МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____Юрій КОНДРАТЕНКО

«____»_____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

ОНЛАЙН-СЕРВІС ПОШУКУ РОБОТИ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____ Катерина СИРЦЕВА

«___»_____ 2025 р.

Керівник старший викладач кафедри КІ _____ Євген ДАРНАПУК

«___»_____ 2025 р.

Миколаїв – 2025

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Юрій КОНДРАТЕНКО

«____» _____ 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Сирцевої Катерини Дмитрівни
(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «онлайн-сервіс пошуку роботи».

Керівник роботи: Дарнапук Євген Сергійович, старший викладач кафедри КІ.

Затверджена наказом ЧНУ ім. Петра Могили від «18» листопада 2024 р. № 321.

2. Строк представлення кваліфікаційної роботи «19» червня 2025 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: аналіз функціоналу сучасних онлайн-платформ для пошуку роботи, сучасні засоби розробки вебдодатків. Очікується реалізований вебсервіс з можливістю реєстрації користувачів, створення резюме, публікації вакансій, забезпечено функціонал пошуку та фільтрації вакансій.

4. Перелік питань, що підлягають розробці: обґрунтування актуальності теми та постановка мети дослідження; аналіз предметної області та наявних аналогів;

формування технічного завдання на створення онлайн-сервісу; обґрунтування вибору методів, інструментів та технологій розробки; проєктування архітектури інформаційної системи; розробка модулів авторизації, пошуку роботи, створення вакансій/резюме; реалізація взаємодії між frontend і backend; побудова структури бази даних; тестування функціональності системи.

5. Перелік графічних матеріалів: презентація.

Керівник роботи _____ Євген ДАРНАПУК
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Катерина СИРЦЕВА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «20» грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Онлайн-сервіс пошуку роботи

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Отримання завдання на виконання КР	18.12.2024	20.12.2024	Виконано
2.	Аналіз предметної області та постановка задачі	21.12.2024	30.01.2025	Виконано
3.	Огляд літературних джерел за темою кваліфікаційної роботи, зокрема огляд публікацій та аналогічних систем, щодо проектування онлайн-сервісу пошуку роботи	31.01.2025	01.03.2025	Виконано
4.	Огляд існуючих аналогів	02.03.2025	01.04.2025	Виконано
5.	Розробка проєктних рішень	02.04.2025	15.04.2025	Виконано
6.	Кодування та тестування розробленого онлайн-сервісу, аналіз результатів тестування	16.04.2025	15.05.2025	Виконано
7.	Перший попередній захист КР на засіданні комісії кафедри	16.05.2025	16.05.2025	Виконано
8.	Корегування роботи за результатами попереднього захисту	17.05.2025	29.05.2025	Виконано
9.	Другий попередній захист КР на засіданні комісії кафедри	30.05.2025	30.05.2025	Виконано
10.	Доробка та остаточне оформлення КР	31.05.2025	10.06.2025	Виконано
11.	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	11.06.2025	12.06.2025	Виконано

Керівник роботи _____ Євген ДАРНАПУК
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Катерина СИРЦЕВА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану «30» січня 2025 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувачки групи 402 ЧНУ ім. Петра Могили

Сирцевої Катерини Дмитрівни

на тему : **«ОНЛАЙН-СЕРВІС ПОШУКУ РОБОТИ»**

Актуальність цієї роботи полягає у тому, що під час воєнного стану в Україні значна частина населення втратила роботу або була змушена змінити місце проживання і це призводить до зростання соціальної нестабільності та необхідності швидкої адаптації до нових умов життя. Онлайн-сервіси стають важливим засобом підтримки економіки, інтеграції внутрішньо переміщених осіб та сприяють загальному процесу відновлення країни. У таких умовах особливої актуальності набувають онлайн-сервіси, які дозволяють швидко знайти роботу або найняти працівників незалежно від географічного розташування. Під час воєнного стану та у післявоєнний період такі сервіси будуть сприяти стабілізації економіки, підтримці громадян та забезпеченню їх зайнятості.

Мета роботи – розробка сучасного онлайн-сервісу пошуку роботи, який забезпечить зручний та швидкий доступ до вакансій, ефективну фільтрацію пропозицій, можливість створення резюме та подання заявок, а також використання елементів обробки даних для персоналізації результатів пошуку.

Об’єктом роботи є процес розробки онлайн-сервісу пошуку роботи.

Предмет роботи – програмні інструменти та технології для розробки онлайн-сервісу для пошуку роботи.

Кваліфікаційна робота складається із вступу, 3 розділів, висновків та переліку джерел посилання.

У вступі обґрунтовано актуальність теми, визначено мету, завдання та методи дослідження.

У першому розділі розглянуто теоретичні аспекти онлайн-сервісів пошуку роботи, проаналізовано існуючі рішення, їхні функції та недоліки.

Другий розділ присвячено перегляду засобів реалізації майбутнього сервісу, вибору технологій, розробці бази даних та інтерфейсу користувача.

У третьому розділі описано функції онлайн-застосунку реалізовано основні компоненти системи, проведено тестування, оцінено продуктивність і безпеку розробленого сервісу.

У висновках підсумовано результати виконаної роботи, визначено ефективність запропонованого рішення та перспективи його розвитку.

Кваліфікаційна робота викладена на 94 сторінках машинописного тексту, складається із вступу, 3 розділів, загальних висновків, переліку джерел посилання з 27 найменувань та 1 додатку. Праця містить 14 таблиць та 66 рисунків.

Ключові слова*:* онлайн-сервіс, пошук роботи, вакансія, резюме, вебдодаток, клієнт-серверна архітектура, база даних, користувацький інтерфейс.

ABSTRACT

to the qualification work by the student of the group 402 of Petro Mohyla Black Sea National University

Kateryna Syrtseva

“ ONLINE JOB SEARCH SERVICE ”

A relevance of the work lies in the fact that during martial law in Ukraine, a significant part of the population lost their jobs or was forced to relocate, which leads to increased social instability and the need for rapid adaptation to new living conditions. Online services become an important tool for supporting the economy, integrating internally displaced persons, and contributing to the overall recovery of the country. Under such conditions, online services that allow users to quickly find a job or hire employees regardless of geographic location are particularly relevant. During martial law and in the post-war period, such services will support economic stabilization, assist citizens, and ensure their employment.

An object of the work is the process of developing an online job search service.

A subject of the work is software tools and technologies for developing an online job search service.

A purpose of the work is to develop a modern online job search service that provides convenient and quick access to job vacancies, effective filtering of job offers, the ability to create resumes and submit applications, as well as the use of data processing elements for personalized search results.

The qualification work consists of an introduction, 3 sections, conclusions and a list of references.

In the introduction, the relevance of the topic is substantiated, the purpose, tasks, and research methods are defined.

In the first section, theoretical aspects of online job search services are considered, existing solutions, their features and drawbacks are analyzed.

The second section is devoted to reviewing the tools for implementing the future service, choosing technologies, developing the database and the user interface.

In the third section, the functions of the online application are described, the main components of the system are implemented, testing is carried out, and the performance and security of the developed service are evaluated.

In the conclusions, the results of the work are summarized, the effectiveness of the proposed solution is determined, and its development prospects are outlined.

The overall scope of the work is 94 pages of typescript. The thesis consists of an introduction, three chapters, general conclusions, 27 references, and 1 appendic. The work includes 14 tables and 66 figures.

Keywords: online service, job search, vacancy, resume, web application, client-server architecture, database, user interface.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API	– Application Programming Interface
CRUD	– Create, Read, Update, Delete
DB	– Database
DTO	– Data Transfer Object
ER	– Entity-Relationship
GORM	– Go Object Relational Mapping
HTTP	– Hypertext Transfer Protocol
IT	– Information Technology
JSON	– JavaScript Object Notation
JWT	– JSON Web Token
ORM	– Object-Relational Mapping
REST API	– Representational State Transfer
SPA	– Single Page Application
SQL	– Structured Query Language
UI	– User Interface
UUID	– Universally Unique Identifier
UX	– User Experience

ВСТУП

Онлайн-сервіси – це програмні платформи або додатки, які надають користувачам певні послуги через мережу Інтернет. Вони дозволяють взаємодіяти з інформаційними системами дистанційно, без потреби встановлення локального програмного забезпечення. Доступ до таких сервісів здійснюється за допомогою веббраузера або мобільного застосунку. У сучасних умовах цифровізації та воєнного часу особливої важливості набувають інформаційні системи, що забезпечують підтримку громадян у питаннях працевлаштування. Серед широкого спектра інтернет-платформ онлайн-сервіси пошуку роботи займають провідне місце, оскільки вони дозволяють ефективно взаємодіяти роботодавцям і шукачам роботи. На просторах Інтернету пропонують широкий вибір різноманітних сайтів, наприклад dcz.gov.ua [1], work.ua [2], robota.ua [3], однак багато сервісів в Інтернеті мають перевантажений інтерфейс або не надають достатньої кількості безплатного функціоналу.

Особливої актуальності зараз набувають спеціалізовані інформаційні системи, що спрямовані на підтримку фахівців із програмування та суміжних галузей у питаннях працевлаштування. Зокрема, інженери програмного забезпечення, frontend/backend-розробники, DevOps-фахівці, тестувальники, data-аналітики та інші IT-спеціалісти мають потребу в релевантних, сучасних і технічно зручних онлайн-інструментах для пошуку роботи, що відповідає їх очікуванням та рівню цифрової грамотності.

Актуальність даної роботи обумовлена тим, що в умовах воєнного стану в Україні багато громадян втратили роботу або були змушені змінити місце проживання. Це призвело до зростання рівня безробіття, необхідності переорієнтації на нові професії, а також потреби у швидкому пошуку джерел доходу. Онлайн-сервіси можуть виступати ефективним інструментом для розв'язання цієї проблеми, забезпечуючи доступ до вакансій та контактів із роботодавцями.

Мета роботи – розробка сучасного онлайн-сервісу пошуку роботи, який забезпечить зручний та швидкий доступ до вакансій, ефективну фільтрацію

пропозицій, можливість створення резюме та подання заявок, а також використання елементів обробки даних для персоналізації результатів пошуку.

Для досягнення цієї мети необхідно розв’язати такі завдання:

– проаналізувати предметну галузь і сучасні онлайн-рішення;
– сформулювати вимоги до системи;
– вибрати оптимальні технології для реалізації вебдодатку;
– спроєктувати архітектуру інформаційної системи;
– реалізувати основні функціональні модулі;
– протестувати роботу системи;
– оцінити результати реалізації.

Об’єктом роботи є процес розробки онлайн-сервісу пошуку роботи.

Предмет роботи – програмні інструменти та технології для розробки онлайн-сервісу для пошуку роботи.

У роботі використано технології веброзробки React (frontend), Go (backend) PostgreSQL (база даних), а також REST API для взаємодії між компонентами. Для реалізації фільтрації та пошуку даних застосовано базові методи обробки інформації.

1 АНАЛІЗ СУЧАСНИХ ОНЛАЙН-СЕРВІСІВ ПОШУКУ РОБОТИ

1.1 Дослідження використання онлайн-сервісів з пошуку роботи

Онлайн-сервіси сьогодні є надзвичайно популярними й охоплюють майже всі сфери людської діяльності. Вони стали невід'ємною частиною процесу працевлаштування, особливо у галузях, де цифрова присутність є нормою, зокрема у сфері ІТ. На сьогодні практично кожна компанія, магазин, навчальний, спортивний або культурний центр має свій вебдодаток або вебсайт для представлення себе в Інтернеті. Так само існує багато онлайн-сервісів пошуку роботи, які мають різноманітний інтерфейс з різним дизайном, логікою взаємодії та додатковими можливостями, але мають схожий функціонал і виконують функцію посередника між шукачами роботи та роботодавцями. Вони виникли у відповідь на потребу спростити процес працевлаштування як для роботодавців, так і для шукачів. Перші такі платформи з’явилися ще у 1990-х роках, але істотного розвитку набули після широкого поширення Інтернету у 2000-х. Вони дозволили автоматизувати процес подання резюме, відгуку на вакансії, зменшити використання паперових документів і прискорити прийняття рішень про наймання співробітників. Згідно з даними ВОНА хаб, у 2024 році кількість нових вакансій в Україні значно зросла, перевищивши показники 2021 року, що свідчить про активне відновлення ринку праці після кризових періодів [13].

Сьогодні більшість онлайн-сервісів мають адаптивні версії сайтів або мобільні додатки, що дозволяє користувачам шукати роботу будь-де і будь-коли. Це особливо актуально для молоді та спеціалістів у сфері ІТ, які активно користуються смартфонами. Мобільний доступ дозволяє швидко реагувати на нові вакансії та переглядати повідомлення. Онлайн-платформи, такі як Work.ua, Rabota.ua, Jooble [14], LinkedIn [4], Freelancehunt [15], Dou.ua [16] та HeadHunter [17], пропонують широкий спектр вакансій та інструментів для пошуку роботи. Ці сервіси дозволяють користувачам створювати резюме, фільтрувати вакансії за різними критеріями та безпосередньо зв'язуватися з роботодавцями.

У наш час відбувається трансформація ринку зайнятості, а саме його структура та режими трудової активності внаслідок світових глобалізаційних процесів, відбувається нормалізація та розповсюдження дистанційної діяльності на основі інтернет-мереж та цифрових платформ. Удосконалення законодавчої бази цих форм зайнятості мінімізує втрати робочих місць та гарантує заробіток. Масштабні руйнування промислових та інших підприємств, транспортної й іншої інфраструктури в країні, згортання малого та середнього бізнесу, відтік кваліфікованої робочої сили за кордон, зростання кількості внутрішньо переміщених осіб тощо завдали відчутного удару по ринку зайнятості, спричинивши дисбаланс між попитом та пропозицією робочих місць, падіння рівня заробітків, поглиблення зубожіння серед найуразливіших верств населення під час війни. Після припинення бойових дій та встановлення стійкого безпечного середовища в Україні постане питання необхідності відбудови зруйнованої економіки, і питання зайнятості буде одним із нагальних та пріоритетних [5].

Онлайн-сервіси пошуку роботи значно спрощують процес працевлаштування, активно впливають на розвиток цифрової економіки та адаптацію населення до нових умов життя і роботи в періоди нестабільності, по типу війни, пандемії чи кризи, або пост період цих глобальних змін. Сучасні технології також сприяють розвитку нових форм працевлаштування. Зокрема, фриланс стає все більш популярним серед молоді віком 18-35 років, особливо в галузях дизайну, програмування та роботи з текстами.

Проте, незважаючи на переваги, деякі онлайн-сервіси мають недоліки, такі як перевантаженість інтерфейсу, наприклад, різними новинами, функціями, статтями та рекламою, що може ускладнювати навігацію та знижувати ефективність пошуку. Наприклад work.ua, linkedin, robota.ua, а також сайт Державної служби зайнятості. Це може ускладнювати швидкий доступ до основного контенту – вакансій і резюме або ж сильно відвертати увагу.

Основна функціональність сервісів пошуку роботи на просторах Інтернету зазвичай включає в себе:

– створення облікового запису, коли користувач (робітник або роботодавець) реєструється на платформі та вводить свої дані;

– створення резюме та профілю користувача у вигляді анкети, що містить досвід роботи, освіту, навички та контакти;

– публікація вакансій від роботодавців. Вони створюють оголошення про відкриті вакансії, вказуючи умови роботи, обов’язки, вимоги, зарплату та локацію;

– пошук та фільтрація вакансій за ключовими словами, містом, типом зайнятості та категорією;

– чат з роботодавцем або шукачем роботи прямо на сайті для взаємозв’язку;

– шукачі роботи мають можливість подавати заявки на вакансії безпосередньо на платформі або можуть продивитись контакти роботодавця прямо на сайті.

Онлайн-сервіси забезпечують прозорість, оперативність і зручність процесу працевлаштування, даючи змогу максимально швидко адаптуватися до нових обставин.

Онлайн-сервіси орієнтовані на три основні категорії користувачів:

– шукачі роботи – фізичні особи, які шукають роботу. Вони створюють резюме, налаштовують фільтри, подають заявки, отримують сповіщення про нові вакансії;

– роботодавці – компанії, юридичні або фізичні особи, які публікують вакансії та шукають кандидатів. Вони мають доступ до резюме кандидатів, які шукають роботу, і за допомогою цього резюме можуть дізнатись досвід роботи, отриману освіту, знання мов або навички кожного хто міг би їх зацікавити на певну посаду у своїй компанії;

– адміністратори системи або розробники – забезпечують технічну підтримку, контроль за якістю контенту, модерацію вакансій, захист від шахрайства, покращення сервісу.

Сучасні онлайн-сервіси пошуку роботи є важливими елементами цифрового середовища, які дозволяють швидко та ефективно розв'язувати питання

працевлаштування. Їхня роль особливо важлива в періоди кризи, коли зростає потреба у доступних, надійних та гнучких способах взаємодії на ринку праці.

1.2 Огляд та аналіз аналогічних сервісів

Проаналізуємо декілька онлайн-сервісів пошуку роботи, які вже існують на ринку України та за її межами для глибшого розуміння функціональності та особливостей сучасних онлайн-сервісів пошуку роботи. Аналіз функціональних можливостей таких платформ дає змогу сформувати вимоги та побажання до власного проєкту виявити сильні та слабкі сторони конкурентів, а також врахувати користувацький досвід під час розробки. Першим із таких сайтів розглянемо work.ua, представлений на рис. 1.1.

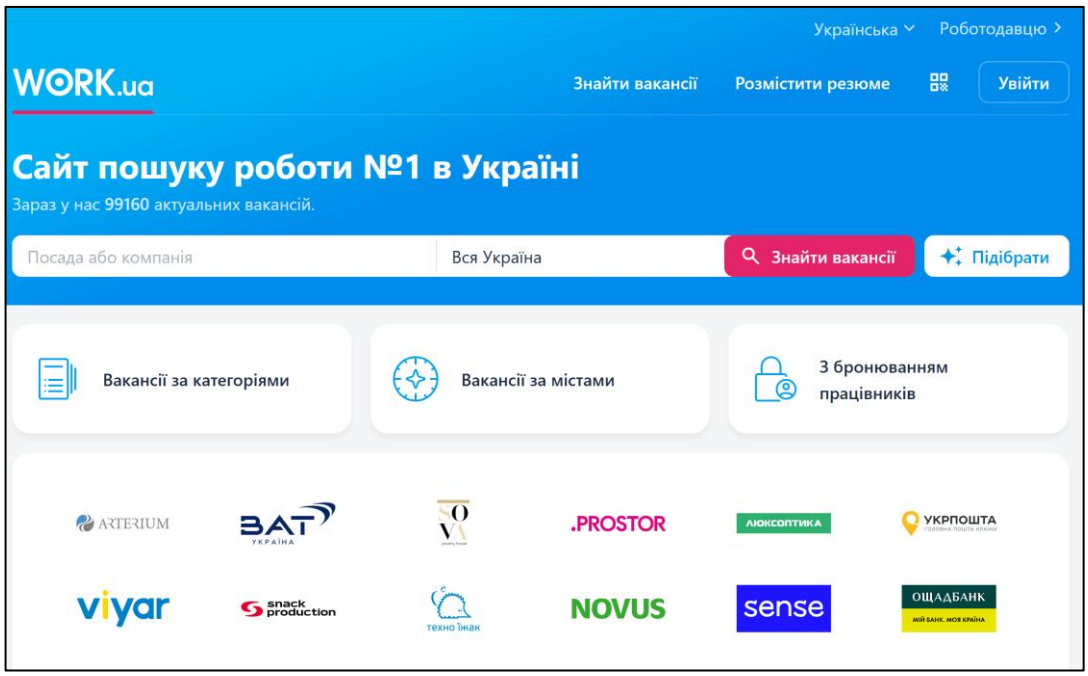


Рисунок 1.1 – Сайт пошуку роботи work.ua

Work.ua – одна з найпопулярніших та найбільших платформ з пошуку роботи в Україні, що пропонує понад 96 000 актуальних вакансій. Сервіс дозволяє користувачам створювати резюме, шукати роботу за різними категоріями та критеріями, підписуватись на розсилки з новими пропозиціями. Також для

роботодавців є можливість знайти співробітників на певні посади з можливістю фільтрації по містах.

Також схожим за використанням є сайт Rabota.ua, який зображено на рис. 1.2.

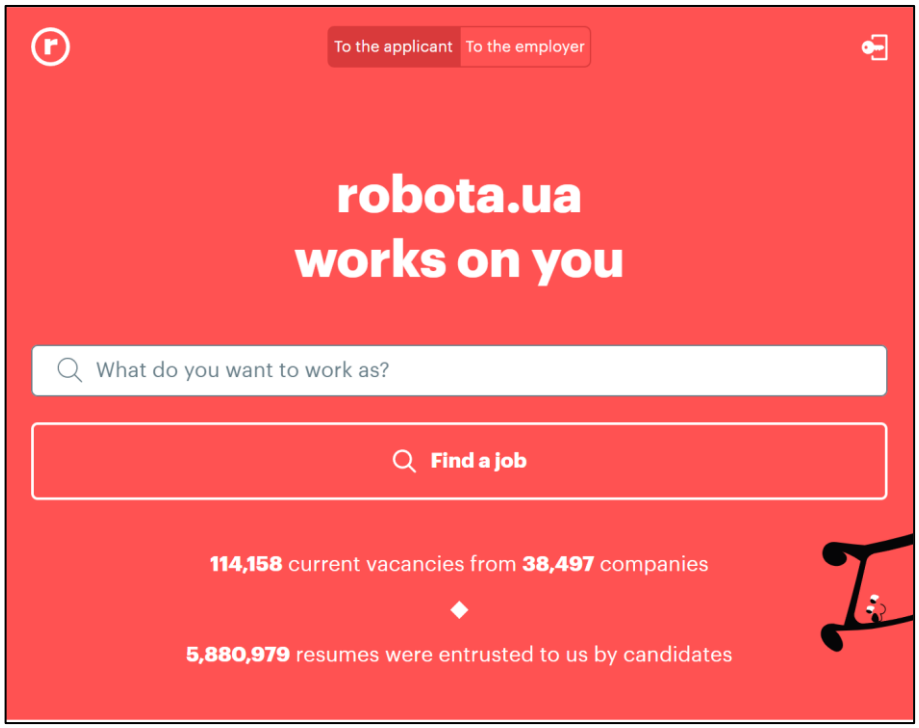


Рисунок 1.2 – Сайт пошуку роботи robota.ua

Цей сайт є платформою з широким вибором вакансій у різних галузях з різним досвідом роботи. Користувачі можуть створювати резюме, отримувати інформацію про роботодавців та налаштовувати фільтри для пошуку. Дуже зручним на сайті є поділ користувачів на дві категорії, а саме роботодавці та шукачі роботи.

Переваги:

– велика база вакансій;
– чіткий поділ сайту для роботодавців та шукачів роботи;
– багато фільтрів для кращого підбору вакансій.

Недоліки:

– має перевантажену сторінку.

Подібним є сайт Державної служби зайнятості України, представлений на рис. 1.3.

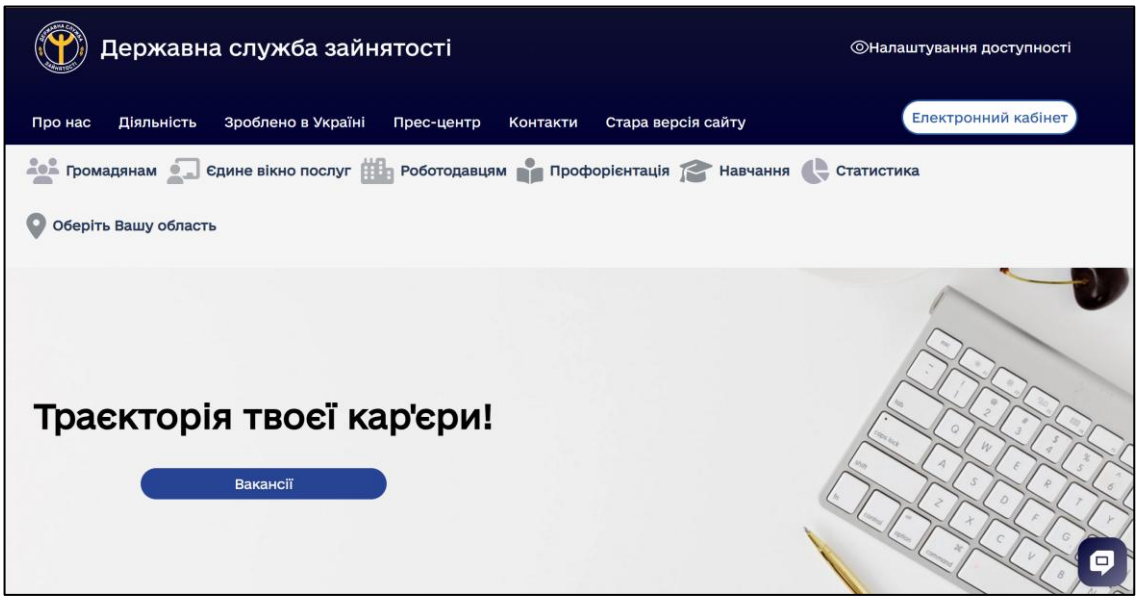


Рисунок 1.3 – Сайт пошуку роботи Державної служби зайнятості

Цей онлайн-сервіс є офіційною платформою Державної служби зайнятості України, яка надає громадянам доступ до актуальних вакансій, інформації про програми підтримки, навчання та інші послуги, спрямовані на сприяння працевлаштуванню.

Переваги:

– є державним сайтом з перевіркою вакансій;
– швидкий доступ до державних послуг;
– надає користувачеві можливість анонімності та конфіденційності.

Недоліки:

– немає прямої подачі заявки;
– інтерфейс може бути менш інтуїтивно зрозумілим;
– обмежена кількість вакансій.

Наступним переглянемо міжнародний сайт пошуку роботи LinkedIn, який представлено на рис. 1.4.

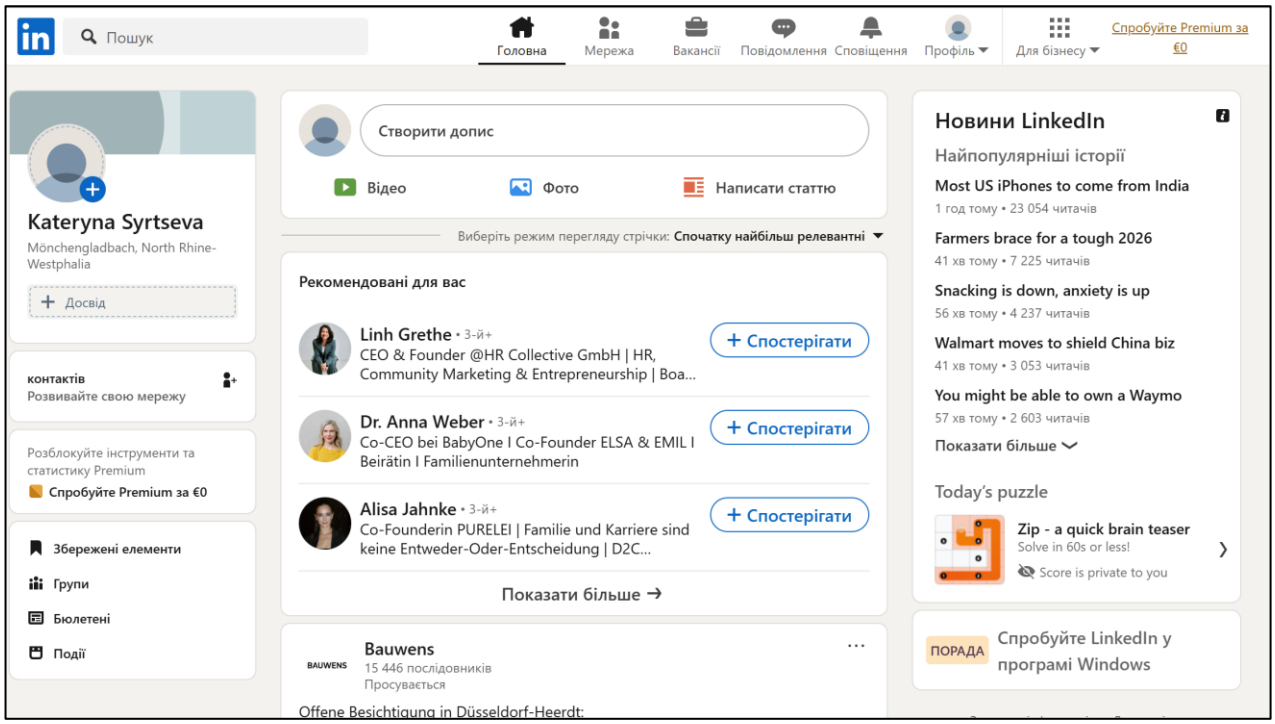


Рисунок 1.4 – Сайт пошуку роботи LinkedIn

LinkedIn – міжнародна платформа для професійного нетворкінгу та пошуку роботи. Вона дозволяє створити детальний профіль, переглядати пости інших користувачів, спостерігати за іншими, підтримувати професійні контакти та відгукуватися на вакансії по всьому світу. Зазвичай цей застосунок використовують професійні фахівці.

Переваги:

– міжнародне охоплення;
– дуже висока популярність;
– підходить для професіоналів та фахівці.

Недоліки:

– складний інтерфейс для новачків;
– основна активність англійською мовою;
– багато платного контенту.

Виконаємо загальний аналіз цих застосунків. У табл. 1.1 представлено порівняння обраних сайтів.

Таблиця 1.1 – Аналіз розглянутих сайтів пошуку роботи

Критерії	**work.ua**	**Державна служба зайнятості**	**LinkedIn**
Неактуальна інформація	–	+	+
Можливість зв'язку з роботодавцем на сайті / подача заявки	+	–	+
Зручність використання	+	–	–
Візуально перевантажена сторінка	+	–	+
Можливість використання фільтрів та пошуку	+	+	+

Проаналізувавши українські та іноземні онлайн-сервіси пошуку роботи зі схожою тематикою, було вирішено розробити онлайн-сервіс який не мав би недоліків, представлених у таблиці 1.1. Ця розробка матиме переваги цих сайтів, а також зручний та унікальний інтерфейс.

1.3 Постановка задачі та формулювання технічних вимог до системи

У сучасному світі пошук роботи став невід’ємною частиною життя для більшості людей. З розвитком цифрових технологій традиційні способи пошуку працевлаштування, такі як перегляд оголошень у газетах або подання заявок особисто, були витіснені онлайн-платформами, що дають змогу швидко знаходити вакансії, подавати резюме та спілкуватися з роботодавцями, або взагалі працювати повністю онлайн [6].

Однак, більшість наявних сервісів мають низку недоліків: перевантажений інтерфейс, недостатню швидкість роботи або відсутність гнучких фільтрів. Тому виникає задача розробити новий онлайн-сервіс пошуку роботи, який буде орієнтований на простоту використання, високу швидкодію, безпеку даних та адаптивність до різних категорій користувачів.

Мета розробки системи – створення вебсервісу, який забезпечить ефективний, безпечний та зручний процес пошуку роботи та підбору персоналу у сфері IT.

Основні задачі, які необхідно реалізувати у межах розробки системи:

– забезпечити реєстрацію та автентифікацію користувачів (шукачів та роботодавців);

– надати можливість створення та редагування резюме шукачами роботи;

– надати можливість шукачами роботи завантажувати своє резюме у вигляді pdf файлу або word документу;

– забезпечити створення, редагування та управління вакансіями роботодавцями;

– реалізувати ефективний пошук та фільтрацію вакансій за заданими критеріями;

– реалізувати можливість подання відгуку на вакансію;

– забезпечити можливість підтримання діалогу через чат в середині застосунку;

– забезпечити захист персональних даних користувачів відповідно до чинних вимог законодавства;

– реалізувати особистий кабінет.

Система буде реалізована за клієнт-серверною архітектурою:

– Frontend – розробка односторінкового застосунку (SPA) з використанням React;

– Backend – реалізується на Go;

– база даних – PostgreSQL, із чітко продуманою структурою таблиць;

– обмін даними – використання протоколу HTTP для обміну даними між клієнтом та сервером.

У результаті постановки задачі сформульовано основні цілі та вимоги до онлайн-сервісу пошуку роботи. Правильне визначення технічних вимог дозволить уникнути помилок у процесі розробки та забезпечить відповідність кінцевого продукту потребам користувачів і вимогам сучасного ринку праці.

Висновки до розділу 1

У першому розділі було проведено дослідження сучасних онлайн-сервісів пошуку роботи, проаналізовано особливості їхньої роботи, основні можливості, переваги та недоліки. Встановлено, що такі платформи вже давно відіграють важливу роль у забезпеченні доступу до ринку праці, особливо у складних соціально-економічних умовах – під час пандемії, війни або масових внутрішніх переміщень населення. Водночас багато з них мають перевантажений інтерфейс, обмежену гнучкість у налаштуванні пошуку або низьку зручність використання.

Проведений аналіз аналогічних сервісів, а саме найбільш популярних платформ, таких як: Work.ua, LinkedIn та портал Державної служби зайнятості, дозволив виокремити ключові функції, які мають бути реалізовані в рамках проєкту, а також уникнути типових недоліків. На основі проведеного аналізу було сформульовано технічне завдання для створення нового онлайн-сервісу пошуку роботи, та визначено перелік основних вимог до майбутнього онлайн-сервісу.

Було визначено, що система повинна включати:

– просту реєстрацію та автентифікацію користувачів;
– підтримку створення резюме і вакансій;
– розширений фільтр пошуку;
– особисті кабінети;
– чат між користувачами;
– захист персональних даних.

У результаті було прийнято рішення реалізувати систему з клієнт-серверною архітектурою, з поділом логіки на frontend і backend частини, де:

– інтерфейс користувача розроблятиметься на React, що дозволяє створити динамічний SPA-застосунок зі зручною навігацією та високою швидкістю роботи;

– серверна частина – на Go, яка забезпечить надійність, простоту підтримки та високу продуктивність системи;

– для зберігання даних використовуватиметься PostgreSQL, що дозволяє ефективно опрацьовувати структуровані дані та забезпечує стабільну роботу сервісу навіть при високих навантаженнях.

Запропоноване технічне рішення націлене на досягнення балансу між функціональністю, зручністю та масштабованістю та має забезпечити високу швидкодію, простоту у використанні, безпеку та адаптивність, що дозволить створити ефективний інструмент для пошуку роботи та підбору персоналу.

2 МОДЕЛІ ТА МЕТОДИ ДЛЯ РОЗРОБКИ ОНЛАЙН-СЕРВІСУ

2.1 Засоби реалізації онлайн-сервісу

Проєкт онлайн-сервісу пошуку роботи вимагає комплексного підходу до вибору інформаційних технологій та моделей взаємодії між користувачами та системою. На цьому етапі реалізації важливо забезпечити оптимальний розподіл функціоналу між клієнтською та серверною частинами, побудувати стабільну архітектуру системи, обрати відповідні технології для забезпечення продуктивності, масштабованості та безпеки.

Система будується за принципом клієнт-серверної архітектури, яка передбачає наявність окремого інтерфейсу користувача (frontend), серверної частини (backend) та бази даних.

Такий підхід дозволяє організувати ефективну взаємодію між компонентами системи, розділити відповідальність між модулями та реалізувати SPA-застосунок (Single Page Application), що підвищує зручність для користувача.

Технологічний стек проєкту включає:

– React – для створення клієнтської частини. Завдяки цьому фреймворку реалізується односторінковий інтерфейс із динамічною навігацією, інтерактивними формами, ефективною фільтрацією вакансій і резюме без перезавантаження сторінки;

– Go (Golang) – для розробки серверної частини. Мова дозволяє створити продуктивний монолітний вебдодаток із вбудованою підтримкою обробки запитів, авторизації, збереження та обробки даних;

– PostgreSQL – реляційна система управління базами даних, яка забезпечує надійне збереження структурованої інформації, включно з акаунтами користувачів, вакансіями, резюме та іншими об’єктами.

Для реалізації онлайн-сервісу пошуку роботи обрано монолітну архітектуру, що передбачає побудову єдиної серверної програми, яка виконує всі основні функції системи: від автентифікації до обробки запитів користувачів та взаємодії з базою даних.

Монолітна архітектура – це підхід до розробки програмного забезпечення, за якого всі компоненти програми (логіка, маршрути, API, обробка даних, доступ до бази) об'єднані в єдину програму, яка запускається як один виконуваний процес.

Переваги монолітного підходу:

– простота розгортання тому, що лише один файл або контейнер потрібно встановити на сервері;

– єдина кодова база спрощує розробку, тестування і супровід;

– швидкий запуск проєкту тому, що не потрібно витрачати час на налаштування окремих мікросервісів або їхню взаємодію;

– зручність для невеликих або середніх проєктів, які ще не потребують масштабування на рівні мікросервісів.

У межах цієї системи всі запити користувачів (реєстрація, створення вакансій, пошук, відгуки) обробляються централізовано в одному монолітному серверному застосунку.

2.2 Обґрунтування вибору інструментів та технологій розробки системи

Для створення ефективного, надійного та зручного онлайн-сервісу пошуку роботи було обрано сучасні технології, які відповідають вимогам до продуктивності, безпеки, масштабованості та швидкості розробки. У цьому підрозділі наводиться обґрунтування вибору основних інструментів, які використовуються для реалізації клієнтської та серверної частин системи, а також бази даних. Кожне з рішень підібрано з урахуванням специфіки задачі, досвіду використання в реальних проєктах і перспектив подальшого розширення функціональності.

Для реалізації клієнтської частини обрано React – сучасний JavaScript-фреймворк, що дозволяє створювати ефективні односторінкові застосунки (SPA). React забезпечує компонентну структуру, повторне використання коду, високу швидкість взаємодії з користувачем, а також просту інтеграцію з REST API. Завдяки екосистемі бібліотек (React Router, Axios, Formik тощо), React дозволяє реалізувати адаптивний інтерфейс з мінімальним навантаженням на сервер.

Серверна частина реалізована на Go (Golang) – компільованій мові програмування, яка поєднує в собі високу продуктивність, вбудовану підтримку паралельного програмування (goroutines), зручність для розробки вебзастосунків і масштабованість. Go забезпечує швидку обробку HTTP-запитів, надійність і простоту розгортання. Його використання дозволяє уникнути надмірної складності в коді та спрощує підтримку проєкту в довгостроковій перспективі.

У якості бази даних обрано PostgreSQL – потужну реляційну СУБД з відкритим кодом. Вона забезпечує ACID-гарантії, підтримує складні SQL-запити, має розвинуту систему індексації, транзакції, збережені процедури, а також можливість роботи з JSONB-полями. PostgreSQL ідеально підходить для зберігання інформації про користувачів, вакансії, резюме та зв’язки між ними.

Взаємодія між frontend і backend частинами здійснюється через REST API, що забезпечує гнучкий, стандартизований і масштабований підхід до обміну даними. Такий формат дає змогу легко підтримувати взаємодію між компонентами, а також інтегрувати зовнішні сервіси в майбутньому (наприклад, платіжні системи чи аналітику).

2.2.1 React для клієнтської частини

У сучасних вебзастосунках клієнтська частина відіграє ключову роль, оскільки саме через неї відбувається основна взаємодія користувача із системою. Для реалізації клієнтської частини онлайн-сервісу пошуку роботи було обрано бібліотеку React, яка є однією з найпопулярніших технологій для створення односторінкових застосунків (SPA) [10].

React – це JavaScript-бібліотека з відкритим кодом, розроблена компанією Meta (Facebook) для створення інтерфейсів користувача. Вона забезпечує високу продуктивність завдяки віртуальному DOM, підтримці компонентного підходу та реактивному оновленню UI.

Переваги використання React у даному проєкті:

– SPA-архітектура, яка дозволяє користувачу взаємодіяти з сайтом без повного перезавантаження сторінок. Це забезпечує швидкість і зручність роботи;

– компонентний підхід, коли кожна частина інтерфейсу реалізується як окремий компонент (наприклад, кнопки, форми, картки вакансій), що спрощує повторне використання коду, тестування та масштабування;

– JSX: синтаксис, який дозволяє комбінувати HTML та JavaScript, підвищуючи читабельність і швидкість розробки;

– підтримка реактивності, яка надає зміни стану, які автоматично відображаються в інтерфейсі без необхідності ручного оновлення DOM;

– React легко працює з REST API через бібліотеки axios або fetch, що забезпечує динамічну взаємодію з backend;

– легко реалізується адаптивна верстка для мобільних і десктопних пристроїв через CSS-модулі або бібліотеки типу TailwindCSS або Bootstrap;

– присутня велика кількість сторонніх бібліотек, таких як React Router для маршрутизації, Redux або Context API для управління станом, Formik для обробки форм тощо;

– React не нав'язує архітектурних обмежень, тому його легко інтегрувати у будь-яку структуру проєкту.

Інтерфейс системи реалізовано у вигляді SPA з такими основними сторінками та компонентами як:

– сторінка реєстрації та входу (з JWT-аутентифікацією);

– головна сторінка;

– сторінка з відображенням та пошуком вакансій;

– сторінка перегляду вакансії з повним описом пропозиції;

– форма створення та редагування резюме;

– кабінет користувача з можливістю змінити дані;

– чат між роботодавцем та шукачем роботи;

– форма подачі заявки/відгуку на вакансію.

Кожна сторінка реалізована як окремий React-компонент, що дозволяє динамічно оновлювати інтерфейс у відповідь на дії користувача.

Виконаємо тепер порівняння, яке представлено у табл. 2.1, React з Vue.js та Angular, щоб зрозуміти, яка технологія краще.

Таблиця 2.1 – Порівняльна таблиця React з Vue.js та Angular

Критерій	**React**	**Vue.js**	**Angular**
Простота входу	середня	проста	складна
SPA підтримка	так	так	так
Документація	чітка	зрозуміла	об’ємна
Швидкість розробки	висока	висока	помірна
Гнучкість	висока	висока	середня
Вага застосунку	невелика	невелика	велика

React надає оптимальний баланс між продуктивністю, гнучкістю та простотою підтримки, тому є ефективним вибором для реалізації онлайн-сервісу пошуку роботи, орієнтованого на комфортну взаємодію з користувачем та масштабовану структуру застосунку.

2.2.2 Go для backend частини

Серверна частина або ж backend частина онлайн-сервісу пошуку роботи буде повністю реалізована за допомогою мови програмування Go (або Golang). Обраний підхід базується на використанні монолітної архітектури, в якій вся логіка – автентифікація, обробка запитів, управління вакансіями, резюме, профілями користувачів та обробка взаємодії з базою даних – реалізується централізовано в одному застосунку.

Мова програмування Go (Golang) поєднує простоту синтаксису, продуктивність, вбудовану підтримку конкурентності та зручність розробки вебзастосунків. Завдяки цим якостям Go здобула широку популярність у сфері розробки серверного програмного забезпечення, хмарних рішень, мікросервісів та веб API.

Завдяки Go можливо ефективно реалізувати логіку без необхідності сторонніх фреймворків – стандартна бібліотека мови покриває більшість задач. Це дозволяє зменшити залежність від стороннього ПЗ, підвищити безпеку та

стабільність коду. Крім того, Go чудово інтегрується з системами логування, аналітики, а також підтримує Docker і Kubernetes «з коробки», що важливо для подальшого масштабування.

Go – це компільована, статично типізована мова з високою швидкістю виконання та вбудованим сміттєзбирачем (garbage collector). Вона спеціально створювалась для розробки високонавантажених мережевих застосунків і серверів, і вже використовується такими компаніями, як Google, Uber, Dropbox, Twitch, SoundCloud та іншими [12].

Go вирізняється тим, що надає всі необхідні засоби для створення сучасного вебсервісу без потреби у використанні сторонніх фреймворків. Стандартна бібліотека включає підтримку HTTP-серверів, обробку JSON, криптографію, логування, а також засоби для конкурентного програмування.

Це дозволяє:

– зменшити кількість залежностей;
– підвищити безпеку коду;
– пришвидшити компіляцію;
– спростити розгортання (виконуваний файл містить усі залежності);
– забезпечити кращу масштабованість.

Go також відзначається гарною підтримкою REST API. За допомогою стандартного пакета net/http можна реалізовувати маршрути, middleware, обробку запитів та відповідей. У межах проєкту REST API дозволить frontend-частині обмінюватися даними з backend за допомогою стандартних HTTP-запитів (GET, POST, PUT, DELETE).

Можна виділити наступні основні переваги Go в контексті створення даного онлайн-сервісу пошуку роботи:

– Go-код компілюється безпосередньо у машинний код, що забезпечує високу швидкість виконання порівняно з інтерпретованими мовами (наприклад, Python або JavaScript);

– конкурентність реалізована за допомогою горутин (goroutines) і каналів (channels), що дозволяє обробляти тисячі паралельних HTTP-запитів без перевантаження пам’яті;

– Go навмисно позбавлений зайвої складності: у ньому немає класичного ООП, перевантаження методів чи винятків. Завдяки цьому код зрозумілий навіть через багато місяців після написання;

– для розробки вебзастосунків не потрібно додаткових бібліотек: усе, що потрібно для створення HTTP-сервера, JSON-обробки, роботи з файлами чи криптографії – уже є в стандарті;

– Go-компілятор збирає весь код і залежності в один виконуваний файл, який легко деплоїти на будь-який сервер без додаткових конфігурацій;

– Go дозволяє уникнути типових помилок роботи з пам’яттю, які властиві C/C++, наприклад, переповнення буфера, некоректне звільнення пам’яті тощо.

У проєкті Go використовується як єдиний серверний рушій для обробки всіх запитів. Система розроблена як монолітний вебзастосунок, який об'єднує наступні компоненти:

– автентифікацію користувачів (реєстрація, вхід, перевірка JWT токенів);

– створення і редагування профілів;

– завантаження резюме у вигляді файлу під час відгуку на вакансію;

– створення чату між шукачем роботи та роботодавцем;

– модуль фільтрації та публікація вакансій;

– інтеграція з PostgreSQL;

– модуль для обробки HTTP-запитів;

– CRUD-операції з вакансіями та резюме.

Go був обраний саме тому, що дозволяє створити швидку, надійну та масштабовану backend-систему без зайвих складнощів.

Виконаємо тепер порівняння мов програмування для розробки онлайн-сервісу, яке представлено у табл. 2.2.

Таблиця 2.2 – Порівняльна таблиця мов програмування

Критерій	**Go**	**Node.js**	**Python**	**Java**
Продуктивність	висока	середня	низька	середня
Простота розгортання	+	+	–	–
Підтримка конкурентності	+	обмежена	–	+
Розмір виконуваного файлу	+	–	–	–
Легка інтеграція з PostgreSQL	+	+	+	+

Go забезпечує чудовий баланс між потужністю і простотою розробки – саме те, що потрібно для невеликих проєктів типу онлайн-сервісів.

2.2.3 PostgreSQL як основна СУБД

У якості основної системи керування базами даних у проєкті буде використовуватись PostgreSQL – одна з найпотужніших та гнучких реляційних СКБД з відкритим вихідним кодом. Вона підтримує широкі можливості для збереження, обробки та пошуку даних, а також має високу надійність, масштабованість і стабільність роботи навіть при значному навантаженні [11].

PostgreSQL активно використовується у корпоративних проєктах, в урядових інформаційних системах, наукових базах даних і великих SaaS-продуктах. Її функціональність значно ширша, ніж у багатьох інших безплатних СУБД.

Серед особливостей PostgreSQL підтримує реплікацію (master-slave та logical replication), що забезпечує підвищену надійність і дає можливість масштабування при зростанні аудиторії. Система дозволяє реалізувати механізми резервного копіювання, а також оптимізувати запити завдяки EXPLAIN ANALYZE [18] для аналізу продуктивності.

Основні переваги PostgreSQL:

– дозволяє будувати чітку структуру даних з відношеннями між таблицями;

– завдяки ACID-гарантіям [19] забезпечується цілісність і узгодженість даних. Це важливо при одночасному редагуванні вакансій, оновленні профілю або обробці заявок;

– JSONB [20] для зберігання напівструктурованих даних. Наприклад, список навичок, сертифікатів або соцмереж з профілю користувача може зберігатись у вигляді JSONB – це дозволяє виконувати пошук всередині JSON та індексувати ці поля;

– гнучкі механізми індексації (GIN, GiST, B-tree), використання яких дозволяє пришвидшити пошук вакансій по ключових словах, фільтрах або регіонах;

– дозволяє розмежовувати права доступу для різних ролей (наприклад, користувач, адміністратор), шифрувати передані дані та вести логування підключень.

Виконаємо порівняння різних баз даних, яке представлено у табл. 2.3.

Таблиця 2.3 – Порівняльна таблиця БД

Харктеристика	**PostgreSQL**	**MySQL**	**SQLite**	**MongoDB**
Тип БД	Реляційна	Реляційна	Реляційна	Документна
Схема	Обов'язкова	Обов'язкова	Обов'язкова	Гнучка
Підтримка SQL	Повна	Часткова	Повна	Немає
ACID	Так	Так	Так	Так
Транзакції	Так	Так	Частково	Так
Швидкість читання	Висока	Дуже висока	Висока	Дуже висока
Швидкість запису	Висока	Дуже висока	Середня	Дуже висока
Індексація	Потужна	Добра	Обмежена	Потужна
Підтримка JSON	Так	Так	Обмежено	Основна
Ліцензія	Open Source	Open Source	Public Domain	Open Source

У рамках розробки онлайн-сервісу PostgreSQL буде використовуватись для:

– зберігання акаунтів користувачів (із хешованими паролями);

– створення резюме та вакансій з відповідними зв’язками через foreign keys;

– підтримання спілкування між роботодавцем та шукачем роботи;

– фільтрації вакансій за назвою та навичками.

Таким чином, PostgreSQL є ідеальним вибором для побудови надійної та масштабованої бази даних у монолітній архітектурі. Її використання дозволяє не лише ефективно зберігати структуровані дані, але й реалізувати розширені функції пошуку, сортування, зв’язків та безпеки, що критично важливо для інформаційних систем із великою кількістю активних користувачів.

Висновки до розділу 2

У другому розділі було детально розглянуто моделі, методи та інформаційні технології, використані для розробки онлайн-сервісу пошуку роботи. Основною архітектурною моделлю обрано монолітну архітектуру, яка дозволяє централізовано реалізувати всю серверну логіку в межах одного застосунку. Такий підхід є оптимальним для середніх за складністю проєктів, оскільки забезпечує простоту в реалізації, розгортанні та супроводі системи.

Для побудови серверної частини обрано мову програмування Go, що відзначається високою продуктивністю, підтримкою паралельних обчислень, простим синтаксисом і зручною роботою з HTTP-запитами та базами даних. Усі ключові функції системи, такі як авторизація, керування вакансіями, обробка резюме та пошук, реалізуються єдиним Go-застосунком.

Клієнтська частина реалізується з використанням React, що дозволяє створити швидкий і зручний односторінковий інтерфейс (SPA). React забезпечує реактивність, модульність і простоту розробки, а також дозволяє користувачам ефективно взаємодіяти із системою без постійного перезавантаження сторінок.

Для збереження інформації про користувачів, вакансії, резюме та інші об'єкти системи використовується реляційна СКБД PostgreSQL, яка має високу продуктивність, підтримку транзакцій, розширень і складних запитів.

Комунікація між frontend і backend частинами організована через REST API, що дозволяє підтримувати чисту, стандартизовану та ефективну взаємодію між клієнтською частиною та сервером.

Таким чином, у розділі було обґрунтовано вибір технологій і засобів, що забезпечують гнучкість, стабільність, масштабованість і безпеку системи, що відповідає сучасним вимогам.

3 ПРОЄКТУВАННЯ, ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ОНЛАЙН-СЕРВІСУ З ПОШУКУ РОБОТИ

3.1 Розробка моделі онлайн-сервісу

Онлайн-сервіс пошуку роботи є складним інформаційним продуктом, який охоплює низку взаємопов’язаних компонентів. Основна мета системи – забезпечення ефективної взаємодії між роботодавцями та шукачами роботи з використанням зручного інтерфейсу, сучасних фільтрів і безпечної передачі даних.

Система охоплює дві умовні основні ролі користувачів:

– шукачі роботи – фізичні особи, які створюють резюме, шукають вакансії, надсилають відгуки на пропозиції;

– роботодавці – компанії або фізичні особи, які мають можливість публікувати вакансії та переглядати резюме кандидатів.

Модель поведінки шукача роботи зображена на рис. 3.1.

Рисунок 3.1 – Поведінка шукача роботи на сайті

Модель поведінки роботодавця зображена на рис. 3.2.

Рисунок 3.2 – Поведінка роботодавця на сайті

Вхідні дані користувачів:

– ім’я;
– електронна пошта;
– пароль;
– місце проживання;
– тип користувача;
– рівень англійської мови;
– досвід роботи (у профілі);
– бажана зарплатня (у профілі);
– навички (у профілі);
– CV у вигляді PDF або Word форматі (при подачі відгуку на вакансію).

Якщо роботодавець буде створювати вакансію, він повинен ввести наступні вхідні дані:

– назва компанії;
– назва вакансії;
– опис;
– місце розташування;
– зарплатня;
– країна;
– необхідний рівень англійської;
– тип зайнятості.

Основні функціональні вимоги до системи:

– авторизація/реєстрація користувачів;
– створення резюме та вакансій та робота з ними;
– фільтрація результатів за заданими параметрами;
– реалізація особистого кабінету з панеллю управління профілем.

Виконаємо моделювання процесу роботи онлайн-сервісу пошуку роботи.

На рівні А0 весь процес розглядається, як функціональний блок з усіма керуючими та робочими об'єктами. Діаграма на рис. 3.3 відображає всю необхідну вхідну та вихідну інформацію, яка використовується при користуванні сервісом.

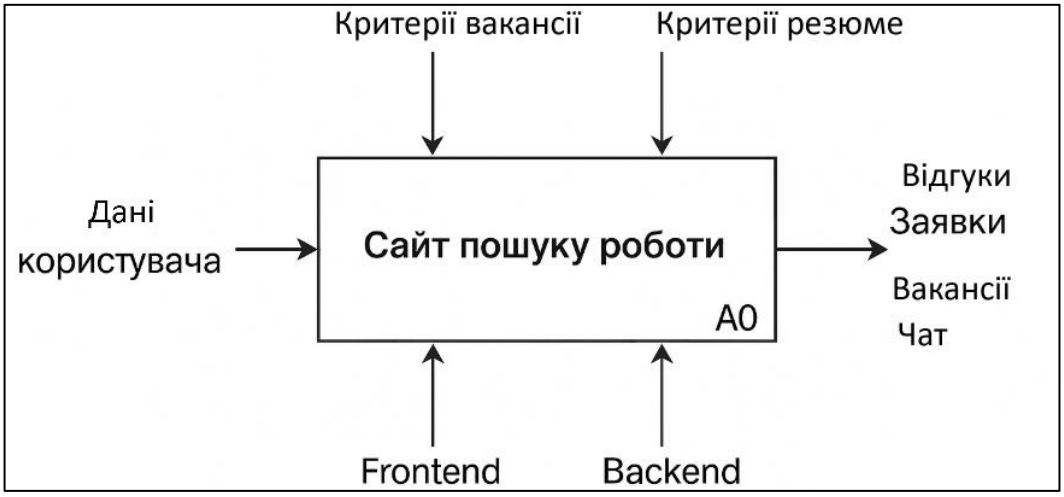


Рисунок 3.3 – Контекстна модель

Діаграма першого рівня, представлена на рис. 3.4, описує функцію обробки нульового рівня і розкладає функціональний блок А0 на набір взаємопов'язаних під функцій.

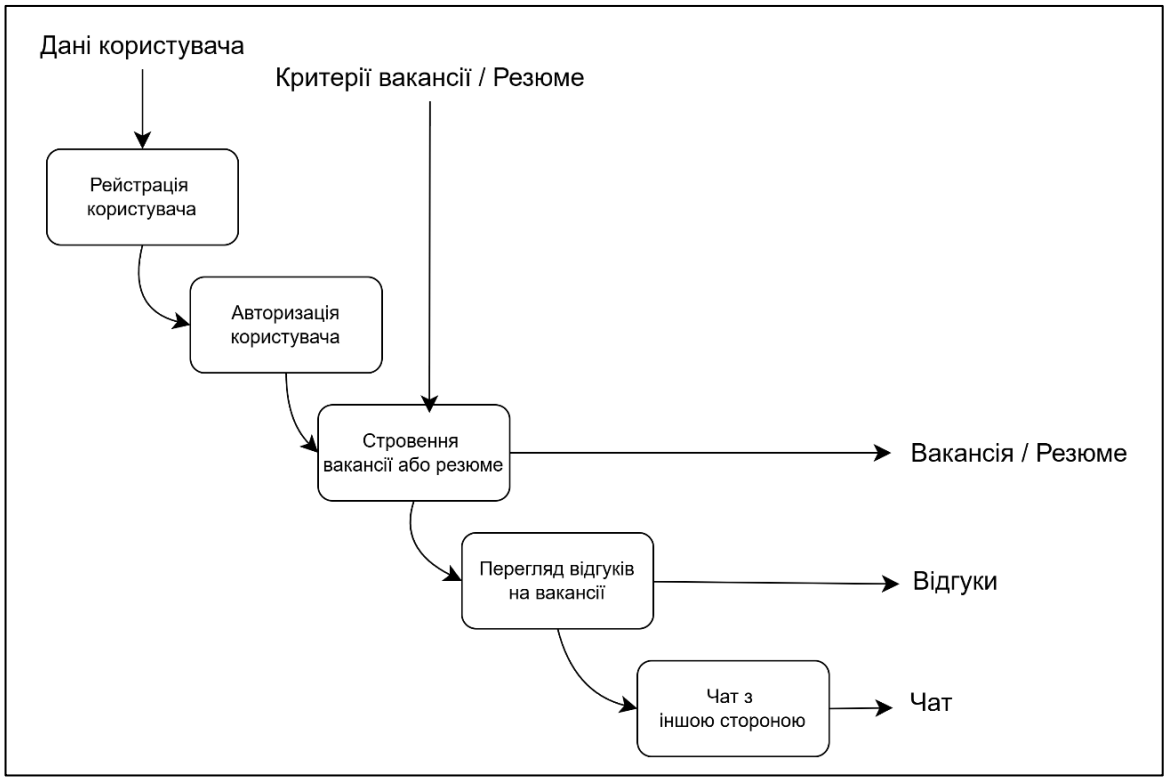


Рисунок 3.4 – Модель користувацької взаємодії

Проаналізувавши роботу вебдодатку було виділено, у табл. 3.1 категорії користувачів онлайн-сервісом та у табл. 3.2 представлено опис функціоналу сайту.

Таблиця 3.1 – Категорії користувачів та їх опис

Назва категорії	**Опис**
Шукач роботи	Людина, яка реєструється на сайті для пошуку роботи, подає відгуки на вакансії, створює резюме та налаштовує свій профіль.
Роботодавець	Користувач, який розміщує вакансії та переглядає відгуки.

Таблиця 3.2 – Опис функціоналу сайту

Функція	**Опис**
CRUD-операції з резюме	Створення, оновлення, видалення, перегляд резюме
CRUD-операції з вакансіями	Створення, оновлення, видалення, перегляд вакансії
Реєстрація через електрону пошту	Створення облікового запису
Автентифікація через електрону пошту	Вхід за email та паролем з JWT
Створення профілю	Додавання персональних даних
Пошук вакансій	За ключовими словами або фільтрами
Сторінка з опублікованими вакансіями	Сторінка для роботодавця з опублікованими вакансіями від його профілю
Обмін даними через JSON-формат	API взаємодія клієнта і сервера
Подання відгуків на вакансії	Кнопка "відгукнутись" на вакансію
Сторінка з відгуками (шукач роботи)	Сторінка з відправленими відгуками на вакансії
Сторінка з відгуками (роботодавець)	Сторінка з отриманими відгуками на опубліковані вакансії
Вихід з акаунту	Виконання виходу зі свого облікового запису
Завантаження файлу з резюме до відгуку	Завантаження пдф або word файлу під час відгуку на вакансії для прикріплення CV
Головна сторінка	Перехід на головну сторінку для зручного перегляду сайту та функціоналу
Сторінка налаштувань	Сторінка налаштувань для зміни кольорової теми онлайн-сервісу

Після створення моделі можна перейти до проєктування самої архітектури застосунку.

3.2 Проєктування архітектури інформаційної системи

Архітектура онлайн-сервісу розроблена за принципом клієнт-серверної взаємодії. Система складається з frontend, backend та бази даних. Frontend реалізований за допомогою React, що дозволяє створити SPA зі зручною навігацією. Backend реалізований на мові Go, який відповідає за авторизацію, обробку заявок, роботу з вакансіями. Для бази даних обрано PostgreSQL, що зберігає дані користувачів, резюме, вакансії та заявки. Обмін даними у цьому онлайн-сервісі відбувається через REST API. На рис. 3.5 зображено структуру проєкту.

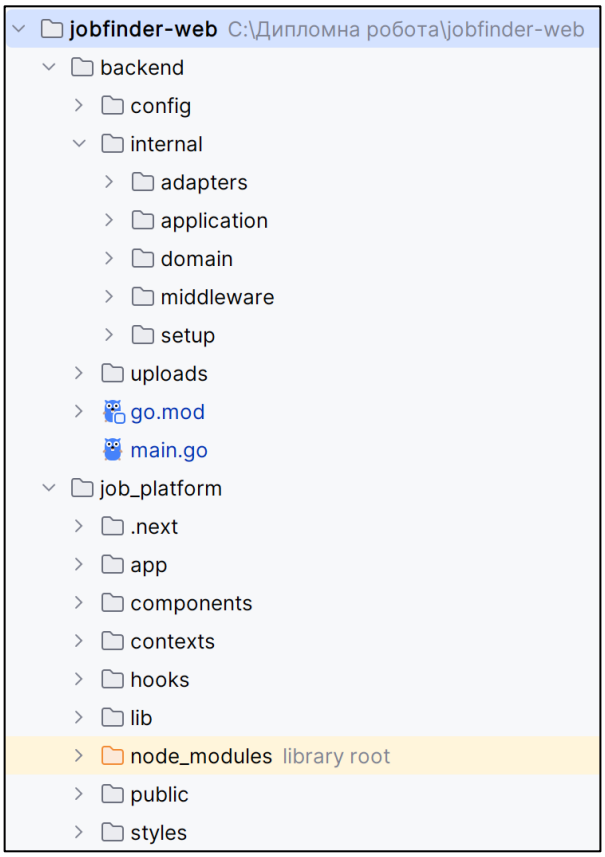


Рисунок 3.5 – Структура backend та frontend частини

Реалізація системи на Go включає у себе:

– авторизація та автентифікація: реалізовано з використанням JWT, зберігання хешованих паролів;

– логування: за допомогою zap усі запити логуються, включаючи HTTP-метод, статус, IP користувача та затримку;

– обробка HTTP-запитів: реалізована через Fiber з використанням middleware;

– міграції: виконуються автоматично при запуску через AutoMigrate;

– CRUD-операції: створення, редагування та перегляд вакансій і резюме;

– створення структури користувача;

– підключення до бази даних PostgreSQL.

У клієнтській частині на React реалізовано:

– SPA з маршрутизацією;

– основні компоненти: MainLayout (кореневий обгортуючий компонент);

– компоненти для вакансії: JobsListPage, JobDetailsPage, JobApplyForm, PostJobPage, EditJobPage;

– компоненти для заявки: ApplicationsListPage, ApplicationDetailsPage;

– допоміжні компоненти: PublicUserProfilePage, UnauthorizedPage;

– компоненти для користувача: LoginPage, RegisterPage, ProfilePage, SettingsPage, API-запити.

Розроблений сервіс демонструє стабільну роботу основного функціоналу: автентифікації, створення резюме, фільтрації вакансій та логування запитів, а розроблена архітектура дозволяє легко масштабувати проєкт.

На рис. 3.6 представлено ER-діаграма бази даних.

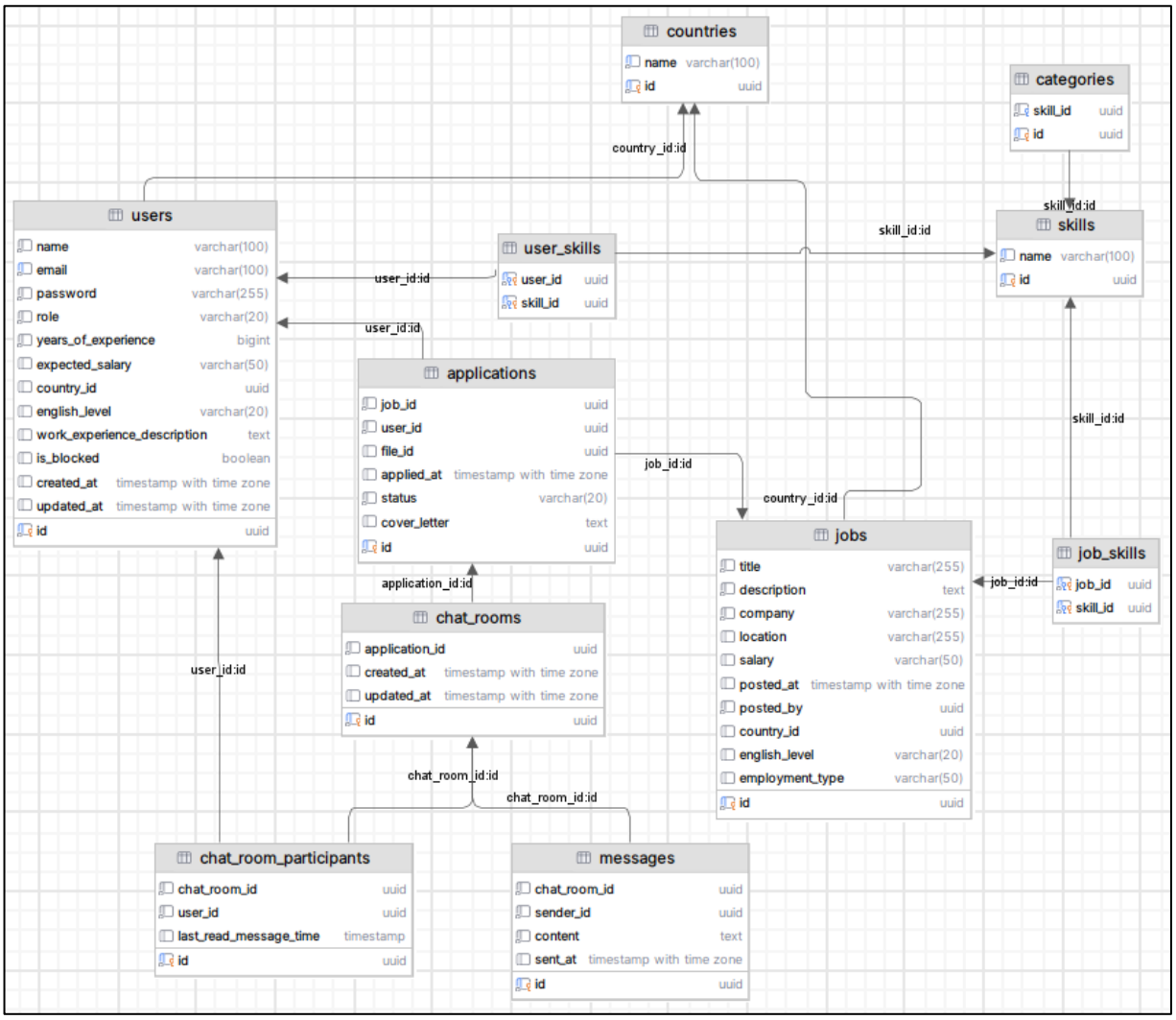


Рисунок 3.6 – ER-діаграма бази даних

У наступному підрозділі буде детально описано реалізований функціонал онлайн-сервісу.

3.3 Опис функцій розробленого сайту

При розробці онлайн-сервісу було реалізовано багато компонентів з використанням Go та React.

3.3.1 Backend частина

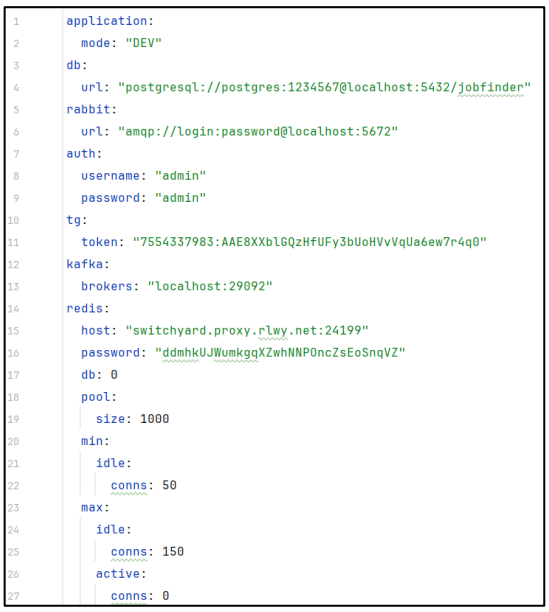
Частина backend побудована за принципами чистої архітектури (Clean Architecture) [21], що розділяє логіку на чіткі рівні відповідальності.

Коротко опишемо частину **config**, яка представлена у Додатку А, а також на рис. 3.7. Ця частина містить конфігураційні файли для запуску додатку. У проекті

використовується окремий модуль config, який відповідає за зчитування параметрів конфігурації з файлу config.yaml, який представлено на рис. 3.8 та у Додатку А, або змінних середовища. Для цього застосовано бібліотеку viper, яка дозволяє автоматично адаптувати значення з різних джерел. Конфігураційні дані зберігаються у структурі Config, яка доступна у вигляді глобальної змінної Conf. Завдяки цьому вся система легко конфігурується без жорстко закодованих значень у коді.

```go
package config

import (
    "fmt"
    "strings"

    "github.com/spf13/viper"
)

type Config struct {
    DB struct {
        URL string
    }
}

var Conf Config

func init() {
    viper.SetConfigName("config")
    viper.SetConfigType("yaml")
    viper.AddConfigPath("./config/")

    viper.SetEnvKeyReplacer(strings.NewReplacer(".", "_"))
    viper.AutomaticEnv()
    if err := viper.ReadInConfig(); err != nil { fmt.Print("Config file not found...") }

    if err := viper.Unmarshal(&Conf); err != nil { panic(fmt.Sprintf("Unable to decode into struct %e", err)) }
}
```

Рисунок 3.7 – Код програми conf.go



```yaml
application:
  mode: "DEV"
db:
  url: "postgresql://postgres:1234567@localhost:5432/jobfinder"
rabbit:
  url: "amqp://login:password@localhost:5672"
auth:
  username: "admin"
  password: "admin"
tg:
  token: "7554337983:AAE8XXblGQzHfUFy3bUoHVvVqUa6ew7r4q0"
kafka:
  brokers: "localhost:29092"
redis:
  host: "switchyard.proxy.rlwy.net:24199"
  password: "ddmhkUJWumkgqXZwhNNPOncZsEoSnqVZ"
  db: 0
  pool:
    size: 1000
  min:
    idle:
      conns: 50
  max:
    idle:
      conns: 150
    active:
      conns: 0
```

Рисунок 3.8 – Код файлу config.yaml

Наступною опишемо частину основної бізнес-логіки та реалізацію сервісу. Для опису візьмемо лише основні компоненти цієї частини, а саме фрагмент коду, який реалізує репозиторій для роботи з відгуками на вакансії (applications) у системі пошуку роботи. Це частина шару доступу до бази даних у архітектурі на Go, побудованій на базі ORM-бібліотеки GORM.

Цей пакет **applicationRepository.go** відповідає за всі CRUD-операції (create, read, update, delete) з таблицею applications у базі даних PostgreSQL. Він реалізує інтерфейс ApplicationRepository, який знаходиться в пакеті domain.

Опишемо функції, які реалізовано у коді.

Функція **NewApplicationRepository** є фабричним методом, який відповідає за створення нового екземпляра репозиторію заявок (application repository) у контексті роботи з базою даних за допомогою ORM-фреймворку GORM. Цей метод приймає вхідний параметр – екземпляр *gorm.DB, що представляє з'єднання з базою даних. Усередині функції створюється новий об'єкт типу applicationRepository, якому передається об'єкт бази даних. Після цього викликається допоміжний метод init(), який використовується для ініціалізації додаткових структур або параметрів репозиторію. Повертається інтерфейс domain.ApplicationRepository, що забезпечує абстрагування доступу до даних згідно з принципами чистої архітектури. Такий підхід дозволяє легко змінювати або тестувати реалізацію без потреби модифікувати інші частини системи. Ця функція відіграє важливу роль у впровадженні шаблону Repository, який ізолює бізнес-логіку від деталей зберігання даних та полегшує підтримку і розширення програмного забезпечення.

Наступною функцією є **init**. Метод init використовується для автоматичної міграції структури сутності Application до бази даних. За допомогою функції AutoMigrate від GORM забезпечується створення необхідної таблиці відповідно до визначення моделі. У разі помилки ініціалізації викликається паніка з текстом помилки. Це дозволяє уникнути подальшого запуску програми у разі неправильної конфігурації бази даних.

Далі розглянемо функцію **CreateApplication**. Цей метод реалізує логіку створення нової заявки. Він приймає контекст виконання (context.Context) та вказівник на об'єкт заявки Application. За допомогою методу Create бібліотеки GORM здійснюється вставка нового запису у відповідну таблицю бази даних. Метод повертає помилку у разі неуспішного виконання запиту.

Наступною представимо функцію **GetApplicationByID**. Цей метод здійснює пошук заявки за її ідентифікатором. Якщо запис знайдено, повертається вказівник на об'єкт Application. У випадку, якщо запис не знайдено, повертається nil без помилки. Якщо виникає інша помилка (наприклад, проблема з підключенням до бази), вона повертається викликачу. Такий підхід дозволяє чітко розрізняти ситуацію, коли запис відсутній, і коли виникла критична помилка.

Перейдемо до методу **ListApplicationsByUser**. Цей метод виконує вибірку всіх заявок, що належать певному користувачу. В якості параметра передається userID, який фільтрується у запиті. Результатом є масив вказівників на об'єкти Application. У разі помилки виконується її обробка та повернення.

Переглянемо схожу функцію **ListApplicationsByJob**. Аналогічно попередньому методу, ListApplicationsByJob виконує вибірку всіх заявок, які пов'язані з конкретною вакансією (jobID). Це дозволяє, наприклад, роботодавцю отримати повний список усіх заявок, поданих на певну вакансію. Дані повертаються у вигляді масиву вказівників на заявки.

Також було розроблено функцію **UpdateApplication**. Цей метод відповідає за оновлення наявного запису заявки в базі даних. Якщо запис оновлено успішно, метод повертає nil. У разі, якщо запис не було змінено (наприклад, через його відсутність), повертається помилка про те, що жоден рядок не було змінено. Таким чином, метод гарантує, що зміни дійсно були застосовані.

Ще одним методом є **DeleteApplication**. Він видаляє заявку за її ідентифікатором. Якщо запис не знайдено або не було видалено, повертається відповідна помилка. Це дозволяє програмі коректно реагувати на ситуації, коли, наприклад, користувач намагається видалити неіснуючу заявку.

Останньою функцією з цього файлу є **ListAllApplications**. Цей метод надає можливість отримати повний список усіх заявок, що зберігаються в базі даних. Цей метод може бути використаний адміністративними модулями для повного аналізу даних або відображення у звітах. Повертається масив вказівників на заявки або помилка у разі невдачі виконання запиту.

У табл. 3.3 продемонстровано коротко усі CRUD-операції з цього файлу, які описано вище.

Таблиця 3.3 – Опис методів з applicationRepository.go

Метод	**Опис**
CreateApplication(ctx, app)	Додає нову заявку в БД
GetApplicationByID(ctx, id)	Повертає заявку по ID
ListApplicationsByUser(ctx, userID)	Повертає всі заявки певного користувача
ListApplicationsByJob(ctx, jobID)	Повертає всі заявки на певну вакансію
UpdateApplication(ctx, app)	Оновлює дані заявки
DeleteApplication(ctx, id)	Видаляє заявку по ID
ListAllApplications(ctx)	Повертає всі заявки з БД

Використання ctx context.Context в кожному методі забезпечує контроль за таймаутами, скасуваннями або логуванням – це стандартна практика в Go для керованих запитів до бази.

Наступним у системі реалізовано репозиторій **categoryRepository**, який відповідає за роботу з категоріями вакансій. Даний репозиторій зображено на рис. 3.9. Він дозволяє створювати нові категорії та отримувати їхній список разом із пов’язаними навичками. Для роботи з базою даних використано GORM, що забезпечує автоматичну міграцію структури таблиці та ефективну ORM-обробку даних. Репозиторій реалізує інтерфейс CategoryRepository із бізнес-шару domain,

що дозволяє легко масштабувати логіку без зміни нижчого рівня. У табл. 3.4 наведено основні методи цієї частини.

```go
package db

import (
    "context"
    "gorm.io/gorm"
    "jobfinder-web/internal/domain"
)

Implement interface
type categoryRepository struct {  4 usages  new *
    db *gorm.DB
}

func NewCategoryRepository(db *gorm.DB) domain.CategoryRepository {  1 usage  new *
    repo := &categoryRepository{db: db}
    repo.init()
    return repo
}

func (r *categoryRepository) init() {  1 usage  new *
    if err := r.db.AutoMigrate(&domain.Category{}); err != nil { panic("Failed to migrate database: " + err.Error()) }
}

func (r *categoryRepository) CreateCategory(ctx context.Context, category *domain.Category) error {  no usages  new *
    return r.db.WithContext(ctx).Create(category).Error
}

func (r *categoryRepository) GetAllCategories(ctx context.Context) ([]*domain.Category, error) {  1 usage  new *
    var categories []*domain.Category
    if err := r.db.WithContext(ctx).Preload( query: "Skill").Find(&categories).Error; err != nil { return nil, err }
    return categories, nil
}
```

Рисунок 3.9 – categoryRepository

Таблиця 3.4 – Опис основних методів з categoryRepository

Метод	**Опис**
CreateCategory(ctx, category)	Додає нову категорію в базу даних
GetAllCategories(ctx)	Отримує всі категорії, включно з пов’язаними навичками (Skill)

У рамках реалізації системи спілкування між шукачами та роботодавцями був створений окремий модуль **chatRoomRepository,** який можна продивитись у Додатку А. Він забезпечує створення та управління чат-кімнатами, додавання користувачів до чату та збереження часу останнього переглянутого повідомлення. Всі операції реалізовані через ORM GORM, що забезпечує безпечну роботу з базою

даних PostgreSQL. Такий підхід дозволяє забезпечити асинхронну взаємодію сторін у межах онлайн-сервісу пошуку роботи. У табл. 3.5 наведено основні методи цієї частини.

Таблиця 3.5 – Опис методів з applicationRepository.go

Метод	**Опис**
CreateChatRoom	Створює нову чат-кімнату в базі даних
GetChatRoomByApplicationID	Повертає чат за ID заявки на вакансію
GetChatRoomByID	Отримує чат по його унікальному ID
CreateParticipant	Додає користувача як учасника до чат-кімнати
GetParticipant	Перевіряє, чи користувач є учасником певної кімнати
UpdateParticipantLastReadTime	Оновлює час останнього прочитаного повідомлення
GetChatRoomsForUser	Повертає всі чати, в яких бере участь користувач
DeleteChatRoom	Видаляє чат-кімнату разом з усіма її учасниками

У цієї частини коду також існують свої унікальності, такі як:

– AutoMigrate гарантує автоматичне створення таблиць при запуску сервісу;

– LastReadMessageTime – використовується для показу непрочитаних повідомлень;

– Joins – запит об'єднує chat_rooms та chat_room_participants, щоб отримати всі чати користувача;

– UUID – використовується як унікальний ідентифікатор для учасників чатів.

У процесі розробки було реалізовано окремий репозиторій **countryRepository**, який зображено на рис. 3.10. Він відповідає за роботу з

таблицею країн у базі даних. Основні функції – створення нових записів про країни та отримання повного списку для подальшого використання у фільтрах вакансій або профілях користувачів. Всі операції виконуються через ORM-бібліотеку GORM з використанням контексту, що забезпечує масштабованість та стабільність системи.

```go
package db

import (
    "context"
    "gorm.io/gorm"
    "jobfinder-web/internal/domain"
)

Implement interface
type countryRepository struct {  4 usages  new *
    db *gorm.DB
}

func NewCountryRepository(db *gorm.DB) domain.CountryRepository {  1 usage  new *
    repo := &countryRepository{db: db}
    repo.init()
    return repo
}

func (r *countryRepository) init() {  1 usage  new *
    if err := r.db.AutoMigrate(&domain.Country{}); err != nil { panic("Failed to migrate database: " + err.Error()) }
}

func (r *countryRepository) CreateCountry(ctx context.Context, country *domain.Country) error {  no usages  new *
    return r.db.WithContext(ctx).Create(country).Error
}

func (r *countryRepository) GetAllCountries(ctx context.Context) ([]*domain.Country, error) {  1 usage  new *
    var countries []*domain.Country
    if err := r.db.WithContext(ctx).Find(&countries).Error; err != nil { return nil, err }
    return countries, nil
}
```

Рисунок 3.10 – countryRepository

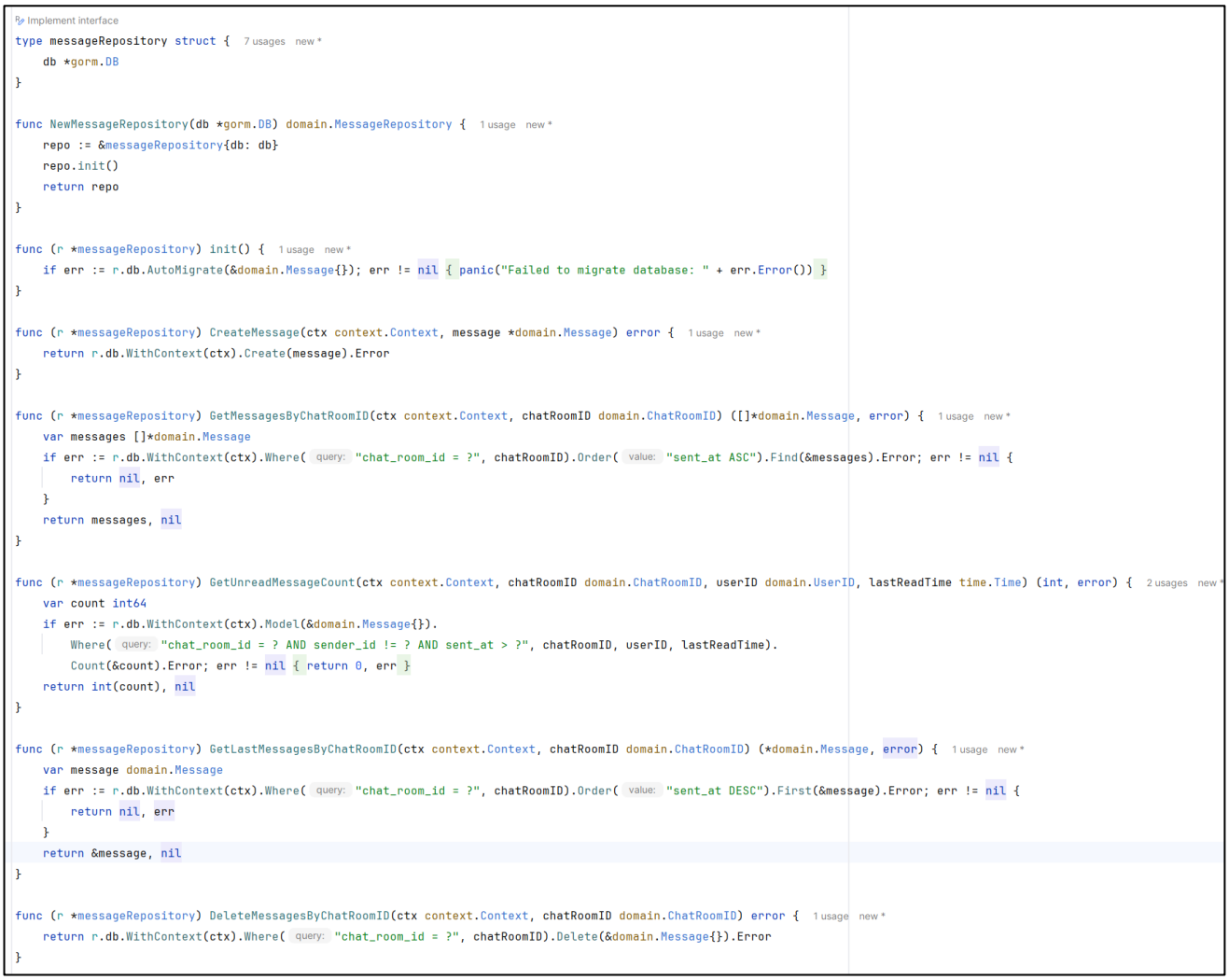
У репозиторії **jobRepository,** який представлено у Додатку А, реалізовано всі базові CRUD-операції над вакансіями, а також механізми пошуку та фільтрації. Найважливішим є метод SearchJobs, який забезпечує складний пошук вакансій за різними критеріями: навички та зарплата. Завдяки застосуванню GORM-підходу з Preload забезпечено ефективне завантаження зв’язаних даних. Операції видалення вакансій виконуються у транзакції, що забезпечує цілісність бази даних. Усе це

дозволяє реалізувати гнучкий і масштабований модуль взаємодії з вакансіями у системі. У табл. 3.6 описано усі операції з вакансіями

Таблиця 3.6 – Операції з вакансіями

Що робить	**Метод**	**Опис**
Ініціалізація	func (r *jobRepository) init()	Автоматично створює таблиці jobs та skills, використовуючи GORM AutoMigrate.
Створення вакансії	func (r *jobRepository) CreateJob(ctx context.Context, job *domain.Job) error	Зберігає вакансію у базі да них.
Отримання вакансії за ID	func (r *jobRepository) GetJobByID(...)	Повертає вакансію разом із пов’язаними навичками (через Preload("RequiredSkills")).
Пошук і фільтрація вакансій	func (r *jobRepository) SearchJobs(ctx context.Context, query string, filters map[string]interface{}, page, pageSize int)	Підтримує повнотекстовий пошук по title і description та фільтрацію.
Отримання вакансій, створених користувачем	func (r *jobRepository) GetJobsByUser(...)	Фільтрує всі вакансії, створені певним користувачем (роботодавцем), з пагінацією.
Оновлення вакансії	func (r *jobRepository) UpdateJob(...)	Оновлює всі поля вакансії. Перевіряє, що запис знайдено (через RowsAffected).
Видалення вакансії	func (r *jobRepository) DeleteJob(ctx context.Context, id domain.JobID)	видаляє пов’язані записи у кількох таблицях очищує зв'язок.
Отримати всі вакансії	func (r *jobRepository) ListAllJobs(...)	Простий метод для виведення всіх вакансій із preload-ом навичок.

У модулі **messageRepository,** який зображено на рис. 3.11, реалізовано всю основну логіку обміну повідомленнями між користувачами в межах конкретної заявки.



```
Implement interface
type messageRepository struct {  7 usages  new *
    db *gorm.DB
}

func NewMessageRepository(db *gorm.DB) domain.MessageRepository {  1 usage  new *
    repo := &messageRepository{db: db}
    repo.init()
    return repo
}

func (r *messageRepository) init() {  1 usage  new *
    if err := r.db.AutoMigrate(&domain.Message{}); err != nil { panic("Failed to migrate database: " + err.Error()) }
}

func (r *messageRepository) CreateMessage(ctx context.Context, message *domain.Message) error {  1 usage  new *
    return r.db.WithContext(ctx).Create(message).Error
}

func (r *messageRepository) GetMessagesByChatRoomID(ctx context.Context, chatRoomID domain.ChatRoomID) ([]*domain.Message, error) {  1 usage  new *
    var messages []*domain.Message
    if err := r.db.WithContext(ctx).Where( query: "chat_room_id = ?", chatRoomID).Order( value: "sent_at ASC").Find(&messages).Error; err != nil {
        return nil, err
    }
    return messages, nil
}

func (r *messageRepository) GetUnreadMessageCount(ctx context.Context, chatRoomID domain.ChatRoomID, userID domain.UserID, lastReadTime time.Time) (int, error) {  2 usages  new *
    var count int64
    if err := r.db.WithContext(ctx).Model(&domain.Message{}).
        Where( query: "chat_room_id = ? AND sender_id != ? AND sent_at > ?", chatRoomID, userID, lastReadTime).
        Count(&count).Error; err != nil { return 0, err }
    return int(count), nil
}

func (r *messageRepository) GetLastMessagesByChatRoomID(ctx context.Context, chatRoomID domain.ChatRoomID) (*domain.Message, error) {  1 usage  new *
    var message domain.Message
    if err := r.db.WithContext(ctx).Where( query: "chat_room_id = ?", chatRoomID).Order( value: "sent_at DESC").First(&message).Error; err != nil {
        return nil, err
    }
    return &message, nil
}

func (r *messageRepository) DeleteMessagesByChatRoomID(ctx context.Context, chatRoomID domain.ChatRoomID) error {  1 usage  new *
    return r.db.WithContext(ctx).Where( query: "chat_room_id = ?", chatRoomID).Delete(&domain.Message{}).Error
}
```

Рисунок 3.11 – Message_Repository

Реалізовано створення повідомлень, отримання історії чату, підрахунок непрочитаних повідомлень і видалення всіх повідомлень за chat_room_id. Це дозволяє реалізувати повноцінну функцію спілкування на платформі, що особливо актуально для платформи пошуку роботи, де роботодавець і кандидат можуть узгодити деталі перед співбесідою. Завдяки використанню GORM-ORM методи реалізовані ефективно та з дотриманням транзакційної безпеки. У табл. 3.7 розписано основні методи логіки обміну повідомлень.

Таблиця 3.7 – Методи обміну повідомленнями

Метод	**Опис**

func (r *messageRepository) CreateMessage(ctx context.Context, message *domain.Message) error	Зберігає нове повідомлення в базі даних. Типовий випадок – користувач надсилає повідомлення в чаті.
func (r *messageRepository) GetMessagesByChatRoomID(...)	Повертає **всі повідомлення** в чаті (за chat_room_id) у хронологічному порядку (Order("sent_at ASC")). Це основна функція для завантаження історії чату.
func (r *messageRepository) GetUnreadMessageCount(...)	Повертає кількість **непрочитаних повідомлень** в чаті, тобто повідомлень, надісланих після останнього прочитання (lastReadTime) і не користувачем (sender_id != ?). Цей метод використовується для відображення "нових повідомлень".
func (r *messageRepository) GetLastMessagesByChatRoomID(...)	Повертає **останнє повідомлення** з чату. Може бути використано в списку чатів для прев'ю останньої активності.
func (r *messageRepository) DeleteMessagesByChatRoomID(...)	Видаляє всі повідомлення з конкретного чату. Наприклад, під час повного видалення чату або заявки.

Наступним опишемо репозиторій навичок **skillRepository,** який відповідає за CRUD-операції над таблицею skills. Цей файл зображено на рис. 3.12. В онлайн-сервісі пошуку роботи ця таблиця використовується для опису необхідних технічних умінь у вакансіях або компетенцій у резюме. Наприклад, JavaScript, React, Go, PostgreSQL тощо. Під час запуску системи відбувається автоматична міграція таблиці. Передбачено функції для створення нових навичок і вибірки всього списку доступних навичок, які можуть бути використані у фільтрації або автозаповненні полів форми.

```go
package db

import (
    "context"
    "gorm.io/gorm"
    "jobfinder-web/internal/domain"
)

Implement interface
type skillRepository struct {  4 usages  new *
    db *gorm.DB
}

func NewSkillRepository(db *gorm.DB) domain.SkillRepository {  1 usage  new *
    repo := &skillRepository{db: db}
    repo.init()
    return repo
}

func (r *skillRepository) init() {  1 usage  new *
    if err := r.db.AutoMigrate(&domain.Skill{}); err != nil { panic("Failed to migrate database: " + err.Error()) }
}

func (r *skillRepository) CreateSkill(ctx context.Context, skill *domain.Skill) error {  no usages  new *
    return r.db.WithContext(ctx).Create(skill).Error
}

func (r *skillRepository) GetAllSkills(ctx context.Context) ([]*domain.Skill, error) {  1 usage  new *
    var skills []*domain.Skill
    if err := r.db.WithContext(ctx).Find(&skills).Error; err != nil { return nil, err }
    return skills, nil
}
```

Рисунок 3.12 – skillRepository

Також у роботі було реалізовано репозиторій користувачів. **userRepository,** який представлено у Додатку А, – це основний компонент backend-частини, який забезпечує збереження та обробку даних профілів користувачів. У межах онлайн-сервісу пошуку роботи він відповідає за створення акаунтів, оновлення профілю, отримання детальної інформації про користувача, включаючи пов’язані навички. Репозиторій використовує ORM GORM для взаємодії з базою даних PostgreSQL і гарантує безпечну та ефективну роботу з інформацією.

У табл. 3.8 наведено опис ключових методів репозиторію користувачів.

Таблиця 3.8 – Ключові методи репозиторію користувачів

Метод	Опис

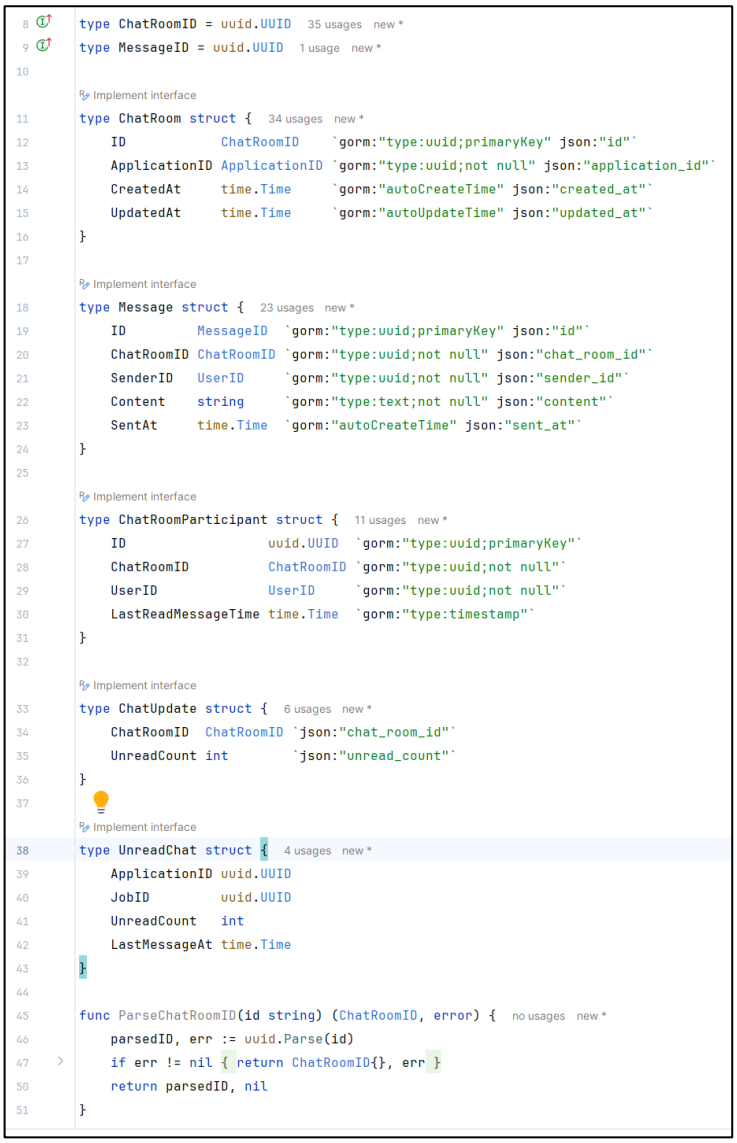
func (r *userRepository) init()	Виконує автоматичну міграцію таблиць users і skills, гарантуючи, що ці таблиці існують перед використанням.
func (r *userRepository) CreateUser(ctx context.Context, user *domain.User) error	Додає нового користувача до бази. Використовується під час реєстрації користувача на сайті.
func (r *userRepository) GetUserByID(ctx context.Context, id domain.UserID) (*domain.User, error)	Отримує користувача за ідентифікатором, також підвантажує навички (Preload("Skills")). Це корисно при відображенні профілю користувача.
func (r *userRepository) FindUserByEmail(ctx context.Context, email string) (*domain.User, error)	Використовується для автентифікації користувача. Шукає користувача за email. Також підтягує навички.
func (r *userRepository) UpdateUser(ctx context.Context, user *domain.User) error	Оновлює дані користувача. Повертає помилку, якщо запис не знайдено (RowsAffected == 0).
func (r *userRepository) DeleteUser(ctx context.Context, id domain.UserID) error	Видаляє користувача з бази.
func (r *userRepository) ListUsers(ctx context.Context) ([]*domain.User, error)	Отримує список усіх користувачів з їхніми навичками. Це може використовуватись у адмін-панелі або для аналітики.

Під час розробки у пакеті domain визначено структуру **Application**, яка описує заявку на вакансію. Код з цього файлу представлено на рис. 3.13. Вона включає ID вакансії, користувача, дату подачі, супровідний лист, прикріплений файл та статус. Статус заявки представлений у вигляді перелічення (enum) зі значеннями pending, accepted, rejected, withdrawn, що полегшує контроль життєвого циклу заявки. Завдяки анотаціям GORM, ця структура автоматично зв'язується з базою даних, забезпечуючи зручність при CRUD-операціях.

```go
package domain

import (
    "github.com/google/uuid"
    "time"
)

type ApplicationID = uuid.UUID  20 usages  new *
Implement interface
type ApplicationStatus string  8 usages  new *

const (
    Pending   ApplicationStatus = "pending"  1 usage  new *
    Accepted  ApplicationStatus = "accepted"  no usages  new *
    Rejected  ApplicationStatus = "rejected"  no usages  new *
    Withdrawn ApplicationStatus = "withdrawn"  no usages  new *
)

Implement interface
type Application struct {  34 usages  new *
    ID          uuid.UUID         `gorm:"type:uuid;primaryKey" json:"id"`
    JobID       JobID              `gorm:"type:uuid;not null" json:"job_id"`
    UserID      UserID             `gorm:"type:uuid;not null" json:"user_id"`
    FileId      uuid.UUID         `gorm:"type:uuid" json:"file_id"`
    AppliedAt   time.Time         `gorm:"autoCreateTime" json:"submitted_at"`
    Status      ApplicationStatus `gorm:"size:20;not null" json:"status"`
    CoverLetter string            `gorm:"type:text" json:"cover_letter"`
}

func ParseApplicationID(id string) (ApplicationID, error) {  1 usage  new *
    parsedID, err := uuid.Parse(id)
    if err != nil { return ApplicationID{}, err }
    return parsedID, nil
```

Рисунок 3.13 – Application.go

Для забезпечення безпосередньої комунікації між роботодавцем і шукачем після подання заявки, у системі реалізовано модуль чату **chat.go,** який зображено на рис. 3.14. Чат функціонує у межах конкретної заявки – після її створення формується окрема чат-кімната (ChatRoom), що пов’язана з ApplicationID. Повідомлення (Message) зберігаються у базі даних і мають дані про відправника, час надсилання та текст. Завдяки структурі ChatRoomParticipant фіксується, які користувачі беруть участь у спілкуванні, а також зберігається інформація про останнє прочитане повідомлення, що дозволяє реалізувати індикатори нових повідомлень. Такий підхід дозволяє зручно реалізувати чат, пов’язаний із конкретною вакансією, прямо в особистому кабінеті.



```go
type ChatRoomID = uuid.UUID  35 usages  new *
type MessageID = uuid.UUID  1 usage  new *

Implement interface
type ChatRoom struct {  34 usages  new *
    ID            ChatRoomID     `gorm:"type:uuid;primaryKey" json:"id"`
    ApplicationID ApplicationID `gorm:"type:uuid;not null" json:"application_id"`
    CreatedAt     time.Time      `gorm:"autoCreateTime" json:"created_at"`
    UpdatedAt     time.Time      `gorm:"autoUpdateTime" json:"updated_at"`
}

Implement interface
type Message struct {  23 usages  new *
    ID         MessageID   `gorm:"type:uuid;primaryKey" json:"id"`
    ChatRoomID ChatRoomID `gorm:"type:uuid;not null" json:"chat_room_id"`
    SenderID   UserID      `gorm:"type:uuid;not null" json:"sender_id"`
    Content    string      `gorm:"type:text;not null" json:"content"`
    SentAt     time.Time   `gorm:"autoCreateTime" json:"sent_at"`
}

Implement interface
type ChatRoomParticipant struct {  11 usages  new *
    ID                  uuid.UUID  `gorm:"type:uuid;primaryKey"`
    ChatRoomID          ChatRoomID `gorm:"type:uuid;not null"`
    UserID              UserID     `gorm:"type:uuid;not null"`
    LastReadMessageTime time.Time  `gorm:"type:timestamp"`
}

Implement interface
type ChatUpdate struct {  6 usages  new *
    ChatRoomID  ChatRoomID `json:"chat_room_id"`
    UnreadCount int        `json:"unread_count"`
}

Implement interface
type UnreadChat struct {  4 usages  new *
    ApplicationID uuid.UUID
    JobID         uuid.UUID
    UnreadCount   int
    LastMessageAt time.Time
}

func ParseChatRoomID(id string) (ChatRoomID, error) {  no usages  new *
    parsedID, err := uuid.Parse(id)
    if err != nil { return ChatRoomID{}, err }
    return parsedID, nil
}
```

Рисунок 3.14 – chat.go

Вакансії у системі моделюються через структуру **Job**, що містить ключову інформацію: назву, опис, компанію, зарплату, тип зайнятості, країну та вимоги до рівня володіння англійською мовою. Код цієї частини зображено на рис. 3.15. Для підтримки зв’язку між вакансіями та навичками використано відношення типу "багато до багатьох" (many-to-many) між сутностями Job і Skill через проміжну таблицю job_skills. Завдяки цьому система дозволяє зручно зберігати та фільтрувати вакансії за вимогами до навичок. Ідентифікатори записів реалізовані на основі UUID, що забезпечує глобальну унікальність і безпеку. Зв’язок вакансії з автором реалізується через поле PostedBy, що відповідає UserID.

```go
package domain

import (
    "github.com/google/uuid"
    "time"
)

type JobID = uuid.UUID  new *

Implement interface
type Job struct {  33 usages  new *
    ID             uuid.UUID    `gorm:"type:uuid;primaryKey" json:"id"`
    Title          string       `gorm:"size:255;not null" json:"title"`
    Description    string       `gorm:"type:text;not null" json:"description"`
    Company        string       `gorm:"size:255;not null" json:"company"`
    Location       string       `gorm:"size:255" json:"location"`
    Salary         string       `gorm:"size:50" json:"salary"`
    PostedAt       time.Time    `gorm:"autoCreateTime" json:"posted_at"`
    PostedBy       UserID       `gorm:"type:uuid;not null" json:"posted_by"`
    CountryID      uuid.UUID    `gorm:"type:uuid" json:"country_id"`
    EnglishLevel   EnglishLevel `gorm:"size:20" json:"english_level"`
    EmploymentType string       `gorm:"size:50" json:"employment_type"`
    RequiredSkills []Skill      `gorm:"many2many:job_skills;" json:"skills"`
}

func ParseJobID(id string) (JobID, error) {  1 usage  new *
    parsedID, err := uuid.Parse(id)
    if err != nil { return JobID{}, err }
    return parsedID, nil
}
```

Рисунок 3.15 – Job.go

Розроблена частина коду у файлі **ports.go** є ключовим інтерфейсним контрактом у проєкті, яка представлена у Додатку А. Він визначає архітектуру взаємодії між рівнями: даних, бізнес-логіки та сервісів. У табл. 3.9 описано інтерфейси репозиторіїв, які є основними блоками файлу. А у табл. 3.10 описано інтерфейси сервісів, які реалізують бізнес-логіку з даного файлу.

Таблиця 3.9 – Repository інтерфейси

Метод	**Опис**
UserRepository	управління обліковими записами користувачів.
JobRepository	робота з вакансіями.
ApplicationRepository	управління поданими заявками на вакансії.
ChatRoomRepository	управління чатами та учасниками.
MessageRepository	управління повідомленнями.
CountryRepository	довідникові таблиці.
CategoryRepository	довідникові таблиці.
SkillRepository	довідникові таблиці.

Таблиця 3.10 – Service інтерфейси

Метод	Опис
UserService	логіка реєстрації, входу, зміни профілю.
JobService	логіка створення, пошуку та редагування вакансій.
ApplicationService	логіка подачі заявок та їх обробки.
ChatService	логіка обміну повідомленнями, створення чатів.
CountryService	для отримання довідкової інформації.
CategoryService	для отримання довідкової інформації.
SkillService	для отримання довідкової інформації.
FileService	збереження файлів (наприклад, PDF-резюме).

Однією з центральних сутностей онлайн-сервісу пошуку роботи є користувач. У системі реалізовано єдину модель User, яка підтримує різні ролі: шукач роботи та роботодавець. Це дозволяє гнучко керувати правами доступу та функціональністю залежно від типу користувача. У коді реалізовано дві ролі, а саме JobSeeker та Employer.

Основні поля у моделі користувачів:

– ID – унікальний ідентифікатор користувача (UUID);

– Name, Email, Password – особисті дані користувача, email є унікальним;

– Role – роль користувача (jobseeker, employer), яка визначає доступ до функціоналу;

– YearsOfExperience, ExpectedSalary, EnglishLevel – поля для фільтрації в пошуку;

– CountryID – зв’язок з країною;

– WorkExperienceDescription – короткий опис професійного досвіду;

– Skills – список пов’язаних навичок;

– IsBlocked – логічне поле для блокування користувача;

– CreatedAt, UpdatedAt – часові мітки створення і оновлення.

Завдяки гнучкому визначенню ролей та підтримці багатьох навичок, ця модель дозволяє адаптувати платформу під сучасні вимоги ринку праці, зокрема в галузі IT.

У відповідності до принципів багаторівневої архітектури, логіка бізнес-операцій у розробленому програмному забезпеченні винесена в окремий шар – **сервісний рівень**. У даному проєкті цей рівень реалізовано у вигляді пакету application, який містить окремі файли сервісів для кожного ключового напряму доменної моделі. Кожен сервіс відповідає за реалізацію операцій, пов'язаних з певною сутністю, та слугує проміжною ланкою між контролерами (API-рівень) і репозиторіями (рівень доступу до даних). Основні функції сервісів – це обробка бізнес-логіки, перевірка вхідних даних, трансформація даних та координація викликів до репозиторіїв.

Почнемо з **application_service.go**, який представлено у Додатку А. Цей файл реалізує логіку роботи з заявками користувачів на вакансії. Він відповідає за координацію взаємодії між рівнем зберігання даних (репозиторієм заявок) та іншими сервісами (наприклад, чат-сервісом).

Основні функції сервісу:

– подача заявки користувачем на вакансію (ApplyToJob), із автоматичним створенням кімнати чату між заявником і роботодавцем;

– отримання списку заявок користувача або вакансії (ListUserApplications, ListJobApplications);

– оновлення заявки, зокрема зміна статусу або супровідного листа (UpdateApplication, UpdateApplicationStatus);

– видалення заявки, а також пов'язаної з нею кімнати чату (DeleteApplication);

– отримання заявки за ID або перегляд усіх наявних заявок (GetApplicationByID, ListAllApplications);

– масове видалення заявок, пов'язаних із певною вакансією (DeleteApplicationsByJobID).

Цей сервіс інкапсулює бізнес-логіку, пов’язану з обробкою заявок, забезпечує валідацію даних і координацію пов’язаних дій у системі.

Наступним є файл **chat_service.go,** який представлено у Додатку А. Цей файл реалізує логіку взаємодії користувачів за допомогою чату, що прив’язаний до конкретної заявки на роботу. Основна мета сервісу – забезпечити приватну комунікацію між шукачем роботи та роботодавцем.

Основні функції:

– створення кімнати чату для заявки (CreateChatRoomForApplication) – ініціалізує кімнату між двома користувачами, а саме шукачем і роботодавцем;

– надсилання повідомлення (SendMessage) – зберігає повідомлення у відповідній кімнаті чату;

– отримання всіх повідомлень у чаті та останнього повідомлення (GetMessages, GetLastMessage);

– отримання кількості непрочитаних повідомлень та оновлення часу останнього прочитання (GetUnreadMessageCount, UpdateLastReadTime);

– відстеження непрочитаних чатів та надання зведеної інформації по кожному з них (GetUnreadChats);

– видалення чату разом з усіма повідомленнями (DeleteChatRoom);

– отримання списку чатів користувача (GetChatRoomsForUser);

– довгополінгове оновлення чату – реалізується через метод GetChatUpdates, який перевіряє наявність нових повідомлень у вказаний період часу.

Завдяки даному сервісу забезпечується асинхронне, безпечне та зручне спілкування між сторонами найму в рамках кожної окремої заявки.

Далі розглянемо **file_service.go,** який зображено на рис. 3.16. Цей модуль реалізує функціональність зберігання та доступу до файлів, які прикріплюються користувачами при подачі заявки на вакансію. Цей сервіс дозволяє забезпечити централізоване та унікальне збереження файлів, таких як резюме.

Основні функції:

– SaveFile – приймає файл з HTTP-запиту, генерує унікальний UUID, додає його до імені файлу та зберігає файл у локальну директорію ./uploads. Завдяки цьому імена файлів не конфліктують між собою, і кожен файл має унікальний шлях;

– GetFileByUUID – дозволяє знайти файл у директорії за UUID, що був присвоєний при збереженні, та повертає повний шлях до цього файлу.

Таким чином, сервіс забезпечує безпечне збереження та подальший доступ до файлів, що є важливою частиною функціоналу онлайн-платформи пошуку роботи.

```
Implement interface
type fileService struct{}  3 usages  new *

func NewFileService() domain.FileService { return &fileService{} }

func (s *fileService) SaveFile(file *multipart.FileHeader) (uuid.UUID, error) {  1 usage  new *
    uploadDir := "./uploads"
    if err := os.MkdirAll(uploadDir, os.ModePerm); err != nil { return uuid.Nil, err }

    fileUUID := uuid.New()
    uniqueFileName := fileUUID.String() + "_" + file.Filename
    filePath := filepath.Join(uploadDir, uniqueFileName)

    src, err := file.Open()
    if err != nil { return uuid.Nil, err }
    defer src.Close()

    dst, err := os.Create(filePath)
    if err != nil { return uuid.Nil, err }
    defer dst.Close()

    if _, err := io.Copy(dst, src); err != nil { return uuid.Nil, err }

    return fileUUID, nil
}

func (s *fileService) GetFileByUUID(fileUUID uuid.UUID) (string, error) {  1 usage  new *
    uploadDir := "./uploads"
    files, err := os.ReadDir(uploadDir)
    if err != nil { return "", err }

    for _, file := range files {
        if file.IsDir() {
            continue
        }
        if strings.HasPrefix(file.Name(), fileUUID.String()) { return filepath.Join(uploadDir, file.Name()), nil }
    }

    return "", errors.New( text: "file not found")
}
```

Рисунок 3.16 – file_service.go

Наступним розглянемо **job_service.go,** який представлено у Додатку А. Даний модуль відповідає за основну бізнес-логіку, пов’язану з обробкою вакансій на платформі. Він реалізує інтерфейс JobService і взаємодіє з репозиторієм вакансій (JobRepository) для здійснення CRUD-операцій та пошуку вакансій.

Основні функціональні можливості сервісу:

– PostJob – створює нову вакансію на основі переданих даних (назва, опис, компанія, розташування, зарплата, тощо). Також дозволяє вказувати навички, необхідні для цієї посади;

– FindJob – знаходить вакансію за унікальним ідентифікатором. Якщо вакансія не знайдена, повертає відповідну помилку;

– SearchJobs – реалізує логіку пошуку вакансій за ключовим, словом, фільтрами та з підтримкою пагінації;

– GetJobsByUser – повертає список вакансій, опублікованих конкретним користувачем, також із підтримкою пагінації;

– UpdateJob – дозволяє оновлювати інформацію про вакансію (наприклад, змінювати опис, зарплату, вимоги до навичок тощо);

– DeleteJob – видаляє вакансію за ID;

– ListAllJobs – повертає повний список усіх вакансій, які є у системі.

Цей сервіс є ключовим компонентом платформи, що дозволяє роботодавцям керувати своїми вакансіями, а здобувачам – шукати релевантні пропозиції роботи.

Наступним сервісом є **skill_service.go**, який зображено на рис. 3.17. Він реалізує логіку отримання переліку професійних навичок, доступних у системі та взаємодіє з репозиторієм SkillRepository для отримання даних з бази.

```go
package application

import (
    "context"
    "jobfinder-web/internal/domain"
)

Implement interface
type skillService struct {  2 usages  new *
    repo domain.SkillRepository
}

func NewSkillService(repo domain.SkillRepository) domain.SkillService { return &skillService{repo: repo} }

func (s *skillService) GetAllSkills(ctx context.Context) ([]domain.Skill, error) {  1 usage  new *
    skills, err := s.repo.GetAllSkills(ctx)
    if err != nil { return nil, err }
    result := make([]domain.Skill, len(skills))
    for i, skill := range skills {
        result[i] = *skill
    }
    return result, nil
}
```

Рисунок 3.17 – skill_service.go

Далі переглянемо сервіс **user_service.go,** який представлено у Додатку А. Він відповідає за обробку бізнес-логіки, пов’язаної з управлінням користувачами в онлайн-сервісі та реалізує інтерфейс UserService і взаємодіє з репозиторієм UserRepository.

Основні функції:

– RegisterUser – реєстрація нового користувача, включає перевірку даних, хешування пароля за допомогою bcrypt і збереження даних у базу.

– LoginUser – автентифікація користувача шляхом перевірки email та пароля.

– GetUserByID – отримання користувача за його унікальним ідентифікатором.

– UpdateUser – оновлення профілю користувача, включаючи навички, досвід, зарплатні очікування тощо.

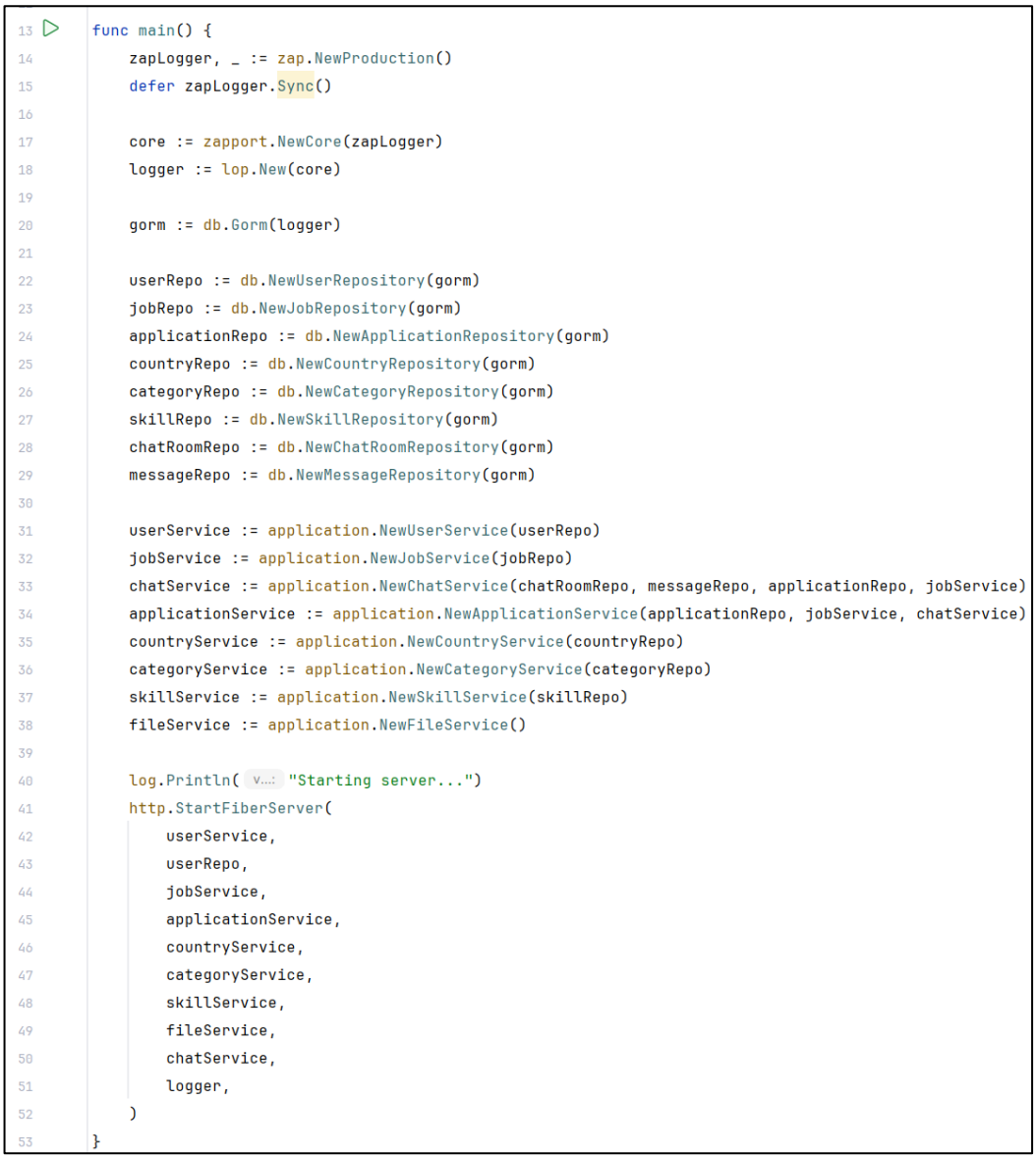
– DeleteUser – видалення користувача із системи.

– ListUsers – отримання списку всіх користувачів.

Цей сервіс забезпечує централізовану логіку обробки користувацьких даних, що є критично важливою частиною будь-якої системи з реєстрацією та авторизацією користувачів.

Під час розробки було створено також **middleware**, який реалізує логіку авторизації користувача на основі JWT-токенів, що написаний на Go з використанням фреймворку Fiber. Важливим компонентом системи є механізм авторизації на основі JWT (JSON Web Token). Це дозволяє надійно та безпечно ідентифікувати користувачів і контролювати доступ до ресурсів. У проєкті реалізовано генерацію токенів при вході в систему та middleware для перевірки авторизації при кожному запиті до захищених ендпоінтів. У коді серед основних функцій модуля є генерація токенів, а саме Access Token з коротким строком дії до 15 хвилин та Refresh Token з довшим строком дії до 7 днів, також реалізовано перевірку токену за допомогою HMAC-алгоритму та Middleware, який перехоплює кожен запит і перевіряє його.

І найголовніший файл з backend частини є **main.go,** який зображено на рис. 3.18. Цей файл є головним модулем застосунку, в якому здійснюється запуск сервісу, ініціалізація репозиторіїв, сервісів, логування та старт вебсервера. Його структура відповідає архітектурі типу Clean Architecture, що передбачає розділення відповідальності між рівнями: інфраструктури, домену, застосунку та інтерфейсів.



```go
func main() {
    zapLogger, _ := zap.NewProduction()
    defer zapLogger.Sync()

    core := zapport.NewCore(zapLogger)
    logger := lop.New(core)

    gorm := db.Gorm(logger)

    userRepo := db.NewUserRepository(gorm)
    jobRepo := db.NewJobRepository(gorm)
    applicationRepo := db.NewApplicationRepository(gorm)
    countryRepo := db.NewCountryRepository(gorm)
    categoryRepo := db.NewCategoryRepository(gorm)
    skillRepo := db.NewSkillRepository(gorm)
    chatRoomRepo := db.NewChatRoomRepository(gorm)
    messageRepo := db.NewMessageRepository(gorm)

    userService := application.NewUserService(userRepo)
    jobService := application.NewJobService(jobRepo)
    chatService := application.NewChatService(chatRoomRepo, messageRepo, applicationRepo, jobService)
    applicationService := application.NewApplicationService(applicationRepo, jobService, chatService)
    countryService := application.NewCountryService(countryRepo)
    categoryService := application.NewCategoryService(categoryRepo)
    skillService := application.NewSkillService(skillRepo)
    fileService := application.NewFileService()

    log.Println( v...: "Starting server...")
    http.StartFiberServer(
        userService,
        userRepo,
        jobService,
        applicationService,
        countryService,
        categoryService,
        skillService,
        fileService,
        chatService,
        logger,
    )
}
```

Рисунок 3.18 – main.go

Основні етапи ініціалізації:

– налаштування логера для якого використовується бібліотека zap та адаптер lop, що забезпечує структуроване логування;

– ініціалізація бази даних з допомогою функції db.Gorm(logger) підключається ORM-бібліотека GORM для роботи з PostgreSQL. Далі ініціалізуються всі необхідні репозиторії;

– створення сервісів;

– запуск вебсервера (HTTP API) через функцію http.StartFiberServer(...), у яку передаються всі сервіси, логер та інші залежності.

Основна рол цього файлу – весь код об'єднує компоненти системи, логіка не залежить від конкретних реалізацій (репозиторіїв, логера), що спрощує масштабування й тестування, а також логер дозволяє записувати події в консоль або файл у структурованому форматі (JSON), що важливо для моніторингу та налагодження.

3.3.2 Frontend частина

Клієнтська частина сайту реалізована за допомогою бібліотеки React, що дозволяє створювати динамічний та зручний для користувача інтерфейс у вигляді SPA. Основна мета клієнтської частини – забезпечення доступу до ключових функцій онлайн-сервісу: реєстрації, входу, перегляду вакансій, подання заявок, створення резюме тощо.

Архітектура frontend побудована на основі компонентного підходу. Кожна окрема сторінка або функціональний блок реалізована у вигляді самостійного компонента, який відповідає за свій шматок логіки та інтерфейсу. Це полегшує супровід коду, тестування та повторне використання.

Почнемо з компонента **layout**, який зображено на рис. 3.19. Цей компонент є кореневим компонуванням інтерфейсу, який визначає базову структуру HTML-документа для всіх сторінок застосунку. Він використовується у фреймворку Next.js для того, щоб одноразово задати спільні стилі, шрифти, теми оформлення та глобальні провайдери контексту, які будуть доступні на всіх сторінках застосунку.

Рисунок 3.19 – Layout.tsx

Наступною переглянемо сторінку **UnauthorizedPage**, яка зображена на рис. 3.20. Ця сторінка відображається, коли користувач намагається перейти до забороненого розділу без належної авторизації. Вона підключає глобальний стан автентифікації через useAuthStore, що дозволяє визначити, чи авторизований користувач і якщо користувач не автентифікований, його автоматично перенаправляє на сторінку входу (/login). Цей підхід дозволяє покращити безпеку інтерфейсу, запобігаючи несанкціонованому доступу та робить обробку помилок доступу інтуїтивно зрозумілою та естетично оформленою.

```tsx
export default function UnauthorizedPage() : Element { Show usages
  const router : AppRouterInstance = useRouter()
  const { user } = useAuthStore()

  useEffect(() : void => {
    if (!user) {
      router.push( href: "/login")
    }
  }, [user, router])

  const handleGoBack : () => void = () : void => { Show usages
    router.back()
  }

  const handleGoHome : () => void = () : void => { Show usages
    router.push( href: "/")
  }

  return (
    <div className="flex min-h-screen flex-col items-center justify-center">
      <div className="mx-auto max-w-md text-center">
        <h1 className="text-4xl font-bold">Access Denied</h1>
        <p className="mt-4 text-lg text-muted-foreground">You don&apos;t have permission to access this page.</p>
        <div className="mt-8 flex justify-center space-x-4">
          <Button onClick={handleGoBack} variant="outline">
            Go Back
          </Button>
          <Button onClick={handleGoHome}>Go Home</Button>
        </div>
      </div>
    </div>
```

Рисунок 3.20 – Unauthorized/Page.tsx

Далі переглянемо **main/page**. Ця сторінка є головною сторінкою користувацького інтерфейсу. На головній сторінці онлайн-сервісу реалізовано привітний інтерфейс, який знайомить користувача з можливостями сервісу та спонукає до першої взаємодії (рис. 3.21 – 3.22). Ця сторінка реалізована за допомогою React-компоненту з використанням бібліотеки Next.js і стилізована за допомогою Tailwind CSS.

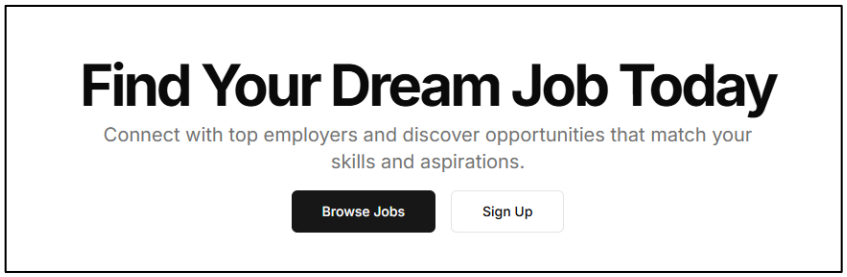


Рисунок 3.21 – Hero Section

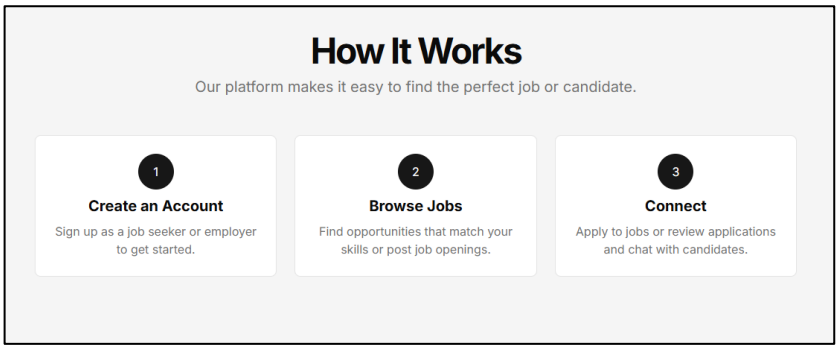


Рисунок 3.22 – Секція How It Works

Далі переглянемо файл, який відповідає за основний макет сторінок **Main Layout** див. рис. 3.23. Усі сторінки вебплатформи розгортаються всередині спільного компонента-макета MainLayout, який визначає єдину структуру та стиль для всього додатку. Це дозволяє досягти послідовності відображення елементів на всіх сторінках, спрощує підтримку дизайну та забезпечує зручність для користувача.

```
export default function MainLayout({  no usages
  children,
}: {
  children: React.ReactNode
}) : Element  {
  return (
    <div className="flex min-h-screen flex-col">
      <Navigation />
      <main className="flex-1 w-full mx-auto">
        <div className="max-w-screen-xl mx-auto px-4 sm:px-6 w-full">{children}</div>
      </main>
      <footer className="border-t py-6">
        <div className="container max-w-screen-xl mx-auto px-4 sm:px-6 flex flex-col items-center justify-between gap-4 md:h-14 md:flex-row">
          <p className="text-center text-sm leading-loose text-muted-foreground md:text-left">
            © {new Date().getFullYear()} JobFinder. All rights reserved.
          </p>
        </div>
      </footer>
    </div>
```

Рисунок 3.23 – MainLayout

Компонент MainLayout реалізовано за допомогою React та включає такі основні частини:

– навігаційна панель (<Navigation />) – фіксований заголовок сторінки, який надає швидкий доступ до основних розділів платформи;

– головна частина сторінки (<main>) – центральна зона, в якій динамічно відображається вміст залежно від маршруту;

– підвал (<footer>) – містить авторські права та загальну інформацію.

Макет використовує стилізацію через Tailwind CSS, зокрема:

– min-h-screen – для забезпечення повної висоти сторінки;

– max-w-screen-xl – обмеження ширини контенту для зручного читання;

– px-4, sm:px-6 – адаптивні відступи для різних екранів.

Завдяки застосуванню компонента MainLayout, надається можливість створювати сторінки, автоматично отримуючи єдину структуру з шапкою, контентом і футером, без дублювання коду. Це підвищує масштабованість та підтримуваність проєкту.

Наступна сторінка **Applications/Page.tsx** реалізує інтерфейс перегляду заявок користувача на роботу. Вона є частиною особистого кабінету здобувача та виконує такі функції:

– завантаження всіх поданих заявок авторизованого користувача;

– для кожної заявки виконується запит на отримання детальної інформації про відповідну вакансію;

– виводиться статус заявки (наприклад, “прийнята”, “відхилена”), дата подачі та базова інформація про компанію;

– якщо для заявки є непрочитані повідомлення у чаті, виводиться відповідний індикатор;

– користувач має змогу перейти до перегляду вакансії або відкрити повну інформацію по заявці, включно з чат-перепискою;

– додано можливість відкликати подану заявку через діалогове вікно підтвердження.

Таким чином, реалізована зручна система управління поданими заявками, яка дозволяє здобувачу оперативно взаємодіяти з потенційними роботодавцями, контролювати статус заявок і підтримувати спілкування.

У цій же теці створено файл **Applications/id/Page.tsx,** який реалізує функціонал, що дозволяє здобувачу переглянути деталі своєї заявки на конкретну

вакансію та вести комунікацію з роботодавцем у вигляді чату. Сторінка містить дві основні секції: інформація про заявку та чат з роботодавцем. Ці секції відповідають за виведення такої інформації, як: назва вакансії, компанії та локація, текст опис, функціональні кнопки, дати заявок, статус заявки на вакансію, відображення історії повідомлень, оновлення інтерфейсу при відправці нового повідомлення.

Наступною розберемо розробку частини **Job**, а саме **Apply/Page.tsx.** Це сторінка подачі заявки, яка реалізує інтерфейс, який дозволяє авторизованому користувачу (здобувачу) подати заявку на обрану вакансію.

Основна функціональність включає:

– завантаження резюме у форматі .pdf, .doc, .docx із перевіркою та відображенням статусу завантаження;

– введення тексту супровідного листа (cover letter);

– подача заявки, що включає передачу ідентифікатора вакансії, файлу резюме та супровідного листа через API-запит;

– візуальний зворотний зв’язок про статус завантаження/відправки (через анімацію та повідомлення Toast);

– компонент також обробляє винятки, які можуть виникнути під час завантаження файлу або відправлення заявки, і повідомляє користувача про можливі помилки;

– реалізована перевірка прав доступу – сторінка доступна лише для авторизованих користувачів з роллю здобувач.

Загалом ця частина застосунку забезпечує ключову взаємодію користувача з платформою, дозволяючи ефективно подавати документи до вакансій.

У цій же частині **Job** переглянемо сторінку пошуку вакансій **Page.tsx.** Даний компонент відповідає за динамічний пошук, фільтрацію та пагінацію вакансій, що зберігаються на сервері.

Цей компонент реалізує такі функціональні можливості:

– після завантаження сторінки виконується запит до API для завантаження доступних навичок, які потім відображаються у вигляді інтерактивних "бейджів", які можна використовувати для фільтрації вакансій;

– за допомогою текстового поля користувач може здійснити пошук вакансій за ключовими словами, який оновлюється динамічно;

– користувач може обрати одну або декілька навичок, щоб обмежити результати лише тими вакансіями, які містять відповідні вимоги;

– виведення результатів реалізовано з підтримкою сторінкової навігації, де користувач може переходити між сторінками результатів, при цьому параметри зберігаються в URL;

– компонент зчитує параметри пошуку та сторінки з адресного рядка й оновлює їх під час взаємодії, що забезпечує зручну навігацію та можливість ділитися посиланнями;

– компонент обгорнуто у ProtectedRoute, що обмежує доступ лише авторизованим користувачам;

– інтерфейс користувача реалізовано з використанням готових UI-компонентів (Card, Button, Input, Badge тощо), що забезпечує сучасний вигляд та зручну взаємодію.

Таким чином, цей компонент забезпечує повноцінний функціонал для взаємодії користувача з базою вакансій через інтуїтивно зрозумілий графічний інтерфейс.

Також однією з важливих сторінок є **LoginPage,** яка забезпечує функціонал авторизації користувачів через введення електронної пошти та пароля. Вона підтримує введення електронної пошти та паролю у відповідні обов’язкові поля, відправлення форми для асинхронного запиту, щоб перевірити правильність введених даних та підтримує перенаправлення на потрібну сторінку при успішній авторизації. Вся логіка написана з використанням React hooks (useState, useRouter, useSearchParams) відповідно до парадигми функціональних компонентів.

Наступною не менш важливою сторінкою є профіль користувача (**ProfilePage**). Він забезпечує перегляд та редагування особистих даних користувача. Цей функціонал є важливим елементом системи, оскільки дозволяє користувачам зберігати актуальну інформацію про свій досвід, навички та побажання щодо роботи. Після відкриття сторінки виконується асинхронне

завантаження інформації про користувача (через API), а також довідкових даних – переліку країн, рівнів володіння англійською мовою та доступних навичок. Користувач може відкрити частину редагування, у яку автоматично заповняться дані для зручного коригування. Після заповнення форми дані відправляються на сервер через API-запит для оновлення профілю. Успішне збереження підтверджується оновленням даних на сторінці. Якщо користувач не знаходиться у режимі редагування, тоді він бачить поточну інформацію у зручному форматі з розбиттям на категорії (роль, досвід, навички тощо). Сторінка профілю є важливим елементом взаємодії користувача з платформою, адже на її основі формується персоналізований підбір вакансій.

Важливим компонентом було розроблено сторінку реєстрації користувача **RegisterPage**. Цей компонент надає користувачеві можливість створення нового облікового запису шляхом заповнення реєстраційної форми. Він є критично важливим для функціонування платформи, оскільки забезпечує початковий доступ до системи. При завантаженні сторінки надсилаються запити до API для отримання списку країн та рівнів володіння англійською мовою, необхідних для заповнення відповідних полів у формі. Також застосунок має поділ на ролі, які визначають подальший функціонал на платформі (розміщення вакансій або подання заявок). Після відправлення форми відбувається передача даних на сервер через API. У випадку успішної реєстрації користувач отримує повідомлення про успіх та автоматично перенаправляється на сторінку входу.

Цікавою сторінкою в онлайн-сервісі є сторінка налаштувань (**SettingsPage**). Вона дозволяє переглянути коротку інформацію про себе та змінити тему застосунку між світлою та темною за допомогою інтерактивного елемента Switch.

Далі переглянемо блок інтерфейсу управління вакансіями для роботодавців. У межах розробки клієнтської частини вебзастосунка було реалізовано набір сторінок для **роботодавців**, що дозволяє повноцінно керувати створеними вакансіями, переглядати заявки кандидатів, змінювати статус заявок і вести листування з кандидатами.

Сторінка **my-jobs/page.tsx** зі списком вакансій відповідає за такі функції як:

– виводить усі вакансії, опубліковані поточним роботодавцем;

– для кожної вакансії відображаються основні дані (назва, компанія, зарплата, навички, дата публікації);

– відображається кількість непрочитаних повідомлень по заявках на кожну вакансію.

Сторінка **my-jobs/page.tsx** має такі доступні дії як:

– перехід до перегляду;

– перегляд заявок;

– редагування вакансії;

– видалення вакансії через діалогове вікно.

Сторінка **my-jobs/[id]/edit/page.tsx** для редагування вакансій відповідає за такі функції як:

– завантажує та відображає дані вакансії у формі;

– дозволяє змінювати всю інформацію: назву, компанію, зарплату, місце роботи, тип зайнятості, навички, опис тощо;

– передбачено інтерактивне обрання навичок;

– після підтвердження зміни надсилаються на сервер через API.

Сторінка **my-jobs/[id]/applications/page.tsx** зі списком заявок на вакансії виводить перелік усіх заявок, поданих на конкретну роботу і показує, яка інформація міститься у кожній з них.

Сторінка **my-jobs/applications/[id]/page.tsx** відповідає за деталі заявки та чат. Ця частина має можливість вивести розгорнуту інформацію про кандидата, змінити статус заявки, має інтегрований чат, який відображається з правого боку, між кандидатом та роботодавцем, та надано можливість надсилання повідомлень та їх автоматичне оновлення у чаті.

Також було створено сторінку **PostJobPage**, яка реалізує функціонал створення вакансії через інтерфейс, доступний лише для авторизованих користувачів із роллю роботодавця (employer). Цей компонент є ключовою частиною функціоналу платформи, що дозволяє наповнювати систему новими пропозиціями роботи. Після ініціалізації сторінки надсилаються паралельні запити

до API для завантаження списку країн, доступних навичок та рівнів володіння англійською мовою. Користувач може заповнити форму з полями для створення нової вакансії та опублікувати її.

І останньою важливою сторінкою є **UserProfilePage**, яка реалізує функціональність перегляду профілю користувача (здобувача) з боку роботодавця. Цей інтерфейс є важливою частиною системи, що дає змогу роботодавцю ознайомитися з розширеною інформацією про кандидата перед прийняттям рішення щодо поданої заявки.

На цій сторінці реалізовані такі функції як:

– завантаження профілю за ID;

– відображення розширеної інформації, такі як ім’я, пошта, роль у системі, очікувана зарплатня, рівень англійської мови, досвід роботи та навички;

– можливість повернутись назад за допомогою кнопки «Back».

Цей компонент є частиною бізнес-логіки роботодавця, що забезпечує прозору оцінку кандидатів на основі їх профілю, і відіграє ключову роль у прийнятті рішень щодо заявок.

3.4 Тестування розробленого сайту

У результаті виконання дипломного проєкту було реалізовано заплановану мету проєкту, тобто – розробку сучасного онлайн-сервісу пошуку роботи, який забезпечить зручний та швидкий доступ до вакансій, ефективну фільтрацію пропозицій, можливість створення резюме та подання заявок.

При першому вході на сайт ми з’являємося на стартовій сторінці, яка зображена на рис. 3.24, де відображено як працює сайт та присутні кнопки для реєстрації та авторизації на сайті, а також кнопка для переходу до перегляду вакансій. Якщо користувач не авторизований, тоді він не зможе переглядати вакансії та для нього буде висвічуватись сторінка авторизації, яка зображена на рис. 3.25.

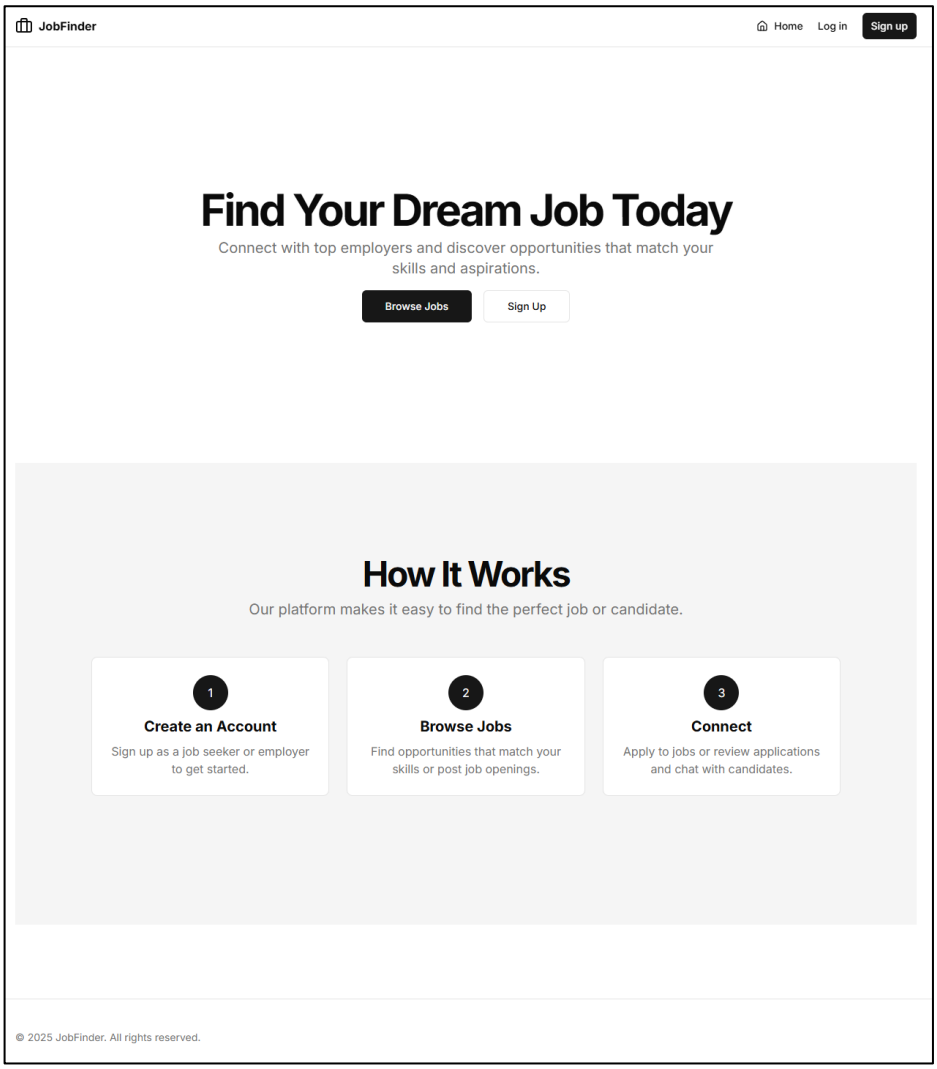


Рисунок 3.24 – Стартова сторінка

Користувач може увійти у свій акаунт за допомогою логіну (електронної пошти) та свого пароля (рис. 3.34).

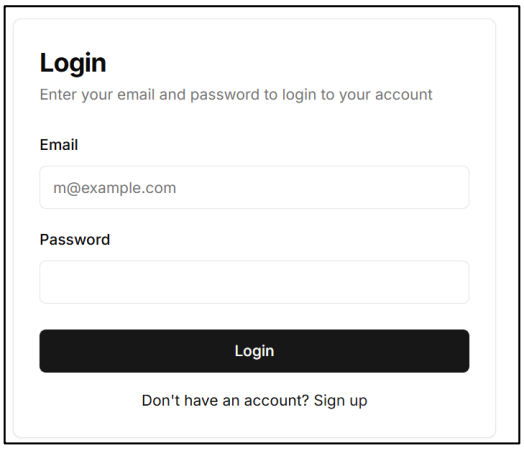


Рисунок 3.25 – Сторінка авторизації

Якщо у користувача немає створеного облікового запису, він може його створити натиснувши кнопку “Sign up”. Після її натискання буде з’являтись сторінка реєстрації на онлайн-сервісі (рис. 3.26). На цій сторінці користувач може ввести свої дані для створення акаунту, а саме ім’я та прізвище, електрону пошту, пароль, обрати роль на сайті (шукач роботи чи роботодавець), країну та рівень англійської мови. Після вводу усіх даних користувач може натиснути кнопку створення акаунту.

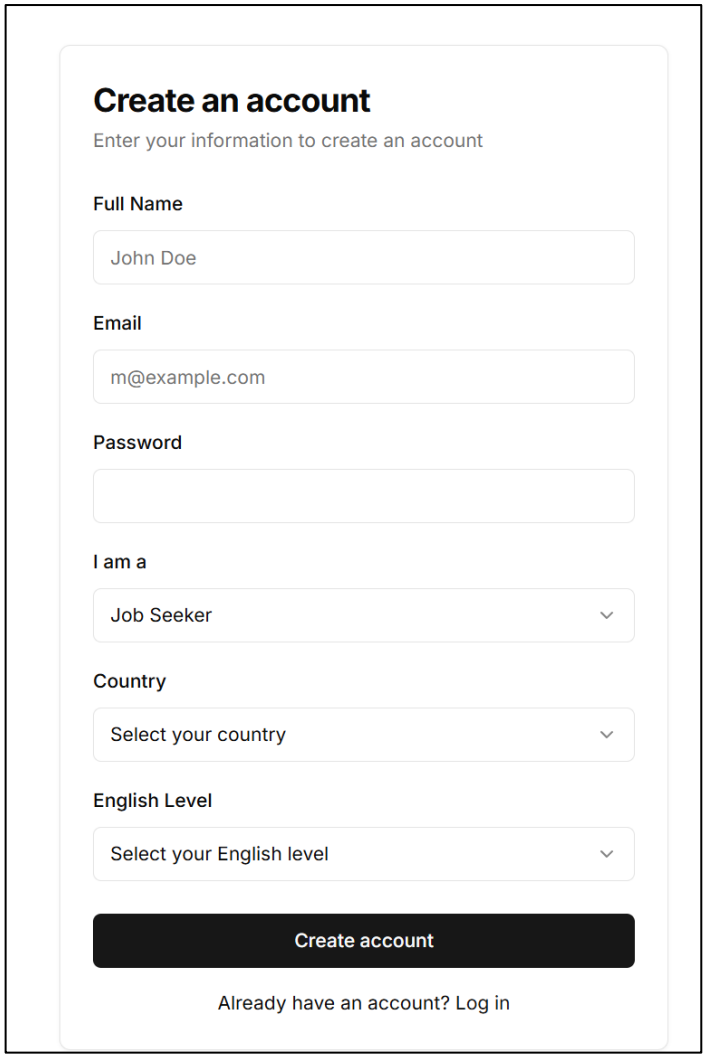


Рисунок 3.26 – Сторінка реєстрації

Оскільки розроблений вебсервіс підтримує два типи користувачів – шукачів роботи та роботодавців, тестування проводилося за кількома сценаріями з

урахуванням їх функціональних ролей. Мета тестування – перевірка коректності бізнес-логіки, валідації форм, обробки помилок, відображення відповідей від API та узгодженості даних в інтерфейсі.

Першим зробимо тестування реєстрації, якщо користувач введе неправильний тип даних для електронної пошти (рис. 3.27), це буде показувати перевірку на правильність валідації полів та відобразить повідомлення про помилку при некоректно введених даних.

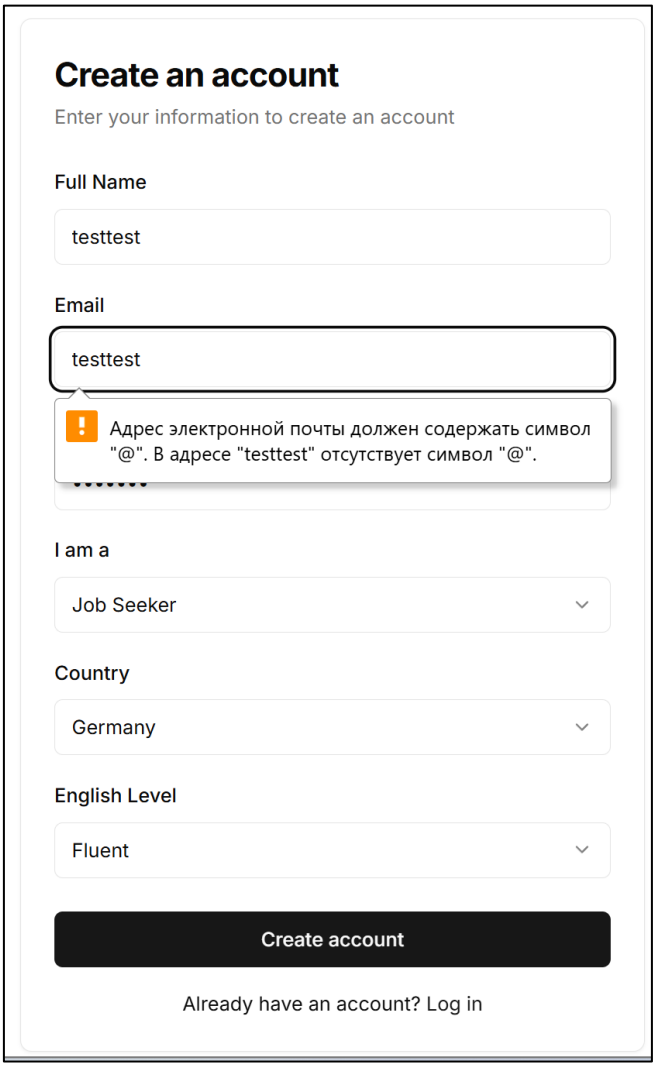


Рисунок 3.27 – Тестування реєстрації

Продивимось сценарій користування сайтом від шукача роботи. Першим є авторизація у свій акаунт та перегляд стартової сторінки (рис. 3.28).

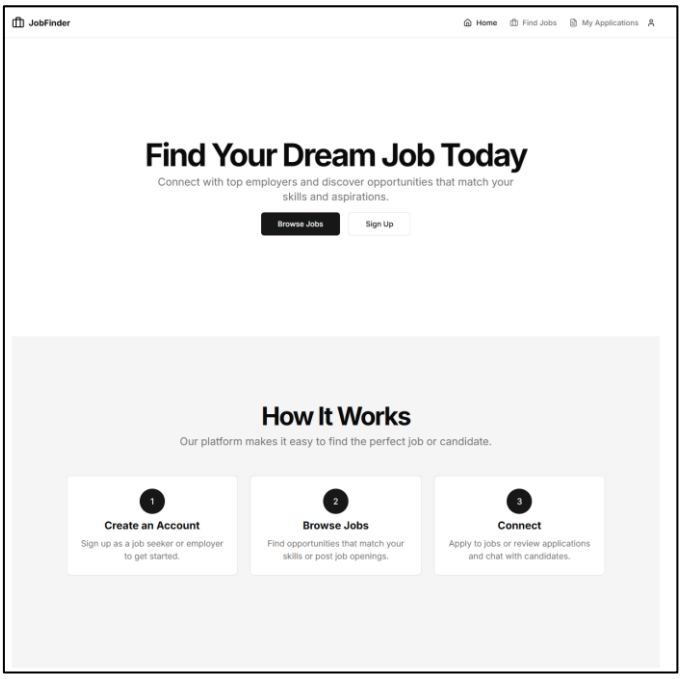


Рисунок 3.28 – Перегляд стартової сторінки (шукач роботи)

Наступним кроком переглянемо та протестуємо сторінку з пошуком вакансій (рис. 3.29), перевіримо як працює пошук за ключовими словами (рис. 3.30 – 3.31). та як працює фільтр навичок (рис. 3.32).

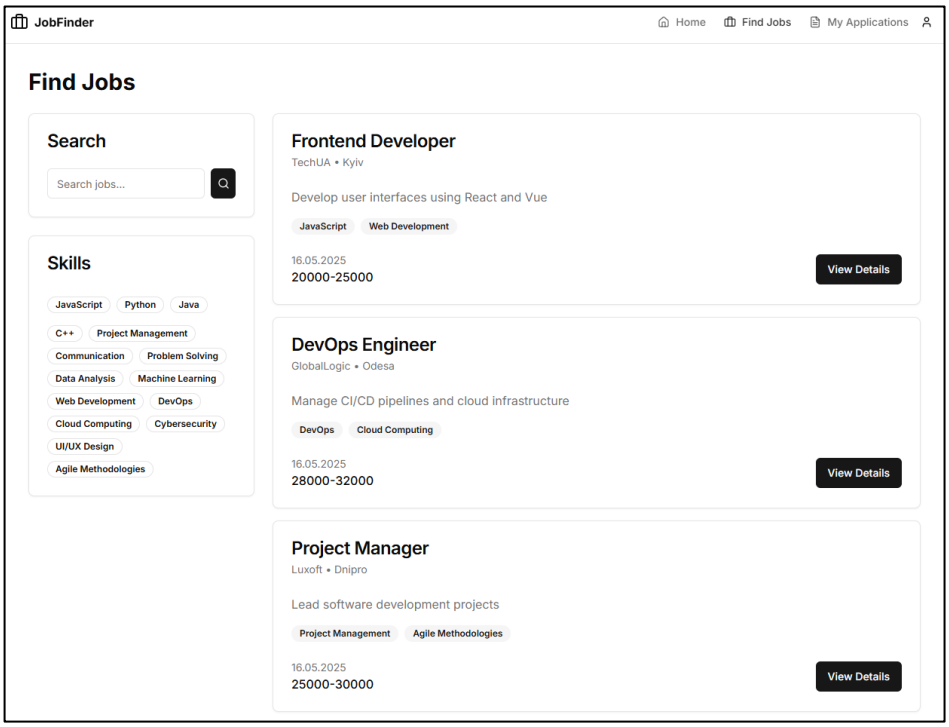


Рисунок 3.29 – Сторінка вакансій

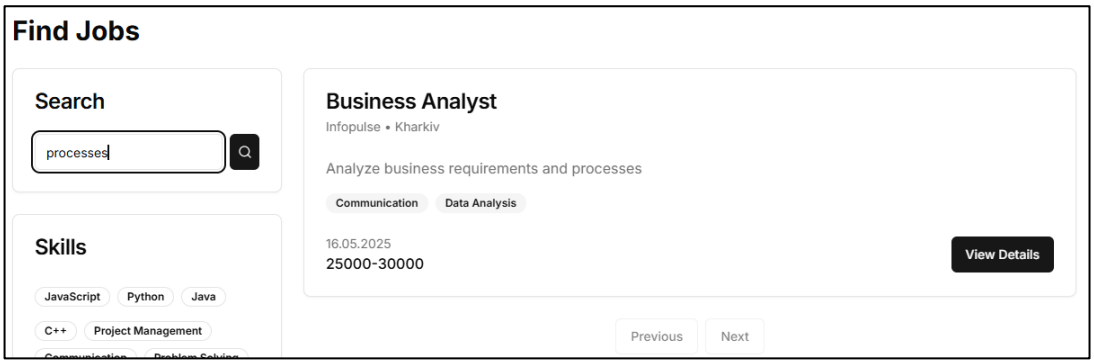


Рисунок 3.30 – Пошук вакансій (за описом)

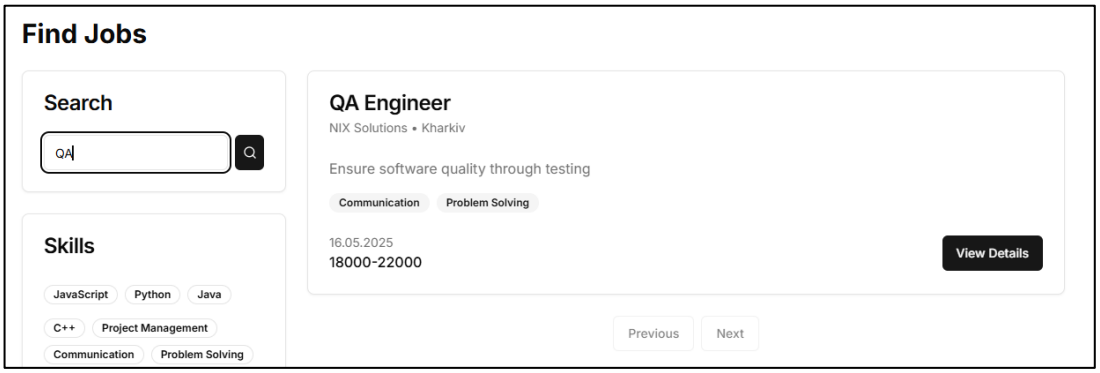


Рисунок 3.31 – Пошук вакансій (за назвою)

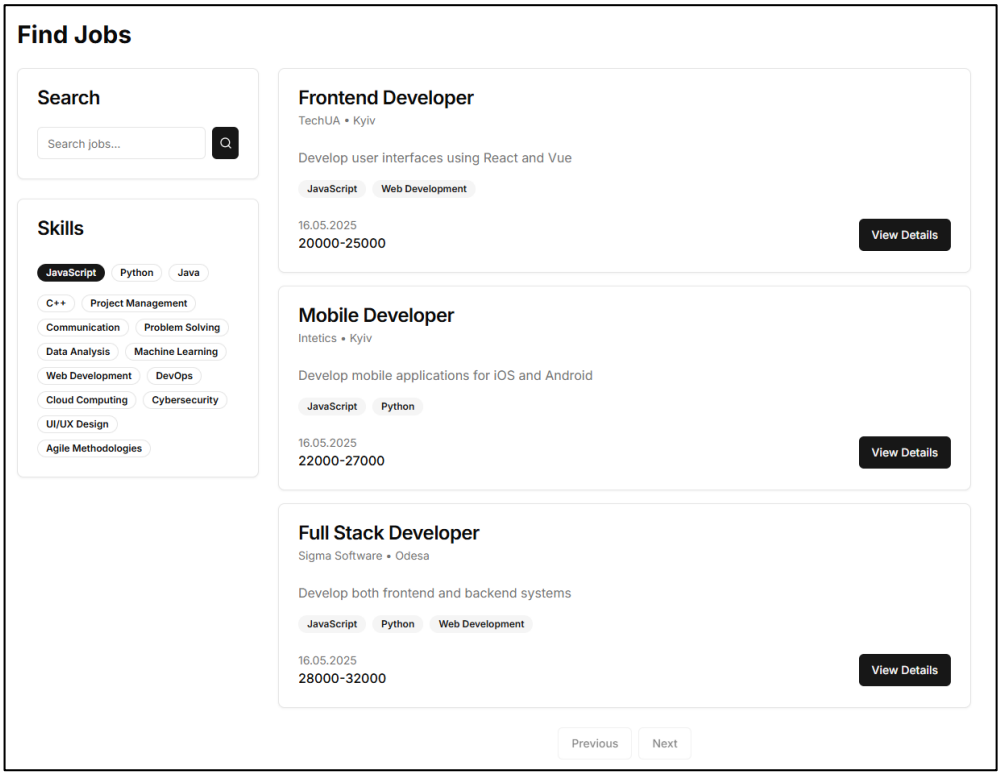


Рисунок 3.32 – Фільтрація вакансій

Також можливо переглянути детальну інформацію про вакансію натиснувши «View Details» (рис. 3.33). На сторінці перегляду вакансії можливо вийти назад, відгукнутись на вакансію та побачити опис, який включає у себе назву вакансії, назву компанії, опис вакансії, місто, коли було створено, заробітну плату, тип зайнятості, бажані навички та бажане знання англійської мови.

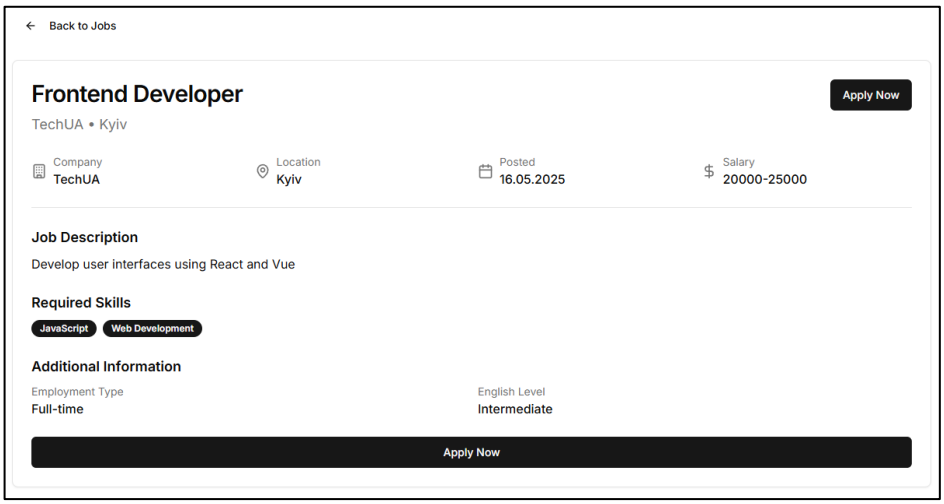


Рисунок 3.33 – Перегляд вакансії

Протестуємо можливий сценарій з акаунту шукача роботи, а саме подання заявки на вакансію і подальший її перегляд, спочатку перейдемо на сторінку формування відгуку на вакансію (рис. 3.34). У цьому прикладі кандидат подає заявку на позицію Frontend Developer у компанії TechUA (місце розташування – Київ).

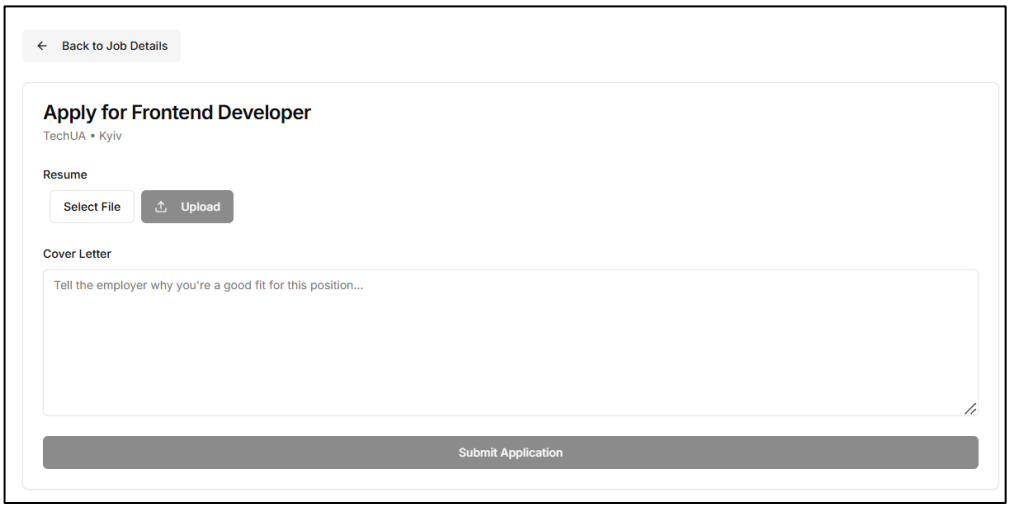


Рисунок 3.34 – Пуста форма подання відгуку на вакансію

На рис. 3.35 – 3.36 зображено перевірку чи є можливість завантажити файли з власним резюме потрібного формату на сторінку. Після цього користувач повинен натиснути кнопку «Upload», щоб завантажити свій документ до бази даних для збереження на онлайн-сервісі, а також написати супровідний лист для роботодавця для подання відгуку (рис. 3.37). Після успішного завантаження файлу відображається підтвердження: *"Resume uploaded successfully"*.

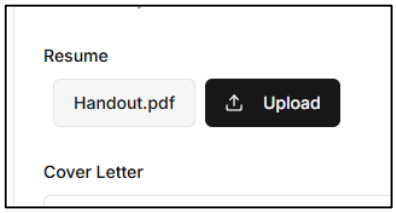


Рисунок 3.35 – Перевірка завантаження pdf файлів

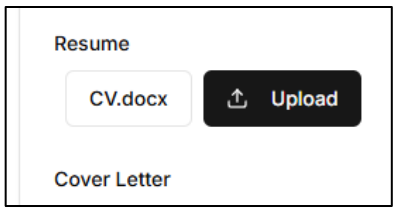


Рисунок 3.36 – Перевірка завантаження docx файлів

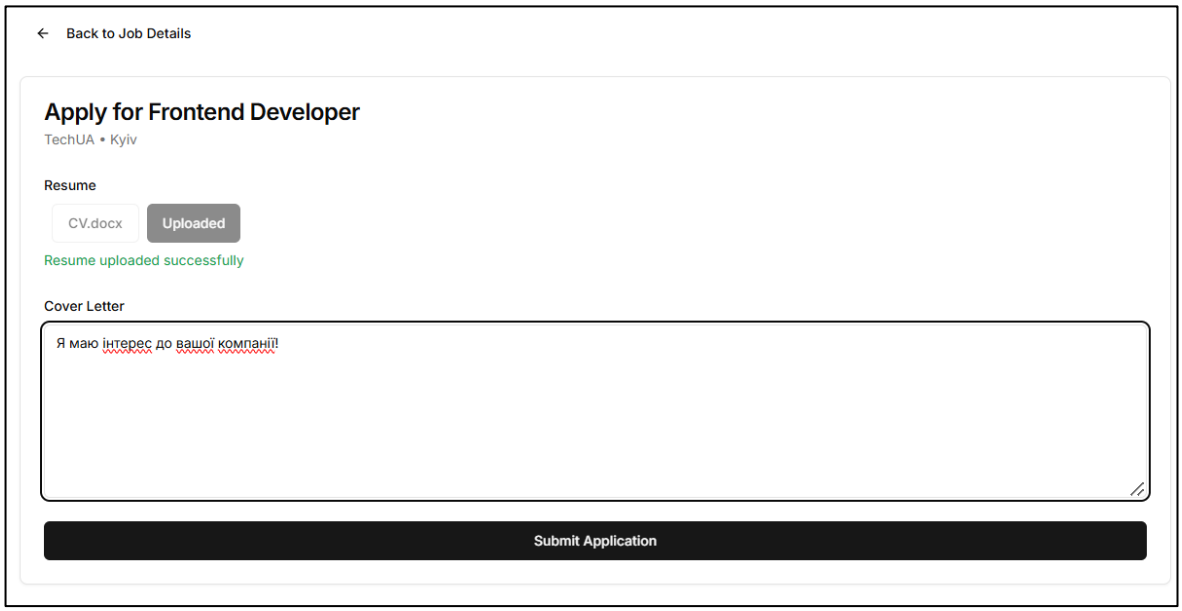


Рисунок 3.37 – Кінцеве формування відгуку

Даний сценарій тестування включає у себе перевірку валідності файлу резюме, перевірку обов’язкових полів, логування подій на профілі користувача. Зрештою, користувач після відправки форми перенаправляється на сторінку з відгуками та може продивитись свій відгук, продивитись роботу або видалити свій відгук (рис. 3.38).

Рисунок 3.38 – Сторінка відгуків (шукач роботи)

Продовжимо тестування розроблених функцій на акаунті шукача роботи. Наступним протестуємо функціонування перегляду вакансії зі сторінки відгуків (рис. 3.39), також протестуємо перегляд відгуку (рис. 3.40) і його видалення.

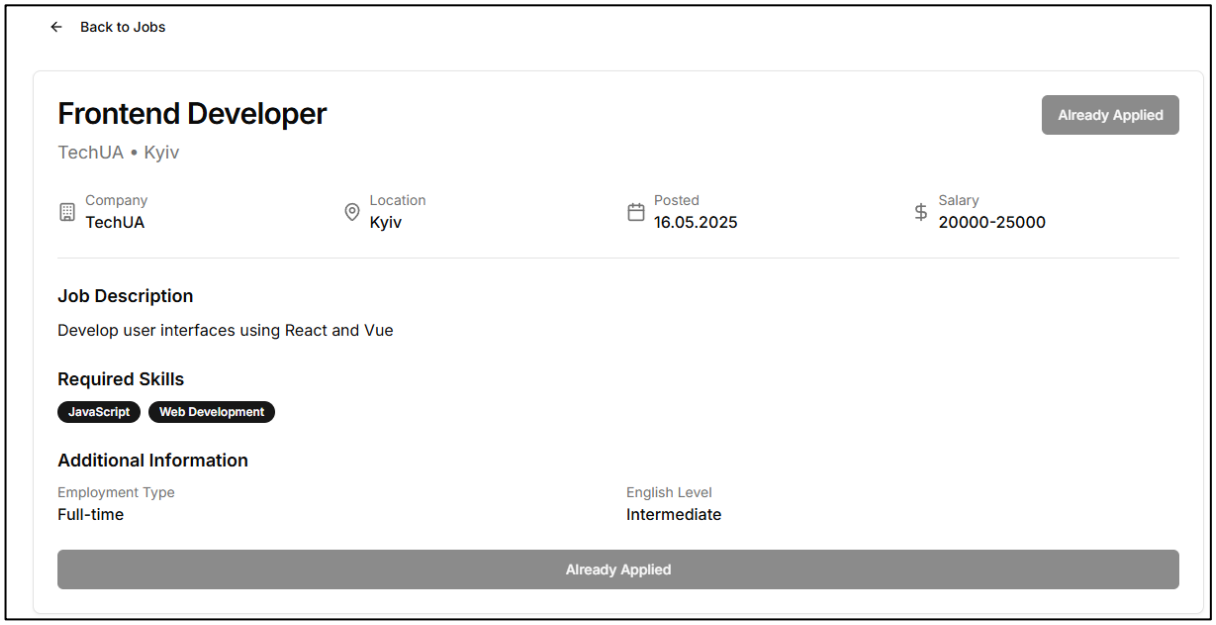


Рисунок 3.39 – Перегляд вакансії після відгуку

Як можна побачити на рис. 3.39 після відправлення відгуку кнопка відгуку міняє текст на «Already Applied», стає сірого кольору та неактивною. Це означає, що користувач уже подав заявку на цю вакансію, і повторне відправлення відгуку є неможливим. Наявність блокування повторної подачі підвищує зручність та виключає дублікати у базі даних.

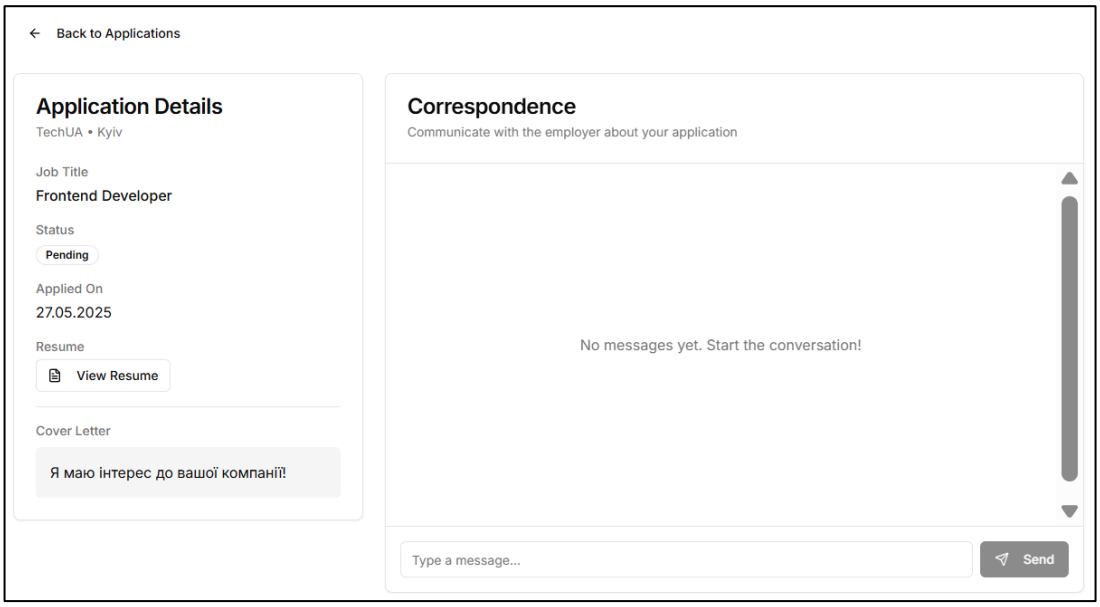


Рисунок 3.40 – Перегляд відгуку

На рис. 3.40 представлено сторінку перегляду детальної інформації про подану заявку з боку шукача роботи. У цьому прикладі користувач подав заявку на вакансію Frontend Developer у компанії TechUA та має можливість взаємодіяти з роботодавцем через вбудований чат. Ліва частина форми відповідає за опис деталей заявки, а права частина за листування з роботодавцем. Натиснувши на файл з резюме у лівій частині форми, він буде відкриватись окремо на пристрої. Розроблений чат працює в межах заявки, повідомлення надсилаються через API та зберігаються на сервері, відбувається періодичне оновлення списку повідомлень, а також повідомлення можуть бути прочитані та марковані як "прочитані", що синхронізується з системою сповіщень. Даний інтерфейс повинен забезпечити

зручну комунікацію між кандидатами та роботодавцями та дозволити уточнення деталей вакансії, графіку, очікувань тощо без сторонніх засобів зв’язку.

Перевіримо більш детально роботу чату. На рис. 3.41 зображено початковий стан чату, коли повідомлення між сторонами ще не надсилалися. Виводиться повідомлення “No messages yet. Start the conversation!”, що заохочує користувача до першого контакту. Також можна помітити, що кнопка відправлення повідомлення є неактивною до тих пір поки не буде написано якесь повідомлення у відповідну строку.

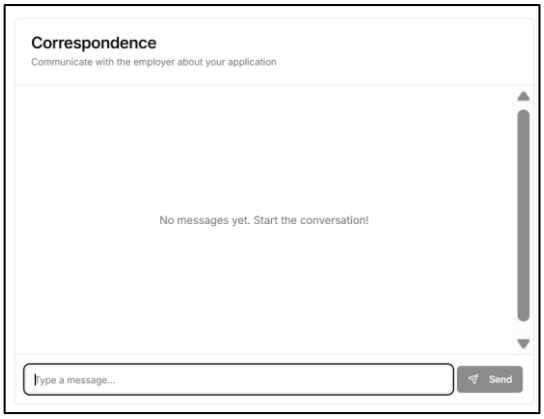


Рисунок 3.41 – Початковий стан чату

На рис. 3.42 користувач вводить перше повідомлення в текстовому полі перед натисканням кнопки Send. Це демонструє зручність введення тексту та інтуїтивно зрозумілий інтерфейс. Після вводу повідомлення кнопка відправки стає активною і користувач має можливість надіслати своє повідомлення.

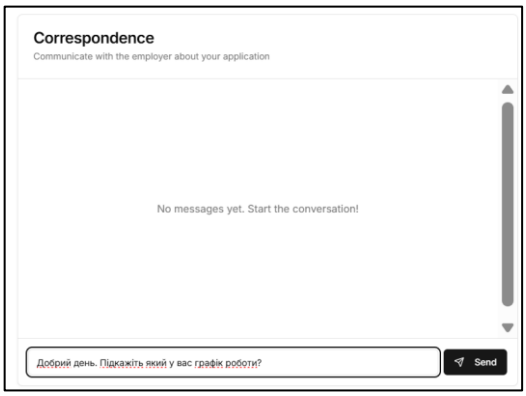


Рисунок 3.42 – Введення повідомлення

На рис. 3.43 можна переглянути обмін повідомленнями з написанням точного часу відправки. Повідомлення чітко розділені між двома користувачами. Повідомлення кандидата виділене чорним фоном праворуч, відповідь – сірим зліва разом з ім’ям відправника.

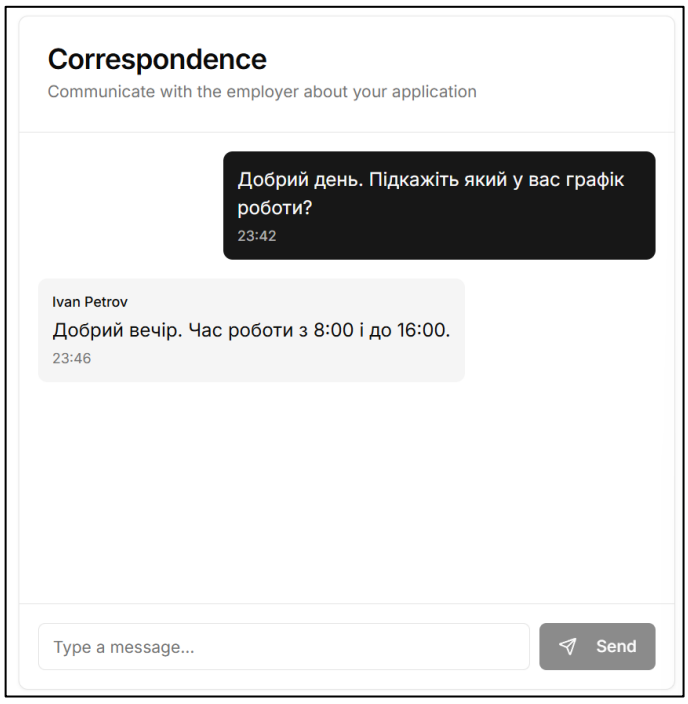


Рисунок 3.43 – Обмін повідомленнями

Також було розроблено сповіщення про нові повідомлення у вкладці My Applications у вигляді червоної крапки (рис. 3.44), а також напис 1 new навпроти відгуку (рис. 3.45), на який прийшло це повідомлення. Це вказує користувачеві на те, що з’явилося нове непрочитане повідомлення в одній із заявок, на які він відгукнувся.

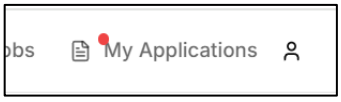


Рисунок 3.44 – Червоний індикатор повідомлень

Рисунок 3.45 – Індикатор нових повідомлень

На сторінці My Applications також видно загальний статус заявки (прийнято, відхиллено, очікується) та дату її подачі (рис. 3.45).

Тепер проведемо тестування онлайн-сервісу від облікового запису роботодавця. Роботодавець має окремий набір функцій, доступних після реєстрації та входу в систему.

Після авторизації користувач бачить стартову сторінку (рис. 3.46), яка дозволяє йому швидко перейти до публікації вакансій або перегляду вже створених. У центральній частині розміщено пояснення про те, як працює платформа. Роботодавець також може бачити вакансії свої та інші, як було представлено на рис. 3.29.

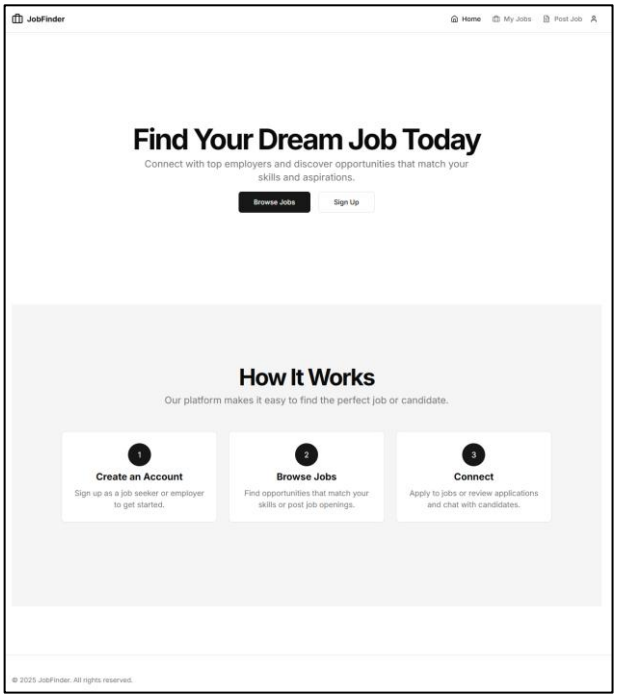


Рисунок 3.46 – Стартова сторінка (роботодавець)

Кожну вакансію можна детально переглянути, включаючи опис, вимоги до кандидатів, рівень англійської мови та тип зайнятості (рис. 3.47). Це дає змогу перевірити, як виглядає публікація для кандидатів.

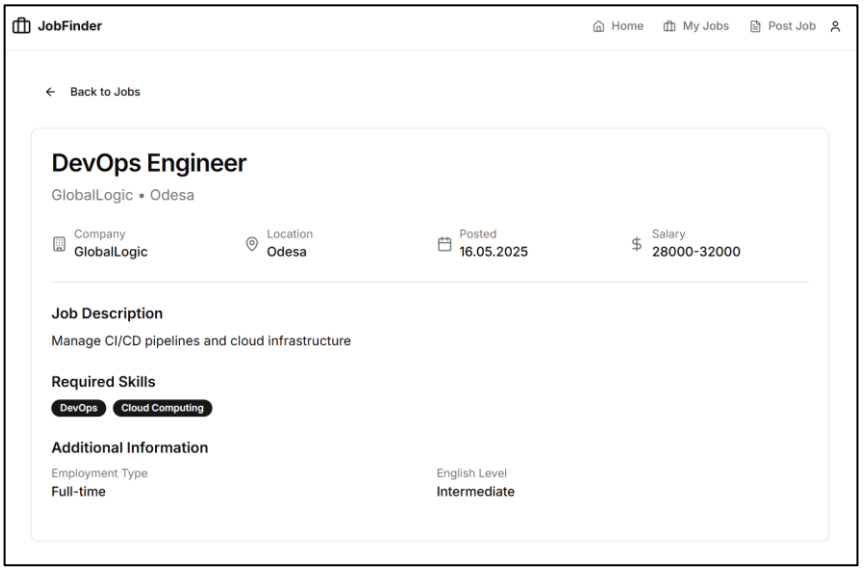


Рисунок 3.47 – Перегляд вакансій (роботодавець)

Роботодавець має змогу переглядати всі створені ним вакансії, редагувати або видаляти їх, а також переглядати список відгуків до кожної вакансії (рис. 3.48).

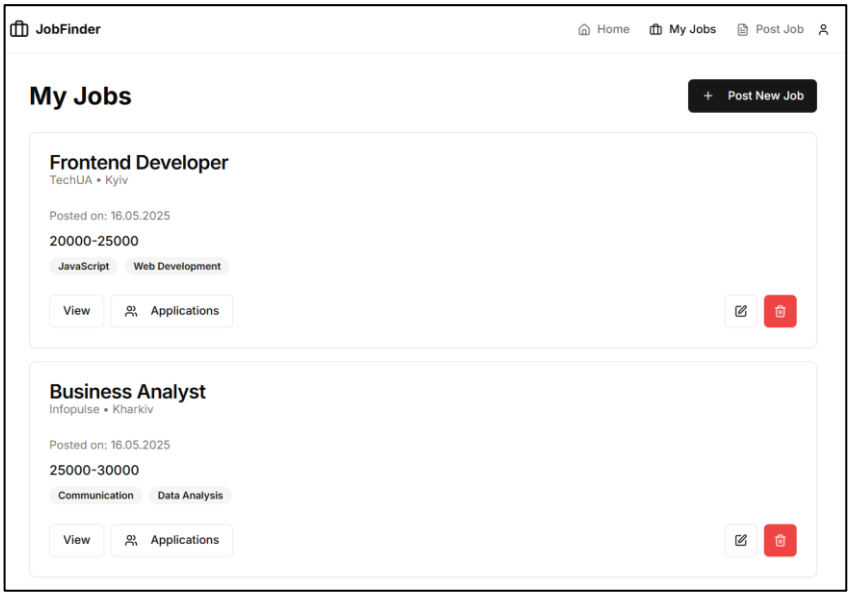


Рисунок 3.48 – Опубліковані роботи

Роботодавець також має можливість перегляду списку відгуків для кожної зі своїх вакансій (рис. 3.49). Якщо шукач роботи відправив свій відгук, роботодавець може його видалити, переглянути завантажене резюме, продивитись відгук разом з чатом, а також прийняти чи відхилити заявку на роботу.

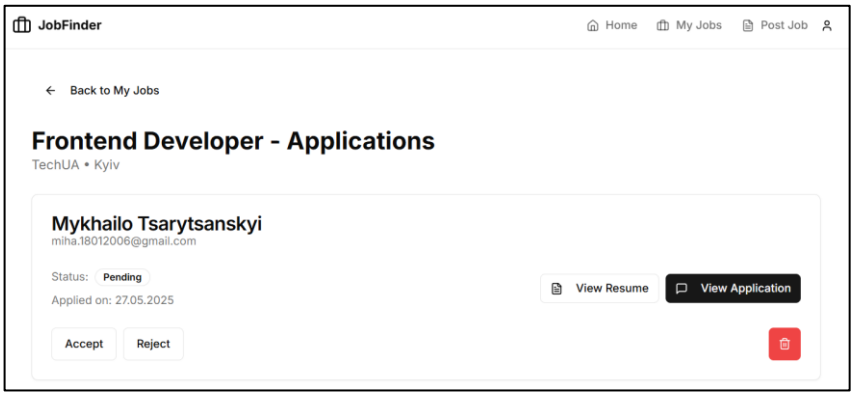


Рисунок 3.49 – Перегляд відгуків від роботодавця

Відкривши деталі заявки, можна переглянути контактну інформацію кандидата, його навички, надіслане резюме, супровідний лист, прийняти чи відхилити його заявку, а також вести листування безпосередньо вбудованим чатом (рис. 3.50).

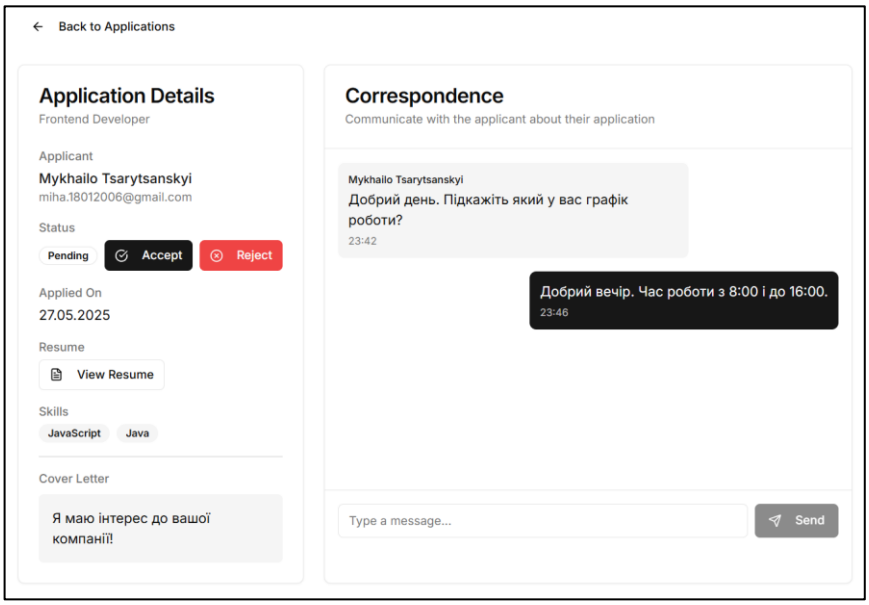


Рисунок 3.50 – Детальний перегляд відгуку

Вбудований чат дозволяє уточнювати деталі вакансії, графік або інші організаційні моменти без використання сторонніх месенджерів. Усі повідомлення зберігаються в рамках окремої заявки.

Переглянемо те як виглядає прийняття та відхилення заявки зі сторони роботодавця (рис. 3.51 – 3.52). Та продивимось як це відображається у профілі шукача роботи (рис. 3.53). У профілі шукача роботи буде просто змінюватись статус його заявки.

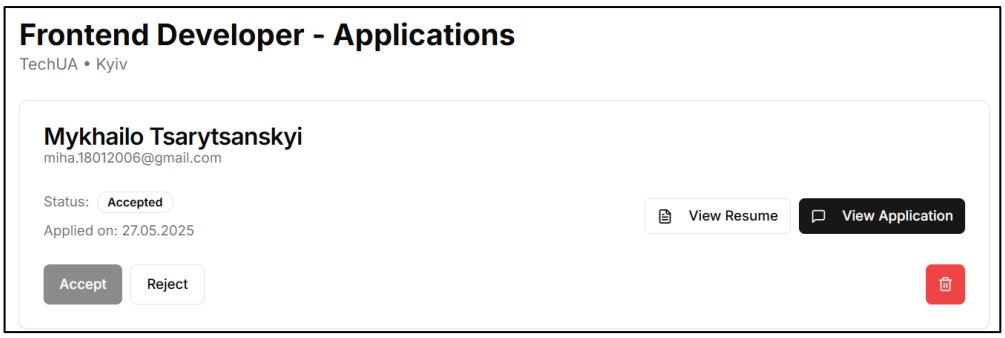


Рисунок 3.51 – Прийняття заявки

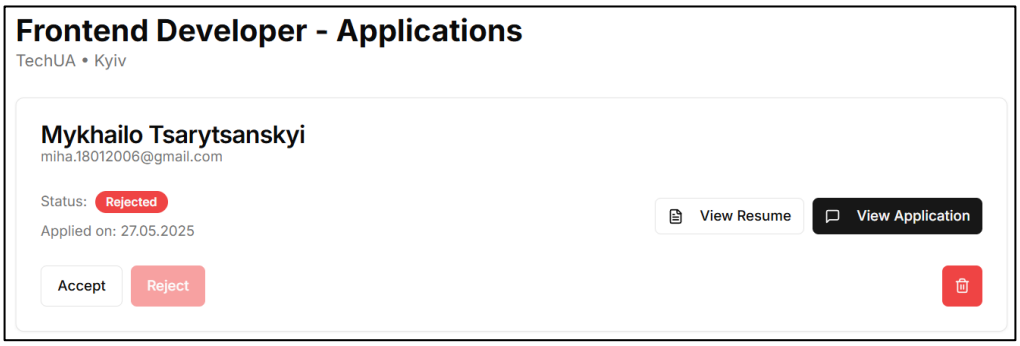


Рисунок 3.52 – Відхилення заявки

Рисунок 3.53 – Змінення статусу заявки (шукач роботи)

На рис. 3.54 зображено сторінку редагування вже створеної вакансії. Цей інтерфейс дозволяє роботодавцю змінювати всю важливу інформацію про вакансію: назву посади, назву компанії, місце розташування, діапазон заробітної плати, країну, необхідний рівень володіння англійською мовою, тип зайнятості, опис обов'язків, а також перелік необхідних навичок. Після внесення змін роботодавець натискає кнопку «Update Job», яка надсилає оновлені дані до API для збереження змін.

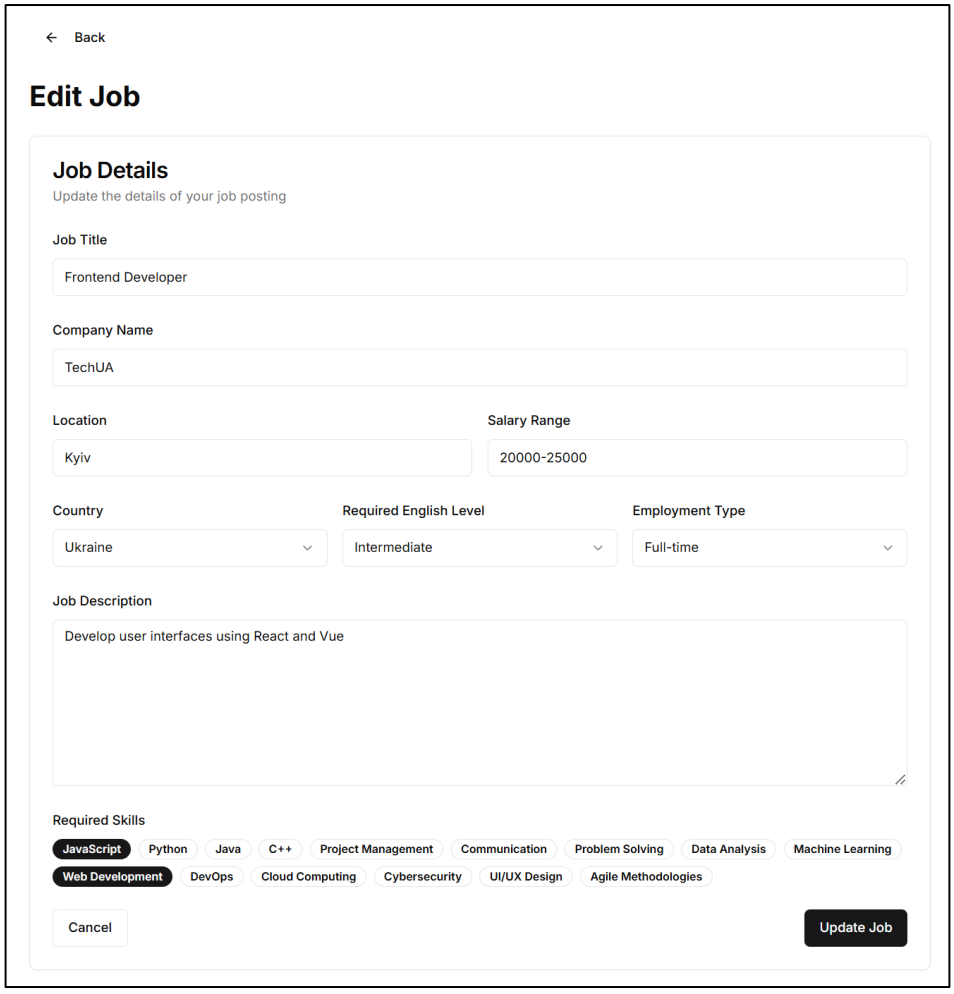


Рисунок 3.54 – Сторінка редагування вакансії

На рис. 3.55 представлено сторінку створення нової вакансії. Вона має аналогічну структуру до сторінки редагування, але призначена для введення нової інформації. Після заповнення всіх обов’язкових полів, таких як: назва посади,

компанія, місце, зарплатний діапазон, країна, англійська мова, тип зайнятості, опис та перелік навичок – користувач натискає кнопку «Post Job», яка ініціює запит до серверної частини застосунку для створення нової вакансії.

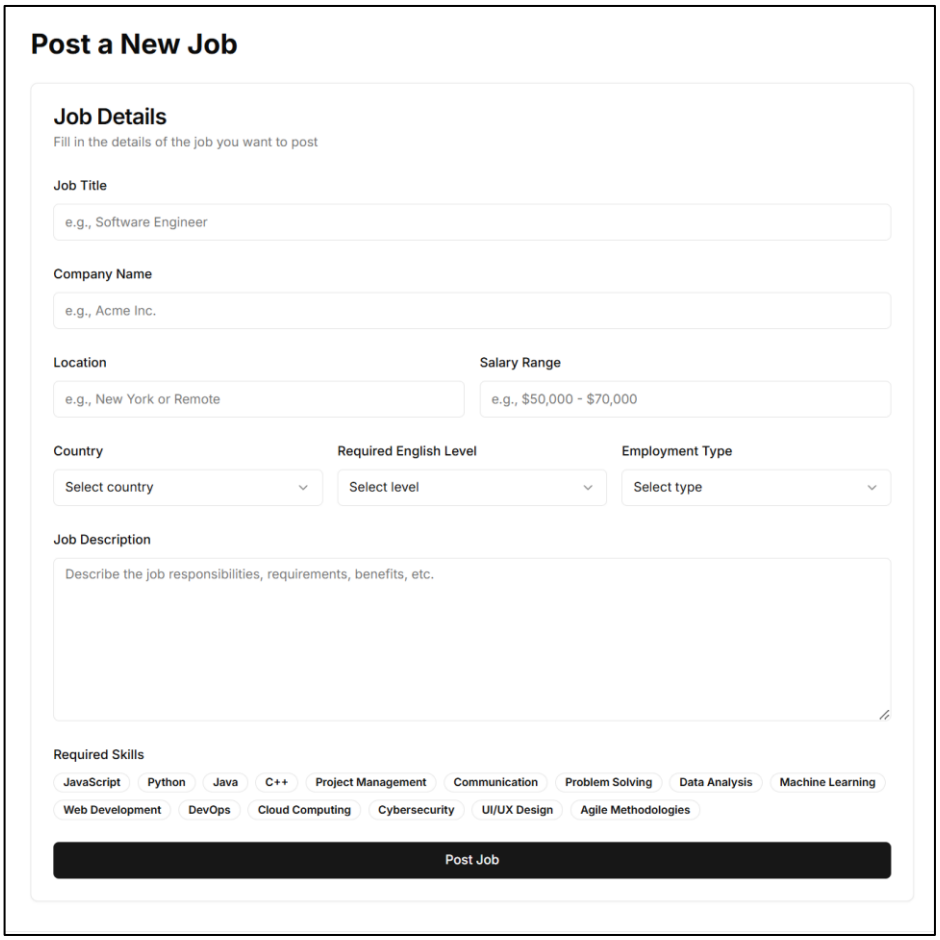


Рисунок 3.55 – Сторінка створення вакансії

На рис. 3.56 зображено сторінку помилки доступу. Вона з'являється в разі, якщо користувач намагається перейти на сторінку, яка не відповідає його ролі або правам доступу. Це дозволяє реалізувати контроль доступу на основі ролей (RBAC), що важливо для безпеки системи.

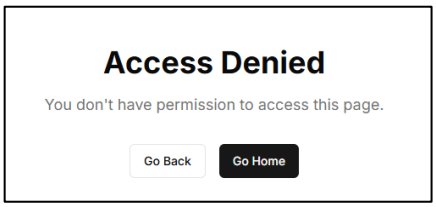


Рисунок 3.56 – Сторінка помилки

На рис. 3.57 показано випадаюче меню користувача. Звідси користувач може перейти до свого профілю, налаштувань або виконати вихід із системи.

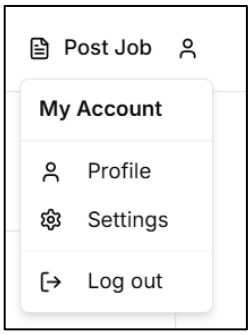


Рисунок 3.57 – Меню

Рис. 3.58 демонструє сторінку особистого профілю. На ній відображаються особисті дані користувача: ім’я, email, роль у системі (роботодавець або шукач роботи), кількість років досвіду, рівень англійської мови, очікувана заробітна плата, опис досвіду роботи та перелік навичок.

Рисунок 3.58 – Сторінка особистого профілю

Рис. 3.59 ілюструє сторінку редагування особистого профілю, де користувач має можливість оновити свої персональні дані, включно з досвідом, навичками та іншими важливими атрибутами. Це дозволяє підтримувати інформацію в актуальному стані.

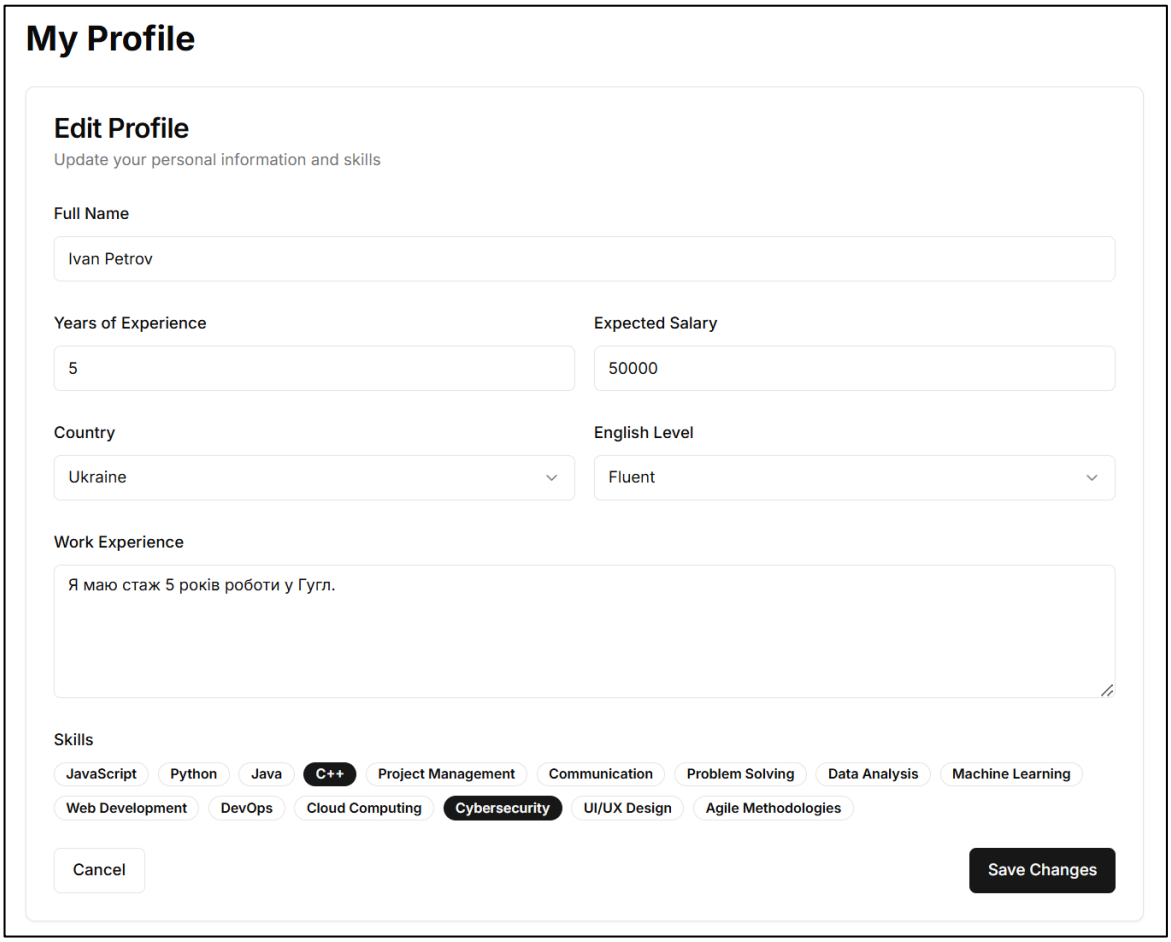


Рисунок 3.59 – Сторінка редагування профілю

Переглянемо, що входить во вкладки налаштувань. На рис. 3.60 зображено вкладку «Account» в розділі налаштувань. Тут відображається email користувача та його роль. Дана сторінка є інформативною та використовується для перегляду ключової інформації облікового запису.

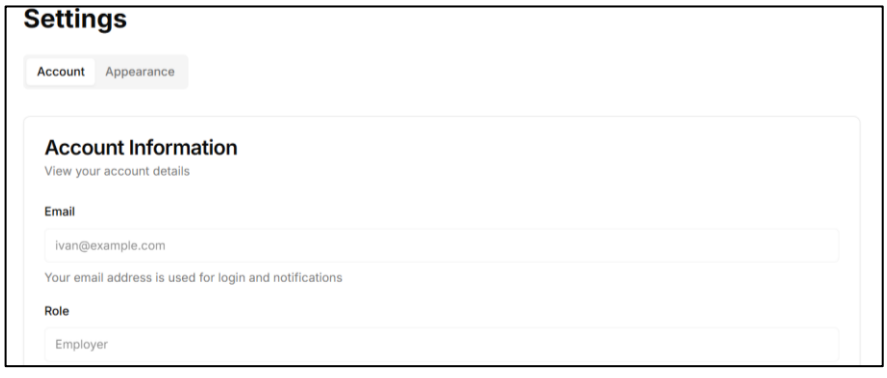


Рисунок 3.60 – Сторінка налаштувань (Акаунт)

На рис. 3.61 зображено вкладку «Appearance» у налаштуваннях, де користувач може переключити зовнішній вигляд додатку між світлою та темною темами.

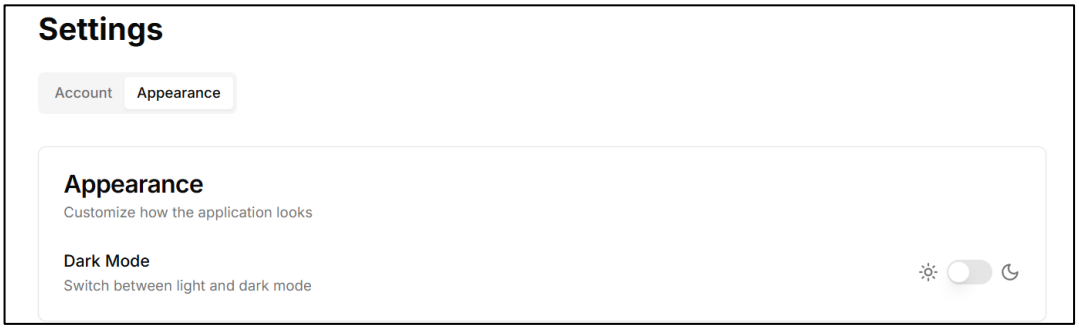


Рисунок 3.61 – Сторінка налаштувань (Вигляд)

Рис. 3.62 демонструє активовану темну тему після перемикання. Така функціональність дозволяє підвищити комфорт роботи користувача, особливо при низькому освітленні.

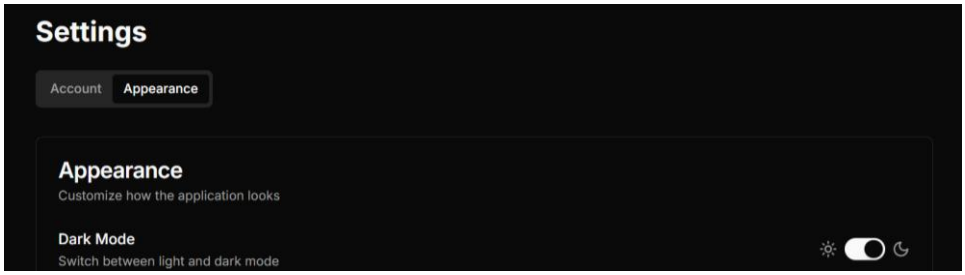


Рисунок 3.62 – Зміна кольорової теми

Усі представлені функції спрямовані на забезпечення зручної, безпечної та персоналізованої взаємодії користувача з онлайн-сервісом.

Висновки до розділу 3

У цьому розділі було детально розглянуто процес моделювання, проєктування, реалізації та тестування онлайн-сервісу пошуку роботи. На основі поставлених вимог було створено повноцінну інформаційну систему з чітко

визначеною архітектурою, яка поєднує frontend (React), backend (Go) та базу даних (PostgreSQL).

Сервіс підтримує дві основні ролі користувачів – шукача роботи та роботодавця, кожна з яких має доступ до персоналізованого функціоналу. Реалізовано гнучку систему керування профілями, створення та редагування вакансій і резюме, подання та обробки заявок, а також вбудований чат для комунікації між сторонами. Інтерфейс побудований за принципами SPA, що забезпечує швидку та зручну взаємодію користувача з додатком.

Особливу увагу приділено безпеці системи – реалізовано автентифікацію з JWT, перевірку прав доступу, логування дій та валідацію даних. Всі дії на платформі протестовані для обох типів користувачів, що підтверджує її функціональну цілісність та готовність до використання в реальних умовах.

Отже, поставлені завдання виконано у повному обсязі, а розроблений онлайн-сервіс відповідає сучасним вимогам до вебзастосунків у сфері пошуку роботи.

ВИСНОВКИ

У межах даної кваліфікаційної роботи було виконано повний цикл розробки онлайн-сервісу пошуку роботи – від аналізу існуючих платформ до реалізації власного вебзастосунку з використанням сучасних інформаційних технологій. Робота охоплює як теоретичні, так і практичні аспекти, що дозволяє зробити низку важливих висновків щодо актуальності теми, ефективності обраних підходів і результатів реалізації.

На етапі дослідження було проаналізовано функціональність популярних онлайн-сервісів пошуку роботи, таких як Work.ua, Jooble, Rabota.ua, Indeed, LinkedIn тощо. Це дозволило виділити основні модулі, які обов’язково мають бути присутні в подібних системах: реєстрація користувачів, створення резюме, створення та управління вакансіями, фільтрація пропозицій, пошук, особистий кабінет, а також механізми зв’язку між роботодавцем і здобувачем. Також було визначено ключові недоліки типових сервісів, серед яких – складність інтерфейсу, наявність реклами, відсутність гнучкої фільтрації або обмежена підтримка новітніх технологій.

На основі проведеного аналізу було сформульовано вимоги до власного сервісу, в якому вдалося реалізувати як базову функціональність, так і додаткові елементи, що покращують зручність використання та розширюють можливості користувача. Архітектура системи побудована відповідно до принципів розділення відповідальностей.

Система підтримує два основні типи користувачів – роботодавців і шукачів. Кожен тип має власний функціонал і доступ до відповідних можливостей. Для шукачів реалізовано зручну форму створення резюме, можливість його редагування, перегляд вакансій і подання заявок. Роботодавці, у свою чергу, мають можливість створювати й редагувати вакансії, переглядати кандидатів, змінювати статуси заявок, а також спілкуватися із претендентами за допомогою внутрішнього чату.

Особливу увагу було приділено безпеці системи. Реалізовано захищену авторизацію та автентифікацію, обробку токенів доступу, а також обмеження

доступу до приватних даних залежно від ролі користувача. Структура бази даних передбачає чітке логічне розділення сутностей, що полегшує масштабування проєкту в майбутньому.

Крім основного функціоналу, система має потенціал для подальшого розвитку. Можна впровадити рекомендовану систему на основі штучного інтелекту, яка автоматично підбиратиме релевантні вакансії або кандидатів. Також перспективним є створення мобільного застосунку, підтримка багатомовності, інтеграція з популярними месенджерами (наприклад, Telegram, Viber), а також можливість проведення відеоінтерв’ю прямо через платформу.

Підсумовуючи результати, можна стверджувати, що мета дипломної роботи була досягнута. Створений вебзастосунок успішно виконує всі поставлені перед ним функції, має сучасний інтерфейс і може використовуватися як база для реального онлайн-сервісу пошуку роботи. Реалізація даного проєкту підтверджує практичну значущість теми та демонструє вміння використовувати сучасні технології для вирішення прикладних задач у сфері веброзробки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Державна служба зайнятості. URL: https://dcz.gov.ua/ (дата звернення: 24.04.2025).
2. Онлайн-сервіс пошуку роботи work.ua. URL: https://www.work.ua/ (дата звернення: 24.04.2025).
3. Онлайн-сервіс пошуку роботи robota.ua. URL: https://robota.ua/ (дата звернення: 24.04.2025).
4. Онлайн-сервіс пошуку роботи linkedin. URL: https://www.linkedin.com/jobs/ (дата звернення: 24.04.2025).
5. Олена Іваненко. Ринок праці і зайнятість під час війни: стан та перспективи. Соціологія: теорія, методи, маркетинг, 2022, 4. DOI 10.15407/sociology2022.04.056. URL: https://stmm.in.ua/archive/ukr/2022-4/7.pdf (дата звернення: 26.04.2025).
6. Лизанець А.Г., Нестерова С.В.. Розвиток віртуального ринку праці в умовах інформаційного суспільства. Економіка і суспільство, випуск №12, 2017, с. 480 – 485, УДК 331.5. URL: https://economyandsociety.in.ua/journals/12_ukr/81.pdf (дата звернення: 26.04.2025).
7. Балабанюк Ж.В. Пошуки вакансії в Мережі: особливості віртуального ринку праці. URL: http://forbes.net.ua/ua/opinions/1424726-poshuki-vakansiyi-v-merezhiosoblivosti-virtualnogo-rinku-praci (дата звернення: 27.04.2025)
8. Борюшкіна О.В. Визначення поняття «ринок праці» в економічній та соціологічній науках / О.В. Борюшкіна // Методологія, теорія та практика соціологічного аналізу сучасного суспільства : збірник наукових праць. – Х. : Видавничий центр Харківського національного університету імені В.Н. Каразіна, 2005. – С. 206–210 (дата звернення: 28.04.2025).
9. Дегтярьова Л. М., Гроза П. М., Сомов С. М. Технології розробки програмного забезпечення: навчальний посібник. Полтава: ПолтНТУ, 2017. 218с (дата звернення: 29.04.2025).
10. React Documentation. URL: https://react.dev/ (дата звернення: 30.04.2025).

11. PostgreSQL Official Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 30.04.2025).

12. Go.dev – Official Go Programming Language Documentation. URL: https://go.dev/doc/ (дата звернення: 30.04.2025).

13. ВОНА хаб. vonahub.org.ua. Короткий огляд сучасного ринку праці в Україні. URL: https://vonahub.org.ua/krashchi-resursy-dlia-poshuku-roboty-j-kar-iernoho-zrostannia (дата звернення: 03.05.2025).

14. Онлайн-сервіс пошуку роботи jooble. URL: https://de.jooble.org/ (дата звернення: 24.04.2025).

15. Онлайн-сервіс пошуку роботи freelancehunt. URL: https://freelancehunt.com/ (дата звернення: 24.04.2025).

16. Онлайн-сервіс пошуку роботи dou.ua. URL: https://dou.ua/ (дата звернення: 24.04.2025).

17. Онлайн-сервіс пошуку роботи headhunter. URL: https://www.headhunter-europe.com/ (дата звернення: 24.04.2025).

18. Харб. Оптимізація запитів. Основи EXPLAIN у PostgreSQL. URL: https://habr.com/ru/articles/203320/ (дата звернення: 24.04.2025).

19. Харб. Навіщо існують вимоги ACID для БД? URL: https://habr.com/ru/articles/535616/ (дата звернення: 24.04.2025).

20. Типи даних JSON/JSONB. URL: https://docs.arenadata.io/ru/ADPG/current/how-to/queries/use-complex-types/json-jsonb.html (дата звернення: 24.04.2025).

21. The Clean Code Blog. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (дата звернення: 24.04.2025).

22. OpenAPI Specification. Swagger Documentation. URL: https://swagger.io/specification/ (дата звернення: 30.04.2025).

23. Docker Docs. Docker Overview. URL: https://docs.docker.com/get-started/overview/ (дата звернення: 30.04.2025).

24. OWASP Foundation. Top 10 Web Application Security Risks. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 30.04.2025).

25. JWT - JSON Web Tokens. Introduction. URL: https://jwt.io/introduction (дата звернення: 30.04.2025).

26. Express.js Documentation. URL: https://expressjs.com/ (дата звернення: 30.04.2025).

27. Node.js Documentation. URL: https://nodejs.org/en/docs (дата звернення: 02.05.2025).

ДОДАТОК А

Програмна реалізація

conf.go

```
package config
import (
    "fmt"
    "strings"
    "github.com/spf13/viper")
type Config struct {
    DB struct {
       URL string    }}
var Conf Config
func init() {
    viper.SetConfigName("config")
    viper.SetConfigType("yaml")
    viper.AddConfigPath("./config/")
    viper.SetEnvKeyReplacer(strings.NewReplacer(".", "_"))
    viper.AutomaticEnv()
    if err := viper.ReadInConfig(); err != nil {
       fmt.Print("Config file not found...")    }
    if err := viper.Unmarshal(&Conf); err != nil {
       panic(fmt.Sprintf("Unable to decode into struct %e", err))    }}
```

config.yaml

```
application:
  mode: "DEV"
db:
  url: "postgresql://postgres:1234567@localhost:5432/jobfinder"
rabbit:
  url: "amqp://login:password@localhost:5672"
auth:
  username: "admin"
  password: "admin"
tg:
  token: "7554337983:AAE8XXblGQzHfUFy3bUoHVvVqUa6ew7r4q0"
kafka:
  brokers: "localhost:29092"
redis:
  host: "switchyard.proxy.rlwy.net:24199"
  password: "ddmhkUJWumkgqXZwhNNPOncZsEoSnqVZ"
  db: 0
  pool:
    size: 1000
  min:
    idle:
      conns: 50
  max:
    idle:
      conns: 150
    active:
      conns: 0
```

chat_room_repository

```
package db
import (
    "context"
    "errors"
```

```
    "github.com/google/uuid"
    "gorm.io/gorm"
    "jobfinder-web/internal/domain"
    "time")
type chatRoomRepository struct {
    db *gorm.DB}
func NewChatRoomRepository(db *gorm.DB) domain.ChatRoomRepository {
    repo := &chatRoomRepository{db: db}
    repo.init()
    return repo}
func (r *chatRoomRepository) init() {
    if err := r.db.AutoMigrate(&domain.ChatRoom{}, &domain.ChatRoomParticipant{});
err != nil {
       panic("Failed to migrate database: " + err.Error())    }}
func (r *chatRoomRepository) CreateChatRoom(ctx context.Context, chatRoom
*domain.ChatRoom) error {
    return r.db.WithContext(ctx).Create(chatRoom).Error}
func (r *chatRoomRepository) GetChatRoomByApplicationID(ctx context.Context,
applicationID domain.ApplicationID) (*domain.ChatRoom, error) {
    var chatRoom domain.ChatRoom
    if err := r.db.WithContext(ctx).Where("application_id = ?",
applicationID).First(&chatRoom).Error; err != nil {
       if errors.Is(err, gorm.ErrRecordNotFound) {
          return nil, nil       }
       return nil, err    }
    return &chatRoom, nil}
func (r *chatRoomRepository) GetChatRoomByID(ctx context.Context, chatRoomID
domain.ChatRoomID) (*domain.ChatRoom, error) {
    var chatRoom domain.ChatRoom
    if err := r.db.WithContext(ctx).Where("id = ?",
chatRoomID).First(&chatRoom).Error; err != nil {
       if errors.Is(err, gorm.ErrRecordNotFound) {
          return nil, nil       }
       return nil, err    }
    return &chatRoom, nil}
func (r *chatRoomRepository) CreateParticipant(ctx context.Context, participant
*domain.ChatRoomParticipant) error {
    return r.db.WithContext(ctx).Create(participant).Error}
func (r *chatRoomRepository) GetParticipant(ctx context.Context, chatRoomID
domain.ChatRoomID, userID domain.UserID) (*domain.ChatRoomParticipant, error) {
    var participant domain.ChatRoomParticipant
    if err := r.db.WithContext(ctx).Where("chat_room_id = ? AND user_id = ?",
chatRoomID, userID).First(&participant).Error; err != nil {
       if errors.Is(err, gorm.ErrRecordNotFound) {          return nil, nil
}
       return nil, err    }
    return &participant, nil}
func (r *chatRoomRepository) UpdateParticipantLastReadTime(ctx context.Context,
chatRoomID domain.ChatRoomID, userID domain.UserID, lastReadTime time.Time) error
{
    participant, err := r.GetParticipant(ctx, chatRoomID, userID)
    if err != nil {
       return err    }
    if participant == nil {
       participant = &domain.ChatRoomParticipant{
          ID:                 uuid.New(),
          ChatRoomID:         chatRoomID,
          UserID:             userID,
          LastReadMessageTime: lastReadTime,      }
       return r.db.WithContext(ctx).Create(participant).Error   }
    participant.LastReadMessageTime = lastReadTime
    return r.db.WithContext(ctx).Save(participant).Error}
func (r *chatRoomRepository) GetChatRoomsForUser(ctx context.Context, userID
domain.UserID) ([]*domain.ChatRoom, error) {
```

```
    var chatRooms []*domain.ChatRoom
    if err := r.db.WithContext(ctx).Joins("JOIN chat_room_participants ON
chat_room_participants.chat_room_id =
chat_rooms.id").Where("chat_room_participants.user_id = ?",
userID).Find(&chatRooms).Error; err != nil {
       return nil, err    }
    return chatRooms, nil}
func (r *chatRoomRepository) DeleteChatRoom(ctx context.Context, chatRoomID
domain.ChatRoomID) error {
    if err := r.db.WithContext(ctx).Where("chat_room_id = ?",
chatRoomID).Delete(&domain.ChatRoomParticipant{}).Error; err != nil {
       return err    }
    return r.db.WithContext(ctx).Where("id = ?",
chatRoomID).Delete(&domain.ChatRoom{}).Error}
```

job_repository

```
      package db
import (
    "context"
    "errors"
    "gorm.io/gorm"
    "jobfinder-web/internal/domain")
type jobRepository struct {    db *gorm.DB}
func NewJobRepository(db *gorm.DB) domain.JobRepository {
    jobRepo := &jobRepository{db: db}
    jobRepo.init()
    return jobRepo}
func (r *jobRepository) init() {
    if err := r.db.AutoMigrate(&domain.Job{}, &domain.Skill{}); err != nil {
       panic("Failed to migrate database: " + err.Error())    }}
func (r *jobRepository) CreateJob(ctx context.Context, job *domain.Job) error {
    return r.db.WithContext(ctx).Create(job).Error}
func (r *jobRepository) GetJobByID(ctx context.Context, id domain.JobID)
(*domain.Job, error) {
    var job domain.Job
    if err := r.db.WithContext(ctx).Preload("RequiredSkills").Where("id = ?",
id).First(&job).Error; err != nil {
       if errors.Is(err, gorm.ErrRecordNotFound) {
          return nil, nil       }
       return nil, err   }
    return &job, nil}
func (r *jobRepository) SearchJobs(ctx context.Context, query string, filters
map[string]interface{}, page, pageSize int) ([]*domain.Job, error) {
    var jobs []*domain.Job
    dbQuery :=
r.db.WithContext(ctx).Model(&domain.Job{}).Preload("RequiredSkills")
    if query != "" {
       dbQuery = dbQuery.Where("title ILIKE ? OR description ILIKE ?",
"%"+query+"%", "%"+query+"%")    }
    if skillNames, ok := filters["skill_names"].([]string); ok && len(skillNames)
> 0 {
       dbQuery = dbQuery.Joins("JOIN job_skills ON job_skills.job_id = jobs.id").
          Joins("JOIN skills ON skills.id = job_skills.skill_id").
          Where("skills.name IN ?", skillNames)    }
    if salary, ok := filters["salary"].(string); ok {
       dbQuery = dbQuery.Where("salary >= ?", salary)    }
    if employmentType, ok := filters["employment_type"].(string); ok {
       dbQuery = dbQuery.Where("employment_type = ?", employmentType)    }
    if englishLevel, ok := filters["english_level"].(string); ok {
       dbQuery = dbQuery.Where("english_level = ?", englishLevel)    }
    if countryID, ok := filters["country_id"].(string); ok {
       dbQuery = dbQuery.Where("country_id = ?", countryID)   }
    if postedBy, ok := filters["posted_by"].(string); ok {
```

```
       dbQuery = dbQuery.Where("posted_by = ?", postedBy)    }
    offset := (page - 1) * pageSize
    if err := dbQuery.Offset(offset).Limit(pageSize).Find(&jobs).Error; err != nil
{
       return nil, err    }
    return jobs, nil}
func (r *jobRepository) GetJobsByUser(ctx context.Context, userID domain.UserID,
page, pageSize int) ([]*domain.Job, error) {
    var jobs []*domain.Job
    offset := (page - 1) * pageSize
    if err := r.db.WithContext(ctx).Preload("RequiredSkills").Where("posted_by =
?", userID).Offset(offset).Limit(pageSize).Find(&jobs).Error; err != nil {
       return nil, err    }
    return jobs, nil}
func (r *jobRepository) UpdateJob(ctx context.Context, job *domain.Job) error {
    result := r.db.WithContext(ctx).Save(job)
    if result.Error != nil {
       return result.Error  }
    if result.RowsAffected == 0 {
       return errors.New("no rows affected, job not found") }
    return nil}
func (r *jobRepository) DeleteJob(ctx context.Context, id domain.JobID) error {
    return r.db.WithContext(ctx).Transaction(func(tx *gorm.DB) error {
       // Delete associated chat room participants first
       if err := tx.Where("chat_room_id IN (SELECT id FROM chat_rooms WHERE
application_id IN (SELECT id FROM applications WHERE job_id = ?))",
id).Delete(&domain.ChatRoomParticipant{}).Error; err != nil {
          return err}
       // Delete associated chat rooms
       if err := tx.Where("application_id IN (SELECT id FROM applications WHERE
job_id = ?)", id).Delete(&domain.ChatRoom{}).Error; err != nil {
          return err}
       // Delete associated applications
       if err := tx.Where("job_id = ?", id).Delete(&domain.Application{}).Error;
err != nil {
          return err }
       // Retrieve the job
       var job domain.Job
       if err := tx.First(&job, id).Error; err != nil {
          if errors.Is(err, gorm.ErrRecordNotFound) {
             return errors.New("job not found")    }
          return err     }
       // Clear associated skills
       if err := tx.Model(&job).Association("RequiredSkills").Clear(); err != nil
{
          return err  }
       // Delete the job
       if err := tx.Delete(&job).Error; err != nil {
          return err     }
       return nil    })}
func (r *jobRepository) ListAllJobs(ctx context.Context) ([]*domain.Job, error) {
    var jobs []*domain.Job
    if err := r.db.WithContext(ctx).Preload("RequiredSkills").Find(&jobs).Error;
err != nil {
       return nil, err    }
    return jobs, nil}
```

userRepository

```
      package db
import (
    "context"
    "errors"
    "gorm.io/gorm"
```

```
    "jobfinder-web/internal/domain")
type userRepository struct {
    db *gorm.DB}
func NewUserRepository(db *gorm.DB) domain.UserRepository {
    userRepo := &userRepository{db: db}
    userRepo.init()
    return userRepo}
func (r *userRepository) init() {
    if err := r.db.AutoMigrate(&domain.User{}, &domain.Skill{}); err != nil {
       panic("Failed to migrate database: " + err.Error())    }}
func (r *userRepository) CreateUser(ctx context.Context, user *domain.User) error
{
    return r.db.WithContext(ctx).Create(user).Error}
func (r *userRepository) GetUserByID(ctx context.Context, id domain.UserID)
(*domain.User, error) {
    var user domain.User
    if err := r.db.WithContext(ctx).Preload("Skills").Where("id = ?",
id).First(&user).Error; err != nil {
       if errors.Is(err, gorm.ErrRecordNotFound) {
          return nil, nil       }
       return nil, err    }
    return &user, nil}
func (r *userRepository) FindUserByEmail(ctx context.Context, email string)
(*domain.User, error) {
    var user domain.User
    if err := r.db.WithContext(ctx).Preload("Skills").Where("email = ?",
email).First(&user).Error; err != nil {
       if errors.Is(err, gorm.ErrRecordNotFound) {
          return nil, nil       }
       return nil, err    }
    return &user, nil}
func (r *userRepository) UpdateUser(ctx context.Context, user *domain.User) error
{
    result := r.db.WithContext(ctx).Save(user)
    if result.Error != nil {
       return result.Error    }
    if result.RowsAffected == 0 {
       return errors.New("no rows affected, user not found")    }
    return nil}
func (r *userRepository) DeleteUser(ctx context.Context, id domain.UserID) error {
    result := r.db.WithContext(ctx).Delete(&domain.User{}, id)
    if result.Error != nil {
       return result.Error    }
    if result.RowsAffected == 0 {
       return errors.New("no rows affected, user not found")    }
    return nil}
func (r *userRepository) ListUsers(ctx context.Context) ([]*domain.User, error) {
    var users []*domain.User
    if err := r.db.WithContext(ctx).Preload("Skills").Find(&users).Error; err !=
nil {
       return nil, err    }
    return users, nil}
```

ports.go

```
     package domain
import (
    "context"
    "github.com/google/uuid"
    "mime/multipart"
    "time")
type UserRepository interface {
    CreateUser(ctx context.Context, user *User) error
    GetUserByID(ctx context.Context, id UserID) (*User, error)
```

```
    FindUserByEmail(ctx context.Context, email string) (*User, error)
    UpdateUser(ctx context.Context, user *User) error
    DeleteUser(ctx context.Context, id UserID) error
    ListUsers(ctx context.Context) ([]*User, error)}
type JobRepository interface {
    CreateJob(ctx context.Context, job *Job) error
    GetJobByID(ctx context.Context, id JobID) (*Job, error)
    SearchJobs(ctx context.Context, query string, filters map[string]interface{},
page, pageSize int) ([]*Job, error)
    GetJobsByUser(ctx context.Context, userID UserID, page, pageSize int) ([]*Job,
error)
    UpdateJob(ctx context.Context, job *Job) error
    DeleteJob(ctx context.Context, id JobID) error
    ListAllJobs(ctx context.Context) ([]*Job, error)}
type ApplicationRepository interface {
    CreateApplication(ctx context.Context, app *Application) error
    GetApplicationByID(ctx context.Context, id ApplicationID) (*Application,
error)
    ListApplicationsByUser(ctx context.Context, userID UserID) ([]*Application,
error)
    ListApplicationsByJob(ctx context.Context, jobID JobID) ([]*Application,
error)
    UpdateApplication(ctx context.Context, app *Application) error
    DeleteApplication(ctx context.Context, id ApplicationID) error
    ListAllApplications(ctx context.Context) ([]*Application, error)}
type ChatRoomRepository interface {
    CreateChatRoom(ctx context.Context, chatRoom *ChatRoom) error
    GetChatRoomByApplicationID(ctx context.Context, applicationID ApplicationID)
(*ChatRoom, error)
    GetChatRoomByID(ctx context.Context, chatRoomID ChatRoomID) (*ChatRoom, error)
    CreateParticipant(ctx context.Context, participant *ChatRoomParticipant) error
    GetParticipant(ctx context.Context, chatRoomID ChatRoomID, userID UserID)
(*ChatRoomParticipant, error)
    UpdateParticipantLastReadTime(ctx context.Context, chatRoomID ChatRoomID,
userID UserID, lastReadTime time.Time) error
    GetChatRoomsForUser(ctx context.Context, userID UserID) ([]*ChatRoom, error)
    DeleteChatRoom(ctx context.Context, chatRoomID ChatRoomID) error}
type MessageRepository interface {
    CreateMessage(ctx context.Context, message *Message) error
    GetMessagesByChatRoomID(ctx context.Context, chatRoomID ChatRoomID)
([]*Message, error)
    GetUnreadMessageCount(ctx context.Context, chatRoomID ChatRoomID, userID
UserID, lastReadTime time.Time) (int, error)
    GetLastMessagesByChatRoomID(ctx context.Context, chatRoomID ChatRoomID)
(*Message, error)
    DeleteMessagesByChatRoomID(ctx context.Context, chatRoomID ChatRoomID) error}
type CountryRepository interface {
    CreateCountry(ctx context.Context, country *Country) error
    GetAllCountries(ctx context.Context) ([]*Country, error)}
type CategoryRepository interface {
    CreateCategory(ctx context.Context, category *Category) error
    GetAllCategories(ctx context.Context) ([]*Category, error)}
type SkillRepository interface {
    CreateSkill(ctx context.Context, skill *Skill) error
    GetAllSkills(ctx context.Context) ([]*Skill, error)}
type UserService interface {
    RegisterUser(ctx context.Context, dto RegisterUserDTO) (User, error)
    LoginUser(ctx context.Context, dto LoginUserDTO) (User, error)
    GetUserByID(ctx context.Context, id UserID) (User, error)
    UpdateUser(ctx context.Context, dto UpdateUserDTO) error
    DeleteUser(ctx context.Context, id UserID) error
    ListUsers(ctx context.Context) ([]User, error)}
type JobService interface {
    PostJob(ctx context.Context, dto PostJobDTO) error
```

```
    FindJob(ctx context.Context, id JobID) (Job, error)
    SearchJobs(ctx context.Context, dto SearchJobsDTO) ([]Job, error)
    GetJobsByUser(ctx context.Context, userID UserID, page, pageSize int) ([]Job,
error)
    UpdateJob(ctx context.Context, dto UpdateJobDTO) error
    DeleteJob(ctx context.Context, id JobID) error
    ListAllJobs(ctx context.Context) ([]Job, error)}
type ApplicationService interface {
    ApplyToJob(ctx context.Context, dto ApplyToJobDTO) error
    ListUserApplications(ctx context.Context, userID UserID) ([]Application,
error)
    ListJobApplications(ctx context.Context, jobID JobID) ([]Application, error)
    UpdateApplication(ctx context.Context, dto UpdateApplicationDTO) error
    DeleteApplication(ctx context.Context, id ApplicationID) error
    GetApplicationByID(ctx context.Context, id ApplicationID) (Application, error)
    ListAllApplications(ctx context.Context) ([]Application, error)
    UpdateApplicationStatus(ctx context.Context, dto UpdateApplicationStatusDTO)
error
    DeleteApplicationsByJobID(ctx context.Context, jobID JobID) error}
type ChatService interface {
    CreateChatRoomForApplication(ctx context.Context, applicationID ApplicationID)
(ChatRoom, error)
    SendMessage(ctx context.Context, dto SendMessageDTO) (Message, error)
    GetMessages(ctx context.Context, chatRoomID ChatRoomID) ([]Message, error)
    GetChatRoomByApplicationID(ctx context.Context, applicationID ApplicationID)
(ChatRoom, error)
    GetChatRoomByID(ctx context.Context, chatRoomID ChatRoomID) (ChatRoom, error)
    GetChatUpdates(ctx context.Context, userID UserID, timeout time.Duration)
([]ChatUpdate, error)
    GetUnreadMessageCount(ctx context.Context, chatRoomID ChatRoomID, userID
UserID) (int, error)
    UpdateLastReadTime(ctx context.Context, chatRoomID ChatRoomID, userID UserID,
lastReadTime time.Time) error
    GetChatRoomsForUser(ctx context.Context, userID UserID) ([]ChatRoom, error)
    GetLastMessage(ctx context.Context, chatRoomID ChatRoomID) (Message, error)
    GetUnreadChats(ctx context.Context, userID UserID) ([]UnreadChat, error)
    DeleteChatRoom(ctx context.Context, chatRoomID ChatRoomID) error}
type CountryService interface {
    GetAllCountries(ctx context.Context) ([]Country, error)}
type CategoryService interface {
    GetAllCategories(ctx context.Context) ([]Category, error)}
type SkillService interface {
    GetAllSkills(ctx context.Context) ([]Skill, error)}
type FileService interface {
    SaveFile(file *multipart.FileHeader) (uuid.UUID, error)
    GetFileByUUID(fileUUID uuid.UUID) (string, error)}
```

application_service.go

```
package application
import (
    "context"
    "errors"
    "github.com/google/uuid"
    "jobfinder-web/internal/domain"
    "time")
type applicationService struct {
    repo        domain.ApplicationRepository
    jobService  domain.JobService
    chatService domain.ChatService}
func NewApplicationService(repo domain.ApplicationRepository, jobSvc
domain.JobService, chatSvc domain.ChatService) domain.ApplicationService {
    return &applicationService{repo: repo, jobService: jobSvc, chatService:
chatSvc}}
```

```
func (s *applicationService) ApplyToJob(ctx context.Context, dto
domain.ApplyToJobDTO) error {
    if dto.UserID == uuid.Nil || dto.JobID == uuid.Nil {
       return errors.New("invalid user ID, job ID, or resume ID")    }
    application := domain.Application{
       ID:          uuid.New(),
       UserID:      dto.UserID,
       JobID:       dto.JobID,
       FileId:      dto.FileID,
       CoverLetter: dto.CoverLetter,
       AppliedAt:   time.Now().UTC(),
       Status:      domain.Pending,    }
    if err := s.repo.CreateApplication(ctx, &application); err != nil {
return err   }
    _, err := s.chatService.CreateChatRoomForApplication(ctx, application.ID)
    if err != nil {      return err    }
    return nil}
func (s *applicationService) ListUserApplications(ctx context.Context, userID
domain.UserID) ([]domain.Application, error) {
    if userID == uuid.Nil {       return nil, errors.New("invalid user ID")    }
    apps, err := s.repo.ListApplicationsByUser(ctx, userID)
    if err != nil {
       return nil, err   }
    result := make([]domain.Application, len(apps))
    for i, app := range apps {       result[i] = *app   }
    return result, nil}
func (s *applicationService) ListJobApplications(ctx context.Context, jobID
domain.JobID) ([]domain.Application, error) {
    if jobID == uuid.Nil {
       return nil, errors.New("invalid job ID")   }
    apps, err := s.repo.ListApplicationsByJob(ctx, jobID)
    if err != nil {     return nil, err    }
    result := make([]domain.Application, len(apps))
    for i, app := range apps {       result[i] = *app }
    return result, nil}
func (s *applicationService) UpdateApplication(ctx context.Context, dto
domain.UpdateApplicationDTO) error {
    app, err := s.repo.GetApplicationByID(ctx, dto.ID)
    if err != nil {       return err   }
    if app == nil {       return errors.New("application not found")    }
    if dto.Status != nil {     app.Status = *dto.Status    }
    if dto.CoverLetter != nil {       app.CoverLetter = *dto.CoverLetter    }
    if dto.FileID != nil && *dto.FileID != uuid.Nil {       app.FileId =
*dto.FileID    }
    return s.repo.UpdateApplication(ctx, app)}
func (s *applicationService) DeleteApplication(ctx context.Context, id
domain.ApplicationID) error {
    if id == uuid.Nil {       return errors.New("invalid application ID")    }
    chatRoom, err := s.chatService.GetChatRoomByApplicationID(ctx, id)
    if err == nil {
       err = s.chatService.DeleteChatRoom(ctx, chatRoom.ID)
       if err != nil {
          return err     }    }
    return s.repo.DeleteApplication(ctx, id)}
func (s *applicationService) GetApplicationByID(ctx context.Context, id
domain.ApplicationID) (domain.Application, error) {
    if id == uuid.Nil {
       return domain.Application{}, errors.New("invalid application ID")    }
    app, err := s.repo.GetApplicationByID(ctx, id)
    if err != nil {
       return domain.Application{}, err    }
    if app == nil {
       return domain.Application{}, errors.New("application not found")    }
    return *app, nil}
```

```
func (s *applicationService) ListAllApplications(ctx context.Context)
([]domain.Application, error) {
    apps, err := s.repo.ListAllApplications(ctx)
    if err != nil {
       return nil, err    }
    result := make([]domain.Application, len(apps))
    for i, app := range apps {
       result[i] = *app    }
    return result, nil}
func (s *applicationService) UpdateApplicationStatus(ctx context.Context, dto
domain.UpdateApplicationStatusDTO) error {
    if dto.ApplicationID == uuid.Nil || dto.Status == "" {
       return errors.New("invalid application ID or status")    }
    app, err := s.repo.GetApplicationByID(ctx, dto.ApplicationID)
    if err != nil {
       return err   }
    if app == nil {       return errors.New("application not found")    }
    app.Status = dto.Status
    return s.repo.UpdateApplication(ctx, app)}
func (s *applicationService) DeleteApplicationsByJobID(ctx context.Context, jobID
domain.JobID) error {
    if jobID == uuid.Nil {
       return errors.New("invalid job ID")    }
    apps, err := s.repo.ListApplicationsByJob(ctx, jobID)
    if err != nil {
       return err    }
    for _, app := range apps {
       err = s.DeleteApplication(ctx, app.ID)
       if err != nil {      return err     }   }    return nil}
```

chat_service.go

```
      package application
import (
    "context"
    "errors"
    "github.com/google/uuid"
    "jobfinder-web/internal/domain"
    "time")
type chatService struct {
    chatRoomRepo domain.ChatRoomRepository
    messageRepo  domain.MessageRepository
    appService   domain.ApplicationRepository
    jobService   domain.JobService}
func NewChatService(chatRoomRepo domain.ChatRoomRepository, messageRepo
domain.MessageRepository, appSvc domain.ApplicationRepository, jobSvc
domain.JobService) domain.ChatService {
    return &chatService{chatRoomRepo: chatRoomRepo, messageRepo: messageRepo,
appService: appSvc, jobService: jobSvc}}
func (s *chatService) CreateChatRoomForApplication(ctx context.Context,
applicationID domain.ApplicationID) (domain.ChatRoom, error) {
    existingChatRoom, err := s.chatRoomRepo.GetChatRoomByApplicationID(ctx,
applicationID)
    if err != nil {
       return domain.ChatRoom{}, err  }
    if existingChatRoom != nil {
       return *existingChatRoom, nil   }
    chatRoom := domain.ChatRoom{
       ID:            uuid.New(),
       ApplicationID: applicationID,    }
    if err := s.chatRoomRepo.CreateChatRoom(ctx, &chatRoom); err != nil {
       return domain.ChatRoom{}, err   }
    application, err := s.appService.GetApplicationByID(ctx, applicationID)
    if err != nil {
```

```
        return domain.ChatRoom{}, err    }
    job, err := s.jobService.FindJob(ctx, application.JobID)
    if err != nil {
        return domain.ChatRoom{}, err}
    currentTime := time.Now().UTC()
    participant1 := domain.ChatRoomParticipant{
        ID:                  uuid.New(),
        ChatRoomID:          chatRoom.ID,
        UserID:              application.UserID,
        LastReadMessageTime: currentTime,   }
    participant2 := domain.ChatRoomParticipant{
        ID:                  uuid.New(),
        ChatRoomID:          chatRoom.ID,
        UserID:              job.PostedBy,
        LastReadMessageTime: currentTime    }
    if err := s.chatRoomRepo.CreateParticipant(ctx, &participant1); err != nil {
        return domain.ChatRoom{}, err   }
    if err := s.chatRoomRepo.CreateParticipant(ctx, &participant2); err != nil {
        return domain.ChatRoom{}, err }
    return chatRoom, nil}
func (s *chatService) SendMessage(ctx context.Context, dto domain.SendMessageDTO)
(domain.Message, error) {
    if dto.ChatRoomID == uuid.Nil || dto.SenderID == uuid.Nil || dto.Content == ""
{
        return domain.Message{}, errors.New("invalid input parameters")    }
    message := domain.Message{
        ID:         uuid.New(),
        ChatRoomID: dto.ChatRoomID,
        SenderID:   dto.SenderID,
        Content:    dto.Content,
        SentAt:     time.Now().UTC(),    }
    if err := s.messageRepo.CreateMessage(ctx, &message); err != nil {
        return domain.Message{}, err    }
    return message, nil}
func (s *chatService) GetMessages(ctx context.Context, chatRoomID
domain.ChatRoomID) ([]domain.Message, error) {
    if chatRoomID == uuid.Nil {
        return nil, errors.New("invalid chat room ID")    }
    messages, err := s.messageRepo.GetMessagesByChatRoomID(ctx, chatRoomID)
    if err != nil {
        return nil, err    }
    result := make([]domain.Message, len(messages))
    for i, msg := range messages {
        result[i] = *msg    }
    return result, nil}
func (s *chatService) GetChatRoomByApplicationID(ctx context.Context,
applicationID domain.ApplicationID) (domain.ChatRoom, error) {
    chatRoom, err := s.chatRoomRepo.GetChatRoomByApplicationID(ctx, applicationID)
    if err != nil {
        return domain.ChatRoom{}, err   }
    if chatRoom == nil {
        return domain.ChatRoom{}, errors.New("chat room not found")    }
    return *chatRoom, nil}
func (s *chatService) GetChatRoomByID(ctx context.Context, chatRoomID
domain.ChatRoomID) (domain.ChatRoom, error) {
    chatRoom, err := s.chatRoomRepo.GetChatRoomByID(ctx, chatRoomID)
    if err != nil {
        return domain.ChatRoom{}, err    }
    if chatRoom == nil {
        return domain.ChatRoom{}, errors.New("chat room not found")    }
    return *chatRoom, nil}
func (s *chatService) GetChatUpdates(ctx context.Context, userID domain.UserID,
timeout time.Duration) ([]domain.ChatUpdate, error) {
    startTime := time.Now().UTC()
```

```
    for {
       updates, err := s.checkForUpdates(ctx, userID)
       if err != nil {
          return nil, err    }
       if len(updates) > 0 {
          return updates, nil   }
       if time.Since(startTime) >= timeout {
          return []domain.ChatUpdate{}, nil   }
       time.Sleep(1 * time.Second   }}
func (s *chatService) checkForUpdates(ctx context.Context, userID domain.UserID)
([]domain.ChatUpdate, error) {
    chatRooms, err := s.chatRoomRepo.GetChatRoomsForUser(ctx, userID)
    if err != nil {
       return nil, err}
    var updates []domain.ChatUpdate
    for _, chatRoom := range chatRooms {
       participant, err := s.chatRoomRepo.GetParticipant(ctx, chatRoom.ID, userID)
       if err != nil {
          return nil, err}
       if participant == nil {
          continue}
       lastReadTime := participant.LastReadMessageTime
       unreadCount, err := s.messageRepo.GetUnreadMessageCount(ctx, chatRoom.ID,
userID, lastReadTime)
       if err != nil {
          return nil, err}
       if unreadCount > 0 {
          updates = append(updates, domain.ChatUpdate{
             ChatRoomID:  chatRoom.ID,
             UnreadCount: unreadCount,          })       }    }    return updates,
nil}
func (s *chatService) GetUnreadMessageCount(ctx context.Context, chatRoomID
domain.ChatRoomID, userID domain.UserID) (int, error) {
    participant, err := s.chatRoomRepo.GetParticipant(ctx, chatRoomID, userID)
    if err != nil {
       return 0, err    }
    if participant == nil {
       return 0, errors.New("participant not found")  }
    lastReadTime := participant.LastReadMessageTime
    return s.messageRepo.GetUnreadMessageCount(ctx, chatRoomID, userID,
lastReadTime)}
func (s *chatService) UpdateLastReadTime(ctx context.Context, chatRoomID
domain.ChatRoomID, userID domain.UserID, lastReadTime time.Time) error {
    return s.chatRoomRepo.UpdateParticipantLastReadTime(ctx, chatRoomID, userID,
lastReadTime)}
func (s *chatService) GetChatRoomsForUser(ctx context.Context, userID
domain.UserID) ([]domain.ChatRoom, error) {
    chatRooms, err := s.chatRoomRepo.GetChatRoomsForUser(ctx, userID)
    if err != nil {
       return nil, err   }
    result := make([]domain.ChatRoom, len(chatRooms))
    for i, chatRoom := range chatRooms {
       result[i] = *chatRoom   }
    return result, nil}
func (s *chatService) GetLastMessage(ctx context.Context, chatRoomID
domain.ChatRoomID) (domain.Message, error) {
    messages, err := s.messageRepo.GetLastMessagesByChatRoomID(ctx, chatRoomID)
    if err != nil {
      return domain.Message{}, err  }
    if messages == nil {
       return domain.Message{}, errors.New("no messages found")    }
    return *messages, nil}
func (s *chatService) GetUnreadChats(ctx context.Context, userID domain.UserID)
([]domain.UnreadChat, error) {
```

```
    chatRooms, err := s.chatRoomRepo.GetChatRoomsForUser(ctx, userID)
    if err != nil {
       return nil, err    }
    var unreadChats []domain.UnreadChat
    for _, chatRoom := range chatRooms {
       unreadCount, err := s.GetUnreadMessageCount(ctx, chatRoom.ID, userID)
       if err != nil {
          continue     }
       if unreadCount > 0 {
          application, err := s.appService.GetApplicationByID(ctx,
chatRoom.ApplicationID)
          if err != nil {       continue    }
          job, err := s.jobService.FindJob(ctx, application.JobID)
          if err != nil {
             continue     }
          lastMessage, err := s.GetLastMessage(ctx, chatRoom.ID)
          if err != nil {
             continue      }
          unreadChats = append(unreadChats, domain.UnreadChat{
             ApplicationID: chatRoom.ApplicationID,
             JobID:         job.ID,
             UnreadCount:   unreadCount,
             LastMessageAt: lastMessage.SentAt,          })       }    }
    return unreadChats, nil}

func (s *chatService) DeleteChatRoom(ctx context.Context, chatRoomID
domain.ChatRoomID) error {
    if err := s.messageRepo.DeleteMessagesByChatRoomID(ctx, chatRoomID); err !=
nil {
       return err    }
    if err := s.chatRoomRepo.DeleteChatRoom(ctx, chatRoomID); err != nil {
       return err    }
    return nil}
```

job_service.go

```
      package application
import (
    "context"
    "errors"
    "github.com/google/uuid"
    "jobfinder-web/internal/domain")
type jobService struct {
    repo domain.JobRepository}
func NewJobService(repo domain.JobRepository) domain.JobService {
    return &jobService{repo: repo}}
func (s *jobService) PostJob(ctx context.Context, dto domain.PostJobDTO) error {
    if dto.Title == "" || dto.Description == "" || dto.PostedBy == uuid.Nil {
       return errors.New("missing required fields")    }
    job := domain.Job{
       ID:             uuid.New(),
       Title:          dto.Title,
       Description:    dto.Description,
       Company:        dto.Company,
       Location:       dto.Location,
       Salary:         dto.Salary,
       PostedBy:       dto.PostedBy,
       CountryID:      dto.CountryID,
       EnglishLevel:   dto.EnglishLevel,
       EmploymentType: dto.EmploymentType,    }
    if len(dto.SkillIDs) > 0 {
       for _, skillID := range dto.SkillIDs {
          job.RequiredSkills = append(job.RequiredSkills, domain.Skill{ID:
skillID})       }    }
```

```
    return s.repo.CreateJob(ctx, &job)}
func (s *jobService) FindJob(ctx context.Context, id domain.JobID) (domain.Job,
error) {
    job, err := s.repo.GetJobByID(ctx, id)
    if err != nil {
       return domain.Job{}, err    }
    if job == nil {
       return domain.Job{}, errors.New("job not found")    }
    return *job, nil}
func (s *jobService) SearchJobs(ctx context.Context, dto domain.SearchJobsDTO)
([]domain.Job, error) {
    if dto.Page <= 0 || dto.PageSize <= 0 {
       return nil, errors.New("invalid pagination parameters")    }
    jobs, err := s.repo.SearchJobs(ctx, dto.Query, dto.Filters, dto.Page,
dto.PageSize)
    if err != nil {
       return nil, err    }
    result := make([]domain.Job, len(jobs))
    for i, job := range jobs {
       result[i] = *job    }
    return result, nil}
func (s *jobService) GetJobsByUser(ctx context.Context, userID domain.UserID,
page, pageSize int) ([]domain.Job, error) {
    if userID == uuid.Nil {
       return nil, errors.New("invalid user ID")    }
    if page <= 0 || pageSize <= 0 {
       return nil, errors.New("invalid pagination parameters")    }
    jobs, err := s.repo.GetJobsByUser(ctx, userID, page, pageSize)
    if err != nil {
       return nil, err    }
    result := make([]domain.Job, len(jobs))
    for i, job := range jobs {
       result[i] = *job    }
    return result, nil}
func (s *jobService) UpdateJob(ctx context.Context, dto domain.UpdateJobDTO) error
{
    job, err := s.repo.GetJobByID(ctx, dto.ID)
    if err != nil {
       return err    }
    if job == nil {
       return errors.New("job not found")    }
    if dto.Title != nil {
       job.Title = *dto.Title    }
    if dto.Description != nil {
       job.Description = *dto.Description    }
    if dto.Company != nil {
       job.Company = *dto.Company    }
    if dto.Location != nil {
       job.Location = *dto.Location    }
    if dto.Salary != nil {
       job.Salary = *dto.Salary    }
    if dto.CountryID != nil {
       job.CountryID = *dto.CountryID    }
    if dto.EnglishLevel != nil {
       job.EnglishLevel = *dto.EnglishLevel    }
    if dto.EmploymentType != nil {
       job.EmploymentType = *dto.EmploymentType    }
    if len(dto.SkillIDs) > 0 {
       var skills []domain.Skill
       for _, skillID := range dto.SkillIDs {
          skills = append(skills, domain.Skill{ID: skillID})       }
       job.RequiredSkills = skills    }
    return s.repo.UpdateJob(ctx, job)}
func (s *jobService) DeleteJob(ctx context.Context, id domain.JobID) error {
```

```
    if id == uuid.Nil {
       return errors.New("invalid job ID")    }
    return s.repo.DeleteJob(ctx, id)}
func (s *jobService) ListAllJobs(ctx context.Context) ([]domain.Job, error) {
    jobs, err := s.repo.ListAllJobs(ctx)
    if err != nil {
       return nil, err    }
    result := make([]domain.Job, len(jobs))
    for i, job := range jobs {
       result[i] = *job    }
    return result, nil}
```

user_service.go

```
package application
import (
    "context"
    "errors"
    "github.com/google/uuid"
    "golang.org/x/crypto/bcrypt"
    "jobfinder-web/internal/domain")
type userService struct {
    repo domain.UserRepository}
func NewUserService(repo domain.UserRepository) domain.UserService {
    return &userService{repo: repo}}
func (s *userService) RegisterUser(ctx context.Context, dto
domain.RegisterUserDTO) (domain.User, error) {
    if dto.Name == "" || dto.Email == "" || dto.Password == "" || (dto.Role !=
domain.JobSeeker && dto.Role != domain.Employer) {
       return domain.User{}, errors.New("missing or invalid required fields")    }
    existingUser, err := s.repo.FindUserByEmail(ctx, dto.Email)
    if err != nil {
       return domain.User{}, err    }
    if existingUser != nil {
       return domain.User{}, errors.New("email already registered")    }
    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(dto.Password),
bcrypt.DefaultCost)
    if err != nil {
       return domain.User{}, err    }
    user := domain.User{
       ID:                uuid.New(),
       Name:              dto.Name,
       Email:             dto.Email,
       Password:          string(hashedPassword),
       Role:              dto.Role,
       YearsOfExperience: 0,
       CountryID:         dto.CountryID,
       EnglishLevel:      dto.EnglishLevel,    }
    if err := s.repo.CreateUser(ctx, &user); err != nil {
       return domain.User{}, err    }
    return user, nil}
func (s *userService) LoginUser(ctx context.Context, dto domain.LoginUserDTO)
(domain.User, error) {
    user, err := s.repo.FindUserByEmail(ctx, dto.Email)
    if err != nil {
       return domain.User{}, err    }
    if user == nil {
       return domain.User{}, errors.New("invalid credentials")    }
    if err := bcrypt.CompareHashAndPassword([]byte(user.Password),
[]byte(dto.Password)); err != nil {
       return domain.User{}, errors.New("invalid credentials")    }
    return *user, nil}
func (s *userService) GetUserByID(ctx context.Context, id domain.UserID)
(domain.User, error) {
```

```
    user, err := s.repo.GetUserByID(ctx, id)
    if err != nil {
       return domain.User{}, err    }
    if user == nil {
       return domain.User{}, errors.New("user not found")    }
    return *user, nil}
func (s *userService) UpdateUser(ctx context.Context, dto domain.UpdateUserDTO)
error {
    user, err := s.repo.GetUserByID(ctx, dto.ID)
    if err != nil {
       return err    }
    if user == nil {
       return errors.New("user not found")    }
    if dto.Name != nil {
       user.Name = *dto.Name    }
    if dto.YearsOfExperience != nil {
       user.YearsOfExperience = *dto.YearsOfExperience    }
    if dto.ExpectedSalary != nil {
       user.ExpectedSalary = *dto.ExpectedSalary    }
    if dto.CountryID != nil {
       user.CountryID = *dto.CountryID    }
    if dto.EnglishLevel != nil {
       user.EnglishLevel = *dto.EnglishLevel    }
    if dto.WorkExperienceDescription != nil {
       user.WorkExperienceDescription = *dto.WorkExperienceDescription    }
    if len(dto.SkillIDs) > 0 {
       var skills []domain.Skill
       for _, skillID := range dto.SkillIDs {
          skills = append(skills, domain.Skill{ID: skillID})       }
       user.Skills = skills   }
    return s.repo.UpdateUser(ctx, user)}
func (s *userService) DeleteUser(ctx context.Context, id domain.UserID) error {
    if id == uuid.Nil {
       return errors.New("invalid user ID")    }
    return s.repo.DeleteUser(ctx, id)}
func (s *userService) ListUsers(ctx context.Context) ([]domain.User, error) {
    users, err := s.repo.ListUsers(ctx)
    if err != nil {
       return nil, err    }
    result := make([]domain.User, len(users))
    for i, user := range users {
       result[i] = *user    }
    return result, nil}
```