МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Веб додаток для інвентаризації речей загального призначення»

Завідувач випускаючої кафедри — **Довбиш А.С.**

Керівник роботи — **Коробов А.Г.**

Студента групи ІНЗ – 73 – 9с — **Шапаренко І.В.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної і вечірньої форм навчання

Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2021 р.

ЗАВДАННЯ

до випускної роботи

Студента четвертого курсу, групи ІНз-73-9С спеціальності "Комп'ютерні науки" заочної форми навчання Шапаренка Івана Валерійовича.

Тема: "**Веб додаток для інвентаризації речей загального призначення**"

Затверджена наказом по СумДУ

№_____ от _____2021 р.

Зміст пояснювальної записки: 1) інформаційний огляд; 2) постановка задачі – вибір технологій; 3) програмна реалізація;

Дата видачі завдання "_____"_____2021 р.

Керівник випускної роботи _____ Коробов А.Г.

Завдання прийняв до виконання _____ Шапаренко І.В.

РЕФЕРАТ

Записка: 74 стор., 55 рис., 44 джерел.

Об’єкт дослідження — інвентаризації речей загального призначення

Мета роботи — розробка системи для інвентаризації речей загального призначення.

Результати — розроблено web-додаток для інвентаризації речей загального призначення. Готова система добре адаптується під різні сфери застосування. Існуюча кодова база дозволяє з легкістю вносити зміни для нарощування функціональності.

WEB, GIT, JAVASCRIPT, HTML, CSS, SPA, VUE, VUETIFY,
NODE, EXPRESS, SEQUELIZE, POSTGRESQL, DOCKER,
NGINX, GOOGLE CLOUD PLATFORM

ЗМІСТ

ВСТУП

Дотепер зачатки інвентаризаційних процесів присутні в багатьох сферах діяльності. Людину постійно оточує велика кількість речей різного призначення, розміру, структури, форми, тощо. Фізично вона не в змозі тримати в своїй оперативній пам’яті такий об’єм даних, особливо про речі, які використовує рідко. Дана проблема є актуальною для різних сфер бізнесу: магазини, склади, аптеки, тощо. Для державних організацій: заклади освіти, медичні заклади, тощо. А також безпосередньо для місця проживання людини.

Оскільки дана проблема не нова, вже існують різні рішення направленні на певну сферу діяльності, але вони адаптовані лише під конкретні умови, та насилу підходять для вирішення інших задач. Більшість рішень направленні на збереження даних про речі з використанням певної структури (шаблону) для більш зручних подальших маніпуляцій. Тобто людині потрібно використовувати різні рішення та системи в залежності від її діяльності (робота, будинок, тощо).

Головною метою є створення універсальної системи, яка дозволить людині використовуючи один додаток, отримувати доступ до інвентаризаційних даних з різних сфер її діяльності. Дана система дасть можливість заносити та корегувати необхідні дані, структурувати за категоріями, проглядати їх в зручному форматі, швидко знаходити необхідне.

Для вирішення цієї проблеми необхідно провести аналіз наявних рішень та на їх основі використавши найсучасніші інструменти, створити систему яка буде відповідати вимогам.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

Люди так чи інакше займаються інвентаризацією речей, як вдома, так і на роботі. Ми всі зберігаємо продукти харчування, одяг, предмети домашнього вжитку, зображення, інструменти і багато інших товарів. У нас також є дефіцитні та екстрені покупки. Деяким доводиться регулярно викидати вміст холодильника, тому що він був там деякий час та зіпсувався, тому інвентарний контроль – це природне заняття, яким всі займаються, деякі – більш успішно, ніж інші. [1]

Головною проблемою є те. що людина фізично не в змозі зберігати такий об’єм даних в своїй оперативній пам’яті. Тому потрібно задіяти додаткові носії інформації, які дозволять зберігати та отримувати швидкий доступ до необхідної інформації. Носій інформації – це будь-який матеріальний об'єкт або середовище, який використовується людиною, здатне досить тривалий час зберігати (нести) в своїй структурі занесену на нього інформацію. [2]

Один із найдавніших носіїв, який використовується і на теперішний момент – папір. Наприклад, на даний час активно використовуються журнали обліку для ведення інвентаризації. Такий спосіб має певні переваги:

- не потрібно додаткових складних інструментів (лише ручка та журнал)
- папір може зберігатися сотні років при правильних умовах [3]

Але має і ряд недоліків:

- проблематично проводити пошук, бо записів може буди доволі багато, при цьому вони можуть зберігатися в різних журналах.
- доволі проблематично редагувати записи.
- якщо записи ведуться не в друкованому вигляді, то іноді проблематично розуміти почерк людини, яка зробила запис.
- якщо потрібно мати ці дані завжди при собі, то доведеться носити журнал з собою.

- одночасно використовувати журнал може лише одна людина.
- відсутня система контролю прав користувача.

З появою комп’ютерів, та різноманітних програм для роботи з текстом, ведення інвентаризації стає більш комфортним та продуктивним. Наприклад, використання Microsoft Excel. Такий підхід має переваги:

- можливість редагування даних.
- дані зберігаються в читабельному форматі.
- всі дані можна зберігати в одному файлі.
- можна використати одних із готових шаблонів для створення базової структури під різні потреби [4]

Недоліки:

- пошук по файлу можливий, але лише конкретного значення, наприклад, при пошуку «стіл» знаходимо всі записи де зустрічається дане слово, але проблематично знайти всі меблі (стіл, стілець, софа, тощо).
- поріг входження, Excel багатофункціональна та має доволі складний інтерфейс, тому для початку роботи потрібно ознайомитись з хоча б частково з документацією.
- Excel зберігає дані локально, тому дуже не зручно коли над одним фалом працює декілька людей, та не можливо працювати з файлом одночасно.

На теперішній момент існують більш спеціалізовані додатки для ведення інвентаризації:

Home Inventory – інструмент для систематизації відомостей про будинок і майно, які можна використовувати для різних потреб. Має такі функції:

- можливість зберігати квитанції, керівництва по використанню техніки, гарантії, різні замітки та інші важливі документи в даному додатку, що забезпечує швидкий і легкий доступ до них.
- можливість збереження відомостей про цінні речі: Марка, модель, серійний номер, ціна покупки, дата покупки.
- користувач може робити фотографії і сканувати чеки зі свого iPhone або iPad прямо через Home Inventory.
- створення графіку обслуговування будинку, інтегрований з календарем і нагадуваннями.
- додаток досить гнучкий для управління будь-яким типом колекційних предметів.

Головний недоліком додатку являється підтримка лише систем на базі macOS. [5] [6]

StuffKeeper – мобільний додаток, який дозволяє легко каталогізувати і швидко знаходити речі, які користувач використовує нечасто, але без яких не обійтися в потрібний момент. Додаток так само допомагає людям з різними розладами пам'яті, інформаційним перевантаженням. Принцип роботи додатку нагадує матрьошку.

Наприклад: спальна кімната, в ній – гардероб, в ньому – полки, на полицях – одяг. Даний алгоритм вкладень дозволяє легко і зручно каталогізувати будь-які речі і швидко знаходити їх. Має наступний функціонал:

- можливість створити новий предмет.
- редагування даних.
- переміщення і видалення об'єктів.
- можливість вказати місце розташування на фото.
- синхронізація і створення копії даних.
- сортування списку об'єктів.

- можливість вибрати тему оформлення та мову інтерфейсу.
- пошук речей.

Із суттєвих недоліків: безкоштовна версія додатку має ліміт на кількість створених предметів, а також додаток працею лише на мобільних телефонах (iOS, Android). [7]

Sortly – мобільний додаток, який дозволяє організувати домашній інвентар в декількох різних папках і підпапках, щоб спростити пошук. Папки для групування елементів за їх фізичного розташування (в якій кімнаті вони зберігаються) або за типом елементів. Має наступний функціонал:

- можливість з'єднати штрих-коди або QR-ярлики.
- отримати оповіщення про низький рівень запасів і нагадування про дату".
- сканувати вхідні / вихідні елементи.
- працює з ручним сканером.
- створення і друк користувальницьких штрих-кодів або QR-наклейок.
- згенерувати призначені для користувача звіти.
- давати доступ команді.
- можливість відстежувати активність користувачів.
- можливість візуальної інвентаризації.

Як в попередніх представників один із головних недоліків є підтримка лише мобільних телефонів (iOS, Android), та відсутність безкоштовної версії додатку.[8]

Проаналізувавши існуючи рішення були сформовані вимоги до системи, які наслідують переваги попередників та компенсують їх недоліки:

- система має бути легко-доступною для користувача, працювати на різних ОС.
- користувач зможе ввійти систему лише авторизувавшись.

- можливість створення групи користувачів, які одночасно зможуть користуватися системою.
- простий та інтуїтивно зрозумілий інтерфейс.
- реалізувати можливість створення, редагування та видалення даних.
- дати можливість створювати категорії, та групувати дані згідно категорій.
- реалізувати комфортний пошук створених даних.

2 ПОСТАНОВКА ЗАДАЧІ – ВИБІР ТЕХНОЛОГІЙ

Виходячи з вимог до системи, для розробки системи була обрана каскадна модель. Каскадна або водоспадна (waterfall) модель життєвого циклу програмного забезпечення — модель процесу розробки програмного забезпечення, в якій процес розробки виглядає як потік, що послідовно проходить фази аналізу вимог, проектування, реалізації, тестування, інтеграції та підтримки. (див. рис. 2.1) [9] Наступна фаза після завершення фази формування вимог – проектування.

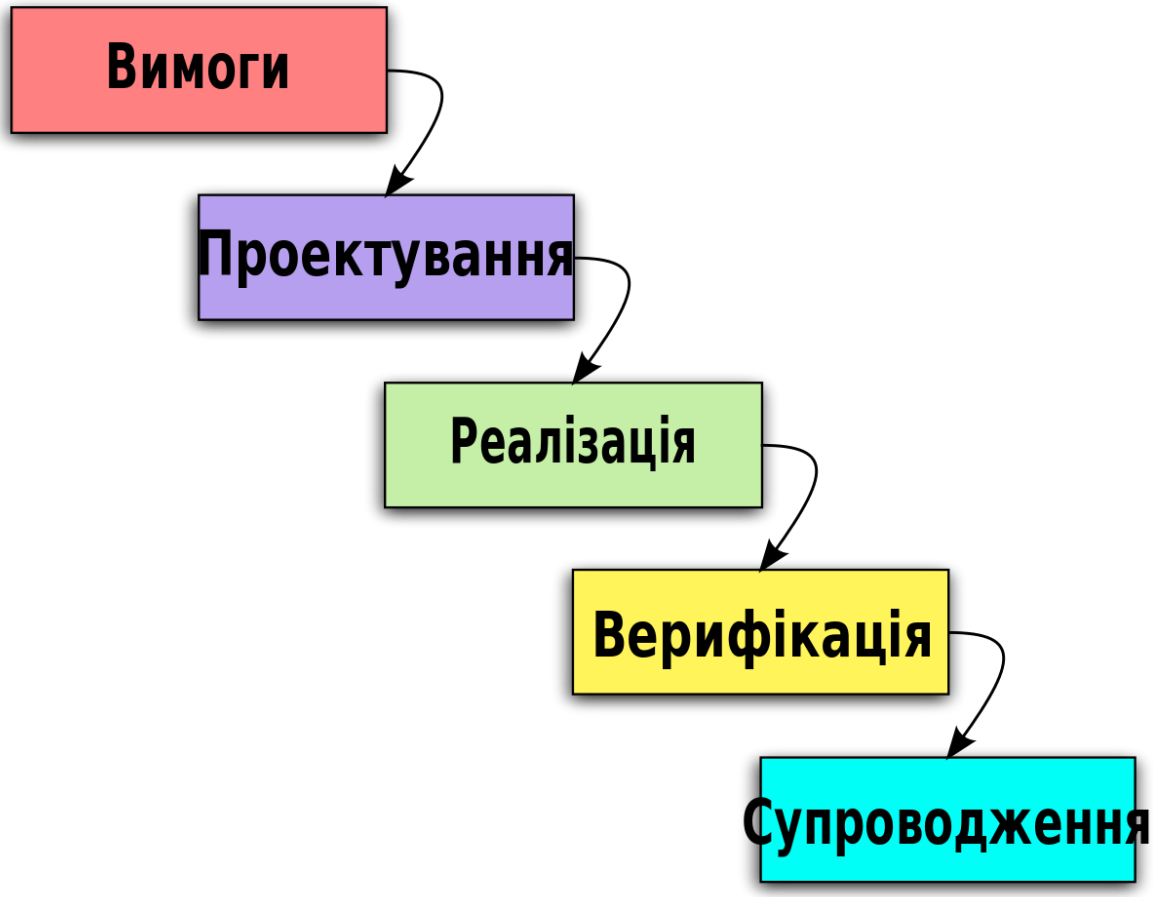

Рисунок 2.1 – Графічне зображення каскадної моделі розробки програмного забезпечення

Щоб забезпечити легкого-доступність системи, було прийнято рішення робити її кросплатформенною, тобто система зможе працювати більш ніж на

одній програмній (в тому числі — операційній системі) або апаратній платформі. Кросплатформність дозволяє суттєво скоротити витрати на розробку нового або адаптацію існуючого програмного забезпечення. [10]. Отже, прийнято рішення розробляти системи в вигляді WEB-додатку, та як згідно дослідженню Digital 2020 Reports, підготовленому компаніями We Are Social Inc. і Hootsuite Inc., число користувачів інтернету по всьому світу збільшується на 9 осіб в секунду. Це означає, що кожен день до світового онлайн-спільноті приєднується понад 800 тисяч людей, які користуються настільними або мобільними пристроями. [11] [12].

Одним із найпоширеніших архітектурних підходів на теперішній час по взаємодії компоненті в WEB розробці є REST (Representational State Transfer) API (Application Programming Interface). REST – архітектурний стиль взаємодії компонентів розподіленого додатка в мережі. REST це узгоджений набір обмежень, що враховуються при проектуванні розподіленої гіпермедіа-системи. В більшості випадків використання REST призводить до підвищення продуктивності і спрощення архітектури. У широкому сенсі компоненти взаємодіють на зразок взаємодії клієнтів і серверів у Всесвітній павутині (див. рис. 2.2) [13]. Переваги REST добре підходять для розробки системи.

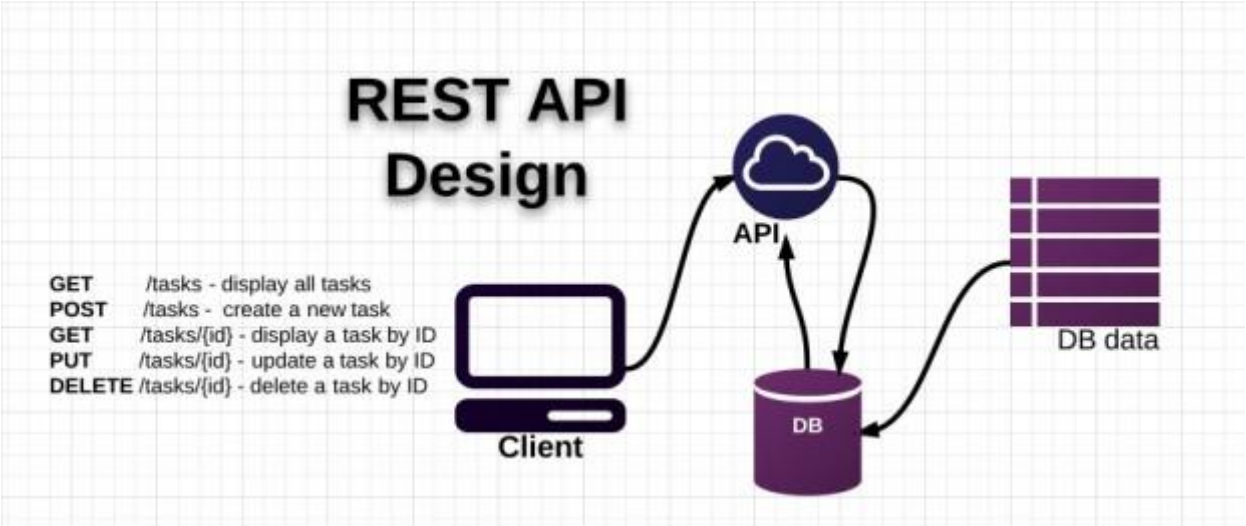

Рисунок 2.2 – Графічне зображення принципу REST API

Згідно вибраного архітектурному патерну, необхідно було реалізувати наступні компоненти:

- клієнтська частина (фронтенд) – інтерфейс з яким взаємодіє користувач, також клієнтська частина реалізує взаємодію з API серверної частини.
- серверна частина (бэкенд) – являється містком між клієнтською частиною та базою даних, та описує основні бізнес процеси додатку.
- база даних – дозволить зберігати і обробляти інформацію в структурованому вигляді.

Проаналізувавши детальніше кожний із вище перерахованих компонентів, були обрані необхідні інструменти для їхньої реалізації.

2.1 Клієнтська частина (фронтенд)

Клієнтська частина виконується в браузері та написана на мові програмування JavaScript, вона використовує HTML (HyperText Markup Language — мова розмітки гіпертексту) та CSS (Cascading Style Sheets – каскадні таблиці стилів) для побудови інтерфейсу. На даний період велику популярність при розробці фронтенду набули SPA додатки.

SPA – це веб-додаток, або веб-сайт, який взаємодіє з користувачем шляхом динамічного переписування поточної веб-сторінки з новими даними з веб-сервера, замість того, щоб браузер за замовчуванням завантажував цілі нові сторінки. Мета – швидший перехід, завдяки якому веб-сайт відчуває себе більше як рідна програма. [14]

У SPA всі необхідні коди HTML, JavaScript та CSS отримуються браузером або з одним завантаженням сторінки, або відповідні ресурси динамічно завантажуються та додаються на сторінку за необхідності, як правило, у відповідь на дії користувача. Сторінка не перезавантажується в будь-який момент процесу, а також не передає управління на іншу сторінку,

хоча хеш розташування або API історії HTML5 можуть бути використані для забезпечення сприйняття та навігації окремих логічних сторінок у програмі. [14]

Ось найбільш відомі приклади застосування SPA:

- Gmail
- Twitter
- Google Maps
- Facebook
- Google Drive

Для створення SPA додатку існують готові бібліотеки, найпопулярніші з них: React, Angular і Vue. Для створення системи немає суттєвої різниці між цими бібліотеками, але була обрана Vue, так як вона має більш детальну та комфортну документацію порівняно з іншими.

Vue – це прогресивний фреймворк для створення користувацьких інтерфейсів. На відміну від фреймворків-монолітів, Vue створений придатним для поступового впровадження. Його ядро в першу чергу вирішує завдання рівня уявлення (view), що спрощує інтеграцію з іншими бібліотеками та існуючими проектами. [15]

Для того, щоб не витрачати час на створення UI (User Interface) елементів, так як унікальний дизайн не є головним пріоритетом, був використаний фреймворк Vuetify, який містить набір готових елементів.

Vuetify – бібліотека з готовими компонентами для користувальницького інтерфейсу, побудована поверх Vue.js. [16]

2.2 Серверна частина (бекенд)

Це програмно-апаратна частина сервісу. Це набір засобів, за допомогою яких відбувається реалізація логіки веб-додатку. Для бекенд можна використовувати будь-які інструменти, доступні на сервері (який, по суті, є просто комп'ютером, налаштованим для відповідей на запити). Це означає, що можна використовувати будь-яку універсальну мову

програмування: Ruby, Java, PHP, Python, JavaScript / Node і т.д. Для розробки була використана Node.js, тому, що це дозволило використовувати одну мову програмування – JavaScript.

Nodejs – програмна платформа, заснована на движку V8 (здійснює трансляцію JavaScript в машинний код), що перетворює JavaScript з вузькоспеціалізованого мови в мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями введення-виведення через свій API, написаний на C++, підключати інші зовнішні бібліотеки, написані на різних мовах, забезпечуючи виклики до них з JavaScript-коду. [17]

Для створення базового каркасу та отримання ряду готових абстракцій, які спрощують створення серверної сторони, використовувався один із найпопулярніших фреймворків Express.js.

Express – фреймворк web-додатків для Node.js, реалізований як вільне і відкрите програмне забезпечення під ліцензією MIT. Він спроектований для створення веб-додатків і API. Де-факто є стандартним каркасом для Node.js. [18]

2.3 База даних

В якості бази даних та СУБД (Система управління базами даних) вибір пав на реляційну базу даних PostgreSQL, вона являться однією з найкращих на теперішний час, а також, що важливо, безкоштовною (opensource). [19]

Також до початки розробки системою контроля версій був обраний Git.

Був сформований набір технологій для розробки, та встановлені необхідні програми для початку роботи:

- Visual Studio Code – IDE (Integrated Development Environment), редактор коду.
- Node.js (версія 14.15.5) – для розробки серверною частити, а також для установки модулів за допомогою NPM (node package manager).
- Express-generator – генератор проекту з використання Express.js.

- Vue-cli (версія 3) – генератор проекту на Vue.
- Git (версія 2.20.1) – система контролю версій або VCS (Version Control System).
- PostgreSQL (версія 13.2) – база даних включно з PgAdmin (графічний інтерфейс).
- Postman – інструмент тестування і розробки API.
- CASE Studio – інструмент проектування баз даних.
- Figma – інструмент для розробки інтерфейсів і прототипування.
- Docker – інструмент контейнеризації додатків.

Був сформований список покрокових дій, що дозволило оцінювати прогрес та визначило пріоритетність задач:

1. Створити два окремих репозиторія для клієнтської і серверної частини.
2. Створити шаблонні проекти.
3. Сформувати структуру взаємозв’язків таблиць для БД.
4. Налаштувати зв'язок БД з серверною частиною.
5. Створити базові моделі таблиць для БД.
6. Реалізувати контролери.
7. Реалізувати серверну маршрутизацію (роути).
8. Реалізувати взаємодію з арі на клієнтській частині.
9. Реалізувати клієнтську маршрутизацію (роути).
10. Створити логотип додатку.
11. Створити сторінки додатку:
    - 11.1. Створити сторінки авторизації та реєстрації.
    - 11.2. Створити сторінку профілю користувача.
    - 11.3. Створити сторінки створення та редагування однієї категорії.
    - 11.4. Створити сторінку категорій.
    - 11.5. Створити сторінки створення та редагування одного предмету.
    - 11.6. Створити сторінку предметів.

11.7. Створити сторінку групи користувача.

12. Провести контейнеризацію системи за допомогою Docker та розгорнути систему на хмарному сервісі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Створення репозиторіїв

В якості платформи зберігання проектів був обраний Bitbucket. [20]

Bitbucket – веб-сервіс для хостингу проектів і їх спільної розробки, заснований на системах контролю версій Mercurial і Git. [21]

Створено проект під назвою «Item manager», та два окремих репозиторія для клієнтської (Item-manager-frontend-vue) та серверної (item-manager-server-node) частин. (див. рис 3.1.1 – 3.1.3)

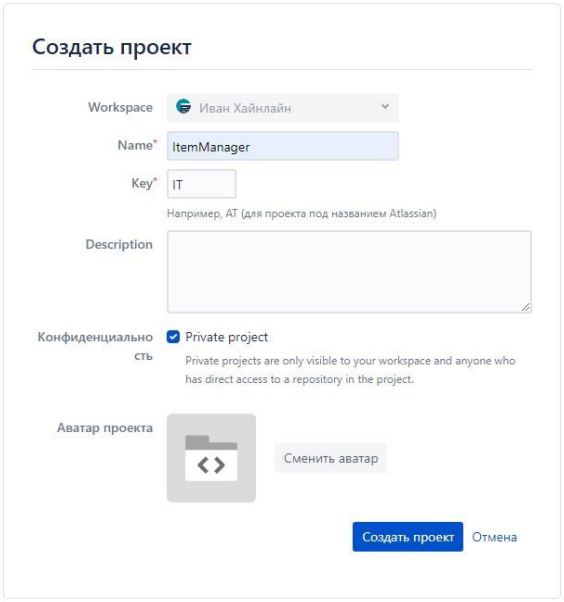

Рисунок 3.1.1 – Створення проекту на Bitbucket

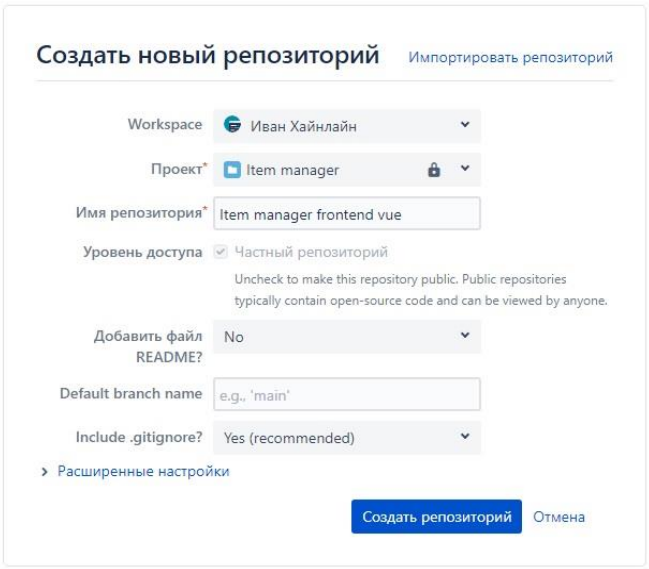

Рисунок 3.1.2 – Створення репозиторію на Bitbucket

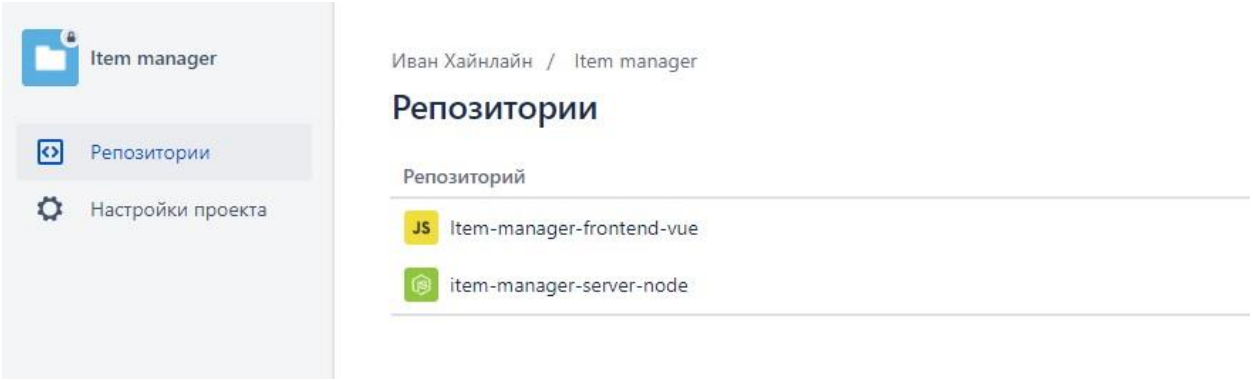

Рисунок 3.1.3 – Створений проект та репозиторії для розробки

Для того, щоб структурувати комміти та покращити їх читабельність були використані наступні правила [22]:

- Відокремлюйте заголовок від тіла порожнім рядком.
- Обмежуйте заголовок 50 символами.
- Пишіть заголовок з великої літери.
- Не пишіть крапку в кінці заголовку.
- Використовуйте наказовий спосіб в заголовку.
- Переходьте на наступний рядок в тілі на 72 символах.
- У тілі відповідайте на питання що і чому, а не як.

В якості стратегії розгалуження гілок репозиторія для системи добре підходять «Довго живучі гілки» (Long-Running Branches). Головною характеристикою якої є наявність двох основних гілок master – в якій знаходиться код, який був або буде випущений в реліз, та develop – в якій відбувається розробка та тестування. Коли функціонал розроблений в develop гілки стає стабільний, його оновлюють на master гілці. (див. рис. 3.1.4) [23]

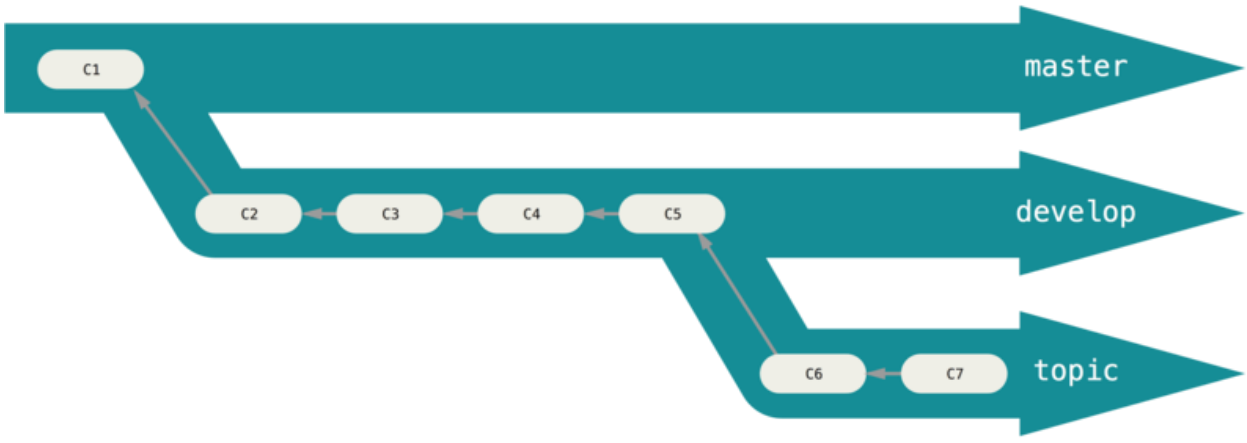

Рисунок 3.1.4 – Графічне зображення стратегії розгалуження гілок «Довго живучі гілки»

3.2 Створення шаблонних проектів

Був створений шаблонний проект для клієнтської частини. Для цього був використаний візуальний інтерфейс Vue, за допомогою команди «vue ui» який став доступний після установки Vue-cli. Створено новий проект та налаштовані необхідні залежності. (див. рис. 3.2.1, 3.2.2)

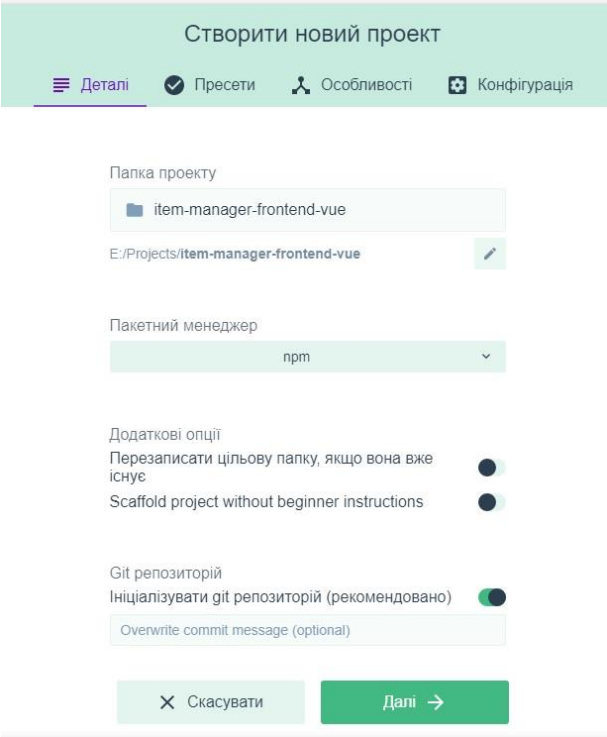

Рисунок 3.2.1 – Створення шаблонного проекту на Vue

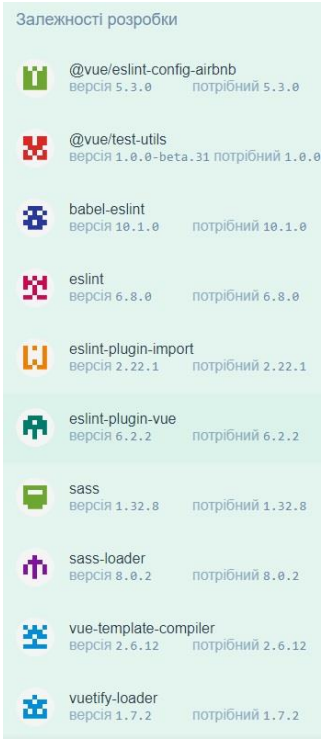

Рисунок 3.2.2 – Залежності розробки проекту на Vue

Наступним був створений шаблонний проект для серверної частини, за допомогою команди, яка стала доступною після установки Express generator:

```
express --no-view item-manager-server-node
```

Далі встановлені всі необхідні залежності для розробки:

```
npm install
```

Отримані шаблони були дещо модернізовані для більш комфортної розробки, коли кодова база проекту стане значно більшою. (див. рис. 3.2.3, 3.2.4)

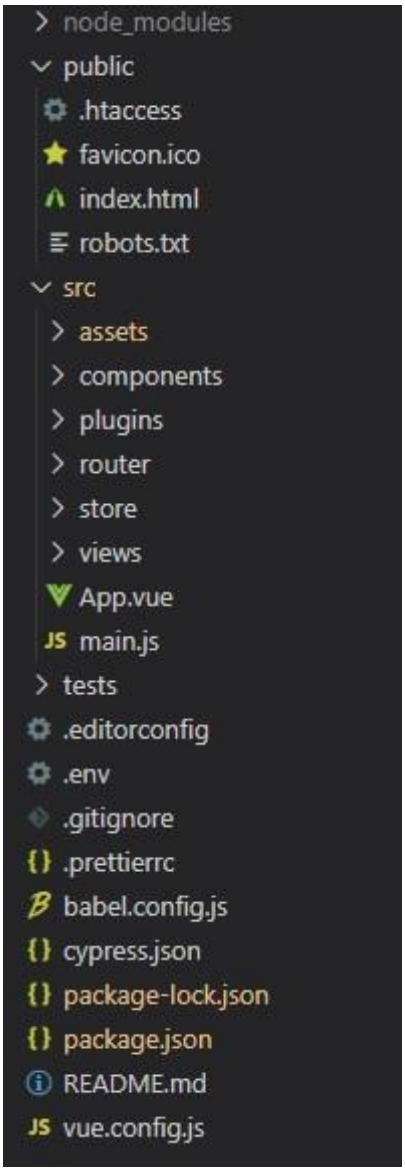

Рисунок 3.2.3 – Структура проекту клієнтської частини

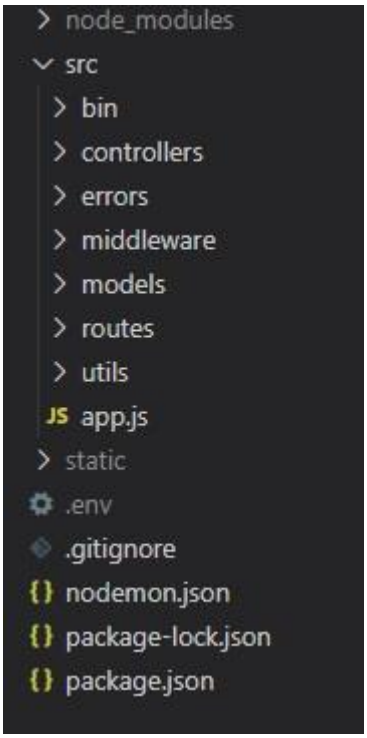

Рисунок 3.2.3 – Структура проекту серверної частини

3.3 Структура взаємозв'язків таблиць для БД

Щоб сформувати структуру бази даних, потрібно виділити основні таблиці та зв'язки між ними. Для цього за допомогою програми CASE Studio було створено DFD (data flow diagram) нульового та першого рівня.

DFD – діаграма, яка наочно відображає перебіг інформації в межах процесу або системи. Для зображення вхідних та вихідних даних, точок зберігання інформації і шляхів її пересування між джерелами і пунктами доставки в таких діаграмах застосовуються стандартні фігури, такі як прямокутники і кола, а також стрілки і короткі текстові мітки. [24]

Перш за все була створена DFD-0 – абстракція, що показує систему як єдиний процес з її зв’язком із зовнішніми об’єктами. (див. рис. 3.3.1)

Рисунок 3.3.1 – DFD-0 рівня

Проаналізувавши DFD-0, були виділені головні таблиці, які потрібні для створення системи:

- Користувачі.
- Група
- Предмети.
- Категорії.
- Предмети та категорії

Наступним кроком була сформована DFD-1 – діаграма, яка розкладається на кілька процесів. На цьому рівні виділяються основні функції системи та декомпозується високорівневий процес 0-рівня DFD на підпроцеси. (див. рис. 3.3.2)

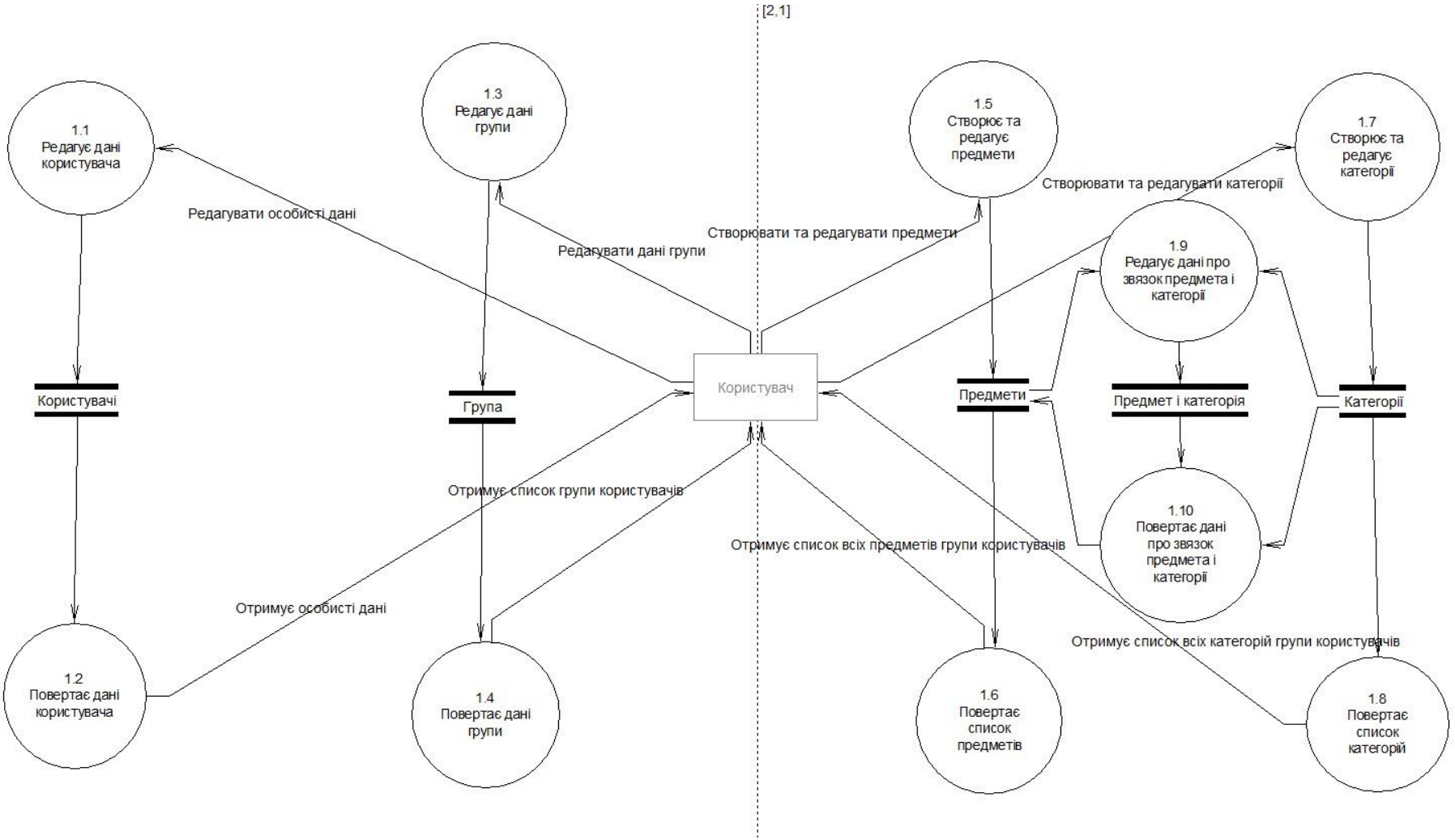

Рисунок 3.3.2 – DFD-1 рівня

Для більш детального розуміння взаємозв’язків між таблицями та визначення необхідних полів для кожної із таблиць, реалізована ERD (entity-relationship diagram) (див. рис. 3.3.3)

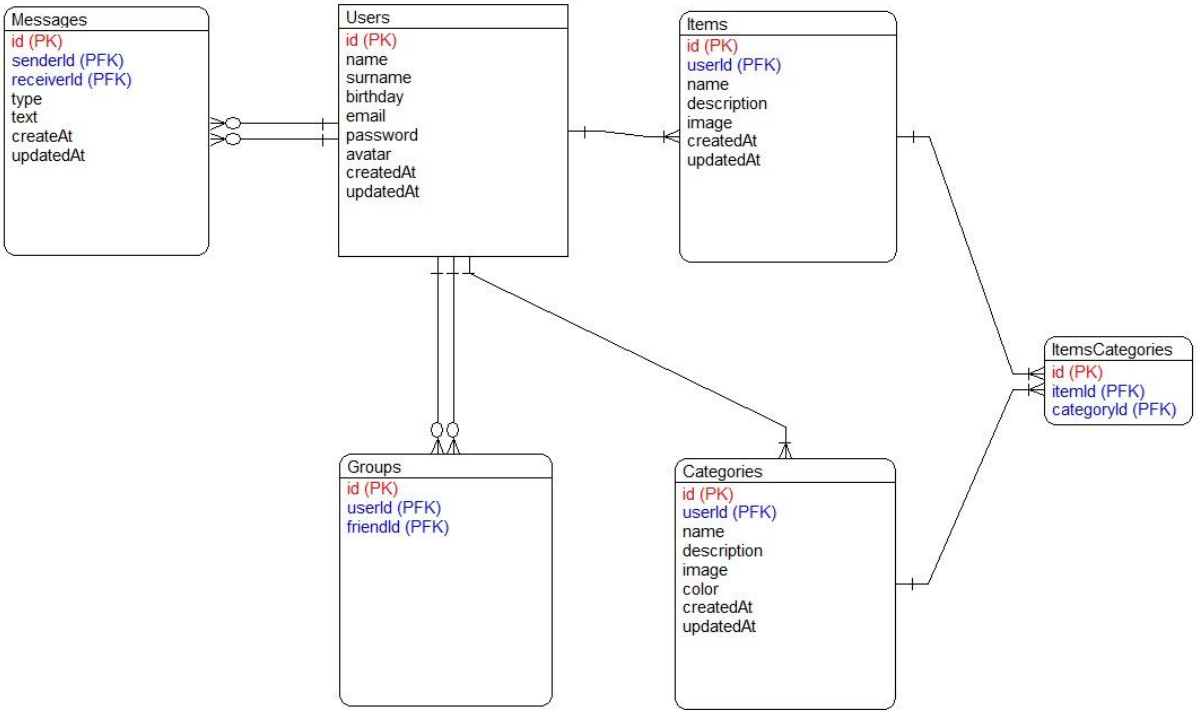

Рисунок 3.3.3 – ERD діаграма структури БД

3.4 Підключення бази даних

Для того, щоб прискорити розробку та власноруч не писати SQL-запити до бази, допомагає технологія ORM (Object-Relational Mapping).

ORM – технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних». Забезпечує роботу з даними в термінах класів, а не таблиць даних, і, навпаки, перетворює терміни і дані класів в дані, придатні для зберігання в СУБД. Також надає інтерфейс для CRUD (create, read, update, delete) – операцій над даними. [25].

Був використаний ORM плагін Sequelize. Sequelize – Node.js ORM на основі промісів для Postgres, MySQL, MariaDB, SQLite та Microsoft SQL Server. [26]

Встановлення Sequelize локально до проекту:

```
npm install --save sequelize
```

Для налаштування підключення створено окремий файл database.js:

```
const { Sequelize } = require('sequelize');

const { env } = process;

const sequelize = new Sequelize(
  env.PGDATABASE,
  env.PGUSER,
  env.PGPASSWORD,
  {
    host: env.PGHOST,
    dialect: 'postgres',
    port: env.PGPORT,
  },
);

module.exports = sequelize;
```

Підключення файлу до точки входу в програму, та з'єднання з базою даних:

```
const database = require('../utils/database');

const start = async () => {
  try {
    await database.authenticate();
    await database.sync();
    …
  }
  …
};
```

Все працює вірно після запуску сервера. (див. рис. 3.4.1)

```
[nodemon] starting `node ./src/bin/www.js`
[nodemon] forking
[nodemon] child pid: 2668
[nodemon] watching 24 files
Executing (default): SELECT 1+1 AS result
```

Рисунок 3.4.1 – Успішне підключення до БД

3.5 Створення базових таблиць для БД

Express базується на патерні MVC. Патерн MVC є одним з поширених патернів, що застосовуються в веб-додатках, та включає наступні компоненти:

- Моделі – визначають структуру і логіку використовуваних даних.
- Уявлення (views) – визначає візуальну частину, як дані будуть відображатися.
- Контролери – обробляють вхідні http(s)-запити, використовуючи для обробки моделі та подання, і відправляє у відповідь клієнту деякий результат обробки, нерідко у вигляді html-коду.
- Система маршрутизація – як додатковий компонент зіставляє запити з маршрутами і вибирає для обробки запитів певний контролер. [27]

Так, як клієнтська частина розробляється в вигляді SPA, то реалізовувати уявлення (views) не потрібно, тобто в відповідь на запит сервер повертає не html-код, а лише дані в JSON-форматі, а за уявлення буде відповідати клієнтська частина.

Згідно ERD діаграми (див. рис. 3.3.3) були створені моделі базових таблиць БД. Для цього потрібно задати набір полів та їх тип згідно документації Sequalize [28].

Модель таблиці користувача (User):

```
const sequelize = require('../utils/database');
const { DataTypes } = require('sequelize');

module.exports = sequelize.define(
  'user',
  {
    name: {
      type: DataTypes.STRING,
      allowNull: true,
    },
    surname: {
      type: DataTypes.STRING,
      allowNull: true,
```

```
    },
    avatar: {
      type: DataTypes.STRING,
      allowNull: true,
    },
    birthday: {
      type: DataTypes.DATEONLY,
      allowNull: true,
    },
    email: {
      type: DataTypes.STRING,
      unique: true,
    },
    password: {
      type: DataTypes.STRING,
    },
  },
  {
    defaultScope: {
      attributes: { exclude: ['password', 'createdAt', 'updatedAt'] },
    },
  },
);
```

Аналогічно були створені моделі для таблиць: Категорії (Category), Предмету (Item), Предмету-Категорії (ItemCategory) та Повідомлень (Message).

Наступним кроком необхідно було сформувати взаємозв'язки для таблиць:

```
const User = require('./User');
const Item = require('./Item');
const Category = require('./Category');
const ItemCategory = require('./ItemCategory');
const Group = require('./Group');
const Message = require('./Message');

User.hasMany(Item);
Item.belongsTo(User);

User.hasMany(Category);
Category.belongsTo(User);

Item.belongsToMany(Category, { through: ItemCategory });
Category.belongsToMany(Item, { through: ItemCategory });
```

```
User.belongsToMany(User, { as: 'friend', through: Group });

User.hasMany(Message);
Message.belongsTo(User);
```

Все працює вірно після запуску серверної частини. (див. рис. 3.5.1, 3.5.2)

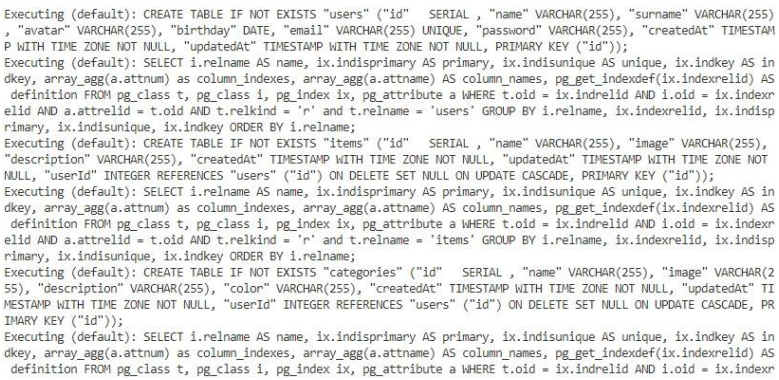

Рисунок 3.5.1 – Вивід до консолі сформованих SQL-команд для створення таблиць

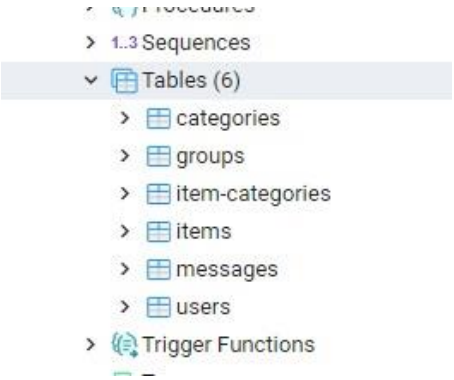

Рисунок 3.5.2 – Список створених таблиць в PgAdmin

3.6 Реалізація контролерів

Список реалізованих контролерів:

- userController
- itemController
- categoryController
- groupController
- messageController

Спершу був створений userController, так як він має більш складну реалізацію. По перше при збереженні користувача до бази необхідно також зберігати і його пароль. Але зберігати його в початкову вигляді не безпечно. Якщо копією бази заволодіють зловмисники, вони одразу отримають доступ до паролів, та спробую використати його, щоб заволодіння аккаунтами користувача. Та навіть без зловмисників, паролі користувачів зможе бачити

адміністратор сайту, який має доступи до БД. Тому пароль необхідно зберігати в хешованому вигляді.

Хешування – це процес незворотній, на відміну від шифрування. Тобто маючи хеш деякої сутності не можливо отримати оригінал даної сутності. Хешування перетворює строку в іншу строку фіксованого розміру, яку можна розглядати як її "відбиток пальця" – єдиний і неповторний.

Отже, алгоритм реєстрацію та авторизації з хешування має наступні кроки:

1. Користувач створює аккаунт.
2. Пароль проходить хешування і записується в базу даних.
3. Коли користувач намагається ввійти до системи, введений ним пароль проходить через хеш-функцію і порівнюється з хешем, збереженим в базі даних.
4. Якщо хеш-кодування збігаються, користувач отримує доступ до захищених розділів. В іншому випадку система запитує авторизовані дані знову.
5. Кроки 3 і 4 повторюються кожен раз, коли користувач проходить авторизацію.[29]

Для хешування використано модуль bcrypt[30]:

```
npm i bcrypt --save
```

Хешування паролю перед записом до бази:

```
const bcrypt = require('bcrypt');
...
const hashPassword = await bcrypt.hash(password, HASH_COUNT);
const user = await User.create({
  email,
  password: hashPassword,
});
...
```

Перевірка введеного користувачем паролю, попередньо хешованого, з хешом який зберігається в базі:

```
const isPassworEqual = bcrypt.compareSync(password, user.password);

if (!isPassworEqual) return next(ApiError.badRequest(errorMessage));
```

Згідно вимог системи, доступ до даних може отримати лише авторизований користувач. Для цього був використаний JWT (JSON Web Token) для перевірки запитів від клієнту на предмет авторизованості. JWT – закодована строчка JSON формату, який визначений у відкритому стандарті RFC 7519. Токен вважається одним з безпечних способів передачі інформації між двома учасниками. Він складається з трьох частин: заголовок (header) із загальною інформацією по токену, корисні дані (payload), такі як id користувача, його роль, тощо. і підписи (signature) (див. рис. 3.6.1) [31].

Алгоритм авторизації за допомогою JWT має таку послідовність:

1. Користувач авторизується за допомогою логіна та паролю.
2. Сервер перевіряє достовірність введених даних і створює JWT та відправляє його користувачеві.
3. Коли користувач робить запит до API додатка, він додає до нього отриманий раніше JWT.
4. Отриманий запит на сервері перевіряється на наявність валідного токена, також з JWT сервер отримує дані про користувача, тим самим зменшує кількість запитів до бази. [31]

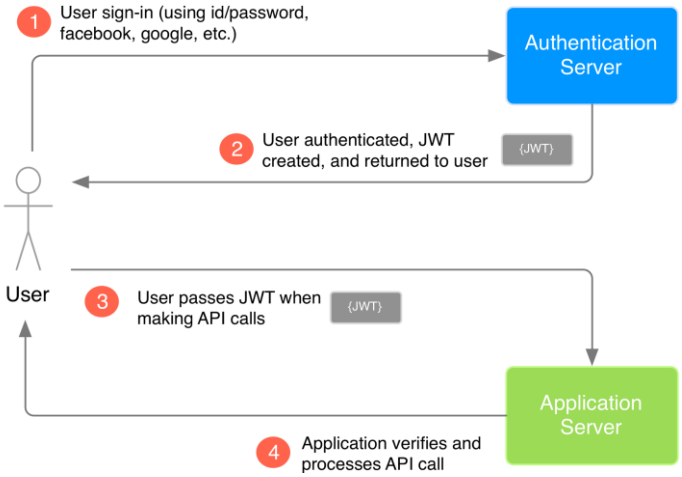

Рисунок 3.6.1 – Графічне зображення алгоритму роботи з JWT

Використано модуль jsonwebtoken [32]:

```
npm i jsonwebtoken–save
```

Функція для створення токену:

```
const jwt = require('jsonwebtoken');
…
const generateJwt = (data) =>
  jwt.sign(data, process.env.JWT_SECRET_KEY, {
    expiresIn: '24h',
  });
```

Створення токену та відправка його на клієнт:

```
…
const token = generateJwt({ id: user.id, email: user.email });

return res
  .status(201)
  .json({ message: 'User logged in', data: { user, token } });
```

Далі був створений authMiddleware, який перевіряє на наявність та валідність токена, та використовується для захисту необхідних роутів. Тобто authMiddleware при необхідності обриває виконання подальшого коду та повертає помилку на клієнт:

```
const jwt = require('jsonwebtoken');
const ApiError = require('../errors/ApiError');

module.exports = (req, res, next) => {
  if (req.method === 'OPTIONS') next();
  const errorMessage = 'User unauthorized';

  try {
```

```
    const token = req.headers.authorization.split(' ')[1];

    if (!token) return next(ApiError.unauthorization(errorMessage));

    const decoded = jwt.verify(token, process.env.JWT_SECRET_KEY);
    req.user = decoded;

    next();
  } catch (e) {
    return next(ApiError.unauthorization(errorMessage));
  }
};
```

Згідно вимог до системи та ERD (див. рис. 3.3.3) користувач зможе завантажити зображення для аватарки, предмету та категорії. Зберігання цілого зображення в базі даних є антипатерном. Для цього був використаний наступних принцип:

1. Після отримання сервером зображення, створюється унікальна назва для зображення.
2. Отримана назва зберігається в базі даних.
3. Зображення зберігається під новим унікальним іменем до директорії, яка відповідає за видачу статичних файлів з серверу.

Таким чином зменшиться навантаження на базу. Також це дозволить серверу та клієнту закешувати ці дані, що досить істотно збільшить продуктивність.

Наступний код відповідає за видачу сервером статичних даних:

```
const path = require('path');
…
app.use(express.static(path.resolve(__dirname, '../static')));
```

Для більш комфортної роботи з запитами які містять файли, був використаний модуль multer [33]:

```
npm i multer --save
```

Підключення модулю до проекту:

```
const multer = require('multer');
const upload = multer();
…
```

```
app.use(upload.any());
```

Для генерації унікального імені зображення використано модуль uuid [34]:

```
npm i uuid -save
```

Був створений окремий модуль для роботи з зображеннями:

```
const path = require('path');
const uuid = require('uuid');
const { unlink, writeFile } = require('fs/promises');

const STATIC_FOLDER_PATH = '../../static/';

const getImageName = (img, folderName = '') => {
  const extension = img.originalname.split('.').pop();
  return `${folderName}/${uuid.v4()}.${extension}`;
};

module.exports = {
  saveImage(image, folderName) {
    const imageName = getImageName(image, folderName);
    writeFile(path.resolve(__dirname,
                           STATIC_FOLDER_PATH,
                           imageName,
              ), image.buffer);
    return imageName;
  },
  deleteImage(imageName) {
   return unlink(path.resolve(__dirname, STATIC_FOLDER_PATH, imageName));
  },
};
```

Даний модулю використовується при обробці запитів різними контролерами:

```
const { saveImage, deleteImage } = require('../utils/images');
…

if (avatar && avatar !== newAvatar) {
  await deleteImage(avatar).catch((err) =>
    next(ApiError.badRequest(err.message)),
  );
  avatar = null;
}


if (req.files && req.files[0]) {
  avatar = saveImage(req.files[0], AVATARS_FOLDER_NAME);
}
…
```

3.7 Реалізація серверних маршрутів (роути)

Маршрутизація визначає, як сервер відповідає на клієнтський запит до конкретної адреси (URI).

Був створений окремий файл роут для кожного контролера:

- userRouter
- itemRouter
- category Router
- groupRouter
- message Router

Наприклад userRouter:

```
const userController = require('../controllers/userController');
const router = require('express').Router();
const authMiddleware = require('../middleware/authMiddleware');

router
  .post(
    '/registration',
    userController.registrValidation,
    userController.registration,
  )
  .post('/login', userController.login)
  .get('/auth', authMiddleware, userController.check)
  .post(
    '/',
    authMiddleware,
    userController.updateValidation,
    userController.update,
  );

module.exports = router;
```

Підключення всіх роутів до єдиного модулю:

```
const Router = require('express');

const router = new Router();
const userRouter = require('./userRouter');
const itemRouter = require('./itemRouter');
const categoryRouter = require('./categoryRouter');
const groupRouter = require('./groupRouter');
const messageRouter = require('./messageRouter');
```

```
router.use('/user', userRouter);
router.use('/item', itemRouter);
router.use('/category', categoryRouter);
router.use('/group', groupRouter);
router.use('/message', messageRouter);

module.exports = router;
```

Та підключення до додатку:

```
const indexRouter = require('./routes/index');
…
app.use('/api', indexRouter);
```

Для тестування маршрутизації використовувався Postman. Був створений окремий проект ItemManagerNode, та необхідні колекції запитів. (див. рис. 3.7.1)

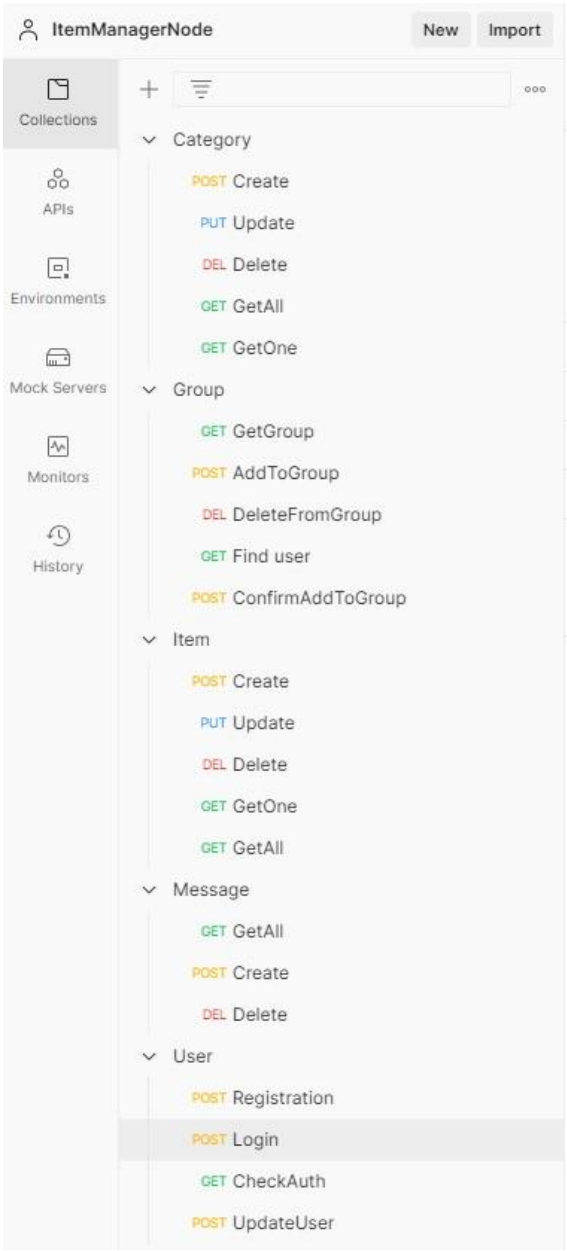

Рисунок 3.7.1 – Список запитів створених в Postman

Так як більша частина роутів доступна лише для авторизованого користувача, то для більш зручнішого тестування, був доданий автоматичний запис токена (див. рис. 3.7.2)

Рисунок 3.7.2 – Автоматичний запис токена до змінних Postam після авторизації

Проведено тестування всіх роутів серверу. (див. рис. 3.7.3)

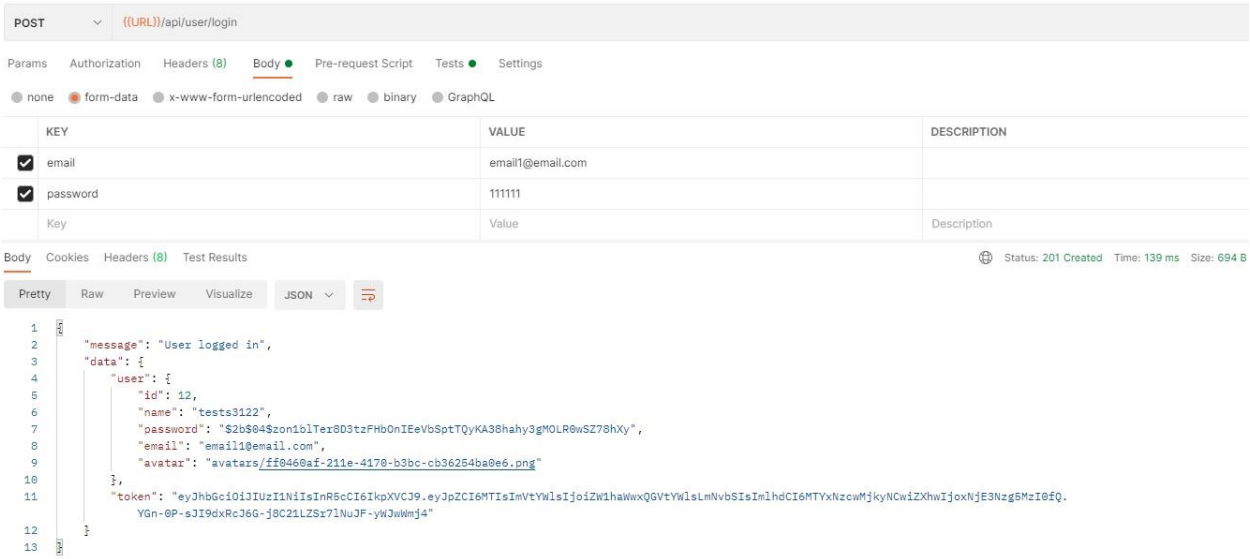

Рисунок 3.7.3 – Успішний запит до серверу з використанням Postman

3.8 Реалізація взаємодії клієнтської частини з API серверу.

Спочатку створено функцію, яка будуємо URL (маршрут) для запиту до серверу:

```
const SERVER_URL = process.env.VUE_APP_SERVER_URL;
const getPath = (path) => (path ? `/${path}` : '');
const getApiUrl = (prefix = '') =>
     (path = '') => `${SERVER_URL}/api/${prefix}${getPath(path)}`;
```

VUE_APP_SERVER_URL – являється змінною оточення, тобто при запуску додатку на іншому сервері достатньо буде змінити файл .env, це дозволить не руйнувати цілісність вихідного коду.

Були створені файли з описом маршрутів для кожного роуту:

- user
- category
- item

- group
- message

Наприклад user:

```
import { getApiUrl } from './getUrl';

const getUserUrl = getApiUrl('user');

export default {
  registration: {
    method: 'POST',
    url: getUserUrl('registration'),
  },
  login: {
    method: 'POST',
    url: getUserUrl('login'),
  },
  checkAuth: {
    method: 'GET',
    url: getUserUrl('auth'),
  },
  update: {
    method: 'POST',
    url: getUserUrl(),
  },
};
```

Для відправки запитів до серверу використовується плагін axios. Axios – JavaScript-бібліотека для виконання HTTP-запитів в Node.js, та XMLHttpRequests в браузері. Встановлення axios [35]:

```
npm i axios –save
```

Далі був створений окремий модуль з типовими запитами на основі axios, щоб зменшити повторюваність коду:

```
import axios from 'axios';
import qs from 'qs';
import { serialize } from 'object-to-formdata';

export const commonRequest = (config) => {
  const { method, url } = config;

  return axios({
    url,
    method,
```

```
  });
};

export const getRequestWithParams = (config, query, opt = {}) => {
  const { method, url } = config;

  return axios({
    url,
    method,
    params: query,
    paramsSerializer(params) {
      return qs.stringify(params, { arrayFormat: 'brackets' });
    },
    ...opt,
  });
};

export const sendDataRequest = (config, data) => {
  const { method, url } = config;

  return axios({
    url,
    method,
    data: serialize(data, { indices: true }),
  });
};
```

На основі даного модуля були реалізовані запити для кожного із роутів. Наприклад user:

```
import URLS from '@/assets/js/config/urls/user';
import { commonRequest, sendDataRequest } from './commonRequests';

export const axiosRgistration = (data) =>
     sendDataRequest(URLS.registration, data);
export const axiosLogin = (data) => sendDataRequest(URLS.login, data);
export const axiosCheckAuth = () => commonRequest(URLS.checkAuth);
export const axiosUserUpdate = (data) =>
     sendDataRequest(URLS.update, data);
```

Отримані дані будуть зберігатися в глобальному сховищі, та як деякі дані використовуюся в різних компонентам клієнта, це дозволить зменшити кількість запитів до бази даних та покращить швидкість роботи додатку. Використовується офіційний плагін Vue – Vuex. Vuex – централізоване сховище даних для всіх компонентів програми з правилами, що гарантують,

що стан може бути змінено тільки передбачуваним чином, а також яке реактивно реагує на зміни стану. Тобто при зміні даних, сховище автоматично оновить всі компоненти, які пов'язані з цими даними. Алгоритм роботи Vuex складається з наступних 4 сутностей (див. рис. 3.8.1) [36]:

- Стан (State) – це дані, які можуть бути змінені лише в сховищі.
- Геттери (Getters) – динамічні дані, які формуються на основі стану.
- Мутації (Mutations) - виконують оновлення стану. Мутація є синхронною операцією (програма чекає, поки мутація не буде завершена). Мутації повинні бути єдиним способом оновлення стану, щоб забезпечити управління станом передбачуваним.
- Дії / Екшени (Actions) – містять бізнес-логіку і не дбають про оновлення стану безпосередньо. Причина в тому, що дії є асинхронними, це корисно при роботі API серверу. Дія в свою чергу викликає мутацію.

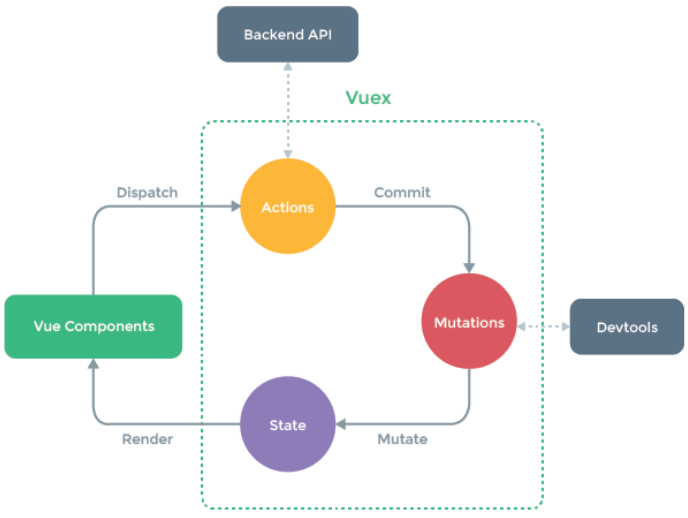

Рисунок 3.8.1 – Алгоритм роботи Vuex

Отже, опираючись на логіку роботи сховища, були створені необхідні модулі для роботи з API і для зберігання станів додатку. Наприклад модуль user:

```
import {
  axiosLogin,
  …
} from '@/assets/js/api/user';
import { showError, showSuccess } from '@/store/snackBar';
import { formatDate } from '@/assets/js/lib/date';

export default {
  namespaced: true,
  state: {
    user: null,
    token: null,
  },
  getters: {},
  mutations: {
    SET_USER(state, val) {
      const user = { ...val };
      if (user.birthday)
           user.birthday = formatDate(new Date(user.birthday));
      state.user = user;
    },
    …
  },
  actions: {
    …
    login({ commit }, data) {
      return axiosLogin(data)
        .then((res) => {
          const { message } = res.data;
          showSuccess(commit, message);
          const { user, token } = res.data.data;
          commit('UPDATE_TOKEN', token);
          commit('SET_USER', user);
          return Promise.resolve(user);
        })
        .catch((err) => {
          showError(commit, err.response.data.message);
          return Promise.reject(err.response.data);
        });
    },
    …
};
```

Зроблений перший запит з клієнту, все відпрацювало вірно. (див. рис. 3.8.2, 3.8.3)

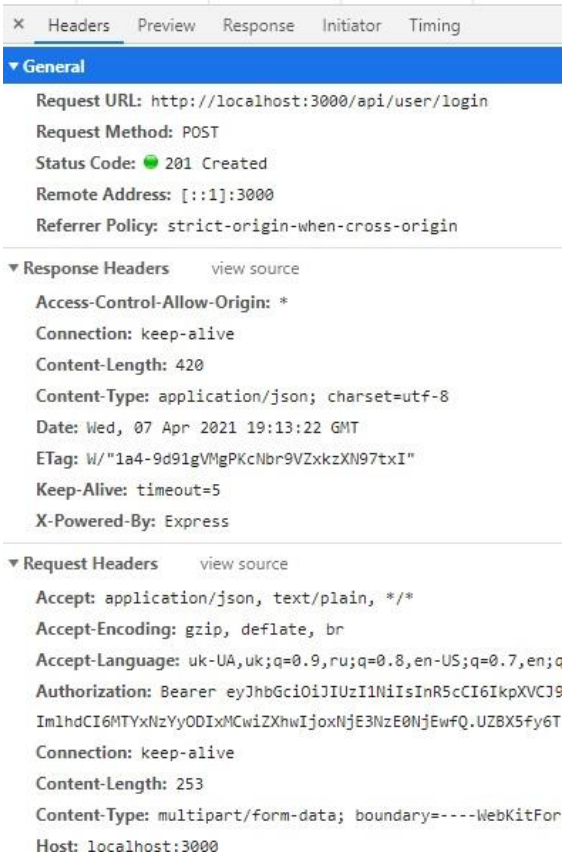

Рисунок 3.8.2 – Успішний запит з клієнту, вкладка (Headers)

Рисунок 3.8.3 – Успішний запит з клієнту, вкладка (Preview)

3.9 Реалізація клієнтської маршрутизації

Згідно вимог до системи користувач отримує доступ до основних функцій додатку лише авторизувавшись. Аналогічно з блокуванням запитів для не авторизованого користувача блокуються і частина сторінок. Було прийнято рішення зобразити схематично ієрархію сторінок, та визначити, які сторінки доступні лише для авторизованих користувачів, а які лише для відвідувачів. (див. рис. 3.9.1)

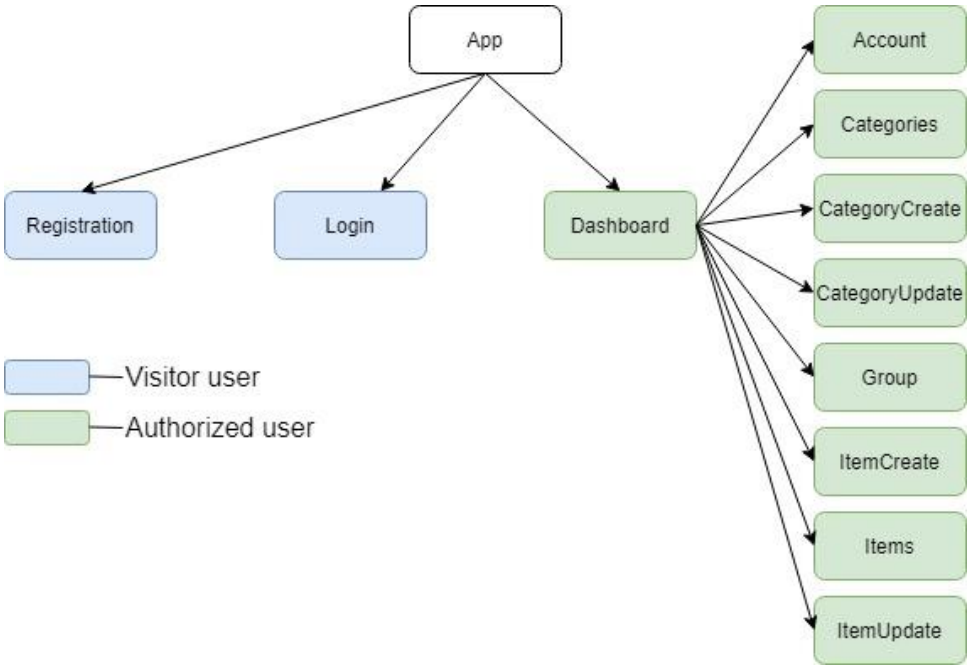

Рисунок 3.9.1 – Графічне зображення ієрархії сторінок клієнтської частини

Для реалізації маршрутизації був сформований конфігураційний файл з описом всіх сторінок, та визначено який користувач має дозвіл до кожної сторінки, згідно схеми (див. рис. 3.9.1):

```
import permitions from './permitions';

export default {
  dashboard: {
    path: '/',
    name: 'Dashboard',
    title: 'Dashboard',
    meta: {
      [permitions.forAuth]: true,
    },
  },
  …
  login: {
    path: '/login',
    name: 'Login',
    title: 'Login',
    meta: {
      [permitions.forVisitor]: true,
    },
  },
  …
};
```

Наступник кроком була сформована необхідна ієрархія орієнтуючись на схему (див. рис. 3.9.1). Для цього використано офіційний модуль Vue – Vue Router. Vue Router надає наступні можливості:

- Вкладені маршрути / уявлення.
- Модульна конфігурація маршрутизатора.
- Доступ до параметрів маршруту, query, wildcards.
- Анімація переходів уявлень на основі Vue.js.
- Зручний контроль навігації.
- Автоматичне проставляння активного CSS класу для посилань.
- Режими роботи HTML5 history або хеш, з авто-перемиканням в IE9.
- Настроюється поведінка прокрутки сторінки. [37]

Router index файл:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import PAGES from '@/assets/js/config/pages/index';
import Dashboard from '@/views/Dashboard/index.vue';

Vue.use(VueRouter);

const routes = [
  {
    path: PAGES.dashboard.path,
    name: PAGES.dashboard.name,
    meta: PAGES.dashboard.meta,
    component: Dashboard,
    children: [
      {
        path: PAGES.items.path,
        name: PAGES.items.name,
        meta: PAGES.items.meta,
        component: () => import('@/views/Dashboard/childrens/Items.vue'),
      },
      …
    ],
  },
  {
    path: PAGES.login.path,
```

```
    name: PAGES.login.name,
    meta: PAGES.login.meta,
    component: () => import('@/views/Login.vue'),
  },
  …
];

const router = new VueRouter({
  mode: 'history',
  routes,
});

export default router;
```

Під час кожної спроби користувача перейти на іншу сторінку необхідно перевіряти чи він авторизований, при спробі перейти на заборонену сторінку, користувача буде перенаправляти на одну із сторінок за замовчування, для авторизованого – Items, для не авторизованого – Login:

```
…
router.beforeEach(async (to, from, next) => {
  const user = await store.dispatch('user/checkAuth').catch((err) => {
    console.error(err);
  });
  const isAuth = !!user?.id;

  if (to.meta[permitions.forVisitor] && isAuth) {
    return next({ name: pages.items.name });
  }

  if (to.meta[permitions.forAuth] && !isAuth) {
    return next({ name: pages.login.name });
  }

  return next();
});
…
```

3.10 Створення логотипу додатку

Впровадження логотипу дозволить покращити візуальне сприйняття користувачем і також додає індивідуальності додатку. Для того, щоб не витрачати на це багату часу, за основу було використано готове векторне

зображення, з сайту https://www.flaticon.com, та модифіковане за допомогою програми Figma. (див. рис. 3.10.1)

Рисунок 3.10.1 – Логотип додатку

На основі логотипу побудований Favicon для додатку. Favicon – значок веб-сайту чи веб-сторінки. Відображається браузером у вкладці перед назвою сторінки, а також в якості картинки поруч із закладкою, у вкладках і в інших елементах інтерфейсу. [38] Для цього був використаний сервіс https://realfavicongenerator.net, за допомогою якого згенеровано зображення необхідних розмірів. Підключення до проекту:

```
<link rel="icon" href="<%= BASE_URL %>favicon/favicon.ico">
<link rel="apple-touch-icon" sizes="180x180"
     href="<%= BASE_URL %>favicon/apple-touch-icon.png">
<link rel="icon" type="image/png" sizes="32x32"
     href="<%= BASE_URL %>favicon/favicon-32x32.png">
<link rel="icon" type="image/png" sizes="16x16"
     href="<%= BASE_URL %>favicon/favicon-16x16.png">
<link rel="manifest" href="<%= BASE_URL %>favicon/site.webmanifest">
<link rel="mask-icon"
     href="<%= BASE_URL %>favicon/safari-pinned-tab.svg" color="#5bbad5">
<meta name="msapplication-TileColor" content="#603cba">
<meta name="theme-color" content="#ffffff">
```

Результат вірно налаштованого favicon. (див. рис. 3.10.2)

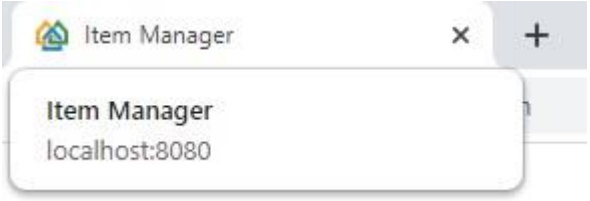

Рисунок 3.10.2 – Відображення браузером favicon додатку

3.11 Створення сторінок додатку

Vue для побудови сторінок використовує компонентний підхід. Компонентний підхід дозволяє уникнути мішанини коду, зменшити його повторюваність та чітко вибудовувати архітектуру програми. Сторінка також являється компонентом, який завжди можна розбити на менші складові. Таким чином кожний компонент простіше підтримувати, а при необхідності повторювати розбиття всередині компонента на ще менші частини.

Перевагою подібного розбиття на компоненти буде зручність в підтримці – відбувається декомпозиція логіки додатку на окремі не залежні частини, що значно покращую нарощуванню подальшого фунціоналу та тестуванню. Також ізольованість компонентів позбавить від появи конфліктів з іншими частинами програми. (див. рис. 3.11.1) [39]

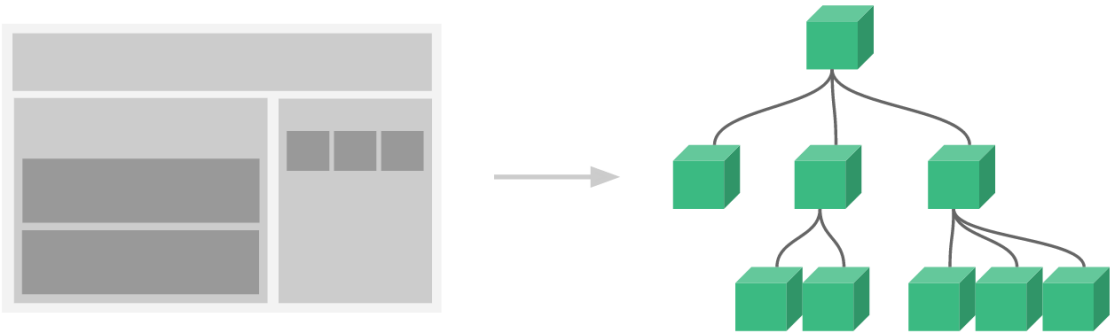

Рисунок 3.11.1 – Графічне зображення компонентного підходу Vue

Згідно документації Vue одно-файловий компонент складається з трьох основних частин:

- <template> – в більшості складається з HTML розмітки компоненту, але також включає і логічні операції.
- <script> – містить скриптову частин, яка формує логіку компонента.
- <style> – містить опис CSS стилів для компоненту. [40]

3.11.1 Створення сторінок авторизації та реєстрації

Дані сторінки доволі прості, та мають схожу реалізацію, головним елементом сторінок є форма для відправки даних на сервер. Під час розробки серверної частини, була реалізована перевірка даних (валідація) для всіх запитів які зберігають дані від клієнта, щоб наприклад, користувач не зміг ввести не вірного формати email чи пустий пароль. Тобто серверна перевірка необхідна для безпеки систем. Але гарною практикою являється створення перевірки також і для клієнтської частини. Адже це дозволяє зменшити кількість запитів та покращує зручність використання для користувача (прямий зворотний зв'язок без зворотного зв'язку з сервером). Для створення форми та елементів форми використаємо Vuetifyjs. Згідно документації фрейворку був створений модуль з описом всіх необхідних валідаційних правил, які в подальшому використаємо для перевірки даних для інших форм [41]:

```
const EMAIL_REG = /\S+@\S+\.\S+/;
const required = (v) => !!v || 'Field is required';

export const checkRule = (val, rules) => {
  const errorMsgs = rules
     .map((rule) => rule(val)).filter((v) => typeof v === 'string');
  const [firstMsg] = errorMsgs;
  return firstMsg || '';
};

export default {
  required: [required],
  name: [
      required,
      (v) => v.length > 3 || 'Name must be more than 3 characters long',
  ],
  email: [
      required,
      (v) => EMAIL_REG.test(v)
           || 'Email should be in the format "test@test.com"',
  ],
  password: [
      required,
      (v) => v.length >= 6 || 'Password must be more than 5 symbols',
  ],
};
```

Далі були створені окремі компоненти форм для авторизації (Login) та реєстрації (Registration). Наприклад, форма авторизації:

```
<template>
  <v-form ref="form" v-model="isValid" class="im-login-form">
    <v-text-field
      class="mb-4"
      v-model="user.email"
      outlined
      label="Email"
      hide-details="auto"
      :rules="RULES.required"
      :prepend-inner-icon="ICONS.label"
    />

    <v-text-field
      class="mb-4"
      v-model="user.password"
      outlined
      type="password"
      label="Password"
      hide-details="auto"
      :rules="RULES.required"
      :prepend-inner-icon="ICONS.key"
    />

    <v-btn
      color="success"
      large
      width="100%"
      :loading="isLoading"
      :disabled="isLoading"
      @click="submitHandler"
    >
      Login
    </v-btn>
  </v-form>
</template>

<script>
import { mapActions } from 'vuex';
import ICONS from '@/assets/js/config/icons/index';
import PAGES from '@/assets/js/config/pages/index';
import RULES from '@/assets/js/lib/rules';

export default {
  name: 'ImLoginForm',
  data() {
    return {
      isValid: false,
      isLoading: false,
      user: {
        login: '',
        password: '',
      },
    };
```

```
  },
  methods: {
    ...mapActions({
      login: 'user/login',
    }),
    submitHandler() {
      if (!this.$refs.form.validate()) return;
      this.isLoading = true;
      const { email, password } = this.user;
      this.login({ email, password })
        .then(() => {
          this.$router.push({ name: PAGES.items.name });
        })
        .finally(() => {
          this.isLoading = false;
        });
    },
  },
  computed: {
    ICONS() {
      return ICONS;
    },
    RULES() {
      return RULES;
    },
  },
};
</script>

<style lang="scss">
.im-login-form {
}
</style>
```

Після готовності всіх необхідних компонентів для побудови були сформовані відповідні сторінки. Наприклад Login:

```
<template>
  <div class="im-login">
    <v-container class="im-login__inner">
      <v-icon size="200" class="mx-auto d-block mb-4">$logo</v-icon>
      <im-login-form class="pb-3"/>
      <v-btn text width="100%" :to="{ name: PAGES.registration.name }">
        Registration
      </v-btn>
    </v-container>
  </div>
</template>

<script>
import ImLoginForm from '@/components/forms/LoginForm.vue';
import PAGES from '@/assets/js/config/pages/index';
```

```
export default {
  name: 'ImLogin',
  components: {
    ImLoginForm,
  },
  computed: {
    PAGES() {
      return PAGES;
    },
  },
};
</script>

<style lang="scss">
@import '@/assets/scss/general/index';

.im-login {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 100%;

  &__inner {
    @include media('>tiny') {
      max-width: 400px;
    }

    @include media('<=tiny') {
      width: 100%;
    }
  }
}
</style>
```

Візуально отримано наступний результат (див. рис. 3.11.1.1, 3.11.1.2)

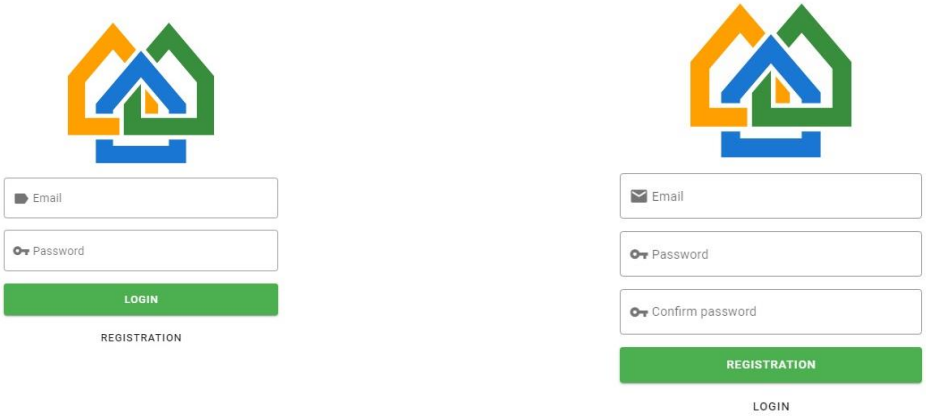

Рисунок 3.11.1.1 – Сторінка

авторизації

Рисунок 3.11.1.2 – Сторінка реєстрації

Так як подальша реалізація дещо схожа та кодова частина наступних сторінок доволі велика, далі буде лише короткий опис сторінки та зображення остаточного результату.

3.11.2 Створення сторінки профілю користувача

Головна частина сторінки профілю складає форма (LoginForm.vue) для редагування даних користувача. Для її реалізації знадобилось додатково реалізувати компонент для завантаження зображення користувачем (InputImg.vue). (див. рис. 3.11.2.1, 3.11.2.2)

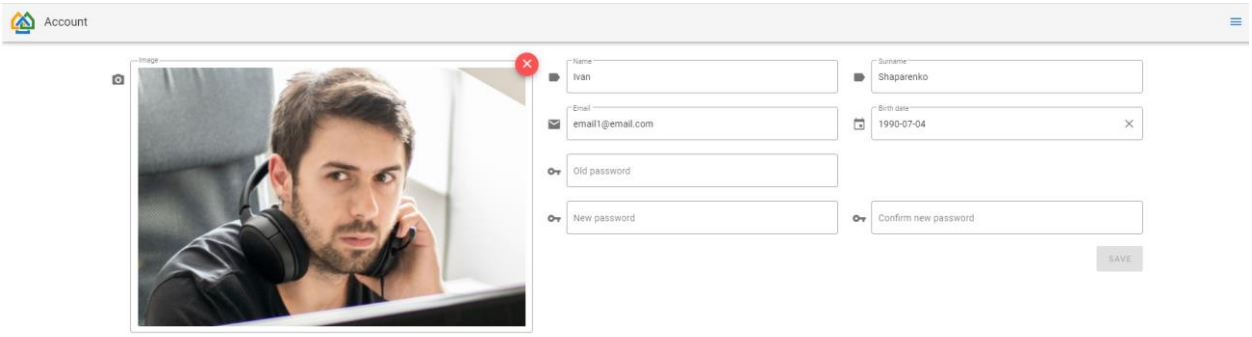

Рисунок 3.11.2.1 – Сторінка профілю користувача (вигляд на персональному комп’ютері)

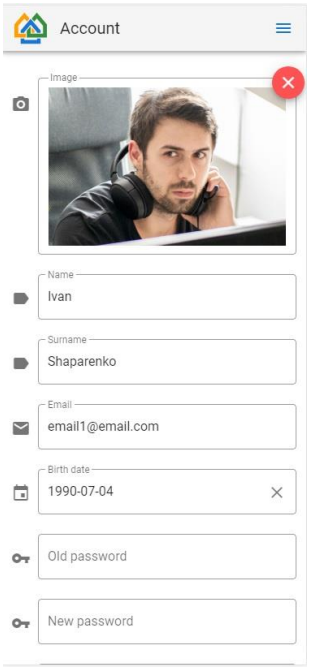

Рисунок 3.11.2.2 – Сторінка профілю користувача (вигляд на мобільному телефоні)

3.11.3 Реалізація сторінки створення та редагування однієї категорії

Структури даних сторінок дуже схожі, основна частина складає форма (CategoryForm.vue). Різниця лише в тому, що сторінки по різному реагують на відправку даних з форми, як очевидно з назви одна сторінка дозволяє створити предмет, а інша відредагувати або видалити. (див. рис. 3.11.3.1, 3.11.3.2)

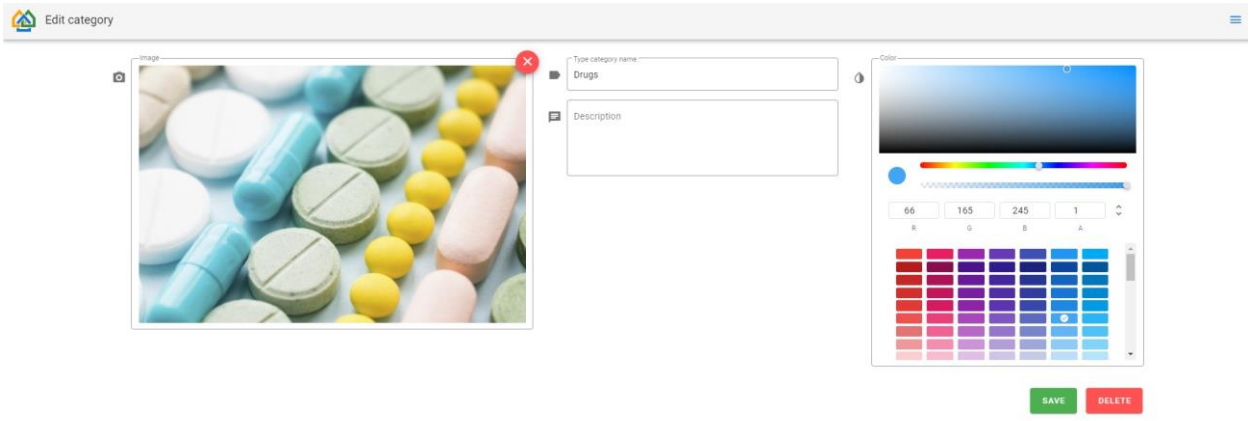

Рисунок 3.11.3.1 – Сторінка редагування категорії (вигляд на персональному комп’ютері)

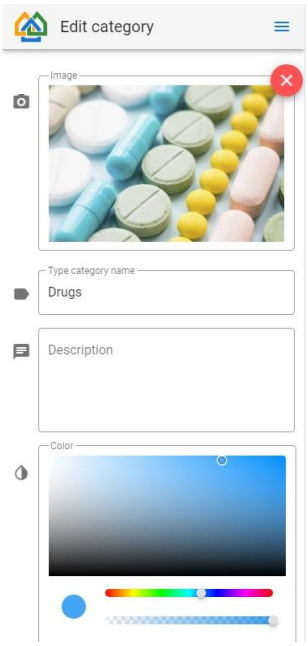

Рисунок 3.11.3.2 – Сторінка редагування категорії (вигляд на мобільному телефоні)

3.11.4 Створення сторінки категорій

Основна мета даної сторінки вивести список всіх доступних категорій користувачеві. Додатково для сторінки реалізований компонент однієї категорії (CategoryCard.vue), основна функція якого стисло вивести актуальну інформації про категорію. (див. рис. 3.11.4.1, 3.11.4.2)

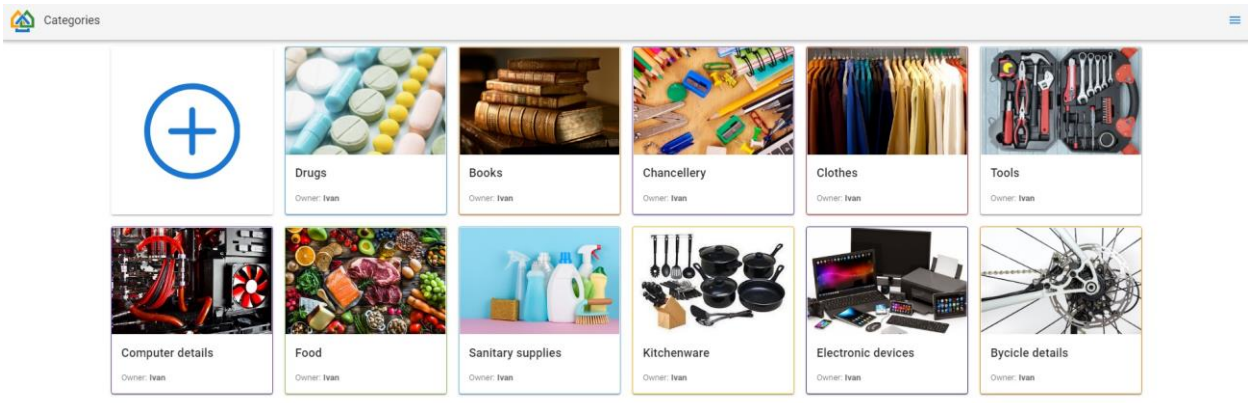

Рисунок 3.11.4.1 – Сторінка категорій (вигляд на персональному комп’ютері)

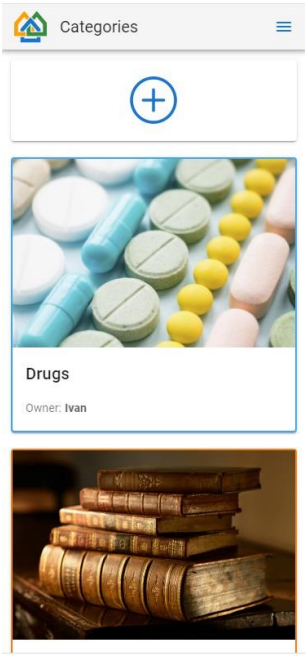

Рисунок 3.11.4.2 – Сторінка категорій (вигляд на мобільному телефоні)

3.11.5 Реалізація сторінок створення та редагування одного предмету

Структури даних сторінок дуже схожі, основна частини складає форма (ItemForm.vue). Функціонально дані сторінки схожі на сторінки створення та редагування категорій. (див. рис. 3.11.5.1, 3.11.5.2)

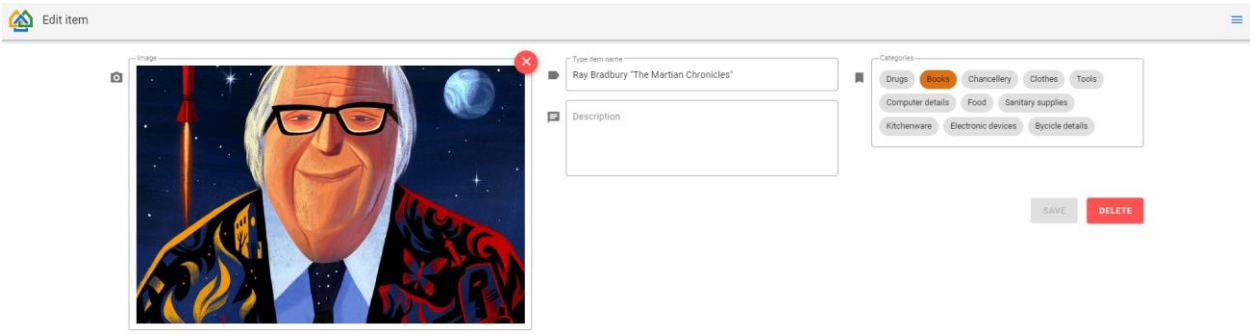

Рисунок 3.11.5.1 – Сторінка редагування предмету(вигляд на персональному комп’ютері)

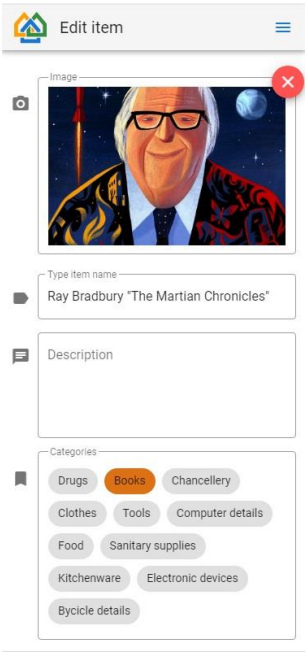

Рисунок 3.11.5.2 – Сторінка редагування предмету(вигляд на мобільному телефоні)

3.11.6 Створення сторінки предметів

Основним завданням даної сторінки являється показ всіх створених предметів, для цього був створений компонент для відображення одного предмету (ItemCard.vue). Також даний компонент в залежність від конфігурації може змінювати формат виводу даних про предмет, від більш простої до детальної форми. Так як предметів може буди багато, для даної сторінки реалізоване часткове завантаження контенту. При першому завантаженні клієнт відправляє запит до серверу з вказівкою повернути перші 30 предметів та відображає їх на сторінці. Коли користувач доскролює сторінку майже до кінця відбувається повторний запит за контентом, але тепер клієнт просить повернути наступні 30 предметів, і після їх отримання з’єднує нові дані зі старими. Запити мають наступний вигляд (див. рис. 3.11.6.1, 3.11.6.2)

Рисунок 3.11.6.1 – Запит з клієнту за першою частиною предметів

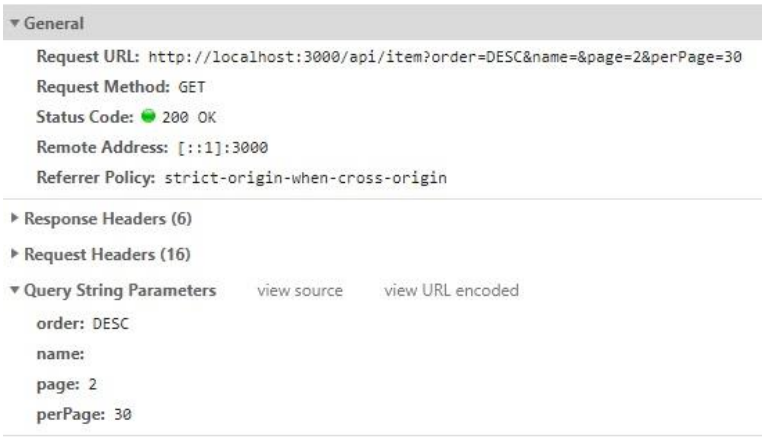

Рисунок 3.11.6.2 – Запит з клієнту за наступною частиною предметів

Параметр perPage зберігає необхідну кількість предметів які потрібно повернута, а page відповідає за відступ від початку всіх предметі, тобто показує яку саме порцію даних потрібно повернути. Візуально завантаження даних схоже з лінійний конвеєром.

Частина коду серверної частини, яка відповідає за часткову віддачу контенту (itemController.js):

```
const limit = parseInt(perPage, 10) || 30;
const offset = (parseInt(page, 10) || 1) * limit - limit;
const items = await Item.findAndCountAll({
  where,
  include: [Category, User],
  order,
  limit,
  offset,
});
```

Частина коду клієнта:

```
scrollHandler(e) {
  const { target } = e;
  const SCROLL_OFFSET = 200;
  if (this.page * this.perPage >= this.total || this.isItemsLoading)
    return;

  const leftToScroll =
    target.scrollHeight - target.scrollTop - window.innerHeight;
  if (leftToScroll > SCROLL_OFFSET) return;

  this.isItemsLoading = true;
  this.page += 1;
  const query = {
    ...this.filters,
    page: this.page,
    perPage: this.perPage,
  };
  this.getItems(query)
    .catch((err) => {
      console.error(err);
    })
    .finally(() => {
      this.isItemsLoading = false;
    });
  },
```

Для зручності пошуку, який є однією із вимог до систему, реалізовано бокове меню з можливістю знайти необхідний предмет за такими критеріями:

- пошук по назві, відбувається шляхом пошуку підрядка в рядку. Наприклад, предмет з назвою «Предмет» буде знайдений, якщо вписати «дме», також пошук не чутливий до реєстру букв.
- можливість вивести спочатку нові, або старі предмети, тобто сортувати за датою створення.
- фільтрувати дані за приналежністю до категорії.
- фільтрувати дані за користувачем який створив предмет.
- обрати дату створення або діапазон дат для фільтрації.

Також бокове меню містить налаштування для зміни візуального відображення предметів. Для зручності дозволяє користувачеві вимкнути відображення певних даних предмету. (див. рис. 3.11.6.3)

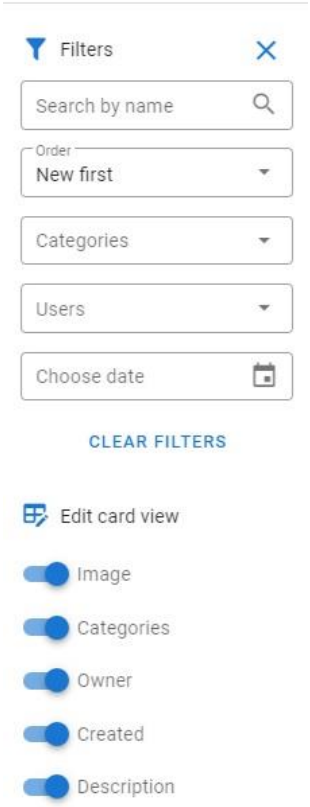

Рисунок 3.11.6.3 – Бокове меню сторінки предметів

Остаточна реалізація сторінки має наступний вигляд (див. рис. 3.11.6.4 – 3.11.6.7)

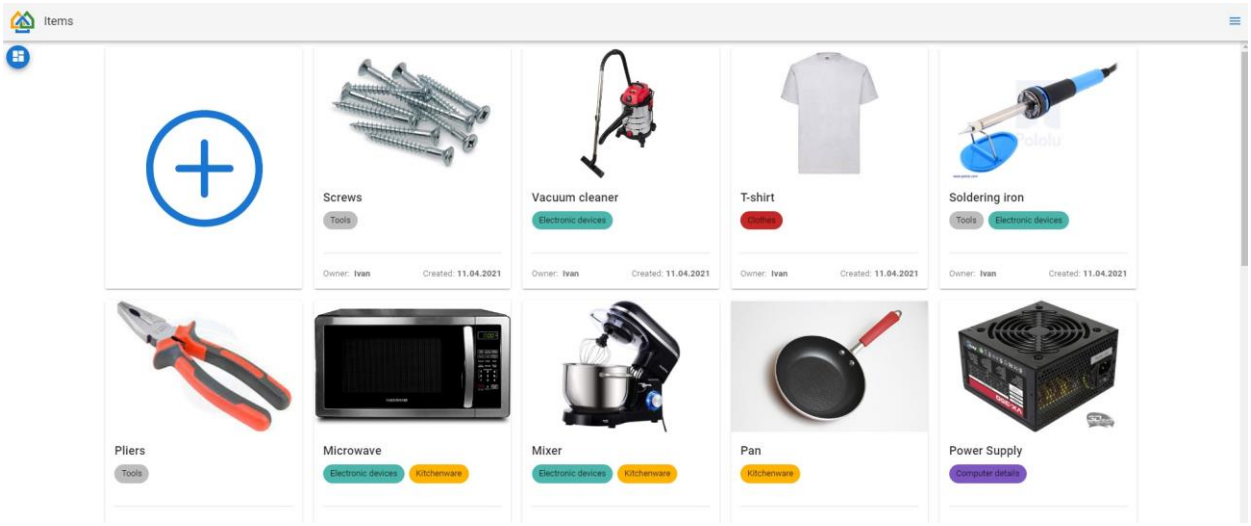

Рисунок 3.11.6.4 – Сторінка предметів (вигляд на персональному комп’ютері)

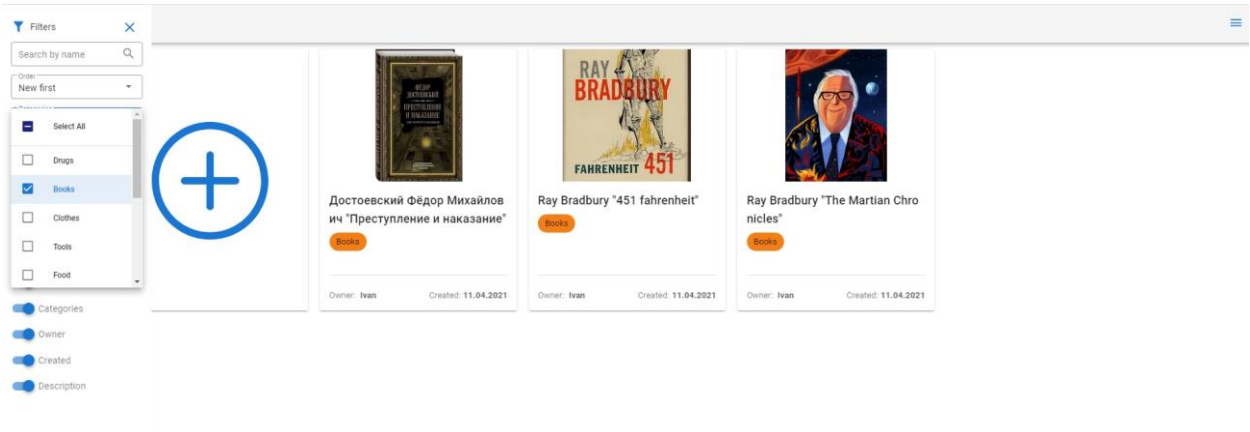

Рисунок 3.11.6.5 – Сторінка предметів, фільтрації даних за категорією «Books» (вигляд на персональному комп’ютері)

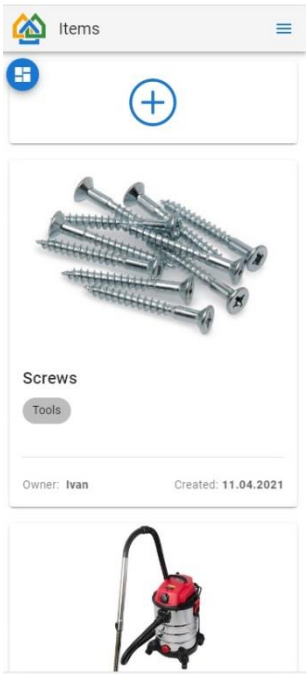

Рисунок 3.11.6.6 – Сторінка предметів (вигляд на мобільному телефоні)

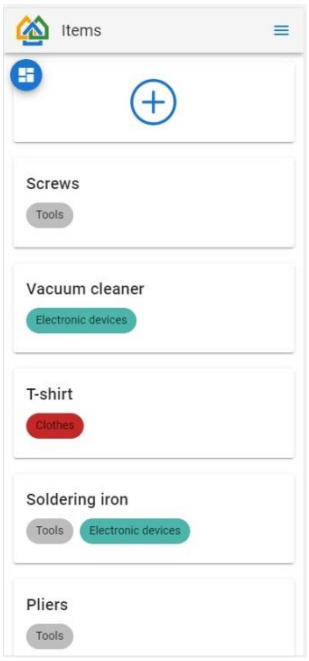

Рисунок 3.11.6.7 – Сторінка предметів, налаштовано вивід лише назви та категорії предметів (вигляд на мобільному телефоні)

3.11.7 Створення сторінки групи користувача

Користувач має змогу додати до своєї групи іншого користувача, наприклад, якщо вони проживають в одному будинку або працюють разом та мають спільний доступ до предметів. Тобто користувач в групі зможе

маніпулювати з предметами та категоріями всієї групи. Дана сторінка дозволяє додати та видалити користувача з групи. А також виводить список всієї групи. Для виводу стислої інформації про користувача був створений окремий компонент (UserCard.vue). (див. рис. 3.11.7.1, 3.11.7.2)

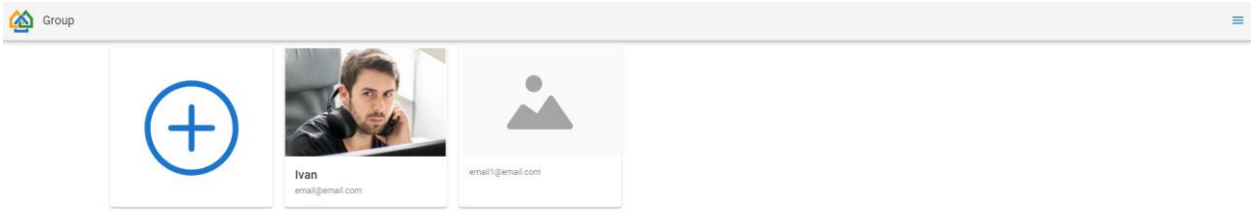

Рисунок 3.11.7.1 – Сторінка групи користувача (вигляд на персональному комп’ютері)

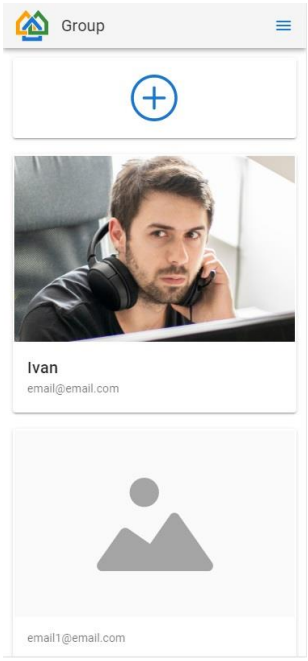

Рисунок 3.11.7.2 – Сторінка групи користувача (вигляд на мобільному телефоні)

Після запиту додати до групи нового користувача, він додається не одразу, а відправляється повідомлення яке чекає від користувача підтвердження того, що інша людина намагається додати його до своєї групи.

Це зроблено з метою захисту даних користувача, виключає випадок коли хтось сторонній приєднає його до своєї групи. (див. рис. 3.11.7.3)

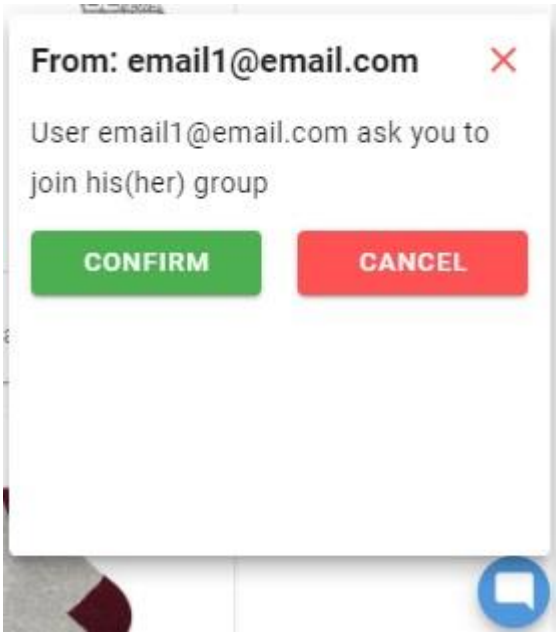

Рисунок 3.11.7.3 – Повідомлення про спробу приєднатися до групи іншого користувача

3.12 Контейнеризація додатку за допомогою Docker та розгортання на хмарному сервісі

Docker – програмне забезпечення для автоматизації розгортання і управління додатками в середовищах з підтримкою контейнеризації. Дозволяє «упакувати» додаток з усім його оточенням і залежностями в контейнер, який може бути розгорнутий на будь-який Linux-системі, а також надає набір команд для управління цими контейнерами. [42]

Використання Docker дозволить значно спростити процес установки та подальшого оновлення системи на сервері.

Для контейнеризації системи по перше необхідно створити образ (image). Image – це read-only шаблон, з якого створюється контейнер. Кожен образ складається з набору рівнів. Docker використовує union file system для поєднання цих рівнів в один образ. Union file system дозволяє файлам і директоріями з різних файлових систем (різних гілок) прозоро накладатися,

створюючи когерентну файлову систему. [43] Для створення образу необхідно створити Dockerfile, в якому задати необхідні налаштування. Тобто за допомогою Docker команд відбувається копіювання необхідних файлів проекту та встановлення залежностей.

Dockerfile клієнтської частини:

```
FROM node:lts-alpine
RUN npm install -g http-server
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
RUN npm run build
EXPOSE 8080
CMD http-server dist
```

Dockerfile серверної частини:

```
FROM node:14.15.5-alpine
WORKDIR /item-manager-node
COPY package.json ./
RUN npm install --only=production
COPY . .
EXPOSE 3000
CMD npm run start
```

Необхідно додати папку залежностей та статичних файлів до файлу .dockerignore, щоб не відбувалось їх копіювання до образу:

```
node_modules
static
```

На основі Dockerfile був створений образ клієнтської частини. Команда для створення (див. рис. 3.12.1):

```
docker build -t item-manager-vue .
```

```
Build an image from a Dockerfile
PS E:\Projects\item-manager-frontend-vue> docker build -t item-manager-vue .
[+] Building 256.9s (12/12) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 219B
 => [internal] load .dockerignore
 => => transferring context: 71B
 => [internal] load metadata for docker.io/library/node:lts-alpine
 => [1/7] FROM docker.io/library/node:lts-alpine@sha256:ed51af876dd7932ce5c1e3b16c2e83a3f58419d824e366de1f7b00f40c848c40
 => => resolve docker.io/library/node:lts-alpine@sha256:ed51af876dd7932ce5c1e3b16c2e83a3f58419d824e366de1f7b00f40c848c40
 => => sha256:c88490a07c2dde907e45eaef78acaa7cdb763a542c6e43700a4b27fc3127fd4d 6.73kB / 6.73kB
 => => sha256:ddad3d7c1e96adf9153f8921a7c9790f880a390163df453be1566e9ef0d546e0 2.82MB / 2.82MB
 => => sha256:f540588972008e57755cef67c2889591f90c1667173bfb65204bd150451d0712 35.76MB / 35.76MB
 => => sha256:b9eac74e56b40e2bc15c800307823509d94e5c3be057e549a1ed4e8d96db65c0 2.24MB / 2.24MB
 => => sha256:ed51af876dd7932ce5c1e3b16c2e83a3f58419d824e366de1f7b00f40c848c40 1.43kB / 1.43kB
 => => sha256:03b02e5b45b5e9c59dd1dea146f05a64599465a21caf81494ca8f3513cf3a090 1.16kB / 1.16kB
 => => extracting sha256:ddad3d7c1e96adf9153f8921a7c9790f880a390163df453be1566e9ef0d546e0
 => => sha256:13e85f13e8706d844dbe9b25863dbd25aa4d301395db18ced3ae27d0de5a622d 283B / 283B
 => => extracting sha256:f540588972008e57755cef67c2889591f90c1667173bfb65204bd150451d0712
 => => extracting sha256:b9eac74e56b40e2bc15c800307823509d94e5c3be057e549a1ed4e8d96db65c0
 => => extracting sha256:13e85f13e8706d844dbe9b25863dbd25aa4d301395db18ced3ae27d0de5a622d
 => [internal] load build context
 => => transferring context: 237.87kB
 => [2/7] RUN npm install -g http-server
 => [3/7] WORKDIR /app
 => [4/7] COPY package.json ./
 => [5/7] RUN npm install
 => [6/7] COPY . .
 => [7/7] RUN npm run build
 => exporting to image
 => => exporting layers
 => => writing image sha256:ab13bca71a99e8fac0810b27427c32d573baca131687ad93272e0f1bd010e6ba
 => => naming to docker.io/library/item-manager-vue
PS E:\Projects\item-manager-frontend-vue>
```

Рисунок 3.12.1 – Створення образу клієнтської частини

Для створення образу серверної частини необхідно зробити налаштування взаємодії з базою даних, а отже необхідно виконати декілька команд на відміну від клієнтської. Роботи це в консолі доволі не зручно, адже запис доволі великий, тому існує додатковий інструмент docker-compose. Docker-compose дозволяє прописати всі необхідні команди для створення образу в файлі docker-compose.yml та виконати його.

Docker-compose серверної частини:

```yaml
services:
  postgres:
    image: postgres:13.2
    ports:
      - "2345:5432"
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: 04071990
      POSTGRES_DB: ItemManager
    volumes:
      - item-manager-db
```

```
app:
  build: .
  depends_on:
    - postgres
  environment:
    NODE_ENV: production
    PORT: 3000
    PGDATABASE: ItemManager
    PGUSER: postgres
    PGPASSWORD: 04071990
    PGHOST: postgres
    PGPORT: 5432
  ports:
    - "3000:3000"
```

Для створення образу за допомогою docker-compose необхідно виконати команду:

```
docker-compose up
```

Отримано образи клієнтської і серверної частин (див. рис. 3.12.2)

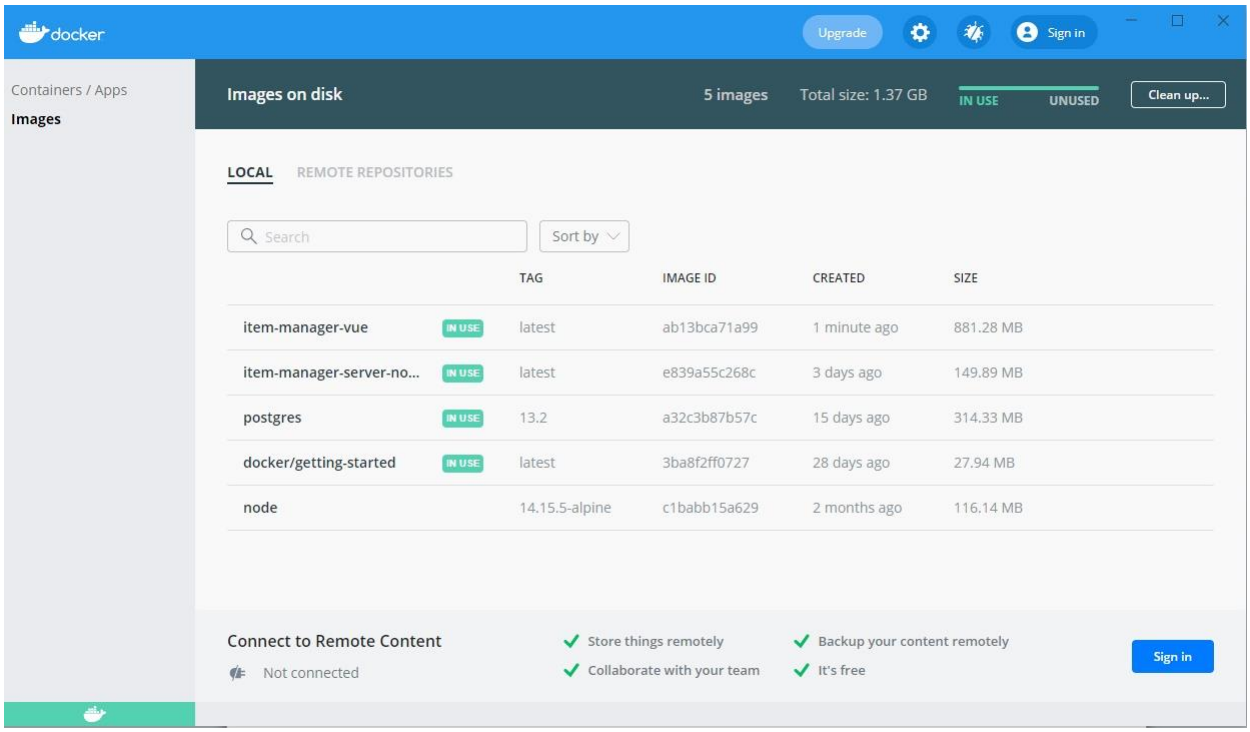

Рисунок 3.12.2 – Docker desktop, список створених образів

На основі образів необхідно створити контейнери. Коли docker запускає контейнер, він створює рівень для читання / запису зверху образу, в

якому може бути запущено додаток. Може бути створено безліч контейнерів на основі образу.

Було створено та запущено два контейнери використовуючи інтерфейс програми. (див. рис. 3.12.3)

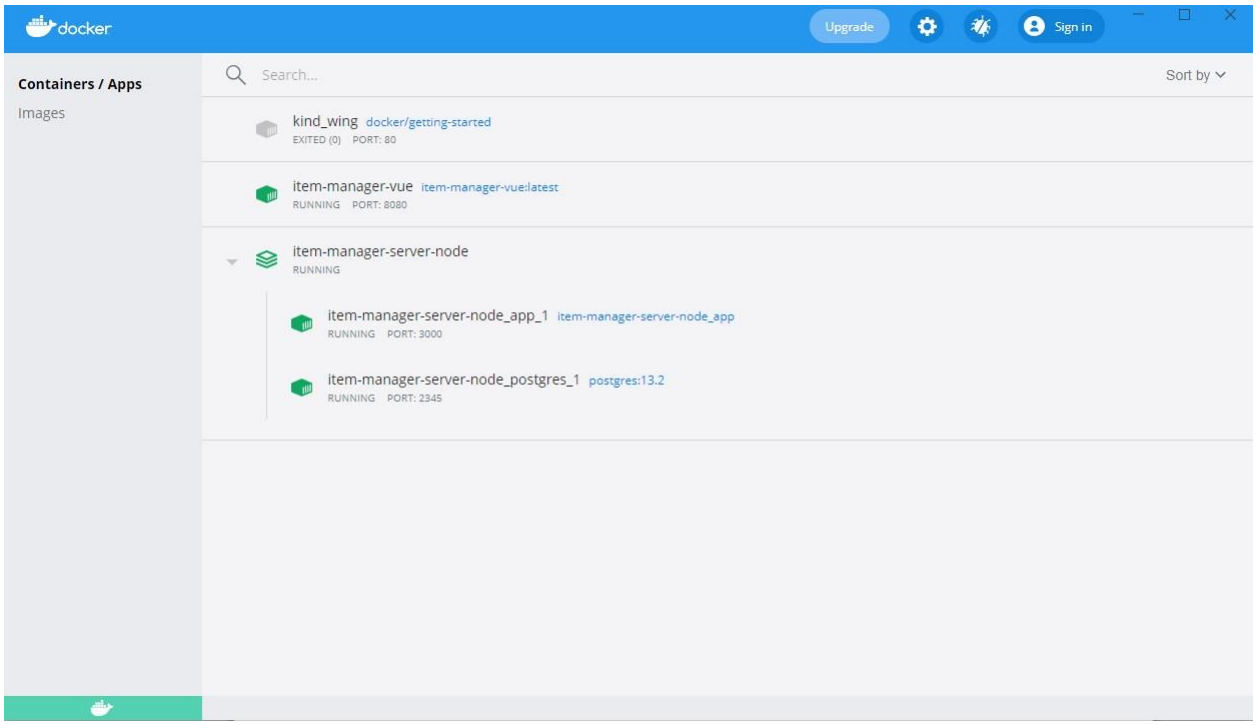

Рисунок 3.12.3 – Docker desktop, список створених контейнерів

Для розгортання системи був обраний Google Cloud Platform. Створено проект ItemManager та за допомогою Google Compute Engine було сконфігуровано віртуальну машину (instance-1) на базі OS Ubuntu (див. рис. 3.12.4).

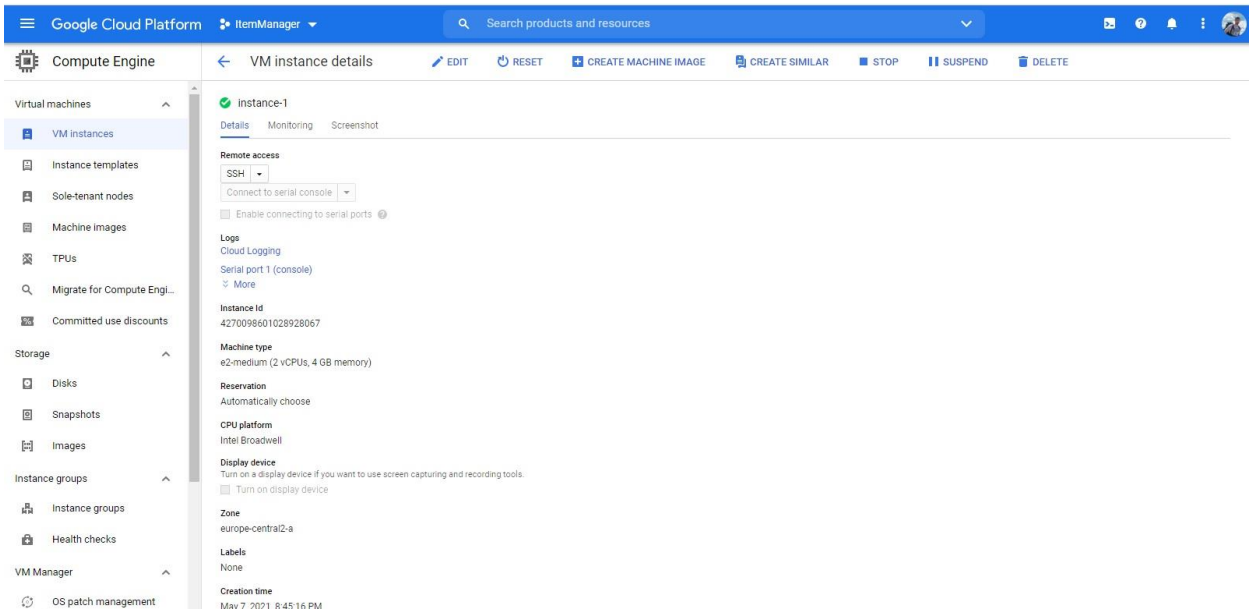

Рисунок 3.12.4 – Віртуальна машина створена на Google Cloud Platform

На віртуальній машині були розгорнуті та запущені Docker контейнери (див. рис. 3.12.5)

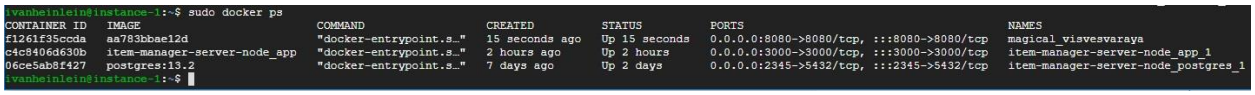

Рисунок 3.12.5 – Список працюючих Docker контейнерів на віртуальній машині

Наступний кроком був встановлений Nginx. Nginx – це HTTP-сервер і зворотний проксі-сервер, поштовий проксі-сервер, а також TCP / UDP проксі-сервер загального призначення. [44]

Перед тестуванням Nginx необхідно виконати настройку брандмауера. Для цього виконавши команду `sudo ufw allow 'Nginx Full'`, був дозволений трафік на портах 80 і 443 (див. рис. 3.12.6).

```
ivanheinlein@instance-1:~$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere
Nginx Full                 ALLOW       Anywhere
OpenSSH (v6)               ALLOW       Anywhere (v6)
Nginx Full (v6)            ALLOW       Anywhere (v6)
```

Рисунок 3.12.6 – Список дозволеного трафіка брандмауера

Потім необхідно було конфігурувати сервер наступним чином (див. рис. 3.12.7)

```
server_name _;


location / {
        try_files $uri $uri/ /index.html;
}

location /images {
        proxy_pass http://localhost:3000/images;
}

location /api {
        proxy_pass http://localhost:3000;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-NginX-Proxy true;

        proxy_http_version 1.1;
        proxy_cache_bypass $http_upgrade;
}
```

Рисунок 3.12.7 – Файл etc/nginx/sites-available/default, конфігурація серверу для коректної роботи додатку

Сервер був перезавантажений, щоб нова конфігурація набула сили. При переході на зовнішній IP адрес віртуальної машини (http://34.118.64.226/) бачимо, що все працею вірно (див. рис. 3.12.8).

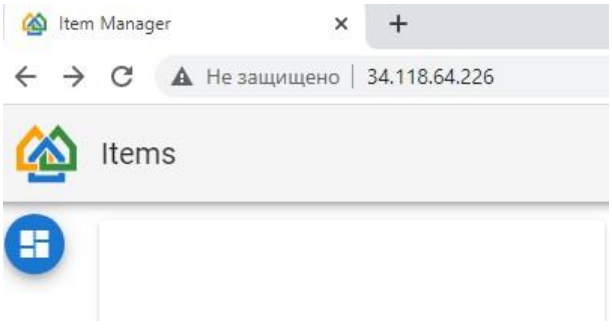

Рисунок 3.12.8 – Додаток запущений на віртуальній машині

ВИСНОВОК

У даній роботі була розглянута проблема інвентаризації речей різного призначення. Для вирішення проблеми була сформована і створена система, яка дозволяє зберігати дані про речі та зручно ними маніпулювати. Щоб побудувати необхідну системи була використана waterfall модель життєвого циклу програмного забезпечення, що дозволило розділити розробку на окремі самодостатні етапи.

По перше, на основі аналізу були сформовані вимоги, які наслідують переваги сучасних рішень, та компенсують їх недоліки.

На етапі проектування, були сформовані головні бізнес процеси додатку. Також обрані технології та інструменти для реалізації додатку. Та сформовані кроки для програмної реалізації.

На етапі розробки реалізовано три незалежних компонента:

- Клієнтська частина, яка являє собою SPA додаток. Та створена з використання бібліотеки Vue та Vuetify.
- Серверна частина. Створена з використання Node.js та фреймворку Express.
- База даних – PostgreSQL. Було спроектовано структуру БД реалізувавши DFD-0,1 і ERD діаграмми.

Розділення системи на незалежні компоненти дозволить, за необхідності, зручно збільшувати функціональність. Такий підхід надає можливість клієнтській частині використовувати додатково API іншого серверу, а також теперішний API може бути використаним інший клієнтом.

Після етапу розробки, було проведено мануальне тестування додатку, та зроблений підсумок, що система повністю відповідає сформованим вимогам.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Wild T. Best practice in inventory management / T. Wild. – 2018. – 276 p.
2. Носитель информации — Википедия [Electronic resource]. – Electronic data. – Access mode : https://ru.wikipedia.org/wiki/Носитель_информации. – 17.04.2021.
3. Бумага — Википедия [Electronic resource]. – Electronic data. – Access mode : https://ru.wikipedia.org/wiki/Бумага. – 17.04.2021.
4. Бесплатные шаблоны управления инвентарём в Excel [Electronic resource]. – Electronic data. – Access mode : https://ru.smartsheet.com/free-excel-inventory-templates. – 18.04.2021.
5. 12 приложений для домашней инвентаризации [Electronic resource]. – Electronic data. – Access mode : https://blog.themarfa.name/12-prilozhienii-dlia-domashniei-invientarizatsii/. – 13.04.2021.
6. Home Inventory | Binary Formations, LLC [Electronic resource]. – Electronic data. – Access mode : https://binaryformations.com/products/home-inventory/. – 13.04.2021.
7. StuffKeeper: Catalog of personal things. Home inventory app. [Electronic resource]. – Electronic data. – Access mode : https://www.stuffkeeper.app/. – 14.04.2021.
8. New Message! [Electronic resource]. – Electronic data. – Access mode : https://www.sortly.com/. – 14.04.2021.
9. Каскадная модель — Википедия [Electronic resource]. – Electronic data. – Access mode : https://ru.wikipedia.org/wiki/Каскадная_модель. – 11.04.2021.
10. Багатоплатформність — Вікіпедія [Electronic resource]. – Electronic data. – Access mode : https://uk.wikipedia.org/wiki/Багатоплатформність. – 11.04.2021.
11. Кроссплатформенная разработка мобильных приложений в 2020 году / Хабр [Electronic resource]. – Electronic data. – Access

mode : https://habr.com/ru/post/491926/. – 11.04.2021.

12. Digital 2020: Global Digital Overview — DataReportal – Global Digital Insights [Electronic resource]. – Electronic data. – Access mode : https://datareportal.com/reports/digital-2020-global-digital-overview. – 11.04.2021.
13. REST — Национальная библиотека им. Н. Э. Баумана [Electronic resource]. – Electronic data. – Access mode : https://ru.bmstu.wiki/REST. – 11.04.2021.
14. Single-page application - Wikipedia [Electronic resource]. – Electronic data. – Access mode : https://en.wikipedia.org/wiki/Single-page_application. – 11.04.2021.
15. Введение — Vue.js [Electronic resource]. – Electronic data. – Access mode : https://ru.vuejs.org/v2/guide/index.html. – 11.04.2021.
16. Vuetify — A Material Design Framework for Vue.js [Electronic resource]. – Electronic data. – Access mode : https://vuetifyjs.com/en/. – 11.04.2021.
17. Node.js — Википедия [Electronic resource]. – Electronic data. – Access mode : https://ru.wikipedia.org/wiki/Node.js. – 11.04.2021.
18. Express.js - Wikipedia [Electronic resource]. – Electronic data. – Access mode : https://en.wikipedia.org/wiki/Express.js. – 11.04.2021.
19. PostgreSQL: The world’s most advanced open source database [Electronic resource]. – Electronic data. – Access mode : https://www.postgresql.org/. – 11.04.2021.
20. Обзор — Bitbucket [Electronic resource]. – Electronic data. – Access mode : https://bitbucket.org/dashboard/overview. – 11.04.2021.
21. Bitbucket — Википедия [Electronic resource]. – Electronic data. – Access mode : https://ru.wikipedia.org/wiki/Bitbucket. – 11.04.2021.
22. Как следует писать комментарии к коммитам / Хабр [Electronic resource]. – Electronic data. – Access mode : https://habr.com/ru/post/416887/. – 11.04.2021.

23. Git - Branching Workflows [Electronic resource]. – Electronic data. – Access mode : https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows. – 11.04.2021.
24. Урок по диаграммам DFD | Lucidchart [Electronic resource]. – Electronic data. – Access mode : https://www.lucidchart.com/pages/ru/диаграмма-dfd. – 11.04.2021.
25. Object–relational mapping - Wikipedia [Electronic resource]. – Electronic data. – Access mode : https://en.wikipedia.org/wiki/Object–relational_mapping. – 11.04.2021.
26. Sequelize ORM [Electronic resource]. – Electronic data. – Access mode : https://sequelize.org/. – 11.04.2021.
27. Node.JS | Паттерн MVC. Контроллеры [Electronic resource]. – Electronic data. – Access mode : https://metanit.com/web/nodejs/7.1.php. – 11.04.2021.
28. Manual | Sequelize [Electronic resource]. – Electronic data. – Access mode : https://sequelize.org/master/manual/model-basics.html. – 11.04.2021.
29. Управление паролями. Разбираемся, как правильно шифровать… | by Bohdan Balov | Medium [Electronic resource]. – Electronic data. – Access mode : https://medium.com/@balovbohdan/управление-паролями-82d99005207. – 11.04.2021.
30. kelektiv/node.bcrypt.js: bcrypt for NodeJs [Electronic resource]. – Electronic data. – Access mode : https://github.com/kelektiv/node.bcrypt.js#readme. – 11.04.2021.
31. Пять простых шагов для понимания JSON Web Tokens (JWT) / Хабр [Electronic resource]. – Electronic data. – Access mode : https://habr.com/ru/post/340146/. – 11.04.2021.
32. auth0/node-jsonwebtoken: JsonWebToken implementation for node.js http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html [Electronic resource]. – Electronic data. – Access mode : https://github.com/auth0/node-jsonwebtoken#readme. – 11.04.2021.

33. expressjs/multer: Node.js middleware for handling `multipart/form-data`. [Electronic resource]. – Electronic data. – Access mode : https://github.com/expressjs/multer#readme. – 11.04.2021.
34. uuidjs/uuid: Generate RFC-compliant UUIDs in JavaScript [Electronic resource]. – Electronic data. – Access mode : https://github.com/uuidjs/uuid#readme. – 11.04.2021.
35. axios/axios: Promise based HTTP client for the browser and node.js [Electronic resource]. – Electronic data. – Access mode : https://github.com/axios/axios. – 11.04.2021.
36. Что такое Vuex? | Vuex [Electronic resource]. – Electronic data. – Access mode : https://vuex.vuejs.org/ru/. – 11.04.2021.
37. Vue Router [Electronic resource]. – Electronic data. – Access mode : https://router.vuejs.org/ru/. – 11.04.2021.
38. Favicon — Википедия [Electronic resource]. – Electronic data. – Access mode : https://ru.wikipedia.org/wiki/Favicon. – 11.04.2021.
39. Основы компонентов — Vue.js [Electronic resource]. – Electronic data. – Access mode : https://ru.vuejs.org/v2/guide/components.html. – 11.04.2021.
40. Однофайловые компоненты — Vue.js [Electronic resource]. – Electronic data. – Access mode : https://ru.vuejs.org/v2/guide/single-file-components.html. – 11.04.2021.
41. v-input API — Vuetify [Electronic resource]. – Electronic data. – Access mode : https://vuetifyjs.com/en/api/v-input/#props. – 11.04.2021.
42. Docker — Википедия [Electronic resource]. – Electronic data. – Access mode : https://ru.wikipedia.org/wiki/Docker. – 15.04.2021.
43. Понимая Docker / Хабр [Electronic resource]. – Electronic data. – Access mode : https://habr.com/ru/post/253877/. – 15.04.2021.
44. nginx [Electronic resource]. – Electronic data. – Access mode : https://nginx.org/ru/. – 15.05.2021.