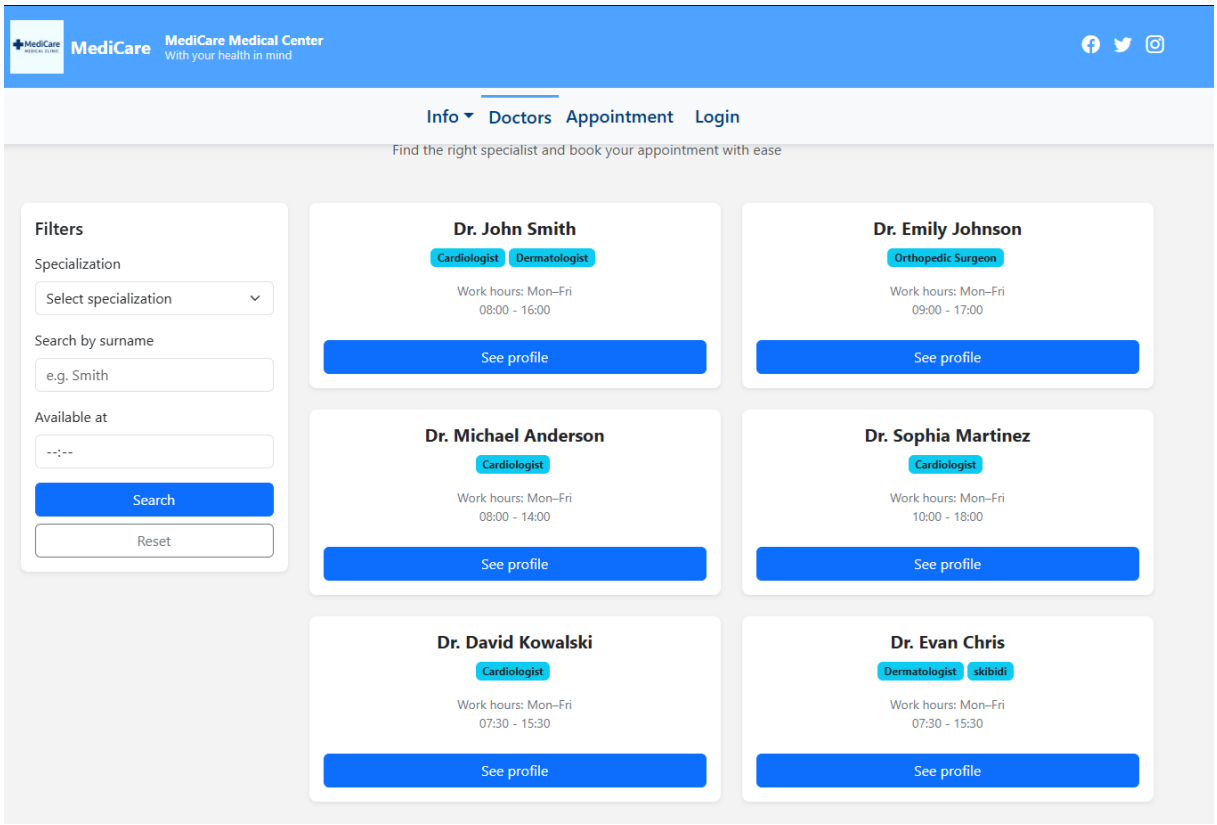
Znaczenie w systemie Klasa `JwtTokenHelper` stanowi centralny element mechanizmu bezpieczeństwa w systemie MediCare. Zapewnia spójny sposób generowania tokenów, kontrolę czasu ich ważności oraz obsługę odświeżania sesji. Rozwiązanie to umożliwia bezpieczne zarządzanie dostępem do zasobów API i jest zgodne z dobrymi praktykami w zakresie uwierzytelniania aplikacji webowych.

Interfejs użytkownika

[5], [7], [8] W tej części dokumentacji przedstawiono interfejs aplikacji **MediCare** z perspektywy różnych typów użytkowników systemu. Każdy z nich posiada odmienny zakres funkcjonalności oraz dostęp do danych.

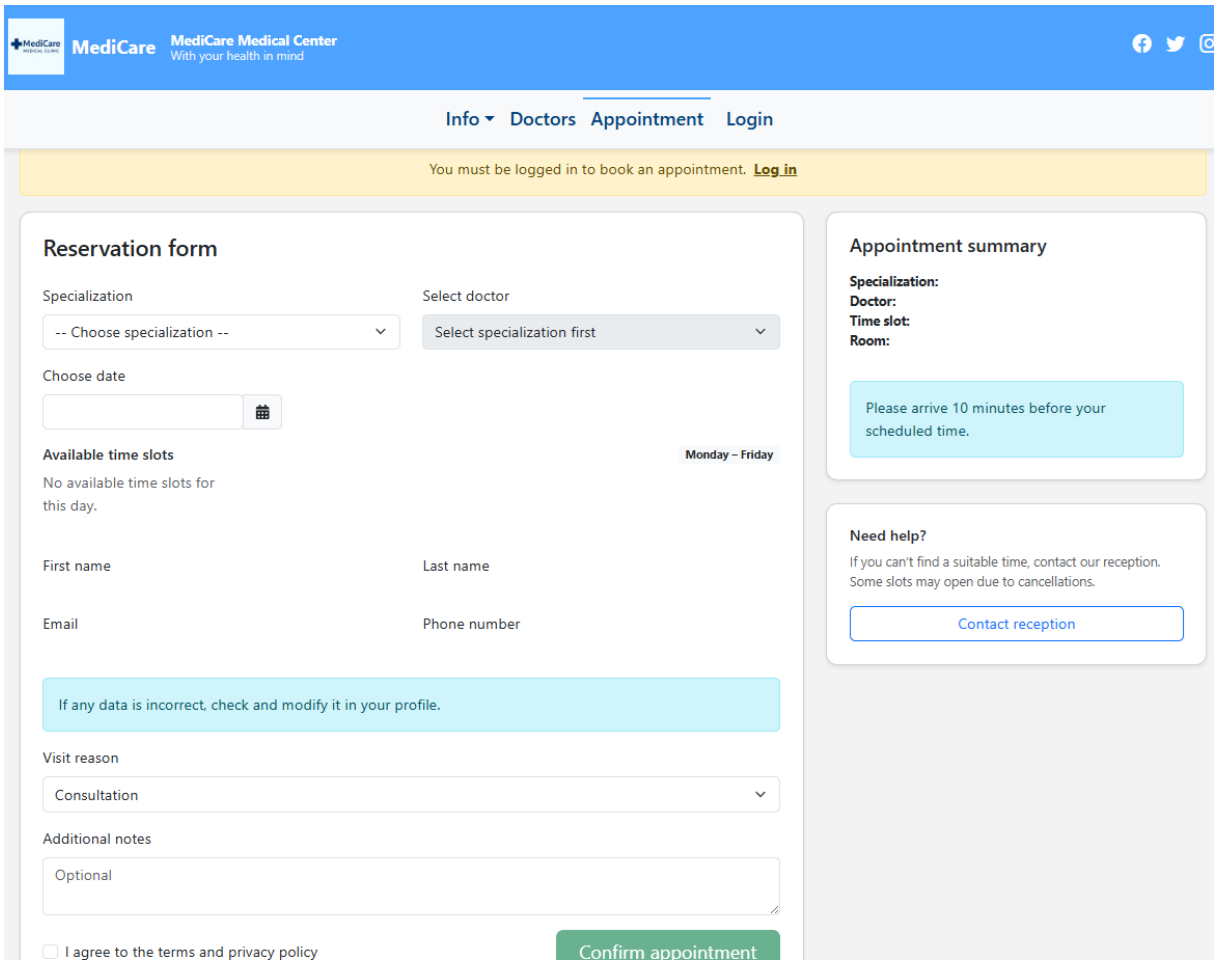
5.1 Użytkownik niezalogowany

1. Strona główna i informacje o systemie.
2. Formularz logowania i rejestracji.
3. Lista doktorów – możliwość podglądnięcia profili lub wyszukiwania. Rys. 5.1



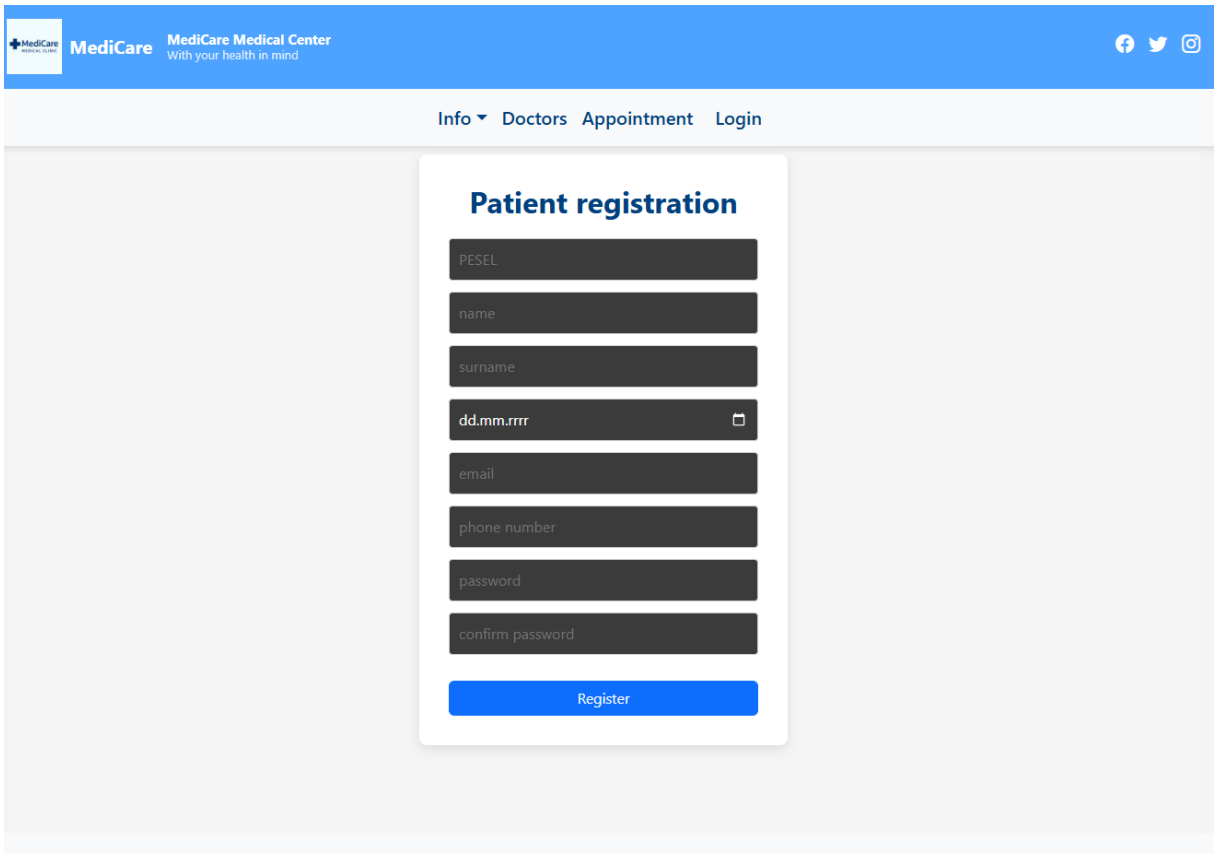
Rys. 5.1: Spis doktorów.

4. Formularz umawiania wizyty – wymaga zalogowania. Rys. 5.2



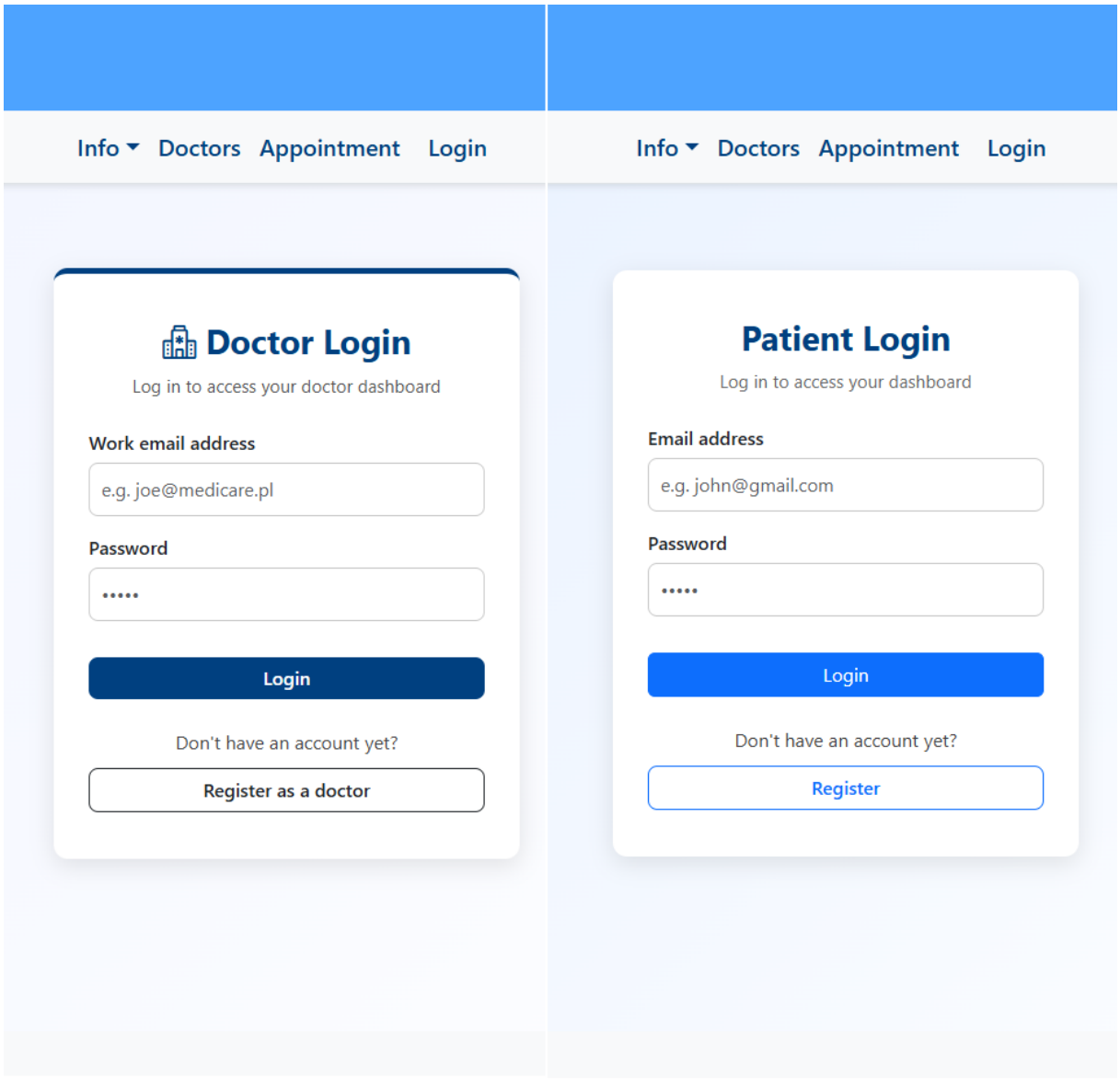
Rys. 5.2: Formularz tworzenia wizyty.

5. Formularz rejestracji pacjenta. Rys. 5.3



Rys. 5.3: Formularz rejestracji pacjenta.

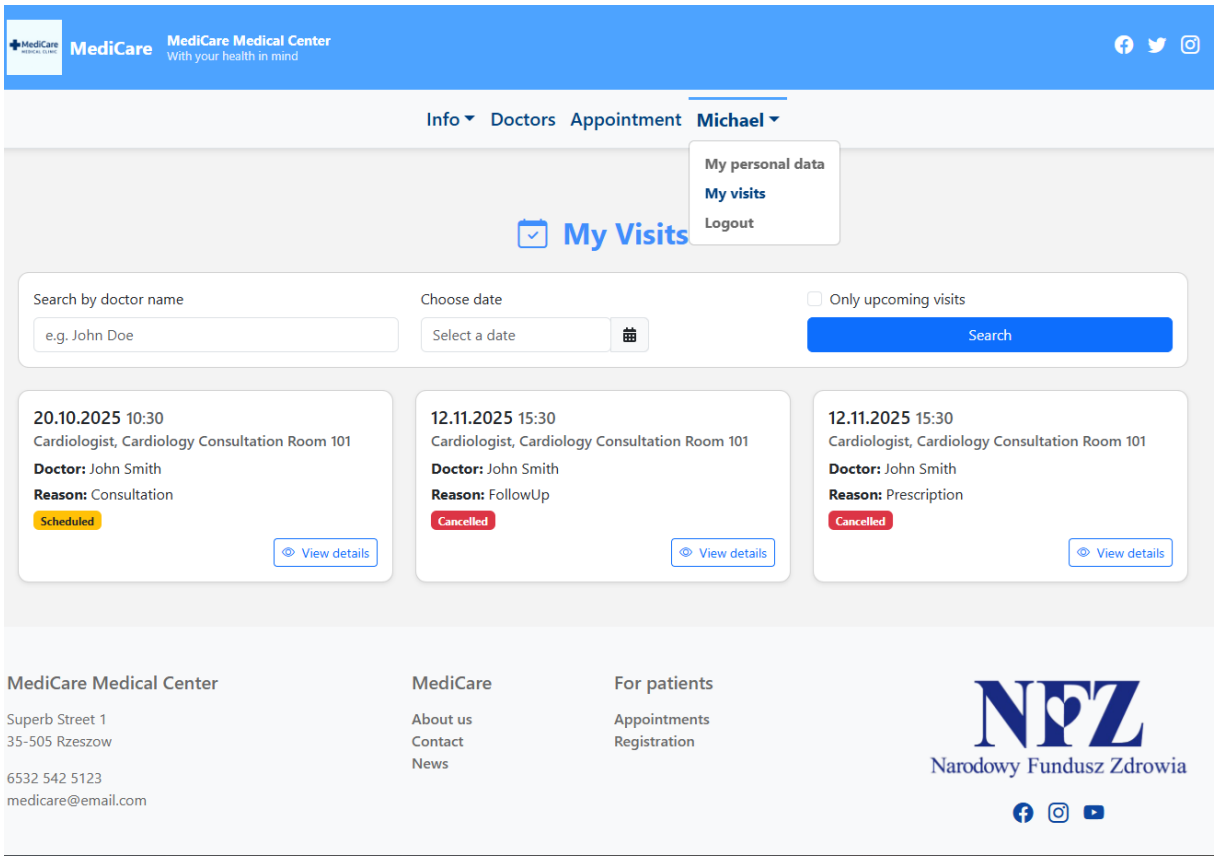
6. Formularz logowania dla doktora i pacjenta. Rys. 5.4



Rys. 5.4: Formularz logowania.

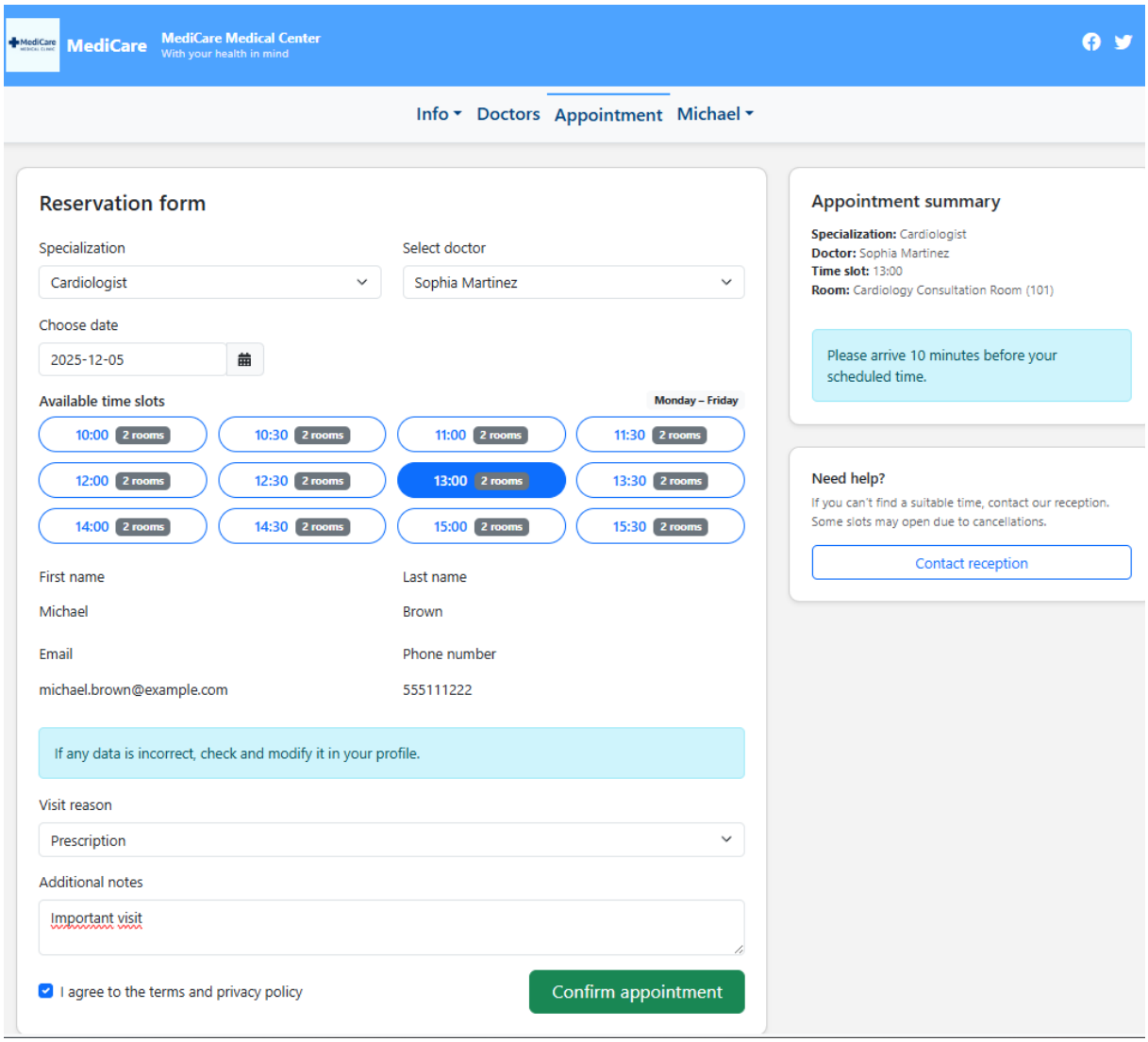
5.2 Pacjent

1. Panel pacjenta – lista wizyt z filtracją (doktor, data, nadchodzące). Rys. 5.5



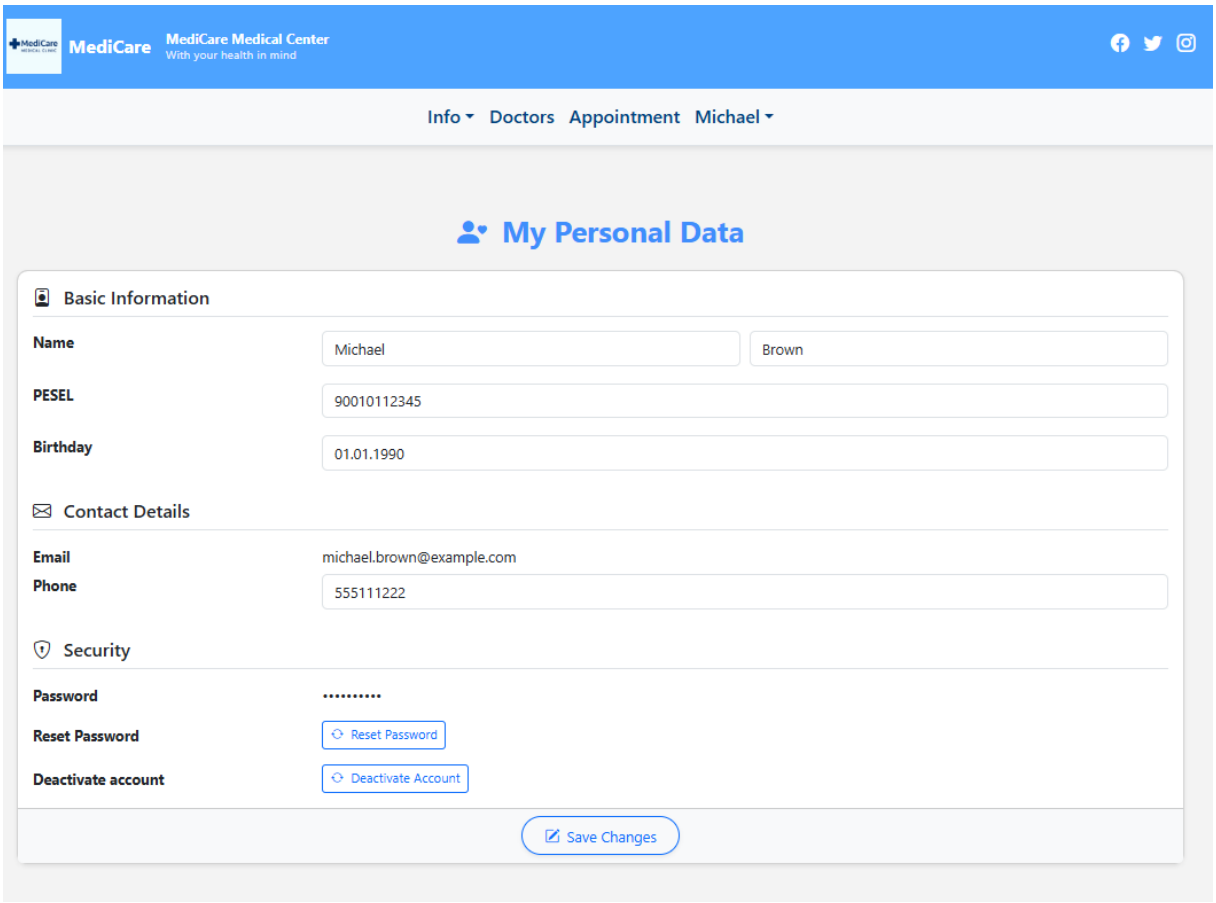
Rys. 5.5: Lista wizyt pacjenta.

2. Rezerwacja nowych wizyt. Rys. 5.6



Rys. 5.6: Tworzenie nowej wizyty.

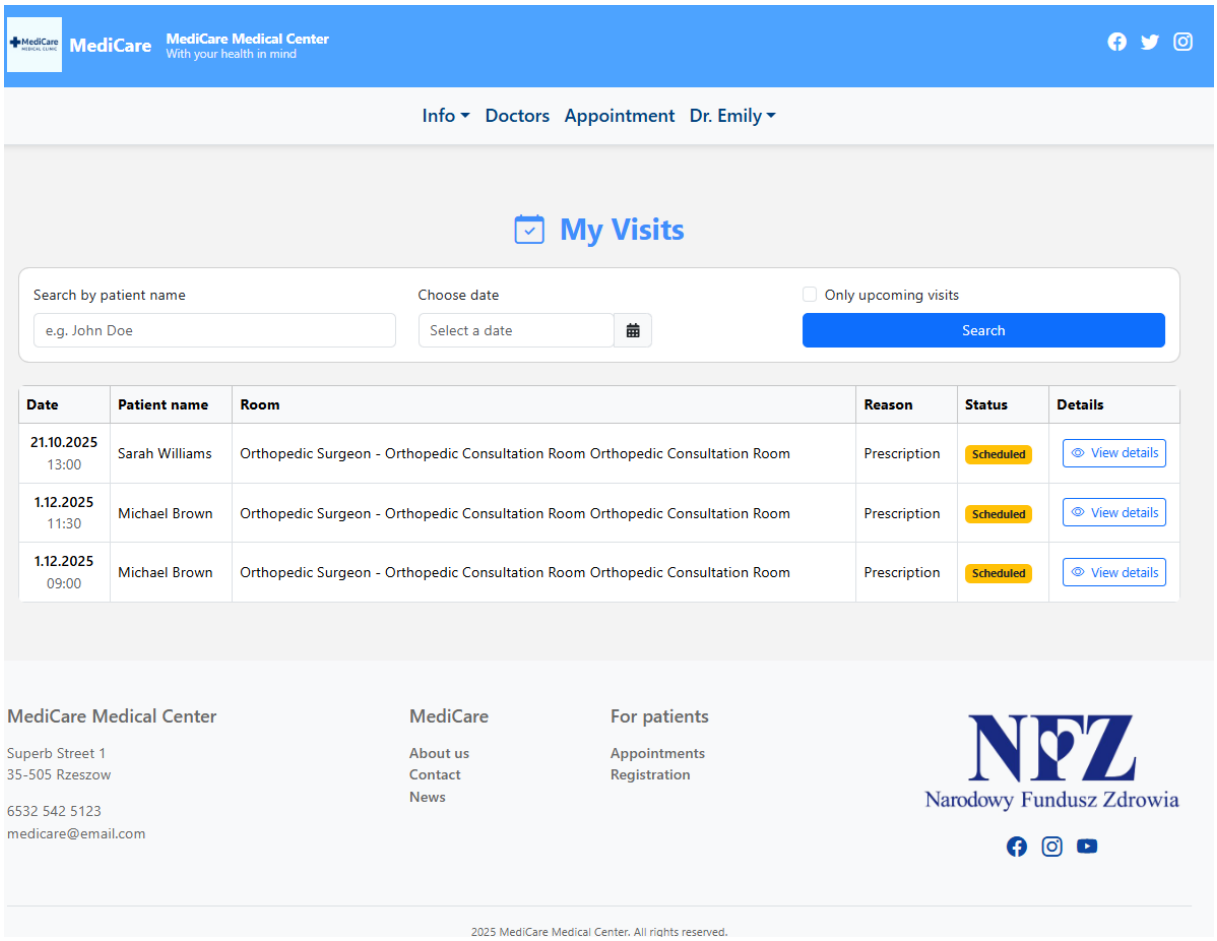
3. Edycja danych osobowych i ustawień konta. Rys. 5.7



Rys. 5.7: Edycja profilu pacjenta.

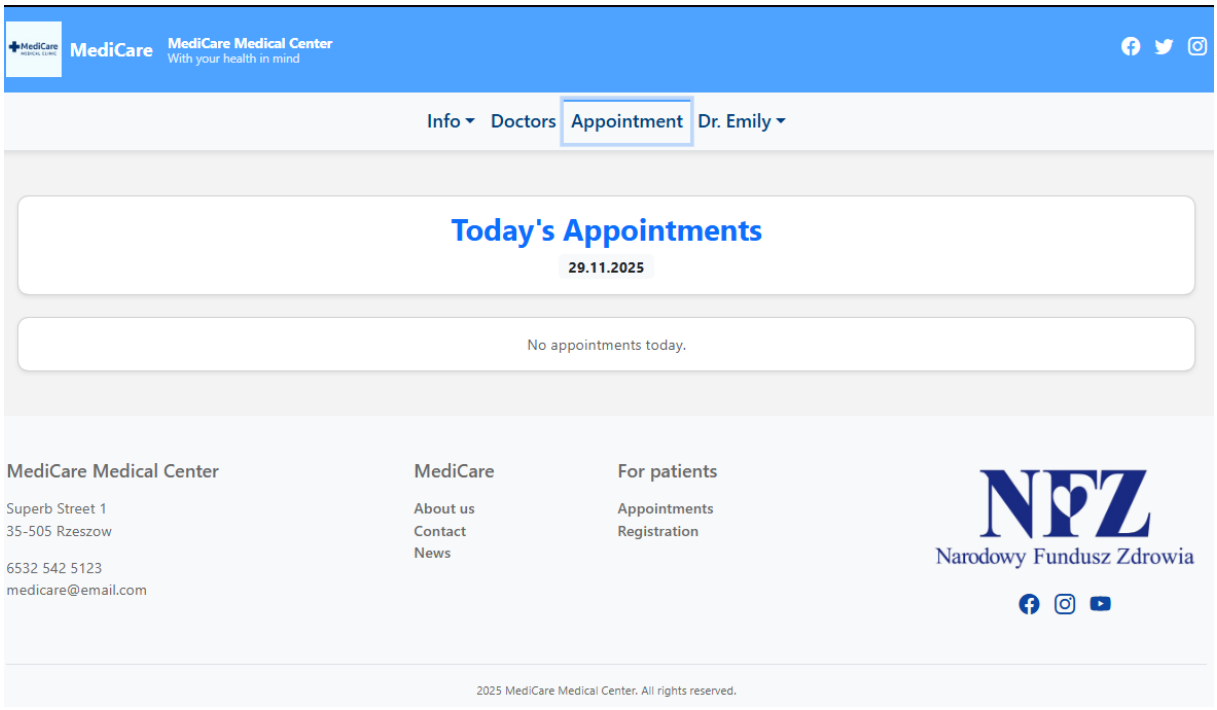
5.3 Doktor

1. Harmonogram wizyt – pełna lista z możliwością edycji. Rys. 5.8



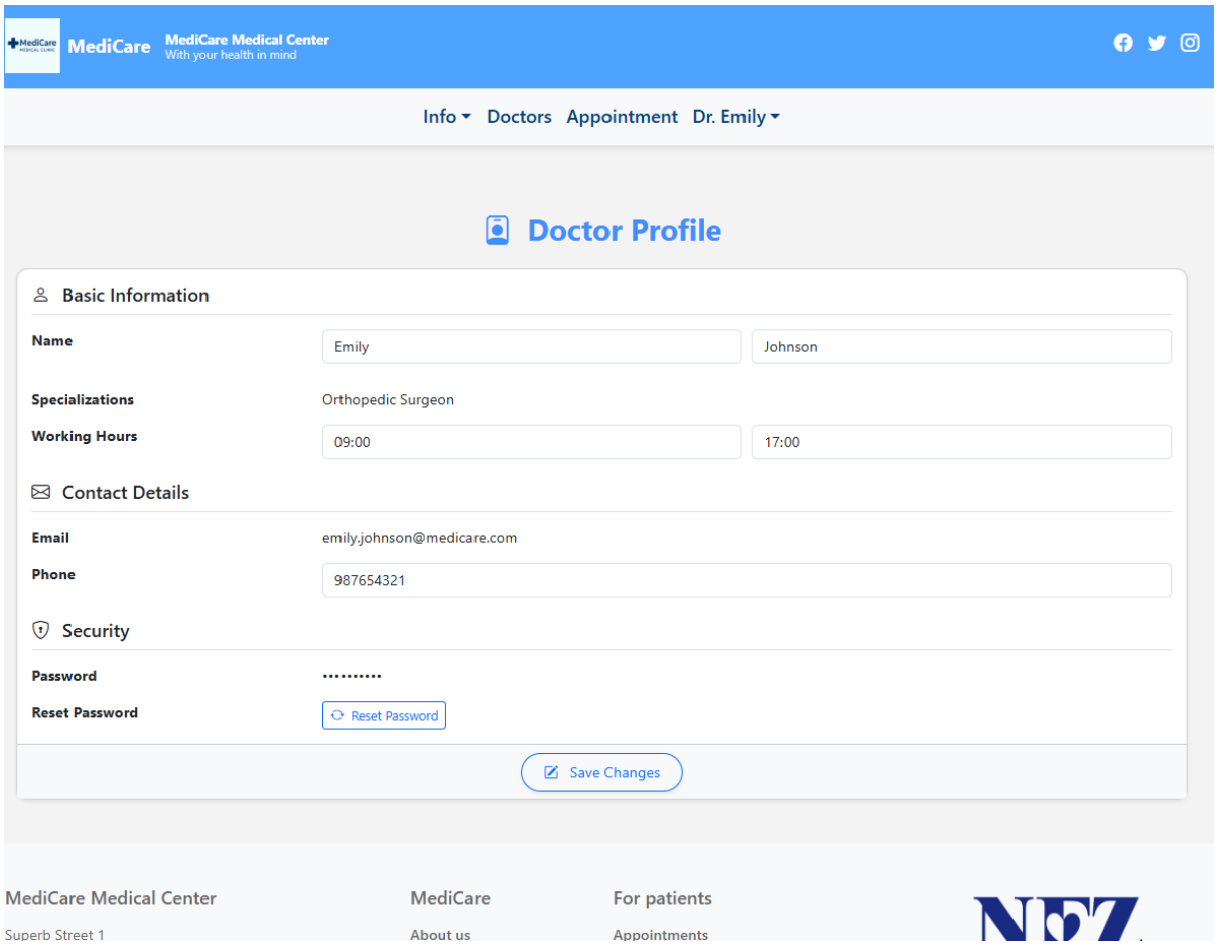
Rys. 5.8: Harmonogram wizyt.

2. Wizyty przewidziane na aktualny dzień. Rys. 5.9



Rys. 5.9: Lista wizyt na aktualny dzień.

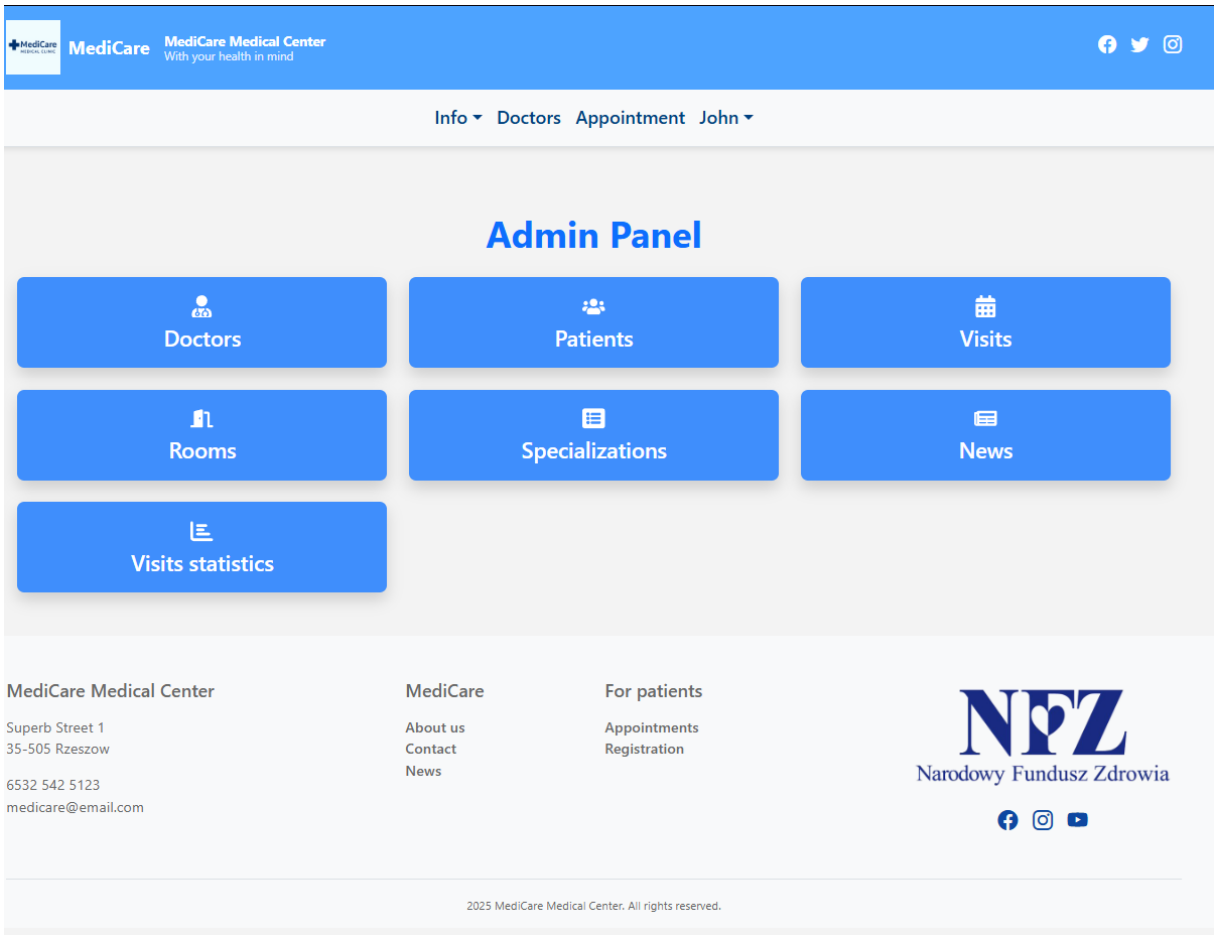
3. Edycja danych osobowych doktora. Rys. 5.10



Rys. 5.10: Edycja profilu doktora.

5.4 Administrator

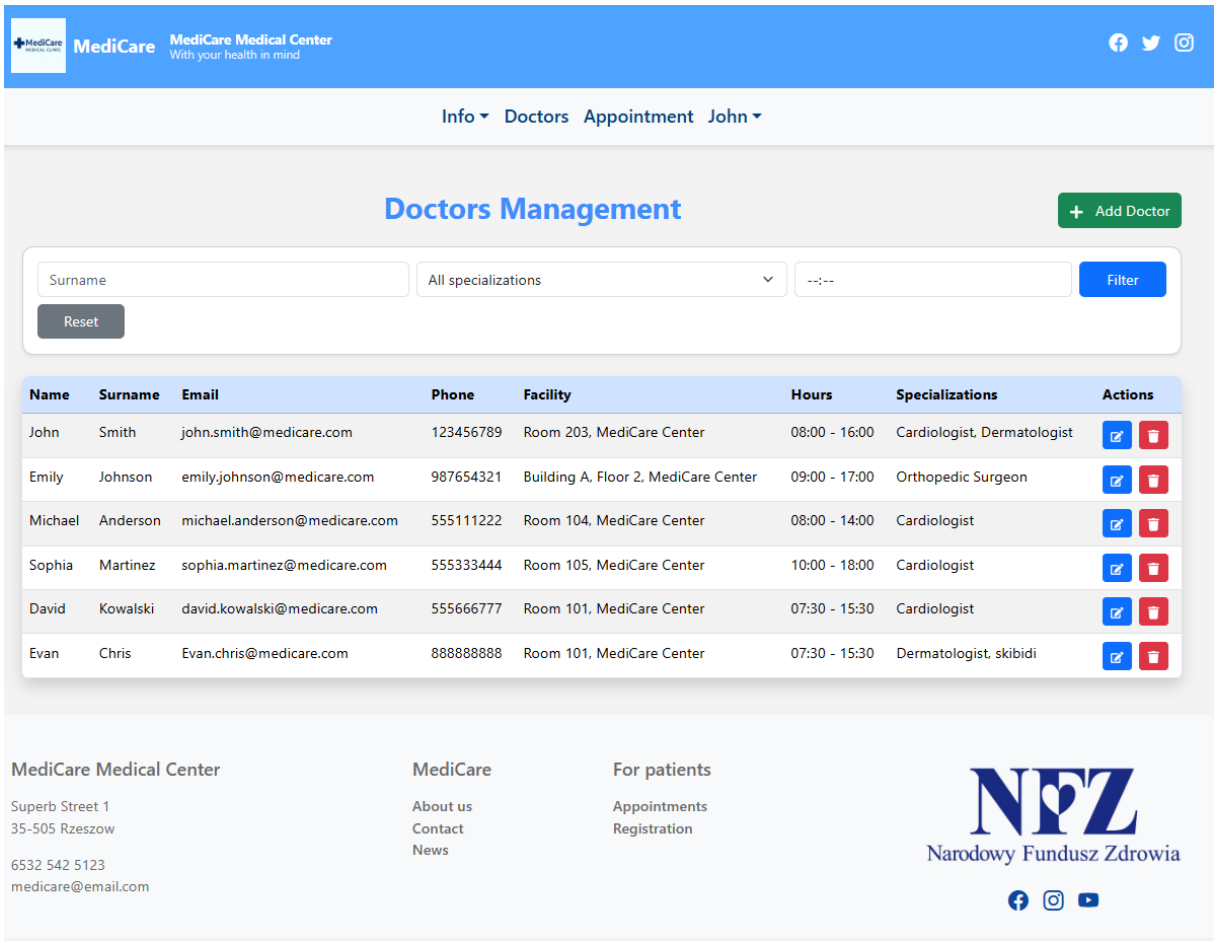
1. Panel administratora – siatka przycisków funkcyjnych (doktorzy, pacjenci, wizyty, pokoje, specjalizacje, newsy, statystyki). Rys. 5.11



Rys. 5.11: Panel administratora.

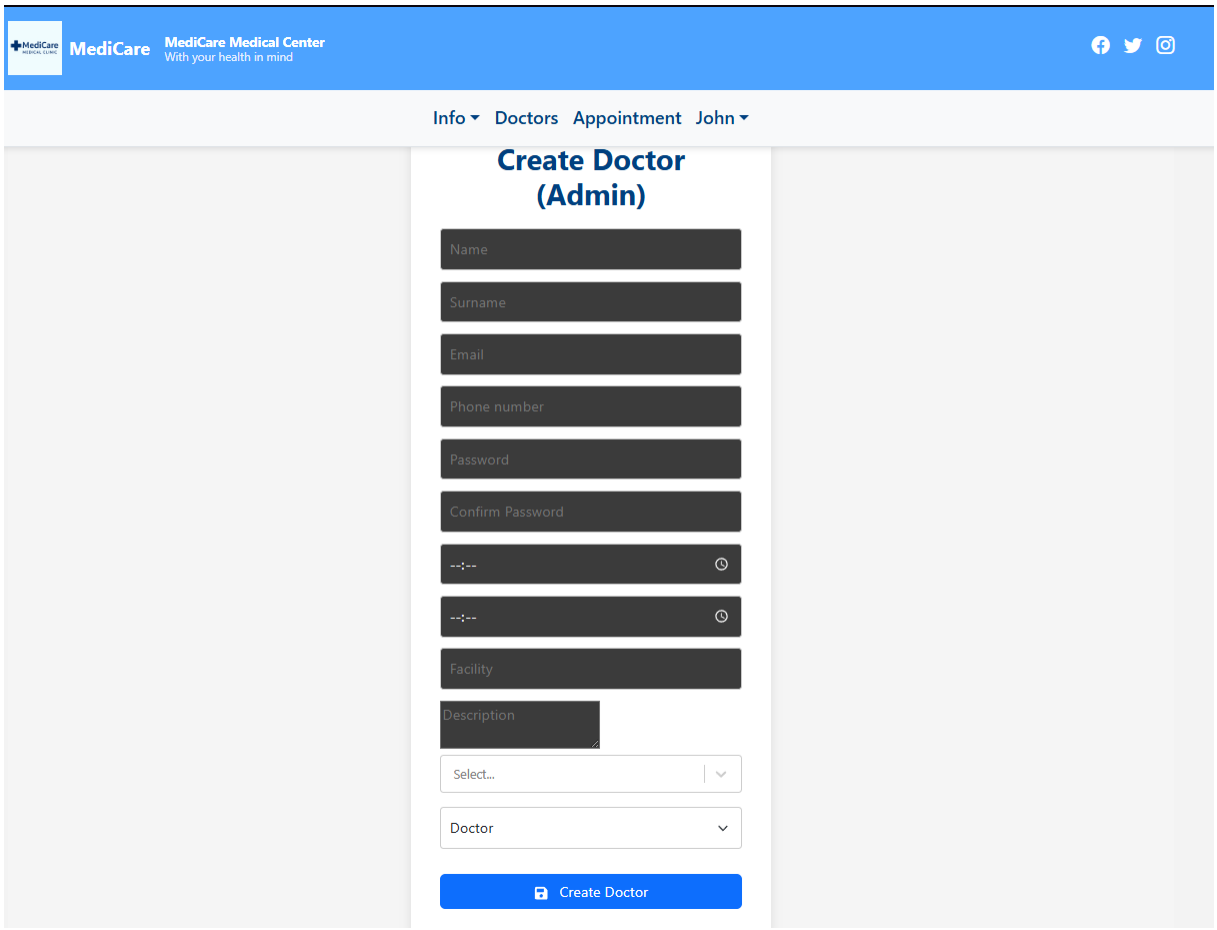
2. Zarządzanie doktorami:

   (a) Lista doktorów – wyszukiwanie po nazwisku, specjalizacji, godzinach pracy. Rys. 5.12



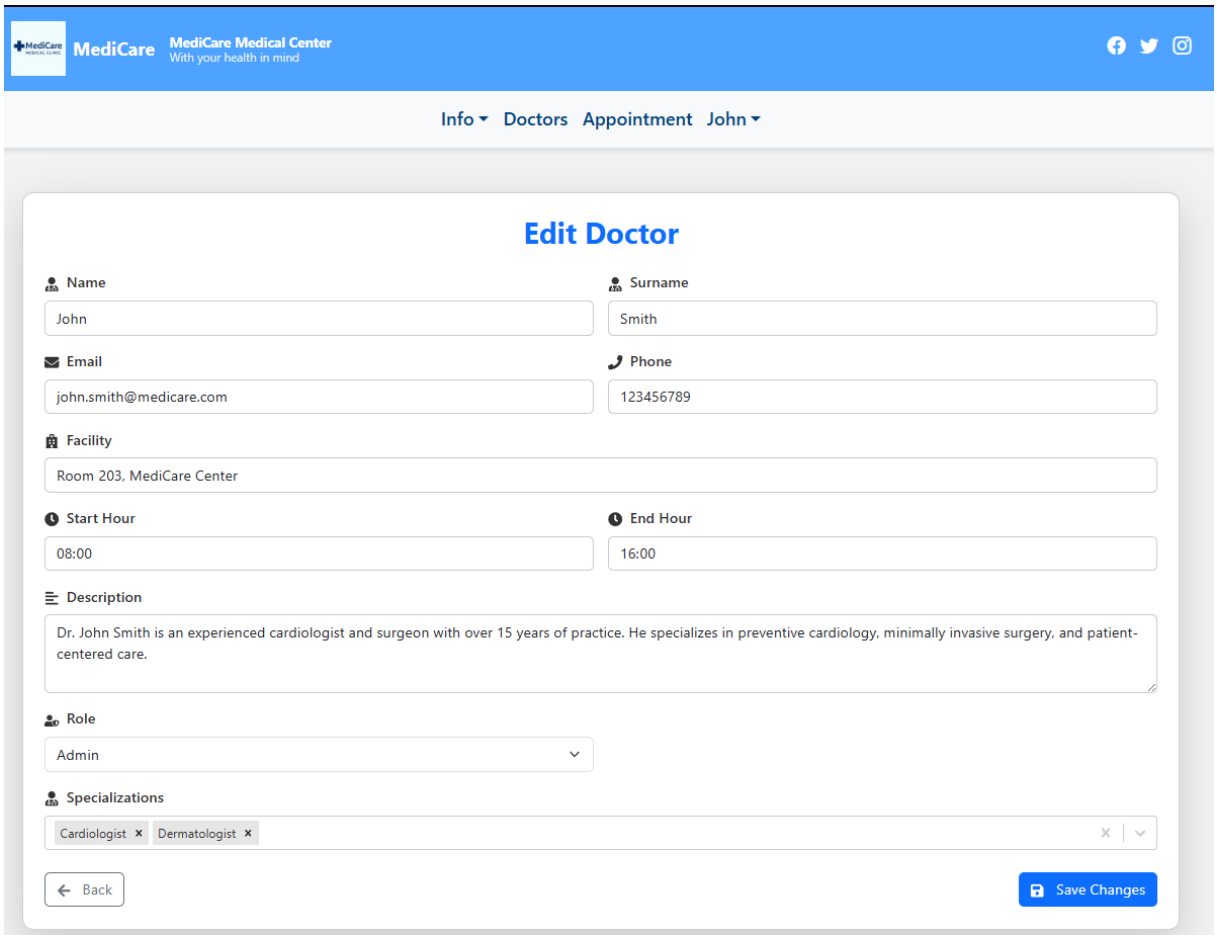
Rys. 5.12: Zarządzanie doktorami.

(b) Dodawanie doktora – formularz „Add Doctor”. Rys. 5.13



Rys. 5.13: Tworzenie nowego doktora.

(c) Edycja doktora – przycisk „Edit”. Rys. 5.14

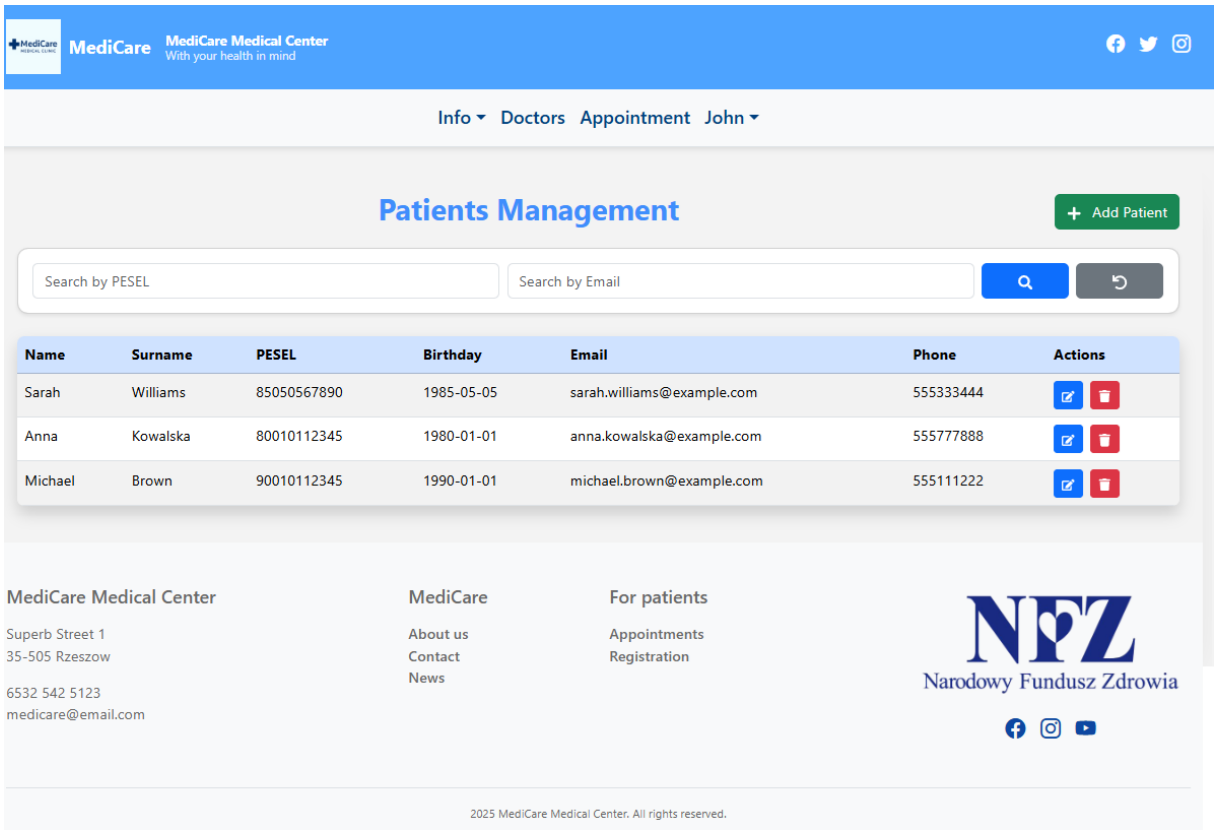


Rys. 5.14: Edycja doktora.

(d) Usuwanie doktora – przycisk „Delete”.

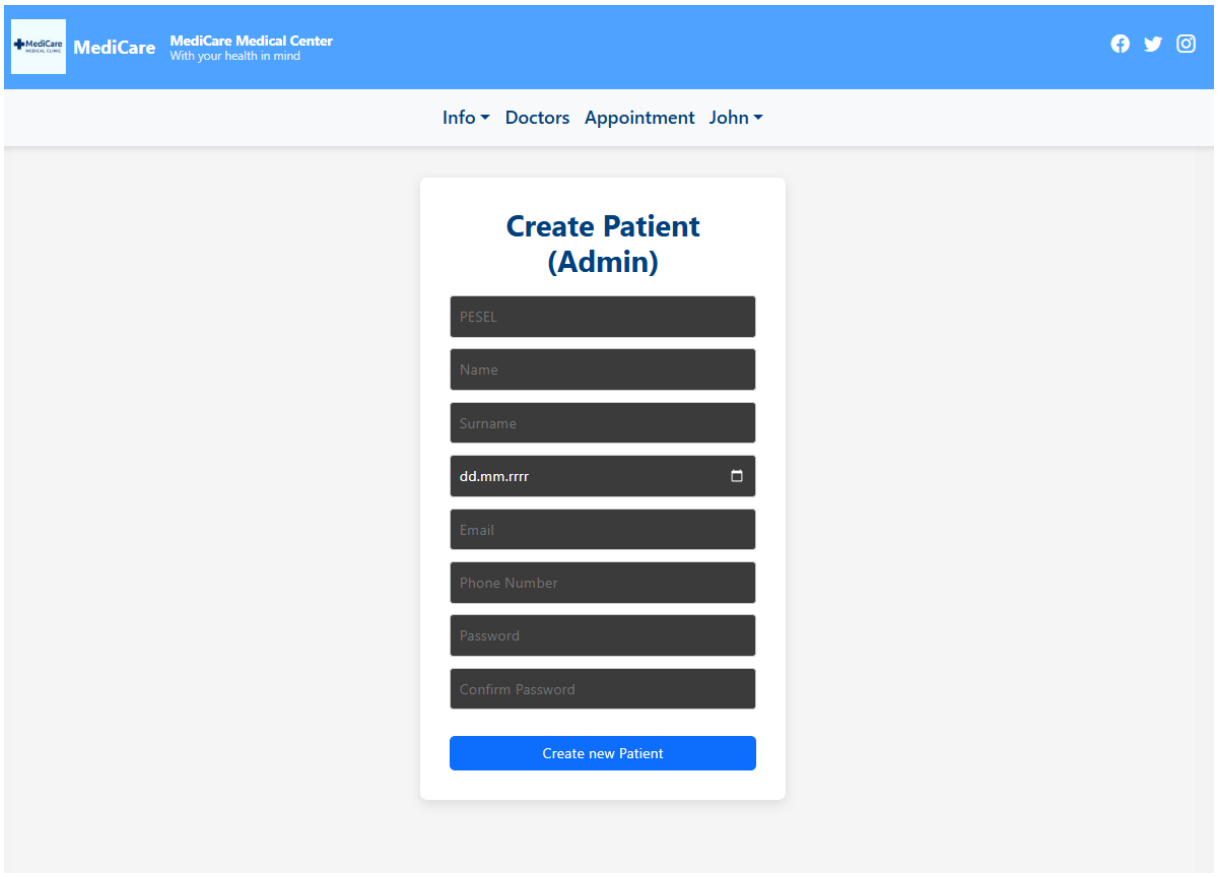
3. Zarządzanie pacjentami:

(a) Lista pacjentów – wyszukiwanie po PESELu lub e-mailu. Rys. 5.15



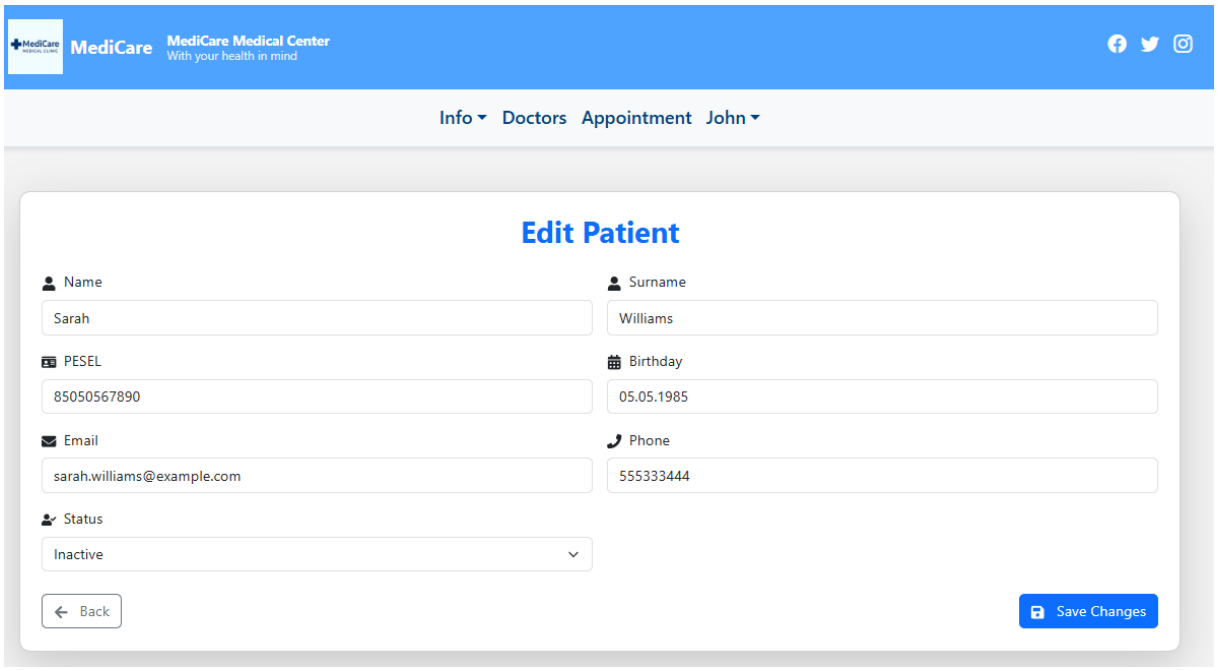
Rys. 5.15: Zarządzanie pacjentami.

(b) Dodawanie pacjenta – formularz „Add Patient”. Rys. 5.16



Rys. 5.16: Tworzenie nowego pacjenta.

(c) Edycja pacjenta – przycisk „Edit”. Rys. 5.17

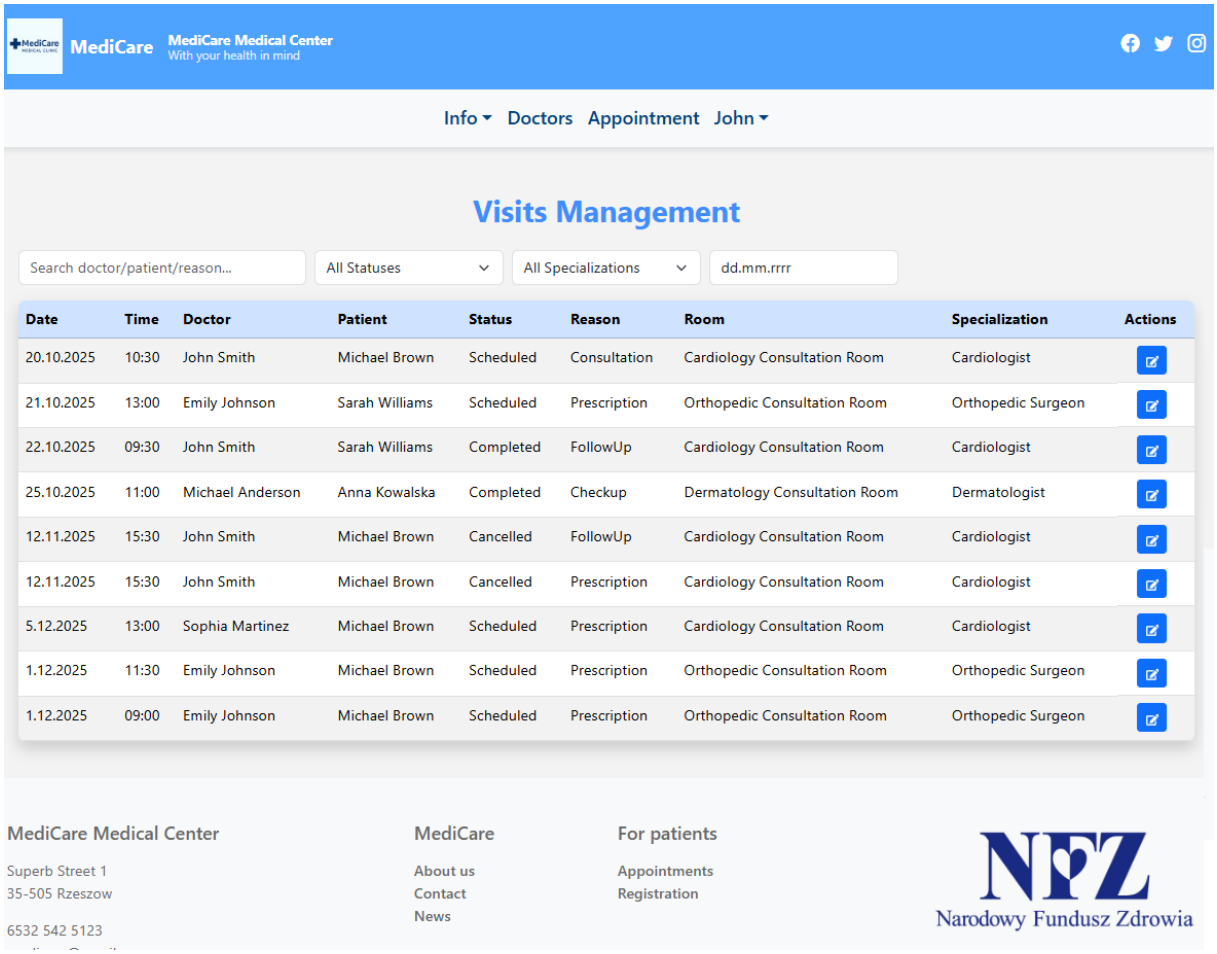


Rys. 5.17: Edycja pacjenta.

(d) Usuwanie pacjenta – przycisk „Delete”.

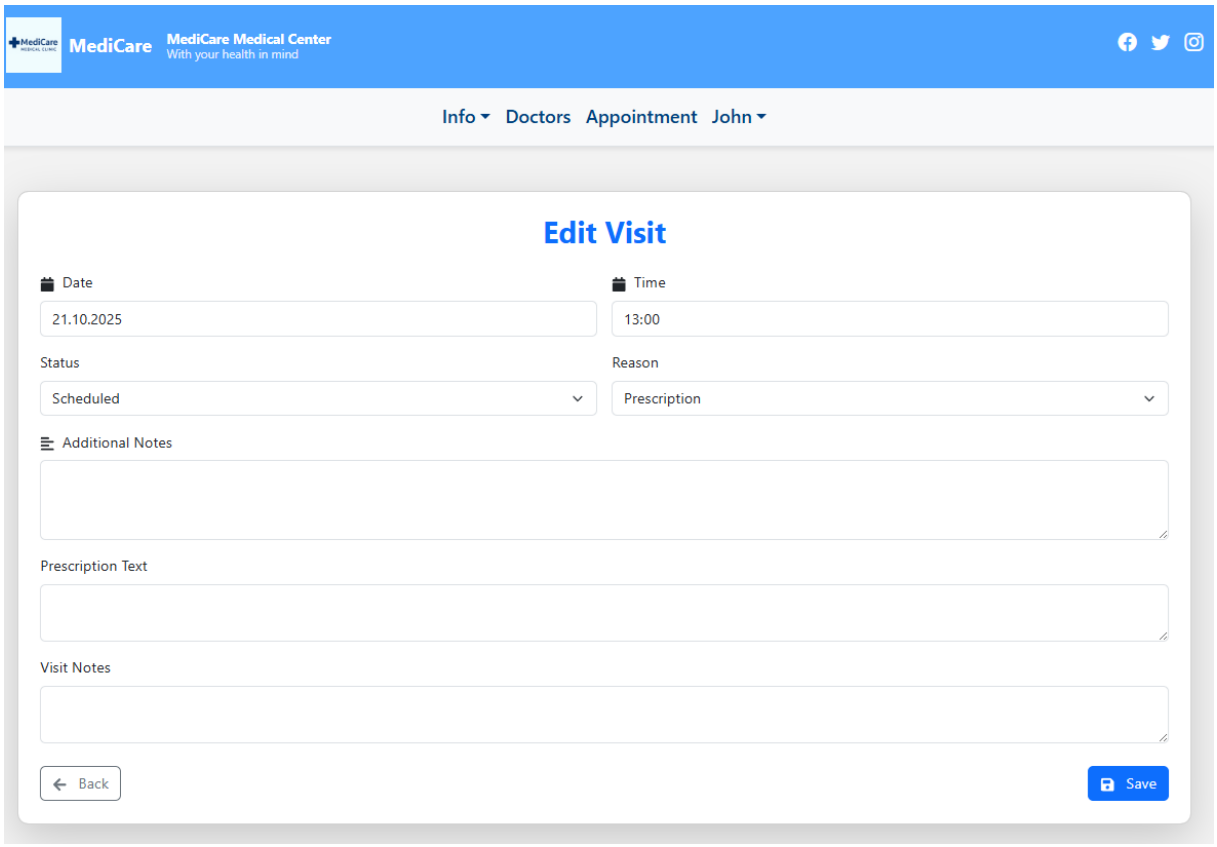
4. Zarządzanie wizytami:

   (a) Lista wizyt - filtrowanie wizyt po danych doktora, pacjenta, powodziem statusie wizyty, dla wybranej specjalizacji lub konkretnej daty. Rys. 5.18



Rys. 5.18: Lista wizyt.

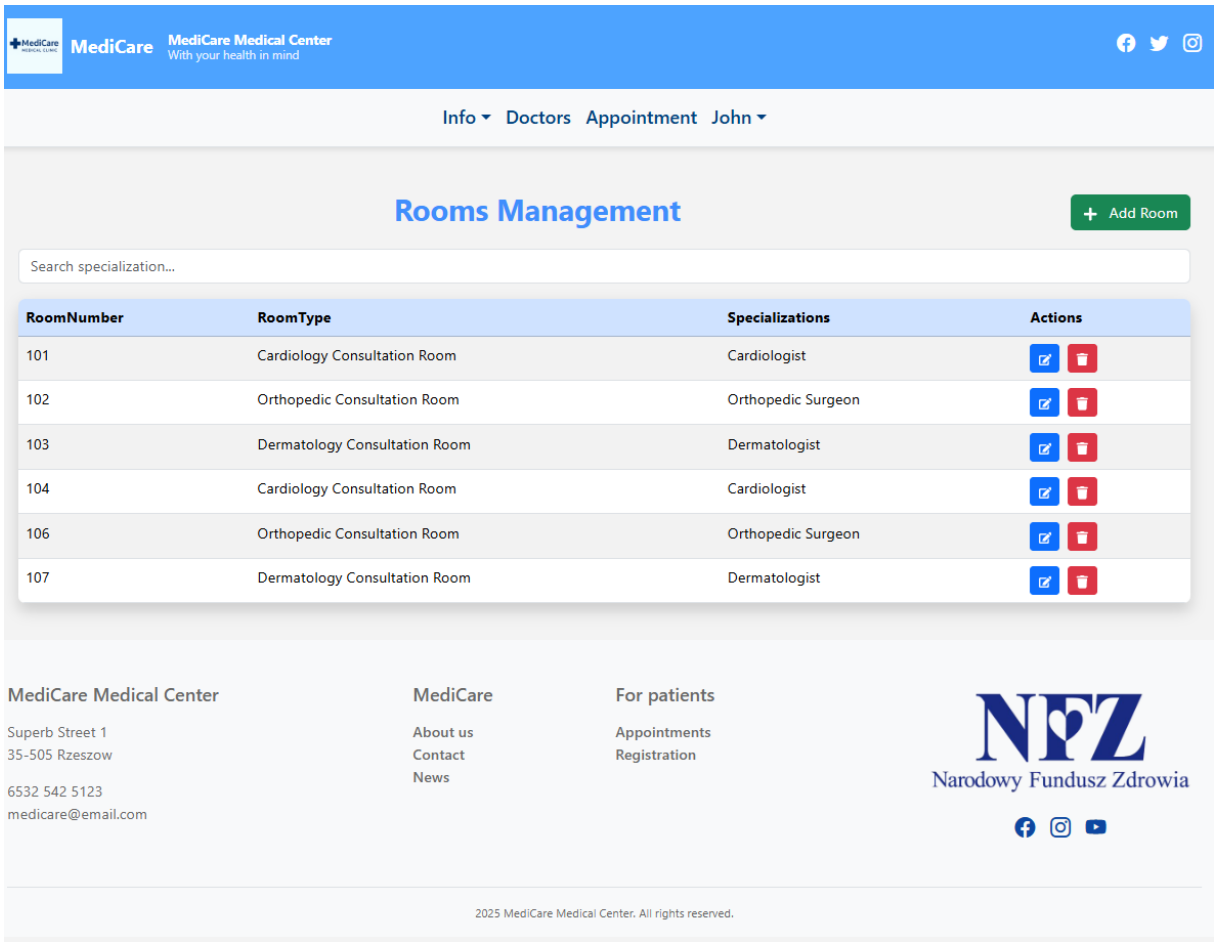
   (b) Edycja wizyty - przycisk "Edit"z kolumnt "Actions"Rys. 5.19



Rys. 5.19: Edycja wizyty.

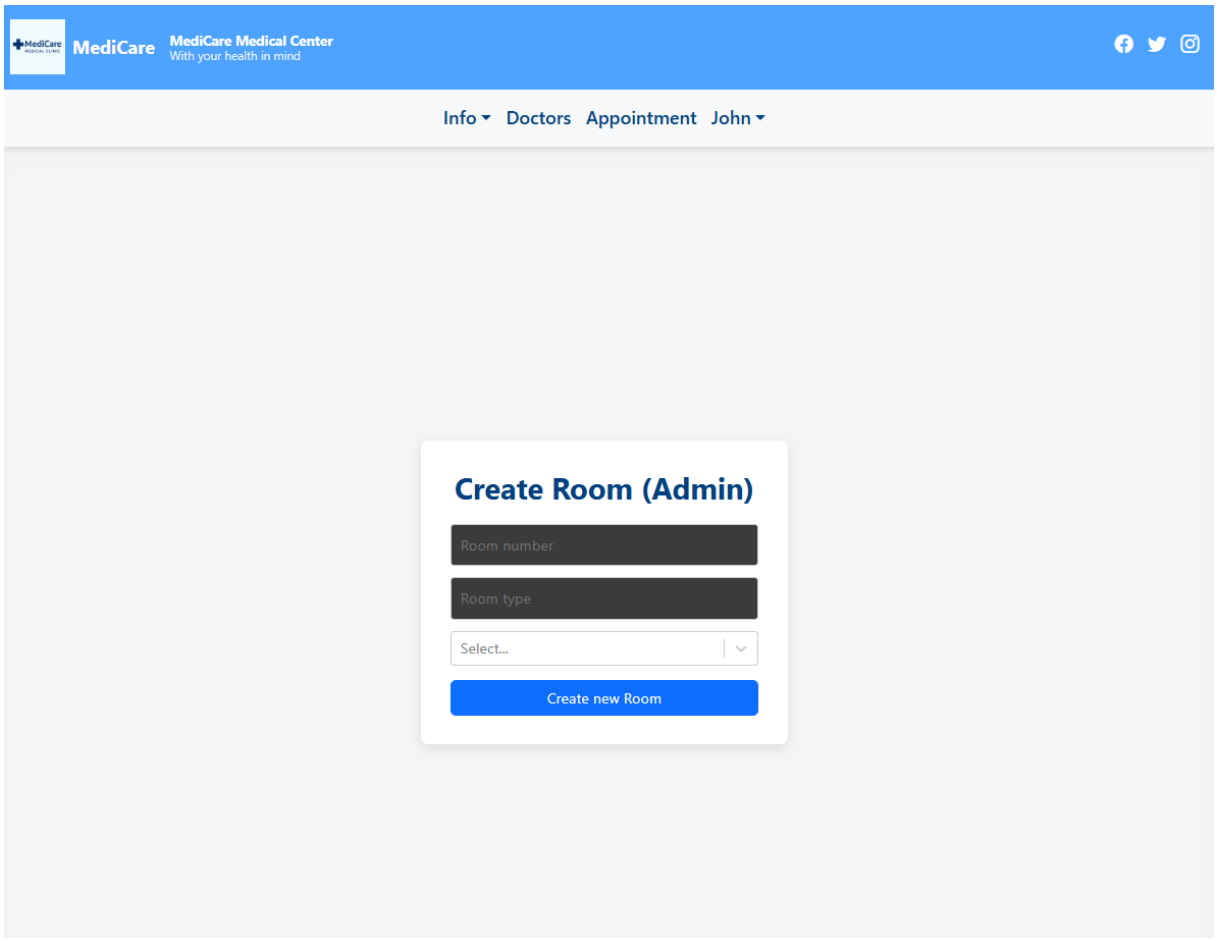
5. Zarządzanie pokojami:

   (a) Lista pokoi - filtrowanie pokoi po specjalizacji przypisanej do danego pokoju Rys. 5.20



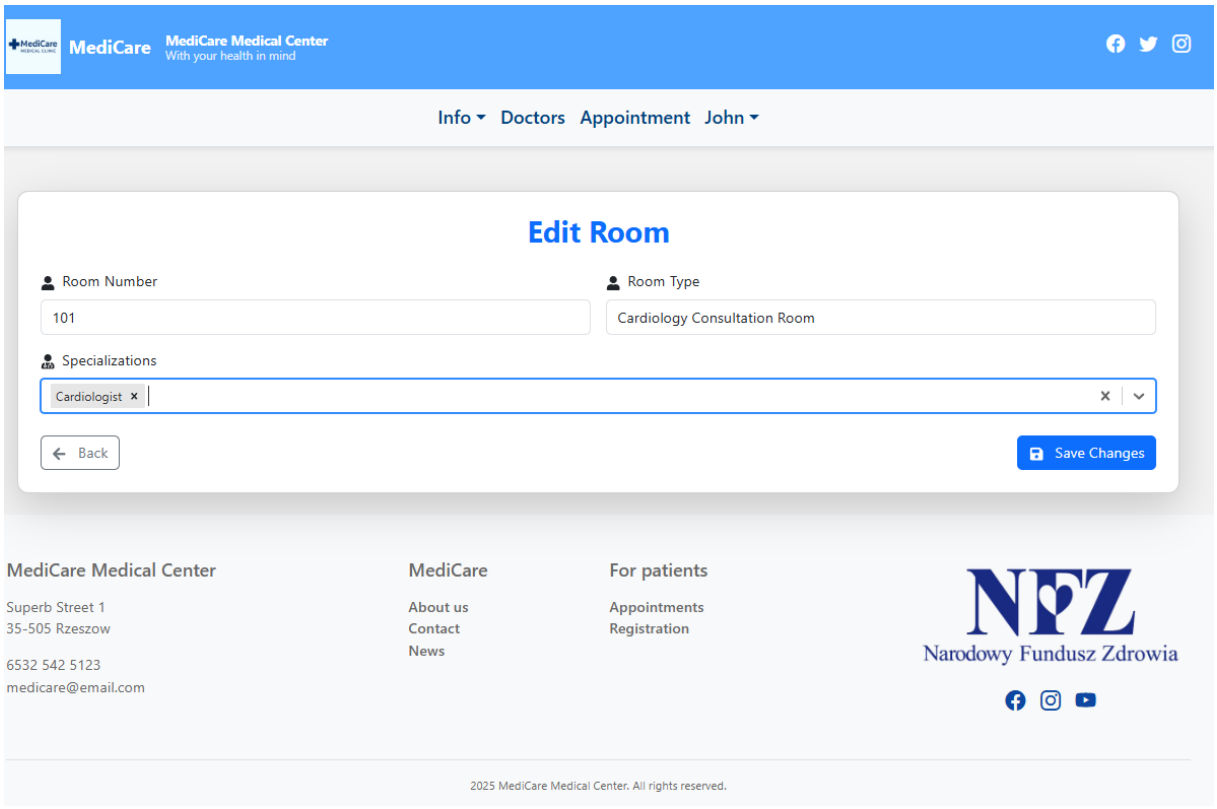
Rys. 5.20: Lista pokoi.

(b) Dodawanie pokoju - formularz "Add Room"Rys. 5.21



Rys. 5.21: Dodawanie pokoju.

(c) Edytowanie pokoju - formularz przycisku "Edit"z kolumny "Actions"Rys. 5.22

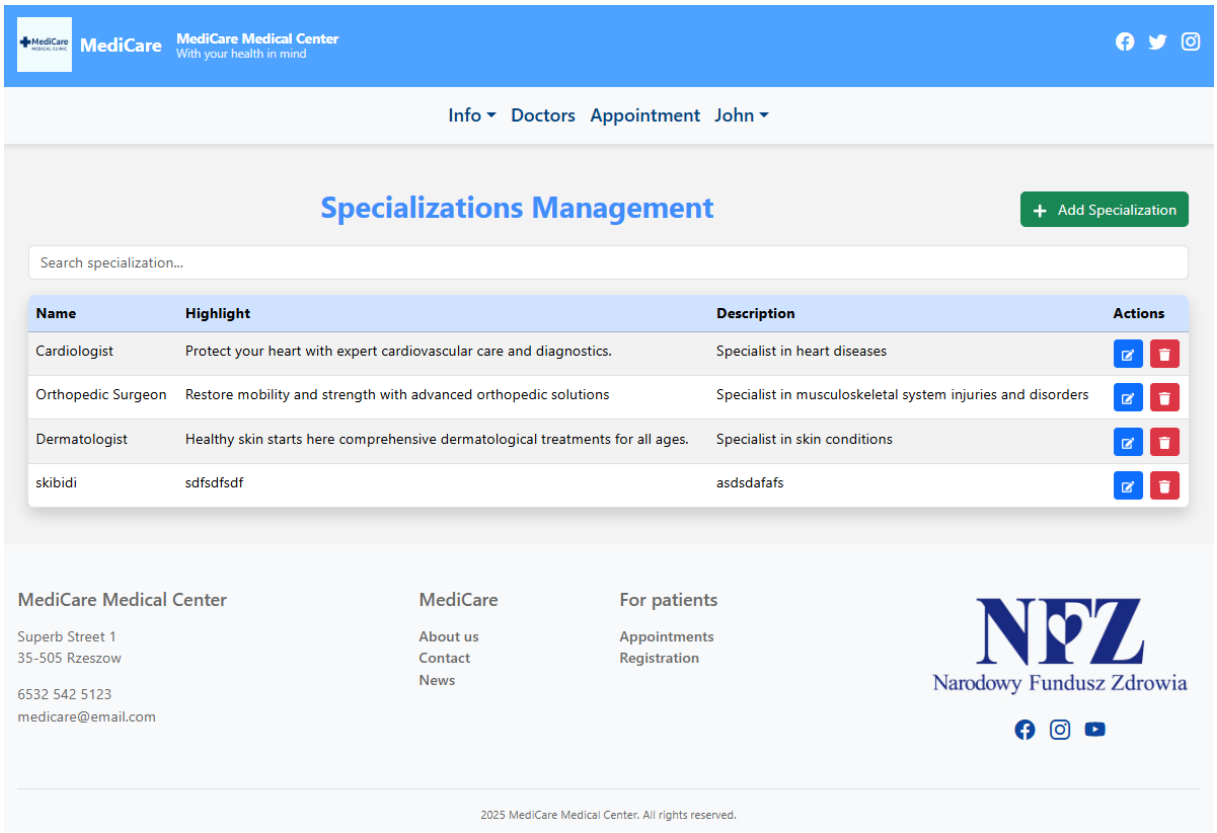


Rys. 5.22: Edycja pokoju.

(d) Usuwanie pokoju - czerwony przycisk "Delete"z kolumny "Actions"

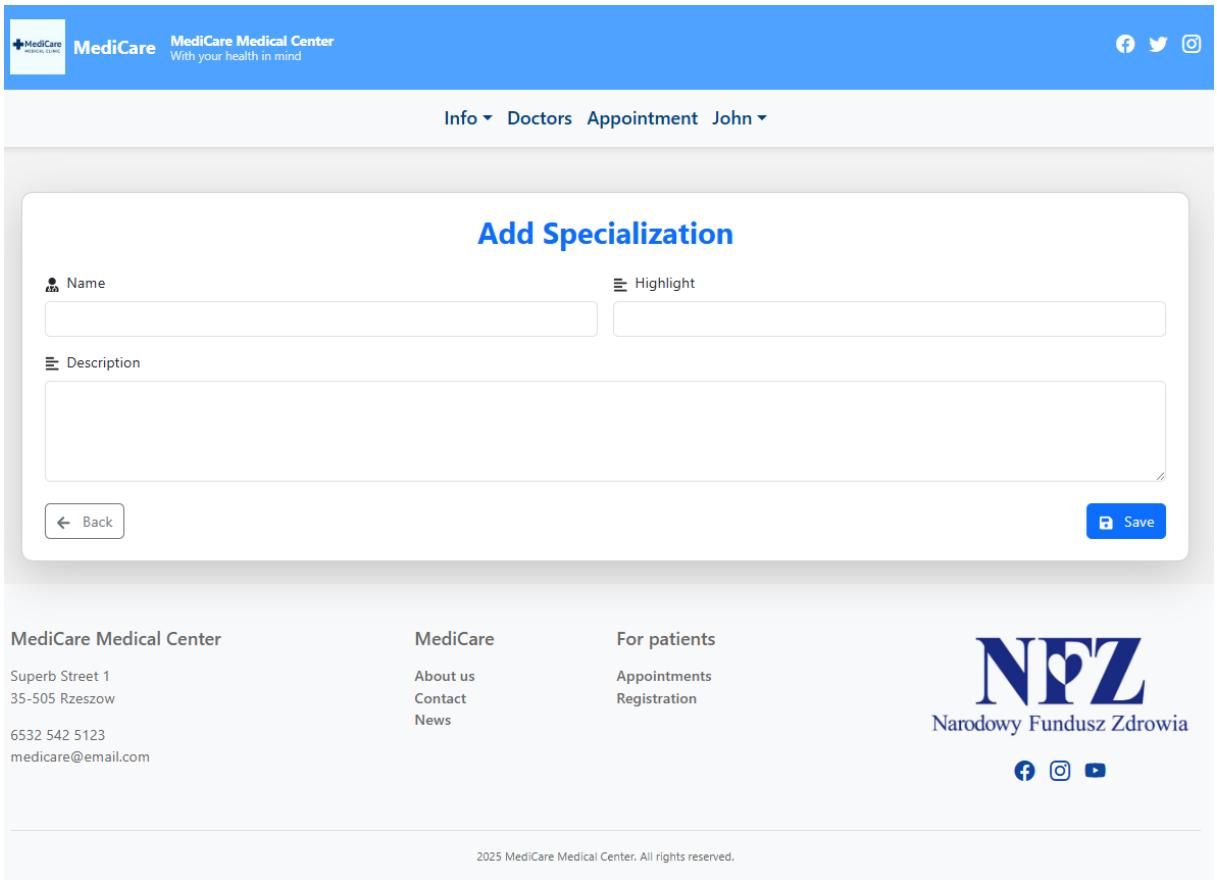
6. Zarządzanie specjalizacjami:

(a) Lista specjalizacji - wyszukiwanie specjalizacji po jej nazwie Rys. 5.23



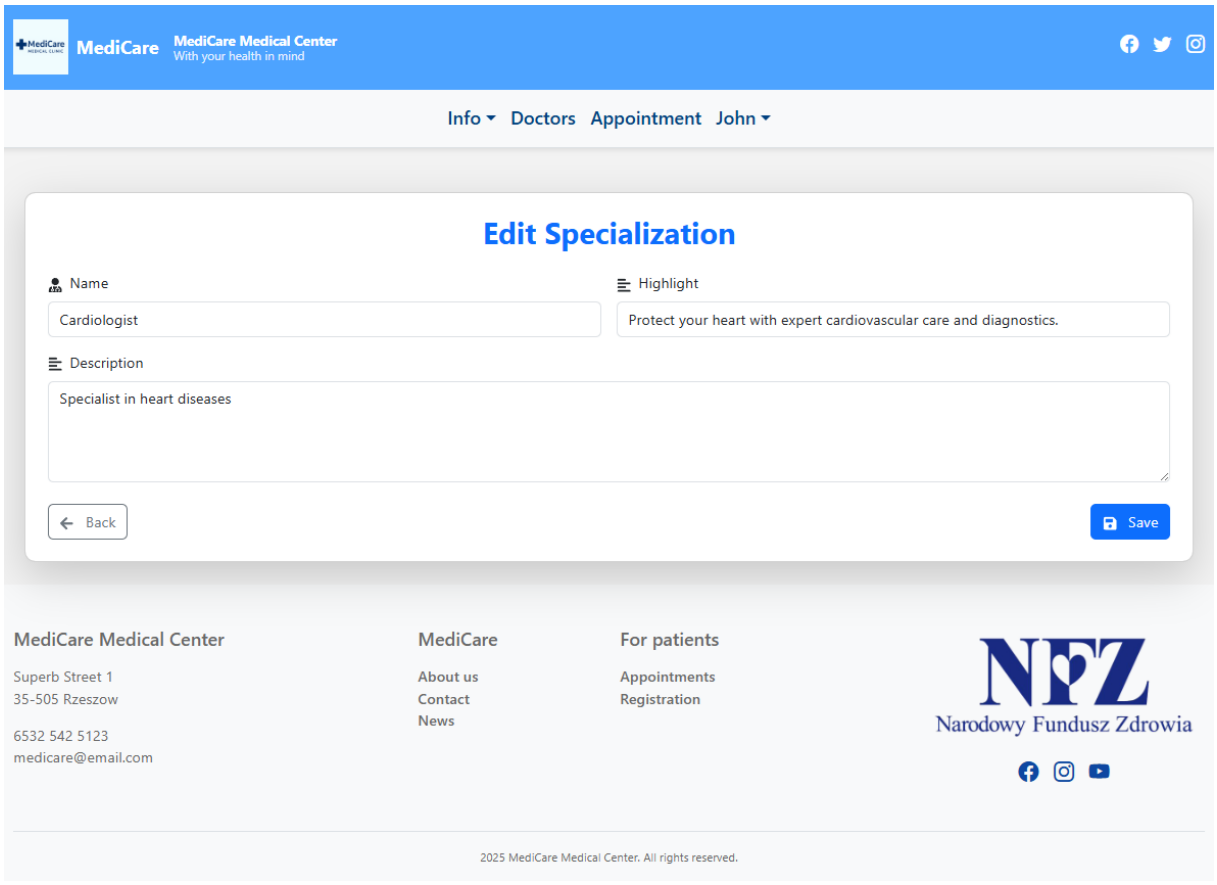
Rys. 5.23: Lista specjalizacji.

(b) Dodawanie specjalizacji - formularz "Add Specialization"Rys. 5.24



Rys. 5.24: Dodawanie specjalizacji.

(c) Edycja specjalizacji - niebieski przycisk "Edit"z kolumny "Actions"Rys. 5.25

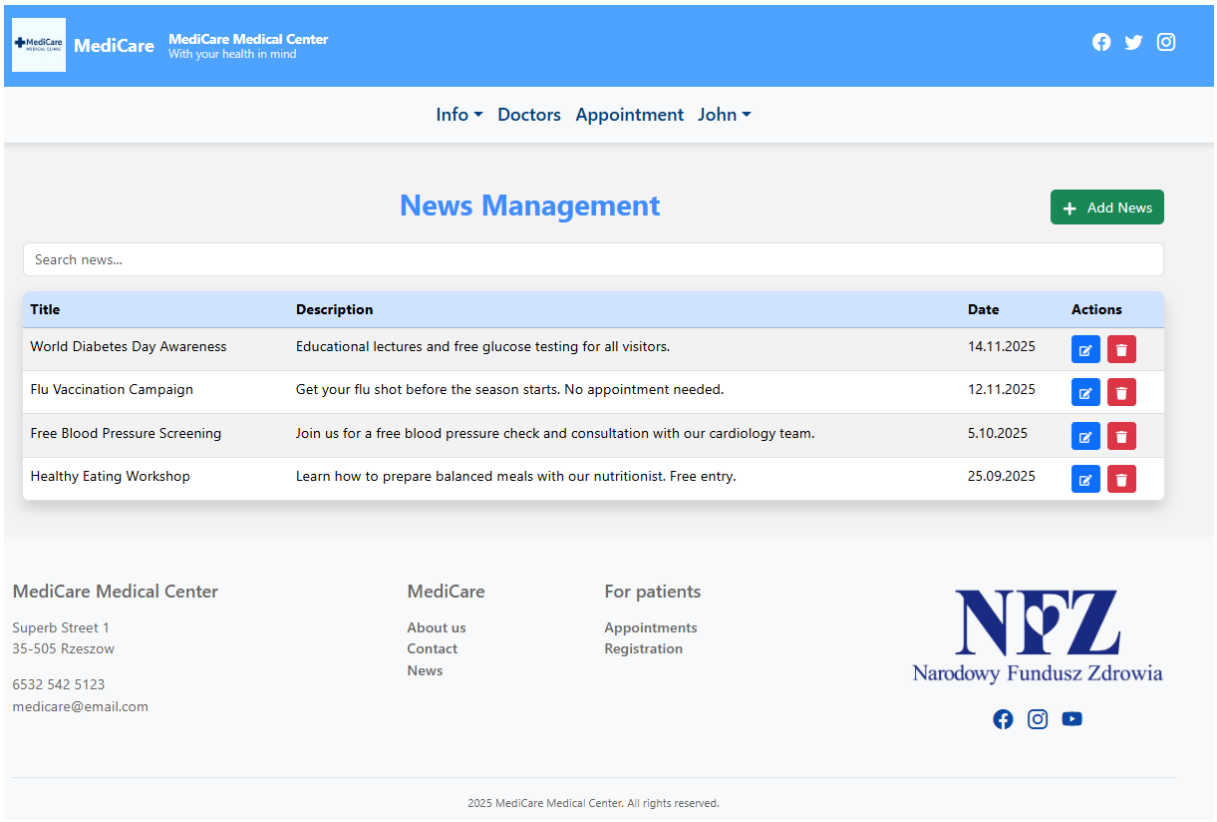


Rys. 5.25: Edycja specjalizacji.

(d) Usuwanie specjalizacji - czerwony przycisk "Delete"z kolumny "Actions"

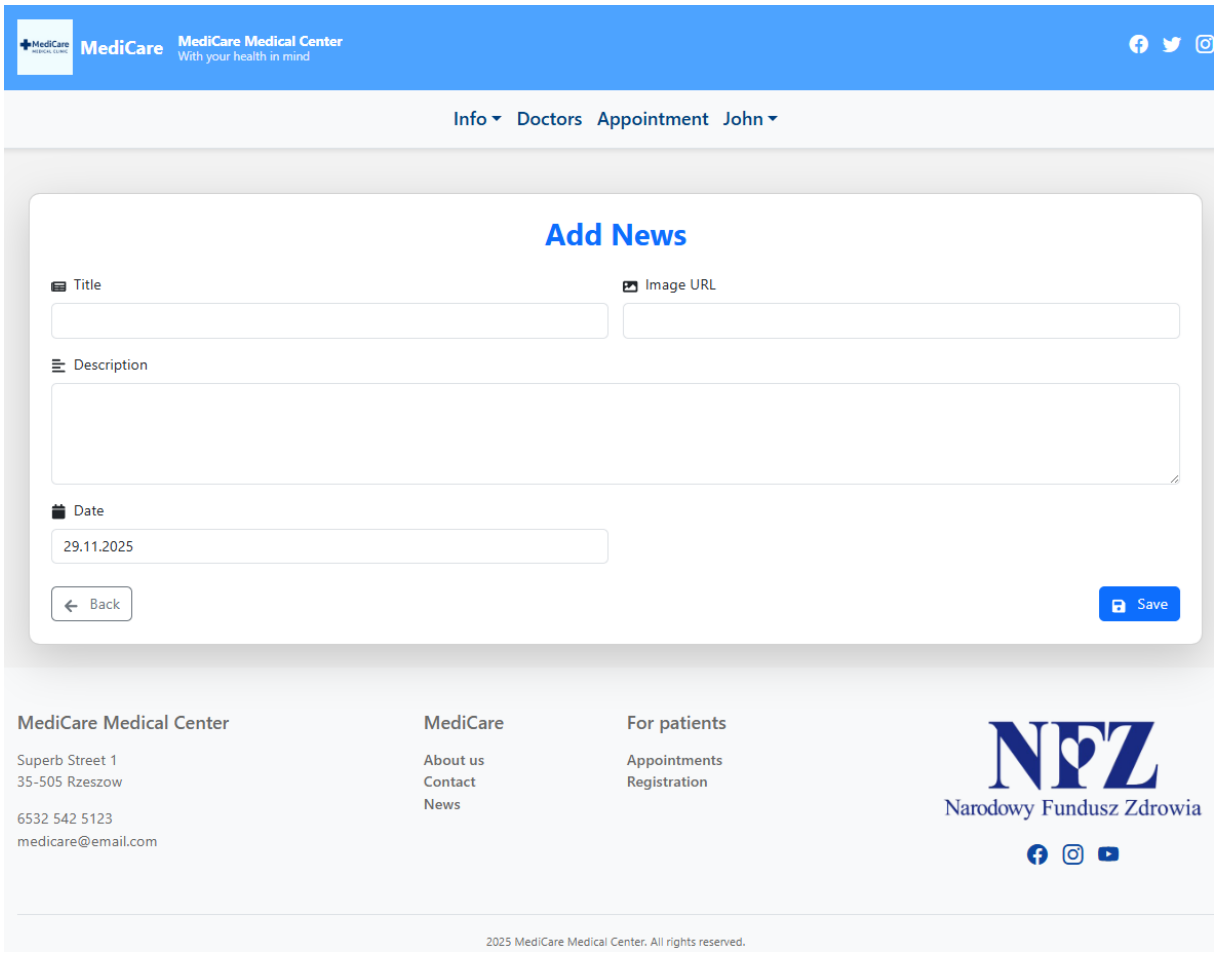
7. Zarządzanie artykułami:

(a) Lista artykułów - wyszukiwanie artykułów po słowach znajdujących sie w artykule Rys. 5.26



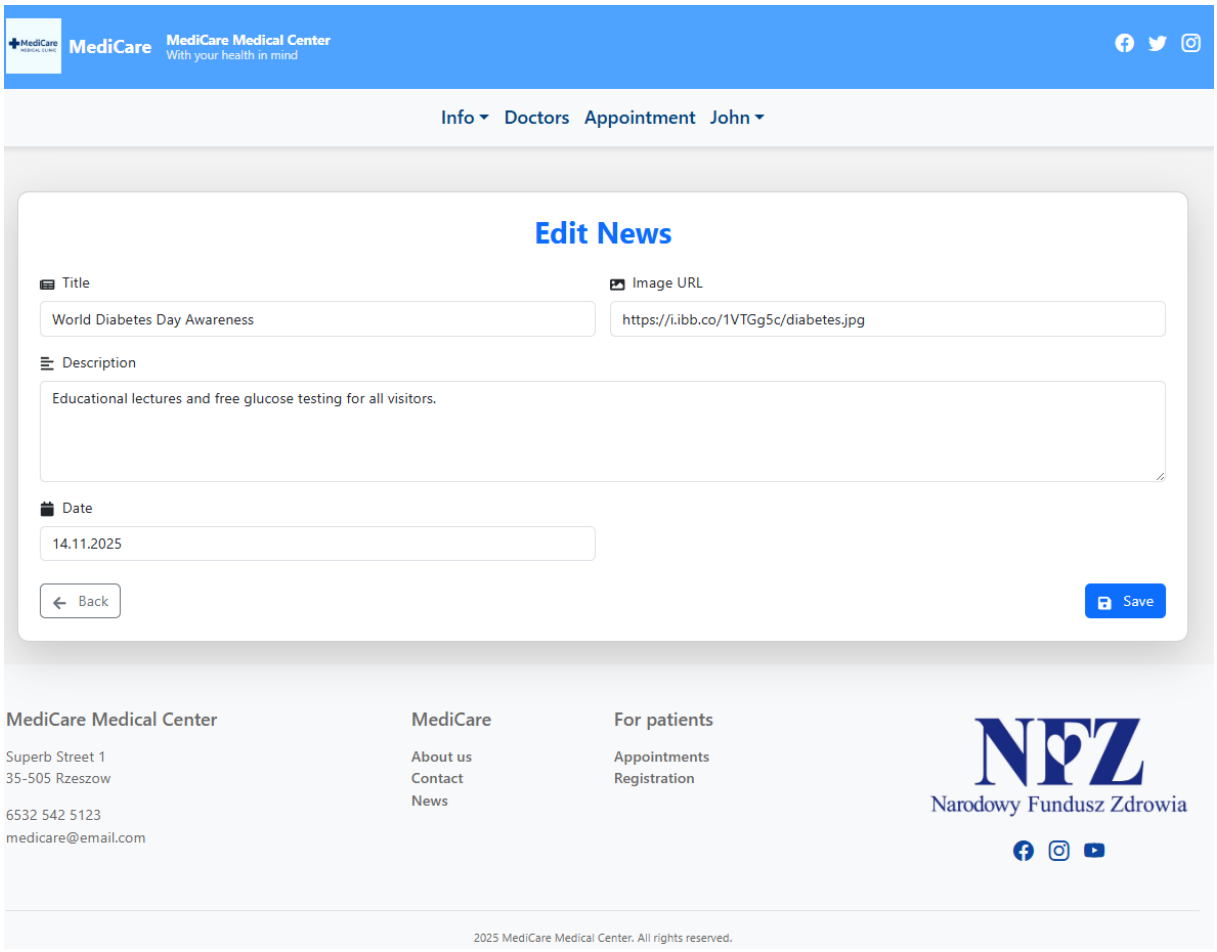
Rys. 5.26: Lista artykułów.

(b) Dodawanie artykułu - formularz "Add News"Rys. 5.27



Rys. 5.27: Formularz tworzenia artykułu.

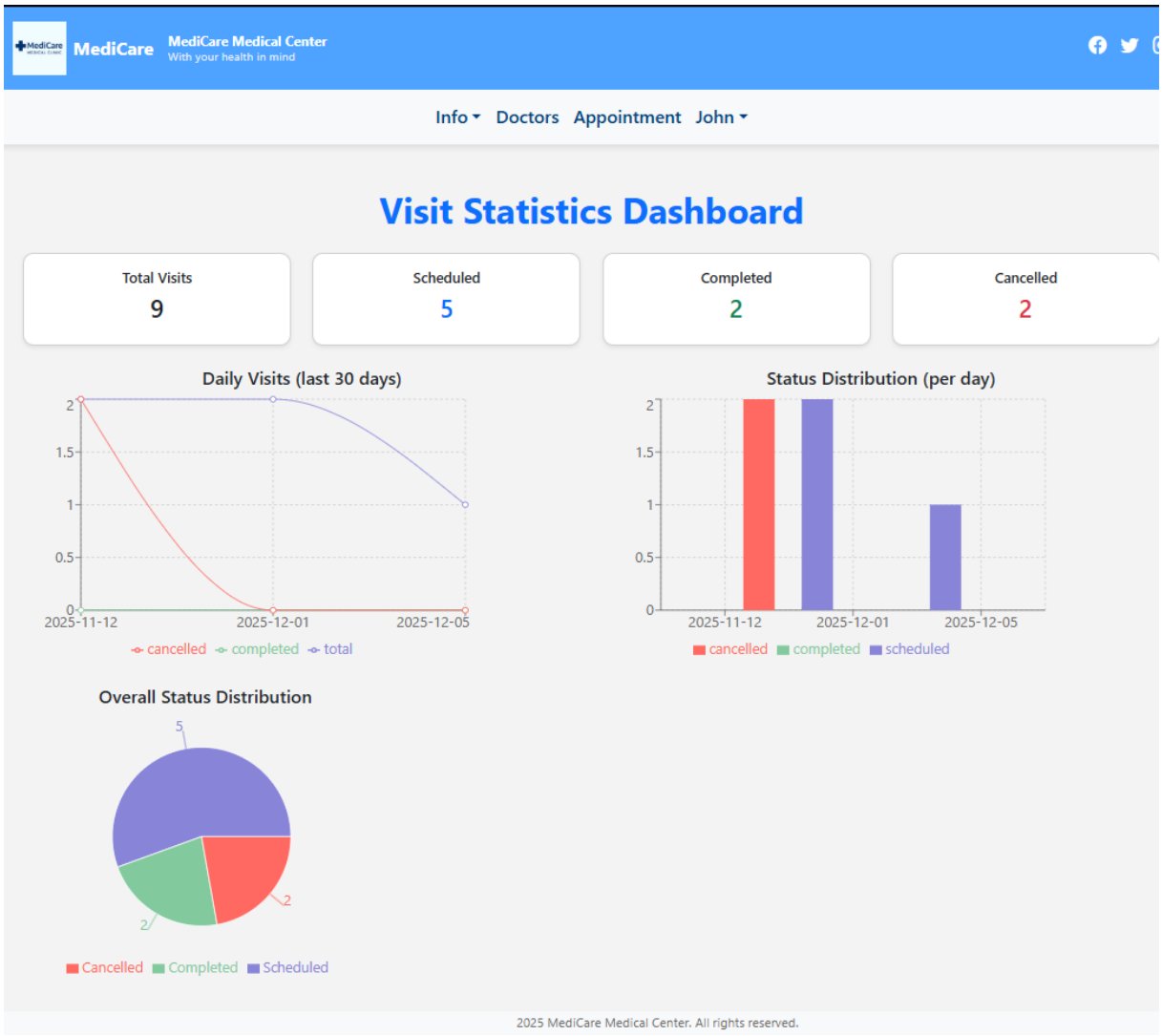
(c) Edycja artykułu - niebieski przycisk "Edit"z kolumny "Actions"Rys. 5.28



Rys. 5.28: Edycja artykułu.

(d) Usuwanie artykułu - czerwony przycisk "Delete"z kolumny "Actions"

8. Statystyki wizyt - spis statystyk dotyczące wizyt, zaplanowane, ukonczone lub anulowane Rys. 5.29



Rys. 5.29: Statystyki wizyt.

Zabezpieczenia

[4] Rozdział ten przedstawia mechanizmy bezpieczeństwa zastosowane w systemie **MediCare**, obejmujące zarówno warstwę backendową, jak i frontendową. Celem jest zapewnienie bezpiecznego uwierzytelniania użytkowników, kontrola dostępu do zasobów API oraz ochrona sesji przed nieautoryzowanym użyciem.

6.1 Mechanizmy backendowe

System **MediCare** wykorzystuje mechanizmy oparte na tokenach JWT (JSON Web Token) do uwierzytelniania i autoryzacji użytkowników. Dzięki temu możliwe jest bezpieczne zarządzanie sesją oraz kontrola dostępu do zasobów API.

6.1.1 Generowanie tokenów JWT

Za generowanie tokenów odpowiada klasa `JwtTokenHelper`. Tworzy ona podpisany token JWT zawierający podstawowe informacje o użytkowniku (ID, adres e-mail, imię, rola). Token jest podpisywany algorytmem `HMAC-SHA256` przy użyciu klucza symetrycznego przechowywanego w konfiguracji systemu. Czas życia tokenu został ograniczony do jednej godziny, co zwiększa bezpieczeństwo i ogranicza ryzyko przejęcia sesji.

6.1.2 Tokeny odświeżające

Oprócz tokenów dostępowych system generuje również *refresh tokeny*, które są przechowywane w bazie danych i powiązane z konkretnym użytkownikiem (pacjentem lub lekarzem). Kontroler `AuthController` udostępnia endpoint `POST /api/auth/refresh`, który umożliwia uzyskanie nowej pary tokenów (access i refresh) na podstawie ważnego, nieodwołanego refresh tokenu. W przypadku niepoprawnego lub wygasłego tokenu system zwraca odpowiedź `401 Unauthorized`.

6.2 Mechanizmy frontendowe

Po stronie klienta system **MediCare** wykorzystuje bibliotekę `axios` do komunikacji z serwerem API. Dzięki temu wszystkie wywołania są wykonywane w sposób spójny i centralnie zarządzany.

6.2.1 Interceptor żądań

Przed wysłaniem każdego żądania sprawdzane jest, czy w pamięci przeglądarki (`localStorage`) znajduje się aktualny token JWT. Jeśli tak, zostaje on automatycznie dodany do nagłówka

`Authorization`. Zapewnia to, że wszystkie wywołania API wymagające autoryzacji są wykonywane z odpowiednimi uprawnieniami.

6.2.2 Interceptor odpowiedzi

W przypadku odpowiedzi serwera z kodem `401 Unauthorized` lub `403 Forbidden`, mechanizm automatycznie próbuje odświeżyć token dostępowy przy użyciu refresh tokenu. Dzięki temu użytkownik nie musi ponownie się logować, a sesja pozostaje aktywna. Jeśli odświeżenie się nie powiedzie, użytkownik zostaje wylogowany i przekierowany na stronę główną. Dodatkowo obsługiwane są błędy serwera (`500`) oraz brak zasobu (`404`), które skutkują przekierowaniem na odpowiednie strony błędów.

6.2.3 Ochrona tras (ProtectedRoute)

Dostęp do komponentów frontendu został zabezpieczony przy użyciu komponentu `ProtectedRoute`. Sprawdza on, czy użytkownik jest zalogowany oraz czy posiada odpowiednią rolę (np. `Patient`, `Doctor`, `Admin`). W przypadku braku autoryzacji użytkownik zostaje przekierowany na stronę logowania lub stronę `/unauthorized`. Rozwiązanie to zapewnia kontrolę dostępu do widoków aplikacji zgodnie z uprawnieniami użytkownika.

6.3 Podsumowanie

Zastosowane mechanizmy bezpieczeństwa zapewniają:

- bezpieczne uwierzytelnianie i autoryzację użytkowników,
- kontrolę dostępu do zasobów API w zależności od roli (pacjent, lekarz, administrator),
- odporność na przejęcie sesji dzięki krótkiej ważności tokenów dostępowych,
- automatyczne odświeżanie sesji i wygodę użytkownika po stronie klienta,
- centralne zarządzanie błędami i przekierowaniami w aplikacji frontendowej.

Testowanie systemu

[2], [4] Rozdział ten przedstawia proces testowania systemu **MediCare**, którego celem było zapewnienie poprawności działania kluczowych komponentów oraz weryfikacja zgodności z wymaganiami funkcjonalnymi. Testy zostały podzielone na dwie główne kategorie: testy jednostkowe, które sprawdzają poprawność działania pojedynczych metod i funkcji w izolacji, oraz testy integracyjne, które weryfikują współdziałanie wielu elementów systemu (np. kontrolerów API, warstwy bazy danych i mechanizmów bezpieczeństwa).

7.1 Infrastruktura testowa

7.1.1 Testowa infrastruktura – EmptyDbFactory

W celu uruchamiania testów integracyjnych w systemie **MediCare** przygotowano specjalną klasę `EmptyDbFactory`, która rozszerza `WebApplicationFactory<Program>`. Jej zadaniem jest skonfigurowanie środowiska testowego z pustą bazą danych działającą w trybie `InMemory`. Dzięki temu każdy test uruchamiany jest na świeżej instancji bazy, co zapewnia pełną izolację danych i eliminuje ryzyko wpływu wyników jednego testu na kolejny.

Konfiguracja W metodzie `ConfigureWebHost` usuwana jest domyślna konfiguracja kontekstu `MediCareDbContext`, a następnie dodawany jest nowy kontekst oparty o bazę `InMemoryDatabase`. Każde uruchomienie testu korzysta z unikalnej nazwy bazy (`Guid.NewGuid().ToStr`
co gwarantuje niezależność danych pomiędzy kolejnymi przypadkami testowymi.

Znaczenie `EmptyDbFactory` stanowi część infrastruktury testowej systemu. Umożliwia uruchamianie testów integracyjnych w kontrolerach bez konieczności korzystania z fizycznej bazy danych PostgreSQL. Rozwiązanie to przyspiesza proces testowania, upraszcza konfigurację oraz zapewnia powtarzalność wyników.

7.1.2 Testowa infrastruktura – SeededDbFactory

Klasa `SeededDbFactory` rozszerza `WebApplicationFactory<Program>` i służy do uruchamiania testów integracyjnych z wykorzystaniem bazy danych działającej w trybie `InMemory`. W odróżnieniu od `EmptyDbFactory`, baza danych jest tutaj wypełniana przykładowymi danymi (ang. *seed data*), co pozwala na testowanie bardziej złożonych scenariuszy biznesowych.

Konfiguracja W metodzie `ConfigureWebHost` usuwana jest domyślna konfiguracja kontekstu `MediCareDbContext`, a następnie dodawany jest nowy kontekst oparty o bazę

InMemoryDatabase. Przed dodaniem danych baza jest czyszczona, aby zapewnić spójność i powtarzalność wyników testów.

Seedowanie danych Do bazy dodawane są przykładowe rekordy dla kluczowych encji systemu:

- Specializations – przykładowe specjalizacje medyczne (kardiolog, ortopeda, dermatolog),
- Doctors – lekarze z przypisanymi godzinami pracy i opisami,
- Patients – pacjenci z podstawowymi danymi osobowymi,
- Rooms – pokoje konsultacyjne przypisane do specjalizacji,
- Visits – przykładowe wizyty powiązane z pacjentami i lekarzami,
- NewsItems – przykładowe ogłoszenia i wydarzenia.

Znaczenie SeededDbFactory umożliwia uruchamianie testów integracyjnych w realistycznym środowisku, gdzie dostępne są przykładowe dane odzwierciedlające rzeczywiste przypadki użycia systemu. Dzięki temu możliwe jest testowanie scenariuszy takich jak rejestracja wizyty, filtrowanie lekarzy czy pobieranie statystyk, bez konieczności ręcznego przygotowywania danych w każdym teście.

7.2 Testy integracyjne

7.2.1 Testy integracyjne pacjenta

W celu weryfikacji poprawności działania kontrolera PatientsController przygotowano zestaw testów integracyjnych. Testy te uruchamiane są na bazie danych w trybie InMemory, dzięki czemu możliwe jest sprawdzenie pełnej ścieżki od wywołania endpointu API do zapisu i odczytu danych.

Przykład 1: Pobieranie listy pacjentów Test sprawdza, czy wywołanie GET /api/patients zwraca poprawną listę pacjentów. Oczekiwany wynik to kod 200 OK oraz lista obiektów PatientDto z wypełnionymi polami.

```
[Fact]
public async Task GetPatients_ReturnOk()
{
    var client = new SeededDbFactory().CreateClient();
    var response = await client.GetAsync("api/patients");

    Assert.Equal(HttpStatusCode.OK, response.StatusCode);

    var patients = await response.Content.ReadFromJsonAsync<List<PatientDto>>();
    Assert.NotNull(patients);
    Assert.All(patients, p =>
```

```
    {
        Assert.False(string.IsNullOrEmpty(p.Name));
        Assert.False(string.IsNullOrEmpty(p.Surname));
    });
}
```

Przykład 2: Rejestracja pacjenta Test sprawdza, czy wywołanie `POST /api/patients/register` tworzy nowego pacjenta. Oczekiwany wynik to kod `201 Created` oraz zwrócenie obiektu `PatientDto` zgodnego z danymi wejściowymi.

```
[Fact]
public async Task CreatePatient_ReturnsCreated()
{
    var client = new EmptyDbFactory().CreateClient();

    var dto = new PatientRegisterDto
    {
        PESEL = "12345678900",
        Name = "John",
        Surname = "Smith",
        Birthday = DateTime.Now,
        Email = "john@email.com",
        PhoneNumber = "123457777",
        Password = "Test1234!"
    };

    var response = await client.PostAsJsonAsync("/api/patients/register", dto);

    Assert.Equal(HttpStatusCode.Created, response.StatusCode);

    var created = await response.Content.ReadFromJsonAsync<PatientDto>();
    Assert.NotNull(created);
    Assert.Equal("John", created!.Name);
}
```

Przykład 3: Usuwanie pacjenta Test sprawdza, czy wywołanie `DELETE /api/patients/{id}` usuwa pacjenta z systemu. Oczekiwany wynik to kod `204 No Content`, a kolejne wywołanie `GET` dla tego samego identyfikatora powinno zwrócić `404 Not Found`.

```
[Fact]
public async Task DeletePatient_ReturnsNoContent()
{
    var client = new SeededDbFactory().CreateClient();
    var list = await client.GetFromJsonAsync<List<PatientDto>>("/api/patients");
    int existingId = list.First().ID;

    var response = await client.DeleteAsync($"/api/patients/{existingId}");
    Assert.Equal(HttpStatusCode.NoContent, response.StatusCode);
```

```
    var getResponse = await client.GetAsync($"/api/patients/{existingId}");
    Assert.Equal(HttpStatusCode.NotFound, getResponse.StatusCode);
}
```

7.2.2 Testy integracyjne wizyt

W celu weryfikacji poprawności działania kontrolera `VisitsController` przygotowano zestaw testów integracyjnych. Testy te uruchamiane są na bazie danych w trybie `InMemory`, co pozwala sprawdzić pełną ścieżkę od wywołania endpointu API do zapisu i odczytu danych.

Przykład 1: Pobieranie wszystkich wizyt Test sprawdza, czy wywołanie `GET /api/visits` zwraca poprawną listę wizyt. Oczekiwany wynik to kod `200 OK` oraz lista obiektów `VisitResponseDto` z wypełnionymi polami.

```
[Fact]
public async Task GetVisitsReturn_Ok()
{
    var client = new SeededDbFactory().CreateClient();
    var response = await client.GetAsync("api/visits");

    response.EnsureSuccessStatusCode();
    var visits = await response.Content.ReadFromJsonAsync<List<VisitResponseDto>>();

    Assert.NotNull(visits);
    Assert.All(visits, v => Assert.True(v.ID > 0));
}
```

Przykład 2: Anulowanie wizyty Test sprawdza, czy wywołanie `POST /api/visits/canceledVisit` zmienia status wizyty na `Cancelled`. Oczekiwany wynik to kod `200 OK` oraz poprawnie zaktualizowany obiekt `VisitResponseDto`.

```
[Fact]
public async Task CancelVisitReturnOk()
{
    var client = new SeededDbFactory().CreateClient();
    var token = TestJwtTokenHelper.GenerateTestToken("1", "michael.brown@example.com"
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Beare

    var response = await client.PostAsync("api/visits/canceledVisit/1", null);
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);

    var visit = await client.GetFromJsonAsync<VisitResponseDto>("api/visits/1");
    Assert.Equal("Cancelled", visit!.Status);
}
```

Przykład 3: Tworzenie wizyty Test sprawdza, czy wywołanie `POST /api/visits` tworzy nową wizytę przy poprawnych danych. Oczekiwany wynik to kod `201 Created` oraz obiekt `VisitResponseDto` zgodny z danymi wejściowymi.

```
[Fact]
public async Task CreateVisit_ReturnsCreatedVisit_WhenValid()
{
    var client = new SeededDbFactory().CreateClient();
    var dto = new { VisitDate = DateOnly.FromDateTime(DateTime.Today.AddDays(1)),
                    VisitTime = new TimeOnly(10, 0),
                    DoctorID = 1, PatientID = 1,
                    SpecializationID = 1, RoomID = 1, Reason = 1 };

    var response = await client.PostAsJsonAsync("api/visits", dto);
    Assert.Equal(HttpStatusCode.Created, response.StatusCode);
}
```

7.2.3 Testy integracyjne lekarzy

W celu weryfikacji poprawności działania kontrolera `DoctorsController` przygotowano zestaw testów integracyjnych. Testy te obejmują scenariusze rejestracji, aktualizacji danych, resetu hasła oraz filtrowania lekarzy według kryteriów.

Przykład 1: Rejestracja lekarza Test sprawdza, czy wywołanie `POST /api/doctors/register` tworzy nowego lekarza. Oczekiwany wynik to kod `201 Created` oraz obiekt `DoctorDto` zgodny z danymi wejściowymi.

```
[Fact]
public async Task CreateDoctor_ReturnsCreated()
{
    var client = new EmptyDbFactory().CreateClient();
    var dto = new DoctorRegisterDto
    {
        Name = "John",
        Surname = "Smith",
        Email = "john@email.com",
        PhoneNumber = "123457777",
        Password = "Test1234!",
        SpecializationIds = new List<int> { 1 }
    };

    var response = await client.PostAsJsonAsync("/api/doctors/register", dto);
    Assert.Equal(HttpStatusCode.Created, response.StatusCode);
}
```

Przykład 2: Aktualizacja danych lekarza Test sprawdza, czy wywołanie `PUT /api/doctors/upda` poprawnie aktualizuje dane istniejącego lekarza. Oczekiwany wynik to kod `200 OK` oraz zaktualizowany obiekt `DoctorDto`.

```
[Fact]
public async Task UpdateDoctor_ReturnsOkWithUpdatedDoctor()
{
    var client = new SeededDbFactory().CreateClient();
    var existing = (await client.GetFromJsonAsync<List<DoctorDto>>("/api/doctors")).Fi

    var update = new DoctorUpdateDto
    {
        Name = "Newname",
        Surname = existing.Surname,
        PhoneNumber = existing.PhoneNumber,
        StartHour = existing.StartHour,
        EndHour = existing.EndHour
    };

    var token = TestJwtTokenHelper.GenerateTestToken("1", "john.smith@medicare.com", "
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer

    var response = await client.PutAsJsonAsync("/api/doctors/update", update);
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

Przykład 3: Reset hasła Test sprawdza, czy wywołanie `PUT /api/doctors/password-reset` zmienia hasło lekarza. Oczekiwany wynik to kod `204 No Content`, a logowanie działa tylko z nowym hasłem.

```
[Fact]
public async Task ReestPassword_ReturnsNoContent_WhenValid()
{
    var client = new SeededDbFactory().CreateClient();
    var token = TestJwtTokenHelper.GenerateTestToken("1", "john.smith@medicare.com", "
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer

    var dto = new PasswordResetDto { OldPassword = "1234", NewPassword = "NewPass123!"
    var response = await client.PutAsJsonAsync("/api/doctors/password-reset", dto);

    Assert.Equal(HttpStatusCode.NoContent, response.StatusCode);
}
```

Przykład 4: Filtrowanie lekarzy Test sprawdza, czy wywołanie `GET /api/doctors/by-filter?spe` zwraca tylko lekarzy o podanej specjalizacji i nazwisku. Oczekiwany wynik to kod `200 OK` oraz lista lekarzy spełniających oba kryteria.

```
[Fact]
public async Task GetDoctors_FilterBySurnameAndSpecialization_ReturnsOnlyMatching()
{
    var client = new SeededDbFactory().CreateClient();
    var response = await client.GetAsync("/api/doctors/by-filter?specializationID=1&&s
```

```
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

7.3 Testy jednostkowe

Walidacja hasła Testy sprawdzają, czy metoda `ValidatePassword` poprawnie wykrywa błędy takie jak zbyt krótki ciąg znaków lub brak wielkiej litery, oraz czy zwraca pustą listę dla poprawnego hasła.

```
[Fact]
public void ValidatePassword_ReturnsError_WhenTooShort()
{
    var errors = controller.ValidatePassword("abc");
    Assert.Contains("Password must be at least 8 characters long.", errors);
}

[Fact]
public void ValidatePassword_ReturnsEmpty_WhenValid()
{
    var errors = controller.ValidatePassword("Test1234!");
    Assert.Empty(errors);
}
```

Walidacja numeru telefonu Testy sprawdzają, czy metoda `ValidatePhoneNumber` odrzuca puste wartości oraz numery rozpoczynające się od zera, a akceptuje poprawne numery.

```
[Fact]
public void ValidatePhoneNumber_ReturnsError_WhenStartsWithZero()
{
    var errors = controller.ValidatePhoneNumber("012345678");
    Assert.Contains("Phone number cannot start with zero.", errors);
}

[Fact]
public void ValidatePhoneNumber_ReturnsEmpty_WhenValid()
{
    var errors = controller.ValidatePhoneNumber("123456789");
    Assert.Empty(errors);
}
```

Walidacja imienia i nazwiska Testy sprawdzają, czy metody `ValidateName` oraz `ValidateSurname` odrzucają puste wartości oraz ciągi zawierające cyfry, a akceptują poprawne dane składające się wyłącznie z liter.

```
[Fact]
public void ValidateName_ReturnsError_WhenContainsDigits()
{
```

```
    var errors = controller.ValidateName("John1");
    Assert.Contains("Name must contain only letters.", errors);
}

[Fact]
public void ValidateSurname_ReturnsEmpty_WhenValid()
{
    var errors = controller.ValidateSurname("Smith");
    Assert.Empty(errors);
}
```

Walidacja godzin pracy Testy sprawdzają, czy metoda `ValidateWorkHours` odrzuca brak danych oraz sytuację, w której godzina rozpoczęcia jest późniejsza niż godzina zakończenia, a akceptuje poprawny przedział godzin.

```
[Fact]
public void ValidateWorkHours_ReturnsError_WhenStartLaterThanEnd()
{
    var errors = controller.ValidateWorkHours(new TimeOnly(16, 0), new TimeOnly(8, 0))
    Assert.Contains("Start hour must be earlier than end hour.", errors);
}

[Fact]
public void ValidateWorkHours_ReturnsEmpty_WhenValid()
{
    var errors = controller.ValidateWorkHours(new TimeOnly(8, 0), new TimeOnly(16, 0))
    Assert.Empty(errors);
}
```

7.4 Helper

7.4.1 Testowa infrastruktura – TestJwtTokenHelper

Klasa `TestJwtTokenHelper` jest statycznym pomocnikiem wykorzystywanym w testach integracyjnych, które wymagają uwierzytelnienia użytkownika. Jej głównym zadaniem jest generowanie tokenów JWT na podstawie przykładowych danych (ID, adres e-mail, imię oraz rola użytkownika). Dzięki temu możliwe jest uruchamianie testów kontrolerów zabezpieczonych mechanizmem autoryzacji bez konieczności wykonywania pełnego procesu logowania.

Konfiguracja Helper korzysta z pliku `appsettings.json`, z którego odczytywana jest konfiguracja sekcji `Jwt`. Na tej podstawie tworzony jest obiekt `JwtTokenHelper`, który generuje podpisany token JWT zgodny z ustawieniami systemu.

Znaczenie `TestJwtTokenHelper` stanowi istotny element infrastruktury testowej, ponieważ umożliwia łatwe i szybkie uzyskanie tokenów potrzebnych w testach integracyjnych. Rozwiązanie to zapewnia spójność z mechanizmem produkcyjnym (wykorzystując

te same klucze i ustawienia JWT), a jednocześnie upraszcza proces testowania kontrolerów wymagających autoryzacji.

Podsumowanie projektu

Projekt **MediCare** stanowi kompletny system informatyczny wspierający pracę placówki medycznej. Został zaprojektowany w sposób modułowy, obejmując obsługę pacjentów, lekarzy, wizyt, pokoi oraz panel administracyjny. System zapewnia mechanizmy autoryzacji i ról użytkowników, umożliwia zarządzanie danymi oraz prezentację statystyk.

W pracy przedstawiono strukturę projektu, zastosowane technologie, architekturę systemu, implementację backendu i interfejsu użytkownika, a także mechanizmy bezpieczeństwa oraz testy. Całość kończy podsumowanie wraz z propozycjami dalszego rozwoju, takimi jak ulepszenie systemu bezpieczeństwa 2FA, zrobienie większej ilości ról użytkowników oraz funkcjonalności dla nich, więcej podsumowań w panelu administratora lub powiadomienia mailowe dla użytkowników.

Dokumentacja powstała w języku LATEX[1]

Literatura

1. Dokumentacja LaTeX – https://www.latex-project.org/ [Dostęp na dzień: 24.11.2025]
2. Dokumentacja C# - https://learn.microsoft.com/pl-pl/dotnet/csharp/ [Dostęp na dzień: 24.11.2025]
3. Dokumentacja Docker – https://docs.docker.com/ [Dostęp na dzień: 24.11.2025]
4. Dokumentacja ASP.NET Core – https://learn.microsoft.com/aspnet/core [Dostęp na dzień: 24.11.2025]
5. Dokumentacja React – https://react.dev/ [Dostęp na dzień: 24.11.2025]
6. Dokumentacja PostgreSQL – https://www.postgresql.org/docs/ [Dostęp na dzień: 24.11.2025]
7. Dokumentacja CSS - https://developer.mozilla.org/en-US/docs/Web/CSS [Dostęp na dzień: 24.11.2025]
8. Dokumentacja TypeScript - https://www.typescriptlang.org/docs/ [Dostęp na dzień: 24.11.2025]