Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)
Інститут комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))
Кафедра комп'ютерних наук
(повна назва кафедри, циклової комісії)

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(повна назва кафедри, циклової комісії)

на тему: Розроблення інтерактивного вебзастосунку для барбершопу засобами React

Виконав: студент 4 курсу групи КН-41

спеціальності 122 "Ком'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Нижник Андрій Сергійович
(прізвище та ініціали)

Керівник Паславський М.М
(прізвище та ініціали)

Рецензент Руд С.О.
(прізвище та ініціали)

Львів - 2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакаравський)
Спеціальність 122 "Комп'ютерні науки"

ЗАТВЕРДЖУЮ:
Завідувач кафедри КН
_________ Борецька І.Б.
"10" червня 2025 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Нижник Андрій Сергійович
(прізвище, ім'я, по батькові)

1.Тема бакалаврської роботи: Розроблення інтерактивного вебзастосунку для барбершопу засобами React
керівник роботи Паславський М.М.ас. кафедри комп'ютерних наук
затверджені наказом вищого навчального закладу від "15" листопада 2024 р. № С-882

2. Термін подання студентом роботи 10 червня 2025р.

3. Вихідні дані до роботи Розробити інтерактивний вебзастосунок для барбершопу засобами React. Даний застосунок призначений для взаємодії барбершопу з клієнтами.

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Вступ
Стан проблемної області
Інформаційне забезпечення
Програмне та технічне забезпечення
Висновки
Список використаних джерел
Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Підготовка матеріалу до доповіді.

6. Дата видачі завдання 18 листопада 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд літератуних даних.	18.12.2024 – 05.01.2025	Виконано
2	Розділ 1. Стан проблемної області	19.01.2025 – 07.02.2025	Виконано
3	Розділ 2. Інформаційне та математичне забезпечення.	08.02.2025 – 24.03.2025	Виконано
4	Розділ 3. Програмне та технічне забезпечення	25.03.2025 – 10.05.2025	Виконано
5	Аналіз отриманих результатів та написання висновків. Оформлення дипломної роботи	11.05.2025 – 29.05.2025	Виконано
6	Здача пояснювальної записки на перевірку курівнику, виправлення помилок та здача роботи рецензенту.	31.05.2025 – 10.06.2025	Виконано

Студент ____________ (підпис) Нижник А. С. (прізвище та ініціали)

Керівник роботи ____________ (підпис) Паславський М.М. (прізвище та ініціали)

АНОТАЦІЯ

У даній дипломній роботі розроблено вебзастосунок для барбершопу, який дозволяє клієнтам здійснювати онлайн-запис, обираючи зручні дату, час, послугу та майстра. Система забезпечує авторизацію та автентифікацію користувачів з використанням JWT-токенів, а також надає особистий кабінет, де користувач може переглядати історію своїх візитів.

Адміністративна частина передбачає можливість керування майстрами та послугами через захищену панель керування. Серверна частина реалізована на базі Node.js з використанням Express.js та MongoDB, клієнтська – на основі React із застосуванням Redux Toolkit для управління станом. Розробка виконана з використанням мови програмування TypeScript, що забезпечує надійність і масштабованість застосунку.

Ключові слова: барбершоп, вебзастосунок, MERN, React, TypeScript, JWT, REST API, Redux Toolkit.

ANNOTATION

This thesis presents the development of a web application for a barbershop, allowing clients to book appointments online by selecting a convenient date, time, service, and barber. The system supports user registration and authentication using JSON Web Tokens (JWT) and provides a personal account where users can view their visit history.

An administrative panel is implemented to manage barbers and services through a secure interface. The backend is developed using Node.js with Express.js and MongoDB, while the frontend is built with React, utilizing Redux Toolkit for state management. The entire application is written in TypeScript, ensuring high reliability and scalability of the system.

Keywords: barbershop, web application, MERN, React, TypeScript, JWT, REST API, Redux Toolkit.

Keywords MERN, MongoDB, Express.js, React, Node.js, TypeScript, REST API, full-stack, web development, NoSQL

Технічне завдання

Розроблення вебзастосунку для барбершопу із можливістю онлайн-запису та адміністрування.

Розроблена система повинна мати:

- Систему реєстрації та входу користувачів з використанням JWT авторизації
- Особистий кабінет користувача з історією візитів та поточними записами.
- Адміністративну панель для:
    a. додавання, редагування та видалення майстрів;
    b. керування списком послуг.
- REST API для взаємодії між клієнтом та сервером.
- Управління глобальним станом за допомогою Redux Toolkit.
- Базу даних MongoDB для збереження інформації про користувачів, майстрів, послуги та записи.
- Кодову базу на TypeScript як для клієнтської, так і для серверної частини.
- Адаптивний інтерфейс користувача, реалізований на React.

Зміст

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ.................................... 7
Вступ.. 8
Розділ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ 9
1.1 Аналіз потреби у цифровізації послуг барбершопу 9
1.2 Проблеми управління доступом та ролями користувачів............................ 9
1.3 Аналіз існуючих рішень та їх недоліків 10
1.4 Переваги розробленого рішення.. 10
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 12
2.1 Технології MERN-стеку з TypeScript.. 12
2.2 Структура бази даних MongoDB 15
2.3 Реалізація авторизації та розмежування доступу 18
2.4 Алгоритм планування записів.. 20
2.5 Інструменти розробки та розгортання 21
2.6 Мова програмування TypeScript.. 22
РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ.......................... 28
3.2 Проектування баз даних 29
3.3 Розробка серверної частини 33
3.4 Розробка клієнтської частини 41
ВИСНОВОК... 48
СПИСОВ ВИКОРИСТАНИХ ДЖЕРЕЛ 49
Додаток А 50
Додаток Б 62

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

JavaScript (JS) – динамічна, об'єктно-орієнтована прототипна мова програмування.

TypeScript (TS) – строго типізована надбудова над JavaScript .

MERN – стек технологій: MongoDb, Express.js, React, Node.js.

MongoDB – документно-орінєнтована NoSQL база даних.

Express.js – фреймворк для створення серверної розробки.

React – бібліотека JavaScript для побудови користувацького інтерфейсу.

API (application programming interface) – набір чітко визначених методів взаємодії між програмами.

HTTPS - hyperText Transfer Protocol Secure – захищений протокол передачі даних.

Rest - архітктурний стиль побудови API .

Redux – бібліотека керування глобальним станом у React-додатках.

БД – база даних – декларативна мова програмування для взаємодії користувача з базою даних..

Auth – Authentication – автенфікація користувача.

JWT – JSON WEB TOKEN – формат безпечної передачі токенів автентифікації.

JWT access token - токен доступу – короткочасний токен для автентифікації.

JWT refresh token - токен оновлення – використовується для отримання нового access token.

Middleware - Проміжне програмне забезпечення для обробки запитів.

Вступ

У сучасному світі цифрові технології стали невід'ємною частиною бізнес-процесів у різних сферах, зокрема у сфері послуг. Все більше компаній прагнуть представити себе в онлайн-середовищі, надаючи користувачам зручний та функціональний інструмент для взаємодії з їхніми послугами. Барбершопи не є винятком — попит на онлайн-бронювання, перегляд розкладу майстрів та ведення запису клієнтів постійно зростає. Вебсайт із сучасним дизайном і надійним функціоналом стає важливим інструментом для обслуговування клієнтів.

Об'єктом дослідження є процес розроблення вебзастосунку для барбершопу. Предметом дослідження є методи побудови клієнт-серверного вебзастосунку із використанням стеку MERN, мови TypeScript, механізму авторизації через JWT та управління станом за допомогою Redux Toolkit.

Метою дипломної роботи є розроблення вебсайту для барбершопу, який дозволяє користувачам переглядати список послуг, здійснювати онлайн-бронювання, а адміністраторам — керувати даними через захищену систему авторизації.

Завдання дослідження:

1. Розробити вебзастосунок для барбершопу з можливістю онлайн-запису клієнтів на вибрану дату, час, послугу та майстра.
2. Реалізувати систему реєстрації та аутентифікації користувачів з використанням JWT.
3. Створити особистий кабінет користувача з переглядом історії візитів та поточних записів.
4. Розробити адміністративну панель для керування переліком послуг та майстрів.
5. Забезпечити інтерактивний інтерфейс користувача на основі React з використанням Redux Toolkit для управління станом.
6. Забезпечити захищену взаємодію між клієнтською та серверною частинами через REST API.

Розділ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Аналіз потреби у цифровізації послуг барбершопу

У сучасному світі, де попит на зручні та доступні онлайн-послуги постійно зростає, цифровізація традиційних сфер, таких як барбершопи, стає актуальною. З одного боку, клієнти прагнуть швидко записуватися на послуги, обирати зручний час і майстра, а з іншого — власники барбершопів стикаються з необхідністю ефективного управління записами, майстрами та послугами. Традиційні методи запису через телефон чи особисто часто призводять до помилок, накладок у графіку та незадоволення клієнтів.
Одним із рішень цієї проблеми є створення веб-сайту для барбершопу, який би надавав можливість онлайн-реєстрації з підтвердженням на пошту, перегляду профілю користувача з історією відвідувань, а також вибору послуг і майстрів із гнучким графіком. Такий підхід дозволяє автоматизувати процеси, підвищити зручність для клієнтів і оптимізувати роботу адміністрації.

1.2 Проблеми управління доступом та ролями користувачів

Зростання кількості онлайн-сервісів призводить до необхідності надійної системи авторизації та розмежування прав доступу. Для барбершопу важливо забезпечити безпечний вхід для звичайних користувачів (клієнтів) і адміністраторів, які потребують додаткових функцій, таких як управління майстрами, послугами та перегляд усіх записів. Наявність уразливостей у системі авторизації може призвести до витоку персональних даних клієнтів або несанкціонованого доступу до адмін-панелі.

Для вирішення цієї проблеми розроблено веб-сайт із системою логіну та реєстрації, що включає підтвердження на пошту, профіль користувача з персональними даними та історією візитів, а також адмін-панель для

управління контентом. Система побудована з використанням стеку MERN (MongoDB, Express.js, React, Node.js) із TypeScript, що забезпечує типізацію та підвищує безпеку коду. Авторизація реалізовано через захищені сесії та токени, а доступ до адмін-панелі обмежено ролями.

1.3 Аналіз існуючих рішень та їх недоліків

На ринку вже існують платформи для онлайн-запису до салонів краси та барбершопів, однак вони часто мають обмеження: висока вартість підписки, відсутність гнучкості у кастомізації або складність інтеграції з внутрішніми процесами. Більшість рішень не надають повноцінної адмін-панелі для управління майстрами та послугами, а також не завжди враховують потреби локальних бізнесів у персоналізації.

Розроблений веб-сайт усуває ці недоліки, надаючи відкрите рішення на базі MERN із TypeScript, яке дозволяє адаптувати функціонал під конкретні потреби барбершопу. Використання MongoDB забезпечує гнучке зберігання даних про користувачів, записи та послуги, а React із TypeScript гарантує зручний і безпечний інтерфейс. Розгортання на хмарних платформах (наприклад, Heroku чи Vercel) робить систему доступною з будь-якого пристрою.

1.4 Переваги розробленого рішення

Розроблений веб-сайт забезпечує зручність для клієнтів завдяки можливості онлайн-запису з вибором дати, часу, послуги та майстра, а також підтвердження через електронну пошту. Для адміністраторів передбачена адмін-панель, що дозволяє додавати, редагувати та видаляти майстрів і послуги, а також переглядати всі записи. Використання TypeScript підвищує надійність коду, а MERN-стек забезпечує масштабованість і простоту

підтримки. Система авторизації з розмежуванням ролей гарантує безпеку даних, що є ключовим для довіри клієнтів і бізнесу.

Таким чином, розроблене рішення автоматизує та оптимізує процеси барбершопу, підвищуючи рівень сервісу та конкурентоспроможність на ринку. Воно може бути адаптоване для інших подібних бізнесів, що потребують цифровізації.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Технології MERN-стеку з TypeScript

Для реалізації веб-сайту барбершопу використано сучасний стек технологій MERN (MongoDB, Express.js, React, Node.js) із TypeScript. MongoDB забезпечує гнучке зберігання даних, таких як інформація про користувачів, записи, майстрів і послуги, завдяки своїй NoSQL-структурі. Express.js на базі Node.js використовується для створення REST API, що забезпечує зв'язок між клієнтською частиною та базою даних. React із TypeScript відповідає за побудову інтерактивного інтерфейсу користувача, а TypeScript додає статичну типізацію, що зменшує кількість помилок під час розробки.

Клієнтська частина (React) взаємодіє з сервером через HTTP-запити з використанням бібліотеки Axios. Для авторизації застосовуються JSON Web Tokens (JWT), які забезпечують безпечний обмін даними між клієнтом і сервером. Підтвердження реєстрації через електронну пошту реалізовано за допомогою бібліотеки Nodemailer, яка інтегрована в серверну частину. Для маршрутизації на клієнтській стороні використовується бібліотека React Router, що дозволяє створювати зручну навігацію між сторінками, такими як список майстрів, форма запису та профіль користувача. На сервері для валідації вхідних даних застосовується бібліотека Joi, що забезпечує перевірку коректності даних перед їх обробкою.

Приклад серверного контролера для авторизації

Нижче наведено приклад контролера ClientController, який обробляє запити на реєстрацію та вхід користувачів із використанням Express.js і TypeScript. Цей код демонструє, як серверна частина інтегрується з базою даних через сервіс і повертає відповіді з JWT-токенами (див. рисунок 2.1):

```
 6  class ClientController {
      Windsurf: Refactor | Explain | Generate JSDoc | ×
 7    async registration(req: Request, res: Response, next: NextFunction) {
 8      try {
 9        const errors = validationResult(req);
10        if (!errors.isEmpty()) {
11          return next(new AppError(400, "Validation error", errors.array()));
12        }
13        const { name, email, password } = req.body;
14        const clientData = await clientService.registration(
15          name,
16          email,
17          password,
18          "client"
19        );
20        res.cookie("refreshToken", clientData.refreshToken, {
21          maxAge: 30 * 24 * 60 * 60 * 1000,
22          httpOnly: true,
23        });
24        return res.json(clientData);
25      } catch (e) {
26        next(e);
27      }
28    }
29
      Windsurf: Refactor | Explain | Generate JSDoc | ×
30    async login(req: Request, res: Response, next: NextFunction) {
31      try {
32        const { email, password } = req.body;
33        const clientData = await clientService.login(email, password);
34        res.cookie("refreshToken", clientData.refreshToken, {
35          maxAge: 30 * 24 * 60 * 60 * 1000,
36          httpOnly: true,
37        });
38        return res.json(clientData);
39      } catch (e) {
40        next(e);
41      }
42    }
43
```

Рисунок 2.1 – Приклад серверного контролера для авторизації

Приклад сервісу авторизації на клієнтській стороні

Для спрощення взаємодії з сервером на клієнтській стороні створено сервісний клас AuthService, який інкапсулює запити для авторизації. Нижче наведено приклад із AuthService.ts, що демонструє методи для входу та реєстрації з використанням Axios і TypeScript (див. рисунок 2.2):

```
import $api from "./httpCommon";
import { AuthResponse } from "../interfaces/authResponse";

You, 4 weeks ago | 1 author (You) | Windsurf: Refactor | Explain
export default class AuthService {
  Windsurf: Refactor | Explain | Generate JSDoc | ×
  static async login(email: string, password: string) {
    return $api.post<AuthResponse>("/clients/login", { email, password });
  }

  Windsurf: Refactor | Explain | Generate JSDoc | ×
  static async registration(name: string, email: string, password: string) {
    return $api.post<AuthResponse>("/clients/registration", {
      name,
      email,
      password,
    });
  }

```

Рисунок 2.2 - Сервіс авторизації на клієнтській стороні

Налаштування HTTP-клієнта

Для забезпечення коректної взаємодії між клієнтською частиною (React) і сервером у проєкті налаштовано HTTP-клієнт на базі бібліотеки Axios. Налаштування включає базову URL-адресу API, заголовки запитів, обробку авторизації через JWT-токени та механізм автоматичного оновлення токенів у випадку їх прострочення. Це дозволяє підтримувати безперервну авторизацію користувача, навіть якщо access-токен стає недійсним.

Нижче наведено приклад із файлу httpCommon.ts, який демонструє налаштування HTTP-клієнта з перехоплювачами запитів і відповідей (див. рисунок 2.3):

```typescript
import axios from "axios";
import { AuthResponse } from "../interfaces/authResponse";

export const API_URL = "http://localhost:5000/api";
const $api = axios.create({
  baseURL: API_URL,
  headers: {
    "Content-type": "application/json",
  },
  withCredentials: true,
});

$api.interceptors.request.use((config) => {
  config.headers.Authorization = `Bearer ${localStorage.getItem("token")}`;
  return config;
});

$api.interceptors.response.use(
  (config) => {
    return config;
  },
  async (error) => {
    const originalRequest = error.config;
    if (error.response.status === 401 && error.config && !error.config._isRetry) {
      originalRequest._isRetry = true;
      try {
        const response = await axios.get<AuthResponse>(
          `${API_URL}/clients/refresh`,
          { withCredentials: true }
        );
        localStorage.setItem("token", response.data.accessToken);
        return $api.request(originalRequest);
      } catch (e) {
        console.log("User is not authorized");
      }
    }
    throw error;
  }
);

export default $api;
```

Рисунок 2.3 - Налаштування HTTP-клієнта

2.2 Структура бази даних MongoDB

MongoDB використовується для зберігання всіх даних веб-сайту барбершопу. Для роботи з базою даних застосовується бібліотека Mongoose із TypeScript, що забезпечує типізацію моделей і спрощує взаємодію з MongoDB. База даних включає наступні основні колекції:

- **Користувачі (Clients):** містить дані про користувачів (name, email, пароль у захешованому вигляді, роль — "client" або "admin"), а також пов’язаний список їхніх візитів (visits). Паролі зберігаються з використанням бібліотеки bcrypt для хешування.

- **Візити (Visits):** зберігає інформацію про записи (дата і час — date, коментар — comment, обрана послуга — favor, майстер — barber, користувач — client).
- **Майстри (Barbers):** містить дані про майстрів (фотографія — image, категорія — barberCategory, переклади — translation, список візитів — visits, коефіцієнт для ціни — coef).
- **Послуги (Favors):** включає інформацію про послуги (тривалість — time, ціна — price, переклади — translations, список візитів — visits, зв’язок із категоріями майстрів — BarberCategoryFavor).
- **Категорії майстрів (BarberCategories):** містить дані про категорії майстрів (назва — categoryName, список майстрів — barbers, зв’язок із послугами через BarberCategoryFavor).
- **Переклади майстрів (BarberTranslations):** зберігає переклади для майстрів (мова — language, ім’я — name, прізвище — surname, зв’язок із майстром — barber).
- **Переклади послуг (FavorTranslations):** містить переклади для послуг (мова — language, назва — name, зв’язок із послугою — favor).
- **Зв’язок між категоріями та послугами (BarberCategoryFavors):** забезпечує зв’язок між категоріями майстрів і послугами (barberCategory, favor).
- **Токени (Tokens):** зберігає токени для авторизації (refreshToken, зв’язок із користувачем — client).

Для забезпечення швидкого доступу до даних створено індекси на часто використовуваних полях, наприклад, на email користувача (у колекції Clients) та дату візиту (у колекції Visits). Mongoose-схеми з TypeScript дозволяють чітко визначити структуру даних і уникнути помилок під час збереження.

Структура бази даних і зв’язки між колекціями наведені на діаграмі класів (див. рисунок 2.2). Діаграма відображає всі основні моделі та їхні взаємозв’язки:

- Модель Client пов’язана з Visits (1-to-many) і Tokens (1-to-many).

- Модель Barber пов’язана з Visits (1-to-many), BarberCategory (many-to-1) і BarberTranslation (1-to-many).
- Модель Favor пов’язана з Visits (1-to-many), FavorTranslation (1-to-many) і BarberCategoryFavor (1-to-many).
- Модель BarberCategory пов’язана з Barbers (1-to-many) і BarberCategoryFavor (1-to-many).
- Зв’язки між моделями забезпечують гнучкість і дозволяють ефективно отримувати пов’язані дані, наприклад, список візитів користувача або доступні послуги для певного майстра.

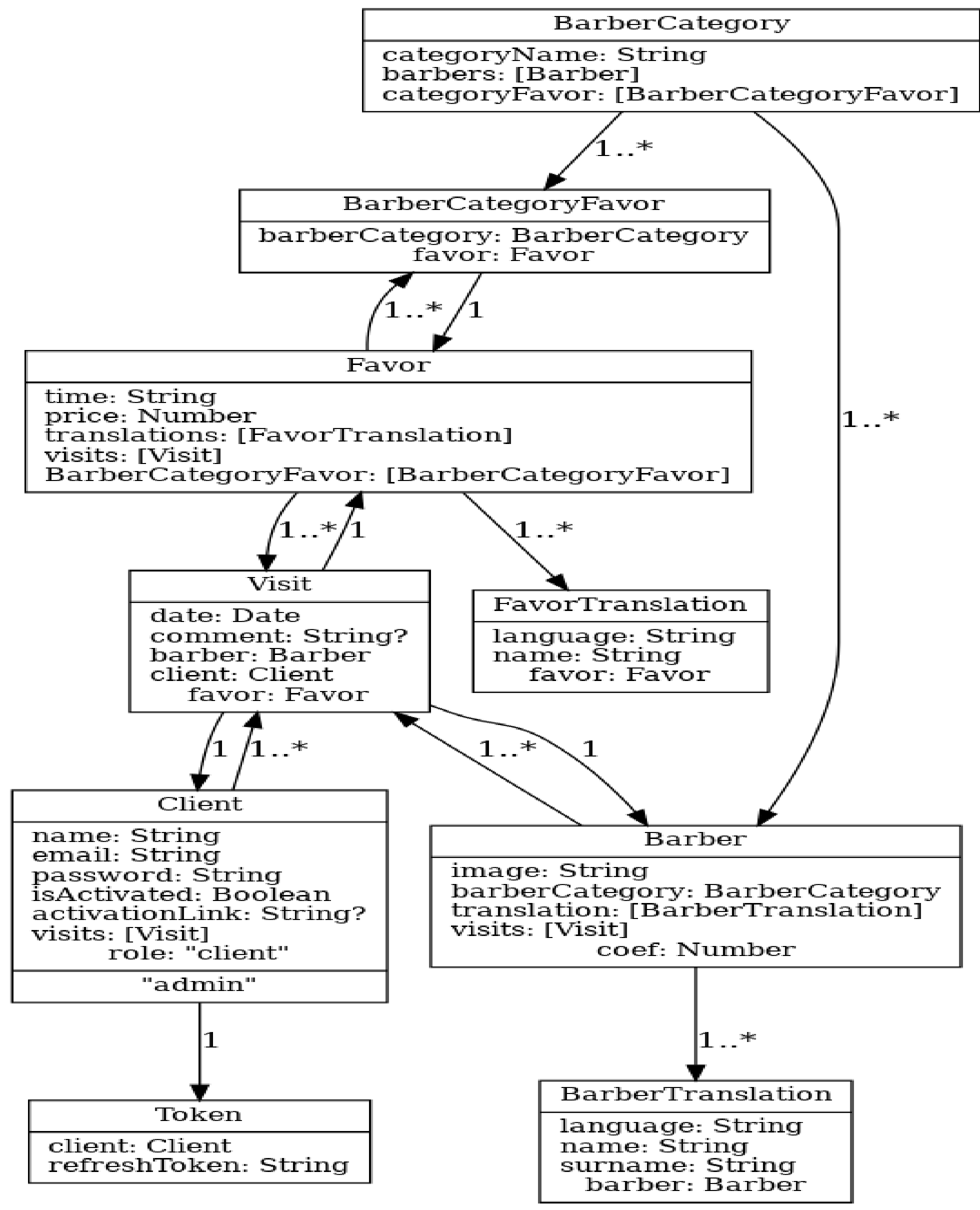

Рисунок 2.4 – Діаграма класів

2.3 Реалізація авторизації та розмежування доступу

Система авторизації базується на JWT-токенах. Після успішної реєстрації або входу користувач отримує токен, який зберігається на клієнтській стороні та додається до заголовків HTTP-запитів для доступу до захищених маршрутів. Для захисту паролів використовується бібліотека bcrypt, а для генерації та перевірки токенів — бібліотека jsonwebtoken.

Розмежування доступу реалізовано через ролі користувачів (клієнт або адмін). Адмін-панель доступна лише користувачам із роллю "адмін". Для цього на сервері створено middleware, який перевіряє роль користувача перед виконанням запитів до захищених маршрутів (наприклад, додавання/видалення майстрів). Підтвердження email під час реєстрації забезпечує додатковий рівень безпеки, запобігаючи створенню фейкових акаунтів.

Математична ***основа шифрування паролів за допомогою bcrypt***

Bcrypt — це алгоритм хешування паролів, спеціально розроблений для забезпечення безпеки шляхом ускладнення атак типу "brute force". Він базується на криптографічному алгоритмі Blowfish і використовує кілька математичних і обчислювальних підходів для створення безпечного хешу.

1. **Генерація солі (salt):** bcrypt автоматично генерує унікальну сіль — випадкову послідовність бітів (зазвичай 128 бітів, що кодується у 22 символи у форматі Base64). Сіль додається до пароля перед хешуванням, щоб запобігти атакам із використанням попередньо обчислених таблиць (rainbow tables). Математично сіль забезпечує унікальність хешу навіть для однакових паролів.
2. **Ітеративне хешування з використанням Blowfish:** bcrypt використовує адаптовану версію алгоритму Blowfish для створення хешу. Blowfish — це симетричний блоковий шифр, який працює з 64-бітними блоками і ключем змінної довжини (до 448 бітів). У bcrypt пароль і сіль комбінуються, після чого Blowfish застосовується у

процесі, що називається "key stretching". Основна функція в bcrypt — це EksBlowfish (Expensive Key Schedule Blowfish), яка включає:

   - Ініціалізацію стану Blowfish із використанням солі та пароля.
   - Виконання 64 раундів шифрування для кожного блоку (Blowfish зазвичай має 16 раундів, але bcrypt збільшує їх кількість для підвищення безпеки).
   - Ітеративне повторення цього процесу відповідно до "cost factor" (фактора вартості), який визначає кількість ітерацій у степені двійки (наприклад, cost factor = 12 означає 12^{12} = 4096 ітерацій). Це робить обчислення хешу навмисно повільним, ускладнюючи атаки перебором.

3. **Форматом результату** роботи bcrypt є: рядок, який складається з трьох частин:
   - Версія алгоритму (наприклад, $2b$ — версія bcrypt).
   - Фактор вартості (наприклад, 12 — 4096 ітерацій).
   - Сіль і хеш (сіль — перші 22 символи, хеш — решта 31 символ). Наприклад: $2b$12$jXgTG7QfX5gX8X9X9X9X9e.abcdefg1234567890abcdefg1234567890.

     Загальна довжина хешу становить 60 символів.
4. **Переваги математичного підходу:**
   - **Сіль:** унеможливлює використання попередньо обчислених таблиць, оскільки кожен хеш унікальний завдяки солі.
   - **Ітеративність:** великий cost factor робить атаки перебором обчислювально дорогими. Наприклад, із cost factor 12 сучасний комп’ютер витрачає приблизно 0.3 секунди на обчислення одного хешу, що для користувача непомітно, але для зловмисника, який перебирає мільйони паролів, стає значною перешкодою.

- **Адаптивність:** фактор вартості можна збільшувати з ростом обчислювальних потужностей, зберігаючи безпеку.

Таким чином, bcrypt забезпечує високий рівень безпеки паролів завдяки комбінації солі, ітеративного хешування та алгоритму Blowfish, що робить його стійким до сучасних методів злому. У проєкті цей механізм надійно захищає паролі користувачів від несанкціонованого доступу.

2.4 Алгоритм планування записів

Для реалізації системи записів розроблено алгоритм, який враховує графік майстрів, тривалість послуг і зайняті слоти часу. Алгоритм працює наступним чином:

- **Вибір майстра, дати, часу та послуги:** Користувач спочатку обирає майстра зі списку доступних спеціалістів, потім визначає дату і час візиту через календарний інтерфейс, а на завершальному етапі обирає послугу з доступного переліку. Цей процес відображається на інтуїтивному інтерфейсі, де користувач проходить кроки: вибір спеціаліста, вибір дати та часу, вибір послуги (див. рисунок 2.5).
- **Перевірка доступності:** На основі обраних параметрів (майстер, дата, час) система надсилає запит до сервера для перевірки графіка обраного майстра та наявності вільних слотів. MongoDB-запит порівнює обраний час із уже зайнятими візитами (колекція Visits), щоб уникнути накладок у графіку.
- **Збереження запису:** Запис автоматично зберігається у MongoDb, користувач отримає сповіщення підтвердження. Система надсилає запит на сервер для створення нового запису. Запис додається до колекції Visits із прив’язкою до користувача (client), майстра (barber), послуги (favor), дати та часу.

- **Сповіщення:** Після успішного створення запису сервер генерує email-підтвердження через бібліотеку Nodemailer. Користувачу надсилається лист із деталями візиту (майстер, дата, час, послуга).

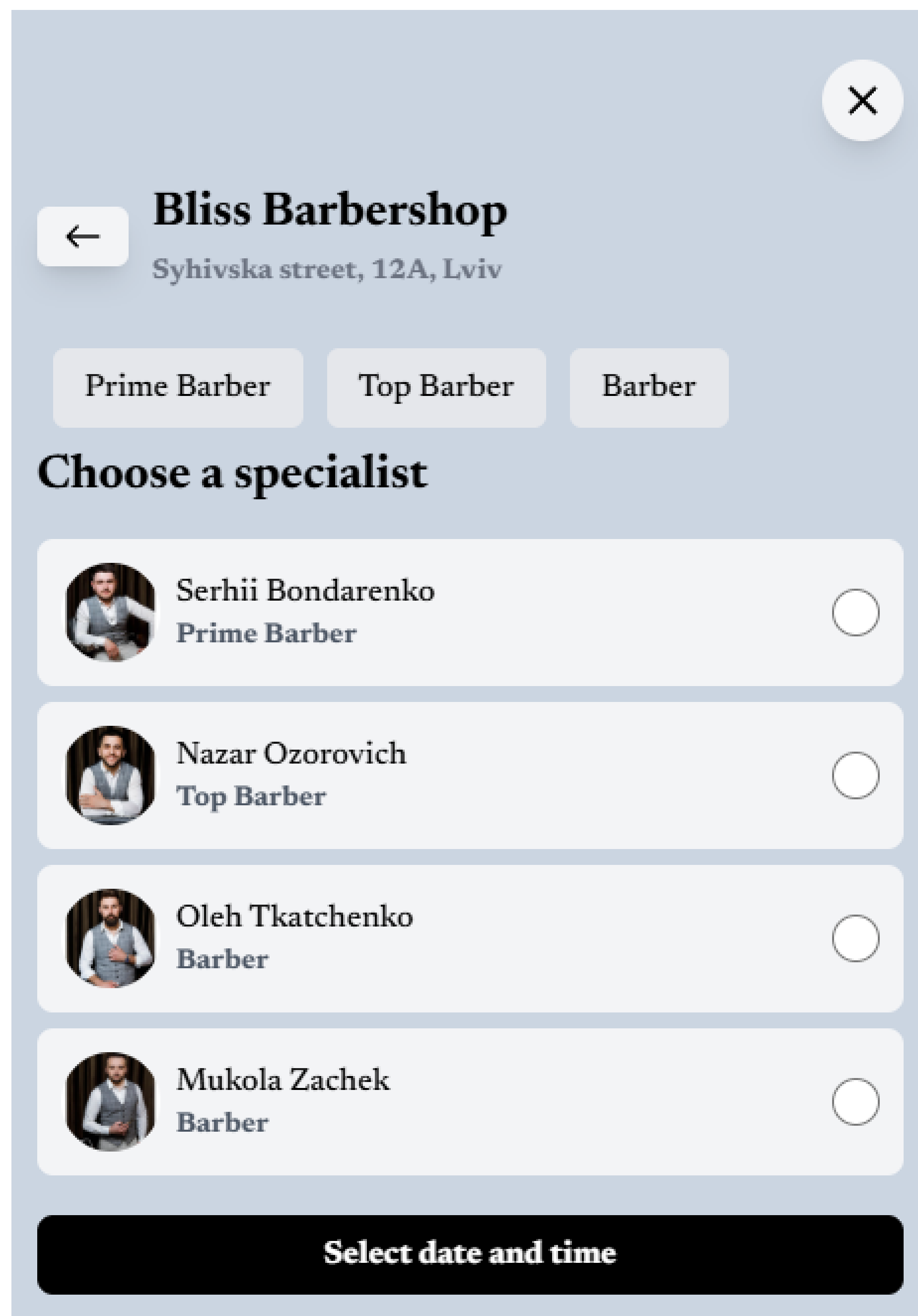

Рисунок 2.5 – Фрагмент користувацького інтерфейсу для вибору майстра

2.5 Інструменти розробки та розгортання

Розробка веб-сайту барбершопу проводилася у середовищі Visual Studio Code із плагінами для TypeScript, React і MongoDB, що забезпечують зручну

роботу з кодом, автодоповнення та підтримку синтаксису. Для локального тестування бази даних використовувався MongoDB Compass, який дозволяє візуально переглядати та тестувати колекції MongoDB.

Серверна частина, побудована на Express.js, розгорнута на платформі Heroku, а клієнтська частина, що використовує React і бібліотеку Axios для HTTP-запитів, розгорнута на Vercel, що гарантує доступність додатка з будь-якого пристрою. Для запуску серверного додатка застосовується команда npx ts-node app.ts, що інтегрує TypeScript із Node.js.

Для управління залежностями використано пакетний менеджер npm, а залежності включають бібліотеки для серверної частини: bcryptjs для хешування паролів, cookie-parser і cors для обробки cookie та міждоменних запитів, dotenv для роботи з змінними середовища, express-validator і validator для валідації даних, jsonwebtoken для роботи з JWT-токенами, mongoose для взаємодії з MongoDB, multer для завантаження файлів, nodemailer для відправки email, uuid для генерації унікальних ідентифікаторів, cloudinary для роботи з хмарним зберіганням зображень, date-fns для обробки дат, crypto для криптографічних операцій і mongodb як драйвер для MongoDB. Для розробки також використані типизації (@types/bcrypt, @types/dotenv, @types/node) і TypeScript (версія 5.7.2) у якості dev-залежностей.

Для форматування коду та забезпечення його якості застосовуються Prettier і ESLint із правилами для TypeScript, що допомагають підтримувати єдиний стиль і виявляти потенційні помилки.

2.6 Мова програмування TypeScript

TypeScript — це надбудова над JavaScript, розроблена компанією Microsoft і вперше випущена у 2012 році. Вона є мовою програмування з відкритим вихідним кодом, яка додає статичну типізацію до JavaScript, зберігаючи при цьому його гнучкість. TypeScript призначений для розробки великих і складних додатків, забезпечуючи кращу підтримку інструментів

розробки, таких як автодоповнення, рефакторинг і виявлення помилок на етапі компіляції. Код TypeScript компілюється в JavaScript, що дозволяє виконувати його в будь-якому середовищі, де працює JavaScript, наприклад, у браузерах або на сервері через Node.js.

Основні особливості TypeScript

TypeScript має низку характеристик, які роблять його потужним інструментом для розробки:

- **Статична типізація:** На відміну від JavaScript, де типи даних визначаються динамічно під час виконання, TypeScript дозволяє визначати типи змінних, параметрів і повернення функцій на етапі написання коду. Це зменшує кількість помилок, пов'язаних із типами, і покращує читабельність коду.
- **Сумісність із JavaScript:** TypeScript є надбудовою над JavaScript, тому весь існуючий JavaScript-код є валідним TypeScript-кодом. Це дозволяє поступово інтегрувати TypeScript у проєкти, не переписуючи їх із нуля.
- **Підтримка сучасних стандартів:** TypeScript підтримує всі сучасні можливості JavaScript (ES6 і новіші), такі як стрілкові функції, модулі, асинхронне програмування, а також додає власні можливості, наприклад, інтерфейси та абстрактні класи.
- **Інструменти розробки:** Завдяки типізації TypeScript забезпечує покращену підтримку в IDE (наприклад, Visual Studio Code), включаючи автодоповнення, перевірку типів і попередження про помилки ще до виконання коду.
- **Активна спільнота та екосистема:** TypeScript активно розвивається і має велику підтримку спільноти. Існують численні бібліотеки типів (@types/*) для популярних JavaScript-бібліотек, що спрощує їх використання в TypeScript-проєктах.

TypeScript широко використовується в веб-розробці, зокрема для створення фронтенд-додатків із React, Angular чи Vue, а також бекенд-додатків із Node.js і Express.js, як у випадку з веб-сайтом барбершопу.

2.6.1 Основи об'єктно-орієнтованого програмування в TypeScript

TypeScript підтримує об’єктно-орієнтоване програмування (ООП), яке є однією з ключових парадигм для створення модульного, масштабованого та зрозумілого коду. ООП у TypeScript базується на тих самих принципах, що й у JavaScript, але додає сувору типізацію та додаткові можливості, такі як модифікатори доступу та інтерфейси.

Процедурний підхід

Процедурний підхід у TypeScript (і JavaScript) базується на написанні функцій, які виконують певні задачі. Функції отримують дані, обробляють їх і повертають результат. Цей підхід корисний для простих скриптів, але має обмеження для великих проєктів.

Переваги процедурного підходу:

- Простота: Легко писати та розуміти для невеликих задач.
- Ефективність: Менше накладних витрат порівняно з ООП для простих операцій.

Недоліки процедурного підходу:

- Складність масштабування: У великих проєктах код стає заплутаним через відсутність чіткої структури.
- Повторення коду: Часто доводиться дублювати логіку.
- Неконтрольований доступ до даних: Глобальні змінні можуть призводити до помилок.

Об’єктно-орієнтоване програмування

ООП у TypeScript дозволяє створювати класи й об’єкти, які моделюють реальні сутності. Класи визначають структуру та поведінку об’єктів, а об’єкти є екземплярами цих класів.

Переваги ООП:

- Модульність: Код поділений на класи, що спрощує його розуміння та модифікацію.
- Повторне використання: Успадкування та поліморфізм дозволяють уникати дублювання коду.

- Контроль доступу: Модифікатори доступу (public, private, protected) забезпечують безпеку даних.
- Чітка структура: Ієрархія класів відображає логіку системи.

Недоліки ООП:

- Складність: Для новачків може бути важким через абстрактні концепції.
- Збільшення обсягу коду: Опис класів і методів може зробити код довшим.
- Накладні витрати: Для простих задач ООП може бути надмірним.

Принципи ООП у TypeScript

ООП у TypeScript базується на чотирьох основних принципах:

1. **Інкапсуляція:** Дані та методи об'єднуються в класі, а модифікатори доступу (private, protected, public) контролюють доступ до них. Це захищає внутрішню реалізацію від зовнішнього впливу.
2. **Абстракція:** Класи та інтерфейси дозволяють приховати деталі реалізації, надаючи лише необхідний інтерфейс для взаємодії.
3. **Успадкування:** Класи можуть успадковувати властивості та методи інших класів, що сприяє повторному використанню коду.
4. **Поліморфізм:** Об'єкти різних класів можуть реагувати на однакові методи по-різному, що робить код гнучким.

Властивості та методи

Властивості класу (атрибути) оголошуються з типами, що забезпечує чітке визначення структури об'єкта. Методи визначають поведінку об'єкта та реалізують його логіку. Модифікатори доступу (public, private, protected) контролюють видимість властивостей і методів. Наприклад, у класі AppError властивості status і errors оголошено без модифікатора (за замовчуванням public), а методи UnauthorizedError, BadRequest і ForbiddenError є статичними і доступні для виклику без створення екземпляра класу (див. рис. 2.6).

```
You, 4 weeks ago | 1 author (You) | Windsurf: Refactor | Explain
1   class AppError extends Error {
2     status: number;
3     errors: string[];
      Windsurf: Refactor | Explain | Generate JSDoc | ×
4     constructor(status: number, message: string, errors = []) {
5       super(message);
6       this.status = status;
7       this.errors = errors;   }
8
      Windsurf: Refactor | Explain | Generate JSDoc | ×
9     static UnauthorizedError() {
10      return new AppError(401, "User is not authorized");
11    }
      Windsurf: Refactor | Explain | Generate JSDoc | ×
12    static BadRequest(message:string, errors = []) {
13      return new AppError(400, message, errors);
14    }
      Windsurf: Refactor | Explain | Generate JSDoc | ×
15    static ForbiddenError() {
16      return new AppError(403, "Access forbidden");
17    }
18  }
19
20  export default AppError;
```

Рисунок 2.6 - Клас AppError:

Доступ до властивостей і методів

Доступ до властивостей і методів здійснюється через крапкову нотацію. Приватні властивості (private) доступні лише всередині класу, для зовнішнього доступу зазвичай створюються гетери та сетери. У прикладі з AppError властивості status і errors є публічними, тому до них можна звертатися напряму. Статичні методи, такі як UnauthorizedError, викликаються через назву класу (AppError.UnauthorizedError()).

Конструктор

Конструктор у TypeScript визначається за допомогою методу constructor. У класі AppError конструктор приймає параметри status, message і необов’язковий масив errors, ініціалізуючи відповідні властивості об’єкта. Він

також викликає конструктор батьківського класу Error через super(message) для успадкування стандартної поведінки помилок.

Успадкування

TypeScript підтримує успадкування через ключове слово extends. Клас AppError успадковує Error, додаючи власні властивості та методи. Нижче наведено приклад із вашим кодом і додатковим розширенням (див. рис.2.7)

```
1   import AppError from "./appError";
2
    Windsurf: Refactor | Explain
3   class CustomAppError extends AppError {
4     private details: string;
5
      Windsurf: Refactor | Explain | Generate JSDoc | ×
6     constructor(status: number, message: string, errors: string[], details: string) {
7       super(status, message, errors);
8       this.details = details;
9     }
10
      Windsurf: Refactor | Explain | Generate JSDoc | ×
11    public getDetails(): string {
12      return `${this.message} - Деталі: ${this.details}`;
13    }
14  }
15
16  const customError = new CustomAppError(400, "Невірний запит", ["Помилка в даних"], "Перевірте поля");
17  console.log(customError.getDetails());
18  console.log(customError.status);
```

Рисунок 2.7 – Фрагмент коду з переліком доступних послуг для запису до майстра

У цьому прикладі клас CustomAppError успадковує AppError, додаючи приватну властивість details і метод getDetails для доступу до неї. Виклик super() передає параметри до конструктора батьківського класу.

TypeScript завдяки своїй типізації та підтримці ООП забезпечує надійність і масштабованість коду, що робить його ідеальним вибором для проєктів, таких як веб-сайт барбершопу, де важливо чітко структурувати дані про майстрів, послуги та записи.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Інтегроване середовище розробки

Інтегроване середовище розробки (IDE) — це спеціалізована програма, призначена для полегшення створення, редагування та тестування програмного забезпечення розробниками. У процесі розробки веб-сайту барбершопу я використав Visual Studio Code (VSCode), розроблений компанією Microsoft. VSCode є безкоштовним редактором із відкритим вихідним кодом, доступним на всіх основних операційних системах, що робить його зручним інструментом для роботи. Завдяки своїй інтуїтивності, потужним функціям і великій кількості розширень, VSCode став одним із моїх улюблених інструментів серед розробників.

Основні переваги та можливості VSCode, які я використав під час роботи, включають:

- **Підтримка мов програмування:** VSCode підтримує широкий спектр мов, таких як TypeScript, JavaScript, Python, C++, Java та інші, що дало мені гнучкість у реалізації як фронтенд-, так і бекенд-частин проєкту.
- **Підсвічування синтаксису та автодоповнення:** Редактор автоматично підсвічує синтаксис, підвищуючи читабельність коду, а також пропонує варіанти автодоповнення, що прискорило мій процес кодування.
- **Інтелектуальне завершення коду:** Аналізуючи написаний код, VSCode пропонує релевантні підказки для завершення, що допомогло мені уникнути типових помилок і заощадити час.
- **Рефакторинг коду:** Інструменти рефакторингу дозволили мені оптимізувати код, роблячи його більш структурованим і легким для подальшого супроводу.
- **Вбудований Git:** Інтеграція з Git спростила мені управління версіями проєкту, дозволяючи відстежувати зміни та співпрацювати над кодом.

- **Розширення:** Завдяки великій екосистемі розширень я додав підтримку TypeScript, React, MongoDB та інших інструментів, адаптувавши VSCode під мої потреби в розробці.

Використання VSCode у поєднанні з відповідними розширеннями значно підвищило продуктивність і комфорт під час створення веб-сайту барбершопу, дозволяючи ефективно вирішувати завдання на всіх етапах розробки.

3.2 Проектування баз даних

Для реалізації веб-сайту барбершопу було спроєктовано базу даних, яка забезпечує зберігання та ефективну взаємодію між різними сутностями, такими як користувачі, майстри, послуги та записи на візити. База даних побудована з використанням MongoDB — NoSQL бази даних, яка ідеально підходить для проєктів із гнучкою структурою даних, таких як цей. MongoDB дозволяє легко масштабувати дані та працювати з документами у форматі JSON, що спрощує інтеграцію з JavaScript і TypeScript, які використовуються в проєкті. Для роботи з MongoDB застосовується бібліотека Mongoose, яка забезпечує зручне створення схем, валідацію даних і зв'язки між колекціями.

Структура бази даних

База даних складається з кількох основних колекцій, які відповідають за різні аспекти функціоналу веб-сайту барбершопу. Кожна колекція представляє певну сутність і містить відповідні поля для зберігання даних. Нижче наведено опис основних колекцій, їхньої структури та зв'язків між ними, що відображено на діаграмі (див. рисунок 3.1).

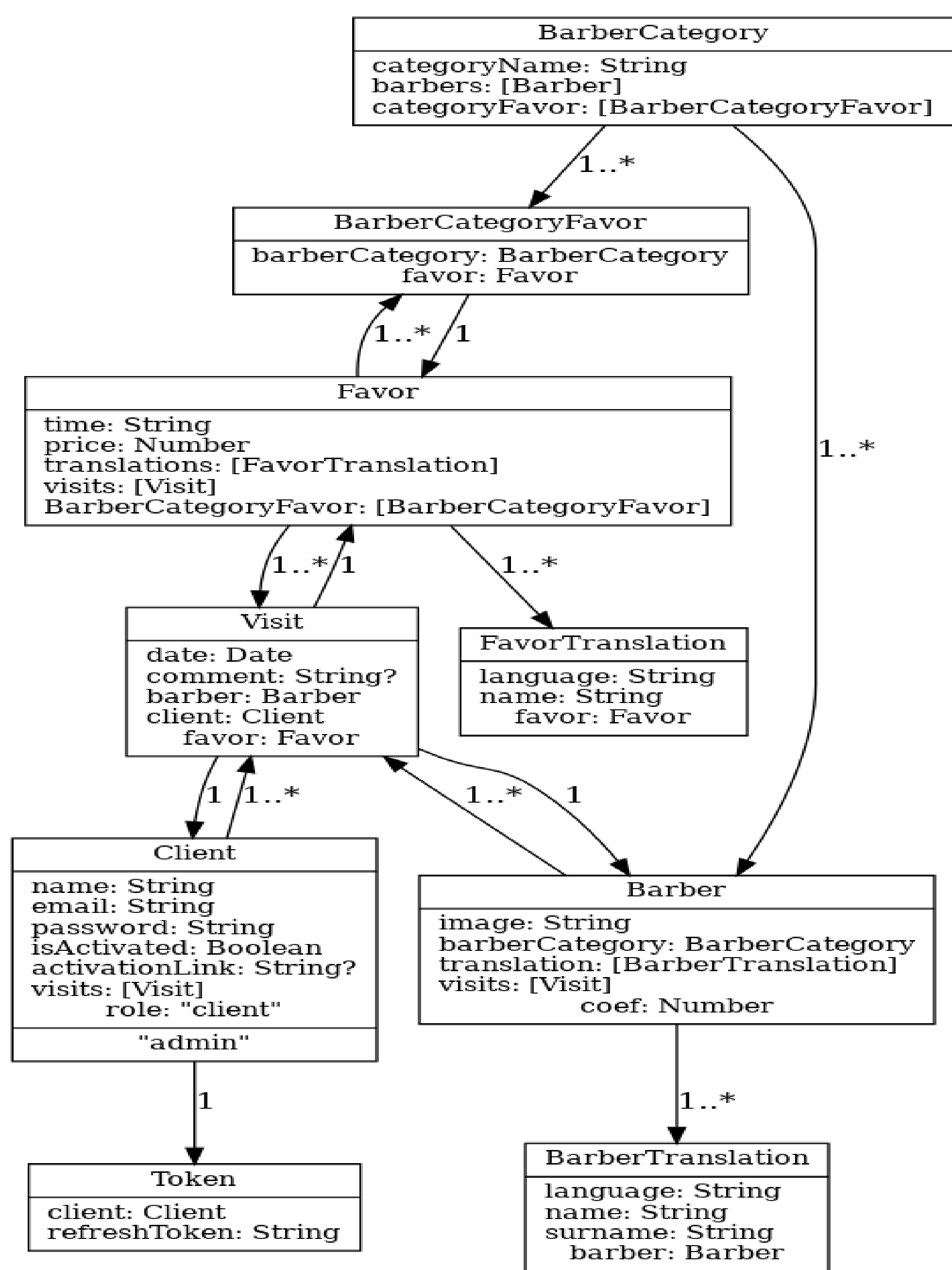

Рисунок 3.1 – Діаграма класів

Колекція Client

Колекція Client зберігає інформацію про користувачів, які реєструються на сайті для запису до майстрів.

Поля:

1. name: String — ім’я користувача (обов’язкове).
2. email: String — електронна пошта (обов’язкове, унікальне, проіндексоване).

3. password: String — пароль користувача (обов’язкове, захешоване за допомогою bcryptjs).
4. role: String — роль користувача (enum: ["client", "admin"], за замовчуванням "client").
5. visits: ObjectId[] — масив посилань на записи користувача в колекції Visit.

Зв’язки:

- Один до багатьох із колекцією Visit через поле visits (користувач може мати багато записів).
- Один до багатьох із колекцією Token через поле client у Token (для авторизації).

Колекція Visit

Колекція Visit зберігає інформацію про записи клієнтів до майстрів на певний час і послуги.

- **Поля:**
    1. date: Date — дата і час візиту (обов’язкове, проіндексоване).
    2. comment: String — коментар до запису (необов’язкове).
    3. barber: ObjectId — посилання на майстра з колекції Barber (обов’язкове).
    4. client: ObjectId — посилання на клієнта з колекції Client (обов’язкове).
    5. favor: ObjectId — посилання на послугу з колекції Favor (обов’язкове).
- **Зв’язки:**
    1. Багато до одного з Client, Barber і Favor через поля client, barber, favor.

Колекція Barber

Колекція Barber містить дані про майстрів, які надають послуги в барбершопі.

- **Поля:**

   1. image: String — посилання на зображення майстра (необов'язкове).
   2. barberCategory: ObjectId — посилання на категорію майстра з колекції BarberCategory (обов'язкове).
   3. translation: ObjectId[] — масив посилань на переклади з колекції BarberTranslation.
   4. visits: ObjectId[] — масив посилань на записи майстра в колекції Visit.
   5. coef: Number — коефіцієнт для розрахунку ціни (обов'язкове).
- **Зв'язки:**
   1. Один до багатьох із Visit через поле visits.
   2. Багато до одного з BarberCategory через поле barberCategory.
   3. Один до багатьох із BarberTranslation через поле translation.

Колекція Favor

Колекція Favor зберігає інформацію про доступні послуги в барбершопі.

- **Поля:**
   1. time: Number — тривалість послуги в хвилинах (обов'язкове).
   2. price: Number — базова ціна послуги (обов'язкове).
   3. translations: ObjectId[] — масив посилань на переклади з колекції FavorTranslation.
   4. visits: ObjectId[] — масив посилань на записи, пов'язані з цією послугою, в колекції Visit.
- **Зв'язки:**
   1. Один до багатьох із Visit через поле visits.
   2. Один до багатьох із FavorTranslation через поле translations.
   3. Багато до багатьох із BarberCategory через допоміжну колекцію BarberCategoryFavor.

Допоміжні колекції

- **BarberCategory:** Зберігає категорії майстрів (наприклад, "Barber", "Top Barber").

    - Поля: name: String, favors: ObjectId[] (посилання на BarberCategoryFavor).
- **BarberTranslation та FavorTranslation:** Використовуються для локалізації (перекладів) назв майстрів і послуг.
    - Поля: language: String, name: String, посилання на Barber або Favor.
- **BarberCategoryFavor:** Допоміжна колекція для зв’язку між BarberCategory і Favor.
    - Поля: barberCategory: ObjectId, favor: ObjectId.
- **Token:** Зберігає токени для авторизації (refresh-токени).
    - Поля: client: ObjectId, refreshToken: String.

Особливості проєктування

1. **NoSQL-підхід:** Використання MongoDB дозволяє зберігати дані у вигляді документів, що забезпечує гнучкість при зміні структури даних (наприклад, додавання нових полів до Favor чи Barber).
2. **Зв’язки:** Mongoose дозволяє створювати зв’язки між колекціями через ObjectId і поле ref, що спрощує вибірку пов’язаних даних (наприклад, отримання всіх візитів клієнта).
3. **Індексація:** Поля, які часто використовуються в запитах (наприклад, email у Client, date у Visit), проіндексовані для прискорення пошуку.
4. **Локалізація:** Колекції BarberTranslation і FavorTranslation забезпечують підтримку багатомовності, дозволяючи зберігати переклади для різних мов.

3.3 Розробка серверної частини

Як було зазначено раніше, серверна частина веб-сайту барбершопу розроблена для автоматизованого управління базою даних MongoDB, обробки запитів від клієнтської частини через REST API та забезпечення функціоналу, такого як реєстрація користувачів, управління записами, майстрами та послугами. Основна мета — створити стабільну, масштабовану та безпечну серверну інфраструктуру, яка інтегрується з фронтендом і забезпечує зручний доступ до даних.

Серверна частина побудована з використанням Node.js та фреймворку Express.js, що дозволяє створювати швидкий і гнучкий API. Для взаємодії з базою даних MongoDB використовується бібліотека Mongoose, яка забезпечує зручне визначення схем і зв'язків між колекціями. Керування авторизацією реалізовано через JSON Web Tokens (JWT), а для обробки помилок та валідації даних застосовуються кастомні middleware-функції.

Архітектура та основні компоненти

Серверна частина включає кілька ключових модулів і класів, які реалізують основний функціонал:

- **Налаштування сервера:** Основний файл (app.ts) ініціалізує Express-додаток, підключає middleware такі як: cors, cookieParser, express.json і встановлює з'єднання з базою даних MongoDB.
- **Моделі даних:** Визначено схеми для колекцій, таких як Client, Barber, Favor, Visit тощо, з використанням Mongoose.
- **Сервіси:** Клас ClientService, VisitService, TokenService та MailService реалізують бізнес-логіку, таку як реєстрація, авторизація, відправка email та управління записами.
- **Контролери:** Ресурси API (наприклад, clientRouter, barberRouter) обробляють HTTP-запити та викликають відповідні методи сервісів.
- **Middleware:** Функції валідації (authMiddleware, favorValidation) та обробки помилок (errorMiddleware) забезпечують безпеку та коректність даних.

Налаштування сервера

Сервер ініціалізується в головному файлі з використанням Express.js. Підключення до MongoDB налаштовується через змінну середовища DB_URL, а порт визначається через PORT (за замовчуванням 5000). Middleware, такі як cors із підтримкою credentials: true, дозволяють фронтенду (http://localhost:5173) надсилати запити з кукісами. Обмеження розміру JSON-запитів встановлено на 500 кб для оптимізації.

Приклад ініціалізації сервера:

```
19  const app = express();
20  app.use(express.json({ limit: "500kb" }));
21  app.use(cookieParser());
22  app.use(
23    cors({
24      origin: process.env.CLIENT_URL || "http://localhost:5173" ,
25      credentials: true,
26    })
27  );
    Windsurf: Refactor | Explain | Generate JSDoc | ×
28  const main = async () => {
29    try {
30      await mongoose.connect(DB_URL);        You, 1 second ago • Uncommitted
31      console.log("Connected to DB");
32
33      app.listen(PORT, () => {
34        console.log(`Server started on port ${PORT}`);
35      });
36    } catch (e) {
37      console.log(e);
38    }
39  };
```

Рисунок 3.3 – Ініціалазія сервера та з’єднання з MongoDB

Обробка помилок

Для обробки помилок створено клас AppError, який розширює стандартний Error і включає статус-код та масив помилок. Middleware errorMiddleware обробляє винятки та повертає відповідний JSON-відповідь:

```
class AppError extends Error {
  status: number;
  errors: string[];
  constructor(status: number, message: string, errors = []) {
    super(message);
    this.status = status;
    this.errors = errors;  }

  static UnauthorizedError() {
    return new AppError(401, "User is not authorized");
  }
  static BadRequest(message:string, errors = []) {
    return new AppError(400, message, errors);
  }
  static ForbiddenError() {
    return new AppError(403, "Access forbidden");
  }
}

export default AppError;
```

Рисунок 3.4 - Створено клас AppError для обробки помилок

Реалізація API

REST API реалізовано за допомогою маршрутів, розподілених по модулях (наприклад, clientRouter, barberRouter). Кожен роутер обробляє специфічні ресурси:

- **/api/clients:** Реєстрація, логін, оновлення пароля, активація акаунту.
- **/api/barbers:** Додавання, оновлення та видалення майстрів.
- **/api/favors:** Управління послугами.
- **/api/visits:** Створення та отримання записів.

Приклад маршруту для майстрів:

```
import express from "express";
import { addBarber, deleteBarber, getBarbers, updateBarber } from "../controllers/barbers";
import { barberValidation } from "../middleware/barberValidation";
import { upload } from "../config/cloudinaryConfig";

const barberRouter = express.Router();

barberRouter.get("/", getBarbers);
barberRouter.post("/", barberValidation, addBarber);
barberRouter.delete("/:id", deleteBarber);
barberRouter.put("/:id",upload.single("image"), updateBarber);

export { barberRouter };

```

Рисунок 3.5 - REST API реалізація маршрутів для

Сервіси та бізнес-логіка

- **Клас ClientService:** Реалізує реєстрацію з генерацією JWT-токенів, активацію акаунту через email (використовуючи mailService) та зміну пароля. Паролі хешуються за допомогою bcryptjs.
- **Клас VisitService:** Додає нові записи, оновлює зв'язки між колекціями (Client, Barber, Favor) та відправляє сповіщення про запис.
- **Клас TokenService:** Генерує та валідує токени (access та refresh), використовуючи jsonwebtoken.
- **Клас MailService:** Використовує nodemailer для відправки email-повідомлень про активацію, записи та зміну пароля.

Авторизація та аутентифікація

Авторизація в системі реалізовано через комбінацію middleware, сервісів і контролерів, що забезпечує безпечний доступ до ресурсів API. Основна логіка авторизації базується на використанні JSON Web Tokens (JWT), які генеруються, валідуються та зберігаються за допомогою класу TokenService. Middleware authMiddleware виконує початкову перевірку наявності та валідності токена в заголовку Authorization, але остаточна обробка авторизованих запитів відбувається в контролерах і сервісах.

- **Генерація та валідація токенів:** Клас TokenService відповідає за створення access- і refresh-токенів за допомогою бібліотеки jsonwebtoken. Access-токен має термін дії 15 хвилин, а refresh-токен —

30 днів. Наприклад, метод generateTokens генерує пару токенів на основі даних клієнта:

```
You, last month | 1 author (You) | Windsurf: Refactor | Explain
4   class TokenService {
      Windsurf: Refactor | Explain | Generate JSDoc | ×
5     generateTokens(payload: any) {
6       const accessToken = jwt.sign(payload, process.env.JWT_ACCESS_SECRET, {
7         expiresIn: "15m",
8       });
9       const refreshToken = jwt.sign(payload, process.env.JWT_REFRESH_SECRET, {
10        expiresIn: "30d",
11      });
12
13      return { accessToken, refreshToken };
14    }
```

Рисунок 3.6 – Метод для генерації токенів

Валідація токенів виконується через методи validateAccessToken і validateRefreshToken, які повертають дані користувача або null у випадку невалідного токена.

Зберігання токенів: Refresh-токени зберігаються в базі даних MongoDB у колекції Token, пов’язаній із моделлю Client через поле user. Метод saveToken оновлює або створює запис:

```
49    async saveToken(userId: string, refreshToken: string) {     You, 3 mont
50      const tokenData = await Token.findOne({ user: userId });
51
52      if (tokenData) {
53        tokenData.refreshToken = refreshToken;
54        return tokenData.save();
55      }
56      const token = await Token.create({ user: userId, refreshToken });
57      return token;
58    }
```

Рисунок 3.7 – Метод, який оновлює або створює запис

Middleware authMiddleware: Цей middleware перевіряє наявність токена в заголовку Authorization і валідує його за допомогою TokenService. Якщо токен валідний, дані користувача додаються до об’єкта запиту (req.client), і запит передається до відповідного контролера:

```
15  export const authMiddleware = async (
16    req: Request & AuthRequest,
17    res: Response,
18    next: NextFunction
19  ) => {
20    try {
21      const authorizationHeader = req.headers.authorization;
22      if (!authorizationHeader) {
23        return next(AppError.UnauthorizedError());
24      }
25
26      const accessToken = authorizationHeader.split(" ")[1];
27      if (!accessToken) {
28        return next(AppError.UnauthorizedError());
29      }
30
31      const clientData = tokenService.validateAccessToken(accessToken);
32      if (!clientData) {
33        return next(AppError.UnauthorizedError());
34      }
35
36      req.client = clientData;
37      next();
38    } catch (e) {
39      return next(AppError.UnauthorizedError());
40    }
41  }
42
```

Рисунок 3.8 – Middleware, який перевіряє наявність токена

Обробка в контролері та сервісі: Логіка входу (логін) і оновлення токенів реалізовано в ClientService і ClientController. Наприклад, метод login у ClientService перевіряє пароль, генерує токени та зберігає їх:

```
async login(email: string, password: string) {
  const client = await Client.findOne({ email }).select("+password");

  if (!client) {
    throw AppError.BadRequest("Client not found");
  }

  const isPassEquals = await bcrypt.compare(password, client.password);
  if (!isPassEquals) {
    throw AppError.BadRequest("Incorrect password");
  }

  const clientDto = new ClientDto(client);
  const tokens = tokenService.generateTokens({ ...clientDto });

  await tokenService.saveToken(clientDto.id, tokens.refreshToken);

  return { ...tokens, client: clientDto };
}
```

Рисунок 3.9 – Сервісна логіка входу (login)

У ClientController токени зберігаються в кукісах і повертаються клієнту:

```
async login(req: Request, res: Response, next: NextFunction) {
  try {
    const { email, password } = req.body;
    const clientData = await clientService.login(email, password);
    res.cookie("refreshToken", clientData.refreshToken, {
      maxAge: 30 * 24 * 60 * 60 * 1000,
      httpOnly: true,
    });
    return res.json(clientData);
  } catch (e) {
    next(e);
  }
}
```

Рисунок 3.10 – метод у контролері (login)

Оновлення токенів: Метод refresh у ClientService використовує refresh-токен для генерації нової пари токенів, якщо попередній токен валідний:

```
async refresh(refreshToken: string) {
  if (!refreshToken) {
    throw AppError.UnauthorizedError();
  }

  const clientData = tokenService.validateRefreshToken(refreshToken);
  const tokenFromDB = await tokenService.findToken(refreshToken);
  if (!clientData || !tokenFromDB) {
    throw AppError.UnauthorizedError();
  }

  const client = await Client.findById(clientData.id).select("+role");
  const clientDto = new ClientDto(client);
  const tokens = tokenService.generateTokens({ ...clientDto });

  await tokenService.saveToken(clientDto.id, tokens.refreshToken);


  return { ...tokens, client: clientDto };
}
```

Рисунок 3.11 – Оновлення токенів

Таким чином, авторизація є результатом інтеграції authMiddleware (для початкової перевірки), TokenService (для управління токенами) і ClientService/ClientController (для логіки входу та оновлення). Це дозволяє забезпечити безпечний доступ до захищених ресурсів, таких як /api/visits або /api/clients/me.

3.4 Розробка клієнтської частини

Для реалізації клієнтської частини веб-сайту барбершопу використано бібліотеку React із мовою програмування TypeScript, що забезпечує типізацію та масштабованість коду. Клієнтська частина розроблена як Single Page Application (SPA), що дозволяє швидко завантажувати сторінки без повного перезавантаження. Для управління станом використано Redux Toolkit, а для маршрутизації — бібліотека React Router. Стилізація компонентів виконана за допомогою CSS із підтримкою адаптивного дизайну через Tailwind CSS (хоча у коді вказано використання прямих CSS-класів). Для локалізації застосовано бібліотеку react-i18next, що забезпечує підтримку української та англійської мов. Анімація реалізована через бібліотеку Framer Motion, яка додає плавні переходи та ефекти.

Клієнтська частина інтегрується з сервером через REST API, використовуючи асинхронні запити (наприклад, для отримання списку майстрів, послуг, записів). Для повідомлень користувачу використано бібліотеку react-toastify, яка відображає сповіщення (наприклад, про успішний логін або створення запису).

Основна сторінка

Основна сторінка (HomePage) є головним інтерфейсом для неавторизованих користувачів і слугує для ознайомлення з барбершопом. Вона складається з кількох секцій, кожна з яких реалізує окремий функціонал:

- **Шапка (Header):** Компонент містить навігаційне меню з посиланнями на секції "Про нас", "Майстри", "Контакти". Є кнопка для переходу до

онлайн-запису (Sidebar), а також можливість вибору мови через компонент Language. Для авторизованих користувачів відображається кнопка переходу до профілю, а для адміністраторів — додаткова іконка для доступу до адмін-панелі. Адаптивний дизайн дозволяє відкривати бічне меню на мобільних пристроях.

- **Секція "Про нас" (About):** Відображає загальну інформацію про барбершоп із заголовком і описом, які перекладаються залежно від обраної мови. Використано анімацію Framer Motion для плавного появи тексту при скролі.
- **Список послуг (FavorsList):** Відображає перелік доступних послуг, отриманих із сервера через Redux-дію fetchFavors. Користувач може фільтрувати послуги за категоріями майстрів через випадаючий список (на мобільних пристроях). Для кожної послуги показано її назву, ціну та тривалість через компонент Favors.
- **Список майстрів (BarberList):** Показує картки майстрів із фотографіями та інформацією (ім'я, категорія), отриманими через Redux-дію fetchBarbers. Використовується адаптивна сітка для відображення на різних пристроях.
- **Контакти (Contacts):** Містить інформацію про адресу, графік роботи та номер телефону барбершопу. Вбудовано інтерактивну карту Google Maps із можливістю масштабування та анімацією при скролі через Framer Motion.
- **Футер (Footer):** Відображає іконки для зв'язку) та рік створення сайту, який оновлюється динамічно.

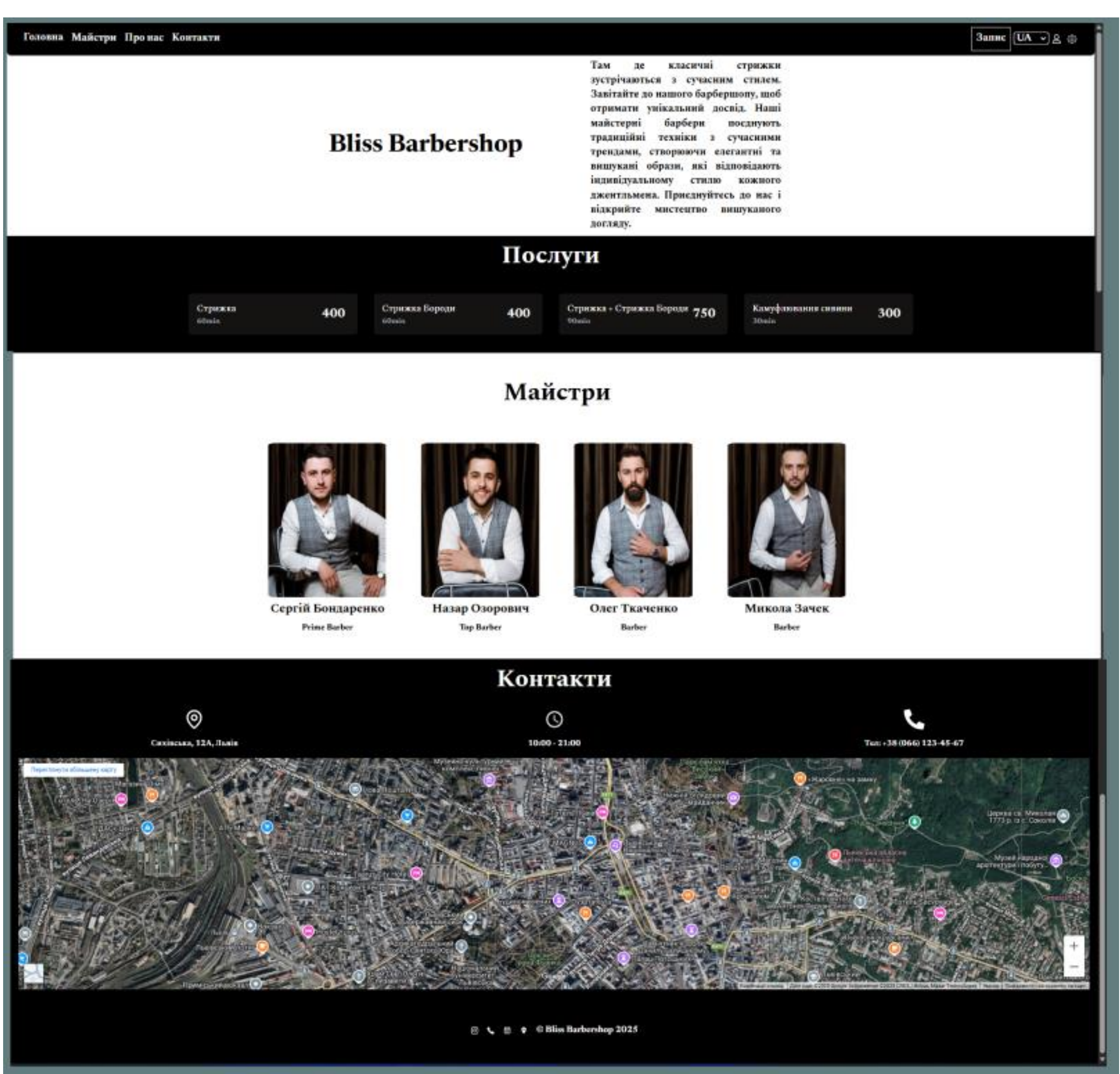

Рисунок 3.12 - Головна сторінка

Логін та реєстрація користувача та профіль

Процес авторизації реалізовано через модальне вікно (LoginPage), яке відкривається при натисканні на кнопку входу в шапці сайту. Форма логіну побудована з використанням бібліотеки Formik, яка забезпечує валідацію полів (email і пароль) відповідно до схеми LoginValidationSchema. Після успішного входу користувач автоматично перенаправляється на сторінку профілю (UserProfile). Якщо користувач ще не зареєстрований, у вікні LoginPage він може перейти до форми реєстрації, після чого, підтвердивши акаунт через email, також буде перенаправлений на сторінку профілю.

- **Сторінка логіну (LoginPage):** Модальна форма містить поля для введення email і пароля, а також посилання "Забули пароль?", яке веде на сторінку відновлення пароля. У разі помилки (наприклад, невірний пароль) користувач отримує повідомлення через бібліотеку react-toastify. Після успішного логіну відображається сповіщення про успіх, і користувач перенаправляється на маршрут /profile.

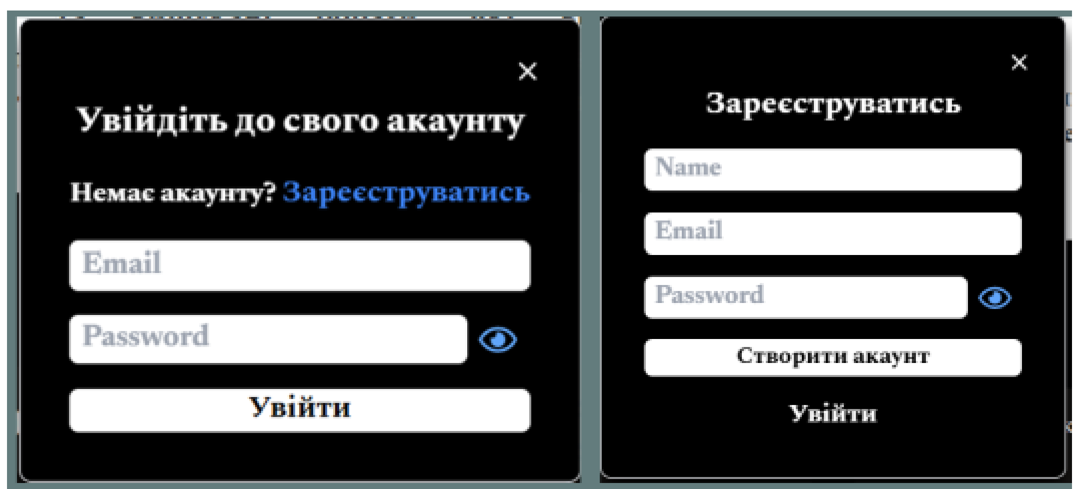

Рисунок 3.13 - Фрагмент логіну та реєстрації

Профіль користувача (UserProfile): Сторінка профілю відображається лише для авторизованих користувачів, які активували свій акаунт. Вона включає:

- **Інформацію користувача:** Ім’я та email, отримані через запит до сервера (ClientService.fetchClient).
- **Список записів (Visit):** Відображає історію записів користувача, отриманих через Redux-селектор visits. Для адміністраторів показуються всі записи (allVisits), а для звичайних користувачів — лише їхні власні. Якщо записів немає, відображається повідомлення "У вас немає записів".
- **Кнопка виходу (Exit):** Дозволяє користувачу вийти з акаунту та повернутися на головну сторінку.
- Анімація через Framer Motion додає плавний ефект появи елементів сторінки при завантаженні.

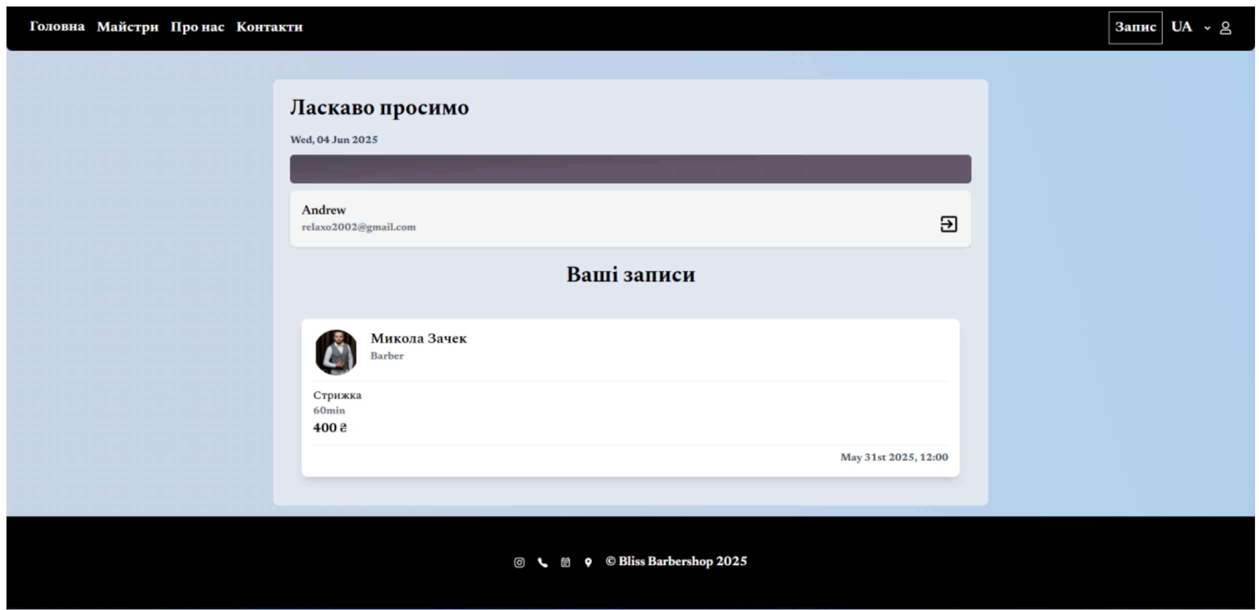

Рисунок 3.14 – Профіль користувача

Онлайн-запис

Функціонал онлайн-запису реалізовано через бічну панель (Sidebar), яка відкривається при натисканні кнопки "Booking" у шапці. Для неавторизованих користувачів відображається модальне вікно для реєстрації (SignupPage). Процес запису складається з кількох кроків:

- **Вибір майстра (Employee):** Користувач обирає майстра зі списку (BarberCards), який фільтрується за категоріями через кнопки (наприклад, "Senior Barber", "Junior Barber"). Картка майстра (BarberCard) показує фото, ім’я, прізвище та категорію, залежно від обраної мови.

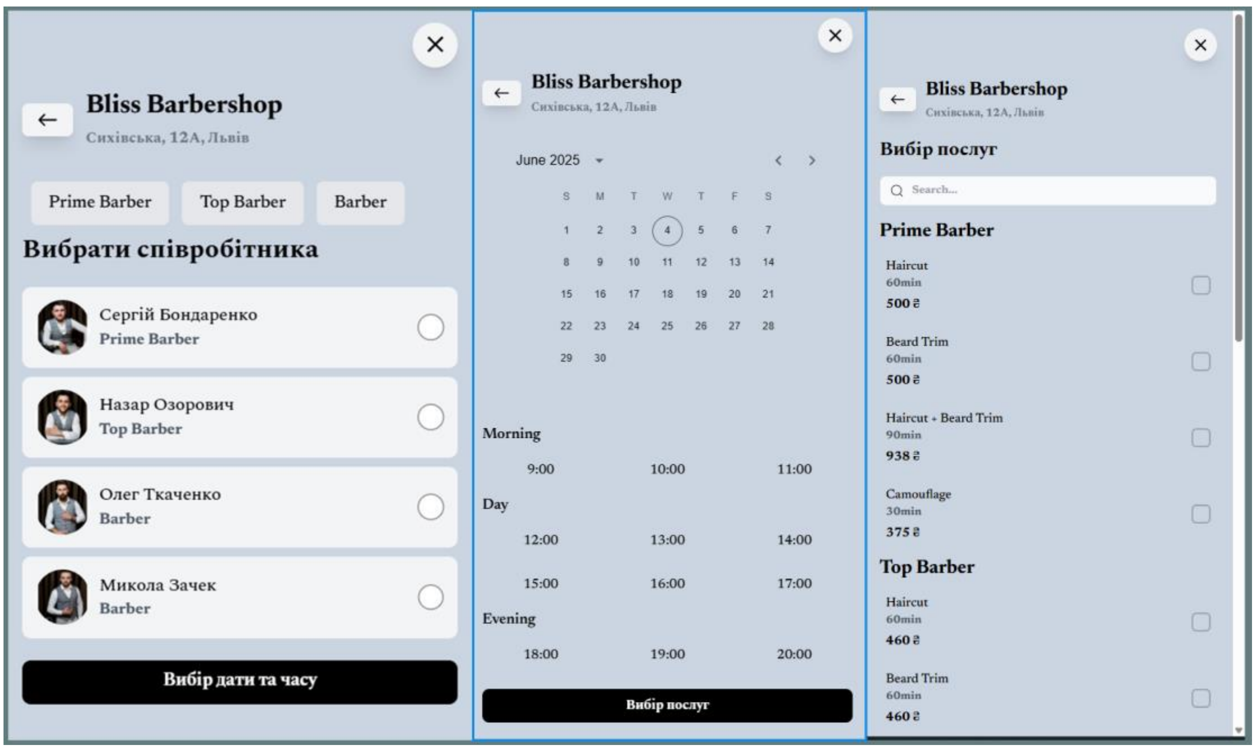

Рисунок 3.15 – Онлайн запис

Адмін-панель

Адмін-панель (AdminPanel) доступна лише користувачам із роллю "admin" і відображається за маршрутом /admin-panel, а також з'являється додаткова кнопка у меню (див. рис. 4.6). Вона дозволяє адміністратору додавати нових майстрів і послуги:

- **Додавання майстра (AddBarber):** Форма включає поля для введення URL зображення, імені та прізвища (англійською та українською), категорії та коефіцієнта. Використовується Formik із валідацією через AddBarberSchema. Після відправки форми дані передаються на сервер через Redux-дію addBarber. Є кнопка для перегляду списку майстрів (AdminEditBarberList).

  Рисунок 4.5 - Форма додавання майстра

  (Скріншот із формою для введення даних майстра.)

- **Додавання послуги (AddFavor):** Форма дозволяє ввести назву послуги (англійською та українською), тривалість і ціну. Дані валідуються через AddFavorSchema, а після відправки передаються на сервер через дію

addFavor. Також є кнопка для перегляду списку послуг (AdminEditFavorList).

Admin Panel

Add New Barber

Image URL

Name (English)

Name (Ukrainian)

Surname (English)

Surname (Ukrainian)

Category

Select a category

Coefficient

1

Add Barber

Barber List

Add New Favor

Name (English)

Name (Ukrainian)

Time

0

Price

0

Add Favor

Favor List

Рисунок 3.16 – Адмін панель

Рисунок 3.17 – Додаткова кнопка для адміністратора

ВИСНОВОК

Було розроблено вебдодаток для барбершопу, який забезпечує простий та зручний інтерфейс для онлайн-запису на послуги. Для взаємодії клієнта з сервером створено REST API, що дозволяє ефективно обробляти запити щодо користувачів, майстрів та послуг.

Під час розробки дипломного проєкту було реалізовано:

- Серверну частину для обробки користувачів та послуг
- Сервер доступу через REST API
- Клієнтський інтерфейс для ознайомлення наданих послуг

СПИСОВ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Node.js Documentation [Електронний ресурс]. – Режим доступу: https://nodejs.org/en/docs/
2. Express.js Guide [Електронний ресурс]. – Режим доступу: https://expressjs.com/
3. MongoDB Manual [Електронний ресурс]. – Режим доступу: https://docs.mongodb.com/
4. React Official Documentation [Електронний ресурс]. – Режим доступу: https://reactjs.org/docs/getting-started.html
5. Redux Toolkit Documentation [Електронний ресурс]. – Режим доступу: https://redux-toolkit.js.org/
6. JWT Introduction. JSON Web Tokens [Електронний ресурс]. – Режим доступу: https://jwt.io/introduction/
7. TypeScript Handbook [Електронний ресурс]. – Режим доступу: https://www.typescriptlang.org/docs/
8. Masse M. REST API Design Rulebook. – O'Reilly Media, 2011. – 130 p.
9. Коваленко В.І. Веброзробка: навчальний посібник. – Київ: Техніка, 2020. – 320 с.
10. Петренко О.М. Основи розробки REST API. – Львів: Видавництво ЛНУ, 2019. – 200 с.
11. Гаврилюк С.П. Реалізація систем авторизації на основі JWT // Інформаційні технології. – 2022. – № 3. – С. 45–53.
12. Luksa M. Scalable Microservices with Kubernetes. – Manning Publications, 2018. – 250 p.
13. Fielding R.T. Architectural Styles and the Design of Network-based Software Architectures: PhD thesis. – University of California, Irvine, 2000.
14. Brown E. Web Development with Node and Express. – O'Reilly Media, 2019. – 350 p.
15. Haverbeke M. Eloquent JavaScript: A Modern Introduction to Programming. – No Starch Press, 2018. – 472 p.
16. Copeland R. MongoDB Applied Design Patterns. – Addison-Wesley, 2013. – 300 p.
17. Schwarzmüller M. React – The Complete Guide [Електронний ресурс]: Udemy Course, 2021. – Режим доступу: https://www.udemy.com/course/react-the-complete-guide/
18. Siriwardena P. OAuth 2.0 and OpenID Connect: The Professional Guide. – Apress, 2017. – 220 p.
19. Crockford D. JavaScript: The Good Parts. – O'Reilly Media, 2008. – 176 p.
20. McDonald M. Web Security for Developers. – No Starch Press, 2017. – 250 p.

Додаток А

Код серверної частини

app.ts

```
import mongoose from "mongoose";
import express from "express";
import cors from "cors";
import cookieParser from "cookie-parser";

import { DB_URL, PORT } from "./config/config";
import { barberRouter } from "./routes/barbers";
import { favorRouter } from "./routes/favors";
import { clientRouter } from "./routes/client";
import { barberTranslationsRouter } from "./routes/barberTranslation";
import { barberCategoryRouter } from "./routes/barberCategory";
import seedRouter from "./routes/seed";
import { visitRouter } from "./routes/visit";
import { favorTranslationsRouter } from "./routes/favorTranslations";
import { errorMiddleware } from "./middleware/errorMiddleware";
import { tempRouter } from "./routes/tempRouter";
import { adminRouter } from "./routes/admin";

const app = express();
app.use(express.json({ limit: "500kb" }));
app.use(cookieParser());
app.use(
  cors({
    origin: process.env.CLIENT_URL || "http://localhost:5173" ,
    credentials: true,
  })
);

app.use("/api/admin", adminRouter);

app.use("/api/barbers", barberRouter);
app.use("/api/barberCategory", barberCategoryRouter);
app.use("/api/barberTranslations", barberTranslationsRouter);
app.use("/api/favors", favorRouter);
app.use("/api/favorTranslations", favorTranslationsRouter);
app.use("/api/visits", visitRouter);
app.use("/api/seed", seedRouter);
app.use("/temp", tempRouter);

app.use("/api/clients", clientRouter);
app.use(errorMiddleware);

const main = async () => {
  try {
    await mongoose.connect(DB_URL);
```

```
    console.log("Connected to DB");

    app.listen(PORT, () => {
      console.log(`Server started on port ${PORT}`);
    });
  } catch (e) {
    console.log(e);
  }
};

main();
```

barber.ts

```
import express from "express";
import { addBarber, deleteBarber, getBarbers, updateBarber } from "../controllers/barbers";
import { barberValidation } from "../middleware/barberValidation";
import { upload } from "../config/cloudinaryConfig";

const barberRouter = express.Router();

barberRouter.get("/", getBarbers);
barberRouter.post("/", barberValidation, addBarber);
barberRouter.delete("/:id", deleteBarber);
barberRouter.put("/:id",upload.single("image"), updateBarber);

export { barberRouter };
```

client.ts

```
import express from "express";
// import client from "../controllers/client";
import client from "../controllers/authController";

import { body } from "express-validator";
import { authMiddleware } from "../middleware/authMiddleware";

const clientRouter = express.Router();

clientRouter.post(
  "/registration",
  body("email").isEmail(),
  body("password").isLength({ min: 6 }),
  client.registration
);

clientRouter.post("/login", client.login);
```

```
clientRouter.post("/logout", client.logout);
clientRouter.post("/confirm-password-change", client.confirmPasswordChange);
clientRouter.post(
  "/request-password-change",
  authMiddleware,
  client.requestPasswordChange
);
// clientRouter.get("/activate/:link");
clientRouter.get("/activate/:link", client.activate);

clientRouter.get("/refresh", client.refresh);
clientRouter.get("/", authMiddleware, client.getClients);
clientRouter.get("/me", authMiddleware, client.getClient);

export { clientRouter };
```

barber.ts

```
import mongoose from "mongoose";

const barbersSchema = new mongoose.Schema({
  image: { type: String, required: true },
  barberCategory: { type: mongoose.Schema.Types.ObjectId, ref: "BarberCategory" },
  translation: [{ type: mongoose.Schema.Types.ObjectId, ref: "BarberTranslation" }],
  visits: [{ type: mongoose.Schema.Types.ObjectId, ref: "Visit" }],
  coef: { type: Number, default: 1.0 },
});


const Barber = mongoose.model("Barber", barbersSchema);
export default Barber;
```

client.ts

```
import mongoose from "mongoose";

export interface IClient extends Document {
  _id: string;
  name: string;
  email: string;
  password: string;
  isActivated: boolean;
  activationLink?: string;
  visits: string[];
  role: "client" | "admin";}

const clientSchema = new mongoose.Schema<IClient>({
```

```
  name: {
    type: String,
    required: true,
    minlength: 2,
  },
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true,
  },
  password: {
    type: String,
    required: true,
    minlength: 6,
    // select: false,
  },

  isActivated: {type: Boolean, default: false},
  activationLink:{
    type: String
  },

  visits: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "Visit",
    },
  ],
  role:{
    type: String,
    enum: ["client", "admin"],
    default: "client"
  }
});

const Client = mongoose.model<IClient>("Client", clientSchema);
export { Client };
```

favors.ts

```
import mongoose from "mongoose";

const favorsSchema = new mongoose.Schema({
  time: {
    type: String,
    required: true,
    minlength: 1,
```

```
    maxlength: 7,
  },
  price: {
    type: Number,
    required: true,
    minlength: 1,
  },

  translations: [
    { type: mongoose.Schema.Types.ObjectId, ref: "FavorTranslation" },
  ],

  visits: [{ type: mongoose.Schema.Types.ObjectId, ref: "Visit" }],

  BarberCategoryFavor: [
    { type: mongoose.Schema.Types.ObjectId, ref: "BarberCategoryFavor" },
  ],
});

export const Favor = mongoose.model("Favor", favorsSchema);
```

token.ts

```
import mongoose from "mongoose";

const tokenSchema = new mongoose.Schema({
    client:{ type: mongoose.Schema.Types.ObjectId, ref: "Client" },
    refreshToken: { type: String, required: true },
});

const Token = mongoose.model("Token", tokenSchema);
export default Token;
```

visit.ts

```
import mongoose from "mongoose";

const visitSchema = new mongoose.Schema({
  date: { type: Date, required: true },
  comment: { type: String, required: false },
  barber: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Barber",
    required: true,
  },
  client: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Client",
```

```
    required: true,
  },
  favor: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Favor",
    required: true,
  },
});

const Visit = mongoose.model("Visit", visitSchema);
export default Visit;
```

barber.ts

```
import { NextFunction, Request, Response } from "express";
import { BarberTranslation } from "../models/barberTranslations";
import { cloudinary } from "../config/cloudinaryConfig";
import Barber from "../models/barbers";
import Visit from "../models/visit";
import BarberCategory from "../models/barberCategory";

const getBarbers = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const barbers = await Barber.find()
      .populate("barberCategory", "categoryName")
      .populate("translation", "language name surname barber")
      .populate("visits", "date comment client")
      .select("image barberCategory coef translation visits")
      .select("-__v");

    if (!barbers || barbers.length === 0) {
      return res.status(404).json({ error: "Barbers not found" });
    }

    res.status(200).json(barbers);
  } catch (e) {
    next(e);
  }
};

const addBarber = async (req: Request, res: Response, next: NextFunction) => {
  try {

    const { image, barberCategory, translation, visits, coef } = req.body;

    // Check if required fields are provided
    if (!image || !barberCategory || !translation || !Array.isArray(translation)) {
      return res.status(400).json({ error: "Missing required fields: image, barberCategory, or translation" });
    }
```

```
    const category = await BarberCategory.findById(barberCategory);
    if (!category) {
      return res.status(404).json({ error: "Barber category not found" });
    }

    const barber = new Barber({
      image,
      barberCategory,
      visits: visits || [],
      coef: coef ?? 1.0,
    });

    const translationDocs = await BarberTranslation.insertMany(
      translation.map((t: any) => ({
        language: t.language,
        name: t.name,
        surname: t.surname,
        barber: barber._id
      }))
    );

    barber.translation = translationDocs.map((t: any) => t._id);
    await barber.save();
    if(!category.barbers.includes(barber._id)){

      category.barbers.push(barber._id);
    }
    await category.save();

    res.status(201).json(barber);
  } catch (e) {
    console.error("Error adding barber:", e);
    next(e);
  }
};

const deleteBarber = async (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  try {
    const { id } = req.params;
    await BarberTranslation.deleteMany({ barber: id });
    await Visit.deleteMany({ barber: id });
    const barber = await Barber.findByIdAndDelete(id);
    if (!barber) {
      return res.status(404).json({ error: "Barber not found" });
    }
```

```
    res.status(200).json(barber);
  } catch (e) {
    next(e);
  }
};

const updateBarber = async (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const { id } = req.params;
  const { image, barberCategory, coef } = req.body;
  try {
    const barber = await Barber.findById(id);
    if (!barber) {
      return res.status(404).json({ error: "Barber not found" });
    }
    let updatedImage = barber.image;

    if (image) {
      updatedImage = image;
    } else if (req.file) {
      const result = await cloudinary.uploader.upload(req.file.path, {
        folder: "barbers",
      });
      updatedImage = result.secure_url;
    }

    const updatedBarber = await Barber.findByIdAndUpdate(
      id,
      {
        image: updatedImage,
        barberCategory: barberCategory ?? barber.barberCategory,
        coef: coef ?? barber.coef,
      },
      { new: true }
    );


    res.status(200).json(updatedBarber);
  } catch (e) {
    next(e);
  }
};

export { getBarbers, addBarber, deleteBarber, updateBarber };
```

visit.ts

```
import { NextFunction, Request, Response } from "express";
import Visit from "../models/visit";
import { Client } from "../models/client";
import Barber from "../models/barbers";
import { Favor } from "../models/favors";
import AppError from "../utils/appError";
import mailService from "../service/mailService";
import VisitService from "../service/visitService";
const getVisits = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const clientId = req.client?.id;
    if(!clientId){
      return next(AppError.BadRequest("Client not found"));
    }
    const visits = await VisitService.getVisits(clientId);

    res.status(200).json(visits);
  } catch (e) {
    next(e);
  }
};
const getAllVisits = async (req: Request, res: Response, next: NextFunction) => {
  try {

    const visits = await VisitService.getAllVisits();

    res.status(200).json(visits);
  } catch (e) {
    next(e);
  }
};

const addVisits = async (req: Request, res: Response, next: NextFunction) => {
  const { date, time, barberId, favorId, comment, clientId } = req.body;
  try {
    const visit = await VisitService.addVisits(
      date,
      time,
      barberId,
      favorId,
      comment,
      clientId
    );

    res.status(200).json(visit);
  } catch (e) {
    next(e);
  }
```

```
};



export { getVisits, getAllVisits, addVisits };
```

mailService.ts

```
import nodemailer, { Transporter } from "nodemailer";
import { format } from "date-fns";
import { uk } from "date-fns/locale";

class MailService {
  private transporter: Transporter;

  constructor() {
    this.transporter = nodemailer.createTransport({
      host: process.env.SMTP_HOST,
      port: Number(process.env.SMTP_PORT),
      secure: false,
      auth: {
        user: process.env.SMTP_USER,
        pass: process.env.SMTP_PASS,
      },
    });
  }

  async sendActivationMail(to: string, link: string) {
    await this.transporter.sendMail({
      from: `"Bliss Barbershop" <${process.env.SMTP_USER}>`,
      to,
      subject: `Activation account on ${process.env.API_URL}`,
      html: `
        <div>
          <h1>Activate your account</h1>
          <a href="${link}">${link}</a>
        </div>
      `,
    });
  }
  async sendRecordInformation(to: string, date: Date) {
    const formattedDate = format(date, "dd MMMM yyyy 'o' HH:mm", {
      locale: uk,
    });
    await this.transporter.sendMail({
      from: `"Bliss Barbershop" <${process.env.SMTP_USER}>`,
      to,
      subject: `Інформація про запис`,
      html: `
```

```
      <div>
        <h1>Добрий день! Ви записані на: ${formattedDate}</h1>
      </div>
     `,
    });
  }
  async sendPasswordChanging(to: string, link: string) {
    await this.transporter.sendMail({
      from: `"Bliss Barbershop" <${process.env.SMTP_USER}>`,
      to,
      subject: `Підтвердіть зміну паролю`,
      html: `
       <div>
       <h2>Ми отримали запит на зміну паролю для вашого акаунту на нашому сайті.</h2>
       <h2>Якщо це були ви, будь ласка, натисніть на посилання нижче для підтвердження зміни паролю:</h2>
         <a href="${link}">${link}</a>
       </div><br>
       <h2>Якщо ви не запитували зміну паролю, ви можете проігнорувати це повідомлення.</h2>
     `,
    });
  }
}

export default new MailService();
```

tokenService.ts

```
import jwt from "jsonwebtoken";
import Token from "../models/token";

class TokenService {
  generateTokens(payload: any) {
    const accessToken = jwt.sign(payload, process.env.JWT_ACCESS_SECRET, {
      expiresIn: "15m",
    });
    const refreshToken = jwt.sign(payload, process.env.JWT_REFRESH_SECRET, {
      expiresIn: "30d",
    });

    return { accessToken, refreshToken };
  }
  generatePasswordChangeToken(payload: { id: string; newPassword: string }) {
   const token = jwt.sign(payload, process.env.JWT_PASSWORD_CHANGE_SECRET
    , {
      expiresIn: "15m",
    });
    return token;
```

```
  }
  validatePasswordChangeToken(token: string) {
    try {
      const payload = jwt.verify(token, process.env.JWT_PASSWORD_CHANGE_SECRET);
      return payload;
    } catch (err) {
      return null;
    }
  }


  validateAccessToken(token: string) {
    try {
      const userData = jwt.verify(token, process.env.JWT_ACCESS_SECRET);
      return userData;
    } catch (err) {
      return null;
    }
  }
  validateRefreshToken(token: string) {
    try {
      const userData = jwt.verify(token, process.env.JWT_REFRESH_SECRET);
      return userData;
    } catch (err) {
      return null;
    }
  }

  async saveToken(userId: string, refreshToken: string) {
    const tokenData = await Token.findOne({ user: userId });

    if (tokenData) {
      tokenData.refreshToken = refreshToken;
      return tokenData.save();
    }
    const token = await Token.create({ user: userId, refreshToken });
    return token;
  }
  async removeToken(refreshToken: string) {
    const tokenData = await Token.deleteOne({ refreshToken });
    return tokenData;
  }

  async findToken(refreshToken: string) {
    const tokenData = await Token.findOne({ refreshToken });
    return tokenData;
  }
}

export default new TokenService();
```

Додаток Б

Код клієнта

AdminPanel.tsx

```
import { useState } from "react";
import AddBarber from "../AddBarber/AddBarber";
import { AddFavor } from "../AddFavor/AddFavor";
import { IoMdExit } from "react-icons/io";
import { useNavigate } from "react-router-dom";

export const AdminPanel = () => {
  const [isOpen, setIsOpen] = useState<Boolean>(false);
  const navigate = useNavigate();

  const handleExit = () => {
    navigate("/");
  };
  return (
    <div className="w-1/2 mx-auto">
      <h2 className="text-center text-3xl font-bold mb-2">Admin Panel</h2>
      <div>
        <button onClick={handleExit}>
          <IoMdExit size={25} />
        </button>
      </div>
      <div className="space-y-4 p-6 rounded-lg shadow-lg bg-slate-200">
        <AddBarber />
        <AddFavor />
      </div>
    </div>
  );
};
```

UserProfile.tsx

```
import { useEffect, useState } from "react";
import { Exit } from "../Exit/Exit";
import { useNavigate } from "react-router-dom";
import { motion } from "framer-motion";
import IClient from "../../interfaces/IClient";
import ClientService from "../../Services/clientService";
import { useTranslation } from "react-i18next";
import { Visit } from "../Visit/Visit";
import { useAppSelector } from "../../hooks/useAppSelector";
// import { EditUserData } from "../EditUserData/EditUserData";

export const UserProfile: React.FC = () => {
```

```
const [date, setDate] = useState(new Date());
const [user, setUser] = useState<IClient | null>(null);
const { t } = useTranslation();
const navigate = useNavigate();

const { visits } = useAppSelector((state) => state.visits);
const { isAuth, client } = useAppSelector((state) => state.auth);

useEffect(() => {
  const fetchClient = async () => {
    const token = localStorage.getItem("token");
    if (!token) {
      navigate("/");
      return;
    }

    try {
      const response = await ClientService.fetchClient(token);

      if (response?.data) {
        setUser(response.data);
      } else {
        setUser(null);
      }
    } catch (e) {
      console.log(e);
      setUser(null);
    }
  };

  fetchClient();
  const interval = setInterval(() => {
    setDate(new Date());
  }, 10000);
  return () => clearInterval(interval);
}, [navigate]);

const cardAnimation = {
  hidden: {
    x: -100,
    opacity: 0,
  },
  visible: {
    x: 0,
    opacity: 1,
    transition: { delay: 0.4, duration: 0.55 },
  },
};
const textAnimation = {
  hidden: {
```

```
    x: -100,
    opacity: 0,
  },
  visible: (custom: number) => ({
    x: 0,
    opacity: 1,
    transition: { delay: custom * 0.47 },
  }),
 };
 return (
  <motion.div
    initial="hidden"
    whileInView={"visible"}
    viewport={{ once: true }}
    variants={cardAnimation}
    className="text-black flex justify-center "
  >
    <div className="max-w-[1000px] w-full  bg-slate-200 rounded-lg p-6">
      <div>
        <motion.h1
          variants={textAnimation}
          custom={1}
          className="text-3xl mb-4"
        >
          {t("welcome")}
        </motion.h1>
        <motion.p
          variants={textAnimation}
          custom={2}
          className="text-gray-700 text-sm"
        >
          {date.toLocaleDateString("en-GB", {
            weekday: "short",
            day: "2-digit",
            month: "short",
            year: "numeric",
          })}
        </motion.p>
      </div>
      <hr />

      <motion.div
        variants={textAnimation}
        custom={3}
        className="mt-2 mb-2 w-full rounded-md h-10 bg-gradient-to-br from-[#4f4654] via-[#625867] to-
[#625867]"
      ></motion.div>

      <hr />
```

```
    <div className="flex flex-col gap-6">

     <motion.div
      variants={textAnimation}
      custom={4}
      className="flex items-center space-x-4 p-4 bg-gray-100 rounded-lg shadow"
     >
      <div className="flex-grow">
       <h2 className="text-lg font-semibold">{user?.name}</h2>

       <p className="text-gray-500 text-sm">{user?.email}</p>
      </div>
      <div>
       <div
        className="flex justify-end mt-4"
        onClick={() => navigate("/")}
       >
        <Exit setIsExitVisible={() => {}} />
       </div>
      </div>
     </motion.div>

     <motion.h1
      variants={textAnimation}
      custom={5}
      className="text-3xl text-center"
     >
      {isAuth && client?.role === "admin" ? t("allVisits") : visits.length === 0 ? t("areVisits") : t("visits") }
     </motion.h1>
     <motion.div
      variants={textAnimation}
      custom={5}
      className="flex flex-col gap-4 items-center justify-center"
     >
      <div className="p-4 w-full">
       <Visit />
      </div>
     </motion.div>
    </div>
   </div>
  </motion.div>
 );
};
```