МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ВЕБ ДОДАТКУ «ХМАРНЕ СХОВИЩЕ» ЗАСОБАМИ REACTJS ТА GRAPHQL»**

Виконав: студент 2 курсу, групи 8.1210-з
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
М.М. Славенко
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Мухін В.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної математики, доцент, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
«ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент
_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2021 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Славенко Марку Михайловичу
(прізвище, ім’я та по-батькові)

1. Тема роботи (проекту) Розробка веб додатку «Хмарне сховище» засобами ReactJS та GraphQL

керівник роботи (проекту) Мухін Віталій Вікторович, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від «09» 06 2021 року № 850-с

2. Строк подання студентом роботи 25.11.2021

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області.
2. Проектування веб додатку.
3. Розробка веб додатку.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
Презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Мухін В. В., к.т.н., доцент	01.09.2021	25.09.2021
2	Мухін В. В., к.т.н., доцент	25.09.2021	15.10.2021
3	Мухін В. В., к.т.н., доцент	15.10.2021	30.10.2021
Додатки	Мухін В. В., к.т.н., доцент	30.10.2021	10.11.2021

7. Дата видачі завдання 01.08.2021

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	01.08.2021	
2.	Збір вихідних даних.	01.08.2021	
3.	Обробка методичних та теоретичних	20.08.2021	
	джерел.		
4.	Розробка першого та другого розділів.	01.09.2021	
5.	Розробка третього розділу.	15.10.2021	
6.	Оформлення та нормоконтроль		
	кваліфікаційної роботи.	15.11.2021	
7.	Захист кваліфікаційної роботи.	09.12.2021	

Студент _____ М.М.Славенко
(підпис) (ініціали та прізвище)

Керівник роботи _____ В.В. Мухін
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ С. П. Швидка
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка веб додатку «Хмарне сховище» засобами ReactJS та GraphQL»: 42 с., 12 рис., 5 джерел, 2 додатки.

GRAPHQL, ES6, JAVASCRIPT, REACT.JS, SPA, ХМАРНЕ СХОВИЩЕ, ВЕБ ДОДАТОК, MINIO, APOLLO GRAPHQL, PRISMA.

Об’єкт дослідження – процес розробки веб додатку з використанням бібліотеки ReactJS та GraphQL.

Мета роботи – розробка веб додатку хмарного сховища.

Метод дослідження – методи збору та аналізу вимог до програмного забезпечення, методи моделювання та проектування програмного забезпечення.

У результаті виконання кваліфікаційної роботи отримано веб додаток хмарного сховища з використанням GraphQL. Окрім цієї мови запитів для розробки Front-end частини, використовувався фреймворк ReactJS. Також використовувалися Apollo GraphQL, який значно розширює можливості GraphQL, та MinIO для організації сховища. Усе це дозволило розробити цікавий веб додаток зі зручним інтерфейсом та широким набіром функціоналу. А також зробити розробку гнучкою, а подальша підтримка додатку легко масштабованою.

SUMMARY

Master`s qualifying paper «Development of the “Cloud storage” web application using ReactJS and GraphQL»: 42 pages, 12 figures, 5 references, 2 supplements.

GRAPHQL, ES6, JAVASCRIPT, REACT.JS, SPA, CLOUD STORAGE, WEB APP, MINIO, APOLLO GRAPHQL, PRISMA.

The object of the study is the software development process using the ReactJS and GraphQL library.

The aim of the study is development of cloud storage web app.

The methods of research are collecting and analyzing software requirements, methods of modeling and designing software.

As As a result of qualification work, a cloud storage web application with using GraphQL was received. In addition to this query language, the ReactJS framework was used to develop the Front-end part. Apollo GraphQL, which greatly expands the capabilities of GraphQL, and MinIO were also used to organize the storage. All this allowed us to develop an interesting web application with a user-friendly interface and a wide range of functionality. And also it helps to make a development process flexible, and the support of application easily scalable.

ЗМІСТ

ВСТУП

У наш час, еру цифрових технологій інформації стає все більше і більше. А її обсяги та вага збільшуються у геометричній прогресії. Ті мільярди фото, котрі люди роблять кожного дня. Мільйони відео записуються, аби зберегти назавжди у пам’яті важливі життєві моменти. Оцифровуються старі матеріали. Будь які програми також намагаються зберегти свої дані на носіях користувачів. Не дивно, але все частіше і частіше об’єму пам’яті на фізичних носіях стає недостатньо.

Крім самої проблеми обмеженості пам’яті на носіях, з’являються і інші. Наприклад, як забезпечити ту чи іншу інформацію достатнім рівнем захисту та стабільності? Або, наприклад, зробити цю інформацію легко доступною для користувачів у будь-який частині світу?

Відповіддю, а також рішенням усіх цих проблем, стали хмарні сховища. Це такі ж звичайні носії інформації, які можуть знаходитися у вашому комп’ютері, але у дуже дуже великих масштабах. Ці сховища поєднують у собі важкі архітектурні рішення, як і на програмному рівні, так і на рівні технічного забезпечення. Увесь цей комплекс технологій дозволяє давати безперебійний доступ до інформації користувачів з будь-якої точки планети. А також забезпечувати безпеку даних, та захист від шкідливих програм. І звичайно ж, заощаджувати простір на гаджетах користувачів.

Тож було вирішено розробити хмарне сховище, з використанням найсучасніших технології у сфері розробки програмних продуктів.

Це буде веб додаток, Front-end частина котрого буде запрограмована за допомогою бібліотеки ReactJS. За взаємодію з Back-end частиною відповідатиме мова запитів GraphQL. Також будуть використані Apollo GraphQL, щ тісно співпрацює з бібліотекою ReactJS, та дуже спрощує використання GraphQL. А також має серверну частину Apollo Server, що

допомагає розробляти моделі для бази даних, та спрощує отримання цих самих даних з неї.

Окремою частиною буде саме організація сховища, тут на допомогу прийде MinIO – це популярний високопродуктивний сервер зберігання об'єктів з відкритим вихідним кодом. А також зручною клієнт частиною, для взаємодії з сервером.

Мета роботи – розробити веб додаток «Хмарного сховища», з використанням ReactJS та GraphQL.

У кваліфікаційній дипломній роботі поставлено такі завдання:

1) ознайомитися з бібліотекою ReactJS та мовою запитів GraphQL;

2) розробити сучасний веб додаток хмарного сховища з використанням бібліотеки ReactJS та мови запитів GraphQL. Додаток має мати увесь необхідний функціонал, котрий може бути потрібен користувачу для зберігання своїх файлів. А також мати основні можливості популярних хмарних сховищ, таких як наприклад, створення папок, копіювання файлів та можливістю ділитися файлами з іншими користувачами.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГИ

1.1 Теоретичний аналіз аналогів

Для початку давайте розглянемо популярних конкурентів в сфері хмарних сховищ. Сьогодні дуже багато компаній пропонують свої послуги, але давайте розглянме два найбільш популярних хмарних сховища:

— Google Drive (drive.google.com);

— один із найстаріших та найпопулярніших сервісів хмарного зберігання Dropbox (www.dropbox.com).

Порівняльна характеристика:

Google Drive [1].

Розробка – Google Inc.

Плюси:

— є частиною системи Google, що забезпечує тісну інтеграцію з іншими сервісами;

— можливість створювати та редагувати деякі типи файлів прямо на сайті;

— має історію змінення файлів;

— гнучка система для спільної роботи.

Мінуси:

— тісна взаємодія з іншими сервісами Google може бути також і мінусом;

— пам’ять для пошти та сховища спільна, тож електронна пошта буде займати простір диску;

— перевантажений інтерфейс.

Dropbox.

Розробка – Dropbox Inc.

Плюси:

— має найбільший вибір різноманітних сервісів для інтеграції (Trello, Slack, Zoom тощо);

— має реферальну програму, що дозволяє отримати безкоштовне сховище;

— більшість функцій які є і у сховищі від Google.

Мінуси:

— безкоштовний функціонал дуже обмежений;

— безкоштовний об’єм пам’яті малий, усього 2 гігабайти.

Розроблене хмарне сховище.

Плюси:

— простий та зручний інтерфейс, не перевантажений великою кількістю надлишкового функционалу;

— повний набір головного функціоналу (папки, створення файлів, копіювання та інше);

— гнучкий у налаштуванні, завдяки використанню сучасних технологій при розробці.

Мінуси:

— немає інтеграції з сервісами, оскільки це потребує багато часу для розробки.

— немає можливості одночасного редагування файлів.

1.2 Призначення та цілі створення

Функціональне призначення системи – у результаті аналізу та згідно поставленої задачі результатом кваліфікаційної роботи має бути розроблений веб додаток хмарного сховища з використанням бібліотеки ReactJS та мови запитів GraphQL.

Експлуатаційне призначення системи – у результаті розробки буде отримано сучасний веб додаток. З використанням такого стеку технологій як:

ReactJS, GraphQL, Apollo, MinIO, Prisma. Ця робота, може стати гарною структурою для майбутньої розробки комерційного проекту. Або ж прикладом для розробників схожої системи.

Мета створення системи – розробити веб додаток з використанням сучасних технологій. Та на основі цього отримати знання та навички роботи з ReactJS та GraphQL.

1.3 Вимоги

Додаток не матиме спеціальних типів користувачів, лише звичайного юзера. Тож розглянемо основні вимоги від веб додатку для користувача хмарного сховища:

— можливість створення файлів, їх копіювання, видалення, переміщення у папки, перейменування;

— можливість створення папок, їх переміщення, перейменування;

— отримання інформації про той чи інший файл;

— можливість ділитися доступом до файлів з іншими користувачами сховища;

— базові можливості редагування профілю.

Серед технічних вимог які застосовуються до майбутнього веб додатку, можна виділити:

— кроссбраузерність;

— кроссплатформенність;

— захищеність;

— швидкість роботи.

1.4 Програмні засоби

Із теми роботи, повністю зрозуміло, на якій технології буде створюватися Front-end частина хмарного сховища – React.js, у поєднанні з ES6, це дуже потужний інструмент. Для взаємодії з Back-end частиною та збереження файлів будуть використовуватися GraphQL та Apollo відповідно. Тож давайте розглянемо ці технології ближче.

React.js – бібліотека, або фреймворк, підтримуваний компанією Facebook, який набрав величезну популярність в останні роки як один із найкращих JavaScript фреймворків, для побудування веб додатків. Має багато переваг, таких як віртуальний DOM, JSX-розмітка та різноманітні методи життєвого циклу. А завдяки модульній системі з можливостям імпорту і експорту, розробка стане набагато легше і приємніше [3].

ECMAScript (або ES) – специфікація мови сценаріїв, стандартизована Ecma International. Була створена для стандартизації JavaScript для сприяння створенню декількох незалежних реалізацій. JavaScript залишається найбільш широко використовуваною реалізацією ECMAScript з моменту вперше опублікування стандарту з іншими реалізаціями, включаючи JScript та ActionScript. ECMAScript зазвичай використовується для сценаріїв на стороні клієнта у всесвітній павутині, і все частіше використовується для написання серверних програм та служб за допомогою Node.js [4].

GraphQL - це мова з відкритим вихідним кодом запитів та маніпуляцій для API, а також середовище виконання для виконання запитів з наявними даними. Ця мова забезпечує підхід до розробки веб-API та порівнюється з REST та іншими архітектурами веб-сервісів та протиставляється їм. Вона дозволяє клієнтам визначати структуру необхідних даних, і та ж структура даних повертається з сервера, що запобігає поверненню занадто великих обсягів даних, але це впливає на ефективність веб-кешування результатів запиту може бути. Гнучкість та багатство мови запитів також додають складності, які можуть не мати сенсу для найпростіших API. Вона

складається з системи типів, мови запитів та семантики виконання, статичної перевірки та самоаналізу типу [5].

MinIO — це високопродуктивне сховище об’єктів, випущене під ліцензією GNU Affero General Public License v3.0. Воно сумісне із API із хмарним сховищем Amazon S3. Може обробляти неструктуровані дані, такі як фотографії, відео, файли журналів, резервні копії та зображення контейнерів з (наразі) максимальним підтримуваним розміром об’єкта 5 ТБ [7].

Система Apollo Graphql побудована на основі клієнта та сервера GraphQL з відкритим вихідним кодом. Платформа надає інструменти та послуги розробника для прискорення розробки, захисту інфраструктури та масштабування графіка даних для кількох команд. Метою Apollo є надання єдиного джерела інформації для структури графіка шляхом відстеження схеми в центральному реєстрі. Реєстр схем діє як центр, який координує всі системи та інструменти, які утворюють графік, і зберігає схему приватною.

2 ПРОЕКТУВАННЯ

2.1 Проектування сутностей веб додатку

Проектування Back-end логіки починається з розгляду потрібних у веб додатку сутностей. А потім більш детального опису кожної із них, та того які точки доступу, або взаємодії будуть потрібні для роботи веб додатку.

Перша і можливо найважливіша сутність додатку це користувач. Саме він є основною одиницею взаємодії із додатком. Звичайно опис сутності користувача є нетривіальним завданням, оскільки часто потрібні поля не сильно відрізняються від проекту до проекту. Але все ж таки, користувач матиме такі поля: унікальний ідентифікаційний номер, ім'я, фамілія, аватар, електронна адреса, пароль.

Наступною, основною сутністю хмарного сховища, є сутність файлу. Ця сутність має нести у собі основну інформацію про файл а також мати усі поля для використання у системі. Вона матиме такі поля:

— унікальний ідентифікаційний номер;

— назва файлу, котре буде використовуватися для показу на Front-end частині додатку;

— дату створення;

— дату змінення;

— адрес для доступу у сховище MinIO;

— ID юзера, котрий створив цей файл;

— ID папки в котрій цей файл знаходиться.

Усі ці поля потрібні для функціонування та доступу до файлів створених користувачами.

Останньою необхідною сутністю веб додатку є сутність папки, в котрих будуть зберігатися файли юзерів. Вона матиме чимось схожу структуру з файлом, та матиме такі поля:

— унікальний ідентифікаційний номер;

— назва папки, для відображення на Front-end частині;

— дату створення;

— ID юзера, котрий створив цю папку;

— ID папки в котрій ця папка знаходиться, оскільки хмарне сховище матиме можливість створювати деревоподібну структуру для збереження файлів;

Якщо подивитися на ці сутності (див. рис. 2.1), то може здатися що вони не повні, наприклад не зрозуміло як буде реалізована можливість надання доступу до файлів іншим користувачам. Стає зрозуміло що для цього потрібні сміжні таблиці у базі даних, або так звані PIVOT таблиці. Але усе це буде вирішено можливостями Apollo GraphQL та Prisma.

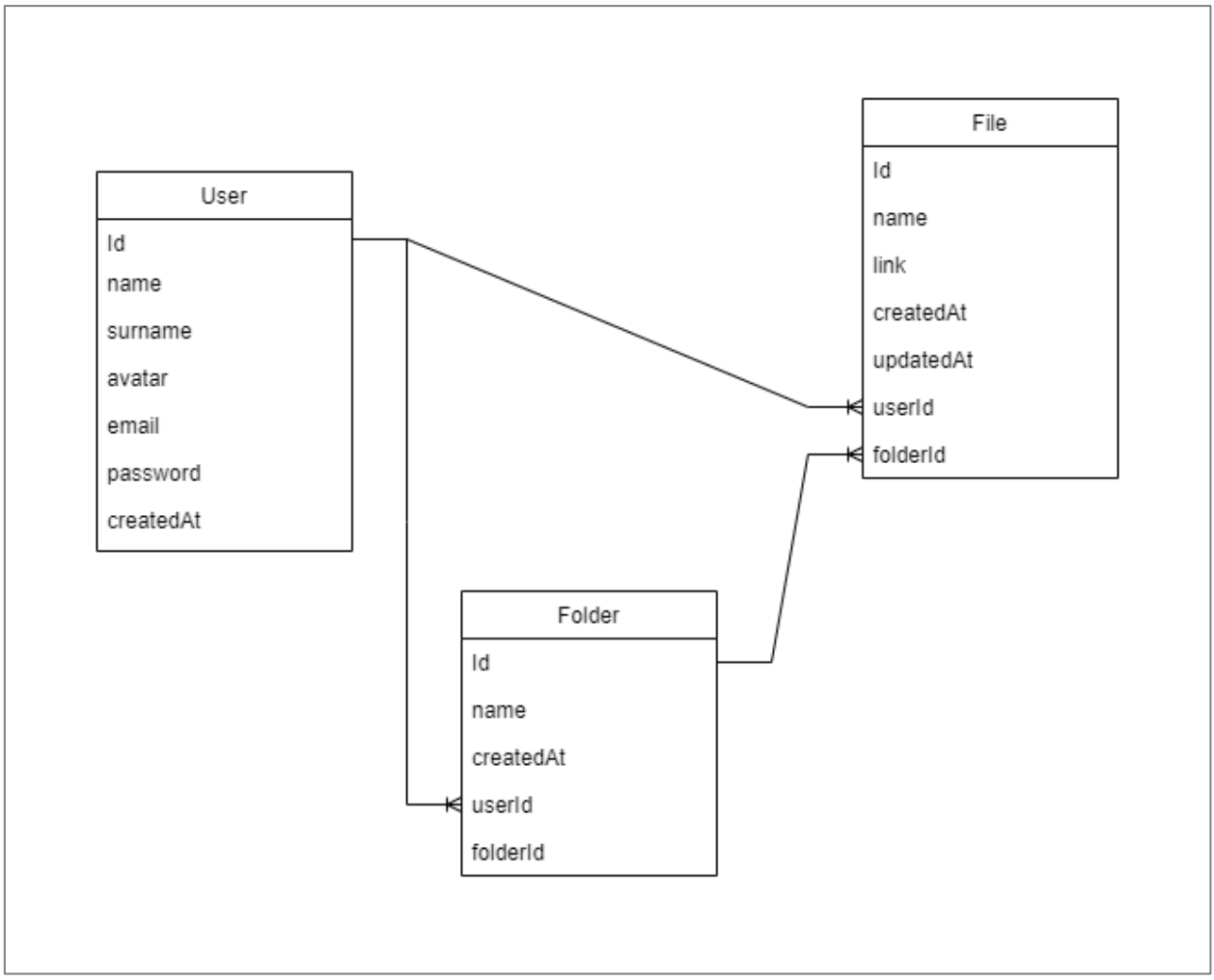

Рисунок 2.1 – Сутності хмарного сховища

2.2 Проектування «ендпоінтів» серверної частини

Наступним етапом буде опис усіх необхідних ендпоінтів серверної частини. Це будуть незвичайні CRUD ендпоінти, оскільки GraphQL та Apollo мають специфічну структуру їх створення та опису. Далі розглянемо їх список для кожної сутності, і для чого потрібен кожен із них.

Ендпоінти для сутності користувача:

— findUsers. Перший ендпоінт для пошуку користувачів за іменем або email, потрібен, наприклад, для пошуку користувача для надання доступу до файлів;

— createUser. Ендпоінт створення юзера при реєстрації нового користувача хмарного сховища;

— saveUser. Цей ендпоінт потрібен для оновлення інформації користувача, наприклад ім’я чи email адреси;

— deleteUser. Для видалення користувача.

Ендпоінти для сутності файлу:

— getFiles. Отримання усіх файлів, чи інформації для відображення, користувача. Створених користувачем, або тих до котрих він має доступ;

— createFile. Завантаження нового файлу до хмарного сховища;

— editFile. Ендпоінт для оновлення даних файлу, наприклад назва файлу для відображення;

— deleteFiles. Видалення файлів;

— shareFile. Ендпоінт для надання доступу до файлу іншим користувачам додатку;

— downloadFiles. Ендпоінт для отримання самих файлів зі сховища.

— changeDirectory. Зміна розташування файлу, або зміна папки. З флагом копіювання або повного переміщення.

Ендпоінти для сутності папки:

— getRootFolders. Отримання усіх папок вищого рівня деревоподібної структури;

— getSubFolders. Отримання папок, що знаходяться у папці, ID котрої передано у параметрах запиту;

— getFolderFiles. Отримання файлів, що знаходяться у папці, ID котрої передано у параметрах запиту;

— getFolderPath. Отримання повного шляху до папки, реалізаця цього ендпоінту буде разглянута окремо;

— createFolder. Створення папки;

— editFolder. Зміна назви;

— deleteFolder. Видалення папки разом із файлами що в ній знаходяться.

— changeDirectory. Зміна розташування папки.

2.3 Проектування Front-end частини.

Як було вирішено раніше, розробка Front-end частини буде здійснюватиметься силами фреймворку ReactJS. Цей фреймворк пропонує компонентний підхід до реалізації поставленого завдання, тому в частині проектування потрібно зрозуміти які компоненти необхідно реалізувати для функціонування хмарного сховища. А також які сторінки повинен мати веб додаток.

Звичайно, оскільки додаток створюється для реальних людей, або користувачів, в першу чергу він має мати у собі сторінки для логіну, реєстрації та редагування свого профілю. Ця частина не буде докладно описуватися, оскільки не несе в собі нічого інноваційного, та є практично в кожному проекті.

Звичайно цікавіше буде розглянути сторінку взаємодії з файлами і папками хмарного сховища. Найзручніше буде реалізувати цю сторінку на зразок звичайного «провідника», звичного для користувачів систем Windows, Linux тощо.

Зліва додаток буде мати сайдбар з кількома вкладками:

— My storage. Це буде посилання на сторінку кореневої папки сховища користува. Це будуть відображатися файли та папки з пустим значенням поля folderId, тобто ті що не мають «батьківської» папки;

— Recent. На цій сторінці будуть відображатися файли відсортовані за часом створення або редагування, відкриття;

— Shared. На цій сторінці будуть знаходитися файли, до яких поточний користувач матиме доступ, наданий від інших користувачів;

Який же вигляд матиме сама сторінка перегляду файлів? Як було сказано раніше, вид цієї сторінки повинен бути максимально наближений до «провідника» звичайної операційної системи. А отже, повинен мати такий функціонал:

— перегляд всіх файлів та папок, у зручному інтерфейсі, з автоматичним визначенням розширенням файлу та відображенням відповідної іконки для цього файлу. А також ці компоненти мають відображати інформацію про файл, його назви, дату створення;

— добре було б мати кнопку, що змінює тип відображення файлів (у вигляді списку чи плиток);

— окрему кнопку для створення завантаження нового файлу, або створення папки;

— відображення шляху до поточно відкритої папки, у вигляді «хлібних крихт»;

— можливість виділення кількох файлів, чи папок для виконання дій над ними. Наприклад для копіювання, переміщення, скачування файлів чи їх видалення;

— отримання списку дій при натисканні на праву кнопку миші. Те що буде з’являтися у випадаючому меню, буде змінюватися від того, на якому об’єкті була натиснута кнопка. Наприклад, на окремому файлі, папці, на пустому просторі, чи коли є виділені файли;

— має існувати окреме спливаюче вікно, для редагування інформації файлу чи папки. Наприклад зміна назви;

— також ця сторінка має мати поле для вводу та пошуку файлів за назвою;

— окремо потрібно відзначити можливість створення копій і переміщення файлів і папок, щоб була зручна можливість робити ці дії;

— та останнім пунктом, що має бути реалізований з функціоналу це можливість змінювати розташування файлів, шляхом їх «перетягування» до інших папок чи наводячи прямо на «хлібні крихти».

Тож виходячи с отриманих вимог на етапі проектування отримуємо приблизний макет додатку (див. рис. 2.2).

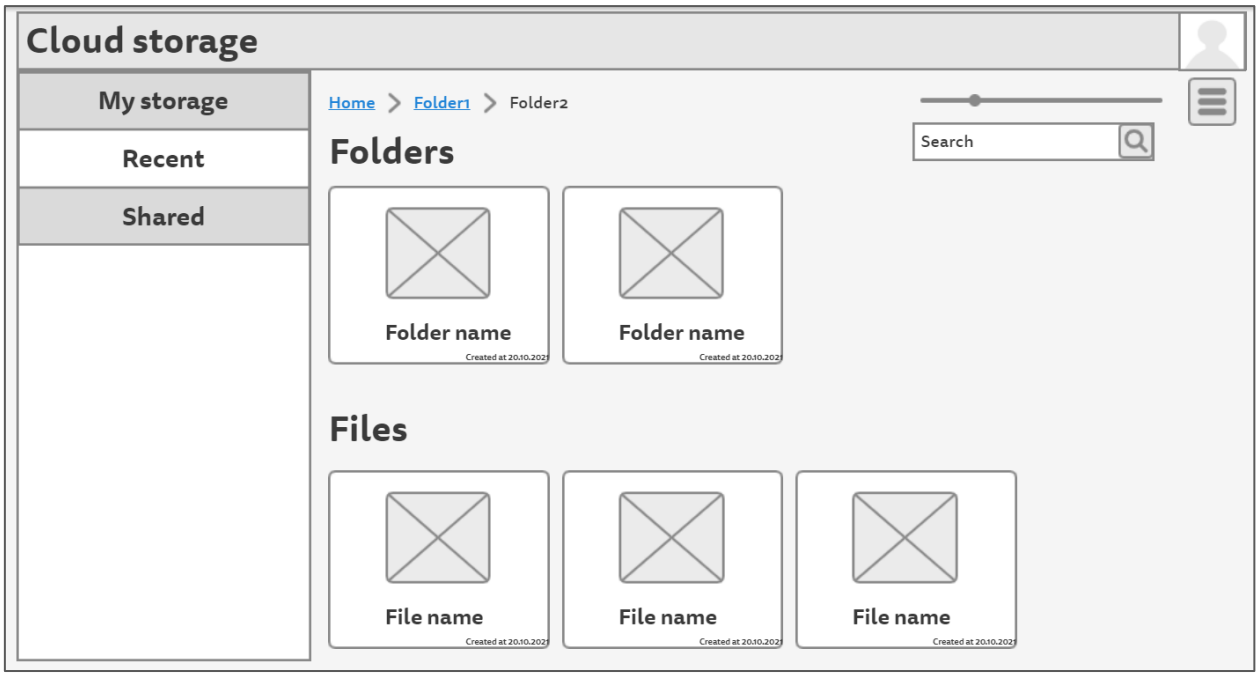

Рисунок 2.2 – Макет веб додатку

3 РЕАЛІЗАЦІЯ

3.1 Створення Back-end серверу

Для розробки як Back-end частини, так і Front-end частини було обрано мову програмування TypeScript. Ця мова відрізняється від звичайного JavaScript тим, що дозволяє вказувати чітку типізацію для змінних. А також створювати інтерфейси та спеціальні типи для інших одиниць розробки, таких як об'єкти, методи та інші. Хоча розробка на TypeScript може займати трохи більше часу, те що вона привносить в процес розробки та оладки додатку – набагато важливіше витрат часу.

Перейдемо до безпосереднього створення та запуску бекенд сервера. Для розгортання було обрано пакет graphql-yoga. Цей пакет, або бібліотека, добре зарекомендував себе у розробці веб додатків з використанням GraphQL. Усередині себе представляє звичайний Express сервер, а також інтеграцію з сервером Apollo. Серед його переваг можна назвати підтримку підписок GraphQL. А також підтримку Playground та middlewares прямо з «коробки». Ще можно виділити те, що цей сервер доволі розповсюджений та проблем з його розгортанням не буде на будь-яких популярних хостингах проектів.

На цьому етапі створюється файл index.ts. Він міститиме лише один метод «run», який буде викликатись наприкінці. Ось код цього метода, за виключенням декількох строк коду:

```
async function run() {
try {
  const server = new GraphQLServer({
    typeDefs,
    resolvers,
    context: (req) => req,
    middlewares: [authenticate],
```

```
    })

    server.express.use(cors())

    await server.start({
      playground: true,
      port: SERVER_PORT,
      bodyParserOptions: {
        limit: '10mb',
        type: 'application/json',
      },
    })
    console.log(`Server ready at http://localhost:${SERVER_PORT}`)
  } catch (err) {
    console.log(err)
  }
}

run()
```

У цьому прикладі ми бачимо метод «run» що має у собі ключові виклики для роботи серверу. Де «new GraphQLServer» створення об’єкту сервера, з такими параметрами як «typeDefs» – додавання схеми, «resolvers» – обробники запитів на сервер, , «context» – додавання контексту та «middlewares» – додавання проміжних обробників запитів, у нашому випадку перевірка аутентифікації.

Строка «server.express.use(cors())» відповідає за правильність роботи політки спільного використання ресурсів.

Та сам метод «start» з об’єкту сервера, що запускає сервер, та отримує параметри для старту. А саме «playground» – активацію плейграунду GraphQL, спеціальної утиліти для виконання запитів. Наступним параметром йде порт на котрому буде працювати сервер та «bodyParserOptions» – базовий об’єкт що описує те ліміт та тип для запитів, які отримує сервер.

На цьому етапі створення та старт сервера закінчено, але у метод «run», можливо додавати і інші методи та конфігурації для роботи сервера. Наприклад додавання сокетів, створення бакетів, або обробку окремих ендпоінтів. Усе це можливо, оскільки усередині graphql-yoga базується на Express.

3.2 Створення сховища на основі MinIO

Як обговорювалося раніше, на етапі проектування для сховища самих файлів був обраний сервер сховище MinIO (див. рис. 3.1). Сховище є сервером з відкритим вихідним кодом, а також має сумісність Амазон S3. Сам сервер має файл Dockerfile, так що ніяких проблем з його установкою бути не може. Якщо говорити про взаємозв'язок сервера веб додатка та сервера MinIO, для цього є дуже зручний npm пакет minio. Який є клієнтом MinIO для використання MinIO сервера на стороні клієнта.

Для нашого серверу не потрібно багато функціоналу зі сховища MinIO. Усе що потрібно, це створення «бакета», іншими словами папки де будуть зберігатися файли. Достатньо буде однієї такої «папки». А також будуть потрібні методи. А саме, для створення бакета, а також для збереження та отримання файлів зі сховища.

Метод створення бакету:

```
export const initializeMinioBuckets = async () => {
const buckets = [MINIO_BUCKET_MAIN]
await Promise.all(
  buckets.map(async (bucket) => {
    if (!(await minioClient.bucketExists(bucket))) {
      await minioClient.makeBucket(bucket, '')
      console.log(`Bucket "${bucket}" created!`)
    }
  }),
```

```
  )
}
```

Цей метод викликається при старті головного сервера. Він створює масив «buckets», що отримує список назв потрібних бакетів взятих з ENV файлу, у нашому випадку він один, «MAIN». А потім виконує їх створення методом перебору кожної назви за допомогою метода «map». Перед створенням, перевіряється чи не був створений цей бакет раніше, та у разі успішного створення записує повідомлення у консоль про успіх.

```
import * as uniqueFileName from 'unique-filename'
import prisma from '../shared/src/prisma/prisma'
import minioClient from '../minio'

const saveFileHelper = async ( file, name = null ) => {
  const fileNamePath = uniqueFileName()

  minioClient.putObject(
    process.env.MINIO_BUCKET_MAIN,
    fileNamePath,
    file,
    function (err) {
      if (err) return console.log(err)
    },
  )

  const newFile = await prisma.file.create({
    data: {
      nameToShow: name,
      link: fileNamePath,
    }
  })

  return newStaticFile.id
}
export default saveFileHelper
```

Вище можна побачити реалізацію метода збереження файлу у сховищі MinIO. На вході методи отримує сам файл, та назву яка буде потім збережена у базі даних. Метод «uniqueFileName» з однойменного пакету створює унікальне ім’я, що буде знаходитися у сховищі. Метод «minioClient.putObject» з пакету minio збережує файл у сховищі. На вході приймає ім’я бакету, ім’я файлу, сам файл та функцію-callback яка буде викликана у разі помилки. Далі йде виклик методу з бібліотеки prisma, про котру буде сказано пізніше. Але як видно з назви методу, він створює новий запис у базі даних, та отримує назву файлу та назву файлу у сховищі MinIO. Та повертає об’єкт з інформацією про створений файл. Наприкінці метод saveFileHelper повертає ID щойно створеного файлу для подальшого використання.

Для отримання файлу зі сховища, викликається інший метод MinIO клієнту. А саме «minioClient.getObject», що на вході отримує три параметра: ім’я бакету, ім’я файлу та функцію-callback що буде викликана у разі помилки.

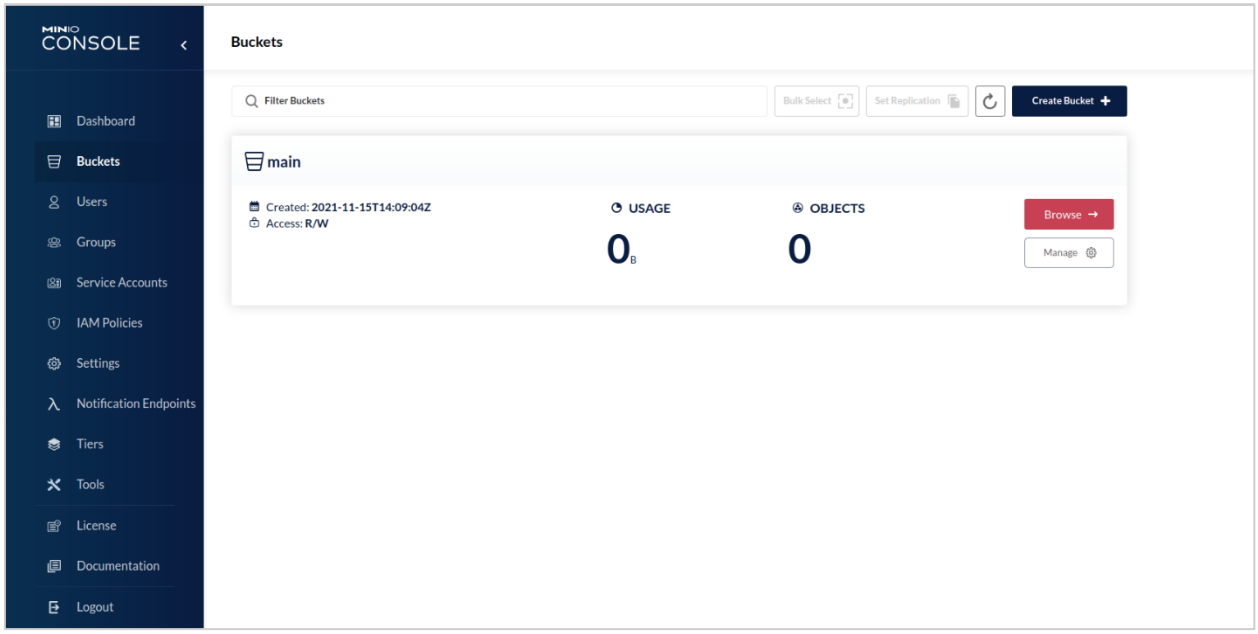

Рисунок 3.1 – Огляд MinIO серверу

3.3 Логіка сутностей та створення бази даних

Для спрощення опису сутностей та генерації схеми бази даних на допомогу була обрана така утиліта як Prisma, яка є надбудовою для Apollo сервером. Вона дозволяє описувати сутності в більш розгорнутому вигляді, а також відразу вказувати необхідні типи даних, значення за замовчуванням, відносини з іншими сутностями тощо. На підставі цих схем, а також їх подальшої зміни Prisma створює міграції для потрібної бази даних, дані для підключення до якої також вказуються у файлі конфігурації.

Нижче приведена частину коду з файлу «schema.prisma» з описанням сутності юзера, в якій описується схема, чимось схожа на опис схеми GraphQL з файлу typeDefs, але розширена додатковими даними.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id                Int            @id @default(autoincrement())
  email             String         @unique
  name              String
  password          String
  surname           String
  createdAt         DateTime       @default(now())
  avatar            String         @default("")
  files             File[]
  folders           Folder[]
  accessibleFiles   File[]      @relation("AccessibleFiles")
}
```

На підставі цієї схеми, Prisma розуміє, які міграції потрібно створювати. А також виконує GraphQL запити так само орієнтуючись на цю схему. Перші два об’єкти описують для якої бази даних створювати міграції та дані до її підключення. Далі знаходиться опис сутності юзера. Кожне поле існує у виді ключів назви, типу даних та додаткових параметрів. Тут бачимо звичайні поля, зі звичайними типами Int, String, DateTime, тощо. Більш цікавими є опис таких полей як files, folders або accessibleFiles. Тут вони мають тип даних у вигляді масиву. Це не означає, що в базі даних ці поля зберігатимуть у собі масив. Насправді, в міграціях будуть створені окремі PIVOT таблиці, для опису відносин цих сутностей. Зокрема файлів користувача, його папок і файлів до яких користувач має доступ.

Нижче можна побачити схему сутності файлу, а схему сутності папки можна буде знайти у додатку роботи.

```
model File {
  id                Int            @id @default(autoincrement())
  name              String
  link              String
  userId            Int
  createdAt         DateTime       @default(now())
  updatedAt         DateTime       @default(now()) @updatedAt
  folderId          Int?
  folder            Folder?        @relation(fields: [folderId], references: [id])
  user              User           @relation(fields: [userId], references: [id])
  usersWithAccess   User[]         @relation("AccessibleDocuments")
}
```

Для порівняння приведена безпосередньо GraphQL схема з файлу typeDefs. Ця схема вже потрібна для роботи самого GraphQL, аби він розумів які сутності має, та які дані може повертати. Схема для папок:

```
type Folder {
  id:        Int!
  name:      String!
  createdAt: DateTime!
```

```
  folderId:  Int
  folders:   [Folder]!
  files:     [File]!
  user:      User!
}
```

3.4 Розробка GraphQL ендпоінтів

Як описувалося раніше, мова запитів GraphQL відрізняється від звичайного уявлення створення API сервера. Ендпоінти більше схожі на виклики методів з певними параметрами, що передаються в цей метод.

Опис ендпоінтів GraphQL сервера відбувається у тому файлі де і описуються типи сутностей. А саме у файлі typeDefs. Тут існують 2 об'єкти «Query» та «Mutation». У першому описуються методи які відповідають отримання даних, на кшталт GET запитів. А в другому описуються методи зміни, зберігання або видалення даних, за кшталт POST, PUT, DELETE.

Роздивимося декілька з них. Першим таким прикладом буде ендпоінт для пошуку користувачів. Його код виглядає наступним чином: «findUsers(searchPhrase: String!): [User]». Про що говорить те, як написаний цей приклад? Метод називається «findUsers». Вхідним параметром він приймає один аргумент, це «searchPhrase» – фраза для пошуку користувачів, за ім’ям чи електронною адресою. Далі вказується тип, цього аргументу, String, чи строковий. А знак оклику вказує на те, що цей аргумент є обов’язковим. Далі, через двокрапку вказується тип даних для повернення. У цьому випадку це масив об’єктів типу «User». Цей тип оголошено раніше, у цьому ж файлі.

Звичайно, ці приклади можуть виглядати значно більше із використанням створених типів. Наприклад «getFilesFromSearchAndFilters(searchPhrase: String!, filters: [FilterPayload]!): [File]». Метод називається «getFilesFromSearchAndFilters». Він відповідає за

пошук файлів, за назвою із використанням фільтрів. Тут вже два вхідних параметри, перший «searchPhrase», строкового типу – це фраза за якою буде відбуватися пошук, а друга це масив фільтрів із «кастомним» типом «filterPayload». На виході з ендпоінта повертається масив файлів. Ось як виглядає створений тип. Він має два поля строкового типу, із назвою фільтру та значенням.

```
input FilterPayload {
  filter: String!
  value: String!
}
```

Роздивимося ще один ендпоінт, але вже іншого типу, а саме «Mutation». Він виглядає так «deleteFile(files: [FilesToChangeModel]!): Boolean!». Якщо звернути увагу, то він сильно не відрізняється своєю структурою, але у даному випадку виконує видалення файлів. А як результат виконання повертає булеве значення про успішність виконання операції.

Опис усіх інших ендпоінтів, або методів, не сильно відрізняється, та загалом має дуже схожу структуру. Далі буде розглянуто декілька прикладів реалізації безпосередньо логіки роботи ендпоінтів.

Першим, дуже простим прикладом буде ендпоінт логіну користувача, код цього методу виглядає так:

```
const login = async (root, { email, password }) => {
  const user = await prisma.user.findFirst({
    where: {
      email,
    },
  })

  if (!user || !(await checkHash(password, user.password))) {
    return null
  }

  return await getJWTToken(_.omit(user, 'avatar'))
```

```
}
```

Бачимо звичайну асинхронну функцію. Її логіка проста і методи, які вона викликає в собі, у тому числі з використання синтаксису GraphQL говорять самі за себе. Як правило, всі методи з використанням Apollo Сервера приймають у собі як мінімум 2 аргументи. Перший, «root» об'єкт, несе в собі інформацію від сервера, наприклад, тут може знаходитися інформація про користувача який здійснив запит, якщо той вже виконав вхід раніше. Другим аргументом йде об'єкт запиту. В даному випадку, електронна адреса та пароль користувача. Далі використовуючи Prisma виконується пошук за заданою електронною поштою. Після чого відбувається перевірка правильності пароля, з тим який має користувач із зазначеною електронною адресою. Якщо паролі різні, повертається помилка. Якщо перевірка пройшла успішно, користувач який відправив запит, отримує токен для користування додатком.

Ще один цікавий ендпоінт, котрий потрібно розглянути в межах роботи, це ендпоінт отримання «хлібних крихт», для відображення шляху знаходження файлу, або папки, у дереві інших папок. Його код:

```
const getFolderPath = async (parent, args, { user }) => {
  try {
    isLogged(user)
    const pathData = []
    let folderIdExist = false

    do {
      folderIdExist = false

      const folderData = await prisma.folder.findFirst({
        where: {
          userId: user.id,
          id: !pathData.length
            ? args.folderId
            : pathData[pathData.length - 1].folderId,
```

```
      },
      select: {
        id: true,
        name: true,
        folderId: true,
      },
    })
    pathData.push(folderData)

    if (folderData.folderId) {
      folderIdExist = true
    }
  } while (folderIdExist)

  return pathData
 } catch (err) {
  logError(err, 'getFolderPath')
  return []
 }
}
```

Із чого складається логіка цієї функції? Насамперед метод «isLogged» перевірить, чи «залогінений» користувач. Оскільки отримувати інформацію про файли можуть лише користувачі які здійснили вхід. Далі, після створення необхідних змінних, знаходиться цикл "do-while", який має таку логіку. Він отримує дані про запитувану папку, а саме ID папки, її ім'я та ID батьківської папки. В кінці циклу, перевіряється, якщо у поточної папки є батьківська, то цикл потрібно повторити і так доти, поки папка не матиме батьківської папки. Наприкінці метод повертає масив знайдених папок, а саме ID та ім'я кожної з них, для відображення на Front-end частині веб-додатку.

3.5 Структура Front-end частини

Як було вирішено раніше, розробка Front-end частини відбуватиметься з використанням сучасної та дуже гнучкої бібліотеки ReactJS. Як правило, проект на React завжди має певну структуру, нехай і не чітко визначену. Все залежить від побажань та вимог розробника. В даному випадку немає сенсу розглядати створення такого проекту з нуля, тому що прикладів того, як це зробити безліч в інтернеті. Набагато цікавіше буде розглянути структуру поточного проекту (див. рис. 3.2). А також компоненти з яких складатиметься веб додаток. А також те, яку логіку він міститиме.

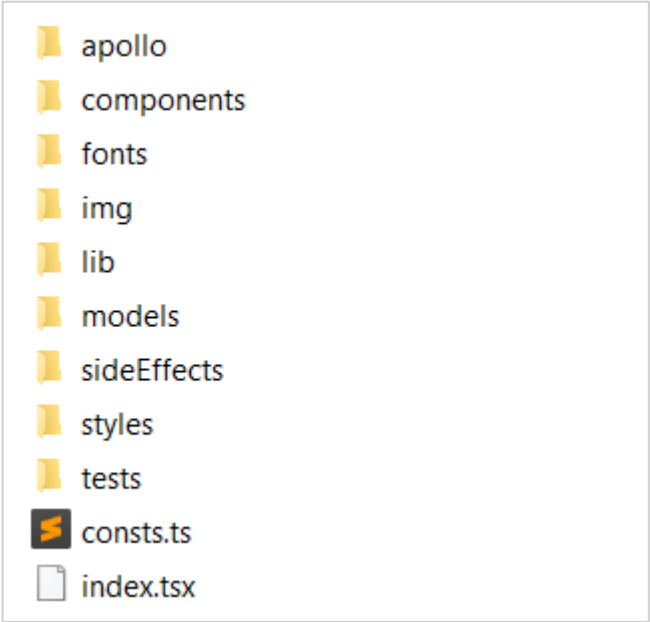

Рисунок 3.2 – Структура ReactJS додатку з папки «src»

Така структура проекту не є обов'язковою для розробки веб додатку і зв'язці з GraphQL. Але була створена саме така, для гнучкості та зручності розробки. Тут можна знайти такі папки та файли:

— «apollo». Як зазначалося раніше Apollo це бібліотека, котра вбудовується як у сервер так і у Front-end частину. У цій папці знаходяться

усі запити до ендпоінтів Back-end, із зазначенням необхідних даних для відправлення та отримання з серверу;

— «components». Тут усе просто, у цій папці знаходяться усі компоненти додатку. Зазвичай усі компоненті у React розділяють на такі, котрі відповідають лиш за відображення контенту, та ті, що відповідають за логіку додатку;

— «fonts» та «img». У ціх папках знаходяться статичні файли, необхідні для додатку. Файли шрифтів та малюнки відповідно;

— «lib». Ця папка розміщює у собі набір різних допоміжних функцій логіки, котрі можуть використовуватися у різних компонентах веб додатку хмарного сховища;

— «models». Як відмічалося раніше, для розробки використовувалася мова програмування TypeScript, котра потребує чіткої типізації для змінних та об’єктів. У цій папці можна знайти «моделі» або типи об’єктів що будуть повертатися з Back-end;

— «sideEffects». Ця папка також містить у собі проміжні виклики методів з папки «apollo», безпосередньо ці методи викликаються з компонентів. Та відповідають за валідацію «моделей», повернення оброблених даних та відловлювання помилок;

— «styles». У цій папці знаходяться різні scss файли, котрі додають глобальні стилі чи наприклад різноманітні константи або міксини.

— «test». Папка з тестами проекту;

— Файл «consts.ts» з різними константами для веб додатку. Та index.tsx – головний файл для монтування ReactJS додатку до сторінки.

3.5 Взаємодія веб додатку з сервером

Взаємодія з Back-end сервером відбуватиметься за допомогою пакета Apollo Client. Клієнт дозволяє створювати більш зручне та гнучке з'єднання з

сервером для виконання GraphQL запитів або, наприклад, підключення веб-сокетів. Також Apollo Client має можливість створювати централізоване сховище додатку, яке може стати відмінною заміною Redux. Таке сховище використовується в додатку, але не буде докладно розглядатися так як не входить до об'єкта дослідження.

Розглянемо приклад виконання запиту до серверу з використанням Apollo Client та GraphQL синтаксису для отримання файлів з папки.

```
const getFolderFiles: QueryCall<{
  getFolderFiles: IFile[] | null
}> = (payload: IGetSubFoldersPayload) =>
  client.query({
    query: gql`
      query getFolderFiles ($folderId: Int!) {
        getFolderFiles (folderId: $folderId) {
          id
          name
          createdAt
          type
        } } `,
    variables: payload,
  })

interface IGetSubFoldersPayload {
  folderId: number
}
```

У цьому прикладі знаходиться типовий виклик ендпоінту серверу за допомогою Apollo Client. Метод називається «getFolderFiles». На вході він отримує аргументи типу «IGetSubFoldersPayload», що знаходиться нижче, а саме ID папки з котрої очікується отримати файли. У тілі виклику бачимо GraphQL синтаксим, а саме те, які поля потрібно отримати з серверу. Як результат виконання очікується масив типу «IFile», інтерфейс котрого знаходиться у папці «models».

Усі інші запити до серверу з використанням мови GraphQL та Apollo клієнту мають ідентичну структуру. Змінюється тільки тип запиту, наприклад «query» на «mutate». А також вхідні дані, та ті котрі очікуються на повернення. Звичайно, за допомогою GraphQL можно коригувати, які самі дані потрібно отримати.

3.6 Реалізація інтерфейсу та його компонентів

На останньому буде проведено огляд розроблених компонентів та функціоналу додатку. Спочатку розглянемо сторінку редагування даних користувача (див. рис. 3.3).

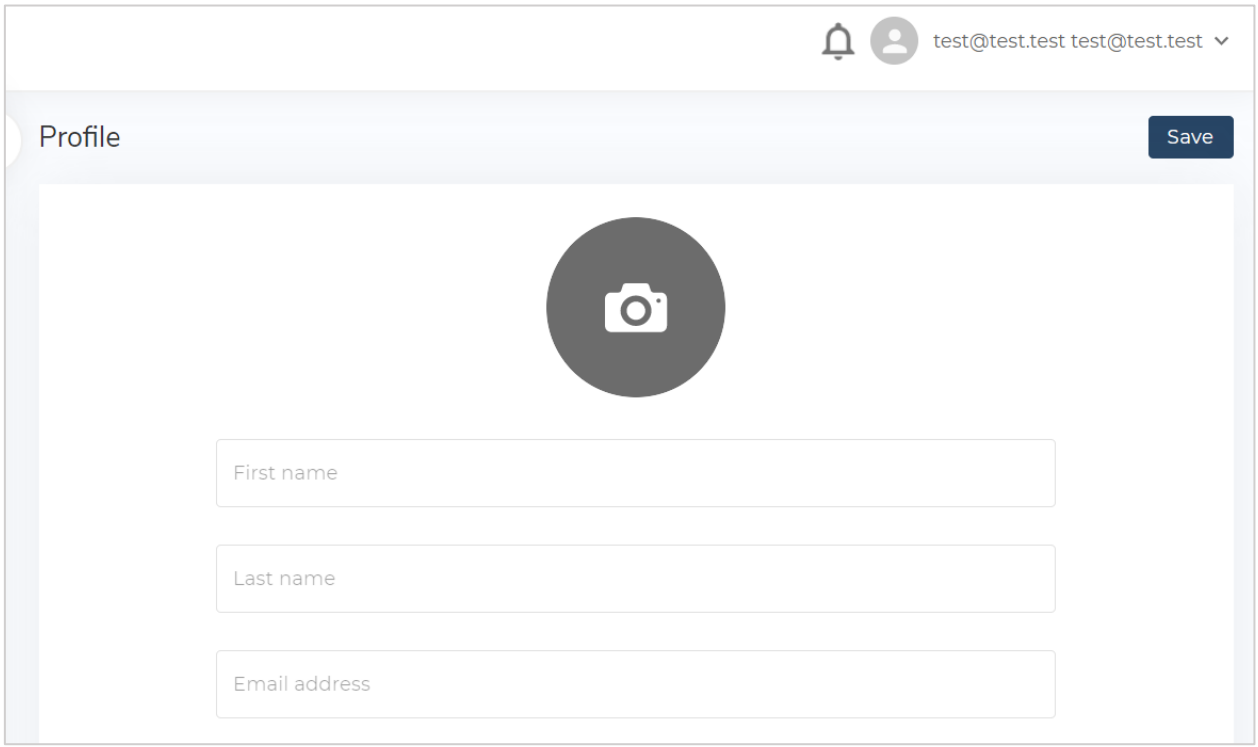

Рисунок 3.3 – Сторінка редагування даних користувача

Перейдемо безпосередньо на сторінку хмарного сховища, де відображаються створені файли, папки та інше. Бачимо створене меню на у виді «сайдбару», із можливістю переходу на різні сторінки. А також сам

простір «провідника», де знаходяться файли та папки. І елементи управління для зміни відображення та пошуку (див. рис. 3.4).

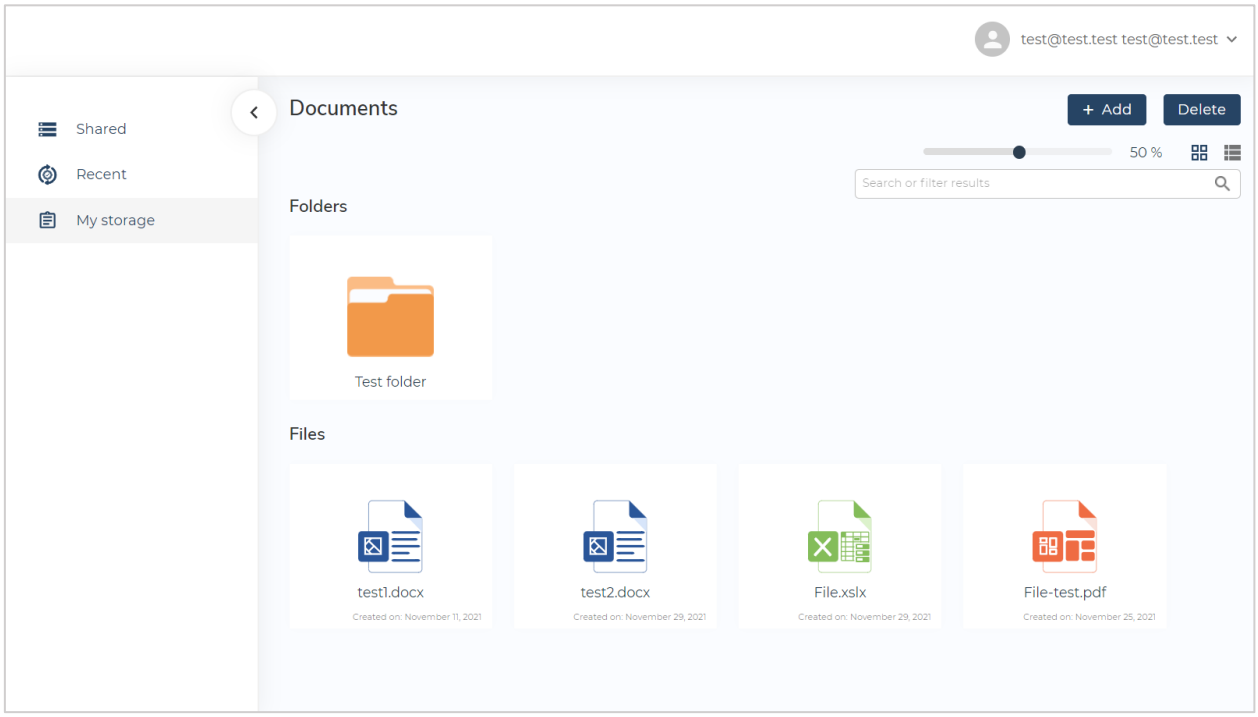

Рисунок 3.4 – Сторінка «провідника»

Наступний рисунок відібражає результат з ендпоінту для отримання «хлібних крихт», або повного шляху до папки (див. рис. 3.5).

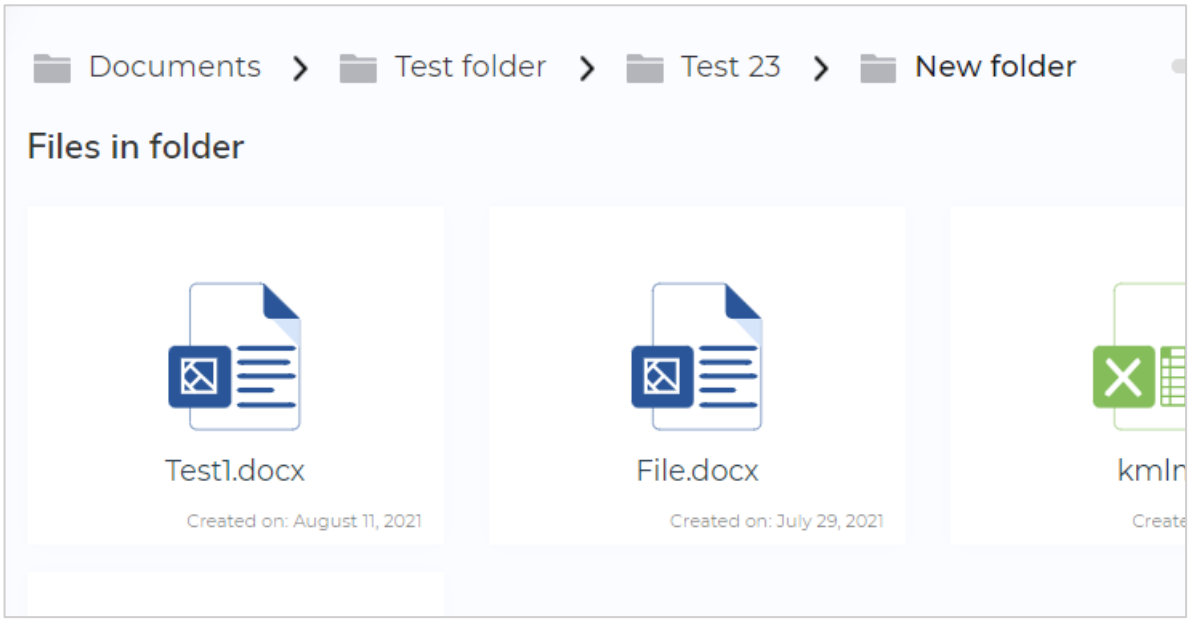

Рисунок 3.5 – Відображення шляху до папки

Останніми двома рисунками що буде розглянуто у цьому підрозділі буде огляд дій доступних для файлів чи папок при натисканні правої кнопки миші та зміна розташування файлу, шляхом його «перетягування» (див. рис. 3.6 – 3.7).

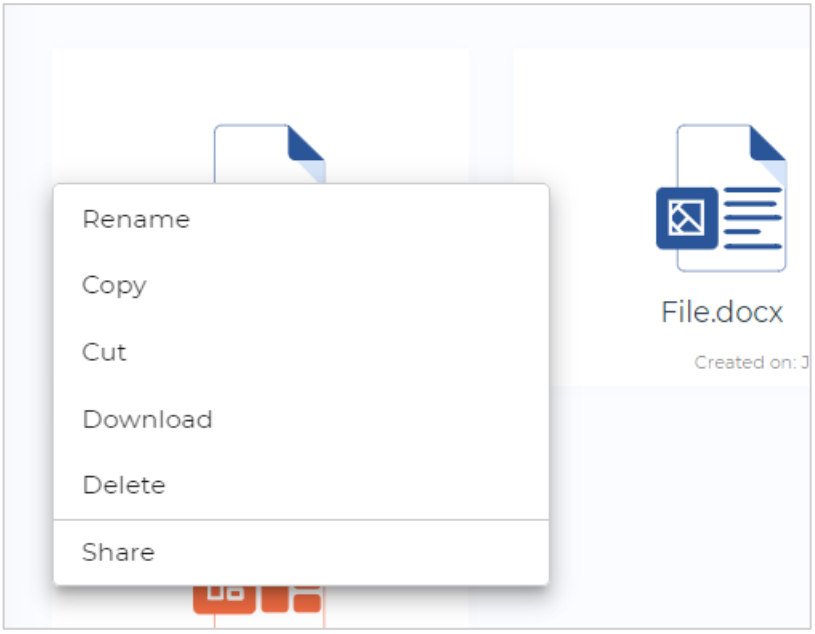

Рисунок 3.6 – Дії при натисканні на праву кнопку миші

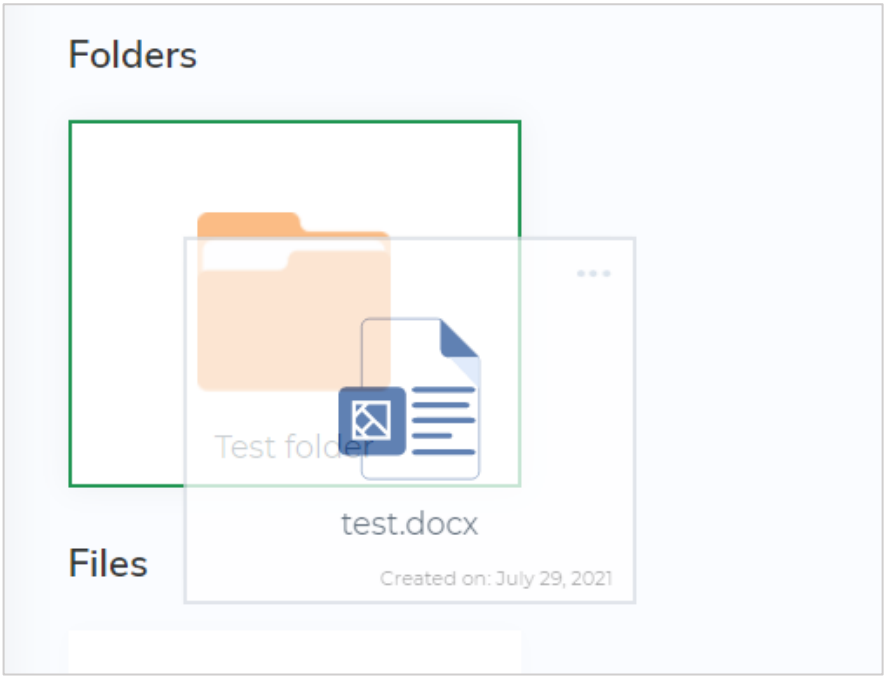

Рисунок 3.7 – Можливість переміщення файлів

ВИСНОВКИ

У час, коли об’єми даних користувачів постійно ростуть, гостро стає питання безпечного збереження цих файлів, та отримання до них доступу з будь-якої точки планети. На допомогу приходять різноманітні хмарні сховища, котрі вирішують проблеми людей у багатьох сферах життя. Реалізувати таке сховище можна за допомогою сучасної Front-end бібліотеки ReactJS та передової мови запитів GraphQL, що і було зроблено в межах виконання кваліфікаційної роботи.

У результаті виконання дипломної кваліфікаційної роботи магістра, ознайомився та отримав навички роботи з бібліотекою ReactJS та мовою запитів GraphQL. Ці технології є передовими у веб розробці, та їх знання можна назвати обов’язковим для розробників у сфері веб.

Другою частиною кваліфікаційної роботи, була розробка веб додатку «Хмарного сховища», з використанням ReactJS та GraphQL. Поставлена задача була успішно виконана. У результаті був створений сучасний веб додаток «Хмарного сховища», з використанням ReactJS та GraphQL, а також сумісних технологій, таких як Apollo GraphQL, MinIO, Prisma та інших. Веб додаток отримав сучасний інтерфейс. Підтримку усіх потрібних функцій для збереження файлів, їх редагування, видалення, копіювання, тощо. Додаток було реалізовано у вигляді провідника, так що інтерфейс є звичним для усіх користувачів будь-яких операційних систем. З підтримкою створення папок, їх переносу та інших звичних функції. Також завдяки використанню можливостей бібліотеки React.js та мови запитів GraphQL, веб додаток є гнучким для розширення функціоналу, підтримки роботи, розгортання або тестування розробленого проекту.

ПЕРЕЛІК ПОСИЛАНЬ

1) Google Drive — Матеріал з Вікіпедії. URL: https://uk.wikipedia.org/wiki/Google_Drive (дата звернення 15.10.2021);

2) React.js — Official site. URL: https://reactjs.org/ (дата звернення 15.10.2021);

3) React — Матеріал з Вікіпедії. URL: https://uk.wikipedia.org/wiki/React (дата звернення 15.10.2021);

4) ECMAScript — Матеріал з Вікіпедії. URL: https://ru.wikipedia.org/wiki/ECMAScript (дата звернення 15.10.2021);

5) GraphQL — Матеріал з Вікіпедії. URL: https://uk.wikipedia.org/wiki/GraphQL (дата звернення 15.10.2021);

6) Apollo GraphQL — Official site. URL: https://www.apollographql.com/ (дата звернення 15.10.2021);

7) MinIO — Матеріал з Вікіпедії. URL: https://en.wikipedia.org/wiki/MinIO (дата звернення 15.10.2021);

8) GraphQL — Official site. URL: https://graphql.org/ (дата звернення 15.10.2021);

9) Prisma — Official site. URL: https://www.prisma.io/ (дата звернення 15.10.2021);

10) Cloud storage — Матеріал з Вікіпедії. URL: https://en.wikipedia.org/wiki/Cloud_storage (дата звернення 15.10.2021).

ДОДАТОК А

Схема «папки» у файлі «schema.prisma»

```
model Folder {
  id          Int          @id @default(autoincrement())
  name        String
  createdAt   DateTime     @default(now())
  userId      Int
  folderId    Int?
  folder      Folder?      @relation("FolderToFolder", fields: [folderId], references: [id])
  user        User         @relation(fields: [userId], references: [id])
  files       File[]
  folders     Folder[]     @relation("FolderToFolder")
}
```

Опис типів «користувача» та «файлу» у файлі «typeDefs»

```
  type User {
    id: Int!
    email: String!
    name: String
    password: String!
    surname: String!
    createdAt: DateTime!
    avatar: String
  }


  type File {
    id: Int!
    name: String!
```

```
  link: String!
  createdAt: DateTime!
  updatedAt: DateTime!
  user: User!
  folder: Folder
}
```

ДОДАТОК Б

Приклади взаємодії з веб додатком

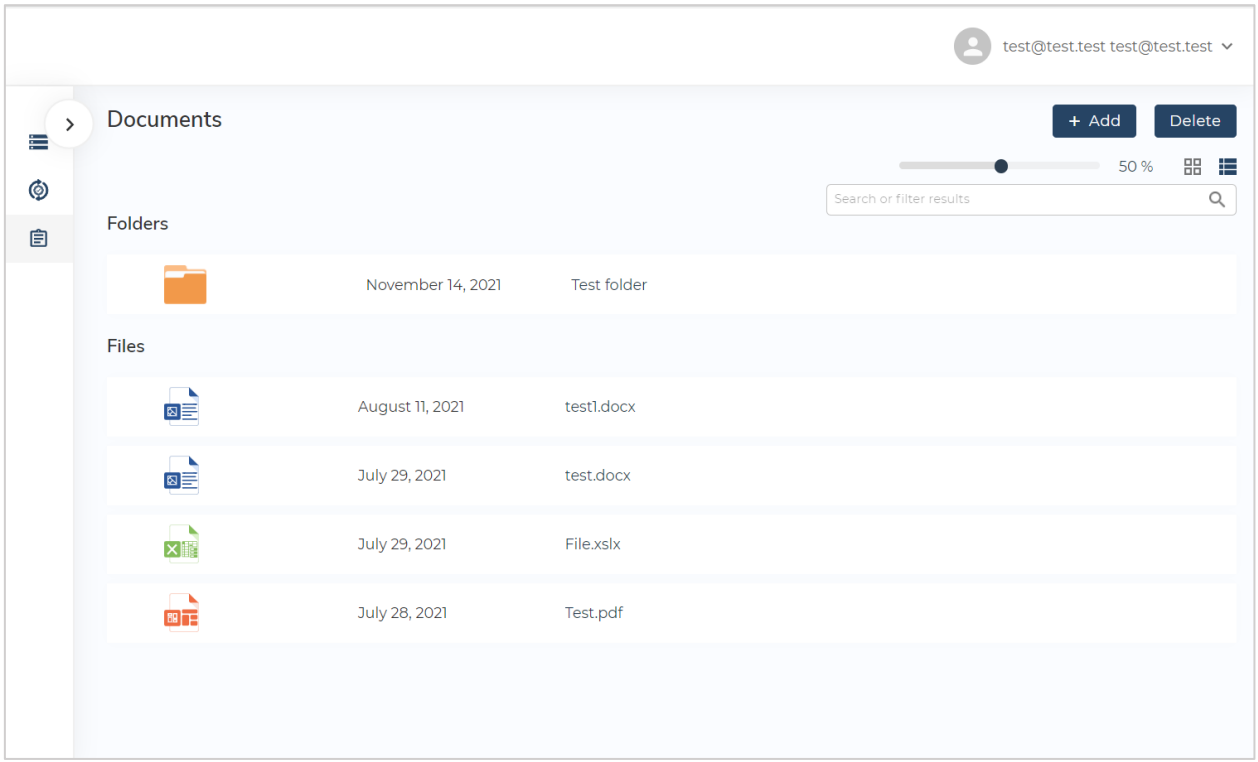

Рисунок Б.1 – Зміна типу відображення хмарного сховища

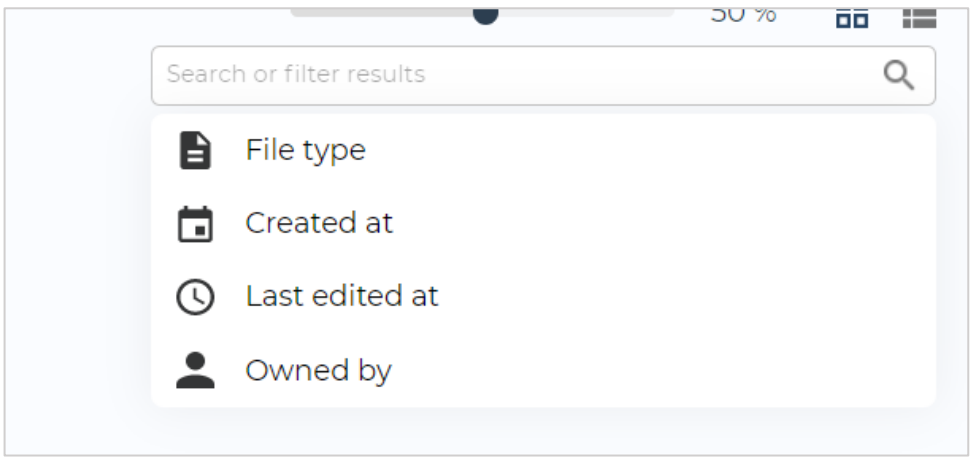

Рисунок Б.2 – Пошук та фільтрація

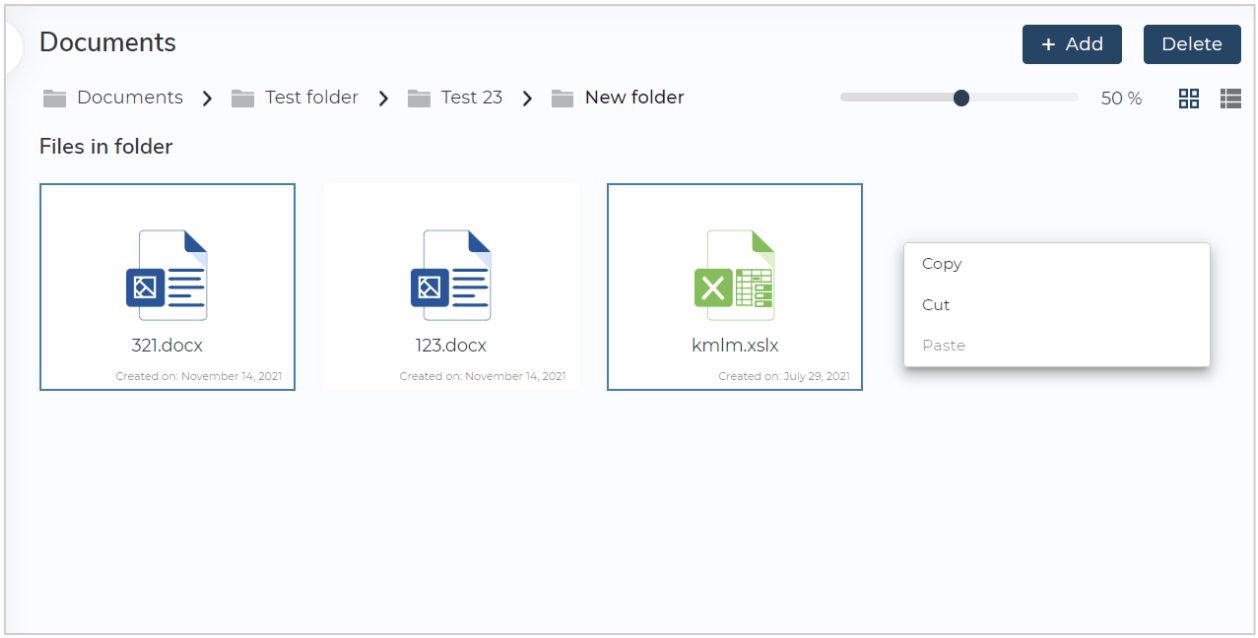

Рисунок Б.3 – Можливість виділення декількох файлів та дії над ними