ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи

бакалавра
(освітньо-кваліфікаційний рівень)

на тему *«Розробка веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API»*

Виконав: здобувач *4* року навчання
групи *4ПР1*
напряму підготовки (спеціальності)
121-Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Кікнадзе Артем Рудольфович
(підпис, прізвище та ініціали)
Керівник *к.т.н., доцент Хохлов В. А.*
(підпис, прізвище та ініціали)
Рецензент *к.т.н., доцент Вороненко М. О.*
(прізвище та ініціали)
Нормоконтролер *Комісаров О. С.*
(підпис, прізвище та ініціали)

Хмельницький – 2026

ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Інститут, факультет, відділення *Інформаційних технологій та дизайну*
Кафедра, циклова комісія *Програмних засобів і технологій*
Освітньо-кваліфікаційний рівень *бакалавр*
Освітньо-професійна підготовка *Програмна інженерія*
(шифр і назва)
Спеціальність *121-Інженерія програмного забезпечення*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри *програмних засобів і технологій*
_____ О.Є. Огнєва
«___» _____ 2026 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА

Кікнадзе Артему Рудольфовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Розробка веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API»

керівник роботи к.т.н., доцент Хохлов В. А.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «16» січня 2026 року №82-с

2. Строк подання здобувачем проєкту (роботи) 10.06.2026

3. Вихідні дані до проєкту (роботи) літературні та періодичні джерела, матеріал переддипломної практики

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз бізнес-процесів програмного забезпечення.
2. Проєктування веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API
3. Розробка програмного продукту
4. Тестування та впровадження програмного продукту

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

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 10.02.2026

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів виконання роботи	Термін виконання етапів роботи	Примітки
1.	Отримання завдання	10.02.2026	Виконано
2.	Підбір літератури	12.02.2026-20.02.2026	Виконано
3.	Аналіз предметної області	21.02.2026-27.02.2026	Виконано
4.	Розробка та обґрунтування завдання	28.02.2026-13.03.2026	Виконано
5.	Розробка концептуальної моделі	14.03.2026-20.03.2026	Виконано
6.	Моделювання та проектування системи	21.03.2026-03.04.2026	Виконано
7.	Моделювання та проектування бази даних	04.04.2026-10.04.2026	Виконано
8.	Розробка інтерфейсу сайту	11.04.2026-17.04.2026	Виконано
9.	Тестування сайту	18.04.2026-24.04.2026	Виконано
10.	Оформлення пояснювальної записки	25.04.2026-01.06.2026	Виконано
11.	Захист кваліфікаційної роботи		Виконано

Здобувач _____ А. Р. Кікнадзе
(підпис) (прізвище та ініціали)

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

РЕФЕРАТ

Кваліфікаційна робота бакалавра: 65 сторінок, 17 рисунків, 1 таблиця, 2 додатки, 31 джерело.

Мета роботи:

Розробити веб-сервіс для синхронізації мультимедійного контенту між месенджерами на основі відкритого API.

Об'єкт розробки:

Методи та технології, що використовуються для розробки веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API.

Предмет розробки:

Веб-сервіс для синхронізації мультимедійного контенту між месенджерами на основі відкритого API.

Інструменти та засоби розробки:

Для розробки створюваного веб-сервісу використано TypeScript, React, SCSS, NestJS, PostgreSQL, Docker, Git, Github Actions, Nginx, Firebase, Amazon Web Services S3, JetBrains WebStorm, Windows Subsystem for Linux.

Результати роботи:

Результатом роботи є веб-сервіс для синхронізації мультимедійного контенту між месенджерами на основі відкритого API.

Новизна рішення:

Запропоновано новий підхід до синхронізації мультимедійного контенту між месенджерами через платформу з відкритим API; удосконалено підхід до організації доступу до мультимедійних ресурсів із використанням механізмів авторизації для зовнішніх клієнтів; розроблено архітектуру сервісу, що забезпечує централізоване зберігання контенту, його пошук за ключовими словами та можливість інтеграції сторонніх застосунків.

КЛЮЧОВІ СЛОВА: *СЕРВІС, МУЛЬТИМЕДІЙНИЙ КОНТЕНТ, NESTJS, REACT, TYPESCRIPT, AWS S3, POSTGRESQL, DOCKER, FIREBASE, REST API, CI/CD.*

АНОТАЦІЯ

Кваліфікаційна робота бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі під назвою «Аналіз бізнес-процесів програмного забезпечення» виконано загальну характеристику предметної області. Окрім цього було змодельовано та проаналізовано бізнес-процеси сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API і постановку задачі проектування.

Другий розділ «Проєктування веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API» складається з наступних підрозділів: «Аналіз програмних інструментів», «Розробка проєктних специфікацій» та «Висновки до розділу 2». У розділі виконано обґрунтування вибору технологій розробки, сформовано функціональні та нефункціональні вимоги до системи, розроблено структуру бази даних, діаграми взаємодії компонентів та специфікації REST API.

У третьому розділі «Розробка програмного продукту» наведено чотири підрозділи: «Розробка серверної частини», «Розробка клієнтської частини», «Розробка моделей даних» та «Висновки до розділу 3». Розглянуто особливості реалізації серверної та клієнтської частин системи та структуру програмного продукту, описано реалізовані модулі, роботу із мультимедійними файлами та інтеграціями. Окрім цього наведено опис основних сторінок користувацького інтерфейсу та моделей даних, що використовуються у системі.

Четвертий розділ «Тестування та впровадження програмного продукту» складається з п’яти розділів: «Тестування програмного додатку», «Розробка інтеграції з Telegram», «Робота користувача зі створеним веб-сервісом», «Встановлення та експериментальне застосування програми», «Висновки до розділу 4». У ньому описано процес тестування серверної та клієнтської частин застосунку, розробки інтеграції із Telegram та роботи користувача із системою. Окрім цього розглянуто особливості розгортання веб-сервісу із використанням Docker та засобів безперервної інтеграції і доставки програмного забезпечення.

Отримані результати підтвердили коректність роботи системи та її відповідність поставленим вимогам.

ABSTRACT

The bachelor's thesis consists of an introduction, four chapters, conclusions, a list of references, and appendices.

The first chapter, titled “Analysis of Software Business Processes,” provides a general overview of the subject area. In addition, the business processes of a service for synchronizing multimedia content between messengers based on an open API were modeled and analyzed, and the design problem was formulated.

The second chapter, “Design of a Web Service for Synchronizing Multimedia Content Between Messengers Based on an Open API,” consists of the following subsections: “Analysis of Software Tools,” “Development of Design Specifications,” and “Conclusions to Chapter 2.” This chapter justifies the choice of development technologies, defines the functional and non-functional requirements for the system, and develops the database structure, component interaction diagrams, and REST API specifications.

The third chapter, “Software Product Development,” contains four subsections: “Server-Side Development,” “Client-Side Development,” “Data Model Development,” and “Conclusions for Chapter 3.” It examines the specifics of implementing the server-side and client-side components of the system and the structure of the software product, and describes the implemented modules, working with multimedia files, and integrations. Additionally, it provides a description of the main user interface pages and the data models used in the system.

The fourth chapter, “Testing and Deployment of the Software Product,” consists of five sections: “Testing the Software Application,” “Development of Integration with Telegram,” “User Interaction with the Created Web Service,” “Installation and Experimental Use of the Application,” and “Conclusions to Chapter 4.” It describes the process of testing the server-side and client-side components of the application, developing integration with Telegram, and user interaction with the system. In addition, it examines the specifics of deploying the web service using Docker and continuous integration and software delivery tools. The results obtained

confirmed the correct operation of the system and its compliance with the specified requirements.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API - Application Programming Interface, інтерфейс прикладного програмування

REST - Representational State Transfer, передача репрезентативного стану

JSON - JavaScript Object Notation, нотація об’єктів JavaScript

SQL - Structured Query Language, мова структурованих запитів

ORM - Object Relational Mapping, об’єктно-реляційне відображення

CI/CD - Continuous Integration / Continuous Delivery, безперервна інтеграція безперервна доставка

HTTP - HyperText Transfer Protocol, протокол передачі гіпертексту

ВСТУП

На сьогоднішній день для обміну інформацією між користувачами знайшли широке застосування месенджери та соціальні мережі. Проте, організація збереження, пошуку та синхронізації мультимедійного контенту між різними платформами є непростим завданням через необхідність реалізації зберігання мультимедійних файлів користувачів у сховищі, реалізації пошукової системи, створення API та зрозумілої документації, спрощення інтеграції для розробників.

Актуальність теми. На сьогоднішній день у сучасному цифровому середовищі комунікація між користувачами відбувається через месенджери і соціальні мережі. У таких комунікаціях використовуються різноманітні медіафайли: зображення, стікери, відео, GIF-анімації та текстові повідомлення. Для звичайних користувачів вони здебільшого виконують функцію вираження емоцій, проте бізнес використовують їх для реклами своїх послуг, надання інформації про них та ін. Тому у користувачів виникає потреба у системах, що дозволяють зручно зберігати, знаходити та використовувати повторно медіа-матеріали.

Для визначення основних вимог до створюваної інформаційної системи було проведено аналіз існуючих аналогів. Серед найбільш відомих подібних веб-сервісів не було знайдено повноцінних альтернатив до створюваного додатку.

Об'єкт розробки - методи та технології, що використовуються для синхронізації мультимедійного контенту між різними месенджерами за допомогою відкритого API.

Предмет розробки - веб-сервіс для синхронізації мультимедійного контенту між месенджерами на основі відкритого API.

Інструменти та засоби дослідження: Для розробки веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API використовуються методи та технології розробки веб-додатків, такі як TypeScript, React, SCSS, NestJS, PostgreSQL, Docker, Git, Github Actions,

Nginx, Firebase, Amazon Web Services S3, JetBrains WebStorm, Windows Subsystem for Linux.

Мета і задачі дослідження. Метою кваліфікаційної роботи бакалавра є створення веб-сервісу для синхронізації зображень/GIF/стікерів між месенджерами. Синхронізація відбувається через відкрите API, що дозволяє розробникам реалізувати інтеграцію до будь-якого месенджеру самостійно.

Основним завданням кваліфікаційної роботи бакалавра на тему «Розробка веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API» є дослідження технології та створення веб-додатку на основі отриманих знань та навичок.

Для досягнення поставленої мети необхідно виконати ряд **завдань**:

- Проаналізувати існуючі підходи та сервіси;
- Розробити архітектуру веб-сервісу з можливістю масштабування;
- Спроєктувати та реалізувати API для взаємодії із зовнішніми застосунками;
- Реалізувати механізм авторизації та керування доступом для користувачів і сторонніх розробників;
- Забезпечити збереження, обробку та швидкий доступ до медіафайлів;
- Реалізувати клієнтську частину;
- Провести тестування працездатності системи та оцінити ефективність її роботи.

Новизна рішення. Наукова новизна одержаних результатів полягає у наступному:

- запропоновано новий підхід до синхронізації мультимедійного контенту між месенджерами через платформу з відкритим API;
- удосконалено підхід до організації доступу до мультимедійних ресурсів із використанням механізмів авторизації для зовнішніх клієнтів;
- розроблено архітектуру сервісу, що забезпечує централізоване зберігання контенту, його пошук за ключовими словами та можливість інтеграції сторонніх застосунків;

Практичне значення одержаних результатів. Практичним значенням роботи є створення готового веб-сервісу для збереження та пошуку за ключовими словами мультимедійного контенту і можливість інтеграції сторонніх застосунків.

Апробація результатів бакалаврської кваліфікаційної роботи. Одержані результати можна використовувати для удосконалення методики та інструментарію для створення веб-додатків для синхронізації мультимедійного контенту між месенджерами на основі відкритого API, запропонований веб-додаток використовувати для подальшого удосконалення цієї методики.

Структура: робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел, додатків. Кваліфікаційна робота бакалавра складається з 65 сторінок, 17 рисунків, 1 таблиці, 2 додатків, 31 джерела.

РОЗДІЛ 1. АНАЛІЗ БІЗНЕС-ПРОЦЕСІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Опис та загальна характеристика предметної області

На сьогоднішній день у сучасному цифровому середовищі комунікація між користувачами відбувається через месенджери і соціальні мережі. У таких комунікаціях використовуються різні медіа-матеріали: зображення, стікери, відео, GIF-анімації та текстові повідомлення. Для звичайних користувачів вони здебільшого виконують функцію вираження емоцій: сміх, сум, здивування та ін. Тим часом бізнес використовують їх для реклами своїх товарів і надання інформації про них. Тому у користувачів виникає потреба у системах, що дозволяють зручно зберігати, знаходити та використовувати повторно медіа-матеріали.

У рамках даної роботи предметною областю є процес організації централізованого збереження та синхронізації мультимедійного контенту між месенджерами та платформами за допомогою веб-сервісу із відкритим API. Основною задачею створюваної інформаційної системи є надання користувачам можливості зберігати медіафайли та текстові записи у єдиному середовищі з можливістю їх пошуку за ключовими словами.

Для визначення основних вимог до системи необхідно провести аналіз існуючих аналогів, а саме визначити їх недоліки та переваги, щоб сформувати перелік необхідних функціональних можливостей. Серед найбільш відомих подібних сервісів можна виділити наступні:

- **Pastebin**. Дозволяє створювати текстові записи та встановлювати рівень приватності для них. На відміну від інших аналогів має відкрите API, яке дозволяє отримувати створені користувачем записи. Проте, сервіс не підтримує мультимедійні дані, які є надто важливими у наш час.

- **Pinterest**. Є соціальною платформою для організації та збереження зображень. Проте, незважаючи на наявність API, не дозволяє повноцінно зберігати та управляти контентом аналогічно розроблюваній системі.
- **Tenor**. Є пошуковою системою та базою даних GIF, має велику користувацьку базу та вже інтегрований в багатьох месенджерах. Але 3 січня 2026 року API Tenor було закрито компанією Google.

Отже, провівши аналіз предметної області ми бачимо, що дійсно існує необхідність у створенні веб-сервісу для централізованого збереження мультимедійного контенту, його пошуку та інтеграції пошуку у сторонні сервіси через відкритий API. Використання сучасних веб-технологій дозволяє реалізувати масштабовану систему з можливістю подальшого розвитку та розширення функціоналу.

1.2. Моделювання та аналіз бізнес-процесів предметної області

Перш ніж почати проєктування й розробку веб-сервісу для синхронізації мультимедійного контенту необхідно проаналізувати, як саме працює бізнес-система у поточний момент часу.

На сьогоднішній день більшість користувачів активно використовують декілька платформ одночасно. Наприклад, Telegram використовується для спілкування в особистих повідомленнях та групах, Discord - для комунікації у спільнотах, а Viber - для комунікації у шкільних та робочих процесах. Перелічені платформи не мають синхронізації мультимедійних файлів між собою, у результаті чого мультимедійний контент зберігається у різних місцях, що значно ускладнює його організацію та повторне використання.

Основним бізнес-процесом у даній предметній області є процес збереження, пошуку та повторного використання мультимедійного контенту. Через те, що частіше всього цей процес виконується користувачами вручну, виникають наступні проблеми:

- складність організації великої кількості мультимедійних файлів;

- відсутність синхронізації між пристроями;
- відсутність інтеграції сторонніх застосунків;
- втрата часу на пошук необхідного контенту.

У сучасній веб-розробці для організації API-запитів найчастіше використовуються REST API та GraphQL.

GraphQL - це мова запитів та маніпулювання даними, створене компанією Facebook у 2015 році [10]. Особливістю даної мови є можливість клієнта самостійно визначати, які саме дані необхідно отримати від сервера. Також GraphQL дозволяє формувати гнучкі запити, що містять лише необхідні поля. Це дозволяє значно зменшити обсяг переданих даних, та тим самим підвищити ефективність взаємодії між клієнтом і сервером.

REST API - це архітектурний підхід до побудови веб-сервісів [14], який надає доступ до інформаційних ресурсів через HTTP-запити. Дані у цьому підході повинні передаватися у вигляді невеликої кількості стандартних форматів (HTML, XML, JSON) [16]. REST API повинен підтримувати кешування і не повинен зберігати інформації про стан між парами «запит-відповідь». Проте при роботі зі складними структурами даних клієнт може отримувати надлишкову інформацію і буде змушений виконувати декілька запитів для отримання необхідних даних.

Тим не менш, у розробленому веб-сервісі реалізовано архітектуру REST API через простоту її інтеграції із зовнішніми сервісами, що є критично важливим для сервісу з відкритим API. У результаті процес збереження та повторного використання медіафайлів виглядає таким чином: користувач знаходить необхідний мультимедійний контент у одному месенджері та за допомогою чат-бота зберігає його у створюваному сервісі. Згодом користувач має змогу дістати цей файл у будь-якій іншій платформі. Наприклад, GIF-анімація, що була збережена через Telegram-бота, може бути знайдена та відправлена користувачу через Discord або інший підключений до системи через API застосунок.

Важливим бізнес-процесом є процес створення нового запису. Користувач завантажує файл через веб-інтерфейс сервісу або через інтерфейс стороннього застосунку. Після цього сервер перевіряє коректність запиту, виконує авторизацію користувача за допомогою Firebase Authentication [7] або API-токену та зберігає файл у хмарному сховищі.

Отже бачимо, що існуючі підходи до організації мультимедійного контенту між месенджерами мають ряд недоліків через відсутність централізованого збереження та складність пошуку контенту та інтеграції між платформами. Запропонований веб-сервіс дозволяє вирішити такі проблеми шляхом використання відкритого API, централізованого сховища та механізмів пошуку за ключовими словами.

1.3. Постановка задачі проектування

Після проведення аналізу предметної області та існуючих бізнес-процесів було встановлено, що сучасні месенджери та соціальні мережі не надають універсального механізму для централізованого збереження та синхронізації мультимедійного контенту між платформами. Користувачам необхідно власноруч пересилати файли між застосунками та дублювати контент у різних сервісах.

Існуючі рішення не підтримують повноцінно мультимедійний контент та мають обмежені можливості інтеграції через API. Через це виникає необхідність у створенні веб-сервісу для забезпечення централізованого збереження, пошук та синхронізацію мультимедійного контенту між різними месенджерами.

Основною метою проєктування є розробка веб-сервісу із відкритим API для збереження та синхронізації мультимедійного контенту між сторонніми платформами та месенджерами.

Для досягнення поставленої мети необхідно виконати ряд задач:

- Проаналізувати існуючі підходи та сервіси;

- Розробити архітектуру веб-сервісу з можливістю масштабування;
- Спроєктувати та реалізувати API для взаємодії із зовнішніми застосунками;
- Реалізувати механізм авторизації та керування доступом для користувачів і сторонніх розробників;
- Забезпечити збереження, обробку та швидкий доступ до медіафайлів;
- Реалізувати клієнтську частину;
- Провести тестування працездатності системи та оцінити ефективність її роботи.

Таким чином, задача полягає у створенні сучасного веб-сервісу для синхронізації мультимедійного контенту між месенджерами із використанням відкритого API, хмарного сховища та клієнт-серверної архітектури. Для взаємодії між клієнтською та серверною частинами використовується архітектурний підхід REST API.

1.4. Висновки до розділу 1

В цьому розділі була розглянута бізнес-система, яка потребує комплексної веб-платформи для організації і синхронізації мультимедійного контенту між сучасними месенджерами і платформами. Було проведено детальний опис і аналіз існуючих аналогів, визначено їх переваги та недоліки. Існуючі сервіси не забезпечують повноцінної синхронізації та мають обмеження щодо інтеграції через API.

Результати проведеного аналізу підтверджують актуальність теми та необхідність створення веб-сервісу для синхронізації мультимедійного контенту між месенджерами із використанням сучасних веб-технологій та клієнт-серверної архітектури.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБ-СЕРВІСУ ДЛЯ СИНХРОНІЗАЦІЇ МУЛЬТИМЕДІЙНОГО КОНТЕНТУ МІЖ МЕСЕНДЖЕРАМИ НА ОСНОВІ ВІДКРИТОГО API

2.1. Аналіз програмних інструментів

На основі проведеного у першому розділі аналізу предметної області було визначено функціональні та нефункціональні вимоги до програмного додатку. Функціональні вимоги описують поведінку між вхідними та вихідними даними. Вони включають маніпулювання та обробку даних, розрахунки, технічні деталі та інші специфічні функції. Нефункціональні вимоги описують як саме повинна працювати система.

Отже, основними функціональними вимогами нашої системи є:

- реєстрація та авторизація користувачів;
- створення, завантаження та збереження текстових та мультимедійних записів;
- створення ключових слів для створених записів;
- зручний та гнучкий пошук записів за ключовими словами;
- забезпечити стороннім розробникам можливість інтегрувати свої застосунки через API;

Основними нефункціональними вимогами системи є адаптивність, зберігання даних, документація, надійність, масштабованість, безпека даних, можливість підтримки та тестування, зручність використання, інтернаціоналізація та локалізація.

Для розробки створюваного веб-сервісу для синхронізації мультимедіа між месенджерами було використано низку технологій для забезпечення високої продуктивності, масштабованості та безпеки.

У якості мови програмування використовується TypeScript, яка є розширенням JavaScript та забезпечує статичну типізацію з необов’язковими анотаціями типів [3, 30]. Це дозволить зменшити кількість помилок під час

розробки за рахунок типізації даних. На відміну від мов програмування, що використовуються лише для серверної частини, TypeScript дозволяє реалізовувати повний стек веб-застосунку із використанням єдиного набору технологій. Це спрощує підтримку проєкту та дозволяє швидше вносити зміни до клієнтської та серверної частин системи.

Для реалізації клієнтської частини використовується бібліотека React [23] з використанням модульного архітектурного підходу. Структура проєкту поділена на окремі функціональні модулі, що забезпечує кращу масштабованість та повторне використання компонентів системи. Для тестування клієнтської частини використовується фреймворк Vitest.

Для збірки та запуску клієнтської частини використовується Vite. Vite забезпечує швидкий запуск середовища розробки, підтримує гаряче оновлення модулів (Hot Module Replacement) та дозволяє пришвидшити процес збірки проєкту у порівнянні із традиційними інструментами збірки. Окрім цього Vite має зручну інтеграцію із React та TypeScript.

Для стилізації інтерфейсу клієнтської частини використовується SCSS [25] - розширення CSS, що дозволяє використовувати вкладеність, змінні та повторне використання окремих стилістичних елементів. Це спрощує підтримку та масштабування стилів.

За допомогою бібліотеки TanStack Query [27] реалізовано керування серверним станом у клієнтській частині. Дана технологія дозволяє спростити роботу із API-запитами, автоматичним оновленням інформації та обробкою асинхронних операцій. TanStack Query допомагає значно зменшити кількість ручного керування станом застосунку.

Для реалізації серверної частини було обрано фреймворк NestJS [18], що працює на базі Node.js [20]. Він використовує прогресивний TypeScript (хоча й надає розробникам можливість писати код на чистому JavaScript), а також поєднує елементи ООП (об’єктно-орієнтованого програмування), ФП (функціонального програмування) та ФРП (функціонально-реактивного програмування).

Авторизація користувачів реалізована завдяки Firebase Authentication [7]. Firebase забезпечує безпечну автентифікацію користувачів та підтримує різні методи входу до системи.

За допомогою Swagger створюється документація створюваного REST API [21, 26]. Дана технологія дозволяє автоматично генерувати документацію API, що спрощує інтеграцію сторонніх застосунків із веб-сервісом. NestJS має спеціальні модулі та плагіни для генерації таких специфікацій за допомогою декораторів.

Для збереження даних використовується СУБД PostgreSQL [22]. Це забезпечує високу надійність, підтримку складних SQL-запитів та хорошу продуктивність при роботі із великими обсягами даних.

Взаємодія серверу із базою даних відбувається завдяки TypeORM [29]. Дана бібліотека дозволяє працювати із базою даних через об’єктну модель та спрощує виконання SQL-операцій.

За збереження мультимедійних файлів відповідає хмарне сховище AWS S3 [2]. Даний сервіс забезпечує надійне збереження файлів. Під час локальної розробки використовувався MinIO - програмне забезпечення, сумісне із S3 API, що дозволяє тестувати систему без використання реального хмарного сховища.

За допомогою Docker забезпечується контейнеризації застосунку [4], що дозволяє забезпечити однакове середовище виконання на різних пристроях та спрощує процес розгортання системи. Docker також дозволяє ізолювати окремі компоненти системи, завдяки чому спрощується їх масштабування.

Для організації роботи веб-сервера та проксіювання HTTP-запитів використовується Nginx [19]. Даний сервер забезпечує перенаправлення запитів між клієнтською та серверною частинами системи, а також може використовуватись для балансування навантаження та обробки статичних файлів.

CI/CD (Continuous Integration / Continuous Delivery) підхід відповідає за автоматизацію збірки, тестування та розгортання застосунку. Його реалізація у нашому проєкті здійснюється за допомогою GitHub Actions [9].

Для забезпечення контролю якості програмного коду використовується ESLint [5]. Він дозволяє знаходити потенційні помилки, дотримуватися єдиного стилю та читабельності програмного коду.

Система контролю версій Git використовується для керування змінами програмного коду [8]. Для стандартизації повідомлень комітів використовується підхід Conventional Commits [1], який передбачає використання єдиного формату назв комітів. Основними типами комітів є feat, fix та refactor.

У якості інтегрованого середовища розробки використовується JetBrains WebStorm. Він надає зручні інструменти для роботи із TypeScript та Node.js, підтримує автоматичний аналіз коду, налагодження, інтеграцію із системами контролю версій та ESLint.

Під час розробки використовується Windows Subsystem for Linux [31], що дозволяє запускати Linux-середовище у Windows без використання окремої віртуальної машини. Використання WSL спрощує роботу із Docker, Node.js та іншими інструментами веб-розробки, що орієнтовані загалом на Linux-системи.

2.2. Розробка проєктних специфікацій

Розробка сучасного веб-сервісу із використанням REST API архітектури вимагає точного опису взаємодії між клієнтською та серверною частинами системи. Проєктні специфікації дозволяють визначити структуру REST API, правила взаємодії між компонентами системи, формати передачі даних та механізми авторизації користувачів і сторонніх застосунків. Специфікації REST API створюваного веб-сервісу представлені у табличному вигляді та винесені у додаток до пояснювальної записки.

Розроблена специфікація містять 9 ендпоїнтів, що згруповані за функціональними блоками. Передача даних між клієнтською та серверною частинами здійснюється у форматі JSON. Для завантаження мультимедійних файлів використовується формат multipart/form-data. Для реалізації API використовуються стандартні HTTP-методи [16]: GET, POST, PATCH, DELETE.

Ключовим модулем створюваного REST API є модуль мультимедійного контенту (Pasta). Передбачено повний набір CRUD операцій для створення, оновлення, перегляду та видалення записів.

Не менш важливим є модуль сторонніх інтеграцій (Applications). Даний модуль дозволяє створювати API-ключі та керувати інтеграціями користувача. Основними ендпоїнтами applications є отримання списку, створення та видалення інтеграцій і оновлення API-токену.

У рамках проєктних специфікацій також визначено правила обробки помилок. У випадку виникнення помилки сервер повертає відповідний HTTP статус та JSON-відповідь із описом проблеми.

Проєктні специфікації також використовуються під час тестування серверної частини. Створений опис API дозволяє виконувати автоматизоване тестування HTTP-запитів та перевіряти коректність роботи окремих модулів системи.

Таким чином, розроблені проєктні специфікації формують технічну основу веб-сервісу для синхронізації мультимедійного контенту між месенджерами. Вони забезпечують стандартизовану взаємодію між компонентами системи, спрощують інтеграцію сторонніх застосунків та дозволяють підтримувати масштабованість створюваного програмного продукту.

2.3. Висновки до розділу 2

У даному розділі було здійснено аналіз програмних інструментів та технологій, необхідних для розробки створюваного веб-сервісу синхронізації мультимедійного контенту між месенджерами. Функціональні та нефункціональні вимоги допомогли у формуванні стеку технологій для реалізації клієнтської та серверної частин системи.

Використання TypeScript, React, NestJS та PostgreSQL дозволяє реалізувати масштабований веб-сервіс із можливістю подальшого розвитку та

підтримки. Для збереження мультимедійного контенту було обрано хмарне сховище AWS S3, а для забезпечення безпечної автентифікації користувачів - Firebase Authentication.

У рамках розділу було виконано розробку проєктних специфікацій REST API. Було визначено основні функціональні модулі системи, структуру HTTP-запитів та формат взаємодії між клієнтською і серверною частинами застосунку.

Даний розділ формує технічну основу створюваного веб-сервісу та дозволяє перейти до етапу реалізації програмного продукту.

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

3.1. Розробка серверної частини

Для реалізації серверної частини додатку було використано фреймворк NestJS, що працює на базі Node.js, та мову програмування TypeScript. Для обробки запитів використовуються контролери, що приймають HTTP-запити та передають їх у сервіси, які реалізують основну бізнес-логіку системи. Контролери та сервіси є складовими модулів, кожен модуль відповідає за окрему частину функціоналу системи. Список модулів та пов'язаних із ними сервісів і контролерів наведено у табл. 3.1.

Модуль AppModule є головним модулем, який підключає та з’єднує інші модулі системи (рис. 3.1). CoreModule також є об’єднуючим модулем, але він об’єднує такі компоненти системи, як DatabaseModule, FirebaseModule та StorageModule (рис. 3.2).

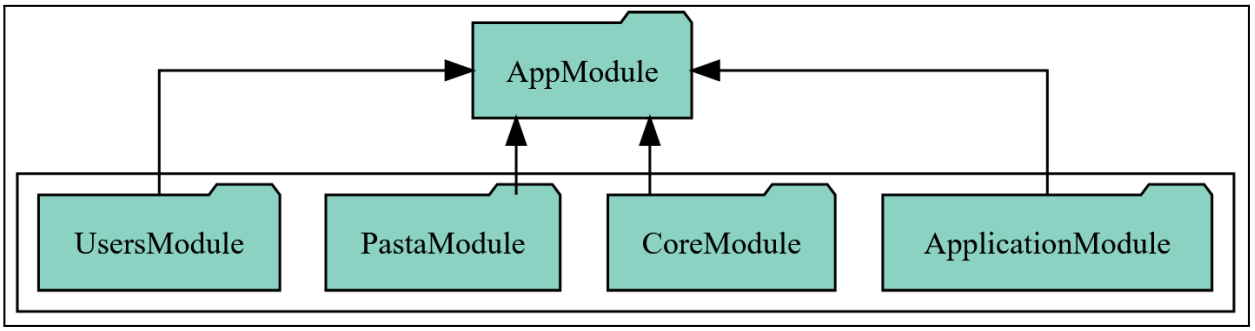

Рисунок 3.1. Залежності модуля AppModule

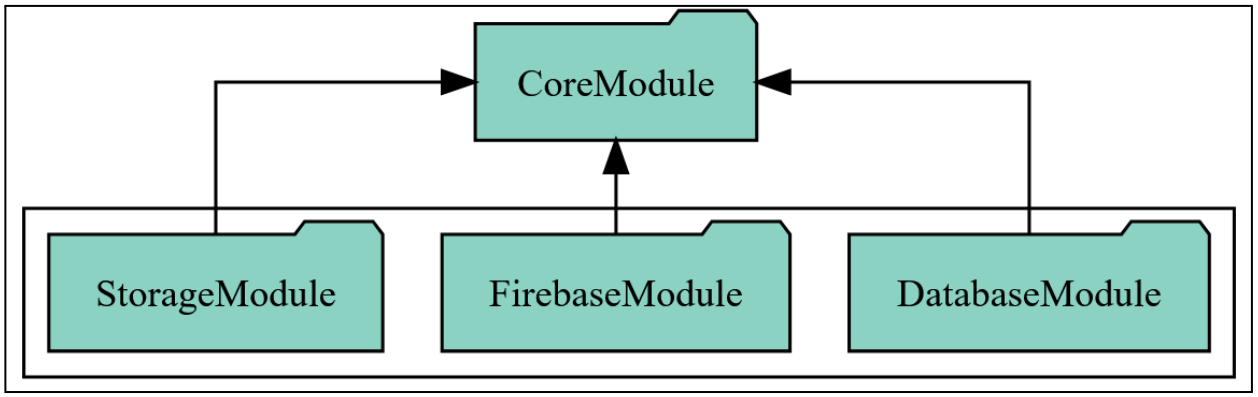

Рисунок 3.2. Залежності модуля CoreModule.

Таблиця 3.1

Модулі серверної частини веб-застосунку

Модуль	**Контролер**	**Сервіс**	**Опис**
AppModule	-	-	Головний модуль, підключення інших модулів системи
UsersModule	-	UsersService	Створення, пошук та керування користувачами
PastaModule	PastaController	PastaService	Створення, редагування, видалення та пошук записів
ApplicationModule	ApplicationController	ApplicationService	Створення інтеграцій, генерація та перевірка API-ключів, керування інтеграціями
AuthModule	-	-	Авторизація користувачів, перевірка Firebase та API токенів, захист маршрутів
CoreModule	-	-	Містить компоненти системи, перехоплювачі (interceptors) та допоміжні утиліти
DatabaseModule	-	DatabaseService	Підключення та конфігурація бази даних
FirebaseModule	-	FirebaseService	Взаємодія із Firebase Authentication
StorageModule	-	StorageService	Завантаження, збереження та отримання мультимедійних файлів із AWS S3 та MinIO.

ApplicationModule (рис. 3.3) відповідає за створення інтеграцій, генерацію та перевірку API-ключів і керування інтеграціями. Модуль імпортує UsersModule для авторизації у контролерах та DatabaseModule для збереження даних інтеграцій. ApplicationService експортується для використання у інших модулях.

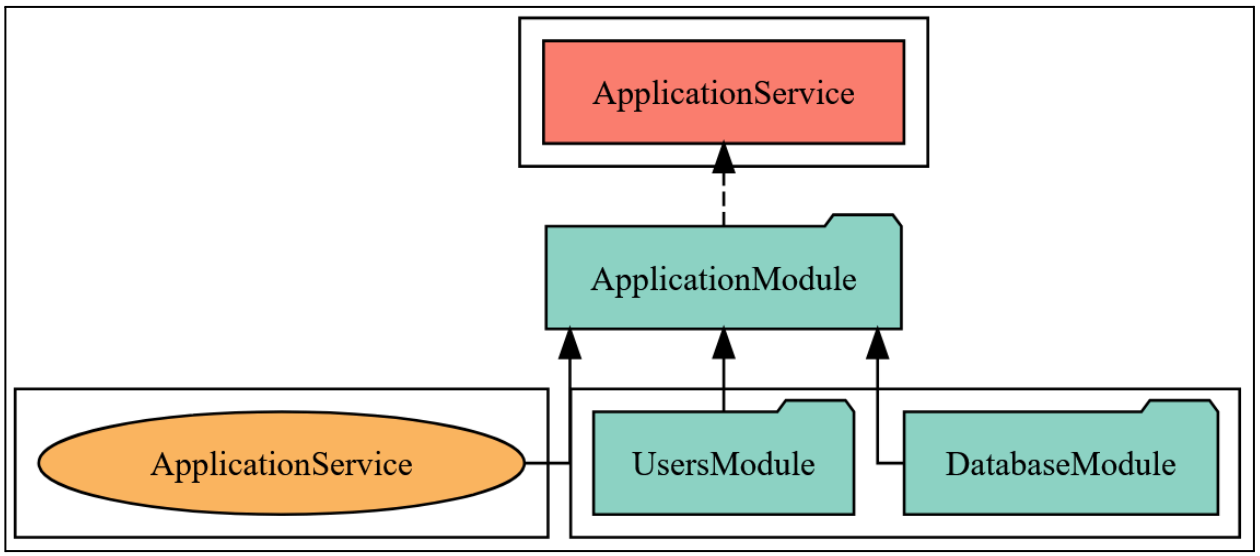

Рисунок 3.3. Модуль ApplicationModule.

PastaModule (рис. 3.4) відповідає за створення, редагування, видалення та пошук мультимедійних записів. Модуль імпортує UsersModule для авторизації у контролерах, StorageModule для збереження та отримання мультимедійних файлів, DatabaseModule для збереження даних інтеграцій. ApplicationModule використовується для логування дій сторонніх застосунків. PastaService експортується для використання у інших модулях.

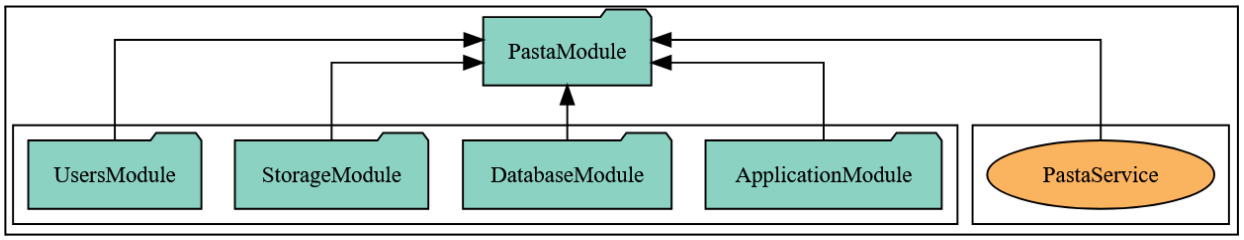

Рисунок 3.4. Модуль PastaModule.

На лістингу 3.1 наведено фрагмент методу створення мультимедійного запису. Метод виконує перевірку вхідних даних, завантажує мультимедійний файл до сховища, формує новий об'єкт сутності Pasta та зберігає його у базі даних. Після успішного виконання операції створюється запис у журналі подій інтеграції, що дозволяє відстежувати використання відкритого API сторонніми застосунками.

Для взаємодії з базою даних використовується репозиторій Repository<Pasta>, який надається ORM-бібліотекою TypeORM. Репозиторій

реалізує шаблон проєктування Repository та забезпечує абстракцію над операціями взаємодії із базою даних. Тип Repository<T> є узагальненим класом, де параметр T визначає сутність, з якою буде виконуватись робота. У даному випадку використовується Repository<Pasta>, що забезпечує типобезпечну роботу із сутністю Pasta та відповідною таблицею бази даних. Репозиторій надає набір готових методів для створення, пошуку, оновлення та видалення записів, завдяки чому операції виконуються без написання SQL-запитів вручну.

Ін'єкція репозиторію виконується за допомогою реалізованого у фреймворку NestJS механізму впровадження залежностей (Dependency Injection). Під час запуску застосунку модуль DatabaseModule створює провайдер репозиторію та реєструє його у контейнері залежностей. Декоратор @Inject('PASTA_REPOSITORY') вказує контейнеру залежностей, який саме об'єкт необхідно передати до змінної pastaRepository. Використання Dependency Injection зменшує зв'язність між компонентами системи та спрощує тестування програмного забезпечення.

Сервіс StorageService забезпечує завантаження мультимедійних файлів до AWS S3 (або сховища MinIO під час розробки), отримання посилань на збережені файли та їх видалення у разі необхідності. Виділення роботи зі сховищем в окремий сервіс дозволяє ізолювати логіку збереження файлів від бізнес-логіки застосунку, завдяки чому спрощується можлива заміна логіки роботи з файлами без необхідності зміни коду сервісів, які їх використовують.

Для отримання параметрів конфігурації використовується вбудований у NestJS ConfigService. Він надає доступ до змінних середовища, які містять параметри підключення до бази даних, файлового сховища, Firebase та інших сервісів. Завдяки цьому немає потреби зберігати конфігураційні дані безпосередньо у програмному коді, що могло би привести до витоку даних.

ApplicationService використовується для роботи з інтеграціями, які взаємодіють із веб-сервісом через відкритий API. Під час виконання CRUD-операцій над мультимедійними даними сервіс формує записи про

виконані дії, що дозволяє відстежувати активність інтеграцій та спрощує діагностику можливих помилок.

Лістинг 3.1. Фрагмент методу створення мультимедійного запису

```
@Injectable()
export class PastaService {
  constructor(
    @Inject('PASTA_REPOSITORY')
    private readonly pastaRepository: Repository<Pasta>,
    private readonly storageService: StorageService,
    private readonly applicationService: ApplicationService,
    private configService: ConfigService,
  ) {}
  async create(
    req: Request, createPastaDto: CreatePastaDto, file?: Express.Multer.File,
  ) {
    if (!createPastaDto.text && !file) {
      throw new NoTextOrFileProvidedException();
    }

    const storageFile = file && (await this.storageService.uploadFile(
        file, `${req.user!.uid}-${new Date().getTime()}`, file.mimetype,
    ));

    const pasta = this.pastaRepository.create({
      owner: { uid: req.user!.uid }, keywords: createPastaDto.keywords,
      text: createPastaDto.text, ...(file
        ? {
            file: {
              url: storageFile
                ? '/s3/' + storageFile.Bucket + '/' + storageFile.Key
                : undefined,
              mimetype: file?.mimetype,
              size: file?.size,
            },
          }
        : {}),
    });
    const createRequest = await this.pastaRepository.save(pasta);

    if (createRequest?.id && req.application) {
      this.applicationService
```

```
        .createLog({...})
        .catch(() => {});
    }
    return this.fullFileUrl(createRequest);
  }
  ...
}
```

Контролер PastaController (лістинг 3.2) відповідає за обробку HTTP-запитів від клієнтської частини. Для захисту маршрутів використовується механізм UseGuards, який перевіряє автентифікацію користувача перед виконанням операцій. Створений AnyAuthGuard перевіряє отриманий у запиті токен, визначає, чи був він наданий користувачем або сторонньою інтеграцією, та перевіряє токен на валідність. Контролер реалізує набір маршрутів для отримання, створення, редагування та видалення мультимедійного контенту, після чого передає обробку запитів до сервісного рівня застосунку.

Лістинг 3.2. Контролер PastaController

```
@UseGuards(AnyAuthGuard)
@Controller('pasta')
export class PastaController {
  constructor(private readonly pastaService: PastaService) {}

  @Get()
  async findByUser(...) {
    return await this.pastaService.findByUser(req, query);
  }

  @Post()
  @UseInterceptors(FileInterceptor('file'))
  async create(...) {
    return await this.pastaService.create(req, createPastaDto, file);
  }

  @Patch(':id')
  async edit(...) {
    return await this.pastaService.update(+id, req, updatePastaDto);
  }

  @Delete(':id')
```

```
  async remove(...) {
    return await this.pastaService.remove(+id, req);
  }
}
```

Взаємодія серверу із базою даних відбувається завдяки TypeORM. Використання ORM дозволяє описувати структуру таблиць у вигляді класів мови програмування TypeScript. Кожна сутність відповідає окремій таблиці бази даних. Зв'язки між таблицями описуються за допомогою декораторів, що дозволяє працювати із пов'язаними даними через об'єктну модель. У лістингу 3.3 наведено приклад коду створення сутності Pasta, яка створює таблицю у базі даних PostgreSQL. Ініціалізація TypeORM та підключення до бази даних PostgreSQL відбувається у модулі DatabaseModule.

Лістинг 3.3. Сутність Pasta

```
@Entity()
export class Pasta {
  @PrimaryGeneratedColumn()
  id: number;

  @ManyToOne(() => User, (user) => user.pastas)
  owner: User;

  @Column({ nullable: true })
  text?: string;

  @Column({ length: 2048 })
  keywords: string;

  @OneToOne(() => File, (file) => file.pasta, {
    nullable: true,
    cascade: true,
    eager: true,
  })
  @JoinColumn()
  file?: File;
}
```

Взаємодія між клієнтською та серверною частинами відбувається через HTTP-запити у форматі JSON. Повний перелік реалізованих кінцевих точок REST API наведено у додатку А.

Для документування REST API використовується Swagger (рис. 3.5), який автоматично генерує документацію, що значно спрощує інтеграцію сторонніх застосунків. У документації використовуються лише /pasta ендпоїнти, адже /applications не призначена для сторонніх розробників.

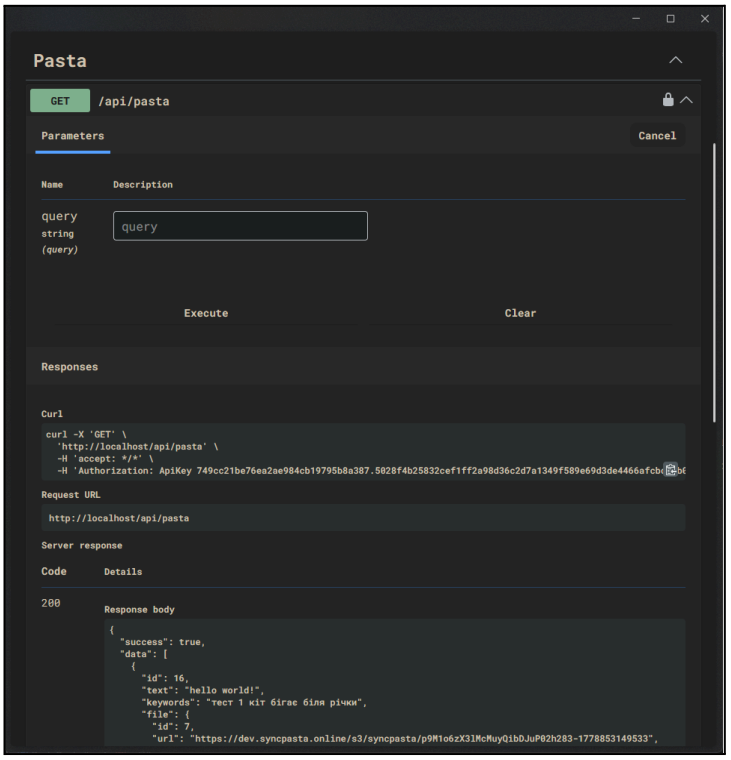

Рисунок 3.5. Приклад документації ендпоїнту /api/pasta

3.2. Розробка клієнтської частини

Для реалізації клієнтської частини використовується бібліотека React. У якості мови програмування також використовується TypeScript. Перехід між сторінками застосунку використовується без повторного завантаження завдяки використанню технології Single Page Application (SPA) [24]. Для збірки та запуску клієнтської частини використовується інструмент Vite, який окрім цього забезпечує швидкий запуск середовища розробки, підтримує гаряче оновлення

модулів (Hot Module Replacement) та значно пришвидшує процес збірки проєкту.

Структура клієнтської частини організована за модульною архітектурою, яка поділяється на рівні. Існує 6 рівнів, розташованих від найбільшої відповідальності та залежності до найменшої:

- App - усе, що забезпечує роботу додатка: маршрутизація, точки входу, глобальні стилі та провайдери.
- Pages - повні сторінки, містять у собі Widgets.
- Widgets - великі самостійні блоки функціональності або інтерфейсу, що зазвичай реалізують цілий сценарій використання.
- Features - повторно використовувані реалізації цілих функцій продукту, тобто дій, що приносять користувачеві бізнес-цінність.
- Entities - бізнес-суб’єкти (наприклад: pasta або application).
- Shared - функціональність, що підлягає повторному використанню, наприклад: компоненти інтерфейсу, axios instance, firebase config.

Для реалізації клієнтської частини було розроблено наступні сторінки:

- головна сторінка (рис. 3.6)

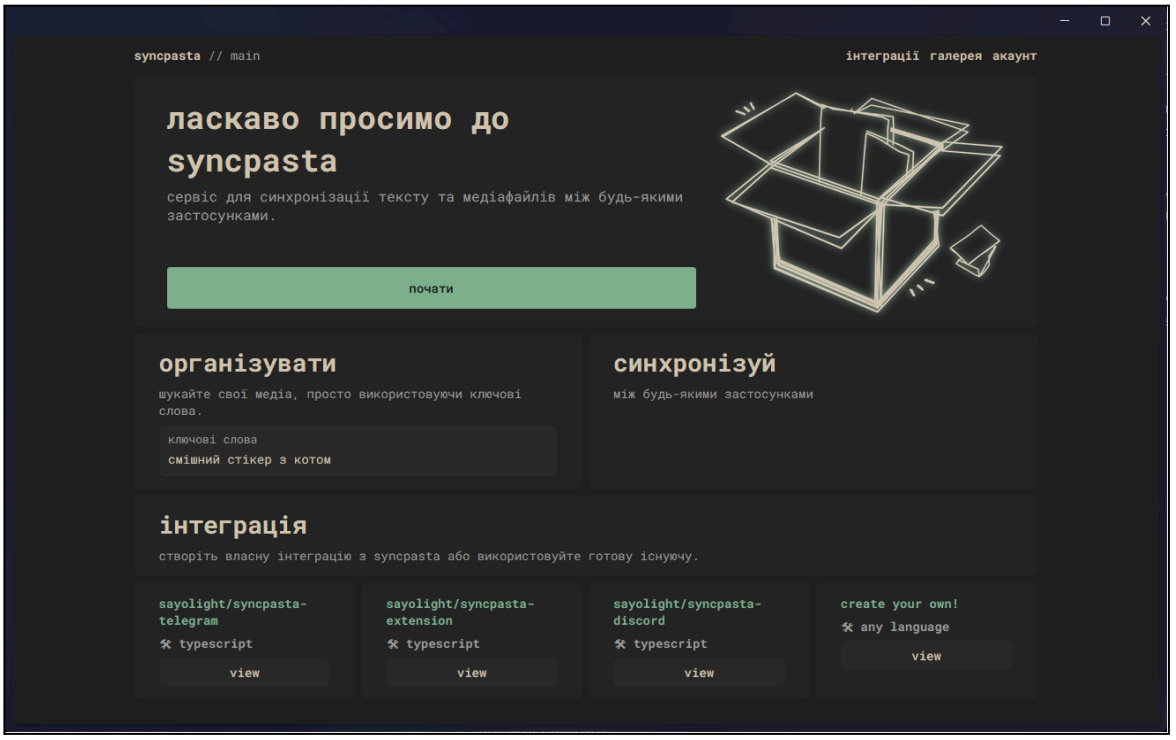

Рисунок 3.6. Головна сторінка

- сторінка авторизації (рис. 3.7)

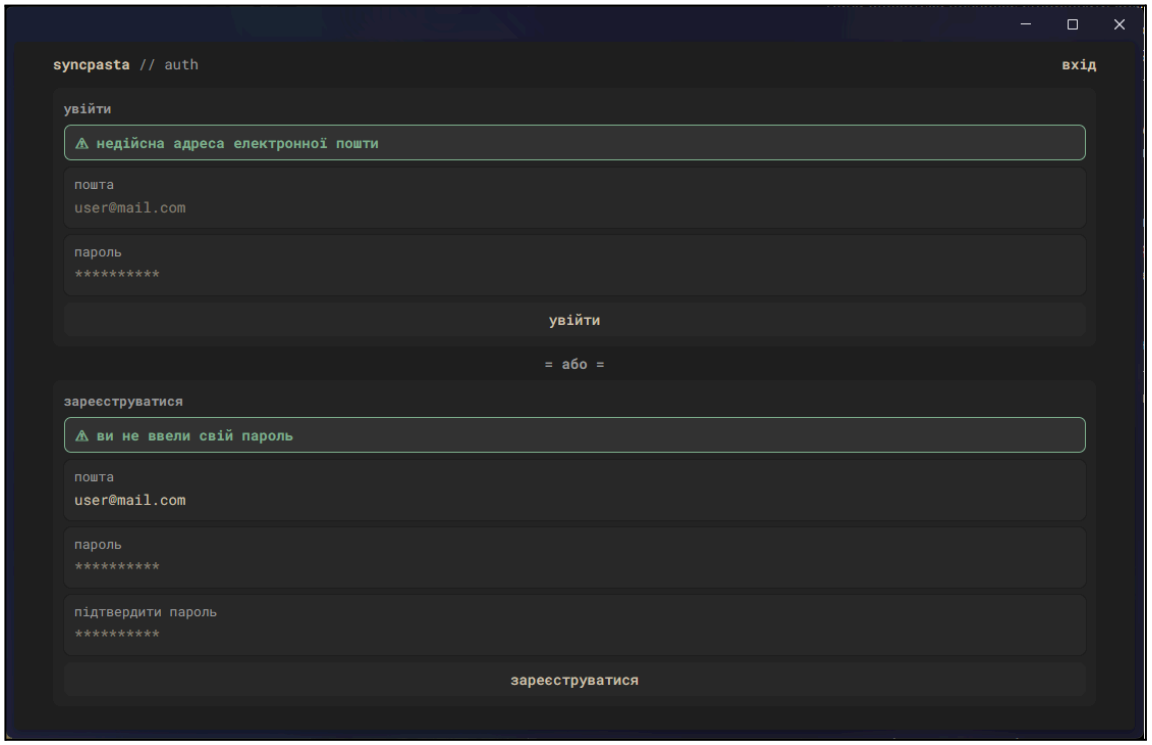

Рисунок 3.7. Сторінка авторизації

- сторінка галереї (рис. 3.8)

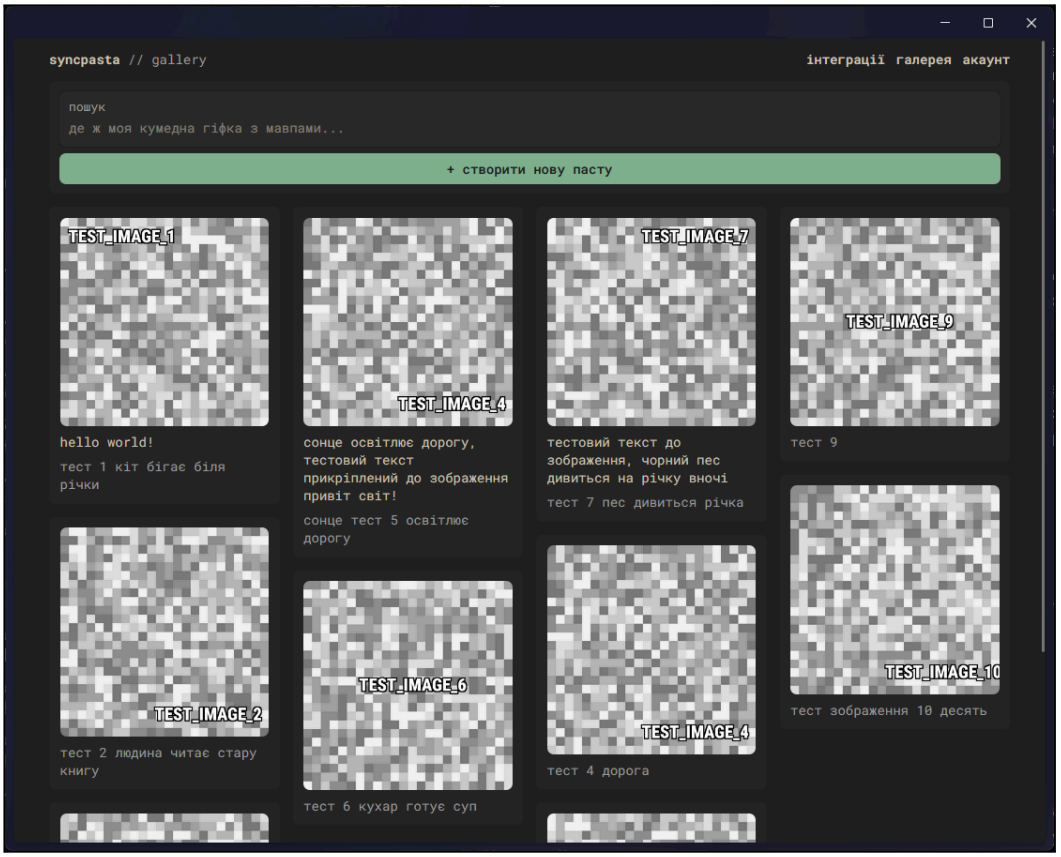

Рисунок 3.8. Сторінка галереї

- сторінка інтеграцій (рис. 3.9)

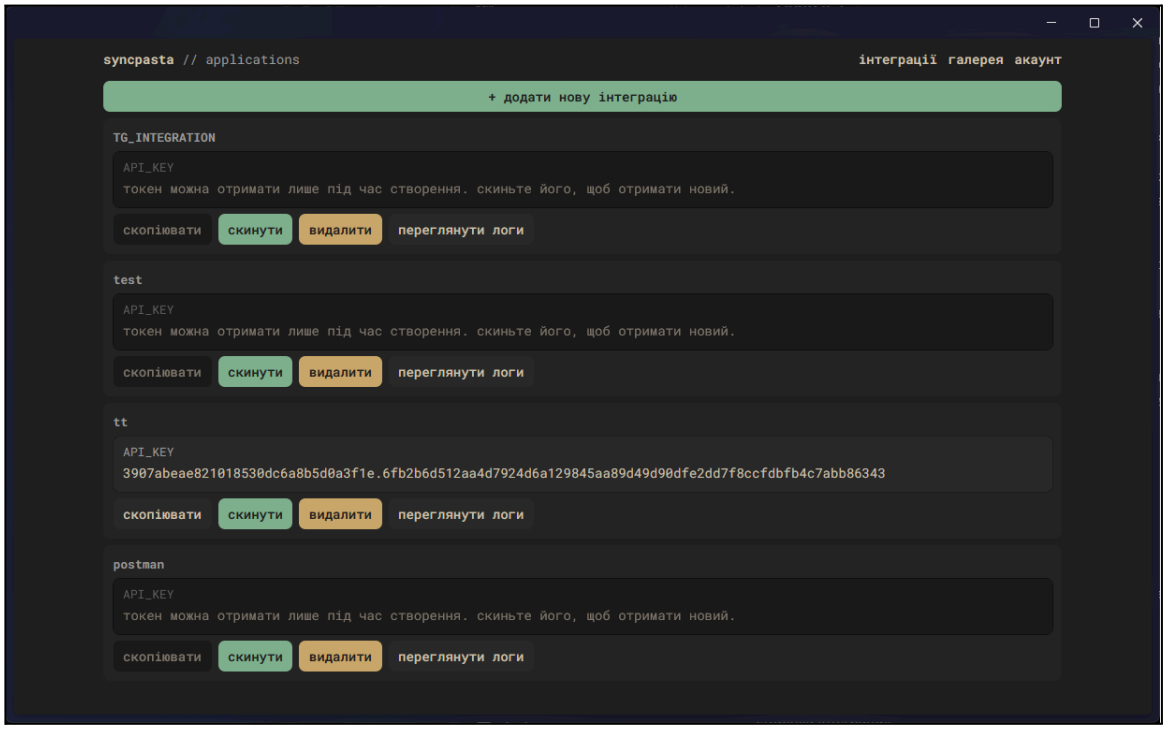

Рисунок 3.9. Сторінка інтеграцій

- сторінка профілю користувача (рис. 3.10)

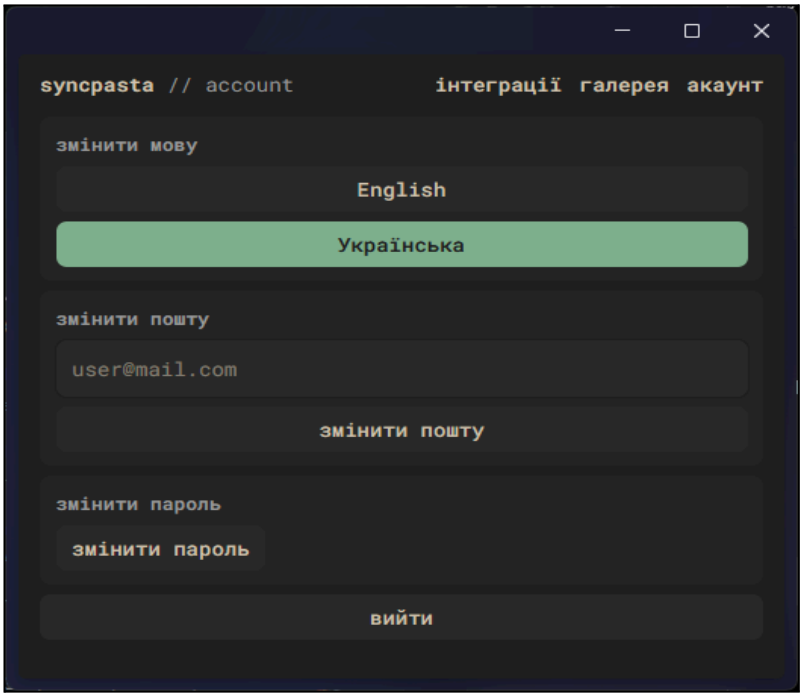

Рисунок 3.10. Сторінка профілю користувача

Бібліотека React Router відповідає за реалізацію навігації між сторінками додатку без перезавантаження сторінки, завдяки чому кінцевий користувач отримує більш зручну та швидку взаємодію із веб-застосунком.

Для виконання HTTP-запитів до серверної частини використовується бібліотека Axios. Дана бібліотека спрощує роботу із REST API, обробку помилок та налаштування HTTP-запитів. За допомогою interceptors реалізовано автоматичне додавання токену авторизації запитів та обробку відповідей у більш зручний формат. TanStack Query забезпечує отримання, кешування та автоматичного оновлення даних із серверної частини. Дана бібліотека дозволяє не створювати вручну React-хуки для кожного запиту, щоб отримувати стан (error, isError, isPending, data) та спрощує роботу із асинхронними операціями.

Компоненти застосунку (Рис. 3.11) розміщені у директорії /shared/ui та містять у собі React tsx файл, SCSS модуль стилів та index.ts файл для експортування компоненту. Для клієнтської частини було створено 10 UI-компонентів:

- Alert - компонент повідомлень. Використовується для відображення інформації про успішний або помилковий запит.
- Block - компонент блоку. Використовується для виокремлення сегментів сторінки.
- Button - кнопка.
- Illustration - компонент ілюстрацій. Використовується для сторінок помилок, порожніх станів галереї та інтеграцій.
- Input - компонент однорядкового вводу тексту.
- Textarea - компонент багаторядкового вводу тексту.
- Loader - компонент анімованої ілюстрації завантаження. Відображається під час отримання та оновлення даних.
- Modal - компонент модального вікна. Використовується для відображення форм.
- Timeline - компонент хронології подій. Використовується для відображення логів дій сторонніх застосунків.
- Typography - текстовий компонент для використання єдиного стилю форматування тексту.

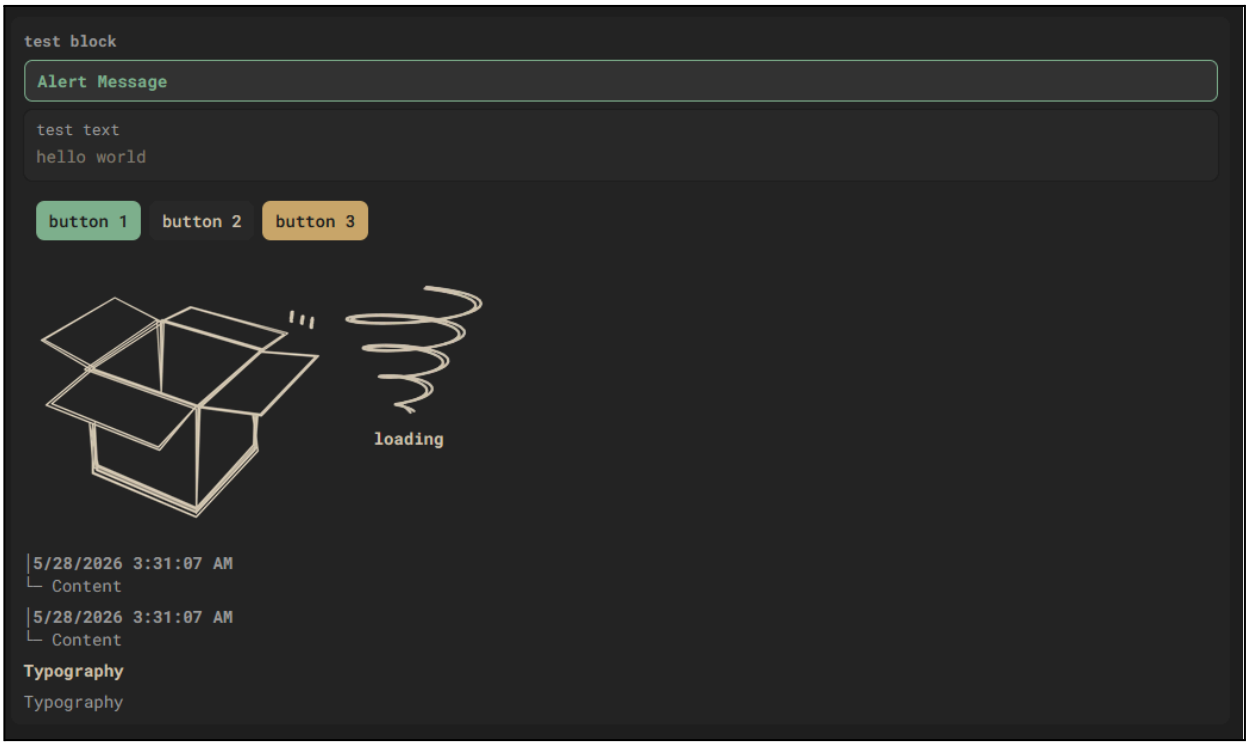

Рисунок 3.11. UI-компоненти у різних станах

Entities є поняттям з реального світу, з якими працює проект. Це терміни, які є описом продукту. У створюванному веб-сервісі використовуються бізнес-об’єкти application і pasta. Application відповідає за роботу із інтеграціями сторонніх застосунків і містить у собі інтерфейси Application та ApplicationLogs. Окрім цього, Application містить HTTP-запити, реалізовані через TanStack Query, до серверу для отримання, створення, видалення та оновлення інтеграцій.

Pasta відповіає за мультимедійний контент користувача. Містить CRUD HTTP-запити до /pasta, інтерфейси Pasta та File і UI-компонент PastaCard, що відповідає за відображення мультимедійного контенту у галереї. У лістингу 3.4 та 3.5 наведено код запитів для отримання списку записів та створення нового контенту.

Лістинг 3.4. React-хук запиту для отримання списку записів з використанням TanStack Query

```
import { http } from "@/shared/api/https.ts";
import { useQuery } from "@tanstack/react-query";
import type { Pasta } from "@/entities/pasta";
```

```
async function fetchPastas(query?: string): Promise<Pasta[]> {
  const response = await http.get<Pasta[]>(`/pasta`, {
    params: {
      query,
    },
  });
  return response.data;
}

export function usePastas(query?: string) {
  return useQuery({
    queryKey: ["pastas", query],
    queryFn: () => fetchPastas(query),
  });
}
```

Лістинг 3.5. React-хук запиту для створення нового запису з використанням TanStack Query

```
import { http } from "@/shared/api/https.ts";
import { type Pasta } from "@/entities/pasta";
import { useMutation, useQueryClient } from "@tanstack/react-query";

async function createPasta(data: FormData): Promise<Pasta> {
  const response = await http.post<Pasta>("/pasta/", data);
  return response.data;
}

export function useCreatePasta() {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: createPasta,
    onSuccess: () => {
      queryClient.invalidateQueries({
        queryKey: ["pastas"],
      });
    },
  });
}
```

Widgets використовується для побудови окремих функціональних сегментів сторінки. Вони є великими незалежними компонентами інтерфейсу,

які об’єднують у собі UI-компоненти, бізнес-логіку та взаємодію із entities/features.

У створюваному веб-сервісі реалізовано віджети email-update для оновлення пошти, error-alert для відображення інформації про помилки, password-reset для скидання паролю та pasta-board для відображення галереї мультимедійного контенту. Реалізовані віджети використовуються або можуть при масштабуванні використовуватися повторно у різних частинах веб сайту.

Таким чином, використання React, TypeScript та сучасних бібліотек дозволило забезпечити зручність використання, масштабованість та швидку взаємодію клієнту із серверною частиною веб-сервісу.

3.3. Розробка моделей даних

Важливим етапом розробки створюваного веб-сервісу є проєктування бази даних (рис. 3.12). Правильна організація моделей сприяє швидкості роботи системи, зручності підтримки програмного коду та можливості подальшого масштабування проєкту. Для реалізації сховища даних використовується реляційна система керування базами даних PostgreSQL.

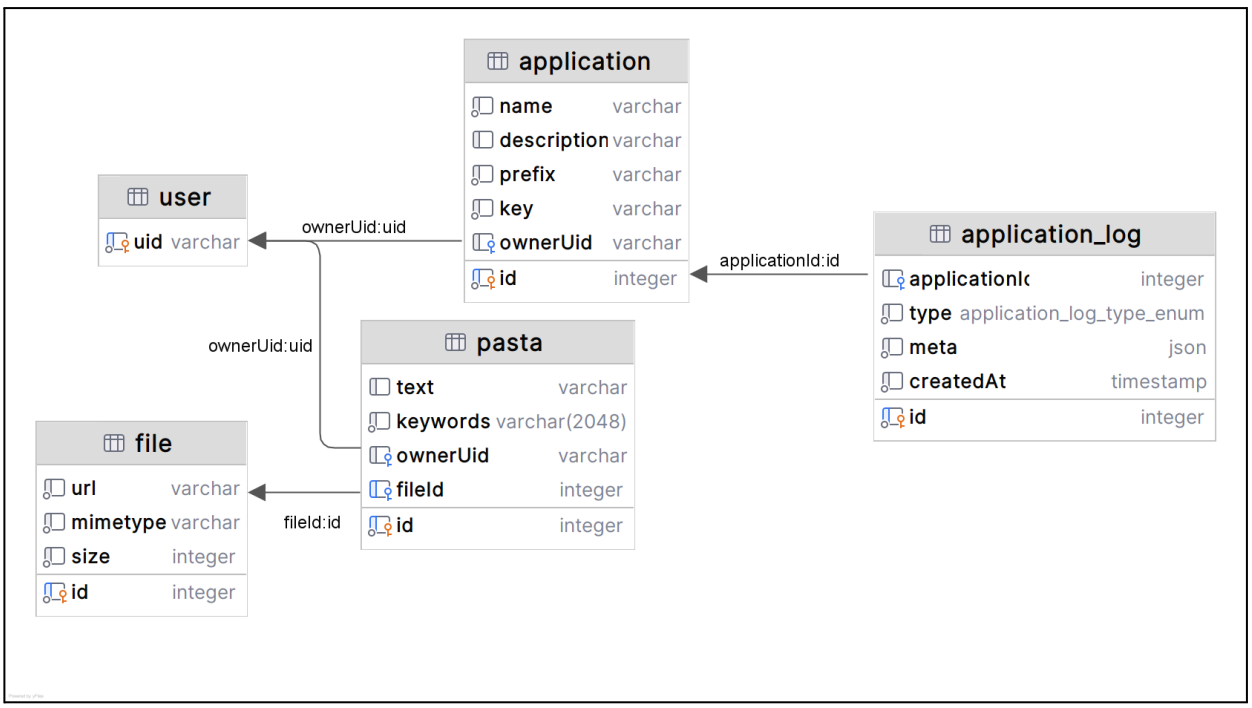

Рисунок 3.12. Діаграма бази даних

У результаті аналізу предметної області було визначено п’ять основних сутностей системи: User, Pasta, File, Application та ApplicationLog.

Сутність User використовується для збереження інформації про користувачів веб-сервісу. Завдяки використанню Firebase Authentication у базі даних зберігається лише id користувача.

Сутність Pasta є основною сутністю системи та використовується для збереження мультимедійного контенту. Запис містить текстову інформацію, зображення або відеофайл, а також набір ключових слів для пошуку. Між сутностями User та Pasta реалізовано зв’язок “один до багатьох”, адже один користувач має багато записів. Основними атрибутами Pasta є:

- id - ідентифікатор контенту;
- text - текст;
- keywords - ключові слова;
- ownerUid - ідентифікатор власника;
- fileId - ідентифікатор файлу;

Для кожного запису додатково зберігається інформація про файл у таблиці File. Основними атрибутами File є:

- id - ідентифікатор файлу;
- url - посилання на файл у S3 сховище;
- mimetype - MIME-тип файлу;
- size - розмір файлу;

Сутність Application містить інформацію про інтеграції сторонніх застосунків. Кожен користувач може створювати декілька інтеграцій для різних платформ. Між сутностями User та Application реалізовано зв’язок “один до багатьох”, адже один користувач може мати декілька інтеграцій. Основними атрибутами сутності Application є:

- id - ідентифікатор інтеграції;
- name - назва інтеграції;
- description - опис інтеграції;
- prefix - нехешований префікс API-ключу;

- key - хешований API-ключ;
- ownerUid - ідентифікатор власника.

Сутність ApplicationLog використовується для логування дій сторонніх застосунків. Вона дозволяє відстежувати виконані запити, завдяки чому можна перевірити, чи не виконує стороння інтеграція зайві запити. Між сутностями Application та ApplicationLog реалізовано зв’язок “один до багатьох”, адже одна інтеграція має багато записів журналу. Основними атрибутами сутності є:

- id - ідентифікатор запису журналу;
- applicationId - ідентифікатор інтеграції, що створила запис;
- type - тип події;
- meta - додаткова інформація про подію у форматі JSON;
- createdAt - дата створення запису;

Для взаємодії серверної частини із базою даних використовується ORM-бібліотека TypeORM. Вона дозволяє описувати моделі даних у вигляді TypeScript-класів та автоматично формувати SQL-запити для роботи із PostgreSQL.

Отже, у результаті проєктування моделей даних було сформовано структуру бази даних. Розроблена модель забезпечує цілісність даних, підтримує масштабування системи та дозволяє ефективно реалізувати необхідні функції.

3.4. Висновки до розділу 3

У даному розділі було виконано розробку веб-сервісу для синхронізації мультимедійного контенту між месенджерами на основі відкритого API. Реалізовано серверну частину системи із використанням фреймворку NestJS та мови програмування TypeScript. Для збереження мультимедійних файлів використано хмарне сховище AWS S3, а для роботи із реляційною базою даних PostgreSQL застосовується ORM-бібліотека TypeORM.

Окрім цього було розроблено клієнтську частину із використанням бібліотеки React та мови програмування TypeScript. Інтерфейс користувача побудовано на основі багаторазових UI-компонентів, що забезпечують зручність використання та спрощують подальший розвиток проєкту.

Окрему увагу приділено проєктуванню моделей даних. Було сформовано структуру бази даних, що забезпечує збереження інформації про користувачів, мультимедійні записи, інтеграції та журнали подій. Розроблена модель підтримує цілісність даних та масштабованість системи.

Отримані результати підтверджують можливість створення сучасного веб-сервісу для централізованого збереження та синхронізації мультимедійного контенту між різними платформами із використанням відкритого API.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ ПРОГРАМНОГО ПРОДУКТУ

4.1. Тестування програмного додатку

Після завершення етапу розробки було проведено тестування веб-сервісу для перевірки коректності роботи реалізованого функціоналу. Основною метою тестування є виявлення можливих помилок у роботі серверної та клієнтської частин. Тестування проводилося на етапах розробки та після завершення реалізації основного функціоналу. Для тестування серверної частини використовувалась бібліотека Jest [12], а для клієнтської - фреймворк Vitest.

У рамках тестування серверної частини було перевірено коректність роботи механізмів авторизації користувачів, створення та пошуку мультимедійних записів та роботи інтеграцій. На лістингу 4.1 наведено приклад теста сервісу PastaService, що використовується для опрацювання записів користувача.

Лістинг 4.1. Тест сервісу PastaService

```
import { Test, TestingModule } from '@nestjs/testing';
import { ConfigService } from '@nestjs/config';
import { Request } from 'express';
import { PastaService } from './pasta.service';
import { StorageService } from '../../core/storage/storage.service';
import { ApplicationService } from '../applications/application.service';
import {
  NoTextOrFileProvidedException,
  PastaNotFoundException,
} from './pasta.exceptions';

describe('PastaService', () => {
  let pastaService: PastaService;

  const req = {
    user: {
      uid: '1',
    },
  } as unknown as Request;
```

```
const pastaRepositoryMock = {
  findBy: jest.fn(),
  findOne: jest.fn(),
  create: jest.fn(),
  save: jest.fn(),
  delete: jest.fn(),
};

const storageServiceMock = {
  uploadFile: jest.fn(),
};

const applicationServiceMock = {
  createLog: jest.fn().mockResolvedValue(undefined),
};

const configServiceMock = {
  get: jest.fn().mockReturnValue('https://api.test.com'),
};

beforeEach(async () => {
  jest.clearAllMocks();

  const module: TestingModule = await Test.createTestingModule({
    providers: [
      PastaService,
      {
        provide: 'PASTA_REPOSITORY',
        useValue: pastaRepositoryMock,
      },
      {
        provide: StorageService,
        useValue: storageServiceMock,
      },
      {
        provide: ApplicationService,
        useValue: applicationServiceMock,
      },
      {
        provide: ConfigService,
        useValue: configServiceMock,
```

```
      },
    ],
  }).compile();

  pastaService = module.get(PastaService);
});

describe('findByUser', () => {
  it('should return pastas', async () => {
    pastaRepositoryMock.findBy.mockResolvedValue([
      {
        id: 1,
        text: 'hello',
      },
    ]);

    const result = await pastaService.findByUser(req);

    expect(pastaRepositoryMock.findBy).toHaveBeenCalled();
    expect(result).toHaveLength(1);
  });

  it('should search by query', async () => {
    pastaRepositoryMock.findBy.mockResolvedValue([]);

    await pastaService.findByUser(req, 'test');

    expect(pastaRepositoryMock.findBy).toHaveBeenCalledWith(
      expect.objectContaining({
        owner: { uid: '1' },
      }),
    );
  });
});

describe('create', () => {
  it('should throw if no data provided', async () => {
    await expect(
      pastaService.create(
        req,
        {
          text: '',
```

```
        keywords: '',
      },
      undefined,
    ),
  ).rejects.toThrow(NoTextOrFileProvidedException);
});

it('should create text pasta', async () => {
  const pasta = {
    id: 1,
    text: 'hello',
    keywords: 'test',
  };

  pastaRepositoryMock.create.mockReturnValue(pasta);
  pastaRepositoryMock.save.mockResolvedValue(pasta);

  const result = await pastaService.create(req, {
    text: 'hello',
    keywords: 'test',
  });

  expect(pastaRepositoryMock.create).toHaveBeenCalled();
  expect(pastaRepositoryMock.save).toHaveBeenCalled();

  expect(result).toEqual(pasta);
});

it('should save pasta with file', async () => {
  storageServiceMock.uploadFile.mockResolvedValue({
    Bucket: 'syncpasta',
    Key: 'file.png',
  });

  // eslint-disable-next-line @typescript-eslint/no-unsafe-return
  pastaRepositoryMock.create.mockImplementation((x) => x);

  pastaRepositoryMock.save.mockResolvedValue({
    id: 1,
    file: {
      url: '/s3/syncpasta/file.png',
      mimetype: 'image/png',
```

```
        size: 100,
      },
    });

    const file: Express.Multer.File = {
      mimetype: 'image/png',
      size: 100,
      buffer: Buffer.from('buffer'),
      originalname: 'file.png',
    } as unknown as Express.Multer.File;

    const result = await pastaService.create(
      req,
      {
        keywords: 'file',
      },
      file,
    );

    expect(storageServiceMock.uploadFile).toHaveBeenCalled();

    expect(result.file?.url).toBe(
      'https://api.test.com/s3/syncpasta/file.png',
    );
  });
});

describe('update', () => {
  it('should throw if pasta does not exist', async () => {
    pastaRepositoryMock.findOne.mockResolvedValue(null);

    await expect(
      pastaService.update(1, req, {
        text: 'updated',
      }),
    ).rejects.toThrow(PastaNotFoundException);
  });

  it('should update pasta', async () => {
    pastaRepositoryMock.findOne.mockResolvedValue({
      id: 1,
      text: 'old',
```

```
      });

      pastaRepositoryMock.save.mockResolvedValue({
        id: 1,
        text: 'updated',
      });

      const result = await pastaService.update(1, req, {
        text: 'updated',
      });

      expect(pastaRepositoryMock.save).toHaveBeenCalled();
      expect(result.text).toBe('updated');
    });
  });

  describe('remove', () => {
    it('should delete pasta', async () => {
      pastaRepositoryMock.delete.mockResolvedValue({
        affected: 1,
      });

      const result = await pastaService.remove(1, req);

      expect(pastaRepositoryMock.delete).toHaveBeenCalledWith({
        id: 1,
        owner: {
          uid: '1',
        },
      });

      expect(result).toBe(1);
    });
  });
});
```

За результатами проведеного тестування критичних помилок виявлено не було. Усі основні функції веб-сервісу працюють відповідно до визначених проєктних специфікацій.

4.2. Розробка інтеграції з Telegram

Головною перевагою створеного веб-сервісу є можливість інтеграції із сторонніми платформами через відкритий API. Для демонстрації можливостей розробленої системи було розроблено інтеграцію із месенджером Telegram у вигляді чат-бота. Взаємодія між ботом та серверною частиною системи здійснюється через REST API з використанням спеціального токен інтеграції, що створюється користувачем на сторінці керування інтеграціями.

Під час першого запуску користувач виконує прив'язку Telegram-бота до свого облікового запису. Після успішної авторизації бот отримує можливість виконувати запити до API від імені користувача. Для створення нового запису користувач надсилає боту мультимедійний файл (рис. 4.1). Бот отримує повідомлення через Telegram Bot API [28] та передає відповідні дані до серверу. Після успішного збереження (рис. 4.2) сервер повертає інформацію про створений запис.

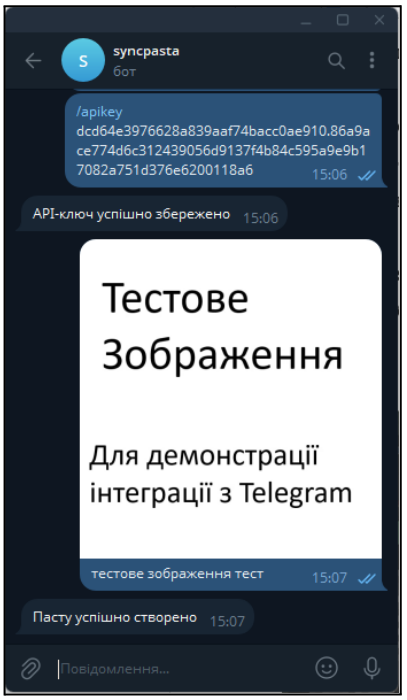

Рисунок 4.1. Взаємодія користувача із Telegram-ботом

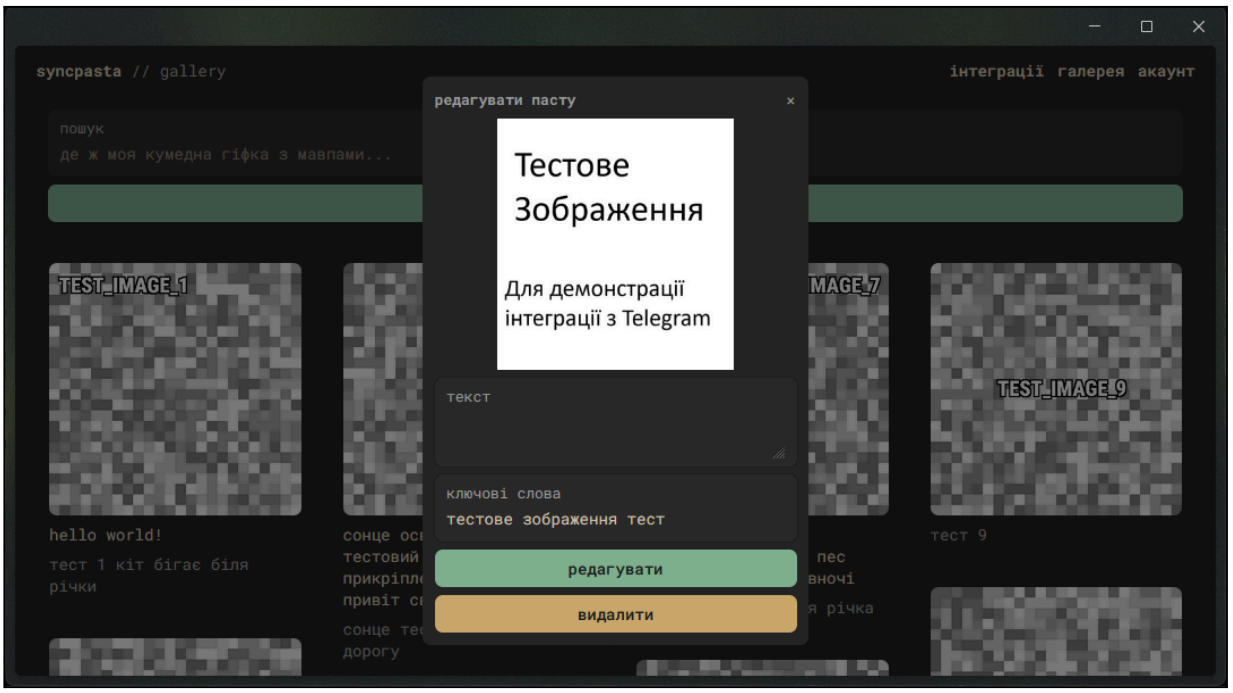

Рисунок 4.2. Успішне збереження файлу через Telegram-бота

У лістингу 4.3 наведено код збереження файлу за допомогою Telegram-інтеграції.

Лістинг 4.3. Збереження файлу через Telegram-бота

```
import { Composer } from "grammy";
import { MyContext } from "../types/MyContext";

export const composer = new Composer<MyContext>();
const create = composer.chatType("private");

create.on([":photo", ":media"], async (ctx: MyContext) => {
    if (ctx.message?.via_bot?.id === ctx.me.id) return;

    const file = await ctx.getFile();
    const fileUrl =
`https://api.telegram.org/file/bot${process.env.BOT_TOKEN}/${file.file_path}`;

    const keywords = ctx.message?.caption;

    const user = await ctx.db.user.findUnique({
        where: { telegramId: ctx.from!.id },
    });

    if (!user || !user.apiKey) {
        await ctx.reply(ctx.t("no_api_key_message"));
```

```
        return;
    }

    try {
        const telegramFileResponse = await fetch(fileUrl);
        const buffer = await telegramFileResponse.arrayBuffer();

        const formData = new FormData();
        formData.append("keywords", keywords ?? "");

        formData.append(
            "file",
            new Blob([buffer], { type: "image/jpeg" }),
            file.file_unique_id
        );

        await fetch(process.env.SYNCPASTA_API_URL + "pasta", {
            method: "POST",
            headers: {
                "Authorization": "ApiKey " + (user.apiKey ?? "")
            },
            body: formData,
        });

        await ctx.reply(ctx.t("create_success"));
    } catch (e) {
        await ctx.reply(ctx.t("create_error"));
    }
});
```

Пошук записів реалізовано за допомогою вбудованих запитів (рис. 4.3). За допомогою них користувачі можуть викликати бота, ввівши його ім’я та запит у полі введення тексту в будь-якому чаті. Запит надсилається боту у вигляді оновлення. Таким чином, користувачі можуть запитувати контент у бота в будь-якому своєму чаті, групі чи каналі, взагалі не надсилаючи повідомлень [11]. Код реалізованого пошуку записів через вбудовані запити наведено у лістингу 4.4.

Лістинг 4.4. Пошук записів через вбудовані запити

```
import axios from "axios";
import {Composer} from "grammy";
```

```
import {MyContext} from "../types/MyContext";

export const composer = new Composer<MyContext>();

composer.inlineQuery(/.*/, async (ctx) => {
    const query = ctx.inlineQuery.query;
    const user = await ctx.db.user.findUnique({
        where: {telegramId: ctx.from!.id},
    });

    if (!user || !user.apiKey) {
        await ctx.answerInlineQuery([{
            title: ctx.t("no_api_key_title"),
            type: "article",
            id: "no_api_key",
            input_message_content: {
                message_text: ctx.t("no_api_key_message"),
            },
        }], {cache_time: 0});
        return;
    }

    try {
            const pastaRequest = await axios.get(process.env.SYNCPASTA_API_URL +
"pasta", {
            params: {
                query: query
            },
            headers: {
                "Authorization": "ApiKey " + (user.apiKey ?? "")
            }
        });

        const result = pastaRequest.data.data.map((item: any) => {
            if (item.file) {
                return {
                    type: "photo",
                    id: String(item.id),
                    title: item.keywords,
                    photo_url: item.file.url,
                    thumb_url: item.file.url,
                    description: item.text,
```

```
                    caption: item.text,
                }
            } else {
                return {
                    type: "article",
                    id: String(item.id),
                    title: item.keywords,
                    description: item.text,
                    input_message_content: {
                        message_text: item.text,
                    }
                }
            }
        });

        await ctx.answerInlineQuery(result, {cache_time: 0});
    } catch (e) {
        await ctx.answerInlineQuery([{
            title: ctx.t("error"),
            type: "article",
            id: "error",
            input_message_content: {
                message_text: ctx.t("error"),
            }
        }])
    }
});
```

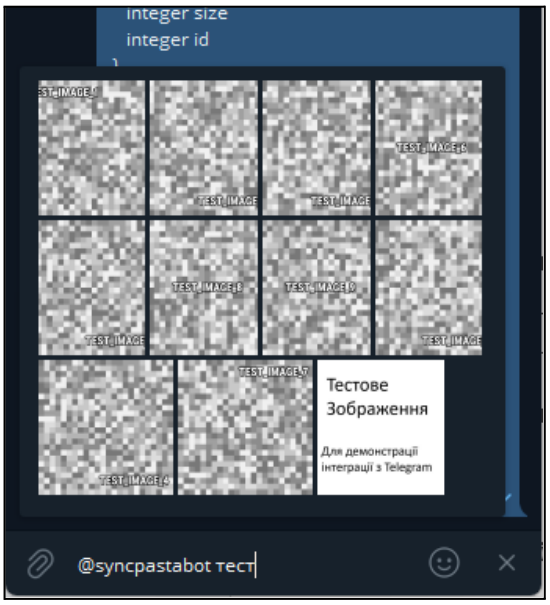

Рисунок 4.3. Пошук записів за допомогою вбудованих запитів

Розроблена інтеграція підтверджує можливість використання відкритого API для взаємодії із сторонніми платформами та демонструє один із можливих сценаріїв використання створюваного веб-сервісу.

4.3. Робота користувача зі створеним веб-сервісом

Розроблений веб-сервіс не потребує від користувача спеціальної підготовки для роботи із системою завдяки інтуїтивно зрозумілому інтерфейсу. Основними задачами користувача є створення та пошук мультимедійного контенту, а також керування інтеграціями із зовнішніми застосунками.

Для початку роботи користувач повинен пройти процедуру авторизації (рис. 3.7). Авторизація у системі реалізована за допомогою сервісу Firebase Authentication. Після успішної авторизації користувач отримує доступ до всіх функцій веб-сервісу та потрапляє на головну сторінку застосунку (рис. 3.6). Основним розділом є галерея мультимедійного контенту, у якій відображаються всі створені користувачем записи (рис. 3.8). Для кожного запису відображаються ключові слова, текстовий опис та, за наявності, прикріплений мультимедійний файл.

Для створення нового запису користувач натискає кнопку створення запису, що відкриває модальне вікно введення даних (рис. 4.4). У формі створення запису користувач повинен ввести текст, набір ключових слів та завантажити мультимедійний файл. Після натискання на “створити” інформація передається до серверу та зберігається у базі даних, а мультимедійний файл завантажується до хмарного сховища. Окрім створення нових записів користувач може редагувати та видаляти існуючі. Під час редагування доступна зміна текстового опису та ключових слів.

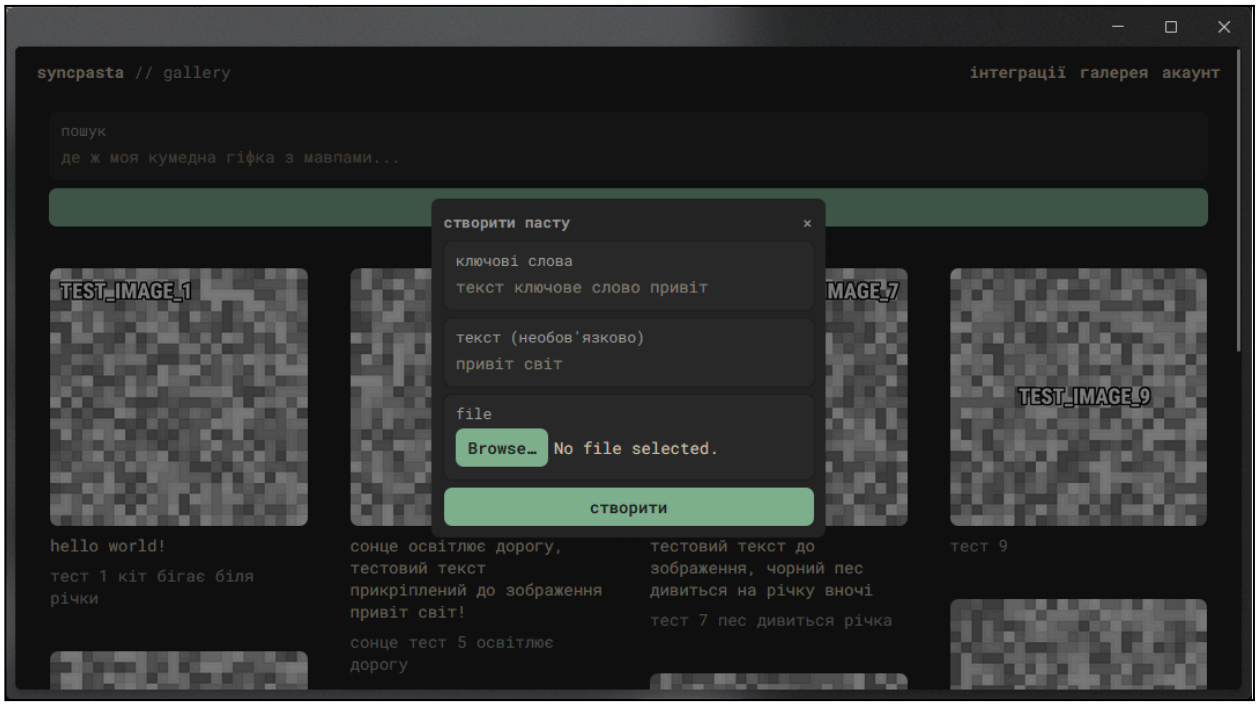

Рисунок 4.4. Модальне вікно створення запису

Для пошуку необхідних записів реалізовано пошук за ключовими словами. Користувач вводить пошуковий запит у відповідне поле, після чого система виконує запит до серверу та відображає відповідні записи. Такий підхід дозволяє швидко знаходити необхідний контент навіть при великій кількості збережених записів.

Ще однією важливою складовою веб-сервісу є підтримка сторонніх інтеграцій. У розділі керування інтеграціями користувач може створювати нові, переглядати та видаляти існуючі інтеграції, генерувати і оновлювати API-токени (рис. 3.9).

Після створення інтеграції користувач отримує API-токен, який використовуватиметься у сторонніх застосунках для доступу до функцій веб-сервісу через REST API. Такий механізм дозволяє інтегрувати систему із месенджерами, чат-ботами та іншими зовнішніми платформами.

Таким чином, реалізований інтерфейс користувача забезпечує зручну роботу із мультимедійним контентом, дозволяє швидко виконувати пошук необхідних записів та надає можливість інтеграції веб-сервісу із зовнішніми застосунками через відкритий API.

4.4. Встановлення та експериментальне застосування програми

Для забезпечення простого розгортання веб-сервісу було використано технологію контейнеризації Docker. Даний підхід дозволяє створити однакове середовище виконання незалежно від операційної системи користувача, завдяки чому мінімізувати кількість помилок, пов'язаних із налаштуванням програмного оточення.

Вихідний код проєкту зберігається у системі контролю версій Git та розміщується у репозиторії GitHub. Посилання на репозиторій та файл README з описом процесу встановлення і запуску наведено у додатку Б. Для автоматизації процесів перевірки та розгортання використовується підхід CI/CD (Continuous Integration / Continuous Delivery). Після створення нового коміту автоматично виконується перевірка вихідного коду та збірка проєкту (рис. 4.5).

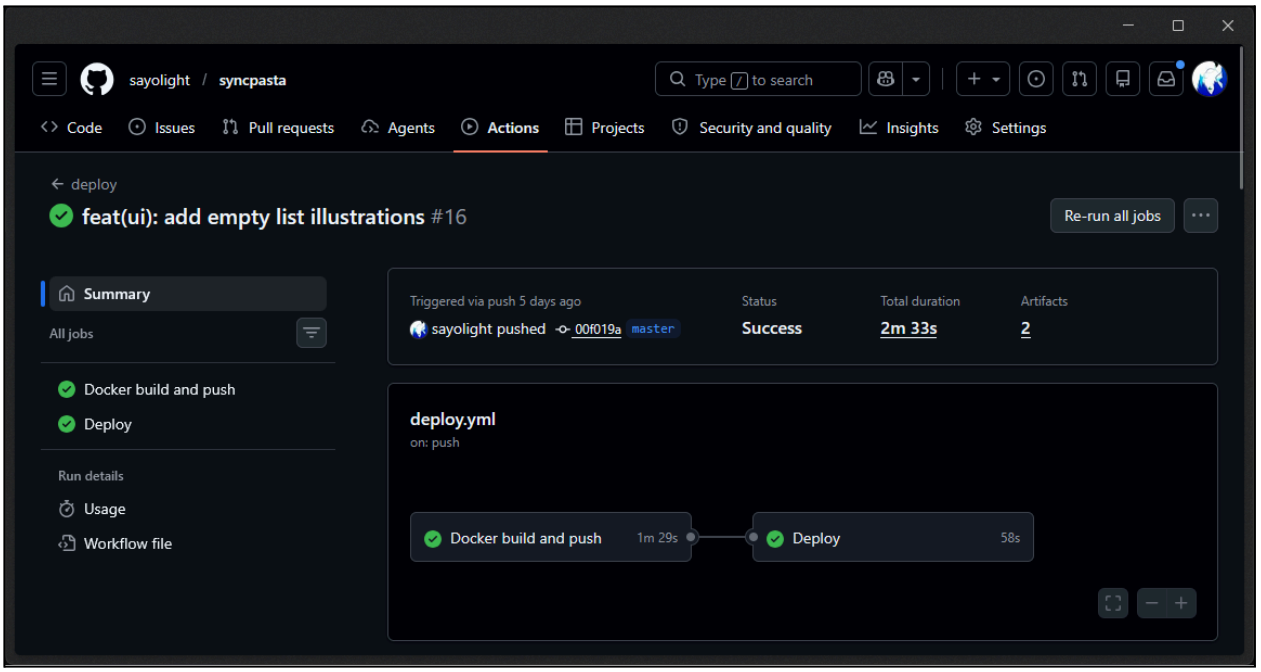

Рисунок 4.5. CI/CD процеси у Github Actions

Перед встановленням системи необхідно підготувати конфігураційний файл .env на основі шаблону .env.example (лістинг 4.5). У файлі .env вказуються параметри підключення до сервісів.

Лістинг 4.5. Шаблон конфігураційного файлу

```
APP_HOST=dev.syncpasta.online
APP_URL=https://dev.syncpasta.online
```

```
# Postgres
POSTGRES_HOST="db"
POSTGRES_PORT=5432
POSTGRES_DB="syncpasta_db"
POSTGRES_USER="syncpasta_admin"
POSTGRES_PASSWORD="syncpasta_password"

# Firebase
FIREBASE_TYPE=""
FIREBASE_PROJECT_ID=""
FIREBASE_PRIVATE_KEY_ID=""
FIREBASE_PRIVATE_KEY=""
FIREBASE_CLIENT_EMAIL=""
FIREBASE_CLIENT_ID=""
FIREBASE_AUTH_URI=""
FIREBASE_TOKEN_URI=""
FIREBASE_AUTH_PROVIDER_X509_CERT_URL=""
FIREBASE_CLIENT_X509_CERT_URL=""
FIREBASE_UNIVERSE_DOMAIN=""
FIREBASE_EMULATOR_HOST=""

# S3
S3_ENDPOINT="http://s3:9000"
S3_ACCESS_KEY_ID=""
S3_SECRET_ACCESS_KEY=""
S3_BUCKET_NAME="syncpasta"

# CLOUDFLARED
CLOUDFLARED_TOKEN=""
```

Після завершення налаштування запуск веб-сервісу здійснюється за допомогою команди docker compose up -d. Під час запуску автоматично створюються та запускаються контейнери клієнтської та серверної частини, бази даних PostgreSQL і Nginx. Після успішного запуску користувачу стає доступний веб-інтерфейс системи та програмний інтерфейс REST API.

Після встановлення було проведено експериментальне застосування веб-сервісу. Для перевірки працездатності системи було створено тестові облікові записи користувачів та набір мультимедійних записів різних типів. Під

час випробувань перевірено роботу механізмів авторизації, створення та редагування записів, пошуку за ключовими словами, роботи інтеграцій та журналу подій.

Окрім цього було проведено перевірку роботи відкритого API. Створено тестові інтеграції та виконано CRUD-запити до мультимедійного контенту. Для демонстрації можливостей відкритого API було реалізовано інтеграцію із месенджером Telegram. Telegram-бот використовує API веб-сервісу для завантаження та отримання мультимедійного контенту.

У результаті проведеного експериментального застосування бачимо, що програмний продукт стабільно функціонує в умовах реальної експлуатації. Використання Docker та CI/CD дозволяє значно спростити процес подальшого розгортання нових версій програмного забезпечення.

4.5. Висновки до розділу 4

У даному розділі було проведено тестування розробленого веб-сервісу та перевірено його відповідність поставленим вимогам. У ході тестування було виконано перевірку основних функціональних можливостей системи.

Також було описано процес встановлення та розгортання веб-додатку із використанням технології контейнеризації Docker. Використання автоматизованого процесу розгортання та засобів CI/CD дозволяє спростити розгортання системи та забезпечити стабільність її роботи.

Окрім цього було перевірено працездатність системи в наближених до реальної експлуатації умовах. Результати підтвердили коректну роботу серверної та клієнтської частин, а також можливість інтеграції зі сторонніми застосунками за допомогою відкритого API.

Отримані результати підтверджують, що розроблений програмний продукт відповідає визначеним вимогам, забезпечує централізоване збереження та пошук мультимедійного контенту, а також може використовуватися як основа для подальшого розвитку системи.

ВИСНОВКИ

Під час виконання роботи було розглянуто проблему складності організації та синхронізації мультимедійного контенту між різними месенджерами та платформами з можливістю інтеграції сторонніх застосунків. Аналіз предметної області показав, що існуючі рішення не забезпечують централізованого збереження мультимедійних матеріалів та мають обмежені можливості інтеграції.

У процесі виконання роботи було досліджено сучасні підходи до розробки веб-сервісів, проаналізовано існуючі програмні рішення та визначено основні функціональні й нефункціональні вимоги до створюваної системи. На основі проведеного аналізу було обрано стек технологій, що включає TypeScript, React, SCSS, NestJS, PostgreSQL, Docker, Git, Github Actions, Nginx, Firebase та Amazon Web Services S3.

У рамках роботи було спроєктовано архітектуру веб-сервісу, розроблено проєктні специфікації REST API та створено структуру бази даних. Особливу увагу приділено питанням масштабованості та можливості інтеграції зі сторонніми платформами. Для демонстрації можливостей відкритого API було розроблено інтеграцію із месенджером Telegram у вигляді чат-бота.

Реалізовано серверну частину із використанням фреймворку NestJS та клієнтську частину за допомогою React. Для збереження мультимедійних файлів використовується хмарне сховище AWS S3, а у якості бази даних обрано PostgreSQL.

Проведене тестування підтвердило коректність роботи системи. У ході експериментального застосування було перевірено механізми авторизації, створення та пошуку мультимедійного контенту, роботу інтеграцій та взаємодію через REST API.

Таким чином, поставлена мета була досягнута. Розроблений веб-сервіс може використовуватися як платформа для централізованого збереження

мультимедійного контенту та інтеграції із різними месенджерами, соціальними мережами та іншими платформами.

Повний листінг розробленної програми наявний у репозиторії GitHub за посиланням https://github.com/sayolight/syncpasta.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Conventional Commits. Політика комітів [Електронний ресурс]. - Режим доступу: https://www.conventionalcommits.org/uk/v1.0.0/ (дата звернення: 18.02.2026).
2. Amazon Simple Storage Service Documentation [Електронний ресурс]. - Режим доступу: https://docs.aws.amazon.com/s3/ (дата звернення: 19.02.2026).
3. Cherny B. Effective TypeScript: 62 Specific Ways to Improve Your TypeScript. - O'Reilly Media, 2019. - 250 p.
4. Docker Documentation [Електронний ресурс]. - Режим доступу: https://docs.docker.com/ (дата звернення: 20.02.2026).
5. Documentation. ESLint [Електронний ресурс]. - Режим доступу: https://eslint.org/docs/latest/ (дата звернення: 24.02.2026).
6. Firebase Admin SDK Documentation [Електронний ресурс]. - Режим доступу: https://firebase.google.com/docs/admin/setup (дата звернення: 23.02.2026).
7. Firebase Authentication Documentation [Електронний ресурс]. - Режим доступу: https://firebase.google.com/docs/auth (дата звернення: 23.02.2026).
8. Git Documentation [Електронний ресурс]. - Режим доступу: https://git-scm.com/docs (дата звернення: 17.02.2026).
9. GitHub Actions Documentation [Електронний ресурс]. - Режим доступу: https://docs.github.com/en/actions/ (дата звернення: 26.02.2026).
10. GraphQL Documentation [Електронний ресурс]. - Режим доступу: https://graphql.org/learn/ (дата звернення: 25.02.2026).
11. Inline Bots. Telegram [Електронний ресурс]. - Режим доступу: https://core.telegram.org/bots/inline (дата звернення: 27.02.2026).
12. Jest Documentation [Електронний ресурс]. - Режим доступу: https://jestjs.io/docs/getting-started (дата звернення: 26.02.2026).
13. JSON Web Token (JWT) Introduction [Електронний ресурс]. - Режим доступу: https://jwt.io/introduction (дата звернення: 24.02.2026).

14. Masse M. REST API Design Rulebook. - O'Reilly Media, 2011.
15. McGregor M. Full Stack JavaScript Strategies: The Hidden Parts Every Mid-Level Developer Needs to Know. - O'Reilly Media, 2025. - 755 p.
16. MDN Web Docs. HTTP Overview [Електронний ресурс]. - Режим доступу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (дата звернення: 25.02.2026).
17. Miceli M. Frontend Architecture for Design Systems: A Modern Blueprint for Scalable and Sustainable Websites. - O'Reilly Media, 2016. - 198 p.
18. NestJS Documentation [Електронний ресурс]. - Режим доступу: https://docs.nestjs.com/ (дата звернення: 16.02.2026).
19. Nginx Documentation [Електронний ресурс]. - Режим доступу: https://nginx.org/en/docs/ (дата звернення: 20.02.2026).
20. Node.js Documentation [Електронний ресурс]. - Режим доступу: https://nodejs.org/docs/latest/api/ (дата звернення: 24.02.2026).
21. Ponelat J., Rosenstock L. Designing APIs with Swagger and OpenAPI. - Manning Publications, 2022. - 416 p.
22. PostgreSQL Documentation [Електронний ресурс]. - Режим доступу: https://www.postgresql.org/docs/ (дата звернення: 18.02.2026).
23. React Documentation [Електронний ресурс]. - Режим доступу: https://react.dev/learn (дата звернення: 15.02.2026).
24. React - The Complete Guide (Includes Hooks, React Router, and Redux). Second Edition [Електронний ресурс]. - Режим доступу: https://www.oreilly.com/videos/react-the/9781801812603/ (дата звернення: 27.02.2026).
25. Sass Documentation [Електронний ресурс]. - Режим доступу: https://sass-lang.com/documentation/ (дата звернення: 19.02.2026).
26. Swagger Documentation [Електронний ресурс]. - Режим доступу: https://swagger.io/docs/ (дата звернення: 23.02.2026).

27. TanStack Query React Docs [Електронний ресурс]. - Режим доступу: https://tanstack.com/query/latest/docs/framework/react/overview (дата звернення: 17.02.2026).
28. Telegram Bot API [Електронний ресурс]. - Режим доступу: https://core.telegram.org/bots/api (дата звернення: 27.02.2026).
29. TypeORM Documentation [Електронний ресурс]. - Режим доступу: https://typeorm.io/docs/ (дата звернення: 21.02.2026).
30. TypeScript Documentation [Електронний ресурс]. - Режим доступу: https://www.typescriptlang.org/docs/ (дата звернення: 15.02.2026).
31. Windows Subsystem for Linux Documentation [Електронний ресурс]. - Режим доступу: https://learn.microsoft.com/en-us/windows/wsl/ (дата звернення: 20.02.2026).

ДОДАТОК А.

ТАБЛИЦЯ СПЕЦИФІКАЦІЙ REST API

№	Метод	Маршрут API	Опис
1	GET	/pasta	Отримання списку мультимедійних записів та пошук за ключовими словами за допомогою параметру query
2	POST	/pasta	Створення нового мультимедійного запису
3	PATCH	/pasta/{id}	Редагування ключових слів та тексту існуючого мультимедійного запису
4	DELETE	/pasta/{id}	Видалення мультимедійного запису
5	POST	/applications	Створення нової інтеграції
6	GET	/applications	Отримання списку інтеграцій користувача
7	POST	/applications/revoke	Видалення інтеграції
8	POST	/applications/reset	Отримати новий API-ключ інтеграції
9	GET	/applications/{id}/logs	Отримання журналу активності інтеграції

ДОДАТОК Б.

README ПРОЄКТУ ТА ПОСИЛАННЯ НА РЕПОЗИТОРІЙ

Посилання на репозиторій проєкту: https://github.com/sayolight/syncpasta

Syncpasta - Веб-сервіс для синхронізації мультимедійного контенту між платформами на основі відкритого API

Стек: TypeScript, NestJS, React, PostgreSQL, AWS S3, Firebase, Docker, Nginx, GitHub Actions.

Запуск

1. Клонуйте репозиторій: git clone https://github.com/sayolight/syncpasta.git
2. Скопіюйте файл .env.example у .env і заповніть його своїми даними.
3. Зробіть свій AWS S3 bucket відкритим для перегляду
4. Запуск за допомогою Docker: docker compose up -d

Конфігурація

- APP_HOST - доменне ім'я застосунку
- APP_URL - доменне ім'я застосунку із протоколом https
- POSTGRES_HOST - адреса сервера PostgreSQL
- POSTGRES_PORT - порт PostgreSQL
- POSTGRES_DB - назва бази даних
- POSTGRES_USER - користувач PostgreSQL
- POSTGRES_PASSWORD - пароль користувача
- FIREBASE_* - дані Firebase Service Account
- S3_ENDPOINT - ендпоїнт S3-сховища
- S3_ACCESS_KEY_ID - ключ доступу до сховища
- S3_SECRET_ACCESS_KEY - секретний ключ доступу до сховища
- S3_BUCKET_NAME - назва S3 bucket для збереження мультимедійного контенту
- CLOUDFLARED_TOKEN - API-токен Cloudflared Tunnel (при розробці)