БАКАЛАВРСЬКА РОБОТА

БР.КІ-40.00.00.000 ПЗ

Група КІ-22-1

Винник Владислав

2026

Міністерство освіти і науки України

Івано-Франківський національний технічний університет нафти і газу
Інститут інформаційних технологій

Кафедра комп'ютерних систем і мереж

Винник Владислав Іванович

УДК 004.02

БАКАЛАВРСЬКА РОБОТА

Розробка web-орієнтованої системи менеджменту консультаційних послуг на базі платформи AWS

Комп'ютерна інженерія
(назва освітньої програми)

123 – Комп'ютерна інженерія
(шифр і назва спеціальності)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Здобувач освітнього ступеня _____ *Винник В.І.*
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____ *Слабінога М.О., доцент* _____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри
д.т.н., професор /С. І. Мельничук/
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026 рік

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*

Кафедра *Комп'ютерних систем і мереж*

Освітньо-кваліфікаційний рівень *бакалавр*

Спеціальність *123 – Комп'ютерна інженерія*

ЗАТВЕРДЖУЮ:

Зав. кафедрою КСМ

С.І. Мельничук

«21 » *квітня* *2026*року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Виннику Владиславу Івановичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи*) Розробка web-орієнтованої системи менеджменту консультаційних послуг на базі платформи AWS*

керівник проекту (роботи) *Слабінога Мар'ян Остапович, доцент* .

затверджені наказом вищого навчального закладу від *21.04.2026 № 180/7*

2. Строк подання студентом проекту (роботи) *11 червня 2026р.*

3. Вихідні дані до роботи *Методичні вказівки, технічна література* .

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *1. Підтримка менеджменту консультаційних послуг як технічна задача. 2. Розробка структури веб-орієнтованої системи менеджменту консультаційних послуг. 3. Реалізація веб-орієнтованої системи менеджменту консультаційних послуг.*

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

6. Консультанти розділів роботи

7. Дата видачі завдання *02 лютого 2026 р.* .

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	*Аналіз предметної області та існуючих рішень для підтримки менеджменту консультаційних послуг.*	*Лютий 2026р*	Виконав
2	*Формування вимог до веб-орієнтованої системи та постановка задачі бакалаврської роботи.*	*Березень 2026р*	Виконав
3	*Проєктування структури системи, бази даних, сценаріїв використання та архітектури розгортання на базі AWS.*	*Квітень 2026р*	Виконав
4	*Програмна реалізація клієнтської та серверної частин системи, API, моделей бази даних і основних функціональних модулів.*	*Травень 2026р*	Виконав
5	*Перевірка функціонування системи, оформлення пояснювальної записки, підготовка графічних матеріалів та презентації до захисту.*	*Червень 2026р*	Виконав

Студент _____ *Винник В.І.*

Керівник роботи _____ *Слабінога М.О.*

АНОТАЦІЯ

У бакалаврській роботі розглянуто процес розробки веб-орієнтованої системи менеджменту консультаційних послуг. Актуальність роботи зумовлена потребою у створенні програмних засобів, які спрощують взаємодію між клієнтами, консультантами та адміністратором системи, а також забезпечують структуроване зберігання даних.

У роботі проаналізовано процес менеджменту консультаційних послуг як технічну задачу, розглянуто існуючі інформаційні рішення у цій сфері та сформульовано вимоги до системи. На етапі проєктування обґрунтовано вибір засобів реалізації, описано структуру системи на базі AWS, базу даних та основні сценарії використання програми.

Практична частина передбачала розробку клієнтської та серверної частин веб-застосунку. Клієнтська частина реалізована з використанням Next.js, React і TypeScript, серверна частина — з використанням FastAPI, SQLAlchemy та PostgreSQL. Для контейнеризації застосовано Docker і Docker Compose, а для розгортання — AWS та Terraform.

Результатом роботи є веб-орієнтована система, яка підтримує основні процеси менеджменту консультаційних послуг і може бути використана як основа для подальшого розвитку програмного продукту.

Ключові слова: веб-система, консультаційні послуги, менеджмент послуг, fastapi, next.js, postgresql, aws, docker, sqlalchemy, terraform.

SUMMARY

This bachelor's thesis focuses on the development of the client-side component consulting services management. The relevance of the work is determined by the need to create software tools that simplify interaction between clients, consultants and the system administrator, as well as provide structured data storage.

The thesis analyzes consulting services management as a technical task, reviews existing information systems in this field and defines the requirements for the developed system. At the design stage, the choice of implementation tools was justified, and the AWS-based system structure, database structure and main use cases were described.

The practical part included the development of the client-side and server-side parts of the web application. The client-side part was implemented using Next.js, React and TypeScript, while the server-side part was implemented using FastAPI, SQLAlchemy and PostgreSQL. Docker and Docker Compose were used for containerization, and AWS together with Terraform was used for deployment.

The result of the work is a web-oriented system that supports the main processes of consulting services management and can be used as a basis for further development of the software product.

Keywords: web system, consulting services, service management, fastapi, next.js, postgresql, aws, docker, sqlalchemy, react.

ЗМІСТ

ВСТУП

Актуальність обраної теми. Сфера консультаційних послуг є однією з важливих складових сучасного ринку послуг. Консультації використовуються у бізнесі, освіті, правовій сфері, фінансах, інформаційних технологіях та багатьох інших напрямах, де клієнту потрібна допомога спеціаліста для прийняття рішень або вирішення конкретної проблеми. Зі зростанням кількості таких послуг виникає потреба не лише в якісній роботі консультантів, а й у зручній організації процесу взаємодії між клієнтом і спеціалістом.

Один із важливих аспектів підтримки консультаційної діяльності — це використання сучасних інформаційних технологій. Веб-орієнтовані системи дозволяють спростити пошук консультантів, зберігати інформацію про користувачів, надавати доступ до профілів спеціалістів, організовувати взаємодію між сторонами та забезпечувати більш структуроване управління даними. Завдяки цьому частина процесів, які раніше виконувалися вручну, може бути автоматизована або значно спрощена.

Питання побудови веб-застосунків, організації клієнт-серверної взаємодії, роботи з базами даних, захисту інформації та розгортання систем у хмарному середовищі розглядаються у працях і технічній документації українських та зарубіжних фахівців у галузі інформаційних систем, програмної інженерії та комп’ютерних мереж. Для систем менеджменту консультаційних послуг ці питання мають практичне значення, оскільки така система повинна не лише відображати інформацію, а й забезпечувати збереження даних, авторизацію користувачів, обробку запитів, підтримку різних ролей і стабільну роботу у веб-середовищі.

Дана тема є актуальною у зв’язку з тим, що користувачі все частіше очікують швидкого доступу до послуг через веб-застосунки. Для клієнта важливо мати можливість переглянути інформацію про консультанта, оцінити доступні послуги та виконати потрібну дію без зайвих етапів. Для консультанта або адміністратора системи важливо мати інструмент, який дозволяє впорядкувати інформацію, зменшити кількість ручної роботи та забезпечити стабільну роботу основних

процесів. Сучасний стан інженерного завдання характеризується активним використанням веб-технологій, клієнт-серверної архітектури, реляційних баз даних, контейнеризації та хмарної інфраструктури.

Зважаючи на це, в рамках бакалаврської роботи було розроблено веб-орієнтовану систему менеджменту консультаційних послуг. Вона призначена для підтримки основних процесів, пов’язаних із роботою користувачів, консультантів та даних про консультаційні послуги. Система поєднує клієнтську частину, серверну логіку та базу даних, що дозволяє забезпечити взаємодію між користувачем і програмним продуктом у веб-середовищі. Окрема увага в роботі приділяється структурі системи, сценаріям використання, базі даних, програмній реалізації та розгортанню на базі платформи AWS.

Об’єкт дослідження. Об’єктом дослідження є процес менеджменту консультаційних послуг в інформаційних системах.

Предмет дослідження. Предметом дослідження є структура, програмна реалізація та сценарії використання веб-орієнтованої системи менеджменту консультаційних послуг.

Мета і завдання роботи – метою бакалаврської роботи є розробка веб-орієнтованої системи менеджменту консультаційних послуг, яка забезпечує підтримку основних процесів роботи користувачів, збереження даних та взаємодію між клієнтською і серверною частинами програмного продукту.

Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати процес менеджменту консультаційних послуг як технічну задачу;

– розглянути існуючі інформаційні рішення у цій сфері;

– сформулювати вимоги до системи;

– обґрунтувати вибір засобів реалізації;

– розробити структуру системи та бази даних;

– описати основні сценарії використання програми;

– реалізувати основні компоненти системи;

– перевірити функціонування розробленого програмного продукту.

Методи дослідження. У роботі використано методи аналізу предметної області, порівняння існуючих рішень, структурного проєктування, моделювання бази даних та практичної перевірки роботи програмного забезпечення. Аналіз предметної області використано для визначення основних процесів менеджменту консультаційних послуг. Порівняння існуючих рішень застосовано для виявлення типових функцій веб-систем у цій сфері. Структурне проєктування та моделювання бази даних використано для побудови логіки системи, її сутностей і зв'язків між ними. Практична перевірка дала змогу оцінити працездатність реалізованого програмного продукту.

Практичне значення отриманих результатів. Розроблена веб-орієнтована система може бути використана як основа для організації консультаційних послуг у веб-середовищі. Вона дозволяє структурувати дані про користувачів, консультантів, консультаційні запити, теги та відгуки, а також забезпечує взаємодію між клієнтською і серверною частинами системи. Практична цінність роботи полягає в тому, що створений програмний продукт демонструє повний цикл розробки веб-системи: від аналізу предметної області та проєктування структури до реалізації, розгортання і перевірки функціонування.

У майбутньому систему можна доповнити новими модулями, покращити інтерфейс користувача, розширити можливості адміністрування, додати повноцінну онлайн-оплату, чат між клієнтом і консультантом та додаткові механізми взаємодії між користувачами.

1 ПІДТРИМКА МЕНЕДЖМЕНТУ КОНСУЛЬТАЦІЙНИХ ПОСЛУГ ЯК ТЕХНІЧНА ЗАДАЧА

1.1 Процес менеджменту консультаційних послуг в інформаційних системах

Сфера консультаційних послуг охоплює різні напрями діяльності, у яких клієнт звертається до спеціаліста для отримання професійної поради, аналізу ситуації, підбору рішення або супроводу певного процесу. Консультації можуть надаватися у сфері бізнесу, права, фінансів, освіти, інформаційних технологій, маркетингу, управління персоналом та інших напрямах. Незалежно від конкретної галузі, процес надання консультаційних послуг має спільні риси: клієнт повинен знайти відповідного спеціаліста, ознайомитися з інформацією про нього, обрати послугу, сформувати запит, отримати відповідь або домовитися про подальшу взаємодію.

Менеджмент консультаційних послуг можна розглядати як сукупність процесів, спрямованих на організацію, облік, контроль і підтримку взаємодії між клієнтами, консультантами та адміністраторами системи. До таких процесів належать збирання і зберігання інформації про користувачів, формування профілів консультантів, підтримка пошуку і перегляду послуг, обробка запитів, управління ролями, збереження історії взаємодії та забезпечення доступу до даних. Якщо ці процеси виконуються вручну або за допомогою розрізнених інструментів, виникають проблеми з обліком інформації, дублюванням даних, повільною обробкою запитів і складністю контролю роботи системи.

Інформаційна система дозволяє впорядкувати ці процеси та перенести значну частину операцій у цифрове середовище. Вона виступає посередником між користувачем, консультантом і даними, які необхідні для організації консультаційної діяльності. Завдяки цьому користувач може швидше отримати доступ до потрібної інформації, а адміністратор або власник системи — мати більш зрозумілий механізм управління даними. Для консультаційних послуг це має

практичне значення, оскільки якість роботи залежить не лише від професійності консультанта, а й від зручності процесу взаємодії.

У найпростішому вигляді процес менеджменту консультаційних послуг складається з декількох етапів. Спочатку користувач заходить у веб-застосунок і отримує доступ до основної інформації про систему. Далі він може переглянути доступних консультантів або послуги, ознайомитися з описом, вибрати потрібний варіант і виконати подальшу дію. Якщо система підтримує авторизацію, користувач може створити обліковий запис, увійти до системи та працювати з персональними даними.

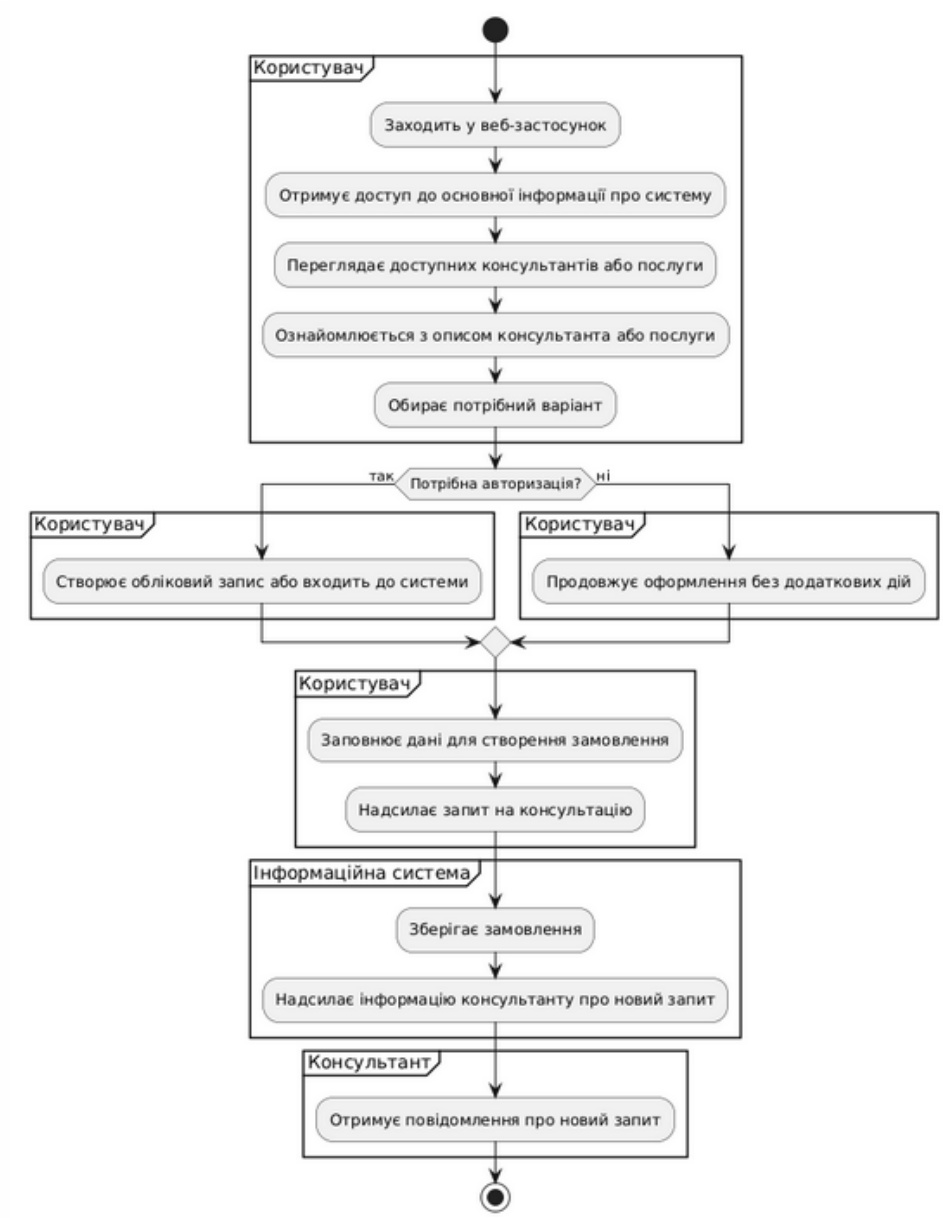

Рисунок 1.1 – Діаграма створення замовлення

Консультант, у свою чергу, може мати власний профіль, де вказано інформацію про його спеціалізацію, досвід, послуги або інші характеристики.

Процес розміщення консультаційної пропозиції зі сторони консультанта починається з авторизації в системі. Після входу консультант заповнює власний профіль, додає опис консультаційної послуги, вказує напрям роботи та вартість консультації. Далі введені дані надсилаються до системи, де вони перевіряються, зберігаються та використовуються для публікації пропозиції. Після цього консультаційна пропозиція відображається у каталозі консультантів і стає доступною для перегляду клієнтами.

Адміністратор може відповідати за підтримку правильності даних, контроль користувачів, оновлення інформації та загальне управління системою.

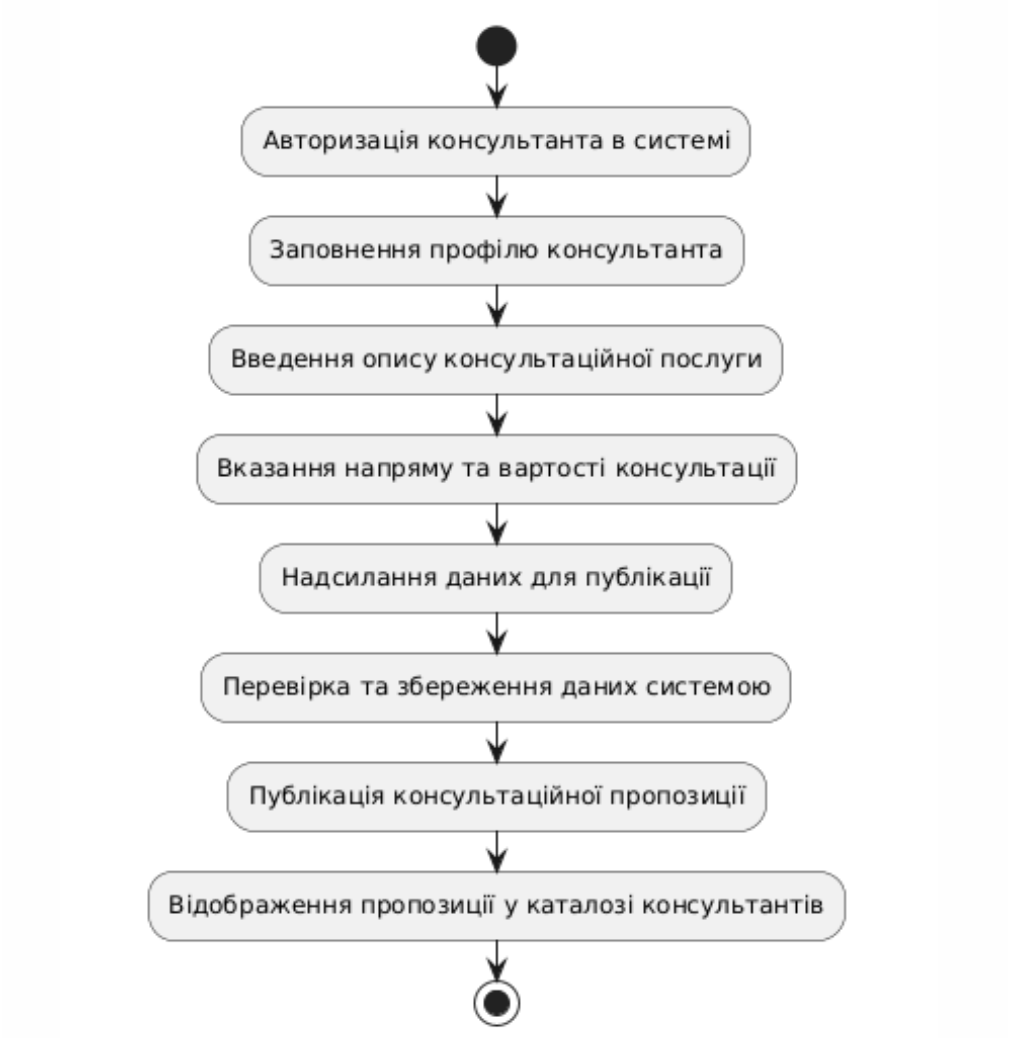

Рисунок 1.2 – Діаграма публікації консультаційної пропозиції

Одним із основних завдань інформаційної системи є централізоване зберігання даних. Для консультаційних послуг це особливо важливо, оскільки система повинна працювати з різними типами інформації: даними користувачів, профілями консультантів, описами послуг, заявками, повідомленнями, транзакціями або іншими об’єктами залежно від функціональності проєкту. Якщо така інформація зберігається у базі даних, її можна структуровано обробляти, швидко отримувати, оновлювати та використовувати в різних частинах системи. Це зменшує ризик втрати даних і дозволяє уникнути хаотичного зберігання інформації у вигляді окремих файлів або таблиць без єдиної логіки.

База даних у такій системі виконує не лише роль сховища, а й основу для побудови логіки роботи програмного продукту. Наприклад, якщо в системі є таблиця користувачів, вона може використовуватися для авторизації, збереження персональної інформації та зв’язку з іншими сутностями. Якщо є таблиця консультантів або профілів, вона може містити дані, які відображаються на сторінках системи. Якщо передбачені заявки або замовлення, вони можуть бути пов’язані з конкретним клієнтом і консультантом. Таким чином, структура бази даних впливає на те, наскільки зручно буде розвивати систему та додавати нові функції.

Не менш важливою частиною є клієнтський інтерфейс. Саме через нього користувач взаємодіє з системою, тому інтерфейс повинен бути зрозумілим, логічним і доступним. Для клієнта важливо швидко зрозуміти, які можливості має система, де знайти потрібного консультанта, як переглянути інформацію та які дії можна виконати далі. Якщо інтерфейс перевантажений, має незрозумілу навігацію або не адаптований до різних пристроїв, це ускладнює використання системи навіть за умови правильно реалізованої серверної частини.

Веб-орієнтований підхід є зручним для реалізації системи менеджменту консультаційних послуг, оскільки користувач може отримати доступ до системи через браузер без необхідності встановлення окремого програмного забезпечення. Це спрощує використання системи як для клієнтів, так і для консультантів. Веб-застосунок може бути доступний з персонального комп’ютера, ноутбука, планшета

або смартфона, що підвищує гнучкість використання. Для сфери послуг це важливо, тому що користувачі часто очікують швидкого доступу до потрібної інформації незалежно від місця перебування.

Серверна частина системи відповідає за обробку запитів, реалізацію бізнес-логіки, роботу з базою даних і передачу даних клієнтській частині. У клієнт-серверній архітектурі frontend відповідає за відображення інтерфейсу та взаємодію з користувачем, а backend — за логіку, перевірку даних, збереження інформації та виконання операцій. Такий поділ дозволяє окремо розвивати інтерфейс і серверну частину, що робить систему більш гнучкою для підтримки та розширення.

Важливим елементом системи є авторизація користувачів. Вона потрібна для того, щоб система могла відрізняти різних користувачів, зберігати їхні дані та надавати доступ лише до дозволених функцій. Наприклад, звичайний клієнт не повинен мати доступ до адміністративних можливостей, а адміністратор може мати розширені права для керування даними. Розмежування доступу є важливим не лише з точки зору зручності, а й з точки зору безпеки. Воно допомагає уникнути ситуацій, коли користувач може випадково або навмисно змінити дані, до яких не повинен мати доступ.

Для консультаційних систем також важливою є цілісність даних. Якщо дані про користувачів, консультантів або послуги зберігаються некоректно, це може призвести до помилок у роботі системи. Наприклад, неправильні зв’язки між сутностями можуть ускладнити відображення інформації, обробку запитів або формування списків. Саме тому під час проєктування потрібно продумати структуру бази даних, типи полів, зв’язки між таблицями та правила обробки даних. Добре спроєктована база даних зменшує кількість помилок і спрощує подальше вдосконалення системи.

Інформаційна система також допомагає зменшити ручну роботу. У традиційному підході менеджер або адміністратор може вручну приймати заявки, вести записи, відповідати на повторювані питання, оновлювати інформацію і контролювати взаємодію між клієнтами та консультантами. Частина цих процесів може бути автоматизована. Наприклад, користувач самостійно переглядає

доступну інформацію, обирає потрібного спеціаліста, працює з власним профілем або надсилає запит через систему. Це зменшує навантаження на адміністратора і робить процес більш швидким.

Окремим завданням є забезпечення доступності системи. Веб-застосунок повинен бути доступним для користувачів у потрібний момент, оскільки консультаційні послуги часто пов’язані з потребою швидко отримати інформацію або підтримку. Якщо система часто недоступна або працює нестабільно, це знижує довіру користувачів. Тому під час розробки потрібно враховувати не лише функціональність, а й питання надійності, продуктивності та стабільності роботи.

Безпека є ще одним важливим аспектом. Система може працювати з персональними даними користувачів, інформацією про профілі, історією взаємодії або іншими даними, які не повинні бути доступними стороннім особам. Тому потрібно передбачати базові механізми захисту: перевірку введених даних, обмеження доступу, безпечне зберігання конфіденційної інформації, використання захищеного з’єднання під час розгортання системи. Навіть якщо система є навчальним проєктом, ці питання потрібно враховувати під час проєктування, оскільки вони впливають на якість програмного продукту.

Менеджмент консультаційних послуг в інформаційній системі також пов’язаний з прозорістю процесів. Коли дані зберігаються в єдиній системі, легше зрозуміти, які користувачі зареєстровані, які консультанти доступні, які послуги представлені та які дії виконуються. Це створює основу для подальшого аналізу роботи системи. У майбутньому на основі таких даних можна додати статистику, звіти, систему оцінювання консультантів або інші інструменти, які покращують управління консультаційною діяльністю.

Ще одним важливим аспектом є масштабованість. На початковому етапі система може містити обмежений набір функцій, але з часом може виникнути потреба додати нові ролі, модулі, способи комунікації, оплату, календар записів або адміністративну панель. Якщо структура системи відразу побудована хаотично, подальше розширення буде складним. Тому технічна задача полягає не лише в

тому, щоб реалізувати поточний функціонал, а й у тому, щоб створити основу для майбутнього розвитку програмного продукту.

Зручність використання також має велике значення. У сфері консультаційних послуг користувач не завжди має високий рівень технічної підготовки, тому система повинна бути інтуїтивною. Основні дії мають виконуватися без зайвих кроків, сторінки повинні мати зрозумілу структуру, а повідомлення про помилки мають допомагати користувачу виправити неправильні дії. Якщо система складна у використанні, користувач може відмовитися від неї навіть за наявності корисного функціоналу.

Таким чином, процес менеджменту консультаційних послуг в інформаційних системах охоплює не лише зберігання даних або створення веб-сторінок. Він передбачає комплексну організацію взаємодії між клієнтами, консультантами, адміністраторами, серверною логікою, базою даних і клієнтським інтерфейсом. У межах такої задачі потрібно враховувати функціональні вимоги, зручність використання, безпеку, доступність, масштабованість і можливість подальшого розвитку системи. Саме тому розробка веб-орієнтованої системи менеджменту консультаційних послуг є повноцінною технічною задачею, яка потребує аналізу предметної області, проєктування структури системи та практичної реалізації програмного продукту.

1.2 Огляд рішень в галузі інформаційних систем для підтримки менеджменту консультаційних послуг

Перед розробкою власної веб-орієнтованої системи доцільно розглянути існуючі рішення, які частково або повністю виконують подібні функції. Аналіз аналогів дозволяє визначити, які можливості є типовими для таких систем, які підходи використовуються для організації взаємодії між користувачами та спеціалістами, а також які недоліки можуть бути враховані під час розробки власного проєкту.

Для порівняння було обрано декілька рішень, пов’язаних із плануванням консультацій, пошуком експертів, взаємодією між клієнтами та фахівцями, а також організацією надання професійних послуг у веб-середовищі. До таких рішень можна віднести Calendly, Clarity.fm, Upwork та MentorCruise. Вони мають різне призначення, але кожне з них демонструє окремий підхід до підтримки процесів, пов’язаних із наданням послуг або консультацій.

Calendly є веб-сервісом для автоматизованого планування зустрічей. Його основна ідея полягає в тому, щоб спростити процес узгодження часу між учасниками. Користувач може створити посилання з доступними часовими слотами, а інша сторона може самостійно обрати зручний час. Такий підхід зменшує кількість повідомлень і ручних погоджень, які зазвичай потрібні для організації зустрічі.

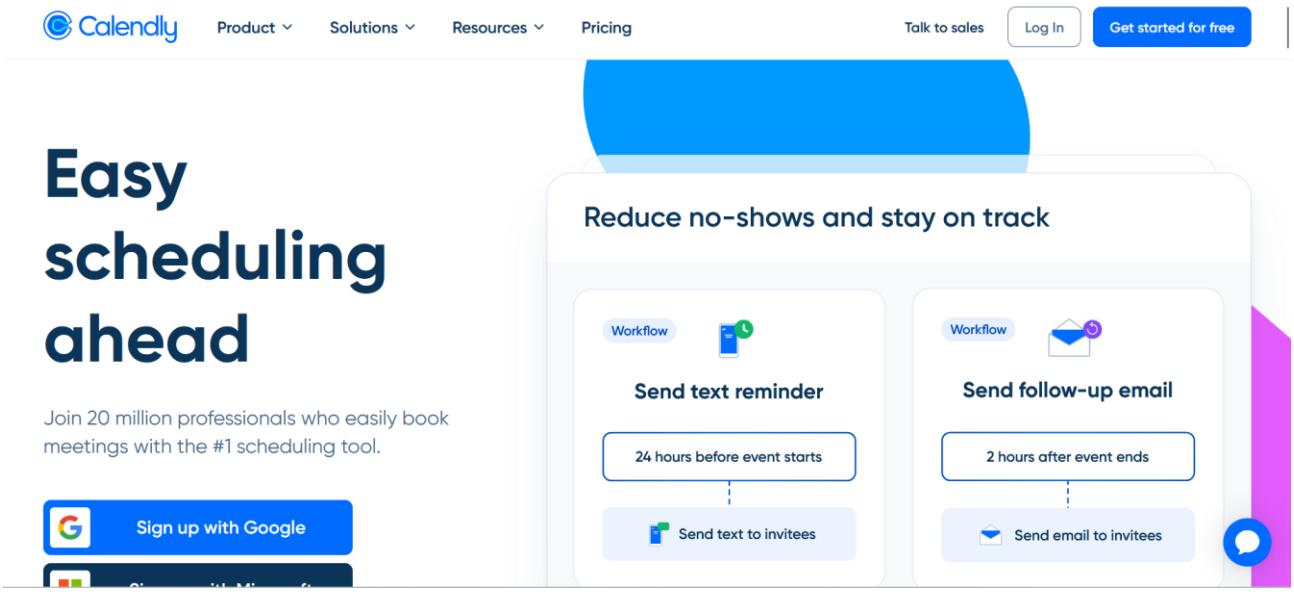

Рисунок 1.3 – Сервіс Calendly

Перевагою Calendly є простота використання. Сервіс вирішує конкретну проблему — організацію часу для зустрічей і консультацій. Він має зрозумілий інтерфейс, підтримує інтеграцію з календарями та може використовуватися як окремими спеціалістами, так і командами. Для консультаційних послуг такий інструмент є корисним, оскільки багато консультацій потребують попереднього запису або бронювання часу.

Недоліком Calendly є те, що він не є повноцінною системою менеджменту консультаційних послуг. Він добре вирішує задачу планування, але не охоплює повний цикл роботи з клієнтами, профілями консультантів, каталогом послуг або внутрішньою логікою конкретної платформи. Тому для комплексної системи його функціональність може бути недостатньою. Власний проєкт може використати як орієнтир ідею простого доступу до дій користувача, але при цьому мати ширшу структуру, пов'язану з профілями, базою даних і сценаріями взаємодії.

Clarity.fm є платформою, що дозволяє користувачам отримувати консультації від експертів. Її логіка побудована навколо пошуку спеціаліста, вибору експерта, узгодження консультації та оплати за розмову. Такий підхід ближчий до теми менеджменту консультаційних послуг, оскільки система безпосередньо пов'язує клієнта з консультантом.

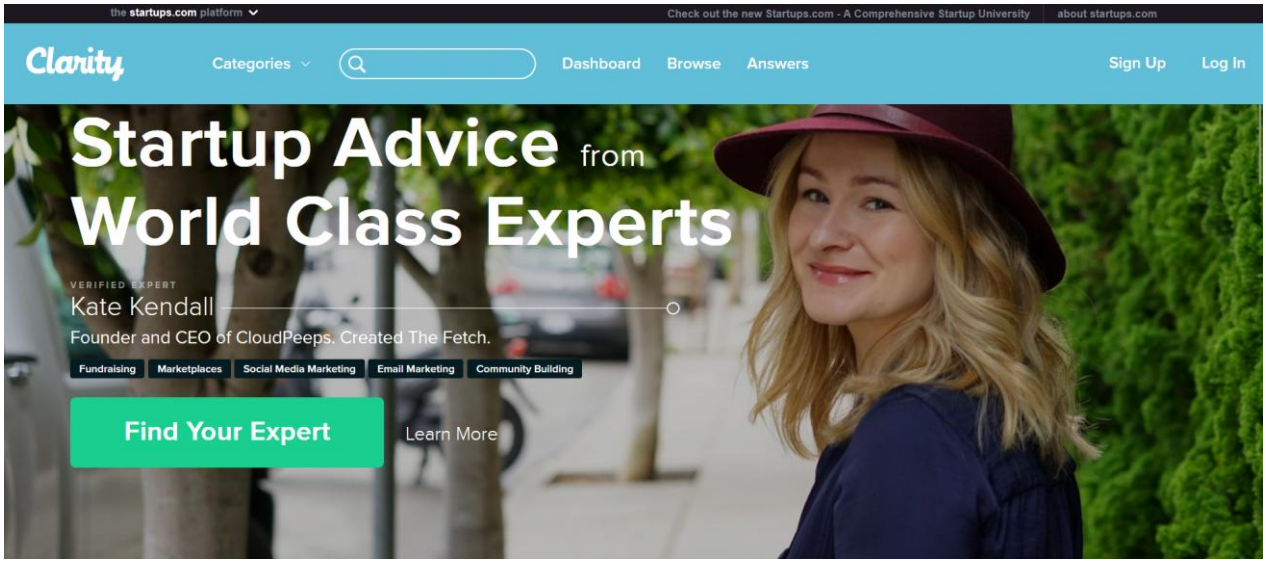

Рисунок 1.4 – Платформа Clarity.fm

Перевагою Clarity.fm є орієнтація саме на експертні консультації. Платформа дозволяє клієнту знайти спеціаліста у потрібній сфері та отримати консультацію у зручному форматі. Важливим елементом є профіль експерта, де користувач може оцінити його досвід і прийняти рішення щодо звернення. Для системи менеджменту консультаційних послуг це є важливим прикладом, оскільки профіль консультанта відіграє значну роль у виборі спеціаліста.

Недоліком такого рішення може бути прив’язка до конкретної моделі консультацій, наприклад до дзвінків або оплати за час. Крім того, готові платформи не завжди дозволяють адаптувати логіку роботи під конкретні вимоги окремого проєкту. Якщо організація хоче мати власні правила, структуру даних, ролі користувачів або додаткові модулі, використання зовнішньої платформи може обмежувати гнучкість.

Upwork є глобальною платформою для пошуку фахівців і виконання проєктної роботи. Хоча вона не є суто консультаційною системою, її можна розглядати як приклад інформаційної системи, яка підтримує взаємодію між клієнтом і виконавцем. Клієнт може знайти спеціаліста, переглянути профіль, оцінити досвід, домовитися про роботу та здійснювати подальшу взаємодію через платформу.

Перевагою Upwork є широкий функціонал для роботи з фахівцями. Платформа підтримує профілі, пошук, комунікацію, відгуки, оплату, історію співпраці та інші інструменти. Це демонструє, що для ефективного менеджменту послуг важливо не обмежуватися лише каталогом користувачів, а створювати повний цикл взаємодії. Для власного проєкту корисною є ідея структурованого профілю спеціаліста та прозорого процесу вибору виконавця.

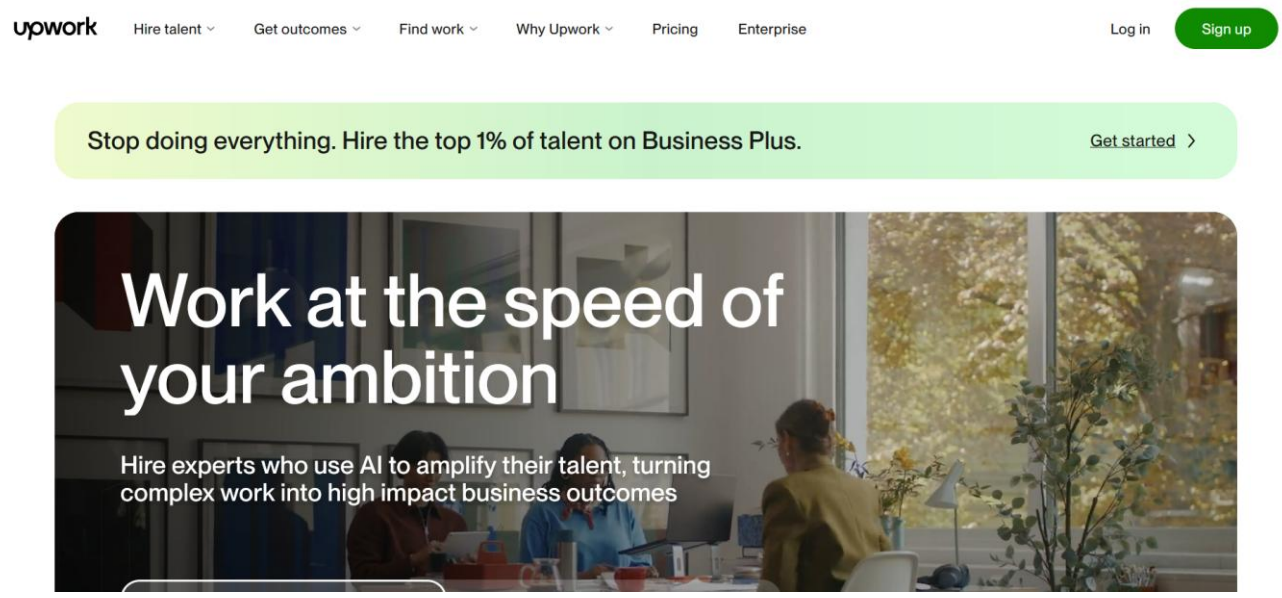

Рисунок 1.5 – Платформа Upwork

Недоліком Upwork у контексті консультаційних послуг є складність і надмірність функціоналу. Для невеликої або спеціалізованої системи така кількість можливостей може бути зайвою. Крім того, платформа орієнтована на широкий ринок фриланс-послуг, а не на конкретну предметну область. Власна система може бути простішою, більш цільовою та краще адаптованою до конкретного набору сценаріїв.

MentorCruise є платформою для пошуку менторів і організації менторської взаємодії. Вона орієнтована на довший формат співпраці між ментором і користувачем, зокрема у сферах програмної інженерії, продуктового менеджменту, дизайну, штучного інтелекту та бізнесу. Таке рішення є близьким до консультаційної тематики, оскільки в його основі лежить взаємодія між користувачем, який потребує допомоги, та спеціалістом, який має відповідний досвід.

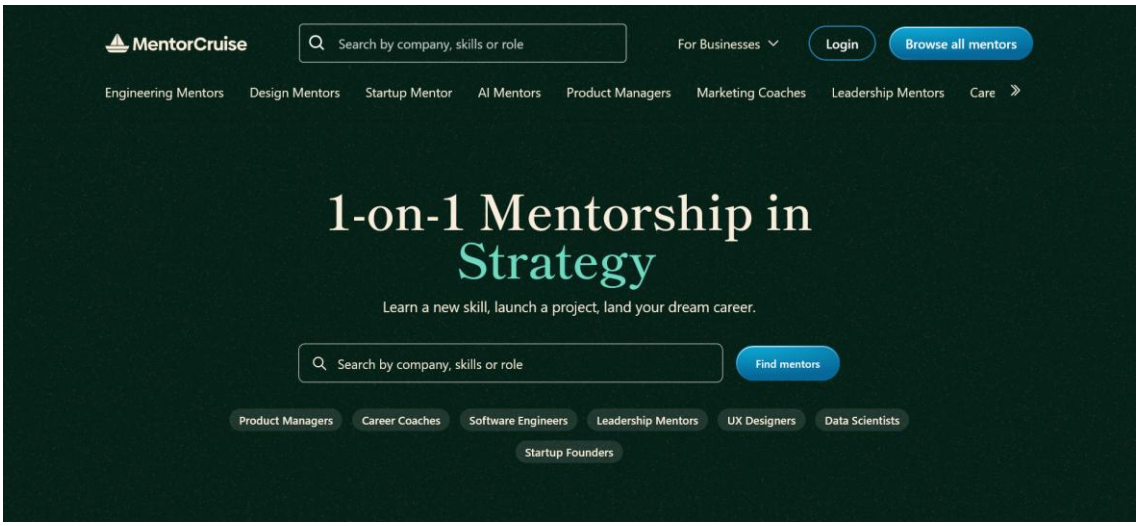

Рисунок 1.6 – Платформа MentorCruise

Перевагою MentorCruise є спеціалізація на професійному розвитку та менторстві. Платформа надає користувачу можливість обрати ментора за напрямом, переглянути профіль, оцінити досвід і формат співпраці. Це демонструє важливість якісного представлення консультанта або ментора в системі. Також корисним є підхід, за якого користувач може обрати спеціаліста не лише за назвою послуги, а й за досвідом, напрямом і очікуваним результатом.

Недоліком MentorCruise може бути те, що платформа орієнтована переважно на менторство і професійний розвиток, а не на універсальний менеджмент консультаційних послуг. Тому її функціональність може не повністю відповідати іншим типам консультацій, наприклад юридичним, фінансовим або освітнім. Власний проєкт може бути побудований більш універсально або адаптований до конкретної предметної області, яку потрібно підтримувати.

У таблиці 1.1 наведено основні рішення для підтримки консультаційних послуг.

Таблиця 1.1 — Основні рішення для підтримки консультаційних послуг.

Критерій порівняння	Calendly	Clarity.fm	Upwork	MentorCruise
Профілі спеціалістів	Частково	Так	Так	Так
Пошук або вибір консультанта	Обмежено	Так	Так	Так
Фільтрація за напрямом	Частково	Так	Так	Так
Перегляд опису послуги	Частково	Так	Так	Так
Створення запиту на консультацію	Частково	Так	Так	Так

За результатами порівняння можна зробити висновок, що існуючі рішення мають корисні функції, які можуть бути враховані під час розробки власної системи. Calendly демонструє важливість простоти запису та планування. Clarity.fm показує приклад платформи для прямої взаємодії клієнта з експертом. Upwork демонструє повний цикл роботи з фахівцями, включаючи профілі, пошук, комунікацію та оплату. MentorCruise є прикладом спеціалізованої платформи для менторства, де важливу роль відіграє довіра до спеціаліста та якісне представлення його досвіду.

Водночас жодне з розглянутих рішень не є повністю універсальним для всіх задач менеджменту консультаційних послуг. Частина платформ орієнтована на планування зустрічей, інша — на фріланс або менторство. Крім того, готові сервіси не завжди дозволяють змінювати внутрішню структуру системи, логіку бази даних, ролі користувачів або сценарії взаємодії. Тому розробка власної веб-орієнтованої системи є доцільною, якщо потрібно створити програмний продукт, адаптований до конкретної предметної області, навчальних вимог і обраної архітектури.

1.3 Постановка задачі

На основі аналізу процесу менеджменту консультаційних послуг та огляду існуючих рішень можна сформулювати задачу бакалаврської роботи. Необхідно розробити веб-орієнтовану систему менеджменту консультаційних послуг, яка забезпечує підтримку основних процесів взаємодії між користувачами та системою, збереження даних у базі даних, відображення інформації через клієнтський інтерфейс і обробку запитів на серверній частині.

Метою розробки є створення програмного продукту, який дозволяє організувати роботу з консультаційними послугами у веб-середовищі. Розробка має включати аналіз предметної області, вибір технологій, проєктування структури системи, опис бази даних, реалізацію основних компонентів і перевірку працездатності.

Основними функціональними вимогами до системи є:

− можливість роботи користувача з веб-інтерфейсом;

− відображення інформації про консультаційні послуги або консультантів;

− підтримка основних сценаріїв взаємодії користувача із системою;

− збереження даних у базі даних;

− обробка запитів на серверній частині;

− взаємодія клієнтської частини із серверною;

− можливість авторизації косристувача, якщо це передбачено реалізацією;

− можливість подальшого розширення функціональності.

До нефункціональних вимог можна віднести:

− зручність використання інтерфейсу;

− логічну структуру системи;

− стабільність роботи основних функцій;

− коректну обробку даних;

− можливість розгортання у веб-середовищі;

− підтримку подальшого розвитку програмного продукту;

− базове врахування питань безпеки та доступу до даних.

У системі можуть бути передбачені такі основні ролі: користувач, консультант та адміністратор. Користувач взаємодіє з веб-застосунком, переглядає інформацію, обирає консультанта або послугу та виконує доступні дії. Консультант може бути представлений у системі через профіль або інші дані, які дозволяють користувачу оцінити його спеціалізацію. Адміністратор може відповідати за підтримку даних, контроль роботи системи та керування окремими елементами.

Очікуваним результатом роботи є веб-орієнтована система, яка демонструє основні принципи побудови програмного продукту для менеджменту консультаційних послуг. Вона повинна містити клієнтську частину, серверну частину та базу даних, а також забезпечувати взаємодію між цими компонентами. Така система може бути використана як основа для подальшого розвитку: додавання повноцінного модуля запису на консультації, системи повідомлень, онлайн-оплати, рейтингу консультантів, адміністративної панелі або аналітичних інструментів.

Під час виконання бакалаврської роботи необхідно вирішити такі задачі:

− проаналізувати предметну область менеджменту консультаційних послуг;

− визначити роль інформаційних систем у підтримці консультаційної діяльності;

− розглянути існуючі веб-рішення, які мають подібну функціональність;

− сформулювати вимоги до розроблюваної системи;

− обрати засоби реалізації програмного продукту;

− спроєктувати структуру системи;

− описати структуру бази даних і функціональне призначення її полів;

− визначити основні сценарії використання програми;

− реалізувати основні модулі системи;

− перевірити функціонування програмного продукту.

2 РОЗРОБКА СТРУКТУРИ ВЕБ-ОРІЄНТОВАНОЇ СИСТЕМИ МЕНЕДЖМЕНТУ КОНСУЛЬТАЦІЙНИХ ПОСЛУГ

2.1 Вибір засобів для реалізації

Під час розробки веб-орієнтованої системи менеджменту консультаційних послуг було використано набір технологій, які забезпечують створення клієнтської частини, серверної частини, роботу з базою даних, авторизацію користувачів, контейнеризацію та розгортання системи. Вибір засобів реалізації є важливим етапом проєктування, оскільки саме від нього залежить зручність розробки, підтримка програмного продукту, можливість розширення функціональності та стабільність роботи системи.

Для реалізації системи було використано набір інструментів, які забезпечують роботу клієнтської частини, серверної логіки, бази даних та інфраструктури розгортання. Клієнтська частина побудована на основі Next.js, що дозволяє створювати сторінки веб-застосунку та організовувати взаємодію користувача з інтерфейсом. Серверна частина реалізована за допомогою FastAPI, який використовується для створення API, обробки запитів, авторизації користувачів та роботи з основними сутностями системи.

Для зберігання структурованих даних використовується PostgreSQL, а Redis застосовується для кешування та тимчасового зберігання даних. Boto3 забезпечує взаємодію backend-частини з AWS-сервісами, зокрема з Amazon S3, де зберігаються файли користувачів. Docker використовується для контейнеризації компонентів системи, а Terraform — для опису та створення хмарної інфраструктури у вигляді коду.

Для розгортання системи використовується хмарна платформа AWS. У межах проєкту застосовуються сервіси та ресурси, які забезпечують запуск веб-застосунку, зберігання файлів, налаштування доступу та керування інфраструктурою.

– Amazon EC2 — використовується для розміщення серверної інфраструктури та запуску контейнерів із frontend, backend, PostgreSQL, Redis і Traefik;

– Amazon S3 — використовується як файлове сховище для зберігання аватарів користувачів;

– Elastic IP — використовується для закріплення постійної публічної IP-адреси за EC2 instance;

– Security Group — використовується для налаштування мережевого доступу до сервера та обмеження відкритих портів;

– IAM Role та Instance Profile — використовуються для надання EC2 instance доступу до S3 без збереження ключів доступу безпосередньо в коді застосунку.

Опис інфраструктури виконується за допомогою Terraform. Це дозволяє зберігати конфігурацію AWS-ресурсів у вигляді коду, повторно створювати середовище розгортання та зменшити кількість ручних дій під час налаштування системи.

Такий підхід також спрощує підтримку інфраструктури, оскільки всі основні параметри розгортання зберігаються в одному конфігураційному наборі.

У таблиці 2.1 наведено основні технології, використані під час розробки системи.

Таблиця 2.1 — Засоби реалізації веб-орієнтованої системи менеджменту консультаційних послуг

Технологія	Частина системи	Призначення
Next.js	Frontend	Побудова клієнтської частини веб-застосунку
React	Frontend	Створення компонентів інтерфейсу
TypeScript	Frontend	Типізація структур даних
Chakra UI	Frontend/UI	Побудова форм, кнопок, dashboard-компонентів
Tailwind CSS	Frontend/UI	Додаткова стилізація інтерфейсу
Redux Toolkit / RTK Query	Frontend/API	Організація API-запитів і кешування відповідей
i18next	Frontend	Локалізація інтерфейсу
Js-cookie	Frontend/Auth	Зберігання токенів у браузері
react-easy-crop	Frontend/Images	Обрізання фото користувача
FastAPI	Backend	Реалізація HTTP API

Кінець таблиці 2.1

Uvicorn	Backend	Запуск FastAPI-застосунку
Pydantic	Backend	Валідація вхідних даних
SQLAlchemy	Backend/ORM	Робота з моделями бази даних
Alembic	Backend/DB	Міграції структури бази даних
PostgreSQL	База даних	Постійне зберігання структурованих даних
Redis	Кешування	Тимчасове зберігання даних і кешування
JWT	Auth	Авторизація користувачів
argon2-cffi	Auth	Хешування паролів
Docker	Deploy	Контейнеризація компонентів системи
Docker Compose	Deploy	Запуск набору сервісів
Traefik	Infrastructure	Маршрутизація запитів між frontend і backend
Terraform	AWS	Опис інфраструктури як коду
Amazon EC2	AWS	Зберігання аватарів користувачів
Boto3	Backend/AWS	Взаємодія Python-коду з Amazon S3

Таким чином, обраний стек технологій дозволяє реалізувати повноцінну веб-орієнтовану систему з клієнтською частиною, API, базою даних, авторизацією, кешуванням, контейнерним запуском і можливістю розгортання на базі AWS. Кожен інструмент виконує окрему роль у загальній структурі системи, а їх поєднання забезпечує логічну побудову програмного продукту.

2.2 Розробка структури системи на базі платформи AWS

Розроблювана система має клієнт-серверну архітектуру. Користувач взаємодіє з веб-застосунком через браузер. Клієнтська частина відповідає за відображення інтерфейсу, форми, списки, сторінки особистого кабінету та адміністративні розділи. Серверна частина приймає HTTP-запити, виконує перевірку даних, обробляє бізнес-логіку, взаємодіє з базою даних і повертає відповіді frontend-частині.

У production-структурі система може бути розгорнута на Amazon EC2. На сервері запускається набір контейнерів, які забезпечують роботу основних компонентів системи. До них належать frontend-контейнер, backend-контейнер, контейнер PostgreSQL, контейнер Redis і Traefik як reverse proxy. Такий підхід

дозволяє розмістити всі основні компоненти застосунку на одному сервері, але при цьому логічно розділити їх за функціями.

Основним зовнішнім входом у систему є HTTP або HTTPS-запит від користувача. Користувач відкриває веб-сайт у браузері, після чого запит потрапляє до Traefik. Traefik визначає, куди потрібно спрямувати запит. Якщо користувач звертається до звичайної сторінки веб-застосунку, запит передається до frontend-контейнера. Якщо запит має префікс /api, він спрямовується до backend-контейнера. Завдяки такому підходу користувач працює з єдиним доменом, а внутрішня маршрутизація виконується автоматично.

Frontend взаємодіє з backend через базовий API URL. У клієнтській частині використовується змінна середовища NEXT_PUBLIC_URL_TO_API. Якщо вона не задана, використовується значення /api/. У production-конфігурації API URL може вказувати на домен із префіксом /api/. Це дозволяє frontend-частині не прив’язуватися до конкретної адреси backend під час розробки і змінювати її залежно від середовища запуску.

Взаємодія frontend і backend відбувається через окремі API-модулі. Наприклад, usersApi.ts відповідає за роботу з користувачами, авторизацією та частиною адміністративних запитів. ordersApi.ts використовується для роботи з консультаційними запитами. tagsApi.ts відповідає за теги, reviewsApi.ts — за відгуки, paymentsApi.ts — за платіжні запити, а passwordRecoverApi.ts — за відновлення пароля. Така структура дозволяє розділити API-логіку за функціональними напрямами.

Backend-частина реалізована як FastAPI-застосунок. Вона має router-файли, які відповідають за окремі групи ендпоінтів: авторизацію, читання і оновлення користувачів, консультаційні запити, теги, відгуки, платежі та адміністративні функції. Такий поділ дозволяє організувати backend-код за предметними областями. Наприклад, логіка реєстрації та авторизації знаходиться в router-файлах користувачів, логіка роботи з консультаційними запитами — в окремому orders router, а адміністративні можливості — в admin router.

Backend взаємодіє з PostgreSQL через SQLAlchemy і DAO-класи. DAO-рівень відповідає за виконання операцій з базою даних: створення записів, пошук користувачів, оновлення даних, отримання консультаційних запитів, роботу з тегами та відгуками. Такий підхід дозволяє відокремити логіку доступу до даних від router-файлів, які обробляють HTTP-запити. Це покращує структуру backend-частини і спрощує підтримку коду.

PostgreSQL використовується як основне сховище даних. У ньому зберігаються користувачі, консультаційні запити, відгуки, теги, зв'язки користувачів із тегами та платіжні записи. Оскільки ці дані мають чітку структуру і між ними існують зв'язки, реляційна база даних добре підходить для цього проєкту. Наприклад, один користувач може бути клієнтом у багатьох консультаційних запитах, інший користувач може бути консультантом, а відгуки можуть бути пов'язані з клієнтом і консультантом.

Redis використовується як допоміжний компонент для кешування і тимчасових даних. У системі він застосовується для кешування списків консультантів і збереження коду відновлення пароля на обмежений час. Це дозволяє не зберігати короткочасні дані у PostgreSQL і зменшити навантаження на основну базу даних.

Окремим компонентом системи є Amazon S3. Він використовується для зберігання аватарів користувачів. Коли користувач оновлює фото профілю, backend обробляє завантаження і передає файл до S3 за допомогою Boto3. У базі даних може зберігатися посилання на фото, а сам файл розміщується у файловому сховищі. Такий підхід є більш гнучким, ніж зберігання файлів у контейнері, оскільки контейнери можуть перезапускатися, оновлюватися або переноситися.

Для розгортання інфраструктури використовується Terraform. У Terraform-конфігурації описано EC2 instance, Elastic IP, security group, IAM role, instance profile, S3 bucket для аватарів і опційний Route53 A-record. Security group визначає, які порти відкриті для доступу. Зазвичай для веб-застосунку потрібні порти 80 і 443, а порт 22 використовується для SSH-доступу до сервера. IAM role та instance

profile дозволяють EC2 отримати доступ до S3 без необхідності вручну зберігати ключі доступу всередині коду.

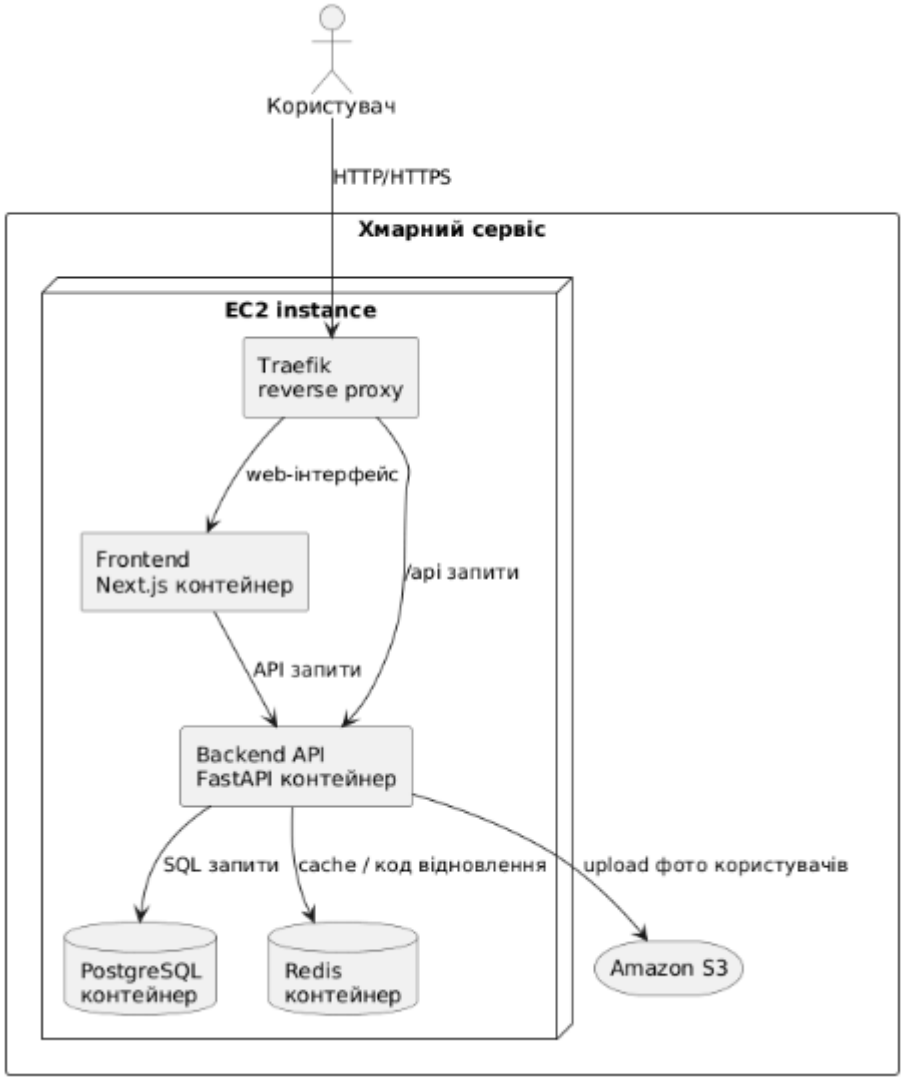

Рисунок 2.1 – Структура системи на базі AWS

Elastic IP використовується для надання серверу сталої публічної IP-адреси. Це важливо для production-середовища, оскільки IP-адреса сервера не повинна змінюватися після перезапуску. Route53 може використовуватися для створення DNS A-record, який прив’язує домен до IP-адреси сервера. У проєкті Route53 описаний як опційний ресурс, тому його потрібно розглядати як можливість production-конфігурації, а не як обов’язковий компонент.

2.3 Структура бази даних та функціональне призначення полів

База даних є одним із ключових компонентів системи менеджменту консультаційних послуг. Вона забезпечує зберігання основних сутностей системи: користувачів, консультаційних запитів, відгуків, тегів, зв'язків між користувачами і тегами, а також платіжної інформації. У розроблюваній системі база даних реалізована на PostgreSQL, а моделі описані за допомогою SQLAlchemy.

Структура бази даних побудована навколо таблиці users, оскільки користувач може виконувати різні ролі в системі. Один і той самий запис користувача може представляти клієнта, консультанта або адміністратора залежно від значень відповідних полів. Такий підхід дозволяє не створювати окремі таблиці для кожного типу користувача, а керувати ролями через ознаки is_consultant та is_admin.

Таблиця users призначена для зберігання облікових записів користувачів системи. Вона містить персональні дані, контактну інформацію, дані для авторизації, інформацію про профіль консультанта та службові ознаки ролей. У таблиці також передбачене поле для зберігання посилання на фото користувача.

Таблиця 2.2 — Таблиця users, яка містить дані користувачів системи.

Ключове поле	Назва стовпця	Тип даних	Довжина	Вміст
+	id	uuid		ID користувача
	first_name	varchar	255	Ім'я користувача
	last_name	Varchar	255	Прізвище користувача
	phone_number	varchar	255	Номер телефону
	email	varchar	255	Електронна пошта
	password	varchar	255	Пароль користувача
	photo	varchar	255	Фото користувача
	description	varchar	255	Опис профілю
	price	Int		Вартість консультації
	created_at	Timestamp		Дата створення
	is_consultant	Boolean		Ознака консультанта
	is_admin	boolean		Ознака адміністратора

Таблиця order призначена для зберігання консультаційних запитів. Вона пов’язує клієнта і консультанта, містить тему консультації, повідомлення клієнта, ціну, статус, запланований час і тривалість консультації.

Таблиця 2.3 — Таблиця order, яка містить дані консультаційних замовлень.

Ключове поле	Назва стовпця	Тип даних	Довжина	Вміст
+	id	uuid		ID замовлення
	price	Int		Статус замовлення
	status	Varchar	30	Статус замовлення
	topic	Varchar	255	Тема консультанта
	message	Carchar	1000	Повідомлення клієнта
	scheduled_at	Datetime		Запланований час консультації
	duration_minutes	int		Тривалість консультації
	created_at	timestamp		Дата створення
	consultant_id	Uuid		ID консультанта
	client_id	uuid		ID клієнта

Таблиця review використовується для зберігання відгуків клієнтів про консультантів. Вона містить текст відгуку, числову оцінку, дату створення, а також посилання на клієнта і консультанта.

Таблиця 2.4 — Таблиця review, яка містить дані відгуків клієнтів.

Ключове поле	Назва стовпця	Тип даних	Довжина	Вміст
+	id	uuid		ID відгуку
	description	Varchar	255	Текст відгуку
	rating	Int		Оцінка консультанта
	created_at	Timestamp		Дата створення
	consultant_id	Uuid		ID консультанта
	client_id	uuid		ID клієнта

Таблиця tag призначена для зберігання тематичних тегів. Теги використовуються для класифікації консультантів за напрямами діяльності або спеціалізаціями. Наприклад, консультант може бути пов’язаний з тегами, які описують його професійний напрям.

Завдяки використанню тегів користувач може швидше знайти консультанта за потрібною тематикою. Також теги спрощують структурування даних у системі,

оскільки дозволяють групувати консультантів не лише за загальним описом профілю, а й за конкретними напрямами консультаційних послуг.

Таблиця 2.5 — Таблиця tag, яка містить дані тематичних тегів.

Ключове поле	Назва стовпця	Тип даних	Довжина	Вміст
+	id	Uuid		ID тегу

Кінець таблиці 2.5

	name	Varchar	255	Назва тегу
	description	Varchar	255	Опис тегу
	created_at	Timestamp		Дата створення

Оскільки один консультант може мати декілька тегів, а один тег може належати багатьом консультантам, для зв’язку між користувачами і тегами використовується проміжна таблиця tags_users.

Таблиця tags_users реалізує зв’язок багато-до-багатьох між таблицями users і tags. Її первинний ключ складається з двох полів: tag_id і user_id. Завдяки цьому один користувач може мати декілька тегів, а один тег може бути пов’язаний з багатьма користувачами.

Таблиця 2.6 — Таблиця tags_users, яка містить дані зв’язку користувачів із тегами.

Ключове поле	Назва стовпця	Тип даних	Довжина	Вміст
+	tag_id	Uuid		ID тегу
+	user_id	uuid		ID користувача

Також у базі даних системи для основних ідентифікаторів використовується тип UUID. Такий ідентифікатор є унікальним значенням, яке може генеруватися автоматично та не залежить від послідовної нумерації записів. У розроблюваній системі UUID застосовується для таблиць користувачів, замовлень, відгуків і тегів, що дозволяє однозначно визначати кожен запис у базі даних.

Використання UUID має декілька переваг. По-перше, такі ідентифікатори зменшують ризик конфліктів між записами, оскільки кожне значення є практично унікальним. По-друге, UUID зручні для систем, які можуть розширюватися або працювати з різними джерелами даних.

По-третє, такі ідентифікатори не розкривають кількість записів у таблиці, на відміну від звичайних послідовних числових ID. Це робить структуру даних більш гнучкою та придатною для подальшого розвитку системи. Крім того, UUID зручно використовувати у зв'язках між таблицями, оскільки кожен користувач, запит, відгук або тег має стабільний унікальний ідентифікатор у межах усієї системи.

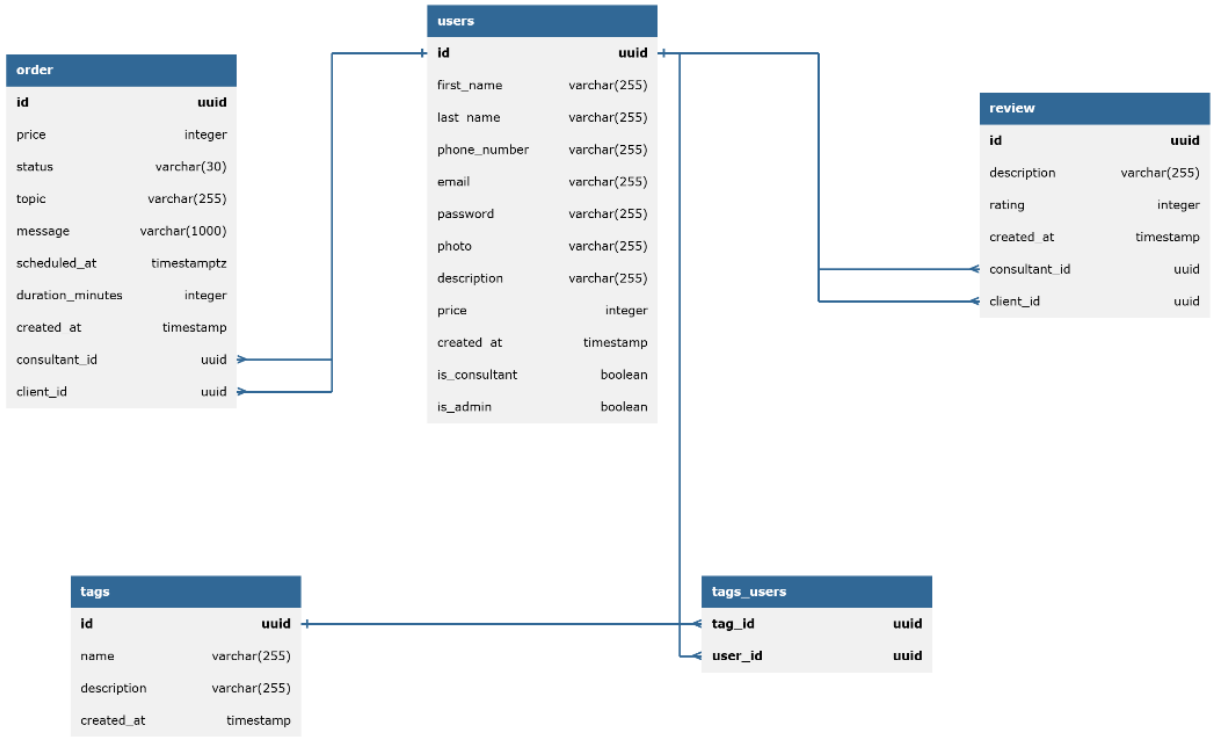

Рисунок 2.2 – ER-діаграма бази даних

Зв'язки між таблицями бази даних мають таку структуру:

– один користувач може бути клієнтом у багатьох консультаційних запитах;

– один користувач може бути консультантом у багатьох консультаційних запитах;

– один користувач може залишити багато відгуків як клієнт;

– один консультант може мати багато відгуків;

– користувачі та теги мають зв'язок багато-до-багатьох через таблицю tags_users.

Таким чином, структура бази даних забезпечує збереження основних даних системи та відображає логіку взаємодії між клієнтами, консультантами, консультаційними запитами, відгуками і тегами.

Використання зв'язків між таблицями дозволяє уникнути дублювання інформації, підтримувати цілісність даних і забезпечити коректну роботу основних сценаріїв веб-орієнтованої системи.

2.4 Сценарії використання програми

Сценарії використання описують взаємодію користувачів із системою. Вони дозволяють визначити, які дії може виконувати кожна роль і які компоненти системи беруть участь у виконанні цих дій. У розроблюваній системі основними ролями є клієнт, консультант і адміністратор. Усі ці ролі представлені через таблицю users, а їхні можливості визначаються полями is_consultant та is_admin.

Основними сценаріями використання системи є:

– реєстрація користувача;

– авторизація користувача;

– відновлення пароля;

– перегляд списку консультантів;

– фільтрація консультантів за тегом;

– перегляд деталей консультанта;

– редагування профілю;

– створення консультаційного запиту;

– перегляд консультацій в особистому кабінеті;

– зміна статусу консультації;

– робота з відгуками;

– адміністративне керування.

Сценарій реєстрації користувача починається з того, що неавторизований користувач відкриває сторінку реєстрації. Він вводить ім'я, прізвище, email і пароль. Після цього frontend надсилає запит до backend на ендпоінт POST

/auth/signup. Backend перевіряє, чи вже існує користувач із таким email. Якщо такого користувача немає, пароль хешується, а новий запис створюється в таблиці users. Результатом сценарію є створення нового облікового запису.

Сценарій перегляду консультацій в особистому кабінеті доступний авторизованому користувачу. Після входу в dashboard frontend отримує дані поточного користувача, після чого викликає endpoint GET /orders/account/{user_id}. Backend перевіряє, що користувач переглядає власні консультації або має права адміністратора. Після цього повертається список консультацій, пов'язаних з користувачем. Такий сценарій дозволяє клієнту або консультанту контролювати власні консультаційні запити.

Сценарій зміни статусу консультації використовується консультантом або адміністратором. У системі консультаційний запит має певний статус. Frontend показує доступні переходи статусів відповідно до визначеної логіки.

Після вибору нового статусу надсилається запит PATCH /orders/{order_id}/status. Backend перевіряє права користувача і допустимість переходу статусу, після чого оновлює запис у базі даних. Це дозволяє контролювати життєвий цикл консультаційного запиту.

Сценарій роботи з відгуками дозволяє клієнтам залишати оцінки консультантам. Frontend надсилає запит POST /reviews/, а backend зберігає текст відгуку, рейтинг, ідентифікатор клієнта та ідентифікатор консультанта.

Адміністратор може переглядати всі відгуки через адміністративний endpoint і видаляти їх у разі потреби. Відгуки використовуються для формування рейтингу консультанта, який може відображатися в картці спеціаліста.

Адміністративне керування доступне користувачам з ознакою is_admin = true. Адміністратор може переглядати користувачів, консультаційні запити, відгуки, теги і статистику. Також він може змінювати admin-статус користувачів, керувати тегами і виконувати інші дії, передбачені адміністративними ендпоінтами. Адміністративна панель є важливою для підтримки системи, оскільки дозволяє контролювати її дані.

Таке розділення функцій між користувачем, консультантом та адміністратором дозволяє впорядкувати доступ до різних частин системи і забезпечити контроль над основними даними.

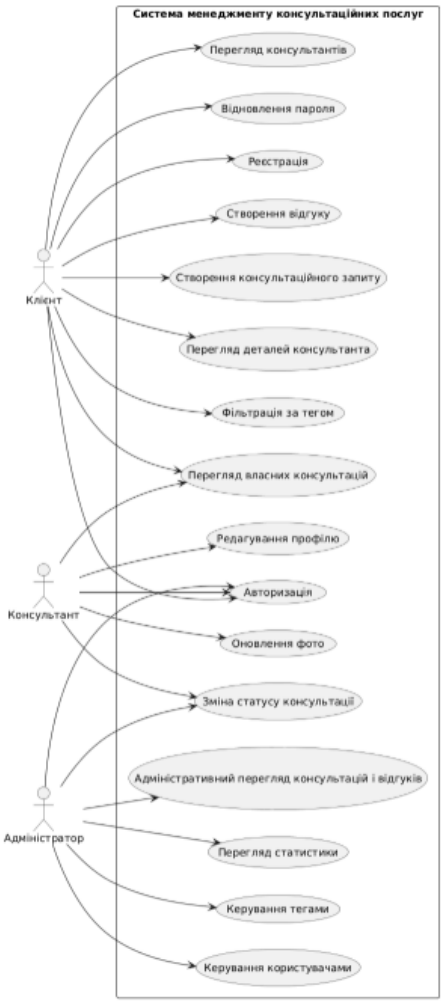

Рисунок 2.3 – Діаграма варіантів використання системи

Якщо користувач уже авторизований, backend перевіряє, щоб клієнт не створював консультаційний запит самому собі. Якщо користувач не авторизований, серверна частина виконує пошук користувача за email. У випадку, якщо такого користувача ще немає в базі даних, система створює новий запис у таблиці users.

Після визначення клієнта backend створює консультаційний запит через DAO-рівень. Дані записуються у таблицю order, після чого система отримує створений запис разом із пов’язаними даними та повертає результат frontend-

частині. У відповідь frontend отримує дані консультаційного запиту та інформацію про подальший сценарій авторизації або реєстрації користувача.

На рисунку 2.3 зображено діаграму послідовності створення консультаційного запиту. Вона демонструє порядок взаємодії між користувачем, клієнтською частиною системи, серверною частиною, DAO-рівнем та базою даних PostgreSQL.

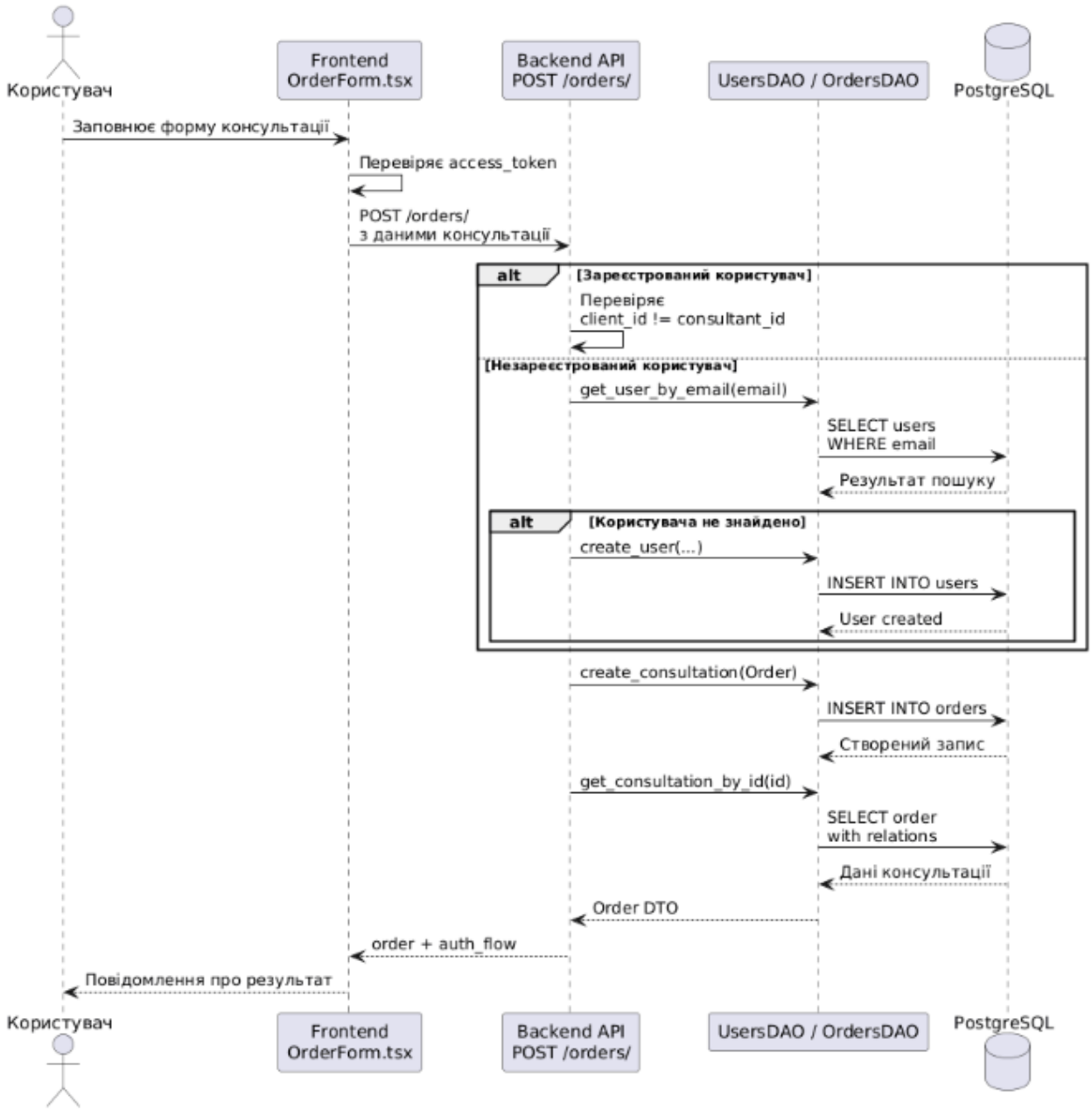

Рисунок 2.4 – Діаграма послідовності створення консультаційного запиту

Таким чином, діаграма послідовності показує внутрішню логіку створення консультаційного запиту та демонструє, як взаємодіють основні компоненти системи під час виконання одного з ключових сценаріїв програми.

3 РЕАЛІЛАЦІЯ ВЕБ-ОРІЄНТОВАНОЇ СИСТЕМИ МЕНЕДЖМЕНТУ КОНСУЛЬТАЦІЙНИХ ПОСЛУГ

3.1 Програмна реалізація системи

Програмна реалізація системи складається з кількох основних частин: frontend-застосунку, backend-застосунку, бази даних, конфігураційних файлів для запуску та файлів для розгортання інфраструктури. Такий поділ дозволяє логічно відокремити інтерфейс користувача, серверну логіку, зберігання даних і процес розгортання. Завдяки цьому кожна частина системи виконує окрему функцію і може розроблятися, тестуватися та оновлюватися незалежно від інших компонентів.

У репозиторії frontend-частина розміщена в папці advicerr-frontend, а backend-частина — у папці consulting-backend. Окремо присутні файли для запуску системи через Docker Compose та папка deployment, яка містить файли, пов’язані з розгортанням системи на базі AWS. Така структура репозиторію спрощує навігацію в проєкті, оскільки дозволяє швидко визначити, де знаходиться код інтерфейсу, серверна логіка, конфігурації запуску та інфраструктурні налаштування.

Загальна структура проєкту може бути подана так:

− advicerr-frontend — клієнтська частина системи;

− consulting-backend — серверна частина системи;

− docker-compose.dev.yml — конфігувація для запуску системи в режимі розробки;

− docker-compose.prod.yml — конфігувація для production-запуску;

− deployment — файли для розгортання інфраструктури;

− deployment/infra/prod — Terraform-конфігурація production-інфраструктури;

− deployment/infra/s3_avatars — Terraform-конфігурація для S3-сховища аватарів;

− deployment/scripts — допоміжні скрипти для налаштування і розгортання.

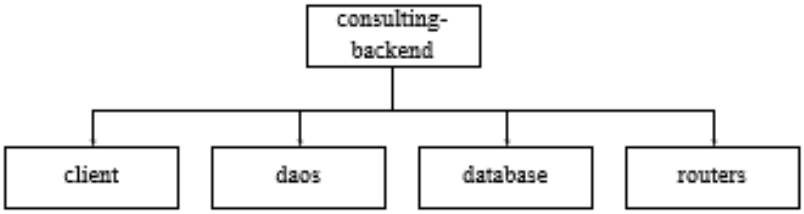

Рисунок 3.1 – структура каталогів backend частини

Клієнтська частина системи реалізована у папці advicerr-frontend. Вона побудована на основі Next.js, React і TypeScript. Основним файлом головної сторінки є advicerr-frontend/src/app/page.tsx. Саме ця сторінка відповідає за початковий інтерфейс користувача та відображення консультантів. Для відображення карток консультантів використовується компонент ConsultantCardV3.tsx, а для пагінації списку консультантів — компонент Pagination.tsx.

Файл advicerr-frontend/src/app/page.tsx відповідає за головну сторінку системи. На ній користувач може переглядати список консультантів, взаємодіяти з картками спеціалістів і переходити до створення консультаційного запиту. Головна сторінка є однією з ключових частин frontend-застосунку, оскільки саме з неї починається основний сценарій роботи клієнта.

Файл ConsultantCardV3.tsx відповідає за виведення інформації про окремого консультанта. У картці може відображатися ім’я, прізвище, фото, опис, вартість консультації, теги, рейтинг і відгуки. Також через картку користувач може перейти до створення консультаційного запиту. Такий підхід дозволяє зібрати основну інформацію про консультанта в одному інтерфейсному блоці.

Фрагмент коду компоненту ConsultantCard:

```
<Drawer isOpen={isChatOpen} placement="right" onClose={toggleChat} size="md">
```

```
        <DrawerOverlay>
          <DrawerContent>
            <DrawerCloseButton />
            <DrawerHeader>{user.name} {user.surname}</DrawerHeader>
            <DrawerBody>
              <CurrentChat chat={chats.find(chat => chat.id === user.id) ||
chats[0]}
              messages={messages}/>
            </DrawerBody>
          </DrawerContent>
        </DrawerOverlay>
      </Drawer>

      <Drawer isOpen={isOrderOpen && !isOwnConsultantCard} placement="right"
onClose={toggleOrder} size="md">
        <DrawerOverlay>
          <DrawerContent>
            <DrawerCloseButton />
            <DrawerHeader>{t("orderConsultationLabel")} {user.name}
{user.surname}</DrawerHeader>
            <DrawerBody>
              {/* <Payment payment={payments.find(payment => payment.id ===
user.id) || payments[0]}></Payment> */}
              <OrderForm consultantId={user.id}
price={user.pricePerHour}></OrderForm>
            </DrawerBody>
          </DrawerContent>
        </DrawerOverlay>
      </Drawer>
```

Форма створення консультаційного запиту реалізована у файлі OrderForm.tsx. Вона дозволяє користувачу передати тему консультації, повідомлення, бажаний час, тривалість та контактні дані. Якщо користувач авторизований, frontend використовує його ідентифікатор. Якщо користувач не авторизований, система може передати додаткові дані для створення або пошуку користувача на backend-стороні.

Файл OrderForm.tsx є важливим для основного бізнес-сценарію системи, оскільки саме через нього користувач створює консультаційний запит. Після заповнення форми frontend надсилає дані до backend через API-модуль ordersApi.ts.

Код запиту для створення замовлення:

```
createOrder: builder.mutation<TCreateOrderResponse, TOrderForRegisteredUser |
TOrderForUnregisteredUser>({
```

```
      query: (body) => ({
        url: "orders/",
        method: "POST",
        body: body
      }),
    })
```

Особистий кабінет користувача реалізований у файлі advicerr-frontend/src/app/dashboard/Dashboard.tsx. Цей компонент відповідає за отримання даних поточного користувача, відображення персональної інформації, консультацій, адміністративних розділів та інших елементів dashboard. Якщо користувач має права адміністратора, у dashboard можуть відображатися додаткові адміністративні блоки.

Файл PersonalInformation.tsx відповідає за редагування персональних даних користувача. У ньому користувач може змінити ім’я, прізвище, номер телефону, опис, ціну консультації, теги та ознаку консультанта. Це дозволяє звичайному користувачу налаштувати власний профіль і, за потреби, відображатися в системі як консультант.

Файл ImageUploader.tsx використовується для завантаження та обрізання фотографії користувача. Для цього застосовується бібліотека react-easy-crop. Після обробки фото frontend надсилає його до backend, а backend завантажує файл до Amazon S3. У результаті в базі даних зберігається посилання на фото користувача.

Код запиту для оновлення інформації користувача:

```
updateUser: builder.mutation<User, { id: string; body: TUserToUpdate }>({
      query: ({ id, body }) => ({
        url: `users/update/${id}`,
        method: 'PATCH',
        body: body
      })
    })
```

Серверна частина системи розміщена в папці consulting-backend. Основним файлом запуску backend є consulting-backend/core/main.py. У ньому створюється FastAPI-додаток, підключаються router-и та налаштовуються основні параметри

роботи API. Саме через цей файл backend об’єднує окремі модулі в єдиний серверний застосунок.

Основна логіка backend розподілена за router-файлами в папці consulting-backend/core/routers. Такий поділ дозволяє окремо реалізувати різні групи функціональності. Наприклад, router-и користувачів відповідають за реєстрацію, авторизацію, читання й оновлення даних користувачача, orders_router.py відповідає за консультаційні запити, tags_router.py реалізує роботу з тегами, reviews_router.py — роботу з відгуками, а admin_router.py — адміністративні можливості.

Файл consulting-backend/core/routers/users/auth.py реалізує сценарії реєстрації, авторизації, оновлення токена та відновлення пароля. Під час реєстрації backend перевіряє, чи існує користувач із таким email, після чого хешує пароль і створює запис у таблиці users. Під час авторизації backend перевіряє email і пароль, а потім повертає access token і refresh token.

Код ендпоінту для реєстрації користувача:

```
@auth_router.post("/signup", summary="Create new user")
async def create_user(data: UserSchemaRegister):
    logger.info("Signup attempt received for email=%s", data.email)

    users_dao = UsersDAO(uri=db_uri)
    user = users_dao.get_user_by_email(data.email)

    if user is not None:
        logger.warning("Signup rejected for duplicate email=%s", data.email)
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="User with this email already exist"
        )

    user_to_create = User(
        email=data.email,
        password=get_hashed_password(data.password),
        first_name=data.first_name,
        last_name=data.last_name,
    )
    created_user = users_dao.create_user(user_to_create)
       logger.info("Signup succeeded for email=%s user_id=%s", data.email,
created_user.id)
    return created_user
```

Для захисту паролів використовується хешування. Це означає, що пароль користувача не зберігається в базі даних у відкритому вигляді. Під час реєстрації пароль перетворюється на хеш, а під час авторизації введений пароль порівнюється з хешем. Такий підхід підвищує безпеку системи.

Файл consulting-backend/core/utils.py містить допоміжні функції, які використовуються для авторизації, створення токенів, перевірки паролів або інших службових операцій. Файл consulting-backend/core/deps.py містить залежності FastAPI, які можуть використовуватися для отримання поточного користувача, перевірки токена або перевірки прав адміністратора.

Код перевірки JWT-токена:

```
async def get_current_user(token: str = Depends(reusable_oauth_for_user)):
    try:
        payload = jwt.decode(
            token, JWT_SECRET_KEY, algorithms=[ALGORITHM]
        )
        token_data = TokenPayload(**payload)
        if datetime.fromtimestamp(token_data.exp) < datetime.now():
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Token expired",
                headers={"WWW-Authenticate": "Bearer"},
            )
    except (jwt.JWTError, ValidationError):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Could not validate credentials",
            headers={"WWW-Authenticate": "Bearer"},
        )

    users_dao = UsersDAO(uri=db_uri)
    user = users_dao.get_user_by_email(token_data.sub)

    if user is None:
        raise HTTPException(
            status_code=status.HTTP_404_BAD_REQUEST,
            detail="Could not find user"
        )

    return user[0]
```

Файл consulting-backend/core/routers/orders_router.py відповідає за створення та отримання консультаційних запитів. Основним сценарієм є створення запиту через endpoint POST /orders/. Під час виконання цього сценарію backend отримує дані з форми, перевіряє їх, визначає клієнта і консультанта, створює запис у таблиці order та повертає результат frontend-частині.

Система підтримує створення консультаційного запиту як для авторизованого, так і для неавторизованого користувача. Якщо користувач не авторизований, backend може знайти користувача за email або створити новий запис. Це дозволяє зробити процес звернення до консультанта більш гнучким, оскільки клієнт може залишити запит без попереднього повного проходження реєстрації.

Фрагмент коду для створення замовлення на серверній частині:

```
consultation_for_insert = Order(
            price=consultation.price,
            status="new",
            topic=consultation.topic,
            message=consultation.message,
            scheduled_at=consultation.scheduled_at,
            duration_minutes=consultation.duration_minutes,
            consultant_id=consultation.consultant_id,
            client_id=consultation.client_id
        )
consultations_dao = OrdersDAO(uri=db_uri)
    created_consultation =
consultations_dao.create_consultation(consultation_for_insert)
```

Файл admin_router.py відповідає за адміністративну частину системи. Через нього адміністратор може отримувати списки користувачів, консультаційних запитів, відгуків, статистику, а також виконувати окремі дії, пов’язані з керуванням системою. Доступ до таких ендпоінтів повинен бути обмежений лише користувачами з ознакою is_admin.

Адміністративний функціонал є важливим для підтримки системи, оскільки дозволяє контролювати дані, переглядати активність, керувати користувачами, тегами, відгуками та консультаційними запитами. У frontend-частині

адміністративні елементи відображаються в dashboard тільки для користувачів, які мають відповідні права.

Файл tags_router.py реалізує API для роботи з тегами. Теги використовуються для категоризації консультантів і фільтрації списку спеціалістів. Користувач може мати декілька тегів, а один тег може бути пов’язаний із багатьма користувачами. Такий зв’язок реалізовано через проміжну таблицю tags_users.

Файл reviews_router.py відповідає за роботу з відгуками. Клієнт може залишити відгук про консультанта, а адміністратор може переглядати або видаляти відгуки. Відгуки використовуються для формування рейтингу консультанта, який відображається в системі. Це допомагає користувачам краще оцінювати спеціалістів під час вибору консультації.

Робота з базою даних реалізована через моделі SQLAlchemy, які описані у файлі consulting-backend/core/database/models.py. У цьому файлі визначено таблиці users, order, review, tags і tags_users. Кожна модель відповідає окремій сутності системи.

Код таблиці «Order»:

```
class Order(Base):
    __tablename__ = 'order'

    id: Mapped[uuid.UUID] = mapped_column(
        UUID(as_uuid=True),
        primary_key=True,
        server_default=text("gen_random_uuid()")      )
    price = Column(Integer)
    status = Column(String(30), nullable=False, server_default="new")
    topic = Column(String(255))
    message = Column(String(1000))
    scheduled_at = Column(DateTime(timezone=True))
    duration_minutes = Column(Integer, nullable=False, server_default="60")
    created_at = Column(TIMESTAMP, server_default=func.now())
       consultant_id:   Mapped[uuid.UUID]   =   mapped_column(ForeignKey("users.id",
ondelete="CASCADE"))
        client_id:   Mapped[uuid.UUID]   =   mapped_column(ForeignKey("users.id",
ondelete="CASCADE"))

       consultant: Mapped["User"] = relationship(
        "User",
        foreign_keys=[consultant_id],
```

```
        back_populates="consultations_as_consultant",
    )
    client: Mapped["User"] = relationship(
        "User",
        foreign_keys=[client_id],
        back_populates="consultations_as_client",
    )
```

Для виконання операцій з базою даних використовуються DAO-класи, розміщені в папці consulting-backend/core/daos. DAO-рівень відповідає за виконання запитів до бази даних і дозволяє відокремити доступ до даних від router-файлів. Це покращує структуру backend-частини, оскільки HTTP-ендпоінти не містять усю SQLAlchemy-логіку безпосередньо.

Файл consulting-backend/core/clients/DBClient.py відповідає за створення підключення до бази даних і роботу із сесіями. У ньому налаштовується engine SQLAlchemy і механізм отримання session. Це є основою для всіх операцій, які виконуються з PostgreSQL.

Код методу для підключення до бази даних:

```
def _get_session(self):
        if not self._async_session:
            self._async_session = scoped_session(
                sessionmaker(bind=self._engine, expire_on_commit=False)
            )
        return self._async_session
```

Для керування змінами структури бази даних використовується Alembic. Файли міграцій розміщені в папці consulting-backend/core/database/migrations/versions. Під час запуску backend-контейнера виконується команда оновлення схеми бази даних до останньої версії. Це дозволяє автоматично застосовувати зміни структури таблиць під час запуску системи.

Схеми вхідних і вихідних даних описані у файлі consulting-backend/core/pydantic_models.py. У ньому визначаються моделі, які використовуються для перевірки даних, що надходять від frontend. Наприклад, окремі Pydantic-моделі можуть описувати дані користувача, консультаційного запиту, статусу, відгуку або платежу. Завдяки цьому backend може перевірити структуру запиту ще до виконання основної логіки.

Файл consulting-backend/core/serializers.py відповідає за структуру відповідей API. Серіалізатори допомагають перетворити дані з моделей бази даних у формат, який може бути переданий frontend-частині. Це важливо, оскільки frontend не повинен отримувати зайві або службові дані, а відповіді API мають бути передбачуваними.

Контейнеризація системи реалізована за допомогою Docker. Backend має власний Dockerfile, а frontend — prod.Dockerfile. Для запуску всіх компонентів разом використовуються файли docker-compose.dev.yml і docker-compose.prod.yml. У Docker Compose описуються сервіси frontend, backend, PostgreSQL, Redis і Traefik.

Файл docker-compose.dev.yml використовується для локального запуску системи під час розробки. Він дозволяє швидко підняти необхідні сервіси без ручного встановлення кожного компонента. Файл docker-compose.prod.yml призначений для production-запуску і містить налаштування, які використовуються під час розгортання системи на сервері.

Фрагмент коду для розгортання UI-частини проєкту:

```
frontend:
    build:
      context: ./advicerr-frontend
      dockerfile: dev.Dockerfile
    working_dir: /app
    command: sh -c "npm run dev -- --hostname 0.0.0.0 --port 3000"
    env_file:
      - ./advicerr-frontend/.env
    environment:
      NEXT_PUBLIC_URL_TO_API: ${NEXT_PUBLIC_URL_TO_API:-/api/}
      WATCHPACK_POLLING: "true"
    volumes:
      - ./advicerr-frontend:/app
      - frontend_node_modules:/app/node_modules
      - frontend_next:/app/.next
    depends_on:
      - backend
    labels:
      - traefik.enable=true
      - traefik.docker.network=bachelors_dev_web
      - traefik.http.routers.frontend-dev.rule=Host(`${TRAEFIK_DEV_HOST:-
localhost}`)
```

```
    - traefik.http.routers.frontend-dev.entrypoints=web
    - traefik.http.routers.frontend-dev.priority=1
    - traefik.http.services.frontend-dev.loadbalancer.server.port=3000
networks:
  - web
```

Traefik виконує роль reverse proxy. Він приймає запити користувача і спрямовує їх до відповідного контейнера. Якщо запит стосується frontend-сторінок, він передається до frontend. Якщо запит має префікс /api, він передається до backend. Такий підхід дозволяє працювати з системою через один домен і приховує внутрішню структуру контейнерів від користувача.

Розгортання на AWS описане в папці deployment. У папці deployment/infra/prod розміщені Terraform-файли для створення production-інфраструктури. У цих файлах описано EC2 instance, Elastic IP, security group, IAM role, instance profile, S3 bucket для аватарів і опційний Route53 A-record. Також використовується user data script, який може виконувати дії з підготовки сервера та запуску application stack.

Файли deployment/scripts/setup_s3_avatars.py і deployment/scripts/deploy_prod.py використовуються для допоміжних операцій, пов’язаних з розгортанням і налаштуванням S3-сховища для аватарів. Завдяки цьому частина інфраструктурних дій автоматизується.

Фрагмент коду опису EC2-інстансу використовуючи Terraform:

```
variable "instance_type" {
  type        = string
  description = "EC2 instance type."
  default     = "t3.small"
}
resource "aws_iam_instance_profile" "ec2_app" {
  name = "${var.project_name}-prod-instance-profile"
  role = aws_iam_role.ec2_app.name
}
resource "aws_instance" "app" {
  ami                         = data.aws_ami.ubuntu.id
  instance_type               = var.instance_type
  key_name                    = var.key_name
  subnet_id                   = data.aws_subnets.default.ids[0]
  vpc_security_group_ids      = [aws_security_group.app.id]
  associate_public_ip_address = true
  iam_instance_profile        = aws_iam_instance_profile.ec2_app.name
  user_data                   = local.user_data

  root_block_device {
    volume_size = var.root_volume_size
```

```
    volume_type = "gp3"
  }

  tags = merge(local.common_tags, {
    Name = "${var.project_name}-prod"
  })
}
```

Таким чином, програмна реалізація системи має чіткий поділ на frontend, backend, базу даних та інфраструктуру. Frontend відповідає за інтерфейс користувача і взаємодію з API. Backend реалізує основну логіку системи, авторизацію, обробку консультаційних запитів, роботу з користувачами, тегами, відгуками та файлами. База даних зберігає основні сутності системи, Redis використовується для тимчасових даних, а Amazon S3 — для зберігання аватарів. Docker і Terraform забезпечують запуск та розгортання системи.

3.2 Перевірка функціонування системи

Після реалізації основних компонентів системи необхідно перевірити її функціонування. Перевірка потрібна для того, щоб переконатися у коректній роботі клієнтської частини, серверної частини, бази даних, авторизації, створення консультаційних запитів, редагування профілю, роботи з відгуками, тегами та адміністративними розділами. Також перевірка дозволяє виявити можливі помилки у взаємодії між окремими частинами системи та переконатися, що дані коректно передаються між frontend, backend і базою даних.

Перевірка виконувалася шляхом проходження основних сценаріїв використання системи. Для кожного сценарію перевірялося, чи правильно відображається інтерфейс, чи надсилаються API-запити, чи повертає backend коректні відповіді, чи зберігаються дані в базі даних і чи правильно обробляються помилки. Особлива увага приділялася сценаріям, які є основними для роботи системи: реєстрації користувача, авторизації, перегляду консультантів, створенню консультаційного запиту та роботі з особистим кабінетом.

Першим етапом перевірки є запуск системи. Для цього необхідно підняти всі основні компоненти: frontend, backend, PostgreSQL, Redis і reverse proxy. Після

запуску потрібно перевірити, чи відкривається головна сторінка веб-застосунку і чи доступні API-ендпоінти. Якщо всі компоненти запускаються без помилок, це підтверджує, що базова конфігурація системи є коректною і можна переходити до перевірки окремих функціональних можливостей.

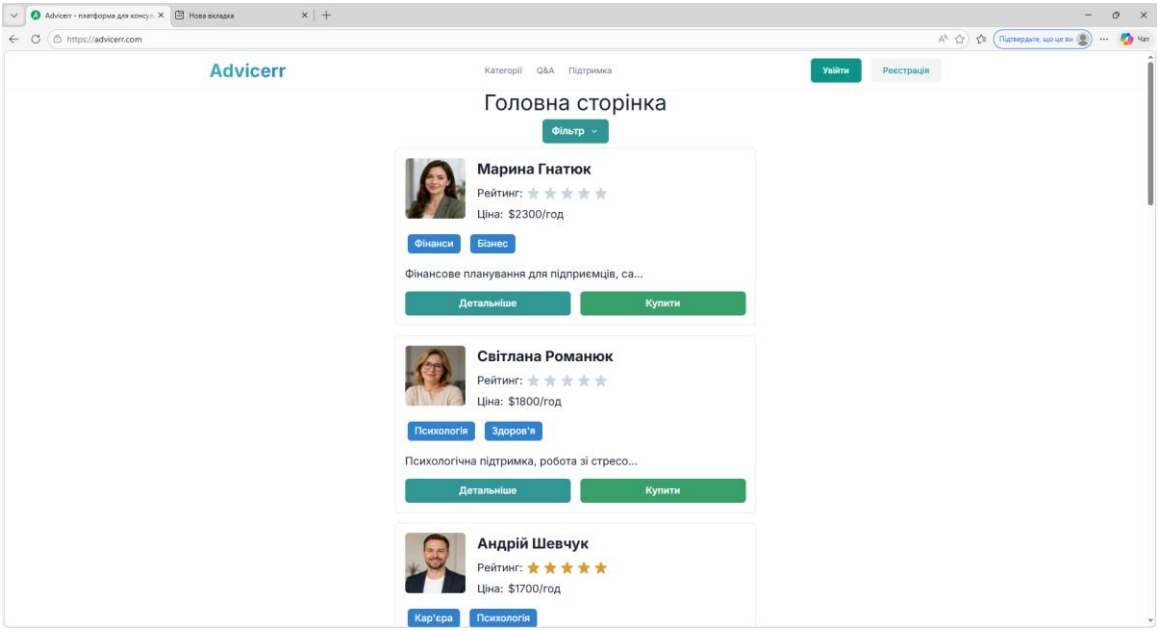

Рисунок 3.2 – Головна сторінка

Для перевірки фільтрації за тегами потрібно вибрати певний тег і переконатися, що список консультантів оновлюється відповідно до вибраного напряму. Для тестування було обрано напрям “Здоров’я”. Після вибору цього тегу система повинна надіслати запит до backend-частини, отримати консультантів, які пов’язані з відповідним тегом, і відобразити їх у списку на головній сторінці.

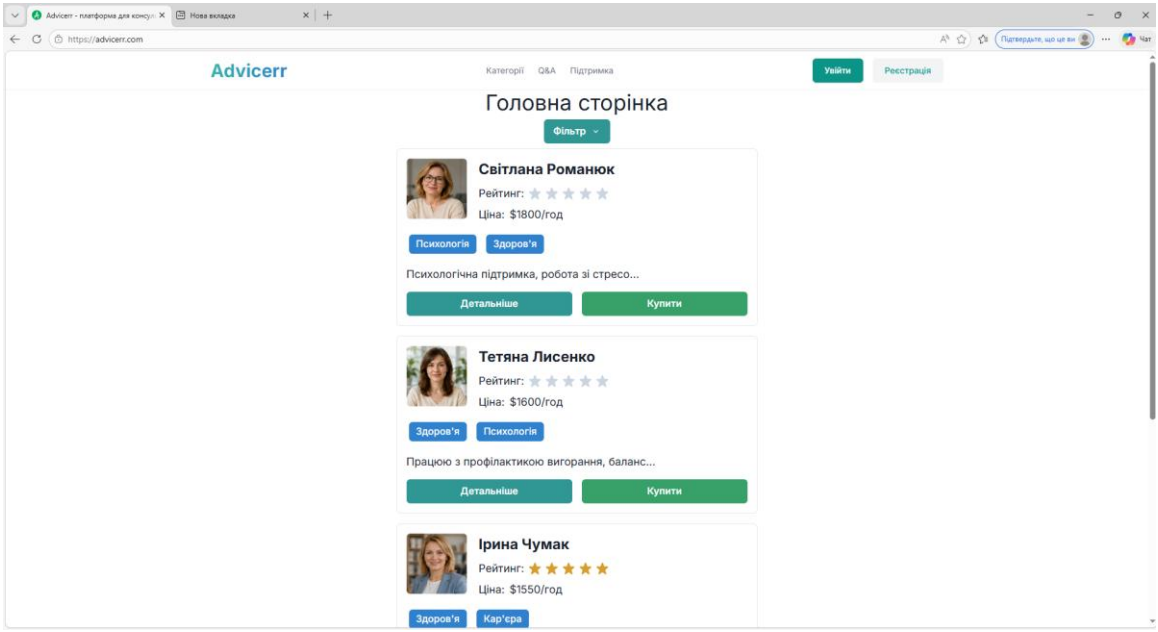

Рисунок 3.3 – Відфільтрований результат

Перевірка реєстрації користувача виконується через сторінку реєстрації. Користувач вводить ім’я, прізвище, email і пароль, після чого система надсилає дані на backend. Очікуваним результатом є створення нового запису в таблиці users. Також потрібно перевірити, що система не дозволяє створити двох користувачів з однаковим email.

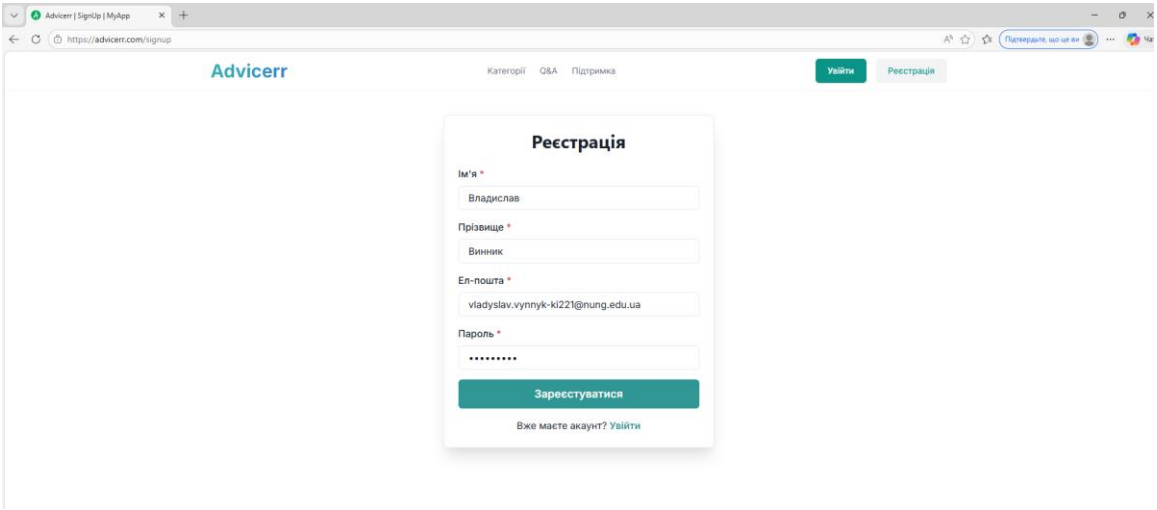

Рисунок 3.4 – Форма реєстрації

Перевірка авторизації виконується через сторінку входу. Користувач вводить email і пароль. Якщо дані правильні, backend повертає access token і refresh token,

які frontend зберігає в cookies. Після цього користувач отримує доступ до особистого кабінету.

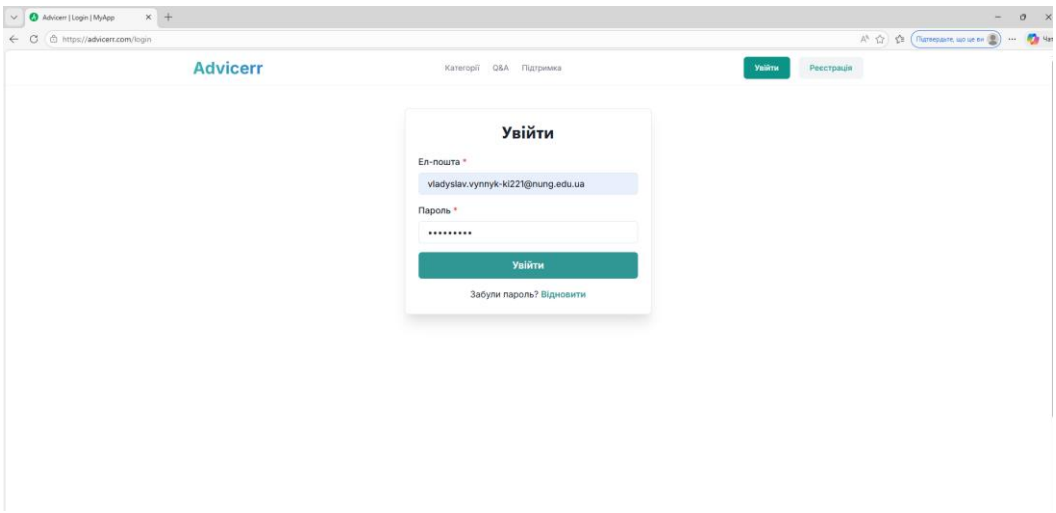

Рисунок 3.5 – Форма для входу в кабінет

Бачимо, що авторизація пройшла успішно. Після введення коректних облікових даних система обробила запит на серверній частині, перевірила email і пароль користувача та надала доступ до особистого кабінету. Це підтверджує, що механізм входу в систему працює правильно, а frontend і backend коректно взаємодіють між собою під час виконання сценарію авторизації.

Після успішного входу користувач отримує можливість працювати з функціями, доступними лише авторизованим користувачам. Зокрема, він може переглядати особисту інформацію, редагувати профіль, створювати консультаційні запити та працювати з іншими розділами системи відповідно до своєї ролі.

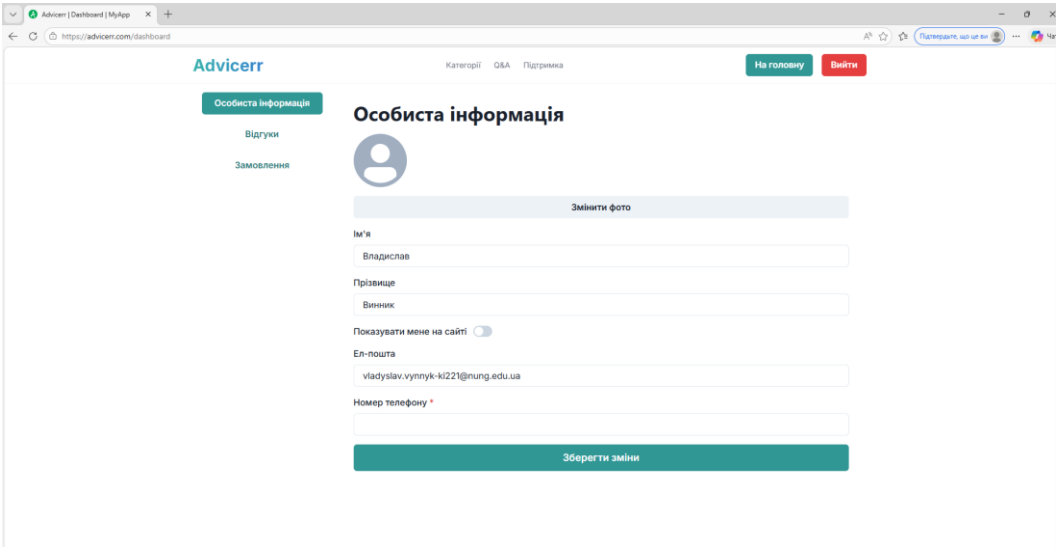

Рисунок 3.6 – Особистий кабінет з формою редагування профілю

Наступне, що потрібно перевірити — редагування профілю. Воно виконується в особистому кабінеті. Користувач змінює персональні дані, наприклад ім’я, прізвище, телефон, опис, ціну консультації або теги. Після збереження змін потрібно переконатися, що frontend показує оновлені дані, а backend зберігає зміни в таблиці users і пов’язаних таблицях. Це підтверджує коректну роботу механізму оновлення профілю користувача.

Для перевірки завантаження фото користувача потрібно обрати зображення, обрізати його через інтерфейс і зберегти. Очікуваним результатом є завантаження файлу до Amazon S3 та оновлення поля photo у таблиці users.

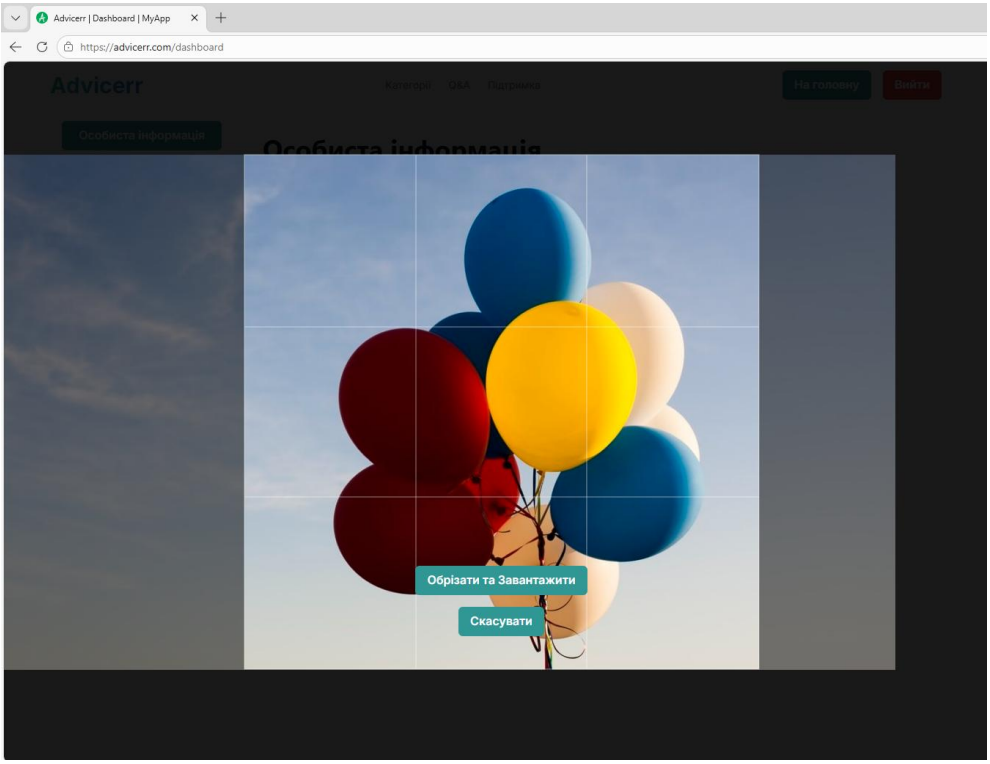

Рисунок 3.7 – Завантаження та обрізання фото

В результаті бачимо, що фото було успішно оновлено. Після вибору нового зображення система обробила файл, передала його на серверну частину та оновила дані профілю користувача. Нове фото відображається в особистому кабінеті, що підтверджує коректну роботу механізму завантаження та оновлення аватара.

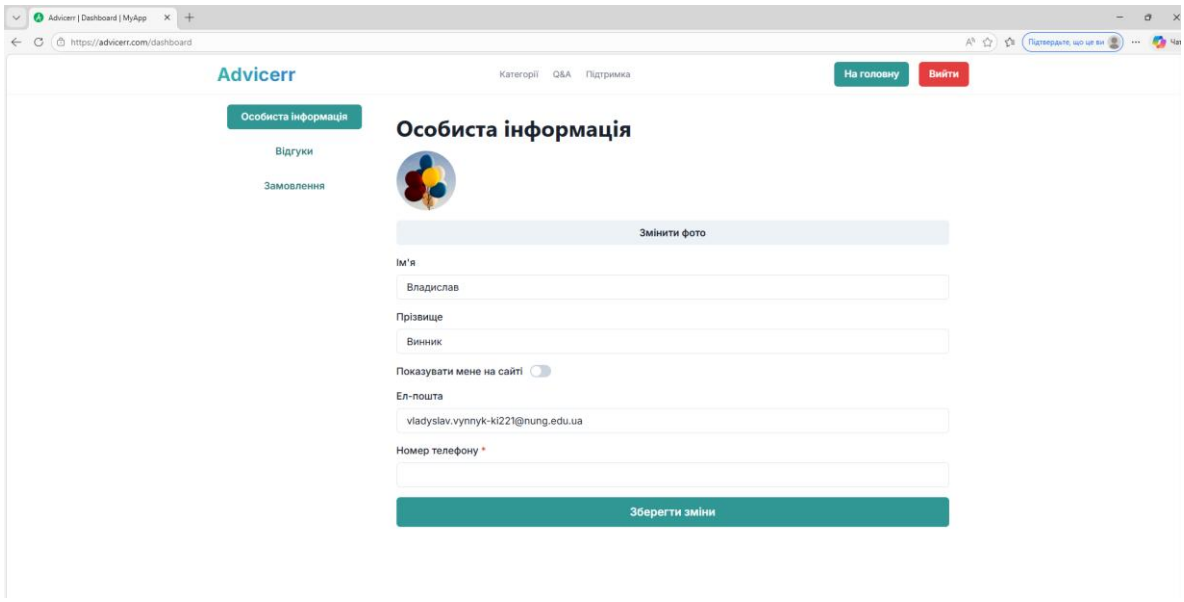

Рисунок 3.8 – Оновлений кабінет

Перевірка створення консультаційного запиту є одним із головних етапів тестування. Користувач відкриває картку консультанта, заповнює форму консультації та надсилає її. Після цього backend повинен створити запис у таблиці order зі статусом new. Якщо користувач не авторизований, система повинна обробити його контактні дані та створити або знайти відповідного користувача.

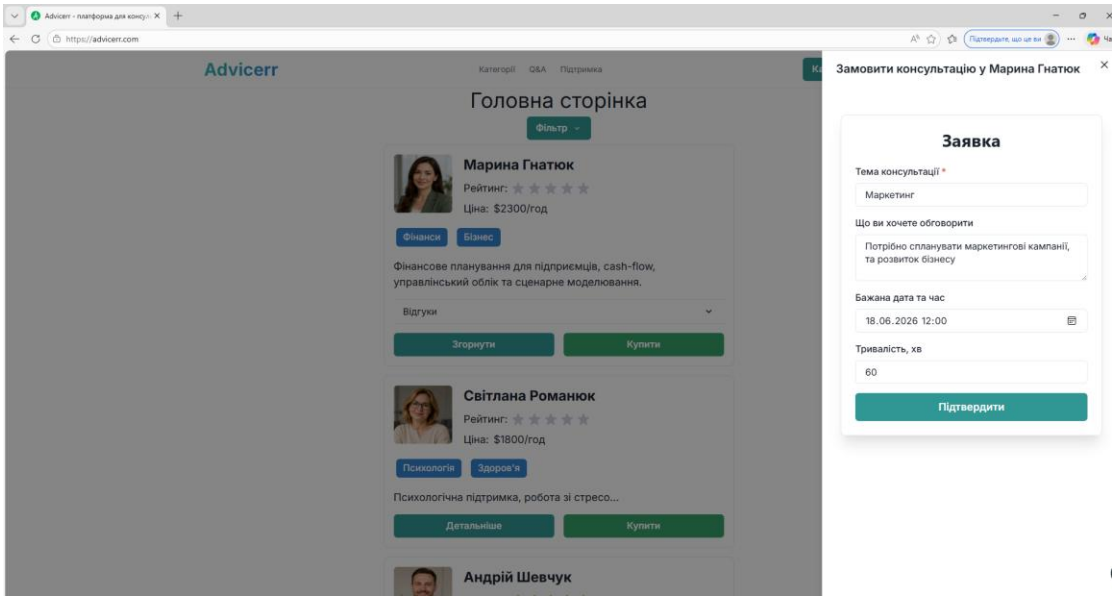

Рисунок 3.9 – Форма для створення заявки

В кабінеті консультанта можемо побачити, що новостворена заявка відображається зі статусом “Нова”. Це підтверджує, що після створення консультаційного запиту дані були успішно передані з клієнтської частини на сервер, збережені в базі даних і коректно відображені в особистому кабінеті консультанта.

Статус “Нова” показує початковий стан заявки, тобто вона ще не була підтверджена, взята в роботу або завершена. Завдяки цьому консультант може швидко бачити нові звернення від клієнтів і надалі змінювати їхній статус відповідно до етапу опрацювання консультації.

Також це демонструє правильну роботу зв’язку між клієнтом, консультантом і консультаційним запитом у базі даних. Заявка відображається саме в кабінеті відповідного консультанта, що підтверджує коректне використання зв’язків між таблицею користувачів і таблицею консультаційних запитів.

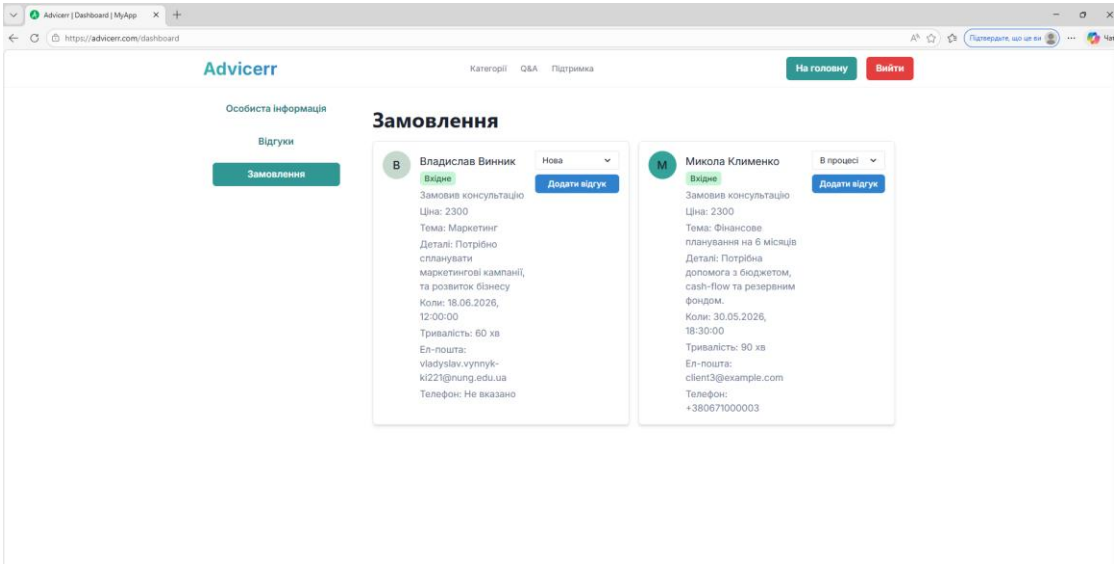

Рисунок 3.10 – Новостворена заявка

Для того, щоб змінити статус консультації, потрібно, щоб користувач з відповідними правами змінив статус консультації. Код на серверній частині перевіряє права користувача і допустимість переходу статусу. Після успішної зміни новий статус повинен відображатися в інтерфейсі та зберігатися в базі даних.

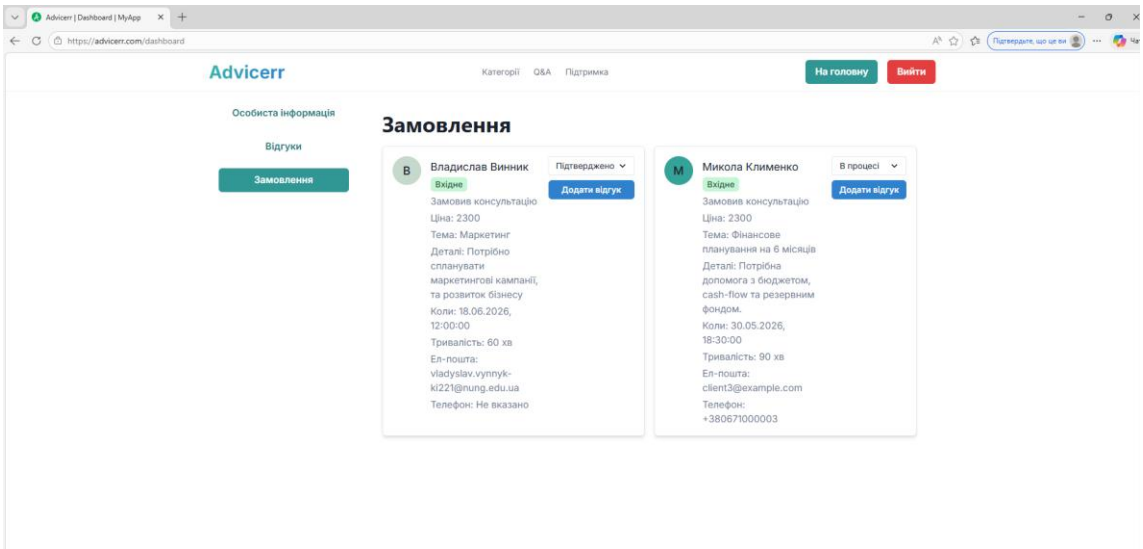

Рисунок 3.11 – Оновлений статус заявки

Перевірка роботи з відгуками виконується шляхом створення відгуку про консультанта. Користувач вводить текст відгуку і оцінку. Після надсилання backend створює запис у таблиці review. Далі потрібно перевірити, що відгук

відображається в картці консультанта, а рейтинг консультанта змінюється відповідно до оцінок.

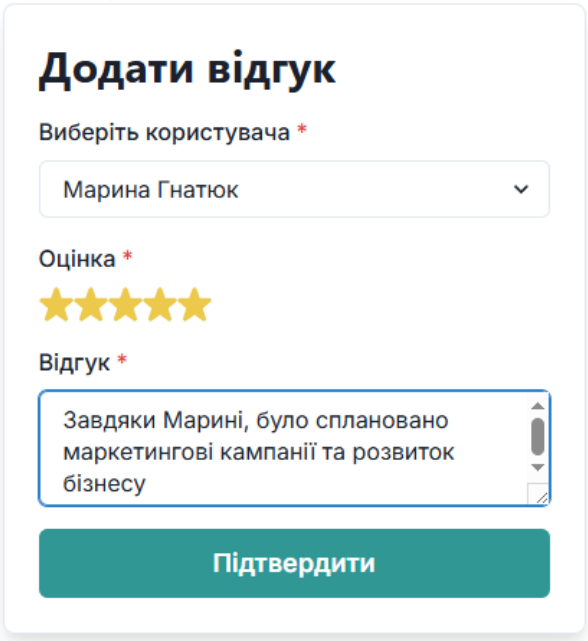

Рисунок 3.12 – Форма для створення відгуку

Після створення відгуку система зберігає введений текст та оцінку в базі даних. Далі відгук відображається в картці відповідного консультанта, що дозволяє іншим користувачам переглядати попередні оцінки та коментарі. Це підтверджує, що механізм створення і виведення відгуків працює коректно, а зв’язок між клієнтом, консультантом і записом відгуку реалізовано правильно.

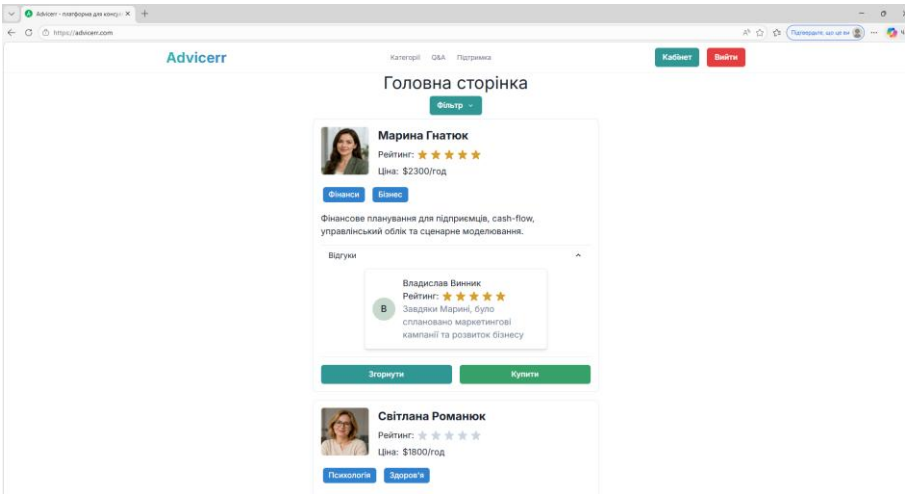

Рисунок 3.13 – Картка консультанта з відгуком

Перевірка адміністративного функціоналу виконується під обліковим записом користувача з ознакою is_admin = true. Після входу в систему адміністратор повинен бачити додаткові розділи dashboard.

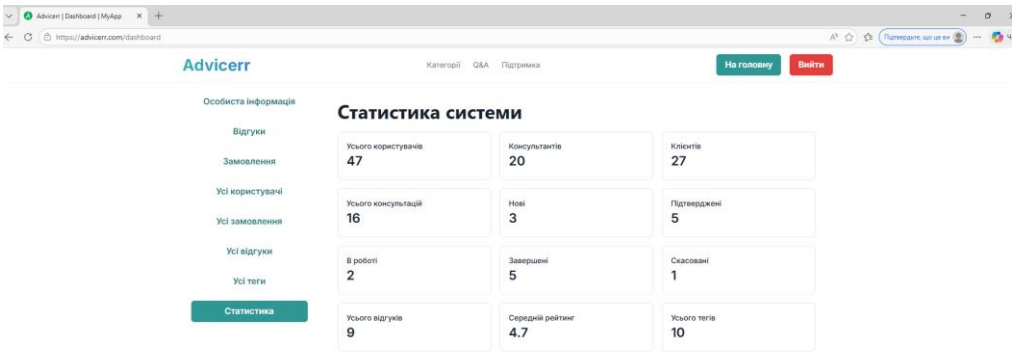

Рисунок 3.14 – Кабінет адміністратора

У цих розділах адміністратор має можливість переглядати основні дані системи, зокрема користувачів, консультаційні запити, відгуки, теги та статистичну інформацію. Також передбачено керування тегами, зміна статусів консультаційних запитів і надання або скасування адміністративних прав для окремих користувачів.

ВИСНОВКИ

Під час виконання бакалаврської роботи було систематизовано, закріплено та розширено теоретичні і практичні знання, набуті під час вивчення спеціальних дисциплін, а також застосовано їх для розв’язання конкретної технічної задачі. У межах роботи було розроблено веб-орієнтовану систему менеджменту консультаційних послуг, яка забезпечує підтримку основних процесів взаємодії між клієнтами, консультантами та адміністратором системи.

На першому етапі було проаналізовано процес менеджменту консультаційних послуг як технічну задачу. Визначено, що інформаційні системи дозволяють спростити пошук консультантів, структурувати дані про користувачів і послуги, підтримувати створення консультаційних запитів та зменшувати кількість ручних операцій. Також було розглянуто існуючі рішення у сфері підтримки консультаційних послуг, визначено їхні переваги та недоліки. На основі проведеного аналізу сформульовано постановку задачі, функціональні та нефункціональні вимоги до розроблюваної системи.

На етапі проєктування було визначено структуру веб-орієнтованої системи та обґрунтовано вибір засобів реалізації. Для створення клієнтської частини використано Next.js, React і TypeScript, для серверної частини — FastAPI, SQLAlchemy та Pydantic. Як основну базу даних використано PostgreSQL, а Redis застосовано для кешування і тимчасового зберігання даних. Для контейнеризації системи використано Docker і Docker Compose, а для розгортання на базі платформи AWS — Terraform, Amazon EC2 та Amazon S3.

У другому розділі також було описано структуру бази даних і функціональне призначення її полів. База даних містить таблиці користувачів, консультаційних запитів, відгуків, тегів, зв’язків між користувачами і тегами, а також платіжної інформації. Така структура дозволяє зберігати основні дані системи, підтримувати ролі клієнта, консультанта та адміністратора, а також забезпечувати зв’язки між ключовими сутностями. Крім цього, було розроблено діаграму варіантів

використання та діаграму послідовності для опису основних сценаріїв роботи програми.

На етапі програмної реалізації було створено клієнтську та серверну частини системи. Клієнтська частина забезпечує відображення головної сторінки, карток консультантів, форм реєстрації та авторизації, особистого кабінету, адміністративних розділів і форм для створення консультаційних запитів. Серверна частина реалізує API для роботи з користувачами, авторизацією, консультаційними запитами, відгуками, тегами, завантаженням фото та адміністративними функціями.

Для роботи з базою даних використано SQLAlchemy ORM, що дозволило описати таблиці у вигляді моделей та організувати доступ до даних через DAO-рівень. Було реалізовано механізми реєстрації, авторизації користувачів, відновлення пароля, редагування профілю, створення консультаційного запиту, перегляду консультацій, зміни статусів, роботи з відгуками та адміністративного керування. Для зберігання аватарів користувачів використано Amazon S3, що дозволяє винести файлові дані за межі основного контейнера застосунку.

Після реалізації основних компонентів було проведено перевірку функціонування системи. Перевірка охоплювала запуск застосунку, роботу головної сторінки, реєстрацію, авторизацію, відновлення пароля, редагування профілю, завантаження фото, створення консультаційного запиту, перегляд власних консультацій, зміну статусу, роботу з відгуками, адміністративну панель. У результаті перевірки підтверджено, що система виконує основні поставлені задачі та забезпечує взаємодію між frontend, backend і базою даних.

Отже, у результаті виконання бакалаврської роботи було розроблено веб-орієнтовану систему менеджменту консультаційних послуг, яка поєднує інтерфейс користувача, серверну логіку, базу даних та інфраструктуру для розгортання. Розроблений програмний продукт відповідає поставленій задачі, підтримує основні сценарії роботи користувачів і може бути використаний як основа для подальшого розвитку.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Next.js Documentation. URL: https://nextjs.org/docs (дата звернення: 04.06.2026).
2. React Reference Documentation. URL: https://react.dev/reference/react (дата звернення: 03.04.2026).
3. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 03.04.2026).
4. Chakra UI Documentation. URL: https://chakra-ui.com/docs/components/concepts/overview (дата звернення: 04.06.2026).
5. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs (дата звернення: 03.04.2026).
6. Redux Toolkit Documentation. URL: https://redux-toolkit.js.org/ (дата звернення: 03.04.2026).
7. RTK Query Overview. URL: https://redux-toolkit.js.org/rtk-query/overview (дата звернення: 05.04.2026).
8. i18next Documentation. URL: https://www.i18next.com/ (дата звернення: 05.04.2026).
9. js-cookie Documentation. URL: https://www.npmjs.com/package/js-cookie (дата звернення: 09.04.2026).
10. 12.react-easy-crop Documentation. URL: https://www.npmjs.com/package/react-easy-crop (дата звернення: 09.04.2026).
11. FastAPI Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 17.04.2026).
12. Uvicorn Documentation. URL: https://www.uvicorn.org/ (дата звернення: 17.04.2026).
13. Pydantic Documentation. URL: https://pydantic.dev/docs/ (дата звернення: 17.04.2026).
14. SQLAlchemy Documentation. URL: https://docs.sqlalchemy.org/ (дата звернення: 17.04.2026).

15. Alembic Documentation. URL: https://alembic.sqlalchemy.org/ (дата звернення: 02.05.2026).

16. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 02.05.2026).

17. Redis Documentation. URL: https://redis.io/docs/latest/ (дата звернення: 02.05.2026).

18. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 02.05.2026).

19. Docker Compose Documentation. URL: https://docs.docker.com/compose/ (дата звернення: 02.05.2026).

20. Traefik Proxy Documentation. URL: https://doc.traefik.io/traefik/ (дата звернення: 08.05.2026).

21. Terraform Documentation. URL: https://developer.hashicorp.com/terraform/docs (дата звернення: 08.05.2026).

22. Amazon EC2 Documentation. URL: https://docs.aws.amazon.com/ec2/ (дата звернення: 08.05.2026).

23. Amazon S3 User Guide. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (дата звернення: 12.05.2026).

24. AWS Identity and Access Management Documentation. URL: https://docs.aws.amazon.com/iam/ (дата звернення: 12.05.2026).

25. Boto3 Documentation. URL: https://docs.aws.amazon.com/boto3/latest/ (дата звернення: 12.05.2026).

Додатки

ДОДАТОК А

Програмне забезпечення для розробленої системи

Головний файл backend-сервісу

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from routers.users.create import create_users_router
from routers.users.read import read_users_router
from routers.users.update import update_users_router
from routers.users.delete import delete_users_router
from routers.users.me import me_users_router
from routers.users.auth import auth_router

from routers.admin_router import admin_router
from routers.orders_router import orders_router
from routers.tags_router import tags_router
from routers.reviews_router import reviews_router


origins = ["*", "localhost:3000","https://consulting-frontend-
production.vercel.app"]

app = FastAPI()


app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(create_users_router)
app.include_router(read_users_router)
app.include_router(update_users_router)
app.include_router(delete_users_router)
app.include_router(me_users_router)
app.include_router(auth_router)


app.include_router(admin_router)
app.include_router(orders_router)
app.include_router(tags_router)
app.include_router(reviews_router)


@app.get("/")
def read_root():
    return {"Hello": "World"}
```

Файл orders_router.py

```
from __future__ import annotations


from fastapi import APIRouter, Depends, HTTPException, status
```

```
from database.models import Order, User
from pydantic_models import ConsultationSchema, RegisteredUserSchema,
UnregisteredUserSchema
from daos.orders_dao import OrdersDAO
from daos.users_dao import UsersDAO
from deps import get_current_user
from serializers import serialize_order


from routers.users.read import get_user


from settings import get_settings


db_uri = get_settings().db_uri


orders_router = APIRouter(
    prefix="/orders",
)
@orders_router.post("/")
def add_consultation(consultation_data: ConsultationSchema):
    consultation = consultation_data.consultation
    auth_flow = {
        "email": None,
        "requires_complete_registration": False,
        "should_login": False,
        "message": None,
    }


    # Handle registered user case
    if isinstance(consultation, RegisteredUserSchema):
        if consultation.consultant_id == consultation.client_id:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="You cannot order your own consultation.",
            )
```



```
        consultation_for_insert = Order(
```

```
            price=consultation.price,
            status="new",
            topic=consultation.topic,
            message=consultation.message,
            scheduled_at=consultation.scheduled_at,
            duration_minutes=consultation.duration_minutes,
            consultant_id=consultation.consultant_id,
            client_id=consultation.client_id
        )
    # Handle unregistered user case
    elif isinstance(consultation, UnregisteredUserSchema):
        users_dao = UsersDAO(uri=db_uri)
        existing_user = users_dao.get_user_by_email(consultation.email)
        current_user_id = None

        if existing_user is not None:
            existing_user_record = existing_user[0]
            current_user_id = existing_user_record.id
            auth_flow["email"] = consultation.email

            if existing_user_record.password:
                auth_flow["should_login"] = True
                auth_flow["message"] = "Користувач уже зареєстрований. Увійдіть
у систему, щоб відстежувати статус консультації."
            else:
                auth_flow["requires_complete_registration"] = True
                auth_flow["message"] = "Заяву створено. Завершіть реєстрацію,
щоб відстежувати статус консультації."
        else:
            user_for_insert = User(
                first_name=consultation.first_name,
                last_name=consultation.last_name,
                email=consultation.email,
                phone_number=consultation.phone_number,
            )
            created_user = users_dao.create_user(user_for_insert)
```

Продовження додатку А

```
current_user_id = created_user.id
```

```
            auth_flow["email"] = consultation.email
            auth_flow["requires_complete_registration"] = True
            auth_flow["message"] = "Заяву створено. Завершіть реєстрацію, щоб відстежувати статус консультації."
        consultation_for_insert = Order(
            price=consultation.price,
            status="new",
            topic=consultation.topic,
            message=consultation.message,
            scheduled_at=consultation.scheduled_at,
            duration_minutes=consultation.duration_minutes,
            consultant_id=consultation.consultant_id,
            client_id=current_user_id
        )


    print("Consultation: ", consultation)
    consultations_dao = OrdersDAO(uri=db_uri)
    created_consultation = consultations_dao.create_consultation(consultation_for_insert)


    consultation_with_relations = consultations_dao.get_consultation_by_id(created_consultation.id)
    return {
        "order": serialize_order(consultation_with_relations[0]),
        "auth_flow": auth_flow,
    }
@orders_router.get("/")
def get_consultations():
    consultations_dao = OrdersDAO(uri=db_uri)
    consultations = consultations_dao.get_all_consultations()
    return [serialize_order(order) for order in consultations]


@orders_router.get("/{id}")
def get_consultation(consultation_id: str):
    consultations_dao = OrdersDAO(uri=db_uri)
    consultation = consultations_dao.get_consultation_by_id(consultation_id)
```

Продовження додатку А

```
      response = {"id": consultation[0].id, "first_name_of_client":
consultation[0].first_name_of_client, "surname_of_client":
consultation[0].surname_of_client,

                "phone_number_of_client":
consultation[0].phone_number_of_client, "messenger": consultation[0].messenger,
"consultant_id": consultation[0].consultant_id}

    return response


# get all consultations which is not paid for single consultant

@orders_router.get("/unpaid/consultant/{consultant_id}", summary="Get not paid
consultations for single consultant")

def get_consultations(consultant_id: str):

    consultations_dao = OrdersDAO(uri=db_uri)

    consultations = consultations_dao.get_unpaid_consultations(consultant_id)

    return [serialize_order(order) for order in consultations]


# get all consultations which is paid for single consultant

@orders_router.get("/paid/consultant/{consultant_id}", summary="Get paid
consultations for single consultant")

def get_consultations(consultant_id: str):

    consultations_dao = OrdersDAO(uri=db_uri)

    consultations = consultations_dao.get_paid_consultations(consultant_id)

    return [serialize_order(order) for order in consultations]


# get consultations created by client

@orders_router.get("/account/client/{client_id}", summary="Get consultations for
single client")

def get_consultations(client_id: str, current_user: User =
Depends(get_current_user)):

    if str(current_user.id) != client_id and not current_user.is_admin:

        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="You
cannot view these consultations.")

    consultations_dao = OrdersDAO(uri=db_uri)

    consultations = consultations_dao.get_consultations_by_client_id(client_id)

    return [serialize_order(order, viewer=current_user) for order in
consultations]


# get consultations, created by consultant

@orders_router.get("/account/consultant/{consultant_id}", summary="Get
consultations for consultant")

def get_consultations(consultant_id: str, current_user: User =
Depends(get_current_user)):
```

```
if str(current_user.id) != consultant_id and not current_user.is_admin:

        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="You
cannot view these consultations.")

    consultations_dao = OrdersDAO(uri=db_uri)

    consultations =
consultations_dao.get_consultations_by_consultant_id(consultant_id)

    return [serialize_order(order, viewer=current_user) for order in
consultations]


@orders_router.get("/account/{user_id}")

def get_consultations(user_id: str, current_user: User =
Depends(get_current_user)):

    if str(current_user.id) != user_id and not current_user.is_admin:

        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="You
cannot view these consultations.")

    orders_dao = OrdersDAO(uri=db_uri)

    orders = orders_dao.get_consultations_by_user_id(user_id)

    return [serialize_order(order, viewer=current_user) for order in orders]


@orders_router.patch("/{id}")

def update_consultation(consultation_id: str, updated_consultation:
ConsultationSchema):

    consultations_dao = OrdersDAO(uri=db_uri)

    # print("model dump: ", updated_consultant.model_dump())

    consultation_to_update =
consultations_dao.patch_consultation(consultation_id,
updated_consultation.model_dump())

    return consultation_to_update

@orders_router.delete("/{id}")

def delete_consultant(consultation_id: str):

    consultations_dao = OrdersDAO(uri=db_uri)

    consultation_to_delete =
consultations_dao.delete_consultation(consultation_id)

    deleted_consultation =
consultations_dao.delete_consultation(consultation_to_delete[0])

    response = {"id": deleted_consultation[0].id, "first_name_of_client":
deleted_consultation[0].first_name_of_client, "surname_of_client":
deleted_consultation[0].surname_of_client,

                "phone_number_of_client":
deleted_consultation[0].phone_number_of_client, "messenger":
deleted_consultation[0].messenger, "consultant_id":
deleted_consultation[0].consultant_id}

    return response
```

Файл MainContainer.tsx

```
import React from 'react';
import Head from "next/head";
import {Header} from "../Header/Header";
import {Footer} from "../Footer/Footer";

type Props = {
    children: JSX.Element
}

const MainContainer: React.FC<Props> = ({children}: Props) => {
    return (
        <>
            <Head>
                {/* <title>Advicerr</title>
                <meta name="description" content="Generated by create next app"
/> */}
                <meta name="viewport" content="width=device-width, initial-
scale=1" />
                {/*{BOXICONS}*/}
                <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/boxicons@latest/css/boxicons.min.css"/>
                {/*Bootstrap*/}
                <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC"
crossOrigin="anonymous"></link>

                <link rel="icon" href="/favicon.ico" />
            </Head>
            <Header/>
                <main className={`container-lg max-w-screen-xl px-4 sm:px-6
lg:px-8 mx-auto`} style={{minHeight: "90vh"}}>
                    {children}
                </main>
            <Footer/>
        </>
    );
};

export default MainContainer;
```

Файл LoginForm.tsx

```
"use client"
import React from 'react';
import Link from 'next/link';
import { useRouter, useSearchParams } from 'next/navigation'
import Cookies from 'js-cookie';

import { Alert, AlertIcon, Box, Button, FormControl, FormLabel, Input, VStack,
Heading, Text, FormErrorMessage } from '@chakra-ui/react';

import { useLoginUserMutation } from '@/app/store/apis/usersApi';

import usePrefixedTranslation from '@/app/hooks/usePrefixedTranslation';

const LoginForm = () => {
```

```
  const { t } = usePrefixedTranslation('Components.LoginForm');
  const [loginUser, { data, isSuccess, isLoading }] = useLoginUserMutation();
  const router = useRouter();
  const searchParams = useSearchParams();

  // Form state
  const [email, setEmail] = React.useState(searchParams.get('email') ?? '');
  const [password, setPassword] = React.useState('');
  const [error, setError] = React.useState<string | null>(null);
  const infoMessage = searchParams.get('after_order') === 'true'
    ? 'Заявку створено. Увійдіть у систему, щоб відстежувати статус
консультації.'
    : null;


  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();

    const formData = new FormData();
    formData.append('username', email);
    formData.append('password', password);

    try {
      const response = await loginUser(formData).unwrap();
      console.log('Login successful:', response);
    } catch (err: any) {
      console.error('Login failed:', err);
      setError(err.data?.message || t('loginErrorLabel'));
    }
  };

  React.useEffect(() => {
    if (isSuccess && data?.access_token && data?.refresh_token) {
      console.log('Login successful:', data);
      Cookies.set('access_token', data.access_token);
      Cookies.set('refresh_token', data.refresh_token);

      router.push('/dashboard'); // Redirect to dashboard
    }
  }, [isSuccess, data, router]);

  return (
    <Box
      borderWidth="1px"
      borderRadius="lg"
      overflow="hidden"
      p={6}
      maxW="md"
      mx="auto"
      mt={10}
      boxShadow="xl"
    >
      <Heading as="h1" size="lg" textAlign="center" mb={6}>
        {t("logInLabel")}
      </Heading>
      <form onSubmit={handleSubmit}>
        <VStack spacing={4}>
          <FormControl id="email" isRequired >
            <FormLabel>{t("emailLabel")}</FormLabel>
            <Input
              type="email"
              placeholder={t("emailInputPlaceholder")}
```

```
value={email}
              onChange={(e) => {
                setEmail(e.target.value);
              }}
            />
          </FormControl>

          <FormControl id="password" isRequired>
            <FormLabel>{t("passwordLabel")}</FormLabel>
            <Input
              type="password"
              placeholder={t("passwordInputPlaceholder")}
              value={password}
              onChange={(e) => {
                setPassword(e.target.value);
              }}
            />
          </FormControl>

          {error && (
            <Alert status="error">
              <AlertIcon />
              {error}
            </Alert>
          )}

          {infoMessage && (
            <Alert status="info">
              <AlertIcon />
              {infoMessage}
            </Alert>
          )}

          <Button colorScheme="teal" size="lg" width="full" type="submit"
isLoading={isLoading}>
            {t("logInButtonLabel")}
          </Button>
        </VStack>
      </form>
      <Text textAlign="center" mt={4}>
        {t("forgotPasswordLabel")} <Button variant="link"
colorScheme="teal"><Link
href={"/reset"}>{t("recoverPasswordLabel")}</Link></Button>
      </Text>
    </Box>
  );
};

export default LoginForm;
```

Файл store.ts

```
import { configureStore } from '@reduxjs/toolkit'
import { setupListeners } from '@reduxjs/toolkit/query'
import { usersApi } from './apis/usersApi'
import { paymentsApi } from './apis/paymentsApi'
import { ordersApi } from './apis/ordersApi'
import { reviewsApi } from './apis/reviewsApi'
import { passwordRecoverApi } from './apis/passwordRecoverApi'
import { tagsApi } from './apis/tagsApi'
```

```
export const store = configureStore({
  reducer: {
    [usersApi.reducerPath]: usersApi.reducer,
    [paymentsApi.reducerPath]: paymentsApi.reducer,
    [ordersApi.reducerPath]: ordersApi.reducer,
    [reviewsApi.reducerPath]: reviewsApi.reducer,
    [passwordRecoverApi.reducerPath]: passwordRecoverApi.reducer,
    [tagsApi.reducerPath]: tagsApi.reducer
  },

  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(usersApi.middleware, paymentsApi.middleware,
ordersApi.middleware,
      reviewsApi.middleware, passwordRecoverApi.middleware, tagsApi.middleware),
})

// setupListeners(store.dispatch)
export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

Файл ordersApi.ts

```
import { createApi } from '@reduxjs/toolkit/query/react'
import { Order, TCreateOrderResponse, TOrderForRegisteredUser,
TOrderForUnregisteredUser, TOrderStatusUpdate } from '@/app/types/OrderTypes'
import { apiBaseQuery } from './apiClient'


export const ordersApi = createApi({
  reducerPath: 'ordersApi',
  baseQuery: apiBaseQuery,
  endpoints: (builder) => ({
    getAllOrders: builder.query<Array<Order>, { id: string; token: string }>({
      query: ({ id, token }) => ({
        url: `orders/account/${id}`,
        headers: {
          "Authorization": `Bearer ${token}`
        }
      }),
    }),
    getAdminOrders: builder.query<Array<Order>, string>({
      query: (token) => ({
        url: `orders/admin/all`,
        headers: {
          "Authorization": `Bearer ${token}`
        }
      }),
    }),
    createOrder: builder.mutation<TCreateOrderResponse, TOrderForRegisteredUser
| TOrderForUnregisteredUser>({
      query: (body) => ({
        url: "orders/",
        method: "POST",
        body: body
      }),
    }),
    updateOrderStatus: builder.mutation<Order, { token: string; orderId: string;
body: TOrderStatusUpdate }>({
      query: ({ token, orderId, body }) => ({
        url: `orders/${orderId}/status`,
        method: "PATCH",
```

```
headers: {
          "Authorization": `Bearer ${token}`
        },
        body
      }),
    }),
  }),
})

export const { useGetAllOrdersQuery, useGetAdminOrdersQuery,
useCreateOrderMutation, useUpdateOrderStatusMutation } = ordersApi
```

Файл docker-compose.prod.yml

```
services:
  traefik:
    image: traefik:v3.3
    restart: unless-stopped
    command:
      - --providers.docker=true
      - --providers.docker.endpoint=unix:///var/run/docker.sock
      - --providers.docker.exposedbydefault=false
      - --entrypoints.web.address=:80
      - --entrypoints.websecure.address=:443
      - --entrypoints.web.http.redirections.entrypoint.to=websecure
      - --entrypoints.web.http.redirections.entrypoint.scheme=https
      - --certificatesresolvers.letsencrypt.acme.email=${LETSENCRYPT_EMAIL}
      - --certificatesresolvers.letsencrypt.acme.storage=/letsencrypt/acme.json
      - --certificatesresolvers.letsencrypt.acme.httpchallenge=true
      - --certificatesresolvers.letsencrypt.acme.httpchallenge.entrypoint=web
    ports:
      - "80:80"
      - "443:443"
    environment:
      DOCKER_API_VERSION: ""
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - traefik_letsencrypt:/letsencrypt
    networks:
      - web

  frontend:
    build:
      context: ./advicerr-frontend
      dockerfile: prod.Dockerfile
      args:
        NEXT_PUBLIC_URL_TO_API: https://${DOMAIN}/api/
    restart: unless-stopped
    depends_on:
      - backend
    environment:
      NEXT_PUBLIC_URL_TO_API: https://${DOMAIN}/api/
    labels:
      - traefik.enable=true
      - traefik.docker.network=bachelors_prod_web
      - traefik.http.routers.frontend-prod.rule=Host(`${DOMAIN}`)
      - traefik.http.routers.frontend-prod.entrypoints=websecure
      - traefik.http.routers.frontend-prod.tls=true
      - traefik.http.routers.frontend-prod.tls.certresolver=letsencrypt
      - traefik.http.services.frontend-prod.loadbalancer.server.port=3000
    networks:
```

```
- web

  backend:
    build:
      context: .
      dockerfile: consulting-backend/Dockerfile
    restart: unless-stopped
    env_file:
      - ./.env
    environment:
      DB_URI:
postgresql+psycopg2://${POSTGRES_USER}:${POSTGRES_PASSWORD}@db:5432/${POSTGRES_D
B}
      REDIS_HOST: redis
      REDIS_PORT: 6379
      UVICORN_RELOAD: "false"
    depends_on:
      db:
        condition: service_healthy
      redis:
        condition: service_started
    labels:
      - traefik.enable=true
      - traefik.docker.network=bachelors_prod_web
      - traefik.http.routers.backend-prod.rule=Host(`${DOMAIN}`) &&
PathPrefix(`/api`)
      - traefik.http.routers.backend-prod.entrypoints=websecure
      - traefik.http.routers.backend-prod.tls=true
      - traefik.http.routers.backend-prod.tls.certresolver=letsencrypt
      - traefik.http.routers.backend-prod.middlewares=backend-prod-strip
      - traefik.http.middlewares.backend-prod-strip.stripprefix.prefixes=/api
      - traefik.http.services.backend-prod.loadbalancer.server.port=8000
    networks:
      - web
      - internal

  db:
    image: postgres:15-alpine
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test:
        - CMD-SHELL
        - pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}
      interval: 10s
      timeout: 5s
      retries: 10
    networks:
      - internal

  redis:
    image: redis:7-alpine
    restart: unless-stopped
    command: redis-server --appendonly yes
    networks:
      - internal
```

```
networks:
  web:
    name: bachelors_prod_web
  internal:
    name: bachelors_prod_internal

volumes:
  postgres_data:
  traefik_letsencrypt:
```

Файл s3_avatars/main.tf

```
terraform {
  required_version = ">= 1.5.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}

resource "aws_s3_bucket" "avatars" {
  bucket = var.avatars_bucket_name

  tags = {
    Project = var.project_name
    Purpose = "consultant-avatars"
  }
}

resource "aws_s3_bucket_ownership_controls" "avatars" {
  bucket = aws_s3_bucket.avatars.id

  rule {
    object_ownership = "BucketOwnerPreferred"
  }
}

resource "aws_s3_bucket_public_access_block" "avatars" {
  bucket = aws_s3_bucket.avatars.id

  block_public_acls       = false
  block_public_policy     = false
  ignore_public_acls      = false
  restrict_public_buckets = false
}

resource "aws_s3_bucket_cors_configuration" "avatars" {
  bucket = aws_s3_bucket.avatars.id

  cors_rule {
    allowed_headers = ["*"]
    allowed_methods = ["GET", "HEAD"]
    allowed_origins = var.allowed_origins
    expose_headers  = ["ETag"]
```

```
max_age_seconds = 3000
  }
}

resource "aws_s3_bucket_policy" "avatars_public_read" {
  bucket = aws_s3_bucket.avatars.id

  depends_on = [
    aws_s3_bucket_public_access_block.avatars
  ]

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid       = "PublicReadGetObject"
        Effect    = "Allow"
        Principal = "*"
        Action    = ["s3:GetObject"]
        Resource  = "${aws_s3_bucket.avatars.arn}/*"
      }
    ]
  })
}
```

Файл deploy_prod.py

```
from __future__ import annotations

import json
import shutil
import subprocess
import sys
from pathlib import Path

try:
    import boto3
    from botocore.exceptions import BotoCoreError, NoCredentialsError
except ImportError as exc:  # pragma: no cover - environment dependent
    print("boto3 is required for production deploy automation.",
file=sys.stderr)
    raise SystemExit(1) from exc


ROOT_DIR = Path(__file__).resolve().parents[2]
PROD_INFRA_DIR = ROOT_DIR / "deployment" / "infra" / "prod"
UPLOAD_SCRIPT = ROOT_DIR / "deployment" / "scripts" / "upload_avatars_to_s3.py"


def ensure_command_exists(command_name: str) -> None:
    if shutil.which(command_name) is None:
        raise RuntimeError(f"`{command_name}` is not installed or is not
available in PATH.")


def ensure_aws_credentials() -> None:
    try:
        session = boto3.Session()
        credentials = session.get_credentials()
        if credentials is None:
            raise NoCredentialsError()
        sts = session.client("sts")
```

```
sts.get_caller_identity()
    except (BotoCoreError, NoCredentialsError, Exception) as exc:  # pragma: no
cover
        raise RuntimeError(
            "AWS credentials are not available. Configure the standard AWS
credentials chain before deployment."
        ) from exc


def run_command(command: list[str], cwd: Path) ->
subprocess.CompletedProcess[str]:
    return subprocess.run(
        command,
        cwd=str(cwd),
        check=True,
        capture_output=True,
        text=True,
    )


def terraform_outputs() -> dict[str, str]:
    output = run_command(["terraform", "output", "-json"], PROD_INFRA_DIR)
    payload = json.loads(output.stdout)
    return {key: value["value"] for key, value in payload.items()}


def upload_avatars(bucket_name: str, aws_region: str) -> None:
    command = [
        sys.executable,
        str(UPLOAD_SCRIPT),
        "--bucket-name",
        bucket_name,
        "--aws-region",
        aws_region,
        "--prefix",
        "avatars",
    ]
    result = run_command(command, ROOT_DIR)
    if result.stdout:
        print(result.stdout)
    if result.stderr:
        print(result.stderr, file=sys.stderr)


def main() -> int:
    try:
        ensure_command_exists("terraform")
        ensure_aws_credentials()
    except RuntimeError as exc:
        print(str(exc), file=sys.stderr)
        return 1

    print(f"Running Terraform in {PROD_INFRA_DIR}")
    try:
        init_result = run_command(["terraform", "init"], PROD_INFRA_DIR)
        print(init_result.stdout)

        apply_result = run_command(["terraform", "apply", "-auto-approve"],
PROD_INFRA_DIR)
        print(apply_result.stdout)
    except subprocess.CalledProcessError as exc:
        print(f"Terraform failed: {' '.join(exc.cmd)}", file=sys.stderr)
```

```
if exc.stdout:
            print(exc.stdout, file=sys.stderr)
        if exc.stderr:
            print(exc.stderr, file=sys.stderr)
        return 1

    outputs = terraform_outputs()

    try:
        upload_avatars(
            bucket_name=outputs["avatars_bucket_name"],
            aws_region=outputs["aws_region"],
        )
    except subprocess.CalledProcessError as exc:
        print("Avatar upload failed.", file=sys.stderr)
        if exc.stdout:
            print(exc.stdout, file=sys.stderr)
        if exc.stderr:
            print(exc.stderr, file=sys.stderr)
        return 1

    app_url = outputs["app_url"]
    ec2_ip = outputs["ec2_public_ip"]
    bucket_name = outputs["avatars_bucket_name"]
    avatars_base_url = outputs["avatars_base_url"]
    aws_region = outputs["aws_region"]
    dns_managed = outputs.get("dns_managed_by_route53", False)

    print("Production deploy finished.")
    print(f"App URL: {app_url}")
    print(f"EC2 public IP: {ec2_ip}")
    print(f"S3 bucket: {bucket_name}")
    print(f"Avatars base URL: {avatars_base_url}")
    print(f"AWS region: {aws_region}")
    print(f"DNS status: {'Route53 managed' if dns_managed else 'Manual DNS
required'}")
    print("SSH command:")
    print("  ssh -i /path/to/your-key.pem ubuntu@" + ec2_ip)
    print("Post-deploy checks:")
    print(f"  - open {app_url}")
    print(f"  - open {app_url}/api/docs")
    print("  - ssh into EC2 and run `docker ps`")
    print("  - inspect `docker logs <traefik-container>` if SSL is not issued")
    print("  - open several avatar URLs from
deployment/generated/avatar_urls.json")

    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: Розробка web-орієнтованої системи менеджменту консультаційних послуг на базі платформи AWS

Обсяг пояснювальної записки 56 аркушів:

7 таблиць;

24 рисунків;

1 додаток.

Дата завершення роботи: *10 червня 2025р*.

Підпис студента- _____*Винник В.І.*.