МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет

імені Петра Могили

Факультет комп’ютерних наук

Кафедра комп'ютерної інженерії

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри,
д-р техн. наук, проф.

_____ І. М. Журавська

« __ » _____ 2024 р.

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Архітектура кросплатформного застосунку на базі React та REST API

Спеціальність 123 Комп’ютерна інженерія

123 – КБР.01 – 405.22010512

Студент _____ В. О. Матвєєв
підпис
«__» _____ 202__ р.

Керівник канд. техн. наук, доцент _____ Я. М. Крайник
підпис
«__» _____ 202__ р.

Миколаїв – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили
Факультет комп'ютерних наук
Кафедра комп'ютерної інженерії

ЗАТВЕРДЖУЮ

Зав. кафедри _____ І. М. Журавська

«_____» _____ 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної бакалаврської роботи

Видано студенту групи 405 факультету комп'ютерних наук

Матвєєву В'ячеславу Олександровичу
(прізвище, ім'я, по батькові студента)

1. Тема кваліфікаційної роботи

Архітектура кросплатформного застосунку на базі React та REST API

Затверджена наказом по ЧНУ ім. Петра Могили від 30.01.2024 № 17.

2. Строк представлення кваліфікаційної роботи «____» __червня___ 2024 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні
Очікується аналіз наявних архітектурних рішень для застосунків на базі React та REST API, спроєктована архітектура і застосована для програмної реалізації клієнтської частини комплексу. Створити апаратну частину комплексу. Забезпечити кросплатформність та пояснити фактори і шляхи забезпечення.

4. Перелік питань, що підлягають розробці
- програмно-апаратний комплекс на основі React та Rest API;
- огляд складових архітектури застосунку на базі React та REST API;
- аналіз поняття REST API;
- огляд поширених архітектурних рішень для таких застосунків;
- проєктування архітектури для клієнтської частини застосунку;
- розробка програмної частини комплексу;
- розробка апаратної частини комплексу;
- забезпечення кросплатформності застосунку;

5. Перелік графічних матеріалів
Презентація

6. Завдання до спеціальної частини
Описати складові архітектури застосунків на базі React та REST API, розглянути існуючі рішення архітектурної побудови таких застосунків, спроєктувати архітектуру клієнтської частини застосунку, створити програмну частину комплексу, створити апаратну частину комлексу, забезпечити кросплатформність застосунку і пояснити яким чином це були реалізовано.

7. Консультанти:

Консультант	Кафедра (організація)	Частина роботи
Макарова О. В. ст. викл.	кафедра екології Медичного інституту ЧНУ імені Петра Могили	Спеціальна частина з охорони праці

Керівник роботи
_____доцент Крайник Ярослав Михайлович_____
(посада, прізвище, ім'я, по батькові)

_____
(підпис)

Завдання прийнято до виконання

_____Матвєєв В'ячеслав Олександрович_____
(прізвище, ім'я, по батькові студента)

_____
(підпис)

Дата видачі завдання «____» _____ 20_____ р.

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

Тема: Архітектура кросплатформного застосунку на базі React та REST API

№	Найменування роботи	Початок	Закінчення	Примітки
1	Розробка та затвердження завдання на виконання КР	21.12.2023	21.12.2023	Виконано
2	Огляд літератури за темою роботи	27.12.2023	27.01.2024	Виконано
3	Складання календарного плану БКР	28.01.2024	01.02.2024	Виконано
4	Аналіз предметної області	02.02.2024	02.03.2024	Виконано
5	Розробка проєктних рішень	02.03.2024	23.03.2024	Виконано
6	Програмна реалізація застосунку	23.03.2024	23.04.2024	Виконано
7	Оформлення КРБ та презентації	23.04.2024	27.05.2024	Виконано
8	Попередній захист	28.05.2024	28.05.2024	Виконано
9	Розробка частини з охорони праці	29.05.2024	31.05.2024	Виконано
10	Попередній захист	04.06.2024	04.06.2024	Виконано
11	Завершення оформлення КР та презентації	05.06.2024	16.06.2024	Виконано
12	Відгук керівника КРБ	16.06.2024	16.06.24	Виконано
13	Рецензування	16.06.2024	16.06.2024	Виконано
13	Захист бакалаврської кваліфікаційної роботи	24.06.2024	24.06.2024	Виконано

Розробив здобувач ВО ____Матвєєв В’ячеслав Олександрович___

_____ _____

(прізвище, ім’я, по батькові) *(підпис)*

«____» _____ 20_24_ р.

Керівник роботи _____Крайник Ярослав Михайлович___ _____

(посада, прізвище, ім’я, по батькові) *(підпис)*

«____» _____ 20_24_ р.

АНОТАЦІЯ

до кваліфікаційної бакалаврської роботи
«Архітектура кросплатформного застосунку на базі React та REST API»
Студент 405 гр.: Матвєєв В’ячеслав Олександрович
Керівник: канд. техн. наук, доцент Крайник Ярослав Михайлович

Кваліфікаційна бакалаврська робота присвячена розробці програмно-апаратного комплексу, дослідженню архітектури застосунку на базі React та REST API, аналізу взаємодії клієнтської частини застосунку з сервером, та розгляду необхідних інструментів для забезпечення кросплатформності. Актуальність теми роботи обумовлена стрімким розвитком технологій та постійним зростанням попиту на кросплатформні застосунки, які використовують такі інструменти, як React та REST API.

Комплекс складається з клієнтського та серверного застосунків та апаратної частини розгорнутого серверного застосунку. Програмний код написано на мові JavaScript.

В результаті роботи було програмно реалізовано прототип застосунку з типом зв’язку клієнт-сервер. Окрім цього, реалізовано та розгорнено прототип серверного застосунку.

Робота складається з вступу, чотирьох розділів, висновків та додатків. У першому розділі проведено аналітичний огляд архітектури кросплатформного застосунку на базі React та REST API. У другому розділі проєктування архітектури клієнтської частини застосунку та здійснено програмну реалізацію комплексу. У третьому розділі створення апаратної частини комплексу. Четвертий розділ присвячений забезпеченню кросплатформності клієнтської та серверної частин застосунку. У висновках підсумовано та проаналізовано результати виконаної у розділах роботи.

Кваліфікаційна бакалаврська робота викладена на 60 сторінках, містить 4 розділи, 39 рисунків, 28 джерел посилання, 1 додаток.

Ключові слова: *клієнт, клієнтська частина застосунку, сервер, серверна частина застосунку, кросплатформність, React, REST API, JavaScript, Heroku, Postman.*

ABSTRACT
of the Bachelor's Thesis
"Architecture of a cross-platform application based on React and REST API"
Student: Matvieiev Viacheslav Oleksandrovych
Supervisor: Ph.D. technical sciences, associate professor Kraynyk Yaroslav Mykhaylovych

The bachelor's qualification thesis is devoted to the development of a software and hardware complex, the study of the application architecture based on React and REST API, the analysis of the interaction between the client-side of the application and the server, and the consideration of the necessary tools to ensure cross-platform compatibility. The relevance of the topic is due to the rapid development of technologies and the constantly growing demand for cross-platform applications that use such tools as React and REST API.

The complex consists of client and server applications and the hardware part – the deployed server application. The software is written in JavaScript.

As a result of the work, a prototype of the application with a client-server connection type was programmatically implemented. Additionally, a prototype of the server application was implemented and deployed.

The work consists of an introduction, four chapters, conclusions, and appendices. The first chapter provides an analytical review of the architecture of a cross-platform application based on React and REST API. The second chapter covers the design of the client-side architecture of the application and the software implementation of the complex. The third chapter is dedicated to the creation of the hardware part of the complex. The fourth chapter focuses on ensuring the cross-platform compatibility of the client and server parts of the application. The conclusions summarize and analyze the results of the work carried out in the chapters.

The bachelor's qualification work is presented on 60 pages, contains 4 chapters, 39 figures, 28 references, and 1 appendix.

Keywords: *client, client-side application, server, server-side application, cross-platform compatibility, React, REST API, JavaScript, Heroku, Postman.*

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface
CLI – Command-Line Interface
CPU – Central Processing Unit
CRUD – Create, Read, Update, Delete
CSS – Cascading Style Sheets
HTTP – Hypertext Transfer Protocol
HTTPS – Hypertext Transfer Protocol Secure
ID – Identification
IoT – Internet of things
ORM – Object–Relational Mapping
RAM – Random-Access Memory
REST – Search Engine Optimization
SEO – Representational State Transfer
SPA – Single-Page Application
PWA – Progressive Web App
SSR – Server-Side Rendering
TMDB – The Movie Database
UI – User Interface
UX – User Experience
SOAP – Simple Object Access Protocol
DSL – Domain-Specific Language
CMS – Content Management System

ВСТУП

Розробка програмного забезпечення зазнала значних змін у результаті великого попиту на вебзастосунки. Стрімкий розвиток технологій спричинив збільшення кількостей платформ, тож одним із ключових викликів, з яким стикаються розробники, стала необхідність створення застосунків, які можуть функціонувати на різних платформах. Така потреба зумовлює необхідність використання кросплатформних рішень, які дозволяють зменшити витрати часу та ресурсів на розробку та підтримку програмного забезпечення.

Одним з найбільш популярних та ефективних інструментів для створення застосунків стала бібліотека React, яка надає можливість створювати інтерфейси користувача, що легко адаптуються до різних платформ. Використання REST API дозволяє організувати ефективну взаємодію між клієнтською частиною застосунку та сервером, забезпечуючи гнучкість та масштабованість.

Актуальність теми роботи, а саме «Архітектура кросплатформного застосунку на базі React та REST API», як було зазначено вище, зумовлена швидким розвитком технологій та постійним зростанням попиту на кросплатформні застосунки, які використовують такі інструменти, як React та REST API. Ці технології дозволяють створювати високопродуктивні та легко адаптовані до різних платформ застосунки, що робить їх важливими інструментами в розробці програмного забезпечення сьогодення.

Застосунок такого роду являє собою взаємозв’язок клієнтського вебзастосунку на базі React з сервером і використанням прикладного програмного інтерфейсу REST. Таким чином архітектура застосунку містить лише дві складові поєднані інтерфейсом, дозволяючи створювати водночас як прості, так і комплексні кросплатформні застосунки шляхом додавання нових складових. Таке рішення дозволяє створювати масштабовані кросплатормні застосунки, що є ідеальним вибором для проєктів які швидко розвиваються.

До того ж, як серверна частина застосунку, так і клієнтська, мають свою внутрішню архітектуру. Для кожного застосунку архітектура формується у результаті специфіних потреб. Кількість потреб зазвичай збільшується по ходу розробки та масштабування.

Мета кваліфікаційної бакалаврської роботи полягає в розробці програмно-апаратного комплексу, дослідженні архітектури застосунку на базі React та REST API, аналізу взаємодії клієнтської частини застосунку з сервером при використанні прикладного програмного інтерфейсу REST, розгляду необхідних інструментів та засобів для дотримання кросплатформності.

Завдання:

– провести аналітичний огляд архітектури кросплатформного застосунку на базі React та REST API та визначити її складові;

– розглянути поширені архітектурні рішення для такого застосунку та на базі цих рішень спроєктувати архітектуру для програмної реалізації застосунку на базі React та REST API;

– розробити апаратну частину комплексу та перевірити апаратне забезпечення на стійкість до навантажень;

– забезпечити кросплатформність застосунку та протестувати цю властивість.

Об'єкт дослідження: Об'єктом дослідження виступає архітектура кросплатформного застосунку на базі React та REST API.

Предмет дослідження: Предметом дослідження є розробка архітектури кросплатформного застосунку застосунку на базі React та REST API для перегляду інформації про фільми, телешоу та людей, пов’язаних з медіапростором.

Практичне значення: Результати роботи можуть бути використані для проєктування та розробки кросплатформних застосунків на базі React та REST API.

1 АНАЛІТИЧНИЙ ОГЛЯД АРХІТЕКТУРИ КРОСПЛАТФОРМНОГО ЗАСТОСУНКУ НА БАЗІ REACT ТА REST API

Динамічність та адаптивність наразі є ключовими аспектами успішних застосунків, тож вибір правильної архітектури стає не тільки технічним рішенням, але й стратегічним вибором, який може визначити ефективність та масштабованість проекту. React, бібліотека для розробки користувацьких інтерфейсів, створена командою Meta [1], та REST API, стандарт обміну даними між системами, стали двома ключовими стовпами у сучасній сфері розробки вебзастосунків. Вони дозволяють розробникам створювати високоефективні і гнучкі застосунки, що працюють у реальному часі.

Комбінація React і REST API дозволяє реалізовувати різноманітні архітектурні рішення, які адаптуються під конкретні потреби проекту та бізнес-вимоги. Від односторінкових застосунків (SPA), які забезпечують швидку та плавну взаємодію з користувачем, до складних мікросервісних архітектур, які дозволяють незалежно масштабувати окремі компоненти системи. Існують також ізоморфні архітектури, які оптимізують початкове завантаження та покращують SEO, роблячи React-застосунки більш доступними для пошукових систем.

1.1 Складові архітектури кросплатформного застосунку на базі React та REST API

У контексті використання React та REST API, існують основні складові, які формують архітектурний каркас будь-якого застосунку на цій базі, а саме:

– **клієнтська частина застосунку:** клієнтська частина програми, реалізована на базі React відповідає за представлення інтерфейсу користувача та взаємодію з ним (UI/UX). Оскільки React дозволяє будувати компонентні ієрархії, архітектура клієнтської частини може включати такі аспекти, як становий менеджмент [2] (за допомогою Redux, MobX чи інших бібліотек),

роутинг (React Router) та інтеграцію з ресурсом (серверною частиною), наприклад через REST API. Якість клієнтської архітектури визначається забезпеченням швидкодії, зручністю використання та масштабованістю користувацького інтерфейсу;

– **серверна частина застосунку:** серверна частина відповідає за обробку запитів, надісланих клієнтом, керування даними, взаємодію з базами даних та іншими зовнішніми сервісами, а також за відповіді на запити клієнтської частини. Ця частина може бути реалізована у форматі монолітних додатків або з використанням мікросервісів [3] залежно від вимог до масштабованості та комплексності системи. У випадку застосування REST API важливо щоб сервер був здатний ефективно взаємодіяти з клієнтською частиною через добре налагоджені кінцеві точки;

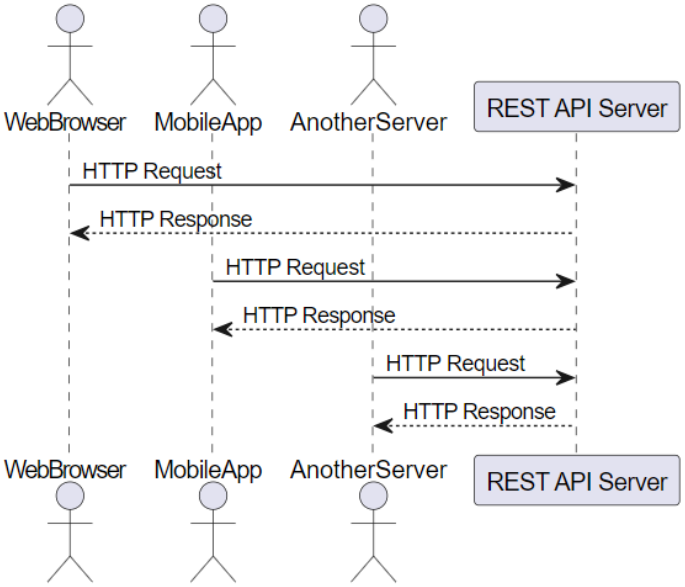


Рисунок 1.1 – Діаграма зв'язку клієнтських застосунків з REST API сервером

– **REST API:** REST API диктує ряд архітектурних обмежень [4], а саме єдність інтерфейсу, розділення на клієнтну та серверну частини, безстатусність, кешування, рівневу архітектуру системи зв'язку між клієнтом та сервером та один необов'язковий принцип – можливість серверної частини відповісти кодом замість статичних даних. Ці обмеження мають бути

враховані під час розробки як клієнтської, так і серверної частин застосунку, оскільки вони впливають на способи взаємодії між частинами застосунку та загальну архітектуру.

Детальне розуміння складових архітектури та взаємозв’язків – це критично важливий фактор для проєктування та реалізації ефективних, безпечних та легко масштабованих вебзастосунків.

1.2 Архітектурні принципи REST API

REST API встановлює обмеження, які регулюють взаємодію між клієнтською та серверною частинами застосунків. Ці обмеження спрямовані на оптимізацію взаємодії частин застосунків. Завдяки ним взємодія уніфікується і спрощуються розробка, підтрика і оновлення проєктів. Тож, у першу чергу слід розглянути якими принципами керується REST API, а вже потім досліджувати архітектури інших складових застосунку. Розглянемо кожен принцип:

1) **єдність інтерфейсу (uniform interface):** принцип єдності інтерфейсу стандартизує спосіб взаємодії з ресурсами (серверними частинами) системи. Це забезпечує простоту та зручність використання для різних клієнтів, оскільки інтерфейс залишається однаковим незалежно від платформи чи мови програмування. Слід зазначити, що саме цей принцип робить сервену частину застосунку кросплатформною:

   – приклад: HTTP-запити GET, POST, PUT та DELETE використовуються для взаємодії з ресурсами в інтернеті. Це стандартний інтерфейс, який дозволяє різним системам здійснювати звернення до веб-ресурсів;

2) **розділення на клієнтну та серверну частини (client-server decoupling):** принцип передбачає розділення системи на клієнтську та серверну частини, які взаємодіють між собою через мережу. Це дозволяє їм розвиватися незалежно один від одного та спрощує масштабування та підтримку системи:

– приклад: клієнтська частина додатка, така як мобільний додаток для читання новин, звертається до сервера, щоб отримати оновлені статті. Сервер в свою чергу виконує свої внутрішні процеси і повертає відповідь на запит клієнтській частині. Таким чином частини діють незалежно і мають зв’язок лише у форматі запит-відповідь;

3) **безстатусність (statelessness):** принцип безстатусності означає, що кожен запит має містити усі необхідні дані для його виконання. Тобто сервер не зберігає жодної інформації про стан клієнта між запитами, не створює жодних сесій. Це дозволяє підвищити масштабованість системи, оскільки сервер може виконувати більше запитів, а також спрощує кешування і підтримку, оскільки кожен запит обробляється незалежно:

– приклад: пошукова система Google, використовує безстатусність для обробки мільйонів запитів без необхідності зберігати інформацію про стан користувача;

4) **кешування (cacheability):** принцип кешування передбачає, що сервер повинен повідомляти клієнтів про можливість кешування відповідей. Це дозволяє клієнтам зберігати копії ресурсів та використовувати їх замість повторних запитів до сервера, що підвищує швидкодію та знижує навантаження на сервер:

– приклад: відповідь на запит може бути закешована у браузері на заданий проміжок часу. Застосовується, наприклад, для швидкого завантаження при наступних відвідуваннях веб-сайту;

5) **рівнева архітектура системи зв’язку між клієнтом та сервером (layered system architecture):** принцип рівневої системи передбачає, що клієнтська та серверна програми не завжди підключаються безпосередньо одна до одної. У комунікаційному контурі може бути кілька різних посередників, також ні клієнт, ні сервер не можуть визначити, чи він спілкується з кінцевою програмою чи посередником. Це забезпечує покращення масштабованості, безпеки та зручності процесу розробки:

– приклад: між React клієнтом та API може бути посередник у вигляді SSR застосунку;

6) **відповідь кодом на вимогу (code on demand):** REST API зазвичай надсилає відповідь у вигляді статичних даних, але в деяких випадках відповідь може містити код. У цих випадках код може бути виконаним лише за вимогою:

– приклад: вебсервер може надіслати JavaScript-код[5] для відображення віджетів або інтерактивних елементів на вебсторінці.

REST API встановлює правила взаємодії між клієнтською та серверною частинами застосунків, що оптимізує їх роботу. Завдяки стандартизації інтерфейсів, спрощується розробка, підтримка та оновлення проєктів. Принципи REST API забезпечують ефективну і надійну взаємодію між різними компонентами системи, це дозволяє застосункам залишатися кросплатформними, масштабованими та швидкодійними.

1.3 Поширені архітектурні рішення для побудови клієнтської частини застосунку на базі React

Клієнтська частина програми – це частина програмного забезпечення, яка виконується на боку користувача, зазвичай у веббраузері. Її головна роль полягає в тому, щоб надати користувачеві можливість взаємодії з програмою шляхом відображення інтерфейсу та обробки вводу. До основних завдань клієнтської частини входять: відображення інтерфейсу, зберігання та управління станом, взаємодія з серверною чатиною, логіка взаємодії та обробка подій.

Для ефективного виконання цих завдань архітектурний підхід до розробки клієнтської частини є критично важливим. Наразі існує безліч підходів до створення архітектури React застосунків. Розглянемо деякі найпоширеніші з них.

1.3.1 Компонентна архітектура клієнтської частини застосунку на базі React

Компонентна архітектура є одним з найпоширеніших підходів до розробки React-додатків і базується на ідеї розбиття програми на невеликі, самодостатні компоненти. Кожен компонент відповідає за свій конкретний функціонал або відображення частини користувацького інтерфейсу. Цьому архітектурному рішенню притаманні такі характеристики:

– **розділення на компоненти:** додаток розбивається на невеликі компоненти, кожен з яких відповідає за певний функціонал або відображення. Наприклад, кнопки, форми, таблиці, списки тощо;

– **повторне використання:** компоненти можуть бути повторно використані в різних частинах додатка або навіть в різних проектах;

– **композиція:** компоненти можуть бути об'єднані в більш складні структури шляхом їх композиції. Наприклад, компонент може містити в собі інші компоненти для створення складніших ієрархій;

– **властивості та стан:** компоненти можуть приймати дані від батьківських компонентів через властивості та управляти своїм станом. Це дозволяє динамічно змінювати відображення компонентів в залежності від вхідних даних або стану додатка.

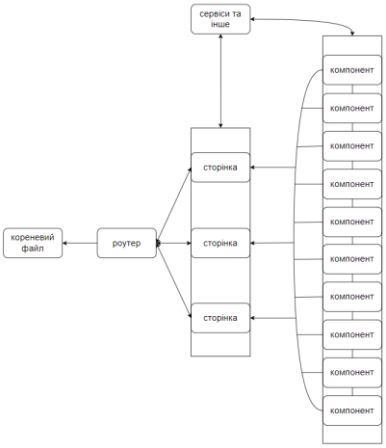


Рисунок 1.2 – Узагальнена схема компонентної архітектури React SPA застосунку

Переваг такої структури небагато, популярність зумовленна простотою реалізації та доцільністю використання у невеликих додатках. Для великих застосунків використання такого компонування недоцільне. Відсутня жорстка структура, тож у випадку великої кількості компонентів стає проблемою редагування будь-якого з них, адже це може викликати непередбачувані наслідки у інших компонентах та сторінках. Таким чином, доведеться створювати новий компонент кожного разу і в результаті нести величезні втрати в області оптимізації застосунку. Слід зазначити, що зв’язок з серверною частиною при такій архітектурі може відбуватись багатьма шляхами: завдяки сервісам, безпосередньо у компонентах, змішано. Це створює ще одну складність – підтримка та управління вкрай проблемні.

1.3.2 Атомна архітектура клієнтської частини застосунку на базі React

Атомна архітектура [6] в React базується на розбитті інтерфейсу на найменші будівельні блоки, які називаються «атомами». Ці атоми потім об'єднуються в більш складні структури, які називаються «молекулами» та «організмами». Основні складові атомної архітектури:

– **«атоми»:** «атоми» є найменшими будівельними блоками інтерфейсу. Це можуть бути прості елементи, такі як кнопки, поля вводу, текстові блоки, зображення тощо. Кожен «атом» має свої властивості та може бути використаний відразу в багатьох місцях додатка;

– **«молекули»: «**молекули» складаються з кількох атомів, які об'єднані для виконання певного функціоналу. Наприклад, заголовок разом з кнопкою, форма з полями вводу та кнопкою відправки, тощо. «Молекули» можуть включати один або більше атомів;

– **«організми»:** найбільші структури в атомній архітектурі, які складаються з кількох «молекул» та можуть представляти більш складні

частини інтерфейсу, такі як заголовки, футери, меню тощо. «Організми» можуть включати в себе «молекули» та «атоми».

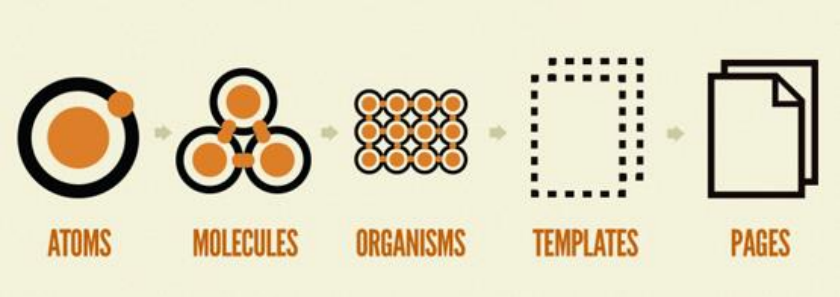


Рисунок 1.3 – Схема компонування атомної архітектури React застосунку

Переваги атомної архітектури:

- систематичність;
- модульність;
- можливість перевикористання;
- легкість рефакторингу та розширення додатку;
- піддатливість у командній роботі;
- легкість тестування та підтримки коду.

Недоліки атомної архітектури:

- початкова складність налаштування;
- збільшений розмір проєкту;
- складнощі з динамічною обробкою вмісту;
- складна в початковому опануванні.

Ця архітектура має чітке визначення та вимоги, тож такий вибір може бути хорошим варінтом для розробки великих та командних проектів. У випадку використання такої архітектури командою слід періодично проводити перегляд усього коду для збереження сталої структури.

1.3.3 Модульна архітектура клієнтської частини застосунку на базі React

Модульна архітектура [7] в React – це підхід до розробки, в якому складові додатку розділяються на окремі модулі. Кожен модуль відповідає за певну частину додатку і може містити в собі компоненти, ресурси, стилі, логіку та інше. Принципи модульної архітектури:

– **розділення на модулі:** додаток розділяється на логічні модулі або розділи, які відповідають за певні частини функціональності. Наприклад, модуль для авторизації, модуль для керування користувачами, модуль для відображення даних тощо;

– **інкапсуляція:** кожен модуль має бути незалежним та самодостатнім, тобто інкапсульованим. Це дозволяє спрощувати розробку, тестування та рефакторинг коду. Інкапсуляція забезпечує захист внутрішнього стану модулю від зовнішніх впливів, що допомагає уникнути небажаних побічних ефектів та сприяє більш надійній роботі системи;

– **вкладена компонентна структура:** кожен модуль може містити свою власну компонентну структуру, яка складається з компонентів React, які відповідають за відображення інтерфейсу модулю;

– **логіка та стилі:** кожен модуль може містити в собі свою власну логіку, обробники подій, стилі, шаблони та інші ресурси, які використовуються в цьому модулі. Це забезпечує високу модульність та ізоляцію коду, що полегшує підтримку та повторне використання компонентів. Використання CSS-модулів або styled-components дозволяє кожному модулю мати свої власні стилі, які не будуть конфліктувати з іншими частинами додатку.;

– **підтримка та масштабування:** модульна архітектура дозволяє легко підтримувати та розширювати вміст додатку, оскільки кожен модуль може бути розроблений та тестований окремо від інших. Це полегшує процес внесення змін та додавання нових функціональностей. Крім того, модульний

підхід сприяє кращій масштабованості додатку, оскільки можна легко додавати нові модулі без необхідності значних змін у вже існуючих.

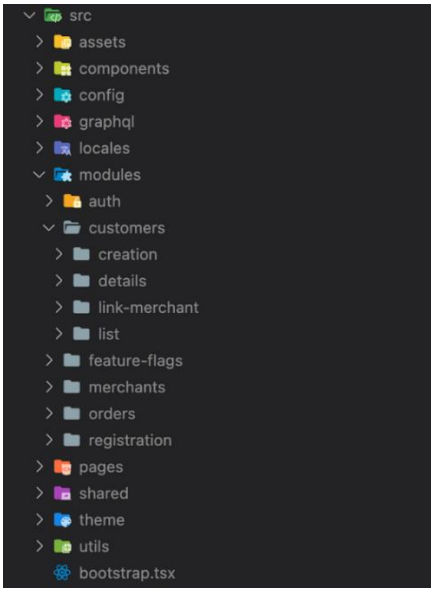


Рисунок 1.4 – Приклад загальної структури React проєкту за модульної архітектури

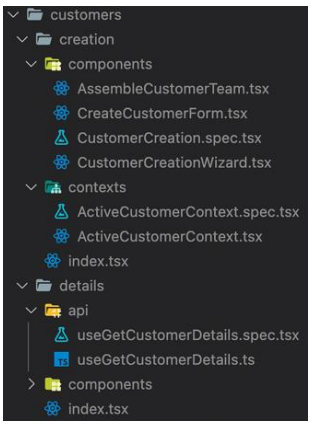


Рисунок 1.5 – Приклад структури окремого модуля

Переваги модульної архітектури:

- систематичність;
- інкапсуляція;
- можливість перевикористання;
- легкість рефакторингу та розширення додатку;
- піддатливість у командній роботі;

– легкість тестування та підтримки коду;
– можливість модифікацій.

Недоліки модульної архітектури:

– початкова складність налаштування;
– збільшений розмір проєкту;

Модульна архітектура в React передбачає розділення додатку на окремі модулі, кожен з яких відповідає за певну частину функціональності і містить компоненти, ресурси, стилі та логіку, чим спрощує тестування та підтримку коду. Ця архітектура дозволяє легше розширювати і модифікувати додаток, що особливо важливо для командної роботи. Проте, вона може вимагати значних зусиль на початкове налаштування і збільшувати розмір проєкту.

1.4 Поширені архітектурні рішення для побудови REST API серверу

Два найпоширені архітектурні підходи до побудови REST API серверу – це монолітний та мікросервісний. Кожна з цих архітектур має свої особливості, переваги та недоліки, і вибір між ними залежить від конкретних вимог до проєкту, його розміру та масштабів.

Монолітна архітектура є традиційним підходом, де весь функціонал додатку об'єднується в одному програмному коді та виконується як єдине ціле. У цьому підході всі компоненти, такі як база даних, бізнес-логіка та інтерфейс, розглядаються як частина одного додатку.

Натомість, у мікросервісній архітектурі додаток розділяється на невеликі та незалежні сервіси, які виконують певні функції або процеси. Кожен сервіс може бути реалізований, розгорнутий та масштабований незалежно від інших

1.4.1 Монолітна архітектура REST API серверу

При монолітному підході до побудови серверної частини додатка весь функціонал, включаючи бізнес-логіку, базу даних та інтерфейс, об'єднується в одному програмному коді та виконується як єдине ціле. Весь функціонал обробляється в рамках одного програмного додатку, який може бути розгорнутий на одному сервері або наборі серверів. Всі запити від клієнтів надсилаються до цього централізованого додатку, який відповідає на них та виконує необхідні операції. Характеристики монолітної архітектури REST API серверу:

– **централізований контроль:** усі компоненти та функціонал обробляються в межах одного додатку, що дозволяє забезпечити централізований контроль над усім функціоналом;

– **простота розробки та розгортання:** монолітна архітектура дозволяє швидко розробляти та розгортати додаток, оскільки всі компоненти знаходяться в одному місці;

– **складність масштабування:** монолітна архітектура може стати складною у масштабуванні, оскільки весь додаток має бути масштабований разом, навіть якщо лише деякі його частини потребують більшої потужності.

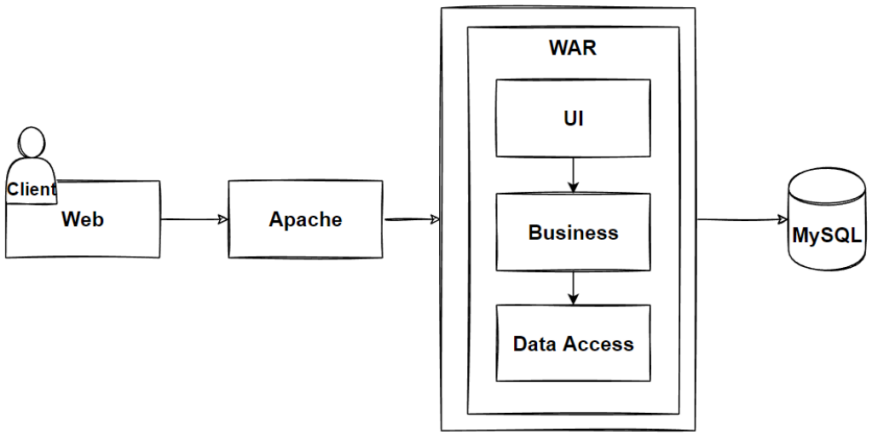


Рисунок 1.6 – Схема роботи монолітного серверу

Монолітна архітектура може бути корисною для невеликих або середніх проєктів з обмеженим обсягом функціоналу та потребами щодо масштабування. Однак для великих або складних проектів, де потрібно забезпечити високу доступність, масштабованість та розширюваність, тож мікросервісна архітектура в таких випадках буде доцільнішою.

1.4.2 Мікросервісна архітектура REST API серверу

При мікросервісному підході до побудови серверної частини додатка функціонал розділяється на невеликі та незалежні сервіси, які виконують конкретні функції або процеси. Кожен сервіс є окремим програмним компонентом, який виконує певну функцію та має власний набір API для взаємодії з іншими сервісами та клієнтами. Кожен сервіс може бути реалізований, розгорнутий та масштабований незалежно від інших. Характеристики мікросервісної архітектури REST API серверу:

– **розділення функціоналу:** функціонал додатку розділяється на невеликі та незалежні сервіси, що дозволяє зменшити залежності між компонентами та полегшити розробку, тестування та підтримку;

– **гнучкість та масштабованість:** кожен сервіс може бути масштабований окремо в залежності від навантаження, що дозволяє забезпечити гнучкість та ефективність у використанні ресурсів;

– **ізоляція помилок:** у разі виникнення помилок або витоку ресурсів у одному сервісі, інші сервіси можуть продовжувати працювати безперервно, що забезпечує вищу надійність та стабільність системи;

– **ізольованість оновлень та розгортання:** кожен сервіс може бути розгорнутий та оновлений незалежно від інших, що полегшує управління версіями та швидкість впровадження змін.

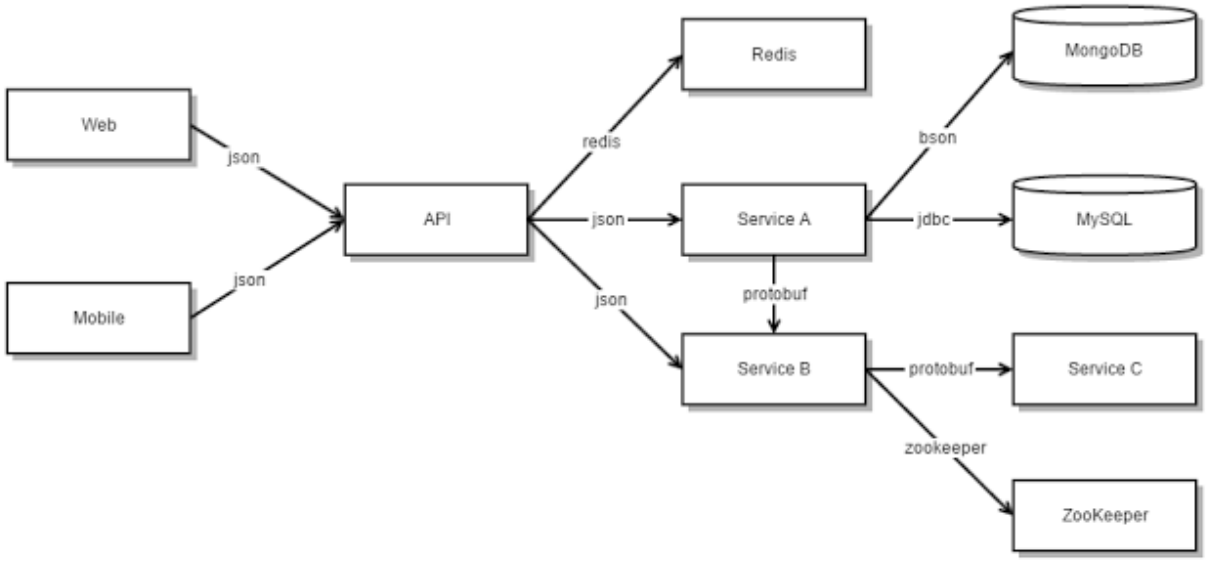


Рисунок 1.7 – Схема роботи мікросервісного серверу

Мікросервісна архітектура може бути особливо корисною для великих та складних проектів, де потрібно забезпечити високу доступність, масштабованість та розширюваність. Однак вона недоречна для реалізації менших проектів або там, де існує обмежений обсяг ресурсів.

При використанні мікросервісної архітектури з REST API кожен сервіс має власний набір API, який використовується для взаємодії з іншими сервісами та клієнтами. Це дозволяє забезпечити чітке розмежування відповідальності між сервісами та полегшує інтеграцію з різними клієнтами, такими як веб-додатки, мобільні додатки та інші сервери.

Однак мікросервісна архітектура не позбавлена недоліків. Вона може бути недоречною для реалізації менших проектів або там, де існує обмежений обсяг ресурсів. Це пов’язано з тим, що мікросервісна архітектура може вимагати більше часу на налаштування та управління інфраструктурою, а також складніших підходів до моніторингу та забезпечення безпеки.

Висновки до розділу

У результаті аналізу архітектури кросплатформного застосунку на базі React та REST API було визначено, що архітектура такого застосунку складається з трьох складових: клієнтська частина, серверна частина та REST API.

Було визначено за якими принципами визначається та функціонує REST API. Оскільки для взаємодії використовуються HTTP-запити, можна підсумувати, що серверний застосунок буде кросплатформним за замовчуванням.

У результаті розгляду поширених архітектурних рішень як для клієнтської, так і для северної частин застосунку стало можливим підсумувати, що для найкращого вибору архітектури слід розуміти масштаби та цілі створення тої, чи іншої частини застосунку кожна архітектура має свої переваги та недоліки.

2 ПРОГРАМНА ЧАСТИНА КОМПЛЕКСУ

У цьому розділі буде розглянуто проєктування архітектури React застосунку, програмна реалізація клієнтської частини комплексу з урахуванням спроєктованої архітектури, програмна реалізація взаємодії клієнтської частини комплексу з REST API, кінцеві точки TMDB API[8], програмна реалізація серверної частини комплексу(REST API).

2.1 Програмна реалізація клієнтської частини комплексу

Для програмної реалізації клієнтської частини комплексу було визначено вимоги до архітектури, підібрано архітектуру шо задовільняє цим вимогам, створено структуру проєкту згідно спроєктованої архітектури та її модифікаціям та описано програмну реалізацію зв’язку клієнтської частини комплексу з REST API.

2.1.1 Проєктування архітектури React застосунку

Першим чином було визначено вимоги до застосунку:

– **масштаб:** застосунок являє собою великий вебсервіс для перегляду величезної кількості різноманітної інформації. Таким чином прості архітектурні рішення недоцільні;

– **взаємодія з Rest API:** необхідно виділити конретні рішення щодо до обробки запитів та збереження і передачі станів;

– **можливість авторизації:** TMDB API надає можливість реалізації авторизації шляхом виділення токену. Цей токен використовується для запитів на ресурси для авторизованих користувачів і, згідно з принципами REST API, його необхідно передавати серверній частині із кожним новим запитом. Тож, необхідно створити рішення для запису та зберігання цього токену у пам’яті браузера а також шляху швидкого доступу до нього;

– **кросплатформність:** REST API забезпечує кросплатформний доступ до ресурсу, але для забезпечення кросплатформності клієнтської

частини слід використати каскадні таблиці стилів (CSS) та протестувати результат на різних платформах.

Згідно вимог, для реалізації клієнтської частини, було обрано модульну архітектуру з модифікаціями та використанням React ES6[9] і React хуків[10]. Модульна архітектура є ідеальним вибором для реалізації застосунків великих масштабів за рахунок інкапсуляції частин проєкту і дозволяє виносити сервіси у зовнішній простір, що вирішить питання запитів до API, збереження і передачі станів, запису та доступу до токена. Окрім цього, для забезпечення зручної роботи зі стилями було використано модульне рішення надане матамовою на основі CSS – SASS[11].

2.1.2 Створення структури проєкту React застосунку згідно спроєктованої архітектури

У результаті програмної реалізації прототипу проєкту було створено таку структуру:

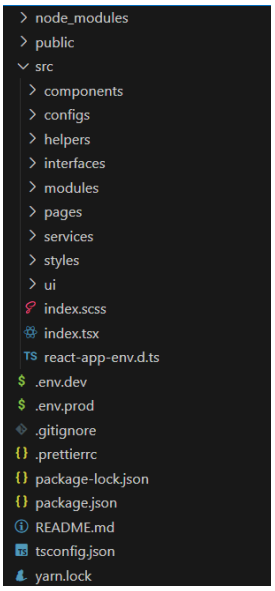


Рисунок 2.1 – Загальна структура прототипа проєкту React застосунку

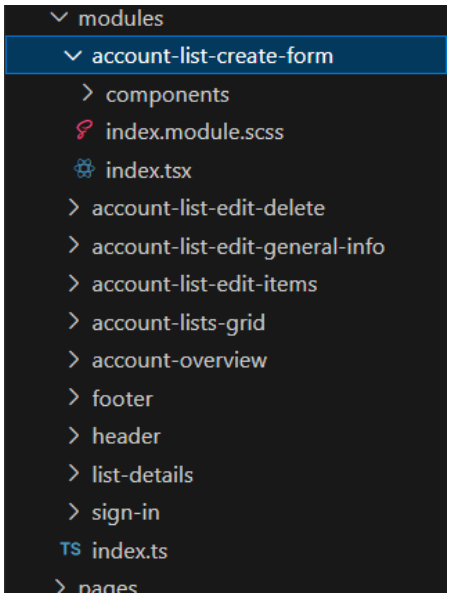


Рисунок 2.2 – Структура окремого модуля

Як можна побачити з рисунків, архітектура складається з наступних слоїв:

– **сторінки:** у найкращому вигляді має містити лише перелік модулів. Необхідно уникати будь-якої логіки на рівні сторінок. Має бути максимально незалежною структурою: при видалені сторінки, застосунок має продовжувати свою роботу. Програмна реалізація кореневої сторінки акаунту:

```
export const AccountPage: FC = () => {
  return (
    <div className={styles.wrapper}>
      <Menus />
      <div className={styles.outlet}>
        <Outlet />
      </div>
    </div>
  );
};
```

Рисунок 2.3 – Компонент кореневої сторінки акаунту *AccountPage*

– **модулі:** дуже незалежна структура, при видаленні, сторінка має продовжити працювати, зникне лише відповідний модуль. Інкапсульовує логіку та вкладені компоненти. Робота з провідниками глобального стану та

запити до API виконуються на цьому рівні. Модуль не може містити всередині себе інший модуль. Програмна реалізація модуля що представляє собою відображення списків створених користувачем:

```
export const AccountListsGrid: FC = () => {
  const { isSession, accountId } = useSession();
  const getPromiseForInfiniteScroll = (page: number) => {
    if (isSession && accountId) {
      return ListsPromises.getListsCollection(accountId, page);
    }
    return null;
  };
  const { data, isPending, error } = useInfiniteScroll<{},
IListGeneralInfo>(getPromiseForInfiniteScroll);
  if (error) return <ErrorBanner errorDescription={error} errorInfo="Error" />;
  if (data && isPending){
    return (
      <>
        <ListsCardsGrid lists={data.results} isPending={false} />
        <ScrollLoader />
      </>
    );
  return <ListsCardsGrid lists={data?.results} isPending={isPending} />;
};
```

Рисунок 2.4 – Модуль відображення сітки списків створених користувачем *AccountListsGrid*

– **компоненти:** інкапсульовані всередині модулів. Жодної бізнес логіки – вона має надходити з модуля. У більщості випадків використовують рівень UI у якості складових, впроваджуючи логіку, деталізацію та конкретику. У вийняткових випадках компоненти можуть знаходитись на зовнішньому рівні, не інкапсульовано всередині модуля. Такий підхід слід максимально уникати – з'являються неявні зв'язки. Але бувають випадки коли компонент необхідно винести окремо, наприклад модальне вікно що з'являється у кількох модулях. Це модальне вікно може викликатись в різних

модулях, при цьому всередині логіка всередині не змінюється, тож доцільно винести такий компонент на зовнішній рівень, а не створювати дублікат у кожному модулі. Програмна реалізація компоненту що був винесений на зовнішій рівень:

```
export const ModalLightboxPopup: FC<IModalLightboxPopupProps> = (props) => {
  const { content } = props;
  useEffect(() => {
    document.body.classList.add('scroll-freeze');
    return () => {
      document.body.classList.remove('scroll-freeze');
    };
  }, []);
  const onCloseButtonClick = () => {hideModalLightboxPopup();};
  return (
    <div className={styles.wrapper}>
      <div className={styles.contentWrapper}>
        <OnlyIconButton onClick={onCloseButtonClick} icon={<SvgCloseIcon />} />
          <div className={styles.content}>{content}</div>
      </div>
    </div>
  );
};
```

Рисунок 2.5 – Зовнішній компонент модульного вікна *ModalLightboxPopup*

– **UI:** ніякої логіки всередині, лише базові компоненти інтерфейсу що отримують властивості ззовні. Більшість стилів прописується на цьому рівні. Програмна реалізація UI елменту:

```
export const RoundAvatar: FC<IRoundAvatarProps> = (props) => {
  const { img, name } = props;
  const imageContent: string | JSX.Element =
    img ? img : name ? name.charAt(0) : '?';
  return <div className={styles.roundAvatar}>{imageContent}</div>;
};
```

Рисунок 2.6 – Компонент користувацького інтерфейсу *RoundAvatar*

Для вирішення питань запитів, збереження і передачі станів, запису та доступу до токена було створено відповідні сервіси. Зокрема, для збереження і передачі станів було використано бібліотеку Redux[12].

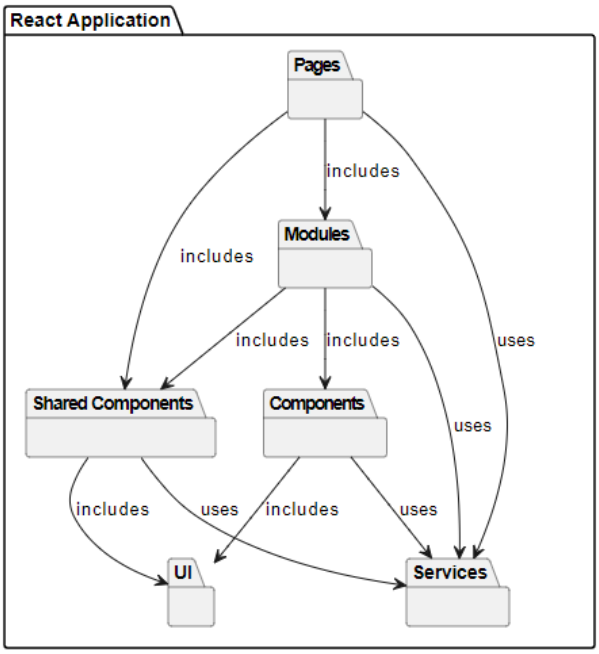


Рисунок 2.7 – Структурна діаграма проєкту React застосунку

Усе, окрім сервісів для збереження та зчитування даних з локального сховища браузера, і сервісів у яких збережені методи для асинхронних запитів до REST API серверу, було реалізовано функціональним програмуванням. Сервіси що стали виключенням були виконані у вигляді класів з двох причин:

– зручна організація однотипних методів;

– використовується кілька методів одного класу майже в один час.

До того ж, робота з цими методами лише іншими сервісами, недопускаючи зайву логіку у модулях.

2.1.3 Зв’язок клієнтського React застосунку з REST API сервером

Зв’язок клієнтського React застосунку з REST API сервером було розбито на декілька етапів, для кожного з яких було створено власні сервіси. Така організація необхідна для легкої підтримки, адже кожен з цих сервісів виконує свій вид роботи.

2.1.3.1 Перехоплювачі

Перехоплювачі – це всього екземпляри axios[13], які містять загальні конфігурації для багатьох запитів. Це лише кілька екземплярів, створених в одному місці та використовуються для багатьох запитів, щоб запобігти дублюванню шаблонного коду. Хоч і схоже що це щось статичне, але застосовуються змінні середовища, тож це схоже на конфігураційний етап для запитів до REST API. Приклад перехоплювача:

```
$apiV4.interceptors.request.use((config: any) => {
  config.headers.Authorization = `Bearer
    ${process.env.REACT_APP_TMDB_ACCESS_TOKEN}`;
  config.headers.accept = 'application/json';
  return config;
});
```

Рисунок 2.8 – Перехоплювач запитів *$apiV4*

Зазвичай він містить такі налаштування:

– Базовий шлях до REST API
– Токен доступу
– "config.headers.accept = 'application/json';"

Тож, перехоплювачі – це екземпляри з загальними конфігураціями для багатьох запитів, що запобігає дублюванню шаблонного коду. Це, по суті, конфігураційний етап для запитів до REST API, який спрощує та уніфікує їх налаштування.

2.1.3.2 Клас із методами для запитів до кінцевих точок

«Обіцянки» зберігаються в класах зі статичними методами. Кожен метод може використовувати певний перехоплювач залежно від типу запиту та заповнювати його додатковими даними/параметрами/заголовками, якщо потрібно. Кожен метод створений для окремої кінцевої точки і містить власні

налаштування, хоч структура у них майже ідентична. Приклад статичного методу, який повертає обіцянку:

```
static async getRequestToken():
  Promise<AxiosResponse<IGetRequestTokenResponse>> {
    return $apiV4.post<IGetRequestTokenResponse>(
      '/auth/request_token',
      {redirect_to: `${process.env.REACT_APP_URL_HOST}/sign-in`},
      {headers: { 'content-type': 'application/json' }}
    );
}
```

Рисунок 2.9 – Метод що повертає «обіцянку» *getRequestToken*

Тож, «обіцянки» зберігаються в класах зі статичними методами, де кожен метод може використовувати певний перехоплювач залежно від типу запиту та додавати необхідні дані чи заголовки. Кожен метод відповідає окремій кінцевій точці та містить власні налаштування, хоча структура методів майже ідентична.

2.1.3.3 Використання методів для запитів до кінцевих точок

Клас із запитами до ендпоентів може бути використаний у будь-якомі місці, хоча більшість запитів на етапі прототипу виконуються у інших сервісах.

Приклади використання запитів у модулях:

– запит на отримання проміжкового токену авторизації;
– запит на створення колекції авторизованим користувачем;
– запит на інформації про фільм;

Приклад запиту всередині модуля:

```
const data = await requestWithNotificationsAndPendingSetter(
  AuthPromises.getRequestToken(), setIsRequestTokenPending, false);
```

Рисунок 2.10 – Використання методу *getRequestToken* для отримання «обіцянки»

Приклади використання запитів у інших сервісах:

– запити для реалізації безкінцевого скролу вниз;

– запит на отримання повної інформації про користувача що авторизувався;

– запит на перевірку валідності токену;

Програмний приклад запиту всередині redux сервісу:

```
export const getAccountDetails = createAsyncThunk('user', async (sessionId:
string, { rejectWithValue }) => {
  try {
    const response = await AccountPromises.getAccountDetails(sessionId);
    const accountDetails = response.data;
    return accountDetails;
  } catch (error) {
    if (axios.isAxiosError(error)) {
      const message = error.message;
      if (message) return rejectWithValue(message);
      return rejectWithValue('Unexpected error occured');
    }
    const message = String(error);
    return rejectWithValue(message);
  }
});
```

Рисунок 2.11 – Запит *getAccountDetails* для отримання деталей акаунту всередині redux сервісу

Клас із запитами до ендпоінтів може використовуватися будь-де, хоча більшість запитів на етапі прототипування виконуються в інших сервісах. Приклади використання запитів у модулях включають отримання проміжкового токену авторизації, створення колекції авторизованим користувачем та отримання інформації про фільм. В інших сервісах запити використовуються для реалізації безкінечного скролу, отримання повної інформації про авторизованого користувача та перевірки валідності токену.

2.2 Кінцеві точки TMDB

TMDB API це надійний та потужний інструмент, що надає доступ до величезної кількості даних про фільми та телевізійні шоу. TMDB API надає можливість використовувати різноманітні дані про медіа, включаючи інформацію про акторів, рейтинги, відгуки та багато іншого, що зробить застосунок більш привабливим для користувачів та функціонально повним.

Для зручності використання та тестування TMDB API має власний веб-сервіс. Цей сервіс містить дві версії API, інструкції з використання, документацію до API та окремих кінцевих точнок, та історію змін.

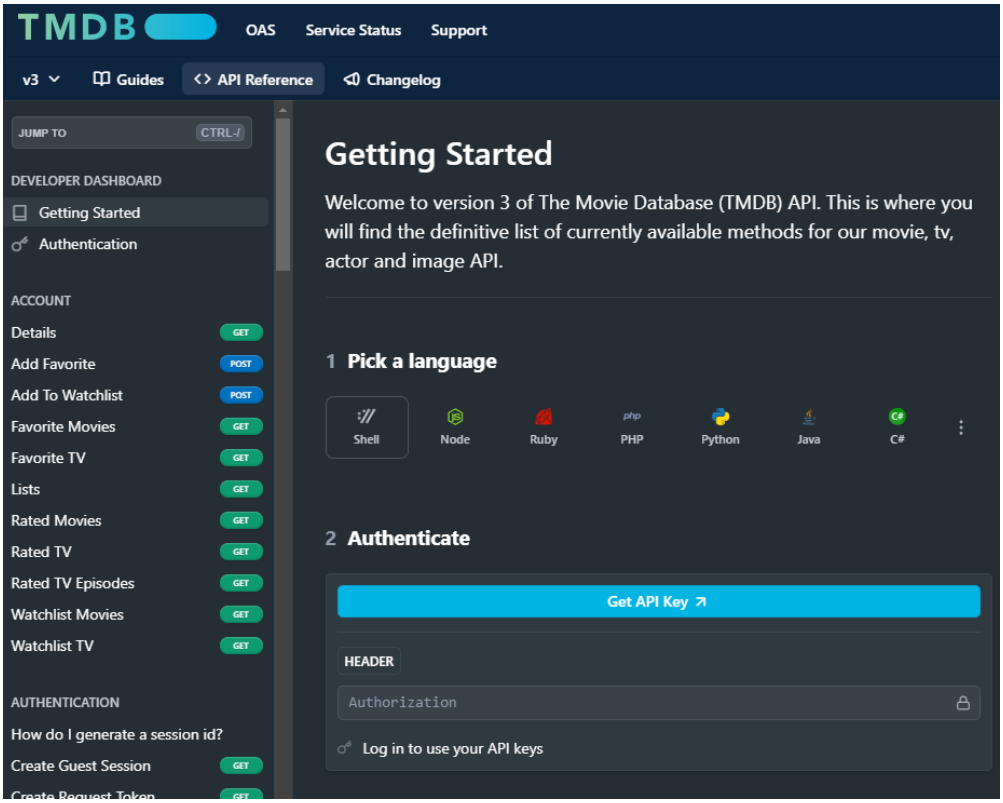


Рисунок 2.12 – Огляд документації TMDB API

Кожна кінцева точнка містить інструкції та можливість тестування на різних мовах програмування:

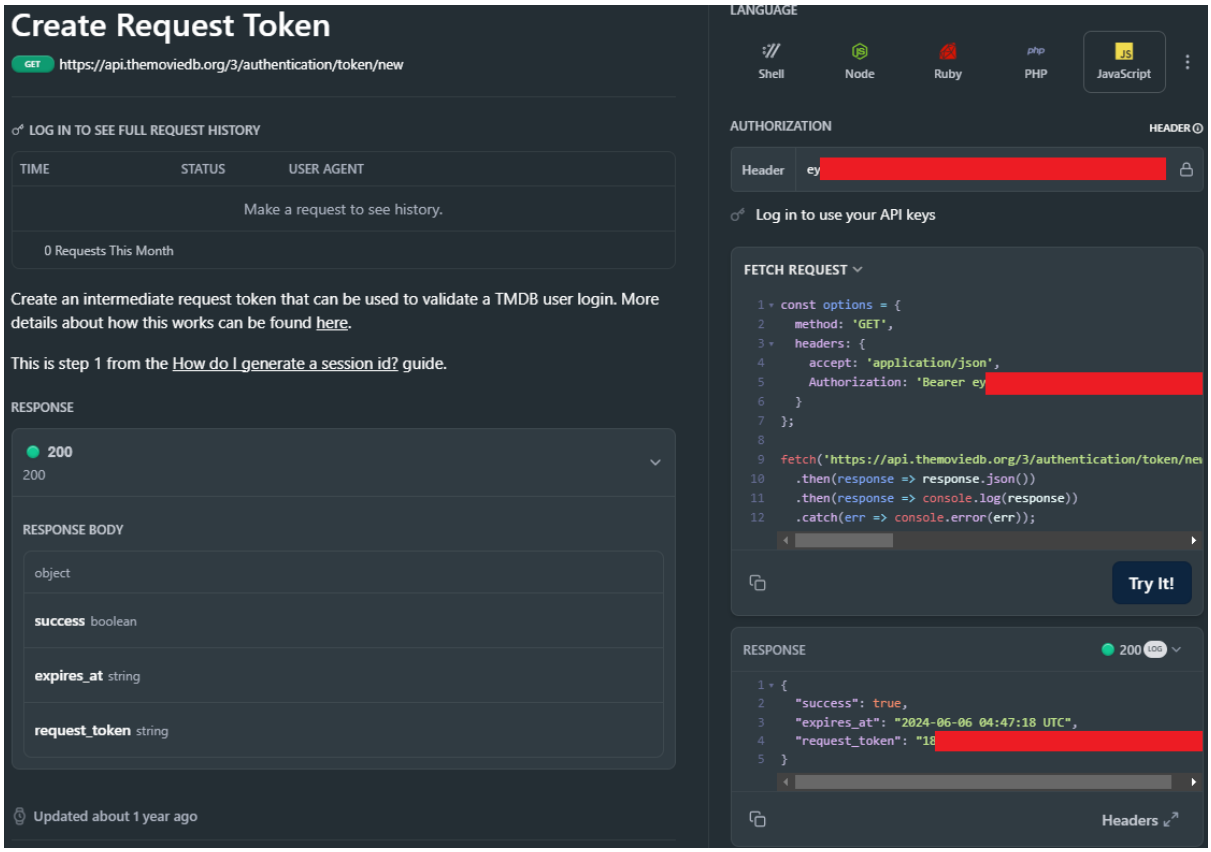


Рисунок 2.13 – Огляд та тестування конкретної кінцевої точки TMDB API

Оскільки у попередньому розділі було визначено принципи REST API, можливо визначити чи відповідає TMDB API цим критеріям:

1) **уніфікований інтерфейс:**

– TMDB API використовує стандартні HTTP-методи (GET, POST, DELETE тощо) для виконання операцій над ресурсами, такими як фільми, актори та облікові записи користувачів;

– кінцеві точки передбачувані та ресурсно-орієнтовані, наприклад, /movie/{movie_id} та /person/{person_id}, що відповідає принципам REST;

2) **розділення на клієнтну та серверну частини:**

– TMDB API дотримується цього принципу, дозволяючи клієнтам взаємодіяти з ресурсами через URI, не потребуючи розуміння серверної логіки. Клієнти використовують URI, такі як https://api.themoviedb.org/3/movie/{movie_id}, щоб отримати доступ до даних;

3) **безстанність:**

– кожен запит API TMDB містить всю необхідну інформацію, включаючи ключі API та параметри, для незалежної обробки запиту. Сервер не зберігає даних сеансів між запитами;

4) **кешованість:**

– TMDB API підтримує механізми кешування, надаючи відповідні заголовки HTTP. Це дозволяє клієнтам кешувати відповіді, де це можливо, що покращує продуктивність;

5) **шарувата система:**

– Дизайн API забезпечує використання проміжних вузлів, таких як балансувальники навантаження та проксі-сервери, без впливу на взаємодію запит-відповідь між клієнтом і сервером;

6) **Код на вимогу:**

– TMDB API зазвичай не включає цю функцію, що не є необхідним для відповідності стандартам REST.

З огляду на відповідність цим принципам, TMDB API дійсно можна вважати REST API.

2.3 Програмна реалізація серверної частини комплексу

Було програмно реалізовано серверний Node.js[14] додаток з використанням фреймворку NestJS[15]. Node.js – це платформо-незалежне середовище, яке дозволяє виконувати JavaScript-код на сервері. Його вибір зумовлений високою продуктивністю та можливістю масштабування, що особливо важливо для веб-застосунків з великим навантаженням.[16]

2.3.1 Проєктування архітектури серверного застосунку

NestJS – це прогресивний фреймворк для створення ефективних, надійних і масштабованих серверних додатків. Його переваги включають:

– **модульну архітектуру:** дозволяє розділяти код на модулі, що спрощує управління проектом;

– **вбудовану підтримку TypeScript:** забезпечує типізацію та допомагає зменшити кількість помилок у коді;

– **підтримку різних ORM:** спрощує роботу з базами даних.

Для репрезентації REST API було створено кілька простих ендпоінтів, які виконують наступні функції:

– створення нового запису відстеження кількості кліків за сесію;

– отримання всіх записів кількості кліків створених за усі сесії;

– отримання запису кількості кліків за ID сесії;

– збільшення кількості кліків для запису за ID;

– видалення запису кількості кліків сесії за ID та видалення самої сесії.

Ці ендпоінти можуть бути використані для збору аналітики активності користувачів на певних елементах клієнтської частини комплексу.

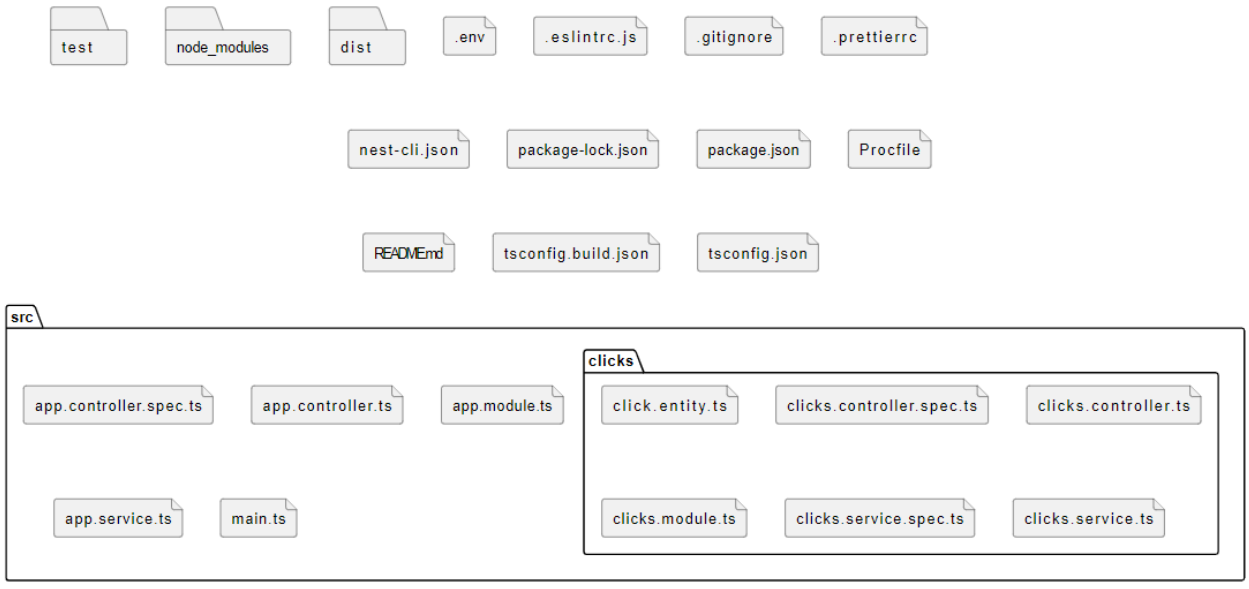


Рисунок 2.14 – Структура серверної частини комплексу

У якості архітектури було обрано монолітну структуру, що полегшує подальше розгортання застосунку. Монолітна архітектура значно легша для розгортання, оскільки всі компоненти системи зібрані в одному місці, що

спрощує управління кодом та конфігураціями. Вона вимагає менше налаштувань для взаємодії між компонентами, оскільки всі частини програми спілкуються безпосередньо. Це також зменшує складність налаштування інфраструктури, оскільки немає потреби налаштовувати окремі середовища для кожного сервісу, як у випадку з мікросервісною архітектурою. У результаті розгортання, оновлення та моніторинг додатку стають більш прямолінійними та менш ресурсомісткими. Для створення простого REST API такий вибір є однозначним.

2.3.2 Взаємодія серверної частини комплексу з локальною базою даних

Для локальної перевірки CRUD операцій було використано PostgreSQL [17]. PostgreSQL – це одна з найбільш потужних і надійних систем управління базами даних. Вона підтримує складні запити, транзакції та розширення, що робить її ідеальною для використання у великих проектах. PostgreSQL залишається актуальним завдяки таким факторам:

– **надійність:** PostgreSQL забезпечує високу надійність і стійкість до відмов;

– **масштабованість:** Легко масштабується для обробки великої кількості запитів і даних;

– **підтримка стандартів:** підтримує стандарт SQL [18] і дозволяє використовувати розширені функції.

Конфіденційні данні було збережено в файлі .env, який містить змінні середовища. Це дозволяє зберігати секретну інформацію, таку як дані для підключення до бази даних, окремо від коду [19].

Для управління базою даних PostgreSQL було використано графічний інтерфейс pgAdmin [20], який дозволяє легко створювати та управляти базами даних, таблицями, запитами та користувачами. Цей додаток забезпечує

зручний інтерфейс для роботи з базою даних, що спрощує адміністрування та управління даними.

Рисунок 2.15 – Структура сутності сгенерована в pgAdmin

Навіть з однією таблицею без зв'язків можливо створити різні ендпоінти, маніпулюючи запитами. Наприклад, можна отримати всі дані або лише одне поле за ID.

2.3.3 Локальне тестування серверної частини комплексу

Для тестування ендпоінтів було використано Postman. Postman – це інструмент для тестування REST API з зручним інтерфейсом, який дозволяє легко створювати, зберігати та виконувати HTTP-запити. Переваги використання застосунку Postman[21]:

– **не потрібно налаштовувати запити через браузер:** Postman дозволяє легко створювати та налаштовувати запити без необхідності використовувати незручний для тестування інтерфейс запитів браузеру;

– **зручний інтерфейс:** інтерфейс Postman інтуїтивно зрозумілий і зручний для користувачів.

– **можливість створення та збереження колекцій запитів:** Postman дозволяє створювати та зберігати колекції запитів для повторного використання;

– **можливість створення навантажувальних перевірок:** Postman дозволяє створювати штучне напруження на сервер, даючи змогу протестувати його стійкість.

– **можливість працювати в команді:** Postman дозволяє легко ділитися колекціями запитів з іншими членами команди.

Рисунок 2.16 – Колекція запитів до локальних кінцевих точок в Postman

Postman можна вважати клієнтом, адже він створює запити до сервера, хоч і в цілях тестування, тож діаграму взаємозв’язку можна представити як класичну схему зв’язку з REST API.

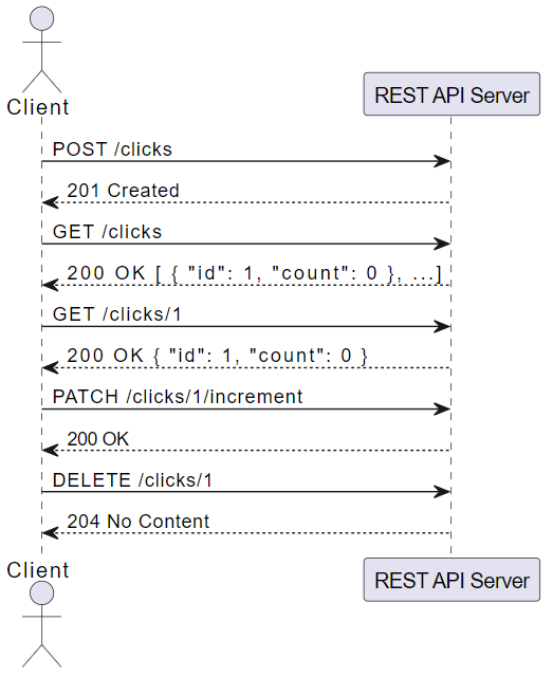


Рисунок 2.17 – Схема зв’язку серверної частини застосунку з клієнтською

Локальне тестування показало, що всі ендпоінти працюють коректно, дозволяючи виконувати CRUD-операції над даними в базі даних.

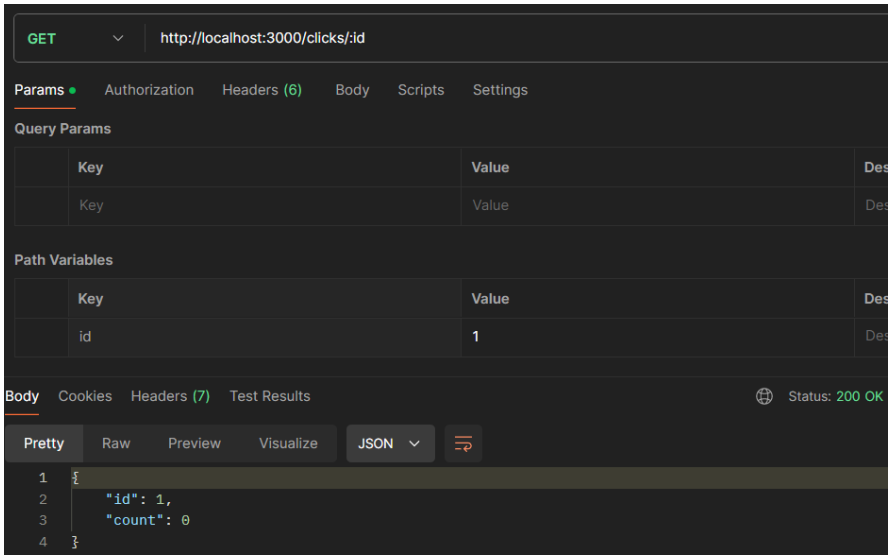


Рисунок 2.18 – Тестування запиту до налаштованої локальної кінцевої точки

Тож, використання Postman значно спростило процес завдяки своєму зручному інтерфейсу та можливості створювати і зберігати HTTP-запити. Використання Postman усуває необхідність налаштування запитів через браузер, забезпечуючи інтуїтивно зрозумілий інтерфейс для тестування. Локальне тестування підтвердило коректність роботи всіх ендпоінтів, дозволяючи виконувати CRUD-операції над даними в базі даних:

Окрім використання Postman, існують інші популярні інструменти та підходи для тестування REST API. Кожен з них має свої переваги та особливості, які можуть бути корисними в різних ситуаціях:

1) **Swagger**: інструмент для документування та тестування REST API, який дозволяє створювати інтерактивну документацію, де користувачі можуть безпосередньо взаємодіяти з API. Характеристики:

– інтерактивна документація: дозволяє легко тестувати API через веб-інтерфейс;

– генерація коду: можна генерувати клієнтський і серверний код на основі опису API;

– відмінна підтримка для створення документації та автоматизації тестування;

2) **JUnit з RestAssured:** JUnit разом з RestAssured дозволяє розробникам писати автоматизовані тест-кейси для REST API на Java. RestAssured пропонує зручний синтаксис для формування запитів та перевірки відповідей. Характеристики:

– автоматизація тестування: можливість інтеграції з CI/CD процесами;

– детальна перевірка відповідей: дозволяє писати складні перевірки для відповідей серверу;

– підтримка методів HTTP: GET, POST, PUT, DELETE та інших;

3) **SoapUI:** інструмент для тестування веб-сервісів, який підтримує як SOAP, так і REST API. Він дозволяє створювати складні тестові сценарії, включаючи функціональні та навантажувальні тести. Характеристики:

– підтримка складних сценаріїв тестування: можливість створювати і виконувати складні тести;

– інтеграція з іншими інструментами: наприклад, з Jenkins для автоматизації тестування;

– потужний інтерфейс для роботи з різними типами веб-сервісів;

4) **Newman:** командний інструмент для запуску колекцій запитів Postman. Він дозволяє інтегрувати тести, створені в Postman, в CI/CD пайплайни. Характеристики:

– інтеграція з CI/CD: можливість автоматичного виконання тестів під час кожного коміту або злиття змін;

– використання вже створених колекцій Postman: не потрібно переписувати тести для іншого інструменту;

– висока швидкість виконання тестів;

5) **Insomnia:** інструмент для тестування API, який надає зручний інтерфейс для створення, збереження та виконання HTTP-запитів. Характеристики:

– зручний інтерфейс: схожий на Postman, але з додатковими функціями.

– підтримка GraphQL: можливість тестування як REST, так і GraphQL API.

– підтримка плагінів: розширення функціональності через встановлення додаткових плагінів.

6) **Karate:** фреймворк для тестування API на основі DSL, який дозволяє писати тести у вигляді зрозумілих сценаріїв. Він підтримує тестування REST API та інтеграційні тести. Характеристики:

– прості та зрозумілі сценарії: можливість писати тести у вигляді бізнес-логіки.

– інтеграція з JUnit: дозволяє запускати тести з популярними інструментами для тестування.

– підтримка мікросервісів: дозволяє тестувати взаємодію між різними сервісами.

Кожен з цих інструментів має свої переваги та підходить для різних завдань. Вибір інструменту залежить від конкретних вимог проекту, наявних ресурсів та потреб команди розробників. Незалежно від вибору інструменту, важливо забезпечити належне покриття тестами для всіх важливих ендпоінтів API, щоб забезпечити надійність та стабільність роботи системи.

.

Висновки до розділу

На базі проаналізованих архітектур клієнтської частини застосунку було було спроєктовано власну, модульну архітектуру, що була модифікована з урахуванням протреб при програмній реалізації.

Спроєктована ахітектура має чіткі ієрархічні зв’зки з визначеним порядком взаємодії. Тож, така архітектура може бути застосована для подального масштабування застосунку та перетворення його із прототипу у функціонуючий сервіс. З урахуванням створеної архітектури було скомпоновано та чітко визначено область відповідальну за взаємодію з REST API сервіси.

Для реалізації зв’язку з REST API було використано існуючий сервіс «The Movie Database». Було детально розглянуто кінцеві точки та доведено що цей сервіс надає сервер який повністю відповідає принципам REST API. Таким чином було отримано доступ до великої кількості різноманітних кінцевих точок, що дозволило вільно реалізовувати клієнтську частину застосуну.

У якості серверного застосунку було реалізовано прототип який містить кілька кінцевих точок. Головною метою цього прототипу є подальше розгортання для доступності на багатьох платформах, тож спочатку було проведено тестування роботи серверного застосунку на локальному рівні.

Локальне тестування прототипу серверної частини застосунку було успішним, показавши що ця частина застосунку реагує на запити та без сумніву являється REST AP. Таким чином, подальшим кроком неодмінно стане її розгортання, тобто реалізація апаратної частини комплексу, та перевірка на кросплатформність.

3 АПАРАТНА ЧАСТИНА КОМПЛЕКСУ

Для розробки апаратної частини комплексу було використано серверну частину, а саме для розгортання на апаратному забезпеченні. Для цього було використано сервіс Heroku [22], який забезпечує розгортання на виділених сервісом потужностях. Heroku – це платформа, яка спрощує розгортання, управління та масштабування додатків. Вона надає зручні інтерфейси налаштувань і взаємодії, що дозволяють легко розгортати додатки, не турбуючись про управління серверною інфраструктурою.

Heroku використовує хмарні сервіси Amazon Web Services [23] для забезпечення своїх інфраструктурних потужностей, надаючи при цьому власний інтерфейс для спрощення роботи з розгортанням. Крім того, Heroku здатний надавати метрики використання ресурсів апаратного забезпечення, що дозволяє знімати показники навантаження, аналізувати продуктивність додатку, та планувати зміну потужностей.

3.1 Процес розгортання серверого застосунку на платформі Heroku

Розгортання серверного застосунку на платформі Heroku є ключовим етапом у процесі створення програмного комплексу. Heroku забезпечує зручне та надійне середовище для розгортання, управління та масштабування веб-додатків. У цьому розділі буде розглянуто детальні кроки для успішного розгортання серверного застосунку, включаючи підготовчі заходи, процес налаштування, розгортання та моніторинг.

3.1.1 Підготовка середовища розробки

Перед розгортанням серверного застосунку на Heroku, необхідно виконати кілька підготовчих кроків:

– **інсталювання Heroku CLI:** Heroku CLI необхідний для керування додатками на Heroku з командного рядка. Його можна завантажити та інсталювати з офіційного сайту Heroku;

– **реєстрація та аутентифікація:** після інсталювання Heroku CLI потрібно зареєструвати обліковий запис на Heroku та виконати вхід у систему через командний рядок за допомогою команди "heroku login".

Підготовка середовища розробки – це найперший етап для забезпечення успішного розгортання та стабільної роботи додатку. Цей етап включає встановлення необхідних інструментів та налаштування облікових записів. Підготовка середовища дозволяє розробникам автоматизувати процеси розгортання та уникати можливих помилок.

3.1.2 Підготовка конфігураційних файлів

Для підготовки додатку до розгортання на Heroku потрібно переконатися, що проект належним чином налаштований. Для цього слід створити та налаштувати наступні файли:

– **файл "*package.json*":** необхідно переконатись, що у файлі "package.json" вказано скрипти для запуску та розгортання додатку:

```
"scripts": {
  "start": "node dist/main.js",
  "build": "nest build",
  "heroku-postbuild": "npm install && npm run build"
}
```

Рисунок 3.1 – Скрипти запуску та розгортання серверного застосунку

– **файл "*Procfile*":** необхідно створити у кореневій директорії проекту файл "Procfile", який визначає команди, необхідні для запуску вашого додатку на Heroku. Наприклад:

```
web: npm run start
```

Рисунок 3.2 – Налаштування файлу "*Procfile*"

– **файл “*.env*”:** потрібно переконатись, що конфіденційні дані (такі як ключі API або паролі до бази даних) зберігаються у файлі “*.env*”, який не слід додавати до системи контролю версій. Приклад структури файлу:

```
DB_HOST=host.com
DB_PORT=5432
DB_USERNAME=username
DB_PASSWORD=password
DB_NAME=database
```

Рисунок 3.3 – Приклад структури файлу “*.env*”

Підготовка конфігураційних файлів є важливим кроком у процесі розгортання, оскільки вони визначають, як додаток буде запускатися, взаємодіяти з іншими сервісами та зберігати конфіденційну інформацію. Коректна підготовка конфігураційних файлів забезпечує стабільну роботу додатку та його безпеку.

3.1.3 Розгортання додатку на сервісі Heroku

Для розгртання застосунку на платформі Heroku необхідно спочатку створити репозиторій на самій платформі, під’єднатись до нього та завантажити свій локальний застосунок на цей репозиторій:

– **створення репозиторію:** команда “heroku create” створює нову порожню програму на Heroku разом із пов’язаним порожнім репозиторієм Git. Якщо ця команда виконується з кореневого каталогу програми, порожнє сховище Heroku Git автоматично під’єднується як віддалене для локального сховища;

– **розгортання:** щоб розгорнути свою програму на Heroku, необхідно скористатись командою “git push”, щоб перемістити код із головної гілки локального сховища на віддалений пристрій Heroku.

У разі виникнення будь-яких помилок під час розгортання Heroku CLI надасть детальний звіт з причини помилок та можливих рішень. До того ж,

платформа завжди зберігає логи, відкриваючи цим зручний шлях для ідентифікації проблем.

3.2 Тестування навантаження на апаратне забезпечення

При розгортанні серверного застосунку необхідно враховувати його потенційну навантаженість. Регулярний збір та аналіз метричних даних навантаженості критично важливі для забезпечення стійкої роботи серверу та запобігання проблемам продуктивності.

Heroku надає вбудовані інструменти для моніторингу продуктивності додатку. Таким чином, можливо протестувати стікість апартаного забезпечення до навантажень, отримавши метрики використання ресурсів, такі як використання RAM, швидкість відповіді на запит, співвідношення відхилених запитів та виконаних та використання CPU.

Іншим інструментом для тестування стійкості серверу до навантажень слугує Postman. Цей додаток дозволяє створювати штучне навантаження шляхом послідовних та одночасних циклів запитів прямо в налаштованій колекції.

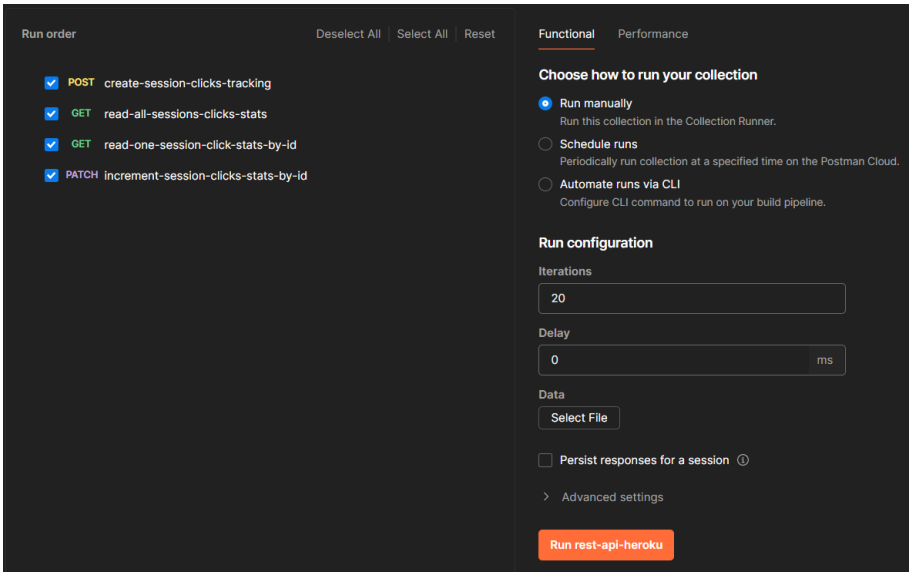


Рисунок 3.4 – Налаштування тесту на стійкість до навантажень серверу

Вказавши 0 мілісекунд затрики між циклами запитів можливо протестувати невелике навантаження на сервер, адже таким чином буде запущено 20 циклів по 4 запити кожний, без затримки між циклами.

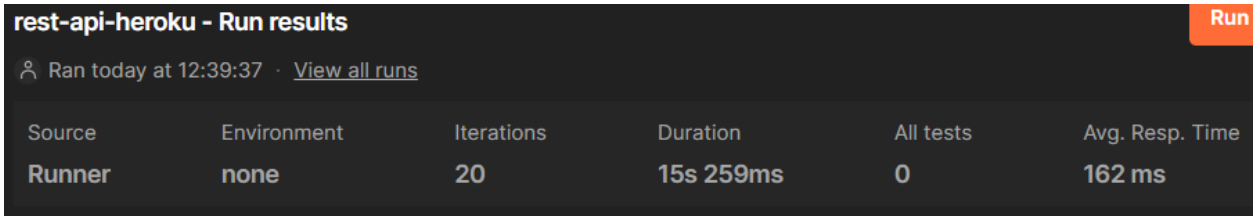


Рисунок 3.5 – Результати запуску циклів запитів в Postman

Як показав звіт в Postman, усього було виконано усі 20 циклів за приблизно 15.3 секунди, при тому середній час відповіді складав 162 мілісекунди. Такий показник затримки доволі звичайний для звичайного серверу, але враховуючи те, що навантаження було невелике, такого програмного забезпечення буде замало для розгортання промислового серверного застосунку.

До того ж, близькі до цих даних, але не ідентичні, були і метрики з платформи Heroku: час відповіді досягав 135 мілісекунд.

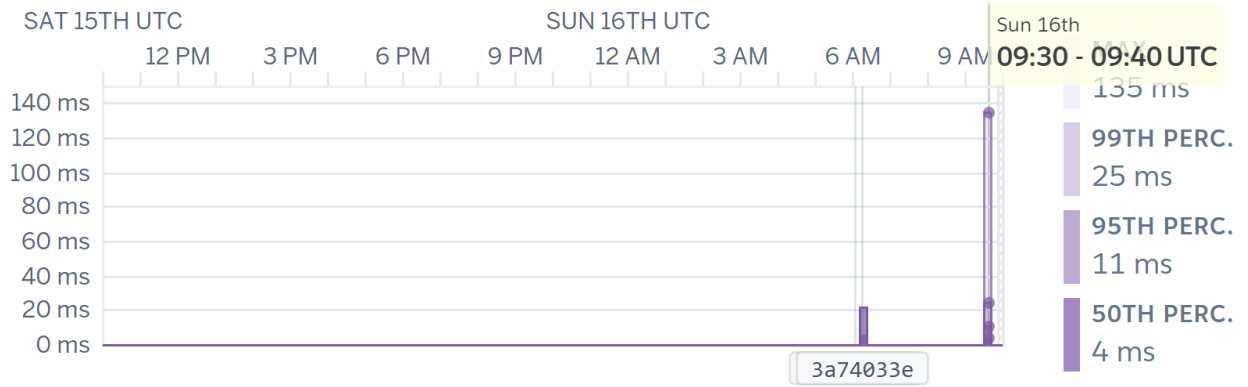


Рисунок 3.6 – Метрики часу відповіді на запит зафіксовані під час навантажувального випробування Heroku

Хоч запитів була і не мізерна кількість, але усі вони були послідовні, тож додаткове навантаження на RAM було відсутнє.

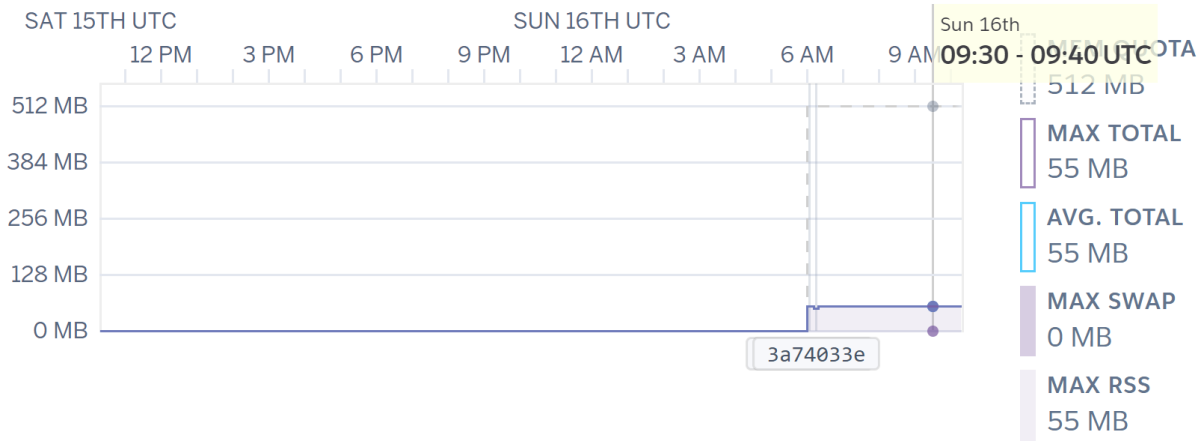


Рисунок 3.7 – Метрики використання RAM зафіксовані під час навантажувального випробування Heroku

Навантаження на CPU було відсутнє взагалі.

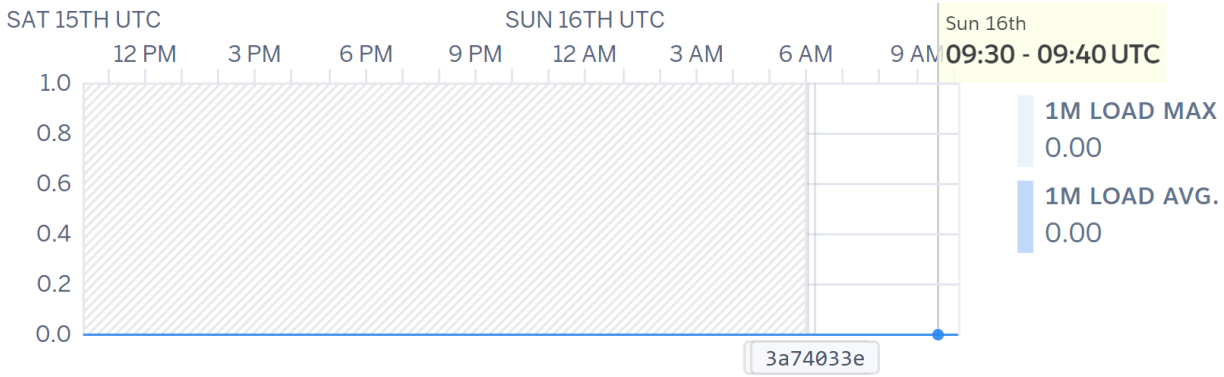


Рисунок 3.8 – Метрики використання CPU зафіксовані під час навантажувального випробування Heroku

Таким чином можемо підсумувати, що дане програмне забезпечення здатне витримувати навантаження навіть середнього рівня за умови що такий час відповіді на запити прийнятний.

Висновки до розділу

За результатом програмної реалізації серверної частини застосунку була розпочата робота над реалізацією апаратної частини комплексу. Визначивши потреби до апаратної частини комплексу, було вирішено використати сервіс Heroku, адже цей сервіс надає можливість розгорнути серверний застосунок та відстужувати навантаження на апаратне забезпечення. Таким чином стало можливо визначити стійкість апаратної частини комплексу до навантажень.

Розгрорувши серверний застосунок за допомогою сервісу Heroku, першим чином, було протестовано працездатність кінцевих точок розгорнутого REST API за допомогою застосунку Postman. Результати тестування кінцевих точок показали що проблем з розгорнутим сервером не виникло.

Останнім кроком стало тестування апаратного забезпечення на стійкість до навантажень. Було виконано 20 циклів запитів на сервер без затримки. За результатами тестування було зроблено висновок про стійкість апаратного забезпечення до навантажень може витримувати середні навантаження без критичних помилок та затримок.

4 ЗАБЕЗПЕЧЕННЯ КРОСПЛАТФОРМНОСТІ ЗАСТОСУНКУ

У цьому розділі буде розглянуто інструменти та засоби забезпечення кросплатформності клієнтської частини комплексу та перевірено цю властивість як у клієнтської, так і у серверної частини комплексу.

4.1 Перевірка кросплатформності серверної частини застосунку

Серверний застосунок, розроблений на базі Node.js з використанням фреймворку NestJS – це однозначно кросплатформне рішення. Сервер здатний обслуговувати запити з різних платформ і клієнтів, незалежно від їхнього типу і операційної системи. Кросплатформність в контексті веб-застосунків – це вкрай важлива характеристика, оскільки вона забезпечує гнучкість і широку сумісність з різними пристроями і технологіями.[27]

Серверний застосунок використовує стандартні протоколи HTTP/HTTPS для обміну даними з клієнтами. Це означає, що будь-який клієнт, здатний надсилати HTTP-запити і обробляти HTTP-відповіді, може взаємодіяти з сервером. Такими клієнтами можуть бути веб-браузери, мобільні додатки, інші сервери, IoT-пристрої тощо.

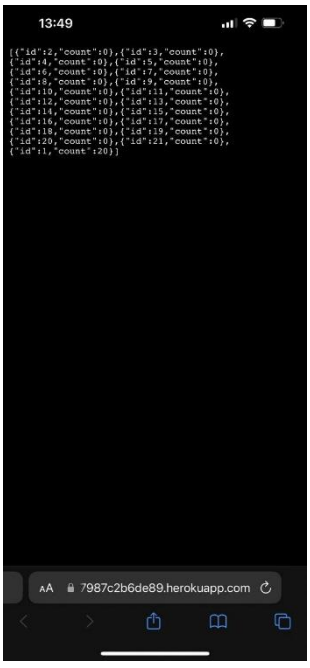


Рисунок 4.1 – HTTP-запиту на сервер виконаний на мобільному девайсі

Для перевірки кросплатформності існують і інші способи перевірки:

– **docker:** docker дозволяє створювати ізольовані середовища для запуску додатків. Це зручний інструмент для тестування застосунку на різних операційних системах без необхідності фізично встановлювати ці ОС;

– **віртуальні машини:** використання віртуальних машин дозволяє створити повністю ізольовані середовища з різними операційними системами.

– **cloud-сервіси:** розгортання додатку на різних хмарних платформах дозволяє перевірити його роботу в реальних умовах на різних інфраструктурах;

– **локальні середовища:** тестування на локальних машинах з різними ОС дозволяє швидко виявити специфічні проблеми, пов'язані з певною операційною системою:

– **CI/CD системи:** системи безперервної інтеграції та доставки дозволяють автоматизувати процеси тестування на різних платформах:

– **різні СУБД:** перевірка роботи додатку з різними системами управління базами даних дозволяє впевнитися в його гнучкості та сумісності.

Таким чином, серверний застосунок, розроблений на базі Node.js з використанням фреймворку NestJS, є кросплатформним рішенням, здатним обслуговувати запити з різних платформ і клієнтів незалежно від їх типу та операційної системи. Кросплатформність забезпечує гнучкість і широку сумісність з різними пристроями та технологіями, що є важливою характеристикою веб-застосунків. Використання стандартних протоколів HTTP/HTTPS дозволяє будь-якому клієнту, який може надсилати HTTP-запити і обробляти HTTP-відповіді, взаємодіяти з сервером, включаючи веб-браузери, мобільні додатки, інші сервери та IoT-пристрої.

4.2 Забезпечення кросплатформності клієнтської частини застосунку

В основі React.js полягає філософія розробки на основі компонентів. Ця філософія гармонійно перегукується з принципами кросплатформенної розробки, оскільки компоненти можуть бути створені так, щоб вони не залежали від платформи, сприяючи багаторазовому використанню коду та послідовностей. Віртуальний DOM забезпечує ефективне оновлення та рендеринг, а його декларативний синтаксис спрощує розробку інтерфейсу користувача. Ці атрибути роблять React ідеальним кандидатом для створення крос-платформних програм, які забезпечують зручний та привабливий користувацький інтерфейс.

Кросплатформна розробка з React знаходить свою основу в багатьох випадках використання. Стартапи з обмеженими ресурсами можуть використовувати потужність єдиної кодової бази для націлювання на різноманітну аудиторію користувачів на різних платформах. Підприємства, яким потрібна налагоджена взаємодія з користувачем, можуть забезпечити одноманітність зовнішнього вигляду своїх додатків, незалежно від того, доступ до них здійснюється через веб-браузери, мобільні пристрої чи встановлення на робочому столі.

Рисунок 4.2 – Приклад використання однотипних компонентів інтерфейсу

Крім того, навчальні заклади, засоби масової інформації, підприємства електронної комерції та розважальні платформи можуть отримати вигоду від кросплатформних додатків, які відповідають уподобанням користувачів і вибраним пристроям.

React уможливлює різноманітні кросплатформні проєкти завдяки своїм унікальним функціям:

– **інтерактивність інтерфейсу користувача (UI):** архітектура React на основі компонентів і ефективне відтворення роблять його чудовим вибором для додатків, яким потрібен динамічний та інтерактивний інтерфейс користувача. Він відмінно справляється зі складними компонентами інтерфейсу користувача та керуванням станом;

– **односторінкові програми (SPA):** React часто використовується для створення SPA, де вся програма завантажується на одній сторінці, а вміст динамічно оновлюється, коли користувачі взаємодіють з нею. SPA виграють від здатності React ефективно керувати оновленнями інтерфейсу без перезавантаження повної сторінки;

– **прогресивні веб-програми (PWA):** React можна використовувати для створення PWA, тобто веб-програм, які пропонують нативний досвід, подібний до додатків, включаючи офлайн-можливості, push-сповіщення та адаптивний дизайн. React у поєднанні з сервіснами та іншими бібліотеками може допомогти ефективно створювати PWA;

– **оновлення в режимі реального часу:** React є хорошим вибором для створення додатків у режимі реального часу, які потребують постійного оновлення, таких як додатки для чату, інформаційні панелі та інструменти для спільної роботи. Його віртуальний DOM і ефективне відтворення роблять його придатним для обробки регулярних оновлень;

– **платформи електронної комерції:** React можна використовувати для створення вебсайтів і платформ електронної комерції, де каталоги

продуктів, фільтрація та кошики для покупок вимагають динамічних оновлень і плавної взаємодії з користувачем;

– **системи керування вмістом (CMS):** React можна інтегрувати в системи CMS для створення власних інтерфейсів адміністратора з адаптивними та інтерактивними компонентами для керування вмістом і даними;

– **інструменти візуалізації даних:** фреймворк підтримує комплексну візуалізацію даних, важливу функцію для інструментів аналітики та бізнес-аналітики;

– **корпоративні програми:** стабільність React і можливість повторного використання компонентів ідеально відповідають потребам великих корпоративних програм;

– **вебсайти зручні для SEO:** хоча React сам по собі не покращує SEO, його можна використовувати в поєднанні з рішеннями для рендеринга на стороні сервера (SSR), такими як Next.js, для створення веб-сайтів, які пошукові системи можуть сканувати та ефективно індексувати;

– **можливість розробки у великій групі розробників:** структура проєктів React і чіткий розподіл завдань роблять його хорошим вибором для проєктів з великими командами розробників, де різні розробники можуть працювати над ізольованими компонентами, не заважаючи один одному.

Для забезпечення кросплатформності клієнтської частини розробленого застосунку було програмно реалізовано гумову розмітку за модулів SASS та принципу «Mobile First»[28]. Переваги використання модулів SASS:

– **організація стилів:** модулі SASS дозволяють легко організувати та управляти стилями застосунку. Вони дозволяють розділити стилі на логічні блоки, такі як загальні стилі, компоненти, макети тощо, що полегшує їхнє редагування та підтримку;

– **модульність:** завдяки модульності SASS, кожен компонент або модуль може мати свої власні стилі, які легко зберігати та використовувати.

Це дозволяє підтримувати чистоту коду та уникати конфліктів між стилями різних частин застосунку;

– **перевикористання стилів:** модулі SASS дозволяють використовувати стилі повторно, що сприяє швидкому розвитку та підтримці застосунку. Наприклад, можливо створити бібліотеку стилів з загальними компонентами, які можна використовувати в різних частинах застосунку;

– **кросплатформність:** модулі SASS можуть бути легко адаптовані для різних платформ та пристроїв, що дозволяє забезпечити однаковий вигляд та поведінку застосунку на різних пристроях. Можна використовувати медіа-запити та інші функції SCSS для адаптації стилів до різних екранів та роздільної здатності.

Приклад програмної реалізації стилізації із забезпеченням адаптивності до розміру екрану компонента:

```
.content {
  display: flex;
  padding: 20px;
  box-shadow: rgba(99, 99, 99, 0.2) 0px 2px 8px 0px;
  @media (min-width: 480px) {padding: 40px;}
}
.bannerWrapper {
  @media (min-width: 1024px) {width: 60%;}
  @media (min-width: 1440px) {width: 50%;}
}
```

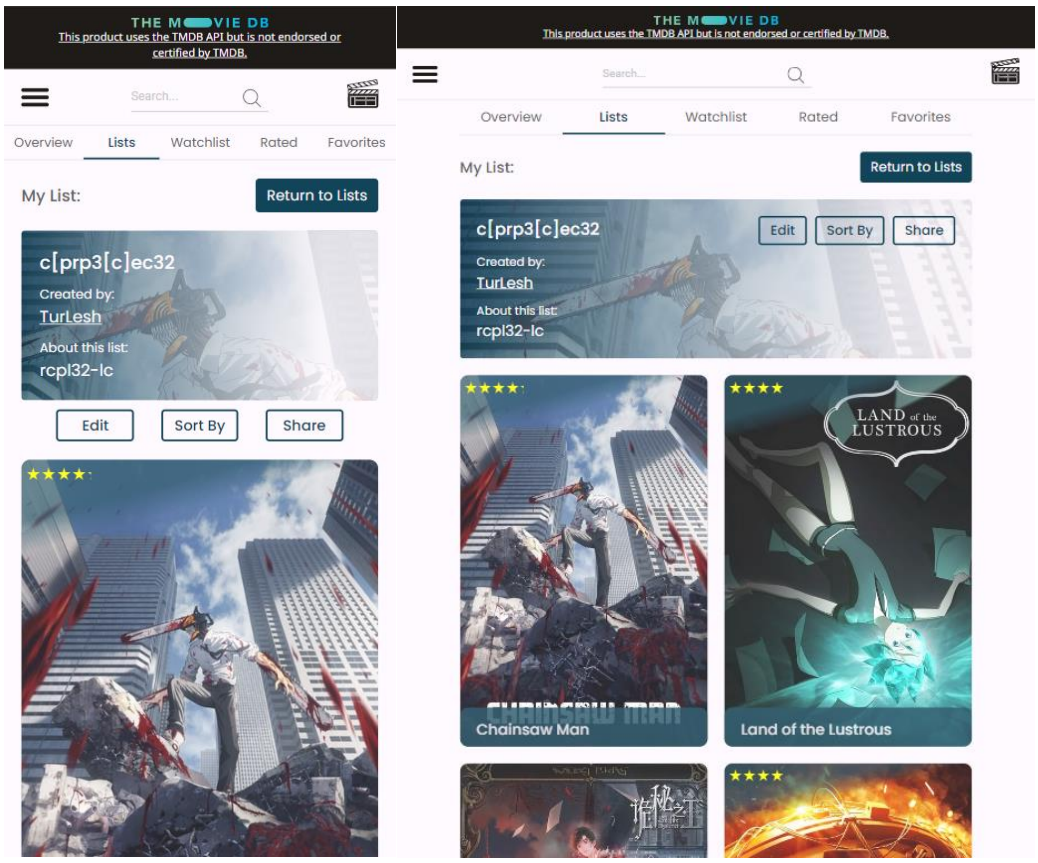
Рисунок 4.3 – Стилізація компонентів *content* та *bannerWrapper* з використанням медіавиразів

Стилізація відіграє вирішальну роль у створенні привабливих і професійних інтерфейсів користувача для вебсайтів. Щоб скористатися розширеними функціями стилю та покращити робочий процес розробки, інтеграція SASS у проєкт React є чудовим вибором. SASS надає такі функції, як змінні, вкладення, міксини та успадкування, які допомагають вам писати

більш модульний, підтримуваний код CSS. Файли SASS компілюються у звичайні файли CSS.

Перевірити результат забезпечення кросплатформності клієнтської частини застосунку можливо завдяки інструментам розробника у будь-якому браузері. Іструменти розробника – це набір інструментів, вбудованих безпосередньо у браузер. Вони дозволяють швидко редагувати сторінки та швидко діагностувати проблеми, що допомагає створювати якісніші застосунки.

Приклад сторінки конкретного списку створеного користувачем надано на рисунках нижче.



а) б)

Рисунок 4.4 – Список створений користувачем: а) перегляд клієнтської частини комплексу на мобільному девайсі «iPhone 14 Pro Max»; б) перегляд клієнтської частини комплексу на планшетному девайсі «iPad Mini»

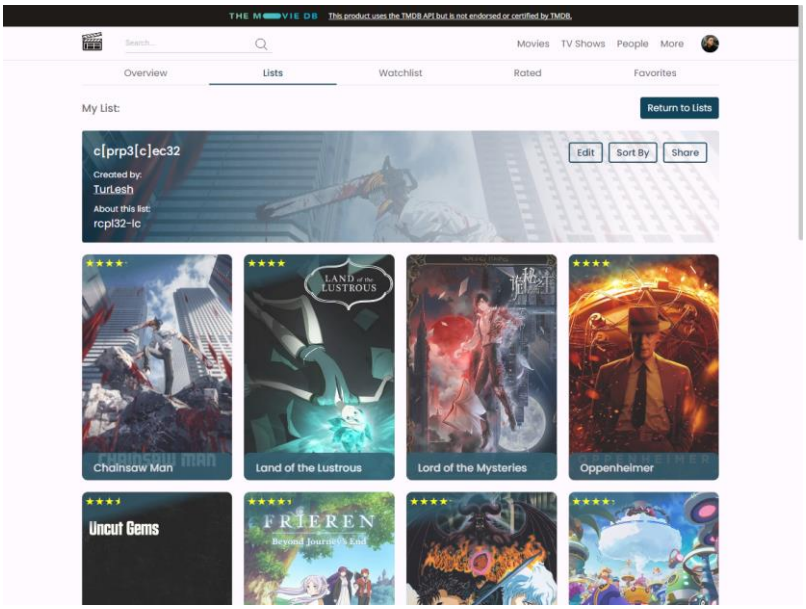


Рисунок 4.5 – Перегляд клієнтської частини комплексу на планшетному девайсі «iPad Pro»

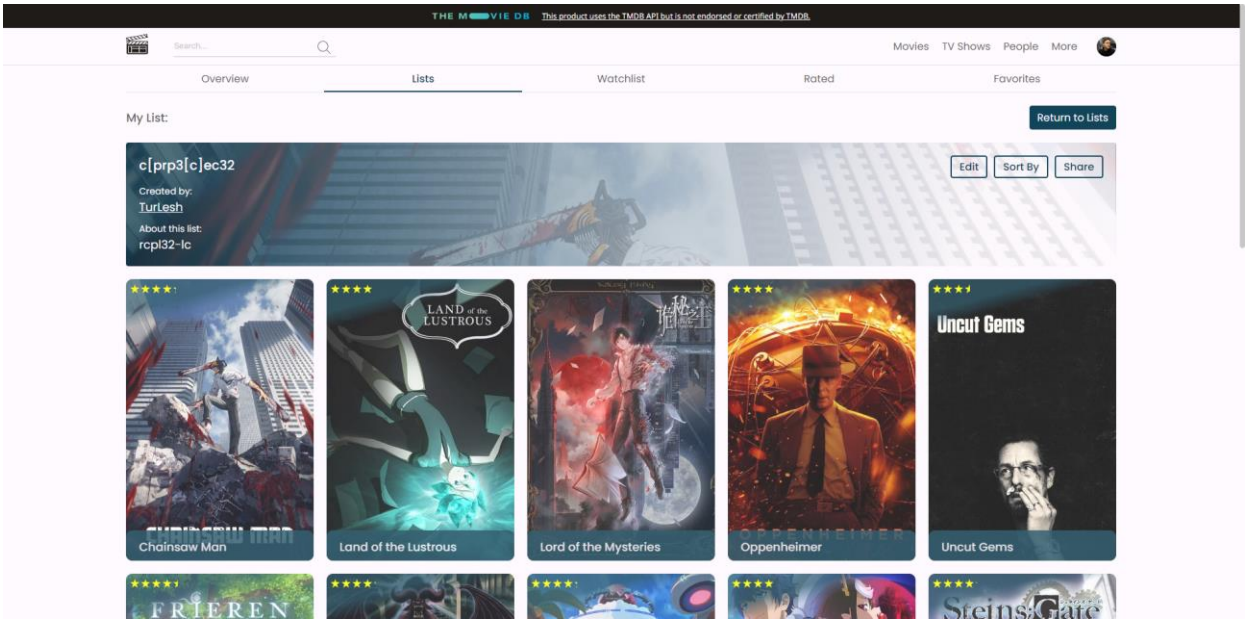


Рисунок 4.6 – Перегляд клієнтської частини комплексу на екрані ноутбука

Окрім інтсрументів розробника існують інші шляхи перевірки та забезпечення кросплатформності:

– **браузери:** перевірка роботи застосунку у різних браузерах забезпечує сумісність та коректне відображення інтерфейсу;

– **пристрої:** тестування на реальних пристроях дозволяє виявити специфічні проблеми, пов’язані з різними платформами. Так, наприклад, розгорнутий застосунок можливо перевірити прямо на своєму мобільному девайсі;

– **UI бібліотеки:** використання готових бібліотек компонентів (наприклад, Material-UI, Ant Design) можуть стати вірішенням питання забезпечення кросплатформного інтерфейсу. Такі бібліотеки впроваджують кросплатформні рішення, і до того ж, компоненти можливо перевірити на кросплатформність прямо на сторінках цих бібліотек;

– **поліфіли та транслятори:** можуть бути використані для забезпечення сумісності з різними браузерами та платформами;

– **умовний рендеринг:** використання умовного рендерингу може вирішити проблему адаптації інтерфейсу під різні платформи.

Підсумовуючи, для забезпечення кросплатформності клієнтської частини було програмно реалізовано гумову розмітку за допомогою модулів SASS та принципу «Mobile First». Переваги використання модулів SASS включають організацію стилів у логічні блоки, модульність, що дозволяє кожному компоненту мати свої власні стилі, та можливість перевикористання стилів для швидкого розвитку та підтримки застосунку. Завдяки адаптації стилів для різних платформ та пристроїв, застосунок має однаковий вигляд і поведінку на всіх пристроях, що можна перевірити за допомогою інструментів розробника у будь-якому браузері.

Висновки до розділу

Останній розділ був присвячений забезпеченню та тестуванню кросплатформності частин застосунку, адже сучасний вебзастосунок має бути здатним до підтримки більшості платформ у зв’язку з великим попитом.

Оскільки раніше було визначено що серверна частина застосунку є кросплатформною за замовчуванням, було вирішено, першим чином, протестувати це. Для цього був використаний мобільний гаджет для відсилання запиту на сервер в браузері. У результаті було доведено, що серверна частина застосунку є кросплатформною мобільний гаджет отримав очікувану відповідь.

Залишаючи серверну частину, було виконано роботу із забезпечення кросплатформності клієнтської частини застосунку. Завдяки модулям SASS, що інкапсульовували стилі, було легко виконувати налаштування розмітки макету. В результаті було реалізовано гумовий дизайн, що дозволив клієнтській частині застосунку виглядати гарно практично на будь-якому екрані. Завдяки інструментам розробника в браузері гумовоа розмітка була протестована, а результати представлені у вигляді клієнтського застосунку на трьох різних розмірах екрану.

Підсумовуючи, кросплатформність застосунку було успішно реалізовано та протестовано.

ВИСНОВКИ

У процесі дослідження та розробки кросплатформного застосунку на базі React та REST API, було досягнуто виконано поставлену мету, а саме: було розроблено програмно-апаратний комплекс, досліджено архітектуру застосунку на базі React та REST API, проведено аналіз взаємодії клієнтської частини застосунку з сервером при використанні прикладного програмного інтерфейсу REST, розглянено необхідні інструменти та засоби для забезпечення кросплатформності.

У ході виконання робот було виконано усі поставлені завдання: було проведено аналітичний огляд архітектури кросплатформного застосунку на базі React та REST API та визначиено її складові, було розглянуто поширені архітектурні рішення для такого застосунку та на базі цих рішень спроєктовано архітектуру для програмної реалізації застосунку на базі React та REST API, було розроблено апаратну частину комплексу та перевірено апаратне забезпечення на стійкість до навантажень, а також було забезпечено і протестовано кросплатформність розробленого застосунку.

Аналіз і проєктування архітектури застосунку показали, що використання React та REST API дозволяє створювати масштабовані та легко адаптовані рішення, що є особливо актуальним в умовах стрімкого розвитку технологій та зростання вимог до програмного забезпечення. Загалом, проведене дослідження підтвердило актуальність використання React та REST API для створення кросплатформних застосунків.

Результати роботи можуть бути корисні розробникам та тестувальникам програмного забезепечення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Meta. URL: https://about.meta.com (Last accessed: 09.05.2024).

2. Thinking in React. URL: https://react.dev/learn/thinking-in-react (Last accessed: 09.05.2024).

3. Narayan P. Modern JavaScript Applications : Packt Publishing, 25.07.2016. 330 c. URL: https://www.google.com.ua/books/edition/Modern_JavaScript_Applications/eJWqDQAAQBAJ?hl=en&gbpv=1&pg=PA2&printsec=frontcover (Last accessed: 09.05.2024).

4. What is a REST API? URL: https://www.ibm.com/topics/rest-apis (Last accessed: 09.05.2024).

5. JavaScript. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (Last accessed: 09.05.2024).

6. Frost B. Atomic Design : Brad Frost Web, 28.11.2016. 193 c. URL: https://www.google.com.ua/books/edition/Atomic_Design/1e92vgAACAAJ?hl=en (Last accessed: 09.05.2024).

7. Modular React applications. URL: https://javascript.plainenglish.io/modular-react-applications-c316783df0aas (Last accessed: 09.05.2024).

8. Get started with the basics of the TMDB API. URL: https://developer.themoviedb.org/reference/docs/getting-started (Last accessed: 10.05.2024).

9. React ES6. URL: https://www.w3schools.com/react/react_es6.asp (Last accessed: 09.05.2024).

10. Introducing Hooks. URL: https://legacy.reactjs.org/docs/hooks-intro.html (Last accessed: 09.05.2024).

11. Sass: Syntactically Awesome Style Sheets. URL: https://sass-lang.com/ (Last accessed: 10.05.2024).

12. Redux – A JS library for predictable and maintainable global state management. URL: https://redux.js.org/ (Last accessed: 10.05.2024).

13. Axios. URL: https://axios-http.com/ (Last accessed: 10.05.2024).

14. Node.js – Run JavaScript Everywhere. URL: https://nodejs.org/en (Last accessed: 10.05.2024).

15. NestJS – A progressive Node.js framework. URL: https://nestjs.com/ (Last accessed: 10.05.2024).

16. Fernando Doglio. Pro REST API Development with Node.js : Apress, 26.05.2015. 196 с. URL: https://books.google.com.ua/books?id=kjUwCgAAQBAJ&source=gbs_navlinks_s (Last accessed: 11.05.2024).

17. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: https://www.postgresql.org/ (Last accessed: 12.05.2024).

18. What is SQL (Structured Query Language)? URL: https://aws.amazon.com/what-is/sql/ (Last accessed: 13.05.2024).

19. Node.js Everywhere with Environment Variables! URL: https://medium.com/the-node-js-collection/making-your-node-js-work-everywhere-with-environment-variables-2da8cdf6e786 (Last accessed: 13.05.2024).

20. pgAdmin – PostgreSQL Tools. URL: https://www.pgadmin.org/ (Last accessed: 14.05.2024).

21. Postman API Platform. URL: https://www.postman.com/ (Last accessed: 14.05.2024).

22. Heroku: Cloud Application Platform. URL: https://www.heroku.com/ (Last accessed: 15.05.2024).

23. Cloud Computing Services – Amazon Web Services (AWS). URL: https://aws.amazon.com/ (Last accessed: 15.05.2024).

24. The Heroku CLI. URL: https://devcenter.heroku.com/articles/heroku-cli (дата звернення: 15.05.2024).

25. Heroku Postgres. URL: https://devcenter.heroku.com/articles/heroku-postgresql (Last accessed: 16.05.2024).

26. Git. URL: https://www.git-scm.com/ (Last accessed: 16.05.2024).

27. REST API Design Best Practices Handbook – How to Build a REST API with JavaScript, Node.js, and Express.js. URL: https://www.freecodecamp.org/news/rest-api-design-best-practices-build-a-rest-api (Last accessed: 17.05.2024).

28. What is Mobile First Design? Why It’s Important & How To Make It? URL: https://medium.com/@Vincentxia77/what-is-mobile-first-design-why-its-important-how-to-make-it-7d3cf2e29d00 (Last accessed: 18.05.2024).

ДОДАТОК А

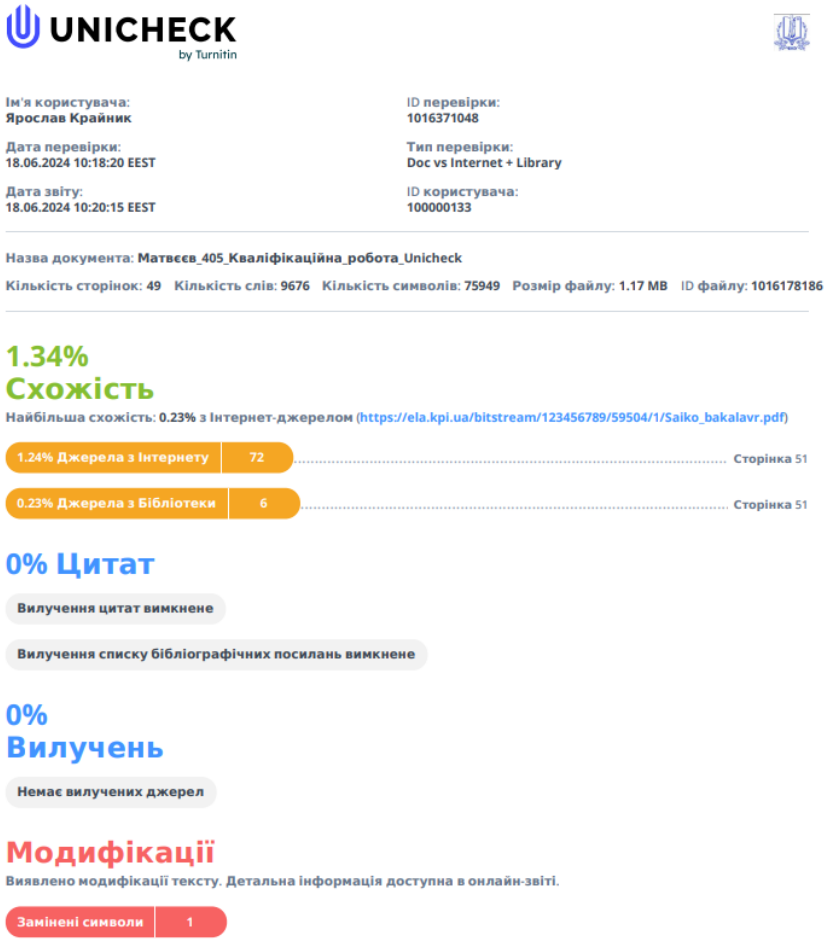
Довідка

про перевірку на унікальність пояснювальної записки

бакалаврської кваліфікаційної роботи на тему:
«Архітектура кросплатформного застосунку на базі React та REST API»

студента спеціальності 123 «Комп'ютерна інженерія», 405 групи
Матвєєва В'ячеслава Олександровича
(прізвище, ім'я, по-батькові)

Перевірку тексту здійснено сервісом: онлайн-сервіс Unicheck
Результат перевірки тексту бакалаврської кваліфікаційної роботи: схожість складає 1,34%.


UNICHECK
by Turnitin

Ім'я користувача: Ярослав Крайник
ID перевірки: 1016371048
Дата перевірки: 18.06.2024 10:18:20 EEST
Тип перевірки: Doc vs Internet + Library
Дата звіту: 18.06.2024 10:20:15 EEST
ID користувача: 100000133

Назва документа: Матвєєв_405_Кваліфікаційна_робота_Unicheck
Кількість сторінок: 49 Кількість слів: 9676 Кількість символів: 75949 Розмір файлу: 1.17 MB ID файлу: 1016178186

1.34%
Схожість
Найбільша схожість: 0.23% з Інтернет-джерелом (https://ela.kpi.ua/bitstream/123456789/59504/1/Saiko_bakalavr.pdf)

1.24% Джерела з Інтернету 72 Сторінка 51
0.23% Джерела з Бібліотеки 6 Сторінка 51

0% Цитат
Вилучення цитат вимкнене
Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень
Немає вилучених джерел

Модифікації
Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.
Замінені символи 1

Здобувач:

_____ В. О. Матвєєв
(підпис) *(ініціали, прізвище)*

Керівник:
ст. викладач
_____ Я. М. Крайник
(підпис) *(ініціали, прізвище)*

Дата: «___» _____ 2024 р.