Spis treści

[WSTĘP 2](#_Toc202712613)

[1. ANALIZA WYMAGAŃ 3](#_Toc202712614)

[1.1. Wymagania biznesowe 3](#_Toc202712615)

[1.2. Diagram modeli domenowych 5](#_Toc202712616)

[1.3. Diagram przypadków użycia 7](#_Toc202712617)

[1.4. Diagram klas 9](#_Toc202712618)

[2. ARCHITEKTURA SYSTEMU 11](#_Toc202712619)

[2.1. Zarys architektury 11](#_Toc202712620)

[2.2. Bezpieczeństwo systemu 13](#_Toc202712621)

[2.2.1. Uwierzytelnienie przy użyciu Microsoft Entra ID 13](#_Toc202712622)

[2.2.2. Szyfrowane protokoły komunikacyjne przy użyciu TLS 16](#_Toc202712623)

[2.3. Frontend 18](#_Toc202712624)

[2.3.1. HTML 18](#_Toc202712625)

[2.3.2. CSS 19](#_Toc202712626)

[2.3.3. TypeScript 20](#_Toc202712627)

[2.3.4. Angular 20](#_Toc202712628)

[2.4. Backend 22](#_Toc202712629)

[2.4.1. REST API 22](#_Toc202712630)

[2.4.2. Java SE 23](#_Toc202712631)

[2.4.3. Spring 23](#_Toc202712632)

[2.4.4. Gradle 24](#_Toc202712633)

[2.4.5. Liquibase 24](#_Toc202712634)

[2.4.6. Lombok 24](#_Toc202712635)

[2.4.7. JUnit 4 24](#_Toc202712636)

[2.5. Baza danych 25](#_Toc202712637)

[3. IMPLEMENTACJA SYSTEMU 25](#_Toc202712638)

[3.1. Implementacja frontendu 25](#_Toc202712639)

[3.1.1. Konfiguracja 25](#_Toc202712640)

[3.1.2. Podstrona Wizyty z pacjentami 31](#_Toc202712641)

[3.1.3. Podstrona Dostępność 33](#_Toc202712642)

[3.1.4. Podstrona Lekarze 37](#_Toc202712643)

[3.1.5. Podstrona Pacjenci 44](#_Toc202712644)

[3.1.6. Podstrona Specjalizacje 46](#_Toc202712645)

[3.1.7. Podstrona Usługi 48](#_Toc202712646)

[3.1.8. Podstrona Wizyty 52](#_Toc202712647)

[3.1.9. Klient API Backendu 56](#_Toc202712648)

[3.2. Implementacja backendu 58](#_Toc202712649)

[3.2.1. Konfiguracja 58](#_Toc202712650)

[3.2.2. Omówienie REST API - /api/doctors 62](#_Toc202712651)

[3.2.3. Omówienie REST API - /api/doctors/{doctorUuid}/availabilities 63](#_Toc202712652)

[3.2.4. Omówienie REST API - /api/patients 63](#_Toc202712653)

[3.2.5. Omówienie REST API - /api/services 63](#_Toc202712654)

[3.2.6. Omówienie REST API - /api/specialties 63](#_Toc202712655)

[3.2.7. Omówienie REST API - /api/users 63](#_Toc202712656)

[4. WDROŻENIE NA CHMURĘ OBLICZENIOWĄ AZURE 64](#_Toc202712657)

[4.1. Microsoft Entra ID 64](#_Toc202712658)

# WSTĘP

# ANALIZA WYMAGAŃ

## Wymagania biznesowe

Budowę każdego systemu komputerowego, należy rozpocząć od ustalenia problemu, który ma on rozwiązać. Następnie należy dostrzec możliwe rozwiązania i wybrać jedno z nich. Powszechnie przyjętym sposobem na opis dlaczego organizacja implementuje system są tzw. wymagania biznesowe skupiające się na celach biznesowych[[1]](#footnote-1). Poniżej przedstawiona jest analiza wymagań biznesowych klienta.

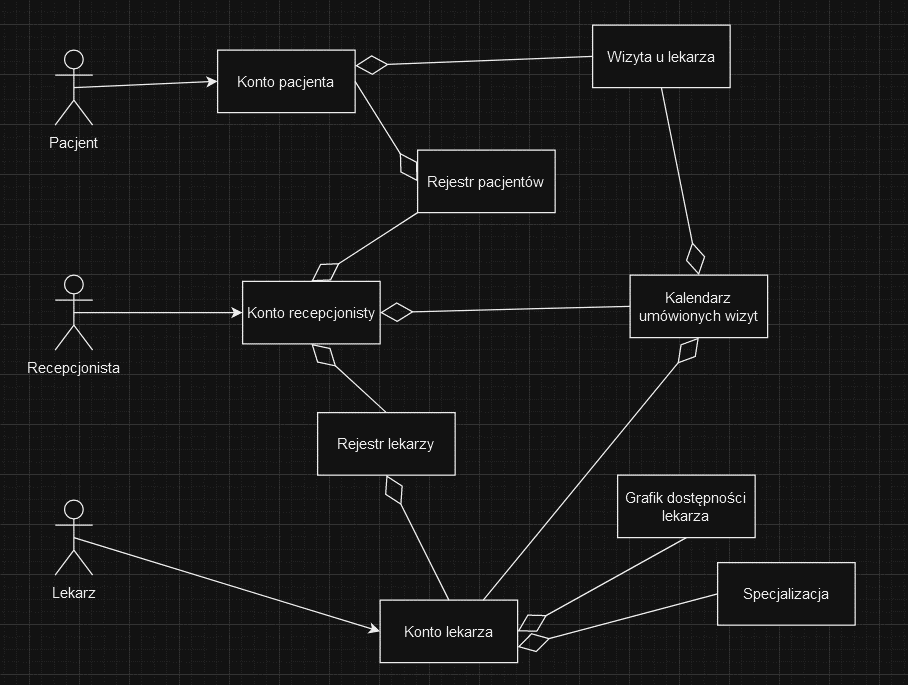
Przychodnia o nazwie *Great Health* potrzebuje systemu, który odciąży jej pracowników w rejestracji pacjentów, tym samym zwiększając wydajność, zmniejszając zapotrzebowanie na rekrutację nowej kadry oraz poprawiając konkurencyjność. Obecnie każdy pacjent musi się najpierw zarejestrować w przychodni poprzez stawienie się fizycznie w przychodni oraz przedstawienie swojego dowodu osobistego. Wówczas recepcjonista prowadząc rejestr pacjentów w formie papierowej, wpisuje nowo zarejestrowaną osobę na liście. Ponadto, pracownicy muszą prowadzić rejestr lekarzy. Podobnie jak z pacjentami, każdy lekarz musi być wpisany w papierowym rejestrze, a w przypadku zwolnienia, z niego wypisany. Oprócz wyżej wspomnianych rejestrów, prowadzony jest grafik dostępności lekarzy, ponieważ większość z nich pełni swoje obowiązki również w innych placówkach. Kiedy recepcjonista placówki ma już przed sobą rejestr pacjentów, lekarzy oraz grafik dostępności specjalistów, może on przyjmować osobiście stawionych pacjentów proszących o umówienie na wizytę z konkretnym lekarzem, o konkretnej godzinie. Informacje te są przechowywane w formie kalendarza. W przypadku jakichkolwiek zmian terminów, pracownik na recepcji musi wykreślić daną osobę z kalendarza i znaleźć jej nowy termin. Cały ten proces, jest żmudny i łatwo o popełnienie w nim błędu, co może skutkować pewnym zamieszaniem oraz opóźnieniami. Zaś z perspektywy samych lekarzy, również nie jest to najlepszy system zarządzania, ponieważ są oni informowani o każdej zmianie z pewnym opóźnieniem. W idealnej sytuacji, lekarze wiedzą kiedy mają pacjenta, a kiedy nie, najszybciej jak tylko się da, oraz nie muszą być o tym informowani przez pracowników z recepcji. Recepcjoniści powinni tylko i wyłącznie weryfikować tożsamość oraz obecność pacjentów gotowych na wizytę, a następnie przekierowywać ich do konkretnego gabinetu. Zaś z perspektywy pacjentów, najlepiej byłoby nie wychodzić z domu, aby umówić się do lekarza oraz mieć jasny obraz wolnych terminów na wybrany dzień.

## Diagram modeli domenowych

Na podstawie wyżej opisanych wymagań biznesowych przychodni, należy przełożyć je na wymagania użytkownika[[2]](#footnote-2). Najlepiej zrobić to w postaci diagramów przypadków użycia. Jednakże najpierw pomocnym jest zacząć od zdefiniowania modeli domenowych, które tworzyć będą system[[3]](#footnote-3). Tworząc taki diagram, warto trzymać się pewnych zasad. Są to[[4]](#footnote-4):

* Koncentracja na obiektach występujących w domenie biznesowej
* Używanie agregacji w celu odzwierciedlenia relacji między obiektami
* Zorganizowanie obiektów wokół kluczowych abstrakcji w domenie biznesowej
* Wykorzystywanie modeli domenowych w celu utworzenia słownika projektu
* Utworzenie diagramu modeli domenowych przed diagramem klas tak, aby w tym drugim uniknąć wieloznaczności używanych terminów

Na podstawie powyższych rekomendacji utworzony został diagram modeli domenowych, który znaleźć można na rysunku 1.1. Wyszczególnionych zostało trzech aktorów – pacjent, recepcjonista oraz lekarz. Dla każdego z nich, utworzony został obiekt będącym kontem w systemie. Konto pacjenta zawiera listę wizyt u lekarzy oraz jest częścią rejestru pacjentów. Natomiast konto recepcjonisty ma dostęp do rejestru pacjentów oraz rejestru lekarzy. Pracownik na recepcji będzie również zarządzać kalendarzem umówionych wizyt, dlatego też jego konto powiązane jest z kalendarzem umówionych wizyt. Ostatnim kontem, jest konto lekarza, do którego przypisany jest grafik dostępności, zaś samo konto jest częścią rejestru lekarzy oraz kalendarza umówionych wizyt.



Rys. 1.1 Diagram modeli domenowych (opracowanie własne)

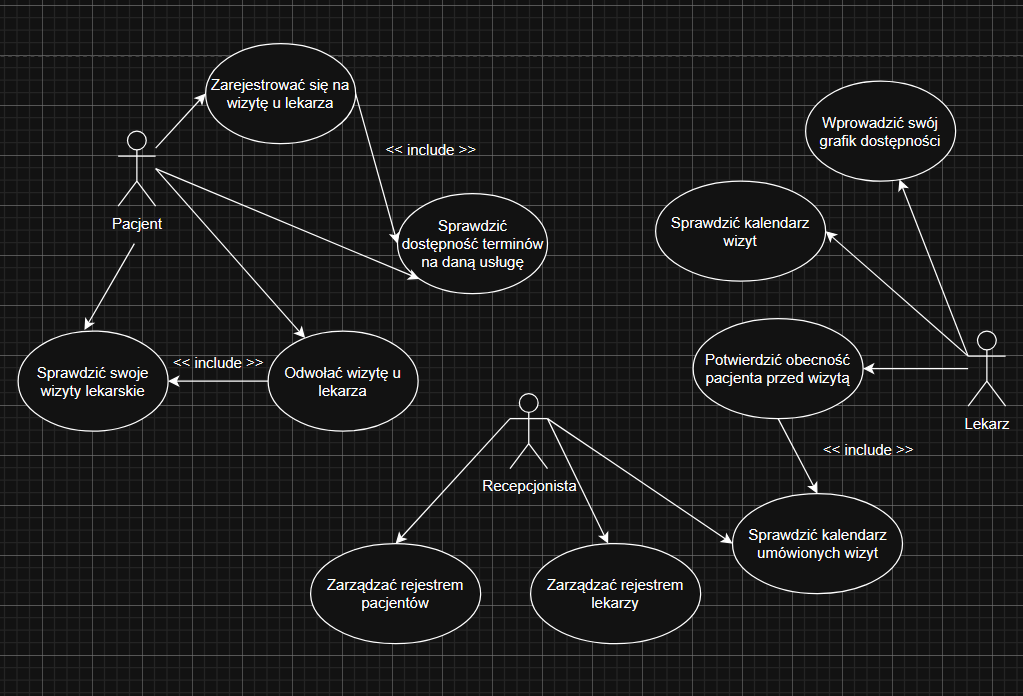
## Diagram przypadków użycia

Kiedy utworzony już został diagram modeli domenowych, można rozpocząć tworzenie diagramu przypadków użycia. Ten rodzaj diagramu pozwala uchwycić wymagania zachowania systemu tak, aby możliwym było go zaprojektowanie[[5]](#footnote-5). Ponadto, „k*ażdy przypadek użycia powinien zostać udokumentowany co najmniej trzema elementami:*

1. *opisem stanu początkowego (warunków początkowych),*
2. *scenariuszem,*
3. *opisem oczekiwanego stanu końcowego”[[6]](#footnote-6)*

Diagram przypadków użycia składa się z graficznego odwzorowania aktorów systemu oraz samych przypadków użycia, w których to aktorzy biorą udział[[7]](#footnote-7). Ponadto, diagram może zawierać również dwa dodatkowe typy relacji: *extend* oraz *include[[8]](#footnote-8)*. Pierwszy rozszerza przypadek użycia o scenariusz alternatywny. Drugi typ relacji zaś oznacza, że jeden z przypadków użycia składa się z jednego lub kilku innych scenariuszy

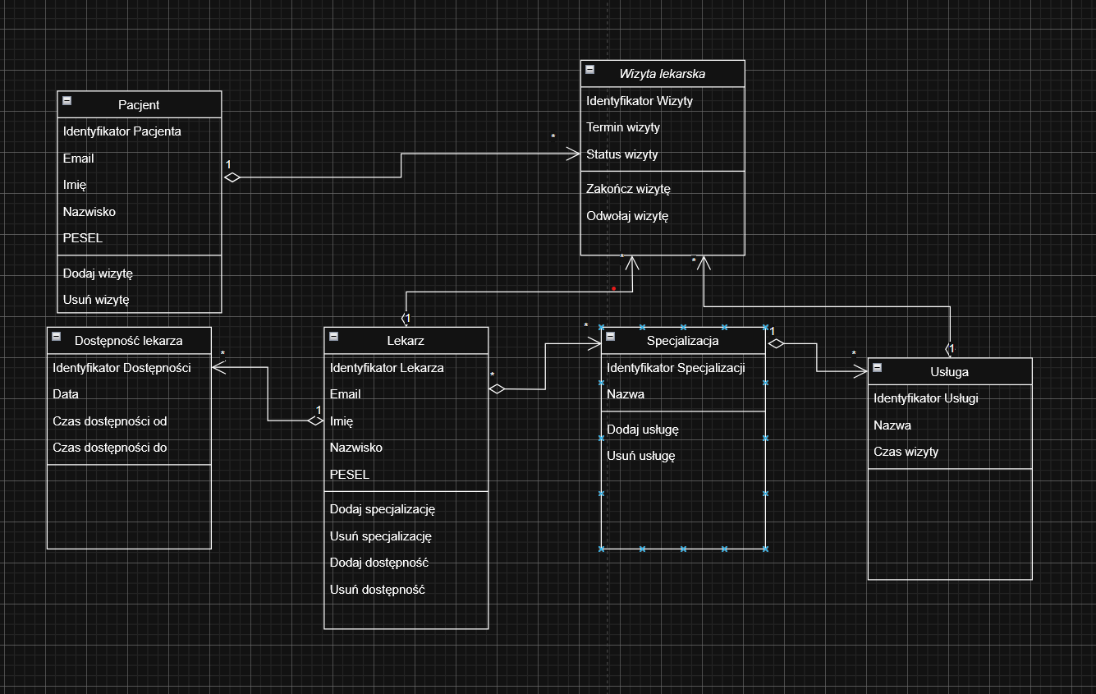
Na podstawie opisanych zasad, utworzonego diagramu modeli domenowych  
z rysunku 1.1, oraz opisu wymagań biznesowych, wykonany został diagram przypadków użycia przedstawiony na rysunku 1.2. Zawiera on trzech aktorów. Pierwszym z nich jest pacjent. Ma on możliwość umówienia się na wizyty u lekarza. Częścią tego przypadku użycia jest również sprawdzenie dostępności lekarzy. Pacjent ma również możliwość odwołać umówioną wcześniej wizytę u lekarza. W celu odwołania wizyty, użytkownik musi najpierw znaleźć swoją wizytę w systemie poprzez sprawdzenie swoich wizyt lekarskich. Kolejnym aktorem jest lekarz. Ma on możliwość wprowadzić swój grafik dostępności, sprawdzić kalendarz wizyt z pacjentami oraz zakończyć wizytę w systemie. Ostatni zaś aktor to recepcjonista. Jego zadaniem jest zarządzanie rejestrem lekarzy oraz rejestrem pacjentów. Ponadto, recepcjonista ma możliwość sprawdzenia kalendarza umówionych wizyt.



Rys. 1.2 Diagram przypadków użycia (opracowanie własne)

## Diagram klas

Zwieńczeniem analizy wymagań jest diagram klas przedstawiony na rysunku nr 1.3. „*Diagram klas to graficzny sposób odwzorowania klas zidentyfikowanych podczas zorientowanej obiektowo analizy oraz zachodzących między nimi relacji*”[[9]](#footnote-9). Analiza ta została wykonana na podstawie opisu wymagań biznesowych, diagramu modeli domenowych (rysunek nr 1.1) oraz diagramu przypadków użycia (rysunek nr 1.2). Wyodrębnionych zostało sześć klas. W skład każdej z nich wchodzi identyfikator, który będzie pomijany w następującym opisie. Będzie to atrybut szczególnie przydatny przy operacjach na bazie danych. Pierwszą klasą jest *Pacjent*. Składa ona się z następujących pól: *Email, Imię* oraz *Nazwisko*. Ponadto na obiektów klasy *Pacjent*, możliwe jest wywołanie metod *Dodaj wizytę* oraz *Usuń wizytę*. Kolejnym typem danych znajdującym się na diagramie jest *Wizyta lekarska*, która jednocześnie jest połączona z klasą *Pacjent* relacją agregacji w taki sposób, aby pacjent mógł posiadać wiele wizyt lekarskich. Atrybutami składającymi się na typ danych o nazwie *Wizyta lekarska* to *Termin wizyty* oraz *Status wizyty*. Ponadto, zadeklarowane zostały metody *Zakończ wizytę* i *Odwołaj wizytę*. Klasa *Wizyta lekarska* zagregowana jest również w typach danych o nazwach *Lekarz* oraz *Usługa*. Pierwszy z nich, zadeklarowane ma następujące pola: *Email, Imię, Nazwisko* i *Płeć*. Poza tym, występują tam takie metody jak - *Dodaj specjalizację*, *Usuń specjalizację, Dodaj dostępność*, czy *Usuń dostępność.* Natomiast *Usługa* jest typem danych złożonym m.in. z pól *Nazwa* oraz *Czas wizyty*. Oprócz klasy *Wizyta lekarska*, łączącej typy *Lekarz* i *Usługa*, na diagramie znajduje się również klasa o nazwie *Specjalizacja*. W jej skład wchodzi atrybut *Nazwa*. Klasa ta również posiada listę usług, stąd na diagramie zdefiniowana została relacja agregacji z typem *Usługa* oraz dwie metody – *Dodaj usługę* i *Usuń usługę*. Klasa *Specjalizacja* jest też zagregowana w typie *Lekarz*. Ostatnim typem danych znajdującym się na diagramie jest *Dostępność lekarza*. Podobnie do klasy *Specjalizacja*, jest ona zagregowana w typie *Lekarz*. Ponadto, w jej skład wchodzą następujące atrybuty – *Czas dostępności od* oraz *Czas dostępności do*.

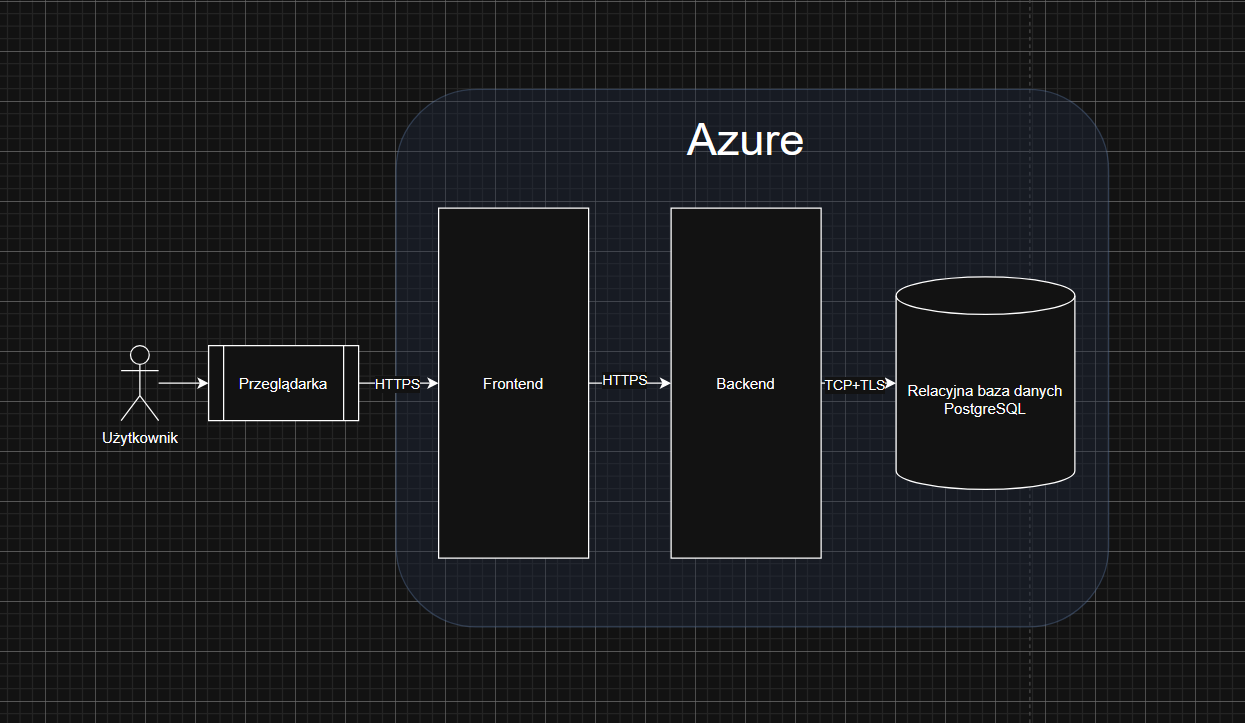


Rys. 1.3 Diagram klas (opracowanie własne)

# ARCHITEKTURA SYSTEMU

## Zarys architektury

System jest zbudowany z wykorzystaniem wzorca *Three-Tier Client Server Architecture[[10]](#footnote-10)*, który polega na podzieleniu całego systemu na trzy komponenty – warstwy prezentacji, warstwy logiki biznesowej oraz warstwy bazy danych. Każda z warstw jest osobną usługą sieciową. Pierwsza z nich, czyli warstwa prezentacji, jest również znana pod nazwą *frontend*, zaś druga pod nazwą *backend[[11]](#footnote-11).* Schemat w jakim pracują te trzy usługi polega na ich komunikacji między sobą - frontendu z backendem używając protokołu komunikacyjnego HTTPS oraz backendu z bazą danych używając protokołu TCP zaszyfrowanego protokołem TLS. Wizualizacja tej architektury trójwarstwowej przedstawiona jest na rysunku nr 2.1.



Rys. 2.1 Diagram architektury systemu (opracowanie własne)

Użytkownik wykorzystuje przeglądarkę do wyświetlenia warstwy prezentacji. Frontend generuje wizualizację danych, oraz umożliwia wykonanie na nich operacji. Kiedy użytkownik naciska któryś z wyświetlonych przycisków, generowane jest żądanie i przesyłane do warstwy logiki biznesowej. Backend wówczas wykonuje manipulacje na danych poprzez wysyłanie zapytań do bazy danych i na koniec zwraca zaktualizowane dane do frontendu, aby ten na koniec je wyświetlił użytkownikowi w przeglądarce. Żądania, które są generowane na frontendzie, wysyłane są z komputera użytkownika. Dlatego też, zarówno frontend, jak i backend są udostępnione w Internecie. Natomiast baza danych nie musi być dostępna z poziomu Internetu. Musi one jedynie znajdować się w tej samej sieci prywatnej, co backend. W ten sposób komunikacja backend – baza danych będzie przebiegać pomyślnie, a nikt spoza tej sieci nie będzie mógł wykonywać manipulacji na danych bezpośrednio się z nią komunikując, nawet jeżeli zna hasło, co jest dodatkowym zabezpieczeniem.

Głównymi zaletami architektury trójwarstwowej jest skalowalność, elastyczność oraz bezpieczeństwo[[12]](#footnote-12). Skalowalność odnosi się do możliwości zwiększania lub zmniejszania liczby instancji usługi osobno na każdym z trzech poziomów tak, aby obsłużyć wszystkich aktualnie korzystających z systemu użytkowników. Elastyczność zaś odzwierciedla fakt, że programiści mogą niezależnie pracować nad frontendem, backendem, a także bazą danych. Częstą praktyką jest również dzielenie zespołów na specjalistów z danej warstwy, to znaczy tzw. frontend developerów oraz backend developerów[[13]](#footnote-13). Natomiast zwiększone bezpieczeństwo wynika wprost z zastosowania zabezpieczeń na każdej warstwie aplikacji. Także wyżej wspomniane wyizolowanie bazy danych i umieszczenie jej w sieci prywatnej jest dobrym przykładem jak można skorzystać z wielowarstwowości architektury.

## Bezpieczeństwo systemu

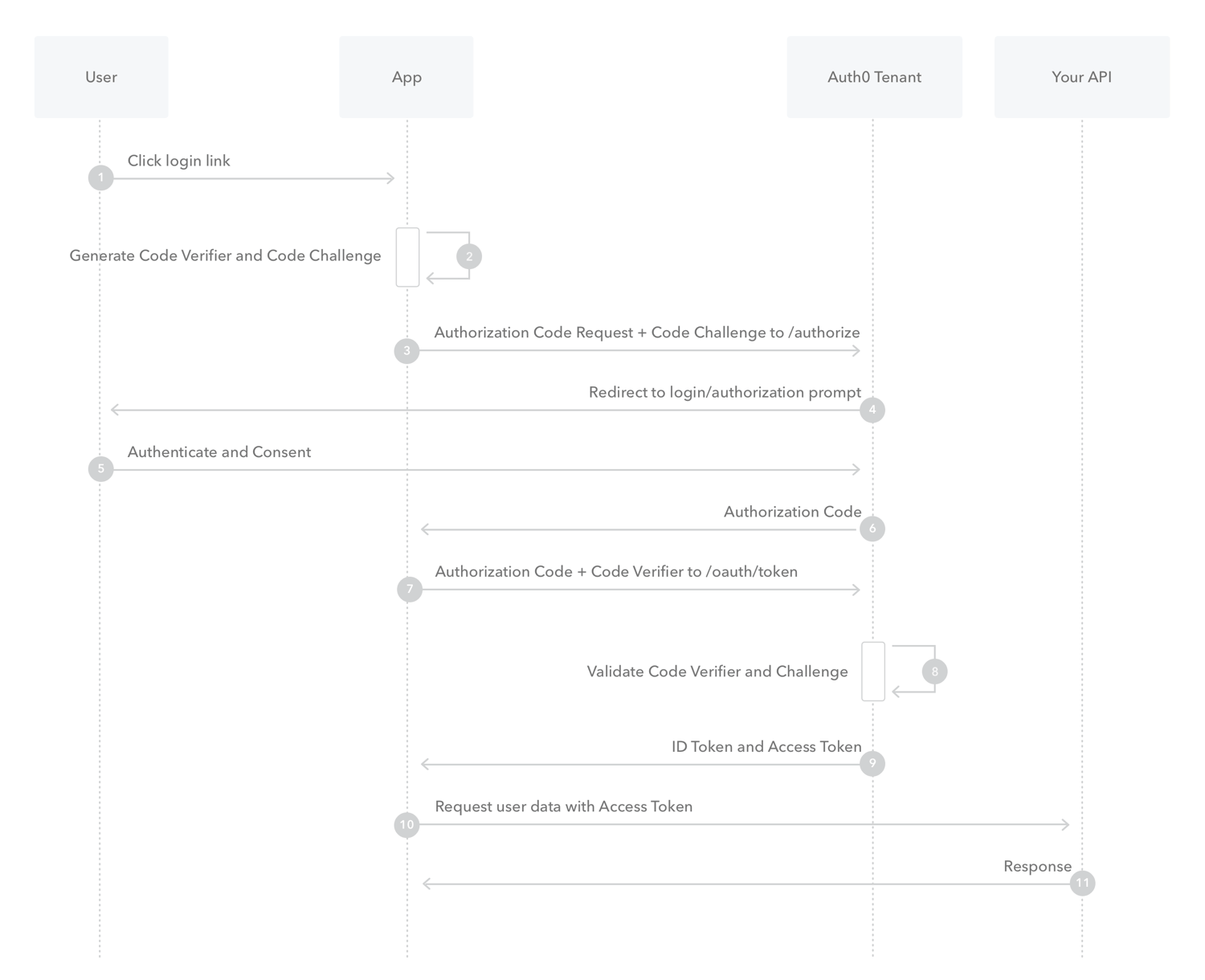
System jest zabezpieczony na wielu poziomach. Przede wszystkim są to:

* Uwierzytelnienie przy użyciu Microsoft Entra ID
* Szyfrowane protokoły komunikacyjne przy użyciu TLS/SSL
* Uwierzytelnienie do bazy danych poprzez podanie nazwy użytkownika bazodanowego oraz hasła

Pierwsze dwa zostaną bardziej szczegółowo opisane w następujących podrozdziałach.

## Uwierzytelnienie przy użyciu Microsoft Entra ID

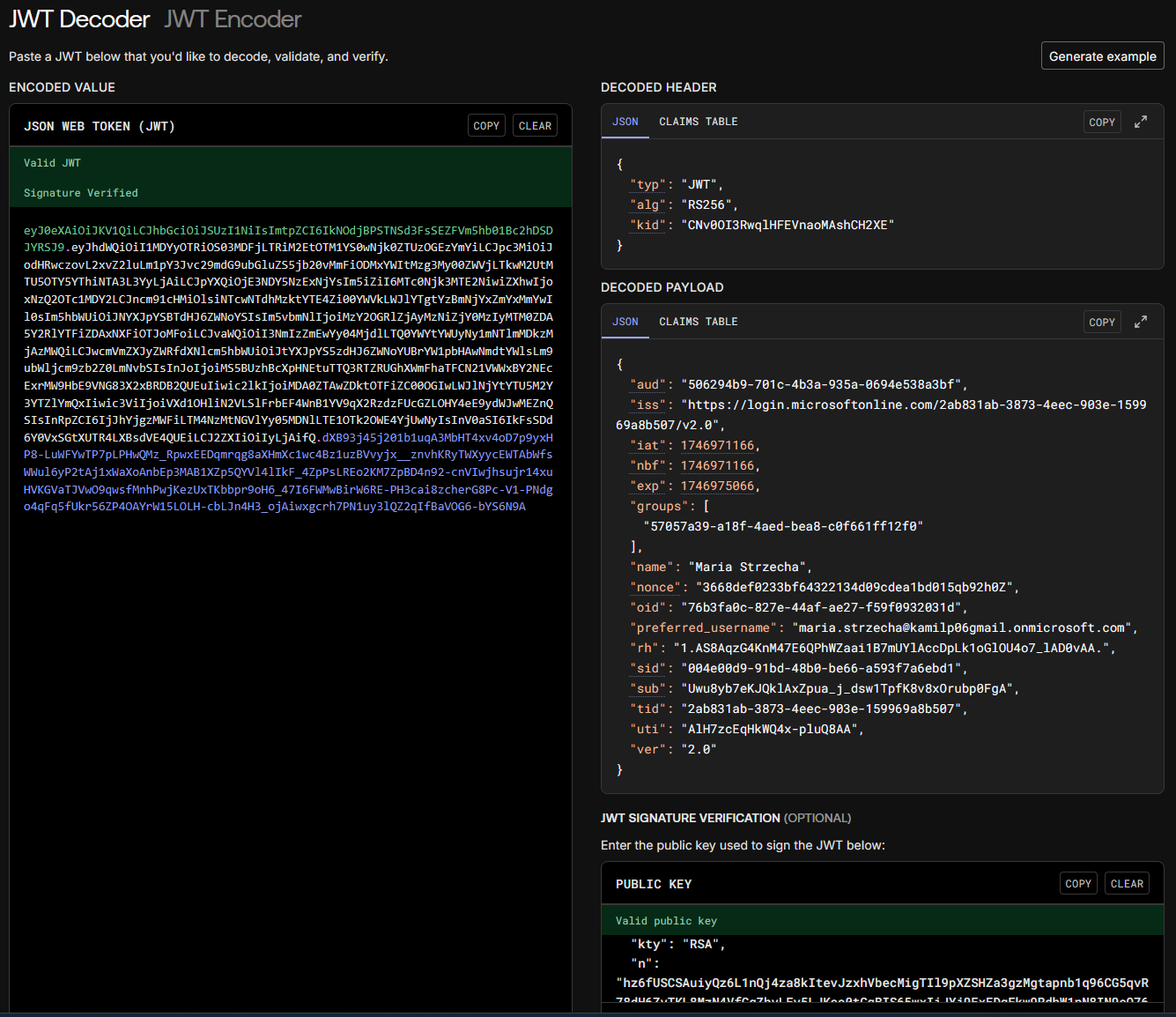
Pierwszym poziomem zabezpieczenia jest frontend. Kiedy użytkownik chce otworzyć jakąkolwiek stronę aplikacji w przeglądarce, zostaje automatycznie przekierowany na stronę logowania dostarczoną przez usługę Microsoft Entra ID z chmury Azure. Każdy, kto chce skorzystać z aplikacji musi mieć uprzednio utworzone konto w Microsoft Entra ID oraz dodany do odpowiedniej grupy. Grupy odzwierciedlają role, które odgrywają użytkownicy w systemie. W aplikacji *GreathHealth* są to pacjent, recepcjonista oraz lekarz. Po wprowadzeniu nazwy użytkownika oraz hasła, przeglądarka z powrotem przekierowuje użytkownika na stronę, którą chciał on odwiedzić. Po tym wydarzeniu, możliwe jest otwieranie wszystkich pozostałych udostępnionych stron poprzez panel nawigacyjny. Cały ten proces uwierzytelniania nosi nazwę *Authorization Code Flow with PKCE[[14]](#footnote-14)*. Schemat oraz kroki, które się na niego składają można zobaczyć na rysunku nr 2.2.



Rys. 2.2 Schemat działania uwierzytelnienia Authornization Code Flow with PKCE (https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce)

Kiedy użytkownik zostanie uwierzytelniony, frontend uzyskuje tzw. access token, który jest ciągiem znaków alfanumerycznych zawierającym dane dot. zalogowanego użytkownika. Dane te zapisane są zgodnie z formatem JSON Web Token (w skrócie JWT)[[15]](#footnote-15), który oprócz tego, że posiada dane dotyczące użytkownika, pozwala również na weryfikację źródła pochodzenia oraz integralności danych zakodowanych w tokenie. Weryfikacja ta możliwa jest dzięki temu, że token zawiera również podpis, który zweryfikować można za pomocą klucza publicznego dostarczonego przez Microsoft Entra ID[[16]](#footnote-16). Na rysunku nr 2.3 przedstawiony jest przykładowy token w formacie JWT wygenerowany z pomocą Microsoft Entra ID. Jak można zauważyć, składa się on z 3 części. Pierwsze dwie to nagłówek oraz dane użytkownika. Są one zakodowane przy użyciu algorytmu Base64[[17]](#footnote-17). Trzecia część to podpis umożliwiający weryfikację tokena.

Frontend uzyskując access token z Microsoft Entra ID, uzyskuje tym samym dostęp do m.in. imienia i nazwiska, adresu email oraz grup, do których należy użytkownik. Imię i nazwisko wyświetlane są na pasku nawigacyjnym w celu poinformowaniu kto został uwierzytelniony. Natomiast adres email wykorzystywany jest do identyfikacji użytkownika w backendzie i przypisywaniu do niego danych przechowywanych w bazie danych. Grupy zaś informują o tym, jakie role zostały przypisane do użytkownika. Na podstawie ról, frontend jest w stanie określić jakie strony i funkcjonalności powinny zostać wyświetlone.



Rys. 2.3 Przykład tokena JWT wygenerowanego przez Microsoft Entra ID oraz zdekodowanego na stronie https://jwt.io

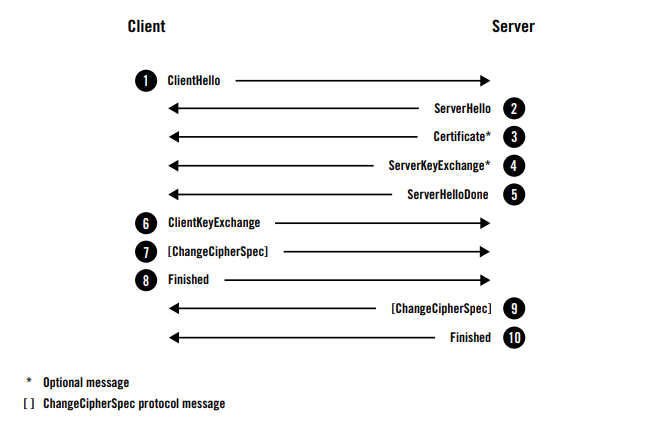
Frontend nie może być jedyną warstwą, która jest zabezpieczona przy użyciu Microsoft Entra ID. Backend, jako że wystawia API na zewnątrz, do Internetu, również musi weryfikować czy klient interfejsu (użytkownik) ma dostęp do wybranych funkcjonalności bądź danych. W tym celu, frontend przy każdym wysłanym żądaniu zawiera wspomniany token. Dzięki temu, backend może zweryfikować pochodzenie tokena. Ponadto, identyfikuje on tożsamość oraz sprawdza role zalogowanego użytkownika.

## Szyfrowane protokoły komunikacyjne przy użyciu TLS

Kolejnym komponentem zabezpieczenia systemu jest zastosowanie protokołu TLS w celu szyfrowania komunikacji między przeglądarką a usługami oraz między samymi usługami w systemie. Element ten jest konieczny w celu ochrony przed atakiem Man In The Middle (MITM). Tego typu atak polega na przechwyceniu komunikacji przez pośredniczący element sieciowy i jej odczycie lub modyfikacji tak, aby żadna ze stron o tym nie wiedziała[[18]](#footnote-18). W ten sposób atakujący może np. dokonać przelew z czyjegoś konta lub pozyskać dane poufne. Aplikacja *GreatHealth* wykorzystuje protokół HTTPS do ruchu sieciowego między przeglądarką a frontendem i backendem, który jest połączeniem protokołów HTTP oraz TLS[[19]](#footnote-19). Natomiast komunikacja pomiędzy backendem a bazą danych PostgreSQL wykorzystuje protokół TCP oraz TLS[[20]](#footnote-20). Pierwszy oraz drugi rodzaj ruchu sieciowego został zilustrowany na rysunku nr 2.1.

Zasada działania protokołu TLS opiera się na bezpiecznej wymianie utworzonego wspólnie przez obie strony klucza prywatnego, a następnie używania go do szyfrowania oraz deszyfrowania komunikatów. Proces tworzenia wspomnianego klucza prywatnego nazywany jest TLS Handshake i jest kluczowym elementem protokołu TLS[[21]](#footnote-21). Możemy wyróżnić następującego kroki.

1. Klient inicjuje TLS Handshake
2. Serwer odpowiada na zainicjowany TLS Handshake wraz z parametrami połączenia
3. Serwer wysyła swój certyfikat TLS tak, aby klient mógł zweryfikować jego poprawność
4. Serwer wysyła dane potrzebne do wygenerowania klucza prywatnego
5. Server oznajmia, że zakończył wysyłanie danych na tym etapie
6. Klient wysyła dane potrzebne do wygenerowania klucza prywatnego, szyfrując je kluczem publicznym serwera
7. Klient zaczyna wysyłać dane zaszyfrowane utworzonym kluczem prywatnym
8. Server zaczyna wysyłać dane zaszyfrowane utworzonym kluczem prywatnym



Rys. 2.4 Protokół TLS Handshake (Ivan Ristić, Bulletproof SSL and TLS, Feisty Duck Limited, s. 27)

Istotnym aspektem SSL Handshake jest to, że komunikacja już na tym etapie jest szyfrowana. Kiedy serwer wysyła swój certyfikat, zawiera on również klucz publiczny, za pomocą którego klient następnie szyfruje swoją kolejną wiadomość wysyłaną do serwera (pkt 6). Wiadomość ta zawiera pewną wartość, która połączona z poprzednią wartością wysłaną przez serwer, daje możliwość wygenerowania klucza prywatnego. Kiedy serwer otrzyma tę wiadomość, odszyfrowuje ją za pomocą swojego klucza prywatnego i jest w stanie wygenerować klucz prywatny po swojej stronie. Wówczas obie strony posiadają klucz prywatny, który następnie używany jest w komunikacji w celu ochrony przed atakiem Man In The Middle[[22]](#footnote-22).

## Frontend

Frontend jest usługą sieciową komunikującą się przy użyciu protokołu HTTPS. Przy użyciu serwera Nginx zwraca ona między innymi pliki JavaScript, CSS oraz HTML, które następnie służą do wyświetlenia aplikacji internetowej przez przeglądarkę. Komunikacja między frontendem a backendem również odbywa się przy pomocy protokołu HTTPS. Warto zauważyć, że to przeglądarka wykonuje kod źródłowy aplikacji frontendowej, a więc tak naprawdę komunikacja frontendu z backendem jest realizowana jako komunikacja przeglądarki z serwerem, ponieważ to w przeglądarce realizowany jest kod frontendu, zaś kod backendu na serwerze. Aplikacja została stworzona przy użyciu frameworka Angular[[23]](#footnote-23), gdzie kod źródłowy jest napisany w języku programowania TypeScript[[24]](#footnote-24), CSS[[25]](#footnote-25) oraz HTML[[26]](#footnote-26).

## HTML

HyperText Markup Language (w skrócie HTML)[[27]](#footnote-27) jest językiem znaczników, który definiuje treść strony internetowej wyświetlanej przez przeglądarkę. Kod źródłowy strony internetowej jest zapisywany w formie tekstowej w pliku o rozszerzeniu *.html*. Następnie po jego otwarciu przez przeglądarkę, zostaje wyświetlona treść strony internetowej. Kod źródłowy napisany w języku HTML ma formę hierarchiczną. Głównym komponentem występującym w tym języku jest znacznik. Najbardziej powszechnymi znacznikami są m.in. *body* (definiujący główną treść strony), *p* (definiujący paragraf), *h1* (definiujący nagłówek), *a* (definiujący odnośnik do innej strony), *div* (definiujący blok treści oddzielony nową linią) oraz *span* (definiujący blok treści nieoddzielony nową linią). Znaczniki mogą być zagnieżdżane, co sprawia, że cała struktura strony internetowej ma postać hierarchiczną. Przykładem kodu HTML oraz tego, jaki jest rezultat wyświetlenia go przez przeglądarkę można zaobserwować na rysunku nr 2.5.

Każdy rodzaj znacznika posiada pewną grupę atrybutów, które można im przypisać. Przykładem może być znacznik typu *img*, który definiuje obraz oraz atrybut *src* wskazujący plik graficzny. Innym przykładkę może być paragraf, czyli znacznik typu *p* wraz z atrybutem *style*, opisujący cechy stylistyczne paragrafu takie jak rozmiar czcionki bądź jej kolor.

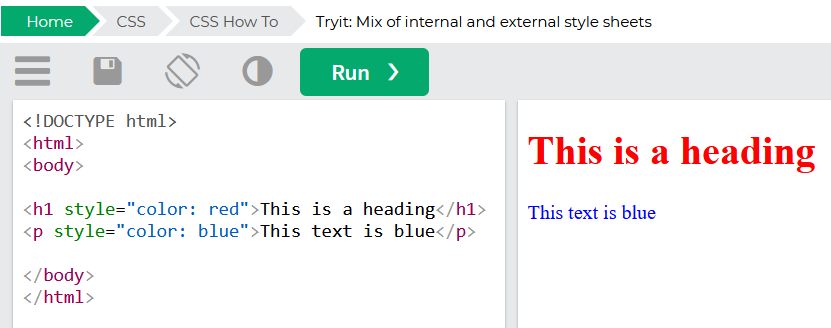


Rys. 2.5 Przykład kodu HTML oraz wyświetlenia go w przeglądarce (<https://udigroup.pl/blog/jezyk-html-co-to-jest-do-czego-sluzy-jak-wyglada/>)

## CSS

Cascading Style Sheets (w skrócie CSS)[[28]](#footnote-28) jest językiem uzupełniającym HTML, który służy do opisu warstwy prezentacji strony internetowej. Za pomocą języka CSS można panować nad rozmieszczeniem poszczególnych elementów na stronie, ich stylem oraz rozmiarem. Kod CSS można zastosować bezpośrednia w plikach *.html* przy użyciu atrybutu *style*, bądź jako oddzielny plik tekstowy o rozszerzeniu *.css* ładowany przez przeglądarkę przy pomocy znacznika *link* w pliku *.html* w znaczniku *head*.

Wyróżnia się 3 główne komponenty języka CSS. Pierwszym z nich jest selektor, który służy do wskazania elementów HTML, które zostaną objęte daną konfiguracją stylu. Wskazywać można rodzaje znaczników, konkretne elementy za pomocą identyfikatora lub na podstawie wartości atrybutów. Jednak najbardziej popularnym i polecanym selektorem jest selektor typu klasa, którą następnie można zastosować wybiórczo do wielu elementów HTML przy użyciu atrybutu *class* w wybranym znaczniku. Kolejnymi dwoma komponentami języka CSS są właściwości oraz wartości. Za ich pomocą definiować można cechy wyglądu elementów HTML. Dla przykładu przy użyciu właściwości *color* oraz wartości *red* można sprawić, że czcionka będzie czerwona. Przykład takie użycia zobrazowany jest na rysunku nr 2.6.



Rys. 2.6 Przykład użycia CSS (opracowanie własne przy użyciu strony <https://www.w3schools.com>)

## TypeScript

TypeScript**[[29]](#footnote-29)** jest językiem programowania, który jest rozszerzeniem języka JavaScript**[[30]](#footnote-30)**. Wprowadza on m.in. statyczne typowanie, które zdecydowanie zmniejsza liczbę błędów w kodzie oraz zapewnia dodatkową dokumentację. TypeScript jest kompilowany do języka JavaScript, który to kod jest wówczas wykonywany przez przeglądarkę. Kompilacja ta jest częścią procesu budowy projektu, co oznacza, że na etapie instalacji aplikacji istnieje już tylko skompilowany kod w języku JavaScript. Oba te języki wprowadzają dynamikę do strony internetowej. Pozwalają na zaprogramowanie akcji dostępnych na stronie oraz integrację z backendem, bądź innymi usługami sieciowymi.

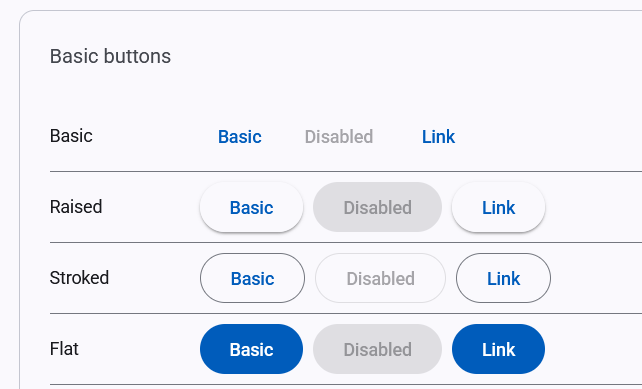
## Angular

Angular[[31]](#footnote-31) jest frameworkiem frontendowym napisanym w języku TypeScript przez firmę Google. Wprowadza on narzędzia do tworzenia aplikacji internetowych m.in. typu SPA (Single Page Application). Aplikacje tego typu w porównaniu do SSR (Server Side Rendering) charakteryzują się ogólnie szybszym działaniem, kosztem wolniejszego załadowania się strony na samym początku. Angular zapewnia projektowi spójną strukturę i wprowadza możliwość tworzenia komponentów, na które składają się plik z kodem HTML, plik z kodem CSS oraz plik z kodem TypeScript. Dzięki takiej strukturze, element ten jest spójnym tworem, gdzie kod HTML przedstawia treść, kod CSS wprowadza styl, natomiast kod TypeScript logikę biznesową. Komponent może definiować podstronę, ale również powtarzalny element, który jest wykorzystywany w wielu miejscach, w projekcie.

Ponadto, Angular wprowadza tzw. framework CSS o nazwie Angular Material[[32]](#footnote-32). Narzędzie to wprowadza grupę predefiniowanych komponentów dzielących ten sam styl, które można wykorzystać do zbudowania aplikacji. Przykładowymi komponentami, które wprowadza Angular Material są ikona, przycisk, panel nawigacyjny, kalendarz, czy też tekstowe pole formularza. Wygląd kilku z nich przedstawiony jest na rysunkach o numerach 2.7 oraz 2.8.



Rys. 2.7 Przykładowe pola formularza z Angular Material (https://material.angular.dev/components/input/overview)



Rys. 2.8 Przykładowe przyciski z Angular Material (https://material.angular.dev/components/button/overview)

## Backend

Backend jest usługą sieciową działającą na serwerze, która wystawia REST API (skrót od Representational State Transfer Application Programming Interface**)**[[33]](#footnote-33), orazkomunikującą się za pomocą protokołu HTTPS. Usługa ta jest napisana w języku programowania Java SE 17 przy użyciu frameworka Spring. Ponadto, użyte zostały narzędzia takie jak Maven, Liquibase, Lombok oraz JUnit 4.

## REST API

REST API jest zbiorem zasad, które powodują, że komunikacja z klientem (tj. frontendem) jest prostsza. Przede wszystkim cechami szczególnymi tego typu interfejsu są bezstanowość, podzielność oraz standaryzacja. Bezstanowość polega na tym, że każde żądanie wysyłane do serwera zawiera wszystkie informacje potrzebne do jego przetworzenia. Serwer nie przechowuje żadnych informacji dot. sesji, tak więc nie zawiera stanu. Druga cecha tego typu interfejsu jest oparta o specyfikę schematu API. Endpointy REST API są nazwane w taki sposób, aby odwzorować zależność między zasobami. Dla przykładu tworzony jest endpoint, za pomocą którego klient może otrzymać listę specjalizacji wybranego doktora. Adres URL takiego endpointu wygląda wówczas następująco: *GET api/doctors/{doctorId}/specialties*. Można zauważyć, że pierwsza część adresu identyfikuje doktora, a następnie dopiero druga część nawiązuje do specjalności, które zostaną zwrócone. Cały adres pokazuje zależność między doktorem a specjalnościami. Specjalności stanowią część modelu doktora. Trzecią cechą REST API jest standaryzacja, która polega na wykorzystaniu metod protokołu HTTPS w określonych przypadkach. Następująca lista opisuje zastosowanie pięciu najbardziej powszechnie używanych metod.

* GET – używana przy idempotentnych endpointach, które tylko i wyłącznie zwracają dane, a więc nie występuje modyfikacja danych
* POST – używana przy nieidempotentnych endpointach tworzących nowe dane
* PUT – używana przy idempotentnych endpointach całościowo aktualizujących istniejące dane
* PATCH – używana przy nieidempotentnych endpointach częściowo aktualizujących istniejące dane
* DELETE – używana przy idempotentnych endpointach usuwających dane

Warto zwrócić uwagę na pojęcie idempotencji. Jeżeli wybrany endpoint jest idempotentny, wówczas użycie go dwa lub więcej razy będzie skutkować tym samym rezultatem[[34]](#footnote-34). Jest to istotna cecha wybranych metod HTTPS, która powinna być brana pod uwagę przy projektowaniu interfejsu REST API.

## Java SE

Java SE jest obiektowym językiem programowania[[35]](#footnote-35) szeroko stosowanym m.in. w implementacji aplikacji serwerowych. Java jest oparta o tzw. maszynę wirtualną JVM, która w czasie działania programu konwertuje kod bajtowy Javy na kod maszynowy[[36]](#footnote-36). Przewagą tego typu rozwiązania jest możliwość uruchomienia tego samego kodu na różnych systemach operacyjnych bez większych problemów, o ile zainstalowane jest środowisko uruchomieniowe Javy. Ponadto, maszyna wirtualna w czasie rzeczywistym optymalizuje wykonanie programu m.in. poprzez proces zwany Garbage Collection[[37]](#footnote-37), który polega na uwolnieniu zasobów pamięci operacyjnej RAM, gdy nie są używane.

## Spring

Spring Framework jest zbiorem narzędzi napisanych w języku Java, które ułatwiają tworzenie aplikacji biznesowych w językach opartych o JVM, w tym Javy[[38]](#footnote-38). Jednymi z najbardziej popularnych modułów Springa jest Spring Framework[[39]](#footnote-39), Spring Web MVC[[40]](#footnote-40), Spring Data[[41]](#footnote-41) oraz Spring Boot[[42]](#footnote-42). Spring Framework wprowadza Dependency Injection[[43]](#footnote-43), które implementuje wzorzec IoC Container. Umożliwia on łatwe testowanie oraz konfigurację aplikacji poprzez zastosowanie tzw. Beanów[[44]](#footnote-44), które są instancjami klas przechowywanymi w kontenerze zależności zarządzanym przez Springa. Spring MVC wprowadza m.in. możliwość wystawienia REST API, w którym komunikacja zachodzi przy użyciu protokołu HTTP, zaś komunikaty wysyłane są w formacie JSON. Kolejny moduł tj. Spring Data wprowadza integrację z bazami danych, w tym również relacyjnymi bazami danych. Ostatni wymieniony moduł, czyli Spring Boot, ułatwia konfigurację stosowanych modułów Spring w projekcie. Wprowadza on tzw. autokonfigurację, która przyjmuje pewne domyślne wartości dla większości parametrów, co znacznie ułatwia zintegrowanie nowych narzędzi. Ponadto, konfiguracja przy użyciu Spring Boota bardzo często jest oparta o parametry zdefiniowane w pliku YAML, co znacznie ułatwia i przyspiesza pracę.

## Gradle

Gradle[[45]](#footnote-45) jest narzędziem wykorzystywanym do zarządzania zależnościami w projektach opartych o języki JVM, do których zalicza się Java. Dzięki wprowadzeniu pliku konfiguracyjnego *build.gradle*, w którym wymienione są wszystkie biblioteki używane w projekcie, dodawanie oraz modyfikacja już istniejących narzędzi jest znacznie prostsza. Poza zarządzaniem zależnościami, Gradle wprowadza również możliwość tworzenia skryptów do budowania projektu tak, aby można było utworzyć paczkę JAR, która może być uruchomiona na innym sprzęcie komputerowym, np. na serwerze.

## Liquibase

Liquibase[[46]](#footnote-46) to biblioteka, która służy do zarządzania zmianami wprowadzanymi do relacyjnej bazy danych. Umożliwia ona tworzenie skryptów zapisanych w formacie XML, które następnie przy uruchomieniu aplikacji są wykonywane tak, aby uzyskać oczekiwany stan bazy danych. Skrypty te mogą być pisane zarówno w sposób abstrahujący od konkretnej bazy danych, jak i w sposób, gdzie używa się funkcji bazodanowych dedykowanych dla wybranej bazy.

## Lombok

Lombok[[47]](#footnote-47) jest biblioteką napisaną w Javie, która wprowadza adnotacje umożliwiające tworzenie kodu bez napisania go w sposób bezpośredni. Przykładem mogą być metody zmieniające stan pól klasy nazywane Setterami[[48]](#footnote-48). Lombok umożliwia definicję ich za pomocą adnotacji @Setter.

## JUnit 4

JUnit4[[49]](#footnote-49) to narzędzie, dzięki któremu można w łatwy sposób pisać testy jednostkowe. Wprowadza ona kilka adnotacji, które umożliwiają oznaczenie metody testującej, metody przygotowującej dane przed testem oraz metody wykonywane po teście np. aby przywrócić stan pierwotny w bazie danych.

## Baza danych

Zastosowaną bazą danych w projekcie *Great Health* jest PostgreSQL[[50]](#footnote-50). Jest to powszechnie wykorzystywana relacyjna baza danych, która wprowadza wiele funkcji ułatwiających manipulowanie danymi. Tak jak w każdej relacyjnej bazie danych, struktura danych oparta jest o zastosowanie tabeli składających się z kolumn o zdefiniowanym typie oraz wierszy reprezentujących rekordy danych. Ponadto, możliwe jest powiązanie między tabelami tak, aby można było tworzyć relacje między różnymi zbiorami danych. PostgreSQL umożliwia operacje na danych przy użyciu języka SQL[[51]](#footnote-51) będącego standardem komunikacji z relacyjnymi bazami danych.

# IMPLEMENTACJA SYSTEMU

System składa się z dwóch aplikacji – frontendu (warstwa prezentacji) oraz backendu (warstwa logiki biznesowej). Omówienie implementacji systemu warto zacząć od pierwszego komponentu tj. frontendu, a następnie płynnie przejść do warstwy backendowej.

## Implementacja frontendu

Frontend napisany jest przy użyciu frameworka Angular, m.in. w języku TypeScript. Najpierw zostanie omówiona konfiguracja aplikacji.

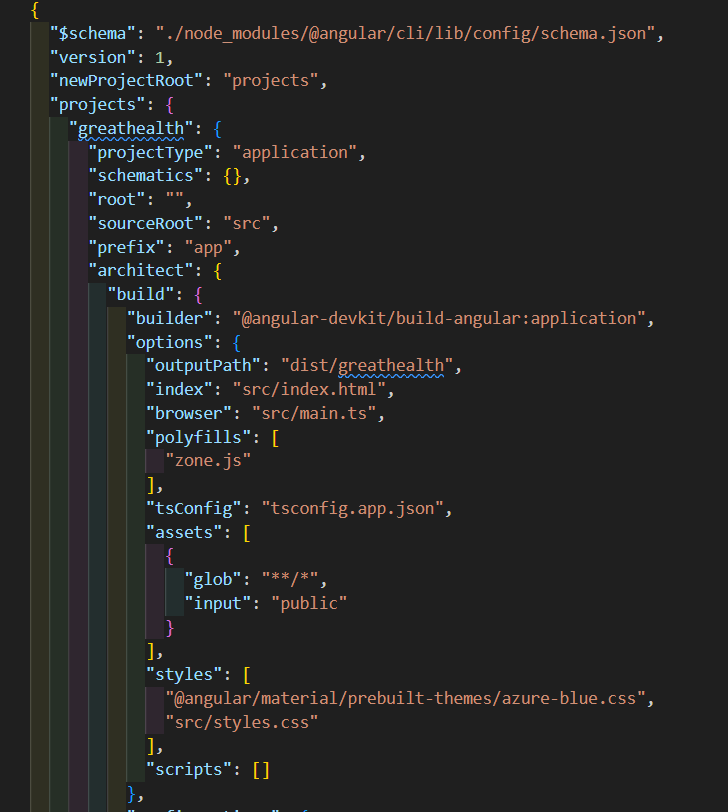
## Konfiguracja

Zarządzanie zależnościami odbywa się w pliku *package.json*, co zostało zaprezentowane na rysunku nr 3.1.



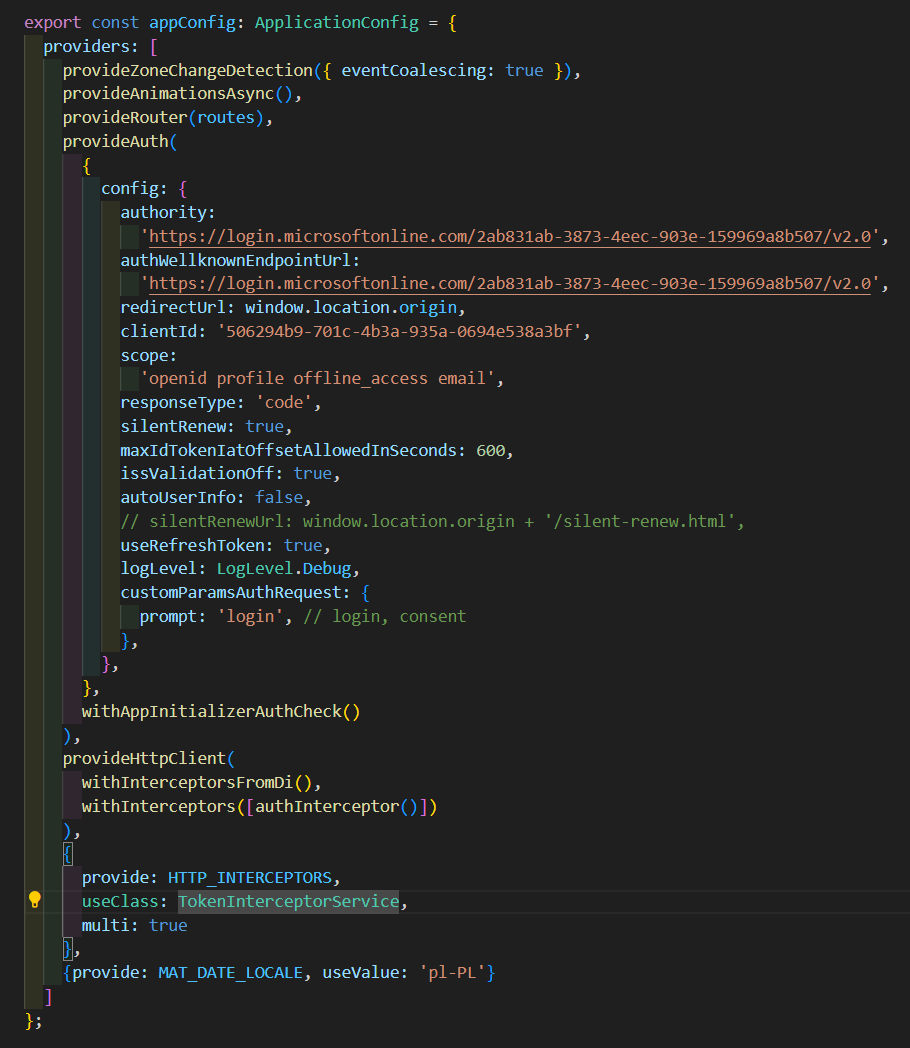
Rys. 3.1 Plik package.json zarządzający zależnoścami projektu (opracowanie własne)

Natomiast globalna konfiguracja Angularowa umieszczona jest w pliku *angular.json*, co można zaobserwować na rysunku 3.2. Zawiera ona m.in. odwołania do plików CSS/SCSS definiujące style, czyli wygląd aplikacji wyświetlanej w przeglądarce.



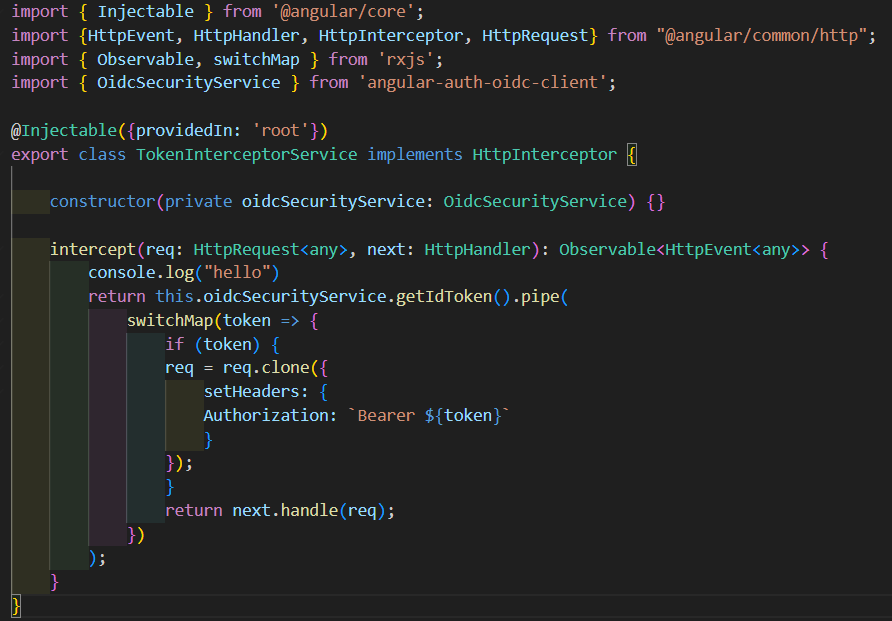
Rys. 3.2 Plik angular.json będący globalną konfiguracją Angularową (opracowanie własne)

Oprócz globalnej konfiguracji Angularowej, istnieje również plik konfiguracyjny w języku TypeScript, który nosi nazwę *app.config.ts*. Zawiera on m.in. konfigurację warstwy uwierzytelnienia realizowana przy użyciu Microsoft Entra ID, konfigurację regionu dla biblioteki Angular Material, definicję serwisu odpowiedzialnego za dodawanie access tokena do każdego żądania HTTP wysłanego do backendu oraz konfigurację listy podstron internetowych dostępnych w aplikacji frontendowej. Plik *app.config.ts* można zaobserwować na rysunku 3.3.



Rys. 3.3 Plik app.config.ts stanowiący konfigurację aplikacji w języku TypeScript (własne opracowanie)

Kolejnym ważny elementem konfiguracji jest klasa *TokenInterceptorService* przedstawiona na rysunku 3.4. Przechwytuje ona każde żądanie HTTP wysyłane do backendu i dodaje nagłówek o nazwie *Authorization*, gdzie umieszcza access token pozyskany od Microsoft Entra ID. W ten sposób zapewnione jest uwierzytelnienie w warstwie komunikacji pomiędzy frontendem a backendem.



Rys. 3.4 Klasa TokenInterceptorService (opracowanie własne)

Oprócz powyższych elementów konfiguracji, istotnym z punktu widzenia projektu jest również lista zdefiniowanych podstron dostępnych w aplikacji frontendowej. Znajduje się ona w pliku *app.routes.ts,* co zostało przedstawione na rysunku 3.5. Każda podstrona identyfikowana jest zapomocą atrybutu *path*, który odzwierciedla część linku URL prowadzącego do danej podstrony. Ponadto, każda z podstron zdefiniowana jest za pomocą komponentu Angularowego.

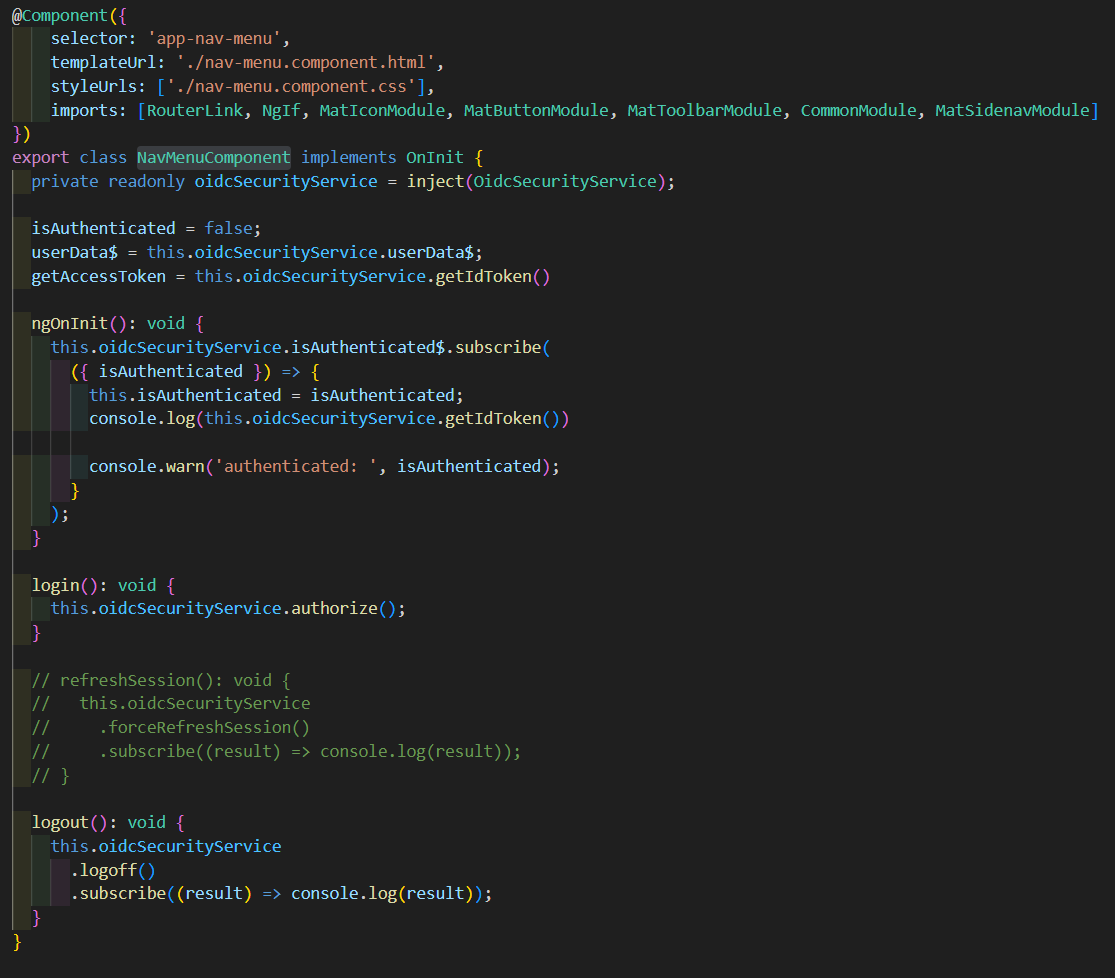


Rys. 3.5 Lista podstron aplikacji zdefiniowana w pliku app.routes.ts (własne opracowanie)

Nie każda podstrona jest dostępna dla wszystkich użytkowników. Konfiguracja dostępów do podstron została zdefiniowana w komponencie menu o nazwie *NavMenuComponent,* a konkretnie w warstwie prezentacji tego komponentu, czyli w pliku *nav-menu.component.html*. Kod ten pokazany został na rysunku 3.6. Można zauważyć, że zastosowany został atrybut *ngIf,* który decyduje o tym, czy dana podstrona zostanie zaprezentowana w menu, czy też nie. Decyzja ta oparta jest na podstawie przynależności zalogowanego użytkownika do odpowiedniej grupy w Microsoft Entra ID. Grupy, do których przynależy użytkownik zawarte są w treści access tokena, a następnie przekazywane do pola o nazwie *userData* w komponencie *NavMenuComponent*, co pokazane zostało na rysunku 3.7.

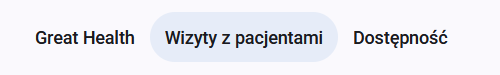


Rys. 3.6 Warstwa prezentacji komponentu menu (własne opracowanie)



Rys. 3.7 Warstwa logiki komponentu menu (własne opracowanie)

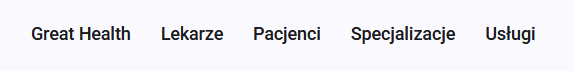
Na rysunkach 3.8, 3.9 oraz 3.10 zaprezentowane zostało menu odpowiednio dla lekarza, pacjenta oraz recepcjonisty.



Rys. 3.8 Menu wyświetlane dla użytkownika będącego lekarzem (własne opracowanie)



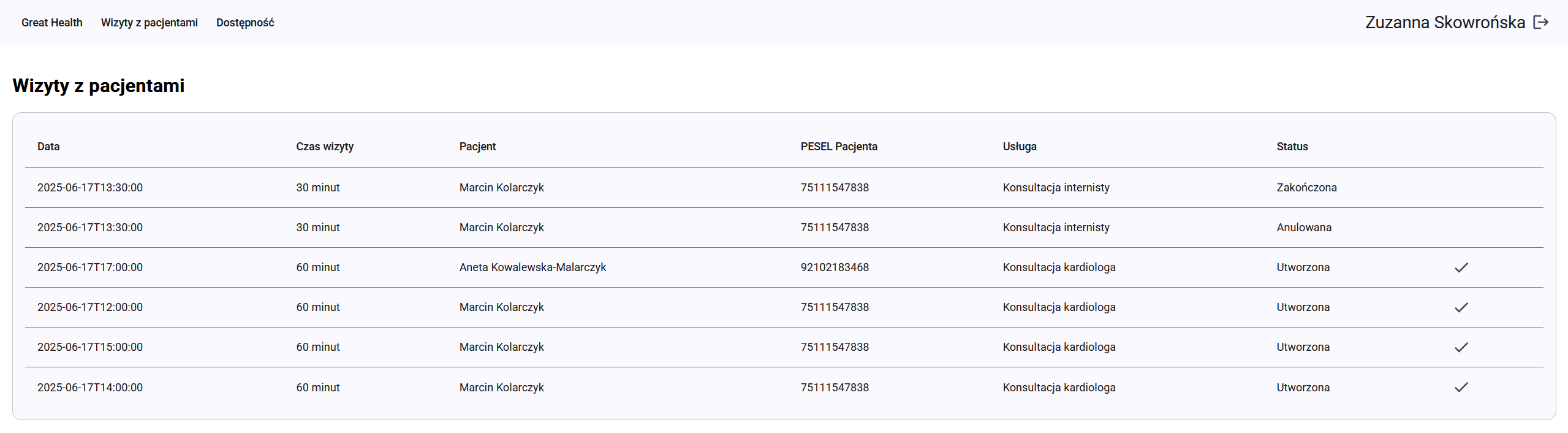
Rys. 3.9 Menu wyświetlane dla użytkownika będącego pacjentem (własne opracowanie)



Rys. 3.10 Menu wyświetlane dla użytkownika będącego recepcjonistą (własne opracowanie)

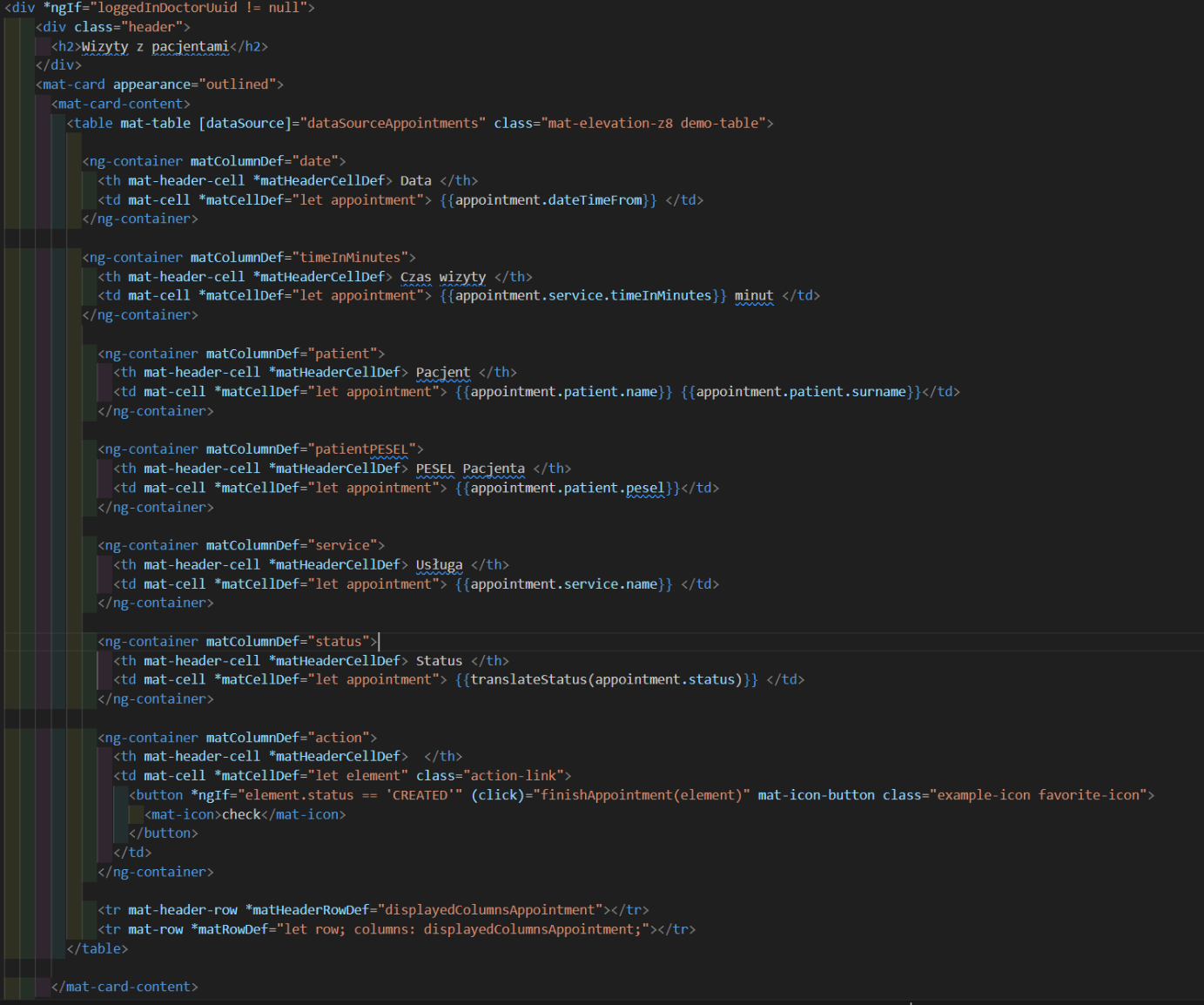
## Podstrona Wizyty z pacjentami

Do podstrony z wizytami z pacjentami dostęp ma jedynie lekarz. Na rysunku 3.11 przedstawiony jest jej wygląd. Składa się ona z tabeli wyświetlającej wizyty z pacjentami wraz ze szczegółami takimi jak data wizyty, czas wizyty, imię i nazwisko pacjenta, PESEL pacjenta, nazwa usługi medycznej, oraz statusu. Ponadto, po prawej stronie dla aktywnych wizyt, lekarz ma możliwość potwierdzić zakończenie wizyty poprzez naciśnięcie na ikonę. Wówczas status wizyty zostanie zmieniony na *Zakończona*. Wizyta może być anulowana tylko i wyłącznie przez pacjenta.



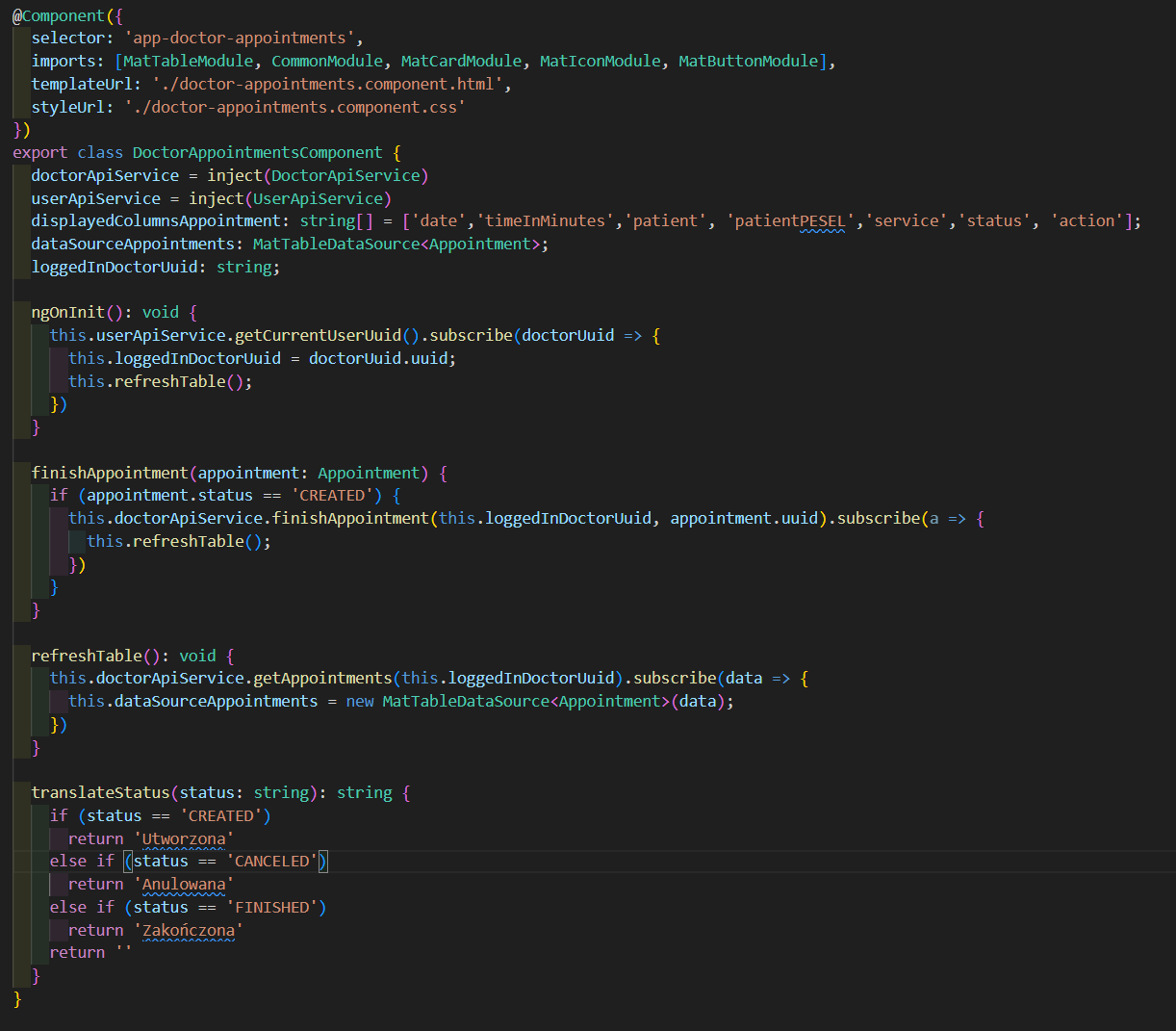
Rys. 3.11 Podstrona Wizyty z pacjentami (opracowanie własne)

Rysunek 3.12 przedstawia kod warstwy prezentacji komponentu definiującego podstronę Wizyty z pacjentami. Zawiera on definicję tabeli wraz ze wszystkimi jej kolumnami. Poza tabelą, utworzona została również ikona, za pomocą której lekarz może potwierdzić zakończenie wizyty. Warto również zwrócić uwagę na element *div* będący kontenerem dla całej podstrony. Został dodany dla niego atrybut *ngIf*, który kontroluje kiedy element istnieje, a kiedy nie. Warunkiem na jego istnieje w tym przypadku jest zalogowany użytkownik. Jeżeli użytkownik nie został zalogowany, podstrona nie powinna się wyświetlać.



Rys. 3.12 Warstwa prezentacji komponentu dla podstrony Wizyty z pacjentami (opracowanie własne)

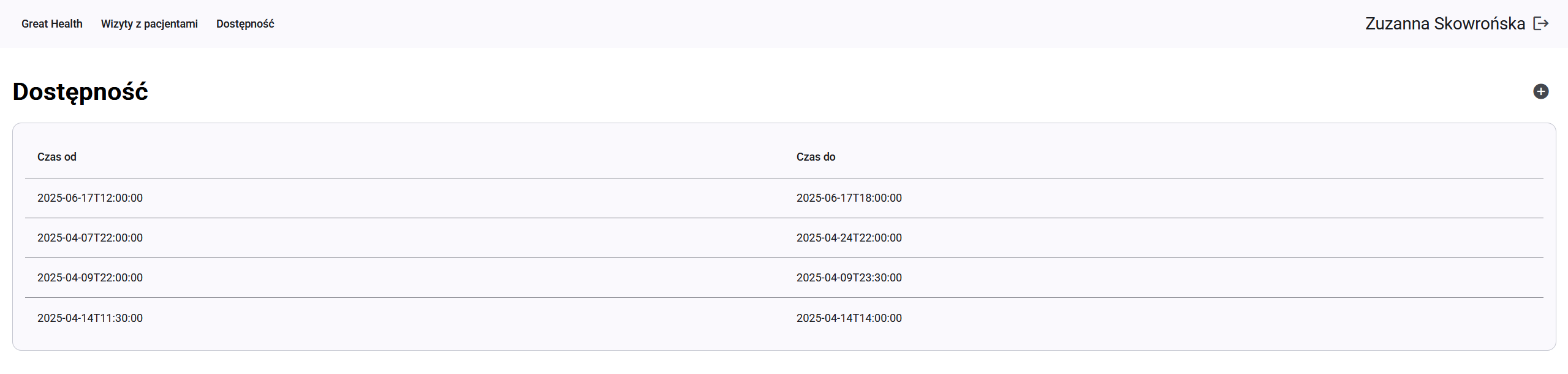
Na rysunku 3.13 przedstawiony został kod logiki komponentu. Można zauważyć metodę *ngOnInit* wykonywaną, gdy podstrona jest tworzona. Metoda ta odpowiada za pozyskanie identyfikatora aktualnie zalogowanego użytkownika oraz za pobranie danych wyświetlanych w tabeli. Ponadto, w warstwie logiki utworzona została również metoda kończąca wizytę, która jest wykonywana z poziomu warstwy prezentacji, gdy użytkownik naciśnie na odpowiedni przycisk. Warto również zauważyć, że istnieje tam metoda konwertująca techniczne wartości statusu wizyty na status w języku polskim wyświetlany użytkownikowi.



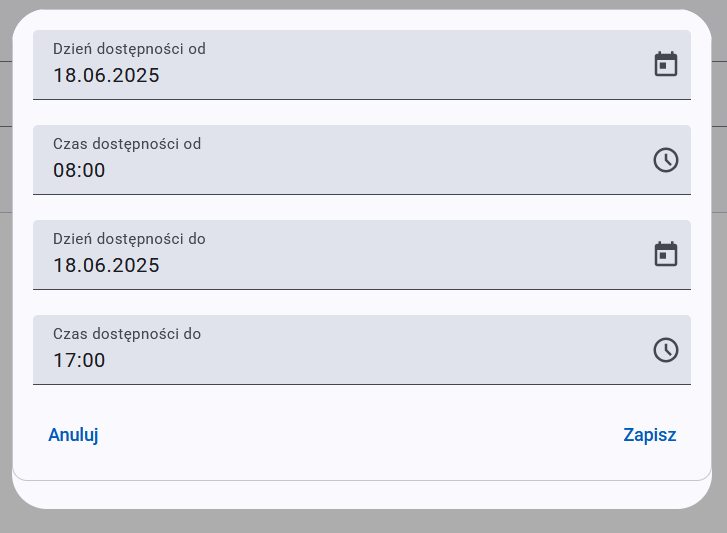
Rys. 3.13 Definicja warstwy logiki dla podstrony Wizyty z pacjentami (opracowanie własne)

## Podstrona Dostępność

Podstrona Dostępność dostępna jest tylko i wyłącznie dla lekarzy. Umożliwia ona zarządzanie dostępnością czasową lekarza. Dzięki wprowadzeniu tych danych, pacjenci mogą umawiać się na wizyty w określonych terminach. Rysunek nr 3.14 przedstawia wygląd tej podstrony. Zawiera on tabelę wyświetlającą grafiki pracy zalogowanego lekarza. Lekarz może dodawać nowe terminy dostępności poprzez naciśnięcie ikony plusa znajdującej się w prawym górnym rogu. Rysunek 3.15 przedstawia formularz służący do dodawania nowym terminów dostępności.

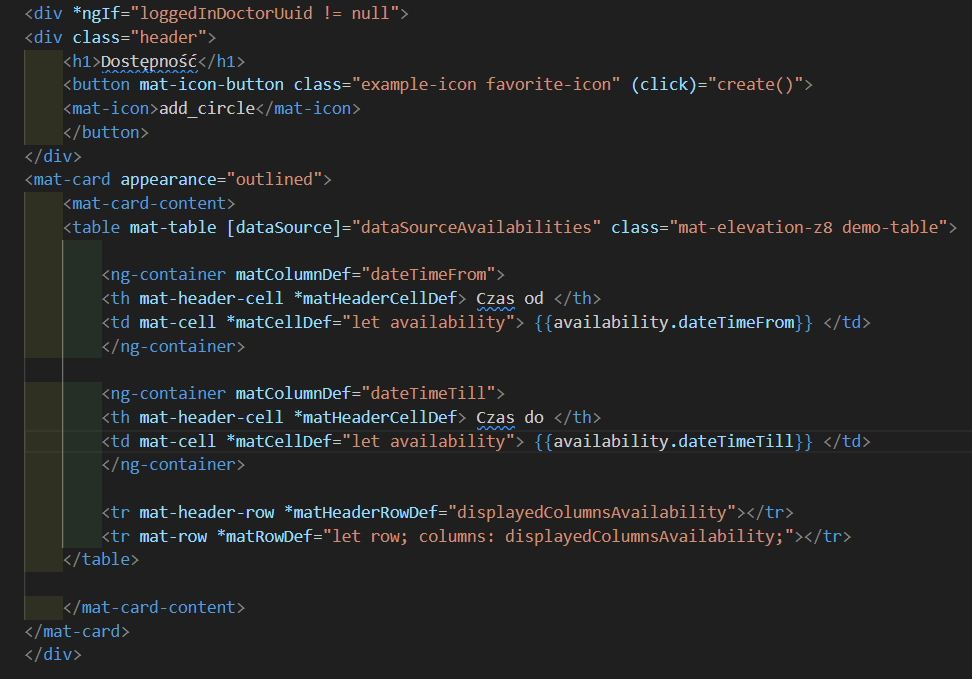


Rys. 3.14 Wygląd podstrony Dostępność (opracowanie własne)



Rys. 3.15 Formularz dodający nowy termin dostępności lekarza (opracowanie własne)

Rysunek 3.16 przedstawia kod warstwy prezentacji podstrony Dostępność, zaś rysunek 3.17 kod logiki biznesowej dla tej podstrony. Podstrona zawiera tabelę z dwoma kolumnami oraz ikonę, po której naciśnięciu wywoływana jest metoda *create*, która tworzy i wyświetla formularz. Kiedy formularz zostaje zamknięty i model przez niego zwrócony nie jest pusty, wówczas zostaje on przesłany do API backendu w celu zapisania nowo dodanych terminów lekarza.

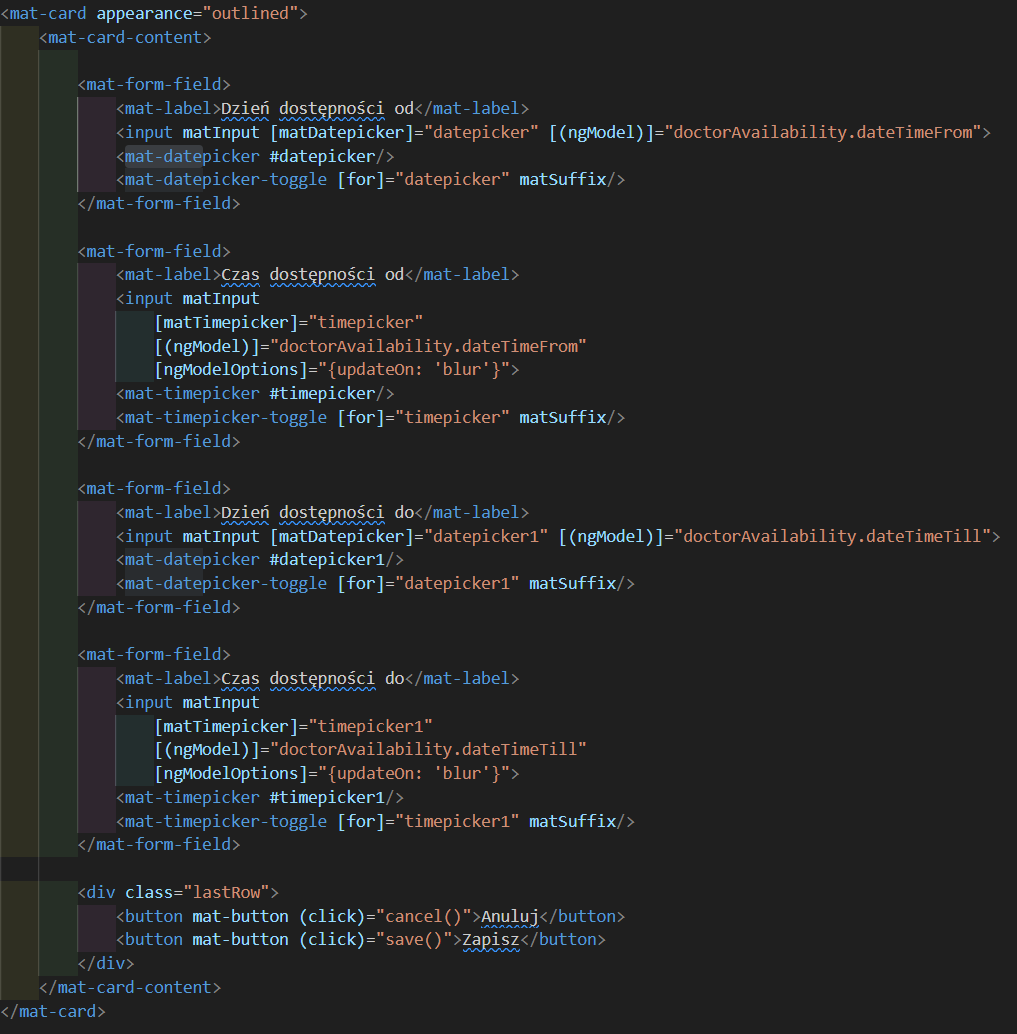


Rys. 3.16 Warstwa prezentacji komponentu dla podstrony Dostępność (opracowanie własne)

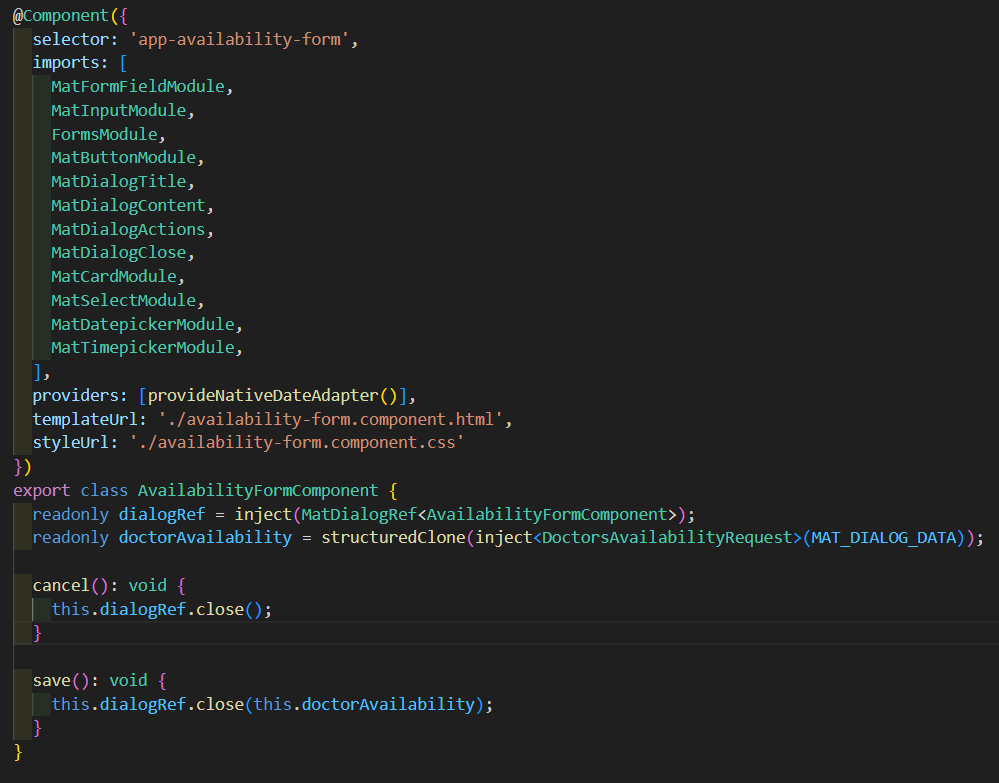


Rys. 3.17 Definicja warstwy logiki dla podstrony Wizyty z pacjentami (opracowanie własne)

Kod warstwy prezentacji formularza przedstawiony jest na rysunku 3.18, zaś logiki biznesowej na rysunku 3.19. W warstwie prezentacji zdefiniowane są pole określające zakres czasu, w którym lekarz jest dostępny. Poniżej zdefiniowane są przyciski odpowiednio anulujący lub zapisujący nowy termin dostępności lekarza. Metody *save* lub *cancel* są wówczas wywoływane, a co za tym idzie formularz jest zamykany z zapisanym modelem lub bez.



Rys. 3.18 Warstwa prezentacji formularza służącego do dodawania dostępności (opracowanie własne)



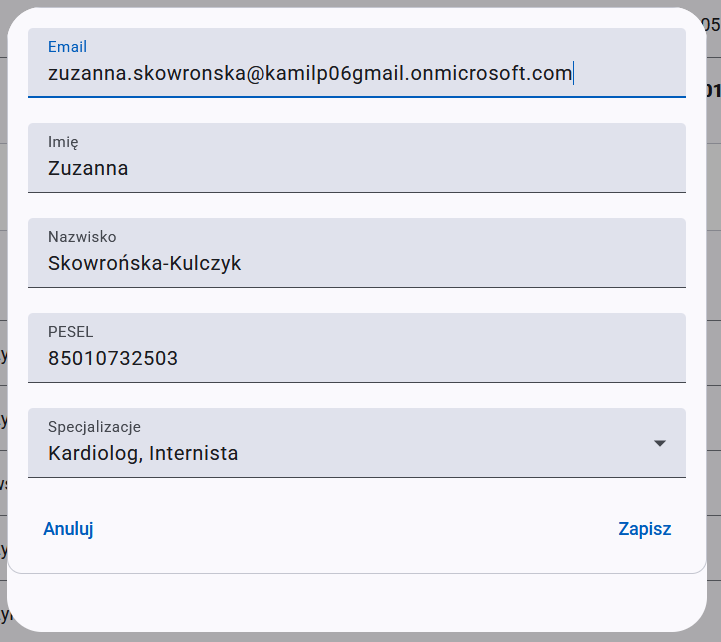
Rys. 3.19 Definicja warstwy logiki dla formularza służącego do dodawania dostępności (opracowanie własne)

## Podstrona Lekarze

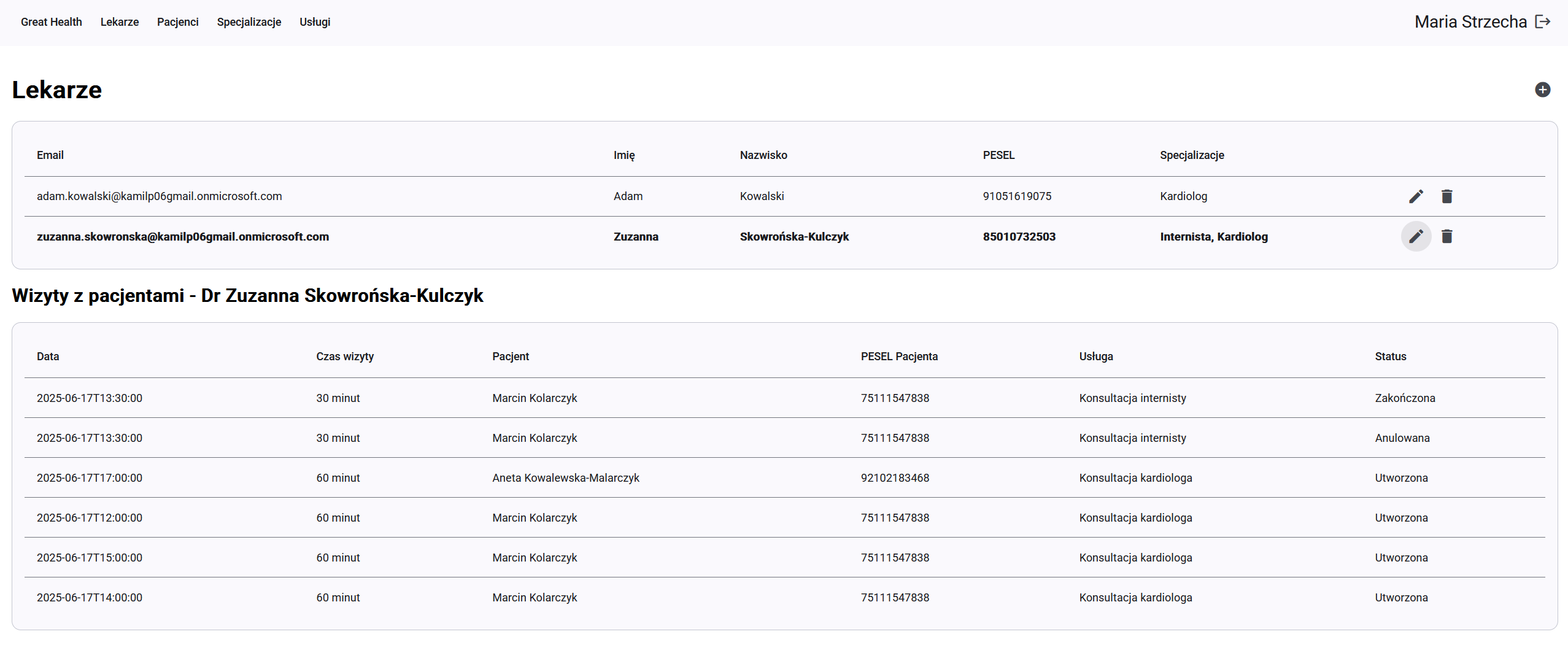
Podstrona Lekarze jest dostępna tylko i wyłącznie dla recepcjonisty. Jej wygląd jest zaprezentowany na rysunku 3.20. Składa się na nią tabela przedstawiająca listę lekarzy wraz z ich danymi takimi jak email, dane osobowe i specjalizacje. Ponadto, znajdują się tam przyciski służące do modyfikacji tych danych, usuwania oraz dodawania lekarzy. Po naciśnięciu przycisku edycji danych lekarza lub przycisku do dodawania nowych lekarzy pojawia się formularz przedstawiony na rysunku 3.21. W przypadku edycji formularz jest od razu wypełniony istniejącymi już danymi, natomiast w przypadku dodawania nowego lekarza formularz jest pusty. Oprócz wymienionych już funkcji, podstrona Lekarze pozwala na wybranie lekarza i podgląd jego wizyt. Rysunek 3.22 przedstawia tę funkcję dla wybranego lekarza.



Rys. 3.20 Wygląd podstrony Lekarze (opracowanie własne)

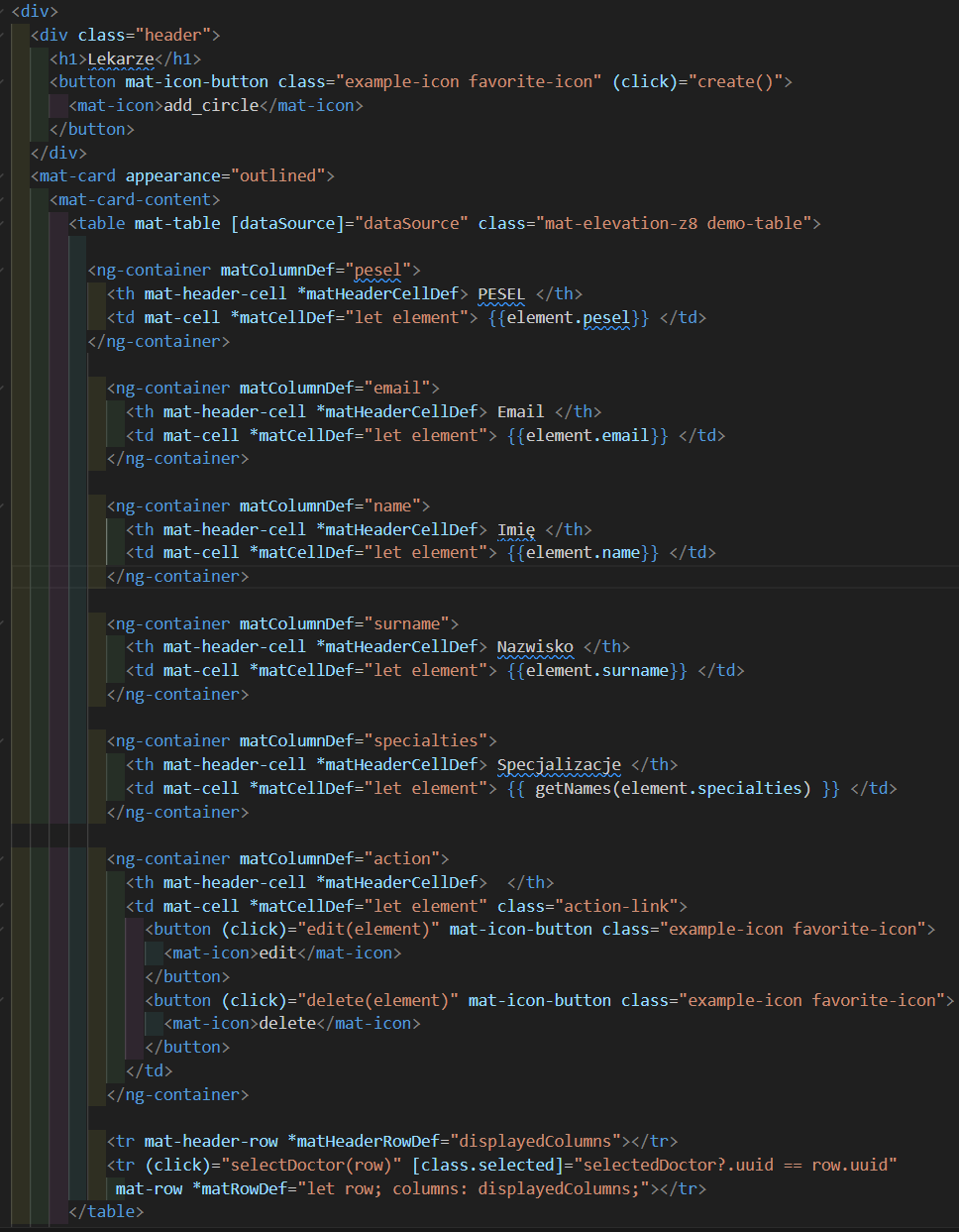


Rys. 3.21 Formularz dodający/edytujący dane lekarza (opracowanie własne)



Rys. 3.22 Podgląd wizyt wybranego lekarza na podstronie Lekarze (opracowanie własne)

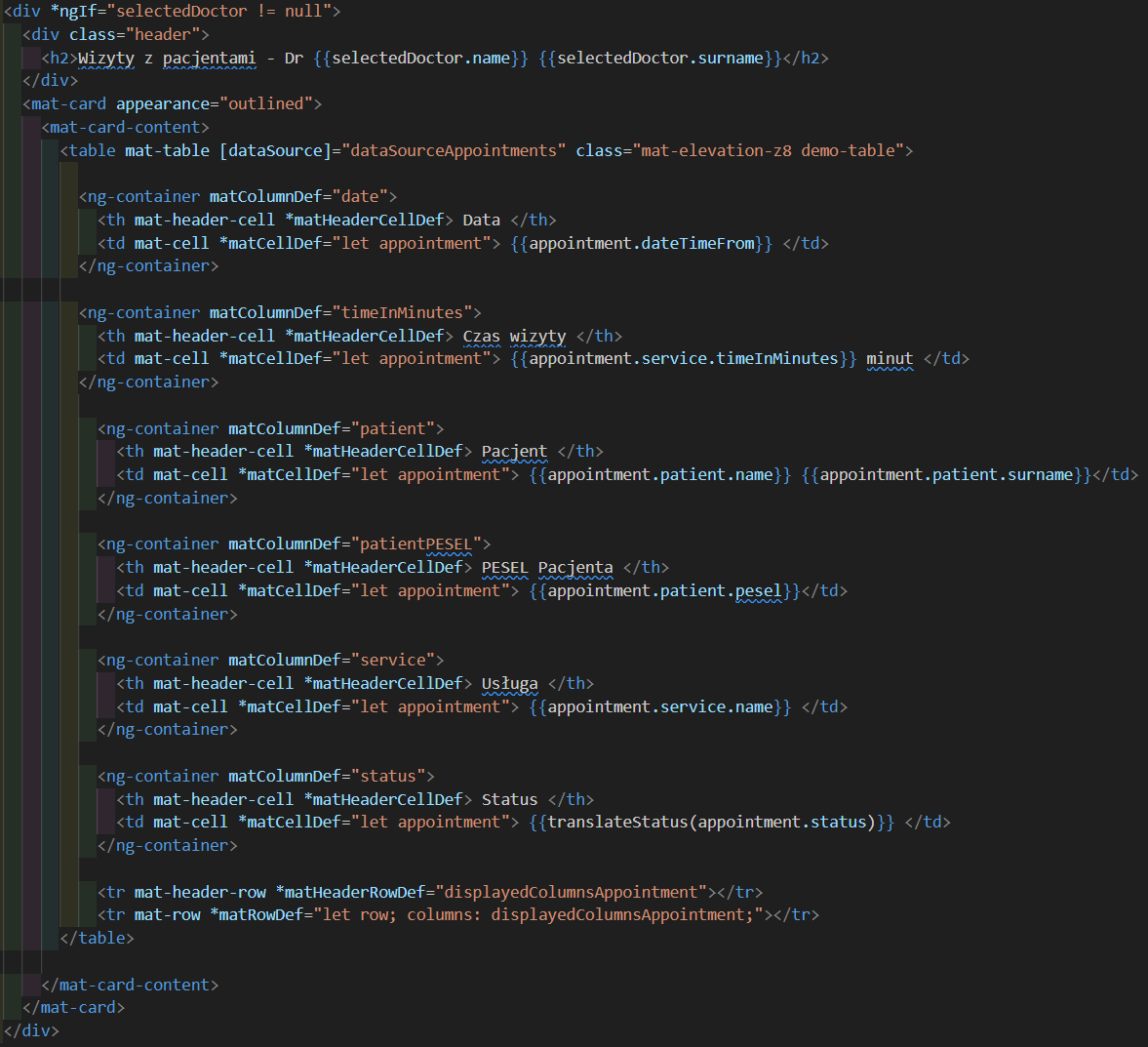
Na rysunku 3.23 przedstawiony jest fragment kodu warstwy prezentacji podstrony Lekarze wyświetlający tabelę z lekarzami. Zaś na rysunku 3.24 przedstawiony jest kod logiki biznesowej, obsługujący tę tabelę. Sama tabela jest obsługiwana w podobny sposób do poprzednio przedstawionych, natomiast na wyszczególnienie zasługuje użycie metody *getNames*, która przyjmuje jako argument listę specjalizacji danego lekarza, a następnie zwraca ciąg znaków zawierający ich nazwy odseparowane przecinkiem. Na rysunku 3.25 przedstawiony jest kod warstwy prezentacji dla tabeli przedstawiającej wizyty wybranego pacjenta, którego to wybór odbywa się na pomocą metody *selectDoctor* z rysunku 3.24.



Rys. 3.23 Fragment kodu z warstwy prezentacji wyświetlający tabelę z lekarzami na podstronie Lekarze (opracowanie własne)

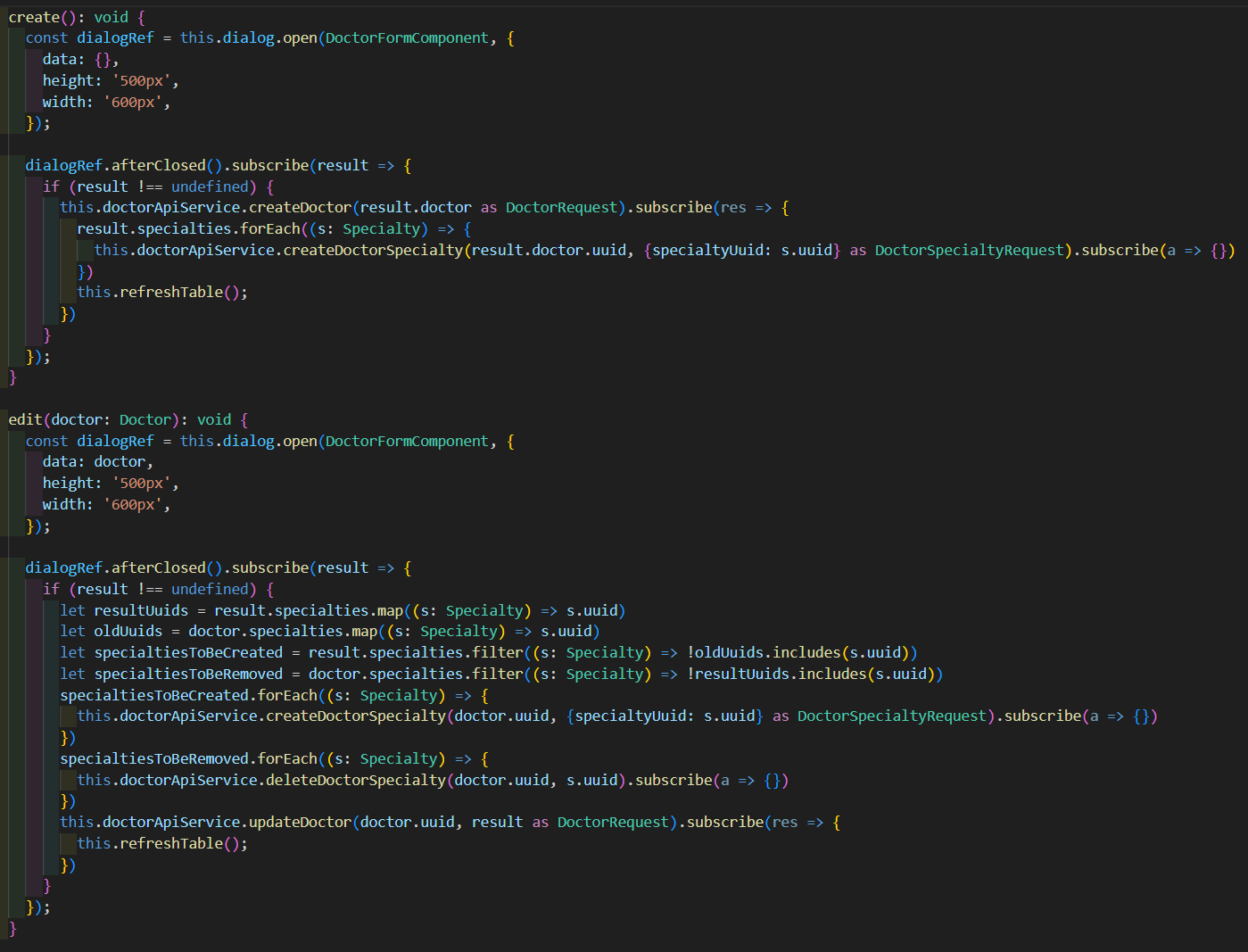


Rys. 3.24 Fragment kodu z warstwy logiki obsługującej tabelę z doktorami na podstronie Lekarze (opracowanie własne)



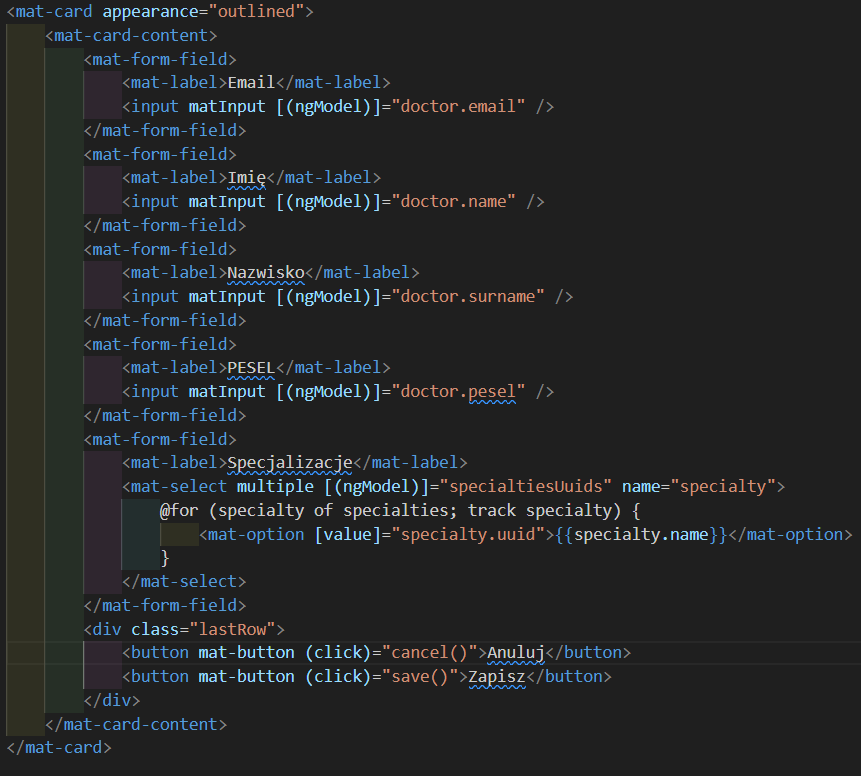
Rys. 3.25 Fragment kodu z warstwy prezentacji wyświetlający tabelę z wizytami dla wybranego lekarza na podstronie Lekarze (opracowanie własne)

Na uwagę zasługuje również kod powiązany z formularzem edycji oraz dodawania doktorów. Rysunek 3.26 przedstawia fragment z warstwy logiki, który jest wykonywany w celu utworzenia formularza, a także gdy jest on zamykany. W sytuacji, gdy formularz zapisuje dane nowego lekarza, zapisywany jest lekarz poprzez wywołanie odpowiedniego endpointa z backendu, a następnie przypisywane są do tego lekarza wybrane specjalizacje. W przypadku, gdy formularz zapisuje dane istniejącego już lekarza proces oprócz modyfikacji danych bezpośrednio związanych z doktorem i dodania nowych specjalizacji, musi również uwzględnić możliwość usunięcia istniejących specjalizacji.

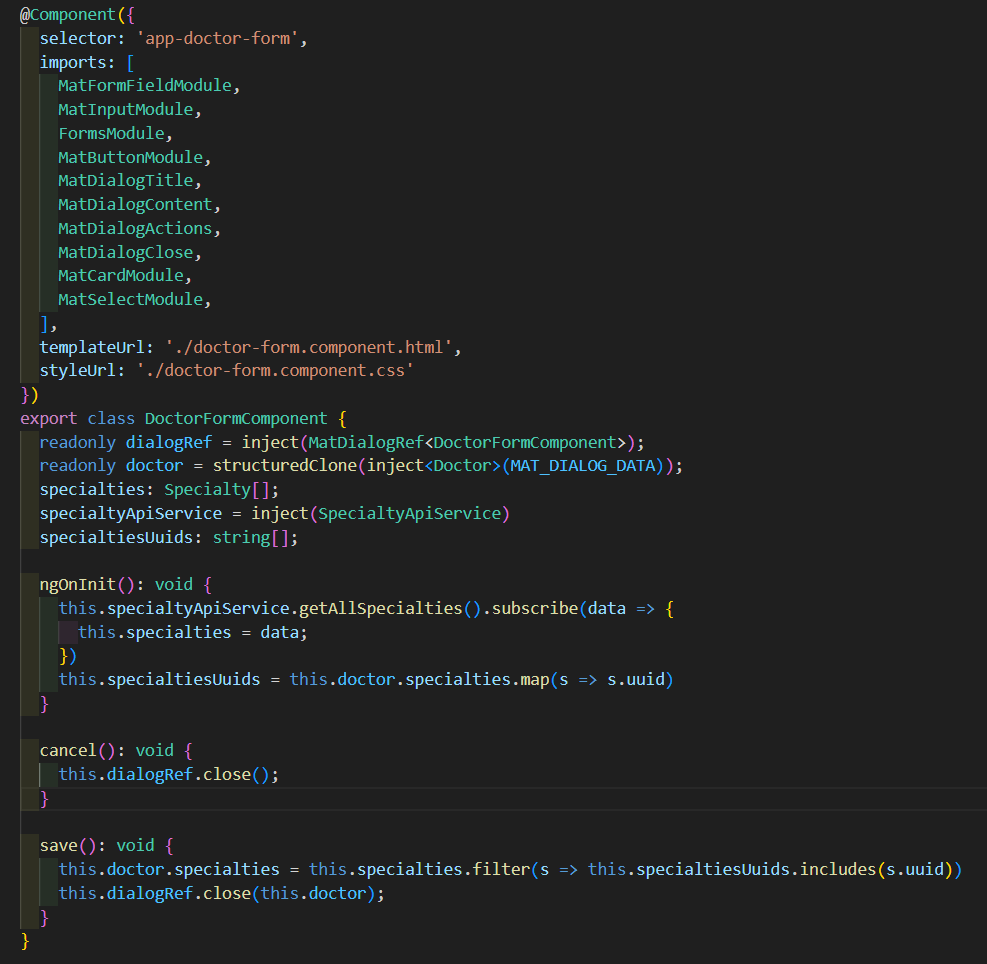


Rys. 3.26 Fragment kody z warstwy logiki obsługujący tworzenie oraz zamykanie formularza na podstronie Lekarze (opracowanie własne)

Rysunek 3.27 przedstawia kod z warstwy prezentacji formularza, zaś rysunek 3.28 przedstawia kod z warstwy logiki. Formularz zawiera pola z danymi osobowymi lekarza oraz pole pozwalające na zaznaczenie wielu specjalizacji. Specjalizacje są pobierane z API backendu, a następnie wyświetlane w warstwie prezentacji.



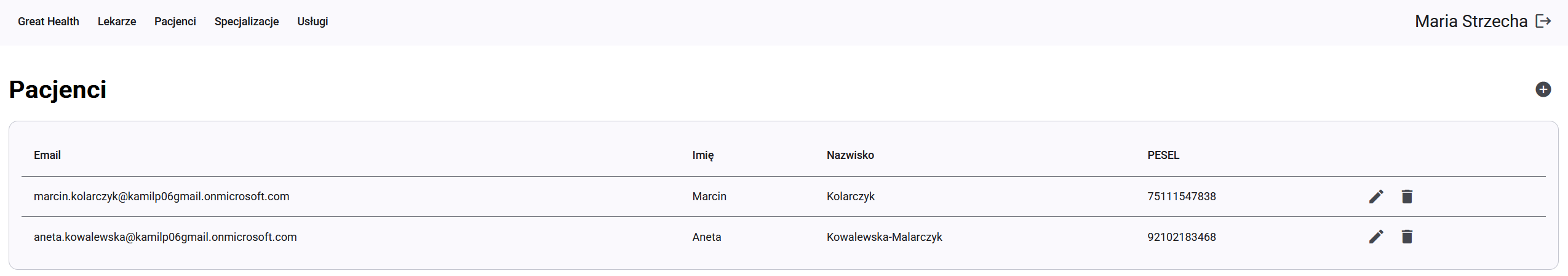
Rys. 3.27 Kod z warstwy prezentacji formularza z podstrony Lekarze (opracowanie własne)



Rys. 3.28 Kod z warstwy logiki formularza z podstrony Lekarze (opracowanie własne)

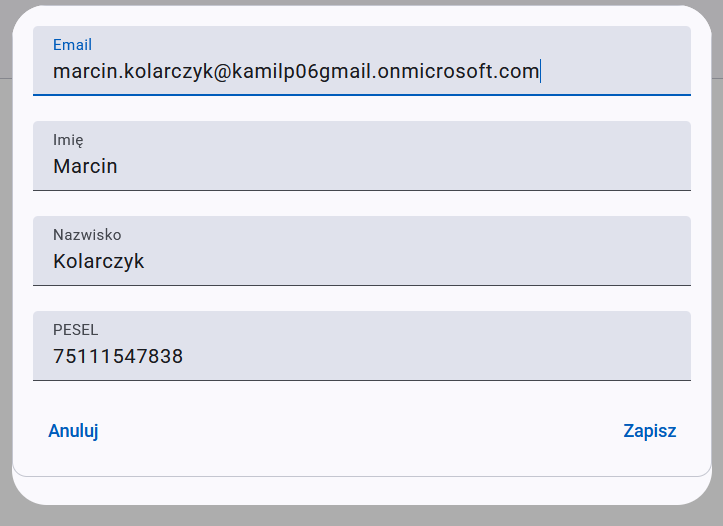
## Podstrona Pacjenci

Podstrona Pacjenci wyświetlana jest tylko recepcjonistom i pozwala na zarządzanie danymi pacjentów. Rysunek nr 3.29 przedstawia jej wygląd. Podstrona ta składa się z tabeli wyświetlającej dane osobowe pacjentów wraz z ich adresami email. Ponadto, tabela zawiera przyciski służące do modyfikacji danych istniejących pacjentów oraz ich usuwania. Podstrona umożliwia także dodanie nowego pacjenta poprzez naciśnięcie ikony plusa.



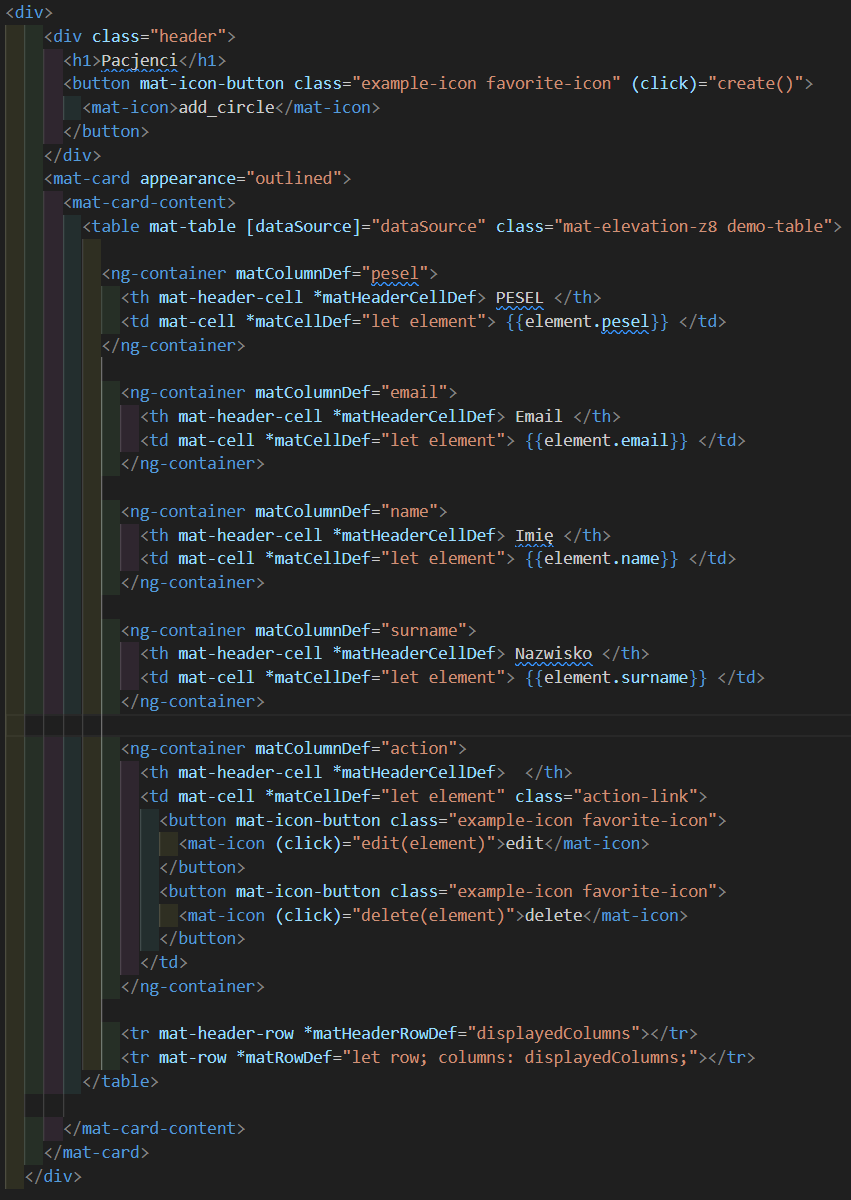
Rys. 3.29 Wygląd podstrony Pacjenci (opracowanie własne)

Rysunek nr 3.30 przedstawia formularz wyświetlany po naciśnięciu przycisku dodania lub edycji pacjenta. Pozwala on na wprowadzenie danych pacjenta oraz ich zapis.

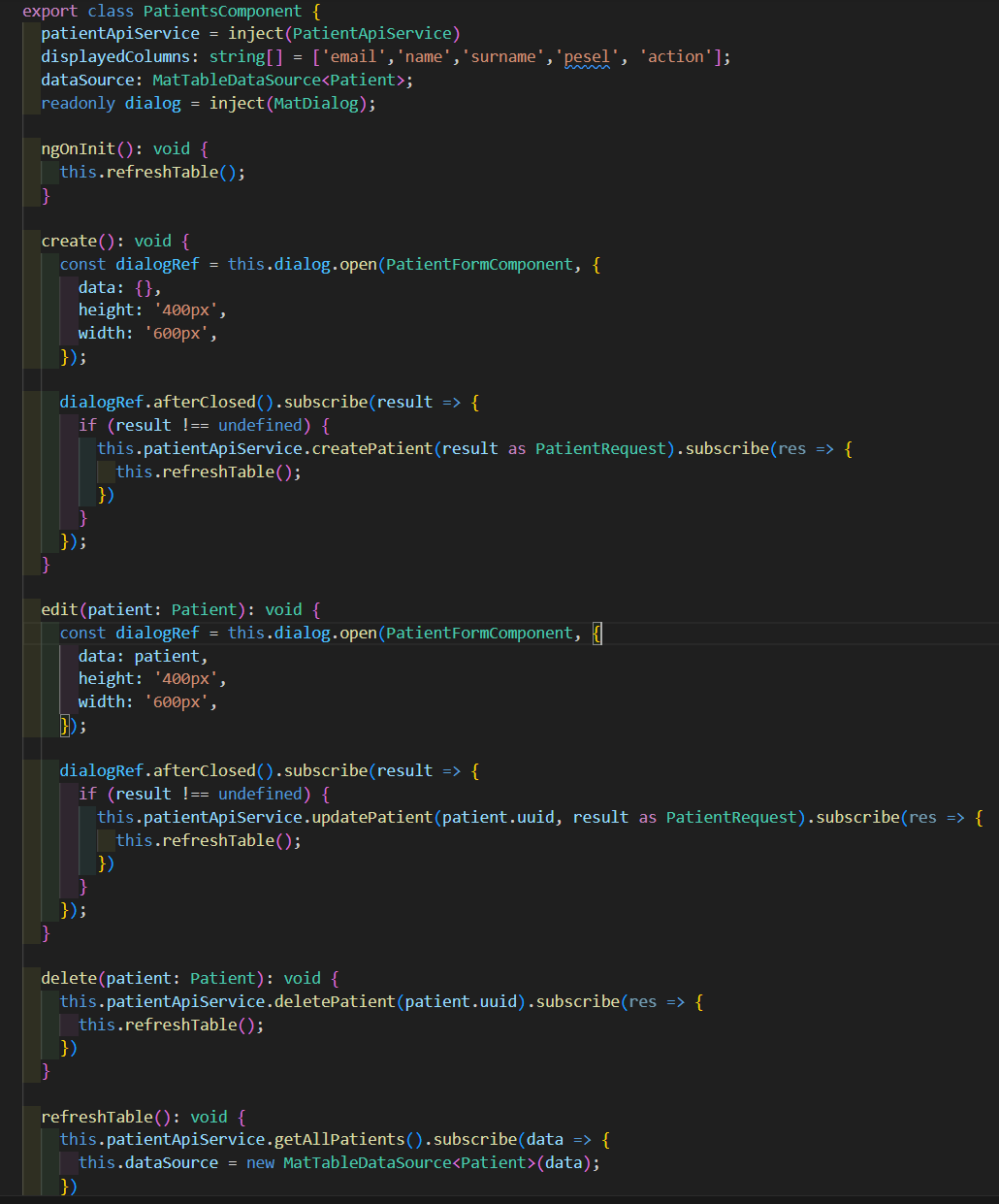


Rys. 3.30 Wygląd formularza na podstronie Pacjenci (opracowanie własne)

Rysunek nr 3.31 przedstawia kod warstwy prezentacji podstrony Pacjenci, natomiast rysunek nr 3.32 kod logiki.

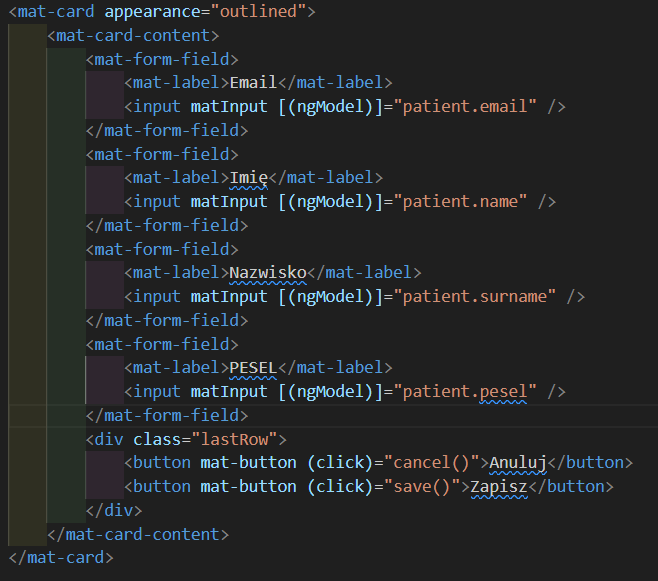


Rys. 3.31 Kod warstwy prezentacji podstrony Pacjenci (opracowanie własne)

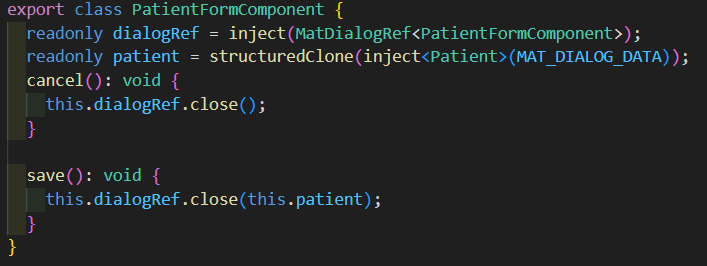


Rys. 3.32 Kod warstwy logiki podstrony Pacjenci (opracowanie własne)

Warstwa prezentacji na zdefiniowaną tabelę wraz z danymi osobowymi pacjenta i jego adresem email. Ponadto, w prawym górnym rogu znajduje się przycisk dodający nowego pacjenta.   
W ostatniej kolumnie zdefiniowane są zaś przyciski służące do edycji i usuwania pacjentów. W przypadku edycji wywoływana jest metoda *edit*, zaś usuwania metoda *delete*. Metoda *delete* wywołuje odpowiedni endpoint z API backendu w celu usunięcia pacjenta. Natomiast, edycja jak i dodanie pacjenta, skutkują utworzeniem formularza. Kod warstwy prezentacji formularza znajduje się na rysunku nr 3.33, zaś jego logiki na rysunku 3.34. Formularz ten zawiera pola z danymi osobowymi oraz adresem email pacjenta. W przypadku edycji, przekazywany jest model pacjenta do warstwy logiki formularza tak, aby ten wyświetlił istniejące już dane pacjenta. W przypadku tworzenia nowego pacjenta, formularz jest pusty.



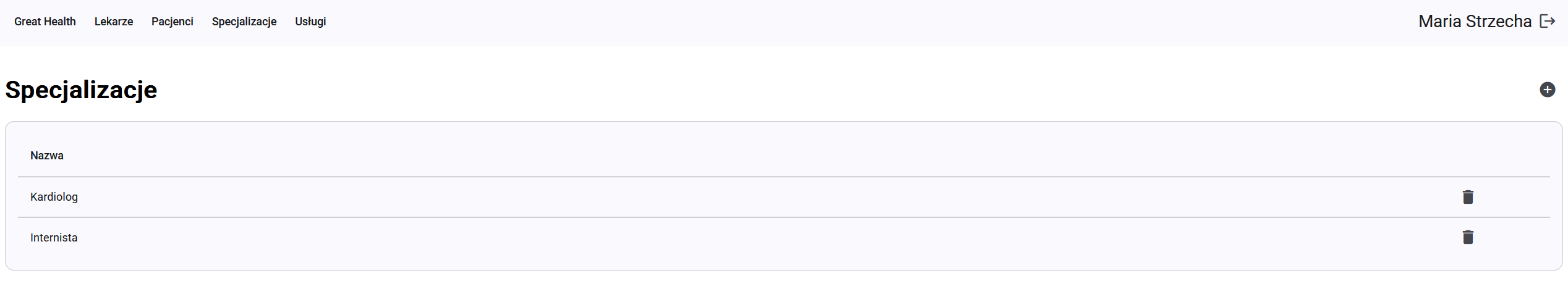
Rys. 3.33 Kod warstwy prezentacji formularza z podstrony Pacjenci (opracowanie własne)



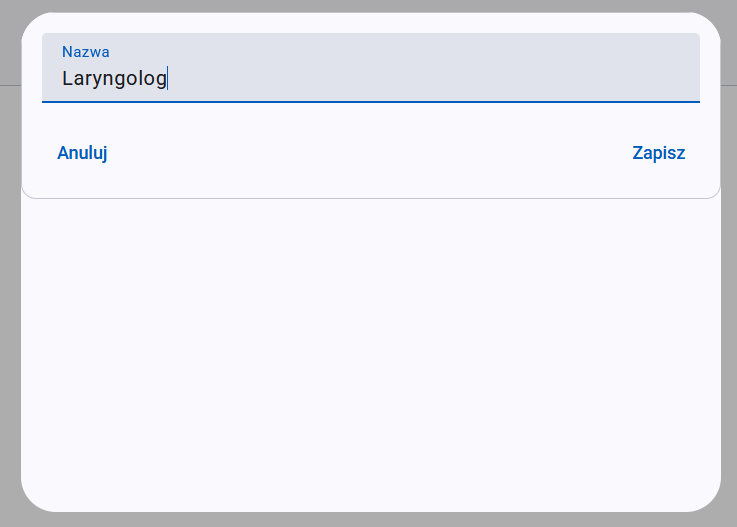
Rys. 3.34 Kod warstwy logiki formularza z podstrony Pacjenci (opracowanie własne)

## Podstrona Specjalizacje

Podstrona Specjalizacje dostępna jest tylko i wyłącznie dla użytkowników będących recepcjonistami. Służy ona do zarządzania danymi dot. specjalizacji. Rysunek nr 3.35 przedstawia wygląd tej podstrony. Zawiera on tabelę przedstawiające specjalizacje wraz z możliwością ich usunięcia (ikona kosza) i dodania (ikona plusa).

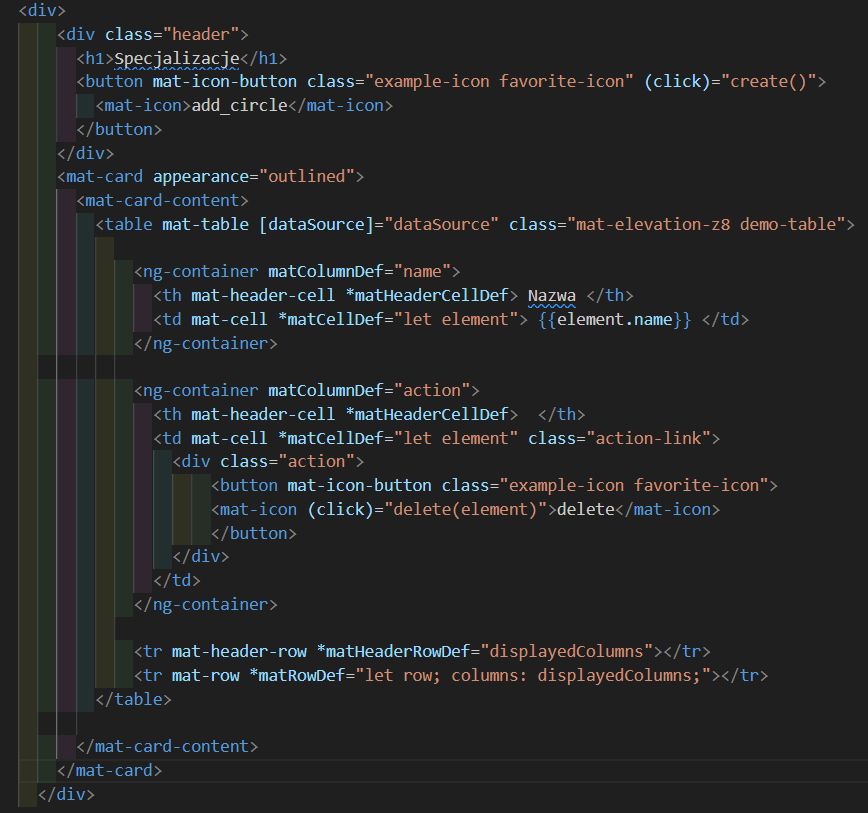


Rys. 3.35 Wygląd podstrony Specjalizacje (opracowanie własne)

Po naciśnięciu przycisku dodawania nowej specjalizacji wyświetlone zostaje nowe okno przedstawione na rysunku nr 3.36. Zawiera ono jedynie pole tekstowe z nazwą nowo tworzonej specjalizacji. Po naciśnięciu przycisku Zapis, nowa specjalizacja zostaje zapisana w systemie.

Rys. 3.36 Wygląd okna służącego do dodawania nowych specjalizacji (opracowanie własne)

Rysunek nr 3.37 przedstawia kod źródłowy warstwy prezentacji podstrony Specjalizacje. Zawiera on definicję tabeli oraz dwóch przycisków w postaci ikon. Rysunek nr 3.38 przedstawia kod źródłowy warstwy logiki, gdzie zdefiniowane zostały funkcje wywoływane po naciśnięciu dostępnych przycisków – funkcja *create* oraz *delete*.



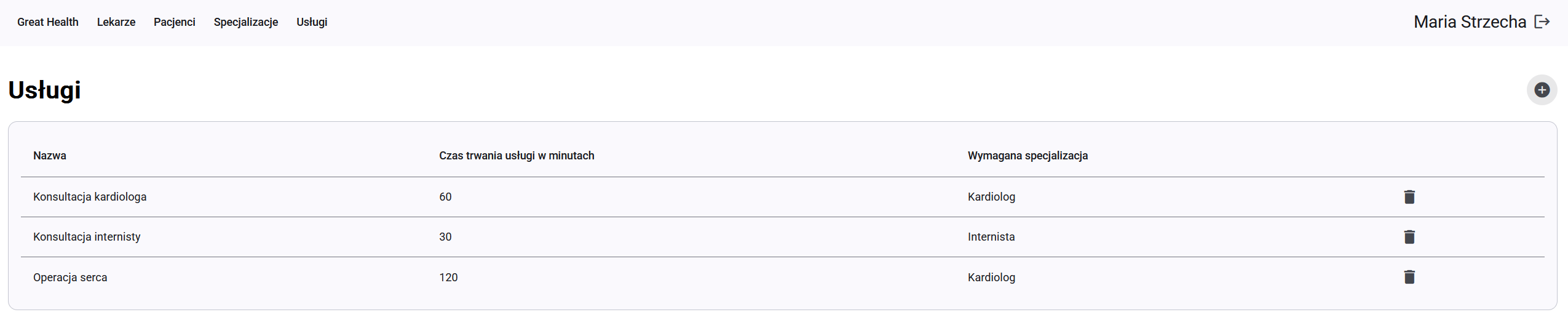
Rys. 3.37 Kod źródłowy warstwy prezentacji podstrony Specjalizacje (opracowanie własne)



Rys. 3.38 Kod źródłowy warstwy logiki podstrony Specjalizacje (opracowanie własne)

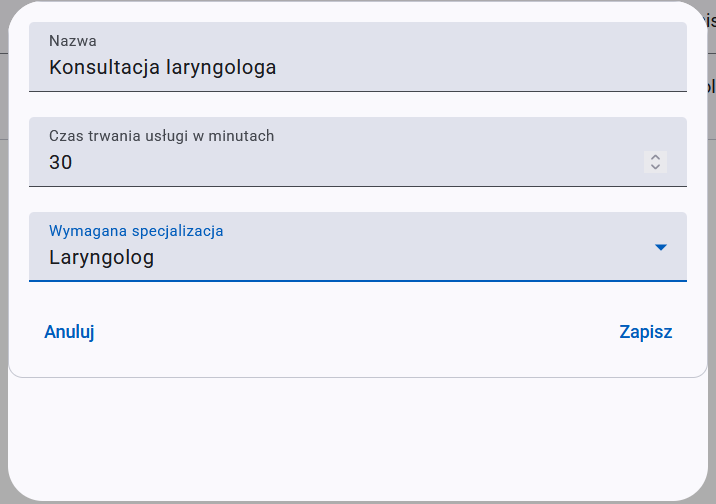
## Podstrona Usługi

Podstrona Usługi dostępna jest dla użytkowników będących recepcjonistami i służy do zarządzania usługami medycznymi. Zawiera ona tabelę z usługami, która przedstawia nazwę usługi, czas jej trwania oraz wymaganą specjalizację, które musi posiadać dany lekarz, aby móc wykonać usługę. Rysunek nr 3.39 przedstawia wygląd tej podstrony.



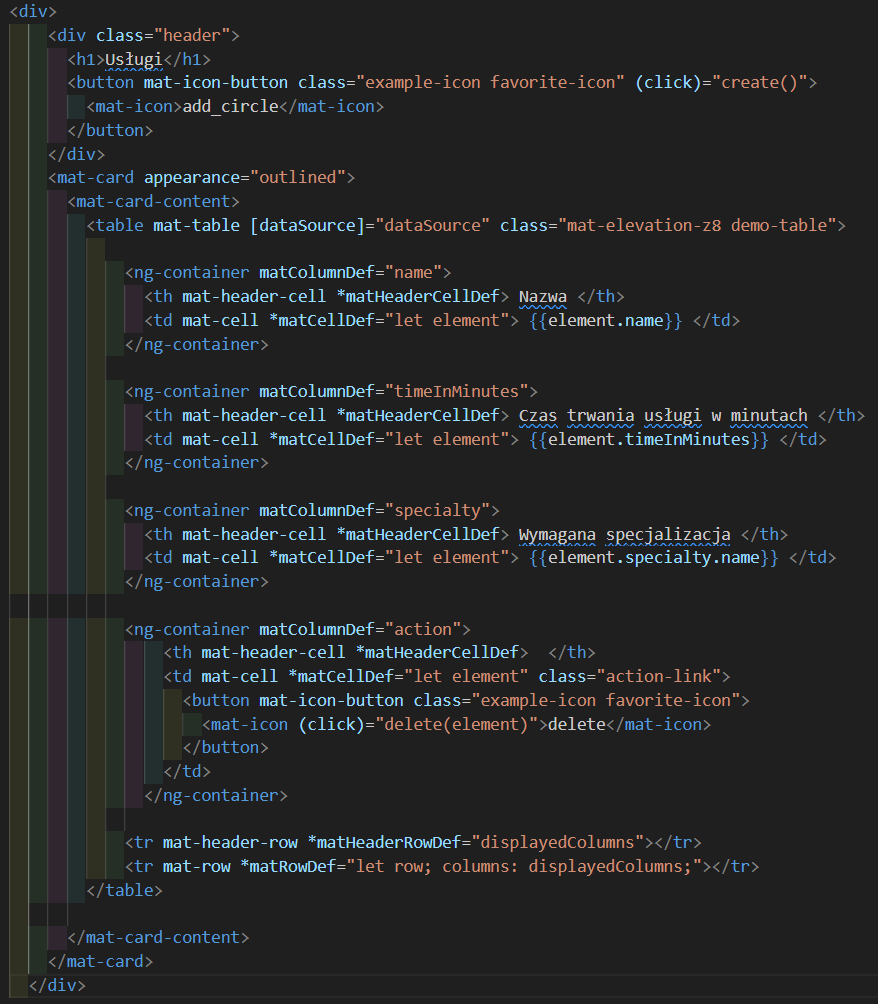
Rys. 3.39 Wygląd podstrony Usługi (opracowanie własne)

Oprócz tabeli zawiera ona również przyciski do usunięcia usługi oraz dodania nowej. Po naciśnięciu przycisku w kształcie plusa wyświetlone zostaje nowe okno służące do tworzenia nowej usługi. Wygląd tego okna został przedstawiony na rysunku nr 3.40.

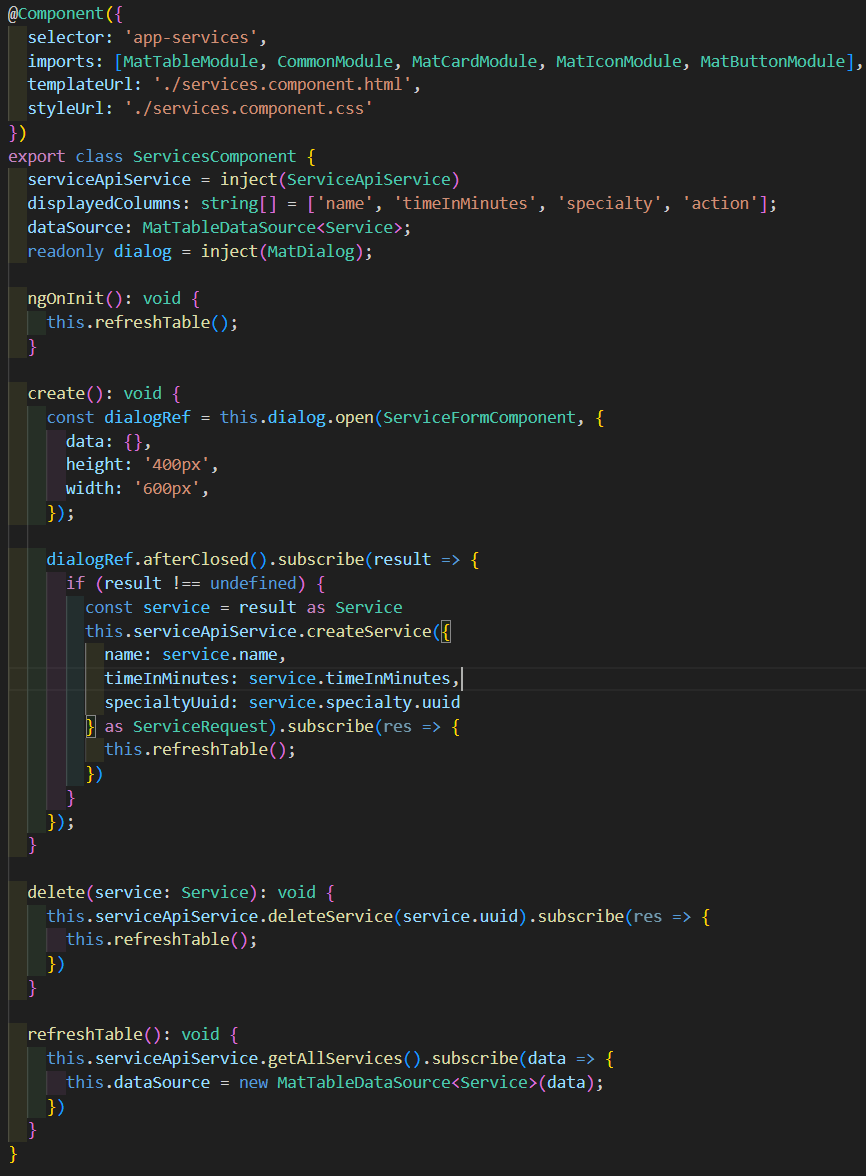


Rys. 3.40 Wygląd okna umożliwiającego dodanie nowej usługi medycznej (opracowanie własne)

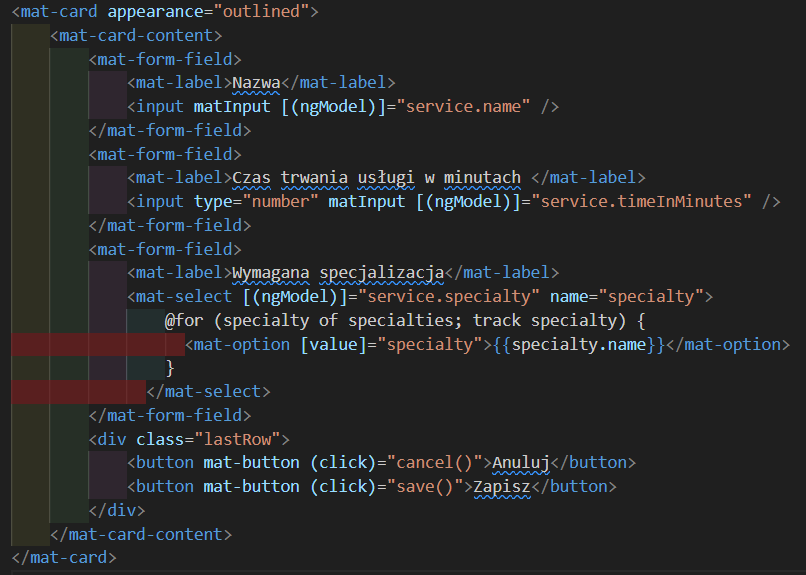
Kod źródłowy pokazany na rysunku nr 3.41 definiuje warstwę prezentacji podstrony Usługi. Natomiast rysunek nr 3.42 przedstawia kod źródłowy dla warstwy logiki. Funkcja o nazwie *create* jest wywoływana, gdy zostaje naciśnięty przycisk w kształcie plusa, i tworzy nowe okno pozwalające na dodanie nowej usługi. Kod źródłowy tego też okna pokazany został na rysunku nr 3.43 oraz 3.44 odpowiednio dla warstwy prezentacji i logiki.



Rys. 3.41 Kod źródłowy warstwy prezentacji podstrony Usługi (opracowanie własne)



Rys. 3.42 Kod źródłowy warstwy logiki podstrony Usługi (opracowanie własne)



Rys. 3.43 Kod źródłowy warstwy prezentacji okna do dodawania nowych usług (opracowanie własne)

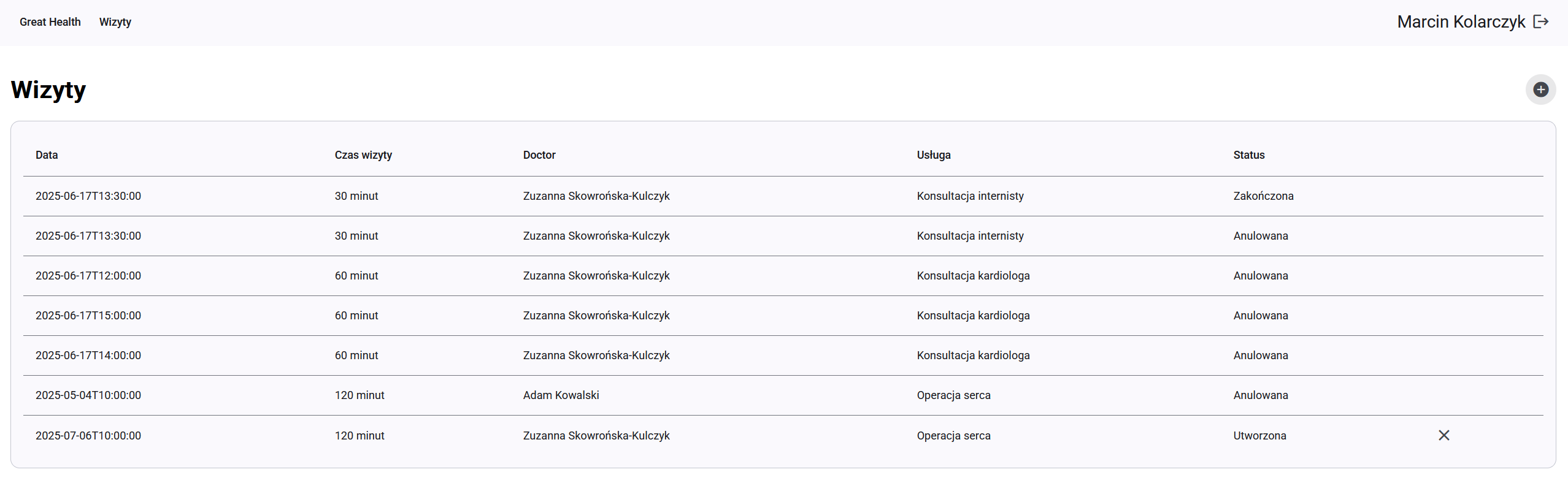


Rys. 3.44 Kod źródłowy warstwy logiki okna do dodawania nowych usług (opracowanie własne)

Charakterystycznym punktem dla kodu źródłowego okna do dodawania nowych usług jest pobranie oraz wyświetlenie dostępnych specjalizacji. W warstwie logiki, funkcja o nazwie *ngOnInit*, która wykonywana jest, gdy okno jest tworzone, pobiera wszystkie specjalizacje z backendu i przypisuje do zmiennej o nazwie *specialties*. Następnie w warstwie prezentacji, zmienna ta wyświetlana jest w postaci listy rozwijanej jednokrotnego wyboru utworzonej przy pomocy komponentu z *Angular Material* o nazwie *mat-select*.

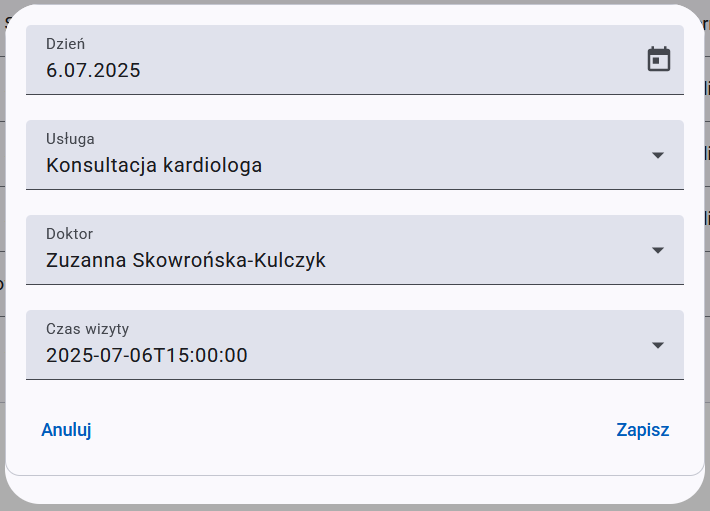
## Podstrona Wizyty

Podstrona Wizyty dostępna jest tylko i wyłącznie dla użytkownika będącego pacjentem. Umożliwia ona umówienie się na wizytę z lekarzem. Na stronie możliwe jest przeglądanie wizyt, anulowanie ich oraz umówienie nowych. Na rysunku nr 3.45 przedstawiony jest wygląd postronny Wizyty. W celu anulowania istniejącej wizyty użytkownik musi nacisnąć na przycisk w formie krzyżyka przy wybranej wizycie. Wówczas jej status zostanie zmieniony na *Anulowana*, co zostanie od razu odzwierciedlone na tabeli.



Rys. 3.45 Wygląd podstrony Wizyty (opracowanie własne)

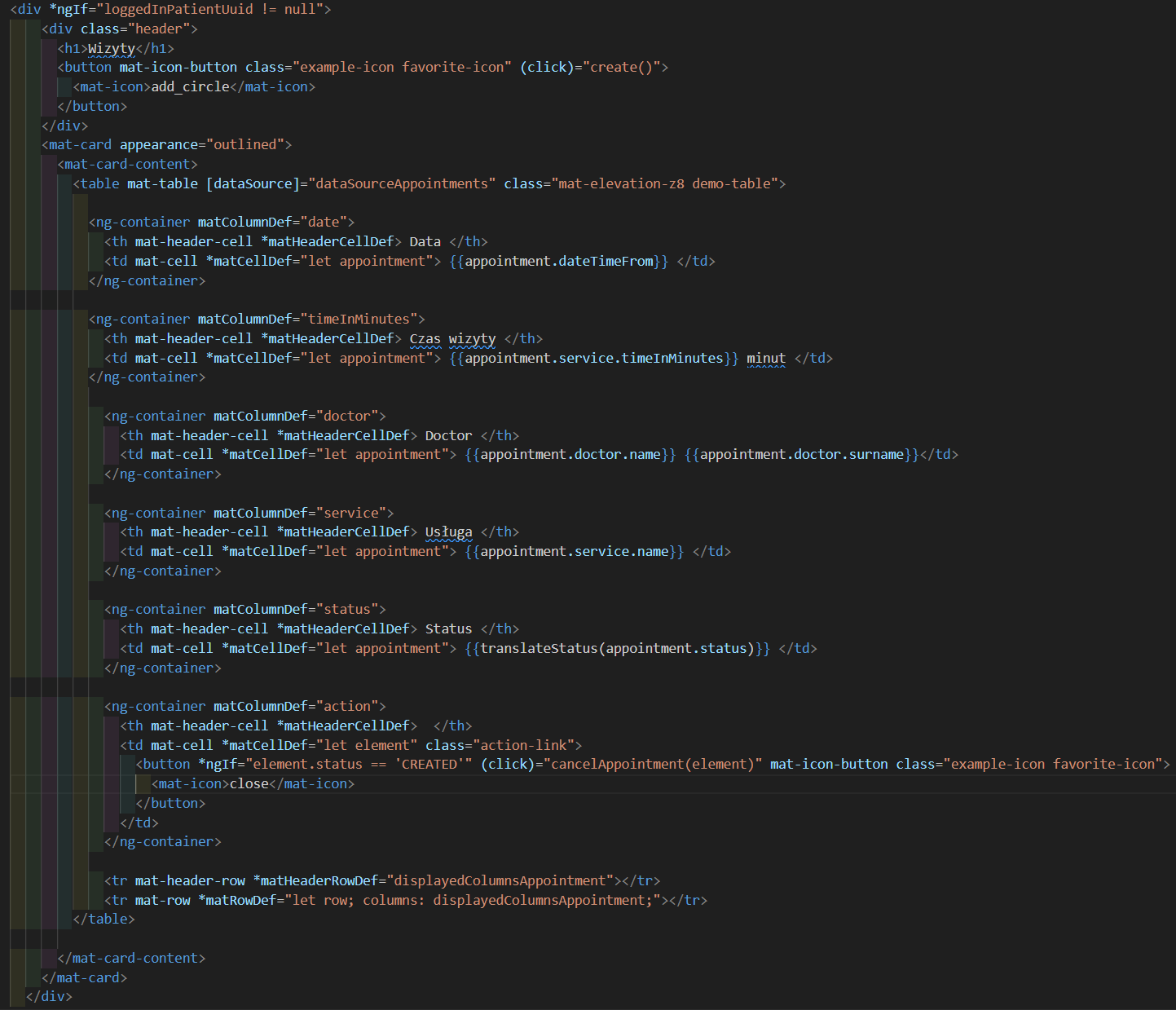
W celu utworzenia nowej wizyty, użytkownik musi nacisnąć na przycisk w formie plusa w prawym górnym rogu. Po jego naciśnięciu wyświetlane jest nowe okna służące do tworzenia nowych wizyt. Wygląd tego okna został zaprezentowany na rysunku nr 3.46.



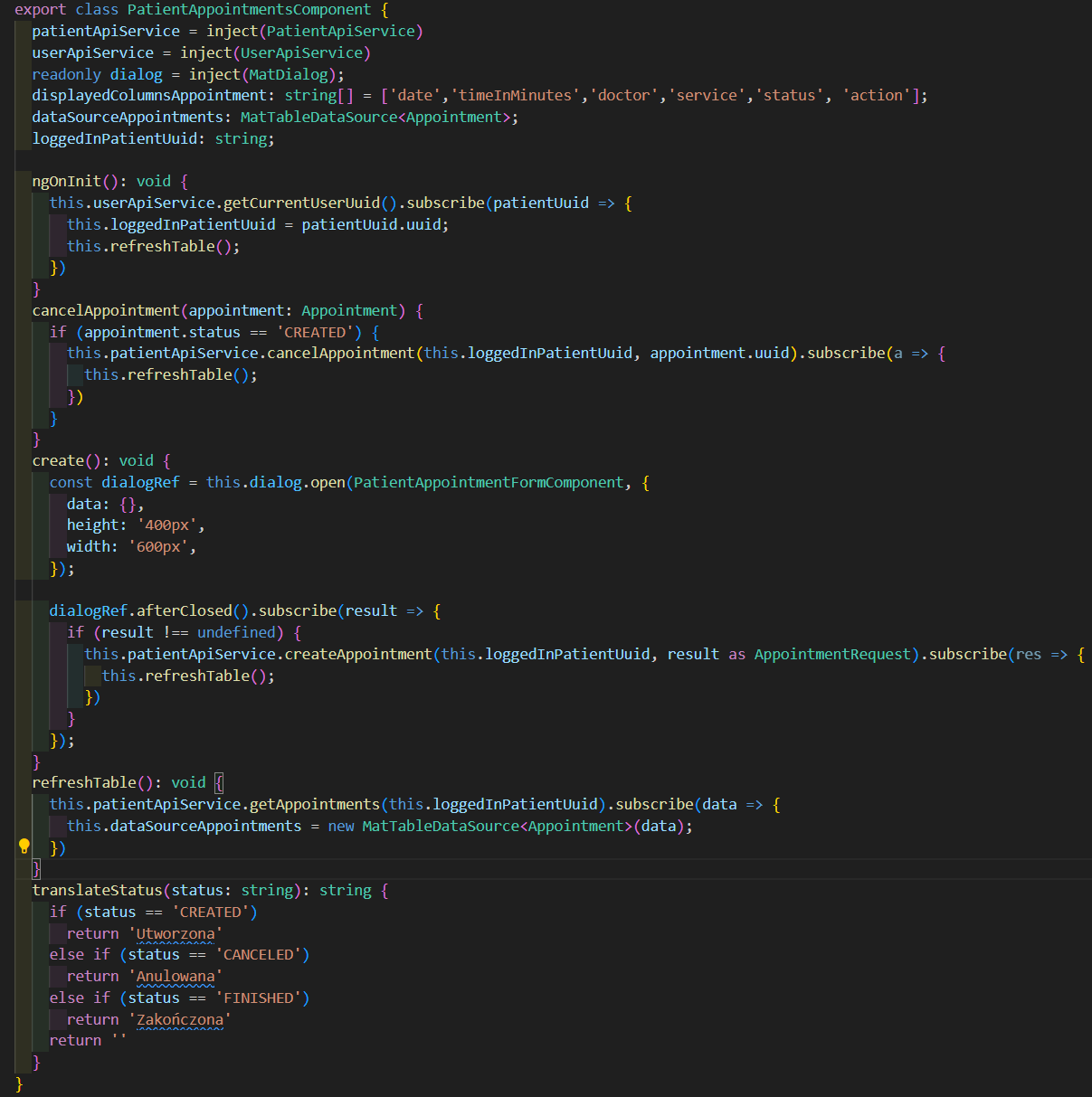
Rys. 3.46 Wygląd okna służącego do umawiania się na wizytę z lekarzem (opracowanie własne)

W celu umówienia wizyty, należy najpierw określić dzień, w którym chciałoby się mieć wizytę. Następnie należy wybrać usługę medyczną. Kiedy usługa zostanie wybrana, kolejne pole będące listą rozwijaną dostępnych lekarzy, zostaje zaktualizowane tak, aby wyświetlani byli tylko i wyłącznie doktorzy posiadający wymaganą specjalizację do wykonania wybranej usługi. Ostatnim polem jest godzina wizyty, która wyświetlana jest po wybraniu lekarza. Na podstawie długości wykonywania wybranej usługi oraz dostępności lekarza tworzone są możliwe terminy i wyświetlane w postaci listy rozwijanej jednokrotnego wyboru.

Na rysunkach nr 3.47 oraz 3.48 przedstawiony został kod źródłowy dla kolejno warstwy prezentacji oraz warstwy logiki dla podstrony *Wizyty.*



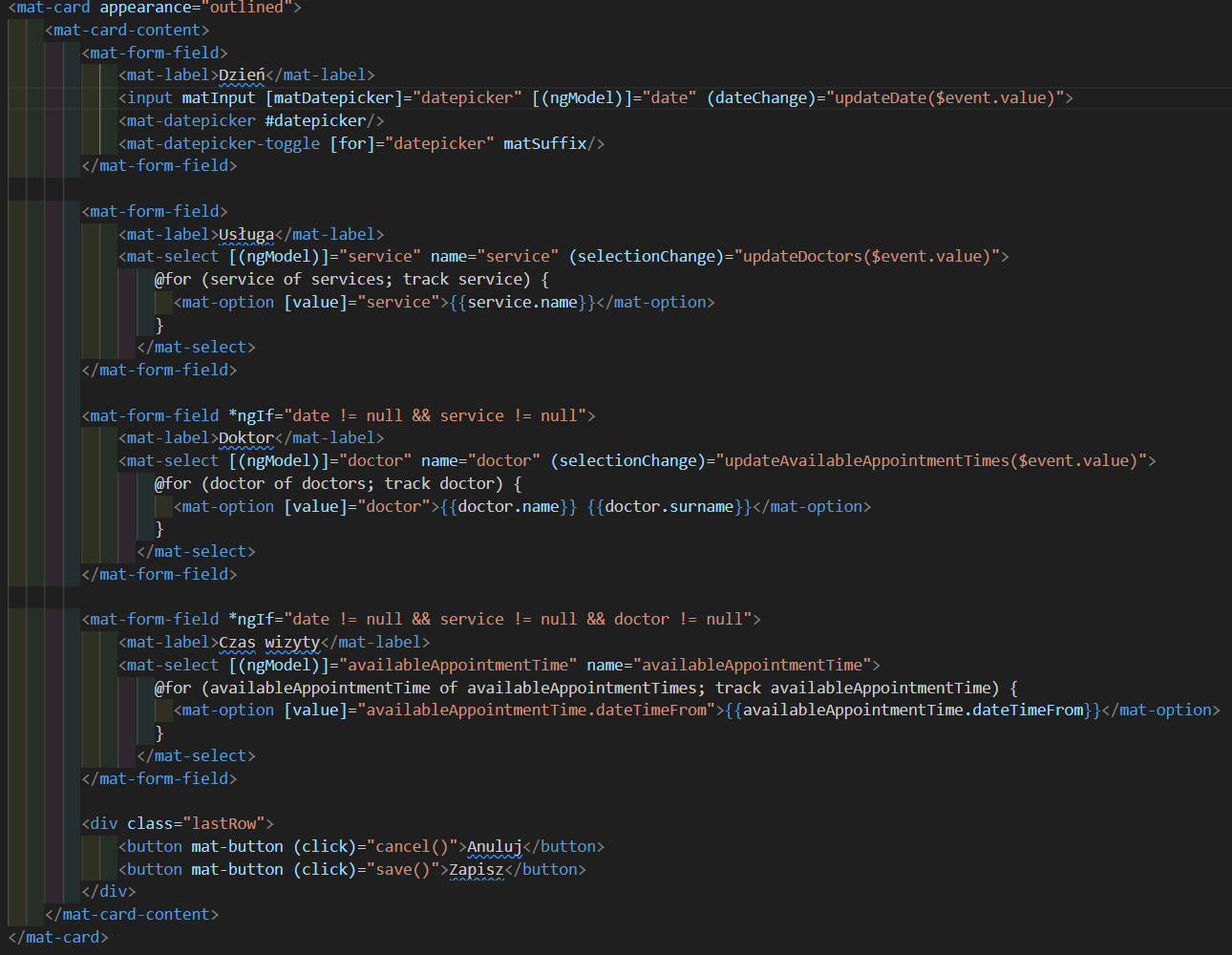
Rys. 3.47 Kod źródłowy warstwy prezentacji podstrony Wizyty (opracowanie własne)



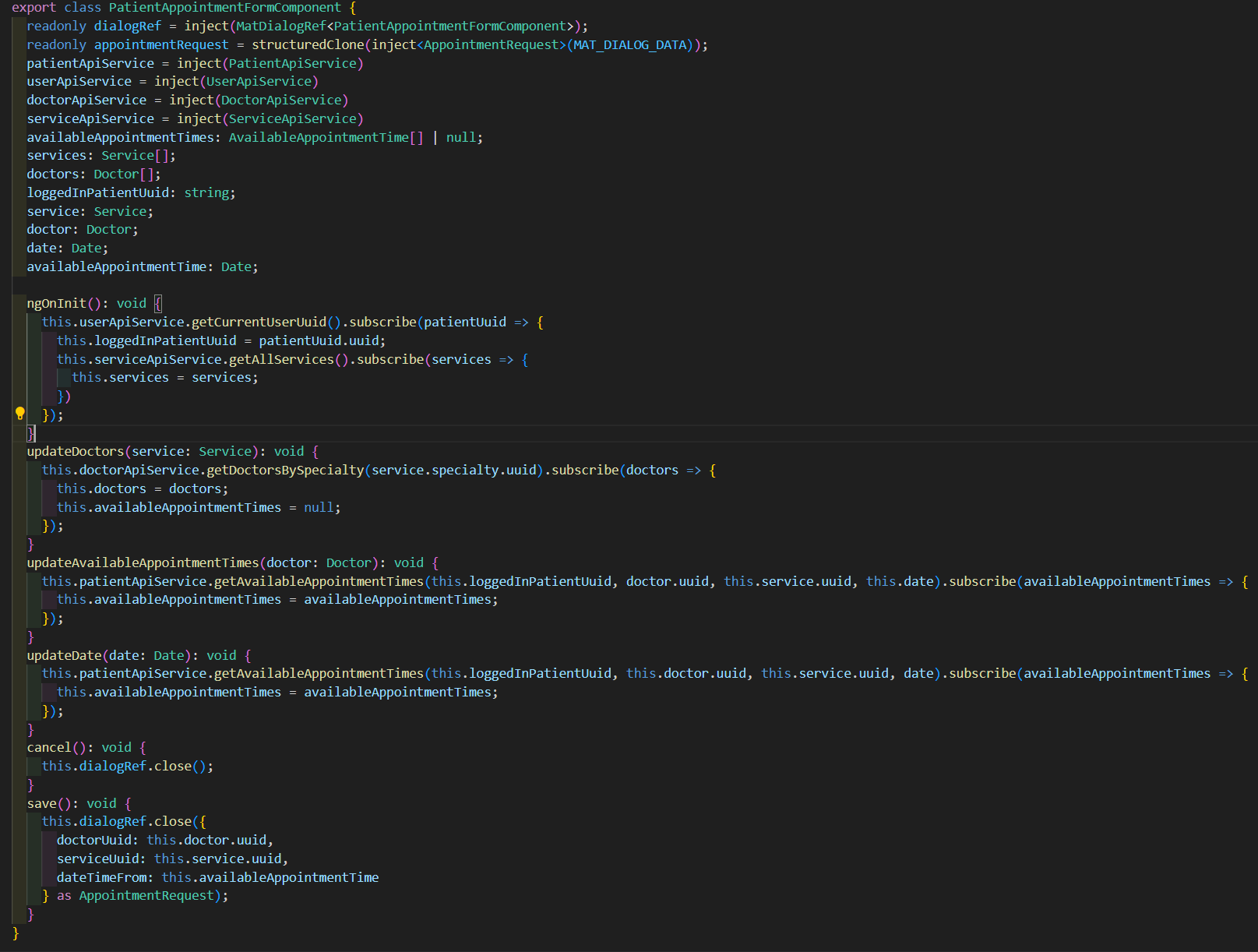
Rys. 3.48 Kod źródłowy warstwy logiki podstrony Wizyty (opracowanie własne)

Warstwa prezentacji definiuje tabelę wyświetlającą dane wizyty zalogowanej użytkownika. Ponadto, wyświetla dwa przyciski, gdzie pierwszy służy do anulowania wizyty, co skutkuje wywołaniem funkcji *cancelAppointment*, zaś drugi służy do umawiania nowej wizyty, co z kolei skutkuje wywołaniem funkcji *create*. Funkcja ta tworzy nowe okno, które umożliwia umówienie nowej wizyty, z wykorzystaniem serwisu z frameworka *Angular Material* o nazwie *MatDialog*. Po zamknięciu okna, komponent będący tym oknem, przekazuje dane nowo utworzonej wizyty, które to wówczas są wysyłane do backendu w celu zapisania jej w bazie danych.

Kod źródłowy warstwy prezentacji oraz logiki dla okna służącego do tworzenia nowych wizyty został pokazany kolejno na rysunku nr 3.49 oraz 3.50. Po wybraniu dnia wizyty oraz usługi na formularzu zostaje wywołana kolejno funkcja *updateDate* oraz *updateDoctors*. Funkcja *updateDate* aktualizuje listę dostępnych terminów na wybrany dzień. Natomiast funkcja *updateDoctors* aktualizuje listę lekarzy. Kiedy wybrane zostaną zarówno data wizyty oraz usługa, kolejne pole zostaje wyświetlone poprzez spełnienie warunku zdefiniowanego za pomocą Angularowego atrybutu *ngIf*. Pole *Doktor* jest listą jednokrotnego wyboru wyświetlającą dostępnych lekarzy. Gdy lekarz zostaje wybrany, wywoływana zostaje funkcja *updateAvailableAppointmentTimes*, która aktualizuje listę dostępnych terminów na wybrany dzień z wybranym lekarzem. Wówczas formularz wyświetla ostatnie pole będące listą rozwijaną z dostępnymi terminami. Po wybraniu terminu użytkownik może nacisnąć na przycisk *Zapisz*, co skutkować będzie zamknięciem okna i przekazaniem danych do podstrony *Wizyty* w celu jej zapisania w bazie danych.

**

Rys. 3.49 Kod źródłowy warstwy prezentacji okna do dodawania nowych wizyt (opracowanie własne)



Rys. 3.50 Kod źródłowy warstwy logiki okna do dodawania nowych wizyt (opracowanie własne)

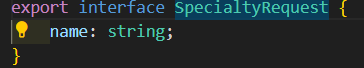
## Klient API Backendu

Klient API Backendu zdefiniowany jest jako zbiór serwisów Angularowych, których nazwy kończą się z postfiksem *ApiService*. Przykładowym tego typu serwisem jest *SpecialtyApiService*, który został przedstawiony na rysunku nr 3.51.



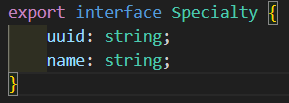
Rys. 3.51 Kod źródlowy serwisu SpecialtyApiService (opracowanie własne)

Funkcje zdefiniowane w serwisie *SpecialtyApiService* odzwierciedlają endpointy zdefiniowane w backendzie. Funkcja *createSpecialty* wywołuje endpoint *POST* */api/specialties* przekazując payload w postaci interfejsu *SpecialtyRequest* w formacie JSON, który definiuje dane potrzebne do utworzenia nowej specjalizacji. Klasa *SpecialtyRequest* przedstawiona została na rysunku nr 3.52.



Rys. 3.52 Kod źródłowy interfejsu SpecialtyRequest (opracowanie własne)

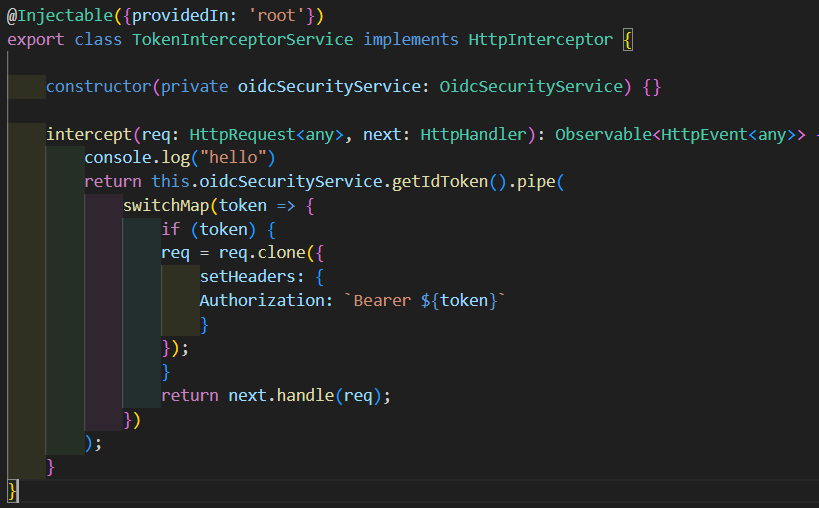
Serwis ten posiada również funkcję *getAllSpecialties* zwracającą wszystkie specjalizacje. Wywołuje ona endpoint po stronie backendu *GET /api/specialties* i zwraca listę obiektów interfejsu *Specialty*, której to klasy definicja zaprezentowana została na rysunku 3.53.



Rys. 3.53 Kod źródłowy interfejsu Specialty (opracowanie własne)

Oprócz wyżej wymienionych istnieją również funkcja *deleteSpecialty* reprezentująca endpoint *DELETE /api/specialties/{specialtyId}* oraz funkcja *getSpecialtyById* odzwierciedlająca endpoint *GET /api/specialties/{specialtyId}*. Funkcje te wykorzystują parametr *specialtyId*, czyli identyfikator specjalizacji w celu wykonania kolejno usunięcia lub zwrotu specjalizacji.

Warto również wspomnieć o uwierzytelnieniu, które następuje poprzez przekazanie tokena JWT zalogowanego użytkownika poprzez header *Authorization* dodawany do każdego z żądań. Za realizację tego zadania odpowiedzialny jest serwis o nazwie *TokenInterceptorService*, którego definicja została przedstawiona na rysunku nr 3.54.



Rys. 3.54 Kod źródłowy serwisu TokenInterceptorService (opracowanie własne)

## Implementacja backendu

Backend napisany jest przy użyciu frameworka Spring w języku Java 17. Najpierw zostanie omówiona konfiguracja aplikacji.

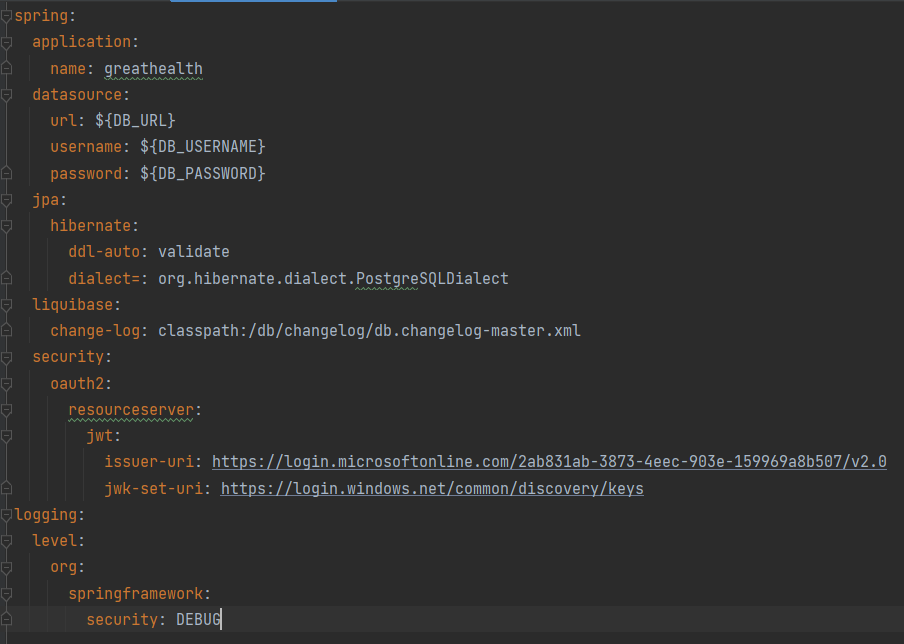
## Konfiguracja

Zarządzanie zależnościami backendu realizowane jest przy użyciu *Gradle* w pliku o nazwie *build.gradle*, co zostało zaprezentowanie na rysunku nr 3.54.



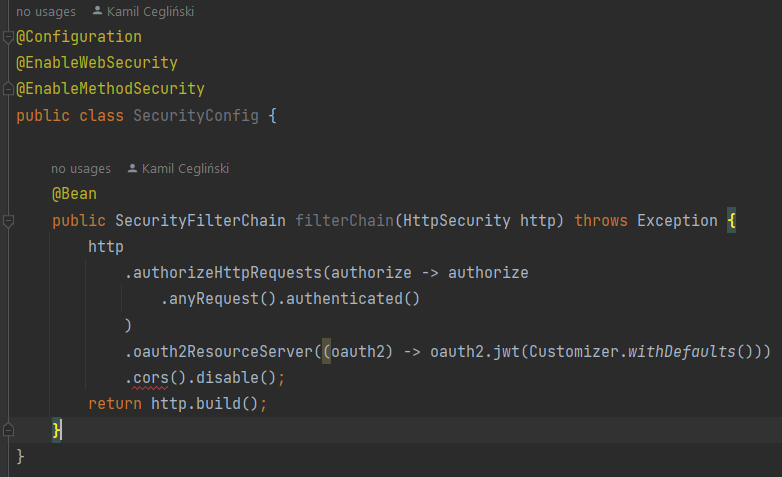
Rys. 3.55 Zarządzanie zależnościami backendu - plik build.gradle (opracowanie własne)

Natomiast globalna konfiguracja projektu znajduje się w pliku *application.yml*, co zostało pokazane na rysunku nr 3.56.



Rys. 3.56 Globalna konfiguracja backendu – plik application.yml (opracowanie własne)

Można zauważyć, że w konfiguracji w *application.yml* zostały zdefiniowane parametry takie jak dane uwierzytelniające do bazy danych, typ bazy danych, wskazanie lokalizacji skryptów *Liquibase* zarządzających bazą danych oraz parametrów dotyczących zabezpieczenia REST API za pomocą *Microsoft Entra ID*. Dodatkowa konfiguracja, ale zapisana już w języku Java znajduje się w pliku *SecurityConfig*, co zostało zaprezentowane na rysunku nr 3.57   
i wymusza uwierzytelnienie klienta API dla wszystkich endpointów REST API.

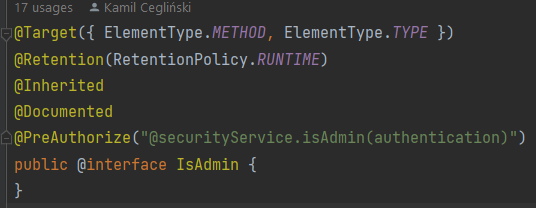


Rys. 3.57 Konfiguracja SecurityConfig.java (opracowanie własne)

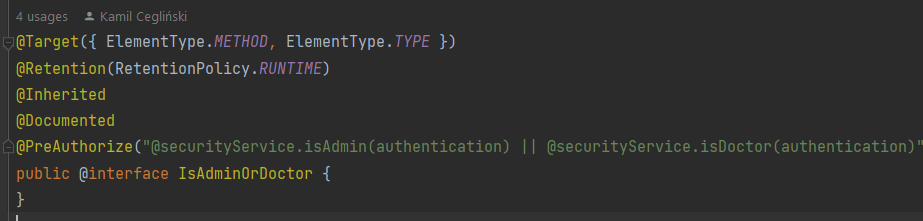
Serwis *SecurityService* znajdujący się na rysunku 3.58 wraz z pomocniczymi adnotacjami zaprezentowanymi na rysunkach 3.59-3.64 pomagają w autoryzacji, czyli decydowaniu czy klient API ma dostęp do danego endpointu i danych, o które prosi.



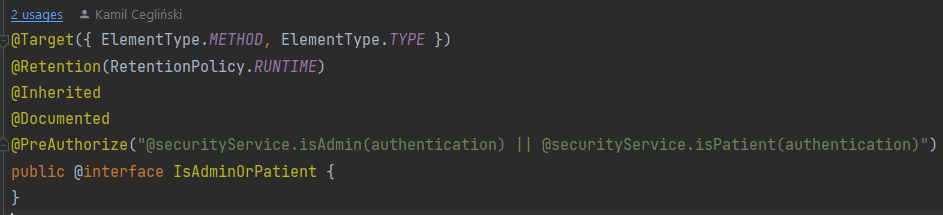
Rys. 3.58 Serwis SecurityService.java (opracowanie własne)



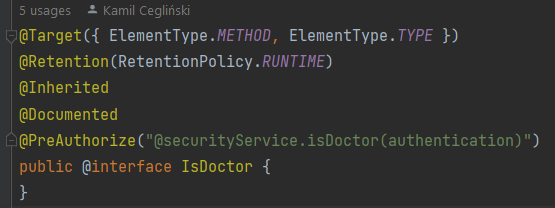
Rys. 3.59 Adnotacja IsAdmin (opracowanie własne)



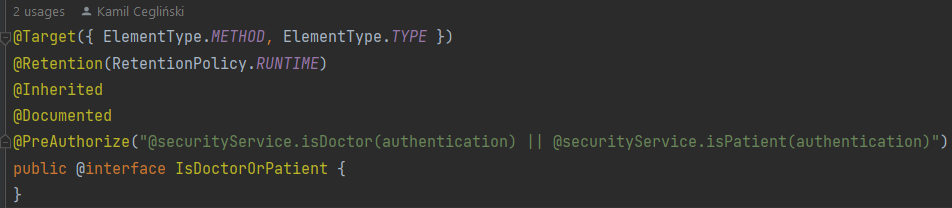
Rys. 3.60 Adnotacja IsAdminOrDoctor (opracowanie własne)



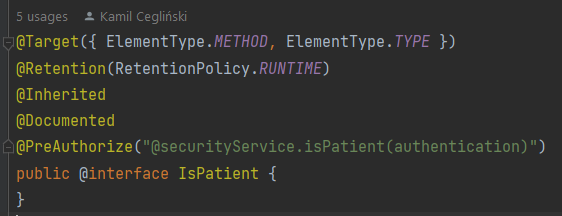
Rys. 3.61 Adnotacja IsAdminOrPatient (opracowanie własne)



Rys. 3.62 Adnotacja IsDoctor (opracowanie własne)



Rys. 3.63 Adnotacja IsDoctorOrPatient (opracowanie własne)



Rys. 3.64 Adnotacja IsPatient (opracowanie własne)

Warto również wspomnieć o skryptach *Liquibase*, które zapewniają wymagany schemat bazy danych. Rysunek 3.65 przedstawia część skryptu o nazwie *001\_init.xml*, który tworzy tabele bazodanowe oraz inne komponenty w bazie takie jak indeksy, które wymagane są, aby aplikacja prawidłowo działała.

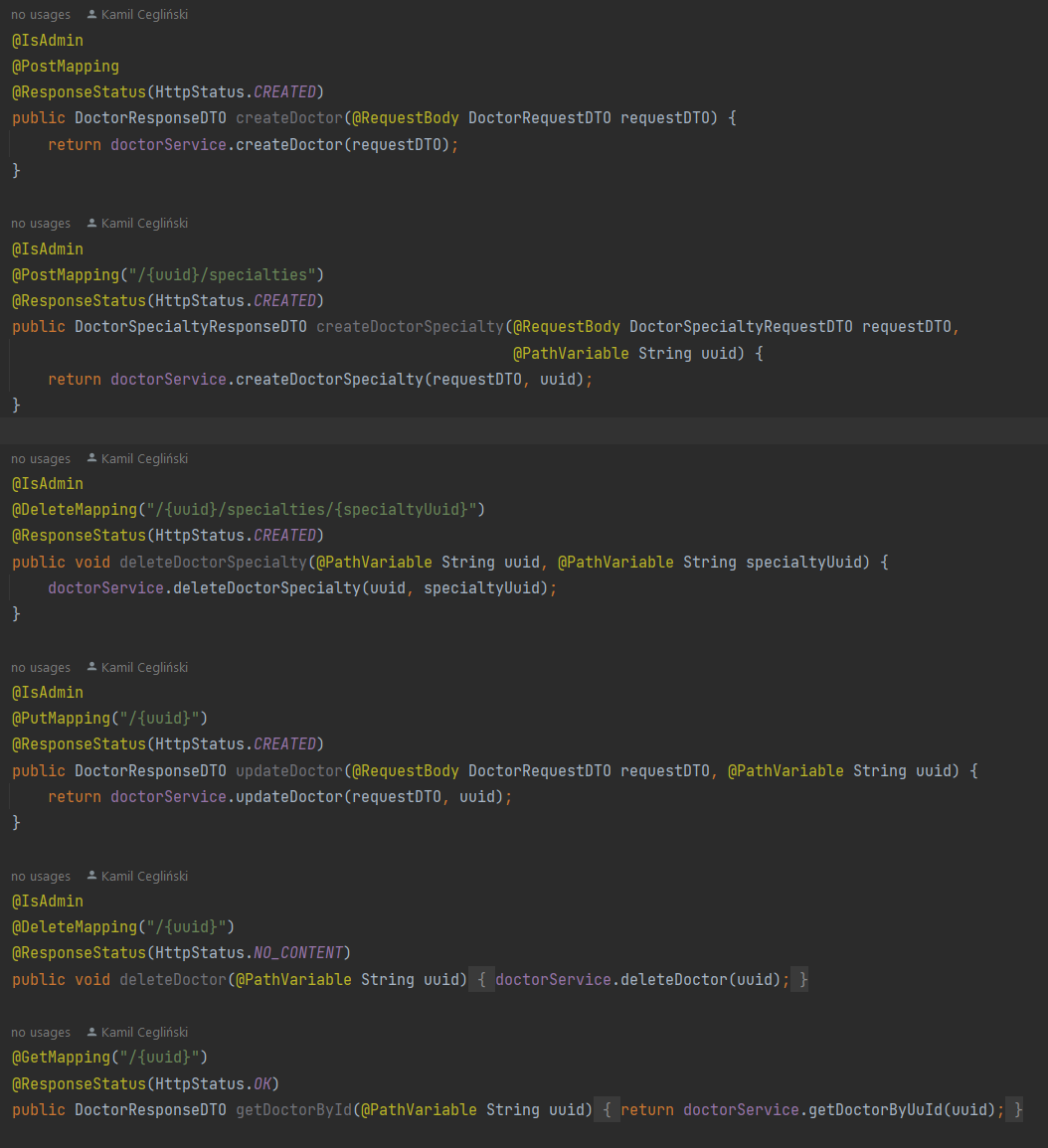


Rys. 3.65 Skrypt Liquibase 001\_init.xml (opracowanie własne)

## Omówienie REST API - /api/doctors

Klasa *DoctorController*, której część została przedstawiona na rysunku nr 3.63, definiuje następujące endpointy z przedrostkiem */api/doctors*:

* POST /api/doctors
* POST /api/doctors/{uuid}/specialties
* DELETE /api/doctors/{uuid}/specialties/{specialtyUuid}
* PUT /api/doctors/{uuid}
* DELETE /api/doctors/{uuid}
* GET /api/doctors/{uuid}
* GET /api/doctors
* GET /api/doctors/{uuid}/appointments
* DELETE /api/doctors/{uuid}/appointments/{appointmentUuid}



Rys. 3.66 Część kodu źródłowego klasy DoctorController (opracowanie własne)

## Omówienie REST API - /api/doctors/{doctorUuid}/availabilities

## Omówienie REST API - /api/patients

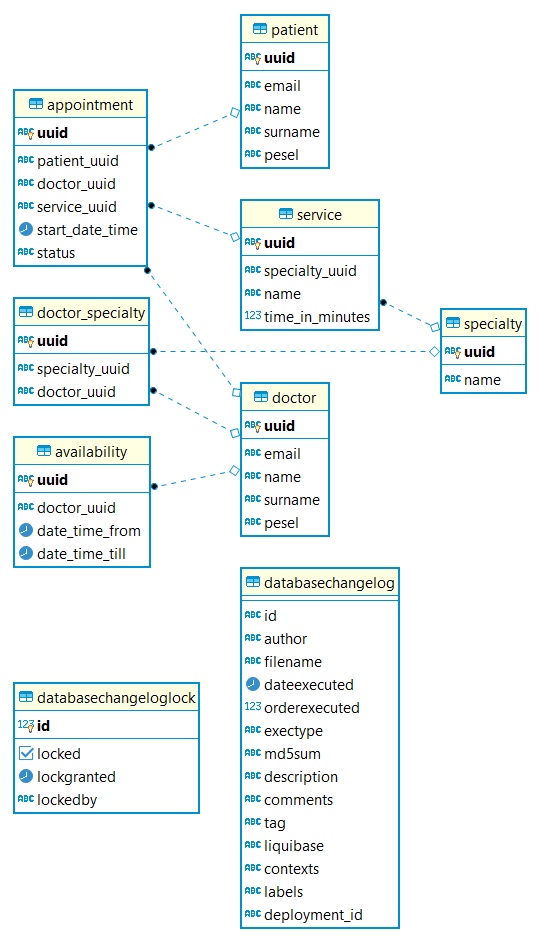
## Omówienie REST API - /api/services

## Omówienie REST API - /api/specialties

## Omówienie REST API - /api/users

## Schemat bazy danych

Skrypt *Liquibase* o nazwie *001\_init.xml,* którego część przedstawiona jest na rysunku 3.65, zawiera definicję schematu bazy danych. Rysunek 3.XX prezentuje schemat bazy, po wykonaniu tego też skryptu. Można zauważyć, że utworzone zostały tabele *patient, appointment, service, doctor\_specialty, specialty, doctor, availability, databasechangelog, databasechangeloglock*. Dwie ostatnie tabele są tabelami technicznymi wykorzystywanymi przez *Liquibase* w celu zapewnienia, że każdy skrypt wykonywany jest tylko raz w celu otrzymania zadanego stanu na bazie danych.



# WDROŻENIE NA CHMURĘ OBLICZENIOWĄ AZURE

## Microsoft Entra ID

1. Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 35 [↑](#footnote-ref-1)
2. Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 35 [↑](#footnote-ref-2)
3. Doug Rosenberg, Matt Stephens, Use Case Driven Object Modeling with UML. Theory and Practice. Appress, 2007, s. 25 [↑](#footnote-ref-3)
4. Doug Rosenberg, Matt Stephens, Use Case Driven Object Modeling with UML. Theory and Practice. Appress, 2007, s. 26 [↑](#footnote-ref-4)
5. Doug Rosenberg, Matt Stephens, Use Case Driven Object Modeling with UML. Theory and Practice. Appress, 2007, s. 49 [↑](#footnote-ref-5)
6. Jarosław Żeliński, Analiza Biznesowa. Praktyczne modelowanie organizacji, Helion, 201, s. 74 [↑](#footnote-ref-6)
7. Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 172 [↑](#footnote-ref-7)
8. Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 179 [↑](#footnote-ref-8)
9. Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 261 [↑](#footnote-ref-9)
10. Pethuru Raj, Anupama Raman, Harihara Subramanian, Architectural Patterns, Packt Publishing, 2017, s. 58 [↑](#footnote-ref-10)
11. https://www.geeksforgeeks.org/frontend-vs-backend/ [↑](#footnote-ref-11)
12. Pethuru Raj, Anupama Raman, Harihara Subramanian, Architectural Patterns, Packt Publishing, 2017, s. 60 [↑](#footnote-ref-12)
13. https://medium.com/buildingminds-technologies/importance-of-collaboration-between-frontend-and-backend-teams-4b05e8fd29f9 [↑](#footnote-ref-13)
14. https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce [↑](#footnote-ref-14)
15. https://learn.microsoft.com/en-us/entra/identity-platform/access-token-claims-reference [↑](#footnote-ref-15)
16. https://learn.microsoft.com/en-us/entra/identity-platform/security-tokens#token-endpoints-and-issuers [↑](#footnote-ref-16)
17. https://jwt.io/introduction [↑](#footnote-ref-17)
18. Avijit Mallik, Mhia Md. Zaglul Shahadat, Jia-Chi Tsou, Abid Ahsan, Man-in-the-middle-attack: Understanding in simple words, International Journal of Data and Network Science, artykuł na Research Gate, s. 80 [↑](#footnote-ref-18)
19. https://www.rfc-editor.org/rfc/rfc2818 [↑](#footnote-ref-19)
20. https://www.postgresql.org/docs/current/ssl-tcp.html [↑](#footnote-ref-20)
21. Ivan Ristić, Bulletproof SSL and TLS, Feisty Duck Limited, s. 25 [↑](#footnote-ref-21)
22. https://auth0.com/blog/the-tls-handshake-explained/ [↑](#footnote-ref-22)
23. https://angular.dev/ [↑](#footnote-ref-23)
24. https://www.typescriptlang.org/ [↑](#footnote-ref-24)
25. https://developer.mozilla.org/en-US/docs/Web/CSS [↑](#footnote-ref-25)
26. https://developer.mozilla.org/en-US/docs/Web/HTML [↑](#footnote-ref-26)
27. https://udigroup.pl/blog/jezyk-html-co-to-jest-do-czego-sluzy-jak-wyglada/ [↑](#footnote-ref-27)
28. https://webporadnik.pl/css-co-to-jest-i-jak-dziala-css-podstawowe-i-najwazniejsze-informacje-o-css/ [↑](#footnote-ref-28)
29. https://www.typescriptlang.org/ [↑](#footnote-ref-29)
30. https://www.w3schools.com/Js/ [↑](#footnote-ref-30)
31. https://angular.dev/overview [↑](#footnote-ref-31)
32. https://material.angular.dev/ [↑](#footnote-ref-32)
33. https://poradnikinzyniera.pl/rest-api-co-to-jest-jak-dziala-i-jak-z-niego-korzystac/ [↑](#footnote-ref-33)
34. https://restfulapi.net/idempotent-rest-apis/ [↑](#footnote-ref-34)
35. https://www.oracle.com/pl/java/ [↑](#footnote-ref-35)
36. Cay S. Horstmann, Java Podstawy. Wydanie XI, Helion, 2019, s. 25 [↑](#footnote-ref-36)
37. https://www.oracle.com/webfolder/technetwork/Tutorials/obe/java/gc01/index.html [↑](#footnote-ref-37)
38. https://spring.io/projects/spring-framework [↑](#footnote-ref-38)
39. https://spring.io/projects/spring-framework [↑](#footnote-ref-39)
40. https://docs.spring.io/spring-framework/reference/web/webmvc.html [↑](#footnote-ref-40)
41. https://spring.io/projects/spring-data [↑](#footnote-ref-41)
42. https://spring.io/projects/spring-boot [↑](#footnote-ref-42)
43. https://docs.spring.io/spring-framework/reference/core/beans.html [↑](#footnote-ref-43)
44. https://docs.spring.io/spring-framework/reference/core/beans/definition.html [↑](#footnote-ref-44)
45. https://gradle.org/ [↑](#footnote-ref-45)
46. https://www.liquibase.com/ [↑](#footnote-ref-46)
47. https://projectlombok.org/ [↑](#footnote-ref-47)
48. https://projectlombok.org/features/GetterSetter [↑](#footnote-ref-48)
49. https://junit.org/junit4/ [↑](#footnote-ref-49)
50. https://www.postgresql.org/ [↑](#footnote-ref-50)
51. https://www.geeksforgeeks.org/what-is-sql/ [↑](#footnote-ref-51)