



AKADEMIA WIT
w WARSZAWIE

WYDZIAŁ INFORMATYKI I GRAFIKI

KIERUNEK INFORMATYKA

**PRACA DYPLOMOWA
INŻYNIERSKA**

Kamil Cegliński

Implementacja systemu do rezerwacji wizyt pacjentów w przychodni na platformie Azure

Promotor pracy:

Dr inż. Jarosław Sikorski

WARSZAWA, rok akademicki 2024/2025

—

—

OŚWIADCZENIE AUTORA PRACY

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami prawa, w szczególności z ustawą z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 1994 r. Nr 24, poz. 83 – tekst pierwotny i Dz. U. z 2000 r. Nr 80, poz. 904 – tekst jednolity, z późn. zm.). Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w szkole wyższej. Oświadczam także, że wersja pracy składana w dziekanacie, pliki elektroniczne pracy zawierają identyczną treść. Oświadczam również, że wszystkie narzędzia informatyczne zastosowane do wykonania niniejszej pracy wykorzystałem zgodnie z obowiązującymi przepisami prawa w zakresie ochrony własności intelektualnej i przemysłowej. Wyrażam zgodę na przetwarzanie następujących danych osobowych: imiona, nazwisko, nr albumu, ukończony kierunek, specjalność oraz rok, system i rodzaj studiów, wynik ukończenia, miejsce urodzenia w celu wydrukowania dyplomu na zlecenie uczelni przez drukarnię ZPW Bogucin k/Poznania z siedzibą w Kobylnicy, ul. Gnieźnieńska 127 zgodnie z Rozporządzeniem Parlamentu Europejskiego i Rady Unii Europejskiej 2016/679 z dnia 27 kwietnia 2016 r. w sprawie ochrony osób fizycznych w związku z przetwarzaniem danych osobowych i w sprawie swobodnego przepływu takich danych oraz uchylenia dyrektywy 95/46/WE (RODO).

Wyrażam zgodę na przetwarzanie następujących danych osobowych: imiona, nazwisko, nr albumu, ukończony kierunek, specjalność oraz rok, system i rodzaj studiów, wynik ukończenia, miejsce urodzenia w celu wydrukowania dyplomu na zlecenie uczelni przez drukarnię ZPW Bogucin k/Poznania z siedzibą w Kobylnicy, ul. Gnieźnieńska 127 zgodnie z Rozporządzeniem Parlamentu Europejskiego i Rady Unii Europejskiej 2016/679 z dnia 27 kwietnia 2016 r. w sprawie ochrony osób fizycznych w związku z przetwarzaniem danych osobowych i w sprawie swobodnego przepływu takich danych oraz uchylenia dyrektywy 95/46/WE (RODO).

20/07/2025

(data)

17694

(album)

Kamil Cegliński

(student)

.....
(podpis studenta)

Spis treści

WSTĘP	9
1. ANALIZA WYMAGAŃ	10
1.1. Wymagania biznesowe	10
1.2. Diagram modeli domenowych	12
1.3. Diagram przypadków użycia	14
1.4. Diagram klas.....	16
2. ARCHITEKTURA SYSTEMU.....	18
2.1. Zarys architektury.....	18
2.2. Bezpieczeństwo systemu	20
2.2.1. Uwierzytelnienie przy użyciu Microsoft Entra ID	20
2.2.2. Szyfrowane protokoły komunikacyjne przy użyciu TLS	23
2.3. Frontend.....	25
2.3.1. HTML.....	25
2.3.2. CSS	26
2.3.3. TypeScript.....	27
2.3.4. Angular	27
2.4. Backend	29
2.4.1. REST API	29
2.4.2. Java SE	30
2.4.3. Spring	30
2.4.4. Gradle	31
2.4.5. Liquibase	31
2.4.6. Lombok.....	31
2.4.7. JUnit 4	31
2.5. Baza danych.....	32
3. IMPLEMENTACJA SYSTEMU.....	33
3.1. Implementacja frontendu	33
3.1.1. Konfiguracja	33
3.1.2. Podstrona Wizyty z pacjentami	38
3.1.3. Podstrona Dostępność	40
3.1.4. Podstrona Lekarze	44
3.1.5. Podstrona Pacjenci.....	51
3.1.6. Podstrona Specjalizacje	53

3.1.7.	Podstrona Usługi	55
3.1.8.	Podstrona Wizyty.....	59
3.1.9.	Klient API Backendu	63
3.2.	Implementacja backendu	65
3.2.1.	Konfiguracja	65
3.2.2.	Schemat bazy danych	69
3.2.3.	Omówienie REST API - /api/doctors	70
3.2.4.	Omówienie REST API - /api/doctors/{doctorUuid}/availabilities.....	72
3.2.5.	Omówienie REST API - /api/patients.....	73
3.2.6.	Omówienie REST API - /api/services	76
3.2.7.	Omówienie REST API - /api/specialties	78
3.2.8.	Omówienie REST API - /api/users	79
4.	WDROŻENIE NA CHMURE OBLICZENIOWĄ AZURE	81
4.1.	Microsoft Entra ID	81
4.2.	Docker	82
4.3.	Azure Database for PostgreSQL	83
4.4.	App Service	84
4.5.	GitHub Actions.....	85
	ZAKOŃCZENIE	88
	BIBLIOGRAFIA	89
	Książki	89
	Strony internetowe	89
	WYKAZ RYSUNKÓW	93
	STRESZCZENIE PRACY	98
	Streszczenie w języku polskim	98
	Streszczenie w języku angielskim	98

WSTĘP

Celem pracy jest zaprojektowanie, implementacja oraz wdrożenie systemu do rezerwacji wizyt w fikcyjnej przychodni przy użyciu usług chmury obliczeniowej Azure. W ramach projektu utworzone zostały aplikacja frontendowa typu SPA oraz serwis Javowy wystawiający REST API. Obie aplikacje zostały wdrożone na chmurę Azure stosując przy tym skrypty wdrożeniowe CI/CD.

Praca złożona jest z czterech rozdziałów. Pierwszy z nich skupia się na sformułowaniu wymagań biznesowych oraz ich analizie. Tworzone są diagramy UML takie jak diagram modeli domenowych, diagram przypadków użycia oraz diagram klas. Drugi rozdział omawia architekturę systemu. Opisane zostają poszczególne technologie wykorzystane w projekcie oraz w jaki sposób są one ze sobą powiązane. Ponadto, następuje podział systemu na dwie warstwy – warstwę wizualną tj. frontend oraz warstwę logiki biznesowej tj. backend. Obie warstwy stanowią odrębne aplikacje komunikujące się ze sobą. Trzeci rozdział skupia się na implementacji tychże aplikacji. Omówiona zostaje szczegółowo zarówno implementacja frontendu, jak i backendu w osobnych podrozdziałach. Dużą wagę przyłożono do konfiguracji oraz najważniejszych aspektów aplikacji takich jak logika samej rezerwacji wizyty w przychodni oraz zapewnienie, że wizyty pacjentów nie nakładają się na siebie. Ponadto, omówione zostały wszystkie podstrony oraz endpointy systemu. Ostatni rozdział prezentuje konfigurację wdrożenia na chmurę obliczeniową Azure oraz skrypty wdrożeniowe automatyzujące ten proces.

Praca została napisana w oparciu o własne wieloletnie doświadczenie branżowe z wykorzystaniem źródeł w postaci literatury oraz stron internetowych. Kody źródłowe zostały umieszczone na portalu GitHub¹.

¹ <https://github.com/oputk/wit-computer-science-engineering-thesis>

1. ANALIZA WYMAGAŃ

1.1. 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 biznesowych². 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. Recepcjonisci powinni tylko i wyłącznie weryfikować tożsamość oraz obecność pacjentów gotowych na wizytę, a następnie przekierowywać ich do konkretnego

² Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 35

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.

1.2. Diagram modeli domenowych

Na podstawie wyżej opisanych wymagań biznesowych przychodni, należy przełożyć je na wymagania użytkownika³. Najlepiej zrobić to w postaci diagramów przypadków użycia. Jednakże najpierw pomocnym jest zacząć od zdefiniowania modeli domenowych, które tworzyć będą system⁴. Tworząc taki diagram, warto trzymać się pewnych zasad. Są to⁵:

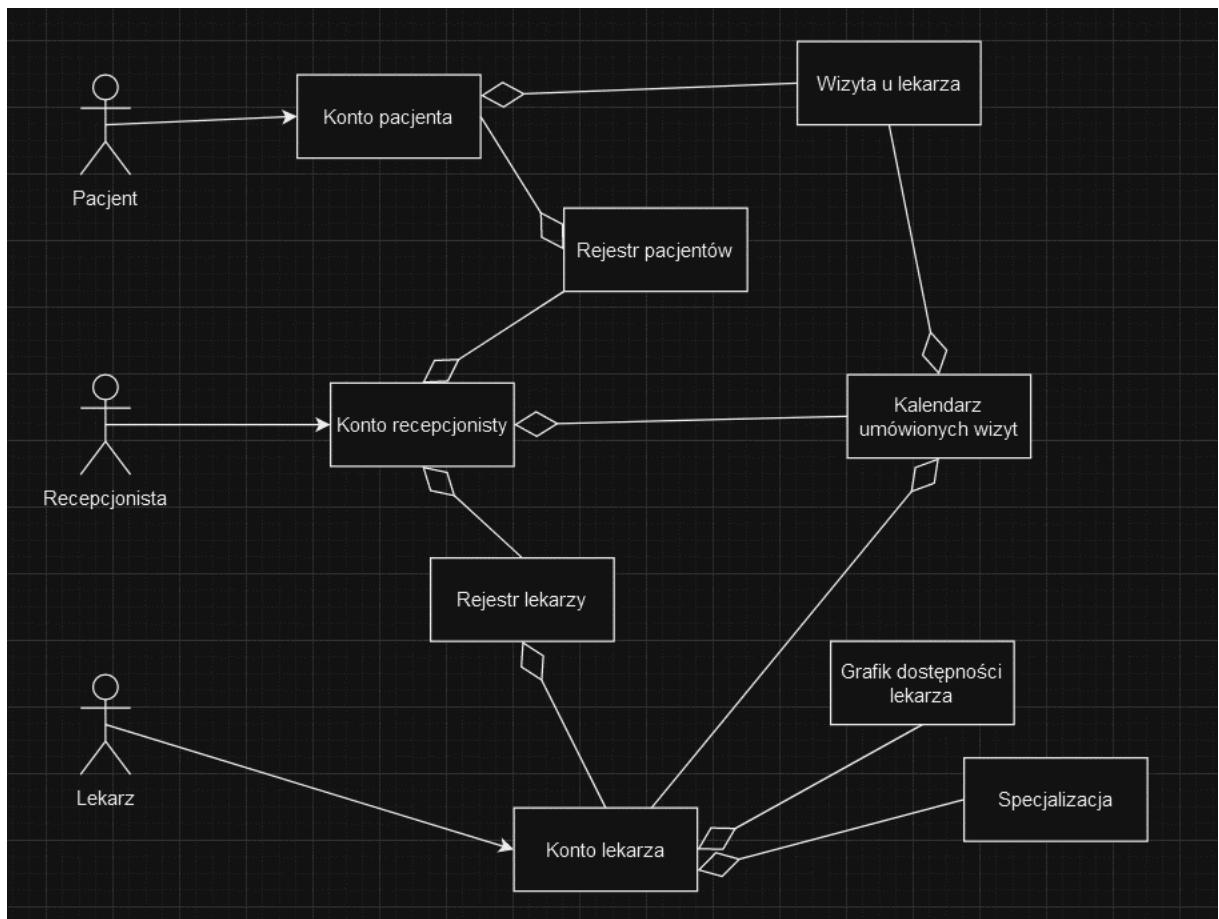
- Koncentracja na obiektach występujących w domenie biznesowej
- Używanie agregacji w celu odzwierciedlenia relacji między obiektami
- Zorganizowanie obiektów wokół kluczowych abstrakcji w domenie biznesowej
- Wykorzystywanie modeli domenowych w celu utworzenia słownika projektu
- Utworzenie diagramu 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 będący 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.

³ Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 35

⁴ Doug Rosenberg, Matt Stephens, Use Case Driven Object Modeling with UML. Theory and Practice. Appress, 2007, s. 25

⁵ Doug Rosenberg, Matt Stephens, Use Case Driven Object Modeling with UML. Theory and Practice. Appress, 2007, s. 26



Rys. 1.1 Diagram modeli domenowych (opracowanie własne)

1.3. 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 zaprojektowanie⁶. Ponadto, „*każdy przypadek użycia powinien zostać udokumentowany co najmniej trzema elementami:*

- 1) opisem stanu początkowego (warunków początkowych),
- 2) scenariuszem,
- 3) opisem oczekiwanej stanu końcowego”⁷

Diagram przypadków użycia składa się z graficznego odwzorowania aktorów systemu oraz samych przypadków użycia, w których to aktorzy biorą udział⁸. Ponadto, diagram może zawierać również dwa dodatkowe typy relacji: *extend* oraz *include*⁹. 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

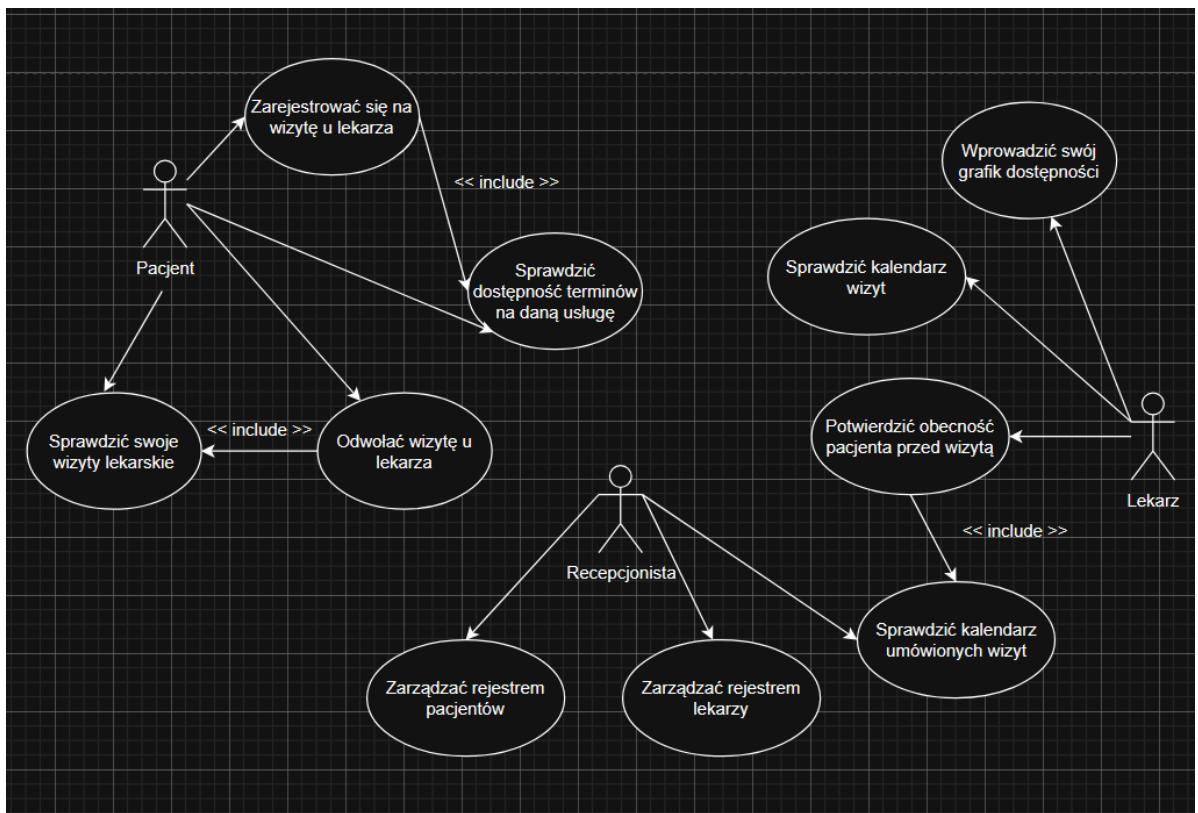
Na podstawie opisanych zasad, utworzonego diagramu modeli domenowych z rysunku 1.1, oraz opisu wymagań biznesowych, wykonany został diagram przypadków użycia przedstawiony na rysunku 1.2. Zawiera on trzech aktorów. Pierwszym z nich jest pacjent. Ma on możliwość umówienia się na wizyty u lekarza. Częścią 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 rejestrów lekarzy oraz rejestrów pacjentów. Ponadto, recepcjonista ma możliwość sprawdzenia kalendarza umówionych wizyt.

⁶ Doug Rosenberg, Matt Stephens, Use Case Driven Object Modeling with UML. Theory and Practice. Appress, 2007, s. 49

⁷ Jarosław Żeliński, Analiza Biznesowa. Praktyczne modelowanie organizacji, Helion, 2016, s. 74

⁸ Karl E Wieggers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 172

⁹ Karl E Wieggers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 179

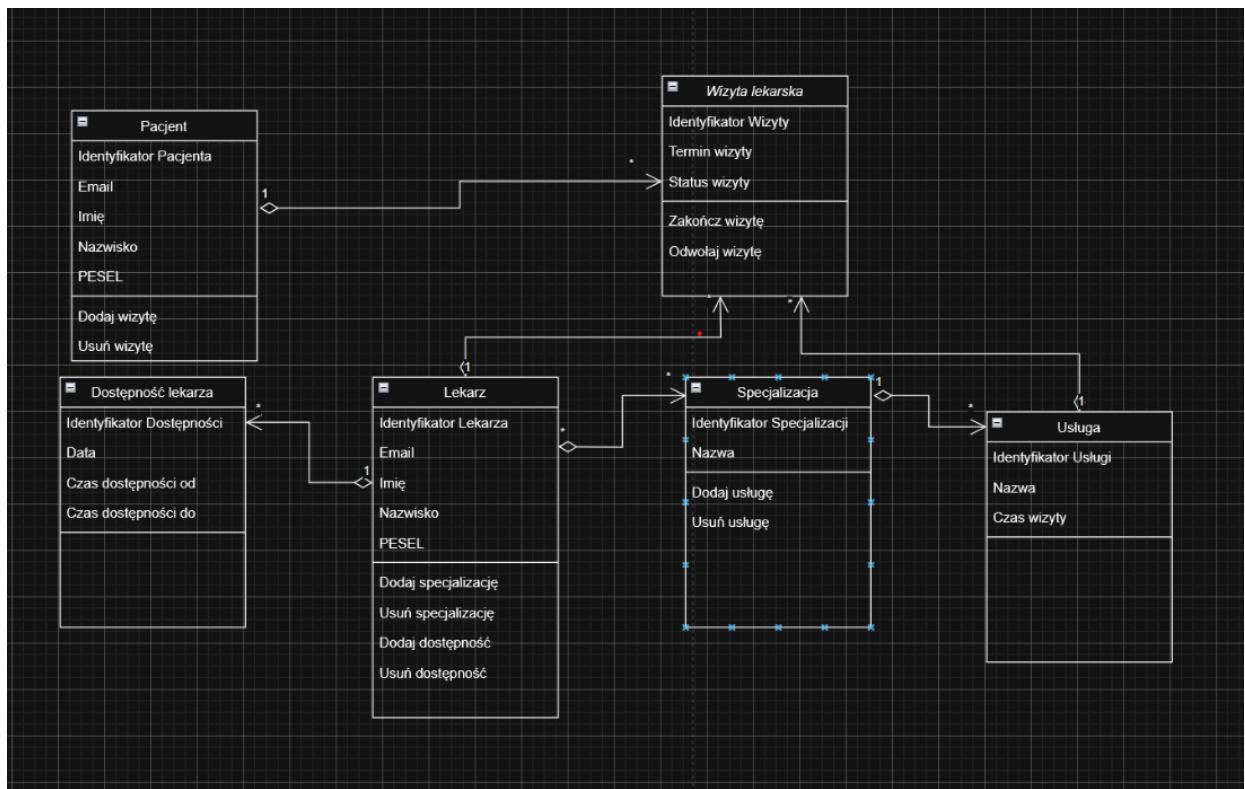


Rys. 1.2 Diagram przypadków użycia (opracowanie własne)

1.4. 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”¹⁰. 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*.

¹⁰ Karl E Wiegers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014, s. 261

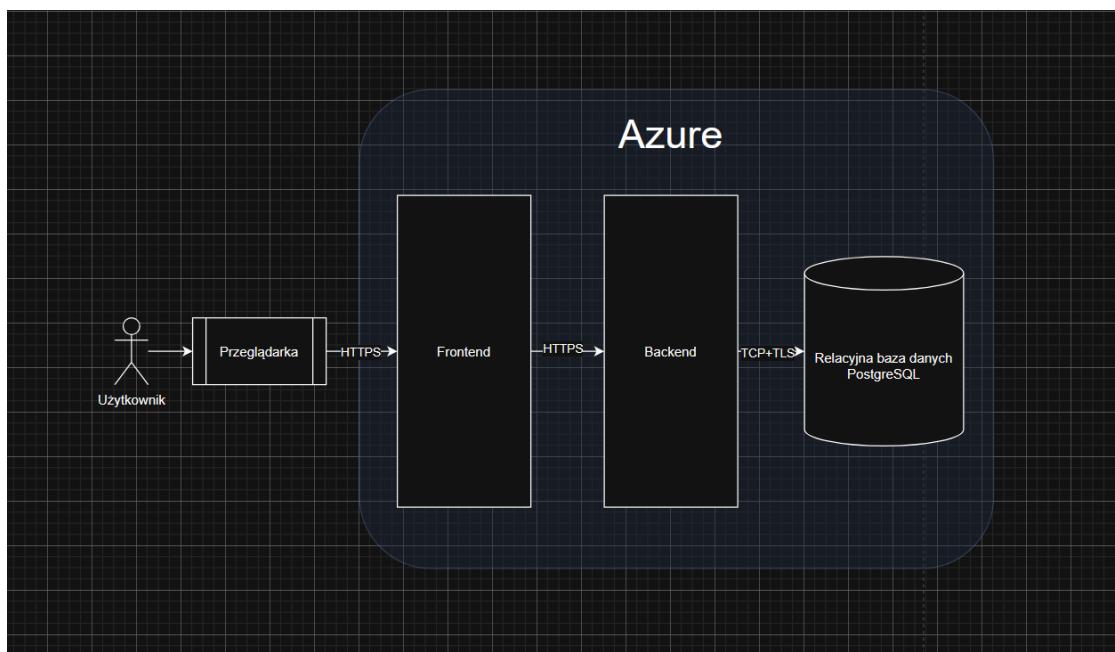


Rys. 1.3 Diagram klas (opracowanie własne)

2. ARCHITEKTURA SYSTEMU

2.1. Zarys architektury

System jest zbudowany z wykorzystaniem wzorca *Three-Tier Client Server Architecture*¹¹, 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*¹². 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

¹¹ Pethuru Raj, Anupama Raman, Harihara Subramanian, Architectural Patterns, Packt Publishing, 2017, s. 58

¹² <https://www.geeksforgeeks.org/frontend-vs-backend/>

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 bezpieczeństwo¹³. 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 developerów¹⁴. 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 wielowarstwości architektury.

¹³ Pethuru Raj, Anupama Raman, Harihara Subramanian, Architectural Patterns, Packt Publishing, 2017, s. 60

¹⁴ <https://medium.com/buildingminds-technologies/importance-of-collaboration-between-frontend-and-backend-teams-4b05e8fd29f9>

2.2. Bezpieczeństwo systemu

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

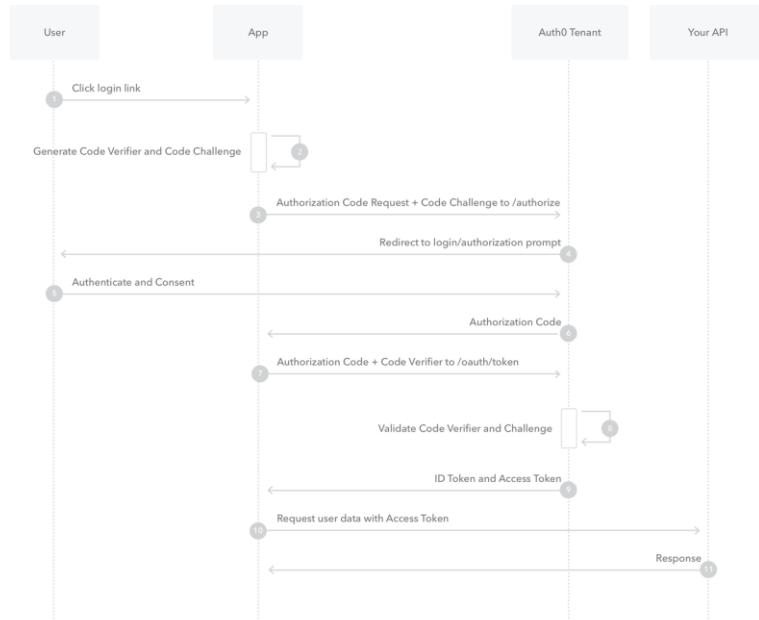
- Uwierzytelnienie przy użyciu Microsoft Entra ID
- Szyfrowane protokoły komunikacyjne przy użyciu TLS/SSL
- Uwierzytelnienie do bazy danych poprzez podanie nazwy użytkownika bazodanowego oraz hasła

Pierwsze dwa zostaną bardziej szczegółowo opisane w następujących podrozdziałach.

2.2.1. 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*¹⁵. Schemat oraz kroki, które się na niego składają można zobaczyć na rysunku nr 2.2.

¹⁵ <https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce>



Rys. 2.2 Schemat działania uwierzytelnienia Authorization 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)¹⁶, 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 ID¹⁷. 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 Base64¹⁸. 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 poinformowania kto został uwierzytelniony. Natomiast adres email wykorzystywany jest do identyfikacji użytkownika w backendzie i przypisywaniu do niego danych przechowywanych w bazie danych. Grupy zaś

¹⁶ <https://learn.microsoft.com/en-us/entra/identity-platform/access-token-claims-reference>

¹⁷ <https://learn.microsoft.com/en-us/entra/identity-platform/security-tokens#token-endpoints-and-issuers>

¹⁸ <https://jwt.io/introduction>

informują o tym, jakie role zostały przypisane do użytkownika. Na podstawie ról, frontend jest w stanie określić jakie strony i funkcjonalności powinny zostać wyświetlane.

JWT Decoder JWT Encoder

[Generate example](#)

Paste a JWT below that you'd like to decode, validate, and verify.

ENCODED VALUE

JSON WEB TOKEN (JWT)

[COPY](#)

[CLEAR](#)

Valid JWT

Signature Verified

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzIiImtpZC16IkNodjPSTMs5fS5EZFVm5h01Bc2hDSd
JYR519...eyJhdWQiO1IIMDyyOTR1OS03MDFjLTrIMtY5wWjk0ZTUyOGEZYmY1LCJpc3IoiJ
odHRwczovL2xvZ2luLm1pV3yc29md9ubgluZS5jb28Mmf10MbXMyIMzg3My08ZwVjLTkm2UeM
TU50T5YTHNTA3L3YyLjA1LCJpYXQ1OjE3NDY5NzExNyIsIm5zL1lMcToNjKmTE2N1wLZhwiJo
xNzQ2OTc1MDY2LCJncm91cHiM1oIsINTcwNtdhMzktyTEAzI08YWkLNjLYTgtYzBmNyJyXzUmNyAyI
10sIm5hbWUiO1JWYXppYSBTDH62ZWNoYSIsIm5vdmNIjoimZy20GRLjyMzN1zJyBmHtIyMT0ZDA
5Y2R1YTf1D2AxNxF1OTj0MoI1LCJvaQ10iI3NmIzMwEwy04Mjd1LTQ8WYtYUyJy1mNT1m0KzCm
jaZMwQ1LJcmVmZXyjZWRfdxN1cmShbmUiO1jtYXJpYSzdhJ6ZwNoYUbW1pbhAwNmdtYw1sLn9
ubWlJcm9bzBZLmBvNsIsInJoiJjoMS5BuZhbCxpINetuTTQ3RTZRUghx0mFhaTCN21VmxBY2NEc
ExMm9HE9WNG83X2xRBD2QUEuic21kIjoiMDA0ZtAwZDkctOTf1ZC00GIwLNj1NjYTUT5M2Y
3VT1YmQxiTiwei3VtIjoiVxD10hN2VL5fRfbF4FnB1YVgX2RdzFucGL0H9Yeet9dwJMeZhQ
SIsInRpZC16IjHjYjgMMwIi1TM4NmZtMtNGV1Yy05MDN1LTe10k20W4YjUyNIsInVba5I6IKfSsd
6Y8VxSGtxUTRAxL8sdvE4QUEi1Lc2ZXi10iyLjA1fq.QxR93jA5281b1uqJ3hbtJ4x4d07p9yXH
P8-LuNfWytP7pLPhWqfz_RpxEEQdmrgg8aXtmC1kw48zluBvVyxj_x_znhvKryTnWycceEMTA0Wf
MuU6yP2ta1xWaxoAnbep3iAB1Xz5QWY141kI_42p5lsREoKm7zpbD4n92-cnV1WjhsuJr14xu
HVKGVaTJWq09wsfInhPwkJezUxTKbpr9oH_4_716FWWWbirWGRE-PH3ca1bzcherGPBc-V1-PNdg
o4qFq5fuKr56ZP40AYrW15LOLhcbLJn4H3_o_jAiwxgrh7PN1y3lZq1fBaVOG6-byS6N9A
```

DECODED HEADER

JSON CLAIMS TABLE

[COPY](#)

[↗](#)

```
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "CNv00I3RwlHFEVnaoMAshCH2XE"
}
```

DECODED PAYLOAD

JSON CLAIMS TABLE

[COPY](#)

[↗](#)

```
{
  "aud": "5062949b-701c-4b3a-935a-0694e538a3bf",
  "iss": "https://login.microsoftonline.com/2ab831ab-3873-4eec-903e-1599
69a8b507/v2.0",
  "iat": 1746971166,
  "nbf": 1746971166,
  "exp": 1746975066,
  "groups": [
    "57057a39-a18f-4aed-bea8-c0f661ff12f0"
  ],
  "name": "Maria Strzecha",
  "nonce": "3668def0233bf64322134d09cdea1bd915qb92h0Z",
  "oid": "76b3fa0c-827e-44af-ae27-f59f0932031d",
  "preferred_username": "maria.strzecha@kamilp06@gmail.onmicrosoft.com",
  "rh": "1.A8A8qzG4KnM47E6QPhWZai1B7mUlyAccDpLkiG0U4o7_1AD9vAA",
  "sid": "004e00d9-91bd-48b0-be66-a593f7a6ebd",
  "sub": "Uwu8yb7eKJQklAxZpua_j_dsw1TpfkV8x0rubp0FgA",
  "tid": "2ab831ab-3873-4eec-903e-159969ab507",
  "uti": "AIh7zcEqHKWQ4x-p1uQ8AA",
  "ver": "2.0"
}
```

JWT SIGNATURE VERIFICATION (OPTIONAL)

Enter the public key used to sign the JWT below:

PUBLIC KEY

[COPY](#)

[CLEAR](#)

Valid public key

"kty": "RSA",

"n":

```
"h6z7USCSAuiyQz6L1nQj4za8kItevJzxhVbecMigT1l9pXZSHza3gZMgtapnb1q96CG5qrR
7o4uL7...TV1...DM-MAI/C4...7k1...E1...I...K...G...P...D...T...C...E...W...T...I...V...O...C...E...E...L...W...O...D...B...W...1...N...T...H...0...7...2...
```

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 tokenu. Ponadto, identyfikuje on tożsamość oraz sprawdza role zalogowanego użytkownika.

2.2.2. 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 wiedziała¹⁹. 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 TLS²⁰. Natomiast komunikacja pomiędzy backendem a bazą danych PostgreSQL wykorzystuje protokół TCP oraz TLS²¹. 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 TLS²². Możemy wyróżnić następującego kroki.

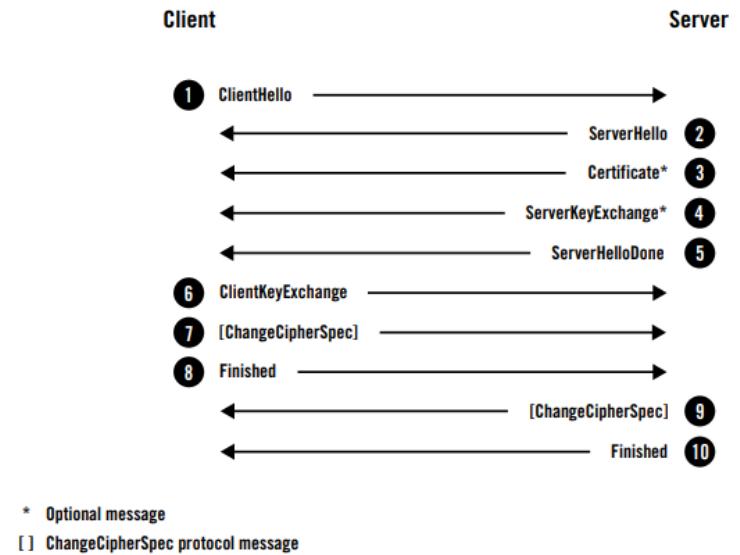
1. Klient inicjuje TLS Handshake
2. Serwer odpowiada na zainicjowany TLS Handshake wraz z parametrami połączenia
3. Serwer wysyła swój certyfikat TLS tak, aby klient mógł zweryfikować jego poprawność
4. Serwer wysyła dane potrzebne do wygenerowania klucza prywatnego
5. Server oznajmia, że zakończył wysyłanie 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

¹⁹ 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

²⁰ <https://www.rfc-editor.org/rfc/rfc2818>

²¹ <https://www.postgresql.org/docs/current/ssl-tcp.html>

²² Ivan Ristić, Bulletproof SSL and TLS, Feisty Duck Limited, 2022, s. 25



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 wyslaną 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 Middle²³.

²³ <https://auth0.com/blog/the-tls-handshake-explained/>

2.3. 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 Angular²⁴, gdzie kod źródłowy jest napisany w języku programowania TypeScript²⁵, CSS²⁶ oraz HTML²⁷.

2.3.1. HTML

HyperText Markup Language (w skrócie HTML)²⁸ jest językiem znaczników, który definiuje treść strony internetowej wyświetlanej przez przeglądarkę. Kod źródłowy strony internetowej jest zapisywany w formie tekstu 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 przykładem może być paragraf, czyli znacznik typu *p* wraz z atrybutem *style*, opisujący cechy stylistyczne paragrafu takie jak rozmiar czcionki bądź jej kolor.

²⁴ <https://angular.dev/>

²⁵ <https://www.typescriptlang.org/>

²⁶ <https://developer.mozilla.org/en-US/docs/Web/CSS>

²⁷ <https://developer.mozilla.org/en-US/docs/Web/HTML>

²⁸ <https://udigroup.pl/blog/jazyk-html-co-to-jest-do-czego-sluzy-jak-wyglada/>



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

2.3.2. CSS

Cascading Style Sheets (w skrócie CSS)²⁹ 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 bezpośrednio 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 takiego użycia zobrazowany jest na rysunku nr 2.6.

²⁹ <https://webporadnik.pl/css-co-to-jest-i-jak-dziala-css-podstawowe-i-najwazniejsze-informacje-o-css>

The screenshot shows a web-based code editor interface for CSS. At the top, there's a navigation bar with 'Home', 'CSS', 'CSS How To', and a search bar containing 'Tryit: Mix of internal and external style sheets'. Below the navigation is a toolbar with icons for file operations and a 'Run' button. The main area contains an HTML code editor with the following content:

```
<!DOCTYPE html>
<html>
<body>
<h1 style="color: red">This is a heading</h1>
<p style="color: blue">This text is blue</p>
</body>
</html>
```

To the right of the code editor, the output is displayed in a large red font: "This is a heading". Below it, in blue text, is "This text is blue".

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

2.3.3. TypeScript

TypeScript³⁰ jest językiem programowania, który jest rozszerzeniem języka JavaScript³¹. Wprowadza on m.in. statyczne typowanie, które zdecydowanie zmniejsza liczbę błędów w kodzie oraz zapewnia dodatkową dokumentację. TypeScript jest komplikowany 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.

2.3.4. Angular

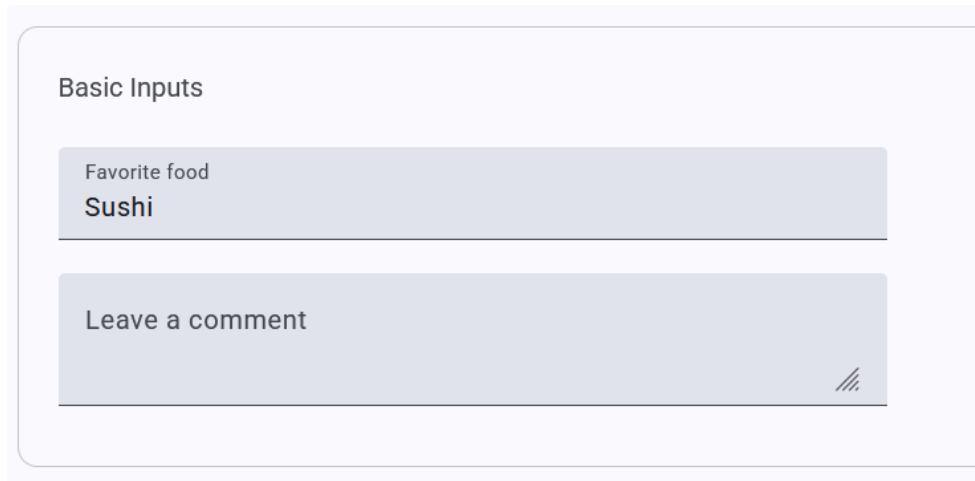
Angular³² jest frameworkiem frontendowym napisanym w języku TypeScript przez firmę Google. Wprowadza on narzędzia do tworzenia aplikacji internetowych m.in. typu SPA (Single Page Application). Aplikacje tego typu w porównaniu do SSR (Server Side Rendering) charakteryzują się ogólnie szybszym działaniem, 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.

³⁰ <https://www.typescriptlang.org>

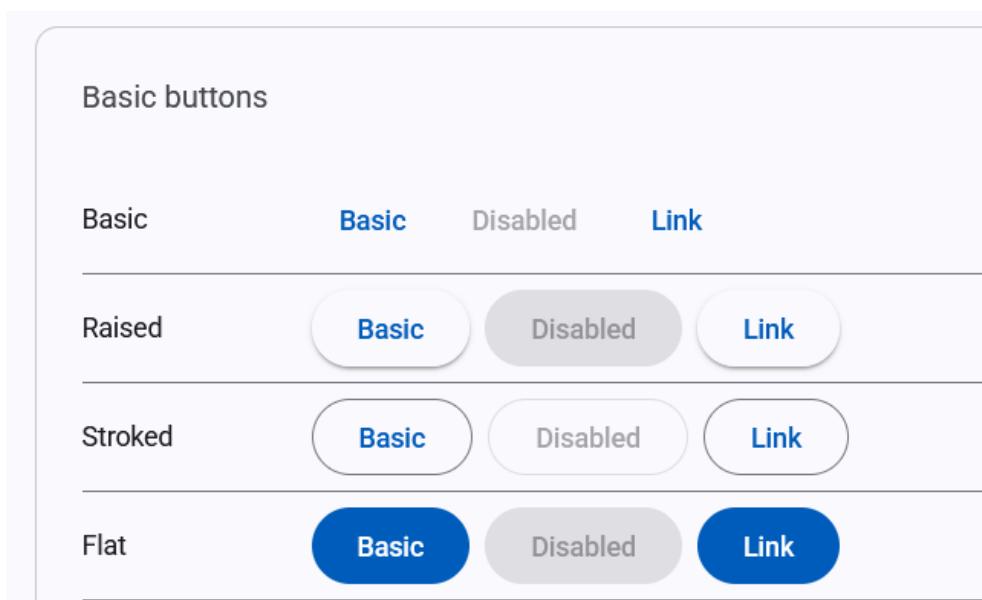
³¹ <https://www.w3schools.com/Js>

³² <https://angular.dev/overview>

Ponadto, Angular wprowadza tzw. framework CSS o nazwie Angular Material³³. 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>)

³³ <https://material.angular.dev/>

2.4. Backend

Backend jest usługą sieciową działającą na serwerze, która wystawia REST API (skrót od Representational State Transfer Application Programming Interface)³⁴, oraz komunikującą 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.

2.4.1. 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 całkowicie aktualizujących istniejące dane
- PATCH – używana przy nieidempotentnych endpointach częściowo aktualizujących istniejące dane
- DELETE – używana przy idempotentnych endpointach usuwających dane

³⁴ <https://poradnikinżyniera.pl/rest-api-co-to-jest-jak-dziala-i-jak-z-niego-korzystac/>

Warto zwrócić uwagę na pojęcie idempotencji. Jeżeli wybrany endpoint jest idempotentny, wówczas użycie go dwa lub więcej razy będzie skutkować tym samym rezultatem³⁵. Jest to istotna cecha wybranych metod HTTPS, która powinna być brana pod uwagę przy projektowaniu interfejsu REST API.

2.4.2. Java SE

Java SE jest obiektowym językiem programowania³⁶ 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 maszynowy³⁷. 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 Collection³⁸, który polega na uwolnieniu zasobów pamięci operacyjnej RAM, gdy nie są używane.

2.4.3. Spring

Spring Framework jest zbiorem narzędzi napisanych w języku Java, które ułatwiają tworzenie aplikacji biznesowych w językach opartych o JVM, w tym Javy³⁹. Jednymi z najbardziej popularnych modułów Springa jest Spring Framework⁴⁰, Spring Web MVC⁴¹, Spring Data⁴² oraz Spring Boot⁴³. Spring Framework wprowadza Dependency Injection⁴⁴, które implementuje wzorzec IoC Container. Umożliwia on łatwe testowanie oraz konfigurację aplikacji poprzez zastosowanie tzw. Beanów⁴⁵, 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

³⁵ <https://restfulapi.net/idempotent-rest-apis/>

³⁶ <https://www.oracle.com/pl/java/>

³⁷ Cay S. Horstmann, Java Podstawy. Wydanie XI, Helion, 2019, s. 25

³⁸ <https://www.oracle.com/webfolder/technetwork/Tutorials/obe/java/gc01/index.html>

³⁹ <https://spring.io/projects/spring-framework>

⁴⁰ <https://spring.io/projects/spring-framework>

⁴¹ <https://docs.spring.io/spring-framework/reference/web/webmvc.html>

⁴² <https://spring.io/projects/spring-data>

⁴³ <https://spring.io/projects/spring-boot>

⁴⁴ <https://docs.spring.io/spring-framework/reference/core/beans.html>

⁴⁵ <https://docs.spring.io/spring-framework/reference/core/beans/definition.html>

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.

2.4.4. Gradle

Gradle⁴⁶ 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.

2.4.5. Liquibase

Liquibase⁴⁷ 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.

2.4.6. Lombok

Lombok⁴⁸ 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 Setterami⁴⁹. Lombok umożliwia definicję ich za pomocą adnotacji @Setter.

2.4.7. JUnit 4

JUnit4⁵⁰ 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.

⁴⁶ <https://gradle.org/>

⁴⁷ <https://www.liquibase.com/>

⁴⁸ <https://projectlombok.org/>

⁴⁹ <https://projectlombok.org/features/GetterSetter>

⁵⁰ <https://junit.org/junit4/>

2.5. Baza danych

Zastosowaną bazą danych w projekcie *Great Health* jest PostgreSQL⁵¹. 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 SQL⁵² będącego standardem komunikacji z relacyjnymi bazami danych.

⁵¹ <https://www.postgresql.org/>

⁵² <https://www.geeksforgeeks.org/what-is-sql/>

3. IMPLEMENTACJA SYSTEMU

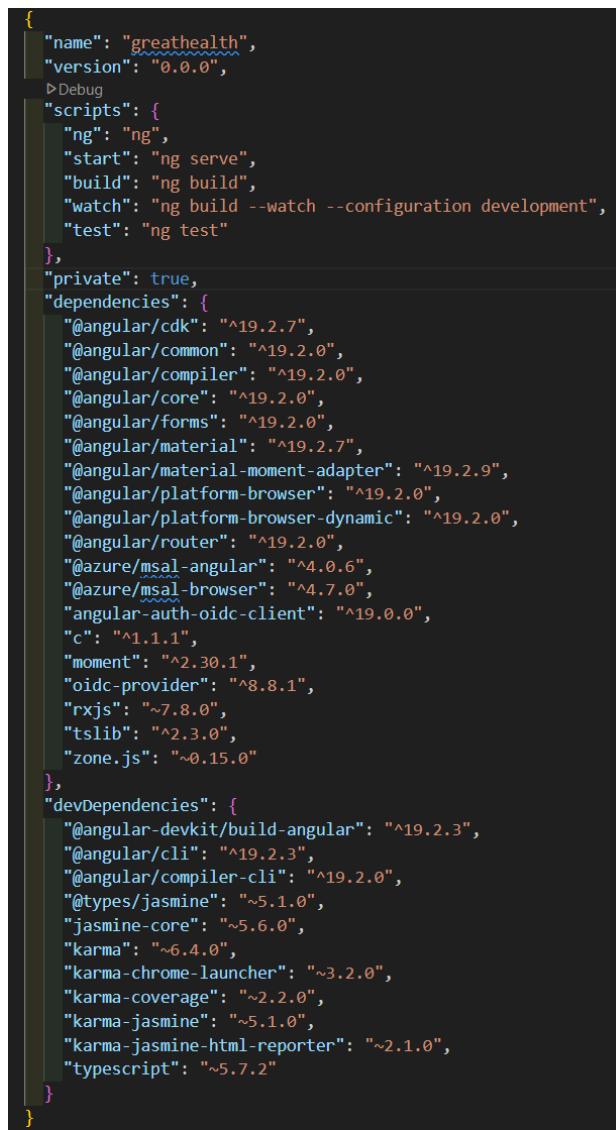
System składa się z dwóch aplikacji – frontendu (warstwa prezentacji) oraz backendu (warstwa logiki biznesowej). Omówienie implementacji systemu warto zacząć od pierwszego komponentu tj. frontendu, a następnie płynnie przejść do warstwy backendowej.

3.1. Implementacja frontendu

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

3.1.1. Konfiguracja

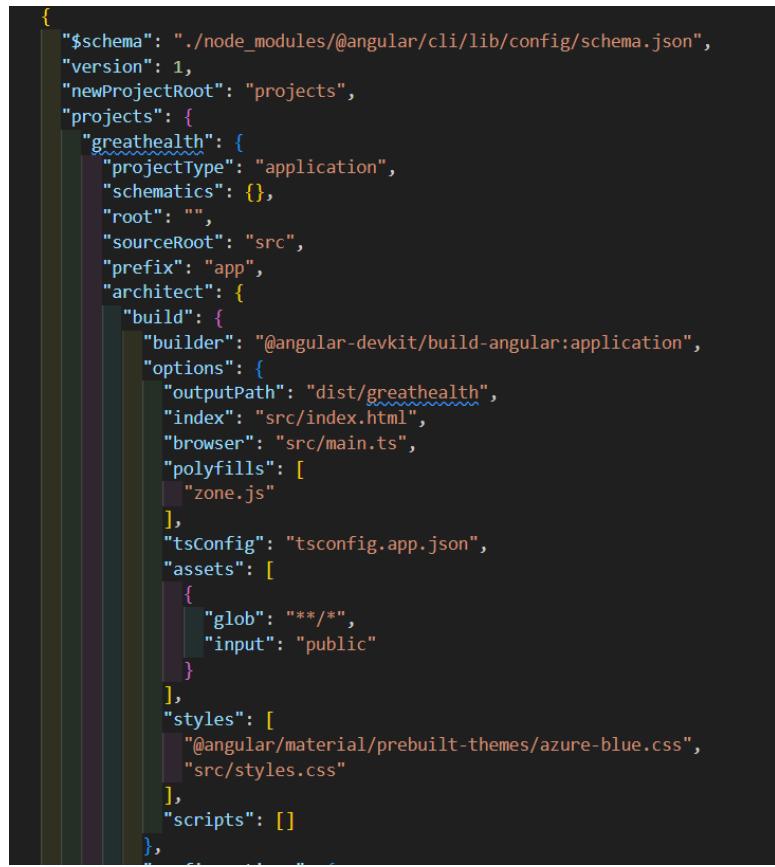
Zarządzanie zależnościami odbywa się w pliku *package.json*, co zostało zaprezentowane na rysunku nr 3.1.



```
{
  "name": "greathealth",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test"
  },
  "private": true,
  "dependencies": {
    "@angular/cdk": "^19.2.7",
    "@angular/common": "^19.2.0",
    "@angular/compiler": "^19.2.0",
    "@angular/core": "^19.2.0",
    "@angular/forms": "^19.2.0",
    "@angular/material": "^19.2.7",
    "@angular/material-moment-adapter": "^19.2.9",
    "@angular/platform-browser": "^19.2.0",
    "@angular/platform-browser-dynamic": "^19.2.0",
    "@angular/router": "^19.2.0",
    "@azure/msal-angular": "^4.0.6",
    "@azure/msal-browser": "^4.7.0",
    "angular-auth-oidc-client": "^19.0.0",
    "c": "^1.1.1",
    "moment": "2.30.1",
    "oidc-provider": "8.8.1",
    "rxjs": "7.8.0",
    "tslib": "2.3.0",
    "zone.js": "0.15.0"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "19.2.3",
    "@angular/cli": "19.2.3",
    "@angular/compiler-cli": "19.2.0",
    "@types/jasmine": "5.1.0",
    "jasmine-core": "5.6.0",
    "karma": "6.4.0",
    "karma-chrome-launcher": "3.2.0",
    "karma-coverage": "2.0.0",
    "karma-jasmine": "5.1.0",
    "karma-jasmine-html-reporter": "2.1.0",
    "typescript": "5.7.2"
  }
}
```

Rys. 3.1 Plik *package.json* zarządzający zależnościami 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.



```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,  
  "newProjectRoot": "projects",  
  "projects": {  
    "greathealth": {  
      "projectType": "application",  
      "schematics": {},  
      "root": "",  
      "sourceRoot": "src",  
      "prefix": "app",  
      "architect": {  
        "build": {  
          "builder": "@angular-devkit/build-angular:application",  
          "options": {  
            "outputPath": "dist/greathealth",  
            "index": "src/index.html",  
            "browser": "src/main.ts",  
            "polyfills": [  
              "zone.js"  
            ],  
            "tsConfig": "tsconfig.app.json",  
            "assets": [  
              {  
                "glob": "**/*",  
                "input": "public"  
              }  
            ],  
            "styles": [  
              "@angular/material/prebuilt-themes/azure-blue.css",  
              "src/styles.css"  
            ],  
            "scripts": []  
          },  
          "devServer": {}  
        }  
      }  
    }  
  }  
}
```

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 realizowaną 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.

```

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideAnimationsAsync(),
    provideRouter(routes),
    provideAuth(
      {
        config: {
          authority:
            | 'https://login.microsoftonline.com/2ab831ab-3873-4eec-903e-159969a8b507/v2.0',
            | authWellKnownEndpointUrl:
            | 'https://login.microsoftonline.com/2ab831ab-3873-4eec-903e-159969a8b507/v2.0',
            redirectUrl: window.location.origin,
            clientId: '506294b9-701c-4b3a-935a-0694e538a3bf',
            scope:
              | 'openid profile offline_access email',
              responseType: 'code',
              silentRenew: true,
              maxIdTokenLifetimeAllowedInSeconds: 600,
              issValidationOff: true,
              autoUserInfo: false,
              // silentRenewUrl: window.location.origin + '/silent-renew.html',
              useRefreshToken: true,
              logLevel: LogLevel.Debug,
              customParamsAuthRequest: {
                | prompt: 'login', // login, consent
              },
            },
        },
        withAppInitializerAuthCheck()
      ),
      provideHttpClient(
        withInterceptorsFromDi(),
        withInterceptors([authInterceptor()])
      ),
      [
        {
          provide: HTTP_INTERCEPTORS,
          useClass: TokenInterceptorService,
          multi: true
        },
        {provide: MAT_DATE_LOCALE, useValue: 'pl-PL'}
      ]
    );
  ];
}

```

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

Kolejnym ważnym 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.

```

import { Injectable } from '@angular/core';
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from "@angular/common/http";
import { Observable, switchMap } from 'rxjs';
import { OidcSecurityService } from 'angular-auth-oidc-client';

@Injectable({providedIn: 'root'})
export class TokenInterceptorService implements HttpInterceptor {

  constructor(private oidcSecurityService: OidcSecurityService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    console.log("hello")
    return this.oidcSecurityService.getIdToken().pipe(
      switchMap(token => {
        if (token) {
          req = req.clone({
            setHeaders: {
              Authorization: `Bearer ${token}`
            }
          });
        }
        return next.handle(req);
      })
    );
  }
}

```

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.

```

export const routes: Routes = [
  {
    path: '',
    component: HomeComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  {
    path: 'doctors',
    component: DoctorsComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  {
    path: 'patients',
    component: PatientsComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  {
    path: 'specialties',
    component: SpecialtiesComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  {
    path: 'services',
    component: ServicesComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  {
    path: 'doctor-appointments',
    component: DoctorAppointmentsComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  {
    path: 'availability',
    component: AvailabilityComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  {
    path: 'patient-appointments',
    component: PatientAppointmentsComponent,
    canActivate: [autoLoginPartialRoutesGuard],
  },
  { path: 'unauthorized', component: UnauthorizedComponent },
];

```

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.

```

<mat-toolbar>
  <section>
    <a mat-button href="/">Great Health</a>
    <a mat-button *ngIf="(userData$ | async)?.userData?.groups?.includes('57057a39-a18f-4aed-bea8-c0f661ff12f0')" href="/doctors">Lekarze</a>
    <a mat-button *ngIf="(userData$ | async)?.userData?.groups?.includes('57057a39-a18f-4aed-bea8-c0f661ff12f0)" href="/patients">Pacjenci</a>
    <a mat-button *ngIf="(userData$ | async)?.userData?.groups?.includes('57057a39-a18f-4aed-bea8-c0f661ff12f0)" href="/specialties">Specializacje</a>
    <a mat-button *ngIf="(userData$ | async)?.userData?.groups?.includes('57057a39-a18f-4aed-bea8-c0f661ff12f0)" href="/services">Usługi</a>
    <a mat-button *ngIf="(userData$ | async)?.userData?.groups?.includes('2350f29d-627c-4bd0-b5a4-e757825ba21c)" href="/doctor-appointments">Wizyty z pacjentami</a>
    <a mat-button *ngIf="(userData$ | async)?.userData?.groups?.includes('2350f29d-627c-4bd0-b5a4-e757825ba21c)" href="/availability">Dostępność</a>
    <a mat-button *ngIf="(userData$ | async)?.userData?.groups?.includes('973529ee-6146-4109-b7ef-21ea5f38ebe2)" href="/patient-appointments">Wizyty</a>
  </section>
  <div class="user-info">
    <span>{( userData$ | async)?.userData?.name }</span>
    <button mat-icon-button class="example-icon favorite-icon" *ngIf="isAuthenticated" (click)="logout()">
      <mat-icon>logout</mat-icon>
    </button>
    <button mat-icon-button class="example-icon favorite-icon" *ngIf="!isAuthenticated" (click)="logout()">
      <mat-icon>login</mat-icon>
    </button>
  </div>
</mat-toolbar>

```

Rys. 3.6 Warstwa prezentacji komponentu menu (własne opracowanie)

```

@Component({
  selector: 'app-nav-menu',
  templateUrl: './nav-menu.component.html',
  styleUrls: ['./nav-menu.component.css'],
  imports: [RouterLink, NgIf, MatIconModule, MatButtonModule, MatToolbarModule, CommonModule, MatSidenavModule]
})
export class NavMenuComponent implements OnInit {
  private readonly oidcSecurityService = inject(OidcSecurityService);

  isAuthenticated = false;
  userData$ = this.oidcSecurityService.userData$;
  getAccessToken = this.oidcSecurityService.getIdToken();

  ngOnInit(): void {
    this.oidcSecurityService.isAuthenticated$.subscribe(
      (( isAuthenticated )) => {
        this.isAuthenticated = isAuthenticated;
        console.log(this.oidcSecurityService.getIdToken());
        console.warn('authenticated: ', isAuthenticated);
      }
    );
  }

  login(): void {
    this.oidcSecurityService.authorize();
  }

  // refreshSession(): void {
  //   this.oidcSecurityService
  //     .forceRefreshSession()
  //     .subscribe((result) => console.log(result));
  // }

  logout(): void {
    this.oidcSecurityService
      .logoff()
      .subscribe((result) => console.log(result));
  }
}

```

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.

Great Health Wizyty z pacjentami Dostępność

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

Great Health Wizyty

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

Great Health Lekarze Pacjenci Specjalizacje Usługi

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

3.1.2. 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.

Wizyty z pacjentami					
Data	Czas wizyty	Pacjent	PESEL Pacjenta	Usługa	Status
2025-06-17T13:30:00	30 minut	Marcin Kolarczyk	75111547838	Konsultacja internisty	Zakończona
2025-06-17T13:30:00	30 minut	Marcin Kolarczyk	75111547838	Konsultacja internisty	Anulowana
2025-06-17T17:00:00	60 minut	Aneta Kowalewska-Malarczyk	92102183468	Konsultacja kardiologa	Utworzona ✓
2025-06-17T12:00:00	60 minut	Marcin Kolarczyk	75111547838	Konsultacja kardiologa	Utworzona ✓
2025-06-17T15:00:00	60 minut	Marcin Kolarczyk	75111547838	Konsultacja kardiologa	Utworzona ✓
2025-06-17T14:00:00	60 minut	Marcin Kolarczyk	75111547838	Konsultacja kardiologa	Utworzona ✓

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 istnienie w tym przypadku jest zalogowany użytkownik. Jeżeli użytkownik nie został zalogowany, podstrona nie powinna się wyświetlać.

```

<div *ngIf="loggedInDoctorUuid != null">
  <div class="header">
    <h2>Wizyty z pacjentami</h2>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="dataSourceAppointments" class="mat-elevation-z8 demo-table">

        <ng-container matColumnDef="date">
          <th mat-header-cell *matHeaderCellDef> Data </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.dateTimeFrom}} minut </td>
        </ng-container>

        <ng-container matColumnDef="timeInMinutes">
          <th mat-header-cell *matHeaderCellDef> Czas wizyty </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.service.timeInMinutes}} minut </td>
        </ng-container>

        <ng-container matColumnDef="patient">
          <th mat-header-cell *matHeaderCellDef> Pacjent </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.patient.name}} {{appointment.patient.surname}}</td>
        </ng-container>

        <ng-container matColumnDef="patientPESEL">
          <th mat-header-cell *matHeaderCellDef> PESEL Pacjenta </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.patient.pesel}}</td>
        </ng-container>

        <ng-container matColumnDef="service">
          <th mat-header-cell *matHeaderCellDef> Usługa </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.service.name}} </td>
        </ng-container>

        <ng-container matColumnDef="status">
          <th mat-header-cell *matHeaderCellDef> Status </th>
          <td mat-cell *matCellDef="let appointment"> {{translateStatus(appointment.status)}} </td>
        </ng-container>

        <ng-container matColumnDef="action">
          <th mat-header-cell *matHeaderCellDef> </th>
          <td mat-cell *matCellDef="let element" class="action-link">
            <button *ngIf="element.status == 'CREATED'" (click)="finishAppointment(element)" mat-icon-button class="example-icon favorite-icon">
              <mat-icon>check</mat-icon>
            </button>
          </td>
        </ng-container>

      <tr mat-header-row *matHeaderRowDef="displayedColumnsAppointment"></tr>
      <tr mat-row *matRowDef="let row; columns: displayedColumnsAppointment;"></tr>
    </table>
  </mat-card-content>

```

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.

```

@Component({
  selector: 'app-doctor-appointments',
  imports: [MatTableModule, CommonModule, MatCardModule, MatIconModule, MatButtonModule],
  templateUrl: './doctor-appointments.component.html',
  styleUrls: ['./doctor-appointments.component.css']
})
export class DoctorAppointmentsComponent {
  doctor ApiService = inject(Doctor ApiService)
  user ApiService = inject(User ApiService)
  displayedColumnsAppointment: string[] = ['date', 'timeInMinutes', 'patient', 'patientPESEL', 'service', 'status', 'action'];
  dataSourceAppointments: MatTableDataSource<Appointment>;
  loggedInDoctorUuid: string;

  ngOnInit(): void {
    this.user ApiService.getCurrentUserUuid().subscribe(doctorUuid => {
      this.loggedInDoctorUuid = doctorUuid.uuid;
      this.refreshTable();
    })
  }

  finishAppointment(appointment: Appointment) {
    if (appointment.status == 'CREATED') {
      this.doctor ApiService.finishAppointment(this.loggedInDoctorUuid, appointment.uuid).subscribe(a => {
        this.refreshTable();
      })
    }
  }

  refreshTable(): void {
    this.doctor ApiService.getAppointments(this.loggedInDoctorUuid).subscribe(data => {
      this.dataSourceAppointments = new MatTableDataSource<Appointment>(data);
    })
  }

  translateStatus(status: string): string {
    if (status == 'CREATED')
      return 'Utworzona';
    else if ([status == 'CANCELED'])
      return 'Anulowana';
    else if (status == 'FINISHED')
      return 'Zakończona';
    return '';
  }
}

```

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

3.1.3. 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.

Dostępność

Czas od	Czas do
2025-04-17T11:00:00	2025-06-17T10:00:00
2025-04-07T22:00:00	2025-04-24T22:00:00
2025-04-09T22:00:00	2025-04-09T23:30:00
2025-04-14T11:30:00	2025-04-14T14:00:00

Rys. 3.14 Wygląd podstrony Dostępność (opracowanie własne)

The form consists of four input fields arranged vertically, each with a date icon (calendar) to its right:

- Dzień dostępności od: 18.06.2025
- Czas dostępności od: 08:00
- Dzień dostępności do: 18.06.2025
- Czas dostępności do: 17:00

At the bottom left is a blue "Anuluj" (Cancel) button, and at the bottom right is a blue "Zapisz" (Save) button.

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.

```

<div *ngIf="loggedInDoctorUuid != null">
  <div class="header">
    <h1>Dostępność</h1>
    <button mat-icon-button class="example-icon favorite-icon" (click)="create()">
      <mat-icon>add_circle</mat-icon>
    </button>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="dataSourceAvailabilities" class="mat-elevation-z8 demo-table">
        <ng-container matColumnDef="dateTimeFrom">
          <th mat-header-cell *matHeaderCellDef> Czas od </th>
          <td mat-cell *matCellDef="let availability"> {{availability.dateTimeFrom}} </td>
        </ng-container>

        <ng-container matColumnDef="dateTimeTill">
          <th mat-header-cell *matHeaderCellDef> Czas do </th>
          <td mat-cell *matCellDef="let availability"> {{availability.dateTimeTill}} </td>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="displayedColumnsAvailability"></tr>
        <tr mat-row *matRowDef="let row; columns: displayedColumnsAvailability;"></tr>
      </table>
    </mat-card-content>
  </mat-card>
</div>

```

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

```

@Component({
  selector: 'app-availability',
  imports: [MatTableModule, CommonModule, MatCardModule, MatIconModule, MatButtonModule],
  templateUrl: './availability.component.html',
  styleUrls: ['./availability.component.css']
})
export class AvailabilityComponent {
  @Inject(DoctorAvailabilityApiService) doctorAvailabilityApiService: DoctorAvailabilityApiService;
  (property) AvailabilityComponent.displayedColumnsAvailability: string[];
  displayedColumnsAvailability: string[] = ['dateTimeFrom', 'dateTimeTill'];
  dataSourceAvailabilities: MatTableDataSource<Availability>;
  loggedInDoctorUuid: string;
  readonly dialog = inject(MatDialog);

  ngOnInit(): void {
    this.userService.getCurrentUserUuid().subscribe(doctorUuid => {
      this.loggedInDoctorUuid = doctorUuid.uuid;
      this.refreshTable();
    });
  }

  create(): void {
    const dialogRefRef = this.dialog.open(AvailabilityFormComponent, {
      data: {},
      height: '400px',
      width: '600px',
    });

    dialogRefRef.afterClosed().subscribe(result => {
      if (result !== undefined) {
        this.doctorAvailabilityApiService.createAvailability(this.loggedInDoctorUuid, result as DoctorsAvailabilityRequest).subscribe(res => {
          this.refreshTable();
        });
      }
    });
  }

  refreshTable(): void {
    this.doctorAvailabilityApiService.getAllAvailabilities(this.loggedInDoctorUuid).subscribe(availabilities => {
      this.dataSourceAvailabilities = new MatTableDataSource<Availability>(availabilities);
    });
  }
}

```

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 zamknięty z zapisanym modelem lub bez.

```
<mat-card appearance="outlined">
  <mat-card-content>

    <mat-form-field>
      <mat-label>Dzień dostępności od</mat-label>
      <input matInput [matDatepicker]="datepicker" [(ngModel)]="doctorAvailability.dateTimeFrom">
      <mat-datepicker #datepicker />
      <mat-datepicker-toggle [for]="datepicker" matsuffix/>
    </mat-form-field>

    <mat-form-field>
      <mat-label>Czas dostępności od</mat-label>
      <input matInput
        [matTimepicker]="timepicker"
        [(ngModel)]="doctorAvailability.dateTimeFrom"
        [ngModelOptions]="{{updateOn: 'blur'}}">
      <mat-timepicker #timepicker />
      <mat-timepicker-toggle [for]="timepicker" matsuffix/>
    </mat-form-field>

    <mat-form-field>
      <mat-label>Dzień dostępności do</mat-label>
      <input matInput [matDatepicker]="datepicker1" [(ngModel)]="doctorAvailability.dateTimeTill">
      <mat-datepicker #datepicker1 />
      <mat-datepicker-toggle [for]="datepicker1" matsuffix/>
    </mat-form-field>

    <mat-form-field>
      <mat-label>Czas dostępności do</mat-label>
      <input matInput
        [matTimepicker]="timepicker1"
        [(ngModel)]="doctorAvailability.dateTimeTill"
        [ngModelOptions]="{{updateOn: 'blur'}}">
      <mat-timepicker #timepicker1 />
      <mat-timepicker-toggle [for]="timepicker1" matsuffix/>
    </mat-form-field>

    <div class="lastRow">
      <button mat-button (click)="cancel()">Anuluj</button>
      <button mat-button (click)="save()">Zapisz</button>
    </div>
  </mat-card-content>
</mat-card>
```

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

```

@Component({
  selector: 'app-availability-form',
  imports: [
    MatFormFieldModule,
    MatInputModule,
    FormsModule,
    MatButtonModule,
    MatDialogTitle,
    MatDialogContent,
    MatDialogActions,
    MatDialogClose,
    MatCardModule,
    MatSelectModule,
    MatDatepickerModule,
    MatTimepickerModule,
  ],
  providers: [provideNativeDateAdapter()],
  templateUrl: './availability-form.component.html',
  styleUrls: ['./availability-form.component.css']
})
export class AvailabilityFormComponent {
  readonly dialogRef = inject(MatDialogRef<AvailabilityFormComponent>);
  readonly doctorAvailability = structuredClone(inject<DoctorsAvailabilityRequest>(MAT_DIALOG_DATA));

  cancel(): void {
    this.dialogRef.close();
  }

  save(): void {
    this.dialogRef.close(this.doctorAvailability);
  }
}

```

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

3.1.4. 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.

Lekarze

Email	Imię	Nazwisko	PESEL	Specjalizacje		
adam.kowalski@kamilp06@gmail.onmicrosoft.com	Adam	Kowalski	91051619075	Kardiolog		
zuzanna.skowronska@kamilp06@gmail.onmicrosoft.com	Zuzanna	Skowrońska-Kulczyk	85010732503	Internista, Kardiolog		

Rys. 3.20 Wygląd podstrony Lekarze (opracowanie własne)

05
01

Email
zuzanna.skowronska@kamilp06@gmail.onmicrosoft.com

Imię
Zuzanna

Nazwisko
Skowrońska-Kulczyk

PESEL
85010732503

Specjalizacje
Kardiolog, Internista

Anuluj **Zapisz**

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

Lekarze

Email	Imię	Nazwisko	PESEL	Specjalizacje		
adam.kowalski@kamil06@gmail.onmicrosoft.com	Adam	Kowalski	91051619075	Kardiolog		
zuzanna.skowronska@kamil06@gmail.onmicrosoft.com	Zuzanna	Skowrońska-Kulczyk	85010732503	Internista, Kardiolog		

Wizyty z pacjentami - Dr Zuzanna Skowrońska-Kulczyk

Data	Czas wizyty	Pacjent	PESEL Pacjenta	Usługa	Status
2025-06-17T13:30:00	30 minut	Marcin Kolarczyk	75111547838	Konsultacja internisty	Zakonczona
2025-06-17T13:30:00	30 minut	Marcin Kolarczyk	75111547838	Konsultacja internisty	Anulowana
2025-06-17T17:00:00	60 minut	Aneta Kowalewska-Malarczyk	92102183468	Konsultacja kardiologa	Utworzona
2025-06-17T12:00:00	60 minut	Marcin Kolarczyk	75111547838	Konsultacja kardiologa	Utworzona
2025-06-17T15:00:00	60 minut	Marcin Kolarczyk	75111547838	Konsultacja kardiologa	Utworzona
2025-06-17T14:00:00	60 minut	Marcin Kolarczyk	75111547838	Konsultacja kardiologa	Utworzona

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.

```

<div>
  <div class="header">
    <h1>Lekarze</h1>
    <button mat-icon-button class="example-icon favorite-icon" (click)="create()">
      <mat-icon>add_circle</mat-icon>
    </button>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="dataSource" class="mat-elevation-z8 demo-table">

        <ng-container matColumnDef="pesel">
          <th mat-header-cell *matHeaderCellDef> PESEL </th>
          <td mat-cell *matCellDef="let element"> {{element.pesel}} </td>
        </ng-container>

        <ng-container matColumnDef="email">
          <th mat-header-cell *matHeaderCellDef> Email </th>
          <td mat-cell *matCellDef="let element"> {{element.email}} </td>
        </ng-container>

        <ng-container matColumnDef="name">
          <th mat-header-cell *matHeaderCellDef> Imię </th>
          <td mat-cell *matCellDef="let element"> {{element.name}} </td>
        </ng-container>

        <ng-container matColumnDef="surname">
          <th mat-header-cell *matHeaderCellDef> Nazwisko </th>
          <td mat-cell *matCellDef="let element"> {{element.surname}} </td>
        </ng-container>

        <ng-container matColumnDef="specialties">
          <th mat-header-cell *matHeaderCellDef> Specjalizacje </th>
          <td mat-cell *matCellDef="let element"> {{ getNames(element.specialties) }} </td>
        </ng-container>

        <ng-container matColumnDef="action">
          <th mat-header-cell *matHeaderCellDef> </th>
          <td mat-cell *matCellDef="let element" class="action-link">
            <button (click)="edit(element)" mat-icon-button class="example-icon favorite-icon">
              <mat-icon>edit</mat-icon>
            </button>
            <button (click)="delete(element)" mat-icon-button class="example-icon favorite-icon">
              <mat-icon>delete</mat-icon>
            </button>
          </td>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
        <tr (click)="selectDoctor(row)" [class.selected]="selectedDoctor?.uuid == row.uuid">
          <td mat-rowDef="let row; columns: displayedColumns;"></td>
        </tr>
      </table>
    </mat-card-content>
  </mat-card>

```

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

```

  delete(doctor: Doctor): void {
    this.doctorApiService.deleteDoctor(doctor.uuid).subscribe(res => {
      | this.refreshable();
    })
  }

  refreshable(): void {
    this.doctorApiService.getAllDoctors().subscribe(data => {
      | this.dataSource = new MatTableDataSource<Doctor>(data);
    })
  }

  getNames(specialties: Specialty[]): string {
    return specialties.map(s => s.name).join(', ')
  }

  selectDoctor(doctor: Doctor): void {
    this.selectedDoctor = doctor;
    this.doctorApiService.getAppointments(doctor.uuid).subscribe(data => {
      | this.dataSourceAppointments = new MatTableDataSource<Appointment>(data);
    })
  }
}

```

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

```

<div *ngIf="selectedDoctor != null">
  <div class="header">
    <h2>Wizyty z pacjentami - Dr {{selectedDoctor.name}} {{selectedDoctor.surname}}</h2>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="dataSourceAppointments" class="mat-elevation-z8 demo-table">

        <ng-container matColumnDef="date">
          <th mat-header-cell *matHeaderCellDef> Data </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.dateTimeFrom}} minut </td>
        </ng-container>

        <ng-container matColumnDef="timeInMinutes">
          <th mat-header-cell *matHeaderCellDef> Czas wizyty </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.service.timeInMinutes}} minut </td>
        </ng-container>

        <ng-container matColumnDef="patient">
          <th mat-header-cell *matHeaderCellDef> Pacjent </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.patient.name}} {{appointment.patient.surname}}</td>
        </ng-container>

        <ng-container matColumnDef="patientPESEL">
          <th mat-header-cell *matHeaderCellDef> PESEL Pacjenta </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.patient.pesel}}</td>
        </ng-container>

        <ng-container matColumnDef="service">
          <th mat-header-cell *matHeaderCellDef> Usługa </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.service.name}}</td>
        </ng-container>

        <ng-container matColumnDef="status">
          <th mat-header-cell *matHeaderCellDef> Status </th>
          <td mat-cell *matCellDef="let appointment"> {{translateStatus(appointment.status)}}</td>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="displayedColumnsAppointment"></tr>
        <tr mat-row *matRowDef="let row; columns: displayedColumnsAppointment;"></tr>
      </table>
    </mat-card-content>
  </mat-card>
</div>

```

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.

```

create(): void {
  const dialogRef = this.dialog.open(DoctorFormComponent, {
    data: {},
    height: '500px',
    width: '600px',
  });

  dialogRef.afterClosed().subscribe(result => {
    if (result !== undefined) {
      this.doctorApiService.createDoctor(result.doctor as DoctorRequest).subscribe(res => {
        result.specialties.forEach((s: Specialty) => {
          this.doctor ApiService.createDoctorSpecialty(result.doctor.uuid, {specialtyUuid: s.uuid} as DoctorSpecialtyRequest).subscribe(a => {})
        })
      })
      this.refreshTable();
    }
  });
}

edit(doctor: Doctor): void {
  const dialogRef = this.dialog.open(DoctorFormComponent, {
    data: doctor,
    height: '500px',
    width: '600px',
  });

  dialogRef.afterClosed().subscribe(result => {
    if (result !== undefined) {
      let resultUuids = result.specialties.map((s: Specialty) => s.uuid)
      let oldUuids = doctor.specialties.map((s: Specialty) => s.uuid)
      let specialtiesToBeCreated = result.specialties.filter((s: Specialty) => !oldUuids.includes(s.uuid))
      let specialtiesToBeRemoved = doctor.specialties.filter((s: Specialty) => !resultUuids.includes(s.uuid))
      specialtiesToBeCreated.forEach((s: Specialty) => {
        this.doctor ApiService.createDoctorSpecialty(doctor.uuid, {specialtyUuid: s.uuid} as DoctorSpecialtyRequest).subscribe(a => {})
      })
      specialtiesToBeRemoved.forEach((s: Specialty) => {
        this.doctor ApiService.deleteDoctorSpecialty(doctor.uuid, s.uuid).subscribe(a => {})
      })
      this.doctor ApiService.updateDoctor(doctor.uuid, result as DoctorRequest).subscribe(res => {
        this.refreshTable();
      })
    }
  });
}

```

Rys. 3.26 Fragment kodu 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.

```

<mat-card appearance="outlined">
  <mat-card-content>
    <mat-form-field>
      <mat-label>Email</mat-label>
      <input matInput [(ngModel)]="doctor.email" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>Imię</mat-label>
      <input matInput [(ngModel)]="doctor.name" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>Nazwisko</mat-label>
      <input matInput [(ngModel)]="doctor.surname" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>PESEL</mat-label>
      <input matInput [(ngModel)]="doctor_pesel" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>Specjalizacje</mat-label>
      <mat-select multiple [(ngModel)]="specialtiesUuids" name="specialty">
        @for (specialty of specialties; track specialty) {
          <mat-option [value]="specialty.uuid">{{specialty.name}}</mat-option>
        }
      </mat-select>
    </mat-form-field>
    <div class="lastRow">
      <button mat-button (click)="cancel()">Anuluj</button>
      <button mat-button (click)="save()">Zapisz</button>
    </div>
  </mat-card-content>
</mat-card>

```

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

```

@Component({
  selector: 'app-doctor-form',
  imports: [
    MatFormFieldModule,
    MatInputModule,
    FormsModule,
    MatButtonModule,
    MatDialogTitle,
    MatDialogContent,
    MatDialogActions,
    MatDialogClose,
    MatCardModule,
    MatSelectModule,
  ],
  templateUrl: './doctor-form.component.html',
  styleUrls: ['./doctor-form.component.css']
})
export class DoctorFormComponent {
  readonly dialogRef = inject(MatDialogRef<DoctorFormComponent>);
  readonly doctor = structuredClone(inject<Doctor>(MAT_DIALOG_DATA));
  specialties: Specialty[] = [];
  specialty ApiService = inject(Specialty ApiService);
  specialtiesUuids: string[] = [];

  ngOnInit(): void {
    this.specialty ApiService.getAllSpecialties().subscribe(data => {
      this.specialties = data;
    })
    this.specialtiesUuids = this.doctor.specialties.map(s => s.uuid)
  }

  cancel(): void {
    this.dialogRef.close();
  }

  save(): void {
    this.doctor.specialties = this.specialties.filter(s => this.specialtiesUuids.includes(s.uuid))
    this.dialogRef.close(this.doctor);
  }
}

```

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

3.1.5. 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.

Great Health	Lekarze	Pacjenci	Specjalizacje	Usługi	Maria Strzecha E+																		
Pacjenci																							
<table><thead><tr><th>Email</th><th>Imię</th><th>Nazwisko</th><th>PESEL</th><th></th><th></th></tr></thead><tbody><tr><td>marcin.kolarczyk@kamilp06@gmail.onmicrosoft.com</td><td>Marcin</td><td>Kolarczyk</td><td>75111547838</td><td></td><td></td></tr><tr><td>aneta.kowalewska@kamilp06@gmail.onmicrosoft.com</td><td>Aneta</td><td>Kowalewska-Malarczyk</td><td>92102183468</td><td></td><td></td></tr></tbody></table>						Email	Imię	Nazwisko	PESEL			marcin.kolarczyk@kamilp06@gmail.onmicrosoft.com	Marcin	Kolarczyk	75111547838			aneta.kowalewska@kamilp06@gmail.onmicrosoft.com	Aneta	Kowalewska-Malarczyk	92102183468		
Email	Imię	Nazwisko	PESEL																				
marcin.kolarczyk@kamilp06@gmail.onmicrosoft.com	Marcin	Kolarczyk	75111547838																				
aneta.kowalewska@kamilp06@gmail.onmicrosoft.com	Aneta	Kowalewska-Malarczyk	92102183468																				

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.

Formularz do dodania pacjenta, składający się z czterech polów:

- Email: marcin.kolarczyk@kamilp06@gmail.onmicrosoft.com
- Imię: Marcin
- Nazwisko: Kolarczyk
- PESEL: 75111547838

Na dole znajdują się dwa przyciski: Anuluj (niebieski) i Zapisz (niebieski).

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

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

```

<div>
  <div class="header">
    <h1>Pacjenci</h1>
    <button mat-icon-button class="example-icon favorite-icon" (click)="create()">
      <mat-icon>add_circle</mat-icon>
    </button>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="dataSource" class="mat-elevation-z8 demo-table">
        <ng-container matColumnDef="pesel">
          <th mat-header-cell *matHeaderCellDef> PESEL </th>
          <td mat-cell *matCellDef="let element"> {{element.pesel}} </td>
        </ng-container>

        <ng-container matColumnDef="email">
          <th mat-header-cell *matHeaderCellDef> Email </th>
          <td mat-cell *matCellDef="let element"> {{element.email}} </td>
        </ng-container>

        <ng-container matColumnDef="name">
          <th mat-header-cell *matHeaderCellDef> Imię </th>
          <td mat-cell *matCellDef="let element"> {{element.name}} </td>
        </ng-container>

        <ng-container matColumnDef="surname">
          <th mat-header-cell *matHeaderCellDef> Nazwisko </th>
          <td mat-cell *matCellDef="let element"> {{element.surname}} </td>
        </ng-container>

        <ng-container matColumnDef="action">
          <th mat-header-cell *matHeaderCellDef> </th>
          <td mat-cell *matCellDef="let element" class="action-link">
            <button mat-icon-button class="example-icon favorite-icon">
              <mat-icon (click)="edit(element)">edit</mat-icon>
            </button>
            <button mat-icon-button class="example-icon favorite-icon">
              <mat-icon (click)="delete(element)">delete</mat-icon>
            </button>
          </td>
        </ng-container>
      </table>
    </mat-card-content>
  </mat-card>
</div>

```

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

```

export class PatientsComponent {
  patient ApiService = inject(Patient ApiService);
  displayedColumns: string[] = ['email', 'name', 'surname', 'pesel', 'action'];
  dataSource: MatTableDataSource<Patient>;
  readonly dialog = inject(MatDialog);

  ngOnInit(): void {
    this.refreshTable();
  }

  create(): void {
    const dialogRef = this.dialog.open(PatientFormComponent, {
      data: {},
      height: '400px',
      width: '600px',
    });

    dialogRef.afterClosed().subscribe(result => {
      if (result !== undefined) {
        this.patient ApiService.createPatient(result as PatientRequest).subscribe(res => {
          this.refreshTable();
        })
      }
    });
  }

  edit(patient: Patient): void {
    const dialogRef = this.dialog.open(PatientFormComponent, {
      data: patient,
      height: '400px',
      width: '600px',
    });
    dialogRef.afterClosed().subscribe(result => {
      if (result !== undefined) {
        this.patient ApiService.updatePatient(patient.uuid, result as PatientRequest).subscribe(res => {
          this.refreshTable();
        })
      }
    });
  }

  delete(patient: Patient): void {
    this.patient ApiService.deletePatient(patient.uuid).subscribe(res => {
      this.refreshTable();
    })
  }

  refreshTable(): void {
    this.patient ApiService.getAllPatients().subscribe(data => {
      this.dataSource = new MatTableDataSource<Patient>(data);
    })
  }
}

```

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.

```
<mat-card appearance="outlined">
  <mat-card-content>
    <mat-form-field>
      <mat-label>Email</mat-label>
      <input matInput [(ngModel)]="patient.email" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>Imię</mat-label>
      <input matInput [(ngModel)]="patient.name" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>Nazwisko</mat-label>
      <input matInput [(ngModel)]="patient.surname" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>PESEL</mat-label>
      <input matInput [(ngModel)]="patient_pesel" />
    </mat-form-field>
    <div class="lastRow">
      <button mat-button (click)="cancel()">Anuluj</button>
      <button mat-button (click)="save()">Zapisz</button>
    </div>
  </mat-card-content>
</mat-card>
```

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

```
export class PatientFormComponent {
  readonly dialogRef = inject(MatDialogRef<PatientFormComponent>);
  readonly patient = structuredClone(inject<Patient>(MAT_DIALOG_DATA));
  cancel(): void {
    this.dialogRef.close();
  }

  save(): void {
    this.dialogRef.close(this.patient);
  }
}
```

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

3.1.6. Podstrona Specjalizacje

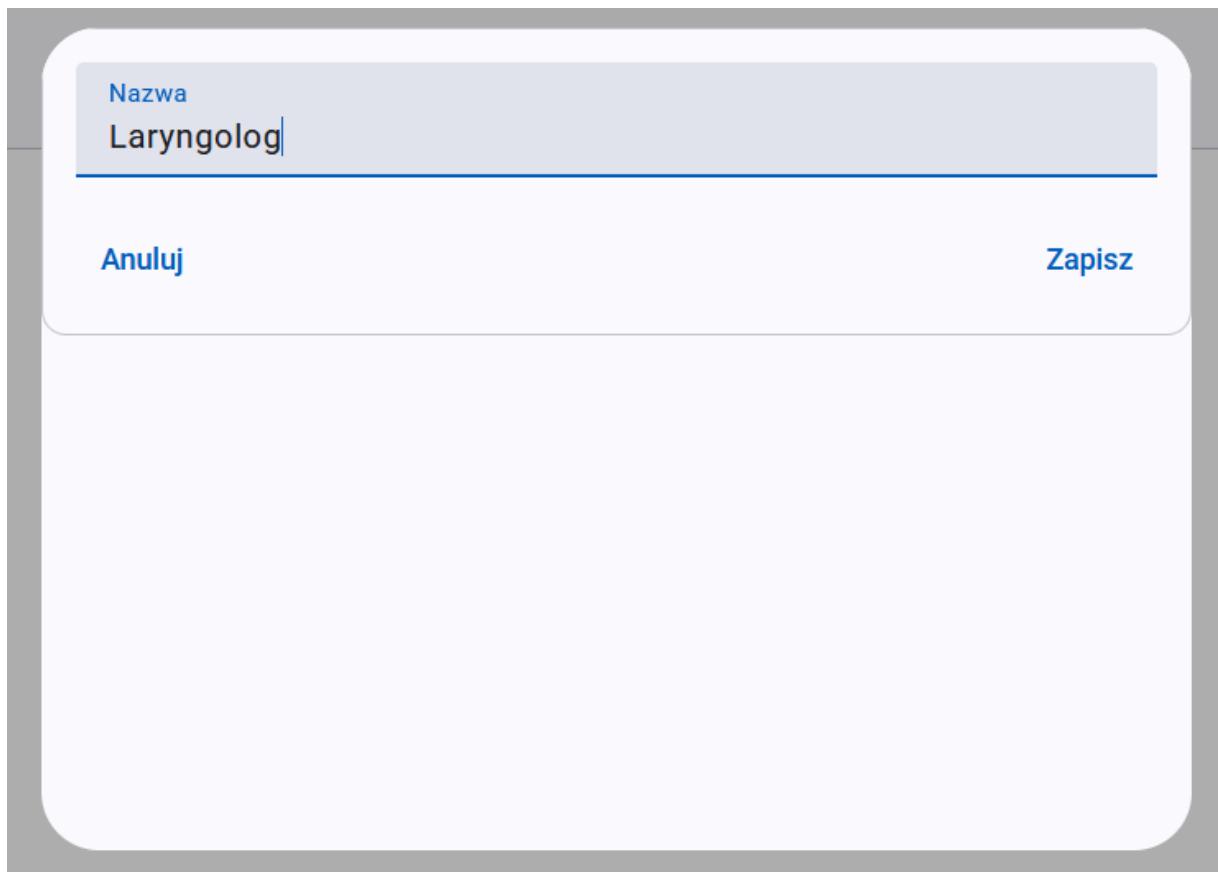
Podstrona Specjalizacje dostępna jest tylko i wyłącznie dla użytkowników będących recepcjonistami. Służy ona do zarządzania danymi dot. specjalizacji. Rysunek nr 3.35 przedstawia wygląd tej podstrony. Zawiera on tabelę przedstawiającą specjalizacje wraz z możliwością ich usunięcia (ikona kosza) i dodania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

Specjalizacje

Nazwa	
Kardiolog	<input type="button" value="■"/>
Internista	<input type="button" value="■"/>

Rys. 3.35 Wygląd podstrony Specjalizacje (opracowanie własne)

Po naciśnięciu przycisku dodawania nowej specjalizacji wyświetlane 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 wywoływanie po naciśnięciu dostępnych przycisków – funkcja *create* oraz *delete*.

```

<div>
  <div class="header">
    <h1>Specjalizacje</h1>
    <button mat-icon-button class="example-icon favorite-icon" (click)="create()">
      <mat-icon>add_circle</mat-icon>
    </button>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="" class="mat-elevation-z8 demo-table">
        <ng-container matColumnDef="name">
          <th mat-header-cell *matHeaderCellDef> Nazwa </th>
          <td mat-cell *matCellDef="let element"> {{element.name}} </td>
        </ng-container>

        <ng-container matColumnDef="action">
          <th mat-header-cell *matHeaderCellDef> </th>
          <td mat-cell *matCellDef="let element" class="action-link">
            <div class="action">
              <button mat-icon-button class="example-icon favorite-icon">
                <mat-icon (click)="delete(element)">delete</mat-icon>
              </button>
            </div>
          </td>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
        <tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
      </table>
    </mat-card-content>
  </mat-card>
</div>

```

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

```

@Directive({
  selector: 'app-specialties',
  imports: [MatTableModule, CommonModule, MatCardModule, MatIconModule, MatButtonModule],
  templateUrl: './specialties.component.html',
  styleUrls: ['./specialties.component.css'],
})
export class SpecialtiesComponent {
  specialty ApiService = inject(Specialty ApiService);
  displayedColumns: string[] = ['name', 'action'];
  dataSource: MatTableDataSource<Specialty>;
  readonly dialog = inject(MatDialog);

  ngOnInit(): void {
    this.refreshTable();
  }

  create(): void {
    const dialogRef = this.dialog.open(SpecialtyFormComponent, {
      data: {},
      height: '400px',
      width: '600px',
    });

    dialogRef.afterClosed().subscribe(result => {
      if (result !== undefined) {
        this.specialty ApiService.createSpecialty(result as SpecialtyRequest).subscribe(res => {
          this.refreshTable();
        });
      }
    });
  }

  delete(specialty: Specialty): void {
    this.specialty ApiService.deleteSpecialty(specialty.uuid).subscribe(res => {
      this.refreshTable();
    });
  }

  refreshTable(): void {
    this.specialty ApiService.getAllSpecialties().subscribe(data => {
      this.dataSource = new MatTableDataSource<Specialty>(data);
    });
  }
}

```

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

3.1.7. 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.

Usługi

Nazwa	Czas trwania usługi w minutach	Wymagana specjalizacja
Konsultacja kardiologa	60	Kardiolog
Konsultacja internisty	30	Internista
Operacja serca	120	Kardiolog

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 wyświetcone zostaje nowe okno służące do tworzenia nowej usługi. Wygląd tego okna został przedstawiony na rysunku nr 3.40.

Nazwa	Konsultacja laryngologa
Czas trwania usługi w minutach	30
Wymagana specjalizacja	Laryngolog

Anuluj Zapisz

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.

```

<div>
  <div class="header">
    <h1>Uslugi</h1>
    <button mat-icon-button class="example-icon favorite-icon" (click)="create()">
      <mat-icon>add_circle</mat-icon>
    </button>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="dataSource" class="mat-elevation-z8 demo-table">

        <ng-container matColumnDef="name">
          <th mat-header-cell *matHeaderCellDef> Nazwa </th>
          <td mat-cell *matCellDef="let element"> {{element.name}} </td>
        </ng-container>

        <ng-container matColumnDef="timeInMinutes">
          <th mat-header-cell *matHeaderCellDef> Czas trwania usługi w minutach </th>
          <td mat-cell *matCellDef="let element"> {{element.timeInMinutes}} </td>
        </ng-container>

        <ng-container matColumnDef="specialty">
          <th mat-header-cell *matHeaderCellDef> Wymagana specjalizacja </th>
          <td mat-cell *matCellDef="let element"> {{element.specialty.name}} </td>
        </ng-container>

        <ng-container matColumnDef="action">
          <th mat-header-cell *matHeaderCellDef> </th>
          <td mat-cell *matCellDef="let element" class="action-link">
            <button mat-icon-button class="example-icon favorite-icon">
              <mat-icon (click)="delete(element)">delete</mat-icon>
            </button>
          </td>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
        <tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
      </table>
    </mat-card-content>
  </mat-card>
</div>

```

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

```

@Component({
  selector: 'app-services',
  imports: [MatTableModule, CommonModule, MatCardModule, MatIconModule, MatButtonModule],
  templateUrl: './services.component.html',
  styleUrls: ['./services.component.css']
})
export class ServicesComponent {
  serviceApiService = inject(Service ApiService);
  displayedColumns: string[] = ['name', 'timeInMinutes', 'specialty', 'action'];
  dataSource: MatTableDataSource<Service>;
  readonly dialog = inject(MatDialog);

  ngOnInit(): void {
    this.refreshTable();
  }

  create(): void {
    const dialogRef = this.dialog.open(ServiceFormComponent, {
      data: {},
      height: '400px',
      width: '600px',
    });

    dialogRef.afterClosed().subscribe(result => {
      if (result !== undefined) {
        const service = result as Service;
        this.serviceApiService.createService([
          { name: service.name, timeInMinutes: service.timeInMinutes, specialtyUuid: service.specialty.uuid }
        ] as ServiceRequest).subscribe(res => {
          this.refreshTable();
        });
      }
    });
  }

  delete(service: Service): void {
    this.serviceApiService.deleteService(service.uuid).subscribe(res => {
      this.refreshTable();
    });
  }

  refreshTable(): void {
    this.serviceApiService.getAllServices().subscribe(data => {
      this.dataSource = new MatTableDataSource<Service>(data);
    });
  }
}

```

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

```

<mat-card appearance="outlined">
  <mat-card-content>
    <mat-form-field>
      <mat-label>Nazwa</mat-label>
      <input matInput [(ngModel)]="service.name" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>Czas trwania usługi w minutach</mat-label>
      <input type="number" matInput [(ngModel)]="service.timeInMinutes" />
    </mat-form-field>
    <mat-form-field>
      <mat-label>Wymagana specjalizacja</mat-label>
      <mat-select [(ngModel)]="service.specialty" name="specialty">
        @for (specialty of specialties; track specialty) {
          <mat-option [value]="specialty">{{specialty.name}}</mat-option>
        }
      </mat-select>
    </mat-form-field>
    <div class="lastRow">
      <button mat-button (click)="cancel()">Anuluj</button>
      <button mat-button (click)="save()">Zapisz</button>
    </div>
  </mat-card-content>
</mat-card>

```

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

```

@Component({
  selector: 'app-service-form',
  imports: [
    MatFormFieldModule,
    MatInputModule,
    FormsModule,
    MatButtonModule,
    MatDialogTitle,
    MatDialogContent,
    MatDialogActions,
    MatDialogClose,
    MatDialogModule,
    MatSelectModule,
  ],
  templateUrl: './service-form.component.html',
  styleUrls: ['./service-form.component.css']
})
export class ServiceFormComponent {
  specialties: Specialty[];
  specialty ApiService = inject(Specialty ApiService);
  readonly dialogRef = inject(MatDialogRef<ServiceFormComponent>);
  readonly service = structuredClone(inject<Service>(MAT_DIALOG_DATA));

  ngOnInit(): void {
    this.specialty ApiService.getAllSpecialties().subscribe(data => {
      this.specialties = data;
    })
  }

  cancel(): void {
    this.dialogRef.close();
  }

  save(): void {
    this.dialogRef.close(this.service);
  }
}

```

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*.

3.1.8. 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.

Data	Czas wizyty	Doctor	Usługa	Status
2025-06-17T19:30:00	30 minut	Zuzanna Skowrońska-Kulczyk	Konsultacja internisty	Zakonficzona
2025-06-17T13:30:00	30 minut	Zuzanna Skowrońska-Kulczyk	Konsultacja internisty	Anulowana
2025-06-17T12:00:00	60 minut	Zuzanna Skowrońska-Kulczyk	Konsultacja kardiologa	Anulowana
2025-06-17T15:00:00	60 minut	Zuzanna Skowrońska-Kulczyk	Konsultacja kardiologa	Anulowana
2025-06-17T14:00:00	60 minut	Zuzanna Skowrońska-Kulczyk	Konsultacja kardiologa	Anulowana
2025-05-04T10:00:00	120 minut	Adam Kowalski	Operacja serca	Anulowana
2025-07-06T10:00:00	120 minut	Zuzanna Skowrońska-Kulczyk	Operacja serca	Utworzona

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 okno służące do tworzenia nowych wizyt. Wygląd tego okna został zaprezentowany na rysunku nr 3.46.

Dzień
6.07.2025

Usługa
Konsultacja kardiologa

Doktor
Zuzanna Skowrońska-Kulczyk

Czas wizyty
2025-07-06T15:00:00

Anuluj Zapisz

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 wyświetlni 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*.

```
<div *ngIf="loggedInPatientUuid != null">
  <div class="header">
    <h1>Wizyty</h1>
    <button mat-icon-button class="example-icon favorite-icon" (click)="create()">
      <mat-icon>add_circle</mat-icon>
    </button>
  </div>
  <mat-card appearance="outlined">
    <mat-card-content>
      <table mat-table [dataSource]="dataSourceAppointments" class="mat-elevation-z8 demo-table">

        <ng-container matColumnDef="date">
          <th mat-header-cell *matHeaderCellDef> Data </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.dateTimeFrom}} minut </td>
        </ng-container>

        <ng-container matColumnDef="timeInMinutes">
          <th mat-header-cell *matHeaderCellDef> Czas wizyty </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.service.timeInMinutes}} minut </td>
        </ng-container>

        <ng-container matColumnDef="doctor">
          <th mat-header-cell *matHeaderCellDef> Lekarz </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.doctor.name}} {{appointment.doctor.surname}}</td>
        </ng-container>

        <ng-container matColumnDef="service">
          <th mat-header-cell *matHeaderCellDef> Usluga </th>
          <td mat-cell *matCellDef="let appointment"> {{appointment.service.name}} </td>
        </ng-container>

        <ng-container matColumnDef="status">
          <th mat-header-cell *matHeaderCellDef> Status </th>
          <td mat-cell *matCellDef="let appointment"> {{translateStatus(appointment.status)}} </td>
        </ng-container>

        <ng-container matColumnDef="action">
          <th mat-header-cell *matHeaderCellDef> </th>
          <td mat-cell *matCellDef="let element" class="action-link">
            <button *ngIf="element.status == 'CREATED'" (click)="cancelAppointment(element)" mat-icon-button class="example-icon favorite-icon">
              <mat-icon>close</mat-icon>
            </button>
          </td>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="displayedColumnsAppointment"></tr>
        <tr mat-row *matRowDef="let row; columns: displayedColumnsAppointment;"></tr>
      </table>
    </mat-card-content>
  </mat-card>
</div>
```

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

```

export class PatientAppointmentsComponent {
  patient ApiService = inject(Patient ApiService)
  user ApiService = inject(User ApiService)
  readonly dialog = inject(MatDialog);
  displayedColumnsAppointment: string[] = ['date', 'timeInMinutes', 'doctor', 'service', 'status', 'action'];
  dataSourceAppointments: MatTableDataSource<Appointment>;
  loggedInPatientUuid: string;

  ngOnInit(): void {
    this.user ApiService.getCurrentUserUuid().subscribe(patientUuid => {
      this.loggedInPatientUuid = patientUuid.uuid;
      this.refreshTable();
    })
  }
  cancelAppointment(appointment: Appointment) {
    if (appointment.status == 'CREATED') {
      this.patient ApiService.cancelAppointment(this.loggedInPatientUuid, appointment.uuid).subscribe(a => {
        this.refreshTable();
      })
    }
  }
  create(): void {
    const dialogRef = this.dialog.open(PatientAppointmentFormComponent, {
      data: {},
      height: '400px',
      width: '600px',
    });
    dialogRef.afterClosed().subscribe(result => {
      if (result != undefined) {
        this.patient ApiService.createAppointment(this.loggedInPatientUuid, result as AppointmentRequest).subscribe(res => {
          this.refreshTable();
        })
      }
    });
  }
  refreshTable(): void {
    this.patient ApiService.getAppointments(this.loggedInPatientUuid).subscribe(data => {
      this.dataSourceAppointments = new MatTableDataSource<Appointment>(data);
    })
  }
  translateStatus(status: string): string {
    if (status == 'CREATED')
      return 'Utworzona';
    else if (status == 'CANCELED')
      return 'Anulowana';
    else if (status == 'FINISHED')
      return 'Zakończona';
    return '';
  }
}

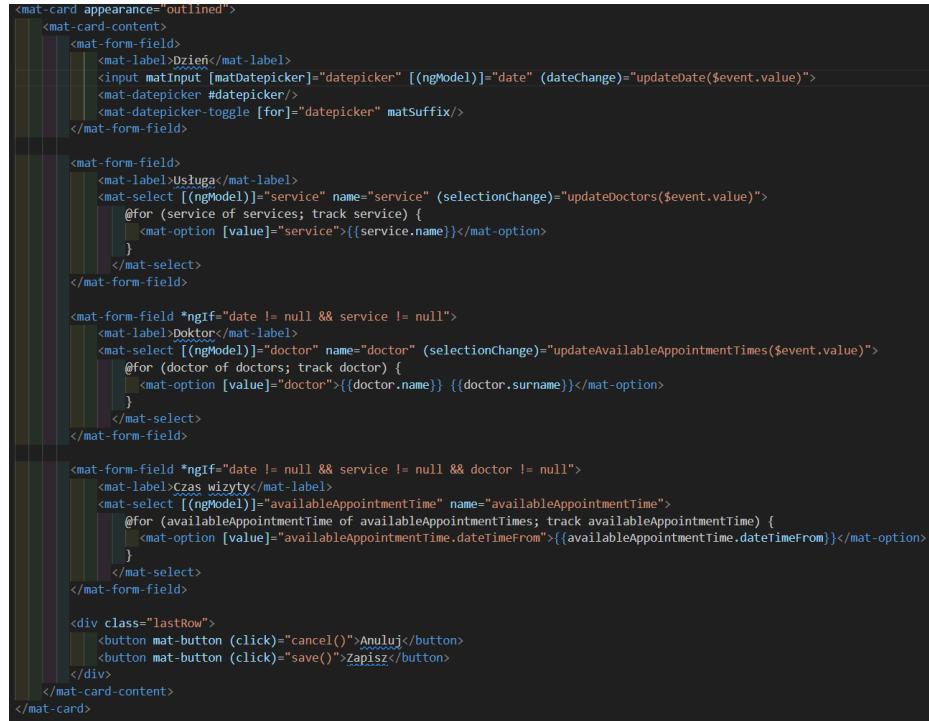
```

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

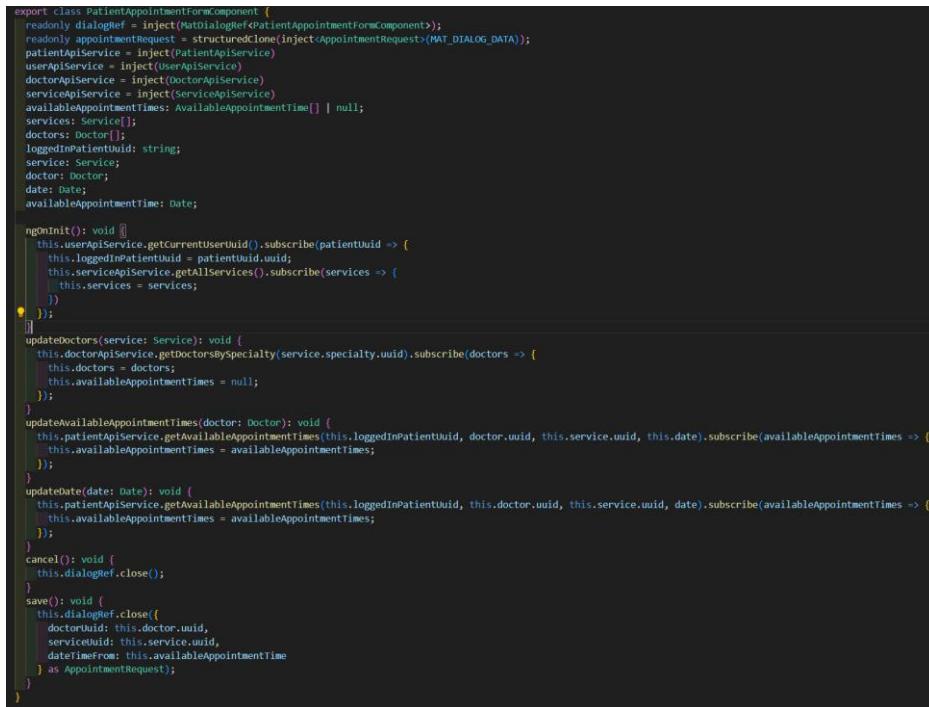
Warstwa prezentacji definiuje tabelę wyświetlającą dane wizyty zalogowanej użytkownika. Ponadto, wyświetla dwa przyciski, gdzie pierwszy służy do anulowania wizyty, co skutkuje wywołaniem funkcji *cancelAppointment*, zaś drugi służy do umawiania nowej wizyty, co z kolei skutkuje wywołaniem funkcji *create*. Funkcja ta tworzy nowe okno, które umożliwia umówienie nowej wizyty, z wykorzystaniem serwisu z framework'a *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)

3.1.9. Klient API Backendu

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

```
@Injectable({providedIn: 'root'})
export class Specialty ApiService {
  constructor(private http: HttpClient) { }

  createSpecialty(specialtyRequest: SpecialtyRequest): Observable<Specialty> {
    return this.http.post<Specialty>('api/specialties', specialtyRequest);
  }

  getSpecialtyById(specialtyUuid: string): Observable<Specialty> {
    return this.http.get<Specialty>('api/specialties/' + specialtyUuid);
  }

  deleteSpecialty(specialtyUuid: string): Observable<void> {
    return this.http.delete<void>('api/specialties/' + specialtyUuid);
  }

  getAllSpecialties(): Observable<Specialty[]> {
    return this.http.get<Specialty[]>('api/specialties');
  }
}
```

Rys. 3.51 Kod źródłowy serwisu *Specialty ApiService* (opracowanie własne)

Funkcje zdefiniowane w serwisie *Specialty ApiService* 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.

```
export interface SpecialtyRequest {
  name: string;
}
```

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.

```
export interface Specialty {
  uuid: string;
  name: string;
}
```

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/specalties/{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 żądania. Za realizację tego zadania odpowiedzialny jest serwis o nazwie `TokenInterceptorService`, którego definicja została przedstawiona na rysunku nr 3.54.

```
@Injectable({providedIn: 'root'})
export class TokenInterceptorService implements HttpInterceptor {
  constructor(private oidcSecurityService: OidcSecurityService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    console.log("hello")
    return this.oidcSecurityService.getIdToken().pipe(
      switchMap(token => {
        if (token) {
          req = req.clone({
            setHeaders: {
              Authorization: `Bearer ${token}`
            }
          });
        }
        return next.handle(req);
      })
    );
  }
}
```

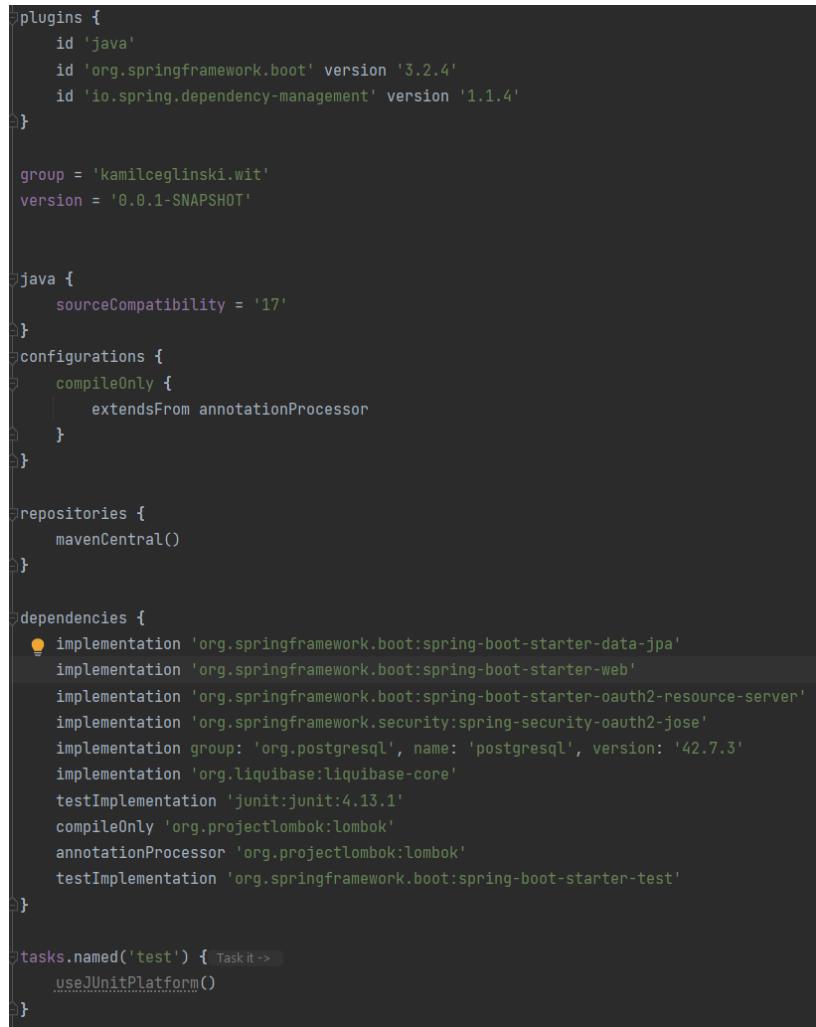
Rys. 3.54 Kod źródłowy serwisu `TokenInterceptorService` (opracowanie własne)

3.2. Implementacja backendu

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

3.2.1. Konfiguracja

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



```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.2.4'
    id 'io.spring.dependency-management' version '1.1.4'
}

group = 'kamilceglinski.wit'
version = '0.0.1-SNAPSHOT'

java {
    sourceCompatibility = '17'
}
configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
    implementation 'org.springframework.security:spring-security-oauth2-jose'
    implementation group: 'org.postgresql', name: 'postgresql', version: '42.7.3'
    implementation 'org.liquibase:liquibase-core'
    testImplementation 'junit:junit:4.13.1'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') { Task it ->
    useJUnitPlatform()
}
```

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.

```

spring:
  application:
    name: greathealth
  datasource:
    url: ${DB_URL}
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
  jpa:
    hibernate:
      ddl-auto: validate
      dialect: org.hibernate.dialect.PostgreSQLDialect
  liquibase:
    change-log: classpath:/db/changelog/db.changelog-master.xml
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://login.microsoftonline.com/2ab831ab-3873-4eec-903e-159969a8b507/v2.0
          jwk-set-uri: https://login.windows.net/common/discovery/keys
  logging:
    level:
      org:
        springframework: DEBUG

```

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.

```

no usages  ▲ Kamil Cegliński
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

  no usages  ▲ Kamil Cegliński
  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
      .authorizeHttpRequests(authorize -> authorize
        .anyRequest().authenticated()
      )
      .oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()))
      .cors().disable();
    return http.build();
  }
}

```

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.

```

2 usages ▲ Kamil Cegiński
@Service
public class SecurityService {

    no usages ▲ Kamil Cegiński
    public boolean isAdmin(Authentication authentication) {
        return getGroups(authentication).contains("57057a39-a18f-4aed-bea8-c0f661ff12f0");
    }

    3 usages ▲ Kamil Cegiński
    public boolean isDoctor(Authentication authentication) {
        return getGroups(authentication).contains("2350f29d-627c-4bd0-b554-e757825ba21c");
    }

    1 usage ▲ Kamil Cegiński
    public boolean isPatient(Authentication authentication) {
        return getGroups(authentication).contains("973529ee-6146-4189-b7ef-21ea5f38eef2");
    }

    3 usages ▲ Kamil Cegiński
    public List<String> getGroups(Authentication authentication) {
        return ((JwtAuthenticationToken) authentication).getToken().getClaims().getOrDefault("groups", List.of());
    }

    1 usage ▲ Kamil Cegiński
    public String getEmail(Authentication authentication) {
        return ((String) ((JwtAuthenticationToken) authentication).getToken().getClaims().get("preferred_username"));
    }
}

```

Rys. 3.58 Serwis SecurityService.java (opracowanie własne)

```

17 usages ▲ Kamil Cegiński
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@PreAuthorize("@securityService.isAdmin(authentication)")
public @interface IsAdmin {
}

```

Rys. 3.59 Adnotacja IsAdmin (opracowanie własne)

```

4 usages ▲ Kamil Cegiński
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@PreAuthorize("@securityService.isAdmin(authentication) || @securityService.isDoctor(authentication)")
public @interface IsAdminOrDoctor {
}

```

Rys. 3.60 Adnotacja IsAdminOrDoctor (opracowanie własne)

```

2 usages ▲ Kamil Cegiński
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@PreAuthorize("@securityService.isAdmin(authentication) || @securityService.isPatient(authentication)")
public @interface IsAdminOrPatient {
}

```

Rys. 3.61 Adnotacja IsAdminOrPatient (opracowanie własne)

```

5 usages ▲ Kamil Cegiński
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@PreAuthorize("@securityService.isDoctor(authentication)")
public @interface IsDoctor {
}

```

Rys. 3.62 Adnotacja IsDoctor (opracowanie własne)

```

  2 usages ▾ Kamil Ceglinski
  ↳ @Target({ ElementType.METHOD, ElementType.TYPE })
    @Retention(RetentionPolicy.RUNTIME)
    @Inherited
    @Documented
    △ @PreAuthorize("@securityService.isDoctor(authentication) || @securityService.isPatient(authentication)")
    public @interface IsDoctorOrPatient {
    }

```

Rys. 3.63 Adnotacja IsDoctorOrPatient (opracowanie własne)

```

  5 usages ▾ Kamil Ceglinski
  ↳ @Target({ ElementType.METHOD, ElementType.TYPE })
    @Retention(RetentionPolicy.RUNTIME)
    @Inherited
    @Documented
    △ @PreAuthorize("@securityService.isPatient(authentication)")
    public @interface IsPatient {
    }

```

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.

```

<?xml version="1.0" encoding="UTF-8"?>
<!databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
                      xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
                                         http://www.liquibase.org/xml/ns/dbchangelog.xsd">
  <changeSet id="001_init" author="kamilceglinski">

    <createTable tableName="patient">
      <column name="uid" type="varchar">
        <constraints primaryKey="true" nullable="false" />
      </column>
      <column name="email" type="varchar">
        <constraints nullable="false" unique="true" />
      </column>
      <column name="name" type="varchar">
        <constraints nullable="false" />
      </column>
      <column name="surname" type="varchar">
        <constraints nullable="false" />
      </column>
      <column name="pesel" type="varchar">
        <constraints nullable="false" />
      </column>
    </createTable>

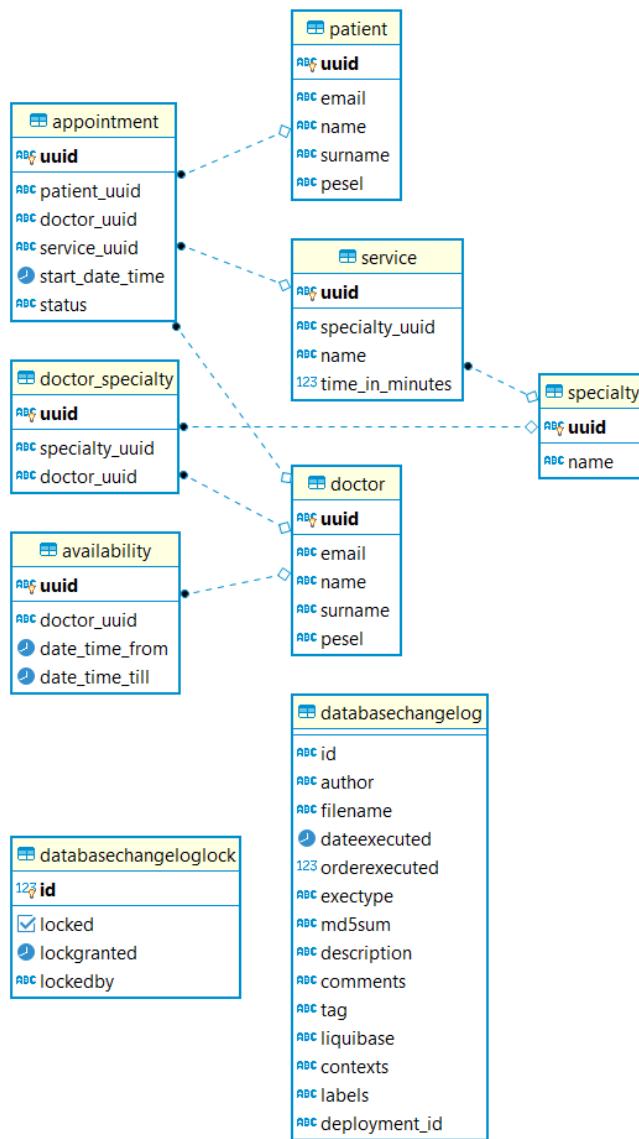
    <createTable tableName="doctor">
      <column name="uid" type="varchar">
        <constraints primaryKey="true" nullable="false" />
      </column>
      <column name="email" type="varchar">
        <constraints nullable="false" unique="true" />
      </column>
      <column name="name" type="varchar">
        <constraints nullable="false" />
      </column>
      <column name="surname" type="varchar">
        <constraints nullable="false" />
      </column>
      <column name="pesel" type="varchar">
        <constraints nullable="false" />
      </column>
    </createTable>
  </changeSet>

```

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

3.2.2. Schemat bazy danych

Skrypt *Liquibase* o nazwie *001_init.xml*, którego część przedstawiona jest na rysunku 3.65, zawiera definicję schematu bazy danych. Rysunek 3.66 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 wykorzystywanyimi przez *Liquibase* w celu zapewnienia, że każdy skrypt wykonywany jest tylko raz w celu otrzymania zadanego stanu na bazie danych.



Rys. 3.66 Schemat bazy danych (opracowanie własne)

3.2.3. Omówienie REST API - /api/doctors

Klasa *DoctorController*, której część została przedstawiona na rysunku nr 3.67, 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}

```
no usages ▲ Kamil Cegiński
@IsAdmin
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public DoctorResponseDTO createDoctor(@RequestBody DoctorRequestDTO requestDTO) {
    return doctorService.createDoctor(requestDTO);
}

no usages ▲ Kamil Cegiński
@IsAdmin
@PostMapping("/{uuid}/specialties")
@ResponseStatus(HttpStatus.CREATED)
public DoctorSpecialtyResponseDTO createDoctorSpecialty(@RequestBody DoctorSpecialtyRequestDTO requestDTO,
    @PathVariable String uuid) {
    return doctorService.createDoctorSpecialty(requestDTO, uuid);
}

no usages ▲ Kamil Cegiński
@IsAdmin
@DeleteMapping("/{uuid}/specialties/{specialtyUuid}")
@ResponseStatus(HttpStatus.CREATED)
public void deleteDoctorSpecialty(@PathVariable String uuid, @PathVariable String specialtyUuid) {
    doctorService.deleteDoctorSpecialty(uuid, specialtyUuid);
}

no usages ▲ Kamil Cegiński
@IsAdmin
@PutMapping("/{uuid}")
@ResponseStatus(HttpStatus.CREATED)
public DoctorResponseDTO updateDoctor(@RequestBody DoctorRequestDTO requestDTO, @PathVariable String uuid) {
    return doctorService.updateDoctor(requestDTO, uuid);
}

no usages ▲ Kamil Cegiński
@IsAdmin
@DeleteMapping("/{uuid}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteDoctor(@PathVariable String uuid) { doctorService.deleteDoctor(uuid); }

no usages ▲ Kamil Cegiński
@GetMapping("/{uuid}")
@ResponseStatus(HttpStatus.OK)
public DoctorResponseDTO getDoctorById(@PathVariable String uuid) { return doctorService.getDoctorByUuid(uuid); }
```

Rys. 3.67 Część kodu źródłowego klasy *DoctorController* (opracowanie własne)

Adnotacje pomocnicze są wykorzystywane do oznaczenia metod definiujących endpointy, aby te zostały zabezpieczone. Na przykładzie widocznym na rysunku nr 3.67 można zauważyć adnotację *IsAdmin*, która powoduje, że dany endpoint może zostać wykonany tylko i wyłącznie przez użytkowników z rolą admin, czyli będących recepcjonistami. Na rysunku nr 3.68 przedstawiona została część kodu klasy *DoctorService*, która jest wykorzystywana przez klasę *DoctorController*. Serwis ten wykorzystuje logikę biznesową dla poszczególnych endpointów. Dla przykładu metoda *createDoctor* przyjmuje obiekt z danymi nowotworzonego lekarza w systemie. W celu jego zapisu, serwis najpierw przekształca go w obiekt nazywany encją, która jest odwzorowaniem jednego rekordu danej tabeli, w tym przypadku tabeli z lekarzami. Dalej, encja ta przekazywana jest do repozytorium, czyli klasy, która służy do operacji na bazie danych na konkretnej tabeli, znów w tym przypadku na tabeli z lekarzami. Kiedy encja ta zostanie już zapisana, jest aktualizowana o identyfikator bazodanowy. Wówczas taka zaktualizowana encja zostaje przekształcona w obiekt zwracany przez endpoint.



```

@Service
public class DoctorService {

    private final DoctorMapper doctorMapper;
    private final DoctorRepository doctorRepository;
    private final SpecialtyRepository specialtyRepository;
    private final DoctorSpecialtyRepository doctorSpecialtyRepository;
    private final DoctorSpecialtyMapper doctorSpecialtyMapper;
    private final AppointmentRepository appointmentRepository;
    private final AppointmentMapper appointmentMapper;

    public DoctorResponseDTO createDoctor(DoctorRequestDTO requestDTO) {
        DoctorEntity doctorEntity = doctorMapper.toDoctorEntity(requestDTO);
        DoctorEntity savedDoctorEntity = doctorRepository.save(doctorEntity);
        return doctorMapper.toDoctorResponseDTO(savedDoctorEntity);
    }

    public DoctorResponseDTO getDoctorByUuid(String uuid) {
        return doctorRepository.findById(uuid).map(doctorMapper::toDoctorResponseDTO)
            .orElseThrow();
    }

    public List<DoctorResponseDTO> getAllDoctors(String specialtyUuid) {
        List<DoctorEntity> doctors;
        if (specialtyUuid != null) {
            doctors = doctorRepository.findBySpecialty(specialtyUuid);
        } else {
            doctors = doctorRepository.findAll();
        }
        return doctors.stream().map(doctorMapper::toDoctorResponseDTO)
            .collect(Collectors.toList());
    }
}

```

Rys. 3.68 Część kodu źródłowego *DoctorService* (opracowanie własne)

3.2.4. Omówienie REST API - /api/doctors/{doctorUuid}/availabilities

Klasa *DoctorsAvailabilityController* przedstawiona na rysunku nr 3.69, definiuje następujące endpointy z przedrostkiem */api/doctors/{doctorUuid}/availabilities*:

- POST /api/doctors/{doctorUuid}/availabilities
- GET /api/doctors/{doctorUuid}/availabilities/{uuid}
- GET /api/doctors/{doctorUuid}/availabilities

```
    @RequestMapping("api/doctors/{doctorUuid}/availabilities")
    public class DoctorsAvailabilityController {

        3 usages
        private final DoctorsAvailabilityService doctorsAvailabilityService;
        3 usages
        private final UserService userService;

        no usages  ▲ Kamil Cegielski
        @IsDoctor
        @PostMapping
        @ResponseStatus(HttpStatus.CREATED)
        public AvailabilityResponseDTO createAvailability(@PathVariable String doctorUuid,
                                                          @RequestBody DoctorsAvailabilityRequestDTO requestDTO,
                                                          Authentication authentication) {
            if (!doctorUuid.equals(userService.getCurrentUserUuid(authentication))) {
                throw new AuthorizationServiceException("Not authorized");
            }
            return doctorsAvailabilityService.createAvailability(doctorUuid, requestDTO);
        }

        no usages  ▲ Kamil Cegielski
        @IsDoctor
        @GetMapping("/{uuid}")
        @ResponseStatus(HttpStatus.OK)
        public AvailabilityResponseDTO getAvailabilityById(@PathVariable String doctorUuid, @PathVariable String uuid,
                                                          Authentication authentication) {
            if (!doctorUuid.equals(userService.getCurrentUserUuid(authentication))) {
                throw new AuthorizationServiceException("Not authorized");
            }
            return doctorsAvailabilityService.getAvailabilityByUuid(doctorUuid, uuid);
        }

        no usages  ▲ Kamil Cegielski
        @IsDoctor
        @GetMapping
        @ResponseStatus(HttpStatus.OK)
        public List<AvailabilityResponseDTO> getAllAvailabilities(@PathVariable String doctorUuid,
                                                               Authentication authentication) {
            if (!doctorUuid.equals(userService.getCurrentUserUuid(authentication))) {
                throw new AuthorizationServiceException("Not authorized");
            }
            return doctorsAvailabilityService.getAllAvailabilities(doctorUuid);
        }
    }
```

Rys. 3.69 Część kodu źródłowego klasy *DoctorsAvailabilityController* (opracowanie własne)

Ponownie wykorzystywane są adnotacje zabezpieczające endpointy, które sprawdzają role przypisane do zalogowanego użytkownika. W tym przypadku wszystkie endpointy zabezpieczone są adnotacją *IsDoctor*, która wymaga, aby użytkownik był lekarzem. Ponadto, warto zauważyć, że każdy z trzech endpointów sprawdza również czy lekarz, którego dotyczy dany endpoint, jest zalogowanym użytkownikiem. W ten sposób system zabezpieczony jest przed możliwą zmianą dostępności jednego lekarza przez innego lekarza. Na rysunku nr 3.70

przedstawiona została część kodu klasy *DoctorsAvailabilityService* odpowiedzialnej za logikę biznesową każdego endpointu. Jedną z metod implementujących jeden z endpointów jest *getAvailabilityUuid*, która wykorzystuje repozytorium o nazwie *AvailabilityRepository* dedykowane operacjom na bazie danych dla tabeli *availability*, gdzie przechowywane są dane dotyczące dostępności lekarzy. Wywołana została metoda o nazwie *findByDoctor_uuidAndUuid*, której nazwa definiuje zapytanie bazodanowe przy użyciu biblioteki *Spring Data JPA*. W ten sposób zwracana jest encja, która następnie jest konwertowana do modelu zwracanego przez endpoint.

```

@Service
public class DoctorsAvailabilityService {

    4 usages
    private final AvailabilityMapper availabilityMapper;
    3 usages
    private final AvailabilityRepository availabilityRepository;
    1 usage
    private final DoctorRepository doctorRepository;

    1 usage ▲ Kamil Cegielski
    public AvailabilityResponseDTO createAvailability(String doctorUuid, DoctorsAvailabilityRequestDTO requestDTO) {
        DoctorEntity doctor = doctorRepository.findById(doctorUuid).orElseThrow();
        AvailabilityEntity availabilityEntity = availabilityMapper.toAvailabilityEntity(doctor, requestDTO);
        AvailabilityEntity savedAvailabilityEntity = availabilityRepository.save(availabilityEntity);
        return availabilityMapper.toAvailabilityResponseDTO(savedAvailabilityEntity);
    }

    1 usage ▲ Kamil Cegielski
    public AvailabilityResponseDTO getAvailabilityByUuid(String doctorUuid, String uuid) {
        return availabilityRepository.findByDoctor_uuidAndUuid(doctorUuid, uuid).Optional<AvailabilityEntity>
            .map(availabilityMapper::toAvailabilityResponseDTO).Optional<AvailabilityResponseDTO>
            .orElseThrow();
    }

    1 usage ▲ Kamil Cegielski
    public List<AvailabilityResponseDTO> getAllAvailabilities(String doctorUuid) {
        return availabilityRepository.findAllByDoctor_uuid(doctorUuid).stream().Stream<AvailabilityEntity>
            .map(availabilityMapper::toAvailabilityResponseDTO).Stream<AvailabilityResponseDTO>
            .collect(Collectors.toList());
    }
}

```

Rys. 3.70 Część kodu źródłowego klasy DoctorsAvailabilityService (opracowanie własne)

3.2.5. Omówienie REST API - /api/patients

Klasa *PatientController*; której część kodu została przedstawiona na rysunku nr 3.71, definiuje następujące endpointy z przedrostkiem */api/patients*:

- POST /api/patients
- PUT /api/patients/{uuid}
- DELETE /api/patients/{uuid}
- GET /api/patients/{uuid}
- GET /api/patients
- POST /api/patients/{uuid}/appointments

- GET /api/patients/{uuid}/available-appointment-times
- GET /api/patients/{uuid}/appointments
- DELETE /api/patients/{uuid}/appointments/{appointmentUuid}

```

@RequestMapping("api/patients")
public class PatientController {

    9 usages
    private final PatientService patientService;
    5 usages
    private final UserService userService;

    no usages ▾ Kamil Cegiński
    @IsAdmin
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public PatientResponseDTO createPatient(@RequestBody PatientRequestDTO requestDTO) {
        return patientService.createPatient(requestDTO);
    }

    no usages ▾ Kamil Cegiński
    @IsAdmin
    @PutMapping("/{uuid}")
    @ResponseStatus(HttpStatus.CREATED)
    public PatientResponseDTO updatePatient(@RequestBody PatientRequestDTO requestDTO, @PathVariable String uuid) {
        return patientService.updatePatient(requestDTO, uuid);
    }

    no usages ▾ Kamil Cegiński
    @IsAdmin
    @DeleteMapping("/{uuid}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deletePatient(@PathVariable String uuid) { patientService.deletePatient(uuid); }

    no usages ▾ Kamil Cegiński
    @IsAdminOrPatient
    @GetMapping("/{uuid}")
    @ResponseStatus(HttpStatus.OK)
    public PatientResponseDTO getPatientById(@PathVariable String uuid,
                                              Authentication authentication) {
        if (!uuid.equals(userService.getCurrentUserUuid(authentication))) {
            throw new AuthorizationServiceException("Not authorized");
        }
        return patientService.getPatientByUuid(uuid);
    }
}

```

Rys. 3.71 Część kodu źródłowego klasy PatientController (opracowanie własne)

Na przedstawionym kodzie na rysunku 3.71, zostały wykorzystane adnotacje *IsAdmin* oraz *IsAdminOrPatient* w celu wymuszenia, aby zalogowany użytkownik był w pierwszym przypadku recepcjonistą, zaś w drugim recepcjonistą lub pacjentem. Rysunek 3.72 przedstawia serwis o nazwie *PatientService*, który jest implementacją logiki biznesowej endpointów dotyczących pacjentów.

```

    usage ▾ Kamil Ceglinski
    public List<AvailableAppointmentTimeResponseDTO> getAvailableAppointmentTimes(String uuid, String doctorUuid,
                                                                                      String serviceUuid, LocalDate date) {
        PatientEntity patientEntity = patientRepository.findById(uuid)
            .orElseThrow();
        DoctorEntity doctorEntity = doctorRepository.findById(doctorUuid)
            .orElseThrow();
        List<SpecialtyEntity> specialties = doctorEntity.getSpecialties().stream()
            .map(DoctorSpecialtyEntity::getSpecialty).collect(Collectors.toList());
        ServiceEntity serviceEntity = serviceRepository.findBySpecialtyInAndUuid(specialties, serviceUuid)
            .orElseThrow();
        int timeInMinutes = serviceEntity.getTimeInMinutes();

        LocalDateTime minLocalDateTime = date.atStartOfDay();
        LocalDateTime maxLocalDateTime = date.plusDays(1).atStartOfDay();
        LocalDateTime currentDateTime = minLocalDateTime;
        List<LocalDateTime> allDateTimesForGivenDay = new ArrayList<>();
        while (currentDateTime.plusMinutes(timeInMinutes).isBefore(maxLocalDateTime)) {
            if (isDateTimeAvailable(doctorEntity, timeInMinutes, currentDateTime)) {
                allDateTimesForGivenDay.add(currentDateTime);
            }
            currentDateTime = currentDateTime.plusMinutes(timeInMinutes);
        }

        return allDateTimesForGivenDay.stream()
            .map(localDateTime -> AvailableAppointmentTimeResponseDTO.builder()
                .doctor(doctorMapper.toDoctorResponseDTO(doctorEntity))
                .service(serviceMapper.toServiceResponseDTO(serviceEntity))
                .dateTimeFrom(localDateTime)
                .build())
            .collect(Collectors.toList());
    }
}

```

Rys. 3.72 Część kodu źródłowego klasy PatientService (opracowanie własne)

Na szczególną uwagę zasługuje implementacja endpointa `GET /api/patients/{uuid}/available-appointment-times`, który zwraca dostępne terminy dostępne dla zalogowanego pacjenta do wybranego lekarza w dany dzień na wykonanie konkretnej usługi. Na początku metoda weryfikuje istnienie zalogowanego pacjenta oraz wskazanego lekarza. Następnie, sprawdzane jest to czy lekarz ten ma wymaganą specjalizację do wykonania wybranej usługi medycznej. Kiedy wspomniane kroki zostaną wykonane, tworzona jest lista wszystkich możliwych czasów wizyty na dany dzień przy użyciu czasu wykonania wybranej usługi. Następnie taka lista zostaje przefiltrowana tak, aby pozostały w niej tylko i wyłącznie wolne godziny dla wybranego lekarza na dany dzień. Filtrem zastosowany w tym celu jest metoda o nazwie `isDateTimeAvailable`, której implementacja została przedstawiona na rysunku nr 3.73. Pierwszym krokiem jest pobranie grafiku dostępności lekarza na dany dzień. Następnie zostają pobrane wszystkie wizyty lekarza na ten konkretny dzień. Ostatnim krokiem jest iteracja przez wszystkie wizyty i weryfikacja czy proponowany termin nowej wizyty nie koliduje z istniejącą wizytą. Jeśli którakolwiek wizyta pokrywa się czasem z terminem nowej wizyty, wówczas funkcja ta zwraca wartość `false`. Jeśli zaś żadna z nich nie koliduje z nową wizytą, wówczas zwracana wartość jest `true` sygnalizując, że wizyta może zostać zapisana.

```

2 usages ▲ Kamil Cegielski
public boolean isDateTimeAvailable(DoctorEntity doctorEntity, int timeInMinutes, LocalDateTime newAppointmentDateTimeFrom) {
    LocalDateTime newAppointmentDateTimeTill = newAppointmentDateTimeFrom.plusMinutes(timeInMinutes);
    AvailabilityEntity availabilityEntity = availabilityRepository.findByDoctorUuidAndDateTime(
        doctorEntity.getUuid(), newAppointmentDateTimeFrom, newAppointmentDateTimeTill) List<AvailabilityEntity>
            .stream() Stream<AvailabilityEntity>
            .findFirst() Optional<AvailabilityEntity>
            .orElse( other: null );
    if (availabilityEntity == null) {
        return false;
    }

    LocalDateTime availabilityDateTimeFrom = availabilityEntity.getDateFrom();
    LocalDateTime availabilityDateTimeTill = availabilityEntity.getDateTill();

    List<AppointmentEntity> existingAppointments = appointmentRepository.findByAvailabilityTime(
        availabilityDateTimeFrom, availabilityDateTimeTill);
    for (AppointmentEntity existingAppointment : existingAppointments) {
        LocalDateTime existingAppointmentDateTimeFrom = existingAppointment.getDateFrom();
        Integer existingAppointmentTimeInMinutes = existingAppointment.getService().getTimeInMinutes();
        LocalDateTime existingAppointmentDateTimeTill = existingAppointmentDateTimeFrom.plusMinutes(existingAppointmentTimeInMinutes);
        if (isInCollision(newAppointmentDateTimeFrom, newAppointmentDateTimeTill,
            existingAppointmentDateTimeFrom, existingAppointmentDateTimeTill)) {
            return false;
        }
    }
    return true;
}

10 usages ▲ Kamil Cegielski
static boolean isInCollision(LocalDateTime newAppointmentDateTimeFrom, LocalDateTime newAppointmentDateTimeTill,
    LocalDateTime existingAppointmentDateTimeFrom, LocalDateTime existingAppointmentDateTimeTill) {
    return isDatePointInDateRange(newAppointmentDateTimeFrom, existingAppointmentDateTimeFrom, existingAppointmentDateTimeTill)
        || (!newAppointmentDateTimeTill.isEqual(existingAppointmentDateTimeFrom) && isDatePointInDateRange(
            newAppointmentDateTimeTill, existingAppointmentDateTimeFrom, existingAppointmentDateTimeTill))
        || isDatePointInDateRange(existingAppointmentDateTimeFrom, newAppointmentDateTimeFrom, newAppointmentDateTimeTill)
        || (!existingAppointmentDateTimeTill.isEqual(newAppointmentDateTimeFrom) && isDatePointInDateRange(
            existingAppointmentDateTimeTill, newAppointmentDateTimeFrom, newAppointmentDateTimeTill));
}

4 usages ▲ Kamil Cegielski
private static boolean isDatePointInDateRange(LocalDateTime point, LocalDateTime x1Inclusive, LocalDateTime x2Exclusive) {
    return !x1Inclusive.isAfter(point) && point.isBefore(x2Exclusive);
}

```

Rys. 3.73 Implementacja metody `isDateTimeAvailable` w klasie `PatientService` (opracowanie własne)

Wracając do metody `getAvailableAppointmentTimes` będącą implementacją endpointa `GET /api/patients/{uuid}/available-appointment-times`,prefiltrowana lista z dostępnymi terminami wizyty zostaje przekształcona w listę modeli zwrotnych i zostaje zwrócona jako wynik wykonania endpointa.

3.2.6. Omówienie REST API - /api/services

Klasa `ServiceController`, której część kodu została przedstawiona na rysunku nr 3.74, definiuje następujące endpointy z przedrostkiem `/api/services`:

- POST /api/services
- GET /api/services/{uuid}
- DELETE /api/services/{uuid}
- GET /api/services

```

@RequestMapping("api/services")
public class ServiceController {

    4 usages
    private final ServiceService serviceService;

    no usages ▲ Kamil Ceglinski
    @IsAdmin
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ServiceResponseDTO createService(@RequestBody ServiceRequestDTO requestDTO) {
        return serviceService.createService(requestDTO);
    }

    no usages ▲ Kamil Ceglinski
    @GetMapping("/{uuid}")
    @ResponseStatus(HttpStatus.OK)
    public ServiceResponseDTO getServiceById(@PathVariable String uuid) {
        return serviceService.getServiceByUuid(uuid);
    }

    no usages ▲ Kamil Ceglinski
    @IsAdmin
    @DeleteMapping("/{uuid}")
    @ResponseStatus(HttpStatus.OK)
    public void deleteService(@PathVariable String uuid) { serviceService.deleteService(uuid); }

    no usages ▲ Kamil Ceglinski
    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<ServiceResponseDTO> getAllServices() { return serviceService.getAllServices(); }
}

```

Rys. 3.74 Część kodu źródłowego klasy *ServiceController* (opracowanie własne)

Endpointy modyfikujące dane zostały opatrzone adnotacją *IsAdmin* tak, aby tylko recepcjonista mógł zarządzać usługami. Natomiast pozostały endpointy mogą być wywoływane przez każdego zalogowanego użytkownika. Rysunek nr 3.75 przedstawia implementację logiki biznesowej dla endpointów dotyczących usług medycznych. Można zauważyć, że metoda tworząca nową usługę weryfikuje również istnienie specjalizacji wymaganej do wykonania tejże usługi.

```

@Service
public class ServiceService {

    4 usages
    private final ServiceMapper serviceMapper;
    4 usages
    private final ServiceRepository serviceRepository;
    1 usage
    private final SpecialtyRepository specialtyRepository;

    1 usage ▲ Kamil Ceglinski
    public ServiceResponseDTO createService(ServiceRequestDTO requestDTO) {
        SpecialtyEntity specialtyEntity = specialtyRepository.findById(requestDTO.getSpecialtyUuid()).orElseThrow();
        ServiceEntity serviceEntity = serviceMapper.toServiceEntity(requestDTO, specialtyEntity);
        ServiceEntity savedServiceEntity = serviceRepository.save(serviceEntity);
        return serviceMapper.toServiceResponseDTO(savedServiceEntity);
    }

    1 usage ▲ Kamil Ceglinski
    public ServiceResponseDTO getServiceByUuid(String uuid) {
        return serviceRepository.findById(uuid) .optional().map(serviceMapper::toServiceResponseDTO).Optional

```

Rys. 3.75 Część kodu źródłowego klasy *ServiceService* (opracowanie własne)

3.2.7. Omówienie REST API - /api/specialties

Klasa *SpecialtyController*, której część kodu została przedstawiona na rysunku nr 3.76, definiuje następujące endpointy z przedrostkiem */api/specialties*:

- POST */api/specialties*
- GET */api/specialties/{uuid}*
- DELETE */api/specialties/{uuid}*
- GET */api/specialties*

```
@RequestMapping("api/specialties")
public class SpecialtyController {

    4 usages
    private final SpecialtyService specialtyService;

    no usages ▲ Kamil Cegiński
    @IsAdmin
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public SpecialtyResponseDTO createSpecialty(@RequestBody SpecialtyRequestDTO requestDTO) {
        return specialtyService.createSpecialty(requestDTO);
    }

    no usages ▲ Kamil Cegiński
    @GetMapping("/{uuid}")
    @ResponseStatus(HttpStatus.OK)
    public SpecialtyResponseDTO getSpecialtyById(@PathVariable String uuid) {
        return specialtyService.getSpecialtyById(uuid);
    }

    no usages ▲ Kamil Cegiński
    @IsAdmin
    @DeleteMapping("/{uuid}")
    @ResponseStatus(HttpStatus.OK)
    public void deleteSpecialty(@PathVariable String uuid) { specialtyService.deleteSpecialty(uuid); }

    no usages ▲ Kamil Cegiński
    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public List<SpecialtyResponseDTO> getAllSpecialties() { return specialtyService.getAllSpecialties(); }
}
```

Rys. 3.76 Część kodu źródłowego klasy *SpecialtyController* (opracowanie własne)

Endpointy, które odczytują dane, nie zostały oznaczone żadnymi adnotacjami wymagającymi konkretnej roli użytkownika. Jednakże endpointy usuwający oraz dodający nową specjalizację zostały oznaczone adnotacją, która powoduje, że są one dostępne tylko i wyłącznie dla recepcjonisty. Na rysunku nr 3.77 przedstawiona została implementacja klasy *SpecialtyService*, która jest serwisem definiującym logikę biznesową dla endpointów związanych ze specjalizacjami lekarzy. Natomiast serwis ten wykorzystuje repozytorium dedykowane operacjom bazodanowym związanym z tabelą przechowującą dane ze specjalizacjami.

```

@Service
public class SpecialtyService {

    4 usages
    private final SpecialtyMapper specialtyMapper;
    4 usages
    private final SpecialtyRepository specialtyRepository;

    1 usage  ▲ Kamil Cegiński
    public SpecialtyResponseDTO createSpecialty(SpecialtyRequestDTO requestDTO) {
        SpecialtyEntity specialtyEntity = specialtyMapper.toSpecialtyEntity(requestDTO);
        SpecialtyEntity savedSpecialtyEntity = specialtyRepository.save(specialtyEntity);
        return specialtyMapper.toSpecialtyResponseDTO(savedSpecialtyEntity);
    }

    1 usage  ▲ Kamil Cegiński
    public SpecialtyResponseDTO getSpecialtyByUuid(String uuid) {
        return specialtyRepository.findById(uuid).Optional<SpecialtyEntity>()
            .map(specialtyMapper::toSpecialtyResponseDTO).Optional<SpecialtyResponseDTO>
            .orElseThrow();
    }

    1 usage  ▲ Kamil Cegiński
    public void deleteSpecialty(String uuid) { specialtyRepository.deleteById(uuid); }

    1 usage  ▲ Kamil Cegiński
    public List<SpecialtyResponseDTO> getAllSpecialties() {
        return specialtyRepository.findAll().stream()
            .map(specialtyMapper::toSpecialtyResponseDTO).Stream<SpecialtyResponseDTO>
            .collect(Collectors.toList());
    }
}

```

Rys. 3.77 Część kodu źródłowego klasy *SpecialtyService* (opracowanie własne)

3.2.8. Omówienie REST API - /api/users

Klasa *UserController* przedstawiona na rysunku nr 3.78 definiuje endpoint *GET /api/users/current/uuid* zwracający identyfikator bazodanowy pacjenta lub lekarza w zależności od tego jaki użytkownik jest obecnie zalogowany.

```

@RequestMapping("api/users")
public class UserController {

    1 usage
    private final UserService userService;

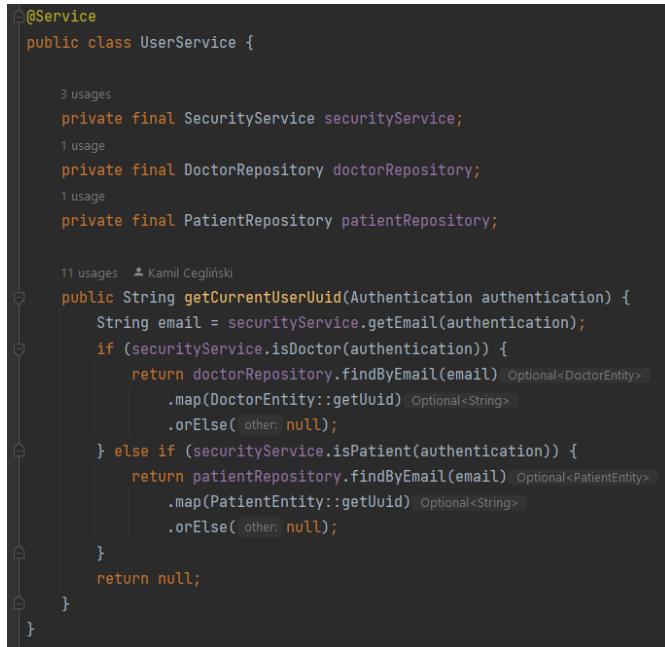
    no usages  ▲ Kamil Cegiński
    @IsDoctorOrPatient
    @GetMapping("current/uuid")
    @ResponseStatus(HttpStatus.OK)
    public CurrentUserUuidResponseDTO getCurrentUserUuid(Authentication authentication) {
        CurrentUserUuidResponseDTO currentUserUuidResponseDTO = new CurrentUserUuidResponseDTO();
        currentUserUuidResponseDTO.setUuid(userService.getCurrentUserUuid(authentication));
        return currentUserUuidResponseDTO;
    }
}

```

Rys. 3.78 Część kodu źródłowego klasy *UserController* (opracowanie własne)

Warto zwrócić uwagę na adnotację *IsDoctorOrPatient*, która powoduje, że endpoint ten może być wywołany jednie przez doktora lub pacjenta. Implementacja logiki biznesowej znajduje się w klasie *UserService*, która została przedstawiona na rysunku nr 3.79. Identyfikacja użytkownika zostaje dokonana na podstawie roli oraz adresu email. Następnie użyte zostaje

repozytorium dedykowane do operacji bazodanowych na tabeli *patient* lub *doctor* w celu zwrotu encji, która posiada identyfikator użytkownika.



```
❶ @Service
❷ public class UserService {
❸
❹     3 usages
❺     private final SecurityService securityService;
❻     1 usage
❼     private final DoctorRepository doctorRepository;
⽿     1 usage
⽾     private final PatientRepository patientRepository;
⽿
⽿     11 usages  ↳ Kamil Cegliński
⽾     public String getCurrentUserUuid(Authentication authentication) {
⽾         String email = securityService.getEmail(authentication);
⽾         if (securityService.isDoctor(authentication)) {
⽾             return doctorRepository.findByEmail(email).Optional<DoctorEntity>()
⽾                 .map(DoctorEntity::getUuid).Optional<String>
⽾                 .orElse( other: null);
⽾         } else if (securityService.isPatient(authentication)) {
⽾             return patientRepository.findByEmail(email).Optional<PatientEntity>()
⽾                 .map(PatientEntity::getUuid).Optional<String>
⽾                 .orElse( other: null);
⽾         }
⽾         return null;
⽾     }
⽾ }
```

Rys. 3.79 Część kodu źródłowego klasy *UserService* (opracowanie własne)

4. WDROŻENIE NA CHMURE OBLICZENIOWĄ AZURE

4.1. Microsoft Entra ID

Microsoft Entra ID jest usługą na platformie Azure, która zapewnia warstwę uwierzytelnienia i bezpieczeństwa. W celach testowych dodanych zostało kilku użytkowników do Microsoft Entra ID, co zostało pokazane na rysunku nr 4.1.

Display name	User principal name	User type	On-premises sync	Identities	Company name	Creation type
Adam Kowalski	adam.kowalski@k...@outlook.com	Member	No	kamilp06@gmail.onmicrosoft.com		
Aneta Kowalewska	aneta.kowalewska@k...@outlook.com	Member	No	kamilp06@gmail.onmicrosoft.com		
Kamil Ceglinski	kamilp06@gmail.onmicrosoft.com	Member	No		MicrosoftAccount	
Marcin Kolarczyk	marcin.kolarczyk@k...@outlook.com	Member	No	kamilp06@gmail.onmicrosoft.com		
Maria Strzecha	maria.strzecha@k...@outlook.com	Member	No	kamilp06@gmail.onmicrosoft.com		
Zuzanna Skworońska	zuzanna.skwronka@k...@outlook.com	Member	No	kamilp06@gmail.onmicrosoft.com		

Rys. 4.1 Dodani użytkownicy do usługi Microsoft Entra ID (opracowanie własne)

Ponadto, zdefiniowane zostały również grupy w Microsoft Entra ID dla 3 рол dostępných w aplikacji tj. recepcjonista, lekarz oraz pacjent, co zostało zaprezentowane na rysunku nr 4.2.

Name	Object Id	Group type	Membership type	Email	Source
admin	57057a39-a1bf-4aed-bea8-c0f661f120	Security	Assigned		Cloud
doctors	2330f09d-427c-4bd0-b554-e75782ba21c	Security	Assigned		Cloud
patients	973529ee-6146-4109-b7ef-21ea5f3febe2	Security	Assigned		Cloud

Rys. 4.2 Dodane grupy do usługi Microsoft Entra ID (opracowanie własne)

Do grup zostali przypisani wybrani użytkowcy. Lista lekarzy została pokazana na rysunku nr 4.3.

Name	Type	Email	User type	Object Id	Device Id
Adam Kowalski	User		Member	81167e97-1c6e-44b2-8da7-56a798e26e17	
Zuzanna Skworońska	User		Member	9960x0bc-b738-424a-9e52-bb4a5df18bed	

Rys. 4.3 Lista lekarzy przypisana do grupy doctors w Microsoft Entra ID (opracowanie własne)

Ostatnim komponentem konfiguracji w Microsoft Entra ID jest dodanie aplikacji *GreatHealth* w zakładce *App Registrations*, co zostało pokazane na rysunku nr 4.4.

The screenshot shows the Microsoft Azure portal's 'App registrations' section. At the top, there are navigation links for 'Home', 'Default Directory | Overview', 'Copilot', and a user profile. Below the header, there are buttons for 'New registration', 'Endpoints', 'Troubleshoot', 'Refresh', 'Download', 'Preview features', and 'Got feedback?'. A message bar at the top states: 'Starting June 30th, 2020 we will no longer add any new features to Azure Active Directory Authentication Library (ADAL) and Azure Active Directory Graph. We will continue to provide technical support and security updates but we will no longer provide feature updates. Applications will need to be upgraded to Microsoft Authentication Library (MSAL) and Microsoft Graph. Learn more...'. The main area displays a table with one application entry:

Display name	Application (client) ID	Created on	Certificates & secrets
greathealth	506294b9-701c-4b3a-935a-0694e538a3bf	3/15/2025	-

Rys. 4.4 Aplikacja GreatHealth dodana do usługi Microsoft Entra ID (opracowanie własne)

4.2. Docker

Do realizacji wdrożenia został użyty *Docker*⁵³, który ułatwia instalację oprogramowania. W celu utworzenia paczki instalacyjnej, definiuje się obraz Dockerowy, który następnie służy do utworzenia instancji tego obrazu. Instancja ta jest nazywana kontenerem Dockerowym i jest swego rodzaju maszyną wirtualną, ale znacznie lżejszą i wydajniejszą⁵⁴.

Rysunek nr 4.5 przedstawia plik *Dockerfile*, który definiuje obraz Dockerowy. Strukturę tego kodu źródłowego można podzielić na dwie części. Pierwsza sekcja jest odpowiedzialna za budowę projektu frontendowego, zaś druga backendowego, który scalą oba projekty w jeden. Scalenie obu projektów w jeden polega na wygenerowaniu plików HTML, CSS oraz JS, a następnie umieszczenie ich po stronie backendu w odpowiednim folderze, aby następnie serwer aplikacyjny zwracał te pliki do przeglądarki, gdy ta wysyła żądania niezaczynające się przedrostkiem */api*. Natomiast wszystkie żądania zaczynające się na ten przedrostek trafiają do obsługi kontrolerów po stronie backendu.

⁵³ <https://www.docker.com/>

⁵⁴ <https://www.freecodecamp.org/news/docker-vs-vm-key-differences-you-should-know/>

```

FROM node:alpine as frontendbuild
WORKDIR /usr/src/app
COPY ./frontend /usr/src/app
RUN npm install -g @angular/cli
RUN npm install
RUN npm run build

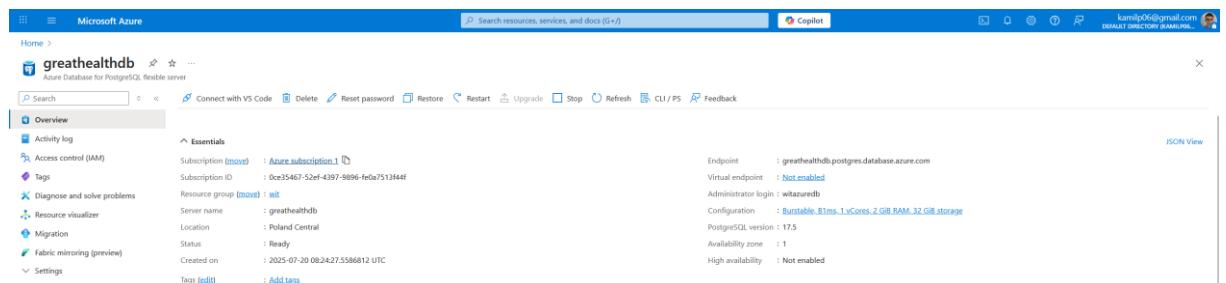
FROM eclipse-temurin:17-jdk-alpine
WORKDIR /opt/app
COPY ./backend/ ./
COPY --from=frontendbuild /usr/src/app/dist/greathealth/browser/ /opt/app/src/main/resources/static/
ENV WEB_PORT 80
RUN chmod +x gradlew
RUN ./gradlew bootJar
ENTRYPOINT ["java", "-jar", "./build/libs/greathealth-0.0.1-SNAPSHOT.jar"]

```

Rys. 4.5 Definicja obrazu Dockerowego dla aplikacji GreatHealth (opracowanie własne)

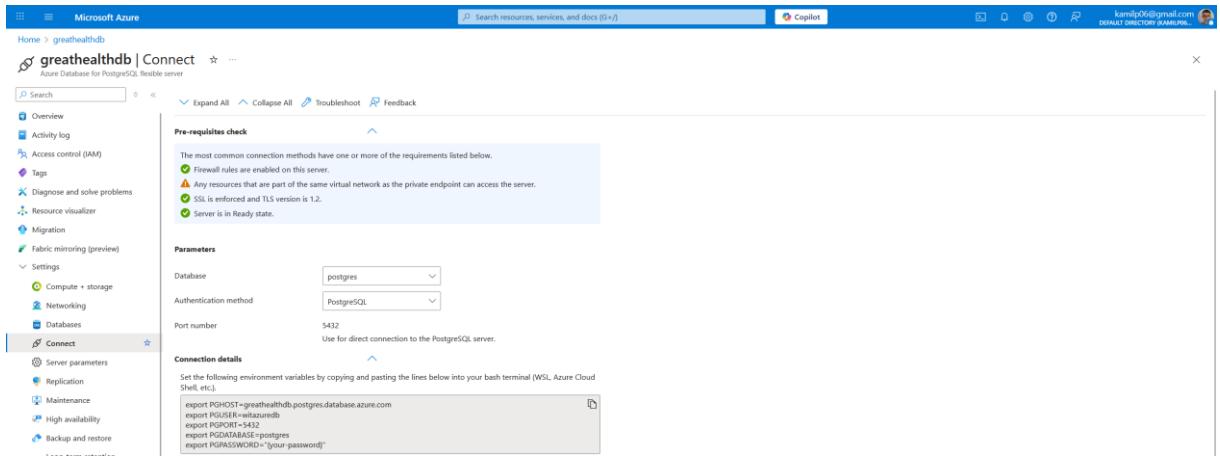
4.3. Azure Database for PostgreSQL

Na platformie Azure utworzona została baza danych za pomocą usługi Azure Database for PostgreSQL⁵⁵, co zostało zaprezentowane na rysunku nr 4.6. Nazwa bazy danych to *greathealthdb*, zaś URL, za pomocą którego można połączyć się do bazy to *greathealthdb.postgres.database.azure.com*. Podczas tworzenia bazy danych, utworzone zostało hasło oraz nazwa użytkownika, za pomocą których można się połączyć do bazy. Domyślnym portem dla bazy PostgreSQL jest 5432, co zostało zaprezentowane na rysunku nr 4.7.



Rys. 4.6 Utworzona baza danych na platformie Azure (opracowanie własne)

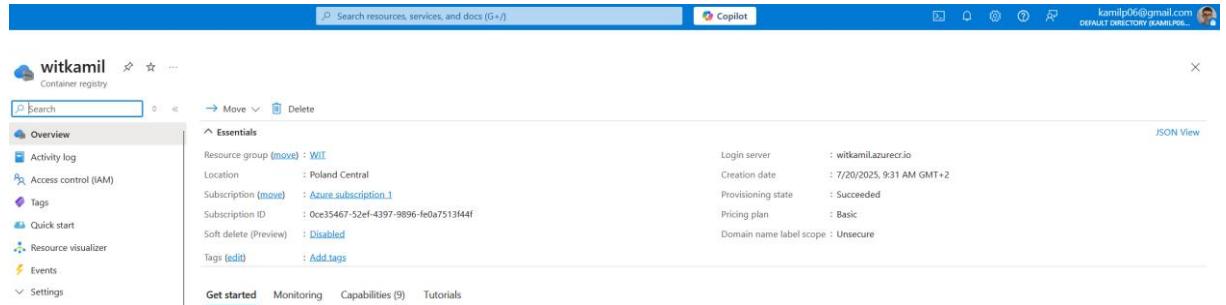
⁵⁵ <https://azure.microsoft.com/en-us/products/postgresql/>



Rys. 4.7 Konfiguracja bazy danych na platformie Azure (opracowanie własne)

4.4. App Service

Następnym krokiem do wdrożenia aplikacji jest konfiguracja usługi Azure App Service⁵⁶, która będzie oparta o obraz Dockerowy. W tym celu, należy najpierw utworzyć rejestr obrazów Dockerowych w Azure korzystając z usługi Azure Container Registry⁵⁷, co zostało zaprezentowane na rysunku nr 4.8.



Rys. 4.8 Utworzony rejestr obrazów Dockerowych na platformie Azure (opracowanie własne)

Kiedy rejestr już istnieje, można utworzyć konfigurację w usłudze Azure App Service, co zostało pokazane na rysunku nr 4.9.

⁵⁶ <https://azure.microsoft.com/pl-pl/products/app-service/>

⁵⁷ <https://learn.microsoft.com/en-us/azure/container-registry/>

Rys. 4.9 Konfiguracja Azure App Service (opracowanie własne)

Warto również wspomnieć o trzech zmiennych środowiskowych zawierających dane potrzebne aplikacji do połączenia z bazą danych. Rysunek nr 4.10 przedstawia tę konfigurację w Azure App Service.

Name	Value	Deployment slot setting	Source
DB_PASSWORD	(Show value)		App Service
DB_URL	(Show value)		App Service
DB_USERNAME	(Show value)		App Service

Rys. 4.10 Zdefiniowane zmienne środowiskowe dotyczące połączenia z bazą danych w Azure App Service (opracowanie własne)

4.5. GitHub Actions

Ostatnim krokiem jest automatyzacja wdrożenia, czyli tzw. CI/CD (continuous integration/continuous deployment). Ten komponent został zrealizowany za pomocą usługi GitHub Actions⁵⁸. W związku z tym, że cały kod źródłowy aplikacji jest przechowywany w repozytorium Gita⁵⁹ na platformie GitHub⁶⁰, integracja i wykorzystanie usługi GitHub Actions jest bardzo prosta i polega na zdefiniowaniu konfiguracji w pliku *deployment.yml* w folderze *.github\workflows*, co zostało zaprezentowane na rysunku nr 4.11.

⁵⁸ <https://docs.github.com/en/actions>

⁵⁹ <https://git-scm.com/>

⁶⁰ <https://github.com/>

```

name: GreatHealth CI/CD

on:
  push:
    branches:
      - deployment

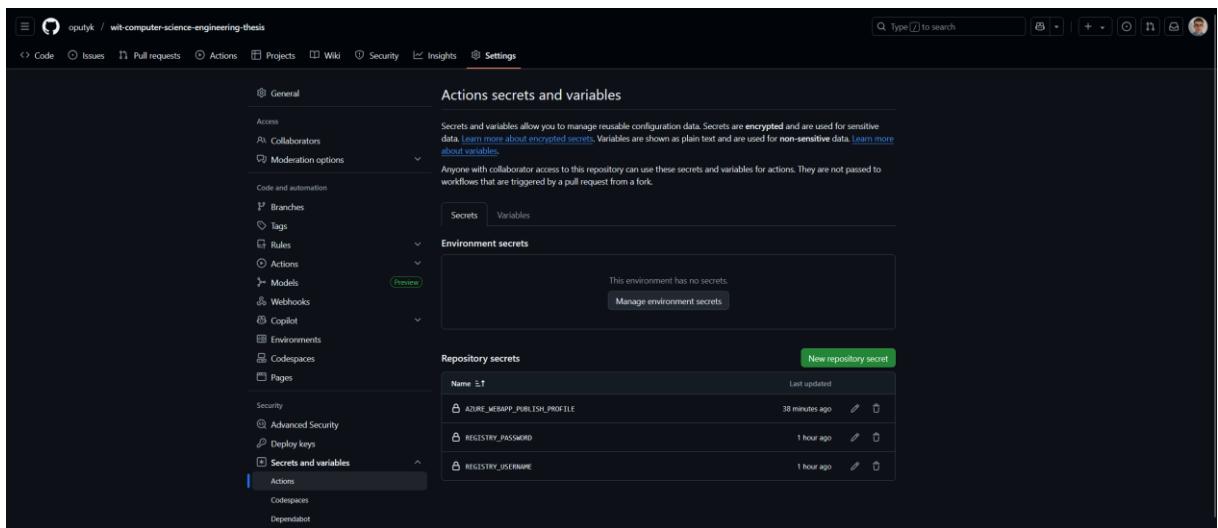
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - uses: azure/docker-login@v1
        with:
          login-server: witkamil.azurecr.io
          username: ${{ secrets.REGISTRY_USERNAME }}
          password: ${{ secrets.REGISTRY_PASSWORD }}
      - run:
          docker build . -t witkamil.azurecr.io/greathealth:${{ github.run_number }}
          docker push witkamil.azurecr.io/greathealth:${{ github.run_number }}
      - uses: azure/webapps-deploy@v2
        with:
          app-name: 'greathealth'
          publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
          images: 'witkamil.azurecr.io/greathealth:${{ github.run_number }}'

```

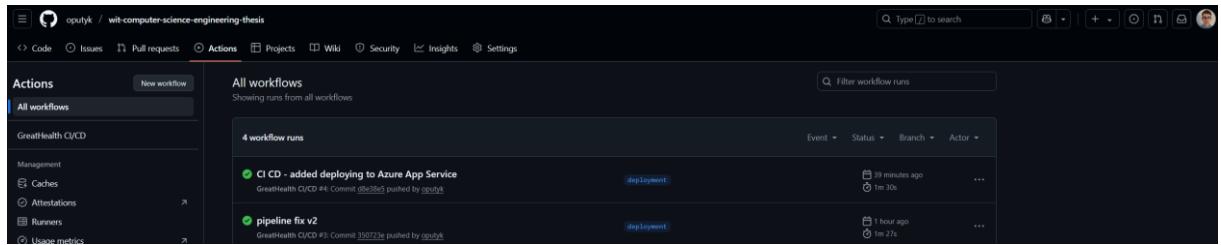
Rys. 4.11 Definicja skryptu GitHub Actions (opracowanie własne)

Skrypt wdrożeniowy wykonywany jest gdy nowe zmiany pojawią się na branchu o nazwie *deployment* na platformie GitHub. Skrypt ten można podzielić na dwa etapy. W pierwszym tworzony jest obraz Dockerowy, zaś w drugim inicjowany jest proces wdrożenia nowego obrazu w usługi Azure App Service. W celu realizacji obu zadań wymagane są dane uwierzytelniające do usług Azure Container Registry oraz Azure App Service, które zostały przekazane za pomocą *Secrets* zdefiniowanych na platformie GitHub, co zostało pokazane na rysunku nr 4.12.



Rys. 4.12 Secrets zdefiniowane na platformie GitHub (opracowanie własne)

Rysunek nr 4.13 pokazuje listę wykonanych wdrożeń, zaś rysunek nr 4.14 wynik wdrożenia zaprezentowany po przejściu na stronę aplikacji GreatHealth.



The screenshot shows the GitHub Actions interface for the repository 'wilt-computer-science-engineering-thesis'. The 'Actions' tab is selected. On the left, there's a sidebar with 'Actions', 'GreatHealth CI/CD', 'Management' (with 'Caches', 'Attalations', 'Runners', and 'Usage metrics' sections), and a 'New workflow' button. The main area displays 'All workflows' and 'All workflow runs'. There are four workflow runs listed:

Workflow	Description	Type	Time Ago	More Options
CI CD - added deploying to Azure App Service	GreatHealth CI/CD #4: Commit dbe38d5 pushed by oputyk	deployment	39 minutes ago	...
pipeline fix v2	GreatHealth CI/CD #3: Commit 350723e pushed by oputyk	deployment	1 hour ago	...
CI CD - added deploying to Azure App Service	GreatHealth CI/CD #3: Commit dbe38d5 pushed by oputyk	deployment	1m 30s	...

Rys. 4.13 Lista wykonanych wdrożeń (opracowanie własne)



The screenshot shows the GreatHealth application homepage. At the top, there's a navigation bar with links for 'Great Health', 'Lekarze', 'Pacjenci', 'Specjalizacje', 'Usługi', and a user profile for 'Maria Strzecha'. Below the navigation, a banner says 'Witaj Adminie!'. The main content area is currently empty, showing a light gray background.

Rys. 4.14 Wynik wdrożenia na platformie Azure (opracowanie własne)

ZAKOŃCZENIE

System do rezerwacji wizyt w fikcyjnej przychodni został zrealizowany oraz wdrożony przy użyciu usług chmurowych Azure. Podzielony został on na dwie osobne aplikacje, gdzie pierwsza z nich jest warstwą prezentacji wyświetlaną w przeglądarce, zaś druga realizuje logikę biznesową, jest wykonywana na serwerze i łączy się z relacyjną bazą danych. Zastosowane wzorce projektowe powodują, że dodawanie nowych funkcjonalności oraz edycja istniejących nie wymaga dużego nakładu pracy. Ponadto, system jest elastyczny i umożliwia wdrożenie nie tylko na chmurę obliczeniową Azure. Przy lekkiej modyfikacji sposobu uwierzytelniania, możliwa jest migracja systemu do innych usługodawców rozwiązań chmurowych, co jest dużą zaletą na dynamicznym rynku IT.

System jest gotowy do produkcyjnego zastosowania w wybranej przychodni, umożliwiając recepcjonistce dodanie do systemu lekarzy, pacjentów, specjalizacji oraz usług medycznych tak, aby następnie lekarze mogli wprowadzić swój grafik do systemu. Wówczas pacjenci są w stanie rezerwować wizyty w przychodni. Ponadto, logika rezerwacji wizyt jest zaimplementowana w taki sposób, aby żadna z wizyt nie nachodziła na siebie. Sam proces rezerwacji wizyty jest bardzo płynny i prosty z perspektywy pacjenta, umożliwiając mu wybór lekarza, usługi oraz terminu spośród dostępnych opcji. Z perspektywy lekarza, system również wprowadza komfort pracy, zapewniając przejrzystą listę wizyt na dany dzień, które uwzględniają wcześniej wprowadzony przez niego grafik godzin pracy w przychodni.

Z pewnością system posiada pewne ograniczenia, które można rozwiązać poprzez dodatkowy nakład pracy. Jednym z takich ograniczeń jest automatyzacja skalowalności rozwiązania. Obecna konfiguracja wdrożenia nie pozwala na automatyczną skalowalność aplikacji przy wzmożonym ruchu w systemie. Jednakże, rozwiązanie tego problemu wymaga jedynie zmiany konfiguracji w wybranych usługach sieciowych tak, aby taką funkcjonalność wprowadzić. Sama implementacja systemu pozwala na skalowanie poprzez manipulację liczbą instancji aplikacji. Kolejnym możliwym usprawnieniem działania systemu mogłoby być wprowadzenia funkcjonalności wskazującej, gdzie znajduje się dany lekarz. Taka funkcja z pewnością ułatwiłaby życie pacjentom, którzy przybywają do przychodni na wizytę, ale również i samym recepcjonistom, którzy nie musieliby ciągle podawać tej informacji pacjentom.

BIBLIOGRAFIA

Książki

1. Karl E Wieggers, Joy Beatty, Specyfikacja oprogramowania. Inżynieria wymagań. Wydanie III, Helion, 2014
2. Doug Rosenberg, Matt Stephens, Use Case Driven Object Modeling with UML. Theory and Practice. Appress, 2007
3. Jarosław Żeliński, Analiza Biznesowa. Praktyczne modelowanie organizacji, Helion, 2016
4. Pethuru Raj, Anupama Raman, Harihara Subramanian, Architectural Patterns, Packt Publishing, 2017
5. Ivan Ristić, Bulletproof SSL and TLS, Feisty Duck Limited, 2022
6. Cay S. Horstmann, Java Podstawy. Wydanie XI, Helion, 2019

Strony internetowe

1. Kamil Cegliński (27.07.2025), Kod źródłowy projektu do pracy dyplomowej. Pobrano 27.07.2025: <https://github.com/oputyk/wit-computer-science-engineering-thesis>
2. Sanchhaya Education Private Limited (11.07.2025), Frontend vs Backend Development. Pobrano 15.07.2025: <https://www.geeksforgeeks.org/frontend-vs-backend/>
3. MunnaPraWiN (13.05.2024), Effective Collaboration Between Frontend and Backend Development Teams. Pobrano 10.07.2025: <https://medium.com/buildingminds-technologies/importance-of-collaboration-between-frontend-and-backend-teams-4b05e8fd29f9>
4. Okta (2025), Authorization Code Flow with Proof Key for Code Exchange (PKCE). Pobrano 27.05.2025: <https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce>
5. Microsoft (23.10.2023), Access token claims reference. Pobrano 27.05.2025: <https://learn.microsoft.com/en-us/entra/identity-platform/access-token-claims-reference>
6. Microsoft (21.03.2025), Tokens and claims overview. Pobrano 27.05.2025: <https://learn.microsoft.com/en-us/entra/identity-platform/security-tokens#token-endpoints-and-issuers>.

7. Okta (2025), Introduction to JSON Web Tokens. Pobrano 27.05.2025:
<https://jwt.io/introduction>
8. E. Rescorla (05.2000), HTTP Over TLS. Pobrano 27.05.2025: <https://www.rfc-editor.org/rfc/rfc2818>.
9. The PostgreSQL Global Development Group (2025), Secure TCP/IP Connections with SSL. Pobrano 29.05.2025: <https://www.postgresql.org/docs/current/ssl-tcp.html>.
10. Arthur Bellore (07.03.2023), The TLS Handshake Explained. Pobrano 30.05.2025:
<https://auth0.com/blog/the-tls-handshake-explained/>.
11. Google (2025), Angular. Pobrano 01.06.2025: <https://angular.dev/>.
12. Microsoft (2025), TypeScript. Pobrano 01.06.2025: <https://www.typescriptlang.org/>.
13. Mozilla Contributors (14.07.2025), CSS: Cascading Style Sheets. Pobrano 20.07.2025: <https://developer.mozilla.org/en-US/docs/Web/CSS>
14. Mozilla Contributors (09.07.2025), HTML: HyperText Markup Language. Pobrano 20.07.2025 <https://developer.mozilla.org/en-US/docs/Web/HTML>
15. UDI Group (28.02.2022), Język HTML – co to jest, do czego służy i jak wygląda?. Pobrano 01.06.2025: <https://udigroup.pl/blog/jezyk-html-co-to-jest-do-czego-sluzy-jak-wyglada/>
16. Piotr Broniewski (2025), CSS. Co to jest i jak działa CSS. Podstawowe i najważniejsze informacje o CSS. Pobrano 01.06.2025: <https://webporadnik.pl/css-co-to-jest-i-jak-dziala-css-podstawowe-i-najwazniejsze-informacje-o-css>
17. Refsnes Data (2025), JavaScript Tutorial. Pobrano 01.06.2025:
<https://www.w3schools.com/Js>
18. Google (2025), What is Angular? Pobrano 01.06.2025: <https://angular.dev/overview>
19. Google (2025), Angular Material, Material Design components for Angular. Pobrano 01.06.2025: <https://material.angular.dev/>
20. Michał Wrochna (07.11.2024), Rest API – Co to jest, jak działa i jak z niego korzystać? Pobrano 05.06.2025: <https://poradnikinżyniera.pl/rest-api-co-to-jest-jak-dziala-i-jak-z-niego-korzystac/>
21. Lokesh Gupta (05.11.2023), Idempotent REST API. Pobrano 05.06.2025:
<https://restfulapi.net/idempotent-rest-apis/>
22. Oracle (2025), Java. Pobrano 05.06.2025: <https://www.oracle.com/pl/java/>
23. Oracle (2025) Java Garbage Collection Basics. Pobrano 05.06.2025:
<https://www.oracle.com/webfolder/technetwork/Tutorials/obe/java/gc01/index.html>

24. Broadcom (2025). Spring Framework 6.2.9. Pobrano 05.06.2025:
<https://spring.io/projects/spring-framework>
25. Broadcom (2025). Spring Web MVC. Pobrano 05.06.2025:
<https://docs.spring.io/spring-framework/reference/web/webmvc.html>
26. Broadcom (2025), Spring Data 2025.0.2. Pobrano 05.06.2025:
<https://spring.io/projects/spring-data>
27. Broadcom (2025), Spring Boot 3.5.4. Pobrano 05.06.2025:
<https://spring.io/projects/spring-boot>
28. Broadcom (2025), The IoC Container. Pobrano 05.06.2025:
<https://docs.spring.io/spring-framework/reference/core/beans.html>
29. Broadcom (2025), Bean Overview. Pobrano 05.06.2025: <https://docs.spring.io/spring-framework/reference/core/beans/definition.html>
30. Gradle (2025), Gradle. Pobrano 05.06.2025: <https://gradle.org/>
31. Liquibase (2025), Liquibase. Pobrano 05.06.2025: <https://www.liquibase.com/>
32. The Project Lombok Authors (2025), Project Lombok. Pobrano 05.06.2025:
<https://projectlombok.org/>
33. The Project Lombok Authors (2025), @Getter and @Setter. Pobrano 05.06.2025:
<https://projectlombok.org/features/GetterSetter>
34. JUnit (2025), JUnit 4, Pobrano 05.06.2025: <https://junit.org/junit4/>
35. The PostgreSQL Global Development Group (2025), PostgreSQL: The World's Most Advanced Open Source Relational Database. Pobrano 05.06.2025
<https://www.postgresql.org/>
36. Sanchhaya Education Private Limited (26.02.2025), What is SQL? Pobrano 05.06.2025: <https://www.geeksforgeeks.org/what-is-sql/>
37. Docker (2025), Docker. Pobrano 05.06.2025: <https://www.docker.com/>
38. Bala Priya C (04.10.2022), Docker vs Virtual Machine (VM) – Key Differences You Should Know, Pobrano 05.06.2025: <https://www.freecodecamp.org/news/docker-vs-vm-key-differences-you-should-know/>
39. Microsoft (2025), Azure Database for PostgreSQL. Pobrano 05.06.2025:
<https://azure.microsoft.com/en-us/products/postgresql/>
40. Microsoft (2025), Azure App Service. Pobrano 05.06.2025:
<https://azure.microsoft.com/pl-pl/products/app-service/>
41. Microsoft (2025), Azure Container Registry documentation. Pobrano 05.06.2025:
<https://learn.microsoft.com/en-us/azure/container-registry/>

42. GitHub (2025), GitHub Actions documentation. Pobrano 05.06.2025:
<https://docs.github.com/en/actions>
43. Brak autora i daty, Git, Pobrano 05.06.2025: <https://git-scm.com/>
44. GitHub (2025), GitHub. Pobrano 05.06.2025: <https://github.com/>

WYKAZ RYSUNKÓW

Rys. 1.1 Diagram modeli domenowych (opracowanie własne)	13
Rys. 1.2 Diagram przypadków użycia (opracowanie własne).....	15
Rys. 1.3 Diagram klas (opracowanie własne)	17
Rys. 2.1 Diagram architektury systemu (opracowanie własne)	18
Rys. 2.2 Schemat działania uwierzytelnienia Authorization Code Flow with PKCE (https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce)	21
Rys. 2.3 Przykład tokena JWT wygenerowanego przez Microsoft Entra ID oraz zdekodowanego na stronie https://jwt.io	22
Rys. 2.4 Protokół TLS Handshake (Ivan Ristić, Bulletproof SSL and TLS, Feisty Duck Limited, s. 27)	24
Rys. 2.5 Przykład kodu HTML oraz wyświetlenia go w przeglądarce (https://udigroup.pl/blog/jazyk-html-co-to-jest-do-czego-sluzy-jak-wyglada/).....	26
Rys. 2.6 Przykład użycia CSS (opracowanie własne przy użyciu strony https://www.w3schools.com).....	27
Rys. 2.7 Przykładowe pola formularza z Angular Material (https://material.angular.dev/components/input/overview)	28
Rys. 2.8 Przykładowe przyciski z Angular Material (https://material.angular.dev/components/button/overview)	28
Rys. 3.1 Plik package.json zarządzający zależnościami projektu (opracowanie własne)	33
Rys. 3.2 Plik angular.json będący globalną konfiguracją Angularową (opracowanie własne)	34
Rys. 3.3 Plik app.config.ts stanowiący konfigurację aplikacji w języku TypeScript (własne opracowanie)	35
Rys. 3.4 Klasa TokenInterceptorService (opracowanie własne)	35
Rys. 3.5 Lista podstron aplikacji zdefiniowana w pliku app.routes.ts (własne opracowanie)	36
Rys. 3.6 Warstwa prezentacji komponentu menu (własne opracowanie).....	37
Rys. 3.7 Warstwa logiki komponentu menu (własne opracowanie).....	37
Rys. 3.8 Menu wyświetlane dla użytkownika będącego lekarzem (własne opracowanie)	37
Rys. 3.9 Menu wyświetlane dla użytkownika będącego pacjentem (własne opracowanie)	38

Rys. 3.10 Menu wyświetlane dla użytkownika będącego recepcjonistą (własne opracowanie)	38
Rys. 3.11 Podstrona Wizyty z pacjentami (opracowanie własne)	38
Rys. 3.12 Warstwa prezentacji komponentu dla podstrony Wizyty z pacjentami (opracowanie własne).....	39
Rys. 3.13 Definicja warstwy logiki dla podstrony Wizyty z pacjentami (opracowanie własne)	40
Rys. 3.14 Wygląd podstrony Dostępność (opracowanie własne)	41
Rys. 3.15 Formularz dodający nowy termin dostępności lekarza (opracowanie własne)	41
Rys. 3.16 Warstwa prezentacji komponentu dla podstrony Dostępność (opracowanie własne)	42
Rys. 3.17 Definicja warstwy logiki dla podstrony Wizyty z pacjentami (opracowanie własne)	42
Rys. 3.18 Warstwa prezentacji formularza służącego do dodawania dostępności (opracowanie własne).....	43
Rys. 3.19 Definicja warstwy logiki dla formularza służącego do dodawania dostępności (opracowanie własne)	44
Rys. 3.20 Wygląd podstrony Lekarze (opracowanie własne).....	45
Rys. 3.21 Formularz dodający/edytujący dane lekarza (opracowanie własne)	45
Rys. 3.22 Podgląd wizyt wybranego lekarza na podstronie Lekarze (opracowanie własne) ...	46
Rys. 3.23 Fragment kodu z warstwy prezentacji wyświetlający tabelę z lekarzami na podstronie Lekarze (opracowanie własne)	47
Rys. 3.24 Fragment kodu z warstwy logiki obsługującej tabelę z doktorami na podstronie Lekarze (opracowanie własne)	47
Rys. 3.25 Fragment kodu z warstwy prezentacji wyświetlający tabelę z wizytami dla wybranego lekarza na podstronie Lekarze (opracowanie własne)	48
Rys. 3.26 Fragment kody z warstwy logiki obsługujący tworzenie oraz zamykanie formularza na podstronie Lekarze (opracowanie własne)	49
Rys. 3.27 Kod z warstwy prezentacji formularza z podstrony Lekarze (opracowanie własne)	50
Rys. 3.28 Kod z warstwy logiki formularza z podstrony Lekarze (opracowanie własne)	50
Rys. 3.29 Wygląd podstrony Pacjenci (opracowanie własne)	51
Rys. 3.30 Wygląd formularza na podstronie Pacjenci (opracowanie własne).....	51
Rys. 3.31 Kod warstwy prezentacji podstrony Pacjenci (opracowanie własne)	52

Rys. 3.32 Kod warstwy logiki podstrony Pacjenci (opracowanie własne).....	52
Rys. 3.33 Kod warstwy prezentacji formularza z podstrony Pacjenci (opracowanie własne). .	53
Rys. 3.34 Kod warstwy logiki formularza z podstrony Pacjenci (opracowanie własne)	53
Rys. 3.35 Wygląd podstrony Specjalizacje (opracowanie własne)	54
Rys. 3.36 Wygląd okna służącego do dodawania nowych specjalizacji (opracowanie własne)	54
.....	
Rys. 3.37 Kod źródłowy warstwy prezentacji podstrony Specjalizacje (opracowanie własne)	55
.....	
Rys. 3.38 Kod źródłowy warstwy logiki podstrony Specjalizacje (opracowanie własne).....	55
Rys. 3.39 Wygląd podstrony Usługi (opracowanie własne).....	56
Rys. 3.40 Wygląd okna umożliwiającego dodanie nowej usługi medycznej (opracowanie własne).....	56
Rys. 3.41 Kod źródłowy warstwy prezentacji podstrony Usługi (opracowanie własne)	57
Rys. 3.42 Kod źródłowy warstwy logiki podstrony Usługi (opracowanie własne)	57
Rys. 3.43 Kod źródłowy warstwy prezentacji okna do dodawania nowych usług (opracowanie własne).....	58
Rys. 3.44 Kod źródłowy warstwy logiki okna do dodawania nowych usług (opracowanie własne).....	58
Rys. 3.45 Wygląd podstrony Wizyty (opracowanie własne)	59
Rys. 3.46 Wygląd okna służącego do umawiania się na wizytę z lekarzem (opracowanie własne)	59
.....	
Rys. 3.47 Kod źródłowy warstwy prezentacji podstrony Wizyty (opracowanie własne)	60
Rys. 3.48 Kod źródłowy warstwy logiki podstrony Wizyty (opracowanie własne)	61
Rys. 3.49 Kod źródłowy warstwy prezentacji okna do dodawania nowych wizyt (opracowanie własne).....	62
Rys. 3.50 Kod źródłowy warstwy logiki okna do dodawania nowych wizyt (opracowanie własne).....	62
Rys. 3.51 Kod źródłowy serwisu Specialty ApiService (opracowanie własne).....	63
Rys. 3.52 Kod źródłowy interfejsu Specialty Request (opracowanie własne)	63
Rys. 3.53 Kod źródłowy interfejsu Specialty (opracowanie własne).....	63
Rys. 3.54 Kod źródłowy serwisu TokenInterceptorService (opracowanie własne)	64
Rys. 3.55 Zarządzanie zależnościami backendu - plik build.gradle (opracowanie własne)....	65
Rys. 3.56 Globalna konfiguracja backendu – plik application.yml (opracowanie własne)	66
Rys. 3.57 Konfiguracja SecurityConfig.java (opracowanie własne).....	66

Rys. 3.58 Serwis SecurityService.java (opracowanie własne)	67
Rys. 3.59 Adnotacja IsAdmin (opracowanie własne).....	67
Rys. 3.60 Adnotacja IsAdminOrDoctor (opracowanie własne)	67
Rys. 3.61 Adnotacja IsAdminOrPatient (opracowanie własne)	67
Rys. 3.62 Adnotacja IsDoctor (opracowanie własne).....	67
Rys. 3.63 Adnotacja IsDoctorOrPatient (opracowanie własne)	68
Rys. 3.64 Adnotacja IsPatient (opracowanie własne).....	68
Rys. 3.65 Skrypt Liquibase 001_init.xml (opracowanie własne)	68
Rys. 3.66 Schemat bazy danych (opracowanie własne)	69
Rys. 3.67 Część kodu źródłowego klasy DoctorController (opracowanie własne)	70
Rys. 3.68 Część kodu źródłowego DoctorService (opracowanie własne)	71
Rys. 3.69 Część kodu źródłowego klasy DoctorsAvailabilityController (opracowanie własne)	72
Rys. 3.70 Część kodu źródłowego klasy DoctorsAvailabilityService (opracowanie własne)..	73
Rys. 3.71 Część kodu źródłowego klasy PatientController (opracowanie własne)	74
Rys. 3.72 Część kodu źródłowego klasy PatientService (opracowanie własne)	75
Rys. 3.73 Implementacja metody isDateTimeAvailable w klasie PatientService (opracowanie własne).....	76
Rys. 3.74 Część kodu źródłowego klasy ServiceController (opracowanie własne).....	77
Rys. 3.75 Część kodu źródłowego klasy ServiceService (opracowanie własne)	77
Rys. 3.76 Część kodu źródłowego klasy SpecialtyController (opracowanie własne).....	78
Rys. 3.77 Część kodu źródłowego klasy SpecialtyService (opracowanie własne)	79
Rys. 3.78 Część kodu źródłowego klasy UserController (opracowanie własne)	79
Rys. 3.79 Część kodu źródłowego klasy UserService (opracowanie własne)	80
Rys. 4.1 Dodani użytkownicy do usługi Microsoft Entra ID (opracowanie własne)	81
Rys. 4.2 Dodane grupy do usługi Microsoft Entra ID (opracowanie własne).....	81
Rys. 4.3 Lista lekarzy przypisana do grupy doctors w Microsoft Entra ID (opracowanie własne)	81
Rys. 4.4 Aplikacja GreatHealth dodana do usługi Microsoft Entra ID (opracowanie własne)	82
Rys. 4.5 Definicja obrazu Dockerowego dla aplikacji GreatHealth (opracowanie własne)	83
Rys. 4.6 Utworzona baza danych na platformie Azure (opracowanie własne)	83
Rys. 4.7 Konfiguracja bazy danych na platformie Azure (opracowanie własne).....	84
Rys. 4.8 Utworzony rejestr obrazów Dockerowych na platformie Azure (opracowanie własne)	84

Rys. 4.9 Konfiguracja Azure App Service (opracowanie własne).....	85
Rys. 4.10 Zdefiniowane zmienne środowiskowe dotyczące połączenia z bazą danych w Azure App Service (opracowanie własne)	85
Rys. 4.11 Definicja skryptu GitHub Actions (opracowanie własne)	86
Rys. 4.12 Secrets zdefiniowane na platformie GitHub (opracowanie własne)	86
Rys. 4.13 Lista wykonanych wdrożeń (opracowanie własne).....	87
Rys. 4.14 Wynik wdrożenia na platformie Azure (opracowanie własne).....	87

STRESZCZENIE PRACY

Streszczenie w języku polskim

Tytuł: Implementacja systemu do rezerwacji wizyt pacjentów w przychodni na platformie Azure.

Streszczenie: Niniejsza praca dyplomowa przedstawia cały proces budowy systemu do rezerwacji wizyt pacjentów w przychodni przy użyciu usług chmurowych na platformie Azure. Omówione zostają cztery etapy począwszy od analizy wymagań biznesowych, przez definicję architektury systemu, aż po jego implementację i wdrożenie na chmurę obliczeniową. Wynikiem końcowym jest w pełni działający produkt możliwy do wdrożenia na produkcję.

Streszczenie w języku angielskim

Title: Implementation of a patient appointment booking system for a clinic using the Azure platform.

Abstract: This thesis describes the entire process of building a patient appointment booking system using the Azure platform. There are 4 stages that have been discussed - business requirements analysis, system architecture definition, the implementation and cloud deployment. The end result is a fully working product that can be used on production.