BAZY DANYCH – PROJEKT

DOKUMENTACJA DO ETAPU 3 I 4

Emilia Kowalczyk 272539

Julia Rojek 272529

SPIS TREŚCI

1 WSTĘP

1.1 Cel dokumentacji

Dokumentacja ma na celu przedstawienie struktury i funkcjonalności strony internetowej służącej do rezerwacji domków. Użytkownicy mogą logować się jako klienci lub właściciele, a także dokonywać rezerwacji przez formularz. System wspiera autoryzację oraz pozwala na edycję danych użytkowników i rezerwacji.

1.2 Opis projektu

Projekt polega na stworzeniu aplikacji webowej umożliwiającej rezerwację domków na pobyt. System będzie składał się z dwóch głównych typów użytkowników:

1. **Klienci** – mogą rezerwować dostępne domki, edytować swoje dane, sprawdzać status rezerwacji oraz je anulować.
2. **Właściciele** – mogą sprawdzać rezerwacje, akceptować je lub odrzucać, edytować swoje dane oraz sprawdzać dostępność swoich domków.

1.3 Technologie użyte w projekcie

Projekt wykorzystuje nowoczesne technologie zarówno po stronie frontendowej, jak i backendowej, zapewniające wydajność, bezpieczeństwo i łatwą integrację. W warstwie backendowej zastosowano **Node.js** z frameworkiem **Express.js** do obsługi API, **MySQL** do przechowywania danych oraz biblioteki takie jak **Multer** (do obsługi plików) i **Bcrypt** (do bezpiecznego haszowania haseł). W warstwie frontendowej użyto **React** do tworzenia dynamicznych interfejsów, **React Router** do nawigacji oraz **Axios** do komunikacji z backendem. Dodatkowo, projekt wspiera bezpieczne przechowywanie danych przy użyciu **Dotenv** i **CORS**.

2 ARCHITERKTURA SYSTEMU

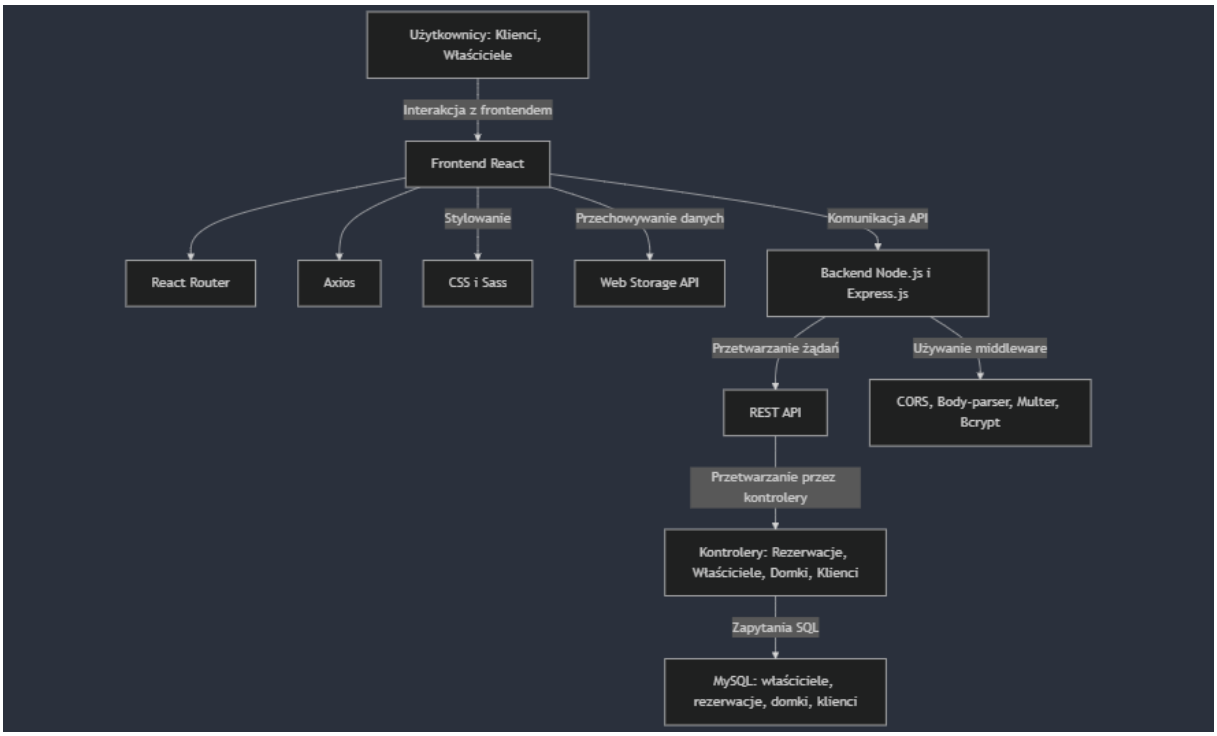
2.1 Ogólny opis architektury

Architektura systemu została zaprojektowana z myślą o wydajności, skalowalności oraz bezpieczeństwie. Aplikacja składa się z trzech głównych warstw: frontend, backend i baza danych, które współpracują ze sobą w celu zapewnienia płynnego działania całego systemu.

- **Frontend** jest odpowiedzialny za interakcję z użytkownikami, zapewniając dynamiczny interfejs w technologii **React**. Aplikacja frontendowa komunikuje się z backendem za pomocą **REST API** realizowanego przez **Node.js** i **Express.js**.
- **Backend** obsługuje logikę biznesową aplikacji, zarządza użytkownikami, rezerwacjami oraz innymi zasobami. Backend jest oparty na **Node.js** i **Express.js**, a dane przechowywane są w relacyjnej bazie danych **MySQL**. Komunikacja z bazą danych odbywa się za pomocą biblioteki **MySQL2**.
- **Baza danych** przechowuje dane użytkowników, rezerwacji, dostępności domków i inne istotne informacje. Baza jest zaprojektowana w modelu relacyjnym, co pozwala na łatwe zarządzanie danymi i ich integrację.

System jest zaprojektowany tak, aby umożliwić łatwą rozbudowę i integrację z dodatkowymi usługami, a także skalowanie w przypadku rosnącego obciążenia.

2.2 Diagram architektury systemu



Rysunek 1: Diagram architektury systemu

3 FRONTEND

3.1 Przegląd technologii frontendowej

Frontend aplikacji został zaprojektowany z wykorzystaniem nowoczesnych technologii webowych, które zapewniają wysoką wydajność, elastyczność oraz łatwość w utrzymaniu. Kluczowe technologie użyte w projekcie to:

- **React** – biblioteka JavaScript do budowy interfejsów użytkownika, wykorzystywana do tworzenia komponentowych aplikacji internetowych. React umożliwia szybkie renderowanie UI dzięki wykorzystaniu Virtual DOM, co zapewnia lepszą wydajność przy minimalnej ilości przeładowań strony. Komponentowe podejście React pozwala na modularne i łatwe zarządzanie kodem.

- **React Router** – biblioteka do zarządzania routingiem w aplikacjach typu Single Page Application (SPA). React Router pozwala na dynamiczną zmianę widoków w aplikacji bez przeładowywania strony, co zapewnia płynne doświadczenie użytkownika i efektywne zarządzanie trasami w obrębie aplikacji.

- **React Context API** – mechanizm służący do zarządzania globalnym stanem aplikacji. React Context umożliwia przekazywanie danych (np. informacji o logowaniu, ustawieniach użytkownika) między komponentami na różnych poziomach hierarchii komponentów bez konieczności przekazywania propsów przez wszystkie pośrednie komponenty. Dzięki temu zarządzanie stanem jest bardziej przejrzyste i efektywne.

- **React Hooks** – funkcje, które pozwalają na korzystanie ze stanu oraz innych funkcji React w komponentach funkcyjnych. Wykorzystanie Hooków, takich jak useState, useEffect, useContext, pozwala na łatwiejsze i bardziej deklaratywne zarządzanie stanem i logiką komponentów. Dzięki Hookom aplikacja jest bardziej zwięzła i czytelna.

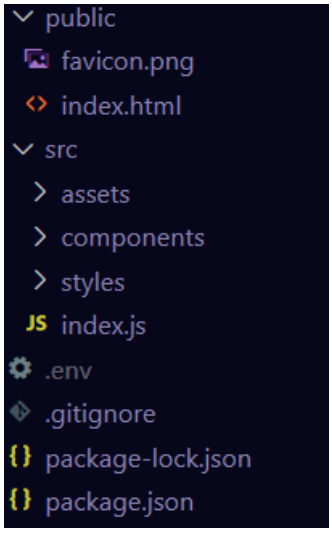
- **Axios** – biblioteka do realizacji zapytań HTTP, wykorzystywana do komunikacji z backendem aplikacji. Dzięki Axios, frontend może wykonywać zapytania GET i POST do serwera, odbierać odpowiedzi w formacie JSON oraz obsługiwać błędy. Axios obsługuje także asynchroniczność zapytań, co poprawia wydajność i responsywność aplikacji.
- **CSS & Sass** – technologie stosowane do stylowania aplikacji. CSS jest używany do podstawowego definiowania stylów, natomiast Sass (preprocesor CSS) zapewnia dodatkowe funkcje, takie jak zmienne, zagnieżdżanie selektorów i dziedziczenie stylów, co poprawia organizację kodu CSS oraz ułatwia jego utrzymanie i rozwój.
- **Web Storage API** – API umożliwiające przechowywanie danych lokalnie w przeglądarce użytkownika. W projekcie wykorzystano Web Storage do przechowywania danych sesji, co pozwala na utrzymanie stanu aplikacji (np. zalogowanego użytkownika) pomiędzy przeładowaniami strony oraz poprawia doświadczenie użytkownika.

Wykorzystanie tych technologii umożliwia budowę wydajnej i skalowalnej aplikacji, zapewniając jednocześnie intuicyjny interfejs i wygodną obsługę danych.

3.2 Struktura katalogów w frontendzie

Struktura katalogów w części frontendowej aplikacji została zaprojektowana z myślą o modularności, łatwości utrzymania i rozwoju. Wszystkie pliki i foldery zostały odpowiednio uporządkowane, co zapewnia przejrzystość kodu oraz wygodne zarządzanie poszczególnymi komponentami i zasobami.

Poniżej przedstawiona jest ogólna struktura katalogów frontendowej części aplikacji:



Rysunek 2: Struktura katalogów frontendu

Wyjaśnienie poszczególnych katalogów:

- **/public** - zawiera statyczne pliki, takie jak główny plik HTML (index.html) oraz zasoby, które nie wymagają przetwarzania przez Webpack (np. czcionki)
- **/src** - główna część aplikacji, zawierająca wszystkie komponenty, strony, logikę aplikacji oraz stylowanie.
    - **/assets** - folder przechowujący zasoby, takie jak obrazy, które są wykorzystywane w różnych miejscach aplikacji.

- **/components** - zawiera komponenty React, które stanowią podstawowe elementy interfejsu użytkownika aplikacji. Folder /pages wewnątrz /components jest podzielony na dwa podfoldery:
  - **/client** - komponenty stron i widoków, które są dostępne tylko dla klientów.
  - **/owner** - komponenty stron i widoków, które są dostępne tylko dla właścicieli domków.
- **/styles** - folder przechowujący pliki stylów. Struktura folderów wewnątrz /styles odpowiada strukturalnie folderowi /components z komponentami, dzięki czemu style są łatwo przypisane do odpowiednich komponentów.

3.3 Opis głównych komponentów

3.3.1 Komponenty wspólne

Komponenty wspólne są używane w różnych częściach aplikacji i odpowiadają za wyświetlanie elementów, które są wspólne dla wszystkich użytkowników (klientów, właścicieli i niezalogowanych). Do takich komponentów należą:

- **Home_page** - Komponent odpowiedzialny za wyświetlanie strony głównej aplikacji, która zawiera sekcje z wyszukiwarką i polecanymi domkami. Jest to pierwsza strona, którą widzą użytkownicy po wejściu na stronę.
- **House** – Komponent służący do wyświetlania szczegółów pojedynczego domku. Zawiera informacje o cenie, dostępności oraz zdjęcia domku.
- **Search_page** - Strona, na której użytkownicy mogą przeszukiwać dostępne domki na podstawie wybranych kryteriów (np. lokalizacja, daty). Komponent obsługuje proces filtracji ofert.
- **App** - Główny komponent aplikacji, który odpowiada za renderowanie wszystkich podstron aplikacji oraz zarządzanie routingiem.

```
function App() {
  return (
    <AuthProvider>
      <Router>
        <div className="App">
          <Header />
          <main>
            <Routes>
              <Route path="/" element={<Home_page />} />
              <Route path="/SignUp" element={<Sign_up />} />
              <Route path="/SignIn" element={<SignIn />} />
              <Route path="/SignInAsOwner" element={<SignInAsOwner />} />
              <Route
                path="/owner/profile"
                element={
                  <Private_owner_route>
                    <Profile />
                  </Private_owner_route>
                }
              />
```

Rysunek 3: Fragment komponentu App

- **Footer** - Komponent wyświetlający stopkę strony, zawierającą informacje ogólneoraz odnośniki do najważniejszych podstron.
- **Home_description** - Komponent prezentujący misję aplikacji.
- **Owner_details** – Komponent służący do wyświetlania opisu oraz danych kontaktowych właściciela. Jest używany na stronie konkretnego domku.
- **Recommended** - Komponent odpowiedzialny za wyświetlanie rekomendowanych domków.
- **Search_form** – Komponent wyszukiwarki z kryteriami wyszukiwania.
- **Search** – Komponent używany na stronie głównej, renderujący Search_form oraz baner ze zdjęciem.

3.3.2 Komponenty dedykowane klientom

Te komponenty są dostępne tylko dla użytkowników posiadających rolę "client". Odpowiadają za interfejs umożliwiający rezerwację domków oraz zarządzanie swoimi danymi.

- **Client_settings** - strona pozwalająca klientowi edytować swoje dane osobowe, takie jak adres e-mail, numer telefonu, czy hasło lub usunąć konto.
- **My_reservation** - strona, na której klient może przeglądać szczegóły swoich dokonanych rezerwacji, sprawdzać status i anulować rezerwacje, jeśli to konieczne.
- **Reservation_form** - Formularz umożliwiający klientowi dokonanie rezerwacji domku. Wysyła dane do backendu w celu zarezerwowania domku.

3.3.3 Komponenty dedykowane właścicielom

Te komponenty są dostępne tylko dla użytkowników posiadających rolę "owner". Odpowiadają za interfejs umożliwiający właścicielom zarządzanie swoimi domkami i rezerwacjami.

- **My_offers** - Komponent wyświetlający listę domków należących do właściciela. Umożliwia podejrzenie szczegółów oferty oraz dostępności domku.
- **Owner_settings** - strona pozwalająca właścicielowi edytować hasło lub usunąć konto.
- **Profile** - Komponent pozwalający na edycję danych osobowych właściciela, jego opisu oraz zdjęcia.
- **Reservation_modal** - Modalne okno wyświetlane po kliknięciu w rezerwację, pozwalające właścicielowi na zatwierdzenie lub odrzucenie rezerwacji klienta.
- **Reservations** - Komponent wyświetlający wszystkie rezerwacje dla właściciela. Renderuje komponenty Confirmed oraz Waiting.
- **Confirmed** - Komponent wyświetlający listę potwierdzonych rezerwacji. Właściciel może je odrzucić.
- **Waiting** - Komponent wyświetlający rezerwacje oczekujące na zatwierdzenie. Właściciel może je zaakceptować lub odrzucić.

3.3.4 Komponenty związane z autoryzacją

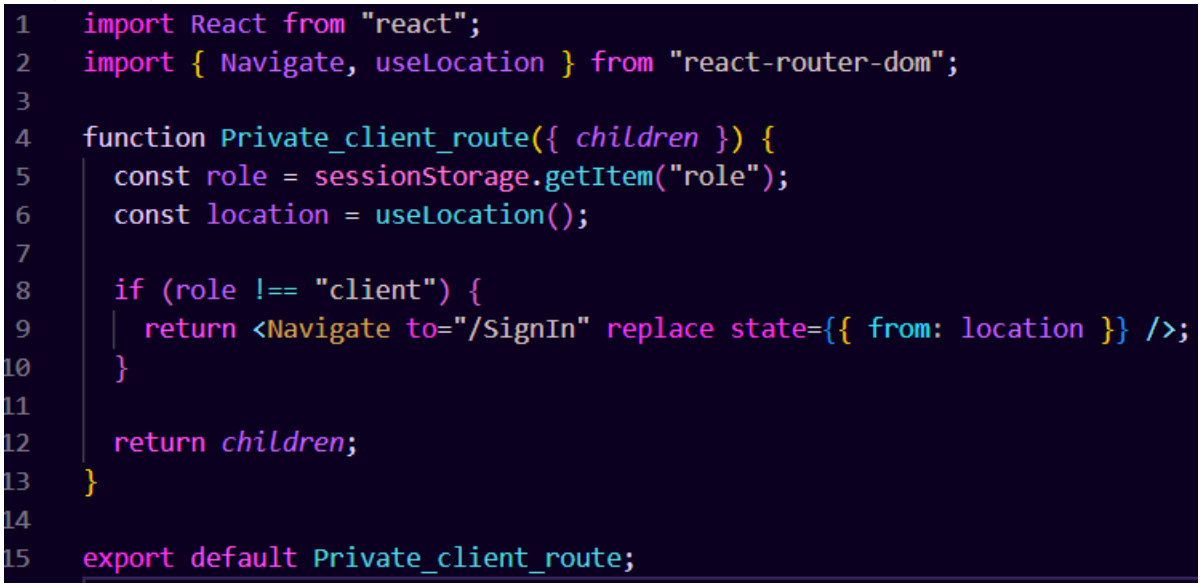
Komponenty związane z logowaniem i rejestracją użytkowników:

- **Sign_in_owners** - Komponent logowania dedykowany właścicielom. Umożliwia właścicielom zalogowanie się do systemu za pomocą loginu i hasła.
- **Sign_in** - Komponent logowania dedykowany klientom. Umożliwia klientom zalogowanie się do systemu za pomocą loginu i hasła.
- **Sign_up** - Komponent rejestracji umożliwiający nowym użytkownikom (klientom) stworzenie konta w systemie.

3.3.5 Komponenty nawigacyjne

Te komponenty są odpowiedzialne za nawigację pomiędzy różnymi stronami aplikacji.

- **Header** - Komponent nagłówka, zawierający logo aplikacji, przyciski nawigacyjne, oraz elementy takie jak przyciski logowania, rejestracji lub wylogowywania.
- **Private_client_route** - Komponent zabezpieczający dostęp do stron tylko dla zalogowanych użytkowników o roli "client". Używany do ochrony prywatnych ścieżek dedykowanych klientom.



```
1   import React from "react";
2   import { Navigate, useLocation } from "react-router-dom";
3
4   function Private_client_route({ children }) {
5     const role = sessionStorage.getItem("role");
6     const location = useLocation();
7
8     if (role !== "client") {
9       return <Navigate to="/SignIn" replace state={{ from: location }} />;
10    }
11
12    return children;
13  }
14
15  export default Private_client_route;
```

Rysunek 4: Fragment komponentu Private_client_route

- **Private_owner_route** - Komponent zabezpieczający dostęp do stron tylko dla zalogowanych użytkowników o roli "owner". Używany do ochrony prywatnych ścieżek dedykowanych właścicielom.

3.3.6 Komponenty zarządzania stanem

W tej sekcji znajdują się komponenty i mechanizmy odpowiedzialne za zarządzanie stanem aplikacji.

- **Auth_context** - jest komponentem odpowiedzialnym za zarządzanie stanem autoryzacji w aplikacji. Używa on kontekstu React (Context API) do przechowywania informacji o bieżącym użytkowniku, jego roli oraz statusie logowania. Zapewnia on centralne miejsce do przechowywania danych dotyczących autentykacji, takich jak dane użytkownika, jego rola ("client" lub "owner"), oraz flagę logowania (isLoggedIn).

```
const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [isLoggedIn, setIsLoggedIn] = useState(
    sessionStorage.getItem("role") !== null
  );
  const [role, setRole] = useState(sessionStorage.getItem("role"));
  const [userData, setUserData] = useState(
    JSON.parse(sessionStorage.getItem("userData")) || {}
  );

  const login = (userRole, data) => {
    setIsLoggedIn(true);
    setRole(userRole);
    sessionStorage.setItem("role", userRole);
    sessionStorage.setItem("userData", JSON.stringify(data));
    setUserData(data);

    // Przechowuj specyficzne identyfikatory
    if (userRole === "client") {
      sessionStorage.setItem("clientId", data.id_klienta);
```

Rysunek 5: Fragment komponentu Auth_context

3.4 Opis funkcji i procesów

W tej sekcji znajdują główne funkcje i procesy, które realizowane są w aplikacji webowej. Każda funkcjonalność systemu ma na celu usprawnienie obsługi rezerwacji domków, zarządzania danymi użytkowników oraz umożliwienie interakcji między klientami i właścicielami. Funkcje zostały podzielone na kilka obszarów, takich jak rejestracja i logowanie, rezerwacja domków, zarządzanie rezerwacjami, oraz obsługa użytkowników.

3.4.1 Logowanie i rejestracja użytkowników

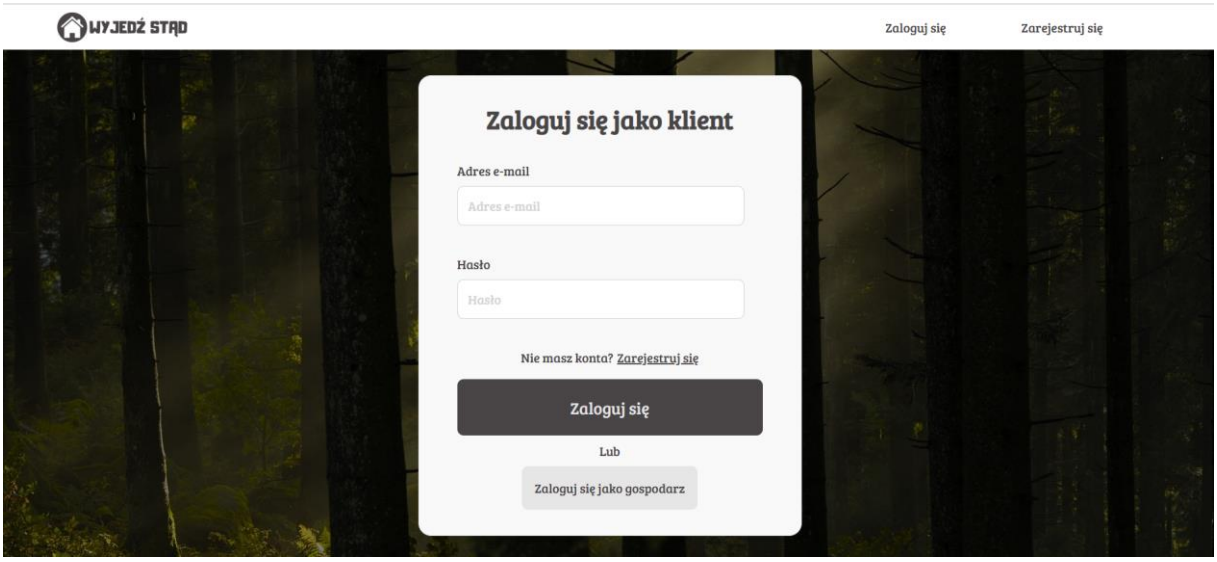
Aplikacja umożliwia użytkownikom rejestrację i logowanie się, przy czym dostępne są dwa typy użytkowników:

- **Klienci** – mogą rejestrować się i logować w celu dokonywania rezerwacji domków.
- **Właściciele** – mogą zarządzać rezerwacjami.

Proces logowania i rejestracji przebiega w następujący sposób:

1. Użytkownik wybiera odpowiednią opcję logowania/rejestracji.
2. W przypadku nowego użytkownika wypełnia formularz rejestracyjny (z podaniem danych takich jak imię, e-mail, hasło).
3. W przypadku już istniejącego użytkownika, podaje dane logowania (e-mail i hasło).
4. Po pomyślnym zalogowaniu użytkownik zostaje przekierowany do strony głównej, gdzie ma dostęp do funkcji zgodnych z jego rolą (klient lub właściciel).

Proces logowania jest obsługiwany za pomocą komponentu AuthContext, który zarządza stanem logowania i przechowuje dane w sessionStorage.



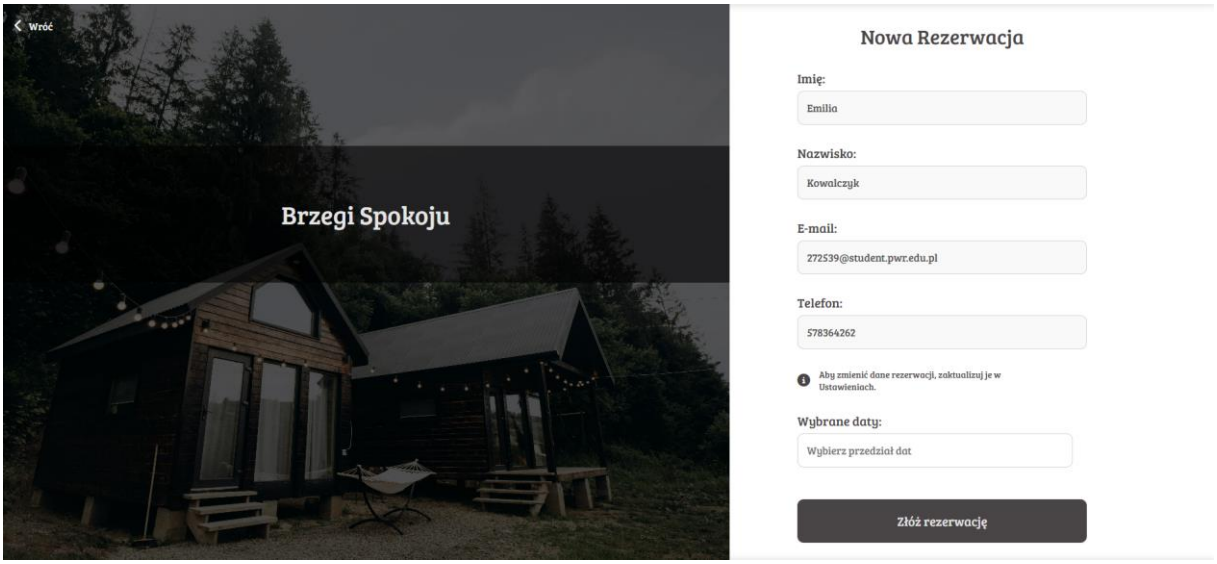
Rysunek 6: Formularz logowania dla klientów

3.4.2 Rezerwacja domków przez klientów

Klienci mogą rezerwować dostępne domki za pomocą formularza rezerwacji pojawiającego się po wyborze domku, który umożliwia:

1. **Wybór terminu** – Klient wybiera datę przyjazdu oraz datę wyjazdu.
2. **Składanie rezerwacji** – Po dokonaniu wyboru klient potwierdza rezerwację.

Funkcjonalność rezerwacji jest realizowana w komponencie Reservation_form. Po dokonaniu rezerwacji klient otrzymuje szczegóły rezerwacji w swojej sekcji „Moje rezerwacje”.



Rysunek 7: Formularz rezerwacji

3.4.3 Wyszukiwanie domków

W aplikacji użytkownicy mogą wyszukiwać dostępne domki na podstawie różnych kryteriów, takich jak:

- **Termin pobytu** – Wyszukiwanie według daty przyjazdu i wyjazdu.
- **Lokalizacja** – Wybór miasta.

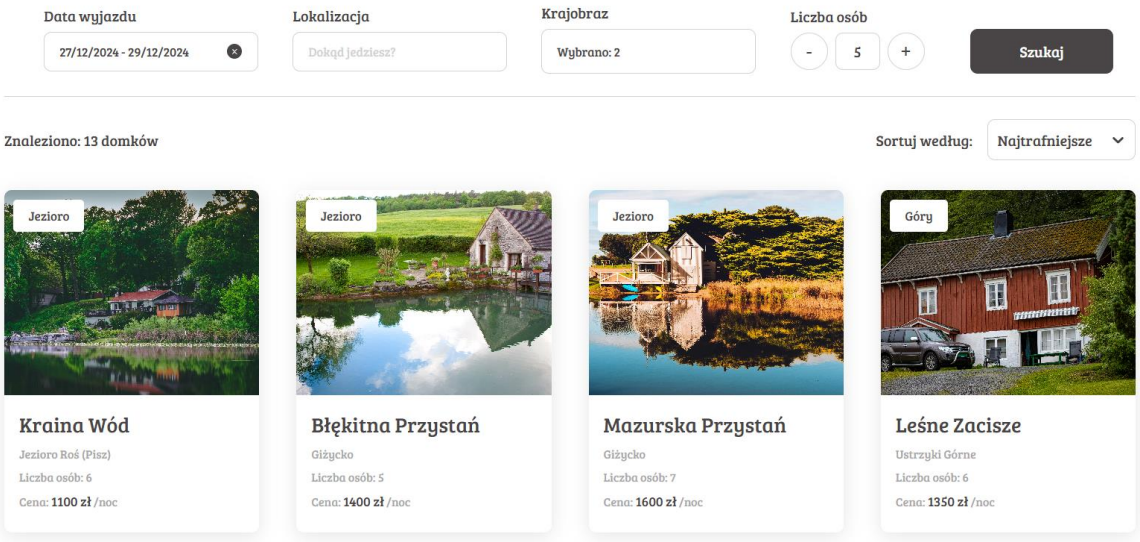
- **Kategoria** – Wybór krajobrazu.
- **Liczba osób** – Wybór maksymalnej pojemności domku.

Proces wyszukiwania jest realizowany przy użyciu komponentu **Search_form**, który umożliwia klientowi wybór odpowiednich filtrów. Po przesłaniu formularza użytkownik otrzymuje listę dostępnych domków spełniających zadane kryteria. Wyszukiwanie jest dynamiczne, z wynikami aktualizowanymi w czasie rzeczywistym. Wyniki można przefiltrować według ceny oraz polecanych.

Komponenty odpowiedzialne za wyszukiwanie:

- **Search_form**: Formularz umożliwiający wprowadzenie kryteriów wyszukiwania.
- **Search_page**: Strona z wynikami wyszukiwania, wyświetlająca dostępne domki.

Po dokonaniu wyboru domku użytkownik może przejść do formularza rezerwacji, gdzie wpisuje szczegóły dotyczące pobytu.



Rysunek 8: Strona wyszukiwarki domków

3.4.4 Obsługa rezerwacji przez właścicieli

Właściciele zarządzają rezerwacjami swoich domków, mając dostęp do:

1. **Przeglądu wszystkich rezerwacji** – Lista rezerwacji z podziałem na statusy
2. **Zatwierdzania lub odrzucania rezerwacji** – Właściciel decyduje, czy przyjąć rezerwację klienta.
3. **Przegląd szczegółów rezerwacji** – Właściciel może wyświetlić szczegółowe informacje dotyczące klienta oraz rezerwacji.

Komponenty obsługujące rezerwacje właściciela:

- **Reservations**: Lista rezerwacji renderująca kolejne komponenty.
- **Confirmed**: Widok potwierdzonych rezerwacji.
- **Waiting**: Widok rezerwacji oczekujących na decyzję właściciela.

- **Reservation_modal**: Modal wyświetlający szczegóły konkretnej rezerwacji oraz umożliwiający jej akceptację lub odrzucenie.

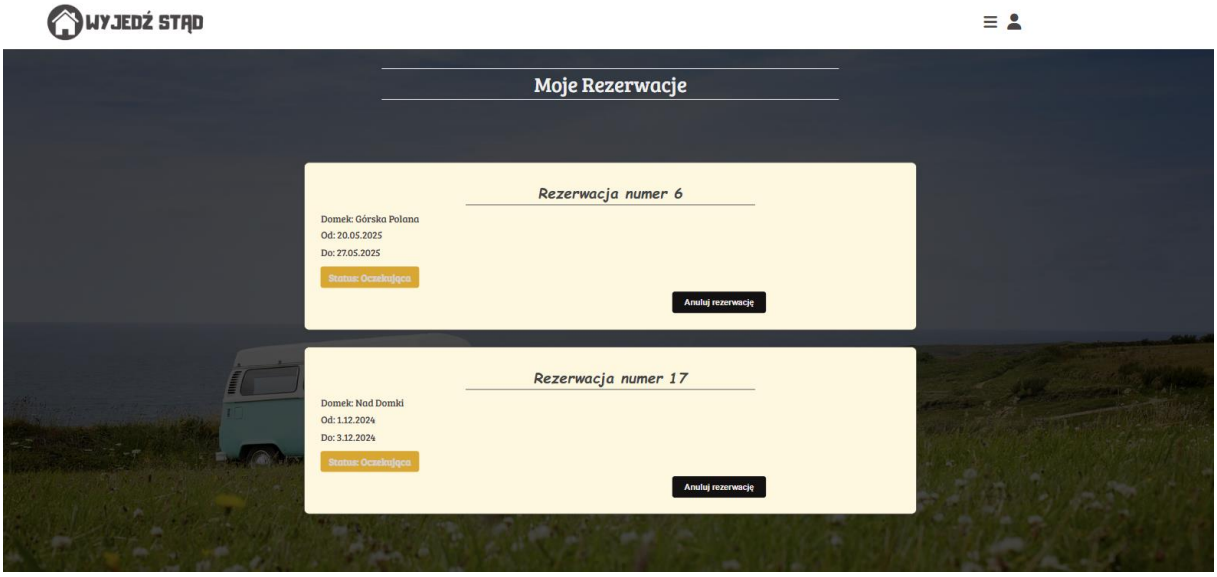
Rysunek 9: Strona z rezerwacjami dla właściciela

3.4.5 zarządzanie rezerwacjami przez klienta

Proces zarządzania rezerwacjami przez klienta obejmuje:

1. Przegląd rezerwacji w sekcji „Moje rezerwacje”.
2. Możliwość anulowania rezerwacji.

Zarządzanie rezerwacjami odbywa się w ścisłej współpracy między frontendem (React) a backendem (Node.js, MySQL), gdzie wszystkie operacje na danych są synchronizowane w bazie danych.



Rysunek 10: Strona z rezerwacjami dla klienta

3.4.6 Zarządzanie danymi użytkowników

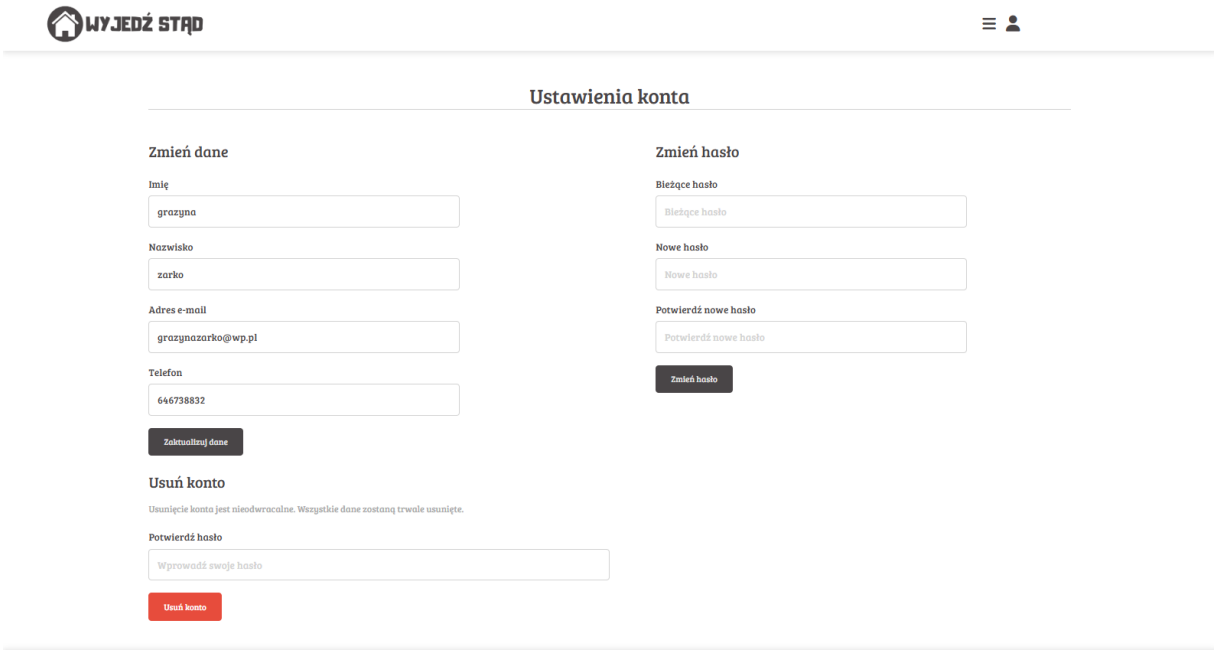
Każdy użytkownik może zarządzać swoimi danymi osobowymi:

- **Edycja profilu** – Zmiana danych osobowych, takich jak e-mail, numer telefonu czy hasło.

- **Podgląd danych** – Klienci mogą przeglądać historię swoich rezerwacji, a właściciele informacje o swoich domkach.

Komponenty do zarządzania danymi:

- **Client_settings** – Edycja danych klienta.
- **Owner_settings i Profile** – Edycja danych właściciela.



Rysunek 11: Ustawienia klienta

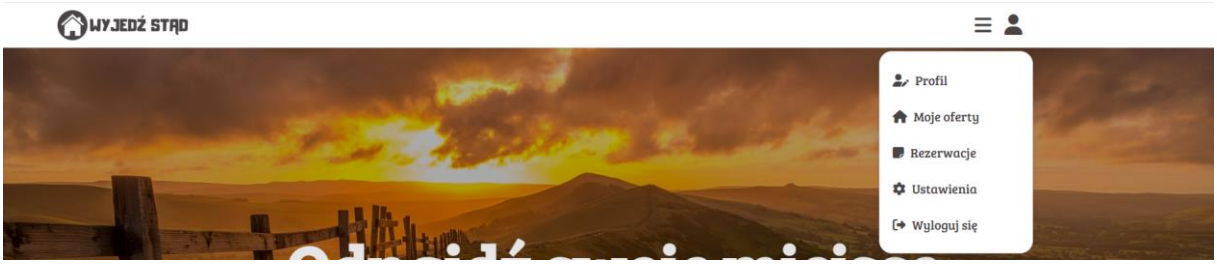
3.4.7 Nawigacja po aplikacji

Aplikacja wykorzystuje **React Router**, który pozwala na płynną nawigację między stronami:

- Klienci mają dostęp do podstron związanych z rezerwacjami i edycją danych.
- Właściciele mogą przeglądać swoje oferty i zarządzać rezerwacjami.

Komponenty nawigacyjne:

- **Header**: Pasek nawigacyjny z linkami do głównych sekcji aplikacji.
- **Private_client_route**: Zapewnia dostęp do tras dedykowanych klientom.
- **Private_owner_route**: Zapewnia dostęp do tras dedykowanych właścicielom.



Rysunek 12: Header dla zalogowanego właściciela

3.4.8 Przechowywanie stanu aplikacji

Dane globalne aplikacji są zarządzane za pomocą AuthContext oraz React Context API. Używane są do przechowywania:

- Stanu logowania (isLoggedIn).
- Danych użytkownika (rola, dane osobowe).

Dane są przechowywane w pamięci przeglądarki (sessionStorage) w celu zwiększenia spójności między stronami.

3.4.9 Bezpieczeństwo aplikacji

Aplikacja chroni dane użytkowników oraz zasoby systemu dzięki:

- **Autoryzacji** – Dostęp do określonych stron zależy od roli użytkownika (klient lub właściciel).
- **Szyfrowaniu haseł** – Wszystkie hasła są haszowane przed zapisaniem w bazie danych.
- **Middleware backendowe** – Mechanizmy ochrony przed atakami SQL Injection i XSS.

3.5 Implementacja routingu

Routing w aplikacji jest odpowiedzialny za zarządzanie nawigacją pomiędzy różnymi widokami i stronami aplikacji, umożliwiając użytkownikom przechodzenie pomiędzy nimi bez przeładowania strony. W aplikacji wykorzystujemy bibliotekę **React Router**, która pozwala na obsługę routingu w aplikacjach typu SPA (Single Page Application).

3.5.1 Konfiguracja Routingu

W pliku App.js zaimplementowano konfigurację routingu, w której wykorzystano **BrowserRouter** jako kontener do zarządzania trasami. Wszystkie trasy zostały zdefiniowane wewnątrz komponentu <Routes>, co pozwala na przypisanie odpowiednich komponentów do poszczególnych ścieżek URL.

```
function App() {
  return (
    <AuthProvider>
      <Router>
        <div className="App">
          <Header />
          <main>
            <Routes>···
            </Routes>
          </main>
          <Footer />
        </div>
      </Router>
    </AuthProvider>
  );
}

export default App;
```

Rysunek 13: Implementacja routingu

3.5.2 Zastosowanie tras chronionych

W aplikacji zaimplementowane są **trasy chronione** (private routes), które dostępne są tylko dla zalogowanych użytkowników o określonej roli. W tym celu wykorzystano komponenty Private_client_route oraz Private_owner_route, które sprawdzają, czy użytkownik jest zalogowany i posiada odpowiednią rolę przed przekierowaniem go na stronę. Jeśli warunki nie są spełnione, użytkownik zostaje przekierowany na stronę logowania.

3.5.3 Nawigacja pomiędzy stronami

Do nawigacji pomiędzy stronami aplikacji wykorzystywane są komponenty **Link, a** oraz **NavLink**. Dzięki nim użytkownicy mogą przechodzić do innych sekcji aplikacji, takich jak strona logowania, rejestracji czy podglądu oferty właściciela.

3.5.4 Obsługa dynamicznych tras

Aplikacja obsługuje również dynamiczne ścieżki URL, dzięki czemu użytkownicy mogą przechodzić do konkretnych ofert domków, korzystając z unikalnych identyfikatorów. Przykład: /houses/:id_domku pozwala na wyświetlenie szczegółów domku na podstawie przekazanego identyfikatora.

```
<Route path="/houses/:id_domku" element={<House />} />
<Route
  path="/rezerwacja/:id_domku"
  element={
    <Private_client_route>
      <Reservation_form />
    </Private_client_route>
  }
/>
```

Rysunek 14: Przykład dynamicznej trasy

3.6 Komunikacja z backendem

Komunikacja z backendem w aplikacji jest realizowana przy użyciu metod fetch oraz axios, które umożliwiają wysyłanie zapytań HTTP (GET, POST, PATCH) i odbieranie odpowiedzi z serwera.

- **fetch** – wykorzystywana głównie do prostych zapytań, np. pobierania lub wysyłania danych w formacie JSON. Zapewnia pełną kontrolę nad zapytaniem i odpowiedzią.

```
fetch("http://localhost:5000/api/search", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(searchParams),
})
  .then((response) => response.json())
  .then((data) => {
    if (props.onSearchResults) {
      props.onSearchResults(data, true, searchParams);
    }
  })
  .catch((error) => {
    console.error(
      "Błąd podczas wysyłania zapytania wyszukiwania:",
      error
    );
  });
```

Rysunek 15: Przykład użycia fetch

- **axios** - używane do bardziej złożonych zapytań, zapewnia prostszą obsługę błędów i automatyczną konwersję odpowiedzi na JSON.

```
try {
  const waitingRes = await axios.get(
    `http://localhost:5000/api/owner/${ownerId}/waiting-reservations`
  );
  setWaitingReservations(waitingRes.data);

  const confirmedRes = await axios.get(
    `http://localhost:5000/api/owner/${ownerId}/confirmed-reservation
  );
  setConfirmedReservations(confirmedRes.data);
} catch (error) {
  console.error("Błąd podczas pobierania rezerwacji:", error);
}
```

Rysunek 16: Przykład użycia axios

Aplikacja umożliwia zarówno wysyłanie danych (POST, PATCH), jak i ich pobieranie (GET) z backendu, z wykorzystaniem odpowiednich metod HTTP. Komunikacja opiera się na danych JSON i zapewnia efektywne przesyłanie informacji między frontendem a serwerem.

4 BACKEND

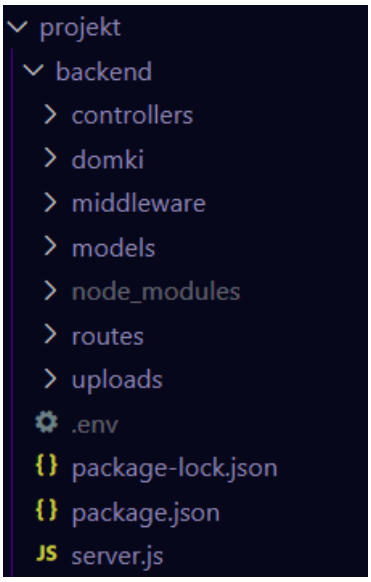
4.1 Przegląd technologii backendowej

- Backend aplikacji został zbudowany przy użyciu nowoczesnych technologii, które zapewniają wydajność, bezpieczeństwo i łatwą integrację z frontendem. Oto szczegółowy opis zastosowanych narzędzi:

- **Node.js**
  Node.js to środowisko uruchomieniowe JavaScript, które umożliwia budowanie skalowalnych i wydajnych aplikacji serwerowych. Jego architektura oparta na zdarzeniach i asynchronicznym modelu obsługi pozwala na efektywne zarządzanie dużą liczbą jednoczesnych połączeń.
- **Express.js**
  Express.js to framework webowy oparty na Node.js, który upraszcza tworzenie API i zarządzanie trasami w aplikacji. Dzięki Express.js możliwe jest szybkie tworzenie endpointów, obsługa middleware i zarządzanie żądaniami HTTP.
- **MySQL**
  MySQL to relacyjna baza danych używana do przechowywania kluczowych danych aplikacji, takich jak użytkownicy, rezerwacje i informacje o domkach. Dzięki wsparciu dla języka SQL, MySQL zapewnia wydajność i niezawodność w operacjach na dużych zbiorach danych.
- **MySQL2**
  MySQL2 to biblioteka Node.js, która umożliwia łączenie się z bazą danych MySQL. W projekcie odpowiada za nawiązywanie połączeń, wykonywanie zapytań SQL oraz ich optymalizację, zwłaszcza w kontekście operacji asynchronicznych.
- **Multer**
  Multer to middleware do obsługi przesyłania plików w aplikacjach Express. W projekcie Multer jest używany do przesyłania i walidacji zdjęć użytkowników oraz domków, co pozwala na ich bezpieczne przechowywanie.
- **CORS (Cross-Origin Resource Sharing)**
  Middleware CORS umożliwia kontrolowanie dostępu do API z różnych domen. Jest to niezbędne w sytuacji, gdy frontend i backend działają na różnych portach lub domenach, zapewniając bezpieczną komunikację między nimi.
- **Body-parser**
  Body-parser to biblioteka, która umożliwia parsowanie treści żądań HTTP na obiekt JSON. Jest to szczególnie ważne w przypadku obsługi żądań POST i PUT, gdzie dane wysyłane przez klienta muszą być interpretowane przez serwer.
- **Bcrypt**
  Bcrypt to biblioteka służąca do bezpiecznego haszowania haseł. W aplikacji wszystkie hasła użytkowników są zaszyfrowane przed zapisaniem w bazie danych, co zwiększa bezpieczeństwo i chroni dane przed nieautoryzowanym dostępem.
- **Dotenv**
  Dotenv to narzędzie do zarządzania zmiennymi środowiskowymi, które umożliwia przechowywanie poufnych danych (np. poświadczeń bazy danych) w pliku .env. Dzięki temu konfiguracja aplikacji jest bardziej bezpieczna i elastyczna.
- **Path**
  Path to moduł wbudowany w Node.js, który umożliwia zarządzanie ścieżkami do plików i katalogów w sposób niezależny od systemu operacyjnego. Jest wykorzystywany do obsługi zasobów statycznych, takich jak obrazy czy pliki.

4.2 Struktura katalogów w backendzie

Struktura katalogów w backendzie została zaprojektowana w sposób, który umożliwia łatwą nawigację, utrzymanie porządku w kodzie oraz skalowalność projektu. Poniżej ogólny układ katalogów w aplikacji:



Rysunek 17: Struktura katalogów backendu

Wyjaśnienie poszczególnych katalogów:

- **/controllers** - Zawiera pliki odpowiedzialne za logikę biznesową aplikacji. Każdy kontroler obsługuje różne operacje, takie jak dodawanie użytkowników, tworzenie rezerwacji, edytowanie danych, itp. Kontrolery współpracują z modelem i odpowiadają za przetwarzanie danych i generowanie odpowiedzi.
- **/domki** - Katalog zawierający zdjęcia dla poszczególnych domków. Każdy domek ma swój osobny podfolder, w którym przechowywane są zdjęcia związane z danym obiektem.
- **/middleware** - Znajduje się tutaj plik upload.js, który jest odpowiedzialny za obsługę przesyłania plików – umożliwia zmianę zdjęcia właściciela domku. Middleware zapewnia, że plik jest poprawnie przesyłany i zapisany w odpowiednim miejscu w systemie plików.
- **/models** - Zawiera pliki odpowiedzialne za interakcję z bazą danych. W pliku db.js znajduje się konfiguracja połączenia z bazą danych.
- **/routes** - W tym folderze znajdują się pliki odpowiedzialne za definicję tras API. Trasy wskazują, które kontrolery mają być wywołane w odpowiedzi na konkretne żądanie HTTP (np. POST, GET, PUT, DELETE). Każdy plik odpowiada za różne aspekty aplikacji).
- **/uploads** - Folder przechowujący zdjęcia właścicieli domków. Pliki są zapisywane w tym katalogu, skąd mogą być później pobierane przez frontend.

4.3 Opis API i dostępnych endpointów

Wszystkie dostępne endpointy zostały zorganizowane w odpowiednich trasach (routes) i podzielone na sekcje, które odpowiadają za różne funkcjonalności aplikacji. Każda z sekcji posiada swój własny zestaw endpointów, które umożliwiają użytkownikowi interakcję z systemem.

API aplikacji jest zorganizowane w kilka grup tras, odpowiadających różnym funkcjonalnościom:

- **Endpointy związane z autentykacją (authRoutes)**
  - **POST /api/auth/register** – Rejestracja nowego użytkownika (klienta).
    - Body: { imie, nazwisko, telefon, email, haslo }
    - Opis: Rejestracja użytkownika po walidacji danych, sprawdzeniu unikalności adresu e-mail oraz zaszyfrowaniu hasła.

        - Statusy: 201 – sukces, 400 – brak wymaganych danych, 409 – istniejące konto, 500 – błąd serwera.
    - **POST /api/auth/login** – Logowanie klienta.
        - Body: { email, haslo }
        - Opis: Logowanie klienta na podstawie podanych danych. Weryfikacja hasła przy użyciu bcrypt.
        - Statusy: 200 – sukces, 400 – brak danych, 401 – nieprawidłowe dane, 500 – błąd serwera.
    - **POST /api/auth/loginOwner** – Logowanie właściciela.
        - Body: { email, haslo }
        - Opis: Logowanie właściciela, analogicznie jak dla klienta.
        - Statusy: 200 – sukces, 400 – brak danych, 401 – nieprawidłowe dane, 500 – błąd serwera.
    - **PUT /api/auth/changeOwnerPassword** – Zmiana hasła właściciela.
        - Body: { ownerId, currentPassword, newPassword }
        - Opis: Zmiana hasła po autoryzacji obecnego hasła właściciela.
        - Statusy: 200 – sukces, 400 – brak danych, 401 – błędne hasło, 404 – właściciel nie znaleziony, 500 – błąd serwera.
- **Endpointy związane z klientem (clientRoutes)**
    - **GET /api/clients/:id** – Pobranie danych klienta.
        - Opis: Zwraca dane klienta na podstawie jego unikalnego ID.
        - Statusy: 200 – sukces, 404 – klient nie znaleziony, 500 – błąd serwera.
    - **GET /api/clients/:id/reservations** – Pobranie rezerwacji klienta.
        - Opis: Zwraca wszystkie rezerwacje klienta, łącznie z nazwą domku i jego zdjęciem.
        - Statusy: 200 – sukces, 500 – błąd serwera.
    - **PUT /api/clients/:id/password** – Zmiana hasła klienta.
        - Body: { currentPassword, newPassword }
        - Opis: Umożliwia klientowi zmianę hasła po weryfikacji bieżącego hasła.
        - Statusy: 200 – sukces, 400 – brak danych, 401 – błędne hasło, 404 – klient nie znaleziony, 500 – błąd serwera.
    - **PUT /api/clients/:id** – Aktualizacja danych klienta.
        - Body: { imie, nazwisko, email, telefon }
        - Opis: Umożliwia klientowi aktualizację swoich danych osobowych.
        - Statusy: 200 – sukces, 400 – brak danych, 500 – błąd serwera.
    - **DELETE /api/clients/:id** – Usunięcie konta klienta.
        - Body: { password }
        - Opis: Umożliwia usunięcie konta klienta po weryfikacji hasła.
        - Statusy: 200 – sukces, 400 – brak danych, 401 – błędne hasło, 404 – klient nie znaleziony, 500 – błąd serwera.
- **Endpointy związane z właścicielami (ownerRoutes)**
    - **GET /api/owners/:id** – Pobranie danych właściciela.
        - Opis: Zwraca dane właściciela na podstawie jego unikalnego ID.
        - Statusy: 200 – sukces, 404 – właściciel nie znaleziony, 500 – błąd serwera.
    - **PUT /api/owners/:id/description** – Aktualizacja opisu właściciela.
        - Body: { opis }
        - Opis: Umożliwia właścicielowi aktualizację opisu swojego profilu.
        - Statusy: 200 – sukces, 400 – brak danych, 500 – błąd serwera.
    - **PUT /api/owners/:id/data** – Aktualizacja danych właściciela.

        - Body: { imie, nazwisko, telefon, email }
        - Opis: Umożliwia właścicielowi aktualizację swoich danych osobowych.
        - Statusy: 200 – sukces, 400 – brak danych, 500 – błąd serwera.
    - **PUT /api/owners/:id/photo** – Aktualizacja zdjęcia właściciela.
        - Body: { zdjecie } – nowy plik zdjęcia.
        - Opis: Umożliwia właścicielowi zaktualizowanie swojego zdjęcia profilowego.
        - Statusy: 200 – sukces, 400 – brak danych, 500 – błąd serwera.
    - **GET /api/owners/:ownerId/houses** – Pobranie ofert domków dla właściciela.
        - Opis: Zwraca listę domków należących do danego właściciela na podstawie jego ID.
        - Statusy: 200 – sukces, 500 – błąd serwera.
    - **GET /api/owners/:ownerId/reservations/waiting** – Pobranie rezerwacji oczekujących dla właściciela.
        - Opis: Zwraca rezerwacje, które mają status "Oczekująca" dla domków należących do właściciela.
        - Statusy: 200 – sukces, 500 – błąd serwera.
    - **GET /api/owners/:ownerId/reservations/confirmed** – Pobranie rezerwacji potwierdzonych dla właściciela.
        - Opis: Zwraca rezerwacje o statusie "Potwierdzona" dla domków należących do właściciela.
        - Statusy: 200 – sukces, 500 – błąd serwera.
    - **PUT /api/owners/reservations/:id/confirm** – Potwierdzenie rezerwacji.
        - Opis: Potwierdza rezerwację, zmieniając jej status na "Potwierdzona".
        - Statusy: 200 – sukces, 404 – rezerwacja nie znaleziona, 500 – błąd serwera.
    - **PUT /api/owners/reservations/:id/reject** – Odrzucenie rezerwacji.
        - Opis: Odrzuca rezerwację, zmieniając jej status na "Odrzucona".
        - Statusy: 200 – sukces, 404 – rezerwacja nie znaleziona, 500 – błąd serwera.
    - **DELETE /api/owners/:ownerId** – Usunięcie konta właściciela.
        - Body: { password }
        - Opis: Usunięcie konta właściciela po weryfikacji hasła.
        - Statusy: 200 – sukces, 400 – brak danych, 401 – błędne hasło, 404 – właściciel nie znaleziony, 500 – błąd serwera.
- **Endpointy związane z polecanymi domkami (recommendedHousesRoutes)**
    - **GET /api/houses/recommended** – Pobranie polecanych domków.
        - Opis: Zwraca listę domków, które zostały oznaczone jako "polecane" (z flagą polecany = true).
        - Statusy: 200 – sukces, 500 – błąd serwera.
- **Endpointy związane z rezerwacjami (reservationController)**
    - **POST /api/reservations**
        - Body: { id_domku, id_klienta, start, end }
        - Opis: Tworzenie nowej rezerwacji dla danego domku przez klienta.
        - Statusy: 201 – Rezerwacja została pomyślnie złożona, 400 – Błąd walidacji (brak danych lub nieprawidłowe daty), 500 – Błąd serwera.
    - **GET /api/houses/:id/reservations**
        - Opis: Zwraca wszystkie rezerwacje dla danego domku. Wymaga podania ID domku w parametrze URL.

        - Statusy: 200 – Pomyślnie zwrócono rezerwacje, 500 – Błąd serwera.
    - **DELETE /api/reservations/:id_rezerwacji**
        - Opis: Usuwa rezerwację o podanym ID. Przed jej usunięciem system sprawdza, czy rezerwacja istnieje w bazie danych.
        - Statusy: 200 – Rezerwacja została anulowana, 404 – Rezerwacja o podanym ID nie istnieje, 500 – Błąd serwera.
- **Endpointy związane z wyszukiwaniem domków (searchController)**
    - **POST /api/search**
        - Body: { dateRange, location, guests, categories, sort, page, limit }
        - Opis: Umożliwia wyszukiwanie dostępnych domków na podstawie różnych filtrów. Klient może określić zakres dat, lokalizację, liczbę gości, preferowane kategorie domków, sposób sortowania wyników oraz numer strony i limit wyników na stronę.
        - Statusy: 200 – Pomyślnie zwrócono wyniki wyszukiwania, 500 – Błąd serwera.

Serwer obsługuje przesyłanie plików (np. zdjęć) za pomocą multer, z walidacją, która zapewnia obsługę tylko obrazów. Endpointy są zabezpieczone CORS i obsługują błędy przesyłania plików.

4.4 Mechanizm autentykacji

Mechanizm autentyfikacji w aplikacji zapewnia bezpieczne zarządzanie dostępem użytkowników oraz ich sesjami. Składa się on z kilku komponentów, które współpracują ze sobą, aby umożliwić rejestrację, logowanie oraz zarządzanie sesjami użytkowników. W skład mechanizmu wchodzą następujące elementy:

- **Kroki logowania i rejestracji**
    - Rejestracja i logowanie: Użytkownicy mogą się zarejestrować lub zalogować do systemu. Podczas rejestracji użytkownik podaje dane takie jak imię, nazwisko, telefon, email i hasło. Hasło jest następnie haszowane przy użyciu algorytmu bcrypt, co zapewnia jego bezpieczeństwo w bazie danych. W przypadku logowania użytkownik podaje swój email i hasło, które są porównywane z danymi zapisanymi w bazie.
    - Walidacja: Przed zapisaniem danych do bazy oraz przed logowaniem, system przeprowadza walidację danych. Na przykład, sprawdzany jest unikalny email podczas rejestracji, a w przypadku logowania weryfikowana jest poprawność wprowadzonego hasła za pomocą funkcji bcrypt.compare.
- **Przechowywanie danych użytkownika**
    - AuthContext: Autentyfikacja jest zarządzana za pomocą kontekstu w React (AuthContext). Przechowuje on dane użytkownika w sesji aplikacji, w tym rolę (np. klient lub właściciel), identyfikatory oraz inne dane użytkownika. Dzięki temu komponenty mogą dynamicznie dostosować swoje zachowanie w zależności od zalogowanego użytkownika i jego roli.
    - sessionStorage: Po zalogowaniu, dane o użytkowniku, takie jak identyfikator, rola oraz dane osobowe, są przechowywane w sessionStorage. Użycie sessionStorage zapewnia, że dane będą dostępne tylko podczas trwania sesji przeglądarki. Po jej zamknięciu, dane są automatycznie usuwane, co zapobiega nieautoryzowanemu dostępowi do informacji użytkownika po zakończeniu sesji.
- **Zarządzanie sesjami**
    - Sesja użytkownika: Po udanym logowaniu, sesja użytkownika jest aktywowana, a jego dane są przechowywane w kontekście aplikacji (AuthContext) oraz w

sessionStorage. Każda aplikacja sprawdza, czy użytkownik jest zalogowany na podstawie dostępnych informacji w sessionStorage. Dzięki temu użytkownik nie musi ponownie wprowadzać swoich danych po przeładowaniu strony, o ile sesja nie wygasła.
    - Wylogowanie: Aby użytkownik mógł się wylogować, wystarczy, że kliknie przycisk wylogowania, co powoduje usunięcie danych użytkownika z kontekstu aplikacji oraz z sessionStorage. Tym samym sesja jest zakończona, a użytkownik zostaje przekierowany na stronę logowania.
- **Bezpieczeństwo**
    - Haszowanie haseł: Wszystkie hasła przechowywane w bazie danych są haszowane przy użyciu algorytmu bcrypt. To zapewnia, że nawet w przypadku wycieku bazy danych, rzeczywiste hasła użytkowników nie będą dostępne, ponieważ są przechowywane w postaci nieodwracalnej.
    - Zarządzanie dostępem: Na podstawie roli użytkownika (np. klienta lub właściciela) aplikacja kontroluje, do jakich zasobów ma dostęp dany użytkownik. Informacje o roli użytkownika są przechowywane w sessionStorage, dzięki czemu można dynamicznie zarządzać dostępem do różnych części aplikacji.
- **Błędy i odpowiedzi serwera**
    - Błędy logowania i rejestracji: W przypadku niepoprawnych danych logowania (np. błędnego hasła lub adresu e-mail) serwer zwraca odpowiedni błąd, np. 401 (Unauthorized) lub 400 (Bad Request). Również przy rejestracji, jeśli dane są niepełne lub email jest już zajęty, zwrócony zostaje błąd, np. 409 (Conflict).
- **Rola AuthContext w zarządzaniu stanem**
    - AuthContext pełni kluczową rolę w przechowywaniu i zarządzaniu stanem autentyfikacji w aplikacji. Działa na poziomie komponentów aplikacji React, umożliwiając dostęp do danych o użytkowniku w różnych częściach interfejsu użytkownika. Dzięki temu system może dynamicznie decydować, które komponenty mają być widoczne, w zależności od roli użytkownika.

5 BAZA DANYCH

5.1 Opis bazy danych

Baza danych aplikacji została zaprojektowana w oparciu o model relacyjny, co pozwala na przechowywanie i zarządzanie danymi w sposób wydajny, zorganizowany i bezpieczny. Do zarządzania danymi wykorzystano **MySQL**, które umożliwia obsługę złożonych relacji oraz zapewnia wsparcie dla języka SQL.

5.2 Struktura tabel

Poniżej znajduje się szczegółowy opis głównych tabel bazy danych aplikacji:

- Tabela **klienci** - Przechowuje dane użytkowników aplikacji o roli klienta. Zawiera kolumny:
    - id_klienta (INT, PRIMARY KEY, AUTO_INCREMENT, NOT NULL) – Unikalny identyfikator klienta.
    - imie (VARCHAR, NOT NULL) – Imię klienta.
    - nazwisko (VARCHAR, NOT NULL) – Nazwisko klienta.
    - email (VARCHAR, UNIQUE, NOT NULL) – Adres e-mail klienta.
    - telefon (VARCHAR, NOT NULL) – Numer telefonu klienta.
    - haslo (VARCHAR, NOT NULL) – Hasło w formie zaszyfrowanej.
    - data_rejestracji (DATE, NOT NULL) – Data rejestracji użytkownika.

- Tabela **wlasciciele** - Przechowuje dane użytkowników aplikacji o roli właściciela. Zawiera kolumny:
    - id_wlasciciela (INT, PRIMARY KEY, AUTO_INCREMENT, NOT NULL, UNSIGNED) – Unikalny identyfikator właściciela.
    - imie (VARCHAR, NOT NULL) – Imię właściciela.
    - nazwisko (VARCHAR, NOT NULL) – Nazwisko właściciela.
    - email (VARCHAR, UNIQUE, NOT NULL) – Adres e-mail właściciela.
    - telefon (VARCHAR, NOT NULL) – Numer telefonu właściciela.
    - haslo (VARCHAR, NOT NULL) – Hasło w formie zaszyfrowanej.
    - opis (TEXT) – Opis właściciela.
    - zdjecie (VARCHAR) – Ścieżka do zdjęcia właściciela.
- Tabela **domki** - Przechowuje informacje o dostępnych domkach. Zawiera kolumny:
    - id_domku (INT, PRIMARY KEY, AUTO_INCREMENT, NOT NULL, UNSIGNED) – Unikalny identyfikator domku.
    - nazwa (VARCHAR, NOT NULL) – Nazwa domku.
    - opis (TEXT) – Szczegóły dotyczące domku.
    - liczba_osob (INT, NOT NULL, UNSIGNED) – Maksymalna liczba gości.
    - cena_za_noc (INT, NOT NULL, UNSIGNED) – Cena wynajmu za noc.
    - kategoria (VARCHAR, NOT NULL) – Kategoria (krajobraz) domku.
    - lokalizacja (VARCHAR, NOT NULL) – Lokalizacja (miasto) domku.
    - id_wlasciciela (INT, FOREIGN KEY, NOT NULL, UNSIGNED) – Powiązanie z tabelą wlasciciele.
    - zdjecie (VARCHAR, NOT NULL) – Ścieżka do zdjęć domku.
    - polecany (TINYINT, NOT NULL) – Czy domek jest polecany.
- Tabela **rezerwacje** - Przechowuje informacje o rezerwacjach dokonywanych przez klientów. Zawiera kolumny:
    - id_rezerwacji (INT, PRIMARY KEY, AUTO_INCREMENT, NOT NULL, UNIQUE) – Unikalny identyfikator rezerwacji.
    - id_klienta (INT, FOREIGN KEY, NOT NULL) – Powiązanie z tabelą klienci.
    - id_domku (INT, FOREIGN KEY, NOT NULL, UNSIGNED) – Powiązanie z tabelą domki.
    - data_od (DATE, NOT NULL) – Data rozpoczęcia rezerwacji.
    - data_do (DATE, NOT NULL) – Data zakończenia rezerwacji.
    - status (VARCHAR, NOT NULL) – Status rezerwacji (np. "Oczekująca", "Potwierdzona").
    - data_dokonania_rezerwacji (DATETIME) – Data złożenia rezerwacji.
    - data_potwierdzenia (DATE) – Data potwierdzenia rezerwacji przez właściciela.
- Tabela **dostepnosc_domkow** – Przechowuje okresy niedostępności domków z innych powodów niż rezerwacja, np. remont. Zawiera kolumny:
    - id_dostępności (INT, AUTO_INCREMENT) – Unikalny identyfikator niedostępności.
    - id_domku (INT, PRIMARY KEY, FOREIGN KEY) – Powiązanie z tabelą domki.
    - data_od (DATE, PRIMARY KEY) – Data rozpoczęcia niedostępności.
    - data_do (DATE, PRIMARY KEY) – Data zakończenia niedostępności.

5.3 Relacje między tabelami

Relacje w bazie danych aplikacji umożliwiają powiązanie różnych elementów systemu, takich jak klienci, właściciele, domki i rezerwacje:

- **Klienci i rezerwacje**:

Relacja 1:N – Jeden klient może mieć wiele rezerwacji. Klucz obcy: id_klienta w tabeli rezerwacje odnosi się do klucza głównego id_klienta w tabeli klienci.

- **Właściciele i domki**:
  Relacja 1:N – Jeden właściciel może posiadać wiele domków. Klucz obcy: id_wlasciciela w tabeli domki odnosi się do klucza głównego id_wlasciciela w tabeli wlasciciele.
- **Domki i rezerwacje**:
  Relacja 1:N – Jeden domek może mieć wiele rezerwacji. Klucz obcy: id_domku w tabeli rezerwacje odnosi się do klucza głównego id_domku w tabeli domki.

5.4 Kluczowe zapytania SQL

Poniżej znajdują się przykładowe zapytania SQL, które są używane w aplikacji:

- **Pobieranie dostępnych domków w wybranym okresie**:

```
if (startDate && endDate) {
  baseQuery += `
    AND d.id_domku NOT IN (
      SELECT dd.id_domku FROM dostepnosc_domkow dd
      WHERE dd.id_domku = d.id_domku
        AND dd.data_od <= ?
        AND dd.data_do >= ?
    )
    AND d.id_domku NOT IN (
      SELECT r.id_domku FROM rezerwacje r
      WHERE r.id_domku = d.id_domku
        AND (
          r.data_od <= ?
          AND r.data_do >= ?
        )
    )
  `;
```

Rysunek 18: Zapytanie SQL - pobranie domków w wybranym okresie

- **Dodawanie nowej rezerwacji**:

```
// Wstawienie nowej rezerwacji do bazy danych
const insertSql = `
  INSERT INTO rezerwacje (id_domku, id_klienta, data_od, data_do, status)
  VALUES (?, ?, ?, ?, 'Oczekująca')
`;
```

Rysunek 19: Zapytanie SQL - tworzenie nowej rezerwacji

- **Pobieranie danych klienta**:

```
const sql = `
  SELECT imie, nazwisko, email, telefon
  FROM klienci
  WHERE id_klienta = ?
`;
```

Rysunek 20: Zapytanie SQL - pobieranie danych klienta

- **Pobieranie dostępności wybranego domku**:

```
const sqlReservations = `
  SELECT data_od, data_do
  FROM rezerwacje
  WHERE id_domku = ?
  AND status IN ('Potwierdzona', 'Oczekująca')
`;
const sqlUnavailable = `SELECT data_od, data_do FROM dostepnosc_domkow WHERE id_domku = ?`;
```

Rysunek 21: Zapytanie SQL - pobieranie dostępności wybranego domku

- **Pobieranie rezerwacji klienta**:

```
const sql = `
  SELECT
    rezerwacje.id_rezerwacji,
    rezerwacje.id_domku,
    rezerwacje.data_od,
    rezerwacje.data_do,
    rezerwacje.status,
    rezerwacje.data_dokonania_rezerwacji,
    rezerwacje.data_potwierdzenia,
    domki.nazwa AS nazwa_domku,
    domki.zdjecie AS zdjecie_domku
  FROM rezerwacje
  JOIN domki ON rezerwacje.id_domku = domki.id_domku
  WHERE rezerwacje.id_klienta = ?
`;
```

Rysunek 22: Zapytanie SQL - pobieranie rezerwacji klienta

5.5 Mechanizmy bezpieczeństwa bazy danych

Aby zapewnić bezpieczeństwo danych w aplikacji, zastosowano następujące mechanizmy:

- **Haszowanie haseł**: Wszystkie hasła użytkowników (klientów i właścicieli) są przechowywane w postaci zaszyfrowanej za pomocą algorytmu bcrypt.
- **Unikalne klucze**: Klucz unikalny dla pól takich jak email zapobiega duplikatom.
- **Walidacja wejścia**: Przed zapisaniem danych w bazie są one walidowane zarówno na poziomie frontendowym, jak i backendowym.
- **Uprawnienia dostępu**: W systemie dostępu do bazy danych zastosowano ograniczenia dostępu, zapewniając, że użytkownicy mogą wykonywać tylko operacje zgodne z ich rolą.
- **Mechanizm CORS**: Zapewnia, że dostęp do API bazy danych mają tylko uprawnione domeny.

6 BEZPIECZEŃSTWO

6.1 Wprowadzenie do zabezpieczeń aplikacji

Aplikacja implementuje podstawowe mechanizmy ochrony, takie jak uwierzytelnianie użytkowników, bezpieczne przechowywanie haseł, kontrola dostępu na podstawie ról oraz zabezpieczenia dotyczące wymiany danych (CORS). Dzięki tym mechanizmom zapewnione jest odpowiednie zabezpieczenie przed nieautoryzowanym dostępem i ryzykiem utraty danych.

6.2 Uwierzytelnianie użytkowników

Aplikacja wykorzystuje **uwierzytelnianie oparte na e-mail i haśle**. Użytkownicy logują się za pomocą swojego adresu e-mail oraz hasła. Po pomyślnym zalogowaniu, użytkownicy otrzymują dostęp do funkcji aplikacji, do których są uprawnieni, takich jak przeglądanie rezerwacji czy zarządzanie dostępnością domków.

```
// Logowanie klienta
exports.login = (req, res) => {
  const { email, haslo } = req.body;

  if (!email || !haslo) {
    return res.status(400).send("E-mail i hasło są wymagane");
  }

  // Sprawdzenie, czy użytkownik istnieje w bazie danych
  const sql = `SELECT * FROM klienci WHERE email = ?`;
  db.query(sql, [email], async (err, results) => {
    if (err) {
      console.error("Błąd podczas pobierania danych:", err);
      return res.status(500).send("Błąd serwera");
    }

    if (results.length === 0) {
      return res.status(401).send("Błędne hasło lub e-mail");
    }

    const user = results[0];

    // Porównanie hasła
    const isMatch = await bcrypt.compare(haslo, user.haslo);
    if (!isMatch) {
      return res.status(401).send("Błędne hasło lub e-mail");
    }

    const { haslo: _, ...userData } = user;
```

Rysunek 23: Kod odpowiedzialny za logowanie klienta

6.3 Przechowywanie haseł

Wszystkie hasła użytkowników są **bezpiecznie przechowywane** w bazie danych przy użyciu algorytmu **bcrypt**. Hasła są haszowane, co oznacza, że w bazie danych przechowywana jest tylko ich zaszyfrowana wersja, a nie hasła w postaci jawnej. Dzięki temu, nawet w przypadku wycieku danych, nie ma możliwości odzyskania oryginalnych haseł.

Rysunek 24: Zaszyfrowane hasła

6.4 CORS (Cross-Origin Resource Sharing)

Aplikacja implementuje zabezpieczenie **CORS**, które ogranicza dostęp do zasobów API tylko z określonych, zaufanych domen. To zabezpieczenie zapobiega atakom typu **Cross-Site Request Forgery (CSRF)** oraz nieautoryzowanemu dostępowi do API aplikacji z zewnętrznych, niezaufanych źródeł.

6.5 Uprawnienia i role użytkowników

Aplikacja posiada system ról, który rozróżnia dwa typy użytkowników: **klientów** oraz **właścicieli domków**. Każda rola ma przypisane różne uprawnienia:

- **Klient**: Może dokonywać rezerwacji i zarządzać swoimi rezerwacjami.
- **Właściciel**: Ma możliwość zarządzania rezerwacjami dokonanych przez klientów i przeglądania swoich ofert.

Kontrola dostępu na podstawie ról zapewnia, że użytkownicy mogą tylko wykonywać operacje, do których są uprawnieni.

6.6 Zarządzanie danymi poufnymi (plik .env)

Aplikacja korzysta z pliku **.env** w celu bezpiecznego przechowywania danych konfiguracyjnych i poufnych, takich jak:

- Dane dostępu do bazy danych (host, użytkownik, hasło, nazwa bazy),
- Port, na którym działa serwer,
- Inne kluczowe ustawienia, które nie powinny być przechowywane w kodzie źródłowym.

Wykorzystanie pliku .env pozwala na oddzielenie konfiguracji środowiska od logiki aplikacji, co zwiększa bezpieczeństwo oraz ułatwia zarządzanie różnymi środowiskami (np. deweloperskim, testowym i produkcyjnym).

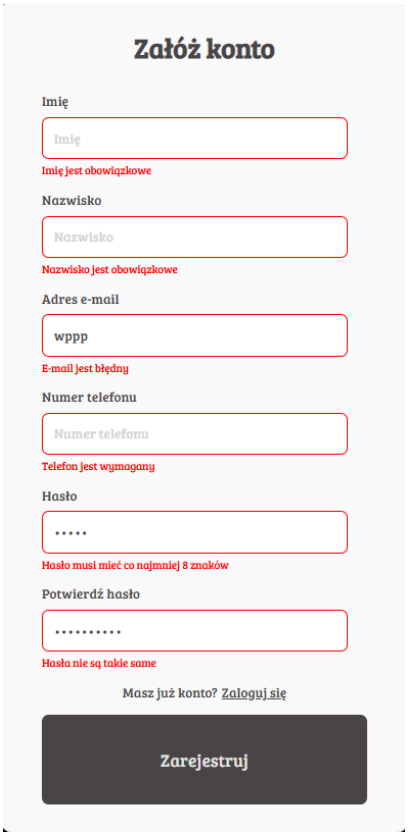
Aby uniknąć przypadkowego ujawnienia danych poufnych, plik **.env** został dodany do pliku **.gitignore**, dzięki czemu nie jest przesyłany do systemu kontroli wersji (np. Git).

7. TESTOWANIE

Proces testowania aplikacji był przeprowadzony w środowisku deweloperskim i obejmował zarówno testowanie frontendowe, jak i backendowe. Testy miały na celu weryfikację poprawności działania systemu, wykrywanie błędów oraz sprawdzanie, czy funkcjonalności spełniają wymagania.

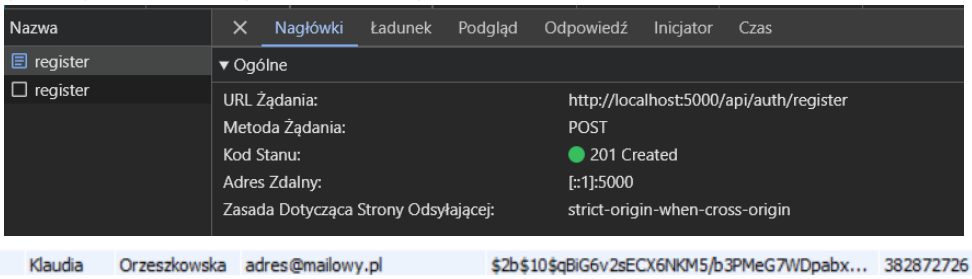
7.1 Testy jednostkowe i integracyjne

- **Testy jednostkowe**: Pojedyncze funkcjonalności, takie jak walidacja danych w formularzach, zostały przetestowane manualnie poprzez symulację różnych scenariuszy w aplikacji.



Rysunek 25: Test jednostkowy - walidacja danych w formularzu rejestracji

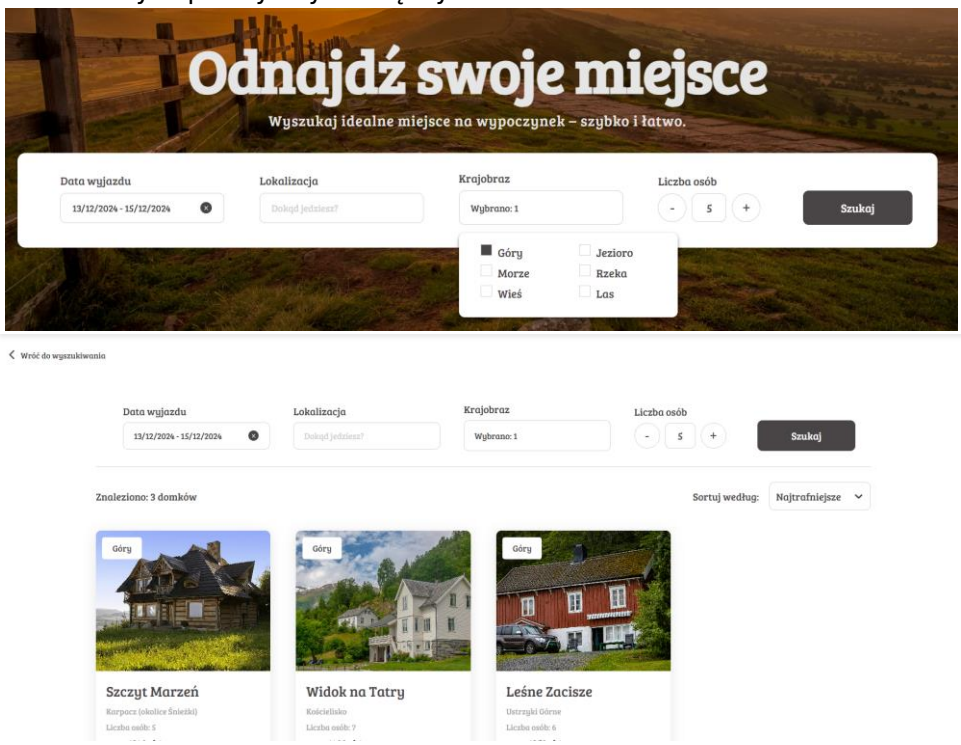
- **Testy integracyjne**: Przetestowano współdziałanie różnych komponentów, takich jak komunikacja między frontendem a backendem, poprzez manualne wysyłanie zapytań z interfejsu użytkownika i sprawdzanie wyników.



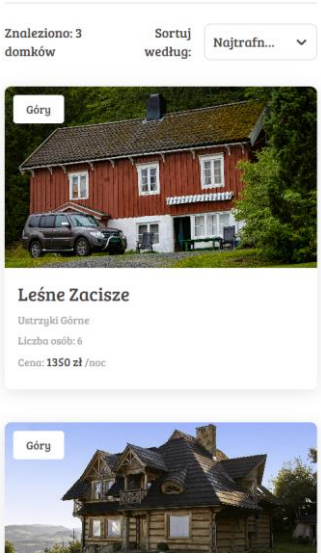
Rysunek 26: Test integracyjny rejestracji

7.2 Testy frontendowe

- **Testowanie interfejsu**: Sprawdzono, czy wszystkie komponenty frontendowe renderują się poprawnie, a interakcje użytkownika (np. klikanie przycisków, wypełnianie formularzy) działają zgodnie z założeniami.
- **Debugowanie w DevTools**: Używano narzędzi deweloperskich w przeglądarce, aby monitorować stan aplikacji, analizować komunikację z serwerem oraz sprawdzać poprawność danych przesyłanych między frontendem a backendem.



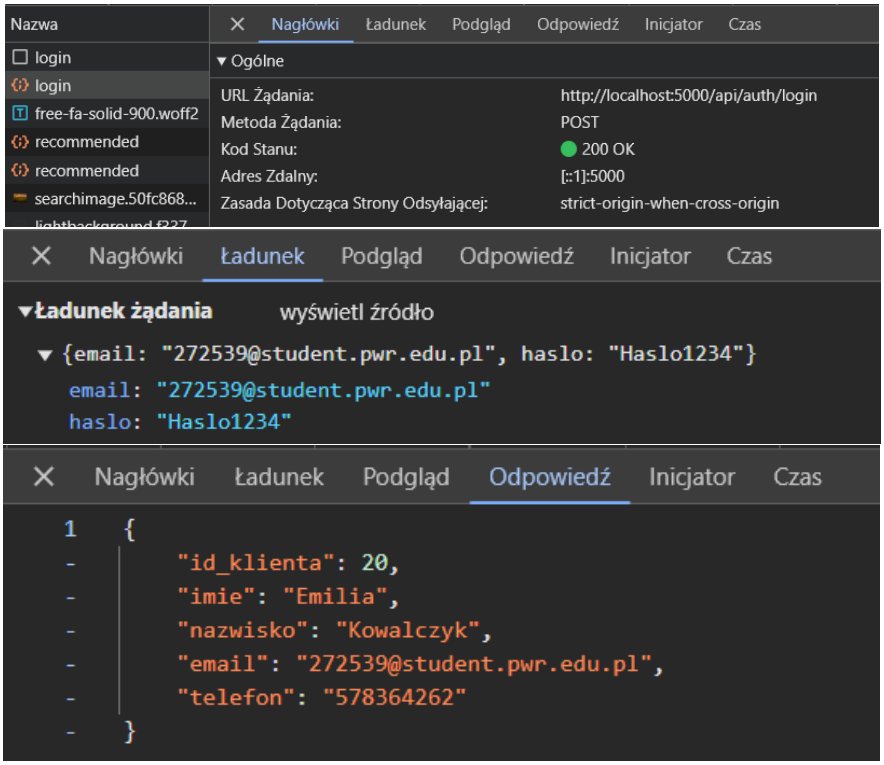
Rysunek 27: Test frontendowy – renderowanie wyszukiwarki



Rysunek 28: Test fronendowy - responsywność wyszukiwarki

7.3 Testy API

- **Testowanie manualne**: Weryfikacja działania endpointów API została przeprowadzona poprzez bezpośrednie wysyłanie zapytań HTTP (np. GET, POST, PUT) z poziomu przeglądarki.
- **Sprawdzanie odpowiedzi**: Analizowano zwracane odpowiedzi serwera (status HTTP, dane JSON), aby upewnić się, że działają zgodnie z oczekiwaniami.
- **Warunki brzegowe**: Testy obejmowały również przypadki wprowadzania niepełnych lub nieprawidłowych danych wejściowych w formularzach i obserwację zachowania API.

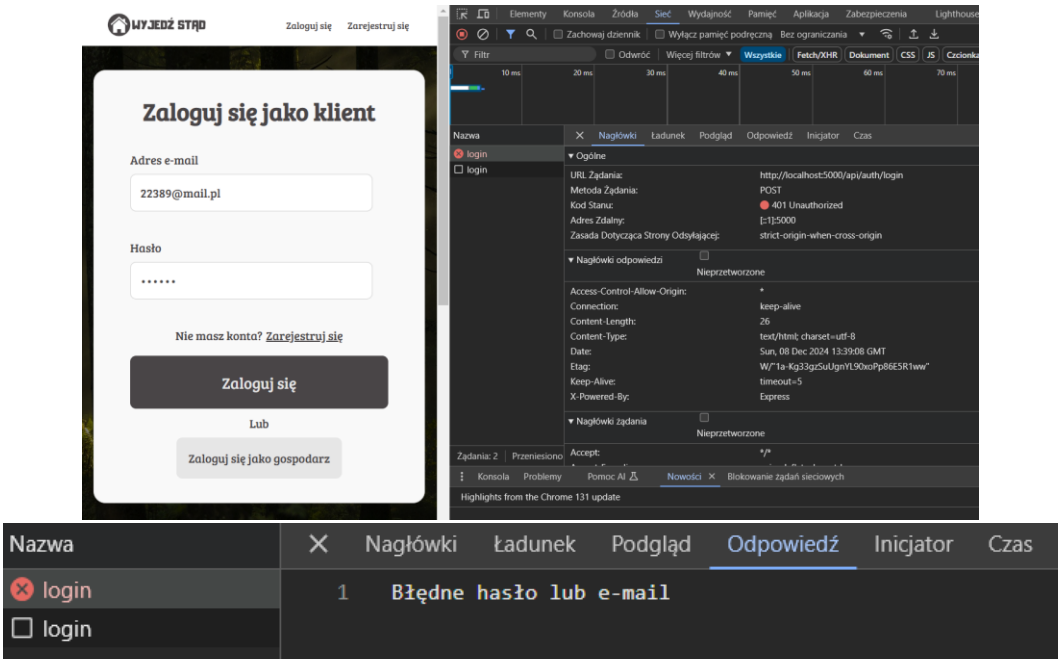


Rysunek 29: Test API - endpoint POST /api/auth/login

Rysunek 30: Test API - endpoint /api/houses/:id

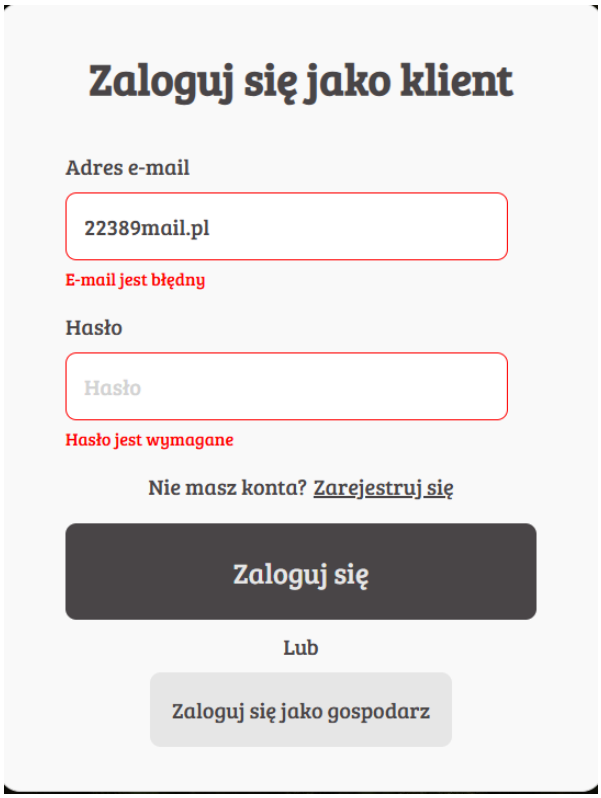
7.4 Testy bezpieczeństwa

- **Sprawdzanie uwierzytelniania**: Przetestowano poprawność procesów logowania i rejestracji, w tym obsługę nieprawidłowych danych.



Rysunek 31: Test bezpieczeństwa - błędne dane logowania

- **Walidacja danych wejściowych**: Weryfikowano, czy aplikacja odpowiednio odrzuca błędne dane wejściowe.



Rysunek 32: Test bezpieczeństwa - walidacja danych wejściowych przy logowaniu

- **Uprawnienia użytkowników**: Przetestowano, czy użytkownicy mają dostęp wyłącznie do zasobów i funkcji przypisanych do ich roli (np. klient nie ma dostępu do panelu właściciela).

8. PRZYSZŁE ROZSZERZENIA

Aby jeszcze bardziej udoskonalić aplikację i dostosować ją do potrzeb użytkowników, w przyszłości planowane są następujące rozszerzenia:

8.1 Edycja ofert dla właścicieli

Właściciele będą mieli możliwość edytowania szczegółów swoich ofert, takich jak nazwa domku, opis, liczba miejsc, cena za noc oraz zdjęcia. Pozwoli to na łatwiejsze zarządzanie ofertami i ich aktualizację.

8.2 Dodawanie okresów niedostępności

Wprowadzenie funkcji umożliwiającej właścicielom oznaczanie okresów, w których domek jest niedostępny z powodów niezależnych od rezerwacji, np. remontów lub prywatnego użytku.

8.3 Zwiększenie bezpieczeństwa

Rozważa się wdrożenie dodatkowych zabezpieczeń, w tym:

- Mechanizmów ochrony przed potencjalnymi atakami na JavaScript (np. XSS).
- Wzmocnienia ochrony bazy danych i całego serwisu, aby lepiej zabezpieczyć dane użytkowników.

8.4 Ukończenie konfiguracji CORS

Wdrożenie restrykcyjnej konfiguracji CORS, aby zezwolić na dostęp do API wyłącznie z określonych, zaufanych domen, co zwiększy bezpieczeństwo wymiany danych między frontendem a backendem.

8.5 Uwierzytelnianie za pomocą tokenów lub ciasteczek

Przejście na bardziej zaawansowany mechanizm uwierzytelniania, oparty na:

- **Tokenach JWT** (przechowywanych np. w pamięci przeglądarki lub ciasteczkach),
- Lub **ciasteczkach sesyjnych** z odpowiednio ustawionymi flagami bezpieczeństwa (HttpOnly, Secure).

8.6 Czyszczenie bazy danych

Wprowadzenie automatycznego mechanizmu archiwizacji i usuwania starych rezerwacji oraz nieaktualnych danych w celu optymalizacji bazy i zwiększenia wydajności.

8.7 Rozbudowa filtrów wyszukiwania

Dodanie nowych opcji filtrowania domków, takich jak:

- Wyposażenie (np. basen, sauna, kominek),
- Lokalizacja w konkretnym województwie lub na określonym terenie (np. Bieszczady, Mazury).

8.8 Podpowiadanie miast w polu lokalizacji

Ułatwienie wyszukiwania domków poprzez dynamiczne podpowiadanie nazw miast podczas wpisywania lokalizacji w formularzu. Funkcja ta może być wspierana przez autouzupełnianie z listy popularnych lokalizacji.