МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавра

на тему: «**Freelance Marketplace Platform»**

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Лавренюк Андрій Вікторович

Керівник: *старший викладач кафедри економіко-математичного моделювання та інформаційних технологій,*
Клебан Юрій Вікторович

Рецензент: *кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса*
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ
Завідувач кафедри інформаційних технологій та аналітики даних
_____ (проф., д.е.н. Кривицька О.Р.)
Протокол № 11 від 20 травня 2026 р.

Острог, 2026

Завдання на кваліфікаційну роботу

Мета кваліфікаційної роботи полягає в розробці повнофункціональної веб-платформи фріланс-маркетплейсу, яка забезпечує ефективну взаємодію між роботодавцями та фрілансерами з сучасним рівнем користувацького досвіду, безпеки та продуктивності.

Основні завдання роботи включають: проведення аналізу існуючих фріланс-платформ та визначення їхніх переваг і недоліків; розробку функціональних та нефункціональних вимог до системи; проектування реляційної бази даних у PostgreSQL; реалізацію backend-частини на ASP.NET Core з використанням Repository Pattern, CQRS-архітектури та бібліотеки MediatR; інтеграцію платіжної системи Stripe для обробки платежів та escrow; впровадження сповіщень у реальному часі за допомогою SignalR WebSocket; розробку frontend-частини на React (Vite + TypeScript) з використанням Redux Toolkit (RTK Query); реалізацію ключових модулів системи, таких як реєстрація та автентифікація користувачів, створення та публікація проектів (фіксованих і погодинних), систему пропозицій (Bids/Quotes), укладання контрактів, управління етапами (milestones), портфоліо фрілансерів, відгуки та рейтинги, а також модуль вирішення спорів; проведення інтеграційного тестування основних функціональних блоків; розробку сучасного адаптивного інтерфейсу користувача; забезпечення безпеки та масштабованості рішення.

У роботі передбачається використання технологій: ASP.NET Core 8, Entity Framework Core, PostgreSQL, MediatR, SignalR, Stripe.NET на backend та React 18 з TypeScript, Vite, Redux Toolkit на frontend. Особлива увага приділяється Clean Architecture, інтеграційному тестуванню та якості коду.

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: *Розробка веб-платформи фріланс-маркетплейсу з використанням сучасних технологій backend та frontend (Freelance Marketplace Platform)*

Автор: *Лавренюк Андрій Вікторович*

Науковий керівник: *Клебан Юрій Вікторович, старший викладач кафедри економіко-математичного моделювання та інформаційних технологій.*

Захищена «……»…… 2026 року.

Пояснювальна записка до кваліфікаційної роботи: *84 с., 19 рис., 1 табл., 3 додатків, 39 джерел.*

Ключові слова: *фріланс-платформа, маркетплейс, CQRS, MediatR, SignalR, WebSocket, Stripe, PostgreSQL, Repository Pattern, React, Redux Toolkit, TypeScript, Vite, інтеграційне тестування.*

Короткий зміст праці:

У роботі розроблено повноцінну веб-платформу фріланс-маркетплейсу, яка забезпечує взаємодію між роботодавцями та фрілансерами. Реалізовано основні функціональні модулі: реєстрація та автентифікація користувачів, створення і публікацію проектів, систему пропозицій (Bids/Quotes), укладання контрактів, поетапну оплату через milestones, систему відгуків та рейтингів, портфоліо фрілансерів, а також модуль вирішення спорів. Backend-частина розроблена на технологічному стеку ASP.NET Core з використанням Repository Pattern, CQRS (MediatR), PostgreSQL як основної бази даних. Реалізовано інтеграцію з платіжною системою Stripe, сповіщення в реальному часі за допомогою SignalR WebSocket, а також комплекс інтеграційних тестів. Frontend-частина виконана на React (Vite + TypeScript) з використанням Redux Toolkit (RTK) для управління станом додатку. Забезпечено сучасний адаптивний інтерфейс, зручну навігацію та високу швидкодію. Розроблена платформа підтримує два типи проектів (фіксовані та погодинні), систему escrow-платежів, управління ролями, а також забезпечує високий рівень безпеки та масштабованість. У роботі проведено аналіз аналогічних

рішень, обґрунтовано вибір технологічного стеку, розроблено базу даних, спроєктовано API, реалізовані основні модулі та проведено тестування системи. Ключові результати: створено робочий прототип фріланс-платформи, готової до подальшого розвитку та комерційного використання.

This work presents the development of a full-fledged freelance marketplace web platform that facilitates interaction between employers and freelancers. The main functional modules have been implemented, including user registration and authentication, project creation and publication, a bidding system (Bids/Quotes), contract formation, milestone-based payments, a review and rating system, freelancer portfolios, and a dispute resolution module. The backend is developed using the ASP.NET Core technology stack, utilizing the Repository Pattern, CQRS (MediatR), and PostgreSQL as the primary database. Furthermore, integration with the Stripe payment system, real-time notifications via SignalR WebSockets, and a comprehensive suite of integration tests have been successfully implemented. The frontend part is built with React (Vite and TypeScript) using Redux Toolkit (RTK) for application state management, providing a modern, responsive interface, intuitive navigation, and high performance. The developed platform supports two types of projects (fixed-price and hourly), an escrow payment system, and role management, while ensuring a high level of security and scalability. The study includes an analysis of analogous solutions, justification of the chosen technology stack, database development, API design, implementation of the core modules, and system testing. Key results include the creation of a working prototype of the freelance platform, which is ready for further development and commercial use.

ЗМІСТ

ВСТУП

В умовах стрімкого розвитку цифрової економіки та глобалізації ринку праці сфера фрілансу набуває безпрецедентної популярності, перетворюючись на основний формат роботи для мільйонів спеціалістів у всьому світі. Сучасний ринок вимагає створення надійних, швидких та зручних інструментів для взаємодії між замовниками та виконавцями. Особливої актуальності ця тема набуває на тлі стрімкого впровадження технологій штучного інтелекту, який кардинально змінює ландшафт віддаленої роботи. Алгоритми машинного навчання та генеративні моделі не лише автоматизують рутинні процеси виконавців, але й породжують абсолютно нові категорії послуг. Це вимагає від сучасних платформ надзвичайної гнучкості, здатності адаптуватися до нових форматів співпраці, а в перспективі – готовності до інтеграції інтелектуальних систем для автоматизованого метчингу фрілансерів із релевантними проектами. Саме тому розробка сучасної платформи, яка здатна витримувати високі навантаження, забезпечувати миттєву комунікацію та гарантувати безпеку фінансових операцій, є вкрай затребуваним інженерним завданням.

Метою даного дослідження є безпосереднє створення та розробка програмного продукту, а саме повноцінної веб-орієнтованої інформаційної системи у вигляді платформи для фрілансу, яка буде об'єднувати передові підходи до побудови мікросервісної або модульної монолітної архітектури та забезпечуватиме високий рівень надійності.

Для досягнення поставленої мети було визначено низку ключових завдань дослідження, які необхідно вирішити в процесі роботи. Насамперед потрібно провести комплексний аналіз предметної області, дослідивши специфіку взаємодії користувачів на ринку віддаленої праці та бізнес-логіку існуючих рішень. Наступним кроком є ретельне проектування інформаційної системи, що включає моделювання реляційної бази даних, розробку архітектури серверної частини з розділенням відповідальності та планування структури клієнтського додатку. Заключним, але найважливішим завданням є обґрунтований вибір інструментарію,

методів реалізації та тестування програмного продукту. Це передбачає визначення оптимального технологічного стеку для бекенду та фронтенду, налаштування платіжних шлюзів, реалізацію механізмів роботи в реальному часі та покриття критичного функціоналу інтеграційними тестами.

Об’єктом даного дослідження є безпосередньо сама інформаційна система, тобто програмний комплекс платформи для фрілансу, що проектується та розробляється.

Предметом дослідження виступають інструментальні засоби, інформаційні технології, архітектурні патерни та методи програмної інженерії, що застосовуються для реалізації об'єкта роботи. До них належать платформи розробки, такі як .NET та екосистема React, системи управління базами даних PostgreSQL, патерни проектування Repository та CQRS, протоколи двостороннього зв'язку через WebSockets, а також методи організації безпечних фінансових транзакцій через зовнішні API.

РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Аналіз предметного середовища та класифікація ринку фріланс-послуг

1.1.1. Сучасний стан та тенденції розвитку глобального ринку віддаленої роботи

На сучасному етапі розвитку світової економіки ми спостерігаємо фундаментальний зсув парадигми трудових відносин, спричинений стрімким переходом до цифрової економіки. Глобальний ринок віддаленої роботи, який раніше вважався нішевим сегментом, сьогодні трансформувався в один із найпотужніших драйверів економічного зростання. Ця трансформація була суттєво прискорена глобальними подіями останніх років, які змусили бізнес адаптуватися до децентралізованих моделей управління персоналом. У результаті компанії усвідомили економічну ефективність залучення незалежних підрядників: це дозволяє оптимізувати операційні витрати, зменшити податкове навантаження на утримання штату та отримати миттєвий доступ до глобального пулу талантів без географічних обмежень. Відповідно, попит на спеціалістів категорії фріланс зростає експоненціально, формуючи так звану гіг-економіку (gig economy), де короткострокові контракти та проектна робота стають домінуючим форматом зайнятості для мільйонів професіоналів.

Окремим і надзвичайно потужним фактором, що формує сучасні тенденції розвитку цього ринку, є вплив технологічних інновацій, зокрема стрімкий розвиток систем штучного інтелекту. Алгоритми машинного навчання та генеративні моделі кардинально змінюють ландшафт професій. З одного боку, штучний інтелект автоматизує рутинні завдання, що вимагає від спеціалістів постійного підвищення своєї кваліфікації та переходу до вирішення більш складних, креативних або архітектурних завдань. З іншого боку, впровадження цих технологій породжує абсолютно нові категорії послуг на ринку фрілансу, такі як AI-консалтинг,

промпт-інжиніринг, розробка та інтеграція інтелектуальних моделей у бізнес-процеси замовників. У таких динамічних умовах класичні підходи до пошуку виконавців та управління проектами стають неефективними. Це обґрунтовує необхідність розробки моєї нової інформаційної системи, яка буде здатна гнучко адаптуватися до мінливих вимог ринку, надавати сучасні інструменти для швидкої комунікації та гарантувати безпеку в умовах високої швидкості укладання цифрових угод.

1.1.2. Класифікація фріланс-послуг та моделей співпраці

Для якісного проектування архітектури бази даних та бізнес-логіки моєї платформи критично важливо розуміти багатогранну класифікацію послуг, що надаються на ринку, та існуючих моделей фінансової взаємодії. За галузевою ознакою ринок фрілансу традиційно поділяється на кілька великих сегментів. Домінуючим залишається сектор інформаційних технологій (IT), який охоплює веб-розробку, створення мобільних додатків, системне адміністрування, тестування програмного забезпечення та управління базами даних. Наступним за обсягом є креативний сектор, що включає графічний дизайн, UI/UX проектування інтерфейсів, відеомонтаж, 3D-моделювання та створення анімацій. Не менш вагомим є сегмент цифрового маркетингу, SEO-оптимізації, копірайтингу та перекладів. Враховуючи цю різноманітність, у моєму програмному продукті передбачено гнучку систему категорій проектів та динамічне формування портфоліо і списку навичок (Skills) спеціаліста, що дозволяє максимально точно ідентифікувати експертизу фрілансера незалежно від його галузі.

Класифікація за форматами фінансової співпраці є ще більш визначальною для архітектури маркетплейсу. Першою базовою моделлю є робота з фіксованою оплатою за весь проект (Fixed Price). Цей підхід ідеально підходить для завдань із чітко визначеним технічним завданням, невеликим обсягом робіт та зрозумілими критеріями приймання. Другою моделлю є погодинна оплата (Hourly Rate), яка застосовується в довгострокових проектах, де обсяг робіт важко оцінити заздалегідь, або під час надання послуг технічної підтримки та консалтингу. Третьою, найбільш

складною та найефективнішою для великих замовлень моделлю, є поетапне виконання робіт з використанням проміжних контрольних точок — майлстоунів (Milestones). Саме ця модель закладена в основу управління контрактами у моєму проекті. Використання майлстоунів дозволяє розбити масштабний проект на логічні частини. Роботодавець має змогу резервувати кошти та приймати роботу покроково, а фрілансер отримує фінансову гарантію та регулярні виплати за фактично виконані частини технічного завдання, що суттєво знижує ризики для обох сторін угоди.

1.1.3. Специфіка взаємодії суб'єктів на ринку фрілансу

Аналіз предметної області вимагає глибокого розуміння психології та практичних потреб головних дійових осіб платформи — роботодавців (Employer) та спеціалістів категорії фріланс (Freelancer). Взаємодія між цими суб'єктами часто ускладнюється відсутністю особистого контакту, географічною віддаленістю та різницею в менталітетах чи часових поясах. Ключовим "болем" для роботодавця є ризик отримання неякісного результату, зрив дедлайнів та загроза витоку конфіденційної інформації чи втрати прав інтелектуальної власності. Роботодавець потребує інструментів для прозорого контролю за ходом виконання робіт, можливості ознайомитися з достовірним рейтингом та реальними відгуками про виконавця до моменту укладання контракту. Крім того, замовнику критично важливий безпечний фінансовий механізм, який гарантуватиме, що його кошти будуть перераховані виконавцю виключно після повного затвердження виконаної роботи.

Зі свого боку, фрілансер стикається зі специфічним набором викликів, головним з яких є фінансова незахищеність та ризик неоплати виконаної роботи недобросовісним замовником. Для спеціаліста надзвичайно важливою є гарантія резервування коштів на платформі (депозит) ще до початку виконання завдання. Також фрілансери часто страждають від так званого "розповзання меж проекту" (scope creep), коли замовник вимагає виконання додаткових робіт, не передбачених початковим контрактом, без відповідної доплати. Для вирішення цих проблем моя платформа повинна забезпечувати абсолютно чітку фіксацію умов у системі

котирувань (Quote) перед створенням контракту. Спільною потребою для обох сторін є забезпечення швидкої, безперебійної та задокументованої комунікації. Саме тому обмін повідомленнями в реальному часі є фундаментальною вимогою до моєї системи. Крім того, враховуючи людський фактор, специфіка взаємодії вимагає наявності незалежного інституту арбітражу. У випадку виникнення непереборних розбіжностей щодо якості роботи або термінів, моя система передбачає механізм відкриття суперечок (Dispute), де модератор платформи виступає незалежним суддею, маючи доступ до історії повідомлень та контрактних зобов'язань для винесення справедливого рішення (Dispute Resolution).

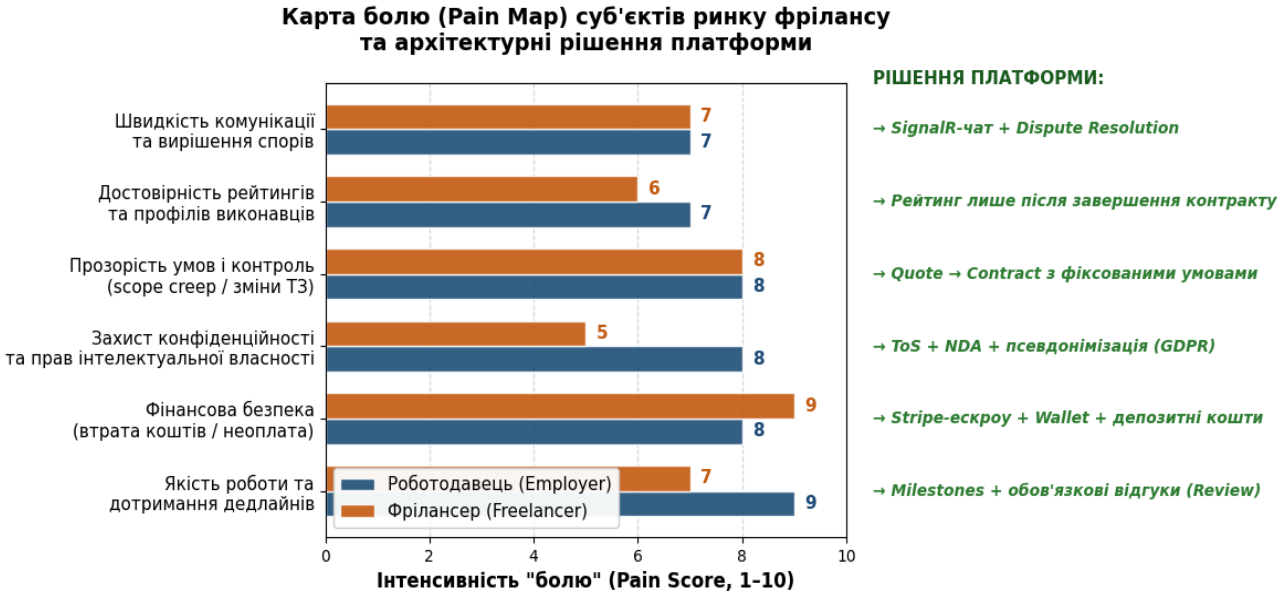


Рис. 1.1. Карта болю (Pain Map) суб'єктів ринку фрілансу та архітектурні рішення платформи

Джерело: Розроблено автором на основі аналізу предметної області (розд. 1.1.3)

1.2. Нормативно-правове регулювання та міжнародні стандарти діяльності

1.2.1. Правові основи електронної комерції та цифрових контрактів

Функціонування будь-якої сучасної інформаційної системи, що забезпечує взаємодію користувачів на глобальному ринку праці та передбачає фінансовий

обмін, вимагає суворого дотримання низки національних та міжнародних нормативно-правових актів. Правовою основою для роботи розроблюваної мною платформи є законодавство у сфері електронної комерції та цивільного права, яке прирівнює електронні угоди до їх класичних паперових аналогів. Оскільки спеціаліст категорії фріланс виступає як незалежний підрядник, взаємовідносини між ним, замовником та моїм маркетплейсом оформлюються у вигляді публічної оферти та загальної користувацької угоди (Terms of Service). Ці юридичні документи приймаються кожним користувачем під час реєстрації в системі і регламентують механізми передачі прав інтелектуальної власності на створений цифровий продукт, а також правила вирішення можливих суперечок через інститут модерації.

З технічної точки зору, процес укладання угоди на моїй платформі повністю оцифрований. Коли роботодавець переглядає надіслане фрілансером котирування (Quote) і натискає кнопку його затвердження в клієнтському інтерфейсі на React, система автоматично генерує об'єкт контракту (Contract). Ця дія є електронним підтвердженням згоди (акцептом) з усіма умовами, включаючи етапи виконання (Milestones) та зафіксовану вартість робіт (agreed rate). Електронний цифровий слід цієї операції, що включає унікальні ідентифікатори користувачів, мітки часу та незмінні параметри завдання, надійно фіксується у моїй базі даних PostgreSQL. У разі виникнення конфліктної ситуації, ці цифрові записи стають беззаперечною доказовою базою для модератора платформи під час відкриття суперечки (Dispute) та винесення остаточного рішення (Dispute Resolution). Таким чином, моя система не просто виступає дошкою оголошень, а забезпечує повноцінний легітимний простір для укладання та супроводу цифрових контрактів, знімаючи з користувачів тягар паперової бюрократії.

1.2.2. Стандарти захисту персональних даних

Враховуючи глобальний характер діяльності мого маркетплейсу та потенційне залучення клієнтів з країн Європейського Союзу та інших юрисдикцій із суворим законодавством, архітектура моєї системи проектувалася з урахуванням повної сумісності з вимогами Загального регламенту про захист даних (GDPR). Цей

міжнародний стандарт встановлює жорсткі правила щодо збору, обробки, зберігання та видалення персональної інформації. Згідно з принципом мінімізації даних, моя платформа збирає лише ту інформацію, яка є критично необхідною для коректного функціонування бізнес-логіки: електронну пошту, країну резиденції, інформацію про професійні навички та елементи портфоліо. Я реалізував надійні механізми захисту цієї інформації: паролі користувачів ніколи не зберігаються у відкритому вигляді, а криптографічно хешуються перед записом у базу даних. Авторизація та ідентифікація сесій відбувається виключно через захищені JWT-токени з обмеженим терміном дії або через делеговану авторизацію Google OAuth 2.0.

Крім того, стандарт GDPR вимагає надання користувачам безперешкодної можливості реалізувати право на доступ до своїх даних та "право на забуття" (повне видалення профілю). У розробленій мною архітектурі передбачені спеціальні механізми обробки таких запитів. Однак, оскільки моя платформа також оперує фінансовими контрактами та податково-значущими транзакціями, просте фізичне видалення записів з таблиці User є неможливим через ризик порушення цілісності реляційної бази даних та фінансової звітності. Тому я імплементував алгоритми псевдонімізації та "м'якого видалення" (Soft Delete). При ініціюванні видалення акаунту персональна публічна інформація з профілів Freelancer чи Employer фізично стирається або замінюється на анонімні ідентифікатори, тоді як історичні записи в таблицях фінансових транзакцій зберігаються в знеособленому вигляді для дотримання вимог фінансового моніторингу. Такий збалансований підхід дозволяє моїй системі відповідати найвищим стандартам приватності без шкоди для внутрішньої логіки маркетплейсу.

1.2.3. Міжнародні вимоги до безпеки фінансових транзакцій

Окремим і чи не найважливішим аспектом нормативно-правового регулювання є безпека фінансових взаємозв'язків. Оскільки моя платформа виконує роль фінансового гаранта, утримуючи депозитні кошти роботодавця до моменту успішного виконання етапів контракту (Milestones), система підпадає під дію суворих міжнародних регуляцій. Найголовнішим стандартом у цій сфері є Payment

Card Industry Data Security Standard (PCI DSS). Цей стандарт вимагає проходження надзвичайно складних і дорогих процедур аудиту безпеки для будь-якої системи, яка фізично зберігає, обробляє або передає повні реквізити банківських карток (номери карток, терміни дії, CVV-коди). Щоб уникнути цих юридичних та інфраструктурних ризиків і гарантувати користувачам абсолютну фінансову безпеку, я прийняв стратегічне архітектурне рішення делегувати обробку платежів сертифікованому зовнішньому платіжному шлюзу Stripe.

Використання Stripe дозволяє моєму проекту повністю задовольнити вимоги PCI DSS без необхідності створення власного ізольованого сховища карток. Під час ініціалізації платежу для депозиту за контрактом, клієнтський додаток на React напряму взаємодіє із захищеними серверами Stripe. Секретні платіжні дані токенізуються виключно на стороні браузера клієнта і ніколи не проходять через мій бекенд на ASP.NET Core. Мій сервер оперує лише знеособленими криптографічними ідентифікаторами платіжних сесій (Session IDs) та безпечними вебхуками, які підтверджують факт успішного зарахування коштів у потрібній валюті. У моїй базі даних PostgreSQL, в таблицях ContractPayment, Payment та WalletTransaction, зберігається лише метадані: суми, дати, ідентифікатори користувачів та посилання на транзакції у системі Stripe. Такий архітектурний розподіл відповідальності повністю захищає платформу від витоків чутливої фінансової інформації у випадку гіпотетичних кібератак та формує високий рівень довіри до маркетплейсу з боку як замовників, так і спеціалістів категорії фріланс.

1.3. Функціональне моделювання системи та опис бізнес-процесів

1.3.1. Визначення акторів та їх ролей у системі

Для забезпечення безпеки, розмежування доступу та правильної маршрутизації бізнес-логіки, моя інформаційна система базується на моделі управління доступом на основі ролей (Role-Based Access Control). Всього в проекті 4 ролі - адмін, модератор і власне роботодавець з фрілансером. Кожна з цих ролей

наділена чітко регламентованим набором повноважень, які визначають її можливості в системі.

Адміністратор (admin) має повний доступ до системи. Ця роль передбачає найвищий рівень привілеїв, що дозволяє здійснювати глобальне управління платформою, переглядати статистику, керувати категоріями та навичками, а також модерувати облікові записи інших користувачів у разі порушення ними правил платформи. Роль адміністратора є критичною для підтримки загальної інфраструктури та операційної діяльності маркетплейсу.

Модератор (moderator) займається розглядом та вирішенням спорів. Цей актор виступає в ролі незалежного арбітра, коли між замовником та виконавцем виникають конфліктні ситуації. Модератор не має доступу до глобальних налаштувань системи, але наділений повноваженнями вивчати історію повідомлень, аналізувати докази сторін та ухвалювати фінансово значущі рішення щодо переказу чи повернення зарезервованих коштів.

Роботодавець (employer) відповідає за управління проектами, контрактами та прийняття квот. Цей актор формує попит на платформі. Роботодавець має повноваження публікувати нові технічні завдання, управляти фінансовим балансом свого гаманця, ініціювати пряму комунікацію з кандидатами та, найголовніше, управляти життєвим циклом контракту, поетапно затверджуючи виконану роботу.

Фрілансер (freelancer) має можливість подачі заявок, виконання контрактів та ведення портфоліо. Цей актор формує пропозицію послуг. Повноваження фрілансера зосереджені навколо презентації власної експертизи, пошуку релевантних замовлень, формування комерційних пропозицій та управління статусами виконання завдань у межах активних контрактів.

1.3.2. Моделювання прецедентів (Use Cases) взаємодії користувачів

Функціональне моделювання моєї платформи передбачає опис ключових прецедентів, які покривають усі етапи взаємодії користувача із системою. Першим і найважливішим прецедентом є реєстрація та автентифікація. Реєстрація і логін можуть здійснюватися за допомогою google OAuth. Під час цього процесу система

збирає базову інформацію: хешований пароль, електронну пошту, країну, аватар зображення та зазначення кількох мов з рівнем володіння. Ключовим моментом прецеденту є те, що якщо при реєстрації користувач обирає певну роль, то на цій основі створюється або додаткова табличка Freelancer або Employer.

Наступним прецедентом є управління профілем спеціаліста. Якщо користувач обрав роль фрілансер, то він може заповнити свій профіль різними елементами Portfolio. У портфоліо він може зазначити певні проекти над якими він працював поза межеми платформи. Також для фрілансера є можливість для профілю зазначити його певні навички. Це створює цифрове резюме, необхідне для успішної конкуренції.

Прецедент пошуку проектів та подачі заявок об'єднує дії обох сторін. Спочатку роботодавець створює проект з статусом open, та з певними позначками категорії. Після публікації фрілансери можуть знайти його за допомогою потрібних фільтрів. Знайшовши завдання, ініціюється алгоритм подачі заявки (Bidding), що запускає механізм прямої взаємодії та торгу між акторами.

Для забезпечення безпеки, розмежування доступу та правильної маршрутизації бізнес-логіки система базується на моделі управління доступом на основі ролей (Role-Based Access Control, RBAC), формалізованій Ferraiolo та Kuhn та стандартизованій ANSI/INCITS 359-2004. Матриця прав доступу наведено в табл. 1.1.

Таблиця 1.1

Операція	Адміністратор	Модератор	Роботодавець	Фрілансер
Реєстрація / Авторизація	+	+	+	+
Управління профілем	+	+	+	+
Публікація проектів	+	−	+	−
Подача заявок (Bid)	+	−	−	+

Продовження табл. 1.1

Формування Quote	+	−	−	+
Створення контракту	+	−	+	−
Управління Milestones	+	±	±	+
Фінансові операції (Wallet)	+	−	+	+
Чат / Повідомлення	+	+	+	+
Відгуки (Review)	+	−	+	+
Відкриття суперечки (Dispute)	+	+	+	+
Арбітраж (DisputeResolution)	+	+	−	−
Управління категоріями / навичками	+	−	−	−
Модерація користувачів	+	+	−	−
Перегляд статистики	+	−	−	−
Управління ролями	+	−	−	−

Примітка: «+» — повний доступ; «±» — обмежений доступ; «−» — доступ заборонено.

Джерело: складено автором на основі моделі RBAC та стандарту ANSI/INCITS 359-2004

1.3.3. Опис ключових бізнес-процесів життєвого циклу проекту

Життєвий цикл проекту на моїй платформі — це суворо детермінований алгоритм, що складається з низки послідовних бізнес-процесів. Він розпочинається з публікації роботодавцем технічного завдання. До проекту можна додати milestones, для детальнішого опису роботи і оплати в майбутньому. Фрілансери які знайшли підходящий для себе проект спочатку відправляють Bid. Після цього роботодавець може написати через табличку Message повідомлення. У процесі переговорів формується Quote з більш детальними умовами.

Якщо роботодавець задоволений певним Quote фрілансера, він через нього може створити Contract. Цей процес юридично і технічно фіксує домовленості. Contract створюється на основі проекту, включно з його milestones. Важливою фінансовою деталлю є те, що у контракт в agreed rate вноситься саме сума з quote.

Безпосереднє виконання роботи моделюється через зміну статусів. При роботі фрілансер змінює статус на або "в роботі", або "виконано". Після чого роботодавець може змінити статус на "переглядається" і "виконано" - піся чого відбувається переказ коштів на гаманець фрілансера. З точки зору фінансового обліку, в залежності від транзакції йде запис в таблиці ContractPayment, Payment, WalletTransaction, а самі кошти зберігаються в таблиці User wallet. Після успішного завершення роботи над проектом як і роботодавець так і фрілансер можуть залишити відгуки про співпрацю через Табличку Review.

Однак бізнес-процес передбачає і сценарії відхилення. Якщо виникають певні труднощі в роботі, наприклад пропущений дедлайн або виконана робота не так як домовлено в контракті, будь яка з сторін може відкрити Dispute. Ця дія призупиняє фінансові операції за контрактом. Спір має вирішуватись модератором, який завдяки наданим свідченням може змінити статус Dispute, статус contract milestone і створити сам DisputeResolution. У DisputeResolution буде описане рішення яке було прийняти модератором. Цей комплексний алгоритм гарантує безпеку, прозорість та контрольованість кожного етапу співпраці на платформі.

1.4. Огляд ринку програмних продуктів та порівняльний аналіз рішень

1.4.1. Аналіз провідних глобальних платформ-аналогів

Сучасний глобальний ринок віддаленої роботи характеризується високим рівнем конкуренції та домінуванням кількох великих технологічних платформ, які встановили базові стандарти взаємодії між замовниками та спеціалістами. Для якісного проектування власної інформаційної системи я провів глибокий аналіз найпопулярніших платформ-аналогів, таких як Upwork, Fiverr та Freelancer.com. Upwork є беззаперечним лідером у корпоративному сегменті та спеціалізується на довгострокових контрактах із розширеним функціоналом трекінгу робочого часу (Time Tracker). Її сильною стороною є потужна система фільтрації та корпоративні інструменти для управління цілими командами віддалених працівників.

Платформа Fiverr пропонує кардинально іншу бізнес-модель, яку можна назвати "продуктизацією послуг". Замість класичного пошуку проектів та подачі заявок, фрілансери тут створюють готові пакети послуг (Gigs) із фіксованою ціною та чіткими термінами виконання. Сильною стороною такого підходу є максимальне спрощення процесу покупки для замовника, що нагадує звичайний інтернет-магазин. Платформа Freelancer.com, у свою чергу, виділяється потужними інструментами гейміфікації та проведенням конкурсів (Contests), де замовник може отримати десятки готових варіантів роботи ще до оплати, що особливо популярно у сфері дизайну. Дослідження цих платформ дозволило мені зібрати найкращі практики організації користувацького інтерфейсу, структурування профілів та етапності фінансових розрахунків для імплементації їх у моєму власному продукті.

1.4.2. Порівняльна характеристика та виявлення архітектурних недоліків існуючих систем

Незважаючи на статус лідерів ринку, детальний порівняльний аналіз виявив низку суттєвих недоліків існуючих систем, які створюють значний дискомфорт для

користувачів та відкривають нішу для нових конкурентних рішень. Найбільш очевидною проблемою є агресивна фінансова політика: платформи часто стягують високі комісійні збори (від 10% до 20%) з фрілансерів за кожну транзакцію, а також додаткові приховані комісії із замовників за поповнення рахунку та конвертацію валют. Це змушує досвідчених користувачів шукати способи обходу систем платформи для прямої співпраці, що руйнує екосистему маркетплейсу.

З технічної точки зору, головним недоліком більшості гігантів є використання застарілої монолітної архітектури. Оскільки ці платформи створювалися більше десяти років тому, їхні бекенд-системи обтяжені величезною кількістю легасі-коду (legacy code). Це призводить до повільного завантаження динамічного контенту та ускладнює впровадження нових фіч. Крім того, на багатьох платформах відсутні справді оптимізовані механізми реального часу для комунікації. Замість повноцінного використання постійних веб-сокет з'єднань, чати часто працюють за принципом короткого опитування (short polling) або мають значні затримки в доставці повідомлень. Це створює серйозні бар'єри в моменти, коли замовнику та виконавцю потрібно миттєво обговорити критичні зміни в технічному завданні або оперативно вирішити проблему під час здачі проекту.

1.4.3. Постановка задачі дослідження та формування вимог до програмного продукту

Враховуючи виявлені недоліки існуючих рішень, метою моєї кваліфікаційної роботи є створення сучасної, високопродуктивної та гнучкої платформи фріланс-маркетплейсу, яка забезпечить користувачам безпечне, прозоре та миттєве середовище для співпраці. Для досягнення цієї мети я сформулював чіткі архітектурні та технологічні вимоги до розроблюваного програмного продукту.

Серверна частина (API) повинна бути побудована на базі принципів чистої архітектури (Clean Architecture), яка структурно розділена на Domain layer, Business logic layer, Data access layer та API layer. Для подолання проблеми монолітності та забезпечення високої швидкодії при асиметричних навантаженнях, я поставив завдання імплементувати архітектурний патерн CQRS з використанням бібліотеки

MediatR. Це дозволить розділити операції читання та запису. Крім того, система має включати автоматичну перевірку даних введених у команди за допомогою бібліотеки FluentValidation.

В якості сховища даних вимагається використання надійної реляційної СУБД PostgreSQL, до якої будуть звертатися через реалізацію generic репозиторіїв. Для вирішення проблеми затримок у спілкуванні, система повинна використовувати технологію SignalR для реалізації миттєвих push-сповіщень. Фінансове ядро має бути реалізоване з використанням платіжного шлюзу Stripe для безпечної оплати депозиту з обов'язковим врахуванням поля валюти.

Клієнтська частина повинна бути розроблена у вигляді інтерактивного односторінкового додатку (SPA) на базі React із використанням сучасного інструментарію RTK Query для кешування станів. Це забезпечить миттєву реакцію інтерфейсу на дії користувача без необхідності повного перезавантаження сторінок. Нарешті, для гарантування стабільності продукту, висувається вимога до створення понад 200 тестів для швидкого виявлення помилок та налаштування контейнеризації через Docker і docker-compose. Реалізація цих вимог дозволить мені створити платформу, яка якісно перевершує існуючі аналоги за показниками швидкодії, надійності та зручності користувацького інтерфейсу.

Враховуючи виявлені недоліки існуючих рішень, метою кваліфікаційної роботи є створення сучасної, високопродуктивної та гнучкої платформи фріланс-маркетплейсу. На основі аналізу предметної області сформовано функціональні вимоги до програмного продукту, класифіковані за методологією MoSCoW (Must have / Should have / Could have / Won't have), як показано на Рис. 1.2. Критичними (Must) визначено 12 вимог, що забезпечують повний життєвий цикл співпраці: від реєстрації та авторизації через JWT/Google OAuth до фінансових розрахунків через Stripe та системи відгуків. Вимоги категорії Should (арбітраж, адмін-панель, модерація) підвищують довіру до платформи. Перспективні напрями (Could) — AI-метчинг та push-сповіщення — закладені в архітектуру для подальшого масштабування. Мікросервісна архітектура та мобільний додаток (Won't) свідомо відкладені на наступні етапи розвитку продукту.

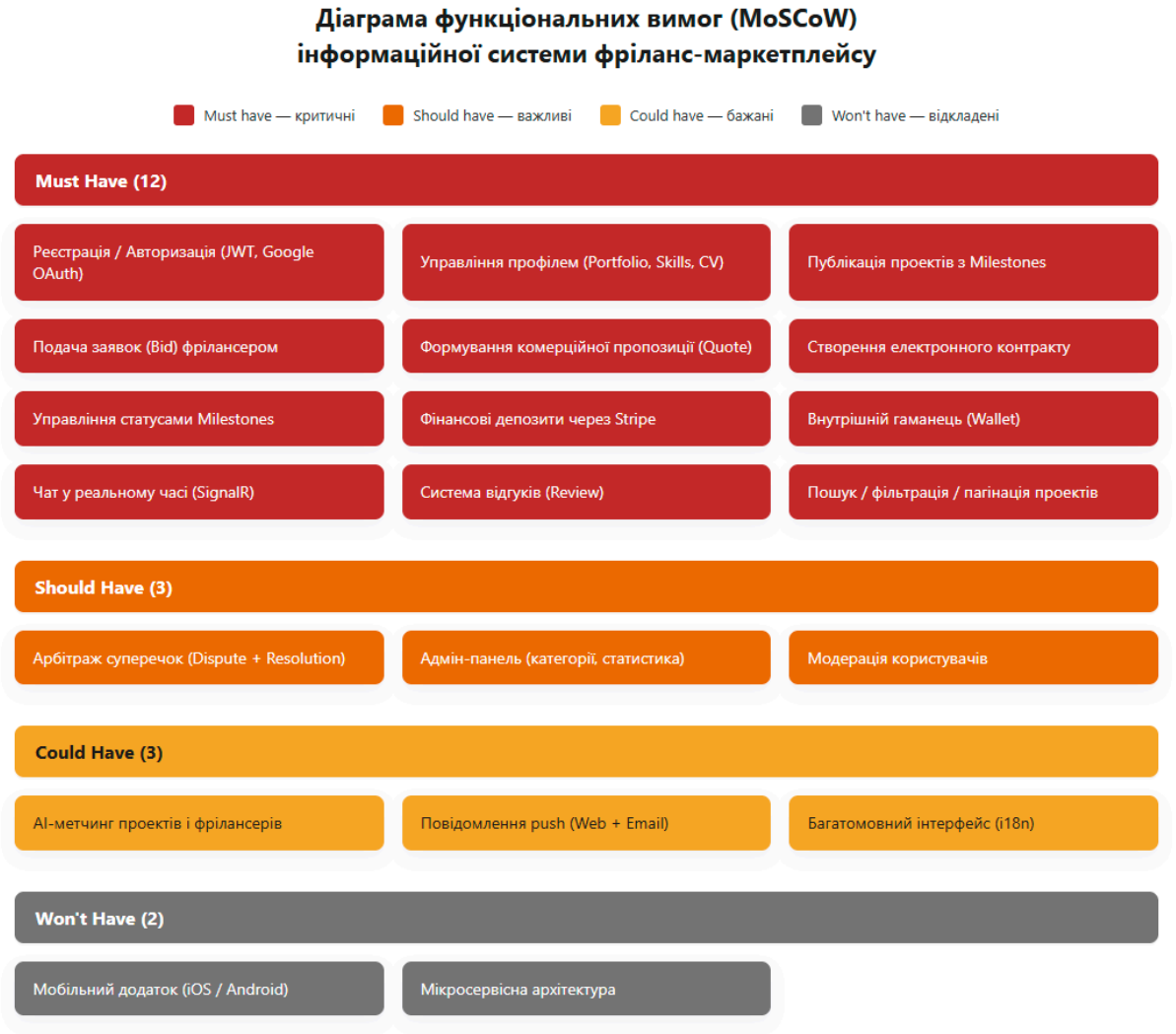


Рис. 1.2. Діаграма функціональних вимог (MoSCoW) інформаційної системи фріланс-маркетплейсу

Джерело: розроблено автором на основі постановки задачі дослідження (розд. 1.4.3) та архітектури системи (розд. 3.3)

Висновки до розділу 1

У першому розділі моєї дипломної роботи було проведено комплексний аналіз предметної області та детально досліджено специфіку функціонування глобального ринку віддаленої роботи. Я здійснив класифікацію послуг та форматів співпраці, що дозволило мені чітко визначити цільову аудиторію мого майбутнього програмного продукту, а саме замовників та спеціалістів категорії фріланс. Дослідження існуючої нормативно-правової бази та міжнародних стандартів дало мені змогу сформувати

строгі нефункціональні вимоги до безпеки моєї інформаційної системи. Зокрема, я обґрунтував критичну необхідність відповідності європейському регламенту GDPR щодо обробки персональних даних, а також обов'язковість дотримання фінансового стандарту PCI DSS, що у моєму проекті буде реалізовано завдяки делегуванню процесів токенізації та обробки платежів захищеному шлюзу Stripe.

Під час огляду ринку існуючих програмних продуктів я проаналізував домінуючі платформи, такі як Upwork та Fiverr. Цей порівняльний аналіз дозволив мені виявити суттєві недоліки конкурентів, серед яких варто виділити високі комісійні збори, перевантажені інтерфейси та, найголовніше, застарілу монолітну архітектуру, яка обмежує можливості систем у забезпеченні миттєвої реакції на дії користувачів. Опис функціональної моделі та бізнес-процесів моєї системи допоміг мені візуалізувати архітектурну логіку від моменту публікації проекту до фінального розрахунку між сторонами. На основі всіх цих аналітичних даних я сформулював остаточну постановку задачі мого дослідження.

Я дійшов висновку, що для створення конкурентоспроможної, надійної та масштабованої платформи мені необхідно відмовитися від класичних підходів на користь сучасного технологічного стеку та оптимізованих архітектурних патернів. Саме тому концепція мого рішення базується на використанні .NET 8 із суворим розділенням відповідальності за принципом CQS за допомогою бібліотеки MediatR. Для забезпечення абсолютної цілісності транзакційних даних я обрав реляційну СУБД PostgreSQL у поєднанні з патерном Repository, а для реалізації інтерактивного спілкування в реальному часі — технологію веб-сокетів SignalR. Вибір клієнтських технологій, таких як React на базі швидкого збирача Vite разом із Redux Toolkit та RTK Query, був продиктований моєю необхідністю створити максимально відгукливий інтерфейс, здатний ефективно управляти складним глобальним станом. Таким чином, результати, отримані в першому розділі, сформували міцне аналітичне та концептуальне підґрунтя для подальшого математичного, алгоритмічного та архітектурного проектування моєї інформаційної системи в наступних розділах роботи.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Аналіз предметної області платформи

2.1.1. Виділення ключових об’єктів дослідження інформаційної системи

У контексті обраного технологічного стеку на базі .NET 8 та реляційної системи управління базами даних PostgreSQL, кожен виділений об'єкт дослідження проектується як окрема доменна сутність мовою C# з відповідним відображенням у табличній структурі. Базовим об'єктом виступає «Користувач», який інкапсулює унікальні ідентифікатори, криптографічні хеші паролів, токени доступу та логічне розмежування ролей між замовниками та спеціалістами категорії фріланс. Цей об'єкт безпосередньо пов'язаний із сутністю «Профіль», що містить розширену інформацію: історію рейтингів, портфоліо та масиви професійних навичок, які на рівні бази даних можуть зберігатися у форматі JSONB для оптимізації повнотекстового пошуку. Наступним комплексним об'єктом є «Проект», який характеризується не лише базовими атрибутами на кшталт заголовка та бюджету, але й складними часовими мітками, географічними обмеженнями та статусною моделлю. Для відображення конкурентного середовища платформи виділено об'єкт «Пропозиція», що створюється, коли фріланс відгукується на проект, фіксуючи запропоновану ставку, мотиваційний лист та очікувані терміни реалізації. Процес комунікації базується на об'єкті «Повідомлення», що несе в собі текстове навантаження, інформацію про відправника, отримувача та статус прочитання, який оптимізований для миттєвої передачі через веб-сокети. Фінансова архітектура описується об'єктом «Транзакція», який нерозривно пов'язаний із зовнішніми ідентифікаторами платіжної системи Stripe, фіксуючи суму, валюту, утриману комісію маркетплейсу та поточний стан платежу. Логічним завершенням циклу взаємодії є об'єкт «Відгук», що формує кількісну та якісну оцінку співпраці, яка

згодом агрегується для динамічного перерахунку глобального рейтингу користувачів.

2.1.2. Побудова інформаційної моделі взаємодії між замовником та фріланс

Інформаційна модель детально регламентує життєвий цикл процесів та потоки даних між фронтенд-додатком на React та бекенд-інфраструктурою, використовуючи патерн CQS для суворого розділення запитів на читання та команд на зміну стану системи. Взаємодія ініціюється етапом публікації, коли замовник через інтерфейс генерує команду створення проекту, яка після успішної перевірки записується в PostgreSQL, змінюючи стан системи та ініціюючи оновлення пошукових індексів. Після цього фріланс, використовуючи оптимізовані запити на читання, знаходить релевантне завдання та надсилає власну пропозицію. На етапі обговорення деталей інформаційна модель переходить у режим реального часу, де комунікація забезпечується постійним двостороннім з'єднанням через SignalR, що дозволяє миттєво доставляти повідомлення та оновлювати статуси без необхідності перезавантаження сторінки клієнтом. Коли сторони досягають згоди, генерується команда на формування контракту, яка ініціює безпечну інтеграцію зі Stripe. Для забезпечення неперервності та цілісності фінансового процесу навіть на етапах розробки, коли локальне середовище тимчасово відкривається для зовнішніх запитів через захищені тунелі, інформаційна модель передбачає асинхронну обробку вебхуків. Ці зовнішні сигнали від платіжного шлюзу підтверджують успішне резервування коштів, автоматично переводячи контракт у статус активного виконання. Після здачі роботи та її затвердження замовником, система генерує фінальну команду на вивільнення депозиту, розподіляючи кошти між рахунком, який належить фріланс, та системним рахунком платформи, після чого життєвий цикл проекту закривається збереженням взаємних відгуків.

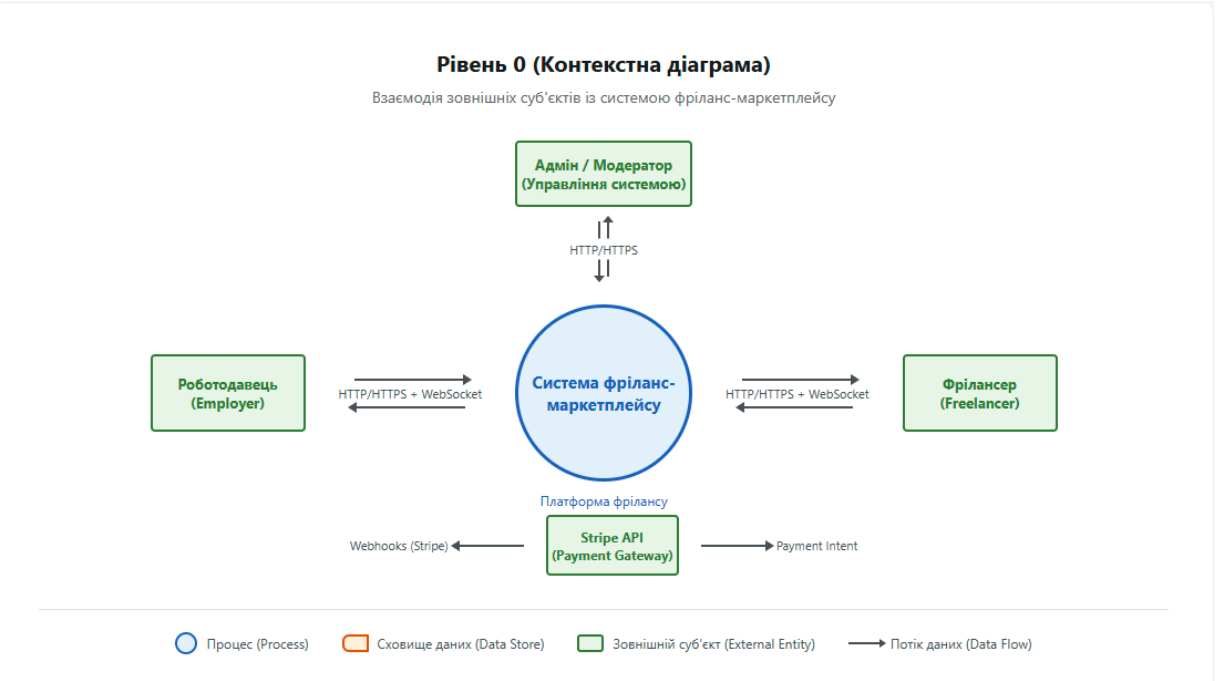


Рис. 2.1 Діаграма потоків даних (DFD) рівня 0 інформаційної системи фріланс-маркетплейсу

Джерело: створено AI на основі промпту автора

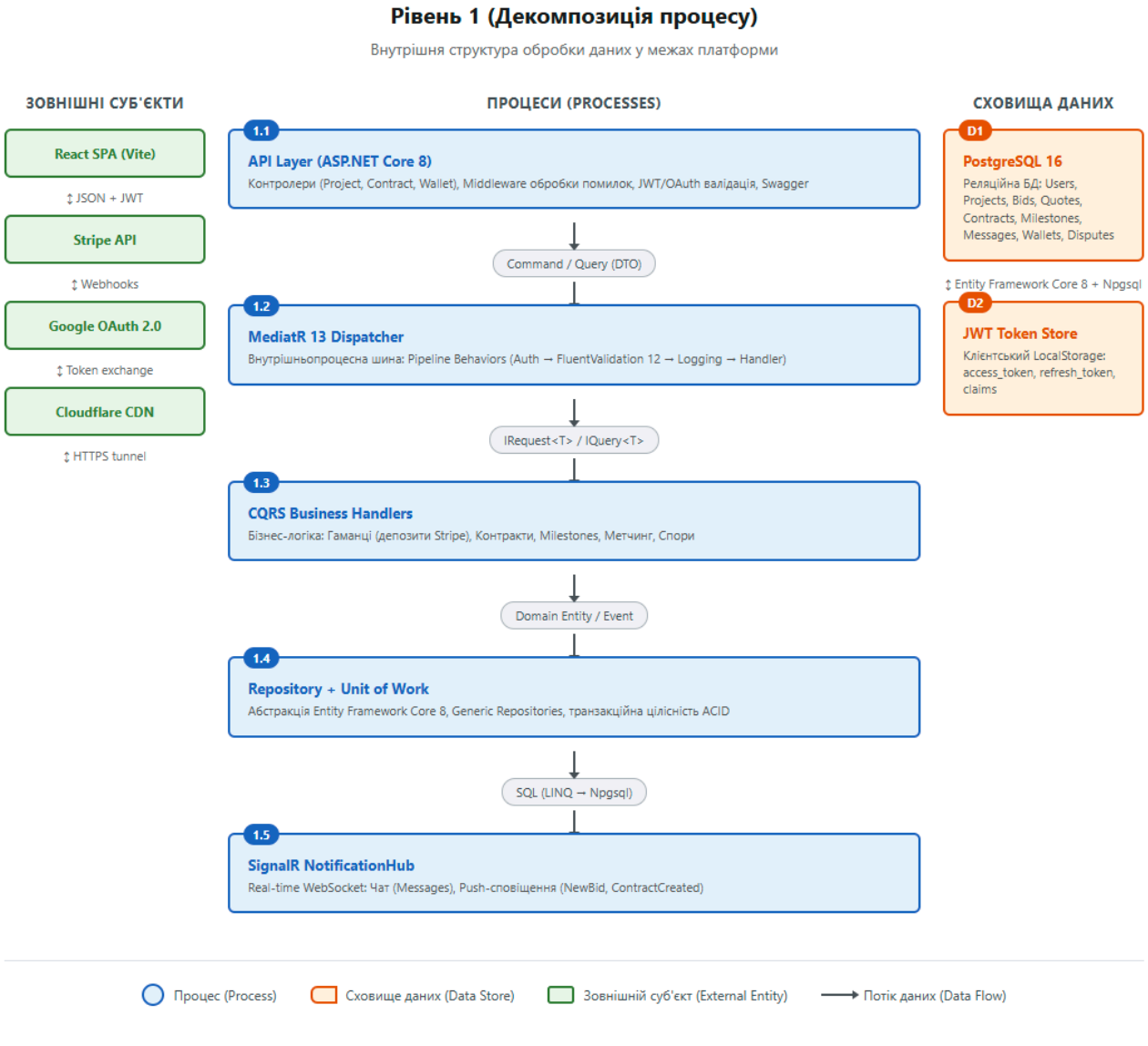


Рис. 2.2 Діаграма потоків даних (DFD) рівня 1 інформаційної системи фріланс-маркетплейсу

Джерело: створено AI на основі промпту автора

2.1.3. Опис існуючих обмежень на вхідні та вихідні дані API

Архітектура платформи передбачає багаторівневу систему обмежень для забезпечення стабільності, захисту від вразливостей та оптимізації мережевого трафіку. На рівні клієнтського додатку первинна фільтрація вхідних даних здійснюється за допомогою зв'язки React Hook Form, що не дозволяє відправити

запит до API, якщо інформація не відповідає суворим декларативним схемам. На серверній стороні ці обмеження повністю дублюються та розширюються в межах конвеєра обробки запитів завдяки механізмам Pipeline Behaviors бібліотеки MediatR. Усі текстові поля мають жорсткі ліміти на довжину на рівні колонок бази даних типу VARCHAR для запобігання атакам, пов'язаним із переповненням пам'яті. Оскільки бекенд-сервіси проектуються з урахуванням розгортання в ізольованих Docker-контейнерах через автоматизовані CI/CD конвеєри, існують суворі системні обмеження на максимальний розмір корисного навантаження запиту, зокрема під час завантаження медіафайлів для портфоліо, що додатково контролюється конфігурацією веб-сервера. Вихідні дані API також суттєво лімітовані, тому будь-які списки обов'язково повертаються виключно через механізми пагінації. Для управління станом на стороні клієнта використовується RTK Query, що накладає власні правила на інвалідацію кешу та запобігає надлишковим мережевим викликам до бази даних. Додатково застосовуються обмеження частоти запитів на рівні ендпоінтів для захисту від автоматизованих атак, а також суворі правила авторизації, згідно з якими вихідні дані фільтруються залежно від ролі: замовник бачить лише публічну або довірену інформацію, тоді як фріланс отримує доступ виключно до деталей своїх поточних контрактів та персональних фінансових звітів.

2.2. Проектування системи

2.2.1. Концептуальне проектування архітектури на базі патернів CQRS та Repository

Концептуальне проектування архітектури сучасної інформаційної системи є фундаментальним етапом, який визначає здатність програмного продукту до подальшого масштабування, підтримки та інтеграції нових функціональних можливостей. У контексті розробки платформи для спеціалістів категорії фріланс та замовників, архітектурне рішення базується на принципах чистої архітектури (Clean Architecture) з використанням платформи .NET 8. Основною парадигмою

проектування серверної частини було обрано патерн CQS (Command Query Separation), який в архітектурному масштабі трансформується у CQRS (Command Query Responsibility Segregation). Цей підхід кардинально змінює традиційне уявлення про монолітні додатки, чітко розділяючи потоки даних на дві незалежні магістралі: операції читання (запити) та операції зміни стану (команди). Таке розділення є критично важливим для платформ з асиметричним навантаженням, де кількість запитів на перегляд каталогів проектів або профілів значно перевищує кількість операцій зі створення нових сутностей чи проведення платежів. Завдяки CQRS система отримує здатність до незалежного масштабування цих складових, дозволяючи в майбутньому оптимізувати бази даних для читання окремо від баз даних для запису.

Технічна реалізація даного патерну досягається завдяки використанню потужної диспетчерської бібліотеки MediatR, яка виконує роль внутрішньопроцесного медіатора. Замість прямого виклику сервісів з контролерів, клієнтські HTTP-запити трансформуються у строго типізовані об'єкти команд або запитів, які передаються в шину MediatR. Бібліотека самостійно знаходить відповідний обробник (Handler), гарантуючи слабку зв'язність (loose coupling) між компонентами системи. Надзвичайно важливою перевагою такого підходу є можливість використання механізму конвеєрної обробки (Pipeline Behaviors). Це дозволяє елегантно вирішувати наскрізні завдання (cross-cutting concerns), такі як логування дій користувачів, глобальна обробка виключень та перевірка прав доступу, не забруднюючи при цьому код основної бізнес-логіки. Крім того, саме на рівні конвеєра реалізується первинна валідація вхідних даних для команд, що унеможливлює потрапляння некоректної або шкідливої інформації до доменного рівня додатку.

Для забезпечення надійної взаємодії обробників команд із реляційною базою даних PostgreSQL використовується структурний патерн Repository у поєднанні з концепцією Unit of Work. Патерн Repository створює абстрактний шар доступу до даних, ізолюючи бізнес-логіку від специфіки конкретного ORM-фреймворку та синтаксису SQL-запитів. Це означає, що доменні сервіси працюють із колекціями

об'єктів у пам'яті, не маючи жодного уявлення про те, як саме ці дані фізично зберігаються на диску. Такий рівень абстракції є абсолютно необхідним для забезпечення високого рівня тестованості коду. Завдяки репозиторіям, під час написання модульних та інтеграційних тестів розробник може легко підмінити реальну базу даних PostgreSQL на її імітацію (mock) або використовувати легкоструменеву базу даних у пам'яті. У свою чергу, патерн Unit of Work гарантує транзакційну цілісність бізнес-операцій, відстежуючи всі зміни, внесені до сутностей протягом одного бізнес-сценарію, і синхронно фіксуючи їх у базі даних єдиною транзакцією або скасовуючи всі зміни у разі виникнення помилки під час виконання, що є критичним для обробки фінансових операцій зі Stripe.

Проектування клієнтської частини системи, яка реалізується за допомогою бібліотеки React та середовища збірки Vite, також вимагає суворого архітектурного контролю. Використання TypeScript додає статичну типізацію на рівні фронтенду, що дозволяє синхронізувати моделі даних з бекендом і виявляти архітектурні невідповідності ще на етапі компіляції коду. Управління глобальним станом додатку та кешуванням мережевих запитів делеговано екосистемі Redux Toolkit (RTK), зокрема модулю RTK Query. Архітектурно це означає відмову від ручного управління станами завантаження чи помилок у компонентах. RTK Query бере на себе відповідальність за взаємодію з REST API мого бекенду, автоматично кешує результати запитів і фоново оновлює їх при зміні пов'язаних даних. Це ідеально доповнює серверну архітектуру CQRS, оскільки фронтенд чітко розрізняє мутації (команди) та запити на читання, автоматично інвалідуючи кеш списку проектів після успішного виконання мутації зі створення нового замовлення.

Окремим архітектурним викликом є інтеграція механізмів реального часу та обробка зовнішніх асинхронних подій. Для реалізації миттєвих сповіщень та чату проектується використання технології WebSockets через екосистему SignalR. Архітектурно хаби SignalR виступають ще однією точкою входу в систему, паралельно з REST-контролерами. Вони також інтегровані з шиною MediatR, що дозволяє подіям, згенерованим у веб-сокеті (наприклад, відправка повідомлення), проходити той самий конвеєр валідації та обробки, що й звичайні HTTP-запити.

Фінансова підсистема на базі Stripe вимагає проектування безпечних вебхук-ендпоінтів, які повинні відповідати принципу ідемпотентності — здатності системи коректно обробляти один і той самий сигнал про успішний платіж декілька разів без ризику подвійного нарахування коштів. Надійність усіх цих архітектурних рішень підтверджується на етапі розробки комплексним покриттям коду інтеграційними тестами. Вони розгортають повноцінне тестове середовище з реальною базою даних PostgreSQL у Docker-контейнерах та перевіряють працездатність усього ланцюжка: від відправки HTTP-запиту, проходження через MediatR, збереження даних у репозиторії до генерації відповідного сповіщення через SignalR.

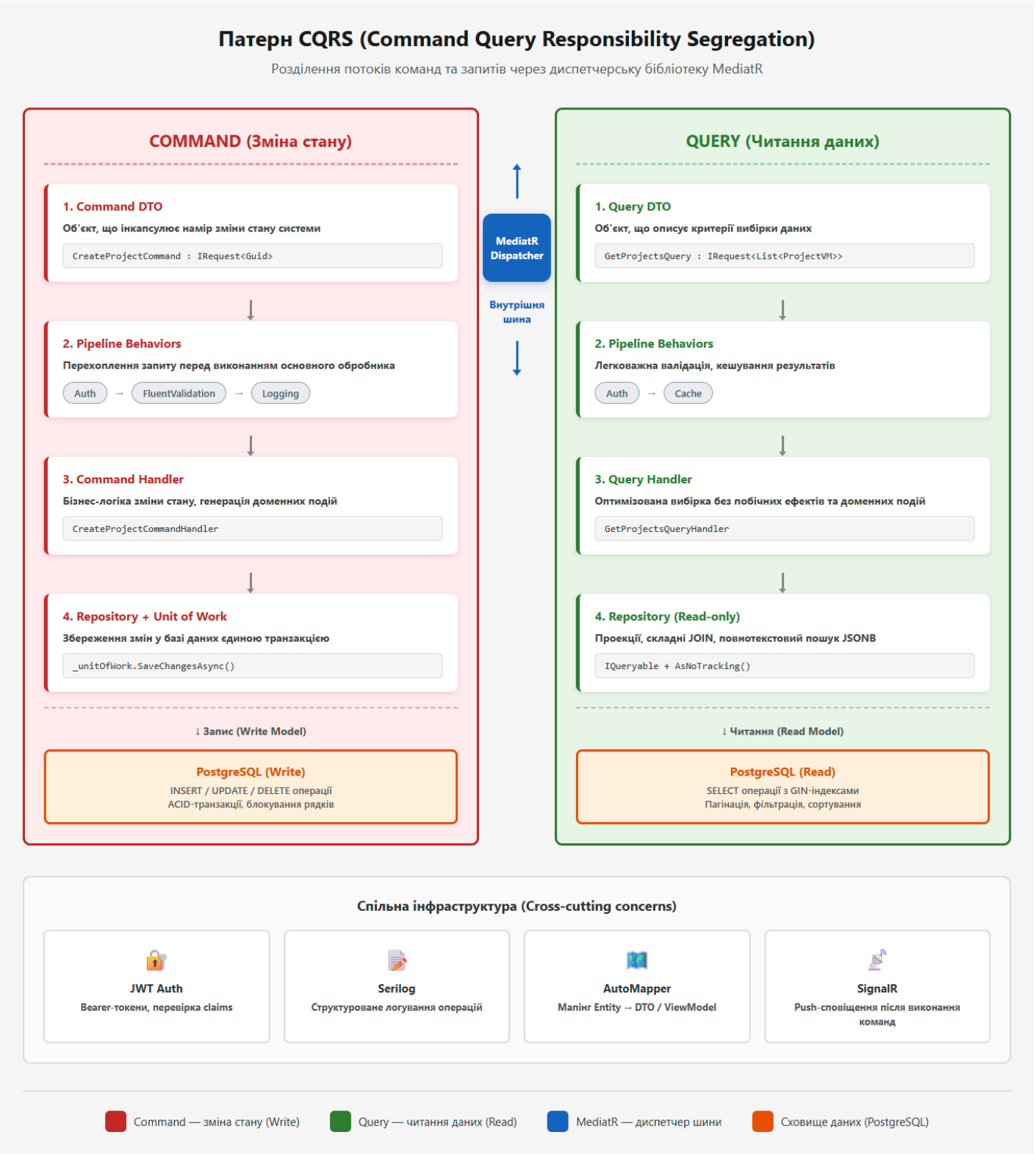


Рис. 2.3. Діаграма реалізації патерну CQRS через диспетчерську бібліотеку MediatR

Джерело: створено AI на основі промпту автора

2.2.2. Побудова концептуальної моделі предметної області на основі принципів предметно-орієнтованого проектування

Відходячи від класичних статичних графічних нотацій, концептуальне проектування розроблюваної платформи для віддаленої роботи базується на методології предметно-орієнтованого проектування (Domain-Driven Design, DDD). Цей підхід дозволяє перенести фокус із простого опису структур даних на глибоке моделювання складних бізнес-правил та процесів, що відбуваються на ринку фріланс-послуг. Фундаментом концептуальної моделі в рамках DDD є формування «єдиної мови» (Ubiquitous Language), яка гарантує, що розробники, архітектори та експерти предметної області використовують абсолютно ідентичну термінологію для опису процесів. На базі цієї мови інформаційна система концептуально розділяється на ізольовані семантичні межі, відомі як обмежені контексти (Bounded Contexts). Таке розділення ідеально синхронізується з обраною архітектурою CQRS та дозволяє уникнути створення монолітних, сильно зв'язаних структур. У моїй платформі виділяється чотири ключові обмежені контексти: контекст управління ідентичністю та профілями, контекст управління проектами та контрактами, фінансовий контекст для обробки транзакцій та комунікаційний контекст, що відповідає за обмін повідомленнями.

Усередині кожного обмеженого контексту концептуальна модель деталізується через побудову агрегатів (Aggregates), сутностей (Entities) та об'єктів-значень (Value Objects). Агрегат виступає як кластер пов'язаних об'єктів, які розглядаються як єдине ціле з точки зору зміни даних, і має свій корінь (Aggregate Root), через який відбуваються будь-які транзакції. Наприклад, у контексті управління проектами головним коренем агрегату є сутність «Проект». Вона інкапсулює в собі правила валідації та повністю контролює життєвий цикл підпорядкованих сутностей, таких як «Пропозиція» від спеціаліста категорії фріланс або «Етап виконання». Будь-яка зміна статусу пропозиції чи оновлення технічного завдання відбувається виключно через виклик методів кореня агрегату, що гарантує абсолютну консистентність бізнес-логіки перед тим, як патерн Repository зафіксує ці зміни у базі даних

PostgreSQL. Для опису атрибутів, які не мають власної ідентичності, але є критично важливими для логіки, концептуальна модель використовує об'єкти-значення. Найяскравішим прикладом є об'єкт «Гроші» (Money), який містить не лише числове значення суми, але й тип валюти. Це є критично необхідним для правильної інтеграції зі Stripe, оскільки платіжний шлюз вимагає абсолютно точного та безпечного форматування фінансових даних без ризику виникнення помилок заокруглення, притаманних стандартним примітивним типам даних.

Динамічна поведінка концептуальної моделі реалізується через механізм доменних подій (Domain Events), що є сполучною ланкою між предметно-орієнтованим проектуванням та обраним патерном CQRS на базі бібліотеки MediatR. Коли всередині агрегату відбувається важлива бізнес-дія, наприклад, замовник затверджує пропозицію і формує контракт, агрегат не звертається напряму до сторонніх сервісів, а генерує відповідну доменну подію (наприклад, ContractCreatedEvent). Ця модель поведінки забезпечує слабку зв'язність системи. Згенерована подія публікується у внутрішньопроцесну шину MediatR, де її можуть перехопити декілька незалежних обробників. Один обробник може відповісти за ініціалізацію платіжної сесії через API Stripe, резервуючи кошти на ескроу-рахунку. Інший обробник паралельно сформує команду для оновлення пошукових індексів або проекцій читання в PostgreSQL, забезпечуючи актуальність даних для швидких запитів клієнтського додатку. Третій обробник перехопить цю ж подію для того, щоб передати команду сервісу SignalR, який миттєво відправить веб-сокет сповіщення на клієнтський додаток обох сторін угоди.

На рівні клієнтської архітектури, розробленої з використанням React, TypeScript та середовища Vite, концептуальна модель предметної області знаходить своє симетричне відображення. Типізація TypeScript дозволяє створити точні клієнтські копії серверних об'єктів-значень та моделей читання (Read Models), що повертаються через API. Управління станом цих концептуальних сутностей на фронтенді цілком делеговано інструментарію Redux Toolkit та RTK Query. Оскільки моя серверна концептуальна модель розділяє команди (мутації) та запити (отримання даних), RTK Query ідеально адаптується до цієї парадигми через

систему тегів інвалідації кешу. Коли клієнт ініціює мутацію, що змінює стан агрегату на сервері (наприклад, відправка нового повідомлення в чаті), RTK Query автоматично розуміє, які концептуальні моделі в локальному кеші стали неактуальними, і у фоновому режимі виконує синхронізацію з бекендом. Такий підхід до концептуального моделювання, відмовляючись від статичних малюнків на користь живої предметно-орієнтованої архітектури, забезпечує високу надійність, тестованість за допомогою інтеграційних тестів та гнучкість платформи в умовах постійного розширення бізнес-вимог ринку фрілансу.

2.2.3. Проектування структури реляційної бази даних PostgreSQL та розробка ER-діаграми

Проектування структури зберігання даних є найкритичнішим етапом створення будь-якої інформаційної системи, оскільки від нього безпосередньо залежить продуктивність, масштабованість, безпека та надійність усього програмного комплексу. Для реалізації моєї платформи було обрано об'єктно-реляційну систему управління базами даних PostgreSQL, яка є індустріальним стандартом для високонавантажених фінансових та комунікаційних додатків завдяки своїй повній відповідності принципам ACID (атомарність, узгодженість, ізольованість, довговічність) та підтримці багатоверсійного управління конкурентним доступом (MVCC). Процес проектування розпочинається з логічного моделювання та нормалізації даних до третьої нормальної форми (3NF). Цей підхід гарантує мінімізацію надмірності збереженої інформації та повне усунення аномалій оновлення, вставлення або видалення записів. В якості первинних ключів для абсолютно всіх таблиць застосовується формат універсальних унікальних ідентифікаторів (UUID). На відміну від стандартних автоінкрементних цілих чисел, UUID генеруються децентралізовано, що робить систему готовою до потенційного горизонтального масштабування та шардингу бази даних у майбутньому, а також унеможливлює проведення атак типу перебору ідентифікаторів (enumeration attacks) з боку зловмисників.

Фізична структура бази даних розділена на кілька логічних схем, які відповідають обмеженим контекстам моєї архітектури. Ядром системи є таблиці управління ідентичністю та профілями, де реалізовано суворий розподіл між автентифікаційними даними (таблиця користувачів) та публічною інформацією (таблиця профілів). Оскільки ринок віддаленої роботи характеризується надзвичайною динамічністю професійних навичок, збереження компетенцій спеціаліста категорії фріланс у класичній реляційній структурі було б неефективним. Тому для опису навичок, сертифікатів та елементів портфоліо використовується спеціалізований тип даних JSONB, який підтримується в PostgreSQL. Це архітектурне рішення дозволяє зберігати неструктуровані масиви інформації безпосередньо в таблиці профілю та накладати на них інвертовані індекси (GIN), що забезпечує блискавичну швидкість виконання складних повнотекстових пошукових запитів при метчингу проектів. Таблиці проектів та пропозицій пов'язані суворими обмеженнями зовнішніх ключів (Foreign Keys) з каскадними правилами м'якого видалення (Soft Delete), що означає збереження історичних даних у базі навіть після їх візуального приховування в інтерфейсі клієнтського додатку на React.

Для візуалізації цих складних взаємозв'язків та обмежень розроблена детальна логічна модель, яка відображає архітектуру даних моєї системи що подана в **додатку А**.

Окремої уваги вимагає проектування комунікаційного та фінансового модулів бази даних. Для забезпечення надійної роботи інтегрованої системи сповіщень на базі SignalR та веб-сокетів, структура таблиць повідомлень оптимізована для високочастотних операцій запису (Insert). Таблиця повідомлень містить індексовані часові мітки та логічні прапорці статусу прочитання, що дозволяє швидко витягувати невеликі порції історії діалогів за допомогою оптимізованих запитів на читання з боку MediatR, мінімізуючи затримки у передачі даних. Фінансовий модуль спроектовано з урахуванням найсуворіших стандартів обробки транзакцій. Таблиця фінансових контрактів та платежів використовує виключно точні числові формати даних, такі як DECIMAL або NUMERIC із фіксованою точністю, для унеможливлення будь-яких математичних похибок під час заокруглення сум або

розрахунку комісійних зборів. Крім того, ця таблиця містить спеціальні індексовані колонки для зберігання зовнішніх ідентифікаторів платіжної сесії (Session ID) та намірів платежу (Payment Intent ID) від екосистеми Stripe.

Надійність спроектованої структури бази даних перевіряється та підтверджується на етапі розробки за допомогою комплексного набору інтеграційних тестів. Оскільки моя бізнес-логіка ізольована абстракцією патерну Repository, інтеграційні тести піднімають тимчасові ізольовані екземпляри PostgreSQL (наприклад, за допомогою технології Testcontainers), автоматично накочують усі міграції схеми та виконують реальні транзакції. Це дозволяє гарантовано перевірити спрацювання обмежень унікальності, каскадних видалень та коректність роботи індексів ще до моменту розгортання інформаційної системи в робочому середовищі. Такий симбіоз надійної реляційної структури, сучасних типів даних та суворого інтеграційного тестування створює бездоганний фундамент для стабільного функціонування платформи під будь-якими навантаженнями.

2.3. Алгоритмічне забезпечення та логіка управління даними

2.3.1. Алгоритмічні моделі реалізації основної бізнес-логіки та архітектурних патернів

Оскільки розробка сучасних веб-орієнтованих систем фокусується на обробці великих масивів інформації та забезпеченні високої швидкодії, класичне математичне моделювання у моєму дипломному проекті поступається місцем складному алгоритмічному забезпеченню та моделюванню бізнес-процесів. В основі функціонування серверної частини моєї платформи лежить алгоритмічна модель конвеєрної обробки даних, яка базується на архітектурному принципі CQS та реалізована за допомогою бібліотеки MediatR. Алгоритм виконання будь-якої операції розпочинається з ініціалізації запиту на стороні мого клієнтського додатку, створеного за допомогою React та Vite. Після відправки мережевого запиту управління переходить до бекенду, де алгоритм маршрутизації MediatR перехоплює HTTP-запит і пропускає його через серію проміжних обробників (Pipeline Behaviors).

Цей конвеєрний алгоритм послідовно виконує автентифікацію токена, авторизацію прав доступу до конкретної дії та глибинну бізнес-валідацію. Лише у випадку успішного проходження всіх цих перевірок алгоритм передає виконання основному класу-обробнику, який імплементує безпосередню логіку зміни стану системи або вибірки даних.

Алгоритмічна взаємодія з базою даних у моєму проекті повністю інкапсульована в межах патерну Repository. Це означає, що алгоритми доменного рівня працюють виключно з абстрактними колекціями об'єктів, тоді як репозиторій відповідає за трансляцію цих дій у конкретні SQL-запити до PostgreSQL. Для забезпечення ефективного пошуку проектів та спеціалістів категорії фріланс я розробив алгоритм метчингу, який спирається на інструментарій повнотекстового пошуку бази даних та роботу з типом даних JSONB. Алгоритм пошуку не обмежується простим порівнянням рядків, а генерує динамічне дерево логічних умов, аналізуючи масиви професійних навичок, збережених у неструктурованому форматі. При формуванні складних вибірок застосовується алгоритм пагінації на основі курсора або зміщення (offset/limit), що дозволяє обробляти потенційно величезні обсяги записів без перевантаження оперативної пам'яті сервера. Для гарантування безпомилковості роботи всіх цих алгоритмів я впровадив комплексне покриття коду інтеграційними тестами, які алгоритмічно імітують повний життєвий цикл запиту в ізольованому середовищі з реальною тестовою базою даних, перевіряючи коректність спрацювання кожного логічного розгалуження.

2.3.2. Алгоритми обробки фінансових транзакцій та асинхронної взаємодії в реальному часі

Алгоритмічне забезпечення фінансового ядра моєї платформи побудоване на основі концепції скінченного автомата (State Machine), що гарантує суворий контроль над життєвим циклом кожної транзакції. Алгоритм безпечної оплати стартує на фронтенді, де мій додаток за допомогою Redux Toolkit (RTK) ініціює запит на створення наміру платежу (Payment Intent). Серверна частина зв'язується з

API платіжного шлюзу Stripe і генерує унікальний секретний ключ сесії. Найважливішим алгоритмічним обмеженням тут є те, що повні реквізити платіжної картки ніколи не проходять через мій бекенд, а токенізуються безпосередньо скриптами Stripe на клієнті. Після успішного списання коштів емітентом алгоритм переходить до критичної фази обробки асинхронних вебхуків. Мій сервер отримує криптографічно підписаний сигнал від Stripe, алгоритмічно перевіряє його справжність шляхом порівняння хеш-сум і лише після цього ініціює команду на оновлення статусу контракту в PostgreSQL. Для унеможливлення подвійного зарахування коштів я застосував алгоритм ідемпотентності, який перевіряє унікальний ідентифікатор події Stripe і блокує повторне виконання логіки, якщо вебхук був надісланий платіжною системою більше одного разу через мережеві затримки. Діаграма станів платежу знаходиться в **додатку Б**.

Для організації безперервної двосторонньої комунікації між користувачами я розробив алгоритми управління веб-сокет з'єднаннями на базі технології SignalR. Алгоритм встановлення сеансу зв'язку передбачає передачу авторизаційного токена під час початкового рукостискання, після чого сервер алгоритмічно зіставляє унікальний ідентифікатор підключення (Connection ID) з ідентифікатором профілю користувача. Це дозволяє реалізувати точну адресну маршрутизацію. Коли замовник відправляє повідомлення, мій бекенд алгоритмічно зберігає текст у базу даних, визначає активні підключення отримувача та транслює подію безпосередньо в його сокет. На клієнтській стороні цей процес обробляється за допомогою алгоритмів RTK Query. Я налаштував інструментарій таким чином, щоб при отриманні сигналу від SignalR стан додатку оновлювався оптимістично (Optimistic Updates). Алгоритм локально модифікує кеш повідомлень або змінює статус замовлення в інтерфейсі без необхідності робити додатковий запит на сервер для завантаження всього списку. Це архітектурне рішення значно розвантажує серверну інфраструктуру моєї платформи та забезпечує користувачам миттєвий зворотний зв'язок, характерний для найсучасніших веб-додатків.

Висновки до розділу 2

У другому розділі моєї дипломної роботи було виконано комплексне інформаційне, архітектурне та алгоритмічне проектування платформи для віддаленої роботи. На етапі аналізу предметної області я виділив ключові сутності системи, такі як профілі користувачів, проекти, пропозиції, контракти та фінансові транзакції. Я побудував детальну інформаційну модель, яка описує життєвий цикл взаємодії замовника та спеціаліста категорії фріланс від моменту публікації технічного завдання до фінального безпечного розрахунку та обміну відгуками. Окрім цього, я визначив та задокументував суворі обмеження на вхідні та вихідні дані, спроектувавши надійний механізм дворівневої валідації. На сервері валідація відбувається через механізм конвеєрної обробки Pipeline Behaviors у межах бібліотеки MediatR за допомогою FluentValidation, що гарантує абсолютну консистентність даних перед їх потраплянням до бази.

Проектування архітектури моєї системи базувалося на принципах предметно-орієнтованого проектування (DDD), що дозволило мені розділити складну бізнес-логіку маркетплейсу на ізольовані обмежені контексти. Я обґрунтував стратегічний вибір архітектурного патерну CQRS для чіткого розмежування операцій читання та мутації стану системи. Це рішення в поєднанні з абстракцією Repository суттєво підвищило гнучкість мого коду та його готовність до написання комплексних інтеграційних тестів. Надзвичайно важливим кроком стало проектування структури реляційної бази даних PostgreSQL. Застосування глобальних ідентифікаторів UUID для захисту від перебору, використання формату JSONB для зберігання динамічних масивів професійних навичок у портфоліо та впровадження точних числових типів для фінансових колонок дозволило мені створити оптимізований та безпечний фундамент для зберігання даних моєї платформи.

Розробка алгоритмічного забезпечення та логіки управління даними стала завершальним етапом проектування у цьому розділі, повністю замінивши класичне математичне моделювання. Я описав алгоритми обробки команд, механізми

метчингу та складного повнотекстового пошуку релевантних проектів. Для фінансового ядра моєї платформи я спроектував алгоритмічну модель на базі скінченного автомата, яка надійно контролює всі етапи резервування та переказу коштів через інтеграцію зі Stripe. Ця модель включає критично важливий алгоритм ідемпотентної обробки зовнішніх вебхуків для беззаперечного захисту від подвійних фінансових транзакцій. Для забезпечення високої інтерактивності я розробив алгоритми маршрутизації миттєвих повідомлень та сповіщень через веб-сокети за допомогою SignalR. На стороні мого клієнтського додатку, побудованого на React та Vite, ці асинхронні події ефективно обробляються інструментарієм RTK Query із застосуванням стратегії оптимістичних оновлень інтерфейсу. Таким чином, результати другого розділу сформували вичерпну архітектурну специфікацію, яка є повністю готовою базою для безпосередньої програмної реалізації та розгортання мого програмного продукту.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Засоби розробки

3.1.1. Огляд та обґрунтування інструментарію серверної частини (Backend)

Вибір технологічного стеку для серверної частини моєї платформи фріланс-маркетплейсу був продиктований необхідністю забезпечення високої продуктивності, безпеки обробки фінансових транзакцій та здатності до масштабування під великими навантаженнями. Фундаментом бекенду я обрав платформу ASP.NET Core 8 у поєднанні з об'єктно-орієнтованою мовою програмування C#. Восьма версія фреймворку .NET надає передові можливості для створення високопродуктивних REST API, містить оптимізований компілятор (JIT) та покращений механізм збирання сміття (Garbage Collector). Завдяки вбудованій асинхронності та кросплатформності, ASP.NET Core ідеально підходить для розгортання в ізольованих контейнерах на серверах з операційними системами сімейства Linux. Мова C# зі своєю суворою статичною типізацією дозволяє мені виявляти більшість архітектурних та логічних помилок ще на етапі компіляції коду, що критично важливо при розробці складних бізнес-додатків із фінансовим модулем.

В якості основного сховища даних я обрав надійну реляційну систему управління базами даних PostgreSQL 16. Оскільки моя платформа працює з контрактами, фінансовими розрахунками та складними зв'язками між сутностями, використання реляційної СУБД з повною підтримкою транзакцій (ACID) є безальтернативним рішенням. Шістнадцята версія PostgreSQL пропонує розширені можливості роботи з типом даних JSONB для гнучкого зберігання портфоліо та навичок, а також оптимізовані механізми індексування. Для взаємодії між об'єктно-орієнтованим кодом мого додатку та реляційною базою даних я використав Entity Framework Core 8 у зв'язці з провайдером Npgsql. Цей сучасний ORM-інструментарій дозволяє мені використовувати міць мови інтегрованих запитів

(LINQ) для безпечного формування SQL-команд, а також керувати еволюцією структури бази даних за допомогою механізму міграцій (Code-First підхід), абстрагуючись від написання низькорівневих SQL-скриптів.

Лістинг 3.1. Підключення бази даних і репозиторіїв для використання в Program.cs.

```
public static void AddDataAccess(this IServiceCollection services,
WebApplicationBuilder builder)
    {
        var dataSourceBuild = new
NpgsqlDataSourceBuilder(builder.Configuration.GetConnectionString("Default"));

        dataSourceBuild.EnableDynamicJson();
        var dataSource = dataSourceBuild.Build();

        services.AddDbContext<AppDbContext>(options => options
            .UseNpgsql(
                dataSource,
                npgsqlDbContextOptionsBuilder =>

npgsqlDbContextOptionsBuilder.MigrationsAssembly(typeof(AppDbContext).Assembly.FullNam
e))
            .UseSnakeCaseNamingConvention()
            .ConfigureWarnings(w =>
w.Ignore(CoreEventId.ManyServiceProvidersCreatedWarning)));

        services.AddScoped<ApplicationDbContextInitializer>();
        services.AddRepositories();
    }
```

Для реалізації чистої архітектури та підтримки патерну CQRS я імплементував бібліотеку MediatR 13. Вона виступає в ролі внутрішньопроцесного диспетчера, розв'язуючи залежності між API-контролерами та обробниками бізнес-логіки. Важливою перевагою MediatR є підтримка механізму конвеєрної обробки (Pipeline Behaviors), в який я безшовно інтегрував бібліотеку FluentValidation. Це дозволило мені винести правила валідації вхідних даних (DTO) в окремі класи та забезпечити автоматичну перевірку всіх команд ще до того, як вони досягнуть доменного рівня. Для автоматичного перетворення складних доменних моделей у спрощені об'єкти передачі даних (View Models) я застосував інструмент AutoMapper, що значно скоротило обсяг шаблонного коду. Організація впровадження залежностей

(Dependency Injection) була оптимізована за допомогою бібліотеки Scrutor, яка автоматично сканує збірки та реєструє сервіси за інтерфейсами, позбавляючи мене необхідності прописувати кожну залежність вручну. Зрештою, для забезпечення надійного моніторингу та структурованого логування роботи мого API я використав потужний фреймворк Serilog із виведенням логів у консоль для зручного аналізу через логіку контейнерів.

3.1.2. Огляд та обґрунтування інструментарію клієнтської частини (Frontend)

Клієнтська частина мого маркетплейсу спроектована з фокусом на інтерактивність, швидкість відгуку та зручність підтримки кодової бази. В якості основного інструменту розробки інтерфейсу користувача я обрав популярну бібліотеку React. Її компонентний підхід дозволяє мені інкапсулювати логіку та візуальне представлення окремих частин інтерфейсу, роблячи код придатним для багаторазового використання. Замість класичних і повільних збирачів проектів я застосував сучасне середовище Vite, яке забезпечує блискавичний старт локального сервера розробки та миттєве гаряче оновлення модулів (HMR), що кардинально підвищує мою продуктивність. Увесь фронтенд-додаток написаний з використанням суворої типізації TypeScript. Цей вибір був зумовлений необхідністю синхронізувати типи даних між бекендом та клієнтом, що дозволяє автодоповненню в редакторі коду працювати безпомилково та запобігає виникненню несподіваних помилок у середовищі виконання (runtime errors) під час маппінгу JSON-відповідей від API.

Оскільки моя платформа оперує великими обсягами динамічних даних, які часто оновлюються в реальному часі (сповіщення, статуси контрактів, чати), управління глобальним станом потребувало надійного інструментарію. Для цього я інтегрував екосистему Redux Toolkit (RTK), яка є сучасним стандартом роботи з Redux, мінімізуючи необхідність написання великої кількості шаблонного коду (boilerplate). Особливу роль у моєму клієнтському додатку відіграє модуль RTK Query. Він повністю бере на себе завдання з виконання асинхронних HTTP-запитів до мого ASP.NET Core API, автоматично кешує результати вибірок даних та

самостійно керує станами завантаження (isLoading) і помилок (isError). Я налаштував систему тегів RTK Query таким чином, щоб після успішного виконання мутації (наприклад, створення нової пропозиції фрілансером) кеш відповідних запитів автоматично інвалідувався, викликаючи фонове оновлення даних в інтерфейсі без втручання користувача.

Для візуального оформлення платформи я відмовився від написання традиційних CSS-файлів чи використання важких UI-фреймворків на користь утилітарного підходу з Tailwind CSS. Цей фреймворк дозволяє мені стилізувати компоненти прямо в розмітці JSX, використовуючи готові атомарні класи, що гарантує єдину дизайн-систему та автоматично видаляє невикористані стилі у фінальній збірці, роблячи додаток надзвичайно легким. Значна частина взаємодії з моїм додатком відбувається через різноманітні форми: від авторизації до створення складних проектів з етапами (milestones). Для забезпечення максимальної продуктивності під час заповнення цих форм я використав бібліотеку React Hook Form. На відміну від традиційних підходів, вона мінімізує кількість перемалювань (re-renders) компонентів під час введення тексту та ідеально інтегрується зі схемами валідації, що дозволяє мені синхронізувати правила перевірки даних на клієнті з тими, що налаштовані у FluentValidation на бекенді.

3.1.3. Інфраструктурні та апаратні засоби розробки

Для забезпечення абсолютної ідентичності середовищ розробки, тестування та експлуатації я обрав технологію контейнеризації Docker. Увесь бекенд мого проекту, включаючи саму базу даних PostgreSQL 16 та зібраний ASP.NET Core додаток, описаний у вигляді інфраструктури як коду (Infrastructure as Code) у файлі docker-compose.yml. Це дозволяє мені розгорнути повноцінно працюючий комплекс бази даних та API однією командою, ізолювавши процеси від впливу локальної операційної системи. Кожен компонент працює у власному контейнері зі своїм набором змінних оточення (з файлу .env), що гарантує високий рівень безпеки та портативності мого програмного забезпечення.

Лістинг 3.2. Конфігураційний файл docker-compose.yml для серверного розгортання.

```
services:
  db:
    image: postgres:16
    container_name: freelance-db
    restart: unless-stopped
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: freelance-db
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

  api:
    image: ghcr.io/andromaan/freelance-api:latest
    container_name: freelance-api
    restart: unless-stopped
    ports:
      - "8080:8080"
    environment:
      ASPNETCORE_ENVIRONMENT: Production
      ASPNETCORE_URLS: http://+:8080
      ConnectionStrings__Default: "Server=db;Port=5432;Database=freelance-db;User
Id=postgres;Password=postgres;"
      GoogleAuthSettings__ClientId: ${GOOGLE_CLIENT_ID}
      AuthSettings__key: ${JWT_KEY}
      AuthSettings__issuer: ${JWT_ISSUER}
      AuthSettings__audience: ${JWT_AUDIENCE}
      Stripe__SecretKey: ${STRIPE_SECRET_KEY}
      Stripe__PublishableKey: ${STRIPE_PUBLISHABLE_KEY}
    depends_on:
      - db
    volumes:
      - images_data:/app/wwwroot/images

  react:
    container_name: freelance-react
    image: ghcr.io/andromaan/freelance-react:latest
    ports:
      - "3000:3000"
    environment:
      DISABLE_HTTPS: "true"
      VITE_API_URL: ${VITE_API_URL}
```

```
      VITE_GOOGLE_CLIENT_ID: ${VITE_GOOGLE_CLIENT_ID}
      VITE_STRIPE_PUBLISHABLE_KEY: ${VITE_STRIPE_PUBLISHABLE_KEY}
    depends_on:
      - api

volumes:
  postgres_data:
  images_data:
```

Апаратною базою для розробки та розгортання проекту виступає моя локальна комп'ютерна техніка. Роль повноцінного виділеного сервера (Host Environment) для деплою API-частини моєї платформи відіграє ноутбук, на якому встановлено стабільний і легкий дистрибутив Arch Linux. Ця операційна система без графічного інтерфейсу споживає мінімум системних ресурсів, віддаючи всю доступну оперативну пам'ять та потужність процесора під потреби Docker-контейнерів бази даних та самого додатку. Для віддаленого адміністрування цього сервера я використовую професійний інструментарій MobaXterm з моєї основної робочої машини, який забезпечує зручний доступ до терміналу Arch Linux через захищений протокол SSH та дозволяє зручно керувати конфігураційними файлами.

Оскільки мій локальний сервер фізично знаходиться за NAT-маршрутизатором, для публічного доступу до API (за адресою api.freelance-marketplace.pp.ua) та для успішного отримання вебхуків від зовнішньої платіжної системи Stripe, я впровадив технологію безпечного тунелювання. Для цього я налаштував службу Cloudflared на своєму Arch Linux сервері. Цей інструмент створює зашифроване з'єднання між моїм локальним Docker-контейнером та глобальною мережею Cloudflare, не вимагаючи при цьому відкриття жодних вхідних портів на моєму домашньому роутері. Це інфраструктурне рішення не лише робить моє API доступним з будь-якої точки світу по протоколу HTTPS, але й забезпечує потужний захист від DDoS-атак та автоматичне управління SSL-сертифікатами на рівні глобальної CDN.

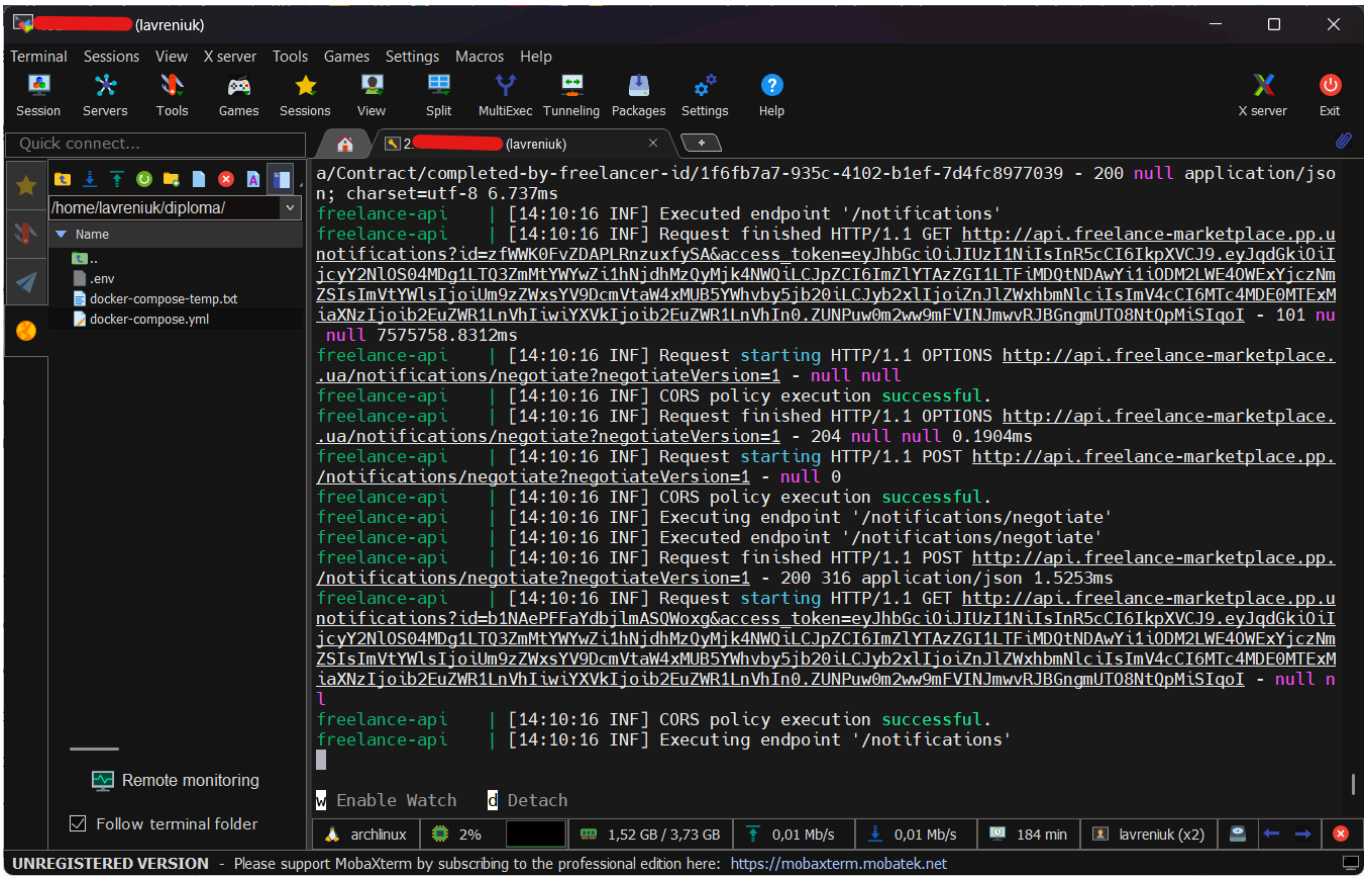


Рис 3.1. Сеанс віддаленої роботи з сервером у консолі MobaXterm.

Джерело: створено автором

3.2. Вимоги до технічного та програмного забезпечення

3.2.1. Архітектура взаємодії відповідно до моделі OSI

Архітектура мережевої взаємодії компонентів мого програмного продукту була спроектована з глибоким спиранням на багаторівневу еталонну модель базової еталонної моделі взаємодії відкритих систем (OSI), що гарантує сумісність, безпеку та надійність передачі даних. На найвищому, прикладному рівні (Application Layer - рівень 7), мій додаток інтенсивно використовує протоколи HTTP та його захищену версію HTTPS для здійснення всіх синхронних викликів до REST API. Саме на цьому рівні відбувається передача JWT-токенів у заголовках авторизації (Authorization Bearer) для перевірки прав доступу до захищених ендпоінтів моєї системи. Окрім традиційного HTTP-зв'язку, прикладний рівень моєї архітектури включає використання протоколу WebSocket (відповідно до стандарту RFC 6455).

Цей протокол є критично важливим для роботи мого хабу NotificationHub, побудованого на базі SignalR. На відміну від HTTP, де ініціатором завжди виступає клієнт, WebSocket встановлює постійне повнодуплексне з'єднання, що дозволяє моєму серверу миттєво «проштовхувати» (push) сповіщення про нові повідомлення або зміни статусів контрактів безпосередньо в браузер користувача. На рівні представлення даних (Presentation Layer - рівень 6) архітектура використовує формат JSON для серіалізації та десеріалізації об'єктів під час обміну даними між клієнтом та сервером, що забезпечує легковаговість мережевих пакетів.

Спускаючись до транспортного рівня (Transport Layer - рівень 4), архітектура моєї системи повністю покладається на стек протоколів TCP (Transmission Control Protocol). Цей протокол встановлює сеанс зв'язку та гарантує доставку пакетів даних у правильній послідовності без втрат. Гарантована доставка є абсолютно критичною вимогою для фінансового ядра моєї платформи, оскільки втрата пакетів під час отримання асинхронних вебхуків від платіжної системи Stripe могла б призвести до фатальних помилок у статусах контрактів або втрати інформації про успішний переказ коштів. На мережевому рівні (Network Layer - рівень 3) взаємодія забезпечується протоколом IP. Однак, з огляду на те, що мій фізичний сервер знаходиться в локальній мережі за NAT-маршрутизатором і не має статичної білої IP-адреси, я інтегрував технологію Cloudflared. Цей інструмент працює як захищений тунель, який інкапсулює мережевий трафік рівня 3 і 4, створюючи надійний міст між глобальною граничною мережею (Edge Network) Cloudflare та моїм локальним сервером. Це дозволяє здійснювати маршрутизацію запитів із зовнішньої мережі безпосередньо до мого локального API, абсолютно усуваючи необхідність налаштування вразливої переадресації портів (port forwarding) на домашньому роутері.

3.2.2. Вимоги до серверного програмного забезпечення (Host Environment)

Вимоги до системного програмного забезпечення фізичного сервера, який виступає хостом для мого бекенду, були сформовані з урахуванням концепції максимальної ізоляції процесів, безпеки та максимально ефективного використання

апаратних ресурсів. В якості базової операційної системи я обрав дистрибутив Arch Linux. Цей вибір є стратегічним, оскільки Arch Linux базується на моделі плаваючих релізів (rolling release) і поширюється без жодних встановлених за замовчуванням графічних середовищ чи непотрібних фонових служб. Завдяки цьому операційна система має мінімальний цифровий слід (footprint), що дозволяє мені виділити майже весь доступний обсяг оперативної пам'яті та потужності процесора виключно під обслуговування бази даних та логіки мого додатку. Крім того, мінімальна кількість встановлених пакетів суттєво зменшує площу потенційної атаки (attack surface) на мій сервер.

Головною та найбільш критичною вимогою до системного програмного забезпечення на моєму хості є наявність встановленого рушія Docker Engine та інструменту оркестрації Docker Compose. Оскільки я повністю імплементував підхід контейнеризації, мені не потрібно інсталювати середовище виконання .NET 8 SDK/Runtime або сервер PostgreSQL безпосередньо в базову операційну систему Arch Linux. Docker Engine бере на себе всю відповідальність за управління життєвим циклом контейнерів, використовуючи механізми ядра Linux (cgroups та namespaces) для жорсткого обмеження споживання ресурсів та ізоляції процесів. Це означає, що мій контейнер freelance-db, всередині якого розгорнуто ядро бази даних PostgreSQL 16, та контейнер freelance-api, який містить скомпільований код мого ASP.NET Core сервера, працюють у повністю незалежних середовищах. Таке архітектурне рішення робить мій серверний простір імунізованим до конфліктів версій системних бібліотек, гарантує абсолютну відтворюваність середовища на будь-якому іншому сервері та значно спрощує процес резервного копіювання даних через монтування ізольованих файлових томів (Docker Volumes).

3.2.3. Вимоги до клієнтського програмного забезпечення

Оскільки клієнтська частина мого фріланс-маркетплейсу реалізована у вигляді складного сучасного односторінкового веб-додатка (SPA) на базі екосистеми React, вимоги до клієнтського програмного забезпечення користувачів є мінімальними, але вони мають чіткі технологічні рамки. Для повноцінної роботи з моєю платформою

замовникам та спеціалістам категорії фріланс не потрібно завантажувати чи інсталювати жодних додаткових десктопних програм або плагінів. Єдиною вимогою є наявність актуальної версії сучасного веб-браузера (наприклад, Google Chrome, Mozilla Firefox, Microsoft Edge або Apple Safari), який побудований на базі високопродуктивних рушіїв обробки JavaScript, таких як V8 або WebKit. Браузер обов'язково повинен підтримувати сучасний синтаксис ECMAScript (ES6+), оскільки вся бізнес-логіка маршрутизації, обробки форм через React Hook Form, управління глобальним станом через Redux Toolkit та динамічний рендеринг компонентів відбувається виключно на стороні клієнта за допомогою JavaScript.

Додатковою суворою вимогою до клієнтського програмного забезпечення є підтримка безпечних контекстів (Secure Contexts) та технології HTML5 WebSockets. Оскільки моя платформа інтегрована з платіжним шлюзом Stripe, офіційні клієнтські скрипти Stripe.js (які відповідають за безпечну токенізацію даних банківських карток) вимагають виконання виключно по захищеному протоколу HTTPS. Аналогічно, встановлення постійного з'єднання з моїм хабом NotificationHub для отримання сповіщень у реальному часі відбувається через захищений протокол wss:// (WebSocket Secure). У разі використання користувачами застарілих версій браузерів, які не підтримують ці сучасні криптографічні стандарти (TLS 1.2 або вище), або якщо користувач перебуває за корпоративним брандмауером, що агресивно блокує веб-сокет з'єднання, мій додаток зможе функціонувати лише в режимі базового HTTP-опитування, що значно погіршить інтерактивний досвід користування платформою. Завдяки використанню Tailwind CSS, мій інтерфейс є повністю адаптивним, що дозволяє користувачам працювати з платформою як з настільних комп'ютерів, так і з мобільних пристроїв без втрати функціональності.

3.2.4. Топологія мережі та деплоймент

Топологія розгортання моєї інформаційної системи детально визначає шляхи проходження мережевого трафіку в глобальній мережі інтернет, а також внутрішню структуру ізольованої взаємодії контейнерів безпосередньо на моєму локальному сервері. У глобальній мережі мій розроблений бекенд публічно доступний за

доменним ім'ям api.freelance-marketplace.pp.ua. Маршрутизація запиту починається з моменту, коли клієнтський додаток (наприклад, для виконання запиту авторизації) звертається до цього домену. Спочатку запит потрапляє на найближчий до користувача периферійний сервер глобальної мережі доставки контенту (CDN) Cloudflare. Саме там відбувається термінація SSL-сертифікатів, захист від DDoS-атак та фільтрація підозрілого трафіку на рівні Web Application Firewall (WAF). Після успішної перевірки на стороні Cloudflare, трафік перенаправляється через захищений інкапсульований тунель Cloudflared, який постійно підтримує активне з'єднання між серверами Cloudflare та демоном, що працює на моєму Arch Linux сервері. Це дозволяє мені отримувати зовнішні запити навіть попри те, що мій сервер не має виділеної публічної IP-адреси.

Внутрішня топологія розгортання на самому сервері суворо регламентується декларативним інфраструктурним файлом docker-compose.yml. Під час запуску цього файлу Docker Engine створює віртуальну ізольовану мережу типу bridge. У цій мережі розгортаються два ключові контейнери: база даних та API. Контейнер мого API (freelance-api) налаштований таким чином, щоб приймати перенаправлені HTTP-запити від локального тунелю Cloudflared на внутрішній порт 8080. З іншого боку, контейнер бази даних PostgreSQL (freelance-db) працює на стандартному порті 5432, проте в моїй топології цей порт принципово не експонується (не прокидається) в локальну операційну систему хоста. Доступ до бази даних можливий виключно зсередини віртуальної мережі Docker, і звернутися до неї може лише контейнер мого API за допомогою внутрішнього DNS-імені. Така закрита дворівнева топологія гарантує найвищий можливий рівень безпеки: база даних повністю схована від зовнішнього світу та будь-яких спроб прямого підключення, а єдиною точкою входу в систему залишається мій ретельно спроектований та захищений API-інтерфейс. Всі чутливі дані, такі як паролі до бази даних, секретні ключі JWT або токени Stripe, не «зашиті» в код, а передаються всередину контейнерів виключно через зовнішній файл змінних оточення .env під час розгортання.

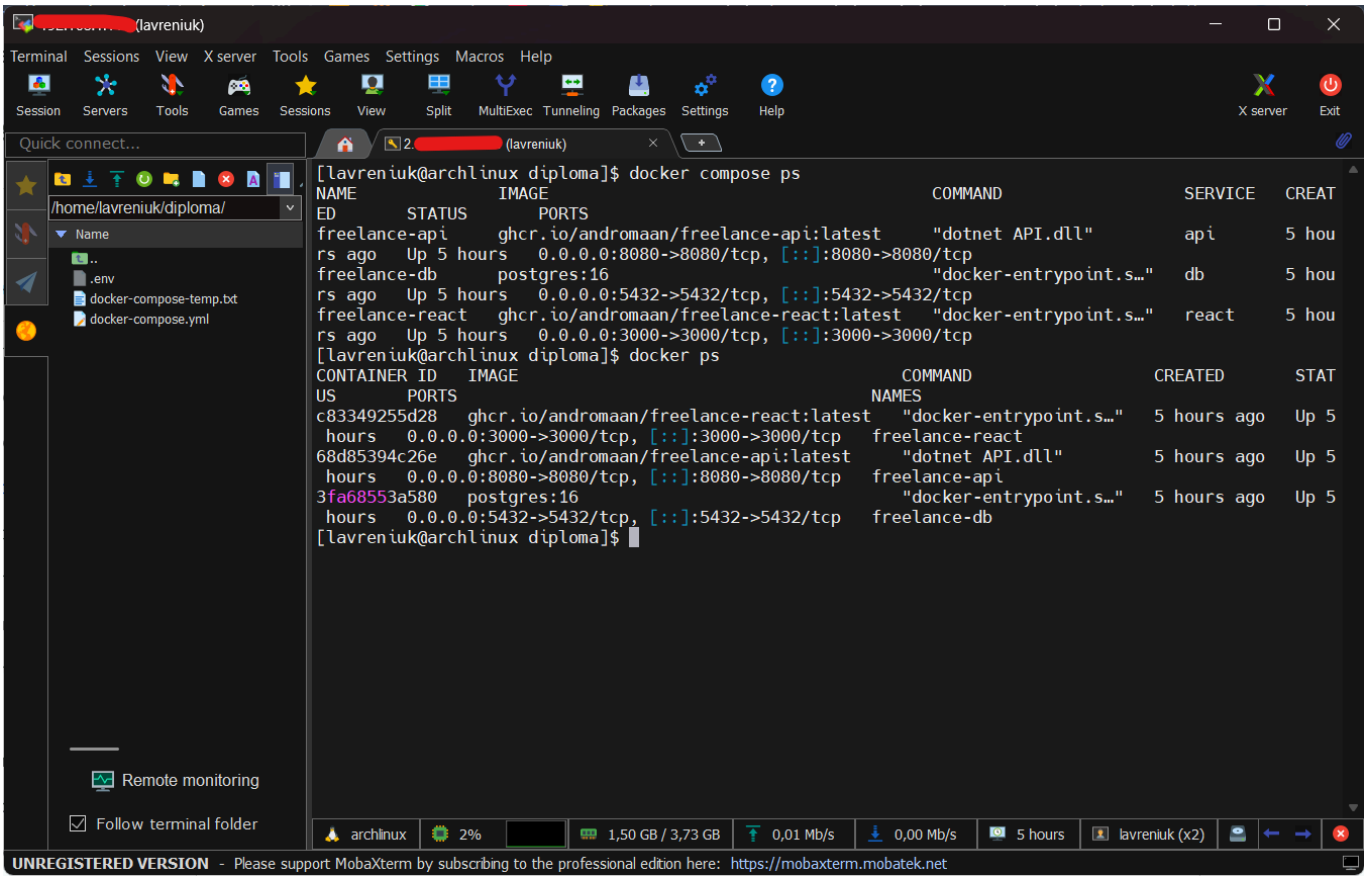


Рис. 3.2. Вивід команди docker ps і docker compose ps на сервері (підтвердження роботи контейнерів).

Джерело: створено автором

3.3. Опис програмної реалізації

3.3.1. Архітектура програмного коду бекенду (Clean Architecture)

Програмна реалізація моєї серверної частини дотримується принципів чистої архітектури (Clean Architecture), що дозволило мені розділити відповідальність між компонентами та забезпечити високу тестованість коду. Архітектура проекту фізично поділена на чотири основні шари. Фундаментальним рівнем є Domain Layer, який містить чисті доменні моделі сутностей, такі як User, Project, Contract, а також усі необхідні перелічення (Enums) і базові абстракції. Цей шар не має жодних зовнішніх залежностей від інших проектів чи фреймворків, що гарантує незалежність моєї бізнес-логіки від інфраструктурних рішень.

Наступним іде рівень доступу до даних, або Data Access Layer (DAL). Тут я реалізував взаємодію з базою даних через клас AppDbContext за допомогою Entity Framework Core. У цьому ж шарі знаходяться файли міграцій для еволюції схеми бази даних та імплементація патерну Generic Repositories, що дозволило мені уніфікувати виконання типових CRUD-операцій і уникнути дублювання коду для кожної окремої сутності.

Основна логіка платформи зосереджена в Business Logic Layer (BLL). Саме тут я реалізував патерн CQRS за допомогою бібліотеки MediatR, розділивши операції на команди (зміна стану) та запити (отримання даних). Цей шар також містить усі сервіси, профілі маппінгу для AutoMapper, класи хабів SignalR та механізми автоматичної валідації через FluentValidation, які спрацьовують у конвеєрі MediatR перед виконанням будь-якої команди.

Верхнім рівнем виступає API Layer, який відповідає за прийом HTTP-запитів. Він містить структуру контролерів, конфігурацію контейнера залежностей (через Scrutor), налаштування документації Swagger / OpenAPI для зручного тестування ендпоінтів та глобальний перехоплювач помилок MiddlewareExceptionsHandling, який автоматично трансформує винятки у відповідні HTTP-статуси (наприклад, ValidationException у 400 Bad Request).

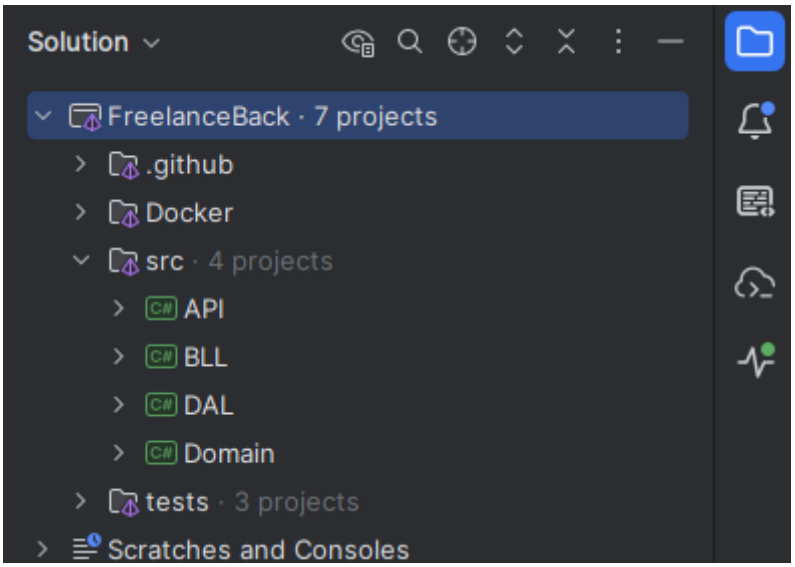


Рис. 3.3. Структура проєкту серверної частини за принципами чистої архітектури (Clean Architecture).

Джерело: створено автором

3.3.2. Програмна реалізація основної бізнес-логіки маркетплейсу

Процес управління ідентичністю та ролями у моєму маркетплейсі починається з базової таблиці User, де зберігаються хешований пароль, електронна пошта, країна, аватар та інформація про володіння мовами. Системою передбачено чотири базові ролі в таблиці Role: admin, moderator, employer та freelancer. Реєстрація та логін здійснюються через AccountController за допомогою класичного JWT-токена або через безпечну інтеграцію з Google OAuth 2.0. Контролер UserController дозволяє авторизованим користувачам отримувати власні дані та оновлювати профіль.

Управління профілями реалізовано через динамічне створення додаткових записів. Якщо під час реєстрації користувач обирає певну роль, у базі автоматично створюється відповідний запис у таблицях Freelancer або Employer для зберігання специфічних деталей. Для фрілансерів я розробив контролери Portfolio та Skills, які дозволяють заповнювати профіль інформацією про минулі проекти (навіть поза межами платформи) та зазначати професійні навички.

Життєвий цикл проекту ініціюється роботодавцем, який створює запис у таблиці Project зі статусом «відкрито», призначаючи відповідні категорії з таблиці Categories. Для більшої деталізації технічного завдання та майбутньої оплати роботодавець може додати проміжні етапи за допомогою Project Milestone. Щоб забезпечити швидку роботу інтерфейсу, до кожного GET-запиту в ProjectController я додав дефолтну пагінацію та розширені фільтри пошуку.

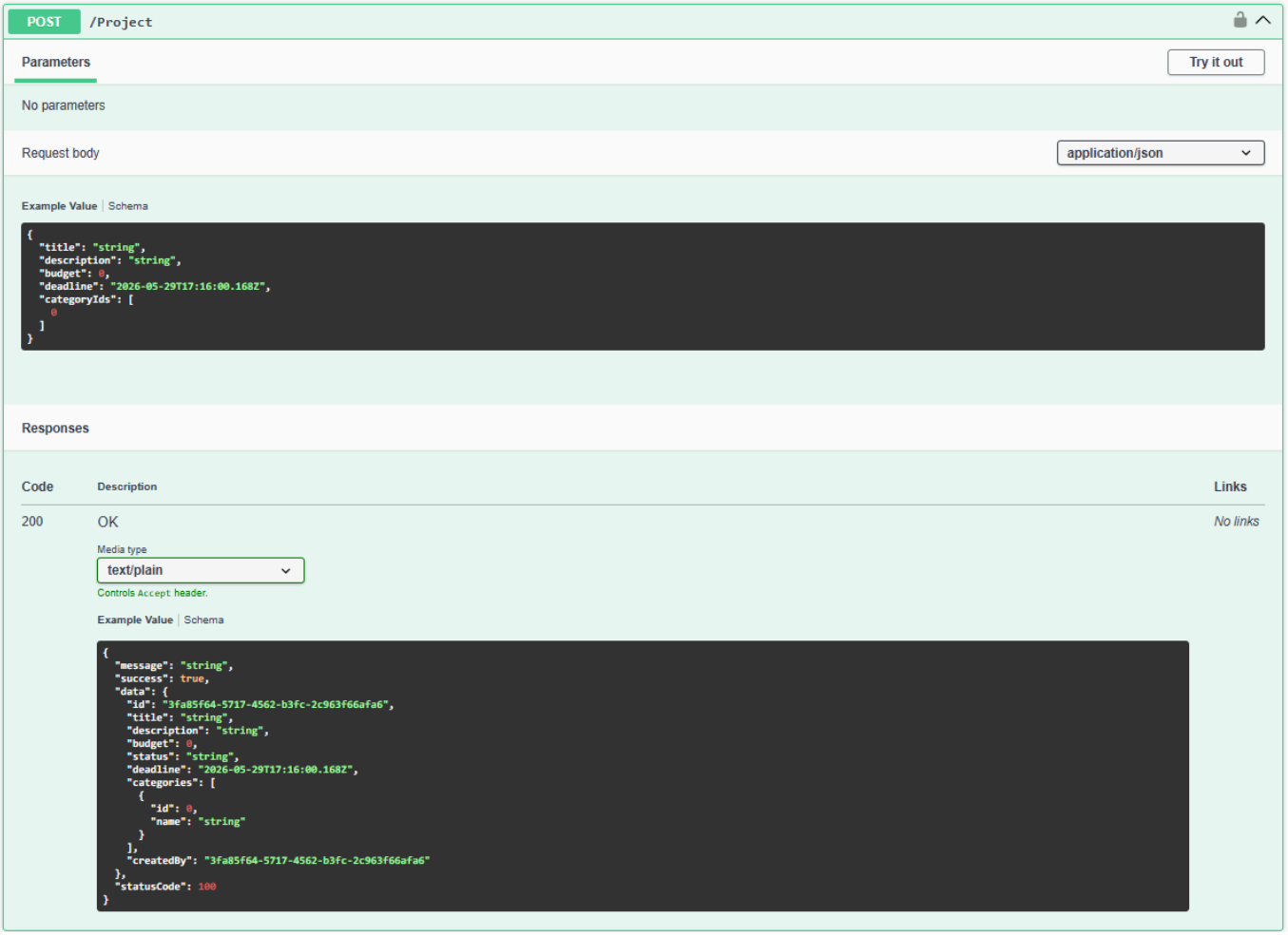


Рис. 3.4. Документація кінцевої точки POST /Project у середовищі Swagger з схемами запиту/відповіді.

Джерело: створено автором

Система пропозицій (Bidding) дозволяє фрілансерам, які знайшли підходящий проект, відправити свою первинну заявку (Bid). Після цього роботодавець може ініціювати спілкування через MessageController (на цьому етапі повідомлення працюють без прив'язки до контракту). У процесі обговорення фрілансер формує детальну пропозицію — Quote, яка містить конкретні умови співпраці.

Управління контрактами базується на логіці трансформації Quote у Contract. Якщо роботодавця влаштовує пропозиція, він через неї генерує контракт. Контракт успадковує прив'язку до проекту, а його проміжні етапи (Contract Milestone) формуються на основі відповідних етапів проекту. Сума з Quote фіксується як agreed rate. Робота над проектом відбувається через зміну статусів Contract Milestone:

фрілансер переводить етап у стан «в роботі» або «виконано», після чого роботодавець перевіряє роботу і змінює статус на «переглядається» або затверджує її, ініціюючи виплату.

Фінансові транзакції у моєму додатку безпечно обробляються через програмну інтеграцію зі Stripe для депозитів, де також враховується поле валюти. Залежно від типу транзакції, система робить відповідні записи в таблицях ContractPayment, Payment та WalletTransaction, а фактичний баланс відображається в таблиці User wallet.

Система арбітражу та відгуків завершує життєвий цикл співпраці. Після успішного контракту обидві сторони можуть залишити відгуки через ReviewController. Проте, якщо виникають проблеми (наприклад, пропущений дедлайн), будь-яка сторона може відкрити суперечку через DisputeController. Для роботи модераторів я розробив DisputeResolutionController, який дозволяє вивчати докази, змінювати статуси етапів контракту (вирішуючи долю зарезервованих коштів) та фіксувати остаточне рішення.

3.3.3. Реалізація механізмів реального часу

Для миттєвого інформування користувачів я налаштував NotificationHub на базі технології SignalR. Це push-only хаб, який надсилає подію ReceiveNotification безпосередньо на клієнтський додаток через IHubContext. Ідентифікація правильного отримувача реалізована мною у класі NotificationUserIdProvider, який витягує унікальний Guid користувача з JWT claims. Механізм передбачає різні типи сповіщень залежно від ролі: роботодавець миттєво отримує сигнали про NewBidReceived, NewMessage чи ReviewLeft, тоді як фрілансер сповіщається про MilestoneApproved, ContractCreated або PaymentReceived, що робить взаємодію з платформою надзвичайно живою та інтерактивною.

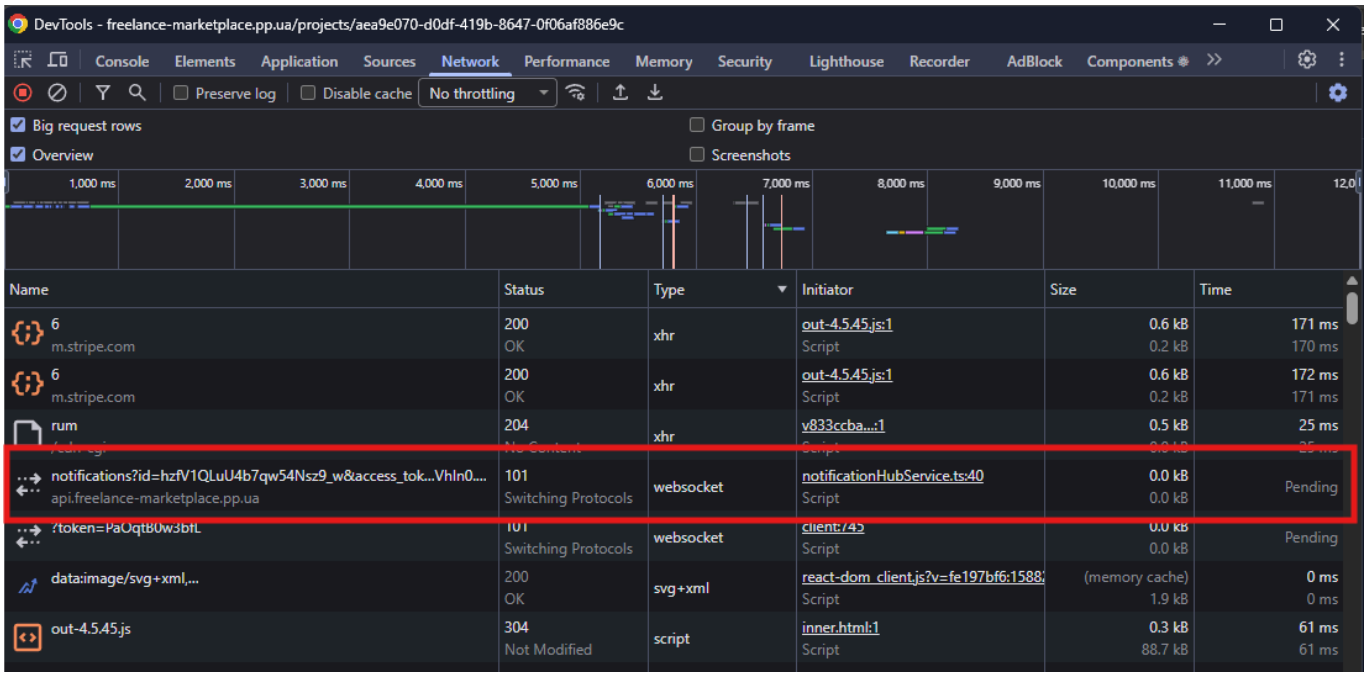


Рис. 3.5. Перевірка WebSocket-з'єднання з хабом сповіщень у панелі Network.

Джерело: створено автором

3.3.4. Програмна реалізація клієнтського додатку (React)

Структура мого клієнтського проекту побудована з використанням бібліотеки React, середовища Vite та мови TypeScript. Архітектура фронтенду базується на компонентному підході, де кожен логічний блок інтерфейсу (наприклад, картка проекту або форма пропозиції) винесений в окремий компонент. Сторінки додатку збираються з цих компонентів, а навігація між ними налаштована за допомогою бібліотеки маршрутизації (React Router), що забезпечує плавний перехід між профілем, списком проектів та активними контрактами без перезавантаження сторінки.

Інтеграція з бекендом повністю реалізована мною через інструментарій RTK Query. Замість ручного написання функцій для fetch-запитів, я сконфігурував централізоване API, яке автоматично генерує React-хуки для отримання даних (queries) та їх зміни (mutations). Ці хуки інкапсулюють у собі вхідні параметри, обробку відповідей, стани завантаження та кешування. Наприклад, хук створення контракту передає необхідні дані на сервер і, у разі успішної відповіді, автоматично

інвалідує кеш списку активних контрактів, завдяки чому інтерфейс користувача оновлюється миттєво і завжди відображає найактуальнішу інформацію з бази даних.

3.4. Опис програмної реалізації клієнтського додатку та інтерфейсу користувача

3.4.1. Архітектура та структура проекту

Фізична організація кодової бази фронтенду відображає функціональне розділення відповідальності. Коренева директорія src/ містить наступні логічні шари:

- services/ — централізований шар взаємодії з бекендом. Тут розташовані базовий API-слайс RTK Query (baseApi.ts або api.ts), а також доменні сервіси (ендпоінти для проєктів, контрактів, повідомлень, авторизації). Ця папка інкапсулює всю логіку HTTP-запитів, тегів кешу та інтерсепторів;
- store/ — конфігурація Redux-store (store.ts), об'єднання редюсерів, налаштування middleware RTK Query;
- routes/ — декларативна конфігурація маршрутизації React Router. Тут визначено всі публічні та приватні маршрути, логіка захисту ресурсів та lazy-імпорти сторінок;
- pages/ — сторінки-екрани, що відповідають окремим маршрутам (наприклад, LoginPage.tsx, ProjectListPage.tsx, ContractDetailsPage.tsx);
- components/ — загальні презентаційні та розумні компоненти, що використовуються на різних сторінках (наприклад, Layout, Header, ProjectCard, MilestoneTimeline);
- hooks/ — кастомні React-хуки, зокрема хук для інтеграції з SignalR (useSignalR.ts) та допоміжні хуки для роботи з формами чи авторизацією;
- types/ — глобальні TypeScript-інтерфейси та типи, що дублікують DTO серверної частини;
- utils/ — допоміжні функції (форматування дат, валідація, обробка помилок);

- constants/ — статичні константи додатку (маршрути, статуси етапів, ролі користувачів);
- context/ — React Context для глобальних даних, що не потребують Redux (наприклад, AuthContext для синхронного доступу до даних поточного користувача у глибоко вкладених компонентах);
- assets/ — статичні ресурси (зображення, іконки);
- styles/ — глобальні CSS-файли та додаткові налаштування Tailwind.

Контейнеризація фронтенду реалізована через Dockerfile та docker-compose.yml у корені проєкту, що дозволяє зібрати production-optimized бандл та розгорнути його у складі загальної інфраструктури.

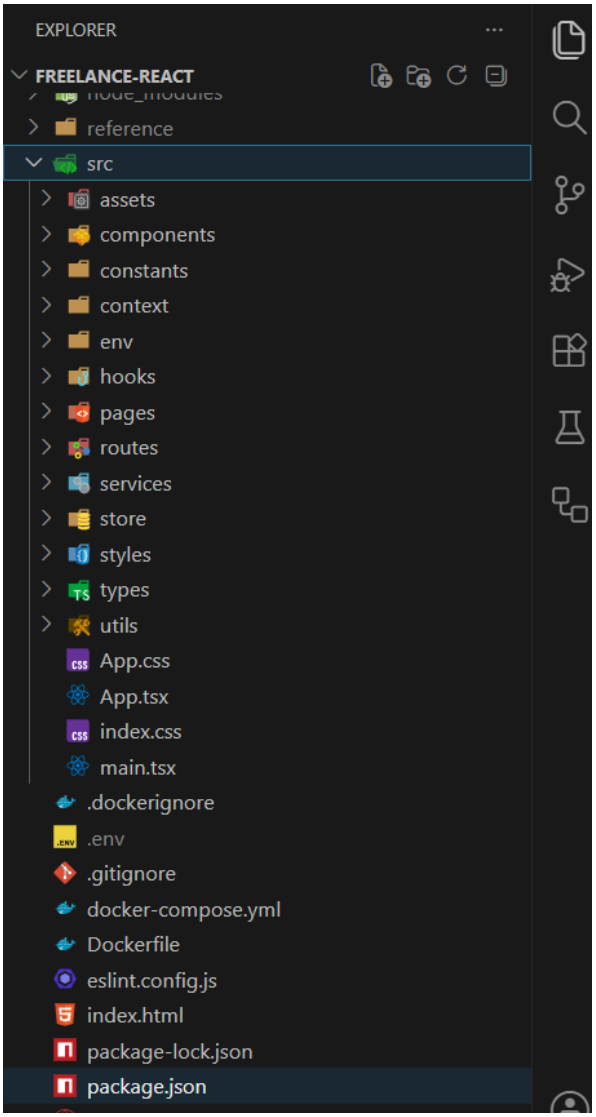


Рис. 3.6. Скріншот дерева папок проєкту у VS Code.

Джерело: створено автором

3.4.2. Опис ключових сторінок користувацького інтерфейсу

Розроблений користувацький UX/UI-інтерфейс веб-додатка повністю покриває всі етапи життєвого циклу взаємодії між замовником (роботодавцем) та виконавцем (фрілансером). Нижче наведено детальний опис функціональних сторінок платформи:

Модуль автентифікації та реєстрації (/login, /register): Забезпечує безпечний доступ до системи. Окрім стандартної форми авторизації за допомогою електронної пошти та пароля, у модуль інтегровано сервіс Google OAuth, що дозволяє користувачам створювати обліковий запис в один клік. Після первинної реєстрації додаток пропонує пройти етап онбордингу для вибору ролі в системі (Employer або Freelancer).

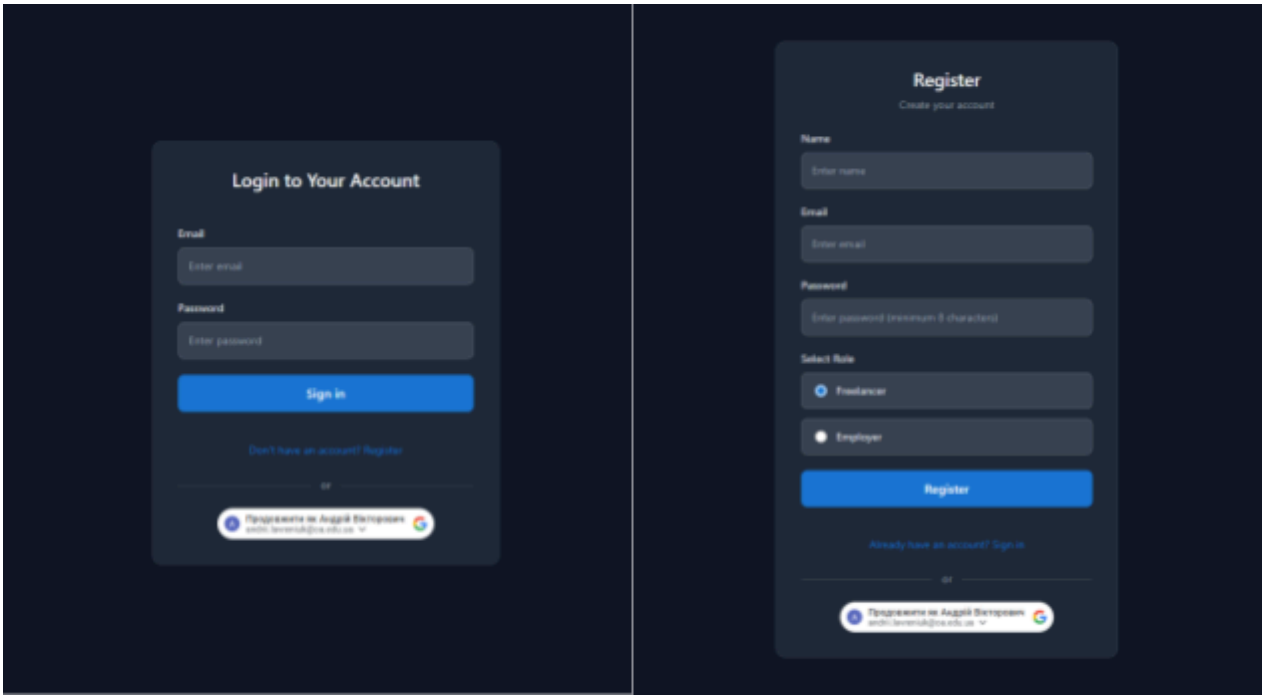


Рис. 3.7. Інтерфейс сторінки автентифікації та реєстрації користувачів з підтримкою Google OAuth.

Джерело: створено автором

Сторінки профілю користувача (/profile, /freelancers/:userId): Для фрілансера сторінка є цифровим резюме, що містить блок аватарки, портфоліо та список професійних навичок. І у /profile/edit-profile він може налаштувати свою сторінку.

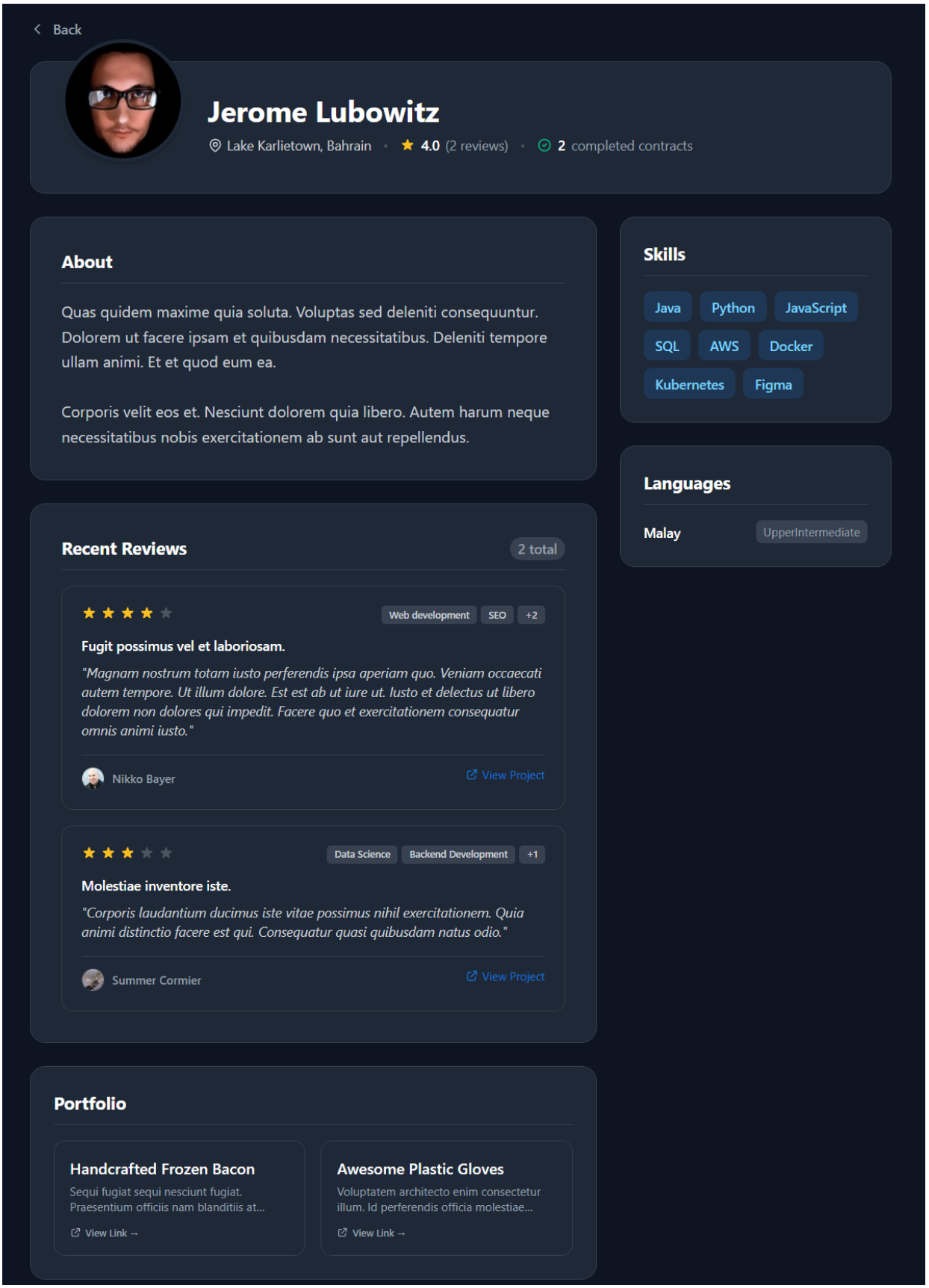


Рис. 3.8. Профіль фрілансера із заповненою інформацією про нього.

Джерело: створено автором

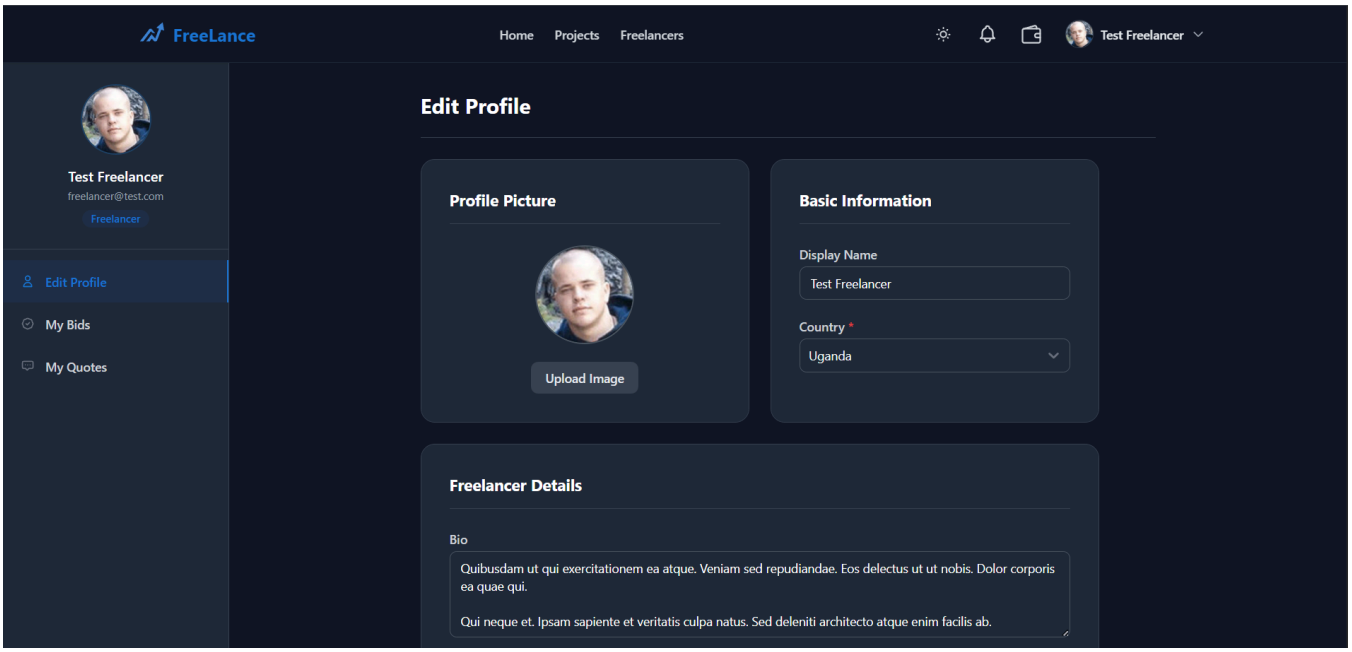


Рис. 3.9. Сторінка редагування профілю фрілансера.

Джерело: створено автором

Каталог проєктів (Marketplace) (/projects, /freelancers): Головна публічна сторінка для пошуку замовлень. В інтерфейсі реалізовано інтерактивну панель фільтрів, яка дозволяє сортувати та відсіювати проекти за категоріями, бюджетом та термінами виконання. Для запобігання перевантаженню мережевого трафіку вивід карток проектів обмежений клієнтською пагінацією. Обидві сторінки мають фільтри для зручного пошуку як фрілансерів та і проектів.

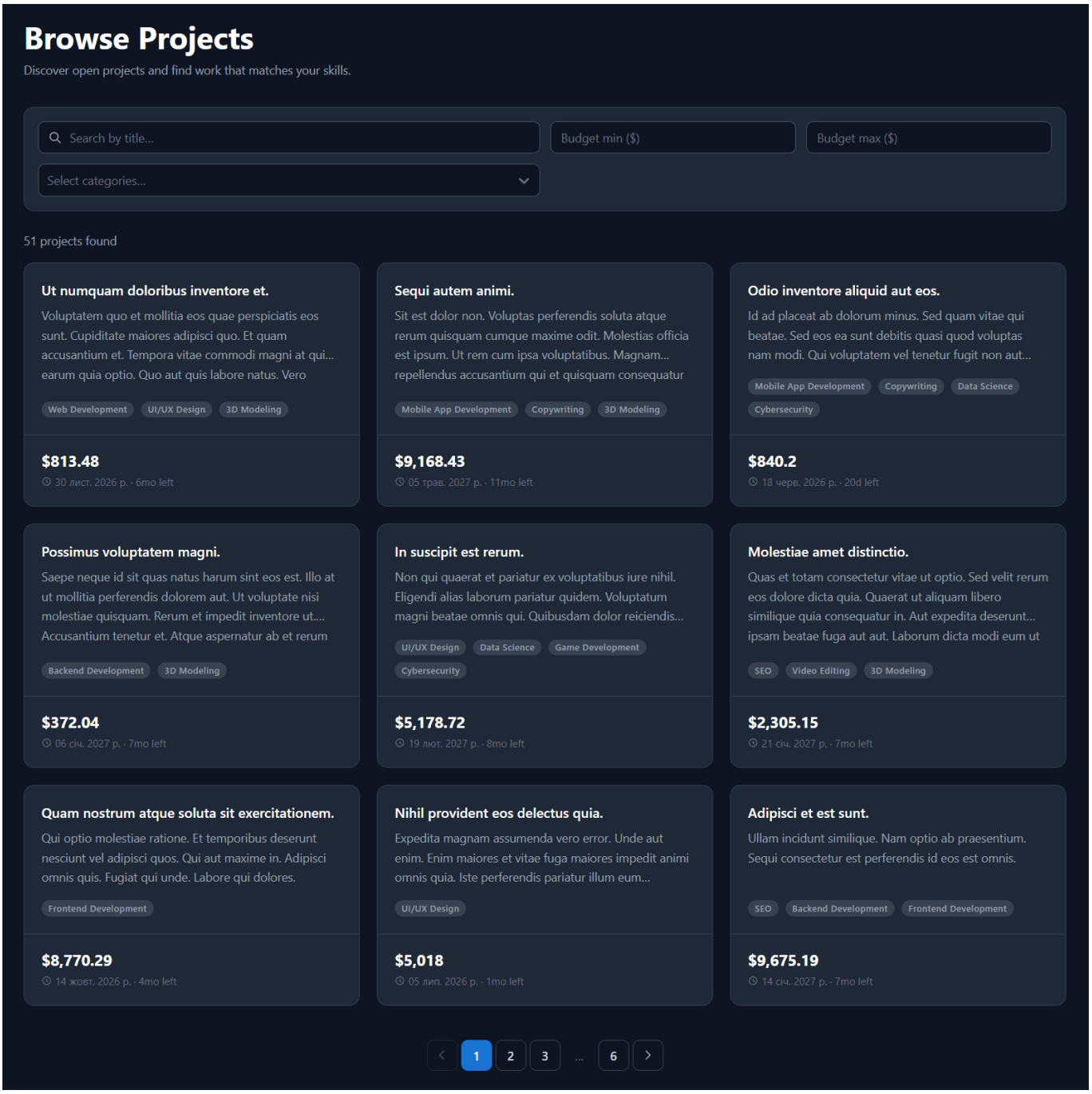


Рис . 3.10. Сторінка пошуку проєктів з інструментами фільтрації за категоріями та бюджетом з пагінацією.

Джерело: створено автором

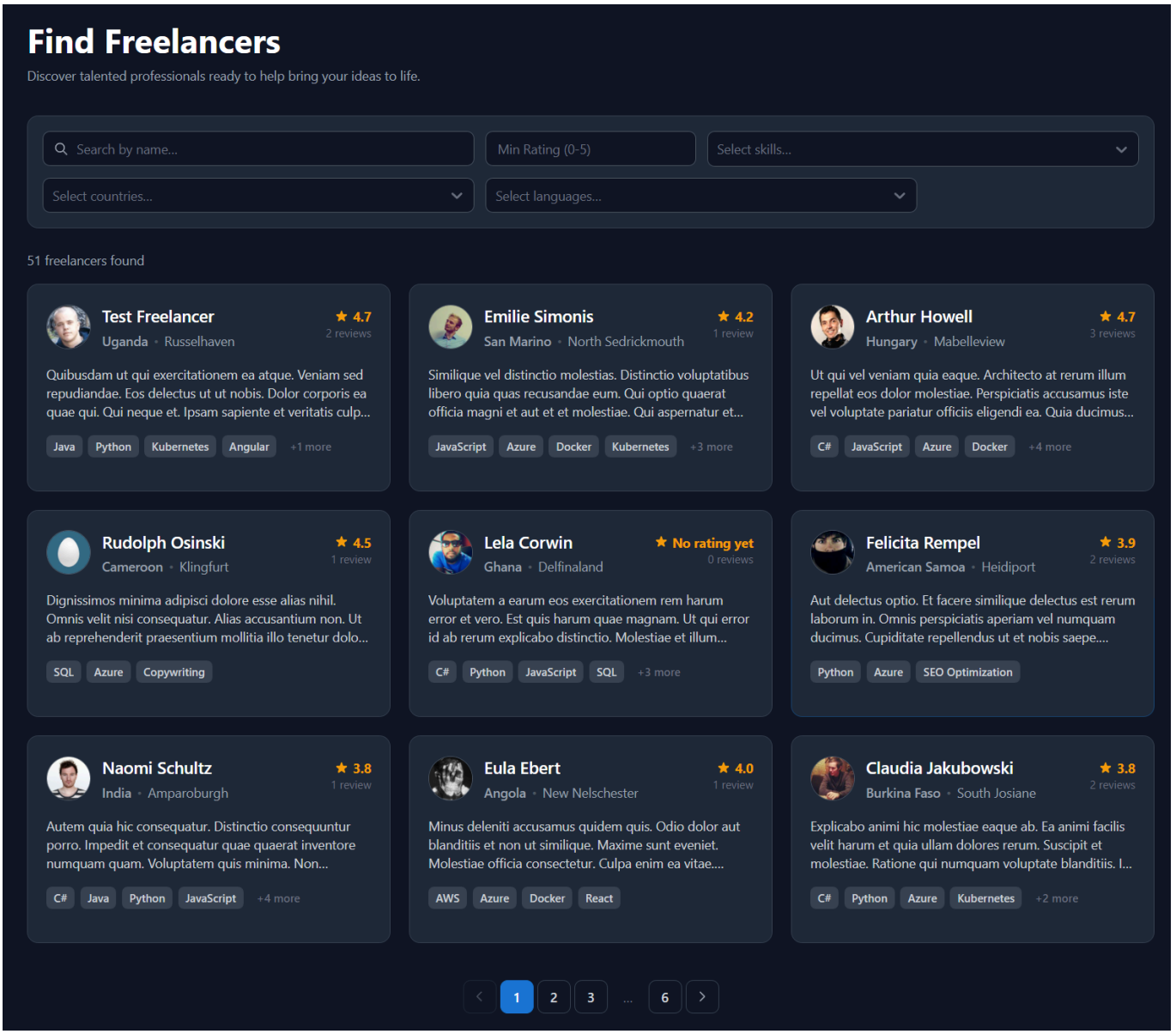


Рис . 3.11. Сторінка пошуку фрілансерів з інструментами фільтрації за вміннями, країною, мовою та рейтингом з пагінацією.

Джерело: створено автором

Детальна сторінка проєкту та система заявок (/projects/:projectId): Відображає повне технічне завдання, вимоги та опис проекту. Якщо сторінку переглядає фрілансер, йому доступна форма подачі цінової пропозиції (Bid). Якщо сторінку переглядає автор проекту (роботодавець), інтерфейс трансформується в панель моніторингу, де відображаються картки всіх поданих заявок від розробників із можливістю переходу до чату чи затвердження виконавця.

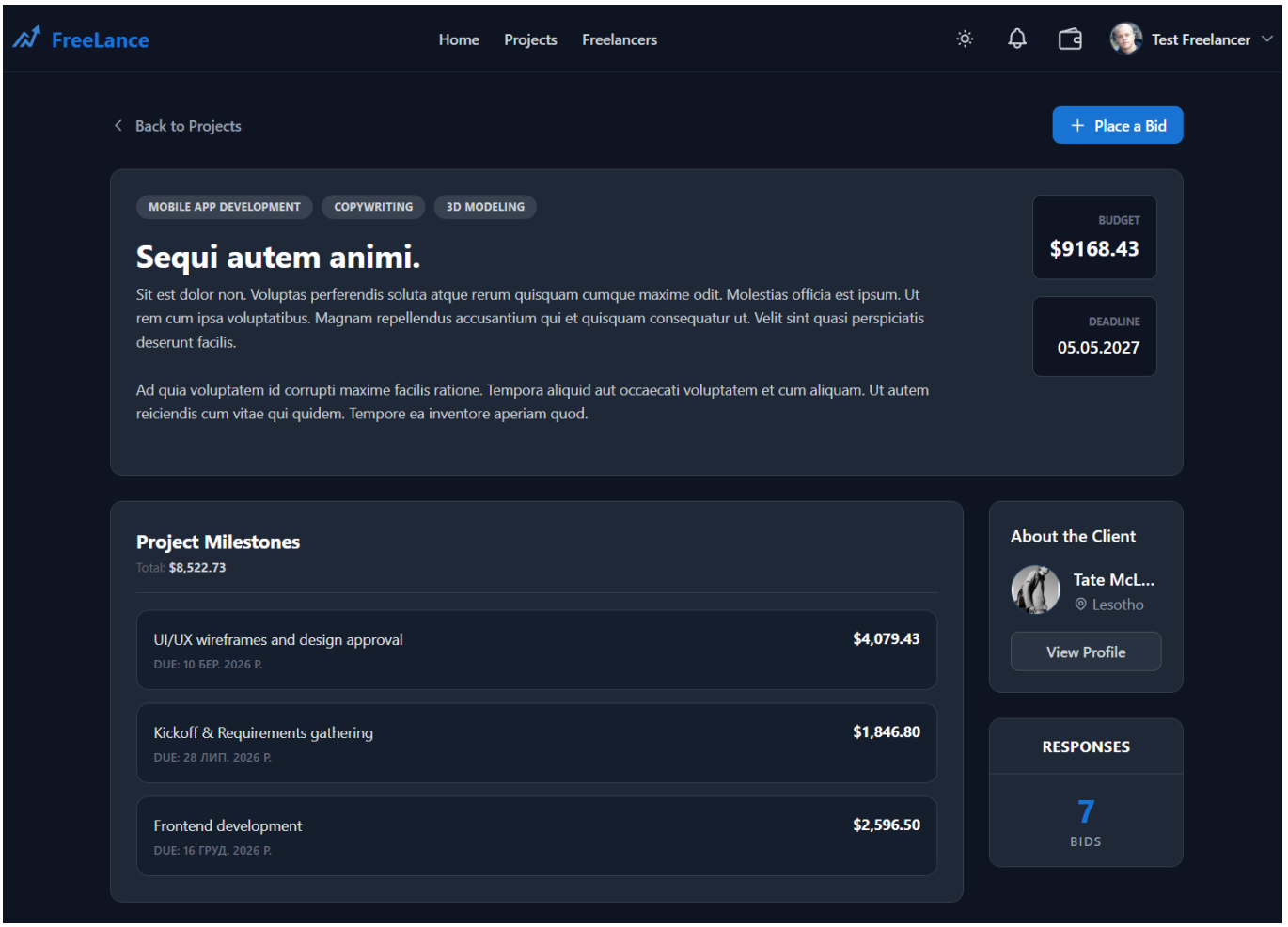


Рис . 3.12. Сторінка проекту з усією інформацією.

Джерело: створено автором

Комунікаційний модуль (сповіщення) (/notifications та контекстні діалоги): Забезпечує отримання повідомлень про деталі контракту та про інші дії. Завдяки інтеграції бібліотеки @microsoft/signalr та встановленому WebSocket-з'єднанню з сервером, повідомлення з'являються в інтерфейсі миттєво в режимі реального часу (Real-time). Для інформування про нові події поза чатом використовуються спливаючі вікна, реалізовані через react-toastify.

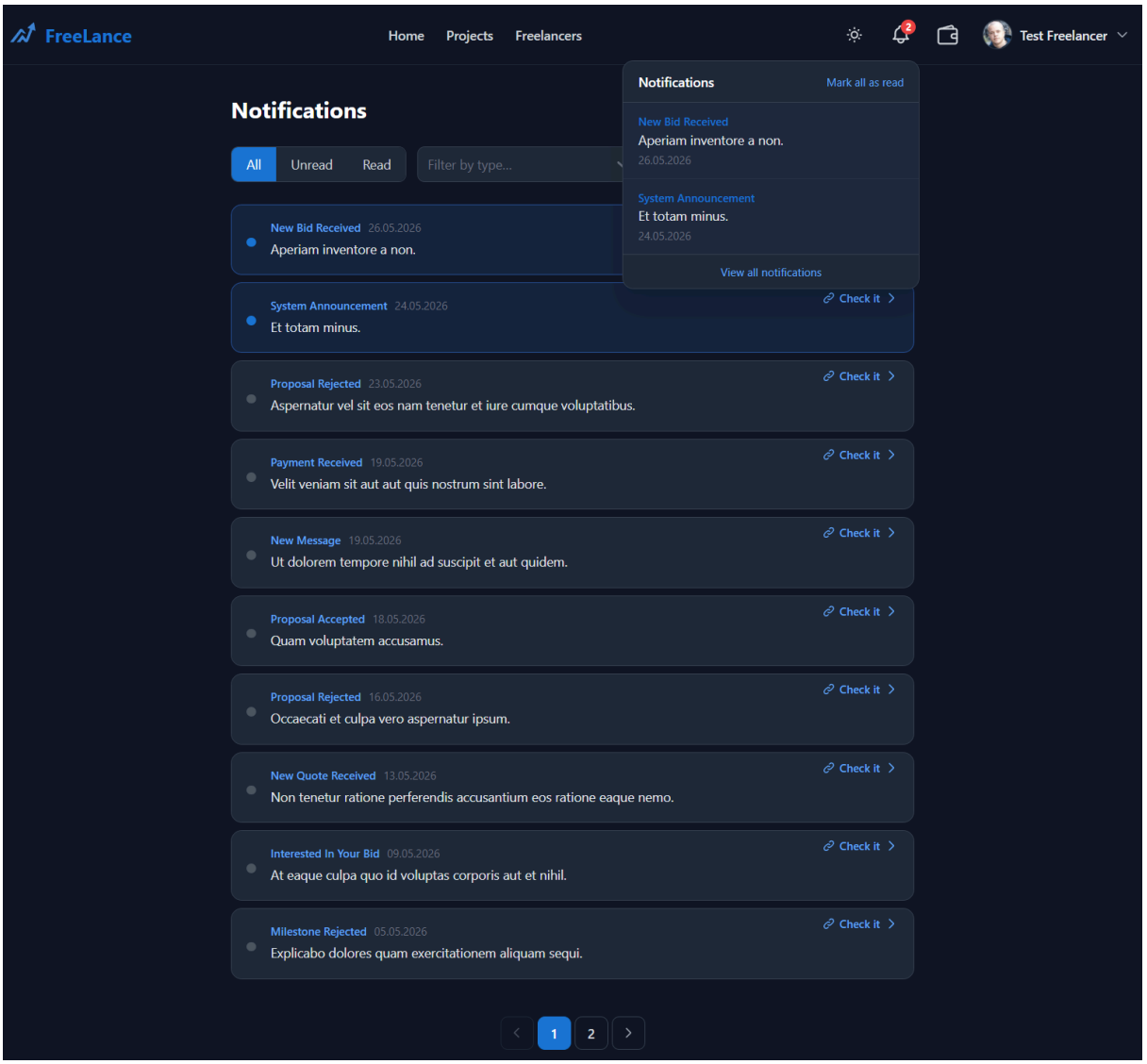


Рис . 3.13. Сторінка сповіщень з фільтрами і пагінацією та додатковим вікном доступним по всьому проекту.

Джерело: створено автором

3.5. Забезпечення надійності, CI/CD автоматизація та демонстрація системи.

3.5.1. Методологія та інструменти тестування платформи

Для гарантування найвищого рівня надійності та безперебійної роботи моєї інформаційної системи я обрав комплексну методологію автоматизованого інтеграційного тестування, відмовившись від виключного використання простих

модульних тестів (Unit Tests). Моя стратегія базується на перевірці працездатності всього ланцюжка обробки запиту: від HTTP-маршрутизації та конвеєра MediatR до збереження даних у реальній базі даних PostgreSQL. Основним інструментом для реалізації цього підходу став фреймворк xUnit у тісній інтеграції з класом WebApplicationFactory. Цей спеціалізований інструмент дозволяє мені програмно розгортати повноцінний тестовий сервер (TestHost) у пам'яті під час виконання тестів, імітуючи реальне середовище виконання ASP.NET Core. Завдяки цьому я маю можливість відправляти реальні HTTP-запити до свого API та перевіряти отримані JSON-відповіді і відповідні зміни в тестовій базі даних.

Моя база автоматизованих тестів є надзвичайно розгалуженою і налічує понад 200 детально прописаних сценаріїв, які покривають усі критичні бізнес-процеси маркетплейсу. Я приділив особливу увагу тестуванню ключових контролерів. Наприклад, тести для AccountController перевіряють не лише успішну реєстрацію чи генерацію JWT-токена, але й коректну відмову системи при спробі створення дубліката електронної пошти. Тестові сценарії для ProjectController валідують алгоритми повнотекстового пошуку, фільтрації та правильність розрахунку параметрів пагінації. Найскладніші інтеграційні тести написані мною для BidController та ContractController, оскільки вони перевіряють складну логіку переходів станів: від подачі заявки фрілансером до генерації котирування (Quote) та фінального створення контракту з прив'язкою відповідних етапів фінансування (Contract Milestones). Таке глибоке покриття коду інтеграційними тестами дозволяє мені впевнено проводити рефакторинг та швидко виявляти будь-які регресійні помилки.

3.5.2. Обробка виключних ситуацій та стабільність системи

Стабільність серверної частини моєї платформи та передбачуваність її поведінки у нештатних ситуаціях забезпечується розробленою мною системою глобального перехоплення винятків. Замість того, щоб перевантажувати кожен контролер конструкціями try-catch, я реалізував централізований механізм на рівні проміжного програмного забезпечення (Middleware). Створений мною клас

MiddlewareExceptionsHandling перехоплює будь-які необроблені помилки, що виникають під час виконання HTTP-запиту, та трансформує їх у стандартизовані відповіді формату ProblemDetails. Цей підхід дозволяє клієнтському додатку на React отримувати чітку та структуровану інформацію про причину збою. Я налаштував специфічне мапування помилок: наприклад, якщо система генерує SecurityTokenException під час перевірки прав доступу, мій перехоплювач автоматично повертає HTTP-статус 426 Upgrade Required (або відповідний статус проблем з авторизацією). Усі непередбачувані технічні збої або падіння бази даних безпечно обгортаються у статус 500 Internal Server Error, приховуючи від кінцевого користувача чутливі технічні деталі інфраструктури (Stack Trace).

Окремою та надзвичайно важливою складовою стабільності моєї системи є механізм автоматичної валідації вхідних даних. Я повністю ізолював правила перевірки від бізнес-логіки, використавши бібліотеку FluentValidation. Ці правила інтегровані безпосередньо в конвеєр обробки запитів MediatR за допомогою спеціального патерну Pipeline Behavior. Коли клієнт відправляє команду, наприклад, на створення нового проекту, конвеєр автоматично знаходить відповідний валідатор і перевіряє дані. Якщо дані не відповідають вимогам (наприклад, відсутній заголовок або вказано від'ємний бюджет), система навіть не намагається звернутися до бази даних. Натомість генерується ValidationException, який мій Middleware автоматично перетворює на HTTP-статус 400 Bad Request, додаючи до відповіді детальний словник із зазначенням конкретних полів, що містять помилки. Це захищає ядро моєї платформи від «брудних» даних та значно спрощує обробку форм на клієнті.

3.5.3. Автоматизація розгортання (CI/CD)

Для оптимізації процесу розробки та гарантування того, що на мій робочий сервер потрапляє виключно перевірений та стабільний код, я спроектував та імплементував повноцінний конвеєр безперервної інтеграції та безперервної доставки (CI/CD). Цей конвеєр реалізовано за допомогою інструментарію GitHub Actions і конфігурується файлом ci-cd.yml, повний текст конфігурації якого наведено у **додатку В** . Робота алгоритму автоматизації запускається тригерами: при кожному

пуші до головної гілки репозиторію (main) або під час створення запиту на злиття (pull request). Процес інтеграції складається з кількох послідовних кроків. Спочатку віртуальна машина в хмарі GitHub завантажує мій вихідний код, налаштовує середовище .NET 8 та виконує команду відновлення залежностей (dotnet restore). Наступним кроком відбувається компіляція проекту та обов'язковий запуск усіх моїх інтеграційних тестів. Результати тестування автоматично конвертуються у файли звітів формату TRX, які публікуються в інтерфейсі GitHub для зручного візуального аналізу покриття коду та виявлення причин можливих збоїв.

Якщо етап збірки та тестування завершується без жодної помилки, мій конвеєр автоматично переходить до фази безперервної доставки. На цьому етапі відбувається авторизація в системному реєстрі контейнерів GitHub Container Registry (GHCR) за допомогою захищених секретних токенів. Після цього виконується збірка оптимізованого Docker-образу мого API на основі написаного мною Dockerfile. Створений образ маркується двома тегами: унікальним ідентифікатором комміту (SHA) для строгого версіонування та тегом latest для позначення найактуальнішої стабільної версії. Після цього образ автоматично завантажується до реєстру GHCR. Такий архітектурний підхід дозволяє мені максимально спростити процес оновлення мого локального сервера на базі Arch Linux: мені достатньо лише виконати команду завантаження нового образу та перезапустити контейнери через Docker Compose, маючи абсолютну впевненість у тому, що нова версія успішно пройшла всі етапи автоматизованого тестування.

3.5.4. Демонстрація роботи системи

Робота з розробленою платформою для кінцевого користувача є максимально прозорою та інтуїтивною, що досягається завдяки сучасному клієнтському додатку на базі React. Демонстрація типового користувацького сценарію розпочинається з етапу авторизації. Користувач заходить на веб-сайт і має можливість створити новий обліковий запис класичним способом, заповнивши форму з email та паролем, або скористатися безпечним та швидким входом через Google OAuth 2.0. Після успішного входу користувач потрапляє на сторінку деталізації профілю, де обирає

свою основну роль. Якщо це фрілансер, він заповнює розділ Portfolio, додаючи інформацію про свої попередні роботи, та формує список професійних навичок (Skills). Роботодавець, у свою чергу, переходить до панелі управління проектами, де через зручну форму публікує нове технічне завдання, вказує очікуваний бюджет та формує конкретні етапи (Milestones) для майбутньої оплати.

Після публікації проекту він миттєво стає доступним у загальному каталозі, де фрілансери можуть його знайти за допомогою фільтрів та системи пагінації. Фрілансер детально вивчає вимоги та надсилає свою заявку (Bid). Роботодавець отримує push-сповіщення в реальному часі завдяки інтеграції з SignalR, переглядає пропозицію та може ініціювати прямий діалог для уточнення деталей. Узгодивши умови, формується фінальне котирування (Quote), яке після затвердження трансформується у юридичний електронний контракт. На цьому етапі до процесу підключається платіжний шлюз Stripe. Інтерфейс безпечно перенаправляє роботодавця на форму введення реквізитів банківської картки для депозиту коштів. Після успішної транзакції робота над проектом стартує, і сторони можуть змінювати статуси виконання етапів, завершуючи співпрацю написанням взаємних відгуків або, у разі конфлікту, ініціюванням суперечки (Dispute) для модераторів.

Для адміністраторів платформи та мене як розробника існує альтернативний, технічний шлях взаємодії із системою за допомогою інтерфейсу Swagger UI. Цей інструмент, інтегрований безпосередньо в мій API шар, автоматично генерує візуальну документацію на основі коду. Відкривши відповідну сторінку, я маю перед очима повний перелік усіх доступних контролерів та їх ендпоінтів. Swagger дозволяє мені без використання React-клієнта проводити пряме тестування бекенду: я можу авторизуватися, передавши JWT-токен у заголовок Authorization, формувати JSON-тіла запитів, відправляти їх на сервер та миттєво бачити HTTP-статуси і структуру відповідей від бази даних. Це є незамінним інструментом для перевірки складних адміністративних функцій, роботи з даними пагінації та тестування коректності спрацювання моїх алгоритмів валідації.

Висновки до розділу 3

У третьому розділі моєї дипломної роботи було здійснено вичерпний опис програмної реалізації, обґрунтування вибору інструментарію та розгортання моєї платформи для фрілансу. Я детально аргументував вибір технологічного стеку, поклавши в основу серверної частини високопродуктивну платформу ASP.NET Core 8 та надійну реляційну базу даних PostgreSQL. Для клієнтського інтерфейсу я застосував сучасну бібліотеку React у поєднанні з екосистемою Redux Toolkit, інструментом RTK Query та середовищем збірки Vite. Аналіз вимог до програмного та технічного забезпечення дозволило мені чітко визначити архітектуру мережевої взаємодії відповідно до моделі OSI, де безпечна комунікація відбувається через протоколи HTTPS та WebSockets. Я успішно спроектував ізольовану топологію розгортання на базі мого локального сервера під управлінням легкої операційної системи Arch Linux. Усі ключові компоненти системи контейнеризовані за допомогою Docker, а безпечний доступ із глобальної мережі до API забезпечується захищеним тунелем Cloudflared, що виключає необхідність прямого відкриття портів.

Безпосередній опис програмної реалізації підтвердив суворе дотримання мною принципів чистої архітектури (Clean Architecture). Я імплементував архітектурний патерн CQRS за допомогою диспетчера MediatR, що дозволило чітко розмежувати процеси читання та запису даних, а також безшовно інтегрував механізм автоматичної валідації вхідних команд через FluentValidation. У ході роботи я успішно запрограмував усю критичну бізнес-логіку платформи: безпечну реєстрацію користувачів із підтримкою JWT та Google OAuth, алгоритми формування заявок і котирувань, а також складну систему укладання електронних контрактів із проміжними етапами фінансування (Milestones). Окремим досягненням стала інтеграція платіжного шлюзу Stripe для обробки фінансових депозитів та реалізація механізмів реального часу через хаб SignalR для миттєвої доставки сповіщень. У керівництві користувача я підтвердив надійність мого рішення, описавши створену базу з понад 200 інтеграційних тестів на основі фреймворку

xUnit, яка перевіряє всю систему від API до бази даних. Крім того, я налаштував автоматизований CI/CD конвеєр у GitHub Actions, який гарантує безперервне тестування та безпечну доставку перевіреного коду у вигляді Docker-образів. У підсумку, розроблений мною програмний продукт повністю відповідає поставленим завданням дослідження, демонструє високий рівень технічної зрілості та є повністю працездатною системою, готовою до реальної експлуатації.

ВИСНОВКИ

У кваліфікаційній роботі було успішно вирішено актуальне науково-прикладне завдання розробки сучасної, високопродуктивної та безпечної інформаційної системи для організації віддаленої роботи. Мета дослідження, яка була чітко сформульована у вступі і полягала у створенні повноцінного програмного продукту для забезпечення взаємодії між замовниками та спеціалістами категорії фріланс, досягнута в повному обсязі. Послідовне розв’язання поставлених завдань дослідження дозволило мені пройти повний цикл розробки програмного забезпечення: від глибокого аналізу вимог ринку до архітектурного проектування, безпосередньої програмної реалізації та автоматизованого тестування готової платформи.

Вирішуючи перше завдання дослідження, я провів комплексний аналіз предметної області та існуючих на ринку рішень. Було доведено, що сучасні глобальні платформи часто страждають від застарілої монолітної архітектури, яка ускладнює підтримку механізмів реального часу та гнучкого ціноутворення. Мій особистий внесок на цьому етапі полягав у формуванні жорстких нефункціональних вимог до розроблюваної системи, зокрема щодо обов'язкової відповідності європейським стандартам захисту персональних даних (GDPR) та міжнародному стандарту безпеки фінансових транзакцій (PCI DSS), що в подальшому визначило вектор проектування архітектури безпеки.

Найважливішим теоретичним та концептуальним результатом роботи стала розробка архітектури системи на основі принципів чистої архітектури (Clean Architecture) та методології предметно-орієнтованого проектування (DDD). Мій внесок у розв’язання завдання проектування полягає у вдалому застосуванні архітектурного патерну CQRS за допомогою бібліотеки MediatR. Це рішення забезпечило суворе розділення логіки обробки команд та запитів, що гарантує платформі високу здатність до горизонтального масштабування під великими навантаженнями. Крім того, я спроектував оптимізовану структуру реляційної бази даних PostgreSQL, застосувавши гібридний підхід: використання суворих

реляційних зв'язків для фінансових контрактів та типу даних JSONB для гнучкого зберігання динамічних масивів професійних навичок у профілях спеціалістів. Теоретичне підґрунтя також доповнено розробкою алгоритмічних моделей на базі скінченних автоматів для ідемпотентної обробки фінансових транзакцій.

Практичні результати дослідження втілилися у повністю функціонуючому веб-додатку. Мною було розроблено надійну серверну частину на базі ASP.NET Core 8 із застосуванням патерну Repository, що дозволило інкапсулювати бізнес-логіку. Я успішно реалізував безпечний платіжний модуль через інтеграцію з API Stripe для резервування коштів та систему миттєвих сповіщень на базі веб-сокетів SignalR. Клієнтська частина платформи створена з використанням React, TypeScript та RTK Query, що забезпечило надзвичайно швидкий та оптимістичний інтерфейс користувача. Якість мого коду та стабільність бізнес-логіки підтверджена написанням понад 200 автоматизованих інтеграційних тестів за допомогою фреймворку xUnit та створенням безперервного CI/CD конвеєра в GitHub Actions, який автоматично збирає Docker-образи та розгортає їх на локальному Linux-сервері через захищений тунель Cloudflared.

Здобуті результати мають високу практичну цінність, оскільки розроблена платформа є готовим до використання (production-ready) інструментом, що забезпечує повний життєвий цикл управління проектами, контрактами, суперечками та фінансами. Рекомендації щодо подальшого практичного використання моєї системи включають її пілотне впровадження для потреб малих та середніх підприємств, які шукають незалежних підрядників. З наукової та інженерної точок зору, закладена мною архітектура CQRS створює ідеальний фундамент для майбутніх розширень, зокрема для інтеграції алгоритмів штучного інтелекту з метою автоматизованого метчингу замовлень із виконавцями та впровадження мікросервісів для аналітики великих даних без порушення існуючої інфраструктури ядра платформи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

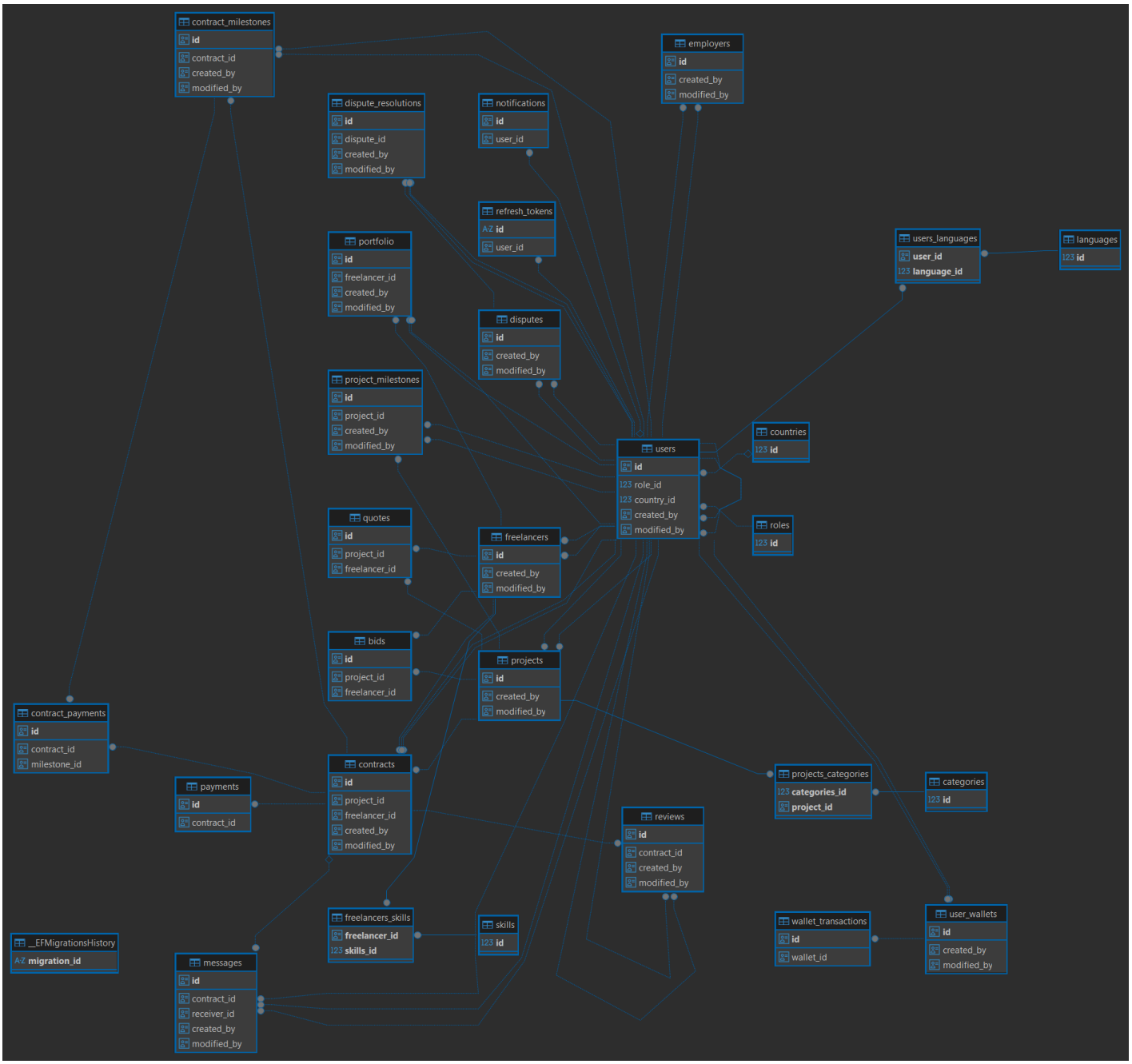
1. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003. 560 p.
2. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.
3. Fowler M. CQRS (Command Query Responsibility Segregation) [Електронний ресурс] // MartinFowler.com. – URL: https://martinfowler.com/bliki/CQRS.html
4. Офіційна документація Microsoft щодо розробки веб-додатків на базі ASP.NET Core 8 [Електронний ресурс] // Microsoft Learn. – URL: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-8.0
5. Робота з даними та ORM за допомогою Entity Framework Core 8 [Електронний ресурс] // Microsoft Learn. – URL: https://learn.microsoft.com/en-us/ef/core/
6. Офіційна документація системи управління базами даних PostgreSQL 16 [Електронний ресурс] // PostgreSQL Global Development Group. – URL: https://www.postgresql.org/docs/16/index.html
7. Інтеграція та використання бібліотеки MediatR для реалізації патерну CQRS у .NET [Електронний ресурс] // GitHub. – URL: https://github.com/jbogard/MediatR
8. Валідація об'єктів у .NET за допомогою бібліотеки FluentValidation [Електронний ресурс] // FluentValidation Documentation. – URL: https://docs.fluentvalidation.net/
9. Комунікація в реальному часі за допомогою ASP.NET Core SignalR [Електронний ресурс] // Microsoft Learn. – URL: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction
10. Офіційна документація платіжної системи Stripe API та налаштування безпечних вебхуків [Електронний ресурс] // Stripe Documentation. – URL: https://stripe.com/docs/api
11. Розробка користувацьких інтерфейсів за допомогою бібліотеки React [Електронний ресурс] // React Official Documentation. – URL: https://react.dev/
12. Управління глобальним станом та кешування запитів за допомогою Redux Toolkit та RTK Query [Електронний ресурс] // Redux Toolkit Official Docs. – URL: https://redux-toolkit.js.org/
13. Організація середовища збірки фронтенд-додатків за допомогою Vite [Електронний ресурс] // Vite Next Generation Frontend Tooling. – URL: https://vitejs.dev/
14. Інтеграційне тестування веб-додатків ASP.NET Core з використанням xUnit та WebApplicationFactory [Електронний ресурс] // Microsoft Learn. – URL: https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests

15. Автоматизація розгортання та контейнеризація за допомогою Docker та Docker Compose [Електронний ресурс] // Docker Documentation. – URL: https://docs.docker.com/
16. Міжнародний стандарт безпеки даних індустрії платіжних карток (PCI DSS) [Електронний ресурс] // PCI Security Standards Council. – URL: https://www.pcisecuritystandards.org/
17. Загальний регламент Європейського Союзу про захист персональних даних (GDPR) [Електронний ресурс] // General Data Protection Regulation. – URL: https://gdpr-info.eu/
18. Clean Architecture with .NET 8 | Complete Guide [Відеозапис] / Milan Jovanović // YouTube. – URL: https://www.youtube.com/watch?v=yF9Swrq0KdE
19. Implementing CQRS and MediatR in ASP.NET Core [Відеозапис] / Nick Chapsas // YouTube. – URL: https://www.youtube.com/watch?v=vB22KylKudI
20. React Redux Toolkit Query Tutorial [Відеозапис] / Dave Gray // YouTube. – URL: https://www.youtube.com/watch?v=HyZzCHgG3AY
21. Офіційна документація мови програмування TypeScript [Електронний ресурс] // TypeScript. – URL: https://www.typescriptlang.org/docs/
22. Документація CSS-фреймворку Tailwind CSS [Електронний ресурс] // Tailwind CSS. – URL: https://tailwindcss.com/docs
23. Роутинг у односторінкових додатках за допомогою React Router [Електронний ресурс] // React Router. – URL: https://reactrouter.com/
24. Генерація інтерактивної документації API за допомогою Swagger у ASP.NET Core [Електронний ресурс] // Swagger. – URL: https://swagger.io/tools/swagger-ui/
25. Автоматичне відображення об'єктів між типами за допомогою бібліотеки AutoMapper [Електронний ресурс] // AutoMapper Documentation. – URL: https://docs.automapper.org/
26. Структуроване логування у .NET-додатках за допомогою Serilog [Електронний ресурс] // Serilog. – URL: https://serilog.net/
27. Покращена читабельність тестів у .NET за допомогою FluentAssertions [Електронний ресурс] // FluentAssertions Documentation. – URL: https://fluentassertions.com/

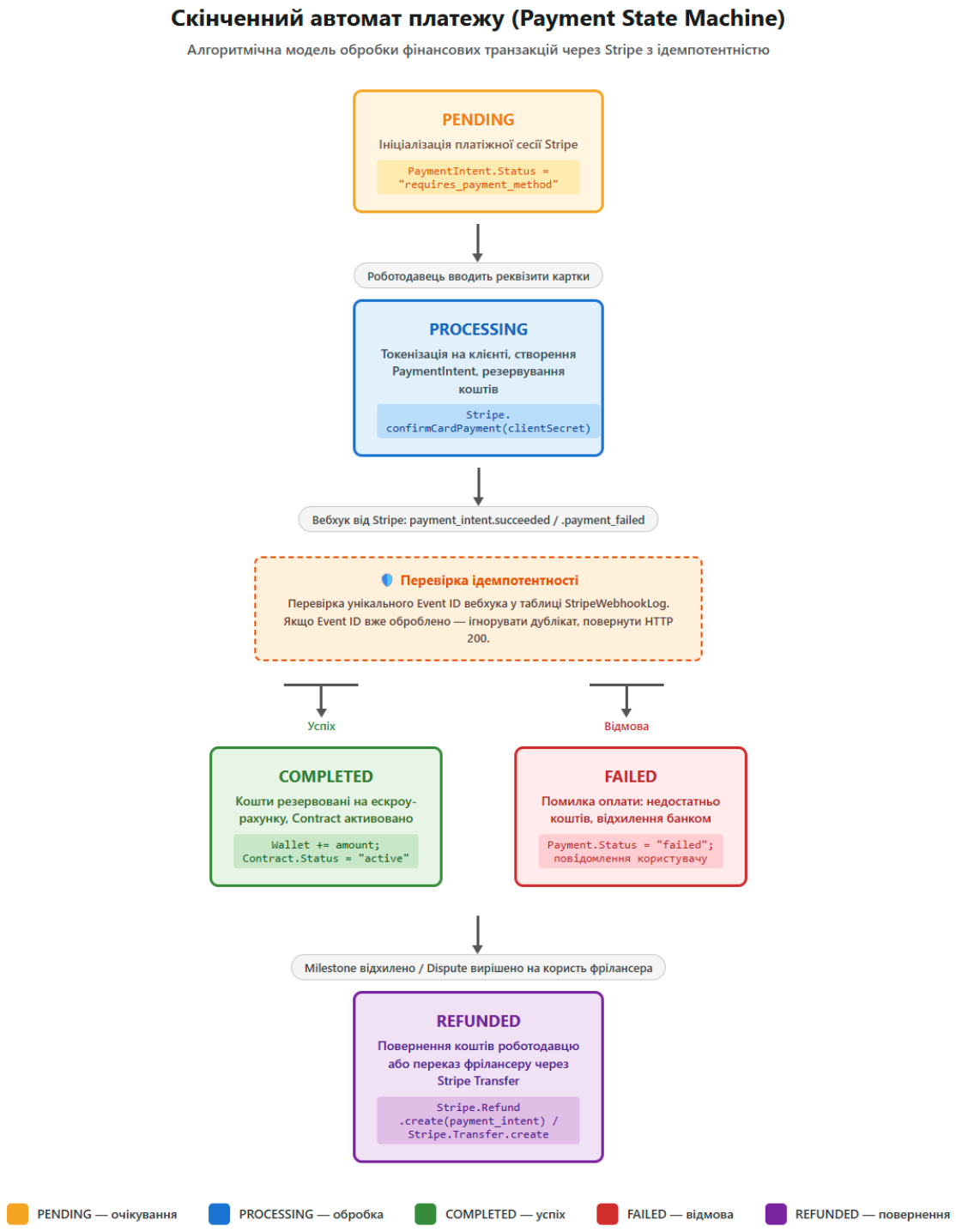
28. Імітація залежностей при модульному тестуванні за допомогою Moq [Електронний ресурс] // GitHub. – URL: https://github.com/devlooped/moq
29. Загальні положення щодо JSON Web Tokens (JWT) [Електронний ресурс] // JWT.io. – URL: https://jwt.io/introduction
30. Аутентифікація користувачів через Google OAuth 2.0 [Електронний ресурс] // Google for Developers. – URL: https://developers.google.com/identity/protocols/oauth2
31. Аутентифікація та авторизація у веб-додатках ASP.NET Core [Електронний ресурс] // Microsoft Learn. – URL: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/
32. Веб-сервер та зворотний проксі Nginx [Електронний ресурс] // Nginx Documentation. – URL: https://nginx.org/en/docs/
33. Безперервна інтеграція та розгортання за допомогою GitHub Actions [Електронний ресурс] // GitHub Docs. – URL: https://docs.github.com/en/actions
34. Ефективне управління формами у React за допомогою React Hook Form [Електронний ресурс] // React Hook Form. – URL: https://react-hook-form.com/
35. Робота з датами та часом у JavaScript за допомогою date-fns [Електронний ресурс] // date-fns. – URL: https://date-fns.org/
36. HTTP-клієнт для браузера та Node.js Axios [Електронний ресурс] // Axios. – URL: https://axios-http.com/
37. Бібліотека доступних компонентів інтерфейсу shadcn/ui [Електронний ресурс] // shadcn/ui. – URL: https://ui.shadcn.com/
38. Міжнародний стандарт управління безпекою інформації ISO/IEC 27001 [Електронний ресурс] // ISO. – URL: https://www.iso.org/standard/54534.html
39. Топ-10 найбільш критичних ризиків безпеки веб-додатків OWASP Top 10 [Електронний ресурс] // OWASP. – URL: https://owasp.org/www-project-top-ten/

ДОДАТКИ

Додаток А



Додаток Б


Скінченний автомат платежу (Payment State Machine)
Алгоритмічна модель обробки фінансових транзакцій через Stripe з ідемпотентністю
PENDING
Ініціалізація платіжної сесії Stripe
PaymentIntent.Status = "requires_payment_method"
Роботодавець вводить реквізити картки
PROCESSING
Токенізація на клієнті, створення PaymentIntent, резервування коштів
Stripe.confirmCardPayment(clientSecret)
Вебхук від Stripe: payment_intent.succeeded / .payment_failed
Перевірка ідемпотентності
Перевірка унікального Event ID вебхука у таблиці StripeWebhookLog. Якщо Event ID вже оброблено — ігнорувати дублікат, повернути HTTP 200.
Успіх
Відмова
COMPLETED
Кошти резервовані на ескроу-рахунку, Contract активовано
Wallet += amount; Contract.Status = "active"
FAILED
Помилка оплати: недостатньо коштів, відхилення банком
Payment.Status = "failed"; повідомлення користувачу
Milestone відхилено / Dispute вирішено на користь фрілансера
REFUNDED
Повернення коштів роботодавцю або переказ фрілансеру через Stripe Transfer
Stripe.Refund.create(payment_intent) / Stripe.Transfer.create
PENDING — очікування
PROCESSING — обробка
COMPLETED — успіх
FAILED — відмова
REFUNDED — повернення

Додаток В

Лістинг 3.3. Сценарій конвеєра безперервної інтеграції та доставки (CI/CD) у середовищі GitHub Actions.

```yaml
name: CI/CD

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  test:
    name: Build & Test
    runs-on: ubuntu-latest

    permissions:
      contents: read
      checks: write

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: "8.0.x"

      - name: Verify Docker is available
        run: docker info

      - name: Restore dependencies
        run: dotnet restore FreelanceBack.sln

      - name: Build
        run: dotnet build FreelanceBack.sln --no-restore -c Release

      - name: Test
        run: dotnet test FreelanceBack.sln --no-build -c Release --verbosity
normal --logger trx --results-directory ./test-results
```

```
      env:
        DOCKER_HOST: unix:///var/run/docker.sock

    - name: Publish test results
      uses: dorny/test-reporter@v1
      if: always()
      with:
        name: Test Results
        path: ./test-results/*.trx
        reporter: dotnet-trx
        fail-on-error: true

    - name: Upload test results artifact
      uses: actions/upload-artifact@v4
      if: always()
      with:
        name: test-results
        path: ./test-results/

docker:
  name: Build & Push Docker Image
  runs-on: ubuntu-latest
  needs: test
  # Публікуємо image тільки при push (не для PR)
  if: github.event_name == 'push'

  permissions:
    contents: read
    packages: write

  steps:
    - name: Checkout
      uses: actions/checkout@v4

    - name: Log in to GitHub Container Registry
      uses: docker/login-action@v3
      with:
        registry: ${{ env.REGISTRY }}
        username: ${{ github.actor }}
        password: ${{ secrets.GITHUB_TOKEN }}

    - name: Extract Docker metadata (tags & labels)
      id: meta
      uses: docker/metadata-action@v5
      with:
        images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
        tags: |
          type=sha,prefix=sha-,format=short
```

```
            type=raw,value=latest,enable={{is_default_branch}}

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build and push
        uses: docker/build-push-action@v5
        with:
          context: .
          file: ./Dockerfile
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```