**Uniwersytet Jagielloński w Krakowie**

Wydział Fizyki, Astronomii i Informatyki Stosowanej

**Daniel Wielgosz**

Nr albumu: 1157140

**Projekt i implementacja usługowego serwisu ogłoszeniowego**

Praca magisterska  
na kierunku Informatyka Stosowana

Praca wykonana pod kierunkiem

dr hab. Barbara Strug

Zakład Projektowania i Grafiki Komputerowej

Kraków 2024

Spis treści

[1. Wstęp 4](#_Toc176040234)

[1.1 Uzasadnienie wyboru tematu 4](#_Toc176040235)

[1.2 Cel zaprojektowania strony internetowej 5](#_Toc176040236)

[1.3 Zakres pracy 6](#_Toc176040237)

[2. Opis istniejących rozwiązań na rynku 7](#_Toc176040238)

[2.1 Fixly 7](#_Toc176040239)

[2.2 Oferteo 8](#_Toc176040240)

[2.3 Homerun 10](#_Toc176040241)

[2.4 Podsumowanie i porównanie 11](#_Toc176040242)

[3. Projekt aplikacji 12](#_Toc176040243)

[3.1 Wymagania funkcjonalne i niefunkcjonalne 12](#_Toc176040244)

[3.1.1 Wymagania funkcjonalne 12](#_Toc176040245)

[3.1.2 Wymagania niefunkcjonalne 14](#_Toc176040246)

[3.2 Przypadki użycia 14](#_Toc176040247)

[4. Użyte technologie informatyczne 26](#_Toc176040248)

[4.1 PostgresDB 26](#_Toc176040249)

[4.2 Golang 28](#_Toc176040250)

[4.3 React 29](#_Toc176040251)

[5. Implementacja 32](#_Toc176040252)

[5.1 Wstęp 32](#_Toc176040253)

[5.2 Użyte środowiska i narzędzia 32](#_Toc176040254)

[5.3 Schemat bazy danych 33](#_Toc176040255)

[5.4 Architektura kodu po stronie backendu 35](#_Toc176040256)

[5.4.1 Utworzenie bazy danych 36](#_Toc176040257)

[5.4.2 Połączenie z bazą danych 37](#_Toc176040258)

[5.4.3 Wykonywanie operacji na bazie danych 38](#_Toc176040259)

[5.4.4 Definiowanie logiki biznesowej 39](#_Toc176040260)

[5.4.5 Obsługa zapytań HTTP 40](#_Toc176040261)

[5.5 Architektura kodu po stronie frontendu 42](#_Toc176040262)

[5.5.1 Komunikacja z backendem 42](#_Toc176040263)

[5.5.2 Model danych 43](#_Toc176040264)

[5.5.3 Logika biznesowa 43](#_Toc176040265)

[5.5.4 Wyświetlanie danych 45](#_Toc176040266)

[5.6 Implementacja wybranych funkcjonalności 46](#_Toc176040267)

[5.6.1 Rejestracja specjalisty w serwisie 46](#_Toc176040268)

[5.6.2 Wyświetlanie kalendarza dostępności specjalisty i rezerwacja usługi przez klienta 61](#_Toc176040269)

[5.6.3 Wyświetlanie kalendarza rezerwacji klienta, ich modyfikacja oraz dodawanie opinii 75](#_Toc176040270)

[6. Instrukcja obsługi 87](#_Toc176040271)

[6.1 Strony ogólnodostępne 87](#_Toc176040272)

[6.2 Instrukcja obsługi dla zalogowanych klientów 87](#_Toc176040273)

[6.3 Instrukcja obsługi dla zalogowanych specjalistów 87](#_Toc176040274)

[7. Podsumowanie 87](#_Toc176040275)

[8. Bibliografia 87](#_Toc176040276)

# Wstęp

Od wielu lat Internet towarzyszy w codzienności każdego z nas, obejmując większość sfer życia społecznego. Jego ciągły rozwój z każdym dniem daje coraz większe możliwości, od komunikacji z bliskimi, przez zapewnienie rozrywki, aż po dokonywanie rezerwacji wszelkiego rodzaju usług lub wizyt. Szczególnie to ostatnie staje się coraz powszechniejsze, obejmujące coraz więcej dziedzin, chociażby branży medycznej, która umożliwia rezerwację wizyt u lekarzy, sfery szeroko pojętej rozrywki, przez rezerwację stolików w restauracjach, czy także branży budowlanej, poprzez rezerwację usług np. hydraulików, elektryków czy stolarzy. Bez względu na obszar, dla wszystkich użytkowników elektroniczna forma rezerwacji stanowi duże ułatwienie zarówno wśród klientów jak i osób oferujących usługi. Przede wszystkim zapewnia dużą oszczędność czasu oraz pieniędzy, jest również możliwa z każdego miejsca na ziemi [1]. Dodatkowo pozwala poszerzyć horyzonty klientom oraz daje im więcej opcji, tak aby mogli oni dokonać najlepszego dla siebie wyboru. Z kolei osoby oferujące swoje usługi, mogą w ten sposób zareklamować się i dotrzeć do większego grona potencjalnych klientów oraz lepiej zarządzać swoim czasem pracy.

## Uzasadnienie wyboru tematu

Głównym powodem, dla którego autor pracy zdecydował się wybrać taki temat pracy, jest aktualność obecnej tematyki w dobie wciąż rozwijającego się Internetu oraz chęć stworzenia rozwiązania ułatwiającego rezerwację usług wszystkim użytkownikom, a zwłaszcza osobom starszym.

Rosnąca z dnia na dzień cyfryzacja obejmuje kolejne dziedziny naszego życia. Coraz więcej spraw, które do tej pory byliśmy w stanie zrealizować jedynie osobiście, teraz można załatwić nie wychodząc z domu. Jest to również pokłosie pandemii Covid, która wymusiła na rynku powstanie odpowiednich stron do rezerwacji usług, jak np. strony z rezerwacją i realizacją wizyt u lekarza czy usług fachowców z danej dziedziny. W tamtym czasie zapewniało to bezpieczeństwo użytkownikom przez unikanie możliwości zarażenia wirusem przez bezpośredni kontakt. Nie wychodząc z domu, mogli oni po kilku kliknięciach umówić się na wizytę lub zarezerwować usługę.

W obecnych czasach, w których rządzi pośpiech i praktyczność, rozwiązanie to powinno idealnie wpasować się w wymagania przeciętnego użytkownika. Stanowi ono bowiem duże ułatwienie oszczędzające przede wszystkim czas spędzony na wyszukanie np. odpowiedniego fachowca do zrealizowania konkretnej usługi. Dzięki niemu, klienci w jednym miejscu mogą wyszukać osobę, która spełni ich wymagania – nie muszą fizycznie spotkać się z danym fachowcem aby zamówić jego usługę. Dodatkowo posiłkując się ocenami innych użytkowników, zamieszczonymi w serwisie, mogą mieć pewność, że wybrana osoba zrealizuje zlecenie zgodnie z ich kryteriami. Z kolei fachowcy oferujący swoje usługi mogą dotrzeć do większego grona zainteresowanych osób oraz umocnić swoją pozycję na rynku. Oprócz tego, mając wszystkie swoje zlecenia na jednej stronie, mogą w łatwy sposób zarządzać swoim czasem i kolejnymi zleceniami.

Możliwość rezerwowania usług przez Internet może stanowić również ułatwienie dla osób starszych. Ci często ze względu na swój wiek oraz stan zdrowia nie są w stanie radzić sobie z drobnymi pracami lub naprawami. Mogą również nie mieć wystarczającej wiedzy o fachowcach działających w okolicy i reklamujących się online, a tym bardziej sił, aby spędzać czas na długotrwałym ich poszukiwaniu poza domem .

Rozwiązaniem w tej sytuacji mogą być serwisy internetowe umożliwiające rezerwację usług, jednak dla osób starszych, które na co dzień nie korzystają z tego typu udogodnień, oraz nie są zaznajomione z nowymi technologiami, poziom ich skomplikowania może okazać się nie do przejścia. Twórcy stron często zapominają o takich osobach, przez co tworzone przez nich narzędzia stają się zbyt trudne do opanowania przez seniorów. Z tego względu, jednym z powodów podjęcia takiego tematu pracy przez autora była chęć stworzenia intuicyjnego i czytelnego rozwiązania, które dzięki łatwości w obsłudze, nie będzie ograniczało osób starszych, a wręcz będzie stanowić dla nich pomoc i ułatwienie w procesie elektronicznej rezerwacji usług fachowców.

Autor przy wyborze tematu pracy kierował się również chęcią stworzenia serwisu przy wykorzystaniu nowych i popularnych technologii oraz języków programowania. Proces tworzenia narzędzia do rezerwacji usług fachowców stworzył przestrzeń do poszerzenia i ugruntowania wiedzy dotyczącej aktualnych trendów w zakresie architektury stron oraz wykorzystywanych do tego narzędzi. Oprócz tego autor mógł rozwinąć umiejętności dotyczące wyszukiwania informacji, samodzielnego rozwiązywania problemów czy właściwego analizowania i planowania dalszej pracy.

## Cel zaprojektowania strony internetowej

Obecne życie każdego człowieka bardzo mocno opiera się na technologii, która pomaga nam w wielu czynnościach naszego codziennego życia [2]. Dlatego prezentowany w pracy serwis, miał na celu przede wszystkim ułatwienie i przyspieszenie procesu rezerwacji usług fachowców, oraz łatwiejsze dotarcie do potencjalnych klientów przez usługodawców.

Za pomocą takiego serwisu użytkownicy, w sposób szybki, mogą z listy osób wybrać tę, która najbardziej odpowiada ich potrzebom, bez konieczności wychodzenia z domu lub wykonywania niepotrzebnych telefonów. Pomocne przy tym są również opinie innych klientów, a ich duża ilość powinna dać pełniejszy obraz danego fachowca oraz pewność jego niezawodności.

Dodatkowo możliwość zarządzania dokonanymi rezerwacjami z jednego miejsca pozwala użytkownikom łatwiej zaplanować swój czas. Przydatne jest to zwłaszcza dla usługodawców, którzy przy natłoku zleceń potrzebują łatwego sposobu do zarządzania nimi, co również było celem opisywanego serwisu.

Łatwość i prostota poruszania się po nim to również jeden z celów, jakie postawił sobie autor produkując te rozwiązanie. Zbyt duży poziom jego skomplikowania mógłby tylko utrudnić korzystanie z niego przez użytkowników i rodzić w nich niechęć z jego użytkowania. Dzięki swojej intuicyjności serwis skraca czas potrzebny na znalezienie fachowca, nawet w kryzysowej sytuacji.

Prostota jest również ważna w kontekście wykorzystywania serwisu przez osoby starsze. Nawet mimo małego obycia z nową technologią, mogą oni z pomocą przystępnego interfejsu wykorzystać jej potencjał i użyć do realizacji swoich potrzeb.

Kolejnym aspektem opisywanego rozwiązania jest budowania własnej marki przez fachowców. Wielu z nich trudno jest dotrzeć do szerszego grona klientów, z kolei tutaj mają szansę zostać znalezionymi przez innych, a rzetelne opinie zadowolonych klientów mogą tylko ugruntować ich pozycję na rynku.

## Zakres pracy

Niniejsza praca zawiera osiem rozdziałów. W pierwszym z nich zawarto podstawowe informacje, takie jak uzasadnienie wybranego tematu i cel pracy. Opis istniejących na rynku rozwiązań z dziedziny rezerwacji usług przedstawiono w rozdziale drugim. Z kolei w trzecim szczegółowo opisano wymagania funkcjonalne stworzonego serwisu, wraz z opisem jego funkcji, warunkami wstępnymi czy danymi wejściowymi. W kolejnym, czwartym rozdziale, zawarto przypadki użycia, w tym interakcje w ramach danych funkcjonalności systemu. W końcu autor w rozdziale piątym opisuje technologie informatyczne, jakie zostały wykorzystane do zaprojektowania aplikacji. Dalej szczegółowo opisana została implementacja poszczególnych części serwisu w rozdziale szóstym. W kolejnym rozdziale znalazła się instrukcja obsługi serwisu, pokazująca zrzuty ekranu i koje kroki, jakie należy wykonać w celu wykonania danej akcji. W rozdziale ósmym, zamykającym pracę, znalazło się miejsce na podsumowanie, wraz z opisem potencjalnego dalszego rozwoju aplikacji.

# Opis istniejących rozwiązań na rynku

Już od wielu lat, Internet traktowany jest przez użytkowników jako przestrzeń do wyszukiwania usługodawców, którzy mogliby wykonać dane zlecenie. Początkowo do tych celów służyły specjalistyczne fora, na których użytkownicy tworzyli odpowiednie wątki związane z ich problemem, oraz grupy w serwisach społecznościowych, gdzie internauci poprzez umieszczanie postów, starali się znaleźć fachowca odpowiadającego ich potrzebom [3]. Z czasem jednak, zaczęły powstawać dedykowane serwisy, umożliwiające tworzenie ofert oraz wyszukiwanie usługodawców według podanych kryteriów. Obecnie jest ich wiele na rynku, a każdy z nich różni się między sobą sposobem użytkowania czy dostępnymi funkcjonalnościami.

Poniżej przedstawiono i opisano najpopularniejsze z nich, wraz z przybliżeniem ich najważniejszych funkcji.

## Fixly

Obraz zawierający tekst, zrzut ekranu, Czcionka, Strona internetowa

Opis wygenerowany automatycznie

Rysunek 1. Strona główna *Fixly* [4]

*Fixly* jest jednym z najpopularniejszych serwisów pozwalających na rejestrowanie wykonawców oraz wyszukiwanie ich przez zainteresowanych klientów. Założony w 2017 roku przez *Grupę OLX* [5], zawiera wiele funkcji oraz narzędzi ułatwiających poruszanie się po kolejnych stronach oraz rezerwowanie usług.

Już na stronie głównej serwisu (rys. 1) podana jest informacja o dużej ilości zarejestrowanych wykonawców, co stanowi sugestię, że każdy powinien znaleźć w serwisie to czego szuka. Wyświetlona jest również lista dostępnych kategorii oraz pole wyszukiwania, za pomocą których, użytkownik może zaraz po wejściu do serwisu rozpocząć poszukiwania. Użytkownicy mogą również wybrać konkretną lokalizację wykonania usługi, co pozwala bardziej spersonalizować wyniki wyszukiwania.

W zależności od wybranej wcześniej przez użytkownika kategorii, wypełnia on odpowiedni formularz ze szczegółami opisującymi usługę jaka go interesuje – np. jej rodzaj, miejsce wykonania czy termin realizacji. Użytkownik może wprowadzić również dodatkowe dane, takie jak szczegółowy opis czy zdjęcia, aby móc jeszcze lepiej zobrazować swój problem.

Następnie usługodawcy wysyłają swoje zgłoszenia na chęć realizacji usługi do użytkownika. Mając dostęp do profili danych wykonawców, może on porównać ich oferty, przeczytać o nich opinie wystawione przez poprzednich klientów, czy sprawdzić zrealizowane przez fachowca do tej pory usługi. Dzięki temu klient może wybrać osobę, która spełni wszystkie jego oczekiwania.

W trakcie dokonywania wyboru, użytkownik może również skontaktować się z wykonawcą za pomocą podanego numeru telefonu, lub poprzez czat, aby jeszcze dokładniej opisać swój problem i w ten sposób uniknąć nieporozumień w dalszej części realizacji usługi.

Tworzenie ofert i poruszanie się po serwisie jest proste i intuicyjne. Jednak w razie problemów, użytkownicy mają możliwość zgłoszenia się o pomoc podając swój numer telefonu lub adres email w odpowiednim okienku.

Serwis posiada również sekcje, które mają zachęcić użytkowników do skorzystania z niego. Wyświetlane są komunikaty informujące np. o ilości wykonawców, którzy zareagowali na ofertę utworzoną przez innego użytkownika. Dostępny jest również blog, na którym umieszczane są wpisy stanowiące porady z różnych kategorii. Istnieje także sekcja z najpopularniejszymi usługami w ostatnim czasie, co pozwala użytkownikom szybciej utworzyć ofertę.

Serwis istnieje również jako aplikacja na telefony – jest ona łatwo dostępna, dzięki czemu użytkownicy mogą korzystać z serwisu w najwygodniejszy dla nich sposób.

## Oferteo

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

Rysunek 2. Strona główna *Oferteo* [6]

Kolejnym przykładem serwisu umożliwiającego wyszukiwanie usługodawców w różnych dziedzinach jest *Oferteo*. Podobnie jak w przypadku *Fixly*, już na stronie startowej twórcy zaznaczają swój duży wkład w rynek usługowy – podane są informacje o milionach ofert złożonych do tej pory w serwisie oraz tysiącach zarejestrowanych w nim wykonawców usług.

Oprócz sekcji informacyjnej, na stronie startowej (rys. 2) znajduje się również lista kategorii obsługiwanych przez serwis oraz pole do wyszukiwania konkretnych usług. Po wybraniu jednego ze sposobów wyszukiwania, użytkownik proszony jest o uzupełnienie formularza o szczegółowe informacje, w zależności od wybranej kategorii. Dla uzupełnienia opisu, może on również opisać zakres usługi do zrealizowania, oraz dodać odpowiadające jej zdjęcia.

Po przejściu całego procesu, serwis przygotowuje listę wykonawców spełniających podane kryteria, oraz wysyła do nich utworzoną ofertę wykonania usługi. Z poziomu tej listy, klient może wejść w rozmowę z usługodawcą za pomocą czatu, oraz odwiedzić jego profil – gdzie znajdują się szczegółowe informacje o fachowcu: dane kontaktowe, zrealizowane poprzednio usługi, opinie innych czy zakres wykonywanych prac. Wszystkie te elementy w sprawny i szybki sposób pomagają znaleźć odpowiednią osobę, mając dodatkowo pewność dokładnego zrozumienia problemu, z którym zgłasza się klient.

Po utworzeniu zlecenia, w odpowiedniej zakładce, użytkownik może również przeglądać oferty wystawione przez innych klientów, a po założeniu konta wykonawcy, może w łatwy sposób złożyć ofertę realizacji usługi.

Serwis prócz sekcji z ofertami posiada również blog z poradami. W nim użytkownicy mogą znaleźć poradniki z każdej kategorii usługowej obsługiwanej na stronie. Dzięki czemu przed stworzeniem własnej oferty, mogą oni nabyć dodatkową wiedzę z zakresu danej usługi.

W serwisie znajdują się również opinie specjalistów na temat korzystania z *Oferteo* – podkreślają w ten sposób korzyści jakie przyniosła im rejestracja w serwisie, jednocześnie zachęcając do tego nowych specjalistów

Dodatkowo serwis, na jednej z podstron, opisuje korzyści jakie mogą otrzymać zleceniodawcy, podkreślając zalety swojego rozwiązania. Dostępne są również inne podstrony, dające wskazówki, zarówno dla klientów ja i wykonawców, jak poruszać się po serwisie – wszystko po to, aby użytkownicy mogli w łatwy sposób z niego korzystać.

Oprócz wersji przeglądarkowej, serwis dostępny jest również w formie aplikacji na telefon, dzięki czemu użytkownicy mogą wygodnie z niego korzystać bez względu na miejsce czy sprzęt.

## Homerun

Obraz zawierający tekst, zrzut ekranu, Ludzka twarz, ubrania

Opis wygenerowany automatycznie

Rysunek 3. Strona główna *homerun* [7]

Ostatnim z prezentowanych serwisów do tworzenia ofert z danych kategorii, jest *homerun*. Tak jak poprzednie z przytoczonych przykładów, może on pochwalić się dużą ilością zarejestrowanych wykonawców, co potwierdza informacja na stronie głównej (rys. 3). Daje to pewność użytkownikom, że ze znalezieniem specjalisty spełniającego ich oczekiwania nie powinno być problemu.

Tworzenie oferty można zacząć w serwisie na dwa sposoby – poprzez wpisanie szukanej usługi w polu wyszukiwania, lub poprzez wybranie jednej z listy umieszczonej na stronie głównej. W niej znajdują się najpopularniejsze usługi, z których korzystali użytkownicy w ostatnim czasie – stanowi to duże ułatwienie dla nowych osób i pozwala w szybszy sposób znaleźć to czego szukają.

W kolejnym kroku, w zależności od wybranej kategorii, użytkownik wypełnia formularz ze szczegółami odnośnie usługi. W porównaniu do poprzednich serwisów, ilość informacji które musi podać użytkownik jest mniej, dzięki czemu nie traci on dużo czasu na stworzenie oferty.

Po uzupełnieniu formularza, użytkownik ma wgląd do utworzonej oferty. Wyświetlona zostaje również lista wykonawców, którzy zaoferowali wykonanie usługi – z jej poziomu użytkownik ma możliwość zapoznania się z profilem danego fachowca i wybrać tego, który najbardziej przypadnie mu do gustu. Pomocne w tym stają się szczególnie opinie poprzednich klientów, umieszczone na profilu fachowca.

Dla łatwiejszego tworzenia ofert, serwis posiada również sekcje z listą miast, w której po wybraniu jednego z nich wyświetlana jest lista możliwych do zrealizowania w nim usług – stąd klienta dzieli już tylko kilka kroków do skompletowania oferty.

Oprócz podstron służących do składania ofert, *homerun* posiada sekcję „Pomoc”, gdzie umieszczone są odpowiedzi na najczęstsze pytania użytkowników – aby ich doświadczenie korzystania z serwisu było jak najlepsze i bezproblemowe.

## Podsumowanie i porównanie

Na podstawie przytoczonych powyżej przykładów serwisów można stwierdzić, iż każdy z nich cieszy się popularnością wśród klientów i stanowi dla nich wygodne narzędzie do wyszukiwania wykonawców usług.

*Fixly* dzięki szerokiej gamie kategorii oraz bardzo szczegółowym formularzom, pozwala znaleźć fachowca najlepiej dopasowanego do preferencji klienta. *Oferteo* natomiast posiada wiele sekcji z poradami ułatwiającymi użytkownikom realizację usług oraz poruszanie po serwisie. Z kolei *homerun* dzięki uproszczonemu interfejsowi oraz formularzom, pozwala szybciej znaleźć to, czego oczekuje klient.

Oprócz przytoczonych funkcjonalności wyróżniających opisane serwisy, tym co je łączy jest możliwość wystawiania ofert do realizacji dla fachowców, przeglądanie ich profili oraz kontakt z nimi poprzez specjalny komunikator. Każde z opisanych narzędzi posiada również wielość kategorii, podkategorii oraz szczegółów, które klient musi określić, a które mogą również odpychać poziomem swojej złożoności.

Pod tym kątem, prezentowana w pracy aplikacja różni się od przytoczonych serwisów. Jest to narzędzie zapewniające podstawowe funkcjonalności, które posiadają najpopularniejsze strony w tej kategorii, przy jednoczesnym zawężeniu szczegółowości i rozbudowania, dzięki czemu narzędzie to staje się łatwiejsze w obsłudze i bardziej przystępne chociażby dla osób starszych.

# Projekt aplikacji

## Wymagania funkcjonalne i niefunkcjonalne

### Wymagania funkcjonalne

Wszystkie przytoczone powyżej serwisy posiadają szereg funkcjonalności, umożliwiających użytkownikom rezerwację usług specjalistów oraz ułatwiających im ten proces. Wiele z nich występuje we wszystkich serwisach, są też takie, które wyróżniają każdy z nich.

Opisywany w pracy serwis powinien posiadać więc funkcjonalności pozwalające na wyszukiwanie, rezerwowanie usług i zarządzanie rezerwacjami, zarówno przez klientów jak i usługodawców. Wszystkie z nich powinny być zaimplementowane w taki sposób, aby były jak najbardziej intuicyjne i łatwe w użyciu dla korzystających z serwisu użytkowników. Funkcjonalności będą podzielone według trzech ich rodzajów: zwykłych użytkowników, czyli niezalogowanych osób odwiedzających serwis, oraz klientów i specjalistów, czyli użytkowników zalogowanych. Funkcjonalności, które powinny zostać zaimplementowane dla wspomnianych grup:

* **rejestracja nowego konta**: użytkownicy serwisu powinni mieć możliwość utworzenia konta po wprowadzeniu odpowiednich danych. W przypadku konta klienta, użytkownik powinien podać imię, nazwisko, adres email, hasło do konta, oraz listę adresów (nazwa miasta, nazwa ulicy, numer budynku, opcjonalnie również numer mieszkania). Natomiast usługodawca powinien wprowadzić imię oraz nazwisko, adres email, hasło, nazwę miasta, numer telefonu, opis swojej osoby, nazwę specjalizacji, oraz wybrać z przypisanej do niej listy usługi, podając przy tym zakres cenowy ich realizacji. W oby przypadkach użytkownik powinien podać wszystkie informacje oznaczone gwiazdką („\*”). W przypadku nie wypełnienia któregoś z wymaganych pól, na ekranie zwrócona zostanie informacja o błędzie. Po naciśnięciu przycisku „Zarejestruj” w przypadku powodzenia operacji, na ekranie powinno pojawić się okienko z informacją oraz możliwością przejścia do strony logowania przez naciśnięcie przycisku „Zaloguj”,
* **logowanie do konta**: serwis powinien dać możliwość zarejestrowanym użytkownikom zalogowania się do swojego konta po podaniu adresu email oraz hasła. W przypadku błędu, zostanie on wyświetlony na ekranie, natomiast w przypadku powodzenia operacji, użytkownik zostanie przekierowany na odpowiednią stronę główną – klienta lub specjalisty,
* **wyszukiwanie specjalistów**: klient powinien móc wyszukać specjalistów według specjalizacji, miasta oraz usługi. Po naciśnięciu na stronie głównej „wyszukaj usługę”, klient z listy specjalizacji powinien wybrać jedną z nich. Następnie zostanie on przekierowany na stronę z wyświetloną listą specjalistów przypisanych do wybranej specjalizacji,
* **filtrowanie wyników wyszukiwania**: klient powinien mieć opcję filtrowania listy specjalistów w wyszukiwaniuwedług miejscowości, specjalizacji oraz usługi. Po wybraniu odpowiednich kryteriów i kliknięciu w przycisk „wyszukaj”, na ekranie wyświetlona zostanie lista specjalistów spełniających wprowadzone filtry,
* **przeglądanie listy specjalistów**: po wybraniu kryteriów wyszukiwania, użytkownik powinien mieć możliwość przeglądania listy specjalistów, wraz z podstawowymi informacjami o nich, takich jak imię, nazwisko, specjalizacja oraz miejscowość działalności,
* **przeglądanie kont specjalistów**: serwis powinien umożliwić klientowi odwiedzenie profilu każdego ze specjalistów z wyświetlonej w wyszukiwaniu listy. Po kliknięciu w przycisk „odwiedź profil” przy danym usługodawcy, klient zostanie przekierowany na stronę, na której wyświetlone zostaną szczegółowe informacje o specjaliście, takie jak imię, nazwisko, adres email, numer telefonu, miasto działalności, opis, data rejestracji w serwisie, specjalizacja, lista wykonywanych usług wraz z cenami, oceny innych klientów oraz kalendarz zarezerwowanych usług,
* **rejestracja wizyty**: klient powinien mieć dostęp do funkcji rezerwowania usług u danego specjalisty. Po wybraniu w kalendarzu konkretnej daty, na ekranie wyświetli się formularz, w którym klient powinien wybrać usługę, adres jej realizacji, oraz wprowadzić jej opis. Po kliknięciu przycisku „zarezerwuj”, klient zostanie poinformowany o powodzeniu operacji, lub o ewentualnych błędach w formularzu,
* **przeglądanie listy zarezerwowanych wizyt**: klient oraz specjalista powinien móc przeglądać listę złożonych rezerwacji na usługi. Po kliknięciu w rozsuwanym menu „moje rezerwacje”, zalogowany użytkownik zostanie przekierowany na stronę z kalendarzem wyświetlającym wizyty użytkownika – zaakceptowane, odrzucone, oraz oczekujące na akcję klienta lub specjalisty,
* **modyfikacja wizyt**: system powinien umożliwić klientom oraz specjalistom modyfikację wizyt. W przypadku specjalisty, powinien on móc zaakceptować, odrzucić wizytę lub zmienić jej szczegóły (datę wykonania, szacowaną cenę), natomiast klient powinien być w stanie, po zmianach wprowadzonych przez specjalistę, do zaakceptowania ich, odrzucenia bądź wprowadzenia własnych zmian (jak zmiana daty realizacji),
* **wystawianie ocen**: po wykonaniu usługi, klient powinien mieć opcję wystawienia oceny specjaliście. Po kliknięciu w daną usługę, klient wypełni odpowiedni formularz poprzez wybranie wartości liczbowej, wpisanie opinii w odpowiednim polu oraz kliknięcie przycisku „wystaw opinie”,
* **przeglądanie listy powiadomień**: klient oraz specjalista powinien mieć możliwość przeglądania listy powiadomień utworzonych po wykonaniu działania na rezerwacji,
* **wylogowanie**: zalogowany użytkownik musi być w stanie wylogować się z serwisu. Po otworzeniu rozsuwanego menu i kliknięciu przycisku „wyloguj”, użytkownik zostanie wylogowany i przeniesiony do strony logowania.

Przedstawione wyżej funkcjonalności będą niezbędne do właściwego funkcjonowania serwisu, a odpowiednia ich implementacja znacząco przyczyni się do stworzenia serwisu na miarę potrzeb obecnego rynku.

### Wymagania niefunkcjonalne

Oprócz wymagań funkcjonalnych, ważnym fundamentem zapewniającym wysokiej jakości doświadczenie użytkownika są również wymagania niefunkcjonalne. Definiują one atrybuty jakościowe systemu, jego właściwości, oraz pomagają zapewnić skuteczność jego funkcji, dalsze utrzymanie i rozwój. Określają one również oczekiwania co do jakości działania systemu oraz pożądanego jego zachowania.

Chcąc więc spełnić potrzeby użytkowników serwisu, powinien on posiadać poniższe wymagania:

* **użyteczność**
  + system powinien być łatwy w obsłudze oraz intuicyjny, zwłaszcza dla osób starszych, jego kolorystyka kontrastowa a design przyjazny dla użytkowników,
  + system powinien działać na wszystkich nowoczesnych przeglądarkach,
  + w formularzach, pola obowiązkowe powinny być oznaczone w inny sposób niż pola nieobowiązkowe,
* **bezpieczeństwo**
  + zalogowani użytkownicy systemu, w przypadku bezczynności, po upływie 15 minut powinni zostać automatycznie wylogowani,
  + system powinien również zabezpieczyć odpowiednie strony (np. strony z rezerwacjami) i akcje (np. rezerwacja usługi) przed nieautoryzowanym dostępem innych użytkowników,
* **dostępność / niezawodność** – system powinien być dostępny 24 godziny na dobę, 7 dni w tygodniu przez cały rok,
* **wydajność** - czas odpowiedzi na zapytanie wysłane do API nie powinien przekraczać 5 sekund,
* **konserwacja** – system powinien być łatwy w utrzymaniu i aktualizacji
* **pojemność**
  + system powinien umożliwić rejestrację dużej ilości użytkowników (kilkanaście tysięcy),
  + ilość danych przechowywanych w bazie nie będzie większa niż 32 TB.

Przedstawione i opisane wyżej wymagania niefunkcjonalne powinny zapewnić zgodność działania systemu z oczekiwaniami użytkowników oraz sprostać wymaganiom panującym na rynku.

## Przypadki użycia

Opisane powyżej wymagania funkcjonalne stanowią ogólny zarys akcji, jakie mogą wykonywać użytkownicy w systemie. Natomiast do opisu sekwencji interakcji, które muszą zajść pomiędzy użytkownikiem a serwisem w celu poprawnego wykonania akcji, służą przypadki użycia. Stanowią one również ważny element w procesie projektowania interfejsu użytkownika.

Przypadki użycia opisywanego serwisu wypisano poniżej. W opisie, „użytkownik” rozumiany jest jako osoba nieposiadająca konta w serwisie, „klientem” jest zalogowany użytkownik posiadający konto klienta, natomiast „specjalistą” jest zalogowany użytkownik posiadający konto specjalisty:

* **Rejestracja nowego konta klienta**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Użytkownik |
| Warunki wstępne | Użytkownik nie posiada konta klienta w serwisie |
| Warunki końcowe | Konto klienta istnieje w bazie danych |
| Scenariusz główny | 1. Użytkownik na stronie wyboru sposobu rejestracji wybiera opcję rejestracji konta klienta. 2. System wyświetla formularz rejestracyjny wraz z polami: imię, nazwisko, adres email, hasło, lista adresów. 3. Użytkownik wypełnia formularz rejestracyjny. 4. Użytkownik zatwierdza dane klikając przycisk „Zarejestruj”. 5. System sprawdza poprawność danych. 6. System zapisuje dane klienta w bazie danych. 7. System wyświetla na ekranie informacje o pomyślnej rejestracji. 8. Użytkownik po kliknięciu w przycisk zostaje przeniesiony na stronę logowania klienta. |
| Wyjątki | 5a. Wprowadzone dane są niepoprawne   1. System wyświetla informację o wprowadzeniu niepoprawnych danych z zaznaczeniem konkretnych pól formularza. 2. Użytkownik wprowadza zmiany zgodnie z wyświetlonymi informacjami i klika przycisk „Zarejestruj”. |

* **Rejestracja nowego konta specjalisty**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Użytkownik |
| Warunki wstępne | Użytkownik nie posiada konta specjalisty w serwisie |
| Warunki końcowe | Konto specjalisty istnieje w bazie danych |
| Scenariusz główny | 1. Użytkownik na stronie wyboru sposobu rejestracji wybiera opcję rejestracji konta specjalisty. 2. System wyświetla formularz rejestracyjny wraz z polami: imię, nazwisko, adres email, hasło, miasto, numer telefonu, nazwa specjalizacji, lista usług, opis. 3. Użytkownik wypełnia formularz rejestracyjny. 4. Użytkownik zatwierdza dane klikając przycisk „Zarejestruj”. 5. System sprawdza poprawność danych. 6. System zapisuje dane specjalisty w bazie danych. 7. System wyświetla na ekranie informacje o pomyślnej rejestracji. 8. Użytkownik po kliknięciu w przycisk zostaje przeniesiony na stronę logowania specjalisty. |
| Wyjątki | 5a. Wprowadzone dane są niepoprawne   1. System wyświetla informację o wprowadzeniu niepoprawnych danych z zaznaczeniem konkretnych pól formularza. 2. Użytkownik wprowadza zmiany zgodnie z wyświetlonymi informacjami i klika przycisk „Zarejestruj”. |

* **Logowanie na konto klienta**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Użytkownik |
| Warunki wstępne | Użytkownik posiada konto klienta w serwisie |
| Warunki końcowe | Użytkownik ma dostęp do funkcji przypisanych dla klienta |
| Scenariusz główny | 1. Użytkownik na stronie wyboru sposobu logowania wybiera opcję logowania na konto klienta. 2. System wyświetla formularz logowania wraz z polami: adres email i hasło. 3. Użytkownik wypełnia formularz logowania. 4. Użytkownik zatwierdza dane klikając przycisk „Zaloguj”. 5. System sprawdza poprawność danych. 6. System wyświetla na ekranie informacje o pomyślnym logowaniu. 7. Klient zostaje przekierowany na stronę główną klienta. |
| Wyjątki | 5a. Wprowadzone dane są niepoprawne   1. System wyświetla informację o wprowadzeniu niepoprawnych danych z zaznaczeniem konkretnych pól formularza. 2. Użytkownik wprowadza zmiany zgodnie z wyświetlonymi informacjami i klika przycisk „Zaloguj”. |

* **Logowanie na konto specjalisty**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Użytkownik |
| Warunki wstępne | Użytkownik posiada konto specjalisty w serwisie |
| Warunki końcowe | Użytkownik ma dostęp do funkcji przypisanych dla specjalisty |
| Scenariusz główny | 1. Użytkownik na stronie wyboru sposobu logowania wybiera opcję logowania na konto specjalisty. 2. System wyświetla formularz logowania wraz z polami: adres email i hasło. 3. Użytkownik wypełnia formularz logowania. 4. Użytkownik zatwierdza dane klikając przycisk „Zaloguj”. 5. System sprawdza poprawność danych. 6. System wyświetla na ekranie informacje o pomyślnym logowaniu. 7. Specjalista zostaje przekierowany na stronę główną specjalisty. |
| Wyjątki | 5a. Wprowadzone dane są niepoprawne   1. System wyświetla informację o wprowadzeniu niepoprawnych danych z zaznaczeniem konkretnych pól formularza. 2. Użytkownik wprowadza zmiany zgodnie z wyświetlonymi informacjami i klika przycisk „Zaloguj”. |

* **Przeglądanie listy specjalistów**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Użytkownik, klient |
| Warunki wstępne | - |
| Warunki końcowe | Wyświetlona na ekranie lista specjalistów |
| Scenariusz główny | 1. Użytkownik/klient klika przycisk „wyszukaj usługę”. 2. System przekierowuje użytkownika/klienta i wyświetla listę dostępnych kategorii. 3. Użytkownik/klient klika w interesującą go kategorię. 4. Użytkownik/klient zostaje przekierowany na stronę z listą specjalistów przypisanych do wybranej kategorii wraz z podstawowymi informacjami o nich. 5. Użytkownik/klient wybiera filtry i klika przycisk „Szukaj” aby wyświetlić listę specjalistów według preferencji. |
| Wyjątki | 4a. Brak specjalistów dla danej kategorii   1. System wyświetla informację braku specjalistów dla danej kategorii. 2. Użytkownik/klient wybiera filtry i klika przycisk „Szukaj”. |

* **Odwiedzenie profilu specjalisty**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Użytkownik, klient |
| Warunki wstępne | Na ekranie wyświetlona lista specjalistów |
| Warunki końcowe | Wyświetlony na ekranie profil wybranego specjalisty |
| Scenariusz główny | 1. Użytkownik/klient wybiera specjalistę z listy klikając w przycisk „odwiedź profil”. 2. System przekierowuje użytkownika/klienta na profil specjalisty. 3. System wyświetla informacje o specjaliście: szczegółowe dane, listę oferowanych usług, opinie klientów oraz kalendarz dostępności. |

* **Rezerwacja usługi**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Klient |
| Warunki wstępne | Klient znajduje się na stronie z profilem specjalisty |
| Warunki końcowe | Utworzenie nowej rezerwacji i zapisanie w bazie danych |
| Scenariusz główny | 1. Klient znajduje się na profilu specjalisty. 2. Klient klika w wybraną godzinę na kalendarzu. 3. System wyświetla formularz rezerwacji usługi wraz z polami: data, rodzaj usługi, adres realizacji, opis usługi. 4. Klient wypełnia formularz danymi i klika przycisk „Zarezerwuj”. 5. System sprawdza poprawność danych. 6. System wyświetla informacje o pomyślnym utworzeniu rezerwacji. 7. Utworzona rezerwacja zostaje wyświetlona w kalendarzu na profilu specjalisty. |
| Wyjątki | 3a. Wybrana data jest datą przeszłą   1. System wyświetla informację iż wybrana data jest datą przeszłą oraz informację o konieczności wybrania innego terminu. 2. Klient wybiera nowy termin rezerwacji.   3b. Wybrana data nakłada się na inną rezerwację   1. System wyświetla informację iż wybrana data nakłada się na istniejącą rezerwację oraz informację o konieczności wybrania innego terminu. 2. Klient wybiera nowy termin rezerwacji.   3c. Wybrana data nakłada się na urlop specjalisty   1. System wyświetla informację iż wybrana data nakłada się na urlop specjalisty oraz informację o konieczności wybrania innego terminu. 2. Klient wybiera nowy termin rezerwacji.   5a. Wprowadzone dane są niepoprawne   1. System wyświetla informację o wprowadzeniu niepoprawnych danych z zaznaczeniem konkretnych pól formularza. 2. Użytkownik wprowadza zmiany zgodnie z wyświetlonymi informacjami i klika przycisk „Zarezerwuj”. |

* **Przeglądanie listy rezerwacji klienta**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Klient |
| Warunki wstępne | Klient powinien być zalogowany w systemie |
| Warunki końcowe | Wyświetlenie kalendarza z rezerwacjami klienta |
| Scenariusz główny | 1. Klient klika w imię w prawym, górnym rogu ekranu. 2. Z rozwijanego menu wybiera opcję „moje rezerwacje”. 3. System przekierowuje klienta i wyświetla kalendarz z rezerwacjami. 4. Klient klika w rezerwację. 5. System wyświetla okno z informacjami o wybranej rezerwacji: dane specjalisty, data rozpoczęcia i zakończenia, adres realizacji, nazwa usługi, cena realizacji, opis rezerwacji. |

* **Przeglądanie listy rezerwacji specjalisty**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Specjalista |
| Warunki wstępne | Specjalista powinien być zalogowany w systemie |
| Warunki końcowe | Wyświetlenie kalendarza z rezerwacjami Specjalista |
| Scenariusz główny | 1. Specjalista klika w imię w prawym, górnym rogu ekranu. 2. Z rozwijanego menu wybiera opcję „moje rezerwacje”. 3. System przekierowuje Specjalista i wyświetla kalendarz z rezerwacjami. 4. Specjalista klika w rezerwację. 5. System wyświetla okno z informacjami o wybranej rezerwacji: dane klienta, data rozpoczęcia i zakończenia, nazwa usługi, cena realizacji, opis rezerwacji. |

* **Modyfikacja rezerwacji klienta**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Klient |
| Warunki wstępne | Klient znajduje się na stronie z rezerwacjami i posiada rezerwację oznaczoną jako „wymaga akcji” |
| Warunki końcowe | Zmiana statusu rezerwacji |
| Scenariusz główny | 1. Klient znajduje się na stronie z rezerwacjami. 2. Klient wybiera rezerwację oznaczoną jako „wymaga akcji”. 3. System wyświetla okno z informacjami o rezerwacji wraz z formularzem (data rozpoczęcia i zakończenia, adres realizacji, nazwa usługi, cena, opis usługi). 4. Klient wprowadza modyfikacje w wybranych polach (data rozpoczęcia i zakończenia, adres realizacji, opis usługi). 5. Klient naciska przycisk „Zmodyfikuj”. 6. System sprawdza poprawność danych. 7. System wyświetla informacje o pomyślnej modyfikacji rezerwacji. |
| Wyjątki | 6a. Wprowadzone dane są niepoprawne   1. System wyświetla informacje o niepoprawności wprowadzonych danych. 2. Klient zmienia dane oznaczone jako niepoprawne. 3. Klient klika przycisk „Zmodyfikuj”. |

* **Modyfikacja rezerwacji specjalisty**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Specjalista |
| Warunki wstępne | Specjalista znajduje się na stronie z rezerwacjami i posiada rezerwację oznaczoną jako „wymaga akcji” |
| Warunki końcowe | Zmiana statusu rezerwacji |
| Scenariusz główny | 1. Specjalista znajduje się na stronie z rezerwacjami. 2. Specjalista wybiera rezerwację oznaczoną jako „wymaga akcji”. 3. System wyświetla okno z informacjami o rezerwacji wraz z formularzem (data rozpoczęcia i zakończenia, nazwa usługi, cena, opis usługi). 4. Specjalista wprowadza modyfikacje w wybranych polach (data rozpoczęcia i zakończenia, cena). 5. Specjalista naciska przycisk „Zmodyfikuj”. 6. System sprawdza poprawność danych. 7. System wyświetla informacje o pomyślnej modyfikacji rezerwacji. |
| Wyjątki | 6a. Wprowadzone dane są niepoprawne   1. System wyświetla informacje o niepoprawności wprowadzonych danych. 2. Specjalista zmienia dane oznaczone jako niepoprawne. 3. Specjalista klika przycisk „Zmodyfikuj”. |

* **Odrzucenie / akceptacja rezerwacji klienta**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Klient |
| Warunki wstępne | Klient znajduje się na stronie z rezerwacjami i posiada rezerwację oznaczoną jako „wymaga akcji” / „wymaga akcji specjalisty” lub „usługa potwierdzona” |
| Warunki końcowe | Zmiana statusu rezerwacji |
| Scenariusz główny | 1. Klient znajduje się na stronie z rezerwacjami. 2. Klient wybiera rezerwację oznaczoną jako „wymaga akcji” / „wymaga akcji specjalisty” lub „usługa potwierdzona”. 3. System wyświetla okno rezerwacji wraz z informacjami (dane specjalisty, data rozpoczęcia i zakończenia, adres realizacji, nazwa usługi, cena, opis usługi). 4. Klient naciska przycisk „Zaakceptuj” lub „Odrzuć”. 5. System wyświetla informacje o pomyślnej akceptacji lub odrzuceniu rezerwacji. |

* **Odrzucenie / akceptacja rezerwacji specjalisty**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Specjalista |
| Warunki wstępne | Specjalista znajduje się na stronie z rezerwacjami i posiada rezerwację oznaczoną jako „wymaga akcji” / „wymaga akceptacji klienta” lub „usługa potwierdzona” |
| Warunki końcowe | Zmiana statusu rezerwacji |
| Scenariusz główny | 1. Specjalista znajduje się na stronie z rezerwacjami. 2. Specjalista wybiera rezerwację oznaczoną jako „wymaga akcji” / „wymaga akcji klienta” lub „usługa potwierdzona”. 3. System wyświetla okno rezerwacji wraz z informacjami (dane klienta, data rozpoczęcia i zakończenia, nazwa usługi, cena, opis usługi). 4. Specjalista naciska przycisk „Zaakceptuj” lub „Odrzuć”. 5. System wyświetla informacje o pomyślnej akceptacji lub odrzuceniu rezerwacji. |

* **Utworzenie urlopu specjalisty**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Specjalista |
| Warunki wstępne | Specjalista znajduje się na stronie z rezerwacjami |
| Warunki końcowe | Utworzenie nowego urlopu |
| Scenariusz główny | 1. Specjalista znajduje się na stronie z rezerwacjami. 2. Specjalista zaznacza na kalendarzu przedział czasowy. 3. System wyświetla okno z formularzem (data rozpoczęcia i zakończenia) wypełnionym na podstawie zaznaczonego przedziału czasowego. 4. Specjalista wprowadza modyfikacje w wybranych polach. 5. Specjalista naciska przycisk „Utwórz”. 6. System sprawdza poprawność danych. 7. System wyświetla informacje o pomyślnym utworzeniu urlopu. |
| Wyjątki | 6a. Wprowadzone dane są niepoprawne   1. System wyświetla informacje o niepoprawności wybranych dat. 2. Specjalista zmienia daty w polach oznaczonych jako niepoprawne. 3. Specjalista klika przycisk „Utwórz”. |

* **Wystawienie oceny po zrealizowaniu rezerwacji**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Klient |
| Warunki wstępne | Klient znajduje się na stronie z rezerwacjami i posiada rezerwację z przeszłości, oznaczoną jako „usługa potwierdzona” |
| Warunki końcowe | Dodanie opinii w profilu specjalisty przypisanego do rezerwacji |
| Scenariusz główny | 1. Klient znajduje się na stronie z rezerwacjami. 2. Klient wybiera rezerwację z przeszłości oznaczoną jako „usługa potwierdzona”. 3. System wyświetla okno z informacjami o rezerwacji wraz z formularzem do wystawienia opinii (ocena, komentarz). 4. Klient wprowadza dane (ocena, komentarz). 5. Klient naciska przycisk „Dodaj opinię”. 6. System sprawdza poprawność danych. 7. System wyświetla informacje o pomyślnym dodaniu opinii. |
| Wyjątki | 6a. Nie wprowadzono wymaganych danych   1. System wyświetla informacje o nie wypełnieniu danych. 2. Klient uzupełnia dane. 3. Klient klika przycisk „Dodaj opinie”. |

* **Przeglądanie listy powiadomień**

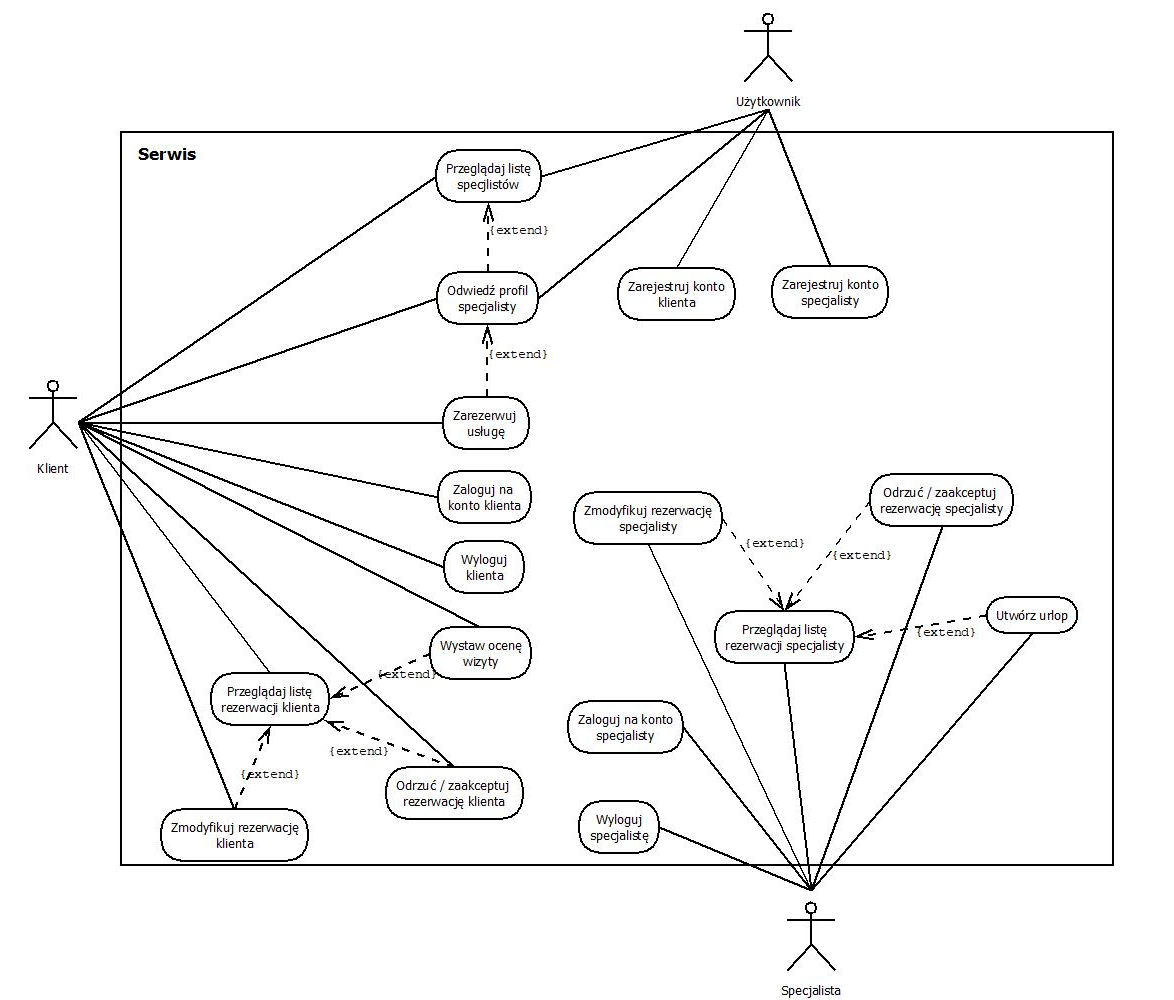
|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Klient, specjalista |
| Warunki wstępne | Klient / specjalista jest zalogowany w systemie |
| Warunki końcowe | Wyświetlenie wizyty przypisanej do wybranego powiadomienia |
| Scenariusz główny | 1. Klient / specjalista klika w ikonę dzwonka umieszczoną na pasku u góry strony. 2. System wyświetla listę powiadomień przypisaną do klienta / specjalisty. 3. Klient / specjalista wybiera powiadomienie z listy i klika w nie. 4. System przekierowuje klienta / specjalistę na stronę z rezerwacjami i wyświetla okno z informacjami o rezerwacji przypisanej do wybranego powiadomienia. 5. Jeśli wybrane powiadomienie było nowym powiadomieniem (oznaczonym kropką oraz pogrubioną czcionką), system oznacza je jako „przeczytane”. |
| Wyjątki | 2a. Brak powiadomień przypisanych dla klienta / specjalisty   1. System wyświetla informacje o braku powiadomień do wyświetlenia. |

* **Wylogowanie z konta**

|  |  |
| --- | --- |
| Sekcja | Treść |
| Aktorzy | Klient, specjalista |
| Warunki wstępne | Klient / specjalista jest zalogowany w systemie |
| Warunki końcowe | Klient / specjalista jest wylogowany |
| Scenariusz główny | 1. Klient / specjalista klika w swoje imię w prawym, górnym rogu ekranu. 2. System rozwija listę opcji do wyboru. 3. Klient / specjalista wybiera opcję „wyloguj”. 4. System wylogowuje klienta / specjalistę i przekierowuje na stronę główną. |

Opisane powyżej scenariusze przypadków użycia pozwalają stworzyć diagram przypadków użycia, który stanowić będzie ich wizualizację. Przedstawia on relacje pomiędzy aktorami występującymi w systemie a funkcjami, które są w nim dostępne. Pozwala przedstawić scenariusze, w których system wchodzi w interakcję z aktorami, oraz cele które pomaga im osiągnąć.

Diagram przypadków użycia opisywanego w pracy systemu przedstawiono poniżej (Rys. 4):

Rys. 4 – Diagram przypadków użycia serwisu [źródło własne]

# Użyte technologie informatyczne

Na etapie planowania pracy nad implementacją opisywanego, usługowego serwisu ogłoszeniowego, zdecydowano się na wybranie trzech technologii, które miały posłużyć do zarządzania bazą danych aplikacji, wykonywania operacji na tych danych, oraz wyświetlania ich użytkownikom. Głównym kryterium wyboru była ich dotychczasowa znajomość, oraz chęć poszerzenia wiedzy i umiejętności w ich zakresie przez autora. Ważnym czynnikiem była również kwestia dostępności tych rozwiązań na rynku – każde z nich jest powszechnie używane przez szerokie grono programistów na całym świecie, co powinno ułatwić rozwiązywanie problemów, w momencie ich pojawienia. Duża społeczność wpływa również pozytywnie na dalszy rozwój i aktualizacje technologii, co otwiera możliwość dalszego poszerzania serwisu o kolejne funkcjonalności w przyszłości.

Wspomniane trzy technologie, wykorzystane przy implementacji serwisu, zostały podzielone i odpowiadają za backend[[1]](#footnote-1) oraz frontend[[2]](#footnote-2). Ten pierwszy, służący do zarządzania danymi oraz odpowiadający za logikę aplikacji, zaimplementowany został przy użyciu języka *Go*, oraz z wykorzystaniem bazy danych *PostgreSQL*. Z kolei komunikacja z backendem i wyświetlanie danych użytkownikowi odbywa się przy użyciu technologii *React* z wykorzystaniem języka *TypeScript*.

## PostgresDB

Obiektowo relacyjna baza danych stanowi połączenie cech relacyjnej bazy danych oraz obiektowej bazy danych. Dane zorganizowane są w postaci tabel, składających się z wierszy, posiadających unikatowy identyfikator zwany kluczem, oraz kolumn, zawierających nazwy oraz rodzaj przechowywanych danych. Elementy tabel z kolei są połączone ze sobą relacjami za pomocą kluczy obcych. Dodatkowo, funkcje znane z baz relacyjnych są tutaj poszerzone o elementy charakterystyczne dla obiektowości: klasy, obiekty, oraz mechanizmy pozwalające na pracę z nimi, jak chociażby dziedziczenie. Stąd bazy tego typu pozwalają na operowanie na danych w taki sam sposób jak na obiektach.

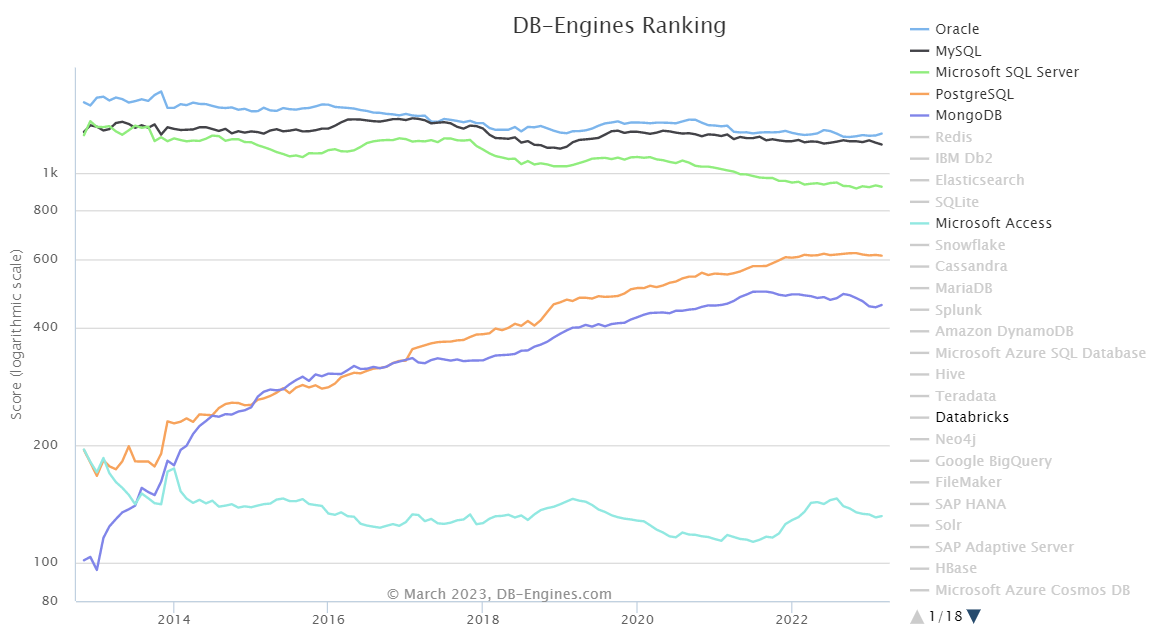
Jednym z najpopularniejszych systemów do zarządzania takimi bazami (*ORDBMS* - Object-Relational Database Management System) jest *PostgreSQL*. Powstały w 1986 r. na Uniwersytecie Kalifornijskim i działający na licencji open-source[[3]](#footnote-3) [8], określany jest jako niezawodny oraz bezpieczny, czym zdobył zaufanie wśród deweloperów. Jest on zgodny ze zbiorem właściwości przetwarzania transakcji w bazach danych takimi jak niepodzielność, spójność, izolacja i trwałość (*ACID*).

Dodatkowo system ten jest zgodny ze wszystkimi najpopularniejszymi systemami operacyjnymi, takimi jak *Windows*, *Linux* czy *MacOS*, dzięki czemu każdy użytkownik może z niego korzystać, bez względu na swoje preferencje.

Kolejną z jego zalet jest skalowalność. Doskonale sprawdza się on w projektach Big Data, gdyż system ten potrafi radzić sobie z największymi bazami danych, posiadających chociażby tabele o rozmiarze 32 TB. Dodatkowo *PostgreSQL* posiada indeksowanie *GiST* (Generalized Search Tree), zapewniające algorytmy do sortowania i wyszukiwania danych [9], co przy dużej ich ilości działa na korzyść użytkowników.

Warto również dodać, że *PostgreSQL* obsługuje format *JSON*[[4]](#footnote-4)do obsługi zapytań i wymiany danych między serwerem a aplikacją WWW. Jest on prosty w użytku, a jego składnia jest czytelna nawet dla niedoświadczonych użytkowników, co stanowi również ułatwienie przy wykonywaniu operacji na danych [10].

*PostgreSQL* zyskuje w ostatnich latach popularność wśród deweloperów. Jego zalety przyciągają nowych użytkowników, o czym świadczą badania *StackOverflow* z 2022 roku, z których wynika, że system ten w tymże roku zanotował wzrost popularności względem roku 2020 o 6% [11]. Potwierdza to również poniższy wykres (rys. 4):



Rys. 4 – Popularność systemów bazodanowych [12]

Wspomniana popularność przekłada się na zaangażowanie użytkowników w rozwój i dalsze usprawnianie opisywanego systemu. W połączeniu z jego niezawodnością, cechy te stanowiły główny powód wyboru *PostgreSQL* przez autora przy implementacji bazy danych serwisu.

## Golang

Za zarządzanie danymi oraz logikę aplikacji, odpowiada język *Go* (znany również jako *Golang*). Stworzony w 2009 roku przez trójkę programistów z firmy *Google* łączy w sobie cechy charakterystyczne zarówno dla języków dynamicznych, takich jak *JavaScript* czy *Python*, oraz języków kompilowanych, jak *C++* czy *Pascal*.

Przy jego tworzeniu, autorzy postawili sobie za cel przede wszystkim utworzenie języka prostego. Projektanci chcieli, aby nie wymagał on złożonego kodowania i był łatwy do nauczenia dla początkujących użytkowników [13]. O jego prostocie świadczy chociażby fakt, iż posiada on jedynie 25 słów kluczowych [14], co przydaje się zwłaszcza mając doświadczenie z poprzednich języków – dzięki temu proces implementacji jest szybszy i bardziej intuicyjny.

Pracę przy kodzie w *Go* ułatwiają również zaimplementowane w nim różne dobre praktyki programistyczne.

Ważną cechą języka jest również wsparcie dla współbieżności. Z pomocą gorutyn[[5]](#footnote-5) oraz kanałów, kilka czynności może być wykonywane jednocześnie z wykorzystaniem wielu wątków przy jednoczesnym minimalnym obciążeniu, co w przypadku aplikacji serwerowych jest szczególnie ważne.

Przyspieszenie pracy w *Go* zapewnia również fakt, iż jest on językiem kompilowanym. Kompilacja odbywa się tutaj o wiele szybciej niż w przypadku języków interpretowanych, a już na jej etapie, dzięki statycznemu typowaniu, programista może wykryć błędy swojego programu. Dzięki kompilacji, również proces debugowania i testowania staje się szybszy.

Dodatkowo, opisywany język posiada w sobie rozbudowaną funkcję garbage collectora, który odpowiada za zarządzanie pamięcią i jej czyszczenie, tak aby była ona jak najlepiej zoptymalizowana. Zapobiega on również wyciekom pamięci, które często mogą być trudne do manualnego skontrolowania przez programistę, przy tym odciążając go i ułatwiając jego pracę.

*Go* ze względu na swoje zalety zyskuje coraz większą popularność, a kolejne duże firmy wykorzystują go do swoich celów. Oprócz *Google*, swoje rozwiązania przy użyciu *Go* implementują takie firmy jak *Dropbox*, *Microsoft* czy *Slack*. Jego wciąż rosnące zasięgi są szczególnie widoczne w raportach *TIOBE Index*. Według nich, na początku 2024 roku, *Go* znalazł się wśród dziesięciu najpopularniejszych języków, a w połowie 2024 roku – zajął miejsce siódme, co potwierdza poniższa tabela (rys. 5):

Obraz zawierający zrzut ekranu, tekst, numer, oprogramowanie

Opis wygenerowany automatycznie

Rys. 5 – Tabela popularności języków programowania w lipcu 2024 r. [15]

Wraz z rosnącą popularnością, rośnie również cały ekosystem języka – powstają nowe biblioteki czy narzędzia, które ułatwiają tworzenie API[[6]](#footnote-6), zarządzanie bazą danych, czy tworzenie aplikacji terminalowych. Duża społeczność użytkowników, coraz więcej forów poruszających tematy związane z *Go*, pozwalają również na łatwe rozwiązywanie pojawiających się problemów. Pomocna przy tym jest również rozbudowana dokumentacja, która zawiera szczegółowe informacje na temat działania poszczególnych pakietów.

Wspomniana popularność, prostota i inne, wymienione wcześniej zalety, stanowiły główny powód wybrania *Go* jako technologii do zarządzania logiką i danymi w opisywanej aplikacji.

## React

*JavaScript* jest dynamicznym, jednowątkowym językiem programowania wysokiego poziomu, służącym do tworzenia interaktywnych stron internetowych. Oprócz wyświetlania statycznych informacji, umożliwia on obsługę zmian treści w zależności od sytuacji, a także dodawanie na stronach takich elementów jak animacje, mapy, filmy czy grafiki 2D i 3D. Wraz z technologiami *HTML*[[7]](#footnote-7) i *CSS*[[8]](#footnote-8), stanowi on podstawę przy tworzeniu stron internetowych.

Został stworzony przez Brendana Eicha, jako element pracy w firmie *Nerscape*, w 1995 roku, a jego celem w tym procesie było uczynienie go językiem prostym i łatwym [16]. Tak też się stało – *JavaScript* jest przystępny i zrozumiały nawet dla początkujących użytkowników, również ze względu na dużą społeczność oraz liczbę zasobów edukacyjnych, pomocnych w procesie nauki. Dodatkowo, do pisania kodu wystarczy jedynie notatnik i przeglądarka.

Do głównych cech, które wyróżniają ten język na tle innych należy fakt, iż jest on językiem interpretowanym. Jego kod wykonywany jest przez interpreter w czasie rzeczywistym, bez potrzeby wcześniejszej jego kompilacji. Dodatkowo wykonuje się on po stronie klienta przez przeglądarkę internetową, co umożliwia modyfikację zawartości strony bez konieczności komunikacji z serwerem po wykonaniu akcji przez użytkownika [17].

Jednak z rozwojem języka zaczął stawać się on coraz bardziej złożony, przez co przez co programiści zaczęli szukać alternatyw. Rozwiązaniem na tą sytuacje miało być powstanie w 2012 roku *TypeScriptu*, czyli języka programowania, który stanowi nadzbiór *JavaScriptu*. Jego składnia jest identyczna, jednak dodaje on funkcjonalności ułatwiające tworzenie aplikacji. Główną z nich jest silniejszy system typów, który pozwala na definiowanie wprost rodzajów zmiennych wykorzystywanych przez programistę, co w dużej mierze przyspiesza pracę oraz ułatwia utrzymanie kodu. Oprócz tego *TypeScript* dostarcza elementy charakterystyczne dla programowania obiektowego – interfejsy, modyfikatory dostępu, abstrakcje, co czyni kod bardziej przejrzystym i stanowi duże ułatwienie dla programistów znających inne języki obiektowe [18]. Dodatkowo TypeScript pozwala na tworzenie wielu niezależnych względem siebie części aplikacji za pomocą modułów. Takie podejście zapewnia lepszą organizację i zarządzanie kodem.

Wspomniane wyżej cechy, oraz fakt iż *TypeScript* jest językiem działającym na licencji open-source, przyczyniły się do szybkiego wzrostu jego popularności wśród programistów oraz dużych firm, które zaczęły z niego korzystać, takich jak *Netflix*, *LinkedIn* czy *Facebook*. Również autor pracy, biorąc pod uwagę powyższe rzeczy, zdecydował się na wybranie tego języka przy implementacji *frontendu*.

*TypeScript* obsługiwany jest przez *React* – jeden z najpopularniejszych frameworków[[9]](#footnote-9) *JavaScriptowych*, którego używanie przekłada się między innymi na większą niezawodność aplikacji. Powstał w 2011 roku z inicjatywy jednego z pracowników *Facebooka*, a głównym celem który mu przyświecał, było uproszczenie procesu budowania interfejsu. Wprowadzone przy tym rozwiązania i innowacje, które pomagają przy tworzeniu złożonych interfejsów użytkowników do różnego rodzaju aplikacji powodują, że *React* zdobywa coraz większe uznanie wśród programistów i podbija branżę programistyczną.

Jedną z nich jest fakt, iż *React* oparty jest w dużej mierze na komponentach. Raz utworzone, mogą być używane w wielu miejscach i na różnych poziomach niezależnie od siebie, co zapobiega duplikowaniu kodu, wpływa pozytywnie na zarządzanie nim i zwiększa wydajność programisty.

Ważną cechą *Reacta*, która wyróżnia go wśród innych rozwiązań tego typu, jest wysoka szybkość implementacji. Dzieje się to między innymi za sprawą dynamicznego budowania interfejsów przez *React*, w których aktualizacje strony odbywają się w czasie rzeczywistym. Stało się to możliwe dzięki używaniu przez framework własnej kopii *Virtual DOM*[[10]](#footnote-10), do której trafiają nawet najmniejsze zmiany, a dopiero później pozwalają na aktualizowanie oryginalnej struktury *DOM* [19].

Oprócz tego w *React* ważny element stanowi przepływ danych, który odbywa się tutaj z góry na dół. Elementy nadrzędne przekazują dane do elementów potomnych, a te nie mogą wpływać na komponenty rodzicielskie. Dzięki temu tylko wyznaczone przez programistę komponenty zostają zaktualizowane, co zwiększa stabilność całego kodu [20].

*React* dzięki swoim zaletom stworzył dużą społeczność programistów, którzy śledzą jego rozwój, a także sami wpływają na niego poprzez tworzenie nowych funkcjonalności i rozszerzeń. Wszystkie z nich można w łatwy sposób pobrać za pomocą managera pakietów *NPM*, a dzięki istniejącym forom i dostępnej dokumentacji można w łatwy sposób zaimplementować je i dostosować do własnego kodu.

Rosnącą popularność *Reacta* zauważają również duże firmy, które coraz częściej decydują się na wykorzystanie go do swoich celów, jak np. *Instagram* czy *Pinterest*. Powszechną popularność potwierdzają również dostarczane co roku raporty i wykresy, jak np. poniższy z 2023 roku (rys. 6):

Obraz zawierający tekst, zrzut ekranu, numer, Równolegle

Opis wygenerowany automatycznie

Rys. 6 – Wykres popularności frameworków webowych w 2023 roku, [21]

Fakt przytoczonych powyżej zalet, wynikającej z nich popularności i szerokiej gamy funkcji i rozwiązań zdecydował, że *React* został wybrany przy implementacji opisywanego serwisu usługowego.

# Implementacja

## Wstęp

W tym rozdziale przedstawiono i omówiono, w sposób szczegółowy, architekturę kodu aplikacji, jej strukturę, oraz użyte narzędzia w procesie tworzenia.

W pierwszym podrozdziale 5.2 opisano narzędzia, które posłużyły do zaimplementowania kodu, zarówno po stronie *backendu* jak i *frontendu*, oraz do testowania aplikacji. Następnie, w podrozdziale 5.3 przedstawiono i opisano schemat bazy danych, na którym opiera się cały system. W kolejnych podrozdziałach (5.4 i 5.5) opisano strukturę kodu służącego do wyświetlania oraz zarządzania danych, wraz z opisem jej elementów, natomiast w ostatnim podrozdziale 5.6 przedstawiono sposób implementacji wybranych funkcjonalności serwisu

## Użyte środowiska i narzędzia

W celu stworzenia opisywanego serwisu, wykorzystano kilka narzędzi i technologii. Do uruchomienia bazy danych wykorzystano *Dockera*, w którym odpowiednio skonfigurowany kontener inicjuje i przechowuje bazę danych i udostępnia ją na zewnątrz. Jest to narzędzie typu open source, cały czas rozwijane przez społeczność programistów, do tego jest łatwe w obsłudze i pozwala na dużą elastyczność.

Aby w łatwy sposób móc zarządzać bazą danych i znajdującymi się w niej danymi, posłużono się aplikacją *Beekeeper Studio* – to darmowe narzędzie open source, będące edytorem *SQL* wspierającym wiele systemów zarządzania relacyjnymi bazami danych: *SQLite*, *MySQL* czy *PostgeSQL*.

Do implementacji kodu w *Go* po stronie *backendu* wykorzystano środowisko *Goland*, stworzone przez firmę *JetBrains*. To darmowe narzędzie posiadające wiele udogodnień ułatwiających programowanie, takich jak inteligentne uzupełnianie kodu, debugowanie, czy integracja z systemem kontroli wersji *Git*.

Aby móc przetestować endpointy[[11]](#footnote-11), które wystawia API udostępniane przez *backend*, wykorzystano *Postmana* – darmowe narzędzie pozwalające na wywoływanie endpointów i grupowanie ich w kolekcje, które porządkują powiązanie zapytania wraz z niezbędnymi danymi i pozwalają na ponowne ich użycie w przyszłości.

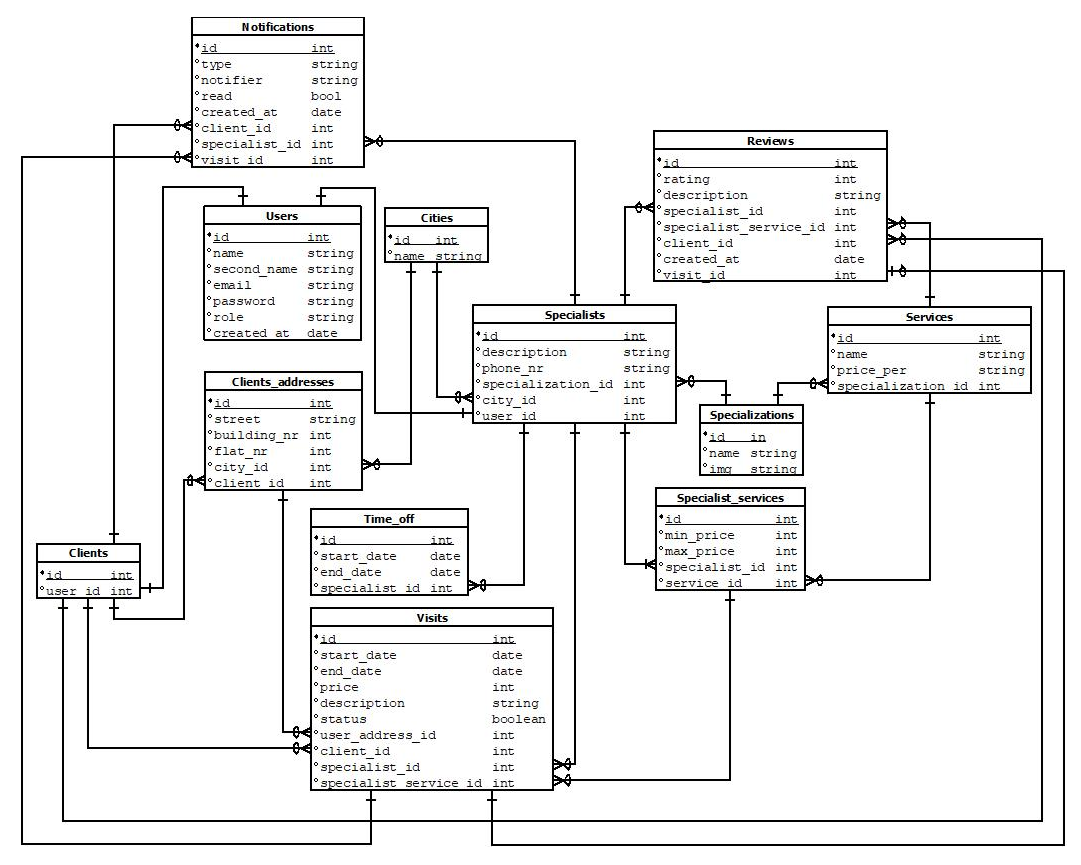
Przy tworzeniu kodu po stronie *frontendu* posłużono się innym narzędziem od firmy *JetBrains* – *WebStorm*. Podobnie jak *Goland* jest on darmowy i posiada te same cechy, dzięki czemu proces kodowania prosty i przyjemny.

Przytoczone powyżej cechy, darmowy dostęp i szereg udogodnień opisanych narzędzi zdecydowały o ich wyborze przy tworzeniu serwisu opisywanego w pracy.

## Schemat bazy danych

Przy tworzeniu bazy danych omawianego serwisu, głównym celem jaki przyświecał autorowi było optymalne rozmieszczenie danych w tabelach, oraz utworzenie odpowiednich związków między nimi, tak aby dostęp do danych był jak najszybszy.

Poniższy diagram (Rys. 7) przedstawia wygląd bazy opisywanego serwisu. W dalszej części tego podrozdziału omówiono występujące w bazie tabele, oraz relacje między nimi.



Rys. 7 - Diagram encji [źródło własne]

Bazując na powyższym diagramie, możemy wyodrębnić tabele wchodzące w skład bazy danych:

* **Cities** (miasta) – jest tabelą, której wiersze reprezentują miasta dostępne w serwisie. Każde z nich posiada numer identyfikacyjny (*id*) oraz nazwę (*name*),
* **Users** (użytkownicy) – stanowi reprezentację użytkowników zarejestrowanych w systemie. Każdy z nich posiada unikatowy identyfikator (*id*), imię (*name*), nazwisko (*second\_name*), adres email (*email*), hasło (*password*), rolę jaką pełni w systemie (role), która może przyjąć jedną z dwóch wartości: „client” (klient) lub „specialist” (specjalista), oraz datę utworzenia konta (*created\_at*),
* **Clients** (klienci) – tabela zawierająca zarejestrowanych w systemie klientów. Posiadają oni numer identyfikacyjny (*id*) oraz id użytkownika (*user\_id*) z tabeli *Users*, do którego są przypisani. Jeden klient może być przypisany tylko do jednego użytkownika, oraz jeden użytkownik może być powiązany tylko z jednym klientem,
* **Clients\_addresses** (adresy klientów) – tabela, której wiersze reprezentują adresy klientów serwisu. Adres identyfikuje się za pomocą identyfikatora (id), posiada również nazwę ulicy (**street**), numer budynku (**building\_id**) oraz numer mieszkania (**flat\_nr**). Każdy z nich przypisany jest do dokładnie jednego miasta (**city\_id**) oraz klienta (**client\_id**). Jedno miasto może być przypisane do wielu użytkowników, natomiast jeden klient może być powiązany z wieloma adresami,
* **Specialists** (specjaliści) – reprezentuje specjalistów zarejestrowanych w serwisie. Każdy wiersz tabeli posiada unikalny identyfikator (*id*), opis (*description*) oraz numer telefonu (*phone\_nr*). Każdy specjalista przypisany jest do jednej specjalizacji za pomocą klucza obcego *specialization\_id*, do jednego miasta (*city\_id*) oraz użytkownika (*user\_id*),
* **Time\_off** (urlopy) – tabela zawierająca urlopy specjalistów. Każdy z nich posiada unikalny identyfikator (*id*), datę rozpoczęcia (*start\_date*) oraz datę zakończenia (*end\_date*). Pojedynczy urlop przypisany jest do dokładnie jednego specjalisty za pomocą klucza obcego *specialist\_id*. Jeden specjalista może posiadać wiele urlopów,
* **Specializations** (specjalizacje) – jest reprezentacją specjalizacji dostępnych w serwisie. Zawierają one numer identyfikacyjny (*id*), nazwę (*name*), oraz adres URL zdjęcia (*img*),
* **Services** (usługi) – tabela, której wiersze reprezentują usługi dostępne w serwisie. Usługa posiada unikalny numer identyfikacyjny (*id*), nazwę (*name*), oraz określenie jednostki płatności (*price\_per*), która może przyjąć trzy wartości: „meter” (cena za m2), „amount” (cena za sztukę) lub pusty string „”. Każda usługa jest przypisana do dokładnie jednej specjalizacji za pomocą pola *specialization\_id*. Z kolei jedna specjalizacja może być przypisana do kilku specjalizacji,
* **Specialist\_services** (usługi specjalistów) – zawiera usługi realizowane przez specjalistów. Posiadają one unikalny identyfikator (*id*), cenę minimalną (*price\_min*), oraz cenę maksymalną (*price\_max*). Każdy wiersz tabeli przypisany jest do dokładnie jednego specjalisty poprzez klucz obcy *specialist\_id*, oraz do dokładnie jednej usługi (*service\_id*). Jeden specjalista może posiadać wiele usług specjalisty, natomiast jedna usługa może być przypisana do wielu usług specjalistów,
* **Visits** (wizyty) – reprezentuje rezerwacje usług. Wszystkie z nich posiadają unikalny numer identyfikacyjny (id), datę rozpoczęcia (*start\_date*) i zakończenia (*end\_date*), cenę (*price*) oraz opis (*description*). Oprócz tego rezerwacja zawiera również status (status), który może przyjąć jedną z pięciu wartości: „accepted” (zaakceptowana), „declined” (odrzucona), „specialist\_action\_required” (wymaga akcji specjalisty) oraz „client\_action\_required” (wymaga akcji klienta). Każda rezerwacja przypisana jest do dokładnie jednego adresu (*client\_address\_id*), dokładnie jednego klienta (*client\_id*), jednego specjalisty (*specialist\_id*) oraz jednej usługi specjalisty (*specialist\_service\_id*). Jeden adres może być przypisany do wielu rezerwacji, również jeden klient może posiadać wiele rezerwacji, tak samo specjalista może być przypisany do wielu rezerwacji, a jedna usługa specjalisty może być realizowana w ramach wielu rezerwacji,
* **Reviews** (oceny) – tabela reprezentująca oceny zrealizowanych rezerwacji usług. Pojedyncza ocena posiada wartość oceny (rating), komentarz (description), oraz datę utworzenia (created\_at). Oprócz tego, każda ocena przypisana jest do dokładnie jednego specjalisty (specialist\_id), jednej usługi specjalisty (specialist\_service\_id), jednego klienta (client\_id) oraz jednej rezerwacji (visit\_id). Specjalista może posiadać wiele ocen, jedna usługa specjalisty może być powiązana z wieloma ocenami, klient może być przypisany do wielu ocen, natomiast jedna wizyta może posiadać maksymalnie jedną ocenę,
* **Notifications** (powiadomienia) – tabela, której rekordy reprezentują powiadomienia. Pojedynczy wiersz identyfikowany jest za pomocą unikalnego identyfikatora (*id*), zawiera również status (*type*) mogący przyjąć wartość *true* lub *false*, oraz czas utworzenia (*created\_at*). Każde powiadomienie posiada również typ (*type*), który może przyjąć jedną z ośmiu wartości: „created” (oznacza utworzoną nową rezerwację), „declined” (rezerwacja odrzucona), „accepted” (rezerwacja zaakceptowana), „modified” (rezerwacja zmodyfikowana), „modified\_price” (cena rezerwacji została zmodyfikowana), „modified\_date” (data rezerwacji uległa zmianie), „modified\_address” (zmodyfikowano adres realizacji rezerwacji), „modified\_description” (zmieniono opis rezerwacji). Każda rezerwacja posiada również pole *notifier* (oznaczające typ użytkownika, który utworzył rezerwację), mogące przyjąć jedną z dwóch wartości: „client” (klient) lub „specialist” (specjalista). Pojedyncze powiadomienie przypisane jest do dokładnie jednego klienta (*client\_id*), jednego specjalisty (*specialist\_id*) oraz jednej wizyty (*visit\_id*). Klient może być powiązany z wieloma powiadomieniami, specjalista może być przypisany do wielu powiadomień, także pojedyncza wizyta może być powiązana z wieloma powiadomieniami.

## Architektura kodu po stronie backendu

Serwer, którego zadaniem jest zarządzanie bazą danych oraz logiką biznesową, został napisany przy użyciu języka *Go*. Z jego pomocą realizowany jest dostęp do tabel bazodanowych, modyfikacja przechowywanych w nich danych, oraz dostęp do nich przez klienta za pomocą utworzonego REST API[[12]](#footnote-12). W poniższym rozdziale przedstawiono i opisano schemat kodu, który do tego posłużył.

### Utworzenie bazy danych

Baza danych *PostgreSQL*, z którą łączy się serwer, inicjowana jest z pliku „init\_db.sql”. Zawarte w nim zostały definicje konkretnych tabel, wraz z ich sekwencjami, danymi początkowymi, oraz relacje między nimi określone za pomocą kluczy obcych. Przykład inicjalizacji tabeli Services został ukazany poniżej (fragment kodu 1):

1. CREATE TABLE public.services

2. (

3. id integer PRIMARY KEY NOT NULL,

4. name character varying(255) NOT NULL,

5. price\_per character varying(255),

6. specialization\_id integer NOT NULL

7. );

8.

9. ALTER TABLE public.services ALTER COLUMN id ADD GENERATED ALWAYS AS IDENTITY (

10. SEQUENCE NAME public.services\_id\_seq

11. START WITH 1 linie 26-27

12. INCREMENT BY 1

13. NO MINVALUE

14. NO MAXVALUE

15. CACHE 1

16. );

17.

18. INSERT INTO public.services (name, price\_per, specialization\_id)

19. VALUES ('Wymiana instalacji elektrycznej', 'meter', 1),

20. ('Montaż instalacji elektrycznej', 'meter', 1),

21. ('Montaż domofonu', '', 1),

22. … ;

23.

24. SELECT pg\_catalog.setval('public.services\_id\_seq', 64, true);

25.

26. ALTER TABLE public.services

27. ADD CONSTRAINT service\_price\_per CHECK ( price\_per IN ('meter', 'amount', ''));

28.

29. ALTER TABLE ONLY public.services

30. ADD CONSTRAINT specialization\_user\_id\_fk FOREIGN KEY (specialization\_id) REFERENCES public.specializations(id)

31. ON UPDATE CASCADE

32. ON DELETE CASCADE;

Fragment kodu 1. Inicjalizacja tabeli Services [źródło własne]

W powyższym fragmencie w linijkach 1-7 tworzona jest tabela z określonymi polami o odpowiednim typie, a pole *id* definiowane jest jako klucz główny. Dalej określane jest automatyczne generowanie kolejnych wartości pola *id* (linie 9-16), miejsce ma również inicjalizacja tabeli nowymi wartościami (linie 18-22), ustawiana jest również wartość sekwencji pola *id* na 64 (linia 24), określane są wartości jakie może przyjąć pole *price\_per* (26-27), oraz definiowany jest pole *specialization\_id* jako klucz obcy (linie 29-32).

Następnie w pliku „docker-compose.yml” tworzony jest kontener Dockerowy, w którym skonfigurowana zostaje baza danych *PostgeSQL*. Kod omawianego pliku znajduje się poniżej (fragment kodu 2):

1. services:  
2. postgres:

3. image: 'postgres:14.5'

4. restart: always

5. environment:  
6. POSTGRES\_DB: classifieds\_service

7. POSTGRES\_USER: admin

8. POSTGRES\_PASSWORD: password

9. logging:  
10. options:  
11. max-size: 10m

12. max-file: "3"

13. ports:  
14. - '5432:5432'

15. volumes:  
16. - ./postgres-data:/var/lib/postgresql/data 777

17. - ./sql/init\_db.sql:/docker-entrypoint-initdb.d/init\_db.sql

Fragment kodu 2. Konfiguracja kontenera zawierającego bazę danych [źródło własne]

W powyższym fragmencie określona jest wersja *PostgreSQL*, która ma być użyta w kontenerze (linia 3), określane są też zmienne środowiskowe, takie jak nazwa bazy danych (*POSTGRES\_DB*) oraz dane logowania (*POSTGRES\_USER* i *POSTGRES\_PASSWORD*) (linie 6-8). Skonfigurowane zostaje również mapowanie portu z maszyny lokalnej na port w kontenerze (linia 14), oraz mapowane są katalogi z maszyny lokalnej na katalogi wewnątrz kontenera (w tym plik inicjalizujący bazę danych) (linie 16-17), tak aby baza danych była dostępna z zewnątrz.

### Połączenie z bazą danych

Po uruchomieniu tak skonfigurowanego kontenera z bazą danych, serwer łączy się z nim za pomocą poniższego kodu (fragment kodu 3, fragment kodu 4):

1. flag.StringVar(
2. &app.DataSourceName,
3. "dsn",
4. "host=localhost dbname=classifieds\_service port=5432 user=admin password=password
5. timezone=UTC sslmode=disable connect\_timeout=5",
6. "Text for connecting to Postgres db"
7. )

Fragment kodu 3. Definicja flagi linii poleceń konfigurujące string połączenia z bazą danych [źródło własne]

1. func (app \*Application) ConnectToDB() error {  
2. db, err := sql.Open("pgx", app.DataSourceName)  
3. if err != nil {  
4. return err   
5. }

6.  
7. err = db.Ping()  
8. if err != nil {  
9. return err

10. }  
11.  
12. app.DB = &postgres.PG{DB: db}  
13. return nil  
14.}

Fragment kodu 4. Ustanowienie połączenia z bazą danych [źródło własne]

W powyższych fragmentach, na początku tworzona jest flaga (fragment kodu 3), której wartość przypisywana jest do zmiennej *DataSourceName* w instancji *app* typu *Application* (przechowującej konfigurację aplikacji, w tym dane połączenia z bazą danych). Wartość ta zawiera wszystkie dane potrzebne do połączenia z bazą danych.

Następnie w funkcji *ConnectToDB* (fragment kodu 4), dochodzi do połączenia z bazą danych poprzez wywołanie funkcji *sql.Open* (linia 2) z argumentem *DataSourceName*, który ustawiono poprzednio. Utworzone połączenie zostaje przypisane do pola *DB* w strukturze *PG* (przechowującej połączenie z bazą danych *PostgreSQL*), która następnie przypisana zostaje do pola *DB* w strukturze *app*.

### Wykonywanie operacji na bazie danych

Struktura *PG* implementuje interfejs *DAL*[[13]](#footnote-13) (fragment kodu 5), który posiada definicje metod służących do wykonywania operacji na bazie danych. Umożliwiają one dodawanie, usuwanie, pobieranie oraz modyfikację danych w określonych tabelach. Przykład takiej metody, służącej do pobierania listy usług dostępnych w serwisie przedstawiono poniżej (fragment kodu 5):

1. func (m \*PG) GetServices() ([]models.Service, error) {  
2. var services []models.Service

3. ctx, cancel := context.WithTimeout(context.Background(),timeout)  
4. defer cancel()  
5.  
6. q := sql.GetServices  
7.  
8. rows, err := m.DB.QueryContext(ctx, q)  
9. if err != nil {  
10. return nil, fmt.Errorf("error retrieving data: %w", err)  
11. }  
12. defer func() {  
13. \_ = rows.Close()  
14. }()  
15.  
16. for rows.Next() {  
17. var service models.Service  
18.  
19. err := rows.Scan(  
20. &service.Id,  
21. &service.Name,  
22. &service.PricePer,  
23. &service.SpecializationId)

24.  
25. if err != nil {  
26. return nil, fmt.Errorf("error scanning row: %w", err)  
27. }  
28.  
29. services = append(services, service)  
30. }  
31.  
32. return services, nil  
33.}

34.

35. const *GetServices* = `  
36. SELECT \* FROM services;  
37. `

Fragment kodu 5. Metoda pobierająca usługi z bazy danych [źródło własne]

W powyższym kodzie, na początku pobierane jest zapytanie *SQL* (linia 6), które ma zostać wywołane (linia 6 oraz linie 35-37). Następnie wywoływana zostaje funkcja *QueryContext*, która wykonuje pobrane wcześniej zapytanie na bazie danych (linia 8). W dalszej części kodu, w pętli (linie 16-30) dochodzi do iteracji po wierszach otrzymanych po wywołaniu zapytania, które są mapowane na obiekt typu *Service* (fragment kodu 6) i dodawane do tabeli *services* (linia 29), która zwracana jest na końcu metody.

1. type Service struct {

2. Id int `json:"id"`

3. Name string `json:"name"`

4. PricePer string `json:"price\_per"`

5. SpecializationId int `json:"specialization\_id"`

6. }

Fragment kodu 6. Struktura reprezentująca tabelę *Services* z bazy danych [źródło własne]

### Definiowanie logiki biznesowej

Za logikę biznesową serwera odpowiadają metody znajdujące się w pliku „api.go”, które powiązanie są ze strukturą *Application* (która przechowuje konfigurację aplikacji, w tym połączenie z bazą danych). Zajmują się one wykonywaniem określonych operacji na danych, ich modyfikacji, pobieraniem i umieszczaniem w bazie danych. Wykorzystują przy tym metody powiązane ze strukturą *PG* (opisaną w podrozdziale 5.4.3).

Przykład metody, która zwraca szczegółowe informacje o specjaliście (fragment kodu 7):

1. func (app \*Application) GetSpecialistDetailedInfo(w http.ResponseWriter, r \*http.Request) {

2. specialistId, err := strconv.Atoi(chi.URLParam(r, "specialist\_id"))

3. if err != nil {

5. \_ = app.errorJSON(w, err)

6. return

7. }

8.

9. specialistServices, err := app.DB.GetSpecialistServicesBySpecialistId(specialistId)

10. if err != nil {

11. \_ = app.errorJSON(w, err)

12. return

13. }

14.

15. specialistProfileInfo, err := app.DB.GetSpecialistProfileInfoBySpecialistId(specialistId)

16. if err != nil {

17. \_ = app.errorJSON(w, err)

18. return

19. }

20.

21. specialistReviews, err := app.DB.GetReviewsBySpecialistId(specialistId)

22. if err != nil {

23. \_ = app.errorJSON(w, err)

24. return

25. }

26.

27. specialist := models.SpecialistExtendedInfo{

28. Info: \*specialistProfileInfo,

29. Services: specialistServices,

30. Reviews: specialistReviews,

31. }

32.

33. \_ = app.writeJSON(w, http.StatusOK, specialist)

34. }

Fragment kodu 7. Metoda pobierająca informacje o specjaliście [źródło własne]

W zaprezentowanej metodzie, na początku pobierane są usługi realizowane przez specjalistę za pomocą metody *GetSpecialistServicesBySpecialistId* (linia 9), następnie pobierane są szczegółowe informacje o specjaliście (*GetsSpecialistProfileInfoBySpecialistId* -linia 15), na końcu pobierane są opinie przypisane do danego specjalisty (*GetReviewsBySpecialistId* – linia 21). Następnie na podstawie otrzymanych danych formowany jest nowy obiekt (linie 27-31), który następnie jest zwracany (linia 33).

### Obsługa zapytań HTTP

Aby dane mogły być pobierane, modyfikowane lub dodawane do bazy danych, stosowany jest interfejs *HTTP* - *REST API*. *API* jest to sposób interakcji pomiędzy aplikacją kliencką a serwerem w celu obsługi żądań *HTTP*, natomiast *REST* jest stylem architektonicznym, który za pomocą reguł określa sposób tej komunikacji.

Do obsługi żądań wykorzystywane jest kilka metod *HTTP*: *GET* do pobrania zawartości, *POST* do dodania nowych danych, *PATCH* do aktualizacji konkretnych danych, oraz *DELETE* do usunięcia danych.

W celu obsługi żądań, przy uruchomieniu serwera tworzony jest serwer *HTTP* nasłuchujący na porcie 8080. Jego utworzenie odbywa się za pomocą poniższego kodu (fragment kodu 8):

1. func main() {

2. ...

3. err = http.ListenAndServe(fmt.Sprintf(":%d", port), app.Routes())

4. ...

5. }

Fragment kodu 8. Utworzenie serwera *HTTP* [źródło własne]

W powyższym kodzie, jako handler do obsługi żądań przekazywana jest funkcja *Routes*, umieszczona poniżej (fragment kodu 9):

1. func (app \*Application) Routes() http.Handler {

2. mux := chi.NewRouter()

3.

4. mux.Use(middleware.Recoverer)

5. mux.Use(app.enableCORS)

6.

7. mux.Get("/cities", app.GetAllCities)

8.

9. mux.Route("/client", func(mux chi.Router) {

10. mux.Use(app.authRequired)

11. mux.Get("/reservations", app.GetClientReservations)

12. mux.Get("/info/{user\_id}", app.GetClientInfoByUserId)

13. mux.Get("/addresses/{client\_id}", app.GetClientAddressesByClientId)

14. mux.Post("/visit/create", app.CreateVisit)

15. mux.Patch("/visit/update", app.UpdateVisitByClient)

16. ...

17. })

18. ...

19. return mux

20. }

Fragment kodu 10. Implementacja metody do obsługi żądań [źródło własne]

W powyższej metodzie, na początku tworzony jest nowy *router* *HTTP* (linia 2), który przekierowuje otrzymane żądania do odpowiednich funkcji. Do niego dodawana jest funkcjonalność obsługująca *CORS*[[14]](#footnote-14), który udostępnia *API* serwera na zewnątrz (linia 5). W dalszej części definiowana jest grupa ścieżek za pomocą funkcji *Route* (linia 9), która obsługuje zapytania *GET*, *POST* i *PATCH*. Każde zapytanie ma określoną ścieżkę, do której przypisana jest funkcja, zajmująca się obsługą danego żądania.

## Architektura kodu po stronie frontendu

Komunikacja z serwerem, a przy tym pobieranie danych oraz wyświetlanie ich na ekranie odbywa się za pomocą aplikacji klienckiej napisanej przy użyciu frameworku *React*.

Jej architektura oparta jest na komponentach, gdzie w każdym z nich odbywa się zarówno pobieranie danych z serwera, zarządzanie logiką i operacje na danych, oraz renderowanie ich. Szczegółowy opis poszczególnych elementów tej architektury opisano w tym podrozdziale.

### Komunikacja z backendem

Aby dane mogły być wysyłane lub odbierane z serwera, wykorzystywana jest funkcja *fetch*, umożliwiająca realizację zapytań *HTTP*. Przykład wywołania jej znajduje się poniżej (fragment kodu 11):

1. const [cities, setCities] = useState<City[]>([])

2.

3. useEffect(() => {

4. const headers = new Headers()

5. headers.append("Content-Type", "application/json")

6.

7. const requestOptions = {

8. method: "GET",

9. headers: headers

10. }

11.

12. fetch(`http://localhost:8080/cities`, requestOptions)

13. .then((response) => response.json())

14. .then((data) => {

15. setCities(data)

16. })

17. .catch(err => {

18. console.log("Error retrieving Cities: ", err)

19. })

20. }, [])

Fragment kodu 11. Pobranie listy miast z serwera [źródło własne]

W powyższym kodzie, w linii 12 wysyłane jest zapytanie pod określony adres *URL*. W celu jego realizacji, przy wywołaniu funkcji *fetch* podawane są opcje żądania, które definiowane są w liniach 7-10: nagłówek oraz metoda *GET*. Po wykonaniu zapytania, zwracana jest tzw. obietnica (Promise) z odpowiedzią serwera, która przetwarzana jest z pomocą metod *.then* i *.cache*. Wywołanie pierwszej z nich (w linii 13) przekształca otrzymane dane na format *JSON*, natomiast jej drugie wywołanie (linie 14-16) aktualizuje stan komponentu (linia 1), przechowujący pobrane dane. Z kolei metoda *.cache* (wywołana w linii 17) służy do obsługi błędów.

Dzięki wywołaniu funkcji *fetch* w *Hooku[[15]](#footnote-15) useEffect*, pobranie danych z serwera odbywa się zaraz po wyrenderowaniu komponentu, co pozwala na wyświetlenie użytkownikowi strony z wszystkimi potrzebnymi danymi.

### Model danych

Dane, na których operują komponenty, upakowane są w odpowiednie typy, zapisane w osobnych plikach aplikacji klienckiej. Reprezentują one modele danych, do których mapowane są dane wysyłane do serwera oraz te, które są z niego pobierane.

Poniżej przedstawiono typ *TimeOff*, który reprezentuje urlop specjalisty pobrany z serwera za pomocą wywołania odpowiedniego żądania (fragment kodu 12):

1. export interface TimeOff {

2. id: number

3. start\_date: string

4. end\_date: string

5. specialist\_id: number

6. }

Fragment kodu 12. Typ reprezentujący urlop specjalisty w aplikacji klienckiej [źródło własne]

W powyższym kodzie możemy wyróżnić argumenty typu, takie jak identyfikator (*id*), czas rozpoczęcia urlopu (*start\_date*) oraz zakończenia (*end\_date*), oraz identyfikator przypisanego do niego specjalisty (*specialist\_id*).

### Logika biznesowa

Jak wspomniano na początku podrozdziału, logika biznesowa aplikacji obsługiwana jest wewnątrz komponentu. Odbywa się to za pomocą funkcji, które są wywoływane w wyniku akcji podejmowanych przez użytkownika, a które wykonują odpowiednie operacje na danych, zmieniając przy tym stan komponentu, wysyłają dane do serwera lub pobierają nowe dane, aktualizując przy tym widok.

Poniżej przedstawiono fragment kodu, biorący udział przy logowaniu klienta w serwisie (fragment kodu 13):

1. function ClientLogin() {

2. ...

3. const handleEmailChange = (event: React.FormEvent<HTMLInputElement>) => {

4. setUser({ ...user, email: event.currentTarget.value })

5. setEmailError("")

6. }

7. ...

8. function checkForm() {

9. if (user.email === "") {

10. setEmailError("Podaj adres email")

11. }

12. if (user.password === "") {

13. setPasswordError("Podaj hasło")

14. }

15. if (user.email === "" || user.password === "") {

16. return false

17. }

18. return true

19. }

20. ...

21. const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {

22. event.preventDefault()

23.

24. if (checkForm()) {

25. ...

26. fetch(`/authenticate`, {

27. ...

28. })

29. .then((response) => response.json())

30. .then((data) => {

31. if (data.error) {

32. setErrorMsg(data.message)

33. ...

34. } else {

35. ...

36. fetch(`/client/info/${data.user\_id}`, {

37. ...

38. }).then((response) => response.json())

39. .then((data) => {

40. const client: Client = {

41. id: data.id,

42. name: data.name,

43. second\_name: data.second\_name,

44. email: data.email,

45. user\_id: data.user\_id,

46. created\_at: data.created\_at

47. }

48. sessionStorage.setItem("client", JSON.stringify(client))

49. setName(data.name)

50. })

51. ...

52. setTimeout(() => {

53. setJwtToken(data.access\_token)

54. setUserRole(data.user\_role)

55. toggleRefresh(true)

56. document.body.style.cursor = "default"

57. navigate("/")

58. }, 2000)

59. }

60. })

61. .catch((err) => {

62. console.log(err)

63. })

64. }

Fragment kodu 13. Kod realizujący logowanie klienta w serwisie [źródło własne]

W powyższym kodzie znajduje się funkcja anonimowa[[16]](#footnote-16) *handleEmailChange* (linie 3-6), wywoływana w momencie zmiany pola w formularzu przechowującego adres email. W dalszej części umieszczono funkcję *checkForm* (8-19), która służy do walidacji pól formularza logowania. Owa funkcja wykorzystywana jest w funkcji *handleSubmit* (linie 21-63), która wywoływana jest w momencie, kiedy klient klika przycisk „Zaloguj”, a w której klient jest autentykowany przez wysłanie zapytania pod odpowiedni adres *URL* (linia 26). Po uzyskaniu odpowiedzi, wysyłane jest kolejne zapytanie, tym razem w celu pobrania danych klienta oraz tokenu, które zapisywane są w pamięci sesji przeglądarki (linia 48) oraz w pamięci komponentu (linia 53). Po wszystkich operacjach, klient zostaje również przeniesiony na stronę główną (linia 57).

### Wyświetlanie danych

Dane które pobierane są z serwera, przechowywane jako stan komponentu oraz przetwarzane przez jego funkcje, wyświetlane są na stronie za pomocą funkcji zwracającej JSX[[17]](#footnote-17), znajdującej się w komponencie. Dzięki wykorzystaniu odpowiednich elementów HTML w niej umieszczonych, oraz powiązaniu ich z odpowiednimi funkcjami, użytkownik może wejść w interakcje z danymi, co w rezultacie może doprowadzić do ich zmiany w bazie danych oraz aktualizacji interfejsu.

Poniżej przedstawiono fragment kodu odpowiedzialny za wyświetlenie informacji o specjaliście na profilu (fragment kodu 14):

1. function SpecialistProfile() {

2. const [specialist, setSpecialist] = useState<SpecialistExtendedInfo>()

3. …

4. return (

5. <div className="flex flex-col items-center overflow-auto h-full bg-fixed fixed w-full

pb-32">

6. ...

7. <div className="flex flex-row text-3xl font-bold mb-6 justify-center">

8. <h1>Szczegóły profilu</h1>

9. </div>

10.

11. <div className="flex flex-row">

12. ...

13. <div className="flex flex-col justify-center">

14. <p className="font-bold text-3xl pb-1">{specialist?.specialist.name}

15. {specialist?.specialist.second\_name}

16. </p>

17. </div>

18. <div className="flex flex-row justify-evenly w-full font-extrabold">

19. <div className="w-1/3">

20. <div className="flex flex-row py-2 items-center">

21. ...

22. <p className="pl-3 font-bold">

23. {specialist?.specialist.specialization}

24. </p>

25. </div>

27.   <div className="flex flex-row py-1 items-center">

28. ...

29. <p className="pl-3 font-bold">

30. {specialist?.specialist.city}

31. </p>

32. </div>

33. </div>

34. </div>

35. </div>

36. </div>

37. )}

Fragment kodu 14. Kod wyświetlający informacje nt. specjalisty [źródło własne]

W powyższym kodzie w linii 8 wyświetlany jest zwykły tekst. W dalszej części (linia 15, 16, 23 i 30) wyświetlane są kolejno imię i nazwisko, specjalizacja oraz miasto specjalisty, odczytywane z obiektu *specialist* przechowywanego jako stan komponentu (linia 2).

## Implementacja wybranych funkcjonalności

Po opisowej analizie kodu po stronie backendu jak i frontendu, w tym podrozdziale skupiono się na opisie implementacji poszczególnych funkcji serwisu. Szczegółowo opisano przepływ danych, strukturę kodu oraz zamysł który przyświecał autorowi w czasie tworzenia owych funkcjonalności

### Rejestracja specjalisty w serwisie

Strona, która pozwala na rejestrację nowego konta specjalisty w serwisie, znajduje się pod adresem „/specjalista/rejestracja”, zaś komponent odpowiedzialny za jej wyrenderowanie oraz obsługę jej logiki, znalazł się w pliku *SpecialistRegister.tsx*. W nim główny element stanowi formularz o znaczniku *<form>*, który służy do uzupełnienia zmiennej *specialist*, przechowującej takie informacje jak imię, nazwisko, adres email, hasło, identyfikator miasta numer telefonu, identyfikator specjalizacji, listę usług oraz opis. Do wypełniania pól owej zmiennej służą odpowiednie pola formularza: pola do wprowadzania danych (znacznik *<input>*) oraz wyboru z listy (znacznik *<select>*). Przykład jednego z takich pól znajduje się w poniższym fragmencie kodu (fragment kodu 15):

1. <Input

2. labelName="Email\*"

3. name="email"

4. onChange={handleSpecialistChange()}

5. placeholder="Adres email"

6. type="email"

7. value={specialist.email}

8. error={errors["email"]}

9. />

Fragment kodu 15. Implementacja pola do wpisania adresu email [źródło własne]

Dla poprawnego działania powyższego kodu, musiały się znaleźć w nim takie elementy jak:

* *labelName* – określa tekst etykiety pola, która będzie nad nim wyświetlana,
* *name* – definiuje nazwę pola,
* *onChange* – określa funkcję, która będzie wywoływana, gdy wartość pola ulegnie zmianie,
* *placeholder* – tekst, który będzie wyświetlany wewnątrz pola inputowego (do czasu aż wpisana zostanie do niego wartość),
* *type* – określa typ danych, który przyjmuje dane pole,
* *value* – ustawia bieżącą wartość pola w podanej zmiennej,
* *error* – określa jaki błąd powinien zostać wyświetlony w przypadku wpisania niepoprawnej wartości.

Funkcja *handleSpecialistChange* (linia 4), która znajduje się w powyższym fragmencie, służy do aktualizowania odpowiednich pól w zmiennej specialist. Jej implementację zaprezentowano w poniższym kodzie (fragment kodu 16):

1. const handleSpecialistChange = () => (event: React.FormEvent<HTMLInputElement>) => {

2. let value = event.currentTarget.value

3. let name = event.currentTarget.name

4. setSpecialist({

5. ...specialist,

6. [name]: value,

7. })

8. clearErrors(name)

9. }

Fragment kodu 16. Implementacja funkcji do obsługi zmian w formularzu [źródło własne]

W powyższej funkcji, *event* który jest argumentem (linia 1), pochodzi z elementu formularza i zawiera w sobie wartość wpisaną w to pole (linia 2), oraz jego nazwę (linia 3). Na tej podstawie aktualizowane jest odpowiednie pole zmiennej *specialist* (linie 4-7).

W podobny sposób jak pole zawierające adres email (fragment kodu 15), zaimplementowane zostały pola do wpisania imienia, nazwiska, hasła oraz numeru telefonu. Z kolei do wpisania przez użytkownika opisu profilu wykorzystano znacznik *<textarea>*, którego implementacje prezentuje poniższy kod (fragment kodu 17):

1. <textarea

2. id="description"

3. name="description"

4. placeholder="Pare słów o tobie"

5. value={specialist.description}

6. onChange={handleDescriptionChange}

7. rows={6}

8. cols={40}

9. className={`w-full drop-shadow-2xl border-2 focus:border-4 rounded-2xl text-2xl p-6

${errors.description ? 'border-red-500' : 'border-amber-900 mb-8'}`}

10. />

Fragment kodu 17. Implementacja pola do wpisania opisu specjalisty [źródło własne]

Elementy przedstawionego kodu są takie same jak w przypadku pola *<input>*, występuje również kilka dodatkowych:

* *id* – służy do identyfikacji pola,
* *rows* – ustawia wysokość pola tekstowego,
* *cols* – ustawia szerokość pola.

Funkcja *handleDescriptionChange* (linia 6) służy w tym przypadku do aktualizacji pola *description* zmiennej *specialist*.

Do wyboru specjalizacji przez użytkownika z listy dostępnych opcji, użyto element o znaczniku <select>, którego implementacje przedstawiono poniżej (fragment kodu 18):

1. <Select<Specialization>

2. labelName="Specjalizacja\*"

3. name="specialization\_id"

4. placeholder="Nazwa specjalizacji"

5. onChange={handleSelectChange}

6. value={specialist.specialization\_id}

7. options={specializations}

8. error={errors["specialization\_id"]}

9. disable={true}

10. />

Fragment kodu 18. Implementacja pola do wyboru specjalizacji [źródło własne]

Oprócz pól które pojawiły się w elementach typu *<input>* oraz *<textarea>*, znajduje się tutaj pole *disable*, które określa czy pierwsza, domyślna opcja w liście wyboru, powinna być możliwa do zaznaczenia przez użytkownika. Oprócz tego występuje tutaj pole *options*, do którego podawana jest lista dostępnych opcji do wyboru zapisanych w zmiennej *specializations*, która przechowuje listę specjalizacji. Jest ona pobierana z serwera w funkcji *useEffect*, zaraz po otwarciu strony, przy pomocy funkcji *fetch*. Jej wywołanie przedstawia poniższy fragment kodu (fragment kodu 19):

1. useEffect(() => {

2. const headers = new Headers()

3. headers.append("Content-Type", "application/json")

4.

5. const requestOptions = {

6. method: "GET",

7. headers: headers

8. }

9. …

10. fetch(`http://localhost:8080/specializations`, requestOptions)

11. .then((response) => response.json())

12. .then((data) => {

13. setSpecializations(data)

14. })

15. .catch(err => {

16. console.log("Error retrieving Specializations: ", err)

17. })

18. …

19. }, [])

Fragment kodu 19. Implementacja funkcji pobierającej listę specjalizacji z serwera [źródło własne]

W powyższym kodzie wysyłane jest zapytanie *GET* pod endpoint „/specializations” (linia 10). Adres ten po stronie *backendowej* obsługiwany jest przez metodę *GetAllSpecializations*, której implementacje przedstawia poniższy kod (fragment kodu 20):

1. func (app \*Application) GetAllSpecializations(w http.ResponseWriter, r \*http.Request) {

2. specializations, err := app.DB.GetSpecializations()

3. if err != nil {

4. if err != nil {

5. fmt.Println("error getting Specializations from db: ", err)

6. \_ = app.errorJSON(w, err)

7. return

8. }

9. }

10. \_ = app.writeJSON(w, http.StatusOK, specializations)

11. }

Fragment kodu 20. Implementacja metody obsługującej żądanie pobrania listy specjalizacji [źródło własne]

Metoda ta z kolei korzysta z metody *GetSpecializations* (fragment kodu 21), która pobiera listę specjalizacji z bazy danych:

1. func (m \*PG) GetSpecializations() ([]models.Specialization, error) {

2. var specializations []models.Specialization

3. ctx, cancel := context.WithTimeout(context.Background(), timeout)

4. defer cancel()

5.

6. q := sql.GetSpecializations

7. rows, err := m.DB.QueryContext(ctx, q)

8. if err != nil {

9. return nil, fmt.Errorf("error retrieving data: %w", err)

10. }

11. defer func() {

12. \_ = rows.Close()

13. }()

14.

15. for rows.Next() {

16. var specialization models.Specialization

17. err := rows.Scan(

18. &specialization.Id,

19. &specialization.Name,

20. &specialization.Img,

21. )

22. if err != nil {

23. return nil, fmt.Errorf("error scanning row: %w", err)

24. }

25. specializations = append(specializations, specialization)

26. }

27. return specializations, nil

28. }

Fragment kodu 21. Implementacja metody pobierającej specjalizacje z bazy danych [źródło własne]

W powyższej metodzie za pomocą odpowiedniego zapytania *SQL* (linia 6), pobierane są dane z bazy danych (linia 7) (id, nazwa, adres URL zdjęcia), a następnie mapowane do typu *Specialization* (linie 17-21) oraz zwracane jako lista specjalizacji (linia 27).

Po otrzymaniu tej listy, w metodzie *GetAllSpecializations* (fragment kodu 20) jest ona zwracana jako odpowiedź na zapytanie wysłane z aplikacji klienckiej (fragment kodu 20, linia 10).

Otrzymane z serwera dane są mapowane do formatu JSON (fragment kodu 19, linia 11), oraz zapisywane do zmiennej *specializations* za pomocą funkcji *setSpecializations* (linie 12-14), a następnie wyświetlane w rozsuwanej liście za pomocą elementu <select> (opisanego wyżej).

Po wybraniu przez użytkownika jednej ze specjalizacji z listy, jej identyfikator (id) jest wpisywany do pola *specialization*\_id w zmiennej *specialist*, za pomocą poniższego kodu (fragment kodu 22):

1. const handleSelectChange = (event: React.FormEvent<HTMLSelectElement>) => {

2. let name = event.currentTarget.name

3. if (name === "specialization\_id" || name === "city\_id") {

4. setSpecialist({

5. ...specialist,

6. [name]: parseInt(event.currentTarget.value),

7. services: [],

8. })

9. } else {

10. setSpecialist({

11. ...specialist,

12. [name]: event.currentTarget.value,

13. })

14. }

15. clearErrors(name)

16. }

Fragment kodu 22. Implementacja funkcji obsługującej wybranie specjalizacji z listy [źródło własne]

Po wybraniu specjalizacji przez użytkownika, na ekranie pojawia się sekcja z wyborem listy usług, które przypisane są do danej specjalizacji. Te z kolei, podobnie jak lista specjalizacji, pobierane są po otwarciu strony rejestracji, za pomocą zapytania *GET* wysłanego pod adres „/services” (fragment kodu 23):

1. useEffect(() => {

2. const headers = new Headers()

3. headers.append("Content-Type", "application/json")

4. const requestOptions = {

5. method: "GET",

6. headers: headers

7. }

8. ...

9. fetch(`http://localhost:8080/services`, requestOptions)

10. .then((response) => response.json())

11. .then((data) => {

12. setSetvices(data)

13. })

14. .catch(err => {

15. console.log("Error retrieving Services: ", err)

16. })

17. ...

18. }, [])

Fragment kodu 23. Implementacja funkcji pobierającej listę usług z serwera [źródło własne]

W serwerze, metodą która obsługuje powyższe zapytanie jest metoda *GetAllServices*, której implementacje ukazano poniżej (fragment kodu 24):

1. func (app \*Application) GetAllServices(w http.ResponseWriter, r \*http.Request) {

2. services, err := app.DB.GetServices()

3. if err != nil {

4. if err != nil {

5. fmt.Println("error getting Services from db: ", err)

6. \_ = app.errorJSON(w, err)

7. return

8. }

9. }

10. \_ = app.writeJSON(w, http.StatusOK, services)

11. }

Fragment kodu 24. Implementacja metody obsługującej żądanie pobrania listy usług [źródło własne]

Wykorzystuje ona metodę *GetServices* (fragment kodu 25), która z kolei przy użyciu odpowiedniego zapytania *SQL*, pobiera z bazy listę usług (id, nazwa, jednostka płatności, id specjalizacji), mapuje ją do odpowiedniego typu (*Service*) i zwraca.

1. func (m \*PG) GetServices() ([]models.Service, error) {

2. var services []models.Service

3. ctx, cancel := context.WithTimeout(context.Background(), timeout)

4. defer cancel()

5.

6. q := sql.GetServices

7. rows, err := m.DB.QueryContext(ctx, q)

8. if err != nil {

9. return nil, fmt.Errorf("error retrieving data: %w", err)

10. }

11. defer func() {

12. \_ = rows.Close()

13. }()

14.

15. for rows.Next() {

16. var service models.Service

17. err := rows.Scan(

18. &service.Id,

19. &service.Name,

20. &service.PricePer,

21. &service.SpecializationId,

22. )

23. if err != nil {

24. return nil, fmt.Errorf("error scanning row: %w", err)

25. }

26. services = append(services, service)

27. }

28. return services, nil

29. }

Fragment kodu 25. Implementacja metody pobierającej usługi z bazy danych [źródło własne]

Tak otrzymane dane są zwracane w metodzie *GetAllServices* (fragment kodu 24) jako odpowiedź na żądanie. Aplikacja kliencka mapuje je do formatu *JSON*, a następnie zapisuje do zmiennej *services* za pomocą metody *setServices* (fragment kodu 23, linie 10-13).

Lista usług, przefiltrowana według identyfikatora wybranej wcześniej specjalizacji, wyświetlana jest na stronie za pomocą poniższego kodu (fragment kodu 26):

1. <div className="flex flex-row items-center my-3">

2. <div className="w-2/4 flex flex-row justify-evenly">

3. <input

4. type='checkbox'

5. name={service.name}

6. value={service.id}

7. checked={specialist.services.some(s => s.service\_id === service.id)}

8. onChange={e => handleServiceChange(e, service.id)}

9. className="w-1/12"

10. />

11. {service.price\_per === "" &&

12. <p className="w-11/12 text-center">{service.name}</p>

13. }

14. {service.price\_per === "meter" &&

15. <p className="w-11/12 text-center">{service.name} (m<sup>2</sup>)</p>

16. }

17. {service.price\_per === "amount" &&

18. <p className="w-11/12 text-center">{service.name} (szt.)</p>

19. }

20. </div>

21. <input

22. type="number"

23. id="min\_price"

24. name="min\_price"

25. placeholder="Cena"

26. disabled={tempSpecialistService === undefined}

27. onChange={e => handlePriceChange(e, service.id, "min")}

28. value={tempSpecialistService?.min\_price || 0}

29. className={`w-1/4 h-14 drop-shadow-2xl border-2 focus:border-4 rounded-2xl text-2xl

px-6 border-amber-900 mr-1`}

30. />

31. <input

32. type="number"

33. id="max\_price"

34. name="max\_price"

35. placeholder="Cena"

36. disabled={tempSpecialistService === undefined}

37. onChange={e => handlePriceChange(e, service.id, "max")}

38. value={tempSpecialistService?.max\_price || 0}

39. className={`w-1/4 h-14 drop-shadow-2xl border-2 focus:border-4 rounded-2xl text-2xl

px-6 border-amber-900 ml-1`}

40. />

41. </div>

Fragment kodu 26. Implementacja kodu wyświetlającego pojedynczą usługę [źródło własne]

W powyższym kodzie, przy każdej nazwie usługi, która wyświetlana jest wraz z odpowiednią jednostką płatności (linie 11-19), wyświetlany jest element <input> typu „checkbox” – po kliknięciu w niego, wywoływana jest funkcja handleServiceChange (fragment kodu 27):

1. const handleServiceChange = (event: React.ChangeEvent<HTMLInputElement>, serviceId: number)

=> {

2. if (event.target.checked) {

3. const newService: SpecialistServiceRequest = {service\_id: serviceId, min\_price: 0,

max\_price: 0}

4. setSpecialist({

5. ...specialist,

6. services: [...specialist.services, newService]

7. })

8. } else {

9. setSpecialist({

10. ...specialist,

11. services: specialist.services.filter(service => service.service\_id !== serviceId)

12. })

13. }

14. clearErrors("services")

15. };

Fragment kodu 27. Implementacja funkcji obsługującej zaznaczenie pola typu „checkbox” [źródło własne]

Funkcja ta sprawdza, czy *checkbox* został zaznaczony przez użytkownika. Jeśli tak, tworzony jest obiekt *newService* (linia 3), który zawiera identyfikator wybranej usługi (*service\_id*), oraz cenę minimalną i maksymalną (*min\_price* oraz *max\_price*), których wartość początkowo ustawiana jest na 0. Tak stworzony obiekt dodawany jest do listy *services* w obiekcie *specialist* (linie 4-7). Z kolei, jeśli *checkbox* przypisany do danej usługi został odznaczony przez użytkownika, usługa ta usuwana jest z listy *services* (linie 9-12).

Wspomniane pola zawierające cenę minimalną i maksymalną, za pomocą których specjalista wycenia realizowane przez siebie usługi, wypełniane są za pomocą elementów typu *<input>* (fragment kodu 26, linie 21-29 oraz linie 31-39), oraz funkcji *handlePriceChange*, której implementacje ukazano poniżej (fragment kodu 28):

1. const handlePriceChange = (event: React.ChangeEvent<HTMLInputElement>, serviceId: number,

priceType: 'min' | 'max') => {

2. const updatedServices = specialist.services.map(service => {

3. if (service.service\_id === serviceId) {

4. return {

5. ...service,

6. [priceType === 'min' ? 'min\_price' : 'max\_price']:

parseInt(event.target.value, 10)

7. };

8. }

9. return service;

10. });

11. setSpecialist({

12. ...specialist,

13. services: updatedServices

14. });

15. clearErrors("services")

16. };

Fragment kodu 28. Implementacja funkcji obsługującej wpisanie ceny w formularzu [źródło własne]

W momencie wprowadzenia przez użytkownika ceny, wywoływana jest powyższa funkcja z argumentami *serviceId* (reprezentującym identyfikator usługi, do której wpisana została wartość) oraz *priceType* (określającym która cena powinna być zaktualizowana - minimalna czy maksymalna). Na ich podstawie, z listy wybranej do tej pory usług przez użytkownika (znajdującej się w zmiennej *services* w obiekcie *specialist*), wybierana jest ta z *id* równym *serviceId* (linia 3). Następnie, zgodnie z rodzajem ceny (określonym w zmiennej *priceType*), aktualizowana jest odpowiednia cena owej usługi, o wartość wprowadzoną w formularzu (linia 6). Tak zaktualizowana lista usług wybranych przez użytkownika, zapisywana jest w zmiennej *updatedServices*, a następnie przypisana do zmiennej *services* w obiekcie *specialist* aktualizując go (linie 11-14).

Ostatnim polem w formularzu wypełnianym przez użytkownika, jest pole z wyborem miasta, w którym realizowane będą usługi. Jego implementacje przedstawia poniższy kod (fragment kodu 29):

1. useEffect(() => {

2. const headers = new Headers()

3. headers.append("Content-Type", "application/json")

4. const requestOptions = {

5. method: "GET",

6. headers: headers

7. }

8. ...

9. fetch(`http://localhost:8080/cities`, requestOptions)

10. .then((response) => response.json())

11. .then((data) => {

12. setCities(data)

13. })

14. .catch(err => {

15. console.log("Error retrieving Cities: ", err)

16. })

17. ...

18. },[])

Fragment kodu 29. Implementacja funkcji pobierającej listę miast z serwera [źródło własne]

W powyższym kodzie wysyłane jest zapytanie *GET* do serwera pod adres „/cities”, który po stronie *backendowej* obsługiwany jest przez metodę *GetAllCities*, ukazaną poniżej (fragment kodu 30):

1. func (app \*Application) GetAllCities(w http.ResponseWriter, r \*http.Request) {

2. cities, err := app.DB.GetCities()

3. if err != nil {

4. fmt.Println("error getting Cities from db: ", err)

5. \_ = app.errorJSON(w, err)

6. return

7. }

8. \_ = app.writeJSON(w, http.StatusOK, cities)

9. }

Fragment kodu 30. Implementacja metody obsługującej żądanie pobrania listy miast [źródło własne]

Przedstawiona funkcja korzysta z kolei z funkcji *GetCities* (linia 2), która za pomocą odpowiedniego zapytania *SQL*, pobiera dostępne miasta (ich id oraz nazwa), konwertuje je do tablicy obiektów typu *City* i zwraca ją (fragment kodu 31):

1. func (m \*PG) GetCities() ([]models.City, error) {

2. var cities []models.City

3. ctx, cancel := context.WithTimeout(context.Background(), timeout)

4. defer cancel()

5.

6. q := sql.GetCities

7. rows, err := m.DB.QueryContext(ctx, q)

8. if err != nil {

9. return nil, fmt.Errorf("error retrieving data: %w", err)

10. }

11. defer func() {

12. \_ = rows.Close()

13. }()

14.

15. for rows.Next() {

16. var city models.City

17. err := rows.Scan(&city.Id, &city.Name)

18. if err != nil {

19. return nil, fmt.Errorf("error scanning row: %w", err)

20. }

21. cities = append(cities, city)

22. }

23. return cities, nil

24. }

Fragment kodu 31. Implementacja metody pobierającej miasta z bazy danych [źródło własne]

Otrzymana w ten sposób lista miast zwracana jest jako odpowiedź na zapytanie do aplikacji klienckiej, gdzie następnie jest zapisywana do zmiennej *cities* (fragment kodu 29, linie 11-13). Następnie lista ta wyświetlana jest za pomocą elementu <select>, który ukazano poniżej (fragment kodu 32):

1. <Select<City>

2. labelName="Miasto\*"

3. name="city\_id"

4. placeholder="Nazwa miasta"

5. onChange={handleSelectChange}

6. value={specialist.city\_id}

7. options={cities}

8. error={errors["city\_id"]}

9. disable={true}

10. />

Fragment kodu 32. Implementacja kodu wyświetlającego listę miast [źródło własne]

Po wyborze przez użytkownika jednego z miast, jego id przypisywane jest do zmiennej *city\_id* w obiekcie *specialist* za pomocą funkcji *handleSelectChange* (fragment kodu 22).

Kiedy użytkownik wypełni wszystkie wymagane pola, wybierze również miasto, specjalizację, listę usług, oraz kliknie przycisk „Zarejestruj”, uruchomiona zostanie funkcja *handleSubmit*, której implementację ukazano poniżej (fragment kodu 33):

1. const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {

2. event.preventDefault()

3. if (checkForm()) {

4. const headers = new Headers()

5. headers.append("Content-Type", "application/json")

6. const method = "POST"

7.

8. let requestOptions = {

9. body: JSON.stringify(specialist),

10. method: method,

11. headers: headers,

12. }

13. fetch(`/register\_specialist`, requestOptions)

14. .then((response) => response.json())

15. .then((data) => {

16. if (data.error) {

17. setErrorMsg(data.message)

18. Swal.fire({

19. didOpen: () => setShowErrorMsg(true),

20. didClose: () => setShowErrorMsg(false),

21. showConfirmButton: false,

22. })

23. } else {

24. Swal.fire({

25. didOpen: () => setShowSuccessMsg(true),

26. didClose: () => setShowSuccessMsg(false),

27. showConfirmButton: false,

28. })

29. }

30. })

31. .catch((err) => {

32. console.log(err)

33. })

34. } else {

35. console.log("ERRORS")

36. }

37. }

Fragment kodu 33. Implementacja metody realizującej rejestrację nowego specjalisty [źródło własne]

W powyższym kodzie, na początku za pomocą funkcji *checkForm* (linia 3) sprawdzana jest poprawność danych zapisanych w zmiennych obiektu *specialist* (które były wypełniane w czasie uzupełniania przez użytkownika formularza). W przypadku błędów, informacja o nich zwracana jest na ekranie użytkownika. Kiedy jednak pola formularza zostały poprawnie uzupełnione, pod adres „/register\_specialist” wysyłane jest zapytanie *POST* (linia 13). Wraz z nim wysyłany jest przekształcony obiekt *specialist* jako ciąg znaków w formacie *JSON* (linia 9). Po stronie backendowej adres ten obsługiwany jest przez metodę CreateSpecialist, której implementację przedstawiono poniżej (fragment kodu 34):

1. func (app \*Application) CreateSpecialist(w http.ResponseWriter, r \*http.Request) {

2. var newSpecialist models.SpecialistRegister

3. err := app.readJSON(w, r, &newSpecialist)

4. if err != nil {

5. \_ = app.errorJSON(w, err)

6. return

7. }

8.

9. specialist, err := app.DB.GetUserByEmailAndRole(newSpecialist.Email, "specialist")

10. if specialist != nil {

11. \_ = app.errorJSON(w, fmt.Errorf("specjalista o adresie email '%s' już

istnieje", newSpecialist.Email), http.StatusFound)

12. return

13. }

14.

15. userId, err := app.DB.CreateUser(newSpecialist.Name, newSpecialist.SecondName,

newSpecialist.Email, newSpecialist.Password, "specialist")

16. if err != nil {

17. \_ = app.errorJSON(w, err)

18. return

19. }

20.

21. specialistId, err := app.DB.CreateSpecialist(newSpecialist.PhoneNr,

newSpecialist.Description, newSpecialist.CityId, userId,

newSpecialist.SpecializationId)

22. if err != nil {

23. \_ = app.errorJSON(w, err)

24. return

25. }

26. for \_, service := range newSpecialist.Services {

27. \_, err := app.DB.CreateSpecialistService(service.MinPrice, service.MaxPrice,

specialistId, service.ServiceId)

28. if err != nil {

29. \_ = app.errorJSON(w, err)

30. return

31. }

32. }

33. \_ = app.writeJSON(w, http.StatusOK, specialistId)

34. }

Fragment kodu 34. Implementacja metody obsługującej żądanie utworzenia nowego specjalisty [źródło własne]

W powyższym kodzie, na początku odczytywane są dane *JSON* z ciała żądania i dekodowane są do obiektu *newSpecialist*. Następnie za pomocą metody *GetUserByEmailAndRole* (fragment kodu 35) sprawdzane jest, czy w tabeli *users* w bazie danych istnieje już użytkownik o podanym adresie email oraz roli „specialist” (linia 9).

1. func (m \*PG) GetUserByEmailAndRole(email string, role string) (\*models.User, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.GetUserByEmailAndRole

6. var user models.User

7. row := m.DB.QueryRowContext(ctx, q,

8. email,

9. role,

10. )

12. err := row.Scan(&user.Id, &user.Name, &user.SecondName, &user.Email, &user.Password,

&user.CreatedAt, &user.Role)

13. if err != nil {

14. return nil, fmt.Errorf("error getting User: %w", err)

15. }

16. return &user, nil

17. }

Fragment kodu 35. Implementacja metody pobierającej użytkownika z bazy na podstawie adresu email oraz roli [źródło własne]

W przedstawionej metodzie wykonywane jest zapytanie *SQL* wraz z podanymi do funkcji argumentami, a następnie wynik tego zapytania mapowany jest do zmiennej *user* typu *User*. Jeśli użytkownik istnieje i mapowanie zostanie wykonane poprawnie, zmienna *user* zostanie zwrócona przez metodę. Wtedy metoda wywołująca (*CreateSpecialist*), w odpowiedzi na zapytanie zwróci informacje o błędzie, wraz z kodem błędu (fragment kodu 34, linie 10-13). W przeciwnym wypadku stworzony zostanie nowy użytkownik za pomocą metody *CreateUser* (linia 15) (fragment kodu 36), przy której wywołaniu podawane są imię, nazwisko, adres email, hasło, oraz rola nowego użytkownika.

1. func (m \*PG) CreateUser(name string, secondName string, email string, password string, role

string) (int, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.CreateUser

6. var newUserId int

7. err := m.DB.QueryRowContext(ctx, q,

8. name,

9. secondName,

10. email,

11. password,

12. time.Now().Add(2\*time.Hour),

13. role,

14. ).Scan(&newUserId)

15. if err != nil {

16. return 0, fmt.Errorf("cannot create new User: %w", err)

17. }

18. return newUserId, nil

19. }

Fragment kodu 36. Implementacja metody tworzącej nowego użytkownika w bazie [źródło własne]

Powyższa metoda wykonuje odpowiednie zapytanie *SQL* wraz z podanymi argumentami tworząc przy tym nową instancję w tabeli *users*. W przypadku poprawnego utworzenia nowego użytkownika, zwrócony zostaje jego identyfikator. Następnie wywoływana jest metoda *CreateSpecialist* (fragment kodu 34, linia 21), która odpowiedzialna jest za utworzenie nowego specjalisty w tabeli *specialists* na podstawie podanych argumentów (numer telefonu, opis, id miasta, id nowo utworzonego użytkownika oraz id specjalizacji). Jej implementację ukazano poniżej (fragment kodu 37):

1. func (m \*PG) CreateSpecialist(phoneNr string, description string, cityId int, userId int,

specializationId int) (int, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.CreateSpecialist

6. var newSpecialistId int

7. err := m.DB.QueryRowContext(ctx, q,

8. phoneNr,

9. description,

10. cityId,

11. userId,

12. specializationId,

13. ).Scan(&newSpecialistId)

14. if err != nil {

15. return 0, fmt.Errorf("cannot create new Specialist: %w", err)

16. }

17. return newSpecialistId, nil

18. }

Fragment kodu 37. Implementacja metody tworzącej nowego specjalistę w bazie [źródło własne]

W metodzie tej wykonywanie jest zapytanie *SQL* odpowiedzialne za utworzenie nowego specjalisty na podstawie informacji podanych jako argumenty metody. W przypadku poprawnego utworzenia rekordu w bazie danych, zwrócony zostanie przez metodę identyfikator nowego specjalisty.

Po wykonaniu metody *CreateSpecialist*, wykonywana jest pętla po wszystkich usługach zapisanych w zmiennej *newSpecialist.Services* (fragment kodu 34, linie 26-32). W niej, dla każdej usługi wykonywana jest metoda CreateSpecialistService wraz z argumentami zawierającymi informacje o usłudze (cena minimalna, cena maksymalna, id usługi) oraz identyfikatorem specjalisty. Implementację wspomnianej metody przedstawia poniższy kod (fragment kodu 38):

1. func (m \*PG) CreateSpecialistService(minPrice int, maxPrice int, specialistId int, serviceId

int) (int, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.CreateSpecialistService

6. var newSpecialistId int

7. err := m.DB.QueryRowContext(ctx, q,

8. minPrice,

9. maxPrice,

10. specialistId,

11. serviceId,

12. ).Scan(&newSpecialistId)

13.

14. if err != nil {

15. return 0, fmt.Errorf("cannot create new Client: %w", err)

16. }

17. return newSpecialistId, nil

18. }

Fragment kodu 38. Implementacja metody tworzącej nową usługę specjalisty w bazie [źródło własne]

W powyższym fragmencie kodu, wywoływane jest odpowiednie zapytanie *SQL* wraz z danymi podanymi jako argumenty przy wywołaniu metody. Dzięki temu tworzona jest nowa usługa specjalisty w bazie danych, a jej id zwracane jest na końcu metody.

Po pomyślnym wykonaniu wszystkich metod w *CreateSpecialist* (fragment kodu 34), metoda jako odpowiedź na zapytanie zwraca id nowo utworzonego specjalisty oraz kod informujący o pomyślnym wykonaniu operacji. Klient odbiera tę informację i wyświetla na ekranie okno z wiadomością o pomyślnym zarejestrowaniu, oraz z przyciskiem przekierowującym użytkownika na stronę logowania dla specjalistów. Za wyrenderowanie wspomnianego okna, odpowiada poniższy kod (fragment kodu 39):

1. {showSuccessMsg &&

2. createPortal(

3. <div className="flex flex-col items-center">

4. <h1 className="text-3xl font-bold my-4">Pomyślnie zarejestrowano!</h1>

5. <p className="text-xl my-2">

6. Kliknij poniższy przycisk aby przejść do logowania

7. </p>

8. <Link to="/specjalista/login" onClick={() => Swal.close()}

9. className="flex w-52 justify-center items-center mt-5">

10. <div

11. className="border-4 border-amber-900 text-white rounded-2xl cursor-

pointer p-2 transition ease-in-out delay-0 bg-amber-900

hover:border-amber-900 hover:bg-white hover:text-amber-900

duration-300 ...">

12. <span className="mx-3 my-2 text-xl">zaloguj</span>

13. </div>

14. </Link>

15. </div>,

16. Swal.getHtmlContainer()!,

17. )

18. }

19.

Fragment kodu 39. Implementacja kodu wyświetlającego informację o pomyślnym zarejestrowaniu w serwisie [źródło własne]

### Wyświetlanie kalendarza dostępności specjalisty i rezerwacja usługi przez klienta

Dokonywanie rezerwacji usług przez klientów odbywa się na profilu konkretnego specjalisty, który znajduje się pod adresem „/specjalista/szczegóły”, a za którego wyrenderowanie i logikę biznesową odpowiada plik *SpecialistProfile.tsx*. Owa strona oprócz szczegółowych informacji o danym specjaliście, wyświetla również opinie wystawione przez klientów oraz kalendarz dostępności specjalisty, poprzez który klienci mogą dokonywać rezerwacji usług, a który wyświetlany jest za pomocą komponentu *react-big-calendar*, którego implementację przedstawia poniższy kod (fragment kodu 40):

1. <Calendar

2. culture={"pl"}

3. localizer={localizer}

4. events={visits}

5. defaultView={Views.WEEK}

6. step={15}

7. views={[Views.WEEK]}

8. min={new Date(0, 0, 0, 6, 0, 0)}

9. max={new Date(0, 0, 0, 22, 0, 0)}

10. onSelectSlot={handleSelectSlot}

11. startAccessor={(event: VisitCalendar) => event.info.start\_date}

12. endAccessor={(event: VisitCalendar) => event.info.end\_date}

13. style={{

14. height: (specialist?.reviews && specialist?.reviews.length! <= 2) ? 900 : 130

15. fontSize: "x-large"

16. }}

17. messages={{

18. previous: "Poprzedni",

19. next: "Następny",

20. today: "Dziś",

21. }}

22. eventPropGetter={eventBackgroundAdjustment}

23. selectable={true}

24. components={{

25. event: EventComponent

26. }}

27. onSelectEvent={(event) => alert(event.service.name)}

28. slotPropGetter={timeOffSlotAdjustment}

29. />

Fragment kodu 40. Implementacja kodu renderującego kalendarz dostępności lekarza w profilu [źródło własne]

W powyższym kodzie znajduje się wiele zmiennych, które wpływają na wygląd i zachowanie kalendarza:

* *culture* – ustawia język kalendarza na polski,
* *localizer* – odpowiada za prawidłowe wyświetlanie dat w odpowiednim formacie,
* *events* – zawiera listę wydarzeń (w tym wypadku wizyt zarezerwowanych u danego specjalisty), które mają być wyświetlone w kalendarzu,
* *defaultView* – ustawia domyślny widok, w jakim wyświetlony będzie kalendarz. W tym wypadku jest to widok tygodniowy,
* *views* – ustawia widok tygodniowy jako jedyny dostępny (widok dzienny i miesięczny nie są dostępne),
* *step* – ustawia długość segmentów, na które podzielony jest kalendarz, na 15 minut,
* *min* i *max* – ustawia zakres godzinowy, w jakim wyświetlany jest kalendarz, na przedział 6:00 – 22:00,
* *onSelectSlot* – przypisuje funkcję, która zostanie wywołana po kliknięciu przez klienta przedziału czasowego w kalendarzu,
* *startAccessor* i *endAccessor* – określa które pola obiektu event powinny być brane jako data rozpoczęcia i zakończenia wyświetlanej rezerwacji,
* *style* – określa niestandardowy styl dla kalendarza,
* *messages* – ustawia niestandardowy tekst dla przycisków, w tym wypadku pozwalających na nawigację po kolejnych tygodniach kalendarza,
* *eventPropGetter* – funkcja dostosowująca wygląd rezerwacji w kalendarzu,
* *selectable* – umożliwia interakcje z kalendarzem przez klienta,
* *components* – dostosowuje sposób renderowania rezerwacji w kalendarzu, za pomocą przypisanego komponentu,
* *onSelectEvent* – funkcja wywoływana w momencie kliknięcia przez klienta w rezerwację,
* *slotPropGetter* – funkcja pozwalająca na oznaczenie poszczególnych slotów czasowych w kalendarzu jako niedostępne.

Aby kalendarz mógł wyświetlić rezerwacje przypisane do specjalisty, muszą one zostać wcześniej pobrane. Dzieje się to po wejściu klienta na profil dzięki wywołaniu funkcji *getSpecialistVisits* (fragment kodu 41) wraz z identyfikatorem specjalisty przekazywanym jako dodatkowe dane podczas nawigacji na stronę:

1. useEffect(() => {

2. ...

3. getSpecialistVisits(location.state.specialistId)

4. ...

5. }, [jwtToken, location.state.specialistId, navigate, userRole])

6.

7. function getSpecialistVisits(specialistId: number) {

8. const headers = new Headers()

9. headers.append("Content-Type", "application/json")

10. const requestOptions = {

11. method: "GET",

12. headers: headers,

13. }

14. fetch(`http://localhost:8080/visits/${specialistId}/0`, requestOptions)

15. .then((response) => response.json())

16. .then((data) => {

17. const tmp: VisitCalendar[] = []

18. if(data) {

19. data.forEach((v: VisitCalendar) => {

20. v.info.start\_date = new Date(v.info.start\_date)

21. v.info.end\_date = new Date(v.info.end\_date)

22. if (v.info.status !== 'declined') {

23. tmp.push(v)

24. }

25. })

26. }

27. setVisits(tmp)

28. })

29. .catch(err => {

30. console.log("Error retrieving Visits: ", err)

31. })

32. }

Fragment kodu 41. Implementacja funkcji odpowiedzianej za pobranie rezerwacji specjalisty z serwera [źródło własne]

W powyższym kodzie, adres *URL*, pod który wysyłane jest zapytanie *GET*, uzupełniany jest o identyfikator specjalisty. Adres ten po stronie serwera obsługiwany jest przez metodę *GetCalendarVisitsBySpecialistIdOrClientId*, której implementację ukazano poniżej (fragment kodu 42):

1. func (app \*Application) GetCalendarVisitsBySpecialistIdOrClientId(w http.ResponseWriter,

r \*http.Request) {

2. var specId \*int

3. var clId \*int

4.

5. specialistId, err := strconv.Atoi(chi.URLParam(r, "specialist\_id"))

6. if err != nil {

7. fmt.Println("cannot find parameter specialist\_id: ", err)

8. \_ = app.errorJSON(w, err)

9. return

10. }

11. if specialistId == 0 {

12. specId = nil

13. } else {

14. specId = &specialistId

15. }

16.

17. clientId, err := strconv.Atoi(chi.URLParam(r, "client\_id"))

18. if err != nil {

19. fmt.Println("cannot find parameter client\_id: ", err)

20. \_ = app.errorJSON(w, err)

21. return

22. }

23. if clientId == 0 {

24. clId = nil

25. } else {

26. clId = &clientId

27. }

28.

29. visits, err := app.DB.GetCalendarVisitsBySpecialistIdOrClientId(specId, clId)

30. if err != nil {

31. fmt.Println("error getting Calendar Visits from db: ", err)

32. \_ = app.errorJSON(w, err)

33. return

34. }

35. \_ = app.writeJSON(w, http.StatusOK, visits)

36. }

Fragment kodu 42. Implementacja metody obsługującej żądanie pobrania rezerwacji [źródło własne]

W powyższym kodzie, na początku odczytywane są wartości *id* specjalisty oraz *id* klienta z adresu *URL* (linia 5 i linia 17), które następnie są przekazywane do funkcji *GetCalendarVisitsBySpecialistIdOrClientId*. Ponieważ metoda ta służy do pobrania z bazy rezerwacji przypisanych do specjalisty o podanym *id* oraz klienta o podanym *id*, a w tym przypadku chcemy pobrać wszystkie rezerwacje dla konkretnego specjalisty bez względu na *id* klienta, jako drugi argument podawana jest wartość *nil*. Kod przytoczonej funkcji przedstawiono poniżej (fragment kodu 43):

1. func (m \*PG) GetCalendarVisitsBySpecialistIdOrClientId(specialistId \*int, clientId \*int)

([]models.VisitCalendar, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.GetCalendarVisitsBySpecialistIdOrClientId

6. var visits []models.VisitCalendar

7. rows, err := m.DB.QueryContext(ctx, q, specialistId, clientId)

8. if err != nil {

9. return nil, fmt.Errorf("error retrieving data: %w", err)

10. }

11. defer func() {

12. \_ = rows.Close()

13. }()

14.

15. for rows.Next() {

16. var v models.VisitCalendar

17. err := rows.Scan(

18. &v.Info.Id,

19. &v.Info.StartDate,

20. &v.Info.EndDate,

21. ...)

22. if err != nil {

23. return nil, fmt.Errorf("error scanning row: %w", err)

24. }

25. v.Info.StartDate = v.Info.StartDate.Add(-2 \* time.Hour)

26. v.Info.EndDate = v.Info.EndDate.Add(-2 \* time.Hour)

27. visits = append(visits, v)

28. }

29. return visits, nil

30. }

Fragment kodu 43. Implementacja metody pobierającej rezerwacje z bazy danych [źródło własne]

Ukazana wyżej metoda wykorzystuje odpowiednie zapytanie *SQL*, które wywołane wraz z danymi podanymi jako argumenty, pobiera dane z bazy danych, a następnie konwertuje je do listy obiektów typu *VisitCalendar* (linie 15-28). Definicję tego typu przedstawia poniższy kod (fragment kodu 44):

1. type VisitCalendar struct {

2. Info Visit `json:"info"`

3. Specialist SpecialistProfileInfo `json:"specialist"`

4. Client Client `json:"client"`

5. Service SpecialistService `json:"service"`

6. }

Fragment kodu 44. Typ danych reprezentujący rezerwacje w systemie [źródło własne]

Powyższy typ posiada w sobie pole *Info* typu *Visit*, które zawiera informacje o rezerwacji (czas rozpoczęcia i zakończenia, cena, opis, adres realizacji), także pole *Specialist* z informacjami na temat specjalisty, pole *Client* z informacjami o przypisanym do rezerwacji kliencie, oraz pole *Service* z informacjami o usłudze jaka ma być realizowana w ramach usługi.

Otrzymana lista rezerwacji jest zwracana przez metodę *GetCalendarVisitsBySpecialistIdOrClientId* (fragment kodu 42, linia 29), a następnie zwracana jest jako odpowiedź na zapytanie wysłane przez aplikację kliencką (fragment kodu 42, linia 35).

Otrzymane dane z serwera przetwarzane są do formatu *JSON* (fragment kodu 41, linia 15), a następnie zapisywane są do zmiennej *visits* za pomocą funkcji *setVisits* (fragment kodu 41). Zmienna ta podawana jest w kalendarzu, dzięki czemu rezerwacje zostają w nim wyświetlone.

Aby rezerwacja usługi została wykonana przez klienta, musi on kliknąć w interesujący go slot czasu w kalendarzu. Wywoływana zostaje wtedy funkcja *handleSelectSlot*, której implementacje przedstawiono poniżej (fragment kodu 45):

1. const [newVisit, setNewVisit] = useState<VisitRequest>({

2. start\_date: new Date(),

3. end\_date: new Date(),

4. description: "",

5. client\_id: 0,

6. client\_address\_id: 0,

7. specialist\_id: 0,

8. specialist\_service\_id: 0,

9. })

10.

11. const handleSelectSlot = ({start}: { start: Date }) => {

12. if (jwtToken === "") {

13. ...

14. }

15. if (!isDateFromFuture(start)) {

16. ...

17. }

18. if (!isDateAvailable(start)) {

19. ...

20. }

21. if (isDateOverlapping(start)) {

22. ...

23. }

24.

25. const tmp = start.getTime();

26. setNewVisit({

27. ...newVisit,

28. specialist\_id: specialist?.specialist.id!,

29. start\_date: start,

30. end\_date: new Date(tmp + 2 \* 60 \* 60 \* 1000)

31. });

32. Swal.fire({

33. didOpen: () => setShowRegisterWindow(true),

34. didClose: () => {

35. setShowRegisterWindow(false)

36. setNewVisit({

37. ...newVisit,

38. specialist\_service\_id: 0,

39. client\_address\_id: 0,

40. description: ""

41. })

42. ...

43. },

44. showConfirmButton: false,

45. })

46. };

Fragment kodu 45. Implementacja funkcji obsługującej zdarzenie kliknięcia w slot czasu w kalendarzu [źródło własne]

W powyższym kodzie przekazywany jest argument *start*, reprezentujący slot czasu wybrany przez klienta. Na początku funkcji sprawdzane jest czy użytkownik jest zalogowany (linia 12), następnie dokonywana jest walidacja wybranego slotu czasu: funkcja *isDateFromFuture* (linia 15) sprawdza czy klient nie zaznaczył daty z przeszłości, funkcja *isDateAvailable* (linia 18) sprawdza czy wybrany slot czasu nie nakłada się na urlop specjalisty, natomiast funkcja *isDateOverlapping* (linia 21) waliduje czy data nie pokrywa się z istniejącą rezerwacją. Jeśli walidacja zakończy się błędem, okno z informacją o błędzie zostanie wyświetlone na ekranie. W przeciwnym razie, gdy walidacja przebiegnie pomyślnie, inicjowany jest obiekt *newVisit* za pomocą funkcji *setNewVisit* (linie 26-31), oraz wyświetlane jest okno z formularzem umożliwiającym stworzenie rezerwacji. W nim, za pomocą komponentu *<DateTimePicker>* wyświetlane jest pole z wyborem daty rezerwacji (fragment kodu 46):

1. <DateTimePicker

2. defaultValue={dayjs(newVisit.start\_date)}

3. ampm={false}

4. minutesStep={15}

5. minTime={dayjs().set('hour', 6).set('minute', 0)}

6. maxTime={dayjs().set('hour', 21).set('minute', 0)}

7. onChange={(value) => {

8. setNewVisit({

9. ...newVisit,

10. start\_date: value!.toDate()

11. })

12. ...

13. }}

14. />

Fragment kodu 46. Implementacja pola umożliwiającego wybór daty rezerwacji [źródło własne]

Początkowo, po wyświetleniu formularza, pole to wyświetla datę, którą klient kliknął w kalendarzu. Jednak po kliknięciu w to pole, klient może wybrać inny dzień oraz godzinę rezerwacji – nowa data zostanie wpisana do zmiennej *start\_date* w obiekcie *newVisit* przez wywołanie przypisanej do pola funkcji *onChange* (linie 7-15).

Kolejnym polem w formularzu jest pole umożliwiające wybór usługi, która ma być realizowana w ramach danej rezerwacji. Jego implementację ukazano poniżej (fragment kodu 47):

1. <select

2. id="service\_id"

3. name="usługa"

4. value={newVisit.specialist\_service\_id}

5. onChange={(value) => {

6. setNewVisit({

7. ...newVisit,

8. specialist\_service\_id: parseInt(value.currentTarget.value)

9. })

10. ...

11. }}

12. className={`w-full h-14 border-2 text-lg border-gray-300 rounded-md pl-2`}

13. >

14. <option value="" disabled={newVisit.specialist\_service\_id !== 0}>

15. Nazwa usługi

16. </option>

17. {specialist?.services!.map((option) => {

18. return (

19. <option

20. key={option.id}

21. value={option.id}

22. >

23. {option.name}

24. </option>

25. )

26. })}

27. </select>

Fragment kodu 47. Implementacja pola wyboru usługi w ramach rezerwacji [źródło własne]

Powyższy kod umożliwia wyświetlenie listy usług realizowanych przez specjalistę (linie 14-26), które znajdują się w zmiennej *services* w obiekcie *specialist*, zawierającego informacje na temat danego specjalisty, które pobierane są po wejściu klienta na stronę specjalisty. Po wybraniu przez klienta jednej z usług, jej *id* przypisywane jest do zmiennej *specialist\_service\_id* w obiekcie *newVisit* za pomocą funkcji *onChange* (linie 5-11).

Informacją, którą musi również podać klient przy tworzeniu nowej rezerwacji, jest adres realizacji usługi. Jego wybór możliwy jest z rozsuwanej listy, która renderowana jest na stronie za pomocą poniższego kodu (fragment kodu 48):

1. <select

2. id="client\_address\_id"

3. name="adres"

4. value={newVisit.client\_address\_id}

5. onChange={(value) => {

6. setNewVisit({

7. ...newVisit,

8. client\_address\_id: parseInt(value.currentTarget.value)

9. })

10. ...

11. }}

12. className={`w-full h-14 border-2 text-lg border-gray-300 rounded-md pl-2`}

13. >

14. <option value="" disabled={newVisit.client\_address\_id !== 0}>

15. Adres

16. </option>

17. {clientAddresses!.map((address) => {

18. return (

19. <>

20. {address.flat\_nr === 0 ?

21. <option key={address.id} value={address.id}>

22. {address.city.name}, ul. {address.street} {address.building\_nr}

23. </option>

24. :

25. <option key={address.id} value={address.id}>

26. {address.city.name}, ul. {address.street

{address.building\_nr}/{address.flat\_nr}

27. </option>

28. }

29. </>

30. )

31. })}

32. </select>

Fragment kodu 48. Implementacja pola wyboru adresu realizacji rezerwacji [źródło własne]

Powyższy kod wyświetla listę adresów przypisanych do konta klienta: miasto, nazwę ulicy, oraz numer budynku i numer mieszkania. Po wybraniu przez klienta jednego z adresów, jego numer *id* przypisywany jest do zmiennej *client\_address\_id* w obiekcie *newVisit* za pomocą funkcji *onChange* (linie 5-11). Wspomniana lista adresów, znajduje się w zmiennej *clientAddresses*, która z kolei jest wypełniana przy wejściu klienta na stronę, za pomocą poniższego kodu (fragment kodu 49):

1. useEffect(() => {

2. ...

3. const user = sessionStorage.getItem(userRole)

4. if (user) {

5. var c = JSON.parse(user)

6. setNewVisit({

7. ...newVisit,

8. client\_id: c.id

9. })

10. headers.append("Authorization", "Bearer " + jwtToken)

11. fetch(`/client/addresses/${c.id}`, {

12. method: "GET",

13. headers: headers,

14. credentials: 'include'

15. }).then((response) => response.json())

16. .then((data) => {

17. setClientAddresses(data)

18. })

19. }

20. }, [jwtToken, location.state.specialistId, navigate, userRole])

21.

Fragment kodu 49. Implementacja funkcji odpowiedzianej za pobranie adresów klienta z serwera [źródło własne]

W powyższym kodzie, pobierane są dane na temat użytkownika, zapisane w sesji przeglądarki (linia 3). Następnie wysyłane jest zapytanie *GET* pod adres *URL* „/client/addresses/” wraz z numerem *id* klienta dodanym na koniec adresu. Ponieważ dostęp do danych pod tym adresem mają tylko zalogowani klienci, do zapytania dołączany jest również nagłówek zawierający token *jwtToken* (linia 10). Po stronie serwera, dany adres obsługiwany jest przez funkcję *GetClientAddressesByClientId*, której implementację ukazano poniżej (fragment kodu 50):

1. func (app \*Application) GetClientAddressesByClientId(w http.ResponseWriter, r \*http.Request) {

2. clientId, err := strconv.Atoi(chi.URLParam(r, "client\_id"))

3. if err != nil {

4. \_ = app.errorJSON(w, err)

5. return

6. }

7.

8. clientAddresses, err := app.DB.GetClientAddressesByClientId(clientId)

9. if err != nil {

10. \_ = app.errorJSON(w, err)

11. return

12. }

13. \_ = app.writeJSON(w, http.StatusOK, clientAddresses)

14. }

Fragment kodu 50. Implementacja metody obsługującej żądanie pobrania adresów klienta [źródło własne]

W powyższej funkcji, z adresu *URL* odczytywany jest numer *id* klienta (linia 2), a następnie wstawiany jako argument przy wywołaniu funkcji *GetClientAddressesByClientId* (linia 8). Ta z kolei używa tego numeru przy wywołaniu odpowiedniego zapytania *SQL,* które pobiera listę adresów klienta o podanym *id*. Następnie otrzymane dane są mapowane do odpowiedniego typu i zwracane w postaci listy. Całość tego procesu realizowana jest przez poniższy kod (fragment kodu 51):

1. func (m \*PG) GetClientAddressesByClientId(clientId int) ([]models.ClientAddress, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.GetClientAddressesByClientId

6. var clientAddresses []models.ClientAddress

7. rows, err := m.DB.QueryContext(ctx, q, clientId)

8. if err != nil {

9. return nil, fmt.Errorf("error retrieving data: %w", err)

10. }

11. defer func() {

12. \_ = rows.Close()

13. }()

14.

15. for rows.Next() {

16. var ca models.ClientAddress

17. err := rows.Scan(

18. &ca.Id,

19. &ca.Street,

20. &ca.BuildingNr,

21. &ca.FlatNr,

22. &ca.City.Id,

23. &ca.City.Name)

24. if err != nil {

25. return nil, fmt.Errorf("error scanning row: %w", err)

26. }

27. clientAddresses = append(clientAddresses, ca)

28. }

29. return clientAddresses, nil

30. }

Fragment kodu 51. Implementacja metody pobierającej adresy klienta z bazy danych [źródło własne

Ostatnim polem formularza, który wypełnia klient przy tworzeniu rezerwacji, jest opis. Jego wprowadzenie umożliwia element *<textarea>*, którego implementację ukazano poniżej (fragment kodu 52):

1. <textarea

2. id="description"

3. name="description"

4. placeholder="Opis usługi"

5. value={newVisit.description}

6. onChange={(value) => {

7. setNewVisit({

8. ...newVisit,

9. description: value.currentTarget.value

10. })

11. ...

12. }}

13. rows={6}

14. cols={40}

15. className={`w-full border-2 text-lg border-gray-300 rounded-md p-3`}

16. />

Fragment kodu 52. Implementacja pola umożliwiającego wprowadzenie opisu rezerwacji [źródło własne]

W przedstawionym kodzie, w momencie wprowadzenia przez klienta opisu, uruchamiana jest funkcja *onChange* (linie 6-12), która do zmiennej *description* w obiekcie *newVisit* zapisuje wpisany tekst.

Po wypełnieniu wszystkich wymaganych pól przez klienta, klika on w przycisk „Zarezerwuj”, po czym uruchamiana jest funkcja *createVisit*, która przedstawiona jest poniżej (fragment kodu 53):

1. const createVisit = () => {

2. if (!checkNewVisitErrors()) {

3. return

4. }

5.

6. const headers = new Headers()

7. headers.append("Content-Type", "application/json")

8. headers.append("Authorization", "Bearer " + jwtToken)

9. const method = "POST"

10. fetch(`/client/visit/create`, {

11. body: JSON.stringify(newVisit),

12. method: method,

13. headers: headers,

14. credentials: 'include'

15. })

16. .then((response) => response.json())

17. .then((data) => {

18. if (data.error) {

19. console.log("ERRORS")

20. setUniversalError(data.message)

21. } else {

22. console.log("SUCCESSFULLY CREATED VISIT")

23. setSuccessMessage("Pomyślnie utworzono rezerwację")

24. getSpecialistVisits(specialist?.specialist.id!)

25. }

26. })

27. .catch((err) => {

28. console.log(err)

29. })

30. }

Fragment kodu 53. Implementacja funkcji wysyłającej żądanie utworzenia nowej rezerwacji [źródło własne]

W powyższej funkcji, na początku wykonywana jest walidacja uzupełnionego za pomocą formularza obiektu *newVisit*, przez funkcję *checkNewVisitErrors* (linia 2). Sprawdza ona czy wybrana data nie jest datą przeszłą, nie nakłada się na urlop specjalisty lub na żadną istniejącą już rezerwację. Sprawdzane jest również, czy pozostałe pola formularza zostały wypełnione. Jeśli w czasie walidacji znaleziona zostanie jakaś nieprawidłowość, system wyświetli odpowiedni tekst z informacją o błędzie. W przeciwnym wypadku, wysłane zostanie żądanie *POST* na adres *URL* „/client/visit/create”, wraz z przekonwertowanym do formatu *JSON* obiektem *newVisit*. Serwer odbiera żądanie i obsługuje je przy pomocy funkcji CreateVisit, której implementację przedstawiono poniżej (fragment kodu 54):

1. func (app \*Application) CreateVisit(w http.ResponseWriter, r \*http.Request) {

2. var newVisitRequest models.VisitRequest

3. err := app.readJSON(w, r, &newVisitRequest)

4. if err != nil {

5. \_ = app.errorJSON(w, err)

6. return

7. }

8.

9. visits, err := app.DB.GetCalendarVisitsBySpecialistIdOrClientId(&newVisitRequest.SpecialistId, nil)

10. if err != nil {

11. \_ = app.errorJSON(w, err)

12. return

13. }

14. for \_, v := range visits {

15. if v.Info.Status == "declined" {

16. continue

17. }

18. if isVisitOverlapping(newVisitRequest.StartDate, newVisitRequest.EndDate,

v.Info.StartDate, v.Info.EndDate) {

19. \_ = app.errorJSON(w, fmt.Errorf("Wybrany termin rozpoczęcia lub

zakończenia nakłada się na istniejącą już wizytę"))

20. return

21. }

22. }

23.

24. timeOffs, err := app.DB.GetTimeOffBySpecialistId(newVisitRequest.SpecialistId)

25. for \_, t := range timeOffs {

26. if isVisitOverlapping(newVisitRequest.StartDate, newVisitRequest.EndDate,

t.StartDate, t.EndDate) {

27. \_ = app.errorJSON(w, fmt.Errorf("Wybrany termin nakłada się na

termin urlopu specjalisty"))

28. return

29. }

30. }

31.

32. visitId, err := app.DB.CreateVisit(newVisitRequest)

33. if err != nil {

34. \_ = app.errorJSON(w, err)

35. return

36. }

37.

38. notification := models.NotificationRequest{

39. Type: "created",

40. Notifier: "client",

41. ClientId: newVisitRequest.ClientId,

42. SpecialistId: newVisitRequest.SpecialistId,

43. VisitId: visitId,

44. }

45. \_, err = app.DB.CreateNotification(notification)

46. if err != nil {

47. log.Println(err)

48. \_ = app.errorJSON(w, err)

49. return

50. }

51. \_ = app.writeJSON(w, http.StatusOK, visitId)

52. }

Fragment kodu 54. Implementacja funkcji wysyłającej żądanie utworzenia nowej rezerwacji [źródło własne]

Na początku metody, odczytywane są przesłane w żądaniu dane i mapowane do zmiennej *newVisitRequest* (linia 3). Następnie wywoływana jest metoda *GetCalendarVisitsBySpecialistIdOrClientId* (linia 9), dla zapisanego w *newVisitRequest* *id* specjalisty, która zwraca listę jego rezerwacji. Przy jej użyciu, w pętli dochodzi do sprawdzenia za pomocą funkcji *isVisitOverlapping*, czy nowa rezerwacja nie nakłada się na już istniejące w systemie rezerwacje (linie 14-22). Następnie wywoływana jest metoda *GetTimeOffBySpecialistId* (linia 24), której implementacja została ukazana poniżej (fragment kodu 55):

1. func (m \*PG) GetTimeOffBySpecialistId(specialistId int) ([]models.TimeOff, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.GetTimeOffBySpecialistId

6. var timeOffs []models.TimeOff

7. rows, err := m.DB.QueryContext(ctx, q, specialistId)

8. if err != nil {

9. return nil, fmt.Errorf("error retrieving data: %w", err)

10. }

11. defer func() {

12. \_ = rows.Close()

13. }()

14.

15. for rows.Next() {

16. var t models.TimeOff

17. err := rows.Scan(

18. &t.Id,

19. &t.StartDate,

20. &t.EndDate,

21. &t.SpecialistId)

22. if err != nil {

23. return nil, fmt.Errorf("error scanning row: %w", err)

24. }

25. t.StartDate = t.StartDate.Add(-2 \* time.Hour)

26. t.EndDate = t.EndDate.Add(-2 \* time.Hour)

27. timeOffs = append(timeOffs, t)

28. }

29. return timeOffs, nil

30. }

Fragment kodu 55. Implementacja funkcji pobierającej listę urlopów danego specjalisty [źródło własne]

W powyższej funkcji wykonywane jest odpowiednie zapytanie *SQL* wraz z *id* specjalisty podanym jako argument metody (linia 7). Otrzymane w wyniku tej operacji dane mapowane są na obiekty typu *TimeOff* (linie 17-21) oraz dodawane do zmiennej *timeOffs*, która zwracana jest na końcu metody. Tak otrzymana lista wykorzystywana jest do sprawdzenia, przy pomocy funkcji *isVisitOverlapping*, czy nowa rezerwacja nie nakłada się na istniejący w systemie urlop (fragment kodu 54, linie 25-30).

Po pomyślnej walidacji wywoływana jest metoda *CreateVisit*, która jako argument przyjmuje obiekt *newVisitRequest* zawierający informacje o nowej rezerwacji. Jej implementację ukazano poniżej (fragment kodu 56):

1. func (m \*PG) CreateVisit(visitRequest models.VisitRequest) (int, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.CreateVisit

6. var newVisitId int

7. err := m.DB.QueryRowContext(ctx, q,

8. visitRequest.StartDate.Add(2\*time.Hour),

9. visitRequest.EndDate.Add(2\*time.Hour),

10. visitRequest.Description,

11. visitRequest.ClientAddressId,

12. visitRequest.ClientId,

13. visitRequest.SpecialistId,

14. visitRequest.SpecialistServiceId,

15. ).Scan(&newVisitId)

16. if err != nil {

17. return 0, fmt.Errorf("cannot create new Visit: %w", err)

18. }

19. return newVisitId, nil

20. }

21.

Fragment kodu 56. Implementacja funkcji tworzącej nową rezerwację w bazie danych [źródło własne]

W powyższym kodzie, przy użyciu odpowiedniego zapytania *SQL* oraz wartości zapisanych w obiekcie *visitRequest* (linie 5-15), tworzona jest nowa rezerwacja w bazie danych.

Po poprawnym utworzeniu rezerwacji w bazie danych, tworzony jest obiekt *notification* typu *NotificationRequest* (fragment kodu 54, linie 38-44), do którego wpisywane jest *id* klienta, *id* specjalisty, *id* utworzonej przez poprzednią metodę rezerwacji, typ powiadomienia jako „created”, oraz typ powiadamiającego jako „client”. Tak utworzony obiekt przekazywany jest jako argument przy wywołaniu metody *CreateNotification* (linia 45). Jej implementację ukazano poniżej (fragment kodu 57):

1. func (m \*PG) CreateNotification(notification models.NotificationRequest) (int, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. var newNotificationId int

6. q := sql.CreateNotification

7. err := m.DB.QueryRowContext(ctx, q,

8. notification.Type,

9. notification.Notifier,

10. false,

11. notification.ClientId,

12. notification.SpecialistId,

13. notification.VisitId,

14. time.Now().Add(2\*time.Hour),

15. ).Scan(&newNotificationId)

16. if err != nil {

17. return 0, fmt.Errorf("cannot create new Review: %w", err)

18. }

19. return newNotificationId, nil

20. }

Fragment kodu 57. Implementacja funkcji tworzącej powiadomienie w bazie danych [źródło własne]

Powyższy kod, na podstawie otrzymanego obiektu *notification* oraz przy użyciu odpowiedniego zapytania *SQL*, tworzy w bazie danych nowe powiadomienie.

Po wykonaniu tej metody, metoda *CreateVisit* (fragment kodu 54) jako odpowiedź na wysłane przez aplikację kliencką żądanie, zwraca *id* nowo utworzonej rezerwacji oraz kod informujący o pomyślnym przebiegu operacji. Klient odbiera te dane (fragment kodu 53), wyświetla na ekranie tekst o pomyślnym utworzeniu rezerwacji przez klienta, oraz wywołuje funkcję *getSpecialistVisits* (fragment kodu 41), aktualizując przy tym kalendarz specjalisty w profilu.

### Wyświetlanie kalendarza rezerwacji klienta, ich modyfikacja oraz dodawanie opinii

Pod adresem „klient/rezerwacje” umieszczona została strona służąca do wyświetlania kalendarza z rezerwacjami danego klienta. Na stronie tej, oprócz instrukcji oraz legendy, znajduje się kalendarz, osadzony za pomocą komponentu *<CalendarForVisits>*, którego bazę stanowi *react-big-calendar* (którego elementy opisano w poprzednim podrozdziale). Użycie w kodzie tego komponentu przedstawiono poniżej (fragment kodu 58):

1. <CalendarForVisits

2. events={visits}

3. handleSelectSlot={() => null}

4. style={{

5. height: 1500,

6. fontSize: "x-large"

7. }}

8. eventBackgroundAdjustment={eventBackgroundAdjustment}

9. selectable={false}

10. views={[Views.DAY, Views.WEEK, Views.MONTH]}

11. EventComponent={EventComponent}

12. onSelectEvent={(event) => selectVisit(event)}

13. timeOffSlotAdjustment={() => {

14. return {}

15. }}

16. />

Fragment kodu 58. Użycie komponentu *CalendarForVisits* do wyświetlania rezerwacji klienta [źródło własne]

W powyższym kodzie, do pola *events* przekazywana jest zmienna *visits*, przechowująca listę rezerwacji klienta. Zmienna *handleSelectSlot* obsługuje zachowanie kalendarza w przypadku kliknięcia na nim w slot czasowy – w tym przypadku nie jest podejmowana żadna akcja. *Style* określa niestandardowy styl dla kalendarza, z kolei do pola *eventBackgroundAdjustment* przypisywana jest funkcja dostosowująca wygląd rezerwacji w kalendarzu. *Selectable*, ustawione na *false*, uniemożliwia tworzenie w kalendarzu nowych rezerwacji. *Views* określa jakie widoki kalendarza są dostępne – w tym przypadku widok dzienny, tygodniowy i miesięczny. Do pola *EventComponent* przypisywana jest funkcja zwracająca komponent odpowiedzialny za wyświetlenie pojedynczej rezerwacji w kalendarzu. Pole *onSelectEvent* reprezentuje zachowanie systemu w przypadku kliknięcia w rezerwację w kalendarzu – taka rezerwacja przetwarzana jest za pomocą funkcji *selectVisit*. Ostatnim polem komponentu jest *timeOffSlotAdjustment*, które obsługuje wyświetlanie urlopów, stąd w tym przypadku nie przypisywana jest mu żadna konkretna funkcja.

Aby rezerwacje klienta zostały wyświetlone w kalendarzu, muszą być wpierw pobrane z serwera. Odbywa się to w momencie wejścia klienta na stronę, za pomocą poniższego kodu (fragment kodu 59):

1. useEffect(() => {

2. if (client) {

3. const headers = new Headers()

4. headers.append("Content-Type", "application/json")

5. const requestOptions = {

6. method: "GET",

7. headers: headers,

8. }

9. fetch(`http://localhost:8080/visits/0/${client.id}`, requestOptions)

10. .then((response) => response.json())

11. .then((data) => {

12. data.forEach((v: VisitCalendar) => {

13. v.info.start\_date = new Date(v.info.start\_date)

14. v.info.end\_date = new Date(v.info.end\_date)

15. })

16. setVisits(data)

17. })

18. .catch(err => {

19. console.log("Error retrieving Visits: ", err)

20. })

21. ...

22. }

23. }, [client, jwtToken]);

Fragment kodu 59. Implementacja kodu wysyłającego żądanie pobrania listy rezerwacji klienta [źródło własne]

W powyższym kodzie, wysyłane jest zapytanie *GET* pod adres „/visits”, wraz z *id* klienta (znajdującym się w zmiennej *client*, pobranej z sesji przeglądarki). Adres ten, po stronie serwera obsługiwany jest przez metodę *GetCalendarVisitsBySpecialistIdOrClientId* (fragment kodu 42), której działanie opisano w poprzednim podrozdziale. Otrzymane w wyniku zapytania dane, mapowane są na typ *VistCalendar* i dodawane do zmiennej *visits* za pomocą funkcji *setVisits* (linie 11-16).

W momencie kliknięcia przez klienta jednej z wizyt w kalendarzu, wywołana zostaje funkcja selectVisit, której implementację przedstawiono poniżej (fragment kodu 60):

1. const selectVisit = (visit: VisitCalendar) => {

2. setBaseVisit(visit)

3. setSelectedVisit(visit)

4. setNotification({

5. ...notification,

6. client\_id: visit.client.id,

7. specialist\_id: visit.specialist.id,

8. visit\_id: visit.info.id,

9. })

10. setNewReview({

11. ...newReview,

12. specialist\_id: visit.specialist.id,

13. client\_id: visit.client.id,

14. visit\_id: visit.info.id,

15. specialist\_service\_id: visit.service.id,

16. })

17. if (visit.info.start\_date < new Date()) {

18. getReviewForVisit(visit.info.id)

19. ...

20. }

21. if (visit.info.status === 'declined') {

22. ...

23. } else if (visit.info.status === 'client\_action\_required') {

24. setOpenCalendar(true)

25. ...

26. fetch(`http://localhost:8080/time\_off/${visit.specialist.id}`, requestOptions)

27. .then((response) => response.json())

28. .then((data) => {

29. setSelectedSpecialistTimeOffs(data)

30. })

31. .catch(err => {

32. console.log("Error retrieving TimeOff: ", err)

33. })

34. fetch(`http://localhost:8080/visits/${visit.specialist.id}/0`, requestOptions)

35. .then((response) => response.json())

36. .then((data) => {

37. const tmp: VisitCalendar[] = []

38. data.forEach((v: VisitCalendar) => {

39. v.info.start\_date = new Date(v.info.start\_date)

40. v.info.end\_date = new Date(v.info.end\_date)

41. if (v.info.status !== 'declined') {

42. tmp.push(v)

43. }

44. })

45. setSelectedSpecialistVisits(tmp)

46. })

47. .catch(err => {

48. console.log("Error retrieving Visits: ", err)

49. })

50. ...

51. } else if (visit.info.status === 'specialist\_action\_required') {

52. ...

53. } else if (visit.info.status === 'accepted') {

54. ...

55. }

56. }

Fragment kodu 60. Implementacja funkcji wywoływanej przy kliknięciu w rezerwacje [źródło własne]

W powyższej funkcji, na początku za pomocą przekazanego obiektu reprezentującego klikniętą rezerwację, ustawiane są zmienne *baseVisit* (linia 2), *selectedVisit* (linia 3), *notification* (linie 4-9), oraz *newReview* (linie 10-16). Następnie dochodzi do sprawdzenia, czy wybrana rezerwacja jest rezerwacją przeszłą – jeśli tak, wywoływana jest funkcja *getReviewForVisit*, której implementację ukazano poniżej (fragment kodu 61):

1. const getReviewForVisit = (visitId: number) => {

2. const headers = new Headers()

3. headers.append("Content-Type", "application/json")

4. const requestOptions = {

5. method: "GET",

6. headers: headers,

7. }

8. fetch(`http://localhost:8080/reviews/${visitId}`, requestOptions)

9. .then((response) => response.json())

10. .then((data) => {

11. if (data === null) {

12. setReviewNotExists(true)

13. return

14. }

15. })

16. .catch(err => {

17. console.log("Error retrieving Review: ", err)

18. })

19. }

Fragment kodu 61. Implementacja funkcji wysyłającej żądanie pobrania opinii rezerwacji [źródło własne]

W powyższym kodzie wysyłane jest zapytanie *GET* pod adres „/reviews” wraz z podaniem *id* rezerwacji. Serwer odbiera je i obsługuje metodą *GetReviewByVisitId* (fragment kodu 62):

1. func (app \*Application) GetReviewByVisitId(w http.ResponseWriter, r \*http.Request) {

2. visitId, err := strconv.Atoi(chi.URLParam(r, "visit\_id"))

3. if err != nil {

4. fmt.Println("cannot find parameter visit\_id: ", err)

5. \_ = app.errorJSON(w, err)

6. return

7. }

8. review, err := app.DB.GetReviewByVisitId(visitId)

9. if err != nil {

10. fmt.Println("error getting Review from db for visitId=", visitId, " : ", err)

11. \_ = app.errorJSON(w, err)

12. return

13. }

14. app.writeJSON(w, http.StatusOK, review)

15. }

16.

Fragment kodu 62. Implementacja metody obsługującej żądanie pobrania opinii rezerwacji [źródło własne]

Powyższa metoda pobiera wartość *visit\_id* z adresu *URL* (linia 2) i przy jej pomocy wywołuje metodę *GetReviewByVisitId* (linia 8), która poprzez odpowiednie zapytanie *SQL* pobiera z bazy opinię przypisaną do rezerwacji o podanym *id*. Implementację tej metody ukazano poniżej (fragment kodu 63):

1. func (m \*PG) GetReviewByVisitId(visitId int) (\*models.Review, error) {

2. q := sql.GetReviewByVisitId

3. var review models.Review

4. err := m.DB.QueryRow(q, visitId).Scan(&review.Id,

5. &review.Rating,

6. &review.Client.Id,

7. ...)

8. if errors.Is(err, sql2.ErrNoRows) {

9. return nil, nil

10. } else if err != nil {

11. return nil, fmt.Errorf("error running query: %w", err)

12. }

13. return &review, nil

14. }

Fragment kodu 63. Implementacja metody pobierającej opinię na temat rezerwacji z bazy danych [źródło własne]

Pobrana z bazy wartość zostaje następnie zwrócona jako odpowiedź do klienta, po czym na ekranie wyświetlane jest okno z informacjami na temat wybranej przez klienta rezerwacji: dane specjalisty, termin wykonania rezerwacji, rodzaj usługi, cena, oraz opis (fragment kodu 64):

1. <div

2. ...

3. <Link to="/specjalista/szczegóły" ...>

4. ...

5. <p className="...">{selectedVisit?.specialist.name}

{selectedVisit?.specialist.second\_name}</p>

6. <div className="...">

7. ...

8. <p className="...">{selectedVisit?.specialist.specialization}</p>

9. ...

10. <p className="...">{selectedVisit?.specialist.city}</p>

11. ...

12. <p className="...">{selectedVisit?.specialist.phone\_nr}</p>

13. ...

14. <p className="...">{selectedVisit?.specialist.email}</p>

15. ...

16. </Link>

17. </div>

18. {isVisitOld && selectedVisit?.info.status === 'accepted' ?

19. <>

20. ...

21. <p className="...">Data:</p>

22. <p className="…"> {selectedVisit.info.start\_date.getHours()} :

{selectedVisit.info.start\_date.getMinutes() === 0 ? '00' :

selectedVisit.info.end\_date.getMinutes()} - {selectedVisit.info.end\_date.getHours()} :

{selectedVisit.info.end\_date.getMinutes() === 0 ? '00' : selectedVisit.info.end\_date.getMinutes()}({selectedVisit.info.end\_date.toLocaleDateString()})</p>

23. ...

24. <p className="...">Usługa:</p>

25. <p className="...">{selectedVisit.service.name}</p>

26. ...

27. <p className="...">Cena:</p>

28. <p className="...">{selectedVisit.info.price} zł</p>

29. ...

30. <p className="...">Opis:</p>

31. <p className="...">{selectedVisit.info.description}</p>

32. ...

33. </>

34. }

Fragment kodu 63. Implementacja kodu wyświetlającego informacje o wizycie [źródło własne]

Jeśli w wyniku zapytania została zwrócona wartość *nil* (opinia dla danej wizyty nie istnieje), pod informacjami na temat rezerwacji pojawia się sekcja z formularzem umożliwiającym wystawienie opinii. Kod który za to odpowiada ukazano poniżej (fragment kodu 64):

1. {reviewNotExists &&

2. ...

3. <p className="font-bold text-3xl pb-1">Wystaw opinię</p>

4. <div ...>

5. ...

6. <p className="w-1/3 font-bold text-right mr-5">Ocena:</p>

7. <div className="flex flex-row mr-3 pl-5 w-2/3">{renderStars()}</div>

8. ...

9. <p className="w-1/3 font-bold text-right mr-5">Komentarz:</p>

10. ...

11. <textarea

12. id="description"

13. name="description"

14. placeholder="Opisz doświadczenia z wizyty"

15. value={newReview.description}

16. rows={6}

17. cols={40}

18. className={`w-full border-2 text-lg border-gray-300 rounded-md mr-3 p-3`}

19. onChange={(value) => {

20. setNewReview({

21. ...newReview,

22. description: value.currentTarget.value

23. })

24. }}

25. />

26. ...

27. <div onClick={() => addReview()} ...>

28. <span className="mx-3 my-2 text-xl">Dodaj opinię</span>

29. </div>

30. ...

31. </div>

32. }

Fragment kodu 64. Implementacja kodu umożliwiającego wystawienie opinii [źródło własne]

W powyższym kodzie, funkcja *renderStars* (linia 7) odpowiada za wyświetlenie skali w formie gwiazdek – po zaznaczeniu ich ilości przez klienta, wartość ta wpisywana jest przez funkcję do zmiennej *rating* w obiekcie *newReview*. Z kolei sekcja *<textarea>* za pomocą funkcji *onChange* (linie 19-24) pobiera od klienta opis rezerwacji i przypisuje do zmiennej *description* w obiekcie *newReview*. Po kliknięciu przez klienta w przycisk „Dodaj opinię”, wywoływana jest funkcja *addReview*, której implementację ukazano poniżej (fragment kodu 65):

1. function addReview() {

2. if (newReview.rating === 0) {

3. ...

4. }

5. ...

6. const method = "POST"

7. fetch(`/client/review/create`, {

8. body: JSON.stringify(newReview),

9. method: method,

10. headers: headers,

11. credentials: 'include'

12. })

13. .then((response) => response.json())

14. .then((data) => {

15. if (data.error) {

16. ...

17. } else {

18. ...

19. }

20. })

21. .catch((err) => {

22. console.log(err)

23. })

24. }

25.

Fragment kodu 65. Implementacja kodu wysyłającego żądanie utworzenia opinii [źródło własne]

Ukazana funkcja wysyła żądanie *POST* pod adres „/review/create” wraz z obiektem *newReview* w formacie *JSON* w ciele żądania. Po stronie backendu, żądanie to obsługiwane jest przez metodę CreateReview (fragment kodu 66):

1. func (app \*Application) CreateReview(w http.ResponseWriter, r \*http.Request) {

2. var newReview models.ReviewRequest

3. err := app.readJSON(w, r, &newReview)

4. if err != nil {

5. \_ = app.errorJSON(w, err)

6. return

7. }

8. newReviewId, err := app.DB.CreateReview(newReview)

9. if err != nil {

10. log.Println(err)

11. \_ = app.errorJSON(w, err)

12. return

13. }

14. \_ = app.writeJSON(w, http.StatusOK, newReviewId)

15. }

Fragment kodu 66. Implementacja metody obsługującej żądanie utworzenia opinii [źródło własne]

Powyższa metoda odczytuje dane z ciała żądania do obiektu *newReview* (linia 3), a następnie podaje go jako argument przy wywołaniu funkcji *CreateReview* (linia 8) (fragment kodu 67):

1. func (m \*PG) CreateReview(review models.ReviewRequest) (int, error) {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. var newReviewId int

6. q := sql.CreateReview

7. err := m.DB.QueryRowContext(ctx, q,

8. review.Rating,

9. review.Description,

10. ...,

11. ).Scan(&newReviewId)

12. if err != nil {

13. return 0, fmt.Errorf("cannot create new Review: %w", err)

14. }

15. return newReviewId, nil

16. }

Fragment kodu 67. Implementacja metody tworzącej opinię w bazie danych [źródło własne]

Przedstawiona metoda za pomocą odpowiedniego zapytania *SQL* i danych zapisanych w obiekcie *review*, tworzy nową opinię w bazie danych.

W przypadku kiedy rezerwacja kliknięta przez klienta jest rezerwacją z datą przyszłą, sprawdzany jest jej status. W przypadku kiedy jest ona oznaczona jako „client\_action\_required”, wykonywane jest zapytanie *GET* w celu pobrania listy urlopów specjalisty przypisanego do danej rezerwacji (fragment kodu 60, linie 26-33), oraz listy wszystkich jego rezerwacji (fragment kodu 60, linie 34-49). Pobrane dane zapisywane są kolejno w zmiennej *selectedSpecialistTimeOffs* oraz *selectedSpecialistVisits*. Następnie na ekranie wyświetlane jest okno, gdzie wyświetlane są dane specjalisty przypisanego do rezerwacji, oraz kalendarz jego rezerwacji utworzony przy użyciu opisanego wcześniej komponentu <CalendarForVisits>, zasilanego danymi zapisanymi w zmiennych *selectedSpecialistTimeOffs* i *selectedSpecialistVisits* (fragment kodu 68):

1. <CalendarForVisits

2. events={selectedSpecialistVisits!}

3. handleSelectSlot={() => {

4. return {}

5. }}

6. style={{

7. height: 800,

8. fontSize: "x-large"

9. }}

10. eventBackgroundAdjustment={specialistEventBackgroundAdjustment}

11. selectable={false}

12. views={[Views.DAY, Views.WEEK, Views.MONTH]}

13. EventComponent={SpecialistEventComponent}

14. onSelectEvent={() => {

15. return {}

16. }}

17. timeOffSlotAdjustment={timeOffSlotAdjustment}

18. />

Fragment kodu 68. Implementacja kodu wyświetlającego kalendarz dostępności specjalisty w oknie rezerwacji [źródło własne]

Oprócz tego wyświetlany jest formularz, umożliwiający klientowi wprowadzenie zmian do danej rezerwacji: data rozpoczęcia i zakończenia, adres realizacji, rodzaj wykonywanej usługi, szacowana cena oraz opis usługi. Pola zawierające daty wyświetlane są za pomocą komponentu opisanego wcześniej (fragment kodu 46) – w tym jednak przypadku funkcja *onChange* zapisuje nowo wybrane daty w polach *info.start\_dat*e i *info.end\_date* w obiekcie *selectedVisit*. Podobnie w przypadku adresu realizacji rezerwacji – adresy klienta, zapisane w zmiennej *clientAddresses*, wyświetlane są za pomocą elementu *<select>* (fragment kodu 48), a wybór jednego z nich wywołuje funkcję *changeVisitAddress* (fragment kodu 69), która wpisuje nowy adres do obiektu *selectedVisit*:

1. const changeVisitAddress = (event: React.FormEvent<HTMLSelectElement>) => {

2. const tmp = clientAddresses!.find((element) => {

3. return element.id === parseInt(event.currentTarget.value);

4. })

5. if (selectedVisit) {

6. setSelectedVisit({

7. ...selectedVisit,

8. info: {

9. ...selectedVisit.info,

10. client\_address: {

11. ...selectedVisit.info.client\_address,

12. id: tmp!.id,

13. street: tmp!.street,

14. building\_nr: tmp!.building\_nr,

15. flat\_nr: tmp!.flat\_nr,

16. city: tmp!.city

17. }

18. }

19. })

20. }

21. ...

22. }

23.

Fragment kodu 69. Implementacja kodu wywoływanego w momencie wybrania nowego adresu realizacji rezerwacji [źródło własne]

Ostatnim polem, które może zostać zmodyfikowane przez klienta, jest pole z opisem. Ponownie jak przy tworzeniu rezerwacji, zostało ono zaimplementowane przy użyciu elementu *<textarea>* (fragment kodu 52). Po wprowadzeniu nowego opisu przez klienta, w funkcji *onChange* aktualizowane jest pole *info.description* w obiekcie *selectedVisit*.

Po wprowadzeniu zmian w rezerwacji i kliknięciu przez niego przycisku „Zmodyfikuj” u dołu okna rezerwacji, wywoływana jest funkcja *modifyVisit*, której implementację przedstawiono poniżej (fragment kodu 70):

1. const modifyVisit = () => {

2. if (!hasVisitChanged()) {...}

3. if (!isDateFromFuture(selectedVisit!.info.start\_date)) {...}

4. if (!isDateFromFuture(selectedVisit!.info.end\_date)) {...}

5. if (selectedVisit!.info.start\_date.getTime() > selectedVisit!.info.end\_date.getTime())

{...}

6. if (selectedVisit!.info.start\_date.getTime() === selectedVisit!.info.end\_date.getTime())

{...}

7. if (selectedVisit!.info.start\_date.getDate() !== selectedVisit!.info.end\_date.getDate())

{...}

8. if (isNewTimeOverlappingVisits(selectedVisit!.info.start\_date,

selectedVisit!.info.end\_date, selectedVisit!.info.id)) {...}

9. if (selectedVisit!.info.description.length === 0) {...}

10. const updatedVisit = cloneDeep(selectedVisit!)

11. updatedVisit!.info.status = 'specialist\_action\_required'

12. updateVisit(updatedVisit, 'specialist\_action\_required', "Pomyślnie zaktualizowano wizytę")}

Fragment kodu 70. Implementacja funkcji modifyVisit [źródło własne]

W powyższym kodzie przeprowadzana jest walidacja danych wprowadzonych przez klienta, które zawarte są w obiekcie *selectedVisit*. Sprawdzane jest, czy klient wprowadził jakiekolwiek zmiany (funkcja *hasVisitChanged* – linia 2), czy wprowadzone daty są poprawne (linie 3-8), oraz czy opis nie jest pusty (linia 9). Po przeprowadzeniu walidacji, na podstawie *selectedVisit* tworzony jest obiekt *updatedVisit* z polem *info.status* ustawionym na „specialist\_action\_required”. Tak utworzony obiekt przekazywany jest do funkcji *updateVisit*, wraz z tekstem „specialist\_action\_required”. Implementację wspomnianej funkcji ukazano poniżej (fragment kodu 71):

1. function updateVisit(visit: VisitCalendar, newStatus: string, message: string) {

2. ...

3. const method = "PATCH"

4. fetch(`/client/visit/update`, {

5. body: JSON.stringify(visit),

6. method: method,

7. headers: headers,

8. credentials: 'include'

9. })

10. .then((response) => response.json())

11. .then((data) => {

12. if (data.error) {...}

13. else {

14. ...

15. if (!data.error && selectedVisit) {

16. setSelectedVisit({

17. ...selectedVisit,

18. info: {

19. ...selectedVisit?.info,

20. status: newStatus

21. }

22. })

23. }

24. ...

25. const requestOptions = {

26. method: "GET",

27. headers: headers,

28. }

29. fetch(`http://localhost:8080/visits/0/${client!.id}`, requestOptions)

30. .then((response) => response.json())

31. ...

32. createNotification(newStatus)

33. }

34. })

35. .catch((err) => {

36. console.log(err)

37. })

38. }

Fragment kodu 71. Implementacja funkcji updateVisit [źródło własne]

W powyższym kodzie na początku wysyłane jest zapytanie *POST* pod adres „/client/visit/update” wraz z obiektem *visit* w formacie *JSON* jako ciało zapytania (linie 4-9). Serwer obsługuje to zapytania za pomocą metody *UpdateVisitByClient*, a jej implementację przedstawiono poniżej (fragment kodu 72):

1. func (app \*Application) UpdateVisitByClient(w http.ResponseWriter, r \*http.Request) {

2. var visit models.VisitCalendar

3. err := app.readJSON(w, r, &visit)

4. ...

5. if !(visit.Info.Status == "declined") {

6. visits, err :=

app.DB.GetCalendarVisitsBySpecialistIdOrClientId(&visit.Specialist.Id, nil)

7. if err != nil {

8. fmt.Println("error getting Calendar Visits from db: ", err)

9. \_ = app.errorJSON(w, err)

10. return

11. }

12. for \_, v := range visits {

13. if v.Info.Status == "declined" {

14. continue

15. }

16. if v.Info.Id != visit.Info.Id {

17. if isVisitOverlapping(visit.Info.StartDate,

visit.Info.EndDate, v.Info.StartDate, v.Info.EndDate) {

18. fmt.Println("visit overlapping on existing visits")

19. \_ = app.errorJSON(w, fmt.Errorf("Wybrany termin nakłada się na

istniejącą wizytę specjalisty"))

20. return

21. }

22. }

23. }

24. timeOffs, err := app.DB.GetTimeOffBySpecialistId(visit.Specialist.Id)

25. for \_, t := range timeOffs {

26. if isVisitOverlapping(visit.Info.StartDate, visit.Info.EndDate, t.StartDate,

t.EndDate) {

27. fmt.Println("visit is overlapping timeOff")

28. \_ = app.errorJSON(w, fmt.Errorf("Wybrany termin nakłada się na termin

urlopu specjalisty"))

29. return

30. }

31. }

32. }

33. err = app.DB.UpdateVisit(visit)

34. if err != nil {

35. \_ = app.errorJSON(w, err)

37. return

38. }

39. \_ = app.writeJSON(w, http.StatusOK, visit.Info.Id)

40. }

Fragment kodu 72. Implementacja metody obsługującej żądanie modyfikacji wizyty [źródło własne]

W powyższej metodzie, na początku otrzymane dane mapowane są do zmiennej *visit*. Następnie pobierana jest lista wizyt specjalisty przypisanego do modyfikowanej rezerwacji za pomocą metody *GetCalendarVisitsBySpecialistIdOrClientId*, którą opisano wcześniej (fragment kodu 42). Na podstawie otrzymanej listy, wykonywana jest pętla, w której za pomocą funkcji *isVisitOverlapping* sprawdzane jest czy zmodyfikowana rezerwacja nie nakłada się na inną rezerwację (linie 12-23). W kolejnym kroku za pomocą metody *GetTimeOffBySpecialistId* (również opisanej poprzednio – fragment kodu 55), pobierana jest lista urlopów specjalisty, a następnie w pętli sprawdzane jest, czy zmodyfikowana rezerwacja nie nakłada się na żaden z nich (linie 23-31). Po poprawnej walidacji, wywoływana jest metoda *UpdateVisit* (linia 33), która na podstawie obiektu *visit* oraz odpowiedniego zapytania *SQL* modyfikuje rezerwację w bazie danych. Jej implementację ukazano poniżej (fragment kodu 73):

1. func (m \*PG) UpdateVisit(visit models.VisitCalendar) error {

2. ctx, cancel := context.WithTimeout(context.Background(), timeout)

3. defer cancel()

4.

5. q := sql.UpdateVisit

6. \_, err := m.DB.ExecContext(ctx, q,

7. visit.Info.Id,

8. visit.Info.StartDate.Add(2\*time.Hour),

9. visit.Info.EndDate.Add(2\*time.Hour),

10. visit.Info.Price,

11. visit.Info.Status,

12. visit.Info.Description,

13. visit.Info.ClientAddress.Id,

14. )

15. if err != nil {

16. return fmt.Errorf("cannot update Visit: %w", err)

17. }

18. return nil

19. }

Fragment kodu 73. Implementacja metody modyfikującej rezerwację w bazie danych [źródło własne]

Po pomyślnym zmodyfikowaniu rezerwacji, w aplikacji klienckiej wyświetlana jest informacja o pomyślnym przeprowadzeniu modyfikacji, po czym wykonywane jest zapytanie *GET*, pobierające listę rezerwacji klienta (fragment kodu 71, linia 29). Za pomocą funkcji *createNotification* (linia 32) tworzone jest również nowe powiadomienie w bazie.

Rezerwacje, które w systemie posiadają status „specialist\_action\_required”, „declined” lub „accepted”, również wyświetlają formularz jak w przypadku rezerwacji o statusie „client\_action\_required”, jednak interakcja z nim nie jest umożliwiona.

W przypadku rezerwacji posiadających status „accepted”, „specialist\_action\_required” oraz „client\_action\_required”, u dołu okna rezerwacji znajduje się przycisk „Odrzuć”. Gdy rezerwacja posiada status „accepted” lub „client\_action\_required”, wyświetlany jest także przycisk „Zaakceptuj”. Po kliknięciu w jeden z nich, wywoływana jest funkcja changeVisitStatus wraz z odpowiednim statusem podanym jako argument („accepted” lub „declined”). Implementację tej funkcji przedstawiono poniżej (fragment kodu 74)

1. const changeVisitStatus = (status: string) => {

2. const updatedVisit = cloneDeep(visits!.find((element) => {

3. return element.info.id === selectedVisit!.info.id;

4. }))

5. updatedVisit!.info.status = status

6. if (status === 'declined') {

7. updateVisit(updatedVisit!, 'declined', "Pomyślnie odrzucono wizytę")

8. setNotification({

9. ...notification,

10. type: 'declined'

11. })

12. } else if (status === 'accepted') {

13. updateVisit(updatedVisit!, 'accepted', "Pomyślnie zaakceptowano wizytę")

14. setNotification({

15. ...notification,

16. type: 'accepted'

17. })

18. }

19. }

20.

Fragment kodu 74. Implementacja funkcji changeVisitStatus [źródło własne]

Powyższa funkcja w zależności od podanego nowego statusu, wywołuje funkcję *updateVisit*, której implementację opisano poprzednio (fragment kodu 71), a która ustawia status rezerwacji w bazie danych jako „declined” lub „accepted”.

# Instrukcja obsługi

Zawiera zrzuty ekranu pokazujące kolejne kroki, jakie powinien podjąć użytkownik, aby wykonać konkretną funkcjonalność w serwisie,

## Strony ogólnodostępne

## Instrukcja obsługi dla zalogowanych klientów

## Instrukcja obsługi dla zalogowanych specjalistów

# Podsumowanie

Zawiera informacje o możliwym dalszym rozwoju aplikacji, listę funkcjonalności, które zostały zrealizowane oraz te niezrealizowane.

# Bibliografia

|  |  |
| --- | --- |
| [1] | e-pasje, „e-pasje,” 19 06 2024. [Online]. Available: https://e-pasje.pl/wady-i-zalety-internetu/. |
| [2] | Maksym248, „Silver Sharky,” 05 07 2023. [Online]. Available: https://silversharky.pl/pl/blog/Jak-Technologia-wplywa-na-nasze-zycie/74. |
| [3] | Sprzedajemy.pl, „Gdzie szukać usług, zleceń, fachowców, wykonawców? Internet, gazety, znajomi, gdzie jeszcze?,” [Online]. Available: https://sprzedajemy.pl/poradnik/gdzie-szukac-uslug-zlecen-fachowcow-wykonawcow-internet-gazety-znajomi-gdzie-jeszcze-85b084-art3015. |
| [4] | „Fixly,” [Online]. Available: https://fixly.pl/. |
| [5] | A. Wojtaszek, 28 08 2017. [Online]. Available: https://fixly.pl/niezbednik/fixly-o-nas/. |
| [6] | „Oferteo,” [Online]. Available: https://www.oferteo.pl/. |
| [7] | „homerun,” [Online]. Available: https://homerun.pl/. |
| [8] | „Fingoweb,” [Online]. Available: https://www.fingoweb.com/pl/slownik-it/postgresql/. |
| [9] | „vavatech,” [Online]. Available: https://vavatech.pl/technologie/bazy-danych/postgresql. |
| [10] | „OVHcloud,” [Online]. Available: https://www.ovhcloud.com/pl/lp/postgresql-definition/. |
| [11] | „linuxpolska,” [Online]. Available: https://linuxpolska.com/pl/baza-wiedzy/blog/postgresql-najczesciej-wybierana-baza-danych-przez-deweloperow-i-przyszlosc-dbms/. |
| [12] | „linuxpolska,” [Online]. Available: https://linuxpolska.com/pl/baza-wiedzy/blog/postgresql-najczesciej-wybierana-baza-danych-przez-deweloperow-i-przyszlosc-dbms/. |
| [13] | T. Kozon, „Boring Owl,” 26 9 2023. [Online]. Available: https://boringowl.io/blog/go-jezyk-programowania-ksztaltujacy-przyszlosci. |
| [14] | „mindboxgroup,” [Online]. Available: https://mindboxgroup.com/pl/do-jakich-zastosowan-najlepszy-bedzie-jezyk-go/. |
| [15] | „TIOBE,” [Online]. Available: https://www.tiobe.com/tiobe-index/. |
| [16] | D. Bigosiński, „semcore,” 5 9 2023. [Online]. Available: https://semcore.pl/co-to-jest-java-script-na-stronie-internetowej/. |
| [17] | K. Dopierała, „widoczni,” [Online]. Available: https://widoczni.com/blog/najwiekszy-slownik-marketingu-internetowego-w-polsce/javascript/. |
| [18] | P. Ćwik, „Clockwork Java,” 4 7 2019. [Online]. Available: https://clockworkjava.pl/2019/07/co-to-typescript/. |
| [19] | A. Siczek, „global4net,” 27 8 2020. [Online]. Available: https://global4net.com/blog/ecommerce/czym-jest-react-i-jakie-ma-zalety/. |
| [20] | „smartbees,” 4 2 2020. [Online]. Available: https://smartbees.pl/blog/react-js. |
| [21] | „statista,” [Online]. Available: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/. |

1. Backend to element infrastruktury niewidoczny dla użytkowników, odpowiadający za logikę i przetwarzanie danych aplikacji, https://cyberfolks.pl/slownik/backend/ [↑](#footnote-ref-1)
2. Frontend odpowiada za wyrenderowanie warstwy wizualnej użytkownikowi, wyświetlenie danych z backendu oraz przekazanie ich z powrotem w formie zapytań, https://webmakers.expert/blog/co-to-jest-front-end [↑](#footnote-ref-2)
3. Open source to oprogramowanie, którego kod źródłowy udostępniany jest bezpłatnie do użytku, rozpowszechniania i modyfikowania przez użytkowników, https://harbingers.io/definicje/open-source [↑](#footnote-ref-3)
4. JSON to otwarty format zapisu i wymiany struktur danych, składający się z par klucz – wartość, https://grupa-improve.pl/json-co-to-jest/ [↑](#footnote-ref-4)
5. Gorutyna to wydajny wątek wykonujący funkcje, https://sii.pl/blog/wspolbieznosc-w-go/ [↑](#footnote-ref-5)
6. API (Application Programming Interface) to zbiór reguł i protokołów umożliwiających komunikację między oprogramowaniem różnego typu, https://coderslab.pl/pl/blog/co-to-jest-api-wszystko-o-interfejsie-programowania-aplikacji [↑](#footnote-ref-6)
7. HTML (Hyper Text Markup Language) to standardowy język znaczników, wykorzystywany przy tworzeniu i wyświetlaniu stron internetowych w przeglądarce, https://adboosters.pl/tworzenie-stron-internetowych/co-to-jest-html-i-do-czego-sluzy-poznaj-najpopularniejsze-znaczniki-i-atrybuty/ [↑](#footnote-ref-7)
8. CSS (Cascading Style Sheets) to język używany do opisu układu elementów i sposobu ich wyświetlania na stronie, https://udigroup.pl/blog/co-to-jest-jezyk-css-kaskadowe-arkusze-stylow/ [↑](#footnote-ref-8)
9. Framework to szkielet budowy aplikacji, dostarczający biblioteki oraz komponenty potrzebne do działania aplikacji, https://mayko.pl/slownik-e-marketingu/framework/ [↑](#footnote-ref-9)
10. DOM (Document Object Model) reprezentuje w pamięci komputera strukturę dokumentu HTML w postaci modelu obiektowego, https://miroslawzelent.pl/kurs-javascript/document-object-model-dom-hierarchia/ [↑](#footnote-ref-10)
11. Endpoint – adres docelowy obsługujący konkretne zapytanie HTTP, http://www.infected.pl/programowanie/kursy-online-pozostale/kurs-testowanie-api/czym-sa-endpointy-w-api.html [↑](#footnote-ref-11)
12. REST API – styl architektury oprogramowania, składający się ze zbioru reguł opisujących sposób, w jaki zdefiniowane są zasoby, https://www.czyitjestdlamnie.pl/post/czym-jest-i-jak-dziala-api-http [↑](#footnote-ref-12)
13. DAL (Data Access Layer) – moduł odpowiedzialny za logikę dostępu do bazy danych, https://bykowski.pl/dao-czy-repository-jaka-warstwe-dostepu-do-danych-wykorzystywac/ [↑](#footnote-ref-13)
14. CORS (Cross-Origin Resource Sharing) – mechanizm umożliwiający bezpieczny dostęp do zasobów z innych domen, https://dhosting.pl/pomoc/baza-wiedzy/czym-jest-cors-i-jak-dziala/ [↑](#footnote-ref-14)
15. Hook – funkcja zawierająca akcje odpowiedzialne za stan komponentu i jego funkcje, https://www.wakeupandcode.pl/react-hooks/ [↑](#footnote-ref-15)
16. Funkcja anonimowa – funkcja która nie posiada nazwy, https://kursjs.pl/kurs/super-podstawy/funkcje#funkcja-anonimowa [↑](#footnote-ref-16)
17. JSX – składnia rozszerzenia JavaScript, pozwalająca na dodanie kodu HTML do kodu JavaScript, https://boringowl.io/blog/czym-jest-jsx [↑](#footnote-ref-17)