Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет інформаційних радіотехнологій та технічного захисту інформації
(повна назва)

Кафедра медіаінженерії та інформаційних радіоелектронних систем
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти *другий (магістерський)*
(позначення документа)

Розробка веб-додатку для збереження інформації про музичні групи та їхні треки
(тема)

Виконав:
студент 2 курсу, групи СТМм-22-1
Олег Нікулін
(прізвище, ініціали)

Спеціальність 171 Електроніка
(код і повна назва спеціальності)

Тип програми *освітньо-професійна*
(освітньо-професійна або освітньо-наукова)

Освітня програма *Системи, технології і комп'ютерні засоби мультимедіа*
(повна назва освітньої програми)

Керівник *доц. Роман Цехмістро*
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Володимир КАРТАШОВ
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційних радіотехнологій та технічного захисту інформації
Кафедра Медіаінженерії та інформаційних радіоелектронних систем

Рівень вищої освіти другий (магістерський)
Спеціальність 171 Електроніка
(код і повна назва)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма "Системи, технології і комп'ютерні засоби мультимедіа"

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студентові *Нікуліну Олегу Ігоревичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Розробка веб-додатку для збереження інформації про музичні групи та їхні треки*
затверджена наказом по університету від *" 20 " 11 2023 р. № 1371 СТ*
2. Термін подання студентом роботи *20.12.2023 р.*
3. Вихідні дані до проекту (роботи)
1. Аналітичний огляд предметної області
2. Аналіз інструментів для створення Full-stack WEB застосунків
3. Розробка застосунку з використанням сучасних WEB технологій
4. Перелік питань, що потрібно опрацювати в роботі
ВСТУП
1. Аналіз предметної області
2. Вибір технологій та стеку
3. Архітектура застосунку
4. Розробка функціоналу
ВИСНОВКИ
ПЕРЕЛІК ПОСИЛАНЬ
ДОДАТКИ
5. Перелік графічного матеріалу із зазначенням обов'язкових креслеників, схем, плакатів, комп'ютерних ілюстрацій.
1. Назва теми і роботи .
2. Постановка задачі .
3. Актуальність дослідження ПЗ .
4. Вимоги бізнесу .
5. Вибір технологій та стеку .
6. Архітектура за стосунку .
7. Обрана архітектура застосунку .
8. Розробка серверної частини .
9. Розробка клієнтської частини .
10. Вигляд основної таблиці .
11. Додавання гуртів та треків .
12.Редагування треків .

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термин виконання етапів роботи	Примітка
1.	Аналіз предметної області	10.9.23–8.10.23	
2.	Вибір технологій та стеку	9.10.23–18.10.23	
3.	Архітектура застосунку	19.10.23–19.11.23	
4.	Розробка функціоналу	20.11.23–15.12.23	
5.	Перевірка керівником	16.12.23-24.12.23	
6.	Перевірка на академічний плагіат	24.12.23-26.12.23	
7.	Перевірка завідувачем кафедри, рецензування	27.12.23-10.1.24	

Дата видачі завдання *20.11.2023 р.*

Студент _____ *Олег НІКУЛІН*
(підпис)

Керівник роботи _____ *доц. Цехмістро Р.І.*
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи має: стр. 157, рис. 131, джерел 46.

JS, JSX, ВЕБ-ДОДАТОК, ORM, КОМПОНЕНТИ, АЛГОРИТМ, АРХІТЕКТУРА, РОЗРОБКА

Об'єкт дослідження – комплексний аналіз, розробка та оцінка технічних, архітектурних та безпекових аспектів проекту для забезпечення його ефективного функціонування.

Предмет дослідження – процес розробки та реалізації Full-stack веб-застосунку, з фокусом на технічних аспектах, архітектурних рішеннях, безпеці, тестуванні, інтеграції, деплойменті та оптимізації для забезпечення його стабільної та продуктивної роботи.

Мета роботи - розробка та імплементація Full-stack веб-застосунку, а також глибокий аналіз технічних, архітектурних, безпекових та продуктивності аспектів для створення ефективного та функціонального веб-додатку.

Методи дослідження – аналіз літератури, написання коду, тестування та аналіз коду.

У цій роботі створений застосунок є музичною платформою, де користувачі можуть створювати та управляти інформацією про гурти та їх треки. Основні можливості включають додавання нових гуртів, вказання інформації про гурт (назва, дата заснування, місцезнаходження, тощо) та додавання треків до кожного гурту. Застосунок також надає можливість сортування, пагінації та видалення треків через відповідні таблиці та модальні вікна. Взаємодія з сервером відбувається за допомогою Apollo Client та GraphQL-запитів. Застосунок використовує Firebase для зберігання зображень гуртів. Реалізо-

вана також можливість видалення треків гурту через відповідне модальне вікно з підтвердженням.

ABSTRACT

The explanatory note of the qualification work has: p. 157, fig. 131, 46 sources.

JS, JSX, WEB APPLICATION, ORM, COMPONENTS, ALGORITHM, ARCHITECTURE, DEVELOPMENT

The object of research is a comprehensive analysis, development and assessment of the technical, architectural and security aspects of the project to ensure its effective functioning.

The subject of research is the process of developing and implementing a Full-stack web application, with a focus on technical aspects, architectural solutions, security, testing, integration, deployment and optimization to ensure its stable and productive work.

The purpose of the work is to develop and implement a Full-stack web application, as well as an in-depth analysis of technical, architectural, security and performance aspects to create an effective and functional web application.

Research methods - literature analysis, code writing, testing and code analysis.

In this work, the created application is a music platform where users can create and manage information about bands and their tracks. Key features include adding new bands, specifying band information (name, founding date, location, etc.), and adding tracks to each band. The application also provides the ability to sort, paginate and delete tracks through appropriate tables and modal windows. Interaction with the server takes place using Apollo Client and GraphQL requests. The app uses Firebase to store images of bands. It is also possible to delete the tracks of the group through the corresponding modal window with confirmation.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SCSS – розширена синтаксисна оболонка для CSS, яка надає можливість використовувати змінні, вкладені правила, міксини та інші покращення для зручності написання стилів;

ORM – техніка програмування, що дозволяє взаємодіяти з базою даних за допомогою об'єктно-орієнтованого програмування;

Prisma – SQL-клієнт та ORM для Node.js і TypeScript, який допомагає взаємодіяти з базами даних, такими як PostgreSQL, MySQL, та SQLite;

PostgreSQL – вільна та джерельна реляційна система управління базами даних;

SQL – мова запитів, використовувана для взаємодії з реляційними базами даних;

HTTP – протокол передачі даних, що використовується для звернення до ресурсів в мережі Інтернет;

CSS – мова опису стилів для представлення зовнішнього вигляду веб-сторінок;

MUI – бібліотека React для створення інтерфейсів користувача, дотримуючись дизайну Material Design;

HTML – мова розмітки для створення структури веб-сторінок;

DOM – інтерфейс для представлення структури документа та взаємодії з ним через програми;

JSX – синтаксис розширення JavaScript, який дозволяє використовувати HTML-подібні вирази в коді React;

JS – мова програмування, що використовується для створення динамічних елементів веб-сторінок;

TS – розширення JavaScript, яке додає строгу типізацію та інші функції до мови;

TSX – розширення JavaScript, що використовується для написання коду React зі строгим типізованим підходом;

API – набір правил та інструментів для побудови програмного забезпечення;

GIT – система контролю версій для відстеження та керування змінами в програмному коді;

СКБД – система управління базами даних, набір взаємопов'язаних даних (база даних) і програм для доступу до цих даних;

JSON – текстовий формат обміну даними, що базується на JavaScript.

ЗМІСТ

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 13
1.1 Аналіз предметної галузі 13
1.2 Виявлення основних проблем або викликів, з якими стикаються представники галузі 15
1.3 Висновок до розділу 17
2. ВИБІР ТЕХНОЛОГІЙ ТА СТЕКУ 18
2.1 Бібліотека React для створення інтерфейсів 18
2.2 Бібліотека React Router для навігації по веб-додатку 23
2.3 Використання мови розмітки HTML у React 26
2.4 Опис мови програмування JavaScript 28
2.5 Використання мови JavaScript в біліотеці React 31
2.6 Стилі CSS та препроцесор SCSS 32
2.7 Препроцесор SCSS та його використання у бібліотеці React 35
2.8 Різниця між стилями SCSS та звичайним CSS 37
2.9 Опис мови програмування TypeScript 39
2.10 Використання мови програмування TypeScript в бібліотеці React. . 40
2.11 Використання бібліотеки MUI разом з бібліотекою React 44
2.12 Інструменти бібліотеки Material-UI 45
2.13 Використання бібліотек GraphQL та Apollo 48
2.14 Огляд бібліотеки Apollo Server. 51
2.15 Фреймворк Express.js для серверної частини 53
2.16 СКБД PostgreSQL та Prisma ORM для управління базою даних 55

2.17 Бібліотеки Formik та Yup..... 57
2.18 Висновки до розділу..... 59
3. АРХІТЕКТУРА ЗАСТОСУНКУ..... 60
3.1 Структурна організація проекту..... 61
3.2 Вміст папки client..... 64
3.3 Вміст папки server..... 69
3.4 Висновки до розділу..... 71
4. РОЗРОБКА ФУНКЦІОНАЛУ..... 72
4.1 Встановлення програми Visual Studio Code..... 72
4.2 Встановлення програми Node.js..... 74
4.3 Встановлення програми PostgreSQL..... 75
4.4 Встановлення програми PGAdmin..... 76
4.5 Встановлення програми Git..... 77
4.6 Первинна ініціалізація проекту та створення репозиторію..... 79
4.7 Створення монорепозиторію..... 79
4.8 Встановлення базових пакетів до кліенту..... 80
4.9 Встановлення базових пакетів до сервера..... 83
4.10 Встановлення та конфігурація ESlint та Prettier для форматування коду. 84
4.11 Конфігурація мови програмування TypeScript..... 90
4.12 Створення репозиторію на GitHub та перший пуш..... 92
4.13 Написання коду для запуску серверу..... 95
4.14 Написання коду для ORM Prisma..... 100
4.15 Написання коду для GraphQL схеми..... 104
4.16 Написання коду для GraphQL кверів..... 107

4.17 Написання коду для GraphQL мутацій 110
4.18 Запуск серверу 114
4.19 Конфігурація бібліотеки Apollo Client 115
4.20 Написання сторінки для регістрації 118
4.21 Написання сторінки для логіну. 120
4.22 Написання сторінки для основної таблиці сайту 122
4.23 Написання сторінки для профілю користувача 135
4.24 Висновки до розділу 139
ВИСНОВКИ 140
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ ..... **Ошибка! Закладка не определена.**
ДОДАТКИ 150
ДОДАТОК А 151
ДОДАТОК В 159

ВСТУП

У епоху сучасних технологій веб-розробка виходить за межі простих інструментів, надаючи можливість створювати інноваційні та функціональні Full-stack веб-застосунки. Історія веб-розробки свідчить про постійний розвиток, але в сучасному світі створення Full-stack веб-застосунків вимагає комплексного підходу та використання різноманітних технологій. У процесі розробки даного веб-застосунку використовуються сучасні засоби та бібліотеки, такі як React для реалізації клієнтської частини, TypeScript для забезпечення статичного типізації та покращеної розробки, SCSS для стилізації, Material UI для естетичного та функціонального дизайну і MUI Chart для візуалізації даних. Клієнтська частина проекту розробляється за допомогою інструменту Vite, що дозволяє отримати швидку та ефективну компіляцію веб-застосунку. Дипломний проект є результатом прагнення вивчення та вдосконалення найновіших технологій для ефективної розробки та оптимізації веб-додатків.

На серверній стороні використовуються технології, такі як TypeScript для реалізації серверного коду, Express для створення веб-сервера, GraphQL для взаємодії з клієнтом та Apollo для зручного управління запитами та даними. Робота з базою даних забезпечується за допомогою PostgreSQL та Prisma для зручного доступу та маніпуляції даними. У контексті валідації та обробки форм використовуються бібліотеки yup та formik.

Цей проект не лише демонструє високий рівень технічних навичок, але й відзначається використанням сучасних та ефективних інструментів для створення високоякісного Full-stack веб-застосунку. У рамках даного дипломного проекту розглядатимуться ключові аспекти розробки, архітектурні вирішення, безпека, тестування та інші складові, що сприяють успішній реалізації та функціонуванню веб-додатку.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної галузі

Веб-розробка - це область, яка постійно змінюється та вдосконалюється. Щороку з'являються нові технології, фреймворки, бібліотеки та інструменти, які роблять веб-сайти та веб-додатки більш функціональними, красивими та зручними для користувачів. Full-stack розробка - це здатність працювати з різними частинами веб-проекту, включаючи фронтенд (інтерфейс), бекенд (логіка) та бази даних.

Серед сучасних тенденцій у веб-розробці та Full-stack технологіях можна виділити такі:

- Jamstack - це архітектура, яка базується на використанні JavaScript, API та попередньо згенерованого HTML. Jamstack дозволяє створювати швидкі, безпечні та масштабовані веб-сайти, які не залежать від традиційних серверів. Jamstack використовує статичні генератори сайтів, такі як Next.js, Gatsby, Nuxt.js та інші, а також різні сервіси для хостингу, автентифікації, CMS і т.д;
- Serverless - це парадигма, яка передбачає, що розробник не дбає про серверну інфраструктуру, а лише пише код, який виконується на хмарних платформах, таких як AWS Lambda, Google Cloud Functions, Azure Functions та інші. Serverless дозволяє знизити витрати, спростити розгортання і масштабування, а також підвищити надійність і продуктивність веб-додатків;
- GraphQL - це мова запитів та схеми, яка дозволяє ефективно працювати з даними на фронтенді та бекенді. GraphQL дозволяє вимагати лише ті дані, які потрібні, а також отримувати їх у зручному форматі. GraphQL також підтримує реальний час, підписки, кешування та інші можливос-

ті. GraphQL використовується з різними фреймворками і бібліотеками, такими як React, Angular, Vue, Apollo, Relay та інші;

- React - це одна з найпопулярніших бібліотек для створення інтерфейсів на JavaScript. React дозволяє створювати компоненти, які можуть перевикористовуватися, мають свій стан та реагують на зміни даних. React також підтримує різні розширення та інструменти, такі як React Hooks, React Router, Redux, Next.js, Gatsby та інші;
- Vue – це ще одна популярна бібліотека для створення інтерфейсів JavaScript. Vue відрізняється від React тим, що використовує декларативний шаблон синтаксис, а також надає більше вбудованих можливостей, таких як директиви, фільтри, слоти і т.д. Vue також підтримує різні розширення та інструменти, такі як Vue Router, Vuex, Nuxt.js, Quasar та інші;
- Angular – це фреймворк для створення односторінкових програм на TypeScript. Angular відрізняється від React і Vue тим, що використовує модель MVC (Model-View-Controller), а також надає більше функціональності коробки, такої як залежності, сервіси, модулі, директиви, пайпи і т.д. Angular також підтримує різні розширення та інструменти, такі як Angular Material, Angular CLI, RxJS та інші;
- Svelte - це компілятор, який дозволяє створювати інтерфейси JavaScript. Svelte відрізняється від React, Vue та Angular тим, що не використовує віртуальний DOM, а генерує ефективний код, який безпосередньо оновлює DOM. Svelte також підтримує різноманітні можливості, такі як реактивність, анімація, транзишні, слоти і т.д;
- Tailwind CSS – це утилітарний фреймворк для стилізації веб-сайтів за допомогою CSS. Tailwind CSS відрізняється від традиційних фреймворків, таких як Bootstrap, Foundation та інші, тим, що не надає готових компонентів, а лише набір класів, які можна комбінувати та налаштовувати. Tailwind CSS дозволяє створювати адаптивні, кастомізовані та консистентні дизайни;

- WebAssembly – це бінарний формат, який дозволяє виконувати код, написаний різними мовами програмування, у браузері. WebAssembly дозволяє прискорити роботу веб-застосунків, а також використовувати такі можливості, як 3D-графіка, аудіо, відео, машинне навчання та інші. WebAssembly підтримується більшістю сучасних браузерів і може працювати разом із JavaScript;
- Progressive Web Apps (PWA) - це веб-програми, які поєднують у собі переваги веб-сайтів та нативних програм. PWA дозволяють працювати офлайн та відправляти пуш нотифікації, використовувати іконки на головному екрані та інші можливості. PWA також покращує продуктивність, надійність та доступність веб-додатків.

1.2 Виявлення основних проблем або викликів, з якими стикаються представники галузі

Full-stack веб-розробка - це здатність працювати з різними частинами веб-проекту, включаючи фронтенд (інтерфейс), бекенд (логіка) та бази даних. Це вимагає від розробників широкого спектру знань та навичок, а також постійного навчання та адаптації до нових технологій та трендів.

Серед основних проблем або викликів, з якими стикаються представники галузі Full-stack веб розробки. Існує безліч мов програмування, фреймворків, бібліотек та інструментів, які можна використовувати для створення веб-проектів. Не завжди легко визначити, які з них краще підходять для конкретного завдання, які з них сумісні один з одним, які мають хорошу документацію і підтримку, які з них будуть затребувані в майбутньому. Розробники повинні постійно вивчати нові технології та інструменти, а також враховувати їх переваги та недоліки при виборі стеку для своїх проектів.

Веб-проекти часто працюють з різними даними, у тому числі з особистими та конфіденційними даними користувачів, партнерів, клієнтів тощо. Розробники повинні забезпечити захист цих даних від несанкціонованого

доступу, витоку, крадіжки, підробки та інших загроз. Для цього необхідно використовувати різні методи та технології, такі як шифрування, автентифікація, авторизація, валідація, санітизація, тестування тощо. Розробники також повинні дотримуватись різних законів і норм, що регулюють обробку та зберігання даних, такі як GDPR, CCPA та інші.

Веб-проекти повинні бути швидкими, чуйними та надійними, щоб забезпечити хороший користувальницький досвід та конкурентоспроможність. Розробники повинні враховувати різні фактори, що впливають на продуктивність веб-проектів, такі як розмір та швидкість завантаження ресурсів, кількість та складність запитів до сервера та бази даних, архітектура та дизайн коду, кешування, стиснення, мініфікація тощо. Розробники також повинні використовувати різні інструменти та методи для вимірювання, аналізу та оптимізації продуктивності веб-проектів, таких як Lighthouse, PageSpeed Insights, DevTools та інші.

Адаптивність та доступність. Веб-проекти повинні бути адаптовані до різних пристроїв, браузерів, дозволів екрану, орієнтацій тощо, щоб забезпечити зручність та універсальність використання. Розробники повинні використовувати різні технології та підходи для створення адаптивних веб-проектів, таких як медіа-запити, гнучкі сітки, відносні одиниці виміру, векторна графіка тощо. Розробники також повинні враховувати потреби та особливості різних груп користувачів, у тому числі людей з обмеженими можливостями, та використовувати різні технології та підходи для створення доступних веб-проектів, такі як семантична розмітка, контрастність кольорів, альтернативний текст, клавіатурна навігація, скринрідери тощо. буд.

Веб-проекти повинні бути якісними, стабільними та безпомилковими, щоб забезпечити функціональність та надійність. Розробники повинні використовувати різні методи та інструменти для тестування та налагодження веб-проектів, такі як юніт-тести, інтеграційні тести, функціональні тести, регресійні тести, E2E-тести, TDD, BDD та інші. Розробники також повинні враховувати різні сценарії та випадки використання веб-проектів, такі як

крос-браузерність, крос-платформність, крос-пристрій, реальний час, оф-флайн-режим і т.д.

1.3 Висновок до розділу

У цьмоу розділу було розглянуто предметну галузь, обговорено сучасні технології для веб розробки та описані плюси з мінусами. Приділено увагу проблемам з якими може стикнутись розробник, їх причина та варіанти вирішення.

2. ВИБІР ТЕХНОЛОГІЙ ТА СТЕКУ

Обгрунтування вибору технологій для розробки веб-застосунку включає врахування кількох ключових критеріїв, таких як продуктивність, зручність в розробці, швидкість розгортання, спільнота та підтримка, а також сумісність зі специфікаціями проекту. Нижче приведено детальний опис технологій та порівняно їх з декількома аналогами.

2.1 Бібліотека React для створення інтерфейсів

React - це одна з найпопулярніших і затребуваних бібліотек для створення інтерфейсів на JavaScript. React дозволяє створювати компоненти, які можуть перевикористовуватися, мають свій стан і реагують на зміни даних. React також підтримує різні розширення та інструменти, такі як React Hooks, React Router, Redux, Next.js, Gatsby та інші.

Вибір React обгрунтовується його високою продуктивністю, великою спільнотою розробників та гнучкістю компонентної архітектури. В порівнянні з Angular та Vue.js, Angular відзначається повнішим фреймворком, але має вищий поріг входження. Vue.js — простий, але менше використовується в великих корпоративних проектах.

Компоненти React дозволяют будувати веб-інтерфейси шляхом розбиття їх на невеликі, повторно використовувані компоненти. Компоненти можуть бути класовими або функціональними.

React використовує віртуальний DOM для оптимізації оновлення інтерфейсу. Замість безпосередньої маніпуляції реальним DOM, React працює з віртуальним DOM, зменшуючи кількість операцій на реальному DOM.

JSX це розширення синтаксису JavaScript, яке дозволяє писати HTML-подібний код безпосередньо в JavaScript. JSX полегшує роботу з React-елементами та компонентами.

Стан (State), це компоненти можуть мати власний стан, який може змінюватися в результаті взаємодії користувача або зовнішніх подій. Зміна стану спричиняє перерендеринг компоненту.

Компоненти можуть приймати дані через пропси. Пропси передаються в компоненти як атрибути, і це дозволяє їм бути конфігурованими та повторно використовуваними з різними наборами даних.

React компоненти мають життєвий цикл, який включає різні етапи від створення до знищення компонента. Це дозволяє вам виконувати операції в різних моментах життєвого циклу, таких як підписка на дані, оновлення стану тощо.

React використовує одностороннє зв'язування даних, що означає, що дані потрапляють вниз по ієрархії компонентів через пропси, і зміни в стані можуть впливати лише на компонент, що їх має.

Приклад React компоненти HelloComponent, це функціональний компонент, що приймає props (властивості). Компонента відображає привітання з іменем, яке передається через пропс name. Використовує JSX для опису структури HTML-подібного коду внутрішнього вмісту компоненти, як показано на (рис 2.1).

```jsx
// Простий функціональний компонент
import React from 'react';

const HelloComponent = (props) => {
  return (
    <div>
      <h1>Hello, {props.name}!</h1>
    </div>
  );
};

export default HelloComponent;
```

Рисунок 2.1 - Простий функціональний компонент

Використання цієї компоненти в іншому файлі у компоненті App:

Компонента App є батьківською для HelloComponent. Використовує HelloComponent та передає йому ім'я "John" через пропс name. Це єдиний кореневий компонент в даному прикладі, що наведено на (рис 2.2).

```jsx
import React from 'react';
import HelloComponent from './HelloComponent';

const App = () => {
  return (
    <div>
      <HelloComponent name="John" />
    </div>
  );
};

export default App;
```

Рисунок 2.2 - Компонента App

Приклад класової компоненти в React та пояснення коду:

- імпорт React та Component import React, { Component } from 'react'; імпортує бібліотеку React та клас Component з неї;
- class Counter extends Component визначає клас компоненти, який розширює базовий клас React Component;

- constructor(props) викликається при створенні об'єкта класу. В даному випадку, він викликає конструктор базового класу (Component) і встановлює початковий стан об'єкта, де count: 0;
- increment = () => - це метод класу, визначений як стрілкова функція. Він викликається при натисканні кнопки;
- this.setState({ count: this.state.count + 1 }); - оновлює стан компоненти, збільшуючи значення count на 1;
- render() - метод, який повертає вузол React, який буде відображений;
- Count: {this.state.count} - відображає поточне значення count;
- <button onClick={this.increment}>Increment</button> - кнопка, при натисканні якої викликається метод increment;
- export default Counter експортує класову компоненту для використання в інших файлах.

У цьому прикладі, що показано на (рис 2.3) класова компонента Counter є простим лічильником, який можна збільшувати кнопкою. Метод increment змінює стан, і при кожному оновленні стану викликається метод render, який відображає поточне значення лічильника.

```
import React, { Component } from 'react';

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

Рисунок 2.3 - Класова компонента Counter з функціоналом лічильника

Переваги класових компонент:

- стан (State) та життєвий цикл, це ласові компоненти які можуть мати внутрішній стан і використовувати різні методи життєвого циклу, такі як componentDidMount, componentDidUpdate, і т. д. Це дає більший контроль над поведінкою компоненту;
- методи, це класові компоненти можуть мати власні методи, що дозволяє організовувати логіку компоненту більш прозоро та гнучко;
- Ref-и, класові компоненти дозволяють використовувати ref-и для отримання доступу до елементів DOM та інших класових компонентів.

Недоліки класових компонент:

- синтаксичний бар'єр, це код класових компонент зазвичай більший та складніший на вигляд, що може робити його менш зрозумілим для новачків;
- більше коду для простих випадків, т.е. компонентів, які використовуються тільки для відображення інформації, класові компоненти можуть

виглядати зайвими. Функціональні компоненти з використанням хуків можуть бути більш компактними;

- збереження контексту (this), тобто у класових компонентах необхідно вручну обробляти прив'язку this в методах або використовувати стрілкові функції.
- деякі нові функціональності та бібліотеки, наприклад, хуки, можуть мається підтримки лише для функціональних компонентів.

Ці концепції роблять React потужним інструментом для розробки веб-інтерфейсів, забезпечуючи простоту, ефективність та повторне використання коду.

2.2 Бібліотека React Router для навігації по веб-додатку

У проекті використовується зазначенний раніше бібліотека React Router.

React Router - це бібліотека для реактивного роутінгу в додатках, побудованих на React. Роутінг - це процес переходу користувача між різними сторінками (або "роутами") в веб-додатку. React Router надає інструменти для визначення, які компоненти React відображаються при різних URL-адресах.

Основні концепції React Router включають BrowserRouter та HashRouter, це два основних компонента для обгортання вашого додатка. BrowserRouter використовує HTML5 API для роутінгу і виглядає природньо в URL-адресах, тоді як HashRouter використовує hash-частину URL і може бути використаний там, де важливо уникнути проблем з серверним роутінгом. Приклад коду наведено нижче на (рис 2.4).

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-
dom';

const App = () => (
  <Router>
    <Switch>
      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </Switch>
  </Router>
);
```

Рисунок 2.4 – Приклад використнання React Router

Route визначає відповідність між URL-шляхом і компонентом React, який має бути відображений при відповідному шляху. Link та NavLink використовуються для створення навігаційних посилань. Link генерує звичайне посилання, тоді як NavLink може надавати додаткові стилі для відзначення поточної сторінки. Приклад використання наведено нижче на (рис 2.5)

```
import { Link, NavLink } from 'react-router-dom';

const Navigation = () => (
  <nav>
    <ul>
      <li><NavLink to="/" activeClassName="active">Home</NavLink></li>
      <li><NavLink to="/about"
activeClassName="active">About</NavLink></li>
      <li><NavLink to="/contact"
activeClassName="active">Contact</NavLink></li>
    </ul>
  </nav>
);
```

Рисунок 2.5 – Приклад використнання Link та NavLink

Switch, це обгортка для Route, яка рендерить тільки перший збігаючий шлях. Це дозволяє вам визначати роути в порядку їх додавання, а не у порядку їх визначення в Switch.

React Router дозволяє створювати динамічні та зручні для користувача веб-додатки, забезпечуючи ефективний спосіб управління роутінгом в React-

проектах. Це робить навігацію в додатках більш інтуїтивно зрозумілою та легко підтримуваною. На прикладі вищче представлено 5 версію бібліотеки. У проекті використовується нова 6 версія. Нижче на (рис 2.6) наведено приклад використання React Router версії 6 та описано нові концепції використання:

```
import { BrowserRouter as Router, Routes, Route, Link } from 'react-
router-dom';

const Home = () => <h1>Home</h1>;
const About = () => <h1>About</h1>;
const Contact = () => <h1>Contact</h1>;

const App = () => (
  <Router>
    <nav>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/contact">Contact</Link></li>
      </ul>
    </nav>

    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
    </Routes>
  </Router>
);

export default App;
```

Рисунок 2.6 – Приклад використнання React Router 6 версії

Основні відмінності від попередніх версій:

- Routes, тепер використовується компонент Routes для визначення набору роутів, і в кожному роуті використовується element замість component;
- Link, інтеграція посилань (Link) не вимагає використання to prop як стрічки; замість цього використовується to як об'єкт;
- Switch, більше не потрібен, оскільки Routes автоматично вибирає перший збігаючий роут. В версії 6 можна використовувати нові концепції, такі як useRoutes для власної логіки роутінгу поза компонентами, і більш динамічне визначення роутів.

2.3 Використання мови розмітки HTML у React

HTML (Hypertext Markup Language) - є стандартною мовою розмітки для створення веб-сторінок та веб-додатків. Він використовується для структурування та відображення вмісту веб-сайтів за допомогою тегів та їхніх атрибутів. Основні елементи HTML включають заголовки, параграфи, списки, таблиці, зображення та інші.

DOM (Document Object Model) дерево є структурою, що представляє структуру HTML-документу або XML-документу в браузері. Кожен елемент (такий як тег , <p>, тощо) та текстовий вузол представлені як об'єкти, що утворюють ієрархію вузлів. DOM дерево надає мовам програмування, таким як JavaScript, можливість динамічно змінювати структуру, стилі та вміст веб-сторінок.

Процес побудови DOM дерева:

- парсинг, коли браузер аналізує HTML-або XML-код і створює DOM в пам'яті;
- побудова дерева, колі кожен тег стає вузлом в дереві, атрибути - властивостями вузлів, текст - текстовим вмістом вузлів (рис. 2.7).

У цьому прикладі, DOM дерево буде містити вузли для <html>, <head>, <title>, <body>, , <p>, , і текстового вузла.

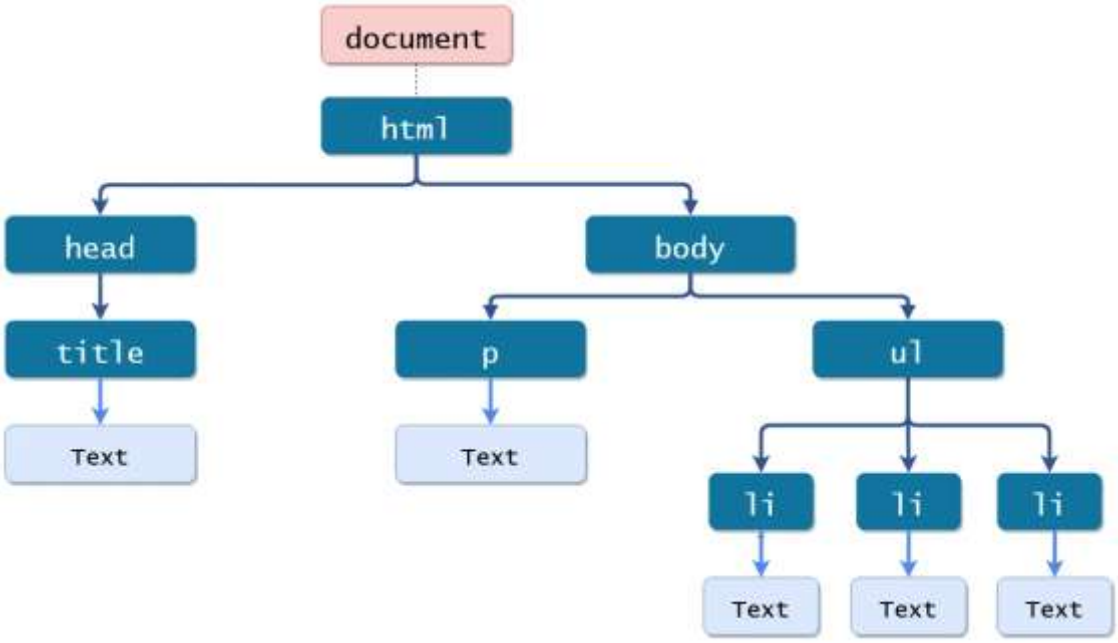


Рисунок 2.7 – Структурна схема DOM дерева

У React, HTML подаваний в компонентах за допомогою JSX - це розширення синтаксису JavaScript, яке нагадує HTML. JSX дозволяє описувати структуру інтерфейсу за допомогою тегів, що зроблює код більш зрозумілим та декларативним.

Приклад використання HTML в React через JSX наведено на (рис 2.8).

```
import React from 'react';

const MyComponent = () => {
  return (
    <div>
      <h1>Hello, React!</h1>
      <p>This is a simple React component.</p>
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
      </ul>
    </div>
  );
};

export default MyComponent;
```

Рисунок 2.8 – Приклад використання HTML в React через JSX

У цьому прикладі, , <h1>, <p>, <ul>, та <li> - це теги HTML, які використовуються в JSX для створення віртуального DOM в React.

Важливо відзначити, що атрибути та події в JSX вказуються так само, як і в HTML. Приклад наведено на (рис. 2.9).

```
const ButtonComponent = () => {
  return (
    <button onClick={() => alert('Button clicked!')}>
      Click me
    </button>
  );
};
```

Рисунок 2.9 – Завдання атрибуту події onClick

У цьому прикладі, onClick - це подія, що реагує на клік по кнопці, а alert('Button clicked!') - це JavaScript-код, який виконується при натисканні кнопки.

Таким чином, використання HTML в React відбувається за допомогою JSX, що дозволяє легко описувати структуру інтерфейсу та взаємодіяти з DOM елементами.

2.4 Опис мови програмування JavaScript

JavaScript - це високорівнева, об'єктно-орієнтована мова програмування, яка використовується для веб-розробки та інших сфер програмування. Основні ідеї JavaScript включають:

- динамічність мови, JavaScript є мовою з динамічною типізацією, що дозволяє змінювати типи змінних під час виконання програми;
- об'єктно-орієнтованість, JavaScript базується на об'єктно-орієнтованому підході, де програма організована як набір об'єктів, що взаємодіють один з одним.

Мова легко інтеграется з HTML/CSS, JavaScript використовується для динамічного змінення структури HTML та стилів CSS на стороні клієнта.

Функціональність програмування, JavaScript підтримує функціональні елементи, такі як замикання (closures) та вищі функції, що дозволяють писати більш зручний та ефективний код.

Важною частиною JS є механізм Event Loop. Механізм Event Loop в JavaScript використовується для забезпечення асинхронного програмування. Хоча JavaScript є однопотоковим, можна створити ілюзію багатопотоковості (паралельна модель) за допомогою структур даних.

Код JavaScript працює у однопотоковому режимі, що означає, що в один момент часу відбувається лише одна подія. Це спрощує процес програмування, оскільки відсутні проблеми паралелізму. Проте більшість браузерів мають свій власний цикл подій для кожної вкладки.

Спільним для всіх середовищ є вбудований механізм, відомий як Event Loop. Цей механізм контролює виконання різних фрагментів програми, викликаючи двигун JavaScript.

Принцип роботи можна подати наступним чином:

- javaScript очікує та чекає своє завдання;
- коли завдання з'являється, двигун починає їх виконання, починаючи з першого отриманого;

Якщо приходить нове завдання, але двигун зайнятий виконанням попереднього, воно додається в чергу. Візуально процес можна представити так, як на (рис. 2.10).

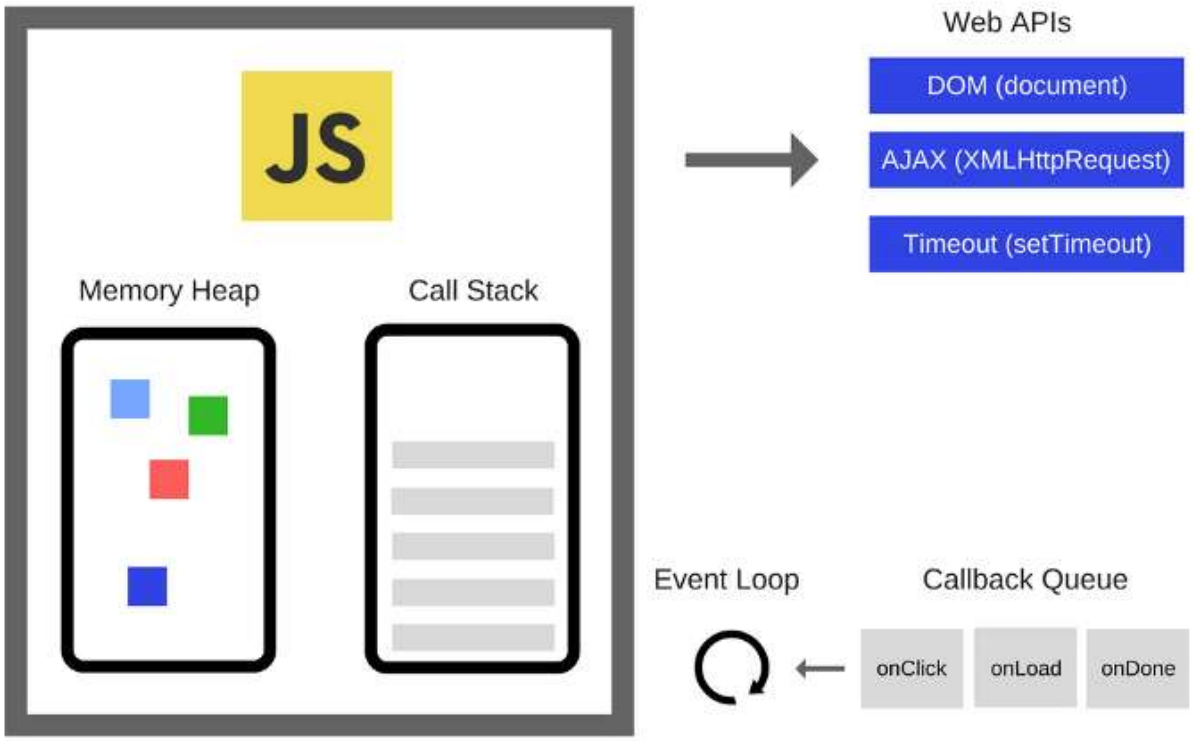


Рисунок 2.10 – Механізм Event Loop в JavaScript

Stack (Стек) представляє потік виконання JavaScript. Цикл подій виконує одну просту задачу - контролює стек викликів та чергу зворотних викликів. Якщо стек викликів порожній, цикл подій візьме першу подію з черги та відправить її в стек викликів для виконання. При виклику нового методу вгорі стеку виділяється окремий блок пам'яті. Стек викликів відповідає за відстеження всіх операцій в черзі, які мають бути виконані. Після завершення черги вона видаляється зі стеку.

Heap (Купа) використовується для створення нового об'єкта.

Queue (Черга) подій відповідає за відправку нових функцій на обробку. Це структура даних черги, яка підтримує правильну послідовність, в якій всі операції повинні відправлятися на виконання.

Web API не є частиною JavaScript, вони, скоріше, створені на основі JS. Кожен раз, коли викликається асинхронна функція, вона відправляється в

API браузера. За командою, отриманою зі стеку викликів, API запускає власну однопотокову операцію.

Цикл подій в JavaScript відрізняється від інших мов тим, що його потік виконання ніколи не блокується, за винятком деяких випадків, таких як alert або синхронний HTTP-запит, які не рекомендується використовувати. Тому навіть коли програма очікує запити зі сховища або відповідь від сервера, вона може обробляти інші процеси, такі як введення користувача.

2.5 Використання мови JavaScript в біліотеці React

JavaScript використовується в React для управління динамічними елементами інтерфейсу, взаємодії з користувачем та маніпуляцій з даними.

В React, використовується JavaScript для створення React-елементів, які представляють віртуальний DOM. JSX (розширений синтаксис JavaScript) дозволяє описувати їх з використанням синтаксису, що нагадує HTML: const element = <h1>Hello, React!</h1>.

Стан (State) та Життєвий цикл: JavaScript використовується для управління станом компонентів та реагування на життєвий цикл компоненту, що наведено на (рис. 2.11).

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return <p>Count: {this.state.count}</p>;
  }
}
```

Рисунок 2.11 – Демонстрація використання стану(State)

Обробники подій у JavaScript використовується для визначення та обробки подій, таких як кліки, зміни, тощо. Приклад наведено на (рис. 2.12).

```
handleClick = () => {
  alert('Button clicked!');
};

render() {
  return <button onClick={this.handleClick}>Click me</button>;
}
```

Рисунок 2.12 – Обробка події натискання на кнопку

Використання JavaScript-функцій дозволяє визначати та викликати функції, які використовуються для реалізації бізнес-логіки та зручності коду. Приклад наведено на (рис. 2.13).

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Oleh',
  lastName: 'Nikulin'
};

const greeting = <p>Hello, {formatName(user)}!</p>;
```

Рисунок 2.13 – Обробка події натискання на кнопку

JavaScript взаємодіє з React для створення динамічних та інтерактивних інтерфейсів, які можуть ефективно відображати та оновлювати дані для користувача.

2.6 Стилі CSS та препроцесор SCSS

CSS (Cascading Style Sheets) є мовою стилізації, яка використовується для оформлення веб-сторінок та визначення їх зовнішнього вигляду. Зазвичай використовується разом з HTML для розділення структури веб-документа від його стилів.

Селектори (Selectors) визначають, які елементи на сторінці будуть стилізовані. Приклад наведено на (рис. 2.14).

```
/* Приклад селектору для стилізації всіх заголовків h1 */
h1 {
  color: blue;
}
```

Рисунок 2.14 – Приклад селектору для стилізації всіх заголовків h1

Властивості (Properties) в изначають конкретні аспекти стилю, такі як колір тексту, розмір шрифту, тощо. Приклад наведено на (рис. 2.15).

```
/* Приклад властивостей для тексту заголовка h1 */
h1 {
  color: blue;
  font-size: 24px;
}
```

Рисунок 2.15 – Приклад властивостей для текста заголовка h1

Значення (Values), онкретні значення, які присвоюються властивостям. Приклад наведено на (рис. 2.16).

```
/* Приклад значень для властивостей кольору та розміру тексту */
h1 {
  color: #3366cc;
  font-size: 1.5em;
}
```

Рисунок 2.16 – Приклад значень для властивостей кольору та розміру тексту

Блочна модель (Box Model),визначає, як елементи в HTML розташовані та взаємодіють один з одним. Приклад наведено на (рис. 2.17).

```
/* Приклад визначення блочної моделі для елементу з класом "box" */
.box {
  width: 200px;
  height: 100px;
  border: 1px solid #000;
  padding: 10px;
  margin: 20px;
}
```

Рисунок 2.17 – Приклад визначення блочної моделі для елементу з класом "box"

Позиціонування (Positioning),визначає, як елементи позиціонуються на сторінці. Приклад наведено на (рис. 2.18).

```css
/* Приклад позиціонування елементу з ідентифікатором "header" */
#header {
  position: relative;
  top: 20px;
  left: 30px;
}
```

Рисунок 2.18 – Приклад позиціонування елементу з ідентифікатором "header"

Flexbox і Grid, модулі CSS, що дозволяють легко розміщувати елементи в просторі. Приклад наведено на (рис. 2.19).

```css
/* Приклад використання Flexbox для розміщення елементів в ряд */
.container {
  display: flex;
  justify-content: space-between;
}
```

Рисунок 2.19 – Приклад позиціонування елементу з ідентифікатором "header"

Приклад використання CSS в HTML наведено на (рис. 2.20).

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="styles.css">
  <title>Phonoteka</title>
</head>
<body>
  <header>
    <h1>Welcome to Phonoteka </h1>
  </header>
  <section class="content">
    <p>This is some content on my page.</p>
    <div class="box">A box with some text.</div>
  </section>
  <footer>
    <p>© 2024 Phonoteka </p>
  </footer>
</body>
</html>
```

Рисунок 2.20 – Приклад HTML з використанням CSS

CSS-код (у файлі styles.css) наведено на (рис 2.21).

```css
body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 0;
  background-color: #f0f0f0;
}

header {
  background-color: #333;
  color: #fff;
  padding: 10px;
  text-align: center;
}

h1 {
  margin: 0;
}

.content {
  padding: 20px;
}

.box {
  width: 200px;
  height: 100px;
  background-color: #e0e0e0;
  border: 1px solid #ccc;
  padding: 10px;
  margin-top: 20px;
}

footer {
  background-color: #333;
  color: #fff;
  padding: 10px;
  text-align: center;
}
```

Рисунок 2.21 – Приклад використання CSS в HTML

Цей приклад використовує CSS для стилізації заголовку, тексту, блоків, інтерфейсу користувача, а також розташування елементів на сторінці. CSS додає стиль та красу до HTML-структури, забезпечуючи приємний зовнішній вигляд веб-сторінки.

2.7 Препроцесор SCSS та його використання у бібліотеці React

SCSS (Syntactically Awesome Style Sheets) - це розширення мови Sass, яке додає синтаксичний цукор та додаткові функції до звичайного CSS. Син-

таксис SCSS ближчий до звичайного CSS, але дозволяє використовувати зручні функції та конструкції для полегшення написання та управління стилями.

Інсталяція SCSS у React виконується встановленням пакету node-sass або sass за допомогою npm або yarn та командою npm install node-sass або yarn add node-sass.

Написання стилів у файлу SCSS продемонстровано на (рис. 2.22).

```scss
// styles.scss

$primary-color: #3498db;

body {
  background-color: #f0f0f0;
}

header {
  background-color: $primary-color;
  color: #fff;
}
```

Рисунок 2.22 – Написання стилів у файлу SCSS

Імпорт стилів SCSS у React-компонент приведено на (рис. 2.23).

```jsx
// MyComponent.jsx

import React from 'react';
import './styles.scss'; // Шлях до вашого SCSS-файлу

const MyComponent = () => {
  return (
    <div>
      <header>
        <h1>Hello, React with SCSS!</h1>
      </header>
      <p>This is a simple React component.</p>
    </div>
  );
};

export default MyComponent;
```

Рисунок 2.23 – Імпорт стилів SCSS у React-компонент

2.8 Різниця між стилями SCSS та звичайним CSS

Змінні (Variables) SCSS дозволяє використовувати змінні для зберігання значень та використання їх у різних місцях, що показано:

```
SCSS
$primary-color: #3498db; // декларація змінної
body {
  background-color: $primary-color; // використання змінної
}
/* CSS (згенерований з SCSS) */
body {
  background-color: #3498db;
}.
```

Міксини (Mixins) SCSS дозволяє створювати та використовувати міксини для повторного використання блоків стилів, як показано:

```
// SCSS
@mixin center-element {
    display: flex;
    justify-content: center;
    align-items: center;
  }
  .box {
    @include center-element;
    width: 200px;
    height: 100px;
```

```
}
/* CSS (згенерований з SCSS) */
.box {
  display: flex;
  justify-content: center;
  align-items: center;
  width: 200px;
  height: 100px;
}.
```

Вкладеність (Nesting) SCSS дозволяє вкладати правила стилів, що полегшує читання та управління кодом, як показано:

```
// SCSS
header {
  h1 {
    font-size: 24px;
  }
  p {
    font-size: 16px;
  }
}
/* CSS (згенерований з SCSS) */
header h1 {
  font-size: 24px;
}
header p {
  font-size: 16px;
}.
```

Імпорт інших файлів SCSS дозволяє імпортувати інші SCSS-файли в основний файл, як показано:

```
// SCSS
@import 'reset'; // Імпорт файлу зі стилями скидання (reset)
body {
  background-color: #f0f0f0;
}
/* CSS (згенерований з SCSS) */
/* Стилі імпортовані з reset.scss */
body {
  margin: 0;
  padding: 0;
}.
```

SCSS полегшує написання та управління стилями завдяки своїм додатковим функціям та зручному синтаксису. При цьому, він компілюється в звичайний CSS для використання у веб-проектах.

2.9 Опис мови програмування TypeScript

TypeScript - це мова програмування, що розширює JavaScript, додаючи до нього статичну типізацію. Вона розроблена компанією Microsoft та є відкритою (open source) мовою. Основною метою TypeScript є полегшення розробки великих та складних проектів, забезпечуючи можливість визначення типів для змінних, параметрів функцій та об'єктів.

Основні ідеї TypeScript складается в статичноі типізації - TypeScript дозволяє визначати типи для змінних, функцій та інших об'єктів, let message: string = "Hello, TypeScript!". Це дозволяє виявляти та виправляти помилки на етапі розробки, що полегшує підтримку коду та зменшує кількість помилок на етапі виконання.

Об'єктно-орієнтований TypeScript підтримує об'єктно-орієнтовану розробку, включаючи можливість використання класів, інтерфейсів, успадкування та інших ООП-концепцій, так як це показано:

```
// Приклад використання класу та інтерфейсу
interface Person {
```

```
  name: string;
  age: number;
}
class Student implements Person {
  constructor(public name: string, public age: number) {}
}.
```

TypeScript код потрібно компілювати в JavaScript перед тим, як він виконуватиметься в браузері або на сервері. TypeScript компілюється у чистий, стандартний JavaScript, що дозволяє використовувати його в усіх середовищах, де підтримується JavaScript.

TypeScript підтримує нові функції та стандарти ECMAScript, дозволяючи використовувати сучасні можливості JavaScript, як показано

```
// Приклад використання стрілкових функцій (Arrow Functions)
const add = (a: number, b: number): number => a + b.
```

TypeScript дозволяє використовувати модульний підхід до розробки, що полегшує організацію та управління кодом, це продемонсровано:

```
// Приклад експорту та імпорту модулів
// math.ts
export const add = (a: number, b: number): number => a + b;
// app.ts
import { add } from './math';
console.log(add(2, 3)); // Вивід: 5
```

2.10 Використання мови програмування TypeScript в бібліотеці React.

Використання TypeScript в React дозволяє покращити структуру та надійність вашого коду за допомогою статичної типізації та інших особливостей TypeScript. Ось кілька ключових аспектів використання TypeScript у React. Створення TypeScript-компонентів:

```
// Простий приклад TypeScript-компоненту
import React, { FC } from 'react';
```

```
interface Props {
  name: string;
  age: number;
}
const MyComponent: FC<Props> = ({ name, age }) => {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
    </div>
  );
};
export default MyComponent.
```

В цьому прикладі, що наведено, використовуємо інтерфейс Props для визначення типів властивостей, які можуть бути передані в компонент. Компонент потім приймає об'єкт props з визначеними типами.

Основні особливості TypeScript:

1. Статична типізація: TypeScript дозволяє визначати типи, що робить код більш надійним, полегшує відлагодження та робить проект більш масштабованим;

2. Підтримка ООП: TypeScript має багатий інструментарій для об'єктно-орієнтованої розробки, що робить його ідеальним для великих проектів та команд розробників;

3. Зменшення помилок: Статична типізація та інші особливості TypeScript дозволяють виявляти багато типових помилок на етапі розробки, що призводить до зменшення кількості помилок в робочому коді;

3. Розширені можливості IDE: Багато редакторів коду та інтегрованих середовищ розробки (IDE), таких як Visual Studio Code, мають вбудовану підтримку TypeScript, що дозволяє отримати розширений функціонал при роботі з кодом;

4. Підтримка сучасних стандартів: TypeScript дозволяє використовувати нові функції та стандарти ECMAScript раніше, ніж вони офіційно підтримуються браузерами;

У прикладі нижче, використовуємо useState для управління станом компонента та визначаємо тип стану як number:

```
import React, { FC, useState } from 'react';
interface CounterProps {
  initialValue: number;
}
const Counter: FC<CounterProps> = ({ initialValue }) => {
  const [count, setCount] = useState<number>(initialValue);
  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
export default Counter.
```

У класових компонентах також можна використовувати TypeScript, визначаючи типи для властивостей та стану, як це показано на прикладі:

```
import React, { Component } from 'react';
interface State {
  count: number;
}
class CounterClass extends Component<{}, State> {
```

```
  constructor(props: {}) {
    super(props);
    this.state = {
      count: 0,
    };
  }
  increment = () => {
    this.setState((prevState) => ({ count: prevState.count + 1 }));
  };
  decrement = () => {
    this.setState((prevState) => ({ count: prevState.count - 1 }));
  };
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
        <button onClick={this.decrement}>Decrement</button>
      </div>
    );
  }
}
export default CounterClass.
```

TypeScript також підтримує використання Generics для створення загальновикористовуваних компонентів та хуків, як це показано:

```
// Приклад Generics у компоненті
interface BoxProps<T> {
    content: T;
```

```
}
const Box: FC<BoxProps<string>> = ({ content }) => <div>{content}</div>;


// Приклад Generics у хуці
import { useState } from 'react';
function useLocalStorage<T>(key: string, initialValue: T) {
  const [storedValue, setStoredValue] = useState<T>(() => {
    const item = window.localStorage.getItem(key);
    return item ? JSON.parse(item) : initialValue;
  });
  const setValue = (value: T) => {
    setStoredValue(value);
    window.localStorage.setItem(key, JSON.stringify(value));
  };
  return [storedValue, setValue] as const;
}
// Використання хука
const [value, setValue] = useLocalStorage<number>('count', 0).
```

У цих прикладах Generics дозволяє параметризувати типи для компонентів та хуків, щоб робити їх більш універсальними.

Загальне використання TypeScript в React допомагає виявляти та усувати помилки на етапі розробки, полегшує роботу з великими кодовими базами та покращує загальну надійність та читабельність коду.

2.11 Використання бібліотеки MUI разом з бібліотекою React

Material-UI (MUI) - це бібліотека компонентів для React, яка реалізує дизайн інтерфейсу в стилі Material Design від Google. Вона надає готові компоненти для розробки сучасних та стильних веб-застосунків.

Основні плюси використання Material-UI в React-застосунках:

- Material-UI надає широкий набір готових компонентів, таких як кнопки, форми, таблиці, таби, карточки та інші. Це дозволяє розробникам швидко та легко створювати стильні інтерфейси без необхідності писати код з нуля;
- Material Design, це бібліотека дотримується принципів Material Design, які розроблені Google. Це означає, що ваша програма буде мати чистий та сучасний вигляд, з урахуванням принципів дизайну, таких як тіні, анімації та контрастність;
- Material-UI дозволяє налаштовувати стилі компонентів з використанням тим, що дозволяє легко внести зміни у вигляд вашого додатка. Адаптивність, компоненти Material-UI реагують на різні розміри екрану, допомагаючи створювати адаптивні та мобільно-дружні інтерфейси;
- Material-UI має велику та активну спільноту розробників, що дозволяє швидко отримати допомогу, вирішити проблеми та отримати підтримку;
- Інтеграція з React: Material-UI розроблено спеціально для інтеграції з React, використовуючи його природний підхід до компонентного програмування.

2.12 Інструменти бібліотеки Material-UI

ThemeProvider дозволяє легко налаштовувати теми для вашого додатка. Можливо змінювати кольори, шрифти, розміри та інші стилі з допомогою тем, як це продемонстровано на (рис. 2.24).

```
import { createTheme, ThemeProvider } from '@mui/material/styles';

const theme = createTheme({
  palette: {
    primary: {
      main: '#2196f3',
    },
    secondary: {
      main: '#f50057',
    },
  },
});

function MyApp() {
  return (
    <ThemeProvider theme={theme}>
      {/* Ваш компонент тут */}
    </ThemeProvider>
  );
}
```

Рисунок 2.24 – Приклад використання ThemeProvider

Material-UI включає гнучку систему сітки, яка допомагає розміщати компоненти на сторінці відповідно до потреб дизайну, це продемостровано на (рис. 2.25).

```
import Grid from '@mui/material/Grid';

function MyGrid() {
  return (
    <Grid container spacing={2}>
      <Grid item xs={12} md={6}>
        {/* Вміст для першого стовпця */}
      </Grid>
      <Grid item xs={12} md={6}>
        {/* Вміст для другого стовпця */}
      </Grid>
    </Grid>
  );
}
```

Рисунок 2.25 – Приклад використання Grid System

Бібліотека також включає набір іконок, які відповідають Material Design. Є можливість легко використовувати ці іконки у проекті, так як це показано:

```
import IconButton from '@mui/material/IconButton';
import DeleteIcon from '@mui/icons-material/Delete';
function MyIconButton() {
 return (
   <IconButton color="primary">
     <DeleteIcon />
   </IconButton>
 );
}.
```

Material-UI також надає компоненти для роботи з графіками, які включаються у розділ MUI Charts. Ці компоненти дозволяють створювати різноманітні графіки та діаграми ваших даних. Основні компоненти для роботи з графіками в Material-UI показано на (рис. 2.26) та (рис. 2.27):

```
import { LineChart, Line, XAxis, YAxis, CartesianGrid, Tooltip, Legend }
from 'recharts';

function MyLineChart() {
  const data = [
    { name: 'January', uv: 4000, pv: 2400, amt: 2400 },
    { name: 'February', uv: 3000, pv: 1398, amt: 2210 },
    // ...
  ];

  return (
    <LineChart width={500} height={300} data={data}>
      <XAxis dataKey="name" />
      <YAxis />
      <CartesianGrid stroke="#eee" strokeDasharray="5 5" />
      <Line type="monotone" dataKey="uv" stroke="#8884d8" />
      <Line type="monotone" dataKey="pv" stroke="#82ca9d" />
      <Tooltip />
      <Legend />
    </LineChart>
  );
}
```

Рисунок 2.26 – Приклад використання LineChart та Line

Ці компоненти базуються на бібліотеці Recharts і дозволяють легко інтегрувати графіки та діаграми в ваші Material-UI проекти. Вони надають різні налаштування та можливості для створення різноманітних типів графіків з використанням вашого набору даних.

```
import { BarChart, Bar, XAxis, YAxis, CartesianGrid, Tooltip, Legend }
from 'recharts';

function MyBarChart() {
  const data = [
    { name: 'January', uv: 4000, pv: 2400, amt: 2400 },
    { name: 'February', uv: 3000, pv: 1398, amt: 2210 },
    // ...
  ];

  return (
    <BarChart width={500} height={300} data={data}>
      <XAxis dataKey="name" />
      <YAxis />
      <CartesianGrid stroke="#eee" strokeDasharray="5 5" />
      <Bar dataKey="uv" fill="#8884d8" />
      <Bar dataKey="pv" fill="#82ca9d" />
      <Tooltip />
      <Legend />
    </BarChart>
  );
}
```

Рисунок 2.27 – Приклад використання BarChart та Bar

2.13 Використання бібліотек GraphQL та Apollo

GraphQL - це мова запитів для API, яка дозволяє клієнту запитувати лише ті дані, які йому потрібні, і отримувати їх у форматі, зручному для нього. Розроблений Facebook, GraphQL надає більш гнучкий та ефективний спосіб взаємодії з сервером порівняно з традиційним REST.

Основні особливості GraphQL:

1. Гнучкість запитів: Клієнт визначає структуру даних, яку він хоче отримати, і сервер повертає лише ті дані, які запитані. Це уніфікує та оптимізує передачу даних між клієнтом та сервером;
2. Єдиний точка входу: Всі запити виконуються через один єдиний ендпоінт, що спрощує управління та відслідковування запитів;
3. Типізація: GraphQL визначає схему з типами даних, яка описує, які дані можна запитати та які можна отримати;
4. Підтримка реального часу: GraphQL дозволяє використовувати підписки, що дозволяють серверу активно відправляти зміни клієнту при їх виникненні.

Apollo Client - це бібліотека для роботи з GraphQL в клієнтському JavaScript-коді. Вона надає інструменти для виконання запитів до GraphQL-

серверів, управління локальним станом та автоматичну кешування даних.

Розглянемо основні можливості Apollo Client.

Запити та мутації GraphQL: Apollo Client дозволяє легко виконувати запити та мутації до GraphQL-серверів, це продемонстровано на (рис. 2.39).

Query (запит), це GraphQL-запит, що визначає дані, які клієнт хоче отримати від сервера. У прикладі це query, який запитує ім'я, електронну пошту та інформацію про пости для користувача з ідентифікатором:

```
import { useQuery, gql } from '@apollo/client';

const GET_USERS = gql`
  query {
    users {
      id
      name
    }
  }
`;

function MyComponent() {
  const { loading, error, data } = useQuery(GET_USERS);
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;
  return (
    <ul>
      {data.users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>.
```

Apollo Client дозволяє легко управляти локальним станом за допомогою Apollo Cache.

Приклад нижче демонструє використання кешу в Apollo Client. Apollo Client використовує кеш для зберігання та управління даними, отриманими з GraphQL-запитів та мутацій. Кеш дозволяє ефективно оновлювати та відо-

бражати дані в інших частинах вашого додатку без додаткових мережевих запитів:

```
import { useQuery, gql, useMutation } from '@apollo/client';
const GET_TODO_LIST = gql`
 query {
  todos {
   id
   text
   completed
  }
 }
`;
const TOGGLE_TODO = gql`
 mutation ToggleTodo($id: ID!) {
  toggleTodo(id: $id) {
   id
   completed
  }
 }
`;

function TodoList() {
 const { loading, error, data } = useQuery(GET_TODO_LIST);

 const [toggleTodo] = useMutation(TOGGLE_TODO);

 const handleToggleTodo = (id) => {
  toggleTodo({
   variables: { id },
   refetchQueries: [{ query: GET_TODO_LIST }],
  });
 };
 if (loading) return <p>Loading...</p>;
 if (error) return <p>Error: {error.message}</p>;
 return (
  <ul>
   {data.todos.map((todo) => (
    <li key={todo.id} onClick={() => handleToggleTodo(todo.id)}>
     {todo.text} - {todo.completed ? 'Completed' : 'Incomplete'}
    </li>
   ))}
```

```
        </ul>
      );
}.
```

Ця опція – refetchQueries, вказує Apollo Client, які запити повинні бути перезавантажені після успішного виклику мутації. У цьому випадку, ми передаємо об'єкт, що містить query: GET_TODO_LIST. Це означає, що ми хочемо перезавантажити дані за допомогою GraphQL-запиту GET_TODO_LIST.

Ця стратегія використовується для оновлення кешу та відображення найновіших даних у нашому компоненті, але це не єдиний спосіб використання кешу. Apollo Client також автоматично оновлює кеш для даного типу даних після виклику мутації за певними умовами (наприклад, якщо id мутованого об'єкта є в кеші).

Використання кешу в Apollo Client дозволяє додаткам більш ефективно та ефективно взаємодіяти з даними, зменшуючи кількість мережевих запитів та поліпшуючи продуктивність.

2.14 Огляд бібліотеки Apollo Server.

Apollo Server - це реалізація серверної частини GraphQL, розроблена для використання з Apollo Client та іншими клієнтами GraphQL. Вона дозволяє легко створювати та розгортати GraphQL-сервери з мінімальними зусиллями.

Серед основнх аспектів використання Apollo Server є визначення схеми:

- Визначаємо GraphQL-схему, яка визначає типи даних та операції, які можна виконувати на сервері;
- Використовуємо типову мову запитів GraphQL (SDL - Schema Definition Language) для визначення схеми, як це зроблена на (рис. 2. 28).

```
type Todo {
    id: ID!
    text: String!
    completed: Boolean!
  }

  type Query {
    todos: [Todo]
  }

  type Mutation {
    toggleTodo(id: ID!): Todo
  }
```

Рисунок 2.28 – Типова мова запитів GraphQL

Ще один крок це налаштування Apollo Server:

- Встановлюємо Apollo Server та його залежності в проекті (зазвичай використовуючи npm або yarn);
- Створюємо екземпляр Apollo Server та передаємо йому схему та інші конфігураційні параметри, як це вказано на (рис. 2.29).

```
const { ApolloServer, gql } = require('apollo-server');

const typeDefs = gql`
  // Тут вставляємо GraphQL-схему
`;

const server = new ApolloServer({ typeDefs });
```

Рисунок 2.29 – Налаштування Apollo Server

Не менш важливою є реалізація резольверів, складается з функції, яка виконує запит та повертає відповідь. Приклад наведено на (рис. 2.30).

```
const resolvers = {
    Query: {
      todos: () => {/* Логіка отримання списку завдань */},
    },
    Mutation: {
      toggleTodo: (_, { id }) => {/* Логіка зміни статусу завдання */},
    },
  };
```

Рисунок 2.30 – Реалізація резольверів

Викличаємо метод listen для запуску сервера та прослуховування запитів GraphQL, як це вказано:

```
server.listen().then(({ url }) => {
   console.log(`Server ready at ${url}`);
 }).
```

Apollo Server дозволяє додавати middleware, розширювати функціональність та використовувати різні рівні кешування та оптимізації. Приклад наведено на (рис. 2.31).

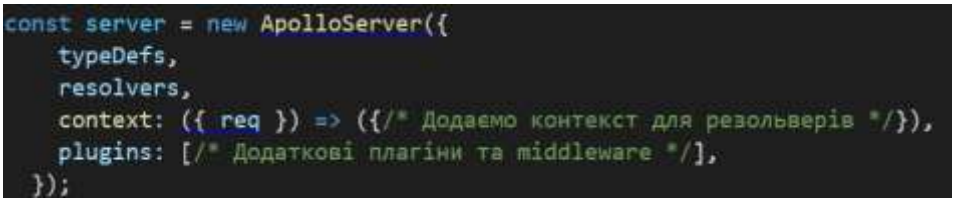


Рисунок 2.31– Повний приклад реілізації класу ApolloServer

Загалом, Apollo Server робить створення та розгортання GraphQL-серверів приємним та ефективним процесом, надаючи високий рівень абстракції та багатофункціональність для розробників.

2.15 Фреймворк Express.js для серверної частини

Express.js - це веб-фреймворк для Node.js, який дозволяє легко створювати веб-додатки та API. Основні особливості Express.js включають:

1. Маршрутизація**:** Express дозволяє визначати обробники запитів для різних шляхів (маршрутів) у вашому додатку;
2. Middleware**:** Ви можете використовувати middleware для обробки запитів перед тим, як вони досягнуть обробника запиту. Це дозволяє вам виконувати різні операції, такі як аутентифікація, логування тощо;
3. Шаблонізація**:** Express підтримує використання шаблонізаторів для генерації HTML (або інших форматів) на основі даних та шаблонів;
4. Управління статичними файлами: Вбудована підтримка для обробки та надсилання статичних файлів, таких як зображення, CSS та інші;

5. Middleware сторонніх розробників: Великий екосистем middleware сторонніх розробників, які полегшують роботу з різними аспектами веб-розробки.

Приклад реалізації серверу з використанням Express та Apollo Server.

В цьому прикладі, що показано на (рис. 2.46) використовується Express для створення серверу та Apollo Server для роботи з GraphQL. Схема GraphQL завантажується з файлу за допомогою GraphQLFileLoader, а потім передається у конфігурацію Apollo Server. Middleware Apollo Server додається до Express, і можемо слухати запити на порті 3000.

Цей приклад дозволяє легко поєднати можливості Express та Apollo Server для створення потужних та гнучких GraphQL-серверів:

```
import express from 'express';
import { ApolloServer } from 'apollo-server-express';
import { GraphQLFileLoader } from '@graphql-tools/graphql-file-loader';
import { loadSchemaSync } from '@graphql-tools/load';
const app = express();
// Завантажуємо GraphQL-схему з файлу (наприклад, schema.graphql)
const schema = loadSchemaSync('path/to/schema.graphql', {
  loaders: [new GraphQLFileLoader()],
});
// Створюємо Apollo Server з використанням схеми та інших конфігурацій
const server = new ApolloServer({ schema });
// Додаємо middleware Apollo Server до Express
server.applyMiddleware({ app });
// Запускаємо Express на порту 3000
app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000/graphql');
}).
```

2.16 СКБД PostgreSQL та Prisma ORM для управління базою даних

PostgreSQL - це об'єктно-реляційна система керування базами даних (ORDBMS), яка використовує SQL для взаємодії з даними. PostgreSQL є потужною та розширюваною СКБД з великою кількістю функцій, які підтримують стандарти SQL та забезпечують ефективність та надійність.

Порівнювати PostgreSQL з іншими системами керування базами даних (СКБД) можна залежно від конкретних потреб та вимог проекту, а також від особливостей кожної СКБД. Нижче подано загальний порівняльний огляд між PostgreSQL та деякими іншими відомими СКБД.

Порівняння PostgreSQL з MySQL:

PostgreSQL:
- Підтримка високого рівня стандартів SQL;
- Повна підтримка транзакцій та вищого рівня конфігурацій;
- Можливості розширення та додаткові функції.

MySQL:
- Простота використання та швидкість виконання;
- Відмінна підтримка для операцій зчитування даних;
- Використовується більше веб-проектами та стартапами.

Порівняння PostgreSQL з Oracle.

PostgreSQL:

- Відкритий код та безкоштовний;
- Гнучкість у роботі з різними типами даних;
- Заснований на спільноті та розвивається активно.

Oracle:

- Комерційна система з високою ціною;
- Великий екосистем та додаткові функції для корпоративних потреб;
- Використовується великими корпораціями та установами.

Порівняння PostgreSQL з Microsoft SQL Server.

PostgreSQL:

- Відкритий код та безкоштовний;
- Кросплатформенність, підтримка для різних операційних систем;

- Повна підтримка JSON та інших нетрадиційних типів даних.

Microsoft SQL Server:

- Комерційна система, пов'язана з екосистемою Microsoft;
- Широкий функціонал для бізнес-застосувань та великих підприємств;
- Інтеграція з іншими продуктами Microsoft.

Кожна з цих СКБД має свої переваги та недоліки, і вибір залежить від конкретних потреб та вимог проекту. PostgreSQL зазвичай визначається своєю гнучкістю, розширюваністю та відкритістю. Важливо ретельно оцінити потреби проекту, особливості кожної СКБД та екосистему, яка вас задовольнить.

Prisma - це сучасний інструмент для роботи з базами даних, який надає ORM (Object-Relational Mapping) та відображення схеми бази даних. Він підтримує різні бази даних, включаючи PostgreSQL, MySQL та SQLite. Prisma використовує декларативний підхід до взаємодії з базою даних та дозволяє легко виконувати операції CRUD (створення, читання, оновлення, видалення) через генеровані класи та методи.

Встановлюємо Prisma CLI та створюємо новий проект, як це показано:

npm install @prisma/cli -g

prisma init my-prisma-project.

Визначаємо конфігураційний файл schema.prisma та вказуємо підключення до бази даних (PostgreSQL), як це показано:

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
 }.
```

Запускаємо команду для генерації моделей та міграцій на основі схеми, командами, які показані:

-npx prisma generate;

-npx prisma migrate dev.

Використовуємо згенеровані Prisma класи та методи для взаємодії з базою даних. Наприклад, якщо є модель User, можна використовувати метод findMany для отримання всіх користувачів, const users = await prisma.user.findMany().

Також можна використовувати методи для виконання операцій CRUD, так як це зазначено на (рис. 2.32).

```
const newUser = await prisma.user.create({
    data: {
      name: `Oleh Nikulin`,
      email: `oleh.nikulin@nure.ua`,
    },
  });
```

Рисунок 2.32 – Створення користувача за допомогою метода prisma.user.create

Цей приклад показує основні кроки використання Prisma з PostgreSQL. Prisma дозволяє легко та зручно взаємодіяти з базами даних, а генеровані класи та методи полегшують створення безпечних та ефективних запитів до бази даних.

Зважаючи на те, що Prisma надає абстракцію від конкретних SQL-запитів, порівняння буде включати прості приклади CRUD-операцій між Prisma та чистими SQL-запитами для PostgreSQL.

Prisma дозволяє виконувати ті ж операції CRUD, що й чисті SQL-запити, але робить це за допомогою зручного та безпечного інтерфейсу за допомогою генерованих методів та класів.

CRUD - акронім, що позначає чотири базові функції, що використовуються під час роботи з базами даних: створення, читання, модифікація, видалення. Введений Джеймсом Мартіном в 1983 як стандартна класифікація функцій з маніпуляції даними.

2.17 Бібліотеки Formik та Yup

Formik - це бібліотека для управління формами в React. Вона надає зручний та потужний інтерфейс для створення та керування формами, дозволяючи легко обробляти стан, валідацію, подачу та інші операції пов'язані з формами. Formik спрощує процес розробки форм та забезпечує багато корисних функцій, таких як управління станом, валідація, подача форми та обробка помилок.

Yup - це бібліотека для валідації об'єктів в JavaScript та TypeScript. Вона дозволяє вам визначати схеми та правила валідації для вашого даних зручним та декларативним способом. Yup є популярним інструментом у світі реактивного програмування та веб-розробки, особливо коли йдеться про валідацію даних введених користувачами.

У цьому прикладі на (рис. 2.33) використовується Yup для визначення схеми валідації, яка включає правила для поля "name" (обов'язкове поле, строка) та "email" (обов'язкове поле, електронна адреса).

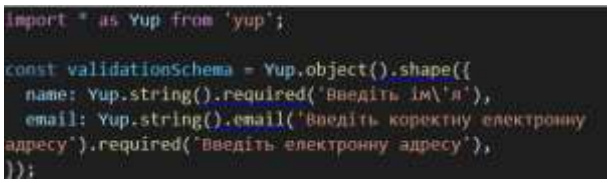


Рисунок 2.33 – Схема валідації

Рисунок 2.34 – Підключення бібліотек yup до Formik

У цьому прикладі, що на (рис. 2.34) використовується Formik разом із Yup для створення форми з валідацією. Схема валідації Yup передається як параметр validationSchema в компонент Formik. Yup автоматично застосовує валідацію для полів форми на основі визначених правил в схемі.

2.18 Висновки до розділу

У цьому розділі було розглянуто сучасні технології для веб розробки, порівняно аналоги, досліджено плюси та мінуси кожної з них. Було наведено приклади використання технології та описано їх роботу.

3. АРХІТЕКТУРА ЗАСТОСУНКУ

Архітектура веб-застосунку - це структурний план та організація веб-застосунку, який визначає розподіл функцій, модулів та компонентів, а також взаємодію між ними. Гарна архітектура дозволяє розробникам легко розширювати та підтримувати код, а також підвищує працездатність, масштабованість та безпеку веб-застосунку. Основні складові архітектури веб-застосунків включають клієнт-серверну архітектуру:

- Застосунок розділяється на дві основні частини: клієнт (браузер або мобільний додаток) та сервер (сервер, що обробляє запити);
- Розділення забезпечує масштабованість, оскільки можна масштабувати клієнтську та серверну частини незалежно одна від одної.

Веб-додатки використовують також однорівневу або монолітну архітектури:

- Всі компоненти та функціональність розташовані на одному сервері чи в одному кодовому базисі;
- Всі компоненти та функціональність розташовані на одному сервері чи в одному кодовому базисі;
- Легко встановлювати та підтримувати, але може бути менш масштабованою та менш гнучкою.

Розділення на клієнтську та серверну частину (Client-Server):

- Всі клієнтські та серверні компоненти розташовані на своїх серверах;
- Забезпечує покращену масштабованість та гнучкість, але може вимагати більше зусиль для розробки та підтримки.

Модульність:

- Розділення застосунку на модулі чи компоненти допомагає полегшити розробку та зробити код більш підтримуваним;
- Кожен модуль виконує конкретну функцію та може бути незалежно розвиваним та тестованим.

Сервіс-Орієнтована архітектура (Service-Oriented Architecture - SOA):

- Застосунок складається з набору незалежних сервісів, кожен із яких надає конкретну функціональність;

- Забезпечує високу гнучкість, можливість масштабування та повторне використання коду.

Мікросервісна архітектура:
- Великий застосунок розбивається на невеликі та незалежні мікросервіси;
- Кожен мікросервіс може бути розвиваний та масштабований незалежно;
- Забезпечує гнучкість та швидкість видачі нових функціональних можливостей.

RESTful API або GraphQL:
- Застосунок може використовувати RESTful API або GraphQL для обміну даними між клієнтом та сервером;
- RESTful API базується на ресурсах та методах HTTP, тоді як GraphQL надає гнучкий підхід до отримання даних.

У атестаційній роботі використовується комбінація двох вищеперераховних підходів до побудови проекту, це Розділення на клієнтську та серверну частину (Client-Server), тобто проект розділений на дві частини, які пов’язані між собою GraphQL запитами.

3.1 Структурна організація проекту

Як було зазначено вище проект використовує Client-Server архітектуру і розділений на дві частини. Важливо зазначити, що проект використовує технологію монорепозиторію. Монорепозиторій (Monorepo) - це тип репозиторію для управління різними проектами чи компонентами програмного забезпечення в одному централізованому місці. Замість того, щоб розділяти код на кілька окремих репозиторіїв, всі проекти чи компоненти розташовані в одному репозиторії.

Проект ініціалізован за допомгою Yarn package manager. Yarn - це менеджер пакетів для мови програмування JavaScript, призначений для ефективного управління залежностями та пакетами в проектах.

Yarn Workspaces - це функціональність, яка дозволяє управляти залежностями та пакетами в монорепозиторії за допомогою Yarn, популярного менеджера пакетів для JavaScript. З допомогою цієї функціональності, можна працювати з кількома проектами в одному репозиторії та спільно використовувати залежності між ними.

Основні особливості Yarn Workspaces:

- Репозиторій містить кілька проектів, і вони можуть спільно використовувати однакові залежності, що спрощує управління версіями та зменшує обсяг файлів;
- Залежності кожного проекту розміщені в спільному node_modules на рівні репозиторію, що дозволяє економити місце та покращує ефективність;
- Yarn надає команди вищого рівня для управління всіма проектами в репозиторії, такі як yarn install, яка встановлює всі залежності для всіх проектів;
- Можливість використовувати спільні скрипти для всіх проектів, що полегшує підтримку та розвиток кодової бази.

На (рис 3.1) наведено загальну структуру монорепозиторію. У корневій папці Phonoteka знаходяться наступні файли та папки:

1. .vscode – папка з налаштуваннями редактору;
2. node_modules – папка з встановленими спільмини пакетами для серверної та клієнтської частини проекту;
3. packages – папка з репозиторіями client та сервер;
4. .eslintrc.js – файл з налаштуванням правил для написання коду;
5. .gitignore – файл з шляхами до папок та файлів, що не потрібно включати до гіта;
6. package.json – файл з налаштуванням спільних пакетів та монорепозиторію;
7. prettier.config.js – ще один файл з налаштуванням правил для написання коду;
8. yarn.lock – файл містить конкретні версії кожної залежності та всіх підзалежностей. Це дозволяє забезпечити встановлення та використання тих самих версій пакетів для всіх членів розробницького колективу або на різних системах.

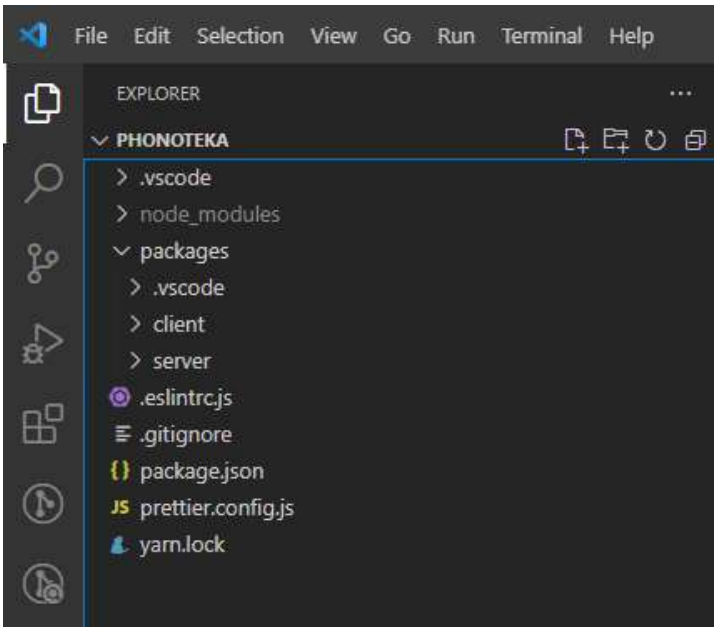


Рисунок 3.1 – Загальна структура монорепозиторію

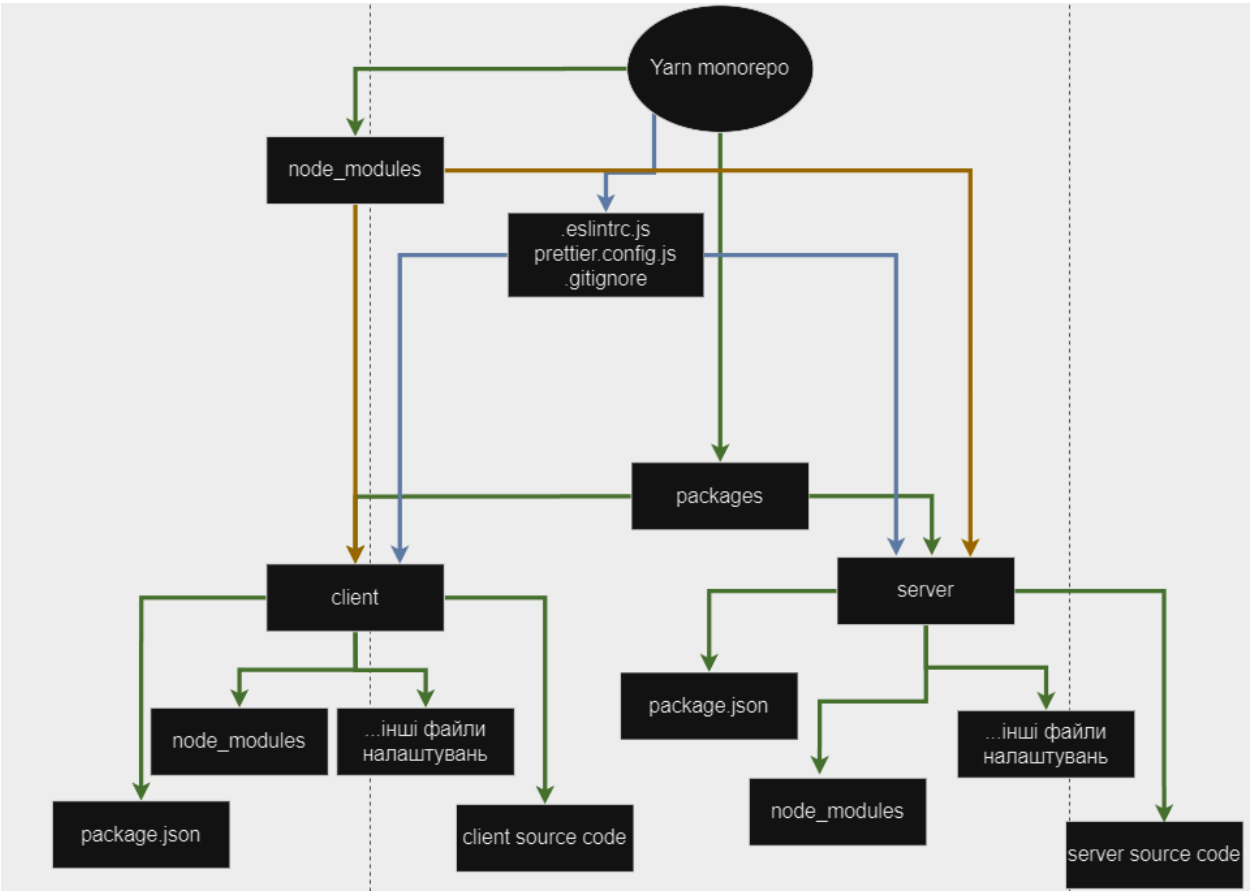


Рисунок 3.2 – Схема розподілу інформації в монорепозиторії

На (рис. 3.2) продемонстровано, що ярн розділяє дані між усіма вкладеми репозиторіями у папці packages.

3.2 Вміст папки client

Основна папка client містить:

- Папку apollo – яка містить у собі папку з мутаціями та кверями, необхідними для комунікації з сервером. Також присутній файл client.ts в якому знаходится основна конфігурація Apollo client. Структуру наведено на (рис. 3.3);

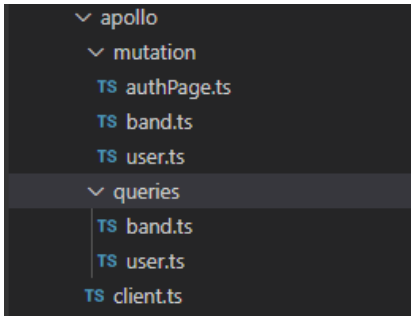


Рисунок 3.3 – Структура папки apollo

- Папку assests, яка містить у собі декалька картинок, які показано на (рис. 3.4), що використовуються у проекті, в тому числі логотип;

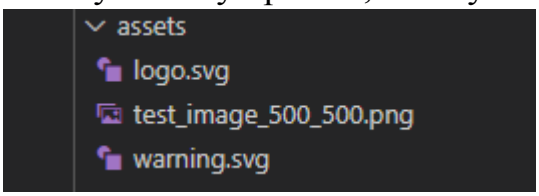


Рисунок 3.4 – Структура папки assests

- Папку components, яка містить більшість файлів, які використані для написання проекту.

Папка Components містить наступні компонети, що наведено на (рис. 3.4):

- ApplicationWrapper;
- Footer;
- Header;
- Modals;
- ProfileChart;
- Table;
- TableControlsHeader;
- UI;

- UserProfile.

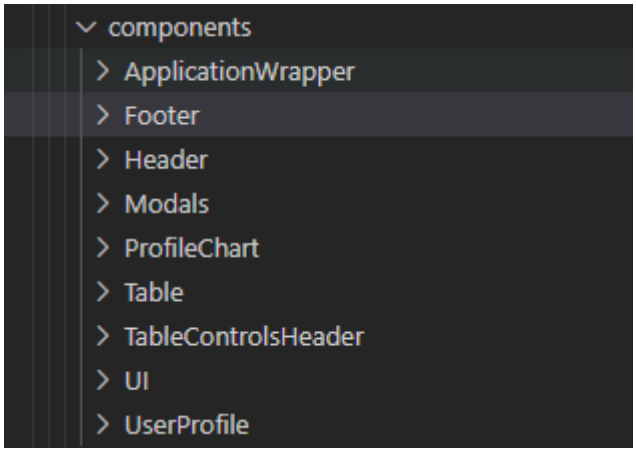


Рисунок 3.4 – Структура папки components

Загалом кожна з папок містить однонайменні файли з росширенням .tsx – сама React компонента, та .scss – файл стилів для компоненти, але є винятки, серед яких це:

- Modals – вміст папки показано на (рис. 3.5). Папка містить компоненти з модальними вікнами та допоміжні файли для роботи з формами в цих вікнах;

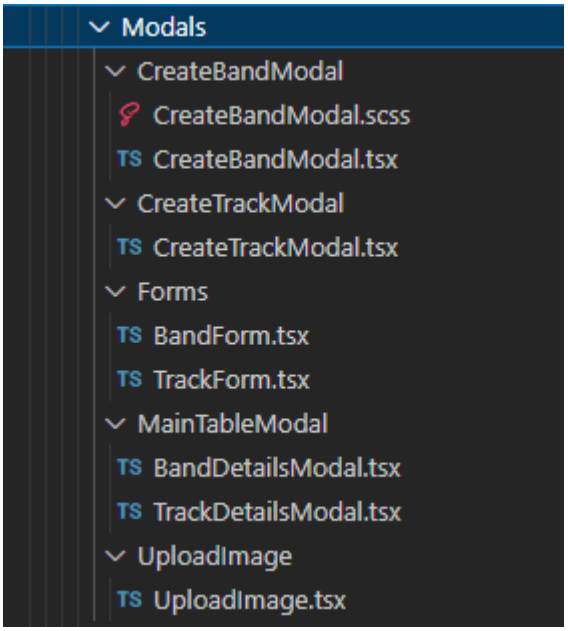


Рисунок 3.4 – Структура папки Modals

- Table – вміст папки показано на (рис. 3.5). Папка містить основний компонент Table та допоміжні компоненти для побудови таблиці;

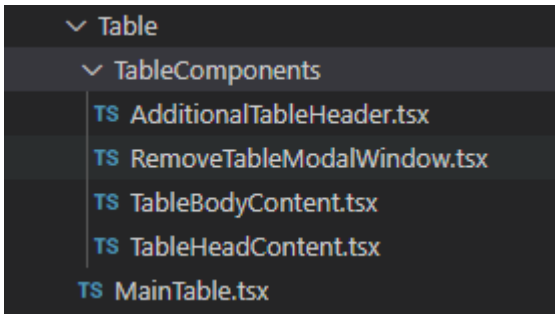


Рисунок 3.5 – Структура папки Table

- UI – вміст папки показано на (рис. 3.6). Папка містить UI елементи, які можуть бути перевикористані у всьмоу проекті. Це допомогає притримуватись єдених стилів та мінімізувати код. Підпапка MUI містить styled MUI компоненти;

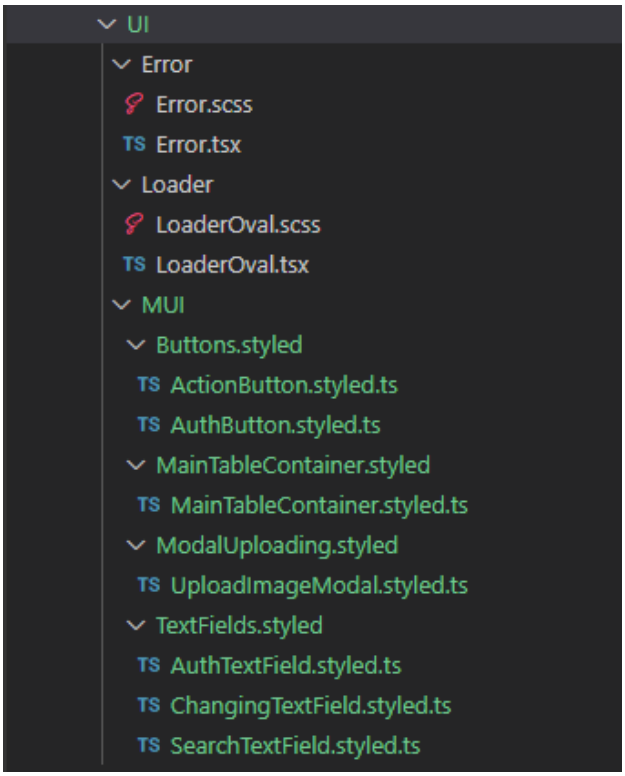


Рисунок 3.6 – Структура папки UI

Папка pages, вміст папки наведено на (рис. 3.7), яка містить всі сторінки проекту:

- Login page;
- Register page;
- Main page;
- User profile page;

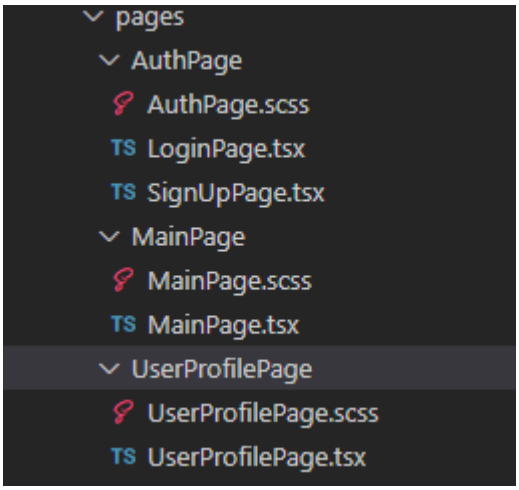


Рисунок 3.7 – Структура папки pages

- Папку reactiveVars, яка містить файл з змінними для Apollo client, структуру показано на (рис. 3.8);

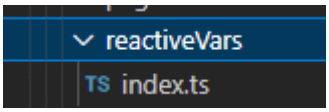


Рисунок 3.8 – Структура папки reactiveVars

- Папку utils, яка містить файли необхідні для виконання вторичної роботи, які наведено на (рис. 3.9), наприклад для визначння чи приватний роут, ініціалізації хмарної бази даних Firabase чи додавання токену до куків;

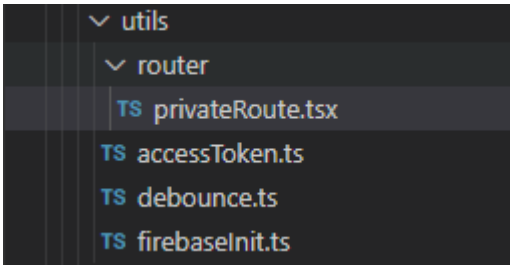


Рисунок 3.9 – Структура папки utils

- Папку validations, яка містить файлі валідації з використанням Yup. Структуру наведено на (рис. 3.10);

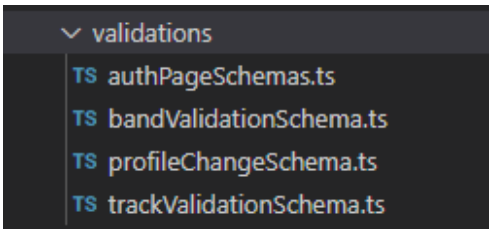


Рисунок 3.10 – Структура папки utils

- Папку variables, яка містить декілька файлів з константами для використання TypeScript, або константи з кольорами. Структура папки наведена на (рис. 3.11);

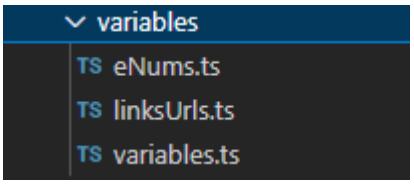


Рисунок 3.11 – Структура папки utils

Останні файли, що лежать в папці src та client є базовими файлами з ініціалізацією проекта, їх можна побачити на (рис. 3.12), наприклад App.tsx, apollo-json.config.ts, tsconfig.json або index.html, який є вхідним файлом до всього коду.

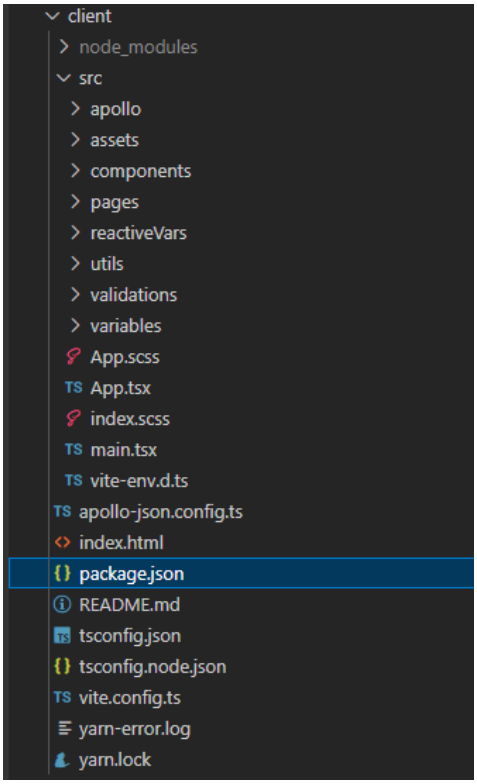


Рисунок 3.12 – Структура клієнтської частини проекту

3.3 Вміст папки server

Основна папка server містить:

- Папку graphql, яка містить папки з кверями та резолверами, також основний graphql файл в якому прописані шляхи за якими звертається клієнт до серверу. Зміст папки приведено на (рис. 3.13);

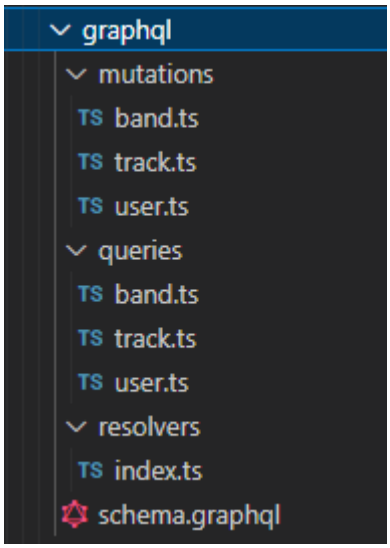


Рисунок 3.13 – Структура папки graphql

- Папку middlewares, яка містить лише один файл – authenticate, який використовується для перевірки токену користувача. Структура наведена на (рис. 3.14);

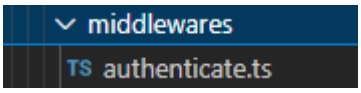


Рисунок 3.14 – Структура папки middlewares

- Папку prisma, яка містить налаштування бази даних PostgreSQL та Prisma ORM для керування запитами до бази. Також присутня папка з міграціями, це розробницькі файли, яки з’являлися під час написання коду та модифікування бази даних. Структуру наведено на (рис. 3.15);

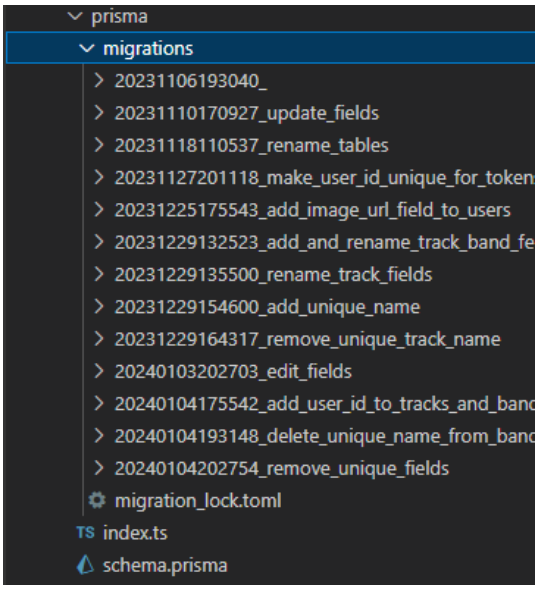


Рисунок 3.15 – Структура папки prisma

- Папку utils, яка містить файли для обчислень, які повторюються в коді, наприклад генерація токену, або його відправка у куках. Структуру наведено на (рис. 3.16);

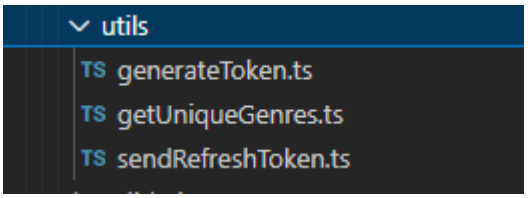


Рисунок 3.16 – Структура папки utils

- Папку validations, яка містить файли валідації на основі Yup для вхідних запитів через GraphQL. Структуру наведено на (рис. 3.17);

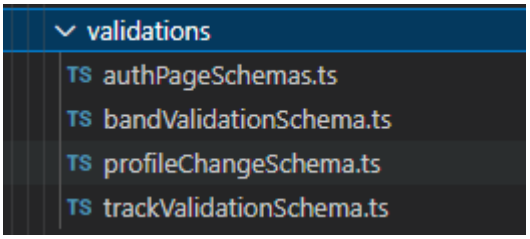


Рисунок 3.17 – Структура папки validations

Останні файлі, що лежить в паці server, та які наведено на (рис. 3.18) є основними файлами ініціалізації сервеної частини проекту. Наприклад: index.ts – основний файл, який запускає сервер, .env файл з змінними оточення, або tsconfig.json з конфігурацією TypeScript.

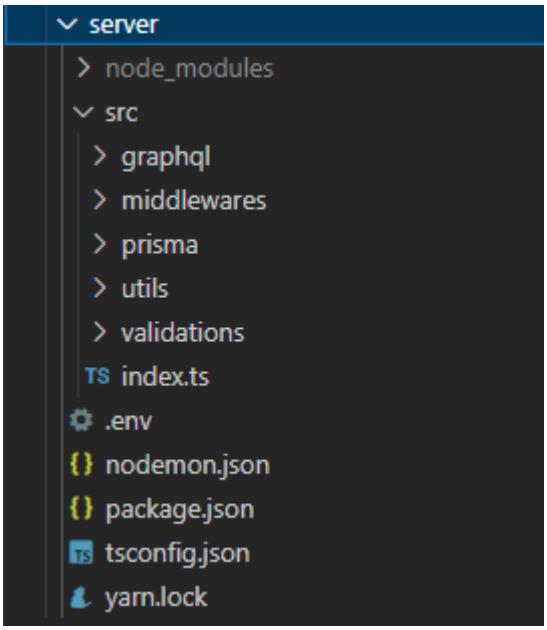


Рисунок 3.18 – Структура серверної частини

3.4 Висновки до розділу

У цьому розділі було розглянуто види архітектур побудови веб-застосунків. Описано плюси та мінуси кожного підходу та аргументовано вибір архітектури для подальшої розробки.

4. РОЗРОБКА ФУНКЦІОНАЛУ

Перед написанням коду необхідно ініціалізувати монорепозиторій за допомогою Yarn, встановити базові пакети для розробки, налаштувати правила для форматування та стилю коду, зробоити репозиторій за допомгою Git та зробити деплой базової заготовки на основній dev гілці до GitHub.

Для ініціалізації знадоблятся наступні програми для ОС Windows 10:

- VS Code;
- Node.js;
- PostgreSQL;
- PGAdmin;
- Git.

4.1 Встановлення програми Visual Studio Code

Завантажуємо та Встановлюємо Visual Studio Code з офіційного сайту, який показано на (рис. 4.1).

Рисунок 4.1 – Приклад офіційного сайту VS Code

Після установки відкривається програма с пустим вікном, як на (рис. 4.2).

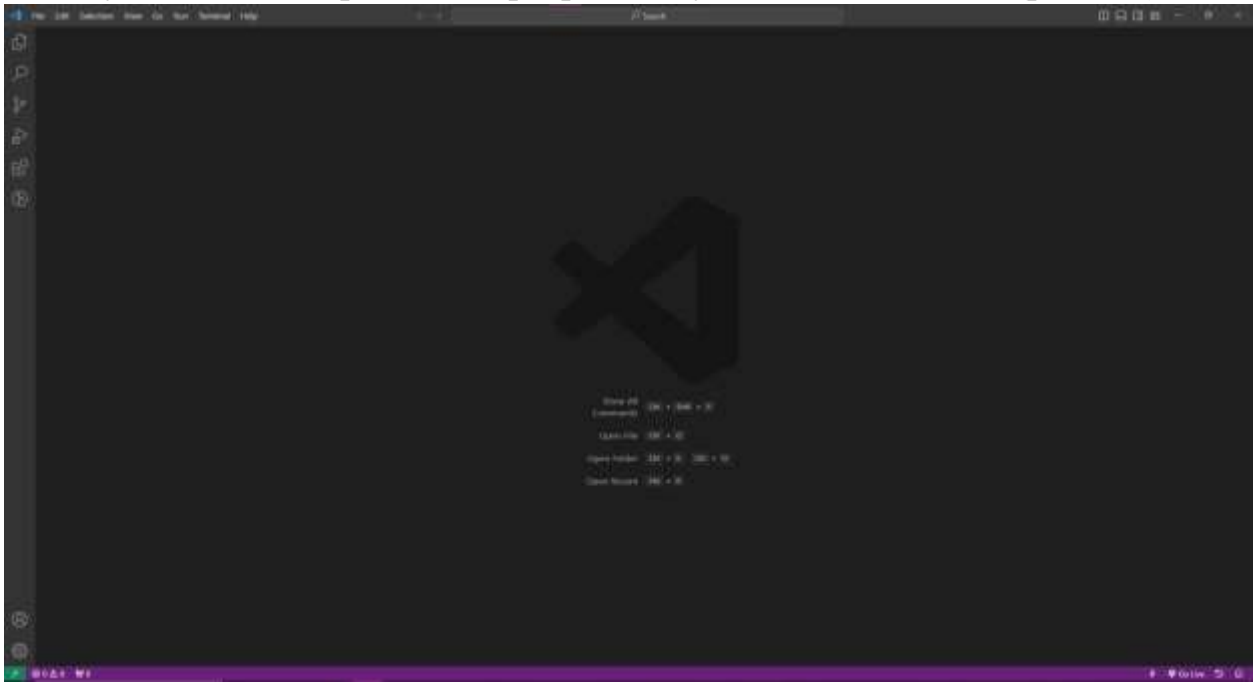


Рисунок 4.2 – Встановлена програма VS Code

Встановимо необхідні плагіні, які поліпшать процес написаня коду. Приклад деяких з них наведено на (рис. 4.3):

- Auto Rename Tag – усі парні html теги будуть автоматично змінювати назву;
- Code Spell Checker – перевірка написання різніх слів, щоб уникнути непотрібних казусів;
- ESLint – плагін для налаштування правил для код стилю;
- Prettier - Code formatter – це плагін для форматування коду у читабельний вид за правилами, які будуть описані в файлі налаштувань;
- Git essentials – плагін з набором основних плагінів по роботі з системою управління версіями git;
- GraphQL for VSCode – VS Code за замовчення не підтримує синтаксис файлу .graphql, цей плагін вирішує цю пробелму;
- Prisma – VS Code за замовчення не підтримує синтаксис файлу .prisma, цей плагін вирішує цю проблему.

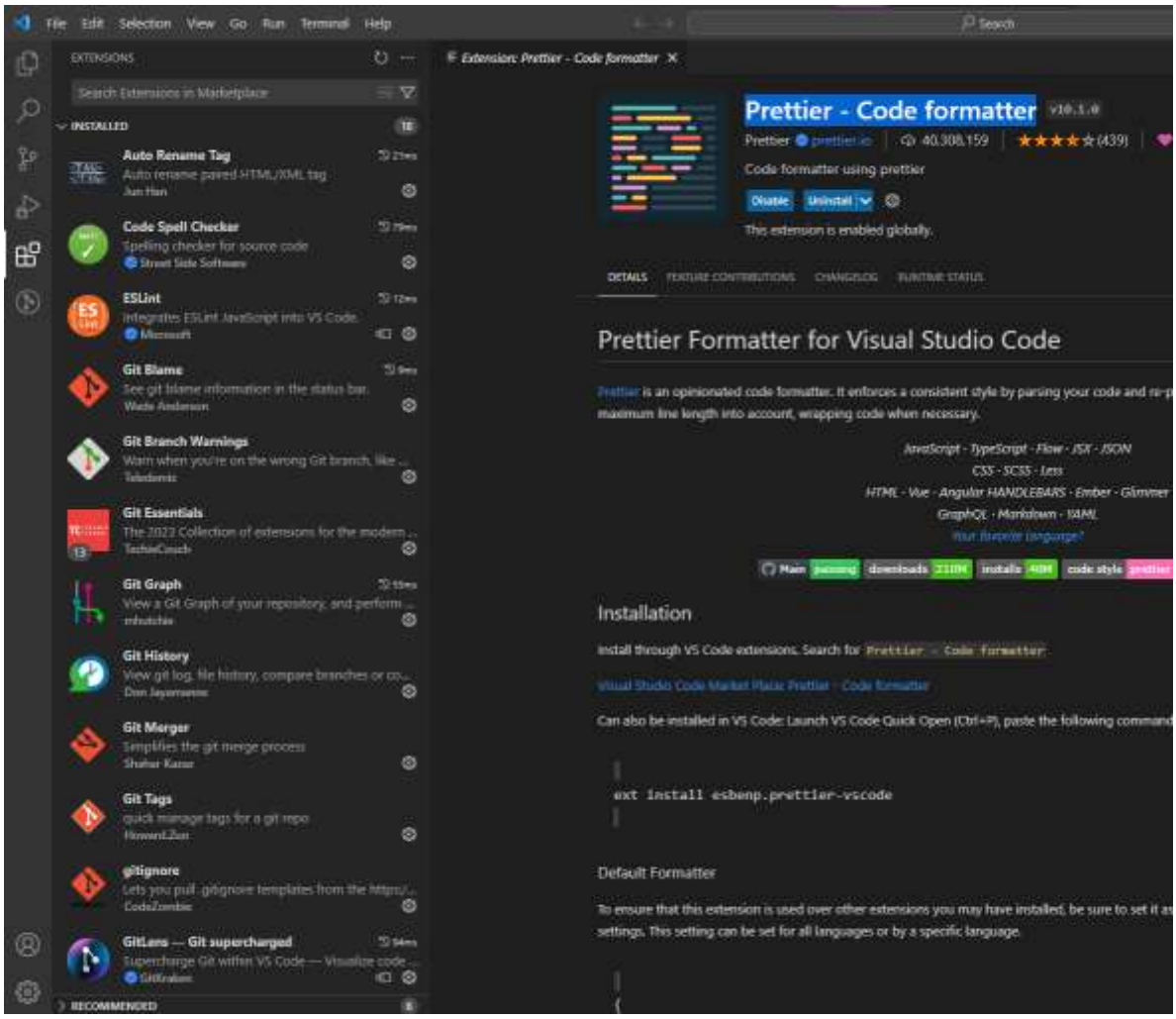


Рисунок 4.3 – Приклад встановлених плагінів в VS Code

4.2 Встановлення програми Node.js

Для встановлення Node.js перейдемо до офіціального сайту, який показано на (рис. 4.4) та завантажимо програму, дотримуючись базових інструкцій встановлення буде закінчено та після цього нам нічого робити не потрібно. Встановлену програму можно перевірити за допомогою команди node –v у командному вікні програми VS Code, як це показано на (рис. 4.5) та побачити встановленую версію ноди, у моєму випадку це версія 18.16.0.

Рисунок 4.4 – Приклад офіціального сайту Node.js

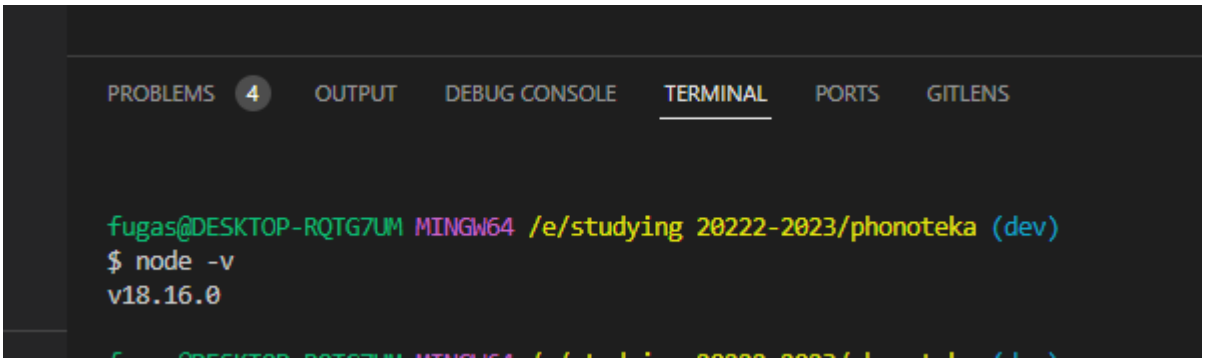


Рисунок 4.5 – Приклад перевірки інсталяції Node.js

4.3 Встановлення програми PostgreSQL

Для встановлення PostgreSQL перейдемо до офіціального сайту, який показано на (рис. 4.6) та завантажимо програму, дотримуючись базових інструкцій встановлення буде закінчено та після цього потрібно перезавантажити комп’ютер. Встановлену програму можно перевірити у диспетчері задач, декілька процесів повині фоново запуститись після перезавантаження комп’ютера, як це показано на (рис. 4.7).

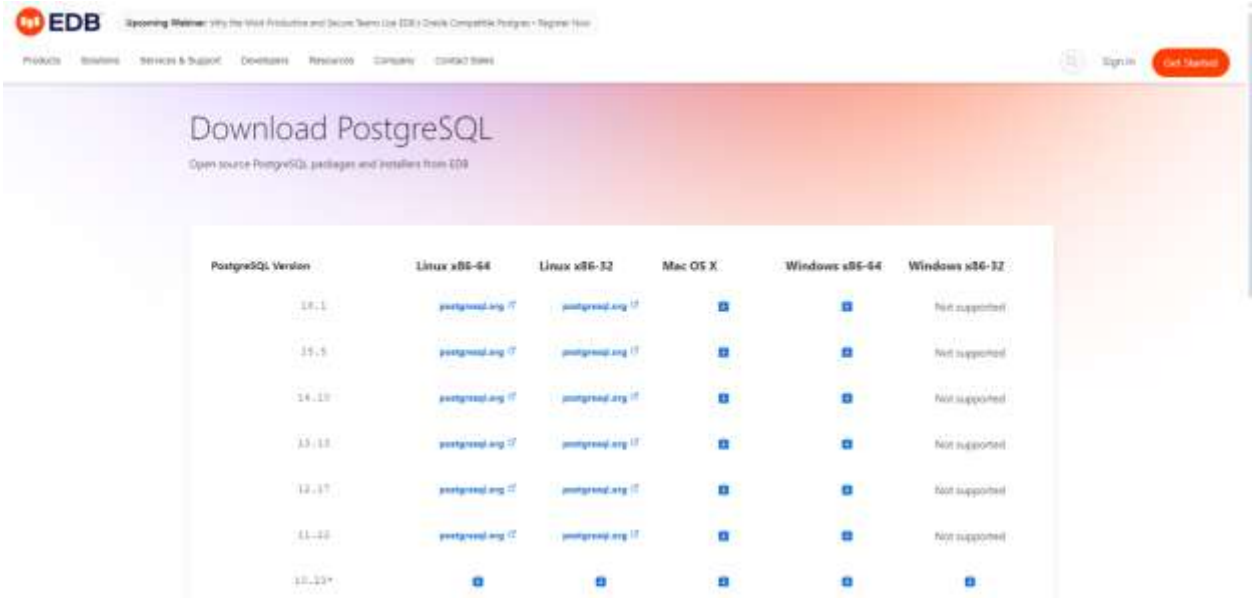


Рисунок 4.6 – Приклад офіціального сайту PostgreSQL

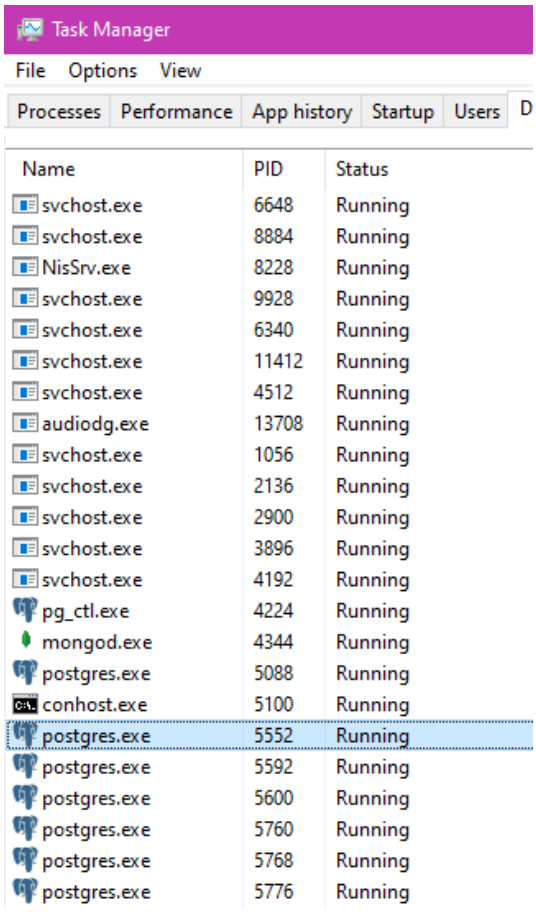


Рисунок 4.7 – Приклад встановленної СКБД PostgreSQL

4.4 Встановлення програми PGAdmin

Для встановлення PGAdmin перейдемо до офіціального сайту, який показано на (рис. 4.8) та завантажимо програму, дотримуючись базових інструкцій встановлення буде закінчено та після цього нам нічого робити не потрібно.

Рисунок 4.7 – Приклад офіціального сайту PGAdmin

Ця програма необхідна для зрозумілої візулізації таблиць та перевірки данних у них. Приклад приведено на (рис. 4.8).

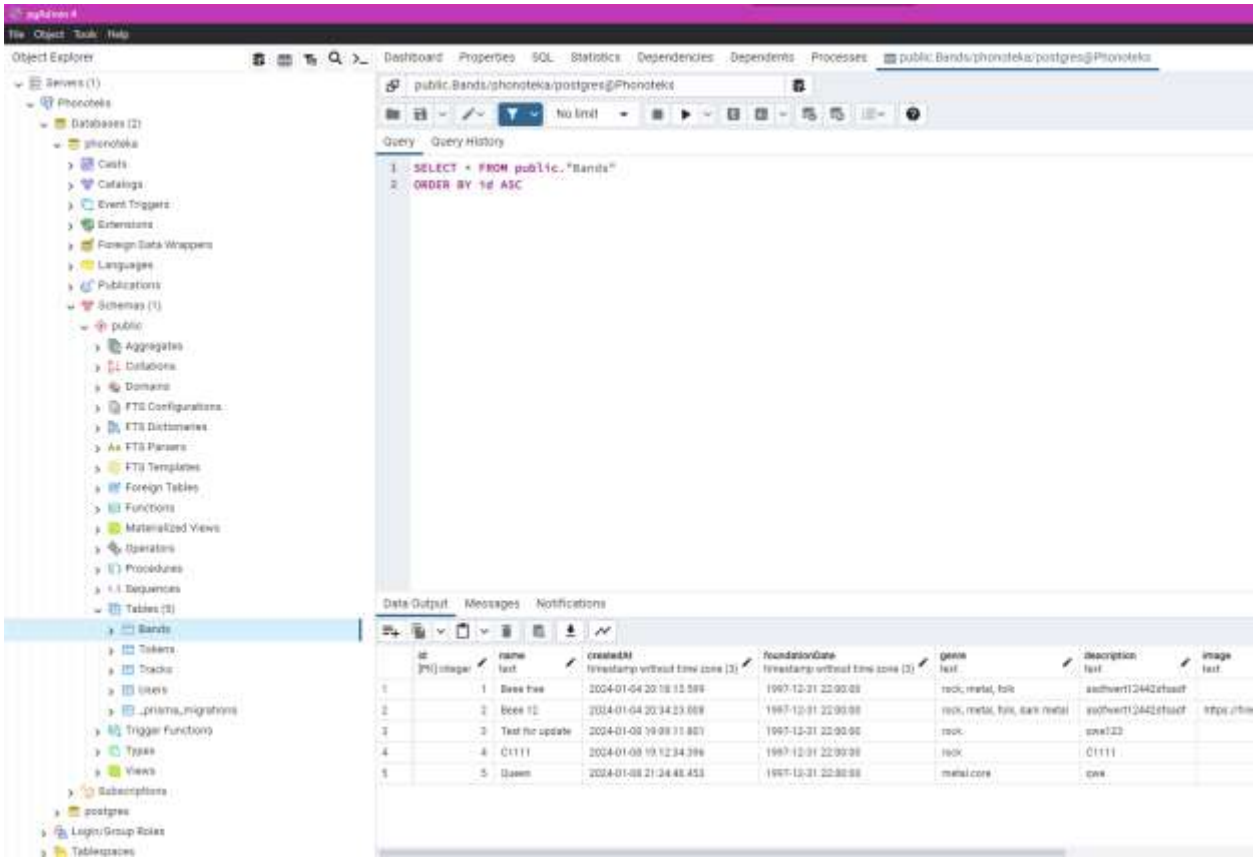


Рисунок 4.8 – Приклад використання програми PGAdmin

4.5 Встановлення програми Git

Для встановлення Git перейдемо до офіціального сайту, який показано на (рис. 4.9) та завантажимо програму, дотримуючись базових інструкцій встановлення буде закінчено та після цього нам нічого робити не потрібно.

Git - це система керування версіями, що використовується для відстеження змін у програмному коді та спільної роботи над проектами розробки програмного забезпечення. Git дозволяє розробникам вести історію змін, ефективно співпрацювати з іншими розробниками та відмічати версії.

Git - це система керування версіями, що використовується для відстеження змін у програмному коді та спільної роботи над проектами розробки програмного забезпечення. Git дозволяє розробникам вести історію

змін, ефективно співпрацювати з іншими розробниками та відмічати версії програмного коду.

Основні команди Git:

- git init: Ініціалізує новий репозиторій;
- git clone <URL>: Клонує існуючий репозиторій;
- git add <файл>: Додає зміни файлу до індексу;
- git commit -m "Коментар до коміту": Зберігає зміни, що були додані до індексу;
- git status: Показує стан робочого каталогу та індексу;
- git log: Показує історію комітів;
- git branch: Показує список гілок, активну гілку позначено зірочкою;
- git checkout <гілка>: Перемикається між гілками;
- git merge <гілка>: Зливає зміни з іншої гілки в поточну;
- git pull: Завантажує зміни з віддаленого репозиторію та об'єднує їх з локальним репозиторієм;
- git push: Відправляє коміти на віддалений репозиторій;
- git remote add origin <URL>: Додає віддалений репозиторій під ім'ям "origin";
- git diff: Показує різницю між робочим каталогом і індексом;
- git reset <файл>: Скасовує зміни, що були додані до індексу;
- git rm <файл>: Видаляє файл із відстежування.

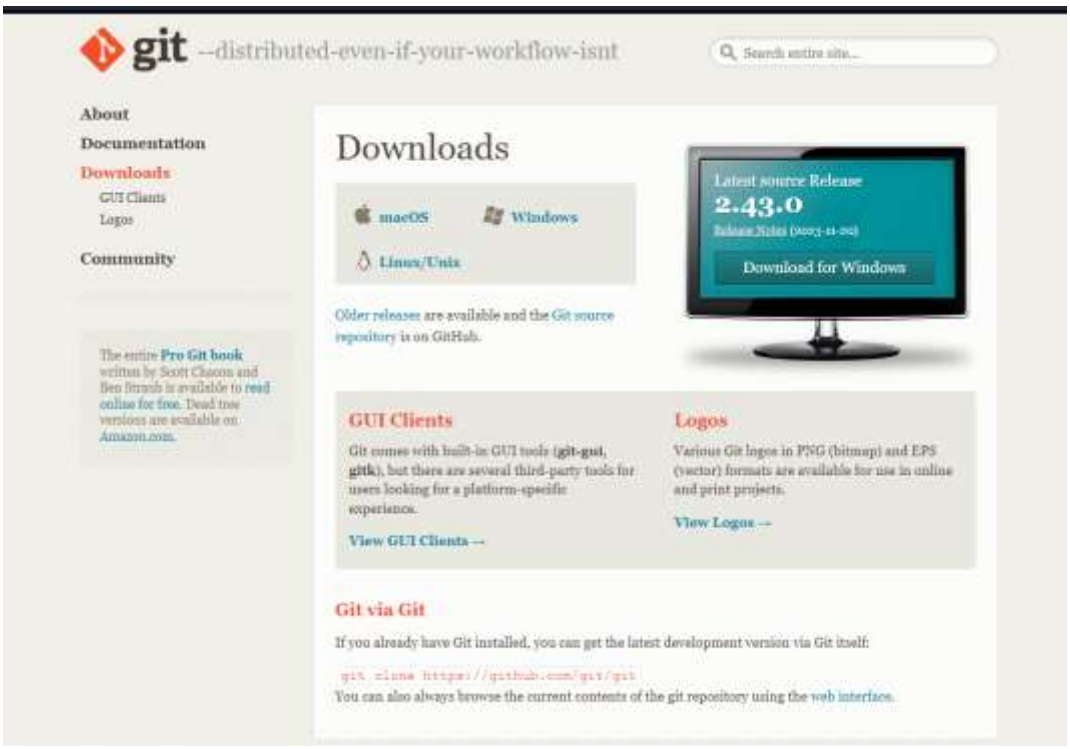


Рисунок 4.9 – Приклад офіціального сайту Git

4.6 Первинна ініціалізація проекту та створення репозиторію

Ініціалізація проекту включає в себе наступні кроки:

1. Створення монорепозиторію;
2. Встановлення базових пакетів до кліенту;
3. Встановлення базових пакетів до серверу;
4. Встановлення та конфігурація ESlint та Prettier;
5. Конфігурація TypeScript;
6. Створення репозиторію на GitHub та перший пуш.

4.7 Створення монорепозиторію

За допомогою команди “yarn init” у корнівій папці Phonoteka ініціалізуємо проект, з’явится перший файл package.json. Після цього необхідно створити папки з назвою packages в котрій будут міститись папки client та server. Коли папки створені можна приступити до редагування package.json файлу. Додамо наступні поля, які показано на (рис. 4.10):

- name - назва проекту;
- version - це поле вказує на версію проекта. Версія допомагає визначити, яка саме використовується та які оновлення були внесені;
- description - опис поля містить короткий опис проекта. Це може бути корисним для інших розробників, які шукають інформацію про проект;
- author - це поле вказує на автора чи авторів проекта;
- license - ліцензія, яку можливо обрати для свого проекту. Це поле важливе для визначення правил використання коду іншими користувачами;
- private - якщо це поле встановлено в true, то цей проекту не може бути опублікований на npm. Використовується, наприклад, для приватних проектів, які не призначені для загального використання;
- workspaces - це поле вказує, які каталоги (робочі області) повинні бути включені в монорепозиторій. Можливо вказати папки, які містять пакети, щоб Yarn або npm розпізнавали їх як частину спільного проекту. "packages/* означає, що всі папки в цій паці будуть включені до монорепозиторію.

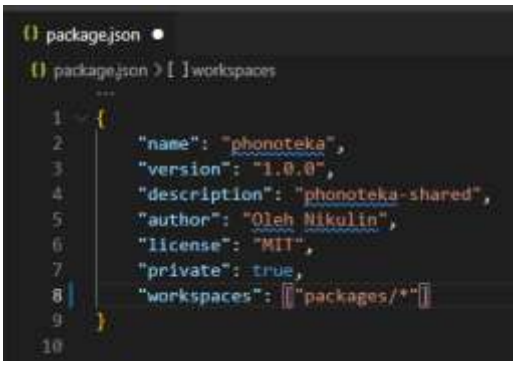


Рисунок 4.10 – Зміст файлу package.json

4.8 Встановлення базових пакетів до кліенту

Для ініціалізації кліентної частини знобидться написати команду знаходячись у папці packages: yarn create vite –template react-ts.
Збірка кліента буде проходити за допомогою Vite.

Vite - це новий інструмент для розробки веб-застосунків, призначений для забезпечення швидкості та ефективності. Розроблений Evan You, творцем фреймворку Vue.js, Vite виокремлюється як "Next Generation Frontend Tooling" та відмінно працює як для розробки малих проектів, так і для великих, завдяки своїй унікальній архітектурі.

Основні особливості Vite:

1. Швидкість розробки: Vite дозволяє запускати розробку на основі ESM (ECMAScript Modules) без передкомпіляції. Це робить процес розробки швидшим і миттєвим;
2. Миттєва перезавантаження: Під час розробки Vite може виготовляти лише ті ресурси, які потрібні для конкретного компонента або сторінки. Це забезпечує миттєве перезавантаження під час редагування кодуж;
3. Уніфікована розробка для Vue, React, і Svelte: Vite підтримує різні фреймворки, такі як Vue, React і Svelte, а також може бути використаний для створення звичайних HTML/CSS/JS проектів;
4. Гаряча заміна модулів (HMR): Vite використовує HMR для миттєвої заміни змінених модулів без повного перезавантаження сторінки;
5. Підтримка TypeScript: Vite надає нативну підтримку TypeScript, дозволяючи вам легко інтегрувати TypeScript;
6. Безпечні за замовчуванням налаштування CORS: Vite конфігурується так, щоб бути безпечним для використання CORS з великою кількістю додатків.

Vite надає новий погляд на розробку веб-застосунків, спрощуючи процес та забезпечуючи високу продуктивність під час розробки.

Після цього буде запропановано ввести назву проект у моєму випадку це client, це виглядає як на (рис. 4.11).

```
fugas@DESKTOP-RQTG7UM MINGW64 /e/Masters 2023/moe/examples/phonoteka/packages (master)
$ yarn create vite --template react-ts
yarn create v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Installed "create-vite@5.1.0" with binaries:
      - create-vite
      - cva
? Project name: » client
```

Рисунок 4.11 – Ініціалізації кліентної частини

Ця команда створить базовий темплейт вже з заповниними файлами основних конфігурація такі як: package.json, tsconfig,json, .eslint.cjs, які пізніше будуть відредаговані. Це показано на (рис. 4.12).

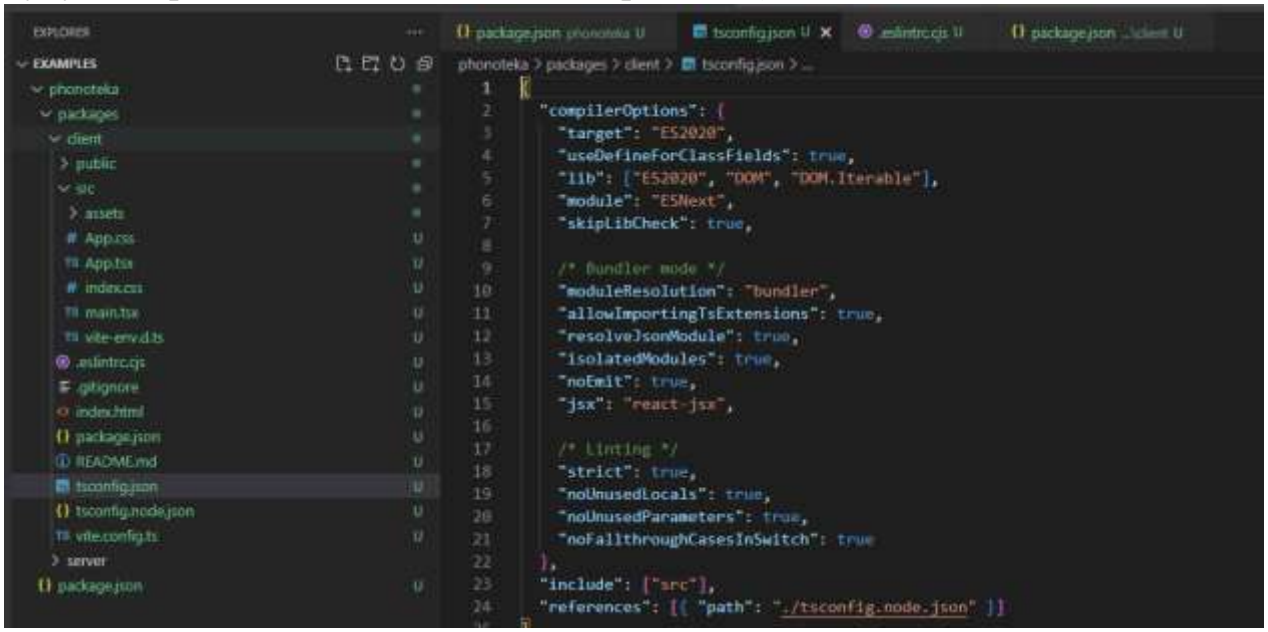


Рисунок 4.12 – Базовий Vite темплейт

Тепер встановимо основі пакети для роботи з кліентською частиною за допомогою команди:
“yarn add @apollo/client @emotion/react @emotion/styled @mui/icons-material @mui/material @mui/x-charts apollo-link-token-refresh chart.js firebase formik graphql js-cookie jwt-decode moment react react-cookie react-dom react-loader-spinner react-player react-router react-router-dom uuid yup @types/js-cookie @types/react @types/react-dom @types/uuid @vitejs/plugin-react sass vite –dev”.

Розглянемо кожен з пакетів з списку та їхні основні функції:

1. @apollo/client: Клієнт Apollo для роботи з GraphQL. Забезпечує інструменти для взаємодії з сервером GraphQL та роботи з даними;
2. @emotion/react та @emotion/styled: Бібліотека для роботи з емоційними стилями в React. Забезпечує зручний спосіб створення та використання стилів;
3. @mui/icons-material та @mui/material: Матеріальні компоненти та ікони для MUI (Material-UI). Material-UI - це бібліотека React-компонентів для створення інтерфейсу в стилі Material Design;
4. @mui/x-charts: Розширені графіки для Material-UI;
5. apollo-link-token-refresh: Apollo Link для автоматичного оновлення токенів авторизації;
6. chart.js: Бібліотека для створення інтерактивних графіків та діаграм на веб-сторінках;
7. firebase: Клієнт Firebase для роботи з Firebase, який забезпечує різноманітні хмарні послуги, такі як база даних та інші;
8. formik: Бібліотека для управління формами в React. Забезпечує зручні інструменти для валідації та обробки форм;
9. graphql: Основний пакет GraphQL для визначення типів та використання GraphQL мови запитів;
10. js-cookie: Бібліотека для роботи з cookie в JavaScript;
11. jwt-decode: Бібліотека для декодування JWT (JSON Web Tokens);
12. moment: Бібліотека для роботи з датами та часом у JavaScript;
13. react: Основна бібліотека React для створення інтерфейсу користувача;
14. react-cookie: Плагін для роботи з cookie в React;
15. react-dom: Допоміжна бібліотека для взаємодії React з DOM;
16. react-loader-spinner: Компонент React для відображення анімованих завантажувачів;
17. react-player: Компонент React для відтворення медіафайлів;
18. react-router та react-router-dom: Бібліотека для навігації в React-додатках;
19. uuid: Генератор UUID (унікальних ідентифікаторів);
20. yup: Бібліотека для валідації об'єктів та схем у JavaScript;
21. @types/js-cookie, @types/react, @types/react-dom, @types/uuid: TypeScript типи для відповідних бібліотек;
22. @vitejs/plugin-react: Плагін для Vite, який дозволяє використовувати React у проекті;
23. sass: Препроцесор для стилів, який дозволяє використовувати Sass у проекті;
24. vite: Швидкий та сучасний бандлер та сервер для розробки веб-застосунків.

4.9 Встановлення базових пакетів до сервера

Перейдемо до папки server та за допомогою команди “yarn init” створимо файл package.json та встановимо всі необхідні пакети для розробки серверної частини за допомогою команди:

“yarn add @apollo/server @graphql-tools/graphql-file-loader @graphql-tools/load @prisma/client bcryptjs -D @types/bcryptjs @types/cookie-parser @types/cors nodemon prisma ts-node cookie-parser cors dotenv express @types/express graphql @types/jsonwebtoken jsonwebtoken moment yup”

Розглянемо кожен з пакетів, вміст файлу package.json вказано на (рис. 4.13). Визначимо основні функції пакетів:

1. @apollo/server: Це пакет, який надає реалізацію сервера GraphQL за допомогою Apollo Server. Використовується для створення сервера GraphQL, обробки запитів та відправки відповідей клієнтам;
2. @graphql-tools/graphql-file-loader: Пакет для завантаження GraphQL схеми з файлів. Це допоміжний інструмент, який може використовуватися при побудові схеми GraphQL;
3. @graphql-tools/load: Цей пакет використовується для збору та завантаження схем GraphQL. Він допомагає ефективно організовувати та завантажувати GraphQL схеми;
4. @prisma/client: Клієнт Prisma генерує програмний інтерфейс для взаємодії з базою даних через Prisma ORM. Забезпечує типізований доступ до бази даних;
5. bcryptjs та @types/bcryptjs: Bcryptjs - це бібліотека для хешування паролів. @types/bcryptjs містить TypeScript типи для bcryptjs;
6. cookie-parser та @types/cookie-parser: cookie-parser - middleware для розбору cookie з HTTP-запиту.@types/cookie-parser` - TypeScript типи для cookie-parser;
7. cors та @types/cors: cors - middleware для обробки CORS (Cross-Origin Resource Sharing) у веб-додатках.@types/cors - TypeScript типи для cors;
8. dotenv: Завантажує значення з файлу .env у змінні середовища;
9. express та @types/express: express - веб-фреймворк для Node.js.@types/express` - TypeScript типи для express;
10. graphql: Основний пакет GraphQL для визначення типів та використання GraphQL мови запитів;
11. jsonwebtoken та @types/jsonwebtoken: jsonwebtoken- бібліотека для створення та перевірки JWT (JSON Web Tokens).@types/jsonwebtoken` - TypeScript типи для jsonwebtoken;

12. moment: Бібліотека для роботи з датами та часом у JavaScript;
13. yup: Бібліотека для валідації об'єктів та схем у JavaScript;
14. nodemon: Інструмент для автоматичного перезапуску Node.js додатків при змінах у файловій системі;
15. prisma: Командний інструмент для роботи з Prisma, такий як генерація коду, міграції та інше;
16. ts-node: Дозволяє виконувати файли TypeScript без передкомпіляції у JavaScript. Використовується для розробки Node.js додатків на TypeScript.

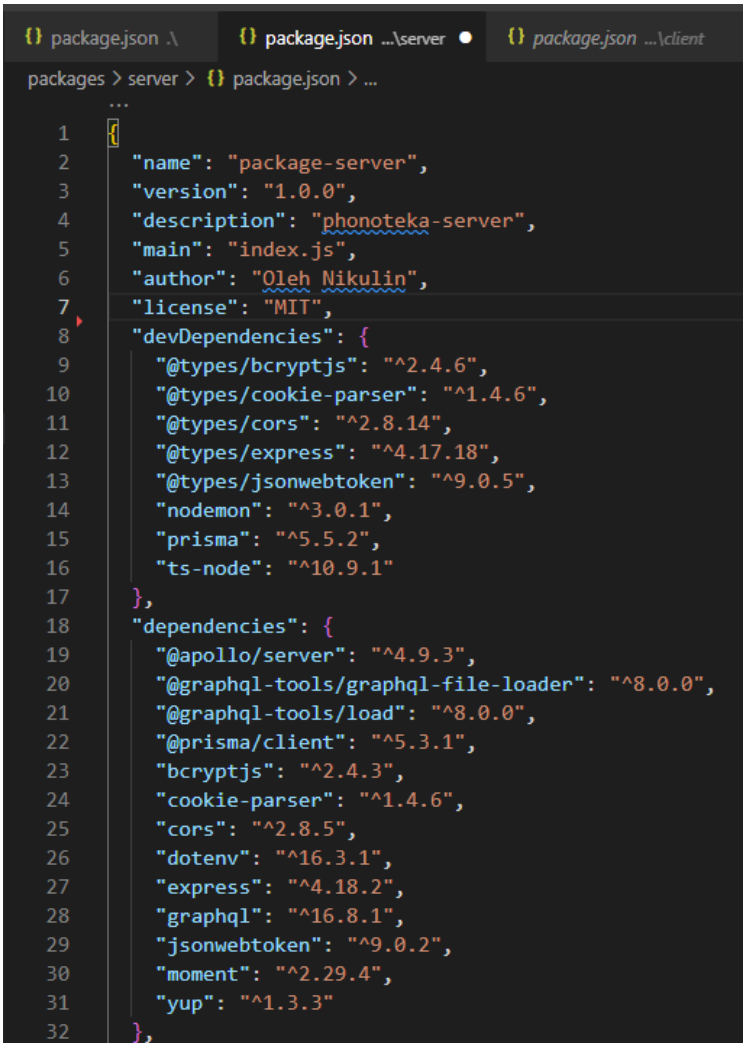


Рисунок 4.13 – Встановлені пакети на сервер

4.10 Встановлення та конфігурація ESlint та Prettier для форматування коду

Для почтаку налаштувань необіхдно зазначити за що відповідають ці пакети. ESLint та Prettier - це інструменти для роботи з кодом в процесі розробки програмного забезпечення, і вони здатні взаємодіяти для покращення якості та консистентності коду.

ESLint - це лінтер для JavaScript, що допомагає виявляти та виправляти проблеми в коді, пов'язані з його якістю, стилем та стандартами. Основні функції ESLint включають:

1. Виявлення помилок та неправильного використання: ESLint перевіряє ваш код на виявлення можливих помилок та неправильного використання мовних функцій;
2. Форматування коду: ESLint допомагає вам дотримуватися визначених стилів коду, форматуючи його відповідно до правил конфігурації.
3. Визначення та застосування стандартів коду: ESLint дозволяє використовувати або налаштовувати різні стандарти коду (наприклад, Airbnb, Google, тощо) та застосовувати їх до вашого проекту;
4. Підтримка для JSX та ES6+: ESLint адаптований для сучасних функцій JavaScript, таких як JSX та ES6+.

Prettier - це інструмент для автоматичного форматування коду, який допомагає забезпечити консистентність та читабельність в кодовій базі. Основні функції Prettier включають:

1. Автоматичне форматування коду: Prettier автоматично форматує код згідно встановлених правил, що допомагає уникнути ручного форматування;
2. Підтримка різних мов: Prettier підтримує форматування коду для різних мов програмування, включаючи JavaScript, TypeScript, HTML, CSS, JSON, і багато інших;
3. Легка інтеграція: Prettier легко інтегрується з іншими інструментами та редакторами коду, дозволяючи автоматично форматувати код під час збереження файлу або використовувати плагіни для інтеграції з IDE;
4. Конфігурація через файли проекту: Prettier дозволяє визначити правила форматування в проекті за допомогою файлів конфігурації, наприклад, .prettierrc або prettier.config.js.

Для налаштування поперше необхіжно видалити файли eslint та prettier з кліенту, які були ствоерні під час ініціалізації проекту. Перейдемо до папки

packages та встановимо наступні пакети за допомогою команди, також одразу встановимо пакет typescript, який знадобится трози пізніше під час його налаштування:
“yarn add --dev @typescript-eslint/eslint-plugin @typescript-eslint/parser eslint eslint-config-prettier eslint-plugin-prettier eslint-plugin-react eslint-plugin-react-hooks eslint-plugin-react-refresh prettier typescript”.

Розглянемо кожен пакет з списку та їхню роль:

1. @typescript-eslint/eslint-plugin: Цей плагін надає правила для ESLint, які розуміють TypeScript. Використовується для виявлення та виправлення проблем у коді, написаному з використанням TypeScript;
2. @typescript-eslint/parser: Це парсер TypeScript для ESLint. Використовується для аналізу TypeScript-коду та генерації абстрактного синтаксичного дерева (AST), яке ESLint може використовувати для аналізу коду;
3. eslint: Основний пакет ESLint, який використовується для аналізу та виявлення проблем в JavaScript та TypeScript-коді;
4. eslint-config-prettier: Конфігурація ESLint, яка вимикає всі правила, які можуть конфліктувати з Prettier. Забезпечує сумісність між ESLint та Prettier;
5. eslint-plugin-prettier: Плагін для ESLint, який дозволяє інтегрувати Prettier у ESLint, надаючи можливість використовувати Prettier як правило ESLint;
6. eslint-plugin-react: Плагін для ESLint, який містить правила для роботи з React-компонентами та JSX;
7. eslint-plugin-react-hooks: Даний плагін додає правила для ESLint, пов'язані з правильним використанням React Hooks;
8. eslint-plugin-react-refresh: Це плагін для підтримки "Fast Refresh" у React, який дозволяє гаряче перезавантаження компонентів;
9. prettier: Це інструмент для автоматичного форматування коду. Prettier гарантує однаковий стиль у всьому проекті, полегшуючи спільну роботу;
10. typescript: TypeScript - це мова програмування, яка розширює JavaScript, додаючи до нього статичну типізацію та інші функції для роботи з великими проектами.

Ці пакети разом допомагають створити потужне середовище розробки для проектів, які використовують TypeScript та React, забезпечуючи високий рівень якості та консистентності коду.

Створимо файл .eslintrc.js та напишемо наступний код, який приведено на (рис. 4.14).

```
// eslint-disable-next-line no-undef
module.exports = {
  root: true,
  parser: '@typescript-eslint/parser',
  plugins: ['@typescript-eslint'],
  extends: [
    'eslint:recommended',
    'plugin:@typescript-eslint/recommended',
    'plugin:react/recommended',
    'plugin:react-hooks/recommended',
    'plugin:prettier/recommended',
  ],
  rules: {
    'prettier/prettier': 'error',
  },
  overrides: [
    {
      files: ['packages/client/**/*.tsx', 'packages/client/**/*.ts'],
      env: {
        browser: true,
        es2021: true,
      },
      extends: [
        'eslint:recommended',
        'plugin:react/recommended',
        'plugin:react-hooks/recommended',
        'plugin:@typescript-eslint/recommended',
        'prettier',
      ],
      rules: {},
    },
    {
      files: ['packages/server/**/*.ts'],
      env: {
        node: true,
        es2021: true,
      },
      extends: ['eslint:recommended', 'plugin:@typescript-eslint/recommended', 'prettier'],
      rules: {},
    },
  ],
}
```

Рисунок 4.14 – Вміст файлу .eslintrc.js

Розглянемо ретельніше його частини:

1. root: true: Цей параметр вказує, що конфігурація ESLint повинна бути застосована до поточного каталогу та всіх його підкаталогів;
2. parser: '@typescript-eslint/parser': Вказує, що в якості парсера коду використовується @typescript-eslint/parser, який розуміє TypeScript;

3. plugins: ['@typescript-eslint']: Визначає, які плагіни ESLint слід використовувати, у цьому випадку — плагін для підтримки TypeScript;
4. extends: Вказує на збірку конфігурацій, які слід використовувати. Тут використовуються декілька стандартних конфігурацій, таких як 'eslint:recommended', а також конфігурації для TypeScript ('plugin:@typescript-eslint/recommended'), React ('plugin:react/recommended', 'plugin:react-hooks/recommended') та Prettier ('plugin:prettier/recommended');
5. rules: Визначає правила для проекту. Тут встановлено правило 'prettier/prettier': 'error', що означає виведення помилки, якщо Prettier виявляє розходження від конфігурації форматування;
6. overrides: Дозволяє визначати конфігурації, які специфічні для певних типів файлів чи каталогів.

Тепер створимо файл prettier.config.js та напишимо наступні правила, які зазначено на (рис. 4.15).

```
1  /* eslint-disable no-undef */
2  module.exports = {
3    printWidth: 100,
4    singleQuote: true,
5    tabWidth: 2,
6    trailingComma: 'all',
7    semi: false,
8    arrowParens: 'avoid',
9    endOfLine: 'auto',
10   jest: true,
11 }
```

Рисунок 4.15 – Вміст файлу prettier.config.js

Розглянемо кожне з цих налаштувань:

1. printWidth: 100: Задає максимальну ширину рядка коду. Якщо рядок перевищує цю ширину, Prettier спробує розбити його на кілька рядків;
2. singleQuote: true: Вказує, що використовуйте одинарні лапки для рядків, там де це можливо. Наприклад, ' замість ";
3. tabWidth: 2: Задає кількість пробілів для одного табуляційного символу;
4. trailingComma: 'all': Додає кому в кінці списку елементів об'єктів та масивів, включаючи останній елемент;

5. semi: false: Вимикає використання крапки з комою в кінці кожного оператора;
6. arrowParens: 'avoid': Вказує, щоб обхідні були круглі дужки для єдиного параметра стрілкових функцій;
7. endOfLine: 'auto': Автоматично визначає символ кінця рядка в залежності від операційної системи (LF на Unix-подібних системах, CRLF на Windows);
8. jest: true: Вказує Prettier форматувати файли конфігурації Jest (наприклад, jest.config.js).

Цей файл визначає основні налаштування Prettier для проекта, і Prettier використовуватиме ці параметри під час форматування коду відповідно до вказаних величин.

Останім кроком буде перехід до пакетів кліенту та серверу, та додаваня наступної строки "prettier": "../../prettier.config.js" в файл package.json. Це означає, що кожен пакет повинен силатись на конфігурацію прітіера, що знаходиться на дві дерикторії вище. Приклад приведено на (рис. 4.16).

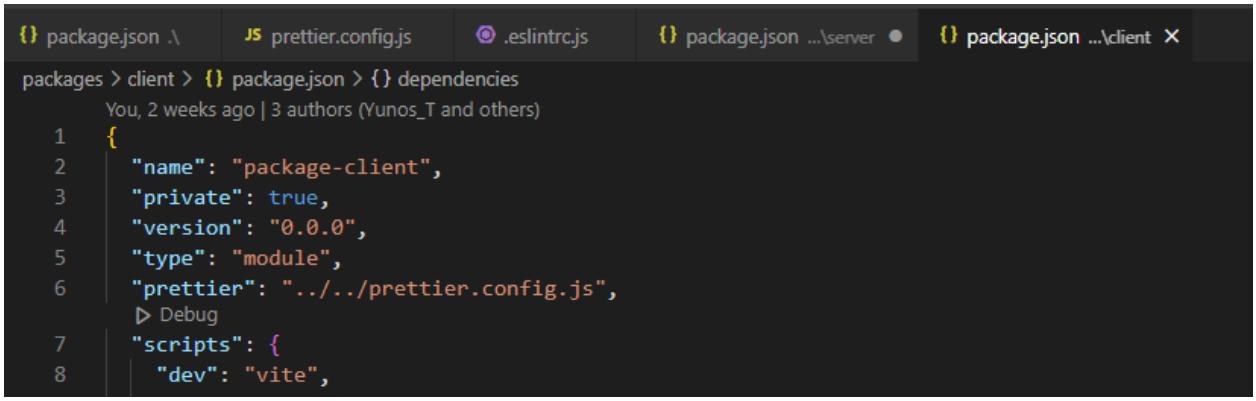


Рисунок 4.16 – Визначення конфігурації для prettier

Перевірити працездатність можливо, якщо порушити якесь з правил, що були описані у файлах, приклад наведено на (рис. 4.17).

Рисунок 4.17 – Відображення помилки після конфігурації eslint та prettier

Ще одним прикладом э, коли ми об’явимо зміну та не будемо її використовувати та додамо до неї в кінці крапку з комою. Редактор коду підскаже, що це є помилкою як на (рис. 4.18).

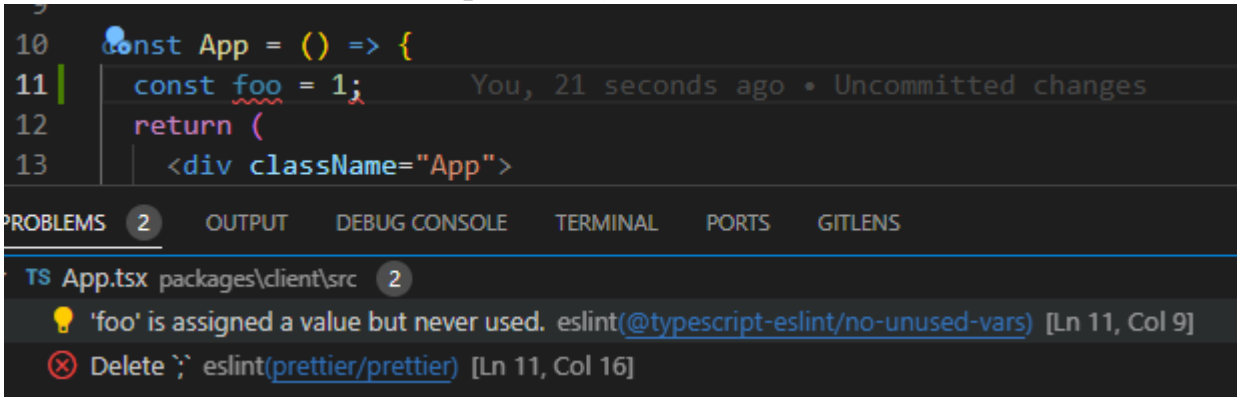


Рисунок 4.18 – Відображення помилки після конфігурації eslint та prettier

4.11 Конфігурація мови програмування TypeScript

Перейдемо до папки client та відректуємо вже інсуючі файли tsconfig.json та tsconfig.node.json, які були створені під час ініціалізації vite проекту, як це вказано на (рис. 4.19) для файлу tsconfig.json та на (рис. 4.20) для файлу tsconfig.node.json.

Рисунок 4.19 – Конфігурація файлу tsconfig.json

Рисунок 4.20 – Конфігурація файлу tsconfig.node.json

Ці два файли працюють разом для забезпечення налаштувань TypeScript для проекту, де tsconfig.node.json визначає особливості для використання в середовищі Node.js, що необхідно для коректної роботи Vite компілятора, а tsconfig.json — для загального використання.

Тепер перейдемо до папки server та створемо файл tsconfig.json з базовою конфігурацією. Файл для сервера більш простий, адже код компілюєця у середовищі node.js без допомоги інших бібліотек або компіляторів, як на клієнті. Приклад файлу наведено на (рис. 4.21).

Рисунок 4.21 – Конфігурація файлу tsconfig.json для серверу

Якщо файли були не правильно сконфігуровані, то редактор коду у відповідній вкладці – problems висвітить помилки пов’язані с конфігурацією, як це показано на (рис. 4.22).

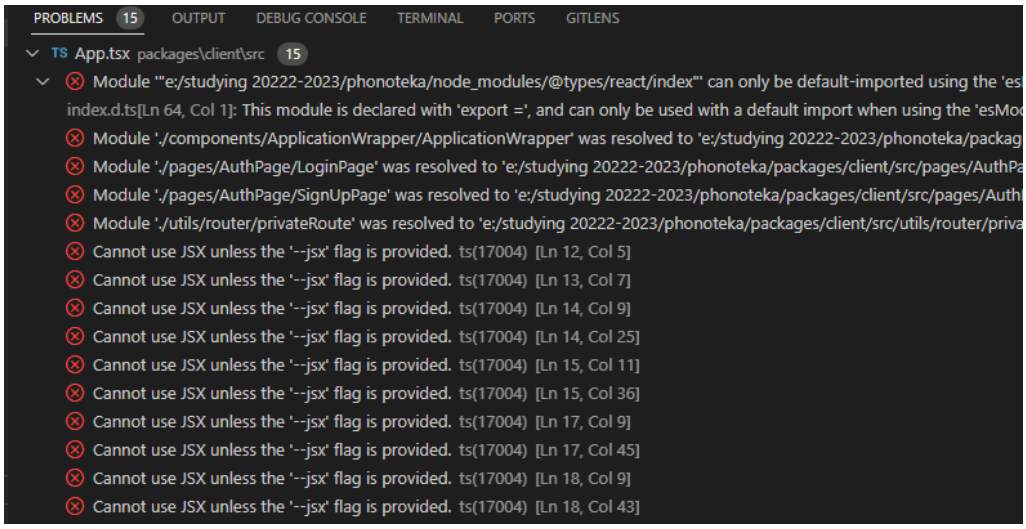


Рисунок 4.22 – Помилки при неправильній конфігурації TypeScript

Якщо все було зроблено правильно, то можна побачити повідомлення як на (рис. 4.23)

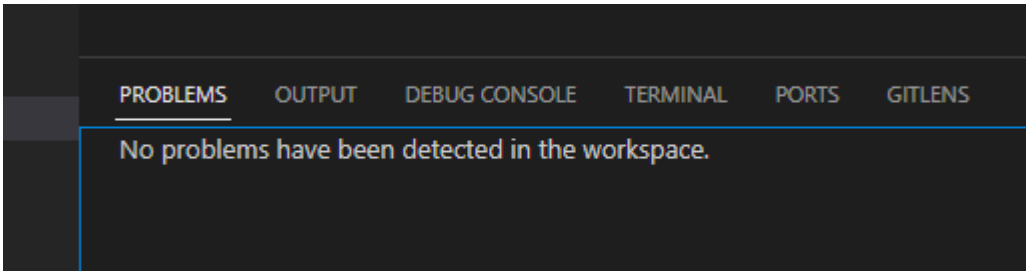


Рисунок 4.23 – Повідомлення про відсутність помилок конфігурації

4.12 Створення репозиторію на GitHub та перший пуш

Зайдемо на сайт https://github.com та створемо новий репозиторій з ім’я Phonoteka та залишимо всі налаштування за замовченням. Репозиторій зробимо публічним, щоб всі люди мали можливість проглядувати вихідній код, так як це показано на (рис. 4.24).

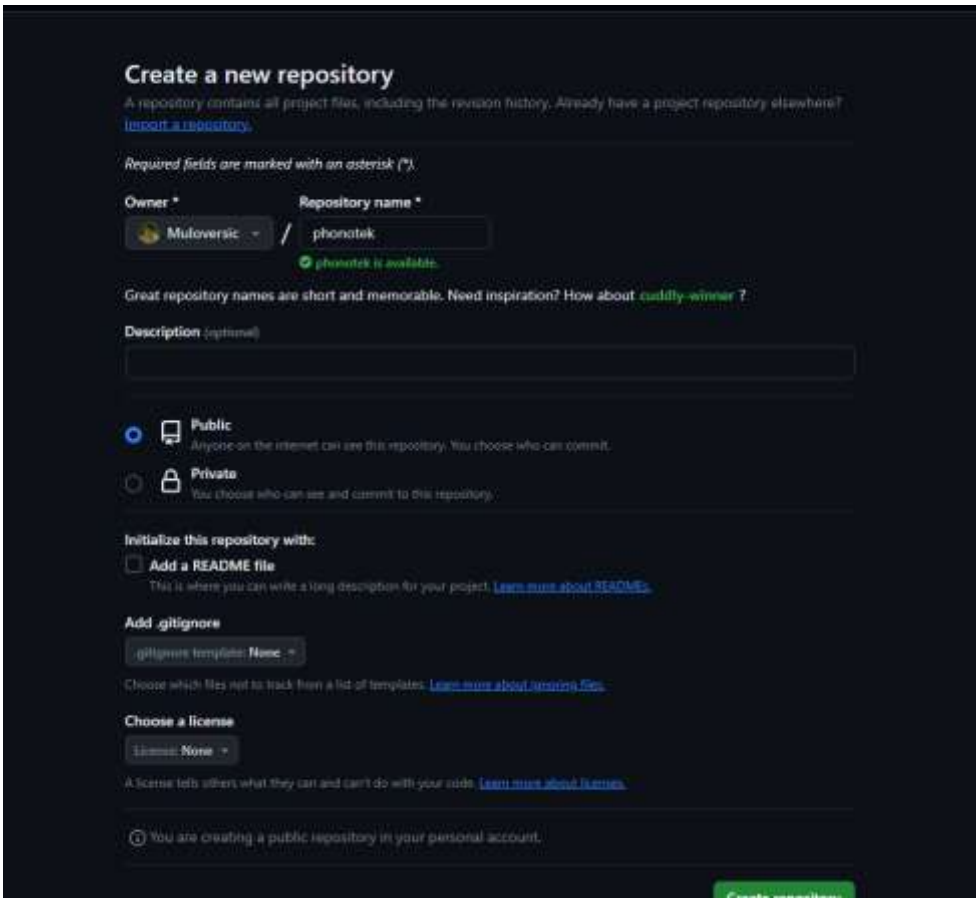


Рисунок 4.24 – Створення нового репозиторію на GitHub

Після цього перейдемо до Visual Studio Code та в терменалі виберемо Git bash, як це показано на (рис. 4.25).

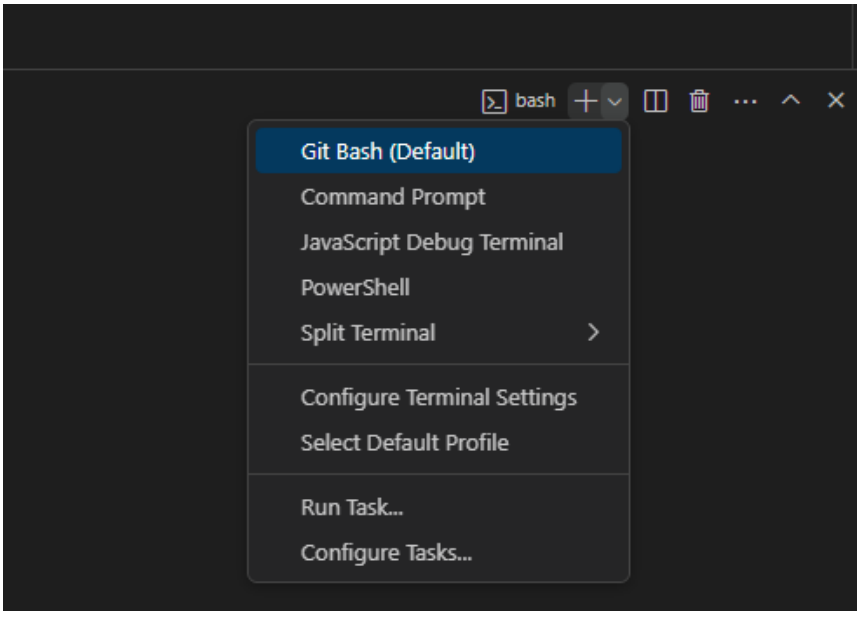


Рисунок 4.25 – Вибір Git Bash

GitHub після створення пропонує набір команд, які показані на (рис. 4.26), для додавання репозиторію, скористаємося ними, але замість master гілки зробимо dev гілку, для цього пропишимо команду git branch –b dev.

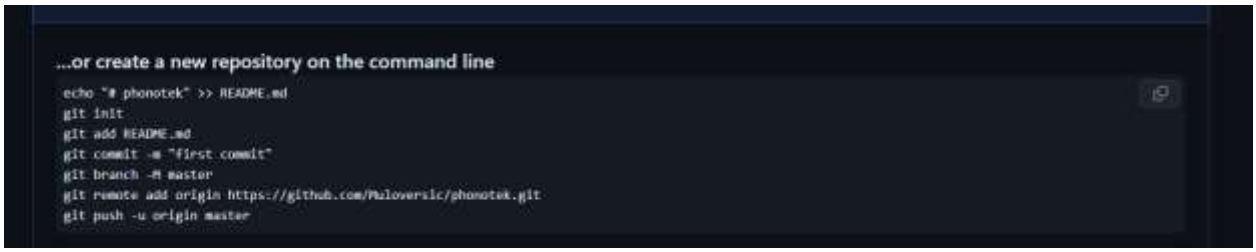


Рисунок 4.26 – Набір команд для ініціалізації репзиторію та додвання його на GitHub

Можна буде побачити репозиторій на сайті, як це показано на (рис. 4.27). Тепер для кожної окремої розробки частини проекту буде робитися окрема гілка та змержуватись с основною dev гілкою. На випадок, якщо данні на комп’ютері будуть втрачені вони залишаться на GitHub. Також є можливість відкатитсь до будь якого коміта, які будуть зроблені у процессі розробки проетку, на випадок, якщо нова версія не буде стабільною.

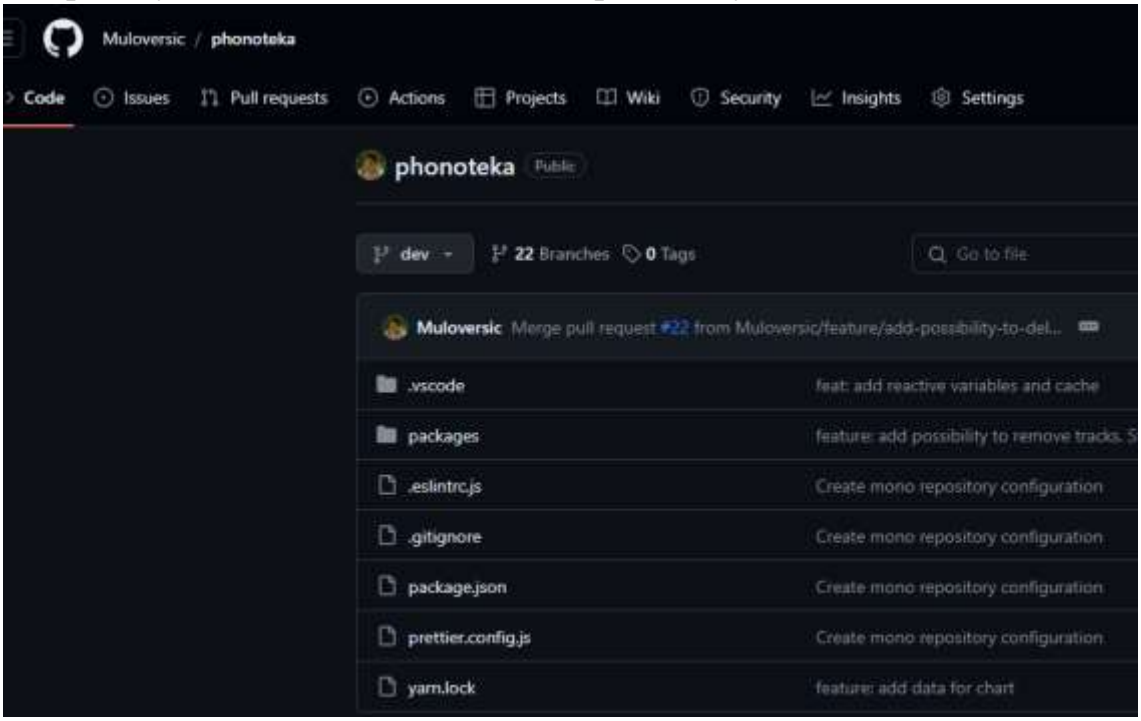


Рисунок 4.27 – Запушений репозиторій

4.13 Написання коду для запуску серверу

Імпорти, які зазначені на (рис. 4.28) використовуються для реалізації серверної частини веб-додатку на базі Express та Apollo Server для обробки GraphQL запитів. Розглянемо кожен імпорт окремо та пояснимо їхнє використання:

- express: Це фреймворк для створення веб-додатків на Node.js. Використовується для створення та налаштування сервера;
- { Application, Request, Response }: Визначає базові типи та об'єкти Express, такі як Application (екземпляр додатку Express), Request (об'єкт запиту) та Response (об'єкт відповіді). Ці типи використовуються для визначення параметрів функцій обробників маршрутів та middleware;
- JwtPayload: Це інтерфейс або тип даних, який представляє структуру токена JWT. Використовується для декларування структури токенів при їх розкодуванні;
- cors: Це пакет, який дозволяє налаштовувати міжсайтові запити (CORS) для сервера Express. Використовується для забезпечення безпечного обміну ресурсами між різними доменами;
- dotenv: Це пакет, який дозволяє завантажити значення змінних середовища з файлу .env в змінній process.env. Використовується для безпечного зберігання конфіденційних даних, таких як ключі API чи налаштування, у файлі конфігурації;
- join: Це функція Node.js для об'єднання шляхів файлів чи каталогів. Використовується для створення абсолютного шляху до файла схеми GraphQL;
- ApolloServer: Це клас, який надає можливість налаштовувати і запускати сервер GraphQL за допомогою Apollo Server;
- GraphQLRequestContext: Це інтерфейс, який представляє контекст запиту GraphQL. Використовується для доступа до інформації про запит в процесі обробки;
- BaseContext: Це інтерфейс, який визначає базовий контекст для запиту GraphQL. Використовується для передачі додаткової інформації та функціональності в різні частини системи;
- cookieParser: Це middleware Express, яке дозволяє обробляти та розкодовувати куки в запитах. Використовується для роботи з куками, зокрема для отримання токенів;
- jwt: Це бібліотека для роботи з токенами JWT (JSON Web Tokens). Використовується для створення та перевірки підписаних токенів, що забезпечують аутентифікацію та авторизацію;

- expressMiddleware: Це функція, яка дозволяє легко інтегрувати Apollo Server з існуючим сервером Express. Використовується для обробки GraphQL запитів через Express;
- GraphQLFileLoader: Це завантажувач для GraphQL-схеми, який дозволяє завантажувати схему з файлів. Використовується для зручного роботи з файловою структурою GraphQL;
- loadSchemaSync: Це функція для синхронного завантаження GraphQL-схеми з використанням завантажувачів;
- resolvers: Це файл, який містить резольвери GraphQL для обробки різних запитів та мутацій;
- prisma: Це інстанція Prisma Client, яка використовується для взаємодії з базою даних;
- generateToken: Це функція для створення токенів доступу та оновлення.
- sendRefreshToken: Це функція для встановлення токену оновлення в HTTP куках.

Ці імпорти об'єднуються, щоб створити сервер GraphQL, який має всі необхідні функції для обробки запитів, взаємодії з базою даних та забезпечення безпеки та аутентифікації.

```
1  import express, { Application, Request, Response } from 'express'
2  import { JwtPayload } from 'jsonwebtoken'
3  import cors from 'cors'
4  import dotenv from 'dotenv'
5  import { join } from 'node:path'
6  import { ApolloServer, GraphQLRequestContext, BaseContext } from '@apollo/server'
7  import cookieParser from 'cookie-parser'
8  import jwt from 'jsonwebtoken'
9  import { expressMiddleware } from '@apollo/server/express4'
10 import { GraphQLFileLoader } from '@graphql-tools/graphql-file-loader'
11 import { loadSchemaSync } from '@graphql-tools/load'
12 import resolvers from './graphql/resolvers'
13 import prisma from './prisma/index'
14 import generateToken from './utils/generateToken'
15 import sendRefreshToken from './utils/sendRefreshToken'
```

Рисунок 4.28 – Імпорти до основного фалу сервера

Наступний код, що показано на (рис. 4.29) має за мету налаштування та конфігурацію серверної частини GraphQL за допомогою Apollo Server. Давайте розглянемо кожну частину коду та пояснимо її функції:

```
export interface CustomJwtPayload extends JwtPayload {
  userId: number       You, last month • fix: application refactor
}
You, last month | 1 author (You)
export interface MyContext extends BaseContext {
  req: Request & { payload?: CustomJwtPayload }
  res: Response
}

const typeDefs = loadSchemaSync(join(__dirname, 'graphql/schema.graphql'), {
  loaders: [new GraphQLFileLoader()],
})

dotenv.config()
const port = process.env.PORT || 4040

const loggerPlugin = {
  // Fires whenever a GraphQL request is received from a client.
  async requestDidStart(requestContext: GraphQLRequestContext<BaseContext>) {
    console.log('Request started! Query:\n' + requestContext.request.query)

    return {
      // Fires whenever Apollo Server will parse a GraphQL
      // request to create its associated document AST.
      async parsingDidStart() {
        console.log('Parsing started!')
      },

      // Fires whenever Apollo Server will validate a
      // request's document AST against your GraphQL schema.
      async validationDidStart() {
        console.log('Validation started!')
      },
    }
  },
}

const responsePlugin = {
  async requestDidStart() {
    return {
      // eslint-disable-next-line @typescript-eslint/no-explicit-any
      async willSendResponse(requestContext: { response: any }) {
        const { response } = requestContext
      },
```

Рисунок 4.29 – Налаштування та конфігурація серверної частини GraphQL за допомогою Apollo Server

- CustomJwtPayload: Це інтерфейс, який розширює стандартний JwtPayload з пакету jsonwebtoken. Визначає структуру JWT-токена з

додатковим полем userId.prisma: Це інстанція Prisma Client, яка використовується для взаємодії з базою даних;

- MyContext: Це інтерфейс, який розширює BaseContext з Apollo Server. Визначає контекст для кожного GraphQL запиту, де req (об'єкт запиту Express) має додаткове поле payload, яке може містити інформацію про користувача;
- loadSchemaSync: Це функція з бібліотеки @graphql-tools/load, яка синхронно завантажує схему GraphQL з файлу schema.graphql. Використовується для визначення схеми GraphQL на сервері;
- loggerPlugin: Це об'єкт, який містить метод requestDidStart, що логує інформацію про GraphQL-запити, їхнє парсінг та валідацію. Використовується для виведення інформації про запити у консоль;
- responsePlugin: Це об'єкт, який містить метод requestDidStart, що викликається перед відправленням відповіді GraphQL на клієнта. Використовується для виконання додаткової логіки перед відправленням відповіді.

Цей код встановлює базові конфігурації для Apollo Server, такі як схема GraphQL, обробник логування та обробник відповідей.

Наступна функція setupServer, що показана на (рис. 4.30) та (рис. 4.31) встановлює та запускає сервер застосунку на основі бібліотеки Express та Apollo Server. Основні кроки, які вона виконує, включають:

1. Створення екземпляру Express застосунку;
2. Налаштування CORS (Cross-Origin Resource Sharing) для дозволу запитів від іншого домену. Запити дозволяються від 'http://localhost:5173';
3. Додавання middleware для обробки данних, які приходять у формі application/x-www-form-urlencoded;
4. Додавання middleware для обробки кук (cookies) за допомогою cookie-parser;
5. Створення екземпляру Apollo Server з використанням заданих схем, резольверів та плагінів;
6. Запуск Apollo Server за допомогою apolloServer.start();
7. Додавання маршруту для обробки GET-запиту до кореневого шляху ('/') та відправлення відповіді 'Hello';

8. Додавання маршруту для обробки POST-запиту до '/refresh-token', який використовує токен, що передається в куках, для оновлення та відправлення нового токену;
9. Додавання middleware для обробки JSON-даних та обробки запитів до '/graphql' через Apollo Server;
10. Запуск сервера застосунку на певному порті.

Цей код визначає основні частини сервера GraphQL на Express, включаючи обробку запитів, автентифікацію через токени та інші аспекти, необхідні для правильної роботи GraphQL сервера.

```
64  const setupServer = async () => {
65    const app: Application = express()
66    app.use(
67      cors<cors.CorsRequest>({
68        origin: 'http://localhost:5173',
69        credentials: true,
70      }),
71    )
72    app.use(express.urlencoded({ extended: true }))
73    app.use(cookieParser())
74
75    const apolloServer = new ApolloServer<MyContext>({
76      typeDefs,
77      resolvers,
78      plugins: [loggerPlugin, responsePlugin],
79    })
80
81    await apolloServer.start()
82
83    app.get('/', (req: Request, res: Response) => {
84      res.send('Hello')
85    })
86
87    app.post('/refresh-token', async (req: Request, res: Response) => {
88      const token = req.cookies.urt
89
90      if (!token) {
91        return res.send({ ok: false, token: '' })
92      }
93
94      // eslint-disable-next-line @typescript-eslint/no-explicit-any
95      let payload: any = null
96      try {
97        payload = jwt.verify(token, process.env.SECRET_REFRESH!)
98      } catch (error) {
99        console.log('error', error)
100       return res.send({ ok: false, token: '', message: 'Jwt verify error' })
101     }
102
103     const user = await prisma.users.findUnique({ where: { id: payload.userId } })
104     if (!user) {
105       return res.send({ ok: false, token: '', message: 'User not found' })
106     }
107
```

Рисунок 4.30 – Функція запуску серверу

```
107
108     const { token: accessToken, refreshToken } = generateToken(user.id)
109     sendRefreshToken(res, refreshToken)
110
111     return res.send({ ok: true, token: accessToken })
112   })
113
114   app.use(
115     '/graphql',
116     express.json(),
117     expressMiddleware(apolloServer, {
118       context: async ({ req, res }) => {
119         return { req, res }
120       },
121     }),
122   )
123
124   app.listen(port, () => {
125     console.log(`Server listens at http://localhost:${port}`)
126     console.log(`Graphql server available at http://localhost:${port}/graphql`)
127   })
128 }
129
130 setupServer()
```

Рисунок 4.31 – Функція запуску серверу

4.14 Написання коду для ORM Prisma

Напустпним кроком визначемо необхідні моедлі до бази даних та створемо підключення до неї. Зробимо файл schema.prisma та опишимо моделі, які зазначено на (рис. 4.32) та (рис. 4.33). Код представляє схему бази даних для GraphQL-сервера, який використовує Prisma ORM для зв'язку з PostgreSQL базою даних. Давайте розглянемо кожен елемент цієї схеми:

- Generator prisma-client-js;
- Datasource db;
- Model Users;
- Model Tokens;
- Model Bands;
- Model Tracks.

Кожена model визначає таблицю в базі даних, а також типи та зв'язки для GraphQL-схеми. Ця схема дозволяє здійснювати операції CRUD (створення, читання, оновлення, видалення) над користувачами, токенами, гуртами та треками у базі даних.

Також зробимо файл .env куди запишемо зімнні оточення:

- PORT – порт на якому запущен сервер;
- DATABASE_URL – шлях по якому сервер буде підключатись до бази даних;
- SECRET – таємниє слово, яке використовується для шифрування точену;
- SECRET_REFRESH – таємниє слово, яке використовується для шифрування рефреш точену.

```
generator client {
  provider = "prisma-client-js"
}

You, 2 months ago | 1 author (You)
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

You, 2 weeks ago | 1 author (You)
model Users {
  id        Int      @id @default(autoincrement())
  email     String   @unique
  name      String
  password  String
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  imgUrl    String?
  Tokens    Tokens[]
  Bands     Bands[]
  Tracks    Tracks[]
}

You, 2 months ago | 1 author (You)
model Tokens {
  id           Int      @id @default(autoincrement())
  user         Users    @relation(fields: [userId], references: [id])
  userId       Int      @unique
  token        String   @unique
  refreshToken String   @unique
  createdAt    DateTime @default(now())
  expiresAt    DateTime
  updatedAt    DateTime @updatedAt

  @@index([userId], name: "idx_userId")
}
```

Рисунок 4.33 – Код ініціалізації таблиць PostgreSQL

```
You, 2 weeks ago | 1 author (You)
model Bands {       You, 2 months ago • feature: create auth logic
  id              Int      @id @default(autoincrement())
  name            String
  createdAt       DateTime @default(now())
  user            Users    @relation(fields: [userId], references: [id])
  userId          Int
  foundationDate DateTime
  genre           String
  members         String
  description     String?
  about           String?
  location        String?
  image           String?
  tracks          Tracks[]

  @@index([userId], name: "idx_userId_band")
}

You, 2 weeks ago | 1 author (You)
model Tracks {
  id         Int      @id @default(autoincrement())
  user       Users    @relation(fields: [userId], references: [id])
  userId     Int
  name       String
  year       DateTime
  album      String
  createdAt DateTime @default(now())
  genre      String
  url        String?
  format     String
  band       Bands    @relation(fields: [bandId], references: [id])
  bandId     Int

  @@index([bandId], name: "idx_bandId")
  @@index([userId], name: "idx_userId_track")
}
```

Рисунок 4.33 – Код ініціалізації таблиць PostgreSQL

Після написання коду для таблиць занесемо їх до бази даних за допомогою команди npx prisma migrate dev. Ця команда підключится до локальної бази даних PostgreSQL та внесе необхідні зміни. Запустивши програму PGAdmin можна перевірити зміни, та побачити, що все пройшло успішно побачивши таблиці, що відображені на (рис. 4.34). Також PGAdmin автоматично генерує діаграму відношення ERD(Entity-Relationship Diagram) таблць одна до одної, що відображено на (рис. 4.35).

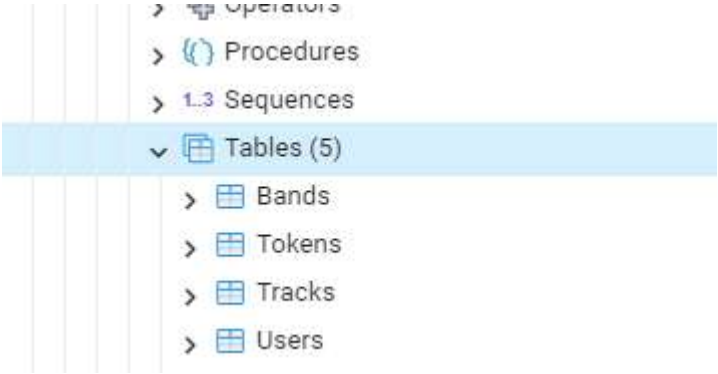


Рисунок 4.34 – Створені таблиці

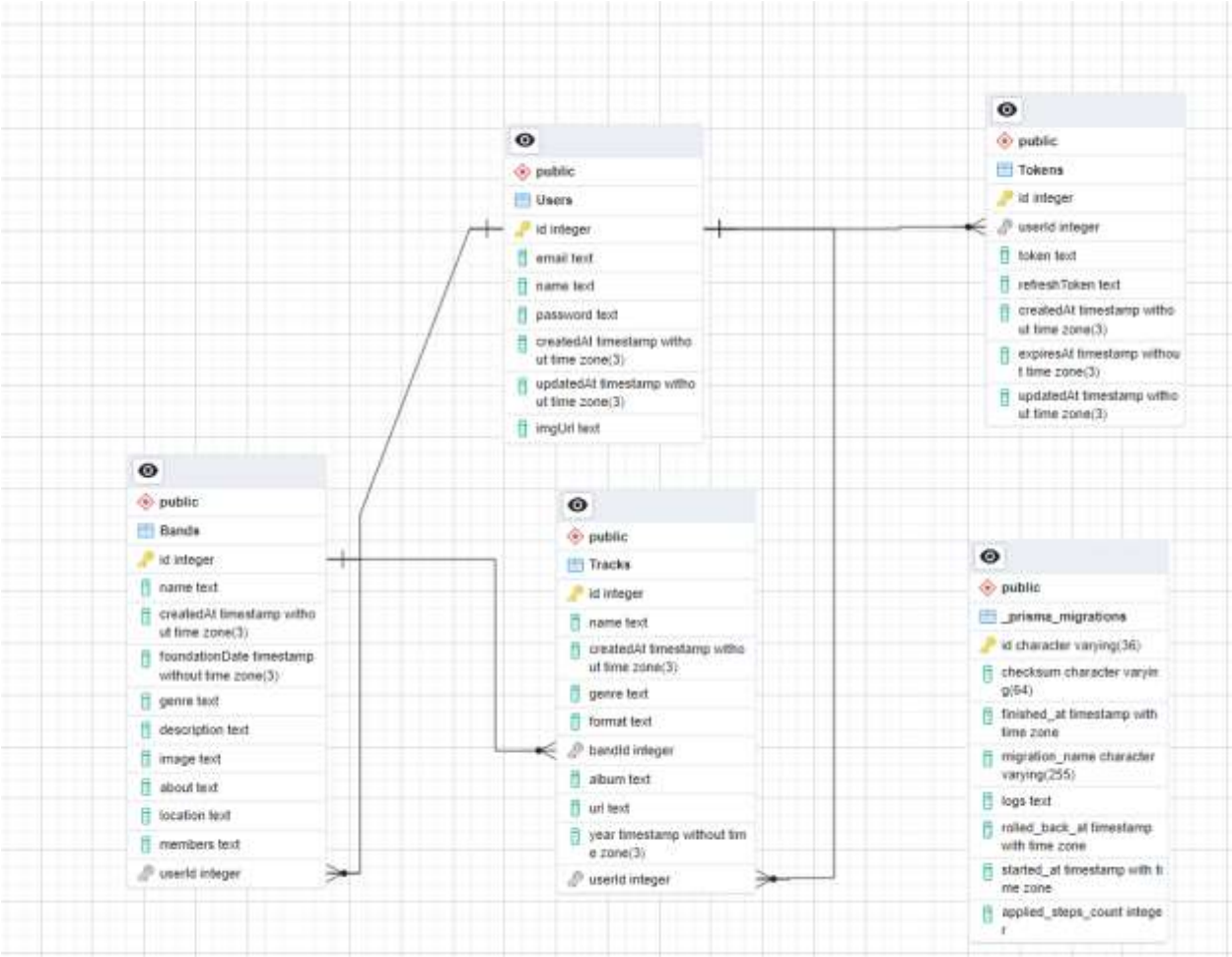


Рисунок 4.35 – ERD створених таблиць

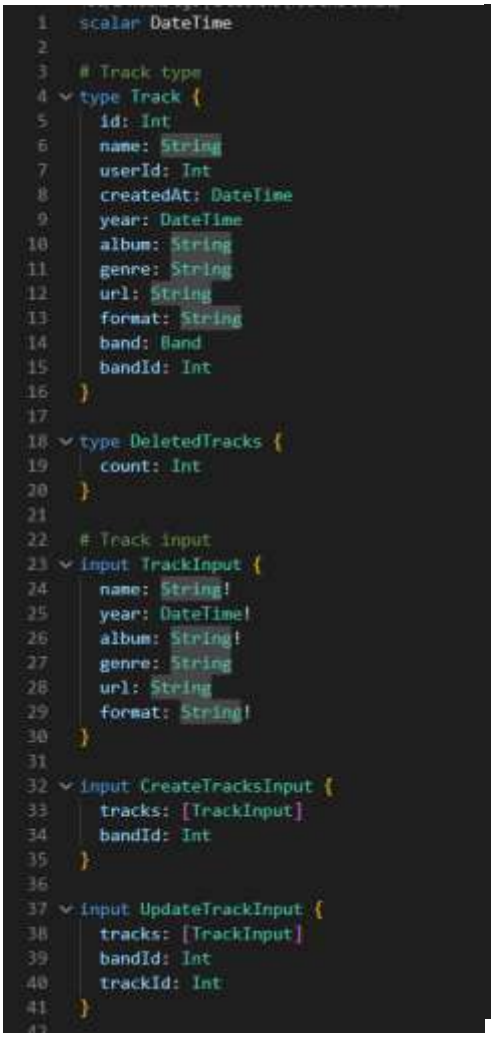
4.15 Написання коду для GraphQL схеми.

Наступник кроком буде створення GraphQL схеми та написання резолверів для запитів.

В папці graphql створемо файл schema.graphql в якому напишемо основні типи для даних, які будуть прийматись сервером, данні які будуть відправлятись та типи для квері і мутацій. Цей код, що приведено на (рис. 4.36), (рис. 4.37), (рис. 4.38) та (рис. 4.39) описує GraphQL-схему, яка визначає типи, запити (queries), та мутації (mutations), що доступні для взаємодії з сервером. Давайте розглянемо основні елементи цієї схеми:

- Скаляр DateTime;
- Тип Track;
- Тип DeletedTracks;
- Тип TrackInput;
- Тип CreateTracksInput;
- Тип UpdateTrackInput;
- Тип UpdateTracksInBandInput;
- Тип Band;
- Тип BandInput;
- Тип UpdateBandInput;
- Тип UserInput;
- Тип User;
- Тип AuthResponse;
- Тип TracksQuery;
- Тип ChartGenreData;
- Тип ChartDataResponse;
- Запити (queries);
- Мутації (mutations).

Ця схема дозволяє клієнтам виконувати різні запити та мутації для роботи з даними користувачів, гуртів, треків, а також отримувати дані для побудови діаграм.



```
scalar DateTime

# Track type
type Track {
  id: Int
  name: String
  userId: Int
  createdAt: DateTime
  year: DateTime
  album: String
  genre: String
  url: String
  format: String
  band: Band
  bandId: Int
}

type DeletedTracks {
  count: Int
}

# Track input
input TrackInput {
  name: String!
  year: DateTime!
  album: String!
  genre: String
  url: String
  format: String!
}

input CreateTracksInput {
  tracks: [TrackInput]
  bandId: Int
}

input UpdateTrackInput {
  tracks: [TrackInput]
  bandId: Int
  trackId: Int
}
```

Рисунок 4.36 – Опис схеми GraphQL, частина 1

```
input UpdateTracksInBandInput {
  id: Int
  bandId: Int
  userId: Int
  name: String!
  year: DateTime!
  createdAt: DateTime
  album: String!
  genre: String
  url: String
  format: String!
}

# Band type
type Band {
  id: Int
  name: String
  userId: Int
  createdAt: DateTime
  foundationDate: DateTime
  genre: String
  members: String
  description: String
  about: String
  location: String
  image: String
  tracks: [Track]
}

# Band input
input BandInput {
  name: String!
  foundationDate: DateTime!
  description: String
  members: String!
  about: String
  location: String
  tracks: [TrackInput!]
  image: String
}
```

Рисунок 4.37 – Опис схеми GraphQL, частина 2

```
input UpdateBandInput {
  id: Int
  name: String!
  foundationDate: DateTime!
  description: String
  members: String!
  about: String
  location: String
  tracks: [UpdateTracksInBandInput!]
  image: String
}

# Band input
input UserInput {
  name: String!
  imgUrl: String
  email: String!
}

#User ---------------------------
type User {
  id: Int!
  email: String!
  password: String!
  name: String
  imgUrl: String
  createdAt: DateTime!
  updatedAt: DateTime!
}

type AuthResponse {
  token: String!
  refreshToken: String!
}

type TracksQuery {
  tracks: [Track]
  allTracksCount: Int
}

type ChartGenreData {
  value: Int
  label: String
  id: Int
}
```

Рисунок 4.38 – Опис схеми GraphQL, частина 3

```
type ChartDataResponse {
  tracksAmount: Int
  genreData: [ChartGenreData]
}

# End of User definitions ---------------------------

type Query {
  getBandById(id: Int): Band
  getUserById: User
  getAllBands: [Band]
  getChartData: ChartDataResponse
  getAllTracks(
    order: String
    sortBy: String
    search: String
    pageSize: Int
    pageNumber: Int
  ): TracksQuery
}

type Mutation {
  createBand(input: BandInput!): Band
  updateBand(input: UpdateBandInput!): Band
  updateUser(input: UserInput!): User
  createTracks(input: CreateTracksInput!): [Track]
  deleteTracks(ids: [Int]!): DeletedTracks
  updateTrack(input: UpdateTrackInput!): [Track]
  login(email: String!, password: String!): AuthResponse
  register(email: String!, username: String!, password: String!, rePassword: String!): AuthResponse
}
```

Рисунок 4.39 – Опис схеми GraphQL, частина 4

Далі зробимо файли у відповідній структурі, як зазначено на (рис. 4.40)

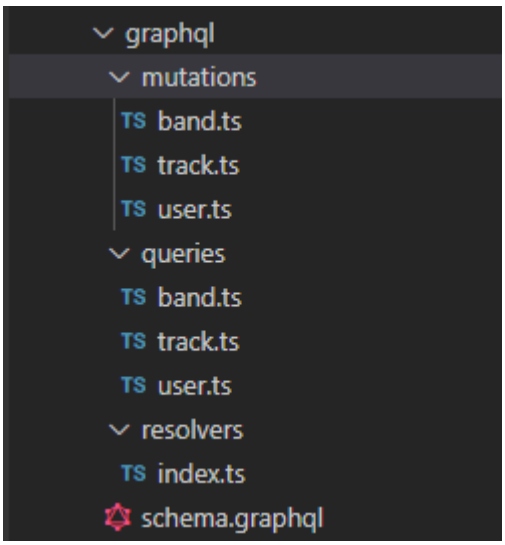


Рисунок 4.40 – Структура файлів у папці graphql

До файлу в папці resolvers будемо імпортувати функції кверів та мутації, напишемо їх.

4.16 Написання коду для GraphQL кверів

Почнемо з кверів для користувача. Код для запитів щодо користувача приведено на (рис. 4.41). Код представляє набір GraphQL-запитів (queries), які можна використовувати для отримання даних про користувачів та для отримання даних, які використовуються для побудови діаграм. Давайте розглянемо основні елементи коду:

- Мутація getUserById;
- Мутація getChartData;
- Заборона використання any;

```
You, 2 weeks ago | 1 author (You)
/* eslint-disable @typescript-eslint/no-explicit-any */
import { MyContext } from 'index'
import authenticate from '../../middlewares/authenticate'
import prisma from '../../prisma/index'

const userQueries = () => {
  return {
    getUserById: (_: any, variables: null, { req }: MyContext) => {
      authenticate(req)
      return prisma.users.findUnique({
        where: {
          id: req.payload?.userId,
        },
      })
    },
    getChartData: async (_: any, variables: null, { req }: MyContext) => {
      authenticate(req)
      const tracksAmount = await prisma.tracks.count()

      const trackCountsByGenre = await prisma.tracks.groupBy({
        by: ['genre'],
        _count: {
          id: true,
        },
      })

      const genreData = trackCountsByGenre.map((group, i) => ({
        id: i,
        label: group.genre,
        value: group._count.id,
      }))

      return {
        tracksAmount,
        genreData,
      }
    },
  }
}

export default userQueries
```

Рисунок 4.41 – Код для кверів користувача

Наступним будуть квері для запитів щодо гуртів. Цей код, що представлено на (рис. 4.42) містить набір GraphQL-запитів (queries), які можна використовувати для отримання даних про музичні гурти (bands). Давайте розглянемо основні елементи коду:

- Мутація getBandById;
- Мутація getAllBands;
- Заборона використання any.

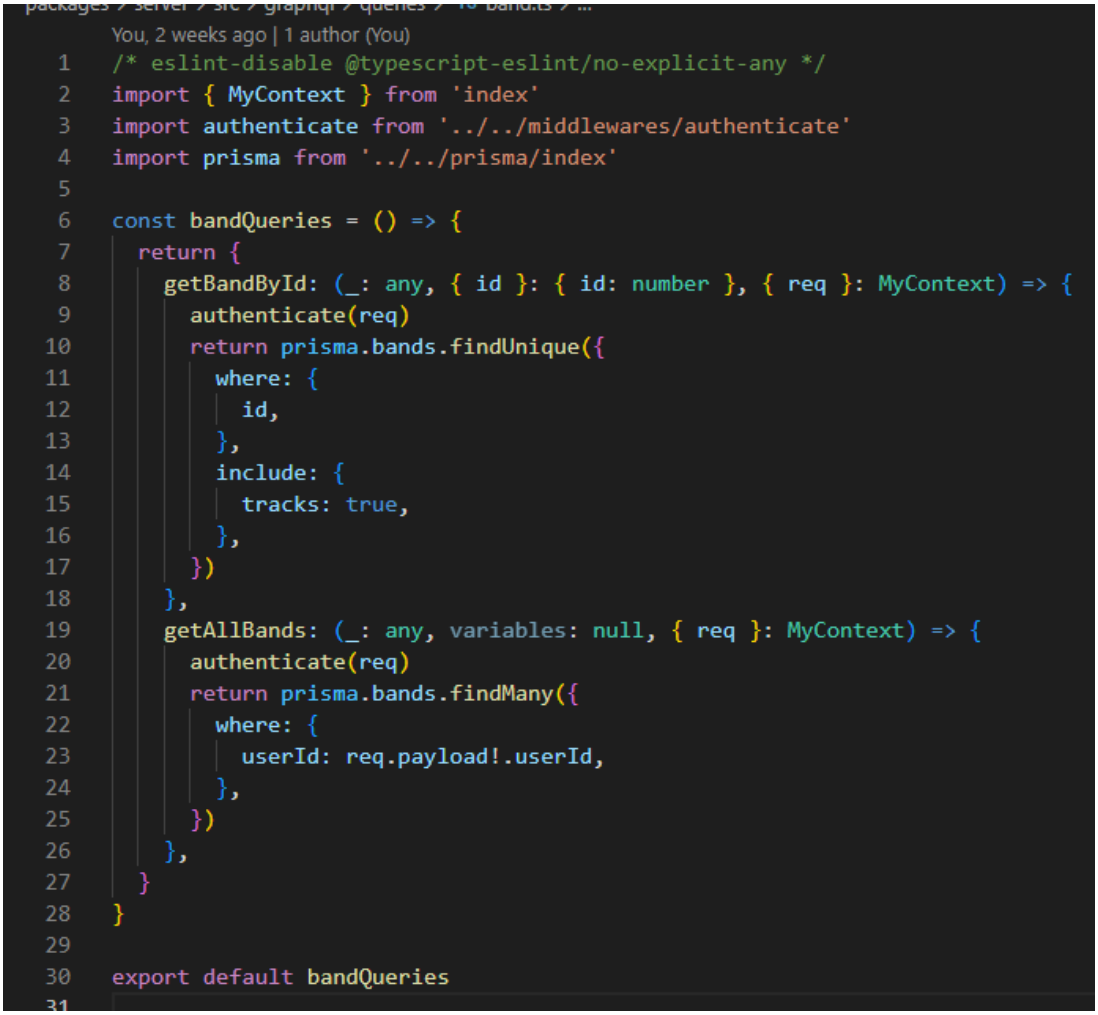
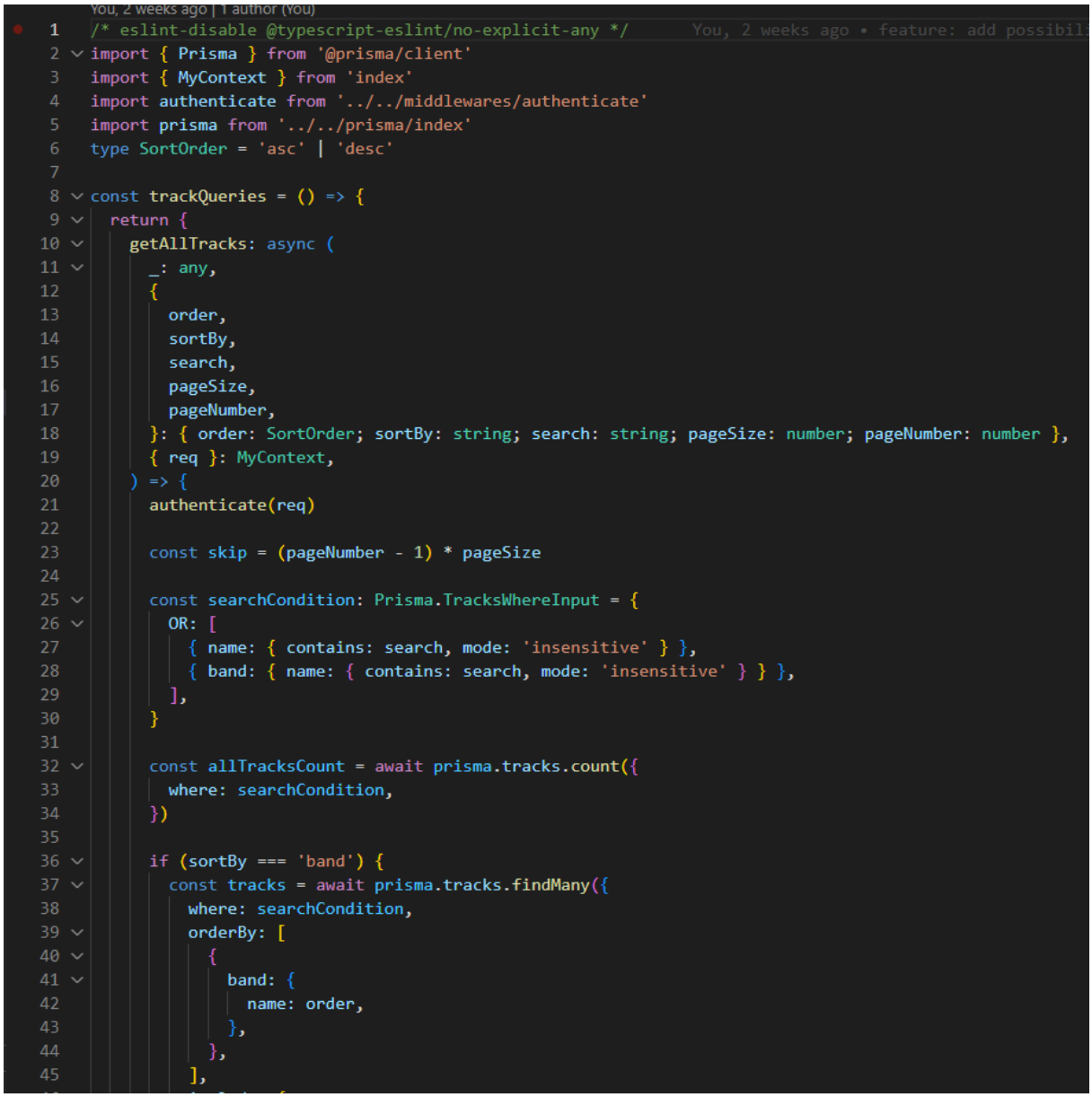


Рисунок 4.42 – Код для кверів гуртів

Наступним будуть квері для запитів щодо треків. Цей код, що показано на (рис. 4.43) містить приклад GraphQL-запитів (queries), які використовуються для отримання даних про треки (tracks). Давайте розглянемо основні елементи коду:

- Мутація getAllTracks;
- Опції сортування та фільтрації;
- Формування умови пошуку (searchCondition);
- Використання prisma для отримання даних;
- Повернення результатів;
- Заборона використання any.



```
You, 2 weeks ago | 1 author (You)
/* eslint-disable @typescript-eslint/no-explicit-any */      You, 2 weeks ago • feature: add possibil
import { Prisma } from '@prisma/client'
import { MyContext } from 'index'
import authenticate from '../../middlewares/authenticate'
import prisma from '../../prisma/index'
type SortOrder = 'asc' | 'desc'

const trackQueries = () => {
  return {
    getAllTracks: async (
      _: any,
      {
        order,
        sortBy,
        search,
        pageSize,
        pageNumber,
      }: { order: SortOrder; sortBy: string; search: string; pageSize: number; pageNumber: number },
      { req }: MyContext,
    ) => {
      authenticate(req)

      const skip = (pageNumber - 1) * pageSize

      const searchCondition: Prisma.TracksWhereInput = {
        OR: [
          { name: { contains: search, mode: 'insensitive' } },
          { band: { name: { contains: search, mode: 'insensitive' } } },
        ],
      }

      const allTracksCount = await prisma.tracks.count({
        where: searchCondition,
      })

      if (sortBy === 'band') {
        const tracks = await prisma.tracks.findMany({
          where: searchCondition,
          orderBy: [
            {
              band: {
                name: order,
              },
            },
          ],
```

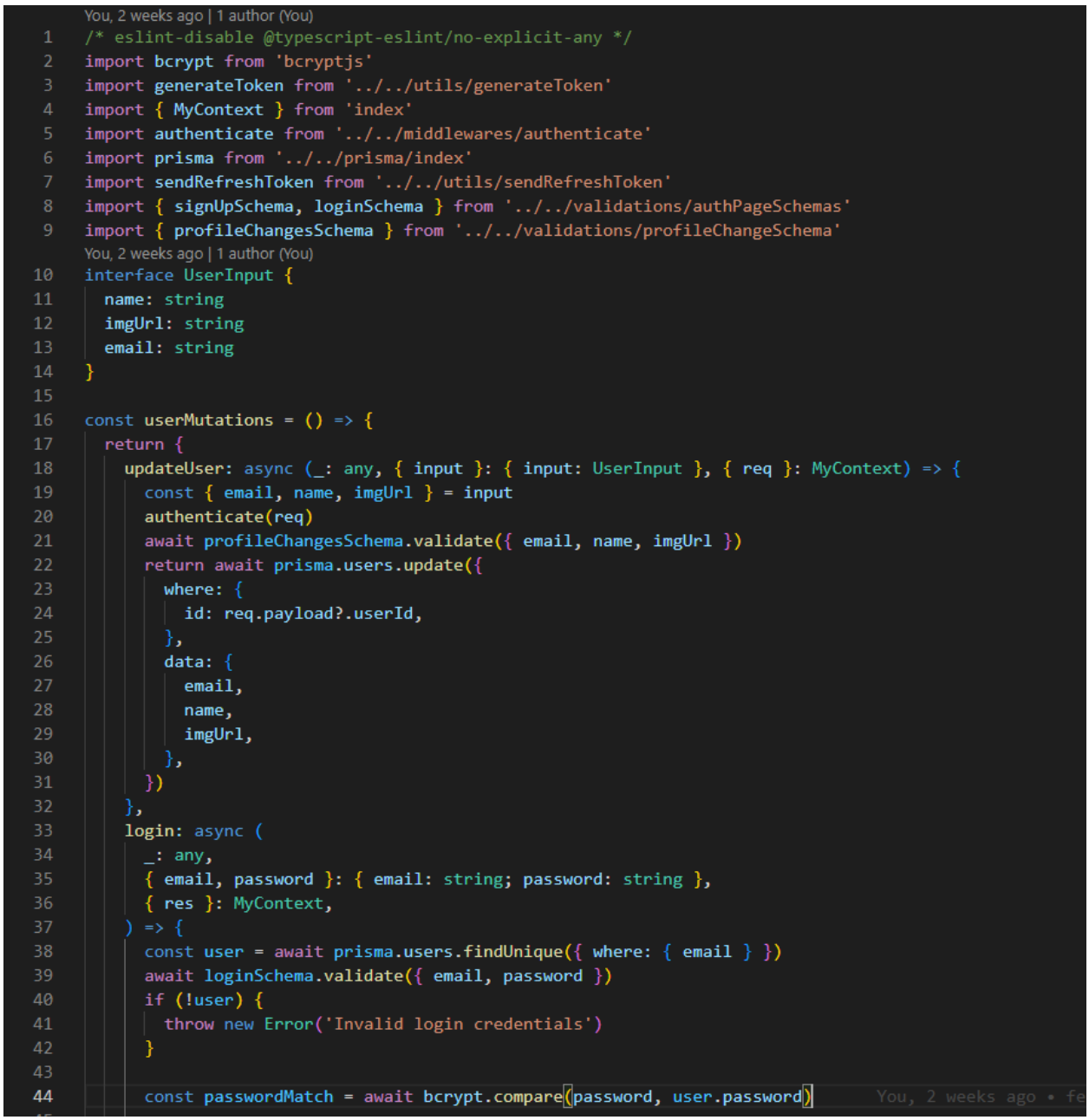
Рисунок 4.43 – Приклад коду для кверів треків

4.17 Написання коду для GraphQL мутацій

Почнемо написання також з мутацій для користувача. Цей код, що показано на (рис. 4.44) містить набір GraphQL-мутацій (mutations), які відповідають за зміни в даних користувачів. Розглянемо основні елементи коду:

- Оголошення залежностей та інтерфейсу;
- Оголошення інтерфейсу UserInput;
- Функція userMutations;

- Мутації;
- Заборона використання any.



```
You, 2 weeks ago | 1 author (You)
/* eslint-disable @typescript-eslint/no-explicit-any */
import bcrypt from 'bcryptjs'
import generateToken from '../../utils/generateToken'
import { MyContext } from 'index'
import authenticate from '../../middlewares/authenticate'
import prisma from '../../prisma/index'
import sendRefreshToken from '../../utils/sendRefreshToken'
import { signUpSchema, loginSchema } from '../../validations/authPageSchemas'
import { profileChangesSchema } from '../../validations/profileChangeSchema'
You, 2 weeks ago | 1 author (You)
interface UserInput {
  name: string
  imgUrl: string
  email: string
}

const userMutations = () => {
  return {
    updateUser: async (_: any, { input }: { input: UserInput }, { req }: MyContext) => {
      const { email, name, imgUrl } = input
      authenticate(req)
      await profileChangesSchema.validate({ email, name, imgUrl })
      return await prisma.users.update({
        where: {
          id: req.payload?.userId,
        },
        data: {
          email,
          name,
          imgUrl,
        },
      })
    },
    login: async (
      _: any,
      { email, password }: { email: string; password: string },
      { res }: MyContext,
    ) => {
      const user = await prisma.users.findUnique({ where: { email } })
      await loginSchema.validate({ email, password })
      if (!user) {
        throw new Error('Invalid login credentials')
      }

      const passwordMatch = await bcrypt.compare(password, user.password)
```

Рисунок 4.44 –Приклад коду для мутацій користувача

Наступним напишемо мутації для гуртів. Код, що представлено на (рис. 4.45) реалізує GraphQL мутації для оновлення та створення інформації про гурти та їх треки з використанням Prisma, Apollo Server та Express. Нижче наведено ретельний опис окремих елементів коду:

- Типи та Імпорти;

- Мутації для Оновлення та Створення Гуртів (Bands);
- Автентифікація та Валідація;
- Оновлення та Створення Треків;
- Перевірка Існування Гурту;
- Повернення Результатів;
- Унікальність Назв Гуртів.

Цей код взаємодіє з базою даних за допомогою Prisma та надає GraphQL API для роботи з інформацією про гурти та їх треки через відповідні мутації.

Рисунок 4.45 – Код для мутацій гуртів

Останнім кроком є написання мутацій для треків. Код, що представлено на (рис. 4.46) реалізує GraphQL мутації для взаємодії з треками (Tracks) гуртів використовуючи Prisma, Apollo Server та Express. Нижче наведено ретельний опис окремих елементів коду:

- Типи та Імпорти;
- Мутації для Роботи з Треками;
- Автентифікація та Валідація;
- Перевірка Існування Гурту;
- Створення та Оновлення Треків;
- Повернення Результатів.

Цей код дозволяє клієнту GraphQL взаємодіяти з треками гуртів, включаючи їх створення, оновлення та видалення.

Рисунок 4.46 – Приклад коду для мутацій треків

4.18 Запуск серверу

У файлі package.json, що знаходится в сервері додамо команду на запуск серверу: start, як це показано на (рис. 4.47). Таким чином створений файл nodemoon.json, який містить конфігурацію, що показано на (рис. 4.48) для запуску серверу почне роботу та в консолі можно буде побачити логи, що свідчять про запуск серверу та що все працює, це можна побачити на (рис. 4.49).

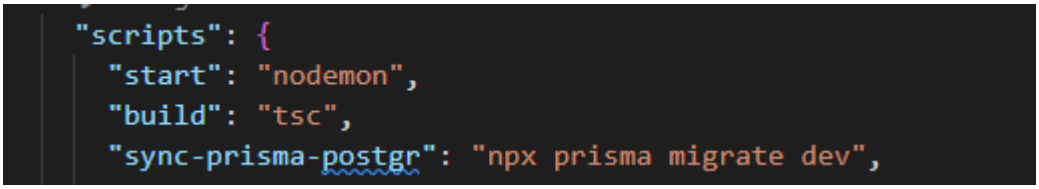


Рисунок 4.47 – Скріпт запуску серверу

Рисунок 4.48 – Конфігурація файлу nodemoon.json

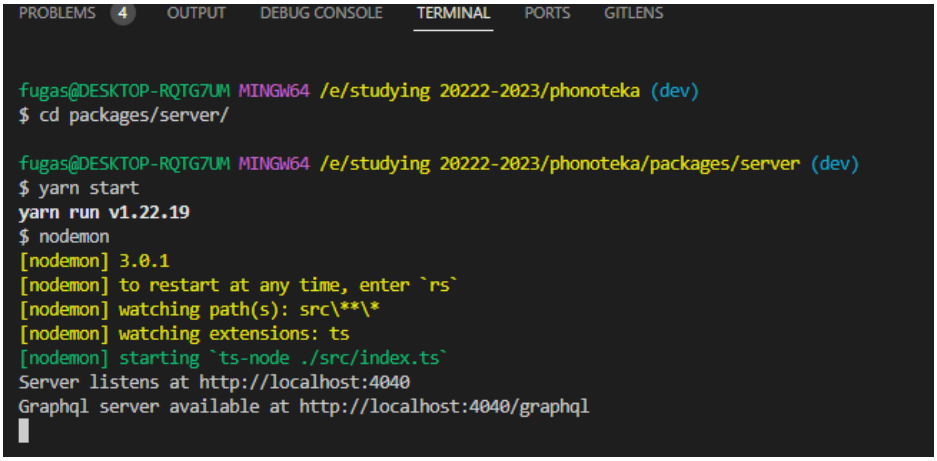


Рисунок 4.49 – Процес запуску серверу, та його логи.

4.19 Конфігурація бібліотеки Apollo Client

Написання клієнтської частини почнемо з конфігурації Apollo, то імпорту його до основного файлу, в якому Apollo огорне весь додаток, та його функції будуть доступні повсюди у коді. Для цього у папці src створемо папку apollo в якій файл client.ts має наступний код конфігурації, як показано на (рис. 4.50) та (рис. 4.51).

```
import { ApolloClient, InMemoryCache, HttpLink, from } from '@apollo/client'
import { getAccessToken, setAccessToken } from '../utils/accessToken'
import { onError } from '@apollo/client/link/error'
import { TokenRefreshLink } from 'apollo-link-token-refresh'
import { jwtDecode } from 'jwt-decode'

const cache = new InMemoryCache()

const httpLink = new HttpLink({
  uri: 'http://localhost:4040/graphql',
  credentials: 'include',
})

const authLink = new TokenRefreshLink({
  accessTokenField: 'token',
  isTokenValidOrUndefined: () => {
    const token = getAccessToken()

    if (!token) {
      console.log('Token is undefined')
      return Promise.resolve(true)
    }

    try {
      const { exp } = jwtDecode(token)

      console.log('Current timestamp:', Date.now())
      console.log('Token expiration timestamp:', exp! * 1000)
      You, last month • fix: application refactor
      if (Date.now() >= exp! * 1000) {
        console.log('Token has expired')
        return Promise.resolve(false)
      } else {
        console.log('Token is still valid')
        return Promise.resolve(true)
      }
    } catch (e) {
      console.log('Error decoding token:', e)
      return Promise.resolve(false)
    }
  },
  fetchAccessToken: async () => {
    const response = await fetch('http://localhost:4040/refresh-token', {
      method: 'POST',
      credentials: 'include',
```

Рисунок 4.50 – Конфігурація Apollo, частина 1

```
    } catch (e) {
      console.log('Error decoding token:', e)
      return Promise.resolve(false)
    }
  },
  fetchAccessToken: async () => {
    const response = await fetch('http://localhost:4040/refresh-token', {
      method: 'POST',
      credentials: 'include',       You, last month • fix: application refactor
    })

    return response
  },
  handleFetch: accessToken => {
    setAccessToken(accessToken)
  },
  handleError: err => {
    console.warn('Your refresh token is invalid. Try to relogin')
    console.error(err)
  },
})

const errorLink = onError(({ graphQLErrors, networkError }) => {
  console.log(graphQLErrors)
  console.log(networkError)
})

export const client = new ApolloClient({
  cache,
  link: from([authLink, errorLink, httpLink]),
})

```

Рисунок 4.51 – Конфігурація Apollo, частина 2

Цей код налаштовує клієнт Apollo для взаємодії з GraphQL сервером, забезпечуючи автоматичне оновлення токенів доступу для аутентифікації. Розглянемо його крок за кроком:

- Імпортовані Apollo Client, InMemoryCache, HttpLink для встановлення основних компонентів клієнта;
- Імпортовані функції для отримання та встановлення токенів доступу (getAccessToken і setAccessToken відповідно);
- Імпортовані onError з Apollo Client для обробки помилок та TokenRefreshLink для автоматичного оновлення токенів доступу;
- Створений об'єкт cache для InMemoryCache, який зберігає кеш запитів і відповідей для ефективного управління даними;

- Створений об'єкт httpLink для HttpLink, який вказує адресу GraphQL сервера та налаштовує опцію credentials на 'include' для передачі куки або HTTP-авторизації;
- Створений об'єкт authLink для TokenRefreshLink, який відповідає за автоматичне оновлення токенів доступу;
- isTokenValidOrUndefined: Функція, яка перевіряє час дії токену та повертає true, якщо токен дійсний або не визначений, та false, якщо токен протермінований чи невірний;
- fetchAccessToken: Функція, яка викликає точку REST API для оновлення токену доступу. Викликається, якщо токен дійсний, але вже протермінований;
- handleFetch: Функція, яка встановлює новий токен доступу після оновлення;
- handleError: Функція, яка виводить повідомлення про невірний оновлювальний токен та виводить деталі помилки у випадку виникнення помилок під час оновлення;
- errorLink: Створений об'єкт для обробки помилок в GraphQL запитах. Логи помилок виводяться в консоль;
- client: Створений об'єкт ApolloClient, який об'єднує всі компоненти (authLink, errorLink, httpLink) та використовує їх для взаємодії з GraphQL сервером.

Усі ці кроки забезпечують належну автентифікацію з використанням токенів доступу, а також автоматичне їх оновлення при необхідності.

До основного файлу main.tsx зробимо імопрт налаштованної конфігурації, та обгорнемо додаток у нього, як це показано на (рис. 4.52).

```tsx
import ReactDOM from 'react-dom/client'
import React from 'react'
import { BrowserRouter } from 'react-router-dom'
import { ApolloProvider } from '@apollo/client'

import { client } from './apollo/client'

import App from './App'

import './index.scss'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <ApolloProvider client={client}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </ApolloProvider>,
)
```

Рисунок 4.52 – Обгоратння всього застосунку у ApolloProvider

4.20 Написання сторінки для регістрації

Цей код, що представлено на (рис. 4.53) представляє приклад компонент React для сторінки реєстрації (Sign Up). Давайте детально розглянемо його структуру та функціональність:

1. Імпорт бібліотек та компонентів:
2. Оголошення змінних та хуків стану:
3. Обробка подій та функції:
4. Відображення інтерфейсу:
5. Відповідно до умов.

Цей компонент реалізує користувацький інтерфейс для сторінки реєстрації та використовує Apollo Client для взаємодії з GraphQL сервером.

Рисунок 4.53 – Приклад логіки для сторінки регістрації

Фінальний вигляд можна побачити на (рис. 4.54) та стан с помилками при введені даних на (рис. 4.55).

Рисунок 4.54 – Фінальний вигляд сторінки регістрації

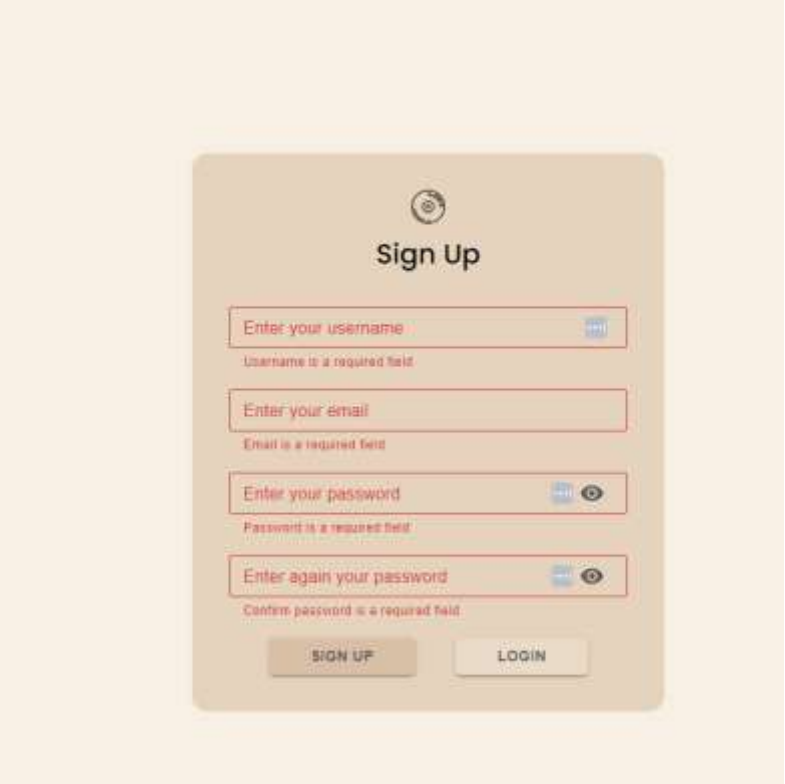


Рисунок 4.55 – Фінальний вигляд сторінки регістрації з помилками

4.21 Написання сторінки для логіну.

Фінальний вігляд сторінки показано на (рис. 4.56). Також варіант з валідацією показан на (рис. 4.57).

Рисунок 4.56 – Фінальний вигляд сторінки для логіну

Рисунок 4.57 – Фінальний вигляд сторінки для логіну з помилками

Цей код на (рис. 4.58) представляє приклад компонент React для сторінки входу в систему (логіну). Давайте детально розглянемо його структуру та функціональність:

- Імпортується низка бібліотек та компонентів, таких як React, useCallback, useEffect, useState, useNavigate для навігації, а також компоненти з бібліотек Material-UI;

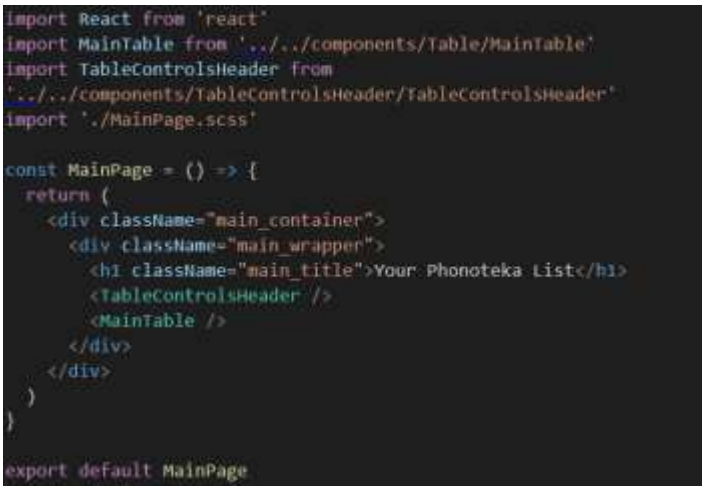
- Імпортується схема валідації (loginSchema) для перевірки правильності введених даних;
- navigate: Хук useNavigate для навігації між сторінками;
- LoginMutation: Функція-мутація Apollo Client для виклику мутації логіну;
- useEffect: Хук useEffect для відстеження змін у стані та автоматичного редіректу на головну сторінку при успішному вході;
- Форма для введення email та пароля;
- Відображення/приховування пароля за допомогою іконки "eye";
- Обробка завантаження (loading) з використанням компонента LoaderOval;
- Виведення помилок сервера або помилок валідації від formic;
- Використання куки для збереження токену після успішного входу;
- Перевірка наявності помилок та виведення відповідного повідомлення.

Цей компонент реалізує користувацький інтерфейс для входу в систему та використовує Apollo Client для взаємодії з GraphQL сервером.

Рисунок 4.58 – Логіка для сторінки логіну

4.22 Написання сторінки для основної таблиці сайту

Компонент основної сторінки містить дві вкладені компоненти – це компонента з кнопками для керування таблицею та сама таблиця. Код приведено на (рис. 4.59).



```
import React from 'react'
import MainTable from '../../components/Table/MainTable'
import TableControlsHeader from
'../../components/TableControlsHeader/TableControlsHeader'
import './MainPage.scss'

const MainPage = () => {
  return (
    <div className="main_container">
      <div className="main_wrapper">
        <h1 className="main_title">Your Phonoteka List</h1>
        <TableControlsHeader />
        <MainTable />
      </div>
    </div>
  )
}

export default MainPage
```

Рисунок 4.59 – Код для основної сторінки застосунку

Розглянемо компоненту TableControlsHeader, код якої приведено на (рис. 4.60). В цій компоненти імпортятся декілька стилізованих кнопокой за допомогою MUI, код стилів наведено на (рис. 4.61). Ці кнопки використовуються у всьмоу застосунку. Також імпортится модальне вікно CreateBandModal та CreateTrackModal. Ці дві компонети містять у собі форми для створення групи та треків окремо. Вложені форми також перевикористовуються у компоненті з редагуванням інформації про гурпу або трек.

```
import React, { useState } from 'react'
import { ActionButton } from
'../UI/MuiUI/Buttons.styled/ActionButton.styled'
import CreateBandModal from '../Modals/CreateBandModal/CreateBandModal'
import CreateTrackModal from
'../Modals/CreateTrackModal/CreateTrackModal'

const TableControlsHeader = () => {
  const [openCreateBand, setOpenCreateBand] = useState<boolean>(false)
  const [openCreateTrack, setOpenCreateTrack] = useState<boolean>(false)
  return (
    <div className="table-controls-header">
      <ActionButton onClick={() => setOpenCreateBand(true)}>Create
band</ActionButton>
      <ActionButton onClick={() => setOpenCreateTrack(true)}>Create
track</ActionButton>
      <CreateBandModal handleCloseModal={setOpenCreateBand}
openModal={openCreateBand} />
      {openCreateTrack && (
        <CreateTrackModal handleCloseModal={setOpenCreateTrack}
openModal={openCreateTrack} />
      )}
    </div>
  )
}

export default TableControlsHeader
```

Рисунок 4.60 – Код для компоненти TableControlsHeader

```
import styled from '@emotion/styled'
import { Button } from '@mui/material'
import {
  MAIN_BLUE_COLOR,
  MAIN_DARK_CREAM_COLOR,
  MAIN_GRAY_COLOR,
} from '../../../../variables/variables'

export const ActionButton = styled(Button)({
  '&.MuiButtonBase-root': {
    alignSelf: 'start',
    marginTop: 20,
    padding: '5px 20px',
    borderRadius: '10px',
    color: MAIN_GRAY_COLOR,
    backgroundColor: MAIN_DARK_CREAM_COLOR,
    fontWeight: 600,
    whiteSpace: 'nowrap',
    transition: '0.3s linear',
    '&:hover': {
      filter: 'contrast(105%) drop-shadow(0 0 5px #dac0a3)',
      color: MAIN_BLUE_COLOR,
    },
  },
}) as typeof Button
```

Рисунок 4.61 – Код для стилізованної MUI кнопки

Розглянемо компоненту CreateBandModal, приклад коду якої приведено на (рис. 4.62). Цей код визначає компонент React для модального вікна створення гурту (CreateBandModal). Давайте розглянемо його структуру та функціональність:

- Імпортується низка бібліотек та компонентів, таких як React, useState, useEffect, а також компоненти з Material-UI;
- Імпортується Formik для роботи з формами, функції ref, uploadBytes, та getDownloadURL для роботи з Firebase Storage;
- Імпортується useMutation з Apollo Client для виклику мутації GraphQL;
- Імпортується схема валідації bandValidationSchema та інші компоненти;
- IFormValues: Інтерфейс для форми створення гурту;
- useEffect: Викликається при монтажі компонента для встановлення shouldRefetchTracks(false);
- handleFormSubmit: Обробник подання форми, який викликається при спробі створення гурту. Завантажує зображення в Firebase Storage, отримує його URL та викликає мутацію для створення гурту;
- Використовує компонент DetailModal з Material-UI для створення модального вікна;
- Використовує Formik для створення форми зі змінними values, errors та touched;
- Передає необхідні обробники та дані компоненту BandForm.

Цей компонент реалізує модальне вікно для створення нового гурту, з використанням форми, валідації та взаємодії з GraphQL сервером та Firebase Storage.

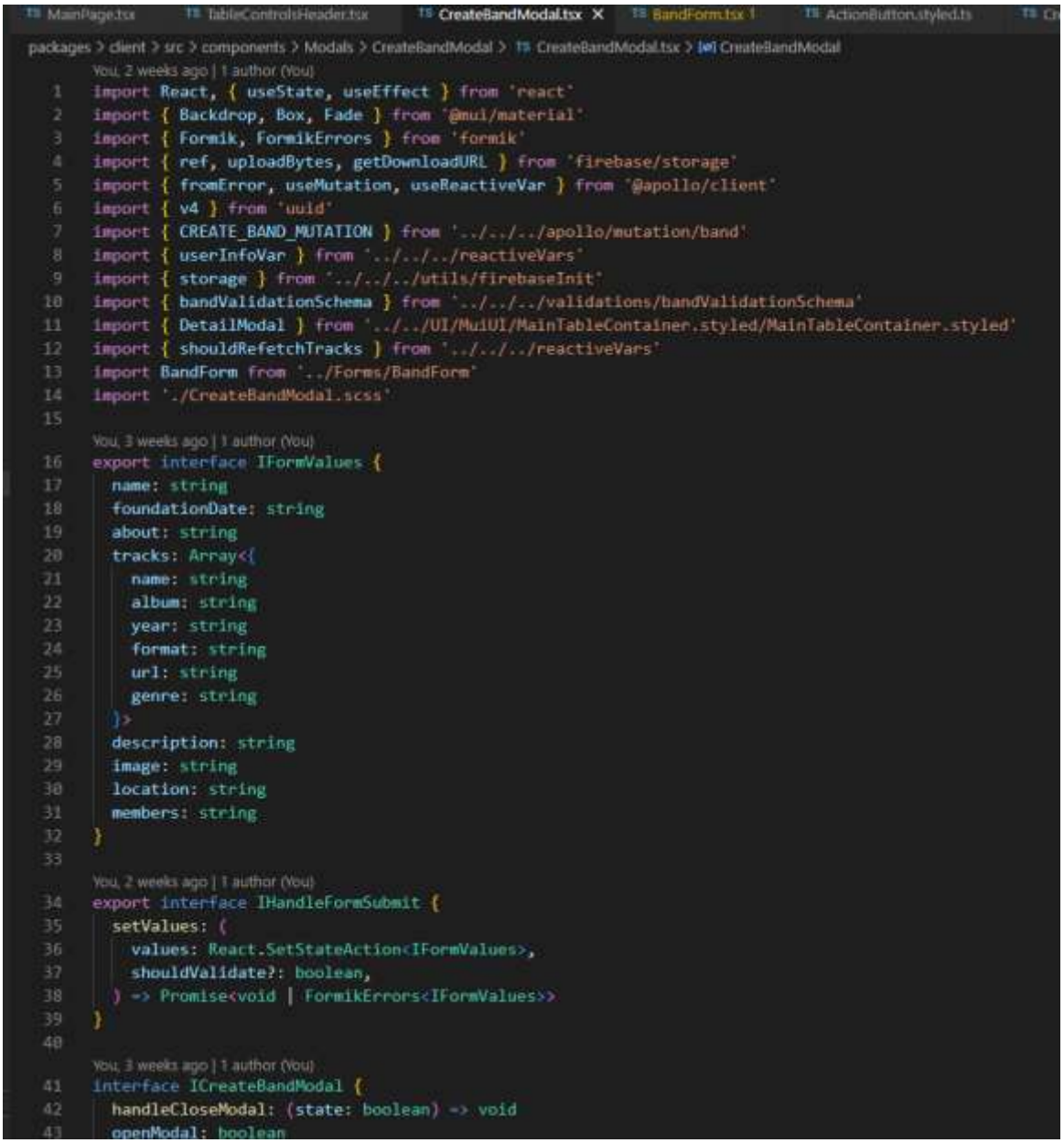


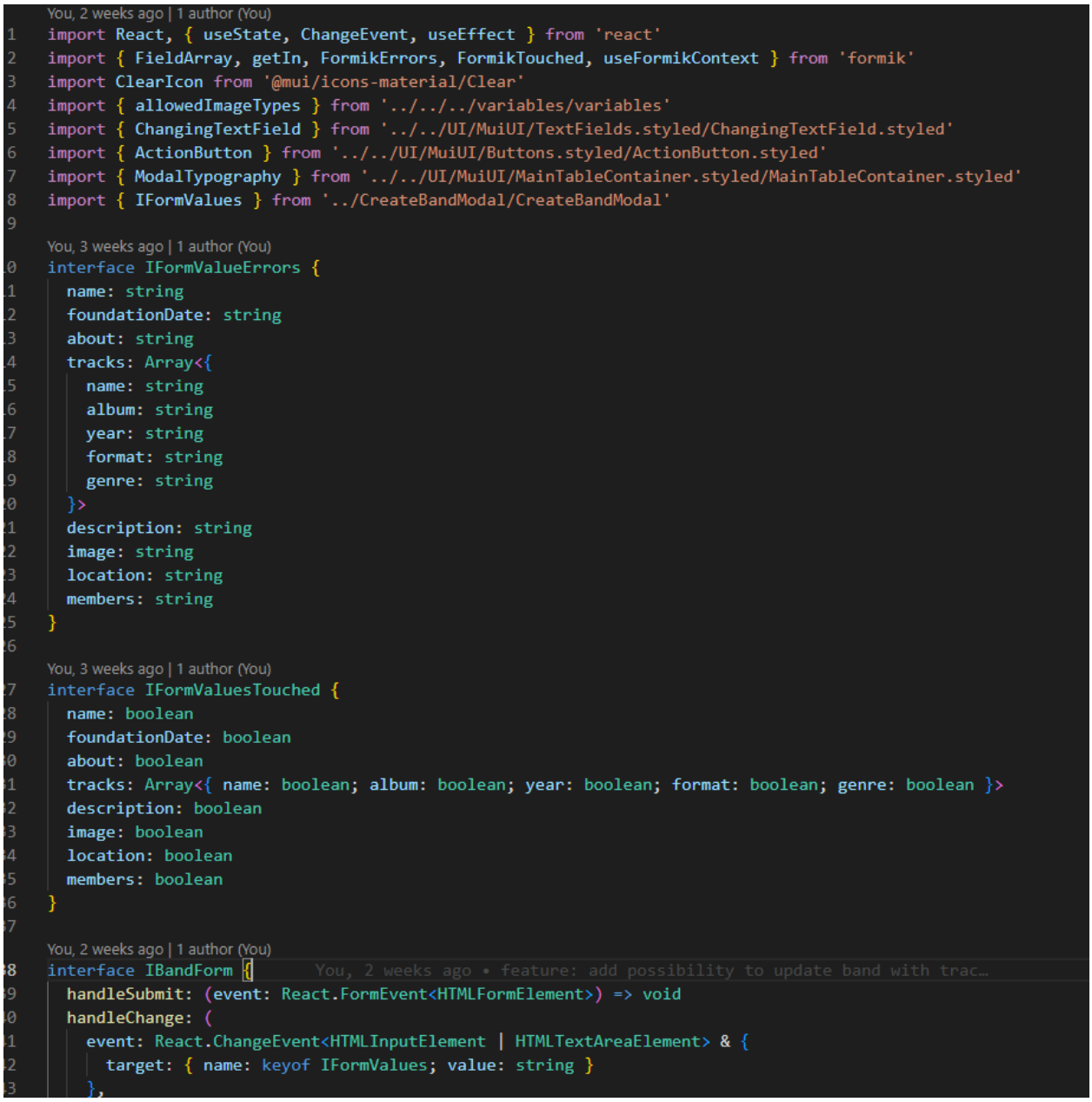
Рисунок 4.62 – Приколад коду компоненти CreateBandModal

Розглянемо також вкладену компоненту BandForm. Цей код, що наведено на (рис. 4.63) представляє компонент React, який відповідає за відображення форми створення гурту. Давайте розглянемо його структуру та функціональність:

- Імпортуються необхідні бібліотеки та компоненти з Material-UI, MUI-Icons та власних джерел;
- Оголошення інтерфейсів;
- Використовує стан previewImg для відображення попереднього зображення обкладинки гурту;

- Використовує стан dndError для відображення повідомлення про помилку у випадку перетягування непідтримуваних файлів;
- Використовує useFormikContext для отримання функцій форми Formik, зокрема resetForm;
- handleDragStart, handleDragLeave: Обробники подій для дозволу та відмови перетягування;
- handleOnDrop: Обробник події, який викликається при киданні файлів на область завантаження. Перевіряє тип файлу та відображає або додає зображення;
- handleFile: Обробник події, який викликається при виборі файлу через стандартний <input type="file">;
- Виводить форму, що складається з різних полів для введення інформації про гурт;
- Використовує компоненти ChangingTextField для текстових полів та ActionButton для кнопки відправки форми;
- Реалізує можливість додавання та видалення треків гурту за допомогою FieldArray та текстових полів для кожного треку;
- Виводить зображення обкладинки гурту з можливістю додавання через перетягування або вибір файлу;
- Приймає різні параметри від батьківського компонента, такі як обробники подій, значення форми, помилки, статус торкання тощо.

Цей компонент дозволяє користувачам вводити та редагувати інформацію про гурт, включаючи треки та зображення обкладинки, і забезпечує валідацію та зручний інтерфейс введення даних.



```
You, 2 weeks ago | 1 author (You)
import React, { useState, ChangeEvent, useEffect } from 'react'
import { FieldArray, getIn, FormikErrors, FormikTouched, useFormikContext } from 'formik'
import ClearIcon from '@mui/icons-material/Clear'
import { allowedImageTypes } from '../../../variables/variables'
import { ChangingTextField } from '../../UI/MuiUI/TextFields.styled/ChangingTextField.styled'
import { ActionButton } from '../../UI/MuiUI/Buttons.styled/ActionButton.styled'
import { ModalTypography } from '../../UI/MuiUI/MainTableContainer.styled/MainTableContainer.styled'
import { IFormValues } from '../CreateBandModal/CreateBandModal'

You, 3 weeks ago | 1 author (You)
interface IFormValueErrors {
  name: string
  foundationDate: string
  about: string
  tracks: Array<{
    name: string
    album: string
    year: string
    format: string
    genre: string
  }>
  description: string
  image: string
  location: string
  members: string
}

You, 3 weeks ago | 1 author (You)
interface IFormValuesTouched {
  name: boolean
  foundationDate: boolean
  about: boolean
  tracks: Array<{ name: boolean; album: boolean; year: boolean; format: boolean; genre: boolean }>
  description: boolean
  image: boolean
  location: boolean
  members: boolean
}

You, 2 weeks ago | 1 author (You)
interface IBandForm {      You, 2 weeks ago • feature: add possibility to update band with trac…
  handleSubmit: (event: React.FormEvent<HTMLFormElement>) => void
  handleChange: (
    event: React.ChangeEvent<HTMLInputElement | HTMLTextAreaElement> & {
      target: { name: keyof IFormValues; value: string }
    },
```

Рисунок 4.63 – Приклад коду компоненти BandForm

Наступна компонента CreateTrackModal зроблена за аналогією, але в неї відсутньо декілька інпутів, та створення треків зроблено на основі вже створеної групи, у селекторі є можливість обрати необхідну групу та додати необхідні треки до неї. Приклад компоненти наведено на (рис. 4.64) та приклад вкладеної форми створення треків на (рис. 4.65).

```
  }

  return (
    <DetailModal
      open={openModal}
      onClose={() => handleCloseModal(false)}
      closeAfterTransition
      disableAutoFocus
      slots={{ backdrop: Backdrop }}
      slotProps={{
        backdrop: {
          timeout: 500,
        },
      }}
    >
      <Fade in={openModal}>
        <Box component="div">
          <Formik
            initialValues={{
              bandId: 0,
              tracks: [{ name: '', album: '', year: '', format: '', url: '', genre: '' }],
            }}
            validationSchema={trackValidationSchema}
            onSubmit={handleFormSubmit}
          >
            {({ handleSubmit, handleChange, values, errors, touched }) => {
              return (
                <TrackForm
                  handleSubmit={handleSubmit}
                  handleChange={handleChange}
                  values={values}
                  errors={errors}
                  touched={touched}
                  bandsData={bandsData?.getAllBands}
                  bandsLoading={bandsLoading}
                  mainFormTitle="Select band to add tracks"
                  tracksFormTitle="Add tracks to band"
                />
              )
            }}
          </Formik>
        </Box>
      </Fade>
    </DetailModal>
  )
}
```

Рисунок 4.64 – Приклад коду компоненти CreateTrackModal

```
94
95        <div className="band_fields_row">
96          <div className="band_fields_column">
97            <InputLabel id="band-name">Selected band</InputLabel>
98            <Select
99              labelId="band-name"
100             id="band-name-select"
101             variant="standard"
102             name="bandId"
103             disabled={isTrackEditing}
104             className="modal-band-selector"
105             value={`${selectedBand?.id ? selectedBand.id : ''}`}
106             onChange={handleSelectChange}
107             label="Selected band"
108             error={touched.bandId && Boolean(errors.bandId)}
109           >
110             {bandsLoading && <LoaderOval height={20} width={20} />}
111             {!bandsLoading &&
112               bandsData.map(band => {
113                 return (
114                   <MenuItem key={band.id} value={band.id}>
115                     {band.name}
116                   </MenuItem>
117                 )
118               })}
119           </Select>
120         </div>
121         {selectedBand && (
122           <ChangingTextField
123             variant="standard"
124             size="small"
125             margin="none"
126             name="location"
127             label="Location"
128             value={selectedBand.location}
129             disabled
130           />
131         )}
132       </div>
133
134       <ModalTypography textAlign="center" variant="h5">
135         {tracksFormTitle}
136       </ModalTypography>
137       <div className="band_fields_row">
138         <div className="band_fields_column">
139           <FieldArray name="tracks">
140             {({ push, remove }) => (
```

Рисунок 4.65 – Приклад коду компоненти TrackForm

Зараз розглянемо основну компоненту MainTable. Цей код, приклад якого наведено на (рис. 4.66) представляє компонент React, який відповідає за відображення таблиці з даними про треки гурту. Давайте розглянемо його структуру та функціональність:

- Імпорт бібліотек та компонентів;
- Оголошення інтерфейсів;

- Використовуються хуки useState для створення станів, таких як порядок сортування (orderDirection), поле сортування (valueToOrderBy), поточна сторінка (page), кількість рядків на сторінці (rowsPerPage), вибрані чекбокси (selectedCheckbox), та відкриття/закриття модального вікна (openModal);
- Використовується хук useQuery для отримання даних про треки гурту з сервера за допомогою запиту GET_ALL_TRACKS_QUERY;
- Відбувається рефетч даних при зміні параметрів сортування, пошуку або призначення для перезавантаження (за допомогою реактивної змінної shouldRefetchTracksValue);
- handleRequestSort: Змінює порядок сортування та поле сортування при кліку на заголовок стовпця;
- handlePageChange: Змінює поточну сторінку при зміні сторінки;
- handleRowsPerPageChange: Змінює кількість рядків на сторінці та скидає поточну сторінку до першої при зміні цієї кількості;
- handleSelectedAllClick: Вибирає всі чекбокси, якщо вибрано загальний чекбокс для вибору всіх;
- handleCheckboxClick: Додає або видаляє ідентифікатор чекбоксу при кліку на нього;
- Використовується модальне вікно (RemoveTableModalWindow), яке відображається при кліку на кнопку видалення (handleOpenModalWindow);
- Модальне вікно дозволяє користувачеві підтвердити видалення вибраних треків гурту;
- Відображається таблиця з даними про треки гурту, з можливістю сортування та вибору рядків для видалення;
- Використовується компоненти Material-UI, такі як Table, TableHead, TableBody, а також власні компоненти для відображення заголовку та тіла таблиці;
- Використовується компонент MainTablePagination для відображення пагінації з можливістю вибору кількості рядків на сторінці;
- При завантаженні даних відображається компонент LoaderOval, який вказує на процес завантаження.

Загалом, цей компонент створений для відображення та управління даними про треки гурту, з можливістю сортування, пагінації та видалення вибраних треків.

Рисунок 4.66 – Приклад коду компоненти MainTable

Приклад фінального вигляду основної сторінки приведено на (рис. 4.67), приклад модального вікна створення групи приведено на (рис. 4.68), приклад створення треків до групи приведено на (рис. 4.69), приклад редагування групи приведено на (рис. 4.70), приклад редагування треку приведено на (рис. 4.71) та приклад видалення треків приведено на (рис. 4.72).

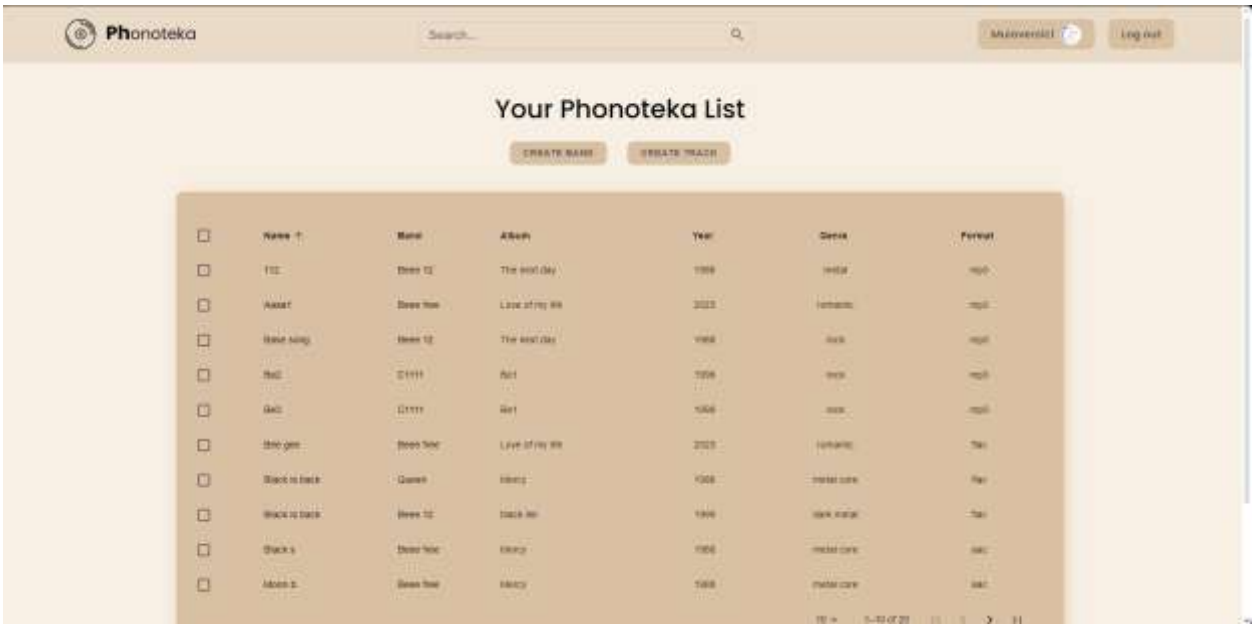


Рисунок 4.67 – Приклад фінального вигляду основної сторінки

Рисунок 4.68 – Приклад фінального вигляду модального вікна ствоерння групи з треками

Рисунок 4.69 – Приклад фінального вигляду модального вікна ствоерння треків до вибраної групи

Рисунок 4.70 – Приклад фінального вигляду модального вікна редагування групи

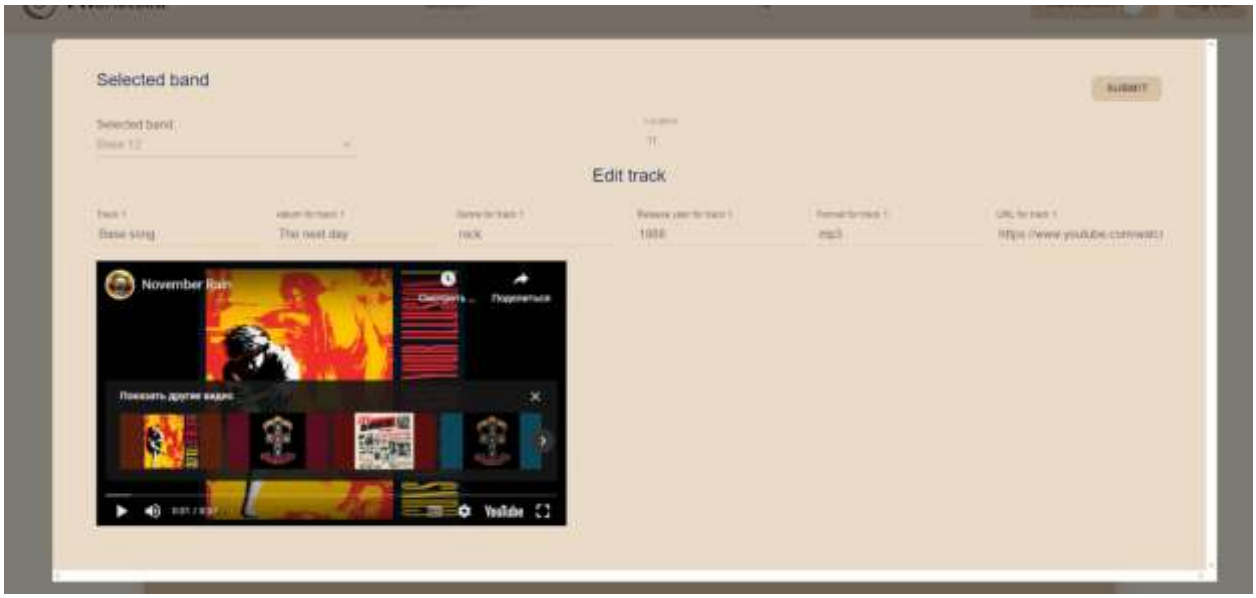


Рисунок 4.71 – Приклад фінального вигляду модального вікна редагування треку

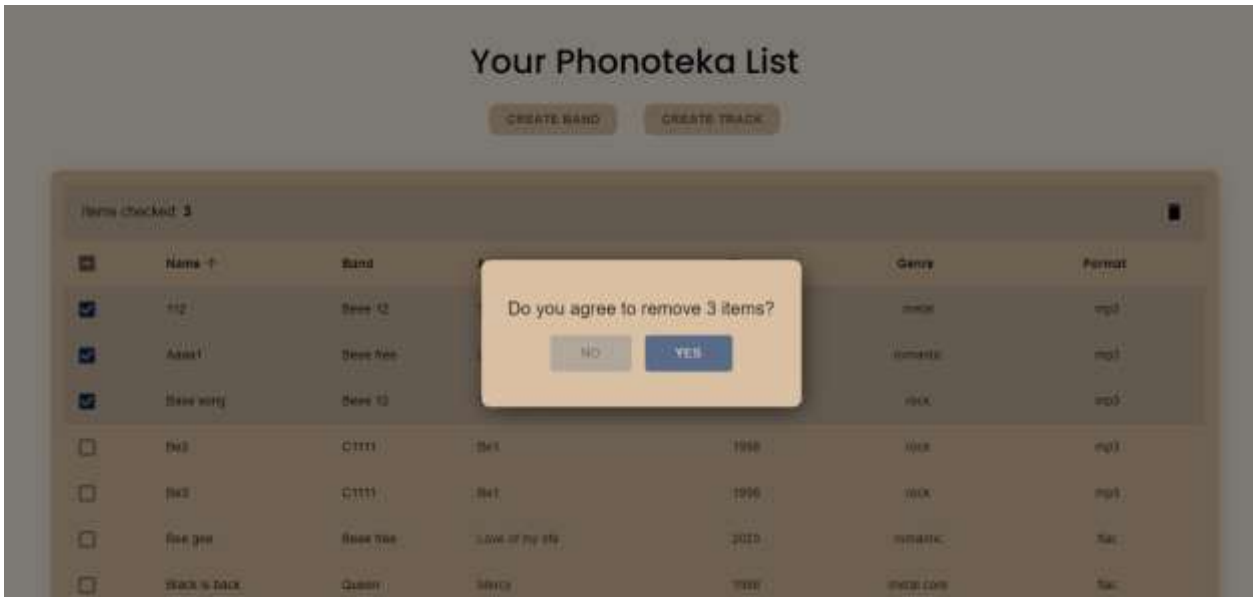


Рисунок 4.72 – Приклад фінального вигляду модального вікна видаленян треків

4.23 Написання сторінки для профілю користувача

Компонента сторінки користувача має функцію редагування та відображення корисної інформації, включаючи дату регістрації, дату оновленя профілю та графік за збереженими даними щодо треків. Приклад коду для профілю користувача приведено на (рис. 4.73), приклад коду для редагування профілю приведено на (рис. 4.74) та приклад коду для графіку приведено на (рис. 4.75).

Рисунок 4.73 – Приклад коду для сторінки користувача

```
39
40      onSubmit: async values => {
41        try {
42          await updateUserMutation({
43            variables: { input: { ...values, imgUrl: userNewImage || userData.imgUrl } },
44          })
45          handleChangeProfile()
46        } catch (serverError) {
47          console.error('Sign up error!', fromError(serverError))
48        }
49      },
50    })
51
52    const { handleSubmit, handleChange, setValues, values, errors, touched } = formik
53
54    useEffect(() => {
55      setValues({ name: userData.name, email: userData.email })
56    }, [userData])
57
58    const handleOpenModal = () => {
59      setOpenModal(true)
60    }
61
62    const handleCloseModal = () => {
63      setOpenModal(false)
64    }
65
66    const handleChangeProfile = useCallback(() => {
67      if (userProfilePath) {
68        navigate(userProfilePath)
69      }
70    }, [userProfilePath])
71
72    return (
73      <>
74        <div className="profile_user_content">
75          <ul className="profile_user_content_left">
76            <li>
77              <div className="default_item_image" onClick={handleOpenModal}>
78                {userNewImage ? (
79                  <Avatar className="profile_user_content_image" src={userNewImage} />
80                ) : (
81                  <AccountCircleIcon className="default_image" />      You, 4 weeks ago • fea
82                )}
83
84                <AddAPhotoIcon className="default_image_change" />
```

Рисунок 4.74 – Приклад коду для сторінки оновлення даних користувача

```
      setTimeout(() => {
        setChartSpeedAngle(i)
      }, i / speedCoefficient)
    }
  }, [])

  const paletteChart = [
    '#fd7f6f',
    '#7eb0d5',
    '#b2e061',
    '#bd7ebe',
    '#ffb55a',
    '#ffee65',
    '#beb9db',
    '#fdcce5',
    '#8bd3c7',
  ]

  const pieParams = { height: 500, margin: { left: 80, right: 80, top: 40, bottom: 100 } }

  return (
    <PieChart
      series={[
        {
          data: genreData,
          highlightScope: { faded: 'global', highlighted: 'item' },
          faded: { additionalRadius: -5, color: 'gray' },
          cornerRadius: 0,
          startAngle: 0,
          endAngle: chartSpeedAngle,
        },
      ]}
      colors={paletteChart}
      slotProps={{
        legend: {
          direction: 'row',
          position: { vertical: 'bottom', horizontal: 'middle' },
          padding: 0,
        },
      }}
      {...pieParams}
    />
  )
}

export default ProfileChart
```

Рисунок 4.75 – Приклад коду компоненти з графіком даних користувача

Фінальний вигляд сторінки представлено на (рис. 4.76), сторінку редагування показано на (рис. 4.77) та графік представлено на (рис. 4.78).

Рисунок 4.76 – Приклад сторінки користувача

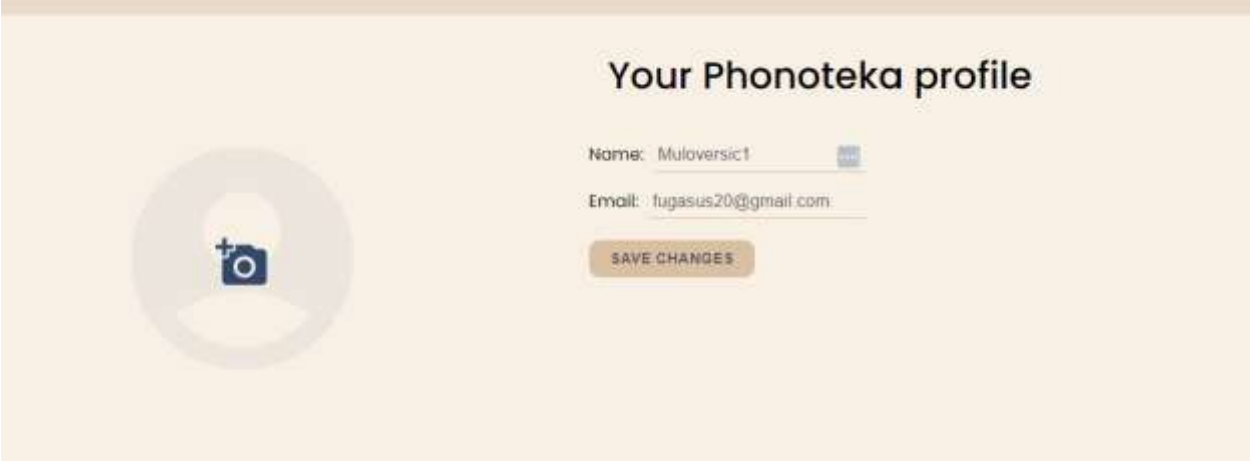


Рисунок 4.78 – Приклад сторінки редагування даних користувача

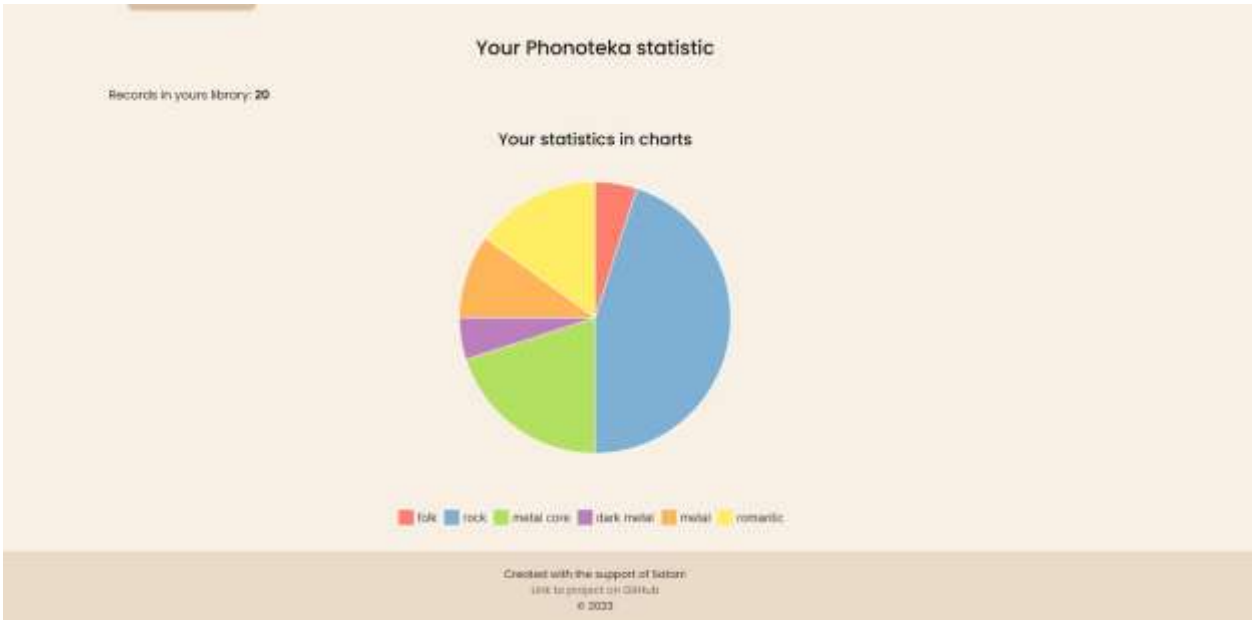


Рисунок 4.79 – Приклад сторінки з графіком даних користувача

4.24 Висновки до розділу

У цьому розділу було ретельно розглянуто та описано процес створення повноціного веб-додатку з використанням сучасних технологій. Описана структура застосунку, розглянуто приклади написаного коду, та його функції. Продемонстровано фінальний вигляд застосунку зі сторони користувача.

ВИСНОВКИ

У ході цієї атестаційної роботи розглянуто та створено веб-застосунок, який демонструє високий рівень сучасності та ефективність використання технологій. Використання React дозволяє створювати компоненти, що полегшує розподіл логіки та покращує читабельність коду. Використання Formik для роботи з формами забезпечує зручну обробку введених даних та їх валідацію.

Архітектура застосунку дозволяє ефективно взаємодіяти з сервером за допомогою Apollo Client і використовувати GraphQL для оптимізації передачі даних між клієнтом і сервером. Використання Prisma як інструменту для взаємодії з базою даних PostgreSQL вказує на сучасний підхід до управління даними. Material-UI використовується для створення інтерфейсу, що відповідає сучасним стандартам дизайну. Застосунок має адаптивний та приємний зовнішній вигляд.

Використання Git для контролю версій гарантує ефективне управління кодом та сприяє спільній роботі над проектом.

Наслідок великої кількості технологій у застосунку - це його високий рівень сучасності, але також може створювати певні труднощі в розумінні та підтримці для новачків або розробників, які не мають досвіду з усіма використаними технологіями. Попри це, взаємодія з GraphQL та використання Apollo Client сприяють оптимізації роботи застосунку та зменшенню трафіку мережі. Також, слід відзначити використання патерну контейнер-представлення (container-presentation) у компонентній структурі застосунку, де розділення логіки та представлення дозволяє зберігати код чистим та легко розширюваним. Додатково, використання реактивних змінних із бібліотекою Apollo Client полегшує відстеження стану даних та автоматизує їх оновлення на клієнті при змінах на сервері.

Застосунок також має можливість додавання та видалення об'єктів (треків у випадку з аудіо-бандою) через інтерфейс користувача, що свідчить про повноту його функціоналу.

Врахування аспектів безпеки та обробка помилок у коді дозволяють забезпечити стабільну роботу застосунку та уникнути потенційних загроз безпеки. Таким чином, дана робота наголошує на тому, що розглянутий веб-застосунок представляє сучасний та добре організований підхід до розробки веб-додатків, використовуючи передові технології та забезпечуючи високий рівень користувацької якості.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Офіційна документація React - https://ru.legacy.reactjs.org (дата звернення: 23.11.2023).

2. Офіційна документація MUI - https://mui.com (дата звернення: 24.11.2023).

3. Офіційна документація Formik - https://formik.org (дата звернення: 24.11.2023).

4. Офіційна документація Apollo/GraphQL - https://www.apollographql.com (дата звернення: 25.11.2023).

5. Офіційна документація Prisma - https://www.prisma.io (дата звернення: 25.11.2023).

6. Офіційна документація PostgreSQL - https://www.postgresql.org (дата звернення: 26.11.2023).

7. Офіційна документація Firebase - https://firebase.google.com (дата звернення: 28.11.2023).

8. С.М. Порошин, В.М. Карташов, В.В. Усик, Р.І. Цехмістро, І.С.Бєліков. Технології створення складових мультимедійного контенту. Анімація та web-анімація. Навчальний посібник –Харків, НТУ.ХПІ, 2022.-314с.

9. М.А. Омаров Основи технологій сучасної web-анімації. [Текст] Навчальний посібник /М.А. Омаров, В.М. Карташов, Р.І. Цехмістро, В.В. Усик – Харків ХНУРЕ- 2022р.- 214с.

10. V. M. Kartashov, V. N. Oleynikov, S. A. Sheyko, I. V. Koryttsev, S. I. Babkin, O. V. Zubkov, "Peculiarities of small unmanned aerial vehicles detection and recognition," Telecommunications and Radio Engineering, 2019, V. 78, Iss. 9, pp. 771–781. DOI: 10.1615/TelecomRadEng.v78.i9.30.

11. V. N. Oleynikov, O. V. Zubkov, V. M. Kartashov, I. V. Korytsev, S. I. Babkin, S.A. Sheiko, "Investigation of detection and recognition efficiency of small unmanned aerial vehicles on their acoustic radiation," Telecommunications and Radio Engineering, 2019, V. 78, Iss. 9, pp. 759–770.

DOI: 10.1615/TelecomRadEng.v78.i9.20.

12. Kartashov, V.M., Sidorov, G.I., Sheiko, S.A., Kolendovska, M.M., Sergienko O.Yu. Principles of Construction and Assessment of technical Characteristics of multi-Frequency atmospheric Sodar in the Humidity Measurement Mode / Telecommunications and Radio Engineering.- New York. - 2020.- Vol. 79, №4.- P.323-333. (стаття). DOI: 10.1615/TelecomRadEng.v79.i4.50.

13. Kartashov, V.M., Oleynikov V.N, Zubkov, O.V., Korytsev I.V., Babkin, S. I., Sheiko, S.A.,. Kolendovskaya, M.M. Spatial-temporal Processing of acoustic Signals of Unmanned Aerial Vehicles; Telecommunications and Radio Engineering, 2020. Vol. 79, Iss, 9, pp.769-780.

14. V.M. Semenets, V.M. Kartashov, V.I. Leonidov. Features of Acoustic Noise of Small Unmanned Aerial Vehicles // Telecommunications and Radio Engineering.- New York. - 2020.- Vol. 79, №11.- P. 985-995. DOI: 10.1615/TelecomRadEng.v79.i11.80 (стаття).

15. Oleynikov V.N., Kartashov, V.M., Babkin, S. I., Zubkov, O.V., Korytsev I.V., Sheiko, S.A.,. Seleznov I.S. Structure and Parameter Unmanned Aerial Vehicles Sound Fields/ Telecommunications and Radio Engineering.- New York. - 2020.- Vol. 79, №17.- P.1539-1550. DOI: 10.1615/TelecomRadEng.v79.i17.50 (стаття).

16. В.А. Тихонов, В.М. Карташов, В.М. Олейніков, В.І. Леонідів, Л.П. Тимошенко, І.С. Селезнєов, Н.В. Рибніков. Виявлення-розпізнавання безпілотних літальних апаратів з використанням складової моделі авторегресії їхнього акустичного випромінювання// Вісник НТУУ «КПІ». Радіотехніка. Радіоапаратобудування. – 2020. – Віп. №81. - С.38-46. (Web of Science) Карташов В.М., Коритцев І.В., Олійников В.М., Зубков О.В., Шейко С.А., Бабкін С.І., Левський Н.А., Селезньов І.С. Алгоритми пеленгації безпілотних літальних апаратів з їхнього акустичного випромінювання// Радіотехніка. (Харків). - 2019. - Вип. 196. - С. 22-31.Карташов В.М., Харченко О.І., Чумаков В.І. Використання ефекту стохастичного резонансу аналізу спектрів акустичного випромінювання малих безпілотних літальних апаратів

// Радіотехніка. (Харьков). — 2019. — Вып. 197. — С. 100-106.

17. Карташов В.М., Сидоров Г.І., Толстих Є.Г., Шаповалов С.В. Акустичний вимірювач швидкості вітру в атмосферному прикордонному шарі// Радиотехника. (Харьков). — 2019. — Вып. 199. – С. 54-58.

18. A Comparative Example Between The Use Of Pca And Mds For Image Classification / Hernandez, W., Mendez, A., Flor-Unda, O., Camejo, I.M., Kolendovska, M.// IEEE International Symposium on Industrial Electronics, 29th IEEE International Symposium on Industrial Electronics, ISIE 2020; Delft; Netherlands; 17 June 2020 до 19 June 2020; Volume 2020-June, June 2020, № 9152565, Pages 1353-1358.

19. Algorithm For Generating Refined Frequency Estimates In Atmospheric Radio Sounding Systems / Kartashov V., Hernandez W., Hernandez-Balbuena D., M. Kolendovska, Konovalenko O., Melnyk V.// IEEE International Symposium on Industrial Electronics, 29th IEEE International Symposium on Industrial Electronics, ISIE 2020; Delft; Netherlands; 17 June 2020 до 19 June 2020; Volume 2020-June, June 2020, № 9152562, Pages 79-82.

20. Application of Fast Frequency Shift Measurement Method for INS in Navigation of Drones / D. Avalos-Gonzalez, D.H. Balbuena, V. Tyrsa, V.M. Kartashov, M. Kolendovska, S. Sheiko, O. Sergiyenko, V. Melnyk, F.N. Murrieta-Rico // IECON 2018 – 44th Annual Conference of the IEEE Industrial Electronics Society. – P. 3159–3164.

21. Avalos-Gonzalez, D., Sergiyenko, O., Hernandez-Balbuena, D., Tyrsa,V., Kartashov V.M., V.,Rivas-Lopes, M., Murrieta-Rico, F.N. Constraints definition and application optimization based on geometric analysis of the frequency measurement method by pulse coincidence// Measurement: Journal of the International Measurement Confederation (USA). 2018, V.126. P. 184-193.

22. Book “Control and Signal Processing Applications for Mobile and Aerial Robotic Systems”, Hardback - Advances in Computational Intelligence and Robotics English. Edited by Oleg Sergiyenko, Moises Rivas-Lopez, Wendy Flores-Fuentes, Julio Cesar Rodríguez-Quiñonez, Lars Lindner. Editorial IGI Global,

Hershey, United States, January 2020, 340 páginas. ISBN10 152259924X, ISBN13 9781522599241

23. Cesar Sepulveda-Valdez ; Oleg Sergiyenko ; Vera Tyrsa ; Wendy Flores-Fuentes ; Julio César Rodríguez-Quiñonez ; Fabian Natanael Murrieta-Rico ; Jesús Elías Miranda-Vega ; Paolo Mercorelli ; Marina Kolendovska. "Geometric analysis of a laser scanner functioning based on dynamic triangulation," 2020 IEEE 29th International Symposium on Industrial Electronics (ISIE), Delft, Netherlands, 17-19 of June 2020, pp. 1398-1403, doi: 10.1109/ISIE45063.2020.9152268, https://ieeexplore.ieee.org/abstract/document/9152268.

24. Cuauhtémoc Mariscal-García; Wendy Flores-Fuentes; Daniel Hernández-Balbuena; Julio C. Rodríguez-Quiñonez ; Oleg Sergiyenko. "Classification of Vehicle Images through Deep Neural Networks for Camera View Position Selection," 2020 IEEE 29th International Symposium on Industrial Electronics (ISIE), Delft, Netherlands, 17-19 of June 2020, pp. 1376-1380, doi: 10.1109/ISIE45063.2020.9152440, https://ieeexplore.ieee.org/abstract/document/9152440.

25. Developing and Applying Optoelectronics in Machine Vision/ O. Sergiyenko, J.C. Rodriguez-Quiñonez, IGI Global, 2016; 341p.

26. Experimental estimation of direction finding to unmanned air vehicles algorithms efficiency by their acoustic emission, /Oleynikov, V., Zubkov, O., Kartashov, V., ...Sheiko, S., Babkin, S.//2019 IEEE International Scientific-Practical Conference: Problems of Infocommunications Science and Technology, PIC S and T 2019 - Proceedings, 2019, стр. 175-178, 9061337.

27. Features of acoustic noise of small unmanned aerial vehicles / Semenets, V.V., Kartashov, V.M., Leonidov, V.I. //Telecommunications and Radio Engineering (English translation of Elektrosvyaz and Radiotekhnika), 2020, 79(11), стр. 985-995.

28. Geometric Analysis Of A Laser Scanner Functioning Based On Dynamic Triangulation /Sepulveda-Valdez, C., Sergiyenko, O., Tyrsa, V, Mercorelli,

P., Kolendovska, M.// IEEE International Symposium on Industrial Electronics, 29th IEEE International Symposium on Industrial Electronics, ISIE 2020; Delft; Netherlands; 17 June 2020 до 19 June 2020; Volume 2020-June, June 2020, № 9152268, Pages 1398-1403https://ieeexplore.ieee.org/abstract/document/9152255 https://ieeexplore.ieee.org/document/9161870.

29. I. Y. A. Corpus, L.Lindner, O.Sergiyenko. "Transimpedance Amplifier for Laser Scanning System Range Extension," 2020 IEEE 29th International Symposium on Industrial Electronics (ISIE), Delft, Netherlands, 17-19 of June 2020, pp. 1421-1426, doi: 10.1109/ISIE45063.2020.9152487. https://ieeexplore.ieee.org/abstract/document/9152487

30. Ivanov, M., Sergiyenko, O., Mercorelli, P., Hernandez, W.c, Rodriguez Quinonez, J.C.d, Katashov V., Kolendovska, M., Iryna, T. Effective informational entropy reduction in multi-robot systems based on real-time TVS. IEEE International Symposium on Industrial Electronics, 2019-June,8781209, с. 1162-1167.

31. Jonathan J. Sanchez-Castro ; Julio C. Rodríguez-Quiñonez ; Luis R. Ramírez-Hernández ; Guillermo Galaviz ; Daniel Hernández-Balbuena ; Gabriel Trujillo-Hernández ; Wendy Flores-Fuentes ; Paolo Mercorelli ; Wilmar Hernández-Perdomo ; Oleg Sergiyenko ; Félix Fernando González-Navarro. "A Lean Convolutional Neural Network for Vehicle Classification," 2020 IEEE 29th International Symposium on Industrial Electronics (ISIE), Delft, Netherlands, 17-19 of June 2020, pp. 1365-1369, doi: 10.1109/ISIE45063.2020.9152274. https://ieeexplore.ieee.org/abstract/document/9152274

32. Lindner, L., Sergiyenko, O., Rivas-López, M., (...), Gurko, A., Kartashov, V.M. Machine vision system for UAV navigation; IEEE, 2016 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles and International Transportation Electrification Conference, ESARS-ITEC, 2016; pp.1–6. DOI: 10.1109/ESARS-ITEC.2016.7841356.

33. M. Ivanov, O. Sergiyenko, V. Tyrsa, P. Mercorelli, V. Kartashov, W. Hernandez, S. Sheiko, M. Kolendovska. Individual scans fusion in virtual knowledge base for navigation of mobile robotic group with 3D TVS // Proceedings of 44th Annual Conference of IEEE Industrial Electronics Society (IECON).. -2018. – Washington DC, USA. -S. 3187-3192. . ISBN 978-1-5090-6683-4/18/.

34. Murrieta-Rico, F.N., Petranovskii, V., Galvan, D.H., Sergiyenko, O., Yocupicio-Gaxiola, R.I., De Dios Sanchez-Lopez, J. Phase effect in frequency measurements of a quartz crystal using the pulse coincidence principle. 2020 IEEE 29th International Symposium on Industrial Electronics (ISIE), Delft, Netherlands, 17-19 of June 2020, pp. 185-190, 9152255, DOI: 10.1109/ISIE45063.2020.9152255.

35. Oleksandr Sotnikov, Vladimir Kartashov, Oleksandr Tymochko, Oleg Sergiyenko, Vera Tyrsa, Paolo Mercorelli, Wendy Flores-Fuentes. Methods for Ensuring the Accuracy of Radiometric and Optoelectronic Navigation Systems of Flying Robots in a Developed Infrastructure. Chapter 16// Machine Vision and Navigation; Springer, Cham. pp.537–578. Editors: Sergiyenko, Oleg, Flores-Fuentes, Wendy, Mercorelli, Paolo. DOI: 10.1007/978-3-030-22587-2_16.

36. Optical detection of unmanned air vehicles on a video stream in a real-time/Kartashov, V., Oleynikov, V., Zubkov, O., Sheiko, S.// 2019 International Conference on Information and Telecommunication Technologies and Radio Electronics, UkrMiCo 2019 - Proceedings, 2019, 9165362/.

37. Principles Of Construction And Assessment Of Technical Characteristics Of Multi-Frequency Atmospheric Sodar In The Humidity Measurement Mode / Kartashov, V.M., Sidorov, G.I., Sheiko, S.A., Kolendovskaya, M.M., Sergienko, O.Yu. // Telecommunications And Radio Engineering (English Translation Of Elektrosvyaz And Radiotekhnika), 2020, ISSN Print: 0040-2508, ISSN Online: 1943-6009, DOI: 10.1615/TelecomRadEng.v79.i4.50, p. 323-333/.

38. Research Of The Uncertainty Of Measurement Frequencies And Definitions Of The Frequency Signal In The Waveguide With Respect To Power / Semenets, V.Zakharov, I. Serhiienko, M., Kartashov, V.M, , Kolendovska, M.,

Hernandez, W., Hipolito, J.I.N., , Tyrsa, V.// 45th Annual Conference of the IEEE Industrial Electronics Society, IECON 2019; Lisbon Congress CenterLisbon; Portugal; 14 October 2019 до 17 October 2019; CFP19IEC-ART; Код 155980, Volume 2019-October, October 2019, № 8927203, Pages 4674-4679.

39. Spatial-Temporal Processing Of Acoustic Signals Of Unmanned Aerial Vehicles /Kartashov V.M., Oleinikov V.N., Zubkov O.V., Sheiko S.A., Kolendovska M.M.// Telecommunications And Radio Engineering (English Translation Of Elektrosvyaz And Radiotekhnika), 2020, ISSN Print: 0040-2508, ISSN Online: 1943-6009, DOI: 10.1615/Telecomradeng.v79.i9.40, p. 769-780.

40. Stereoscopic Vision Systems In Machine Vision, Models, And Applications (Book Chapter)/ Ramírez-Hernández, L.R., Rodríguez-Quiñonez, J.C., Castro-Toscano, M.J., Kolendovska, M., Murrieta-Rico, F.N.// Machine Vision And Navigation, 2019 Machine Vision and Navigation30 September 2019, Pages 241-265.

41. StrelkovaT., KartashovV., Lytyuga A., Strelkov A. Theoretical Methods of Images Processing in Optoelectronic Systems. Chapter 16. // Biometrics: Concepts, Methodologies, Tools, and Applications; Oleg Sergiyenko and Julio C. Rodriguez-Quiñonez. (341p.), IGI Global, 2017; pp. 361-381. DOI: 10.4018/978-1-5225-0983-7.ch016.

42. StrelkovaT., KartashovV., Lytyuga A.,StrelkovA. Theoretical Methods of Images Processing in Optoelectronic Systems. Chapter 6// Developing and Applying Optoelectronics in Machine Vision; Oleg Sergiyenko and Julio C. Rodriguez-Quiñonez. (341p.) – USA, Herhey, IGI Global, 2016; pp.180-205.

43. Sytnik O., KartashovV. Methods and Algorithms for Technical Visionin Radar Introscopy. Chapter 13// Optoelectronics in Machine Vision-Based Theories and Applications. IGI Global, 2019; pp. 373-391.

44. The Use of Factorization and Multimode Parametric Spectra in Estimating Frequency and Spectral Parameters of Signal/Semenets, V., Kartashov, V., Sergiyenko, O., ...Rodriguez-Quinonez, J.C., Flores-Fuentes, W.//IEEE International Symposium on Industrial Electronics, 2020, 2020-June, p. 215-219.

45. Unda, O.F., Hernandez, W., Vargas, O., Mendez, A., Sergiyenko, O., Tyrsa, V. Construction of a robotic platform of differential type for first-year students of electronic engineering, 2020 International Symposium on Power Electronics, Electrical Drives, Automation and Motion, SPEEDAM 2020, 24-26 de junio de 2020, Sorrento, Italia, pp. 538-543, 9161870, DOI: 10.1109/SPEEDAM48782.2020.9161870.

46. Wilmar Hernandez ; Alfredo Mendez ; Omar Flor-Unda ; Vicente Gonzalez-Posada ; Jose Luis Jimenez ; Oleg Sergiyenko ; Julio C. Rodriguez-Quiñonez ; Mykhailo Ivanov ; Ivan Menes Camejo ; Marina Kolendovska. "A comparative example between the use of PCA and MDS for image classification," 2020 IEEE 29th International Symposium on Industrial Electronics (ISIE), Delft, Netherlands, 17-19 of June 2020, pp. 1353-1358, doi: 10.1109/ISIE45063.2020.9152565.