Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка вебзастосунку для пошуку та замовлення побутових послуг

Виконав: студент IV курсу, групи СН-41
спеціальності 122 Комп’ютерні науки
(шифр і назва спеціальності)

_____ Воловнік В. А.
(підпис) (прізвище та ініціали)

Керівник _____ Готович В. А.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Липак Г. І.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент _____ Бревус В.М.
(підпис) (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«__27__» _січня_ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)
за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)
Студенту Воловніку Віталію Андрійовичу
(прізвище, ім'я, по батькові)
1. Тема роботи Розробка вебзастосунку для пошуку та замовлення побутових послуг

Керівник роботи Готович Володимир Анатолійови, к.т.н., доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджені наказом ректора від «*07* « *травня* 20*25* року № *4/7-444*
2. Термін подання студентом завершеної роботи 23 червня 2025 р.
3. Вихідні дані до роботи Літературні та інтернет джерела за темою роботи

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Аналіз предметної області дослідження та порівняльний аналіз відомих рішень. 1.1 Аналіз предметної області. 1.2 Аналіз та порівняння існуючих рішень. 1.2.1 Платформа Handy. 1.2.2 Платформа TaskRabbit. 1.2.3 Вебзастосунок Airtasker. 1.3 Формування вимог до вебзастосунку. 1.4 Вибір технологій та інструментів для розробки вебзастосунку Make a Job. 1.5 Висновки до першого розділу. 2. Проектування архітектури та розробка вебзастосунку Make a Job. 2.1 Проектування архітектури вебзастосунку. 2.2 Розробка серверної частини. 2.2.1 Domain layer. 2.2.2 Data access layer. 2.2.3 Business logic layer. 2.2.4 Presentation layer (API). 2.3 Розробка клієнтської частини. 2.4 Висновки до другого розділу. 3.Розгортання та тестування вебзастосунку Make a Job. 3.1 Конфігурування та розгортання вебзастосунку. 3.2 Розробка Модульних тестів. 3.3 Тестування основних функцій застосунку. 3.4 Висновки до третього розділу. 4. Безпека життєдіяльності, основи охорони праці. 4.1 Долікарська допомого при ураженні електричним струмом. 4.2 Загальні вимоги безпеки з охорони праці для користувачів ПК. 4.3 Висновки до четвертого розділу. Висновки. Перелік джерел. Додатки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	Сенчишин В. С., кандидат технічних наук, доцент кафедри МТ	02.06.2025	05.06.2025

7. Дата видачі завдання 27 січня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	27.01.2025	*Виконано*
2.	Підбір та опрацювання літературних джерел по темі кваліфікаційної роботи	20.04.2025-05.05.2025	*Виконано*
3.	Проектування вебзастосунку та	06.05.2025-22.05.2025	*Виконано*
	Розробка вебзастосунку Make a Job		
4.	Оформлення розділу «Аналіз предметної області та	23.05.2025-01.06.2025	*Виконано*
	постановка завдання»		
5.	Оформлення розділу «Проектування архітектури та	23.05.2025-01.06.2025	*Виконано*
	Розробка вебзастосунку»		
6.	Оформлення розділу «Розгортання та тестування	23.05.2025-01.06.2025	*Виконано*
	Веб застосунку Make a Job»		
7.	Виконання завдання до підрозділу «Безпека життєдіяльності»	02.06.2025-05.06.2025	*Виконано*
8.	Виконання завдання до підрозділу «Основи охорони	02.06.2025-05.06.2025	*Виконано*
	праці»		
9.	Оформлення кваліфікаційної роботи	06.06.2025-10.06.2025	*Виконано*
10.	Нормоконтроль	20.06.2025	*Виконано*
11.	Перевірка на плагіат	20.06.2025	*Виконано*
12.	Попередній захист кваліфікаційної роботи		*Виконано*
13.	Захист кваліфікаційної роботи		

Студент _____ Воловнік В. А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Готович В. А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка вебзастосунку для пошуку та замовлення побутових послуг // Кваліфікаційна робота освітнього рівня «Бакалавр» // Воловнік Віталій Андрійович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СН-41// Тернопіль, 2025 // С. 68, рис. – 18, табл. – 2, кресл. – , додат. – 4, бібліогр. – 44.

Ключові слова: вебзастосунок, ASP.NET, Rest, Angular, API, модульне тестування, Domain-Driven Design

Кваліфікаційна робота присвячена розробці вебзастосунку для пошуку та замовлення побутових послуг Make a Job засобами C# ASP.NET та Angular.

В першому розділі кваліфікаційної роботи проаналізовано предметну область, оглянуто існуючі рішення на ринку, сформовано вимоги до застосунку та вибрано оптимальні інструменти для розробки.

В другому розділі кваліфікаційної роботи проведено проектування архітектури вебзастосунку, розроблено клієнтську, серверну частину застосунку.

В третьому розділі кваліфікаційної роботи описано процес налагодження та розгортання вебзастосунку. Проведено тестування застосунку за допомогою Unit-тестів та тестування в умовах наближених до реальних.

В четвертому розділі кваліфікаційної роботи розглянуто питання безпеки життєдіяльності та основи охорони праці, включаючи принципи надання долікарської допомоги при ураженні електричним струмом, а також загальні вимоги безпеки для користувачів персональних комп'ютерів з посиланнями на відповідні нормативні документи України.

ANNOTATION

Development of a Web Application for Finding and Ordering Household Services // Qualification work of the educational level «Bachelor» //Vitalii Volovnik // Ternopil Ivan Pulyu National Technical University, Computer and Information Systems and Software Engineering Faculty, Computer Sciences Department, group SN-41 // Ternopil, 2025 // P. 68, fig. – 18, tabl. – 2, chair. – , annexes. – 4, references – 44.

Keywords: web application, ASP.NET, Rest, Angular, API, Unit testing, Domain-Driven Design

The thesis is devoted to the development of a web application for searching and ordering household services Make a Job using C# ASP.NET and Angular.

The first chapter of the thesis analyzes the subject area, reviews existing solutions on the market, formulates requirements for the application, and selects the optimal tools for development.

The second chapter of the thesis designs the architecture of the web application and develops the client and server parts of the application.

The third section of the thesis describes the process of debugging and deploying the web application. The application is tested using unit tests and testing in conditions close to real life.

The fourth section of the thesis examines issues of life safety and the basics of occupational safety, including the principles of providing first aid in case of electric shock, as well as general safety requirements for personal computer users with references to the relevant regulatory documents of Ukraine.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

2FA (англ. Two-Factor Authentication) – двофакторна автентифікація, метод підтвердження особи за допомогою двох незалежних факторів.

API (англ. Application Programming Interface) – інтерфейс програмування додатків, набір визначених методів взаємодії між програмними компонентами.

CLI (англ. Command Line Interface) – інтерфейс командного рядка, спосіб взаємодії з програмою через введення текстових команд.

EF (англ. Entity Framework) – ORM-фреймворк від Microsoft для роботи з базами даних у .NET-застосунках.

HTTP (англ. HyperText Transfer Protocol) – протокол передачі гіпертексту, основний для обміну даними у вебсередовищі.

JSON (англ. JavaScript Object Notation) – текстовий формат обміну даними, зручний для зберігання та передачі структурованої інформації.

JWT (англ. JSON Web Token) – формат токена, що використовується для безпечної передачі інформації про користувача між клієнтом і сервером.

LINQ (англ. Language Integrated Query) – мова запитів, інтегрована в .NET, що дозволяє працювати з колекціями об’єктів у вигляді SQL-подібних виразів.

TOTP (англ. Time-based One-Time Password) – одноразовий пароль, який генерується на основі поточного часу, використовується для 2FA.

ЗМІСТ

ВСТУП

В сучасному цифровому суспільстві шалено збільшується потреба у зручних і доступних онлайн-інструментах, які полегшують щоденні задачі, сприяють гнучкій зайнятості та розширюють можливості самозайнятих громадян. Особливої актуальності це здобуває в умовах трансформації українського ринку праці, де з одного боку є значна кількість фахівців, готових надавати побутові послуги, а з іншого – користувачів, котрі прагнуть швидко відшукати виконавця в межах свого регіону. У той час як міжнародні сервіси на кшталт TaskRabbit, Airtasker або Handy є прикладами вдалих реалізацій подібних платформ, вони нерідко не враховують специфіку українського користувача: відсутність локалізації, адаптації до законодавчого поля, зручної платіжної інфраструктури та простого інтерфейсу.

Мета кваліфікаційної роботи – розробка вебдодатку Make a Job, що об'єднуватиме клієнтів та виконавців послуг з обслуговування побутових потреб споживачів.

Завдання дослідження передбачають: аналіз сфери застосування додатку, визначення потреб системи, проєктування архітектури вебзастосунку, впровадження функціональності на клієнтській та серверній сторонах, а також забезпечення безпеки, можливості масштабування та простоти використання. Окремий акцент зроблено на розробці системи рейтингів, взаємних відгуків і фільтрації завдань за місцем розташування.

Об'єктом дослідження є процеси проектування, розробки та розгортання вебзастосунків для пошуку та замовлення побутових послуг.

Предметом дослідження є підходи та інструменти для створення платформи, яка полегшує взаємодію між клієнтами та виконавцями щоденних задач, із врахуванням поточних стандартів безпеки, адаптивності та масштабованості вебзастосунків.

Практична цінність розробки полягає у можливості її використання, після відповідної адаптації, такими категоріями споживачів як малий бізнес,

самозайняті громадяни та звичайні споживачі, для підвищення доступності послуг, формування відкритого ринку мікрозайнятості та підтримки локальної економіки.

Кваліфікаційна робота складається з чотирьох розділів, що охоплюють теоретичні основи тематики, опис архітектури та реалізацію системи, дослідження ефективності й безпеки, а також вимоги з охорони праці. Загальний обсяг роботи становить 68 сторінок. Вона містить 18 рисунків, 2 таблиці, 44 джерел у списку літератури та 4 додатки, в яких наведено фрагменти програмного коду застосунку.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДОСЛІДЖЕННЯ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ ВІДОМИХ РІШЕНЬ

1.1 Аналіз предметної області

У сучасному світі все більше і більше людей прагнуть делегувати власні завдання іншим, для ефективнішого розпоряджання власним часом. Водночас кількість фахівців, які шукають додаткові методи заробітку, постійно зростає. Це створює попит на додатки, які з'єднують замовників та виконавців завдань.

Побутові послуги охоплюють широкий спектр завдань: прибирання приміщень, дрібний ремонт, доставка товарів, догляд за дітьми чи тваринами, збирання меблів, допомога з переїздом та багато інших. Більшість таких послуг має мають локальний характер – клієнти шукають виконавців поблизу, а завдання часто є разовими або короткостроковими. Через це традиційні методи пошуку виконавців за допомогою оголошень або запитів у соціальних мережах часто є неефективними, оскільки не гарантують якості чи своєчасності.

У багатьох країнах вже функціонують цифрові платформи, які автоматизують процес пошуку виконавців: TaskRabbit, Thumbtack, Handy тощо. Однак значна частина з них має обмеження або недоліки, які перешкоджають її ефективному використанню, зокрема:

- Відсутність української локалізації;
- Складна процедура реєстрації або надмірна комісія;
- Відсутність функціональності для перегляду відгуків, рейтингів або портфоліо виконавців;

Крім того, український ринок має свою специфіку. Частина населення відчуває недовіру до онлайн-сервісів через недостатній захисти даних або негативний досвід.

Тому розробка вебзастосунку Make a Job, що дозволяє швидко знайти або запропонувати послуги в конкретному регіоні, з акцентом на простоту, ефективність і безпеку – є актуальним завданням. Такий інструмент може стати

надійним посередником між людьми, які потребують допомоги, та тими, хто може надати.

На початку розробки вебзастосунку потрібно провести аналіз, популярних на думку користувачів вже існуючих рішень. Це дозволяє виявити сильні та слабкі сторони конкурентів, визначити очікування користувачів, а також сформувати функціональні та нефункціональні вимоги до власного продукту.

У сфері пошуку виконавців для побутових завдань найбільш відомими є такі платформи, як TaskRabbit, Airtasker та Handy. Вони всі мають спільну мету – з'єднувати замовників із виконавцями, однак реалізують це за допомогою різних підходів до організації інтерфейсу, системи взаємодії та монетизації. Перейдемо до відповідного порівняльного аналізу.

1.2 Аналіз та порівняння існуючих рішень

1.2.1 Платформа Handy

Handy – це платформа, орієнтована насамперед на послуги прибирання, ремонту, встановлення побутової техніки та інші стандартні побутові завдання. Вона позиціонує себе як сервіс «на вимогу», що дозволяє користувачеві зробити замовлення в декілька кліків, не обираючи конкретного виконавця вручну [1].

Однією з ключових особливостей Handy є автоматичне призначення виконавця. Коли замовник створює заяву на послугу, система самостійно підбирає найвідповіднішого фахівця на основі рейтингу, наявності вільного часу та геолокації. Це дозволяє спростити взаємодію, але позбавляє користувача можливості самостійного вибору.

Ще однією особливістю є фіксована вартість послуг, яка розраховується автоматично відповідно до типу завдання та обсягу роботи. З одного боку, це зручно для клієнтів, але з іншого – не залишає простору для гнучких індивідуальних домовленостей.

Інтерфейс Handy досить лаконічний і мінімалістичний, зображений на рисунку 1.1. Основні елементи доступні вже з головної сторінки: замовлення послуги, перегляд майбутніх бронювань історія замовлень. Важливу роль відіграє інтеграція з календарем та можливість змінювати дати й час візиту виконавця без зайвої бюрократії.

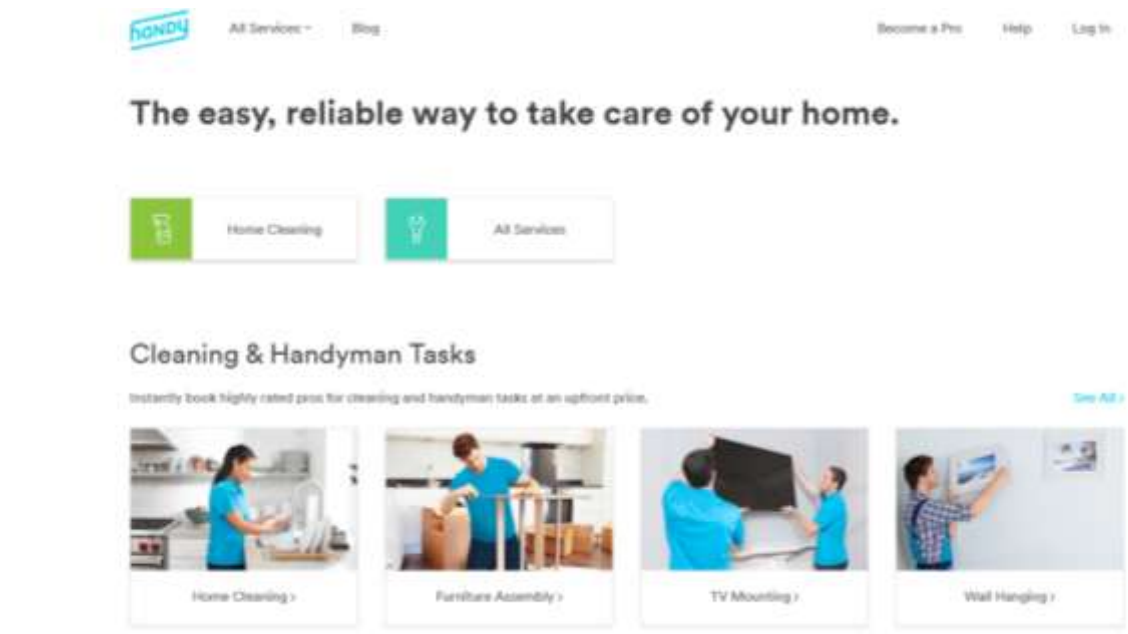

Рисунок 1.1 – Дизайн вебзастосунку Handy

У застосунку також передбачена система рейтингу та відгуків, що дозволяє оцінити якість виконаної роботи. Комунікація між сторонами відбувається через вбудований чат, однак він активується лише після підтвердження замовлення, що повною мірою обмежує можливість попереднього узгодження деталей.

Серед недоліків користувачі часто зазначають:

- обмежений перелік категорій послуг
- складність у вирішені спірних ситуацій
- жорстку політику відміни бронювання

1.2.2 Платформа TaskRabbit

TaskRabbit – одна з найвідоміших платформ для замовлення побутових послуг у США та ряді інших країн. Сервіс орієнтований на широкий спектр завдань: від дрібного ремонту, пакування речей під час переїзду, прибирання, доставки до допомоги з меблями IKEA. TaskRabbit належить саме IKEA, що вплинуло на її інтеграцію з продажом та складанням меблів [2].

Головна відмінність цього сервісу – можливість користувача самостійно обирати виконавця із запропонованого списку. Список формується на основі локації, рейтингу, ціни за годину та спеціалізації виконався. Це дозволяє користувачеві приймати більш зважене рішення, порівнюючи кандидатів між собою.

Після вибору виконавця обидві сторони можуть використовувати вбудований месенджер для узгодження деталей, включаючи точну адресу, особливості виконання тощо. Система нагадувань та повідомлень гарантує, що жоден учасник не пропустить заплановане завдання.

TaskRabbit дозволяє замовникам виставляти оцінки та писати розгорнуті відгуки. При цьому виконавці також можуть залишати відгук про замовлення, що створює двосторонню відповідальність.

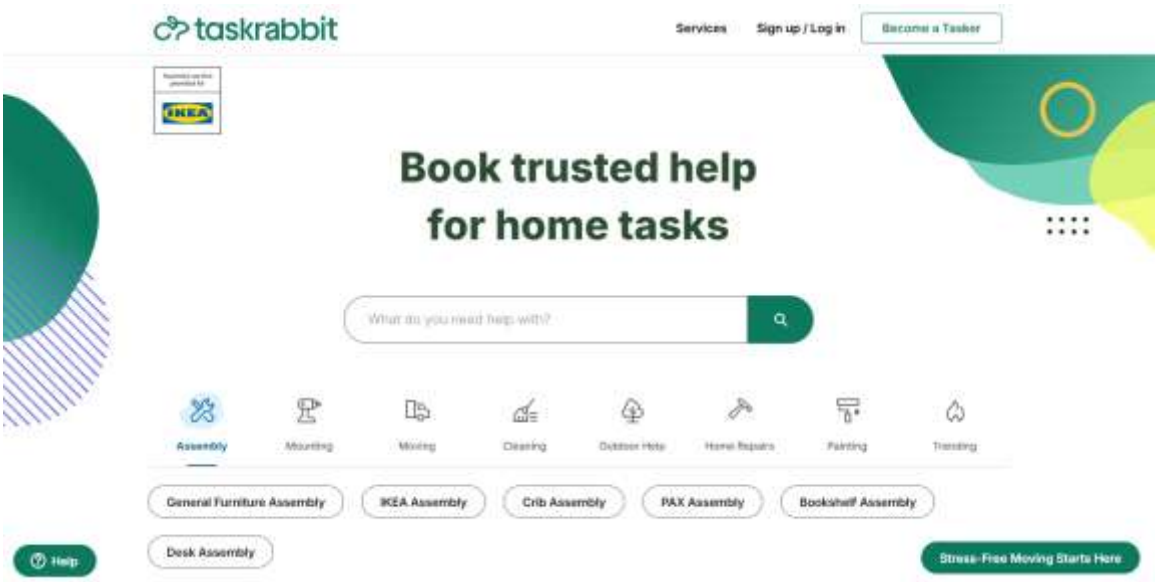

Рисунок 1.2 – Дизайн вебзастосунку TaskRabbit

Інтерфейс TaskRabbit виконаний у мінімалістичному стилі з переважанням світлих кольорів і простих форм, що забезпечує зручність використання як на комп’ютерах, так і на мобільних пристроях. Навігацію інтуїтивно зрозуміла: з головної сторінки користувач одразу може обрати тип послуги та вказати місце її виконання. Перегляд виконавців реалізовано у вигляді компактних карток із фото, рейтингом, ціною та коротким описом.

1.2.3 Вебзастосунок Airtasker

Airtasker – це австралійський вебзастосунок, що стрімко розширює свою присутність і в інших країнах. Платформа створена з метою об’єднання людей які шукають виконавців для побутових або спеціалізованих завдань, і фахівців готових їх виконати. Вона охоплює широкий спектр послуг: від простого прибирання або доставки до складніших завдань, таких як допомога з ремонтом обслуговування, створення вебсайтів або юридичні консультації [3].

На відміну від Handy, Airtasker не призначає виконавців автоматично. Користувач самостійно створює завдання, вказуючи опис терміни, бажану ціну й місце виконання. Після цього зареєстровані виконавці можуть відгукнутися на пропозицію, запропонувати власні умови та вести діалог із замовником. Таким чином, модель Airtasker нагадує біржу праці, де кожна сторона має більше свободи для узгодження умов.

Airtasker вирізняється високим рівнем гнучкості. Замовник може сформувати запит у довільній формі, не обмежуючись лише запропонованими шаблонами. У свою чергу, виконавець має змогу вести переговори щодо умов, запропонувати альтернативну дату чи вартість або навіть відмовитися від завдання без наслідків.

Незважаючи на це платформа має свої недоліки. Наприклад велика конкуренція між виконавцями іноді призводить до демпінгу цін, що може впливати на якість послуг. Крім того, сервіс поки що не підтримує українську локалізацію й не має повноцінної присутності на ринку України.

1.2.4 Порівняльний аналіз та виявлення недоліків відомих рішень

Для повного розуміння ринку та обґрунтування необхідності розробки власного вебзастосунку було проведено порівняльний аналіз ключових характеристик платформ: Handy, TaskRabbit та Airtasker. Результати цього аналізу можна представлено у таблиці 1.1

Таблиця 1.1 – Порівняльний аналіз відомих рішень

Характеристика	**Handy**	**TaskRabbit**	**Airtasker**
Призначення, Фокус	Стандартні побутові	Широкий спектр	Широкий спектр, та спеціалізовані
Вибір виконавця	Автоматичне призначення	Користувач обирає	Вибір за пропозиціями
Ціноутворення	Фіксоване	Погодинна ставка	Договірна
Комунікація	Після підтвердження	До та після замовлення	До та після замовлення
Система відгуків	Одностороння	Двостороння	Двостороння
Гнучкість умов	Низька	Середня	Висока
Локалізація	Переважно США та Європа	Переважно США та Європа	Австралія

Розбір популярних платформ дав змогу виявити ряд значних недоліків, котрі обмежують зручність і доступність таких сервісів для широкої аудиторії. Handy, хоч і гарантує швидке автоматичне призначення виконавців, позбавляє користувача права вибору, що може бути недоречним у ситуаціях, де потрібна довіра чи індивідуальний підхід. До того ж, фіксована модель ціноутворення не враховує специфіку багатьох задач і не дає можливості домовитися безпосередньо.

TaskRabbit, попри ширший функціонал, зорієнтований здебільшого на користувачів із США та деяких країн Європи, що ускладнює його повноцінне

використання в Україні. Також сервіс має помірну гнучкість у формуванні умов, проте не підтримує локалізацію українською мовою.

Airtasker, як найгнучкіший варіант, створює сприятливе середовище для обговорення умов між сторонами, проте надмірна конкуренція між виконавцями призводить до демпінгу, що негативно впливає на якість послуг. Окрім того, сервіс також не адаптований до українського ринку та не підтримує українську локалізацію.

Жодна з проаналізованих платформ не забезпечує повного покриття потреб українських користувачів, зокрема щодо мови інтерфейсу, адаптації до локального ринку, а також прозорості комунікації.

1.3 Формування вимог до вебзастосунку

Після дослідження предметної сфери та зіставлення доступних варіантів, було сформульовано потреби до вебзастосунку Make a Job. Розроблюваний сайт буде дозволяти користувачам шукати та замовляти широкий спектр побутових послуг, включаючи такі ключові категорії, як «Рекомендовані завдання» для загального переліку робіт, «Послуги з переїзду» для повної допомоги при переїзді, «Збирання меблів» для різних меблевих виробів та «Монтаж та встановлення» для професійного кріплення й інсталяції. Окрім цього, платформа охоплюватиме «Прибирання» для різноманітних запитів, «Роботи на подвір'ї» для комплексного догляду за прилеглою територією, «Офісні послуги» для корпоративних потреб, «Свята» для організації свят без клопотів та «Зимові завдання» для підготовки та обслуговування в зимовий період.

Ці вимоги поділяються на функціональні, які описують що саме має робити система, та нефункціональні, що визначають її якісні характеристики та обмеження. Було сформовано наступні функціональні вимоги:

- Реєстрація та авторизація користувачів – усі користувачі можуть створити власний обліковий запис;
- Публікація завдання – користувачі можуть опублікувати завдання;

- Оновлення завдання – користувачі можуть оновлювати інформацію або скасовувати завдання;
- Пошук завдань – користувачі можуть шукати завдання за певними параметрами наприклад за типом завдання, адресом, датою виконання тощо;
- Публікація пропозицій – користувачі можуть залишати пропозиції до певних завдань;
- Вибір виконавця – користувач який опублікував завдання може обрати виконавця серед користувачів які залишили пропозиції;
- Залишення відгуку – виконавці та/або замовники можуть залишати відгуки після виконання завдання;

Також сформовано наступні нефункціональні вимоги:

- Продуктивність – у вебзастосунку повинні швидко завантажуватися сторінки;
- Локалізація – у вебзастосунку має бути українська та англійська локалізація;
- Безпека – у вебзастосунку має бути забезпечена конфіденційності персональних даних користувачів.

Перейдемо до обгрунтування вибору технологій та інструметів для розробки.

1.4 Вибір технологій та інструментів для розробки вебзастосунку Make a Job

Для розробки платформи необхідно вибрати набір технологій, який забезпечить високу продуктивність, масштабованість та зручність обслуговування, а вибір правильних інструментів розробки вплине на забезпечення ефективного процесу створення, тестування, розгортання та супроводу вебзастосунку [5].

Для створення API було обрано ASP.NET Core. ASP.NET – це безкоштовна платформа з відкритим кодом розроблена компанією Microsoft [6]. Основними перевагами цього фреймворку є:

- висока продуктивність;
- кросплатформеність;
- велика кількість документації;
- інтеграція з екосистемою Microsoft, що пришвидшує результат розгортання вебзастосунку.

Для роботи з ASP.NET потрібні певні інструменти [7]. Чудовим рішенням буде .NET CLI [8]. Це кросплатформений набір інструментів для створення, збірки, запуску та публікації додатків .NET. Це інструмент з яким зручно працювати через термінал а також у команді з використанням різних операційних систем [9].

Для розробки візуальної частини було вибрано Angular [10]. Основною відмінністю від більшості технологій створення вебзастосунків є те що він є фреймворком а не бібліотекою для роботи з DOM. У Angular є CLI, що допомагає генерувати певні компоненти та модулі а також тестувати застосунок. У Angular є система модулів, що дозволяє чітко групувати компоненти, що дуже добре впливає на масштабованість [11], [12], [13].

Для зберігання даних обрано MSSQL Server, популярна СУБД (система управління реляційними базами даних) від компанії Microsoft [14]. Це доволі продуктивна система, яка легко масштабується та за допомогою SQL Server Machine Learning Services підтримує велику кількість мов. MSSQL Server також входить до екосистеми Microsoft, що було ключовою причиною вибору цієї СУБД [15], [16]. Для зручності роботи з MSSQL Server обрано DBeaver. Це графічний SQL клієнт, який допомагає в адмініструванні базами даних [17]. DBeaver підтримує такі бази даних як MySQL, MariaDB, PostgreSQL, SQLite [18]. Інтерфейс цієї програми можна побачити на рисунку 1.3.

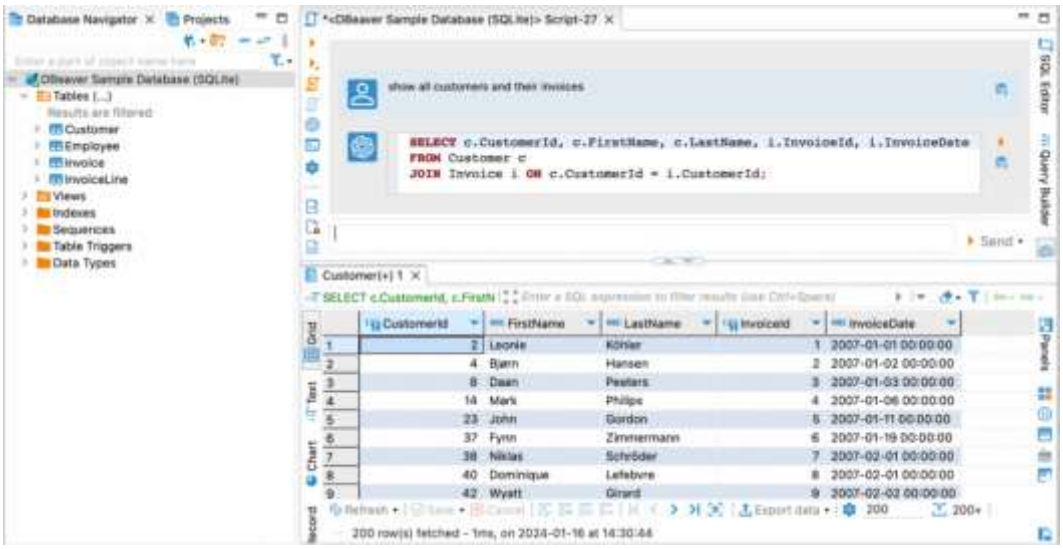

Рисунок 1.3 – Інтерфейс програми DBeaver

Важливо також обрати систему для зберігання чутливої інформації, зберігання секретів напряму у коді та репозиторіях, знижує рівень безпеки. Для вирішення проблеми чудово пі дійде Vault від компанії hashicorp. Ця програма зберігає паролі, ключі шифрування, сертифікати та токени, що дозволить ізолювати чутливу інформацію від коду, чим збільшить надійність застосунку [19].

Останній інструмент – це IDE (Integrated Development Environment). Для роботи з ASP.NET та Angular чудово пі дійде Visual Studio Code (VS Code), інтерфейс якої можна побачити на рисунку 1.4.

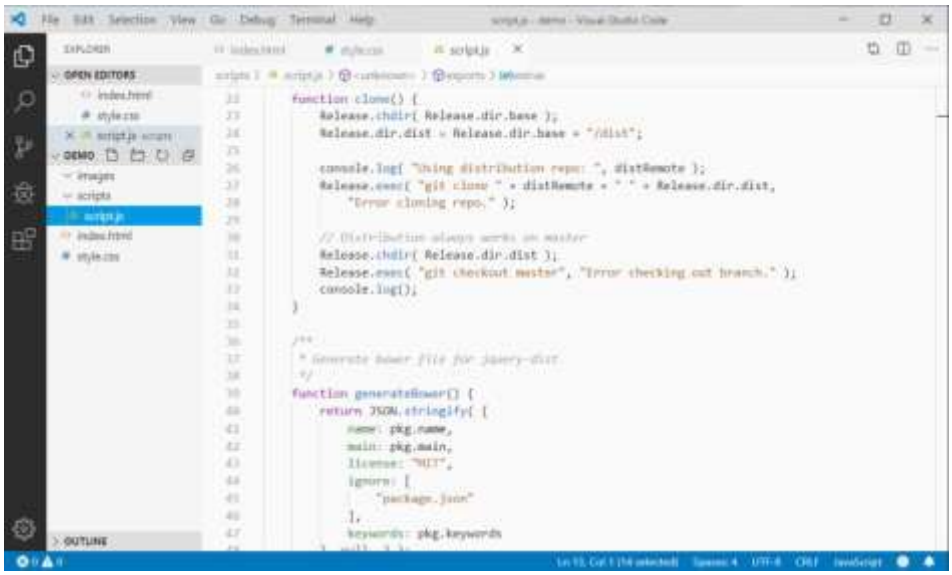

Рисунок 1.4 – Інтерфейс Visual Studio Code

Visual Studio Code, продукт Microsoft, який завдяки розширенням чудово працює з C#, ASP.NET та Angular через що у ньому буде зручно розробляти як і серверну так і клієнтську частину [20].

1.5 Висновки до першого розділу

У першому розділі проаналізовано тематичну область, пов'язану з ринком побутових послуг, зокрема взаємодію між замовниками тимчасових завдань та виконавцями, які шукають додаткові можливості для заробітку. Було виявлено, що існує зростаючий попит на цифрові рішення, які спрощують процес пошуку відповідного фахівця для виконання конкретного завдання, а також допомагають виконавцям швидко та зручно знаходити клієнтів.

На основі аналізу популярних світових платформ, таких як Handy, TaskRabbit та Airtasker, було визначено їхні переваги, функціональні особливості та недоліки. Зокрема, основними перешкодами для їх впровадження на українському ринку є відсутність локалізації, складність реєстрації, високі комісії та недостатня увага до безпеки користувачів. Врахування цих обмежень стало основою для формування вимог до нашого власного вебдодатку.

В результаті було сформульовано концепцію створення вебдодатку Make a Job, який буде орієнтований на український ринок, матиме інтуїтивно зрозумілий інтерфейс, сучасні механізми безпеки, підтримку української мови та можливість швидкого пошуку виконавців у межах конкретного регіону. У розділі також обґрунтовано вибір технологій та інструментів для реалізації проєкту з урахуванням сучасних стандартів розробки, масштабованості та гнучкості архітектури.

РОЗДІЛ 2. ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА РОЗРОБКА ВЕБЗАСТОСУНКУ MAKE A JOB

2.1 Проектування архітектури вебзастосунку

Вибір архітектури є ключовим етапом проектування вебзастосунку, від цього залежить, які сильні та слабкі сторони матиме наш застосунок. Архітектури більшості сучасних вебдодатків можна поділити на два типи [21].

Монолітна архітектура – це архітектура, яка передбачає що усі компоненти (front-end, backend та база даних) знаходяться в одному місці та розробляються як єдина програма, і розгортання зазвичай відбувається на одному сервері, структура якого зображена на рисунку 2.1

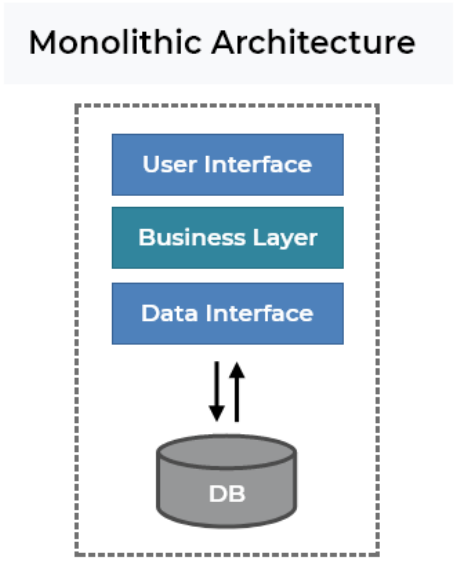

Рисунок 2.1 – Схема монолітної архітектури

Такий підхід має декілька переваг. Додатки з єдиною кодовою базою, розробляються швидше, адже розробникам легко розібратися як компоненти взаємодіють між собою. Розгортати застосунок з такою архітектурою набагато простіше, адже зазвичай прийдеться розгортати тільки один компонент що також робить запуск проекту набагато дешевшим [22].

Але цей підхід має декілька недоліків. Масштабування застосунку часто потребує зміни усього додатку, це може потребувати багатьох ресурсів якщо потрібно змінити лише частину системи. Також проблемою є впровадження нових технологій, адже використання бібліотек та фреймворків в одній частині застосунку може вплинути на інші.

Клієнт-серверна архітектура – це найпопулярніший вид побудови програмних систем, особливе вебзастосунків [23]. Основна концепція моделі полягає в тому, що функціонал розділяється між двома частинами: клієнтом та сервером, структура якого зображена на рисунку 2.2.

Клієнт – це програма з якою безпосередньо взаємодіє користувач. Клієнт відповідає за формування запитів і відображення інформації.

Сервер – це програма, що обробляє запити надіслані клієнтом та надсилає інформацію назад, де вона обробляється та відображається. Зазвичай на сервері знаходиться уся бізнес логіка, та компоненти для взаємодії з базою даних [24].

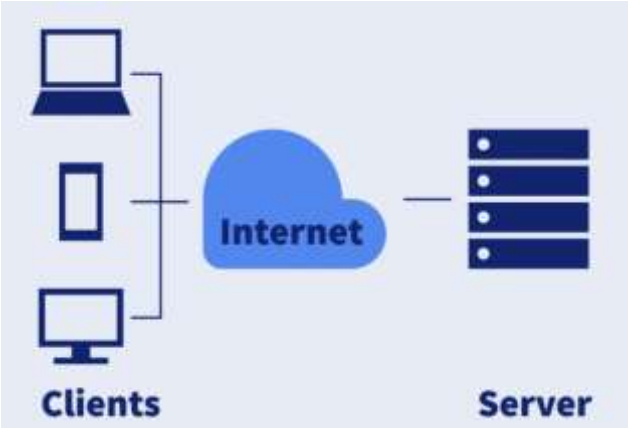

Рисунок 2.2 – Клієнт-серверна архітектура

Використання даної архітектури значно покращує масштабованість застосунку, компоненти можна оновляти незалежно один від одного. Це також дозволяє працювати декільком командам паралельно заздалегідь домовившись про API [25].

У процесі розробки вебзастосунку Make a Job. Було вирішено використовувати клієнт-серверну архітектуру, як основну модуль побудови

системи. Такий вибір зумовлений рядом переваг, які особливо важливі для проекту з потенційним розширеннями функціоналу та можливістю масштабування [26]. Також важливо зазначити що використання клієнт-серверної архітектури підвищує рівень безпеки даних, користувач не має прямого доступу до даних, що зменшує ризик витоку.

2.2 Розробка серверної частини

Для розробки сервера обрано трирівневу архітектуру, яка передбачає розділ застосунку на три компоненти: Data layer, Business layer та Presentation layer, як показано на рисунку 2.3 [27].

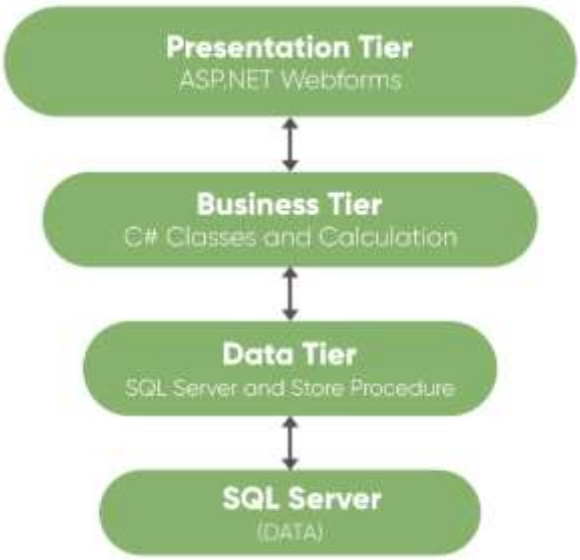

Рисунок 2.3 – .NET 3-рівнева архітектура

У застосунку Make a Job було вирішено створити ще один додатковий рівень – Domain Layer. Цей рівень призначений для зберігання моделей. Виділення нового рівня дозволить досягнути “чистішої архітектури”, адже логіка, пов’язана зі структурою даних не змішується з логікою опрацювання запитів [28].

2.2.1 Domain layer

Domain layer є основою серверної частини вебзастосунку Make a Job. На цьому рівні описуються об’єкти предметної області у вигляді сутностей, які відображають основні концепції системи. Ці об’єкти реалізовані у вигляді класів які містять відповідні властивості (поля) та за потреби, допоміжну поведінку (методи, валідації).

У цьому рівні визначено основні моделі, які відображають ключові об’єкти застосунку, такі як:

- Модель Address відповідає за зберігання детальної інформації про фізичні адреси. Вона включає такі поля, як місто вулиця, район, що дозволяє точно вказувати місцезнаходження для оголошень
- Bid представляє пропозицію подану виконавцем на певне оголошення (Job). Вона зберігає деталі пропозиції, такі як запропонована ціна та термін виконання.
- Category використовується для класифікації оголошень про роботу за основними галузями. Вона допомагає організувати вакансії та полегшує пошук для користувачів.
- Модель Image відповідає за зберігання посилань або метаданих, пов’язаних із зображеннями, які можуть бути прикріплені до оголошення та профілів користувачів.
- Job є основною моделлю, яка представляє собою оголошення про вакансію або проєкт, створений замовником. Вона містить усі необхідні деталі про роботу, такі як опис, вимоги, місце розташування та бюджет, будучи центральним елементом функціональності сервісу;
- Модель PersonalInfo призначена для зберігання додаткових особистих даних користувача, які не є критично важливими для автентифікації. Вона включає дату народження стать, докладніші контактні дані та посилання на портфоліо, що дозволяє відокремити чутливу інформацію.

- Модель Review дозволяє користувачам залишати відгуки та оцінки про виконану роботу або співпрацю. Вона фіксує оцінку та текстовий коментар, що сприяє формуванню репутації та довіри у системі.
- SubCategory слугує для подальшої деталізації класифікації оголошень про роботу, поділяючи основні Category на більш вузькі напрямки. Це покращує точність пошуку та навігації по вакансія для користувачів.
- User є центральною моделлю у системі, що представляє зареєстрованого користувача. Вона містить ключові дані для ідентифікації, автентифікації та авторизації, такі як контактна інформація, дані для входу та роль у системі.
- Модель UserToken відповідає за зберігання різних типів токенів, пов’язаних з користувачем, крім оновлення сесії. Це можуть бути токени для скидання пароля, підтвердження електронної пошти та інші одноразові службові токени, що забезпечують безпеку облікового запису.

Окрему роль на цьому рівні відіграють об’єкти передавання даних (DTO). Їх головне завдання – забезпечити безпечну та ефективну передачу даних між різними шарами застосунку, а особливо між API та бізнес-логікою. DTO не містить бізнес-логіки, але структурують дані необхідні для конкретних операцій.

2.2.2 Data access layer

Data access layer відповідає за взаємодію із зовнішнім джерелом даних – базою даних, що використовується у вебзастосунку Make a Job. Основним завданням цього рівня є забезпечення зручного та безпечного способу зчитування, запису, оновлення та видалення даних без порушення інкапсуляції та незалежності бізнес-логіки від зберігання.

Для взаємодії з БД використовується контекс, реалізований за допомогою технології Entity Framework Core. EF Core це об’єктно-реляційний мапер (ORM) розроблений компанією Microsoft, основне призначення якого це відображення таблиць на колекції даних, відстежування змінних, трансляції запитів створених за допомогою LINQ (Language integrated query) у SQL запити [29].

У проекті Make a Job для взаємодії з базою даних використовується клас MAJContext, який наслідується від базового класу DBContext, що дозволяє скористатися вбудованим механізмом зв’язку між об’єктами .NET і таблицями бази даних.

```
public class MAJContext : DbContext{
    public MAJContext(DbContextOptions<MAJContext> options)
        : base(options){}
```

Контекст містить властивості типу DbSet<TEntity>, які відповідають таблицям бази даних і надають доступ до операцій над ними. Наприклад для таблиці категорій у базі даних використовується колекція з раніше створеною моделлю:

```
public virtual DbSet<Category> Categories { get; set; }
```

Завдяки DbSet можна виконувати стандартні CRUD-операції для відповідної сутності. Подібним чином у контексті визначені й інші колекції: Users, Jobs, Bids тощо.

Для більш гнучкого налаштування схеми бази даних використовується механізм Fluent API, який дозволяє описати структуру таблиць, ключі, зв’язки між сутностями та інші обмеження у методі OnModelCreating, що показано у лістингу 2.1.

Лістинг 2.1 – Конфігурація моделей

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Bid>()
        .HasKey(x => x.Id);

    modelBuilder.Entity<Bid>()
        .HasOne(x => x.Job)
        .WithMany(x => x.Bids)
        .HasForeignKey(x => x.JobId);

    modelBuilder.Entity<Bid>()
        .HasOne(x => x.Performer)
```

```
        .WithMany()
        .HasForeignKey(x => x.PerformerId);
```

Для ізоляції доступу до бази даних та забезпечення зручної і зрозумілої структури роботи з даними в проекті реалізовано шаблон Repository [30].

Цей підхід дозволяє винести логіку зчитування, зберігання та оновлення даних із MAJContext в окремі класи, які відповідають за доступ до конкретних сутностей. Це розділення підвищує модульність, повторне використання коду та полегшує розробку та підтримку вебзастосунку.

Основую слугує інтерфейс який визначає основні операції які є загальні для усіх сутностей.

Лістинг 2.2 – Інтерфейс IRepository

```
public interface IRepository<TEntity> where TEntity : IEntity
{

    Task<TEntity?> GetByIdAsync(int id);
    Task<TEntity?>  FindFirstAsync(Expression<Func<TEntity,  bool>>
prediacate);
    Task<IEnumerable<TEntity>> GetAllAsync();
    Task<IEnumerable<TEntity>>
FindByConditionAsync(Expression<Func<TEntity, bool>> prediacate);

    Task<TEntity?> AddAsync(TEntity entity);
    Task<TEntity?> UpdateAsync(TEntity entity);

    Task DeleteAsync(Expression<Func<TEntity, bool>> predicate);

    Task<PagedResult<TEntity>> GetPagedAsync(PagedRequest
pagedRequest,Expression<Func<TEntity, bool>> prediacate);
}
```

На основі цього інтерфейсу було створено абстрактний клас BaseRepository<TEntity>, який реалізує загальні методи взаємодії з контекстом бази даних, код якого можна побачити у додатку А.

Основні методи, такі як GetByIdAsync, AddAsync, UpdateAsync, реалізовані у відповідності до стандартного CRUD-функціоналу. А також додано підтримку фільтрації та посторінкової вибірки (pagination) для оптимальної роботи з великими обсягами даних.

Після реалізації узагальненого абстрактного класу було створено конкретні реалізації репозиторіїв для кожного сутності предметної області. Для кожного класу також визначено відповідний спеціалізований інтерфейс, який наслідує базовий IRepository. Наприклад для сутності категорії було створено інтерфейс:

```
public interface ICategoryRepository : IRepository<Category>
```

Використання спеціалізованих інтерфейсів дозволяє: ізолювати логіку роботи з конкретними сутностями, забезпечити зручну інтеграцію з механізмом впровадження залежностей, полегшити тестування і розширювати поведінку додаючи специфічні методи для окремих сутностей.

2.2.3 Business login layer

Рівень бізнес-логіки відповідає за реалізацію основних правил, обмежень та операцій, що визначають поведінку системи Make a Job. Саме на цьому рівні відбувається обробка даних, які надходять з користувацького інтерфейсу або з бази даних, відповідно до предметної області.

Центральним компонентом на цьому рівні являються сервіси. На відміну від репозиторіїв де функціонал об’єднується для роботи з конкретною сутністю, то сервіси згруповують логіку за функціональними областями системи. Наприклад JobService відповідає за обробку запитів , пов’язаних не лише з сутністю Job, а й з Bid, оскільки обидві моделі взаємодіють у контексті створення та виконання завдань.

Для реалізації бізнес-логіки в вебзастосунку Make a Job було створено окремі сервіси кожен з яких реалізує функціональність певної області застосунку. Кожен сервіс є звичайним класом який інкапсулює логіку взаємодії з репозиторіями, перевірки вхідних даних, обробки помилок та підготовки

відповіді. Кожен сервіс реалізує одним або декілька інтерфейсів, що спрощує інтеграцію з DI-контейнером.

Одним з прикладів є AuthService, що зображений у додатку Б, який відповідає за авторизацію та автентифікацію користувача. Він реалізує інтерфейс IAuthService, наведений у лістингу 2.3, що визначає усі потрібні методи.

Лістинг 2.3 – Інтерфейс IAuthService

```
public interface IAuthService: IUserService
{
    public Task<UserDTO?> LoginAsync(Credentials credentials);
    public Task<UserDTO?> RegisterAsync(RegisterModel register);
         public    Task<UserDTO?>    UpdateRefreshTokenAsync(string
email,string token, DateTime time);

    public Task<bool> IsEmailExistAsync(string email);

     public Task<UserDTO?> ChangePasswordAsync(ChangePasswordModel
changePasswordModel);
      public Task<UserDTO?> ResetPasswordAsync(ResetPasswordModel
resetPasswordModel);
    public Task<UserTokenDTO?> CreateResetTokenAsync(string email);

    public Task<bool> ConfirmEmailAsync(string token);

    public Task<TfaData?> GetTfaDataAsync(string email);
    public Task<string?> ResetTfaKeyAsync(string email,string key);
    public Task<string?> GetTfaKeyAsync(string email);
    public Task SetTfaEnabledAsync(string email,bool enabled);
}
```

Сервіси мають певну структуру: при створенні AuthService у конструкторі за допмогою механізму DI інжектяться всі необхідні залежності, зокрема:

- IUserRepository, IUserTokenRepository – для доступу до даних користувача пов’язаних з автентифікацією та авторизацією;
- IPasswordHasher – для безпечного зберігання та верифікації паролів;
- IEmailService – для надсилання листів;
- IMapper – для перетворення моделей.

Публічні методи реалізовані в сервісах підпорядковуються певним правилам, що забезпечують узгоджуваність та якісь коду:

- Вхідні та вихідні дані передаються у вигляді DTO, що підвищує безпеку та ізоляцію внутрішньої структури моделі від зовнішнього середовища;
- Усі методи повинні бути асинхронними, щоб забезпечити масштабованість та ефективність використання ресурсів, особливо при роботі з базами даних чи іншими зовнішніми сервісами;
- .Для передачі результату операції, включно з інформацією про помилки або повідомленнями слід використовувати тип Result з бібліотеки FluentResult, що дозволить централізовано обробляти помилки та будувати зрозумілі відповіді.

Для зручності та стандартизації перетворення моделей сутностей та DTO було використано AutoMapper. Ця бібліотека дозволяє автоматизувати мапінг між різними типами даних, що дозволяє зменшити кількість коду та помилок пов’язаних з копіюванням даних [31]. Для роботи даної бібліотеки потрібно налаштувати Mapper, що робиться за допомогою створення нового класу профілю – MapperProfile, показаний у лістингу 2.4, де вказується правила перетворення між об’єктами. У цьому класі вказується, які типи можна взаємно перетворювати , а також за потреби задаються спеціальні правила мапінгу для окремих властивостей.

Лістинг 2.4 – Частина налаштувань MapperProfile

```
public class MapperProfile : Profile
{
    public MapperProfile()
    {
        CreateMap<User, UserDTO>().ReverseMap()
            .ForAllMembers(opt => opt.Condition((src, dest,
srcMember) => !IsDefaultValue(srcMember)));

        CreateMap<UserToken, UserTokenDTO>().ReverseMap()
            .ForAllMembers(opt => opt.Condition((src, dest,
srcMember) => !IsDefaultValue(srcMember)));
```

Цей профіль був зареєстрований у контейнері залежностей і автомачно використовується у сервісах через інтерфейс IMapper, що дозволяє легко конвертувати данні:

```
Var userDTO = _mapper.Map<UserDTO>(user);
```

Окрім перетворення даних сервісам, потрібен зручний спосіб створення умов для фільтрації під час розробки пошуку з багатопараметричною фільтрацією. Щоб не дублювати код, а також забезпечити динамічне формування запитів до репозиторіїв, у проєкті використовується допоміжний клас під назвою ExpressionBuilder. Метод GetPredicate класу ExpressionBuilder приймає список фільтрів, де кожен фільтр складається з назви поля, оператора порівняння та значення. З цього списку формуються умови, з яких створюється предикад за допомогою операції логічного множення. Метод GetPredicate повертає вирази типу Expression<Func<T, bool>> з критеріїв фільтрації. Їх легко використовувати в запитах LINQ, тому логіку фільтрації можна легко налаштувати.

Увеб-застосунку було реалізовано службу MailGunService для надсилання електронних листів, яка реалізує інтерфейс IEmailService. Інтерфейс визначає методи SendEmail, SendConfirmationEmail та SendResetPasswordEmail, кожен з яких генерує запит до API Mailgun з відповідними параметрами: одержувач, тема та HTML-вміст електронного листа.

Реалізація MailGunService використовує інкапсульовану логіку для генерації HTTP-запитів до служби електронної пошти, включаючи надсилання посилань з токенами підтвердження реєстрації або скидання пароля. Усі параметри конфігурації (ключ API, домен, адреса сервера) зберігаються окремо – у Vault або файлах конфігурації – що забезпечує гнучкість і безпеку.

2.2.4 Presentation layer (API)

Presentation layer або рівень представлення відповідає за взаємодію користувача із застосунком. У контексті вебзастосунку Make a Job цей рівень реалізовано у вигляді REST API, що дозволяє зовнішнім клієнтам (frontend або мобільним застосункум) надсилати запити та отримувати відповіді. Основним завданням цього рівня є прийом HTTP-запитів, виклик відповідної бізнес-логіки та повернення результатів у стандартизованому форматі.

У рамках рівня представлення особливу увагу приділено механізмам автентифікації та авторизації, які використовуються для захисту даних користувача та контрольованого доступу до ресурсів програми. У вебзастосунку Make a Job ці механізми реалізовано за допомогою токен-орієнтованої автентифікації через JSON Web Token, що є сучасним і широко підтримуваним стандартом [32].

Після успішної авторизації сервер генерує два типи токенів: токен доступу та токен оновлення. Токен доступу або access токен використовується клієнтами API для аутентифікації запитів і має обмежений термін дії, після чого може бути оновлений за допомогою токена оновлення. Підхід з використанням двох токенів забезпечує оптимальний баланс між безпекою та зручністю.

Для інтеграції JWT-автентифікації в проекті було використано механізм JwtBearer, який налаштовується через систему аутентифікації ASP.NET Core.

Лістинг 2.5 – Налаштування автентифікації

```
var secret = Encoding.UTF8.GetBytes(jwtSettings.SecretKey);

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
JwtBearerDefaults.AuthenticationScheme;
})
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = jwtSettings.Issuer,
            ValidAudience = jwtSettings.Audience,
            IssuerSigningKey = new SymmetricSecurityKey(secret)

        };
    });

builder.Services.AddAuthorization();
```

Цей компонент автоматично перевіряє дійсність підпису, термін дії токена, та дійсність ключа. Таким чином, кожеш запит до захищених маршрутів API має бути авторизований дійсним токеном доступу у заголовку Authorization. Після чого операція дозволяється, якщо користувач має на це право.

Взаємодія користувача з API передбачає, що кожен endpoint, доступний тільки авторизованим користувачам, містить атрибут [Authorize]. Це дозволяє легко відокремити публічно доступні та обмежені маршрути. Внутрішньо контролери можуть отримувати інформацію про користувача через об’єкт User, зокрема через вимоги, наприклад:

```
User.FindFirstValue(ClaimTypes.NameIdentifier);
```

Для генерації токенів було розроблено окремий сервіс під назвою JwtTokenService. Метод GenerateAccessToken створює підписаний JWT із переданими claims, тоді як GenerateRefreshToken генерує криптографічно безпечний випадковий рядок для продовження сесії, а метод GetPrincipalFromExpiredToken дозволяє безпечно витягувати вимоги навіть із простроченого токена доступу, що є важливим для оновлення токена.

Крім звичайної автентифікації за допомогою електронної пошти та паролю, вебдодаток Make a Job включає двофакторну аутентифікацію з використанням TOTP – протоку, який генерує одноразові паролі на основі поточного часу. Цей механізм забезпечує додатковий рівень захисту облікових записів і мінімізує ризики, що виникають внаслідок компрометації пароля [33].

Під час початку процесу активації двофакторної аутентифікації (2FA) сервер повертає спеціальні посилання otpauth:// для створення конфігурації аутентифікації мобільного застосунку. Це посилання може бути використане двома способами:

- Клієнт може видати посилання для копіювання для встановлення в додаток;

- Або клієнт може згенерувати QR-код для зручного сканування, що підтримують більшість застосунків для 2FA.

У реалізації цього функціоналу на рівні API були введені такі методи:

- TfaSetup – генерує секретний ключ та посилання otpauth:// для продовження налаштування в мобільному додатку;
- TfaEnable – активує 2FA після підтвердження успіху з одноразовим кодом, згенерованим з секрету;
- TfaDisable – дозволяє користувачу вимкнути двофакторну автентифікацію;
- TfaLogin – Перевіряє TOTP-код під час авторизації.

Цей метод дозволяє реалізувати 2FA безпечно, де жодні чутливі дані не надсилаються або не обробляються поза сервером.

Після впровадження механізмів автентифікації та авторизації наступним ключовим компонентом рівня представлення є контролери. Контролери в вебдодатку Make a Job використовується як початкова точка для обробки HTTP-запитів, ініційованих клієнтськими додатками.

Архітектурно контролери не містять бізнес-логіки безпосередньо, а делегують її відповідним сервісам, що імплементують інтерфейси на рівні бізнес-логіки. Це дозволяє розділити обов’язки, що корисно для тестування, обслуговування та масштабування.

Вони також відповідають за формування HTTP-відповідей, з правильними кодами статусу, повідомленнями про помилки та результатами виконання запитів. Для викликів, де є дані, наприклад список об’єктів, вони будуть повернуті у серіалізованому форматі JSON. Також є підтримка пагінації, сортування та фільтрації пошуку, щоб клієнтські програми могли зручно працювати з великою кількістю даних.

У застосунку Make a Job реалізовано низку контролерів, кожен з яких відповідає за обробку запитів у своїй функціональній області. Такий підхід забезпечує чітку структуру API та спрощує маштабування проєкту.

Веб-застосунок включає такі контролери:

- AuthController – контролер, код якого зображений в додатку В, відповідальний за інтеграцію авторизації та автентифікації користувачів. Реалізація включає функції реєстрації, входу, оновлення токенів доступу та блок функцій 2FA.
- UserController – контролер. відповідальний за отримання інформації про користувача, основне управління обліковим записом та додаткові адміністративні дії – зміна ролі та блокування.
- PersonalInfoController – обробляє розширені персональні дані користувача, які не підпадають під питання конфіденційності і не потребують додаткового захисту, наприклад контакти, опис користувача, досвіт та інша інформації.
- JobController – ключовий контролер для взаємодії з оголошеннями про роботу. Підтримує створення, перегляд редагування, видалення завдань, а також функціонал призначення виконавців, скасування або завершення завдань.
- CategoryController – дозволяє отримати список доступних категорій, до яких можуть належати створені роботи. Це забезпечує структуризацію та зручний пошук завдань.
- BidController – реалізує можливість користувачам надсилати пропозиції до завдань. Дозволяє залишити заявку, відкликати її або переглядати список поданих заявок.
- ReviewController – обробляє створення та перегляд відгуків про виконані роботи. Це дозволяє створювати рейтинги користувачів та репутацію в системі.

Для зручності тестування та взаємодії з REST API в проєкті Make a Job було додано Swagger (OpenAPI). Swagger-інтерфейс дозволяє автоматично генерувати інтерактивну документацію для всіх доступних ендпоінтів, які визначено в контролерах. Це дозволяє розробникам і тестувальникам бачити всі доступні маршрути, їх атрибути (такі як URI, HTTP-методи, параметри та формат відповіді), а також приклади їх використання з переданими значеннями.

Swagger також має функцію, яка дозволяє надсилати запит безпосередньо з вебінтерфейсу, що дуже корисно для ручного тестування. Документація

Swagger генерується автоматично з атрибутів [ApiController], [HttpGet], [HttpPost], [Authorize] а також враховує XML-коментарі у коді, Що дозволяє підтримувати документацію актуальною без зайвих зусиль.

2.3 Розробка клієнтської частини

Клієнтська частина вебзастосунку Make a Job реалізована за допомогою Angular – фреймворку для створення односторінкових вебзастосунків. Angular забезпечує двосторонню прив’язку даних, реактивність інтеграцію зі стандартними засобами побудови форм, роутингу, валідації, роботи з HTTP та багатьох інших аспектів, які є необхідними для побудови складних фронтенд-застосунків [34].

Клієнтська сторона відповідає за графічне представлення даних, взаємодію з інтерфейсом та надсилання запитів до API-сервера через HTTP. Це включає автентифікацію, обробку форм, виведення помилок, оновлення даних у реальному часті, а також навігацію між різними розділами застосунку без перезавантаження сторінки.

Архітектура Angular-застосунку базується на модульному принципі, що дозволяє ізолювати функціональність в окремих логічних блоках. Модулі можуть мати свої власні компоненти, сервіси, шаблони, стилі та залежності. Це значно підвищує підтримуваність, запобігає повторенню коду, спрощує тестування та є ключовим компонентом для хорошої масштабованості [35].

Фізична структура Angular-застосунку реалізована відповідно до принципів логічного групування за функціональністю. У застосунку знаходиться директорія src/, яка виступає кореневою папкою для усього коду. Всередині неї структура побудована не за предметною областю, наприклад за компонентами призначеними для роботи з оголошеннями, а за технічною або функціональною ознакою, як показана на рисунку 2.4. Такий підхід дозволяє чітку розмежувати відповідальності між сервісами, модулями, конфігураціями тощо.

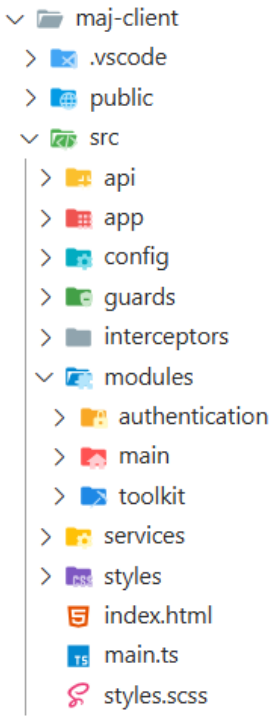

Рисунок 2.4 – Архітектура клієнтської частини застосунку

Основними компонентами є модулі, які являють собою “контейнери” для різних частин застосунку. Воно об’єднують пов’язанні компоненти, директиви та служби в цілісні блоки. У застосунку було створено декілька модулів, кожен з яких відповідає за певну функціональність клієнтської частини. Призначення створених модулей можна побачити у таблиці 2.1:

Назва модуля	**Опис функціональності**
1	**2**
MainModule	Охоплює основну функціональність застосунку. До нього входять головна сторінка, загальний макет інтерфейсу, а також елементи, що забезпечують роботу з завданнями (їх створення, перегляд), управління пропозиціями (надсилання та перегляд), функцію пошуку послуг та персональний кабінет користувача. Цей модуль є центральною частиною клієнтської логіки

Продовження таблиці 2.1

1	2
AuthenticationModule	Містить компоненти, пов'язані з автентифікацією користувача. Цей модуль включає елементи для авторизації, реєстрації та підключення двофакторної аутентифікації (2FA), забезпечуючи безпечний доступ до системи.
ToolkitModule	Допоміжний модуль, що містить часто повторювані компоненти та сервіси, які можуть бути використані в різних частинах застосунку. Він забезпечує перевикористання коду та підтримку єдиного стилю інтерфейсу.

Компоненти Angular є основними засобами для створення інтерфейсу користувача. Кожен компонент в Angular відповідає за відображення певної частини інтерфейсу та за формування його поведінки. У вебзастосунку Make a Job після визначення структури модуля було стврено велику кількість компонентів, кожен з яких реалізує окремий функціональний елемент або розділ застосунку.

Під час розробки компонентів враховувалися принципи розділення відповідальностей, повторного використання та зручності в супроводі. Наприклад для кожної ключової сторінки або елемента інтерфейсу, такого як форма створення завдання, список пропозицій, профіль користувача, був створений окремий компонент Angular. Усі елементи були логічно згруповані у межах своїх модулів.

Оскільки деякі частини програми відповідають за обробку чутливих даних або доступні лише авторизованим користувачам, під час розробки було приділено особливу увагу при впроваджені елементів безпеки на стороні клієнта.

Центральною частиною цього механізму є SecurityService, який надає можливість зберігати інформацією про авторизованого користувача в

локальному сховищі браузера, код якого можна побачити у додатку Г. Під час входу в систему сервіс оновлює відповідні дані в сховищі, а також інформує про зміну стану, використовуючи BehaviorSubject, що дозволяє іншим компонентам слідкувати за фактичним станом аутентифікації в реактивний спосіб.

Крім цього, сервіс також надає методи для отримання токену доступу, перевірки ролі користувача та отримування загального статусу аутентифікації. Це дозволяє використовувати SecurityService для контролю доступу до захищених елементів інтерфейсу, для перевірки авторизації в HTTP-перехоплювачах та guard-ах.

Для контролю доступу до конкретних маршрутів в застосунку було створення кілька guard-ів, які виступають додатковим шаром перед завантаженням сторінки. Зокрема:

- AuthenticatedGuard – надає доступ лише авторизованим користувачем. Він перевіряє, чи є збережений користувач і токен доступу, і якщо ні – перенаправляє користувача на сторінку входу.
- NoAuthenticatedGuard – навпаки, обмежує доступ до певних маршрутів, наприклад сторінки авторизації або реєстрації, для користувачів, які вже увійшли у система.

Такі guard-и забезпечують спосіб логічно сегрегувати доступ на основі статусу автентифікації, що покращує не лише безпеку, але й загальну зручність використання вебзастосунку.

Для централізованої обробки HTTP-запитів було створено два інтерсептора:

- TokenInterceptor – додає токен доступу, який зберігається в SecurityService, до заголовка кожного HTTP-запиту, що потребує авторизованих запитів. Якщо токен відсутній або недійсний, запит надсилається без нього.
- AuthInterceptor – обробляє помилки автентифікації, перенаправляючи на сторінку входу користувачів, у випадку втрати даних пов’язаних з авторизацією.

Маршрути в вебзастосунку організовані у вигляді модульного роутингу. Основні маршрути підключаються через механізм lazy loading, що дозволяє

оптимізувати швидкість завантаження застосунку та зменшити стартовий розмір бандлу – зібраного і оптимізованого набору файлів для завантаження застосунку у браузері [36]. Додаткові обмеження на доступ до окремих сторінок реалізовані за допомогою guard-ів, як показано у конфігурації маршрутизатора у лістингу 2.6.

Лістинг 2.6 – Налаштування маршрутизації модуля автентифікації

```
const routes: Routes = [
    {path: '',
        component: AuthLayoutComponent,
        children: [
            { path: ««, redirectTo: 'login', pathMatch: 'full' },
            { path: 'login', component: LoginComponent },
            { path: 'register', component: RegisterComponent },
            { path: 'reset-password', component:
ResetPasswordComponent },
            { path: 'reset-password/:token', component:
ResetPasswordComponent },
            { path: 'tfa-setup', component: TfaSetupComponent,
canActivate: [AuthenticatedGuard] },
        ]
    }
];
```

Ця конфігурація забезпечує чітке розмежування доступу до різних частин застосунку, а також сприяє підтримці чистої та масштабованої архітектури клієнтської частини [34].

Одним із ключових елементів клієнтської частини вебзастосунку є розробка зручного, функціонального та візуально привабливого інтерфейсу користувача. Для стилізації інтерфейсу в Make a Job використовувалась мова розширення CSS - SCSS (Sass), яка відкрила широкі перспективи для створення адаптивних та багатофункціональних стилів. У застосунку сформовано загальний SCSS-бандл, який містить колірні змінні, типографіку, міксини для адаптивного дизайну, функції, шаблони класів та інші загальні стилі. Цей бандл імпортується у всі компоненти, забезпечуючи єдину візуальну концепцію, швидку модифікацію та підтримку централізованого стилю. Для кожного модуля

розроблені окремі SCSS-файли з локальними стилями, що мінімізує ризик зіткнень між компонентами.

Для реалізації форм у вебзастосунку Make a Job було задіяно як ReactiveFormsModule, так і FormsModule. Ключові складні форми побудовані на основі Reactive Forms, що дало змогу розробити вкладені структури з груп, розширену валідацію та централізоване керування станом форм. У спрощених випадках використовується FormsModule з двонаправленим зв’язуванням ([(ngModel)]). Окрім цього, активно залучено інші модулі Angular: HttpClientModule – для асинхронної взаємодії з API через інтерсептори, та CommonModule – як базовий модуль з необхідними директивами. Реактивна обробка подій, керування потоками даних, обробка запитів та станів реалізовані з використанням RxJS, зокрема через BehaviorSubject, pipe, map, switchMap, catchError та інші оператори. Також на клієнтській стороні реалізована підтримка багатомовності за допомогою бібліотеки @ngx-translate/core, що дає можливість перемикати мову інтерфейсу без перезавантаження сторінки. Переклади зберігаються в JSON-файлах, а доступ до текстів відбувається через директиви translate або сервіс TranslateService. Для перекладу текстових елементів у шаблонах компонентів здійснюється за допомогою пайпу translate, наприклад:

```
<div class=«header__text-small text-small»>
      {{ 'register.subheader' | translate }}
</div>
```

Такий підхід дозволяє легко локалізувати інтерфейс без зміни логіки компонента. У застосунку реалізовано підтримку двох мов – української та англійської – з можливістю перемикання між ними у реальному часі.

2.4 Висновки до другого розділу

У цьому розділі розглянуто основні етапи та рішення, прийняті під час розробки та впровадження вебзастосунку Make a Job. Основна увага приділялася

архітектурі системи, яка була реалізована відповідно до принципів модульності, багаторівневої організації та розподілу відповідальності між окремими частинами вебзастосунку. Це забезпечує простоту обслуговування, масштабованість та надійність програмного забезпечення.

Була розроблена загальна архітектура взаємодії клієнт-сервер, яка включає чотирирівневу модель на стороні сервера:

- Доменний рівень відповідає за доменну модель, забезпечуючи цілісний огляд основних сутностей та їх взаємозв'язків;
- Рівень доступу до даних реалізує абстракцію роботи з базою даних, забезпечуючи зберігання, оновлення та вилучення інформації;
- Рівень бізнес-логіки містить логіку обробки запитів, перевірки умов, прийняття рішень на основі даних та доменних правил;
- Рівень презентації реалізує REST API, який є єдиною точкою входу для клієнтської сторони та сторонніх сервісів.

Серверна частина реалізована з використанням технологій .NET, що дозволило ефективно побудувати RESTful API з підтримкою аутентифікації, авторизації, обробки помилок, логування, валідації та документації Swagger.

Клієнтська частина додатку реалізована з використанням Angular, одного з найпоширеніших фреймворків для побудови SPA-додатків. Вона базується на модульному підході, що дозволяє розділити логіку на окремі функціональні блоки. У клієнтській частині була реалізована інфраструктура маршрутизації, використана підтримка модулів з відкладеним завантаженням, для зберігання та обробки даних використовувалися сервіси, а на клієнтському рівні були впроваджені механізми безпеки.

Особлива увага була приділена реалізації безпечної взаємодії з API, зокрема:

- Централізоване зберігання та управління станом аутентифікації за допомогою SecurityService;
- Реалізація охоронців для обмеження доступу до маршрутів відповідно до прав доступу;

- Створення HTTP-перехоплювачів для автоматичного додавання токенів до запитів і обробки помилок авторизації.

Таким чином, у другому розділі була сформована технічна основа вебдодатку, яка поєднує гнучкість клієнтської частини з надійністю і масштабованістю серверної логіки. Це створює всі необхідні передумови для подальшої розробки, тестування, впровадження та обслуговування вебзастосунку Make a Job.

РОЗДІЛ 3. РОЗГОРТАННЯ ТА ТЕСТУВАННЯ ВЕБЗАСТОСУНКУ MAKE A JOB

3.1 Конфігурування та розгортання вебзастосунку

Розгортання вебзастосунку вимагає попереднього налаштування декількох зовнішніх та внутрішніх конфігурацій, щоб забезпечити коректну роботу його основних функцій. Особливу увагу приділено сервісам, що відповідають за інтеграцію з картами, надсилання електронних листів, зберігання конфіденційної інформації, а також правильну організацію запуску обох частин застосунку [37].

Одним із найважливіших сервісів у клієнтській області вебзастосунку є платформа Google Maps. Її впровадження дозволяє додавати деякі функції, пов’язані з географічними координатами, такі як показ завдань на карті, а також автозаповнення адрес під час створення або редагування завдань користувачем.

Щоб використовувати API Google Maps, необхідно створити обліковий запис Google Cloud та новий проект. У цьому проекті було активовано декілька API: Maps JavaScript API, Places API та Geocoding API. Places API пропонує функцію автозаповнення і може бути корисним для пошуку за ключовими словами, Geocoding API, з іншого боку, повертає координати, необхідні для розміщення адреси на карті.

Після ввімкнення відповідних API в консолі Google Cloud генерується ключ API, який ідентифікує програму під час запиту картографічного сервісу. Цей ключ має бути захищений обмеженнями: дозволені домени, IP-адреси або лише певні API, що можна побачити на рисунку 3.1. У клієнтській програмі ключ можна отримати зробивши запит на сервер де для цього було створено спеціальний контролер.

Оскільки ключ API знаходиться на стороні клієнта, важливо обмежити його можливості, вказавши дозволені домени. Це запобігає зловживанню ключем сторонніми особами.

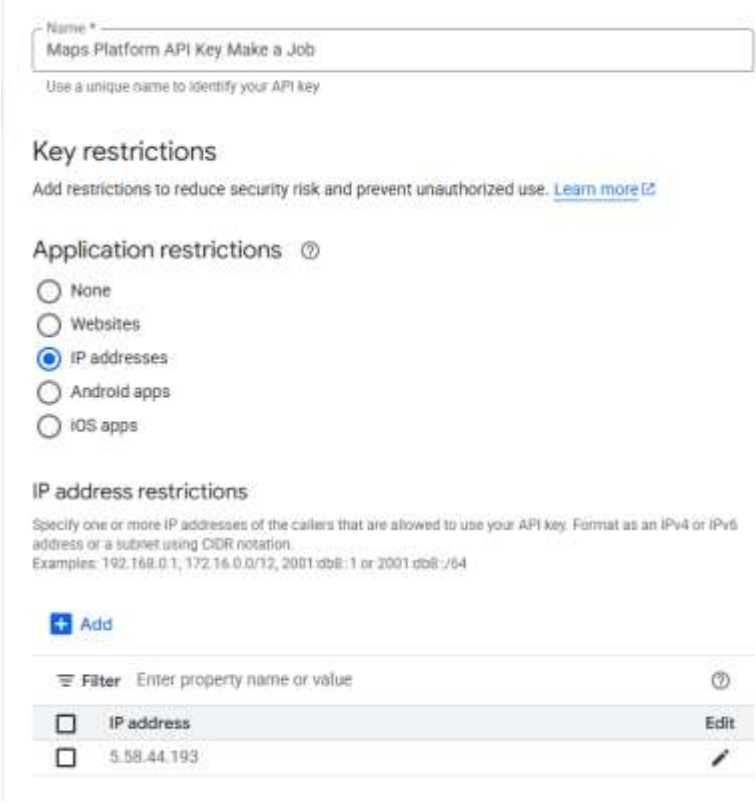

Рисунок 3.1 – Налаштування публічного ключа Google Maps

Для реалізації функції відправлення електронних листів вебдодаток було інтегровано з поштовим сервісом Mailgun, що дозволяє відправляти транзакційні повідомлення, підтвердження реєстрації, сповіщення про оновлення вакансій та інші події. Щоб забезпечити належну роботу цього сервісу, доменне ім'я makeajob.online було спочатку зареєстровано в SpaceShip, який виконує функції реєстратора доменів.

Після реєстрації домену в панелі управління SpaceShip було створено окремий піддомен mg.makeajob.online, який буде використовуватися виключно для надсилання електронних листів через Mailgun. Такий підхід рекомендується, оскільки він дозволяє відокремити діяльність електронної пошти від основного домену та спростити конфігурацію записів DNS.

У Mailgun було створено новий домен, що відповідає піддомену mg.makeajob.online. Після створення сервіс автоматично згенерував список DNS-записів, які слід додати до налаштувань домену на стороні SpaceShip. Ці записи включають:

- MX-записи, які дозволяють обробляти вхідну пошту;
- TXT-записи для аутентифікації SPF, DKIM і DMARC, які підтверджують, що Mailgun має право відправляти електронні листи від імені домену;
- CNAME-запис для відстеження відкриттів і кліків на посилання в електронних листах.

Всі необхідні записи були додані до конфігурації DNS домену через інтерфейс SpaceShip, як показано на рисунку 3.2. Потім, протягом декількох годин, Mailgun автоматично перевірив правильність налаштувань і підтвердив верифікацію домену. Після верифікації був згенерований API-ключ, який зберігається в сховищі секретів Vault і використовується серверною частиною додатка для відправки повідомлень через HTTP-запити до API Mailgun.

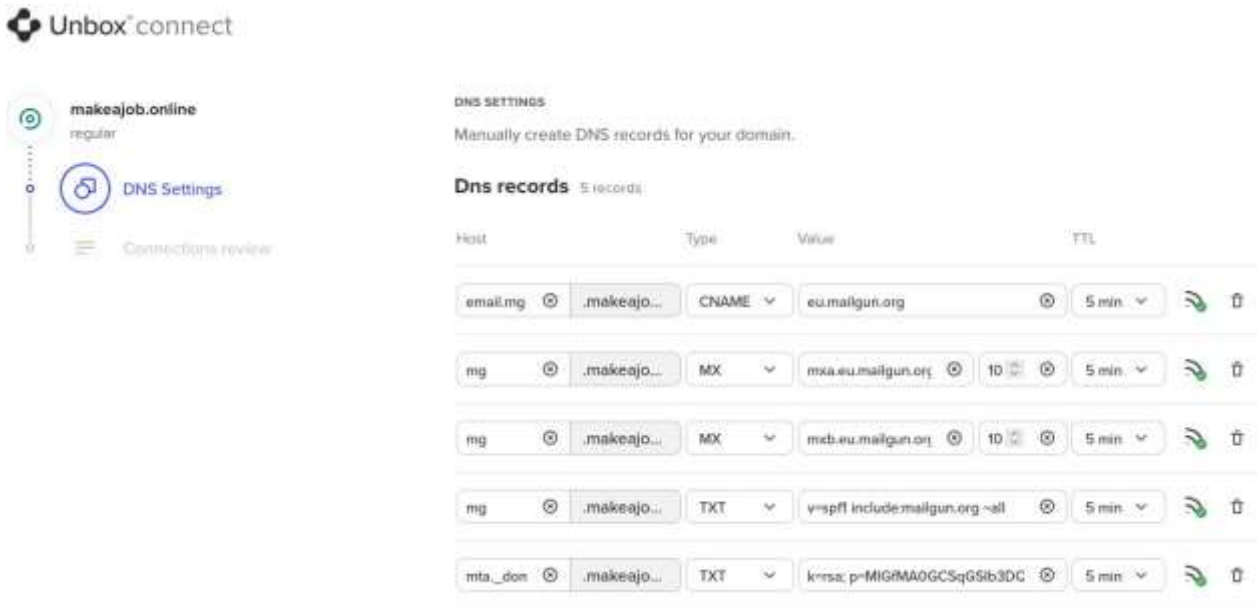

Рисунок 3.2 – Налаштування записів DNS

У додатку, зокрема, коли користувач реєструється або змінює пароль, на вказану адресу електронної пошти надсилається підтверджувальний лист. Зміст листа був створений у вигляді шаблонів, а сам процес надсилання був реалізований в окремому сервісі поштової логіки в бізнес-шарі серверної

частини. Використання Mailgun забезпечило високу доставку листів, підтримку логування та можливість масштабування системи сповіщень у майбутньому.

Для безпечного зберігання чутливої інформації, зокрема API-ключів сторонніх сервісів, токенів доступу та інших конфіденційних даних, у застосунку було використано інструмент HashiCorp Vault. Це спеціалізоване сховище секретів, яке дозволяє централізовано управляти доступом до критичних даних, забезпечуючи високу гнучкість і захист на рівні інфраструктури.

Після встановлення Vault було виконано початкову ініціалізацію сховища, під час якої система згенерувала ключі розпечатування та токен адміністратора кореневого рівня. Ці ключі були збережені в безпечному місці поза сховищем, відповідно до принципів безпечного адміністрування. Vault було налаштовано для локального виконання з файловим сховищем, що спрощує використання під час етапу розробки та тестування додатка.

Для зберігання даних у Vault було створено окремий простір імен або шлях (make-a-job/), в якому секрети розміщуються у вигляді пар ключ-значення. Наприклад, були додані записи з API-ключем для служби Mailgun, API Google Maps, а також інші змінні, пов'язані з конфігурацією середовища, такі як секрет JWT або параметри автентифікації.

Приклад встановлення ключів:

```
vault kv put make-a-job/mailgun apiKey=mailgun-api-key
vault kv put secret/maps apiKey=your-google-maps-api-key
```

Для доступу до цих значень на сервері додатка використовувався Vault CLI або HTTP API. Залежно від середовища (розробка, тестування або виробництво), додаток зчитував відповідні секрети та ініціалізував необхідні служби під час запуску. Щоб зменшити ризик витоку інформації, прямий доступ до Vault був обмежений лише сервером, що працює в безпечному середовищі.

Крім зберігання секретів, Vault дозволяє налаштовувати політики доступу. Для цього була створена окрема політика, яка дозволяє тільки читати необхідні ключі певним службам, без права перезапису або видалення. Це відповідає

принципу мінімальних привілеїв і мінімізує ризики можливого компрометації окремих компонентів.

Після завершення налаштування основних служб наступним кроком було розгортання обох частин програми в локальному середовищі для тестування та перевірки правильності роботи всіх інтеграцій. Це дозволяє виявити можливі помилки, перевірити взаємодію клієнта з сервером і забезпечити правильну роботу функціоналу перед розгортанням у віддалених середовищах.

Серверна частина додатка реалізована за допомогою ASP.NET Core, тому її запускали за допомогою стандартних інструментів .NET CLI або безпосередньо з середовища розробки, такого як Visual Studio Code. Після запуску сервера на вказаному порту (наприклад, https://localhost:5001) автоматично активується попередньо налаштована документація API, створена за допомогою Swagger, частину якої можна побачити на рисунку 3.3. Swagger дозволяє не тільки переглянути опис усіх доступних маршрутів API, але й протестувати їх роботу, передаючи параметри, токени автентифікації та переглядаючи відповіді в зручному інтерфейсі.

Рисунок 3.3 – Swagger документація контролера AuthController

Клієнтська частина, реалізована в Angular, запускається окремо за допомогою Angular CLI з командою ng serve. Після цього додаток стає доступним за адресою http://localhost:4200. Для забезпечення коректної роботи з API-сервером у конфігурації клієнта була вказана базова адреса для HTTP-запитів до серверу з урахуванням портів, що використовуються в локальному середовищі.

Під час локального запуску також була перевірена функціональність функцій, що використовують зовнішні сервіси, такі як Google Maps для автозаповнення та відображення карти, а також можливість відправки електронних листів через Mailgun. При цьому всі необхідні ключі та параметри конфігурації зчитувалися з Vault, що додатково дозволило перевірити правильність роботи з секретами в безпечний спосіб.

Локальний запуск обох частин додатка дозволив нам протестувати повний цикл взаємодії між клієнтом, сервером і сторонніми сервісами, а також забезпечити узгодженість авторизації, валідації та обробки запитів у режимі реального часу. Це важливий етап перед розгортанням у виробничому середовищі.

3.2 Розробка модульних тестів

Модульні тести є важливим компонентом розробки програмного забезпечення, що дозволяє перевіряти окремі частини коду (функції, методи, класи) на правильність роботи в ізоляції від інших компонентів системи. Основна мета таких тестів – переконатися, що логіка програми працює саме так, як передбачалося, а при внесенні змін або рефакторингу можна швидко виявити потенційні помилки. Написання Unit тестів сприяє підвищенню покриття коду, що сприяє підвищення покриття коду, що дозволяє своєчасно виявляти помилки на ранніх етапах розробки [38].

При написанні Unit тестів слід дотримуватися кількох основних принципів. По-перше, кожен тест повинен перевіряти одну конкретну поведінку

або функціональність, забезпечуючи зрозумілість і легкість розуміння тестового випадку. По-друге, тести повинні бути ізольовані від зовнішніх залежностей, таких як база даних, мережеві служби або файлові системи. Для цього сучасні тестові фреймворки широко використовують техніку мокінгу – заміну реальних залежностей спеціальними симуляторами, що імітують їх поведінку. Це дозволяє контролювати середовище виконання тестів і забезпечує повторюваність результатів.

У наведеному в статті прикладі тесту використовувалася бібліотека FakeItEasy, яка служить для створення моків і фейків. Вона дозволяє дуже зручно і гнучко задавати очікувану поведінку залежностей, а також перевіряти, чи були викликані необхідні методи і в якій кількості. Для організації самих тестів використовувалася фреймворк xUnit – популярний інструмент в .NET-спільноті, який надає простий і зрозумілий синтаксис для написання і виконання тестів.

Особливо варто відзначити роботу з базою даних. Для тестування служби автентифікації використовувався варіант InMemory Entity Framework Core. Це дозволяє виконувати операції з даними без підключення до реальної бази даних, що значно підвищує швидкість і стабільність тестів.

Для імітації роботи бази даних в модульних тестах використовується база даних InMemory, реалізована в Entity Framework Core. Такий підхід дозволяє працювати з повнофункціональним контекстом даних, але без необхідності підключення до реальної бази даних, що значно спрощує налаштування тестового середовища і збільшує швидкість виконання тестів. База даних InMemory забезпечує зберігання даних в пам'яті під час виконання тестів, що дозволяє тестувати логіку роботи з репозиторіями та контекстом EF Core в ізольованому середовищі. Це дозволяє ефективно тестувати операції додавання, пошуку, оновлення або видалення даних, не побоюючись вплинути на реальні дані і без необхідності підтримувати окремий сервер для тестування бази даних.

Проект також використовує атрибут [assembly: InternalsVisibleTo(«MAJ.Tests»)], який відкриває внутрішні класи та методи основного проекту для збірки з тестами. Це дозволяє тестувати не тільки

публічний API, але й внутрішню логіку, не порушуючи принципів інкапсуляції та не роблячи всі елементи публічними. Такий підхід допомагає досягти більш повного покриття коду тестами.

Для розробки та тестування в цьому проекті використовувався редактор Visual Studio Code, який надає зручні інструменти для відстеження та виконання модульних тестів. Завдяки інтеграції з популярними фреймворками для тестування, такими як xUnit, VS Code дозволяє легко переглядати список тестів, запускати їх окремо або всі одразу, а також бачити результати в зручному вигляді прямо у середовищі розробки, на рисунку 3.4 можна побачити результати тестування частини сервісів.

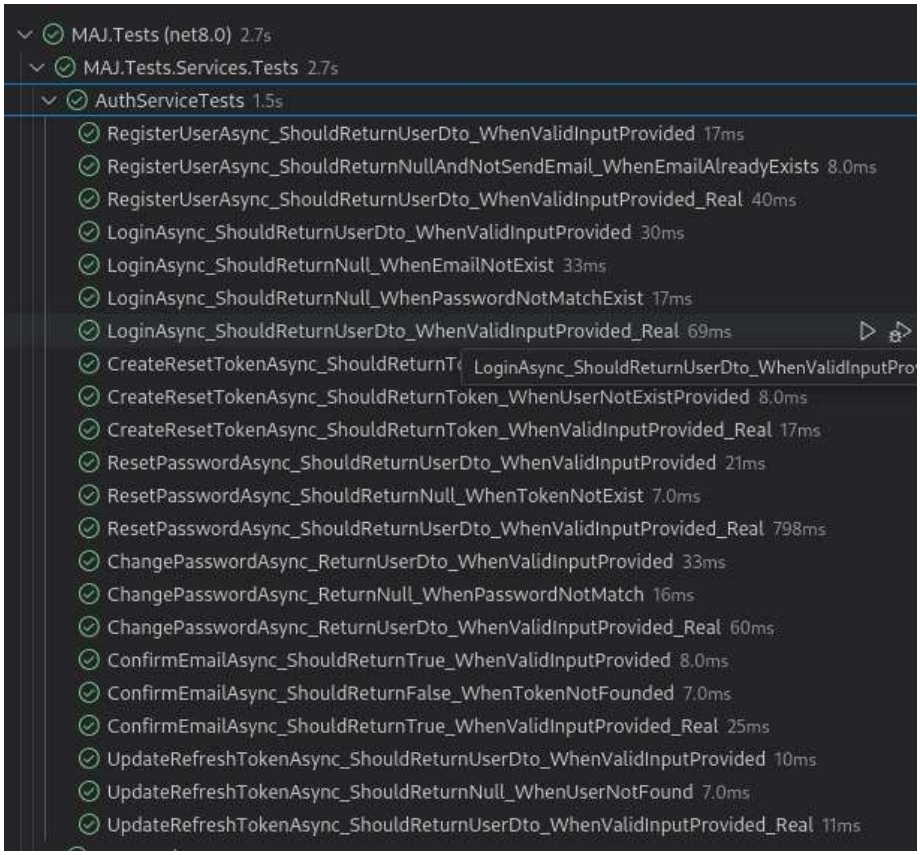

Рисунок 3.4 – Результати тестування сервісу автентифікації

Крім того, розширення VS Code підтримують автоматичне повторне виконання тестів при зміні коду, що значно покращує ефективність налагодження та прискорює процес розробки.

3.3 Тестування основних функцій застосунку

Після завершення реалізації базової функціональності вебзастосунку було проведене функціональне тестування, щоб перевірити коректність роботи, в наближених до реальних умовах. Тестування здійснювалося на рівні кінцевого користувача та API системи, що дало змогу перевірити поведінку основних сценарії, зокрема: авторизацію, реєстрацію, активацію та використання 2FA, створення оголошень та пропозицій, також вибір виконавців, та перевірка роботи усіх зовнішніх сервісів.

Перший етап тестування передбачав перевірку функціональності реєстрації користувачів та надсилання підтверджувального листа. Після заповнення реєстраційної форми система повинна автоматично створити новий обліковий запис та надіслати підтверджувальний лист із токеном. Для тестування цього процесу було використано сервіс TempMail, який надає тимчасову адресу електронної пошти та дозволяє відстежувати отримання повідомлень.

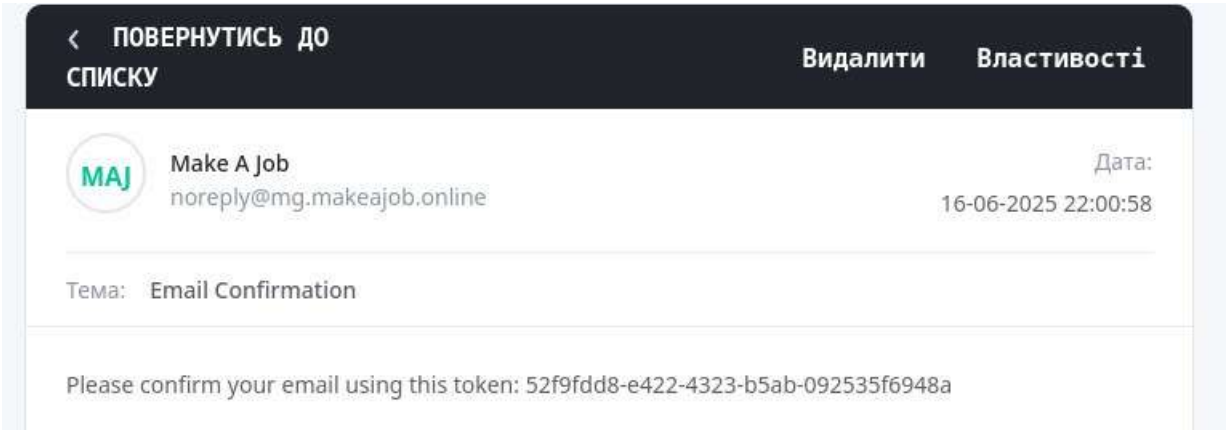

Рисунок 3.5 – Надісланий лист

Як видно з зображення 3.5, система успішно надіслала електронний лист з темою «Підтвердження електронної адреси» від імені служби Make A Job (noreply@mg.makeajob.online). Текст повідомлення містить токен підтвердження, що свідчить про правильну роботу механізму генерації та

доставки електронних листів після реєстрації. Це підтверджує правильність налаштувань служби електронної пошти та логіки, відповідальної за створення та надсилання токенів.

Наступним кроком було тестування процесу авторизації користувача. Після успішної реєстрації та підтвердження електронної адреси користувач увійшов у систему, використовуючи свою електронну адресу та пароль. Потім система автоматично запропонувала йому налаштувати двофакторну автентифікацію (2FA), яка є додатковим механізмом безпеки.

На зображені 3.6 можна побачити згенерований QR код для активації 2FA, який було просканово додатком Google Authenticator. Після того як було додано обліковий запис у додатку, застосунок згенерував одноразовий код, який було введено у поле нижче, що дозволило завершити налаштування двофакторної аутентифікації. Цей крок показав, що інтеграція TOTP працює коректно, а також підтвердив наявність додаткового рівня захисту при вході в обліковий запис.

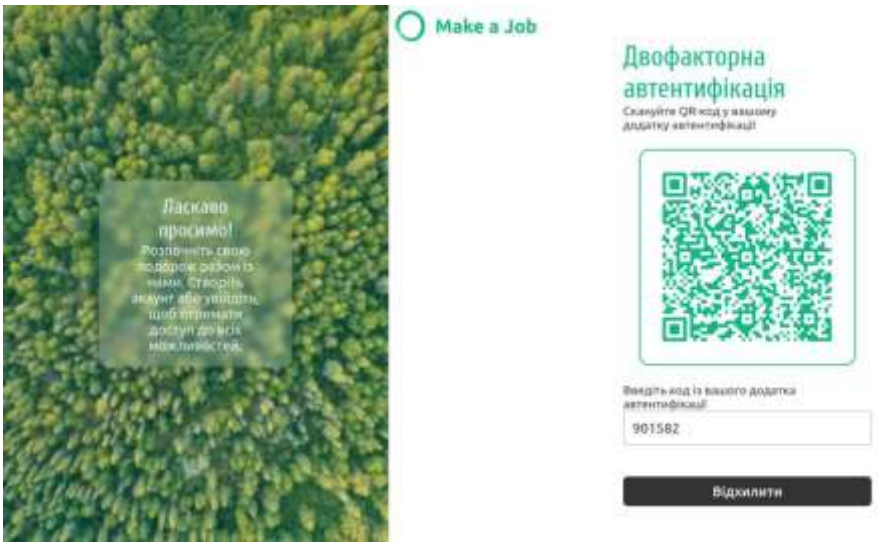

Рисунок 3.6 – Сторінка активації 2FA

Після облікового запису та аутентифікації було протестовано функції публікації оголошень. У цьому сценарії користувач заповнює форма, яка складається з основних деталей оголошення – короткий зміст, опис, суму оплати,

категорію та дати виконання. Одним із ключових елементів на цьому етапі є інтеграції з Google Maps API, а саме функції автозаповнення, що дозволяє легко ідентифікувати адресу, вводячи лише частину назви\

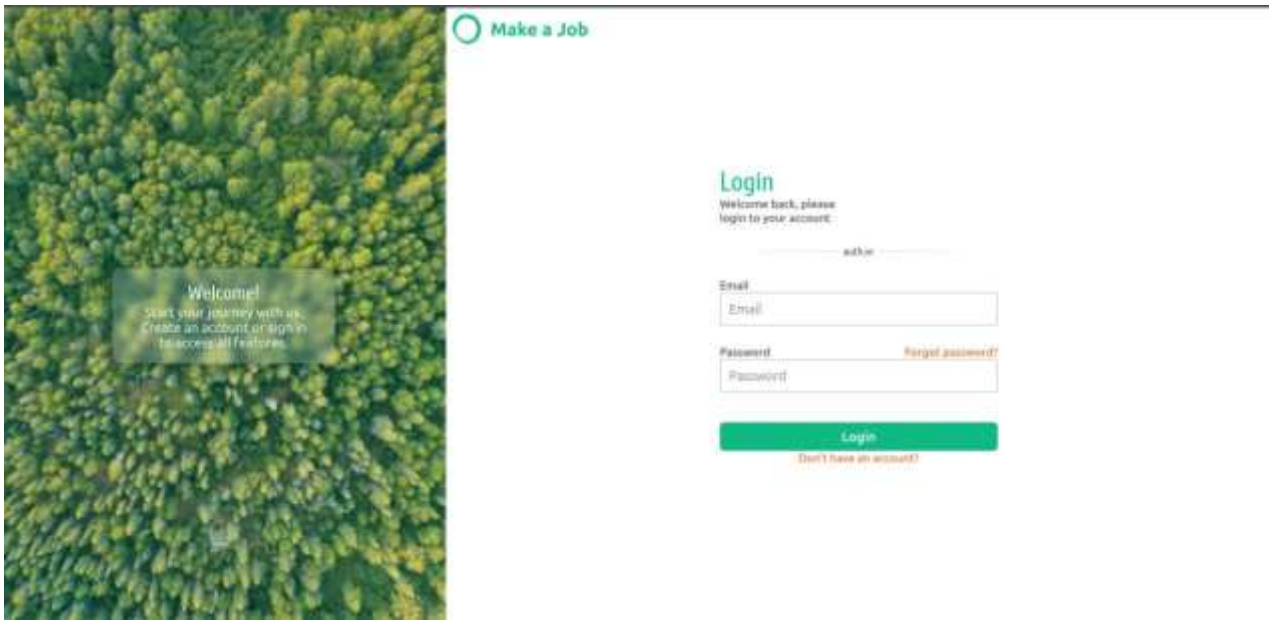

Рисунок 3.7 – Перевірка роботи Google map API

Під час тестування було підтверджено, що поле з адресою коректно взаємодіє з сервісом Google, що видно з рисунку 3.7, пропонуючи релевантні варіанти для автозаповнення. Це підвищує зручність використання та мінімізує ймовірність помилки при введені адреси Таким чином, перевірка створення оголошення засвідчила стабільну роботу клієнтської частини та успішну інтеграцію із зовнішнім API.

На наступному етапі тестування було виконано вхід в систему під іншим користувачем, щоб перевірити функціональність перегляду та взаємодії з оголошеннями. Після переходу на сторінку конкретного завдання було перевірено правильність відображеної інформації – назву, опис, місцезнаходження, дату публікації, винагороду та дані замовника, що зображено на рисунку 3.8. Інформація відображалася правильно, що підтверджує правильність роботи системи передачі та зберігання даних.

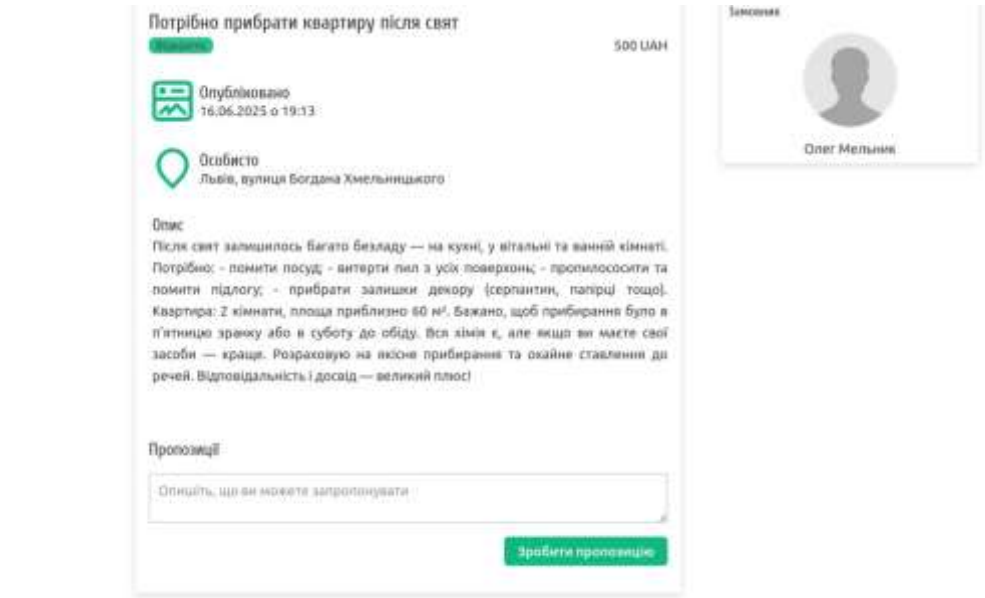

Рисунок 3.8 – Відображення оголошення

Крім того, було перевірено можливість залишити пропозицію щодо оголошення. Відповідне текстове поле та кнопка відправлення працювали належним чином – повідомлення було успішно надіслано.

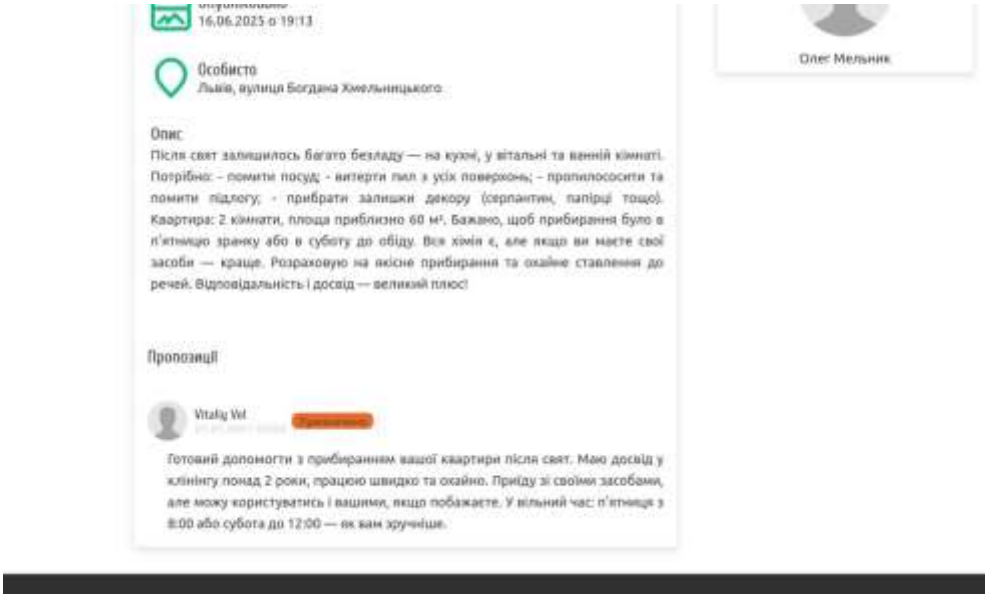

Рисунок 3.9 – Підтверджена пропозиція

Після залишення пропозиції було здійснено вхід під користувача, який створив оголошення. Далі виконано перехід до сторінки оголошення, де було підтверджено прийняття пропозиції виконавця. Це підтверджує коректну роботу

основних функцій узгодження пропозицій між замовником та виконавцем. Відповідна кнопка спрацювала без помилок, а інформація про прийняту пропозицію відобразилася у деталях оголошення, що показано на рисунку 3.9.

3.4 Висновки до третього розділу

Цей розділ присвячений розгортанню, налаштуванню та тестуванню вебдодатку Make a Job, який було розроблено в рамках дипломного проекту. Під час фази розгортання було підготовлено серверне та клієнтське середовища, налаштовано інтеграцію із сторонніми сервісами (зокрема API Google Maps та поштовим сервером) та забезпечено безпеку автентифікації за допомогою двофакторної автентифікації.

Під час написання модульних тестів була перевірена логіка окремих компонентів, що дозволило виявити та усунути локальні помилки до етапу інтеграції. Основні функції додатка були вручну протестовані відповідно до ключових сценаріїв: реєстрація, підтвердження електронної пошти, вхід, налаштування двофакторної аутентифікації, створення та перегляд оголошень, надсилання та підтвердження пропозицій.

Результати тестування підтвердили стабільну роботу основних механізмів системи та відповідність реалізованої функціональності вимогам. Таким чином, реалізацію додатка можна вважати успішною як з точки зору функціональності, так і зручності використання.

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Долікарська допомога при ураженні електричним струмом

Ураження електрикою належить до типових травматичних випадків, котрі можуть призвести до серйозних наслідків для здоров’я людини. Електричний струм здатний викликати опіки тканин, збої у функціонуванні серця, системи дихання, нервової системи та інших життєво важливих органів. Тому надання негайної та грамотної долікарської підтримки має ключове значення для порятунку життя потерпілого й послаблення негативних впливів ураження. Від швидкості реагування та правильності вчинків свідків події чи тих, хто поруч, залежить не лише виживання людини, а й перспективи її подальшого одужання [39].

Найпершим та найважливішим заходом, коли мова йде про надання допомоги при ураженні електричним струмом, є якнайшвидше припинення його впливу на потерпілого. Варто усвідомлювати, що торкатися людини, котра під впливом струму, без попереднього відключення електропостачання, надзвичайно небезпечно для того, хто рятує. Отже, спершу необхідно вимкнути живлення, від'єднати дроти або відсунути постраждалого від джерела струму використовуючи сухі ізоляційні матеріали, наприклад, дерев'яні палиці або вироби з пластмаси, як показано на рисунку 4.1.

Рисунок 4.1 – Рятування при ураженні струмом

Ні в якому разі не можна намагатися звільнити людину, торкаючись до неї голими руками, поки не буде припинено подачу струму. Після того, як вплив електричного струму припинено, першочергово слід оцінити стан постраждалого. Найважливіше – перевірити, чи притомний він, як дихає та чи прощупується пульс. Якщо людина без свідомості, не дихає або не має Необхідно якомога швидше розпочати непрямий масаж серця [40].

Надавлювати слід на нижню частину грудної кістки, занурюючи на глибину від 5 до 6 сантиметрів. Частота натискань має бути в межах 100-120 разів на хвилину з мінімальними перервами. Після кожного натискання грудна клітка має повністю вирівнюватися, не можна спиратися на неї. Масаж потрібно проводити лише на твердій основі. Якщо використовується штучне дихання, після кожних 30 натискань на грудну клітку необхідно зробити 2 вдихи. Якщо штучна вентиляція легень недоступна, потрібно продовжувати натискати на грудну клітку без зупинок.

У ситуаціях, коли постраждалий при тямі, першочергово потрібно його заспокоїти, не дозволяти робити рухи та якнайшвидше викликати медичну допомогу. Необхідно оглянути постраждалого, звернувши увагу на видимі пошкодження: опіки, пошкодження шкірного покриву, судоми, порушення у рухах або чутливості кінцівок. Опіки, що виникли внаслідок дії електричного струму, здатні бути глибокими, а інколи і прихованими, тому, навіть за відсутності візуальних ознак ушкоджень, потрібно забезпечити спокій та турботливе ставлення до потерпілого.

За наявності опіків слід негайно накласти стерильну пов'язку, категорично уникаючи самостійної обробки ран водою чи будь-якими іншими речовинами, якщо немає конкретних вказівок від фахівців. Першочерговою задачею є ретельний контроль за диханням та пульсом потерпілого до прибуття лікарів швидкої допомоги.

Варто розуміти, що контакт з електрикою здатен призвести не тільки до локальних пошкоджень, а й до тяжких збоїв у функціонуванні внутрішніх органів. Електричний струм, проходячи крізь тіло, здатен викликати фібриляцію

серця – безладні скорочення серцевого м'яза, що є загрозливим для життя станом. Отже, після надання першої допомоги потерпілого обов’язково потрібно доставити до медичного закладу для проведення подальшого обстеження та лікування [41].

4.2 Загальні вимоги безпеки з охорони праці для користувачів ПК

Створення вебзастосунків, зокрема таких як система пошуку та замовлення побутових послуг, передбачає тривалу працю користувача з персональним комп’ютером (ПК). Така діяльність має певні небезпеки для здоров’я робітника, особливо при порушенні санітарно-гігієнічних чи ергономічних вимог. Тому дотримання нормативів охорони праці є необхідним для створення безпечних умов праці програмістів, дизайнерів, тестувальників та інших спеціалістів, задіяних у процесі розробки програмного забезпечення.

Основним нормативним документом, що регулює безпеку праці при роботі з електронно-обчислювальними машинами, є державні санітарні норми та правила «Гігієнічні вимоги до організації роботи з візуальними дисплейними терміналами ЕОМ» (ДСанПіН 3.3.2.007-98), затверджені постановою Головного державного санітарного лікаря України. Цей документ охоплює широкий спектр вимог, спрямованих на запобігання несприятливому впливу факторів виробничого середовища та трудового процесу на працівників. Він встановлює стандарти для освітлення, мікроклімату, параметрів робочого місця, а також режиму праці та відпочинку [42].

Робоче місце користувача ПК має бути організоване таким чином, щоб забезпечувати можливість періодичної зміни пози. Робочий стіл повинен мати достатній розмір для зручного розміщення обладнання та документів. Висота столу має дозволяти розташувати руки так, щоб лікті були зігнуті приблизно під кутом 90 градусів, з опорою на підлокітники крісла або стіл. Клавіатуру слід розміщувати на відстані 10-30 см від краю столу, що забезпечує підтримку передпліч. Монітор необхідно встановлювати перед користувачем, на відстані

60-70 см від очей. Верхній край екрана має бути приблизно на рівні очей або трохи нижче, щоб уникнути зайвого нахилу голови та шиї.

Природне освітлення повинно максимально використовуватись, а вікна необхідно обладнати жалюзі або шторами для регулювання рівня освітленості та запобігання прямим відблискам на екрані монітора. Світильники повинні забезпечувати рівномірне, розсіяне світло без сліпучого ефекту та відблисків. Їхнє розташування має виключати пряме попадання світла в очі користувача. Оптимальна освітленість робочих поверхонь з моніторами становить 300-500 люкс.

Оптимальні параметри температури, вологості та швидкості руху повітря регламентуються відповідними державними санітарними нормами мікроклімату виробничих приміщень. Зазвичай, температура має підтримуватися в діапазоні 22-24°C у холодний період та 23-25°C у теплий. Відносна вологість має бути в межах 40-60%, а швидкість руху повітря не повинна перевищувати 0.1-0.2 м/с [43].

Рекомендується запроваджувати регулярні регламентовані перерви протягом робочого дня. Для тих, хто працює за комп'ютером постійно, передбачаються 15-хвилинні перерви кожні 45-60 хвилин безперервної роботи або 10-хвилинні перерви після кожної години. Ці паузи варто використовувати для зміни виду діяльності, виконання легких фізичних вправ, прогулянок або спеціальних вправ для очей [44].

4.3 Висновки до четвертого розділу

Цей розділ присвячений аналізу питань безпеки життєдіяльності та основ охорони праці. Під час дослідження було розглянуто принципи надання долікарської допомоги у критичних ситуаціях, зокрема при ураженні електричним струмом. Особливу увагу приділено першочерговим діям, які включають безпечне припинення дії електричного струму на потерпілого, оцінку

його стану (свідомість, дихання, пульс) та, за необхідності, негайне проведення серцево-легеневої реанімації.

Також було проаналізовано загальні вимоги безпеки з охорони праці для користувачів персональних комп'ютерів, що є вкрай актуальним для розробників та потенційних користувачів системи. Були визначені ключові аспекти ергономіки робочого місця, оптимального освітлення та мікроклімату, а також необхідність дотримання режиму праці та відпочинку для запобігання фізичній та психологічній перевтомі. Ці вимоги були сформовані згідно з положеннями «Гігієнічних вимог до організації роботи з візуальними дисплейними терміналами ЕОМ» (ДСанПіН 3.3.2.007-98).

Дотримання вимог безпеки як у надзвичайних ситуація, так і під час повсякденної роботи з ПК є важливою умовою збереження здоров’я та продуктивності праці. Усвідомлення ризиків і вчасне реагування дозволяють мінімізувати їхній негативний вплив на людину та забезпечити комфортне і безпечне середовище діяльності.

ВИСНОВКИ

Під час виконання виконання кваліфікаційної роботи реалізовано повний цикл розробки сучасного вебдодатку Make a Job, призначеного для пошуку виконавців побутових завдань та надання послуг у конкретному регіоні. Сформульованою метою було створення зручного, безпечного та функціонального сервісу, що поєднує потреби двох основних груп користувачів – замовників та виконавців. Аналіз тематичної області показав актуальність проблеми та наявність попиту на подібні рішення в Україні. Також були розглянуті та оцінені існуючі аналоги, визначені їхні обмеження, що дозволило сформувати власні вимоги до функціональності та якості майбутнього додатка.

В рамках проектної частини розроблена архітектура вебдодатку, яка базується на чіткому розділенні клієнтської та серверної частин з використанням сучасних підходів до організації багаторівневої системи. Сервер був реалізований на базі платформи .NET з використанням архітектурного підходу Domain-Driven Design та принципів REST API. Це забезпечило високу гнучкість, масштабованість та простоту обслуговування. Були реалізовані функції аутентифікації, авторизації, управління користувачами, створення та модерації оголошень, обробки заявок тощо. Особлива увага була приділена безпеці: додано підтримку двофакторної аутентифікації, перевірку ролей користувачів, а також централізовану обробку запитів.

Клієнтська частина створена з використанням Angular. Вона реалізувала зручний і зрозумілий інтерфейс з підтримкою адаптивності, механізмів маршрутизації, реактивного управління станом, захисту маршрутів за допомогою охоронців, інтеграції з Google maps для вибору місця розташування завдань, а також взаємодії з API через перехоплювачі HTTP-запитів. Вся взаємодія між клієнтом і сервером здійснюється за принципами безпечного обміну даними.

Практична частина охоплювала налаштування зовнішніх сервісів (Google Maps API, Mailgun, HashiCorp Vault), налаштування середовищ зберігання

секретів, поштових повідомлень, перевірку підписаних доменів, а також локальне розгортання додатка з доступом до API через документацію Swagger. Особлива увага приділялася тестуванню, як модульному, так і ручному. Це дозволило нам підтвердити функціональність основних функцій, таких як реєстрація, вхід, управління обліковим записом, створення завдань, подання запитів тощо.

ПЕРЕЛІК ДЖЕРЕЛ

1 Handy. URL: https://www.handy.com (дата звернення: 16.04.2025).

2 TaskRabbit. URL: https://www.taskrabbit.com (дата звернення: 16.04.2025).

3 Airtasker. URL: https://www.airtasker.com/ (дата звернення: 16.04.2025).

4 Functional requirements non-functional requirements. Geeksforgeeks. URL: https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/ (дата звернення: 16.04.2025).

5 What is ASP.NET? What are its pros and cons?. Vidhema. URL: https://vidhema.com/blog/what-is-asp-net-what-are-its-pros-and-cons (дата звернення: 16.04.2025).

6 ASP.NET Core Advantages and Disadvantages. REDWERK. URL: https://redwerk.com/blog/asp-net-core-pros-and-cons/ (дата звернення: 16.04.2025).

7 Backend Frameworks List: Choosing the Right One. Daily. URL: https://daily.dev/blog/backend-frameworks-list-choosing-the-right-one (дата звернення: 16.04.2025).

8 Dotnet CLI. Medium. URL: https://medium.com/@ouzatl_88083/dotnet-cli-8799f7eed1ee (дата звернення: 16.04.2025).

9 .NET CLI overview. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/core/tools/ (дата звернення: 16.04.2025).

10 What are the best backend development frameworks? The ultimate comparison. TechWings. URL: https://techwings.com/blog/backend-frameworks-ultimate-comparison (дата звернення: 16.04.2025).

11 Top 7 Frontend Frameworks to Use in 2025: Pro Advice. Roadmap. URL: https://roadmap.sh/frontend/frameworks (дата звернення: 16.04.2025).

12 Порівнюємо React, Angular і Vue – найпопулярніші бібліотеки й фреймворки у 2022 році. Dou. URL: https://dou.ua/forums/topic/39933/ (дата звернення: 16.04.2025).

13 What is Angular?. Angular. URL: https://v17.angular.io/guide/what-is-angular#what-is-angular (дата звернення: 16.04.2025).

14 Understanding the Significance of Microsoft SQL Server: Advantages, Disadvantages, and Impact in AI and Data. LinkedIn. URL: https://www.linkedin.com/pulse/understanding-significance-microsoft-sql-server-ai-ooq8c/ (дата звернення: 16.04.2025).

15 Microsoft SQL Server: Advantages and Best Practices for Technical Corporate Decision Makers. VirtualDBA. URL: https://virtual-dba.com/blog/microsoft-sql-server-advantages-and-best-practices/ (дата звернення: 16.04.2025).

16 Tutorials for SQL Server. Microsoft Learn. URL: https://learn.microsoft.com/en-us/sql/sql-server/tutorials-for-sql-server-2016?view=sql-server-ver17 (дата звернення: 16.04.2025).

17 Universal Database Tool. DBeaver. URL: https://dbeaver.io (дата звернення: 16.04.2025).

18 Засіб для роботи з базами даних DBeaver. Blogger. URL: https://ukr-technologies.blogspot.com/2020/03/dbeaver.html (дата звернення: 16.04.2025).

19 Manage Secrets & Protect Sensitive Data. Hashicorp. URL: https://developer.hashicorp.com/vault#what-is-vault (дата звернення: 16.06.2025).

20 Your code editor.. Visual Studio Code. URL: https://code.visualstudio.com (дата звернення: 16.04.2025).

21 Building the Future- Choosing Between Client-Server, Monolithic, and Microservices.. Medium. URL: https://medium.com/@kulanthamuditha66/building-the-future-choosing-between-client-server-monolithic-and-microservices-2a194677f8eb (дата звернення: 16.04.2025).

22 Основні типи архітектури програмного забезпечення. Art of business analysis. URL: https://www.artofba.com/uk/post/main-types-of-software-architecture#viewer-m63al1878 (дата звернення: 16.04.2025).

23 Client-Server Architecture Explained with Examples, Diagrams, and Real-World Applications. Medium. URL: https://medium.com/nerd-for-tech/client-server-

architecture-explained-with-examples-diagrams-and-real-world-applications-407e9e04e2d1 (дата звернення: 16.04.2025).

24 Готович В.А., Ралік І.Р. Програмне забезпечення на основі клієнт-серверної архітектури для обліку реалізації товарів в торгівлі. Збірник тез доповідей XI Міжнародної науково-практичної конференції молодих учених та студентів «Актуальні задачі сучасних технологій». Тернопіль, 7-8 грудня 2022 р. С. 126.

25 Client Server Model. Scaled Academy. URL: https://www.interviewbit.com/blog/client-server-model/ (дата звернення: 16.04.2025).

26 Готович В.А., Граб Д.В. Актуальність задачі розробки модуля інформаційної системи для управління ІТ-проєктами. Збірник тез доповідей XIII Міжнародної науково-практичної конференції молодих учених та студентів «Актуальні задачі сучасних технологій». Тернопіль, 11-12 грудня 2024 р. С. 426-427.

27 The benefits of a three-layered application architecture. VFunction. URL: https://vfunction.com/blog/the-benefits-of-a-three-layered-application-architecture/ (дата звернення: 16.04.2025).

28 Use A 3-Tier Architecture With C#. Kens Learning Curve. URL: https://kenslearningcurve.com/tutorials/use-a-3-tier-architecture-with-c/#The_Hidden_Layer_The_Domain (дата звернення: 16.04.2025).

29 Step-by-Step Guide to Entity Framework in .NET. Medium. URL: https://medium.com/@aleksej.gudkov/step-by-step-guide-to-entity-framework-in-net-da5fcda99e11 (дата звернення: 16.04.2025).

30 Implementing the Repository Pattern in C# and .NET. Medium. URL: https://medium.com/@kerimkkara/implementing-the-repository-pattern-in-c-and-net-5fdd91950485 (дата звернення: 16.04.2025).

31 What is AutoMapper?. Automapper. URL: https://docs.automapper.org/en/latest/Getting-started.html (дата звернення: 16.04.2025).

32 JWT Authentication in .NET 8: A Complete Guide for Secure and Scalable Applications. Medium. URL: https://medium.com/@solomongetachew112/jwt-authentication-in-net-8-a-complete-guide-for-secure-and-scalable-applications-6281e5e8667c (дата звернення: 16.04.2025).

33 Two-Factor Authentication in ASP.NET Core MVC. Medium. URL: https://karaoz-onr.medium.com/two-factor-authentication-in-asp-net-core-mvc-26b4218a1e0b (дата звернення: 16.04.2025).

34 Готович В.А., Козак В.І. Дослідження варіантів проектування інтерфейсу користувача в інформаційних інтерактивних аналітичних панелях. Збірник тез доповідей XII Міжнародної науково-практичної конференції молодих учених та студентів «Актуальні задачі сучасних технологій». Тернопіль, 6-7 грудня 2023 р. С. 385-386.

35 Understanding Angular Modules: A Comprehensive Guide. Medium. URL: https://codewithpawan.medium.com/understanding-angular-modules-a-comprehensive-guide-180100403396 (дата звернення: 16.04.2025).

36 Angular Routing. Medium. URL: https://medium.com/@jstify.community/angular-routing-d25992afe8c7 (дата звернення: 16.04.2025).

37 Готович В.А., Мачужак А.В. Застосування методології CI/CD для автоматизації процесів тестування та розгортання програмного забезпечення. Збірник тез доповідей XI Міжнародної науково-практичної конференції молодих учених та студентів «Актуальні задачі сучасних технологій». Тернопіль, 7-8 грудня 2022 р. С. 131-132.

38 Гайдар А.В., Готович В.А. Розробка платформи для перевірки знань шляхом тестування. Збірник тез доповідей IX науково-технічної конференції ТНТУ імені Івана Пулюя «Інформаційні моделі, системи та технології». Тернопіль, 8-9 грудня 2021 р. С. 37.

39 Долікарська допомога при ураженні електричним струмом. Безпека життєдіяльності. URL: https://web.posibnyky.vntu.edu.ua/fmbt/berezyuk_bezpeka_zhittyediyalnosti/76.htm (дата звернення: 19.04.2025).

40 Інструкція №102-ОП Правила надання першої медичної допомоги при ураженні електричним струмом. На Урок. URL: https://naurok.com.ua/instrukciya-102-op-pravila-nadannya-persho-medichno-dopomogi-pri-urazhenni-elektrichnim-strumom-248392.html?utm_source=chatgpt.com (дата звернення: 19.04.2025).

41 Правила надання першої допомоги при ураженні електричним струмом. Правила надання першої допомоги при ураженні електричним струмом. URL: https://lviv.dsp.gov.ua/pravyla-nadannia-pershoi-dopomohy-pry-u/10723/ (дата звернення: 19.04.2025).

42 Охорона праці та ПК, як безпечно працювати на персональному комп'ютері. КПІ ім. Ігоря Сікорського. URL: https://opcb.kpi.ua/?p=3590&utm_source=chatgpt.com (дата звернення: 19.04.2025).

43 Інструкція з охорони праці при роботі на персональному комп'ютері. Uteka.ua. URL: https://services.uteka.ua/ua/publication/zrazky-34-trudovi-vidnosyny-ta-oplata-pratsi-138-instrukciya-po-oxrane-truda-pri-rabote-na-personalnom-kompyutere-obrazec (дата звернення: 19.04.2025).

44 Інструкція з охорони праці при роботі з ПК. Науково-методичний центр професійно-технічної освіти. URL: http://nmc.ptu.org.ua/wp-content/uploads/2024/05/instrukcziya-z-o.p.-pry-roboti-na-pk.pdf (дата звернення: 19.04.2025).

ДОДАТКИ

Додаток А

Лістинг класу BaseRepository

```
public abstract class BaseRepository<TEntity> : IRepository<TEntity>
where TEntity : IEntity
{
    protected readonly MAJContext _context;

    public BaseRepository(MAJContext context)
    {
        _context = context;
    }

    private DbSet<TEntity> entities;
    protected    DbSet<TEntity>    Entities    =>    entities    ??=
_context.Set<TEntity>();

    public virtual async Task<TEntity?> GetByIdAsync(int id)
    {
        return await Entities
            .FindAsync(id)
            .ConfigureAwait(false);
    }

    public virtual async Task<IEnumerable<TEntity>> GetAllAsync()
    {
        return await Entities
            .ToListAsync()
            .ConfigureAwait(false);
    }

    public      virtual      async      Task<IEnumerable<TEntity>>
FindByConditionAsync(Expression<Func<TEntity, bool>> predicate)
    {
        return await Entities.Where(predicate)
            .ToListAsync()
            .ConfigureAwait(false);
    }

    public virtual async Task<TEntity?> AddAsync(TEntity entity)
    {
        try
        {
            var entry = Entities.Add(entity);

            await SaveChangesAsync();

            return entry.Entity;

        }
        catch (Exception ex)
        {
            return null;
```

```
        }
    }
    public virtual async Task<TEntity?> UpdateAsync(TEntity entity)
    {
        try
        {
            var entry = Entities.Update(entity);

            await SaveChangesAsync();

            return entry.Entity;

        }
        catch (Exception ex)
        {
            return null;
        }
    }
    protected async Task SaveChangesAsync()
    {
        await _context.SaveChangesAsync();
    }

    public          virtual          async          Task<TEntity?>
FindFirstAsync(Expression<Func<TEntity, bool>> prediacate)
    {
        try
        {
            return await Entities.FirstAsync(prediacate);
        }
        catch (Exception ex)
        {
            return null;
        }
    }

    public virtual async Task DeleteAsync(Expression<Func<TEntity,
bool>> predicate)
    {
        try
        {
            var entity = await Entities.FirstAsync(predicate);
            Entities.Remove(entity);
            await SaveChangesAsync();
        }
        catch (Exception ex)
        {

        }
    }
```

```
    public virtual async Task<PagedResult<TEntity>>
GetPagedAsync(PagedRequest pagedRequest, Expression<Func<TEntity,
bool>> predicate)
    {
        var query = Entities.Where(predicate);

        var totalResults = await query.CountAsync();

        var results = await query
            .Skip((pagedRequest.Page - 1) * pagedRequest.PageSize)
            .Take(pagedRequest.PageSize)
            .ToListAsync();

        return new PagedResult<TEntity>
        {
            PagedItems = results,
            TotalCount = totalResults,
            Page = pagedRequest.Page,
            PageSize = pagedRequest.PageSize
        };
    }

}
```

Додаток Б

Лістинг класу AuthService

```
internal class AuthService : UserService, IAuthService
{

    private readonly IUserTokenRepository _userTokenRepository;
    private readonly IEmailService _emailService;
    private readonly ISecurePasswordHasher _passwordHasher;
    private readonly IPasswordValidator _passwordValidator;


    public AuthService(IUserRepository userRepository,
        IUserTokenRepository userTokenRepository,
        IMapper mapper,
        ISecurePasswordHasher passwordHasher,
        IEmailService emailService,
        IPasswordValidator           passwordValidator)           :
base(userRepository, mapper)
    {
        _userTokenRepository = userTokenRepository;
        _passwordHasher = passwordHasher;
        _emailService = emailService;
        _passwordValidator = passwordValidator;
    }

    public                   async                   Task<UserDTO?>
ChangePasswordAsync(ChangePasswordModel changePasswordModel)
    {
        var user = await _userRepository.FindFirstAsync(x => x.Email
== changePasswordModel.Email);
        if           (user           is           null           ||
!_passwordHasher.Verify(changePasswordModel.OldPassword,
user.PasswordHash))
            return null;

        user.PasswordHash                                          =
_passwordHasher.Hash(changePasswordModel.NewPassword);

        user = await _userRepository.UpdateAsync(user);

        return _mapper.Map<UserDTO>(user);

    }

    public async Task<bool> ConfirmEmailAsync(string token)
    {
        var              usedToken              =              await
_userTokenRepository.UseTokenAsync(token,
UserTokenType.EMAILCONFIRMATION);

        if (usedToken is null)
            return false;
```

```
        usedToken.User.EmailConfirmed = true;

        var user = await _userRepository.UpdateAsync(usedToken.User);
        return user is not null;

    }

    public async Task<bool> IsEmailExistAsync(string email)
    {
        var user = await _userRepository.FindFirstAsync(x => x.Email == email);

        return user is not null;
    }

    public async Task<UserDTO?> LoginAsync(Credentials credentials)
    {

        var user = await _userRepository.FindFirstAsync(x => x.Email == credentials.Email);

        if (user is null || !_passwordHasher.Verify(credentials.Password, user.PasswordHash)) return null;

        return _mapper.Map<UserDTO>(user);
    }

    public async Task<UserDTO?> RegisterAsync(RegisterModel register)
    {
        if (await IsEmailExistAsync(register.Email))
            return null;

        var user = new User()
        {
            Email = register.Email,
            EmailUppercase = register.Email.ToUpper(),
            EmailConfirmed = false,

            FirstName = register.FirstName,
            LastName = register.LastName,

            PhoneNumber = register.PhoneNumber,
            PhoneNumberConfirmed = false,
            Role = «USER»,

            PasswordHash = _passwordHasher.Hash(register.Password),
        };

        user = await _userRepository.AddAsync(user);
```

```
        var token = await
GenerateTokenAsync(UserTokenType.EMAILCONFIRMATION, user);
        if(token is null)
            return null;
        _emailService.SendConfirmationEmail(user.Email,
token.Token);

        return _mapper.Map<UserDTO?>(user);

    }

    public async Task<UserDTO?>
ResetPasswordAsync(ResetPasswordModel resetPasswordModel)
    {
        var token = await
_userTokenRepository.UseTokenAsync(resetPasswordModel.Token,
UserTokenType.RESETPASSWORD);
        if (token is null)
            return null;

        token.User.PasswordHash =
_passwordHasher.Hash(resetPasswordModel.NewPassword);

        var user = await _userRepository.UpdateAsync(token.User);
        return _mapper.Map<UserDTO?>(user);
    }

    public async Task<UserTokenDTO?> CreateResetTokenAsync(string
email)
    {
        var user = await _userRepository.FindFirstAsync(x => x.Email
== email);
        if (user is null)
            return null;

        var token = await
GenerateTokenAsync(UserTokenType.RESETPASSWORD, user);

        _emailService.SendResetPasswordEmail(email, token.Token);


        return _mapper.Map<UserTokenDTO>(token);
    }

    private async Task<UserToken> GenerateTokenAsync(UserTokenType
type, User user)
    {
        var token = new UserToken()
        {
            Token = Guid.NewGuid().ToString(),
            TokenExpiry = DateTime.UtcNow.AddDays(1),
            TokenType = type,
```

```
            User = user,
            UserId = user.Id
        };

        token = await _userTokenRepository.AddAsync(token);

        return token;
    }

    public async Task<UserDTO?> UpdateRefreshTokenAsync(string
email, string token, DateTime time)
    {
        var user = await _userRepository.FindFirstAsync(x => x.Email
== email);
        if (user is null) return null;

        user.RefreshToken = token;
        user.RefreshTokenExpiryTime = time;
        user = await _userRepository.UpdateAsync(user);
        return _mapper.Map<UserDTO?>(user);
    }


    public async Task<TfaData?> GetTfaDataAsync(string email)
    {
        var user = await _userRepository.FindFirstAsync(x =>
x.Email == email);
        if (user is null) return null;

        return new TfaData()
        {
            IsTwoFactorEnabled = user.IsTfaEnabled,
            AuthenticatorKey = user.TfaAuthenticatorKey
        };
    }

    public async Task<string?> GetTfaKeyAsync(string email)
    {
        User? user = await _userRepository.FindFirstAsync(x =>
x.Email == email);
        if (user is null) return null;

        return user.TfaAuthenticatorKey;
    }

    public async Task<string?> ResetTfaKeyAsync(string email,string
key)
    {
        var user = await _userRepository.FindFirstAsync(x => x.Email
== email);
        if (user is null) return null;

        user.TfaAuthenticatorKey = key;
```

```
        user = await _userRepository.UpdateAsync(user);
        return user.TfaAuthenticatorKey;
    }

    public async Task SetTfaEnabledAsync(string email, bool enabled)
    {
        var user = await _userRepository.FindFirstAsync(x => x.Email
== email);
        if (user is null) return ;

        user.IsTfaEnabled = enabled;


        await _userRepository.UpdateAsync(user);
    }




}
```

Додаток В

Лістинг класу AuthController

```
[Route(«api/[controller]»)]
[ApiController]
public class AuthController : ControllerBase
{
    private readonly IAuthService _authService;
    private readonly JwtTokenService _jwtTokenService;
    private readonly UrlEncoder _urlEncoder;
    private readonly ITfaService _tfaService;

    public AuthController(
        IAuthService authService,
        JwtTokenService jwtTokenService,
        UrlEncoder urlEncoder,
        ITfaService tfaService)
    {
        _authService = authService;
        _jwtTokenService = jwtTokenService;
        _urlEncoder = urlEncoder;
        _tfaService = tfaService;
    }

    [HttpPost(«login»)]
    public async Task<IActionResult> LoginAsync([FromBody]
Credentials credentials)
    {
        var user = await _authService.LoginAsync(credentials);

        if (user is not null)
        {
            if(!user.IsTfaEnabled)
            {
                var claims = new[]
                {
                    new Claim(ClaimTypes.Email, user.Email),
                    new Claim(ClaimTypes.NameIdentifier,
user.Id.ToString()),
                    new Claim(ClaimTypes.Role, user.Role),
                    new Claim(ClaimTypes.Name,$»{ user.FirstName}
{ user.LastName}»)
                };

                var accessToken =
_jwtTokenService.GenerateAccessToken(claims);
                var refreshToken =
_jwtTokenService.GenerateRefreshToken();

                return Ok(new AuthenticatedUser
                {
                    AccessToken = accessToken,
                    RefreshToken = refreshToken,
```

```
                    User = user
                });
            }
            else
            {
                return Ok(new AuthenticatedUser
                {
                    TfaToken =
_tfaService.GenerateTfaToken(user.Id.ToString()),
                    IsTfaEnabled = true
                });
            }
        }
        return NotFound(«user with email or password not found»);
    }

    [HttpPatch(«refresh-token»)]
    public async Task<IActionResult> RefreshAsync([FromBody]
RefreshRequest refreshRequest)
    {
        var principal =
_jwtTokenService.GetPrincipalFromExpiredToken(refreshRequest.Acces
sToken);
        var email = principal.FindFirst(ClaimTypes.Email)?.Value;

        if (email is null)
            return BadRequest();

        var user = await _authService.GetAsync(email);

        if (user is null || user.RefreshToken !=
refreshRequest.RefreshToken)
            return BadRequest();

        var claims = new[]
        {
            new Claim(ClaimTypes.Email, user.Email),
            new Claim(ClaimTypes.NameIdentifier,
user.Id.ToString()),
            new Claim(ClaimTypes.Role, user.Role),
            new Claim(ClaimTypes.Name,$»{ user.FirstName} {
user.LastName}»)
        };

        var accessToken =
_jwtTokenService.GenerateAccessToken(claims);
        var refreshToken =
_jwtTokenService.GenerateRefreshToken();

        await _authService.UpdateRefreshTokenAsync(email,
refreshToken,
```

```
DateTime.Now.AddDays(_jwtTokenService.RefreshTokenExpirationDays))
;

        return Ok(new AuthenticatedUser
        {
            AccessToken = accessToken,
            RefreshToken = refreshToken,
            User = user
        });
    }

    [HttpPost(«register»)]
    public async Task<IActionResult> RegisterAsync([FromBody]
RegisterModel register)
    {
        var user = await _authService.RegisterAsync(register);

        if (user is not null)
            return Ok();
        return BadRequest();
    }

    [HttpPatch(«change-password»)]
    public async Task<IActionResult>
ChangePasswordAsync([FromBody] ChangePasswordModel
changePasswordModel)
    {
        var user = await
_authService.ChangePasswordAsync(changePasswordModel);

        if (user is not null)
            return Ok();
        return BadRequest();
    }

    [HttpPatch(«reset-password»)]
    public async Task<IActionResult> ResetPasswordAsync([FromBody]
ResetPasswordModel resetPasswordModel)
    {
        var user = await
_authService.ResetPasswordAsync(resetPasswordModel);

        if (user is not null)
            return Ok();
        return BadRequest();
    }

    [HttpPost(«create-reset-token/{email}»)]
    public async Task<IActionResult>
CreateResetTokenAsync([FromRoute] string email)
    {
```

```
        var token = await
_authService.CreateResetTokenAsync(email);

        if (token is not null)
            return Ok();
        return BadRequest();
    }

    [HttpPatch(«confirm-email/{token}»)]
    public async Task<IActionResult> ConfirmEmailAsync([FromRoute]
string token)
    {
        var result = await _authService.ConfirmEmailAsync(token);

        if (result)
            return Ok();
        return BadRequest();
    }

    [HttpHead(«is-email-exist/{email}»)]
    public async Task<IActionResult> IsEmailExistAsync([FromRoute]
string email)
    {

        if (await _authService.IsEmailExistAsync(email))
            return Ok();
        return NotFound();
    }


    [Authorize]
    [HttpDelete(«revoke-token»)]
    public async Task<IActionResult> RevokeTokenAsync()
    {
        var email = User.FindFirst(ClaimTypes.Email)?.Value;
        var result = await
_authService.UpdateRefreshTokenAsync(email, ««,
DateTime.UtcNow.AddDays(-1));
        if (result is not null)
            return Ok();
        return BadRequest();
    }


    [Authorize]
    [HttpGet(«tfa-setup»)]
    public async Task<IActionResult> GetTfaSetupAsync()
    {
        var email = User.FindFirst(ClaimTypes.Email)?.Value;
        if(email is null)
            return Unauthorized();

        var tfaData = await _authService.GetTfaDataAsync(email);
```

```
        if (tfaData is null)
            return NotFound();

        if (string.IsNullOrEmpty(tfaData.AuthenticatorKey))
        {
            tfaData.AuthenticatorKey = await
_authService.ResetTfaKeyAsync(email,
_tfaService.GenerateSecret());
        }


        return Ok(new TfaSetupDto
        {
            Email = email,
            SetupURL = _tfaService.GenerateSetupURL(«Maj»,email,
tfaData.AuthenticatorKey),
        });
    }

    [HttpPost(«tfa-setup»)]
    [AllowAnonymous]
    public async Task<IActionResult> EnableTfaAsync([FromBody]
TfaRequest tfaReq)
    {
        if (string.IsNullOrWhiteSpace(tfaReq.Code) ||
string.IsNullOrWhiteSpace(tfaReq.Email))
            return BadRequest();

        var tfaData = await
_authService.GetTfaDataAsync(tfaReq.Email);

        if (tfaData is null ||
string.IsNullOrWhiteSpace(tfaData.AuthenticatorKey))
            return NotFound();

        var isValid =
_tfaService.VerifyCode(tfaData.AuthenticatorKey, tfaReq.Code);

        if (isValid){
            await _authService.SetTfaEnabledAsync(tfaReq.Email,
true);
            return Ok();
        }

        return BadRequest();
    }

    [Authorize]
    [HttpDelete(«tfa-disable»)]
    public async Task<IActionResult> DisableTfaAsync()
    {
        var email = User.FindFirst(ClaimTypes.Email)?.Value;
```

```
        if(email is null)
            return Unauthorized();

        await _authService.SetTfaEnabledAsync(email, false);

        return Ok();
    }

[AllowAnonymous]
[HttpPost(«tfa-login»)]
public async Task<IActionResult> TfaLoginsync([FromBody]
TfaRequest loginModel)
{

    var user = await _authService.GetAsync(loginModel.Email);
    if (user is null)
        return NotFound();


    var tfaData = await
_authService.GetTfaDataAsync(loginModel.Email);

    if (!tfaData.IsTwoFactorEnabled)
        return BadRequest(«TFA is not enabled»);


    try
    {
        var userIdFromTfaToken =
_tfaService.ValidateTfaToken(loginModel.TfaToken);
        if (userIdFromTfaToken != user.Id)
        {
            return BadRequest(«Invalid TFA token»);
        }
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }

    if (!_tfaService.VerifyCode(tfaData.AuthenticatorKey,
loginModel.Code))
        return BadRequest(«Invalid code»);

    var claims = new[]
    {
        new Claim(ClaimTypes.Email, user.Email),
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
        new Claim(ClaimTypes.Role, user.Role),
        new Claim(ClaimTypes.Name, $»{user.FirstName}
{user.LastName}»)
    };
```

```
    var accessToken =
_jwtTokenService.GenerateAccessToken(claims);
    var refreshToken = _jwtTokenService.GenerateRefreshToken();

    return Ok(new AuthenticatedUser
    {
        AccessToken = accessToken,
        RefreshToken = refreshToken,
        User = user,
        IsTfaEnabled = true
    });
}

}
```

Додаток Д

Лістинг класу SecurityService

```
@Injectable({providedIn: 'root'})
export class SecurityService {
    private readonly USER_KEY = 'security-user';

    constructor() {
    }

    private _isAuthenticated$:BehaviorSubject<boolean> = new
BehaviorSubject<boolean>(false);
    public isAuthenticated$ =
this._isAuthenticated$.asObservable();


    public login(securityUser: SecurityUser): void {
        this.updateUserInLocalStorage(securityUser);
        this._isAuthenticated$.next(true);
    }

    public logout(): void {
        this.removeUserFromLocalStorage();
        this._isAuthenticated$.next(false);
    }

    private updateUserInLocalStorage(securityUser:SecurityUser):
void {
        localStorage.setItem(this.USER_KEY,
JSON.stringify(securityUser));
    }

    private removeUserFromLocalStorage(): void {
        localStorage.removeItem(this.USER_KEY);

    }

    public isAuthenticated(): boolean {
      return localStorage.getItem(this.USER_KEY) !== null;
    }

    public getSecurityUser(): SecurityUser | null {
        const stringUser = localStorage.getItem(this.USER_KEY);
        return  stringUser ? JSON.parse(stringUser) : null;
    }

    public getAccessToken(): string | null {
        const stringUser = localStorage.getItem(this.USER_KEY);
        return  stringUser ? JSON.parse(stringUser).accessToken :
null;
    }

    public hasRole(role: string): boolean {
```

```
        var securityUser = this.getSecurityUser();
        if (securityUser && securityUser.user) {
            return securityUser.user.role === role;
        }
        return false;
    }

    public hasAnyRole(roles: string[]): boolean {
        var securityUser = this.getSecurityUser();
        if (securityUser && securityUser.user) {
            return roles.includes(securityUser.user.role);
        }
        return false;
    }



}
```