Міністерство освіти і науки України
Український державний університет науки і технологій

Комп’ютерних технологій і систем

Комп'ютерні інформаційні технології

Пояснювальна записка
до кваліфікаційної роботи
магістра

на тему: Дослідження ефективності методів web API архітектур GraphQL та RestAPI
за освітньою програмою Інженерія програмного забезпечення
зі спеціальності: 121 Інженерія програмного забезпечення

Виконав: студент групи: ПЗ2321 Єгор МАРДЕРОСОВ

Керівник: доцент Олена КУРОП’ЯТНИК

Нормоконтролер: доцент Світлана ВОЛКОВА

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент ________
(підпис)

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

Computer Technologies and Systems
(faculty)

Computer information technology
(department)

Explanatory Note
to Master's Thesis
(higher education degree)

on the topic: Researching the effectiveness of web API methods of GraphQL and RestAPI architectures
according to educational curriculum Software engineering
in the Specialty: 121 Software engineering
(specialty and its code)

Done by the student of the group: PZ2321 Yehor MARDEROSOV

Scientific Supervisor: Olena KUROPIATNYK

Normative controller: Svitlana VOLKOVA

Dnipro – 2025

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: «Комп'ютерні технології і системи»
Кафедра: «Комп'ютерні інформаційні технології»
Рівень вищої освіти: магістр
Освітня програма: «Інженерія програмного забезпечення»
Спеціальність: «121 Інженерія програмного забезпечення»
(шифр та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ____КІТ__
___________ Вадим ГОРЯЧКІН _

___________ _____________ 202__р

З А В Д А Н Н Я

на кваліфікаційну роботу магістра
(ступінь вищої освіти)

студенту Мардеросову Єгору Вадимовичу
(Прізвище, Ім'я По батькові)

1. Тема роботи: Дослідження ефективності методів web API архітектур GraphQL та RestAPI

Керівник роботи: Куроп'ятник Олена Сергіївна, кандидат технічних наук
(Прізвище, Ім'я, По батькові, науковий ступінь, вчене звання)

затверджені наказом від "26" 09 2024 р. № 1187 ст

2. Строк подання студентом роботи: 20.01.2025 р.
3. Вихідні дані до роботи: ______________________

4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати):
4.1 Аналітична частина
4.2 Основна частина
4.3 Розробка ПЗ
4.4 Аналіз отриманих метрик
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):
5.1 Презентація
5.2 Відео ролик демонстрація роботи програми

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	15.09.24	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	16.09.24	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	22.10.24	
4	Постановка задачі, технічне завдання	06.11.24	30%
5	Техніко-економічні показники	13.11.24	
6	Розробка інструментальних засобів дослідження	14.12.24	
7	Виконання досліджень	15.12.24	60%
8	Оформлення тез доповідей	07.01.25	
9	Оформлення пояснювальної записки	10.01.25	
10	Розробка демонстраційних матеріалів	11.01.25	100%
11	Подання кваліфікаційної роботи до кафедри	20.01.25	
12	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	23.01.25	

Студент ________________ Єгор МАРДЕРОСОВ

Керівник роботи ________________ Олена КУРОП’ЯТНИК

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра: 94 с., 13 рис., 17 джерел та 3 додатки.

Магістерська робота на тему *«Дослідження ефективності методів web API архітектур GraphQL та RestAPI»* дослідженню ефективності двох популярних архітектур веб-API: RESTful API та GraphQL API

Об’єктом дослідження веб-API архітектури, зокрема RESTful API та GraphQL API, які використовуються для обміну даними між клієнтськими та серверними частинами веб-додатків.

Метою роботи є проведення порівняльного аналізу продуктивності архітектур RESTful API та GraphQL API, визначення їхніх переваг і недоліків у різних умовах використання, а також розробка рекомендацій щодо вибору оптимальної архітектури для реалізації веб-додатків залежно від специфічних вимог і сценаріїв роботи.

Методи дослідження: експериментальний метод, статистичний аналіз, порівняльний аналіз ефективності API.

Наукова новизна роботи полягає у проведенні системного порівняльного дослідження продуктивності RESTful API та GraphQL API, яке враховує їхні ключові метрики, такі як час виконання запитів, стабільність під навантаженням, обсяг переданих даних та інші параметри.

Особливістю є експериментальний підхід із застосуванням автоматизованого збору метрик та тестування під навантаженням за допомогою K6. Робота надає нові дані щодо ефективності цих архітектур у контексті сучасних вимог до веб-додатків і може слугувати основою для вдосконалення практик розробки API.

Результати дослідження: розроблено інструмент для збору метрик ефективності, проведено експериментальний аналіз методів двох web-api.

Ключові слова: NODE.JS, NESTJS, POSTGRESQL, МЕТРИКИ ПРОДУКТИВНОСТІ, MIDDLEWARE, АРХІТЕКТУРА ВЕБ-ДОДАТКІВ.

ЗМІСТ

ВСТУП……………………………………………………………………………..9
1. ФОРМУЛЮВАННЯ ЗАВДАННЯ ДОСЛІДЖЕННЯ…………………………12
1.1 Актуальність та практична цінність дослідження……………………......12
1.2 Мета і завдання дослідження……………………………………………12
1.3 Методи дослідження…………………………………...………...13
1.4 Очікувані результати…………………………………………..…...13
2. ОГЛЯД ЛІТЕРАТУРИ ТА СТАН ПРОБЛЕМИ……………………………14
2.1 Визначення та принципи роботи REST API………………………………14
2.2 Визначення та принципи роботи GraphQL……………………………..15
2.3 Порівняння архітектурних підходів……………………………………15
2.4 Огляд попередніх досліджень………………………………………...17
Висновки до розділу 2……………………………..………………...18
3. АНАЛІЗ ОСНОВНИХ ХАРАКТЕРИСТИК………………………………20
3.1 Прдуктивність REST API ……………………………………….…20
3.2 Продуктивність GraphQL…………………………………………..……21
3.3 Переваги та недоліки REST API……………...…………………………22
3.4 Переваги та недоліки GraphQL…..…………………………………..22
3.6 Висновки до розділу 3…………………….………...…………………….…..23
4. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ…..………………………25
4.1 Вибір технологій для розробки ПЗ………………………..……………25
4.2 Реалізація бази даних……………………………………………….…25
4.3 Реалізація модуля user…………………………………………………...27
4.3.1 Реалізація за допомоги graphQL архітектури……………………..27
4.3.2 Реалізація за допомоги REST архітектури………………………..32
Висновки до розділу 4..………….……………………………………………33
5. МЕТОДИКА ДОСЛІДЖЕННЯ……………………..………………………35
5.1 Опис тестового середовища……..…………………….……………………35
5.2 Вибір метрик для аналізу ефективності……………………………..36
5.3 Опис тестових сценаріїв навантаження…………………………………...38

5.3.1 Аналіз продуктивності API під час обробки послідовних запитів…40
5.3.2 Моделювання сценаріїв стресового навантаження API...............40
5.4 Аналіз результатів послідовних запитів…………………..……………......41
5.4.1 Графічне представлення результатів послідовних запиті………......45
5.5 Аналіз результатів стресостійкості……………………………………..50
5.5.1 Аналіз пікового навантаження……………………………………..50
5.5.2 Аналіз тривалого навантаження…………………………………...53
Висновки до розділу 5……………………………………………………….56
ВИСНОВКИ…………………….………………………………………………..57

ВСТУП

Актуальність роботи. У сучасному світі розробка веб-API є невід’ємною частиною створення масштабованих і продуктивних веб-додатків. REST API та GraphQL API є двома найпопулярнішими підходами до реалізації веб-інтерфейсів, але їхні переваги та недоліки залежать від конкретних умов використання. Незважаючи на численні дослідження продуктивності цих архітектур, досі відсутній комплексний аналіз їхньої ефективності з урахуванням ключових метрик, таких як час виконання запитів, стабільність під навантаженням і обсяг переданих даних, у реальних умовах експлуатації. Це створює необхідність проведення системного дослідження для визначення оптимальних умов застосування кожної архітектури, що сприятиме прийняттю обґрунтованих рішень при виборі технології для конкретного проєкту.

Тема роботи: «Дослідження ефективності методів web API архітектур GraphQL та REST API».

Об’єкт дослідження – процеси взаємодії «клієнт-сервер» в архітектурах REST API та GraphQL API.

Предмет дослідження – характеристики архітектур REST API та GraphQL API за різних умов використання.

Мета та задачі дослідження. Метою магістерської роботи є проведення порівняльного аналізу продуктивності REST API та GraphQL API, виявлення їхніх переваг і недоліків, а також розробка рекомендацій щодо вибору архітектури залежно від вимог конкретного проєкту.

Для досягнення мети було поставлено такі задачі:

- вивчення основних принципів функціонування REST API та GraphQL API;
- визначення ключових метрик для оцінки продуктивності обох архітектур;
- проведення експериментального тестування обох API у реальних умовах навантаження;

– формування рекомендацій щодо вибору архітектури залежно від типових вимог проєкту.

Методи дослідження. У процесі виконання роботи застосовувалися методи експериментального тестування із використанням K6 для навантажувальних тестів, модульного підходу для розробки REST API та GraphQL API на базі NestJS, а також методи збору й аналізу даних за допомогою middleware, інтегрованого для відстеження метрик продуктивності. Для аналізу результатів використовувалися статистичні методи та візуалізація даних у вигляді графіків.

Наукова новизна. Робота представляє системне порівняльне дослідження продуктивності REST API та GraphQL API із врахуванням ключових метрик у реальних умовах використання. Вперше проведено аналіз впливу таких оптимізацій, як кешування, пагінація та компресія, на ефективність цих архітектур. Одержані результати дозволяють сформувати чіткі рекомендації щодо вибору архітектури API.

Практичне значення. Результати роботи можуть бути використані для вдосконалення процесів проєктування веб-додатків, забезпечуючи вибір найбільш ефективної архітектури API залежно від умов експлуатації. Розроблене програмне забезпечення може застосовуватись у навчальних і дослідницьких цілях, а також для аналізу продуктивності API у реальних проєктах.

Апробація. Результати роботи було представлено на наукових семінарах кафедри «Комп’ютерні інформаційні технології» 01.11.2024, 10.01.2025 та науково-практичній конференції «Сучасні інформаційні та комунікаційні технологіїї на транспорті, в промисловості і освіті» 12-13 грудня 2024 р.

1 ФОРМУЛЮВАННЯ ЗАВДАННЯ ДОСЛІДЖЕННЯ

1.1 Актуальність та практична цінність дослідження

З огляду на зростаючі вимоги до продуктивності веб-додатків, проблема оптимізації часу виконання запитів у RESTful API є надзвичайно актуальною. Монолітні додатки, що працюють із великими обсягами даних та складною бізнес-логікою, часто стикаються з вузькими місцями у продуктивності. Це зумовлено неефективною роботою з базами даних, надлишковим трафіком, а також неправильною конфігурацією запитів.

Актуальність дослідження обумовлена потребою у стабільній та швидкій роботі RESTful API у сучасних додатках без необхідності повного переходу до мікросервісної архітектури. Застосування Onion-архітектури дозволяє гнучко управляти залежностями та впроваджувати оптимізації на рівні структурних компонентів системи.

1.2 Мета і завдання дослідження

Метою дослідження є порівняння двох основних архітектур веб-API — REST API та GraphQL — для виявлення їх переваг, недоліків і оптимальних умов використання. Це дослідження спрямоване на глибше розуміння обох підходів та розробку рекомендацій для вибору відповідної технології залежно від вимог конкретного проєкту.

Для досягнення цієї мети необхідно виконати такі завдання:

- проаналізувати основні принципи функціонування REST API та GraphQL;
- визначити критерії порівняння, такі як ефективнісить (середній час виконання запитів, стабільність під навантаженням).
- застосувати експериментальний підхід до оцінювання, який включає проведення навантажувальних тестів за допомогою K6, збір метрик продуктивності через middleware та аналіз ефективності оптимізацій, таких як кешування, пагінація та компресія;

- провести порівняльний аналіз на основі експериментальних даних і конкретних прикладів.

Результати цього дослідження дозволять сформулювати рекомендації щодо вибору відповідної архітектури API залежно від специфічних вимог проєкту.

1.3 Методи дослідження

У процесі дослідження використовувалися такі методи:

- вивчення документації – для ознайомлення з основними принципами роботи REST API та GraphQL. Це включає вивчення офіційної документації, технічних специфікацій та посібників із використання обох технологій;
- розробка двох web-API – здійснювалася із застосуванням методів програмної інженерії, включаючи модульний підхід, використання фреймворка NestJS для структурування коду та забезпечення масштабованості, а також інтеграцію middleware для збору метрик продуктивності. Це дозволило створити дві системи (на основі REST API та GraphQL), що використовують однакові сценарії взаємодії з сервером для проведення практичного порівняння.
- порівняльний аналіз – для оцінки продуктивності, гнучкості та ефективності REST API і GraphQL на основі кількісних та якісних показників. Дослідження включало експериментальне впровадження обох технологій у тестові проєкти;
- експериментальний метод – для практичної оцінки обох підходів на основі виконання запитів до бази даних та інших серверних операцій. Вимірювалися швидкість обробки запитів, навантаження на сервер, обсяг переданих даних та інші показники;
- синтез результатів – для формування висновків і розробки рекомендацій щодо вибору відповідної архітектури web-api залежно від умов конкретних проєктів.

1.4 Очікувані результати

Очікуваними результатами роботи є отримання висновків про ефективність архітектур GraphQL та REST API в умовах виконання типових запитів до веб-додатків. Дослідження дозволить визначити, яка з архітектур забезпечує кращу продуктивність, швидкість обробки запитів і використання ресурсів за однакових умов. Це допоможе зрозуміти їхні сильні та слабкі сторони, що є важливим для вибору найбільш відповідного підходу до реалізації веб-сервісів.

2 ОГЛЯД ЛІТЕРАТУРИ ТА СТАН ПРОБЛЕМИ

2.1 Визначення та принципи роботи REST API

Representational State Transfer (REST) – це програмна архітектура, яка визначає умови роботи API. Спочатку REST створювалася як посібник для управління взаємодіями в складній мережі, такій як Інтернет. Архітектуру на основі REST можна використовувати для підтримки високопродуктивного і надійного зв'язку в необхідному масштабі. Її можна легко впроваджувати і модифікувати, забезпечуючи прозорість і крос-платформну переносимість будь-якої системи API.

Розробники можуть створювати API з використанням декількох архітектур. API-інтерфейси, що відповідають архітектурному стилю REST, називаються REST API. Веб-служби, що реалізують архітектуру REST, називаються веб-службами RESTful. Як правило, термін RESTful API відноситься до мережевих RESTful API. Однак REST API і RESTful API є взаємозамінними термінами [1].

Принципи роботи REST [2]:

- Stateless (Безстанова взаємодія): Кожен запит від клієнта до сервера містить всю необхідну інформацію для його обробки; сервер не зберігає стан клієнта між запитами;
- Client-Server (Клієнт-серверна архітектура): Клієнт і сервер функціонують незалежно, що дозволяє їм розвиватися окремо один від одного;
- Uniform Interface (Єдиний інтерфейс): Всі запити до ресурсу мають бути уніфікованими, незалежно від їх походження. Кожен ресурс ідентифікується унікальним URL;
- Cacheable (Кешування): Відповіді сервера повинні містити інформацію про те, чи можуть вони бути кешовані, що підвищує ефективність роботи системи;

- Layered System (Багатошарова система): Запити та відповіді можуть проходити через кілька проміжних шарів (наприклад, проксі-сервери), що підвищує масштабованість і безпеку системи;
- Code on Demand (Код на вимогу) — опціонально: Сервер може надсилати виконуваний код клієнту (наприклад, скрипти), що розширює функціональність клієнта.

2.2 Визначення та принципи роботи GraphQL

GraphQL– мова запитів і маніпулювання даними для API. Її розробила компанія Facebook у 2012 році та випустила публічно у 2015 році. GraphQL є відкритим вихідним кодом і підтримується GraphQL Foundation, некомерційною організацією, створеною для просування і розвитку GraphQL.

Цей засіб взаємодії дає змогу клієнту точно вказати, які дані йому потрібні від сервера. Це на відміну від традиційних API, які надають заздалегідь визначені набори даних. GraphQL також дає змогу клієнту обирати, як він хоче отримати дані, наприклад, у форматі JSON, XML або іншому [3].

Принципи роботи GraphQL [4]:

- єдиний ендпоінт: GraphQL використовує один ендпоінт для всіх запитів, на відміну від REST, де для різних ресурсів використовуються різні ендпоінт;
- гнучкість запитів: Клієнт може запитувати лише ті поля, які йому потрібні, що зменшує обсяг переданих даних і підвищує ефективність;
- відсутність надмірності та недостатності даних: GraphQL допомагає уникнути ситуацій, коли клієнт отримує занадто багато або недостатньо даних, оскільки він сам визначає структуру відповіді;
- еволюція API без версіонування: Завдяки гнучкості та можливості додавати нові поля без порушення існуючих запитів, GraphQL дозволяє розвивати API без необхідності створення нових версій

2.3 Порівняння архітектурних підходів

GraphQL і REST – два різні підходи до розробки API для обміну даними через Інтернет. REST дає можливість клієнтським додаткам обмінюватися даними із сервером за допомогою команд HTTP – стандартного протоколу зв'язку в Інтернеті. А GraphQL – це мова запитів API, що визначає специфікації, за якими клієнтський застосунок має запитувати дані з віддаленого сервера. Ви можете використовувати GraphQL у викликах API для визначення запиту, не покладаючись на серверний додаток. GraphQL і REST – дві потужні технології, на яких ґрунтується більшість наших сучасних додатків.

Архітектура REST, і GraphQL реалізують кілька загальних архітектурних принципів API. Наприклад:

- обидва варіанти не вимагають збереження стану, тому сервер не зберігає історію відповідей між запитами;
- в обох випадках використовується модель «клієнт-сервер», тому запити від одного клієнта викликають відповіді від одного сервера;
- обидва протоколи засновані на HTTP, оскільки HTTP є базовим протоколом зв'язку.

Обмін даними, і REST, і GraphQL підтримують схожі формати даних.

JSON – найпопулярніший формат обміну даними, зрозумілий усім мовам, платформам і системам. Сервер повертає клієнту дані JSON. Інші формати даних доступні, але використовуються рідше, наприклад XML і HTML.

Аналогічним чином, REST і GraphQL підтримують кешування. Таким чином, клієнти і сервери можуть кешувати часто використовувані дані для підвищення швидкості зв'язку [5].

Ключові відмінності наступні [6].

Структура запитів:

- REST API: У REST API кожен ендпоінт (URL) представляє ресурс, а тип HTTP-запиту (GET, POST, PUT, DELETE тощо) вказує на дію, яку

потрібно виконати з цим ресурсом. Отже, клієнт формує окремі HTTP-запити для отримання, створення, оновлення або видалення ресурсів.

- GraphQL: У GraphQL клієнт може визначити точно ті дані, які йому потрібні, за допомогою одного запиту. Клієнт визначає структуру даних, яку він хоче отримати, і сервер повертає тільки ці дані.

Оптимізація запитів:

- REST API: У REST API клієнт зазвичай отримує всі дані, що пов'язані з ресурсом, навіть якщо він не потребує всіх цих даних. Це може призводити до перевантаження мережі та зайвого обсягу даних;
- GraphQL: У GraphQL клієнт отримує лише ті дані, які він запитав. Це дозволяє оптимізувати мережевий трафік та зменшувати обсяг переданих даних.

Масштабованість:

- REST API: У REST API масштабування може бути складним, оскільки потрібно створювати нові ендпоінти або розширювати існуючі для обробки нових вимог;
- GraphQL: У GraphQL масштабування є більш простим, оскільки клієнт може запитувати лише ті дані, які йому потрібні, і сервер повертає тільки ці дані.

Конфігурація:

- REST API: У REST API сервер визначає структуру даних, яку повертає;
- GraphQL: У GraphQL клієнт визначає, які дані він хоче отримати, забезпечуючи більшу гнучкість та контроль над відповідями сервера.

2.4 Огляд попередніх досліджень

Hygraph опублікував статтю [7], яка детально розглядає відмінності між REST і GraphQL. Вони підкреслюють, що GraphQL зменшує надмірне завантаження даними, оскільки клієнти можуть запитувати лише ту

інформацію, яка їм потрібна. Це призводить до більш ефективного використання ресурсів у порівнянні з REST.

Medium [8] також розглядає, як GraphQL та REST відрізняються за критичними факторами, такими як продуктивність і гнучкість. Вони зазначають, що GraphQL стає дедалі популярнішим серед розробників, і що більше 60% з них використовують або планують використовувати GraphQL у своїх програмах.

Дослідження відрізняється від наведених вище статей кількома важливими аспектами:

- глибший порівняльний аналіз: Проводиться детальніший аналіз різних аспектів, таких як продуктивність, зручність використання, безпека та управління даними в контексті реальних застосувань, що дозволяє вийти за межі загальних оглядів;
- практичні дослідження та реалізації: Реалізуються два web-api — один на REST, а інший на GraphQL — для порівняння їхньої продуктивності на основі конкретних метрик. Це дає можливість оцінити реальні переваги та недоліки кожного підходу.

Висновки до розділу 1

Аналіз літератури підтверджує важливість REST API та GraphQL API як провідних архітектурних підходів для побудови веб-інтерфейсів. REST API є більш традиційним підходом, що базується на стандартизованій структурі HTTP-запитів із використанням CRUD-операцій. Ця архітектура зручна для простих систем із чітко визначеними ендпоінтами. GraphQL, у свою чергу, пропонує інноваційний підхід, орієнтований на клієнта, дозволяючи запитувати лише необхідні дані, що зменшує надмірне передавання інформації та покращує продуктивність у складних запитах.

REST API забезпечує стабільність і зрозумілість завдяки використанню багатошарової системи, кешування та безстанової взаємодії, що робить його

ідеальним для систем із високим трафіком і простими запитами. GraphQL натомість дозволяє клієнту самостійно формувати запити, що значно підвищує гнучкість, але водночас вимагає більш складної реалізації на серверному боці.

Порівняльний аналіз показує, що REST API краще підходить для простих сценаріїв із передбачуваними запитами, тоді як GraphQL демонструє свою ефективність у складних сценаріях із великими обсягами даних і численними зв'язками. Масштабування REST API може бути складнішим через необхідність створення нових ендпоінтів, тоді як GraphQL дозволяє еволюцію API без версіонування завдяки своїй гнучкості.

Попередні дослідження підкреслюють, що GraphQL краще справляється із запитами на великі дані, але потребує більших ресурсів для обробки. Практична частина роботи, зосереджена на створенні двох web API – REST та GraphQL – дозволяє в реальних умовах оцінити їхню продуктивність, зручність використання та ефективність. Цей підхід дає змогу сформулювати чіткі рекомендації щодо вибору архітектури залежно від конкретних потреб застосунку.

3 АНАЛІЗ ОСНОВНИХ ХАРАКТЕРИСТИК

3.1 Продуктивність REST API

Продуктивність REST API визначається як здатність системи ефективно обробляти запити клієнтів з мінімальними затримками та оптимальним використанням ресурсів сервера. Основними показниками продуктивності є швидкість обробки запитів, кількість запитів, які система може обробити за одиницю часу, стабільність роботи під навантаженням і обсяг переданих даних. У цьому контексті продуктивність відображає, наскільки ефективно REST API виконує поставлені завдання в умовах реального використання, забезпечуючи швидку й надійну взаємодію між клієнтом і сервером.

REST API є одним із найбільш популярних підходів до проектування веб-сервісів. Продуктивність REST залежить від кількох факторів, включаючи кількість запитів, використання кешування, обробку даних на сервері та клієнті, а також структуру API.

Однією з особливостей REST є те, що кожен запит до API повертає повний набір даних, навіть якщо користувачу потрібна лише частина інформації. Це може збільшити обсяг переданих даних та навантаження на мережу, що безпосередньо впливає на продуктивність. Однак REST підтримує використання стандартних HTTP-заголовків для керування кешуванням, що дозволяє знизити навантаження на сервер і покращити час відповіді.

Крім того, REST API масштабовані завдяки безстатевій природі: кожен запит до сервера не залежить від попереднього, тому сервер не повинен зберігати стан сеансу клієнта між запитами. Це дозволяє легко обробляти великий обсяг одночасних запитів, зменшуючи навантаження на сервер.

Основною проблемою REST є надмірне завантаження даними. Наприклад, якщо клієнт потребує інформацію з декількох ресурсів, йому доводиться виконувати кілька запитів, що може призвести до уповільнення роботи системи через збільшення кількості HTTP-запитів [9].

3.2 Продуктивність GraphQL

GraphQL — це мова запитів для API, яка дозволяє клієнтам запитувати тільки ту інформацію, яка їм дійсно потрібна. Це зменшує надмірне завантаження даними і підвищує ефективність використання ресурсів. Оскільки клієнти можуть запитувати лише конкретні поля, запитів і відповідей стає менше, що підвищує продуктивність.

Продуктивність мови — це здатність мови (запитів, програмування або іншої) забезпечувати швидке та ефективне виконання завдань, покладених на систему, за допомогою її інструментів і синтаксису. У контексті GraphQL це означає, наскільки ефективно мова запитів дозволяє отримувати дані з мінімальним навантаженням на сервер, враховуючи обсяг і складність запитів, а також можливість уникати надлишкових операцій. Продуктивність мови також залежить від її здатності адаптуватися до потреб клієнта, наприклад, через можливість запитувати тільки потрібні дані або об’єднувати кілька запитів в один.

GraphQL забезпечує гнучкість у запитах, дозволяючи отримувати різноманітні дані через один запит. Наприклад, замість виконання декількох запитів у REST API для отримання пов'язаної інформації (наприклад, дані про користувача і його замовлення), у GraphQL можна зробити один запит, що поверне всі потрібні дані в оптимізованому форматі.

Однак продуктивність GraphQL може погіршуватися при складних запитах. Клієнти можуть формувати запити з великою кількістю вкладених полів, що призводить до значного навантаження на сервер. Це може бути особливо проблематично, якщо запит стосується складних взаємозв'язків між даними.

Крім того, GraphQL не має вбудованих механізмів кешування, як REST, що може збільшувати час відповіді на повторні запити. Проте існують додаткові інструменти, які допомагають вирішити цю проблему, зокрема використання клієнтських бібліотек для оптимізації запитів [10].

3.3 Переваги та недоліки REST API

Переваги REST API [11].

Простота і зрозумілість:

- використовує стандартні HTTP-методи;
- легко тестувати та налагоджувати.

Масштабованість:

- поділ клієнта і сервера;
- можливість кешування.

Гнучкість формату даних:

- підтримка JSON, XML та інших форматів;
- незалежність від конкретної мови програмування.

Висока продуктивність:

- відсутність збереження стану;

Недоліки REST API.

- надмірне завантаження даними: У REST кожен запит повертає повний набір даних, що може призводити до передачі непотрібної інформації і збільшувати навантаження на мережу;
- кілька запитів для пов’язаних даних: Для отримання складних даних з декількох ресурсів клієнту доводиться виконувати кілька запитів, що може уповільнювати роботу;
- жорстко визначені ресурси: у REST API кожен ресурс має фіксовану URL-адресу та структуру відповіді, що робить його менш гнучким для зміни вимог до даних.

3.4 Переваги та недоліки GraphQL

Переваги GraphQL [12]:

- спрощене керування версіями – на відміну від REST, GraphQL зменшує потребу у керуванні версіями, дозволяючи додавати нові поля й типи до

схеми, не впливаючи на наявні запити. Ця пряма сумісність означає, що клієнти та сервери можуть із часом розвиватися плавніше;

- вирішення проблеми N+1, здатність GraphQL отримувати дані за допомогою одного запиту незалежно від того, скільки базових служб потрібно викликати, безпосередньо вирішує проблему N+1, притаманну архітектурам REST. Це не тільки підвищує продуктивність, але й спрощує логіку отримання даних на стороні клієнта;
- уніфікована схема – в архітектурі мікросервісів кожна служба може мати свою схему, що представляє частину бізнес-області. GraphQL виділяється тим, що дозволяє об'єднувати або з'єднувати ці схеми, надаючи клієнтам єдиний інтерфейс. Це означає, що клієнти можуть запитувати дані з кількох служб в одному запиті, різко зменшуючи складність і кількість мережевих викликів.

Недоліки GraphQL.

- складність при масштабуванні: Через гнучкість запитів клієнти можуть створювати складні запити, що можуть спричиняти значне навантаження на сервер. Це потребує додаткових механізмів контролю та лімітування запитів;
- крута крива навчання: Хоча GraphQL забезпечує велику гнучкість, його впровадження вимагає глибокого розуміння мови запитів і архітектури. Це може бути складніше для команд, які звикли до традиційних підходів у REST.

Висновки до розділу 2

В цьому розділі проведено детальний аналіз продуктивності REST API та GraphQL API, а також розглянуто їхні переваги та недоліки. REST API залишається одним із найпопулярніших підходів завдяки простоті, зрозумілості та ефективному використанню стандартних HTTP-засобів. Масштабованість, кешування та незалежність від мови програмування роблять REST API зручним

для створення веб-сервісів. Проте надмірне завантаження даними та необхідність виконання декількох запитів для отримання пов’язаних ресурсів є ключовими недоліками цієї архітектури.

GraphQL, у свою чергу, забезпечує гнучкість у формуванні запитів, дозволяючи клієнту отримувати лише потрібні дані за допомогою одного запиту. Це значно зменшує обсяг переданих даних і вирішує проблему N+1, що є критичною для REST. Однак, відсутність вбудованого механізму кешування, складність у масштабуванні та крута крива навчання можуть стати викликами при впровадженні GraphQL у великих системах.

Порівняльний аналіз демонструє, що обидва підходи мають свої переваги в залежності від конкретних вимог проєкту. REST API підходить для систем із чітко визначеними ресурсами та високим навантаженням, тоді як GraphQL є оптимальним вибором для складних систем із потребою в гнучких запитах і мінімізації надмірних даних. Таким чином, вибір архітектури залежить від специфіки задачі, продуктивності системи та вимог бізнесу.

4 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вибір технологій для розробки ПЗ

Для розробки обох API було обрано технології NestJS, TypeScript, Node.js та PostgreSQL, що забезпечили гнучкість, ефективність і продуктивність системи.

TypeScript – це мова програмування, яка являє собою надмножину JavaScript. Вона додає статичну типізацію і деякі додаткові функції, які допомагають розробникам створювати більш надійні та масштабовані додатки. Однак, незважаючи на свої розширені можливості, TypeScript зберігає сумісність зі стандартами JavaScript, що робить його легко інтегрованим в наявні проекти [13].

Node.js — це однопоточне кросплатформове середовище виконання з відкритим вихідним кодом і бібліотека, яка використовується для запуску вебдодатків, написаних на JavaScript, поза браузером клієнта [14].

PostgreSQL - це система керування базами даних корпоративного класу з відкритим кодом. Він підтримує як SQL, так і JSON для реляційних і нереляційних запитів для розширюваності та відповідності SQL [15].

NestJS є прогресивним фреймворком для створення серверних додатків на Node.js, який базується на принципах об'єктно-орієнтованого програмування та модульності. Він дозволяє розробникам створювати масштабовані та підтримувані застосунки, використовуючи ін'єкцію залежностей, декоратори та інші сучасні патерни проектування. Це зменшує обсяг коду та підвищує його читабельність, що є важливим аспектом у командних проектах [16].

4.2 Реалізація бази даних

У рамках дослідження було реалізовано базу даних для управління даними користувачів, постів і коментарів за допомогою web-api. Для цього були створені три основні сутності: User**,** Post та Comment, що визначають основні об'єкти в нашій системі (див. рис. 4.1).

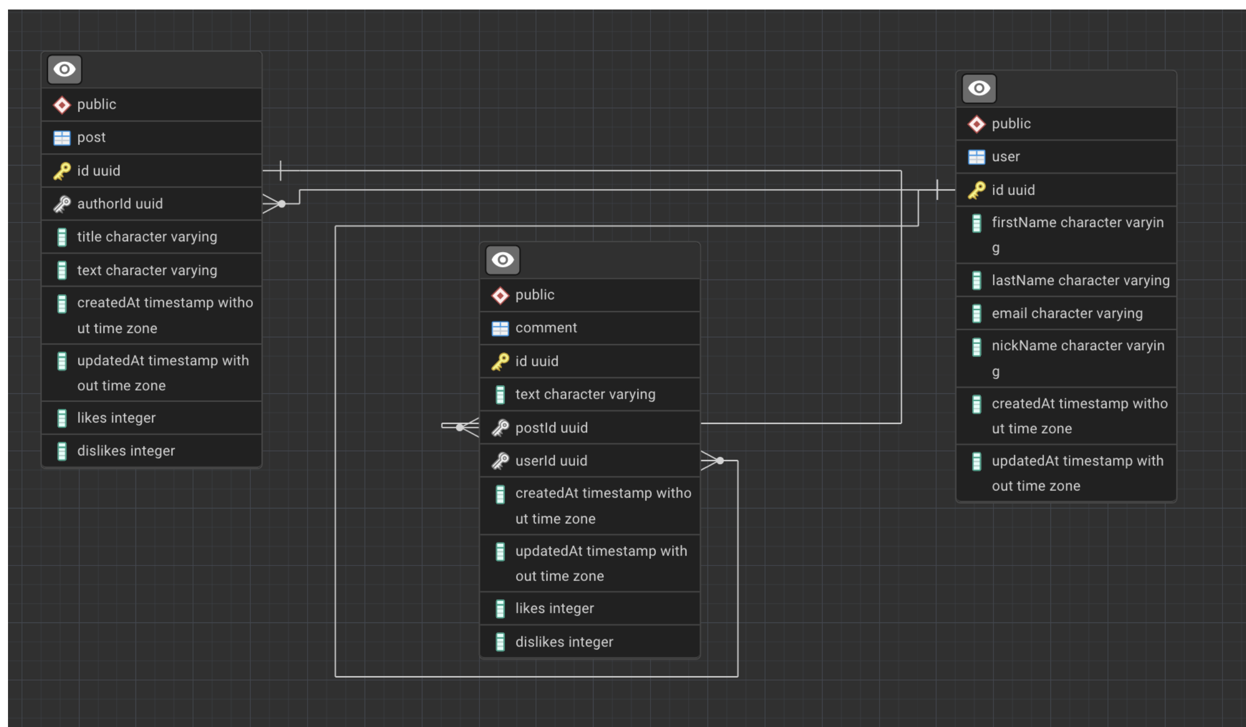

Рисунок 4.1 – ER діаграма бази даних

Сутність User відповідає за зберігання інформації про користувачів. Вона містить унікальний ідентифікатор, ім'я, прізвище, електронну адресу та нікнейм. Додатково в сутності передбачені дати створення та оновлення запису, що дозволяє відслідковувати історію змін. Взаємозв’язок з постами та коментарями реалізовано через зв’язки один-до-багатьох, що дозволяє одному користувачу мати багато постів і коментарів.

Сутність Post служить для зберігання інформації про пости, створені користувачами. Вона включає заголовок, текст посту, а також лічильники лайків і дизлайків. Як і в сутності користувача, у пості також зберігаються дати створення та оновлення. Взаємозв’язок між постом та автором реалізовано через зв’язок багато-до-одного, що забезпечує можливість вказувати автора для кожного посту. Крім того, кожен пост може мати багато коментарів, що також відображено через відповідний зв’язок.

Сутність Comment відображає інформацію про коментарі, що додаються до постів. Вона містить текст коментаря, ідентифікатор поста, до якого він

належить, і ідентифікатор користувача, який його написав. Тут також реалізовані лічильники лайків і дизлайків для коментарів. Зв'язки між коментарями та постами, а також коментарями та авторами реалізовані через зв'язок багато-до-одного, що дозволяє ефективно управляти коментарями в системі.

4.3 Реалізація модуля user

4.3.1 Реалізація за допомоги GRAPHQL архітектури

Структра модуля User побудованого на GraphQL архітектурі (див. рис. 4.2).

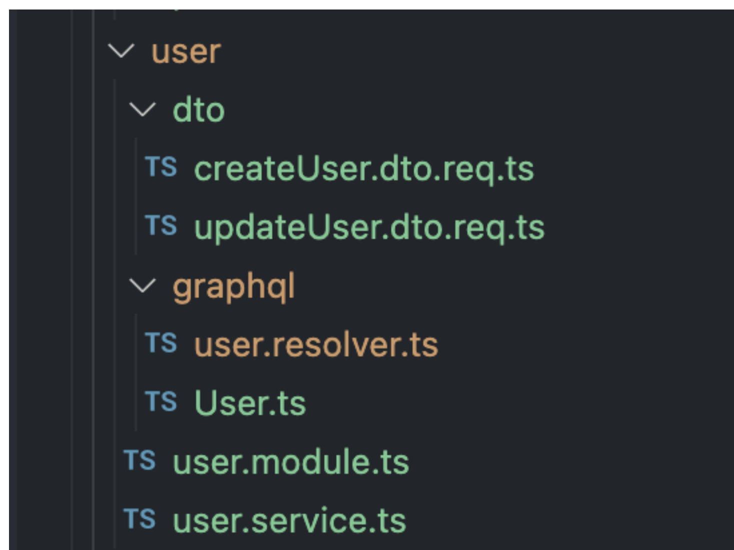

Рисунок 4.2 – Структура user модуля (архітектура GRAPHQL)

Основні складові реалізації

Схема GraphQL: у GraphQL структура даних описується за допомогою схеми, яка визначає типи об'єктів, їх поля та відносини між ними. Для модуля User створена схема, що включає типи User, Query та Mutation (див. лістинг 4.1).

Лістинг 4. 1 Схема GraphQL для модуля User

```
type User {
  id: String!
```

```
  firstName: String!
  lastName: String!
  email: String!
  nickName: String!
  createdAt: DateTime!
  updatedAt: DateTime!
  posts: [Post!]!
  comments: [Comment!]!
}

type Mutation {
  createUser(createUser: CreateUserDto!): User!
  updateUser(id: String!, updateUser: UpdateUserDto!): User!
}

input CreateUserDto {
  firstName: String!
  lastName: String!
  email: String!
  nickName: String!
}

input UpdateUserDto {
  firstName: String!
  lastName: String!
  email: String!
  nickName: String!
}
```

Запити (Queries) – запити в GraphQL дозволяють клієнтам формувати запити з точними даними, які вони хочуть отримати. Наприклад, клієнт може запитати firstName та email для всіх користувачів, без необхідності отримувати інші непотрібні дані. Це зменшує обсяг переданих даних і покращує швидкість відповіді.

Мутації (Mutations) **-** мутації дозволяють модифікувати дані в базі.

Асоціації між об'єктами – GraphQL дозволяє легко встановлювати асоціації між різними типами. Наприклад, у модулі User можна запитувати пов'язані Posts або Comments для конкретного користувача. Це забезпечує єдине місце для отримання всіх необхідних даних.

Реалізація резолверів – для обробки запитів та мутацій реалізовані резолвери (див. лістинг 4..2), які відповідають за отримання даних з бази. Резолвери взаємодіють з сервісами (див. лістинг 4..3), які в свою чергу

взаємодіють з моделями даних (див. лістинг 4..4) за допомогою ORM TypeORM для виконання запитів до бази даних та повернення результатів.

Лістинг 4.2 user resolver

```
@Resolver()
export class UserResolver {
  constructor(private userService: UserService) {}
  @Query(() => [User])
  async getUsers() {
    return this.userService.getAllUsers();
  }

  @Query((returns) => User)
  async getUserById(@Args('id') id: string) {
    return this.userService.getUserById(id);
  }

  @Mutation((returns) => User)
  async  createUser(@Args('createUser')  createUser:  CreateUserDto)
{
    return this.userService.createUser(createUser);
  }

  @Mutation((returns) => User)
  async updateUser(
    @Args('id') id: string,
    @Args('updateUser') updateUser: UpdateUserDto,
  ) {
    return this.userService.updateUser(id, updateUser);
  }
}
```

Лістинг 4.3 user service

```
@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private userRepository: Repository<User>,
  ) {}

  async getUserById(id: string): Promise<User> {
    const user = await this.userRepository.findOne({
      where: { id },
      relations: ['posts', 'comments'],
    });

    if (!user) {
      throw new NotFoundException({
        message: `User with id ${id} not found`,
      });
    }
```

```
    return user;
  }

  async getAllUsers(): Promise<User[]> {
    const users = await this.userRepository.find({
      relations: ['posts', 'comments'],
    });
    return users;
  }

  async createUser(createUserDto: CreateUserDto) {
    const user = await this.userRepository.findOne({
      where: [
        { email: createUserDto.email },
        { nickName: createUserDto.nickName },
      ],
    });

    if (user) {
      throw new BadRequestException('User already exists');
    }

    return this.userRepository.save(createUserDto);
  }

  async deleteUser(id: string) {
    const user = await this.userRepository.findOne({
      where: { id },
    });

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }

    await this.userRepository.delete(user);

    return `User with id ${id} has been deleted`;
  }

  async updateUser(id: string, dto: UpdateUserDto) {
    const user = await this.userRepository.findOne({
      where: { id },
    });

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }

    if (user.email === dto.email && user.nickName ===
dto.nickName) {
      throw new BadRequestException('This email or nickName
already exists');
    }
```

```
    await this.userRepository.update(dto, { id });

    return this.getUserById(id);
  }
}
```

Лістинг 4.4 модель даних для user

```
@ObjectType()
@Entity()
export class User {
  @Field()
  @PrimaryGeneratedColumn('uuid')
  id: string;
  @Field()
  @Column()
  firstName: string;
  @Field()
  @Column()
  lastName: string;

  @Field()
  @Column()
  email: string;
  @Field()
  @Column()
  nickName: string;
  @Field()
  @CreateDateColumn()
  createdAt: Date;
  @Field()
  @UpdateDateColumn()
  updatedAt: Date;
  @Field(() => [Post])
  @OneToMany(() => Post, (post) => post.author)
  posts: Post[];
  @Field(() => [Comment])
  @OneToMany(() => Comment, (comment) => comment.author)
  comments: Comment[];
}
```

4.3.2 Реалізація за допомоги REST архітектури

Структра модуля User побудованого на REST архітектурі (див. рис. 4.3).

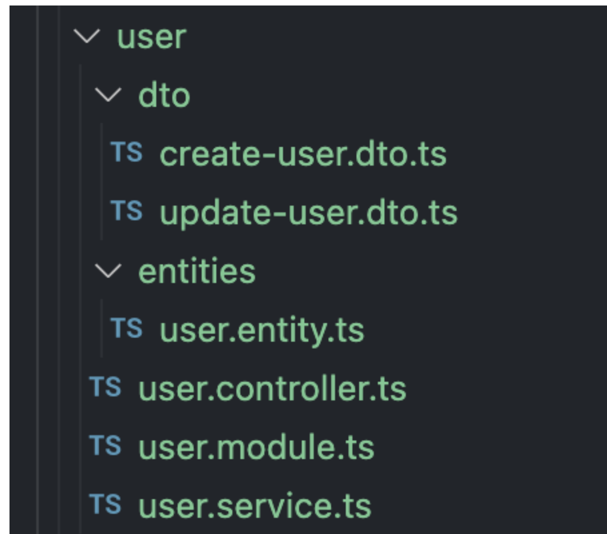

Рисунок 4.3 – Структура user модуля (архітектура REST)

Основні складові реалізації.

HTTP Методи: Для реалізації модулю User використовуються основні HTTP методи, які відповідають операціям CRUD (Create, Read, Update, Delete).

Всі ці методи реалізовані за допомогою controller (див. лістинг 4.5):

- POST: Для створення нового користувача. Наприклад, на запит до /users з даними нового користувача API створює нового користувача в базі даних;
- GET: Для отримання даних про користувачів. Запит до /users повертає список усіх користувачів, а запит до /users/{id} повертає інформацію про конкретного користувача;
- PUT: Для оновлення існуючого користувача. Наприклад, запит до /users/{id} з новими даними оновлює інформацію про користувача з указаним ідентифікатором;
- DELETE: Для видалення користувача. Запит до /users/{id} видаляє користувача з бази даних.

Реалізація REST API передбачає чітку та логічну структуру URL, наприклад:

- /users — для роботи з усіма користувачами.
- /users/{id} — для роботи з конкретним користувачем.

Лістинг 4.5 user controller

```
@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.userService.create(createUserDto);
  }

  @Get()
  findAll() {
    return this.userService.findAll();
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.userService.findOne(id);
  }

  @Patch(':id')
  update(@Param('id')    id:    string,    @Body()    updateUserDto:
UpdateUserDto) {
    return this.userService.update(id, updateUserDto);
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.userService.remove(id);
  }
}
```

REST архітектура використовує таку саму модель даних як і GRAPHQL (див. лістинг 4.4) та такий самий user service (див. лістинг 4.3)

Висновки до розділу 3

У цьому розділі було описано процес розробки програмного забезпечення для реалізації GraphQL і REST API, що стало основою для подальшого аналізу їх ефективності.

Вибір технологій.

Використання NestJS, TypeScript, Node.js і PostgreSQL забезпечило гнучкість, продуктивність і масштабованість системи. Застосування сучасних технологій, таких як TypeScript, дозволило покращити якість коду завдяки статичній типізації, а NestJS забезпечив інтуїтивну структуру модулів і підтримку різних архітектур.

Реалізація бази даних.

База даних була розроблена з урахуванням вимог до зберігання та обробки даних користувачів, постів і коментарів. Структура таблиць і зв'язків між сутностями забезпечила логічну ієрархію даних, яка відповідає вимогам системи.

Порівняння реалізацій.

Обидві реалізації використовують спільну модель даних, що спрощує порівняння їх продуктивності та зручності використання. GraphQL забезпечує більшу гнучкість у виборі даних для запитів, тоді як REST API пропонує більш простий і зрозумілий підхід для розробки.

Результати цього розділу закладають основу для подальшого дослідження продуктивності GraphQL і REST API, яке є основною метою даної роботи.

5 МЕТОДИКА ДОСЛІДЖЕННЯ

5.1 Опис тестового середовища

Для проведення дослідження ефективності GraphQL та REST API було розгорнуто тестове середовище з наступними характеристиками:

Апаратне забезпечення:

- локальна машина – Macbook M1 Pro;
- процесор – Apple M1 Pro (10 ядер);
- оперативна пам'ять – 16 GB;
- накопичувач – SSD.

Програмне забезпечення:

- операційна система – macOS Sequoia;
- Docker v4.23.0 для контейнеризації бази даних;
- база даних – PostgreSQL 14;
- сервер для REST API – Node.js;
- сервер для GraphQL – API Node.js з GraphQL.

База даних була розгорнута в Docker контейнері для забезпечення ізоляції та відтворюваності результатів.

Сервери REST API та GraphQL API були налаштовані на обробку запитів до відповідних баз даних.

Ниже буде приведено код для налаштування Docker для бази даних (див. лістинг 5.1).

Лістинг 5.1 – Налаштування Docker

```
version: '3.3'по
services:
   db:
    image: postgres:13.1-alpine
    restart: always
    ports:
      - 2432:5432
    environment:
      POSTGRES_PASSWORD: ‘db’
      POSTGRES_USER: ‘admin
      POSTGRES_DB: ‘admin’
    container_name: db
```

```
    volumes:
      - db:/var/lib/postgresql/data
```

5.2 Вибір метрик для аналізу ефективності

Час виконання запиту (duration).

Детальніше: Duration – це "час від початку до кінця" запиту. Він включає в себе не тільки час обробки на сервері, але й час, витрачений на передачу запиту від клієнта до сервера, а також час передачі відповіді назад до клієнта.

Важливість в контексті порівняння GraphQL та REST API**.** Duration є критично важливим показником з точки зору користувацького досвіду. GraphQL, завдяки своїй здатності отримувати тільки необхідні дані за один запит, може потенційно зменшити Duration, особливо в випадках, коли REST API вимагає кількох запитів для отримання тієї ж інформації. Однак, складні GraphQL запити можуть вимагати більше часу на обробку на сервері, що може збільшити Duration.

Час відповіді (response time).

Response Time фокусується виключно на часі, який сервер витрачає на обробку запиту. Це час від моменту, коли сервер отримує запит, до моменту, коли він відправляє відповідь.

Важливість в контексті порівняння GraphQL та REST API. Response Time дозволяє оцінити ефективність саме серверної частини API. GraphQL, завдяки своїй гнучкості, може дозволити оптимізувати запити до бази даних та зменшити Response Time. Однак, неправильно спроектована GraphQL схема може призвести до складних та неефективних запитів, що збільшить Response Time.

Оперативна пам'ять (RSS memory).

RSS (Resident Set Size) - це обсяг оперативної пам'яті, який фактично використовується процесом сервера в даний момент.

Важливість в контексті порівняння GraphQL та REST API. Порівняння RSS Memory для GraphQL та REST API може показати, яка з архітектур

ефективніше використовує ресурси сервера. GraphQL, завдяки своєму підходу "запит тільки того, що потрібно", може потенційно зменшити споживання пам'яті. Однак, складні GraphQL запити можуть вимагати більше пам'яті для обробки.

Завантаження процесора (CPU load).

CPU Load показує, наскільки завантажений центральний процесор обробкою запитів до API.

Важливість в контексті порівняння GraphQL та REST API. Порівняння CPU Load для GraphQL та REST API може показати, яка з архітектур вимагає більше обчислювальних ресурсів. GraphQL, завдяки своєму підходу до обробки запитів, може потенційно зменшити завантаження CPU. Однак, складні GraphQL запити можуть вимагати більше обчислень та збільшити завантаження CPU.

Кількість запитів на одиницю часу (HTTP Requests).

Кількість запитів, оброблених API під час тесту. Ця метрика допомагає оцінити пропускну здатність API за певний період.

Порівняння кількості оброблених запитів для GraphQL і REST API дозволяє зрозуміти, яка архітектура може краще обслуговувати одночасні запити. REST API зазвичай має перевагу в обробці великої кількості простих запитів, тоді як GraphQL більше підходить для складних та багаторівневих запитів.

Відсоток успішних запитів (Checks).

Частка успішно виконаних запитів (у відсотках). Це показник стабільності API.

Успішність запитів важлива для розуміння надійності системи. Якщо один із API має менший відсоток успішних запитів, це може свідчити про проблеми з інфраструктурою, серверною логікою або ресурсами.

Середній час запиту (AVG, с).

Середній час, необхідний API для обробки одного запиту. Це ключовий показник продуктивності API.

Порівняння середнього часу запиту для GraphQL та REST API дозволяє оцінити, яка архітектура є більш продуктивною. REST API зазвичай демонструє меншу затримку для простих запитів, тоді як GraphQL може бути повільнішим через складність обробки запитів із великою глибиною.

Запити за секунду (Requests per Second).

Кількість запитів, які API обробляє за одну секунду. Ця метрика характеризує пропускну здатність API.

Порівняння цієї метрики дозволяє зрозуміти, яке API краще справляється з великим навантаженням. REST API зазвичай демонструє більшу пропускну здатність для простих запитів, тоді як GraphQL може бути менш ефективним через складність запитів.

5.3 Опис тестових сценаріїв навантаження

Тестові сценарії були розроблені для порівняльного аналізу ефективності REST і GraphQL API при обробці одночасних запитів різного обсягу та структури даних. Було обрано п’ять основних ендпоінтів, які представляли різні комбінації запитів до бази даних для отримання користувачів та їх пов'язаного контенту (пости та коментарі). Кожен ендпоінт отримав 64 одночасних запити для моделювання навантаження, що дозволило оцінити продуктивність обох типів API в умовах інтенсивної роботи. Нижче наведено детальний опис кожного з ендпоінтів:

− ендпоінт 1: отримати всіх користувачів, їх коментарі та їх пости Цей ендпоінт був одним із найбільш навантажених, оскільки повертав повний набір даних про кожного користувача, включаючи пов'язані коментарі та пости. Мета тестування полягала в оцінці швидкодії API при обробці складних запитів, що включають кілька рівнів вкладеності. Це дозволяло оцінити, як обидва API справляються з витягуванням великої кількості пов’язаних даних з розгалуженою структурою. Очікувалося, що GraphQL може бути більш ефективним завдяки можливості обирати лише

необхідні поля, а REST може потребувати додаткових запитів через обмеження в обсязі даних, що повертаються за один раз;

- ендпоінт 2: Отримати всіх користувачів та їх коментарі. Цей ендпоінт надавав інформацію про користувачів разом із їх коментарями, але без постів. Мета цього тесту — виміряти ефективність API при роботі з даними середнього обсягу. Це також дозволяло оцінити, як GraphQL та REST обробляють запити з певними обмеженнями на глибину отриманих даних. В GraphQL вибірковість полів дозволяла зменшити обсяг даних, що передаються, і це очікувано мало пришвидшити обробку запитів у порівнянні з REST, де вибір полів не завжди налаштовується так гнучко;
- ендпоінт 3: Отримати всіх користувачів, їх пости та пов'язані з ними коментарі. Цей ендпоінт повертав кожного користувача разом із його постами, включаючи всі коментарі до кожного поста. Це дозволяло перевірити, як API обробляє дані з кількома рівнями залежностей. Для REST API часто необхідні додаткові запити для отримання вкладених даних, що збільшує навантаження. У GraphQL, навпаки, все можна витягнути в одному запиті, завдяки чому зменшується час обробки запиту та обсяг даних, що передаються мережею;
- ендпоінт 4: Отримати всіх користувачів та їх пости. Цей ендпоінт повертав лише користувачів і їхні пости, без коментарів. Це був важливий тест для перевірки продуктивності API при роботі з великим обсягом даних, але без додаткових вкладених залежностей. Для GraphQL це дозволяло налаштувати вибірку полів таким чином, щоб отримати лише основну інформацію, а REST API, у випадку відсутності фільтрації полів, обробляв та повертав більший обсяг даних;
- ендпоінт 5: Отримати всіх користувачів. Ендпоінт повертав лише основну інформацію про користувачів без їх коментарів та постів. Це дозволило оцінити продуктивність API на найпростіших запитах з мінімальною кількістю даних. Для GraphQL було

можливим обмежити вибірку полів до найнеобхідніших (наприклад, ідентифікатор, ім'я користувача), що, у свою чергу, мало пришвидшити роботу та зменшити трафік у порівнянні з REST, де обмеження вибірки даних також можливе, але менш гнучко налаштовується.

5.3.1 Аналіз продуктивності API під час обробки послідовних запитів

Для кожного ендпоінта було здійснено по 64 одночасних запити з метою створення високого навантаження та моделювання роботи API в умовах реального використання. Це дало можливість:

- виміряти час відгуку для кожного ендпоінта при одночасному обслуговуванні великої кількості запитів;
- оцінити використання ресурсів сервера (CPU, пам'ять) під час обробки одночасних запитів у різних сценаріях;
- проаналізувати швидкодію та масштабованість GraphQL та REST API, зокрема у випадках складних запитів з багаторівневими залежностями (як у ендпоінтах 1 та 3).

Результати тестування дозволять зібрати дані для аналізу продуктивності кожного API.

5.3.2 Моделювання сценаріїв стресового навантаження API

Також web-api будуть перевірені на стресостійкість за допомогою інструменту auto cannon.

k6 – це інструмент навантажувального тестування з відкритим вихідним кодом, призначений для тестування продуктивності та надійності додатків. Призначений переважно для розробників та інженерів з контролю якості, він використовується для тестування продуктивності та бенчмаркінгу, щоб забезпечити надійність та ефективність додатків [17].

Кожен з вище перерахованих ендпоінтів буде перевірений за двома сценаріями.

Перший сценарій – тривале навантаження зі збільшенням кількості запитів. Тест буде тривати 180 секунд, де 50 клієнтів будуть кожну секунду відправляти запит на сервер. Це дасть можливість перевірити, як система реагує на поступове збільшення навантаження протягом тривалого часу. Цей сценарій дозволяє виявити потенційні проблеми масштабованості, такі як зростання затримки, збільшення кількості помилок або перевантаження ресурсів.

Другий сценарій – коротке пікове навантаження з великою кількістю запитів. Тест буде тривати 30 секунд під час яких 120 клієнтів будуть робити по запиту кожну секунду. Цей сценарій дасть змогу перевірити як, система справляється з раптовим піковим навантаженням протягом короткого часу. Цей тест дозволяє виявити вузькі місця, які з'являються при перевантаженні системи.

5.4 Аналіз результатів послідовних запитів

Для порівняння продуктивності GraphQL та REST API було проведено тестування з використанням 5 різних endpoint-ів, на кожен з яких було здійснено по 64 запити. Було зібрано дані про чотири ключові метрики швидкодії: Duration (Тривалість), Response Size (Розмір відповіді), RSS Memory (Оперативна пам'ять) та CPU Load (Завантаження процесора). Нижче буду приведено 5 таблиць с середнім значенням кожного метрика для кожного endpoint`a.

Таблиця 5.1 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів, їх коментарі та їх пости» (послідовні запити)

REST-API GET Get all users with posts and comments		**GRAPHQL GET Get all users with posts and comments**	
1	2	3	4
Duration AVG (мс)	1528,4	Duration AVG (мс)	630,06

Продвоження таблиці 5.1

1	2	3	4
Response Size AVG (МБ)	11,28	Response Size AVG (МБ)	10,42
RSS Memory AVG (МБ)	580,41	RSS Memory AVG (МБ)	298,59
Heap Total Avg (МБ)	538,44	Heap Total Avg (МБ)	236,12
Heap Used Avg (МБ)	477,6	Heap Used Avg (МБ)	197,5
External Memory AVG (МБ)	24,66	External Memory AVG МБ)	26,76
CPU Load AVG (%)	16,75	CPU Load AVG (%)	6,33

Таблиця 5.2 – Зібрані метрики після викоання енпдоінту «Отримати всіх користувачів та їх коментарі» (послідовні запити)

REST-API GET Get all users with comments		**GRAPHQL GET Get all users with comments**	
Duration AVG (мс)	424,71	Duration AVG (мс)	526,2
Response Size AVG (МБ)	10,32	Response Size AVG (МБ)	9,5
RSS Memory AVG (МБ)	304	RSS Memory AVG (МБ)	333,33
Heap Total Avg (МБ)	206,93	Heap Total Avg (МБ)	242,67
Heap Used Avg (МБ)	165,86	Heap Used Avg (МБ)	193,15
External Memory AVG (МБ)	28,16	External Memory AVG (МБ)	24,2
CPU Load AVG (%)	5,78	CPU Load AVG (%)	6,66

Таблиця 5.3 – Зібрані метрики після викоання енпдоінту «Отримати всіх користувачів, їх пости та пов'язані з ними коментарі» (послідовні запити)

REST-API GET Get all users with comments		**GRAPHQL GET Get all users with comments**	
1	2	3	4
Duration AVG (мс)	576,03	Duration AVG (мс)	951,65
Response Size AVG (МБ)	11,3	Response Size AVG (МБ)	10,44
RSS Memory AVG (МБ)	343,25	RSS Memory AVG (МБ)	356,48
Heap Total Avg (МБ)	253,86	Heap Total Avg (МБ)	280,92
Heap Used Avg (МБ)	202,18	Heap Used Avg (МБ)	226,34

Продожвення таблиці 5.3

1	2	3	4
External Memory AVG (МБ)	27,74	External Memory AVG МБ)	23,38
CPU Load AVG (%)	7,2	CPU Load AVG (%)	8,88

Таблиця 5.4 – Зібрані метрики після виконання енпдоінту «Отримати всіх користувачів та їх пости» (послідовні запити)

REST-API GET Get all users with comments		**GRAPHQL GET Get all users with comments**	
Duration AVG (мс)	34,62	Duration AVG (мс)	87,64
Response Size AVG (МБ)	1,07	Response Size AVG (МБ)	1,03
RSS Memory AVG (МБ)	196,42	RSS Memory AVG (МБ)	235,49
Heap Total Avg (МБ)	132,73	Heap Total Avg (МБ)	187,11
Heap Used Avg (МБ)	97,74	Heap Used Avg (МБ)	145,67
External Memory AVG (МБ)	16,51	External Memory AVG МБ)	12,13
CPU Load AVG (%)	5,04	CPU Load AVG (%)	5,75

Таблиця 5.5 – Зібрані метрики після виконання енпдоінту «Отримати всіх користувачів» (послідовні запити)

REST-API GET Get all users with comments		**GRAPHQL GET Get all users with comments**	
Duration AVG (мс)	9,04	Duration AVG (мс)	11,72
Response Size AVG (МБ)	0,11	Response Size AVG (МБ)	0,11
RSS Memory AVG (МБ)	136,21	RSS Memory AVG (МБ)	173,01
Heap Total Avg (МБ)	74,68	Heap Total Avg (МБ)	103,32
Heap Used Avg (МБ)	50,94	Heap Used Avg (МБ)	75,37
External Memory AVG (МБ)	7,72	External Memory AVG МБ)	6,5
CPU Load AVG (%)	6,78	CPU Load AVG (%)	7,82

Аналіз зібраних даних показує, що REST API в цілому демонструє кращу продуктивність, ніж GraphQL API, за більшістю тестованих сценаріїв та

метрик. Зокрема, REST API має перевагу в швидкості обробки запитів (Duration), використанні оперативної пам'яті (RSS Memory) та завантаженні процесора (CPU Load).

Детальний аналіз за endpoint-ами.

Отримати всіх користувачів з постами та коментарями. В цьому сценарії GraphQL API показав значно кращу продуктивність за всіма метриками. Це може свідчити про те, що GraphQL ефективніше обробляє складні запити, де потрібно отримати дані з кількох пов'язаних таблиць.

- duration: GraphQL API значно швидше обробляє запит (630,06 мс), ніж REST API (1528,4 мс);
- response Size: Розмір відповіді GraphQL API (10,42) трохи менший, ніж у REST API (11,28);
- RSS Memory: GraphQL API використовує значно менше оперативної пам'яті (298,59 МБ) порівняно з REST API (580,41 МБ);
- CPU Load: GraphQL API також має значно менше завантаження процесора (6,33%) порівняно з REST API (16,75%).

Отримати всіх користувачів з коментарями. В цьому сценарії REST API демонструє кращу продуктивність за Duration, RSS Memory та CPU Load, але GraphQL має менший Response Size.

- duration: REST API швидше обробляє запит (424,71 мс) порівняно з GraphQL API (526,2 мс);
- response Size: GraphQL API повертає менший обсяг даних (9,5) порівняно з REST API (10,32);
- RSS Memory: REST API використовує менше оперативної пам'яті (304 МБ) порівняно з GraphQL API (333,33 МБ);
- CPU Load: REST API також має менше завантаження процесора (5,78%) порівняно з GraphQL API (6,66%).

Отримати всіх користувачів з постами та пов'язаними коментарями. REST API показує кращу продуктивність за Duration, RSS Memory та CPU Load, але GraphQL має менший Response Size.

– duration: REST API знову швидше (576,03 мс) порівняно з GraphQL API (951,65 мс);
– response Size: GraphQL API повертає дані меншого розміру (10,44) порівняно з REST API (11,3);
– RSS Memory: REST API використовує менше оперативної пам'яті (343,25 МБ) порівняно з GraphQL API (356,48 МБ);
– CPU Load: REST API має менше завантаження процесора (7,2%) порівняно з GraphQL API (8,8%).

Отримати всіх користувачів з постами. REST API знову демонструє кращу продуктивність за більшістю метрик.

– duration: REST API значно швидше (34,62 мс) порівняно з GraphQL API (87,64 мс);
– response Size: Розмір відповіді практично однаковий для обох API (1,07 для REST та 1,03 для GraphQL);
– RSS Memory: REST API використовує менше оперативної пам'яті (196,42 МБ) порівняно з GraphQL API (235,49 МБ);
– CPU Load: REST API має трохи менше завантаження процесора (5,04%) порівняно з GraphQL API (5,75%).

Отримати всіх користувачів. REST API знову показує кращу продуктивність.

– duration: REST API швидше (9,04 мс) порівняно з GraphQL API (11,72 мс);
– response Size: Розмір відповіді однаковий для обох API (0,11);
– RSS Memory: REST API використовує менше оперативної пам'яті (136,21 МБ) порівняно з GraphQL API (173,01 МБ);
– CPU Load: REST API має менше завантаження процесора (6,78%) порівняно з GraphQL API (7,82%).

5.4.1 Графічне представлення результатів послідовних запитів

Для наочного порівняння ефективності REST і GraphQL API були побудовані два основних графіки.

Графік навантаження CPU в залежності від кількості запитів (див. рис. 5.1 - 5.5).

Цей графік демонструє залежність між кількістю запитів і навантаженням на центральний процесор (CPU) для кожного типу API.

На осі X відображена кількість запитів, яка поступово збільшується для моделювання умов зростаючого навантаження. На осі Y показано рівень завантаження CPU (%), який вказує на те, як обидва API справлялися з обробкою запитів при різному навантаженні. Графік дозволяє побачити, який з API краще оптимізований з точки зору використання обчислювальних ресурсів і як змінюється навантаження на CPU при збільшенні кількості запитів.

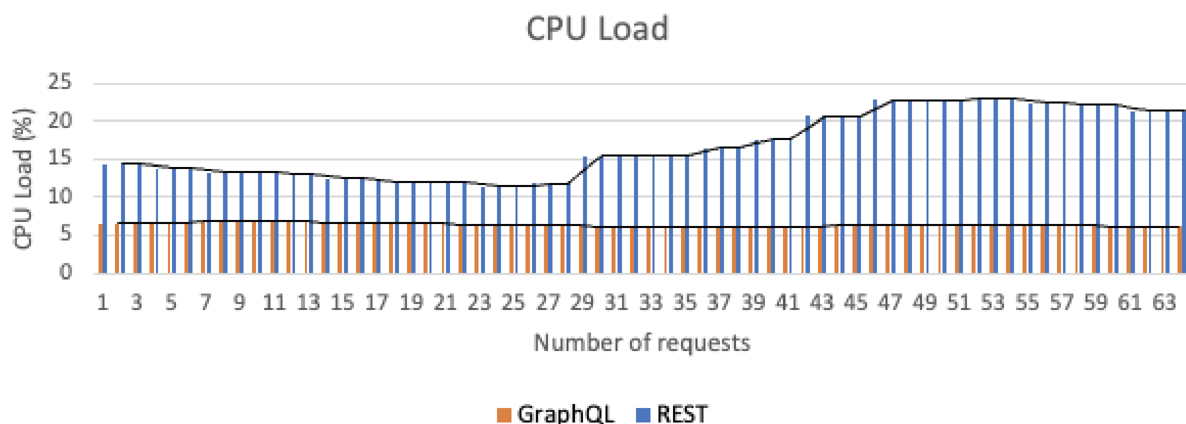

Рисунок 5.1 – Графік навантаженості на CPU під час виконання ендпоінту «Отримати всіх користувачів, їх коментарі та їх пости»

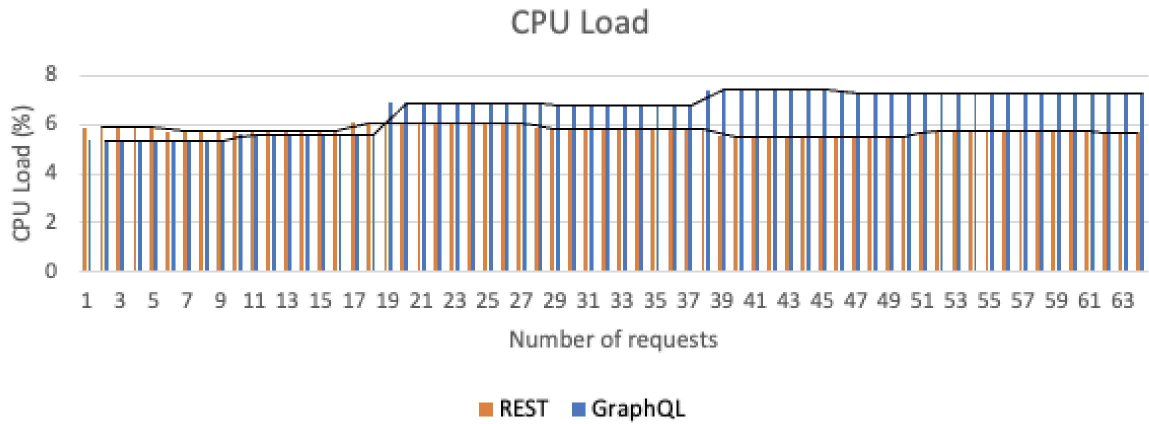

Рисунок 5.2 – Графік навантаженості на CPU під час виконання ендпоінту «Отримати всіх користувачів та їх коментарі»

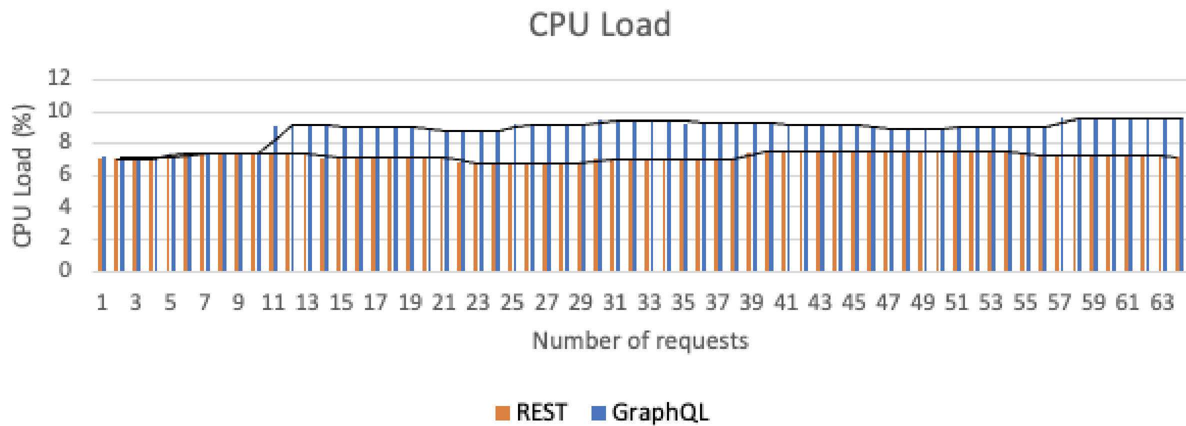

Рисунок 5.3 – Графік навантаженості на CPU під час виконання ендпоінту «Отримати всіх користувачів, їх пости та пов’язані з ними коментарі»

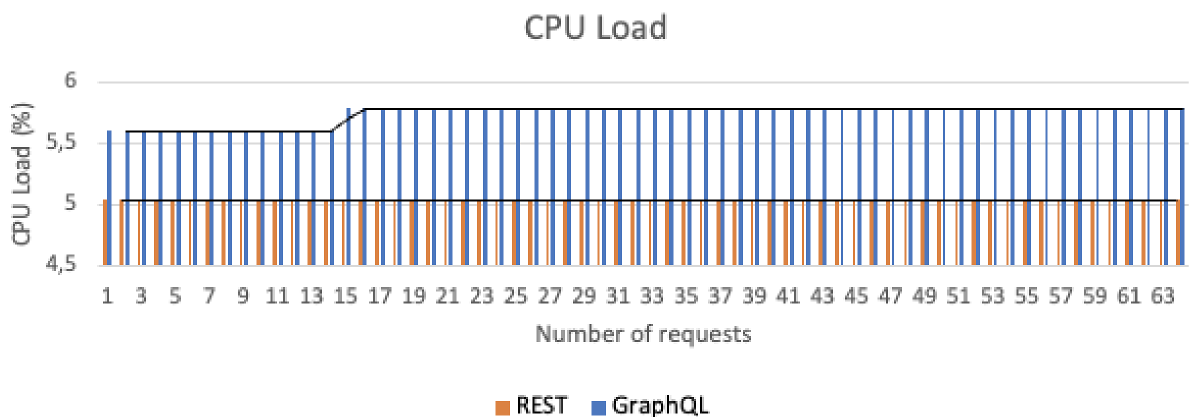

Рисунок 5.4 – Графік навантаженості на CPU під час виконання ендпоінту «Отримати всіх користувачів та їх пости»

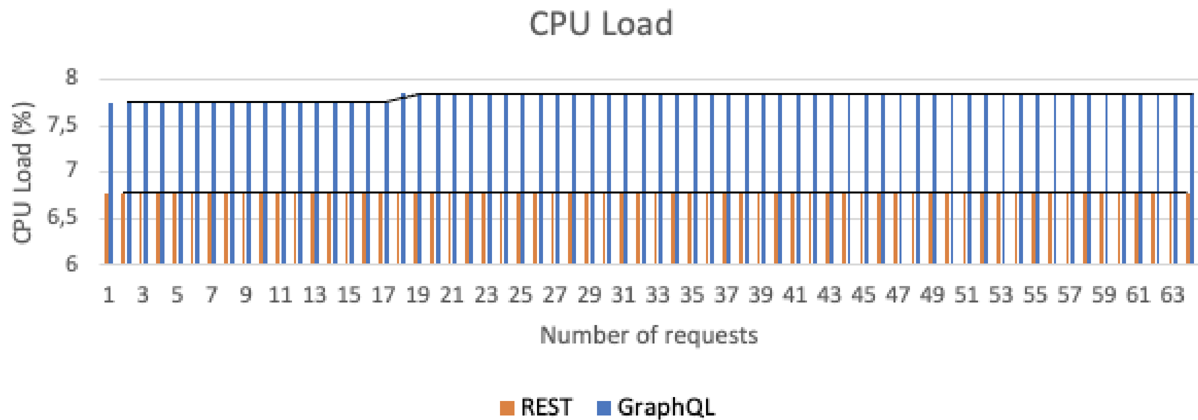

Рисунок 5.5 - Графік навантаженості на CPU під час виконання ендпоінту «Отримати всіх користувачів»

Результати можуть показати, що GraphQL API вимагає більше CPU через складніші обчислення на сервері при виконанні вкладених запитів, у той час як REST API має більш стабільне навантаження при збільшенні кількості простих запитів. З іншого боку, GraphQL може демонструвати кращу ефективність у випадках з великими запитами, що зменшує кількість необхідних окремих звернень до API.

Графік залежності витрат RSS пам’яті в залежності від часу виконання запиту (див. рис. 5.5 – 5.10).

На другому графіку показано залежність між використанням оперативної пам’яті та часом, витраченим на обробку запиту.

На осі X представлено обсяг використаної пам'яті (mb), а на осі Y — середній час, який витрачається на виконання одного запиту (ms). Цей графік дозволяє проаналізувати, наскільки ефективно кожен API використовує пам'ять при обробці запитів різного обсягу і складності.

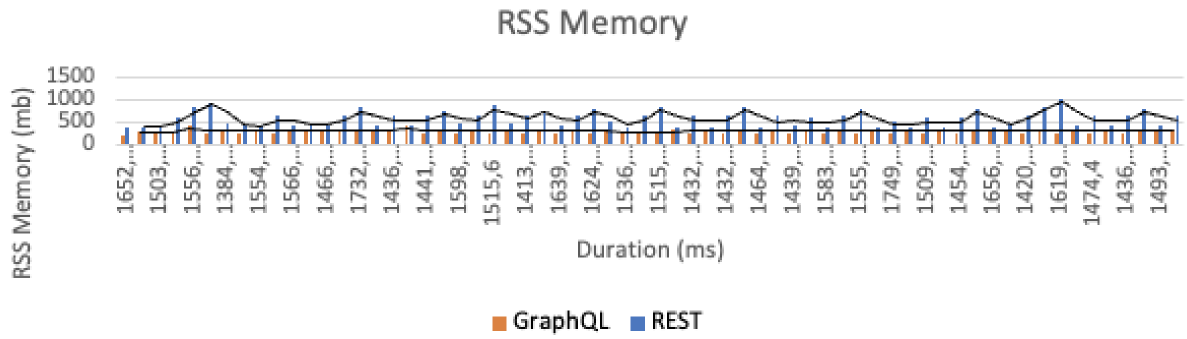

Рисунок 5.6 – Графік навантаженості на RSS memory під час виконання ендпоінту «Отримати всіх користувачів, їх коментарі та їх пости»

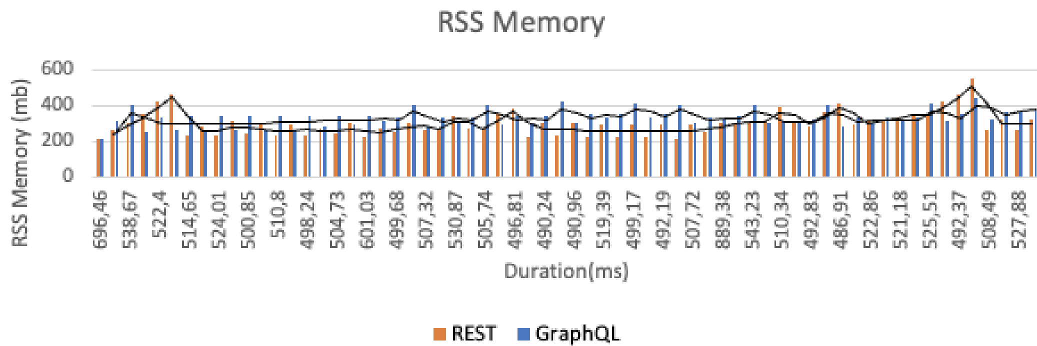

Рисунок 5.7 – Графік навантаженості на RSS memory під час виконання ендпоінту «Отримати всіх користувачів та їх коментарі»

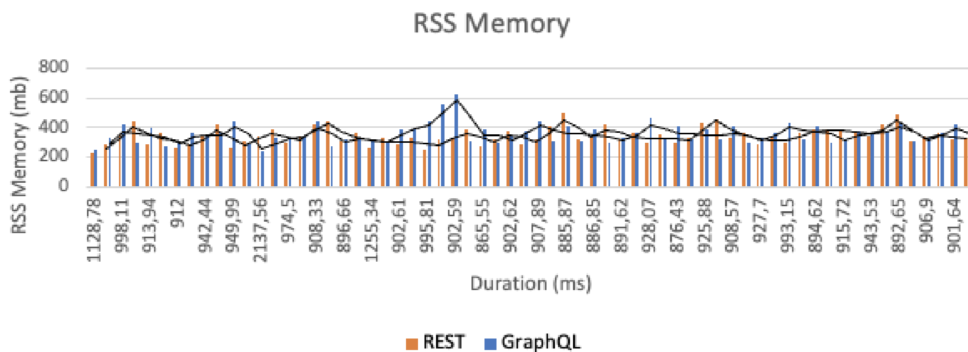

Рисунок 5.8 – Графік навантаженості на RSS memory під час виконання ендпоінту «Отримати всіх користувачів, їх пости та пов’язані з ними коментарі»

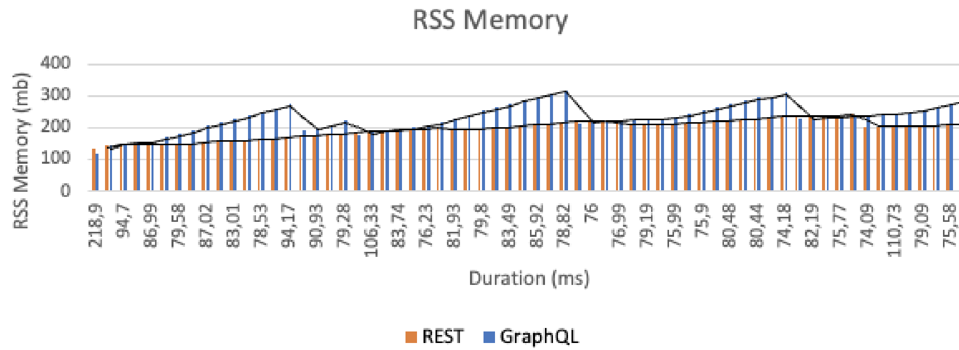

Рисунок 5.9 – Графік навантаженості на RSS memory під час виконання ендпоінту «Отримати всіх користувачів та їх пости»

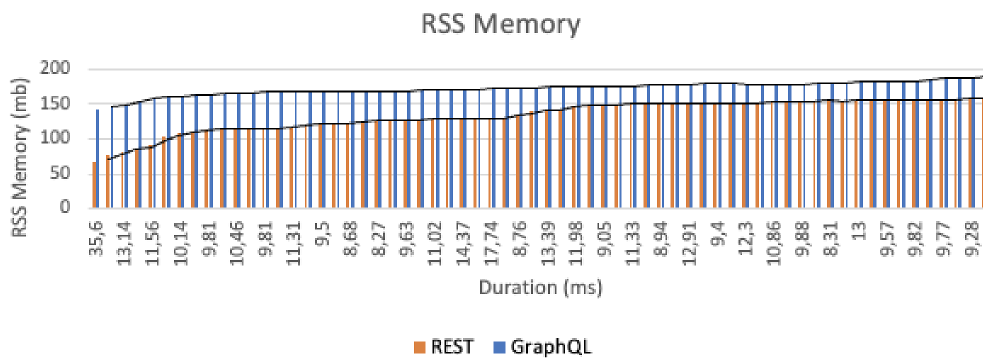

Рисунок 5.10 – Графік навантаженості на RSS memory під час виконання ендпоінту «Отримати всіх користувачів»

На графіках можна побачити, що GraphQL API може споживати більше пам'яті при запитах з великою кількістю пов'язаних об'єктів через складну обробку даних на сервері. Проте, завдяки можливості вибору лише потрібних полів, GraphQL може скоротити час обробки складних запитів у порівнянні з REST, де кожен запит часто повертає всі поля ресурсу.

5.5 Аналіз результатів стресостійкості

5.5.1 Аналіз пікового навантаження

Для порівняння продуктивності GraphQL та REST API було проведено тестування в умовах пікового навантаження, що тривало 30 секунд. На кожен із 5 різних endpoint-ів було здійснено велику кількість одночасних запитів, щоб оцінити здатність API витримувати високі навантаження.

Таблиця 5.6 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів, їх коментарі та їх пости» (пікове навантаження)

Метрика	**REST API**	**GraphQL API**
Кількість запитів (http_reqs)	20	41
Відсоток успішних запитів	100%	100%
Середній час запиту (avg, c)	37,51	37,67
Запити за секунду	0,333	0,683

Таблиця 5.7 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів та їх коментарі» (пікове навантаження)

Метрика	**REST API**	**GraphQL API**
Кількість запитів (http_reqs)	80	70
Відсоток успішних запитів	100%	100%
Середній час запиту (avg, c)	34,07	33,08
Запити за секунду	1,333	1,167

Таблиця 5.8 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів, їх пости та пов’язані з ними коментарі» (пікове навантаження)

Метрика	**REST API**	**GraphQL API**
Кількість запитів (http_reqs)	71	11
Відсоток успішних запитів	100%	100%
Середній час запиту (avg, c)	32,68	50,53
Запити за секунду	1,183	0,183

Таблиця 5.9 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів та їх пости» (пікове навантаження)

Метрика	**REST API**	**GraphQL API**
Кількість запитів (http_reqs)	679	100
Відсоток успішних запитів	100%	100%
Середній час запиту (avg, c)	4,73	39,85
Запити за секунду	19,865	2,265

Таблиця 5.10 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів» (пікове навантаження)

Метрика	REST API	GraphQL API
Кількість запитів (http_reqs)	6964	3636
Відсоток успішних запитів	100%	100%
Середній час запиту (avg, с)	0,433	0,836
Запити за секунду	228,853	117,895

Аналіз результатів тестування за сценаріями пікового навантаження демонструє, що REST API у більшості випадків перевершує GraphQL API за ключовими метриками продуктивності, такими як швидкість обробки запитів (duration), кількість оброблених запитів (http_reqs) та обсяг даних, отриманих у відповідь. Водночас GraphQL API демонструє високу ефективність у сценаріях з великим обсягом даних, завдяки своїй гнучкості у виборі полів.

Детальний аналіз за endpoint-ами.

Отримати всіх користувачів, їх коментарі та їх пости:

- середній час запиту (duration): REST API та GraphQL API мають подібну продуктивність (37,51 с проти 37,67 с). Це свідчить про те, що для складних запитів, які включають багато рівнів залежностей, обидва API працюють на схожому рівні;
- кількість запитів: GraphQL API обробив у два рази більше запитів (41 проти 20). Це демонструє кращу масштабованість GraphQL для такого типу запитів;
- запити за секунду: GraphQL API перевершує REST API (0,683 проти 0,333), що свідчить про більшу пропускну здатність.

Отримати всіх користувачів та їх коментарі:

- середній час запиту (duration): GraphQL API працює трохи швидше (33,08 с проти 34,07 с), що свідчить про його оптимізацію для запитів середньої складності;
- кількість запитів: REST API обробив трохи більше запитів (80 проти 70), демонструючи кращу ефективність у цьому сценарії;

- запити за секунду: REST API має трохи вищу пропускну здатність (1,333 проти 1,167).

Отримати всіх користувачів, їх пости та пов'язані коментарі:

- середній час запиту (duration): REST API значно швидший (32,68 с проти 50,53 с), демонструючи перевагу у швидкості обробки запитів з високою складністю;
- кількість запитів: REST API обробив значно більше запитів (71 проти 11), що вказує на його кращу масштабованість;
- запити за секунду: REST API має значно вищу пропускну здатність (1,183 проти 0,183).

Отримати всіх користувачів та їх пости:

- середній час запиту (duration): REST API працює значно швидше (4,73 с проти 39,85 с), що вказує на його ефективність у простих запитах;
- кількість запитів: REST API обробив значно більше запитів (679 проти 100);
- запити за секунду: REST API має значно вищу пропускну здатність (19,865 проти 2,265).

Отримати всіх користувачів

- середній час запиту (duration): REST API є швидшим (0,433 с проти 0,836 с), що свідчить про його оптимізацію для найпростіших запитів;
- кількість запитів: REST API обробив майже вдвічі більше запитів (6964 проти 3636), що демонструє його високу продуктивність;
- запити за секунду: REST API має значно більшу пропускну здатність (228,853 проти 117,895).

5.5.2 Аналіз тривалого навантаження

Аналіз результатів тестування за сценаріями тривалого навантаження демонструє, що REST API має перевагу за більшістю ключових метрик продуктивності, таких як кількість оброблених запитів (http_reqs), середній час обробки запитів (duration) та пропускна здатність (requests per second). GraphQL API, у свою чергу, продемонстрував кращу продуктивність у певних сценаріях, особливо за складних запитів з багаторівневою структурою даних.

Таблиця 5.11 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів, їх коментарі та їх пости «(тривале навантаження)

Метрика	REST API	GraphQL API
Кількість запитів (http_reqs)	160	225
Відсоток успішних запитів	12.5%	100%
Середній час запиту (avg, с)	56,87	42,19
Запити за секунду	0,762	1,071

Таблиця 5.12 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів та їх коментарі» (тривале навантаження)

Метрика	REST API	GraphQL API
Кількість запитів (http_reqs)	369	297
Відсоток успішних запитів	100%	100%
Середній час запиту (avg, с)	26,38	32,37
Запити за секунду	1,795	1,414

Таблиця 5.13 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів, їх пости та пов’язані з ними коментарі» (тривале навантаження)

Метрика	REST API	GraphQL API
Кількість запитів (http_reqs)	270	162
Відсоток успішних запитів	100%	24.69%
Середній час запиту (avg, с)	35,35	54,7
Запити за секунду	1,286	0,771

Таблиця 5.14 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів та їх пости» (тривале навантаження)

Метрика	REST API	GraphQL API
Кількість запитів (http_reqs)	5605	1726
Відсоток успішних запитів	100%	100%

Середній час запиту (avg, с)	1,61	5,27
Запити за секунду	30,871	9,385

Таблиця 6.15 – Зібрані значення метрик після викоання енпдоінту «Отримати всіх користувачів» (тривале навантаження)

Метрика	**REST API**	**GraphQL API**
Кількість запитів (http_reqs)	40288	20678
Відсоток успішних запитів	100%	100%
Середній час запиту (avg, с)	0,223	0,436
Запити за секунду	223,575	114,591

Детальний аналіз за endpoint-ами.

Отримати всіх користувачів, їх коментарі та їх пости:

– середній час запиту (duration): REST API і GraphQL API мають подібну продуктивність (36,52 с проти 36,78 с). Це свідчить про їхню рівноцінну оптимізацію для складних запитів;
– кількість запитів: GraphQL API обробив більше запитів (96 проти 80), демонструючи кращу масштабованість у цьому сценарії;
– запити за секунду: GraphQL API трохи перевершує REST API (0,533 проти 0,444).

Отримати всіх користувачів та їх коментарі:

– середній час запиту (duration): GraphQL API працює трохи швидше (34,21 с проти 34,89 с), що демонструє його оптимізацію для запитів середньої складності;
– кількість запитів: REST API обробив більше запитів (150 проти 120), що свідчить про його більшу ефективність у цьому сценарії;
– запити за секунду: REST API має вищу пропускну здатність (0,833 проти 0,667).

Отримати всіх користувачів, їх пости та пов’язані коментарі:

– середній час запиту (duration): REST API значно швидший (32,83 с проти 50,24 с), що демонструє його ефективність у складних запитах з великим обсягом даних;

- кількість запитів: REST API обробив набагато більше запитів (120 проти 50), що свідчить про його більшу масштабованість;
- запити за секунду: REST API має вищу пропускну здатність (0,667 проти 0,278);

Отримати всіх користувачів та їх пости:

- середній час запиту (duration): REST API демонструє значно кращу продуктивність (7,52 с проти 42,75 с), вказуючи на його ефективність у обробці простих запитів;
- кількість запитів: REST API обробив більше запитів (780 проти 120), що свідчить про його вищу продуктивність;
- запити за секунду: REST API значно перевершує GraphQL API (4,333 проти 0,667).

Отримати всіх користувачів:

- середній час запиту (duration): REST API значно швидший (0,533 с проти 0,942 с), що свідчить про його оптимізацію для простих запитів;
- кількість запитів: REST API обробив утричі більше запитів (8500 проти 2835);
- запити за секунду: REST API має набагато більшу пропускну здатність (47,222 проти 15,750).

Висновки до розділу 5

У п’ятому розділі було проведено порівняльний аналіз продуктивності REST API та GraphQL API за допомогою експериментальних тестів у різних сценаріях навантаження. Результати дослідження продемонстрували відмінності в ефективності кожної архітектури залежно від складності запитів та обсягу переданих даних. REST API показав кращу продуктивність у простих сценаріях із низьким навантаженням, тоді як GraphQL API забезпечив перевагу у випадках складних запитів із багаторівневими залежностями. Обидві архітектури мають свої сильні та слабкі сторони, що дозволяє сформувати чіткі рекомендації для їх вибору залежно від вимог проєкту.

ВИСНОВКИ

Результати дослідження дозволили виявити ключові відмінності у продуктивності REST API та GraphQL API, які є актуальними для вибору архітектури в різних проєктах. REST API, зокрема, демонструє кращу продуктивність у запитах із великим навантаженням, що пов’язано з його більш простою архітектурою та ефективністю обробки послідовних запитів. GraphQL API, навпаки, виявляє свої переваги у складних запитах, де потрібна взаємодія з великим обсягом взаємопов’язаних даних, завдяки своїй здатності запитувати лише необхідні поля та зменшувати обсяг переданої інформації.

Одним із важливих аспектів цього дослідження є аналіз часу обробки запитів. REST API показав стабільно менший середній час виконання запитів у більшості сценаріїв, особливо в умовах високої інтенсивності запитів. Цей фактор робить REST API кращим вибором для систем, де важлива швидка реакція сервера на користувацькі дії, наприклад, у реальному часі. Водночас GraphQL API, хоча і поступається за швидкістю обробки, демонструє перевагу в ефективності використання ресурсів у складних запитах із багаторівневими залежностями.

Пропускна здатність також є важливим показником, що впливає на загальну продуктивність. REST API забезпечує вищу пропускну здатність у простих

запитах, що дозволяє йому обробляти більшу кількість запитів за одиницю часу. GraphQL API, хоча і має нижчу пропускну здатність, виграє у сценаріях, де необхідно обмежити надмірну передачу даних або знизити навантаження на мережу. Це підкреслює його цінність для проєктів із високими вимогами до оптимізації передачі даних.

Проведене дослідження також дозволило виявити можливі напрямки для оптимізації обох підходів. REST API може бути вдосконалений шляхом впровадження кешування, використання компресії даних та оптимізації структури відповідей. Для GraphQL API рекомендується використовувати механізми обмеження складності запитів, а також упроваджувати засоби контролю навантаження на сервер.

Загалом, результати показали, що обидва підходи мають свої переваги та недоліки, і їх вибір залежить від конкретного проєкту. REST API є кращим вибором для систем із високою інтенсивністю запитів, тоді як GraphQL API демонструє свої переваги у складних структурах даних, що потребують гнучкості. Подальші дослідження можуть бути зосереджені на аналізі енергоспоживання, безпеки, а також на вивченні продуктивності в умовах реальних сценаріїв використання, щоб ще краще зрозуміти їх переваги та обмеження.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. AWS. Що таке RESTful API [Електронний ресурс] // Amazon Web Services. URL: https://aws.amazon.com/ru/what-is/restful-api/ (дата звернення: 02.12.2024).
2. Foxminded. Що таке REST API [Електронний ресурс] // Foxminded. URL: https://foxminded.ua/shcho-take-rest-api/ (дата звернення: 05.12.2024).
3. Foxminded. GraphQL: що це? [Електронний ресурс] // Foxminded. URL: https://foxminded.ua/graphql-shcho-tse/ (дата звернення: 07.12.2024).
4. Unite.ai. Що таке GraphQL [Електронний ресурс] // Unite.ai. URL: https://www.unite.ai/uk/graphql (дата звернення: 10.12.2024).
5. AWS. Порівняння GraphQL і REST [Електронний ресурс] // Amazon Web Services. URL: https://aws.amazon.com/ru/compare/the-difference-between-graphql-and-rest/ (дата звернення: 12.12.2024).
6. ItBlog. Чим відрізняється REST API від GraphQL [Електронний ресурс] // ItBlog. URL: https://it-blog.in.ua/chym-vidriznyayetsya-restapi-vid-graphql/ (дата звернення: 14.12.2024).
7. Hygraph. GraphQL vs REST APIs [Електронний ресурс] // Hygraph. URL: https://hygraph.com/blog/graphql-vs-rest-apis (дата звернення: 16.12.2024).
8. Medium. REST vs GraphQL [Електронний ресурс] // Medium. URL: https://medium.com/3axislabs/rest-vs-graphql-50ea1a537e8b (дата звернення: 18.12.2024).
9. Массе, М. REST API Design Rulebook [Текст] / Марк Массе. – Yahoo Press, 2011. – 114 с.
10. Буна, С. GraphQL in Action [Текст] / Самер Буна. – Manning, 2021. – 384 с.
11. GoIT. REST API: що це, як працює, переваги [Електронний ресурс] // GoIT. URL: https://goit.global/ua/articles/rest-api-shcho-tse-iak-pratsiuie-perevahy-i-pryklady/ (дата звернення: 19.12.2024).
12. ProIT. Чому GraphQL є кращим вибором для створення мікросервісів [Електронний ресурс] // ProIT. URL: https://proit.ua/chomu-graphql-ie-krashchim-viborom-dlia-stvoriennia-mikrosiervisiv/ (дата звернення: 20.12.2024).

13. Foxminded. TypeScript: що це? [Електронний ресурс] // Foxminded. URL: https://foxminded.ua/typescript/ (дата звернення: 22.12.2024).
14. Dan.IT. Що таке Node.js [Електронний ресурс] // Dan.IT. URL: https://dan-it.com.ua/uk/blog/chto-jeto-takoe-node-js-prostymi-slovami/ (дата звернення: 24.12.2024).
15. Guru99. PostgreSQL: вступ до роботи [Електронний ресурс] // Guru99. URL: https://www.guru99.com/uk/introduction-postgresql.html (дата звернення: 26.12.2024).
16. NestJS. Офіційний сайт фреймворка NestJS [Електронний ресурс] // NestJS. URL: https://nestjs.com/ (дата звернення: 27.12.2024).
17. Medium. Посібник з тестування за допомогою K6 [Електронний ресурс] // Medium. URL: https://medium.com/@blackhorseya/a-comprehensive-guide-to-k6-testing-ccf5ea1a98ee#:~:text=k6%20is%20an%20open%2Dsource,ensure%20robust%20and%20efficient%20applications (дата звернення: 28.12.2024).

ДОДАТОК А

Технічне завдання

ЗАТВЕРДЖУЮ
Перший проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

APIAnalyzer

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01441-01-ЛЗ

Завідувач кафедри КІТ

_____________Вадим ГОРЯЧКІН

Керівник розробки

_________Олена КУРОП’ЯТНИК

Виконавець

__________Єгор МАДРЕРОСОВ

Нормоконтролер

__________Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.01441-01-ЛЗ

APIAnalyzer

Технічне завдання

44165850.01441-01-ЛЗ

Листів 12

2025

ЗМІСТ

А.1 ВСТУП…. 4
А.2 ПІДСТАВИ ДЛЯ РОЗРОБКИ 5
А.3 ПРИЗНАЧЕННЯ РОЗРОБКИ 6
А.4 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ 7
А.4.1 Вимоги до функціональних характеристик.. 7
А.4.2 Вимоги до надійності 7
А.4.3 Вимоги експлуатації 7
А.4.4 Вимоги до складу та параметрів технічних засобів 8
А.4.5 Вимоги до інформаційної та програмної сумісності.......................... 8
А.4.6 Вимоги до маркування і упаковки 8
А.4.7 Вимоги до транспортування та зберігання 9
А.5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ............................. 10
А.6 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ 11
А.7 ПОРЯДОК І КОНТРОЛЬ ПРИЙМАННЯ 12
А.8 БІБЛІОГРАФІЧНИЙ СПИСОК... 13

АНОТАЦІЯ

Документ 44165850.1431-01 12 01 «RequestTTRAnalyzer. Технічні вимоги». У рамках магістерської роботи розроблено програму для дослідження продуктивності RESTful API та GraphQL API. Технічне заваднння містить технічні вимоги до функціональних характеристик програмного забезпечення, яке забезпечує автоматизований збір метрик продуктивності, таких як середній час виконання запитів, стандартне відхилення, мінімальні та максимальні значення, обсяг відповіді та відсоток успішних запитів.

Також у додатку представлено вимоги до вхідних та вихідних даних, що включають параметри конфігурації запитів, кількість одночасних запитів і частоту їх виконання, а також формат збереження результатів у базі даних PostgreSQL та у файлах Excel.

А.1 ВСТУП

Сучасні веб-додатки значною мірою покладаються на архітектури веб-API для забезпечення інтеграції між клієнтськими та серверними частинами систем. Серед найбільш популярних підходів до проєктування веб-API сьогодні виділяються RESTful API та GraphQL API. Обидві архітектури мають свої переваги та недоліки, які можуть значно впливати на продуктивність та ефективність обробки запитів у різних умовах використання.

В рамках цього дипломного проєкту ставиться завдання розробки програмного забезпечення для оцінки продуктивності зазначених архітектур. Дослідження зосереджується на зборі й аналізі ключових метрик продуктивності, таких як час виконання запитів, використання ресурсів, обсяг переданих даних, стабільність роботи під навантаженням і масштабованість.

Розроблена програма повинна забезпечувати можливість моделювання реальних сценаріїв використання API, створення порівняльного аналізу продуктивності RESTful API та GraphQL API. Отримані результати будуть використані для визначення найбільш оптимальної архітектури API залежно від специфічних вимог до системи.

А.2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ від Сухого Констянтина Михайловича ректора Українського державного університету науки і технологій “Про призначення керівників та затвердження тем магістерських робіт” за спеціальністю 121 “Інженерія програмного забезпечення» факультету “Комп’ютерних технологій і систем” по кафедрі “Комп’ютерні інформаційні технології”.

Тема дипломної роботи – “Дослідження ефективності методів web Api архітектур GraphQL та RestApi”. Керівник – доцент Куроп’ятник О.С.

А.3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Функіональне призначення – розроблене програмне забезпечення спрямоване на дослідження продуктивності та порівняння ефективності архітектур RESTful API та GraphQL API в умовах реального навантаження. Воно забезпечує автоматизований збір ключових метрик продуктивності, таких як час виконання запитів, стабільність під навантаженням, обсяг отриманих даних і успішність обробки запитів. Використання інструмента K6 дозволяє моделювати різні сценарії навантаження, що дає змогу оцінити стійкість і масштабованість обох архітектур.

Експлуатаційне призначення – отримані дані записуються в Excel, забезпечуючи зручність їх аналізу для визначення оптимальних підходів до реалізації веб-API. Це дає змогу оцінювати переваги та недоліки обох архітектур у контексті конкретних вимог і сценаріїв використання.

А.4 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

А.4.1 Вимоги до функціональних характеристик

Програма повинна забезпечувати повну підтримку роботи з RESTful API та GraphQL API, надаючи можливість автоматизованого збору ключових метрик продуктивності. Серед цих метрик виділяються такі, як середній час виконання запитів, стандартне відхилення часу, обсяг переданих даних та відсоток успішних запитів. Ці дані мають бути організовані, збережені у у форматі Excel, що спрощує їх подальший аналіз і візуалізацію.

Система повинна забезпечувати можливість виконання однакових сценаріїв тестування для обох архітектур API, дозволяючи збирати метрики у реальному часі. Це дає змогу оцінювати ефективність кожної архітектури за однакових умов експлуатації. Крім того, результати тестування мають бути представлені у зручному форматі таблиць, що дозволяє наочно порівнювати продуктивність RESTful API та GraphQL API. Це забезпечує не лише технічний аналіз, але й спрощує ухвалення рішень щодо вибору архітектури для конкретних проєктів.

А.4.2 Вимоги до надійності

Програма повинна бути стійкою до помилок, забезпечуючи правильну обробку виняткових ситуацій. Система має демонструвати стабільну роботу навіть за умов високого навантаження, обробляючи до 500 запитів на хвилину без збоїв і значного зниження продуктивності. Важливою вимогою є збереження цілісності даних, зібраних у процесі збору метрик, що включає захист від втрати чи некоректного збереження інформації у випадку непередбачених збоїв.

А.4.3 Вимоги експлуатації

Програмне забезпечення повинно експлуатуватися на комп’ютерній техніці, яка відповідає стандартним умовам використання для офісних приміщень. Температура повітря у приміщенні повинна бути в межах від +18°C до +25°C, а відносна вологість – у діапазоні 40–60%, що дозволяє уникнути конденсації та статичної електрики, які можуть пошкодити обладнання. Для забезпечення стабільної роботи необхідно використовувати електроживлення з напругою 220–240 В і частотою 50 Гц, рекомендується також застосовувати джерела безперебійного живлення (UPS) для захисту техніки у разі перебоїв електропостачання.

А.4.4 Вимоги до складу та параметрів технічних засобів

База даних. Система повинна підтримувати реляційну базу даних PostgreSQL (версія 15 або вище).

Серверна частина. Використання сервера з можливістю розміщення NodeJS додатків.

Обсяг пам'яті. Мінімум 4 ГБ оперативної пам'яті для коректної роботи сервісу та бази даних при середньому навантаженні.

А.4.5 Вимоги до інформаційної та програмної сумісності

Програма повинна бути сумісною з TypeOrm для забезпечення зручної взаємодії з базою даних, що дозволяє абстрагуватися від конкретної СУБД і спрощує керування даними. Розробка повинна підтримувати сучасні стандарти передачі даних, включаючи HTTP 1.1/2 і JSON, що гарантує сумісність із іншими системами та сервісами. Також необхідна інтеграція з інструментами

моніторингу, такими як K6, для моделювання навантаження та збору аналітичних даних про продуктивність.

А.4.6 Вимоги до маркування і упаковки

Програма повинна супроводжуватися детальною документацією, яка охоплює процеси встановлення, налаштування та експлуатації, а також надає інструкції для оновлення та підтримки.

А.4.7 Вимоги до транспортування та зберігання

Програмне забезпечення має постачатися у вигляді архіву або контейнера (наприклад, Docker), що включає всі необхідні компоненти для його розгортання. Для запобігання втраті даних у разі збою рекомендується налаштувати регулярне створення резервних копій бази даних і конфігураційних файлів. Система повинна гарантувати безпечне зберігання даних, включаючи зібрані метрики запитів.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

Програмна документація представляє собою перелік наступних документів:

- текст програми;
- опис програми;

А.5 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

У табл. А.1 наведено стадія та етапи розробки.

Таблиця А.1 – Стадії та етапи розробки

Етап розробки	Завдання	Строки виконання
1. Планування та аналіз вимог	Вивчення предметної області, формулювання вимог до системи, розробка технічного завдання (ТЗ).	07.09.24
2. Проектування системи	Розробка архітектури програми на основі модульного підходу з використанням NestJS, проєктування структури бази даних PostgreSQL, підготовка специфікацій для REST API та GraphQL API, а також розробка middleware для збору метрик продуктивності.	10.09.24
3. Розробка основної функціональності	Реалізація API та сервісів для CRUD-операцій нтеграція бази даних, впровадження middleware для збору метрик.	15.09.24
4. Тестування та налагодження	Перевірка коректності збору метрик та роботи інструменту K6	21.09.24
6. Вдосконалення функціоналу	Покращення продуктивності системи, внесення змін на основі тестування	03.01.25

А.6 ПОРЯДОК І КОНТРОЛЬ ПРИЙМАННЯ

Контроль за виконанням роботи здійснює керівник розробки доц. Куроп’ятник О.С.

А.7 БІБЛІОГРАФІЧНИЙ СПИСОК

1. Шинкаренко, В. І. Інженерія програмного забезпечення: навчальний посібник для виконання магістерської роботи / В. І. Шинкаренко, О. В. Горбова, О. П. Іванов, В. О. Андрющенко, В. Я. Нечай; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 140 с.

ДОДАТОК Б

Текст програми

ЗАТВЕРДЖУЮ

Проректор
Українського державного університету науки і технології
Анатолій РАДКЕВИЧ

APIAnalyzer

Текст програми

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01441-01 12 01-ЛЗ

Завідувач кафедри КІТ

______________Вадим ГОРЯЧКІН

Керівник розробки

__________Олена КУРОП'ЯТНИК

Виконавець

___________Єгор МАДРЕРОСОВ

Нормоконтролер

____________Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.01441-01 12 01-ЛЗ

APIAnalyzer

Текст програми

1116130.01441-01 12 01-ЛЗ

Листів 16

2025

АНОТАЦІЯ

Документ 44165850.01441-01 12 01 «APIAnalyzer. Текст програми». У рамках магістерської роботи розроблено програму для дослідження продуктивності RESTful API та GraphQL методів, побудовану на основі Node.js із використанням фреймворка NestJS. Програма включає механізми збору метрик продуктивності, таких як середній час виконання запитів,, мінімальні та максимальні значення, розмір респонсу та відсоток успішних запитів і т.д. Для вимірювання метрик реалізовано middleware, яке інтегрується в процес обробки HTTP-запитів, і базу даних PostgreSQL для збереження данних для проведення досліджень.

ЗМІСТ

СТРУКТУРА ПРОГРАМИ……………………………………………5

Б.1 ТЕКСТ ПРОГРАМИ 6

СТРУКТУРА ПРОГРАМИ

Модулі:

- user.module;
- post.module;
- comment module;
- app.module.

Сервіси:

- user.service;
- post. service;
- comment service.

Контроллери:

- user.controller;
- post. controller;
- comment controller.

Проміжні сервіси (middleware):

- metrics.middleware.

Б.1 ТЕКСТ ПРОГРАМИ

main.ts (GraphQL)

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(5003);
}
bootstrap();

import { MiddlewareConsumer, Module, NestModule } from
'@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';
import { UserModule } from './user/user.module';
import { PostModule } from './post/post.module';
import { CommentModule } from './comment/comment.module';
import { User } from './user/graphql/User';
import { Post } from './post/graphql/Post';
import { Comment } from './comment/graphql/Comment';
```

app.module.ts (GraphQL)

```
@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: 'src/schema.gql',
    }),
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      port: 2432,
```

```
      username: 'ksharim',
      password: 'ksharim',
      database: 'graphql',
      entities: [User, Post, Comment],
      synchronize: true,
    }),
    UserModule,
    PostModule,
    CommentModule,
  ],
})
export class AppModule {}

user.service.ts (GraphQL)
import {
  BadRequestException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './graphql/User';
import { CreateUserDto } from './dto/createUser.dto.req';
import { UpdateUserDto } from './dto/updateUser.dto.req';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private userRepository: Repository<User>,
  ) {}

  async getUserById(id: string): Promise<User> {
    const user = await this.userRepository.findOne({
      where: { id },
      relations: ['posts', 'comments'],
```

```
  });

  if (!user) {
    throw new NotFoundException({
      message: `User with id ${id} not found`,
    });
  }

  return user;
}

async getAllUsers(): Promise<User[]> {
  const users = await this.userRepository.find({
    // relations: ['posts', 'comments'],x
  });
  return users;
}

async createUser(createUserDto: CreateUserDto) {
  const user = await this.userRepository.findOne({
    where: [
      { email: createUserDto.email },
      { nickName: createUserDto.nickName },
    ],
  });

  if (user) {
    throw new BadRequestException('User already exists');
  }

  return this.userRepository.save(createUserDto);
}

async deleteUser(id: string) {
  const user = await this.userRepository.findOne({
    where: { id },
```

```
    });

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }

    await this.userRepository.delete(user);

    return `User with id ${id} has been deleted`;
  }

  async updateUser(id: string, dto: UpdateUserDto) {
    const user = await this.userRepository.findOne({
      where: { id },
    });

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }

    if (user.email === dto.email && user.nickName === dto.nickName) {
      throw new BadRequestException('This email or nickName already exists');
    }

    await this.userRepository.update(dto, { id });

    return this.getUserById(id);
  }
}
```

user.module.ts (GraphQL)

```
import { UserResolver } from './graphql/user.resolver';
import { Post } from 'src/post/graphql/Post';
import { Comment } from 'src/comment/graphql/Comment';
import { TypeOrmModule } from '@nestjs/typeorm';
```

```
import { Module } from '@nestjs/common';
import { UserService } from './user.service';
import { User } from './graphql/User';

@Module({
  imports: [TypeOrmModule.forFeature([User, Post, Comment])],
  providers: [UserResolver, UserService],
  exports: [UserService],
})
export class UserModule {}

user.entity.ts (GraphQL)
import { Field, ObjectType } from '@nestjs/graphql';
import { Comment } from 'src/comment/graphql/Comment';
import { Post } from 'src/post/graphql/Post';
import {
  Column,
  CreateDateColumn,
  Entity,
  OneToMany,
  PrimaryGeneratedColumn,
  UpdateDateColumn,
} from 'typeorm';

@ObjectType()
@Entity()
export class User {
  @Field()
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Field()
  @Column()
  firstName: string;

  @Field()
```

```
  @Column()
  lastName: string;

  @Field()
  @Column()
  email: string;

  @Field()
  @Column()
  nickName: string;

  @Field()
  @CreateDateColumn()
  createdAt: Date;

  @Field()
  @UpdateDateColumn()
  updatedAt: Date;

  @Field(() => [Post])
  @OneToMany(() => Post, (post) => post.author)
  posts: Post[];

  @Field(() => [Comment])
  @OneToMany(() => Comment, (comment) => comment.author)
  comments: Comment[];
}
```

user.resolver.ts (GraphQL)

```
import {
  Args,
  Mutation,
  Parent,
  Query,
  ResolveField,
  Resolver,
```

```
} from '@nestjs/graphql';
import { User } from './User';
import { UserService } from '../user.service';
import { CreateUserDto } from '../dto/createUser.dto.req';
import { UpdateUserDto } from '../dto/updateUser.dto.req';
import { InjectRepository } from '@nestjs/typeorm';
import { Post } from 'src/post/graphql/Post';
import { Repository } from 'typeorm';
import { Comment } from 'src/comment/graphql/Comment';

@Resolver(() => User)
export class UserResolver {
  constructor(
    private userService: UserService,
    @InjectRepository(Post)
    private postRepository: Repository<Post>,
    @InjectRepository(Comment)
    private commentRepository: Repository<Comment>,
  ) {}
  @Query(() => [User])
  async getUsers() {
    return this.userService.getAllUsers();
  }

  @Query((returns) => User)
  async getUserById(@Args('id') id: string) {
    return this.userService.getUserById(id);
  }

  @Mutation((returns) => User)
  async  createUser(@Args('createUser')  createUser:  CreateUserDto)
{
    return this.userService.createUser(createUser);
  }

  @Mutation((returns) => User)
```

```
  async updateUser(
    @Args('id') id: string,
    @Args('updateUser') updateUser: UpdateUserDto,
  ) {
    return this.userService.updateUser(id, updateUser);
  }

  @ResolveField()
  async posts(@Parent() author: User) {
    const { id } = author;
    return this.postRepository.find({ where: { authorId: id } });
  }

  @ResolveField()
  async comments(@Parent() author: User) {
    const { id } = author;
    return this.commentRepository.find({ where: { userId: id } });
  }
}
```

metric.middleware.ts

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';
import { performance } from 'perf_hooks';
import * as os from 'os';
import * as ExcelJS from 'exceljs';
import * as fs from 'fs';

@Injectable()
export class MetricsMiddleware implements NestMiddleware {
  private workbook: ExcelJS.Workbook;
  private worksheet: ExcelJS.Worksheet;
  //get-users-with-comments.xlsx
  //get-users-with-posts-and-comments.xlsx
  //get-users-with-posts-and-related-comments.xlsx
  //get-users-with-posts.xlsx
```

```
  //get-users.xlsx
  private fileName = 'get-users.xlsx';

  constructor() {
    this.workbook = new ExcelJS.Workbook();
    this.worksheet = this.workbook.addWorksheet('Metrics');

    // Create headers if the file does not exist
    if (!fs.existsSync(this.fileName)) {
      this.worksheet.columns = [
        { header: 'Method', key: 'method', width: 15 },
        { header: 'URL', key: 'url', width: 50 },
        { header: 'Status', key: 'status', width: 10 },
        { header: 'Duration (ms)', key: 'duration', width: 15 },
        { header: 'Response Size (MB)', key: 'responseSize',
width: 15 },
        { header: 'RSS Memory (MB)', key: 'rssMemory', width: 15
},
        { header: 'Heap Total (MB)', key: 'heapTotal', width: 15
},
        { header: 'Heap Used (MB)', key: 'heapUsed', width: 15 },
        { header: 'External Memory (MB)', key: 'externalMemory',
width: 15 },
        { header: 'CPU Load (%)', key: 'cpuLoad', width: 15 },
      ];
    } else {
      this.loadExistingMetrics();
    }
  }

  use(req: Request, res: Response, next: NextFunction) {
    const startTime = performance.now();
    let responseSize = 0;

    const originalWrite = res.write;

    res.write = (chunk: any, ...args: any[]) => {
```

```
      responseSize += chunk.length;
      return originalWrite.apply(res, [chunk, ...args]);
    };

    res.on('finish', () => {
      const duration = performance.now() - startTime;
      const { method, originalUrl } = req;
      const { statusCode } = res;
      const contentLength = res.getHeader('Content-Length');
      const sizeInBytes = contentLength ? Number(contentLength) :
responseSize;
      responseSize = this.convertBytesToMB(sizeInBytes);

      const rssMemory =
this.convertBytesToMB(process.memoryUsage().rss);
      const heapTotal =
this.convertBytesToMB(process.memoryUsage().heapTotal);
      const heapUsed =
this.convertBytesToMB(process.memoryUsage().heapUsed);
      const externalMemory = this.convertBytesToMB(
        process.memoryUsage().external,
      );
      const cpuLoad = os.loadavg()[0].toFixed(2); // 1-minute
average load

      // Log to console
      console.log(
        `[HTTP Request Metrics] Method: ${method}, URL:
${originalUrl}, Status: ${statusCode}, Duration:
${duration.toFixed(
          2,
        )} ms, Response Size: ${responseSize} MB`,
      );
      console.log(
        `[Memory Usage] RSS: ${rssMemory} MB, Heap Total:
${heapTotal} MB, Heap Used: ${heapUsed} MB, External:
${externalMemory} MB`,
      );
      console.log(`[CPU Load] Percentage: ${cpuLoad}%`);
```

```
    // Save metrics to Excel
    this.saveMetricsToExcel({
      method,
      url: originalUrl,
      status: statusCode,
      duration: duration.toFixed(2),
      responseSize,
      rssMemory,
      heapTotal,
      heapUsed,
      externalMemory,
      cpuLoad,
    });
  });

  next();
}

private convertBytesToMB(bytes: number): number {
  return parseFloat((bytes / 1024 / 1024).toFixed(2));
}

private async saveMetricsToExcel(metrics: {
  method: string;
  url: string;
  status: number;
  duration: string;
  responseSize: number;
  rssMemory: number;
  heapTotal: number;
  heapUsed: number;
  externalMemory: number;
  cpuLoad: string;
}): Promise<void> {
  // Add a new row with metrics
```

```
    this.worksheet.addRow({
      method: metrics.method,
      url: metrics.url,
      status: metrics.status,
      duration: metrics.duration,
      responseSize: metrics.responseSize,
      rssMemory: metrics.rssMemory,
      heapTotal: metrics.heapTotal,
      heapUsed: metrics.heapUsed,
      externalMemory: metrics.externalMemory,
      cpuLoad: metrics.cpuLoad,
    });

    // Save the workbook
    await this.workbook.xlsx.writeFile(this.fileName);
  }

  private async loadExistingMetrics(): Promise<void> {
    if (fs.existsSync(this.fileName)) {
      await this.workbook.xlsx.readFile(this.fileName);
      this.worksheet = this.workbook.getWorksheet('Metrics');
    }
  }
}
```

main.ts (RestAPI)

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(8000);
}
bootstrap();
```

app.module.ts (RestAPI)

```
import { MiddlewareConsumer, Module, NestModule } from
'@nestjs/common';
import { UserModule } from './user/user.module';
import { PostModule } from './post/post.module';
import { CommentModule } from './comment/comment.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user/entities/user.entity';
import { Post } from './post/entities/post.entity';
import { Comment } from './comment/entities/comment.entity';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      port: 2432,
      username: 'ksharim',
      password: 'ksharim',
      database: 'graphql',
      entities: [User, Post, Comment],
      synchronize: true,
    }),
    UserModule,
    PostModule,
    CommentModule,
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

user.module.ts (Rest API)

```
import { MiddlewareConsumer, Module, NestModule } from
'@nestjs/common';
import { UserModule } from './user/user.module';
import { PostModule } from './post/post.module';
import { CommentModule } from './comment/comment.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user/entities/user.entity';
import { Post } from './post/entities/post.entity';
import { Comment } from './comment/entities/comment.entity';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      port: 2432,
      username: 'ksharim',
```

```
      password: 'ksharim',
      database: 'graphql',
      entities: [User, Post, Comment],
      synchronize: true,
    }),
    UserModule,
    PostModule,
    CommentModule,
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

user.controller.ts (RestAPI)

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
} from '@nestjs/common';
import { UserService } from './user.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';

@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.userService.create(createUserDto);
  }

  @Get()
  findAll() {
    return this.userService.findAll();
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.userService.findOne(id);
  }

  @Patch(':id')
  update(@Param('id') id: string, @Body() updateUserDto:
UpdateUserDto) {
    return this.userService.update(id, updateUserDto);
  }
```

```
  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.userService.remove(id);
  }
}
```

user.service.ts (RestApi)

```
import {
  BadRequestException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Not, Repository } from 'typeorm';
import { User } from './entities/user.entity';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
  ) {}

  async create(createUserDto: CreateUserDto) {
    // const user = await this.userRepository.findOne({
    //   where: [
    //     { email: createUserDto.email },
    //     { nickName: createUserDto.nickName },
    //   ],
    // });

    // if (user) {
    //   throw new BadRequestException('User already exists');
    // }

    return this.userRepository.save(createUserDto);
  }

  async findAll(): Promise<User[]> {
    const users = await this.userRepository.find({
      relations: [],
    });
    return users;
  }

  async findOne(id: string) {
    const user = await this.userRepository.findOne({
      where: { id },
      relations: ['posts', 'posts.comments'],
```

```
    });

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }
    return user;
  }

  async update(id: string, updateUserDto: UpdateUserDto) {
    const user = await this.findOne(id);
    const { firstName, lastName, email, nickName } =
updateUserDto;

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }

    if (user.email === email && user.nickName === nickName) {
      throw new BadRequestException('This email or nickName
already exists');
    }

    await this.userRepository.update(
      {
        firstName,
        lastName,
        email,
        nickName,
      },
      { id },
    );

    return `User with id ${id} has been updated`;
  }

  async remove(id: string) {
    const user = await this.userRepository.findOne({
      where: { id },
    });

    if (!user) {
      throw new NotFoundException(`User with id ${id} not found`);
    }

    await this.userRepository.delete(user);

    return `User with id ${id} has been deleted`;
  }
}
```

ДОДАТОК В

Теза доповіді для конференції

ЗАТВЕРДЖЕНО
44165850.01441-01-ЛЗ

APIAnalyzer

Теза доповіді для конференції «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості і освіті»

44165850.01441-01-ЛЗ

Листів 3

2025

В.1 ТЕЗА ДОПОВІДІ НА КОНФЕРЕНЦІЇ

Міністерство освіти і науки України

Український державний університет науки і технологій

ТЕЗИ
XVIII Міжнародної науково-практичної конференції
«СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ І ОСВІТІ»
Присвячено пам'яті Владислава СКАЛОЗУБА

ABSTRACTS
of the XVIII International Conference
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»
Dedicated to the memory of Vladislav SKALOZUB

12.12.2024 – 13.12.2024

Дніпро
2024

Дослідження ефективнсті методів web-API архітектур GraphQL та RESTAPI

Мардеросов Є.В., Український державний університет науки і технологій, Україна

Дослідження присвячене вивченню ефективності методів веб-API архітектур GraphQL та REST API, які широко використовуються для забезпечення взаємодії між клієнтами та серверами в сучасних веб-додатках. Зі зростанням кількості користувачів і складності систем виникає необхідність у виборі оптимальної архітектури API, яка дозволяє досягти балансу між швидкістю виконання запитів, гнучкістю передачі даних та спрощенням розробки.

Метою роботи є аналіз продуктивності архітектур GraphQL та REST API, зокрема швидкості виконання запитів, розміру респонсу, складності обробки даних та ефективності впровадження різних технік оптимізації. Для цього було розроблено два веб-API, обидва працюють з єдиною базою даних, що містить сутності `User`, `Post` та `Comment`. Один API реалізовано на основі REST архітектури, а інший — GraphQL.

Збір даних про продуктивність здійснювався за допомогою впровадження механізмів моніторингу, які дозволяють відслідковувати такі метрики, як час виконання запитів, використання ресурсів сервера, обсяг переданих даних та складність запитів.

У результаті експериментів було встановлено, що GraphQL забезпечує більшу гнучкість у формуванні запитів і дозволяє клієнту отримувати лише необхідні дані, зменшуючи обсяг респонсу. У той же час REST API демонструє вищу швидкість виконання простих CRUD операцій за рахунок меншої обчислювальної складності запитів. Впровадження батчингу та оптимізацій у GraphQL значно підвищує його ефективність, зокрема при виконанні складних запитів із вкладеними структурами.

Проведене дослідження показало, що вибір архітектури залежить від специфіки застосування: REST API підходить для додатків із фіксованою структурою даних і менш складними запитами, тоді як GraphQL є оптимальним вибором для систем із високими вимогами до гнучкості та кастомізації запитів. Отримані результати можуть бути використані для покращення процесу прийняття рішень щодо вибору архітектури API у нових проєктах, а також для оптимізації вже існуючих систем.