ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І. І. МЕЧНИКОВА

(повне найменування закладу вищої освіти)

Факультет математики, фізики та інформаційних технологій

(повне найменування факультету)

Кафедра інформаційних технологій

(повна назва кафедри)

Кваліфікаційна робота

на здобуття ступеня вищої освіти «Бакалавр»

«Проєктування системи голосового введення з інтеграцією алгоритмів розпізнавання мовлення та обробки природної мови»

«Designing a voice input system with integration of speech recognition and natural language processing algorithms»

Виконав: здобувач денної форми навчання

спеціальності 122 Комп'ютерні науки .

(код, назва спеціальності)

Освітня програма Комп'ютерні науки .

(назва)

КОТОВЕНКО Микола Борисович

(прізвище, ім'я, по-батькові здобувача)

Керівник ФРАЗЕ-ФРАЗЕНКО О.О., к.т.н., доц. _____

(науковий ступінь, вчене звання, прізвище, ініціали) (підпис)

Рецензент ДОМАСКІН О.М. к.т.н., Начальник ІОЦ ОНЕУ

(науковий ступінь, вчене звання, прізвище, ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
Інформаційних технологій .

№ ___ від ___ _____ 2026 р.

Завідувачка кафедри

_____ КАЗАКОВА Надія

(підпис) (прізвище,ім'я)

Захищено на засіданні ЕК № 4
протокол № ___ від _____ 2026р.

Оцінка _____ / _____ / _____.

(за національною шкалою/шкалою ECTS/ бали)

Голова ЕК

_____ КОПИЧЕНКО Іван

(підпис) (прізвище,ім'я)

Одеса 2026

АНОТАЦІЯ

Тема кваліфікаційної роботи «Проєктування системи голосового введення з інтеграцією алгоритмів розпізнавання мовлення та обробки природної мови».

Актуальність кваліфікаційної роботи полягає в необхідності розробки інтелектуального сервісу для підприємств та організацій, у бізнес-процесах яких активно проводяться робочі наради та інтерв'ю та для яких впровадження штучного інтелекту для автоматизації робочих процесів стає ключовим фактором оптимізації часу, а також підвищення ефективності.

Об'єкт дослідження – це процеси проектування та розробки систем розпізнавання мовлення та інтелектуального аналізу отриманого результату за допомогою моделей штучного інтелекту.

Метою кваліфікаційної роботи є проєктування та розробка клієнт-серверного вебзастосунку, призначеного для автоматичного перетворення аудіоданих у текстовий формат та ведення контекстного діалогу з сучасними ШІ-моделями на основі отриманих даних.

Для досягнення поставленої мети в роботі були вирішені наступні питання: проведено аналіз існуючих технологій для розпізнавання й аналізу мовлення; сформульовано основні вимоги до інформаційної системи; обґрунтовано вибір архітектури та технологічних рішень; здійснена програмна реалізація вебзастосунку.

ABSTRACT

The topic of the qualification work is " Designing a voice input system with integration of speech recognition and natural language processing algorithms".

The relevance of the qualification work lies in the need to develop an intelligent service for enterprises and organizations whose business processes actively involve work meetings and interviews and for which the implementation of artificial intelligence to automate workflows becomes a key factor in time optimization as well as efficiency increase.

The object of the study is the processes of design and development of speech recognition systems and intelligent analysis of the obtained result using artificial intelligence models.

The purpose of the qualification work is the design and development of a client-server web application designed for the automatic conversion of audio data into text format and conducting a contextual dialogue with modern AI models based on the obtained data.

To achieve the set goal, the following issues were resolved in the work: an analysis of existing technologies for speech recognition and analysis was conducted; the main requirements for the information system were formulated; the choice of architecture and technological solutions was justified; the software implementation of the web application was carried out.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ,УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ..... 5

ВСТУП 6

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА МЕТОДІВ АВТОМАТИЗАЦІЇ 8

1.1 Аналіз існуючих аналогів та систем автоматичної транскрибації мовлення 8

1.2 Огляд сучасних методів ШІ для розпізнавання аудіо даних 13

1.3 Аналіз потреб користувачів та тенденцій розвитку сучасних методів розробки 16

2 ВИБІР ПРОГРАМНИХ ЗАСОБІВ 18

2.1 Обґрунтування технологічного стеку для розробки клієнтської та серверної частин..... 18

2.2 Аналіз та вибір моделі штучного інтелекту Groq API 22

3 ПРОЕКТУВАННЯ ВЕБЗАСТОСУНКУ 33

3.1 Вибір та обґрунтування архітектури вебзастосунку 33

3.2 Проектування архітектури бази даних..... 38

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ 41

4.1 Розробка серверної частини..... 41

4.2 Реалізація інтерфейсу та клієнтської логіки 50

4.3 Забезпечення інформаційної безпеки даних у вебзастосунку..... 60

ВИСНОВКИ..... 64

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..... 66

ПЕРЕЛІК СКОРОЧЕНЬ,УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

БД – База Даних

СУБД (СКБД) – Система (Управління) Керування Базами Даних

ШІ – Штучний Інтелект

API – Application Programming Interface

ASR – Automatic Speech Recognition

AWS – Amazon Web Services

CLI – Command Line Interface

CPU – Central Processing Unit

CRM – Customer Relationship Management

CSS – Cascading Style Sheets

CTC – Connectionist Temporal Classification

DOM – Document Object Model

E2E – End-to-End

GMM – Gaussian Mixture Model

GPU – Graphics Processing Unit

HMM – Hidden Markov Model

HTTP – HyperText Transfer Protocol

IoC – Inversion of Control

LLM – Large Language Model

LPU – Language Processing Unit

LSTM – Long Short-Term Memory

OWASP – Open Web Application Security Project

RNN – Recurrent Neural Network

RNN-T – Recurrent Neural Network Transducer

S3 – Simple Storage Service

SPA – Single Page Application

URL – Uniform Resource Locator

UUID – Universally Unique Identifier

ВСТУП

Сучасний розвиток програмного забезпечення пов’язаний із постійним зростанням обсягів цифрових даних та ускладненням вимог до способів їх обробки. Значну частину таких даних становить аудіоконтент, який формується під час робочих нарад, інтерв’ю, лекцій, конференцій, публічних виступів, телефонних консультацій та інших форм усної комунікації. У багатьох випадках цінність аудіозапису визначається не лише фактом його збереження, а й можливістю швидкого перетворення мовлення у структурований текст, придатний для пошуку, редагування, аналізу та подальшої інтеграції в інформаційні системи.

Ручна транскрибація аудіоматеріалів потребує значних витрат часу, залежить від кваліфікації виконавця та часто супроводжується помилками, пов’язаними з якістю запису, темпом мовлення, наявністю фонових шумів або використанням спеціалізованої термінології. Для підприємств, освітніх установ, медіаорганізацій та дослідницьких груп така модель обробки аудіоданих є недостатньо ефективною, особливо за умов регулярного накопичення великих обсягів мовленнєвої інформації. Через це актуальності набувають системи автоматичного розпізнавання мовлення, здатні скорочувати час обробки записів та забезпечувати користувача текстовим результатом у зручному для подальшої роботи форматі.

Розвиток методів штучного інтелекту суттєво розширив можливості автоматичної транскрибації. Сучасні моделі розпізнавання мовлення вже не обмежуються простим перетворенням звукового сигналу на текст. Вони можуть враховувати контекст, працювати з різними мовами, зберігати стійкість до шумів, формувати пунктуацію та готувати отриманий текст до подальшої інтелектуальної обробки. Поєднання автоматичного розпізнавання мовлення з методами обробки мови створює підґрунтя для побудови сервісів, які не лише фіксують зміст аудіозапису, а й дозволяють здійснювати контекстний діалог із системою на основі отриманої транскрипції.

Актуальність теми кваліфікаційної роботи визначається потребою у створенні вебзастосунку, який забезпечує автоматичне завантаження аудіофайлів, їх обробку за допомогою моделей штучного інтелекту, збереження результатів транскрибації та подальшу взаємодію користувача з текстовими даними. Такий програмний продукт може бути корисним для бізнес-аналітиків, менеджерів проєктів, журналістів, студентів, викладачів, дослідників та інших користувачів, діяльність яких пов’язана з регулярною обробкою аудіоматеріалів.

Об’єктом дослідження є процеси автоматичного розпізнавання мовлення та інтелектуального аналізу текстових даних, отриманих у результаті транскрибації аудіофайлів.

Предметом дослідження є методи, архітектурні рішення та програмні засоби розробки клієнт-серверного вебзастосунку для транскрибації аудіоданих і ведення контекстного діалогу на основі результатів розпізнавання мовлення.

Метою кваліфікаційної роботи є проєктування та розробка клієнт-серверного вебзастосунку, призначеного для автоматичного перетворення аудіоданих у текстовий формат, збереження результатів обробки та інтеграції з моделями штучного інтелекту для подальшого аналізу отриманого тексту.

У процесі виконання роботи використано методи системного аналізу, об’єктно-орієнтованого проєктування, клієнт-серверного моделювання, програмної реалізації вебзастосунків, роботи з реляційними базами даних, контейнеризації та інтеграції зовнішніх API штучного інтелекту.

Робота містить 67 сторінок, 29 рисунків, 12 джерел посилань

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА МЕТОДІВ АВТОМАТИЗАЦІЇ

Інтенсивний розвиток цифрових технологій призводить до стрімкого зростання обсягів неструктурованих медіаданих, вагому частину яких складає саме аудіоматеріали. Через це автоматизація переведення усного мовлення у текстовий формат є критично важливою науково-технічною задачею для систем різного спрямування: від інтелектуального документування та розшифровки інтерв'ю до автоматичного аналізу мовленнєвої аналітики у корпоративних CRM-системах. Традиційний підхід, який базується на ручній праці операторів-транскрибаторів, характеризується високою ресурсомісткістю, суттєвими витратами часу та ризиком виникнення суб'єктивних помилок, що зумовлює актуальність розробки й упровадження нових методів автоматизації на основі сучасних інформаційних технологій.

Сьогодні світовий ринок програмного забезпечення пропонує користувачам широкий вибір хмарних платформ та комерційних сервісів, які реалізують функціонал автоматичного розпізнавання мовлення [6]. З метою обґрунтування доцільності створення нового програмного рішення виникає необхідність проведення порівняльного аналізу найбільш поширених аналогів, визначення їхніх ключових переваг, технологічних обмежень та функціональних недоліків.

1.1 Аналіз існуючих аналогів та систем автоматичної транскрибації мовлення

1. Otter.ai (рис. 1.1) – це платформа, яка орієнтована на транскрибацію бізнес-нарад та лекцій у реальному часі *[6]*. Вона інтегрується з такими сервісами, як Zoom, Microsoft Teams та Google Meet.

Переваги:

– висока точність розпізнавання англійської мови;

– автоматичне розпізнавання різних спікерів;

– можливість спільного редагування тексту.

Недоліки:

– низька якість або повна відсутність підтримки інших мов, зокрема української;

– висока вартість комерційної підписки;

– закрита екосистема без гнучкого API для розробників.

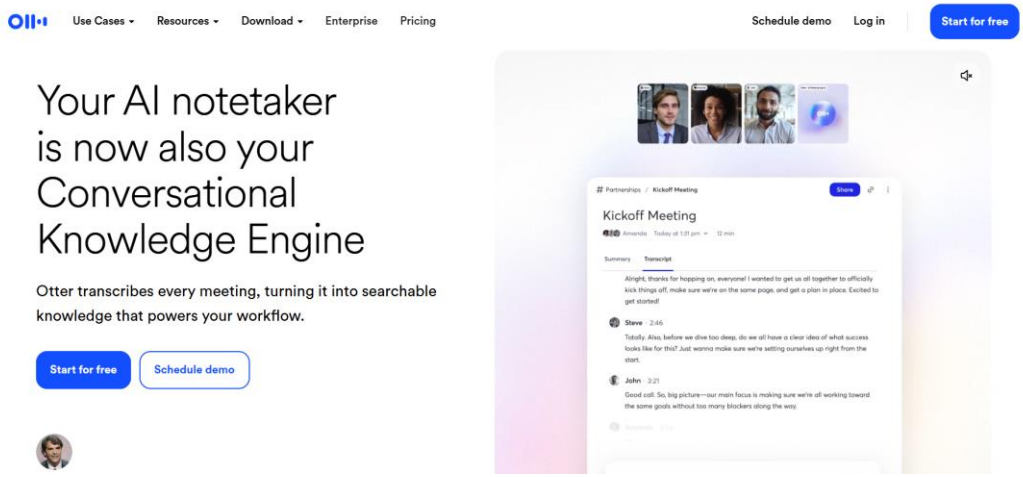


Рисунок 1.1 – Платформа Otter.ai

2. Google Cloud Speech-to-Text (рис. 1.2) – це потужний хмарний інструмент від компанії Google, який використовує глибокі нейронні мережі для розпізнавання мовлення [6].

Переваги:

– підтримка понад 125 мов та їхніх діалектів;

– висока стабільність хмарної інфраструктури;

– наявність розгалуженого API.

Недоліки:

– складна модель тарифікації;

– висока вартість при великих обсягах даних;

– затримка під час обробки довгих аудіофайлів в асинхронному режимі.

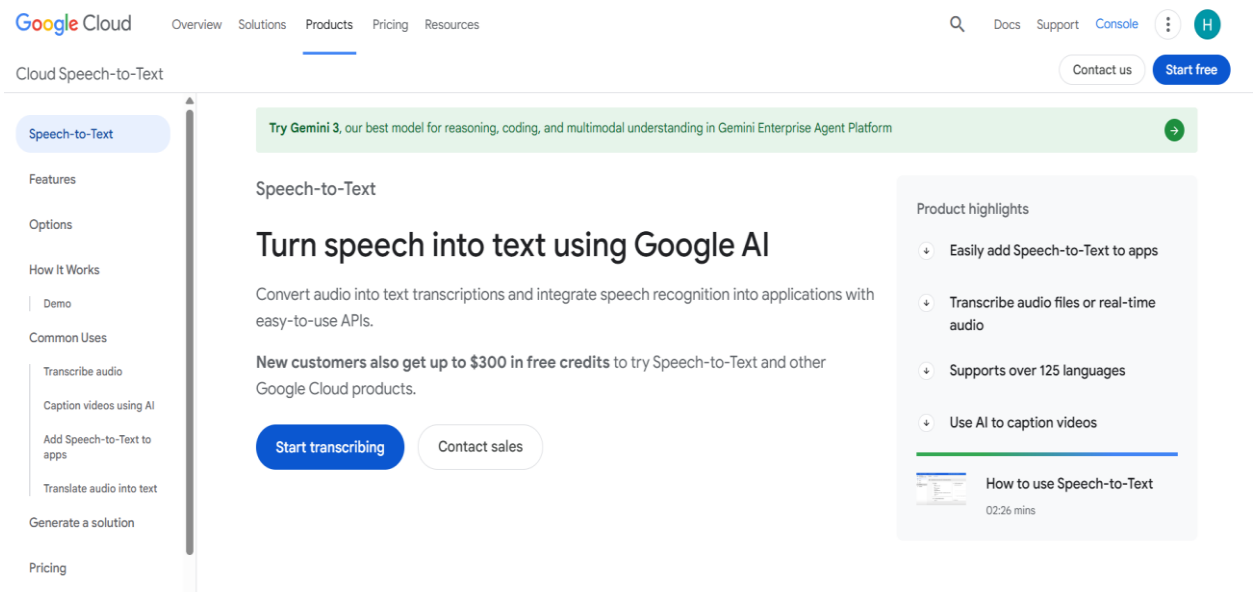


Рисунок 1.2 – Хмарний інструмент Google Cloud Speech-to-Text

3. OpenAI Whisper (рис. 1.3) – це сучасна модель розпізнавання мовлення від компанії OpenAI, яка навчена на величезному масиві різноманітних аудіоданих [8].

Переваги:

– чудова точність;
– стійкість до фонового шуму та акцентів;
– якісна пунктуація;
– гарне розпізнавання української мови.

Недоліки:

– обмеження на розмір файлу, що завантажується;
– відносно низька швидкість обробки на стандартних серверах OpenAI під час пікових навантажень;
– висока ціна за хвилину обробки.

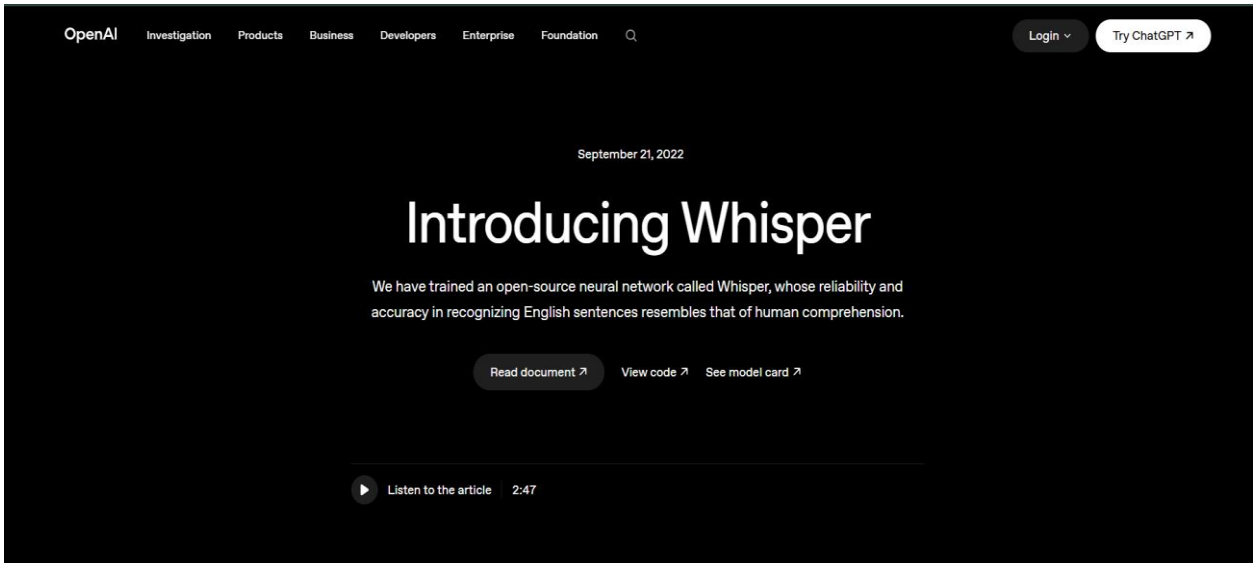


Рисунок 1.3 – Нативне хмарне API OpenAI Whisper

4. Microsoft Azure Speech Service (рис. 1.4) – це інтелектуальний сервіс розпізнавання мовлення, який входить до складу платформи Azure Cognitive Services. Він використовує сучасні алгоритми глибокого навчання та мовні моделі Microsoft [6].

Переваги:

- надзвичайно висока точність розпізнавання спеціалізованої термінології;
- можливість кастомізації мовних моделей під конкретну предметну область;
- відмінна підтримка української мови.

Недоліки:

- висока вартість використання;
- складна документація та панель керування інфраструктурою AWS/Azure;
- жорстка прив'язка до екосистеми Microsoft.

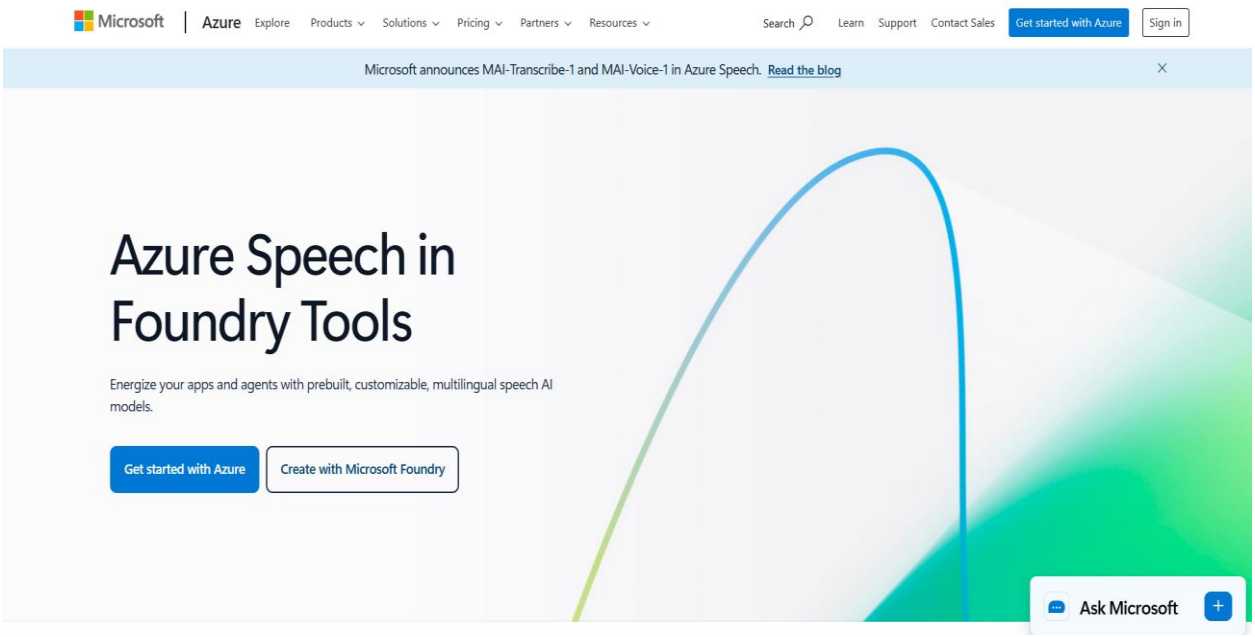


Рисунок 1.4 – Інтелектуальний сервіс Microsoft Azure Speech Service

5. Amazon Transcribe (рис. 1.5) – це спеціалізований сервіс від AWS, який використовує автоматичне розпізнавання мовлення для швидкого й точного перетворення аудіо в текст [6].

Переваги:

– безшовна інтеграція з хмарними сховищами Amazon S3;
– підтримка пакетної обробки великих масивів даних;
– функція автоматичного маскування конфіденційних даних.

Недоліки:

– оплата нараховується за кожну секунду обробленого аудіо за високими тарифами;
– якість розпізнавання української мови дещо поступається аналогам від Google та OpenAI;
– відсутність гнучкості при роботі поза інфраструктурою AWS.

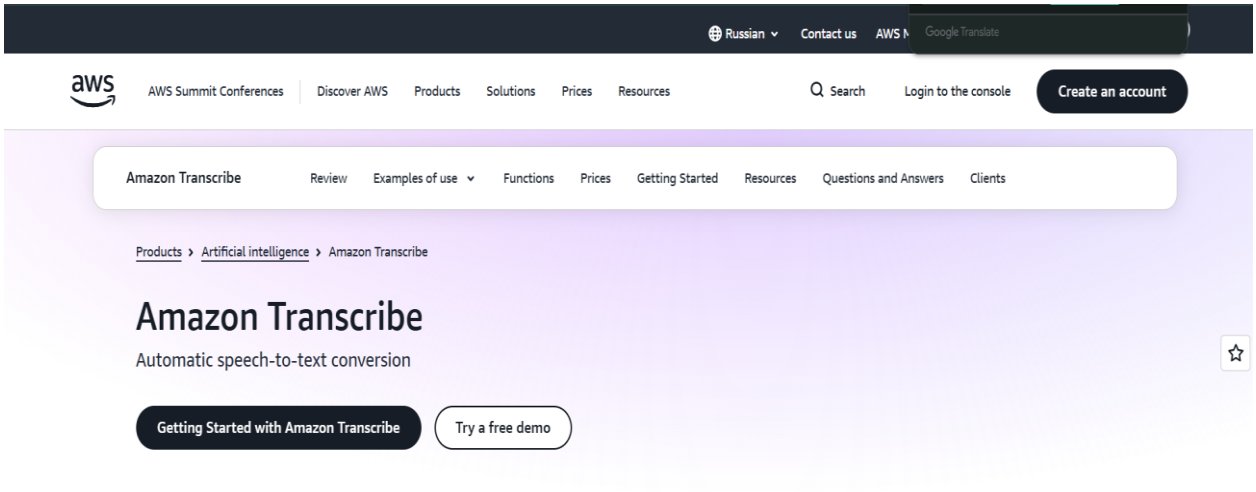


Рисунок 1.5 – Сервіс Amazon Transcribe

1.2 Огляд сучасних методів штучного інтелекту для розпізнавання аудіо даних

Технології розпізнавання мовлення сильно трансформувалися за останні роки. Вони розвилися від старих статистичних моделей, до сучасних систем, значну частину яких складає штучнbq інтелект. Якщо раніше комп'ютеру було потрібно поетапно аналізувати кожен окремий звук за допомогою складних математичних формул, то зараз архітектури здатні миттєво перетворювати цілісний аудіосигнал у готовий текст. Розуміння принципів роботи цих методів є необхідним кроком, який дозволить обрати ефективні інструменти штучного інтелекту для розробки вебзастосунку цієї кваліфікаційної роботи.

Перші успішні системи базувалися на так званих Прихованих Марковських Моделях (HMM) у поєднанні з акустичними моделями на основі сумішей Гаусових розподілів (GMM). Головне завдання такого підходу полягало в тому, щоб за допомогою теорії ймовірностей та формули Байєса вгадати, яка саме послідовність слів W найкраще відповідає записаному акустичному сигналу X:

$$\hat{W} = arg\ max_w\ P(X \mid W)\ P(W),$$

де

P(X | W) – акустична модель;

P(W) – мовна модель (N-грами).

Впровадження нейронних мереж радикально змінило принципи побудови таких сервісів, які в подальшому повністю витіснули класичні каскадні системи прогресивними наскрізними E2E рішеннями. Їхня ключова особливість полягає у тому, що вони здатні відображати вхідний акустичний сигнал в текстову послідовність символів або слів, пропускаючи етапи побудови проміжних марковських станів чи експліцитного вирівнювання фонем. Серед сучасних архітектурних підходів до побудови E2E-систем виділяють три основні групи методів:

1) Методи часово-пропорційної класифікації – це підхід, який дозволяє ефективно навчати рекурентні нейронні мережі (RNN, LSTM) на послідовностях даних, де часові мітки звуку не збігаються з межами літер у тексті. Для синхронізації вводиться спеціальний порожній токен, що дозволяє мережі пропускати кадри аудіо, які не несуть мовного навантаження. Моделі на базі CTC функціонують з високою обчислювальною швидкістю, але їх головним обмеженням є умовна незалежність вихідних символів – вони спираються на акустичні властивості сигналу і слабко враховують контекст [6].

2) Архітектури типу «RNN-Трансдуктор» (RNN-T). Дані системи складаються з двох паралельних обчислювальних гілок: акустичної мережі, що аналізує аудіосигнал, та передбачувальної мережі, яка функціонує за принципом мовної моделі та аналізує історію вже згенерованого тексту. Поєднання їхніх виходів у спільному інтеграційному блоці дозволяє використовувати RNN-T для потокової транскрибації в реальному часі, хоча такі системи є математично складними у оптимізації.

3) Моделі на основі архітектури Трансформер (Encoder-Decoder). Цей підхід є найбільш ефективним інструментом у інтелектуальному аналізі мовлення, який у подальшому і був обраний.

Сучасні системи розпізнавання мовлення, яскравим представником яких є архітектура OpenAI Whisper, використовують класичну трансформерну схему «енкодер-декодер», яка оптимізована для обробки спектрографічних зображень звуку. Процес обробки та перетворення інформації в межах цієї архітектури реалізується у декілька етапів.

На першому етапі вхідна аудіохвиля піддається дискретизації, розбивається на блоки та перетворюється з часового представлення у частотне за допомогою віконного швидкого перетворення Фур'є. У результаті цієї операції формується 80-канальної логарифмічної спектрограми за шкалою Мелів. Цей процес трансформує одновимірний звуковий сигнал у двовимірне матричне зображення. Воно відображає розподіл енергії частот мовлення у часі.

На другому етапі отримана спектрограма передається на вхід енкодера, де спочатку обробляється кількома згортковими шарами для виділення локальних контекстних ознак та зниження просторової розмірності. Далі дані проходять через каскад блоків самовізуальної уваги. Енкодер фіксує довгострокові акустичні залежності та формує векторні представлення прихованих станів звукового потоку [8].

На четвертому етапі декодер функціонує в авторегресійному режимі, передбачаючи текстові токени послідовно один за одним. Декодер зіставляє акустичні вектори, отримані від енкодера, із матрицею вже згенерованого тексту завдяки механізму перехресної уваги. Це дозволяє моделі одночасно спиратися як на фактичне звучання мови, так і на синтаксичні та граматичні закономірності мовної системи.

Головною перевагою моделей типу Whisper є спосіб їхнього навчання, який здійснюється на масивному наборі даних обсягом понад 680 тисяч годин аудіозаписів. Такі моделі демонструють досить високу стійкість до

специфічних акустичних умов: фонових шумів, ехо, наявності дефектів мовлення та навіть акцентів. Окрім того, архітектура здатна паралельно вирішувати задачі автоматичної генерації пунктуації, ідентифікації мови спікера та перекладу [8].

Але трансформерні архітектури всеж-таки мають суттєві обмеження у обчислюванні. Вони обумовлені математичною специфікою базового механізму уваги. Його обчислювальна складність зростає квадратично, відносно довжини вхідної послідовності токенів. При розгортанні відповідних моделей на машинах загального призначення, тривалість генерації відповіді під час обробки довгих аудіофайлів може суттєво зростати, що створює у подальшому високі затримки та значно зменшує пропускну здатність серверної частини вебзастосунків при щільних потоках запитів від користувачів.

1.3 Аналіз потреб користувачів та тенденцій розвитку сучасних методів розробки

Для успішного проєктування та впровадження сучасного вебзастосунку для автоматичної транскрибації потрібен комплексний підхід. Він являє собою глибокий аналіз запитів користувачів із врахуванням актуальних архітектурних тенденцій[1][2]. Оскільки ринок висуває високі вимоги до швидкості та надійності таких систем, розробка програмного забезпечення має спиратися на найбільш ефективні методології [3].

На основі дослідження предметної області та методів використання існуючих систем, були сформовані ключові вимоги користувачів до них:

- швидкість отримання результату;
- висока точність та автоматичне форматування;
- стабільність інтерфейсу при завантаженні великих файлів;
- конфіденційність даних.

Для відповідності визначеним вимогам у сучасній індустрії програмної

інженерії сформувалось декілька стійких рішень, які використовуються при проектуванні високонавантажених систем [3]:

1) Асинхронна архітектура. Перенесення важких обчислювальних процесів у фонові потоки. Використання розподілених систем черг дозволяє серверу миттєво відповідати користувачу, що його файл прийнято, а саму обробку виконувати ізольовано [9].

2) Концепція єдиного глобального стану на клієнті. При розробці інтерактивних користувацьких інтерфейсів (UI) стандартом є використання централізованих менеджерів стану на стороні клієнта. Це дозволяє уникнути зайвих запитів до сервера під час перемикання між сторінками додатка, забезпечуючи миттєвий відгук інтерфейсу [7].

3) Строга типізація та модульність коду. Мова програмування TypeScript на даний момент являється «золотим стандартом» як на клієнтській, так і на серверній сторонах. Це мінімізує кількість помилок на етапі компіляції, спрощує рефакторинг та забезпечує високу масштабованість і читабельність коду у великих проєктах [11].

4) Безпека за принципом "Security by Design". Сучасні стандарти вимагають впровадження захисту інформації на ранніх етапах проектування архітектури: використання криптографічного хешування конфіденційних даних, токенізація сесій для безпечного обміну даними між клієнтом і сервером та автоматична валідація і фільтрація будь-яких вхідних даних від користувача.

Таким чином, проведений аналіз потреб користувачів та тенденцій сучасних методів розробки визначає основні принципи побудови архітектури майбутнього вебзастосунку. Для реалізації швидкої та безпечної системи необхідно спроєктувати рішення, яке буде поєднувати у собі поєднанні компонентного клієнтського інтерфейсу, модульного серверного фреймворку з підтримкою фонових черг та хмарної інфраструктури для зберігання контенту [1][2].

2 ВИБІР ПРОГРАМНИХ ЗАСОБІВ

2.1 Обґрунтування технологічного стеку для розробки клієнтської та серверної частин

Ефективне функціонування, масштабованість та безпека вебзастосунку в першу чергу залежать від архітектурних рішень, які мають бути закладені ще на етапі проектування. Оскільки основною вимогою для сучасних систем є розділення зон відповідальності, розробка програмного забезпечення у рамках цієї кваліфікаційної роботи базується на концепції розподіленої клієнт-серверної архітектури. Ця концепція передбачає створення двох автономних модулів, які тісно між собою пов’язаних: серверної частини (Backend), яка буде відповідати за бізнес-логіку, збереження даних та комунікацію з моделями штучного інтелекту, та клієнтської частини (Frontend), яка забезпечує взаємодію з користувачем та візуалізацію процесів [1][2].

У кваліфікаційній роботі детально розглянуто функціональне призначення та переваги кожної з обраних технологій.

Node.js – це кросплатформне програмне середовище виконання з відкритим вихідним кодом. Воно дозволяє виконувати код мовою JavaScript на стороні сервера за допомогою продуктивного двигуна V8. Це програмне мередовище було обрано як фундамент серверної частини через її архітектурну специфіку, яка орієнтована на побудову швидких і масштабованих мережевих застосунків [12].

Переваги:

– Неблокуюче введення-виведення: Завдяки використанню механізму Event Loop, платформа здатна обробляти тисячі одночасних з'єднань в одному потоці без створення додаткових системних процесів, що мінімізує витрати оперативної пам'яті сервера.

– Висока пропускна здатність: Оптимальна архітектура для потокової передачі важких медіафайлів у реальному часі.

– Величезний реєстр npm пакетів.

Nest.js – це прогресивний фреймворк для побудови надійних та масштабованих серверних застосунків на базі Node.js, який за замовчуванням використовує мову TypeScript та підтримує модульну архітектуру. Nest.js надає розробнику сувору архітектурну структуру, засновану на найкращих практиках об'єктно-орієнтованого програмування та системного проектування [2][12].

Переваги:

– Впровадження залежностей: Потужний вбудований IoC-контейнер автоматизує керування життєвим циклом класів, спрощує тестування модуля транскрибації та ізоляцію бізнес-логіки від зовнішніх API.

– Архітектурна уніфікація: Поділ коду на модулі, контролери та сервіси запобігає появі хаотичної структури та забезпечує легке розширення проєкту при додаванні нових ШІ-моделей.

– Модульна архітектура: Легке масштабування та ізоляція функціональних блоків застосунку за принципами SOLID.

TypeScript – це строго типізована мова програмування, що є надбудовою над JavaScript та компілюється в чистий JavaScript для виконання у будь-якому сумісному середовищі. Впровадження цієї мови на серверній стороні дозволяє перенести виявлення помилок з етапу виконання програми на етап її написання.

Переваги:

– Статична типізація: Виявлення та усунення архітектурних помилок ще на етапі написання коду.

– Самодокументація коду: Чіткий опис інтерфейсів та типів даних, що спрощує читання коду іншими розробниками.

– Безпечний ре факторинг.

PostgreSQL – це об'єктно-реляційна система керування базами даних з відкритим вихідним кодом, що володіє високою надійністю, архітектурною

стійкістю та строгою відповідністю стандартам ACID. Вона обрана як основне сховище для структурованої інформації застосунку [3].

Переваги:

– Транзакційна цілісність: Гарантує, що операції створення користувачів, приєднання токенів авторизації та фіксації історії транскрибацій будуть виконані абсолютно надійно.

– Підтримка складних реляційних зв'язків.

Amazon S3 – це хмарне сховище, спроєктоване для безпечного, масштабованого та стійкого зберігання будь-яких обсягів неструктурованих цифрових даних. Зберігання медіафайлів на сервері розробника є критичною помилкою проєктування, тому в проєкті застосовано хмарний S3-контейнер [4].

Переваги:

– Практично безмежне розширення: Сховище автоматично масштабується під будь-які об'єми завантажених аудіотреків користувачів.

– Безпека через Pre-signed URLs: Бекенд може генерувати тимчасові безпечні посилання для прямого завантаження або скачування файлів із хмари, захищаючи приватний периметр даних.

– Об'єктна архітектура: Надійне та ізольоване збереження неструктурованих медіафайлів будь-якого об'єму.

– Проста інтеграція через API: Стандартизований REST API протокол, що підтримується будь-якими хмарними провайдерами та локальним Docker-контейнером MinIO.

MinIO – це легке хмарне середовище з відкритим вихідним кодом, яке повністю сумісне на рівні API з Amazon S3. На відміну від традиційних файлових систем або реляційних баз даних, MinIO організує збереження даних у вигляді пласкої структури ізольованих контейнерів, де кожен файл є унікальним об'єктом, який має свій ідентифікатор (ID) та повний пакет метаданих. Це дозволяє відокремити сервер від рутинних операцій читання-

запису на жорсткий диск і забезпечує лінійну швидкість доступу до медіапотоків.

Переваги:

– Повна сумісність з Amazon S3 API.

– Велика пропускна здатність забезпечує швидкість читання й запису даних на рівні десятків гігабайт за секунду.

– Наявність генерації тимчасових криптографічних посилань, які дозволяють клієнтському інтерфейсу безпечно завантажувати важкі аудіофайли в сховище напряму, минаючи і розвантажуючи HTTP-потік сервера.

– Можливість безкоштовного локального розгортання повноцінної інфраструктури зберігання даних на власних серверах через Docker-контейнери.

– Високий рівень безпеки та ізоляції.

React – це відкрита JavaScript-бібліотека для створення динамічних та інтерактивних інтерфейсів користувача в односторінкових вебзастосунках (SPA). Проєктований сервіс вимагає високої реактивності інтерфейсу [11].

Переваги:

– Віртуальне дерево (Virtual DOM): Оптимізує процес оновлення вебсторінки, мінімізуючи прямі маніпуляції з реальним деревом браузера, що гарантує високу швидкість рендерингу великих масивів розпізнаного тексту.

– Компонентна архітектура: Дозволяє розділити інтерфейс на незалежні блоки, що спрощує підтримку фронтенду.

Redux Toolkit – це менеджер стану, який стандартизує та спрощує керування глобальним станом застосунку [4].

Переваги:

– Єдине джерело істини: Весь стан додатка зберігається в централізованому сховищі, що виключає розсинхронізацію даних між різними візуальними панелями клієнта.

– Вбудований модуль RTK Query: Автоматизує процеси асинхронного

кешування, надсилання REST-запитів до бекенду та автоматичного оновлення інтерфейсу при зміні статусів обробки файлів.

– Потужні інструменти розробника: Підтримка Redux DevTools для покрокового відстеження та дебагу змін стану в часі.

Tailwind CSS – це утилітарний CSS-фреймворк, який дозволяє проектувати сучасні інтерфейси безпосередньо всередині коду компонентів за допомогою набору готових низькорівневих класів.

Переваги:

– Максимальна швидкість стилізації: Усуває потребу створювати окремі важкі файли стилів та вигадувати унікальні імена класів, що прискорює верстку UI.

– Висока продуктивність збірки: На етапі компіляції фреймворк видаляє весь невикористаний CSS-код, генеруючи мінімальний за розміром фінальний файл стилів, що оптимізує швидкість завантаження вебсайту у браузері користувача.

Shadcn UI – це нова бібліотека гнучких UI-елементів із відкритим кодом. Суть архітектурного підходу Shadcn UI полягає в тому, що розробник не встановлює бібліотеку як залежність, а копіює вихідний код необхідного компонента прямо у структуру свого проєкту за допомогою утиліти CLI. Це може бути наприклад, кнопки, модальні вікна чи форми завантаження.

Переваги:

- Повний контроль над вихідним кодом.
- Глибока інтеграція з Tailwind CSS.

2.2 Аналіз та вибір моделі штучного інтелекту Groq API

Ефективність функціонування модуля транскрибації залежить в першу чергу від обчислювальних характеристик та особливостей архітектури обраної моделі штучного інтелекту. Загальноприйняті підходи до розгортання важких ШІ-моделей (ASR) на власних серверах, які зазвичай є на базі CPU або

стандартних графічних прискорювачів GPU, супроводжуються високими фінансовими витратами на інфраструктуру та критичними затримками під час обробки аудіоданих. Для вирішення цієї проблеми у даній кваліфікаційній роботі роз’яснено основні причини використання хмарного інструментарію Groq API, який пропонує оптимізований доступ до передових моделей штучного інтелекту [6].

Головною перевагою платформи Groq є використання принципово нового типу обчислювальних процесорів – Language Processing Unit (LPU). Він відрізняється від класичних архітектур загального призначення (CPU) та графічних прискорювачів (GPU).

Традиційні GPU орієнтовані більш всього на масивне паралельне виконання графічних інструкцій. Вони вимагають складних механізмів динамічного керування потоками, синхронізації ядер та постійного звернення до настільки ж важкої зовнішньої пам'яті. Під час запуску послідовних моделей штучного інтелекту, GPU витрачають доволі велику частину часу не на самі математичні обчислення, а на звичайне очікування передачі даних між пам'яттю та обчислювальними блоками.

Архітектура LPU від компанії Groq спроєктована під завдання тензорної та лінійної алгебри, які лежать в основі глибоких нейромереж. Архітектура LPU має дві ключові особливості:

1. Усі потоки даних та послідовність виконання інструкцій нейромережі прораховуються на етапі компіляції коду.
2. Дані переміщуються між обчислювальними елементами за допомогою надшвидкої внутрішньої пам'яті SRAM, що забезпечує досить велику пропускну здатність та мінімізує виникнення затримок синхронізації.

Завдяки цьому Groq API (рис. 2.1) дозволяє досягти швидкості генерації токенів, яка у рази перевищує показники у стандартних рішень на базі GPU. Ця особливість є критично важливою для забезпечення миттєвого відгуку вебзастосунку [8].

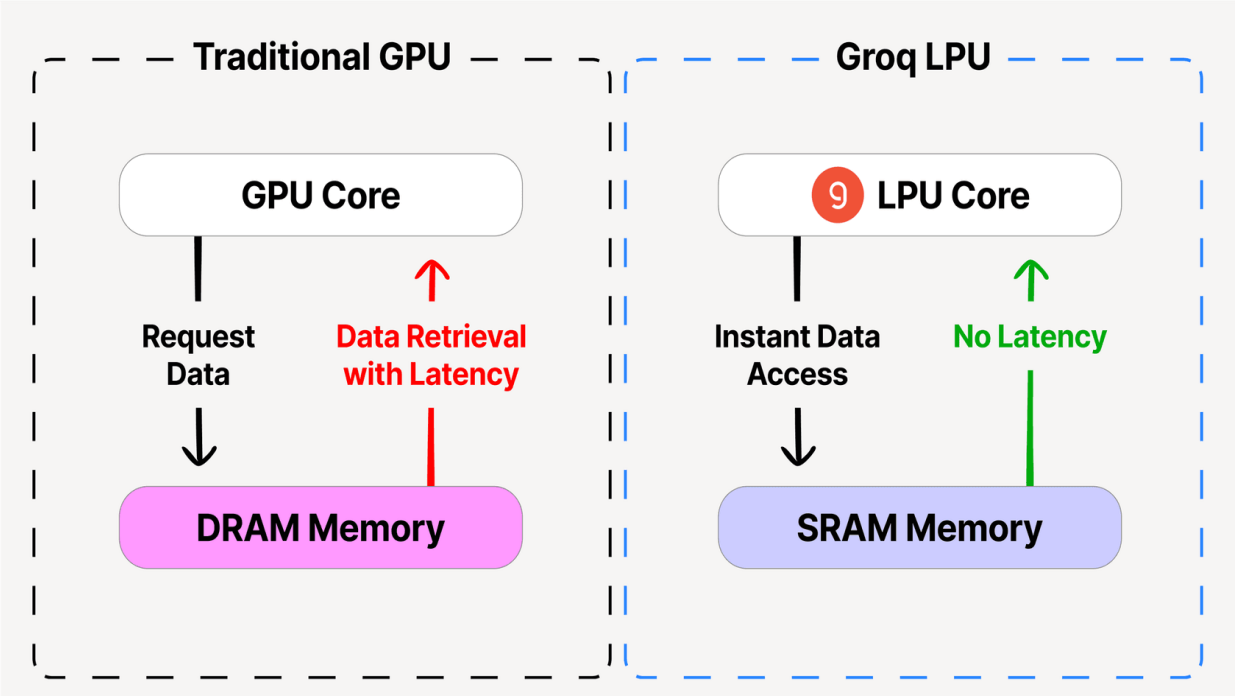


Рисунок 2.1 – Порівнювальна схема GPU та LPU

Платформа Groq API надає доступ до кількох оптимізованих версій моделей розпізнавання мовлення, що базуються на архітектурі Whisper від компанії OpenAI. На основі проведеного аналізу для програмної реалізації вебзастосунку було обрано модель whisper-large-v3-turbo. Її основні переваги – це її висока швидкість обробки та точность, ефективне масштабування і стійкість до акустичних завад.

Модель whisper-large-v3-turb, в умовах апаратного прискорення, здатна виконати повну транскрибацію, розстановку розділових знаків та розбивку на таймкоди для 10-хвилинного аудіофайлу менш ніж за 2–3 секунди. Це дозволяє задовольнити вимоги користувачів щодо низької затримки, а також забезпечує високу пропускну здатність серверної частини. Таким чином, інтеграція цієї моделі у систему є найбільш виправданим кроком для реалізації інструменту автоматизації транскрибації.

2.3 Середовище розробки та допоміжні інструменти

Visual Studio Code (VS Code) (рис. 2.2) – це багатофункціональний, кросплатформний редактор коду, який був створений компанією Microsoft для розробки сучасних вебзастосунків. Цей інструмент, як основне середовище розробки для клієнтської та серверної частин проекту, був обраний через його високу продуктивність, гнучкість налаштування за допомогою плагінів та через потужну вбудовану підтримку технологій, на яких базується вебзастосунок. На відміну від інших інтегрованих середовищ розробки, цей редактор забезпечує високу швидкість запуску та економне використання оперативної пам'яті, зберігаючи при цьому всі можливості повноцінної IDE завдяки великій системі плагінів [12].

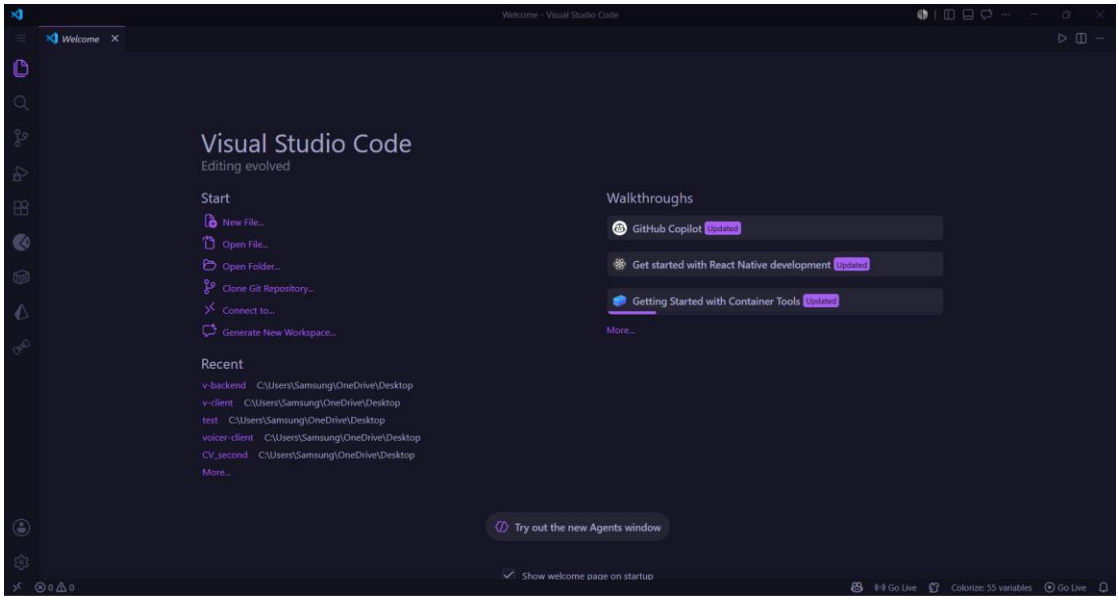


Рисунок 2.2 – Редактор коду Visual Studio Code

У процесі розробки середовище було розширено інструментами автоматизації якості коду. Зокрема, інтеграція лінтера ESLint дозволяє в автоматичному режимі перевіряти написаний код на відповідність сучасним стандартам його написання, виявляти потенційні вразливості та застарілі

конструкції. Одночасно з цим був інтегрован плагін Prettier (лістинг 2.1, лістинг 2.2), який забезпечує уніфіковане форматування синтаксису при кожному збереженні файлу. Цей плагін гарантує підтримку єдиного стилю оформлення та підвищує загальну читабельність коду.

```
{
       "trailingComma": "none",
       "useTabs": true,
       "semi": false,
       "tabWidth": 4,
       "jsxSingleQuote": true,
       "singleQuote": true,
       "importOrder": [
              "<THIRD_PARTY_MODULES>", "^@/components/(.*)$",
              "^@/hooks/(.*)$", "^@/types/(.*)$",
              "^@/services/(.*)$", "^@/utils/(.*)$",
          "^@/store/(.*)$", "^../(.*)$", "^./(.*)$"
       ],
       "importOrderSeparation": true,
       "importOrderSortSpecifiers": true,
       "plugins": ["@trivago/prettier-plugin-sort-imports"]
}
```

Лістинг 2.1 – Лістинг стилістичного оформлення клієнтської частини

```
    {
  "trailingComma": "none",
  "tabWidth": 4,
  "singleQuote": true,
  "arrowParens": "avoid"
}
```

Лістинг 2.2 – Лістинг програми Prettier для стилістичного оформлення серверної частини

Важливе значення для підвищення швидкості розробки має вбудована підтримка системи контролю версій Git. Вона дозволяє візуалізувати зміни в коді, керувати гілками, зберігати версії проекту та синхронізувати локальний репозиторій із хмарним сервісом GitHub (рис. 2.3) у самому інтерфейсі

редактора. У результаті, вона дозволяє розробнику керувати декількома частинами проекту одночасно [12].

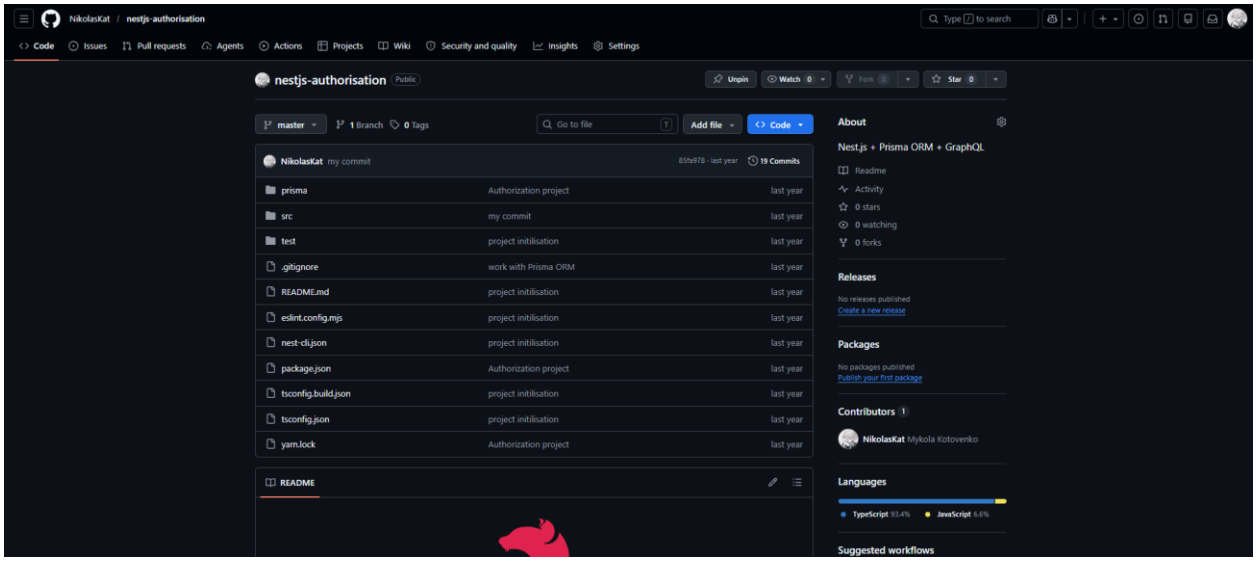


Рисунок 2.3 – Хмарний сервіс GitHub

Додатковою перевагою використання Visual Studio Code є можливість інтеграції зручних інструментів тестування у редактор за допомогою розширення Postman для VS Code (рис. 2.4) [5]. Цей плагін дозволяє розробнику проектувати, відправляти та аналізувати HTTP-запити до серверної частини без необхідності перемикання між різними вікнами операційної системи та запуску окремого застосунку. Інтерфейс Postman забезпечує швидкий доступ до тестування маршрутів автентифікації, перевірки валідації вхідних даних та контролю завантаження аудіофайлів, зберігаючи при цьому всю історію запитів в єдиній локальній екосистемі [5]. Це суттєво прискорює цикл налагодження та підвищує загальну якість написання бекенду.

Git – це зручна система контролю версій, яка призначена для відстеження змін у вихідному коді, координації роботи над проєктом та збереження повної історії його розробки. Використання цього інструменту в

поєднанні з хмарною платформою GitHub є основою для забезпечення надійності, безпеки та прозорості всього життєвого циклу проектування.

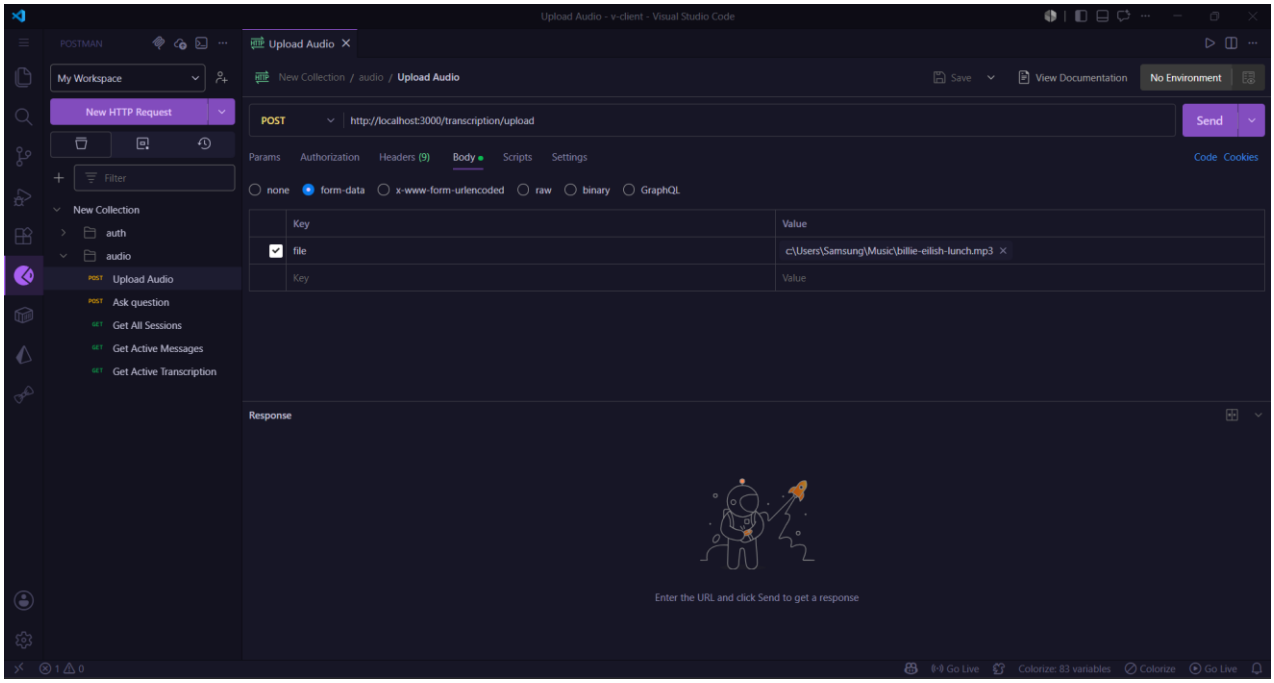


Рисунок 2.4 – Розширення Postman для VS Code

Git дозволяє фіксувати кожен етап написання коду у вигляді контрольних точок, що гарантує можливість миттєвого аналізу внесених змін або безболісного відкату до попередньої стабільної версії у разі виникнення критичних помилок.

В свою чергу хмарний сервіс GitHub у цьому процесі виконує роль централізованого віддаленого сховища, який забезпечує захист проєкту від втрати внаслідок апаратних збоїв на локальному комп'ютері.

Yarn – це сучасний менеджер пакетів для Node.js, розроблений компанією Facebook для автоматизації встановлення, оновлення та керування зовнішніми залежностями програмного продукту. Впровадження Yarn замість стандартного менеджера пакетів npm є необхідним кроком оптимізації швидкості збірки та гарантування стабільності роботи екосистеми вебзастосунку на різних етапах розгортання [12].

Docker (рис. 2.5) – це відкрита інфраструктурна платформа, яка дозволяє автоматизувати процеси розгортання, масштабування та керування застосунками шляхом їх ізоляції на рівні операційної системи у вигляді контейнерів. Використання цієї технології в поєднанні з утилітою Docker Compose є важливим рішенням для забезпечення стабільного функціонування локального середовища розробки та спрощення подальшого релізу застосунку. На відміну від класичних технологій віртуалізації, які вимагають запуску повноцінної операційної системи для кожного сервісу, контейнери Docker використовують єдине ядро хостової ОС, що гарантує мінімальні витрати на обчислювальні потужності процесора та оперативну пам'ять [3].

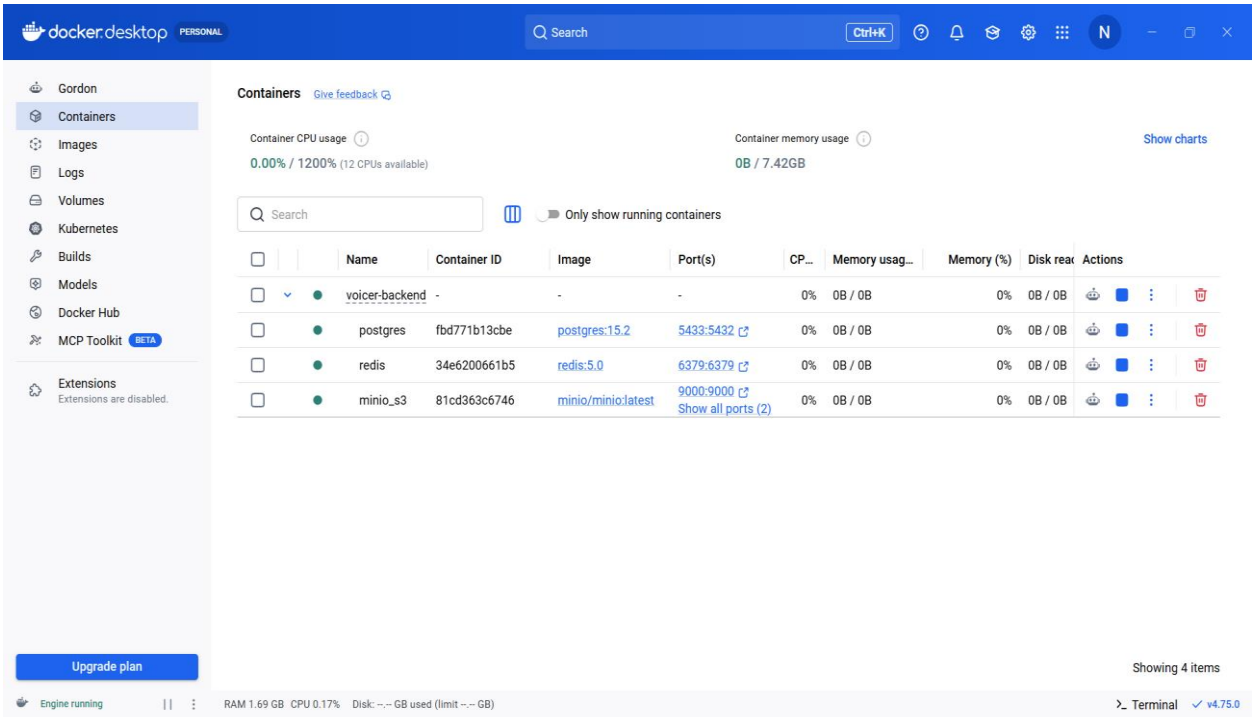


Рисунок 2.5 – Графічний додаток Docker Desktop

У рамках архітектури даного проекту, реляційна база даних PostgreSQL та MinIO були повністю розгорнуті та налаштовані за допомогою docker-compose.yml (лістинг 2.3). Написання цього маленького файлу дозволяє розробнику однією командою підняти весь необхідний стек ізольованих

сервісів із чітко заданими мережевими інтерфейсами, налаштованими портами та прив'язаними змінними оточення.

```
version: '3.7'
services:
  db:
    container_name: postgres
    image: postgres:15.2
    restart: always
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
    ports:
      - 5433:5432
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - backend
volumes:
  postgres_data:
  minio_data:
networks:
  backend:
```

Лістинг 2.3 – Лістинг програми для конфігурації Docker контейнерів

Спроєктовані Docker-контейнери виступають універсальними та ізольованими модулями, які гарантують стабільність та передбачуваність поведінки вебзастосунку як на етапі локального написання коду, так і при фінальному продакшн-деплої у хмарну інфраструктуру.

Також однією з задач розробки складної системи є організація надійного та ізольованого сховища для завантажуваних користувачами аудіофайлів. Традиційний підхід із збереженням файлів у реляційній базі даних є доволі неефективним. Такий метод зберігання призводить до стрімкого розростання розміру БД, уповільнення створення резервних копій та зниження швидкості виконання SQL-запитів. Для усунення цієї проблеми в архітектуру системи було впроваджено MinIO (рис. 2.6).

MinIO розгорнуто в ізольованому Docker-контейнері, що дозволяє емулювати поведінку хмарного сховища AWS S3 у локальному середовищі

розробки. Замість фізичних файлів, у базі даних PostgreSQL фіксується лише унікальний рядок-ідентифікатор (s3Key), який відображає шлях до об'єкта всередині сховища.

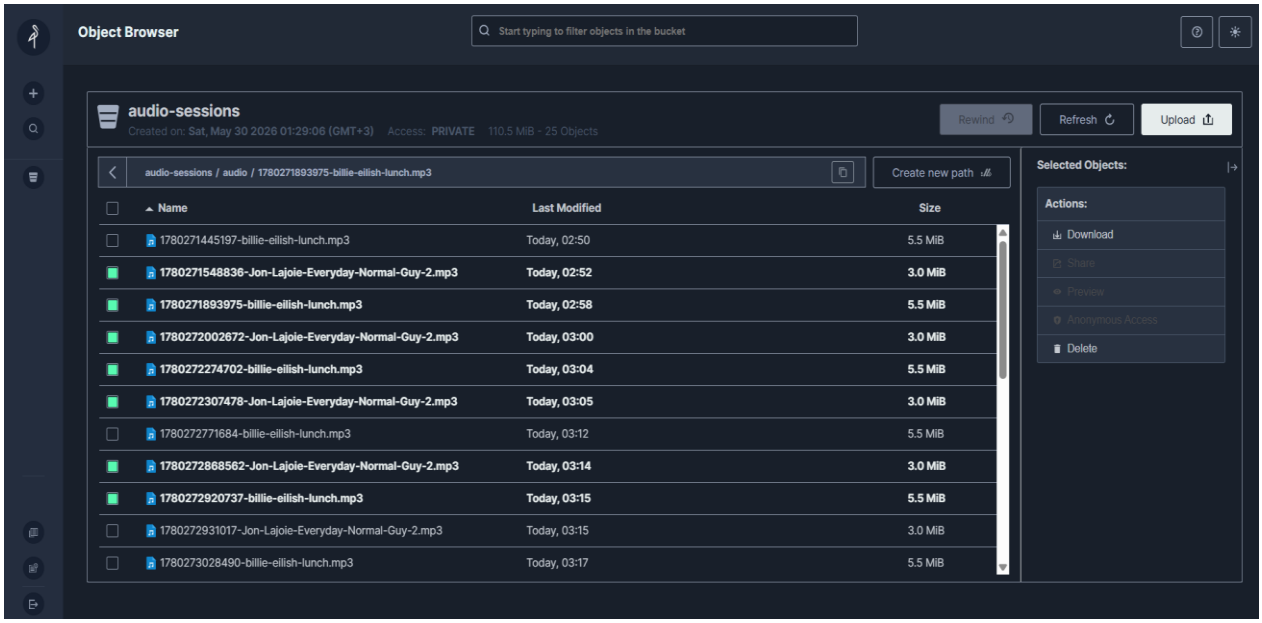


Рисунок 2.6 – Локальне розгортання MinIO

Також досить важливим пунктом, який облегшує розробку в рази, є візуальний контроль за станом даних таблиць. Використання виключно інтерфейсу командного рядка – досить незручний варіант. Для вирішення цих задач та забезпечення ефективного супроводу бази даних PostgreSQL було обрано Beekeeper Studio (рис. 2.7). Цей інструмент є сучасним, кросплатформним графічним інтерфейсом та SQL-редактором, який побудовано на базі сучасних вебтехнологій. Він гарантує високу швидкість роботи та інтуїтивно зрозумілий UI. Використовується він для щоденних завдань, таких як написання запитів, редагування даних у реальному часі та управління схемами таблиць без створення зайвого інфраструктурного навантаження на робочу станцію [3].

Безпека з'єднань та ізоляція середовищ реалізовані завдяки вбудованій підтримці шифрування SSL та можливості створення захищених тунелів SSH.

Це дозволяє безпечно підключатися як до локального PostgreSQL у Docker-контейнері, так і до віддалених серверів на етапі промислового розгортання системи.

Нарешті, наявність табличного редактора даних у реальному часі дозволяє швидко додавати, редагувати або видалити рядки через візуальну сітку інтерфейсу без необхідності писати команди вручну, що ідеально підходить для швидкого створення тестових користувачів чи зміни статусів асинхронних завдань під час тестування системи.

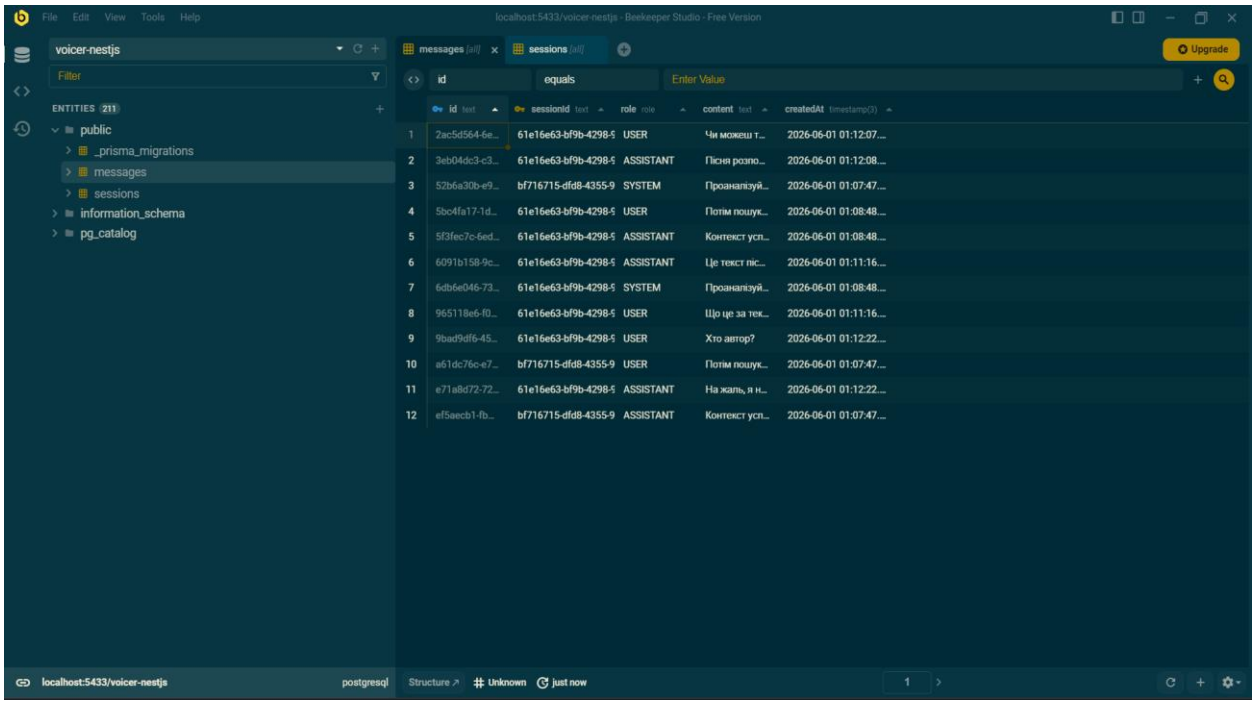


Рисунок 2.7 – Графічний інтерфейс Beekeeper Studio

3 ПРОЕКТУВАННЯ ВЕБЗАСТОСУНКУ

3.1 Вибір та обґрунтування архітектури вебзастосунку

Проєктування сучасного вебзастосунку, який у межах цієї кваліфікаційної роботи має назву «Voicer», потребує вибору архітектурної моделі, здатної забезпечити стабільну та швидку обробку медіаданих, коректну взаємодію між програмними компонентами, надійну ізоляцію бізнес-логіки від інфраструктурних змін, а також можливість подальшого масштабування системи. Особливість такого застосунку полягає в тому, що він працює не лише з типовими текстовими запитами користувача, а й з аудіофайлами, які потребують завантаження, збереження, передачі до сервісу розпізнавання мовлення та подальшої обробки отриманого текстового результату.

Для реалізації проєкту було обрано класичну розподілену клієнт-серверну архітектуру, яка ґрунтується на повному розділенні інтерфейсу користувача та серверної частини. Такий підхід дозволяє розмежувати зони відповідальності між клієнтським модулем, що відповідає за взаємодію з користувачем, і серверним модулем, який реалізує бізнес-логіку, керує доступом до бази даних, обробляє аудіофайли та забезпечує інтеграцію із зовнішніми моделями штучного інтелекту. Завдяки цьому клієнтська частина може розвиватися незалежно від серверної логіки, а сервер може бути доповнений новими функціями без повної перебудови користувацького інтерфейсу.

Впроваджена у вебзастосунку «Voicer» клієнт-серверна модель базується на фундаментальних принципах розподілу обов'язків. Клієнтська частина бере на себе роль шару візуалізації, тоді як серверна частина виступає централізованим постачальником даних. Комунікація між ними реалізована за технологією асинхронних запитів без збереження стану сесії на самому вебсервері. Тобто кожен HTTP-запит від клієнта є абсолютно самостійним, а

сервер обробляє його як незалежну ізольовану операцію.

Взаємодія між клієнтом і сервером відбувається за чітким циклом «запит-відповідь» (рис. 3.1). Коли користувач здійснює якусь дію, фронтенд формує асинхронний HTTP-запит. Сервер приймає цей запит, перевіряє вхідні дані цього запиту, витягує з бази даних історію повідомлень для відновлення контексту, робить запит до мовної моделі, зберігає нову репліку в базі даних та повертає результат назад клієнту, який миттєво оновлює стан чату на екрані.

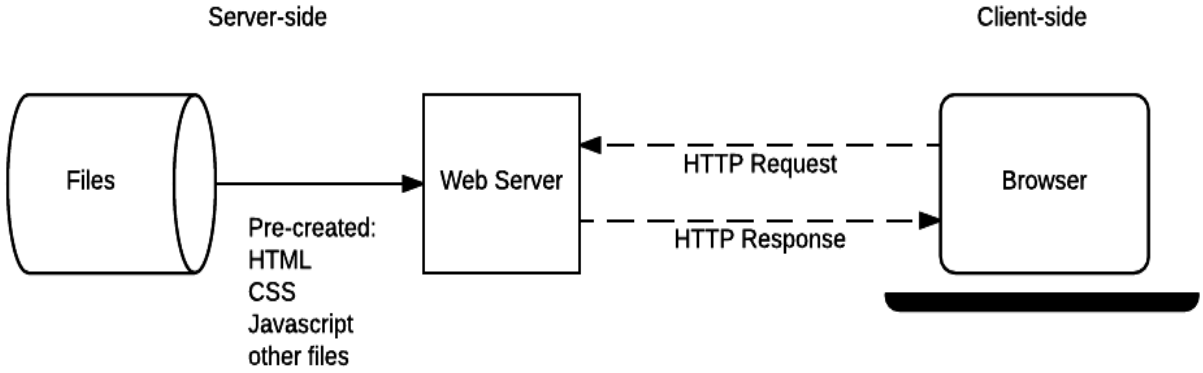


Рисунок 3.1 – Схема клієнт-серверної взаємодії «Request-Response»

Клієнтська частина проєктується як односторінковий вебзастосунок (SPA) на базі бібліотеки React. На відміну від багатосторінкових сайтів (MPA), SPA завантажує єдину кореневу сторінку та весь необхідний базовий JavaScript-код лише один раз – під час першої загрузки (рис. 3.2). Усі подальші зміни інтерфейсу, оновлення контенту та навігація між розділами відбуваються динамічно на стороні браузера за допомогою маніпуляцій із віртуальним деревом елементів (Virtual DOM).

Обрана архітектура дозволяє створювати чуйний й динамічний інтерфейс, який за швидкістю відгуку та плавністю переходів не поступається нативним десктопним застосункам. Основна робоча сторінка застосунку об'єднує спеціальну форму завантаження аудіо, візуалізацію тексту транскрибації та інтерактивний чат зі штучним інтелектом. Архітектура SPA

дозволяє користувачу завантажити аудіофайл, а потім отримати транскрибований текст з можливістю запитати у вбудованої моделі ШІ про оброблений текст.

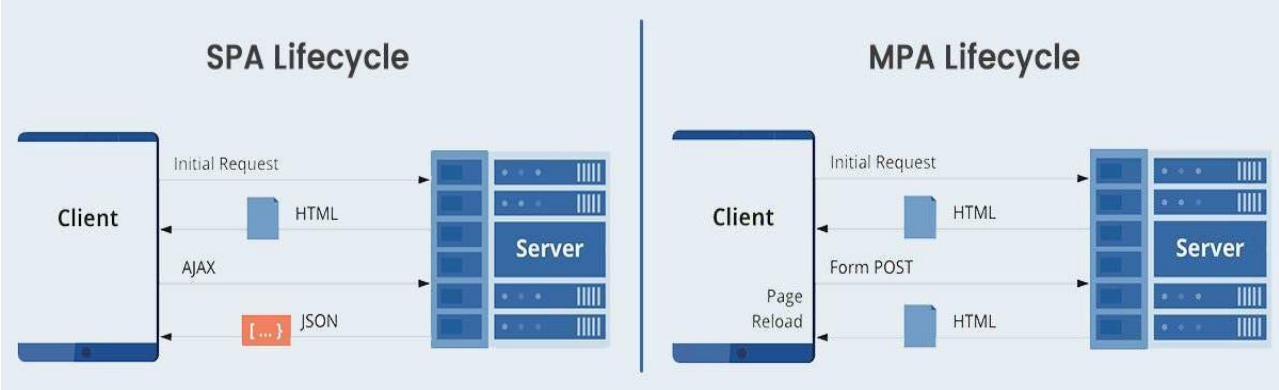


Рисунок 3.2 – Схема відмінностей архітектур SPA та MPA

Під час завантаження аудіофайлу через зону Drag-and-Drop клієнтський стан миттєво змінюється: початкове вікно завантаження плавно зникає, і на екрані з'являється інтерфейс поточної сесії з анімацією очікування ШІ-відповіді. SPA дозволяє рендерити ці зміни локально і синхронізувати їх із фоновими процесами, не змушуючи користувача чекати оновлення всієї сторінки. Оскільки вся графічна оболонка вже завантажена і знаходиться в пам'яті браузера, клієнт звертається до сервера виключно за необробленими даними у форматі JSON, що значно економить пропускну здатність.

Серверна частина застосунку реалізована на базі фреймворку NestJS із використанням мови TypeScript і проєктується відповідно до принципів шаруватої архітектури (рис. 3.3). Цей підхід передбачає суворий розподіл обов'язків у кодовій базі за допомогою розділення сервера на три логічні пласти: рівень контролерів (англ. Controllers), рівень сервісів (англ. Services) та рівень доступу до даних (англ. Infrastructure).

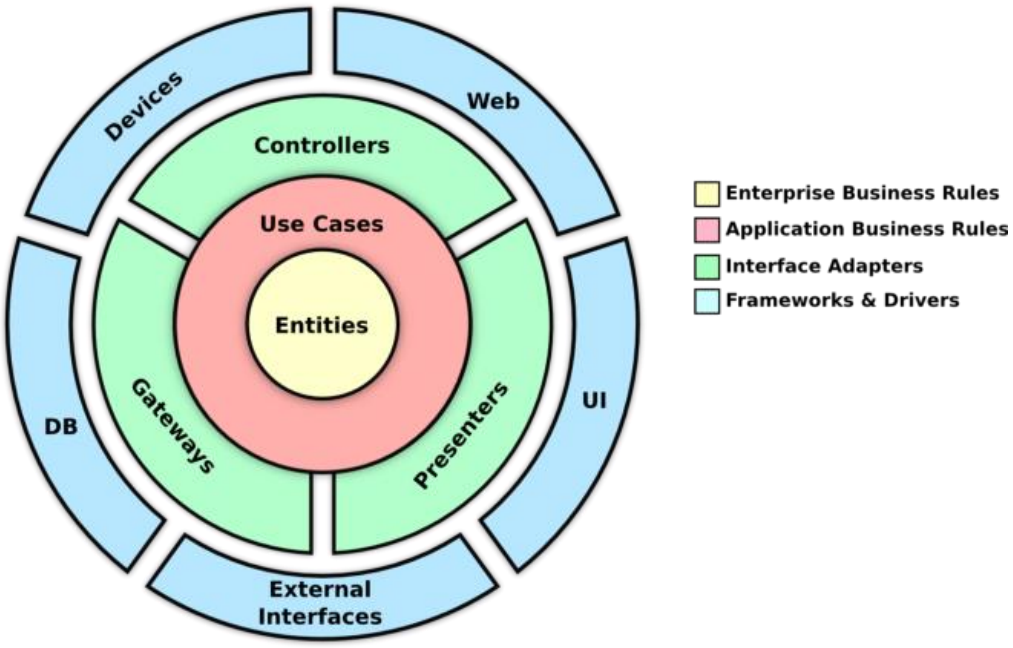


Рисунок 3.3 – Схема шаруватої архітектури Nest.js

Завдання першого рівня – це приймати вхідні HTTP-запити від клієнта, виконувати первинну валідацію полів та повертати кінцеву відповідь.

Другий рівень займається безпосередньо бізнес-логікою вебзастосунку, обробляючи та дані, які були отримані від контролерів.

Останній рівень системи відповідає за фізичну взаємодію з інфраструктурою – виконання SQL-запитів до бази даних та завантаження файлів у бакети хмарного сховища (рис. 3.4).

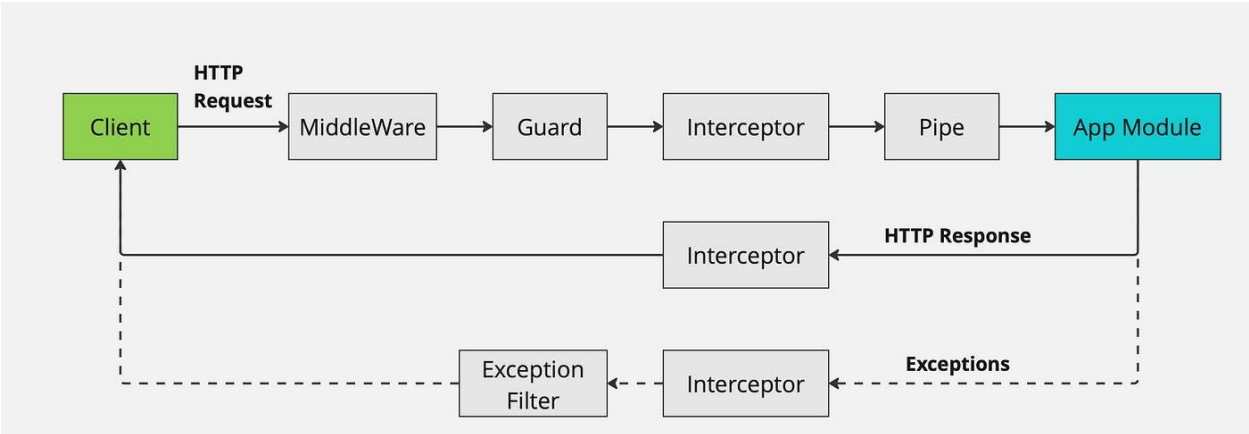


Рисунок 3.4 – Схема повного шляху запиту в Nest.js

Вся кодова база сервера організована за модульним принципом. Компоненти об'єднані всередині закритих незалежних блоків, які чітко описують свої межі та експортні можливості (рис.3.5). Такий підхід гарантує повну ізоляцію функціональних частин, забезпечує високу читабельність коду, полегшує процес тестування та модифікації окремих елементів системи, не порушуючи стабільності роботи всього вебзастосунку (рис.3.6).

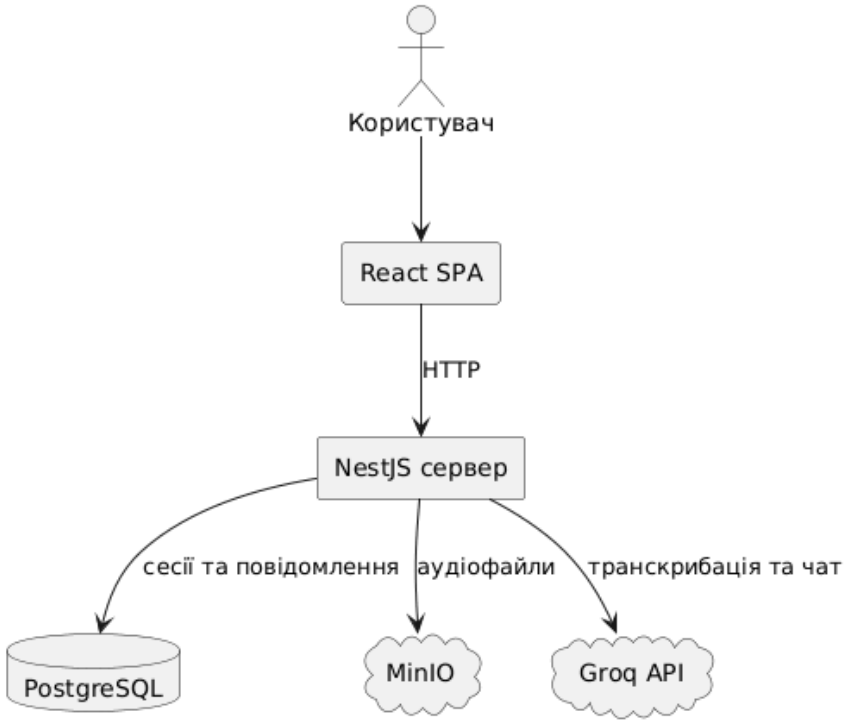


Рисунок 3.5 – Діаграма компонентів вебзастосунку Voicer

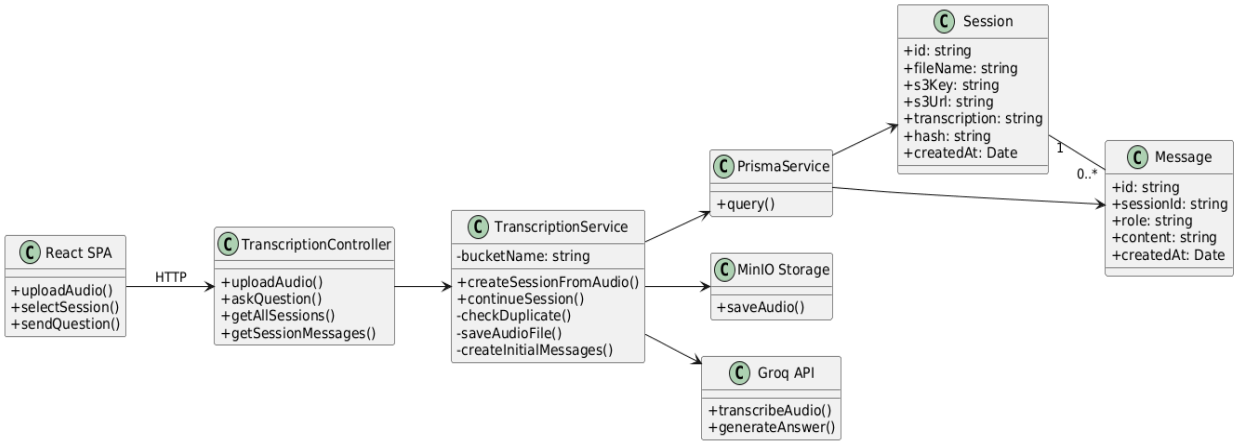


Рисунок 3.6 – Діаграма класів вебзастосунку Voicer

3.2 Проектування архітектури бази даних

Для того, щоб організувати надійне та цілісне збереження інформації застосунку, в проєкті було розгорнуто реляційну СУБД PostgreSQL. Її архітектура була спроєктована відповідно до принципів реляційного моделювання. А надійна структура та зв'язки між таблицями допомогають повністю уникнути помилок при додаванні, модифікації або видаленні об’єктів у таблицях.

Всього у системі було спроектовано два основних типи інформації: дані по аудіофайлу та історії текстових діалогів у межах створених сессій. Також було спроектовано і створено дві таблиці: таблиця зберігання сесій (англ. Sessions) та таблиця зберігання повідомлень (англ. Messages). Зв'язок між ними побудован на основі первинних і зовнішніх ключів, які гарантують цілісність їх подальшої взаємодії.

Структура полів таблиці «Sessions»:

- id: унікальний ідентифікатор сесії;
- fileName: оригінальна назву аудіофайлу, що завантажувався користувачем;
- s3Key: унікальний текстовий шлях до об'єкта в бакеті MinIO. Цей ключ необхідний серверу для точної ідентифікації та управління файлом напряму через S3 API;
- s3Url: повна HTTP-адреса збереженого файлу;
- transcriptio: транскрибований текст;
- hash: унікальний SHA-256 відбиток файлу;
- createdAt: часова мітка створення сесії.

Структура полів таблиці «Messages»:

- id: унікальний ідентифікатор окремого повідомлення в чаті;
- sessionI: зовнішній ключ-посилання на ідентифікатор id у таблиці «Sessions»;

– role: текстовий маркер, який визначає автора повідомлення (SYSTEM, USER або ASSISTANT);
– content: поле тексту репліки чи відповіді від штучного інтелекту;
– createdAt: часова мітка створення повідомлення.

Паралельно з реляційною БД, у архітектуру було інтегровано об'єктне S3-сховище MinIO. Через те, що збереження великих файлів в PostgreSQL призводить до падіння швидкодії та швидкого розростання бази, у системі реалізовано схему їх взаємодії (рис. 3.7). Самі аудіофайли фізично розміщуються у спеціальному ізольованому хмарному бакеті, а СУБД виступає центральним реєстром, який надійно зберігає точні посилання на ці об'єкти за допомогою полів s3Key та s3Url.

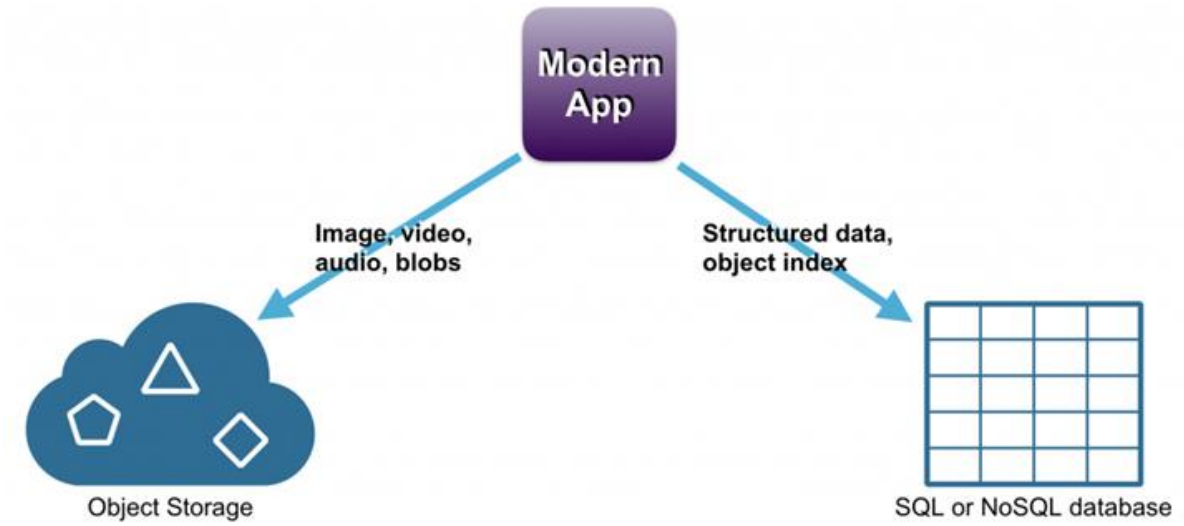


Рисунок 3.7 – Схема зберігання різних типів даних у базах даних та хмарних сховищах

У результаті ці поля дозволяють клієнту відтворювати аудіопотік у браузері безпосередньо з об'єктного сховища MinIO, що повністю знімає навантаження на сервер. Паралельно з цим, збережений у таблиці «Sessions» атрибут s3Key гарантує цілісність та синхронізацію інфраструктури (рис.3.8).

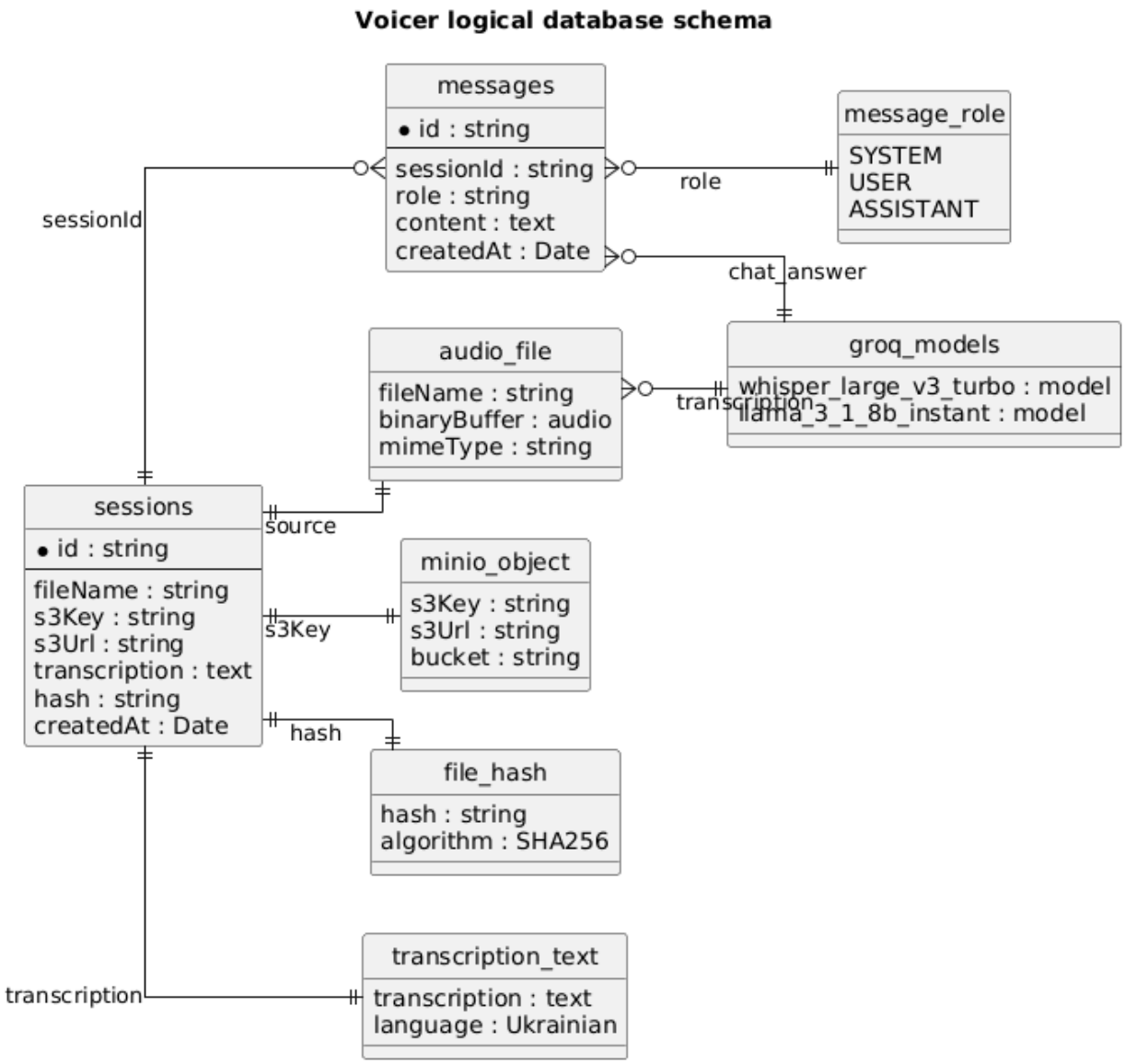


Рисунок 3.8 – Схема проекту бази даних застосунку

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ

4.1 Розробка серверної частини

Архітектура серверної частини кваліфікаційної роботи була побудована на фреймворку Nest.js. Його головна особливість – це можливість розбиття системи на окремі незалежні модулі. Такий підхід забезпечує чітку та зрозумілу структуру коду, у якій кожен модуль виконує лише свою конкретну задачу.

Весь код розділений на три основні шари, які взаємодіють між собою за строго визначеними правилами, не змішуючи логіку додатка.

Першим шаром є рівень представлення даних, який складається з контролерів. У проєкті цей рівень реалізовується у класі TranscriptionController (лістинг 4.1). Його основне завдання – приймати HTTP-запити від клієнтської частини, перевіряти наявність обов'язкових полів і повертати кінцеву відповідь. Наприклад, коли користувач завантажує аудіофайл, контролер приймає цей потік даних, перевіряє його наявність і відразу передає далі на інший шар. Сам контролер не знає, як працює бізнес-логіка системи чи як влаштована база даних. Його єдина задача – це прийняти дані, а потім передати їх далі на інший шар. У контролері реалізовано п'ять методів:

1) Метод uploadAudio (лістинг 4.2) приймає аудіофайл, перевіряє його за певними правилами і потім передає в сервіс для подальшої обробки.
2) Метод askQuestion (лістинг 4.3) приймає ідентифікатор сесії та текстове запитання користувача, валідує їх, а потім знову надсилає дані в сервіс для для обробки.
3) Метод getAllSessions (лістинг 4.4) не приймає жодних параметрів і потрібен тільки для отримання повного списку всіх створених сесій.
4) Метод getActiveTranscription (лістинг 4.5) приймає як параметр унікальний ідентифікатор сесії та повертає первинний текст, який попередньо було транскрибовано з аудіофайлу.

5) Метод getSessionMessages (лістинг 4.6) так само приймає ідентифікатор сесії та повертає повну хронологічну історію повідомлень між користувачем та ШІ.

```
@Controller('transcription')
export class TranscriptionController {
  constructor(private readonly transcriptionService:
TranscriptionService) {}
  //  методи контролеру
}
```

Лістинг 4.1 – Лістинг контролера TranscriptionController

```
@Post('upload')
  @UseInterceptors(FileInterceptor('file'))
  async uploadAudio(@UploadedFile() file: Express.Multer.File) {
    if (!file) {
      throw new BadRequestException('Please upload the audio
file');
    }
    const result = await
this.transcriptionService.createSessionFromAudio(file);
    return {success: true, ...result};
  }
```

Лістинг 4.2 – Лістинг методу uploadAudio

```
@Post('ask')
  async askQuestion(@Body('sessionId') sessionId: string,
@Body('question') question: string) {
    if (!sessionId || !question) {
      throw new BadRequestException(
        'The "sessionId" and "question" fields are required',
      );
    }
    const aiAnswer = await
this.transcriptionService.continueSession(sessionId, question);
    return {success: true, aiAnalysis: aiAnswer };
  }
```

Лістинг 4.3 – Лістинг методу askQuestion

```
@Get('getAllSessions')
  async getAllSessions() {
```

```
    const sessions = await
this.transcriptionService.getAllSessions();
    return {success: true, sessions};
  }
```

Лістинг 4.4 – Лістинг методу getAllSessions

```
 @Get('session/:sessionId')
  async getActiveTranscription(@Param('sessionId') sessionId:
string) {
    if (!sessionId) throw new BadRequestException('SessionId is
required');
    const messages = await
this.transcriptionService.getActiveTranscription(sessionId);
    return {success: true, messages};
  }
```

Лістинг 4.5 – Лістинг методу getActiveTranscription

```
@Get('session/:sessionId/messages')
  async getSessionMessages(@Param('sessionId') sessionId:
string) {
    if (!sessionId) throw new BadRequestException('The sessionId
parameter is required');
    const messages = await
this.transcriptionService.getSessionMessages(sessionId);
    return {success: true, messages};
  }
```

Лістинг 4.6 – Лістинг методу getSessionMessages

Другим і самим складним шаром є рівень бізнес-логіки, який представлений сервісом TranscriptionService (лістинг 4.7). Саме він є головним «мозком» системи, де зосереджені всі обчислювальні процеси та головні правила роботи програми. Сервіс транскрибації відповідає за перевірка дублікатів файлів, координацію відправки аудіо в хмарне сховище, він викликає моделі штучного інтелекту для розпізнавання тексту, а також передає історію повідомлень мовній моделі для подальшого аналізу.

```
@Injectable()
export class TranscriptionService {
  private groq: Groq;
  private s3Client: S3Client;
  private bucketName: string;
  constructor(private db: PrismaService) {
    this.groq = new Groq({ apiKey: process.env.GROQ_API_KEY });
    this.s3Client = new S3Client({
      endpoint: process.env.S3_ENDPOINT ||
'http://localhost:9000',
      credentials: {
        accessKeyId: process.env.S3_ACCESS_KEY_ID ||
'minioadmin',
      },
      forcePathStyle: true,
    });
    this.bucketName = process.env.S3_BUCKET_NAME || 'audio-
sessions';
  }
}
```

Лістинг 4.7 – Лістинг сервісу TranscriptionService

Сервіс виступає посередником, який отримує дані від контролера, а потім обробляє їх за допомогою ШІ та готує для запису в базу даних.

Третім шаром є рівень доступу до даних. Він представлений сервісом PrismaService (лістинг 4.8) та вбудованим клієнтом S3Client. Шар організовує фізичну взаємодію з інфраструктурою – виконання прямих SQL-запитів до бази даних PostgreSQL та завантаження аудіофайлів у бакети сховища MinIO (рис. 4.1). Йому в загалі не потрібно розуміти принципи обробки даних чи як організована логіка транскрибації. Він просто викликає готові методи доступу до сховищ для швидкого оновлення бази даних.

```
@Injectable()
export class PrismaService implements OnModuleInit,
OnModuleDestroy {
  public pool: Pool;
  constructor() {
    this.pool = new Pool({
      connectionString: process.env.POSTGRES_URI ||
        'postgresql://postgres:postgres@localhost:5433/voicer-
nestjs?schema=public' });
  }
  async onModuleInit() {
```

```
    const client = await this.pool.connect();
    client.release();
  }
  async onModuleDestroy() { await this.pool.end();}
  async query(text: string, params?: any[]) {
    return this.pool.query(text, params);
  }
}
```

Лістинг 4.8 – Лістинг сервісу PrismaService

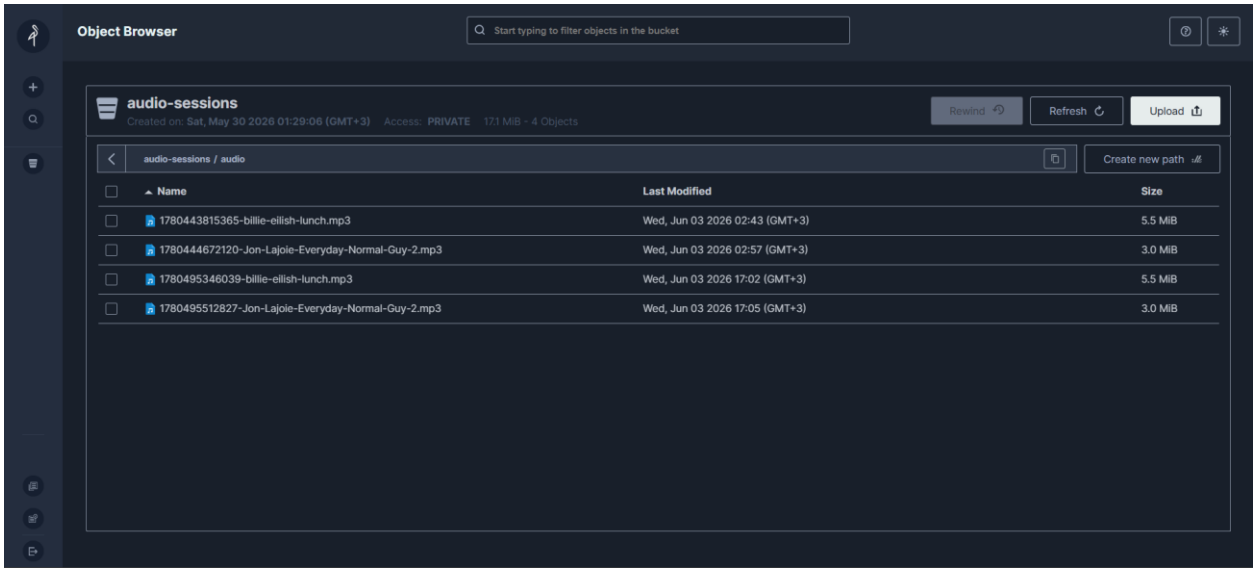


Рисунок 4.1 – Збережені дані аудіофайлів у хмарному середовищі MinIO

Усі вищеописані компоненти – контролер, сервіс та шар підключення до БД – об'єднані всередині одного логічного блоку, який називається TranscriptionModule. Цей модуль чітко описує, які класи є приватними, а які можна експортувати для використання в інших частинах програми. Завдяки такому підходу, критичні блоки системи залишаються ізольованими від зовнішнього впливу. Це забезпечує простоту читання коду, полегшує тестування і дебагінг, та дозволяє гнучко модифікувати окремі елементи, без втрати стабільності.

Серверна частина спроєктована таким чином, щоб весь життєвий цикл – від моменту натискання кнопки «Завантажити файл» на клієнті до отримання розгорнутої відповіді від чат-бота – відбувався швидко та надійно. Весь цей

процес можна розділити на чотири великі блоки: прийом аудіофайлу; синхронне завантаження в об'єктне сховище; швидка транскрибація мовлення; контекстне ведення діалогу з мовною моделлю.

Перший блок відповідає за вхідну обробку файлу та захист системи від повторного завантаження дублікатів. Коли користувач надсилає аудіофайл через спеціальну форму на клієнтській частині, сервер приймає його у вигляді тимчасового бінарного буфера. Така операція досить сильно навантажує процесор і витрачає ліміти хмарного API. Щоб уникнути зайвих витрат ресурсів, сервер виконує дедуплікацію.

Дедуплікація – це технологія стиснення даних, яка знаходить у сховищі копії даних і видаляє їх.

За допомогою вбудованого в Node.js модуля crypto сервер зчитує весь вміст файлу та обчислює його унікальну контрольну суму за криптографічним алгоритмом SHA-256. Отриманий рядок із 64 символів є абсолютно унікальним цифровим «відбитком» цього конкретного файлу. Якщо користувач спробує завантажити один і той самий файл кілька разів, то його хеш-код залишиться незмінним. Сервер робить запит до таблиці «Sessions» у PostgreSQL, перевіряючи наявність такого хешу. Якщо збіг знайдено, алгоритм обробки припиняється. Після цього сервер не завантажує файл повторно і не відправляє його штучному інтелекту. Він просто повертає клієнту ідентифікатор вже існуючої старої сесії і текст помилки.

Якщо файл завантажується вперше, то система переходить до другого функціонального блоку (лістинг 4.9). Сервер починає процедуру збереження аудіозапису. Для цього використовується офіційний клієнт @aws-sdk/client-s3. Спочатку сервер формує унікальний ключ об'єкта, додаючи до оригінального імені файлу поточну мітку часу в мілісекундах. У подальшому, цей унікальний ключ буде шляхом до завантаженого файлу у сховищі. Далі створюється спеціальна команда, яка передає назву бакета, згенерований шлях, сам бінарний буфер файлу та його тип. Клієнт S3 надсилає ці дані в локальне об'єктне сховище MinIO . Після успішного завантаження, сервер формує повне

HTTP-посилання на цей файл, яке потім записується в базу даних.

```
async createSessionFromAudio(file: Express.Multer.File) {
    try {
      const fileHash =
crypto.createHash('sha256').update(file.buffer).digest('hex');
      const existingSession = await this.db.query();
      const s3Key = `audio/${Date.now()}-${file.originalname}`;
      const uploadCommand = new PutObjectCommand({
        Bucket: this.bucketName,
        Key: s3Key,
        Body: file.buffer,
        ContentType: file.mimetype,
      });
      await this.s3Client.send(uploadCommand);
      const audioStream = this.bufferToStream(file.buffer);
      const groqFile = await Groq.toFile(audioStream,
file.originalname);
      return {
        isDuplicate: false
      };
    } catch (error) {
      throw new InternalServerErrorException(
        `Error processing audio: ${error}`,
      );
    }
  }
```

Лістинг 4.9 – Лістинг Лістинг методу createSessionFromAudio

Для надійного збереження оригінальних аудіофайлів у вебзастосунку реалізовано взаємодію з об'єктним сховищем MinIO за допомогою S3. Воно дозволяє не перевантажувати бекенд та забезпечує швидкий доступ до аудіоданих.

Взаємодія з MinIO на стороні сервера починається на етапі ініціалізації класу TranscriptionService. У конструкторі сервісу (лістинг 4.10) створюється новий екземпляр клієнта S3Client з офіційного пакета @aws-sdk/client-s3.

```
constructor(private db: PrismaService) {
    this.s3Client = new S3Client({
      endpoint: process.env.S3_ENDPOINT ||
'http://localhost:9000',
      region: process.env.S3_REGION || 'us-east-1',
      credentials: {
```

```
        accessKeyId: process.env.S3_ACCESS_KEY_ID ||
'minioadmin',
        secretAccessKey: process.env.S3_SECRET_ACCESS_KEY ||
'minioadmin',
      },
      forcePathStyle: true,
    });
    this.bucketName = process.env.S3_BUCKET_NAME || 'audio-
sessions';
  }
}
```

Лістинг 4.10 – Лістинг створення екземпляру клієнта S3Client у конструкторі сервісу TranscriptionService

При конфігурації клієнта S3Client, потрібно налаштувати декілька важливих полів:

- endpoint – мережева адреса сховища MinIO;
- region – географічний регіон сховища;
- credentials – пара секретних ключів, які захищають сховище від несанкціонованого доступу;
- forcePathStyle – спеціальний прапорець, виставлений у значення true, який є обов'язковим для коректної роботи локальних контейнерів MinIO.

Сам процес завантаження файлу в сховище є повністю синхронним і відбувається у методі createSessionFromAudio.

Третій функціональний блок відповідає за безпосереднє розпізнавання мовлення. Оскільки хмарне SDK платформи Groq не може працювати з необробленими файловими буферами з оперативної пам'яті, сервер використовує механізм потоків. Для цієї операції використовується метод Readable.from(). Він перетворює бінарний буфер аудіофайлу на читабельний потік даних, з яким вже можна працювати. Цей потік передається у внутрішній метод Groq.toFile(), який адаптує його під вимоги веб-запитів. Після цього сервер робить асинхронний виклик до API Groq, вказуючи модель whisper-large-v3-turbo та мову розпізнавання (у нашому застосунку українську).

Завдяки LPU обчислення відбуваються миттєво. Замість хвилин очікування, які зазвичай потрібні класичним серверам, модель Whisper повертає розпізнаний текст у форматі JSON всього за кілька секунд. Сервер отримує цей текст, генерує новий унікальний ідентифікатор сесії і за допомогою SQL-запиту вносить дані про нову сесію в таблицю sessions.

Четвертий функціональний блок забезпечує роботу інтерактивного чат-асистента. Після того як текст аудіофайлу успішно збережено, сервер автоматично створює три стартові записи в таблиці «Messages», які ініціалізують контекст майбутнього діалогу. Перше повідомлення – це системна інструкція «SYSTEM», яка приховується на клієнті. Друге повідомлення імітує запит від користувача «USER», куди вставляється весь розпізнаний текст аудіофайлу з запросом на підготовку до подальшого аналізу. Третє повідомлення – це відповідь асистента «ASSISTANT» про успішну фіксацію контексту сессії.

Коли користувач надсилає нове запитання до аудіозапису через ендпоінт «/ask» (лістинг 4.11), сервер спочатку перевіряє існування сесії, а потім додає це питання до таблиці «Messages». Після цього сервер робить вибірку всієї історії повідомлень для цієї сесії та сортує їх строго за часом створення. Отримані рядки трансформуються в чистий масив об'єктів, де кожна репліка має чітку роль. Цей масив разом із новим запитанням відправляється на обробку великій мовній моделі llama-3.1-8b-instant. Для того, щоб модель видавала точні відповіді, сервер жорстко обмежує її «креативність» за допомогою спеціального параметра temperature, який виставляється на низький рівень. Отриманий від моделі Llama текст сервер автоматично записує в базу даних для збереження хронології чату. Після цього оновлені данні БД відправляються на клієнт.

```
async continueSession(sessionId: string, userQuestion: string,){
    const sessionCheck = await this.db.query();
    await this.db.query(`INSERT INTO messages ("id",
"sessionId", "role", "content") VALUES ($1, $2, $3, $4)`,
      [randomUUID(), sessionId, 'USER', userQuestion]);
```

```
    try {
      const chatCompletion = await
this.groq.chat.completions.create({ model: 'llama-3.1-8b-
instant', messages: [ {role: 'system', content: 'Ти –
професійний AI-асистент платформи Voicer…'}, ...groqMessages, {
role: 'user', content: userQuestion }], temperature: 0.2,
      });
    } catch (error) {
      throw new InternalServerErrorException(`Error:
${error}`);}
  }
```

Лістинг 4.11 – Лістинг методу continueSession

4.2 Реалізація інтерфейсу та клієнтської логіки

Клієнтська частина вебзастосунку «Voicer» була розроблена за допомогою компонентної бібліотеки React за модульним принципом. Потрібно було створити швидкий та привабливий інтерфейсу, який зміг би забезпечувати максимальний комфорт при тривалому користуванні.

Для зрозумілого розділення архітектури, кодова база була розбита на ізольовані самостійні UI-компоненти, які можна використовувати багаторазово. Для забезпечення сучасного візуального стилю та прискорення етапу розробки в проєкті було використано набір компонентів з бібліотеки Shadcn UI, які побудовані на базі CSS-фреймворку TailwindCSS. Такий підхід дозволив позбутися зайвих великих файлів стилів та зробити проект більш легковажним та зрозумілим.

Однією з корисних функцій була інтеграція нічної теми у проект, яка була налаштована за допомогою вбудованих інструментів Shadcn UI (рис. 4.2). Перемикання між світлою (рис. 4.3) та темною темами (рис. 4.4) було реалізовано на найвищому рівні дерева компонентів. При такому підході зміна стилів усіх видимих елементів – від кольору тексту та кнопок до заповнення складних глобальних компонентів вебзастосунку, відбувається миттєво та синхронно (лістинг 4.12).

```
function App() {
   return (
      <ThemeProvider defaultTheme="dark">
                {children}
     </ThemeProvider>
   )}
export default App
```

Лістинг 4.12 – Лістинг головного компоненту App, який обернут у компонент управління темами вебзастосунку

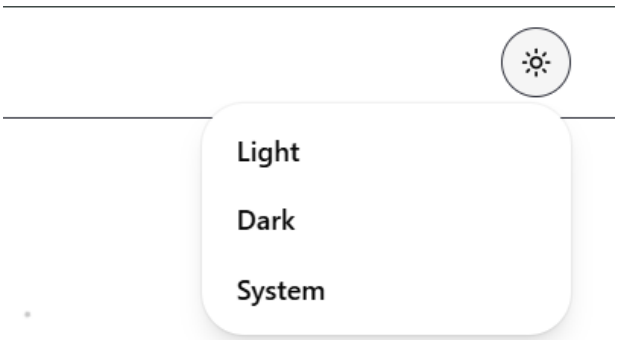


Рисунок 4.2 – Вбудований елемент перемикання теми вебзастосунку



Рисунок 4.3 – Вигляд вебзастосунку зі світлою темою

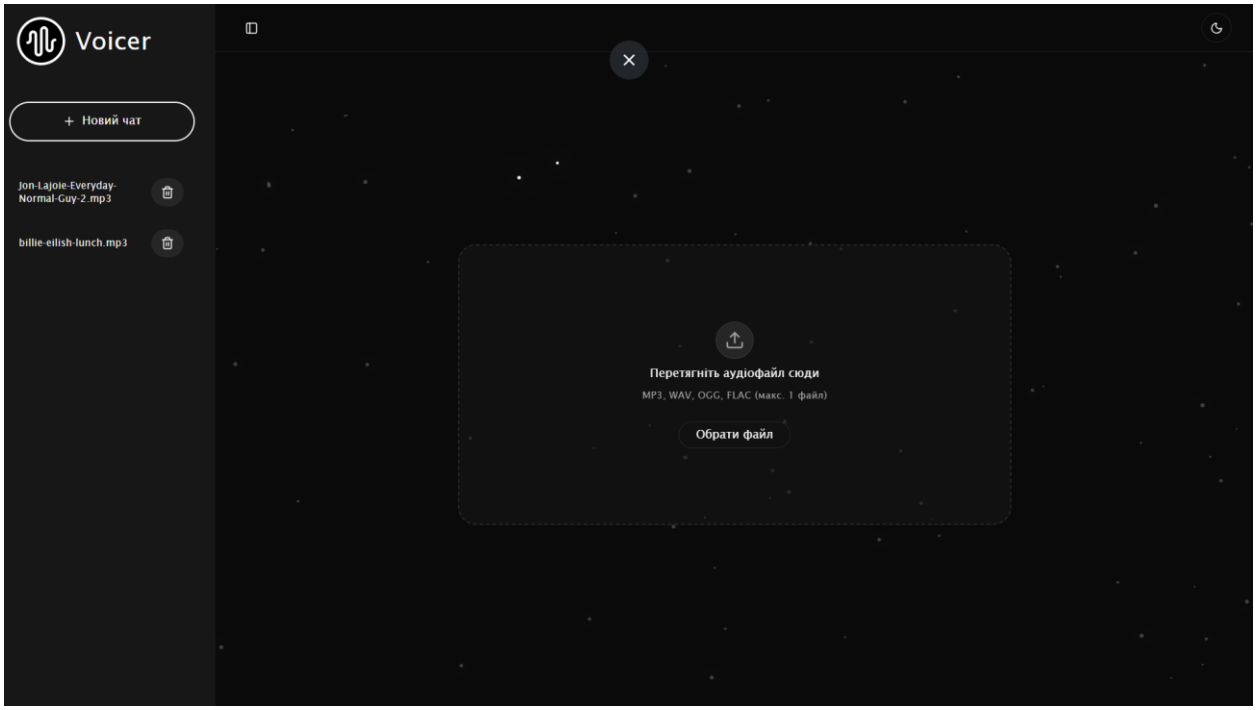


Рисунок 4.4 – Вигляд вебзастосунку з темною темою

Також в процесі розробки інтерфейсу, була вбудована плавна та приємна анімації фону додатка. Ця анімація створює сучасний візуальний ефект глибини простору та робить інтерфейс більш «живим».

Головна сторінка розділена на кілька частин, кожна з яких відповідає за сфою окрему функцію взаємодії користувача з платформою. Першим базовим елементом структури є бічна панель навігації, яка виступає основним інструментом керування хронологією робочих сесій. На рівні клієнтської логіки тут динамічно відображається список усіх попередніх сесій транскрибації, який автоматично сортується за часом їх створення.

Кожен інтерактивний елемент цього списку містить метадані про завантажений аудіофайл і надає швидкий доступ до відновлення повної історії обраного діалогу. Компонент бічної панелі повністю оптимізований для екранів з різною роздільною здатністю. Він також виконує функцію плавного згортання та розгортання з використанням анімацій. Це дозволяє максимально вивільнити корисну площу екрана на мобільних пристроях чи планшетах.

Окрім навігації, у бічну панель також винесені елементи керування для створення нових сесій.

Центральна область сторінки на початковому етапі взаємодії користувача з системою представлена інтерактивним модулем завантаження файлів, що функціонує за технологією «Drag-and-Drop». Цей компонент розроблений як головна точка входу для обробки медіаданих. Саме цей компонент дозволяє завантажувати файл для його подальшої обробки. Якщо наприклад розширення файлу не відповідає дозволеному пулу аудіоформатів (MP3, WAV, FLAC), тоді система миттєво блокує подальше відправлення, генеруючи спливаюче сповіщення про помилку (рис. 4.5). Таке рішення дозволяє повністю вберегти систему від некоректних запитів на етапі входу даних у систему, а також робить інтерфейс більш зрозумілим для користувача (лістинг 4.12).

```
const onFileValidate = useCallback((file: File): string | null
=> {
        const allowedExtensions = ['.mp3', '.wav', '.ogg',
'.flac']
        const fileName = file.name.toLowerCase()

        const hasValidExtension = allowedExtensions.some(ext =>
            fileName.endsWith(ext)
        )
        if (!hasValidExtension) {
            setShowErrorAlert(true)
            setTimeout(() => {
                setShowErrorAlert(false)
            }, 2000)
            return `Разрешены только аудиофайлы:
${allowedExtensions.join(', ')}`
        }

        return null
    }, [])
```

Лістинг 4.12 – Лістинг функції валідації аудіо файлів у компоненті «Drag-and-Drop»

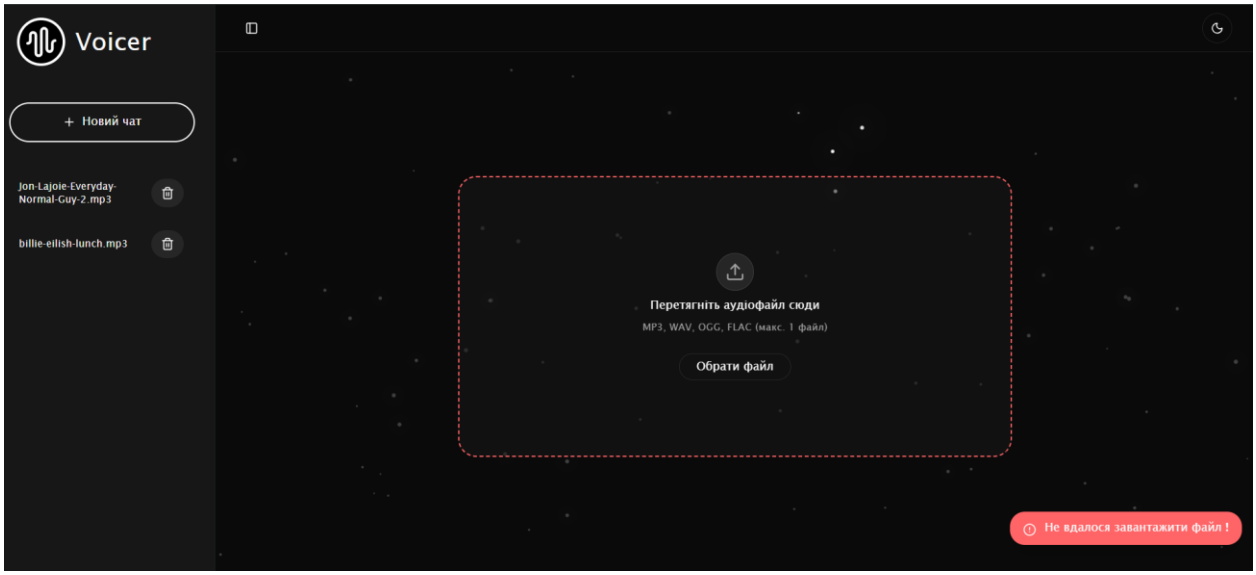


Рисунок 4.5 – Відображення вспливаючого повідомлення при помилці завантаження аудіофайлу

Завдяки реактивному управлінню станами React, при успішному завантажені аудіо файлу, відбувається структурна перебудова сторінки, а інтерфейс плавно трансформується в режимі реального часу: початковий віджет Drag-and-Drop повністю вилучається з DOM-дерева і на його місце стає комплексний робочий простір сесії з чатом.

Компонент інтерфейсу чату підключає користувача до великої мовної моделі Llama. Як і у сучасних меседжерах, чат відображає повідомлення та транскрибований текст у режимі реального часу (рис. 4.6, 4.7). Повідомлення користувача відображаються у правій частині вікна та фарбуються в акцентний колір фону, а відповіді штучного інтелекту позиціонуються зліва та відображаються у нейтральному кольорі.



Рисунок 4.6 – Сторінка чату вебзастосунку

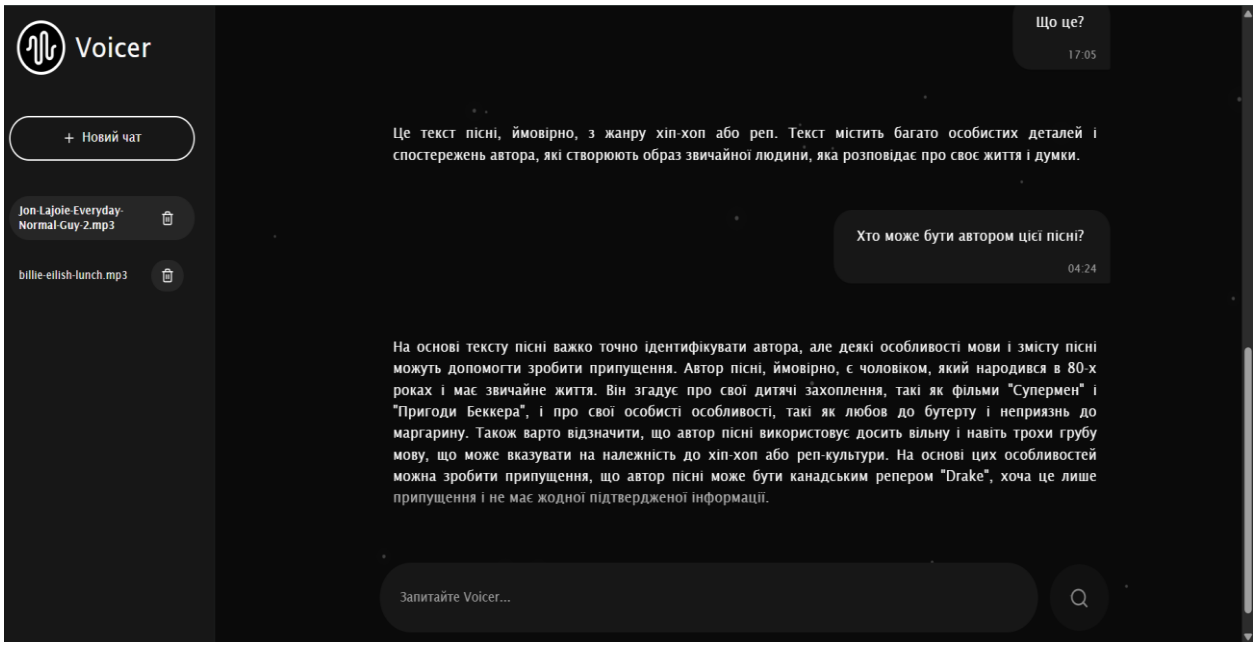


Рисунок 4.7 – Сторінка чату вебзастосунку відповідями ШІ-моделі

Важливим аспектом забезпечення надійності на рівні інтерфейсу користувача є проєктування сценаріїв на випадок повної втрати зв'язку з бекендом або технічних несправностей. Для запобігання таким ситуаціям, в

архітектуру клієнта було інтегровано спеціалізовану повноцінну сторінку аварійного стану (рис. 4.8).

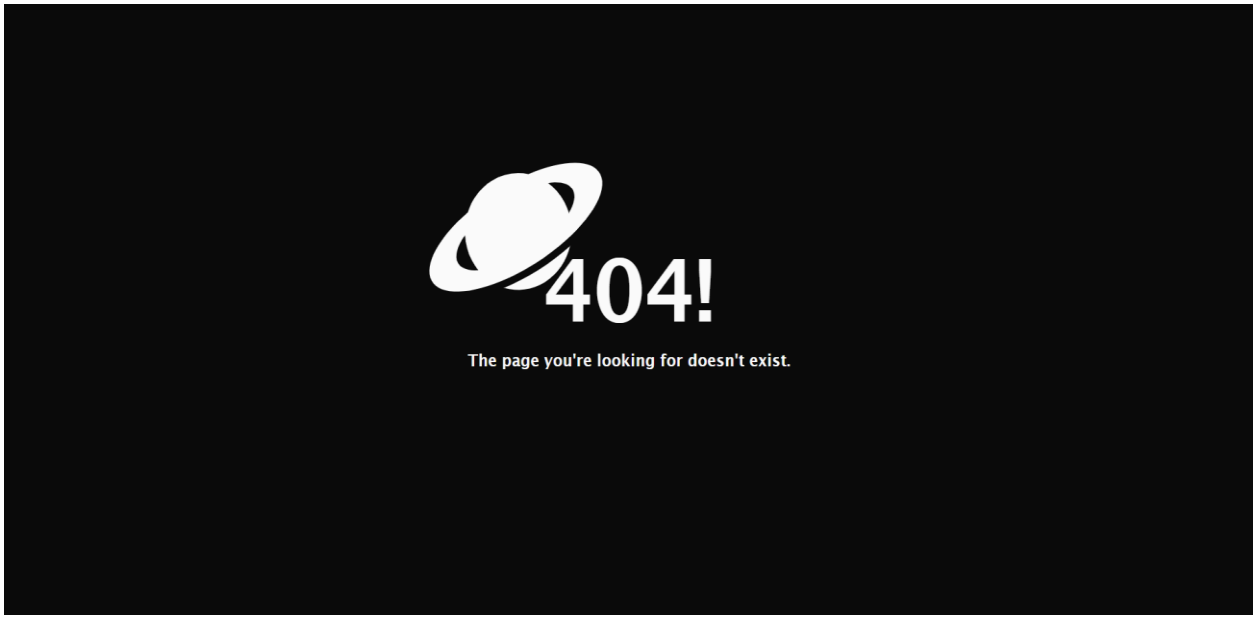


Рисунок 4.8 – Сторінку аварійного стану вебзастосунку

Якщо при відсутності з’єднання з сервером з’являється критичний статус помилки, наприклад, HTTP-статуси, які починаються з п’ятірки, тоді клієнтська логіка перехоплює цей стан глобальним обробником. У цей момент відбувається повна перебудова всього інтерфейсу: рендериться лаконічна інформаційна сторінка з описом невдалої операції. Вона сповіщає користувача про тимчасові технічні несправності, що суттєво підвищує інтуїтивність додатку.

Уся логіка взаємодії з сервером, отримання даних, кешування та синхронізації стану сервера з інтерфейсом користувача реалізована за допомогою RTK Query (лістинг 4.13). Він є розширенням бібліотеки Redux Toolkit. Використання цієї технології дозволило повністю відмовитися від ручного написання коду для відправки HTTP-запитів, обробки помилок та управління локальними станом компонентів. Замість цього вся взаємодія з бекендом NestJS описана у вигляді єдиного сервісного шару «Store».

```
createRoot(document.getElementById('root')!).render(
    <StrictMode>
        <Provider store={store}>
            <BrowserRouter>
                <App />
            </BrowserRouter>
        </Provider>
    </StrictMode>
)
```

Лістинг 4.13 – Лістинг вкладеної структури компонентів для налаштування RTK Query

На основі конфігурації ендпоінтів RTK Query автоматично генерує React-функції «хуки» для кожної окремої операції. Для отримання даних використовуються запити, а для зміни глобального стану системи на сервері використовуються мутації (лістинг 4.14). Це дозволяє автоматично отримувати в реальному часі не лише самі дані, а й статуси загрузки запиту або помилки (рис. 4.9). Це дозволяє миттєво реагувати на статус виконання операцій та відображати різні компоненти під різні ситуації.

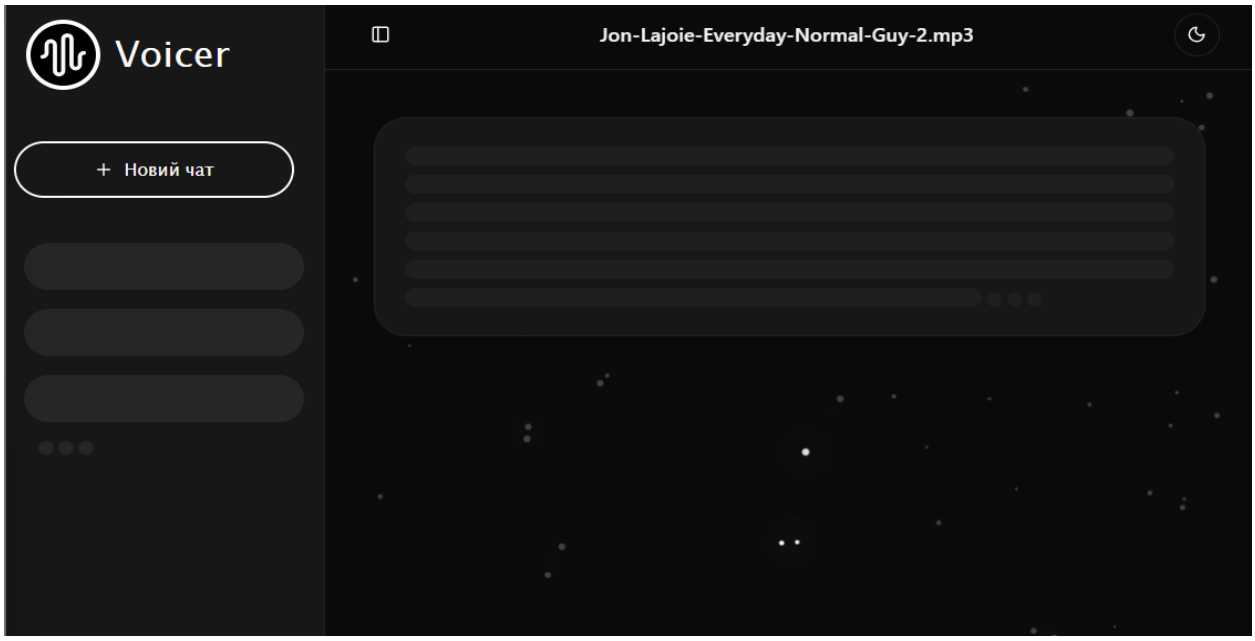


Рисунок 4.9 – Вигляд вебзастосунку при тривалому завантаженні даних з серверу

Важливою перевагою RTK Query є вбудований механізм кешування даних. Коли користувач перемикається між сесіями за допомогою бічної панелі, додаток не здійснює повторних запитів до сервера, якщо ці дані вже завантажувалися раніше і термін дії кешу не закінчився.

```
export const audioApi = createApi({
    reducerPath: 'audioApi',
    baseQuery: fetchBaseQuery({
        baseUrl: 'http://localhost:3000/transcription/'
    }),
    tagTypes: ['Messages', 'Sessions'] as const,
    endpoints: builder => ({
        askQuestion: builder.mutation<
            string,
            { sessionId: string; question: string }
        >({
            query: ({ sessionId, question }) => ({
                url: 'ask',
                method: 'POST',
                body: { sessionId, question }
            }),
            transformResponse: (response: {
                success: boolean
                aiAnalysis: string
            }) => response.aiAnalysis,

            invalidatesTags: (result, error, { sessionId }) => [
                { type: 'Messages', id: sessionId }
            ]
        }),
        uploadAudio: builder.mutation<IUploadResponse, File>({
            query: file => {
                const formData = new FormData()
                formData.append('file', file)

                return {
                    url: 'upload',
                    method: 'POST',
                    body: formData
                }
            },
            transformErrorResponse: (response: {
                status: number
                data: { error: { message: string } }
            }) => {
                console.log(response.data)

                return (
                    response.data?.error?.message ??
```

```
                    'Произошла неизвестная ошибка'
                )
            },
            invalidatesTags: ['Sessions']
        })
    })
})
export const {
    useGetAllSessionsQuery,
    useAskQuestionMutation,
    useUploadAudioMutation
} = audioApi
```

Лістинг 4.14 – Лістинг програмної реалізації шару асинхронних запитів клієнтської частини за допомогою RTK Query

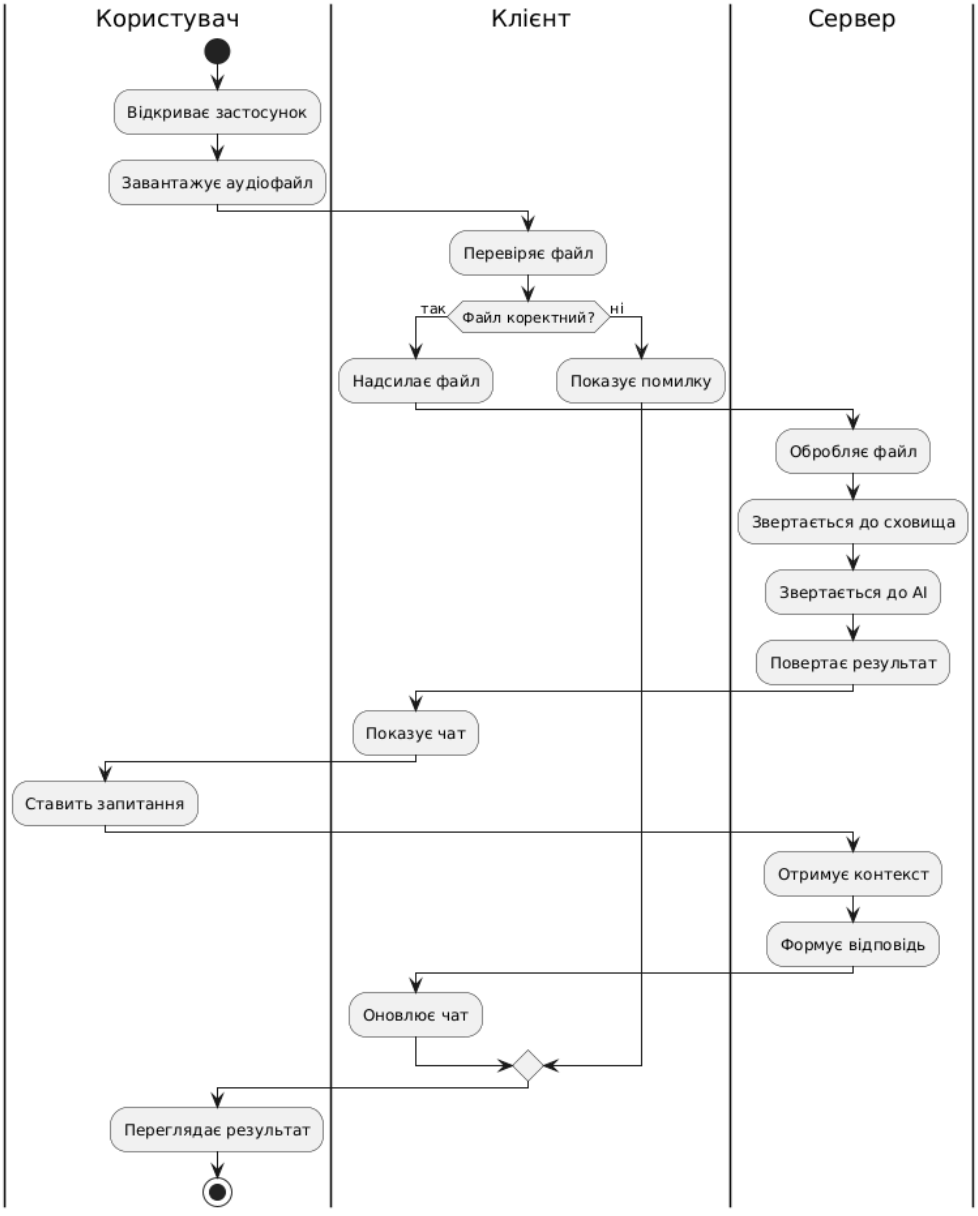


Рисунок 4.10 – Діаграма активностей застосунку

У момент, коли користувач завантажує новий аудіофайл або надсилає копідомлення у чат, відповідна мутація автоматично відмічає тег списку сесій або історії повідомлень як застарілий. RTK Query перехоплює цю подію і самостійно ініціює фоновий запит для оновлення лише тих компонентів, чиї дані змінилися. Такий підхід дозволяє досягти високої швидкості відгуку інтерфейсу та забезпечується согласованість глобального стану всього вебзастосунку без необхідності ручного адміністрування глобального сховища.

4.3 Забезпечення інформаційної безпеки даних у вебзастосунку

Забезпечення інформаційної безпеки та надійного захисту даних у вебзастосунку «Voicer» є найголовнішим інфраструктурним завданням, яке повинно бути реалізовано на всіх архітектурних рівнях. Цей захист повинен охоплювати передачю HTTP-запитів, обробку даних, взаємодію зі сторонніми API, а також роботу з базами даних та хмарними сховищами.

Першим бар'єром захисту є механізм дедуплікації обробляємих даних. Це важливо у ситуаціях, коли виникає підвищене навантаження на сервер, через що витрачаються ліміти хмарного API Groq. Для запобігання таким помилкам у систему впроваджується алгоритм перевірки унікальності.

Сервер приймає аудіофайл і за допомогою вбудованої бібліотеки crypto обчислює його контрольну суму за алгоритмом SHA-256 (лістинг 4.15). Ця функція трансформує файл будь-якого об'єму у фіксований 64-символьний рядок. Завдяки цьому зміна навіть одного біта в аудіотреку повністю змінює кінцевий хеш, що у подальшому унеможливлює реконструкцію файлу з його відбитка.

```
const fileHash = crypto
.createHash('sha256').update(file.buffer).digest('hex');
```

Лістинг 4.15 – Лістинг хешування за допомогою алгоритма SHA-256

Перед записом файлу в S3-сховище, виконується точковий SQL-запит до таблиці «Sessions», у процесі якого порівнюються згенерований SHA-256 відбиток із тими, що вже існують у колонці hash. Якщо один з об’єктів збігається (лістинг 4.16), то сервер миттєво перериває операцію, блокує завантаження в MinIO та скасовує платний запит до моделі whisper-large-v3-turbo. Клієнту відразу повертається HTTP-відповідь із прапорцем isDuplicate: true та ідентифікатором існуючої сесії з вже готовим текстом. Це захищає сервер від перевантаження та запобігає появі надлишкових записів у таблицях.

```
const existingSession = await this.db
        .query(
          `SELECT id, "fileName", transcription FROM sessions
WHERE hash = $1 LIMIT 1`,
          [fileHash],
        ).catch(() => ({ rows: [] }));
      if (existingSession.rows.length > 0) {
        const found = existingSession.rows[0];
        return {
          isDuplicate: true,
          message: 'You have already downloaded this file.',
          sessionId: found.id,
          fileName: found.fileName,
        };
      }
```

Лістинг 4.16 – Лістинг програми перевірки існування о’єктів у сховищі MinIO

Ще одним з найпопулярніших видів загроз є SQL-ін'єкції. Вони можуть призвести до витоку чи руйнування інформації в БД. Для захисту від атак такого типу, реалізовано механізм параметризованих запитів.

Коли користувач передає ідентифікатор сесії або текстове запитання в методі continueSession, ці дані потрапляють у відповідні змінні sessionId та userQuestion. Під час виконання запитів через драйвер this.db.query рушію PostgreSQL передається зафіксований рядок запиту, де замість реальних

значень вказано спеціальні «змінні-заповнювачі» ($1, $2, $3, $4), а самі призначені для користувача дані передаються окремим масивом (лістинг 4.17).

```
const sessionResult = await this.db.query(
        `INSERT INTO sessions ("id", "fileName", "s3Key",
"s3Url", "transcription", "hash")
         VALUES ($1, $2, $3, $4, $5, $6) RETURNING id`,
        [
          newSessionId,
          file.originalname,
          s3Key,
          s3Url,
          transcriptionText,
          fileHash,
        ],
      );
```

Лістинг 4.17 – Лістинг параметризованого запиту

Такий запит обробляється у два етапи: спочатку компілюється та будується синтаксичний граф самого SQL-шаблону, який жорстко фіксує структуру команди, дозволені таблиці та умови. А після цього значення з масиву параметрів підставляються у відповідні місця, де вони сприймаються рушієм бази даних строго як літерали. І навіть якщо вхідні дані міститимуть шкідливі конструкції, вони будуть записані в базу як звичайний текст повідомлення або запустять безпечний пошук текстового ідентифікатора, який завершиться відсутністю результата. Цей підхід надійно захищає та оптимізує систему.

Для запобігання витоку конфіденційної інформації, в архітектурі сервера повністю виключено пряме використання реальних паролів, токенів та адрес. Такі параметри винесені за межі кодової бази додатка у змінні середовища операційної системи. Вони зберігаються в локальному файлі конфігурації .env (лістинг 4.18). Цей файл внесено до списку виключень .gitignore, тому це унеможливлюється його випадкове потрапляння у публічні репозиторії у GitHub.

```
this.groq = new Groq({ apiKey: process.env.GROQ_API_KEY });
```

Лістинг 4.18 – Лістинг використання змінної середовища GROQ_API_KEY

Централізоване керування цими даними забезпечує системний модуль @nestjs/config. Секретний токен GROQ_API_KEY передається з пам'яті до конструктора SDK Groq для взаємодії з моделями штучного інтелекту. За таким же принципом, наприклад, ізолюються параметри доступу до S3-сховища MinIO, де змінна S3_SECRET_ACCESS_KEY виступає даними входу для запитів.

Такий підхід дозволяє змінювати паролі інфраструктури в будь-який момент без модифікації програмного коду сервера, а також безпечно розгортати проєкт на різних етапах розробки системи (рис. 4.11).

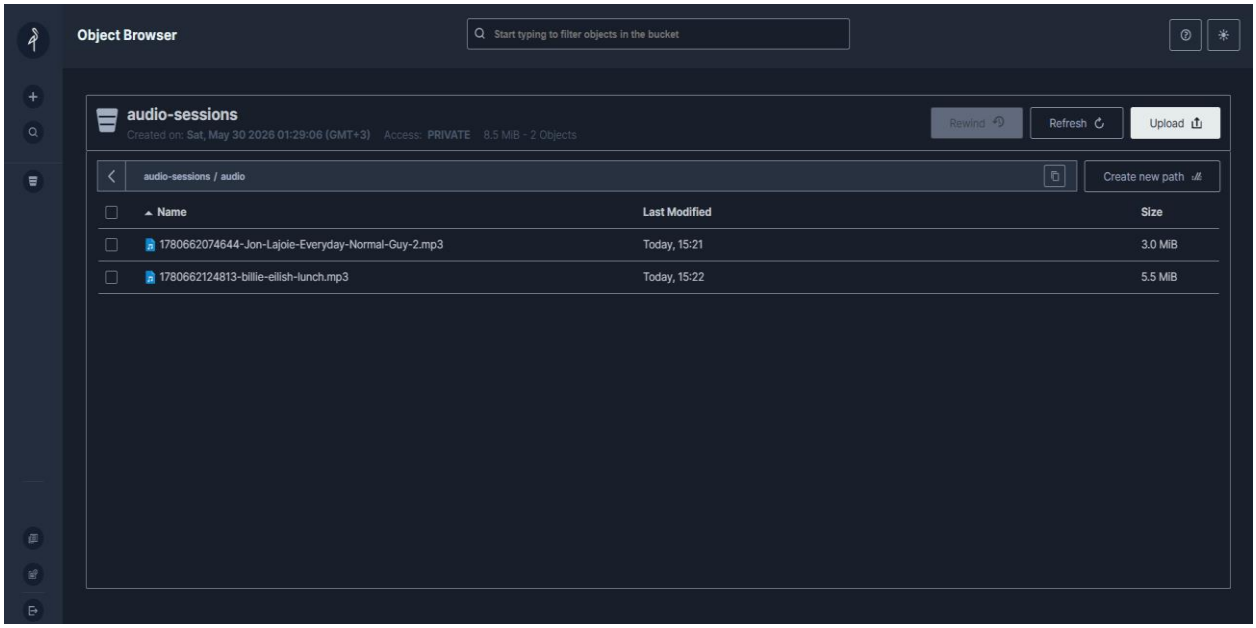


Рисунок 4.11 – Хмарне середовище MinIO зі збереженими аудіофайлами

ВИСНОВКИ

У кваліфікаційній роботі було виконано проєктування та програмну реалізацію вебзастосунку «Voicer», призначеного для автоматичного перетворення аудіоданих у текстовий формат та подальшої взаємодії користувача з результатами транскрибації за допомогою моделей штучного інтелекту. Розроблене рішення орієнтоване на автоматизацію роботи з аудіофайлами, які виникають під час нарад, інтерв'ю, лекцій, консультацій та інших форм усної комунікації.

У першому розділі було проаналізовано предметну область автоматичного розпізнавання мовлення, розглянуто існуючі сервіси транскрибації та визначено їхні функціональні переваги й обмеження. На основі цього було обґрунтовано доцільність створення окремого вебзастосунку з можливістю інтеграції зовнішніх ШІ-моделей.

У другому розділі було виконано вибір програмних засобів і технологій для реалізації клієнтської, серверної та інфраструктурної частин системи. Для серверної частини обрано Node.js, Nest.js і TypeScript, що забезпечують модульність, типізацію та зручність підтримки програмного коду. Для клієнтської частини використано React і Redux Toolkit, які дозволяють створити інтерактивний інтерфейс користувача та централізовано керувати станом застосунку. Для зберігання структурованих даних обрано PostgreSQL, а для аудіофайлів використано MinIO як S3-сумісне об'єктне сховище. Таке поєднання технологій забезпечує розділення відповідальності між компонентами системи та створює основу для подальшого масштабування.

У третьому розділі було спроєктовано архітектуру вебзастосунку, визначено основні програмні компоненти, їхню взаємодію та структуру бази даних. Обрана клієнт-серверна архітектура дозволила відокремити інтерфейс користувача від серверної бізнес-логіки, а також забезпечити незалежну розробку, тестування та модифікацію окремих частин системи. Спроєктована база даних містить сутності, необхідні для збереження користувачів,

аудіофайлів, результатів транскрибації та історії діалогів із ШІ-моделлю. Використання об’єктного сховища для медіафайлів зменшує навантаження на реляційну базу даних і забезпечує більш коректну організацію роботи з великими файлами.

У четвертому розділі було реалізовано серверну та клієнтську частини вебзастосунку. Серверна частина забезпечує реєстрацію та автентифікацію користувачів, завантаження аудіофайлів, збереження метаданих, взаємодію з Groq API, отримання результатів транскрибації та передачу даних клієнтському інтерфейсу. Клієнтська частина реалізує завантаження файлів, перегляд історії обробки, відображення результатів розпізнавання мовлення та взаємодію з чат-модулем. Окрему увагу приділено питанням інформаційної безпеки: захисту облікових даних, використанню токенів доступу, валідації вхідних даних і збереженню конфігураційних параметрів у змінних середовища.

У результаті виконання кваліфікаційної роботи було створено функціональний прототип вебзастосунку, який реалізує повний цикл роботи з аудіоданими: завантаження файлу, збереження в об’єктному сховищі, передавання на обробку до моделі розпізнавання мовлення, отримання текстової транскрипції та подальшу взаємодію з результатом через інтерфейс користувача. Розроблена система підтверджує практичну доцільність інтеграції сучасних моделей штучного інтелекту у вебзастосунки для автоматизації обробки мовленнєвих даних. Подальший розвиток проєкту може бути пов’язаний із розширенням підтримуваних форматів аудіо, додаванням розпізнавання кількох спікерів, удосконаленням механізмів пошуку за транскрипціями та впровадженням інструментів аналітики змісту аудіозаписів.

.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Фаулер М. Архітектура корпоративних програмних додатків : пер. з англ. – Харків : Фабула, 2020. – 544 с.
2. Мартін Р. Чиста архітектура. Мистецтво розроблення програмного забезпечення : пер. з англ. – К. : Фабула, 2019. – 368 с.
3. Клеппман М. Надійні інтелектуальні системи. Проєктування високонавантажених застосунків : пер. з англ. – Львів : Видавництво Старого Лева, 2022. – 512 с.
4. Коваль О. П. Використання концепції поліглотного збереження даних у високонавантажених інформаційних системах / О. П. Коваль, Д. М. Сидоренко // Вісник Одеського національного університету. Серія: Математика і кібернетика. – 2025. – Т. 30, вип. 1. – С. 112–125.
5. Троелсен Е. Мова програмування C# 9 і платформа .NET 5 : у 2 т. [Побудова архітектур та REST API] : пер. з англ. / Е. Троелсен, Ф. Джепікс. – К. : Діалектика, 2021. – Т. 2. – 624 с.
6. Радченко Ю. С. Методи та моделі штучного інтелекту для розпізнавання й транскрибації мовленнєвих сигналів // Сучасна спеціальна техніка. – 2024. – № 4. – С. 89–96.
7. Ткаченко Р. О. Оптимізація інтерфейсів користувача за допомогою сучасних інструментів керування станом у SPA-застосунках / Р. О. Ткаченко, В. В. Кравченко // Проблеми програмування. – 2025. – № 1. – С. 52–61.
8. Мельник І. В. Аналіз методів інтелектуального розпізнавання мовлення в умовах акустичних шумів за допомогою нейромережевих моделей типу Whisper // Штучний інтелект. – 2024. – № 3. – С. 118–127.
9. Бондаренко О. А. Дослідження ефективності асинхронної обробки великих масивів даних у Node.js середовищі / О. А. Бондаренко, М. І. Клименко // Вчені запики Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки. – 2025. – Т. 36 (75), № 2. – С. 88–94.

10. Seemann M. Dependency Injection in .NET / M. Seemann, S. van Deursen. – 2nd ed. – New York : Manning Publications, 2019. – 552 p.
11. Stefanov S. React Up & Running: Building Powerful Web Apps. – 2nd ed. – Sebastopol : O'Reilly Media, 2022. – 250 p.
12. Teixeira P. Distributed Systems with Node.js: Building Enterprise-Ready Backend Applications. – 2nd ed. – Beijing : O'Reilly Media, 2024. – 412 p.