МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)
на тему:

«Розробка платформи для організації та виконання завдань за винагороду з вбудованим чатом»

Виконав студент групи ІПЗ-20-1
ВЕРБОВСЬКИЙ Олександр Юрійович

Керівник роботи:
ЛОКТІКОВА Тамара Миколаївна

Рецензент:
ЄФРЕМОВ Юрій Миколайович

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»
Зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«23» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ВЕРБОВСЬКИЙ Олександр Юрійович**
Керівник роботи: **ЛОКТІКОВА Тамара Миколаївна**
Тема роботи: **«Розробка платформи для організації та виконання завдань за винагороду з вбудованим чатом»**,
затверджена Наказом закладу вищої освіти від **«23» лютого 2024 р. №74с**
Вихідні дані для роботи: **платформа для виконання завдань за винагороду, яка включає різноманітні функції, такі як створення завдань, прийняття до виконання, обговорення умов, вирішення конфліктів та проведення фінансових транзакцій.**
Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	**Консультант**	Завдання видав	Завдання прийняв
1	Локтікова Т.М.	23.02.2024р.	23.02.2024р.
2	Локтікова Т.М.	23.03.2024р.	23.03.2024р.
3	Локтікова Т.М.	13.05.2024р.	13.05.2024р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з вебдодатку платформи та пояснювальної записки. Пояснювальна записка до випускної роботи містить текстову частину, викладену на 110 сторінки друкованого тексту. Список використаних джерел містить 32 найменувань і займає 3 сторінки. У роботі наведено 48 рисунків та 41 таблиця. Загальний обсяг роботи – 120 сторінка.

Метою роботи є створення інтегрованої та зручної платформи, яка забезпечить ефективне виконання завдань за винагороду, сприятиме взаємодії користувачів та забезпечить безпеку та надійність фінансових операцій.

Під час аналізу можливих архітектурних рішень для платформи обміну послугами та завдань було визначено, що клієнт-серверна архітектура є оптимальним вибором. Дана архітектура забезпечує ефективну взаємодію з сервером, можливість асинхронної роботи і чітке відокремлення логіки додатку та інтерфейсу.

Застосунок пропонує зручний інтерфейс, що має простий процес створення завдань, узгодження умов їх виконання, проведення фінансових операцій з реальними коштами користувачів та можливість взаємодії з підтримкою.

КЛЮЧОВІ СЛОВА: АДМІНІСТРУВАННЯ, БАЗА ДАНИХ, ПЛАТФОРМА, КОРИСТУВАЧ, АДМІНІСТРАТОР, ЧАТ, СОКЕТИ, ЗАПИТИ, ЗАВДАННЯ, АВТОРИЗАЦІЯ, ЗАМОВНИК, ВИКОНАВЕЦЬ, НАГОРОДА.

ABSTRACT

The bachelor's thesis consists of a web application platform and an explanatory note. The explanatory note contains a textual part, presented on the 110 pages of printed text. The list of used sources contains 32 titles and occupies 3 pages. The work includes 48 illustrations and 41 tables. The total volume of the work is 120 pages.

The aim of the work is to create an integrated and user-friendly platform that ensures efficient task completion for rewards, promotes user interaction, and ensures the security and reliability of financial operations.

During the analysis of possible architectural solutions for the platform, it was determined that the client-server architecture is the optimal choice. This architecture provides efficient interaction with the server, asynchronous operation, and clear separation of the application logic and interface.

The application offers a convenient interface with a simple process for creating tasks, agreeing on their terms, conducting financial transactions with real users' money, and interacting with support.

KEYWORDS: ADMINISTRATION, DATABASE, PLATFORM, USER, ADMINISTRATOR, CHAT, SOCKETS, REQUESTS, TASKS, AUTHORIZATION, CLIENT, EXECUTOR, REWARD.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

АПЗ - архітектра прогамного забезпечення

API - інтерфейс застосунків

БД - база даних

ПЗ - програмне забезпечення

СУБД - система управління базами даних

MVC - архітектурний шаблон, який розділяє додаток на три основні компоненти: Model, View, Controller

MVVM - архітектурний шаблон, який розділяє додаток на три основні компоненти: Model, View, ViewModel

CRUD - основні операції, що використовуються для управління даними (створення, читання, оновлення, видалення)

РРЧ – режим реального часу

SSD - Solid State Drive

HDD - Hard Disk Drive

SPA - Single Page Applications

ВСТУП

Актуальність теми полягає у стрімкому зростанні популярності веб-технологій, які стимулюють створення різних цікавих платформ, наприклад для ефективної організації та виконання завдань за винагороду. Це особливо актуально у сучасному світі, де люди зіткнулися з труднощами у виконанні звичних завдань, що потребують фізичної присутності у конкретних місцях, через вимушену міграцію або інші проблеми пов’язані з війною чи постпандемійним періодом.

Платформа, яка дозволяє виконувати завдання за винагороду, буде дуже популярною, так як вона буде сприяти виконанню завдань різного характеру та складності, а заодно наданню дистанційної підтримки. Цей підхід ефективно зберігатиме час і ресурси, забезпечуючи при цьому високий рівень вирішення задач.

Така платформа стимулюватиме підтримку та співпрацю між людьми, незалежно від їх місця перебування, чим буде спрощувати життя та допомагатиме зберегти важливі соціальні зв'язки, навіть коли вони фізично неможливі.

Метою даної розробки є створення сучасної та зручної платформи, яка дозволить користувачам виконувати різноманітні завдання за винагороду, забезпечуючи при цьому надійну та ефективну комунікацію між виконавцями та замовниками. Важливим аспектом є створення умов для безпечного та прозорого виконання фінансових транзакцій, а також підтримка високого рівня безпеки даних користувачів. Розробка платформи також включає впровадження інтуїтивно зрозумілого інтерфейсу, який сприятиме комфортній роботі користувачів та оптимізує процес взаємодії на платформі.

Для реалізації поставленої мети потрібно виконати наступні завдання:

1. Провести аналіз аналогів платформи, дослідити існуючі рішення, їх функціональність, переваги та недоліки.

2. Здійснити вибір засобів проектування, ретельно оцінюючи доступні інструменти та технології.
3. Розробити платформу з використанням вибраних засобів, створити архітектуру та основні компоненти системи, інтегрувати всі необхідні функції та модулі для забезпечення повної працездатності платформи.
4. Провести тестування платформи різними методиками для зменшення ймовірності багу чи помилки.

Об’єктом дослідження є платформа для виконання завдань за винагороду, яка включає різноманітні функції, такі як створення завдань, прийняття до виконання, обговорення умов, вирішення конфліктів та проведення фінансових транзакцій. Особлива увага приділяється тому, як користувачі взаємодіють із платформою, яким чином здійснюється управління завданнями, комунікації між собою, а також забезпеченню надійності та безпеки системи. Платформа повинна ефективно обробляти дані, забезпечувати безперебійне виконання процесів та надавати користувачам можливість комфортної роботи.

Предметом дослідження є алгоритми та методи, що забезпечують ефективну роботу платформи, а також взаємодія між користувачами під час створення, виконання та оплати завдань. Це включає дослідження способів оптимізації алгоритмів обробки даних, забезпечення швидкості та надійності фінансових транзакцій, а також методів підтримки високого рівня безпеки і захисту даних користувачів. Крім того, предметом дослідження є розробка і впровадження зручного та інтуїтивного інтерфейсу, який дозволить користувачам легко взаємодіяти з платформою та отримувати максимальну користь від її використання.

За темою випускної кваліфікаційної роботи бакалавра було опубліковано тези: Євдокимов В. В., Марчук Г. В. Краудфандингу WEB 3.0. Тези доповідей V Всеукраїнської науково-технічної конференції Комп’ютерні технології: інновації, проблеми, рішення, 01-02 грудня 2022 року. Житомир: «Житомирська політехніка», 2022. С.164-165.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ПЛАТФОРМИ

1.1 Опис проекту та постановка задачі

На сьогоднішній день, в умовах війни, пост-пандемійного періоду та інших ситуацій, що впливають на рішення людей щодо зміни місця перебування на тривалий термін, дуже складно вирішити якісь прості дії самотужки, так як для цього потрібно витратити багато сил, коштів, енергії та інших ресурсів, щоб дібратись до пункту призначення і виконати цю задачу. Наприклад, дуже складно, привітати свою подругу чи друга з якоюсь подією в червні, подарувавши букетик ромашок та кавун, без допомоги друзів, що перебувають в одній локації, що і потрібна людина.

Даний проект, має на меті полегшити та зробити більш доступними такі види комунікації та взаємодії в умовах глобальної нестабільності. Дана платформа дозволить людям як віддалено обмінюватися реальними матеріальними подарунками, так і просто: співати гарно пісні, розповідати виразно вірші, які ще не бачив світ чи просто передавати пакет продуктів рідній бабусі, яку вже не бачив декілька років, не маючи змоги приїхати і допомогти їй.

Тож одним з найважливіших моментів продукту є забезпечення можливість віртуальної присутності та активної участі у подіях, навіть якщо фізично люди знаходяться далеко від потрібної локації. Така платформа сприятиме відновленню та зміцненню зв'язків між людьми в умовах сучасної нестабільності та важливості віддаленої спільності.

Крім того, враховуючи сучасні технології та потреби співтовариства, інтерфейс продукту повинен бути інтуїтивним та ефективним для користувачів будь-якої вікової категорії та будь-якого досвіду у використанні гаджетів та інтернету, який дозволить навіть бабусям та дідусям, у похилому віці, передавати своїм онукам приємні враженнями та висловлювати теплі слова через віртуальну присутність.

Також в контексті розробки проекту важливим аспектом є впровадження чату у РРЧ з розширеними можливостями відправки різноманітного контенту. Це обґрунтовано необхідністю встановлення надійного та швидкого зв'язку між замовником та виконавцем задачі. Реальний час дозволяє інтерактивно обмінюватися думками, реакціями та динамічно вмішуватись у виконання задачі на різних етапах, та змінювати її хід, напрямок, тривалість і якість, тим самим надаючи повний контроль замовнику над задачею створюючи відчуття, наче він і є виконавцем.

Ще одним важливим аспектом чату є можливість відправки фото, відео та інших файлів в межах чату важливо для розширення спектру виразних можливостей та можливостей комунікації. Відправлення зображень та відео дозволяє виконавцям краще розуміти, що відбувається і що від них вимагається, а замовникам – переконуватись у якості виконання задачі, детальнішому поглибленні у якусь непередбачену проблему та бути “Тут і зараз” в будь-якій точці світу о будь-якій годині доби.

Крім цього у проекті акцентується увага на технічних інноваціях, спрямованих на забезпечення найвищого рівня зручності та функціональності для користувачів. Платформа побудована з використанням передових технологій, що дозволяють забезпечити надійний та швидкий обмін повідомленнями, надійне виконання операцій з коштами, та надійне збереження даних для подальшого відтворення у зв’язку з різними причинами.

Також однією з найважливіших складових платформи є система коментарів, як виконавців, так і замовників, оскільки вона сприяє взаємодії, довірі та взаєморозумінні між користувачами. Також система коментарів надає можливість користувачам висловлювати свої думки та враження від співпраці. Замовники, переглядаючи відгуки, можуть отримати об'єктивне уявлення про рівень професіоналізму та надійності виконавців. Рейтинг та відгуки формують довіру, що є критичним аспектом в прийнятті рішення, так

як на основі цього, у замовника та виконавця формується перше відчуття довіри один до одного.

1.2 Аналіз аналогів програмного продукту

На момент розробки платформи, проведено ретельний аналіз та порівняння з існуючими аналогами, які спрямовані на подібне призначення, функціонал та мають схожу цільову аудиторію замовників. Цими програмами виявились Upwork, Fiverr та TaskRabbit, що вже зарекомендували себе на ринку як визнані лідери у сфері фрілансу та надання послуг.

Upwork – це додаток, визнаний своєю орієнтацією на фріланс та надає можливість замовникам шукати виконавців для виконання різноманітних завдань, починаючи від програмування та дизайну до письмових послуг та перекладу. Платформа дозволяє створювати проекти, розміщувати завдання та взаємодіяти з фрілансерами через вбудований чат. Оплата за виконану роботу здійснюється через систему електронного платежу, що забезпечує безпечні та ефективні транзакції між замовниками та фрілансерами.

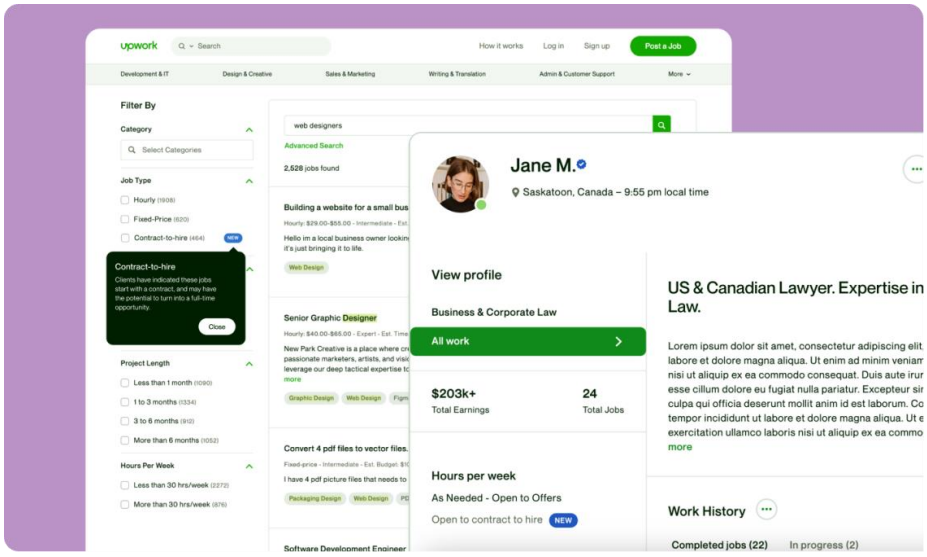

Рисунок 1.1 - Робоча сторінка додатку Upwork

Fiverr – це онлайн-платформа, спрямована на надання послуг у різних областях. Замовники можуть знаходити фрілансерів, готових виконати

конкретні завдання або надати конкретні послуги. Fiverr відомий своєю простою та інтуїтивно зрозумілою платформою, де користувачі можуть замовляти різноманітні послуги, від графічного дизайну до відеомонтажу. Оплата за послуги здійснюється через вбудовану систему оплати, забезпечуючи зручний та безпечний обмін коштами між замовниками та фрілансерами. Комунікація між клієнтом та виконавцем здійснюється через вбудовані інструменти спілкування, включаючи чатову систему та можливість обміну повідомленнями. Це створює ефективний та прозорий зв'язок, дозволяючи обговорювати деталі проекту, уточнювати вимоги та отримувати зворотний зв'язок для досягнення найкращого результату. Також на платформі є прозорий механізм відгуків, який допомагає іншим користувачам обирати виконавців на основі їхньої репутації та якості послуг. Крім того, важливою є можливість перегляду портфоліо виконавців, щоб забезпечити відповідність їхніх навичок та стилю вимогам конкретного проекту.

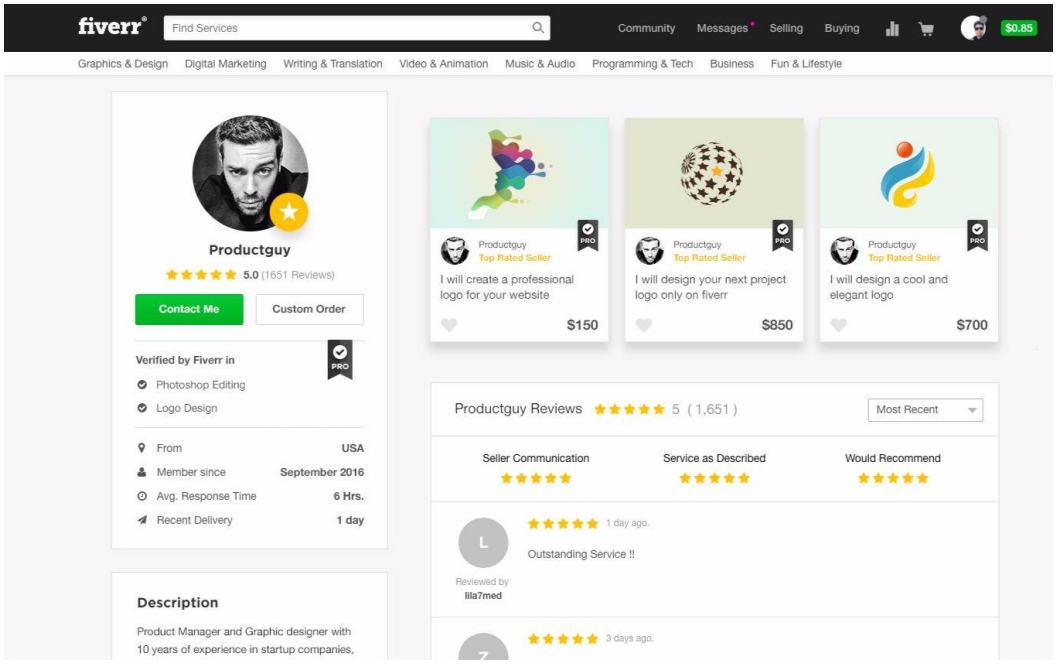

Рисунок 1.2 – Робоча сторінка додатку Fiverr

TaskRabbit – це платформа, орієнтована на надання послуг у сфері реального життя та допомоги у вирішенні різноманітних повсякденних завдань. Замовники можуть шукати фахівців, готових виконати різні

завдання, такі як прибирання, ремонт, переїзд та інші ділові послуги. Платформа дозволяє легко знаходити та спілкуватися з фахівцями, а оплата за виконані послуги також проводиться через систему електронного платежу, щоб забезпечити прозорий та безпечний процес транзакцій між сторонами. TaskRabbit визначається своєю акцентуацією на реальних потребах користувачів і підтримкою вирішення їхніх повсякденних завдань. Комунікація на платформі відбувається через будований чат, що дозволяє сторонам взаємодіяти, обговорювати деталі завдань, узгоджувати умови та надавати необхідні пояснення. Цей механізм сприяє ясному розумінню вимог та забезпечує ефективний обмін інформацією для успішного виконання завдань на платформі TaskRabbit. Крім цього на платформі існує система оглядів та оцінок, яка грає важливу роль у виборі виконавця для конкретного завдання.

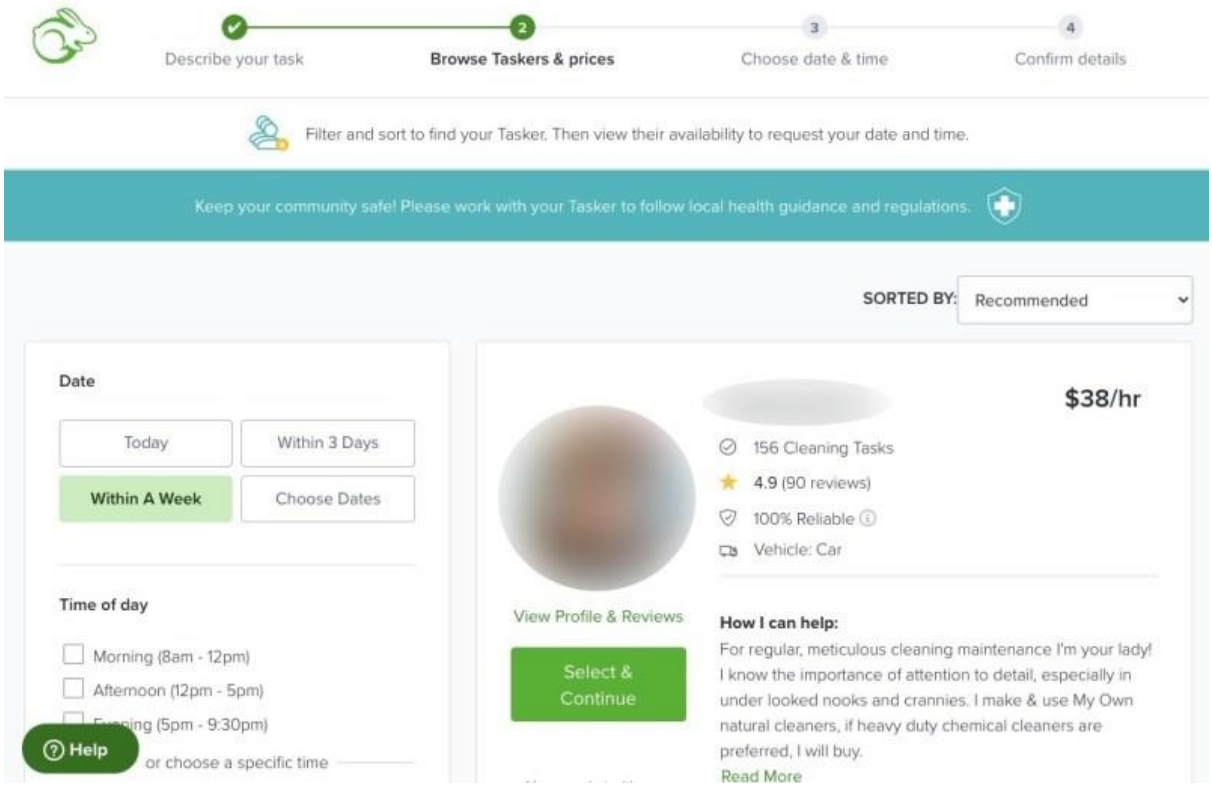

Рисунок 1.3 - Робоча сторінка додатку TaskRabbit

Загальна характеристика і порівняння всіх досліджуваних сайтів наведена нижче (табл. 1.1).

Таблиця 1.1

Порівняльна таблиця

Параметр	Upwork	Fiverr	TaskRabbit
Підсилюючі Функції	Рейтинг, глобальна аудиторія	Рейтинг, чітка категоризація послуг	Локальна орієнтація, рейтинг
Тип Виконавців	Глобальні фрілансери з різних країн	Індивіди та агентства, різноманітні категорії	Локальні виконавці для реальних завдань
Категорії Послуг	Програмування, дизайн, копірайтинг	Графічний дизайн, відеоролики, письменницькі послуги	Допомога в різних сферах життя
Система Оцінок та Відгуків	Є	Є	Є
Глобальне охоплення	Так	Так	Ні (Локально орієнтований)
Інтерфейс	Сучасний, зручний	Простий, з чіткою категоризацією	Легкий та зрозумілий
Адаптивність інтерфейсу	Так	Так	Так
Можливість Відправки Контенту	Файли	Фото, відео, файли	Фото, відео, файли
Вартість Послу	Різноманітна, визначається виконавцем	Різноманітна, визначається виконавцем	Залежить від виду та обсягу послуг
Зручність Використання	Так	Так	Так

1.3 Вибір архітектури веб-орієнтованої платформи

Оскільки платформа повинна бути надійною, безпечною, конкурентно-спроможною, легкою та зрозумілою для користувачів різного віку та націлена існувати досить тривалий час, то вона повинна бути гарно побудована на архітектурному рівні. Це необхідно, так як відразу зробити додаток у найкращій своїй версії – не можливо. Завжди потрібні додаткові

тести, перевірки та новий функціонал. Щоб баги швидко виправлялись, а нові опції швидко додавались до платформи, потрібно правильно підібрати архітектуру програмного забезпечення.

Архітектура програмного забезпечення - це високорівневе структурне представлення програмної системи, яке визначає її ключові компоненти, їх взаємодії та організацію для досягнення визначених функціональних і якісних характеристик. Архітектура програмного забезпечення визначає фундаментальні аспекти системи, такі як структура, поведінка, взаємодія компонентів, а також розподіл ресурсів, забезпечуючи основу для подальшого розвитку, тестування і супроводження програмного продукту.

Під час аналізу можливих АПЗ, а саме перечитуючи статті “Взаємодія клієнт-сервер”, “Монолітна архітектура ПЗ” та “Мікросервісна архітектура ПЗ” було зроблено висновок, що для платформи, спрямованої на обмін послуг та завдань між замовниками та виконавцями, використання клієнт-серверної АПЗ має ряд вагомих переваг [1, 2, 3].

По-перше, взаємодія із сервером. Клієнт-серверна архітектура дозволяє ефективно взаємодіяти із сервером, вимагаючи або передаючи дані лише тоді, коли це необхідно. Це дозволяє зменшити навантаження на мережу та забезпечити більш ефективний обмін інформацією.

По-друге, асинхронність. Клієнт-серверна архітектура може працювати в асинхронному режимі, що є важливим для сучасних веб-застосунків. Це дозволяє обробляти багато запитів одночасно, покращуючи продуктивність та відзначаючи систему як більш масштабовану.

По-третє, відокремлення інтерфейсу та логіки. Клієнт-серверна архітектура дозволяє чітко відокремити логіку додатку на стороні сервера від інтерфейсу користувача на стороні клієнта. Це розділення полегшує розробку, тестування та супровід, оскільки зміни в одній частині системи майже не впливають на іншу.

По-четверте, сприяння масштабованості: Клієнт-серверна архітектура забезпечує гнучкість у розгортанні, що сприяє масштабованості системи.

Можна легко додавати нові клієнти чи розширювати функціонал серверу без значних змін в коді інших компонентів.

По-п’яте, більша безпека. Через те, що логіка додатку зберігається на сервері, це дозволяє ефективніше керувати безпекою та обробкою даних. Чутливі операції можна виконувати тільки на сервері, що зменшує ризик витоку чутливої інформації на клієнтську сторону.

По-шосте, підтримка різних клієнтів. Клієнт-серверна архітектура дозволяє реалізувати різноманітні клієнтські додатки для різних платформ (веб, мобільні пристрої, настільні додатки), обслуговуючи різні потреби користувачів.

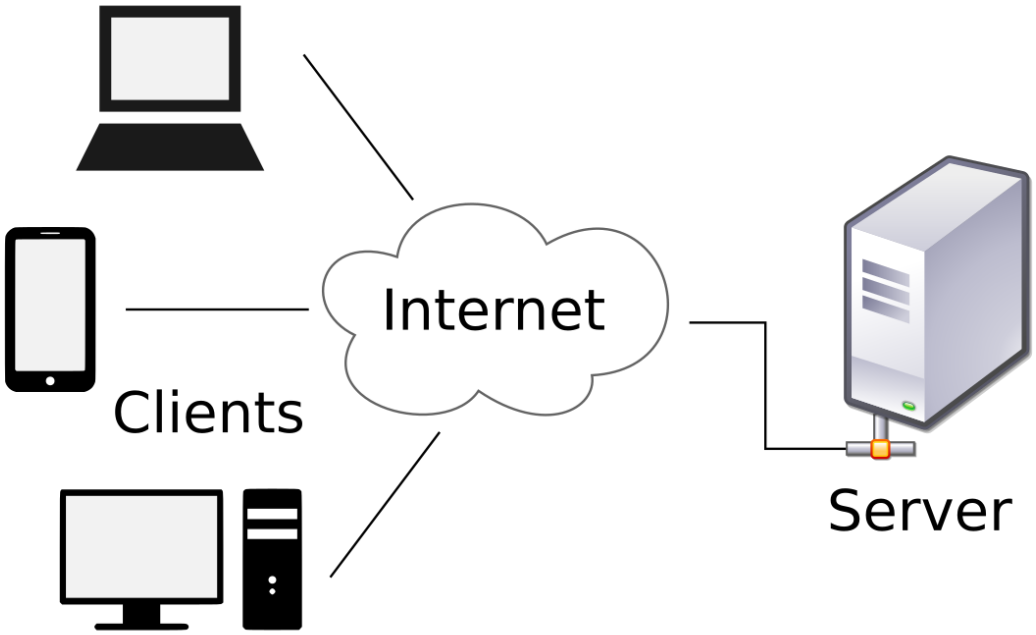

Рисунок 1.4 - Візуалізації клієнт-серверної архітектури

Аналізуючи статтю “Про архітектуру додатків”, було закцентовано увагу на двох найбільшвикористованих архітектурних патернах, а саме: на MVC та MVVM [4]. У статті “MVC проти MVVM – різниця між ними” наведено чітке порівняння між цими паттернами, тому було прийнято рішення взяти саме MVC за основу, так як цей патерн простіший у реалізації та підтримці і не вимагає спеціальних бібліотек для реалізації, тобто є не таким об’ємним за рахунок меншої кількості компонентів [5].

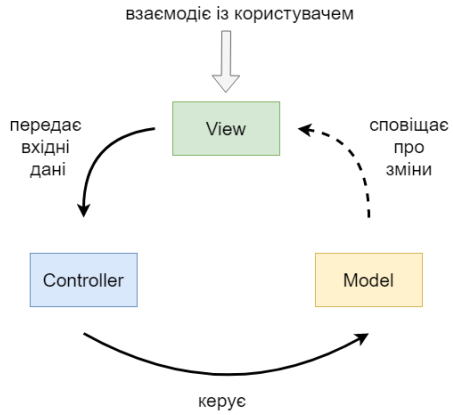

Рисунок 1.5 - Візуалізації MVC патерну

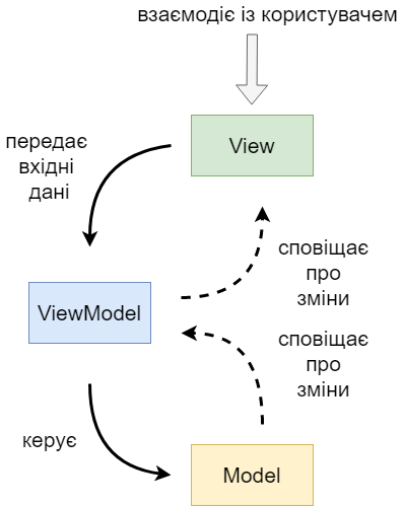

Рисунок 1.6 - Візуалізації MVVM патерну

Крім цього, патерн MVC є дуже доцільним для проекту платформи обміну послуг та завдань. Ось кілька конкретних причин, чому цей патерн підходить для реалізації платформи:

- розділення відповідальностей: MVC визначає три основні компоненти. Модель (Model), Вигляд (View) та Контролер (Controller). Це дозволяє чітко визначити та розділити відповідальності між різними частинами додатку. Модель відповідає за логіку додатку та роботу з даними, Вигляд - за представлення інформації користувачеві, а Контролер - за обробку введених користувачем дій;
- легке розширення та зміни: одна з переваг MVC полягає в тому, що зміни в одній частині системи майже не впливають на інші. Це забезпечує гнучкість та легкість у внесенні змін або розширенні функціоналу;

- покращена організація коду: використання MVC дозволяє розділити код на логічні шари, що полегшує організацію та управління кодовою базою;
- підтримка різних інтерфейсів: контролер дозволяє легко взаємодіяти з різними видами інтерфейсів, такими як веб-сторінки, мобільні додатки, API і т.д. Модель і Вигляд можуть залишатися практично незмінними, що спрощує розробку та супровід.

Для виконання етапу кодування обрано середовище розробки Visual Studio Code (VS Code), в якому буде проводитись подальша розробка додатку, використовуючи інструменти та мови програмування, такі як Node.js, React та MySQL. Visual Studio Code відомий своєю легкістю використання та розширюваністю.

VS Code — це інтегроване середовище розробки, яке надає інтелектуальний редактор з можливостями аналізу коду, автоматизованими засобами виявлення та виправлення помилок, а також широким спектром розширень для підтримки різних мов та технологій. На основі статті “Використання розширення Visual Studio Code” було додано деякі розширення до програми, що зробило її ще зручнішою для розробки [6].

Проект розроблюватиметься за допомогою Node.js для серверної частини, React для клієнтського інтерфейсу та MySQL для управління базою даних. Використання VS Code сприяє зручній та ефективній розробці завдяки його інтуїтивному інтерфейсу та широкому спектру можливостей для роботи з кодом.

1.4 Обгрунтування вибору засобів реалізації та вимоги до апаратного забезпечення

Обгрунтування вибору засобів реалізації та вимоги до апаратного забезпечення є критичним етапом у розробці будь-якого програмного продукту. Доцільність вибору конкретних технологій, мов програмування та апаратного забезпечення визначається широким спектром факторів, таких як

функціональні вимоги продукту, потреби користувачів, вимоги до продуктивності, ефективність розробки та підтримки системи.

В даному проекті, буде використано Node.js для серверної частини проекту, так як він є відкритою та ефективною платформою, побудованою на движку V8 веб-переглядача Google Chrome. Його основна перевага - це здатність виконувати JavaScript на сервері, що робить його ідеальним вибором для розробки масштабованих веб-додатків.

Крім цього Node.js використовує асинхронність, що дозволяє виконувати багато операцій одночасно без очікування завершення кожної з них. Це особливо корисно для веб-серверів, які повинні ефективно обслуговувати багато запитів від користувачів одночасно. Для проекту це означає швидший обмін даними між клієнтом і сервером, зменшуючи час очікування для кінцевого користувача.

Також Node.js здатний легко масштабуватись горизонтально, додаванням нових серверів для обробки збільшеного навантаження. Це дозволяє підтримувати високу доступність та швидкість при зростанні кількості користувачів. Це дає можливість легко розширювати систему з ростом її популярності та користувацької бази.

Крім цього, одним з плюсів Node.js є велика та активна спільнота розробників, яка забезпечує підтримку та постійний розвиток платформи. Широкий вибір пакетів (за допомогою npm - менеджера пакетів Node.js) спрощує використання готових рішень та бібліотек для розширення функціональності проекту.

Ще однією перевагою цієї мови є наявність чіткої документації до самої мови, так і до бібліотек що часто використовуються, наприклад це express.js, документації до зовнішніх апі платформ, що працюють з оплатами чи документація до бібліотеки, що взаємодіє з сокетами [7, 8, 9, 10, 11].

Також, грунтуючись на статтю “Навіщо розуміти backend-частину та чому варто обрати Node.js”, завдяки використанню JavaScript, Node.js робить можливим використання однієї мови програмування як на клієнтській, так і

на серверній частині [12]. Це спрощує взаємодію між розробниками, зменшує поріг входження в проект та зменшує потребу у використанні великої кількості розробників для підтрики коду та проекту у майбутньому.

Ще одним важливим аспектом є зручність реалізації паттерну MVC за допомогою Node.js. Це підтверджується статтею “Створення та структурування програми Node.js MVC”, де чітко вказано, що перевагою є простота [13].

Для клієнтської частини, в даному проекті, буде використано React. React - це бібліотека JavaScript для розробки інтерфейсів користувачів, що дозволяє створювати високопродуктивні та ефективні односторінкові додатки.

Однією з переваг реакту є компонентна архітектура. Платформа, яка дозволяє людям допомагати один одному в різних задачах за грошові нагороди, може включати різноманітні функції та елементи інтерфейсу. React дозволяє розділити ці функції на компоненти, які можуть бути незалежно розроблені, тестовані та підтримувані. Це полегшує масштабування та утримання коду.

Також React може похвалитись своєю швидкодією, адже він використовує віртуальний DOM та ефективний алгоритм оновлення, що робить його особливо швидким для SPA. Користувачі будуть мати швидкий та плавний досвід взаємодії з платформою.

Крім цього React є однією з найпопулярніших та найбільш підтримуваних бібліотек для розробки інтерфейсів. Його активна спільнота забезпечує багатством ресурсів, документацією та готових рішень для різних аспектів розробки.

Також реакт має унікальну в своїй реалізації документацію [14]. Вона дозволяє покроково розвиватись та розбиратись у цій бібліотеці. Ще одним плюсом є наявність великої кількості бібліотек та статей до бібліотек, що часто використовуються у парі з реактом. Наприклад – це документації до використання сокетів у реакті, використання систем оплат [15, 16, 17].

Також у реакта є свій синтаксиз. JSX полегшує опис інтерфейсу, надаючи можливість використовувати синтаксис, схожий на HTML, для створення компонентів. Це робить код більш зрозумілим та легким для розробки. Крім цього всього, у реакта дуже зручна взаємодія з backend. React можна легко інтегрувати з серверною частиною, написаною на Node.js. Це забезпечить гармонійну взаємодію між клієнтом та сервером на обох частинах вашого проекту.

Крім того, реакт дуже зручно комбінувати з бібліотекою бутстрап, яка має зручну документацію як для компонентів, так і для іконок, які гарно виглядають на сайтах, реалізованих за участі неї [18, 19].

Для збереження даних, даний проект, використовуватиме MySQL, так як вона є однією з найпоширеніших та добре вивчених реляційних систем управління базами даних. Вона вже довгий час використовується у великих проектах та корпоративних системах, що свідчить про її стабільність та надійність.

Також MySQL має широку спільноту користувачів, а також обширну документацію [20]. Дана документація гарно доповнюється іншими статями, що забезпечують необхідними ресурсами для розробки, оптимізації та управління базою даних [21, 22]. Стаття “SQL Підзапити” допомогла розібратися з користю та сенсом використання підзапитів при розробці [23].

Крім цього, одним з найважливіших моментів є те, що MySQL є вільною та відкритою системою, що робить її доступною для використання без великих витрат. Це може бути важливим фактором, особливо для початкових етапів розробки та стартапів.

Також MySQL добре інтегрується з серверними застосунками, написаними на Node.js, завдяки наявності бібліотек та драйверів для Node.js, таких як "mysql" або "sequelize". Ці бібліотеки дозволяють виконувати SQL-запити, отримувати та взаємодіяти з даними швидко, зручно і безпечно, що дозволяє в майбутньому оновлювати та покращувати функціонал платформи.

Вимог до апаратного забезпечення небагато, проте вони є, адже навіть з гарно оптимізованим кодом, додаток буде просто набором символів, що не буде приносити жодної користі та задоволення.

Перш за все, для платформи потрібно чотирьохядерний або вище процесор. Так як такий процесор часто використовується для обробки паралельних запитів та оптимізації роботи серверної частини додатку. Це забезпечує ефективну роботу, особливо в умовах великого обсягу одночасних з'єднань.

Ефективна робота серверної частини додатку вимагає значної кількості оперативної пам'яті. Оперативна пам'ять використовується для тимчасового зберігання та швидкого доступу до активних процесів та даних. Використання обсягу оперативної пам'яті у розмірі 8 ГБ, або більше, дозволить серверу зручно та швидко обробляти великі обсяги інформації, таким чином покращуючи загальну продуктивність додатку. Також важливо мати на увазі, що сервер, який оптимально використовує ресурси оперативної пам'яті, може обслуговувати більше одночасних запитів та швидше реагувати на зміни в навантаженні.

Також, на сервері має бути SSD жорсткий диск. Використання SSD є ключовим елементом для оптимізації швидкості операцій з даними на сервері. Оперативна система та серверні програми можуть швидше зчитувати та записувати дані на SSD, оскільки він не має рухомих частин, які спричиняють затримки у порівнянні з традиційним жорстким диском HDD. SSD також забезпечує високий рівень надійності та тривалості роботи, оскільки він менше схильний до механічних поломок, що робить його ідеальним вибором для систем, які працюють без перерв, таких як сервери. Пришвидшення швидкості доступу до даних сприяє зменшенню часу очікування для клієнтів та покращенню загального користувацького досвіду. Так як платформа буде зберігати велику кількість даних, фото та відео-матеріалів, то краще всього, щоб розмір диску був більшим за 500 ГБ.

Висновки до першого розділу

У першому розділі бакалаврської роботи детально проаналізовано особливості сучасних платформ, подібних за ціллю та функціоналом до розробляємої, визначено чітку мету та поставлені завдання проекту.

Також визначено середовище програмного забезпечення для розробки сайту та визначено набір інструментів, які будуть використовуватися. Зокрема, вибір був зроблений на користь Node.js для серверної частини проекту та React для клієнтської. Цей вибір обгрунтовано їх ефективністю, швидкодією та можливістю легкої масштабованості.

У третьому підрозділі бакалаврської роботи була проведена глибока робота над архітектурними аспектами платформи. Увага була зосереджена на визначенні ефективної та добре структурованої архітектури, яка відповідає потребам проекту. З цією метою було обрано патерн проектування MVC, який дозволяє чітко розділити логічні компоненти системи.

Таким чином, перший розділ проекту встановлює ключові принципи та фундаментальні рішення для подальшої розробки та реалізації платформи, визначаючи її стратегічний напрямок та основні технічні параметри.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ПЛАТФОРМИ

2.1 Визначення варіантів використання та об’єктно-орієнтованої структури системи

Платформа надає вебінтерфейс, через який користувачі можуть створювати завдання, вказувати в них очікуваний час роботи та ціну, яку вони хочуть платити за робочу годину виконавця. Крім того, вказують локацію де потрібно виконати завдання, опис для визначення особливостей виконання та умов завершення по яким виконавець буде, за що саме буде платити йому замовник. Для відправки пропозицій на виконання, робітники мають відповідну форму, де вказують необхідний робочий час для виконання завдання та ціну, яку вони хочуть отримувати за годину своєї роботи. Також, платформа надає можливість вибору, тобто кожен замовник може отримати декілька пропозицій на виконання завдання, обговорити з виконавцями деталі через внутрішній чат і після, вирішити яка з них є для нього найпривабливішою та підтвердити її. Після цього платформа надає функціонал для відслідковування та зміни поточного статусу завдання, комунікації між користувачами додатком та зі створенням суперечки з залученням адміністратора сайту. У свою чергу адміністратори мають зручний функціонал для вирішення суперечок, де мають змогу повного перегляду чату між користувачами у яких виник конфлікт (усі зміни та видалення теж відображаються у візуально-зрозумілому вигляді), комунікації з кожним учасником конфлікту та перегляду інформації про них, себто статистики та деяких деталей, таких як пошта, біографія та місце роботи.

Для кращого візуального розуміння взаємодії між елементами додатку було створено діаграму варіантів використання (рис 2.1).

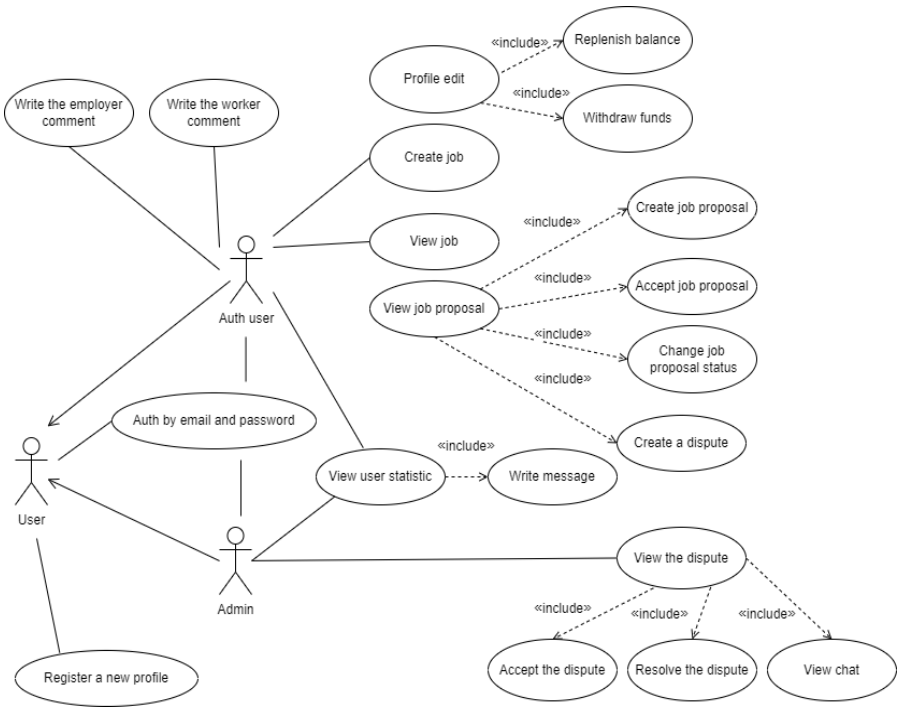

Рисунок 2.1 Діаграма варіантів використання

Опис діаграми варіантів використання наведено в таблицях 2.1 – 2.19.

Таблиця 2.1

Опис прецеденту «Вхід»

Назва варіанту використання	Вхід
Основні актори	Неавторизований користувач
Короткий опис	Користувачі можуть авторизовуватись і залежно від їх ролей на платформі виконувати певні операції

Таблиця 2.2

Опис прецеденту «Реєстрація»

Назва варіанту використання	Реєстрація
Основні актори	Неавторизований користувач
Короткий опис	Користувачі можуть реєструватись для створення нових задач, заробляння коштів, або виконання обов’язків адмінів

Таблиця 2.3

Опис прецеденту «Редагування профілю»

Назва варіанту використання	Редагування профілю
Основні актори	Користувач
Короткий опис	Користувачі можуть дозаповнювати інформацію про свій профіль, або змінювати її, залежно від актуальності

Таблиця 2.4

Опис прецеденту «Поповнення балансу»

Назва варіанту використання	Поповнення балансу
Основні актори	Користувач
Короткий опис	Користувачі можуть поповнювати свій баланс на сайті для виконання внутрішніх фінансових операцій

Таблиця 2.5

Опис прецеденту «Поповнення балансу»

Назва варіанту використання	Поповнення балансу
Основні актори	Користувач
Короткий опис	Користувачі можуть виводити кошти на зручні для них платформи

Таблиця 2.6

Опис прецеденту «Створення задач»

Назва варіанту використання	Створення задач
Основні актори	Користувач
Короткий опис	Користувачі можуть створювати задачі, які їм треба, щоб виконали інші користувачі

Таблиця 2.7

Опис прецеденту «Підтвердження пропозиції»

Назва варіанту використання	Підтвердження пропозиції
Основні актори	Користувач
Короткий опис	Користувачі може підтвердит, що пропозиція йому підходить і автор пропозиції розпочинає її виконувати

Таблиця 2.8

Опис прецеденту «Зміна статусу виконання задачі»

Назва варіанту використання	Зміна статусу виконання задачі
Основні актори	Користувач
Короткий опис	Користувачі може змінити статус виконання задачі залежно від його ролі в задачі та поточного стану виконання. Замовник може підтвердити успішне виконання задачі, а виконавець відправити запит на підтвердження

Таблиця 2.9

Опис прецеденту «Створення суперечок»

Назва варіанту використання	Створення суперечок
Основні актори	Користувач
Короткий опис	При виникненні якоїсь недомовленості при виконанні задачі, користувачі можуть звернутись до адміністратора для вирішення конфлітку

Таблиця 2.10

Опис прецеденту «Перегляд суперечки»

Назва варіанту використання	Перегляд суперечки
Основні актори	Адміністратор
Короткий опис	Адміністратор має можливість переглядати детальну інфомрацію про виникнені суперечки перед тим як братись за їх вирішення

Таблиця 2.11

Опис прецеденту «Перегляд чату суперечки»

Назва варіанту використання	Перегляд чату суперечки
Основні актори	Адміністратор
Короткий опис	Адміністратор має можливість переглядати чат між корисутвачами, які є учасниками суперечки. Це дозволить адміністратуру краще зрозуміти ситуацію і розібратись в ній

Таблиця 2.12

Опис прецеденту «Закріплення суперечки»

Назва варіанту використання	Закріплення суперечки
Основні актори	Адміністратор
Короткий опис	При закріпленні суперечки, адміністратор позначається в ній як виконавець і інші адміністратори не можуть взятись за її вирішення

Таблиця 2.13

Опис прецеденту «Перегляд задачі»

Назва варіанту використання	Перегляд задачі
Основні актори	Користувач
Короткий опис	Користувачі можуть переглядати інформації про задачі, інформацію про власника, його статистику і так далі, щоб впевнитись чи їм підходять умови для успішного виконання

Таблиця 2.14

Опис прецеденту «Вирішення суперечки»

Назва варіанту використання	Вирішення суперечки
Основні актори	Адміністратор
Короткий опис	При вирішенні суперечки, адміністратор може обрати хто є правим у ній і чи отримає власник зарезервовані кошти, чи все таки виконавець забере зароблене

Таблиця 2.15

Опис прецеденту «Перегляд профілю користувача»

Назва варіанту використання	Перегляд профілю користувача
Основні актори	Адміністратор, користувач
Короткий опис	Адміністратор та користувач мають можливість переглядати профілі інших користувачів для вирішення чи варто з ними мати справу, для аналізу коментарів та іншої статистики

Таблиця 2.16

Опис прецеденту «Відправка повідомлень»

Назва варіанту використання	Відправка повідомлень
Основні актори	Адміністратор, користувач
Короткий опис	Адміністратор та користувач мають можливість відправляти повідомлення іншим юзерам з метою покращення комунікації

Таблиця 2.17

Опис прецеденту «Відправка коментаря виконавцю»

Назва варіанту використання	Відправка коментаря виконавцю
Основні актори	Користувач
Короткий опис	Користувачі можуть оцінювати один одного як виконаців різних задач.

Таблиця 2.18

Опис прецеденту «Відправка коментаря замовнику»

Назва варіанту використання	Відправка коментаря замовнику
Основні актори	Користувач
Короткий опис	Користувачі можуть оцінювати один одного як замовників різних задач.

Таблиця 2.19

Опис прецеденту «Перегляд запитів на виконання задач»

Назва варіанту використання	Перегляд запитів на виконання задач
Основні актори	Користувач
Короткий опис	Користувачі можуть переглядати пропозиції на виконання, створених ними, задач

Основна ціль розробки цієї платформи полягає в створенні онлайн-середовища, яке сприяє ефективній комунікації та співпраці між користувачами, які мають завдання для виконання, та тими, хто готовий їх виконати. Платформа спрямована на забезпечення зручного та надійного механізму для створення, призначення та виконання завдань, а також на сприяння обміну ідеями та знаннями у чаті. Основні функціональність платформи включають можливість створювати завдання, встановлювати винагороду за їх виконання, спілкуватися у чаті для обговорення деталей та узгодження умов відстеження прогресу та завершення завдань, а також вирішення суперечок.

Основна мета платформи полягає у створенні ефективного середовища для співпраці між людьми, які мають завдання, що потребують допомоги у їх виконанні, та тими, хто готовий виконати ці завдання за певну винагороду. Платформа надає можливість розміщення різноманітних завдань, від рутинних до складніших, та забезпечує прозорий та безпечний процес співпраці між замовниками та виконавцями. Крім того, вона створює можливості для активної комунікації, обміну ідеями та навичками між користувачами, що сприяє якісній та швидкій реалізації завдань. Такий підхід дозволяє підтримувати розвиток та ефективне виконання проектів у різних сферах діяльності.

Вимоги користувачів:

Внутрішній користувач – користувач:

1. Реєстрація або авторизація для доступу до функціоналу.
2. Редагування профілю.
3. Поповнення балансу.
4. Виведення коштів на зручну платформу.
5. Створення завдання.
6. Перегляд пропозицій виконання певного завдань.
7. Обговорення пропозицій з виконавцями.
8. Перегляд статистики користувача.

9. Створення суперечки в завданні.
10. Відправка запиту на здачу завдання.
11. Підтвердження успішного завершення завдання.
12. Прийняття та відхилення пропозицій.
13. Перегляд доступних завдань для виконання.
14. Вибір та прийняття завдань для виконання.
15. Виконання завдань та надсилання результатів.
16. Отримання винагороди за успішно виконані завдання.
17. Перегляд історії виконаних завдань та отриманих винагород.
18. Написання коментарів з оцінкою виконавцю.
19. Написання коментарів з оцінкою замовнику.
20. Написання коментарів до задачі.

Внутрішній користувач – адміністратор:

1. Можливість керування користувачами та їхніми правами доступу.
2. Модерація завдань та винагород за їхнє виконання.
3. Вирішення конфліктних ситуацій.
4. Перегляд статистики платформи.

Функціональні вимоги:

1. Реєстрація користувачів: Можливість реєстрації користувачів з обов'язковими полями, такими як ім'я, електронна пошта, пароль.
2. Авторизація користувачів: Авторизація зареєстрованих користувачів з використанням електронної пошти та пароля.
3. Редагування профілю: Можливість користувачів редагувати свої профілі, включаючи фотографію, контактну інформацію та навички.
4. Створення та перегляд замовлень: Можливість роботодавців створювати замовлення для виконавців з описом завдань, бюджетом і терміном виконання. Виконавці можуть переглядати список доступних замовлень.
5. Чат для спілкування: Вбудований чат для взаємодії між роботодавцями та виконавцями щодо уточнень, деталей та обговорення завдань.

6. Оцінювання та відгуки: Можливість користувачів залишати відгуки та оцінки після виконання завдань.
7. Система суперечок: Можливість розглядати суперечки між замовниками та виконавцями та вирішувати їх.

Нефункціональні вимоги:

1. Вимоги до продуктивності:
   1.1. Час відгуку системи: Час відгуку для типових завдань повинен бути не більше 15 секунд, а для складних завдань - не більше 30 секунд.
   1.2. Підтримка одночасних користувачів: Система повинна підтримувати мінімум 40 одночасно працюючих користувачів, які взаємодіють із загальною базою даних.
2. Можливості експлуатації:
   2.1. Масштабування: Система повинна мати можливість масштабуватися, щоб збільшувати продуктивність і забезпечувати зручну роботу з ростом кількості користувачів.
   2.2. Забезпечення безпеки даних: Захист особистих даних користувачів та даних замовлень є важливою складовою.

Системні вимоги:

1. Вимоги до середовища виконання: платформа повинна задовільнять вимогам на пристроях, що знаходиться в наступній мінімальній комплектації:
   1.1. Мінімум 2 Гб оперативної пам'яті для забезпечення плавної роботи.
   1.2. Процесор з тактовою частотою не менше 1.4 GHz для швидкого виконання завдань.
   1.3. Доступ до мережі Інтернет для забезпечення з'єднання з платформою.
2. Вимоги до серверу інформаційної системи:
   2.1. У ядрі системи повинна бути встановлена реляційна база даних, що забезпечує надійне зберігання та організацію даних.

2.2.Сервер повинен підтримувати технології NodeJS для ефективної обробки запитів та забезпечення відповідності роботи платформи інформаційним потребам користувачів.

Аналіз функціональних вимог дозволив виділити наступні сутності, що забезпечать програмну реалізацію додатку. Вони представлені у вигляді діаграми основних класів платформи (рис. 2.2).

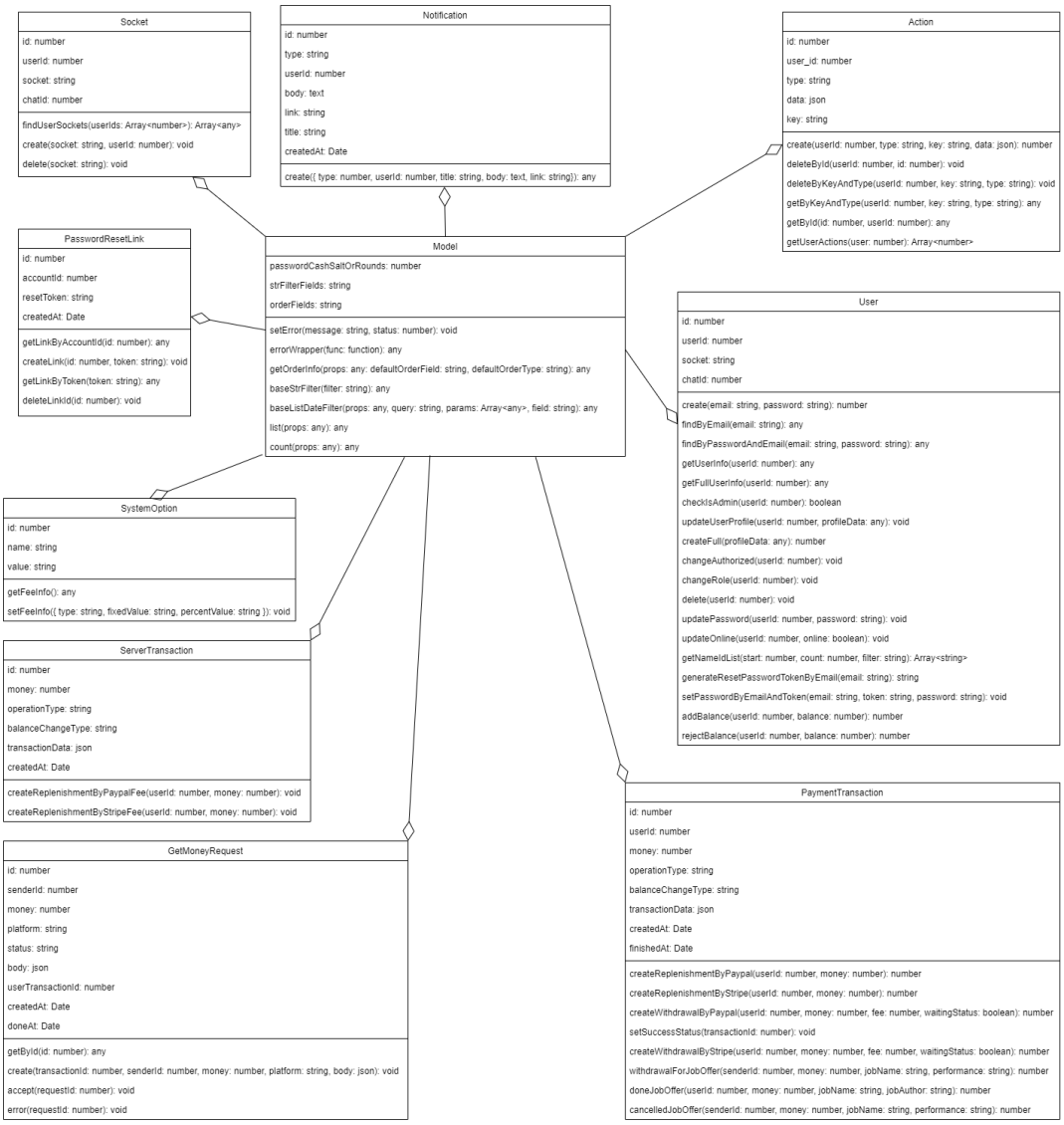

Рисунок 2.2 - Діаграма основних класів

Model – це батьківський клас, який містить декілька загальних методів, на основі яких було спрощено різні операції, які були подібними у всіх наслідувальних класах. Також, клас містить деякі базові дані, а саме: довжину примісу для шифрування паролю, або інших секретних даних,

список полів, по яким можна робити сортування та список полів по яким можна робити рядковий пошук.

Нижче наведено основні методи класу:

- errorWrapper – це метод є представником патерну “Декоратор”. усіх методів класів моделі. Його призначення відловлення помилок та повернення коректної інформації до контролеру;
- setError – генерує помилку для контролера і завершує метод моделі;
- getOrderInfo – це метод, що генерує шматок запиту, що відповідає за сортування вибірки, що отримується з бази даних;
- baseStrFilter – це метод, що генерує шматок запиту, що відповідає за фільтр полів з типом “string” чи “text” у вибірці, що отримується з бази даних;
- baseListDateFilter – це метод, що генерує шматок запиту, що відповідає за фільтр полів з типом “timestamp, тобто дати, у вибірці, що отримується з бази даних;
- list – метод, що отримує список даних на основі заданих умов;
- count – метод, що отримує кількість даних на основі заданих умов.

User – це клас, що представляє користувачів системи. Він містить інформацію про користувачів, таку як їх ідентифікатор, електронну адресу, пароль, ім'я (nick), адресу, аватар, координати на мапі (широту і довготу), прапорці авторизації профілю, прапорці адміністратора та дату створення облікового запису користувача.

- create - метод, що створює користувача лише за поштою та паролем;
- findByEmail – метод, що повертає дані користувача за поштою;
- findByPasswordAndEmail – метод, що повертає дані користувача за поштою та паролем;
- getUserInfo – метод, що повертає дані користувача за його ідентифікатором;
- getFullUserInfo – метод, що повертає повний перелік даних користувача за його ідентифікатором;

- checkIsAdmin – метод, що перевіряє чи є користувач адміністратором;
- updateUserProfile – метод, що оновлює дані про певного користувача;
- createFull – метод, що створює користувача на основі усіх даних про нього;
- changeVerification – метод, що змінює статус верифікованості користувача;
- changeRole – метод, що змінює роль користувача на платформі;
- delete – метод, що видаляє користувача;
- updatePassword – метод, що задає новий пароль користувачу;
- updateOnline – метод, що оновлює статус про онлайн користувача;
- getNameldList – метод, що повертає перелік залежностей ідентифікатор-ім’я усіх користувачів;
- addBalance – метод, що збільшує баланс користувача;
- rejectBalance – метод, що зменшує баланс користувача.

Socket – це клас, що відповідає за роботу з сокетами, які використовуються для спілкування в режимі реального часу між користувачами. Він містить інформацію про ідентифікатор сокету, ідентифікатор користувача, ідентифікатор чату, до якого підключено сокет.

- findUserSockets – це метод, що відповідає за отримання усіх сокетів для шуканих користувачів. Необхідно для дій, пов’язаних з діями у РРЧ. Наприклад створення нотифікацій чи чатингу;
- create – це метод, що зберігає інформацію про сокет користувача, що зайшов на платформу.Вихід в мережу інтернет;
- delete – це метод, що видаляє інформацію про сокет користувача, що вийшов з платформи.

Action - це клас, який використовується для ведення журналу тривалих дій користувачів. Він містить ідентифікатор дії, ідентифікатор користувача, тип дії, додаткові дані та ключ.

- create – метод, що створює запис про тривалу дію, що робить користувач на сайті даний момент;

- deleteById – метод, що стирає запис про тривалу дію, що робив користувач по ідентифікатору операції;
- deleteByKeyAndType – метод, що стирає запис про тривалу дію, що робив користувач по унікальному ключу операції та типу операції;
- getByKeyAndType – метод, що повертає інформацію про тривалу дію користувача, по унікальному ключу операції та типу операції;
- getById – метод, що повертає інформацію про тривалу дію користувача, по ідентифікатору операції;
- getUserActions – метод, що список усіх поточних тривалих дій користувача.

PasswordResetLink – це клас, що відповідає за управління посиланнями для скидання паролю користувачів. Він містить інформацію про ідентифікатор посилання, ідентифікатор облікового запису користувача, токен скидання паролю та час створення посилання.

- getLinkByAccountId – це метод, що повертає інфомрацію про запит на скидання паролю по ідентифікатору користувача;
- createLink - це метод, що зберігає інформацію про користувача та токену для скидання паролю на профілі;
- getLinkByToken - це метод, що повертає інфомрацію про запит на скидання паролю по токену;
- deleteLinkId - це метод, видаляє запит на скидання паролю.

Notification – це клас, що відповідає за нотифікації користувача. Він містить інформацію про ідентифікатор нотифікації, ідентифікатор отримувача, тип нотифікації, заголовок, основний текст, посилання, куди має перекидати нотифікація та час створення нотифікації.

- create – це метод, який зберігає дані про нотифікацію в базі даних.

SystemOption – це клас, що відповідає за основні налаштування платформи. Таким чином, вона містить різні ключові дані, наприклад податок при виводу коштів з платформи, та тип податку.

- getFeeInfo – це метод, який повертає інформацію про податки для виводу коштів;
- setFeeInfo – це метод, який зберігає інформацію про податки для виводу коштів.

ServerTransaction – це клас, що представляє інформацію про всі фінансові операції, в яких був задіяний сервер. Клас містить інформацію про ідентифікатор транзакції, тип операції, з якою сумою було проведено операцію, тип зміни балансу, додаткові дані транзакції та дата виконання операції.

- createReplenishmentByStripeFee – це метод, який створює запис про вивід коштів з гаманця на страйпі;
- createReplenishmentByPaypalFee – це метод, який створює запис про вивід коштів з гаманця на пейпалі.

PaymentTransaction – це клас, що представляє інформацію про всі фінансові операції, в яких був задіяний певний користувач. Клас містить інформацію про ідентифікатор транзакції, ідентифікатор користувача, що був учасником операції, з якою сумою було проведено операцію, тип зміни балансу, тип операції, додаткові дані транзакції, дату створення операції та дату завершення операції.

- createReplenishmentByPaypal – це метод, який створює запис про поповнення балансу через пейпал;
- createReplenishmentByStripe – це метод, який створює запис про поповнення балансу через страйп;
- createWithdrawalByPaypal – це метод, який створює запис про вивід коштів через пейпал;
- createWithdrawalByStripe – це метод, який створює запис про вивід коштів через страйп;
- setSuccessStatus – це метод, який позначає транзацію як завершену;
- withdrawalForJobOffer - це метод, який зберігає інформацію про зменшення балансу внаслідок підтвердження початку виконання задачі;

- doneJobOffer – це метод, який зберігає інформацію про поповнення балансу внаслідок завершення задачі;
- cancelledJobOffer – це метод, який зберігає інформацію про поповнення балансу внаслідок скасування виконання задачі.

GetMoneyRequest – це клас, що представляє інформацію про всі запити виводу коштів з акаунтів. Клас створений для випадків, коли система через якусь проблему не може відразу вивести кошти клієнту на платформу. Цей клас дозволяє ефективно керувати процесом виведення коштів з акаунтів користувачів на платформі та веде історію усіх операцій, щоб у користувача не виникало зайвих питань про зміну балансу. Клас містить інформацію про ідентифікатор запиту, ідентифікатор користувача, суму виводу, платформу на яку виводять кошти, статус, додаткові дані запиту, ідентифікатор транзакції користувача, яка очікує завершення, дату створення операції та дату завершення операції.

- getById – метод, що повертає інформацію про запит на вивід коштів по його ідентифікатору;
- create – метод, що створює запит на вивід коштів;
- accept – метод, що позначає запит на вивід коштів, як завершений;
- error – метод, що позначає запит на вивід коштів, як провалений.

Платформа є об’ємною і має різні модулі. Одним з основних модулів платформи є її чат. Сутності, що реалізовують цей модуль зображені у вигляді діаграми класів (рис. 2.3).

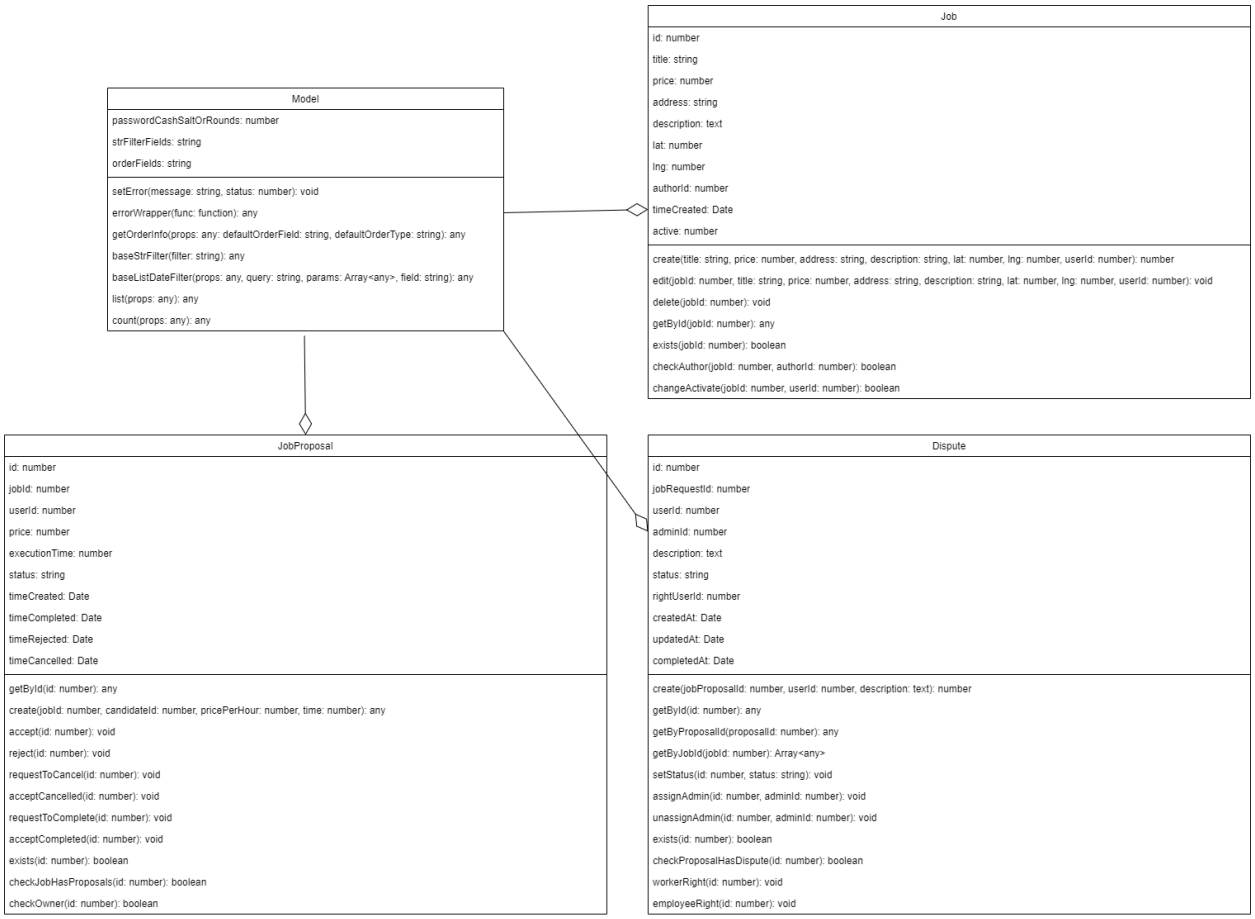

Рисунок 2.3 - Діаграма класів пов’язаних з завданнями

Job – це клас, що представляє роботи або завдання, які користувачі можуть створювати та виконувати. Він містить таку інформацію про завдання: ідентифікатор, назва, ціна за годину, адреса, опис, координати місця, дані про замовника та дату створення.

- create – створює нове завдання у базі даних;
- edit – метод, що оновлює інформацію про завдання;
- delete – метод, що видаляє завдання;
- getById – метод, що отримує інформацію про завдання за ідентифікатором;
- exists – метод, що перевіряє існування роботи за ідентифікатором;
- checkAuthor – метод, що перевіряє, чи користувач є автором роботи;

- changeActive – метод, що змінює статус актуальності задачі. Якщо цей статус має значення негативне, то більше ніхто не зможе відправити запит на створення пропозиції на виконання її.

JobProposal – це клас для роботи з запитами на виконання робіт. Він містить інформацію про ідентифікатор запиту, ідентифікатор роботи, інформацію про виконавця, ціну за годину роботи, час для виконання, статус та дату створення запиту.

- getById – метод, що отримує інформацію про пропозицію за ідентифікатором;
- create – створює нову пропозицію для роботи;
- accept – підтверджеє початок виконання завдання користувачем;
- reject – відхилення пропозиції на виконання завдання користувачем;
- requestToCancel – створює запит на скасування виконання завдання. Потрібний, якщо користувачі дійшли до якоїсь незгоди і хочуть завершити замовлення зберігши кошти та репутацію один одного, без втручання адміністратора;
- acceptCancelled – підтверджує скасування виконання завдання. Потрібний, щоб власники не могли скасувати виконання завдання самостійно в кінці, або в процесі виконання;
- requestToComplete – створює запит на підтвердження завершення завдання;
- acceptCompleted – підтверджує завершення виконання завдання. Потрібний, щоб виконавці не могли забирати кошти після виконання завдання самостійно в кінці, без доказів успішного завершення для замовника;
- exists – перевіряє наявність пропозиції виконання завдання;
- checkJobHasProposals – перевіряє чи завдання має пропозиції на виконання;
- checkOwner – перевіряє чи є користувач виконавцем завдання.

Dispute – це клас, який використовується для управління суперечками щодо виконання робіт. Він містить інформацію про ідентифікатор суперечки, ідентифікатор запиту на виконання роботи, ідентифікатор користувача, що створив суперечку, ідентифікатор адміністратора що взявся за вирішення суперечки, опис суперечки, статус та дату створення і оновлення.

- create – метод, що створює нову суперечку для пропозиції до роботи із зазначеним користувачем та описом;
- getById – метод, що отримує інформацію про суперечку за ідентифікатором спору;
- getByProposalId – метод, що отримує інформацію про суперечку, пов'язану із пропозицією до роботи за ідентифікатором пропозиції;
- getByJobId – метод, що отримує інформацію про суперечку, пов'язану із завданням за ідентифікатором завданням;
- setStatus – метод, є одним з представників реалізації патерну “Фасад”. На основі статті “Патер Фасад” можна сказати, що патерн використовується для надання простого інтерфейсу до складної системи, щоб спростити її використання [24]. У даному випадку це методи, що змінюють статус суперечки за ідентифікатором та слугує основою для методів workerRight та employeeRight;
- assignAdmin – метод, що призначає адміністратора відповідальним для суперечки за ідентифікатором суперечки;
- unassignAdmin – метод, що скасовує відповідальність адміністратора для суперечки;
- exists – метод, що перевіряє наявність суперечки;
- checkProposalHasDispute – метод, що перевіряє наявність суперечки у пропозиції;
- workerRight – метод, що обирає робітника, як того, хто правий, у суперечці;
- employeeRight – метод, що обирає замовника, як того, хто правий, у суперечці.

У Додатку А наведено діаграму класів на пряму пов’язаних з чатом.

Дана структура класів є реалізацією патерну “Компонувальник”. Після прочитання статті “Патер Компонувальник” стало зрозуміло, що цей патерн дозволяє змінювати окремі частини об’єкту не змінюючи його повністю [25]. Так як в одному чаті може бути багато користувачів, повідомлень та версій повідомлень, то логічним було застосування цього патерну для моделювання взаємодії між цими об'єктами в чаті. Наприклад, клас чату виступає у ролі великого компонента, який може містити багато малих компонентів, таких як користувачі та повідомлення. Кожен користувач може бути окремим об'єктом користувача, який, у свою чергу, може містити повідомлення як підкомпоненти. Повідомлення також можуть мати свої версії, які можна розглядати як підкомпоненти повідомлення. Використання патерну “Компоновщик” дозволяє легко маніпулювати всією структурою чату, незалежно від того, чи ми працюємо з окремим користувачем, повідомленням або версією повідомлення. Це спрощує додавання нових, видалення та редагування існуючих елементів чату та виконання різних операцій з ними.

Chat – це основний клас даної структури. Він містить усі основні функції роботи з чатом:

- hasUserAccess – метод, що перевіряє чи є у користувача доступ до чата;
- getById – метод, що отримує скорочену інформацію про чат за його ідентифікатором;
- lastReadMessageldByUser – метод, що отримує ідентифіктаор останнього прочитаного повідомлення користувачем у чаті;
- create – метод, що створює новий чат;
- addUser – метод, що додає користувача до чату;
- getChatRelations – метод, що отримує повний список користувачів чату;
- addManyUsers – метод, що додає список користувачів до чату;
- deleteUserFromChat – в метод, що идаляє користувача з чату;
- setMainAdminRole – метод, що передає права власника чату користувачу;

- setAdminRole – метод, що надає права адміністратора чату користувачу;
- unsetAdminRole – метод, що забирає права адміністратора чату користувачу;
- addContentToMessage – метод, що додає нову версію контенту повідомлення;
- createMessage – метод, що створює повідомлення в чаті;
- hideMessage – метод, що приховує повідомлення в чаті для всіх користувачів;
- hasPersonal – метод, що перевіряє чи є персональний чат між двома користувачами;
- createPersonal – метод, що створює персональний чат для двох користувачів;
- createGroup – метод, що створює групови;
- createSystemChat – метод, що створює системний чат для користувача;
- getAllMessageContents – метод, що отримує увесь список змін повідомлення;
- getMessageContent – метод, що отримує поточний контент повідомлення.
- getAllChats – метод, що отримує усі чати користувача;
- getAllUserSystemChats – метод, що отримує усі системні чати користувачів;
- getUsersToNewAdminChat – метод, що отримує інформацію про користувачів чату адміністраторів сайту;
- getUsersToNewNormalChat – метод, що отримує інформацію про користувачів чату;
- getUsersSocketToSend – метод, що отримує дані про сокети користувачів для розсилки інформації за їх айді;
- getMessageByld – метод, що отримує повну інформацію про повідомлення за його ідентифікатором;
- getChatinfo – метод, що отримує повну інформацію про чат за його ідентифікатором;

- getChatMessagesInfo – метод, що отримує повну інформацію про повідомлення чат за ідентифікатором чату;
- getChatMessages – метод, що отримує скорочену інформацію про повідомлення чат за ідентифікатором чату;
- getUnreadChatMessagesCount – метод, що отримує кількість непрочитаних повідомлень в чаті користувача;
- selectChat – метод, що позначає чат як поточно переглядаємий користувачем і повертає повну інформацію про чат;
- selectSystemChat – метод, що позначає системний чат як поточно переглядаємий користувачем і повертає повну інформацію про чат;
- getChatUsers – метод, що повертає список користувачів чату;
- getUserSocketsFromChat – метод, що повертає список сокетів користувачів чату;
- getUserChatRole – метод, що повертає роль користувача в чаті;
- deactivateUserChatRelation – метод, що деактивовує зв’язок користувача з чатом(видаляє його);
- setTyping – метод, що зберігає інформацію про те чи друкує користувач повідомлення в чаті, чи вже закінчив;
- setLastIdMessage – метод, що зберігає інформацію про останній переглянутий користувачем меседж у чаті;
- setOwnerByFirstPriority – метод, що передає права власника чату найпріоритетнішому користувачу. Актуальне при виході власника групи з чату;
- getUserSystemChatInfo – метод, що отримує інформацію про системния чат користувача.

ChatsUsers – це клас, що відповідає за збереження даних про зв’язки між користувачами та чатами.

Message – це клас, що відповідає за збереження даних повідомлень, які відправляються в чатах.

MessageContent – цей клас, що представляє вміст повідомлень, який може бути текстовим або мультимедійним.

Ще одним важливим модулем платформи є коментарі. Їх класи теж представлено у вигляді відповідної діаграми (рис. 2.4).

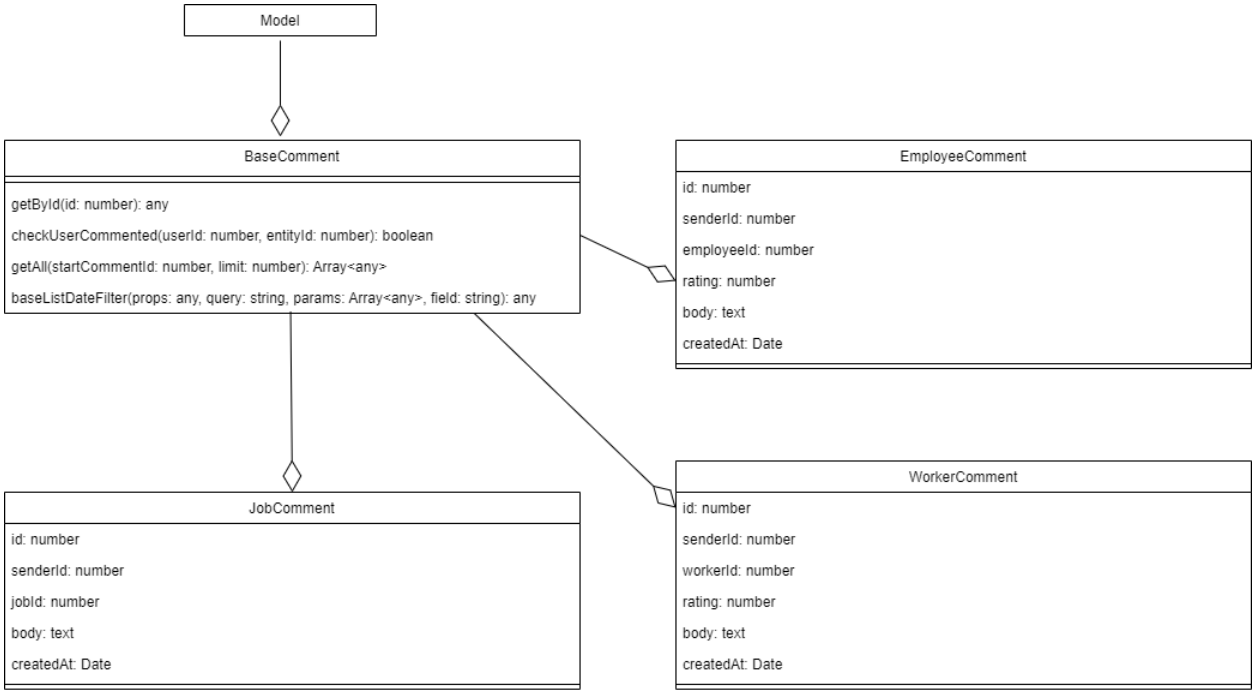

Рисунок 2.4 - Діаграма класів пов’язаних з коментарями

Для реалізації даної структури класів було використано патерн “Міст”. Після прочитання статті “Патерн Міст” було визначено його переваги, це дало можливість покращити розробку [26]. Найдоцільніше було його використати саме тут, так як для продавця, виконавця та замовлення будуть доступні додавання та відповідання на коментарі, а також оцінювання цих сутностей, то лочіним буде той факт, що зберігання, відображення та буде працювати подібно, але зберігатиме різні поля, різні значення і приховану логіку буде мати різну. Наприклад оцінка, або рейтинг, буде лише в коментарях робітника чи виконавця, а в завдання – не буде, так як це недоцільне поле для даного класу.

BaseComment – базовий клас, що є батьківським для всіх класів пов’язаних з коментарями.

- getById – метод, що відповідає за отримання інформації про коментар по його ідентифікатору;
- checkUserCommented – метод, що перевіряє чи робив юзер прямий коментар до сутності раніше;
- baseListDateFilter – метод, що генерує шматок sql-запиту на вибірку коментарів у певному часовому діапазоні;
- getAll – метод, що дозволяє отримувати коментарі пачками, тобто реалізовує безкінечну пагінацію.

JobComment – клас, що відповідає за роботу з коментарями до завдань.

EmployeeComment – клас, що відповідає за роботу з коментарями до замовників.

WorkerComment – клас, що відповідає за роботу з коментарями до виконавців завдань.

2.2 Розробка бази даних системи

Для збереження інформації у додатку використовуються MySQL, це система управління базами даних, яка базується на реляційної моделі. Файли та всі медіа-дані зберігаються на сервері, у відповідному каталозі.

База даних системи складається з двадцяти однієї таблиці:

1. migrations – таблиця, яка містить інформацію про виконані міграції до бази даних.
2. system_options – таблиця, яка містить інформацію про системні параметри, які можуть бути доступними для швидкого редагування адміністратором, наприклад - це дані про комісії і типи комісій.
3. users – таблиця, яка містить дані про користувачів.
4. password_reset_links – таблиця, яка містить інформацію про токени за якими користувачі можуть скинути паролі до своїх акаунтів.
5. notifications – таблиця, яка містить дані про нотифікації користувачів.

6. user_actions – таблиця, яка містить інформацію про незавершені дії користувача, наприклад - це відправка файлів через сокети.
7. jobs – таблиця, яка містить інформацію про задачі.
8. job_requests – таблиця, яка містить інформацію про запити на виконання задач.
9. disputes – таблиця, яка містить інформацію про суперечки до задач.
10. sockets – таблиця, яка зберігає дані про сокети користувачів.
11. get_money_requests – таблиця, що зберігає запити до адміністраторів на зняття коштів з балансів(виникає якщо під час автоматичної дії виникає проблема).
12. payment_transactions – таблиця, що зберігає дані про усі операції в результаті яких було змінено баланс.
13. server_transactions – таблиця, яка містить інформацію про всі операції з коштами, де проміжним етапом переводу коштів був акаунт власника сайту.
14. chats – таблиця, що містить дані про усі створені чати в системі.
15. chats_users – таблиця, що містить дані про зв’язки між чатами та користувачами.
16. messages – таблиця, що зберігає інформацію про створені повідомлення в чатах.
17. message_contents – таблиця, що зберігає інформацію про контенти повідомлень, дозволяє реалізовувати редагування повідомлень.
18. worker_comments – таблиця, що містить інформацію про коментарі робітникам.
19. job_comments – таблиця, що містить інформацію про коментарі задачам.
20. employee_comments – таблиця, що містить інформацію про коментарі замовникам.
21. reply_comments – таблиця, що містить інформацію про коментарі, які є відповідями на інші коментарі.

Опис кожної з таблиць бази даних наведений у таблицях 2.20 – 2.40.

Таблиця 2.20

Опис таблиці “migrations”

Поле	Тип	Призначення
id	int	Ідентифікатор
name	varchar(255)	Назва міграції
run_on	datetime	Час запуску міграції

Таблиця 2.21

Опис таблиці “user_actions”

Поле	Тип	Призначення
id	int	Ідентифікатор
user_id	int	Ідентифікатор користувача
type	text	Тип події
data	text	Додаткові дані події
key	text	Ключ події

Таблиця 2.22

Опис таблиці “jobs”

Поле	Тип	Призначення
id	int	Ідентифікатор
author_id	int	Ідентифікатор автора
title	varchar(255)	Заголовок завдання
price	double	Максимальна ціна
address	text	Адреса початку задачі
description	text	Опис завдання
lat	double	Широта точки початку
lng	double	Широта точки кінця
active	tinyint(1)	Актуальна задача
time_created	timestamp	Час створення

Таблиця 2.23

Опис таблиці “system_options”

Поле	Тип	Призначення
id	int	Ідентифікатор
name	varchar(255)	Назва параметру
value	varchar(255)	Значення параметру

Таблиця 2.24

Опис таблиці “users”

Поле	Тип	Призначення
id	int	Ідентифікатор
email	varchar(255)	Електронна пошта
password	varchar(255)	Захешований пароль
nick	varchar(255)	Нік
address	varchar(255)	Адреса виконання задач
avatar	varchar(255)	Фото профілю
lat	double	Широта користувача
lng	double	Довгота користувача
profile_verified	tinyint(1)	Перевірений користувач
admin	tinyint(1)	Адміністратор
online	tinyint(1)	Онлайн
balance	double	Баланс акаунта
activity_radius	double	Радіус підбору задач
biography	text	Коротка біографія
linkedin_url	varchar(255)	Посилання на лінкедін
instagram_url	varchar(255)	Посилання на інстаграм
phone	varchar(255)	Телефон
time_created	timestamp	Час створення
time_updated	timestamp	Час останнього оновлення

Таблиця 2.25

Опис таблиці “job_comments”

Поле	Тип	Призначення
id	int	Ідентифікатор
sender_id	int	Ідентифікатор відправника
job_id	int	Ідентифікатор завдання
body	text	Повідомлення
created_at	timestamp	Час створення коментаря

Таблиця 2.26

Опис таблиці “job_requests”

Поле	Тип	Призначення
id	int	Ідентифікатор
user_id	int	Ідентифікатор виконавця
job_id	text	Ідентифікатор залачі
price	double	Ціна за годину роботи, яку хоче виконавець
execution_time	int	Робочий час, необхідний на задачу
status	text	Статус прогресу
time_created	timestamp	Час створення
time_completed	timestamp	Час успішної здачі
time_rejected	timestamp	Час відхилення пропозиції
time_cancelled	timestamp	Час скасування виконання задачі

Таблиця 2.27

Опис таблиці “sockets”

Поле	Тип	Призначення
id	int	Ідентифікатор
user_id	int	Ідентифікатор юзера
chat_id	int	Ідентифікатор чату який переглядає користувач
socket	varchar(255)	Ключ сокету

Таблиця 2.28

Опис таблиці “disputes”

Поле	Тип	Призначення
id	int	Ідентифікатор
user_id	int	Ідентифікатор автора суперечки
job_request_id	int	Ідентифікатор запиту виконання завдання
admin_id	int	Ідентифікатор відповідального
right_user_id	int	Ідентифікатор переможця
description	text	Опис проблеми
status	varchar(255)	Статус прогресу
created_at	timestamp	Час створення
updated_at	timestamp	Час останнього оновлення
completed_at	timestamp	Час вирішення суперечки

Таблиця 2.29

Опис таблиці “password_reset_links”

Поле	Тип	Призначення
id	int	Ідентифікатор
account_id	int	Ідентифікатор юзера
reset_token	varchar(255)	Токен скидання паролю
created_at	timestamp	Час створення

Таблиця 2.30

Опис таблиці “notifications”

Поле	Тип	Призначення
id	int	Ідентифікатор
user_id	int	Ідентифікатор користувача
type	varchar(255)	Тип події, що створила нотифікацію
body	text	Основний текст нотифікації
link	varchar(255)	Посилання нотифікації
title	text	Заголовок нотифікації
created_at	timestamp	Час створення

Таблиця 2.31

Опис таблиці “chats”

Поле	Тип	Призначення
id	int	Ідентифікатор
type	varchar(255)	Тип чату
name	varchar(255)	Назва
avatar	varchar(255)	Основне фото чату

Таблиця 2.32

Опис таблиці “get_money_requests”

Поле	Тип	Призначення
id	int	Ідентифікатор
sender_id	int	Ідентифікатор користувача
user_transaction_id	int	Ідентифікатор транзакції, яка не пройшла успішно
money	double	Сума виводу
platform	varchar(255)	Ключ сокету
status	varchar(255)	Статус прогресу
body	text	Опис дії
created_at	timestamp	Час створення
done_at	timestamp	Час виконання

Таблиця 2.33

Опис таблиці “payment_transactions”

Поле	Тип	Призначення
id	int	Ідентифікатор
user_id	int	Ідентифікатор користувача
money	int	Сума зміни балансу
operation_type	varchar(255)	Тип операції
balance_change_type	varchar(255)	Тип зміни балансу
transaction_data	text	Додаткові дані
created_at	timestamp	Час початку операції
finished_at	timestamp	Час завершення операції

Таблиця 2.34

Опис таблиці “messages”

Поле	Тип	Призначення
id	int	Ідентифікатор
chat_id	int	Ідентифікатор чату
sender_id	int	Ідентифікатор відправника
type	varchar(255)	Тип повідомлення
hidden	tinyint(1)	Приховане
time_created	timestamp	Час відправки

Таблиця 2.35

Опис таблиці “chats_users”

Поле	Тип	Призначення
id	int	Ідентифікатор
chat_id	int	Ідентифікатор чату
user_id	int	Ідентифікатор користувача
last_viewed_message_id	int	Останнє переглянуте повідомлення
role	varchar(255)	Роль у чаті
typing	tinyint(1)	Друкує
time_created	timestamp	Час додання користувача до чату
delete_time	timestamp	Час видалення користувача з чату

Таблиця 2.36

Опис таблиці “message_contents”

Поле	Тип	Призначення
id	int	Ідентифікатор
message_id	int	Ідентифікатор повідомлення
content	varchar(255)	Контент повідомлення
time_edited	timestamp	Час додання вмісту

Таблиця 2.37

Опис таблиці “worker_comments”

Поле	Тип	Призначення
id	int	Ідентифікатор
sender_id	int	Ідентифікатор відправника
worker_id	int	Ідентифікатор робітника
rating	int	Оцінка
body	text	Повідомлення
created_at	timestamp	Час створення

Таблиця 2.38

Опис таблиці “server_transactions”

Поле	Тип	Призначення
id	int	Ідентифікатор
money	int	Сума зміни балансу
operation_type	varchar(255)	Тип операції
balance_change_type	varchar(255)	Тип зміни балансу
transaction_data	text	Додаткові дані
created_at	timestamp	Час початку операції

Таблиця 2.39

Опис таблиці “reply_comments”

Поле	Тип	Призначення
id	int	Ідентифікатор
sender_id	int	Ідентифікатор відправника
parent_id	int	Ідентифікатор елемента до якого дали відповідь
reply_comment_id	int	Ідентифікатор батьківского коментаря
parent_type	varchar(255)	Тип коментаря на який зробили відповідь
body	text	Повідомлення
created_at	timestamp	Час створення

Таблиця 2.40

Опис таблиці “employee_comments”

Поле	Тип	Призначення
id	int	Ідентифікатор
sender_id	int	Ідентифікатор відправника
employee_id	int	Ідентифікатор замовника
rating	int	Оцінка
body	text	Повідомлення
created_at	timestamp	Час створення

Таким чином, дана структура бази даних описує усі дані, що зберігаються в проекті.

2.3 Проектування та реалізація алгоритмів системи

Проектування та реалізація алгоритмів системи відповідають за автоматичні перевірки, обробку даних та автоматичне прийняття рішень системою, що забезпечує ефективність та надійність у виконанні завдань користувачами. Ця платформа створена для надання юзерам можливості створювати задачі, виконувати їх, отримувати винагороду та спілкуватися у вбудованому чаті. Алгоритми системи забезпечують надійне управління завданнями, безпечність фінансових транзакцій та захист платформи від потенційних загроз та шахрайств.

Діаграма активності для повної реєстрації акаунту з його авторизацією (рис. 2.5): ця діаграма відображає послідовність дій користувача для реєстрації на сайті та дозаповнення даних для верифікованості та можливості взаємодіяти з платформою. Процес реєстрації не є складним. Він складається лише з двох етапів. Самої реєстрації та дозаповнення даних. Процес розбитий на два підпроцеси, так як користувачам дуже часто потрібно спочатку розібратись з тим чи потрібна їм платформа взагалі, а вже потім виконувати певні дії для виконання операції іншого характеру, тому на діаграмі (див. рис. 2.5) можна побачити здебільшого перевірки та редіректи.

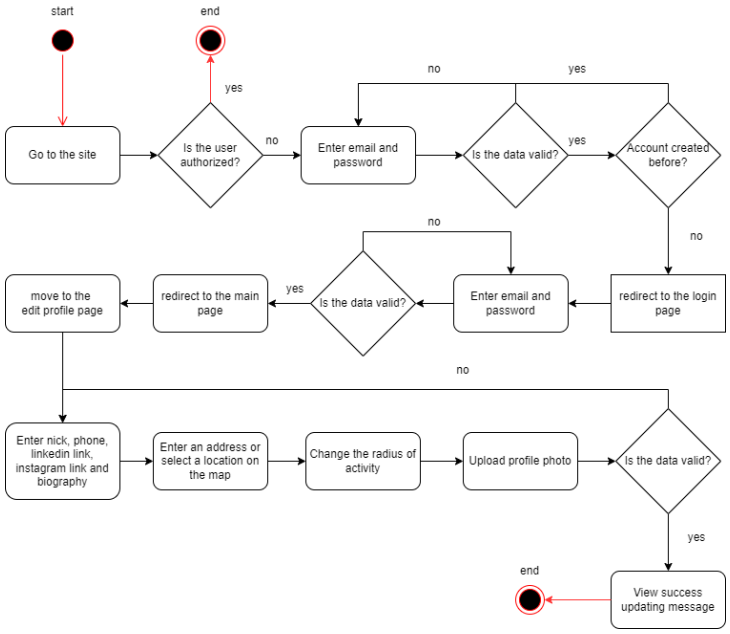

Рисунок 2.5 - Діаграма активності для повної реєстрації акаунту

Діаграма активності для створення задачі (рис. 2.6): ця діаграма відображає послідовність дій авторизованого користувача для створення нової задачі. В основному процес складаєтсья з перевірок, єдиною особливістю є відсутність модерації зі сторони адміністратора. Це є важлим моментом, так як задачі іноді потрібно вирішувати швидко, а ручна модерація є досить тривалим процесом.

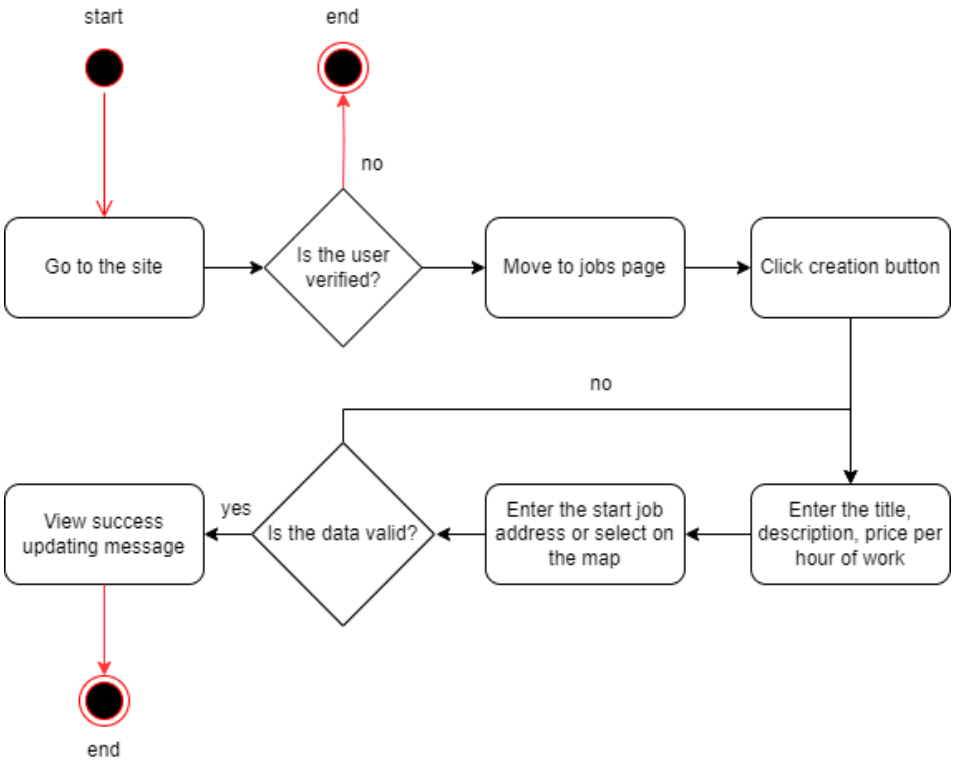

Рисунок 2.6 - Діаграма активності для створення задачі

Діаграма активності для створення пропозиції виконання задачі (рис. 2.7): ця діаграма відображає послідовність дій авторизованого користувача для створення пропозиції щодо виконання завдання. Так як платформа є простою у використанні, то користувачу достатньо просто знайти привабливе для нього завдання, після чого йому достатньо просто натиснути на кноку для відправки запиту на виконання і дочекатись відповіді від користувача.

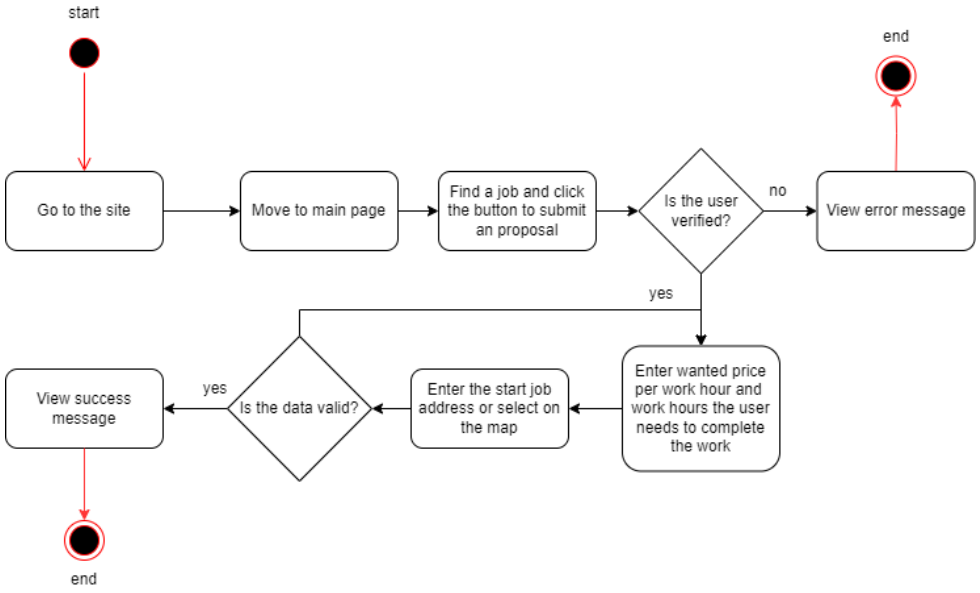

Рисунок 2.7 - Діаграма активності для створення пропозиції виконання задачі

Діаграма активності для прийняття пропозиції виконання задачі(рис. 2.8): ця діаграма відображає послідовність дій авторизованого користувача для прийняття пропозиції на виконання свого завдання. Так як процес створення запиту на виконання задачі складається лише з натискання на одну кнопку, то і процес для підтвердження виконання теж складається лише з однієї кнопки. Користувачу достатньо переглянути пропозицію, можливу скомунікувати з виконавцем і вирішити чи підходять йому умови виконання і чи симпатизує йому виконавець.

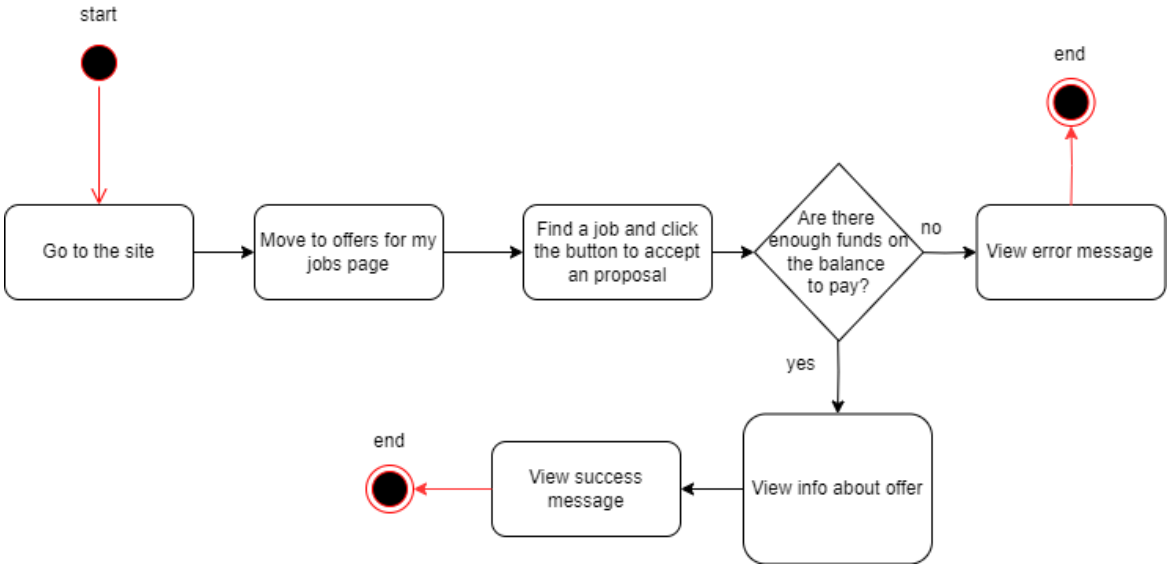

Рисунок 2.8 - Діаграма активності для прийняття пропозиції виконання задачі

Діаграма активності для зміни статусу виконання власної задачі (рис. 2.9): ця діаграма відображає послідовність дій авторизованого користувача для зміни статусу виконання власного завдання. Даний процес є важливим, бо він надає користувачам розуміння того, що саме відбувається з задачею і документно підтверджеє статус процесу. Наприклад, після створення задачі виконавець уже закінчив виконання і хоче закінчити все і отримати кошти. Для цього він не має чекати коли користувач дізнається про це, зайде на платформу і виконає оплату. Він може просто натиснути на кнопку зміну статусу для завершення, користувач побачить зміну і вже прийме, або відхилить запит.

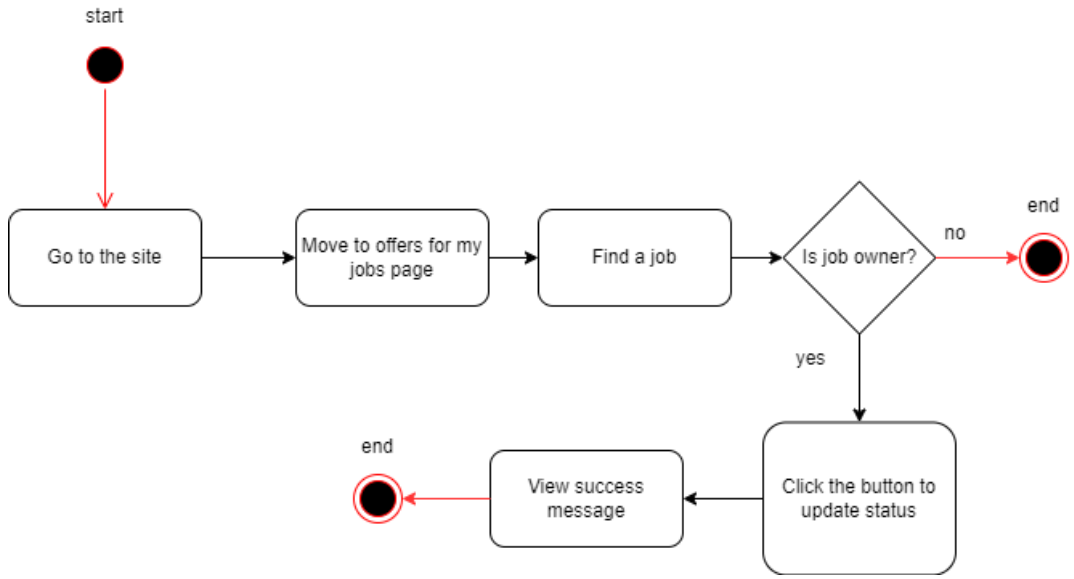

Рисунок 2.9 - Діаграма активності для зміни статусу виконання власної задачі

Діаграма активності для зміни статусу виконання задачі, виконавцем якої є цей користувач (рис. 2.10): ця діаграма відображає послідовність дій авторизованого користувача для зміни статусу виконання завдання, виконавцем якого є цей користувач. Вона включає такі кроки, як авторизація, перегляд списку завдань, вибір конкретного завдання, зміна його статусу, підтвердження та збереження змін.

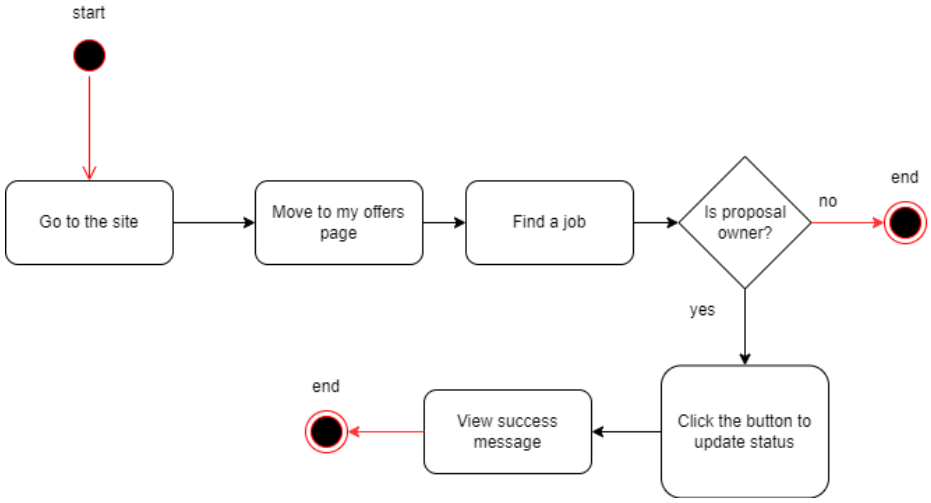

Рисунок 2.10 - Діаграма активності для зміни статусу виконання задачі, виконавцем якої є цей користувач

Діаграма активності для вирішення конфлікту (рис. 2.11): ця діаграма відображає послідовність дій адміністратора для вирішення конфлікту. Процес конфлікту технічно складається з двох етапів – створення конфлікту користувачами та його вирішення адміністратором. Решта блоків – це додаткові можливості для адміністратора, щоб краще розібратись у ситуації і прийняти правильне рішення.

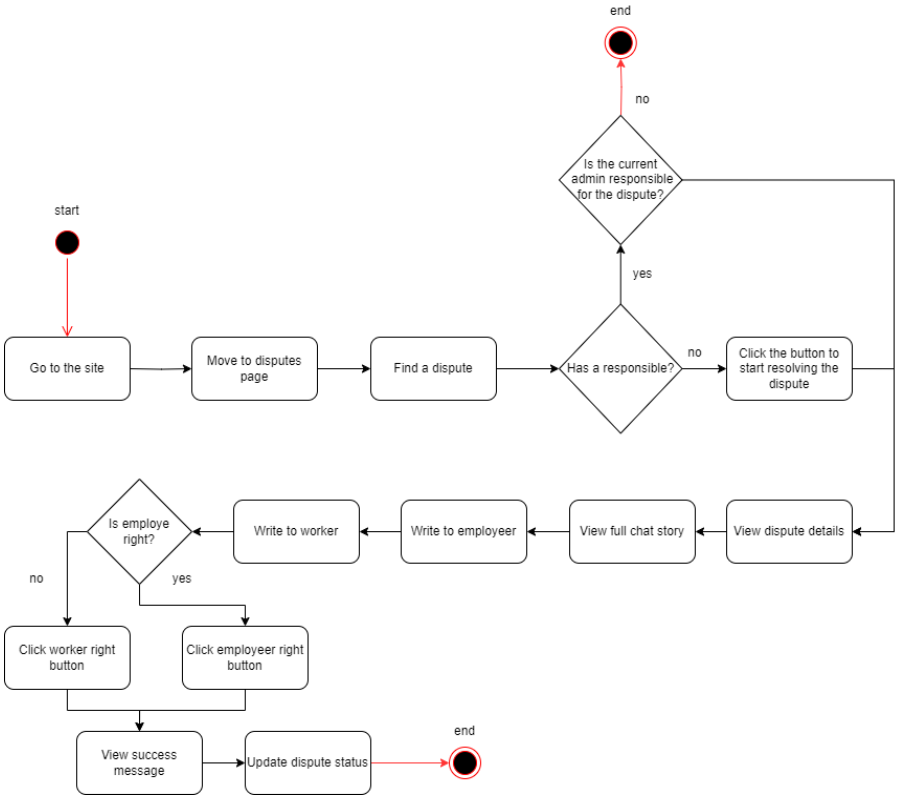

Рисунок 2.11 - Діаграма активності для вирішення конфлікту

Діаграма активності для виведення коштів з платформи (рис. 2.12): ця діаграма відображає послідовність дій верифікованого користувача для виведення коштів з платформи.

Для того, щоб вивести кошти, користувачу достатньо перейти на сторінку де є ця операція, ввести суму для виводу і підтвердити операцію, вказавши куди саме потрібно надіслати кошти. Після цього буде змінено баланс акаунту та переведено кошти на реальний фінансовий носій через деякий час.

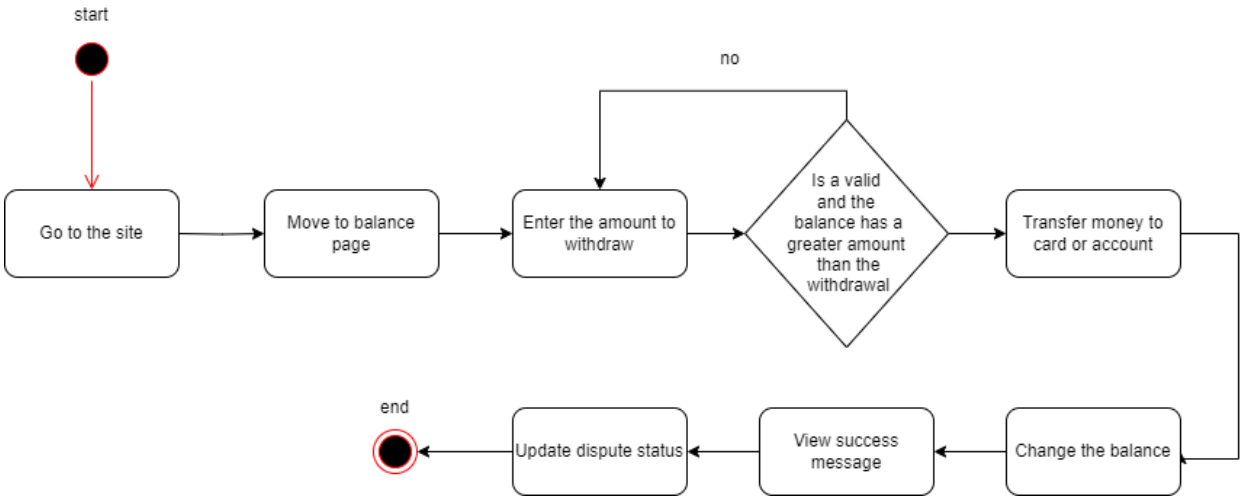

Рисунок 2.12 - Діаграма активності для виведення коштів з платформи

Згадані діаграми активності допомогли зрозуміти послідовність дій і взаємодії користувачів з системою.

Також, важливими складовими є діаграми послідовності. Вони є корисними, оскільки вони забезпечують чітке уявлення про те, як відбуваються взаємодії між користувачами і системою протягом виконання певних процесів. Вони дозволяють візуалізувати порядок виклику методів і передачу повідомлень між об'єктами, що допомагає виявити можливі помилки або неефективності в логіці роботи платформи.

Перевага використання діаграм послідовності в тому, що вони допомагають краще розуміти взаємодії між компонентами та користувачами системи. Вони забезпечують наочність та деталізацію, які важко досягти за

допомогою простого текстового опису. Тому було прийнято рішення для реалізації цих діаграм для основних процесів.

Платформа реалізована для спрощення різних процесів. Одним з таких є виконання задачі. Користувачі повинні відправляти запити на виконання задачі, після цього, замовники, підтверджують її, далі надсилається нотифікація виконавцю, після чого він приступає до виконання і змінює статус залежно від прогресу. Процес відтворено у формі діаграми послідовності (рис. 2.13).

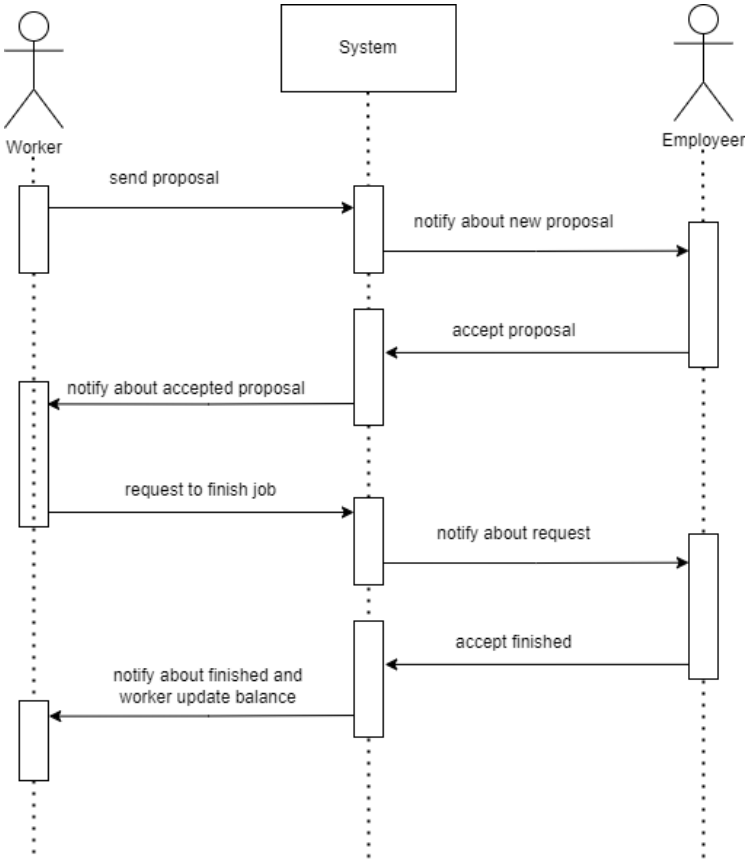

Рисунок 2.13 - Діаграма послідовності виконання задачі

Ще одним ключовим процесом платформи є вирішення суперечок. Як тільки виникає ситуація за якої користувачі не можуть дійти згоди самостійно, вони можуть закликати допомогу від адміністраторів. У такому випадку, вже адміністратор розглядає ситуацію і вирішує що робити з коштами. Візуалізовано у формі діаграми послідовності (рис. 2.14).

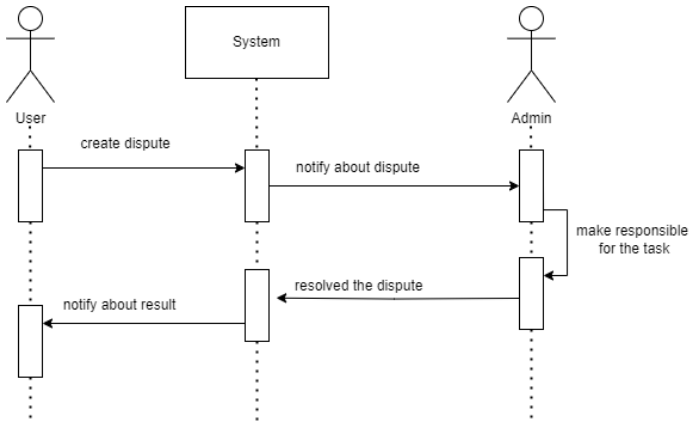

Рисунок 2.14 - Діаграма послідовності вирішення суперечок

Виведення коштів з системи є не менш важливим процесом платформи. Так як робота з коштами – це надзвичайно важливе завдання платформи, то було б безглуздим не реалізувати ручну обробку переведень у випадках, коли під час автотранзакцій виникають помилки. Тому, при виведенні коштів, користувачу вони можуть прийти не відразу, а через деякий час. Транзакція буде відображатись зі статусом у процесі доти, поки адміністратори не вирішать проблему і не виконають транзакцію вручну. Процес також візуалізовано у формі діаграми послідовності (рис. 2.15).

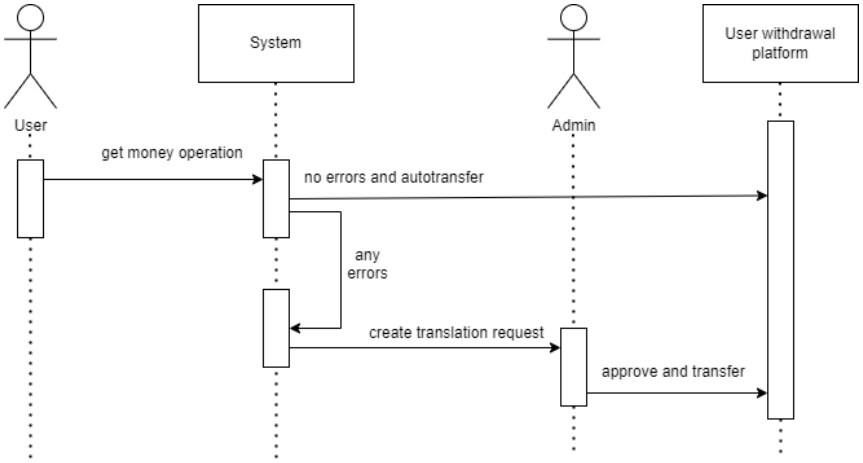

Рисунок 2.15 - Діаграма послідовності виведення коштів з системи

Діаграми послідовності допомогли з виявленням та вирішенням потенційних проблем ще до того, як вони проявилися в реальному середовищі. Вони дозволили ідентифікувати зайві або пропущені кроки у процесах, оптимізувати взаємодії і забезпечити більш ефективне використання ресурсів системи.

2.4 Реалізація платформи

Так як дана платформа скаладється з серверної та клієнтської частини, то одним з основних етапів є авторизація користувача на ній. Ключовими фрагментами є логіка самого логіну, відправки запитів на сервер та визначення відправника запиту.

Розглянемо для початку функцію, що відправляє запит авторизації користувача на сервер.

```
const handleSignIn = async () => {
    const { token, user } = await request({
      url: login.url(),
      type: login.type,
      data: { email, password, rememberMe },
      convertRes: login.convertRes,
    });

    localStorage.setItem("token", token);
    setSuccess("Logged in successfully");
    setSessionUser(user);
    redirect("/");
};
```

Як можна побачити з коду, функція використовує додаткову асинхронну функцію request, яка приймає об’єкт, що містить наступні поля: url, data та convertRes, їх розглянемо пізніше. Ці поля були отримані з об’єкту login, що зберігається у файлі requests.js папки src клієнтської частини.

Сам об’єкт має наступний вигляд:

```
login = {
  url: () => "login",
  type: "post",
  convertRes: (res) => {
    const token = res.headers.authorization.split(" ")[1];
    return { token, user: { ...res.data.user } };
  },
};
```

Як можна побачити, об’єкт має приблизний формат, що приймає метод requests. Більшість функцій, що відповідають за відправку запитів на сервер використовують подібні об’єкти. Вони зібрані в одному файлі для зручної взаємодії та швидкого коригування їх. Структура об’єкта-запиту складається з наступних полів:

- url – функції для генерації шляху запиту. Функція була гарним рішенням для генерації посилань з різними параметрами, наприклад ідентифікаторів елементів чи інформацій про параметри таблиць в адміністраторів;
- type – це тип запиту на сервер, зазвичай або post, або get;
- convertData – це функція, що приймає дані та на основі них генерує об’єкт, підходящий для відправки на сервер у тілі цього запиту;
- convertRes – це функція яка обробляє результат та повертає дані у потрібному форматі для частини, яка викликала функцію.

Щодо об’єкту login можна зауважати, що функція convertRes отримавуючи відповідь аналізує заголовок та витягує звідти авторизаційний токен, який повертається з неї разом з даними користувача. Далі, вже функція, яка викликала запит на сервер зберігає токен в локал стореджі.

Повернемось до функції request. Це одна з основних функцій клієнтської частини, так як усі запити на сервер виконуються саме за її участі. Вона отримується за допомогою хука useAjaxRequest, який лише приймає функцію для обробки помилок – onError.

```
const useAjaxRequest = ({ onError }) => {
  return async function ({
    url,
    type = "get",
    data = {},
    onSuccess = () => {},
    convertRes = () => {},
  }) {
    try {
      let res = null;

      if (type == "get") {
        res = await axios.get(`${config.API_URL}/${url}`);
      } else {
        res = await axios.post(`${config.API_URL}/${url}`, data);
      }

      const convertedData = convertRes ? convertRes(res) : res;

      onSuccess(convertedData);
      return convertedData;
```

```
    } catch (err) {
      const res = err.response;

      if (res && res.status && res.data && res.data.error) {
        onError(res.data.error);
      } else {
        onError(err.message);
      }

      throw new Error(err.message);
    }
  };
};
```

Функція, яку повертає хук слугує обгорткою для усіх запитів до серверу. Як можна побачити, у жодному рядку хуку немає витягування авторизаційного токену зі стореджу, натомість є виклик функції axios, що зберігається в допоміжних функціях додатку.

Розглянемо все, що знаходиться у файлі з допоміжною функцією axios.

```
import axios from "axios";
import mainConfig from "config";

const api = axios.create({
  baseURL: mainConfig.CLIENT_URL,
  withCredentials: true,
  headers: {
    Accept: "application/json",
  },
});

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

export default api;
```

Як можна побачити, уся магія відбувається саме тут. Спочатку створюється інстанція Axios для роботи з API, з налаштувань видно, що кожен запит, зроблений через створений екземпляр, буде мати базову URL-адресу, приймати JSON-відповіді та надсилати куки. Далі, до об’єкту додається інтерцептор для обробки запитів. Інтерцептор – це алгоритм, що

виконується перед відправкою запиту. У даному випадку він перевіряє наявність токена в локальному сховищі. Якщо токен знайдений, він додається до заголовків запиту для авторизації.

Таким чином відбувається генерування та обробка усіх HTTP-запитів від клієнта до сервера.

Розглянемо тепер логіку серверної частини. Починається вона з файлу index.js, де відбувається підключення та налаштування усіх модулів додатку – бази даних, роутингу та сокетів. Код файлу наведено у Додатку Б.

Серверна частина побудована на патерні MVP, що робить написання коду гнучким та легким у розумінні та розділенні відповідальності функцій. Наприклад запит надсилається на посилання “/login”, тому його перехоплює метод login контролера userController. Основною умовою успішного виконання цього запиту є неавторизованість браузеру з якого користувач намагається увійти на сайт. Для виконання цієї перевірки слугує мідлвар, який відпрацьовує до передачі до методу login. Мідлвар генерується за допомогою функції generateIsNotAuth, яка нічого не приймає.

```
function generateIsNotAuth() {
  return function (request, response, next) {
    const authorization = request.headers.authorization;

    if (!authorization) return next();

    const token = authorization.split(" ")[1];
    const userId = validateToken(token);

    if (userId)
      return response.status(403).json({
        message: "Forbidden",
      });

    return next();
  };
}
```

Як можна побачити, метод перевіряє наявність авторизаційного токену у запиту. За його наявності відбувається його дешефрування та отримання інформації про айдішник користувача. Якщо токен валідний і айді користувача в ньому наявний, то повертається помилка зі статусом заборони

доступу, в іншому випадку алгоритм переходить до наступної функції, яка прив’язана до посилання.

Метод login контролера користувачів використовує декоратор errorWrapper. Даний декоратор використовується у всіх методах, що приймають та обробляють дані від клієнтської частини без проміжних методів контролера. Цей декоратор перехоплює усі можливі помилки і повертає їх на бік клієнту в вигляді статусу і зрозумілого меседжу про помилку, а не кладе сервер, як це буває у випадку неочікуваних помилок.

Метод login у свою чергу считує параметри, що йому надійшли, визначає тривалість існування токену на основі наявності поля rememberMe та його значення, після цього отримує інформацію про користувача за логіном та паролем, генерує новий токен користувача і зберігає інформацію про вхід в акаунт та надсилає нотифікацію про вхід. У відповідь повертає дані про користувача та його авторизаційний токен в заголовках.

```
login = (req, res) =>
  this.errorWrapper(res, async () => {
    const { email, password, rememberMe } = req.body;
    const duration = rememberMe
      ? process.env.JWT_REMEMBER_ACCESS_LIFETIME
      : process.env.JWT_DEFAULT_ACCESS_LIFETIME;

    const user = await this.userModel.findByPasswordAndEmail(email, pass-
word);
    const userId = user.id;

    const token = jwt.sign({ userId }, process.env.SECRET_KEY, {
      expiresIn: duration,
    });

    this.createLoginNotification(userId);

    res.set("Authorization", `Bearer ${token}`);

    return this.sendResponseSuccess(
      res,
      "User authoried successfully",
      { userId },
      200
    );
  });
```

У даному методі виконано зверння до моделі userModel, а саме до її методу findByPasswordAndEmail. Цей метод виконує підключення до бази даних, де шукає користувача, а потім, перевіряє співпадіння паролю, що надійшов з клієнтської частини та захешованого паролю користувача, що зберігається в базі і на основі цього повертає дані про користувача, або генерує помилку, яка перехоплюється подібним декоратором до того, що використовується в контролері, і перекидає цю помилку далі - контролеру. Даний декоратор був дуже корисним саме на етапі розробки, так як при отриманні помилки на стороні клієнта, можна було зайти в логи файл логів та переглянути детальнішу інформацію, який метод викинув помилку і в якій моделі. У останніх версіях проекту, логування відсутнє, так як цей модуль перестав бути актуальним. Лістинг методу findByPasswordAndEmail наведено в Додатку В.

Будь-яка задача, будь-які зусилля завжди вимагають плати, тому платформа і надає можливість взаємодії з реальними коштами. Реалізувати поповнення балансу можна було кількома варіантами, обрано було найпростіший для реалізації. Суть алгоритму заключається в тому, що стороння фінансова платформа бере участь лише для поповнення балансу та виведення коштів з платформи, усі інші фінансові операції виконуються саме в додатку, що дозволило значно спростити логіку та зменшити кількість помилок в цьому етапі розробки. Система дозволяє поповнювати і виводити кошти за допомогою страйпу та пейпалу.

Так як загальний алгоритм для обох операцій подібний для обох платформ, то достатньо буде навести по одному прикладу алгоритму взаємодії з фінансовою платформою.

Для початку розглянемо алгоритм поповнення балансу через страйп. Користувач заповнює форму з картковими даними страйпу, після цього відправляється запит на оплату в страйп, а потім вже відправляє запит на сервер з токеном оплати, який згенерував страйп. Після цього сервер розуміє який користувач і на скільки доларів поповнив свій гаманець на платформі.

```
const handleSubmit = async (event) => {
  if (!amount || isNaN(Number(amount))) {
    setAmountError("Invalid field");
    return null;
  }

  const cardElement = elements.getElement(CardElement);

  const { token, error } = await stripe.createToken(cardElement);

  if (error) {
    return;
  }

  const newBalance = await request({
    url: stripeCharge.url(),
    type: stripeCharge.type,
    data: stripeCharge.convertData(amount, token),
    convertRes: stripeCharge.convertRes,
  });

  setSuccess("Operation successful");
  setSessionUser((data) => ({ ...data, balance: newBalance }));
  onComplete();
  setAmount("");
};
```

У даній функції використано об’єкт stripe. Він є результатом виклику хуку useStripe, бібліотеки @stripe/react-stripe-js, що відповідає за взаємодію зі страйпом.

Формою, що заповнює користувач є результат інтеграції бібліотеки в платформу. Якщо переглянути згенерований html, то можна побачити, що страйп вмонтовує в код платформи фрейм, що виконує операції на пряму з фінансовою платформою, без обробки та збереження інформації через розроблену систему, що забезпечує надійність та впевненість користувачів у конфіденційності інформації.

```
<Elements stripe={stripePromise}>
     <form onSubmit={handleSubmit}>
          <div className="card mb-0">
            <div className="card-body">
              <CardElement
                options={...}
              />
            </div>
          </div>
```

```
            <div className="w-100 mt-4">
              <div className="card mb-0">
                <div className="card-body">
                  <Input
                    type="text"
                    label="Money to replenishment"
                    placeholder="Enter money to replenishment"
                    value={amount}
                    onChange={(e) => amountChange(e.target.value)}
                    error={amountError}
                  />

                  <button
                    className="w-100 btn btn-primary"
                    type="submit"
                    disabled={!stripe || loading}
                  >
                    {loading ? "Loading..." : "Pay"}
                  </button>
                </div>
              </div>
            </div>
        </form>
    </Elements>
```

У даному фрагменті наведено реалізацію відображення форми для оплати. stripePromise – це об’єкт, який генерується при підключення страйпу через метод loadStripe бібліотеки @stripe/stripe-js, що приймає публічний ключ від акаунту страйпа, на який приходять кошти від користувачів при поповненні балансу. Компоненти Elements та CardElement – це елементи бібліотеки @stripe/react-stripe-js, що і відповідають за фрейм форми оплати. При виклику методу оплати, дані з фрейму считуються за допомогою elements.getElement, але прямого доступу до них немає, тому вони передаються в наступну функцію - stripe.createToken, що і проводить оплату.

Після відправки запиту на сервер відбувається оновлення форми, даних про користувача і вивід повідомлення про успішне виконання.

На сервері ж виконується метод stripeBalanceReplenishment, що перевіряє валідність токену та ціни яку заплатив користувач, для поповнення балансу на правильну суму, а потім змінює її і відправляє результат з оновленим балансом.

```
stripeBalanceReplenishment = async (req, res) =>
  this.errorWrapper(res, async () => {
    const { userId } = req.userData;
    const { amount, token } = req.body;
    const charge = await stripe.charges.create({
      amount: amount,
      currency: "usd",
      source: token.id,
      description: "Example Charge",
    });

    const newBalance = await this.userModel.addBalance(userId, amount);

    await this.paymentTransactionModel.createReplenishmentByStripe(
      userId,
      Number(amount).toFixed(2)
    );

    return this.sendResponseSuccess(res, "Balance updated success-fully",
{
      newBalance,
    });
  });
```

Метод використовує об’єкт stripe, який є результатом виклику функції stripe цієї ж бібліотеки. Цей метод приймає лише приватний ключ від страйпівського акаунта через який виконуються всі оплати. Сам же метод виконує перевірку, після чого поповнює баланс і створює новий запис у історії оплат на платформі, після чого повертає на сторону клієнта інформацію про оновлений баланс користувача.

Наступним варто розглянути логіку виводу коштів з платформи. Цього разу використаємо пейпал для прикладу. Для виконання цієї операції користувачу достатньо ввести розмір суми, яку він хоче вивести та ідентифікатор, пошту чи номер телефону акаунту пейпалу, на який прийдуть кошти, після чого натиснути відповідну кнопку і далі вся дія виконується сервером, а клієнтській частині додатку достатньо просто оновити дані та вивести повідомлення про успішну операцію. Так як було сказано раніше і будь-яке виконання завдання має свою ціну, то вивід коштів теж займає якийсь відсоток суми. Користувача попереджають про це при виведенні коштів з платформи.

Таким чином, спочатку метод на сервері вираховує комісію, яку потрібно відняти від суми, що знімає користувач, далі списує вказану користувачем суму після чого виводить гроші з урахуванням комісії з акаунту пейпалу власника платформи і переводить їх до користувача. Якщо під час операції виникає якась помилка, то вона фіксується і створює запит на вирішення проблеми вручну. За це вже відповідають адміністратори і вирішують запит, або вручну, після поповнення балансу акаунта власника(якщо виникла проблема саме з надлишком коштів), або змінюючи дані пейпалу отримувача і повторної спроби виконати операцію. Лістинг коду наведено у Додатку Г.

Так як створення задачі є простою операцією і подібною до дозаповнення профілю користувача, то наступним розглянутим фрагментом коду будуть етапи виконання задачі.

Першим кроком є відправка запиту на виконання роботи. Користувач заповнює форму де вказує необхідні дані та після натискання кнопки відправляє їх на сервер, який зберігає пропозицію до задачі та сповіщає замовника про неї.

```
create = async (req, res) =>
  this.errorWrapper(res, async () => {
    const { jobId, price, time } = req.body;

    const userId = req.userData.userId;
    const createdRequest = await this.jobProposalModel.create(
      jobId, userId, price, time
    );

    const user = await this.userModel.getFullUserInfo(userId);
    const job = await this.jobModel.getById(jobId);

    this.sentProposalNotification({...});

    return this.sendResponseSuccess(res, "Request created success-fully",

   {requestId: createdRequest.id}
);
});
```

Наступною дією є підтвердження чи відхилення пропозиції замовником. Для цього використовується метод accept контролера jobProposalController, який у свою чергу використовує базовий метод __changeStatus.

```
__changeStatus = async (req, res, validationType, validationCallback) =>
  this.errorWrapper(res, async () => {
    const { proposalId } = req.body;

    const proposal = await this.jobProposalModel.getById(proposalId);
    const jobId = proposal.jobId;

    const userId = req.userData.userId;
    const validation =
      validationType == "job-owner"
        ? this.__jobOwnerCheck
        : this.__proposalOwnerCheck;
    const resValidation = await validation(res, proposalId, jobId,
userId);

    if (resValidation) return resValidation;
    return await validationCallback(proposalId);
  });
```

Метод приймає тип доступу, який повинен мати користувач до задачі. Якщо це власник задачі, тоді тільки він і зможе виконати запит, що надійшов, робітник же – не зможе нічого зробити, щоб якось змінити статус задачі на бажаний. Після валідації викликається уже функція, яка юула передана як колбек.

Метод accept передає функцію, яка обчислює суму, яку потрібно зарезервувати у користувача, та віднімає її з балансу, у випадку нестачі – повідомляє його про це. Після цього змінює статус пропозиції та повідомляє виконавця про це. Після повертає клієнту інформацію про оновлений баланс та офер.

```
accept = async (req, res) =>
  this.__changeStatus(req, res, "job-owner", async (proposalId) => {
    ...

    const newBalance = await this.userModel.rejectBalance(
      jobAuthorId,
      Number(pricePerHour * hours).toFixed(2)
    );
```

```
        await this.jobProposalModel.accept(proposalId);
        const performance = await
    this.userModel.getFullUserInfo(proposal.userId);

        ...

        return this.sendResponseSuccess(res, "Proposal accepted success", {
          proposal,
          newUserBalance: newBalance,
        });
    });
```

У свою чергу reject просто змінює статус та дає нотифікацію користувачу, що відправив пропозицію.

```
    reject = async (req, res) =>
      this.__changeStatus(req, res, "job-owner", async (proposalId) => {
        const proposal = await this.jobProposalModel.reject(proposalId);

        const userId = req.userData.userId;
        const user = await this.userModel.getFullUserInfo(userId);

        this.rejectJobProposal(
          {
            proposalId: proposal.id,
            senderNick: user.nick,
            senderEmail: user.email,
            jobTitle: proposal.title,
          },
          proposal.userId
        );

        return this.sendResponseSuccess(res, "Proposal rejected success", {
          proposal,
        });
    });
```

Платформа надає ще чотири методи для зміни статусів. Для замовника –requestToCancel та acceptCancelled, а для виконавця – requestToComplete та acceptCompleted. Користувачам достатньо просто натискати кнопки і підтверджувати дії, а сервреpна частина буде виконувати методи подібні до reject для зміни статусів та оповіщень.

На останок варто розглянути методи роботи чату. Відправка повідомлень є головною задачею будь-якого чату, тому фокус буде саме на цій операції.

Самим простим варіантом є звичайна відправка повідомлення, де клієнт просто вводить текст та різні смайлики. Для текстового поля було вибрано елемент div з атрибутом contentEditable, що дозволяє змінювати внутрішній контент блоку. Перевагою такого підходу є простота візуального налаштування блоку, адже звичні інпути та текстеріа не завжди можна налаштувати так, як потрібно.

Обирати емоції користувач може після натискання смайлику на панелі, що відповідає за відправку повідомлень, після чого з'являється попап з можливістю вибору символу, який потрібно додати до текстового поля.

Так як для вставки символу на місце курсору потрібні були додаткові операції, то функція вставки виглядає трохи складнішою, ніж просто додання тексту в innerHTML блоку.

```
const handleEmojiClick = (emoji) => {
  const { startContainer, startOffset, endContainer, endOffset } =
    savedSelection;

  const range = document.createRange();
  range.setStart(startContainer, startOffset);
  range.setEnd(endContainer, endOffset);
  range.deleteContents();

  const textNode = document.createTextNode(emoji);
  range.insertNode(textNode);

  range.setStartAfter(textNode);
  range.setEndAfter(textNode);

  const selection = window.getSelection();
  selection.removeAllRanges();
  selection.addRange(range);

  if (selection.rangeCount) {
    const range = selection.getRangeAt(0);
    setSavedSelection({
      startContainer: range.startContainer,
      startOffset: range.startOffset,
      endContainer: range.endContainer,
      endOffset: range.endOffset,
    });
  }
};
```

Спочатку відбувається перевірка наявності курсору або виділеного тексту в полі. Якщо збережений курсор або виділення існує, витягуються межі (початкова і кінцева позиції) збереженого вибору. Перевіряється, чи знаходяться елементи збереженого вибору в документі. Якщо вони відсутні (наприклад, видалені), емодзі також додається в кінець текстового поля. Створюється новий об'єкт Range (діапазон) і встановлюються його межі відповідно до збереженого вибору. Видаляється вміст, що знаходиться у межах збереженого вибору. Створюється новий текстовий вузол з емодзі і вставляється на місце видаленого вмісту. Оновлюються межі діапазону так, щоб вони були після вставленого емодзі. Поточний вибір на сторінці оновлюється, очищується і встановлюється новий вибір з оновленими межами. Збережений вибір оновлюється новими межами, щоб у майбутньому знову можна було вставляти емодзі у правильне місце.

Так як повідомлення зберігається у форматі блоків, то видалення зайвих пробільних теж має вигляд складніший.

```
const handleSendClick = () => {
  let messageContent = textRef.current.innerHTML;
  let prevLength = 0;

  do {
    prevLength = messageContent.length;
    messageContent = messageContent
      .trim()
      .replace(/^(\s*<div>(?: |<br>|<br\/>|\s)*<\/div>\s*)*/, "")
      .replace(/(\s*<div>(?: |<br>|<br\/>|\s)*<\/div>\s*)*$/, "")
      .replace(/^(\s* \s*)*/, "")
      .replace(/(\s* \s*)*$/, "")
      .trim();
  } while (prevLength > messageContent.length);

  if (!messageContent.length) {
    return;
  }

  handleSendTextMessage(messageContent);
  textRef.current.innerHTML = "";
};
```

Спочатку текстове повідомлення видаляє крайні секції з пробілами, або переходами на новий рядок. Потім видаляє пробільні частини, що

залишились, або переходи на нові рядки. Після цього очищення метод визначає чи можна відправляти повідомлення в чат і очищає текстове поле.

Функція handleSendTextMessage, що використана в самому кінці функції генерується в батьківському компоненті, а саме на сторінці Chat і передається за допомогою контексту до потрібного компонента.

```
const handleSendTextMessage = (message) => {
  if (editMessageId) {
    if (message != editMessageContent) editMessage(editMessageId, mes-
sage);
    unsetEditMessage();
  } else {
    const dop = { tempKey: randomString() };

    if (activeChat.chatType == "personal") {
      dop["chatId"] = activeChat?.chatId;
      dop["getterId"] = activeChat.userId;
    }

    sendMessage(activeChat.chatId, "text", message, activeChat.chatType,
dop);
  }
};
```

У цій функції визначається чи потрібно відредагувати повідомлення чи відправити нове. У випадку нового – створюється додатковий локальний ключ, за яким статус повідомлення буде змінено з “У відправці” на “Відправлено”. Далі викликається функція sendMessage, яка генерується у результаті виклику хука useChatInit. Цей хук працює з сокетами і відповідає за отримання та відправку повідомлень на сервер. При ініціалізації він приймає об’єкт іо необхідний для відправки та отримання повідомлень та різні функції, що мають відпрацьовувати при приході певних повідомлень від серверної частини.

```
const sendMessage = (chatId, typeMessage, content, chatType, dop) => {
  const dataToSend = {
    chatId,
    typeMessage,
    content,
    chatType,
    ...dop,
  };
```

```
    const dataToInsert = {
      chatId: chatId,
      type: typeMessage,
      content,
      chatType,
      userId: sessionUser.id,
      inProcess: true,
      timeSended: new Date().toISOString(),
      ...dop,
    };

    io.emit("send-message", dataToSend);
    onGetMessageForSockets(dataToInsert);
  };
```

Після відправки повідомлення користувачу домальовується тимчасове відображення відправки до тих пір, поки користувач не отримає відповідь від серверу про успішне надсилання меседжу.

Для отримки повідомлень кастомний хук useChatInit використовує хук useEffect в якому прив'язує реакції на повідомлення. Код хука useEffect наведено у Додатку Ґ.

На серверні частині додатку, подія send-message перехоплюється та оброблюється функцією onSendMessage. Прив'язка відбувається однією з функцій вищого порядку bindFuncToEvent. Її особливість заключається у використанні у всіх подіях сокету для спрощення коду.

```
const bindFuncToEvent = (event, func) => {
  socket.on(event, async (data) => {
    try {
      await func(data, {
        socket,
        userId,
        user,
      });
    } catch (e) {
      sendError(e.message);
    }
  });
};
```

Як можна побачити, функція не є складною, вона приймає назву події та функцію, яка має відпрацювати на подію, огортає в блок try catch і при помилці відправляє інформацію про це.

Метод onSendMessage викликає метод стоворення повідомлення у контролера чату. Після успішного створення робить розсилку залежно від типу чату. Повідомлення отримують усі користувачі за виключенням відправника. Користувач, що відправив повідомлення отримує інший тип події, про успішну відправку, яка на стороні клієнта замінить йому статус повідомлення з “У відправці” на “Відправлено”. Лістинг методу onSendMessage наведено у Додатку Ґ, а лістинг метолу createMessage контролера chatController у Додатку Д.

Ще одним варіантом відправки повідомлення є відправка файлу, або запис відео чи аудіо повідомлення. Варіант є цікавим, так як розміри файлів можуть бути дуже великими і передавати разово ці дані від клієнта до сервера може бути проблематично, тому, було розроблено алгоритм, який розбиває файли на масив блобів, які передаються поступово на сервер. Загалом, клієнтська частина зберігає меседж з масивом блобів, та згенерованим випадково ключем, за яким буде визначено яке саме повідомлення треба оновити і відправити наступний шматок на сервер, або ж повністю завершити. Також, цей ключ є важливим для операції скасування відправки, так як за його допомогою можна перервати та видалити більше непотрібний файл.

Функція, що відповідає за розбиття файлів та генерації масиву відправки:

```
async function createMediaActions(data, dataType, filetype, dop) {
  const tempFileKey = randomString();
  let arr = null;
  if (dataType == "media")
    arr = await splitBlob(data, config["BLOB_CHUNK_SIZE"], data.type);
  else if (dataType == "notmedia")
    arr = splitDataIntoChunks(data, config["BLOB_CHUNK_SIZE"], data.type);
  mediaActionsRef.current[tempFileKey] = {
    data: arr,
    percent: 0,
    inQueue: [...arr],
    filetype,
    dop: { ...dop },
  };
```

```
  const blobToSend = mediaActionsRef.current[tempFileKey]["inQueue"][0];
  const last = mediaActionsRef.current[tempFileKey]["inQueue"].length ==
1;

  return {
    tempKey: tempFileKey,
    type: filetype,
    data: blobToSend,
    last,
    ...dop,
  };
}
```

Як було сказано раніше, функція приймає дані, їх тип (медіа чи щось інше), розширення файлу таякісь додаткові дані, якщо вони потрібні при відправці. Також всередині генерується тимчасовий ключ, про який було згадано раніше.

Розбиття файлу на масив блобів відбувається в методі, що відповідає за відправку даних на сервер. Відразу після розбиття відбувається відправка першого шматку файлу. Після цього, система починає очікувати на підтвердження збереження і передачу наступного шматка, якщо відправка не була завершена.

```
const sendMedia = async (data, dataType, filetype, dop, filename) => {
  const dataToSend = await createMediaActions(data, dataType, filetype,
dop);
  const messageType = indicateMediaTypeByExtension(filetype);
  const content = messageType == "file" ? filename : data;
  const dataToInsert = {
    chatId: dataToSend["chatId"],
    type: messageType,
    content: content,
    chatType: dataToSend["chatType"],
    userId: sessionUser.id,
    inProcess: true,
    timeSended: new Date().toISOString(),
    tempKey: dataToSend["tempKey"],
  };

  onGetMessageForSockets(dataToInsert);
  io.emit("file-part-upload", { ...dataToSend });
};
```

Код, що відповідає за відправку наступного фрагменту файлового повідомлення:

```
io.on("file-part-uploaded", async ({ tempKey, message = null }) => {
  const nextPartData = await onSuccessSendBlobPart(tempKey);

  if (!nextPartData) return;
  if (nextPartData == "success saved" && message) {
    onGetMessageForSockets(message);
    return;
  }

  onUpdateMessagePercent({ tempKey, percent: nextPartData["percent"] });
  setTimeout(() => io.emit("file-part-upload", { ...nextPartData }), 1);
});
```

Щодо серверної частини, то метод, що перехоплює інформацію про новий файл, просто викликає метод __uploadToFile для оновлення даних контролера чату визначає який результат повернути користувачу і кому відправити інформацію про зміни. Лістинг методу наведено у Додатку Е.

```
__uploadToFile = async (userId, key, data, type) => {
  const info = await this.actionModel.getByKeyAndType(
    userId,
    key,
    "sending_file"
  );
  let filename = randomString() + "." + type;

  if (!info || !info.data) {
    this.__createFolderIfNotExists(this.__message_folder);
    fs.writeFileSync(this.__message_folder + "/" + filename, data);
    const actionInfo = JSON.stringify({
      filename,
    });
    await this.actionModel.create(userId, "sending_file", key, actionIn-
fo);
  } else {
    const resParsed = JSON.parse(info.data);
    filename = resParsed.filename;
    fs.appendFileSync(this.__message_folder + "/" + filename, data);
  }

  return filename;
};
```

Метод отримує дані про відправлений файл користувачем, якщо раніше такої дії не було, то створюється новий файл з даними що надійшли, якщо була, то дописується до того файлу шматок даних, після чого повертає назву згенерованого файлу.

Записані з платформи відео та аудіо повідомлення мають формат медіа-файлів, тому логіка з їх відправкою на сервер та розбиттям на блоби – однакова. Цікавим є сам запис повідомлення.

Починається метод з виклику функції handleStartRecording, яка отримується від хука useRecorder, що отримує лише один метод, в який відбуваться передача згенерованого файлу.

```
const handleStartRecording = async () => {
  const name = randomString();

  const afterRecording = (blob) => {
    const src = window.URL.createObjectURL(blob);
    const file = { src };
    file["type"] = recordingMediaType === "audio" ? "mp3" : "mp4";
    file["name"] = name + "." + file["type"];
    //close();
    setFile(file);
  };

  const stopper = await startRecording(
    recordingMediaType,
    () => {
      setRecording(true);
      timerRef.current = setInterval(
        () => setRecordingTime((prev) => (prev += 100)),
        100
      );
      setRecorder(() => stopper);
    },
    afterRecording
  );
};
```

Всередині handleStartRecording викликає startRecording, що повертає функцію, яка виконуватиметься при натисканні кнопки зупинки запису корисутвачем. Функція startRecording отримує тип медіа запису, функцію, що має запуститись після початку запису та функцію, що має відпрацювати після кінця запису.

Для початку функція ініціалізує об’єкт constraints, цей об’єкт використовується потім в якості параметра для налаштувань доступів програми. Таким чином, якщо тип запису це відео, то програма запитує ще доступ до відео, а не лише до мікрофона. За отримання медіа-потоку

відповідає функція getMedia яка просто звертається до методу navigator.mediaDevices.getUserMedia і повертає його результат. Після цього створюється об'єкт MediaRecorder з отриманим медіа-потоком для запису та починається запис за допомогою методу mediaRecorder.start(). Після початку відроазу ж викликається передана параметром функція afterStartRecording. Далі встановлюється обробник події dataavailable для об'єкта mediaRecorder і кожного разу, коли стає доступним новий шматок даних, масив voice очищується і додаються нові дані, потім встановлюється обробник події stop для об'єкта mediaRecorder. Коли запис зупиняється, то спочатку зупиняються всі треки медіа-потоку, потім створюється об'єкт Blob з типом audio/mp3, якщо записувалося тільки аудіо, або video/mp4, якщо записувалося відео, далі отриманий об'єкт Blob зберігається у змінну resBlob. Пілся цього викликається функція afterRecording з переданим об'єктом resBlob, щоб обробити записані дані. Створюється функція stopper, яка зупиняє запис, викликаючи метод mediaRecorder.stop(), і встановлює mediaRecorder в null. Повертається функція stopper, щоб її можна було викликати для завершення запису і відображення попапу з відео чи аудіо для підтвердження відправки.

Таким чином було реалізовано основні найзаплутаніші фрагменти коду платформи для виконання завдання за нагороду.

Висновки до другого розділу

Під час розробки платформи для виконання задач за винагороду було проаналізовано всі можливі сценарії використання платформи з урахуванням вимог програмного забезпечення. На основі проведеного аналізу було спроектовано структуру платформи, яка включає в себе моделювання взаємодії між усіма її складовими частинами.

Особливу увагу було приділено проектуванню бази даних, яка служить для надійного зберігання інформації про користувачів, завдання, транзакції та інші важливі дані. Це забезпечує стабільну роботу платформи та швидкий

доступ до необхідної інформації для всіх її компонентів. У процесі проектування також були детально описані алгоритми роботи додатку, що включають послідовні кроки використання функцій платформи.

Основний функціонал додатку передбачає створення завдань користувачами, їх розміщення на платформі для виконання іншими користувачами, обговорення умов завдання та вирішення можливих конфліктів. Додатково розглянуто основні функції авторизації, взаємодії користувачів із завданнями, поповненням балансу та основні можливості чату.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ПЛАТФОРМОЮ

3.1 Порядок встановлення та налаштування параметрів платформи

Дана платформа складається з двох основних компонентів – серверної частини та клієнтської. Серверна частина відповідає за обробку запитів, управління базою даних та розміщення файлового сховища, тоді як клієнтська частина забезпечує інтерфейс користувача та взаємодію з сервером.

Серверна частина включає не лише серверний код, але й файлове сховище та базу даних, для зменшення обсягу роботи. Це дозволяє централізувати управління всіма основними ресурсами в одному місці.

Таким чином, для розгортання платформи буде достатньо двох серверів. Один сервер буде обслуговувати фронтенд частину, яка розроблена на основі JavaScript-бібліотеки React. Цей сервер забезпечуватиме доступ користувачів до інтерфейсу. Другий сервер буде відповідати за бекенд частину, яка розроблена на платформі Node.js з використанням фреймворку Express. Цей сервер оброблятиме всі запити від клієнтів, управлятиме базою даних та забезпечуватиме доступ до файлового сховища.

Взаємодія між клієнтською та серверною частинами виконується через HTTPS-запити та сокети. HTTPS-запити використовуються для стандартних операцій, таких як отримання даних з бази або відправлення даних на сервер. Використання HTTPS забезпечує захищене з'єднання, що гарантує безпеку переданих даних. Сокети використовуються для реалізації функцій, що вимагають реального часу, таких як миттєві повідомлення або оновлення даних у РРЧ. Завдяки цьому платформа може оперативно реагувати на зміни та забезпечувати більш інтерактивний досвід для користувачів.

Такий підхід дозволяє спростити структуру та забезпечити ефективну взаємодію між компонентами платформи, а також забезпечити гнучкість системи при додаванні нових функцій.

Для візуального відображення архітектури платформи було створено діаграму розгортанн (рис. 3.1).

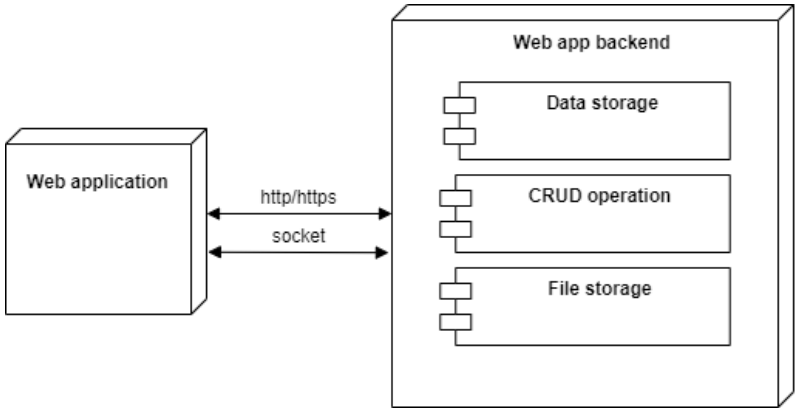

Рисунок 3.1 - Діаграма розгортання платформи

Налаштування параметрів платформи включає заповнення файлу .env для бекенд частини та файлу src/config.js для фронтенд-компонента.

До переліку змінних бекендового .env-файлу належать:

- PORT – порт, на якому буде розгорнуто бекенд частину проекту;
- CLIENT_URL – посилання фронтенд-частини додатку, з якої будуть надсилатись API-запити користувача;
- DB_HOST – хост, за яким сервер буде підключатись до бази;
- DB_USER – користувач, від імені якого, сервер буде підключатись до бази;
- DB_PASSWORD – пароль користувача, від імені якого, сервер буде підключатись до бази;
- DB_DATABASE – ім’я бази даних, до якої сервер буде підключатись;
- DB_CHARSET – тип кодування у базі даних;
- SECRET_KEY – випадковий ключ, що використовується для хешування паролів;
- DEFAULT_AJAX_COUNT_USERS_TO_CHATTING – кількість користувачів, що довантажуються при пошуку в списку чатів;
- DEFAULT_AJAX_COUNT_CHAT_MESSAGES – кількість повідомлень, що довантажуються при пошуку в чаті;

- MAIL_SERVICE – назва поштового сервісу або провайдера, який ви використовуєте для надсилання електронної поштиж
- MAIL_EMAIL – електронна адреса, яка використовується як адреса відправника електронної пошти
- MAIL_FROM – ім'я або псевдонім, який використовується як відправника електронної пошти.
- MAIL_PASSWORD – пароль для облікового запису електронної пошти.
- JWT_REMEMBER_ACCESS_LIFETIME – тривалість життя токену для користувача, що вибрав “remember me” при авторизації;
- JWT_DEFAULT_ACCESS_LIFETIME – тривалість життя токену для користувача, що не вибрав “remember me” при авторизації;
- STRIPE_PUBLIC_KEY – публічний ключ Stripe, використовується для ідентифікації облікового запису на Stripe під час обробки платежів;
- STRIPE_PRIVATE_KEY – приватний ключ Stripe, використовується для автентифікації сервера на Stripe під час створення платежів;
- PAYPAL_CLIENT_ID – ідентифікатор клієнта PayPal, використовується для ідентифікації вашого клієнта або додатку під час взаємодії з API PayPal;
- PAYPAL_SECRET_KEY – секретний ключ PayPal, використовується для забезпечення безпеки та автентифікації на серверній стороні під час взаємодії з API PayPal.

До переліку змінних конфігураційного файлу фронтенду належать:

- API_URL – URL адреса, за якою здійснюється доступ до API;
- CLIENT_URL – це URL адреса, фронтенд-частини платформи;
- BLOB_CHUNK_SIZE – розмір частини BLOB-файлу, що може передаватись сокетами на сервер;
- UNBLOB_CHUNK_SIZE – розмір частини не BLOB, що може передаватись сокетами на сервер;
- JOB_STATUSES – об’єкт, що зберігає дані про можливі статуси прогресу задачі;

- COMMENT_TYPES – об’єкт, що зберігає дані про можливі типи коментарів у додатку;
- CHAT_ROLES – об’єкт, що зберігає дані про можливі типи ролей у чаті додатку;
- CHAT_OWNER_ROLES_SELECT – об’єкт, що зберігає дані про можливі типи ролей, що можуть задавати власники чатів іншим корисутвачам;
- CHAT_ADMIN_ROLES_SELECT – об’єкт, що зберігає дані про можливі типи ролей, що можуть задавати адміни чатів іншим корисутвачам;
- MAP_KEY – ключ, що відповідає за роботу з АПІ гугл-карт;
- STRIPE_PUBLIC_KEY – публічний ключ Stripe, використовується для ідентифікації облікового запису на Stripe під час обробки платежів;
- PAYPAL_CLIENT_ID – ідентифікатор клієнта PayPal, використовується для ідентифікації вашого клієнта або додатку під час взаємодії з API PayPal.

Таким чином було розглянуто розгортання та налаштування компонентів платформи.

3.2 Структура інтерфейсу платформи

Тепер розглянемо інтерфейс і функціональні можливості платформи.

Йог обуло реалізовано на основі статтей щодо візуалу, правильного UX та загальних норм розробки та вигляду сайтів [27, 28, 29, 30]. На скріншотах буде показано, як користувачі можуть створювати завдання, брати їх до роботу, контролювати прогрес, комунікувати та інше. Детальні описи до кожного скріншота підкреслять основні особливості інтерфейсу.

Для початку розглянемо основу будь-якої платформи, а саме форму для авторизації. Її продемонстровано у вигляді скріншоту (рис. 3.2).

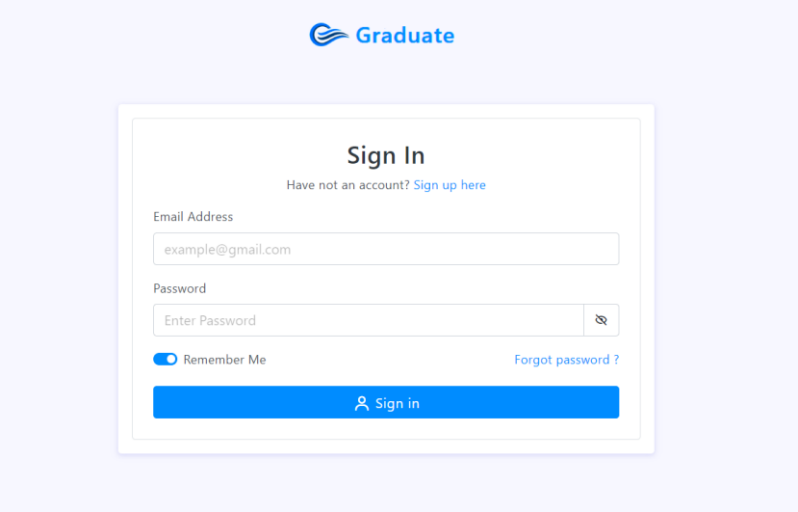

Рисунок 3.2 - Форма авторизації на сайті

Дана сторінка є реалізацією звичайної форми авторизації. Користувач вводить свій логін та пароль, і якщо дані співпадають, то користувач отримує доступ до платформи. Тривалість авторизації залежить від того, чи активовано опцію запам’ятання, чи ні. Також форма має можливості перемикання на форму для реєстрації, призначену для нових користувачів, та форму скидання паролю, призначену для давніх користувачів, що не можуть згадати свій пароль.

Після авторизації користувач бачить задачі відповідно до його збереженого розташування в налашотуваннях профілю. Демонстрацію наведено у вигляді скріншоту (рис. 3.3).

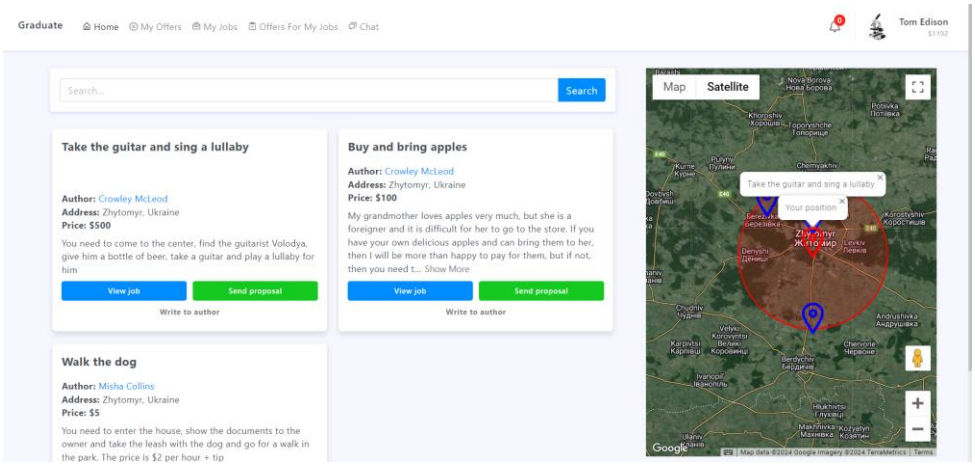

Рисунок 3.3 - Основна сторінка користувача

На сторінці користувач бачить задачі, які він може виконати. Відображаються вони у порядку найближчих до користувача. Червоним кругом позначено ділянку як найпріоритетнішу для користувача. Для перевірки локації задачі, достатньо натиснути на позначкуі побачити її назву. Далі звіритись з картками і переглянути умови. Після того, як користувач розуміє, що задача йому підходить, він може написати власнику щодо додаткових умов чи питань у внутрішньому чаті системи. Демонстрацію візуалу чату наведено у вигляді скріншоту (рис. 3.4).

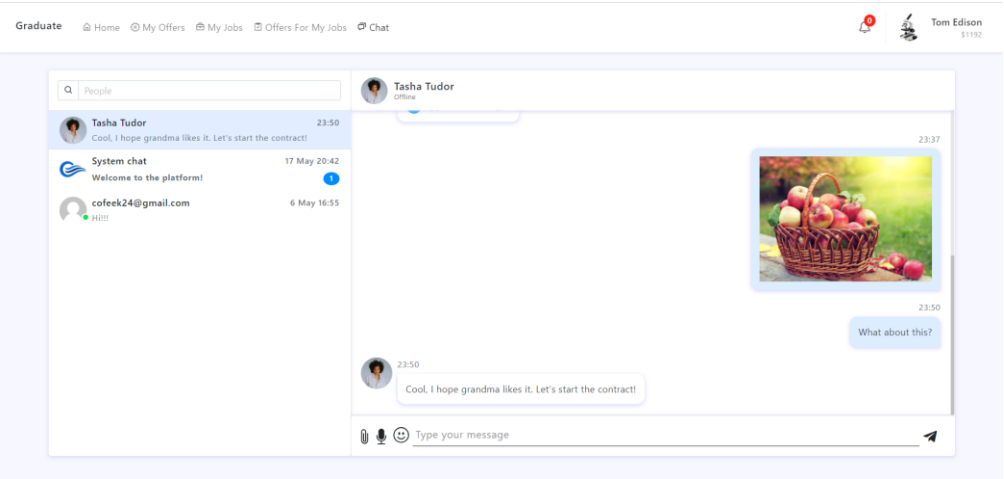

Рисунок 3.4 - Внутрішній чат платформи

Як можна побачити, чат має дві секції. Список з чатами користувача та список повідомлень чату, що переглядається в даний момент. У користувача є чотири варіанти відправки повідомлень - текстово, файлово, або записавши відео чи аудіо-повідомлення. Приклади використання зображені на рисунках 3.5 – 3.8.

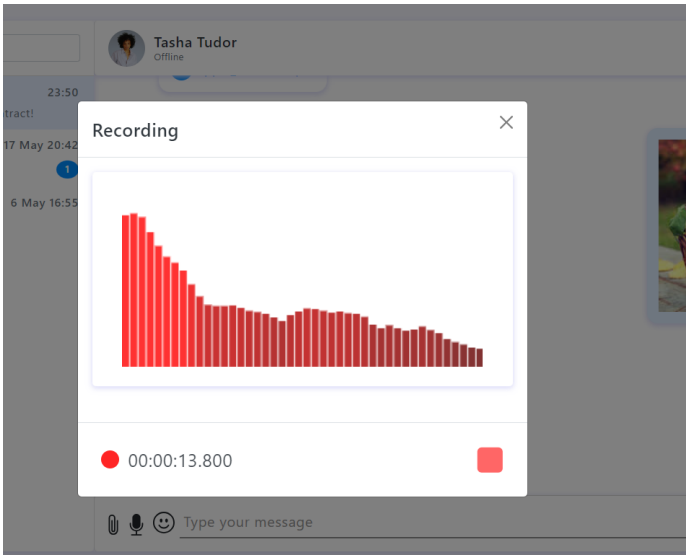

Рисунок 3.5 - Запис аудіо-повідомлення

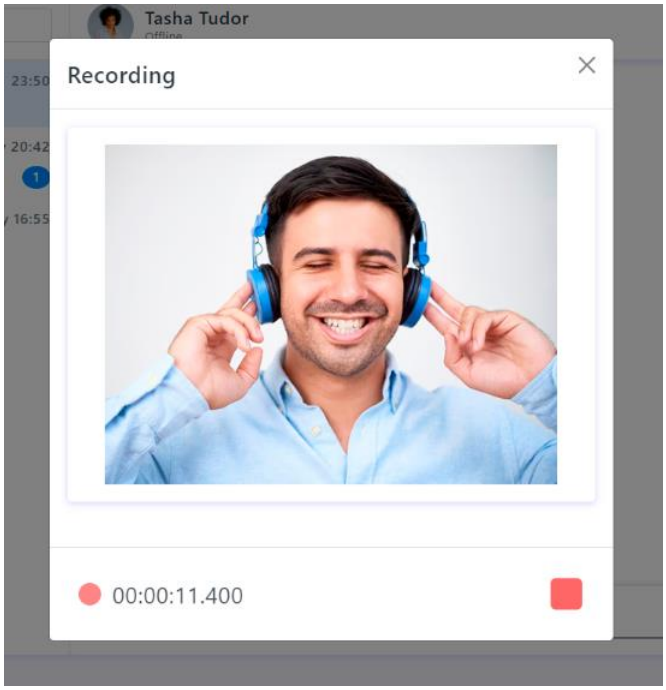

Рисунок 3.6 - Запис відео-повідомлення

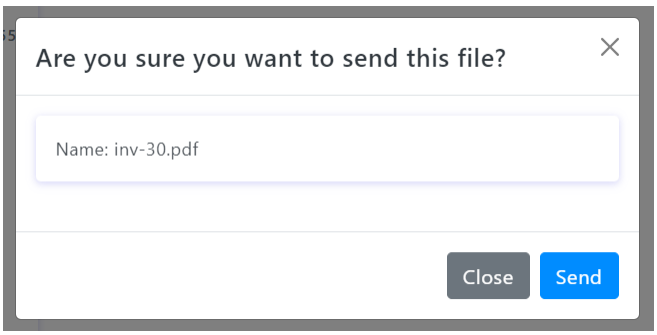

Рисунок 3.7 - Запит на підтвердження відправки файлу

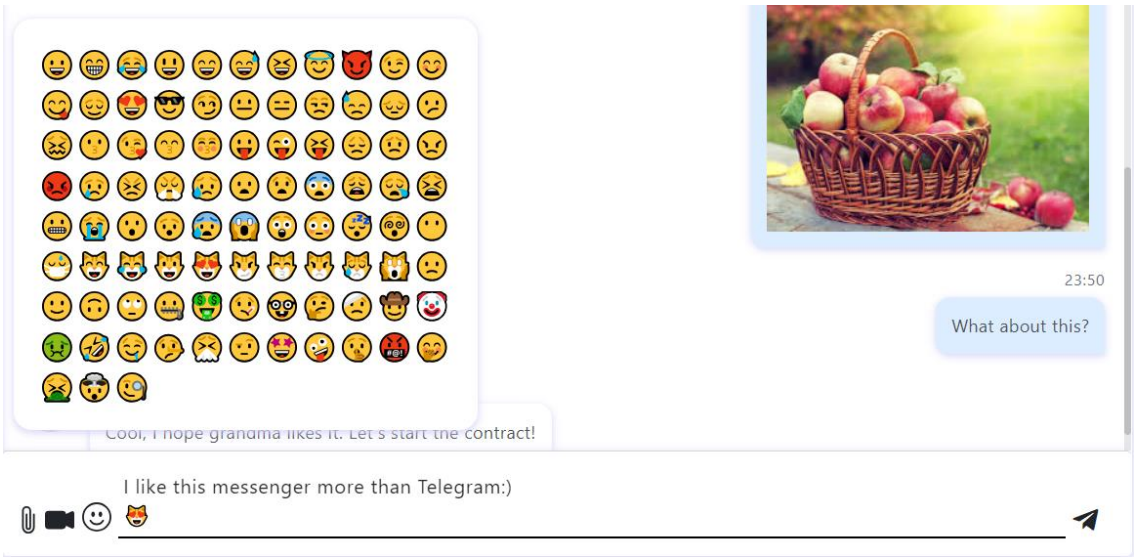

Рисунок 3.8 - Друк текстового повідомлення з емоціями

Також платформа надає зручний інтерфейс для перегляду та керування власними задачами, та задачами, що виконує поточний користувач. Для цього на сторінці користувача наявні три таби на навігаційній панелі. Таб “My Jobs” відповідає за керування задачами, власник може редагувати їх, позначати як неактивними чи активними у випадку якщо задача більше не потребує виконання та переглядати приклад відображення задачі виконавцю. Сторінку продемонстровано у вигляді скріншоту (рис. 3.9). “Offers For My Jobs” – сторінка, де користувач може побачити пропозиції на виконання власних робіт, переглядати їх статуси та просто контролювати. Демонстрацію наведено у вигляді скріншоту (рис. 3.10). “My Offers” – це сторінка для керування пропощиціями та оферами, що виконує користувач. Сторінку продемонстровано у вигляді скріншоту (рис. 3.11).

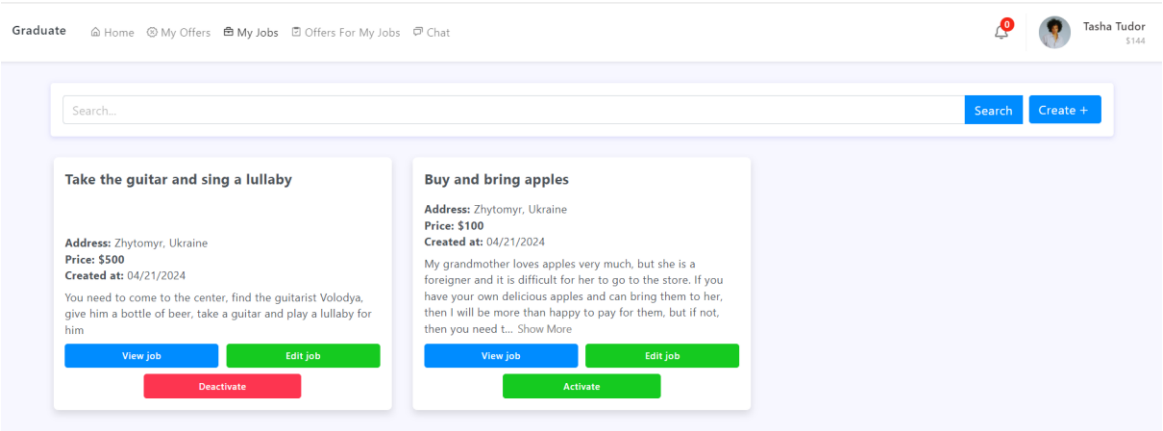

Рисунок 3.9 - Перелік задач, що створив користувач

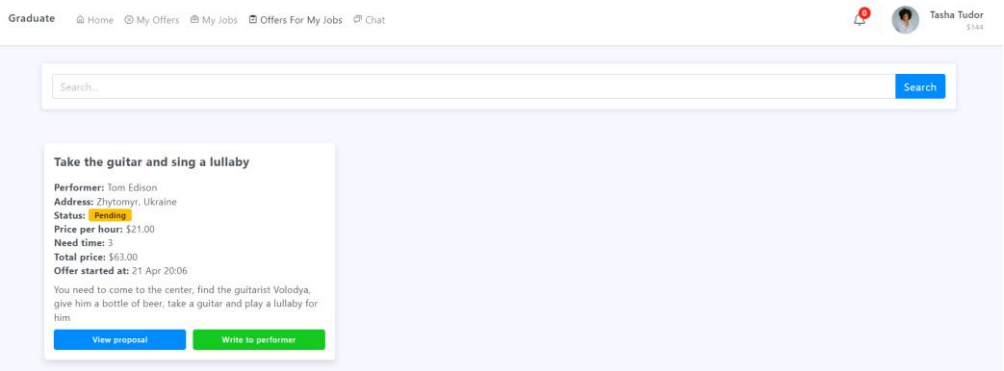

Рисунок 3.10 - Сторінку з оферами користувача, що виконуються для нього

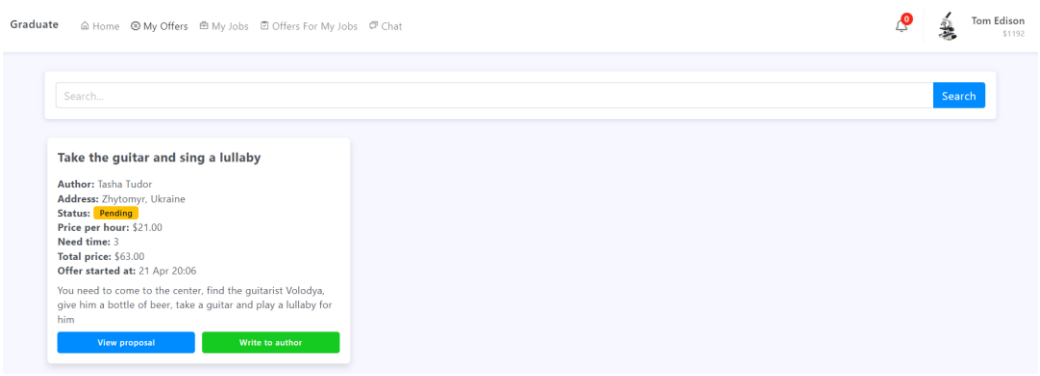

Рисунок 3.11 - Сторінку з оферами користувача, що виконуються ним

Користувачі можуть створювати запити на виконання завдань. Для цього їм потрібно просто заповнити форму з основної сторінки з переліком задач, або ж зі сторінки детального перегляду інформації про задачу. Форма дозволяє виконавцям поторгуватись з замовниками, вона має поле де виконавець вказує ціну, яку він хоче мати за годину роботи та тривалість виконання задачі. Форма продемонстрована у вигляді скріншоту (рис. 3.12).

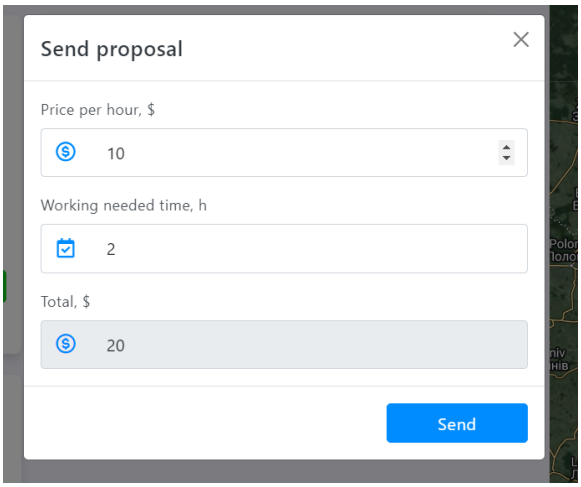

Рисунок 3.12 - Форма відправки запиту на виконання задачі

Замовники ж можуть відмовитись від послуги, якщо ціна їм не подобається, або терміни не ті, або замовник шукає виключно багатозіркового виконавця. Також, можуть погодитись, якщо запропоновані умови підходять і в чаті питань між користувачами не виникло. Візуал панелі керування задачею у замовника представлено у вигляді скріншоту (рис. 3.13).

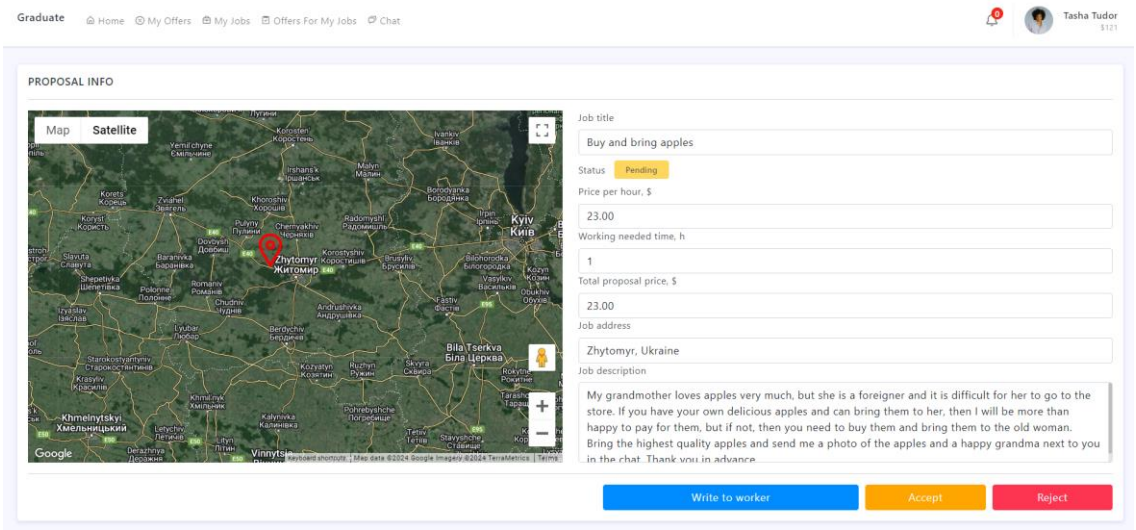

Рисунок 3.13 - Перегляд запиту виконання завдання

Після початку виконання замовлення, виконавець може відправити запит на заверешння роботи, написати замовнику для уточнення питань, або створити суперечку, якщо станеться якась не обговорена ситуація і користувачі не зможуть вирішити її власними силами. Після цього користувач зможе написати замовнику, або ж почати суперечку. Приклад відправки запиту на підтвердження завершення задачі зображено у вигляді скріншоту (рис. 3.14).

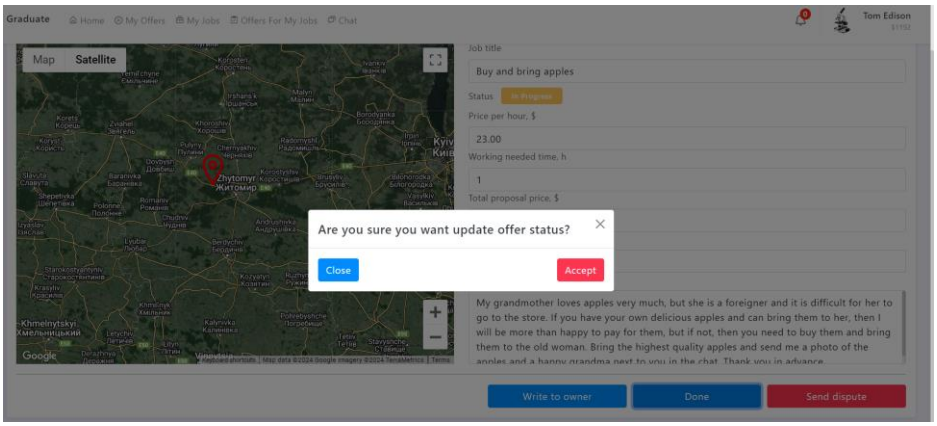

Рисунок 3.14 - Відправку виконавцем запиту на підтвердження виконання

Замовник, коли отримує нотифікацію про запит на вирішення задачі, може спочатку з’ясувати, чи задача була завершена, чи все відбулося так, як того хотів замовник і після цього підтвердити завершення. Якщо якийсь з етапів буде не завершений, користувач може скасувати виконання задачі, після чого виконавець може або підтвердити це і задача закриється, а власник отримає кошти назад, або створити суперечку з участю адміністратора. Приклад зображено у вигляді скріншоту (рис. 3.15).

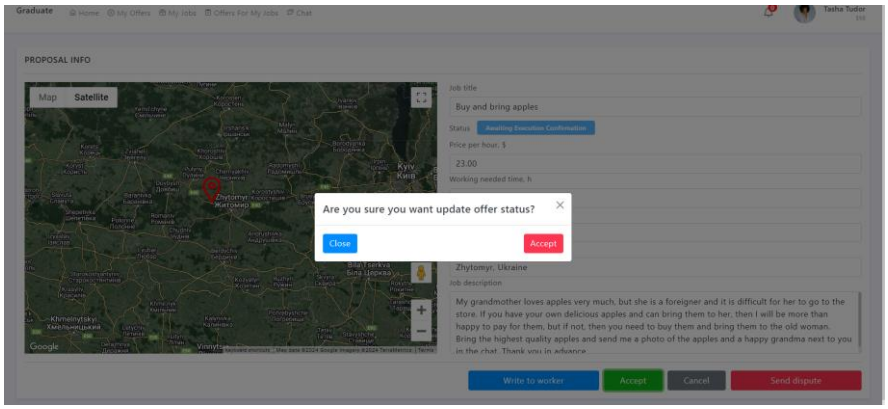

Рисунок 3.15 - Підтвердження виконання задачі

На більшість подій система пропонує оповіщення. Наприклад на авторизацію з якогось пристрою, на надходження нового повідомлення в чаті, на зміну статусу задачі. Приклад нотифікації, що повідомляє про успішне завершення задачі і отримання коштів зображено у вигляді скріншоту (рис. 3.16).

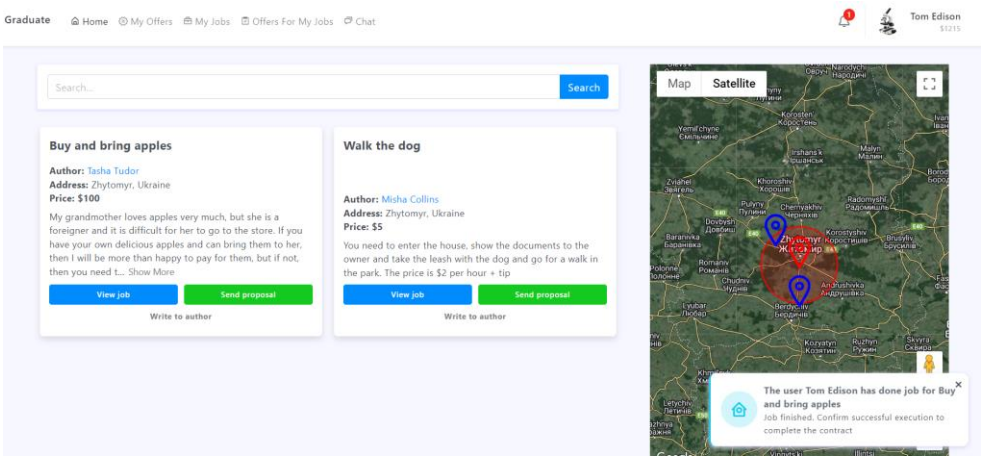

Рисунок 3.16 - Нотифікація про успішне завершення задачі

Ще однією перевагою платформи є взаємодія з реальними коштами. Таким чином, було розроблено сторінку для поповнення внутрішнього балансу коштами з інших платформ, та для виведення їх з платформи. Вона складається з трьох табів. Таб з базвою інформацією про поточний баланс (рис. 3.17), таб для поповнення балансу (рис. 3.18), та таб для виведення коштів (рис. 3.19).

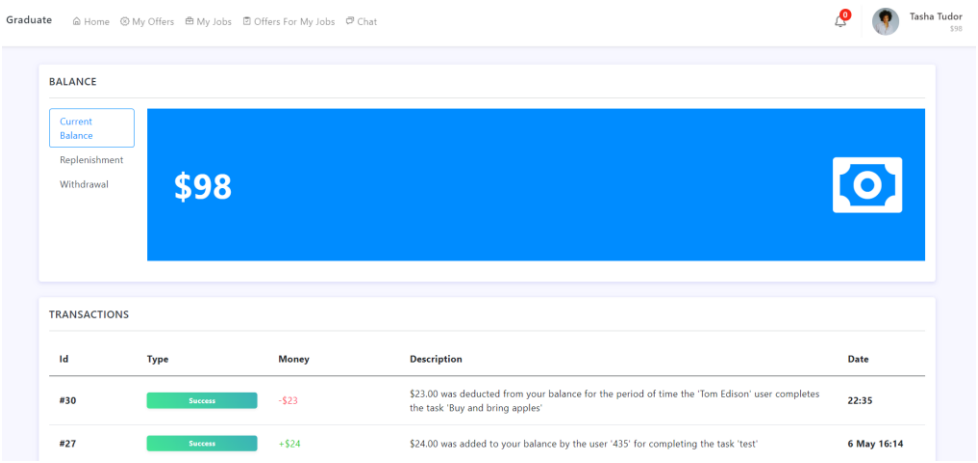

Рисунок 3.17 - Сторінка поповнення балансу користувача

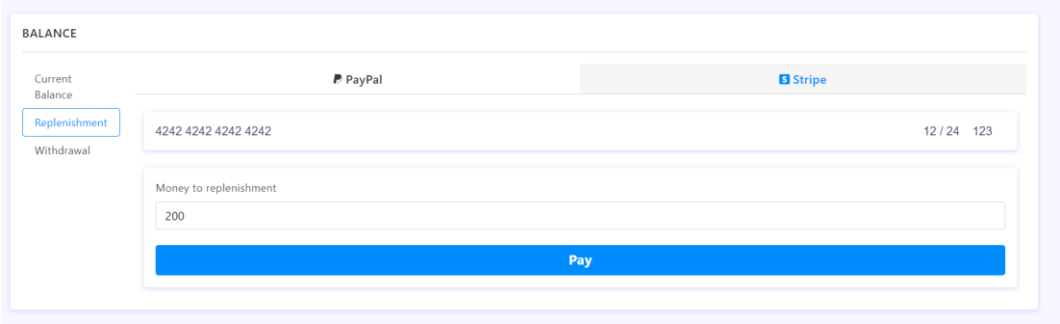

Рисунок 3.18 - Приклад поповнення балансу через страйп

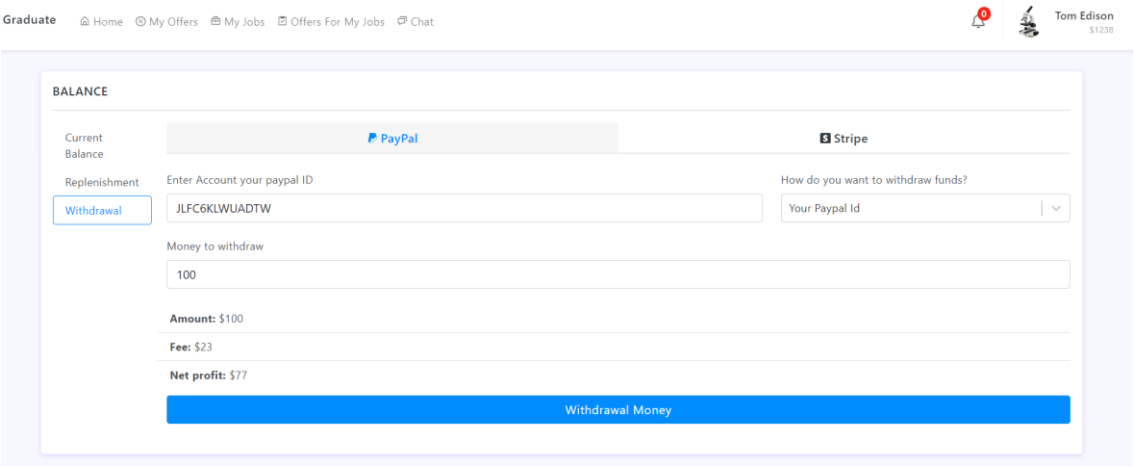

Рисунок 3.19 - Приклад виведення коштів з балансу на акаунт пейпалу

Як було згадано раніше, адміністратор має вирішувати суперечки, які не можуть вирішити користувачі самі. Таким чином, для нього було розроблено інтерфейс для цієї операції. Адміністратор може переглядати суперечки і закріплювати за собою ті, які він готовий вирішити. Таким чином було розроблено таблицю (рис. 3.20) для пошуку суперечок, що може вирішити адміністратор.

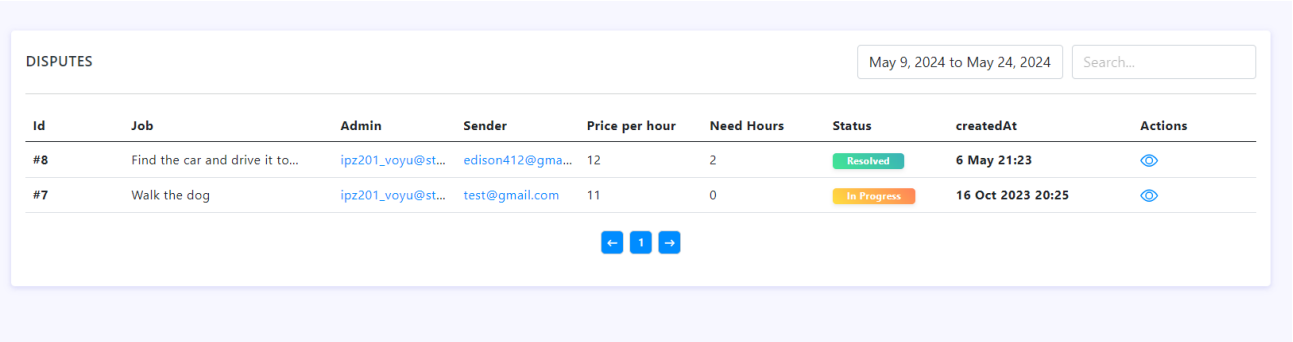

Рисунок 3.20 - Список суперечок у адміністратора

Для вирішення суперечок адміністратор повинен мати можливість переглядати інфомрацію про задачу, пропзицію, суперечку та про користувачів. Основну частину інформації про суперечку, де відображені умови задачі та опис проблеми суперечки продемонстровано у вигляді скріншоту (рис. 3.21).

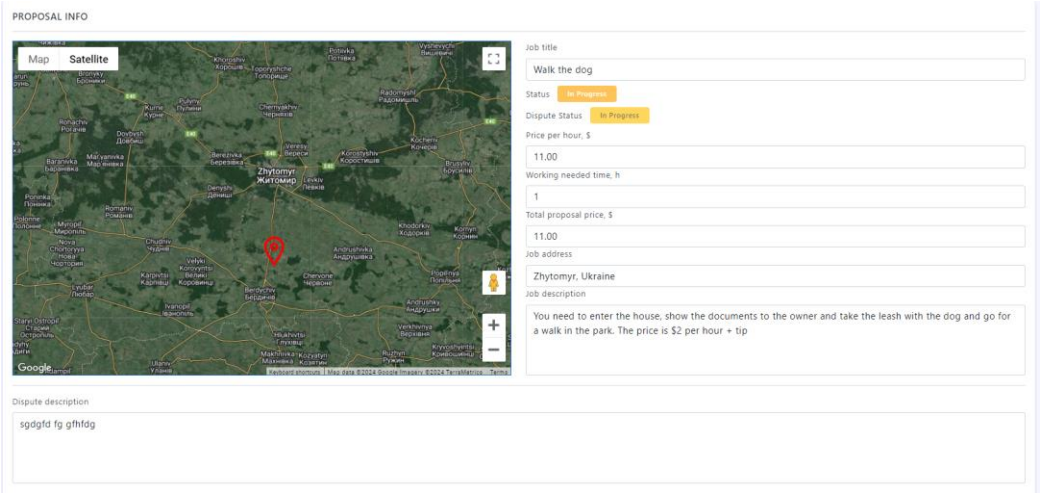

Рисунок 3.21 - Перегляд інформації адміністратором про суперечку

Крім інформації, на сторінці наявні кнопки для вирішення суперечки, для перегляду чату користувачів та відправки повідомлення від імені сисктеми кожному з учасників. Кожна з цих можливостей допомагає

адміністратору розібратись з пробломою та зробити правильний вибір. Крім цього, адміністратор може відмовитись від суперечки, щоб хтось інший, досвідченіший, перехопив її та вирішив. Приклад перегляду фрагменту інформації про суперечку, що відповідає за інформацію про користувачів да дії з суперечкою наведено у вигляді скріншоту частини сторінки (рис. 3.22).

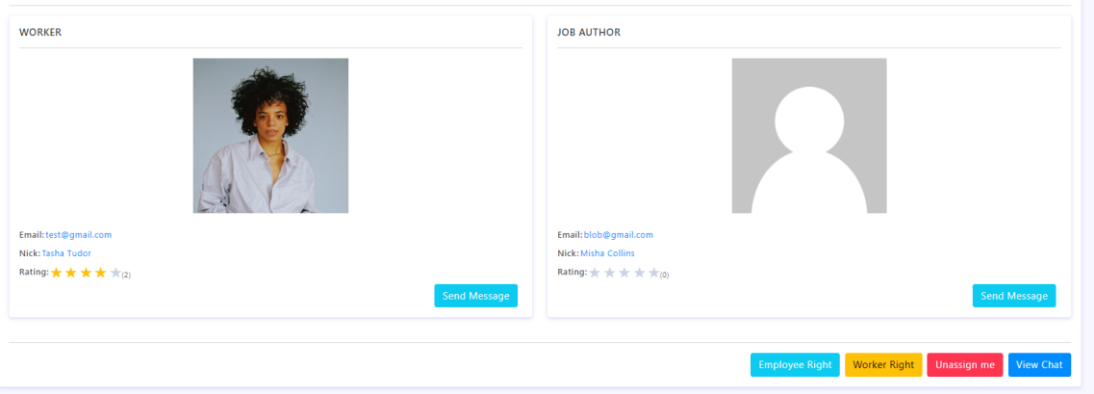

Рисунок 3.22 - Перегляд інформації про учасників суперечки

Платформа надає можливість адміністраторам переглядати чати користувачів між якими виникла суперечка. Це дозволяє поглибитись у сутичку та прочитати про додаткові умови, які не були обговорені в самому описі задачі. Також, адміністратор бачить будь-які зміни чату. Наприклад червоним кольором підсвічено повідомлення, що були видалені, а зеленим – редаговані. Приклад чату, який переглядає адміністратор наведено у вигляді скріншоту (рис. 3.23).

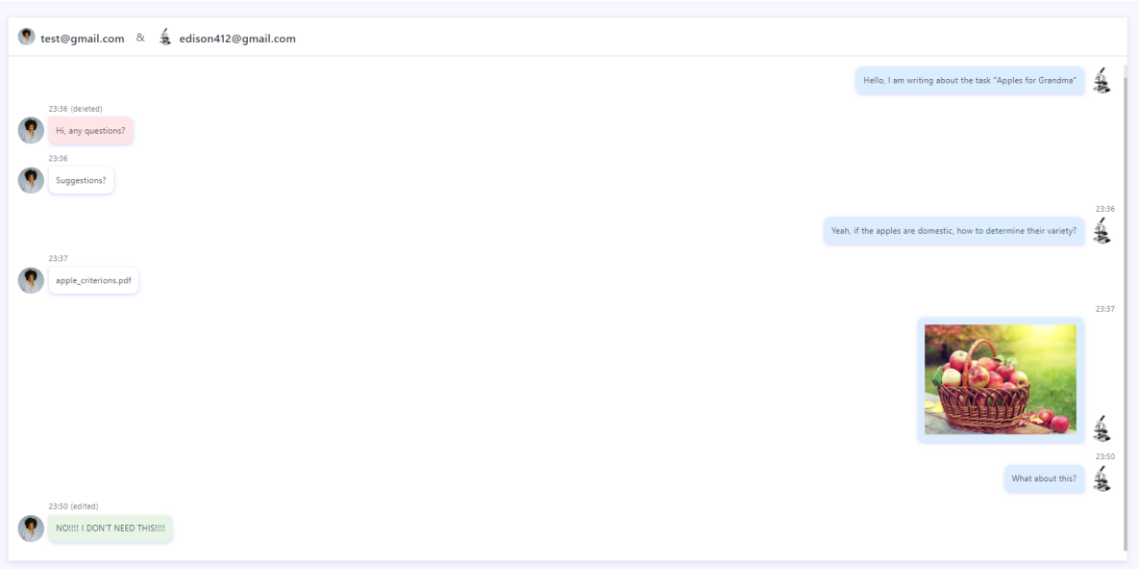

Рисунок 3.23 - Перегляд чату користувачів адміністратором

Для зручності перегляду історії змін повідомлень, було створено окремий попап, який активується при настисканні на слово “edited” біля повідомлення зеленого кольору. Попапу продемонстровано у вигляді скріншоту (рис. 3.24).

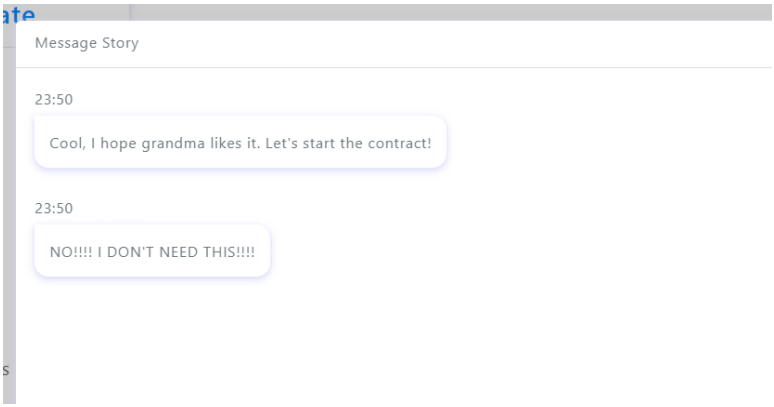

Рисунок 3.24 - Перегляд історії редагування повідомлення адміністратором

Крім того, у адміністратора є можливість переглядати статистику роботи платформи. Платформа включає в панелі адміністратора статистику контролю фінансів. На діаграмі відображається сума, яка надійшла за певний період, до платіжної системи акаунту, та була виведена. Також система включає статистику відвідуваності користувачами платформи, статистику збільшення зареєстрованих користувачів у системі, статистику по задачам, наприклад скільки було виконано і скільки було створено у цей же період, та статистику суперечок – скільки було створено і вирішено за певний проміжок часу. Візуалізацію статистики відвідування платформи наведено у вигляді скріншоту частини сторінки (рис. 3.25).

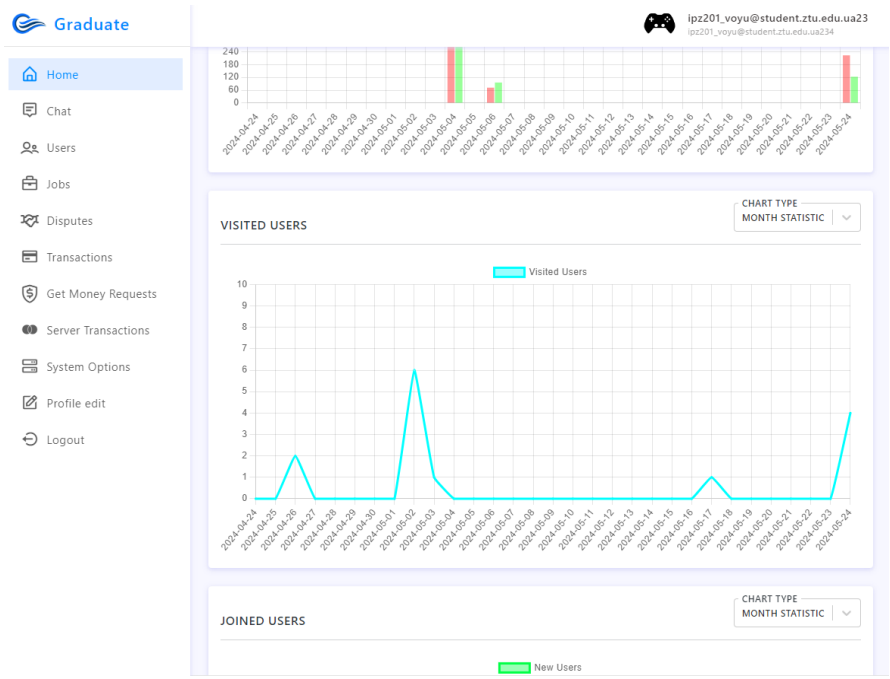

Рисунок 3.25 - Перегляд статистики адміністратором

На додачу до всього додаток пропонує адаптивний дизайн, що виглядає досить гарним та зручним на телефонах. Гарним прикладом є перегляд основних сторінoу з телефону iPhone SE (рис. 3.26). Як можна побачити, платформа у такому розширенні екрану виглядає досить зручною і не втрачає основного функціоналу.

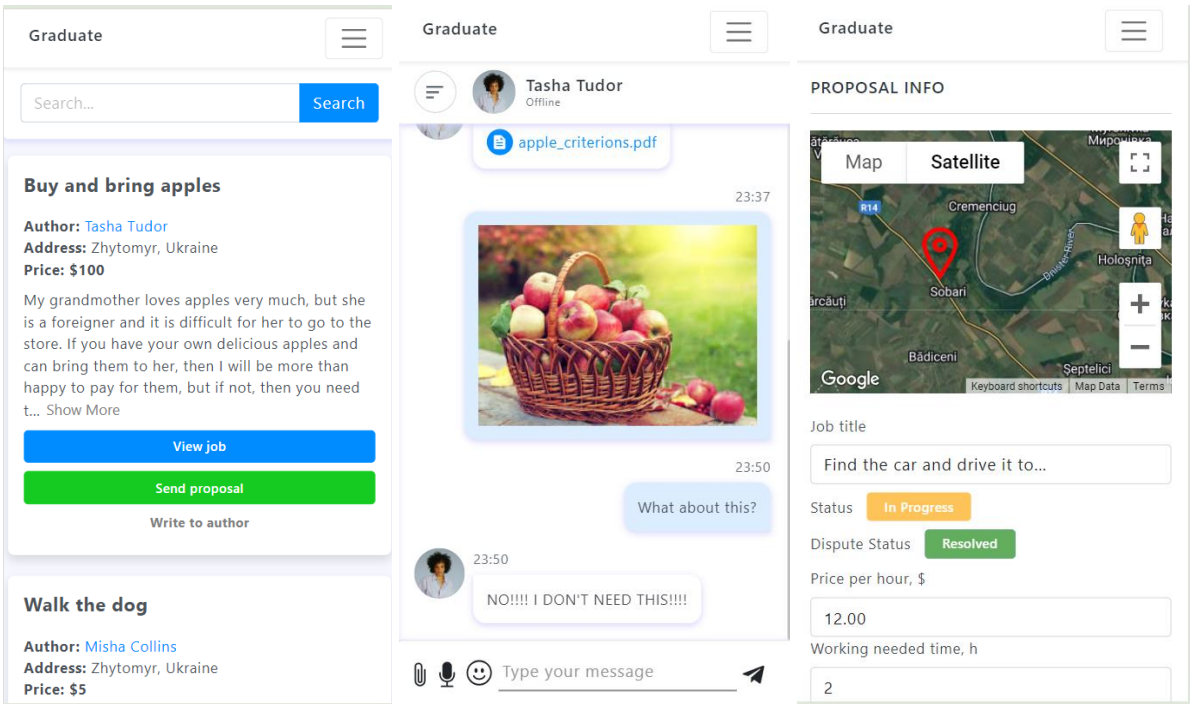

Рисунок 3.26 - Адаптивність основних сторінок платформи

3.3 Тестування платформи

Тестування є одним з ключових аспектів розробки програмного забезпечення, який гарантує якість, надійність та безпеку додатку. Воно допомагає виявити та виправити помилки на ранніх етапах, що значно знижує витрати на виправлення дефектів у майбутньому. Завдяки тестуванню можна забезпечити відповідність додатку вимогам користувачів і бізнесу, що підвищує його конкурентоспроможність і задоволеність клієнтів. Крім того, тестування сприяє підтримці високих стандартів продуктивності та сумісності з різними платформами та пристроями. Тестування платформи відбувалося в двох форматах, що дозволило охопити всі аспекти функціональності та забезпечити високу якість кінцевого продукту.

Першим є ручне регресійне тестування. За допомогою нього відбувалася перевірка на усіх етапах розробки. Після кожного глобального оновлення коду відбувалася перевірка не лише модулів, яких гарантовано зачіпали зміни, а й усіх інших. Це дозволило уникнути помилок різної складності на початкових етапах їх утворення.

Другим форматом є юніт тести. Ними було покрито усі моделі бекендової частини платформи. Вони дозволяли визначати чи не впливають поточні зміни бази даних на виконання коду.

Однією з переваг юніт тестів була можливість перевіряти як відпрацьовують методи різних моделей з реальною базою. Це дозволило перевіряти коректність міграцій, перед використанням їх на реальних серверах. Також, завдяки такому підходу тестування, можна було визначати критичні методи і якось їх оптимізовувати. Приклад виконаних тестів з попередженням про надмірний час виконання зображено у вигляді скріншоту консолі (рис. 3.27).

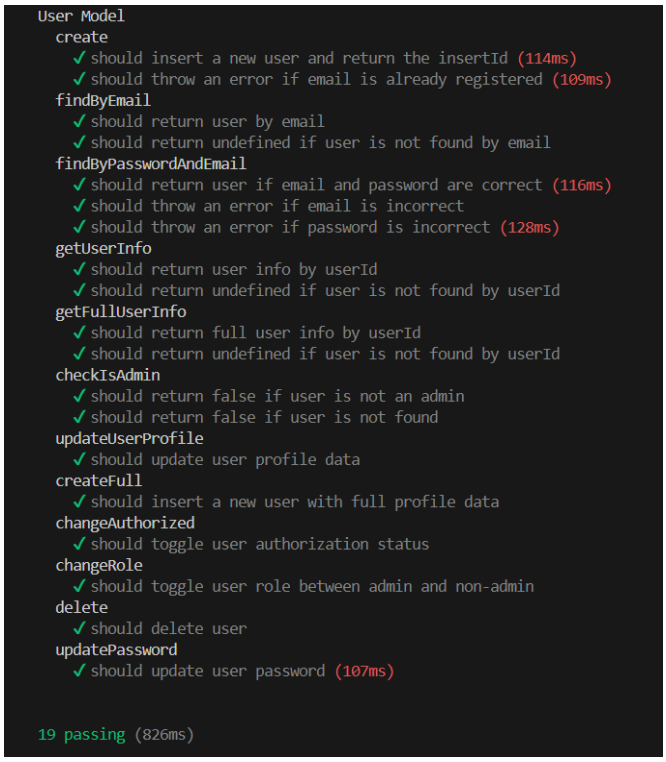

Рисунок 3.27 - Приклад виконаних тестів

Як було сказано раніше, юніт тестами було покрито лише моделі платформи, так як саме в моделях відбувається основна логіка обробки даних. Моделі є центральним компонентом архітектури додатку, оскільки вони відповідають за взаємодію з базою даних та маніпулювання даними. Таким чином, тестами було покрито найважливіші фрагменти коду, що забезпечило перевірку коректності виконання ключових функцій та методів. Це дало можливість виявити потенційні помилки в логіці обробки даних та виправити їх до впровадження у виробництво.

У результаті, тестування допомогло забезпечити стабільність та надійність роботи платформи. Це дозволило охопити всі аспекти функціональності та забезпечити високу якість кінцевого продукту. Юніт тести забезпечували швидке виявлення помилок під час кожного циклу розробки, тоді як ручне тестування допомогло перевірити додаток у різних сценаріях використання. Такий підхід до тестування забезпечив можливість всебічної перевірки та гарантував високу якість кінцевого продукту.

Висновки до третього розділу

У третьому розділі було розкрито важливі аспекти функціонування системи та її інтерфейсу. Під час аналізу параметрів системи, було досліджено спектр налаштувань, які розробник платформи може швидко змінювати і залежно від цього впливати на різні системно важливі процеси, такі як фінансові операції, взаємодія бекенду та фронтенду.

У другій частині цього розділу, було розглянуто ще один важливий фрагмент розробки, а саме - структура інтерфейсу. Основна увага була зосереджена на спрощенні користувальницького досвіду. Було розглянуто різні аспекти інтерфейсу, зокрема розташування елементів, їх дизайн та інтерактивність. Головною метою було створення зручного, адаптивного та інтуїтивно зрозумілого інтерфейсу, який би сприяв ефективній роботі користувачів.

На сам кінець було розглянуто ще один важливий етап розробки, а саме тестування. У цьому розділі було розглянуто види тестування, що було використано до проекту. Результати тестування були детально проаналізовані, а виявлені проблеми та помилки були виправлені. Також було приділено увагу на особливості кожного виду тестування і їх вплив на розробку продукту. Цей етап дозволив переконатися в якості та надійності нашої системи перед її завершенням.

ВИСНОВКИ

Розробка платформи передбачала глибокий аналіз потреб користувачів та характеристик ринку, тому було проведено дослідження різних платформ, що надають подібні послуги, щоб визначити основні вимоги до платформи. На основі цього аналізу була розроблена архітектура платформи, і вибрані найбільш підходящі технології для її реалізації.

Наступним кроком була розробка структури платформи та забезпечення взаємодії між її компонентами. Цей етап дозволив поетапно вдосконалювати платформу та оптимізувати її процеси. Також була розроблена база даних для зберігання інформації про користувачів, завдання, транзакції та інші важливі дані, з урахуванням потреб платформи та забезпеченням доступності даних для всіх компонентів системи.

Після цього, у другому розділі було докладно розглянуто різні способи використання платформи, а також її можливості та способи взаємодії користувачів. Опис було почато з основного функціоналу, зосереджуючись на процесі створення та розміщення завдань для виконання. Крім того, великий акцент був зроблений на можливості обговорення умов та вирішення можливих конфліктів між користувачами. Під час розгляду цих аспектів було докладно проаналізовано кожну можливість та перевагу, яку платформа може запропонувати.

Окрім цього, було приділено належну увагу процесам авторизації, а також взаємодії користувачів з чатом та іншими ключовими модулями платформи. Після завершення етапу, було зроблено висновок, що після розробки було забезпено максимальний рівень зручності та функціональності для користувачів платформи.

У третьому розділі було ретельно розглянуто схему розгортання платформи та описано взаємодію між її компонентами. Також у цьому розділі було представлено детальний опис інтерфейсу програмного продукту для різних категорій користувачів, таких як адміністратори та звичайні

користувачі. Це дозволило забезпечити максимальний рівень зручності та доступності для всіх користувачів платформи. У кінці розділу було проведено фінальний етап – тестування. Під час цього етапу було докладно проаналізовано різноманітні види тестування та їх результати.

Отже, процес розробки та впровадження платформи був складним та об’ємним, але дуже важливим. Розробота кожного над кожним етапом розробки платформи керувалася метою створення надійного та ефективного інструменту для користувачів платформи. Після завершення проекту можна бути впевненим, що платформа готова зустріти вимоги сучасного ринку та стати невід'ємною частиною робочого процесу для великої кількості звичайних людей, що хочуть подарувати трішки щастя своїм рідним чи родичам на значних відстанях від них самих.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Взаємодія клієнт-сервер [Електронний ресурс]. – 2024.– Режим доступу до ресурсу: https://ua5.org/technol/2094-vzayemodiya-kliyent-server.html.
2. Монолітна архітектура ПЗ [Електронний ресурс]. – Режим доступу до ресурсу: https://qalight.ua/baza-znaniy/shho-take-monolitna-arhitektura/.
3. Мікросервісна архітектура ПЗ. [Електронний ресурс]. – Режим доступу до ресурсу: https://qalight.ua/baza-znaniy/shho-take-mikroservisna-arhitektura-pz/.
4. Про архітектуру додатків [Електронний ресурс]. – 2024.– Режим доступу до ресурсу: https://foxminded.ua/arkhitektura-zastosunku/.
5. MVC проти MVVM – різниця між ними [Електронний ресурс] . – 2024. – Режим доступу до ресурсу: https://www.guru99.com/uk/mvc-vs-mvvm.html.
6. Використання розширення Visual Studio Code [Електронний ресурс] . – 2024. – Режим доступу до ресурсу: https://learn.microsoft.com/uk-ua/power-pages/configure/vs-code-extension.
7. Документація Node.js [Електронний ресурс]. – Режим доступу до ресурсу: https://nodejs.org/en/docs/.
8. Документація express.js [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://expressjs.com/en/5x/api.html.
9. Початок використання PayPal REST APIs [Електронний ресурс]. – Режим доступу до ресурсу: https://developer.paypal.com/api/rest/.
10. Stripe Docs [Електронний ресурс]. – Режим доступу до ресурсу: https://docs.stripe.com/get-started/development-environment?lang=node.
11. Як використовувати Socket.io у Node.js [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://socket.io/get-started/chat/.
12. Навіщо розуміти backend-частину та чому варто обрати Node.js [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://robotdreams.cc/uk/blog/241-zachem-ponimat-backend-chast-i-pochemu-stoit-vybrat-node-js.

13. Створення та структурування програми Node.js MVC [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://blog.logrocket.com/building-structuring-node-js-mvc-application/.
14. Покроковий підручник React.js [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://reactjs.org/docs/getting-started.html.
15. Як використовувати Socket.io у React [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://socket.io/how-to/use-with-react.
16. React PayPalButtons [Електронний ресурс]. – Режим доступу до ресурсу: https://paypal.github.io/react-paypal-js/?path=/docs/example-paypalbuttons--default.
17. React Stripe [Електронний ресурс] – Режим доступу до ресурсу: https://docs.stripe.com/stripe-js/react.
18. Документація Bootstrap 5 українською [Електронний ресурс]. – Режим доступу до ресурсу: https://bootstrap21.org/uk/docs/5.0/getting-started/introduction/.
19. Документація використання піктограми Bootstrap 5 українською [Електронний ресурс]. – Режим доступу до ресурсу: https://bootstrap21.org/uk/docs/5.0/extend/icons/.
20. SQL в Access: основні поняття, глосарій і синтаксис [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: http://surl.li/tydde.
21. Пишемо вражаюче швидкі MySQL запити [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: http://surl.li/tyddp.
22. Генерація SQL-запиту засобами MySQL-сервера [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://dou.ua/lenta/articles/sql-query-mysql-server/.
23. SQL Підзапити [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://support.microsoft.com/en-us/topic/run-a-sql-query-d41fc0b1-1c88-40d4-bbd1-951de6e94e2a.
24. Патерн “Фасад” [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/facade.

25. Патерн "Компонувальник" [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/composite.
26. Патерн "Міст" [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://refactoring.guru/uk/design-patterns/bridge.
27. Верстка сайтів [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://webtune.com.ua/statti/web-rozrobka/verstka-sajtiv/.
28. Види верстки сайтів [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://impulse-design.com.ua/ua/chto-takoe-verstka-sajta.html.
29. Що таке адаптивний дизайн та для чого його застосовують [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://onlinemedia.company/blog/sho-take-adaptivnij-dizajn/.
30. 15 Правил UI/UX Дизайна [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://topuser.pro/ux-ui-dizajn-osnovi-pravila-web/.

ДОДАТКИ

Додаток А

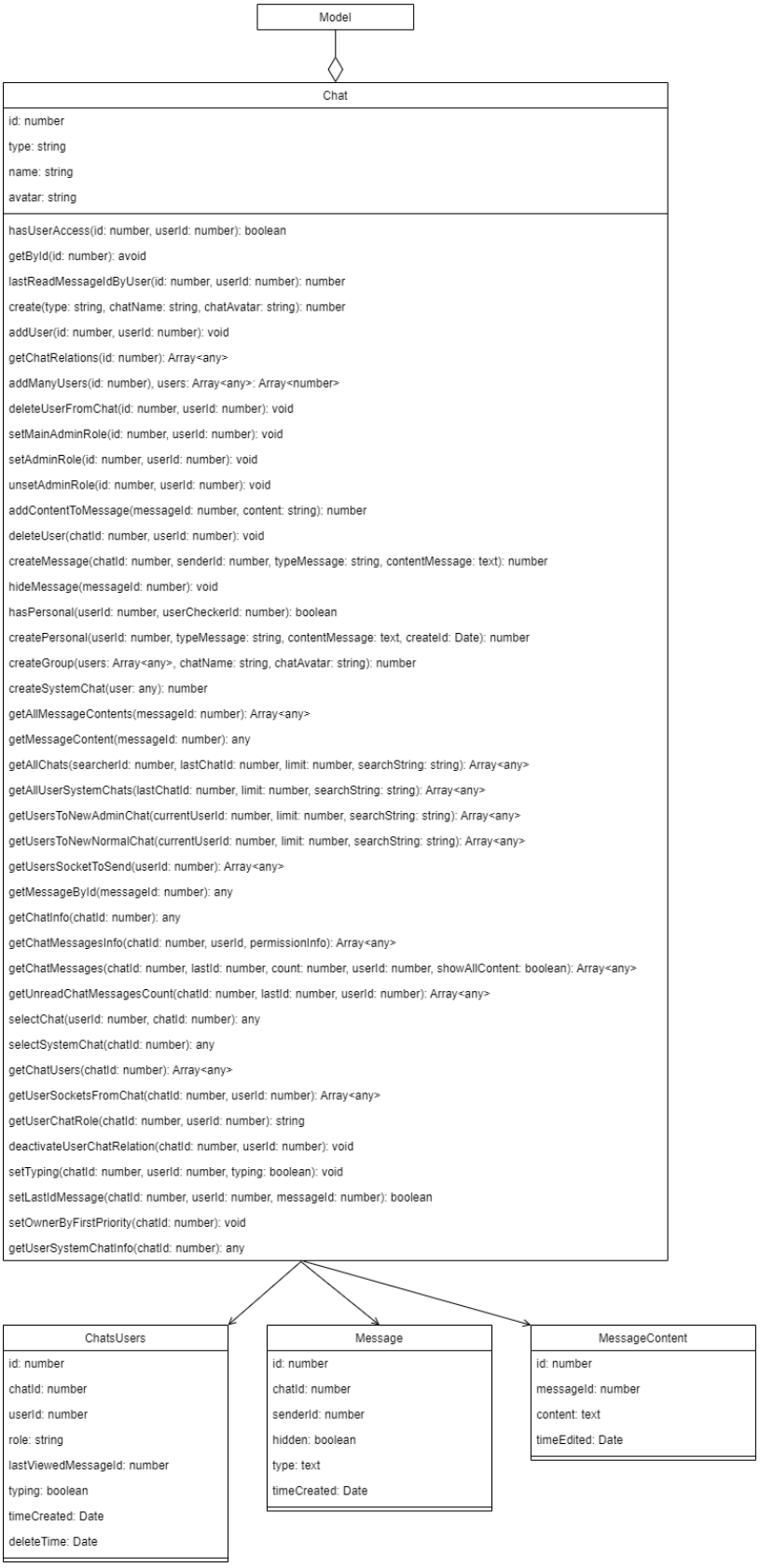

Рисунок А.1 - Діаграма класів на пряму пов’язаних з чатом

Додаток Б

```
require("dotenv").config();

const express = require("express");
const cors = require("cors");
const bodyParser = require("body-parser");
const mysql = require("mysql");
const mainRoutes = require("./routes/main");
const socketIo = require("socket.io");
const { Chat: ChatSocketController } = require("./sockets");

const PORT = process.env.PORT || 5000;

const app = express();
app.use(bodyParser.json());
app.use(express.static("uploads"));

app.use(
  cors({
    credentials: true,
    exposedHeaders: "Authorization",
    origin: "*",
    origin: [process.env.CLIENT_URL, "https://www.sandbox.paypal.com"],
  })
);

const db = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_DATABASE,
  charset: process.env.DB_CHARSET,
});

db.connect((err) => {
  if (err) {
    console.log("this db error: ");
    console.error(err);
  } else {
    console.log("Connected to MySQL database");
  }
});

const server = app.listen(PORT, () => {
  console.log("Server started on port " + PORT);
});

const io = socketIo(server, {
  cors: {
```

```
    credentials: true,
  },
});

mainRoutes(app, db, io);

new ChatSocketController(io, db);
```

Додаток В

```
findByPasswordAndEmail = async (email, password) =>
    await this.errorWrapper(async () => {
      const authError = () => this.setError("Invalid email or password",
401);

      const findUserRes = await this.dbQueryAsync(
        `SELECT * FROM users WHERE email = ?`,
        [email]
      );

      if (!findUserRes.length) authError();

      const user = findUserRes[0];
      const isPasswordValid = await bcrypt.compare(password, us-
er.password);

      if (!isPasswordValid) authError();

      const res = {
        id: user.id,
        email: user.email,
        address: user.address,
        lat: user.lat,
        lng: user.lng,
        nick: user.nick,
        avatar: user.avatar,
        admin: user.admin,
      };

      return res;
});
```

Додаток Г

```
paypalGetMoneyToUser = async (req, res) =>
    this.errorWrapper(res, async () => {
      const { userId } = req.userData;
      const { amount, type, typeValue } = req.body;
```

```
        const feeInfo = await this.systemOptionModel.getFeeInfo();
        const fee = calculateFee(feeInfo, amount);
        const moneyWithoutFee = Number(amount) - fee;

        const result = await sendMoneyToUser(
          type,
          typeValue,
          moneyWithoutFee.toFixed(2),
          "USD"
        );

        const payStory = async (waitingStatus = false) => {
          const newBalance = await this.userModel.rejectBalance(userId,
amount);

          const createdId =
            await this.paymentTransactionModel.createWithdrawalByPaypal(
              userId,
              Number(amount).toFixed(2),
              fee,
              waitingStatus
            );

          return { createdId, newBalance };
        };

        if (result.error) {
          if (
            result.error.toLowerCase() ==
            "Sender does not have sufficient funds. Please add funds and re-
try.".toLowerCase()
          ) {
            const { createdId, newBalance } = await payStory(true);

            await this.getMoneyRequestModel.create(
              createdId,
              userId,
              moneyWithoutFee.toFixed(2),
              "paypal",
              { type, typeValue }
            );

            return this.sendResponseSuccess(
              res,
              "Operation completed successfully",
              { newBalance }
            );
          } else {
            return this.sendResponseError(res, "Operation error", 402);
```

```
        }
      } else {
        const { newBalance } = await payStory();

        await this.serverTransactionModel.createReplenishmentByPaypalFee(
          userId,
          moneyWithoutFee.toFixed(2)
        );

        return this.sendResponseSuccess(
          res,
          "Operation completed successfully",
          { newBalance }
        );
      }
});
```

Додатку Ґ

```
useEffect(() => {
    if (!io) return;

    io.on("created-chat", (data) => onGetNewChat(data));
    io.on("created-group-chat", (data) => {
      onGetNewChat({
        chatId: data.chatId,
        type: data.message.type,
        chatType: data.message.chatType,
        content: data.message.content,
        chatAvatar: data.avatar,
        chatName: data.name,
        timeSended: data.message.timeSended,
        deleteTime: null,
      });
    });

    io.on("success-sended-message", (data) => {
      onGetMessageForSockets(data.message);
    });

    io.on("get-message", (data) => onGetMessageForSockets(data.message));
    io.on("get-message-list", (data) =>
      data.messages.forEach((message) => onGetMessageForSockets(message))
    );

    io.on("success-deleted-message", (data) => onDeleteMessageForSock-
ets(data));
    io.on("deleted-message", (data) => onDeleteMessageForSockets(data));
    io.on("success-updated-message", (data) => onUpdateMessageForSock-
ets(data));
```

```
    io.on("updated-message", (data) => onUpdateMessageForSockets(data));
    io.on("typing", (data) => changeTypingForSockets(data, true));
    io.on("stop-typing", (data) => changeTypingForSockets(data, false));
    io.on("online", (data) => changeOnlineForSockets(data, true));
    io.on("offline", (data) => changeOnlineForSockets(data, false));

    io.on("chat-kicked", (data) => {
      onGetMessageForSockets({ chatId: data.chatId, ...data.message });
      deactivateChat(data.chatId, data.time);
    });

    io.on("file-part-uploaded", async ({ tempKey, message = null }) => {
      const nextPartData = await onSuccessSendBlobPart(tempKey);

      if (!nextPartData) return;
      if (nextPartData == "success saved" && message) {
        onGetMessageForSockets(message);
        return;
      }

      onUpdateMessagePercent({ tempKey, percent: nextPartData["percent"]
});
      setTimeout(() => io.emit("file-part-upload", { ...nextPartData }),
1);
    });

    io.on("message-cancelled", async ({ tempKey }) =>
      onCancelledMessage({ tempKey })
    );
}, [io]);
```

Додаток Д

```
onSendMessage = async (data, sessionInfo) => {
    const userId = sessionInfo.userId;
    const sender = sessionInfo.user;

    const message = await this.chatController.__createMessage(data,
userId);

    if (data.chatType == "personal") {
      let chatId = data.chatId;

      if (!data.chatId) {
        await this.__onCreateNewPersonalChat(message, data, userId);
        chatId = await this.chatController.chatModel.hasPersonal(...);
      }

      this.socketController.sendSocketMessageToUsers(...);
```

```
      this.chatController.sentMessageNotification(...);
    }

    if (data.chatType == "group" || data.chatType == "system") {
      const chatUsers = await
this.chatController.__getChatUsers(data.chatId);
      const chatUserIds = chatUsers.map((chat) => chat.userId);
      const usersToGetMessage = chatUserIds.filter((id) => id != userId);

      if (usersToGetMessage.length) {
        this.socketController.sendSocketMessageToUsers(...);

        const chat = await
this.chatController.chatModel.getById(data.chatId);

        const relations = await
this.chatController.chatModel.getChatRelations(
          data.chatId
        );

        relations.forEach((relation) => {
          if (relation.userId == userId) {
            return;
          }

          this.chatController.sentMessageNotification(...);
        });
      } else if (data.chatType == "system") {
        this.socketController.sendSocketMessageToAdmins(...);
      }
    }

    message["tempKey"] = data["tempKey"];
    message["getterId"] = data["getterId"];

    this.socketController.sendSocketMessageToUsers(
      [userId],
      "success-sended-message",
      {
        message,
      }
    );
};
```

Додаток Е

```
__createMessage = async (data, userId) => {
    const localSend = async (chatId, userId) => {
      const messageId = await this.chatModel.createMessage(
        chatId, userId, data.typeMessage, data.content
```

```
      );
      return await this.chatModel.getMessageById(messageId);
    };

    if (data["chatType"] == "personal") {
      let chatId = await this.chatModel.hasPersonal(data["getterId"],
userId);

      if (chatId) {
        return await localSend(chatId, userId);
      } else {
        chatId = await this.chatModel.create("personal");
        const users = [
          { id: data["getterId"], role: "member" },
          { id: userId, role: "member" },
        ];
        await this.chatModel.addManyUsers(chatId, users);
        const message = await localSend(chatId, userId);

        message["getterInfo"] = await this.userModel.getUserInfo(
          data["getterId"]
        );

        return message;
      }
    } else if (data["chatType"] == "system") {
      const chatId = data["chatId"];
      const userInfo = await this.userModel.getUserInfo(userId);
      const hasUserAccess = await this.chatModel.hasUserAccess(chatId,
userId);

      if (hasUserAccess) {
        return await localSend(chatId, userId);
      } else {
        if (userInfo["admin"]) {
          return await localSend(chatId, userId);
        }

        return null;
      }
    } else {
      return await localSend(data["chatId"], userId);
    }
};
```

Додаток Є

```
onFilePartUpload = async (data, sessionInfo) => {
    const userId = sessionInfo.userId;
    const sender = sessionInfo.user;
```

```
    const { tempKey: tempKey, data: fileBody, type, last } = data;

    const filename = await this.chatController.__uploadToFile(
      userId, tempKey, fileBody, type
    );

    if (!last) {
      this.socketController.sendSocketMessageToUsers(...);
      return;
    }

    const dataToSend = {...};

    const message = await this.chatController.__createMessage(
      dataToSend,
      userId
    );

    if (!data.chatId && data.chatType == "personal")
      await this.__onCreateNewPersonalChat(message, dataToSend, userId);

    await this.chatController.__deleteFileAction(userId, tempKey);

    message["tempKey"] = tempKey;

    if (data.chatType == "group") {
      const chatUsers = await
this.chatController.__getChatUsers(data.chatId);
      const chatUserIds = chatUsers.map((chat) => chat.userId);
      const usersToGetMessage = chatUserIds.filter((id) => id != userId);
      this.socketController.sendSocketMessageToUsers(...);
    } else {
      this.socketController.sendSocketMessageToUsers(...);
      message["getterId"] = data.getterId;
    }

    this.socketController.sendSocketMessageToUsers(...);
};
```