МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

АЛЕКСЮК ВОЛОДИМИР ВІКТОРОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 20__ р.

РОЗРОБКА ВОЛОНТЕРСЬКОЇ ПЛАТФОРМИ ІЗ ЗАСТОСУВАННЯМ NODE.JS ТА REACT

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
І. В. Богач, доцент кафедри
інформаційних технологій,
к.т.н., доцент

_____

Оцінка:____/ ____/ _____
(бали за шкалою СКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця 2024

АНОТАЦІЯ

Алексюк В.В. Розробка волонтерської платформи із застосуванням Node.js та React. Спеціальність 122 «Комп’ютерні науки», освітня програма «Комп’ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2024.

У кваліфікаційній (бакалаврській) роботі досліджено та проаналізовано основні поняття веб-додатків, виділені їх основні етапи розробки. За допомогою таких технологій як: HTML, SCSS, JavaScript, бібліотеки React та Node.js була розроблена веб-платформа для підтримки благодійних ініціатив та збору коштів.

Ключові слова: веб-додаток, веб-розробка, волонтерська платформа, JavaScript, React, Node.js.

82 ст. 41 рис., 1 табл., 3 дод., 40 джерел.

ABSTRACT

Alexyuk V. Development of a Volunteer Platform Using Node.js and React. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl Stus Donetsk National University, Vinnytsia 2024.

In the qualification (bachelor's) work the concept of web applications is researched and analyzed, their basic principles and creation are highlighted. With the help of such technologies as HTML, SCSS, JavaScript, React library, Node.js, this web application was developed, the main purpose of which is to support for charitable initiatives and fundraising.

Keywords: web application, web development, volunteer platform, JavaScript, React, Node.js.

ЗМІСТ

ВСТУП 4
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ 6
1.1 Підстави для створення веб-платформи 6
1.2 Огляд та порівняння існуючих волонтерських веб-платформ 6
1.3 Етапи розробки веб-додатку 7
РОЗДІЛ 2. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ВЕБ-ДОДАТКУ 10
2.1 Аналіз мов програмування та інструментів для розробки Frontend частини веб-додатку 10
2.2 Аналіз мов програмування та інструментів для розробки Back-End частини веб-додатку 15
2.3 Вибір бази даних 21
2.4 Аналіз та вибір інтегрованого середовища розробки 26
2.5 Вибір інструменту для тестування API 31
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ВОЛОНТЕРСЬКОЇ ВЕБ-ПЛАТФОРМИ 33
3.1 Проектування та аналіз функціоналу веб-платформи 33
3.2 Розробка серверної частини веб-платформи 37
3.3 Тестування серверної частини веб-платформи 46
3.4 Розробка клієнтської частини веб-платформи 50
3.5 Тестування розробленого програмного забезпечення 57
ВИСНОВКИ 62
СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ 63
ДОДАТКИ 67
ДОДАТОК А Лістинг основних функцій серверної частини 68
ДОДАТОК Б Лістинг основних функцій клієнтської частини 73
ДОДАТОК В Декларація щодо унікальності текстів роботи та невикористання матеріалів інших авторів без посилань 82

ВСТУП

Актуальність теми дослідження. В контексті країни, що перебуває в стані війни, надзвичайно висока, оскільки волонтерські онлайн платформи стають критично важливим інструментом для забезпечення різних потреб як військових так і цивільних осіб, які постраждали внаслідок конфлікту [1].

Волонтерські платформи відіграють ключову роль у забезпеченні потреб військових: надання необхідного обладнання, засобів для безпеки та ефективності на передовій. Це дозволяє організувати поставку спорядження, транспорту, медикаментів та інших ресурсів для збереження життя та підтримки на лінії фронту.

Проте, із важливістю таких платформ приходить і проблема шахрайства та безпеки. Умови військового конфлікту створюють ідеальне середовище для зловживання та шахрайства, де нечесні особи можуть використовувати волонтерські платформи для виманювання грошей під маскою благодійних зборів чи допомоги.

Тому, водночас з розв'язанням технічних проблем та гарантуванням безпеки даних є також не менш важливим створення ефективних механізмів перевірки й контролю діяльності на платформі, щоб уникнути можливих ситуацій шахрайства й забезпечити користувачам довіру до процесу надання допомоги.

Актуальним є проектування та розробка волонтерської платформи, яка об'єднує волонтерів та користувачів для підтримки благодійних ініціатив та збору коштів, що на відміну від існуючих платформ буде забезпечувати більшу надійність та гнучкість для змін.

Мета роботи полягає у покращенні сервісів волонтерських платформ з використанням сучасних технологій.

Для досягнення поставленої мети потрібно вирішити **наступні завдання дослідження:**

1. Здійснити аналіз предметної галузі та існуючих аналогів.

2. Підібрати оптимальні технології для реалізації волонтерської платформи.

3. Розробити архітектуру веб-платформи.

4. Розробити функціонуючу веб-платформу.

5. Протестувати платформу на відповідність початковим вимогам.

Об’єкт дослідження – процеси проектування та розробки платформ та веб-додатків.

Предмет дослідження – методи та технології проектування, розробки платформ, веб-додатків, та забезпечення надійності розроблених систем.

Теоретичне та практичне значення одержаних результатів полягає в розробці веб-платформи для взаємодії волонтерів та користувачів із застосуванням Node.js та React, що дозволяє легке модифікування системи.

В результаті отримано готову веб-платформу, що здатна забезпечувати надійне функціонування платформи волонтерських зборів, яка в свою чергу є головною ланкою між волонтером та користувачем.

Апробація результатів дослідження

Результати роботи було представлено на Всеукраїнській науково-практичній конференції «Комп’ютерні технології обробки даних» (КТОД-2023) [2].

Впровадження результатів дослідження

Результати роботи, а саме розроблена волонтерська веб-платформа, впроваджується в основний сайт Донецького національного університету імені Василя Стуса.

Структура кваліфікаційної роботи

Робота складається зі вступу, трьох розділів, висновку, списку використаних джерел з 40 джерел включаючи інтернет-ресурси, 1 таблиці та 41 рисунка. Загальний обсяг основної частини роботи - 59 сторінок.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Підстави для створення веб-платформи

Під час аналізу предметної області, слід звернути увагу на різноманітні аспекти, які впливають на розвиток волонтерських ініціатив та платформ у сучасному цифровому середовищі. Метою даного підрозділу є дослідження ключових аспектів, що визначають динаміку та сприйняття волонтерства.

В країні, що знаходиться в стані конфлікту, війни чи будь якої іншої кризи, волонтерство стає необхідним елементом підтримки та солідарності. Цифрові платформи волонтерства відіграють ключову роль у координації гуманітарної допомоги. Однак, враховуючи вразливу ситуацію, важливо розробляти платформи з урахуванням проблем безпеки, конфіденційності та етичних стандартів.

Завдяки стрімкому розвитку технологій, волонтери мають більше можливостей та засобів для реалізації свого потенціалу. Адже так, цифрові платформи дозволяють волонтерам швидше обмінюватись інформацією, співпрацювати та реалізовувати проекти на різних рівнях.

1.2 Огляд та порівняння існуючих волонтерських веб-платформ

Із початком війни в країні, кількість волонтерів стрімко почала зростати (див. рис. 1.1), через що, зросла необхідність в створенні волонтерських платформ.

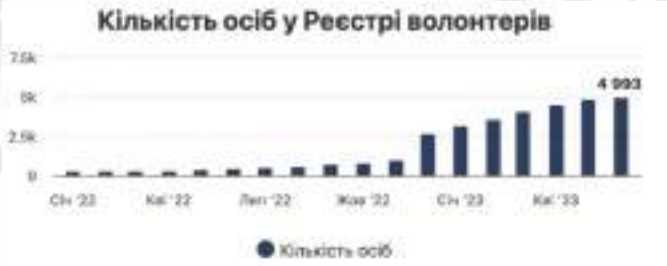


Рисунок 1.1 - Графік кількості осіб у Реєстрі волонтерів

У зв’язку з цим, з’явилося чимало волонтерських платформ, спрямованих на залучення, координацію та підтримку волонтерів у їхній діяльності. Переглянемо вже існуючі, дві, платформи для кращого розуміння їхніх можливостей, відмінностей та функціоналу:

1. HelpUA Foundation – ця платформа спеціалізується на волонтерських зборах різних типів, але переважають саме збори на потреби армії. Що до функціоналу, платформа має всі базові функції, які чудово організовані та інтуїтивно зрозумілі.

2. Help Volunteer – ця платформа має ідентичний функціонал, що і її попередник. Що до спеціалізації платформи, тут все трохи більш збалансовано, платформа націлена на збори для медичних установ, на допомогу армії та постраждалим від конфлікту.

Порівнюючи ці платформи, можна виділити їх відмінну схожість, хоча й обидві чудово виконують свою ціль. Виходячи з цього, можна зробити висновок, що в розробці волонтерської платформи, одними з головних критеріїв є – безпека та довіра користувачів.

1.3 Етапи розробки веб-додатку

Проаналізувавши предметну область та переглянувши приклади вже існуючих волонтерських платформ, перш за все потрібно визначити базовий функціонал веб-додатку. В нього входить:

- Реєстрація або авторизація користувача;
- Створення та видалення зборів користувачем;
- Перегляд та можливість приймати участь у вже існуючих зборах;
- Створення адміністратора, який буде перевіряти та публікувати або видаляти створенні збори.

Перед проектуванням архітектури додатку та підбору технологій детальніше розглянемо етапи розробки веб-додатку.

Етапи розробки веб-додатку [3]:

1. Аналіз необхідного функціоналу веб-додатку: Перший етап передбачає вивчення потреб користувачів та вимог до функціоналу, які повинен мати веб-додаток. Його мета – чітко визначити функції та можливості, які повинен надавати веб-додаток, щоб відповідати потребам користувачів. Результатом цього етапу є розуміння функціональних вимог до веб-додатку, які будуть використанні в подальшому проектуванні та розробці.

2. Проектування архітектури веб-додатку з урахуванням функціональних вимог користувачів: Другий етап включає визначення структури веб-додатку для забезпечення ефективного виконання всіх функцій. Враховуються не лише функціональні вимоги користувачів, а й технічні аспекти, такі як масштабованість, надійність та безпека. Проектування архітектури включає розподіл системи на компоненти, визначення структури бази даних, бізнес-логіки та інших ключових елементів. Важливо враховувати можливість подальшого розширення функціоналу та інтеграції з іншими системами. Метою є створення гнучкої та масштабованої архітектури, яка відповідатиме поточним вимогам, зможе адаптуватися до майбутніх змін та вдосконалень. Особлива увага приділяється вибору технологій та інструментів, які забезпечать високу продуктивність та надійність системи.

3. Розробити функціонуючий веб-додаток на основі вибраних технологій: Третій етап містить в собі розробку повнофункціонального веб-додатку на основі обраних технологій. Використовуючи сучасні інструменти та фреймворки, потрібно створити динамічну платформу, яка буде відповідати вимогам користувачів. Потрібно зосередитись на створенні інтуїтивно зрозумілого інтерфейсу, адже простота та зрозумілість дизайну – важливі чинники для комфортного користування веб-додатком. З точки зору backend-у, метою є – створення надійної та масштабованої системи, здатної ефективно обробляти дані та бізнес-логіку.

4. Протестувати розроблений продукт: Останнім етапом розробки буде тестування розробленого продукту, адже це має важливе значення для

забезпечення його надійності та функціональності. Необхідно провести тестування frontend та backend частин. Frontend тестування забезпечить інтуїтивно зрозумілий та безперебійний користувацький досвід. Аналогічно, буде проведено тестування backend частини для забезпечення надійності та ефективності, що передбачає перевірку обробки даних та коректність роботи API.

РОЗДІЛ 2. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ВЕБ-ДОДАТКУ

2.1 Аналіз мов програмування та інструментів для розробки Frontend частини веб-додатку

Frontend частина веб-додатку – це частина, з якою безпосередньо відбувається взаємодія користувачів. Зазвичай вона написана за допомогою мов та технологій, таких як: HTML, CSS та JavaScript [4]. Ці технології необхідні для створення зручного та зрозумілого інтерфейсу. Але на додаток до цих основних мов та технологій, існує ряд інших інструментів і фреймворків, які розширюють функціонал та відкривають значно більший потенціал використовуваної мови або технології. Деякі з найпопулярніших інструментів і фреймворків для розробки Frontend частини включають в себе:

- React – це JavaScript бібліотека для створення користувацьких інтерфейсів. Вона відома своїм компонентним підходом, що дозволяє легко створювати код, який можна повторно використовувати та підтримувати [5].
- Angular – це TypeScript фреймворк для створення веб-додатків, розроблений командою Angular Team в компанії Google. Відомий своєю архітектурною концепцією, а саме ієрархією компонентів [6].
- Vue.js – це фреймворк JavaScript для створення інтерфейсів. Це легкий і прогресуючий фреймворк, що підходить для невеликих проектів [7].
- jQuery – це бібліотека JavaScript, яка спрощує маніпуляції з DOM та обробку подій. Це популярний вибір для frontend-розробників, оскільки він простий у вивченні та використанні [8].
- Bootstrap – це CSS-фреймворк для створення адаптивних та мобільних веб-сайтів. Це популярний вибір для frontend-розробників, оскільки він надає широкий спектр готових компонентів, завдяки чому економиться велика кількість часу на розробці клієнтської частини веб-додатку [9].

- Tailwind – це модульний CSS-фреймворк, який надає набір готових класів, які можна використовувати для стилізації елементів HTML. Це альтернатива традиційним CSS-фреймворкам, таким як Bootstrap. Tailwind є більш гнучким, ніж його аналог, оскільки він не надає жодних готових компонентів [10].

- SCSS(SASS) – це препроцесор CSS, що розшифровується як Syntactically Awesome Style Sheets. Він додає до CSS ряд функцій, які роблять його більш потужним і простим у використанні. Ці можливості включать в себе:

➢ Змінні: Змінні можна використовувати для зберігання значень, які можна повторно використовувати у файлі CSS.

➢ Вкладеність: Вкладеність дозволяє вкладати селектори CSS один в одного. Це може полегшити написання більш структурованого та організованого коду CSS.

➢ Міксини: Міксини – це багаторазові блоки CSS коду. Їх можна використовувати для створення спільних стилів, які можна повторно використовувати у всьому CSS-файлі.

➢ Функції: Функції дозволяють писати більш складний CSS-код. Вони можуть використовуватись для виконання різних операцій над значеннями CSS, таких як обчислення відсотків або кольорів [11].

Проаналізувавши можливі варіанти технологій, порівняємо їх між собою та виберемо найкращу, для нашого варіанту. Таким чином, перш за все, порівняємо та виберемо фреймворк або бібліотеку для написання JavaScript коду. Як було вище сказано, серед можливих варіантів є такі:

- React: Давайте виділимо переваги цієї технології:

➢ Компонентний підхід: React робить код більш модульним, придатним для багаторазового використання та легшим в підтримці.

➢ Віртуальний DOM: Віртуальний DOM робить React швидшим та ефективнішим, ніж інші фреймворки.

➢ Велика спільнота: React має широку низку спільнот, що дозволяє легко знайти допомогу та ресурси.

Недоліки:

- Великий поріг входження: Для того щоб вивчити React, потрібно затратити більше часу ніж на деякі інші фреймворки.
- Не підходить для малих проектів: React може бути занадто складним для малих проектів [12].

- Vue.js: Виділимо переваги цього фреймворку:

- Малий поріг входження: Vue.js легше вивчити ніж React та Angular.
- Гнучкість: Vue.js дуже гнучкий і може використовуватись для різноманітних проектів.
- Невеликий розмір: Vue.js має невеликий розмір, що робить його хорошим вибором для малих проектів.

Недоліки:

- Менша спільнота: Vue.js має меншу спільноту в порівнянні з React.
- Менш поширений: Vue.js не настільки поширений, як React або Angular, через що може виникнути складність в пошуку ресурсів або допомоги [13].

- Angular: Виділимо переваги цього фреймворку:

- Структура: Angular має жорстку структуру, що робить код більш організованим і легшим в обслуговуванні.
- Багато можливостей: Angular має багато вбудованих функцій, що робить його хорошим вибором для складних проектів.

Недоліки:

- Великий поріг входження: Angular досить важкий в вивченні та використанні.
- Менш гнучкий: Angular не такий гнучкий, як React або Vue.js, тому він може використовуватись не у всіх проектах [14].

- jQuery: Виділимо переваги цієї бібліотеки:

- Простота використання: jQuery дуже простий у вивчені та використанні.

- Багато плагінів: Існує багато плагінів для jQuery, які можуть розширити його функціональність.

Недоліки:

- Відсутність модулів: jQuery не є модульним, що може ускладнити підтримку великих проектів.
- Продуктивність: jQuery не є настільки продуктивним в порівнянні з іншими фреймворками, тому може не підходити для високопродуктивних додатків.
- Оновлення: jQuery досить стара бібліотека, тому вона може бути не найкращим вибором в порівнянні з новими та актуальними технологіями [15].

Для вибору найкращої технології оцінимо вимоги проекту. Оскільки, наш веб-додаток, не буде малим проектом і потрібно залишити можливість масштабування всієї системи, в такому випадку, ми не можемо використовувати Vue.js та jQuery. Залишилось порівняти React та Angular. Оскільки React має надто багато переваг над Angular, наприклад, велика спільнота, компонентний підхід та віртуальний DOM. Не зважаючи на його недолік, а саме: великий поріг входу, його можна компенсувати великою спільнотою, за допомогою якої можна буде швидко знайти допомогу або необхідну документацію. Зрештою, найкращим варіантом в нашому випадку буде – React.

Наступним кроком, потрібно обрати препроцесор або фреймворк для CSS. Для початку, розглянемо переваги та недоліки кожного з варіантів. Відповідно до інформації наведеної раніше, є такі варіанти:

- Bootstrap: Переваги:
- Легкість у вивченні та використанні: Bootstrap дуже простий у вивченні та використанні, навіть для початківців.
- Широкий вибір компонентів: Bootstrap надає широкий спектр готових компонентів, які можуть заощадити час і зусилля під час розробки.
- Адаптивний дизайн: Bootstrap призначений для створення адаптивних веб-сайтів, які добре виглядають на всіх пристроях, від десктопів до смартфонів.

Недоліки:

➢ Складність налаштування: Bootstrap трохи складний в налаштуванні для створення унікального вигляду веб-сайту.

➢ Можливе збільшення часу завантаження сторінки: Bootstrap може збільшити час завантаження сторінки, особливо для великих веб-сайтів.

➢ Відсутність гнучкості: Bootstrap не такий гнучкий, як інші фреймворки CSS, такі як Tailwind або SCSS [16].

- Tailwind: Переваги:

➢ Дуже гнучкий: Tailwind дуже гнучкий і може використовуватись для створення найрізноманітніших дизайнів.

➢ Легкість налаштування: Tailwind дуже легко налаштовувати для створення унікального зовнішнього вигляду веб-сайту.

➢ Швидкість розробки: Tailwind може пришвидшити процес розробки, особливо для невеликих проектів.

➢ Адаптивний дизайн: Tailwind, як й інші фреймворки та препроцесори можна використовувати для реалізації адаптивного дизайну.

Недоліки:

➢ Важкий для новачків: Tailwind має злегка важкий синтаксис, який може бути не зрозумілим для новачків.

➢ Ускладнення читабельності коду: Tailwind може створювати велику кількість класів в одному HTML елементі, через що значно ускладнюється код, який в подальшому важче підтримувати [17].

- SCSS: Переваги:

➢ Більш модульний та зручний для підтримки коду: SCSS може допомогти написати більш модульний і підтримуваний код CSS.

➢ Можливість повторного використання коду: SCSS допомагає писати більш багаторазовий CSS-код завдяки використанню змінних, міксинів та вкладеності.

➢ Нові можливості в коді: SCSS дає можливість користуватись такими функціями як: змінні, вкладеність, міксини, функції, прямо в CSS коді.

➢ Адаптивний дизайн: SCSS теж використовується для створення адаптивного дизайну.

Недоліки:

➢ Додаткові затрати на компіляцію: SCSS вимагає додаткового кроку збірки для компіляції коду SCSS в код CSS.

➢ Важкість вивчення: Вивчення SCSS може бути складнішим, ніж Bootstrap або Tailwind [18].

Проаналізувавши переваги та недоліки цих 3 технологій, можна дійти висновку, що Tailwind та Bootstrap слід використовувати для проектів малого або середнього розміру. В той час, як SCSS може бути універсальним. SCSS єдиний вводить новий функціонал в мову, тому він і є препроцесором. Важкість в вивченні цих 3 технологій, приблизно однакова. Оскільки, всі 3 технології не мають значної переваги між собою, потрібно обрати – SCSS. Він є – легким, швидким, масштабованим, додає новий зручний функціонал та дає змогу повторного використання коду.

2.2 Аналіз мов програмування та інструментів для розробки Back-End частини веб-додатку

Back-end частина веб-додатку – це частина, яка відповідає за зберігання, обробку даних і взаємодію із сервером. Зазвичай вона написана на серверній мові програмування, такій як Python, Java, Node.js, PHP або Ruby [19]. Ці мови необхідні для створення безпечного та масштабованого backend-а. Аналогічно frontend частині, тут, існує ряд інших інструментів і фреймворків для розробки backend частини, які можна використовувати для створення складних і зручних в підтримці веб-додатків. Одні з найпопулярніших інструментів і фреймворків включають в себе:

- Django: Django – це високорівневий фреймворк Python, який сприяє швидкій розробці та чистому, прагматичному дизайну. Побудований на основі мови Python, Django бере на себе більшу частину складнощів веб-розробки,

даючи можливість зосередитись на написанні коду. Django базується на архітектурному шаблоні MVC (model-view-controller), який розділяє код на три взаємопов’язані частини:

- Моделі – відповідають за представлення даних та управління ними.
- Представлення (views) – відповідають за відображення даних користувачам.
- Контролери – відповідають за обробку дій користувачів та делегування цих дій моделям і представленням [20].

- Laravel – це безкоштовний PHP-фреймворк з відкритим вихідним кодом для розробки веб-додатків. Laravel також базується на архітектурному шаблоні MVC [21].
- Spring Boot – це Java-фреймворк з відкритим вихідним кодом, який дозволяє легко створювати автономні веб-додатки. Spring Boot побудований на основі Spring Framework – комплексної Java-платформи, яка надає широкий спектр можливостей для корпоративної Java-розробки [22].
- Express.js – це популярний JavaScript-фреймворк для створення API та веб-додатків, він, відповідно, надає набір функцій для веб- і мобільних додатків та є мінімалістичним та гнучким [23].
- Ruby on Rails – це популярний фреймворк веб-додатків з відкритим вихідним кодом, написаний мовою Ruby. Ruby on Rails використовує архітектурний шаблон MVC. Також Rails відомий завдяки використанню принципу CoC (convention over configuration), який дає змогу зменшити кількість коду, що повторюється [24].

Вибір найкращої backend-технології для веб-додатку є надважливим завданням, яке відповідає за логіку на стороні сервера, зберігання даних і зв'язок з frontend частиною. Вибір найбільш коректної backend-технології має важливе значення для забезпечення продуктивності, масштабованості та безпеки додатку. Зважаючи на велику кількість доступних backend-технологій, кожна з яких має свої сильні та слабкі сторони, прийняття обґрунтованого рішення може бути

непростим завданням. Виділимо декілька факторів, на які слід звертати увагу при виборі backend-технології:

Тепер, вказавши фактори, на які слід звертати увагу при виборі найкращої backend-технології, потрібно порівняти переваги та недоліки кожної з них:

- Django: Переваги:

➢ Функціонал: Django слідує філософії «batteries-included», надаючи широкий спектр вбудованих функцій та модулів, включаючи автентифікацію, інтерфейс адміністратора та ORM (Object-Relational Mapper).

➢ Швидкість розробки: Django сприяє швидкій розробці завдяки високорівневим абстракціям та умовностям, що дозволяє зосередитися саме на логіці, а не на шаблонному коді.

➢ Спільнота: Потужна спільнота Django та велика документація дозволить легко знайти підтримку та ресурси для розробки.

➢ Розширені функції безпеки: Надійний захист від міжсайтового скриптингу (XSS), підробки міжсайтових запитів (CSRF) та клікджекінг атак.

Недоліки:

➢ Складність вивчення: Синтаксис і концепції Django потребують більшу кількість часу для вивчення.

➢ Несумісність з малими проектами: Для невеликих проектів з простими вимогами функціональності може бути надмірним [25].

- Laravel: Переваги:

➢ CLI (command-line interface): Laravel має інтерфейс командного рядка для ефективного управління додатками.

➢ ORM: Laravel має ORM, подібний до ORM Django, що дає можливість гнучко взаємодіяти з базами даних.

➢ Безпека: Laravel пропонує вбудовані функції безпеки для захисту додатків.

Недоліки:

➢ Знання PHP: Вимагає знання PHP для ефективного використання, що в свою чергу знижує швидкість розробки.

➢ Втрати продуктивності: Може мати меншу продуктивність у порівнянні з деякими іншими фреймворками.

➢ Документація: Документація Laravel може бути не такою вичерпною та інформативною, як в інших фреймворках, що іноді призводить до труднощів у пошуку рішень для конкретних проблем [26].

- Spring Boot: Переваги:

➢ Швидка розробка: Spring Boot спрощує конфігурацію та зменшує кількість шаблонного коду, прискорюючи розробку та заощаджуючи час.

➢ Гнучкість: Він має модульну та гнучку архітектуру, що дозволяє розробникам обирати компоненти відповідно до вимог проекту та легко інтегруватися з іншими проектами Spring.

➢ Вбудовані сервери: Spring Boot включає вбудовані сервери, що спрощує розгортання та усуває необхідність у зовнішній конфігурації сервера.

Недоліки:

➢ Важкість вивчення: Spring Boot має доволі важкий синтаксис та екосистему в цілому, що може призвести до труднощів в його вивченні.

➢ Продуктивність: Зручність вбудованих серверів може призвести до більшого споживання ресурсів, що необхідно враховувати в середовищах з обмеженими ресурсами [27].

- Express.js: Переваги:

➢ Швидкість: Однією з ключових переваг використання Express.js є його швидкість. Середовище виконання Node.js базується на циклі обробки подій, що легко і швидко обробляє декілька паралельних запитів.

➢ Легкість в використанні: Express.js - легкий і мінімалістичний, пропонує гнучкість і простоту для створення веб-додатків і API.

➢ Масштабованість: Може обробляти додатки з високим трафіком та ефективно використовувати ресурси.

➢ Гнучкість: Express.js спрощений і незалежний фреймворк, що дозволяє розробникам структурувати проекти відповідно до своїх уподобань і вимог.

Недоліки:

➢ Структура: Мінімалістичний характер може вимагати більше ручного налаштування в порівнянні з деякими фреймворками.

➢ Вбудований функціонал: Express.js не має вбудованих функцій у порівнянні з більш комплексними фреймворками, що змушує розробників покладатися на сторонні бібліотеки для певних функцій [28].

- Ruby on Rails: Переваги:

➢ Швидка розробка: Ruby on Rails надає перевагу конвенціям над конфігурацією (CoC), забезпечуючи просту і продуману структуру для швидкої розробки.

➢ Вбудований функціонал: Він включає такі вбудовані функції, як скаффолдинг, ORM та RESTful маршрутизація, що дозволяє розробникам швидко створювати та розгортати додатки.

➢ Велика спільнота: Ruby on Rails має активну спільноту та розгалужену екосистему бібліотек для розширення функціональності та вирішення різноманітних завдань.

Недоліки:

➢ Продуктивність: Ruby on Rails може бути ресурсоємним і менш продуктивним порівняно з більш легкими фреймворками, особливо для додатків з високим трафіком [29].

Відповідно до проведеного аналізу, кожна технологія бекенда має унікальні характеристики та відповідає певним сценаріям використання. Підсумуємо і виділимо сценарії використання кожної з технологій:

- Django: Django чудово підходить для сценаріїв, де швидка розробка та надійні функції мають першорядне значення. Його підхід «batteries-included» робить його чудовим вибором для створення складних веб-додатків з вбудованою автентифікацією, інтерфейсом адміністратора та ORM. Потужна підтримка спільноти та обширна документація Django ще більше підвищують його привабливість для проектів, що потребують масштабованості та підтримки.

- Laravel: Чистий синтаксис Laravel і широкий набір функцій, включаючи ORM і автентифікацію, сприяють швидкій розробці. Екосистема пакетів і

розширень Laravel також робить її придатною для проектів, що потребують інтеграції зі сторонніми сервісами та розширеннями.

- Spring Boot: Spring Boot ідеально підходить для розробки Java-додатків корпоративного рівня, які вимагають масштабованості, гнучкості та надійності. Його модульна архітектура та тісна інтеграція з іншими проектами Spring роблять його кращим вибором для розробників Java. Spring Boot чудово підходить для сценаріїв, де складна бізнес-логіка та управління транзакціями є критично важливими, що робить його придатним для великомасштабних корпоративних додатків.

- Express.js: Express.js добре підходить для сценаріїв, де перевага надається легким, мінімалістичним фреймворкам, наприклад, для побудови API та мікросервісів. Його простота і гнучкість дозволяють розробникам швидко створювати прототипи і розгортати веб-додатки без накладних витрат на повноцінний фреймворк. Архітектура проміжного програмного забезпечення Express.js також забезпечує легке налаштування та інтеграцію, що робить його ідеальним для проектів, які потребують гнучкості та масштабованості.

- Ruby on Rails: Ruby on Rails чудово підходить для сценаріїв, де пріоритетом є конвенціональність, а не конфігурація та швидка розробка. Його незалежний характер і вбудовані функції, включаючи скаффолдинг і ORM, спрощують процес розробки і зменшують кількість шаблонного коду. Ruby on Rails особливо підходить для стартапів та малих і середніх проектів, які прагнуть продуктивності та швидкого виходу на ринок.

API нашого майбутнього веб-додатку буде простим, адже в нас не велика кількість різноманітних даних, яку потрібно зберігати, в нашому випадку – це користувачі та збори, та операцій з ними. Також планується реалізувати авторизацію та реєстрацію.

Підсумовуючи все вище сказане, найоптимальнішим варіантом є – Express.js, оскільки, він добре підходить для легких та не великих проектів. Завдяки його простій та легкій архітектурі можна забезпечити легке

налаштування та інтеграцію, що ідеально підходить для проектів, де необхідна гнучкість та масштабованість.

2.3 Вибір бази даних

У сфері розробки веб-додатків, база даних слугує основою, репозиторієм, який ретельно зберігає та керує важливими даними програми. Вибір відповідної бази даних є рішенням першорядної важливості, оскільки від цього залежить продуктивність, масштабованість і загальна цілісність додатку.

Подібно до будівництва міцної будівлі, вибір правильної бази даних формує фундамент, на якому будується успішний веб-додаток. Правильно підібрана база даних легко інтегрується з веб-додатком, ефективно управляє зберіганням і пошуком даних, а також забезпечує цілісність інформації, яку вона захищає. І навпаки, погано підібрана база даних може призвести до зниження продуктивності, неузгодженості даних і перешкоджати використанню програми за призначенням.

Тому до процесу вибору бази даних не можна ставитися легковажно. Він вимагає ретельного аналізу вимог програми, характеристик даних і прогнозів щодо можливого масштабування системи. Ретельно оцінивши різні варіанти баз даних та узгодивши їх з потребами проекту, тільки тоді можна прийняти обґрунтоване рішення, яке забезпечить правильну роботу веб-додатку.

Але перш ніж розпочати процес вибору правильної бази даних, дуже важливо добре зрозуміти та засвоїти основні концепції та види баз даних, ознайомитись з різними типами баз даних, їхніми перевагами та недоліками. Сфера баз даних охоплює різноманітний набір технологій, кожна з яких розроблена для задоволення конкретних потреб управління даними та вимог веб-додатків. Розуміння ключових характеристик цих типів баз даних є важливим для того, щоб зробити правильний вибір. До основних типів баз даних належать бази даних SQL (реляційні бази даних), бази даних NoSQL (нереляційні бази даних), бази даних графів, бази даних документів і бази даних типу ключ-

значення. Зазначимо, що останні 3 типи баз даних є підтипами NoSQL баз даних, але про них ми розкажемо пізніше, зараз, ми розберемось, що ж таке реляційні та нереляційні бази даних, яка різниця між ними та виділимо їх переваги й недоліки.

Реляційна база даних - це тип бази даних, яка організовує дані в таблиці з рядків і стовпців. Рядки відповідають за запис, а стовпець це атрибут чи поле запису. Взаємозв’язок між таблицями встановлюються через ключі, зазвичай первинних і зовнішніх. Реляційні бази даних дотримуються принципів реляційної моделі, які включають такі поняття, як нормалізація, посилальна цілісність (referential integrity) і SQL (мова структурованих запитів) для запитів і маніпулювання даними.

Нереляційна база даних, також відома як NoSQL (Not Only SQL) - це тип бази даних, яка не дотримується традиційної реляційної моделі. Замість використання таблиць, рядків і стовпців, бази даних NoSQL використовують різні моделі даних, як було зазначено вище, такі як документи, пари ключ-значення, сімейства стовпців або структури на основі графів. Ці бази даних призначені для обробки неструктурованих, напівструктурованих або швидкозмінних даних, забезпечуючи гнучкість і масштабованість для сучасних додатків. Бази даних NoSQL часто надають перевагу масштабованості, продуктивності та відмовостійкості, а не суворій узгодженості та властивостям ACID (атомарність, узгодженість, ізоляція, довговічність). Вони зазвичай використовуються в розподілених, хмарних середовищах і середовищах великих даних, де традиційні реляційні бази даних можуть зіткнутися з обмеженнями [30].

Таким чином, основною різницею між SQL та NoSQL є те, що реляційні бази даних організовують дані в структуровані таблиці із заздалегідь визначеними схемами і зв'язками, а нереляційні бази даних використовують різні моделі даних, наприклад, документи, пари ключ-значення, сімейства стовпців або структури на основі графів. Давайте ж виділимо переваги та недоліки таких концепцій:

- Реляційні бази даних (SQL): Переваги:

➢ Структуровані дані: Реляційні бази даних організовують дані в таблиці із заздалегідь визначеними схемами, забезпечуючи цілісність та узгодженість даних.

➢ Властивості ACID: Вони дотримуються властивостей ACID (атомарність, узгодженість, ізольованість, довговічність), гарантуючи надійність транзакцій та цілісність даних.

➢ Стандартизована мова запитів: SQL (Structured Query Language - мова структурованих запитів) забезпечує стандартизований і потужний інтерфейс для запитів і маніпулювання даними.

➢ Цілісність даних: Реляційні бази даних забезпечують цілісність посилань за допомогою обмежень зовнішніх ключів, запобігаючи неузгодженості у зв'язках даних.

Недоліки:

➢ Складність: Управління складними реляційними схемами, нормалізацією та об'єднаннями може бути складним завданням, що призводить до збільшення витрат на розробку та обслуговування.

➢ Продуктивність: Операції об'єднання та складні запити можуть впливати на продуктивність, особливо при роботі з великими наборами даних або складними зв'язками між даними.

➢ Жорстка схема: Зміни в схемі бази даних, такі як додавання або модифікація стовпців, можуть бути громіздкими.

- Нереляційні бази даних (NoSQL): Переваги:

➢ Гнучкість: Нереляційні бази даних пропонують гнучкі схеми, що дозволяють зберігати неструктуровані, напівструктуровані або швидкозмінні дані.

➢ Масштабованість: Вони мають відмінну горизонтальну масштабованість.

- Продуктивність: Нереляційні бази даних часто забезпечують високу пропускну здатність і низьку затримку операцій читання і запису, що робить їх придатними для високопродуктивних додатків.

Недоліки:

- Відсутність стандартизації: У базах даних NoSQL відсутня стандартизована мова запитів, така як SQL, що призводить до потенційних проблем із запитами та маніпулюванням даними.
- Узгодженість даних: Деякі бази даних NoSQL жертвують надійними гарантіями узгодженості заради масштабованості, що призводить до того, що моделі узгодженості можуть не підходити для всіх додатків.
- Складність моделювання даних: Проектування та керування моделями даних у базах даних NoSQL може бути складним [30].

Проаналізувавши два типи баз даних, можна дійти висновку, що реляційні бази даних – пристосовані для використання в проектах, з великим обсягом даних, де структурованість, чіткість та можливість легко оперувати даними повинна бути на першому місці, нереляційні бази даних – слід ж використовувати в малих проектах, це дає великий потенціал для масштабованості та свободу від структурованих схем.

Оскільки, наш веб-додаток не буде великим і не планується використовувати велику кількість схем, то потрібно використовувати нереляційну базу даних, що дасть змогу використовувати гнучкі схеми і зберігати неструктуровані, напівструктуровані або швидкозмінні дані. Але постає ще одне питання, який ж підтип нереляційних баз даних вибрати, нагадаю їх існує 4 типи і вони використовують такі моделі даних, як:

- Документи: Ці бази даних зберігають дані у гнучких документах у форматі JSON і зазвичай використовуються для систем управління контентом, блог-платформ та аналітичних програм.
- Пари ключ-значення: Сховища ключ-значення зберігають дані у вигляді простих пар ключ-значення і часто використовуються для кешування, управління сеансами та розподіленого зберігання даних.

- Сімейства стовпців: Ці бази даних організовують дані в стовпці, а не в рядки, і підходять для часових рядів даних, аналітики та сховищ даних.
- Структури на основі графів: Графові бази даних використовують структури графів для семантичних запитів, щоб представляти та зберігати дані.

Найкращим з представлених варіантів буде перший, оскільки він має можливість зберігати документи в форматі, подібному JSON, а значення полів можуть включати інші документи, масиви або масиви документів.

Серед можливих варіантів нереляційних документо-орієнтованих баз даних є такі: MongoDB, CouchDB, Amazon DynamoDB, RavenDB, Apache Cassandra, Firebase Firestore, ArangoDB, Marklogic та OrientDB. Але ми розглянемо та порівняємо тільки найпопулярніші:

Таблиця 2.1 - Порівняльний аналіз найпопулярніших NoSQL баз даних

Назва	Продуктивність	Масштабованість	Надійність	Спільнота	Легкість використання	Ціна
MongoDB	Велика	Чудова горизонтальна	Чудова	Велика	Легко	Є безкоштовний план з потрібним функціоналом
CouchDB	Велика	Чудова горизонтальна	Середня	Середня	Легко	Є безкоштовний план з потрібним функціоналом
Amazon DynamoDB	Велика	Чудова	Чудова	Вище середнього	Легко	Є безкоштовний план з частковим функціоналом
Firebase Firestore	Велика	Чудова	Чудова	Велика	Дуже легко з використанням SDK	Є безкоштовний план з частковим функціоналом

Проаналізувавши можливі варіанти, вибір пав на MongoDB і ось чому:

- MongoDB є одною з найпопулярніших NoSQL баз даних. Дуже велика та активна спільнота забезпечує підтримку, оновлення та вирішення помилок.
- MongoDB за час свого існування стала відомою, завдяки своїй надійності, стабільності та масштабованості.
- MongoDB має безкоштовний план з усіма потрібними функціями.

2.4 Аналіз та вибір інтегрованого середовища розробки

У сфері веб-розробки вибір правильного інтегрованого середовища розробки (IDE) є не менш важливим. IDE слугує комплексним інструментарієм, що спрощує процес розробки та підвищує продуктивність. Зважаючи на велику кількість доступних IDE, кожна з яких може похвалитися унікальними можливостями та функціоналом, вибір найбільш вдалої може бути непростим завданням. Для того, щоб зробити правильний вибір, врахуємо такі критерії, при оцінці IDE:

- Сумісність з платформами: Забезпечення безперешкодної інтеграції IDE з необхідною операційною системою, будь то Windows, macOS або Linux.
- Набір функцій: Оцінка повного набору функцій IDE, включаючи інструменти для редагування коду, налагодження, тестування та розгортання.
- Інтерфейс користувача: Оцінка інтуїтивності, простоти використання та можливості налаштування IDE відповідно до особистих уподобань.
- Підтримка плагінів: Вивчення наявності плагінів і розширень, які можуть розширити функціональність IDE і задовольнити специфічні потреби розробників.
- Спільнота: Величина та активність спільноти IDE, надання цінних ресурсів та допомоги, коли це необхідно.

Серед можливих варіантів IDE слід розглянути такі: Visual Studio Code, WebStorm, IntelliJ IDEA, PHPStorm, Sublime Text.

Наступним кроком буде порівняння всіх цих IDE за критеріями, які ми виділили раніше:

- Visual Studio Code

➢ Сумісність з платформами: Windows, macOS, Linux.

➢ Набір функцій: VS Code пропонує багатий набір функцій, призначених для підвищення продуктивності та ефективності розробників. Він включає розширені можливості редагування коду з підтримкою завершення коду, рефакторингу та інтелектуальної навігації по коду. Крім того, VS Code просто інтегрується із Git та подібними системами, надаючи вбудовану підтримку Git і функції контролю версій вихідного коду.

➢ Інтерфейс користувача: Інтерфейс користувача Visual Studio Code є сучасним, чистим і легко налаштовується, що дозволяє розробникам пристосовувати IDE до своїх уподобань. Він має мінімалістичний макет з інтуїтивно зрозумілою навігацією та легким доступом до основних інструментів і команд. VS Code підтримує кілька тем і наборів іконок, що дозволяє користувачам персоналізувати середовище кодування відповідно до своїх уподобань.

➢ Підтримка плагінів: Visual Studio Code може похвалитися великою екосистемою розширень і плагінів, які розширюють його функціональність і задовольняють широкий спектр потреб розробки. Розробники можуть вдосконалити VS Code за допомогою додаткової мовної підтримки, інструментів налагодження, підвищення продуктивності та інтеграції з різними фреймворками і сервісами. VS Code Marketplace є централізованим центром для пошуку та встановлення розширень, що дозволяє легко налаштувати IDE відповідно до конкретних вимог.

➢ Спільнота: Велика та активна спільнота, яка в разі необхідності допоможе в вирішенні проблеми.

- WebStorm

➢ Сумісність з платформами: Windows, macOS, Linux.

➢ Набір функцій: WebStorm пропонує повний набір функцій, призначених для веб-розробки, включаючи інтелектуальне завершення коду, підсвічування синтаксису, рефакторинг коду і вбудовану підтримку популярних веб-технологій, таких як HTML, CSS, JavaScript, TypeScript і Node.js. Він також включає розширені інструменти налагодження, інтеграцію контролю версій і підтримку популярних фреймворків, таких як Angular, React і Vue.js.

➢ Інтерфейс користувача: Інтерфейс користувача WebStorm є інтуїтивно зрозумілим і зручним, забезпечуючи чистий і організований макет, що сприяє підвищенню продуктивності. Він має гнучкий редактор з підтримкою декількох вкладок, розділених представлень і згортання коду, що дозволяє розробникам ефективно орієнтуватися і керувати своїм середовищем розробки. Інтерфейс WebStorm легко налаштовується, дозволяючи користувачам змінювати теми, шрифти і колірні схеми відповідно до своїх уподобань.

➢ Підтримка плагінів: Хоча WebStorm має потужний набір вбудованих функцій, він також підтримує широкий спектр плагінів і розширень, які розширюють його функціональність. Розробники можуть розширити можливості WebStorm за допомогою додаткових інструментів, інтеграцій та мовної підтримки через репозиторій плагінів JetBrains. Незалежно від того, чи це додавання підтримки нових фреймворків або інтеграція зі сторонніми сервісами, екосистема плагінів WebStorm пропонує гнучкість і можливості налаштування.

➢ Спільнота: Велика та активна спільнота, не менша в порівнянні з VS Code.

- IntelliJ IDEA

➢ Сумісність з платформами: Windows, macOS, Linux.

➢ Набір функцій: IntelliJ IDEA пропонує повний набір функцій, включаючи інтелектуальне завершення коду, інструменти рефакторингу, інтеграцію контролю версій (наприклад, Git, SVN) і вбудовану підтримку популярних фреймворків, таких як Spring, Hibernate і JavaFX. Він також включає в себе розширені функції продуктивності, такі як перевірка коду та потужний відладчик.

➢ Інтерфейс користувача: Інтерфейс користувача IntelliJ IDEA дуже інтуїтивно зрозумілий і налаштовується, що дозволяє розробникам адаптувати IDE до своїх уподобань і робочих процесів. Він має чистий і організований макет з легким доступом до різних інструментів і панелей, що сприяє створенню безладного середовища для кодування. IDE надає підтримку темних і світлих тем, а також опції налаштування шрифтів і кольорів.

➢ Підтримка плагінів: IntelliJ IDEA може похвалитися великою екосистемою плагінів, які розширюють її функціональність і задовольняють специфічні потреби розробки. Розробники можуть розширити можливості IDE за допомогою додаткових інструментів, інтеграцій та мовної підтримки через репозиторій плагінів IntelliJ. IDE також підтримує розробку і розгортання користувацьких плагінів, що дозволяє користувачам адаптувати IntelliJ IDEA до своїх унікальних вимог.

➢ Спільнота: Велика та досить активна спільнота, можливо трішки менша ніж в інших IDE.

- PHPStorm

➢ Сумісність з платформами: Windows, macOS, Linux.

➢ Набір функцій: PHPStorm пропонує повний набір функцій, спеціально розроблених для розробки PHP. Він включає розширені можливості редагування коду з підтримкою підсвічування синтаксису, завершення коду, рефакторингу та інтелектуального аналізу коду. Крім того, PHPStorm надає вбудовану підтримку популярних PHP-фреймворків, таких як Laravel, Symfony і Yii, а також інтеграцію з системами контролю версій.

➢ Інтерфейс користувача: Інтерфейс користувача PHPStorm інтуїтивно зрозумілий і зручний, розроблений для підвищення продуктивності розробників. Він має налаштовуваний редактор з підтримкою декількох вкладок, розділених представлень і згортання коду, що дозволяє розробникам ефективно переміщатися і керувати своїми PHP-проектами. Інтерфейс PHPStorm також включає в себе інструменти для налагодження та тестування PHP-додатків.

➢ Підтримка плагінів: PHPStorm підтримує широкий спектр плагінів і розширень, які розширюють його функціональність і задовольняють специфічні потреби розробки. Розробники можуть розширити можливості PHPStorm за допомогою додаткових інструментів, інтеграцій та мовної підтримки через репозиторій плагінів JetBrains. Незалежно від того, чи це додавання підтримки нових PHP-фреймворків або інтеграція зі сторонніми сервісами, екосистема плагінів PhpStorm пропонує гнучкість і можливості налаштування.

➢ Спільнота: Велика та активна, така ж як і в WEBStorm.

- Sublime Text

➢ Сумісність з платформами: Windows, macOS, Linux.

➢ Набір функцій: Sublime Text пропонує багатий набір функцій, призначених для покращення редагування коду та підвищення продуктивності. Він включає всі ті ж розширені можливості редагування код. Також, Sublime Text надає вбудовану підтримку систем контролю версій.

➢ Інтерфейс користувача: Користувацький інтерфейс Sublime Text чистий, мінімалістичний і легко налаштовується, що дозволяє розробникам пристосувати редактор до своїх уподобань.

➢ Підтримка плагінів: Sublime Text може похвалитися яскравою екосистемою плагінів і розширень, які розширюють його функціональність і задовольняють специфічні потреби розробників. Розробники можуть розширити можливості Sublime Text за допомогою додаткових інструментів, мовної підтримки та інтеграції через менеджер пакетів Package Control. Розгалужена екосистема плагінів пропонує гнучкість і можливості налаштування для задоволення різноманітних вимог розробників.

➢ Спільнота: Середнього розміру, але також активна [31,32].

Отже, аналіз інтегрованих середовищ розробки (IDE), таких як IntelliJ IDEA, Visual Studio Code, WebStorm, PhpStorm та Sublime Text, показує, що вони мають багато спільних рис з точки зору функціональності та можливостей. Хоча кожна IDE може мати свої унікальні сильні сторони та сфери спеціалізації, основний набір функцій, необхідний для сучасної розробки програмного

забезпечення, присутній у всіх них. Зрештою, вибір IDE залежить від індивідуальних уподобань, тому мій вибір пав на – Visual Studio Code.

2.5 Вибір інструменту для тестування API

Останнім кроком в підготовці до розробки веб-додатку, буде вибір інструменту для тестування API. Вибираючи інструмент для тестування API, важливо враховувати кілька факторів, щоб переконатися, що він відповідає вимогам проекту і сприяє ефективному процесу тестування. Виділимо важливі критерії:

- Можливість надсилати HTTP-запити: Інструмент повинен дозволяти тестувальникам створювати та надсилати різні типи HTTP-запитів, такі як GET, POST, PUT, DELETE тощо, для взаємодії з кінцевими точками API.
- Підтримка аутентифікації: Він повинен підтримувати різні методи аутентифікації, такі як базова аутентифікація, OAuth, ключі API тощо, щоб забезпечити безпечний доступ до кінцевих точок API під час тестування.
- Параметризація та тестування на основі даних: Інструмент повинен дозволяти тестувальникам параметризувати запити і відповіді та проводити тестування на основі даних, використовуючи змінні, джерела даних і динамічну генерацію даних.
- Перевірка відповідей: Інструмент повинен надавати механізми для перевірки відповідей API на відповідність очікуваним результатам, включаючи коди статусу, заголовки відповідей, вміст тіла відповіді та перевірку JSON/XML-схем.
- Підтримка автоматизації: Інструмент повинен надавати можливості для автоматизації тестування, дозволяючи тестувальникам створювати і виконувати автоматизовані тести API.

Серед популярних варіантів таких інструментів є такі: Postman, Insomnia, SoapUI, Paw. Порівняємо їх за даними критеріями та оберемо найкращий:

- Postman:

➢ Плюси: Інтуїтивно зрозумілий користувацький інтерфейс, підтримка різних типів запитів, методів автентифікації та написання сценаріїв (на основі JavaScript), широкі можливості збору даних і керування середовищем, вбудована програма для запуску тестів і функції для спільної роботи.

➢ Мінуси: обмежені можливості автоматизації у безкоштовній версії, розширені функції вимагають платної підписки.

- Insomnia:

➢ Плюси: Відкритий і безкоштовний, сучасний і зручний інтерфейс, підтримує різні методи автентифікації, динамічні змінні і середовища, підтримка GraphQL і системи плагінів.

➢ Мінуси: Обмежені можливості командної роботи порівняно з Postman, менше вбудованих інтеграцій та підтримки спільноти.

- SoapUI:

➢ Плюси: Надійна підтримка SOAP та REST API, широкі можливості тестування, включаючи функціональне, навантажувальне та безпекове тестування, інтеграція з інструментами CI/CD.

➢ Мінуси: значно складніший інтерфейс, важкий і ресурсоємний, обмежена підтримка не-SOAP API.

- Paw:

➢ Плюси: Інтуїтивно зрозумілий і візуально привабливий інтерфейс, підтримує різні методи автентифікації, динамічні середовища, розширені можливості написання сценаріїв на JavaScript і гнучке налаштування запитів/відповідей.

➢ Мінуси: доступне лише для macOS, платне програмне забезпечення без безкоштовної версії, бракує широких можливостей автоматизації порівняно з іншими інструментами [33].

Зваживши всі за та проти, слід обрати Insomnia, адже він має чудовий безкоштовний план, широкий спектр функціоналу, а обмежені можливості командної роботи порівняно з Postman, не завадять якісному тестуванню API в нашому випадку.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ВОЛОНТЕРСЬКОЇ ВЕБ-ПЛАТФОРМИ

3.1 Проектування та аналіз функціоналу веб-платформи

Наступним етапом в процесі розробки веб-додатку, буде проектування його архітектури. Процес проектування можна розбити на такі етапи:

- Аналіз вимог до функціоналу.
- Проектування логічної моделі веб-додатку.
- Проектування бази даних.
- Вибір програмної архітектури.
- Розробка блок-схеми веб-додатку

Логічна модель слугує планом нашого веб-додатку, визначаючи його функціональні можливості. Оскільки ми вже визначили базовий функціонал нашого веб-додатку раніше, то нагадаємо його основні функції:

- Додавання\видалення зборів.
- Перегляд вже існуючих зборів.
- Фільтрація зборів за типом.
- Логін\реєстрація користувача.
- Роль адміністратора, який переглядає заявки на створення зборів, приймає їх або відхиляє.

Якщо представити функціонал нашого веб-додатку в вигляді можливих запитів до серверу та продемонструвати їх за допомогою інструменту для тестування API Insomnia, це буде виглядати як на рисунку 3.1.

Одним з найголовніших критеріїв, все ще залишається безпека. Тому для забезпечення надійної передачі даних, ми застосовуємо HTTPS протокол замість HTTP, що дає змогу шифрувати дані, що передаються. HTTPS працює на шифрованих механізмах, що допомагає суттєво захистити з’єднання й перевірити справжність сайту. Зазначимо, що використовують сертифікати SSL (скорочення від «secure sockets layer») та TLS (transport layer security (протокол захисту транспортного рівня)). Безпечне зашифроване з’єднання між браузером

користувача та сервером для захисту рівня зв’язку створюється за рахунок протоколу передачі даних. TLS ж використовує сучасні криптографічні методи, які гарантують, що [34]:

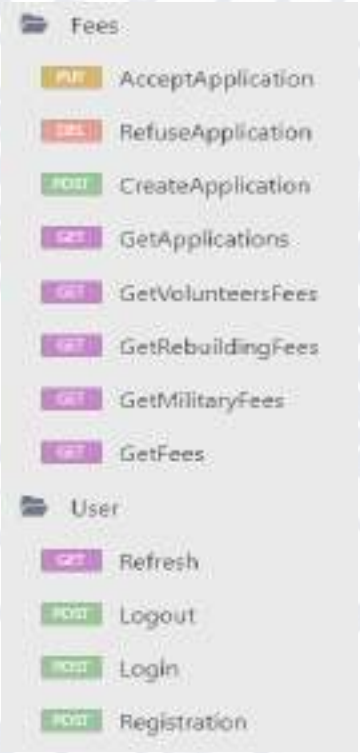


Рисунок 3.1 – Можливі запити до веб-серверу

- Дані не були підроблені після їх надсилання.
- З’єднання відбувається з користувачем, що надіслав запит.
- Особисті дані будуть захищеними.

Наступним кроком спроектуємо базу даних відповідно до функціональних вимог. Зрештою, можна виділити три колекції, які ми будемо зберігати в нашій базі даних:

- Збори: в цій колекції буде зберігатись інформація про збори, така як: унікальний ідентифікаційний номер, електрона пошта користувача, який створив збір, тип збору, назва збору, опис збору, необхідна сума для збору, реквізити, дата закінчення збору, фото для шапки збору та інформація про стан збору, тобто чи був збір прийнятий адміністратором чи ні (див. рис. 3.2).

```
3   const Fee = new Schema({
4     email: { type: String, required: true },
5     type: { type: String, required: true },
6     name: { type: String, required: true },
7     description: { type: String, required: true },
8     sum: { type: Number, required: true },
9     requisite: { type: String, required: true },
10    finish: { type: String, required: true },
11    isAccepted: { type: Boolean, required: true },
12    imageSrc: { type: Buffer, required: true },
13  });
```

Рисунок 3.2 – Схема колекції «ЗБОРИ»

- Користувачі: в цій колекції буде зберігатись інформація про користувача: унікальний ідентифікаційний номер, логін, зашифрований пароль та роль користувача, а саме, чи є він адміністратором (див. рис. 3.3).

```
3   const userSchema = new Schema({
4     login: { type: String, unique: true, required: true },
5     password: { type: String, required: true },
6     isAdmin: { type: Boolean, required: true },
7   });
```

Рисунок 3.3 – Схема колекції «Користувач»

- JWT token: в цій колекції буде зберігатись інформація про refresh токени користувачів, з такими полями: унікальний ідентифікаційний номер, посилання на об’єкт користувача, а саме, кому належить цей токен та сам refresh токен (див. рис. 3.4).

```
3   const tokenSchema = new Schema({
4     user: { type: Schema.Types.ObjectId, ref: "User" },
5     refreshToken: { type: String, required: true },
6   });
```

Рисунок 3.4 – Схема колекції «JWT Token»

Зазначимо, що унікальні ідентифікаційні номери для записів в кожній колекції, Mongo DB створює власноруч, що робить процес налаштування схем колекцій більш легким.

Обираючи архітектурний патерн для веб-додатку, було прийнято рішення використовувати патерн MVC (Model-View-Controller). Цей патерн відповідає вимогам проекту та має численні переваги:

- Перш за все, патерн MVC дозволяє чітко розділити логіку веб-додатку на три основні компоненти: Модель (Model), Вид (View) та Контролер (Controller). Це спрощує процес розробки, оскільки кожен компонент відповідає за свої конкретні функції, що полегшує тестування та підтримку коду.
- Друга важлива перевага полягає в тому, що патерн MVC сприяє відокремленню бізнес-логіки від представлення даних та взаємодії з користувачем. Це забезпечує більшу гнучкість та розширюваність додатку, оскільки зміни в одному компоненті не впливають безпосередньо на інші [35].

Використовуючи MVC патерн, в нашому випадку, кожен з його компонентів можна охарактеризувати так:

- Model – відповідає за управління даними, логікою бізнес-процесів та взаємодію з базою даних. Наприклад, feeController або feeService.
- View – представляє інформацію користувачу та відповідає за візуальне відображення даних. Наприклад, сторінка веб-додатку.
- Controller – обробляє запити користувачів, взаємодіє з Model та View, та визначає, які дії виконуються при кожному запиті. Наприклад, feeRouter.

Загалом, вибір патерну MVC є обґрунтованим рішенням, оскільки він спрощує розробку, полегшує підтримку коду та забезпечує більшу гнучкість та розширюваність.

Для наглядної демонстрації очікуваного функціоналу нашого веб-додатку, побудуємо use case diagram-у (див. рис. 3.5).

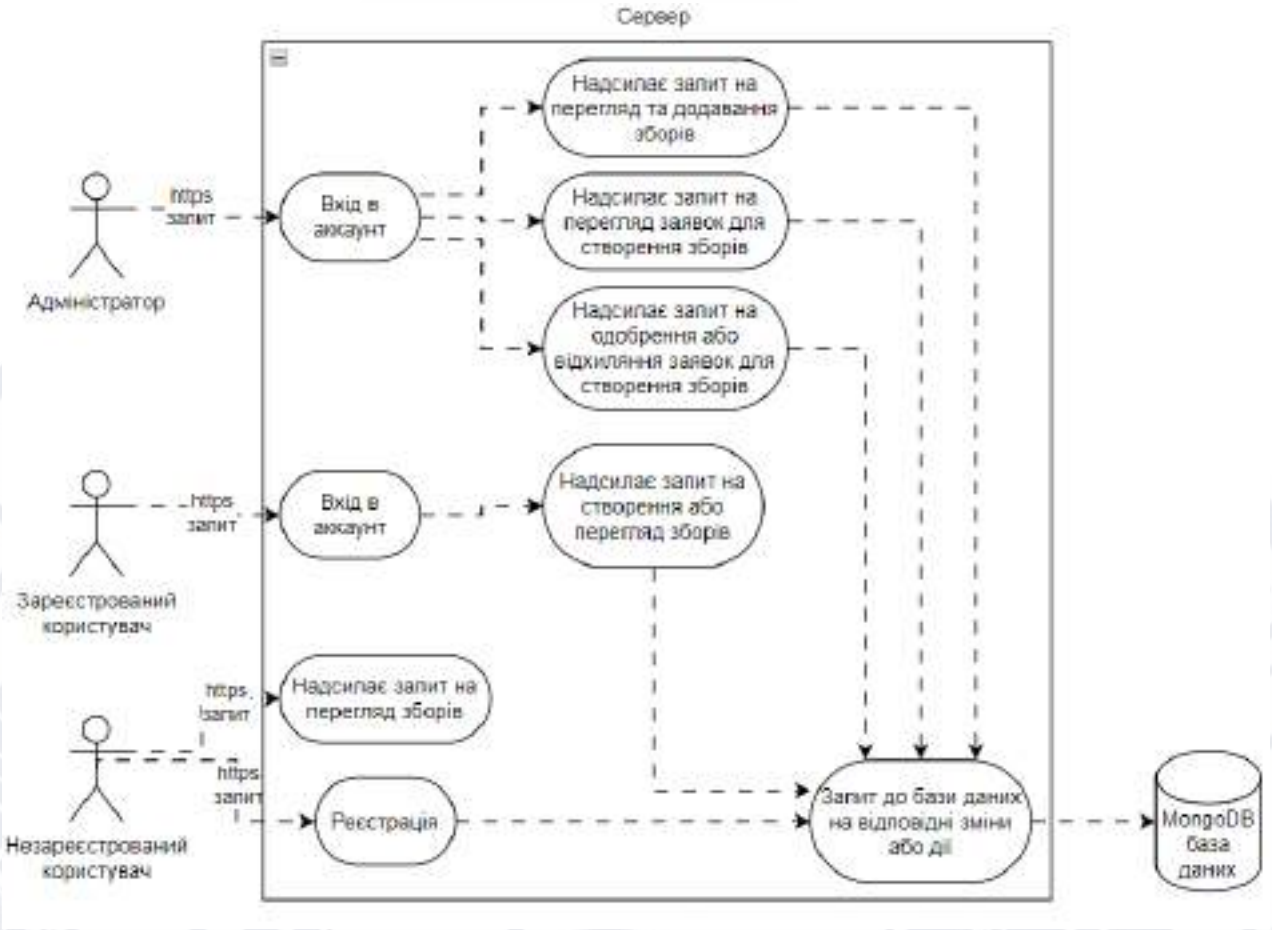


Рисунок 3.5 – Use Case Diagram веб-додатку

Ця діаграма показує, які дії можуть виконувати різні типи користувачів у системі. Наприклад, на ній видно, що звичайні користувачі можуть реєструватися входити в систему, переглядати доступні збори та створювати нові. Адміністратори ж мають додаткові можливості, такі як перевірка нових зборів, їх ухвалення або видалення.

3.2 Розробка серверної частини веб-платформи

Першим кроком в процесі розробки backend частини, буде ініціалізація та налаштування проекту. Створивши папку для нашого проекту, відкриваємо її в терміналі, після чого ініціалізуємо проект командою: *npm init –y*. Після завершення процесу ініціалізації, потрібно встановити необхідні для роботи

пакети, що можна зробити за допомогою команди: *npm i <назва пакету>*. В нашому випадку, необхідно встановити такі пакети:

- Express.js – це JavaScript-фреймворк, на якому буде базуватись наш фронтенд.
- Express-validator – це набір інструментів для перевірки коректності даних, які надходять від користувачів.
- Cors – це набір інструментів для налаштування CORS (Cross-Origin Resource Sharing) механізмів, що дозволяє серверу визначити, які джерела мають дозвіл на доступ до його ресурсів.
- Cookie-parser – це набір інструментів для парсингу та налаштування cookie файлів, які надсилаються клієнтом у HTTP-запитах.
- Dotenv – це бібліотека, що дає можливість завантажувати змінні середовища з файлу *.env* в середовище процесу Node.js.
- Fs – є одним з основних модулів, який дає змогу працювати з файловою системою: додавати, видаляти та переміщати файли в папці проекту.
- Jsonwebtoken – це бібліотека, що надає можливість створювати JWT токени.
- Mongoose – це бібліотека для взаємодії з базою даних MongoDB.
- Multer – це middleware для обробки файлів, які користувачі відправляють через форми на сторінках веб-додатків.
- Nodemon – це інструмент, який допомагає автоматично перезапускати Node.js сервер при внесенні змін у програмний код, що дозволяє значно спростити процес розробки, без перезапуску серверу при кожній зміні коду.
- Uuid – це бібліотека, що використовується для створення унікальних ідентифікаторів.
- Bcryptjs – це бібліотека, що дозволяє хешувати паролі перед їх збереженням у базі даних.

Після встановлення необхідних пакетів, потрібно змінити деякі налаштування в файлі package.json, а саме вказати скрипт, який буде запускати

наш сервер, назву проекту, за необхідності, та файл проекту, з якого буде розпочинатись запуск серверу. Результат показаний на рисунку 3.6:

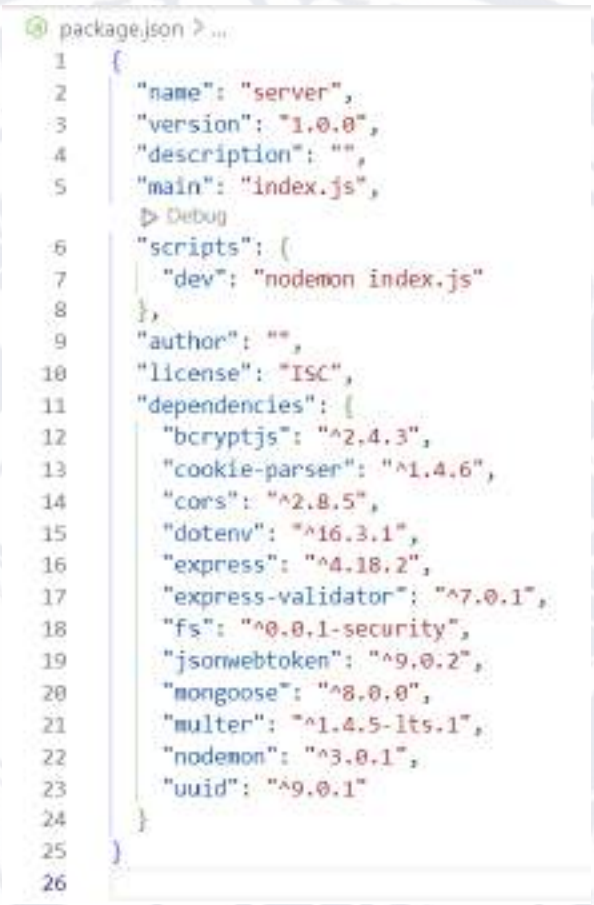


Рисунок 3.6 – Вигляд файлу package.json після налаштування

Наступним кроком потрібно створити головний файл в корінній папці проекту, а саме index.js. Після чого, імпортуємо необхідні пакети та пишемо необхідний код для запуску серверу та з’єднання з базою даних (див. рис. 3.7).

Далі запускаємо наш сервер командою *npm run dev* та перевіряти результат виконаної роботи.

Після підготовчих етапів потрібно реалізувати весь необхідний функціонал по взаємодії клієнта з сервером. Перш за все створимо моделі для кожної з колекцій бази даних, а далі контролер, сервіс та роутер для них. Приклад створення моделі показаний на рисунку 3.8.

```
require("dotenv").config();
const express = require("express");
const cors = require("cors");
const cookieParser = require("cookie-parser");
const mongoose = require("mongoose");
const PORT = process.env.PORT || 5000;
const app = express();

app.use(express.json());
app.use(cookieParser());
app.use(
  cors({
    credentials: true,
    origin: [
      "http://localhost:5173",
      "https://soldiers-hub-project-iitgohiom-botsyundenys.vercel.app",
      "https://soldiers-hub-project.vercel.app",
    ],
  })
);

const start = async () => {
  try {
    await mongoose.connect(process.env.DB_URL, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    app.listen(PORT, () => console.log(`server was started on PORT = ${PORT}`));
  } catch (error) {
    console.log(error);
  }
};

start();
```

Рисунок 3.7 – Код файлу index.js

```
const { Schema, model } = require("mongoose");

const Fee = new Schema({
  email: { type: String, required: true },
  type: { type: String, required: true },
  name: { type: String, required: true },
  description: { type: String, required: true },
  sum: { type: Number, required: true },
  requisite: { type: String, required: true },
  finish: { type: String, required: true },
  isAccepted: { type: Boolean, required: true },
  imageSrc: { type: Buffer, required: true },
});

module.exports = model("Fee", Fee);
```

Рисунок 3.8 – Приклад створення моделі даних для БД

Аналогічно цьому прикладу створюємо й інші моделі: user та token. Після чого можна перейти до написання сервісу (див. рис. 3.9).

```
const Fee = require("../Models/feeModel");

class FeeService {
  async createApplication(fee) {
    const createdFeeApplication = await Fee.create(fee);
    return createdFeeApplication;
  }

  async getApplicationsFee() {
    const feeApplications = await Fee.find({ isAccepted: false });
    return feeApplications;
  }

  async acceptFeeApplication(id) {
    const acceptedFee = await Fee.findByIdAndUpdate(id, {
      isAccepted: true,
    });
    return acceptedFee;
  }

  async refuseFeeApplication(id) {
    const refusedFee = await Fee.findByIdAndDelete(id);
    return refusedFee;
  }
}

module.exports = new FeeService();
```

Рисунок 3.9 – Часткова реалізація основних функцій FeeService

На прикладі асинхронної функції createApplication розберемо код детальніше. Ця функція створює не підтверджений збір, тобто збір, який повинен переглянути адміністратор. Параметром ця функція приймає об’єкт збору, який містить всю необхідну інформацію відповідно до моделі Fee, створеної раніше. І в 1 рядку функції ми викликаємо метод моделі create, передаючи в нього об’єкт з інформацією збору, та створюємо новий запис в базі даних.

Наступним кроком потрібно реалізувати контролер, який буде приймати дані від користувача, обробляти їх та передавати вже готовий сформований об’єкт даних в сервіс. Реалізація показана на рисунку 3.10.

```
1  const FeeService = require("../Services/feeService");
2  const fs = require("fs");
3
4  class FeeController {
5    async createApplication(req, res) {
6      try {
7        const imageSrc = fs.readFileSync("uploads/" + req.file.filename);
8        const fee = await FeeService.createApplication({ ...req.body, imageSrc });
9        return res.json(fee);
10     } catch (e) {
11       res.status(500).json(e);
12     }
13   }
14
15   async getApplications(req, res) {
16     try {
17       const applications = await FeeService.getApplicationsFee();
18       return res.json(applications);
19     } catch (e) {
20       res.status(500).json(e);
21     }
22   }
23 }
24
25 module.exports = new FeeController();
26
```

Рисунок 3.10 – Часткова реалізація основних функцій FeeContoller

Вище було продемонстровано лише 2 функції з FeeContoller, для того щоб показати та пояснити їх роботу, всі інші функції мають схожу реалізацію. Функції отримують два параметри, це request (запит) та response (відповідь). В тілі запиту знаходяться дані, які користувач відправляє на сервер, а в відповіді – необхідні для клієнта дані. В функції createApplication, а саме в 7 рядку, ми бачимо зчитування, надісланого користувачем, зображення в буфер. Буфер – це спеціальний тип об’єкта в Node.js, який використовується для представлення бінарних даних [36]. Далі ми бачимо виклик функції createApplication з класу FeeService, який створений для взаємодії з базою даних. В параметр ми передаємо об'єкт даних, що відповідає створеній моделі. В разі виникнення помилки, її обробку забезпечує try-catch блок, який повертає інформацію про помилку у відповіді на запит.

Тепер необхідно реалізувати роутер, який буде викликати відповідні, до запиту користувача, функції з класу FeeController (див. рис. 3.11).

```
1  const Router = require("express").Router;
2  const FeeController = require("../Controllers/feeController");
3  const { upload } = require("../Services/uploadService");
4  const feesRouter = new Router();
5
6  feesRouter.post(
7    "/createApplication",
8    upload.single("image"),
9    FeeController.createApplication
10 );
11 feesRouter.get("/applications", FeeController.getApplications);
12 feesRouter.put("/acceptApplication/:id", FeeController.acceptFeeApplication);
13 feesRouter.delete("/refuseApplication/:id", FeeController.refuseFeeApplication);
14 feesRouter.get("/getVolunteersFees", FeeController.getVolunteersFees);
15 feesRouter.get("/getMilitaryFees", FeeController.getMilitaryFees);
16 feesRouter.get("/getRebuildingFees", FeeController.getRebuildingFees);
17 feesRouter.get("/getDonnuFees", FeeController.getDonnuFees);
18 feesRouter.get("/getFees", FeeController.getFees);
19
20 module.exports = feesRouter;
21
```

Рисунок 3.11 – Створення роутера для зборів

Імпортувавши всі необхідні модулі та пакети, в рядку 4 ми створюємо новий екземпляр класу Router, після чого створюємо маршрути для запитів використовуючи feesRouter. Першим вказується клас, далі метод цього класу, який вказує на вид HTTP-запиту. Метод приймає 2 параметри: 1 – це шлях, який буде активовувати цей маршрут, а 2 – це функція, яка буде викликана після активування маршруту. Для прикладу, візьмемо маршрут, який створено в рядку 11, якщо користувач зробить запит по такому маршруту *server:port/applications*, то буде викликана функція getApplications класу FeeContoller. Також, зазначимо, в проекті використовуємо 4 різні види HTTP-запитів, це:

- GET – цей запит використовується для отримання даних з серверу.
- POST – використовується при відправці даних на сервер для створення нових записів в БД.
- PUT – використовується для оновлення запису.
- DELETE – використовується для видалення запису.

Після завершення основної частини розробки, перш за все потрібно активувати всі роутери в головному файлі проекту index.js. Це робиться методом

use класу app, наприклад, app.use(‘/fee’, feeRouter). Перед цим, обов’язково потрібно імпортувати створенні нами роутери (див. рис. 3.12).

```
app.use(express.json());
app.use(cookieParser());
app.use(
  cors({
    credentials: true,
    origin: [
      "http://localhost:5173",
      "https://soldiers-hub-project-1itgohixm-botsyundenys.vercel.app",
      "https://soldiers-hub-project.vercel.app",
    ],
  })
);
app.use("/auth", authRouter);
app.use("/fee", feeRouter);
```

Рисунок 3.12 – Активовані маршрути

В процесі може виникнути питання, чому маршрути були розбиті на різні роутери, такі як feeRouter або authRouter. Розбивка маршрутів на окремі файли є рекомендованою практикою в веб-розробці, оскільки вона дає декілька переваг [37]:

- Модульність.
- Організованість коду.
- Легша підтримка.
- Масштабованість.

Останнім етапом розробки backend-у буде реалізація обробника помилок для певних випадків, адже, в даний момент, try-catch блок повертає просто помилку без всілякої конкретики. Наприклад, якщо користувач під час авторизації введе не правильний пароль, повернеться просто помилка без конкретної причини та коду, а нам потрібно, щоб поверталась помилка з кодом 400 та повідомленням про неправильний пароль. Тому наявність такого обробника в проекті є важливим, приступимо до його реалізації. Створимо нову папку ‘Exceptions’, в ній створимо файл ‘ApiError.js’ та напишемо необхідний код (див. рис. 3.13).

```
module.exports = class ApiError extends Error {
  status;
  errors;

  constructor(status, message, errors) {
    super(message);
    this.status = status;
    this.errors = errors;
  }

  static UnauthorizedError() {
    return new ApiError(401, "User isn`t authorized");
  }

  static BadRequest(message, errors = []) {
    return new ApiError(400, message, errors);
  }
};
```

Рисунок 3.13 – Реалізація класу ApiError

На рисунку 3.13 продемонстрована реалізація класу ApiError, яка розширює стандартний клас Error. Конструктор цього класу, розширює конструктор початкового класу Error та має ще два додаткових поля: статус та помилки. В класі реалізовано два методи: 1 – повертає помилку з кодом 401 та фіксованим повідомленням помилки, а 2 – повертає помилку з кодом 400, повідомленням, яке отримав через параметри, та масивом помилок. Далі потрібно реалізувати middleware, який буде перевіряти чи є помилка екземпляром класу ApiError чи ні, якщо так – то відображати необхідну інформацію про помилку, якщо ні – відображати помилку з кодом 500 та повідомлення ‘Something went wrong…’ (див. рис. 3.14).

Для того, щоб цей middleware запрацював, його потрібно підключити, використовуючи метод use класу app, в головному файлі проекту. Зробити це потрібно відразу після останнього використання app.use().

```
const ApiError = require("../Exceptions/apiError");

module.exports = function (err, req, res, next) {
  if (err instanceof ApiError) {
    return res
      .status(err.status)
      .json({ message: err.message, errors: err.errors });
  }
  return res.status(500).json({ message: "Something went wrong..." });
};
```

Рисунок 3.14 – Реалізація errorMiddleware

3.3 Тестування серверної частини веб-платформи

Протестуємо серверну частину (backend) платформи за допомогою API Insomnia. Перш за все протестуємо створення збору користувачем. Вказуємо всю необхідну інформацію та робимо запит (див. рис. 3.15).

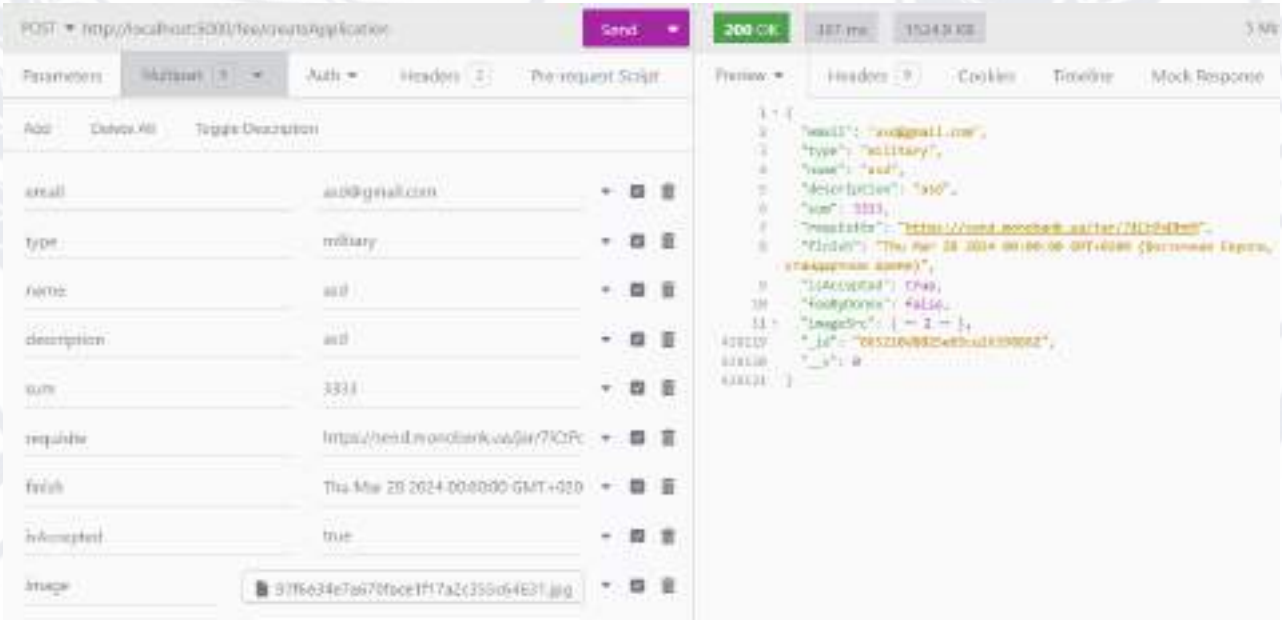


Рисунок 3.15 – Запит на створення збору

Отримуємо відповідь від серверу, в якій вказана вся інформація про збір, що вже була збережена в базі даних. Далі давайте схвалимо цей збір від імені

адміністратора, але в цьому випадку параметр потрібно передати в адресі URL. Як саме, показано на рисунку 3.16.

Рисунок 3.16 – Запит на підтвердження збору адміністратором

В відповіді від серверу, ми отримуємо дані оновленого збору. Наступним, перевіримо запит на отримання всіх зборів, підтверджених адміністратором (див. рис. 3.17).

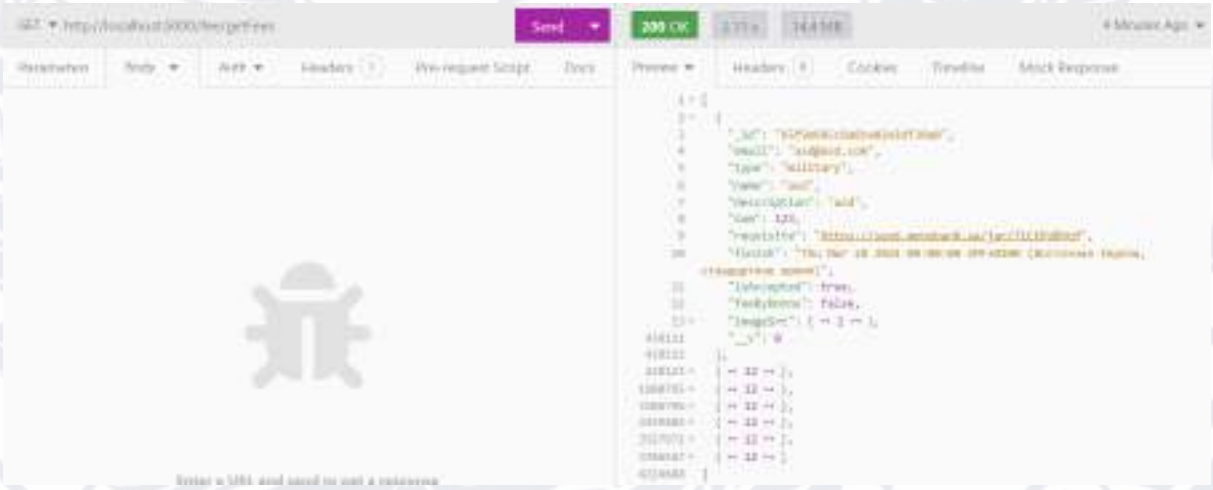


Рисунок 3.17 – Запит на отримання всіх підтверджених зборів

В відповіді ми отримуємо масив об’єктів, які містять дані про збір.

Після перевірки інших функцій пов’язаних з зборами, можна дійти висновку, що реалізація функціоналу була успішною. Далі слід перевірити функціонал авторизації та реєстрації користувача. Спробуємо зареєструвати користувача (див. рис. 3.18).

Рисунок 3.18 – Запит на реєстрацію користувача

Як бачимо, реєстрація пройшла успішно і в відповіді, ми отримуємо логін, ідентифікаційний номер, роль, refresh та access токени. Refresh токен відразу зберігається в базу даних, а access токен буде зберігатись в local storage веб-браузеру користувача і під час кожного запиту, він буде використовуватись, щоб отримати змогу робити запити до серверу, але це буде реалізовано вже на frontend-і. Далі перевіримо авторизацію користувача (див. рис. 3.19).

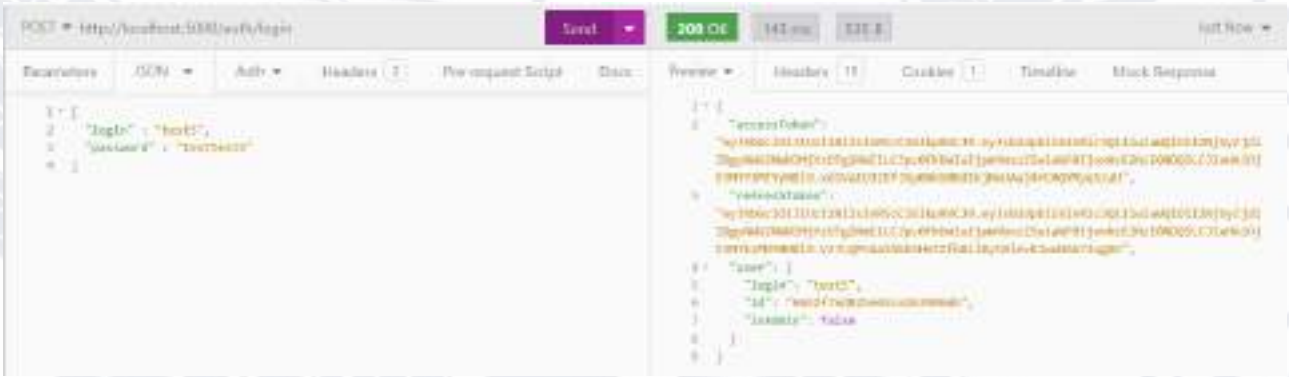


Рисунок 3.19 – Запит на авторизація користувача

Як ми бачимо запит виконався успішно, в відповіді ми отримуємо ті ж дані, що і після реєстрації. Обробка ж цих даних буде відбуватись вже на frontend-і. Також є ще два запити, це на вихід користувача з аккаунту (див. рис. 3.21) та запит на оновлення access токену (див. рис. 3.20).

Рисунок 3.20 – Запит на оновлення access токену

В відповіді на запит оновлення access токену, ми отримуємо ту ж інформацію про користувача, що і в попередніх запитах, але з оновленим access токеном.

Рисунок 3.21 – Запит на вихід з аккаунту

У відповіді на запит виходу з аккаунту, ми отримуємо технічні дані, які не будуть використовуватись на frontend-і, а саме кількість видалених з бази записів. Сам ж запит, виконує видалення refresh токену з бази даних.

Зрештою, слід змоделювати ситуацію в якій виникатиме помилка та перевірити написаний нами обробник помилок (див. рис. 3.22).

Рисунок 3.22 – Запит на авторизацію користувача з неправильним паролем

Як бачимо під час авторизації з вказанням неправильного паролю, в відповідь ми отримуємо помилку з інформацією про це.

Рисунок 3.23 – Запит на реєстрацію вже існуючого користувача

Якщо ж, ми спробуємо зареєструвати вже існуючого користувача, то ми отримаємо помилку з інформацією про це (див. рис. 3.23).

Таким чином, було проведено тестування всього розробленого функціоналу, результат якого – успішний.

3.4 Розробка клієнтської частини веб-платформи

Розробку frontend частини нашого веб-додатку розпочнемо зі створення проекту. Відкриваємо термінал та ініціалізуємо проект командою *npx create-react-app <назва проекту>*. Після чого потрібно відкрити папку *src* в середині папки проекту, адже саме в цій папці буде відбуватись подальша розробка. Видаляємо зайві компоненти, які були створенні за замовчуванням та видаляємо зайвий код з файлу *App.jsx*. Наступним кроком сформулюємо структуру проекту (див. рис. 3.24).

Де, кожна папка містить відповідні файли:

- api – містить файл з конфігурацією з’єднання з сервером
- assets – містить статичні зображення та іконки, які використовуються веб-додатком.
- components – містить окремі компоненти, які можна перевикористати.
- layouts – містить компонент-структуру, який визначає скелет всієї сторінки веб-додатку.

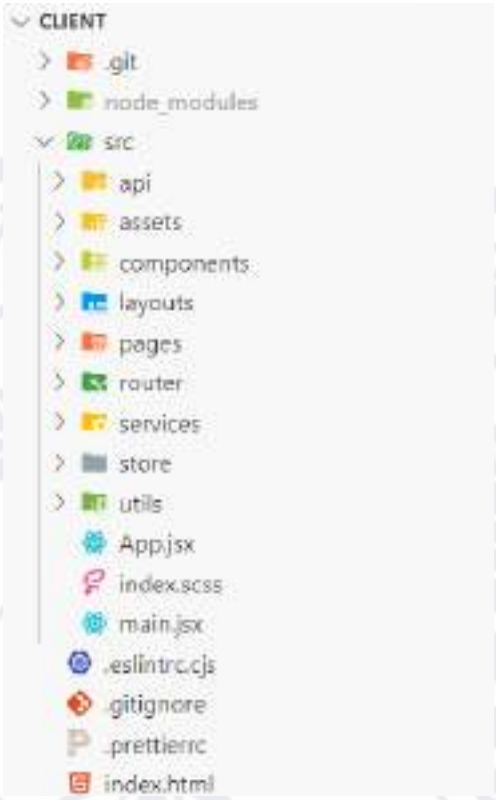


Рисунок 3.24 – Структура frontend проекту

- pages – містить файли розроблених сторінок.
- router – містить конфігурацію шляхів для frontend частини веб-додатку.
- services – містить файли з класами та їх методами, що відповідають за відправку запитів на сервер.
- store – містить конфігурацію глобального сховища додатку.
- utils – містить допоміжні файли, наприклад, variables.scss – змінні для написання стилів.

Наступним кроком потрібно встановити необхідні пакети, таким ж способом, як і під час реалізації backend-у. Необхідні пакети:

- @reduxjs/toolkit – бібліотека для управління глобальним сховищем додатку.
- Axios – бібліотека для здійснення HTTP-запитів.
- Framer-motion – бібліотека, яка розширює можливості по створенню анімацій.
- React-date-picker – компонент для вибору дати.

- React-hook-form – бібліотека, що значно розширює функціонал по керуванню та валідації форм.
- React-select – бібліотека для створення кастомізованих випадаючих списків.
- Sass – це препроцесор, що розширює можливості CSS.

Налаштування файлу package.json не є обов'язковим, оскільки стандартних налаштувань – достатньо для коректної роботи.

Наступним кроком потрібно зверстати всі сторінки та компоненти нашого веб-додатку, а вже потім додавати можливість взаємодіяти з сервером [38,39]. Оскільки верстка всіх компонентів містить велику кількість коду, нижче буде представлено приклад (див. рис. 3.25 та 3.26), аналогічно якому можна створити й інші компоненти, змінюючи тільки необхідну структуру.

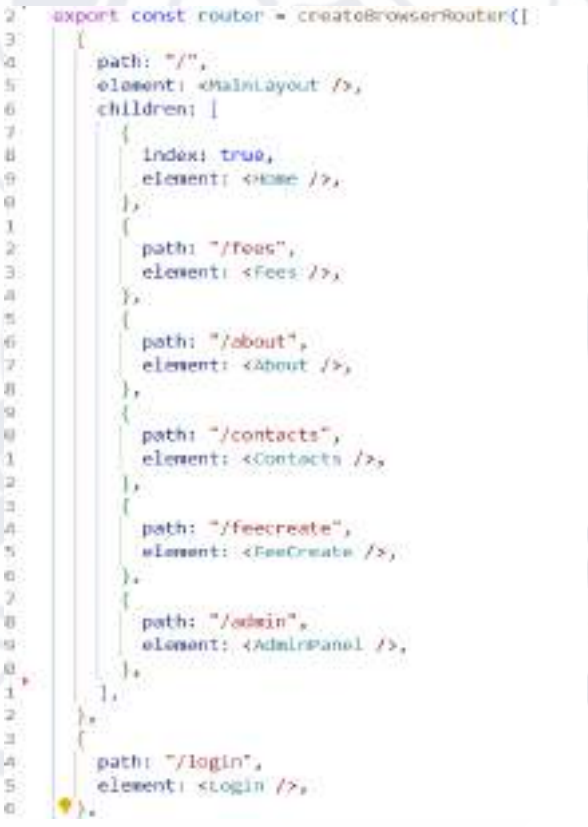
Після підготовки всіх компонентів, налаштуємо маршрутизацію по веб-додатку за допомогою роутера (див. рис. 3.27).

```
src > pages > Home > Home.jsx > ...
import styles from "./Home.module.scss";
import ButtonGradient from "../../components/ButtonGradient/ButtonGradient";
import donation from "../../assets/svg/donation-img.svg";
import heart from "../../assets/svg/heart.svg";
import soldiers from "../../assets/soldiers.png";
import HeroSupports from "../../components/HeroSupports/HeroSupports";

const Home = () => {
  return (
    <main className={styles.main}>
      <section className={styles.intro}>
        <div className={styles.introContainer}>
          <div className={styles.introContent}>
            <h1 className={styles.introTitle}>Разом до перемоги!</h1>
            <p className={styles.introSubtitle}>
              Наші військові - справжні герої, вони віддають своє життя та здоров'я, щоб захистити
              нашу незалежність та забезпечити нам безпеку. Допоможімо нашим воїнам та покажімо, як
              ми їх цінуємо. Разом ми сила!
            </p>
            <ButtonGradient img={heart} />
            <div className={styles.introContentImage}>
              <img src={donation} />
            </div>
          </div>
        </div>
      </section>
      <section className={styles.soldiers}>
        <div className={styles.soldiersContainer}>
          <img classNam={styles.soldiersImg} src={soldiers} />
        </div>
      </section>
      <HeroSupports />
    </main>
  );
};

export default Home;
```

Рисунок 3.25 – Реалізація сторінки Home

```scss
@use "../../utils/variables.scss";

.main {
  section {
    margin-bottom: 130px;
  }
}

.intro {
  max-width: 1950px;
  padding: 0 15px;
  margin: 0 auto;

  background: url("../../assets/intro-img2.png") center right / contain no-repeat;
}

.introContainer {
  max-width: 1462px;
  padding: 0 15px;
  margin: 0 auto;

  .introContent {
    display: flex;
    flex-direction: column;
    justify-content: center;
    row-gap: 40px;
    height: 829px;

    .introTitle {
      font-size: 52px;
      text-transform: uppercase;
      font-weight: 700;
      letter-spacing: 1.56px;
    }

    .introSubtitle {
      font-size: variables.$subtitles-font-size;
```

Рисунок 3.26 – Стилі для сторінки Home



```jsx
export const router = createBrowserRouter([
  {
    path: "/",
    element: <MainLayout />,
    children: [
      {
        index: true,
        element: <Home />,
      },
      {
        path: "/fees",
        element: <Fees />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contacts",
        element: <Contacts />,
      },
      {
        path: "/feecreate",
        element: <FeeCreate />,
      },
      {
        path: "/admin",
        element: <AdminPanel />,
      },
    ],
  },
  {
    path: "/login",
    element: <Login />,
  },
```

Рисунок 3.27 – Приклад часткової реалізації роутера

Наступним етапом є реалізація запитів до серверу. Перш за все в папці api потрібно створити файл api.js, в якому напишемо необхідний код (див. рис. 3.28).

```
1  import axios from "axios";
2
3  export const API_URL = "https://soldiers-hub-project-server.onrender.com/";
4  // export const API_URL = "http://localhost:5000/";
5
6  const api = axios.create({
7    withCredentials: true,
8    baseURL: API_URL,
9  });
10
11 api.interceptors.request.use((config) => {
12   config.headers.Authorization = `Bearer ${localStorage.getItem("token")}`;
13   return config;
14 });
15
16 api.interceptors.response.use(
17   (config) => {
18     return config;
19   },
20   async (error) => {
21     const originalRequest = error.config;
22     if (error.response.status === 401 && error.config && !error.config._isRetry) {
23       originalRequest._isRetry = true;
24       try {
25         const response = await axios.get(`${API_URL}/refresh`, {
26           withCredentials: true,
27         });
28         localStorage.setItem("token", response.data.accessToken);
29         return api.request(originalRequest);
30       } catch (e) {
31         console.log("Not authorized");
32       }
33     }
34     throw error;
35   }
36 );
```

Рисунок 3.28 – Зміст файлу api.js

В коді api.js файлу, представленому на рисунку 3.28, змінна API_URL – це URL шлях до нашого серверу. В рядках 6-9 ми створюємо екземпляр Axios з вказаними параметрами, які будуть використовуватись для здійснення HTTPS-запитів. В рядках 11-14 ми створюємо інтерсептор для запитів, який буде до кожного створеного нами запиту, додавати в cookie існуючий access токен користувача, за умови, якщо користувач ввійшов в аккаунт. В рядках 16-36 ми виконуємо оновлення access токену за допомогою refresh токену, якщо його термін дії вже сплинув.

Далі перейдемо в папку services та створимо класи FeeService, AuthService та їх методи для відправки запитів на сервер з необхідними даними (див. рис. 3.29).

```
import api from "../api/api";

export class FeeService {
  static async getApplications() {
    const data = await api.get("fee/applications");
    return data;
  }

  static async getFees() {
    const data = await api.get("fee/getFees");
    return data;
  }

  static async acceptApplication(id) {
    const data = await api.put(`fee/acceptApplication/${id}`);
    return data;
  }

  static async refuseApplication(id) {
    const data = await api.delete(`fee/refuseApplication/${id}`);
    return data;
  }

  static async createApplication(info) {
    const data = await api.post("fee/createApplication", info, {
      headers: { "Content-Type": "multipart/form-data" },
    });
    return data;
  }
}
```

Рисунок 3.29 – Клас FeeService та його методи

Аналогічно цьому прикладу створюємо клас AuthService. Після чого можемо перейти до налаштування глобального сховища веб-додатку (див. рис. 3.30).

```
const initalFeesState = {
  fees: [],
  volunteersFees: [],
  rebuildingFees: [],
  militaryFees: [],
  applications: [],
  creationSuccess: false,
  error: "",
  loading: false,
};

export const getApplications = createAsyncThunk("fees/getApplication", async () => {
  const response = await FeeService.getApplications();
  return response.data;
});

export const getFees = createAsyncThunk("fees/getFees", async () => {
  const response = await FeeService.getFees();
  return response.data;
});

export const acceptApplication = createAsyncThunk("fees/acceptApplication", async (id) => {
  const response = await FeeService.acceptApplication(id);
  return response.data;
});

export const refuseApplication = createAsyncThunk("fees/refuseApplication", async (id) => {
  const response = await FeeService.refuseApplication(id);
  return response.data;
});
```

Рисунок 3.30 – Стан глобального сховища Fee та його actions (екшени)

Як показано на рисунку 3.30, глобальне сховище було розбито на свої логічні частини, тобто сховище для зборів і сховище пов’язане з авторизацією та інформацією про користувачів знаходяться в окремих файлах. Даний файл має стан за замовчуванням, який знаходиться в змінній initalFeesState, екшени та

редюсери. Якщо з станом за замовчуванням все зрозуміло, то що таке екшени та редюсери потрібно пояснити. Екшени це дії, завдяки яким, ми можемо змінювати стан нашого сховища, а редюсери, це обробники, які визначають як саме буде змінено стан сховища відповідно до якогось екшену [39,40].

В нашому випадку, ми створюємо асинхронний екшен для запиту до серверу. Він створюється завдяки функції createAsyncThunk, яка приймає 2 параметри: 1 – це назва екшена, прийнята норма іменування <Назва слайсу або стору/Назва екшену>, 2 – це колбек функція, яка містить в собі логіку самого екшену.

```
const FeesSlice = createSlice({
  name: "fees",
  initialState: initalFeesState,
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(getApplications.pending, (state) => {
      state.applications = [];
      state.loading = true;
      state.error = "";
    });
    builder.addCase(getApplications.fulfilled, (state, action) => {
      state.loading = false;
      state.applications = action.payload;
    });
    builder.addCase(getApplications.rejected, (state, action) => {
      state.loading = false;
      if (action.error.message) {
        state.error = action.error.message;
      }
    });
```

Рисунок 3.31 – Приклад reducer (редюсеру)

Приклад створення слайсу або іншими словами стору, як і створення редюсеру можна побачити на рисунку 3.31. Створення слайсу, приймає декілька параметрів, а саме: назву, стан за замовчуванням та редюсери. Редюсери ж мають 3 стана: успішне виконання екшену, в процесі виконання та помилка в виконанні. І на основі цих 3 станів, буде відповідно змінюватися значення глобального сховища.

Останнім фінальним етапом в розробці frontend частини, буде додавання можливості взаємодії клієнта з сервером. Розглянемо приклад додавання логіки взаємодії в компонент FeeCreate (див. рис. 3.32).

```
const handleFormSubmit = async (data) => {
  if (isLogged) {
    const application = new FormData();
    application.append("image", data.image[0]);
    application.append("isAccepted", false);
    application.append("feeByDonnu", regex.test(data.email));
    application.append("date", Date.parse(data.finish));
    Object.keys(data).forEach((el) => {
      application.append(el, data[el]);
    });

    dispatch(createApplication(application));
  } else {
    navigate("/login");
  }
};
```

Рисунок 3.32 – Логіка надсилання запиту на створення збору

Як можна побачити на прикладі асинхронної функції handleFormSubmit, яка викликається вже після успішної валідації форми, взаємодію з глобальним сховищем. Зміна isLogged, що повідомляє нам чи є користувач авторизованим чи ні, береться з глобального сховища. Після чого створюється об’єкт необхідних даних та за допомогою функції dispatch, яка доставляє екшени в стор, де він і буде виконаний і оброблений редюсером, відсилається запит на створення збору. Аналогічно цьому прикладу, потрібно додати можливість взаємодії клієнта з сервером де це необхідно, після чого можна приступити до тестування нашого веб-додатку.

3.5 Тестування розробленого програмного забезпечення

Фінальним етапом в розробці нашого веб-додатку, буде його тестування. Перш за все давайте переглянемо результат верстки одної з сторінок (див. рис. 3.33).

Рисунок 3.33 – Вигляд сторінки Home

Тепер спробуємо зареєструвати користувача. Відкриваємо сторінку реєстрації та вводимо дані (див. рис. 3.34).

Рисунок 3.34 – Сторінка реєстрації користувача

Пробуємо зареєструватись. Реєстрація пройшла успішно і нас відразу авторизувало. Для перевірки успішності запиту, можна переглянути інформацію вкладки Network devTools (див. рис. 3.35).

Рисунок 3.35 – Запит на реєстрацію користувача та відповідь

Далі, спробуємо вийти з аккаунта та авторизуватись під логіном admin та паролем admin, цей користувач має роль адміністратора (див. рис. 3.36).

Рисунок 3.36 – Успішна авторизація в аккаунт адміністратора

Тепер спробуємо створити збір, підтвердити його від іменні адміністратора та потім переглянути. Заповнюємо форму для створення збору (див. рис. 3.37):

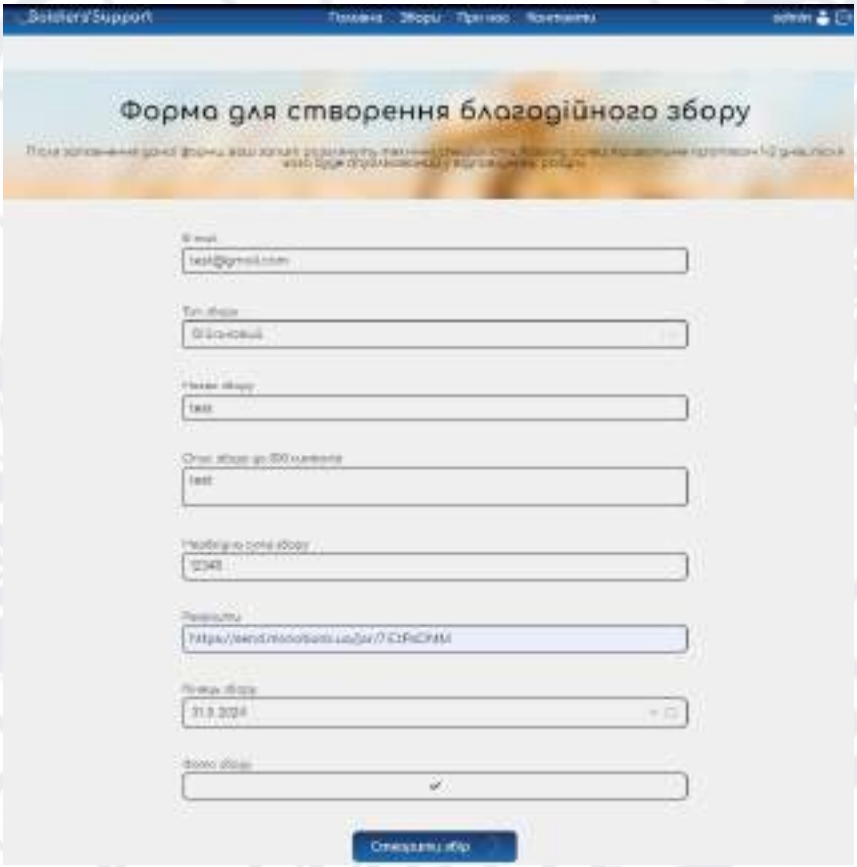


Рисунок 3.37 – Заповнення форми для створення збору

Створюємо збір (див. рис. 3.38):

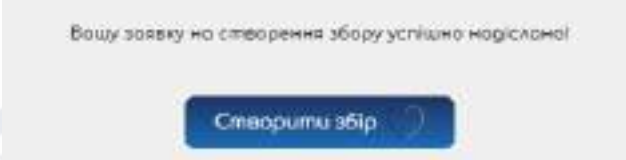


Рисунок 3.38 – Повідомлення про успішне створення збору

Переходимо в адмін панель та дивимось чи є цей збір в списку заявок (див. рис. 3.39).

Рисунок 3.39 – Запит на підтвердження збору в адмін панелі

Схвалюємо його та пробуємо переглянути від імені користувача (див. рис. 3.40).

Рисунок 3.40 – Перегляд збору на сторінці зборів

Як бачимо збір підтверджений та користувач може його переглядати.

Тестування завершено з успішною перевіркою основного функціоналу згідно початкових вимог. Таким чином, розробка нашого веб-додатку успішно завершена.

ВИСНОВКИ

В результаті даної бакалаврської роботи було досягнуто поставлену мету, а саме покрашено сервіс розробки волонтерських програм за рахунок розробки волонтерської платформи, що об'єднує волонтерів та користувачів для підтримки благодійних ініціатив та збору коштів із застосуванням Node.js та React.

В результаті роботи було виконано всі поставлені завдання дослідження, а саме здійснено аналіз предметної галузі та існуючих аналогів; підібрано оптимальні технології для реалізації волонтерської платформи, розроблено архітектуру та функціонуючу веб-платформу та протестовано на відповідність початковим вимогам.

Розроблена волонтерська веб-платформа на даний час впроваджується в основний портал Донецького національного університету імені Василя Стуса та планується бути використана для благодійної діяльності.

Розробка волонтерської платформи із застосуванням Node.js та React є важливим та актуальним завданням, спрямованим на об'єднання волонтерів та користувачів для реалізації благодійних ініціатив та збору коштів. Проектування та реалізація такої платформи має на меті забезпечити надійність та гнучкість для змін, враховуючи потреби користувачів та специфіку благодійних проектів.

Проектування волонтерської платформи вимагало ретельного аналізу потреб користувачів та визначення їх очікувань від платформи. На цьому етапі було прораховано різноманітність потенційних користувачів та спроектовано зручний та інтуїтивно зрозумілий інтерфейс.

Застосування технологій Node.js та React стало ключовим елементом успішної реалізації проекту. Node.js забезпечив потужну та швидку серверну частину, що дозволило ефективно обробляти запити користувачів та забезпечувати надійну роботу платформи. React, у свою чергу, дозволив створювати динамічні та інтерактивні інтерфейси, що забезпечували зручне та приємне користування платформою.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Актуальність волонтерської діяльності у період війни. URL: https://enpuir.npu.edu.ua/bitstream/handle/123456789/38260/Mikheieva.pdf?sequence=1&isAllowed=y (дата звернення: 15.04.2024)

2. Алексюк В.В., Комаров В.М. MONGODB ЯК ЕФЕКТИВНИЙ ЗАСІБ В ЗАДАЧАХ ОБРОБКИ ТА АНАЛІЗУ ДАНИХ // Матеріали IV Всеукраїнської науково-практичної конференції *Комп'ютерні технології обробки даних*: збірник наукових праць – Вінниця, Донецький національний університет імені Василя Стуса, 2023. – 348 с. – С.185-187.

3. Етапи розробки веб-додатків. 2023. URL: https://terentevdesignstudio.com/blog/etapi-rozrobki-veb-dodatkiv/ (дата звернення: 15.04.2024)

4. Frontend-developer: посібник для початківців. 2023. URL: https://genius.space/lab/frontend-developer-posibnik-dlya-pochatkivtsiv/ (дата звернення: 16.04.2024)

5. React. URL: https://uk.legacy.reactjs.org/ (дата звернення: 16.04.2024)

6. Що таке Angular? URL: https://web-developer.in.ua/assets/articles/angular/angular-introduction/angular-introduction.html (дата звернення: 16.04.2024)

7. Що таке Vue.js? 2023. URL: https://lemon.school/blog/shho-take-vue-js (дата звернення: 16.04.2024)

8. Що таке jQuery? URL: https://tonyline.com.ua/glossary/jquery/ (дата звернення: 17.04.2024)

9. Bootstrap. URL: https://druzy.com.ua/bootstrap-sho-ce-z-chogo-pochati-vivchennia-i-iak-vikoristovyvati/ (дата звернення: 17.04.2024)

10. Використання Tailwind CSS для швидкої розробки адаптивного дизайну. 2024. URL: https://it-rating.ua/vikoristannya-tailwind-css-dlya-shvidkogo-rozrobki-adaptivnogo-dizaynu (дата звернення: 18.04.2024)

11. Знати CSS — недостатньо: що таке SCSS та як з ним працювати. Розбір синтаксису. 2023. URL: https://highload.today/uk/scss/ (дата звернення: 20.04.2024)

12. React JS, що і для чого він потрібен? 2023. URL: https://dan-it.com.ua/uk/blog/chto-takoe-react-js-i-dlja-chego-on-nuzhen/ (дата звернення: 19.04.2024)

13. Angular vs Vue.js: що обрати. 2021. URL: https://dou.ua/lenta/columns/angular-vs-vuejs/ (дата звернення: 19.04.2024)

14. Перші кроки в Angular. 2023. URL: https://foxminded.ua/angular/ (дата звернення: 19.04.2024)

15. Що таке jQuery: де використовується і які переваги дає розробникам. 2017. URL: https://phoneinfo8.info/sho-take-jquery/ (дата звернення: 19.04.2024)

16. Bootstrap: призначення і використання. 2022. URL: https://ua-meta.com/didzhytal/modalne-vikno-bootstrap-priznachennya-i-vikoristannya.html (дата звернення: 20.04.2024)

17. Tailwind CSS: Pros and Cons. 2023. URL: https://medium.com/readytowork-org/tailwind-css-pros-and-cons-f1a8fdb1fb47 (дата звернення: 20.04.2024)

18. Pros & Cons of Sass/SCSS. 2019. URL: https://medium.com/@virginialash1/pros-cons-of-sass-scss-4e4152b658f8 (дата звернення: 20.04.2024)

19. What Is Back End? 2021. URL: https://www.codecademy.com/resources/blog/what-is-back-end/ (дата звернення: 21.04.2024)

20. Django Introduction. URL: https://www.w3schools.com/django/django_intro.php (дата звернення: 21.04.2024)

21. Laravel (PHP Framework for Web Artisans). URL: https://laravel.com/ (дата звернення: 21.04.2024)

22. Spring Boot: The Ultimate Guide. 2023. URL: https://masteringbackend.com/posts/spring-boot#complete-spring-boot-overview (дата звернення: 21.04.2024)

23. What is Express.js? URL: https://www.codecademy.com/article/what-is-express-js (дата звернення: 21.04.2024)

24. Getting Started with Rails. URL: https://guides.rubyonrails.org/getting_started.html#what-is-rails-questionmark (дата звернення: 22.04.2024)

25. РОЗРОБКА ВЕБ ДОДАТКІВ З ВИКОРИСТАННЯМ PYTHON І DJANGO. 2021. URL: https://webcase.com.ua/uk/blog/razrabotka-veb-prilozhenij-s-ispolzovaniem-python-i-django/ (дата звернення: 22.04.2024)

26. 8 ключових переваг фрейморку Laravel для веб-розробки. 2024. URL: https://wezom.com.ua/ua/blog/17-preimuschestv-ispolzovanija-laravel-v-it-industrii (дата звернення: 22.04.2024)

27. Spring Boot Advantages and Disadvantages: A Comprehensive Analysis. 2023. URL: https://medium.com/@sureshkrishna75525/spring-boot-advantages-and-disadvantages-a-comprehensive-analysis-9dde37c19926 (дата звернення: 23.04.2024)

28. ExpressJS. Advantages and Disadvantages. URL: https://data-flair.training/blogs/expressjs-advantages-and-disadvantages/ (дата звернення: 23.04.2024)

29. Ruby on Rails: Pros and Cons. URL: https://fullscale.io/blog/ruby-on-rails-pros-and-cons/ (дата звернення: 23.04.2024)

30. Типи баз даних: особливості, відмінності та приклади. 2023. URL: https://dou.ua/lenta/articles/types-of-databases/ (дата звернення: 24.04.2024)

31. The Best IDE For Web Development. 2023. URL: https://algorithmman.com/best-ide-web-development/ (дата звернення: 24.04.2024)

32. IntelliJ IDEA vs PhpStorm vs Visual Studio. 2024. URL: https://stackshare.io/stackups/intellij-idea-vs-phpstorm-vs-visual-studio (дата звернення: 24.04.2024)

33. Top 5 API Testing Tools & Methodologies. 2023. URL: https://speedscale.com/blog/api-testing-tools/ (дата звернення: 25.04.2024)

34. Doug Lowe. Networking All-in-One For Dummies 6th Edition. For Dummies, 2016. 912 с.

35. Розробка веб-сайтів з використанням архітектурного патерну MVC (Model-View-Controller). 2024. URL: https://it-rating.ua/rozrobka-veb-saytiv-z-vikoristannyam-arhitekturnogo-paternu-mvc-model-view-controller (дата звернення: 29.04.2024)

36. Chris Minnick. JavaScript All-in-One For Dummies. For Dummies, 2023. 816 с.

37. Springer S. Node.js: The Comprehensive Guide to Server-Side JavaScript Programming. Rheinwerk Computing, 2022. 834 с.

38. Wolf J. HTML and CSS: The Comprehensive Guide. Rheinwerk Computing, 2023. 814 с.

39. Wieruch R. The Road to React. Independently published, 2018. 286 с.

40. Для чого і коли використовується Redux. 2023. URL: https://foxminded.ua/shcho-take-redux/ (дата звернення: 3.05.2024).

ДОДАТКИ

ДОДАТОК А

Лістинг основних функцій серверної частини

Код файлу feeModel.js

```
const { Schema, model } = require("mongoose");

const Fee = new Schema({
  email: { type: String, required: true },
  type: { type: String, required: true },
  name: { type: String, required: true },
  description: { type: String, required: true },
  sum: { type: Number, required: true },
  requisite: { type: String, required: true },
  finish: { type: String, required: true },
  isAccepted: { type: Boolean, required: true },
  imageSrc: { type: Buffer, required: true },
});
module.exports = model("Fee", Fee);
```

Код файлу FeeService.js

```
const Fee = require("../Models/feeModel");
class FeeService {
  async createApplication(fee) {
    const createdFeeApplication = await Fee.create(fee);
    return createdFeeApplication;
  }
  async getApplicationsFee() {
    const feeApplications = await Fee.find({ isAccepted: false });
    return feeApplications;
  }
  async acceptFeeApplication(id) {
    const acceptedFee = await Fee.findByIdAndUpdate(id, {
      isAccepted: true,
    });
    return acceptedFee;
  }
  async refuseFeeApplication(id) {
    const refusedFee = await Fee.findByIdAndDelete(id);
    return refusedFee;
  }
  async getFees() {
    const fees = await Fee.find({ isAccepted: true });
    return fees;
  }
  async getVolunteersFees() {
    const fees = await Fee.find({ isAccepted: true, type: "volunteer" });
    return fees;
  }
  async getMilitaryFees() {
```

```
    const fees = await Fee.find({ isAccepted: true, type: "military" });
    return fees;
  }
  async getRebuildingFees() {
    const fees = await Fee.find({ isAccepted: true, type: "rebuild" });
    return fees;
  }
  async getDonnuFees() {
    const fees = await Fee.find({ isAccepted: true, feeByDonnu: true });
    return fees;
  }
}

module.exports = new FeeService();
```

Код файлу FeeController.js

```
const FeeService = require("../Services/feeService");
const fs = require("fs");

class FeeController {
  async createApplication(req, res) {
    try {
      const imageSrc = fs.readFileSync("uploads/" + req.file.filename);
      const fee = await FeeService.createApplication({ ...req.body, imageSrc });
      return res.json(fee);
    } catch (e) {
      res.status(500).json(e);
    }
  }

  async getApplications(req, res) {
    try {
      const applications = await FeeService.getApplicationsFee();
      return res.json(applications);
    } catch (e) {
      res.status(500).json(e);
    }
  }

  async acceptFeeApplication(request, response) {
    try {
      const acceptedApplication = await FeeService.acceptFeeApplication(
        request.params.id
      );
      return response.json(acceptedApplication);
    } catch (error) {
      Response.status(500).json(error);
```

```
    }
  }

!!!!  async refuseFeeApplication(request, response) {
    try {
      const refusedApplication = await FeeService.refuseFeeApplication(
        req.params.id
      );
      return res.json(refusedApplication);
    } catch (e) {
      res.status(500).json(e);
    }
  }

  async getFees(request, response) {
    try {
      const fees = await FeeService.getFees();
      return response.json(fees);
    } catch (error) {
      response.status(500).json(error);
    }
  }

  async getVolunteersFees(request, response) {
    try {
      const fees = await FeeService.getVolunteersFees();
      return response.json(fees);
    } catch (error) {
      response.status(500).json(error);
    }
  }

  async getMilitaryFees(request, response) {
    try {
      const fees = await FeeService.getMilitaryFees();
      return response.json(fees);
    } catch (error) {
      response.status(500).json(error);
    }
  }

  async getRebuildingFees(request, response) {
    try {
      const fees = await FeeService.getRebuildingFees();
      return response.json(fees);
    } catch (error) {
      response.status(500).json(error);
    }
  }

  async getDonnuFees(req, res) {
```

```
    try {
      const fees = await FeeService.getDonnuFees();
      return res.json(fees);
    } catch (e) {
      res.status(500).json(e);
    }
  }
}

module.exports = new FeeController();
```

Код файлу FeesRouter.js

```
const Router = require("express").Router;
const FeeController = require("../Controllers/feeController");
const { upload } = require("../Services/uploadService");
const feesRouter = new Router();

feesRouter.post(
  "/createApplication",
  upload.single("image"),
  FeeController.createApplication
);
feesRouter.get("/applications", FeeController.getApplications);
feesRouter.put("/acceptApplication/:id", FeeController.acceptFeeApplication);
feesRouter.delete("/refuseApplication/:id", FeeController.refuseFeeApplication);
feesRouter.get("/getVolunteersFees", FeeController.getVolunteersFees);
feesRouter.get("/getMilitaryFees", FeeController.getMilitaryFees);
feesRouter.get("/getRebuildingFees", FeeController.getRebuildingFees);
feesRouter.get("/getDonnuFees", FeeController.getDonnuFees);
feesRouter.get("/getFees", FeeController.getFees);

module.exports = feesRouter;
```

Код файлу index.js

```
require("dotenv").config();
const express = require("express");
const cors = require("cors");
const cookieParser = require("cookie-parser");
const mongoose = require("mongoose");
const authRouter = require("./Routers/authRouter");
const feeRouter = require("./Routers/feesRouter");
const defendersRouter = require("./Routers/defenderRouter");
const errorMiddleware = require("./Middlewares/errorMiddleware");
const PORT = process.env.PORT || 5000;
const app = express();

app.use(express.json());
app.use(cookieParser());
```

```
app.use(
  cors({
    credentials: true,
    origin: [
      "http://localhost:5173",
      "https://soldiers-hub-project-1itgohixm-botsyundenys.vercel.app",
      "https://soldiers-hub-project.vercel.app",
    ],
  })
);
app.use("/auth", authRouter);
app.use("/fee", feeRouter);
app.use("/defender", defendersRouter);
app.use(errorMiddleware);

const start = async () => {
  try {
    await mongoose.connect(process.env.DB_URL, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    app.listen(PORT, () => console.log(`Server was started on PORT = ${PORT}`));
  } catch (error) {
    console.log(error);
  }
};

start();
```

ДОДАТОК Б

Лістинг основних функцій клієнтської частини

Код файлу api.js

```
import axios from "axios";

export const API_URL = "https://soldiers-hub-project-server.onrender.com/";
// export const API_URL = "http://localhost:5000/";

const api = axios.create({
  withCredentials: true,
  baseURL: API_URL,
});

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

api.interceptors.response.use(
  (config) => {
    return config;
  },
  async (error) => {
    const originalRequest = error.config;
    if (error.response.status === 401 && error.config && !error.config._isRetry) {
      originalRequest._isRetry = true;
      try {
        const response = await axios.get(`${API_URL}/refresh`, {
          withCredentials: true,
        });
        localStorage.setItem("token", response.data.accessToken);
        return api.request(originalRequest);
      } catch (e) {
        console.log("Not authorized");
      }
    }
    throw error;
  },
);

export default api;
```

Код файлу AuthService.js

```
import api from "../api/api";

export class AuthService {
  static async login({ login, password }) {
    const data = await api.post("auth/login", { login, password });
    return data;
```

```
  }

  static async logout() {
    const data = await api.post("auth/logout");
    return data;
  }

  static async registration({ login, password, isAdmin }) {
    const data = await api.post("auth/registration", { login, password, isAdmin });
    return data;
  }

  static async checkAuth() {
    const data = await api.get("auth/refresh");
    return data;
  }
}
```

Код файлу FeeService.js

```
import api from "../api/api";

export class FeeService {
  static async getApplications() {
    const data = await api.get("fee/applications");
    return data;
  }

  static async getFees() {
    const data = await api.get("fee/getFees");
    return data;
  }

  static async acceptApplication(id) {
    const data = await api.put(`fee/acceptApplication/${id}`);
    return data;
  }

  static async refuseApplication(id) {
    const data = await api.delete(`fee/refuseApplication/${id}`);
    return data;
  }

  static async createApplication(info) {
    const data = await api.post("fee/createApplication", info, {
      headers: { "Content-Type": "multipart/form-data" },
    });
    return data;
  }
}
```

Код файлу router.jsx

```
import { createBrowserRouter } from "react-router-dom";
import MainLayout from "../layouts/MainLayout";
import Home from "../pages/Home/Home";
import Fees from "../pages/Fees/Fees";
import About from "../pages/About/About";
import Contacts from "../pages/Contacts/Contacts";
import Login from "../pages/Login/Login";
import Registration from "../pages/Registration/Registration";
import FeeCreate from "../pages/FeeCreate/FeeCreate";
import AdminPanel from "../pages/AdminPanel/AdminPanel";
import DonnuDefenders from "../pages/DonnuDefenders/DonnuDefenders";
import DonnuFallenDefenders from
"../pages/DonnuFallenDefenders/DonnuFallenDefenders";

export const router = createBrowserRouter([
  {
    path: "/",
    element: <MainLayout />,
    children: [
      {
        index: true,
        element: <Home />,
      },
      {
        path: "/fees",
        element: <Fees />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contacts",
        element: <Contacts />,
      },
      {
        path: "/feecreate",
        element: <FeeCreate />,
      },
      {
        path: "/admin",
        element: <AdminPanel />,
      },
      {
        path: "/defenders",
        element: <DonnuDefenders />,
      },
      {
```

```
        path: "/fallendefenders",
        element: <DonnuFallenDefenders />,
      },
    ],
  },
  {
    path: "/login",
    element: <Login />,
  },
  {
    path: "/registration",
    element: <Registration />,
  },
]);
```

Код файлу Home.jsx

```
import styles from "./Home.module.scss";
import ButtonGradient from "../../components/ButtonGradient/ButtonGradient";

import donation from "../../assets/svg/donation-img.svg";
import heart from "../../assets/svg/heart.svg";
import soldiers from "../../assets/soldiers.png";
import HeroSupports from "../../components/HeroSupports/HeroSupports";

const Home = () => {
  return (
    <main className={styles.main}>
      <section className={styles.intro}>
        <div className={styles.introContainer}>
          <div className={styles.introContent}>
            <h1 className={styles.introTitle}>Разом до перемоги!</h1>
            <p className={styles.introSubtitle}>
              Наші військові - справжні герої, вони віддають своє життя та
здоров'я, щоб захистити
              нашу незалежність та забезпечити нам безпеку. Домопожімо нашим воїнам
та покажімо, як
              ми їх цінуємо. Разом ми сила!
            </p>
            <ButtonGradient img={heart} />
            <div className={styles.introContentImage}>
              <img src={donation} />
            </div>
          </div>
        </div>
      </section>
      <section className={styles.soldiers}>
        <div className={styles.soldiersContainer}>
          <img className={styles.soldiersImg} src={soldiers} />
        </div>
      </section>
```

```
      <HeroSupports />
    </main>
  );
};

export default Home;
```

Код файлу FeeSlice.js

```
import { createAsyncThunk, createSlice } from "@reduxjs/toolkit";
import { FeeService } from "../../services/FeeService";

const initalFeesState = {
  fees: [],
  volunteersFees: [],
  rebuildingFees: [],
  militaryFees: [],
  applications: [],
  creationSuccess: false,
  error: "",
  loading: false,
};

export const getApplications = createAsyncThunk("fees/getApplications", async () =>
{
  const response = await FeeService.getApplications();
  return response.data;
});

export const getFees = createAsyncThunk("fees/getFees", async () => {
  const response = await FeeService.getFees();
  return response.data;
});

export const acceptApplication = createAsyncThunk("fees/acceptApplication", async
(id) => {
  const response = await FeeService.acceptApplication(id);
  return response.data;
});

export const refuseApplication = createAsyncThunk("fees/refuseApplication", async
(id) => {
  const response = await FeeService.refuseApplication(id);
  return response.data;
});

export const createApplication = createAsyncThunk("fees/createApplication", async
(application) => {
  const response = await FeeService.createApplication(application);
  return response.data;
});
```

```
export const getVolunteersFees = createAsyncThunk("fees/getVolunteersFees", async
() => {
  const response = await FeeService.getVolunteersFees();
  return response.data;
});

export const getMilitaryFees = createAsyncThunk("fees/getMilitaryFees", async () =>
{
  const response = await FeeService.getMilitaryFees();
  return response.data;
});

export const getRebuildingFees = createAsyncThunk("fees/getRebuildingFees", async
() => {
  const response = await FeeService.getRebuildingFees();
  return response.data;
});

export const getDonnuFees = createAsyncThunk("fees/getDonnuFees", async () => {
  const response = await FeeService.getDonnuFees();
  return response.data;
});

const FeesSlice = createSlice({
  name: "fees",
  initialState: initalFeesState,
  reducers: {},
  extraReducers: (builder) => {
    builder.addCase(getApplications.pending, (state) => {
      state.applications = [];
      state.loading = true;
      state.error = "";
    });
    builder.addCase(getApplications.fulfilled, (state, action) => {
      state.loading = false;
      state.applications = action.payload;
    });
    builder.addCase(getApplications.rejected, (state, action) => {
      state.loading = false;
      if (action.error.message) {
        state.error = action.error.message;
      }
    });
    builder.addCase(getFees.pending, (state) => {
      state.fees = [];
      state.loading = true;
      state.error = "";
    });
    builder.addCase(getFees.fulfilled, (state, action) => {
      state.loading = false;
```

```
  state.fees = action.payload;
});
builder.addCase(getFees.rejected, (state, action) => {
  state.loading = false;
  if (action.error.message) {
    state.error = action.error.message;
  }
});
builder.addCase(acceptApplication.pending, (state) => {
  state.loading = true;
  state.error = "";
});
builder.addCase(acceptApplication.fulfilled, (state, action) => {
  state.applications = state.applications.filter(
    (application) => application._id !== action.payload._id,
  );
  state.loading = false;
});
builder.addCase(acceptApplication.rejected, (state, action) => {
  state.loading = false;
  if (action.error.message) {
    state.error = action.error.message;
  }
});
builder.addCase(refuseApplication.pending, (state) => {
  state.loading = true;
  state.error = "";
});
builder.addCase(refuseApplication.fulfilled, (state, action) => {
  state.applications = state.applications.filter(
    (application) => application._id !== action.payload._id,
  );
  state.loading = false;
});
builder.addCase(refuseApplication.rejected, (state, action) => {
  state.loading = false;
  if (action.error.message) {
    state.error = action.error.message;
  }
});
builder.addCase(createApplication.pending, (state) => {
  state.loading = true;
  state.creationSuccess = false;
  state.error = "";
});
builder.addCase(createApplication.fulfilled, (state) => {
  state.creationSuccess = true;
  state.loading = false;
});
builder.addCase(createApplication.rejected, (state, action) => {
  state.loading = false;
```

```
  if (action.error.message) {
    state.error = action.error.message;
  }
});
builder.addCase(getVolunteersFees.pending, (state) => {
  state.loading = true;
  state.fees = [];
  state.error = "";
});
builder.addCase(getVolunteersFees.fulfilled, (state, action) => {
  state.loading = false;
  state.fees = action.payload;
});
builder.addCase(getVolunteersFees.rejected, (state, action) => {
  state.loading = false;
  if (action.error.message) {
    state.error = action.error.message;
  }
});
builder.addCase(getRebuildingFees.pending, (state) => {
  state.loading = true;
  state.fees = [];
  state.error = "";
});
builder.addCase(getRebuildingFees.fulfilled, (state, action) => {
  state.loading = false;
  state.fees = action.payload;
});
builder.addCase(getRebuildingFees.rejected, (state, action) => {
  state.loading = false;
  if (action.error.message) {
    state.error = action.error.message;
  }
});
builder.addCase(getMilitaryFees.pending, (state) => {
  state.loading = true;
  state.fees = [];
  state.error = "";
});
builder.addCase(getMilitaryFees.fulfilled, (state, action) => {
  state.loading = false;
  state.fees = action.payload;
});
builder.addCase(getMilitaryFees.rejected, (state, action) => {
  state.loading = false;
  if (action.error.message) {
    state.error = action.error.message;
  }
});
builder.addCase(getDonnuFees.pending, (state) => {
  state.loading = true;
```

```
      state.fees = [];
      state.error = "";
    });
    builder.addCase(getDonnuFees.fulfilled, (state, action) => {
      state.loading = false;
      state.fees = action.payload;
    });
    builder.addCase(getDonnuFees.rejected, (state, action) => {
      state.loading = false;
      if (action.error.message) {
        state.error = action.error.message;
      }
    });
  },
});

export default FeesSlice.reducer;
```

Код файлу index.js папки store

```
import { configureStore } from "@reduxjs/toolkit";
import feesReducer from "./slices/FeeSlice";
import authReducer from "./slices/AuthSlice";
import defendersReducer from "./slices/DefenderSlice";

const rootReducer = { fees: feesReducer, auth: authReducer, defenders:
defendersReducer };

export const store = configureStore({ reducer: rootReducer });
```

ДОДАТОК В

Декларація щодо унікальності текстів роботи

та невикористання матеріалів інших авторів без посилань

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)