Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Комп'ютерна система хмарного сховища файлів»
08-54.БКР.009.00.000 ПЗ

Виконав: студент 4 курсу, групи 1КІ-21б
спеціальності 123 — «Комп'ютерна інженерія»
освітня програма – «Комп'ютерна інженерія»
Квятковський А. А.

Керівник: к.т.н., доц. каф. ОТ
Муращенко О. Г.

Рецензент: д.т.н., проф. каф. МБІС
Яремчук Ю. Є.

ДОПУЩЕНО
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д

"18" 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень бакалавр

Галузь знань – 12 – Інформаційні технології

Спеціальність – 123 – Комп'ютерна інженерія

Освітньо-професійна програма – Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. _____ Азаров О. Д
«21» березня 2025 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Квятковському Антону Анатолійовичу

1 Тема роботи: «Комп'ютерна система хмарного сховища файлів», керівник роботи к.т.н., доц. каф. ОТ Муращенко О. Г. затверджені наказом ВНТУ від «20» березня 2025 року №97.

2 Термін подання студентом проєкту: 13.05.2025

3. Вихідні дані до роботи: Вихідні дані до роботи: функціональні вимоги до сервісу, структура бази даних PostgreSQL, архітектура на платформі ASP.NET, підтримка автентифікації через cookies. Застосовне програмне забезпечення: .NET SDK, PostgreSQL, Docker, Postman, JetBrains Rider.

4 Зміст розрахунково-пояснювальної записки: вступ, аналіз предметної області, вибір та обґрунтування технологій розробки, програмна реалізація застосунка

5 Перелік ілюстративного матеріалу: схема залежності класів серверної частини, сторінка з файлами користувача, файли з приміненням сортування, пошук за іменем файлу.

6 Консультанти розділів роботи наведені у таблиці 1

Таблиця 1— Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	завдання прийняв
Спеціальна частина	к.т.н., доц. каф. ОТ Муращенко О. Г.	21.03.25	13.05.25
Нормоконтроль	ас. каф. ОТ Швець С.І.	14.05.25	30.05.25

7 Дата видачі завдання 21.03.2025 р.

8 Календарний план наведений у таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Термін виконання		Примітка
		початок	закінчення	
1	Постановка задачі. Визначення мети, завдань, актуальності та структури системи цифрової соціалізації.	21.03.25	27.03.25	виконано
2	Аналіз існуючих онлайн-сервісів знайомств. Обгрунтування вибору технологій ASP.NET та Nuxt.	28.03.25	04.04.25	виконано
3	Проєктування бази даних, API та структури взаємодії клієнта з сервером.	05.04.25	12.04.25	виконано
4	Реалізація серверної частини: обробка користувацьких даних та автентифікація	13.04.25	27.04.25	виконано
5	Розробка клієнтського інтерфейсу	28.04.25	11.05.25	виконано
6	Оформлення пояснювальної записки та ілюстративного матеріалу	11.05.25	13.05.25	виконано
7	Попередній захист БКР	13.05.25	13.05.25	виконано

Студент _____ (підпис) Квятковський А. А.

Керівник роботи _____ (підпис) Муращенко О. Г.

АНОТАЦІЯ

УДК 004.9

Квятковський А. А. Комп’ютерна система хмарного сховища файлів. Бакалаврська кваліфікаційна робота зі спеціальності 123 — Комп’ютерна інженерія, освітня програма Комп’ютерна інженерія. Вінниця: ВНТУ, 2025. 70 с.

На укр. мові. Бібліогр.: 21 назв; рис.: 26; табл. 4.

Бакалаврська кваліфікаційна робота на тему "Комп’ютерна система хмарного сховища файлів" присвячена розробці комп’ютерній системі хмарного сховища файлів, для забезпечення зручного, безпечного та ефективного зберігання й обміну даними через інтернет. Основна увага приділена архітектурі клієнт-сервер, у якій клієнтська частина реалізує інтерфейс взаємодії користувача, а серверна відповідає за обробку запитів, автентифікацію, управління файлами та збереження метаданих.

У межах системи реалізовано функціональність завантаження, зберігання, перегляду та завантаження файлів користувачем із персонального сховища. Для збереження інформації використано реляційну базу даних PostgreSQL, що забезпечує надійність і цілісність даних. Взаємодія клієнта з сервером відбувається через RESTful API поверх протоколу HTTPS, що гарантує захищену передачу даних.

ABSTRACT

Kviatkovskyi A. A. Computer System for Cloud File Storage. Bachelor's qualification thesis in specialty 123 — Computer Engineering, educational program "Computer Engineering." Vinnytsia: VNTU, 2025. 70 p.

In Ukrainian. Bibliography: 21 sources; illustrations: 26; tables: 4.

The thesis titled "Computer System for Cloud File Storage" is dedicated to the development of a computer system for cloud-based file storage, aimed at providing convenient, secure, and efficient data storage and exchange over the Internet. The primary focus is on the client-server architecture, in which the client side implements the user interaction interface, while the server side is responsible for request processing, authentication, file management, and metadata storage.

The system implements functionality for uploading, storing, viewing, and downloading user files from a personal storage space. A relational PostgreSQL database is used for data storage, ensuring data reliability and integrity. Interaction between the client and the server is carried out through a RESTful API over the HTTPS protocol, which guarantees secure data transmission.

ЗМІСТ

ВСТУП

У сучасному цифровому світі хмарні технології стали невід'ємною частиною повсякденного життя. Вони забезпечують зручний доступ до даних з будь-якого місця та пристрою. Зростає потреба в ефективному зберіганні й управлінні файлами, особливо в умовах віддаленої роботи та глобалізації. Це стимулює розвиток інноваційних рішень. Одним із них є хмарні системи зберігання, які поєднують безпечне збереження даних, зручний інтерфейс, а також функції пошуку, сортування та завантаження файлів. Такі системи є корисними для приватних користувачів, компаній та освітніх установ [1].

Актуальність розробки хмарного сховища пояснюється кількома причинами. По-перше, обсяг цифрових даних стрімко зростає, тому потрібні надійні платформи, які забезпечать централізоване зберігання та захист інформації. По-друге, дедалі більше людей користується мобільними пристроями й інтернетом, тому важливо мати доступ до файлів будь-коли і з будь-якого місця. Крім того, на ринку не вистачає рішень, які одночасно мають високий рівень безпеки, зручний інтерфейс і підтримують різні формати файлів. Саме тому створення такої системи є не лише актуальним, а й практично важливим завданням.

Метою дослідження є створення та оцінка комп'ютерної системи хмарного зберігання файлів, яка буде зручною, безпечною й адаптивною. Система має дозволяти зберігати, шукати, сортувати й завантажувати файли. Окрему увагу приділено підтримці автентифікації, фільтрації, пагінації та адаптивного дизайну.

Завдання дослідження включають кілька етапів:

— вивчення наявних рішень та визначення їх сильних та слабких сторін;

— формулювання вимог до майбутньої системи;

— проєктування її архітектури;

— створення прототипу веб застосунку з основними функціями;

— проведення тестування.

Завершальним етапом є аналіз результатів і визначення напрямів для

подальшого розвитку системи.

Об’єкт дослідження — процес зберігання, управління та обміну файлами за допомогою хмарних технологій. Система має забезпечувати доступ до даних, їхню організацію та захист.

Предмет дослідження — структура та функціональність хмарного сховища. У фокусі — автентифікація, адаптивний інтерфейс, пошук, фільтрація та безпечний доступ до даних для різних користувачів.

Апробація результатів бакалаврського кваліфікаційної роботи була проведена на конференції “Молодь в науці: дослідження, проблеми, перспективи” (МН-2025) [2]:

Квятковський А. А. Комп’ютерна система хмарного сховища файлів соціалізації // Матеріали всеукраїнської науково-практичної Інтернет-конференції студентів, аспірантів та молодих науковців — Режим доступу : https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/view/25460

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Вимоги до веб застосунку хмарного сховища файлів

Вимоги до вебзастосунку хмарного сховища файлів визначають базові та розширені характеристики, необхідні для побудови повнофункціонального, безпечного, стабільного і водночас зручного інструменту для кінцевого користувача. Такий застосунок повинен не лише реалізовувати базову функціональність зберігання та передачі файлів, а й відповідати сучасним вимогам до масштабованості, адаптивності та інтеграції з іншими сервісами. Основна мета вебзастосунку — забезпечити зручний доступ до файлів із будь-якого пристрою, незалежно від платформи, з мінімальними затримками та високим рівнем безпеки. При цьому проєкт спирається на вже наявну програму, проте враховує додаткові запити користувачів і вимоги, зумовлені новими умовами використання.

Система повинна бути орієнтованою на ефективне управління даними, тобто забезпечувати завантаження, перегляд, видалення, сортування та пошук файлів у зручному графічному інтерфейсі. Окрім цього, застосунок має бути готовим до подальшого масштабування — додавання нових функцій, підключення зовнішніх сервісів, а також інтеграції в більші корпоративні екосистеми.

Ключовим функціональним компонентом є надійна система аутентифікації користувачів. Вона має дозволяти реєстрацію нового облікового запису через введення імені, електронної пошти та пароля. Під час реєстрації обов’язковою є базова валідація введених даних: перевірка правильності формату email-адреси, контроль мінімальної довжини пароля, а також бажано — повідомлення користувача про помилки без потреби перезавантаження сторінки. У разі успішної реєстрації користувач автоматично потрапляє на сторінку входу. Після входу в систему він отримує доступ до персонального сховища файлів, інакше кажучи — до захищеної частини вебінтерфейсу. Невідомі або неавторизовані користувачі повинні бути автоматично перенаправлені на форму входу, щоб унеможливити доступ до особистих або конфіденційних даних інших користувачів. Також необхідно реалізувати логіку виходу з облікового запису, яка завершуватиме

активну сесію, очищатиме cookie або токени доступу, після чого користувач повертається на форму входу.

Система має автоматично обробляти метадані файлів — назву, розмір, тип — і накладати обмеження на максимально допустимий розмір та кількість файлів у сесії завантаження. Список завантажених файлів повинен оновлюватися автоматично, без перезавантаження сторінки, та відображати ключову інформацію: ім’я файлу, дату завантаження, розмір, тип і доступні дії (скачати, видалити тощо). Щоб уникнути перевантаження інтерфейсу при великій кількості файлів, слід реалізувати пагінацію — розбиття виводу на сторінки з можливістю перегортання. Оптимальним рішенням є кнопка на кшталт "Завантажити ще", яка поступово підвантажує наступні елементи без повного перезавантаження сторінки, підтримуючи плавність взаємодії.

Функціонал пошуку та фільтрації файлів є обов’язковим для підвищення зручності навігації у випадку великої кількості документів. Пошук реалізується через введення ключового слова або частини назви, після чого система фільтрує і відображає лише ті файли, що відповідають запиту. Для полегшення аналізу результатів має оновлюватися не лише список, а й загальна кількість знайдених файлів. Додатково реалізується система фільтрації та сортування, яка дозволяє впорядковувати список за різними параметрами — датою завантаження, розміром, типом файлу. Користувач має мати змогу вибрати порядок сортування (за зростанням або спаданням), а також кількість елементів, що виводяться на сторінку — наприклад, 10, 15 або 20. Це особливо важливо для мобільних пристроїв, де кількість елементів має бути обмеженою для збереження зручності перегляду.

Елементи інтерфейсу — меню, сітка файлів, кнопки керування — повинні змінювати свій розмір і положення відповідно до ширини вікна браузера. Для цього доцільно використовувати сучасні CSS-фреймворки, які дозволяють легко реалізовувати адаптивні макети. Інтерфейс має залишатися інтуїтивно зрозумілим: усі функції повинні бути логічно згруповані, підписані та реагувати на дії користувача миттєво — без затримок або помилок відображення.

Забезпечення безпеки даних — ще один фундаментальний аспект роботи хмарного сховища. Система має бути захищена від несанкціонованого доступу, SQL-ін’єкцій, CSRF- і XSS-атак. Усі дії повинні відбуватися лише по захищеному протоколу HTTPS, а передача файлів — відбуватися з обов’язковим шифруванням. Крім цього, доцільно реалізувати резервне копіювання файлів, щоб зберігати їх у безпечному середовищі у випадку втрати доступу або технічної несправності. Для підвищення довіри користувачів і відповідності стандартам варто також реалізувати політику збереження та видалення даних відповідно до GDPR або подібних регламентів.

Загалом, вебзастосунок хмарного сховища повинен поєднувати в собі простоту у використанні, високу швидкість обробки запитів, багатофункціональність та надійний захист інформації. Такий баланс є основою для створення сучасного цифрового сервісу, здатного задовольнити потреби широкого кола користувачів [3].

1.2 Аналіз потреб користувачів та вимог до функціоналу

Аналіз потреб користувачів та визначення вимог до функціональних можливостей веб застосунку хмарного сховища файлів, орієнтованого на персональне використання, передбачає врахування індивідуальних звичок, очікувань та пріоритетів цільової аудиторії. Основними критеріями, що визначають цінність такого програмного продукту для користувача, є конфіденційність, зручність у користуванні, безперешкодний доступ до даних та висока стабільність роботи.

Для приватного використання хмарне сховище розглядається як засіб зберігання особистих документів, фотографій, відеоматеріалів чи творчих проєктів, що потребує абсолютного контролю з боку користувача над усіма аспектами доступу до цих даних. Важливою умовою є виключення можливості ознайомлення сторонніх осіб зі вмістом файлів без прямого дозволу власника.

Користувачі очікують наявності постійного доступу до даних незалежно від типу пристрою: смартфона, планшета або персонального комп’ютера.

Актуальність такої вимоги особливо зростає у контексті мобільності — під час подорожей, віддаленої роботи чи навчання. Реєстрація та авторизація в системі мають бути інтуїтивно зрозумілими та максимально спрощеними, базуючись на використанні електронної пошти й пароля з базовою перевіркою коректності введених даних, але без зайвого ускладнення процесу входу.

Після автентифікації користувач повинен отримати доступ до зрозумілого інтерфейсу, який дозволяє здійснювати завантаження файлів у два способи: шляхом перетягування у спеціальну область або через стандартний файловий діалог. Система повинна підтримувати основні формати файлів (документи, зображення, архіви) із розумними обмеженнями розміру для запобігання перевантаженню. Завантажені об’єкти відображаються у вигляді впорядкованого списку, який оновлюється динамічно без потреби в ручному оновленні сторінки, а кожен файл супроводжується елементами керування, що дозволяють його перегляд, завантаження або видалення.

Організація сховища також передбачає функціональні засоби пошуку, які забезпечують швидкий доступ до потрібного файлу за назвою або частиною тексту. Можливість фільтрації за параметрами (розмір, дата, тип) у поєднанні з опцією сортування результатів (за зростанням або спаданням) підвищує ефективність навігації. Для роботи з великими обсягами даних реалізується пагінація, заснована на кнопці для підвантаження наступної порції файлів, а загальна кількість об’єктів відображається у відповідному елементі інтерфейсу, що сприяє кращому контролю обсягу особистого архіву.

Конфіденційність та захист даних залишаються ключовими вимогами. Користувачі потребують впевненості у тому, що жодна особа не отримає доступу до їхніх файлів без дозволу, що зумовлює необхідність реалізації сучасних механізмів шифрування даних як під час передачі, так і при зберіганні. Додатково, впровадження автоматичного резервного копіювання гарантує збереження інформації навіть у разі технічних збоїв.

Окрему увагу слід приділити адаптивності інтерфейсу, що забезпечує коректне відображення та функціонування застосунку на пристроях із різною

роздільною здатністю. Інтерфейс має бути чутливим до взаємодії, забезпечуючи швидку реакцію на дії користувача, зокрема під час пошуку, завантаження чи фільтрації, що істотно підвищує загальну зручність використання та довіру до платформи.

На основі вищенаведеного формулюються функціональні вимоги до вебзастосунку. Система має передбачати:

— реєстрацію та автентифікацію з використанням email і пароля;

— захист від несанкціонованого доступу та можливість виходу з системи;

— головну сторінку із можливістю завантаження файлів, підтримкою основних форматів і обмеженнями за розміром;

— автоматичне оновлення списку файлів та їх інтуїтивне відображення;

— пошук за ключовими словами та фільтрацію за типом, розміром і датою;

— пагінацію з можливістю завантаження нових результатів;

— відображення загальної кількості файлів;

— адаптивний інтерфейс для різних пристроїв;

— шифрування даних і резервне копіювання.

Таким чином, формування функціоналу веб застосунку відбувається на основі реальних запитів користувачів, спрямованих на безпечне, зручне та гнучке зберігання особистої цифрової інформації.

1.3 Огляд та аналіз аналогів

Існує безліч сервісів для зберігання файлів, одними із найпопулярніших є: MEGA, pCloud, Sync.com, Icedrive та Proton Drive. Кожен із них пропонує унікальні особливості, переваги та недоліки, що допомагає зрозуміти їхню придатність для приватного зберігання даних, де доступ до файлів має бути виключно у користувача.

MEGA — один із найвідоміших сервісів для зберігання файлів, що з моменту запуску позиціонує себе як платформа з підвищеним рівнем конфіденційності (рис. 1.1). Він надає 20 ГБ безкоштовного простору, який можна тимчасово розширити до 30 ГБ через виконання простих дій, таких як

встановлення додатку на пристрої або запрошення друзів. Однією з ключових особливостей MEGA є наскрізне шифрування (end-to-end encryption), що гарантує, що файли залишаються доступними лише їхньому власнику — навіть серверна частина не має до них доступу.

Сервіс підтримує зручну веб-версію та мобільні додатки для iOS і Android, що дозволяє працювати з файлами на ходу. Користувачі можуть завантажувати файли через просте перетягування, переглядати архіви, використовувати пошук, і завантажувати вміст без обмежень на розмір, доки не перевищено ліміт простору. Перевагами MEGA є високий рівень конфіденційності, великий стартовий обсяг безкоштовного простору, а також гнучкість у доступі з різних платформ. Водночас сервіс має й недоліки: інтерфейс може здаватися складним для новачків, бонусний простір діє лише тимчасово, що вимагає постійної активності, а функція автоматичного резервного копіювання відсутня, що може бути критичним у разі втрати пристрою [4].

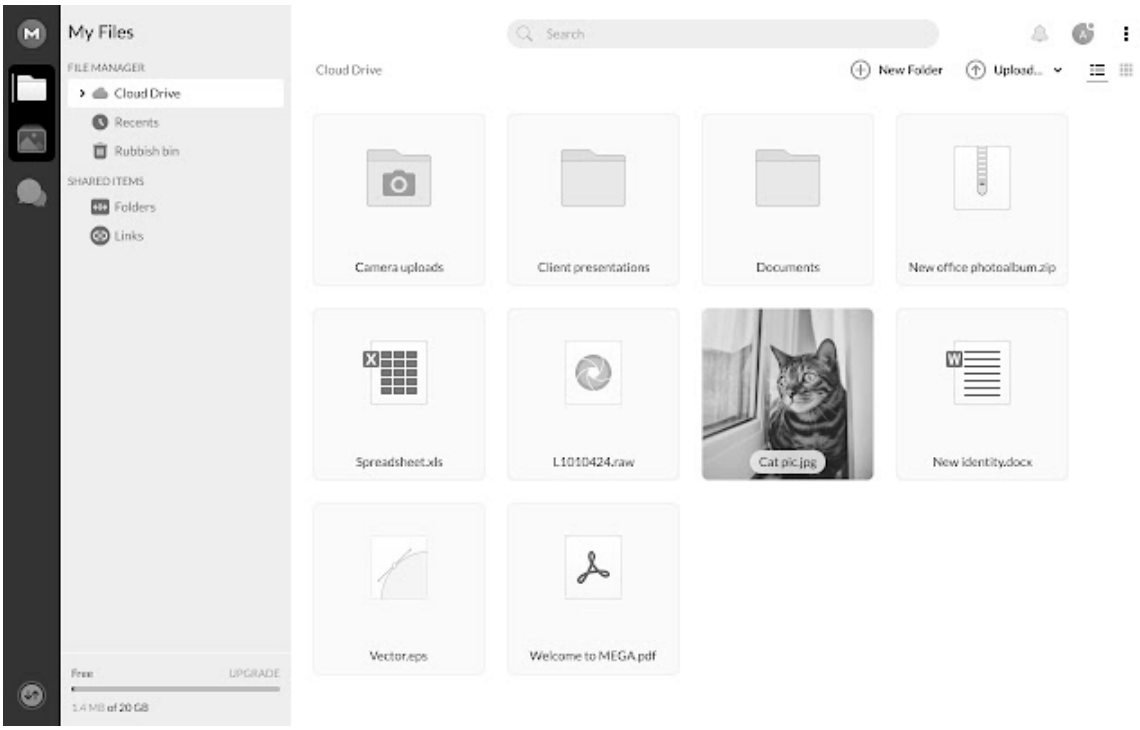

Рисунок 1.1 — Інтерфейс застосунку Mega

Система pCloud надає 10 ГБ безкоштовного простору з можливістю розширення до 15 ГБ за виконання нескладних дій, таких як підтвердження

електронної пошти або запрошення друзів (рис. 1.1). Важливою перевагою pCloud є підтримка потокового відтворення аудіо та відео, що робить сервіс привабливим для зберігання мультимедійного контенту. Ще однією особливістю є опція pCloud Crypto — платна функція клієнтського шифрування, яка забезпечує максимальний рівень приватності, коли лише користувач має ключ до своїх файлів.

Сервіс має підтримку всіх основних платформ: Windows, macOS, Linux, iOS, Android, що робить його універсальним інструментом для роботи на будь-якому пристрої. Користувачі можуть також придбати довічні плани, що є економічно вигідним для довготривалого користування. До переваг належать: зручність інтерфейсу, адаптивний дизайн і оптимізація для медіаконтенту. Серед недоліків — те, що шифрування нульового знання доступне лише за окрему плату, а швидкість завантаження може бути непостійною в деяких регіонах, що впливає на комфорт роботи з великими файлами [5].

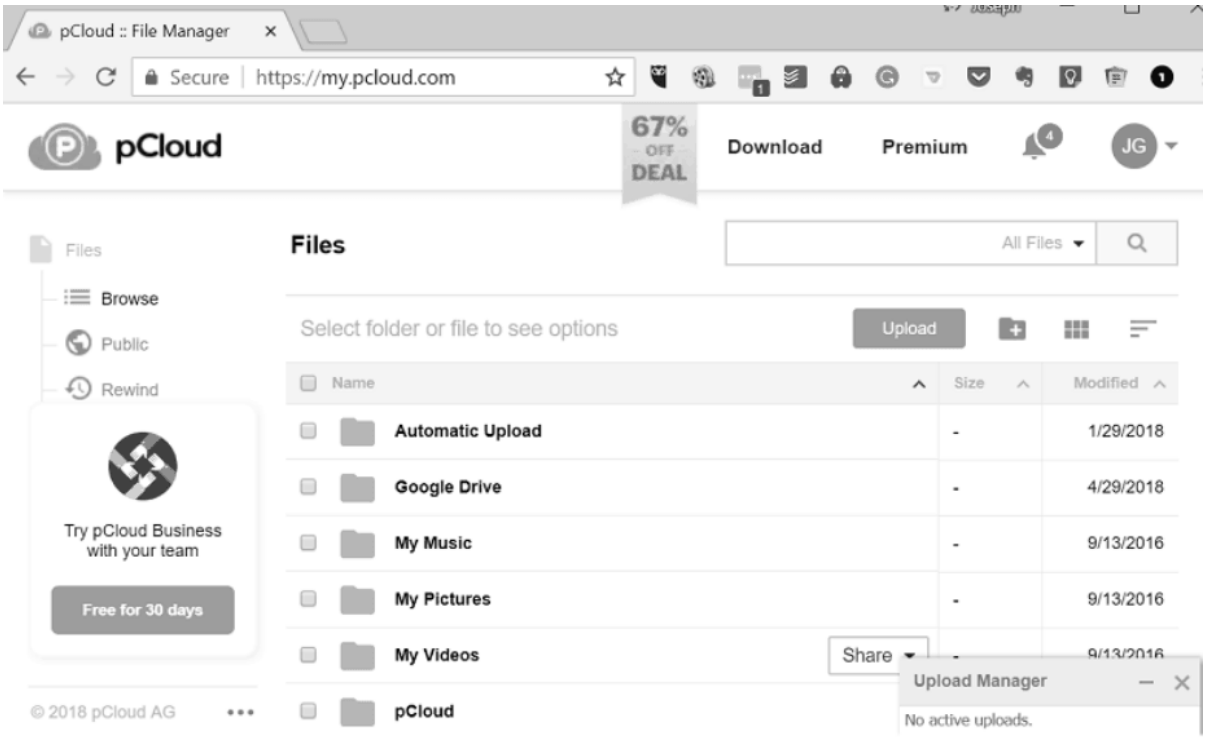

Рисунок 1.2 — Інтерфейс застосунку pCloud

Sync.com орієнтований насамперед на приватність і безпеку. Сервіс надає 5 ГБ безкоштовного простору, який можна розширити до 25 ГБ через реферальну програму (рис. 1.3). Його головною перевагою є zero-knowledge encryption

(шифрування нульового знання) — це означає, що файли не лише шифруються на боці користувача, а й компанія не зберігає жодних ключів, тому навіть адміністратори не можуть отримати до них доступ.

Платформа підтримує Windows, macOS, iOS та Android, але не має клієнта для Linux, що може бути важливо для певної категорії користувачів. У безкоштовному плані можна ділитися лише трьома папками, а також відсутні деякі розширені функції для командної роботи. Переваги Sync.com — це висока безпека, зрозумілий інтерфейс, і зручна синхронізація між пристроями. Проте обмежений безкоштовний обсяг, відсутність підтримки Linux і мінімальні можливості для співпраці у безкоштовному варіанті можуть бути критичними недоліками для деяких користувачів [6].

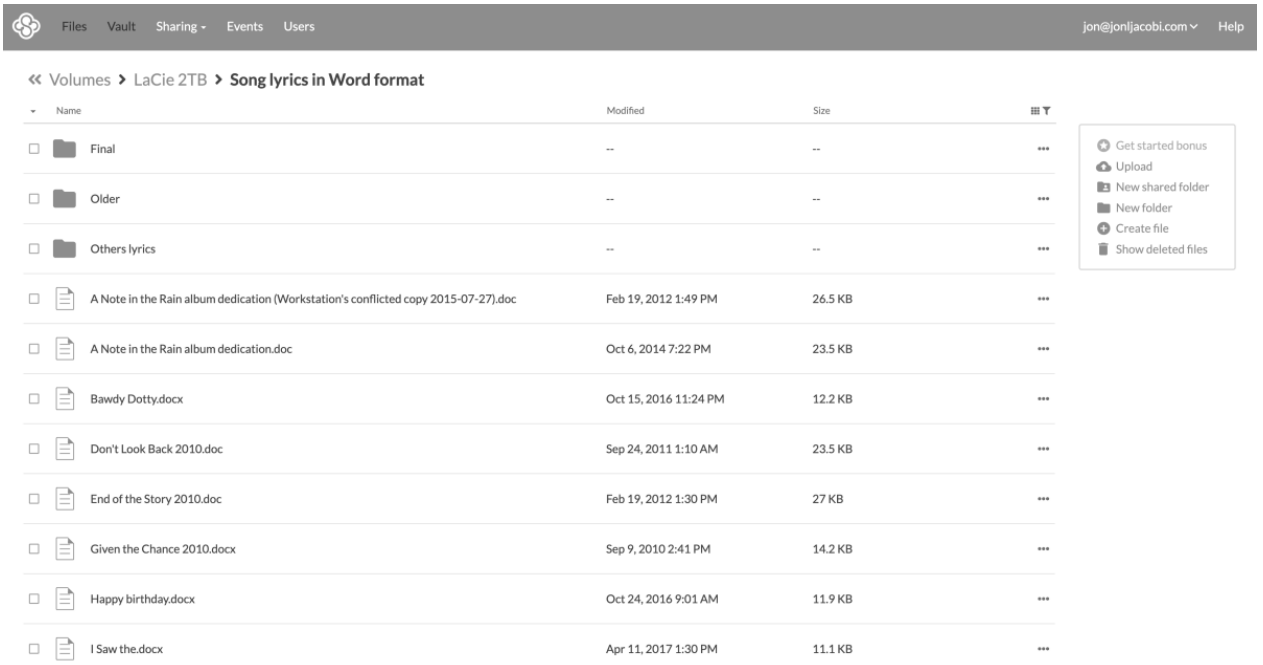

Рисунок 1.3 — Інтерфейс застосунку Sync.com

Icedrive — це порівняно новий гравець на ринку хмарних сховищ, який пропонує 10 ГБ безкоштовного простору. Шифрування AES-256 використовується для зберігання файлів, а в платних тарифах доступне шифрування Twofish із нульовим знанням (рис. 1.4). Це означає, що користувач може бути впевнений у повній конфіденційності збережених даних.

Платформа підтримує всі основні ОС та вирізняється інтуїтивно зрозумілим інтерфейсом із функцією перетягування файлів, що спрощує завантаження. Сервіс дозволяє переглядати файли без необхідності їх завантаження, що економить час і ресурси. Серед переваг — простота у використанні, захищеність файлів, адаптивність до різних пристроїв. Серед недоліків варто зазначити: шифрування нульового знання доступне лише у платних тарифах, відсутність історії версій у безкоштовному плані, а також менш розвинені функції у порівнянні з більш відомими конкурентами [7].

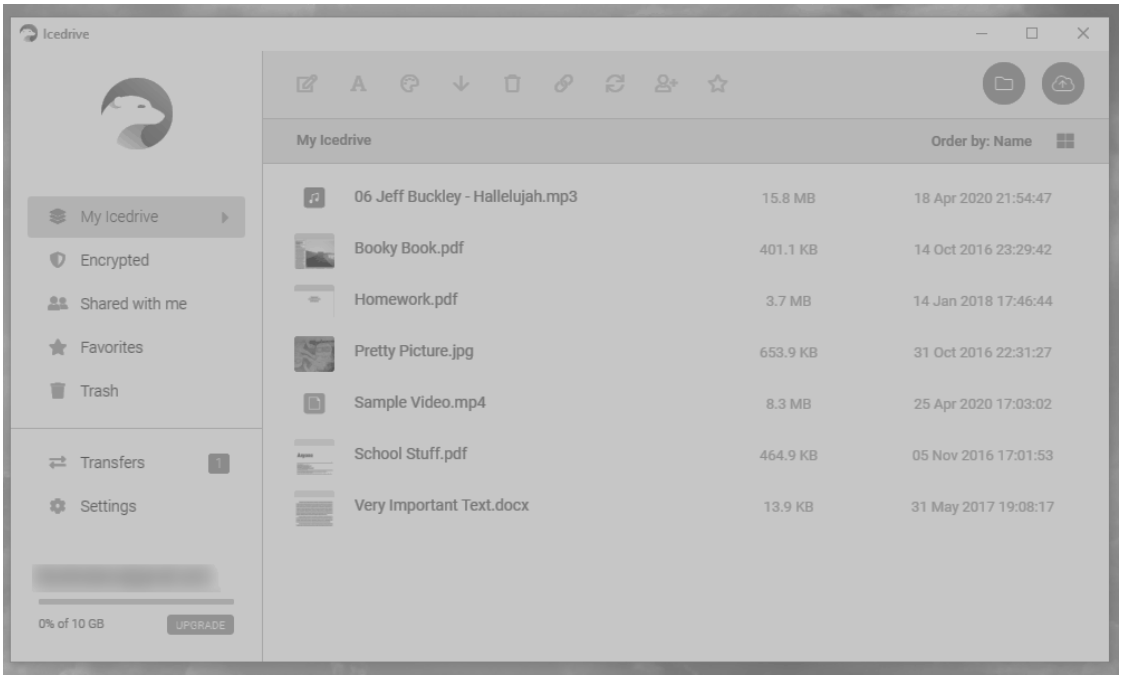

Рисунок 1.4 — Інтерфейс застосунку Icedrive

Proton Drive є частиною екосистеми Proton, відомої своїм фокусом на приватності та безпеці (рис. 1.5). Сервіс надає 5 ГБ безкоштовного простору з наскрізним шифруванням за замовчуванням, що гарантує, що файли не будуть доступні третім особам. Дані зберігаються на серверах у Швейцарії, країні з одними з найсуворіших законів про захист приватності у світі.

Сервіс має простий і зрозумілий інтерфейс, дозволяє зручно завантажувати документи й фотографії, а також має інтеграцію з іншими сервісами Proton, такими як Proton Mail, що спрощує роботу в єдиному захищеному середовищі. До переваг належать: високий рівень безпеки, розташування серверів у нейтральній

юрисдикції, простота у використанні. Проте Proton Drive має і свої обмеження: не надто великий безкоштовний обсяг, відсутність функцій потокового відтворення чи історії версій, що може зменшувати його зручність для користувачів із великими архівами даних.

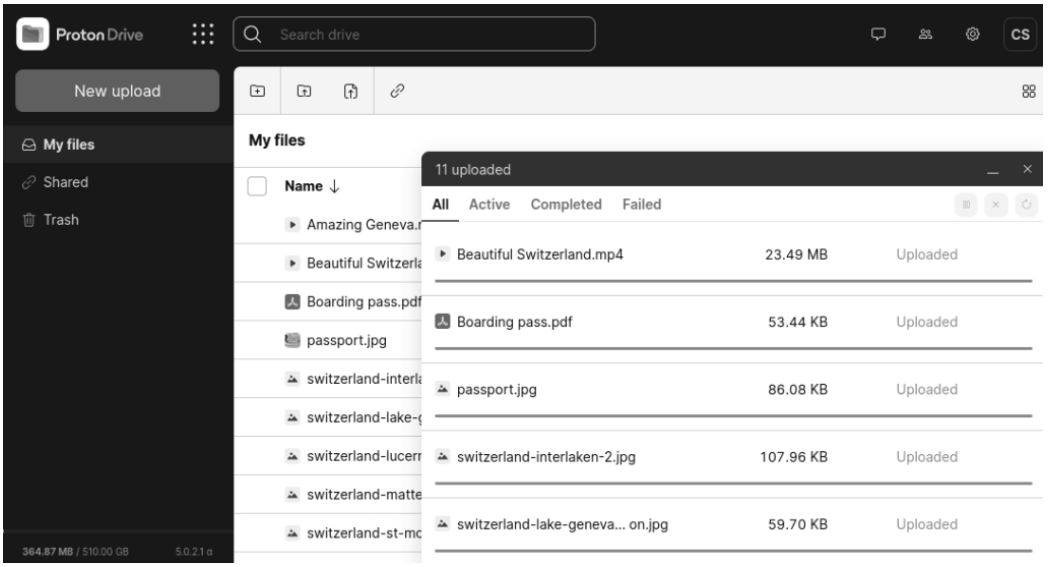

Рисунок 1.5 — Інтерфейс застосунку Proton Drive

2 ВИБІР ТА ОБГРУНТУВАННЯ ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ КОМП'ЮТЕРНОЇ СИСТЕМИ

2.1 Огляд та вибір бази даних для сховища

Бази даних є основою сучасних інформаційних систем, забезпечуючи структуроване зберігання, управління та обробку даних із акцентом на ефективність, цілісність і безпеку. Вони дозволяють організувати інформацію для швидкого доступу, масштабованості та надійності, що критично для застосунків від транзакційних систем до аналітичних платформ.

Реляційні бази даних, засновані на моделі таблиць, залишаються стандартом для систем із чітко визначеною структурою (рис. 2.1). Дані організовано у вигляді рядків і стовпців, пов'язаних через первинні та зовнішні ключі. SQL як мова запитів забезпечує універсальний доступ до даних, дозволяючи виконувати складні операції, такі як об'єднання таблиць чи агрегування. Нормалізація зменшує надлишковість, мінімізуючи аномалії під час вставки, оновлення чи видалення даних. Проте реляційні бази можуть бути обмеженими при роботі з неструктурованими даними чи горизонтальним масштабуванням.

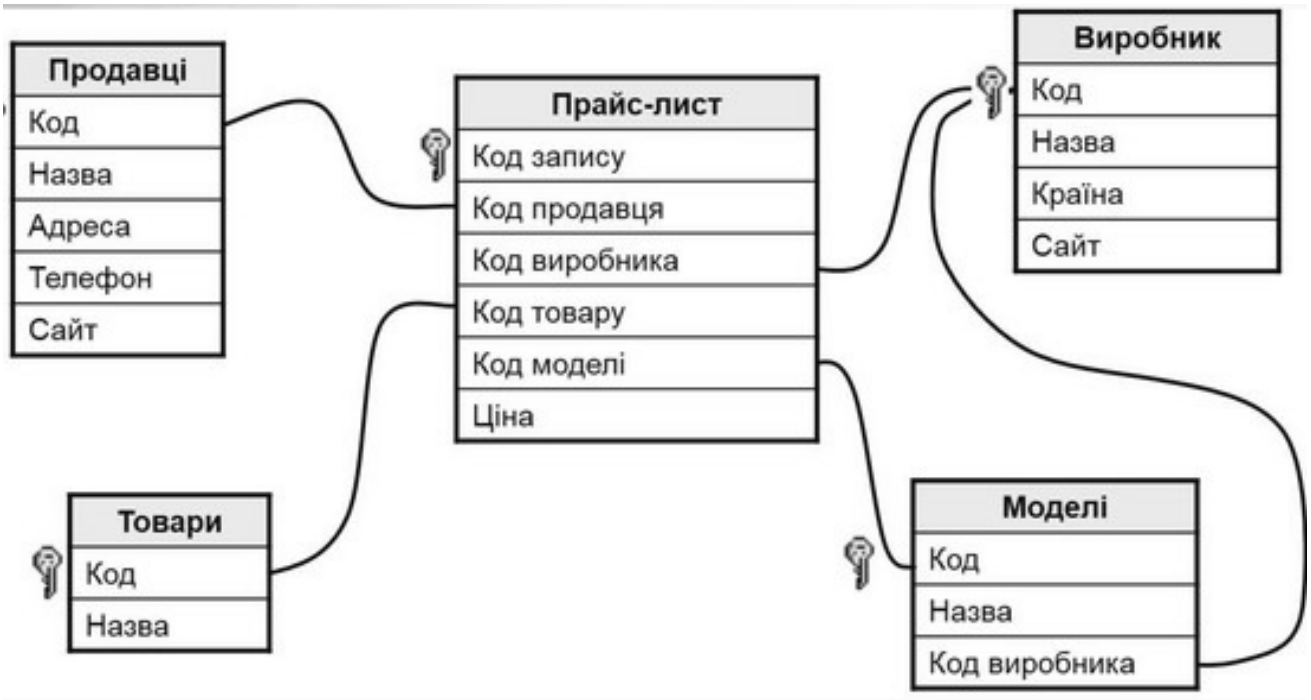

Рисунок 2.1 — Схема збереження даних в реляційній моделі

Нереляційні (NoSQL) бази даних розв’язують ці обмеження, пропонуючи гнучкість для великих обсягів даних і динамічних структур (рис. 2.2). Документоорієнтовані бази, як MongoDB, зберігають дані у форматі JSON або BSON, дозволяючи гнучко змінювати схему. Стовпцеві бази, наприклад Cassandra, оптимізовано для аналітики великих даних із високою доступністю. Графові бази, як Neo4j, ефективні для моделювання складних зв’язків, таких як соціальні мережі чи рекомендаційні системи. Бази типу ключ-значення, наприклад Redis, забезпечують високу швидкість для кешування чи сесійного зберігання [9].

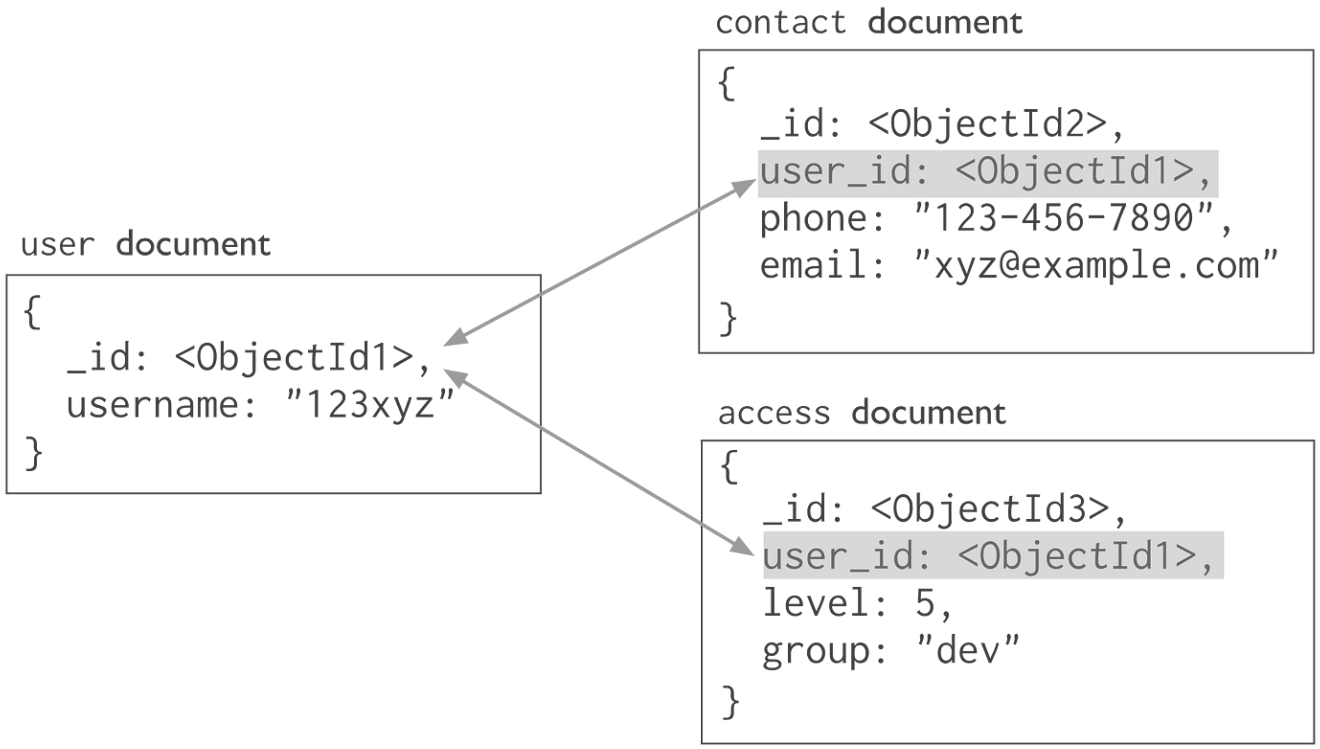

Рисунок 2.2 — Схема збереження даних в нереляційні базі даних MongoDB

Архітектура зберігання даних відіграє критичну роль в інформаційній системі онлайн-сервісу пошуку знайомств, оскільки від її характеристик залежить продуктивність, масштабованість та надійність у роботі з великою кількістю користувачів і транзакцій. Бази даних за архітектурним підходом поділяються на централізовані та розподілені.

Централізовані системи простіші в налаштуванні й обслуговуванні, проте

мають обмеження щодо масштабованості та відмовостійкості. Натомість розподілені бази даних, як-от DynamoDB або CockroachDB, дозволяють ефективно розподіляти дані між декількома вузлами, забезпечуючи високу доступність сервісу навіть при збої частини інфраструктури. Це особливо важливо для онлайн-платформ знайомств, де безперервна доступність та швидкий відгук системи є критичними.

У виборі архітектурного підходу варто враховувати CAP-теорему, яка формулює компроміси між трьома основними властивостями розподілених систем: консистентністю (Consistency), доступністю (Availability) та стійкістю до розділення (Partition tolerance). Наприклад, MongoDB у певних налаштуваннях віддає перевагу доступності, що може бути доцільним у соціальних застосунках, де важливо забезпечити безперервний користувацький досвід навіть за рахунок можливої затримки синхронізації. Натомість традиційні реляційні СУБД, як правило, обирають консистентність, що важливо для збереження цілісності профілів користувачів, повідомлень і транзакцій.

Окремий клас систем — сховища даних (data warehouses) — використовується для зберігання історичних даних та виконання аналітичних запитів. В інформаційній системі знайомств це дозволяє виявляти поведінкові патерни користувачів, ефективність алгоритмів підбору пар, оптимізувати рекомендаційні системи тощо. Рішення на основі Snowflake або Google BigQuery, які підтримують стовпцеву організацію даних і паралельну обробку, дозволяють проводити складний аналіз великих обсягів інформації. Такі системи легко інтегруються з BI-інструментами (наприклад, Tableau), що дозволяє створювати звіти та прогнози щодо зростання аудиторії, популярності функцій або рівня конверсії.

Для зберігання основних даних у розроблюваній інформаційній системі обрано PostgreSQL як основну СУБД. Цей вибір обґрунтований такими перевагами:

— підтримка високонавантажених транзакційних сценаріїв, що є типовим для обробки взаємодій між користувачами (лайки, повідомлення, перегляди анкет

тощо);

— розвинені засоби безпеки: рольова модель, контроль доступу, шифрування з'єднань — що є обов’язковим для захисту особистих даних користувачів;

— можливість масштабування як вертикально, так і горизонтально — за допомогою таких рішень, як Citus (розподілення навантаження по вузлах) чи TimescaleDB (ефективне зберігання тимчасових рядів активності);

— сумісність із сучасними серверними платформами, такими як .NET, а також з екосистемами Docker і Kubernetes, що дозволяє гнучко керувати розгортанням системи у хмарних середовищах [10].

Оскільки PostgreSQL добре інтегрується з сучасними серверними платформами, такими як dotnet і середовищем на основі Docker, її використання дозволяє досягти високої продуктивності та стабільності навіть за умов обмежених ресурсів. Саме ці характеристики є ключовими для задач, пов’язаних із зберіганням користувацької активності, забезпеченням гейміфікованого функціоналу, обробкою досягнень і підтримкою мікросервісної архітектури.

2.2 Огляд сучасних бекенд фреймворків

Серверна частина будь-якої сучасної інформаційної системи виконує роль посередника між зовнішніми клієнтами, такими як веббраузери, мобільні застосунки або десктопні програми, та внутрішніми модулями, які відповідають за зберігання, обробку або передачу даних. Вона виступає центральною ланкою в архітектурі програмного забезпечення, забезпечуючи безпечний, надійний і масштабований зв’язок між користувачем і системними ресурсами. Серверна частина приймає зовнішні запити, інтерпретує їх, виконує відповідну логіку та надсилає сформовану відповідь назад до клієнта. Її ефективність безпосередньо впливає на швидкість роботи системи, якість обслуговування запитів і загальний досвід користувача.

Ключову роль у створенні серверної частини відіграють вебфреймворки — спеціалізовані програмні інструменти, які надають готовий каркас для розробки серверної логіки. Вони значно спрощують процес розробки, зменшуючи потребу в

ручному налаштуванні базових функцій, таких як маршрутизація, обробка запитів, автентифікація користувача, логування, взаємодія з базами даних тощо. У сучасній практиці веброзробки більшість фреймворків реалізовані на асинхронній моделі обробки запитів, що дозволяє обслуговувати велику кількість одночасних підключень без блокування потоків, зменшуючи затримки та покращуючи масштабованість системи.

Важливою характеристикою сучасних вебфреймворків є підтримка middleware-компонентів — незалежних проміжних модулів, які дозволяють формувати гнучкий та налаштований ланцюг обробки HTTP-запиту. Кожен із таких модулів виконує певну функцію, наприклад автентифікацію, ведення журналу, обробку помилок або стиснення даних. Це дає змогу реалізувати чітку модульну архітектуру, у якій кожен компонент відповідальний за окрему частину логіки й може бути легко замінений, оновлений або повторно використаний.

Взаємодія між клієнтом і сервером у більшості випадків здійснюється за допомогою протоколів HTTP або HTTPS (рис. 2.3). Ці протоколи забезпечують стандартизований обмін повідомленнями між сторонами через мережу. При цьому для більш складних сценаріїв або потреб у реальному часі можуть використовуватися додаткові технології, зокрема WebSocket — для забезпечення двостороннього зв’язку між клієнтом і сервером без повторного відкриття з’єднання, або gRPC — для високопродуктивного міжсервісного спілкування у мікросервісних архітектурах [11-12].

Усі ці запити, незалежно від їхнього джерела чи формату, проходять через програмну інфраструктуру вебфреймворку. Саме він відповідає за обробку всіх етапів взаємодії: прийняття запиту, визначення маршруту, перевірку прав доступу, виконання відповідної логіки (наприклад, читання або запису даних до бази), обробку помилок і, нарешті, формування відповіді, яка надсилається назад до клієнта. Завдяки чіткій структурі та розділенню відповідальностей, сучасні фреймворки забезпечують гнучкість, масштабованість і безпеку серверної частини інформаційної системи.

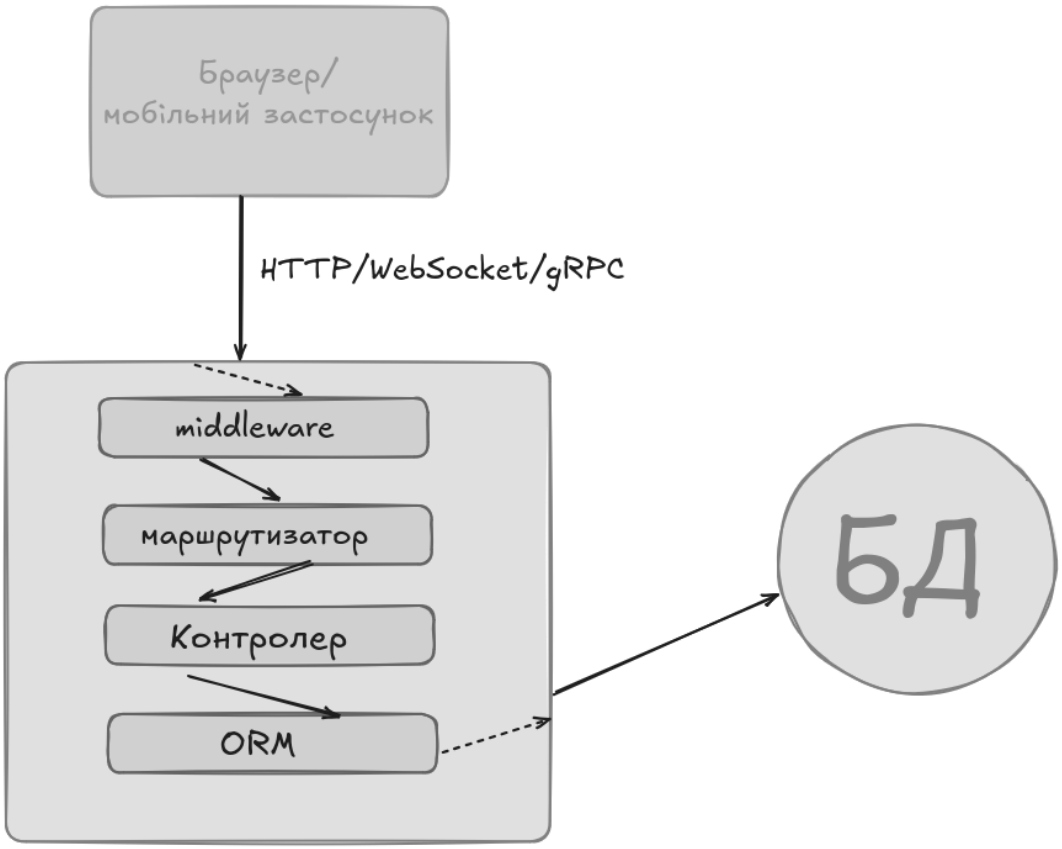

Рисунок 2.3 — Загальна схема роботи фреймворка

Одним з високопродуктивним асинхронним фреймворком є Fastify. Його архітектура дозволяє досягати високої пропускної здатності за рахунок використання внутрішнього HTTP-сервера, побудованого на низькорівневих оптимізаціях. Основний акцент зроблений на швидкості обробки JSON-запитів та розширюваності за допомогою плагінів. Структура Fastify дозволяє автоматично створювати документацію на основі JSON Schema, а також підтримує повну типізацію у зв’язці з TypeScript. Проте через динамічну природу JavaScript складні проєкти на Fastify можуть бути менш контрольованими у великих командах.

FastAPI — це сучасне рішення на базі Python, яке поєднує простоту синтаксису мови з потужністю типізації через Pydantic. Основна ідея фреймворку полягає у мінімалістичній розробці RESTful API з автоматичною генерацією Swagger-документації. Асинхронна модель виконання базується на asyncio, що дозволяє обслуговувати велику кількість одночасних запитів. FastAPI особливо популярний у науковому та аналітичному середовищі, де його часто

використовують у поєднанні з ML-модулями, однак інтерпретована природа Python може бути обмеженням для продуктивності у високонавантажених системах [13].

Gin — високошвидкісний фреймворк на базі мови Go. Завдяки компільованому коду та багатопоточності, яка реалізується через goroutines, Gin може обслуговувати тисячі запитів одночасно без втрати стабільності. Структура застосунку вимагає чіткої організації — кожен обробник, middleware, структура відповіді має бути чітко визначеним. Висока продуктивність робить Gin привабливим для створення мікросервісів, які потребують максимальної ефективності, але складність налаштування розширеної бізнес-логіки може ускладнювати підтримку великого проєкту [14].

Axum — фреймворк для мови Rust, який вирізняється суворою типобезпекою та повною асинхронністю. Він працює на основі Tokio та використовує Tower для побудови middleware-компонентів. Основна ідея Axum — гарантувати безпечність застосунку ще на етапі компіляції. Це досягається шляхом використання сильного типового контролю, що зменшує вірогідність помилок часу виконання. Однак вхідний поріг у Rust та складність читання коду можуть уповільнити розробку на ранніх етапах [15].

ASP.NET Core — універсальне середовище розробки серверних застосунків від Microsoft. Воно підтримує повну асинхронність, вбудовану систему маршрутизації, авторизації, middleware, логування, DI-контейнери, обробку помилок та інтеграцію з будь-якими базами даних через Entity Framework або Dapper. Його архітектура спроєктована з урахуванням модульності, підтримки REST, gRPC, SignalR та масштабованості в клауд-середовищах. Використання C# забезпечує високу надійність і зручність рефакторингу, а можливість розгортання на Windows, Linux та Docker підвищує гнучкість інфраструктури [16].

З урахуванням вимог, які передбачають реалізацію масштабованого API, підтримку гнучкої маршрутизації, обробки стану користувача, авторизації, з’єднання з PostgreSQL, підтримки контейнеризації та продуктивної роботи в продакшн-середовищі, для реалізації застосунку буде використовуватися ASP.NET

Core.

Перевагами ASP.NET Core у цьому контексті є стабільна компільована мова програмування, високий рівень абстракції без втрати контролю, інтеграція з інфраструктурними рішеннями на кшталт Docker, Kubernetes, Azure DevOps, а також наявність потужної екосистеми бібліотек, тестових фреймворків та редакторів коду. Це дозволяє будувати застосунки, які будуть не лише гнучкими у розробці, а й стабільними та масштабованими у майбутньому.

2.3 Огляд сучасних фронтенд фреймворків

Фронтенд-фреймворки відіграють ключову роль у сучасній веброзробці, оскільки вони забезпечують чітку структуру, повторно використовувані шаблони та інструменти для створення динамічних та інтерактивних клієнтських інтерфейсів. Вони значно спрощують процес розробки, особливо у великих проєктах, де важливо підтримувати масштабованість, модульність і зрозумілу архітектуру коду. Такі фреймворки як Vue.js, React або Angular дозволяють розробникам фокусуватись на бізнес-логіці замість рутинного керування DOM або обробки подій.

На відміну від традиційних багатосторінкових застосунків (MPA), де кожна взаємодія з інтерфейсом потребувала завантаження нової HTML-сторінки з сервера, сучасні односторінкові застосунки (SPA — Single Page Application) залишаються в межах однієї вебсторінки. У таких системах вміст динамічно оновлюється на основі дій користувача або змін даних, що значно покращує продуктивність і зменшує затримки при переході між екранами. Це сприяє формуванню більш плавного й природного користувацького досвіду (UX), а також дозволяє реалізувати складну логіку взаємодії на клієнтському боці.

Крім побудови інтерфейсу, фронтенд-фреймворки також виступають у ролі проміжної ланки між користувачем і сервером. Вони реалізують механізми авторизації та автентифікації, обробляють API-запити до серверної частини, зберігають дані у локальному кеші для підвищення швидкодії та реалізують маршрутизацію без повного перезавантаження сторінки. Це дозволяє створювати

комплексні вебзастосунки, які працюють майже з тією ж швидкістю, що й десктопні програми.

Сучасна фронтендна архітектура базується на концепції компонентів. Кожен компонент є автономним фрагментом інтерфейсу, який відповідає за конкретну функціональність — наприклад, відображення кнопки, форми або списку файлів. Компоненти можуть мати власний локальний стан, а для обміну даними між ними використовуються реактивні системи прив'язки даних або централізовані сховища стану (рис. 2.4). Це дозволяє ефективно організовувати взаємодію між різними частинами застосунку, уникаючи дублювання даних і некерованих залежностей.

Для отримання чи надсилання даних до сервера зазвичай використовуються HTTP-запити до REST API або GraphQL-ендпоінтів. У застосунках, де потрібна робота в режимі реального часу, підтримуються WebSocket-з’єднання, які забезпечують двосторонню комунікацію без повторного опитування сервера. Це особливо актуально для чатів, онлайн-редакторів або систем моніторингу.

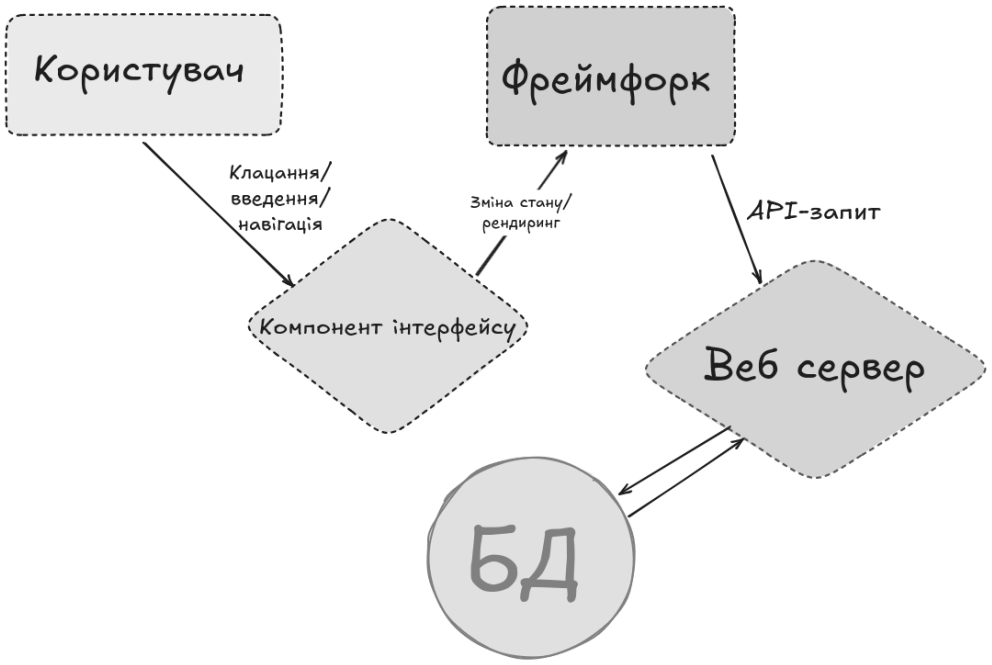

Рисунок 2.4 — Схема користувацької взаємодії

React — одна з найпопулярніших бібліотек для створення користувацького інтерфейсу. Вона побудована навколо концепції віртуального DOM, який дозволяє ефективно оновлювати інтерфейс при зміні стану. Компоненти React є

функціональними або класовими і взаємодіють через props та state. Реактивність досягається за допомогою хуків. React сам по собі не є повноцінним фреймворком — для побудови повного застосунку потрібне додаткове використання таких інструментів, як React Router, Redux, або Next.js для SSR. Гнучкість і велика екосистема роблять його вибором №1 у багатьох великих проєктах [17].

Vue — прогресивний фреймворк, орієнтований на простоту й поступове впровадження. У ньому використовується двостороннє зв’язування даних, шаблони HTML і гнучка система реактивності, яка автоматично оновлює DOM при зміні даних. Починаючи з Vue 3, додано Composition API, який дозволяє ефективно організовувати логіку та перевикористовувати код між компонентами. Vue ідеально підходить для малих та середніх застосунків, однак має достатній потенціал і для побудови великих проєктів з використанням Nuxt [18].

Angular — повноцінний фреймворк від Google, який базується на мові TypeScript і реалізує сувору модульну структуру. У ньому передбачено DI (Dependency Injection), сервіси, модулі, а також потужна система форм, валідацій та роутінгу. Angular забезпечує повну інфраструктуру для корпоративних SPA, але вимагає високого рівня дисципліни в розробці. Складна конфігурація і високий поріг входу компенсуються масштабованістю та розвиненими інструментами тестування [19].

Svelte — фреймворк, який підходить до побудови інтерфейсу не через інтерпретацію віртуального DOM, а шляхом компіляції компонентів у оптимізований JavaScript-код ще на етапі білду. Реактивність у Svelte вбудована безпосередньо в синтаксис, що робить код лаконічним і продуктивним. Завдяки відсутності runtime-надлишків, Svelte дозволяє створювати надзвичайно легкі застосунки. Його архітектура добре підходить для застосунків, де критичні розмір і швидкість [20].

Nuxt — фреймворк на базі Vue, який надає повну інфраструктуру для розробки універсальних (isomorphic) застосунків. Nuxt дозволяє використовувати серверний рендеринг (SSR), статичну генерацію (SSG), або звичайний SPA-режим. Він забезпечує автоматичну маршрутизацію, розбиття коду, мета-теги SEO,

модульну архітектуру з плагінами, підтримку SSR API-запитів через middleware, та інтеграцію з бекендом (REST або GraphQL). За допомогою Nuxt можна створювати швидкі, SEO-оптимізовані сайти, адаптовані під будь-які розгортання: як на сервері, так і у вигляді статичного хостингу.

Для реалізації цього проєкту було обрано Nuxt, оскільки він поєднує гнучкість Vue з готовою архітектурою фреймворку. Його особливістю є можливість працювати в режимах SSR, SPA або SSG, що дозволяє обрати оптимальний варіант для продуктивності та SEO. У цьому застосунку Nuxt використовується в режимі SPA, що дає змогу забезпечити плавну навігацію, реактивну взаємодію з API, кешування на клієнті та гнучку побудову інтерфейсу без перезавантаження сторінки.

Крім того, Nuxt забезпечує модульну організацію структури проєкту, автоматичну маршрутизацію на основі структури /pages, вбудовану підтримку middleware для захисту сторінок, централізоване зберігання стану (через Pinia або Vuex), та можливість підключення Tailwind CSS, що було використано у цьому проєкті для побудови адаптивного інтерфейсу [21].

2.4 Взаємодія між сервером та клієнтом

У сучасному вебзастосунку взаємодія між клієнтською та серверною частиною реалізується за моделлю запит–відповідь через протокол HTTPS. Користувач працює з інтерфейсом, що реалізований у вигляді SPA на базі фреймворку Nuxt. Цей інтерфейс є повністю клієнтським, але побудований так, щоб за потреби звертатися до серверного API, написаного на ASP.NET Core. Весь обмін даними відбувається у форматі JSON, а логіка доступу контролюється за допомогою HTTP cookie, які зберігають ідентифікатор користувача після входу в систему.

Весь процес взаємодії між користувачем і системою можна уявити як ланцюг, де кожна сторона виконує свою чітко визначену роль. Коли користувач натискає, наприклад, кнопку «Додати файл», клієнтська частина формує HTTP-запит. У цьому запиті міститься метод (POST), тіло з метаданими файлу, а

також cookie з ідентифікатором сеансу. Цей запит передається через мережу на сервер.

На сервері запит спершу обробляється маршрутизатором ASP.NET Core. Далі перевіряється автентичність користувача на основі cookie, валідуються дані, і лише після цього виконується збереження через ORM (наприклад, Entity Framework) до бази даних PostgreSQL. Після успішної обробки сервер формує відповідь — наприклад, підтвердження додавання або об’єкт із новим ID — і надсилає її назад на клієнт.

З боку клієнта роль виконує Nuxt — сучасний фронтенд-фреймворк, побудований на базі Vue.js. Він динамічно оновлює інтерфейс без перезавантаження сторінки. Для взаємодії з сервером Nuxt використовує механізми useFetch, useAsyncData або нативні HTTP-запити. Оскільки архітектура SPA (Single Page Application), запити мають бути асинхронними, щоб не блокувати взаємодію з інтерфейсом. Отримані відповіді обробляються у вигляді промісів, що дозволяє реактивно змінювати вміст на сторінці — наприклад, одразу відображати новий файл у списку.

Щоб уникнути повторної автентифікації при кожному запиті, після входу в систему сервер встановлює cookie з маркером сеансу. Ця cookie автоматично додається до всіх подальших запитів. Сервер перевіряє її на захищених маршрутах, перш ніж дозволити доступ до даних. Такий підхід дозволяє підтримувати безперервний сеанс без потреби передавати логін і пароль при кожній операції.

Таке чітке розмежування обов’язків між клієнтом і сервером підвищує гнучкість та масштабованість системи (рис. 2.5). Клієнт відповідає за швидкий і зручний інтерфейс, тоді як сервер гарантує збереження, безпеку й цілісність даних. Завдяки сучасним технологіям — таким як Nuxt на фронтенді та ASP.NET Core на бекенді — забезпечується швидке реагування, надійна робота з великими обсягами інформації та можливість подальшого розвитку системи без серйозних змін її архітектури.

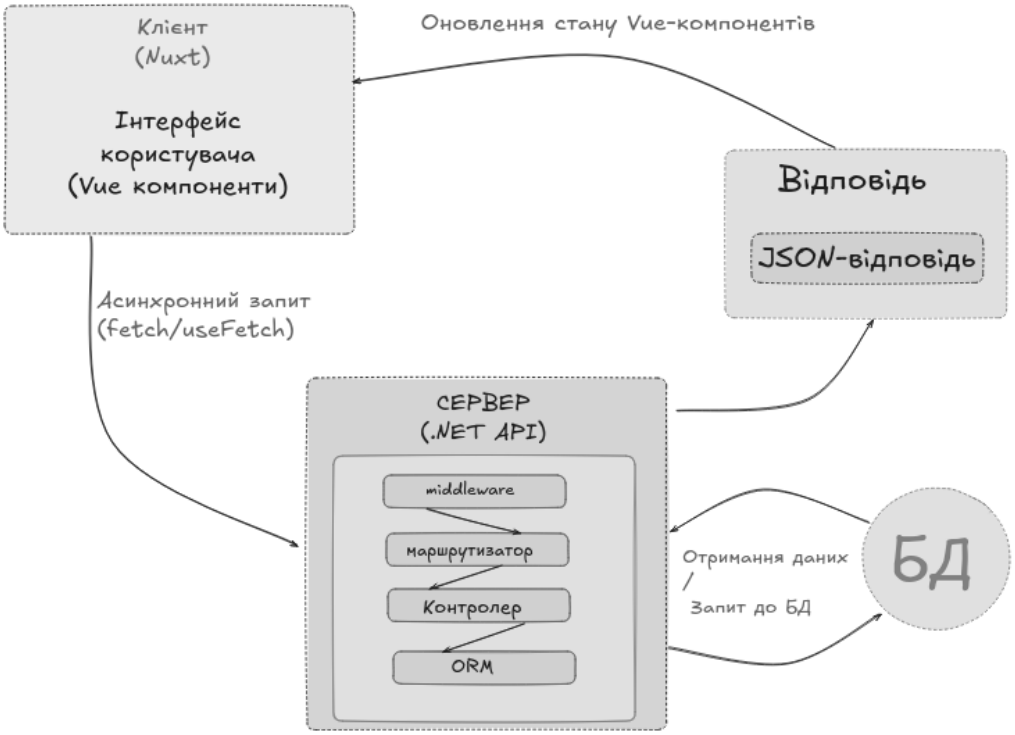

Рисунок 2.5 — Структурна схема взаємодії

2.5 Огляд сучасних текстових редакторів

У сучасному процесі розробки програмного забезпечення вибір редактора або інтегрованого середовища розробки (IDE) є критично важливим фактором, що безпосередньо впливає на продуктивність розробника, підтримку якості коду та швидкість доставки програмного продукту. Редактор або IDE виконує функцію посередника між програмістом та кодом, забезпечуючи не лише зручне введення, але й автоматизацію численних аспектів розробницького циклу.

Середовище розробки забезпечує програмісту наявність багатьох ключових функцій: підсвічування синтаксису, автодоповнення, перевірку типів у реальному часі, налагодження, інтеграцію з системами контролю версій, запуск тестів, взаємодію з базами даних, візуалізацію структури проєкту тощо. У поєднанні з протоколом LSP (Language Server Protocol), який стандартизує обмін інформацією між редактором і мовою програмування, сучасні редактори значною мірою уніфікували досвід розробки для різних мов.

Інтегровані середовища розробки, зокрема продукти JetBrains (рис. 2.6) та

Microsoft Visual Studio, пропонують найбільш повний набір інструментів. Вони надають розширену підтримку рефакторингу, статичного аналізу коду, профілювання, налагодження в реальному часі, управління залежностями, CI/CD інтеграції, запуску юніт-тестів та іншого.

Висока щільність функціоналу робить ці середовища оптимальними для розробки складних систем: корпоративного програмного забезпечення, багаторівневих додатків, мікросервісної архітектури. Проте така потужність супроводжується значним споживанням ресурсів, що обмежує ефективність на малопотужних пристроях. Також вартість комерційних IDE може бути значною, хоча багато продуктів мають безкоштовні освітні чи open-source ліцензії.

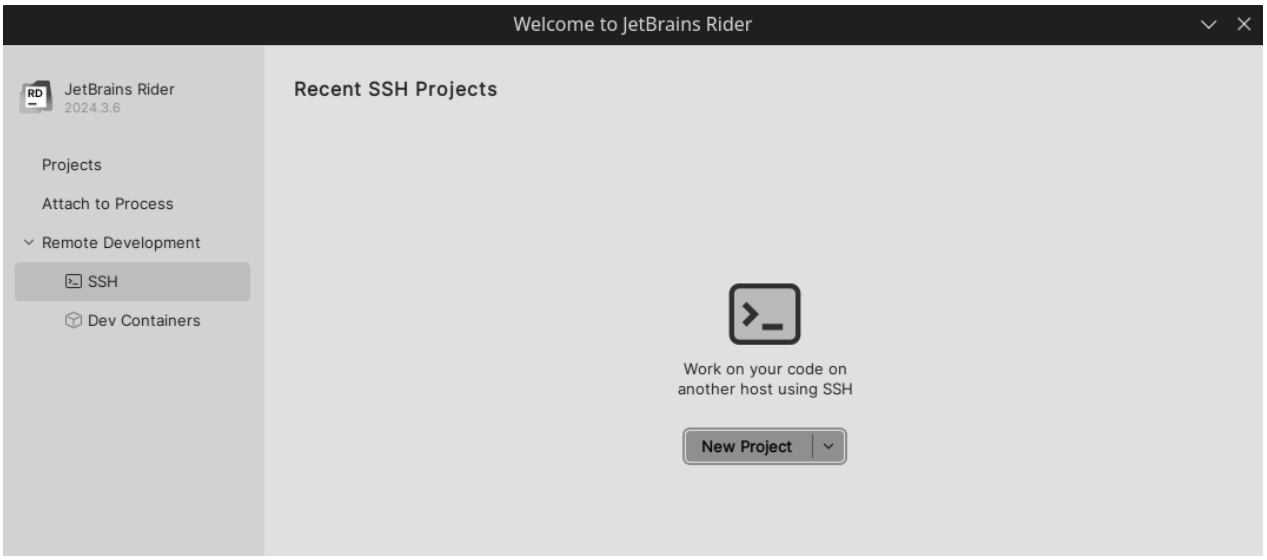

Рисунок 2.6 — JetBrains Rider IDE

Visual Studio Code, розроблений Microsoft, став де-факто стандартом легких, але потужних редакторів (рис. 2.7). Підтримка LSP, багатотисячна екосистема розширень, інтегрований термінал і підтримка налагодження роблять його універсальним інструментом для широкого спектра завдань. Через своє походження він ідеально підходить для веброзробки, розробки на JavaScript, Python, Go, Rust та C#.

Sublime Text — ще один приклад високопродуктивного редактора, що демонструє виняткову швидкодію навіть на старому обладнанні. Проте функціонал

за замовчуванням обмежений, і ефективне використання потребує встановлення численних плагінів.

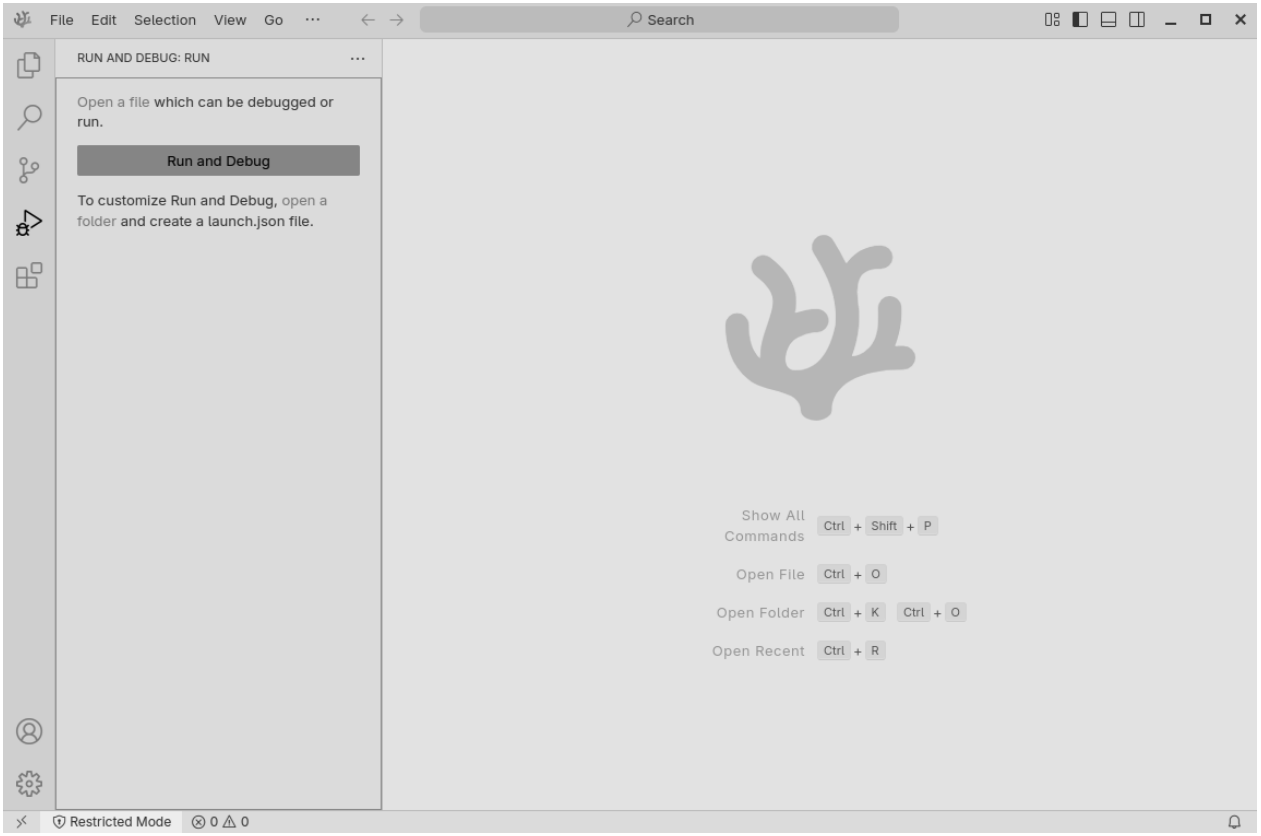

Рисунок 2.7 — Текстовий редактор Visual Studio Code

Консольні редактори такі як Vim, Neovim та Emacs мають іншу філософію — вони мінімалістичні за замовчуванням, але глибоко налаштовувані. Vim та його модернізований варіант Neovim забезпечують максимальну швидкодію, повністю клавіатурне керування та низьке споживання ресурсів (рис. 2.8). Завдяки плагінам вони можуть бути перетворені на повноцінні IDE з підтримкою десятків мов.

Ці редактори зазвичай використовуються розробниками, які працюють на серверних системах, у сфері наукових обчислень або прихильниками філософії "розширювального редактора як операційної системи".

Незалежні редактори, такі як Zed (рис. 2.9), Lite, Lapce прагнуть поєднати мінімалізм і швидкість з розширеністю через підтримку LSP або WebAssembly. Наприклад, Zed розробляється авторами Atom з фокусом на спільній розробці в реальному часі. Lapce, написаний на Rust, є нативним і зосереджується на

надшвидкій обробці великих проєктів.

Ці редактори ще не досягли зрілості традиційних рішень, але активно розвиваються та можуть стати основою нової генерації інструментів для розробників. Порівняльні характеристики всіх наведених редакторів наведено в таблиці 2.1.

Рисунок 2.8 — Консольний редактор Neovim

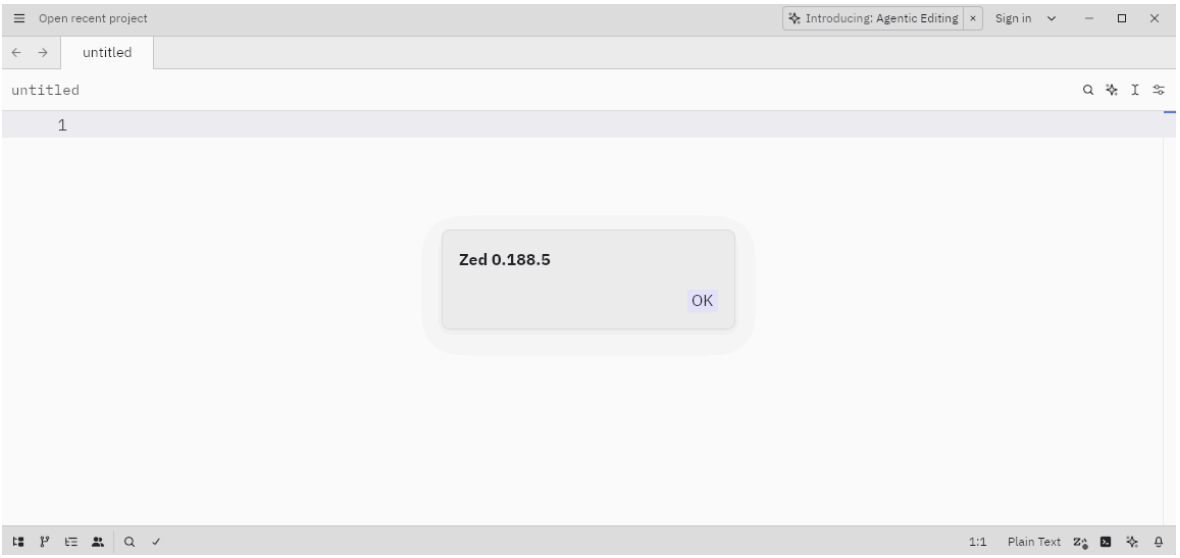

Рисунок 2.9 — Текстовий редактор Zed

Таблиця 2.1— Порівняння популярних редакторів

Редактор / IDE	Тип середовища	Швидкодія запуску	Підтримка LSP	Можливості для командної розробки	Ступінь налаштовуваності	Підходить для галузей
IntelliJ IDEA	Комерційна IDE	Низька	Вбудована	Висока	Середня	Java, Kotlin, Android,
Rider	Комерційна IDE	Низька	Вбудована	Висока	Середня	C#, F#, WebApi
VS Code	Легкий редактор	Висока	Через плагіни	Середня	Висока	Веб, JS, TS, Python,
Neovim	Консольний редактор	Дуже висока	Через плагіни	Низька (CLI)	Дуже висока	DevOps, системне адміністрування
Emacs	Консольна середа	Середня	Через пакети	Середня	Дуже висока	R&D, наука, автоматизація
Sublime Text	Легкий редактор	Висока	Через плагіни	Низька	Середня	Невеликі проєкти,
Zed	Незалежний редактор	Висока	Часткова	Висока (спільна розробка)	Низька	Веб, JS, TS, Python,
Lapce	Незалежний редактор	Висока	Вбудована	Обмежена	Середня	Rust, проєкти з високою продуктивністю

3 РОЗРОБКА ВЕБ ЗАСТОСУНКУ

3.1 Розробка структури бази даних та початкова конфігурацію вебсервера

Початковий етапом проєктування системи є визначення основних функціональних вимоги до роботи з файлами та користувачами. Основа структури бази даних, реалізована з використанням Entity Framework Core, ORM-інструменту, що дозволяє працювати з базою даних у вигляді об'єктів .NET.

Для розгортання бази даних PostgreSQL у контейнері Docker використовується файл compose.yaml, який містить опис служби db.

```
services:
 postgres:
   image: postgres:15
   container_name: dbFileY
   environment:
     POSTGRES_USER: ymonada
     POSTGRES_PASSWORD: 111
     POSTGRES_DB: postgres
   ports:
     - "5445:5432"
   volumes:
     - postgres_data:/var/lib/postgresql/data
   networks:
     - backend_network
volumes:
 postgres_data:
networks:
 backend_network:
```

Цей файл забезпечує автоматичне створення контейнера бази даних із заданими параметрами доступу та збереженням даних у volume pgdata, що гарантує збереження інформації навіть після зупинки або видалення контейнера. Щоб запустити базу даних потрібно виконати команду docker-compose up -d, після чого можна буде підключитися до неї.

База даних складається з двох основних сутностей: User і FileY. Сутність User зберігає дані про користувачів, включаючи ідентифікатор, логін, email, хеш паролю та колекцію пов’язаних файлів (рис. 3.1). Сутність FileY описує збережені файли, містить посилання на користувача, а також назву, тип, шлях, розмір і дату завантаження. Зв'язок між файлами та користувачами є відношенням

один-до-багатьох: один користувач може мати багато файлів.

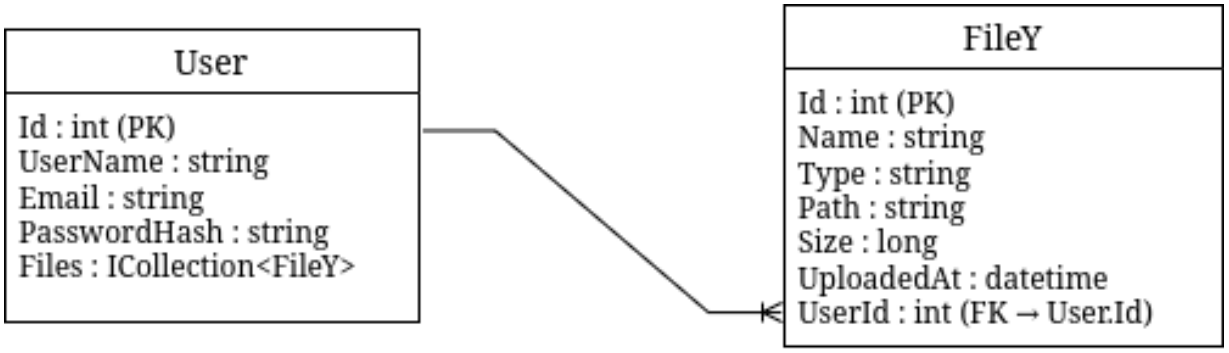

Рисунок 3.1 — Схема бази даних

Для реалізації взаємозв’язків та формування таблиць у базі даних використано механізм міграцій Entity Framework. Початкову міграцію створено через команду dotnet ef migrations add Init1, після чого база була оновлена командою dotnet ef database update.

Початковий проєкт було створено як ASP.NET Core Web API за допомогою команди dotnet new webapi. У файлі Program.cs реалізовано стандартну конфігурацію застосунку, що включає реєстрацію сервісів, контролерів, а також middleware-компонентів. Підключення до бази даних PostgreSQL визначено у конфігураційному файлі appsettings.json:

```
"Logging": {
  "LogLevel": {
    "Default": "Information",
    "Microsoft.AspNetCore": "Warning"
  }
 },
 "ConnectionStrings": {
        "PostgresConnection":   "Host=localhost;Port=5445;Database=dbFileY;Username=dddd;
Password=111" }
```

Конфігурація моделі здійснюється через окремі класи конфігурацій. Наприклад, конфігурація для сутності User у файлі UserConfiguration.cs може виглядати наступним чином:

```
public class UserConfiguration : IEntityTypeConfiguration<User>{
public void Configure(EntityTypeBuilder<User> builder)    {
        builder.HasKey(u => u.Id);
```

```
        builder.Property(u => u.UserName).IsRequired().HasMaxLength(100);
        builder.Property(u => u.Email).IsRequired().HasMaxLength(150);
        builder.Property(u => u.PasswordHash).IsRequired();
        builder.HasMany(u => u.Files)
        .WithOne(f => f.User)
        .HasForeignKey(f => f.UserId);        }}
```

Аналогічно конфігурується і сутність FileY. Ці конфігурації підключаються до контексту бази даних у класі AppDbContext:

```
public class AppDbContext : DbContext{
public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
public DbSet<User> Users => Set<User>();
public DbSet<FileY> Files => Set<FileY>();
protected override void OnModelCreating(ModelBuilder modelBuilder)    {
        modelBuilder.ApplyConfiguration(new UserConfiguration());
        modelBuilder.ApplyConfiguration(new FileYConfiguration());      }}
```

Під час запуску застосунку виконується реєстрація сервісів, зокрема контексту бази даних:

```
builder.Services.AddDbContext<AppDbContext>(options =>
options.UseNpgsql(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Дані налаштування забезпечують початкове налаштування проекту для подальшої роботи.

3.2 Розробка та налаштування основної логіки завантаження та збереження файлів

У межах архітектури даного вебзастосунку використано патерн «Репозиторій» (Repository Pattern), який виступає посередником між бізнес-логікою (сервісами) та джерелом даних (у даному випадку — базою даних через Entity Framework Core). Основне призначення репозиторію полягає в тому, щоб приховати всі деталі доступу до БД за єдиним, стандартизованим API, і при цьому залишити бізнес-логіку незалежною від конкретної реалізації зберігання.

У цьому проєкті реалізовано два основні репозиторії:

— UserRepository для роботи з користувачами;

— FileYRepository для управління файлами, пов’язаними з користувачами.

Кожен із них реалізує відповідний інтерфейс (IUserRepository, IFileYRepository), що дозволяє легко замінювати або тестувати реалізації.

UserRepository дозволяє здійснювати базові CRUD-операції над об’єктами

типу User. Наприклад, метод CreateAsync створює нового користувача в базі.

```
public async Task<User> CreateAsync(User user, CancellationToken ct = default) {
    await  _context.Users.AddAsync(user, ct);
    await _context.SaveChangesAsync(ct);
    return user;  }
```

Інші методи забезпечують пошук користувача за ID чи email, перевірку унікальності email, оновлення даних користувача або видалення.

```
public async Task<User?> GetByIdAsync(int id, CancellationToken ct = default)  {
        return await _context.Users
          .AsNoTracking()
          .FirstOrDefaultAsync(u => u.Id == id, ct);  }
```

Використання AsNoTracking при формуванні запиту, дозволяє досягти краща продуктивність при читанні, оскільки Entity Framework не відстежує зміни в об'єкті після отримання.

FileYRepository керує сутністю FileY, яка представляє файл, завантажений користувачем. На відміну від користувачів, з файлами проводяться складніші операції — наприклад, сторінкова вибірка (GetPageAsync), сортування за параметрами, пошук за збереженим шляхом, фільтрація за частиною назви.

```
public async Task<List<FileY>> GetPageAsync(int userId,
    int page = 1,
    int pageSize = 15,
    string? searchTerm = null,
    string? sortBy = null,
    string? sortOrder = "asc",
    CancellationToken ct = default)   {
    var query = _context.Files
      .AsNoTracking()
      .Where(f => f.UserId == userId);
    if (!string.IsNullOrWhiteSpace(searchTerm)){
      var search = searchTerm.ToLower();
      query = query.Where(f =>
        f.FileName.ToLower().Contains(search));}
    if (!string.IsNullOrWhiteSpace(sortBy)) {
      var selector = GetSortSelector(sortBy);
      query = sortOrder?.ToLower() == "desc"
        ? query.OrderByDescending(selector)
        : query.OrderBy(selector);  }
    else{
      query = query.OrderBy(f => f.Id); }
    return await query
```

```
        .Skip((page - 1) * pageSize)
        .Take(pageSize)
        .AsNoTracking()
        .ToListAsync(cancellationToken: ct);}
```

Суттєвим моментом є реалізація сортування через метод GetSortSelector, який повертає Expression<Func<FileY, object>> залежно від параметру sortBy. Це дозволяє динамічно сортувати запити без дублювання коду.

Також тут реалізовано перевірку володіння файлом (IsOwnerAsync), що дозволяє сервісу безпечно визначити, чи має користувач доступ до конкретного файлу.

Сервісний шар у даній архітектурі відповідає за реалізацію бізнес-логіки застосунку. Він оперує DTO-об'єктами, забезпечує обробку вхідних даних, делегує запити до репозиторіїв і повертає уніфіковану відповідь у вигляді об'єкта ServiceResult<T>. Це дозволяє зберігати чистоту контролерів, де відсутня логіка обробки, і забезпечити кращу модульність.

AuthService — сервіс, що реалізує логіку аутентифікації користувача. Його основна функція — реєстрація, логін, вихід та перевірка аутентифікації. Під час реєстрації сервіс перевіряє, чи є email унікальним, хешує пароль через CryptService, створює нового користувача і формує ClaimsPrincipal, який зберігається у cookie.

```
public async Task<ServiceResult<ClaimsPrincipal>> RegisterUserAsync(
    UserCreateDto userCreate,
    CancellationToken ct = default) {
    if (!await _userRepository.IsEmailUniqueAsync(userCreate.Email, ct)) {
        return new ServiceResult<ClaimsPrincipal>(
            false,
            "User with this email already exists!");    }
    var newUser = new User    {
        Email = userCreate.Email,
        PasswordHash = CryptService.HashPassword(userCreate.Password),
        UserName = userCreate.UserName  };
    await _userRepository.CreateAsync(newUser, ct);
    var claimsPrincipal = CreateClaimsPrincipal(newUser);
    await SignInAsync(claimsPrincipal);
    return new ServiceResult<ClaimsPrincipal>(
        true,
        "User created successfully!",
        claimsPrincipal); }
```

Метод LoginUserAsync аналогічно виконує пошук користувача за email та перевірку паролю, після чого також викликає SignInAsync, що дозволяє користувачу отримати доступ до своїх файлів.

UserService с відповідає за всі операції, пов’язані з користувачами. Він реалізує такі методи: отримання користувача за ID, сторінкова вибірка користувачів, створення нового користувача, оновлення даних, видалення користувача. Наприклад, метод GetUserAsync перевіряє наявність користувача в БД і повертає відповідний результат.

```
public async Task<ServiceResult<User>> GetUserAsync(int userId, CancellationToken ct = default) {
    var res = await _userRepository.GetByIdAsync(userId, ct);
    return res is null
       ? new ServiceResult<User>(false, "Not found User")
       : new ServiceResult<User>(true, "User", res); }
```

Також реалізовані решта методів для оновлення видалення чи отримання користувачів.

FileYService найбільш функціонально насичений сервіс, який обробляє завантаження файлів, їхню сторінкову вибірку, видалення, перегляд за ідентифікатором чи шляхом. Метод LoadFileArrayAsync приймає список файлів, хешує їхній вміст для генерації структури директорій, зберігає на диск та записує метадані у базу:

```
string basePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "files");
using var md5 = System.Security.Cryptography.MD5.Create();
byte[] hash = await md5.ComputeHashAsync(file.OpenReadStream());
int level1 = hash[0] % 256 % 10;
int level2 = hash[1] % 256 % 10;
string nestedDirectory = Path.Combine(basePath, level1.ToString(), level2.ToString());
```

Фізично файл зберігається у визначеній вкладеній директорії, а до БД записується шлях, тип, розмір та дата завантаження.

```
public async Task<ServiceResult<List<FileY>>> LoadFileArrayAsync(int userId, List<IFormFile> files, CancellationToken ct = default){
    string basePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "files");
    var userFiles = new List<FileY>();
    foreach (var file in files{
       using var md5 = System.Security.Cryptography.MD5.Create();
```

```
                using var stream1 = file.OpenReadStream();
                byte[] hash = await md5.ComputeHashAsync(stream1);
                var level0 = Guid.NewGuid();
                int level1 = hash[0] % 256 % 10;
                int level2 = hash[1] % 256 % 10;
                string nestedDirectory = Path.Combine(basePath
                   , level1.ToString()
                   , level2.ToString());
                if (!Directory.Exists(nestedDirectory)) {
                   Directory.CreateDirectory(nestedDirectory);  }
                var newFileName = file.FileName.Replace(' ', '_');
                var filePath = Path.Combine(nestedDirectory, newFileName);
                        var newFilePath = Path.Combine("files", level1.ToString(), level2.ToString(),
file.FileName.Replace(' ', '_'));
                await using var stream = new FileStream(filePath, FileMode.Create);
                await file.CopyToAsync(stream);
                userFiles.Add(new FileY()  {
                   UserId = userId,
                   FileName = newFileName,
                   FilePath = newFilePath,
                   ContentType = file.ContentType,
                   UploadedAt = DateTime.UtcNow,
                   FileSize = file.Length, });  }
             if (userFiles.Count < 1)      {
                return new ServiceResult<List<FileY>>(false, "No files");    }
             var res = await _fileRepository.CreateRangeAsync(userFiles, ct);
             return new ServiceResult<List<FileY>>(true, "Uploaded files", res);    }
```

Метод GetFilePageAsync викликає GetPageAsync з репозиторію, дозволяючи реалізувати сторінкову навігацію з фільтрацією та сортуванням.

Сервісний рівень у проєкті не тільки виконує проміжну роль, а й зосереджує усю складну логіку обробки. Це дозволяє зробити контролери «тонкими», де вся відповідальність — лише маршрутизація та відповіді. Завдяки використанню ServiceResult<T> також забезпечується уніфікація помилок і відповідей по всьому застосунку.

Контролери формують зовнішній інтерфейс застосунку, приймають HTTP-запити, викликають відповідні сервіси й повертають клієнту стандартизовані відповіді. Реалізовано три основні контролери: AuthController, UserController, FileController

Контролер автентифікації реалізує логіку реєстрації, входу, перевірки аутентифікації та виходу користувача. Він використовує сервіс IAuthService, який відповідає за основну бізнес-логіку.

Метод RegisterUserAsync приймає DTO з даними нового користувача, яке складається з ім’я, email, пароля, викликає метод сервісу для реєстрації і повертає результат. Подібним чином працює LoginUserAsync, лише він перевіряє введені облікові дані та створює сесію.

Метод CheckAuth використовується для клієнтської перевірки наявності автентифікованого користувача — він повертає true або false залежно від наявності ClaimTypes.NameIdentifier.

Контролер UserController відповідає за отримання інформації про поточного автентифікованого користувача. Він використовує IUserService, щоб отримати дані користувача з бази.

Метод GetCurrentUser виконує перевірку claims, отримує ID користувача й повертає DTO-об'єкт з інформацією.

```
[Authorize]
  [HttpGet("/user")]
  public async Task<IActionResult> GetCurrentUser()    {
     var userId = GetUserId();
     if (userId == 0)
        return NotFound();
     var res = await _userService.GetUserAsync(userId);
     if (!res.IsSuccess)
        return NotFound(res.Message);
     return Ok(UserMapper.ToDto(res.Data));    }
```

FileController є основним у проєкті й реалізує всю логіку роботи з файлами: завантаження, перегляд, сторінкова вибірка, отримання за ID або URL або підрахунок файлів.

Метод UploadFilesAsync приймає список файлів типу IFormFile, надісланих через multipart/form-data. Він передає їх у сервіс LoadFileArrayAsync, який фізично зберігає їх на диск, генерує структуру директорій та створює записи в базі.

GetFileUrlAsync — завантаження файлу за URL-шляхом дозволяє отримати файл напряму з файлової системи, якщо відомий його URL (наприклад, шлях збереження /files/3/4/document.pdf).

Метод GetFileAsync дозволяє отримати файл за унікальним ідентифікатором. Метод викликається, коли користувач хоче переглянути інформацію про

конкретний файл.

GetPageFilesAsync — сторінкова вибірка, пошук і сортування, що дозволяє клієнту отримати список файлів із фільтрацією, сортуванням і розбиттям на сторінки, де:

— searchTerm дозволяє шукати файли за частиною назви;

— sortBy дозволяє сортувати за назвою, датою або розміром файлів;

— sortOrder дозволяє сортувати за збільшенням або за зменшенням.

```
public async Task<ServiceResult<List<FileY>>> GetFilePageAsync(int userId,
    int page,
    int pageSize,
    string? searchTerm = null,
    string? sortBy = null,
    string? sortOrder = "asc",
    CancellationToken ct = default){
    if (page < 1)
        page = 1;
    if (pageSize < 1)
        pageSize = 10;
     var res = await _fileRepository.GetPageAsync(userId, page, pageSize, searchTerm, sortBy,
sortOrder, ct);
    return new ServiceResult<List<FileY>>(true, "files", res);}
```

Ці параметри далі передаються у репозиторій, де будується динамічний LINQ-запит.

```
[Authorize]
  [HttpGet("/file/page/")]
  public async Task<IActionResult> GetPageFilesAsync([FromQuery] int page,
    int pageSize,
    string? searchTerm = null,
    string? sortBy = null,
    string? sortOrder = "asc", CancellationToken ct = default) {
    var res = await _fileYService.GetFilePageAsync(GetUserId(), page, pageSize, searchTerm,
sortBy, sortOrder, ct);
    return res.IsSuccess ? Ok(res.Data) : BadRequest(res.Message);   }
```

Метод GetFilesCount дозволяє отримати число всіх файлів, завантажених поточним користувачем.

```
public async Task<ServiceResult<long>> GetFilesCount(int userId, CancellationToken ct =
default)   {
    var res = await _fileRepository.GetFileCountAsync(userId, ct);
    return new ServiceResult<long>(true, "Files", res);  }
```

Кінцеву схему роботи та конфігурацію веб сервера, можна зобразити за допомогою схеми, показано на рисунку 3.2.

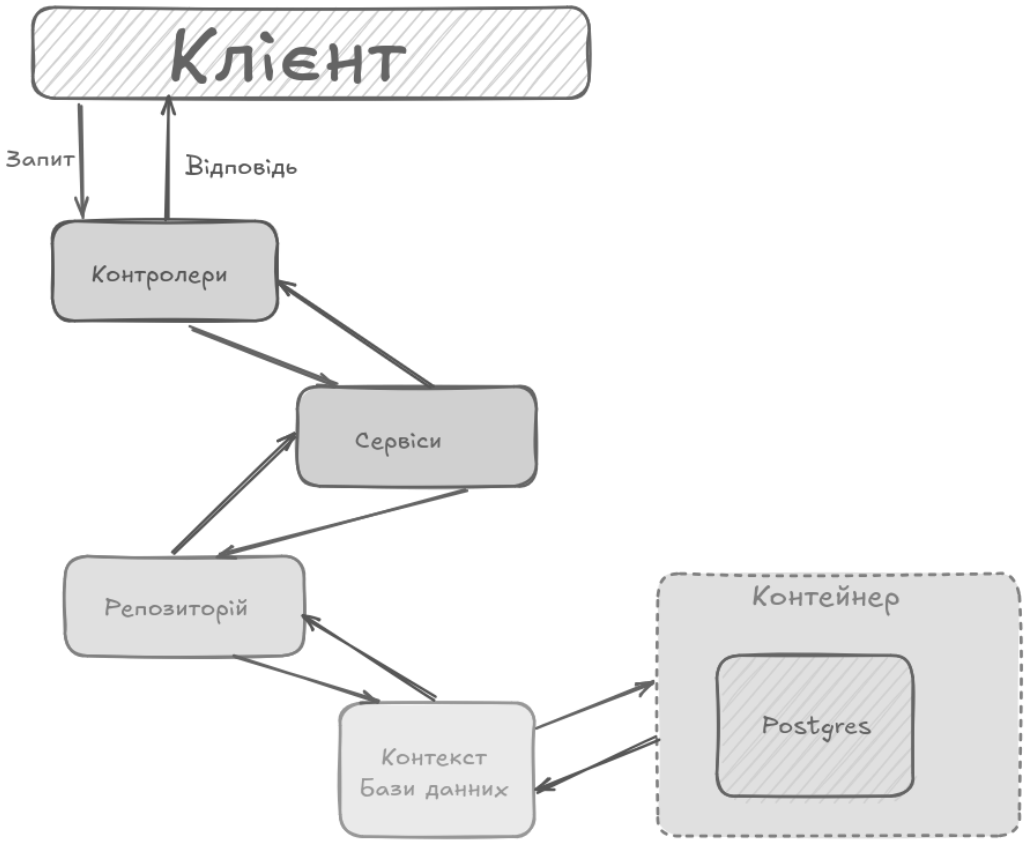

Рисунок 3.2 — Схема роботи веб сервера

3.3 Створення та конфігурування клієнтської частини

Клієнтська частина реалізована за допомогою Nuxt 3 — прогресивного JavaScript-фреймворку, який поєднує в собі можливості Vue 3 та Vite. Для створення проєкту було використано команду npx nuxi init kvFileY, яка згенерувала базову структуру Nuxt-застосунку з усіма необхідними файлами конфігурації. Після цього було виконано встановлення залежностей через npm install, що забезпечило наявність усіх бібліотек, необхідних для подальшої роботи.

Зовнішній вигляд інтерфейсу реалізовано за допомогою Tailwind CSS — утилітарного CSS-фреймворку, який дозволяє будувати адаптивний інтерфейс без написання кастомних класів стилів. Його було підключено до Nuxt-проєкту вручну. Створено файл assets/css/tailwind.css, у якому були оголошені базові директиви

Tailwind (@tailwind base;, @tailwind components;, @tailwind utilities;). Для активації Tailwind CSS у конфігурації Nuxt до nuxt.config.ts було додано відповідний запис у полі css, що дозволяє зробити стилі глобально доступними для всіх компонентів.

Також було налаштовано tailwind.config.js із зазначенням шляхів до всіх шаблонів і сторінок, щоб Tailwind міг "очистити" непотрібні класи й оптимізувати фінальний CSS. Це важливо для зменшення обсягу файлу стилів і підвищення швидкості завантаження.

Після ініціалізації структура застосунку була розширена відповідно до функціональних потреб. Застосунок поділено на логічні модулі: компоненти, сторінки, хуки (composables), сервіси (для взаємодії з API), типи TypeScript, конфігураційні файли та middleware. Це дозволяє легко масштабувати проєкт, додавати нові функції без дублювання коду та дотримуватися принципів розділення відповідальностей.

У каталозі components знаходяться повторно використовувані компоненти, які відповідають за візуальні блоки, зокрема fileComponent.vue, uploadComponent.vue, filterComponent.vue. Кожен із них має власний шаблон, логіку та стилі.

Каталог pages реалізує маршрутизацію. Кожен .vue-файл автоматично створює відповідний маршрут: наприклад, файл login.vue генерує маршрут /login, що значно спрощує навігацію.

У layouts зберігаються шаблони, які обгортають усі сторінки. Це забезпечує єдину структуру сторінок, наприклад, з однаковим заголовком або боковою панеллю.

У middleware реалізовано логіку перевірки доступу. Наприклад, auth.ts гарантує, що лише авторизовані користувачі можуть переглядати певні сторінки.

У папці public зберігаються статичні ресурси, які не змінюються під час роботи програми: іконки, favicon, SVG-графіка тощо. Вони доступні безпосередньо за URL.

Також потрібно налаштувати каталог composables, який буде містити як загальні хуки, так і підкаталог service, у якому реалізується логіка для роботи з

API. Це дозволяє відокремити запити до серверної частини від логіки інтерфейсу. Як приклад можна навести файл fileService.ts, він містить функції для завантаження та видалення файлів, тоді як authService.ts відповідає за авторизацію та реєстрацію. Кожне звернення за допомогою функцій, базується на fetch або useFetch, яке використовується в Nuxt API, і використовує глобально визначену змінну API-адреси з файлу config/api.ts. Це дозволяє зручно змінювати базову адресу бекенду без потреби редагувати кожен запит.

Для підвищення стабільності застосунку всі основні об'єкти типізовано у файлах каталогу types. Наприклад, user.ts, fileYDto.ts, logindto.ts описують структуру відповідних моделей, що використовуються як на рівні frontend, так і для коректної перевірки даних, які приходять із сервера.

3.4 Розробка основної логіки клієнтської частини

Основна логіка клієнтської частини базується на організації роботи з користувачем, управлінні його станом, а також на обробці файлів і захисті маршрутизації. Центральним елементом є кастомний Composition API хук auth.ts, який відповідає за зберігання стану користувача і забезпечує функції для входу, виходу та оновлення інформації. Для прикладу, у цьому хуку реактивна змінна user зберігає дані про поточного користувача, а обчислюване значення isAuthenticated визначає, чи авторизований користувач.

```
import { checkAuth } from "~/composables/service/authService";
export default defineNuxtRouteMiddleware(async (to, from) => {
  let isAuthenticated = await checkAuth();
  if (!isAuthenticated.data || isAuthenticated.error) {
    console.log('isAuthenticated', isAuthenticated);
    return navigateTo('/login');   } })
```

Сторінка login.vue призначена для автентифікації користувачів (рис. 3.3). Вона містить форму з полями для введення email і пароля. Користувач вводить дані, які валідуються. Після відправлення форми додаток надсилає запит до сервера через сервіс loginFetch. У разі успіху користувач перенаправляється на сторінку /files. Якщо виникають помилки, наприклад, невірні дані або проблеми з сервером, відображається повідомлення про помилку. Під час запиту показується

індикатор завантаження.

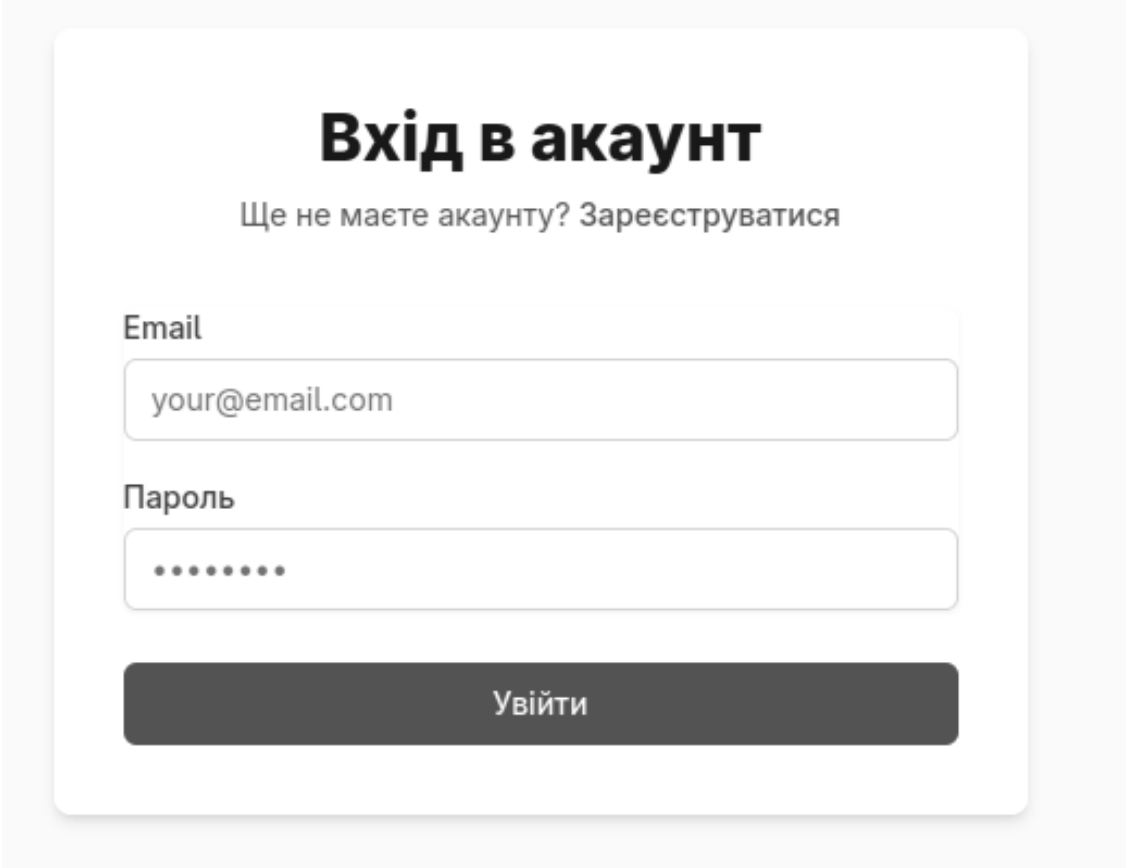

Рисунок 3.3 — Сторінка входу користувача

Сторінка містить форму для входу з двома полями: email і пароль, які зберігаються в реактивному об’єкті data типу LoginDto. Форма валідує введені дані: перевіряє, чи заповнені поля, і чи відповідає email формату через регулярний вираз (/^\S+@\S+\.\S+$/). Функція handleSubmit обробляє відправлення форми, викликаючи loginFetch для автентифікації. У разі успіху користувача перенаправляють на /files через router.push. Помилки (наприклад, невірний пароль) відображаються в errorMessage. Стан isLoading керує індикатором завантаження, який показує анімацію під час запиту.

```
<script setup lang="ts">
import type { LoginDto } from "~/types/logindto";
import {loginFetch } from "~/composables/service/authService";
const router = useRouter();
const data = ref<LoginDto>({ email: "", password: "" });
const errorMessage = ref<string>("");
const isLoading = ref<boolean>(false);
```

```
const handleSubmit = async (e: Event) => {
  e.preventDefault();
  errorMessage.value = "";
  if (!data.value.email || !data.value.password) {
     errorMessage.value = "Будь ласка, заповніть всі поля";
     return;  }
  if (!/^\S+@\S+\.\S+$/.test(data.value.email)) {
     errorMessage.value = "Будь ласка, введіть коректний email";
     return;
  }
  try {
     isLoading.value = true;
     const { error } = await loginFetch(data.value);
     if (error) {
        errorMessage.value = error;
        return;   }
     await router.push("/files");
  } catch (err) {
     errorMessage.value = "Сталася помилка при вході";
     console.error("Login error:", err);
  } finally {
     isLoading.value = false;  }};
</script>
```

Сторінка register відповідає за реєстрацію нових користувачів. Вона містить форму з полями для імені, email, пароля та підтвердження пароля. Форма валідує дані: усі поля мають бути заповнені, email — коректним, паролі — співпадати, а пароль — містити щонайменше 6 символів. Після успішної відправки даних через registerFetch користувача перенаправляють на сторінку входу. У разі помилок, наприклад якщо email уже зайнятий, то відображається повідомлення.

```
<script setup lang="ts">
interface RegisterForm {
 userName: string;
 email: string;
 password: string;
 confirmPassword: string;}
const router = useRouter();
const data = ref<RegisterForm>({
 userName: "",
 email: "",
 password: "",
 confirmPassword: ""
});
const errorMessage = ref<string>("");
const isLoading = ref<boolean>(false);
const handleSubmit = async (e: Event) => {
 e.preventDefault();
```

```
 errorMessage.value = "";
 if (!data.value.email || !data.value.password || !data.value.userName) {
  errorMessage.value = "Будь ласка, заповніть всі обов'язкові поля";
  return; }
 if (!/^\S+@\S+\.\S+$/.test(data.value.email)) {
  errorMessage.value = "Будь ласка, введіть коректний email";
  return; }
 if (data.value.password !== data.value.confirmPassword) {
  errorMessage.value = "Паролі не співпадають";
  return; }
 if (data.value.password.length < 6) {
  errorMessage.value = "Пароль повинен містити щонайменше 6 символів";
  return; }
 try {
  isLoading.value = true;
  const { error } = await registerFetch({
   email: data.value.email,
   password: data.value.password,
   userName: data.value.userName });
  if (error) {
   errorMessage.value = error;
   return;  }
  await router.push("/login");
 } catch (err) {
  errorMessage.value = "Сталася помилка при реєстрації";
  console.error("Registration error:", err);
 } finally {
  isLoading.value = false; }};
</script>
```

Сторінка реалізує форму реєстрації з полями для імені користувача, email, пароля та підтвердження пароля, що зберігаються в data типу RegisterForm (рис. 3.4). Валідатор перевіряє заповненість полів, коректність email, збіг паролів і мінімальну довжину пароля (6 символів). Після відправлення форми викликається registerFetch, і в разі успіху користувача перенаправляють на /login. Помилки відображаються в errorMessage, а isLoading керує станом завантаження. Інтерфейс включає посилання на сторінку входу.

Сторінка files.vue є центральним елементом додатка для керування файлами, призначена для автентифікованих користувачів, що забезпечується middleware auth. Її основна мета — надати зручний інтерфейс для перегляду, пошуку, сортування, фільтрації та завантаження файлів.

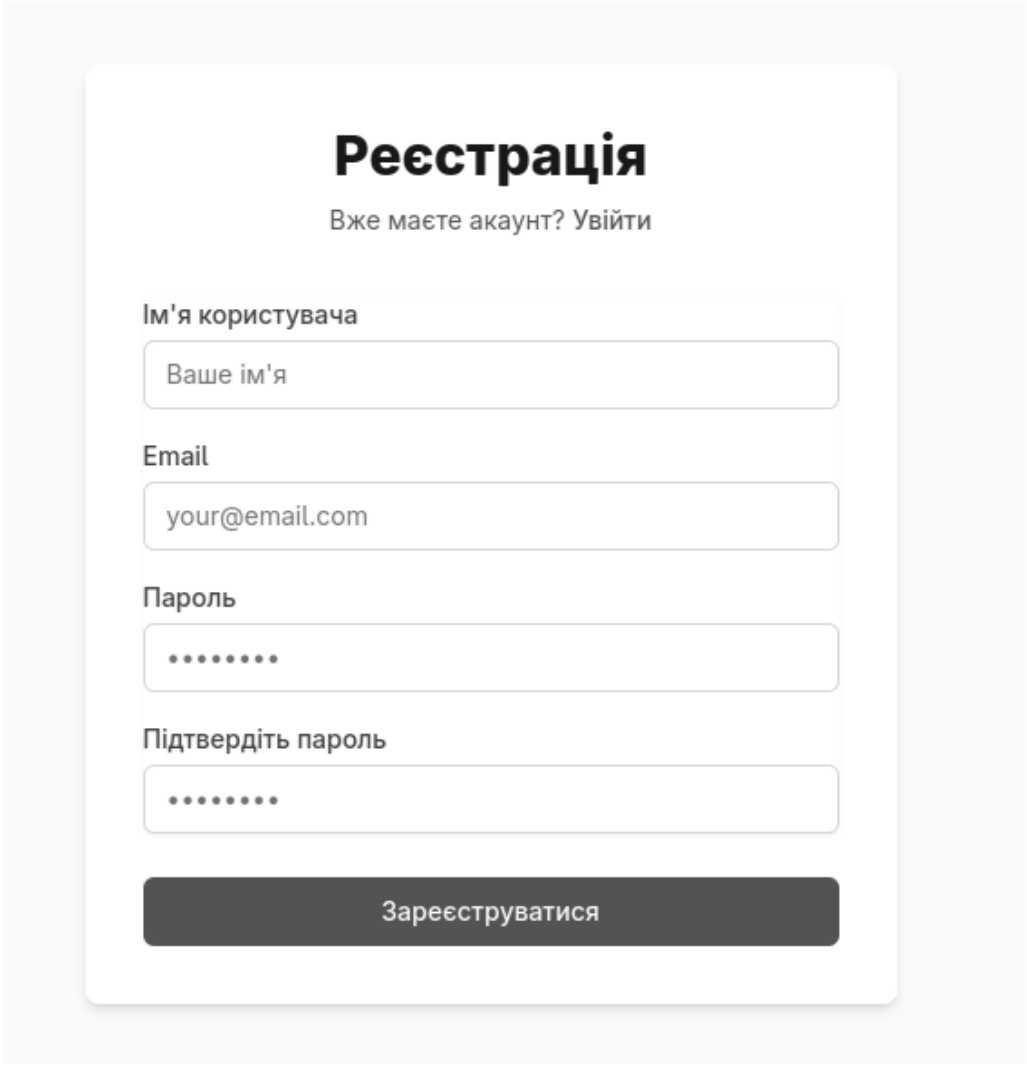

Рисунок 3.4 — Сторінка реєстрації користувача

Користувач бачить адаптивну сітку файлів, яка змінює кількість стовпців залежно від розміру екрана, панель із полем пошуку, кнопкою для додавання файлів і фільтрами. Пошук дозволяє знаходити файли за ключовими словами, а фільтри дають змогу налаштувати сортування та кількість файлів на сторінці. Якщо файлів більше, ніж відображено, з’являється кнопка "Load More" для пагінації. При натисканні на кнопку "Add" відкривається форма для завантаження файлів, після чого список автоматично оновлюється. Сторінка має бути інтуїтивною, швидко реагувати на дії користува.

```
<script setup lang="ts">
const searchTerm = ref<string>('');
const files = ref<FileYDto[]>([]);
const isFileUpload = ref<boolean>(false);
```

```
const pageSize = ref<number>(10);
const sortBy = ref<string>('');
const sortOrder = ref<string>('');
const fileCount = ref<number>(0);
const currentPage = ref<number>(1);
const search = async () => {
   currentPage.value = 1;
   files.value = await getFiles(currentPage.value) || [];
   fileCount.value = await getFileCount() || 0;}
const getFiles = async (page: number)=>{
    const response = await fetchFiles(page, pageSize.value, searchTerm.value,sortBy.value, sortOrder.value);
 if(response.status === 200) {
     return response.data || [];
  } else {
     console.error(response.error);}}
const getFileCount = async ()=> {
 const countRes = await fetchFileCount();
 if(countRes.status === 200) {
  return countRes.data || 0;
  console.log(JSON.stringify(fileCount.value));
 } else {
  console.error(countRes.error);}}
const addNewFile = () => {
   isFileUpload.value = true;}
const cancelUpload = () => {
   isFileUpload.value = false;}
const updateFiles = async (pS:number = 10, sB:string = 'asc', sO:string='name') =>{
  pageSize.value = pS;
  sortBy.value = sB;
  sortOrder.value = sO;
  await search();}
const loadMore = async () => {
 currentPage.value++;
 const newFiles = await getFiles(currentPage.value) || [];
 files.value.push(...newFiles); // додає елементи з newFiles у кінець files};
onMounted(async () => {
   await search();
   JSON.stringify(files?.value);})
const fileCountExceeded = computed(() => {
 return (files.value?.length ?? 0) < fileCount.value;});
</script>
```

У секції <script setup> із TypeScript імпортуються основні залежності з бібліотеки Vue, зокрема ref — для створення реактивних змінних, computed — для визначення обчислювальних властивостей на основі інших значень, і onMounted — для виконання коду після монтування компонента. Крім цього, імпортуються зовнішні сервіси fetchFiles та fetchFileCount, які відповідають за отримання списку

файлів і загальної їх кількості із сервера. Для типізації використовується тип FileYDto, який описує структуру одного файлу. Також підключаються три внутрішні компоненти: FileComponent, FilterComponent та UploadComponent, що відповідають відповідно за візуалізацію окремих файлів, застосування фільтрів і завантаження нових файлів. Для конфігурації сторінки використовується definePageMeta, що встановлює middleware auth для перевірки автентичності користувача та макет active, який забезпечує вбудовування сторінки у загальний інтерфейс додатка.

Усі ключові параметри, що відповідають за стан сторінки, оголошено як реактивні змінні. Змінна searchTerm зберігає текстовий рядок пошукового запиту, files — масив об’єктів-файлів, isFileUpload — логічне значення, яке визначає, чи активовано режим завантаження нового файлу. Змінні pageSize, sortBy і sortOrder дозволяють налаштувати розмір сторінки та параметри сортування. Змінна fileCount відображає загальну кількість файлів, доступних на сервері, тоді як currentPage фіксує номер поточної сторінки. Для оновлення даних використовується функція search, яка скидає currentPage до 1 і запускає функції getFiles і getFileCount, що оновлюють список файлів та їх кількість відповідно.

Функція getFiles виконує асинхронний запит до сервера, передаючи параметри пагінації, пошукового запиту та сортування. У разі успішного запиту повертається масив файлів; у випадку помилки — порожній масив. Функція getFileCount аналогічно запитує загальну кількість доступних файлів, але містить помилку: після return присутній виклик console.log, який не буде виконаний через недосяжність цього коду. Кнопка "Add" активує функцію addNewFile, яка вмикає режим завантаження, тоді як cancelUpload його скасовує. Зміна параметрів сортування або кількості елементів на сторінці викликає функцію updateFiles, яка у свою чергу запускає search для оновлення відображеного списку файлів.

Механізм пагінації реалізовано через функцію loadMore, яка збільшує номер поточної сторінки та додає нові файли до наявного масиву. У хуці onMounted одразу після завантаження сторінки викликається search, що ініціює початкову вибірку даних. Проте, рядок JSON.stringify(files?.value) є надлишковим, оскільки

результат цього виклику не використовується. Додатково визначено обчислювальну властивість fileCountExceeded, яка порівнює кількість завантажених файлів із загальним числом і повертає логічне значення, що визначає, чи потрібно показувати кнопку "Load More".

У секції <template> реалізовано структуру сторінки. Верхня панель інтерфейсу містить інформаційний блок із кількістю файлів, поле введення пошуку, кнопку для запуску пошуку та кнопку "Add" для додавання нового файлу (рис. 3.5). Всі ці елементи мають стилі, визначені з використанням Tailwind CSS, що забезпечує адаптивну верстку. Якщо isFileUpload є false, то основним контентом стають компонент FilterComponent, який дозволяє користувачу обирати фільтри, і сітка з файлами, яка будується через v-for та компонент FileComponent. Залежно від розміру екрана сітка має від одного до чотирьох стовпців, і кожен файл має візуальний ефект масштабування при наведенні курсора. Якщо ж isFileUpload дорівнює true, то замість основного контенту виводиться компонент UploadComponent, який підтримує обробку подій скасування або завершення завантаження. Кнопка "Load More" з’являється тільки тоді, коли є ще невідображені файли, і при її натисканні викликається функція loadMore.

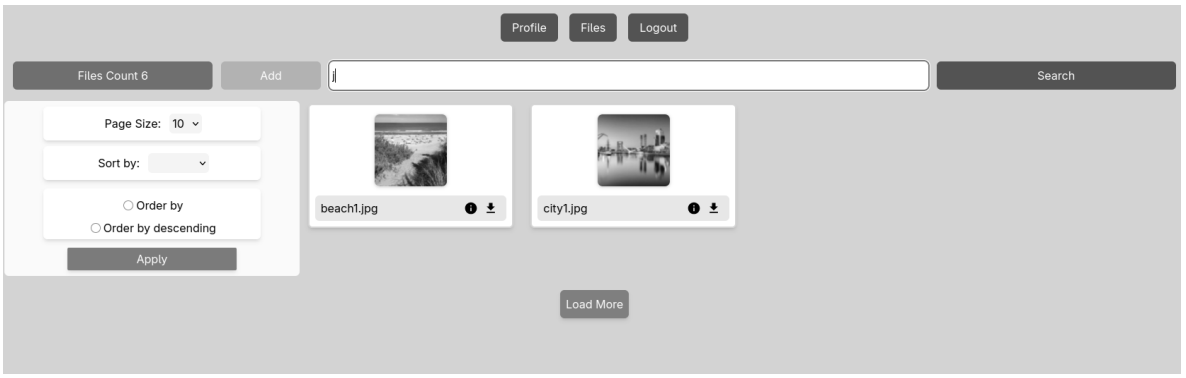

Рисунок 3.5 — Сторінка з файлами користувача

3.5 Тестування та перевірка коректності роботи

Тестування базової функціональності вебсайту передбачає перевірку ключових можливостей без зміни програмного коду. Основними етапами такого тестування є перевірка процесу реєстрації нового користувача, входу до системи,

завантаження файлів, їх коректного відображення у списку та застосування фільтрації. Мета тесту — переконатися, що користувач може взаємодіяти з основними компонентами сайту відповідно до задуманої логіки.

Першим кроком є відвідування сторінки реєстрації, яка завантажується автоматично, якщо користувач ще не автентифікований. На цій сторінці відображається форма для заповнення, яка включає кілька обов’язкових полів: ім’я користувача, адреса електронної пошти, пароль і підтвердження пароля. Користувач вводить тестові дані: у поле "Ім’я" — anton, у поле "Email" — anton@mail.com, у поле "Пароль" — 123456, і той самий пароль — у поле підтвердження. Після натискання кнопки "Зареєструватися", система виконує низку перевірок: чи заповнені всі обов’язкові поля, чи введено email у правильному форматі, чи збігаються паролі між собою та чи дотримано мінімальної довжини пароля — щонайменше 6 символів. Якщо всі умови виконано, запит на реєстрацію відправляється на сервер.

Після успішної реєстрації або при повторному відвідуванні сайту користувач може перейти на сторінку входу в систему. Там необхідно ввести адресу електронної пошти та пароль. У цьому тесті вводиться email anton@ton.com (зміна у домені — можливий тест на валідацію користувача) і пароль 123456. Після натискання кнопки "Увійти" система знову проводить валідацію даних — перевіряє, чи заповнені поля, і формує запит до серверної частини для перевірки облікових даних. У разі успішної автентифікації користувач перенаправляється на головну сторінку з файлами. Якщо ж пароль неправильний або користувача не існує, система відображає повідомлення про помилку автентифікації з відповідним текстом.

Після входу користувач потрапляє на сторінку керування файлами (рис. 3.6). У верхній частині інтерфейсу розташована панель з елементами управління: текстовий блок "Files Count 0", що вказує на поточну кількість файлів у системі, кнопка "Add" для додавання нових файлів, поле пошуку та кнопка "Search" для виконання пошукового запиту. Основна частина сторінки — це сітка файлів, яка наразі порожня, оскільки жодного файлу ще не було додано.

Натиснувши кнопку "Add", користувач відкриває інтерфейс завантаження файлів (рис. 3.7). З'являється форма, яка дозволяє обрати файл із пристрою користувача. Також може бути вказано перелік підтримуваних форматів файлів, що підвантажуються (наприклад, PDF, DOCX, PNG, JPG тощо). Користувач може перевірити, чи коректно працює перевірка формату файлу, натиснувши кнопку завантаження з різними типами файлів (рис. 3.8).

Після вибору одного або кількох файлів на локальному пристрої користувача на екрані з’являється список цих файлів, який дозволяє швидко переглянути їхні назви, типи, а також перевірити, чи всі потрібні файли були обрані.

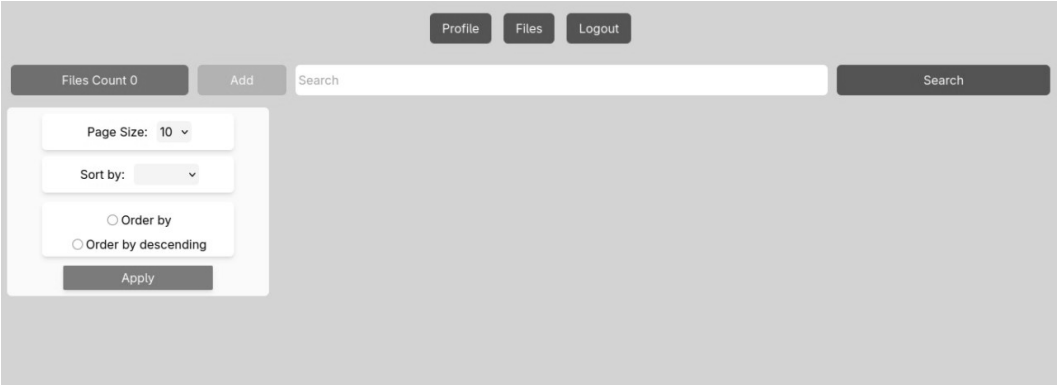

Рисунок 3.6 — Сторінка для відображення файлів користувача

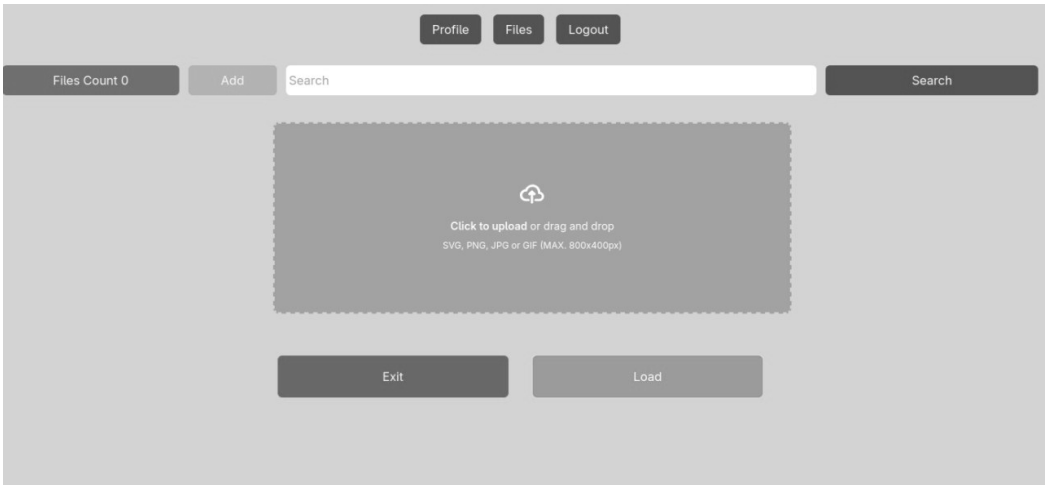

Рисунок 3.7 — Форма для вибору файлів

Рисунок 3.8 — Список вибраних файлів

Під списком файлів доступні дві кнопки: "Load" — для підтвердження та запуску процесу завантаження, і "Exit" — для скасування дії та повернення на основну сторінку без збереження змін. Після натискання кнопки "Load" і завершення завантаження відбувається автоматичне оновлення даних: блок "Files Count" змінює значення на 13, що означає успішне додавання 13 нових файлів до системи. Одночасно основна сітка контенту оновлюється й показує всі файли у вигляді карток або рядків списку, де вказано ім’я файлу, його розмір, а також доступна кнопка для окремого завантаження кожного файлу.

Крім перегляду повного списку файлів, користувач має можливість скористатися панеллю фільтрів, яка дозволяє впорядковувати та обмежувати список відповідно до заданих критеріїв. Наприклад, користувач у блоці фільтрації вибирає значення "Page size" — 15, а також сортування "за розміром" у порядку спадання. Після натискання кнопки "Apply" система виконує запит до сервера з оновленими параметрами та перебудовує відображення файлів: тепер першими

відображаються найбільші за розміром файли. Зокрема, такими можуть бути файли типу "akismc.doc", за яким іде "akismc.jpg" та інші. Оскільки вибране значення розміру сторінки — 15, а загальна кількість файлів становить лише 13, то всі файли відображаються одночасно, і кнопка "Load More" не з’являється, оскільки додаткових сторінок не існує.

Функція пошуку забезпечує можливість швидко знаходити потрібні файли за частиною їхньої назви. У полі пошуку вводиться символ або рядок, наприклад, “a”, після чого натискається кнопка "Search". Система обробляє запит і оновлює список так, щоб відобразити лише ті файли, у назві яких міститься задана послідовність символів. У результаті, користувач бачить скорочений перелік, до якого входять файли на зразок "akismc.doc", "akismc.jpg", "akismc(1).jpg", а також інші файли, що відповідають критерію пошуку. Така функція є особливо корисною, коли в системі зберігається велика кількість файлів, і потрібно швидко знайти потрібний.

Після завершення роботи користувач має змогу завершити сесію, натиснувши кнопку "Logout" у навігаційній панелі або в іншому зручному місці інтерфейсу. Після натискання цієї кнопки система очищає дані автентифікації, завершує сесію й автоматично перенаправляє користувача на сторінку входу. Це підтверджує правильну роботу механізмів авторизації та безпеки.

ВИСНОВКИ

Цю бакалаврську дипломну роботу було присвячено розробці та впровадженню комп’ютерної системи хмарного сховища файлів, яка відповідає вимогам зручності, безпеки та адаптивності. Головною метою дослідження було створення інтерактивного веб застосунку, що дозволяє користувачам ефективно зберігати, переглядати, шукати, сортувати та завантажувати файли різних форматів, забезпечуючи при цьому надійну аутентифікацію та зрозумілий інтерфейс. Система орієнтована на користувачів, яким потрібне централізоване, доступне в будь-який час та з будь-якого місця.

Під час виконання роботи було проведено аналіз сучасних веб технологій, що дозволило обрати оптимальний стек інструментів для реалізації як серверної, так і клієнтської частини. Серверна частина була побудована на базі ASP.NET Core, що забезпечує надійність, масштабованість і високу продуктивність API, а клієнтська — на Nuxt.js, що гарантує швидке завантаження, адаптивність та зручність використання на різних пристроях — від десктопів до мобільних телефонів і планшетів. Для підвищення безпеки користувачів було інтегровано механізми аутентифікації на основі cookie, що дозволяє ідентифікувати користувача та контролювати рівень доступу до ресурсів.

Архітектура системи передбачає чітке розділення на рівні контролерів, сервісів і репозиторіїв, що сприяє підвищенню якості коду, його зручності для підтримки та подальшого розвитку. Така модульність забезпечує простоту тестування окремих компонентів і гнучкість при інтеграції нових функцій. Особлива увага приділялась обробці великих обсягів файлів: реалізовано збереження файлів у структурованих каталогах із урахуванням хешування для уникнення конфліктів і дублювань, а також функціонал пагінації, пошуку та сортування, що значно підвищує зручність і ефективність роботи користувача із даними.

Клієнтська частина на Nuxt.js створює приємний користувацький досвід завдяки інтуїтивному інтерфейсу, який однаково працює на різних платформах і пристроях. Це дозволить користувачам мати повний контроль над файлами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Переваги та недоліки хмарних технологій // ICTinUA. – URL: http://www.ict.in.ua/perevagi-ta-nedoliki-hmarnih-tehnologij

2. Квятковський А. А. Комп'ютерна система хмарного сховища файлів соціалізації // Матеріали всеукраїнської науково-практичної Інтернет-конференції студентів, аспірантів та молодих науковців — Режим доступу : https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/view/25460

3. Що таке хмарне сховище і чому воно стало необхідним? // Business Broker. – URL: https://business-broker.com.ua/blog/iak-vybraty-khmarne-skhovyshche-dlia-biznesu-ta-osobystykh-potreb/

4. MEGA. Офіційний сайт. – URL: https://mega.io/

5. pCloud. Офіційний сайт. – URL: https://www.pcloud.com/

6. Sync.com. Офіційний сайт. – URL: https://www.sync.com/

7. Icedrive. Офіційний сайт. – URL: https://icedrive.net/

8. Proton Drive. Офіційний сайт. – URL: https://proton.me/drive

9. NoSQL Databases Explained. MongoDB Documentation. – URL: https://www.mongodb.com/nosql-explained

10. PostgreSQL. Офіційна документація. – URL: https://www.postgresql.org/docs/

11. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. – University of California, Irvine, 2000. – 162 с.

12. gRPC. Офіційна документація. – URL: https://grpc.io/

13. FastAPI. Офіційна документація. – URL: https://fastapi.tiangolo.com/

14. Gin Web Framework. – URL: https://gin-gonic.com/

15. Axum. Tokio Rust framework. – URL: https://docs.rs/axum/latest/axum/

16. ASP.NET. Microsoft Documentation. – URL: https://learn.microsoft.com/en-us /aspnet/core/

17. React. Офіційна документація. – URL: https://react.dev/

18. Vue.js. Офіційна документація. – URL: https://vuejs.org/

19. Angular. Офіційна документація. – URL: https://angular.io/

20. Svelte. Офіційна документація. – URL: https://svelte.dev/

21. Nuxt. Офіційна документація. – URL: https://nuxt.com/

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. Азаров О.Д..
10.03.2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання бакалаврської кваліфікаційної роботи
«Комп’ютерна система хмарного сховища файлів»
08-54.БКР.012.00.000.ТЗ

Науковий керівник: доцент к.т.н.
_____ Муращенко О. Г.
Студент групи 1КІ-21б
_____ Квятковський А. А.

ВНТУ 2025

1 Підстави для виконання бакалаврської кваліфікаційної роботи (БКР)

1.1 Актуальність дослідження зумовлена зростаючою потребою в надійних та зручних цифрових рішеннях для зберігання, керування та обміну файлами в онлайн-середовищі. У сучасних умовах цифровізації суспільства користувачі очікують швидкого доступу до персональних даних, безпечного обміну файлами. Це дослідження спрямоване на вирішення вказаних потреб шляхом розробки файлового веб-сховища з сучасними підходами до взаємодії користувача з даними.

1.2 Тема бакалаврської кваліфікаційної роботи затверджена наказом ректора університету № 97 від 21.03.2025 року.

2 Мета БКР і призначення розробки

2.1 Метою роботи є створення сучасної інформаційної системи веб-сховища для зберігання та керування файлами з використанням Nuxt.js для клієнтської частини та ASP.NET для реалізації серверної логіки, включно з автентифікацією, обробкою запитів та роботою з базою даних.

2.2 Призначення розробки полягає у забезпеченні користувачів зручним веб-інструментом для завантаження, перегляду, фільтрації та пошуку файлів, з акцентом на безпечну обробку даних, масштабованість та простоту взаємодії.

3 Вихідні дані для виконання БКР

3.1. Базові знання з розробки веб-застосунків.
3.2. Розуміння принципів взаємодії frontend та backend частин.
3.3. Ознайомлення з можливостями фреймворків ASP.NET і Nuxt.
3.4. Навички проєктування баз даних і створення REST API.
3.5. Знання сучасних підходів до розробки дизайну інтерфейсів.
3.6. Основи захисту даних, аутентифікації та авторизації користувачів.
3.7. Досвід використання інструментів для розробки й тестування.

4 Вимоги до виконання БКР

4.1 Провести аналіз потреб користувачів у зберіганні та керуванні особистими файлами в онлайн-середовищі.

4.2 Розробити архітектуру інформаційної системи з урахуванням вимог зручності, безпеки та ефективності.

4.3. Створити серверну та клієнтську частини системи.

4.4 Реалізувати функціонал реєстрації, авторизації, перегляду профілів, пошуку за фільтрами та обміну повідомленнями.

4.5 Забезпечити збереження та обробку даних за допомогою бази даних.

4.6. Протестувати працездатність системи та підготувати пояснювальну записку.

5 Етапи БКР та очікувані результати

А.1. Таблиця А.1 — Етапи БКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз задачі	21.03.25	23.03.25	Аналітичний огляд джерел, задачі досліджень,
2	Огляд аналогічних сервісів, аналіз функціональних вимог	23.03.25	27.03.25	Розділ 1
3	Вивчення предметної області та наповнення веб застосунку	28.03.25	04.04.25	Розділ 2
4	Розробка та програмна реалізація для системи	05.04.25	12.04.25	Розділ 3
5	Оформлення пояснювальної записки, графічного матеріалу і презентації	12.04.25	10.05.25	ПЗ, графіч. матеріал і презентація
6	Підготовка супроводжуючих документів, проходження нормоконтролю та тесту на плагіат	10.05.25	13.05.25	Оформлені документи

6 Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, графічні та ілюстративні

матеріали, протокол попереднього захисту на кафедрі, відгуки наукового керівника та опонента, протоколи складання державних екзаменів, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР діючим вимогам.

7 Порядок контролю виконання та захисту БКР

Виконання етапів документації БКР контролюється науковим керівником згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання БКР

8.1 При оформлюванні БКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 — «Комп’ютерна інженерія»;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21.

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: _Комп'ютерна система хмарного сховища файлів

Тип роботи: _____ бакалаврська кваліфікаційна робота _____
(бакалаврська кваліфікаційна робота)

Підрозділ _____кафедра обчислювальної техніки_____
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism ___1____%

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____Азаров О.Д., зав. каф. ОТ _____ _____
(прізвище, ініціали, посада) (підпис)

____Крупельницький Л.В., доц. каф ОТ_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ ___Захарченко С.М.____
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ _____Муращенко О. Г_____
(підпис) (прізвище, ініціали, посада)

Здобувач _____ _____Квятковський А. А._____
(підпис) (прізвище, ініціали)

ДОДАТОК В

Ілюстративна частина

КОМП’ЮТЕРНА СИСТЕМА ХМАРНОГО СХОВИЩА ФАЙЛІВ

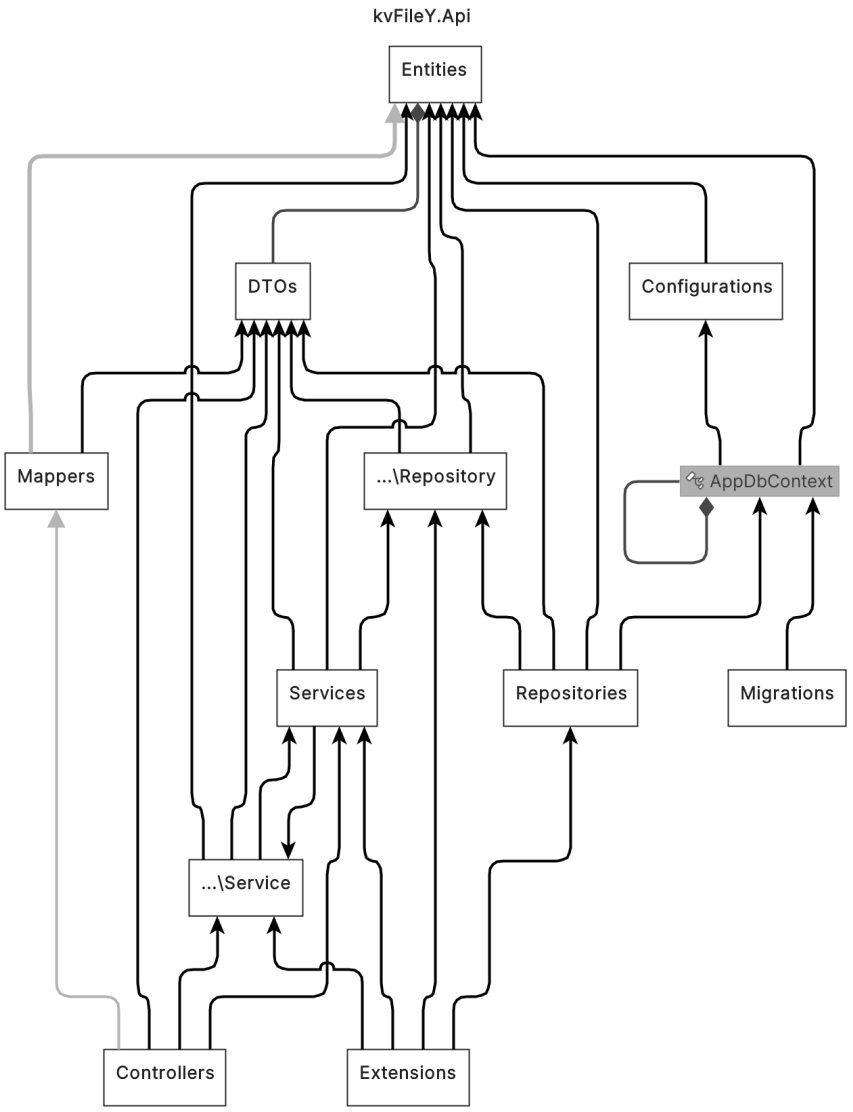

Рисунок В.1 — Схема залежності класів серверної частини

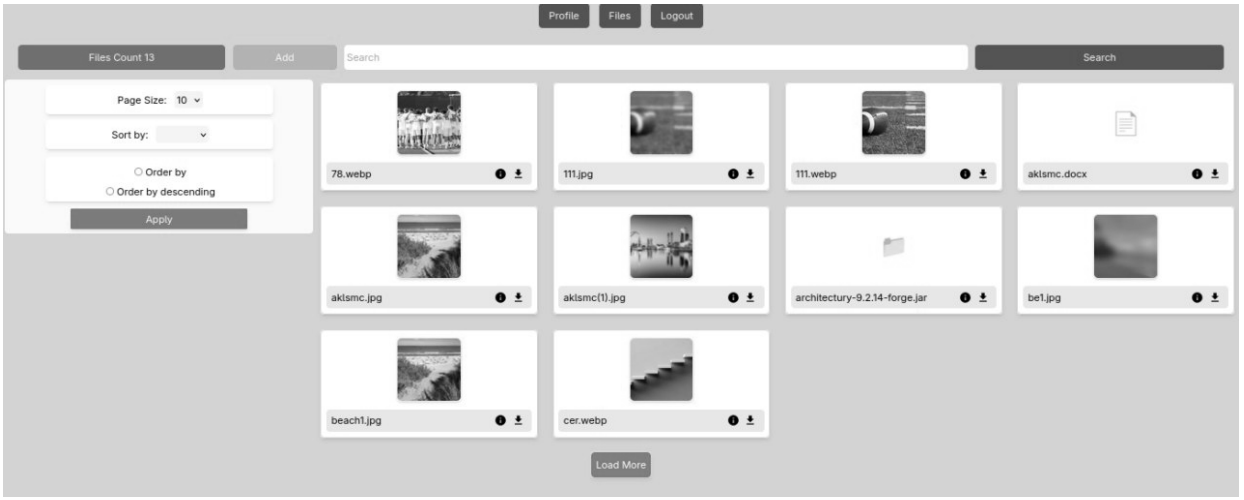

Рисунок В.2 — Сторінка з файлами користувача

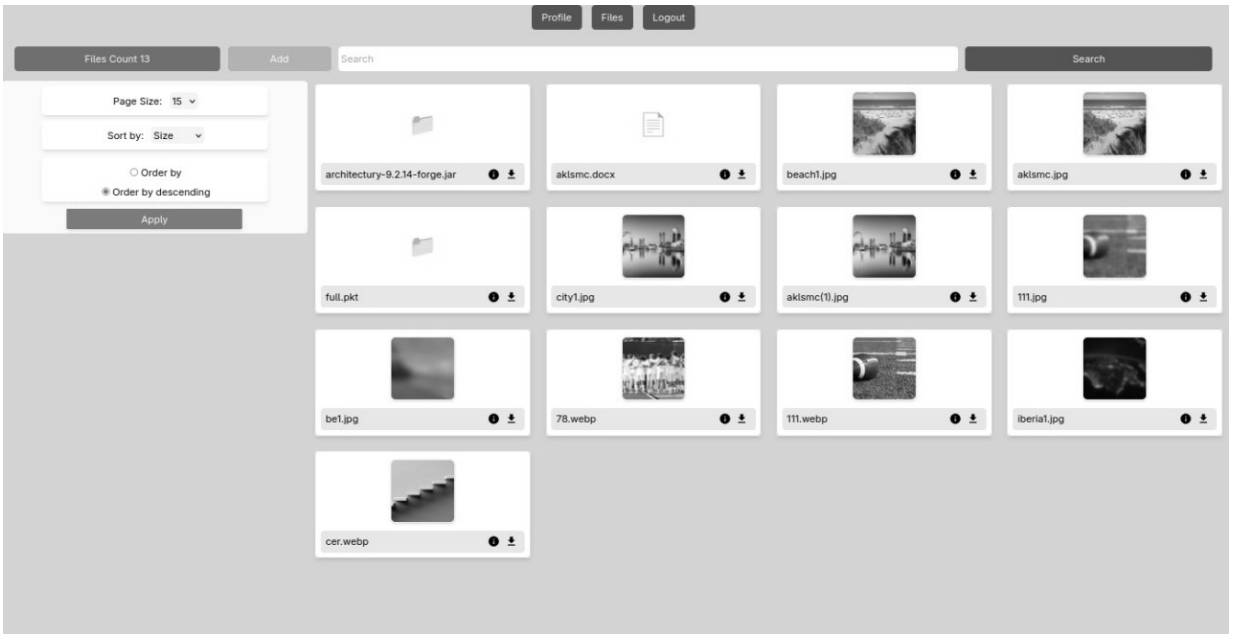

Рисунок В.3 — Файли з приміненням сортування

Рисунок В.4 — Пошук за іменем файлу

ДОДАТОК Г

Лістинг програмного забезпечення

```
public class FileYRepository : IFileYRepository{
    private readonly AppDbContext _context;
    private readonly ILogger<FileYRepository> _logger;
    public FileYRepository(AppDbContext context, ILogger<FileYRepository> logger){
        _context = context;
        _logger = logger;    }
    public async Task<List<FileY>> GetPageAsync(int userId,
        int page = 1,
        int pageSize = 15,
        string? searchTerm = null,
        string? sortBy = null,
        string? sortOrder = "asc",
        CancellationToken ct = default)    {
        var query = _context.Files
            .AsNoTracking()
            .Where(f => f.UserId == userId);
        // Додатковий пошук (якщо вказано)
        if (!string.IsNullOrWhiteSpace(searchTerm))        {
            var search = searchTerm.ToLower();
            query = query.Where(f =>
                f.FileName.ToLower().Contains(search));        }
        if (!string.IsNullOrWhiteSpace(sortBy))        {
            var selector = GetSortSelector(sortBy);
            query = sortOrder?.ToLower() == "desc"
                ? query.OrderByDescending(selector)
                : query.OrderBy(selector);        }
        else        {
            query = query.OrderBy(f => f.Id);        }
        // Пагінація та виконання запиту
        return await query
            .Skip((page - 1) * pageSize)
            .Take(pageSize)
            .AsNoTracking()
            .ToListAsync(cancellationToken: ct);    }
    public async Task<FileY?> GetByIdAsync(int id, CancellationToken ct = default)
    {       return await _context.Files
            .AsNoTracking()
            .FirstOrDefaultAsync(x => x.Id == id, cancellationToken: ct);    }
    public async Task<long> GetFilesCountAsync(int userId, CancellationToken ct = default)    {
        return await _context.Files
        .AsNoTracking()
        .Where(u => u.UserId == userId)
        .LongCountAsync(ct);    }
     public async Task<FileY?> GetByStoredNameAsync(string storedName, CancellationToken
ct = default)    {
        return await _context.Files
            .AsNoTracking()
            .FirstOrDefaultAsync(x => x
                .FileName.ToLower()
```

```
                .Contains(storedName.ToLower()), cancellationToken: ct);    }
            public async Task<bool> ExistsAsync(int id, CancellationToken ct = default)    {
                return await _context.Files.AnyAsync(x => x.Id == id, ct);    }
                public async Task<bool> IsOwnerAsync(int fileId, int userId, CancellationToken ct =
default)    {
                return await _context.Files
                    .AsNoTracking()
                    .Where(f => f.Id == fileId)
                    .Select(f => f.UserId)
                    .FirstOrDefaultAsync(ct) == userId;    }
            public async Task<bool> UpdateAsync(FileYUpdateDto file, CancellationToken ct = default)
{       ArgumentNullException.ThrowIfNull(file);
                var updatedRows = await _context.Files
                    .Where(f => f.Id == file.Id)
                    .ExecuteUpdateAsync(
                        setters => setters
                            .SetProperty(f => f.FileName, file.FileName)
                            .SetProperty(f => f.ContentType, file.ContentType)
                            .SetProperty(f => f.FilePath, file.FilePath)
                            .SetProperty(f => f.FileSize, file.FileSize),
                        ct);
                return updatedRows > 0;    }
            public async Task<FileY> CreateAsync(FileY file, CancellationToken ct = default)    {
                await _context.Files.AddAsync(file, ct);
                await _context.SaveChangesAsync(ct);
                return file;    }
            public async Task<List<FileY>> CreateRangeAsync(List<FileY> files, CancellationToken ct
= default)    {
                await _context.Files.AddRangeAsync(files, ct);
                await _context.SaveChangesAsync(ct);
                return files;    }
            public async Task<bool> DeleteAsync(int id, CancellationToken ct = default)    {
                try        {
                    var affectedRows = await _context.Files
                        .Where(u => u.Id == id)
                        .ExecuteDeleteAsync(ct);
                    return affectedRows > 0; // Повертає true, якщо щось було видалено        }
                catch (DbUpdateException ex)        {
                    _logger.LogError(ex, $"Помилка при видаленні файлу {id}");
                    return false;        }    }
            ///DELETE
            public async Task<bool> DeleteAllAsync(CancellationToken ct = default)
            {        try        {
                    var affectedRows = await _context.Files
                        .Where(u => u.Id > 0)
                        .ExecuteDeleteAsync();
                    return affectedRows > 0; // Повертає true, якщо щось було видалено        }
                catch (DbUpdateException ex)        {
                    return false;        }    }
            public async Task<int> GetFileCountAsync(int userId, CancellationToken ct = default)    {
                                return await _context.Files.AsNoTracking().Where(f => f.UserId ==
userId).CountAsync(ct);    }
```

```
        private static Expression<Func<FileY, object>> GetSortSelector(string sortBy)    {
            return sortBy.ToLower() switch        {
                "name" => f => f.FileName,
                "size" => f => f.FileSize,
                "date" => f => f.UploadedAt,
                _ => f => f.Id // Сортування за замовчуванням        };    }}
    public class UserRepository : IUserRepository{
        private readonly AppDbContext _context;
        private readonly ILogger<UserRepository> _logger;
        public UserRepository(AppDbContext context, ILogger<UserRepository> logger)    {
            _context = context;
            _logger = logger;    }
        public async Task<User?> GetByIdAsync(int id, CancellationToken ct = default)    {
            return await _context.Users
                .AsNoTracking()
                .FirstOrDefaultAsync(u => u.Id == id, ct);    }
        public async Task<User?> GetByEmailAsync(string email, CancellationToken ct = default)
        {       return await _context.Users
                .AsNoTracking()
                .FirstOrDefaultAsync(u => u.Email == email, ct);    }
          public async Task<IReadOnlyList<User>> GetUserPageAsync(int page, int pageSize = 15,
CancellationToken ct = default)    {
            return await _context.Users
                .AsNoTracking()
                .OrderBy(u => u.Id)
                .Skip((page-1) * pageSize)
                .Take(pageSize)
                .ToListAsync(ct);    }
        public async Task<bool> ExistsAsync(int id, CancellationToken ct = default)
        {       return await _context.Users
                .AsNoTracking()
                .AnyAsync(u => u.Id == id, ct);    }
        public async Task<bool> IsEmailUniqueAsync(string email, CancellationToken ct = default)
        {      return !await _context.Users
               .AsNoTracking()
               .AnyAsync(u => u.Email == email, ct);    }
        public async Task<User> CreateAsync(User user, CancellationToken ct = default)
        {       await  _context.Users.AddAsync(user, ct);
            await _context.SaveChangesAsync(ct);
            return user;    }
        public async Task<bool> UpdateAsync(UserUpdateDto user, CancellationToken ct = default)
        {       ArgumentNullException.ThrowIfNull(user);
            await _context.Users.ExecuteUpdateAsync(x=>x
                .SetProperty(u=>u.UserName,user.UserName)
                .SetProperty(u=>u.Email,user.Email),ct);
            return true;    }
        public async Task<bool> DeleteAsync(int id, CancellationToken ct = default)    {       try
            {           int affectedRows = await _context.Users
                    .Where(u => u.Id == id)
                    .ExecuteDeleteAsync(ct);
                return affectedRows > 0; // Повертає true, якщо щось було видалено
            }
```

```
            catch (DbUpdateException ex)        {
                _logger.LogError(ex, $"Помилка при видаленні користувача {id}");
                return false;       }    }}
<script setup lang="ts">
import { ref } from 'vue';
import { fetchFiles, fetchFileCount } from '@/composables/service/fileService';
import type { FileYDto } from '@/types/fileYDto';
import FileComponent from '@/components/fileComponent.vue';
import FilterComponent from '~/components/filterComponent.vue';
import UploadComponent from '@/components/uploadComponent.vue';
import { file } from '@primeuix/themes/aura/fileupload';
import type { RefSymbol } from '@vue/reactivity';
import uploadComponent from '@/components/uploadComponent.vue';
import { sortIcon } from '@primeuix/themes/aura/datatable';
definePageMeta({
  middleware: 'auth',
  layout: 'active'
})
const searchTerm = ref<string>('');
const files = ref<FileYDto[]>([]);
const isFileUpload = ref<boolean>(false);
const pageSize = ref<number>(10);
const sortBy = ref<string>('');
const sortOrder = ref<string>('');
const fileCount = ref<number>(0);
const currentPage = ref<number>(1);
const search = async () => {
  currentPage.value = 1;
  files.value = await getFiles(currentPage.value) || [];
  fileCount.value = await getFileCount() || 0;}
const getFiles = async (page: number) => {
    const response = await fetchFiles(page, pageSize.value, searchTerm.value, sortBy.value,
sortOrder.value);
  if (response.status === 200) {
    return response.data || [];
  } else {
    console.error(response.error);  }}const getFileCount = async () => {
  const countRes = await fetchFileCount();
  if (countRes.status === 200) {
    return countRes.data || 0;
    console.log(JSON.stringify(fileCount.value));
  } else {
    console.error(countRes.error);  }}
const addNewFile = () => {
  isFileUpload.value = true;}
const cancelUpload = () => {
  isFileUpload.value = false;}
const updateFiles = async (pS: number = 10, sB: string = 'asc', sO: string = 'name') => {
  pageSize.value = pS;
  sortBy.value = sB;
  sortOrder.value = sO;
  await search();}
```

```
const loadMore = async () => {
  currentPage.value++;
  const newFiles = await getFiles(currentPage.value) || [];
  files.value.push(...newFiles); // додає елементи з newFiles у кінець files};
onMounted(async () => {
  await search();
  JSON.stringify(files?.value);})
const fileCountExceeded = computed(() => {
  return (files.value?.length ?? 0) < fileCount.value;
});</script>
<template>
  <div class="flex flex-row flex-wrap gap-1 justify-center items-center">
    <div
          class="flex flex-row flex-wrap text-white w-full sm:w-1/4 md:w-1/6 lg:w-1/8 h-10
bg-indigo-500 rounded-md p-1 m-1 justify-center items-center">
      <h3 class="truncate">Files Count {{ fileCount }}</h3>
    </div>
    <button @click="addNewFile"
      class="search-button text-white w-full sm:w-1/12 h-10 bg-yellow-500 rounded-md p-1 m-1
hover:bg-yellow-600">Add</button>
       <input type="text" class="search-input rounded-md w-full sm:w-1/2 h-10 p-1 m-1"
v-model="searchTerm"
      placeholder="Search" />
    <button @click="search()"
      class="search-button text-white w-full sm:w-1/5 h-10 bg-indigo-600 rounded-md p-1 m-1
hover:bg-blue-600">Search</button>
  </div>
  <div v-if="!isFileUpload" class="flex flex-col lg:flex-row justify-center gap-2 p-2">
    <!-- FilterComponent -->
    <div class="w-full lg:w-1/4 mb-2 lg:mb-0">
      <FilterComponent class="w-full" @apply="updateFiles" />
    </div>
    <!-- File grid -->
    <div class="w-full lg:w-3/4">
       <div class="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-3 xl:grid-cols-4
gap-2">
      <div v-for="file in files" :key="file.id">
        <FileComponent :file="file"
          class="hover:scale-105 transition-all duration-100 hover:bg-gray-100 rounded-md" />
      </div>
    </div>
  </div>
</div>
    <div v-if="!isFileUpload && fileCountExceeded" class="flex flex-row justify-center
items-center">
      <button @click="loadMore" class="bg-green-600 text-white rounded-md shadow-md p-2
m-2">Load More</button>
  </div>
  <UploadComponent v-else-if="isFileUpload" @exit="cancelUpload" @upload="search" />
</template>
```