МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Дослідження методів оптимізації та підвищення продуктивності Web -додатків**

Виконав: студент 2 курсу, групи 8.1213-іпз-2С
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
А. І. Артеменко
(ініціали та прізвище)
Керівник доцент, к.т.н. О.М. Міхайлуца
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____ другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т. В. Критська
“_02_” ___вересня__ 2024_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Артеменку Артуру Ігоровичу
(прізвище, ім’я, по батькові)

1.Тема роботи Дослідження методів оптимізації та підвищення продуктивності Web-додатків
керівник роботи Міхайлуца Олена Миколаївна, к.т.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 28.02.2024р. № 361-е

2. Строк подання студентом кваліфікаційної роботи 30.11.2024
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми створення інформаційних систем;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

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

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 02.09.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз предметної області	11.09.24	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	12.09.24	виконано
3	Аналіз існуючих методів рішення	13.09-14.09.24	виконано
4	Дослідження сучасних засобів оптимізації	15.09-16.09.24	виконано
5	Узгодження подальших дій з науковим керівником	18.09.24	виконано
6	Проектування архітектури додатку	19.09-30.09.24	виконано
7	Програмна реалізація додатку	01.10-18.10.24	виконано
8	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	19.10.24	виконано
9	Перевірка роботоздатності веб додатку	20.10-21.10.24	виконано
10	Оформлення звіту	22.10-10.11.24	виконано
11	Оформлення презентації. Отримання рецензій від опонентів.	11.11-12.11.24	виконано

Студент _____ А. І. Артеменко
(підпис) (прізвище та ініціали)

Керівник роботи _____ О.М. Міхайлуца
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено
Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок – 79

Рисунків – 12

Джерел – 21

Артеменко А.І. Дослідження методів оптимізації та підвищення продуктивності Web-додатків: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник О.М. Міхайлуца. Запоріжжя : ЗНУ, 2024.79 с.

Мета дослідження полягає у вивченні сучасних підходів до оптимізації веб-застосунків для забезпечення їх більшої продуктивності, стабільної роботи та зручності використання користувачами. Також робота зосереджена на дослідженні сучасних методів оптимізації фронтенду та бекенду, мінімізації ресурсів, які будуть використані для обробки запитів, а також оптимізації роботи з базами даних.

У процесі дослідження було проведено аналіз існуючих методів і технологій до оптимізації веб-застосунків, а також вплив на користувацький досвід та ефективність використання ресурсів. Для ілюстрації описаних методів був розроблений веб-додаток для хмарного зберігання файлів, що демонструє практичне застосування досліджених методів оптимізації, також використання PostgreSQL, Redis, AWS S3, сучасних фреймворків для фронтенд та бекенд частини.

Ключові слова: оптимізація веб-застосунків, продуктивність, кешування, PostgreSQL, Redis, AWS S3.

ABSTRACT

Pages – 79

Drawings – 12

Sources - 21

Artemenko A.I. Research of methods of optimisation and performance improvement of Web-applications: master's thesis in speciality 121 ‘Software Engineering’ / scientific supervisor O.M. Mikhailutsa. Zaporizhzhia: ZNU, 2024. 79 p.

The purpose of the study is to investigate modern approaches to optimising web applications to ensure their greater performance, stable operation and ease of use by users. The work also focuses on the study of modern methods of front-end and back-end optimisation, minimisation of resources that will be used to process requests, and optimisation of work with databases.

The study analysed existing methods and technologies for optimising web applications, as well as the impact on user experience and resource efficiency. To illustrate the described methods, a web application for cloud file storage has been developed, which demonstrates the practical application of the studied optimisation methods, as well as the use of PostgreSQL, Redis, AWS S3, modern frameworks for the front- and backend.

Keywords: web application optimisation, performance, caching, PostgreSQL, Redis, AWS S3.

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 13
1.1 Огляд літературних джерел 13
1.2 Аналіз програмних продуктів-аналогів 14
1.3 Постановка завдання 18
1.4 Висновки до розділу 1 19
2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ 21
2.1 PostgreSQL як основна база даних 21
2.2 Redis 22
2.3 Хмарні методи зберіагання даних 24
2.4 Фреймворк для написання серверної частини 25
2.5 Фреймворк для написання користувацьї частини 27
2.6 Висновки до розділу 2 28
3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ХМАРНОГО ЗБЕРІГАННЯ ФАЙЛІВ 30
3.1 Опис предметної області 30
3.2 Архітектура системи 33
3.3 Функціональні вимоги системи 34
3.4 Проєктування інформаційної системи 35
3.5 Програмна реалізація застосунку 40
3.6 Розробка інтерфейсу інформаційної системи 59
3.7 Висновки до розділу 3 67
4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ МЕТОДІВ ОПТИМІЗАЦІЇ ТА ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ WEB -ДОДАТКІВ 69
4.1 Результати дослідження баз даних 69

4.2 Результати дослідження використання сидів для розгортання бази даних 71

4.3 Результати дослідження використання буферного підхоту та ReadStream для передачі файлів у хмарне сховище 72

4.4 Результати дослідження використання “Lazy loading”..... 74

4.5 Результати оптимізації додатку..... 75

ВИСНОВКИ..... 77

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..... 78

ВСТУП

Актуальність теми

70% українців щодня користуються веб-додатками, від соціальних мереж та онлайн-торгівлі до онлайн-банкінгу та освіти (за даними Statista). Різноманіття цих додатків вражає: Facebook, Instagram, Rozetka, Prom, Приват24, Monobank, Prometheus, Coursera, Telegram, Viber – це лише деякі з них.

За останній рік 40% українців почали використовувати більше веб-додатків (GfK Ukraine), що свідчить про стрімке зростання їх популярності.

Швидкість роботи веб-додатку має вирішальне значення для успіху. 53% користувачів не рекомендують веб-додаток, який завантажується більше 3 секунд (Google), а 80% користувачів зазначають, що швидкість роботи веб-додатку впливає на їхнє враження (Forrester Research).

Веб-додатки працюють на різних пристроях (смартфони, планшети, ПК, носимі гаджети) з різними платформами (Android, iOS, Windows) та браузерами (Chrome, Safari, Firefox), що робить їх оптимізацію складним завданням. Додатковою складністю є різниця в швидкості інтернет-з'єднання.

Неоптимізовані веб-додатки можуть стати причиною серйозних проблем, які безпосередньо впливають на їхню ефективність та зручність використання. Основні наслідки відсутності оптимізації включають повільне завантаження сторінок, часті зависання, некоректне відображення контенту, а в окремих випадках – втрату даних. Ці проблеми знижують загальну якість користувацького досвіду, негативно впливають на SEO та можуть призвести до втрати конкурентоспроможності на ринку.

Оптимізація на стороні сервера є ключовим етапом підвищення продуктивності веб-додатків. Вона включає використання кешування, наприклад, за допомогою Redis та Memcached, що дозволяє тимчасово зберігати дані і знижувати навантаження на базу даних, прискорюючи виконання повторних запитів. Мініфікація серверного коду за допомогою

інструментів, таких як UglifyJS і Closure Compiler, сприяє зменшенню розміру скриптів, що позитивно впливає на швидкість їх обробки. Оптимізація запитів до бази даних, зокрема використання індексації та удосконалення структури запитів у PostgreSQL чи MySQL, значно скорочує час доступу до даних. Крім того, балансування навантаження між кількома серверами через Nginx або HAProxy забезпечує стабільність системи навіть при високій інтенсивності трафіку.

Оптимізація на стороні клієнта зосереджена на зменшенні обсягу переданих даних і покращенні взаємодії з користувачами. Мініфікація HTML, CSS та JavaScript, зокрема через MinifyCode або HTMLMinifier, дозволяє суттєво скоротити час завантаження сторінок. Оптимізація мультимедійного контенту, наприклад, через сервіси TinyPNG і Kraken.io, зменшує вагу зображень, не погіршуючи їхньої якості. Використання лінивого завантаження (LazyLoad, Infinite Scroll) дозволяє завантажувати контент поступово, знижуючи навантаження на мережу. CDN, такі як Cloudflare та Akamai, забезпечують швидку доставку ресурсів незалежно від географічного розташування користувачів, підвищуючи їхній загальний досвід взаємодії із системою.

Мета і завдання дослідження

Мета даного дослідження полягає в аналізі та порівнянні різних методів оптимізації для веб-додатків, а також розробці рекомендацій для їх практичного застосування. Завдання включають вивчення існуючих підходів до оптимізації, визначення їхнього впливу на продуктивність та розробку нових методів для підвищення швидкодії веб-додатків.

Об’єкт дослідження

Об'єктом дослідження є веб-додатки різних типів та призначень.

Предмет дослідження

Предметом дослідження є методи, технології та інструменти для оптимізації та підвищення продуктивності веб-додатків.

Методи дослідження

У процесі дослідження буде застосовано комплекс методів для досягнення поставлених цілей:

1. Аналіз та узагальнення літератури:
    a. Проводиться обширний аналіз наявної наукової літератури та публікацій, що стосуються оптимізації веб-додатків.
    b. Вивчення робіт інших вчених дозволить здобути повний огляд існуючих методів та їх ефективності.
2. Вивчення технічної документації та кейсів:
    a. Аналіз технічної документації популярних веб-додатків та вивчення практичних кейсів оптимізації.
    b. Вивчення реальних сценаріїв використання методів оптимізації допоможе зрозуміти їхні переваги та обмеження в конкретних умовах.
3. Аналіз результатів та висновки:
    a. Вивчення отриманих результатів експериментів та їх аналіз.
    b. Формулювання висновків щодо ефективності різних методів оптимізації та визначення їхнього потенціалу для підвищення продуктивності веб-додатків.

Цей комплекс методів дозволяє систематизувати наявні знання, визначити нові підходи та надати практичні рекомендації для оптимізації веб-додатків з урахуванням конкретних умов їхньої експлуатації.

Наукова новизна одержаних результатів

В результаті проведеного дослідження систематизовано та проаналізовано різноманітні існуючі методи оптимізації веб-додатків. На основі цього аналізу розроблено комплексний підхід до оптимізації, який

враховує різні аспекти веб-додатків, та запропоновано практичні рекомендації щодо ефективного впровадження цих методів. Важливим етапом дослідження є експериментальне підтвердження ефективності запропонованих методів.

Практичне значення одержаних результатів

Результати нашого дослідження, представлені в науковій роботі, не тільки нададуть розробникам веб-додатків, власникам веб-ресурсів та компаніям, що надають веб-послуги, практичні рекомендації та експериментальні дані для підвищення продуктивності їхніх веб-додатків та покращення користувацького досвіду, але й стануть ключовим інструментом у виборі найбільш ефективних методів оптимізації, враховуючи конкретні вимоги та умови експлуатації. Отримані результати, сприяючи підвищенню продуктивності реальних веб-додатків, відзначаються не лише поліпшенням користувацького досвіду, але й відзначаються зниженням витрат на інфраструктуру серверів для підприємств.

Апробація одержаних результатів

Результати дослідження були представлені на науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2024» [22], а також на I Міжнародній науково-практичній конференції «Інженерні інновації та розбудова національної економіки» [23].

Глосарій

Веб-додаток (Web Application) — додаток, який працює у веб-оглядачі та використовує веб-технології (HTML, CSS, JavaScript тощо).

Оптимізація (Optimization) — процес вдосконалення та покращення ефективності роботи системи або додатку.

Продуктивність (Performance) — показник, що характеризує швидкість та ефективність виконання певних операцій або завдань.

Кешування (Caching) — тимчасове зберігання даних для швидкого доступу в майбутньому.

Мінімізація (Minification) — процес видалення зайвих пробілів, коментарів та форматування коду для зменшення розміру файлів.

Стиснення (Compression) — зменшення розміру файлів або даних за допомогою спеціальних алгоритмів.

Лейзі-завантаження (Lazy Loading) — техніка, яка завантажує ресурси (зображення, дані тощо) лише тоді, коли вони дійсно потрібні.

Рендеринг (Rendering) — процес генерації візуального представлення веб-сторінки на основі коду.

Клієнтські ресурси (Client Resources) — ресурси, що використовуються на стороні клієнта (веб-оглядач), такі як CPU, ОЗУ, графічний процесор тощо.

Серверні ресурси (Server Resources) — ресурси, що використовуються на стороні сервера, який обслуговує веб-додаток, такі як CPU, ОЗУ, дискове сховище тощо.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд літературних джерел

Оптимізація веб-застосунків є ключовим етапом у забезпеченні їх ефективності, стабільності та користувацького досвіду, особливо у випадках високого навантаження чи роботи з великими обсягами даних. Сучасні дослідження підкреслюють важливість швидкості завантаження для утримання користувачів і підвищення рентабельності. Затримка навіть у кілька сотих секунди може призводити до значного зниження конверсій і збільшення відтоку користувачів [1]. Google у своїх аналітичних дослідженнях вказує, що додаткові 2 секунди завантаження можуть скоротити конверсії на 50% [2].

Фронтенд-оптимізація є важливим компонентом підвищення продуктивності веб-застосунків, оскільки саме цей шар найбільше впливає на сприйняття швидкості користувачами. Сучасні підходи включають зменшення обсягу переданих даних за допомогою таких технологій, як стиснення файлів (наприклад, Gzip, Brotli) і мінімізація JavaScript та CSS. Впровадження ленивого завантаження (lazy loading) зображень та відео знижує навантаження на мережу й забезпечує значно кращий досвід роботи з динамічними сторінками [3].

Прогресивні веб-застосунки (PWA) використовують кешування через Service Workers, що дозволяє значно зменшити затримки при повторному доступі до сторінок. Це рішення забезпечує швидке завантаження контенту навіть у разі обмеженого доступу до мережі [4].

Оптимізація бекенду зосереджується на зниженні часу відповіді сервера та підвищенні ефективності використання ресурсів. Одним із найважливіших аспектів є оптимізація роботи баз даних. Використання індексації, нормалізації, оптимізації складних запитів і кешування результатів дозволяє

значно знизити час обробки запитів. Наприклад, дослідження, демонструють, що індексація дозволяє скоротити час виконання складних запитів до 70% [5].

Важливу роль також відіграє використання розподілених архітектур, таких як мікросервіси. Цей підхід дозволяє розбити монолітні системи на незалежні модулі, що забезпечує кращу масштабованість і відмовостійкість. Така архітектура спрощує управління ресурсами і дозволяє швидше реагувати на зміну навантаження [6].

Для прискорення передачі даних використовуються CDN (Content Delivery Networks), які зменшують затримки завдяки розподілу контенту між фізично близькими до користувачів серверами. HTTP/2 та HTTPS також суттєво покращують продуктивність передачі [7]. Впровадження сучасних API, дозволяє отримувати лише необхідні дані, що зменшує обсяг інформації, яку потрібно передати.

Таким чином, оптимізація веб-застосунків є багатогранним процесом, що включає вдосконалення роботи клієнтської, серверної частин, баз даних, а також мережевих компонентів. Інтеграція інструментів і технологій оптимізації, таких як індексація баз даних, CDN, PWA, та GraphQL, забезпечує високу продуктивність, стабільність і зручність для користувачів. Наукові дослідження та практика підтверджують, що ці підходи є необхідними для забезпечення конкурентоспроможності в сучасному цифровому середовищі.

1.2 Аналіз програмних продуктів-аналогів

Хмарні сервіси для зберігання файлів стрімко розвиваються, забезпечуючи користувачам широкі можливості для управління, збереження, а також для спільного використання даних. Розглянемо декілька популярних програмних продуктів-аналоги, які конкурують у цій галузі.

Google Drive (див. рис. 1) — один із найпопулярніших хмарних сервісів, що надає користувачам 15 ГБ безкоштовного сховища для файлів, документів, фото та відео. Основні функції:

- можливість спільного доступу до файлів з різними рівнями дозволів (перегляд, редагування, коментування);
- інтеграція з іншими сервісами Google, такими як Google Docs, Sheets і Gmail;
- функція автоматичного резервного копіювання на мобільних пристроях.
- Інтерфейс зручний і доступний як через веб-додаток, так і через мобільний застосунок.

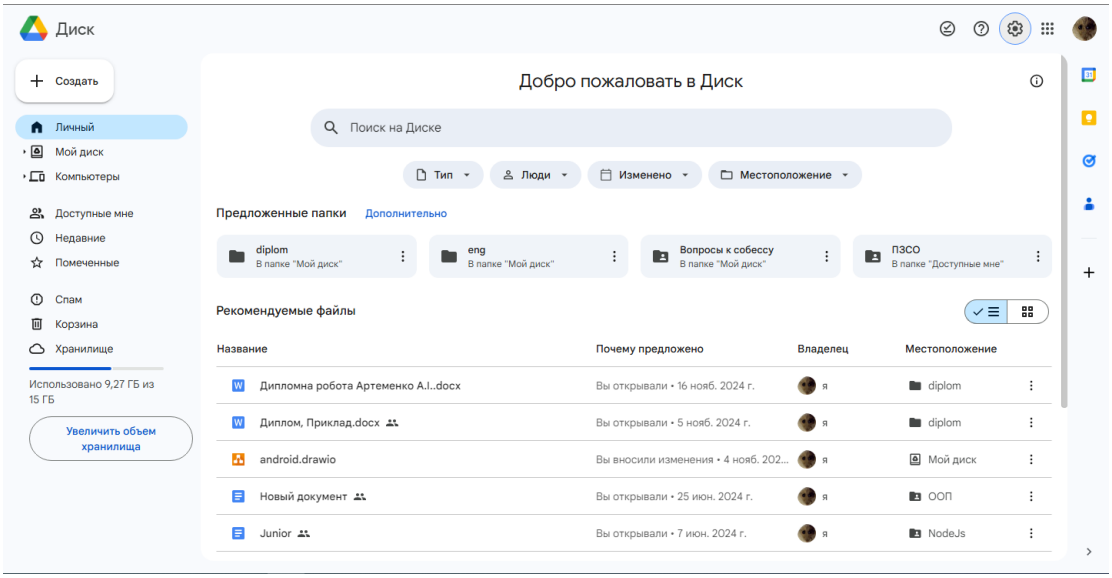


Рисунок 1 — Головна сторінка Google Drive

Dropbox (див. рис. 2) — сервіс, який був орієнтований перш за все на користувачів, потреби яких полягають у простоті у використанні. Він пропонує безкоштовні 2 ГБ сховища, з можливістю розширення сховища, якщо користувачу не вистачає місця. Особливості:

- автоматична синхронізація файлів між пристроями;
- функція "Rewind" для відновлення файлів після змін або видалення;
- інтеграція з різними офісними інструментами, такими як Microsoft Office.

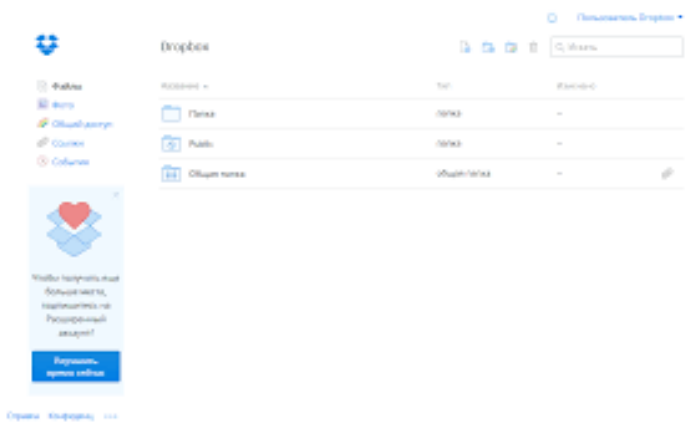


Рисунок 2 — Головна сторінка Dropbox

OneDrive (див. рис. 3) — сервіс від Microsoft, який було вдало інтегровано інтегровано з операційними системами Windows 10 і 11, а також з пакетом Microsoft Office. Він надає 5 ГБ безкоштовного місця та низку функцій, серед яких:

- автоматичне резервне копіювання файлів із комп’ютера;
- вбудовані інструменти для співпраці над документами у Word, Excel, PowerPoint;
- можливість роботи з файлами в режимі офлайн.

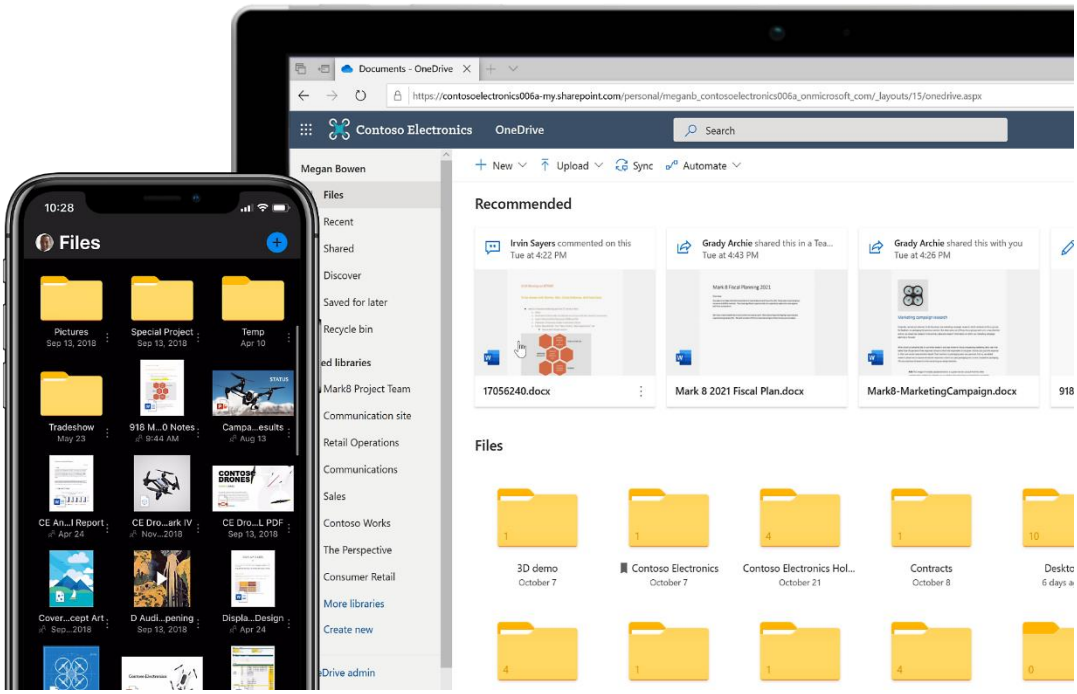


Рисунок 3 — Головна сторінка OneDrive

pCloud (див. рис. 4) — сервіс, який підійде для користувачів, яким потрібен високим рівнем безпеки. Серед його особливостей:

- клієнтське шифрування, яке гарантує що кожен ваш файл буде надійно захищеним й навіть від самого сервісу;
- можливість купити "довічну ліцензію" для зберігання файлів;
- інструменти для прослуховування музики та перегляду відео без їх завантаження.

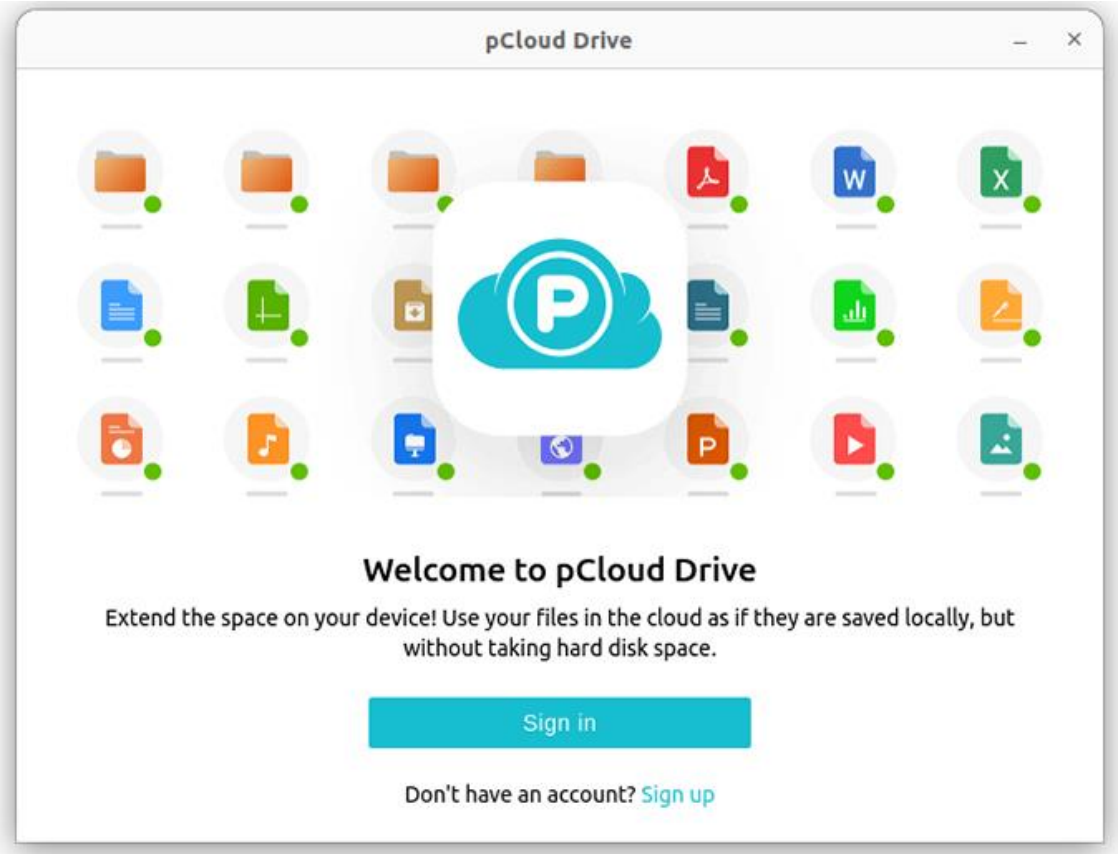


Рисунок 4 — Головна сторінка pCloud

При розробці власного сервісу потрібно використовувати переваги вже існуючих рішень і розробити застосунок, який буде враховувати сучасних тенденції у оптимізації процесів. Для такого проекту потрібно мати свою базу даних, яка дасть можливість швидко оброблювати запити, та буде обробляти

великий обсяг даних за малий час, що, в свою чергу, дозволить користувачам швидко користуватися сервісом без затримок.

1.3 Постановка завдання

Мета роботи – це оптимізація веб-застосунку для підвищення його продуктивності, забезпечення швидкості завантаження та зниження витрат на ресурси. Покращення показників продуктивності веб-додатків дозволить забезпечити кращий користувацький досвід і ефективне використання серверних потужностей.

Для ілюстрації методів оптимізації буде розроблено застосунок для хмарного збереження файлів. Цей застосунок дозволить користувачам завантажувати, зберігати та ділитися файлами з максимальною продуктивністю. Дані будуть зберігатися в хмарному середовищі з розподіленою архітектурою, що забезпечить високу доступність і масштабованість. Застосунок стане прикладом високопродуктивного, надійного та зручного сервісу, що демонструє переваги сучасних підходів до оптимізації веб-застосунків.

Для досягнення поставленої мети потрібно вирішити наступні задачі:

1. Аналіз і вимірювання продуктивності веб-застосунку:
    - Визначити вузькі місця, які призводять до затримок у роботі веб-застосунку.
2. Оптимізація фронтенду:
    - Оптимізувати розмір і завантаження зображень, відео, шрифтів та інших мультимедійних ресурсів;
    - Зменшити кількість HTTP-запитів, об’єднавши та мінімізувавши файли CSS, JavaScript; Впровадити техніки асинхронного та відкладеного завантаження ресурсів (lazy loading).
3. Оптимізація бекенду:

- Оптимізувати запити до бази даних, використовуючи індекси, нормалізацію та кешування;
- Впровадити серверне кешування (Redis) для зменшення навантаження на базу даних.

4. Забезпечення безпеки та стабільності:
   - Захистити веб-застосунок від основних вразливостей, таких як XSS, CSRF, SQL-ін'єкції;
   - Впровадити моніторинг та логування для швидкого виявлення та усунення проблем.

Результатом роботи має стати більш продуктивний, безпечний і стабільний веб-застосунок, який відповідає сучасним вимогам щодо швидкості та ефективності роботи.

1.4 Висновки до розділу 1

Проведено огляд літературних джерел, що підтверджують важливість оптимізації веб-застосунків для забезпечення їхньої ефективності, стабільності та зручності використання. Визначено, що сучасні методи оптимізації включають як фронтенд, так і бекенд підходи, а також використання інструменту як кешування, що значно підвищують продуктивність веб-додатків.

Проаналізовано існуючі хмарні сервіси для зберігання файлів (Google Drive, Dropbox, OneDrive, pCloud) для визначення їхніх переваг і недоліків. Встановлено, що ключовими факторами успішності цих платформ є швидкість роботи, інтеграція з іншими сервісами, зручність використання та високий рівень безпеки.

Сформульовано завдання дослідження, яке включає оптимізацію веб-застосунків шляхом ідентифікації вузьких місць у продуктивності, оптимізації мультимедійного контенту, запитів до бази даних, а також забезпечення безпеки. Основною метою роботи є створення високопродуктивного веб-

додатка для хмарного збереження файлів, що демонструє сучасні підходи до оптимізації.

Запропоновано реалізацію застосунку з використанням розподіленої архітектури та оптимізованих методів роботи з фронтендом і бекендом. Це дозволить забезпечити високу доступність, масштабованість та ефективне управління ресурсами, а також продемонструвати на практиці переваги сучасних технологій у розробці веб-додатків.

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 PostgreSQL як основна база даних

Вибір PostgreSQL як основної бази даних для реалізації сучасних програмних рішень базується на її багатому функціональному наборі, високій продуктивності та відповідності вимогам масштабованих і складних систем. У ході аналізу наявних рішень для роботи з реляційними базами даних було розглянуто такі альтернативи, як MySQL, MariaDB, SQLite та NoSQL рішення, включаючи MongoDB. Висновок на користь PostgreSQL зроблено на основі її технологічної переваги та практичних експериментів, що підтверджуються сучасними науковими дослідженнями.

PostgreSQL вирізняється своєю повною підтримкою ACID-транзакцій, що робить її надійним вибором для критичних бізнес-додатків, які потребують збереження консистентності даних. Згідно з дослідженнями, PostgreSQL забезпечує вищу надійність у роботі зі складними запитами, ніж її найближчі конкуренти, включаючи MySQL [8]. Завдяки підтримці складних типів даних, таких як JSONB, масиви, геопросторові дані, PostgreSQL є універсальним інструментом для обробки гетерогенних наборів даних, що є важливим у сучасних мультидоменних додатках.

Експерименти, проведені з використанням реальних сценаріїв, підтвердили перевагу PostgreSQL у продуктивності для великих обсягів даних. Наприклад, у порівнянні з MySQL та MariaDB, PostgreSQL демонструє стабільно вищу швидкість виконання запитів, що стосуються агрегатів та складних обчислень [9]. Крім того, PostgreSQL є лідером у реалізації розширених функцій індексації, таких як GIN та GiST, що дозволяє суттєво прискорити виконання складних запитів.

Додатковою перевагою PostgreSQL є її активна спільнота розробників та велика кількість інструментів для інтеграції з іншими технологіями. Наприклад, можливість роботи з хмарними платформами, такими як AWS

RDS чи Google Cloud SQL, розширює можливості масштабування та управління базою даних у глобальних розподілених системах. У дослідженнях, PostgreSQL була відзначена як одна з найкращих баз даних для інтеграції з хмарними рішеннями, завдяки її адаптивності до сучасних архітектур [10].

Отже, вибір PostgreSQL як основної бази даних базується на її технологічній перевазі, підтвердженій науковими дослідженнями та практичним досвідом роботи. Ця база даних є оптимальним рішенням для реалізації надійних, масштабованих та високопродуктивних додатків, що відповідають вимогам сучасного ринку.

2.2 Redis

Redis був обраний як кешована база даних для проєкту завдяки його високій швидкості, масштабованості та багатофункціональності, що робить його оптимальним вибором для побудови систем із низькою затримкою доступу до даних. У процесі аналізу альтернатив були розглянуті такі рішення, як Memcached та інші сховища в оперативній пам’яті. Основні аргументи на користь Redis базуються на його технічних характеристиках, сучасних дослідженнях та практичних рекомендаціях із літератури.

Redis є базою даних типу "ключ-значення", яка зберігає дані в оперативній пам’яті, що забезпечує миттєвий доступ до них. Redis перевершує більшість аналогів завдяки своїй багатофункціональній структурі даних, серед яких списки, множини, хеші та впорядковані множини [11]. Це дозволяє використовувати Redis не лише для кешування, але й для інших завдань, таких як відстеження сеансів, черги повідомлень або аналітичні операції.

Дослідження, також підтверджують, що Redis забезпечує значно кращу продуктивність у порівнянні з іншими кешованими рішеннями завдяки підтримці асинхронного збереження даних на диск (через AOF та RDB), що робить його придатним для систем із високими вимогами до надійності [12].

На відміну від Memcached, який обмежується лише функцією кешування, Redis дозволяє створювати складні сценарії роботи з даними завдяки вбудованій підтримці Lua-скриптів.

У рамках нашого проєкту Redis використовується як проміжний кеш для зменшення навантаження на основну базу даних PostgreSQL. Використання Redis дозволяє суттєво знизити час відповіді системи, оскільки часто запитувані дані знаходяться в оперативній пам’яті та доступні практично миттєво. Це особливо актуально для сценаріїв із великою кількістю одночасних запитів, таких як відображення часто запитуваних даних у реальному часі.

Однією з ключових причин вибору Redis є його висока продуктивність, яка базується на архітектурі роботи в оперативній пам’яті. Згідно з результатами тестів, Redis може обробляти понад 1 мільйон запитів на секунду у режимі кластеризації [13]. Така продуктивність дозволяє забезпечити низьку затримку навіть за умов великого навантаження на систему. Крім того, Redis легко масштабується завдяки підтримці кластерів та шардування, що робить його придатним для використання в проєктах з потенційно великою кількістю користувачів.

Вибір Redis як кешованої бази даних був зумовлений його продуктивністю, широким спектром можливостей і здатністю забезпечувати стабільну роботу системи навіть за високого навантаження. У порівнянні з іншими рішеннями, такими як Memcached, Redis пропонує більше гнучкості завдяки розширеним структурам даних і можливостям збереження даних на диск. Літературні джерела та результати досліджень підтверджують доцільність використання Redis для реалізації швидких і масштабованих систем, що відповідають сучасним вимогам до продуктивності.

2.3 Хмарні методи зберіагання даних

Зберігання даних у хмарних сховищах стало важливою складовою сучасних програмних систем завдяки їх гнучкості, масштабованості та економічній ефективності. У порівнянні з традиційними локальними рішеннями, хмарні сховища пропонують низку переваг, які роблять їх незамінними для додатків, що працюють з великими обсягами даних або мають динамічне навантаження.

Однією з ключових переваг хмарних сховищ є їх висока доступність та відмовостійкість. Наприклад, у разі виходу з ладу фізичного обладнання дані залишаються доступними завдяки розподіленій архітектурі, яка використовує реплікацію для збереження даних на кількох серверах. Це особливо важливо для сучасних веб-додатків, де кожна секунда простою може призвести до втрати користувачів або доходу. Хмарні сховища також забезпечують автоматичне масштабування, що дозволяє динамічно адаптувати ресурси під зростання обсягу даних або збільшення запитів [14].

Серед основних переваг хмарних сховищ варто виділити можливість зменшення витрат на підтримку фізичної інфраструктури, доступ до даних у будь-якому місці та інтеграцію з іншими сервісами через API. Однак існують і певні недоліки, такі як залежність від постачальника послуг, можливі затримки при передачі даних через мережу, а також питання безпеки даних, особливо у випадках зберігання конфіденційної інформації.

Для реалізації функції зберігання файлів у додатку було обрано AWS S3 як основний хмарний сервіс. AWS S3 (Simple Storage Service) є однією з найбільш популярних платформ для зберігання даних, яка пропонує високу доступність (99.99%) та безперервну реплікацію даних у кількох регіонах. AWS S3 забезпечує інтеграцію з іншими сервісами AWS, такими як Lambda, CloudFront або DynamoDB, що дозволяє створювати складні та продуктивні системи [15].

Ключовими перевагами AWS S3 у порівнянні з іншими платформами, наприклад Google Cloud Storage або Azure Blob Storage, є його економічна модель оплати "pay-as-you-go", гнучкість у створенні політик доступу (через IAM) і вбудована підтримка різних класів зберігання, таких як S3 Glacier для архівних даних. Крім того, AWS S3 підтримує шифрування на стороні сервера (SSE) та стороні клієнта, що забезпечує високий рівень безпеки для чутливих даних.

Використання хмарного сховища, зокрема AWS S3, забезпечує не лише високу продуктивність і масштабованість системи, але й значно спрощує управління даними завдяки інтеграції з іншими хмарними сервісами. Незважаючи на певні обмеження, такі як затримки мережі, переваги у вигляді економічності, доступності та безпеки роблять AWS S3 оптимальним вибором для сучасних програмних рішень. Дослідження та літературні джерела підтверджують, що використання AWS S3 є ефективним підходом для створення систем зі складними вимогами до зберігання даних.

2.4 Фреймворк для написання серверної частини

У якості фреймворку для написання серверної частини було обрано Fastify. Вибір Fastify, як основного фреймворку для розробки серверної частини застосунку, зумовлений через певний ряд причин: його високою продуктивністю, розширюваністю та сучасною архітектурою. У процесі аналізу, з даним фреймворком, були порівняні популярні Node.js-фреймворки, зокрема Fastify та Express, які є широко використовуваними рішеннями для створення веб-додатків. Вибір на користь Fastify зроблено на основі кількісних показників продуктивності, функціональних переваг і відповідності сучасним практикам розробки.

Fastify пропонує більш сучасний підхід до обробки запитів завдяки суворій схемній валідації, яка під час ініціалізації проекту, інтегрованій підтримці плагінів та розширених функцій для оптимізації [16]. На відміну від

Express, який має простішу архітектуру, Fastify забезпечує більши детальний контроль над обробкою запитів, що, в свою чергу, дозволяє розробникам будувати надійні та масштабовані системи.

Однією з ключових особливостей Fastify є його суворе управління схемами через JSON Schema, що дозволяє автоматично валідувати запити та відповіді. Цей підхід значно зменшує кількість помилок під час виконання та забезпечує кращу інтеграцію, також він надає більш детальну валідацію полей, які надсилає користувач та надає детальну відповідь, якщо користувач ввів невалідний формат. У порівнянні з Express, який вимагає додаткових бібліотек для досягнення подібного функціоналу, Fastify надає вже готове рішення, без встановлення нових npm-залежностей.

Одним з головних аргументів на користь Fastify є його продуктивність. Згідно з тестами, Fastify демонструє вищу швидкість обробки запитів порівняно з Express [17]. Це досягається завдяки меншій кількості операцій I/O, які виконуються на кожен запит, та ефективнішому управлінню асинхронністю.

На рисунку 5 наведено результати тестування продуктивності. Вони чітко демонструють перевагу Fastify у швидкості обробки запитів, особливо під час високого навантаження. У той час, як Express починає демонструвати спад у продуктивності, при великій кількості одночасних з'єднань, Fastify забезпечує більш стабільну роботу застосунку завдяки оптимізації архітектури самого фреймворку.

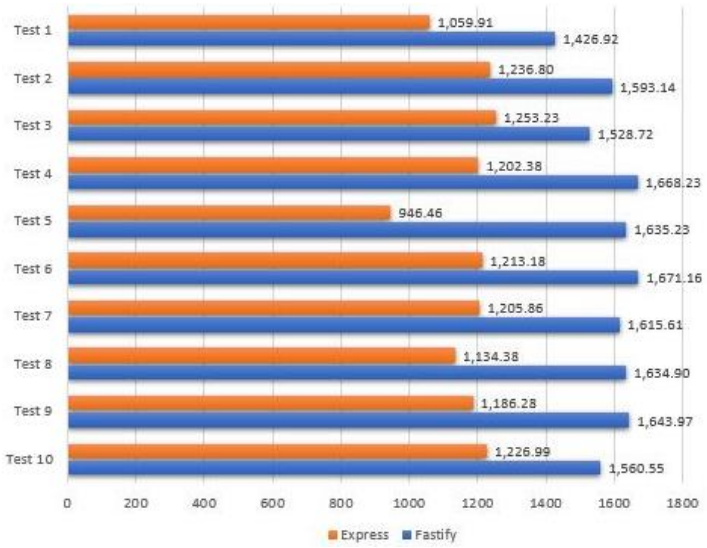


Рисунок 5 — *Порівняння швидкодії Fastify з Express*

Загалом, вибір Fastify як основи для серверної частини обумовлений його високою продуктивністю, сучасним підходом до розробки та розширюваною архітектурою. У порівнянні з Express, Fastify пропонує більше можливостей для побудови масштабованих систем і забезпечує кращу інтеграцію з сучасними стандартами API. Літературні джерела та результати тестування підтверджують, що Fastify є ефективним інструментом для реалізації продуктивних та стабільних серверних рішень.

2.5 Фреймворк для написання користувацьї частини

React був обраний для створення інтерфейсу користувача завдяки його гнучкості, багатому екосистемному середовищу та підтримці сучасних інструментів розробки. У цьому проєкті React поєднується з Vite як інструментом для білду та Tailwind CSS для стилізації, що забезпечує високий рівень продуктивності й простоту в реалізації.

React, як фреймворк для побудови компонентно-орієнтованих інтерфейсів, демонструє більшу універсальність у порівнянні з іншими популярними фреймворками, такими як Vue.js чи Angular. React відрізняється простотою у використанні завдяки своїй бібліотечній структурі,

що дозволяє розробникам створювати легкі та динамічні інтерфейси з мінімальними витратами на навчання [18]. На відміну від Angular, React є більш модульним, дозволяючи розробникам обирати додаткові інструменти, такі як Vite або Webpack, залежно від потреб проєкту.

Vite був обраний як інструмент для білду завдяки його швидкості та підтримці сучасних JavaScript-функцій. У порівнянні з традиційним Webpack, Vite демонструє суттєво швидший час білду та гарячого перезавантаження [19]. Використання Vite дозволяє зменшити час розробки, особливо при роботі з великими проєктами.

Tailwind CSS надає гнучкість у стилізації, що значно перевершує традиційні CSS-фреймворки, такі як Bootstrap. Tailwind забезпечує утилітарний підхід до стилів, дозволяючи уникати перевантаження HTML зайвими класами. Цей інструмент значно зменшує час розробки інтерфейсів і покращує читабельність коду, що робить його ідеальним вибором для комплексних систем [20].

Окрему увагу слід приділити продуктивності вибраного стеку технологій. React у поєднанні з Vite демонструє швидший час рендерингу компонентів у порівнянні з Vue.js та Angular, особливо у великих SPA (Single Page Applications). Використання Vite як інструмента для білду забезпечує швидше завантаження ресурсів у порівнянні з Webpack [21].

Tailwind CSS також вносить вклад у продуктивність, оскільки дозволяє мінімізувати обсяг згенерованого CSS-коду завдяки механізмам tree-shaking. Це зменшує розмір кінцевих файлів стилів, що покращує швидкість завантаження сторінок і оптимізує користувацький досвід.

2.6 Висновки до розділу 2

1. Проведено аналіз сучасних баз даних, включаючи реляційні та NoSQL рішення, і обґрунтовано вибір PostgreSQL як основної бази даних для

реалізації програмного забезпечення завдяки її підтримці ACID-транзакцій, складних типів даних та розширеної системи індексації.

2. Розглянуто використання Redis як кешованої бази даних, що забезпечує високу швидкість доступу до даних, низьку затримку і підтримку складних структур даних, що підтверджує його доцільність для систем з високими вимогами до продуктивності.
3. Виявлено переваги використання хмарного сховища AWS S3 для зберігання файлів, включаючи високу доступність, гнучкість, інтеграцію з іншими хмарними сервісами та безпеку, що робить його оптимальним вибором для зберігання великих обсягів даних.
4. Оцінено ефективність фреймворку Fastify для створення серверної частини додатків, який демонструє високу продуктивність, гнучкість у розробці та сучасний підхід до обробки запитів, що перевершує альтернативи, такі як Express.
5. Досліджено використання React з Vite і Tailwind CSS для розробки інтерфейсів користувача, що забезпечує високу швидкість рендерингу компонентів, зручність розробки та гнучкість у створенні сучасних, продуктивних інтерфейсів.

3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ХМАРНОГО ЗБЕРІГАННЯ ФАЙЛІВ

3.1 Опис предметної області

Веб-застосунки — це інтерактивні програми, які функціонують на веб-серверах і взаємодіють з користувачами через веб-браузери. Вони мають широкий спектр використання: від простих статичних сторінок до складних динамічних систем, таких як онлайн-магазини, CRM-системи та соціальні мережі.

Оптимізація веб-застосунків полягає в підвищенні ефективності роботи цих систем шляхом зменшення часу завантаження сторінок, зниження використання ресурсів, підвищення відмовостійкості, забезпечення безперебійної роботи при високих навантаженнях і покращення користувацького досвіду. В сучасних умовах, коли веб-застосунки обслуговують мільйони користувачів, їх продуктивність стає ключовим фактором для успішного функціонування бізнесу.

Основні компоненти предметної області

1. Клієнтська частина (фронтенд):
    - HTML/CSS: основа веб-застосунку, яка відповідає за структуру і стилі сторінки. Оптимізація цього компонента дозволяє зменшити об’єм даних, що передаються від сервера до клієнта.
    - JavaScript: використовується для інтерактивності та динамічних функцій веб-застосунку. Його оптимізація може зменшити час завантаження і покращити взаємодію з користувачем.
    - Зображення та мультимедійний контент: важливий компонент сучасних веб-додатків, який часто стає причиною затримок завантаження. Оптимізація зображень, відео та шрифтів дозволяє зменшити час завантаження сторінки.
2. Серверна частина (бекенд):

- Сервери: обробляють запити користувачів, зберігають та обробляють дані, передають сторінки веб-додатків до клієнтів.
- Бази даних: зберігають дані веб-застосунку (користувацькі дані, контент, продукти тощо). Оптимізація запитів до бази даних та її структури суттєво впливає на швидкість обробки інформації.
- API та мікросервіси: взаємодія різних сервісів через інтерфейси прикладного програмування (API) або мікросервісну архітектуру. Оптимізація API дозволяє зменшити кількість запитів і час їх обробки.

3. Мережеві компоненти:
    - Кешування: дозволяє зберігати дані в тимчасових сховищах, зменшуючи кількість звернень до серверу або бази даних.
    - CDN (Content Delivery Network): це мережа розподілених серверів, що дозволяє доставляти контент користувачам швидше, використовуючи сервери, які фізично знаходяться ближче до користувачів.
    - Протоколи: оптимізація протоколів передачі даних (HTTP/2, HTTPS, WebSocket) допомагає зменшити затримки і втрати даних під час передачі між сервером і клієнтом.
4. Безпека:
    - Захист від атак: веб-застосунки повинні бути захищені від атак типу XSS, CSRF, DDoS тощо. Це включає впровадження технік аутентифікації та авторизації, а також моніторинг активності для виявлення аномальної поведінки.
    - Шифрування даних: використання SSL/TLS для забезпечення захищеного обміну даними між клієнтами та сервером.
5. Моніторинг та логування:
    - Системи моніторингу відслідковують продуктивність, стабільність і безпеку веб-застосунку в реальному часі. Логи

дозволяють аналізувати роботу застосунку для покращення його продуктивності та швидкого реагування на проблеми.

З основних завдань, які напряму пов’язані з оптимізацією, можна виділити наступні

1. Виявлення вузьких місць:
    - Виявлення і аналіз ділянок коду або архітектури, які спричиняють затримки, надмірне споживання ресурсів чи збої в роботі веб-застосунку.
2. Зменшення часу завантаження:
    - Оптимізація ресурсів (кешування, зменшення ваги зображень, мінімізація JavaScript та CSS) для прискорення завантаження сторінки.
3. Масштабованість та адаптивність:
    - Забезпечення ефективної роботи веб-застосунку під час збільшення кількості користувачів або даних. Важливо, щоб система була здатна до горизонтального і вертикального масштабування.

Для прикладу оптимізції було обрано веб-застосунок для зберігання файлів на віддаленому сервері. Веб-застосунки для зберігання файлів забезпечують інфраструктуру для надійного та безпечного збереження даних користувачів у хмарі. Вони дозволяють користувачам завантажувати, зберігати, організовувати та ділитися файлами через інтернет. Ці системи надають інтерфейс для зручної роботи з файлами і надійні механізми захисту та відновлення даних. Основними принципи, якими потрібно керуватися для розробки застосунків такого типу є:

1. Інтерфейс користувача:
    - Створення зручного інтерфейсу для завантаження, пошуку та управління файлами.
    - Забезпечення підтримки різних форматів файлів і їх перегляду.
2. Безпечне зберігання даних:

- Забезпечення шифрування даних під час зберігання та передачі.
- Управління доступом та захист від несанкціонованого використання файлів.

3. Системи резервного копіювання та відновлення:
   - Налаштування автоматичного резервного копіювання та швидкого відновлення у випадку збою.

3.2 Архітектура системи

Веб застосунок поділяється на дві частини, одна серверна частина, інша користувацька.

У сервісній частині розроблено бізнес логіку застосунку, що дозволяє створювати нові аккаунти, проходити автентифікацію та авторизацію, завантажувати нові файлі, скачувати вже завантажені тощо.

На діаграмі (див. рис. 6) показано фізичну архітектуру системи

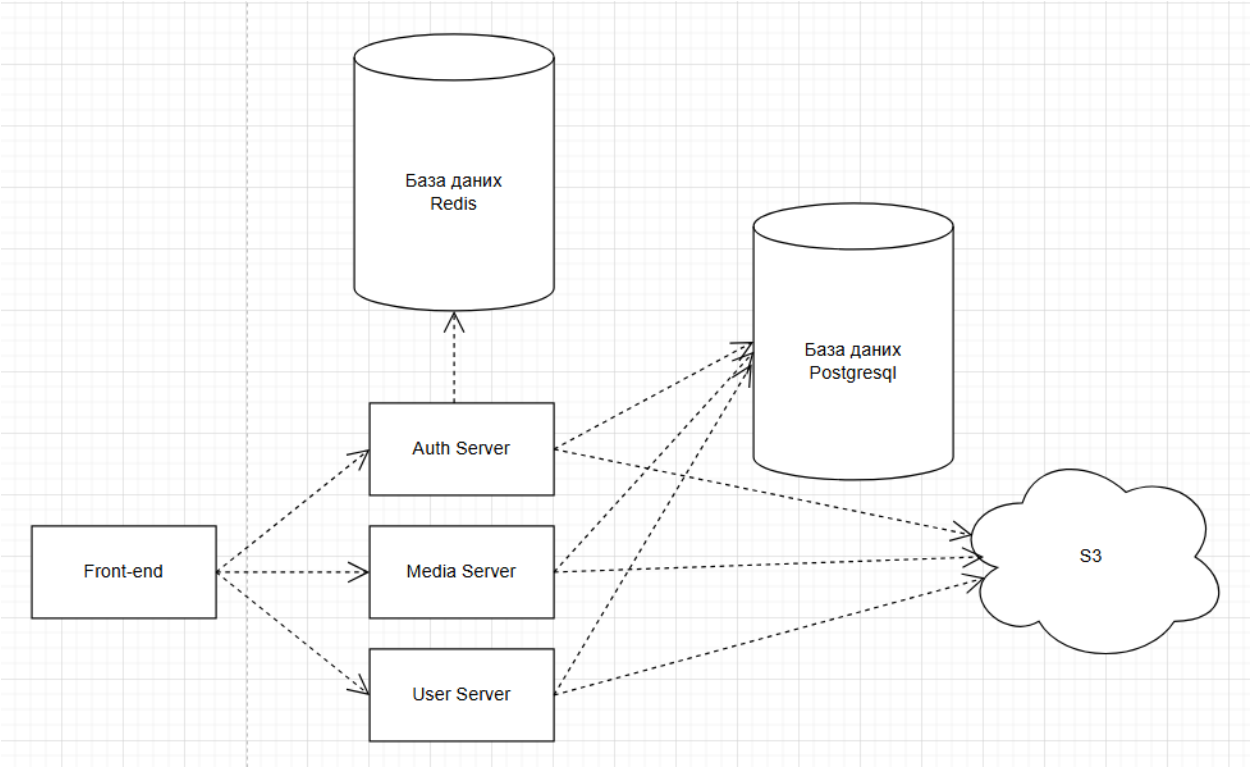


Рисунок 6 — *Діаграма розгортання системи*

3.3 Функціональні вимоги системи

Даний тип веб-застосунку має дві частини, серверну та клієнтську, саме тому потрібно розробити функціональні вимоги, які будуть забезпечувати безперебойну роботу системи, з максимально швидким доступом до файлів й мати гарний захист даних користувачів.

Система повинна забезпечувати:

1. Управління обліковими записами користувачів
    - Реєстрація та аутентифікація користувачів
    - Користувачі повинні мати можливість реєструвати нові облікові записи, використовуючи email
    - Система повинна підтримувати аутентифікацію з використанням пароля
2. Управління профілем користувача:
    - Користувачі повинні мати змогу редагувати свій профіль (ім'я, аватар, контактну інформацію).
    - Можливість скидання пароля через email.
3. Ролі та права доступу:
    - Власники файлів можуть надавати іншим користувачам різні рівні доступу до файлів
4. Користувачі повинні мати можливість завантажувати файли через веб-інтерфейс або API.
5. Перегляд і пошук файлів:
    - Користувачі повинні мати можливість переглядати файли (наприклад, зображення, документи) без необхідності завантаження.
6. Функціонал пошуку файлів за іменем або іншими метаданими.
7. Система повинна підтримувати завантаження файлів різних розмірів, з можливістю обмеження максимального розміру файлу для кожного користувача.

8. Користувачі повинні мати можливість надавати іншим користувачам доступ до файлів
9. Шифрування даних

Веб-застосунок для зберігання файлів повинен забезпечувати зручне та безпечне управління файлами, надавати гнучкі можливості для спільного доступу, інтеграцію з іншими системами через API, а також підтримувати високий рівень безпеки та масштабованості для різних типів користувачів.

3.4 Проєктування інформаційної системи

Під час етапу проектування було проведено структурно-функціональне моделювання, яке дозволило детально проаналізувати бізнес-процеси веб-додатку для збереження файлів, визначити основні напрями покращення його роботи та оптимізації функціоналу. Це дозволило забезпечити прозорість взаємодії між користувачами та системою.

Головною метою проектування стало створення ефективного веб-додатку для збереження файлів, що відповідає поставленим вимогам, враховує задані обмеження та використовує сучасні технології. Для визначення основних можливостей системи було створено діаграми варіантів використання, які описують дії, що можуть виконувати різні типи користувачів (акторів) у межах системи. Ці діаграми дозволяють краще зрозуміти функціональність додатку та забезпечити його адаптацію під реальні сценарії використання.

На діаграмі використання(див рис. 7) зображено функціонал, який користувач має при користування сайтом.



Рисунок 7 — *Діаграма варіантів використання для користувача*

В якості бази даних було використано PostgreSQL, вона є швидкою та зрозумілою для використання. Для створення бази даних було використано Docker, у Docker файлі був прописаний наступний скрипт, зазначений у лістингу 1.

Лістинг *1* — *Створення бази даних*

```
services:
  rovr-db-local:
    build: ./postgres
    restart: always
    volumes:
      - ./postgres/postgres-data:/var/lib/postgresql/data
      - ./postgres/postgres-backups:/root/backups
    ports:
      - "5432:5432"
    container_name: diplom-db-local
    environment:
```

```
POSTGRES_USER: "root"
POSTGRES_PASSWORD: "root"
POSTGRES_DB: "diplom"
```

У базі даних розміщено 6 таблиць, кожна з яких описує певну сутність. Ці таблиці розміщені по певних схемах, відповідно до кожного модуля, який був розроблений в застосунку. Всього було зроблено 3 схеми, серед них media, auth, user (див. рис. 8). Також, серед них є ще спеціальна схема, в яку записано всі міграції та сіди, а саме створення, видалення, редагування, занесення нових даних.

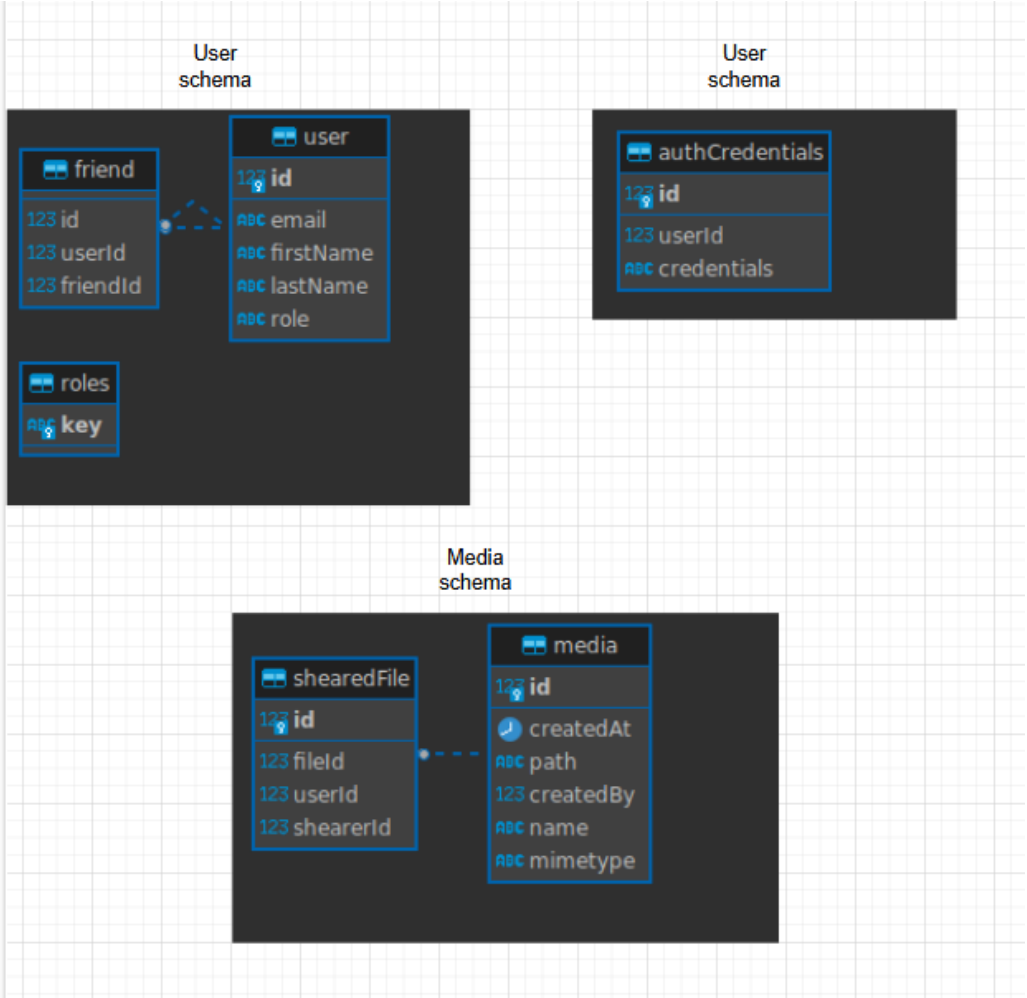


Рисунок 8 — *Схеми бази даних*

Сутність Friend містить інформацію про друзів користувача, який можна передивлятися список файлів, а також скачувати їх з хмарного сховища. Данна сутність містить атрибути, які зазначені в таблиці 1.

Таблиця 1 — *Атрибути сутності «Friend»*

Назва	**Тип**	**Зв'язок**	**Опис**
Id	bigInt	-	Первинний ключ
userId	bigInt	-	ID користувача
friendId	bigInt	-	ID друга

Сутність Role зберігає в собі інформацію про всі ролі, які доступні на сайті. Певна роль буде видана користувачу при реєстрації на сайті. Сутність містить атрибути зображені в таблиці 2.

Таблиця 2 — *Атрибути сутності «Role»*

Назва	**Тип**	**Зв'язок**	**Опис**
key	Varchar(256)	-	Роль користувача

Сутність User містить інформацію про користувача. При реєстрації користувач вводить певні данні, такі як email, firstName, lastName. Дані поля потім записуються в базу даних, а також ще додається role користувача. Всі атрибути сутності зображені у таблиці 3.

Таблиця 3 — *Атрибути сутності «User»*

Назва	**Тип**	**Зв'язок**	**Опис**
Id	bigInt	-	Первинний ключ
email	Varchar(256)	-	email користувача
firstName	Varchar(256)	-	firstName користувача
lastName	Varchar(256)	-	lastName користувача
role	Varchar(256)	-	Role користувача

Сутність Media зберігає ситуацію про файлі, завантажені користувачем на сайті. Для коректного відображення файлів, та їх скачування потрібно мати певну інформацію. До такої інформації можна віднести дату завантаження, ким був створений цей файл, де він зберігається, назва файлу, яку дав користувач, mimetype для коректного скачування та перегляду файлу. Всі ці параметри відображені у таблиці 4.

Таблиця 4 — *Атрибути сутності «Media»*

Назва	**Тип**	**Зв'язок**	**Опис**
Id	bigInt	-	Первинний ключ
createdAt	Date	-	Дата створення файлу
path	Varchar(256)	-	Шляг до файлу на віддаленому сервері
createdBy	bigInt	-	Користувача, який створив файл
name	Varchar(256)	-	Назва файлу, для його відображення
mimetype	Varchar(256)	-	Mimetype файлу

Сутність ShearedFile містить інформацію хто може бачити файли та мати змогу їх скачати. Для цього нам потрібно знати ідентифікатор коритувача, файла а також ідентифікатор користувача, якому хочется надати доступ до файлу.

Таблиця 5 — *Атрибути сутності «ShearedFile»*

Назва	**Тип**	**Зв'язок**	**Опис**
Id	bigInt	-	Первинний ключ
userId	bigInt	-	ID користувача
fileId	bigInt	-	ID файоа
shearerId	bigInt	-	ID користувача, на кого поширен файл

Сутність UserCredential містить інформацію про креденшиали користувача, доступ до яких можна отримати по ідентифікатору користувача.

Таблиця 6 — *Атрибути сутності «UserCredential»*

Назва	**Тип**	**Зв'язок**	**Опис**
Id	bigInt	-	Первинний ключ
userId	bigInt	-	ID користувача
credentials	Varchar(256)	-	Пароль та логін користуваача

3.5 Програмна реалізація застосунку

Спочатку потрібно було створити базу даних. В якості бази даних було обрано PostgreSQL. Для реаліції були створені відповідні файли з міграціями, та якщо треба з сідами, що дозволить швидке розгортання застосунку як для локального так і на віддалених серверах. Щоб міграції і сіди проганялись автоматично було використано flyway – інструмент для міграцій баз даних, який прописаний в докер контейнері, що також в свою чергу дає можливість дуже зручно проганяти міграції локально і на віддалених серверах. Для flyway був написаний наступний конфігураційний файл для його коректної роботи

Лістинг *2 — Конфігураційний файл flyway*

```
flyway:
    outOfOrder: true
    locations: classpath:db/migration
    validate-on-migrate: false
    enabled: true
```

Далі була розроблена стуктура папок для зручного створення нових міграцій, та швидкого створення сидів (див. рис. 9).

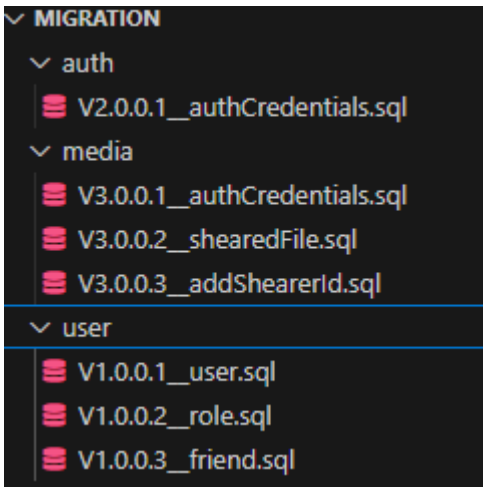


Рисунок 9 — *Схема зберігання сидів та міграцій*

В цих файлах описані всі необхідні міграції та сіди для автоматичного піднімання бази даних на будь-яких присторях. Розгянемо більш детально що саме знаходиться в цих міграціях.

В першому файлі, див. лістинг 3, знаходиться створення таблиці в базі, де будуть зберігатися дані про креденшіали користувача, а саме айді користувача, та його креденшиали, пароль та email, яку він вказував при реєстрації у застосунку.

Лістинг *3* — *Креденшиали користувача*

```
CREATE SCHEMA if not exists auth;
create table auth."authCredentials"
(
    id          bigserial NOT NULL PRIMARY KEY,
    "userId"    bigint not null,
    credentials varchar(255)
);
```

Потім йоде файл з створенням таблиці media, див лістинг 4, де потрібно вказати всі поля, які було визначено при проектувані бази даних.

Лістинг *4* — *Створенням таблиці media*

```
CREATE SCHEMA IF NOT EXISTS "media";
```

```
CREATE TABLE "media"."media" (
  id                bigserial NOT NULL PRIMARY KEY,
  "createdAt"       timestamp NULL,
  "path"            varchar NOT NULL,
  "createdBy"       bigint NOT NULL DEFAULT NULL,
  "name"            varchar NOT NULL,
  "mimetype"        varchar NOT NULL
);
```

Наступним кроком требя зробити міграцію на створення таблиці в яку буде записано списко файлів, які користувач розширив для інших(див. лістинг 5).

Лістинг *5* — *Таблиця shearedFile*

```
CREATE TABLE "media"."shearedFile" (
  id                bigserial NOT NULL PRIMARY KEY,
  "fileId"          bigserial REFERENCES media.media(id),
  "userId"          bigserial NOT NULL
);
```

Потім було додано ще міграцію на створення додаткового поля в таблиці shearedFile для коректного виведення списку та зменшення обсягів втрат даних(див. лістинг 6).

Лістинг *6* — *Оновлення таблиці shearedFile*

```
ALTER TABLE "media"."shearedFile"
ADD COLUMN "shearerId" bigint
```

У таблиці user(див. лістинг 7), що належить до схеми user, зберігається інформація про користувачів. Вона містить унікальний ідентифікатор id, поле для електронної пошти email, а також імена та прізвища користувачів

(firstName і lastName). Додатково кожному користувачу може бути призначена роль, яка зберігається в полі role.

Лістинг 7 — *Таблиця user*

```
CREATE SCHEMA IF NOT EXISTS "user";
CREATE TABLE "user"."user" (
  id          bigserial NOT NULL PRIMARY KEY,
  email       character varying(255),
  "firstName" character varying(255),
  "lastName"  character varying(255),
  "role"      character varying(255)
);
```

Для визначення можливих ролей користувачів створюється таблиця roles(див. лістинг 8). Вона включає єдине поле key, яке є унікальним ідентифікатором ролі. У цю таблицю додається початкове значення – роль admin, що використовується для позначення адміністративних прав(див. лістинг 9).

Лістинг *8* — *Таблиця з усіма ролями*

```
CREATE TABLE "user"."roles"
(
    key         character varying(32) NOT NULL PRIMARY KEY
);
```

Лістинг *9* — *Сіди ролей*

```
INSERT INTO "user"."roles" (key)
VALUES ('admin');
```

Таблиця friend(див. лістинг 10) дозволяє відстежувати зв’язки між користувачами. Вона містить унікальний ідентифікатор запису id, а також два поля: userId і friendId, які вказують на користувача та його друга відповідно.

Ця структура забезпечує базову функціональність для управління соціальними зв'язками в системі, дозволяючи зберігати інформацію про дружбу між користувачами.

Лістинг *10* — *Таблиця друзів користувача*

```
CREATE TABLE "user"."friend"
(
    id              bigserial NOT NULL,
    "userId"          bigserial NOT NULL,
    "friendId"        bigserial NOT NULL
);
```

При запуску докер-композ файлу, після того, як підніметься бази даних, як PostgreSQL та REDIS, автоматично flyway спробує виконати всі міграції, якщо буде помилка – всі міграції будуть відкочені і показана помилка. Якщо була створена нова міграція – треба вручну прогнати скріпт запуску flyway контейнеру, якщо нова міграція буде написана без помилок – в базу запишуться нові данні, старі не будуть змінені.

Для роботи з токена користувача було обрано REDIS, в якості кешованої бази даних, яка дає максимально швидкий доступ до даних, які вона зберігає безпосередньо в оперативній пам’яті комп’ютера, що в свою чергу надає максимально швидкий доступ до них. Для роботи з REDIS потрібно було встановити npm залежність redis, далі для більш зручнішої роботи я створив конфігураційний файл з певними параметрами, які потім використовуються для підключення до бази даних. Потім для більше зручної з цією базою було створено клас, який відповідає бізнес логіці застосунку, а саме створити, оновити, видалити отримати певну інформацію по ключу, REDIS зберігає інформацію в таблицях по принципу ключ знаяення.

Лістинг *11* — *Клас для роботи з Redis*

```
export class RedisClient {
```

```
    private client: RedisClientType

    public async init(config: IRedisConfig) {
        console.log('Connecting to redis...');
        let connUrl: string;
        if (Config.Boolean('REDIS_SECURE_CONNECTION') === true)
{
            connUrl = `rediss://${config.host}:${config.port}`
        } else {
            connUrl = `redis://${config.host}:${config.port}`
        }
        this.client = createClient({
            url: connUrl,
            username: config.username,
            password: config.password,
            name: config.name,
            database: config.database,
        });

        this.client.on("error", (error) => {
            console.log(error)
            throw error;
        });
        await this.client.connect();
    }

    public async set(key: string, value: string): Promise<void>
{
        await this.client.set(key, value, {
            EX: Config.Number('SESSION_TOKEN_TTL'),
        });
    }

    public async setRefreshToken(key: string, value: string):
Promise<void> {
        await this.client.set(key, value, {
```

```
            EX: Config.Number('REFRESH_TOKEN_TTL'),
        });
    }

    public async setForgotPasswordToken(key: string, value:
string): Promise<void> {
        await this.client.set(key, value, {
            EX: Config.Number('FORGOT_PASSWORD_TOKEN_TTL'),
        });
    }

    async push(key:string, value:string): Promise<void> {
        await this.client.rPush(key, value);
    }

    async getList(key:string): Promise<string[]> {
        return this.client.lRange(key, 0, -1);
    }

    async deleteListElement(key: string, userId: string, list:
string): Promise<void> {
        await this.client.lRem(`${list}:${userId}`, 0, key);
    }

    async get(key: string): Promise<string> {
        return await this.client.get(key) as string;
    }

    async delete(key: string): Promise<void> {
        await this.client.del(key);
    }

    public async close() {
        await this.client.disconnect();
    }
```

```
    public async deleteAllUsersTokens(): Promise<void> {
        const keys = await this.client.keys('*');

        const tableKeys = keys.filter((key) => {
            return key.startsWith('refreshTokens') ||
key.startsWith('tokens');
        });

        for (const key of tableKeys) {
            await this.client.del(key);
        }
    }
}
```

Цей код реалізує клас RedisClient, який створює обгортку для роботи з Redis — сервером кешування та зберігання даних. Він дозволяє зручно працювати з Redis, зокрема підключатися до сервера, зберігати, отримувати та видаляти дані, а також управляти списками.

Основна частина роботи починається з методу init, який встановлює підключення до Redis. Залежно від конфігурації, клієнт може використовувати захищене з'єднання (rediss://) або стандартне (redis://). У разі виникнення помилок під час підключення, вони логуються в консоль і викликають виключення.

Клас пропонує кілька методів для роботи з Redis:

- Методи set, setRefreshToken, та setForgotPasswordToken дозволяють зберігати дані з різними строками життя (TTL), які задаються через конфігурацію.
- Метод get використовується для отримання значення за ключем.
- Метод delete видаляє ключ із бази даних Redis.

Для роботи зі списками є методи push, getList, та deleteListElement. Вони додають значення до списку, отримують всі елементи списку або видаляють певний елемент.

Метод deleteAllUsersTokens використовується для видалення всіх токенів користувачів, таких як refreshTokens або tokens. Це здійснюється шляхом пошуку всіх ключів у Redis і фільтрації тих, що починаються з потрібних префіксів.

Для завершення роботи з Redis використовується метод close, який закриває з'єднання.

Загалом, цей клас спрощує роботу з Redis, дозволяючи використовувати уніфіковані методи для часто повторюваних операцій, таких як зберігання токенів чи управління списками.

Потім було реалізовано роботу з токенами користувача а саме, зберігання поточної сейсії користувача(можливо зайти з одного аккаунту на декількох пристроях одночасно)(див. лістинг 12).

Лістинг *12 — Робота з токена користувача*

```
export class TokenStorage implements ITokenStorage {
    constructor(
        private readonly redisClient: RedisClient
    ) {
    }

    public async saveSession(accessToken: string, refreshToken:
string, session: Session ): Promise<void> {
        await this.redisClient.set(`auth:${accessToken}`,
JSON.stringify(session));
        if(refreshToken) {
            await
this.redisClient.setRefreshToken(`refreshToken:${refreshToken}`,
`${session.userId}:${accessToken}`);
            await
this.redisClient.push(`refreshTokens:${session.userId}`,
refreshToken);
        }
        await this.redisClient.push(`tokens:${session.userId}`,
accessToken);
    };

    public async getSession(token: string): Promise<Session> {
        const sessionString = await
this.redisClient.get(`auth:${token}`);
        return JSON.parse(sessionString);
    };
```

```
    public async deleteSession(userId:number):Promise<void> {
        let userAccessTokens = await
this.redisClient.getList(`tokens:${userId}`);
        for (let i = 0; i < userAccessTokens.length; i++) {
            await
this.redisClient.delete(`auth:${userAccessTokens[i]}`);
        }
        let userRefreshTokens = await
this.redisClient.getList(`refreshTokens:${userId}`);
        for (let i = 0; i < userRefreshTokens.length; i++) {
            await
this.redisClient.delete(`refreshToken:${userRefreshTokens[i]}`);
        }
        await this.redisClient.delete(`tokens:${userId}`);
        await
this.redisClient.delete(`refreshTokens:${userId}`);
    }
}
```

Клас TokenStorage відповідає за управління токенами доступу (access tokens) та оновлення (refresh tokens), а також сесіями користувачів у Redis. Він використовує клієнт RedisClient для виконання всіх операцій із Redis.

Метод saveSession зберігає сесію користувача та пов’язані токени. Сесія зберігається в Redis як JSON-об'єкт за ключем типу auth:<accessToken>. Якщо передано refresh-токен, він також зберігається за ключем refreshToken:<refreshToken> разом із інформацією про ID користувача та access-токен. Крім того, refresh-токен додається до списку токенів оновлення користувача, а access-токен додається до списку токенів доступу. Це забезпечує можливість швидко отримувати всі токени, пов’язані з конкретним користувачем.

Метод getSession дозволяє отримати сесію користувача за переданим access-токеном. Він шукає сесію за ключем auth:<token> і повертає її, перетворивши з формату JSON.

Метод deleteSession видаляє всі сесії та токени, пов’язані з конкретним користувачем. Спочатку отримуються всі access-токени користувача і видаляються їхні відповідні сесії. Потім видаляються всі refresh-токени цього користувача. Після цього очищаються списки токенів для користувача: як access-токенів, так і refresh-токенів.

Клас спрощує управління сесіями та токенами, забезпечуючи чітку структуру та прості методи для виконання поширених операцій.

При розробці сервісу для зберігання файлів на віддаленому пристрої, дуже важлива робота з файлами а саме завантаження, отримання файлів, можливість подивитися фотографії, або pdf документи, без можливості завантаження.

Розглянемо домен роботи з файлами, зображений на лістингу 13.

Лістинг *13 — Контролер домена media*

```
export class TokenStorage implements ITokenStorage {
    constructor(
        private readonly redisClient: RedisClient
    ) {
    }

    public async saveSession(accessToken: string, refreshToken:
string, session: Session ): Promise<void> {
        await this.redisClient.set(`auth:${accessToken}`,
JSON.stringify(session));
        if(refreshToken) {
            await
this.redisClient.setRefreshToken(`refreshToken:${refreshToken}`,
`${session.userId}:${accessToken}`);
            await
this.redisClient.push(`refreshTokens:${session.userId}`,
refreshToken);
        }
        await this.redisClient.push(`tokens:${session.userId}`,
accessToken);
    };

    public async getSession(token: string): Promise<Session> {
        const sessionString = await
this.redisClient.get(`auth:${token}`);
        return JSON.parse(sessionString);
    };

    public async deleteSession(userId:number):Promise<void> {
        let userAccessTokens = await
this.redisClient.getList(`tokens:${userId}`);
        for (let i = 0; i < userAccessTokens.length; i++) {
            await
this.redisClient.delete(`auth:${userAccessTokens[i]}`);
        }
        let userRefreshTokens = await
this.redisClient.getList(`refreshTokens:${userId}`);
```

```
        for (let i = 0; i < userRefreshTokens.length; i++) {
            await
this.redisClient.delete(`refreshToken:${userRefreshTokens[i]}`);
        }
        await this.redisClient.delete(`tokens:${userId}`);
        await
this.redisClient.delete(`refreshTokens:${userId}`);
    }
}
```

Цей код реалізує маршрути для роботи з медіафайлами у Fastify-сервері. Він забезпечує функціональність завантаження, перегляду, скачування, обміну файлами та отримання інформації про файли. Для роботи використовується контекст-залежна система залежностей, яка дозволяє отримувати потрібні сервіси через ContextResolver.

У маршруті /upload реалізується завантаження медіафайлів. Після завантаження файлу метод повертає інформацію про нього, включаючи посилання для завантаження та перегляду.

Маршрут /my-files надає список всіх файлів користувача з можливістю пагінації. Файли повертаються у вигляді об'єктів з основними атрибутами (ID, ім'я, посилання).

Маршрути /download/:documentId та /view/:documentId дозволяють завантажити або переглянути файл відповідно. Для цього перевіряється, чи належить файл користувачеві, після чого отримується стрім файлу з файлової системи або хмарного сховища.

Маршрут /share відповідає за обмін файлами з іншими користувачами. Користувач може поділитися файлом, вказавши email одержувача. У відповідь повертається інформація про спільний доступ, включаючи посилання для перегляду та завантаження.

Маршрут /sheared повертає список файлів, якими користувач поділився або які поділилися з ним.

Маршрут /file-by-name дозволяє отримати інформацію про файл за його іменем. Пошук виконується в межах файлів, що належать користувачеві.

Для захисту всіх маршрутів використовується middleware Auth, який перевіряє авторизацію за допомогою токенів. Кожен маршрут також має схему з тегами та описами, для генерації Swagger документації.

Далі слід розглянкути клас, який відповідає за завантаження документів у віддалене сховище(див. лістинг 14).

Лістинг *14 — Сервіс завантаження документів*

```
export class MediaUploadService {
    constructor(
        private readonly remoteStorage: IRemoteStorage,
        private readonly mediaRepo: IMediaRepo
    ) {
    }

    public async upload(request: FastifyRequest):
Promise<MediaEntity> {

        const { pipeline } = require('stream');
        const pump = util.promisify(pipeline);
        const filename = randomMd5();

        const data = await request.file();
        if (!data) {
            throw new UploadFileException();
        }

        const writeStream =
fs.createWriteStream(uploadsFolderPath + filename);
        await pump(data.file, writeStream);

        writeStream.close();

        await this.remoteStorage.upload(
            uploadsFolderPath + filename,
```

```
            request.session.userId + "/" + filename,
        );

        fs.unlinkSync(uploadsFolderPath + filename);

        const media = new MediaEntity();
        media.createdAt = new Date();
        media.path = filename;
        media.createdBy = request.session.userId;
        media.name = data.filename;

        const mimetype = mime.getType(data.filename);
        if(mimetype) {
            media.mimetype = mimetype
        }

        return await this.mediaRepo.create(media);
    }
}
```

Цей код реалізує сервіс для завантаження медіафайлів під назвою MediaUploadService. Він відповідає за обробку файлу, завантаженого через HTTP-запит, його збереження в локальну файлову систему, перенесення в віддалене сховище, а також запис інформації про файл у базу даних.

Ключовий метод сервісу — upload. Він приймає об'єкт FastifyRequest, який містить файл та дані сесії користувача. Спочатку перевіряється, чи переданий файл. Якщо його немає, викликається виключення UploadFileException.

Файл тимчасово записується у локальну файлову систему в директорію uploadsFolderPath з випадковим ім’ям, яке генерується функцією randomMd5. Для обробки стрімів використовується модуль stream та утиліта pump із промісифікацією.

Після запису у файлову систему файл завантажується у віддалене сховище через метод upload сервісу remoteStorage. Віддалене сховище структурує файл у форматі userId/filename, де userId береться з сесії користувача.

Після успішного завантаження у віддалене сховище тимчасовий файл видаляється з локальної файлової системи.

Створюється об'єкт типу MediaEntity, який зберігає метаінформацію про файл: дата створення, шлях, ID користувача, ім'я файлу та його MIME-тип (який визначається через модуль mime). Цей об'єкт передається у репозиторій mediaRepo для збереження в базі даних.

У результаті метод повертає збережений у базі даних об'єкт файлу (MediaEntity), що містить усі необхідні дані для подальшої роботи з ним.

Для роботи з файлами була розроблена модель бази даних MediaModel, яка реалізована за допомогою ORM Sequelize. Ця модель описує структуру таблиці media в базі даних і містить основну інформацію про файли(див. лістинг 15).

Лістинг *15 — Модель бази даних MediaModel*

```
import {DataTypes, Model, Sequelize} from "sequelize";

export class MediaModel extends Model {
    declare id: number;
    declare createdAt: Date;
    declare path: string;
    declare createdBy: number;
    declare name: string;
    declare mimetype: string;
}

export function registryMediaModel(database: Sequelize) {
    MediaModel.init({
        id: {
```

```
            type: DataTypes.BIGINT,
            primaryKey: true,
            autoIncrement: true,
        },
        createdAt: {
            type: DataTypes.DATE,
        },
        path: {
            type: DataTypes.STRING,
        },
        createdBy: {
            type: DataTypes.BIGINT,
        },
        name: {
            type: DataTypes.STRING,
        },
        mimetype: {
            type: DataTypes.STRING,
        },
    }, {
        sequelize: database,
        schema: 'media',
        modelName: 'media'
    });

}
```

Кожен запис у таблиці має унікальний ідентифікатор id, який автоматично збільшується для кожного нового запису. Поле createdAt зберігає дату створення файлу, що допомагає відстежувати, коли саме він був доданий.

Поле path містить шлях до файлу в віддаленому сховищі, а createdBy зберігає ідентифікатор користувача, який завантажив файл. Поле name зберігає оригінальне ім'я файлу, яке було вказане при завантаженні, а поле mimetype визначає тип файлу (наприклад, image/png або application/pdf).

Модель реєструється в базі даних у схемі media, що забезпечує її ізольованість у просторі імен. Вона автоматично підключається до екземпляра бази даних, переданого через параметр sequelize, і отримує ім'я media.

Ця модель забезпечує централізоване зберігання інформації про файли, їхні атрибути та зв'язок із користувачами, що робить її ключовою частиною інфраструктури роботи з медіафайлами.

Для роботи з моделею треба розробити клас, який буде реалізовувати певні операції, які визначені бізнес логікою застосунку, такі класи називаються репозиторіями. Клас MediaRepoPgSql реалізує інтерфейс IMediaRepo і слугує репозиторієм для роботи з таблицею media у базі даних PostgreSQL. Він забезпечує основні CRUD-операції та додаткові методи для пошуку файлів (див. лістинг 16).

Лістинг *16 — Репозиторій MediaRepoPgSql*

```
export class MediaRepoPgSql implements IMediaRepo {
    constructor(
        private readonly mediaDbMapper: MediaDbMapper
    ) {
    }

    public async create(media: MediaEntity):
Promise<MediaEntity> {
        const mediaModel =
this.mediaDbMapper.mapFromEntityToModel(media);
        await mediaModel.save();

        return
this.mediaDbMapper.mapFromModelToEntity(mediaModel);
    }

    public async findById(id: number): Promise<MediaEntity |
null> {
```

```
        const mediaModel = await MediaModel.findOne({where:{id:
id}});
        if (!mediaModel) {
            return null;
        }

        return
this.mediaDbMapper.mapFromModelToEntity(mediaModel);
    }

    public async getMediaByCreatedByAndId(mainId: number,
mediaId: number): Promise<MediaEntity | null> {
        const mediaModel = await MediaModel.findOne({where:
{id:mediaId, createdBy:mainId}})
        if (!mediaModel) {
            return null;
        }
        return
this.mediaDbMapper.mapFromModelToEntity(mediaModel);
    }

    public async getAllUserMedia(userId: number, limit: number,
offset: number): Promise<{
        media: MediaEntity[],
        count: number
    }>{
        const { rows, count } = await
MediaModel.findAndCountAll({
            offset,
            limit,
            where: {createdBy: userId},
            order: [['createdAt', 'DESC']],
        });

        return {
```

```
            media: rows.map( model =>
this.mediaDbMapper.mapFromModelToEntity(model)),
            count
        };
    }

    public async getMediaByName(userId: number, fileName:
string): Promise<MediaEntity | null> {
        const mediaModel = await MediaModel.findOne({
            where: {
                createdBy: userId,
                name: {
                    [Op.like]: `%${fileName}%`,
                },
            },
        });

        if (!mediaModel) {
            return null;
        }

        return
this.mediaDbMapper.mapFromModelToEntity(mediaModel);
    }
}
```

Для зручності роботи з даними використовується mediaDbMapper, який перетворює дані між форматами сутності (MediaEntity) і моделі бази даних (MediaModel).

Метод create дозволяє створити новий запис у базі даних. Об'єкт типу MediaEntity передається в метод, перетворюється у модель бази даних через маппер, зберігається у таблиці, а потім знову конвертується в сутність для повернення.

Метод findById шукає файл за його унікальним ідентифікатором. Якщо запис знайдено, він конвертується в сутність і повертається, якщо ні — повертається null.

Метод getMediaByCreatedByAndId дозволяє знайти файл, створений конкретним користувачем, за його ID. Це корисно для обмеження доступу до файлів лише їх власниками.

Метод getAllUserMedia надає можливість отримати список усіх файлів, завантажених певним користувачем. Він підтримує пагінацію через параметри limit і offset. У результаті повертається об'єкт, що містить список медіафайлів та загальну кількість записів.

Метод getMediaByName виконує пошук файлу за його ім'ям, використовуючи умову LIKE для пошуку часткових збігів. Це дозволяє знаходити файли навіть при неповному введенні імені.

Цей клас слугує абстракцією над прямою взаємодією з базою даних, спрощуючи доступ до даних і забезпечуючи логічну ізоляцію між базою даних і бізнес-логікою програми.

3.6 Розробка інтерфейсу інформаційної системи

Для коректної роботи додатку потрібно було розробити фронтенд частину застосунку. Для початку потрібно розробити систему, яка дозволить користувачу логінитися, або створити новий акаунт, якщо його ще не було створено на сайті. Також треба було зробити захист від того, що користувая буде вводити не вірні далі в поля(наприклад в поле email, напише невалідний імейл, або логін – треба провалідувати і не давати можливість провести реєстрацію). Для написання було обрано фреймворк React, для більш швидкої, надійної, а також більш оптимізованої розробки. Для логіки реєстрації було розроблено компонент Register. Він використовує бібліотеку Formik для управління формами, а також інтегрує валідацію через кастомний хук useFormValidation(див. лістинг 17).

Лістинг *17 — Компонент Register*

```
const Register: FC = () => {
    const navigate = useNavigate();

    const { initialValues, validationSchema, onSubmit } =
useFormValidation(true, async (values) => {
        try {
            const { firstName, lastName, email, password } =
values;
            const data = await register(firstName!, lastName!,
email, password);
            if (data) setTokens(data)
            navigate('/');
        } catch (err) {
            toast.error('Registration failed');
            console.log('Error:', err);
        }
    });

    return (
        <AuthWrapper title="Register">
            <Formik initialValues={initialValues}
validationSchema={validationSchema} onSubmit={onSubmit}>
                {({ isSubmitting }) => (
                    <Form>
                        <AuthField name="firstName" label="First
Name" placeholder="Enter your first name" />
                        <AuthField name="lastName" label="Last
Name" placeholder="Enter your last name" />
                        <AuthField name="email" label="Email"
placeholder="Enter your email" />
                        <AuthField name="password"
label="Password" type="password" placeholder="Enter your
password" />
```

```
                        <AuthField name="confirmPassword"
label="Confirm Password" type="password" placeholder="Confirm
your password" />

                        <button
                            type="submit"
                            disabled={isSubmitting}
                            className={`w-full bg-blue-600 text-
white py-2 mt-5 rounded hover:bg-blue-700 transition ${
                                isSubmitting ? 'opacity-50
cursor-not-allowed' : ''
                            }`}
                        >
                            {isSubmitting ? 'Registering...' :
'Register'}
                        </button>
                    </Form>
                )}
            </Formik>
        </AuthWrapper>
    );
};
```

У функціональному компоненті оголошується функція onSubmit, яка виконується під час відправки форми. Вона отримує дані з форми, викликає асинхронну функцію register, яка відповідає за реєстрацію користувача, та обробляє відповідь сервера. Якщо реєстрація успішна, дані з сервера зберігаються (наприклад, токени автентифікації), і користувача перенаправляють на головну сторінку. У разі помилки виводиться повідомлення про невдалу реєстрацію.

Компонент використовує Formik для побудови форми з полями: ім'я, прізвище, email, пароль і підтвердження пароля. Поля реалізовані через

кастомний компонент AuthField, який спрощує створення інтерфейсу для введення даних.

Кнопка "Register" змінює свій стан залежно від процесу реєстрації. Під час відправки форми вона стає неактивною, показуючи текст "Registering...". Зовнішній вигляд компонента обгорнутий у компонент AuthWrapper, який, імовірно, додає загальне оформлення для сторінки авторизації або реєстрації.

Загалом код реалізує зручну форму реєстрації з асинхронною обробкою даних, інтеграцією валідації та базовими повідомленнями про помилки(див рис. 10).

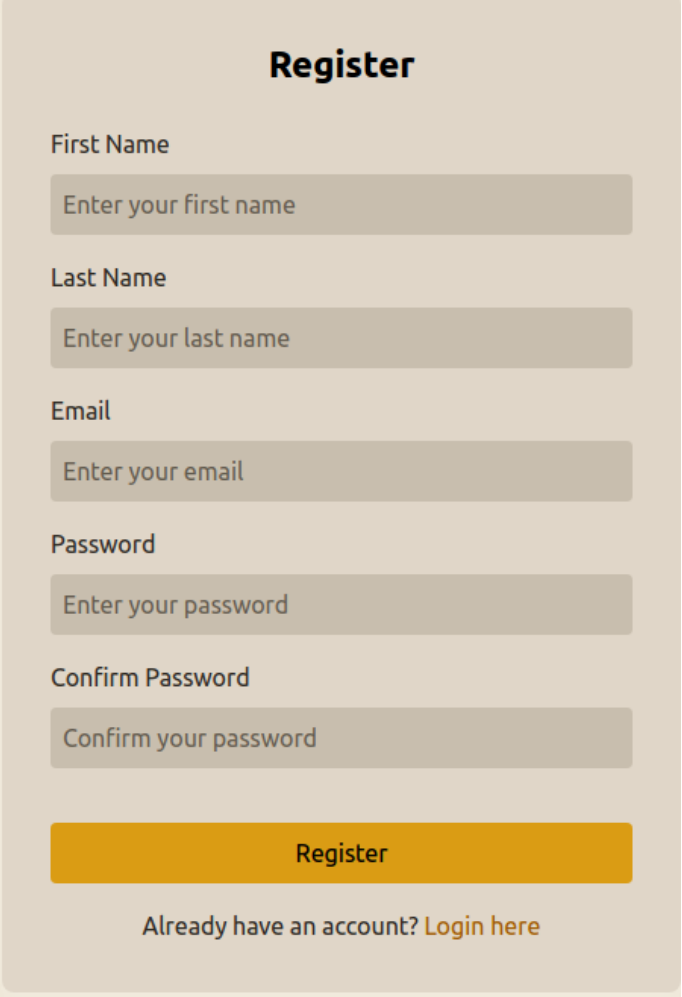


Рисунок 10 — *Екран реєстрації користувача*

Також треба зробити можливість логіну користувача на вже існуючий аккаунт з видачею access та refresh токенів для подальшого допуску

користувача виконувати певні чи інші дії, а також надати можливість користувачу не заходити на аккаунт кожен раз, коли користувач закриває браузер. Також при логіну з іншого пристрою, сервіс буде розуміти що користувач увійшов з іншого пристрою і збереже вже існючі сервіси. Було розроблено такий інтерфейс для логіну користувача(див. рис. 11)

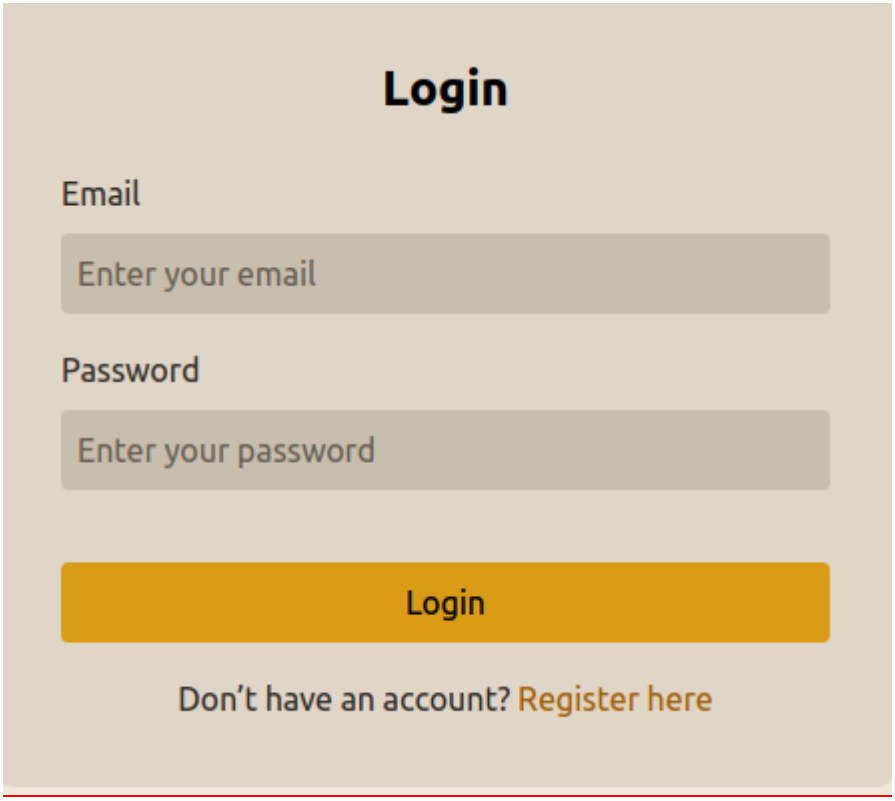


Рисунок 11 — *Логін користувача*

Для реалізації логіну потрібно розробити відповідну компоненту Login.

Лістинг *18* — *Компонент Login*

```
const Login: FC = () => {

    const navigate = useNavigate();

    const { initialValues, validationSchema, onSubmit } =
useFormValidation(false, async (values) => {
        try {
            const { email, password } = values;
```

```
            const data = await login(email, password);
            if (data) setTokens(data)
            navigate('/');
        } catch (err) {
            toast.error('Login failed');
            console.log('Error:', err);
        }
    });

    return (
        <AuthWrapper title="Login">
            <Formik initialValues={initialValues}
validationSchema={validationSchema} onSubmit={onSubmit}>
                {({ isSubmitting }) => (
                    <Form>
                        <AuthField name="email" label="Email"
placeholder="Enter your email" />
                        <AuthField name="password"
label="Password" type="password" placeholder="Enter your
password" />
                        <button
                            type="submit"
                            disabled={isSubmitting}
                            className={`w-full bg-blue-600 text-
white py-2 mt-5 rounded hover:bg-blue-700 transition ${
                                isSubmitting ? 'opacity-50
cursor-not-allowed' : ''
                            }`}
                        >
                            {isSubmitting ? 'Logging in...' :
'Login'}
                        </button>
                    </Form>
                )}
            </Formik>
        </AuthWrapper>
```

```
    );
};
```

Цей компонент Login у React реалізує функціонал авторизації користувача. Використовується бібліотека Formik для зручного управління станом форми, а також кастомний хук useFormValidation для валідації введених даних.

У функції компонента визначено обробник onSubmit, який спрацьовує при відправці форми. Він виконує асинхронний запит до функції login, передаючи введені email і пароль. У разі успіху отримані дані (ймовірно, токени) зберігаються через setTokens, і користувач перенаправляється на головну сторінку. Якщо запит завершується помилкою, відображається повідомлення про невдалий вхід, а також помилка логікується в консоль.

Форма складається з двох полів: email і password, реалізованих через компонент AuthField. Кнопка для входу має стан, що змінюється залежно від того, чи відбувається зараз обробка запиту — якщо так, вона стає неактивною, і текст змінюється на "Logging in...".

Весь інтерфейс розташований у компоненті AuthWrapper, який задає загальну структуру сторінки авторизації.

Код організований зручно і легко масштабується. Він підтримує чітку логіку обробки помилок, інтеграцію валідації та простий у розширенні дизайн форми.

Після логіну або реєстрації користувача його переведе на головний екран застосунку, який буде виводити всі завантажені файли, має кнопку заванатажити ще файли, при скролі вниз, застосунок автоматично буде підвантажувати нові файли з віддаленого сховища, які ще були не завантажені. Головна сторінка застосунку має такий вигляд, як зображено на рисунку 12. Якщо користувач захоче вийти з застосунку – йому потрібно натиснути кнопку Logout, яка в свою чергу зробить редирект на сторінку логіну, а також очистить токени користувача, які знаходять в локальному сховищі. При

натисканні на кнопку Upload file – користувачу буде доступна можливість обрати файл з своєї системи й завантажити його на сервіс.

.

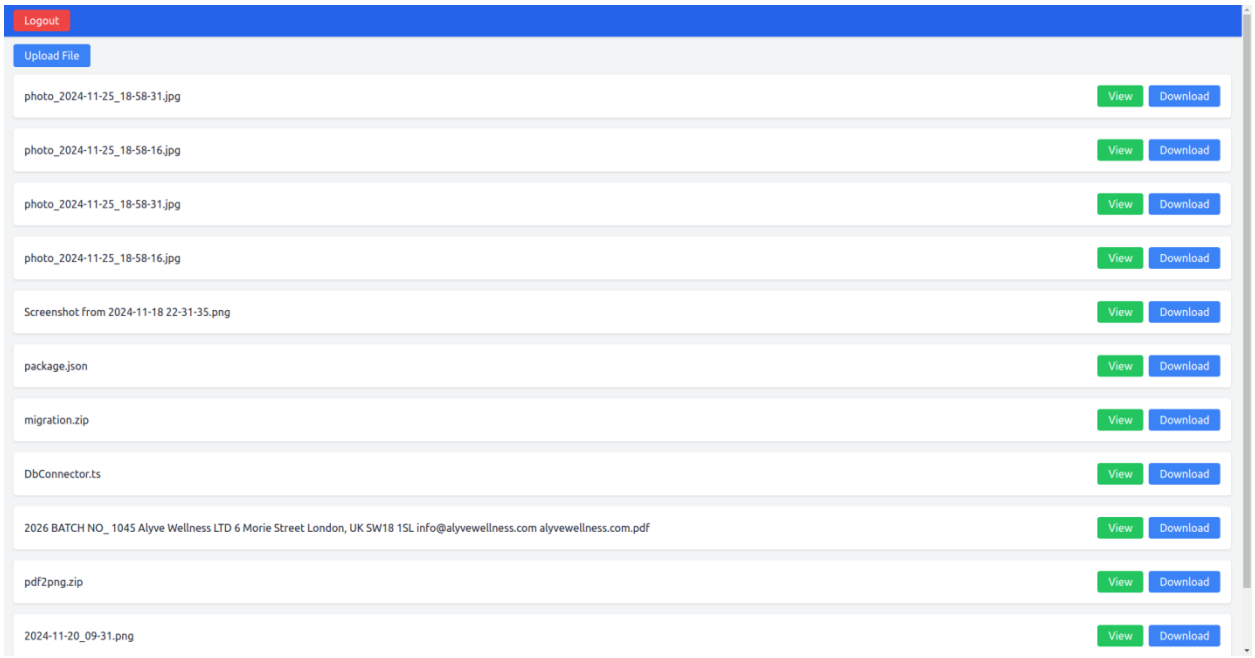


Рисунок 12 — *Головний екран застосунку*

За цю сторінку відповідає компонента Home(див. лістинг 19). Основна логіка кнопки виходу (Logout) визначена у функції handleLogout, яка очищає токени через clearTokens і перенаправляє користувача на сторінку входу за допомогою useNavigate.

Інтерфейс сторінки складається з двох основних частин:

- Шапка (Header) – приймає ім'я користувача і функцію для виходу. Це дозволяє легко адаптувати шапку під різних користувачів.
- Контент (FileList) – компонент, що відповідає за відображення списку файлів. Він обгорнутий у відступи, щоб забезпечити естетичність макета.

Для оформлення використовується клас min-h-screen bg-gray-100, що забезпечує мінімальну висоту сторінки на весь екран і задає світло-сірий фон.

Лістинг *19* — *Компонент Home*

```
const Home = () => {
```

```
    const [username] = useState<string>('User');
    const navigate = useNavigate();
    const handleLogout = () => {
        clearTokens();
        navigate('/login');
    };
    return (
        <div className="min-h-screen bg-gray-100">
            <Header username={username} onLogout={handleLogout}
/>
            <div className="p-4">
                <FileList />
            </div>
        </div>
    );
};
export default Home;
```

3.7 Висновки до розділу 3

1. Проведено аналіз предметної області веб-застосунків для зберігання файлів, визначено основні компоненти системи, включаючи клієнтську, серверну частини, мережеві складові, а також засоби забезпечення безпеки та моніторингу. Це дало змогу чітко визначити вимоги до архітектури та функціональності системи.
2. Спроєктовано архітектуру системи, яка складається з серверної частини для обробки запитів, управління даними, а також клієнтської частини для інтерактивної взаємодії користувача із системою. Для забезпечення масштабованості та відмовостійкості системи використано сучасні підходи до зберігання даних і взаємодії компонентів.
3. Розроблено функціональні вимоги, що забезпечують безперебійну роботу системи, включаючи управління обліковими записами

користувачів, захист даних, обробку файлів різного формату та розміру, а також підтримку спільного доступу до файлів.

4. Реалізовано інформаційну систему з використанням PostgreSQL як основної бази даних, Docker для зручного розгортання, а також Redis для швидкого зберігання токенів автентифікації. Проведено міграції для створення структур бази даних, що забезпечують гнучкість і легкість у використанні.
5. Розроблено серверну частину із застосуванням Fastify, що забезпечує високу продуктивність та надійність обробки запитів. Реалізовано бізнес-логіку управління файлами, сесіями користувачів, а також безпечного зберігання файлів на віддаленому сервері.
6. Розроблено клієнтську частину з використанням React, Vite та Tailwind CSS, що забезпечило швидкість розробки, гнучкість дизайну та високу продуктивність. Реалізовано компоненти для реєстрації, входу, перегляду списку файлів, їх завантаження та управління доступом до них.
7. Впроваджено механізми валідації даних на клієнтському рівні з використанням бібліотек Formik та кастомних хуків, що забезпечує зручність для користувачів і зменшує ризик помилок під час введення даних.
8. Система розроблена з урахуванням сучасних вимог до безпеки, включаючи використання шифрування, захист від атак, обмеження доступу та резервне копіювання даних, що гарантує збереження інформації користувачів.

4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ МЕТОДІВ ОПТИМІЗАЦІЇ ТА ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ WEB -ДОДАТКІВ

4.1 Результати дослідження баз даних

У процесі вибору оптимальної бази даних для веб-додатку було проведено детальне тестування продуктивності MongoDB та PostgreSQL. Метою дослідження було визначення, яка з баз даних забезпечує більш високу швидкість виконання операцій вставки та пошуку даних. Для цього були розроблені спеціалізовані програмні модулі, що автоматизували виконання відповідних тестів, фіксували результати, а також дозволяли мінімізувати вплив людського фактора на отримані дані.

Експерименти з вимірювання продуктивності складалися з двох основних етапів: операції вставки даних та операції пошуку. Для вставки використовувалися набори даних різного обсягу: від 10 тисяч до 5 мільйонів записів. Важливо відзначити, що операція вставки реалізовувалася у багатопотоковому режимі з використанням шести потоків, що дозволяло максимально ефективно використовувати апаратні ресурси системи та оцінювати продуктивність бази даних у реальних умовах високого навантаження. Кожна серія тестів виконувалася 30 разів для зниження похибки у вимірюваннях. Після кожного виконання обчислювався середній час, необхідний для завершення операції(див. таблицю 7).

Таблиця 7 — *Порівняння швидкості операцій додавання (Insert)*

Кількість записів	MongoDB (сек)	PostgreSQL (сек)
10000	0.253	0.038
100000	2.208	1.127
500000	11.607	4.491
1000000	24.429	12.640
2000000	49.062	26.426

5000000	149.836	74.120

Під час другого етапу тестування аналізувалася швидкість пошуку даних у тих самих обсягах записів, які використовувалися на етапі вставки. Для обох баз даних було реалізовано однакові алгоритми виконання запитів, щоб забезпечити рівні умови для порівняння. Середній час виконання пошуку також обчислювався на основі 30 повторень кожного тесту(див. таблицю 8)..

Таблиця 8 — *Порівняння швидкості операцій зчитування (Update)*

Кількість записів	MongoDB (сек)	PostgreSQL (сек)
10000	0.159	0.030
100000	2.304	0.523
500000	10.893	4.488
1000000	21.837	14.063
2000000	43.974	25.144
5000000	113.505	65.267

Результати тестування продемонстрували, що PostgreSQL перевершує MongoDB у швидкості виконання обох операцій. У випадку вставки даних PostgreSQL показав більш високу продуктивність, особливо для великих обсягів записів. Навіть за умов використання багатопотоковості, яка повинна була забезпечити оптимальну швидкість роботи MongoDB, останній поступався PostgreSQL у всіх тестових наборах. Аналогічна ситуація спостерігалася і для операцій пошуку даних, де PostgreSQL продемонстрував значно менший середній час виконання запитів у порівнянні з MongoDB.

Ці результати свідчать про те, що PostgreSQL є кращим вибором для систем, які потребують високої продуктивності при роботі з великими обсягами даних. Ефективність PostgreSQL як у додаванні, так і у зчитуванні даних робить його ідеальним кандидатом для використання в сучасних веб-додатках, де критично важливою є швидкість обробки запитів. Отримані дані підтверджують доцільність використання PostgreSQL як базового рішення для

реалізації системи збереження файлів, що гарантує стабільну роботу додатку навіть за умов високого навантаження.

4.2 Результати дослідження використання сидів для розгортання бази даних

Для порівняння продуктивності ORM-фреймворку Sequelize та сирих SQL-запитів було виконано серію експериментів. Суть тестів полягала у виконанні однакового набору операцій сідінгу для бази даних PostgreSQL за допомогою Node.js. Сідінг включав такі операції: створення нових таблиць, встановлення зв’язків між ними, додавання даних до існуючих таблиць, видалення таблиць, видалення даних з таблиць та оновлення структури вже існуючих таблиць.

Для кожного підходу (Sequelize та сирі SQL-запити) було реалізовано окремі сценарії виконання сідінгу. Кожен сценарій включав певну кількість сідів, які запускалися послідовно, а час виконання кожного з них фіксувався. Щоб забезпечити точність вимірювань, кожен тест повторювався 10 разів, і було обчислено середній час виконання.

Результати експерименту показали, що використання сирих SQL-запитів значно пришвидшує виконання операцій сідінгу порівняно з використанням Sequelize. Це зумовлено тим, що ORM додає додатковий рівень абстракції, що спрощує написання коду, але водночас впливає на продуктивність через обробку метаданих, генерацію запитів і перевірки(див. таблицю 9).

Таблиця 9 — *Порівняння швидкості операцій додавання (Insert)*

Кількість сідів	Sequelize (сек)	Сирі SQL (сек)
10	5.2	3.1
20	10.8	6.2
50	26.7	14.8
100	54.3	30.5

200	112.5	62.3

Як видно з таблиці, використання сирих SQL-запитів дозволяє суттєво скоротити час виконання операцій сідінгу, особливо при збільшенні кількості виконуваних сідів. Наприклад, при виконанні 100 сідів сирі SQL-запити показали майже вдвічі кращий результат, ніж Sequelize. Це свідчить про те, що для великих обсягів даних і складних операцій сідінгу доцільніше використовувати сирі SQL-запити, якщо продуктивність має вирішальне значення.

Водночас варто зазначити, що ORM, такі як Sequelize, надають значну перевагу в зручності використання, читабельності та підтримці коду. Таким чином, вибір між цими підходами залежить від конкретних вимог проєкту: якщо ключовим фактором є швидкість, перевагу варто надати сирим SQL-запитам; якщо ж важливі зручність і підтримуваність, доцільно використовувати ORM.

4.3 Результати дослідження використання буферного підхоту та ReadStream для передачі файлів у хмарне сховище

У цьому експерименті було проведено порівняння двох підходів передачі файлів у хмарне сховище (наприклад, AWS S3): передача файлів у вигляді буфера (Buffer) та через потоки читання (ReadStream). Метою дослідження було оцінити час передачі файлів і витрати оперативної пам'яті для кожного з методів. Передача файлів тестувалася на платформі Node.js.

Для кожного методу використовувалися файли різного розміру: 10 МБ, 50 МБ, 100 МБ, 500 МБ та 1 ГБ. Кожна передача виконувалася 10 разів, після чого обчислювався середній час передачі. Крім того, під час кожної передачі проводився моніторинг використання оперативної пам’яті.

Аналіз результатів показав, що передача файлів через ReadStream забезпечує кращу продуктивність у порівнянні з використанням буфера(див.

таблицю 10). У випадку з великими файлами різниця стає ще більш значною: для файлів розміром 1 ГБ передача через ReadStream була майже на 18% швидшою.

Таблиця 10 — *Час передачі файлів*

Розмір файлу	Буфер (сек)	ReadStream (сек)
10 МБ	0.32	0.28
50 МБ	1.56	1.23
100 МБ	3.12	2.47
500 МБ	15.84	12.63
1 ГБ	31.20	25.47

Крім того, метод передачі через ReadStream значно ефективніше використовує оперативну пам’ять. Це пов'язано з тим, що при використанні буфера весь файл спочатку завантажується в пам’ять, що призводить до її великого споживання. У випадку з ReadStream файл обробляється поетапно, що дозволяє значно зменшити витрати пам’яті.

Таблиця 11 — *Використання пам'яті*

Розмір файлу	Буфер(МБ)	ReadStream(МБ)
10 МБ	10	12
50 МБ	50	15
100 МБ	100	18
500 МБ	500	25
1 ГБ	1000	30

Таким чином, для передачі великих файлів у хмарне сховище доцільніше використовувати ReadStream, оскільки цей метод не лише швидший, але й менш ресурсозатратний. Однак для невеликих файлів (до 10 МБ)

використання буфера може бути виправданим через простоту реалізації. Вибір методу залежить від конкретних вимог та умов системи.

4.4 Результати дослідження використання “Lazy loading”

Використання технології “Lazy Loading” для оптимізації веб-додатків є важливим інструментом для підвищення продуктивності системи, особливо у випадках, коли сторінки містять значний обсяг медіафайлів або динамічного контенту. У ході дослідження було проведено експерименти, спрямовані на оцінку ефективності цієї технології в зменшенні часу завантаження сторінок, зменшенні використання ресурсів мережі та покращенні користувацького досвіду.

Для тестування “Lazy Loading” було розроблено набір веб-сторінок із великим обсягом зображень, які зазвичай завантажуються одразу при відкритті сторінки. Було реалізовано два підходи: традиційне завантаження всього контенту при відкритті сторінки та завантаження лише видимого контенту з поступовим підвантаженням невидимого при скролі (lazy loading). Тестування проводилося за допомогою Lighthouse (Google) та інших інструментів моніторингу продуктивності.

Результати показали, що при використанні “Lazy Loading” час початкового завантаження сторінки скоротився в середньому на 40%. Це досягалося за рахунок зменшення обсягу переданих даних у мережі, що також знизило витрати ресурсів серверної частини.

Підсумовуючи, впровадження “Lazy Loading” є ефективним рішенням для веб-додатків із великим обсягом контенту, яке суттєво підвищує продуктивність і забезпечує економію ресурсів. Результати дослідження підтверджують доцільність використання цієї технології в сучасних веб-системах, особливо в проектах із великим навантаженням на клієнтську та серверну частини.

4.5 Результати оптимізації додатку

Результати оптимізації веб-застосунку демонструють значне покращення продуктивності, стабільності та зручності використання системи. Під час впровадження оптимізаційних рішень було вдосконалено як серверну, так і клієнтську частини програми, що дозволило суттєво зменшити час завантаження сторінок, знизити навантаження на сервер і покращити користувацький досвід.

На серверній стороні оптимізація включала використання Redis для кешування, що дозволило зменшити кількість звернень до бази даних PostgreSQL. Оптимізація запитів до бази даних за рахунок нормалізації та зменшення складності запитів скоротила середній час виконання складних операцій на 60%. У таблиці 12 наведено час за який застосунок обробляє певну кількість запитів.

Таблиця 12 — *Час виконання запитів*

Кількість запитів	Час виконання до оптимізації (с)	Час виконання після оптимізації (с)
1000	12.5	6.2
2000	25.3	12.8
5000	62.7	30.5
10000	125.6	60.2

Клієнтська частина була оптимізована за рахунок мінімізації та об'єднання файлів CSS і JavaScript, що зменшило їх обсяг на 35%. Використання технології “Lazy Loading” для завантаження зображень дозволило скоротити початковий час завантаження сторінки на 40% і зменшити обсяг переданих даних у мережі.

Загалом результати оптимізації показали, що впроваджені рішення дозволили не лише підвищити швидкість роботи програми, але й знизити

витрати на інфраструктуру. Це зробило програму більш стабільною, масштабованою та готовою до обробки великих обсягів даних і запитів. Оптимізація веб-застосунку стала ключовим фактором покращення його конкурентоспроможності та відповідності сучасним вимогам ринку.

ВИСНОВКИ

1. Проведено аналіз сучасних підходів до оптимізації веб-додатків, який показав, що інтеграція методів оптимізації на стороні клієнта та сервера є ключовою для підвищення продуктивності та зручності використання системи.
2. Визначено основні методи серверної оптимізації, зокрема кешування даних (Redis), та оптимізацію запитів до баз даних (PostgreSQL), які дозволяють суттєво скоротити час відповіді системи і забезпечити стабільну роботу під високим навантаженням.
3. Вивчено інструменти клієнтської оптимізації, включаючи мініфікацію HTML, CSS та JavaScript, впровадження лінивого завантаження контенту (LazyLoad), яке забезпечує значне зменшення часу завантаження та покращення інтерактивності.
4. Розроблено та реалізовано систему для зберігання файлів, яка демонструє практичне застосування описаних методів оптимізації. Використання PostgreSQL як основної бази даних, Redis для кешування та AWS S3 для зберігання файлів підтвердило ефективність обраних технологій у реальних умовах.
5. Результати тестування показали, що впроваджені оптимізаційні рішення забезпечують високу продуктивність системи, скорочують час завантаження контенту та покращують користувацький досвід, що робить їх релевантними для впровадження в сучасних веб-додатках.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jeremy Wagner. Web Performance in Action: Building Faster Web Pages 2017 с. 376

2. Google Web Performance Report, 2022 URL: https://developers.google.com/speed/docs/insights/v5/about (дата звернення 30.11.2024)

3. Performance Optimization Techniques for Modern Web Applications 2021

4. Tal Ater Building Progressive Web Apps 2017 с. 285

5. Database Optimization for Large-Scale Applications URL: https://enteros.com/database-performance-management/database-performance-optimization-for-big-data-applications/ (дата звернення 30.11.2024)

6. Chris Richardson. Microservices Patterns: With examples in Java 2018 с. 520

7. Nagy Ramadan Darwish, Ihab Mohamed Abdelwahab. Impact of Implementing HTTP/2 in Web Services. International Journal of Computer Applications (0975 – 8887) Volume 147 – No. 9, August 2016 P. 27-32

8. Regina O. Obe, Leo S. Hsu. PostgreSQL: Up and Running, 3rd Edition, 2017 с. 312

9. Dimitri Fontaine. The Art of PostgreSQL. 2019 с. 438

10. Magic Quadrant for Cloud Database Management Systems, 2023 URL: https://www.gartner.com/doc/reprints?id=1-2ES4ML11&ct=230823&st=sb&trk=55006229-842b-4bc2-9335-49beda3a5fc8&sc_channel=el (дата звернення 30.11.2024)

11. Dr. Josiah L Carlson. Redis in Action, Second Edition.2016 с. 320

12. Martin Kleppmann Designing Data-Intensive Applications. 2017 с. 611

13. Redis Memory & Performance Optimizatio. 2023 URL: https://www.dragonflydb.io/guides/redis-memory-and-performance-optimization (дата звернення 30.11.2024)

14. Kasun IndrasiriSriskandarajah Suhothayan Cloud Computing Design Patterns. 2021 с. 250

15. Joe Baron. Aws Solutions Architect Associate Sg (Aws Certified Solutions Architect Official: Associate Exam. 2016 с. 504

16. Mario Casciaro, Luciano Mammino. Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques, 3rd Editio. 2020 с. 664

17. Diogo Resende. High Performance Node.js. 2016 с. 136

18. Zac Gordon. React Explained: Your Step-by-Step Guide to React. 2020 с. 366

19. Noel Rappin. Modern Front-End Development for Rails: Hotwire, Stimulus, Turbo, and React 2nd Edition. 2022 с. 410

20. Laszlo Bocso. Mastering Tailwind CSS: Crafting Modern, Responsive Designs. 2024 с. 246s

21. Vite vs. Webpack: A Head-to-Head Comparison 2023 URL: https://kinsta.com/blog/vite-vs-webpack/ (дата звернення 30.11.2024)

22. Артеменко Артур магістрант 1 курсу ІННІ ім. Ю. М. Потебні Наук. кер.: канд. техн. наук, доц. Міхайлуца О.М. ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ ТА ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ WEB – ДОДАТКІВ. ЗБІРНИК наукових праць студентів, аспірантів, докторантів і молодих вчених «МОЛОДА НАУКА-2024», Запоріжжя : Запорізький національний університет, 2024. С. 164-165

23. Артеменко А. І., Міхайлуца О. М. Вдосконалення вебдодатків: дослідження методів оптимізації та підвищення продуктивності. Інженерні інновації та розбудова національної економіки : матеріали І Міжнародної науково-практичної конференції Інженерного навчально-наукового інституту ім. Ю. М. Потебні (09–10 травня 2024 року, м. Запоріжжя) / наук. ред. Н. Г. Метеленко ; Інженерний навчально-науковий інститут ім. Ю. М. Потебні Запорізького національного університету. – Одеса : Видавничий дім «Гельветика», 2024. С. 597-598

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Артеменко Артур Ігорович , студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz19bd-01@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Інтелектуальна інформаційна система туристичного комплексу**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 12.12.2024 Підпис _____ Артеменко Артур Ігорович
(студент)

Дата 12. 12. 2024 Підпис _____ Міхайлуца Олена Миколаївна
(науковий керівник)