AKADEMIA TARNOWSKA

Wydział Politechniczny

Kierunek: ***Informatyka***

2024/2025

Klaudia Foszcz

PRACA INŻYNIERSKA

Projekt i implementacja aplikacji webowej do zarządzania rezerwacjami wizyt w salonie fryzjerskim

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2025*

Spis treści

1. Wstęp

Współczesne salony fryzjerskie coraz częściej korzystają z rozwiązań cyfrowych, aby sprostać oczekiwaniom klientów oraz zoptymalizować procesy zarządzania. Jednym z kluczowych wyzwań w branży jest efektywne planowanie wizyt, które pozwala na oszczędność czasu zarówno klientów, jak i personelu. W odpowiedzi na te potrzeby, powstał projekt aplikacji webowej umożliwiającej zarządzanie rezerwacjami w salonie fryzjerskim. Niniejsza praca inżynierska dąży do uproszczenia i usprawnienia organizacji wizyt w salonach fryzjerskich.

1.1. Cel pracy

Celem niniejszej pracy inżynierskiej jest zaprojektowanie i implementacja aplikacji webowej służącej do zarządzania rezerwacjami w salonie fryzjerskim. Aplikacja będzie umożliwiać użytkownikom przeglądanie dostępnych terminów oraz rezerwację wizyt online. Dodatkowo będzie oferowała funkcje przypominania o wizytach i wystawiania ocen za świadczone usługi. System powinien również zawierać panel administracyjny dla fryzjerów, umożliwiający zarządzanie harmonogramem, rezerwacjami oraz danymi klientów.

1.2. Zakres pracy

Zakres pracy inżynierskiej obejmował zaimplementowanie systemu składający się z trzech głównych elementów:

- Aplikacja serwerowa
- Aplikacja webowa
- Baza danych

W ramach pracy wykonano testy, które potwierdziły poprawność działania kluczowych funkcji systemu. Aplikacja została zaprojektowana tak, aby zapewnić intuicyjny interfejs, stabilność działania oraz odpowiednie środki bezpieczeństwa.

2. Analiza biznesowa

W ramach niniejszego rozdziału zostaną przedstawione wymagania funkcjonalne i niefunkcjonalne, dotyczące projektu do zarządzania rezerwacjami wizyt w salonie fryzjerskim. Wymagania te mają na celu określenie kluczowych funkcji systemu oraz standardów jakości, które aplikacja powinna spełniać, aby zapewnić jej efektywne działanie i wygodę użytkowników.

2.1. Wymagania funkcjonalne

2.1.1. Rejestracja i logowanie

Aplikacja powinna umożliwiać użytkownikom zakładanie kont poprzez proces rejestracji, który wymaga podania podstawowych danych. Po pomyślnej rejestracji każdy użytkownik będzie mógł logować się do aplikacji za pomocą swojego adresu e-mail i hasła. Fryzjerzy będą mieli dostęp do dodatkowych funkcji w panelu administracyjnym po zalogowaniu się na swoje konto.

2.1.2. Przeglądanie dostępnych terminów i rezerwacja wizyt

System powinien umożliwiać klientom przeglądanie dostępnych terminów wizyt w salonie fryzjerskim oraz ich rezerwację online. Klienci będą mogli bez trudu wybrać odpowiednią dla nich datę, godzinę oraz fryzjera na wybraną usługę. Tylko wolne godziny będą wyświetlane, a terminy już zajęte nie będą widoczne dla użytkowników. Po zarezerwowaniu wizyty system potwierdzi termin i wyśle klientowi powiadomienie.

2.1.3. Zarządzanie harmonogramem i danymi klientów

Panel administracyjny dostępny dla fryzjerów salonu umożliwi zarządzanie harmonogramem pracy. Fryzjerzy będą mogli aktualizować swoją dostępność oraz zmieniać grafiki pracy. Dodatkowo, panel pozwoli na przeglądanie i zarządzanie danymi klientów, takimi jak historia wizyt oraz informacje kontaktowe.

2.1.4. Przypomnienia o nadchodzących wizytach

System powinien automatycznie wysyłać przypomnienia do klientów o nadchodzących wizytach. Przypomnienia mogą być wysyłane poprzez e-mail, z odpowiednim wyprzedzeniem.

2.1.5. Możliwość oceny usług

Po zakończonej wizycie klient powinien mieć możliwość wystawienia opinii na temat wizyty w salonie fryzjerskim. Ocena może być wyrażona za pomocą gwiazdek, gdzie klient ocenia ogólne zadowolenie z usługi. Dodatkowo, klient będzie miał możliwość dodania opisu, w którym może szczegółowo przedstawić swoje wrażenia.

2.2. Wymagania niefunkcjonalne

2.2.1. Łatwość obsługi

Interfejs aplikacji musi być intuicyjny i prosty w obsłudze zarówno dla klientów, jak i pracowników salonu. Aplikacja powinna oferować przejrzysty układ elementów, z czytelnymi przyciskami oraz zrozumiałymi etykietami, co umożliwi użytkownikom łatwe poruszanie się po systemie. Wszystkie kluczowe funkcje, takie jak rezerwacja wizyt, przegląd dostępnych terminów, zmiana harmonogramu pracowników czy ocena usług, powinny być dostępne w kilku prostych krokach.

2.2.2. Responsywność

Aplikacja powinna być w pełni responsywna, co oznacza, że musi dostosowywać się do różnych urządzeń, takich jak komputery stacjonarne, laptopy, tablety i smartfony. Użytkownicy powinni mieć zapewnioną dobrą jakość obsługi, niezależnie od tego, z jakiego urządzenia korzystają.

2.2.3. Bezpieczeństwo

Dane powinny być przechowywane w sposób zaszyfrowany, aby zapewnić ich integralność i poufność. Aplikacja powinna wykorzystywać mechanizm uwierzytelniania oparty na tokenach JWT (ang. *JSON Web Token*). Po zalogowaniu użytkownika, system będzie generował token, który zawiera zaszyfrowane informacje o użytkowniku oraz jego uprawnieniach. Token ten jest następnie przesyłany do klienta i używany do autoryzacji kolejnych żądań, co pozwala na weryfikację tożsamości użytkownika bez konieczności wielokrotnego logowania się. Dodatkowo, zastosowanie tokenów JWT umożliwi łatwą implementację mechanizmów odświeżania tokenów oraz zarządzania sesjami, co zwiększa bezpieczeństwo aplikacji oraz ochronę przed nieautoryzowanym dostępem.

3. Opis wykorzystanych technologii

W niniejszym rozdziale zostaną omówione technologie wybrane do realizacji projektu. System, którego projekt oraz implementacja stanowią główny cel pracy inżynierskiej, będzie oparty na nowoczesnym stosie technologicznym, który zapewni efektywność i wydajność.

3.1. JavaScript

JavaScript to jeden z najpopularniejszych języków programowania, najczęściej wykorzystywany w aplikacjach internetowych. Został stworzony przez firmę Netscape w 1995 roku, a jego pierwotnym celem było uczynienie stron internetowych bardziej interaktywnymi. JavaScript obsługuje przetwarzanie danych po stronie klienta, umożliwiając tworzenie dynamicznych i responsywnych interfejsów użytkownika.

JavaScript jest językiem interpretowanym, co oznacza, że jego kod jest przetwarzany na bieżąco przez przeglądarkę lub środowisko serwerowe, bez potrzeby wcześniejszej kompilacji. Składnia języka jest zbliżona do C, co czyni go intuicyjnym dla osób zaznajomionych z innymi językami programowania. Współczesny JavaScript, określany jako ECMAScript (standaryzowany przez organizację ECMA International), umożliwia programowanie obiektowe, asynchroniczne, a także programowanie oparte na funkcjach, co zwiększa elastyczność i wszechstronność w tworzeniu aplikacji [1].

JavaScript charakteryzuje się m.in.:

- Obsługą zdarzeń i interaktywnością - funkcje wywoływane są przez zdarzenia użytkownika, np. kliknięcia czy przesunięcia myszką.
- Asynchronicznością - JavaScript obsługuje operacje asynchroniczne (np. promisy, async/await), co umożliwia wykonywanie działań bez blokowania głównego wątku.
- Wirtualnym DOM – w bibliotekach takich jak React, wirtualny DOM przyspiesza proces renderowania przez dynamiczne aktualizowanie tylko zmienionych elementów na stronie [1].

3.2. React.js

React.js to biblioteka oparta na komponentach, wykorzystywana do tworzenia interaktywnych interfejsów użytkownika. Jest to obecnie najpopularniejsze narzędzie frontendowe w środowisku JavaScript. W architekturze MVC (ang. *Model-View-Controller*) pełni rolę warstwy widoku (ang. *View*). React jest rozwijany przez Facebook, Instagram oraz szeroką społeczność programistów indywidualnych i organizacji. Pozwala na budowanie rozbudowanych aplikacji webowych, które dynamicznie zmieniają dane bez konieczności przeładowywania strony, co sprzyja płynniejszemu i bardziej responsywnemu działaniu. Jego głównym celem jest zapewnienie użytkownikowi wygodnych i płynnych doświadczeń oraz szybkie i niezawodne tworzenie nowoczesnych aplikacji internetowych [2].

3.3. Node.js

Node.js to środowisko uruchomieniowe JavaScriptu, które pozwala na wykonywanie kodu po stronie serwera. Powstało w 2009 roku z inicjatywy Ryana Dahla i jest oparte na silniku V8 firmy Google. Node.js umożliwia wykonywanie operacji asynchronicznych, co jest kluczowe w aplikacjach wymagających wysokiej wydajności i skalowalności. W odróżnieniu od tradycyjnych serwerów, aplikacja Node.js działa w jednym procesie, nie generując nowego wątku dla każdego nadchodzącego żądania [3].

Cechy Node.js obejmują:

- Asynchroniczność - obsługa operacji wejścia-wyjścia bez blokowania, co pozwala na efektywną pracę z wieloma zapytaniami jednocześnie.
- Wydajność - dzięki zastosowaniu V8, Node.js jest jednym z najwydajniejszych środowisk JavaScriptu.
- Rozbudowany ekosystem - Node Package Manager (npm) oferuje ogromną liczbę bibliotek, co przyspiesza proces tworzenia aplikacji [3].

3.4. Express.js

Express.js to lekki i szybki framework oparty na Node.js, który ułatwia tworzenie serwerowych aplikacji webowych. Express szybko zdobył popularność ze względu na prostotę i elastyczność, co czyni go idealnym narzędziem do tworzenia rozbudowanych API i backendów. Express jest podstawowym wyborem w stosie MERN, umożliwiając szybkie i wygodne zarządzanie żądaniami HTTP, routingu i integracji z bazami danych, takimi jak MongoDB [4].

Do cech Express.js należą:

- Minimalistyczna budowa - Express oferuje podstawowy zestaw funkcji i jest łatwy do rozszerzania za pomocą middleware, co pozwala na dostosowanie go do konkretnych wymagań projektu.
- Routing - system zarządzania ścieżkami pozwala na tworzenie wyraźnych i uporządkowanych struktur aplikacji, umożliwiając oddzielanie logiki dla różnych operacji.
- Obsługa błędów i middleware - Express umożliwia dodawanie funkcji pośrednich, które wykonują określone operacje przed ostatecznym przetworzeniem żądania [4].

Dzięki Express.js aplikacja do zarządzania rezerwacjami może być wydajna i łatwa do utrzymania. Dodatkowo, dzięki obsłudze middleware, możliwe jest sprawne zarządzanie autoryzacją, logowaniem i bezpieczeństwem danych, co zwiększa komfort jej użytkowania.

3.5. MongoDB

MongoDB to nierelacyjna baza danych (NoSQL), zaprojektowana przez firmę 10gen (obecnie MongoDB Inc.) i wprowadzona na rynek w 2009 roku. MongoDB przechowuje dane w formacie dokumentów JSON, co pozwala na większą elastyczność struktury danych niż w przypadku tradycyjnych relacyjnych baz danych. Dzięki temu jest dobrze dostosowany do nowoczesnych aplikacji, które wymagają skalowalności i elastycznego przetwarzania danych [5].

Charakterystyczne cechy MongoDB obejmują:

- Skalowalność - MongoDB obsługuje poziomą skalowalność, co pozwala na efektywne przetwarzanie dużych zbiorów danych w rozproszonych środowiskach.
- Elastyczny schemat - MongoDB umożliwia zmiany w strukturze dokumentów bez potrzeby modyfikowania całej bazy danych.
- Wsparcie dla indeksów i wyszukiwania - zapewnia szybki dostęp do danych i możliwość tworzenia niestandardowych indeksów [5].

W kontekście aplikacji do zarządzania rezerwacjami MongoDB doskonale sprawdza się w przechowywaniu danych o użytkownikach, rezerwacjach oraz ocenach.

3.6. JSON

JSON (*JavaScript Object Notation*) to format zapisu danych, który stanowi prosty sposób wymiany informacji między systemami. Dzięki swojej strukturze opartej na obiektach jest bardzo czytelny, co ułatwia współpracę między systemami oraz przeglądanie i manipulację danymi w kodzie JavaScript [6].

Cechy JSON obejmują:

- Lekkość - JSON wymaga mniej zasobów niż XML, co sprawia, że idealnie nadaje się do przesyłania danych w aplikacjach webowych.
- Czytelność - dane zapisane w JSON są łatwe do zrozumienia.

W aplikacji JSON służy jako format wymiany danych między frontendem a backendem, umożliwiając sprawną komunikację między różnymi komponentami systemu.

3.7. JWT

JSON Web Token (JWT) to standard wymiany zaszyfrowanych informacji między dwoma stronami. Tokeny JWT mogą przechowywać zaszyfrowane informacje o użytkowniku, które są przekazywane między serwerem a klientem, co eliminuje potrzebę korzystania z sesji na serwerze. JWT wspomaga autoryzację użytkowników w aplikacji, umożliwiając bezpieczne przechowywanie informacji o sesji [7].

3.8. Vite.js

Vite.js to nowoczesne i szybkie narzędzie do tworzenia aplikacji webowych, które zostało zaprojektowane w celu przyspieszenia procesu budowania i uruchamiania aplikacji frontendowych. Vite jest bardzo wydajny, ponieważ podczas developmentu korzysta z natywnego wsparcia dla ES Modules (ESM) w przeglądarkach, co umożliwia natychmiastowe ładowanie i aktualizowanie modułów bez konieczności pełnej kompilacji kodu. Vite.js zyskał ogromną popularność wśród deweloperów frontendowych, ponieważ znacząco upraszcza proces tworzenia aplikacji [8].

3.9. REST API

REST (ang. *Representational State Transfer*) to styl architektury systemów rozproszonych, który jest szeroko stosowany w budowie API dla aplikacji webowych. REST definiuje sposób komunikacji między klientem (np. frontendem aplikacji) a serwerem poprzez zestaw zasad i konwencji, które wspierają prostotę i skalowalność.

RESTful API (czyli API oparte na zasadach REST) wykorzystuje standardowe metody HTTP do zarządzania zasobami:

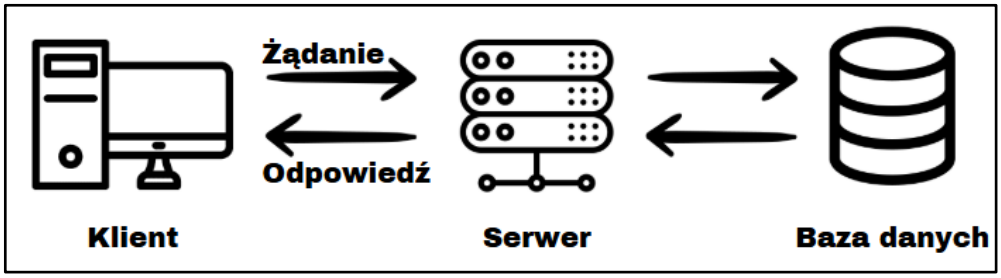
- POST - tworzy nowe zasoby na serwerze.
- GET - pobiera dane z serwera.
- PUT/PATCH - aktualizuje istniejące zasoby.
- DELETE - usuwa zasoby.

REST API często składa się z serii punktów końcowych (endpointów), które reprezentują różne zasoby, np. /users do zarządzania użytkownikami. Klient może uzyskać dostęp do tych ścieżek, wysyłając odpowiednie żądania HTTP, co umożliwia wymianę danych w formacie JSON lub XML. HTTP, czyli Hypertext Transfer Protocol, to podstawowy protokół sieciowy używany w komunikacji pomiędzy przeglądarką użytkownika a serwerem, na którym znajduje się aplikacja webowa. HTTP działa na zasadzie żądań i odpowiedzi: przeglądarka wysyła żądanie (ang. *request*) do serwera, a serwer odpowiada (ang. *response*), przesyłając dane, które przeglądarka następnie wyświetla użytkownikowi. Protokół ten umożliwia pobieranie różnych zasobów, takich jak dokumenty HTML, obrazy, style CSS, skrypty JavaScript i inne [9].

4. Implementacja systemu

Proces implementacji systemu zarządzania rezerwacjami wizyt w salonie fryzjerskim odbywał się w nowoczesnym i zintegrowanym środowisku programistycznym, co ułatwiło szybkie i efektywne tworzenie aplikacji webowej. Prace nad systemem były realizowane przy użyciu Visual Studio Code, popularnego edytora kodu, który oferuje szeroką gamę rozszerzeń i funkcji wspomagających.

4.1. Architektura systemu



Rys. 4.1 Architektura systemu [opracowanie własne]

Aplikacja do zarządzania rezerwacjami wizyt w salonie fryzjerskim została zbudowana w oparciu o architekturę klient–serwer i zgodnie z konwencją REST API, co umożliwia standaryzowaną i skalowalną komunikację pomiędzy komponentami systemu. Wymiana danych odbywa się za pomocą formatu JSON, który dzięki swojej uniwersalności i kompatybilności z językiem JavaScript znacząco usprawnia komunikację między frontendem i backendem.

System został podzielony na trzy główne moduły:

- Aplikacja serwerowa – odpowiada za obsługę logiki biznesowej, zarządzanie żądaniami klientów, autoryzację i uwierzytelnianie użytkowników oraz komunikację z bazą danych.
- Aplikacja webowa (frontend) – interfejs użytkownika został zbudowany przy użyciu nowoczesnych technologii frontendowych, takich jak React.js, co zapewnia dynamiczne i interaktywne doświadczenie użytkownika.
- Baza danych – system przechowywania danych oparty na MongoDB, umożliwiający przechowywanie i zarządzanie informacjami o rezerwacjach, użytkownikach oraz fryzjerach.

4.2. Diagram przypadków użycia

Diagram przypadków użycia przedstawia interakcje pomiędzy różnymi typami użytkowników w systemie zarządzania wizytami w salonie fryzjerskim. Użytkownicy systemu są podzieleni na role:

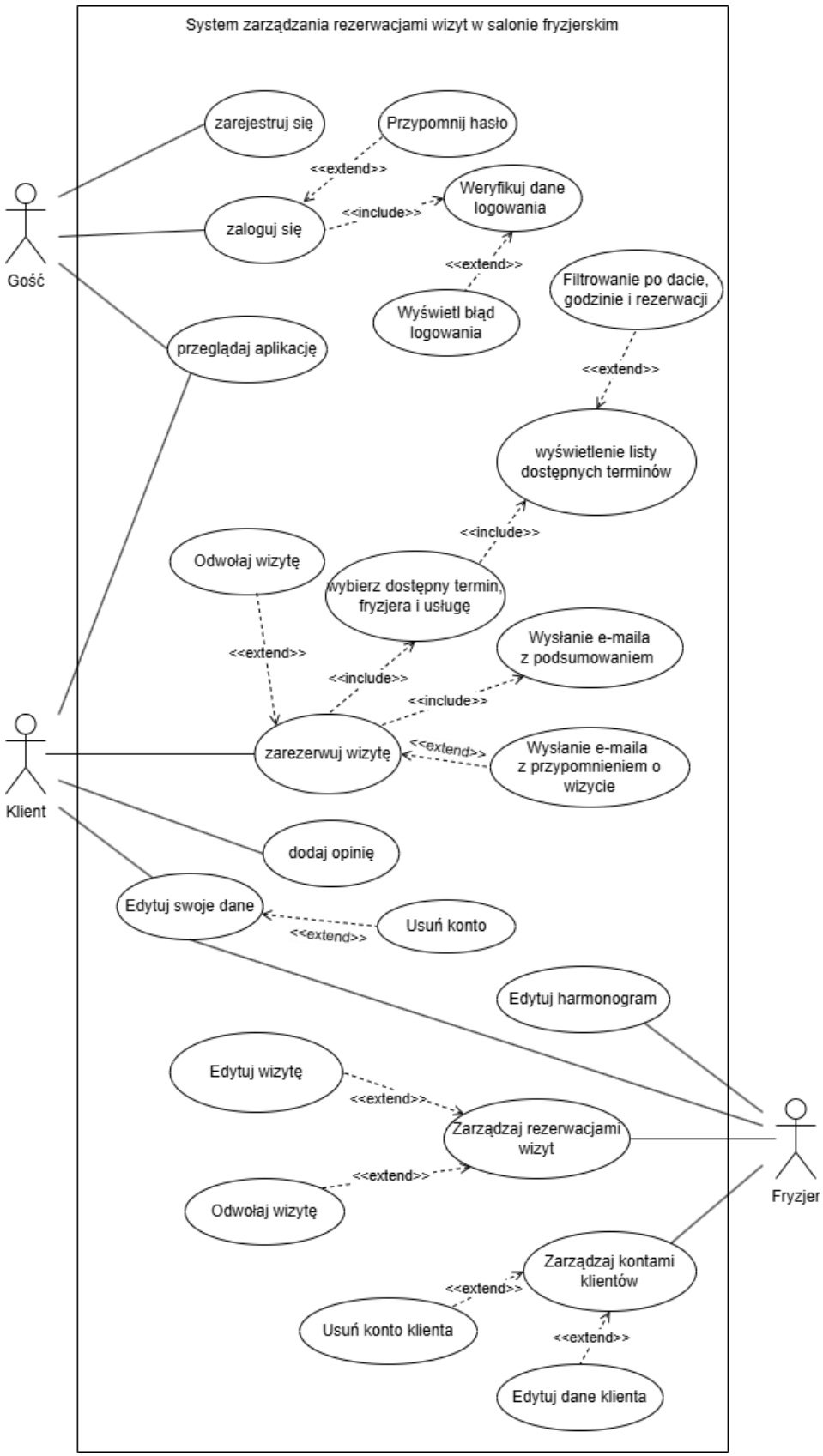
- Gość
- Klient (zalogowany użytkownik)
- Fryzjer (administrator)

Każda z tych ról ma przypisany zestaw uprawnień i funkcjonalności, które umożliwiają korzystanie z systemu w zależności od ich potrzeb oraz poziomu dostępu.

Gość to użytkownik o najniższym poziomie uprawnień, który nie jest zarejestrowany lub zalogowany w systemie. Ma możliwość założenia nowego konta, logowania się na istniejące konto oraz przeglądania aplikacji w ograniczonym zakresie. Gość może zapoznać się z podstawowymi informacjami o salonie, takimi jak dostępne usługi czy ogólne zasady działania aplikacji.

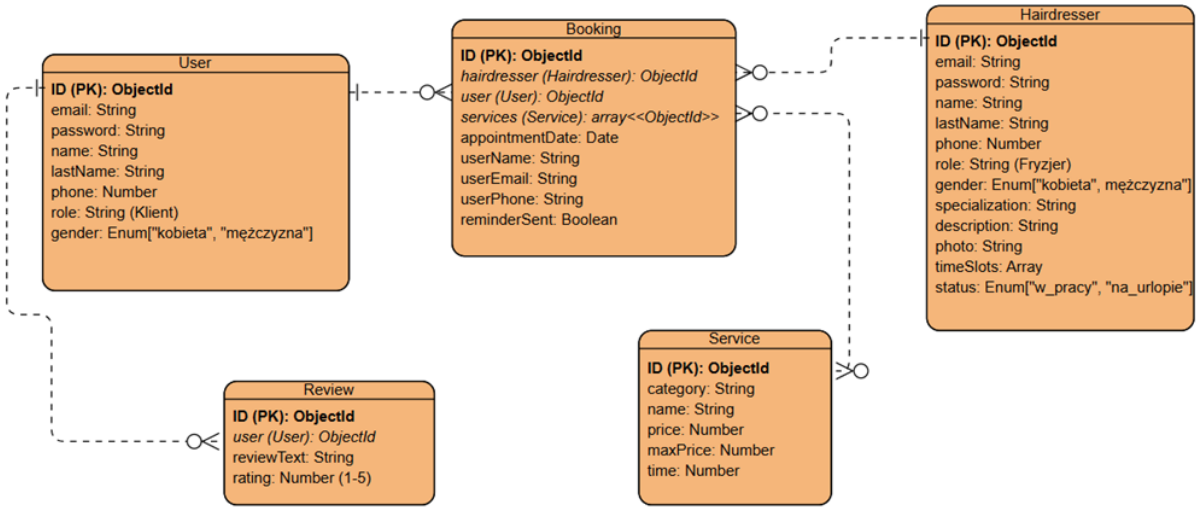
Klient to użytkownik, który po zalogowaniu zyskuje dostęp do większej ilości funkcjonalności aplikacji. Klient ma możliwość przeglądania i filtrowania dostępnych terminów wizyt oraz rezerwowania wizyt, z uwzględnieniem wybranego fryzjera oraz usługi. Po dokonaniu rezerwacji otrzymuje automatyczną wiadomość e-mail z podsumowaniem oraz przypomnienie o wizycie dzień wcześniej przed terminem. Klient może również odwoływać wizyty lub dodawać opinie o odbytych wizytach. Ponadto ma możliwość edycji własnych danych osobowych oraz usunięcia konta z systemu.

Fryzjer pełni rolę administratora systemu, co oznacza, że posiada najwyższy poziom uprawnień. Fryzjer może zarządzać rezerwacjami i swoim harmonogramem, co pozwala na elastyczne dostosowywanie godzin pracy. Dodatkowo posiada możliwość zarządzania kontami klientów, w tym edycji ich danych oraz usuwania kont w uzasadnionych przypadkach.



Rys. 4.2 Diagram przypadków użycia [opracowanie własne]

4.3. Diagram ERD



Rys. 4.3 Diagram ERD [opracowanie własne]

Diagram ERD przedstawiony na rysunku 4.2 prezentuje strukturę danych zaprojektowaną dla bazy w MongoDB. W tym modelu dane są strukturyzowane w sposób bardziej elastyczny, a relacje między dokumentami mogą być realizowane za pomocą:

▪ Referencji (ang. *References*) - dokumenty przechowują identyfikatory (*ObjectId*) odwołujące się do innych dokumentów.

▪ Zagnieżdżania danych (ang. *Embedding*) - dane związane logicznie z dokumentem mogą być przechowywane w jego wnętrzu.

Kolekcja User

Przechowuje dane użytkowników systemu. W MongoDB każdy użytkownik jest reprezentowany jako pojedynczy dokument w kolekcji User.

Struktura dokumentu:

- *email*, *password*, *name*, *lastName*, *phone*, *gender*, *role* - proste pola, przechowujące podstawowe informacje o użytkowniku.

Relacje:

- Referencja do kolekcji *Review* - przechowuje odwołania do opinii wystawionych przez użytkownika.
- Referencja do kolekcji *Booking* - użytkownik może mieć przypisane rezerwacje.

Kolekcja Hairdresser

Reprezentuje fryzjerów w systemie. W odróżnieniu od kolekcji *User*, fryzjerzy mają dodatkowe pola specyficzne dla ich roli w systemie.

Struktura dokumentu:

- *specialization, description, photo, timeSlots, status* – przechowują dane szczególne dla fryzjera, np. jego specjalizację, harmonogram pracy oraz status (*w_pracy*, *na_urlopie*).

Relacje:

- Referencja do kolekcji *Booking* – fryzjer jest powiązany z rezerwacjami, które obsługuje.

Kolekcja Booking

Przechowuje dane o rezerwacjach wizyt w salonie fryzjerskim. Dokumenty w tej kolekcji odwołują się do innych kolekcji za pomocą referencji, co pozwala na elastyczne zarządzanie rezerwacjami:

Struktura dokumentu:

- *hairdresser, user, services* – odwołania do fryzjera, klienta i usług związanych z daną wizytą (za pomocą *ObjectId*).
- *appointmentDate, reminderSent* – pola przechowujące dane związane z wizytą, np. termin wizyty i informację o przypomnieniu.

Relacje:

- Referencje do kolekcji *User* i *Hairdresser* umożliwiają łatwe śledzenie, kto zarezerwował wizytę i kto ją obsługuje.
- Referencje do kolekcji *Service* pozwalają na przypisanie usług do wizyty.

Kolekcja Service

Przechowuje dane o usługach oferowanych w salonie. Usługi są przechowywane jako oddzielne dokumenty, co pozwala na ich łatwą modyfikację i rozszerzanie.

Struktura dokumentu:

- *category, name, price, maxPrice, time* – pola opisujące kategorię, nazwę, ceny i czas trwania usługi.

Relacje:

- Referencje do kolekcji *Booking* – usługi są przypisane do rezerwacji wizyt.

Kolekcja Review

Przechowuje opinie użytkowników o odbytych wizytach.

Struktura dokumentu:

- *user*, *reviewText*, *rating* – użytkownik wystawiający opinię (referencja do kolekcji *User*), treść opinii i ocena.

Relacje:

- Referencja do kolekcji *User* – każda opinia jest przypisana do konkretnego użytkownika.

4.4. Główne ścieżki routingu

W tym podrozdziale zostaną przedstawione kluczowe ścieżki (routes) zaimplementowane w aplikacji:

- Ścieżki autoryzacji

```
router.post('/api/auth/login', login);
router.post('/api/auth/register', register);
router.post('/api/auth/reset-password', resetPassword);
router.post('/api/auth/verify-reset-code', verifyResetCode);
router.put('/api/auth/reset-password', updatePassword);
```

Rys. 4.4 Ścieżki autoryzacji [opracowanie własne]

Ścieżki na rysunku 4.1 przedstawiają routing dla operacji związanych z autoryzacją, logowaniem i rejestracją użytkownika w aplikacji. Oto opis każdej z nich:

- **POST /api/auth/login** – ścieżka służy do obsługi logowania użytkownika. Przyjmuje dane logowania (np. adres e-mail i hasło) i, jeśli są one poprawne, generuje odpowiedni token autoryzacyjny.
- **POST /api/auth/register** – ścieżka do rejestracji nowego użytkownika. Oczekuje danych rejestracyjnych (imię, nazwisko, adres e-mail, hasło, numer telefonu, płeć , rola) i tworzy nowe konto w systemie.
- **POST /api/auth/reset-password** – obsługuje żądanie resetu hasła użytkownika, wysyła kod weryfikacyjny na e-mail użytkownika.

- **POST /api/auth/verify-reset-code** – ścieżka służy do weryfikacji kodu resetu hasła. Sprawdza, czy kod otrzymany przez użytkownika jest poprawny i umożliwia kontynuację resetowania hasła.
- **PUT /api/auth/reset-password** – pozwala użytkownikowi na zmianę hasła po zweryfikowaniu kodu resetu. Wymaga podanie nowego hasła do zapisania w systemie.

Wszystkie wymienione ścieżki zostały zorganizowane w sposób umożliwiający kompleksowe zarządzanie procesem logowania, rejestracji i odzyskiwania hasła, co zapewnia bezpieczeństwo oraz wygodę użytkowników aplikacji.

- Ścieżki użytkowników

```
router.get('/api/users', authenticate, restrick(['fryzjer']), getAllUser);
router.get('/api/users/profile/:id', authenticate, restrick(['klient']), getUserProfile);
router.get('/api/users/myAppointments', authenticate, restrick(['klient']), getMyAppointments);
router.put('/api/users/:id', authenticate, restrick(['klient', 'fryzjer']), updateUser);
router.delete('/api/users/:id', authenticate, restrick(['klient', 'fryzjer']), deleteUser);
router.get('/api/users/:id', authenticate, restrick(['klient']), getSingleUser);
```

Rys. 4.5 Ścieżki użytkowników [opracowanie własne]

Ścieżki na rysunku 4.2 obsługują różne operacje dotyczące użytkowników i są zabezpieczone mechanizmami uwierzytelniania oraz restrykcji dostępu w oparciu o role (fryzjer i klient). Opis każdej z nich:

- **GET /api/users** – pozwala na pobranie listy wszystkich użytkowników. Dostęp do tej ścieżki mają jedynie użytkownicy o roli fryzjer, co umożliwia zarządzanie danymi użytkowników przez fryzjerów, pełniących funkcję administratorów.
- **GET /api/users/profile/:id** – umożliwia pobranie szczegółowego profilu użytkownika na podstawie jego identyfikatora (id). Ścieżka jest dostępna wyłącznie dla użytkowników posiadających rolę klient.
- **GET /api/users/myAppointments** – zwraca listę wszystkich wizyt danego użytkownika. Dostęp do tej ścieżki mają wyłącznie użytkownicy z rolą klient, co umożliwia przeglądanie własnych rezerwacji.
- **PUT /api/users/:id** – pozwala na aktualizację danych użytkownika, którego identyfikator (id) został podany. Operacja ta jest dostępna dla użytkowników o rolach klient i fryzjer, co umożliwia edycję danych użytkowników.

- **DELETE /api/users/:id** – umożliwia usunięcie użytkownika z bazy danych na podstawie jego identyfikatora (id). Omawiana ścieżka jest dostępna dla klientów i fryzjerów, dając im możliwość usuwania swoich kont lub kont innych użytkowników (w ramach odpowiednich uprawnień).
- **GET /api/users/:id** – pozwala na pobranie szczegółowych informacji o pojedynczym użytkowniku na podstawie jego identyfikatora (id). Dostęp do tej ścieżki mają wyłącznie użytkownicy posiadający rolę klient.

Każda z omawianych ścieżek jest odpowiednio zabezpieczona za pomocą funkcji uwierzytelniania i ograniczania dostępu (*authenticate* oraz *restrict*), co zapewnia, że dostęp do operacji mają jedynie uprawnieni użytkownicy, zgodnie z założonymi rolami.

- Ścieżki fryzjerów

```
router.get('/api/hairdressers/myAppointments', authenticate, restrick(['fryzjer']), getHairdresserAppointments);
router.get('/api/hairdressers', getAllHairdresser);
router.put('/api/hairdressers/:id', authenticate, restrick(['fryzjer']), updateHairdresser);
router.put('/api/hairdressers/:id/timeSlots', authenticate, restrick(['fryzjer']), updateTimeSlots);
router.delete('/api/hairdressers/:id', authenticate, restrick(['fryzjer']), deleteHairdresser);
router.get('/api/hairdressers/profile/:id', authenticate, restrick(['fryzjer']), getHairdresserProfile);
router.get('/api/hairdressers/:id/timeSlots', getHairdresserTimeSlots);
```

Rys. 4.6 Ścieżki fryzjerów [opracowanie własne]

Opis ścieżek przedstawionych na rysunku 4.3 dotyczących fryzjerów w aplikacji zarządzania salonem fryzjerskim:

- **GET /api/hairdressers/myAppointments** – umożliwia fryzjerowi dostęp do listy wszystkich jego rezerwacji wizyt. Dostęp do tej ścieżki mają wyłącznie użytkownicy o roli fryzjer, co pozwala im na zarządzanie własnymi wizytami.
- **GET /api/hairdressers** – pozwala na pobranie listy wszystkich fryzjerów. Używana jest do przeglądania dostępnych fryzjerów.
- **PUT /api/hairdressers/:id** – umożliwia aktualizację danych fryzjera na podstawie jego identyfikatora (id). Ta ścieżka jest dostępna wyłącznie dla użytkowników o roli fryzjer, co zapewnia, że aktualizacja danych jest możliwa tylko przez osoby z odpowiednimi uprawnieniami.
- **PUT /api/hairdressers/:id/timeSlots** – służy do aktualizacji harmonogramu danego fryzjera.

- **DELETE /api/hairdressers/:id** – pozwala na usunięcie fryzjera z bazy danych na podstawie jego identyfikatora (id). Dostęp do tej operacji mają wyłącznie fryzjerzy, co zabezpiecza tę funkcję przed nieautoryzowanym dostępem.
- **GET /api/hairdressers/profile/:id** – umożliwia pobranie szczegółowego profilu fryzjera na podstawie jego identyfikatora (id). Ścieżka jest dostępna dla fryzjerów, umożliwiając im przeglądanie szczegółowych danych profilu.
- **GET /api/hairdressers/:id/timeSlots** – pozwala na pobranie informacji o dostępnych godzinach fryzjera na podstawie jego identyfikatora (id). Umożliwia przeglądanie dostępności i zarządzanie harmonogramem.

- Ścieżki rezerwacji

```
router.post('/api/reservations', authenticate, restrick(['klient']), createReservation);
router.get('/api/reservations', getReservations);
router.get('/api/reservations/:hairdresserId/:date', getReservationsByHairdresserAndDate);
router.put('/api/reservations/bookings/:id', authenticate, updateReservation);
router.delete('/api/reservations/bookings/:id', authenticate, restrick(['klient', 'fryzjer']), deleteReservation)
```

Rys. 4.7 Ścieżki rezerwacji [opracowanie własne]

Ścieżki przedstawione na rysunku 4.4 odpowiadają za operacje związane z rezerwacjami wizyt w systemie. Oto opis każdej z nich:

- **POST /api/reservations** – umożliwia utworzenie nowej rezerwacji przez użytkownika. Ścieżka jest dostępna wyłącznie dla użytkowników z rolą klienta po pomyślnej autoryzacji.
- **GET /api/reservations** – pozwala na pobranie listy wszystkich rezerwacji.
- **GET /api/reservations/:hairdresserId/:date** – umożliwia uzyskanie listy rezerwacji przypisanych do konkretnego fryzjera w wybranym dniu. Pozwala na sprawdzenie dostępności fryzjera w określonym terminie.
- **PUT /api/reservations/bookings/:id** – umożliwia aktualizację istniejącej rezerwacji na podstawie jej identyfikatora (id).
- **DELETE /api/reservations/bookings/:id** – pozwala na usunięcie rezerwacji na podstawie jej identyfikatora (id). Operacja jest dostępna zarówno dla klientów i fryzjerów. Dzięki temu klienci mogą odwoływać swoje wizyty, a fryzjerzy mogą usuwać rezerwacje, jeśli jest taka potrzeba.

4.5. Wybrane fragmenty implementacji systemu

4.5.1. Rejestracja użytkownika

```
export const register = async (req, res) => {
    const { email, password, name, lastName, phone, role, gender, status, description, photo } = req.body;
    try {
        if (!email || !password || !name || !lastName || !role) {
            return res.status(400).json({ message: "Proszę podać wszystkie wymagane pola" });
        }
        // Sprawdzenie, czy użytkownik już istnieje
        let user = null;
        if (role === 'klient') {
            user = await User.findOne({ email });
        } else if (role === 'fryzjer') {
            user = await Hairdresser.findOne({ email });
        }
        if (user) {
            console.log("Użytkownik już istnieje");
            return res.status(400).json({ message: "Taki użytkownik już istnieje" });
        }
        // Hashowanie hasła
        const salt = await bcrypt.genSalt(10);
        const hashedPassword = await bcrypt.hash(password, salt);

        if (role === 'klient') {
            user = new User({
                email,
                password: hashedPassword,
                name,
                lastName,
                phone,
                gender: gender || undefined,
                role,
            });
        } if (role === 'fryzjer') {
            user = new Hairdresser({
                email,
                password: hashedPassword,
                name,
                lastName,
                phone,
                gender: gender || undefined,
                role,
                status: status || 'w_pracy',
                description: description || '',
                photo: photo || ''
            });
        }
        await user.save();
        res.status(201).json({ success: true, message: "Użytkownik został zarejestrowany" });

    } catch (error) {
        console.error("Błąd podczas rejestracji:", error);
        res.status(500).json({ success: false, message: "Coś poszło nie tak, spróbuj ponownie." });
    }
};
```

Rys. 4.8 Rejestracja użytkownika [opracowanie własne]

Funkcja przedstawiona na rysunku 4.7 rejestracji użytkownika umożliwia dodawanie nowych użytkowników do systemu, rozróżniając ich role (klient lub fryzjer). Sprawdza, czy wszystkie wymagane pola są wypełnione, sprawdza, czy użytkownik już istnieje na podstawie adresu e-mail. Proces rejestracji obejmuje walidację danych, zabezpieczenie hasła poprzez haszowanie z użyciem biblioteki bcrypt oraz zapis danych w odpowiedniej kolekcji bazy. Funkcja gwarantuje bezpieczeństwo danych użytkowników oraz elastyczność obsługi różnych typów użytkowników.

4.5.2. Pobieranie danych rezerwacji klienta

```
export const getMyAppointments = async (req, res) => {
    try {
        const bookings = await Booking.find({ user: req.userId })
            .populate('hairdresser', 'name lastName email phone')
            .select('-hairdresser.password');

        const sanitizedBookings = bookings.map(booking => ({
            _id: booking._id,
            hairdresser: booking.hairdresser,
            services: booking.services,
            appointmentDate: booking.appointmentDate,
            time: booking.time,
            userName: booking.userName,
            userEmail: booking.userEmail,
            userPhone: booking.userPhone,
            updatedAt: booking.updatedAt
        }));

        res.status(200).json({ success: true, message: 'Pobiera umówione wizyty', data: sanitizedBookings });
    } catch (error) {
        console.error(`Błąd: ${error.message}`);
        res.status(500).json({ success: false, message: 'Coś poszło nie tak' });
    }
};
```

Rys. 4.9 Pobieranie danych rezerwacji klienta [opracowanie własne]

Funkcja przedstawiona na rysunku 4.8 została zaprojektowana, aby umożliwić klientowi przeglądanie listy zarezerwowanych wizyt w sposób przejrzysty i uporządkowany. Zwracane dane obejmują szczegóły wizyty, takie jak informacje o fryzjerze, data, godzina oraz wybrane usługi, a jednocześnie chronione są wrażliwe informacje, które nie są istotne dla klienta. W przypadku wystąpienia błędów system generuje odpowiedni komunikat, gwarantując użytkownikowi jasność w razie ewentualnych problemów.

4.5.3. Pobieranie i aktualizacja harmonogramu fryzjera

```
export const getHairdresserTimeSlots = async (req, res) => {
    try {
        const hairdresser = await Hairdresser.findById(req.params.id);
        if (!hairdresser) {
            return res.status(404).json({ message: 'Fryzjer nie znaleziony' });
        }
        res.json({ timeSlots: hairdresser.timeSlots });
    } catch (error) {
        res.status(500).json({ message: 'Błąd serwera' });
    }
};

// Aktualizacja timeSlots fryzjera
export const updateTimeSlots = async (req, res) => {
    try {
        const { timeSlots } = req.body;
        const hairdresser = await Hairdresser.findByIdAndUpdate(
            req.params.id,
            { timeSlots },
            { new: true }
        );
        if (!hairdresser) {
            return res.status(404).json({ message: 'Fryzjer nie znaleziony' });
        }
        res.json({ message: 'Harmonogram pracy zaktualizowany', hairdresser });
    } catch (err) {
        res.status(500).json({ message: 'Błąd serwera', error: err.message });
    }
};
```

Rys. 4.10 Pobieranie i aktualizacja harmonogramu fryzjera [opracowanie własne]

Funkcja na rysunku 4.8 umożliwia fryzjerowi aktualizację harmonogramu pracy. Przyjmuje nową listę przedziałów czasowych i aktualizuje dane fryzjera w systemie. Po pomyślnej modyfikacji zwraca zaktualizowany harmonogram wraz z potwierdzeniem wykonanej operacji. Jeśli wskazany fryzjer nie istnieje w systemie, zwracany jest stosowny komunikat. W przypadku błędu serwera użytkownik zostaje poinformowany o problemie, co gwarantuje przejrzystość i kontrolę nad procesem. Obie funkcje wspierają elastyczne zarządzanie grafikiem fryzjera, dostosowując go do indywidualnych potrzeb oraz zapewniając aktualność dostępnych terminów w systemie.

4.5.4. Pobieranie, aktualizowanie i usuwanie rezerwacji

```
export const deleteReservation = async (req, res) => {
    const { id } = req.params;
    console.log(`Received request to delete booking with id: ${id}`);
    try {
        const deleted = await Booking.findByIdAndDelete(id);
        if (deleted) {
            res.status(200).json({ success: true, message: 'Usunięto pomyślnie' });
        } else {
            res.status(404).json({ success: false, message: 'Nie znaleziono rezerwacji' });
        }
    } catch (error) {
        console.error(`Błąd: ${error.message}`);
        res.status(500).json({ success: false, message: 'Nie usunięto pomyślnie' });
    }
};

// Aktualizacja rezerwacji
export const updateReservation = async (req, res) => {
    const { id } = req.params;
    const { appointmentDate, time, hairdresser, services } = req.body;
    try {
        const updatedReservation = await Booking.findByIdAndUpdate(
            id,
            { appointmentDate, time, hairdresser, services },
            { new: true }
        );
        if (!updatedReservation) {
            return res.status(404).json({ success: false, message: 'Nie znaleziono rezerwacji' });
        }
        res.status(200).json({ success: true, message: 'Rezerwacja zaktualizowana pomyślnie', updatedReservation });
    } catch (error) {
        console.error(`Błąd: ${error.message}`);
        res.status(500).json({ success: false, message: 'Nie zaktualizowano pomyślnie' });
    }
};

// Pobieranie wszystkich rezerwacji
export const getAllReservations = async (req, res) => {
    try {
        const reservations = await Booking.find().populate('user hairdresser services');
        res.status(200).json(reservations);
    } catch (error) {
        res.status(500).json({ message: 'Błąd podczas pobierania rezerwacji', error });
    }
};
```

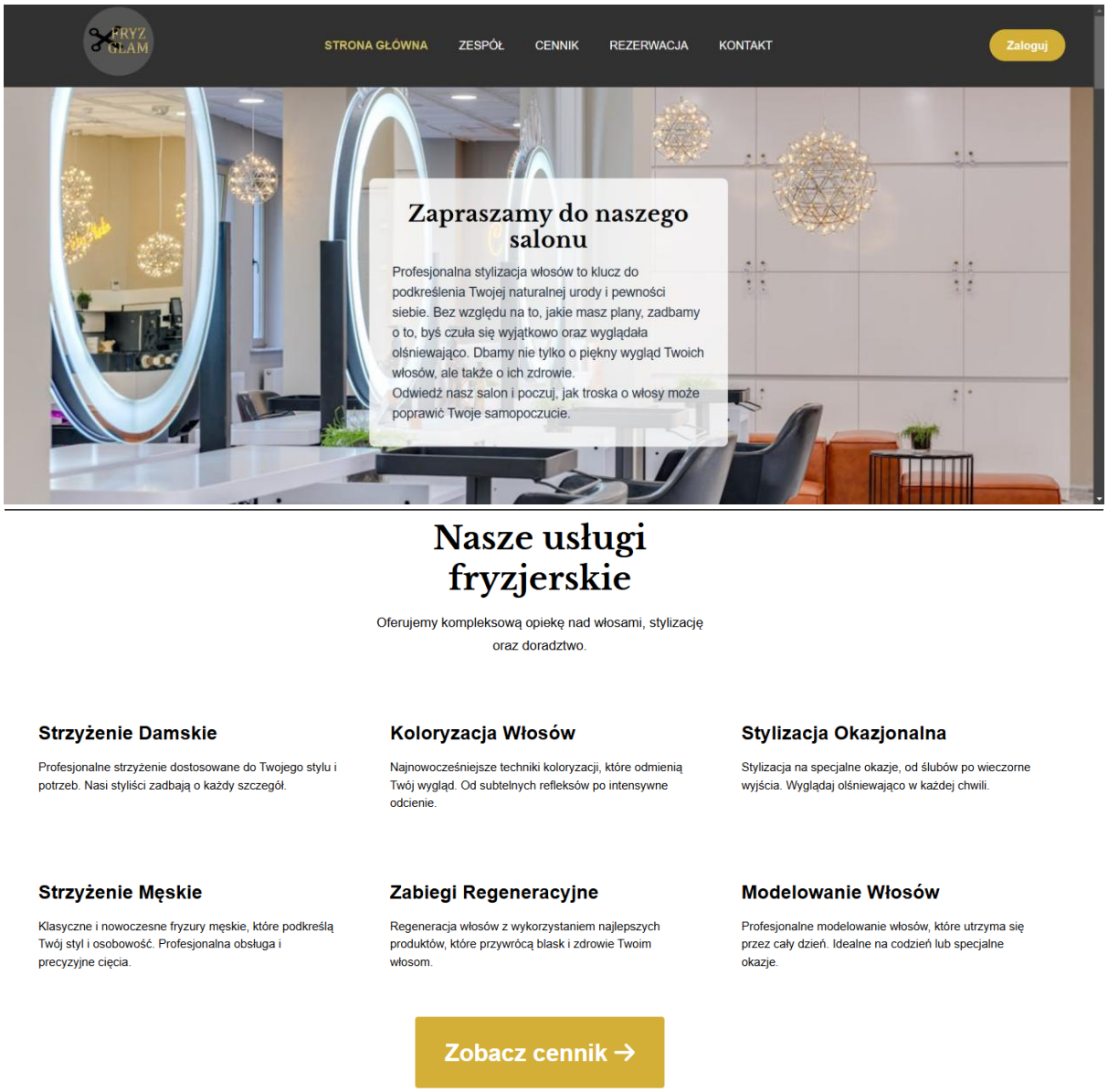
Rys. 4.11 Pobieranie, aktualizowanie i usuwanie rezerwacji [opracowanie własne]

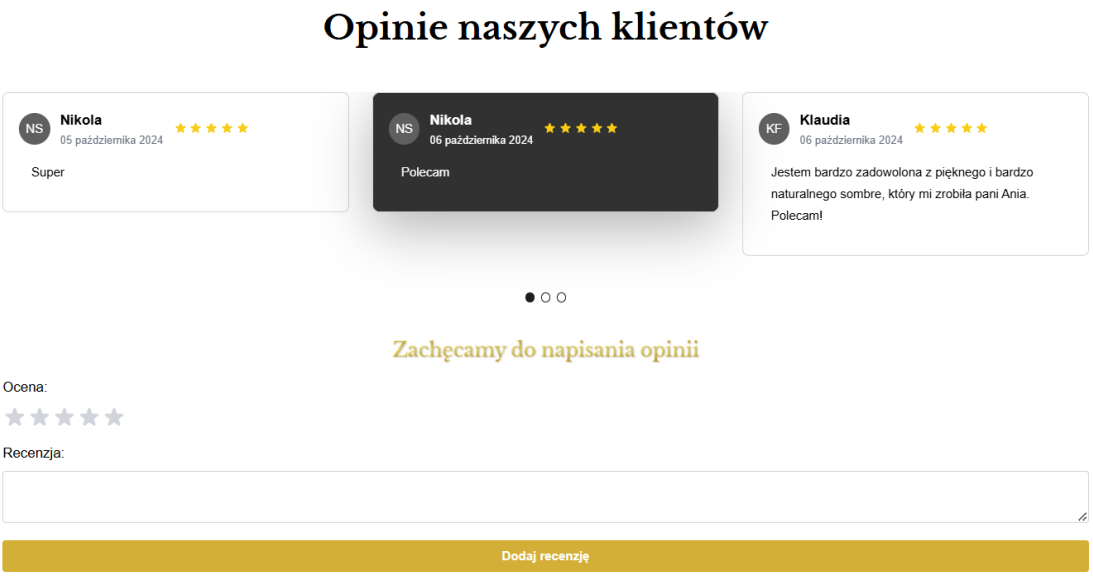
Funkcje przedstawione na rysunku dotyczą zarządzania rezerwacjami w systemie i obejmują trzy kluczowe operacje: usuwanie rezerwacji, aktualizację ich szczegółów oraz pobieranie pełnej listy rezerwacji. Funkcja *deleteReservation* pozwala na usunięcie wybranej rezerwacji na podstawie identyfikatora, zapewniając odpowiednie komunikaty. Funkcja *updateReservation* umożliwia modyfikację istniejącej rezerwacji, takich jak data, godzina, przypisany fryzjer czy wybrane usługi. W przypadku powodzenia zwracane są zaktualizowane dane, a w razie błędów użytkownik otrzymuje odpowiednią informację. Ostatnia funkcja, *getAllReservations*, służy do pobierania pełnej listy rezerwacji w systemie, wzbogaconych o szczegóły użytkowników, fryzjerów i usług.

5. Interfejsy użytkownika

W niniejszym rozdziale zostaną zaprezentowane najważniejsze elementy interfejsu użytkownika aplikacji, które pozwalają na korzystanie z kluczowych funkcji systemu. Omówione zostaną widoki dedykowane poszczególnym grupom użytkowników, takie jak klienci i fryzjerzy, wraz z opisem ich funkcjonalności oraz sposobem interakcji z aplikacją.

5.1. Strona główna





Rys. 5.1 Strona główna aplikacji [opracowanie własne]

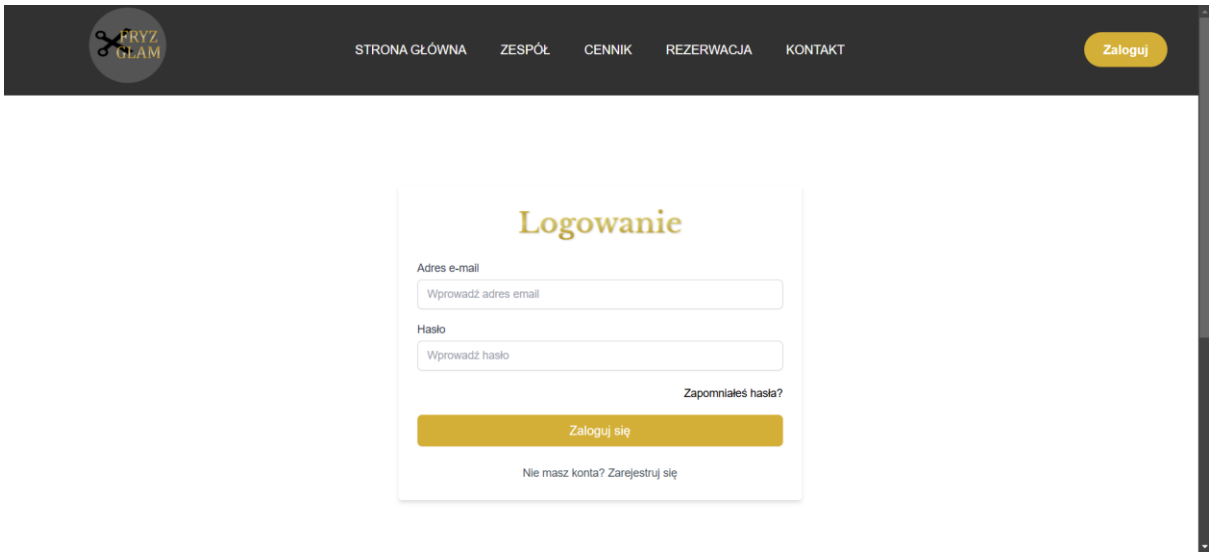
Na rysunku 5.1 przedstawiono wybrane elementy strony głównej aplikacji. Strona główna aplikacji zawiera kilka kluczowych sekcji, które mają na celu przyciągnięcie uwagi użytkowników oraz przedstawienie oferty salonu w sposób przejrzysty i estetyczny. Oto opis poszczególnych elementów strony głównej:

- **Nagłówek strony**:
  W górnej części strony znajduje się ciemny pasek nawigacyjny z logo salonu „Fryz Glam" po lewej stronie oraz odnośnikami do głównych sekcji: *Strona główna, Zespół, Cennik, Rezerwacja, Kontakt*. Po prawej stronie umieszczony jest przycisk *Zaloguj*, umożliwiający zalogowanie się do aplikacji.
- **Sekcja powitalna**:
  Centralną część strony otwiera zachęcające zaproszenie do salonu. W tle znajduje się zdjęcie wnętrza salonu, które dodatkowo wzmacnia wrażenie profesjonalizmu i dbałości o szczegóły.
- **Nasze usługi fryzjerskie**:
  Kolejna sekcja przedstawia ofertę salonu w formie sześciu kart z usługami. Każda karta zawiera krótki opis danej usługi, podkreślając jej jakość. Na dole tej sekcji znajduje się przycisk *Zobacz cennik*, który przenosi użytkownika do sekcji *Cennik*, gdzie znajdują się szczegółowe informacje na temat cen oferowanych usług.

- **Opinie klientów**:

  Na dole strony znajduje się interaktywna sekcja z opiniami klientów. Widać przykładowe recenzje wraz z ocenami w formie gwiazdek. Pod nimi znajduje się formularz umożliwiający pozostawienie własnej opinii z oceną i komentarzem. Formularz jest widoczny dla wszystkich użytkowników, jednak tylko zalogowani klienci mogą dodawać opinie.

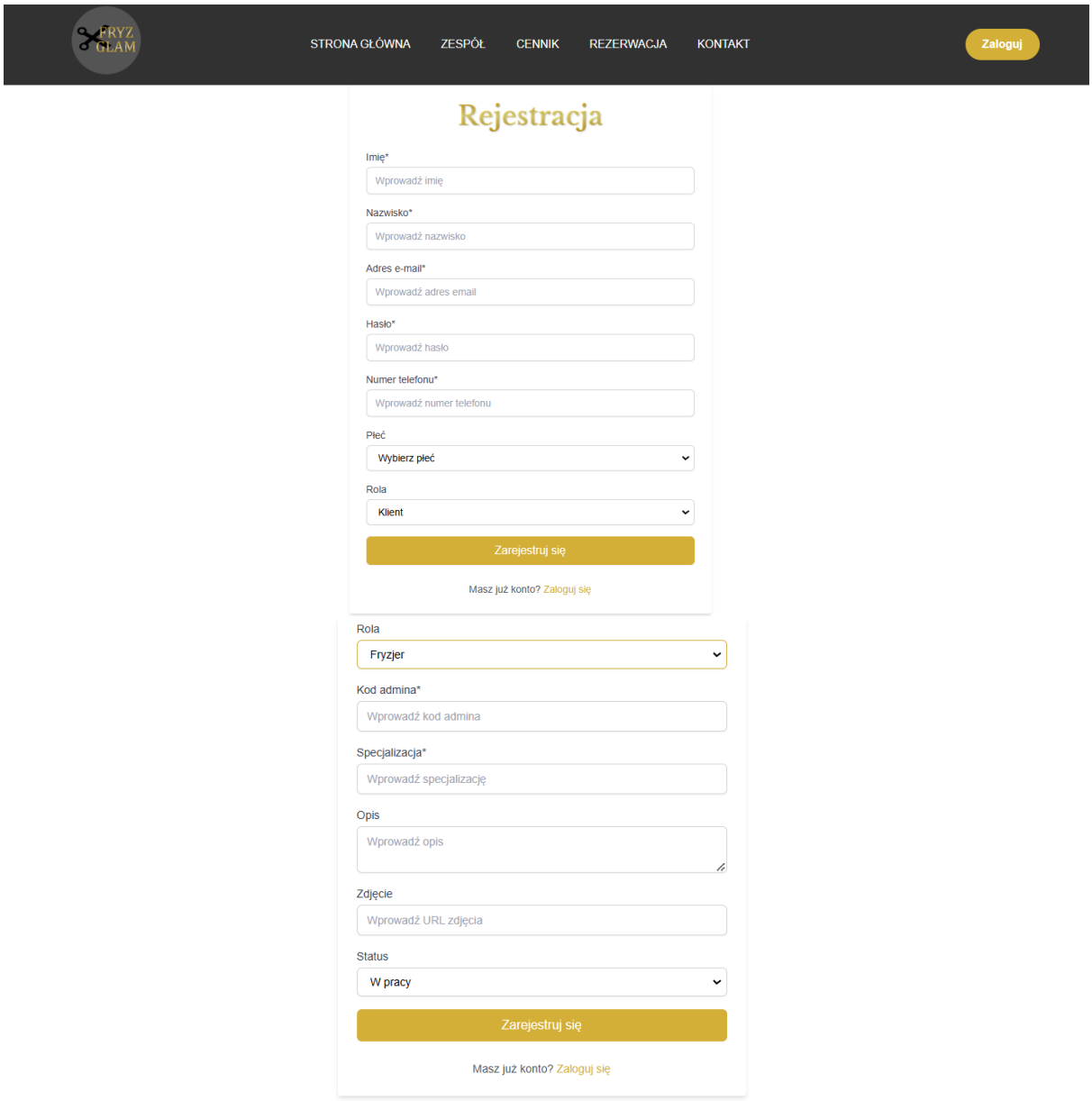
5.2. Logowanie



Rys. 5.2 Logowanie [opracowanie własne]

Rysunek 5.1 przedstawia ekran logowania do aplikacji, gdzie użytkownik wprowadza swój adres e-mail oraz hasło, aby zalogować się na swoje konto. Dodatkowo znajdują się odnośniki: *Zapomniałeś hasła?*, który umożliwia odzyskanie dostępu do konta, oraz *Nie masz konta? Zarejestruj się*, prowadzący do formularza rejestracyjnego.

5.3. Formularz rejestracji



Rys. 5.3 Formularz rejestracji użytkownika [opracowanie własne]

Na rysunku 5.3 znajduje się formularz rejestracji użytkownika. Formularz umożliwia rejestrację dwóch typów użytkowników: klientów oraz fryzjerów. Posiada odpowiednie pola do wprowadzenia danych osobowych i szczegółów wymaganych przy zakładaniu konta.

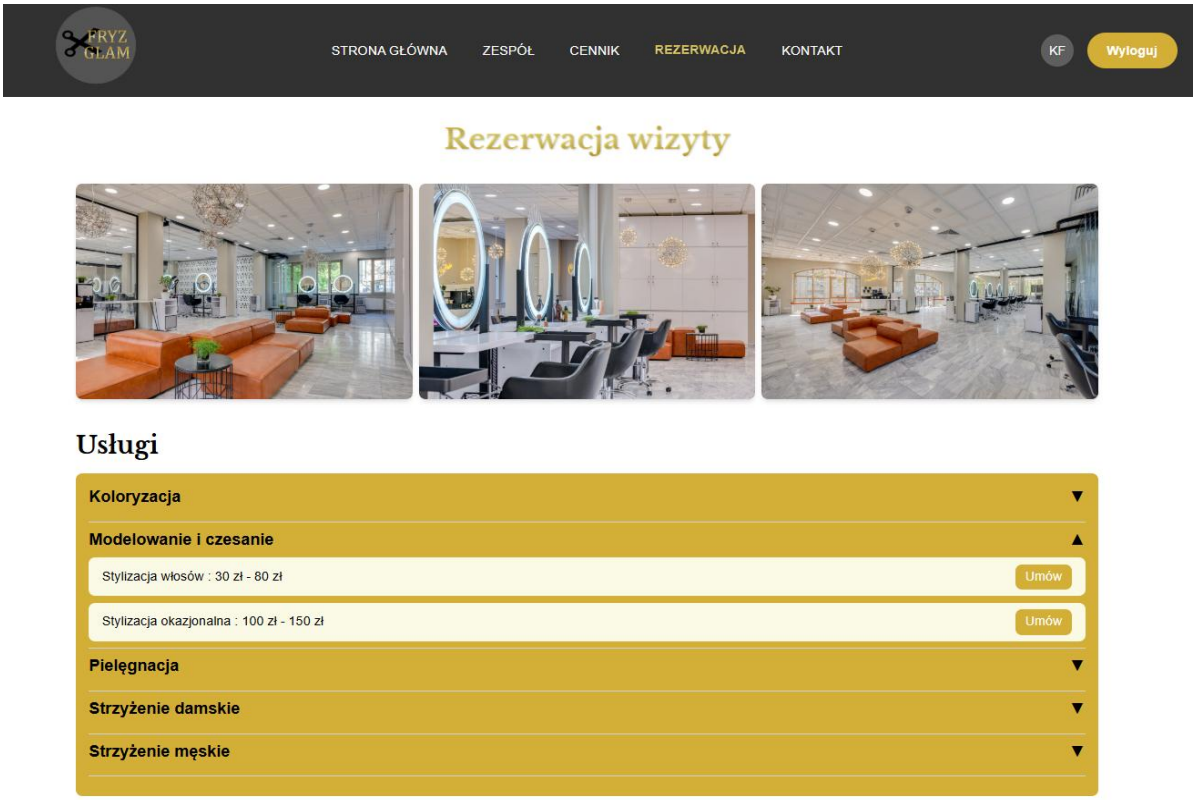
Pola formularza dla klientów:

- *Imię*
- *nazwisko*
- *adres e-mail*
- *hasło*
- *numer telefonu*
- *płeć*
- *rola* - domyślnie ustawiona na *Klient*.

Jeśli użytkownik wybierze rolę *Fryzjer*, formularz rozwija dodatkowe pola:

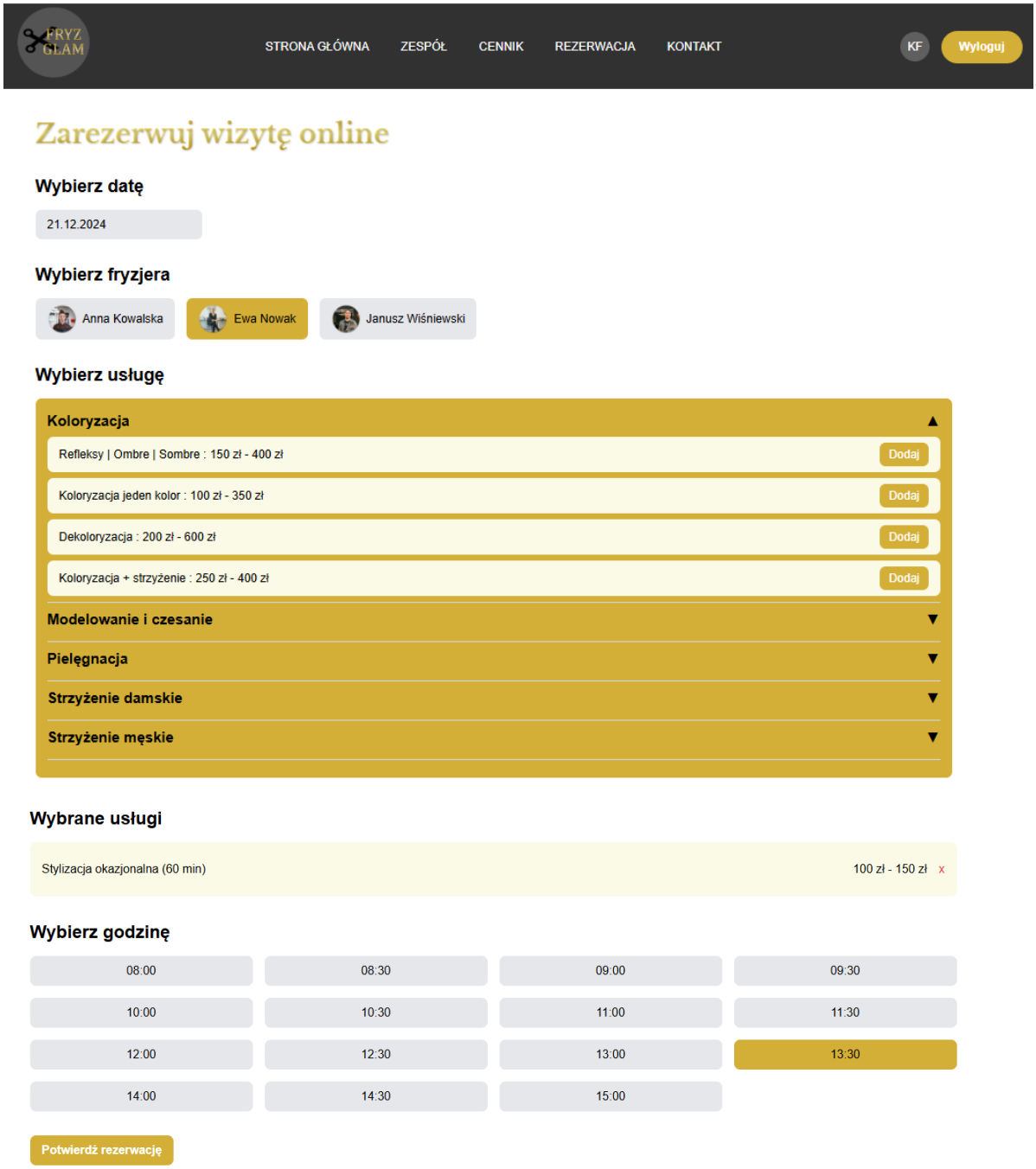
- *Kod admina* - pole do wprowadzenia kodu weryfikacyjnego.
- *Specjalizacja* - miejsce na wpisanie swojej specjalizacji (np. fryzjer damski, barber, master).
- *Opis* - informacje o fryzjerze.
- *Zdjęcie* - pole do wprowadzenia linku URL do zdjęcia fryzjera.
- *Status* - pole wyboru określające stan aktywności fryzjera.

5.4. Rezerwacja wizyty



Rys. 5.4 Przejście do rezerwacji wizyty [opracowanie własne]

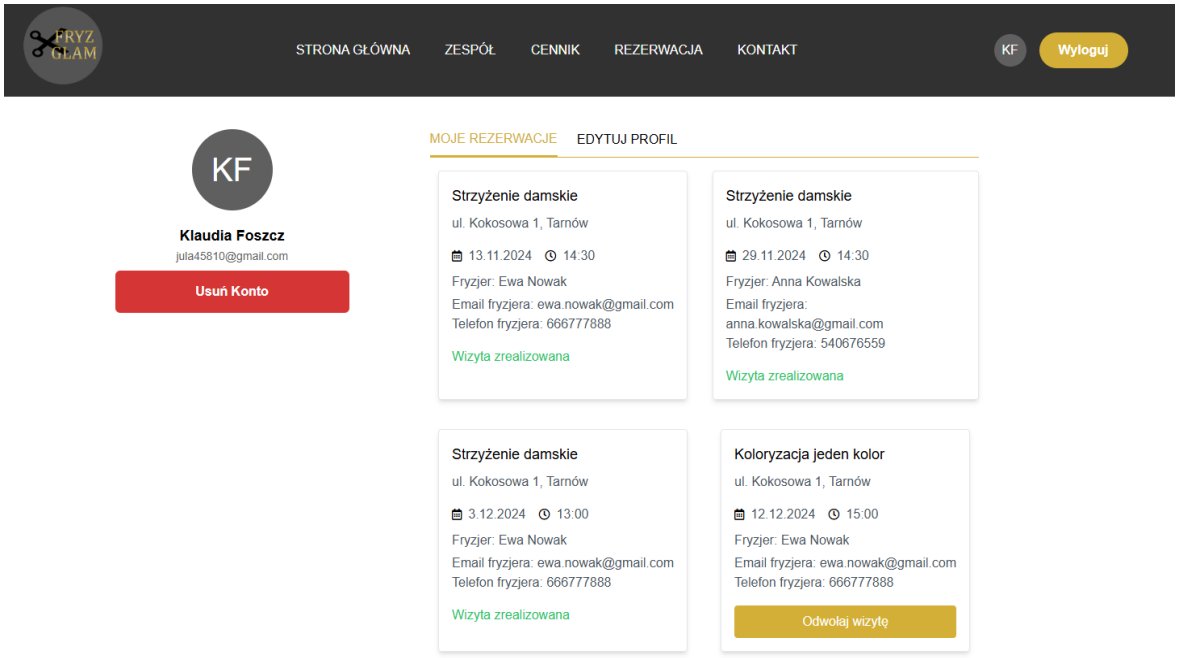
Rysunek 5.4 przedstawia stronę rezerwacji wizyty, na której znajduje się lista usług, podzielona na kategorie. Każda kategoria zawiera ofertę usług z podanymi przedziałami cenowymi. Obok konkretnych usług, jak np. *Stylizacja włosów* znajduje się przycisk *Umów*. Po kliknięciu w ten przycisk użytkownik przechodzi do strony z rezerwacją online. W przypadku, gdy klient nie jest zalogowany, aplikacja przekierowuje go na ekran logowania.
Rezerwacji może dokonać wyłącznie zalogowany klient, a fryzjerzy nie mają możliwości zarezerwowania wizyty.



Rys. 5.5 Rezerwacja wizyty online [opracowanie własne]

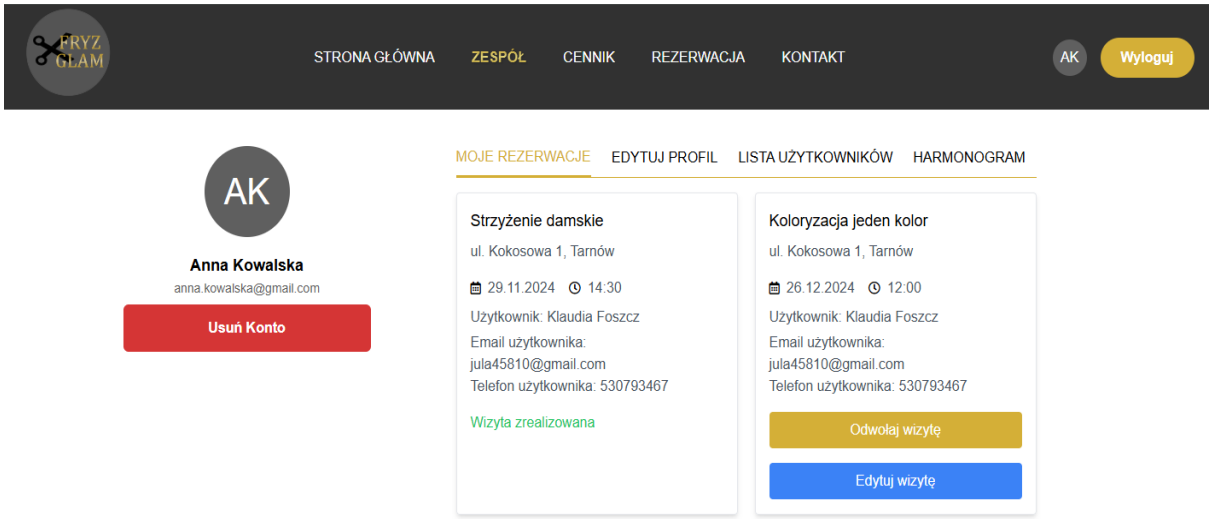
Rysunek 5.5 przedstawia ekran rezerwacji wizyty online, strona zawiera sekcję pozwalającą na szczegółowe zaplanowanie wizyty. Pierwszym krokiem jest wybór daty, który odbywa się poprzez kliknięcie w pola pod napisem *Wybierz datę*. Po kliknięciu rozwija się kalendarz, z którego można wybrać odpowiedni dzień. Następnie użytkownik wybiera fryzjera, klikając na jedną z dostępnych opcji przedstawionych w formie przycisków z imieniem i nazwiskiem fryzjera oraz jego zdjęciem. Kolejnym krokiem jest wybór usługi z rozwijanej listy kategorii, takich jak *Koloryzacja*, *Modelowanie i czesanie, Pielęgnacja, Strzyżenie damskie* lub *Strzyżenie męskie*. Po wybraniu konkretnej usługi, użytkownik dodaje ją do listy klikając przycisk *Dodaj*. W sekcji *Wybrane usługi* pojawiają się szczegóły dodanych usług, takie jak nazwa, czas trwania oraz cena. Użytkownik ma możliwość usunięcia wybranej usługi, jeśli zmieni zdanie. Ostatnim krokiem jest wybór godziny wizyty z dostępnych przedziałów czasowych, wybrana godzina zostaje podświetlona. Na dole strony znajduje się przycisk *Potwierdź rezerwację*, który finalizuje proces rezerwacji. Kliknięcie tego przycisku zapisuje wizytę, o ile wszystkie wymagane kroki zostały wykonane. Na podany adres e-mail wysyłane jest potwierdzenie z informacjami o wizycie. Dodatkowo, dzień przed planowaną wizytą klient otrzymuje wiadomość przypominającą. Jeśli rezerwacja została dokonana tego samego dnia, przypomnienie nie jest wysyłane.

5.5. Profile użytkowników

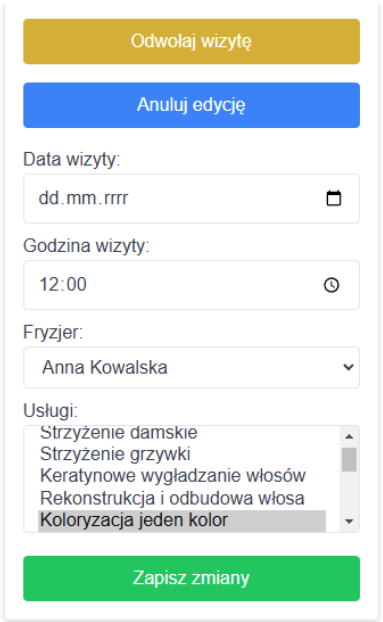


Rys. 5.6 Profil klienta [opracowanie własne]

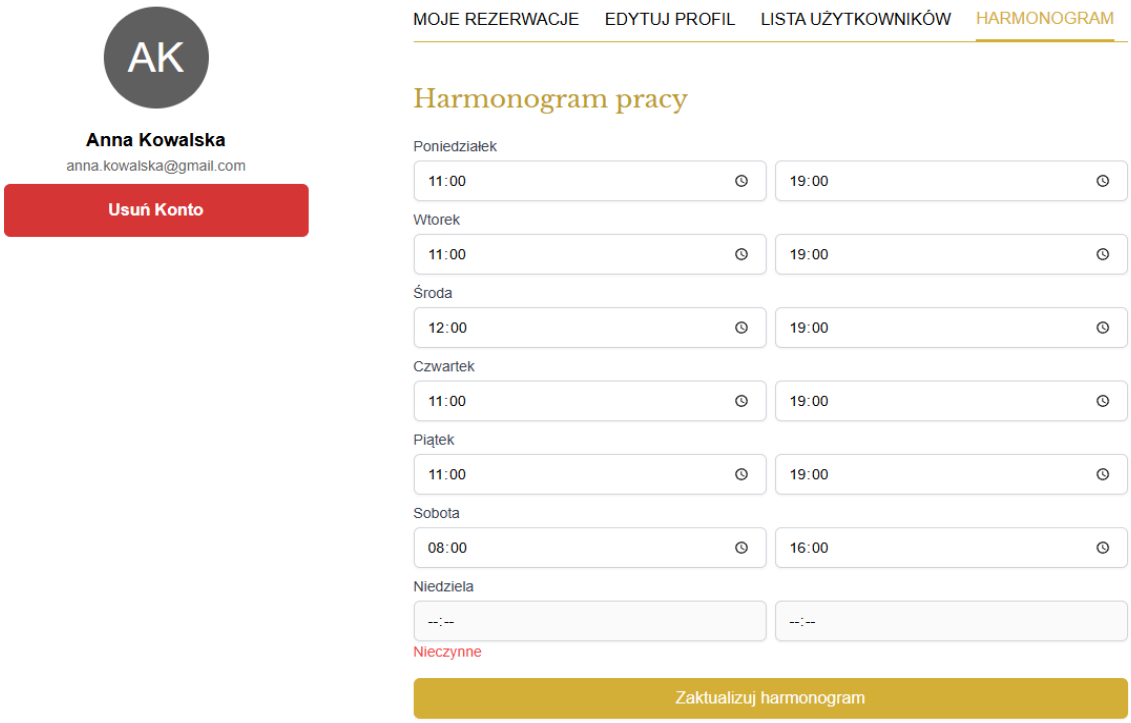
Rysunek 5.6 przedstawia profil klienta. W górnej części strony, na pasku nawigacyjnym, po prawej stronie znajduje się awatar zalogowanego użytkownika z inicjałami imienia i nazwiska oraz przycisk *Wyloguj*. Kliknięcie na awatar przenosi użytkownika właśnie do sekcji profilu, gdzie może przeglądać swoje rezerwacje w sekcji *Moje rezerwacje* i edytować swoje dane w sekcji *Edytuj profil*. Klient ma możliwość odwołania rezerwacji najpóźniej na dzień przed planowanym terminem wizyty. Jeśli wizyta została zrealizowana, czyli termin już minął, pojawi się status *Wizyta zrealizowana*. Dodatkowo użytkownik może usunąć swoje konto, korzystając z przycisku *Usuń konto*.



Rys. 5.7 Profil fryzjera [opracowanie własne]



Rys. 5.8 Edycja wizyty [opracowanie własne]



Rys. 5.9 Edycja harmonogramu fryzjera [opracowanie własne]

Profil fryzjera, zaprezentowany na rysunku 5.7, różni się znacząco od profilu klienta, ponieważ stanowi on panel administracyjny, zawierający dodatkowe funkcje. Oprócz możliwości przeglądania podstawowych informacji, takich jak dane osobowe, fryzjer ma dostęp do dwóch dodatkowych sekcji: *Lista użytkowników* oraz *Harmonogram*. Fryzjer, w przeciwieństwie do klienta, ma możliwość edycji wizyt. Przycisk *Edytuj wizytę,* znajdujący się pod przyciskiem *Odwołaj wizytę*, umożliwia otwarcie panelu edycji. Po jego kliknięciu wysuwa się panel, przedstawiony na rysunku 5.8, w którym fryzjer może zmieniać szczegóły rezerwacji, takie jak data, godzina, przypisany fryzjer czy wybrane usługi. Funkcja ta jest kluczowa w przypadku nagłych zmian w grafiku lub nieoczekiwanych sytuacji, które wymagają modyfikacji wcześniej zaplanowanych wizyt. W sekcji *Lista użytkowników* fryzjer ma możliwość edytowania danych użytkowników lub ich usunięcia. Sekcja *Harmonogram*, przedstawiona na rysunku 5.9, umożliwia fryzjerowi zarządzanie swoim grafikiem pracy, w tym zaktualizowanie swoich godzin pracy, które muszą mieścić się w godzinach otwarcia i zamknięcia salonu.

6. Testy

W niniejszym rozdziale zostanie przedstawiony proces testowania aplikacji, który odgrywa kluczową rolę w zapewnieniu jej jakości oraz niezawodności. Testowanie pozwala na identyfikację błędów, weryfikację zgodności systemu z wymaganiami oraz potwierdzenie prawidłowego działania wszystkich funkcji.

6.1. Testy jednostkowe

Testy jednostkowe są kluczowym elementem procesu zapewniania jakości aplikacji, ponieważ pozwalają na weryfikację poprawności działania poszczególnych komponentów systemu w izolacji. W trakcie implementacji testów jednostkowych w niniejszej aplikacji wykorzystano podejście **Behavior-Driven Development (BDD)**, które ułatwia czytelność i zrozumienie testów dzięki zastosowaniu techniki **Given-When-Then**.

Technika **Given-When-Then** opiera się na jasnym i logicznym podziale scenariuszy testowych:

- **Given -** określa początkowy stan systemu lub wymagane warunki wstępne.
- **When -** definiuje akcję, która ma być wykonana.
- **Then -** opisuje oczekiwany rezultat tej akcji [11].

Do przeprowadzenia testów wykorzystano framework **Jest**, który dostarcza narzędzia pozwalające na efektywne definiowanie przypadków testowych. Dzięki zastosowaniu **Jest Mock Functions** możliwe było symulowanie zależności, takich jak funkcje i metody, co pozwoliło na dokładne testowanie komponentów w pełnej izolacji od reszty systemu.

6.1.1. Testy rezerwacji wizyty

Rysunki 6.1, 6.2 i 6.3 przedstawiają zestaw testów jednostkowych dla funkcji *createReservation*, które odpowiadają za tworzenie nowych rezerwacji w systemie. Testy obejmują różne scenariusze, w tym zarówno przypadek pozytywny, jak i negatywne przypadki. Zanim każdy test zostanie wykonany, inicjowane są obiekty *req* i *res*, które symulują żądanie i odpowiedź przekazywane do funkcji.

Proces został przedstawiony na rysunku 6.1:

- *req.body* zawiera informacje o terminie wizyty (*appointmentDate*), godzinie (*time*), fryzjerze (*hairdresser*), oraz usługach (*services*).
- *req.userId* symuluje ID zalogowanego użytkownika.
- *res* zawiera symulacje metod *status* i *json*, które są wykorzystywane do zwracania odpowiedzi w API.

Test przedstawiony na rysunku 6.1 weryfikuje poprawne działanie funkcji w sytuacji idealnej, gdy wszystkie warunki są spełnione. Sprawdza, czy funkcja działa prawidłowo, gdy użytkownik i fryzjer istnieją w systemie oraz nie ma konfliktu z innymi rezerwacjami w podanym terminie.

```
describe('createReservation', () => {
    let req, res;

    beforeEach(() => {
        req = {
            body: {
                appointmentDate: '2024-12-23',
                time: '13:00',
                hairdresser: new mongoose.Types.ObjectId(),
                services: [new mongoose.Types.ObjectId()]
            },
            userId: new mongoose.Types.ObjectId()
        };
        res = {
            status: jest.fn().mockReturnThis(),
            json: jest.fn()
        };
    });

    it('powinien utworzyć nową rezerwację', async () => {
        // Given: Mockowanie danych
        jest.spyOn(User, 'findById').mockResolvedValue({ name: 'John', email: 'john@example.com', phone: '123456789' });
        jest.spyOn(Hairdresser, 'findById').mockResolvedValue({ name: 'Anna', lastName: 'Kowalska', status: 'w_pracy' });
        jest.spyOn(Service, 'findById').mockResolvedValue({ _id: new mongoose.Types.ObjectId(), time: 60 });
        jest.spyOn(Booking, 'findOne').mockResolvedValue(null);
        jest.spyOn(Booking.prototype, 'save').mockResolvedValue({ _id: new mongoose.Types.ObjectId(), createdAt: new Date(), updatedAt: new Date() });

        // When
        await createReservation(req, res);

        // Then
        expect(res.status).toHaveBeenCalledWith(201);
        expect(res.json).toHaveBeenCalledWith(expect.objectContaining({
            _id: expect.any(mongoose.Types.ObjectId)
        }));
    });
```

Rys. 6.1 Test utworzenia nowej rezerwacji [opracowanie własne]

```
it('nie powinien utworzyć rezerwacji, jeśli klient ma już rezerwację w tym terminie', async () => {
    // Given
    jest.spyOn(Booking, 'findOne').mockResolvedValue({});

    // When
    await createReservation(req, res);

    // Then
    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith({ success: false, message: 'Klient ma już rezerwację w tym terminie' });
});
```

Rys. 6.2 Test rezerwacji w tym samym terminie [opracowanie własne]

Test przedstawiony na rysunku 6.2 weryfikuje działanie funkcji w sytuacji, gdy klient próbuje zarezerwować termin, na który posiada już istniejącą rezerwację. W takim przypadku funkcja powinna odrzucić próbę utworzenia nowej rezerwacji i zwrócić odpowiedni komunikat o błędzie. Test symuluje sytuację, w której baza danych zwraca informacje o kolizji terminów dla tego samego klienta, co pozwala upewnić się, że system skutecznie uniemożliwia podwójne rezerwacje w tym samym czasie.

```
it('nie powinien utworzyć rezerwacji, jeśli fryzjer jest na urlopie', async () => {
    // Given
    jest.spyOn(Hairdresser, 'findById').mockResolvedValue({ name: 'Anna', lastName: 'Kowalska', status: 'na_urlopie' });

    // Given
    jest.spyOn(Booking, 'findOne').mockResolvedValue(null);

    // When
    await createReservation(req, res);

    // Then
    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith({ message: 'Fryzjer jest na urlopie' });
});
```

Rys. 6.3 Test rezerwacji w przypadku niedostępności fryzjera [opracowanie własne]

Test przedstawiony na rysunku 6.3 weryfikuje działanie funkcji w sytuacji, gdy fryzjer jest oznaczony jako *na urlopie*. W takim przypadku funkcja powinna odrzucić próbę utworzenia rezerwacji i zwrócić komunikat informujący o niedostępności fryzjera. Test symuluje scenariusz, w którym dane fryzjera w systemie wskazują na jego nieobecność, co pozwala upewnić się, że system poprawnie obsługuje tego rodzaju ograniczenia i uniemożliwia rezerwację u niedostępnego fryzjera.

```
PASS backend/tests/reservationController.test.mjs
  createReservation
    √ powinien utworzyć nową rezerwację (2203 ms)
    √ nie powinien utworzyć rezerwacji, jeśli klient ma już rezerwację w tym terminie (3 ms)
    √ nie powinien utworzyć rezerwacji, jeśli fryzjer jest na urlopie (3 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        3.586 s
```

Rys. 6.4 Rezultat testów tworzenia rezerwacji [opracowanie własne]

6.1.2. Testy kontrolera fryzjera

Rysunki 6.5 i 6.6 przedstawiają zestaw testów jednostkowych dla funkcji kontrolera fryzjera, które odpowiadają za usuwanie fryzjera i pobieranie wizyt fryzjera. Zanim testy zostaną wykonane, inicjowane są obiekty *req* i *res*, które symulują żądanie i odpowiedź przekazywane do funkcji. Proces jest przedstawiony na rysunku 6.5:

- *params.id* symuluje unikalny identyfikator fryzjera, który ma zostać usunięty. Jest to wartość generowana za pomocą *mongoose.Types.ObjectId()*, która jest wykorzystywana do wskazania konkretnego fryzjera w bazie danych.
- *userId* symuluje identyfikator użytkownika, który wykonuje operację usunięcia.

```
describe('Hairdresser Controller', () => {
    let req, res;

    beforeEach(() => {
        req = {
            params: { id: new mongoose.Types.ObjectId() },
            userId: new mongoose.Types.ObjectId()
        };
        res = {
            status: jest.fn().mockReturnThis(),
            json: jest.fn()
        };
    });

    describe('deleteHairdresser', () => {
        it('powinien usunąć fryzjera', async () => {
            // Given
            jest.spyOn(Hairdresser, 'findByIdAndDelete').mockResolvedValue({});

            // When
            await deleteHairdresser(req, res);

            // Then
            expect(res.status).toHaveBeenCalledWith(200);
            expect(res.json).toHaveBeenCalledWith({ success: true, message: 'Usunięto pomyślnie' });
        });
```

Rys. 6.5 Test usuwania fryzjera [opracowanie własne]

Test przedstawiony na rysunku 6.5 weryfikuje poprawne działanie funkcji usuwania fryzjera. Test ten sprawdza, czy funkcja *deleteHairdresser* działa poprawnie, gdy operacja usuwania fryzjera zakończy się powodzeniem. Obiekty *req* i *res* symulują odpowiednio dane wejściowe i wyjściowe dla funkcji, co pozwala na testowanie logiki funkcji bez potrzeby interakcji z rzeczywistą bazą danych.

```
describe('getHairdresserAppointments', () => {
    it('powinien zwrócić wizyty fryzjera', async () => {
        // Given
        const mockPopulate = jest.fn().mockResolvedValue([{ date: '2024-12-23' }]);
        jest.spyOn(Booking, 'find').mockReturnValue({ populate: mockPopulate });

        // When
        await getHairdresserAppointments(req, res);

        // Then
        expect(res.status).toHaveBeenCalledWith(200);
        expect(res.json).toHaveBeenCalledWith({ success: true, message: 'Znaleziono wizyty', data: [{ date: '2024-12-23' }] });
    });
```

Rys. 6.6 Test pobierania wizyt fryzjera [opracowanie własne]

Test przedstawiony na rysunku 6.6 sprawdza, czy funkcja *getHairdresserAppointments* prawidłowo zwraca wizyty przypisane do fryzjera. W tym przypadku, kiedy dane o wizytach fryzjera są poprawnie dostępne w bazie danych, test weryfikuje, czy funkcja zwraca odpowiedź z kodem 200 oraz poprawną listę wizyt (w tym przypadku zawierającą datę *2024-12-23*). Oczekiwana odpowiedź to komunikat *Znaleziono wizyty* z danymi wizyt.

```
PASS backend/tests/hairdresserController.test.mjs
  Hairdresser Controller
    deleteHairdresser
      √ powinien usunąć fryzjera (4 ms)
    getHairdresserAppointments
      √ powinien zwrócić wizyty fryzjera (2 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.971 s, estimated 1 s
```

Rys. 6.7 Rezultat testów kontrolera fryzjera [opracowanie własne]

6.2. Testy integracyjne

Testy integracyjne odgrywają istotną rolę w procesie zapewniania jakości oprogramowania, umożliwiając weryfikację współpracy różnych komponentów aplikacji. W przeciwieństwie do testów jednostkowych, które koncentrują się na izolowanym testowaniu pojedynczych funkcji lub modułów, testy integracyjne sprawdzają, czy system jako całość działa zgodnie z założeniami. Dzięki temu można wykryć błędy w interakcjach między modułami, takie jak problemy z wymianą danych, nieprawidłowe odpowiedzi API czy błędy komunikacji z bazą danych. Podobnie jak w testach jednostkowych, w testach integracyjnych zastosowano podejście **Behavior-Driven Development** (**BDD**) oraz technikę **Given – When – Then**.

Tak samo jak w testach jednostkowych, do testów integracyjnych wykorzystano framework **Jest**. Dodatkowo wykorzystano bibliotekę **Supertest**, służącą do wysyłania żądań HTTP do aplikacji Node.js w celu testowania jej ścieżek API.

6.2.1. Testy kontrolera uwierzytelniania

```
describe('Auth Controller', () => {
    let token;
    let resetCode;

    afterAll(async () => {
        // Usuwanie konta testowego
        await User.deleteMany({ email: { $in: ['newuser@example.com'] } });
    });

    it('powinien zarejestrować nowego użytkownika', async () => {
        // Given
        const newUser = {
            name: 'New',
            lastName: 'User',
            email: 'newuser@example.com',
            phone: '123456789',
            password: 'Password1',
            role: 'klient'
        };

        // When
        const res = await request(app)
            .post('/api/auth/register')
            .send(newUser);

        // Then
        console.log('Użytkownik zarejestrowany');
        expect(res.statusCode).toEqual(201);
        expect(res.body).toHaveProperty('success', true);
        expect(res.body.message).toBe('Użytkownik został zarejestrowany');
    });
```

Rys. 6.8 Test rejestracji nowego użytkownika [opracowanie własne]

Test przedstawiony na rysunku 6.8 weryfikuje, czy ścieżka API odpowiedzialna za rejestrację nowego użytkownika działa poprawnie. W tym przypadku wysyłane jest żądanie HTTP typu *POST* na ścieżkę */api/auth/register*, zawierające poprawne dane rejestracyjne, takie jak imię, nazwisko, adres e-mail, numer telefonu, hasło oraz rolę użytkownika. Oczekiwanym rezultatem jest odpowiedź z kodem 201 i komunikatem *Użytkownik został zarejestrowany*. Test sprawdza integrację między kontrolerem a bazą danych.

```
it('powinien zalogować użytkownika', async () => {
    // Given
    const loginUser = {
        email: 'newuser@example.com',
        password: 'Password1'
    };

    // When
    const res = await request(app)
        .post('/api/auth/login')
        .send(loginUser);

    // Then
    console.log('Użytkownik zalogowany');
    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveProperty('status', true);
    expect(res.body.message).toBe('Zalogowano pomyślnie');
    token = res.body.token;
});
```

Rys. 6.9 Test logowania użytkownika [opracowanie własne]

Na rysunku 6.9 przedstawiono test sprawdzający poprawność działania logowania użytkownika. W tym przypadku wysyłane jest żądanie HTTP typu *POST* na ścieżkę */api/auth/login*, zawierające adres e-mail oraz hasło zarejestrowanego wcześniej użytkownika. Oczekiwany wynik to odpowiedź z kodem 200, zawierająca komunikat *Zalogowano pomyślnie* oraz token autoryzacyjny. Test weryfikuje, czy proces uwierzytelniania użytkownika poprawnie łączy się z bazą danych oraz generuje token JWT.

```
it('powinien wysłać kod do zresetowania hasła', async () => {
    // Given
    const resetRequest = {
        email: 'newuser@example.com'
    };

    // When
    const res = await request(app)
        .post('/api/auth/reset-password')
        .send(resetRequest);

    // Then
    console.log('Email z kodem resetowania wysłany');
    expect(res.statusCode).toEqual(200);
    expect(res.body.message).toBe('Kod weryfikacyjny został wysłany na email');

    // Pobieranie kodu resetowania z bazy danych
    const user = await User.findOne({ email: 'newuser@example.com' });
    resetCode = user.resetCode;
});
```

Rys. 6.10 Test wysłania kodu resetowania hasła [opracowanie własne]

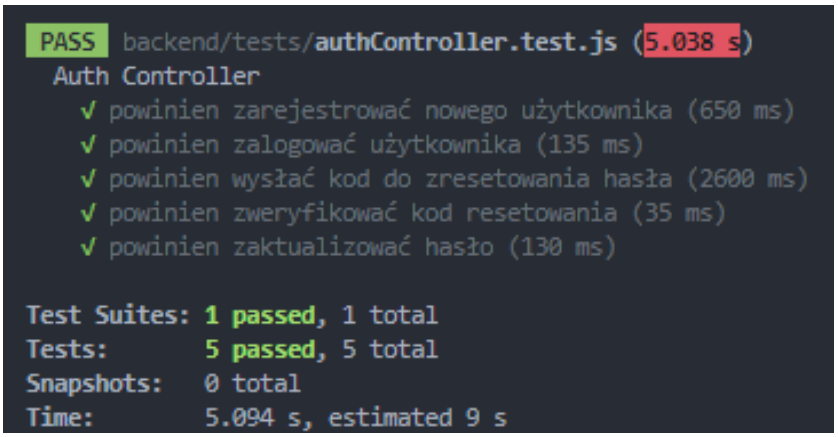
Test przedstawiony na rysunku 6.10 weryfikuje działanie ścieżki API odpowiedzialnej za wysłanie kodu weryfikacyjnego do zresetowania hasła na adres e-mail użytkownika. Po wysłaniu żądania *POST* na ścieżkę */api/auth/reset-password*, oczekiwanym rezultatem jest odpowiedź z kodem 200 i komunikatem 'Kod weryfikacyjny został wysłany na email'. Test sprawdza, czy serwer poprawnie odpowiada oraz czy kod resetowania został zapisany w bazie danych i jest dostępny do kolejnych etapów procesu.

```
it('powinien zweryfikować kod resetowania', async () => {
    // Given
    const verifyRequest = {
        email: 'newuser@example.com',
        code: resetCode
    };

    // When
    const res = await request(app)
        .post('/api/auth/verify-reset-code')
        .send(verifyRequest);

    // Then
    console.log('Kod resetowania zweryfikowany');
    expect(res.statusCode).toEqual(200);
    expect(res.body.message).toBe('Kod weryfikacyjny został zweryfikowany');
});

it('powinien zaktualizować hasło', async () => {
    // Given
    const updateRequest = {
        email: 'newuser@example.com',
        code: resetCode,
        newPassword: 'NewPassword2'
    };

    // When
    const res = await request(app)
        .put('/api/auth/reset-password')
        .send(updateRequest);

    // Then
    console.log('Hasło zaktualizowane');
    expect(res.statusCode).toEqual(200);
    expect(res.body.message).toBe('Hasło zostało zmienione');
});
```

Rys. 6.11 Test weryfikacji kodu resetowania i aktualizacji hasła [opracowanie własne]

Pierwszy test na rysunku 6.11 sprawdza, czy kod resetowania hasła został poprawnie zweryfikowany. Wysyłane jest żądanie *POST* na ścieżkę */api/auth/verify-reset-code*, zawierające adres e-mail oraz kod resetowania otrzymany w poprzednim kroku. Oczekiwany rezultat to odpowiedź z kodem 200 i komunikatem *Kod weryfikacyjny został zweryfikowany*. Test potwierdza, że proces integracji kontrolera z bazą danych działa zgodnie z oczekiwaniami.

Drugi test na rysunku 6.11 weryfikuje poprawność działania ścieżki API odpowiedzialnej za aktualizację hasła użytkownika. Wysyłane jest żądanie HTTP typu *PUT* na ścieżkę */api/auth/reset-password*, zawierające adres e-mail, kod resetowania oraz nowe hasło. Oczekiwany rezultat to odpowiedź z kodem 200 i komunikatem *Hasło zostało zmienione*. Test sprawdza, czy nowe hasło zostało poprawnie zapisane w bazie danych oraz czy system działa zgodnie z założeniami.



Rys. 6.12 Rezultat testów kontrolera uwierzytelniania [opracowanie własne]

7. Podsumowanie i wnioski

Przedmiotem niniejszej pracy inżynierskiej było zaprojektowanie i implementacja aplikacji webowej do zarządzania rezerwacjami w salonie fryzjerskim. Wszystkie wymagania funkcjonalne, określone w podrozdziałach 2.1.1–2.1.5, oraz niefunkcjonalne, określone w podrozdziałach 2.2.1–2.2.3, zostały pomyślnie zrealizowane i zaimplementowane.

System umożliwia użytkownikom zakładanie konta poprzez rejestrację, gdzie należy podać podstawowe dane, a następnie logowanie za pomocą adresu e-mail i hasła. Fryzjerzy po zalogowaniu się mają dostęp do panelu administracyjnego, który pozwala na zarządzanie harmonogramem pracy i edytowaniem dostępności. Klienci mają możliwość przeglądania dostępnych terminów wizyt w salonie fryzjerskim i rezerwowania ich online, wybierając odpowiednią datę, fryzjera, usługi oraz godzinę. System wyświetla tylko wolne terminy, a po dokonaniu rezerwacji wysyła potwierdzenie oraz przypomnienie o nadchodzącej wizycie. Dodatkowo, po zakończeniu wizyty, klienci mogą ocenić usługę za pomocą systemu gwiazdkowego i dodać szczegółowy komentarz, co pomaga w ocenie jakości usług świadczonych przez salon. Dzięki wdrożeniu aplikacji klienci zyskali możliwość szybkiej i wygodnej rezerwacji usług online, dostępnej przez całą dobę. Rozwiązanie to eliminuje konieczność telefonicznego umawiania wizyt i pozwala na zarezerwowanie terminu w dowolnym momencie, niezależnie od godzin otwarcia salonu. Przeprowadzone testy jednostkowe i integracyjne potwierdziły poprawność działania wszystkich kluczowych funkcji systemu. Aplikacja charakteryzuje się intuicyjnym i prostym interfejsem, jest stabilna oraz zapewnia odpowiednie środki bezpieczeństwa.

Dalszy rozwój aplikacji może obejmować dodanie możliwości usuwania i edytowania opinii, co pozwoli użytkownikom na poprawianie swoich ocen. Możliwe jest także wdrożenie modułu do generowania raportów dotyczących liczby wizyt, najpopularniejszych usług i innych statystyk, które ułatwią zarządzanie salonem. Kolejną funkcjonalnością może być automatyczne sprawdzanie dni świątecznych i oznaczanie salonu jako nieczynnego, aby uniknąć nieporozumień. Istotnym usprawnieniem byłoby także wprowadzenie opcji edycji i usuwania usług, co umożliwi elastyczne zarządzanie ofertą salonu.

Implementacja systemu umożliwiła praktyczne zastosowanie umiejętności nabytych w trakcie studiów oraz poznanie nowych technologii, które zostały zastosowane w niniejszej pracy. Wyzwania napotkane podczas tworzenia aplikacji stanowiły cenne doświadczenie, które przyczyniło się do rozwoju zawodowego.

Bibliografia

[1] Marijn Haverbeke, *Eloquent JavaScript*, 4th Edition, No Starch Press, 2024

[2] Eddy Wilson Iriarte Koroliova, *MERN Quick Start Guide: Build web applications with MongoDB, Express.js, React, and Node.js*, Packt Publishing, 2018

[3] Sandro Pasquali, *Mastering Node.js: Expert techniques for building fast servers and scalable, real-time network applications with minimal effort*, Packt Publishing, 2013

[4] Express.js [Online] *https://expressjs.com/en/api.html*

[5] Shakuntala Gupta Edward, Navin Sabharwal, *Practical MongoDB: Architecting, Developing, and Administering MongoDB, Apress, 2015*

[6] JSON [Online] *https://www.json.org/json-en.html*

[7] JWT [Online] *https://jwt.io/introduction*

[8] Vite.js [Online] *https://vite.dev/guide/*

[9] Mark Massé, *REST API Design Rulebook*, O'Reilly Media, 2012

[10] React.js [Online] *https://react.dev/*

[11] Markus Gärtner, *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*, Addison-Wesley, 2013

Spis rysunków