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

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

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка веб-сервісу «Файловий менеджер»

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

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

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

Нормоконтроль _____ Шимчук Г.В.
(підпис) (прізвище та ініціали)

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

Рецензент _____ Бойко І.В
(підпис) (прізвище та ініціали)

Тернопіль
2021

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

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«__» _____ 2021 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

Студенту Мачужаку Андрію Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка веб-сервісу «Файловий менеджер»

Керівник роботи Готович Володимир Анатолійович, к.т.н., доц. кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 02 » березня 2021 року № 4/7-171

2. Термін подання студентом завершеної роботи 22.06.2021р.

3. Вихідні дані до роботи технічна документація, інтернет-джерела.

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1 Аналіз предметної області та вибір технологій для розробки веб-сервісу «Файловий менедежер». 1.1 Дослідження веб-технологій. 1.1.1 Історія. 1.1.2 Веб-стандарти. 1.1.3 Актуальні веб-технології. 1.2 Порівняння готових веб-сервісів та формування технічного завдання.1.3 Вибір технологій для клієнта і сервера. 1.3.1 Вибір серверної технології. 1.3.2 MySQL в якості СКБД. 1.3.3 Вибір клієнтської технології. 1.3.4 Контейнеризація в Docker. . 1.4 Висновок до першого розділу 2 Проектування веб- сервісу «Файловий менеджер».2.1 Налаштування проекту. 2.1.1 Конфігурація Dockerfile та docker-compose.yml.2.1.2 Ініціалізація проекту, завантаження пакетів та бібліотек. 2.2 Розроблення додатка. 2.2.1 Розробка UI. 2.2.2 Схема та міграції БД. 2.2.3 Розробка API. Автентифікація та Авторизація. 2.3 Тестування «Файлового менеджера». 2.4 Висновок до другого розділу. 3 Безпека життєдіяльності, основи охорони праці. 3.1 3.1 Роль центральної нервової системи в трудовій діяльності людини. 3.2 Вплив шуму на організм людини та розробка заходів щодо його зниженню до допустимих величин для обладнання. 3.3 Висновок до третього розділу. Висновки. Перелік джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Титульна сторінка. 2.Актуальність та мета кваліфікаційної роботи. 3. Практичне значення Одержаних результатів. 4. Структура роботи. 5. Огляд готових рішень. 6. Клієнт-серверна архітектура. 7. Клієнт. 8. Базовий компонент. 9. Застосування компонента. 10. Сервер – Node.js 11. API. 12. Авторизація. 13. Автентифікація. 14. JWT-токен. 15. MySQL – як СКБД. 16. Схема БД. 17. Docker & Docker Compose. 18. Docker Compose. 19. Демонстрація. 20. Демонстрація. 21. Створення та додавання об'єктів. 22. Поширення. 23. Висновок.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи хорони праці	Гурик О.Я., доцент кафедри МТ		

7. Дата видачі завдання 25 січня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	26.01.2021	*Виконано*
2.	Підбір джерел про актуальні технології у сфері веб-розробки, та клієнт-серверну архітектуру	08.02.2021-09.02.2021	*Виконано*
3.	Переклад та опрацювання джерел про сучасні технології для розробки веб-застосунків, та клієнт-серверної	10.02.2021-11.03.2021	*Виконано*
	архітектури		
4.	Виконання дослідження щодо технологій для серверної	12.03.2021-16.04.2021	*Виконано*
	частини веб-сервісу «Файловий менеджер»		
5.	Розроблення веб-сервісу «Файловий менеджер»	17.04.2021-15.06.2021	*Виконано*
6.	Оформлення розділу 1 «Аналіз предметної	16.06.2021-17.06.2021	*Виконано*
	області та вибір технологій для розробки		
	веб-сервісу «файловий менедежер»»		
7.	Оформлення розділу 2 Проектування веб-сервісу	17.06.2021	*Виконано*
	«Файловий менеджер»		
8.	Виконання завдання до підрозділу «Безпека	17.06.2021	*Виконано*
	життєдіяльності»		
9.	Виконання завдання до підрозділу «Основи охорони	18.06.2021	*Виконано*
	праці»		
10.	Оформлення кваліфікаційної роботи	19.06.2021	*Виконано*
11.	Нормоконтроль	19.06.2021	*Виконано*
12.	Перевірка на плагіат	19.06.2021	*Виконано*
13.	Попередній захист кваліфікаційної роботи	19.06.2021	*Виконано*
14.	Захист кваліфікаційної роботи	22.06.2021	

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

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

АНОТАЦІЯ

Розробка веб-сервісу «Файловий менеджер» // Кваліфікаційна робота освітнього рівня «Бакалавр» // Мачужак Андрій Володимирович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СН-41 // Тернопіль, 2021 // С.75 , рис. – 11, табл. – 4, кресл. – 0, додат. – 1, бібліогр. – 28.

Ключові слова: веб-розробка, веб-технології, веб-сервіс, база даних, бд, файловий менеджер, веб-інтерфейс, клієнт-сервер.

Кваліфікаційна робота присвячена розробці веб-сервісу «Файловий менеджер» з використанням сучасних засобів веб-розробки.

Мета роботи полягає у вивченні сучасних технологій у сфері веб-розробки та здобуття досвіду для майбутнього працевлаштування на одну з наступних позицій: «Front-end розробник», «Back-end розробник» або ж «Full-stack розробник»

В першому розділі кваліфікаційної роботи розглянуто теоретичні відомості у сфері веб-розробки та проведено аналіз існуючих рішень для формування технічного завдання та вибору технологій для розробки.

В другому розділі кваліфікаційної роботи описано розроблення веб-сервісу «Файловий менеджер» з вибраними технологіями та проведено тестування застосунку на виконання поставлених цілей.

ANNOTATION

Web-service «File Manager» development // Qualification work of the educational level «Bachelor» // Machuzhak Andriy Volodymyrovych // Ternopil National Technical University named after Ivan Pulyuy, Faculty of Computer Information Systems and Software Engineering // Ternopil, 2021 // Explanatory note size – 75 pages, contains 11 illustrations, 4 tables, 1 applications, 28 bibliography items.

Keywords: web development, web technologies, web service, database, DB, file manager, web interface, client-server

This work is devoted to the development of the «File Manager» web service using modern web development tools.

The purpose of the work is to study modern technologies in the field of web development and gain experience for future employment in one of the following positions: «Front-end developer», «Back-end developer» or «Full-stack developer»

The first section of the qualification work considers theoretical information in the field of web development and analyzes existing solutions for the purpose of formation technical specifications for this work and to choose the technologies for development.

The second section of the qualification work describes the development of the «File Manager» web service with the selected technologies and then it was tested if the application fulfilled the goals.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних

БЖД – безпека життєдіяльності

Клієнт (front-end) – все що можна відобразити в браузері – веб-додатки (веб-сайти), тощо

Сервер (back-end) – все, що працює на сервері, тобто «не в браузері» або на комп'ютері, який реагує на повідомлення з інших комп'ютерів.

СКБД – Система керування базами даних

ОП – охорона праці

ТЗ –технічне завдання

SPA – single-page application

UI – user interface, інтерфейс користувача

ЗМІСТ

ВСТУП

Актуальність теми. Всесвітня мережа Інтернет дала можливість та середовище для розвитку нових способів взаємодії між людьми та зберігання різної інформації. Саме зберігання та безпечне передавання інформації у будь-якому її вигляді, є важливою темою для дослідження. Адже користувачі очікують безвідмовну та надійну роботу сервісів, якими вони користуються, та готові платити за це.

Темою цієї роботи було обрано розробку веб-сервісу «Файловий менеджер». На ринку є багато готових рішень для зберігання інформації – від Amazon Web Services до Google Cloud [1]. Розроблений в цій пояснювальній записці веб-сервіс «Файловий менеджер» не претендує на роль конкурента вище згаданим сервісам, но при його розробці буде досягнуто поставлені цілі, які наведено дальше.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи освітнього рівня «Бакалавр» є:

– Проаналізувати стан досліджень в сфері веб-розробки;

– Обрати найактуальніші технології для проектування веб-сервісу «Файловий менеджер»;

– Дослідити можливі архітектури та вибрати найвідповіднішу для майбутнього веб-сервісу та розробити «Файловий менеджер» використовуючи обрані технології;

– Здобути необхідний для працевлаштування досвід на Junior позицію Front end/Back end або Full Stack веб-розробника.

Практичне значення одержаних результатів. В кінцевому результаті буде отримано готовий веб-сервіс «Файловий менеджер» з можливістю зберігати інформацію в будь-якому файловому форматі. У користувача буде можливість завантажити файли на сервер та з сервера, також можна буде поширити файли публічно або ж тільки за email зареєстрованого користувача.

В підсумку студент отримає важливий практичний досвід у сфері веб-розробки з використанням сучасних практик та технологій

Ця дипломна робота складається зі вступу, двох розділів про дослідження та розроблення веб-сервісу «Файловий менеджер», одного розділу з темами «Безпеки життєдіяльності»(БЖД) та «Охорони праці» (ОП) та висновку. Відповідно у першому розділі розповідається про аналіз предметної області та вивчення необхідних технологій, та вибору архітектури для розробки «Файлового менеджера». Другий розділ відведено на опис розроблення веб-сервісу «Файловий менеджер» з використанням вибраних технологій. Третій розділ описує теми БЖД та ОП з прив’язкою до теми кваліфікаційної роботи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ВЕБ-СЕРВІСУ «ФАЙЛОВИЙ МЕНЕДЕЖЕР»

У цьому розділі відбудеться аналіз сфери веб-розробки для визначення актуальних технологій, буде проведено порівняння готових веб-сервісів для зберігання інформації, та на їхній основі буде побудовано список мінімальних вимог для реалізації «Файлового менеджера». Також буде обрано технологічний стек для розроблення додатку.

1.1 Дослідження веб-технологій

Впродовж всього розвитку інтернету розроблялися специфікації, стандарти, також і засоби для роботи з ним, технології для розроблення так званих веб-сторінок – гіпертекстових документів [2], які взаємопов’язані між собою і вміщують посилання на інші сторінки. Ще окрім сторінок створювалися веб-сервери, які надають ці документи за запитом.

1.1.1 Історія

Інтернет почав свою історію в кінці 60-х років в США для військових цілей під назвою ARPANET [3]. У 1980 Тімом Бернерсом-Лі (далі TimBL) було описано перші концепції зв’язків між вузлами. Вже у 1989 TimBL написав публікацію – «Information Management: A Proposal»[4] яка стали опорою того як буде працювати всесвітнє павутиння. До кінця 1990 року TimBL підготував все необхідне для запуску першої версії Інтернету - HTTP, HTML, перший веб-браузер, який називався WorldWideWeb, HTTP сервер і деякі веб-сторінки для перегляду. З появою цих умов всесвітня мережа інтернет стрімко розвивалася: було випущено багато браузерів, веб-серверів та сторінок.

Також у 1994 TimBL заснував Консорціум Всесвітнього павутиння (W3C) [5], організація, яка об'єднує представників багатьох різних технологічних компаній для спільної роботи над створенням специфікацій веб-технологій. Після цього створювалися інші технології, такі як CSS і JavaScript, й Інтернет почав бути більш схожим до його сучасного варіанту.

1.1.2 Веб-стандарти

Веб-стандарти - це технології, які ми використовуємо для створення веб-сайтів. Ці стандарти існують в технічних документах, які називаються специфікаціями, в яких детально описано, як саме повинна працювати технологія. Ці документи не дуже корисні для навчання використанню описаних ними технологій, але замість цього призначені для використання інженерами-програмістами для впровадження цих технологій

Наприклад, HTML Living Standard [6] описує, як саме реалізувати HTML (всі елементи HTML, а також пов'язані з ними API та інші суміжні технології).

W3C є найбільш відомою організацією для веб-стандартів, але є й інші, такі як WHATWG (підтримують HTML), ECMA (стандарт для ECMAScript, на якому базується JavaScript), Khronos (технології для 3D графіки, такі як WebGL) та інші.

«Відкриті» стандарти. Одним з ключових аспектів веб-стандартів полягає в тому що Інтернет (і веб-технології) повинні бути вільними як для внесків в розвиток, так і для використання, а не обтяжені патентами чи ліцензуванням. Тому кожен може написати код, щоб створити веб-сайт безкоштовно, а також може зробити внесок у процес створення стандартів

«Don’t break the web» [7]. Дуже поширена фраза серед відкритих веб-стандартів, - «Don’t break the web» - ідея полягає в тому, що будь-яка нова веб-технологія, яка вводиться, повинна бути зворотно сумісною з тим, що було до цього (тобто старі веб-сайти все ще будуть працювати), і вперед сумісні

(майбутні технології, в свою чергу, будуть сумісні з тим, що ми маємо в даний час).

1.1.3 Актуальні веб-технології

Клієнти та сервери. Комп'ютери, підключені до Інтернету, називаються **клієнтами** та **серверами** [8]. Спрощена схема взаємодії може виглядати так, як зображено на рисунку 1.1:

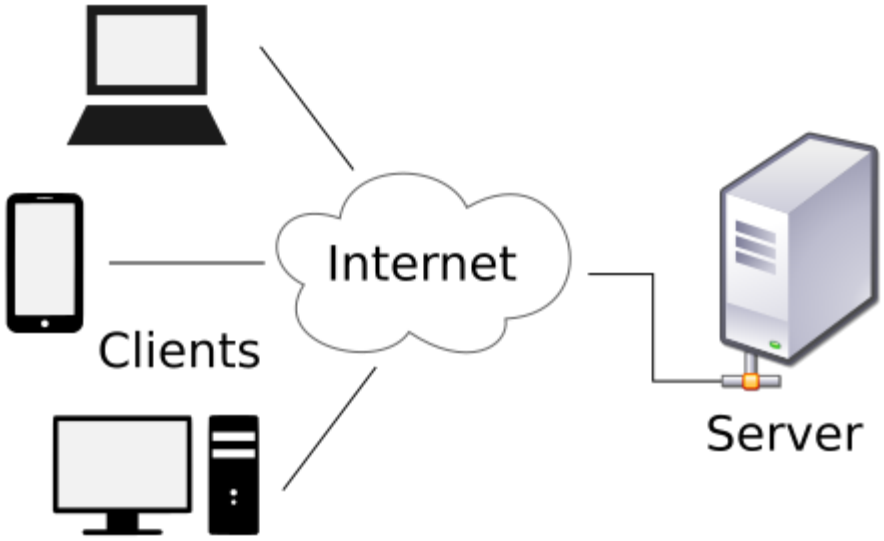

Рисунок 1.1 – Клієнт-серверна архітектура

– Клієнти – це типові пристрої, підключені до Інтернету і програмне забезпечення для доступу до Інтернету, доступне на цих пристроях;
– Сервери – це комп'ютери, на яких зберігаються веб-сторінки, сайти або застосунки. Якщо клієнтський пристрій хоче отримати доступ до веб-сторінки, копія веб-сторінки завантажується із сервера на клієнтський комп'ютер для відображення у браузері користувача.

Браузери. Веб-браузери - це програмне забезпечення, яке люди використовують для відвідування Інтернету, найвідоміші з них – Firefox, Chrome, Opera, Safari та Edge. [9]

HTTP. Hypertext Transfer Protocol, або HTTP – це протокол обміну повідомленнями, який дає змогу браузерам обмінюватися даними з веб-серверами (де зберігаються веб-сайти)[10]. В загальному процес відбувається наступним чином: відбувається запит на конкретний ресурс, а сервер відповідає файлами, які завантажує клієнт, а потім рендерить сторінку.

HTML, CSS, and JavaScript. HTML, CSS і JavaScript - це основні три технології, які використовуються для створення веб-сайту:

Мова розмітки гіпертексту, або HTML, є мовою розмітки, що складається з різних елементів, якими можна маркувати вміст, щоб надати йому значення та структуру. Простий HTML, в лістингу 1.1, виглядає наступним чином:

Лістинг 1.1 – HTML розмітка

```
<h1> Це заголовок</h1>
<p>Це абзац!</p>
<img src="cat.jpg" alt="Зображення мого кота" />
```

Для аналогії можна привести будинок, в ньому HTML був би схожий на фундаменти і стіни, які надають йому конструкцію і тримають її разом.

Каскадні таблиці стилів (CSS) — це мова, заснована на правилах, яка використовується для застосування стилів до HTML, наприклад, встановлення кольорів тексту та фону, додавання меж, анімації або розміщення сторінки певним чином. Як простий приклад, в лістингу 1.2, наступний код перетворить наш HTML абзац на червоний:

Лістинг 1.2 – CSS приклад

```
p {
     color: red;
}
```

За аналогією будинку, CSS схожий на фарбу, шпалери, килими та картини, які б використовувалися, щоб зробити будинок гарним.

JavaScript – це мова програмування, яку ми використовуємо для додавання інтерактивності до веб-сайтів, від динамічного перемикання стилів до отримання оновлень із сервера, і навіть до складної 3D графіки. В наступному прикладі, в лістингу 1.3, JavaScript буде зберігати посилання на наш абзац в пам'яті і змінювати текст всередині нього:

Лістинг 1.3 – JS код

```
let par = document.querySelector('p');
par.textContent ='Новий текст';
```

За аналогією з будинком JavaScript схожий на холодильник, мікрохвильову піч або фен — речі, які надають будинку корисну функціональність

Серверні мови та фреймворки. HTML, CSS та JavaScript є front-end (або клієнтськими) мовами, що означає, що вони виконуються браузером для створення веб-сайту, який можуть використовувати його відвідувачі.

Існує інший клас мов, які називаються мовами back-end (або на стороні сервера) [11], це означає, що вони запускаються на сервері до того, як результат буде надіслано до браузера для відображення. Типовим використанням мови на стороні сервера є отримання відомостей з бази даних і генерування HTML з цією інформацією, перш ніж відправити HTML у браузер.

Приклади серверних мов включають ASP.NET, Python, PHP та NodeJS.

1.2 Порівняння готових веб-сервісів та формування технічного завдання

Для формування вимог щодо розробки веб-сервісу «Файловий менеджер» необхідно проаналізувати вже реалізовані сервіси та на їхній

основі підібрати мінімальні рекомендації щодо реалізації додатка. При наявності вільних ресурсів можна буде розширити список вимог, та додати більше можливостей до розроблюваного «Файлового менеджера».

Для порівняння було обрано наступні найпопулярніші веб-сервіси хмарного зберігання інформації [12]:

– Dropbox;
– Google Drive;
– Onedrive.

Далі буде наведено порівняльну характеристику обраних сервісів (Див. таблицю 1.1) за такими критеріями:

1. Особливості продукту;
2. Ціни;
3. Синхронізація;
4. Поширення файлів;
5. Підтримка мобільних;
6. Безпека;
7. Приватність.

Таблиця 1.1 – Порівняння хмарних сховищ

Критерій	**DropBox**	**Google Drive**	**OneDrive**
Особливості продукту	Власні сервіси	Інтеграція з Google	Інтеграція з Microsoft
Безкоштовний простір	2Gb	15Gb	5Gb
Синхронізація	+		
Поширення файлів	По лінку або за Ідентифікатором(email, тощо)		
Підтримка мобільних	+		
Безпека	AES 256-bit, SSL/TLS		
Приватність	Сумнівна		

Всі ці постачальники хмарних технологій є розробкою великих компаній, та функціонал, який закладений в ці сервіси є дуже різноманітним та продуманим. В межах цієї дипломної роботи реалізувати всі функції є неможливим, тому в якості завдання буде обрано лише найістотніший функціонал, який забезпечить виконання основної мети файлових менеджерів в мережі Інтернет – зберігати та завантажувати файли. Також з порівняльної характеристики було обрано декілька опцій для розроблення:

- Буде реалізовано поширення файлів за лінком та за email;
- Базовий простір обмежений до 2Gb. (в майбутньому можна буде зробити плани);
- При розміщенні «Файлового менеджера» на хостингу буде можливість забезпечити безпечне шифрування даних при передачі даних.

Іншими вимогами будуть: автентифікація та авторизація користувачів, а також зверстати веб-інтерфейс на основі готового дизайну.

1.3 Вибір технологій для клієнта і сервера

Так як для повноцінного функціонування веб-сервісу необхідно як і відображати сторінки в браузері так і зберігати дані на стороні сервера то потрібно обрати технології для обох сфер. З клієнтом ситуація більш-менш зрозуміла, HTML, CSS, JS – це стандартний стек для розробки, варто лише подумати про фреймворк для цього. То з сервером можуть виникнути деякі труднощі, тому що тут вибір набагато більший за браузерну розробку сайтів.

1.3.1 Вибір серверної технології

З метою порівняння, було вирішено розглянути три бекенд-технології, які стрімко зростають у популярності в останні роки: Node.js, Ruby on Rails та

Django [13]. Кожна з цих технологій базується на різних мовах програмування, і кожна з них має певні переваги та недоліки, які потрібно мати на увазі при прийнятті рішення щодо розробки додатку.

Node.js. Node.js — популярне середовище виконання JavaScript з відкритим кодом. Вперше платформа була випущена в 2009 році, і з тих пір вибухнула популярністю [14]. Деякі з найбільш примітних користувачів Node.js: IBM, Microsoft, Netflix, PayPal, Walmart і GoDaddy.

На даний момент JavaScript є найпопулярнішою мовою програмування на GitHub, найбільшому в світі сервісі хостингу програмного забезпечення, і більшість веб-розробників вже знають це. Таким чином, підприємствам слід досить легко найняти кваліфікованих .js розробників.

Оскільки .js на використовує рушій V8 JavaScript від Google Chrome, він надає дуже високу продуктивність і майже необмежену масштабованість.

Одним з важливих недоліків Node.js є мала попередня сумісність версій. Його інтерфейс прикладних програм (API) змінюється дуже часто, і з моменту випуску першої версії Node в 2009 році було випущено ряд зворотно несумісних змін.js а це означає, що іноді потрібно додатковий час і зусилля для підтримки сумісності з останньою версією API Node.js.

Ruby on Rails. Ruby on Rails — це серверний веб-фреймворк, написаний на Ruby [15]. Він був на піку своєї популярності в 2011 році. Тоді в основному домінував на ринку на стороні сервера, і розробники Ruby on Rails були дуже затребувані. Хоча попит на Ruby on Rails розробників все ще існує, популярність фреймворку вже багато років перебуває на постійному зниженні.

Справа не в тому, що Ruby on Rails з роками погіршився (все навпаки), але його повільна швидкість виконання, відсутність гнучкості та висока вартість неправильних рішень зробили його менш бажаним в очах багатьох підприємств. Наприклад, Twitter вирішив відмовитися від Ruby on Rails на користь Java в 2011 році.

Але для підприємств, які не планують досягати масштабу Twitter, Ruby on Rails залишається відмінним вибором з багатьма перевагами, включаючи його високу швидкість розвитку, яскраву спільноту, фантастичне інструментування та сильне дотримання стандартів.

Django. Django — це фреймворк, який підтримується Django Software Foundation (DSF) [16]. Так само, як Ruby on Rails, він був випущений в 2005 році. Його головна мета – спростити створення складних веб-сайтів, орієнтованих на бази даних. Враховуючи, що Django зараз навіть більш популярний, ніж Ruby on Rails, досить безпечно сказати, що це вдалося.

Python є третьою найактивнішою мовою на GitHub, а також найбільш потрібною технологією згідно з опитуванням розробників 2018 року від StackOverflow. Попит на розробників Python настільки високий, що вони можуть попросити значно вищу зарплату, ніж JavaScript або навіть Ruby розробників.

Деякі з найбільш відомих компаній, які використовують Django: Disqus, Instagram, Spotify, YouTube, The Washington Post, Bitbucket, Dropbox і Eventbrite. Django розроблений, щоб допомогти розробникам якомога швидше пройти шлях програми від концепції до готового продукту, що робить його придатним для команд, що працюють у гнучких середовищах. Він розроблений з думкою про безпеку сервісу, і він може гнучко масштабуватися для задоволення будь-яких потреб.

Єдине, що слід мати на увазі, це те, що Django - це монолітний і щільно зв’язний продукт, який примушує своїх користувачів діяти за шаблоном , але деякі бачать це як перевагу.

Неможливо сказати що одна технологія краща за іншу і тільки її варто використовувати. В залежності від задач та можливостей підходять різні технології, які задовільнять вимоги. В нашому випадку Node.js буде хорошим вибором через одну мову програмування на сторонах клієнта та сервера. Веб-сервіс «Файловий менеджер» передбачає інтенсивне навантаження на мережу

та роботу з файловою системою і має бути розрахований на велику кількість користувачів – Node.js чудово підходить під ці цілі, через свою асинхронну модель роботи з I/O та хорошу реалізацію мережевого стеку [17].

1.3.2 MySQL в якості СКБД

MySQL представляє реляційну систему керування базами даних (СКБД). Сьогодні це одна з найпопулярніших систем управління базами даних. Оригінальним розробником цієї бази даних була шведська компанія MySQL AB. У 1995 році вона випустила перший реліз MySQL.

Поточна версія СКБД - 8.0, яка була випущена в січні 2018 року, для якої постійно виходять підверсії. MySQL є кросплатформною, має дистрибутиви для різних ОС, включаючи найпопулярніші версії Linux, Windows, MacOS.

У реляційних системах керування базами даних БД ґрунтується на реляційній моделі. Це означає, що всі таблиці мають принаймні одне відношення до іншої таблиці, і жодна не існує без зв'язку.

MySQL має велику надійність, щоб працювати без перебоїв, не вимагаючи складного процесу усунення несправностей через вузькі місця або інші уповільнення. Вона також включає в себе ряд механізмів підвищення продуктивності, таких як підтримка індексів, утиліти загрузки та кешування в пам'яті. MySQL використовує InnoDB як рушій зберігання даних, який забезпечує високоефективні транзакційні можливості, сумісні з ACID, які забезпечують високу продуктивність та масштабованість. Для вирішення проблеми швидко зростаючої бази даних, Replication MySQL і кластер допомагають масштабувати базу даних. [18]

1.3.3 Вибір клієнтської технології

Тепер варто обрати технологію для front-end частини. Тут вибір зводиться до 3 популярних фреймворків: React, Angular, Vue [19].

Angular, розроблений Google, був вперше випущений в 2010 році, що робить його найстарішим з цього списку. Це фреймворк JavaScript на основі TypeScript. Істотні зміни сталися в 2016 році з випуску Angular 2. Angular 2+ відомий як просто Angular. Хоча AngularJS (версія 1) все ще отримує оновлення, ми зосередимо дискусію на Angular. Остання стабільна версія - Angular 11, яка була випущена в листопаді 2020 року.

Vue, також відомий як Vue.js, є наймолодшим членом групи. Він був розроблений екс-співробітником Google Еваном Ю в 2014 році. За останні три роки Vue спостерігає значне зрушення в популярності, навіть без підтримки великої компанії. Поточна стабільна версія 3.0, випущена у вересні 2020 року. Слід зазначити, що Vue 3 в даний час знаходиться у власному GitHub репозиторії, і перейшов на TypeScript.

React, розроблений Facebook, був спочатку випущений в 2013 році. Facebook широко використовує React у своїх продуктах (Facebook, Instagram і WhatsApp). Поточна стабільна версія - 17.X, випущена в жовтні 2020 року. Варто зазначити що React позиціонує себе як UI-бібліотека, що і є на практиці.

Всі три варіанти мають велику популярність та підтримку спільноти. На рисунку 1.2 можна побачити статистику по використанню сайтами цих технологій:

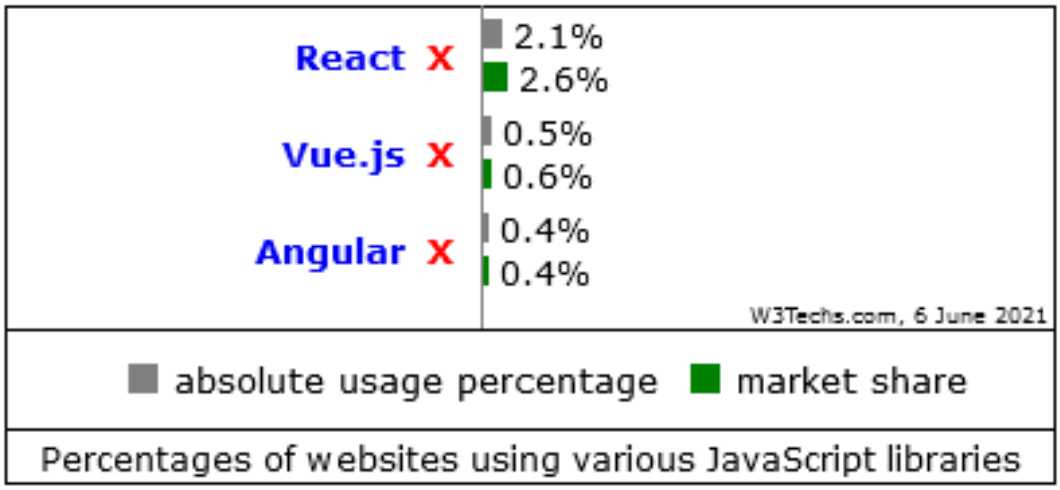

Рисунок 1.2 – Використання front-end фреймворків на ринку та сайтах

Також на рисунку 1.3 наведено історичну діаграму з використання на сайтах за останній рік

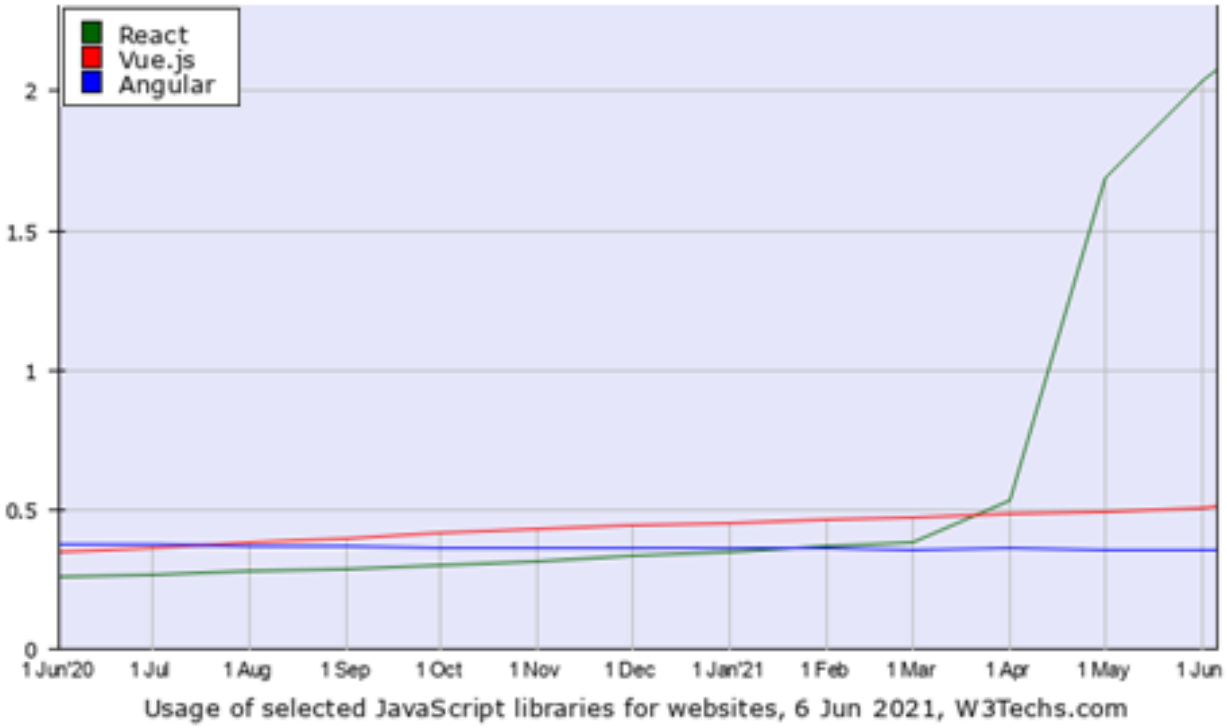

Рисунок 1.3 – історія використання фреймворків за рік

З наведених даних можна робити висновок що React є найпопулярнішою front-end технологією. Саме її ми будемо і використовувати для розробки веб-сервісу «Файловий менеджер». Вибір обґрунтовується тим, що з більшою популярністю присутня і велика підтримка спільноти, таким чином вже багато питань було задано та вирішено. Також серед переваг можна виділити високу продуктивність веб-інтерфейсів написаних на React

1.3.4 Контейнеризація в Docker

Для швидкого розроблення, тестування та розгортання застосунків використовується Docker.

Основний принцип роботи Docker – контейнеризація програм. Цей тип віртуалізації дозволяє упаковувати програмне забезпечення в ізольовані середовища – контейнери. Кожен з цих віртуальних блоків містить всі необхідні елементи для роботи програми. Це дозволяє запускати велику кількість контейнерів на одному хості одночасно [20].

Docker працює в Linux, ядро якого підтримує cgroups, а також ізоляцію простору імен. Є спеціальні Kitematic або Docker Machine утиліти для установки і використання на платформах, інших від Linux.

Переваги використання Docker

- Мінімальне споживання ресурсів – контейнери не віртуалізують всю операційну систему (ОС), а використовують ядро хоста і ізолюють програму на рівні процесу. Останні споживають набагато менше ресурсів локального комп'ютера, ніж віртуальна машина;
- Високошвидкісне розгортання – додаткові компоненти можуть не встановлюватися, а використовувати готові docker-образи(шаблони). Наприклад, немає сенсу весь час встановлювати та налаштовувати Linux Ubuntu. Досить встановити його один раз, створити образ і постійно ним користуватися, тільки оновлюючи версію при необхідності;
- Зручне приховування процесу – для кожного контейнера можна використовувати різні методи обробки даних для приховування фонових процесів;
- Робота з небезпечним кодом – технологія ізоляції контейнера дозволяє запускати будь-який код, не вражаючи ОС.
- Просте масштабування – будь-який проект можна розширити, ввівши нові контейнери;
- Зручний запуск – додаток всередині контейнера можна запустити на будь-якому docker-хості;

– Оптимізація файлової системи – образ складається з шарів, які дозволяють використовувати файлову систему дуже ефективно.

Кожен Dockerfile по суті є скриптом, який автоматично виконує певні дії або команди в базовому образі для формування нового образу. Всі Dockerfile починаються з позначення FROM, дальше йдуть різні методи, команди та аргументами або умови, після виконання яких буде створено контейнер Docker. Детальніше про Dockerfile буде розглянуто в практичній частині роботи, при написанні його конфігурації для нашого проекту.

Використовуючи контейнери розробка веб-сервісу «Файловий менеджер» значно прискориться та спроститься. Це дасть змогу підняти додаток всього лиш однією командою.

1.4 Висновок до першого розділу

В першому розділі кваліфікаційної роботи досліджено стан сфери веб-розробки в сучасному світі, наведено коротку історію Всесвітньої мережі Інтернет, та оглянуто основні веб-стандарти для технологій. Також для визначення та формування технічного завдання провелася порівняльна характеристика готових рішень серед веб-застосунків схожого типу. Закінчується розділ вибором технологій для розробки веб-сервісу «Файловий менеджер»: Node.js для сервера разом з СКБД MySQL та бібліотека React для будування інтерфейсу користувача. Все це буде контейнеризовано в Docker для зручнішого процесу проектування.

2 ПРОЕКТУВАННЯ ВЕБ-СЕРВІСУ «ФАЙЛОВИЙ МЕНЕДЖЕР»

В цьому розділі наведено процес та результати розроблення веб-застосунку з використанням обраних технологій: HTML, CSS, JS та React для клієнтської сторони, та Node.js з СКБД MySQL в контейнерах Docker на сервері.

2.1 Налаштування проекту

Спочатку необхідно задати структуру для проекту, встановити необхідні бібліотеки та інструменти. Припускається що у користувача вже встановлені базові інструменти, а саме: сучасний браузер, редактор коду та Docker. Саме з Docker буде завантажено образи Node.js та MySQL та створено відповідні контейнери.

2.1.1 Конфігурація Dockerfile та docker-compose.yml

Як було описано в пункті 1.3.4 Dockerfile це конфігураційний файл в якому описані певні інструкції, які виконує Docker. Згідно з документацією [21] потрібно створити файл з назвою «Dockerfile».

Починається файл з інструкції «FROM», яка визначає на основі якого стандартного образу буде створено наш власний образ. Так звані батьківські образи знаходяться на серверах Docker Hub. Також важливою інструкцією є «RUN», яка виконає команду та застосує її для створення нового шару в нашому образі. Останньою для розгляду інструкцією є «EXPOSE», яка дозволяє прокинути порти для контейнера на зовні, щоб мати можливість підключитися до сервісів всередині. Всі інші інструкції, які буде використано для побудови Dockerfile можна знайти в офіційній документації.

В результаті для нашого проекту було розроблено наступний Dockerfile, що знаходиться в лістингу 2.1:

Лістинг 2.1 – Dockerfile

```
#node dockerfile
FROM node:lts-slim

WORKDIR /usr/src/my-app

COPY package.json .

RUN npm install

EXPOSE 3000

COPY . .

RUN ["chmod", "+x", "/usr/src/my-app/scripts/run.sh"]

RUN npm run build:frontend

RUN npm run build:backend

RUN apt update && apt install netcat -y
```

Новими командами є «WORKDIR» та «COPY», які відповідно встановлюють робочу директорію в образі та копіюють файли з хоста в образ.

Проте було налаштовано лише образ для Node.js. Щоб налаштувати ще й MySQL та об’єднати ці два контейнери використаємо програму «docker-compose», яка описує та запускає мультиконтейнерні Docker застосунки [22]. Згідно з документацією було побудовано наступний «docker-compose.yml» файл що зображено в лістингу 2.2:

Лістинг 2.2 – Вміст файлу docker-compose.yml

```
version: "3.3"

services:

    db:
        image: mysql:8.0.21
        container_name: db
```

```
        command:--default-authentication-
plugin=mysql_native_password
            restart: always
            environment:
              MYSQL_DATABASE: "${MYSQL_DB}"
              MYSQL_USER: "${MYSQL_USER}"
              MYSQL_PASSWORD: "${DB_PASS}"
              MYSQL_ROOT_PASSWORD: "${DB_PASS}"
            volumes:
                - "${ROOT_DIR}/db:/var/lib/mysql/"
            ports:
                - "3306:3306"
            expose:
                - "3306"

        web:
            command: scripts/run.sh
            build:
                context: .
                dockerfile: Dockerfile
            container_name: web
            restart: always
            volumes:
                - "${ROOT_DIR}:/usr/src/my-app/user_files"
            ports:
                - "3000:3000"
            depends_on:
                - "db"
```

Як бачимо складність вмісту цього файлу є вищою ніж в Dockerfile, тому буде досить коротко описано те, що знаходиться в цьому файлі. Запустивши docker-compose з командою «up –build», буде прочитано конфігураційний файл та створено два образи на основі яких запустяться відповідно 2 контейнери. Ці два образи – це *web* (Node.js: сервер) та *db* (MySQL). Для *web* образу використається готовий Dockerfile, а для *db* буде сконфігуровано в тій самій секції, використовуючи команди, які схожі до тих що є для Dockerfile.

2.1.2 Ініціалізація проекту, завантаження пакетів та бібліотек

Нам потрібно налаштувати наш проект, щоб використовувати node package manager(npm), який поставляється разом з Node.js. Розробники використовують npm для доступу до пакунків Node і керування ними. Node

пакети – це модулі, що містять код JavaScript, написаний іншими розробниками. Їх можна використовувати , щоб зменшити дублювання коду та уникнути помилок [23].

Для ініціалізації проекту використовується команда «npm init», потрібно буде відповісти на типові запитання щодо проекту: назва, автори, тощо. NPM додає файл package.json до того ж каталогу, що і src.

Для додавання та встановлення бібліотек використовується команда «npm install name_of_package». Нам необхідно встановити наступні бібліотеки для front-end та back-end розробки:

Font-end

– Webpack, webpack-cli, webpack-dev-server – компонувальник вихідного коду в так звані бандли;
– @babel/cli, @babel/core, @babel/node, babel-loader – транспілятор коду з однієї версії в іншу;
– React, react-dom, react-router-dom – UI-бібліотека

Back-end

– Bcrypt – для шифрування паролів;
– Dotenv – завантаження змінних середовища;
– express, express-jsonschema – популярний веб-фреймворк для сервера;
– jsonwebtoken – шифрування даних для cookie;
– knex – міграції та зв'язок з БД;
– multer – завантаження файлів ;
– mysql – драйвер для роботи з БД;
– passport, passport-jwt, passport-local – для налаштування реєстрації та логіну.

Вище було наведено основні пакети, які необхідні для розробки веб-сервісу «Файловий менеджер». Особливу увагу слід звернути на Webpack в поєднанні з babel, так як вони забезпечують роботу бібліотеки React.

Саме завдяки Babel ми можемо використовувати JSX (розширений синтаксис для JS [24]) при розробці React-додатків. Наприклад, ось код, який використовує JSX – лістинг 2.3:

Лістинг 2.3 – JSX-синтаксис

```
import React from "react";

function App(){
 return(
  <div>
     <b>Hello world!</b>
  </div>
 )
}

export default App;
```

Цей код виглядає акуратно, зрозуміло, його легко читати і редагувати. Дивлячись на нього, ви можете відразу зрозуміти, що він описує компонент, який повертає елемент «div» що містить текст «Hello World!», виділений жирним шрифтом. І ось приклад коду на лістингу 2.4, що робить те ж саме, без JSX:

Лістинг 2.4 – React «під капотом»

```
"use strict";
Object.defineProperty(exports, "__esModule", {
  value: true
});
var _react = require("react");
var _react2 = _interopRequireDefault(_react);
function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { default: obj }; }
function App(props) {
  return _react2.default.createElement(
    "div",
    null,
    _react2.default.createElement(
      "b",
      null,
      "Hello world!"
    )
  );
```

```
}
exports.default = App;
```

Встановивши необхідні модулі можна і приступати до розробки застосунку. Для зручності створимо папки *client* та *server*, які будуть розділені не тільки у файловій системі, а й архітектурно.

2.2 Розробка додатка

Використовуючи готовий дизайн з параграфа 1.2 буде зверстано сам сайт за допомогою React та його JSX-розмітки, звісно, це все перетворюється в стандартні HTML, CSS та JS на виході. Буде розроблено схему БД та реалізовано її за допомогою міграцій. Останній крок – це розробка прикладного програмного інтерфейсу (API) із автентифікацією та авторизацією.

2.2.1 Розробка UI

Почнемо розробку зі створення вхідного файлу *index.html*. Сервер на всі запити сторінок буде віддавати саме його, а маршрутизацією займеться бібліотека *react-router-dom*, встановлена попередньо. Цей принцип керування маршрутизацією називається Single Page Application (SPA). Вміст файлу наведено в лістингу 2.5:

Лістинг 2.5 – index.html – точка входу

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta     name="viewport"     content="width=device-width,
initial-scale=1.0">
  <title>Document</title>
</head>
<body>
```

```
  <div id="root"></div>
</body>
</html>
```

Разом з цим файлом надсилається й *index.js* – JavaScript код з *React* в лістингу 2.6:

Лістинг 2.6 – index.js – початок JS

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

Виникає закономірне питання щодо того як ці файли поєднаються, адже у *index.html* немає підключення JS-файлу. За нас про це подбає *Webpack,* він згенерує новий HTML-файл в директорію *dist* та додасть до нього всі необхідні імпорти.

Далі розглянемо створення деяких загальних компонентів в *React*. Компоненти дозволяють розділити інтерфейс на незалежні, багаторазові частини, і таким чином думати про кожну частину в ізоляції. В загальному компоненти схожі на функції в JS [25].

Для прикладу, в лістингу 2.7, наведено компонент *Input*, який використовується для створення HTML-елемента з однойменною назвою.

Лістинг 2.7 – простий Input-компонент

```
import React from 'react';
import './Input.css';

export const Input = (props) => {
  return (
    <div>
      <input
        {...props}
        type={props.type || 'text'}
        name={props.name || ''}
        value={props.value || ''}
        placeholder={props.placeholder || ''}
        onChange={props.onChange || null}
```

```
            className={props.className || ''}
          />
        </div>
      );
    };
```

Передаючи різні *props* можна отримати компонент з різними властивостями: кнопка, поле вводу, тощо.

Приклад використання цього компонента наведено в лістингу 2.8:

Лістинг 2.8 – застосування Input-компонента

```
<Input
   type="email"
   name="email"
   className={`${
     !values.email.isValid && values.email.touched && 'red-
border' margin`}
   value={values.email.value}
   placeholder="Email"
   onChange={handleChange}
 />
 <Input
   type="password"
   name="password"
   className={
     !values.password.isValid && values.password.touched &&
'red-border'}
   value={values.password.value}
   placeholder="Password"
   onChange={handleChange}
 />
```

Для маршрутизації використовуються компоненти *Route,* яким треба передати пропс *path.* Так при переході за іншою адресою, відрендериться відповідний компонент в якого співпадає пропс *path* з URL-адресою в браузері, про це детальніше в лістингу 2.9:

Лістинг 2.9– маршрутизація за допомогою компонентів *Route*

```
< import React from 'react';
import HomePage from '../pages/HomePage';
import AuthPage from '../pages/AuthPage';
import DownloadPage from '../pages/Download';
```

```
import { BrowserRouter as Router, Switch, Route } from 'react-
router-dom';

export default function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/">
          <HomePage />
        </Route>
        <Route path='/file/:hash'>
          <DownloadPage />
        </Route>
        <Route path="/">
          <AuthPage />
        </Route>
      </Switch>
    </Router>
  );
}
```

В розробці таких компонентів і їх подальшого використання полягає створення UI для «Файлового менеджера», для більш детального ознайомлення з вихідним кодом можна звернутися в додаток А.

2.2.2 Схема та міграції БД

Для розробки схеми бази даних використовувався сервіс *quickdatabasediagrams.com*. Quick Database Diagrams (QuickDBD) — це простий онлайновий інструмент для швидкого створення діаграм баз даних за допомогою введення тексту. Він розробляє преміальні діаграми і дозволяє ділитися своїми діаграмами в Інтернеті.

Безкоштовна версія QuickDBD обмежує вас однією діаграмою. Їх Pro план дозволяє 30 діаграм за $ 14 на місяць, хоча вони, як відомо, запускають акції, які дають змогу отримати план Pro безкоштовно.

Підтримувані бази даних:

– SQL Server;

– Oracle;

– MySQL;
– PostgreSQL;
– MariaDB.

На рисунку 2.1 приведено структуру БД, їх зв’язок між собою. Створено 3 таблиці: *users*, *Objects*, *shared_objects*. Таблиця *users* має зовнішній ключ до *Objects*, а вона до *shared_objects*.

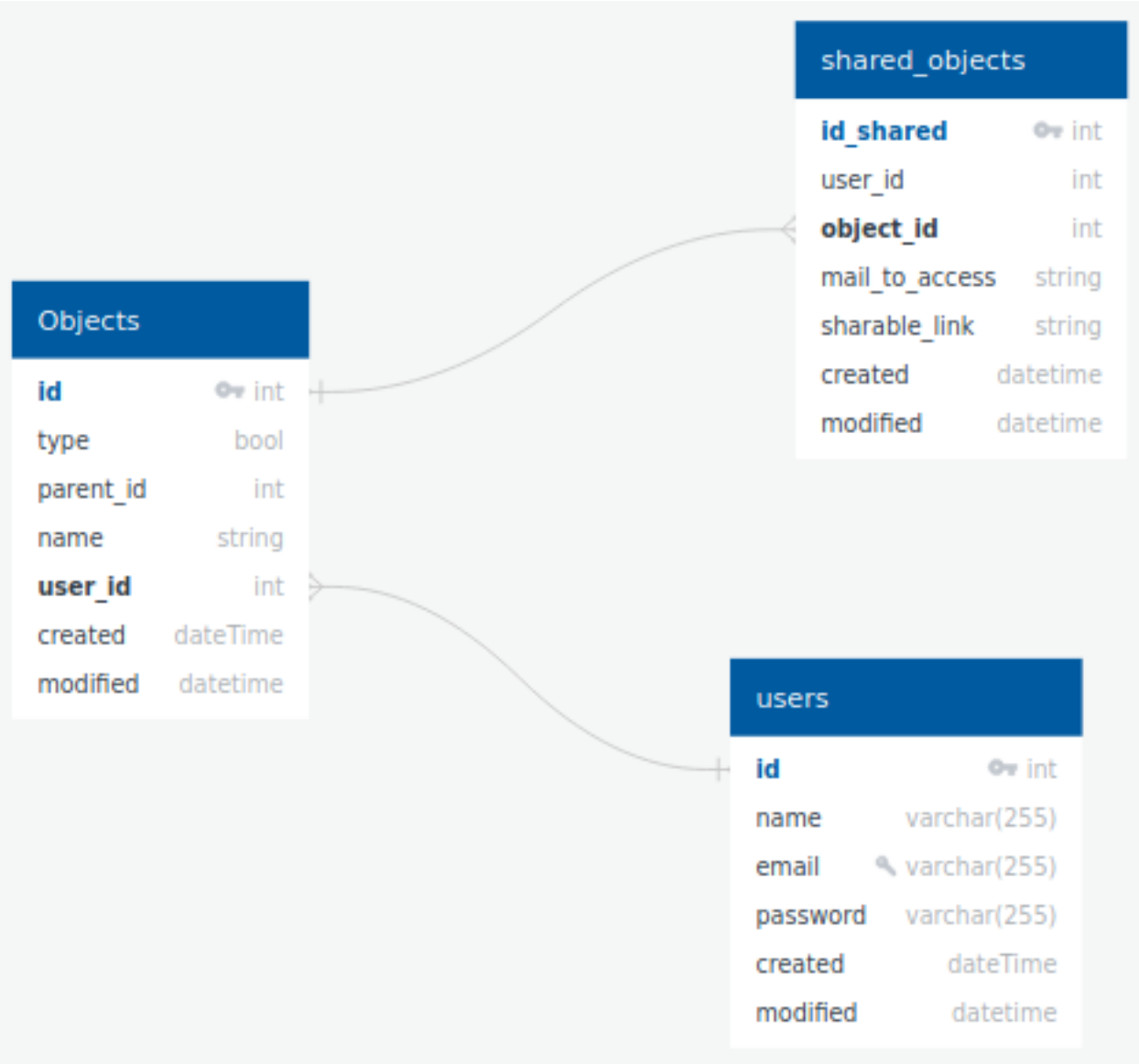

Рисунок 2.1 – Структура таблиць БД

Далі показується опис кожної таблиці БД. У таблиці 2.1 наведено структуру *users* бази даних веб-сервісу «Файловий менеджер»

Таблиця 2.1 – Поля та типи для *users*

Поле	**Тип**
id	PK int AUTOINCREMENT
name	varchar(255)
email	UNIQUE varchar(255)
password	varchar(255)
created	dateTime
modified	dateTime

Об’єкти таблиці *Objects* (таб. 2.2) можуть бути двох типів: файл або папка. Файл може бути всередині папки вказуючи в полі *parent_id* ідентифікатор папки, при цьому всі файли будуть зберігатися в одному каталозі, без реальних папок – вони будуть тільки на клієнті.

Таблиця 2.2 – Поля та типи для *Objects*

Поле	**Тип**
id	pk int AUTOINCREMENT
type	bool
parent_id	int
name	string
user_id	fk >users.
created	dateTime
modified	dateEime

Остання таблиця в БД – *shared_objects*, призначена для файлів, які будуть поширені для інших користувачів, яку подано в таблиці 2.3. При поширенні файлів буде вказуватися email – для зареєстрованих користувачів, а згенерувавши лінк можна буде поширити для будь-кого.

Таблиця 2.3 – Поля та типи для *shared_objects*

Поле	**Тип**
id_shared	pk AUTOINCREMENT int
user_id	int
object_id	fk >Objects.
id	int
mail_to_access	string
sharable_link	string
created	datetime

Для створення структури в базі, будуть використовуватися міграції – програмний опис таблиць та застосування або відкочування програмою змін в базі. Міграції – це зручний спосіб узгоджено змінювати схему бази даних. Вони використовують код вашої мови для розробки, тому не доведеться писати SQL вручну, дозволяючи схемі та новим змінам бути незалежними від бази даних [26]

На рисунку 2.2 наведено всі міграції, які є на кінцевому етапі проекту:

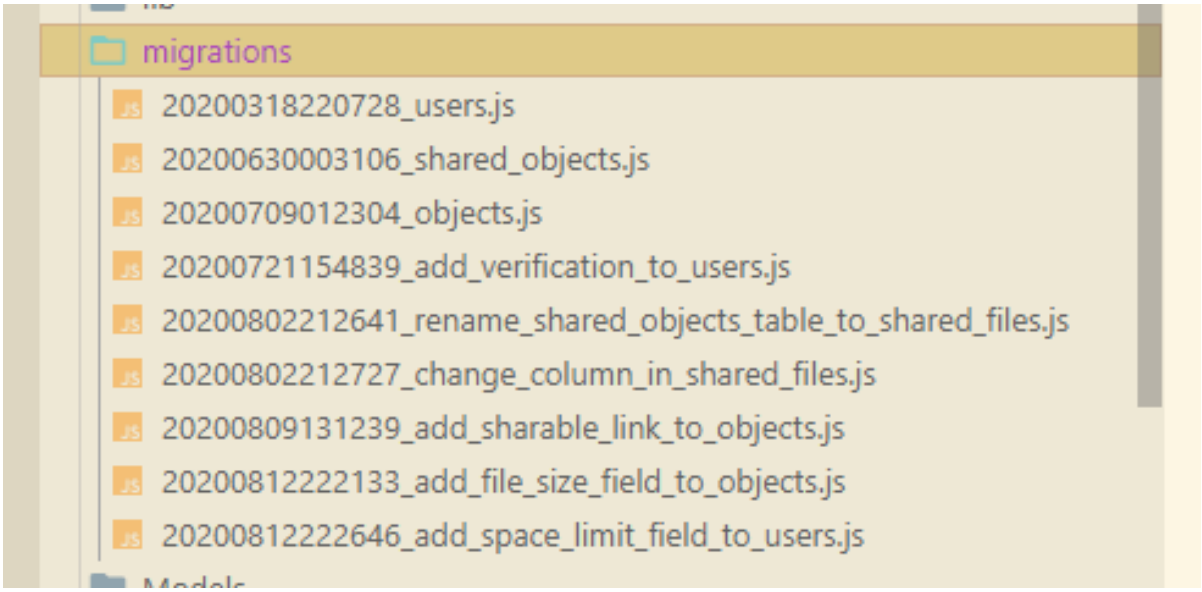

Рисунок 2.2 – міграції написані мовою JavaScript

Для прикладу опишемо створення таблиці для файлів в лістингу 2.10:

Лістинг 2.10 – Міграційний файл Objects

```
exports.up = function (knex) {
  return knex.schema.createTable('objects', (table) => {
    table.increments('id');
    table.enu('type', ['file', 'folder']).notNullable();
    table.integer('parent_id').notNullable();
    table.string('name', 255).notNullable();
    table.integer('user_id').notNullable();
    table.datetime('created').notNullable();
    table.datetime('modified').notNullable();
  });
};

exports.down = function (knex) {
  return knex.schema.dropTable('objects');
};
```

Також наведемо міграцію для додавання поля в таблицю *Objects* в лістингу 2.11:

Лістинг 2.11 – Міграція для додавання поля

```
exports.up = function (knex) {
  return knex.schema.table('objects', (table) => {
    table.string('sharable_link').defaultTo('');
  });
};

exports.down = function (knex) {
  return knex.schema.table('objects', (table) => {
    table.dropColumn('sharable_link');
  });
};
```

Кожен файл повинен мати як створення таблиці(зміни поля, тощо) так і відкочення до початкового стану. Тут це репрезентується функціями up та down. Команда *knex migrate:latest* запустить міграції для бази, а *knex migrate:rollback* поверне базу до попереднього стану.

2.2.3 Розробка API. Автентифікація та Авторизація

Далі почнемо з *index.js* в папці *server*, спочатку підключаємо деякі головні бібліотеки, що показано в лістингу 2.12:

Лістинг 2.12 – Імпорт модулів

```
import express from 'express';
import path from 'path';
import loginSignup from './API/loginSignUpRoute.js';
import getObjects from './API/getObjects.js';
import postObjects from './API/postObjects';
```

В лістингу 2.13 імпортували *express* для створення веб-сервера, *path* для роботи із шляхами в системі та API-файли, там є визначення URL'ів для передачі та отримання даних

Лістинг 2.13 – Базовий веб-сервер

```
const app = express();
app.get('*', (req, res) => {
     res.sendFile(path.resolve(__dirname +
     '/../client/public/index.html'));
});
app.listen(3000, () => {
     console.log('listening on port 3000');
});
```

«const app = express();» – ініціалізація змінної сервера, тепер можна додавати маршрути та використовувати методи для запуску сервера.

«app.get('*', (req, res) => {}» – створити для всіх можливих маршрутів з GET методом таку функцію-обробник. В результаті на кожен GET запит буде віддаватися HTML-файл.

«app.listen(3000, () => {})» – запуск сервера на порті 3000 та очікування на запити.

На основі цієї структури з базового сервера буде побудовано решта компонентів.

Ключовим функціоналом в нашому «Файловому менеджері» є отримання файлів з сервера та їх завантаження на нього, це реалізуємо наступним чином що подано в лістингу 2.14:

Лістинг 2.14 – Отримання всіх файлів користувача

```
objects.post('/objects',auth().authenticate(),async (req,res) =>{
          try {
               const data = {};
               let allObjects = await object.getObjectsSingleLvl(
               //add all objects that is in one folder
               req.body.parentId,
               req.user.id
               )
          res.json(allObjects);
          }
          catch(error){
               console.log(error);
     });
```

Для завантаження файлів написано функцію-обробник, наведену в лістингу 2.15. Файл завантажується на сервер в тимчасову папку, при успішній ідентифікації користувача назва файлу зберігається в базу, та сам файл переміщається в сховище для файлів з ідентифікатором в якості імені:

Лістинг 2.15 – Завантаження файлу на сервер

```
      objects.post('/file', upload.single('files'), async (req,
res) => {
        try {
          const total = await object.getTotalSize(req.user.id);
          const limit = await user.getOneWithColumns(
            { user_id: req.user.id },
            'space_limit'
          );
          if (total[0].total + req.file.size > limit.space_limit)
{
            res.json({ error: true, message: 'You exceeded your
space limit' });
          } else {
            const id = await object.addOne({
              type: 'file',
              parent_id: req.body.parent_id,
              name: req.file.originalname,
              user_id: req.user.id,
```

```
            file_size: req.file.size,
          });

          moveFile(req.file.path, './user_files/' + req.user.id,
id[0])
            .then(() => {
              res.json('File uploaded!');
            })
            .catch((err) => {
              res.json('Internal error');
              console.log(err);
            });
        }
      } catch (error) {
        console.log(error);
        res.json('Internal error');
      }
    });
```

Надіславши файл з браузера за шляхом */file* також перевіряємо чи не перевищено ліміту сховища

API-файли мають таку структуру: об’єкт-роутер, який тримає методи та шляхи (створюється з *express.Router()*), та самі обробники на шляхах, далі в лістингу 2.16 наведено приклад з методом для отримання поширеного файлу:

Лістинг 2.16 – Об’єкт-роутер та обробник шляху «*/sharedFile/:hash*»

```
import express from 'express';
const objects = express.Router();
const app = objects.get('/sharedFile/:hash', async (req, res) =>
{
  const hash = req.params.hash;
  try {
    if (hash.length > 10) {
      const owner = await object.getOne({ sharable_link: hash });
      res.download(
        './user_files/' + owner.user_id + '/' + owner.id,
        owner.name
      );
    } else {
      res.json('Check the link');
    }
  } catch (error) {
    console.log(error);
    res.json('Check the link');
  }
});
```

Тут для того щоб отримати публічний файл потрібно перевірити чи він має унікальний хеш, який використовуватиметься як URL.

Авторизація/Автентифікація. За шляхом */register* створюємо нового користувача. Дані валідуються обробником – *validate({ body: registerSchema }).* Для безпеки паролі хешуються, тому в разі витоку даних користувачів у зловмисників не буде можливості використати паролі для отримання конфіденційних даних. В лістингу 2.17 наведено код реалізації реєстрації:

Лістинг 2.17 – Роутер для реєстрації

```
loginSignup.post(
  '/register',
  validate({ body: registerSchema }),
  async (req, res) => {
    try {
      const hashedPassword = await bcrypt.hash(req.body.password,
saltRounds);
      try {
        /**
         * register user
         */
        const data = await user.add({
          name: req.body.name,
          email: req.body.email,
          password: hashedPassword,
          verified: true,
        });
        /**
         * send email
         */
        res.json({ redirect: '/', success: true, message: data });
      } catch (e) {
        console.log(e);
        let message = '';
        switch (e.errno) {
        case 1062:
          message = 'The user is already exists';
          break;
        default:
          message = 'Unknown error. Try again';
          break;
        }
        res.json({ redirect: '/register', success: false, message
});
      }
    } catch (e) {
```

```
      console.error(e);
      res.json({
        redirect: '/register',
        success: false,
        message: 'Internal error',
      });
    }
  }
);
```

В папці *server/lib/authentication* створено файл для автентифікації користувачів та генерації токену, це показано в лістингу 2.18. Використовуються пакети *jwt, passport, passjwt*. Функція *authenticate()* буде проводити автентифікацію.

Лістинг 2.18 – Автентифікація за допомогою JWT-токена

```
import jwt from 'jsonwebtoken';
import passport from 'passport';
import passjwt from 'passport-jwt';
import { jwtSecret, jwtOptions } from '../../config/config.js';

const JwtStrategy = passjwt.Strategy;

const auth = () => {
  passport.use(
    new JwtStrategy(jwtOptions, (payload, done) => {
      if (!payload.id) {
        return done(null, false, { message: 'JWT token isn`t
valid' });
      } else {
        return done(null, { id: payload.id });
      }
    })
  );
  return {
    init: () => passport.initialize(),
    authenticate: () => passport.authenticate('jwt', { session:
false }),
  };
};

const generateToken = (payload) =>
  jwt.sign(payload, jwtSecret, { expiresIn: '1h' });

export { auth, generateToken };
```

В цьому пункті були описані основні методи для завантаження та надсилання файлів, а також авторизації й автентифікації користувачів. Щоб детальніше ознайомитися з усіма функціями можна звернутися в додаток А.

2.3 Тестування «Файлового менеджера»

Після запуску веб-сервісу ”Файловий менеджер” командою *docker-compose up*, можна проводити тестування функціоналу.

Було проведено тестування вкладок, переходів на сторінки. Проведемо тестування реєстрації та входу на домашню сторінку: здійснюємо реєстрацію, це зображено на рисунку 2.3. Паролі повинні співпадати, інакше буде виведено повідомлення про помилку:

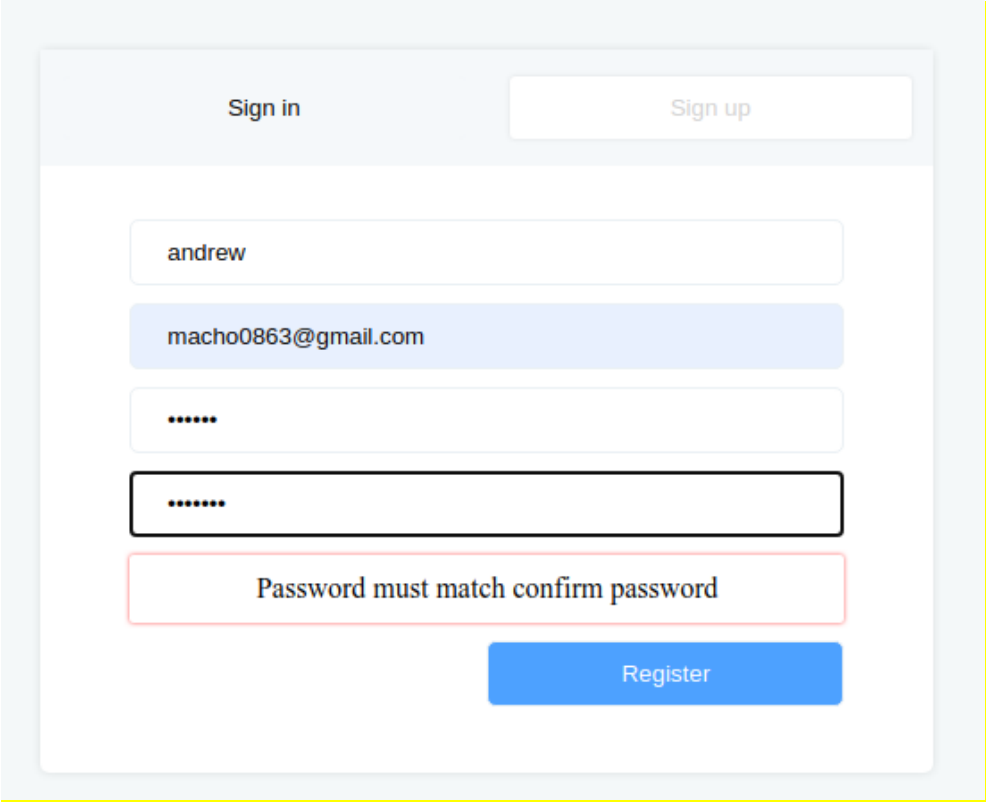

Рисунок 2.3 – Сторінка реєстрації

Далі увійдемо, ввівши правильні реєстраційні дані, що зображено на рисунку 2.4:

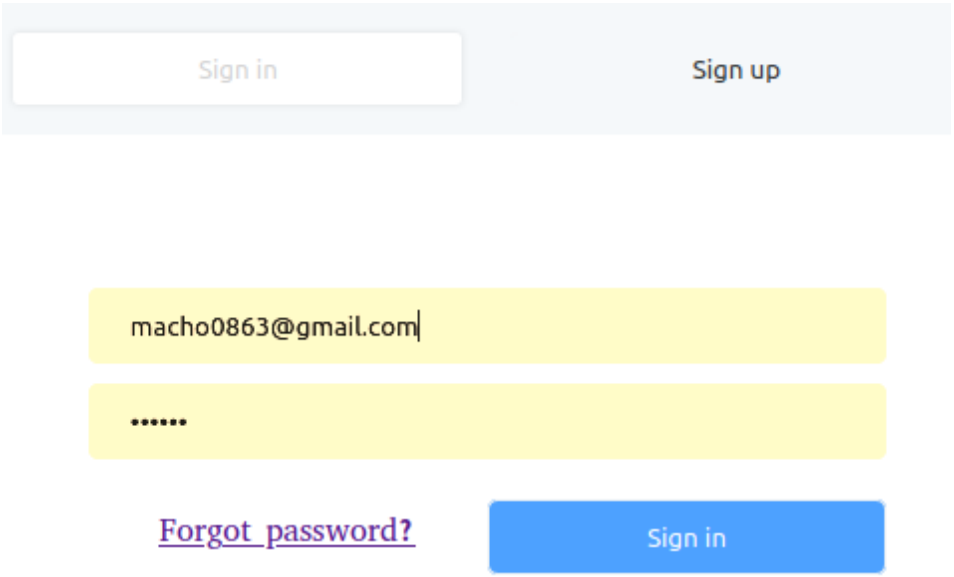

Рисунок 2.4 – Вхід з правильними даними

Перевіряємо можливість завантаження файлів. При кліку на кнопку додавання відкриється модульне вікно веб-додатку, далі при виборі файлів відкривається діалогове вікно браузера для вибору. Файли завантажуються до браузера, і тоді передаються на сервер при кліку *upload a file*:

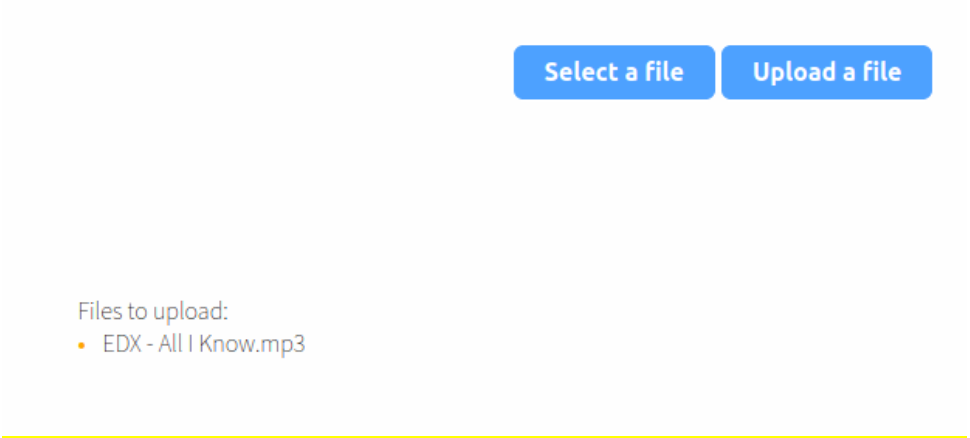

Рисунок 2.5 – Додавання файлу

Після додавання файл з’явився на головній сторінці в користувача, який надіслав файл на сервер, що подано на рисунку 2.6, його можна завантажити, видалити, поширити за посиланням та за email:

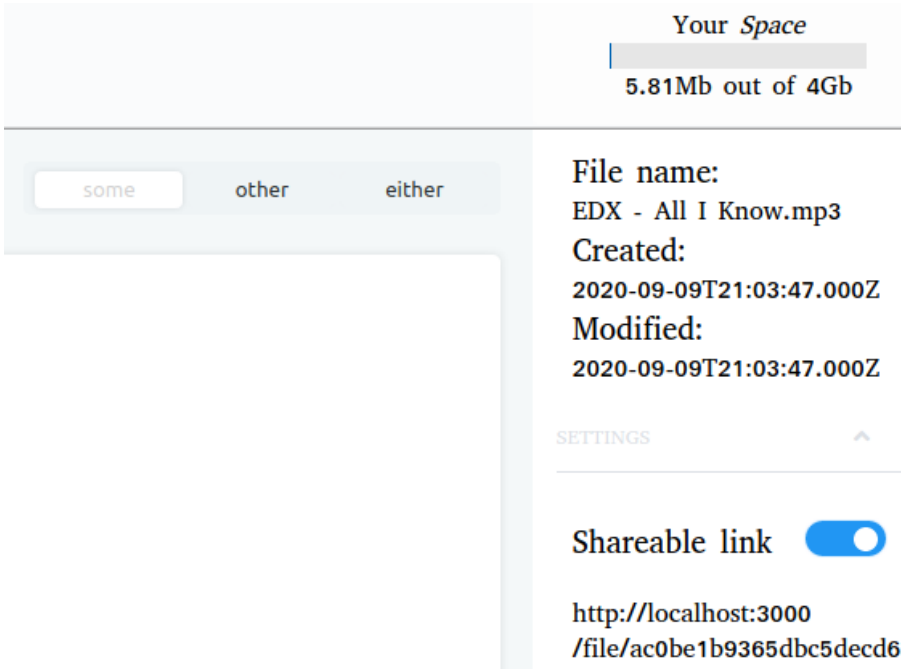

Рисунок 2.6 –Властивості файлу

Якщо файл було поширено для імейлу, то він з’явиться у користувача якому його поширили, що зображено на рисунку 2.7:

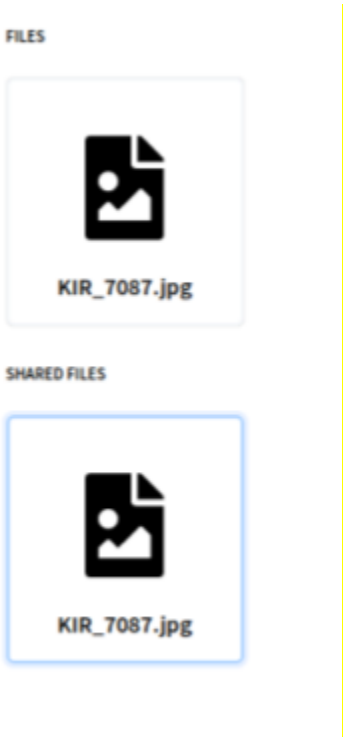

Рисунок 2.7 – поширення за email

Остання можливість – це поширення файлу за згенерованим лінком публічно, що зображено на рисунку 2.8, тоді доступ матимуть неавторизовані користувачі:

Рисунок 2.8 – поширення за лінком

Підсумовуючи можна сказати, що в цьому параграфі відбулося тестування найосновнішого функціоналу спроектованого додатка – від реєстрації та входу користувача в систему до завантаження та поширення файлів іншим користувачам.

2.4 Висновок до другого розділу

В другому розділі кваліфікаційної роботи розроблено веб-сервіс «Файловий менеджер». Робота розділена на 3 етапи: налаштування проекту, власне сама розробка сервісу, а також тестування реалізованого функціоналу.

При конфігуруванні проекту для контейнеризації використано інструмент Docker-compose, який є оболонкою над звичайним Docker та дозволяє за допомогою однієї команди підняти весь веб-сервіс.

Для розробки UI створювалися так звані React-компоненти, які інкапсулюють логіку та зменшують дублювання коду, що дає змогу швидко та якісно писати код.

Серверна розробка відбувалася з використанням Node.js та MySQL. Було створено основні методи для роботи «Файлового менеджера», а саме:

– завантаження файлів на сервер;
– завантаження із сервера;
– поширення за публічним лінком;
– поширення за email.

В результаті отримано готовий веб-сервіс, який здатний виконати поставлені в цій бакалаврській роботі цілі. В кінці його також протестовано для перевірки функціоналу.

3 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ХОРОНИ ПРАЦІ

3.1 Роль центральної нервової системи в трудовій діяльності людини

Нервова система має найголовніше значення в організмі людини. Вона координує, регулює роботу всіх внутрішніх органів і здійснює зв'язок організму із зовнішнім середовищем.

Нервова система людини складається із центральної (ЦНС), яка включає головний і спинний мозок і периферійної (ПНС), яка складається з нервових волокон, що відходять від головного і спинного мозку.

За функціями нервову систему поділяють на соматичну і вегетативну. Соматична нервова система регулює опорно-руховий апарат і всі органи чуття, а вегетативна - процес обміну речовин та роботу всіх внутрішніх органів (серця, нирок, легенів). Найпростіші рухи регулює спинний мозок. Довгастий мозок керує процесами травлення, дихання, кровообігу та іншими життєво важливими функціями. Підкіркова і кіркова частини головного мозку керують усією психічною діяльністю людини.

Центральна нервова система виконує рефлекторну, інтегративну та координаційну функції.

Рефлекторна діяльність мозку зумовлена безумовними та умовними рефлексами. Безумовні рефлекси є вродженими, мають велику стійкість і забезпечують пристосування організму до зовнішнього середовища. Умовні рефлекси набуваються залежно від обставин, розширюють діапазон пристосувальницьких можливостей організму і згасають, якщо потреби в них немає.

Стійка і злагоджена система умовних рефлексів формується у процесі навчання і забезпечує виконання певного виробничого завдання. Стійкість системи умовних рефлексів може бути порушена при відхиленні трудової

діяльності від програми, а надійність - під впливом несприятливих виробничих чинників. Такі порушення, якщо не вжити належних заходів, можуть призвести до зниження працездатності, травм або нещасних випадків.

Виконуючи інтегративну функцію, ЦНС забезпечує злагоджену взаємодію всіх органів і систем організму, підтримує його стійкий внутрішній стан. Несприятливі умови праці можуть призвести до стомлення нервової системи, що послаблює її інтегративну функцію і може спровокувати розлад ряду фізіологічних систем: серцево-судинної, шлунково-кишкової, дихальної тощо або призвести до різних захворювань (інфаркти, інсульти, виразкові хвороби).

Завдяки координаційній функції ЦНС здійснює підпорядкування багатьох рефлексів одному, який має на даний час найважливіше значення для організму.

Усі функції центральної нервової системи реалізуються в кожній конкретній реакції організму, забезпечуючи ефект найбільшого пристосування до мінливих умов зовнішнього середовища і підвищуючи фізіологічну опірність організму шкідливим зовнішнім впливам.

Вища нервова діяльність людини заснована на функціях двох сигнальних систем. Анатомічною основою першої сигнальної системи є аналізатори (зоровий, слуховий). Аналізатор - це система нервових клітин, які сприймають і переробляють інформацію, що надходить до них із зовнішнього та внутрішнього середовища організму.

Анатомічною основою другої сигнальної системи, яка властива тільки людині, є мовно-руховий апарат, тісно пов'язаний із зоровим та слуховим аналізаторами, а її подразником є слово. Мова, в усіх її видах, являє собою найбагатше джерело подразників. За допомогою слова передаються сигнали про конкретні подразники, і в цьому випадку слово служить принциповим подразником - сигналом сигналів, є пусковим механізмом дій і вчинків людей. Мова підвищує здатність мозку відображати дійсність, забезпечує аналіз і

синтез, абстрактне мислення, створює можливість для спілкування, використання і передачі життєвого досвіду, досягнень культури і мистецтва. Але в деяких випадках слово може бути негативним подразником і може призвести до розладів нервової системи, порушень функціонування всіх систем організму і, таким чином, стати небезпечним виробничим фактором.

Центральна нервова система бере участь у прийманні, обробці та аналізі будь-якої інформації, що надходить із зовнішнього і внутрішнього середовищ. При виникненні перенавантажень на організм людини нервова система визначає ступінь їхнього впливу і формує адаптаційно-захисну реакцію.

3.2 Вплив шуму на організм людини та розробка заходів щодо його зниженню до допустимих величин для обладнання.

При розробці веб-сервісу «Файловий менеджер» враховано ДСН 3.3.6.037-99 для визначення значень рівня шуму на робочому місці.

Шум — безладне сполучення різних по силі і частоті звуків, здатне впливати на організм. Джерелом шуму є будь-який процес, що викликає місцеву зміну тиску або механічні коливання твердих, рідких або газоподібних середовищ.

Дія його на організм людини пов’язана головним чином із застосуванням нового, високопродуктивного обладнання, механізацією та трудових процесів, у тому числі переходом на великі швидкості при експлуатації різних верстатів і агрегатів. Джерелами шуму в цьому випадку можуть бути двигуни, насоси, компресори, турбіни, пневматичні та електричні інструменти, молоти, дробарки, верстати, центрифуги, бункери та інші установки, що мають рухомі деталі.

Шум — один із найбільш поширених несприятливих фізичних факторів навколишнього середовища. При запуску реактивних двигунів літаків рівень шуму коливається від 120 до 140 дБ; при клепанні й рубанні листової сталі —

від 118 до 130 дБ; роботі деревообробних верстатів — від 100 до 120 дБ, ткацьких верстатів — до 105 дБ [27].

Коли мова йде про вплив шуму, то зазвичай основну увагу приділяють стану органу слуху, так як слуховий аналізатор у першу чергу сприймає звукові коливання і подразнення його є адекватним дії шуму на організм.

Зміни, що виникають в органі слуху, деякі дослідники пояснюють травмуючою дією шуму на периферичний відділ слухового аналізатора внутрішнього вуха. Основною ознакою впливу шуму є зниження слуху по типу кохлеарного неврит у. Професійне зниження слуху буває зазвичай двостороннім.

Захист від шуму повинен забезпечуватися розробкою шумобезпечної техніки, застосуванням засобів і методів колективного захисту, в тому числі будівельно-акустичних, застосуванням засобів індивідуального захисту.

Колективні засоби захисту поділяються на засоби, що знижують шум у джерелі його виникнення, і засоби, що знижують шум на шляху його поширення від джерела до об’єкта, що захищається.

Зниження шуму в джерелі здійснюється за рахунок поліпшення конструкції машини або зміни технологічного процесу. Методи і засоби колективного захисту, в залежності від способу реалізації, поділяються на будівельно-акустичні, архітектурно-планувальні та організаційно-технічні і включають в себе:

– зміну спрямованості випромінювання шуму;
– раціональне планування підприємств і виробничих приміщень;
– акустичну обробку приміщень;
– застосування звукоізоляції.

У низці випадків величина показника спрямованості досягає 10-15 дБ, що необхідно враховувати при використанні установок з направленим випромінюванням, орієнтуючи ці установки так, щоб максимум

випромінюваного шуму був спрямований у протилежний бік від робочого місця.

Раціональне планування підприємств і виробничих приміщень дозволяє знизити рівень шуму на робочих місцях за рахунок збільшення відстані до джерел шуму.

Засоби індивідуального захисту (ЗІЗ) застосовуються в тому разі, якщо іншими способами забезпечити допустимий рівень шуму на робочому місці не вдається. Принцип дії ЗІЗ — захистити найбільш чутливий канал впливу шуму на організм людини — вухо. Застосування ЗІЗ дозволяє попередити розлад не тільки органів слуху, а й нервової системи від дії надмірного подразника.

Найбільш ефективні ЗІЗ, як правило, в області високих частот.

ЗІЗ включають в себе протишумні вкладиші (беруші), навушники, шоломи і каски, спеціальні костюми.

3.3 Висновок до третього розділу

В третьому розділі кваліфікаційної роботи висвітлено питання БЖД та ОП із застосуванням для цієї дипломної роботи.

В першому параграфі описано значення центральної нервової системи в трудовій діяльності людини. Є центральна нервова система (ЦНС) і периферійна (ПНС). При розробці веб-сервісу «Файловий менеджер» застосовувалися інтелектуальні та психологічні можливості ЦНС для осмислення та формування абстрактних ідей, та їхнє застосування на практиці, використовуючи ПНС.

Другий параграф досліджує дію шуму на організм людини та методи його зменшення до рекомендованих норм. При розробці застосунку для цієї дипломної роботи, було забезпечено оптимальний рівень шуму навколишнього середовища, який не перевищував 50 дБ, що дозволило проводити дослідження та проектування з комфортом [28].

ВИСНОВКИ

В процесі роботи над розробкою веб-сервісу «Файловий менеджер» проведено дослідження та аналіз веб-сфери, визначено актуальні технології для проектування веб-застосунку за вибраною темою. Головними цілями цієї кваліфікаційної роботи було опанування нових технологій та здобуття необхідних вмінь в розробці веб-додатків.

В першому розділі кваліфікаційної роботи освітнього рівня «Бакалавр»:

– Подано коротку історію інтернету та веб-розробки, стандарти, та актуальність веб-технологій;

– Проаналізовано готові рішення на ринку хмарних технологій по зберіганню інформації, та сформовано технічне завдання на основі мінімально обов’язкового функціоналу для файлових менеджерів;

– Вибрано технології для вивчення та застосування при розробці веб-додатка.

В другому розділі кваліфікаційної роботи розроблено сам «Файловий менеджер» та вивчено і використано такі технології: HTML5, CSS3, JavaScript як мови розмітки та програмування; середовище Node.js для виконання серверного JS, React – бібліотека для інтерфейсів користувача; WebPack – плагін-компонувальник для клієнтської частини коду; MySQL – СКБД, яка необхідна для зберігання даних користувачів; Docker – інструмент для контейнеризації програмних продуктів, який використовується для швидкого розроблення, тестування та розгортання застосунків.

У розділі «Безпека життєдіяльності, основи хорони праці» розглянуто важливість центральної нервової системи людини для будь-якої діяльності, а зокрема, при трудовій діяльності. Також досліджено вплив шуму на організм людини та способи зменшення його до рекомендованих меж.

ПЕРЕЛІК ДЖЕРЕЛ

1. What Are Web Services? Easy-to-Learn Concepts with Examples [Електронний ресурс] – Режим доступу: https://www.cleo.com/blog/knowledge-base-web-services – Дата доступу: 23.04.2021 – Назва з екрану.

2. Internet [Електронний ресурс] - Режим доступу: https://en.wikipedia.org/wiki/Internet – Дата доступу: 23.04.2021 – Назва з екрану.

3. Arpanet [Електронний ресурс] – Режим доступу: https://developer.mozilla.org/en-US/docs/Glossary/Arpanet – Дата доступу: 23.04.2021 – Назва з екрану.

4. Information Management: A Proposal [Електронний ресурс] – Режим доступу: https://www.w3.org/History/1989/proposal.html – Дата доступу: 23.04.2021 – Назва з екрану.

5. About W3C [Електронний ресурс] – Режим доступу: https://www.w3.org/Consortium/ – Дата доступу: 02.05.2021 – Назва з екрану.

6. HTML Living Standard [Електронний ресурс] – Режим доступу: https://html.spec.whatwg.org/multipage/ – Дата доступу: 02.05.2021 – Назва з екрану.

7. Don’t break the web: Why web standards matter and how to use them responsibly. [Електронний ресурс] – Режим доступу: https://channel9.msdn.com/Events/Build/2012/3-136 – Дата доступу: 02.05.2021 – Назва з екрану

8. What is a Client? What is a Server? And What is a Host?. [Електронний ресурс] – Режим доступу: https://learntomato.flashrouters.com/what-is-a-client-what-is-a-server-what-is-a-host/ – Дата доступу: 02.05.2021 – Назва з екрану

9. Сім кращих браузерів у прямому порівнянні. [Електронний ресурс] – Режим доступу: https://www.mozilla.org/uk/firefox/browsers/compare/ – Дата доступу: 05.05.2021 – Назва з екрану

10. Basics of HTTP. [Електронний ресурс] – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP – Дата доступу: 05.05.2021 – Назва з екрану

11. Що таке Back-end? Розбираємось із черкаськими розробниками. [Електронний ресурс] – Режим доступу: https://18000.com.ua/strichka-novin/shho-take-back-end-rozbirayemos-iz-cherkaskimi-rozrobnikami/ – Дата доступу: 03.06.2021 – Назва з екрану

12. Google Drive Vs. OneDrive Vs. Dropbox. [Електронний ресурс] – Режим доступу: https://www.cloudally.com/blog/googledrive-onedrive-dropbox-box/ – Дата доступу: 09.05.2021 – Назва з екрану

13. Python vs. Ruby vs. Node.js – Which Platform Is a Fit for Your Project?. [Електронний ресурс] – Режим доступу: https://railsware.com/blog/python-vs-ruby-vs-node-js-which-platform-is-a-fit-for-your-project/ – Дата доступу: 03.06.2021 – Назва з екрану

14. Node.js [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Node.js – Дата доступу: 11.05.2021 – Назва з екрану

15. Чем хорош Ruby on Rails и как он ускоряет разработку [Електронний ресурс] – Режим доступу: https://habr.com/ru/company/skillbox/blog/428487/ – Дата доступу: 03.06.2021 – Назва з екрану

16. Почему Django — лучший фреймворк для разработки сайтов [Електронний ресурс] – Режим доступу: https://ru.hexlet.io/blog/posts/pochemu-django-luchshiy-freymvork-dlya-razrabotki-saytov – Дата доступу: 13.05.2021 – Назва з екрану

17. Why the Hell Would You Use Node.js [Електронний ресурс] – Режим доступу: https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e – Дата доступу: 03.06.2021 – Назва з екрану

18. Mehta, Chintan, et al. MySQL 8 Administrator’s Guide: Effective guide to administering high-performance MySQL 8 solutions. Packt Publishing Ltd, 2018.

19. Comparison of the usage statistics of React vs. Vue.js vs. Angular for websites [Електронний ресурс] – Режим доступу: https://w3techs.com/technologies/comparison/js-angularjs,js-react,js-vuejs – Дата доступу: 03.06.2021 – Назва з екрану

20. Что такое Docker и как его использовать в разработке [Електронний ресурс] – Режим доступу: https://eternalhost.net/blog/razrabotka/chto-takoe-docker – Дата доступу: 03.06.2021 – Назва з екрану

21. Dockerfile reference [Електронний ресурс] – Режим доступу: https://docs.docker.com/engine/reference/builder/ – Дата доступу: 04.06.2021 – Назва з екрану

22. Overview of Docker Compose [Електронний ресурс] – Режим доступу: https://docs.docker.com/compose/ – Дата доступу: 04.06.2021 – Назва з екрану

23. Npm init [Електронний ресурс] – Режим доступу: https://www.codecademy.com/courses/introduction-to-javascript/lessons/browser-compatibility-and-transpilation/exercises/npm-init – Дата доступу: 05.06.2021 – Назва з екрану

24. Вступ до JSX [Електронний ресурс] – Режим доступу: https://uk.reactjs.org/docs/introducing-jsx.html – Дата доступу: 05.06.2021 – Назва з екрану

25. Компоненти і пропси [Електронний ресурс] – Режим доступу: https://uk.reactjs.org/docs/components-and-props.html – Дата доступу: 05.06.2021 – Назва з екрану

26. Active Record Migrations [Електронний ресурс] – Режим доступу: https://guides.rubyonrails.org/active_record_migrations.html– Дата доступу: 03.06.2021 – Назва з екрану

27. Тэйлор Р. Шум. / Р. Тэйлор — М.: Мир, 1978.— 308 с

28. Жидецький В. Ц. Основи охорони праці / В. Ц. Жидецький. — Л. : Афіша, 2005. — 349 с

ДОДАТКИ

Додаток А

Вихідний код основних модулів веб-сервісу «Файловий менеджер»

Файл src/server/API/getObjects.js

```
import express from 'express';
const objects = express.Router();
import { auth } from '../lib/authentication/index.js';
import object from '../Models/object.js';
import sharedFile from '../Models/sharedFile.js';
import { host } from "../config/config";

objects.post('/objects', auth().authenticate(), async (req, res) => {
  try {
    const data = {};

    let allObjects = await object.getObjectsSingleLvl(
      //add all objects that is in one folder
      req.body.parentId,
      req.user.id
    );
    const userList = await sharedFile.getUserList(req.user.id);
    let shareUsers = {};

    /**
     * userList is the list of objects
     * each obj has {id: 0, email: a@a.a} structure
     * below I set shareUsers with key - id and value - email
     * if there are no emails in key, I create an array,
     * otherwise add email to existing array
     */
    for (const obj of userList) {
      shareUsers[obj.id]         =         shareUsers[obj.id]         ?
[...shareUsers[obj.id], obj.email]: [obj.email];
    }
    /**
     * modify items
     * add link and shareUsers lists
     */
    allObjects = allObjects.map((item) => {
      if (item.type === 'file') {
        item.sharable_link = {
          host: host + '/file/',
          path: item.sharable_link,
        };
        item.accessList = shareUsers[item.id]
      }
      return item;
    });
    /**
     *  add all shared files
     */
    let sharedFiles  = await sharedFile.getSharedObjects(req.user.id);
```

```
    sharedFiles = sharedFiles.map((item) => {
      item.type = 'sharedFile';
      return item;
    });
    console.dir(
      sharedFiles
    )

    data.dataObjects = allObjects.concat(sharedFiles);

    if (parseInt(req.body.parentId) !== -1) {
      data.objectInfo = await object.getOne({ id: req.body.parentId });
    } else {
      data.objectInfo = {};
    }

    res.json(data);
  } catch (error) {
    console.log(error);
    res.json('error');
  }
});

objects.get('/sharedFile/:hash', async (req, res) => {
  const hash = req.params.hash;
  try {
    if (hash.length > 10) {
      const owner = await object.getOne({ sharable_link: hash });
      res.download(
        './user_files/' + owner.user_id + '/' + owner.id,
        owner.name
      );
    } else {
      res.json('Check the link');
    }
  } catch (error) {
    console.log(error);
    res.json('Check the link');
  }
});

objects.post('/sharedFile', async (req, res) => {
  try {
    const hash = req.body.hash;

    const file = await object.getOne({ sharable_link: hash });
    // console.log(file);
    res.json({ error: null, file });
  } catch (error) {
    console.log(error);
    res.json({ error: true, message: error });
  }
});

export default objects;
```

Файл src/server/API/loginSignUpRoute.js

```
import express from 'express';
import jsonschema from 'express-jsonschema';
import {
  registerSchema,
  loginSchema,
  emailSchema,
  mailSecret,
} from '../config/config.js';
import sendEmail from '../lib/Email/index.js';
import { auth, generateToken } from '../lib/authentication/index.js';
// config passport: initialization, authentication, token-generating
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';

const loginSignup = express.Router();
const validate = jsonschema.validate;
const saltRounds = 10; // for bcrypt

import user from '../Models/user.js';

loginSignup.post(
  '/login',
  validate({ body: loginSchema }),
  async (req, res) => {
    try {
      const data = await user.getOne({ email: req.body.email });

      if (data.verified) {
        try {
          const comparedPass = await bcrypt.compare(
            req.body.password,
            data.password
          );
          if (comparedPass) {
            const payload = {
              id: data.user_id,
            };
            const token = generateToken(payload);
            res.cookie('JWT', token);
            res.json({ redirect: '/', success: true });
          } else {
            res.json({  message:  'Username  or  password  incorrect',
success: false });
          }
        } catch (e) {
          console.log('comparing error', e);
        }
      } else {
        res.json({ message: 'Username incorrect or you need to confirm
your email' });
      }
    } catch (e) {
      if (e === 'bad statement') {
```

```
        res.json({ message: 'Username or password incorrect' });
      } else {
        res.json({ message: 'Unknown error' });
        console.log(e);
      }
    }
  }
);

loginSignup.post(
  '/register',
  validate({ body: registerSchema }),
  async (req, res) => {
    try {
      const  hashedPassword  =  await  bcrypt.hash(req.body.password,
saltRounds);
      try {
        /**
         * register user
         */
        const data = await user.add({
          name: req.body.name,
          email: req.body.email,
          password: hashedPassword,
          verified: true,
        });
        /**
         * send email
         */
        sendEmail('Register', { id: data[0], receiver: req.body.email,
name: req.body.name });

        res.json({ redirect: '/', success: true, message: data });
      } catch (e) {
        console.log(e);
        let message = '';
        switch (e.errno) {
        case 1062:
          message = 'The user is already exists';
          break;
        default:
          message = 'Unknown error. Try again';
          break;
        }
        res.json({ redirect: '/register', success: false, message });
      }
    } catch (e) {
      console.error(e);
      res.json({
        redirect: '/register',
        success: false,
        message: 'Internal error',
      });
    }
  }
);
```

```
loginSignup.get('/confirmation/:token', async (req, res) => {
  try {
    const { id } = jwt.verify(req.params.token, mailSecret);
    if (id) {
      user.updateVerification(true, id);
      // res.json({ message: 'ok', redirect: '/' });
      res.redirect('/');
    }
  } catch (error) {
    console.log(error);
    res.send('error');
  }
});

loginSignup.post(
  '/forgot-password',
  validate({ body: emailSchema }),
  async (req, res) => {
    const email = req.body.email;
    sendEmail('Forgot password', { id: email, receiver: email });
    res.json('The email with further instructions was sent to you');
  }
);

loginSignup.post('/reset-password/:token', async (req, res) => {
  const token = req.params.token;
  try {
    const { id: email } = jwt.verify(token, mailSecret);
    if (email && req.body.password) {
      const  hashedPassword  =  await  bcrypt.hash(req.body.password,
saltRounds);
      user.updatePassword(email, hashedPassword);
      res.json('Your password was successfully changed.');
    } else {
      res.json('Email or password undefined.');
    }
  } catch (error) {
    console.log(error);
    res.json('Error.');
  }
});

loginSignup.post('/auth', auth().authenticate(), (req, res) => {
  res.json({ data: req.user });
});

loginSignup.post('/logout', auth().authenticate(), (req, res) => {
  res.cookie('JWT', '');
  res.json({ redirect: '/login' });
});

export default loginSignup;
```

Файл src/server/API/loginSignUpRoute.js

```
import express from 'express';
import multer from 'multer';
import { createFolder, isExist, moveFile, deleteFile } from
'../lib/FileSystem';
import { auth } from '../lib/authentication/index.js'; // config
passport: initialization, authentication, token-generating
import object from '../Models/object.js';
import { host } from '../config/config';
import sharedFiles from '../Models/sharedFile';
import user from '../Models/user';
const objects = express.Router();
const storage = multer.diskStorage({});
const upload = multer({ storage });
const crypto = require('crypto');

// midleware for creating a folder if not exist
objects.use('/file', auth().authenticate(), async (req, res, next) =>
{
  isExist('./user_files/' + req.user.id).catch(() => {
    // console.log(error); -> folder doesn't exist, so we'll create one
    createFolder('./user_files/' + req.user.id)
      .then((res) => {
        if (res) {
          // console.log('the folder created');
        }
      })
      .catch((error) => {
        console.log(error);
      });
  });
  next();
});

objects.post('/file', upload.single('files'), async (req, res) => {
  try {
    const total = await object.getTotalSize(req.user.id);
    const limit = await user.getOneWithColumns(
      { user_id: req.user.id },
      'space_limit'
    );
    if (total[0].total + req.file.size > limit.space_limit) {
      res.json({ error: true, message: 'You exceeded your space limit'
});
    } else {
      const id = await object.addOne({
        type: 'file',
        parent_id: req.body.parent_id,
        name: req.file.originalname,
        user_id: req.user.id,
        file_size: req.file.size,
      });

      moveFile(req.file.path, './user_files/' + req.user.id, id[0])
        .then(() => {
          res.json('File uploaded!');
        })
        .catch((err) => {
```

```
          res.json('Internal error');
          console.log(err);
        });
    }
  } catch (error) {
    console.log(error);
    res.json('Internal error');
  }
});

objects.post(
  '/folder',
  express.json(),
  auth().authenticate(),
  async (req, res) => {
    try {
      const id = await object.addOne({
        type: 'folder',
        parent_id: req.body.parentId,
        name: req.body.name,
        user_id: req.user.id,
      });
      if (id) {
        res.json('folder successfully created!');
      } else {
        res.json('error, can`t create a folder');
      }
    } catch (error) {
      console.log(error);
      res.json('error');
    }
  }
);

objects.post(
  '/sharedFiles/add',
  express.json(),
  auth().authenticate(),
  async (req, res) => {
    //console.log('ok');

    try {
      const id = await user.getOne({ email: req.body.email });
      // console.log({email: req.body.email, user: req.user.id, touser:
id.user_id })
      await sharedFiles.add({
        user_id: req.user.id,
        object_id: req.body.objectId,
        to_user_id: id.user_id,
      });
      res.json({ success: true });
    } catch (error) {
      console.error(error);
      res.json({ success: false, message: 'try again' });
    }
  }
);
```

```
objects.post(
  '/sharedFiles/delete',
  express.json(),
  auth().authenticate(),
  async (req, res) => {
    try {
      const id = await user.getOne({ email: req.body.email });
      // console.log(id);
      // console.log({email: req.body.email, user: req.user.id, touser:
id.user_id })
      await                sharedFiles.deleteUserFromList(id.user_id,
req.body.fileId);
      console.log('deleted');
      res.json({ success: true });
    } catch (error) {
      console.error(error);
      res.json({ success: false, message: 'try again' });
    }
  }
);

objects.post(
  '/sharedFiles/toogleSharable',
  express.json(),
  auth().authenticate(),
  async (req, res) => {
    try {
      const shareable = req.body.switchOn;
      const fileId = req.body.fileId;

      if (shareable) {
        crypto.randomBytes(20, (err, buf) => {
          if (err) throw err;
          //using model alter the record of the file, add a link
          object.updateLink(buf.toString('hex'), fileId);
          // console.log(
          //           `${buf.length}    bytes    of    random    data:
${buf.toString('hex')}`
          // );

          res.json({
            host: host + '/file/',
            path: buf.toString('hex'),
          });
        });
      } else {
        // make it empty
        object.updateLink('', fileId);
        res.json({ host: '', path: '' });
      }
    } catch (error) {
      res.json('error');
      console.log(error);
    }
  }
);
```

```
objects.post(
  '/file/delete',
  express.json(),
  auth().authenticate(),
  async (req, res) => {
    //console.log('ok');
    const fileId = req.body.id;
    try {
      object.deleteById(fileId); //delete objects
      //delete from filesystem
      deleteFile('./user_files/'    +    req.user.id    +    '/'    +
fileId).catch(err =>
        console.log(err)
      );
      sharedFiles.deleteByFileId(fileId); //delete from shared_files
      res.json({ success: true });
    } catch (error) {
      console.error(error);
      res.json({ success: false, message: 'try again' });
    }
  }
);

/**
 * deletes folder and its content
 * if file - delete normally
 * if folder - call recursively and delete files and folder itself
 */
const deleteFolderAndContent = async (id, userId) => {
  try {
    const allObjects = await object.getAllByParent(id); // get all
objects folder has
    object.deleteById(id); // delete folder

    allObjects.forEach(obj => {
      if (obj.type === 'file') { // for files just delete as in file
controller
        object.deleteById(obj.id);
        deleteFile('./user_files/' + userId + '/' + obj.id).catch(err
=>
          console.log(err)
        );
        sharedFiles.deleteByFileId(obj.id);
      } else { // otherwise, it's a folder, so we call this function
recursively
        deleteFolderAndContent(obj.id, userId);
        object.deleteById(obj.id);
      }
    });
  } catch (error) {
    console.log(error);
  }


};
objects.post(
```

```
  '/folder/delete',
  express.json(),
  auth().authenticate(),
  async (req, res) => {
    try {
      deleteFolderAndContent(req.body.id, req.user.id);
      res.json({ success: true });
    } catch (error) {
      console.error(error);
      res.json({ success: false, message: 'try again' });
    }
  }
);


objects.post(
  '/spaceInfo',
  express.json(),
  auth().authenticate(),
  async (req, res) => {
    try {
      const total = await object.getTotalSize(req.user.id);
      const limit = await user.getOneWithColumns(
        { user_id: req.user.id },
        'space_limit'
      );
      res.json({
        taken: total[0].total,
        max: limit.space_limit * 1,
      });
    } catch (error) {
      res.json('error');
      console.log(error);
    }
  }
);

export default objects;
```

Файл src/server/config/config.js

```
import dotenv from 'dotenv';
import path from 'path';

dotenv.config({ path: path.resolve(__dirname, '../../.env') });

const jwtSecret = process.env.SECRET_KEY;

const cookieExtractor = (req) => {
  let token = null;
  if (req && req.cookies) {
    token = req.cookies['JWT'];
  }
  return token;
};
```

```
const jwtOptions = {
  jwtFromRequest: cookieExtractor,
  secretOrKey: jwtSecret,
};

/*
  validation schemas start
*/

const registerSchema = {
  type: 'object',
  properties: {
    name: {
      type: 'string',
      required: true,
    },
    email: {
      type: 'string',
      pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/,
      required: true,
    },
    password: {
      type: 'string',
      required: true,
    },
  },
};

const loginSchema = {
  type: 'object',
  properties: {
    email: {
      type: 'string',
      pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/,
      required: true,
    },
    password: {
      type: 'string',
      required: true,
    },
  },
};

const emailSchema = {
  type: 'object',
  properties: {
    email: {
      type: 'string',
      pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/,
      required: true,
    },
  },
};

/*
  validation schemas end
```

```
*/

/*

  validation response function START

*/
const validationResponse = (err, req, res, next) => {
  let responseData;

  if (err.name === 'JsonSchemaValidation') {
    // Log the error however you please
    console.log(err.message);
    // logs "express-jsonschema: Invalid data found"

    // Set a bad request http response status or whatever you want
    res.status(400);

    // Format the response body however you want
    responseData = {
      statusText: 'Bad Request',
      jsonSchemaValidation: true,
      validations:  err.validations,  //  All  of  your  validation
information
    };

    // Take into account the content type if your app serves various
content types

    res.json(responseData);
  } else {
    // pass error to next error middleware handler
    next(err);
  }
};

/*
  validation response function END
*/

const host = 'http://file-man.tk';

import knex from 'knex';
export default knex({
  client: 'mysql',
  connection: {
    host: process.env.MYSQL_HOST || '127.0.0.1',
    user: process.env.MYSQL_USER || 'root',
    password: process.env.DB_PASS || '',
    database: process.env.MYSQL_DB || 'file_manager',
  },
});

const mailSecret = process.env.MAIL_SECRET;
const email = process.env.MAIL;

import sgMail from '@sendgrid/mail';
```

```
sgMail.setApiKey(process.env.SENDGRID_API);



export {
  jwtSecret,
  jwtOptions,
  loginSchema,
  registerSchema,
  emailSchema,
  mailSecret,
  sgMail,
  email,
  host,
  validationResponse,
};
```

Файл client\src\pages\AuthPage\index.jsx

```
import React from 'react';
import Auth from '../../domains/Auth';
import { Route } from 'react-router-dom/cjs/react-router-dom.min';
import PageWrapper from '../../components/PageWrapper';
import Forgot from '../../domains/Auth/Forgot';
import Reset from '../../domains/Auth/Reset';

export default function AuthPage() {
  return (
    <PageWrapper>
      <Route path={`/login`} component={(props) => <Auth isLogin={true}
/>} />
      <Route
        path={`/register`}
        component={(props) => <Auth isLogin={false} />}
      />
      <Route path={`/forgot-password`} component={() => <Forgot />} />
      <Route path={`/reset-password/:token`} children={<Reset />} />
    </PageWrapper>
      );
}
```

Файл client\src\domains\Home\MainContainer\FileManager\FilesBody\ index.js

```
import React, { useState } from 'react';
import './style.css';
import File from '../../../../../components/File';
import Folder from '../../../../../components/Folder';
import Loader from '../../../../../components/Loader';
import ModalConnect from '../ModalConnect';
import withData from '../../../../../lib/withData';

const FilesBody = (props) => {
```

```
  const [selected, setSelected] = useState([]);

  const objects = props.data.dataObjects.map((object, index) => {
    object.selected = false;
    object.index = index;
    return object;
  });

  if (objects.length > 0 && selected.length === 0) {
    setSelected(objects.map((object) => object.selected));
  }
  let files;
  let folders;
  let sharedFiles;

  if (objects.length > 0) {
    files = objects.filter((file) => file.type === 'file');
    folders = objects.filter((folder) => folder.type === 'folder');
    sharedFiles = objects.filter((sharedFile) => sharedFile.type ===
'sharedFile');

    return (
      <div className="files-body">
        <ModalConnect
          displayLevel={props.displayLevel}
          setDisplayLevel={props.setDisplayLevel}
        />

        <div className="light-header-text">Folders</div>
        <Folders
          folders={folders}
          displayLevel={props.displayLevel}
          setDisplayLevel={props.setDisplayLevel}
          objectInfo={props.data.objectInfo}
          selected={selected}
          setSelected={setSelected}
          setFileInfo={props.setFileInfo}
        />
        <div className="light-header-text">Files</div>
        <Files
          files={files}
          selected={selected}
          setSelected={setSelected}
          setFileInfo={props.setFileInfo}
          setDisplayLevel={props.setDisplayLevel}
        />
        <div className='light-header-text'>Shared files</div>
        <Files
          files={sharedFiles}
          selected={selected}
          setSelected={setSelected}
          setFileInfo={props.setFileInfo}
        />
      </div>
    );
  }
  return props.displayLevel === -1 ? (
```

```
    <div>
      There are no files
      <ModalConnect
        displayLevel={props.displayLevel}
        setDisplayLevel={props.setDisplayLevel}
      />
    </div>
  ) : (
    <React.Fragment>
      <div className="files-body">
        <ModalConnect
          displayLevel={props.displayLevel}
          setDisplayLevel={props.setDisplayLevel}
        />

        <div className="light-header-text">Folders</div>
        <Folder
          key={props.data.objectInfo.id}
          folder={{ name: '..' }}
          selected={props.selected}
          setSelected={props.setSelected}
          setDisplayLevel={props.setDisplayLevel}
          index={-1}
          onDoubleClick={() =>
            props.setDisplayLevel({lvl:
props.data.objectInfo.parent_id})
          }
          setFileInfo={props.setFileInfo}
        />
        {/* <div className="light-header-text">Files</div> */}
      </div>
    </React.Fragment>
  );

  //return <Loader />;
};

const Files = (props) => {
  if (props.files) {
    const listOfFiles = props.files.map((file) => {
      return (
        <File
          key={file.id}
          file={file}
          setDisplayLevel={props.setDisplayLevel}
          selected={props.selected}
          setSelected={props.setSelected}
          index={file.index}
          setFileInfo={props.setFileInfo}
        />
      );
    });
    return listOfFiles;
  }
  return <Loader />;
};
```

```
const Folders = (props) => {
  // console.log({ folders: props });
  const folders = props.folders;
  if (folders) {
    const foldersComponents = folders.map((folder) => {
      return (
        <Folder
          key={folder.id}
          folder={folder}
          onDoubleClick={() => {
            props.setDisplayLevel({lvl: folder.id});
            props.setSelected([]);
          }}
          selected={props.selected}
          setSelected={props.setSelected}
          setDisplayLevel={props.setDisplayLevel}
          index={folder.index}
          setFileInfo={props.setFileInfo}
        />
      );
    });
    if (props.displayLevel !== -1) {
      foldersComponents.unshift(
        <Folder
          key={-1}
          folder={{ name: '..' }}
          selected={props.selected}
          setSelected={props.setSelected}
          setDisplayLevel={props.setDisplayLevel}
          index={-1}
          onDoubleClick={() =>
            props.setDisplayLevel({lvl: props.objectInfo.parent_id})
          }
          setFileInfo={props.setFileInfo}
        />
      );
    }

    return foldersComponents;
  }
  return <Loader />;
};

export default withData(FilesBody);
```

Файл client\src\domains\Home\MainContainer\FileManager\FilesHeader\index.js

```
import React, { useEffect } from 'react';
import './FilesHeader.css';

const FilesHeader = (props) => {
  useEffect(() => {
    window.addEventListener('click', handleExit('fileModal'));
    window.addEventListener('click', handleExit('folderModal'));
```

```
    return () => {
      window.removeEventListener('click', handleExit('fileModal'));
      window.removeEventListener('click', handleExit('folderModal'));
    };
  }, []);

  const handleExit = (modalName) => (event) => {
    const modal = document.getElementById(modalName);
    if (event.target == modal) {
      modal.style.display = 'none';
    }
  };

  return (
    <div className="files-header">
      <div className="heading">
        <h1>Documents</h1>
        <div
          className="add-folder"
          onClick={() => {
            const modal = document.getElementById('folderModal');
            modal.style.display = 'block';
          }}
        >
          <i className="fas fa-folder-plus"></i>
        </div>
        <button
          className="add-files"
          onClick={() => {
            const modal = document.getElementById('fileModal');
            modal.style.display = 'block';
          }}
        >
          + Add files
        </button>
      </div>
      <div className="header-buttons">
        <button className="active-button">some</button>
        <button className="">other</button>
        <button className="">either</button>
      </div>
    </div>
  );
};

export default FilesHeader;
```

Файл client\src\domains\Home\MainContainer\FileManager\FileUpload\index.js

```
import React, { useState } from 'react';
import './fileUpload.css';
import { upload, sendData } from '../../../../../lib/PostData';

const FileUpload = (props) => {
```

```
  const [fileUpload, setFileUpload] = useState(null);
  function onFileChange(e) {
    setFileUpload([...e.target.files]);
  }

  function onSubmit(e) {
    e.preventDefault();

    if (fileUpload) {
      let formData = new FormData();
      for (const key of Object.keys(fileUpload)) {
        formData.append('files', fileUpload[key]);
      }
      formData.append('parent_id', props.displayLevel);
      upload(formData, '/api/file')
        .then((response) => {
          alert(response);
          props.setDisplayLevel((prev) => ({ ...prev })); // re-render
to fetch new files
        })
        .catch((error) => {
          console.log(error);
        });
    }
    setFileUpload(null);
  }
  function fileList() {
    const array = fileUpload;
    if (array) {
      //console.log(array);
      return array.map((file) => {
        return <li key={file.size}>{file.name || 'select a file'}</li>;
      });
    }    return <li>select a file</li>;
  }  return (
    <React.Fragment>
      <form onSubmit={onSubmit}>
        <div className="upload-btn-wrapper">
          <button className="btn">
            Select a file
            <input  type="file"  name="myfile"  onChange={onFileChange}
/>
          </button>
          <button className="btn" type="submit">
            Upload a file
          </button>
        </div>
      </form>
      <div className="your-files">
        Files to upload: <br />
        <ul>{fileList()}</ul>
      </div>
    </React.Fragment>
  );
};

export default FileUpload;
```