Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет __Комп'ютерної інженерії та управління_____
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти____перший (бакалаврський)_____
(рівень вищої освіти)

Клієнт-серверна системи управління музичними аудіотреками
(тема)

Виконав:
здобувач ____4_____ року навчання,
групи КІУКІ-21-9

_____Артеменко О. Г._____
(прізвище, ініціали)

Спеціальність *123_Комп'ютерна інженерія*

Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____

_____*Комп'ютерна інженерія*_____
(повна назва освітньої прграми)

Керівник ст.вик. Шевченко О.Ю._
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Чумаченко С.В
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет_____Комп'ютерної інженерії та управління _____

Кафедра_____Автоматизації проектування обчислювальної техніки_____

Рівень вищої освіти____перший (бакалаврський)_____

Спеціальність_____123_Комп'ютерна інженерія _____
(шифр і назва)

Тип програми _____Освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма _____Комп'ютерна інженерія_____
(повна назва

ЗАТВЕРДЖУЮ:

Зав. кафедри ___ __
(підпис)

«__05__»___05_____ 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачу_____Артеменко Олексію Геннадійовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи Клієнт-серверна системи управління музичними аудіотреками____
затверджена наказом університету від 21_____05_____2025 р. №_403Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії_13_____06_____2025р.
3. Вихідні дані до роботи _____
Протоколи HTTP/3,формат серіалізації сервера,_____
Динамічний інтерфейс на основі React.js_____
Середовище розробки WebStormIDE2025.1.2_____
Мова програмування JavaScript ES14_____
4. Перелік питань, що потрібно опрацювати в роботі_____
Постановка задачі._____
Аналіз та огляд існуючих альтернатив._____
Розробка структурної схеми._____
Розробка алгоритму роботи серверної та клієнтської частини._____
Аналіз розробленої системи_____

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)

10 слайдів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Приміт ка
1	Видача теми проекту, узгодження і затвердження теми.	03.05.2025 – 05.05.2025	
2	Аналіз предметної області, постановка задачі, вибір стеку технологій.	05.05.2024 – 11.05.2024	
3	Розробка алгоритму, загальної логіки та схеми роботи додатка.	11.05.2024 – 14.05.2024	
4	Розробка програмної частини додатка.	14.05.2024 – 19.05.2024	
5	Тестування та аналіз розробленого продукту, у порівнянні з конкурентами.	19.05.2024 – 20.05.2024	
6	Оформлення пояснювальної записки.	20.05.2024 – 24.05.2024	
7	Перевірка проєкту науковим керівником, допуск до захисту роботи.	24.05.2024 – 13.06.2025	

Дата видачі завдання 05.05.2025р.

Здобувач _____
(підпис)

Керівник роботи _____ Шевченко О.Ю.
(підпис) (посада,прізвище,ініціали)

РЕФЕРАТ

Записка пояснювальна містить: 46 сторінки, 4 рисунка, 6 джерел за переліком посилань.

ПОТОКОВЕ ВІДТВОРЕННЯ МУЗИКИ, ІНТЕРФЕЙС У РЕАЛЬНОМУ ЧАСІ, REACT, NODE.JS, REST API

Метою кваліфікаційної роботи є розробка та впровадження веб-застосунку для потокового прослуховування музики, основною складовою якого є інтерактивна платформа Harmoniq. Цей застосунок дозволяє користувачам створювати облікові записи, завантажувати аудіотреки, формувати персоналізовані плейлисти та насолоджуватися музикою в реальному часі.

Було розроблено архітектуру клієнт-серверної моделі, де фронтенд, з використанням React, взаємодіє з бекендом на Node.js через REST API.

Особливу увагу приділено впровадженню функціоналу реального часу, який дозволяє оновлювати інтерфейс без необхідності перезавантаження сторінки. Це досягається завдяки динамічній функціональності React та ефективній взаємодії з сервером.

REFERENCE

The explanatory note: 46 page, 4 figures, 6 references.

MUSIC STREAMING, REAL-TIME INTERFACE, REACT, NODE.JS, REST API

The purpose of this thesis is to develop and implement a web-based music streaming application, the main component of which is the interactive Harmoniq platform. This application allows users to create accounts, download audio tracks, create personalized playlists, and enjoy music in real time.

The architecture of the client-server model was developed, where the frontend, using React, interacts with the backend on Node.js via the REST API.

Particular attention is paid to the implementation of real-time functionality that allows you to update the interface without the need to reload the page. This is achieved thanks to the dynamic functionality of React and efficient interaction with the server.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

HTTP – Hypertext Transfer Protocol (Протокол передачі гіпертексту);

IoT – Internet of Things (Інтернетречей);

REST – Representational State Transfer (передача репрезентативного стану);

API – Application Programming Interface (Прикладний програмний інтерфейс);

URL - Uniform Resource Locator (Визначник місцезнаходження сайту в мережі Інтернет);

IDE - Integrated development environment (Інтегроване середовище розробки);

JSON - JavaScript Object Notation (Текстовий формат, призначений для зберігання структурованих даних);

CSS - Cascading Style Sheets (Каскадні таблиці стилів);

JWT - JSON Web Token (стандарт токена доступу на основі JSON);

БД - База дан.

ВСТУП

У сучасну цифрову епоху компанії різних масштабів активно переводять свої сервіси в онлайн-простір, що сприяло стрімкому розвитку стрімінгових платформ, зокрема музичних сервісів. Ці платформи стали важливим інструментом для залучення широкої аудиторії, оптимізації бізнес-процесів і зростання доходів. Музичні сервіси, такі як Spotify, YouTubeMusic чи AppleMusic, кардинально змінили спосіб взаємодії користувачів із музикою, надаючи доступ до мільйонів треків у будь-який час і в будь-якому місці. Завдяки зручності доступу, інтуїтивним інтерфейсам і персоналізованим функціям ці платформи не лише зміцнили позиції компаній у сфері розваг, але й відкрили нові можливості для монетизації контенту та розвитку брендів. Таким чином, створення ефективного веб-додатку для музичних сервісів стало ключовим фактором успіху для компаній, які прагнуть залишатися конкурентоспроможними в умовах швидкозмінного цифрового ринку.

За останні роки технології розробки веб-додатків значно вдосконалилися завдяки появі нових інструментів, підходів і методологій. Адаптивний веб-дизайн дозволяє створювати інтерфейси, які однаково зручно працюють на різних пристроях — від смартфонів до настільних комп’ютерів. Хмарні обчислення забезпечують масштабованість і надійність сервісів, дозволяючи обробляти великі обсяги даних і підтримувати стабільну роботу навіть при високих навантаженнях. Технології штучного інтелекту та машинного навчання відіграють ключову роль у створенні персоналізованих рекомендацій, аналізі поведінки користувачів і оптимізації взаємодії з платформою. Завдяки цим інноваціям компанії можуть створювати інтерактивні, зручні та функціональні музичні веб-додатки, які відповідають сучасним вимогам користувачів. Такі платформи підтримують широкий спектр функцій, включаючи потокове відтворення музики, створення та керування плейлистами, інтеграцію із соціальними мережами, а також аналітику для вдосконалення користувацького досвіду.

Музичні веб-додатки активно використовують передові технології для аналізу вподобань користувачів, що дозволяє створювати персоналізований досвід. Алгоритми машинного навчання аналізують історію прослуховувань, вподобання та поведінку користувачів, щоб пропонувати релевантні треки, альбоми чи плейлисти. Такий підхід не лише підвищує залученість користувачів, але й сприяє утриманню аудиторії та зростанню лояльності до платформи. Крім того, інтеграція з соціальними мережами дозволяє користувачам ділитися улюбленою музикою, створювати спільні плейлисти та взаємодіяти з іншими слухачами, що підсилює соціальний аспект сервісу. Впровадження подібних технологій робить музичні платформи більш ефективними, конкурентоспроможними та привабливими для широкої аудиторії, що, у свою чергу, сприяє розвитку бізнесу та зміцненню ринкових позицій.

Метою даної дипломної роботи є комплексний аналіз розробки веб-додатків для музичних платформ із фокусом на сервіси, подібні до Spotify і YouTubeMusic. У роботі будуть розглянуті ключові переваги використання веб-додатків для стрімінгу музики, включаючи їхню доступність, гнучкість і здатність адаптуватися до потреб користувачів. Особлива увага приділятиметься функціональним можливостям таких платформ, як потокове відтворення, персоналізовані рекомендації, інтеграція із зовнішніми сервісами та підтримка різних пристроїв. Крім того, у роботі буде представлено процес створення власного музичного веб-додатку, включаючи вибір технологічного стеку, проєктування архітектури та реалізацію ключових функцій. Дослідження також охопить аналіз сучасних тенденцій у розробці веб-додатків, що дозволить оцінити перспективи розвитку музичних стрімінгових сервісів у майбутньому.

1 ПОСТАНОВКА ЗАДАЧІ ТА СФЕРА ВИКОРИСТАННЯ

1.1 Аналіз предметної області

Розглянемо ключові відмінності між вебсайтом, веб-браузером і веб-додатком.

Веб-додаток – це складне програмне забезпечення, яке функціонує на віддаленому сервері та стає доступним користувачам через веб-інтерфейс, що відкривається за допомогою веб-браузера. Такі додатки пропонують широкий спектр інтерактивних можливостей, включаючи автоматизацію різноманітних процесів, збереження та обробку даних, а також надсилання повідомлень чи сповіщень користувачам. Наприклад, веб-додатки можуть використовуватися для управління проєктами, онлайн-торгівлі, спілкування в соціальних мережах або обробки великих обсягів інформації в реальному часі. Вони дозволяють користувачам взаємодіяти з контентом динамічно, забезпечуючи персоналізований досвід, наприклад, через збереження налаштувань, обробку введених даних чи інтеграцію з іншими сервісами. Веб-додатки зазвичай мають складну серверну та клієнтську архітектуру, що забезпечує їхню високу функціональність і здатність обробляти складні запити в реальному часі.

На противагу цьому, вебсайти зазвичай обмежуються наданням статичного контенту, такого як текстові матеріали, зображення, відеоролики чи аудіофайли. Вони також можуть містити гіперпосилання, що ведуть до інших сторінок або ресурсів у мережі Інтернет. Вебсайти, як правило, не мають вбудованих інструментів для автоматизації завдань, зберігання персональних даних користувачів чи самостійного надсилання повідомлень без додаткових зовнішніх систем. Їхня основна мета – інформувати відвідувачів або презентувати певний контент, наприклад, у вигляді корпоративних сторінок, блогів чи інформаційних порталів. Вебсайти зазвичай простіші за структурою, ніж веб-додатки, і здебільшого призначені

для одностороннього надання інформації, а не для активної взаємодії з користувачем.

Веб-браузер, у свою чергу, є програмним забезпеченням, яке слугує посередником між користувачем і веб-ресурсами. Він дозволяє відкривати як вебсайти, так і веб-додатки, інтерпретуючи код (наприклад, HTML, CSS, JavaScript) і відображаючи його у зрозумілій для користувача формі. Прикладами браузерів є Google Chrome, Mozilla Firefox чи Safari. Без браузера доступ до веб-додатків чи вебсайтів був би неможливим для більшості користувачів. Браузери також забезпечують безпеку, керуючи cookie-файлами, кешем і сертифікатами, а також дозволяють користувачам налаштовувати інтерфейс і взаємодіяти з веб-ресурсами через введення даних, кліки чи інші дії.

Типи веб-додатків, які можуть варіюватися залежно від їхньої функціональності та призначення, детально представлено на відповідному рисунку. Наприклад, це можуть бути прогресивні веб-додатки (PWA), які дозволяють працювати офлайн і мають функціонал, схожий на мобільні додатки, односторінкові додатки (SPA), що завантажують весь контент на одній сторінці для швидкої взаємодії, або багатосторінкові додатки (MPA), які складаються з кількох сторінок із різними функціями. Кожен із цих типів має свої особливості в реалізації та взаємодії з користувачем, що робить їх придатними для різних сценаріїв використання, від простих інформаційних платформ до складних систем управління. Типи веб-додатків зображено на рисунку 1.1.

Рисунок 1.1 – Типи веб-додатків

1.2 Архітектура веб додатків

На сучасному етапі розвитку інформаційних технологій, які стрімко прогресують, статичні веб-сторінки, що колись домінували в інтернет-просторі, значно втратили свою актуальність через обмеженість функціональних можливостей і неспроможність відповідати сучасним вимогам користувачів. Інтернет зазнав кардинальних змін, еволюціонувавши від простих інформаційних сторінок до складних екосистем, які підтримують динамічні веб-додатки з широким спектром інтерактивних функцій. Ці додатки дозволяють користувачам отримувати персоналізований контент, взаємодіяти з інтерфейсами в реальному часі, обробляти великі обсяги даних і використовувати різноманітні сервіси, такі як онлайн-магазини, соціальні мережі чи хмарні платформи. У зв’язку з цим перед початком розробки будь-якого веб-проєкту надзвичайно важливо ретельно проаналізувати й визначитися з типом архітектури веб-додатків, моделлю їхніх компонентів, технологічним стеком, а також підходами до забезпечення безпеки, масштабованості та продуктивності.

Архітектура веб-додатків є основою, що визначає структуру й принципи взаємодії між різними складовими програмного забезпечення, такими як клієнтські додатки, серверні компоненти, бази даних і проміжне

програмне забезпечення (middleware). Ця структура не лише забезпечує стабільну й ефективну роботу кількох додатків одночасно, але й гарантує їхню сумісність, надійність і здатність обробляти численні запити від користувачів без втрати продуктивності. Наприклад, сучасні архітектури, такі як мікросервісна чи сервер лес-архітектура, дозволяють розробникам створювати гнучкі системи, які легко масштабуються залежно від навантаження. На схемі, зображеній на рисунку 1.3, детально представлено архітектуру веб-додатків у процесі завантаження веб-сторінки, де чітко видно послідовність етапів: від формування запиту користувачем до відображення сторінки в браузері. Ця схема допомагає зрозуміти, як взаємодіють клієнтська та серверна частини, а також які технології залучені на кожному етапі.

Процес завантаження веб-сторінки починається, коли користувач вводить URL-адресу в адресний рядок веб-браузера або переходить за посиланням. У цей момент браузер формує HTTP-запит (або HTTPS для безпечного з’єднання), який спрямовується до відповідного веб-ресурсу, розташованого на сервері. Сервер, отримавши цей запит, обробляє його, звертаючись до необхідних компонентів системи, таких як бази даних, кеш або зовнішні API, щоб зібрати потрібні дані. Після цього сервер формує відповідь, яка зазвичай включає набір файлів, таких як HTML-документи для структури сторінки, CSS-файли для стилізації, JavaScript - скрипти для інтерактивності, а також мультимедійні ресурси, як-от зображення чи відео. Ці файли надсилаються назад до браузера користувача через мережу. Браузер, у свою чергу, інтерпретує отримані дані, виконує код, будує DOM (Document Object Model) і відображає запитувану веб-сторінку на екрані. Цей процес дозволяє користувачу не лише переглядати контент, але й активно взаємодіяти з ним, наприклад, заповнювати форми, натискати кнопки чи прокручувати сторінку.

Ключовим елементом у цьому процесі є код, який обробляється веб-браузером і визначає поведінку сторінки. Цей код, зазвичай написаний на мовах програмування, таких як HTML, CSS і JavaScript, містить детальні

інструкції, які керують реакцією браузера на дії користувача. Наприклад, HTML визначає структуру сторінки, CSS відповідає за її візуальне оформлення, а JavaScript забезпечує інтерактивність, дозволяючи створювати динамічні елементи, такі як спливаючі вікна, анімації, оновлення контенту без перезавантаження сторінки чи обробку подій, як-от натискання кнопок. Крім того, сучасні фреймворки, такі як React, Angular чи Vue.js, значно розширюють можливості JavaScript, дозволяючи розробникам створювати складні клієнтські додатки з модульною структурою. Якісний код не лише забезпечує коректне відображення сторінки, але й покращує користувацький досвід, підвищує швидкість завантаження та оптимізує взаємодію з додатком. Наприклад, асинхронні запити (AJAX) дозволяють оновлювати лише окремі частини сторінки, що зменшує навантаження на сервер і робить взаємодію більш плавною.

Окрім цього, архітектура веб-додатків має враховувати такі аспекти, як кросбраузерна сумісність, оптимізація для мобільних пристроїв і забезпечення доступності (accessibility) для користувачів із різними потребами. Наприклад, адаптивний дизайн, реалізований через CSS media queries, дозволяє сторінці коректно відображатися на пристроях із різними розмірами екранів. Безпека також відіграє важливу роль: захист від атак, таких як XSS (Cross-Site Scripting) чи SQL-ін’єкції, є обов’язковим для збереження даних користувачів. Таким чином, архітектура веб-додатків і якісно написаний код є основою для створення сучасних, зручних і безпечних веб-ресурсів, які відповідають очікуванням користувачів і вимогам ринку.

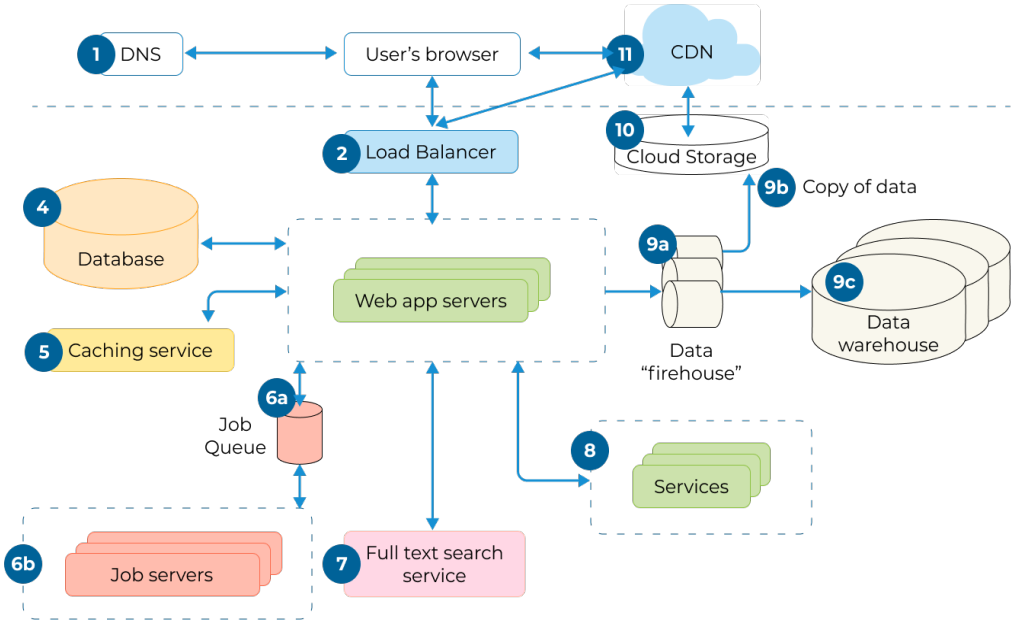

Рисунок 1.3 – Архітектура веб-додатків при завантаженні веб-сторінки

Отже, у сучасному цифровому середовищі архітектура веб-додатків відіграє ключову роль у забезпеченні ефективної взаємодії між додатками та пристроями. Якісна архітектура має враховувати не лише обсяг мережевого трафіку, а й аспекти масштабованості та безпеки. Ці фактори є критично важливими для забезпечення високої продуктивності та зручності для користувачів.

1.3 Функціональність архітектури веб-додатків

Код на стороні клієнта. Цей код виконується у веб-браузері користувача та відповідає за обробку дій користувача. Він забезпечує відображення інтерфейсу та взаємодію з ним на клієнтському боці. Для створення клієнтського коду зазвичай застосовуються технології, такі як HTML, CSS і JavaScript. На відміну від серверного коду, клієнтський код є доступним для перегляду та редагування користувачем. Він взаємодіє із сервером через HTTP-запити.

Код на стороні сервера. Цей код функціонує на сервері та обробляє HTTP-запити, отримані від клієнтської частини. Розробники визначають, як серверний код взаємодіє з клієнтським, задаючи його поведінку. Для програмування серверної частини використовують такі мови, як Python, Java, C#, PHP, Ruby або JavaScript.

Важливо зазначити, що клієнтський і серверний код взаємодіють через HTTP-запити, але клієнтська частина не має прямого доступу до серверних файлів. Обмін даними між ними відбувається через механізм запит-відповідь.

Основні компоненти веб-додатків можна поділити на два типи.

1. UI/UX компоненти. До них належать журнали активності, інформаційні панелі, сповіщення, налаштування, статистичні дані тощо. Ці елементи відповідають за візуальну та інтерактивну складову інтерфейсу веб-додатку. Вони відокремлені від основної архітектури додатку, зосереджуючись на дизайні та зручності використання.

2. Структурні компоненти. Вони поділяються на клієнтську та серверну частини:

– Клієнтський компонент розташований у браузері користувача та створений за допомогою HTML, CSS і JavaScript. Він не потребує спеціальних налаштувань і відповідає за функціональні елементи, з якими взаємодіє користувач.

– Серверний компонент розробляється з використанням мов програмування та фреймворків, таких як Python, Java, .NET, PHP, Node.js або RubyonRails. Він складається з логіки додатку та бази даних. Логіка додатку виконує роль центрального вузла, керуючи обробкою даних і бізнес-логікою, тоді як база даних зберігає постійні дані, необхідні для роботи веб-додатку.

2 АНАЛІЗ ДОСТУПНИХ ЗАСОБІВ ПРОГРАМУВАННЯ

2.1 Порівняння JavaScript та Python як засобів для написання Веб додатків

JavaScript і Python є двома потужними та популярними мовами програмування, які широко застосовуються в розробці веб-додатків, але вони мають різні архітектурні підходи, сильні сторони та оптимальні сценарії використання. JavaScript, завдяки своїй унікальній здатності виконуватися безпосередньо в браузері, є основною мовою для створення інтерактивних клієнтських інтерфейсів, що робить його незамінним інструментом для фронтенд-розробки. Водночас, із появою Node.js, JavaScript також став популярним вибором для серверної розробки, дозволяючи розробникам створювати повноцінні веб-додатки, використовуючи єдину мову для фронтенду та бекенду. Python, навпаки, відомий своєю простотою, читабельністю синтаксису та універсальністю, що робить його ідеальним для швидкої розробки серверної логіки, обробки даних і створення складних бекенд-систем. Хоча Python рідше використовується для фронтенд-розробки, його можна застосовувати в цій сфері через спеціалізовані інструменти, такі як Brython або PyScript, хоча вони менш поширені порівняно з JavaScript.

JavaScript має низку ключових переваг, які роблять його незамінним у веб-розробці. По-перше, це єдина мова програмування, яка підтримується всіма сучасними веб-браузерами без необхідності встановлення додаткових плагінів чи інструментів. Це дозволяє створювати динамічні, інтерактивні інтерфейси, які реагують на дії користувача в реальному часі, наприклад, оновлення вмісту сторінки без перезавантаження чи анімацію елементів. Популярні JavaScript- фреймворки, такі як React, Angular і Vue.js, значно полегшують створення складних клієнтських додатків, надаючи розробникам інструменти для управління станом, компонентною архітектурою та оптимізацією продуктивності. Наприклад, React, створений компанією Meta,

використовується в таких платформах, як Facebook і Instagram, для створення швидких і масштабованих інтерфейсів. Ці фреймворки дозволяють створювати односторінкові додатки (SPA), які забезпечують користувацький досвід, подібний до настільних програм.

Крім фронтенду, JavaScript також активно використовується для серверної розробки завдяки платформі Node.js. Node.js дозволяє виконувати JavaScript поза браузером, використовуючи асинхронну модель на основі подій, що ідеально підходить для обробки великої кількості одночасних запитів, наприклад, у стрімінгових сервісах або чат-додатках. Уявімо проєкт платформи для стрімінгу та прослуховування музики, який використовує стек технологій, що включає React для фронтенду, Redux для управління станом, Node.js та Express для серверної частини, а також JavaScript і TypeScript для написання коду. У такому проєкті JavaScript забезпечує швидке створення інтерактивного інтерфейсу, наприклад, відображення списків відтворення, програвання треків у реальному часі та обробки подій, таких як натискання кнопок. На сервері Node.js обробляє запити, такі як автентифікація користувачів, потокова передача аудіо чи збереження плейлистів у базі даних. TypeScript, який є надбудовою над JavaScript, додає статичну типізацію, що значно зменшує ймовірність помилок у великих проєктах, роблячи код більш передбачуваним і легшим для підтримки.

Однак JavaScript має й певні недоліки. Через свою слабку типізацію (без використання TypeScript) код може бути схильним до помилок, особливо в масштабних проєктах, де велика команда розробників працює над однією кодовою базою. Асинхронна природа JavaScript, хоча й забезпечу є високу продуктивність у задачах реального часу, може ускладнювати розробку через необхідність управління зворотними викликами (callbacks), промісами (Promises) або async/await. Наприклад, неправильне використання асинхронного коду може призвести до так званих "callbackhell" або складнощів із відстеженням помилок. Крім того, JavaScript вимагає ретельного тестування, оскільки браузери можуть по-різному інтерпретувати код, що

створює додаткові виклики для забезпечення кросбраузерної сумісності.

Python, зі свого боку, вирізняється своєю простотою, читабельністю та інтуїтивно зрозумілим синтаксисом, що робить його ідеальним вибором для швидкої розробки серверної логіки. Фреймворки, такі як Django і Flask, дозволяють створювати надійні, безпечні та масштабовані бекенд-додатки з мінімальними зусиллями. Наприклад, Django, завдяки своїй філософії “батарейки в комплекті”, надає вбудовані інструменти для автентифікації, адмін-панелей, роботи з базами даних і захисту від типових вразливостей, таких як SQL-ін’єкції чи міжсайтовий скриптинг (XSS). Flask, у свою чергу, є легшим і більш гнучким фреймворком, що підходить для невеликих проєктів або створення API.

Python особливо ефективний у проєктах, які потребують інтеграції з технологіями обробки даних, машинного навчання чи складних обчислень. Бібліотеки, такі як Pandas, NumPy, SciPyабоTensorFlow, дозволяють легко обробляти великі масиви даних, створювати моделі машинного навчання або виконувати наукові обчислення. Наприклад, у контексті музичної платформи Python може бути використаний для створення API, яке аналізує вподобання користувачів, рекомендує треки на основі алгоритмів машинного навчання або обробляє великі обсяги даних, такі як історія прослуховувань. Django може бути використаний для створення адміністративної панелі для управління контентом платформи, наприклад, додавання нових альбомів чи модерації коментарів.

Однак для фронтенд-розробки Python менш поширений, оскільки він не виконується безпосередньо в браузері. Існують інструменти, такі як Brython або PyScript, які дозволяють запускати Python-код у браузері, але вони мають обмежену продуктивність і не можуть конкурувати з JavaScript у створенні динамічних інтерфейсів. Наприклад, використання PyScript для створення інтерактивного плеєра на музичній платформі було б менш ефективним, ніж використання Reactчи Vue.js, через нижчу швидкість виконання та обмежену підтримку браузерами. Таким чином, у більшості випадків Python

використовується для бекенду, а JavaScript — для фронтенду, що вимагає інтеграційних двох мов у проєктах.

З точки зору продуктивності, JavaScript, особливо в комбінації з Node.js, є кращим вибором для задач реального часу, таких як стрімінг аудіо чи відео, чати або онлайн-ігри. Асинхронна модель Node.js дозволяє обробляти тисячі одночасних підключень із мінімальними ресурсами, що робить його ідеальним для масштабованих систем. Наприклад, платформа для стрімінгу музики, побудована на Node.js, може ефективно обробляти запити від тисяч користувачів, які одночасно слухають музику, завдяки подібно-орієнтованій архітектурі.

Python, хоча й менш ефективний для використання у задач реального часу через свою синхронну природу та інтерпретований характер, виграє в сценаріях, де потрібна швидкість розробки та легкість підтримки коду. Наприклад, створення прототипу API для музичної платформи на Django може зайняти значно менше часу, ніж на Node.js, завдяки високорівневим інструментам Django. Крім того, Python є кращим вибором для проєктів, які інтегруються з технологіями штучного інтелекту чи обробки великих даних. Наприклад, рекомендаційна система для музичної платформи, яка використовує машинне навчання для аналізу вподобань користувачів, може бути легко реалізована за допомогою Python-бібліотек, таких як Scikit-learnабоTensorFlow.

У реальних проєктах JavaScript і Python часто використовуються разом, доповнюючи один одного. Наприклад, у згаданій платформі для стрімінгу музики фронтенд, побудований на React, може взаємодіяти з бекендом на Django через REST API абоGraphQL. JavaScript забезпечує швидке відображення даних на клієнтській стороні, тоді як Python обробляє складну серверну логіку, таку як рекомендації треків чи управління базами даних. Така комбінація дозволяє поєднувати сильні сторони обох мов: швидкість і гнучкість JavaScript для фронтенду та простоту й потужність Python для бекенду.

Екосистема JavaScript є надзвичайно багатою завдяки величезній кількості бібліотек і інструментів, доступних через менеджер пакет ібnpm. Наприклад, розробники можуть використовувати Webpack для збирання проєктів, Jest для тестування або Redux для управління станом. Python також має потужну екосистему, з пакетами, доступними через PyPI, які охоплюють широкий спектр задач, від веб-розробки до аналізу даних. Однак JavaScript має перевагу в контексті веб-розробки завдяки своїй універсальності та прямій інтеграції з браузерами.

JavaScript і Python є потужними інструментами для веб-розробки, але вони мають різні сфери застосування. JavaScript є незамінним для створення інтерактивних фронтенд-інтерфейсів і масштабованих серверних додатків завдяки Node.js, тоді як Python вирізняється простотою та ефективністю в бекенд-розробці, особливо для проєктів, пов’язаних із обробкою даних або машинним навчанням. У проєкті музичної платформи JavaScript ідеально підходить для створення динамічного інтерфейсу та обробки реального часу, тоді як Python може бути використаний для створення надійного API або реалізації складної серверної логіки. Вибір між цими мовами залежить від конкретних вимог проєкту, але в багатьох випадках їх комбінація дозволяє досягти оптимальних результатів.

2.2 Порівняння архітектурних патернів проектування моноліт та микросервиси

Монолітна та мікросервісна архітектури є двома ключовими підходами до проєктування веб-додатків, кожен із яких має свої сильні сторони, слабкі місця та оптимальні сценарії використання. Монолітна архітектура передбачає створення додатка як єдиної цілісної системи, де всі компоненти — фронтенд, бекенд, логіка роботи з базою даних, обробка запитів і бізнес-логіка — тісно пов’язані та розгортаються як єдине ціле. На противагу цьому, мікросервісна архітектура розбиває додаток на набір невеликих, незалежних сервісів, кожен

із яких відповідає за певну функціональність і взаємодіє з іншими через чітко визначені інтерфейси, такі як RESTAPI, GraphQL або повідомлення через брокери, наприклад, RabbitMQ чи Kafka. Вибір між цими архітектурами залежить від багатьох факторів, зокрема розміру проєкту, вимог до масштабованості, складності системи, доступних ресурсів і рівня експертизи команди розробників.

Монолітна архітектура є відносно простою для реалізації, розгортання та управління, що робить її популярним вибором для невеликих або середніх проєктів, а також для стартапів, які прагнуть швидко вивести продукт на ринок. У моноліті весь код додатка зібраний в одній кодовій базі, що значно спрощує розробку, тестування, налагодження та розгортання. Наприклад, розглянемо платформу для стрімінгу музики, побудовану на основі технологічного стеку, що включає React і Redux для фронтенду, Node.js і Express для бекенду, а також HTML5, CSS3, JavaScript, TypeScript, SASS і Vite для оптимізації розробки та зборки проєкту. У такому монолітному додатку фронтенд-компоненти, які відповідають за відображення інтерфейсу користувача (наприклад, списки відтворення, плеєр, сторінка профілю), тісно інтегровані з бекенд-логікою, яка обробляє запити, такі як автентифікація, пошук треків чи потокова передача аудіо. Оскільки всі компоненти знаходяться в одній системі, розробники можуть легко відстежувати потік даних, налагоджувати помилки та вносити зміни без необхідності координувати роботу між кількома сервісами.

Моноліти також мають перевагу в простотіро згортання. Для деплою монолітного додатка достатньо оновити одну кодову базу та розгорнути її на сервері чи в хмарі, наприклад, за допомогою платформ на кшталт AWS ElasticBeanstalk, HerokuабоVercel. У контексті музичної платформи це означає, що оновлення функції, наприклад, додавання можливості створювати плейлисти, потребує лише одного розгортання, яке включає як фронтенд, так і бекенд. Крім того, монолітна архітектура спрощує тестування, оскільки всі компоненти можна перевірити в єдиному середовищі за допомогою

інструментів, таких як Jest для JavaScript-коду чи Cypress для енд-ту-енд тестування. У моноліті також легше забезпечити транзакційну цілісність даних, оскільки вся логіка роботи з базою даних (наприклад, PostgreSQLабоMongoDB) зосереджена в одному місці, що зменшує ймовірність неузгодженостей.

Ще однією перевагою монолітів є менші витрати на інфраструктуру та управління на початкових етапах проєкту. Оскільки весь додаток працює в єдиному середовищі, немає потреби в складних системах оркестрації, таких як Kubernetes, чи в розподілених системах моніторингу, які часто необхідні для мікросервісів. Для невеликих команд із обмеженим досвідом монолітна архітектура дозволяє зосередитися на розробці функціоналу, а не на управлінні складною інфраструктурою.

Проте монолітна архітектура має суттєві обмеження, особливо коли проєкт починає зростати. Кодова база може стати громіздкою, що ускладнює її підтримку та масштабування. Наприклад, у платформі для стрімінгу музики, яка спочатку була невеликим проєктом, додавання нових функцій, таких як рекомендаційна система на основі машинного навчання чи підтримка кількох мов, може призвести до значного ускладнення коду. Розробникам доведеться працювати з однією великою кодовою базою, де зміна одного компонента (наприклад, оновлення API для пошуку треків) може ненавмисно вплинути на інші частини системи, такі як автентифікація чи обробка платежів. Це явище часто називають “технічним боргом”, коли підтримка коду стає дедалі дорожчою.

Ще одним недоліком монолітів є обмежена масштабність. У монолітній архітектурі весь додаток масштабується як єдине ціле, що може бути неефективним. Наприклад, якщо на музичній платформі різко зростає кількість запитів до стрімінгового сервісу, але автентифікація чи каталог треків залишаються менш навантаженими, масштабувати потрібно всю систему, що призводить до надмірного використання ресурсів. Крім того, оновлення монолітного додатка зазвичай вимагає повного перерозгортання,

що може викликати простої або тимчасову недоступність сервісу для користувачів. У великих проєктах це може стати серйозною проблемою, особливо якщо платформа має мільйони активних користувачів.

Мікросервісна архітектура, на відміну від монолітної, пропонує значно більшу гнучкість, масштабованість і незалежність компонентів, що робить її ідеальним вибором для великих, складних і високонавантажених систем. У мікросервісному підході додаток розбивається на набір невеликих, автономних сервісів, кожен із яких виконує одну конкретну функцію та може бути розроблений, розгорнутий і масштабований незалежно від інших. Наприклад, у контексті платформи для стрімінгу музики мікросервісна архітектура могла б включати окремі сервіси для автентифікації користувачів, управління каталогом музики, стрімінгу аудіо, рекомендаційної системи, обробки платежів і аналітики поведінки користувачів. Кожен із цих сервісів може використовувати власну технологію чи мову програмування, що дозволяє командам обирати оптимальні інструменти для конкретних задач.

Однією з ключових переваг мікросервісів є їхня здатність до незалежного масштабування. Наприклад, якщо на музичній платформі зростає попит на стрімінг аудіо, можна масштабувати лише сервіс стрімінгу, не зачіпаючи інші компоненти, такі як автентифікація чи каталог. Це дозволяє ефективніше використовувати ресурси та знижувати витрати на інфраструктуру. Для масштабування мікросервісів часто використовуються хмарні платформи, такі як AWS, Google Cloudабо Azure, а також інструменти оркестрації, наприклад, Kubernetes або DockerSwarm, які автоматизують розгортання та управління контейнерами.

Мікросервіси також дозволяють командам працювати паралельно, що прискорює розробку. У великій команді кожна підкоманда може відповідати за окремий сервіс, наприклад, одна команда розробляє сервіс рекомендацій на Python із використанням бібліотек машинного навчання, таких як TensorFlow, тоді як інша команда створює фронтенд-інтерфейс на JavaScript ізReact. Такий поділ сприяє автономії та зменшує залежності між командами. Крім того,

мікросервіси полегшують оновлення системи, оскільки зміна одного сервісу (наприклад, додавання нової моделі рекомендацій) не вимагає перерозгортання всієї платформи. Це забезпечує безперервну доставку (CI/CD) і дозволяє частіше випускати оновлення без ризиків для користувачів.

Ще однією перевагою мікросервісів є можливість використовувати різні технології для різних сервісів. Наприклад, сервіс стрімінгу може бути написаний на Go для забезпечення високої продуктивності, сервіс обробки даних — на Python для зручної роботи з аналітикою, а фронтенд — на JavaScript із фреймворком Vue.js. Така гнучкість дозволяє оптимізувати кожен компонент для його конкретної задачі. У контексті музичної платформи сервіс рекомендацій може використовувати Python і MongoDB для обробки великих масивів даних, тоді як сервіс автентифікації — Node.js і Redis для швидкої перевірки токенів. Це контрастує з монолітом, де зазвичай використовується однаковий технологічний стек для всіх компонентів.

Незважаючи на численні переваги, мікросервісна архітектура значно складніша в реалізації та управлінні порівняно з монолітною. Оскільки додаток складається з багатьох незалежних сервісів, виникає потреба в управлінні розподіленою системою, що створює додаткові виклики. Наприклад, забезпечення зв’язку між сервісами через API вимагає ретельного проєктування інтерфейсів і обробки помилок. Якщо один із сервісів (наприклад, автентифікація) стає недоступним, це може вплинути на роботу інших залежних сервісів, що вимагає використання таких патернів, як CircuitBreaker чи Retry, для підвищення стійкості системи.

Мікросервіси також ускладнюють тестування, оскільки кожен сервіс потрібно перевіряти окремо, а також у взаємодії з іншими. Наприклад, для тестування музичної платформи потрібно перевірити не лише функціонал сервісу стрімінгу, але й те, як він взаємодіє із сервісом автентифікації чи каталогом треків. Це вимагає створення складних інтеграційних тестів і використання інструментів, таких як Postman або Pact для контрактного тестування. Крім того, мікросервіси потребують розвиненої інфраструктури

для моніторингу та логування, наприклад, використання Prometheus і Grafana для відстеження продуктивності чи ELK Stack для аналізу логів.

Іншим недоліком є вищі витрати на інфраструктуру та управління. Кожен мікросервіс потребує власного сервера, контейнера чи хмарного ресурсу, що збільшує витрати порівняно з монолітом, який працює на одному сервері. У контексті музичної платформи підтримка десятків мікросервісів може вимагати значних інвестицій у хмарні сервіси та інструменти автоматизації. Крім того, мікросервіси потребують складних систем оркестрації, таких як Kubernetes, і високого рівня автоматизації для розгортання, що може бути надмірним для невеликих команд або проєктів.

У контексті платформи для стрімінгу музики монолітна архітектура може бути кращим вибором на ранніх етапах, коли проект ще невеликий, а команда прагне швидко створити прототип і запустити продукт. Наприклад, моноліт на основі Node.js, Express і React дозволяє швидко інтегрувати фронтенд і бекенд, щоб реалізувати базові функції, такі як автентифікація, пошук треків і стрімінг. Такий підхід мінімізує складність інфраструктури та дозволяє команді зосередитися на розробці функціоналу.

Однак, коли платформа починає зростати, а кількість користувачів збільшується до мільйонів, монолітна архітектура може стати перешкодою. Наприклад, якщо попит на стрімінг зростає, а сервіс рекомендацій потребує інтеграції з машинним навчанням, масштабування моноліта може бути неефективним. У такому разі перехід до мікросервісної архітектури дозволяє розділити функціонал на незалежні сервіси, що масштабуються окремо. Наприклад, сервіс стрімінгу може бути розгорнутий на AWS із підтримкою CDN для швидкої доставки контенту, тоді як сервіс рекомендацій працює на Python із TensorFlow для аналізу даних.

Монолітна та мікросервісна архітектури мають свої сильні та слабкі сторони, які визначають їхнє застосування залежно від потреб проєкту. Моноліти є простішими для реалізації та управління, що робить їх ідеальними для невеликих або середніх проєктів, таких як початкові версії музичної

платформи. Однак їхні обмеження в масштабованості та гнучкості можуть стати проблемою для великих систем. Мікросервіси, навпаки, забезпечують високу масштабованість і гнучкість, але потребують складної інфраструктури та ретельного управління. У реальних проєктах часто використовується гібридний підхід, коли проєкт починається як моноліт, а з часом окремі компоненти виносяться в мікросервіси. Для музичної платформи оптимальний вибір залежить від розміру проєкту, кількості користувачів і довгострокових цілей, але обидва підходи можуть бути успішними за правильного застосування.

3 ОСОБЛИВОСТІ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Реалізація Проекту

3.1.1 Реалізація фронтенду

Клієнтська частина є серцем взаємодії користувача з додатком, тому ми приділили особливу увагу створенню зручного, інтуїтивного та візуально привабливого інтерфейсу. Для цього ми обрали React як основну бібліотеку для побудови інтерфейсу, оскільки він дозволяє створювати модульні компоненти, які легко підтримувати та розширювати. Кожен компонент відповідає за певну частину функціоналу: наприклад, є компонент для відтворення музики, який включає кнопки play/pause та прогрес-бар, компонент для пошуку треків, який відображає поле введення та результати, а також компонент для роботи з плейлистами, який дозволяє додавати чи видаляти треки. Такий підхід робить код зрозумілим і легким для тестування.

Для управління станом ми використовуємо ReduxToolkit, який спрощує роботу зі складними даними, такими як список треків, поточний трек чи інформація про автентифікованого користувача. ReduxToolkit дозволяє створювати слайси — окремі модулі стану, які відповідають за різні аспекти програми. Наприклад, слайс auth зберігає дані про користувача та JWT-токен, слайс tracks містить список доступних треків і поточний трек, а слайс playlists відповідає за управління плейлистами. Це дозволяє чітко організувати логіку та уникнути плутанини при роботі з великою кількістю даних.

Стилізація виконана за допомогою SASS, що забезпечує швидке створення сучасного дизайну. Наприклад, навігаційна панель має фон (#1DB954), а основний контент відображається на темному тлі (#191414). SASS дозволяє швидко налаштувати стилі через класи, що економить час і робить код більш читабельним. Для навігації між сторінками (наприклад,

головна, пошук, плейлисти) ми використовуємо ReactRouter, який забезпечує плавні переходи без перезавантаження сторінки.

Ось приклад коду для клієнтської частини, який демонструє базову структуру React-додатку:

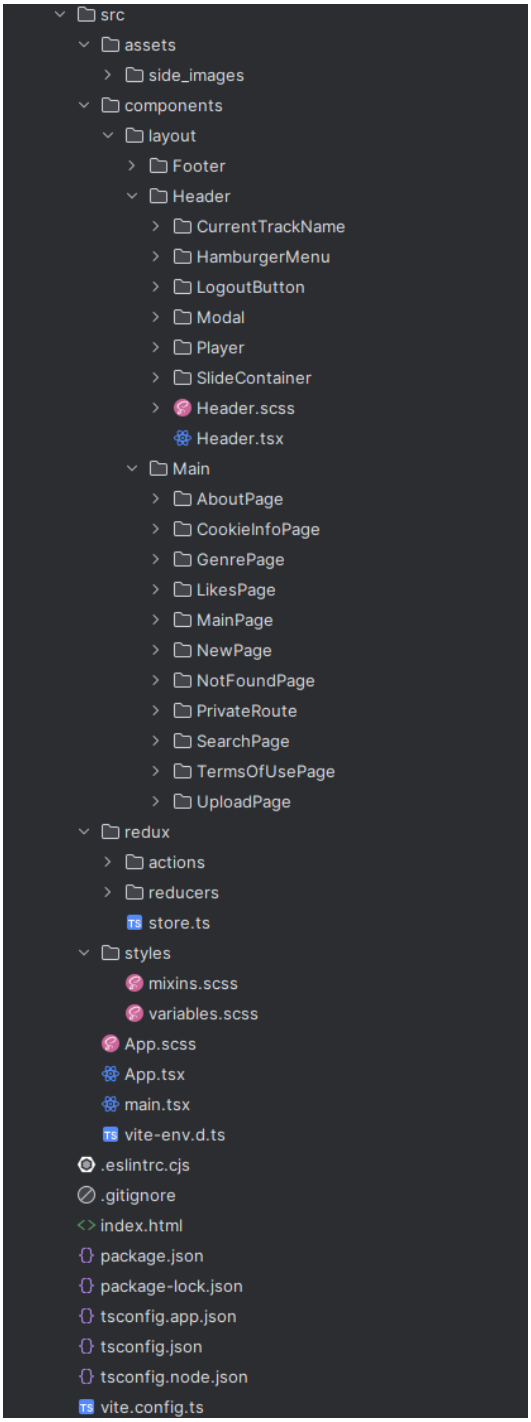

Рисунок 3.1 – Розташування основних файлів для Клієнтської

3.1.2 Реалізація серверної частини

Серверна частина додатку є ключовим компонентом, який відіграє не менш важливу роль, ніж клієнтська частина, оскільки вона відповідає за обробку даних, забезпечення безпеки, управління запитами та надання API для взаємодії з клієнтською частиною. Без надійної серверної інфраструктури неможливо забезпечити стабільну роботу додатку, особливо коли йдеться про обробку великих обсягів даних, таких як аудіофайли, чи захист конфіденційної інформації користувачів. Серверна частина є основою, яка забезпечує функціональність, швидкодію та безпеку, що є критично важливими для сучасних веб-додатків.

Для реалізації серверної частини ми обрали технологію Node.js у поєднанні з фреймворком Express, оскільки ці інструменти ідеально підходять для створення ефективних і масштабованих серверних рішень. Node.js є швидким і асинхронним середовищем виконання JavaScript, яке дозволяє обробляти численні запити одночасно без значного навантаження на сервер. Його асинхронна природа особливо корисна для роботи з потоковими даними, такими як аудіофайли, які потребують швидкої обробки та передачі. Крім того, Node.js має велику екосистему бібліотек, доступних через npm, що значно спрощує розробку та інтеграцію додаткових модулів для реалізації різноманітних функцій.

Фреймворк Express, побудований на основі Node.js, забезпечує зручний і гнучкий спосіб створення REST API. Express дозволяє легко визначати маршрути (endpoints) для обробки HTTP-запитів, що робить його ідеальним вибором для створення серверної логіки. Наприклад, за допомогою Express можна налаштувати маршрути для автентифікації користувачів, управління плейлистами, отримання метаданих треків або передачі аудіофайлів клієнту. Завдяки простоті та гнучкості Express, розробники можуть швидко створювати API, які відповідають потребам клієнтської частини, а також забезпечувати ефективну взаємодію між сервером і клієнтом.

Окрім швидкості та зручності, Node.js і Express дозволяють легко інтегрувати додаткові інструменти для забезпечення безпеки. Наприклад,

можна використовувати бібліотеки для шифрування даних, захисту від атак типу SQL-ін’єкцій або XSS, а також для реалізації механізмів автентифікації, таких як JWT (JSON Web Tokens). Це особливо важливо для додатків, які працюють з особистими даними користувачів, такими як плейлисти чи історія прослуховувань. Завдяки асинхронній обробці запитів, Node.js також забезпечує високу продуктивність навіть при великій кількості одночасних підключень, що робить його ідеальним для потокових сервісів.

Таким чином, вибір Node.js і Express для серверної частини додатку є виправданим завдяки їхній швидкості, гнучкості та широким можливостям для створення сучасних API. Ці технології дозволяють не лише швидко запустити сервер, але й забезпечити його стабільну роботу, безпеку та масштабованість. Вони ідеально підходять для проєктів, які потребують обробки потокових даних, таких як аудіофайли, і дозволяють створювати надійну серверну інфраструктуру, яка відповідає вимогам сучасних веб-додатків.

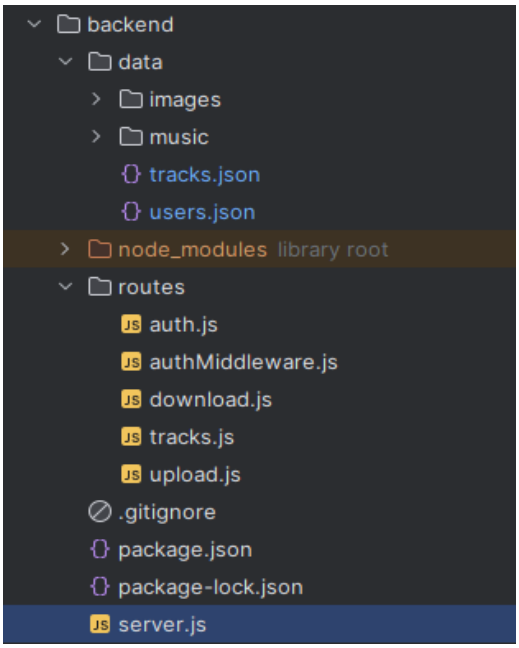

Рисунок 3.2 – Розташування основних файлів для серверної частини

Серверна частина включає кілька ключових маршрутів. Маршрут /api/auth/login відповідає за автентифікацію користувачів, приймаючи ім’я користувача та пароль і повертаючи JWT-токен у разі успіху. JWT використовується для захисту інших маршрутів, таких як створення плейлистів, щоб тільки автентифіковані користувачі могли виконувати ці дії.

Маршрут /api/tracks повертає список доступних треків, які можуть бути використані для пошуку чи відображення в інтерфейсі. У майбутньому цей маршрут можна розширити, додавши фільтрацію за жанром чи виконавцем.

3.2 Огляд проєкту

Розробка веб-додатку, який би став аналогом популярних музичних платформ, таких як Spotify або YouTubeMusic, є амбітним і складним завданням, що вимагає ретельного планування, глибокого аналізу потреб користувачів, вибору оптимальних технологій і створення архітектури, здатної забезпечити високу продуктивність, масштабованість і зручність використання.

Сучасні музичні платформи відіграють ключову роль у цифровому житті мільйонів людей, адже музика стала невід'ємною частиною повсякденності — вона супроводжує нас під час роботи, відпочинку, подорожей, занять спортом чи особистих моментів. Метою цього проєкту є створення зручного, функціонального та сучасного веб-додатку для потокового відтворення музики, який дозволить користувачам слухати улюблені треки, створювати власні плейлисти, шукати музику за різними критеріями та насолоджуватися інтуїтивно зрозумілим інтерфейсом, що забезпечує комфортний і персоналізований музичний досвід.

Додаток має бути не лише приємним у використанні, але й технічно надійним, здатним підтримувати стабільну роботу навіть за умов високого навантаження, наприклад, коли тисячі користувачів одночасно відтворюють музику, створюють плейлисти чи виконують пошук. Основна ідея полягає в тому, щоб поєднати простоту використання з потужними можливостями, які задовольнять як звичайних слухачів, які хочуть швидко отримати доступ до улюбленої музики, так і поціновувачів, які прагнуть мати повний контроль над своїм музичним досвідом, включаючи створення тематичних плейлистів, налаштування рекомендацій чи інтеграцію з іншими сервісами.

Функціонал додатка включає кілька ключових компонентів, які є основою сучасних музичних платформ. Першим і найважливішим елементом є автентифікація користувачів, яка дозволяє створювати персональні облікові записи, входити в систему, зберігати налаштування, улюблені треки, плейлисти та історію прослуховувань. Для забезпечення безпеки автентифікація базується на JSON WebTokens (JWT), що є стандартним і надійним рішенням для захисту даних користувачів. JWT дозволяє генерувати токени після успішного входу, які потім використовуються для авторизації запитів до захищених ресурсів, таких як редагування плейлистів чи доступ до персоналізованих рекомендацій.

Крім того, ми використали бібліотеку bcrypt для шифрування паролів, що забезпечує додатковий рівень захисту від витоку даних. Другим ключовим компонентом є функціонал пошуку треків, який дозволяє користувачам швидко знаходити музику за назвою, виконавцем, альбомом чи жанром. Ця функція є критично важливою, оскільки сучасні музичні бібліотеки можуть містити десятки тисяч композицій, і ефективний пошук значно покращує користувацький досвід. Пошук реалізовано через асинхронні запити до серверної частини, які повертають відфільтровані результати у форматі JSON, що дозволяє відображати їх у реальному часі. Третім важливим елементом є створення та управління плейлистами, що дає користувачам можливість групувати улюблені треки для різних настроїв, подій чи активностей, наприклад, створювати плейлисти для тренувань, роботи чи вечірок. Користувачі можуть додавати, видаляти чи змінювати порядок треків у плейлистах, а також ділитися ними з іншими користувачами через унікальні посилання. Нарешті, відтворення музики реалізовано за допомогою HTML5 Audio API, що забезпечує плавне потокове відтворення аудіо безпосередньо в браузері без необхідності встановлення додаткових плагінів чи програм. HTML5 Audio API дозволяє керувати відтворенням, регулювати гучність, перемотувати треки та підтримувати потокову передачу, що є важливим для забезпечення безперервного прослуховування навіть за умов нестабільного

інтернет-з’єднання. Для підвищення якості відтворення ми також передбачили буферизацію аудіо, щоб мінімізувати затримки, і підтримку адаптивного бітрейту, що дозволяє автоматично підлаштовувати якість звуку залежно від швидкості з’єднання.

Для реалізації цього проєкту було обрано сучасний технологічний стек, який поєднує перевірені інструменти для створення продуктивного, гнучкого та масштабованого веб-додатку. На клієнтській стороні використано React, бібліотеку для створення користувацьких інтерфейсів, яка дозволяє розробляти модульні та повторно використовувані компоненти, що значно спрощує розробку, підтримку та масштабування інтерфейсу. Наприклад, компоненти для плеєра, пошуку, плейлистів і профілю користувача розроблені як окремі модулі, що ізолюють функціонал і полегшують їх тестування та оновлення. React також підтримує швидке оновлення інтерфейсу завдяки віртуальному DOM, що зменшує час рендерингу та забезпечує плавну взаємодію з користувачем. Для управління станом додатка використано ReduxToolkit, який спрощує роботу зі складною логікою, такою як відстеження поточного треку, стану автентифікації чи результатів пошуку.

ReduxToolkit пропонує інструменти, як-от createAsyncThunk і createSlice, що дозволяють ефективно обробляти асинхронні запити до сервера та керувати станом із мінімальною кількістю шаблонного коду. Наприклад, коли користувач додає трек до плейлиста, ReduxToolkit синхронізує стан між клієнтською частиною та сервером, забезпечуючи актуальність даних. Для стилізування інтерфейсу використано Tailwind CSS, утилітарний фреймворк, який дозволяє швидко створювати адаптивні та естетично привабливі дизайни. Tailwind CSS забезпечує гнучкість у стилізуванні, дозволяючи адаптувати інтерфейс до різних пристроїв, таких як смартфони, планшети чи настільні комп’ютери, і підтримує створення темної та світлої тем для підвищення комфорту користувачів.

Наприклад, кнопки плеєра стилізовані з використанням утиліт Tailwind, що забезпечують однаковий вигляд і поведінку на всіх платформах.

На серверній стороні використано Node.js із фреймворком Express для створення REST API, яке обробляє запити від клієнта, такі як автентифікація, пошук треків, створення плейлистів чи збереження налаштувань. Node.js, завдяки своїй асинхронній подієво-орієнтованій архітектурі, ідеально підходить для стрімінгових платформ, де необхідно обробляти тисячі одночасних запитів, наприклад, коли користувачі одночасно відтворюють музику чи шукають треки. Express спрощує створення маршрутів і обробку HTTP-запитів, дозволяючи швидко реалізувати API з чіткою структурою. Наприклад, ендпоінт /api/tracks повертає список треків у форматі JSON, тоді як /api/auth/login обробляє автентифікацію та повертає JWT-токен.

Для зберігання даних у прототипі використано JSON-файли, що є простим і ефективним рішенням на ранніх етапах розробки, оскільки не вимагає налаштування складних баз даних. Наприклад, бібліотека треків зберігається у файлі tracks.json, який містить масив об’єктів із метаданими, такими як назва, виконавець, жанр і URL аудіофайлу, тоді як плейлисти зберігаються у файлі playlists.json із даними про ідентифікатор користувача та список треків. Хоча JSON-файли зручні для прототипу, для продакшен-версії ми рекомендуємо використовувати реляційні бази даних, як-от PostgreSQL, або NoSQL-бази, як-от MongoDB, для підтримки великих обсягів даних і складних запитів, що забезпечить кращу масштабованість і швидкість доступу до даних. Такий підхід дозволяє створити гнучку та масштабовану систему, яка може розвиватися в майбутньому, додаючи нові функції, такі як персоналізовані рекомендації на основі машинного навчання, соціальна взаємодія, наприклад, можливість стежити за плейлистами інших користувачів, або інтеграція з зовнішніми сервісами, такими як соціальні мережі чи платіжні системи для преміум-підписок.

Розробка такого додатка пов’язана з низкою викликів, які потребують ретельного планування та технічних рішень. Одним із ключових викликів є забезпечення безпеки даних користувачів. Використання JWT і bcrypt дозволяє захистити автентифікацію та паролі, але ми також впровадили захист

від типових вразливостей, таких як міжсайтовий скриптинг (XSS) і атаки на API, використовуючи бібліотеки, як-от helmet для Express, які додають заголовки безпеки, і валідацію вхідних даних для запобігання ін’єкціям. Ще одним викликом є оптимізація потокового відтворення, оскільки аудіофайли можуть бути великими, а затримки в передачі даних погіршують користувацький досвід. Для цього ми використали буферизацію аудіо через HTML5 Audio API та розглянули можливість інтеграції з CDN, наприклад, AWS CloudFront, для прискорення доставки контенту до користувачів у різних регіонах. Кросбраузерна сумісність також була важливим аспектом, тому ми протестували додаток у різних браузерах (Chrome, Firefox, Safari) і на різних пристроях, використовуючи сучасні стандарти HTML5 і CSS3, а також інструменти, як-от Jest для юніт-тестів і Cypress для енд-ту-енд тестування, щоб гарантувати коректну роботу всіх функцій. Для оптимізації продуктивності ми застосували техніки ледачого завантаження (lazyloading) для компонентів і ресурсів, таких як обкладинки альбомів, а також кодосплітінг, щоб зменшити розмір початкового бандлаJavaScript і прискорити завантаження сторінок. На серверній стороні ми передбачили можливість масштабування через розгортання в хмарі, наприклад, на AWS ElasticBeanstalk або Vercel, що дозволяє автоматично збільшувати ресурси залежно від навантаження.

Перспективи розвитку додатка включають додавання нових функцій для підвищення зручності та залученості користувачів. Наприклад, інтеграція технологій машинного навчання, таких як TensorFlow.js, може покращити персоналізовані рекомендації, аналізуючи історію прослуховувань і вподобання користувачів у реальному часі. Впровадження WebSocket дозволить реалізувати функції реального часу, такі як оновлення плейлистів або чат між користувачами, що зробить платформу більш інтерактивною.

Підтримка офлайн-режиму за допомогою ServiceWorker дасть змогу користувачам прослуховувати збережені плейлисти без підключення до інтернету, що особливо корисно в подорожах. Крім того, інтеграція з

хмарними сервісами, такими як AWS S3 для зберігання аудіофайлів, або використання Redis для кешування часто запитуваних даних, таких як популярні треки, може значно підвищити продуктивність і масштабованість. У майбутньому додаток може бути розширений до рівня повноцінної платформи з підтримкою преміум-підписок, соціальних функцій, таких як можливість ділитися плейлистами в соціальних мережах, або інтеграції з голосовими асистентами для управління відтворенням голосом. Розроблена система створює міцну основу для створення конкурентоспроможної музичної платформи, яка поєднує простоту, функціональність і сучасні технології, забезпечуючи приємний і персоналізований досвід для користувачів.

3.3 Архітектура системи

Архітектура системи є основою для створення надійного, ефективного та масштабованого веб-додатку, який здатен відповідати вимогам сучасних користувачів, включаючи швидке завантаження сторінок, інтуїтивно зрозумілий інтерфейс і стабільну роботу навіть за умов високого навантаження. У цьому проєкті ми прагнули розробити чітку, логічно структуровану та гнучку архітектуру, яка забезпечує чітке розділення відповідальностей між клієнтською та серверною частинами, полегшує підтримку коду, сприяє масштабуванню та дозволяє легко додавати нові функції без значних змін у наявній кодовій базі. Система поділена на два ключові модулі: клієнтська частина, яка відповідає за взаємодію з користувачем і відображення інтерфейсу, та серверна частина, яка обробляє бізнес-логіку, запити та управління даними. Такий поділ є стандартним для сучасних веб-додатків, але в нашому випадку ми приділили особливу увагу модульності, оптимізації продуктивності та гнучкості, щоб забезпечити швидке завантаження сторінок, плавну взаємодію з користувачем і можливість розширення функціоналу в майбутньому. Клієнтська частина додатка побудована з використанням React, сучасної бібліотеки для створення

користувацьких інтерфейсів, яка дозволяє створювати компонентно-орієнтовані додатки з повторним використанням коду, легкістю підтримки та можливістю швидкого додавання нових функцій.

Організував клієнтську частину за принципом модульності, де кожен компонент відповідає за окрему частину інтерфейсу, що забезпечує ізоляцію функціоналу та спрощує тестування й масштабування. Наприклад, у нашому веб-додатку для потокового відтворення музики ми розробили окремий компонент для плеєра, який відображає інформацію про поточний трек, таку як назва, виконавець і тривалість, а також містить кнопки управління відтворенням, такі як “Play”, “Pause”, “Next” і “Previous”. Цей компонент інтегрований із ReduxToolkit для управління станом відтворення, що дозволяє синхронізувати статус плеєра з іншими частинами додатка, наприклад, із плейлистами. Аналогічно, компонент пошуку дозволяє користувачам вводити запити, відображати результати у вигляді списку треків або альбомів і фільтрувати їх за різними параметрами, такими як жанр чи виконавець, використовуючи асинхронні запити до серверної частини через API для отримання даних у реальному часі.

Компонент плейлистів забезпечує створення, редагування та видалення плейлистів, а також відображення їхнього вмісту, дозволяючи користувачам додавати треки, змінювати їхній порядок або ділитися плейлистами з іншими. Компонент профілю користувача відображає персональну інформацію, таку як ім’я, історію прослуховувань і налаштування, а також дозволяє змінювати параметри, наприклад, тему інтерфейсу (світлу чи темну). Структура клієнтської частини проєкту чітко організована для полегшення розробки та підтримки: у папці public розташовано основний HTML-файл (index.html), який є точкою входу для програми та забезпечує підключення всіх необхідних ресурсів, таких як стилі CSS і скрипти JavaScript, тоді як у папці src зібрано весь код фронтонду, включаючи React-компоненти, логіку ReduxToolkit і головний файл App.jsx, який об’єднує всі компоненти в єдину структуру,

визначає маршрутизацію за допомогою бібліотеки ReactRouter і забезпечує рендеринг основного інтерфейсу.

Стилі, реалізовані за допомогою SASS, дозволяють створювати адаптивні та модульні дизайни, що забезпечують однаковий вигляд інтерфейсу на різних пристроях, від настільних комп'ютерів до мобільних телефонів. Для оптимізації продуктивності клієнтської частини ми використали інструмент Vite, який забезпечує швидку зборку проєкту та гаряче оновлення (hotmodulereplacement) під час розробки, що значно прискорює ітерації розробки. Крім того, ми застосували техніки ледачого завантаження (lazyloading) для компонентів і ресурсів, таких як зображення обкладинок альбомів, щоб зменшити час завантаження сторінок, що особливо важливо для користувачів із повільним інтернет-з'єднанням, оскільки дозволяє швидше відобразити основний інтерфейс, а додаткові ресурси завантажувати за потреби. ReduxToolkit спрощує управління станом додатка, наприклад, збереження стану плеєра, результатів пошуку чи даних користувача, завдяки вбудованим інструментам, таким як createAsyncThunk, що дозволяє легко обробляти асинхронні запити до API, забезпечуючи плавну взаємодію з користувачем. Серверна частина додатка побудована на основі Node.js із фреймворком Express, що забезпечує швидке створення ефективного та масштабованого REST API. Node.js, завдяки своїй асинхронній подієво-орієнтованій архітектурі, ідеально підходить для обробки великої кількості одночасних запитів, що є критично важливим для стрімінгових платформ, де користувачі можуть одночасно відтворювати музику, створювати плейлисти або шукати треки. Express спрощує створення маршрутів і обробку HTTP-запитів, дозволяючи швидко реалізувати необхідний функціонал.

Серверна частина включає маршрути для обробки автентифікації (/api/auth), треків (/api/tracks) і плейлистів (/api/playlists). Наприклад, ендпоінт/api/auth/login приймає дані користувача (email і пароль), перевіряє їх за допомогою bcrypt і повертає JWT-токен для подальшої авторизації, який використовується для захисту інших маршрутів, таких як створення

плейлистів, забезпечуючи безпечний доступ лише для авторизованих користувачів. Маршрути для треків дозволяють отримувати список треків, фільтрувати їх за жанром, виконавцем чи назвою, а також отримувати метадані окремого треку, такі як тривалість, обкладинка альбому чи інформація про автора, наприклад, GET-запит до /api/tracks?genre=rock повертає список треків у жанрі рок у форматі JSON. Маршрути для плейлистів забезпечують створення, редагування, видалення та отримання плейлистів, дозволяючи авторизованим користувачам створювати нові плейлисти з вибраними треками через POST-запит до /api/playlists із JWT-токеном у заголовку.

Для зберігання даних у прототипі ми використовували JSON-файли, що є оптимальним рішенням на ранніх етапах розробки завдяки простоті реалізації та відсутності необхідності налаштування складних баз даних: бібліотека треків зберігається у файлі tracks.json, який містить масив об’єктів із метаданими, такими як назва, виконавець, жанр і URL аудіофайлу, тоді як плейлисти зберігаються у файлі playlists.json, де кожен плейлист представлений як об’єкт із ідентифікатором користувача та списком треків. Такий підхід дозволяє швидко протестувати функціонал, але для продакшен-версії ми рекомендуємо використовувати реляційні бази даних, такі як PostgreSQL, або NoSQL-бази, як MongoDB, для забезпечення масштабованості та швидкого доступу до даних.

Безпека серверної частини забезпечується кількома рівнями захисту: JWT-аутентифікація перевіряє права доступу до захищених маршрутів, паролі користувачів шифруються за допомогою бібліотеки bcrypt, що захищає їх від витоку в разі компрометації даних, а захист від типових вразливостей, таких як міжсайтовий скриптинг (XSS) і SQL-ін’єкції (у разі використання бази даних), забезпечується використанням бібліотек, як-от helmet для Express, які додають додаткові заголовки безпеки. Архітектура API розроблена з урахуванням гнучкості та розширюваності, дозволяючи легко додавати нові ендпоінти, наприклад, /api/recommendations для персоналізованих

рекомендацій на основі історії прослуховувань або /api/analytics для збору статистики, наприклад, найпопулярніших треків, без значних змін у структурі серверної частини завдяки модульній організації коду, де кожен маршрут визначений у окремому файлі, а логіка обробки запитів ізольована від основного додатка. Взаємодія між клієнтською та серверною частинами відбувається через HTTP-запити, що забезпечує чітке розділення відповідальностей: клієнтська частина відповідає за відображення даних і взаємодію з користувачем, тоді як серверна частина обробляє запити, виконує бізнес-логіку та повертає дані у форматі JSON. Наприклад, коли користувач вводить запит у пошуковій панелі, клієнтська частина формує GET-запит до ендпоінта /api/tracks, передаючи параметри пошуку, такі як назва треку чи жанр, а сервер обробляє запит, виконує фільтрацію в базі даних або JSON-файлі та повертає список треків у форматі JSON, який клієнтська частина відображає у вигляді списку результатів.

Для забезпечення швидкої та стабільної взаємодії ми оптимізували запити, використовуючи асинхронні операції: клієнтська частина використовує бібліотеку axios для виконання HTTP-запитів, а ReduxToolkit обробляє асинхронні дії через createAsyncThunk, що дозволяє ефективно керувати станом завантаження даних, наприклад, відображати спінер під час очікування відповіді від сервера. Крім того, ми впровадили кешування даних на клієнтській стороні за допомогою ReduxToolkit, щоб зменшити кількість запитів до сервера для часто використовуваних даних, таких як список популярних треків. Ми також розглянули можливість використання технологій реального часу, таких як WebSocket, для оновлення даних, наприклад, коли користувач додає трек до плейлиста, а зміни мають миттєво відобразитися в інтерфейсі всіх підключених клієнтів, хоча в прототипі ми обмежилися HTTP-запитами, WebSocket може бути доданий у майбутніх версіях для реалізації інтерактивних функцій, таких як чат або спільне редагування плейлистів.

Особливу увагу в архітектурі приділено оптимізації продуктивності та масштабованості: на клієнтській стороні ми використовували техніки, такі як ледаче завантаження компонентів і кодосплітінг, щоб зменшити розмір початкового бандла JavaScript і прискорити завантаження сторінок, тоді як на серверній стороні Node.js із його подієво-орієнтованою моделлю дозволяє ефективно обробляти велику кількість одночасних підключень, що є критично важливим для стрімінгових платформ. Для підвищення масштабованості ми передбачили можливість розгортання серверної частини в хмарі, наприклад, на AWS ElasticBeanstalk або Vercel, що дозволяє автоматично масштабувати ресурси залежно від навантаження. Кросбраузерна сумісність була забезпечена шляхом тестування додатка в різних браузерах (Chrome, Firefox, Safari) і на різних пристроях (десктоп, планшет, смартфон), а використання сучасних стандартів HTML5 і CSS3, а також адаптивного дизайну за допомогою SASS, гарантувало коректне відображення інтерфейсу на всіх платформах. Для тестування ми використовували інструменти, такі як Jest для юніт-тестів React-компонентів і Cypress для енд-ту-енд тестування, що дозволило виявити та виправити потенційні проблеми до розгортання.

Розроблена архітектура створює міцну основу для подальшого розвитку додатка: у майбутньому можна додати підтримку офлайн-режиму за допомогою технології ServiceWorker, що дозволить користувачам прослуховувати збережені плейлисти без підключення до інтернету, а інтеграція з технологіями машинного навчання, такими як TensorFlow.js, може покращити персоналізовані рекомендації, аналізуючи вподобання користувачів у реальному часі. Використання хмарних сервісів, таких як AWS S3 для зберігання аудіофайлів або CloudFront як CDN, може значно прискорити доставку контенту до користувачів у різних регіонах. Архітектура також дозволяє легко перейти від JSON-файлів до повноцінної бази даних, такої як MongoDB або PostgreSQL, для підтримки великих обсягів даних і складних запитів, наприклад, MongoDB може бути використана для зберігання метаданих треків і плейлистів, тоді як Redis може застосовуватися для

кешування часто запитуваних даних, таких як популярні треки, що підвищить продуктивність і зменшить навантаження на сервер. Розроблена архітектура забезпечує чітке розділення відповідальностей між клієнтською та серверною частинами, високу продуктивність і гнучкість для подальшого розвитку, модульна структура клієнтської частини, побудована на React і ReduxToolkit, дозволяє легко додавати нові функції, тоді як серверна частина на Node.js і Express забезпечує швидке та безпечне оброблення запитів, а використання JSON, JWT і bcrypt гарантує ефективне управління даними та безпеку, забезпечуючи міцну основу для створення сучасної стрімінгової платформи з потенціалом для масштабування та інтеграції нових технологій.

ВИСНОВКИ

В результаті написання кваліфікаційної роботи було розроблено та досліджено веб-додаток для потокового відтворення музики, із використанням сучасного технологічного стеку: React і RTK Toolkit на фронтенді та Node.js на серверній частині. Проведене дослідження продемонструвало високу ефективність обраних технологій для створення функціональних, масштабованих і користувацько-орієнтованих рішень, що забезпечують швидке відтворення аудіоконтенту, персоналізовані рекомендації та інтуїтивно зрозумілий інтерфейс. Реалізація REST API, інтеграція з JSON для зберігання метаданих і плейлистів, а також використання асинхронних операцій дозволили оптимізувати обробку великих обсягів даних і забезпечити стабільну роботу системи навіть за умов високого навантаження.

У процесі розробки було подолано низку викликів, зокрема забезпечення безпеки даних, оптимізація потокової передачі та досягнення кросбраузерної сумісності. Ці проблеми вирішувалися шляхом впровадження JWT-аутентифікації, шифрування даних за допомогою bcrypt. Застосування RTK Toolkit значно спростило управління станом додатка, що дозволило ефективно обробляти асинхронні запити та забезпечити плавну взаємодію з користувачем.

Розробка та дослідження веб-додатка для потокового відтворення музики підтвердили високу ефективність і доцільність використання сучасного технологічного стеку, що включає React, RTK Toolkit і Node.js. Ці технології дозволили створити масштабовану, продуктивну та користувацько-орієнтовану систему, яка відповідає сучасним вимогам до веб-додатків для стрімінгу. Зокрема, React забезпечив швидке створення компонентного інтерфейсу, що адаптується до потреб користувачів, дозволяючи реалізувати інтуїтивно зрозумілі елементи управління, такі як плеєр, списки відтворення та персоналізовані рекомендації. RTK Toolkit, як інструмент управління

станом, значно оптимізував обробку асинхронних запитів, зменшивши складність коду та підвищивши продуктивність фронтенду. Node.js, зі своєю асинхронною архітектурою, виявився ідеальним для серверної частини, забезпечуючи швидку обробку великої кількості одночасних запитів, що є критично важливим для стрімінгових платформ із високим навантаженням.

Дослідження також показало, що впровадження REST API з використанням JSON для зберігання метаданих і плейлистів забезпечило гнучкість і ефективність у роботі з даними. Асинхронні операції, реалізовані через проміси та async/await, дозволили оптимізувати потокову передачу аудіоконтенту, зменшивши затримки та забезпечивши стабільну роботу навіть за умов пікового навантаження. Безпека системи була посилена завдяки JWT-аутентифікації, яка надійно захищала доступ до даних користувачів, і шифруванню паролів за допомогою bcrypt, що забезпечило захист від несанкціонованого доступу. Кросбраузерна сумісність, досягнута шляхом ретельного тестування та використання сучасних стандартів HTML5 і CSS3, гарантувала коректну роботу додатка в різних браузерах, що є важливим для забезпечення широкої доступності платформи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. JavaScript - MDN WebDocs [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

2. React [Електронний ресурс] – Режим доступу до ресурсу: https://react.dev/learn.

3. Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/docs/latest/api/.

4. RTK Query [Електронний ресурс] – Режим доступу до ресурсу: https://redux-toolkit.js.org/rtk-query/overview.

5. Express.js [Електронний ресурс] – Режим доступу до ресурсу: https://devdocs.io/express/.

6. Gamble, J., Helm, R., Johnson, R., &Vlissides, J. (2021). DesignPatterns: ElementsofReusableObject-OrientedSoftware. Addison-Wesley Professional.