POLITECHNIKA WROCŁAWSKA, WYDZIAŁ ELEKTRONIKI

Projektowanie telemedycznych systemów internetowych i mobilnych

Projekt

Medication prescriber

Mariusz Makuch, 241364
Kinga Marek, 235280
Mariusz Wiśniewski, 241393

Prowadzący: dr inż. Mariusz Topolski

29 października 2020

SPIS TREŚCI

1 Opis projektu

1.1 Skład grupy

- Mariusz Makuch - *programista Flutter, projektant graficznego interfejsu aplikacji*
- Mariusz Wiśniewski - *programista Flutter, Scrum Master*
- Kinga Marek - *programista .NET, projektant programistycznego interfejsu aplikacji*

1.2 Temat i cel projektu

Projekt ma na celu stworzenie dwóch aplikacji dla służby zdrowia - mobilnej oraz webowej. Mają one za zadanie usprawnienie komunikacji pomiędzy lekarzem oraz pacjentem w kwestii przepisywania leków i zaleceń, a także generację przypomnień dla pacjenta w postaci listy leków, które powinien on danego dnia przyjąć.

1.3 Zasoby ludzkie

- Doktor (użytkownik oraz administrator)
- Pacjent (użytkownik)

1.4 System dla pacjentów

Aplikacja mobilna sprawi, że pacjenci będą na bieżąco z lekami przepisywanymi im przez lekarzy z ich centrum medycznego. Ponadto, będą oni mieli możliwość wyświetlenia listy leków, które powinni spożyć danego dnia wraz z informacjami, przez kogo zostały im one przepisane oraz w jaki sposób powinni je przyjmować.

1.5 System dla lekarzy

Aplikacja webowa umożliwi lekarzowi przeglądanie listy pacjentów oraz przepisywanie im leków wraz z zaleceniami odnośnie ich stosowania. Lekarz pełni w systemie rolę administratora, więc będzie mógł on również zakładać konta nowym pacjentom lub usuwać je tym już istniejącym.

1.6 Uzasadnienie rynkowe realizacji

Wielu ludzi, bez względu na wiek, ma problemy z zapamiętaniem jakie leki, w jaki sposób i o jakiej porze należy przyjmować. Projekt ten zapewni pacjentom wygodę w powyższych kwestiach. Nie będą oni musieli starać się rozszyfrowywać odręcznego pisma na wręczonych im zaleceniach, a co ważniejsze, zniknie konieczność ciągłej ostrożności na to, żeby

ich nie zgubić.

Lekarzom natomiast, znacznie ułatwione zostanie przeglądanie profili pacjentów oraz historii przypisanych im leków. Co więcej, nie będą oni zmuszeni do odręcznego wypisywania zaleceń, co znacząco przyspieszy oraz ułatwi ich pracę.

Korzyści dla centrów medycznych wynikające z tego projektu to przede wszystkim łatwiejsze zarządzanie pacjentami oraz zmniejszenie nakładów finansowych przeznaczonych na papier. Należy pamiętać również o tym, że dokumentacja elektroniczna jest dużo trwalsza niż powszechnie stosowana dokumentacja papierowa.

2 Wymagania funkcjonalne

2.1 Logowanie

2.1.1 Identyfikacja

- Pacjent.
- Lekarz.

2.1.2 Opis wymagania

Odpowiednia aplikacja umożliwia pacjentowi lub lekarzowi zalogowanie się na swoje konto. Użytkownik loguje się podając swój numer identyfikacyjny, będący numerem pesel - dla pacjenta oraz unikalnym numerem - dla lekarza.

2.1.3 Kryterium spełnienia

Zarówno w aplikacji mobilnej, jak i webowej zaimplementowanie ekranu logowania posiadającego pole, w które użytkownik wpisuje swój numer identyfikacyjny. W przypadku podania poprawnych danych logowania pacjent zostaje przekierowany do listy leków na dziś, a lekarz - do listy pacjentów.

2.1.4 Priorytet

Wysoki.

2.2 Wyświetlenie listy leków na dziś

2.2.1 Identyfikacja

Pacjent.

2.2.2 Opis wymagania

Aplikacja mobilna umożliwia zalogowanemu pacjentowi wyświetlenie listy leków, które powinien wziąć danego dnia wraz z krótkimi informacjami odnośnie ich dawkowania oraz sugerowanej pory przyjmowania.

2.2.3 Kryterium spełnienia

Zaimplementowanie w aplikacji mobilnej ekranu wyświetlającego listę leków, które pacjent powinien przyjąć danego dnia wraz z możliwością kliknięcia na konkretny lek w celu przekierowania do ekranu z informacjami o nim.

2.2.4 Priorytet

Wysoki.

2.3 Wyświetlenie informacji o konkretnym leku

2.3.1 Identyfikacja

Pacjent.

2.3.2 Opis wymagania

Aplikacja mobilna umożliwia pacjentowi wyświetlenie informacji o konkretnym leku.

2.3.3 Kryterium spełnienia

Zaimplementowanie ekranu, który przedstawia pacjentowi informacje o wybranym leku, to jest: imię, nazwisko oraz specjalizacja lekarza, który zalecił przyjmowanie leku, okres, w którym należy go przyjmować, dawkowanie, a także sugerowane pory przyjmowania.

2.3.4 Priorytet

Wysoki.

2.4 Wyświetlenie listy pacjentów

2.4.1 Identyfikacja

Lekarz.

2.4.2 Opis wymagania

Aplikacja webowa umożliwia zalogowanemu lekarzowi wyświetlenie listy wszystkich pacjentów centrum medycznego.

2.4.3 Kryterium spełnienia

Zaimplementowanie ekranu, który przedstawia listę pacjentów, wraz z możliwością wyszukania pacjenta po imieniu lub nazwisku. Dodatkowo, kliknięcie na kartę wybranego pacjenta powinno przekierować użytkownika do ekranu z danymi pacjenta.

2.4.4 Priorytet

Średni.

2.5 Dodanie pacjenta

2.5.1 Identyfikacja

Lekarz.

2.5.2 Opis wymagania

Zalogowany w aplikacji mobilnej lekarz ma możliwość utworzenia nowego pacjenta.

2.5.3 Kryterium spełnienia

Stworzenie ekranu, w którym użytkownik może podać dane nowego pacjenta, to jest: numer pesel, imię, nazwisko oraz data urodzenia, a następnie, gdy poprawność danych zostanie potwierdzona, zapisać je w bazie danych.

2.5.4 Priorytet

Wysoki.

2.6 Wyświetlenie danych pacjenta

2.6.1 Identyfikacja

Lekarz.

2.6.2 Opis wymagania

Aplikacja webowa umożliwia lekarzowi wyświetlenie informacji o konkretnym pacjencie.

2.6.3 Kryterium spełnienia

Stworzenie ekranu, który przedstawia lekarzowi szczegółowe dane o pacjencie, to jest: numer pesel, imię, nazwisko, datę urodzenia oraz leki, które przyjmuje, wraz z możliwością wybrania leku w celu wyświetlenia o nim szczegółowych informacji.

2.6.4 Priorytet

Wysoki.

2.7 Edycja danych pacjenta

2.7.1 Identyfikacja

Lekarz.

2.7.2 Opis wymagania

Aplikacja webowa umożliwia lekarzowi zmianę danych wybranego pacjenta.

2.7.3 Kryterium spełnienia

Zaimplementowanie ekranu zawierającego formularze umożliwiające lekarzowi zmianę danych pacjenta takich jak: imię, nazwisko oraz datę urodzenia oraz zapisanie ich do bazy danych, po uprzednim zweryfikowaniu ich poprawności, a także wybór leku, który pacjent przyjmuje w celu wyświetlenia o nim szczegółowych informacji.

2.7.4 Priorytet

Średni.

2.8 Usunięcie pacjenta

2.8.1 Identyfikacja

Lekarz.

2.8.2 Opis wymagania

Aplikacja webowa umożliwia lekarzowi usunięcie wybranego pacjenta.

2.8.3 Kryterium spełnienia

Zaimplementowanie ekranu, który pozwoli lekarzowi usunąć wybranego pacjenta oraz zapisać tą informację do bazy danych.

2.8.4 Priorytet

Średni.

2.9 Przepisanie pacjentowi leku

2.9.1 Identyfikacja

Lekarz.

2.9.2 Opis wymagania

Aplikacja webowa daje lekarzowi możliwość przepisania leku wybranemu pacjentowi.

2.9.3 Kryterium spełnienia

Stworzenie ekranu, który pozwala lekarzowi na stworzenie leku poprzez podanie parametrów takich jak: nazwa, dawkowanie, okres terapii, a także sugerowane pory przyjmowania oraz, po wcześniejszej weryfikacji poprawności, zapisanie wprowadzonych informacji do bazy danych.

2.9.4 Priorytet

Wysoki.

2.10 Edycja przepisanego leku

2.10.1 Identyfikacja

Lekarz.

2.10.2 Opis wymagania

Aplikacja webowa umożliwia lekarzowi zmianę informacji odnośnie konkretnego leku przepisanego pacjentowi.

2.10.3 Kryterium spełnienia

Zaimplementowanie ekranu, który pozwoli lekarzowi na zmianę parametrów wybranego leku, takich jak: nazwa, dawkowanie, okres terapii i sugerowane pory przyjmowania, a także, po wcześniejszej weryfikacji poprawności, zapisze zmienione informacje w bazie danych.

2.10.4 Priorytet

Średni.

2.11 Usunięcie przepisanego leku

2.11.1 Identyfikacja

Lekarz.

2.11.2 Opis wymagania

Aplikacja webowa umożliwia lekarzowi usunięcie przepisanego pacjentowi leku.

2.11.3 Kryterium spełnienia

Zaimplementowanie ekranu pozwalającego lekarzowi na usunięcie przepisanego pacjentowi leku oraz zapisanie tej informacji do bazy danych.

2.11.4 Priorytet

Wysoki.

3 WYKORZYSTANE TECHNOLOGIE

System kontroli wersji, który wybraliśmy do tego projektu to *git*. W znacznym stopniu ułatwił on nam zarządzanie kodem projektowym. Dzięki narzędziu *git-submodules* byliśmy w stanie jednocześnie zarządzać oboma repozytoriami.

Serwisem, na którym zdecydowaliśmy się umieścić nasze repozytoria był *GitHub*. Decyzja umotywowana posiadanym przez ten serwis wsparciem dla platformy umożliwiającej ciągłą integrację, a także wykorzystywany wspomniany wcześniej system kontroli wersji *git*.

Do sprawnego zarządzania zasobami projektowymi wykorzystaliśmy aplikację *clickUp*, w której planowaliśmy nasze zadania, a także tworzyliśmy sprinty. Ponadto, narzędzie to pozwoliło nam na stworzenie diagramu Gantt'a, przedstawiającego harmonogram naszej pracy.

3.1 INTERFEJS PROGRAMISTYCZNY

- C# - najbardziej znany język na platformie .NET, w pełni obiektowy, oferujący bogate wsparcie dla programowania asynchronicznego i wiele bibliotek umożliwiających efektywną komunikację z bazą danych.
- .NET Core 3.1 - jeden z głównym frameworków platformy .NET. W odróżnieniu od starszego .NET Frameworka umożliwia rozwijanie aplikacji wielo-platformowych oraz posiada wiele poprawek wydajnościowych takich jak np. nowe wersje zoptymalizowanych paczek *NuGet*.
- MS SQL Server - komercyjny silnik bazodanowy stworzony przez firmę *Microsoft*. Wykorzystany w ramach licencji studenckiej. Wybrany ze względu na niski koszt udostępniania serwera MS SQL Server na platformie Azure.
- Azure - platforma chmurowa firmy Microsoft umożliwiająca szybki i prosty proces wdrożenia stworzonego API oraz bazy danych do internetu. Dzięki temu część zespołu tworząca aplikację mobilną i webową mogła pracować na wspólnej bazie danych posiadając ciągły dostęp do API bez konieczności instalowania i uruchamiania dodatkowego oprogramowania na własnych urządzeniach.
- Visual Studio 2019 (IDE) - zintegrowane środowisko programistyczne, oferujące wysokie wsparcie dla języka C# i platformy .NET. Umożliwia inteligentne edytowanie, przeszukiwanie kodu, a także *IntelliSense*, czyli automatyczne uzupełnianie i podpowiadanie nazw zmiennych, funkcji i metod. Środowisko to jest dobrze zintegrowane z platformą Azure i umożliwia udostępnianie nowych wersji aplikacji za pomocą wbudowanych funkcji.

3.2 APLIKACJA WEBOWA ORAZ MOBILNA

- CircleCI - platforma, którą wykorzystaliśmy do ciągłego łączenia (CI) oraz dostarczania (CD). Analiza poprawności kodu, jego formatu, budowanie go na utworzonym

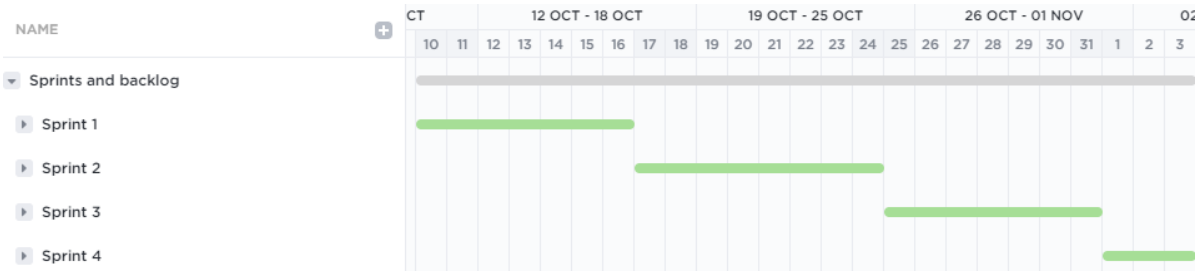
do tego kontenerze, a także automatyczna generacja plików instalacyjnych dla aplikacji w znaczącym stopniu ułatwiła nam zarządzanie jakością projektu. To wszystko sprawiło, że na główną gałąź projektową nie mógł zostać dostarczony niedziałający kod.

- Dart - język programowania, który jest wymagany przez zestaw deweloperski *Flutter*.
- Flutter - zestaw deweloperski oprogramowania UI, który umożliwił nam jednoczesne projektowanie graficznego interfejsu użytkownika dla aplikacji mobilnej (na systemy operacyjne Android i iOS) oraz webowej.
- Android Studio (IDE) - zintegrowane środowisko programistyczne, które dostarczyło nam narzędzia do debugowania oraz poprawy składni kodu, a także emulator urządzenia z systemem Android. Dzięki dostarczanym przez nie opcjom podpowiedzi oraz skrótom klawiszowym mogliśmy w znacznym stopniu przyspieszyć proces pisania kodu.

4 Harmonogram implementacji

Do zarządzania zadaniami w projekcie wykorzystaliśmy aplikację *ClickUp*, która opisana została w sekcji 3. W celu lepszej organizacji pracy, do realizacji projektu wykorzystaliśmy podejście iteracyjne, które polegało na wyznaczeniu krótkich okresów rozliczeniowych, przed którymi organizowaliśmy planowanie, aby wybrać zadania, które wchodziły w zakres sprintu (iteracji), a po których podsumowywaliśmy, podczas retrospektywy, co udało się wykonać w wyznaczonym do tego czasie.

Praca nad projektem zajęła nam niecały miesiąc, co przekłada się na 4 iteracje. Na obrazku 4.1 przedstawiony został pełny harmonogram sprintów.



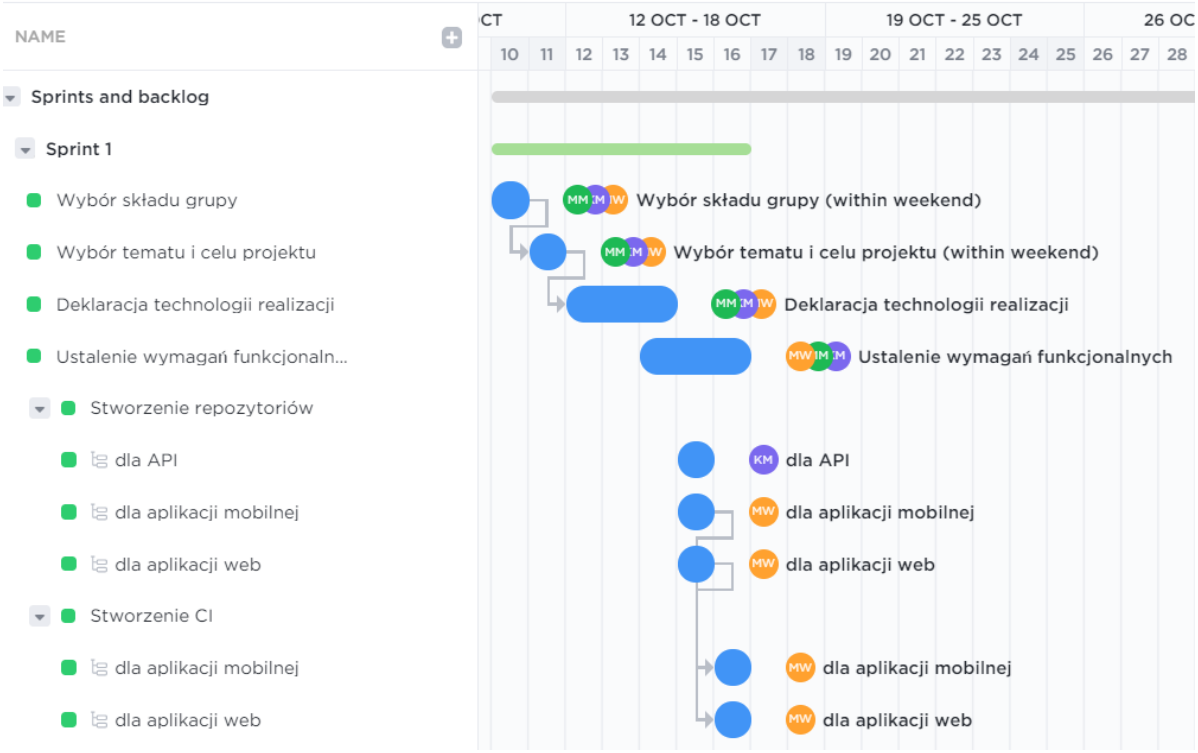
Rysunek 4.1: Diagram Gantt'a przedstawiający wszystkie sprinty

Inicjały przedstawione na diagramach Gantt'a to odpowiednio:

- MM (zielony) - Mariusz Makuch,
- KM (fioletowy) - Kinga Marek,
- MM (pomarańczowy) - Mariusz Wiśniewski.

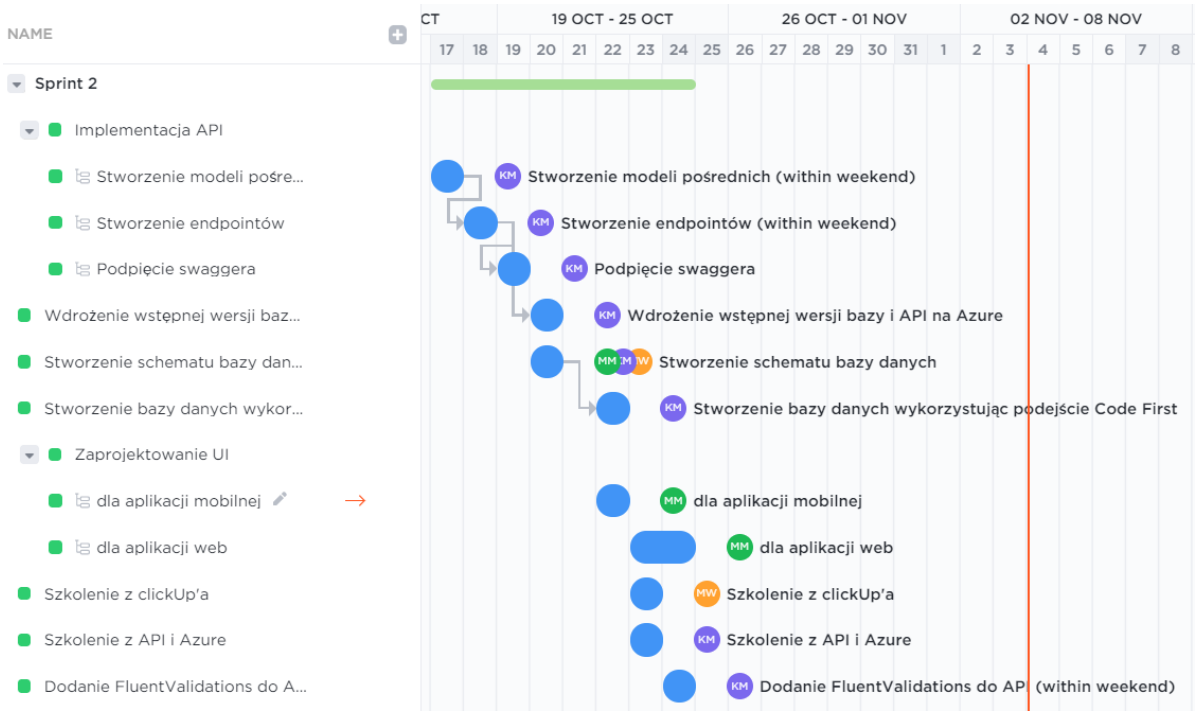
Przedstawiają one osoby odpowiedzialne za poszczególne zadania.

Podczas pierwszego ze sprintów skupiliśmy się na zadaniach organizacyjnych takich jak: wybór tematu i celu projektu, ustalenie wymagań funkcjonalnych aplikacji, czy też deklaracja technologii realizacji. Diagram Gantt'a dla tego sprintu przedstawiono na Rysunku 4.2.



Rysunek 4.2: Diagram Gantt'a - Sprint 1

Druga iteracja zakładała już implementację API, wstępne zaprojektowanie graficznego interfejsu użytkownika oraz zrobienie szkoleń dla członków zespołu, prezentujących zastosowane rozwiązania. Diagram Gantt'a dla tej iteracji zaprezentowany jest na Rysunku 4.3.

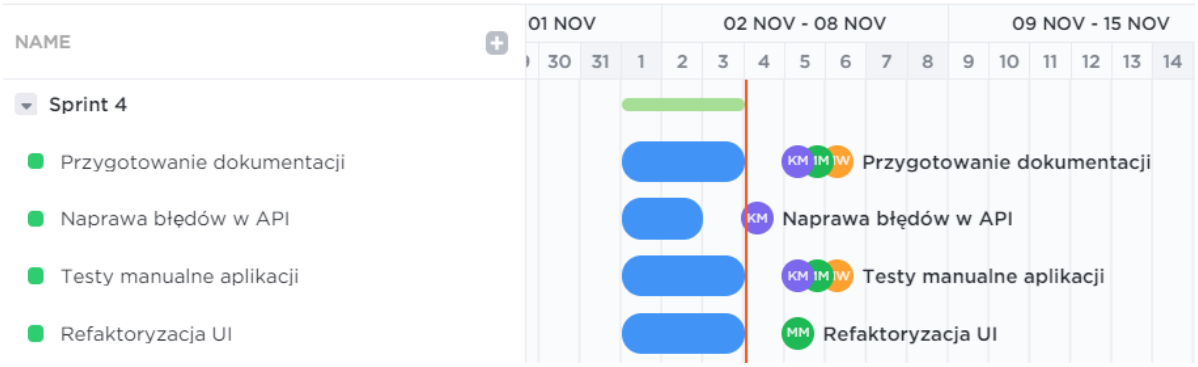


Rysunek 4.3: Diagram Gantt'a - Sprint 2

Na trzeci sprint zaplanowaliśmy implementację graficznego interfejsu użytkownika, a także jego integrację z API. Rysunek 4.4 przedstawia harmonogram tej iteracji.

Rysunek 4.4: Diagram Gantt'a - Sprint 3

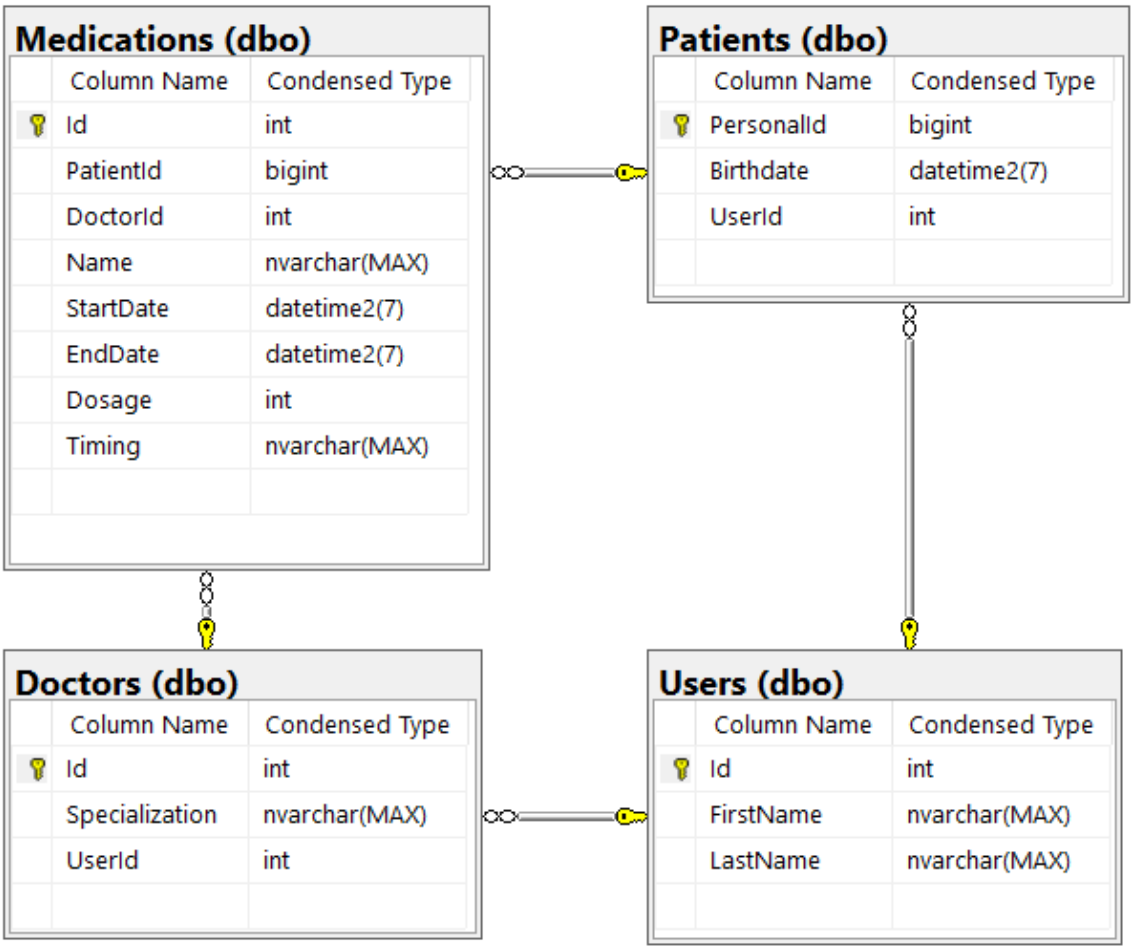
Ostatni ze sprintów, czyli sprint 4 skupił się na finalizacji projektu, w skład której wchodziło przygotowanie dokumentacji, naprawa występujących błędów oraz manualne testy aplikacji mobilnej oraz webowej. Diagram Gantt'a dla tego sprintu przedstawia Rysunek 4.5.



Rysunek 4.5: Diagram Gantt'a - Sprint 4

5 Architektura informacji

5.1 Schemat bazy danych



Rysunek 5.1: Schemat bazy danych

5.2 Utworzenie bazy danych

Baza danych MS SQL Server została utworzona na platformie Azure. Jej stworzenie rozpoczęto od przygotowania potrzebnej grupy zasobów (ang. *resource group*), czyli kontenera, który pozwala grupować i porządkować zasoby tworzone na platformie. Kolejnym krokiem było utworzenie serwera bazodanowego SQL Server wewnątrz stworzonej wcześniej grupy zasobów. Utworzony serwer domyślnie jest chroniony przez zaporę (ang. *firewall*) i nie jest publicznie dostępny. Dostęp do serwera udostępniony jest tylko dla serwisów wewnątrz stworzonej grupy zasobów oraz dla jednego z naszych urządzeń trakcie przeprowadzania migracji tworzącej bazę danych, co opisano w kolejnym punkcie. Ostatnim krokiem było stworzenie właściwej bazy danych w najniższej warstwie cenowej. Ogra-

niczenia nałożone przez tę warstwę to: 2GB pamięci dla danych oraz 5 DTU (ang. *Database Tranasction Unit*).

Do utworzenia modelu bazy danych skorzystano z *Entity Framework Core*, czyli narzędzia do mapowania obiektowo-relacyjnego (ang. *Object-Relational Mapping - ORM*). Narzędzie to pozwala na odzworowywanie relacyjnej bazy danych w postaci modelu obiektowego klas w języku C#. Do stworzenia modelu bazy danych zdecydowano się wykorzystać podejście *Code First*, które na podstawie najpierw zaprojektowanych klas tworzy model logiczny bazy danych. Podejście to wymaga większej znajomości *Entity Framework*, lecz pozwala na elastyczne projektowanie klas odwzorowujących encje bazy danych oraz na wygodne zarządzanie migracjami, czyli zmianami w bazie danych.

5.3 Opis encji w bazie danych

5.3.1 Tabela Users

Tabela zawierająca podstawowe informacje dotyczących użytkowników aplikacji - są w niej zawarte podstawowe dane użytkownika takie jak imię i nazwisko.

Tablica 5.1: Tabela użytkowników

Atrybut	**Klucz**	**Typ**	**Opis**
Id	PK	int	Identyfikator użytkownika
FirstName		nvarchar(MAX)	Imię użytkownika
LastName		nvarchar(MAX)	Nazwisko użytkownika

5.3.2 Tabela Patients

Tabela zawierająca informacje o pacjentach - są w niej zawarte podstawowe dane osobowe.

Tablica 5.2: Tabela pacjentów

Atrybut	**Klucz**	**Typ**	**Opis**
PersonalId	PK	bigint	PESEL pacjenta
Birthdate		datetime2(7)	Data urodzenia pacjenta
UserId	FK	int	Identyfikator klucza obcego do tabeli Users

5.3.3 Tabela Doctors

Tabela zawierająca informacje o lekarzach - są w niej zawarte dane użytkownika oraz nazwa specjalizacji lekarza.

Tablica 5.3: Tabela lekarzy

Atrybut	Klucz	Typ	Opis
Id	PK	int	Identyfikator lekarza
Specialization		nvarchar(MAX)	Specjalizacja lekarza
UserId	FK	int	Identyfikator klucza obcego do tabeli Users

5.3.4 Tabela Medications

Tabela zawierająca informacje o lekach - są w niej zawarte informacje dotyczące nazwy, okresu przyjmowania, ilości oraz sposobu dawkowania. Ponadto, istnieją atrybuty odpowiadające za informacje przez kogo i dla kogo został przypisany lek.

Tablica 5.4: Tabela leków

Atrybut	Klucz	Typ	Opis
Id	PK	int	Identyfikator leku
PersonalId	FK	bigint	Identyfikator pacjenta któremu przypisano lek
DoctorId	FK	int	Identyfikator lekarza przypisującego lek
Name		nvarchar(MAX)	Nazwa leku
StartDate		datetime2(7)	Data rozpoczęcia przyjmowania leku
EndDate		datetime2(7)	Data zakończenia przyjmowania leku
Dosage		int	Dzienna dawka leku
Timing		nvarchar(MAX)	Sposób dawkowania leku w zależności od posiłku

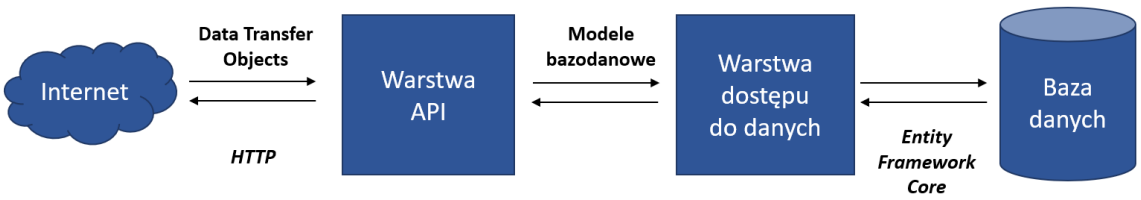
6 Projekt interfejsu programistycznego

Stworzony interfejs programistyczny (ang. *API - Apllication Programming Interface*) służy jako pośrednik do komunikacji ze wspólną bazą danych dla aplikacji mobilnej i webowej.

API zaprojektowane zostało w stylu architektury REST (ang. *REpresentational State Transfer*). Oznacza to między innymi, że do komunikacji używane są zapytania protokołu HTTP, a do opisu zasobów (informacji) udostępnianych przez API użyto jednolitego interfejsu. W tym celu endpointy, czyli adresy URL (ang. *Uniform Resource Locator*) pod jakie można wysyłać zapytania zostały pogrupowane. W obrębie jednej grupy zostały zdefiniowane podstawowe metody HTTP takie jak: GET, PUT, POST, DELETE. Odzwierciedlenie tego jest widoczne w kodzie API - każdy zasób posiada swój własny kontroler definiujący obsługę poszczególnych zapytań. Ponadto każde zapytanie kierowane do API jest bezstanowe. Oznacza to, że każda odpowiedź zawiera pełny zestaw informacji na temat danego zasobu, a serwer API nie przechowuje informacji na temat sesji użytkownika.

W celu rozdzielenia odpowiedzialności projekt API podzielono na warstwy. Najniższa z nich - warstwa dostępu do danych posiada modele bazodanowe oraz kod potrzebny do

generowania migracji używając *Entity Framework Core*. Warstwa wyższa posiada kontrolery, które służą do obsługi zapytań HTTP. Znajduje się w niej więc logika biznesowa oraz modele pośrednie niezależne od bazodanowych - tzw. DTOs (ang. *Data Transfer Objects* - obiekty do transferu danych). Dzięki zastosowanemu podziałowi modele z bazy danych lub modele udostępniane przez API mogą zmieniać się niezależnie. Schemat przedstawiający warstwy projektu API został przedstawiony na Rysunku 5.1.

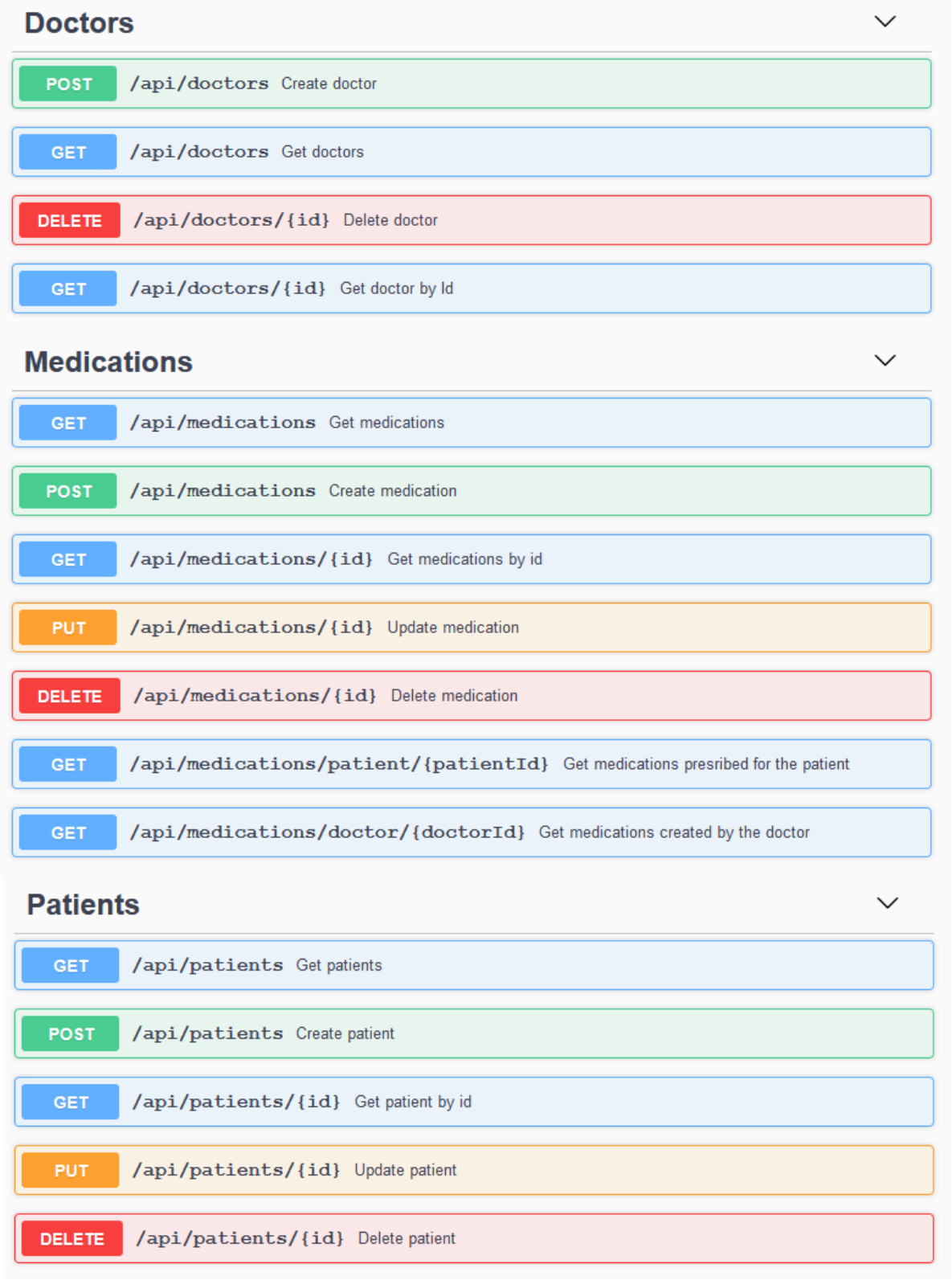


Rysunek 6.1: Schemat API

Testowanie API było możliwe dzięki narzędziu do wizualizacji i opisu RESTowych API *Swagger*. Dzięki paczkce NuGetowej *Swashbuckle* dokumentacja została wygenerowana bezpośrednio z kodu kontrolerów, po dopisaniu odpowiednich adnotacji i komentarzy. Na Rysunku 6.2 przedstawiono komplet dokumentacji stworzonej używając *Swaggera*. Dodatkową rzeczą dodaną do przedstawionej dokumentacji API, której nie widać na załączonym Rysunku, są opisy parametrów oraz spis kodów odpowiedzi, jakie mogą zostać zwrócone w odpowiedzi na zapytania.

Strona wygenerowana przez *Swaggera* umożliwia przetestowanie działania API po kliknięciu w wybrany endpoint, naciśnięciu przycisku 'Try it out' i jeśli potrzebne, uzupełnieniu wymaganych danych. Następnie możemy wysłać zapytanie do serwera API.

Innym branym pod uwagę narzędziem do testowania API był *Postman*. Jest to narzędzie dające większą możliwość modyfikowania tworzonych zapytań oraz przechowujace ich historię. Zdecydowano się jednak na *Swaggera* ze względu na dokumentację generowaną automatycznie po zainstalowaniu pakietu *Swashbuckle* do projektu. Wchodząc na stronę wygenerowaną przez *Swaggera*, możemy zobaczyć pełną listę endpointów, ich parametrów i ich opis, co znacznie ułatwia i przyśpiesza testowanie.



Rysunek 6.2: Dokumentacja API przygotowana używając narzędzia *Swagger*

7 Graficzny projekt interfejsu użytkownika

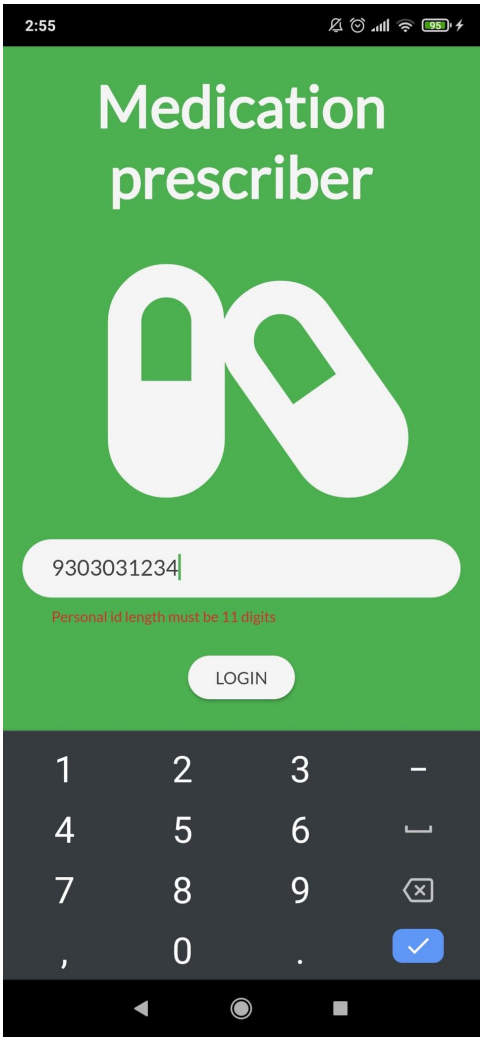
7.1 Aplikacja mobilna

7.1.1 Ekran logowania

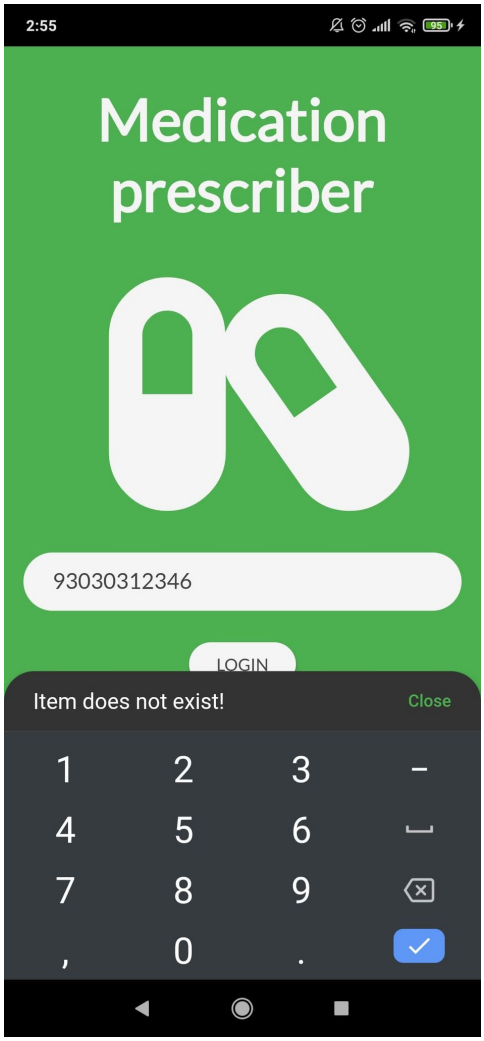
Pierwszym widokiem, który użytkownik zobaczy po zainstalowaniu i uruchomieniu aplikacji jest ekran logowania. W jego górnej części znajduję się nazwa oraz logo aplikacji. Poniżej jest pole tekstowe, w którym użytkownik wpisuje swój numer PESEL. W celu jego zatwierdzenia należy nacisnąć przycisk LOGIN.

Rysunek 7.1: Ekran logowania

W przypadku niepowodzenia - wprowadzony login ma nieodpowiednią długość lub użytkownik o takim loginie nie istnieje w bazie - zostają wyświetlone stosowne komunikaty o błędzie.



Rysunek 7.2: Widok komunikatu o nieodpowiedniej długości numeru PESEL



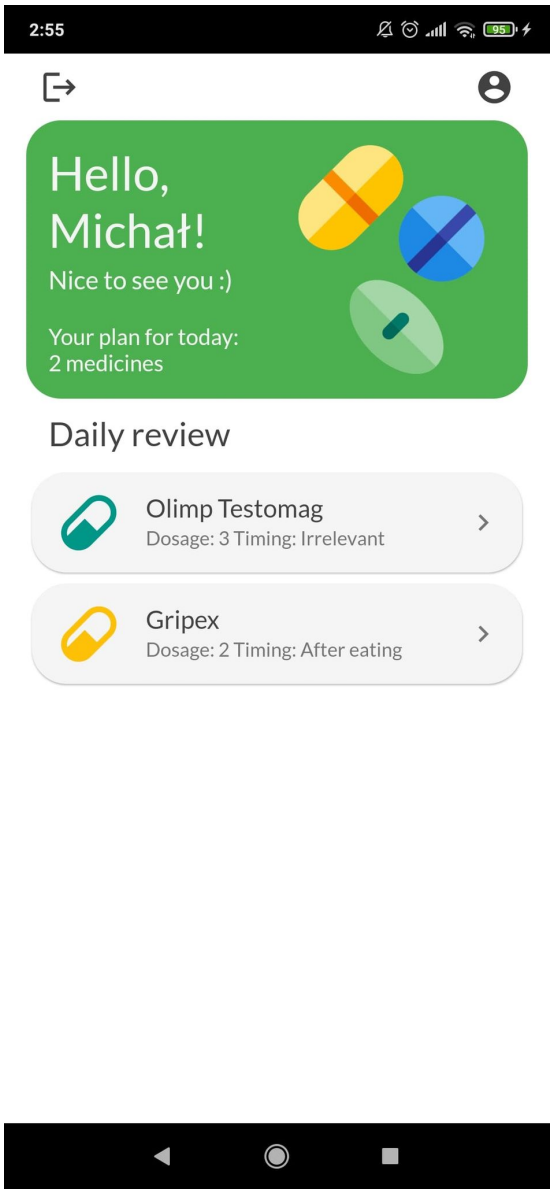
Rysunek 7.3: Widok komunikatu braku pacjenta o podanym numerze PESEL

Natomiast w przeciwnym przypadku użytkownik zostaje zalogowany i ukazuje mu się kolejny ekran.

7.1.2 Ekran głowy

Użytkownikowi po poprawnym zalogowaniu ukazuję się główny widok aplikacji. Na samej górze dostępne są dwa przyciski odpowiadające za wylogowanie się użytkownika i po-

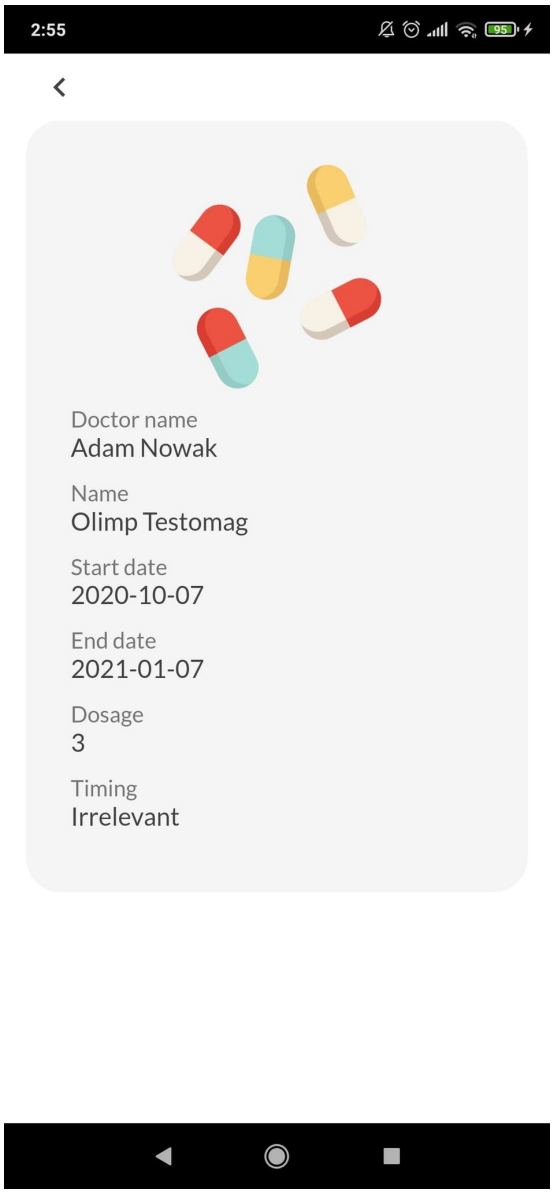
wrót do poprzedniego widoku (Rysunek 7.1.) oraz przejście do widoku szczegółów profilu pacjenta (Rysunek 7.6). Poniżej znajduję się komunikat witający użytkownika w aplikacji oraz przypominający ile ma on lekarstw do wzięcia. W dolnej części widoku ekranu znajduję się lista leków na dziś. Użytkownik może ją w każdej chwili odświeżyć poprzez pociągnięcie palcem w dół. Lista ta zostanie wówczas zaktualizowana tak, aby odwzorowywać obecny stan bazy danych. Przedstawia ona najpotrzebniejsze informacje o leku - zawierające jego nazwę, ilość oraz sposób dawkowania. W celu uzyskania większej ilości informacji o przepisanym leku należy w niego kliknąć. Tym sposobem pojawia się kolejny widok.



Rysunek 7.4: Ekran logowania

7.1.3 Ekran szczegółów leku

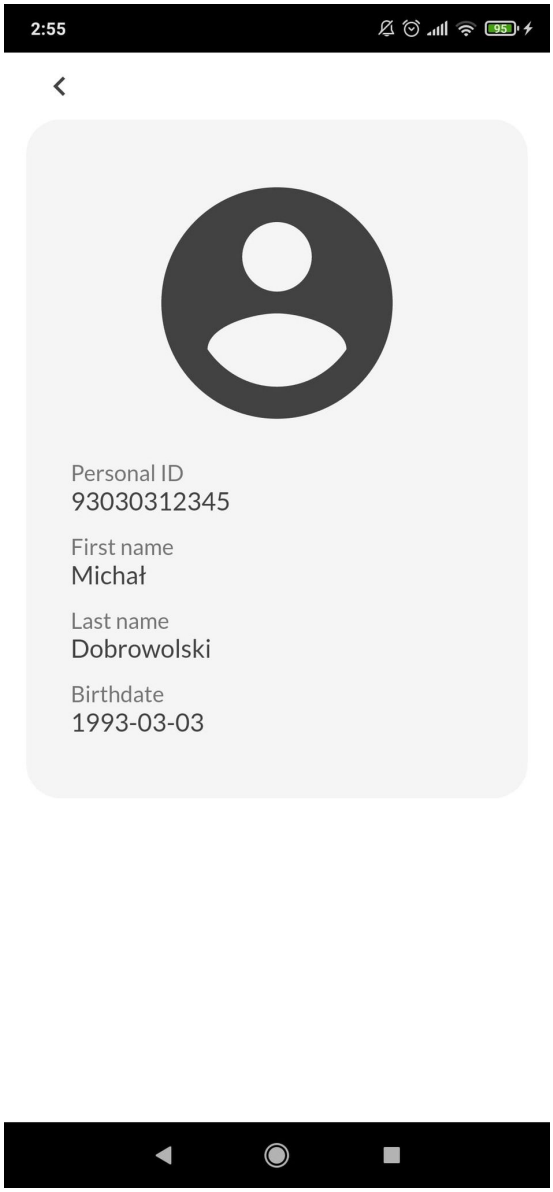
Kolejny widok zawiera szczegółowe informacje dotyczące leku, które uzupełniają brakujące dane z listy. Dodatkowymi atrybutami są: data rozpoczęcia i zakończenia przyjmowania leku oraz imię i nazwisko lekarza, który ten lek przepisał. W lewym górnym rogu znajduję się strzałka nawigacyjna, która umożliwia powrót do poprzedniego ekranu (Rysunek 7.4.).



Rysunek 7.5: Ekran szczegółów dotyczących wybranego leku

7.1.4 Ekran danych pacjenta

Ostatnim możliwym ekranem, do którego pacjent ma dostęp z poziomu aplikacji to widok danych osobowych zalogowanego użytkownika. Jest on zbliżony wyglądem do poprzedniego, omówionego powyżej. Jedyną różnicą jaką można zauważyć jest ikona profilu oraz dane takie jak: PESEL, imię, nazwisko czy data urodzenia.



Rysunek 7.6: Ekran danych pacjenta

7.2 Aplikacja webowa

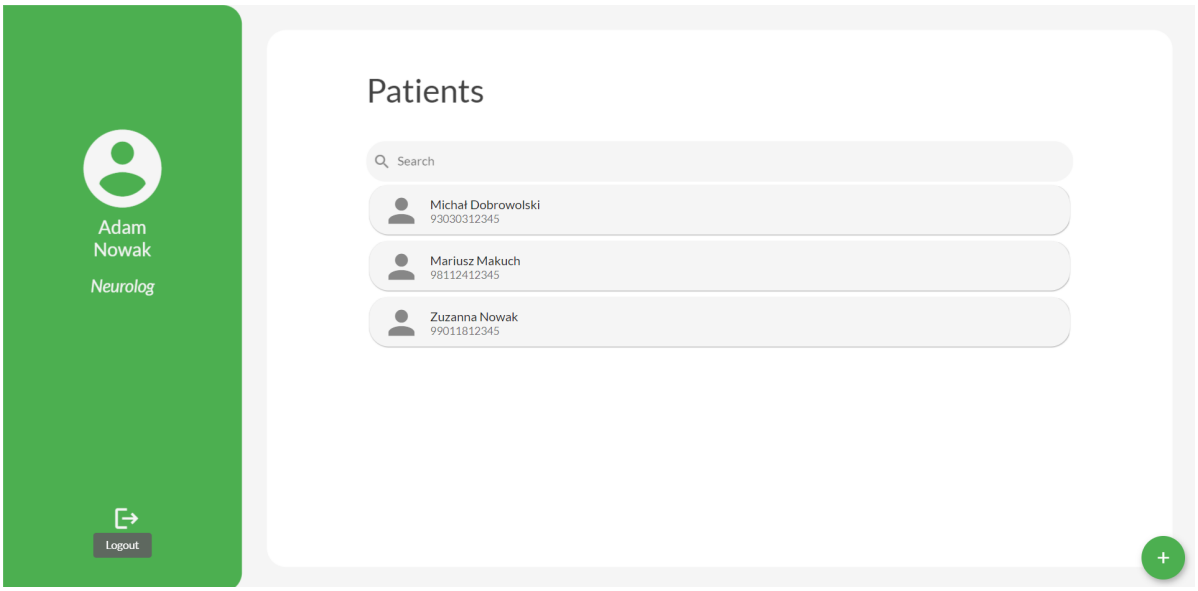
7.2.1 Ekran logowania

Podobnie jak w aplikacji mobilnej pierwszym ekranem po uruchomieniu aplikacji w przeglądarce jest widok logowania. Warstwa wizualna aplikacji webowej i mobilnej jest identyczna poza dopiskiem, że aplikacja webowa to system zarządzania dla lekarza. Powiadomienia o błędach występujących podczas logowania są wyświetlane w takiej samej formie jak na Rysunku 7.2 oraz 7.3.

Rysunek 7.7: Ekran logowania

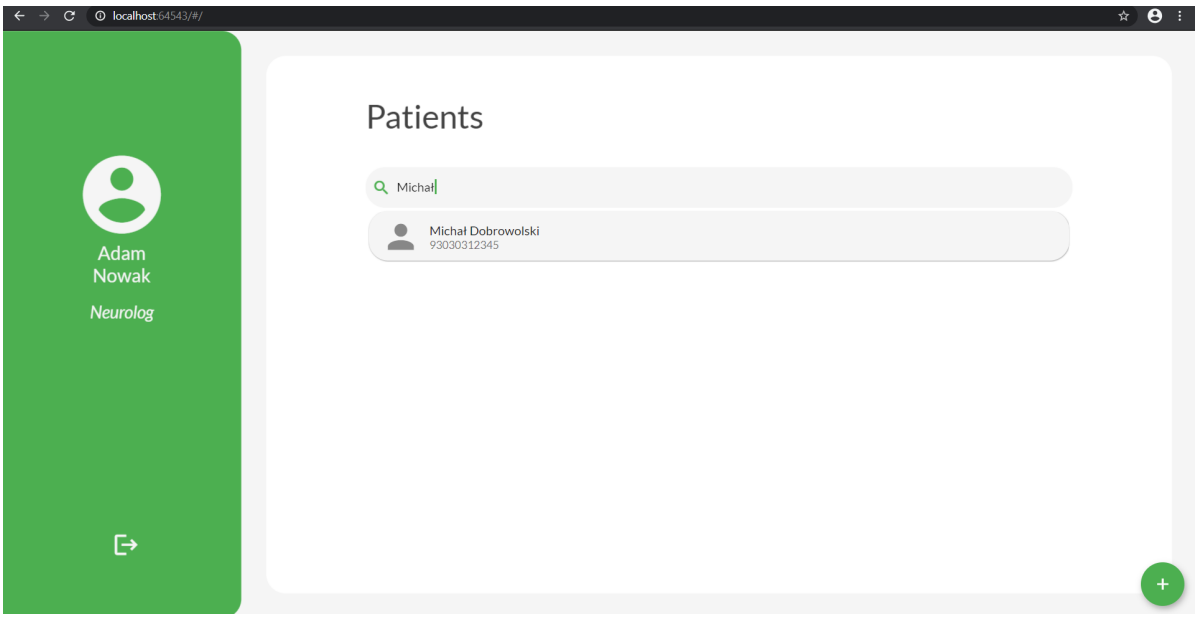
7.2.2 Ekran główny

Po poprawnym zalogowaniu lekarz jest przenoszony do ekranu głównego. Widok ten możemy podzielić na dwie sekcje: lewą, w której znajduję się panel lekarza oraz prawą, z listą wszystkich dostępnych pacjentów w systemie. Na panelu lewym ukazane są podstawowe informacje takie jak: imię, nazwisko, specjalizacja lekarska oraz przycisk, który umożliwia wylogowanie się z konta i powrót do ekranu logowania. Panel ten jest powielany i widoczny jest na każdym kolejnym widoku.



Rysunek 7.8: Ekran głowny

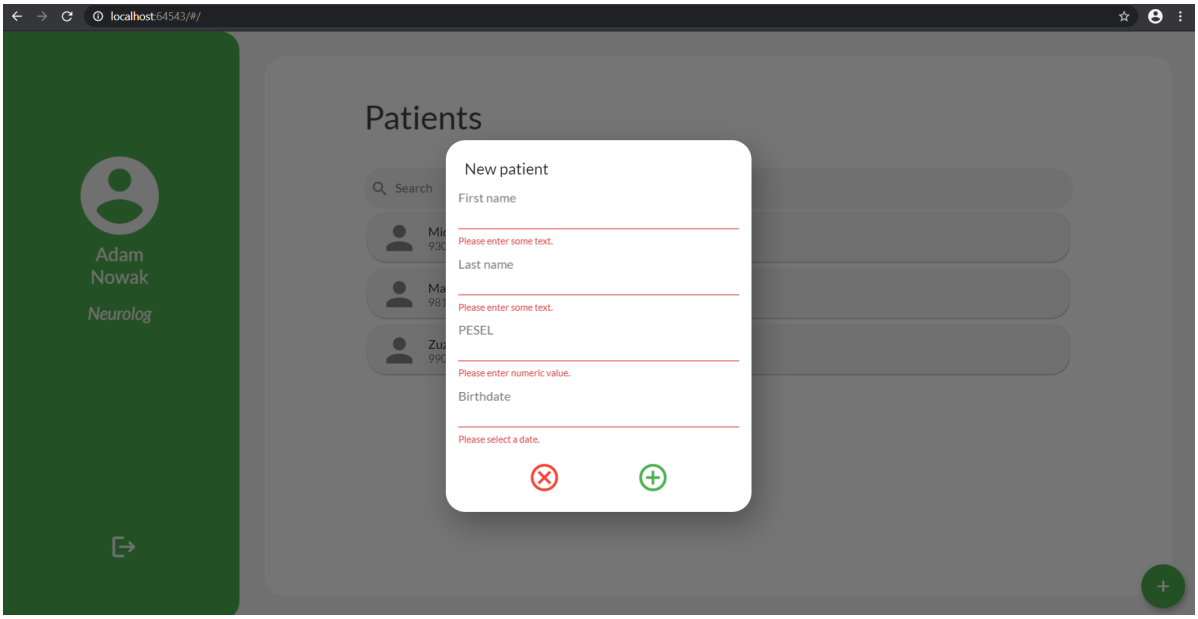
W prawej części widoku znajduję się jeszcze wyszukiwarka dodana w celu łatwiejszego znajdowania konkretnego użytkownika używając jego imienia i nazwiska.



Rysunek 7.9: Wyszukiwanie pacjenta

Ponadto, lekarz z poziomu tego widoku ma możliwość dodania nowego użytkownika. Jest to wykonalne poprzez kliknięcie okrągłego przycisku ze znakiem +. Po wykonaniu akcji użytkownikowi pokaże się okno dialogowe z możliwością wprowadzenia danych. Każde pole jest sprawdzane pod względem poprawności formatu. W celu dodania nowego pa-

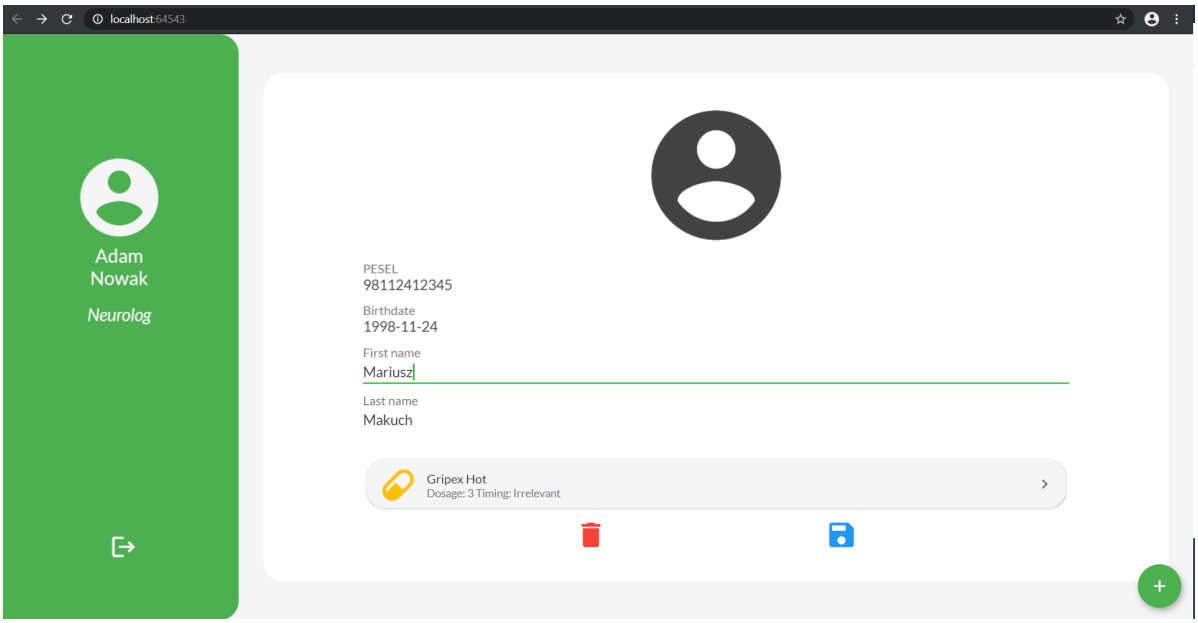
cjenta do bazy należy użyć zielonego przycisku. Spowoduje to zamknięcie okna dialogowego oraz zaktualizowanie widoku głównego - rozszerzy się on o nowego użytkownika. Lekarz ma również możliwość zrezygnowania z akcji i zamknięcia okna poprzez kliknięcie w czerwony przycisk.



Rysunek 7.10: Okno dodania nowego pacjenta z aktywną walidacją

7.2.3 Ekran szczegółów pacjenta

Po naciśnięciu pacjenta znajdującego się na liście głównego ekranu lekarzowi ukazuje się widok dokładnych danych pacjenta oraz lista wszystkich przypisanych mu leków. Lekarz w tym ekranie ma możliwość zmiany imienia i nazwiska użytkownika poprzez wprowadzenie nowych wartości i zapisanie (niebieski przycisk z dyskietką), dodania nowego leku (zielony przycisk ze znakiem plus) a także usunięcia pacjenta z systemu.

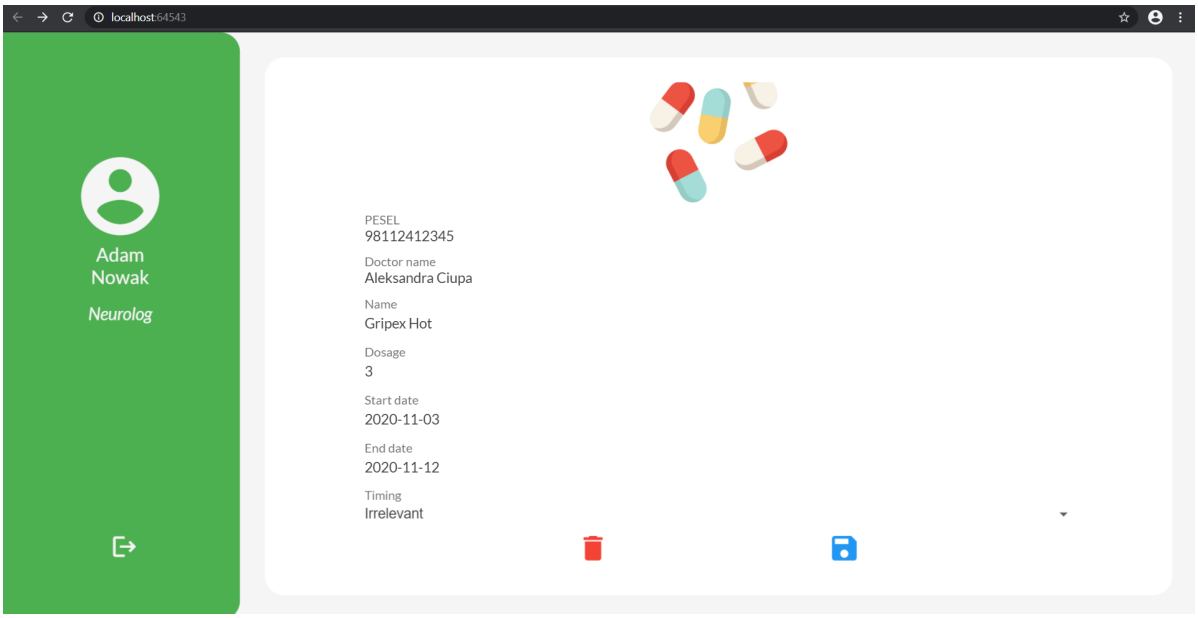


Rysunek 7.11: Widok szczegółów pacjenta z aktywną edycją imienia

Powrót do widoku głównego z poziomu ekranu danych o pacjencie jest możliwy przy użyciu klawisza wstecz w przeglądarce.

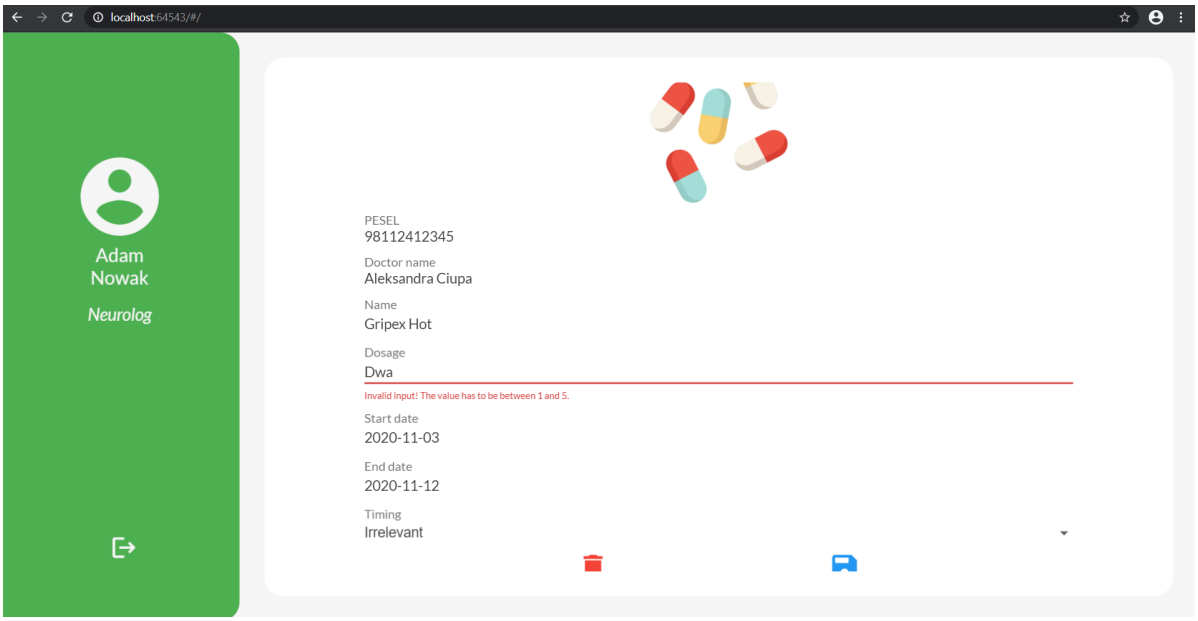
7.2.4 Ekran szczegółów leku

Ostatnim widokiem w aplikacji webowej jest widok szczegółów leku. Jest on dostępny po wybraniu leku z listy leków przepisanych pacjentowi (Rysunek 7.6.) Posiada on wszystkie funkcjonalności, które znajdują się w widoku szczegółów pacjenta (Rysunek 7.2.3).



Rysunek 7.12: Widok szczegółów leku

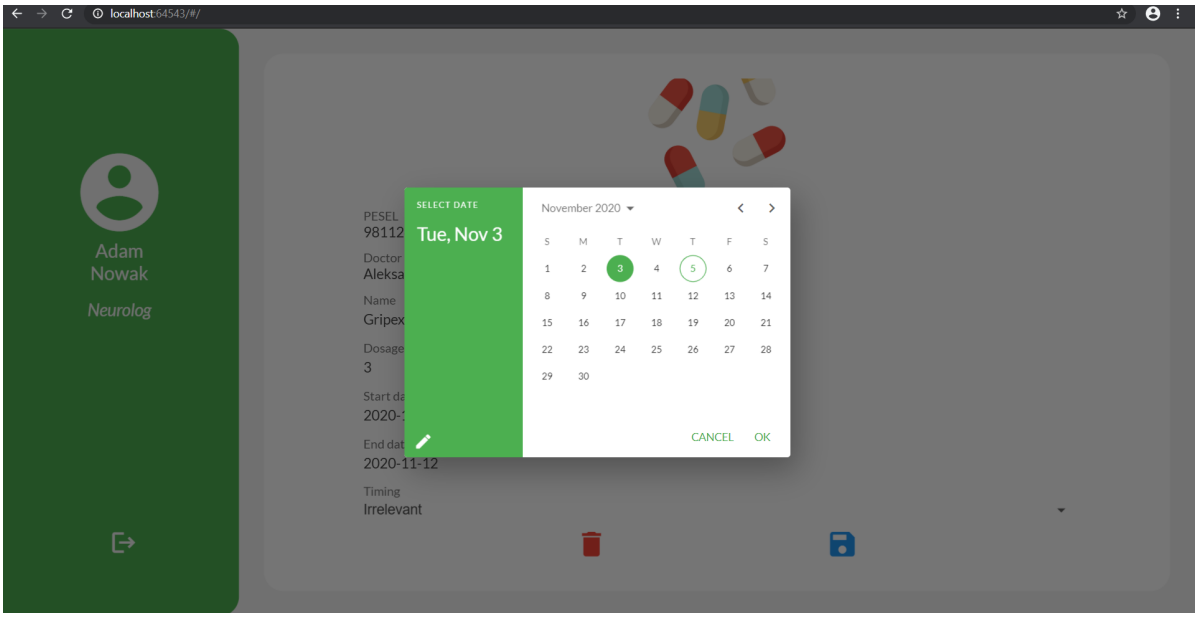
Każda próba zapisania danych o nieprawidłowym formacie skutkuje powiadomieniem o błędzie.



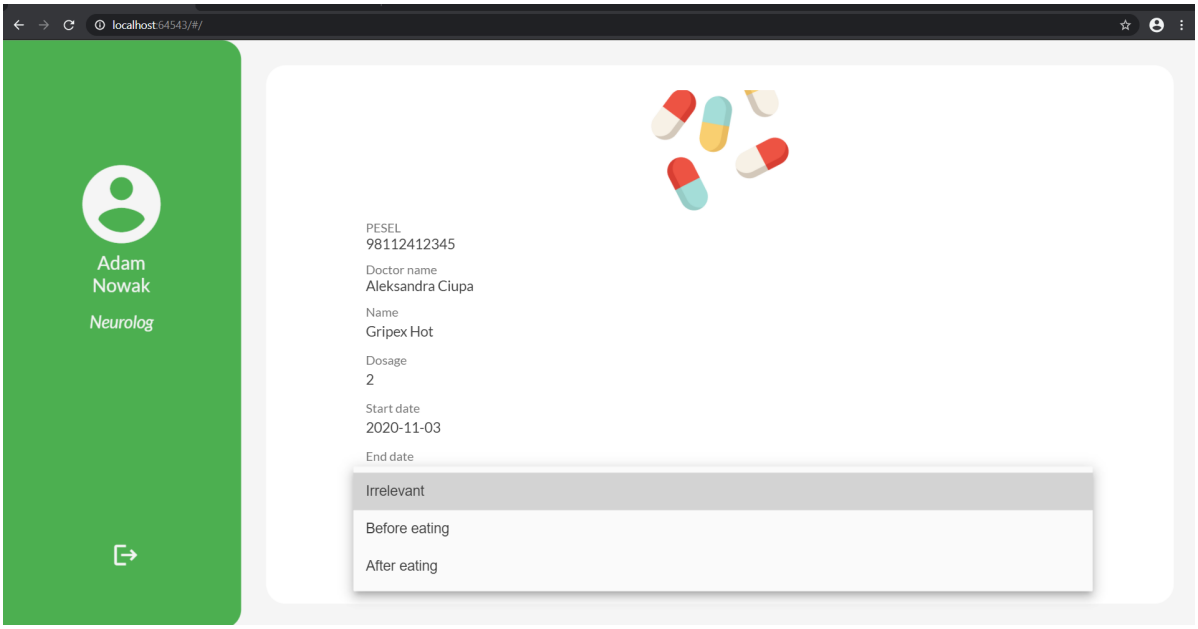
Rysunek 7.13: Próba zapisania zedytowanego leku z błędnymi danymi

Podczas wprowadzania nowej daty lub jej modyfikowania dostępny jest *DatePicker*. Umożliwia on wybranie konkretnej dany z kalendarza z przedziału od 1920 roku do 2050 roku (rys 7.14.) Kolejnym *widgetem*, który jest stosowany podczas wprowadzania danych i edycji jest

DropDownButton. Jest to rozwijana lista zawierająca trzy wartości: *Irrelevant, After eating, Before eating* (Rysunek 7.15).



Rysunek 7.14: Okno dialogowe umożliwiające wybranie daty z kalendarza



Rysunek 7.15: Lista wyboru dawkowania w zalezności od posiłku

8 Opis implementacji

8.1 Interfejs programistyczny

8.1.1 Warstwa API

Warstwa API zawiera w sobie całą logikę biznesową. Punktem startowym aplikacji jest plik *Startup.cs*, który definije tzw. *middleware*, czyli warstwy pośredniczące w przetwarzaniu zapytań, które docierają do API. Jest to miejsce, gdzie został dodany między innymi *Swagger* oraz biblioteka automatycznie sprawdzająca poprawność modeli używanych do tworzenia i edycji zasobów.

Kolejnym ważnym punktem jest folder *Controllers*, który zawiera w sobie wszystkie kontrolery grupujące endpointy. Na Listingu 1 przedstawiono fragment kontrolera *DoctorsController* opisującego obsługę metody HTTP DELETE. W linii siódmej Listingu dodana została adnotacja wskazująca jaką metodę HTTP obsługuje ta funkcja. Argument "{id}"modyfikuje adres pod jakim będzie obsługiwane zapytanie dodając do podstawowego adresu kontrolera pole na id po znaku '/'. Kolejne linie są adnotacjami dla *swaggera* określającymi jakie kody odpowiedzi mogą zostać zwrócone (dodatkowy opis wyświetlany przez *swaggera* dla kodu 404 umieszczony został w komentarzu w linii szóstej). Warto zwrócić uwagę, że dzięki modyfikatorowi *async* oraz operatorowi *await* zapytanie wykonywane przez aplikację działa w sposób asynchroniczny.

Listing 1: Fragment kontrolera *DoctorController.cs* - metoda DELETE

```
1         /// <summary>
2         /// Delete doctor
3         /// </summary>
4         /// <param name="id">Id of the doctor</param>
5         /// <returns></returns>
6         /// <response code="404">If doctor with given id does not
              ↪ exist</response>
7         [HttpDelete("{id}")]
8         [ProducesResponseType((int)HttpStatusCode.OK)]
9         [ProducesResponseType((int)HttpStatusCode.NotFound)]
10        public async Task<IActionResult> DeleteAsync(int id)
11        {
12            var doctorToDelete = _context.Doctors.FirstOrDefault(x
                  ↪ => x.Id == id);
13            if (doctorToDelete is null)
14            {
15                return NotFound();
16            }
17
18            _context.Doctors.Remove(doctorToDelete);
19            await _context.SaveChangesAsync();
20            return Ok();
21        }
```

Listing 1 zawiera też przykład zapytań LINQ (ang. *Language INtegrated Query*), które są

charakterystyczną cechą języka C#. W linii 12 takie zapytanie jest używane do pobrania z bazy danych doktora o wybranym id. Metody LINQ wraz z *Entity Framework Core* pozwalają pracować na tabelach w bazie danych tak samo jak na kolekcjach obiektów. Zastosowana metoda LINQ *FirstOrDefault* zwróci pierwsze wystąpienie pasujące do zastosowanego filtru lub *null*, jeśli nie znajdzie żadnego dopasowania.

Modele *DTO* używane do tworzenia i edytowania danych zostały wyposażone w walidatory stworzone ze wsparciem paczki NuGet *FluentValidations*. Paczka ta pozwala w prosty sposób definiować warunki, jakie muszą być spełnione przez otrzymane dane. Stworzone walidatory można używać w formie jawnej, wewnątrz kontrolerów lub dodać w formie midlleware, które dokona sprawdzenia modelu zanim jeszcze zapytanie zostanie przechwycone przez wybraną metodę kontrolera. W projekcie zdecydowano się na używanie *FluentValidatons* w roli *midlleware*, aby uniknąć pomyłek, jakie mogą wynikać z konieczności ręcznego dodawania jawnych wywołań walidatorów w kontrolerze. Przykład walidatora do modelu *DTO* przedstawiono na Listingu 2.

Listing 2: Walidacja modelu do tworzenia leku z użyciem *FluentValidatons*

```
1   public class CreateMedicationDtoValidator : AbstractValidator<
        ↪ CreateMedicationDto>
2   {
3       public CreateMedicationDtoValidator()
4       {
5           RuleFor(x => x.Name).NotEmpty();
6           RuleFor(x => x.StartDate).NotEmpty();
7           RuleFor(x => x.EndDate).NotEmpty();
8           RuleFor(x => x.Timing).NotEmpty();
9           RuleFor(x => x.Dosage).GreaterThan(0);
10          RuleFor(x => x.EndDate).GreaterThanOrEqualTo(x => x.
                ↪ StartDate);
11          RuleFor(x => x.Timing).Must(x => Tools.
                ↪ IsTimingDefinedByValue(x))
12                  .WithMessage(x => $"{x.Timing} is not a valid
                        ↪ timing, choose from: {Tools.
                        ↪ GetValuesOfTiming()}");
13      }
14  }
```

Na 2 można zobaczyć wykorzystanie metody *RuleFor* zdefiniowanej przez klasę bazową *AbstractValidator* z paczki *FluentValidations*. Metoda ta, po wskazaniu wybranego pola modelu *DTO* umożliwia konstruowanie warunków, jakie będzie musiało spełniać to pole to pole. Najprostszy z nich, wykorzystywany w liniach 5-8 nie pozwala na wysłanie *CreateMedicatinDto* z pustymi lub ustawionymi na *null* polami *Name*, *StartDate*, *EndDate* i *Timing*. W liniach 11-12 stworzono bardziej zaawansowane wymaganie, które wymusza na polu *Timing*, aby tekst znajdujący się w nim był jedną z wartości definiowanych przez typ wyliczeniowy o nazwie *Timing*, co jest sprawdzane wewnątrz metody *IsTimingDefinedByValue*. *FluentValidations* dla prostych walidacji takich np.: *NotEmpty*, *GreaterThan*, *GreaterThanOrEqual* posiada domyślne wiadomości zwracane w odpowiedzi na zapytanie, które w jasny sposób informują użytkownika API o nieprawidłowościach znalezionych w modelu.

Paczka ta pozwala także na modyfikowanie domyślnych wiadomości dla błędów użycia metody *WithMessage*. Metoda ta została użyta w linii 12, gdzie jako dodatkowa informacja dla użytkownika zwracana jest lista możliwości, jakie może mieć pole *Timing*.

Kolejną pomocną paczką NuGet, której użyliśmy w projekcie był *AutoMapper*. Jest to narzędzie służące do mapowania podobnych do siebie struktur danych. W przypadku naszego projektu mapowanymi modelami były modele *DTO* oraz modele bazodanowe z warstwy niższej. *AutoMapper* umożliwia szybkie mapowanie pól i kolekcji klas dzięki użyciu refleksji. Pola o tych samych nazwach i typach mapowane są automatycznie, a dla różnic w modelach jest możliwość zaawansowanego konfigurowania mapowań poszczególnych pól - np. zmiana typu pola, czy mapowanie zagnieżdżonych struktur do bardziej "płaskich". Przykład konfiguracji mapowań został przedstawiony na Listingu 3.

Listing 3: Konfiguracja mapowań *AutoMappera*

```
1  using AutoMapper;
2  using MedicationPrescriber.Api.Dtos;
3  using MedicationPrescriber.Domain.Models;
4
5  namespace MedicationPrescriber.Api.Mapper
6  {
7      public class MappingProfile : Profile
8      {
9          public MappingProfile()
10         {
11             CreateMap<DoctorDto, Doctor>();
12             CreateMap<CreateDoctorDto, Doctor>();
13             CreateMap<Doctor, DoctorDto>()
14                 .ForMember(src => src.FirstName, dst => dst.MapFrom(
                       ↪ x => x.User.FirstName))
15                 .ForMember(src => src.LastName, dst => dst.MapFrom(x
                       ↪  => x.User.LastName));
16
17             CreateMap<PatientDto, Patient>();
```

Mapowania dla poszczególnych par modeli dodaje się wewnątrz klasy dziedziczącej po klasie bazowej *Profile* dostępnej w paczce NuGet. Następnie, przy użyciu metody *CreateMap<src, dest>* definiuje się poszczególne typy mapowań. Dla pól, które nie są wspólne stosuje się wywołanie *ForMember*, które przyjmuje dwa wyrażenia lambda wskazujące jakie pole modelu docelowego konfigurujemy i drugie, mówiące jak otrzymać informacje do uzupełnienia pola.

8.1.2 Warstwa dostępu do danych

W warstwie dostępu do danych, do komunikacji z bazą danych służy wspomniany już w Rozdziale 5.2 *Entity Framework Core*. W warstwie tej znajdują się modele obiektowe odwzorowujące tabele w bazie danych oraz klasa *MedicationPresriberDbContext*, która funkcjonuje jako główny łącznik pomiędzy bazą danych, a projektem. Klasa ta zawiera w sobie pola będące odniesieniami do każdej z tabel, a w nadpisanej metodzie *OnModelCre-*

ating zostały zastosowane poszczególne konfiguracje encji bazodanowych. Klasy konfiguracji tych encji zostały zgromadzone w folderze *Configration*, a fragment jednej z konfiguracji przedstawiono na Listingu 4.

Listing 4: Konfiguracja tabeli Patient

```
1   public class PatientConfiguration : IEntityTypeConfiguration<
        ↪ Patient>
2   {
3       public void Configure(EntityTypeBuilder<Patient> builder)
4       {
5
6           builder.Property(x => x.Birthdate)
7                  .IsRequired();
8           builder.HasOne(x => x.User);
9           builder.HasMany(x => x.Medications).WithOne(x => x.
                ↪ Patient).OnDelete(DeleteBehavior.Cascade);
10          builder.HasKey(x => x.PersonalId);
11          builder.Property(o => o.PersonalId).ValueGeneratedNever
                ↪ ();
12      }
13  }
```

Dzięki konfiguracji przedstawionej na Listingu 4 w linii 9 ustalono, że usunięcie rekordu z tabeli pacjentów spowoduje usunięcie wszystkich przypisanych mu leków z tabeli *Medications*. Innym ciekawym elementem tej klasy może być wyłączenie automatycznego generowania wartości klucza podstawowego tabeli w linii 11. Modyfikacja ta była potrzebna, aby mieć możliwość używania numeru PESEL w roli klucza głównego.

8.2 Aplikacja mobilna i webowa

Technologie *Dart* oraz *Flutter* pozwoliły nam na stworzenie aplikacji mobilnej oraz webowej w jednym, wspólnym projekcie. Dzięki temu wszystkie stworzone przez nas serwisy mogły być wykorzystywane na obu platformach, bez konieczności ich ponownej implementacji.

8.2.1 Wykorzystywane modele

Podstawowymi strukturami danych, które wykorzystaliśmy przy implementacji aplikacji były klasy *Doctor*, *Patient* oraz *Medication*. Przy ich tworzeniu dużą rolę odegrał pakiet *json_annotation*, który wykorzystaliśmy do wygenerowania funkcji mapujących obiekt struktury danych do formatu *JSON*, oraz tworzących obiekt z wiadomości otrzymanej od API we wspomnianym już formacie. Są to odpowiednio - konstruktor wykorzystujący wzorzec projektowy fabryki *fromJson* oraz funkcja *toJson*.

Listing 5: *Doctor* - przykładowa klasa odwzorowująca tabelę w bazie danych

```
1 @JsonSerializable()
```

```
class Doctor {
  int id;
  String specialization;
  String firstName;
  String lastName;

  Doctor({this.id, this.firstName, this.lastName, this.
     ↪ specialization});

  factory Doctor.fromJson(Map<String, dynamic> json) =>
     ↪ _$DoctorFromJson(json);

  Map<String, dynamic> toJson() => _$DoctorToJson(this);
}
```

8.2.2 KOMUNIKACJA Z API

W celu pokrycia wszystkich udostępnionych przez API endpoint'ów stworzyliśmy trzy interfejsy - *IDoctorDataAccess*, *IPatientDataAccess* oraz *IMedicationDataAccess* - po jednym dla każdego z wykorzystywanych modeli. Dzięki temu łatwiej było nam pamiętać o stworzeniu funkcji dla wszystkich udostępnionych endpoint'ów, ponieważ każda funkcja zadeklarowana w interfejsie musi posiadać swoją definicję w klasie go implementującej.

Listing 6: *IDoctorDataAccess* - przykładowy interfejs klasy służącej do komunikacji z API

```
abstract class IDoctorDataAccess {
  Future<List<Doctor>> getDoctors();

  Future<Doctor> getDoctorById(int doctorId);

  Future<void> deleteDoctorById(int doctorId);

  Future<Doctor> createDoctor(Doctor doctor);
}
```

Komunikacja z API odbywała się za pomocą protokołu *HTTP*. Umożliwił nam to pakiet o tej samej nazwie (*http*). Udostępnia on podstawowe metody HTTP takie jak: GET, PUT, POST oraz DELETE, które pozwoliły nam odpowiednio pobierać, edytować, dodawać i usuwać rekordy z bazy danych za pomocą API. Klasy *DoctorDataAccess*, *PatientDataAccess*, a także *MedicationDataAccess* implementują metody zadeklarowane w odpowiadających im interfejsach. Listing 7 przedstawia funkcję wykonującą zapytanie poprzez protokół *HTTP*, a następnie tworzącą obiekt doktora za pomocą, opisanego w sekcji 8.2.1, konstruktora *Doctor.toJson*.

Listing 7: *DoctorDataAccess* - przykładowa klasa służąca do komunikacji z API

```
class DoctorDataAccess implements IDoctorDataAccess {
  static final String doctorsUrl = apiUrl + doctorsEndpoint;

  @override
```

```
5   Future<Doctor> getDoctorById(int doctorId) async {
6     String urlToGet = doctorsUrl + '/${doctorId.toString()}';
7
8     http.Response response = await http.get(
9       urlToGet,
10      headers: {'accept': 'application/json'},
11    );
12
13    if (response.statusCode != 200) {
14      throw HttpException(response.statusCode);
15    }
16
17    Doctor doctor = Doctor.fromJson(
18      json.decode(response.body),
19    );
20
21    return doctor;
22  }
```

8.2.3 Obsługa wyjątków

W celu obsługiwania wyjątków stworzyliśmy własną klasę *HttpException*, która na podstawie kodu statusu odpowiedzi na żądanie wysyłane za pomocą protokołu *HTTP* generuje odpowiedni komunikat błędu. Wszystkie obiekty stanowiące wyjątki przekazywane są następnie do obiektu *ErrorHandlingSnackbar*, który wyświetla odpowiednie komunikaty błędów na ekranie. Przykładowe wykorzystanie tego wyjątku w kodzie przedstawione zostało na Listingu 7, natomiast funkcjonalność *ErrorHandlingSnackbar* widoczna jest na Rysunku 7.3. Listing 8 przedstawia fragment implementacji opisywanej klasy.

Listing 8: *HttpException* - klasa generująca wyjątki

```
1 class HttpException implements Exception {
2   final int statusCode;
3   String message;
4
5   HttpException(this.statusCode) {
6     switch (statusCode) {
7       case 400:
8         message = 'Invalid input!';
9         break;
```

Dodatkowo stworzyliśmy funkcje odpowiadające za sprawdzanie wartości wprowadzanych przez użytkownika po to, aby nie wysyłać nieprawidłowo sformułowanych zapytań do API. Funkcje te zostały przygotowane dla każdego pola z wykorzystywanych przez nas modeli *DTO* zwracanych przez API przedstawionych w sekcji 8.1.1.
Listing 9 przedstawia przykładową funkcję sprawdzającą poprawność wprowadzanej przez lekarza wartości dziennego dawkowania leku.

Listing 9: *validateDosage* - funkcja sprawdzająca poprawność wprowadzanej wartości dawkowania leku

```
String validateDosage(String value) {
  if (value.isEmpty) {
    return 'Please enter numeric value.';
  }

  if (value.contains(RegExp(r'^[1-5]$'))) {
    return null;
  } else {
    return 'Invalid input! The value has to be between 1 and 5.';
  }
}
```

8.2.4 Inne zastosowane rozwiązania

- *DIContainer* - klasa wykorzystująca pakiet *GetIt* odpowiadająca za wygenerowanie singleton'ów dla serwisów wykorzystywanych w aplikacji. Jako, że tylko po jednej instancji obiektów odpowiadających za każdy z użytych tam serwisów jest potrzebne dla całego programu, zdecydowaliśmy się na zastosowanie funkcjonalności *lazySingleton*, która tworzy singleton dopiero przy pierwszym wywołaniu instancji danego serwisu. Kod klasy został przedstawiony na Listingu 10, a użycie funkcji *registerServices* na Listingu 11.

Listing 10: *DIContainer* - klasa rejestrująca używane serwisy jako singletony

```
class DIContainer {
  static final GetIt getIt = GetIt.instance;

  static void registerServices() {
    getIt.registerLazySingleton(() => FlutterSecureStorage());
    getIt.registerLazySingleton(() => PatientDataAccess());
    getIt.registerLazySingleton(() => MedicationDataAccess());
    getIt.registerLazySingleton(() => DoctorDataAccess());
  }
}
```

- *FlutterSecureStorage* - pakiet wykorzystujący szyfrowanie AES dla systemu Android oraz Keychain dia iOS, który wykorzystaliśmy do zapisywania danych zalogowanego użytkownika, aby nie musiał on logować się za każdym razem po ponownym uruchomieniu aplikacji. Fragment kodu prezentujący wykorzystanie tego pakietu jest widoczny na listingu 11.

Listing 11: Wykorzystanie *DIContainer* oraz *FlutterSecureStorage*

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  DIContainer.registerServices();

  Patient patient;
  String patientId;

```

```
8   if (!kIsWeb) {
9     var secureStorage = DIContainer.getIt.get<
          ↪ FlutterSecureStorage>();
10    patientId = await secureStorage.read(key: 'personalId');
11  }
12
13  if (patientId != null) {
14    var patientDataAccess = DIContainer.getIt.get<
          ↪ PatientDataAccess>();
15    patient = await patientDataAccess.getPatientById(int.parse(
          ↪ patientId));
16  }
17
18  runApp(
19    MedicationPrescriber(
20      patient: patient,
21    ),
22  );
23 }
```

9 Podsumowanie

9.1 Zrealizowane założenia

Wszystkie założenia odnośnie aplikacji mobilnej oraz webowej zostały zrealizowane w wyznaczonym do tego czasie. Wymagania funkcjonalne przedstawione w sekcji 2 również zostały zaimplementowane w całości. Kluczową rolę w tak efektywnej realizacji projektu pełniło opisane w sekcji 4 podejście wykorzystujące założenia metodyki zwinnej *Scrum*, które przyczyniło się do bardzo dobrej organizacji pracy zespołu.

9.2 Możliwości rozwoju projektu

Podczas prac nad projektem pojawiło się bardzo wiele pomysłów na dalszy rozwój aplikacji. Te najciekawsze z nich to:

- powiadomienia push dla pacjenta przypominające o konieczności wzięcia leku,
- możliwość zaznaczenia leku jako już wzięty przez pacjenta,
- wyświetlanie historii przepisanych pacjentowi leków dla doktora,
- wyświetlanie harmonogramu przyjmowania leków na najbliższe dni dla pacjenta,
- możliwość dodania słownego komentarza do przepisanego leku przez doktora,
- możliwość wysłania zapytania do doktora przez pacjenta.

Każdy z wymienionych pomysłów sprawiłby, że aplikacja byłaby jeszcze większą pomocą dla pacjenta i jest zgodny z uzasadnieniem rynkowym realizacji projektu przedstawionym w sekcji 1.6.

SPIS RYSUNKÓW

SPIS LISTINGÓW

SPIS TABLIC