Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Машевського Андрія Миколайовича*
(ПІБ)

академічної групи *121М-19-1*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

на тему: *Дослідження ефективності використання інтерфейсу прикладного програмування GraphQL у порівнянні з REST*

______________________ *А.М. Машевський*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	Проф. Мороз Б.І.			
економічний	Доц. Касьяненко Л.В.			

Рецензент	Проф. Корнієнко В.І.			

Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_______________ І.М. Удовик
(підпис) (прізвище, ініціали)

« » __________ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-19-1* *Машевському Андрію Миколайовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Дослідження ефективності використання інтерфейсу прикладного програмування GraphQL у порівнянні з REST*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 р. № 888-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес передачі та/або отримання даних з використанням інтерфейсів прикладного програмування.

Предмет досліджень – моделі та методи розробки інтерфейсів прикладного програмування та взаємодії з ними.

Мета роботи – підвищення ефективності взаємодії між клієнтським застосунком та інтерфейсом прикладного програмування.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна результатів кваліфікаційної роботи полягає в удосконаленні методу розробки інтерфейсів прикладного програмування та методу взаємодії з ними, що дозволяє підвищити ефективність використання інтерфейсу прикладного програмування.

Практична цінність полягає у тому, що результати роботи, отримані в ході дослідження, можуть застосовуватись при проєктуванні, розробці та оптимізації роботи високонавантажених API, веб-сайтів та/або клієнт-серверного програмного забезпечення.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець)**
Аналіз теми та постановка задачі	01.09.2020-30.09.2020
Проєктування клієнтського та серверного середовища для тестування	01.10.2020-31.10.2020
Дослідження відмінностей та подібностей GraphQL та REST, проведення тестів	01.11.2020-06.12.2020

Завдання видав ______________ *Мороз Б. І.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання ______________ *Машевський А.М.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *12.03.2020 р.*

Термін подання кваліфікаційної роботи до ЕК *10.12.2020*

РЕФЕРАТ

Пояснювальна записка: 103 стор., 36 рис., 4 таблиці, 4 додатка, 67 джерел.

Об'єкт дослідження: процес передачі та/або отримання даних з використанням інтерфейсів прикладного програмування.

Предмет дослідження: моделі та методи розробки інтерфейсів прикладного програмування та взаємодії з ними.

Мета магістерської роботи: підвищення ефективності взаємодії між клієнтським застосунком та інтерфейсом прикладного програмування.

Методи дослідження. Для оцінки показників ефективності використання інтерфейсів прикладного програмування застосовані теоретичні методи дослідження та кількісні методи.

Наукова новизна результатів кваліфікаційної роботи полягає в удосконаленні методу розробки інтерфейсів прикладного програмування та методу взаємодії з ними, що дозволяє підвищити ефективність використання інтерфейсу прикладного програмування.

Практичне значення полягає у тому, що результати роботи, отримані в ході дослідження, можуть застосовуватись при проєктуванні, розробці та оптимізації роботи високонавантажених API, веб-сайтів та/або клієнт-серверного програмного забезпечення.

У розділі «Економіка» проведені розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПЗ і тривалості його розробки, а також проведені маркетингові дослідження ринку збуту створеного програмного продукту.

Список ключових слів: GraphQL, API, REST, Apollo, сервер, GraphQL клієнт, оптимізація API, ефективність API, React, JavaScript, NodeJS.

ABSTRACT

Explanatory note: 103 pages, 36 figures, 4 tables, 4 applications, 67 sources.

Object of research: the process of sending and / or receiving data using application programming interfaces.

Subject of research: models and methods of application programming interfaces development and interaction with them.

Purpose of Master's thesis: reducing the size of the transmitted data and the time of their transfer between the client application and the application programming interface.

Research methods. Theoretical research methods and quantitative methods are used to evaluate the efficiency of application programming interfaces.

Originality of research is to improve the method of developing application programming interfaces and the method of interaction with them, which allows to increase efficiency of application programming interfaces usage.

Practical value of the results consists of that the results obtained during the research can be used in the design, development and optimization of high-load APIs, websites and / or client-server software.

In the Economics section we calculated complexity of software development, the cost of creating software and the duration of its development, conducted marketing research for the created software product.

Keywords: GraphQL, API, REST, Apollo, server, GraphQL client, API optimization, API efficiency, React, JavaScript, NodeJS.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – прикладний програмний інтерфейс (інтерфейс програмування застосунків, інтерфейс прикладного програмування);

HTTP (Hyper Text Transfer Protocol) – протокол передачі даних, що використовується в комп'ютерних мережах;

JSON – текстовий формат обміну даними;

ПЗ – програмне забезпечення;

Фреймворк – програмне рішення, що полегшує розробку складних систем;

URI – рядок літер, який однозначно ідентифікує окремий абстрактний чи фізичний ресурс;

XML – eXtensible Markup Language;

REST – це стиль архітектури програмного забезпечення для розподілених систем, таких як World Wide Web;

Клієнт – комп'ютер (або програма), що використовує ресурси, надані іншим комп'ютером (або програмою);

Сервер – комп'ютер (або програма), що надає ресурси для інших комп'ютерів (або програм);

SEO – пошукова оптимізація сайту;

WSDL – мова визначення інтерфейсу вебсервісу;

MERN – MongoDB, Express, React, Node.

ЗМІСТ

ВСТУП………………………………………………………………………... 9
РОЗДІЛ 1. ОГЛЯД ІНТЕРФЕЙСІВ ПРИКЛАДНОГО ПРОГРАМУВАННЯ ТА ВИДИ ЇХ АРХІТЕКТУР………………………………………………. 12
1.1 Інтерфейси прикладного програмування…………………………….. 12
1.2 Архітектури інтерфейсів прикладного програмування………..…. 14
1.2.1 Архітектурний стиль Remote Procedure Call (RPC)……………... 16
1.2.2 Архітектурний стиль Service Object Access Protocol (SOAP).….. 20
1.2.3 Архітектурний стиль Representational State Transfer (REST)….. 22
1.2.4 Архітектурний стиль GraphQL……………...……………...…... 26
1.3 Показники ефективності API……………………………….…….. 31
1.4 Висновки до першого розділу та постановка завдання.................. 32
РОЗДІЛ 2. ПРОЄКТУВАННЯ СЕРЕДОВИЩА ДЛЯ ТЕСТУВАННЯ ЕФЕКТИВНОСТІ API.. 33
2.1 Інструменти для створення клієнтського та серверного середовища……………………………………………………………... 33
2.1.1 Мова програмування JavaScript…………………………….. 33
2.1.2 Асинхронність в JavaScript……………………………….... 34
2.1.3 JavaScript у браузері…………………………………………… 36
2.1.4 Платформа Node.js…………………………………….……... 40
2.1.5 Фреймворк Express……………………………………….. 42
2.1.6 Фреймворк ReactJS………………………….……………….. 43
2.2. Клієнти GraphQL для клієнтського середовища.………….............. 46
2.2.1 Клієнт GraphQL Request...……………………………………... 47
2.2.2 Клієнт Apollo Client……………..................…………….. 47
2.2.3 Клієнт Relay Modern…………………………...……………… 49
2.3. Клієнти GraphQL для серверного середовища….………………….. 50
2.3.1 Клієнт GraphQL.js…...……………………………...……… 50
2.3.2 Клієнт Apollo-server………………………...……………….. 51

2.3.3 Клієнт Express-graphql…………………………...…………………….. 52
2.4 Проєктування клієнтського середовища..………………………….. 52
2.5 Проєктування серверного середовища....………………………….. 58
2.6 Висновки до другого розділу... 59
РОЗДІЛ 3. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ GraphQL У ПОРІВНЯННІ З REST……………………………………….. 60
3.1 Подібності і відмінності GraphQL та REST..................................... 60
3.2 Тестування ефективності за обраними показниками...................... 61
3.2.1 Тест №1…………………………...…………...………………….. 62
3.2.2 Тест №2……………...……………………………...…………….. 65
3.3 Висновки до третього розділу.. 69
РОЗДІЛ 4. ЕКОНОМІЧНИЙ РОЗДІЛ.. 70
4.1 Визначення трудомісткості проведення дослідження та розробки необхідного для його проведення програмного забезпечення…….. 70
4.2 Витрати на створення програмного забезпечення для проведення дослідження…………………………………………..……………… 73
4.3 Маркетингові дослідження ринку використання результатів дослідження.. 75
4.4 Оцінка економічної ефективності впровадження програмного забезпечення……………………………………………………….. 77
ВИСНОВКИ…………... 78
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ………………………………….…..... 79
Додаток А. ЗНІМКИ ЕКРАНУ З РЕЗУЛЬТАТАМИ ТЕСТУВАННЯ....…… 85
Додаток Б. ЛІСТИНГ ПРОГРАМИ………...………………………………..…. 92
Додаток В. ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ……...……. 102
Додаток Г. ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ…………… 103

ВСТУП

Актуальність дослідження. За останні роки швидкість роботи веб-сайту стала одним з найважливіших показників успіху будь-якого веб-сайту в Інтернеті. Це вже не розкіш, а необхідність.

Якщо потрібно додати більше елементів на веб-сайт – треба переконатись, що швидкість веб-сайту не впаде, інакше можна втратити більше, ніж отримати. Незалежно від того, яка мета – отримати більше потенційних клієнтів, продажів або зменшити телефонні дзвінки в службу обслуговування клієнтів, оптимізацію веб-сайту можна використовувати, щоб зробити його більш ефективним у досягненні цих цілей.

Ефективність веб-сайту – це показник того, як швидко веб-сайт завантажується. Навіть якщо у нього чудовий вміст та досконала оптимізація пошукової системи (SEO), повільний веб-сайт відверне відвідувачів і змусить Google двічі задуматися про включення сайту в його результати.

Веб-сайт повинен бути швидким та відповідати очікуванням споживачів. Дослідження показують, що ефективність веб-сайту це дуже важливий показник. Неефективні веб-сайти розчаровують клієнтів, через що вони можуть ніколи не повернутися на сайт і це може нашкодити бізнесу.

У сучасному світі швидкість роботи є одним з головних показників якості, зручності та продуктивності використання не тільки для веб-сторінки, а й також для комп'ютерного або мобільного додатка.

Незалежно від того, чи йдеться мова про стилі (CSS), скрипти (JavaScript) або файли зображень, кожен ресурс, який використовується на веб-сайті, повинен бути завантажений.

Найбільш очевидний показник продуктивності пов’язаний з розміром файлу або інших даних, що передаються. Усі байти, що одержуються, повинні бути завантажені на комп’ютер відвідувача. Чим більше байт у передаваємих даних, тим довше користувачеві доведеться чекати їх завантаження.

Однак не має менш помітне значення має те, що кожен об’єкт, який включений на сторінку, повинен завантажувати браузер. Встановлення та управління цими з'єднаннями за допомогою HTTP запитів вимагає часу на отримання сервером інформації від клієнта, її обробку та повернення відповіді клієнту.

Ще гіршим є той факт, що браузери мають обмеження щодо кількості запитів, які вони дозволяють одночасно. Якщо, наприклад, сторінка містить 30 ресурсів, браузер відвідувача завантажить їх групами (зазвичай від 2 до 8). Це означає, що навіть якщо розміри окремих файлів невеликі, все одно знадобиться значний час, поки всі вони не будуть завантажені.

Мета дослідження полягає у підвищенні ефективності взаємодії між клієнтським застосунком та інтерфейсом прикладного програмування. Відповідно до мети у кваліфікаційній роботі необхідно вирішити **наступні завдання:**

1. Дослідити недоліки та переваги обраних інтерфейсів прикладного програмування.

2. Протестувати ефективність використання інтерфейсів прикладного програмування за обраними показниками.

3. За результатами дослідження зробити висновки щодо ефективності використання GraphQL API у порівнянні з REST API.

Об'єкт дослідження: процес передачі та/або отримання даних з використанням інтерфейсів прикладного програмування.

Предмет дослідження: моделі та методи розробки інтерфейсів прикладного програмування та взаємодії з ними.

Методи дослідження. Для оцінки показників ефективності використання інтерфейсів прикладного програмування застосовані теоретичні методи дослідження та кількісні методи.

Наукова новизна результатів кваліфікаційної роботи полягає в удосконаленні методу розробки інтерфейсів прикладного програмування та

методу взаємодії з ними, що дозволяє підвищити ефективність використання інтерфейсу прикладного програмування.

Практичне значення полягає у тому, що результати роботи, отримані в ході дослідження, можуть застосовуватись при проєктуванні, розробці та оптимізації роботи високонавантажених API, веб-сайтів та/або клієнт-серверного програмного забезпечення.

Особистий внесок автора:

1. Наукові результати роботи отримані автором самостійно.
2. Вибір методів досліджень і технологій реалізації.
3. Розробка середовища для тестування інтерфейсів прикладного програмування GraphQL та REST.
4. Розробка теоретичної частини роботи, в якій досліджені і систематизовані знання про інтерфейси прикладного програмування.
5. Оцінка отриманих результатів.

Структура і обсяг роботи. Робота складається з вступу, трьох розділів і висновків. Містить 103 сторінки, в тому числі 67 сторінок тексту основної частини з 36 рисунками, списку використаних джерел з 67 найменуваннями на 6 сторінках, 4 додатка на 19 сторінках.

РОЗДІЛ 1
ОГЛЯД ІНТЕРФЕЙСІВ ПРИКЛАДНОГО ПРОГРАМУВАННЯ ТА ВИДИ ЇХ АРХІТЕКТУР

1.1. Інтерфейси прикладного програмування

Для взаємодії один з одним два окремих додатки потребують посередника. Для цього розробники будують так звані мости – інтерфейси прикладного програмування (Application Programming Interfaces) щоб дозволити одному з додатків отримати доступ до інформації або функцій іншого, спілкуватися між собою.

Для швидкої і масштабної інтеграції додатків API реалізовані з використанням протоколів та/або специфікацій для визначення семантики і синтаксису повідомлень, переданих по мережі. Ці специфікації складають архітектуру API.

API пропонує простий спосіб підключення, інтеграції та розширення програмних систем [1]. Бізнес, ринки і банки керуються програмним забезпеченням. Процеси промислового виробництва контролюються програмним забезпеченням. Машини, автомобілі та багато споживчих товарів містять програмне забезпечення. Однак ці програмні системи зазвичай ізольовані і до функцій однієї системи не можна отримати доступ з іншої системи. API-інтерфейси надають можливість підключати ці окремі програмні об'єкти. API-інтерфейси надають можливості, необхідні для підключення, розширення і інтеграції програмного забезпечення. З'єднуючи програмне забезпечення, вони з'єднують підприємства з іншими підприємствами, підприємства з їх продуктами, послуги з продуктами.

Загалом кажучи, API – це межа між однією частиною програмної системи та іншою. Воно містить набір операцій, якими одна частина системи забезпечує використання інших частин системи (або інших систем). Наприклад, архів фотографій може надавати API для переліку альбомів фотографій, перегляду

окремих фотографій, додавання коментарів тощо. Інтернет-галерея зображень може використовувати цей API для відображення цікавих фотографій, хоча програма обробки текстових процесорів може використовувати той самий API, щоб дозволити вбудовувати зображення в документ. Як показано на рис. 1.1, API обробляє запити від одного або декількох клієнтів від імені користувачів. Клієнтом може бути веб-сайт або мобільний додаток з інтерфейсом користувача (UI), або це може бути інший API без явного інтерфейсу. Сам API може розмовляти з іншими API, щоб завершити свою роботу.

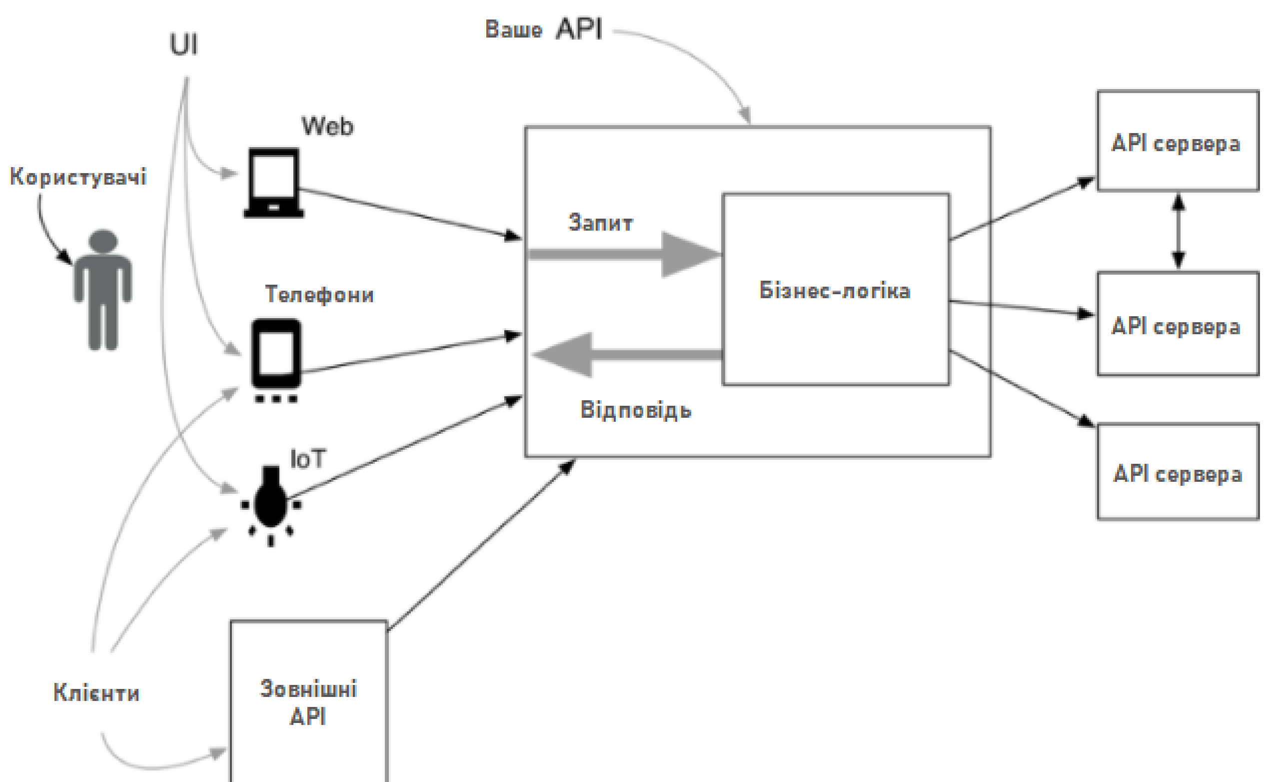

Рис.1.1. Приклад взаємодії з API

Інтерфейси прикладного програмування складаються з двох компонентів:

1. Інтерфейс програмного забезпечення, створений за допомогою мов програмування, спроектований за специфікацією, яка його представляє.

2. Технічна специфікація, яка описує варіанти обміну даними між рішеннями. Специфікація виконана у формі запиту на обробку та протоколи доставки даних.

API, як правило, можуть спростити та пришвидшити розробку програмного забезпечення. Розробники можуть додавати функціональність (тобто механізм рекомендацій, бронювання житла, розпізнавання зображень, обробку платежів) від інших постачальників до існуючих рішень або створювати нові додатки, використовуючи послуги сторонніх постачальників. У всіх цих випадках спеціалістам не доводиться мати справу з програмним кодом кодом, намагаючись зрозуміти, як працює це рішення. Вони просто підключають своє програмне забезпечення до іншого. Іншими словами, API слугують шаром абстракції між двома системами, приховуючи складність та робочі деталі останньої.

1.2. Архітектури інтерфейсів прикладного програмування

Метою специфікацій API є стандартизація обміну даними між веб-службами. У цьому випадку стандартизація означає здатність різноманітних систем, написаних різними мовами програмування та/або працюючих на різних операційних системах, або з використанням різних технологій, безперешкодно спілкуватися між собою.

Більшість книг про розробку API зосереджені на використанні методів HTTP, дизайні URL-адрес, кодах стану HTTP, заголовках HTTP та структурі ресурсів у тілі HTTP[1]. Однак насправді це найменша проблема при побудові API. Справжньою проблемою є пошук архітектури API та визначення методології.

Архітектура API – це більше, ніж правильне застосування принципів REST, SOAP або GraphQL.

Архітектура API охоплює загальну картину API, і її можна побачити з декількох ракурсів [1]:

– архітектура API може використовувати архітектуру повного рішення, що складається не тільки з самого API, але і з клієнта API, такого як мобільний

додаток та кілька інших компонентів. Архітектура рішення API пояснює компоненти та їх взаємозв'язки всередині програмного рішення;

– архітектура API може посилатися на технічну архітектуру API платформи. Створюючи, запускаючи та виставляючи не лише один, а декілька API, стає зрозумілим, що певні будівельні блоки API, функціональні можливості та функції управління API повинні використовуватися знову і знову. Платформа API забезпечує інфраструктуру для розробки, запуску та управління API;

– архітектура API може посилатися на архітектуру API портфоліо. API портфоліо містить усі API підприємства, і ним потрібно керувати, як його продуктом. Архітектура портфоліо API аналізує функціональність API та організовує, керує та повторно використовує API.

Перемістити стовп мосту, виготовленого зі сталі та бетону, дуже важко. Такі зміни важкі, дуже багато коштують та трудомісткі. Ось чому перед створенням мосту створюється план. Це дозволяє продумати всі деталі, обрати серед декількох пропозицій та провести аналіз "що-якщо". Зміни до плану легко і дешево виконати. І вносячи зміни до плану, сподіваємось, не потрібно вносити зміни до справжніх проектів. Те саме стосується API.

Коли API вже створені та активно використовуються, зміни є складними, дорогими та трудомісткими. Ще гірше, що зміни в опублікованих API можуть зламати програмне забезпечення будь-яких клієнтів, які використовують API. Як наслідок, споживачі можуть засмутитися і змінити постачальника API. Щоб цього уникнути, API повинен бути якісно спроектований з самого початку, до його першого опублікування.Цього можна досягти, заздалегідь плануючи використання архітектури API.

Спланована архітектура API підвищує ефективність побудови правильного API, зменшує витрати та час як на будівництво, так і на технічне обслуговування, а отже, зменшує технічний ризик, пов’язаний внесенням змін у працюючий продукт.

Використання архітектур API – це підхід до зменшення ризику, але він працює тільки тоді, коли API добре спроектовано до початку створення. Це

дозволяє уникнути ситуацій, коли ресурси витрачаються на впровадження API, які не можуть справно виконувати свою задачу.

Правильна архітектура API повинна проєктуватись на основі договору, у якому прописані всі вимоги. Після створення та релізу API його можна використовувати не лише провайдерам API для реалізації проксі-сервера API, але й споживачам API для створення нових додатків на основі цього API. Якщо робота ведеться за контрактом розробнику, який буде використовувати API, не потрібно чекати завершення розробки API, тому що розробка API та програми може йти паралельно.

Нефункціональні властивості API не повинні бути задумом. API повинен бути спроектован з самого початку, щоб задовольнити всі нефункціональні властивості, такі як безпека, продуктивність, доступність. Виходячи з архітектури, вплив архітектурного вибору на нефункціональні властивості можна визначити на початку проєктування.

Правильна архітектура та дизайн API – це інвестиція. У довгостроковій перспективі це заощадить час і навіть допоможе уникнути помилок.

1.2.1. Архітектурний стиль Remote Procedure Call (RPC)

У 1960-х роках було винайдено віддалений виклик процедур (RPC) [2]. Клієнтський додаток ініціював RPC, який надіслав повідомлення із запитом на віддалений комп’ютер, щоб щось зробити, віддалений комп'ютер надіслав клієнту відповідь. Ці комп’ютери відрізнялися від клієнтів та серверів, якими ми користуємося сьогодні, але потік інформації в основному був однаковим: записуються деякі дані у клієнта, отримується відповідь від сервера.

Веб-API можуть дотримуватися принципів обміну ресурсами на основі віддаленого виклику процедур. Цей протокол визначає взаємодію між клієнт-серверними програмами. Одна програма (клієнт) запитує дані або функції у іншої програми (сервера), розташованої на іншому комп’ютері в мережі, і сервер надсилає необхідну відповідь.

Віддалений виклик процедур (RPC) – одна з найпростіших парадигм API, в якій клієнт виконує блок коду на іншому сервері. Тоді як REST оперує ресурсами, RPC діями. Клієнти зазвичай передають ім'я методу та аргументи серверу і отримують назад JSON або XML.

RPC API зазвичай дотримуються двох простих правил [3]:

1. Кінцеві точки містять назву операції, яку потрібно виконати.
2. Виклики API здійснюються за допомогою найбільш дієвого дієслова HTTP: GET для запитів лише на читання та POST для інших.

Стиль RPC чудово підходить для API, які виконують складні дії, які можуть мати більше нюансів та ускладнень, ніж може бути інкапсульовано CRUD або для яких є побічні ефекти, не пов’язані з наявним “ресурсом”. API у стилі RPC також вміщують складні моделі ресурсів або дії над різними типами ресурсів.

RPC працює за таким алгоритмом (рис. 1.2):

1. Клієнт викликає віддалену процедуру, серіалізує параметри та додаткову інформацію у повідомленні та надсилає повідомлення (запит) на сервер.
2. Отримавши повідомлення, сервер десеріалізує його вміст, виконує запитувану операцію та надсилає результат назад клієнту.
3. Заглушка сервера та клієнта дбають про серіалізацію та десеріалізацію параметрів.

Переваги архітектури RPC:

1. Пряма і проста взаємодія. RPC використовує GET запити для отримання інформації та POST запити для всього іншого. Механіка взаємодії між сервером і клієнтом зводиться до звернення до кінцевої точки та отримання відповіді.
2. Прості в додаванні функції. Якщо буде отримана нова вимога до API, можна легко додати ще одну кінцеву точку, що виконує цю вимогу дуже просто. Потрібно створити нову функцію та призначити їй кінцеву точку і все, тепер клієнт може звернутися до цієї кінцевої точки та отримати інформацію, що відповідає встановленій вимозі.

3. Висока продуктивність. Невеликі запити легко виконуються у мережі, забезпечуючи високу продуктивність, що важливо для спільних серверів та паралельних обчислень, що виконуються в мережах робочих станцій. RPC здатний оптимізувати мережевий рівень і зробити його дуже ефективним, надсилаючи тонни повідомлень на день між різними службами [3].

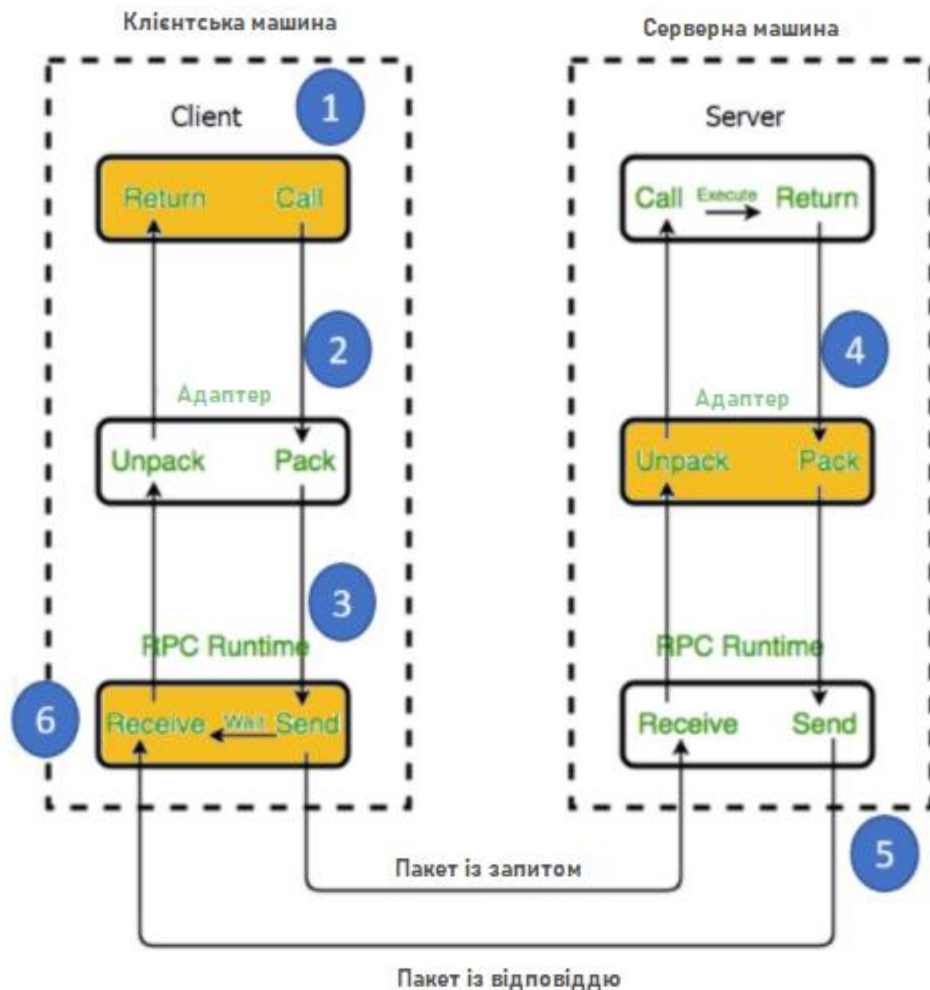

Рис. 1.2. Алгоритм роботи RPC

Недоліки архітектури RPC:

1. Щільне зчеплення з базовою системою [3]. Рівень абстракції API сприяє його повторному використанню. Тісне зв’язування RPC із базовою системою не дозволяє абстрагувати рівень між функціями в системі та зовнішнім API. Це піднімає проблеми безпеки, оскільки досить легко просочити деталі реалізації про базову систему в API. Тісне зчеплення RPC ускладнює вимоги до

масштабованості та вільно зв'язаних команд. Отже, клієнт або турбується про можливі побічні ефекти виклику певної кінцевої точки, або намагається з’ясувати, яку кінцеву точку викликати, оскільки він не розуміє, як сервер називає свої функції.

2. Важко зрозуміти цепочку викликів. У RPC немає можливості проаналізувати API або зрозуміти, яку функцію викликати на основі попередніх запитів.

3. Постійне створення нових функцій. Створювати нові функції дуже просто просто, тому замість редагування існуючих, створюються нові, через що накопичується величезна кількість функцій.

Шаблон RPC почав використовуватися приблизно в 80-х роках, але це не робить його автоматично застарілим. Такі великі компанії, як Google, Facebook (Apache Thrift) та Twitch (Twirp), використовують внутрішні високопродуктивні змінні RPC для виконання надзвичайно високопродуктивних задач. Їх масивні системи мікропослуг вимагають чіткого внутрішнього спілкування, розміщеного в коротких повідомленнях.

RPC – це правильний вибір для надсилання команд у віддалену систему. Наприклад, API Slack дуже орієнтований на команди: приєднатися до каналу, залишити канал та надіслати повідомлення. Саме тому, архітектурні дизайнери Slack API змоделювали його у RPC-подібному стилі, зробивши його маленьким, щільним та простим у використанні.

Маючи пряму інтеграцію між одним провайдером та споживачем, ми не хочемо витрачати багато часу, передаючи безліч метаданих по дроту, як це робить REST API. Завдяки високій швидкості повідомлень та продуктивності повідомлень, gRPC та Twirp є вагомими аргументами для мікросервісів. Використовуючи HTTP “під капотом”, gRPC може оптимізувати мережевий рівень і зробити його дуже ефективним, надсилаючи тисячі повідомлень на день між різними службами. Однак, якщо ціль стоїть не на високу продуктивність мережі, а на стабільний контакт API між командами, що створюють зовсім різні мікросервіси, REST може забезпечити це.

API у стилі RPC не є ексклюзивними для HTTP. Існують і інші високопродуктивні протоколи, доступні для API в стилі RPC, включаючи Apache Thrift та gRPC. Хоча існують параметри JSON для gRPC, як Thrift, так і gRPC запити серіалізовані. Структуровані дані та чітко визначені інтерфейси дозволяють цю серіалізацію. Thrift та gRPC також мають вбудовані механізми редагування структур даних.

RPC також відомий як підпрограма або виклик функції. Одним із двох способів реалізації виклику віддаленої процедури є SOAP.

1.2.2. Архітектурний стиль Service Object Access Protocol (SOAP)

Наприкінці 90-х у Microsoft з'явився протокол простого доступу до об'єктів (SOAP) [4]. SOAP використовував XML для кодування повідомлення та HTTP як транспорт для цього повідомлення. SOAP також використовував систему типів і представив концепцію ресурсно-орієнтованих викликів даних. SOAP давав досить передбачувані результати, але викликав розчарування, оскільки реалізації SOAP були досить складними.

SOAP – це простий протокол для обміну структурованою інформацією в децентралізованому розподіленому середовищі, згідно з визначенням Microsoft, яка його розробила [5]. Взагалі кажучи, ця специфікація містить правила синтаксису повідомлень запитів та відповідей, надісланих веб-програмами. API, які відповідають принципам SOAP, дозволяють обмінюватися повідомленнями XML між системами через HTTP або Простий протокол передачі пошти (SMTP) для передачі пошти.

Розширювана мова розмітки (XML) – це простий і дуже гнучкий формат тексту, який широко використовується для зберігання та обміну даними через Інтернет або інші мережі. XML визначає набір правил для кодування документів у форматі, який можуть читати як люди, так і машини. Мова розмітки – це сукупність символів, які можна розмістити в тексті для окреслення та позначення

частин текстового документа. Текстові документи XML містять самоописні теги об'єктів даних, що робить їх легко читабельними.

SOAP зазвичай використовується з корпоративним веб-програмним забезпеченням для забезпечення високої безпеки переданих даних. SOAP API є кращим серед постачальників платіжних шлюзів, управління ідентифікацією та CRM-рішень, а також фінансових та телекомунікаційних послуг. Відкритий API PayPal є одним із найвідоміших API SOAP. Він також часто використовується для підтримки застарілої системи.

SOAP дотримується стилю RPC і виставляє процедури як основні поняття (наприклад, getCustomer). Він стандартизований W3C і є найбільш широко використовуваним протоколом для веб-служб. SOAP найчастіше використовується для внутрішньої інтеграції на підприємствах.

SOAP пропонує прив'язку до різних транспортних протоколів, таких як HTTP, SMTP, TCP, UDP або JMS. SOAP базується на XML і фактично перетворився на XML-RPC. Серіалізоване SOAP-повідомлення обгортається конвертом, що містить заголовок із метаінформацією та тілом із запитом, відповіддю чи помилкою. Складні структури даних для запиту та відповіді можуть бути визначені та описані за допомогою схеми XML. Інтерфейс служб SOAP описується спеціальною стандартизованою мовою, мовою опису веб-послуг WSDL.

SOAP пропонує безліч розширень для передачі двійкових даних, для безпеки, шифрування та підписання – це лише декілька із багатьох. Ці розширення також відомі як WS- *, і деякі розширення стандартизовані, а інші – специфічні для продукту.

Архітектури стилю SOAP широко використовуються, однак, як правило, лише для внутрішнього використання компанії або для послуг, що викликаються надійними партнерами.

1.2.3. Архітектурний стиль Representational State Transfer (REST)

На сьогоднішній день абревіатура REST стала модним словом, і, таким чином, багато техніків недбало кидають його на цифровий вітер, не розуміючи до кінця, що це насправді це означає.

Те, що можна взаємодіяти з системою за допомогою HTTP і надсилати та отримувати JSON, не означає, що це RESTful система. REST – це набагато більше.

Усе це почалося з Роя Філдінга, американського інформатика, 1965 року народження. Він є одним з головних авторів протоколу HTTP (протоколу, на якому базується вся Інфраструктура Інтернету). Він також є одним із співавторів веб-сервера Apache, і він був головою Фонду програмного забезпечення Apache протягом перших трьох років його існування [7].

Отже, як можна побачити, Філдінг зробив великий внесок у IT-світ, особливо щодо Інтернету, але напевно, що його докторська дисертація – це те, що отримало найбільше визнання та зробило його ім'я відомим для всього світу.

У 2000 році Філдінг презентував докторську дисертацію «Архітектурні стилі та дизайн мережевої архітектури програмного забезпечення». У ньому він ввів термін REST, архітектурний стиль для розподілених гіпермедійних систем.

Хоча REST виявився величезним стрибком уперед щодо взаємозв'язку розподілених систем, до того, як стаття Філдінга стала популярною, розробники все ще шукали рішення проблеми: як легко взаємопов'язати неоднорідний набір систем. Слід пам'ятати, що REST не залежить від протоколу (якщо протокол підтримує схему URI).

REST вважається більш простою альтернативою SOAP, яку багатьом розробникам важко використовувати, оскільки для виконання кожного завдання потрібно ввести багато коду та дотримуватися структури XML для кожного надісланого повідомлення. REST дотримується іншої логіки, оскільки робить дані доступними як ресурси. Кожен ресурс представлений унікальною URL-адресою, і можна запросити цей ресурс, вказавши його URL-адресу.

Веб-API, які відповідають архітектурним обмеженням REST, називаються RESTful API. Ці API використовують HTTP-запити (методи або дієслова) для роботи з ресурсами: GET, PUT, HEAD, POST, PATCH, CONNECT, TRACE, OPTIONS та DELETE (рис. 1.3) [8].

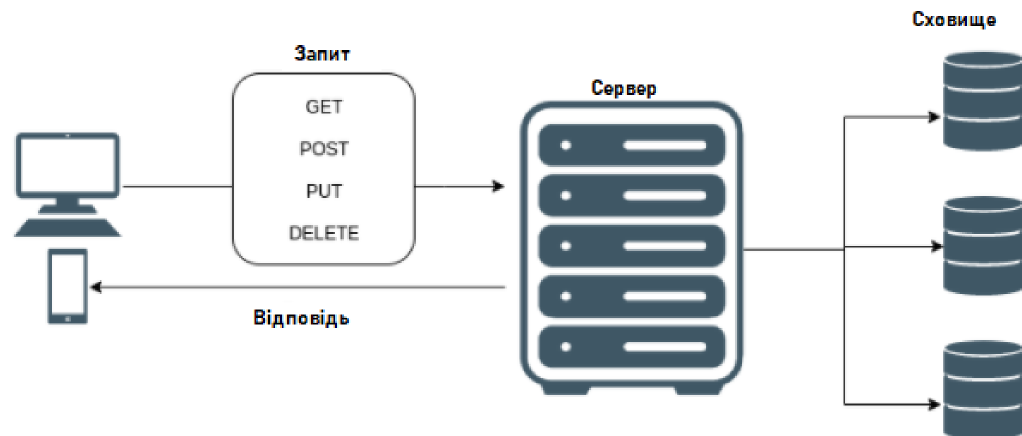

Рис. 1.3. Схема взаємодії з REST API

Системи RESTful підтримують обмін повідомленнями в різних форматах, таких як звичайний текст, HTML, YAML, XML та JSON, тоді як SOAP дозволяє лише XML [8].

Можливість підтримувати кілька форматів для зберігання та обміну даними є однією з причин, чому REST сьогодні є переважаючим вибором для побудови загальнодоступних API.

JavaScript Object Notation (JSON) – це легкий і простий для аналізу текстовий формат для обміну даними. Його синтаксис базується на підмножині Стандартного видання ECMA-262. Кожен файл JSON містить колекції пар імен / значень та впорядковані списки значень (рис. 1.4). Оскільки це універсальні структури даних, формат можна використовувати з будь-якою мовою програмування.

JSON був широко прийнятий завдяки популярності REST.

```
[
    {
        "stage":"CLIENT_DECLINED",
        "name":"Отказанные",
        "docTypeId":71,
        "quantity":1,
        "isProductMenu":false,
        "isShowCase":true
    },
    {
        "stage":"CLIENT_FOR_SEND",
        "name":"К отправке",
        "docTypeId":71,
        "quantity":2,
        "isProductMenu":false,
        "isShowCase":true
    },
    {
        "stage":"CLIENT_FOR_SIGNATURE",
        "name":"На подпись",
        "docTypeId":75,
        "quantity":1,
        "isProductMenu":false,
        "isShowCase":true
    }
]
```


Рис. 1.4. Приклад JSON

Гіганти соціальних медіа та туристичні компанії надають зовнішні API, щоб ще більше покращити видимість свого бренду. Twitter має безліч RESTful API; Expedia має як SOAP, так і RESTful API для своїх партнерів.

Простіше кажучи, REST (скорочення від REpresentational State Transfer) – це архітектурний стиль, визначений для створення та організації розподілених систем. Ключовим словом у цьому визначенні повинен бути стиль, оскільки важливим аспектом REST є те, що це архітектурний стиль – не керівний принцип, не стандарт або щось, що означатиме, що існує набір жорстких правил, яких слід дотримуватися, щоб у підсумку мати архітектуру RESTful.

RESTful система повинна відповідати наступним крітеріям:

1. Система повинна бути розділена на клієнтів і на серверів.
2. Сервер не повинен зберігати будь-якої інформації про клієнтів.

3. Кожна відповідь повинна бути зазначена чи кешується вона чи ні для запобігання повторного використання клієнтами застарілих або некоректних даних у відповідь на подальші запити.

4. Система повинна бути складатися з ієрархії шарів але з умовою, що кожен компонент може бачити компоненти тільки наступного шару.

І оскільки це стиль, і немає запиту на коментарі (RFC) для його визначення, він може спричинити помилкові тлумачення з боку людей, які читають про нього. Мало того, але деякі доходять до того, що залишають деталі осторонь та впроваджують підмножину її функцій, що, у свою чергу, призводить до широкого та неповного ідеалу REST, виключаючи функції, які в іншому випадку були б корисними та допомогли користувачам вашої системи.

Основна ідея REST полягає в тому, що розподілена система, організована RESTfully, покращиться в наступних сферах:

1. Продуктивність: запропонований REST стиль спілкування має бути ефективним і простим, що дозволяє підвищити продуктивність систем, які його приймають.

2. Масштабованість взаємодії компонентів: будь-яка розподілена система повинна мати можливість досить добре обробляти цей аспект, і проста взаємодія, яка запропонована REST, дуже просто та гнучко дозволяє це зробити.

3. Простота інтерфейсу: простий інтерфейс дозволяє легше взаємодіяти між системами, що, в свою чергу, може забезпечити переваги, подібні до тих, що згадувались раніше. Також створення нового REST API не вимагає великих витрат. Для порівняння, створення служб SOAP вимагає більших зусиль через специфікацію файлів WSDL із сумісною реалізацією.

4. Підтримка багатьох типів даних: API може надавати та отримувати ресурс у кількох альтернативних форматах, а клієнт може читати відповіді лише в одному з цих форматів. Механізми узгодження типу даних за допомогою заголовків HTTP запитів визначають, як клієнтський API може обмінюватися інформацією про свої можливості та узгоджувати відповідний тип вмісту. Цей механізм успадковується від HTTP.

З цього списку можна екстраполювати деякі прямі переваги:

1. Компонентно-орієнтована конструкція дозволяє виготовляти системи, які є дуже стійкими до несправностей. Відмова одного компонента не впливає на всю стабільність системи – це велика вигода для будь-якої системи.
2. З'єднати компоненти досить просто, мінімізуючи ризики при додаванні нових функцій або масштабуванні вгору чи вниз.
3. Система, розроблена з урахуванням вимог REST, буде доступною для широкої аудиторії завдяки її мобільності. За допомогою загального інтерфейсу система може використовуватися ширшим колом розробників. Для досягнення цих властивостей та переваг до REST було додано набір обмежень, які допомагають визначити єдиний інтерфейс з'єднувача.

1.2.4. Архітектурний стиль GraphQL

У 2012 році Facebook вирішив, що потрібно оновити свої мобільні додатки [10]. Програми компанії для систем iOS та Android були лише тонкими обгортками навколо перегляду мобільного веб-сайту. Facebook мав сервер RESTful та таблиці даних FQL (версія SQL від SQL). Ефективність роботи була дуже повільною, і програми часто падали. Тоді інженери зрозуміли, що їм потрібно вдосконалити спосіб надсилання даних до своїх клієнтських додатків. Як наслідок, Facebook вирішив створити новий стандарт запиту, який дозволить їм обернути всі необхідні дані в один запит, оскільки їм потрібні були численні запити, щоб отримати, наприклад, всю інформацію, пов'язану з публікацією.

Команда Лі Байрона, Ніка Шрока та Дана Шафера вирішила переглянути свої дані з боку клієнта. Вони створили GraphQL – мову запитів з відкритим вихідним кодом, яка описує можливості та вимоги моделей даних для клієнтських / серверних додатків компанії.

У липні 2015 року команда випустила свою початкову специфікацію GraphQL та довідкову реалізацію GraphQL в JavaScript під назвою graphql.js. На самому початку люди у Facebook дуже скептично ставилися до випуску GraphQL

до спільноти, оскільки це був лише набір PHP-кодів. Але вони зрозуміли, що GraphQL – це набагато більше, ніж просто інструмент, який допомагає їхній компанії отримувати складні речі для свого інтерфейсу.

У вересні 2016 року GraphQL вийшов зі стадії «технічного попереднього перегляду». Це означало, що GraphQL був офіційно готовий до виробництва, хоча він уже роками використовувався у виробництві у Facebook. Сьогодні GraphQL тепер керує майже всіма видами даних Facebook і використовується у розробці IBM, Intuit, Airbnb та інших.

Щоб принести популярність GraphQL, Facebook почав створювати версію GraphQL на основі NodeJS і запропонував спільноті створювати варіанти різними мовами, щоб зробити цю технологію доступною для кожної створеної інфраструктури.

На цьому етапі скептицизму Facebook зробив ще один значний крок до спільноти з відкритим програмним кодом, відмежувавши початковий проект від материнської компанії в 2018 році і зібравши кожну підгрупу в єдиний фонд, GraphQL Foundation.

GraphQL представляє декілька зовсім нових ідей, але все це слід розуміти в історичному контексті передачі даних. Коли ми думаємо про транспорт даних, ми намагаємося зрозуміти, як передавати дані вперед і назад між комп'ютерами. Ми запитуємо деякі дані з віддаленої системи і очікуємо відповіді (рис. 1.5).

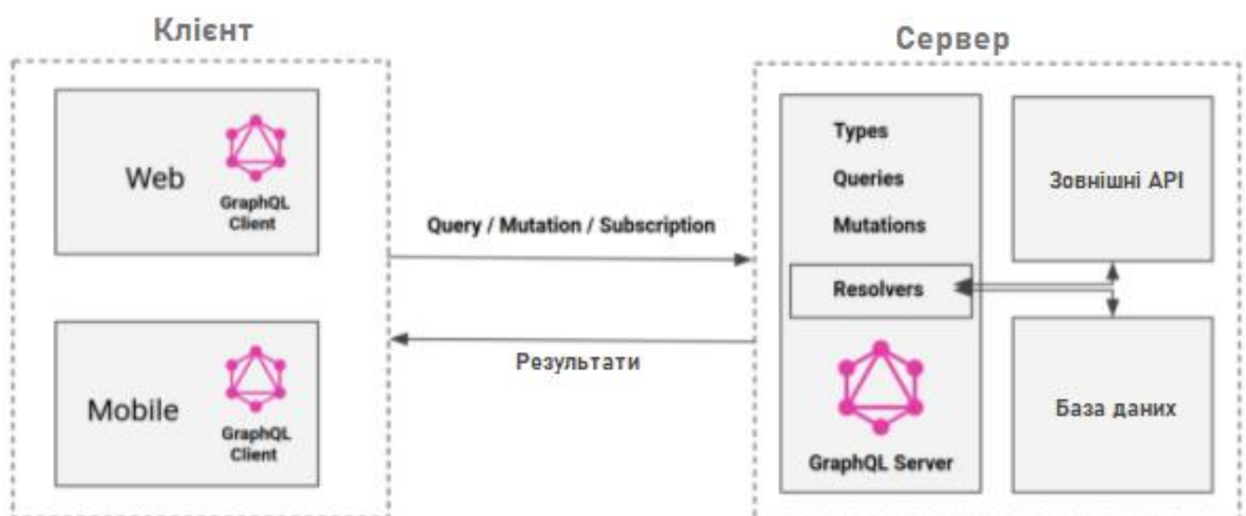

Рис. 1.5. Схема взаємодії з GraphQL API

GraphQL – це альтернатива REST API. На відміну від REST API, GraphQL замість повернення всіх даних із сервера, повертаються лише ті дані, які шукає клієнт.

GraphQL – це специфікація для зв'язку клієнт-сервер. Специфікація описує можливості та характеристики мови. Ми виграємо від специфікацій мови, оскільки вони надають загальний функціонал для використання та найкращі практики щодо використання цієї мови розробниками.

Досить помітним прикладом специфікації програмного забезпечення є специфікація ECMAScript. Час від часу група представників компаній-браузерів, технологічних компаній та спільноти в цілому збирається разом і вирішує, що має бути включене (і не включене) в специфікації ECMAScript. Те саме стосується GraphQL. Група людей зібралася разом і написала, що слід включити в мову (і залишити поза нею). Це служить орієнтиром для всіх реалізацій GraphQL.

Коли специфікація була випущена, творці GraphQL також поділились довідковою реалізацією сервера GraphQL в JavaScript – graphql.js. Це корисно як план, але мета цього довідкового втілення не вказує, яку мову потрібно використовувати для реалізації API. Це лише інструкція. Після того, зрозуміли мову запитів та систему типів, можна створити свій сервер будь-якою мовою, якою захочете.

Якщо специфікація та реалізація відрізняються, що насправді є в специфікації? Специфікація описує мову та граматику, які слід використовувати під час написання запитів. Він також встановлює систему типів, а також механізми виконання та перевірки системи цього типу. Крім цього, специфікація не особливо владна. GraphQL не диктує, яку потрібно мову програмування використовувати, як слід зберігати дані або яких клієнтів підтримувати.

Незважаючи на те, що GraphQL не контролює, як ви створюєте свій API, він пропонує деякі рекомендації щодо того, як думати про API:

1. Ієрархічно. Запит GraphQL є ієрархічним. Поля вкладені в інші поля, а запит має форму даних, які він повертає.

2. Продукто орієнтовано. GraphQL керується потребами клієнта в даних, а також мовою та часом роботи, які підтримують клієнта.

3. Типізовано. Сервер GraphQL підтримується системою типу GraphQL. У схемі кожна точка даних має певний тип, щодо якого вона буде перевірена.

4. Підтримка клієнтських запитів. Сервер GraphQL надає можливості, які клієнти можуть використовувати.

5. Самоаналіз. Мова GraphQL може запитувати систему типів серверів GraphQL.

У інтерфейсі прикладного програмування GraphQL отримати дані можна наступними способами (рис. 1.6):

1. Запит (Query) – це щось подібне до запиту GET у REST API. за допомогою Query ми можемо отримати дані з сервера.

2. Мутація (Mutation) – мутація в GraphQL використовується для оновлення даних на сервері. це схоже на запит POST на сервері.

3. Підписка (Subscription) – вона використовується для обробки зв'язку в режимі реального часу між клієнтом та сервером за допомогою веб-сокетів.

4. Визначення типів (Type Definitions). Визначення типів – це не що інше, як визначення запиту та формату відповіді для конкретного запиту.

Запит у GraphQL має наступну структуру (рис. 1.7) [11]:

1. Тип операції – треба визначити, яку операцію потрібно виконати. Наприклад, щоб отримати Дані, потрібно визначити його як запит (query).

2. Кінцева точка операції – вона визначає операцію, яку ви збираєтеся виконати, незалежно від того, це запит чи мутація.

3. Аргументи запиту – це щось на зразок параметра запиту.

4. Запитуване поле – воно визначає дані, які потрібно отримати з сервера.

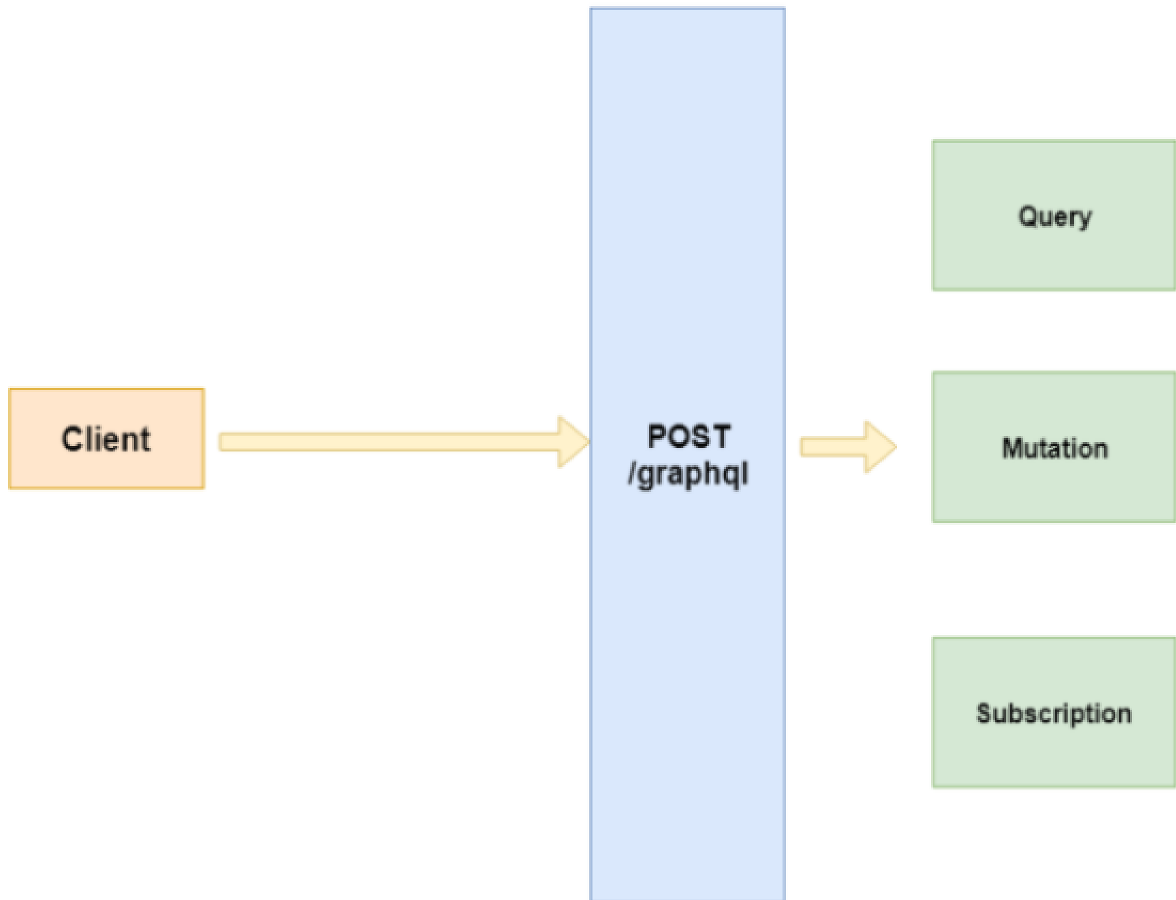

Рис. 1.6. Способи отримання даних за допомогою GraphQL

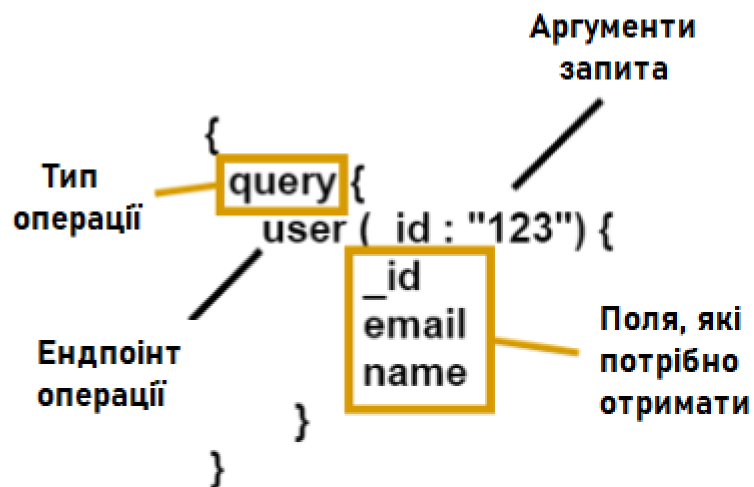

Рис. 1.7. Структура запиту GraphQL

1.3. Показники ефективності API

Моніторинг продуктивності є корисним інструментом для використання, коли потрібна додаткова інформація або необхідна оцінка результатів роботи обчислювального рішення або еталону. Частіше здається, що використання моніторинга продуктивності застосовується лише тоді, коли зустрічаються несподівані результати. З огляду на те, що більшість методів реалізації моніторингу продуктивності вимагають запуску додаткових програм або коду ядра паралельно із самим додатком або тестом, зрозуміло, що може існувати упередження або небажання використовувати можливості моніторингу продуктивності протягом усього періоду оцінки продуктивності, а також поширення його використання на виробниче середовище.

Для оцінки ефективності взаємодії клієнтського застосунку з API объектом для оцінки є запит. З боку інтерфейсу JavaScript отримує доступ до двох різних життєво важливих показників для запиту:

1. Перший показник – це затримка, яку можна виміряти, розмістивши два таймери до та після всіх запитів. За допомогою цієї техніки ми вкажемо, скільки часу пройшло між початком та кінцем усіх запитів.
2. Другий показник – це розмір відповіді. Додавши розмір байтів кожної відповіді, ми можемо визначити, який стандарт API є найлегшим з точки зору передачі даних. Однак міра розміру не відображатиме загальний розмір запиту. Через обмеження браузера інтерфейсний код не має доступу до значення розміру запиту та його заголовка, це означає, що інструмент може обчислювати лише розмір основного тексту.

Підводячи підсумок, у роботі будуть використані наступні показники ефективності:

1. Час виконання – визначає найшвидший, вимірюючи час, який буде потрібно стандарту для виконання всіх запитів.
2. Розмір тіла відповіді – вираховується шляхом додавання всіх розмірів тіл відповіді.

1.4. Висновки до першого розділу та постановка завдання

У ході виконання кваліфікаційної роботи потрібно відповісти на головне питання: чи є GraphQL швидшим та краще оптимізованим, ніж REST?

Для того, щоб відповісти на це питання, буде використаний кількісний підхід до дослідження. Кількісне дослідження використовує цифри та вимірювання, щоб зв'язати емпіричні спостереження з потенційною неупередженою відповіддю. Для збору цих цифр буде запропонована відтворювана методологія, щоб кожен бажаючий міг перевірити правильність роботи. Методологія буде реалізована у вигляді інструменту для тестування.

Прогнози та висновки цих досліджень обмежені невеликою кількістю випадків загального користування. Для виконання вимірювань можна уявити нескінченну кількість ситуацій, але не буде доречним представляти всі ці альтернативи, тому використовуються більш-менш реальні ситуації використання API.

РОЗДІЛ 2
ПРОЄКТУВАННЯ СЕРЕДОВИЩА ДЛЯ ТЕСТУВАННЯ ЕФЕКТИВНОСТІ API

2.1. Інструменти для створення клієнтського та серверного середовища

Оскільки API працюють з HTTP запитами – для створення клієнтського та серверного середовища обрана мова програмування JavaScript, тому що вона надає все необхідне для роботи з HTTP запитами прямо “з коробки”.

Для створення інтерфейсу клієнтської частини обрано фреймворк ReactJS, оскільки він так само, як і GraphQL, був розроблен у Facebook і вважається одним з найкращих фреймворків для створення SPA веб-сайтів.

Для серверної частини обрано NodeJS з фреймворком Express для прийняття та обробки HTTP запитів.

2.1.1. Мова програмування JavaScript

JavaScript спочатку був розроблений як мова для роботи з веб-сторінками за допомогою API, відомого як об'єктна модель документа (DOM) [12]. Спочатку створений у Netscape, він був названий JavaScript як спробу подолати хвилю популярності актуальноі мови програмування Java – рішення, яке роками спричиняло плутанину у нових розробників, оскільки мова має мало спільного з Java. У перші дні JavaScript в Інтернеті, Microsoft розробила власний варіант, відомий просто як JScript, який додав нові функції, такі як XMLHttpRequest (нині частина основної мови і часто скорочується до XHR), що дозволило розробникам робити запити до зовнішніх серверів для оновлення інформації на веб-сторінці. Європейська асоціація виробників комп'ютерів (ECMA) нарешті вирішила спробувати об'єднати ці конкуруючі реалізації в новий стандарт, який називається ECMAScript. Тому те, що в наш час називається JavaScript, насправді

є різними реалізаціями ECMAScript, саме тому можна почути такі терміни, як «ES6» або «ES2018», що використовуються для посилання на JavaScript (раніше він був версіями за номером, а зараз – за роком – ES2018 є дев'яте видання JavaScript). Вони стосуються різних специфікацій, опублікованих ECMA, яким різні браузери відповідають. У цьому розділі я посилаюся на версію мови ES2018. Для розробників, які раніше працювали з JavaScript, ES6 ввів деякі суттєві мовні зміни, такі як введення стрілочної функції.

Можливо, JavaScript досить молода мова програмування, але оскільки вона постійно вдосконалюється, деякі з найбільших проблем на сторонах розробників (особливо тих, що походять з інших мов) зникають, хоча деякі з них все ще залишаються.

Довгий час JavaScript у браузері використовувався лише для додавання додаткових удосконалень на сторінку, таких як вбудована перевірка форми або прості анімації. Це було частково тому, що механізми JavaScript у браузері просто не були досить швидкими, щоб підтримувати складні операції; хоча існували деякі ранні додатки, які були максимально налаштовані на продуктивність і, отже, були складними в побудові. Google змінив усе це, представивши движок V8, який значно пришвидшив JavaScript і дозволив вводити на сторінку набагато складніший код. Пізніше був запущений проект Node, який дозволив JavaScript працювати на сервері, а також нову стандартну бібліотеку для підтримки вимог на стороні сервера. На додаток до включення однієї мови як на стороні клієнта, так і на сервері, це також дозволило спільне використання коду між ними, представивши новий стиль програми, відомий як ізоморфний, або універсальний, JavaScript, де код є агностичним щодо того, де він запущений.

2.1.2. Асинхронність в JavaScript

Головне, що слід зрозуміти про JavaScript – це те, що він є однопотоковим і широко покладається на концепцію асинхронності [12]. Недоліком є те, що під

час виконання JavaScript виконання виконується «заблоковано». У браузері це означає, що весь браузер не буде реагувати на будь-яку взаємодію користувача під час роботи функції JavaScript. На сервері це означає, що інші підключення не будуть прийняті або оброблені.

Багато додатків JavaScript пов'язані з введенням-виводом, а не з процесором, тобто більшу частину цього часу витрачають на очікування зовнішньої події, а не на важкі обчислювальні операції. У таких середовищах такий підхід з єдиним потоком має багато сенсу, оскільки Node не потрібно активно чекати, поки відбудуться ці зовнішні події. Він може просто виконати функцію, а потім, коли ця функція закінчується, він може використовувати зворотні виклики та прослуховувачі подій для різних асинхронних дій. Коли відбуваються події, для яких були налаштовані ці обробники, ці функції можуть виконуватися. У проміжках між цими двома процесами будь-які інші зворотні виклики або прослуховувачі подій можуть виконуватися у відповідь на події, які відбулися або під час запуску оригінальної функції, або сталися, коли відсутній активний метод.

Це спрощує життя, оскільки поняття безпеки потоків не потрібно розглядати, і це виключає цілий клас помилок та складності. Якщо ви обробляєте зворотний виклик, вам не потрібно турбуватися про те, що раптом інша функція може запуститися всередині контексту і залишити якийсь спільний стан у несумісному або несподіваному режимі.

Поки користувач чекає виконання запиту HTTP, він може продовжувати взаємодіяти зі сторінкою відповідно. Це може спричинити ускладнення взаємодії з користувальницьким інтерфейсом. Якщо користувач виконав дію, яка вимагає чогось асинхронного, можливо, не завжди очевидно, що ця дія спрацювала, тому вони натискають на неї ще раз. Щоб уникнути цього, потрібно оновити інтерфейс користувача, щоб включити переривчастий стан "в очікуванні" або "в процесі" у відповідь на дію [12].

Основна проблема асинхронних дій полягає в тому, що часто важко переформатувати API, який раніше працював синхронно, в асинхронний. Це

іноді потрібно, коли спосіб роботи API під капотом змінився на асинхронний. Однак переробляти асинхронний API на синхронний набагато простіше, оскільки зворотний виклик можна просто викликати синхронно. З огляду на це, часто може бути простіше при розробці інтерфейсу в JavaScript зробити його асинхронним, якщо існує якийсь шанс, що йому потрібно буде стати асинхронним у майбутньому.

2.1.3. JavaScript у браузері

JavaScript був розроблений для роботи у веб-браузері, і протягом тривалого часу це було єдине місце, де він працював. Але JavaScript ніколи не був широко улюбленою мовою, і довгий час люди користувались іншими мовами для програмування веб-сайтів. Частково це сталося через помітні недоліки JavaScript та бажання стандартизувати один стек технологій. Це спричинило зростання плагінів веб-переглядача, таких як Flash та Java, але ці плагіни потрапили в немилість, оскільки напруженість між закритим вихідним кодом, власними плагінами та відкритим вихідним кодом у Інтернеті посилилась. JavaScript також еволюціонував для вирішення багатьох питань, через які стало непривабливим бути потужною мовою, якою він є сьогодні. Ще однією рушійною силою альтернативних мов було те, що багато частин функціональності потрібно було впроваджувати двічі – один раз на сервері (наприклад, для перевірки запитів форми) і знову на клієнті (для забезпечення кращої взаємодії з користувачем та швидшого реагування). Деякі фреймворки відповідали автоматичним генеруванням JavaScript для сторінки на основі логіки отриманої з сервера, але це створювало деякі проблеми. Автоматично згенерований код був гнучким і важким для налагодження, і він часто працював погано. З появою Node розробники отримали святий Грааль написання сухого («без повторень») коду, оскільки модулі могли спільно використовуватися між клієнтом і сервером. Це мало різний рівень успіху, оскільки контексти виконання на сервері та виконання

в браузері дуже різні і вимагали ретельної інженерії, щоб отримати цю можливість.

Альтернативним підходом були мови, які компілюються в JavaScript. Клієнтська частина служби GMail Google написана на Java, із компілятором, який видає JavaScript. Інші мови, такі як TypeScript та CoffeeScript, застосували той самий підхід і займають значні ніші, але більшість веб-розробок все ще здійснюється в самому JavaScript. Це частково тому, що ці альтернативні підходи можуть ускладнити налагодження, але також тому, що змішування бібліотек між різними мовами та необхідна додаткова робота над збіркою часто може додати більше ускладнень, ніж заощаджених зусиль. Бібліотека, відома як asm.js, була створена, щоб полегшити можливість компіляції в JavaScript, дозволяючи можливість використання JavaScript, на яку ці компілятори можуть націлити, що, теоретично, забезпечує високий рівень продуктивності. Ця оригінальна ідея переросла у стандарт, який називається WebAssembly, різновид низькорівневої мови збірки, яка може мати високу продуктивність, але не має доступу до DOM, тому не може повністю замінити JavaScript [13].

Важливим винятком, коли компіляція розпочалася, є випадок, коли JavaScript компілюється в іншу форму JavaScript, процес, відомий як транспіляція. Тривалий час JavaScript не підтримував модулі та пакети. Визначення асинхронних модулів (AMD) були першою спробою вирішити це питання, і вони працювали в браузері просто шляхом додавання бібліотеки, але Node використовував альтернативний метод для завантаження модулів, відомий як CommonJS. CommonJS не був сумісним із браузером, тому потрібні були інструменти для упаковки модулів CommonJS у сумісні з браузером скрипти (Browserify – найпопулярніший інструмент, що використовується для цього). Це означало, що більше не існує прямого зіставлення між кодом, написаним в IDE, і кодом, що працює в браузері. Ще до цього інструменти мініфікації використовувались для підвищення продуктивності, застосовуючи оптимізацію коду та надаючи один завантажуваний файл. Як і у випадку з іншими мовами, які компілюються на JavaScript, це ускладнювало відлагодження додатка. На

щастя, була введена техніка, відома як source maps (вихідні карти), яка дозволяла налагоджувачам у браузері допомогти зіставити зменшений або перетворений JavaScript назад до оригіналу. З цим відбувся наступний крок у транспіляції JavaScript.

Незважаючи на те, що багато людей використовували найновіші веб-переглядачі, і завдяки автоматичному оновленню залишались оновленими, все ще існувала значна частина людей, які використовували старіші браузери, або не могли чи не хотіли їх оновити. Для деяких людей це може бути результатом використання старих та непідтримуваних пристроїв, але для організацій політика часто диктує консервативний підхід до розгортання нового програмного забезпечення, поки воно не буде ретельно перевірено, тобто браузери можуть відставати. У міру того, як темпи розвитку Інтернету зростали, розробники хотіли використовувати ці нові функції, але були обмежені старими браузерами. Спочатку застосовували техніку, відому як поліфілінг. Polyfills – це сценарії, що надають версії чистого JavaScript нових API, доступних у JavaScript, і ефективно дозволяли людям використовувати нові API JavaScript у старих версіях JavaScript.

Однак поліфіли не можуть покрити всі діри – наприклад, неможливо додати щось на зразок Geolocation API до браузера, який просто не підтримує. Іншим обмеженням поліфілів є те, що вони фактично не можуть змінити основну мову. У міру розвитку нових версій JavaScript синтаксис змінювався, вводячи нові способи вираження класів та визначення анонімних функцій. Впровадження транспіляторів дозволило розробникам використовувати ці нові функції в коді, який вони написали, але для того, щоб їх можна було перевести в менш читабельний, але зворотний сумісний варіант, що дозволить виконувати їх у старих браузерах [13]. Це можна розглядати як більш вдосконалену версію автопрефіксів, що використовуються в CSS. Однак із цими перетворювачами з’являється все більше технологій, які наближаються до традиційного підходу до компілятора. Реакція Facebook розширює JavaScript на "JSX", що дозволяє вказати структуру HTML компонента React, використовуючи HTML-подібний

синтаксис. Ретельно продумане використання такої технології може бути корисним – у випадку React, оскільки ви можете використовувати JSX лише для написання компонентів React, тоді ви не вводите нової залежності, окрім використання React. Однак широке використання подібних технологій для інших цілей може призвести до уразливих ситуацій, оскільки тоді ви значною мірою залежите від нестандартного підходу для великих частин вашої програми, а не лише в ретельно обстеженому районі.

До впровадження поліфілів у JavaScript застосовувався інший підхід щодо зворотної сумісності. Частково це було пов’язано з відсутністю стандартизації між діалектами JavaScript, яка з’явилася набагато пізніше. Деякі з цих відмінностей були принциповими, наприклад, Mozilla та Internet Explorer, що мають дуже різні моделі для роботи з подіями JavaScript. Щоб впоратися з ними, було набагато простіше використовувати бібліотеку, щоб абстрагувати відмінності, так що ця бібліотека ставала звичайним інтерфейсом до браузера. Безумовно, найпоширенішою бібліотекою, яка використовувалась для цього, була JQuery, яка стала повсюдною в Інтернеті, присутній на більшості веб-сайтів на піку її популярності.

У міру розвитку Інтернету ця практика стала проблематичною. У сучасних браузерах рівень абстракції, запропонований JQuery, значно уповільнює роботу додатка, і багато функцій, які пропонує JQuery, тепер пропонуються безпосередньо самим JavaScript. Крім того, JQuery працює в глобальному просторі імен, що може спричинити проблеми під час спроби розробити слабозв'язані компоненти. JQuery все ще є корисним інструментом, і про нього не слід забувати, але в простих випадках не рідко можна побачити модулі, які безпосередньо взаємодіють з DOM, замість використання бібліотеки JQuery. Використовувати JQuery, коли це потрібно, можна, але треба пам'ятати, що за замовчуванням вона більше не потрібна.

Оскільки люди створюють все більше і більше додатків JavaScript, оригінальне дизайнерське рішення мати лише один потік та використовувати асинхронну модель, щоб підтримувати інтерфейс, що реагує, почало викликати

проблеми. Фронтенд розробники часто називають цю проблему "негідною". Сайт неприємний, коли на ньому все не відповідає плавно (наприклад, повільна анімація або плавна прокрутка, часто спричинена обробником події "scroll"). На щастя, сучасні браузери підтримують "веб-воркерів", які є способом перенесення структури потокового типу в JavaScript. Веб-воркерів можна сприймати більше як фонові процеси, ніж як нитки. Вони працюють у більш обмеженому середовищі без доступу до повного набору API браузера. Найголовніше, що веб-працівник не має доступу до DOM для прямої зміни сторінки. Також немає спільного стану, хоча повідомлення можуть передаватися вперед і назад між основним контекстом JavaScript і робочими.

2.1.4. Платформа Node.js

Як і JavaScript у браузері, JavaScript на стороні сервера є асинхронним, але побічні ефекти цього проявляються дещо інакше. Якщо JavaScript у браузері зайнятий, користувач не може взаємодіяти зі сторінкою, але якщо JavaScript на сервері зайнятий, тоді сервер взагалі не буде обробляти будь-які інші підключення – користувачам доведеться почекати, поки ця одна операція завершено. Хоча спочатку ця асинхронність може здатися важкою для розуміння (особливо для розробника, який звик мати справу з потоками та іншими методами паралелізації роботи), на практиці це набагато простіший та безпечніший підхід, ніж альтернативи інших мов, таких як потокової роботи, коли робота, яку потрібно виконати додатку, пов'язана з IO (тобто очікуванням відповідей інших систем, таких як диски, бази даних або сервери API), а не зв'язана з процесором. Багато веб-додатків дотримуються цієї моделі.

Основним недоліком однопотокової природи JavaScript є те, що на сервері з декількома процесорами обмежується використанням лише одного центрального процесора, і якщо хочеться використовувати всі центральні процесори в багатоядерній системі, потрібно буде запустити декілька процесів програми (на щастя, кластерний модуль Node дозволяє спільно використовувати

порт між усіма цими процесами). Це означає, що все, що зберігається в пам'яті, не обов'язково буде доступним у відповідь на майбутні запити користувача, оскільки інший процес у тому самому вікні може обслуговувати цей запит. Для зберігання будь-якої інформації, яка, можливо, доведеться зберігати більше кількох запитів, використовується зовнішній сервер кешування, такий як Memcached або Redis. Використання цього підходу є гарною практикою, оскільки, якщо програма стане дуже популярною, можливо, захочеться масштабувати її на декілька серверів, що знаходяться за балансиром навантаження, щоб обробляти всі запити. Завдяки зовнішньому серверу кешування настільки ж просто масштабувати до декількох вікон, як і масштабувати до різних процесів. Цей підхід називається горизонтальним масштабуванням, і він обговорюється далі в главі "Системи".

Найбільша різниця між JavaScript у браузері та JavaScript на сервері – це DOM [15]. На сервері його просто немає (і не було б сенсу, щоб він був!). Натомість Node має повністю окрему стандартну бібліотеку, яка забезпечує функції, які мають сенс на сервері, але не в браузері, наприклад, відкриття необроблених сокетів TCP та доступ до файлів. В іншому випадку мова та синтаксис однакові, але для розробників інтерфейсних програм, нових для кодування на стороні сервера, іноді дивно усвідомлювати, що такі речі, як глобальний об'єкт вікна, насправді не є частиною мови JavaScript, а лише DOM .

Звичайно, все ще можна відображати HTML розмітку, але це робиться за допомогою методів шаблонування, а не за допомогою побудови DOM дерева, або за допомогою віртуальної DOM-бібліотеки, яка поводиться як DOM, але видає HTML структуру.

Серверний JavaScript розвинув культуру невеликих, вільно пов'язаних бібліотек, на відміну від інших мов на стороні сервера, де є набагато більші всеохоплюючі фреймворки (наприклад, Java Spring, Laravel або PHP Symfony). Нерідкі випадки, коли у додатку є досить велика кількість залежностей. Це може трохи ускладнити швидкий старт, оскільки перед початком роботи доведеться

зв'язати низку речей, але це також надає надзвичайно велику гнучкість для побудови великої навантаженої системи, або саме того додатка, який потрібен.

2.1.5. Фреймворк Express

Node.js – це просто середовище виконання, яке може запускати JavaScript. Написати повноцінний веб-сервер вручну на Node.js безпосередньо не так просто і не потрібно. Express – це структура, яка спрощує написання коду для сервера.

Express – це найпоширеніший і гнучкий фреймворк для створення веб-додатків на Node. Він надає надійний набір функцій для роботи з SPA та RESTful API [16].

Основною причиною використання Express є те, що Node API не такий надійний для складних програм, оскільки кожне управління маршрутом та інші функції трактуються дуже елементарно. Express слідує передовим практикам розробки, орієнтованій архітектурі RESTful, яка використовує основні методи протоколу HTTP (такі як GET, POST, PUT та DELETE), і дуже важким для побудови складним веб-додаткам.

Основні переваги Express:

– потужне управління маршрутизацією;
– переспрямування для помічників;
– підготовлений до створення RESTful сервісів;
– висока продуктивність роботи.

Фреймворк Express дозволяє визначити маршрути, специфікації того, що робити, коли надходить запит HTTP, що відповідає певному шаблону. Специфікація відповідності базується на регулярних виразах і є дуже гнучкою, як і більшість інших фреймворків веб-додатків. Частина маршрутка із логікою, яку потрібно виконати – це просто функція, яка отримує проаналізований HTTP-запит.

Express розбиває на логічні частини (парсить) URL-адреси, заголовки та параметри. Що стосується відповіді, він має, як і слід було очікувати, всю

функціональність, необхідну веб-програмам. Це включає встановлення кодів відповідей, встановлення файлів cookie, надсилання користувацьких заголовків, тощо. Крім того, ви можете писати проміжне програмне забезпечення Express, яке являє собою спеціальні фрагменти коду, які можна вставити в будь-який шлях обробки запиту / відповіді, щоб досягти загальних функціональних можливостей, таких як реєстрація, аутентифікація, тощо.

Express не має вбудованого механізму шаблонів, але він підтримує будь-який механізм шаблонів на ваш вибір [16], наприклад, PUG, Mustache тощо. Але для SPA не потрібно використовувати механізм шаблонів на стороні сервера. Це пов'язано з тим, що генерування всього динамічного вмісту здійснюється на клієнті, а веб-сервер обслуговує лише статичні файли та дані за допомогою викликів API. Особливо у стеці MERN, генерацією сторінок займається сам React на стороні сервера.

Таким чином, Express – це веб-серверний фреймворк, призначений для Node.js, і він не сильно відрізняється від багатьох інших серверних фреймворків з точки зору того, чого можна досягти за допомогою нього.

2.1.6. Фреймворк ReactJS

В останні роки популярними стали односторінкові програми (SPA). Такі фреймворки, як Angular, Ember та Backbone, допомогли розробникам JavaScript створювати сучасні веб-додатки, без використання jQuery. Список таких фреймворків не є вичерпним. Існує широкий спектр SPA-систем. Якщо врахувати дати випуску, більшість із них належать до першого покоління SPA: Angular 2010, Backbone 2010, Ember 2011.

Початковий випуск React був у 2013 році Facebook. React вважається навість не фреймворком SPA, а бібліотекою для переглядів [17]. Це V у MVC (контролер перегляду моделі). Це лише дозволяє відображати компоненти які видимі у браузері. Проте ціла екосистема навколо React дозволяє створювати односторінкові додатки.

Паттерн Model View Controller (MVC) розділяє додаток на три основні аспекти (рис. 2.1): модель (Model), вигляд (View) та контролер (Controller) [18].

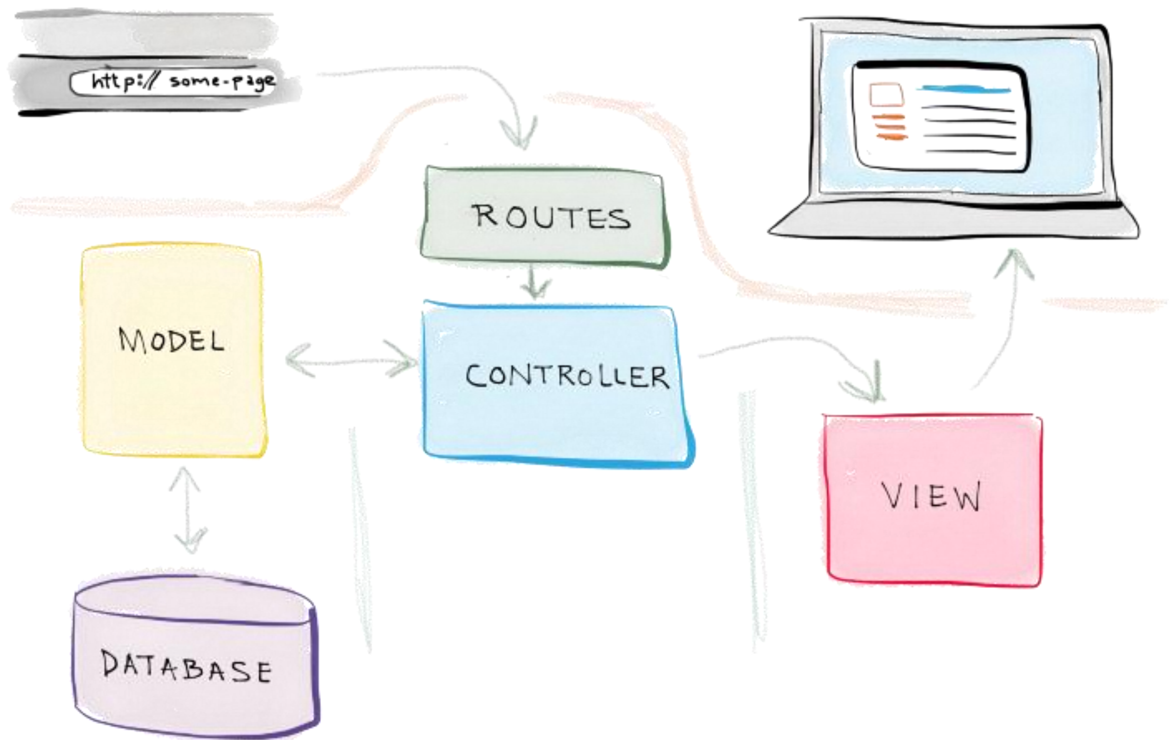

Рис. 2.1. Структура паттерна Model View Controller

Модель означає дані, які необхідні для відображення у поданні. Модель представляє сукупність класів, що описують бізнес-логіку (бізнес-модель та модель даних). Він також визначає бізнес-правила для засобів даних, як те, як дані можна змінювати та маніпулювати ними.

Перегляд представляє такі компоненти інтерфейсу, як XML, HTML тощо. Перегляд відображає дані, отримані від контролера, як результат. У патерну MVC View відстежує модель для будь-яких змін стану та відображає оновлену модель. Модель та перегляд взаємодіють між собою, використовуючи шаблон Observer.

Контролер відповідає за обробку вхідних запитів. Він обробляє дані користувача за допомогою Моделі та передає результати у Перегляд. Зазвичай він виступає посередником між Переглядом і Моделлю.

Хоча перше покоління фреймворків намагалося вирішити багато речей одночасно, React допомагає лише побудувати ваш шар перегляду. Це бібліотека, а не фреймворк.

У React можна зосередитись на поданні інтерфейсів, перш ніж вводити більше аспектів у свою програму. Кожен інший аспект – це ще одна складова вашого SPA. Ці будівельні блоки необхідні для створення повноцінного додатка. Вони мають дві переваги:

1. Спочатку можна поетапно вивчити “будівельні блоки”, з яких будується веб-сайт. Не потрібно турбуватися про те, щоб зрозуміти їх взагалі. React відрізняється від фреймворку, який дає вам кожен блок з самого початку.

2. По-друге, всі “будівельні блоки” взаємозамінні. Це робить екосистему навколо React таким інноваційним місцем. Кілька рішень конкурують між собою. Можна обрати найбільш привабливе рішення для себе та конкретного випадку використання.

Перше покоління SPA-систем вийшло для підприємств. Вони більш жорсткі. React залишається інноваційним і приймається на озброєння багатьма компаніями-лідерами технологічних думок, такими як Airbnb, Netflix і, звичайно, Facebook. Усі вони інвестують у майбутнє React і задовольняються React та самою екосистемою.

React має багато переваг, але однією з головних переваг є швидкість роботи бібліотеки. React був створений для пришвидшення процесу відображення (рендеру). За допомогою чистого JavaScript будь-які зміни веб-сторінки безпосередньо передаються до браузера DOM (об'єктна модель документа). Цей процес візуалізації є обтяжливим для браузера, оскільки йому доводиться щоразу переосмислювати весь документ. Для вирішення цієї проблеми React представив свій віртуальний DOM. Процес візуалізації суттєво відрізняється, оскільки кожна модифікація робиться у цьому віртуальному DOM перед прийняттям рішення про необхідність оновлення реального DOM (рис. 2.2) [19]. Якщо виявляється різниця між цими двома DOM, зміна відображатиметься в реальному DOM.

Таким чином, React зменшує кількість рендерів у реальному DOM і економить багато ресурсів.

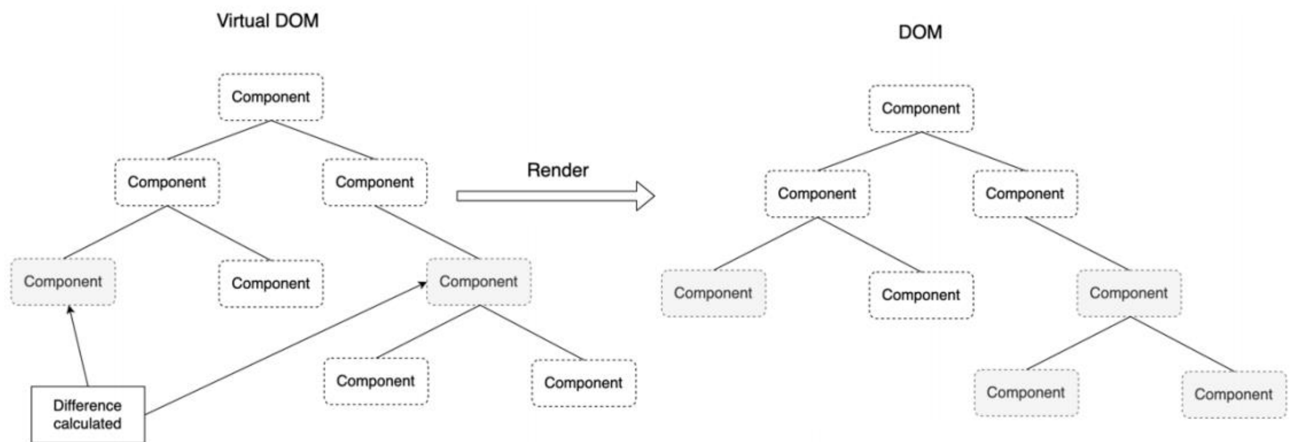

Рис. 2.2. Процес взаємодії Virtual DOM з DOM

React – це, мабуть, один із найкращих варіантів для створення односторінкових додатків сьогодні. Він забезпечує лише шар перегляду, але екосистема React – це цілий гнучкий та взаємозамінний каркас. React має тонкий API, дивовижну екосистему та чудову спільноту. Ви можете прочитати про мій досвід, чому я перейшов з Angular у React. Я настійно рекомендую зрозуміти, чому ви вибрали б React замість іншого фреймворку чи бібліотеки.

2.2. Клієнти GraphQL для клієнтського середовища

Перш за все, клієнському середовищу потрібен клієнт для взаємодії з серверною частиною. На ринку існує безліч різних рішень, які можуть краще працювати з конкретними технологіями.

Ось короткий перелік найпопулярніших клієнтів:

– Apollo Client: мабуть, найвідоміший клієнт JavaScript;

– GraphQL Request: легкий клієнт, вважається найзручнішим у використанні;

– Relay Modern: покращена ітерація першого клієнта Relay, побудованого Facebook на самому початку розвитку GraphQL.

Вибір клієнта може мати вирішальне значення для програми, оскільки кожна реалізація може включати чи ні деякі функції. Наприклад, GraphQL не має власного кешування, як архітектура REST, оскільки він не покладається на HTTP, який може запропонувати можливість кешування запиту автоматично.

Деякі клієнти, такі як Apollo, чудово реалізують кешування, тому дуже важливо грамотно вибирати клієнт.

2.2.1. Клієнт GraphQL Request

GraphQL Request відомий як Ізоморфний, оскільки він працює як з Node, так і з браузерами. Це мінімальний клієнт GraphQL, що підтримує Node, браузери та прості програми, і включає декілька корисних функцій. GraphQL Request – це легкий і простий клієнт GraphQL, який підтримує async та await [21]. Що ще важливіше, він підтримує Typescript.

Основні переваги GraphQL Request:

– найпростіший і легкий клієнт GraphQL;
– API на основі промісів (працює з async / await);
– підтримка Typescript;
– ізоморфний (працює з NodeJS та браузерами).

2.2.2. Клієнт Apollo Client

Apollo Client – це взаємодіючий, надзвичайно гнучкий, керований спільнотою клієнт GraphQL для JavaScript та власних платформ [20]. Його вражаючі функції включають надійний інструмент управління станом, систему кешування з нульовою конфігурацією, декларативний підхід до отримання даних, просту у реалізації пагінацію та оптимістичний інтерфейс для клієнтського додатку.

Клієнт Apollo завоював повагу стількох інженерів та компаній завдяки надзвичайно корисним функціям, які роблять створення сучасних надійних програм легким.

Основні функції Apollo Client [20]:

1. Кешування. Клієнт Apollo підтримує кешування.
2. Optimistic UI. Optimistic UI – це шаблон, який можна використовувати для моделювання результатів мутації та оновлення інтерфейсу ще до отримання відповіді від сервера. Як тільки відповідь отримана від сервера, оптимістичний результат викидається і замінюється фактичним результатом. Клієнт Apollo має круту підтримку Optimistic UI. Він передбачає тимчасове відображення остаточного стану операції (мутації) під час виконання операції. Після завершення операції реальні дані замінюють оптимістичні дані.
3. Пагінація. Apollo Client має вбудовану функціональність, яка дозволяє досить легко впровадити пагінацію у додатку, яка супроводжується багатьма технічними головними болями отримання списку даних, або в патчах, або відразу, використовуючи функцію fetchMore, яка постачається з хуком useQuery.

Apollo Client зберігає не тільки стан із даних, отриманих із сервера, але і стан, який він створив локально на клієнті; отже, він управляє станом як для даних API, так і для локальних даних [20].

Важливо також зазначити, що можна використовувати Apollo Client разом з іншими інструментами управління сховищем без конфліктів, наприклад з Redux. Зрештою, основною метою Apollo Client є надання інженерам плавного запиту даних в API.

Apollo Client використовується для отримання даних з будь-якого сервера GraphQL. Клієнт невеликий, але при цьому гнучкий з безліччю чудових функцій, серед яких найбільш вдячною може бути автоматичне оновлення кешу, яке постачається з клієнтом.

По суті, клієнт Apollo автоматично перевіряє від вашого імені запити та трафік мутацій та використовує останню версію даних, які він бачить у відповіді, так що локальний кеш завжди оновлюється (рис. 2.3).

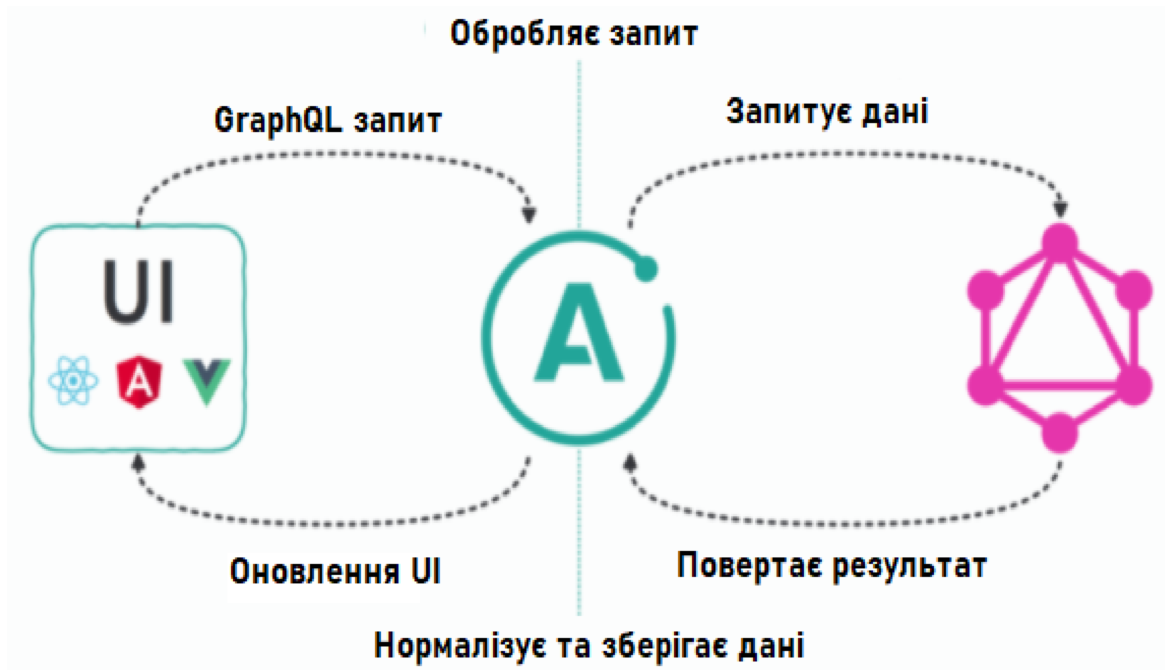

Рис. 2.3. Схема кешування даних за допомогою Apollo Client

2.2.3. Клієнт Relay Modern

Relay – це клієнтська структура GraphQL. Взагалі, клієнтська структура або бібліотека GraphQL дозволяє розробникам усунути вибірку даних, що передаються, і декларативно передавати дані з API GraphQL в компоненти інтерфейсу [22]. У додатку, який створений за допомогою фреймворку React, він буде існувати в схожому на щось на зразок глобального сховища Redux або MobX, і часто може повністю замінити ці системи. Relay також має декількох прямих конкурентів в інтерфейсному просторі вибору GraphQL, включаючи власну бібліотеку Apollo Client для React. Вони поділяють багато однакових цілей і особливостей, але роблять різні компроміси.

Relay Modern – це редизайн ядра та API Relay, з метою досягнення тих самих основних принципів декларативного отримання та вимог до модульних даних, одночасно обрізавши деякі функції, які зробили оригінальну реалізацію більш складною та важчою для оптимізації [22].

Основна ідея Relay полягає в тому, що отримання даних має бути таким же модульним, як і інтерфейс. Це досягається за допомогою дрібного використання

фрагментів GraphQL. Relay призначений для того, щоб дозволити розробникам розміщувати кожен компонент React разом із невеликим фрагментом GraphQL в одному файлі, що дозволяє дуже легко забезпечити отримання завжди потрібних даних для цього компонента. Ці фрагменти можуть бути складені у більші фрагменти і, нарешті, запити, щоб отримати всі дані, необхідні для певного подання, з мінімальною кількістю запитів до сервера.

2.3. Клієнти GraphQL для серверного середовища

Що стосується інтерфейсу, у серверній системі є безліч рішень для розгортання та розвитку сервера GraphQL. На ринку проектів з відкритим кодом можна знайти безліч реалізацій багатьма мовами. З цієї причини в наступному фрагменті будуть розглянуті загальні концепції, перенесені на серверну систему і показані основні приклади в NodeJS:

– GraphQL.js: це еталонна реалізація клієнта для серверної частини, яка створена розробниками GraphQL;

– Express-graphql: реалізація серверного клієнта GraphQL, яка може працювати разом з веб-сервером Express;

– Apollo-server: аналог Apollo Client, але для серверної частини.

2.3.1. Клієнт GraphQL.js

GraphQL.js – це бібліотека загального призначення від розробників GraphQL, яка може використовуватися як на сервері NodeJS, так і в браузері. GraphQL.js надає дві основні можливості: побудова схеми типу та обслуговування запитів цієї схеми типу.

Більшість серверних клієнтів GraphQL побудовані на основі GraphQL.js.

2.3.2. Клієнт Apollo-server

Apollo Server – це сервер GraphQL із відкритим вихідним кодом, сумісний зі специфікаціями, який сумісний з будь-яким клієнтом GraphQL, включаючи Apollo Client [23]. Це найкращий спосіб створити готовий до самостійного самодокументування API GraphQL, який може використовувати дані з будь-якого джерела (рис. 2.4). Він працює з усіма серверними платформами Node.js HTTP: Express, Connect, Hapi, Koa, AWS Lambda, Restify та Micro.

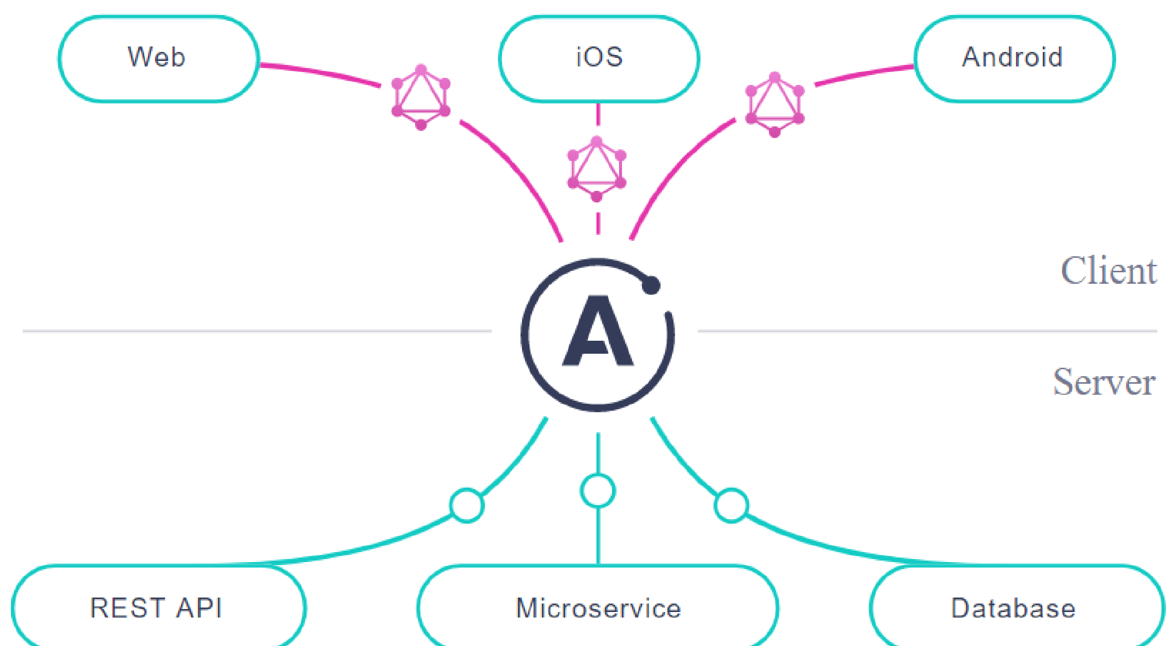

Рис. 2.4. Структура роботи Apollo Server

Сервер Apollo побудований з урахуванням таких принципів:

1. Спільнота для спільноти: Розвиток Apollo Server керується потребами розробників.

2. Простота: сервер Apollo простий у використанні, легко підтримується та безпечніший.

3. Продуктивність: Сервер Apollo добре перевірений і готовий до виробництва – модифікації не потрібні.

Apollo Server можна використовувати як:

– автономний сервер GraphQL, в тому числі в безсерверному середовищі;

– додаток до існуючого проміжного програмного забезпечення Node.js (наприклад, Express або Fastify);

– шлюз для об'єднаного потоку даних.

Apollo Server забезпечує [23]:

– пряме налаштування, тому розробники клієнтів можуть швидко почати отримувати дані;

– поступове прийняття, що дозволяє додавати функції за потребою;

– універсальну сумісність з будь-яким джерелом даних, будь-яким засобом побудови та будь-яким клієнтом GraphQL;

– готовність до виробництва, що дозволяє швидше доставляти функції.

2.3.3. Клієнт Express-graphql

Express-graphql – це проміжне програмне забезпечення (middleware) для NodeJS, який дозволяє використовувати GraphQL разом з веб-фреймворком Express, який дозволяє структурувати веб-додаток для обробки декількох різних http-запитів за певною адресою [24].

Middleware – це будь-яка кількість функцій, які викликаються на рівні маршрутизації Express.js до виклику кінцевого обробнику запиту, і, отже, знаходиться посередині між необробленим запитом та кінцевим запланованим маршрутом.

2.4. Проєктування клієнтського середовища

Для чіткого розуміння того, як повинен працювати тестове середовище необхідні вимоги та проєктування блок-схеми.

Блок-схема – це графічне представлення кроків виконання програми. Вона показує кроки в послідовному порядку і широко використовується для

проєктування алгоритмів, робочого процесу або процесів. Як правило, блок-схема показує кроки як вікна різного типу, з'єднуючи їх стрілками.

За допомогою спроектованої блок-схеми та вимог буде відомо, куди слід помістити вимірювання і переконатися, що результати будуть вирахувані правильно.

Вимоги до клієнтського середовища:

– повинна бути можливість налаштувати послідовність викликів API як для REST, так і для GraphQL;

– необхідно мати можливість збільшити кількість повторень запитів для кожного виклику, щоб імітувати активність декількох користувачів;

– повинна бути можливість паралельного та послідовного виконання запитів;

– середовище повинно дотримуватися затримки та загальний розмір кожного стандарту.

Для реалізації клієнтської частини роботи з GraphQL API обрано Apollo Client через його легкість та офіційну підтримку від розробників GraphQL, робота з REST API виконуватиметься за допомогою вбудованих інструментів мови JavaScript.

Для проведення замірів за обраними показниками ефективності (часу виконання запиту та розміра тіла відповіді) можна виділити 2 алгоритми поведінки:

1. Тестування за допомогою паралельних запитів (рис. 2.5). Емулюється ситуація, коли одночасно декілька користувачів намагаються скористатися запропонованим API. Усі запити будуть запущені одночасно, програмне забезпечення повинно зачекати, доки всі запити завершать виконання і після цього зберегти дані за обраними показниками.

2. Тестування за допомогою послідовних запитів (рис. 2.6). Емулюється ситуація, коли одному користувачу необхідно завантажити, наприклад, інформацію про публікацію та коментарі до неї. Запити будуть запущені по

одному, програмне забезпечення повинно зачекати, доки всі запити завершать виконання і після цього зберегти дані за обраними показниками.

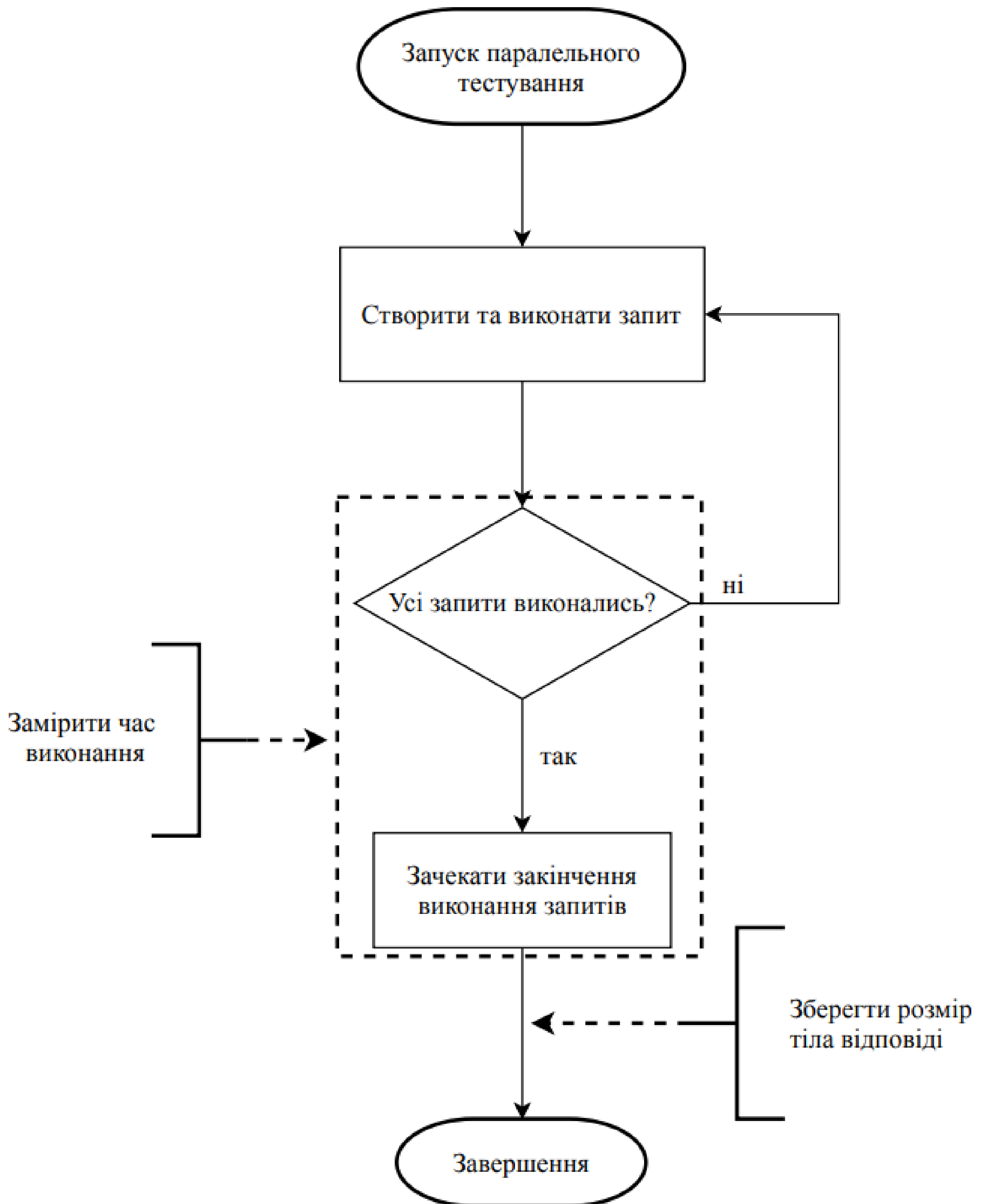

Рис. 2.5. Алгоритм паралельного тестування

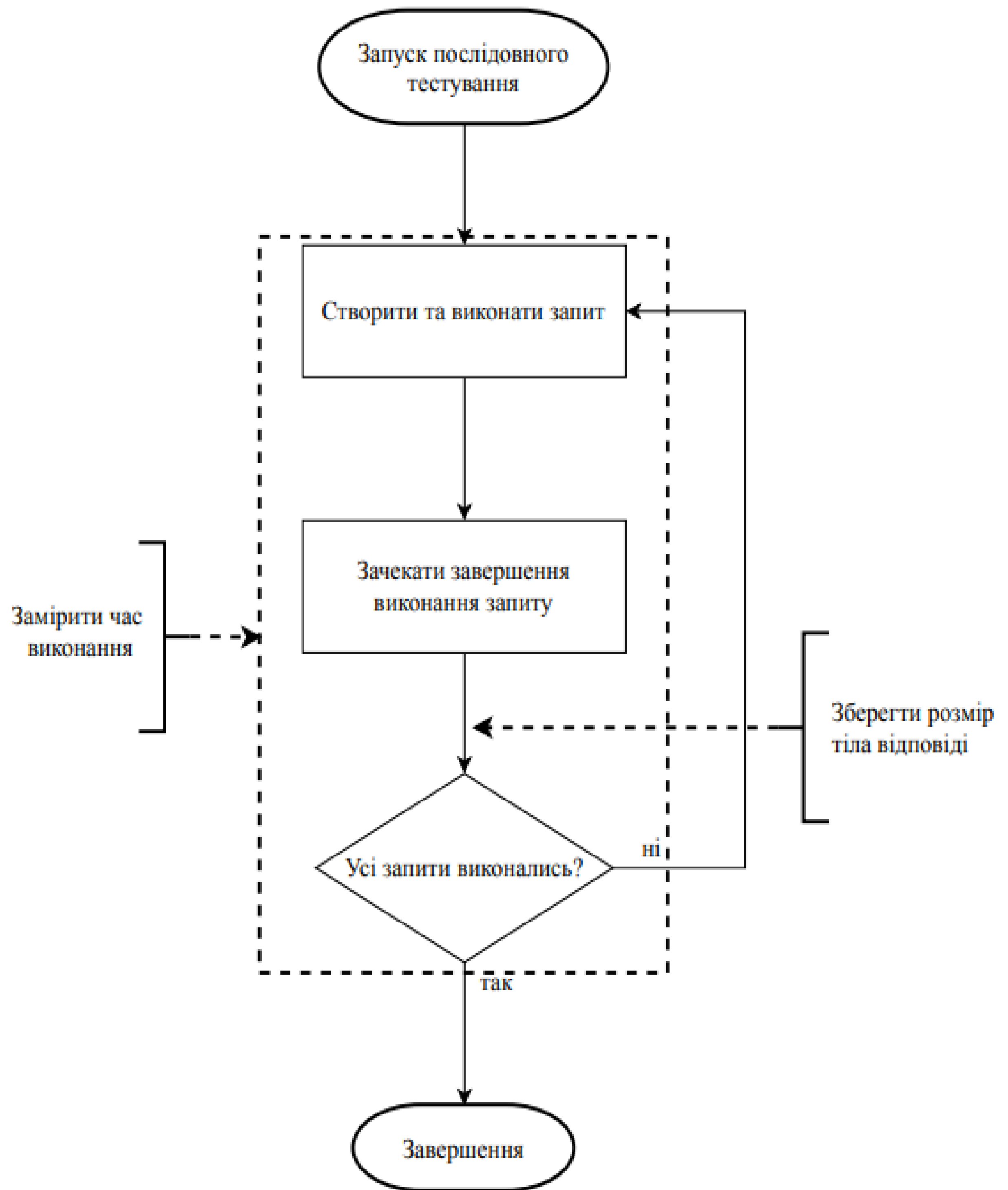

Рис. 2.6. Алгоритм послідовного тестування

Повний алгоритм роботи клієнтського середовища можна побачити на рис. 2.7.

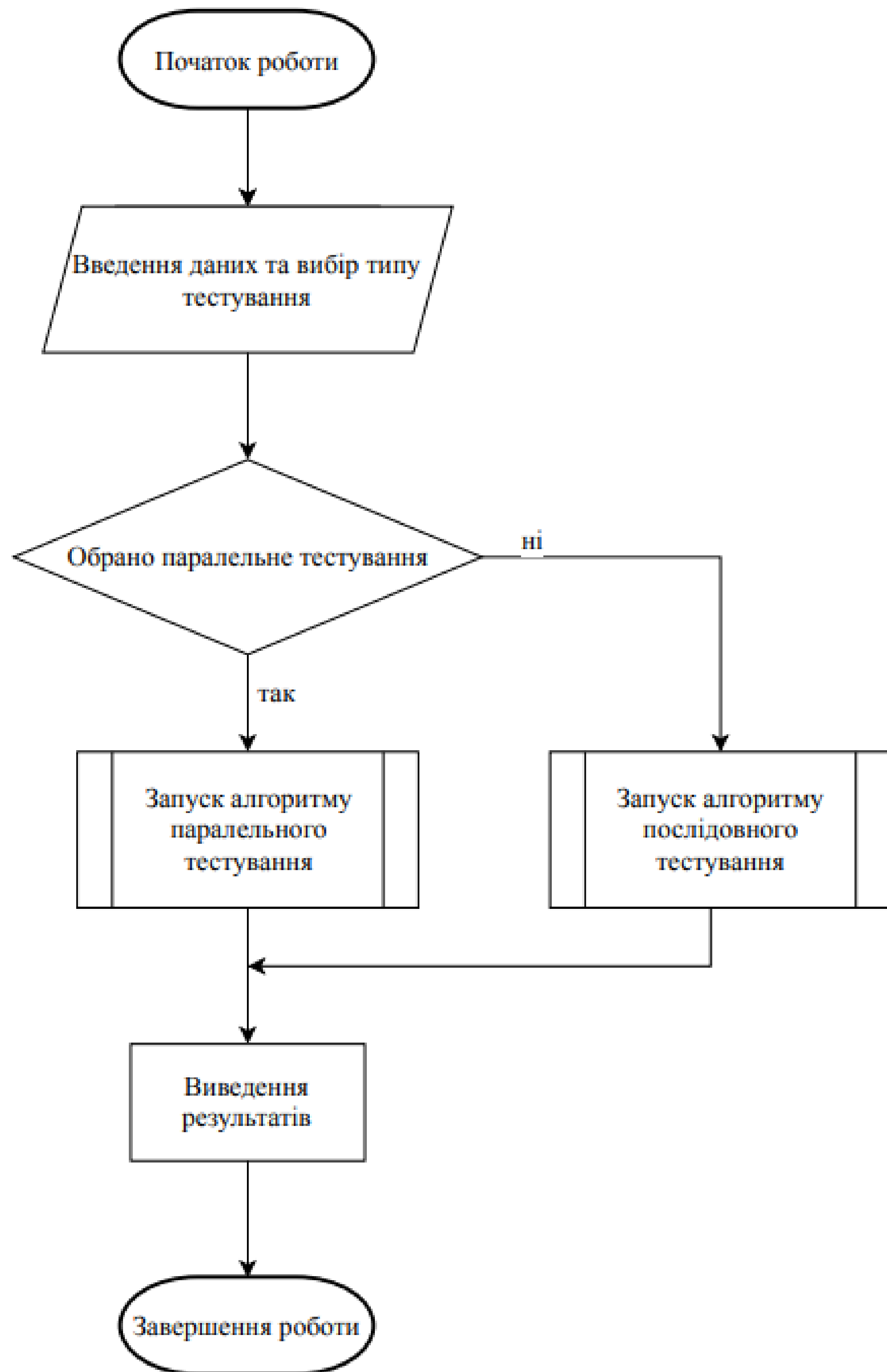

Рис. 2.7. Алгоритм роботи клієнтського середовища

Дизайн для тестування може здатися марною тратою часу, але приємний дизайн допоможе подати результати тестування більш наочно. На рис. 2.8

показан сформований згідно поставлених вимог концептуальний дизайн клієнтського середовища.

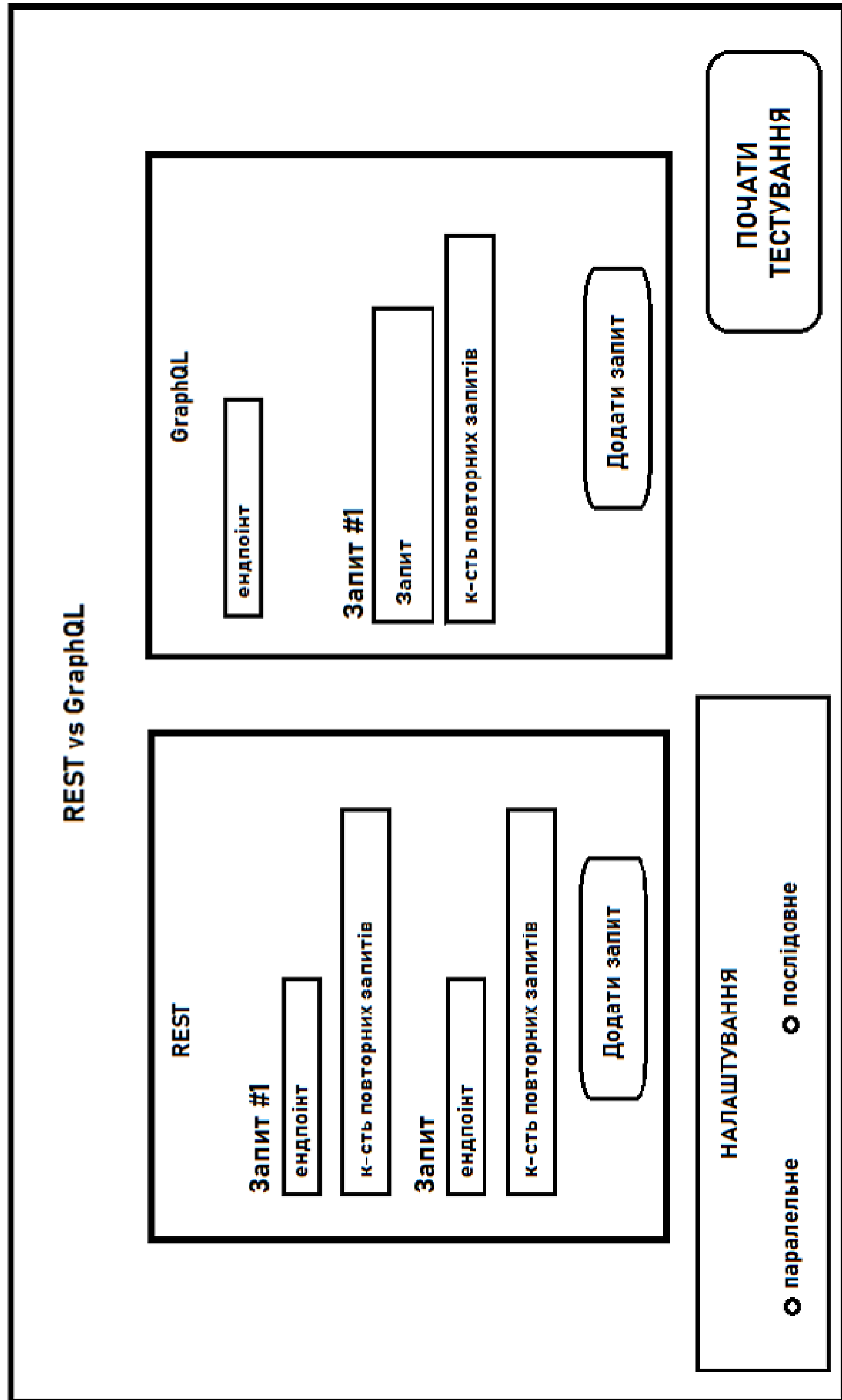

Рис. 2.8. Концептуальний дизайн клієнтського середовища

Весь процес створення базового інтерфейсу не мав би сенсу, якщо результати представлені погано. Мета полягає в тому, щоб представити два стандарти поряд для кожної метрики. Кожне вимірювання буде мати графічне зображення, яке допоможе користувачеві чітко визначити частку кожного результату на перший погляд. Користувач також отримає доступ до точного виміряного значення, щоб використати ці результати для подальшого використання.

Як можна побачити на рис. 2.9, кожен результат фіксує час завершення кожного виконання тесту. Ця інформація може знадобитися як частина контекстної інформації, щоб вказати користувачеві, що тест був проведений саме в цей момент, коли мережа та сервери переживали певну ситуацію в даний момент часу. Це те, що слід враховувати при аналізі результатів. Більше того внизу вікна з результатами знаходиться невелике речення, яке пояснює отримані результати словами для нетехнічних користувачів.

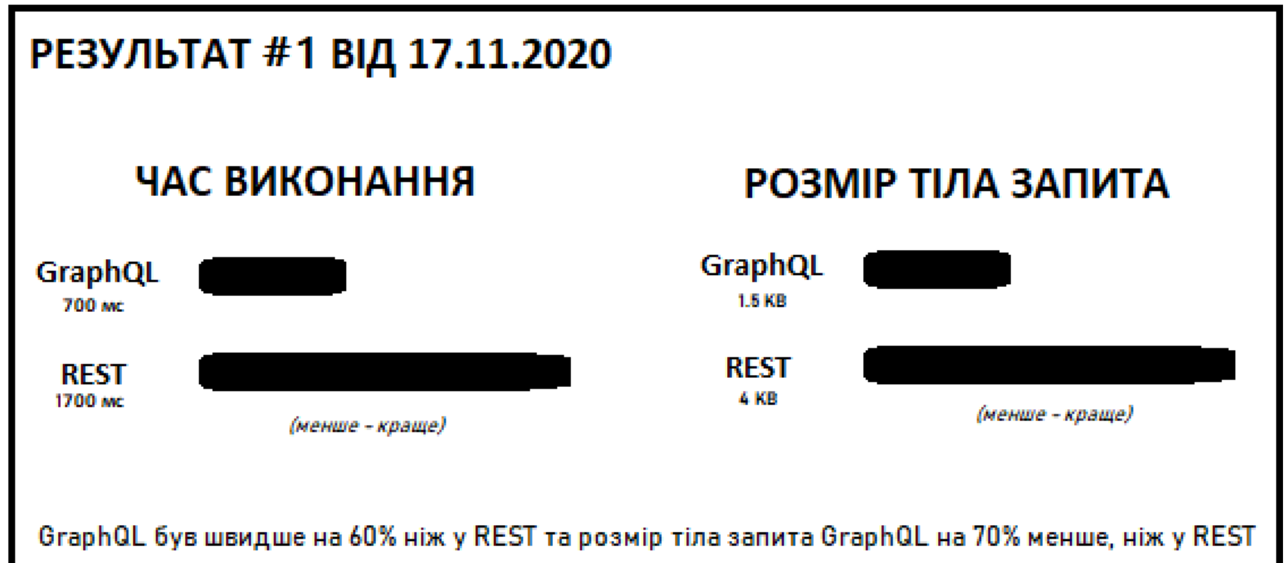

Рис. 2.9. Концептуальний дизайн результатів тестування

2.5. Проєктування серверного середовища

У рамках дослідження ефективності використання GraphQL API у порівняння з REST API, серверна частина не потребує складної архітектури.

Сервер буде отримувати та обробляти запити від клієнта, на кожен запит заздалегідь створені набори тестових даних, які будуть у хаотичному порядку надсилатись як результат виконання запиту (відповідь).

Для реалізації серверної частини обрано сервер GraphQL Express-graphql через його сумісність з фреймворком Express, який обрано як основний фреймворк для зручної роботи з запитами у NodeJS. Робота з кінцевими точками та запитами REST API відбуватиметься виключно за допомогою фреймворку Express, який має все необхідне для цього.

2.6. Висновки до другого розділу

У розділі розглянуті основні інструменти для створення REST та GraphQL API, розглянуті їх переваги та недоліки, а також спроектоване клієнтське і серверне середовище для тестування ефективності використання GraphQL API у порівнянні з REST API за допомогою обраних показників ефективності.

Тестування відбуватиметься за двома алгоритмами:

1. Тестування за допомогою паралельних запитів.
2. Тестування за допомогою послідовних запитів.

Для реалізації інтерфейсів клієнтського середовища обрано JavaScript фреймворк ReactJS та для роботи з GraphQL API клієнт Apollo Client. Робота з REST API виконуватиметься за допомогою вбудованих в мову JavaScript інструментів.

Для реалізації серверної частини обрано сервер Express-graphql для взаємодії з GraphQL та фреймворк Express для зручної роботи із запитами REST.

РОЗДІЛ 3
ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ GraphQL У ПОРІВНЯННІ З REST

3.1. Подібності і відмінності GraphQL та REST

І GraphQL, і REST працюють навколо ідеї ресурсів, заданих ідентифікаторами. У REST ідентичність об’єкта, а також форма та розмір визначаються в кінцевій точці запиту, тоді як у GraphQL формат визначається клієнтом. Обидва API мають список можливих дій із розділенням операцій читання та запису. REST має список кінцевих точок, що починаються з GET, POST, або іншого HTTP методу, тоді як GraphQL має схему запитів та мутацій.

Хоча GraphQL та REST це зовсім різні підходи до роботи із запитами та даними, але в чомусь вони подібні. Більш конкретно:

– обидва засновані на концепції ресурсу і можуть вказувати ідентифікатори ресурсів;

– вони обидва можуть бути використані за допомогою HTTP-запитів;

– обидва можуть повернути дані у форматі JSON.

Також GraphQL та REST в дечому відрізняються:

– кінцева точка, яка викликається в REST, повертає об'єкт у статичному форматі, тоді як кінцева точка у GraphQL нічого не знає про те, що потрібно повернути. Іншими словами, у REST ви визначаєте структуру об'єкта на серверній частині, а у GraphQL визначається структуру об'єкта на клієнтській частині;

– за допомогою REST сервер визначає форму та розмір ресурсу, тоді як у GraphQL сервер просто декларує доступні ресурси, і клієнт може запитати саме те, що йому потрібно;

– REST автоматично кешує дані, тоді як GraphQL не має автоматичної системи кешування (за випадком використання клієнтів з реалізованим кешуванням);

– обробка помилок у REST набагато простіша порівняно з GraphQL, який зазвичай дає вам код стану 200 OK.

3.2. Тестування ефективності за обраними показниками

Етап тестування та збору результатів повинен бути виконан обережно, для того, щоб можна було отримати з нашого вимірювання максимально точні значення. Для цього потрібен план збору даних:

1. Скопіювати та зберегти кожний показник з кожного результату.
2. Зробити докази правдивості результатів виконаних тестів — зробити знімки екрана із відображенням дати та часу виконання.

Збережені докази правдивості даних захищають нас від будь-якої втрати або неточності даних. Більше того, це надає змогу читачеві звіту відстежувати цифри від остаточного висновку до початкового вимірювання.

Окремий результат не є надійною основою для висновків. Оскільки результати можуть відрізнятись через вплив побічних факторів, необхідно виконувати декілька вимірювань, щоб позбутися цих прогалин. Чим більше вимірювань буде зроблено тим точнішим буде кінцевий результат.

Зібрані результати будуть порівняні для того, щоб сформулювати будь-які висновки. Середнє значення по кожному показнику легко порівняти між двома стандартами, розмістивши їх поруч і подивившись, яке значення є найнижчим. Додатково буде представлена різниця між двома технологіями вказана у відсотках. Для цього присутня текстова інформація у наступному форматі: API_1 був швидшим на X%, ніж API_2 та розмір тіла відповіді API_1 на Y% менше, ніж у API_2. Це формулювання робить результат кристально чистим навіть для нетехнічної людини.

3.2.1. Тест №1

У першому тесті емулюється робота невеликого стартапу, який пропонує додаток для соціальних мереж. Компанія готова перевірити запити на одній із своїх сторінок за допомогою GraphQL, щоб визначити, чи можуть вони очікувати збільшення швидкості, переходячи до цієї нової технології. На сторінці повинні відображатися повідомлення від конкретного користувача та його коментарі. До використання GraphQL за допомогою REST API їм потрібно виконати два запита з інтерфейсу для того, щоб отримати всі дані (рис. 3.1).

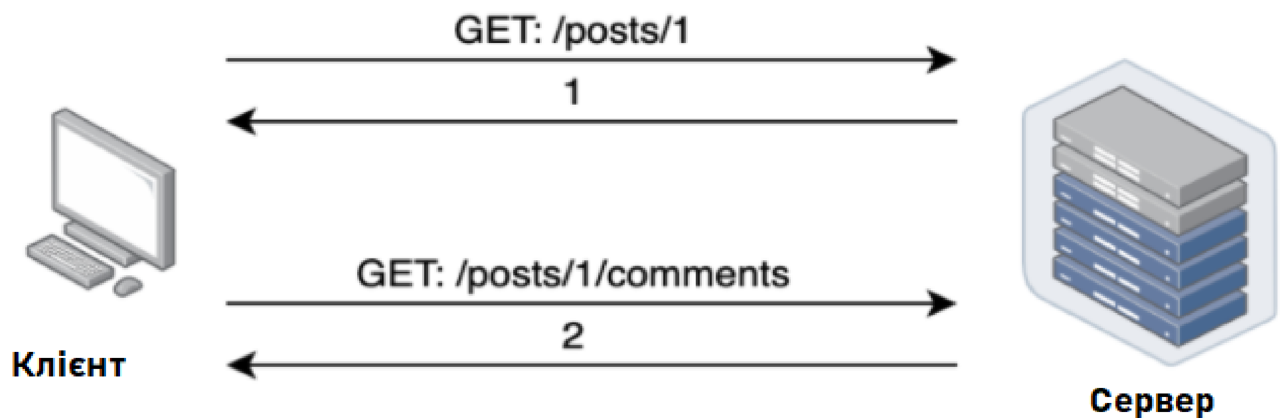

Рис. 3.1. Схема взаємодії з REST API у тесті №1

GraphQL повинен оптимізувати процес отримання даних за допомогою об’єднання двох запитів в один запит. Перш ніж отримати доступ до результатів цього порівняння, слід встановити технічні характеристики нашого шаблону, визначені в частині методології нашого аналізу. Всі необхідні технічні характеристики для теста №1 наведені у табл. 3.1.

Таблиця 3.1

Технічні характеристики до тесту №1

Назва характеристики	Значення
Локація сервера	Берлін, Германія
Локація клієнта	Дніпро, Україна

Продовження таблиці 3.1

Назва характеристики	Значення
Швидкість інтернет-з'єднання між клієнтом та сервером	60 Мб/с
REST запити	GET: /posts/1 Повторень: 1 GET: /posts/1/comments Повторень: 1

GraphQL запит	{
	posts {
	name
	content
	postDate
	likesCount
	imageUrl
	author {
	username
	}
	comments {
	content
	commentDate
	user {
	username
	}
	}
	}
	}
	
	Повторень: 1

Порядок виконання запитів	Послідовний

Проаналізувавши отримані в результаті тестування дані, які розташовані у табл. 3.2, можна помітити, що розрив між виміряними значеннями був мінімальним. З цієї причини, вибірка обмежена десятьма результатами, оскільки більша кількість результатів додасть зайвої точності. Знімки екрану з результатами тесту №1 наведено у Додатку А (рис. А.1 - А.10).

Таблиця 3.2

Результати теста №1

Номер виміру	Дата та час	Час виконання GraphQL (мс)	Час виконання REST (мс)	Розмір тіла відповіді GraphQL (KB)	Розмір тіла відповіді REST (KB)
1	2020-11-24, 21:28:56 +02:00	190	350	4.18	6.07
2	2020-11-24, 21:29:54 +02:00	201	370	4.18	6.07
3	2020-11-24, 21:32:15 +02:00	221	350	4.18	6.07
4	2020-11-24, 21:35:57 +02:00	191	370	4.18	6.07
5	2020-11-24, 21:36:57 +02:00	181	375	4.18	6.07
6	2020-11-24, 21:38:31 +02:00	221	345	4.18	6.07
7	2020-11-24, 21:39:09 +02:00	211	350	4.18	6.07
8	2020-11-24, 21:40:03 +02:00	215	370	4.18	6.07
9	2020-11-24, 21:40:59 +02:00	210	320	4.18	6.07
10	2020-11-24, 21:41:54 +02:00	212	360	4.18	6.07
Середнє значення	-	205.3	356	4.18	6.07

За допомогою отриманих результатів (див. табл. 3.2) можна відразу визначити тенденцію. GraphQL пропонує швидший час виконання для кожної послідовності запитів. Середня різниця часу виконання між двома стандартами становить приблизно 150.7 мс, що робить GraphQL на 42% швидшим, ніж REST у першому тесті. Цей результат очікувався на теоретичному рівні. Оскільки REST потребує виконати в два рази більше запитів, ніж GraphQL, можна очікувати приблизно на 50% більше часу виконання REST API.

Розмір відповіді залишався стабільним для кожного запиту, оскільки обсяг запитуваної інформації був однаковим протягом усього процесу тестування. Розмір відповіді від GraphQL сервера трохи менше, ніж у REST – 4,18 КБ даних. Дельта, що спостерігається, становить 1,89 КБ, що робить розмір тіла відповіді GraphQL на 31% менше, у порівнянні з його опонентом.

3.2.2. Тест №2

У другому тесті пропорційна зміна показників визначається за рахунок збільшення кількості запитів REST. Для цього конфігурується новий варіант використання, який потребує більше запитів REST, тоді як GraphQL дотримуватиметься одного запиту, який повинен запросити всю необхідну інформацію для повноцінної роботи.

Той самий вигаданий стартап з першого тесту хоче протестувати ефективність використання GraphQL замість REST на іншій частині свого інтерфейсу, але на відміну від попереднього інтерфейсу, тут потрібно отримати деталі про десять підписників, яким сподобався пост окремого користувача. Основна відмінність від першого тесту полягає в тому, що запити можуть виконуватися паралельно. На рис. 3.2 наведена описана схема взаємодії клієнта з REST сервером.

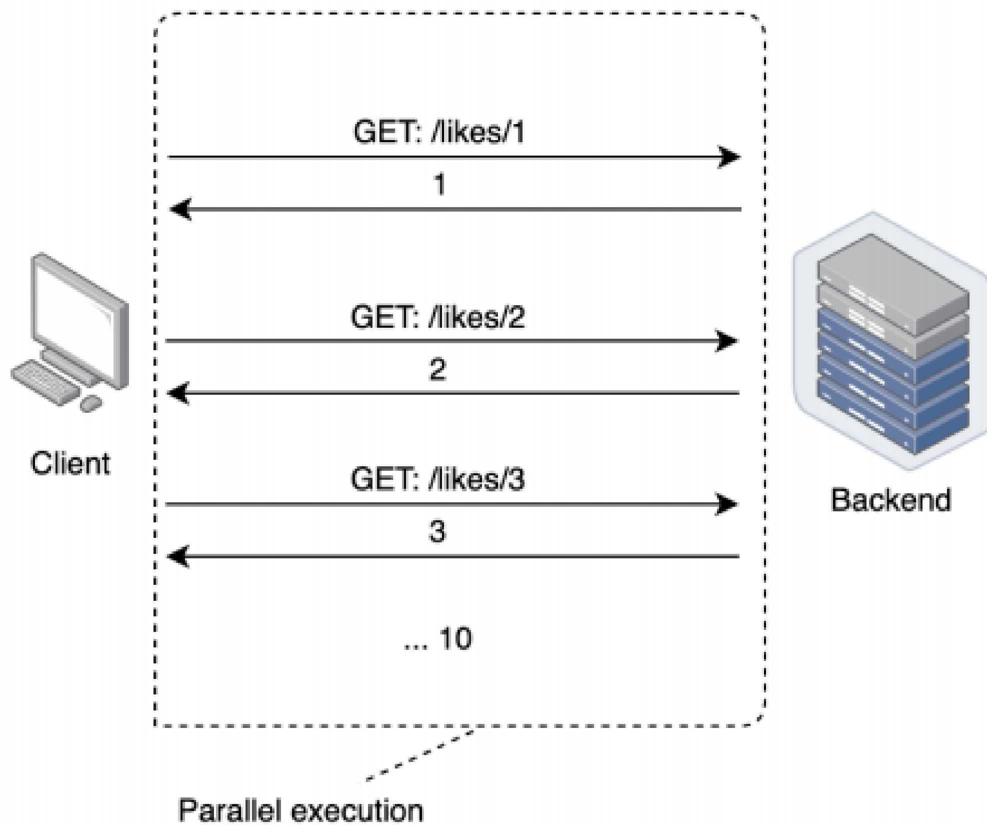

Рис. 3.2. Схема взаємодії клієнта з REST сервером

Така послідовність запитів розглядається як проблемна для інтерфейсу, оскільки кількість вподобань визначатиме кількість запитів. Велика кількість вподобань може спричинити серйозні проблеми з продуктивністю програми. З цієї точки зору GraphQL може допомогти процесу, зменшивши кількість комунікацій до одного.

Таблиця 3.3

Технічні характеристики до тесту №2

Назва характеристики	Значення
Локація сервера	Берлін, Германія
Локація клієнта	Дніпро, Україна
Швидкість інтернет-з'єднання між клієнтом та сервером	60 Мб/с

Продовження таблиці 3.3

Назва характеристики	Значення
REST запити	GET: /likes/1 Повторень: 1 GET: /likes/2 Повторень: 1 …. GET: /likes/10 Повторень: 1

GraphQL запит	{
	likes {
	user {
	username
	email
	}
	}
	}
	
	Повторень: 1

Порядок виконання запитів	Паралельний

Проаналізувавши отримані в результаті тестування дані, які розташовані у табл. 3.4 можна сказати, що у цій вибірці буде звернено увагу на четвертий результат. Цей результат показує дельту лише 4 мс порівняно з іншими, які мають дельту в середньому 82.2 мс. Цей особливий результат ми можемо пояснити аномалією, спричиненою падінням швидкості інтернету або підвищенням клієнта, яка уповільнила запит GraphQL. На цей результат могло впливати багато факторів, тому важко точно пояснити джерело цього конкретного числа.

Знімки екрану з результатами тесту №2 наведено у Додатку А (рис. А.11 - А.20).

Таблиця 3.4

Результати теста №2

Номер виміру	Дата та час	Час виконання GraphQL (мс)	Час виконання REST (мс)	Розмір тіла відповіді GraphQL (KB)	Розмір тіла відповіді REST (KB)
1	2020-11-24, 23:44:03 +02:00	147	251	1.5	5.3
2	2020-11-24, 23:44:03 +02:00	140	270	1.5	5.3
3	2020-11-24, 23:44:03 +02:00	150	240	1.5	5.3
4	2020-11-24, 23:44:03 +02:00	246	250	1.5	5.3
5	2020-11-24, 23:44:03 +02:00	140	230	1.5	5.3
6	2020-11-24, 23:44:03 +02:00	160	255	1.5	5.3
7	2020-11-24, 23:44:03 +02:00	170	210	1.5	5.3
8	2020-11-24, 23:44:03 +02:00	130	266	1.5	5.3
9	2020-11-24, 23:44:03 +02:00	155	223	1.5	5.3
10	2020-11-24, 23:44:03 +02:00	165	230	1.5	5.3
Середнє значення	-	160.3	242.5	1.5	5.3

Незважаючи на отриману аномалію, ці результати ще раз показують, що GraphQL був швидшим і легшим, ніж REST. Пропорційно GraphQL цього разу був на 34% швидшим, що на 8% менше, ніж у тесті №1. Таке невелике зниження

швидкості можна пояснити типом виконання запитів. При послідовному виконанні цей тест міг би зайняти приблизно три рази в довше для запитів REST. Замість цього паралельне виконання запитів дозволило REST бути більш конкурентоспроможним з GraphQL.

3.3. Висновки до третього розділу

Проаналізувавши отримані у результаті обох тестувань дані можна помітити, що результати другого теста значно відрізняються від першого, оскільки розмір тіла відповіді від GraphQL був значно легшим. Причиною є кількість непотрібних даних, що надходять із великою кількістю повідомлень, здійснених REST. Завдяки цьому фактору GraphQL виділяється тим, що розмір тіла відповіді на 72% менше за REST.

В обох випадках отримані результати, які вказують на значну перевагу GraphQL, оскільки він був на 34% та 42% швидшим і розмір тіла відповіді в середньому був менше на 31% та 72%.

РОЗДІЛ 4
ЕКОНОМІЧНИЙ РОЗДІЛ

При розробці програмного забезпечення важливими етапами є визначення трудомісткості розробки ПЗ, розрахунок витрат на створення програмного продукту і аналіз ринку збуту розробленого програмного забезпечення.

4.1. Визначення трудомісткості проведення дослідження та розробки необхідного для його проведення програмного забезпечення

Задані дані:

1. Передбачуване число операторів – 950.
2. Коефіцієнт складності програми – 1,6.
3. Коефіцієнт корекції програми в ході її розробки – 0,3.
4. Годинна заробітна плата програміста, грн/год – 70.
5. Коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,1.
6. Коефіцієнт кваліфікації програміста – 1,5.
7. Вартість машино-години ЕОМ, грн/год – 15.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_и + t_a + t_п + t_{отл} + t_д \text{ ,людино} - \text{годин,} \quad (3.1)$$

де t_o – витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_и$ – витрати праці на дослідження алгоритму рішення задачі;

t_a – витрати праці на розробку блок-схеми алгоритму;

$t_п$ – витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ – витрати праці на налагодження програми на ЕОМ;

$t_д$ – витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q * C * (1 + p), \tag{3.2}$$

де q – передбачуване число операторів;

C – коефіцієнт складності програми;

p – коефіцієнт корекції програми в ході її розробки.

$$Q = 950 * 1{,}6 * (1 + 0{,}3) = 1976, людино - годин \tag{3.3}$$

Витрати праці на вивчення опису задачі $t_и$ визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q * B}{(75 \ldots 85) * k}, \qquad людино - годин, \tag{3.4}$$

де B – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k – коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

$$t_u = \frac{1976 * 1{,}1}{75 * 1{,}5} = \frac{2390{,}96}{112{,}5} = 19{,}32, людино - годин \tag{3.5}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20 \ldots 25) * k}, \qquad людино - годин, \tag{3.6}$$

$$t_a = \frac{1976}{22 * 1,5} = 59,87, людино - годин \tag{3.7}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20 \ldots 25) * k}, людино - годин \tag{3.8}$$

$$t_n = \frac{1976}{20 * 1,5} = 65,86, людино - годин \tag{3.9}$$

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4 \ldots 5) * k}, людино - годин, \tag{3.10}$$

$$t_{отл} = \frac{1976}{4 * 1,5} = 329,3, людино - годин \tag{3.11}$$

– за умови комплексного налагодження завдання:

$$t_{отл}^{k} = 1,5 * t_{отл} \, , людино - годин. \tag{3.12}$$

$$t_{отл}^{k} = 1,5 * 329,3 = 493,95, людино - годин \tag{3.13}$$

Витрати праці на підготовку документації:

$$t_{д} = t_{др} + t_{до}, \qquad людино - годин, \tag{3.14}$$

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису.

$$t_{др} = \frac{Q}{15..20 * k}, людино - годин. \tag{3.15}$$

$$t_{др} = \frac{1976}{15 * 1,5} = 87,82, людино - годин \tag{3.16}$$

$t_{до}$ – трудомісткість редагування, печатки й оформлення документації

$$t_{до} = 0,75 * t_{др}, людино - годин \tag{3.17}$$

$$t_{до} = 0,75 * 87,82 = 65,86, людино - годин \tag{3.18}$$

$$t_{д} = 87,82 + 65,86 = 153,68, людино - годин \tag{3.19}$$

Тепер розрахуємо трудомісткість ПЗ:

$$t = 195,05 + 164,6 + 329,3 + 153,68 = 842,63, людино - годин. \tag{3.20}$$

4.2. Витрати на створення програмного забезпечення для проведення дослідження

Витрати на створення ПЗ Кпо включають витрати на заробітну плату виконавця програми Зз/п і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$К_{по} = З_{зп} + З_{мв}, \textbf{грн.} \quad (3.21)$$

Заробітна плата виконавців визначається за формулою:

$$З_{зп} = t * С_{пр}, грн, \quad (3.22)$$

де: t – загальна трудомісткість, людино-годин;

$С_{пр}$ – середня годинна заробітна плата програміста, грн/година

$$З_{зп} = 842{,}63 * 70 = 58984{,}1, грн. \quad (3.23)$$

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{мв} = t_{отл} * С_{мч}, грн, \quad (3.24)$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год.

$С_{мч}$ – вартість машино-години ЕОМ, грн/год.

$$З_{мв} = 329{,}3 * 15 = 4939{,}5, грн. \quad (3.25)$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення АСУП.

$$К_{по} = 58984{,}1 + 4939{,}5 = 63923{,}6, грн. \quad (3.26)$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k * F_p}, міс, \quad (3.27)$$

де B_k – число виконавців;

F_p – місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

$B_k = 1$

$$T = \frac{842,63}{1 * 176} = 4,7,\ \ міс \tag{3.28}$$

Таким чином, трудомісткість розробки програмного забезпечення становить 4.8 міс.

4.3. Маркетингові дослідження ринку використання результатів дослідження

У сучасному світі швидкість завантаження веб-сайтів, зображень, відео та даних є абсолютною необхідністю для успіху маркетингу.

Швидкість веб-сайту визначається як швидкість перегляду та взаємодії користувачів із вмістом веб-сайту.

Середньостатистичним веб-сайтам електронної комерції може знадобитися близько 7 секунд для повного завантаження, що набагато перевищує ідеальний час завантаження сторінки сайту в 3 секунди (або менше). Повільне завантаження сайту негативно впливає на глибину сеансу користувача, незалежно від того, наскільки мала затримка.

Більше половини людей (53%) опитаних Google заявляють, що покинуть веб-сайт, якщо завантаження триває більше трьох секунд. І все ж, щоб цільова сторінка веб-сайту повністю завантажилася на мобільний телефон, потрібно в середньому 22 секунди.

Зрештою, повільні веб-сайти матимуть низький коефіцієнт конверсії, високий коефіцієнт відмов і низьку кількість переглядів сторінок за відвідування.

Коефіцієнт конверсії визначається кількістю відвідувачів веб-сайту та кількістю тих відвідувачів, що виконують визначену бажану дію, тобто

здійснюють покупку, взаємодіють з контентом, залишають контактний номер, щоб запитати деталі про послуги, тощо.

Показник відмов – це відсоток відвідувачів веб-сайту, які покинули сайт у результаті перегляду лише однієї сторінки.

Перегляд сторінки – це перегляд однієї сторінки на веб-сайті без акценту на унікальність. Якщо користувач перезавантажує сторінку, це зараховується як додатковий перегляд. Якщо користувач переходить на іншу сторінку сайту, а потім повертається на початкову сторінку – це теж зараховується як перегляд сторінки.

Сайт із повільним завантаженням зазвичай викликає розчарування людей, вони закривають вікно браузера або скасовують завантаження сторінки і починають пошук аналогу веб-сайту, але цього разу їм потрібен сайт, що не змушує їх чекати.

Для бізнесу з довгою послідовністю переходів на веб-сайті це ще більш актуально. Завжди виникає випадок, коли послідовність переходів має кілька етапів і сторінок, але веб-сайт досить повільний і користувачі намагаються пройти через більш складну послідовність переходів, це випадання посилюється.

Дані досліджень корпорацій показують як з точки зору користувацького досвіду так і з точки зору фінансового впливу, що є очевидні і цінні переваги в тому, щоб зробити сайт ще швидшим:

– Walmart і Amazon спостерігали збільшення прибутку на 1% на кожні 100 мілісекунд збільшення швидкості веб-сторінки.

– Yahoo побачили збільшення трафіку на 9% на кожні 400 мілісекунд збільшення швидкості веб-сторінки.

– Google втрачає 20% свого трафіку за кожні додаткові 100 мілісекунд швидкості завантаження сторінки.

Одним з варіантів суттєвого підвищення швидкості завантаження веб-сторінки є оптимізація передачі та отримання даних за допомогою вибору найбільш швидко та оптимізованої архітектури API.

Перевага використання результатів досліджень, здобутих у ході виконання роботи, полягає у тому, що правильний вибір архітектури API може підвищити швидкість завантаження даних в середньому на 38% та об'єм передаваємих даних може бути зменшений в середньому на 51.5%. Згідно цих даних можна сказати, що впровадження результатів дослідження може підвищити швидкість роботи веб-сайту майже у 2 рази і це може значно підвищити зацікавленість потенційного клієнта саме цим сайтом і не змусить шукати більш приємні у використанні аналоги.

4.4. Оцінка економічної ефективності впровадження програмного забезпечення

У результаті виконання магістерської роботи досліджується ефективність впровадження сучасної технології у порівнянні з найбільш використовуваною та не розробляється програмне забезпечення, яке можна впровадити. Через відсутність програмного забезпечення, яке можна впровадити, неможливо розрахувати економічний ефект, в якому обсязі необхідні інвестиції, який термін окупності і очікуємий прибуток.

Тому розглядається тільки соціальний ефект.

За допомогою впровадження у існуюче програмне забезпечення отриманих у ході виконання кваліфікаційної роботи результатів може підвищитися швидкість роботи програмного забезпечення, (за умови, якщо технології, які використовуються, не такі ефективні), що надалі може призвести до:

- підвищення зацікавленості сайтом користувачем;
- зниження відсотка користувачів, які покинули сайт при перегляді першої сторінки через повільну роботу сайта;
- підвищення коефіцієнта конверсії;
- просування у списку видачі пошукових систем, що призведе до подальшої безкоштовної реклами, якщо включити витрати на інтеграцію з існуючим продуктом.

ВИСНОВКИ

У результаті розгляду переваг та недоліків як архітектурного стилю GraphQL так і REST можна зробити висновки, що GraphQL, безумовно, сучасний архітектурний стиль, який має багато переваг перед архітектурним стилем REST, але через те, що для роботи з ним потрібно використовувати спеціальні клієнти як для сервера, так і для клієнтського додатка, це робить його використання дещо складним, що робить його не завжди найкращим вибором, якщо розробляється невеликий інтерфейс прикладного програмування.

У ході виконання кваліфікаційної роботи потрібно відповісти на головне питання: чи є GraphQL швидшим та краще оптимізованим, ніж REST? На основі проведених тестів за допомогою сценаріїв з послідовними та паралельними запитами можна підтвердити, що відповідь позитивна для розглянутих ситуацій. В обох випадках отримані результати, які вказують на значну перевагу GraphQL, оскільки він був на 34% та 42% швидшим і розмір тіла відповіді в середньому був менше на 31% та 72%. Треба зауважити, що ці висновки справедливі лише в рамках проведених у результаті дослідження тестів.

Підводячи підсумок, не буде доречним сприймати отримані результати як остаточне твердження про ці дві технології. Існує безліч випадків, які можна перевірити за допомогою розробленого середовища для тестування і, звісно, деякі з них можуть дати зовсім інші результати.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Biehl M., API Architecture: The Big Picture for Building APIs / M. Biehl – CreateSpace, 2015. – 190 P.

2. Remote procedure call – Wikipedia / URL: https://en.wikipedia.org/wiki/Remote_procedure_call . дата звернення: 22.11.2020

3. St. Laurent S. Programming Web services with XML-RPC / S. St. Laurent – O'Reilly Media, 2001. – 230 P.

4. Simple Object Access Protocol – Wikipedia / URL: https://en.wikipedia.org/wiki/SOAP#:~:text=SOAP%20was%20designed%20as%20an,to%20IETF%2013%20September%201999. дата звернення: 23.11.2020

5. What is SOAP API? | AlexSoft / URL: https://www.altexsoft.com/blog/engineering/what-is-soap-formats-protocols-message-structure-and-how-soap-is-different-from-rest/ . дата звернення: 25.11.2020

6. XML — Вікіпедія / URL: https://ru.wikipedia.org/wiki/XML. дата звернення: 25.11.2020

7. Sanjay P. Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS / P. Sanjay – Apress, 2017. – 143 P.

8. Gourley D., Totty B., Sayer M. HTTP: The Definitive Guide / D. Gourley, B. Totty, M. Sayer – O'Reilly Media, 2002. – 656 P.

9. Masse M. REST API Design Rulebook / M. Masse – O'Reilly Media, 2011. – 116 P.

10. Porcello E., Banks A. Learning GraphQL - Declarative Data Fetching For Modern Web Apps / E. Porcello, A. Banks – O'Reilly Media, 2018. – 198 P.

11. Queries and Mutations / URL: https://graphql.org/learn/queries. дата звернення: 25.11.2020

12. Flanagan D. JavaScript: The Definitive Guide / D. Flanagan – O'Reilly Media, 2011. – 1096 P.

13. Simpson K. You Don't Know JS: Async & Performance / K. Simpson – O'Reilly Media, 2015. – 296 P.

14. Lindley C. DOM Enlightenment: Exploring JavaScript and the Modern DOM / C. Lindley – O'Reilly Media, 2013. – 180 P.

15. Cantelon M. Nodejs in Action / M. Cantelon – Manning Publications, 2013. – 416 P.

16. Express - фреймворк веб-приложений Node.js / URL: https://expressjs.com/ru/. дата звернення: 25.11.2020

17. Chinnathambi K. Learning React: A Hands-On Guide to Building Web Applications Using React and Redux / K. Chinnathambi – Manning Publications, 2018. – 304 P.

18. Модель-вид-контролер — Вікіпедія / URL: https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C-%D0%B2%D0%B8%D0%B4-%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%B5%D1%80. дата звернення: 25.11.2020

19. React: The Virtual DOM / URL: https://www.codecademy.com/articles/react-virtual-dom. дата звернення: 25.11.2020

20. Introduction to Apollo Client - Client (React) / URL: https://www.apollographql.com/docs/react/. дата звернення: 25.11.2020

21. graphql-request / URL: https://www.npmjs.com/package/graphql-request. дата звернення: 25.11.2020

22. Introduction to Relay / URL: https://relay.dev/docs/en/introduction-to-relay. дата звернення: 25.11.2020

23. Introduction to Apollo Server / URL: https://www.apollographql.com/docs/apollo-server/. дата звернення: 25.11.2020

24. Running an Express GraphQL Server / URL: https://graphql.org/graphql-js/running-an-express-graphql-server/. дата звернення: 25.11.2020

25. Jin B., Sahni S. Designing Web APIs: Building APIs That Developers Love / B. Jin, S. Sahni – O'Reilly Media, 2018. – 232 P.

26. Hunter K. Irresistible APIs: Designing web APIs that developers will love / K. Hunter – Manning Publications, 2016. – 232 P.

27. Uzayr S. Learning WordPress REST API / S. Uzayr – Packt Publishing, 2016. – 216 P.

28. Doglio F. Pro REST API Development with Node.js / F. Doglio – Apress, 2014. – 196 P.

29. A GraphQL Tutorial / URL: https://www.toptal.com/api-development/graphql-vs-rest-tutorial. дата звернення: 25.11.2020

30. Modern and Legacy API Architectures / URL: https://dev.to/mkamranhamid/api-architecture-4p5h. дата звернення: 25.11.2020

31. What is API Architecture? – More than you think. / URL: https://api-university.com/blog/what-is-api-architecture/. дата звернення: 25.11.2020

32. API Architecture Strategy / URL: https://www.slideshare.net/OCTOTechnology/api-architecture-strategy. дата звернення: 25.11.2020

33. API Management & Microservices / URL: https://abacuscambridge.com/solutions/api-management-and-microservices. дата звернення: 25.11.2020

34. Проектирование веб-API / А. Лоре. – Москва: Вид-во “ДМК Пресс”, 2020. – 440 с.

35. Початок роботи – React / URL: https://uk.reactjs.org/docs/getting-started.html#learn-react. дата звернення: 25.11.2020

36. Wieruch R. The Road to GraphQL : Your journey to master pragmatic GraphQL in JavaScript with React.js and Node.js / R. Wieruch, 2014. – 352 P.

37. React.js Быстрый старт / С. Стефанов. – Пітер: Вид-во “Питер”, 2020. – 304 с.

38. Посібник: знайомство з React / URL: https://uk.reactjs.org/tutorial/tutorial.html. дата звернення: 25.11.2020

39. Рефи та DOM – React / URL: https://uk.reactjs.org/docs/refs-and-the-dom.html. дата звернення: 25.11.2020

40. Orchard L., Pehlivanian A., Koon S. Professional JavaScript Frameworks: Prototype,YUI, ExtJS, Dojo and MooTools / L. Orchard, A. Pehlivanian, S. Koon – Wrox, 2009. – 888 P.

41. Про проект | Node.js / URL: https://nodejs.org/uk/about/. дата звернення: 25.11.2020

42. Set up Apollo Client - Apollo Basics - Apollo GraphQL Docs / URL: https://www.apollographql.com/docs/tutorial/client/. дата звернення: 25.11.2020

43. Cecco R. Supercharged JavaScript Graphics / R. Cecco – O'Reilly Media, 2011. – 280 P.

44. Zakas N. Professional: JavaScript® for Web Developers / N. Zakas – Wrox, 2012. – 960 P.

45. Введение в GraphQL: основные принципы использования / URL: https://medium.com/nuances-of-programming/%D0%B2%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-graphql-%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5-%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D1%8B-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-41b5d50c14c. дата звернення: 25.11.2020

46. Morgan J. Simplifying JavaScript: Writing Modern JavaScript with ES5, ES6, and Beyond / J. Morgan – Pragmatic Bookshelf, 2018. – 284 P.

47. Лекція 2.1. Використання RESTful / URL: http://edu.asu.in.ua/mod/book/tool/print/index.php?id=117. дата звернення: 25.11.2020

48. Erl T. Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services / T. Erl – Pearson, 2004. – 560 P.

49. Подходы к проектированию RESTful API / URL: https://dataart.com.ua/news/podhody-k-proektirovaniyu-restful-api/. дата звернення: 25.11.2020

50. Clements M. Node Cookbook, 2nd Edition: Over 50 recipes to master the art of asynchronous server-side JavaScript / M. Clements – Packt Publishing, 2014. – 378 P.

51. Erl T. SOA: Principles of Service Design / T. Erl – Prentice Hall, 2007. – 573 P.

52. Brajesh D. API Management: An Architect's Guide to Developing and Managing APIs for Your Organization / D. Brajesh – Apress, 2017. – 273 P.

53. Barry D. Web Services, Service-Oriented Architectures, and Cloud Computing: The Savvy Manager's Guide / D. Barry – Morgan Kaufmann, 2013. – 248 P.

54. Введение в web APIs / URL: https://developer.mozilla.org/ru/docs /Learn/JavaScript/Client-side_web_APIs/Introduction. дата звернення: 25.11.2020

55. Fenton S. Pro TypeScript: Application-Scale JavaScript Development / S. Fenton – Apress, 2014. – 248 P.

56. Satheesh M. Web Development with MongoDB and NodeJS / M. Satheesh – Packt Publishing, 2014. – 300 P.

57. Robbins J. Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics / J. Robbins – O'Reilly Media, 2012. – 624 P.

58. Chiarelli A. Beginning React: Simplify your frontend development workflow and enhance the user experience of your applications with React /A. Chiarelli – Packt Publishing, 2018. – 96 P.

59. Mardan A. React Quickly: Painless Web Apps with React, JSX, Redux and GraphQL / A. Mardan – Manning Publications, 2017. – 532 P.

60. Dyl T. Mastering Full Stack React Web Development / T. Dyl – Packt Publishing, 2017. – 532 P.

61. Richardson L. RESTful Web APIs: Services for a Changing World / L. Richardson – O'Reilly Media, 2013. – 406 P.

62. Atencio L. Functional Programming in JavaScript: How to improve your JavaScript programs using functional techniques / L. Atencio – Manning Publications, 2016. – 272 P.

63. Graham S., Davis D., Simeonov S. Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI / S. Graham, D. Davis, S. Simeonov – Sams Publishing, 2004. – 816 P.

64. Haverbeke M. Eloquent Javascript: A Modern Introduction to Programming / M. Haverbeke – No Starch Press, 2018. – 472 P.

65. Zakas N. High Performance JavaScript: Build Faster Web Application Interfaces / N. Zakas – O'Reilly Media, 2010. – 232 P.

66. Vijayakumar T. Practical API Architecture and Development with Azure and AWS / T. Vijayakumar – Apress, 2018. – 188 P.

67. Daigneau R. Service Design Patterns: Fundamental Design Solutions for SOAP WSDL and RESTful Web / R. Daigneau – Addison-Wesley Professional, 2011. – 352 P.

ДОДАТОК А

ЗНІМКИ ЕКРАНУ З РЕЗУЛЬТАТАМИ ТЕСТУВАННЯ

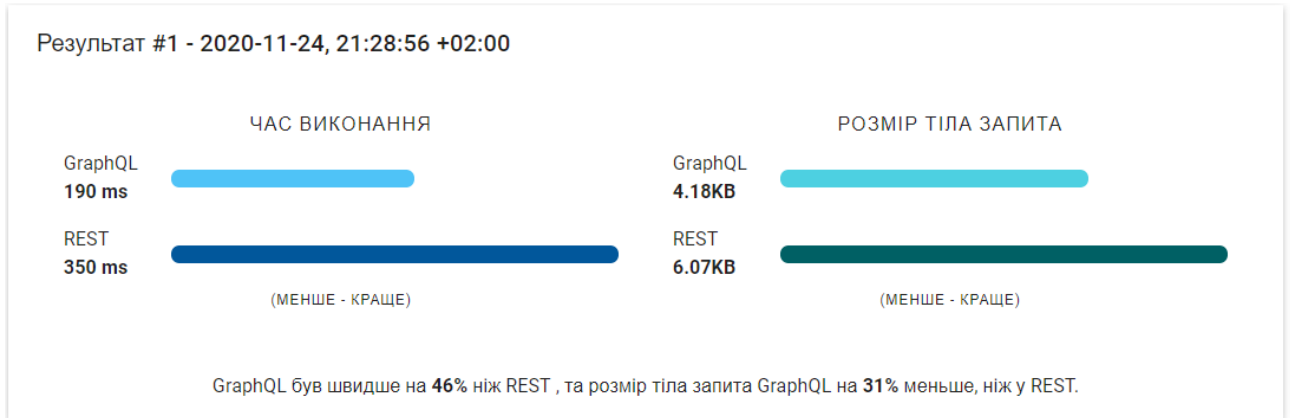

Рис. А.1

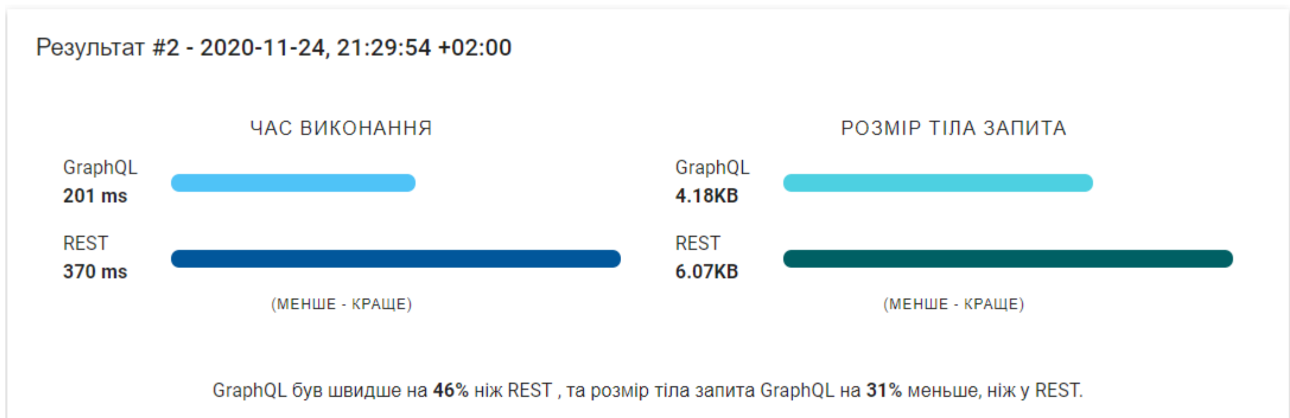

Рис. А.2

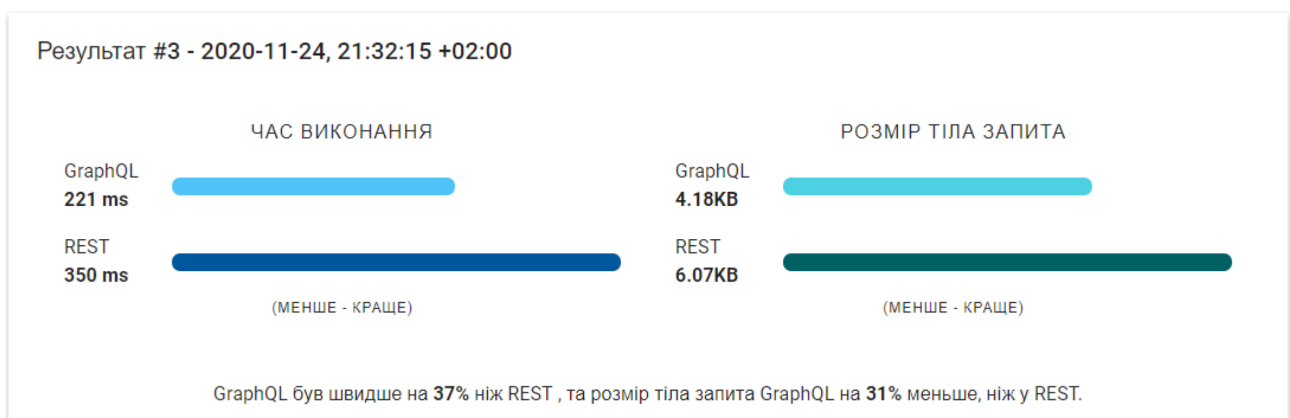

Рис. А.3

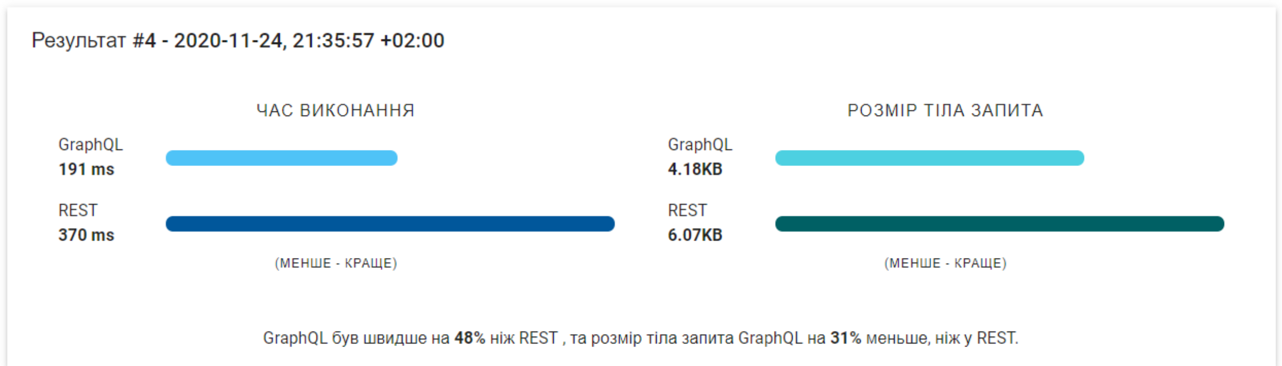

Рис. А.4

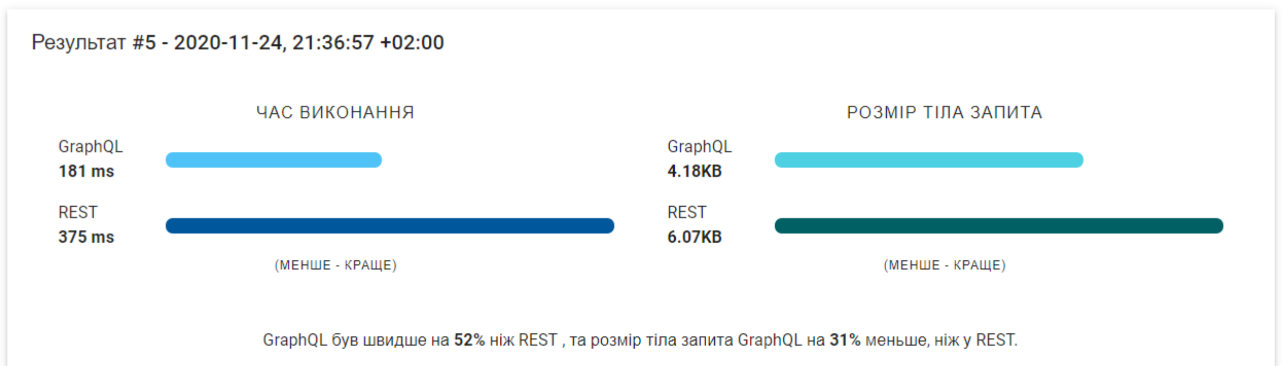

Рис. А.5

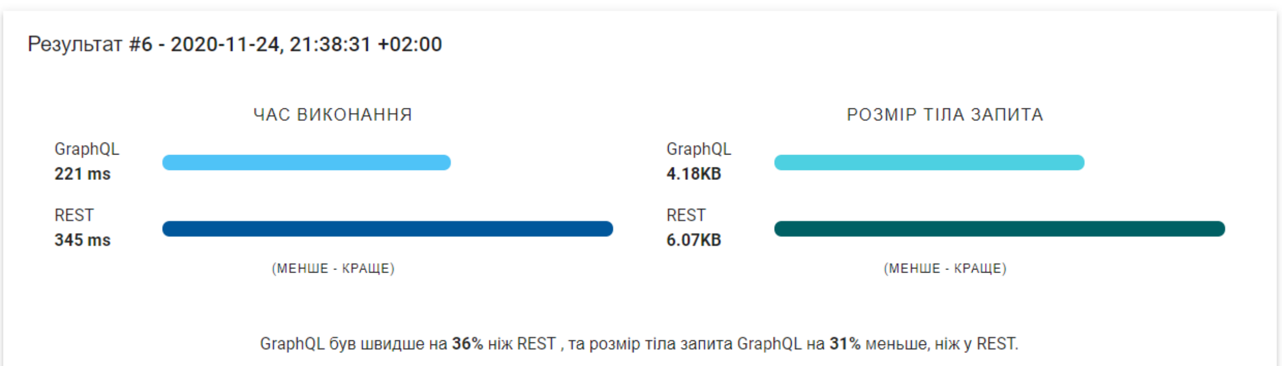

Рис. А.6

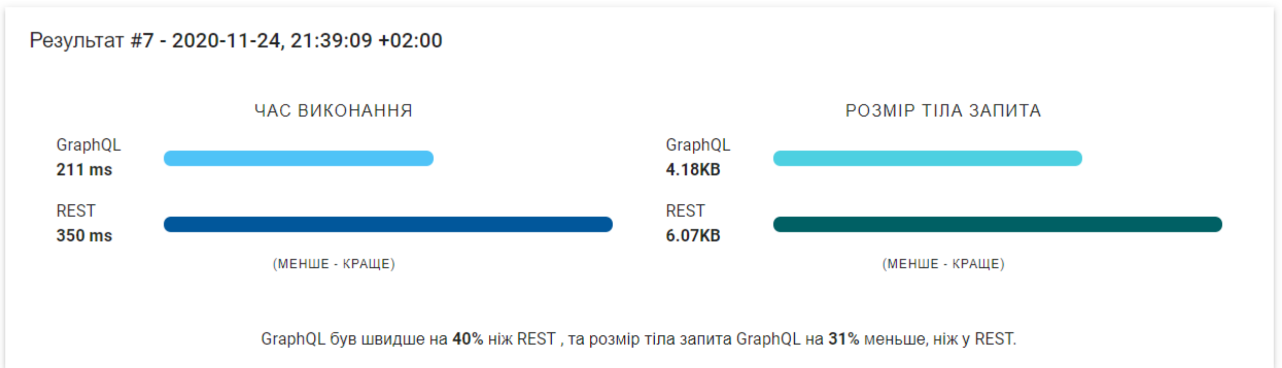

Рис. А.7

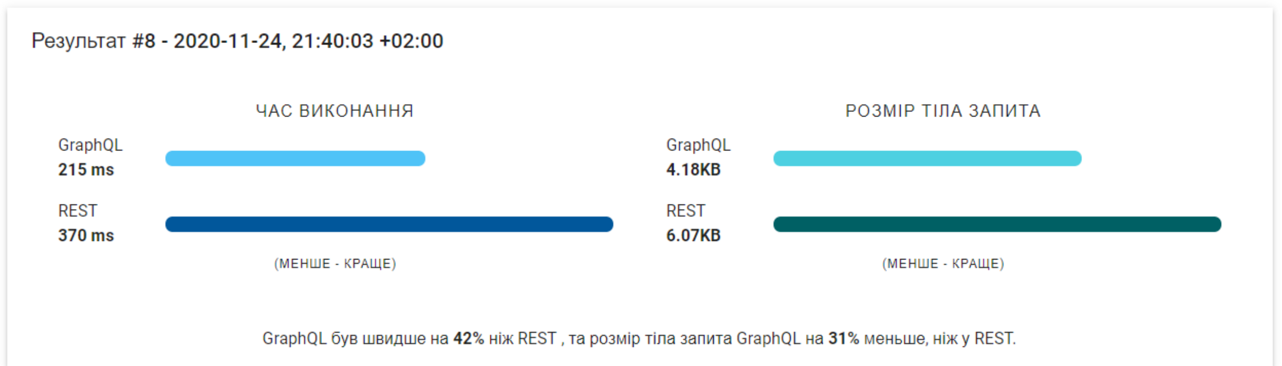

Рис. А.8

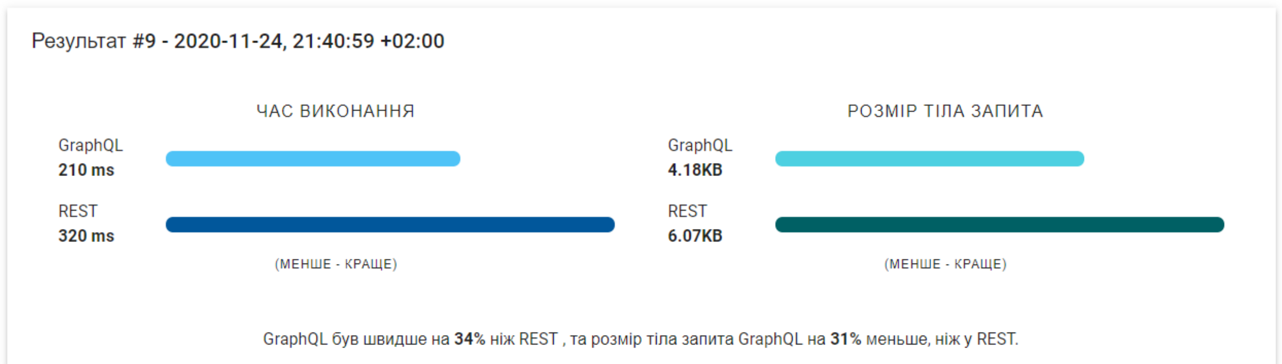

Рис. А.9

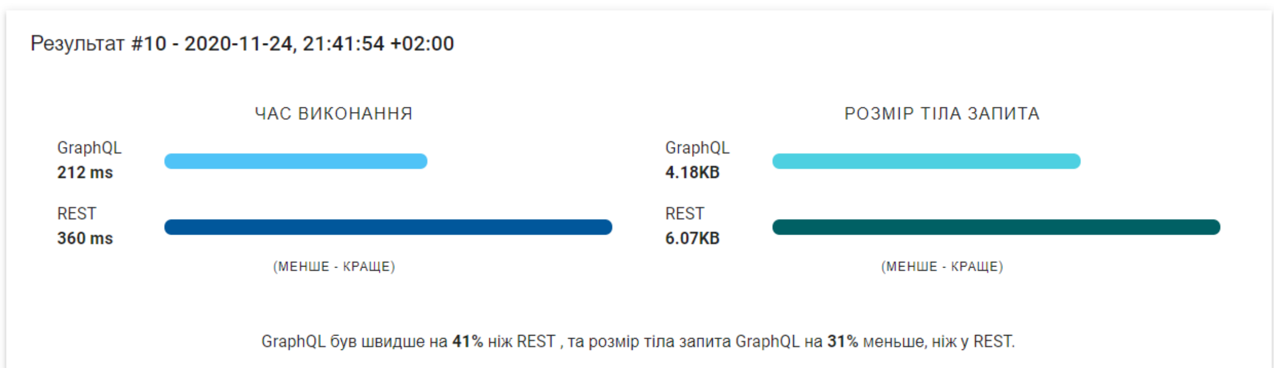

Рис. А.10

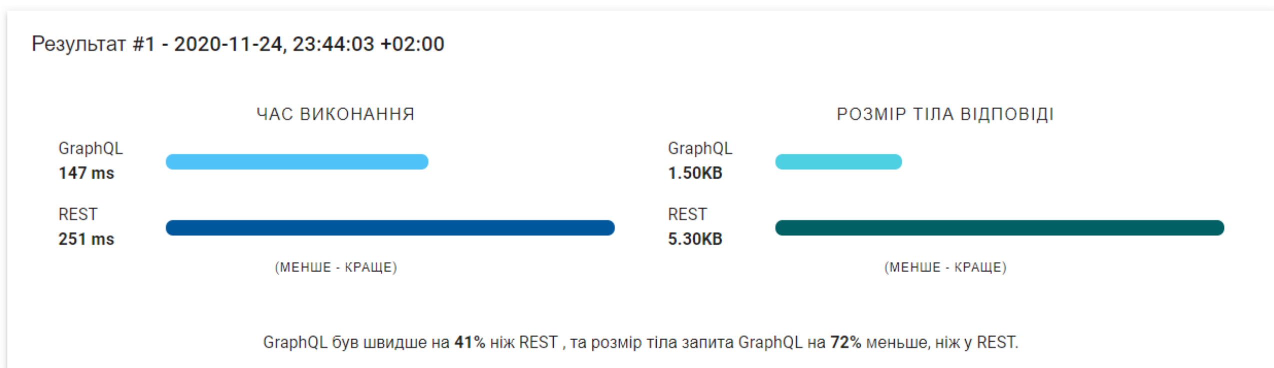

Рис. А.11

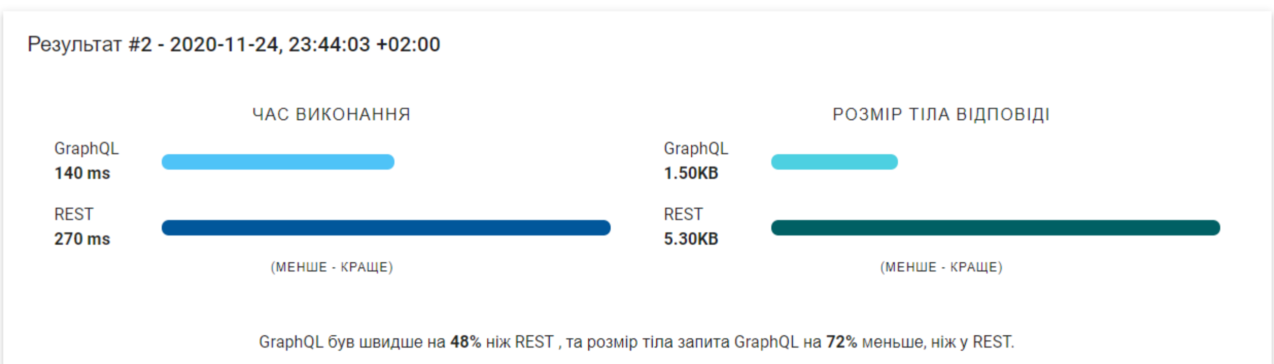

Рис. А.12

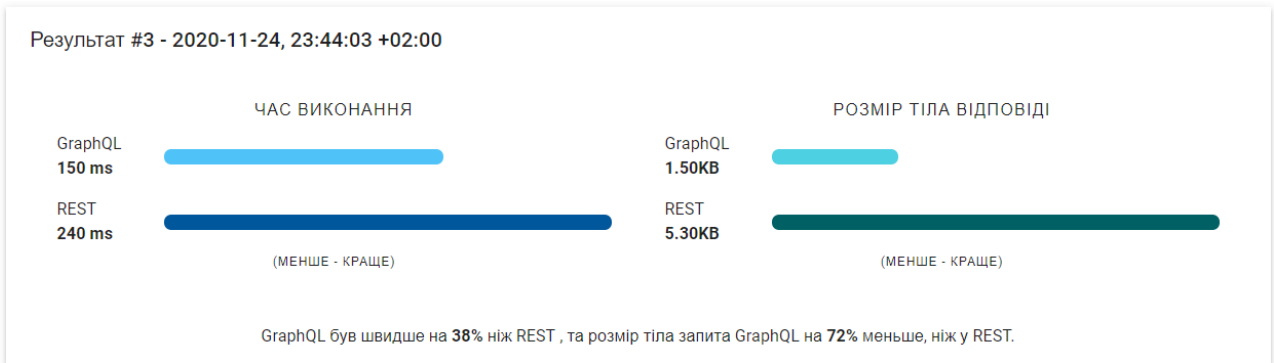

Рис. А.13

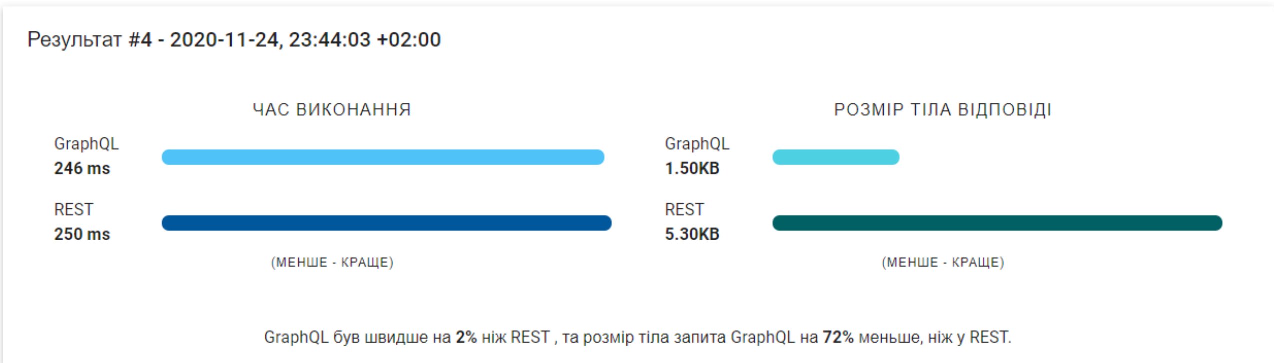

Рис. А.14

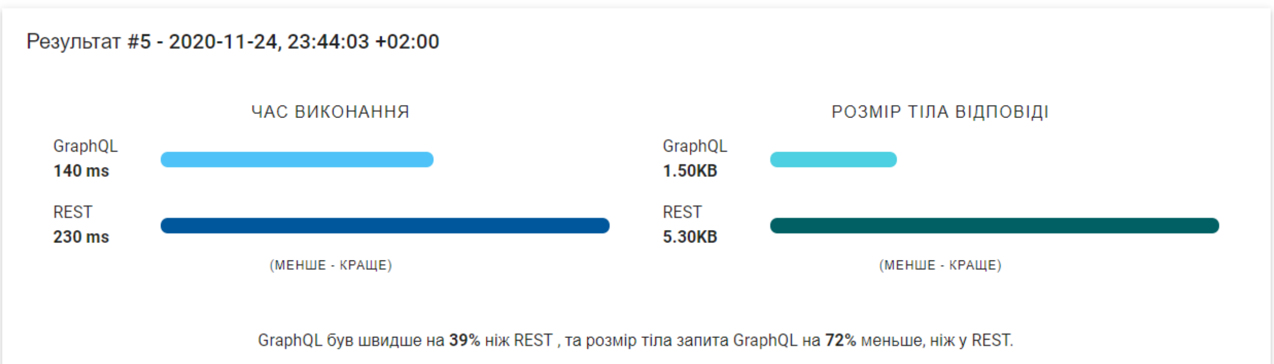

Рис. А.15

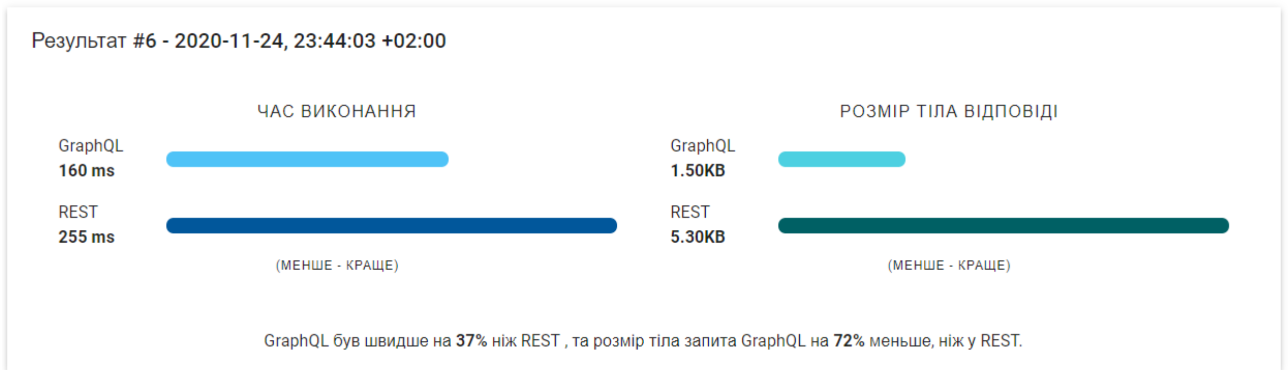

Рис. А.16

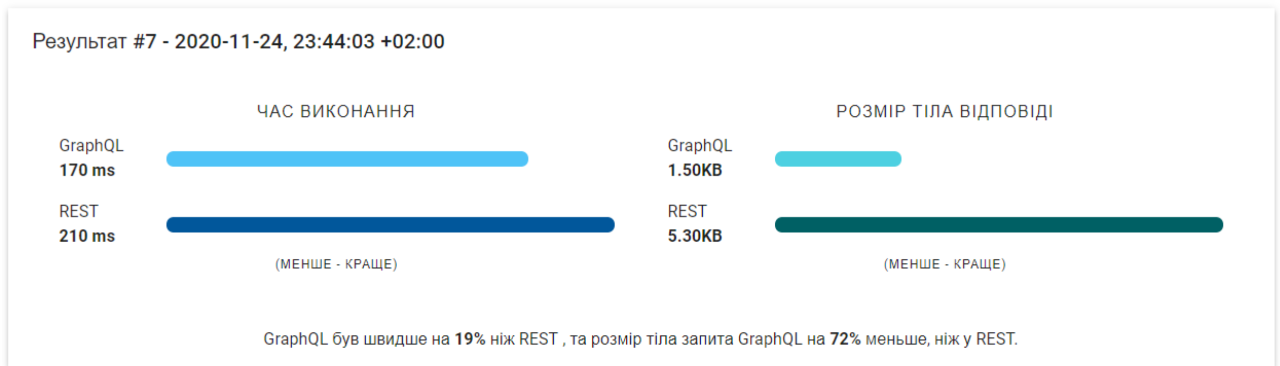

Рис. А.17

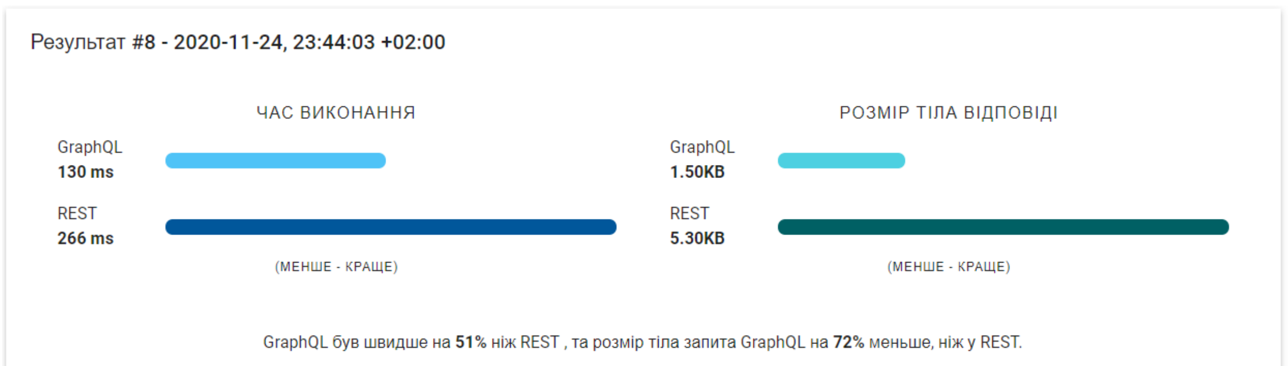

Рис. А.18

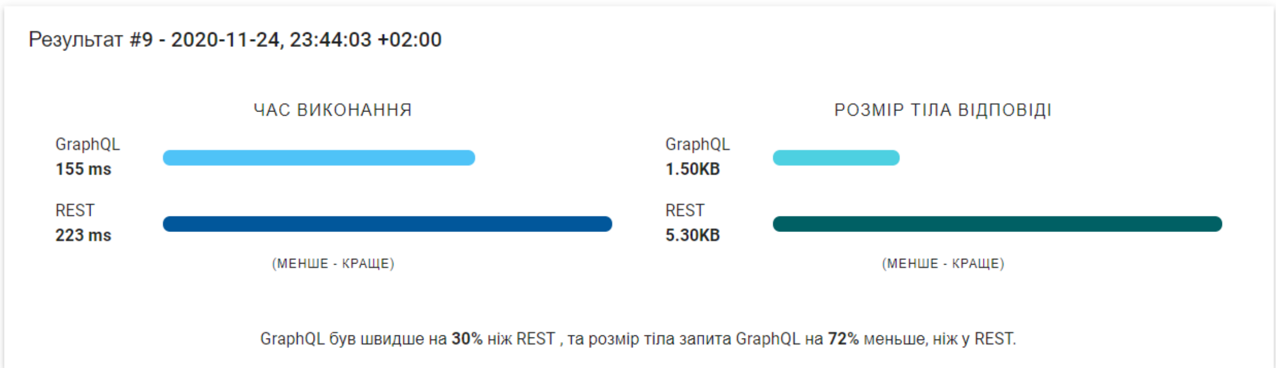

Рис. А.19

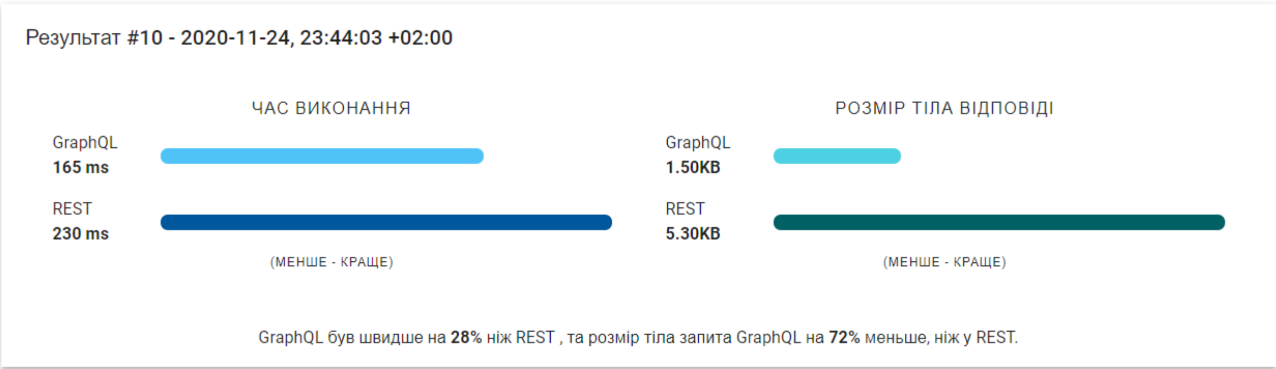

Рис. А.20

ДОДАТОК Б

ЛІСТИНГ ПРОГРАМИ

App.js

```
import React, { useState, useEffect } from "react";

import "typeface-roboto";
import {
  Grid,
  Container,
  Button,
  Paper,
  Typography,
  FormControl,
  FormLabel,
  Radio,
  RadioGroup,
  FormControlLabel,
  CircularProgress,
} from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";

import Header from "./components/Header";
import GraphQL from "./components/GraphQL";
import Rest from "./components/Rest";
import Results from "./components/Results";

import restFetcher from "./lib/restFetcher";
import graphFetcher from "./lib/graphFetcher";

const useStyles = makeStyles((theme) => ({
  container: {
     paddingLeft: theme.spacing(3),
     paddingRight: theme.spacing(3),
  },
}));

function App() {
  const classes = useStyles();
  const [graphQL, setGraphQL] = useState([
     { endpoint: "", query: "", iterations: "" },
  ]);
  const [rest, setRest] = useState([{ endpoint: "", iterations: "" }]);
  const [config, setConfig] = useState({
     executionType: "sequential",
  });
  const [status, setStatuts] = useState("waiting");
  const [graphTime, setGraphTime] = useState(0);
  const [restTime, setRestTime] = useState(0);
  const [graphSize, setGraphSize] = useState(0);
  const [restSize, setRestSize] = useState(0);
  const [results, setResults] = useState([]);

  const start = async (_) => {
     setStatuts("workingRest");
     const rest0 = window.performance.now();
     await restFetcher(rest, config, setRestSize);
     const rest1 = window.performance.now();

     setStatuts("workingGraph");
     const graph0 = window.performance.now();
     await graphFetcher(graphQL, config, setGraphSize);
     const graph1 = window.performance.now();

     setGraphTime(graph1 - graph0);
     setRestTime(rest1 - rest0);
     setStatuts("completed");
  };
```

```
const resetResults = (_) => {
   setGraphSize(0);
   setGraphTime(0);
   setRestSize(0);
   setRestTime(0);
};

useEffect(
   (_) => {
      if (
         graphTime !== 0 &&
         graphSize !== 0 &&
         restTime !== 0 &&
         restSize !== 0
      ) {
         const resultsCopy = [...results];
         resultsCopy.push({
            graphTime: graphTime,
            graphSize: graphSize,
            restTime: restTime,
            restSize: restSize,
            time: new Date(),
         });
         setResults(resultsCopy);
         resetResults();
      }
   },
   [graphSize, graphTime, restSize, restTime, results]
);

return (
   <Container>
      <Grid container justify="space-between" alignItems="center" spacing={6}>
         <Grid item xs={12}>
            <Header />
         </Grid>
         <Grid item xs={6}>
            <Rest rest={rest} setRest={setRest} />
         </Grid>
         <Grid item xs={6}>
            <GraphQL graphQL={graphQL} setGraphQL={setGraphQL} />
         </Grid>
         <Grid item xs={6}>
            <Paper elevation={3} square>
               <Grid
                  container
                  direction="column"
                  justify="center"
                  spacing={4}
                  className={classes.container}
               >
                  <Grid item>
                     <Typography variant="h5">Налаштування</Typography>
                  </Grid>
                  <Grid item xs={12}>
                     <FormControl component="fieldset">
                        <FormLabel component="legend">Порядок виконання</FormLabel>
                        <RadioGroup
                           name="executionType"
                           value={config.executionType}
                           onChange={(e) =>
                              setConfig({ ...config, executionType: e.target.value })
                           }
                           row
                        >
                           <FormControlLabel
                              value="sequential"
                              control={<Radio color="primary" />}
                              label="Послідовний"
                           />
                           <FormControlLabel
                              value="parallel"
                              control={<Radio color="primary" />}
                              label="Паралельний"
                           />
                        </RadioGroup>
                     </FormControl>
                  </Grid>
               </Grid>
            </Paper>
```

```
          </Grid>
          <Grid item>
            <div style={{ position: "relative" }}>
              <Button
                variant="contained"
                size="large"
                onClick={start}
                disabled={
                  status === "workingRest" || status === "workingGraph"
                    ? true
                    : false
                }
              >
                ПОЧАТИ <br />ТЕСТУВАННЯ
              </Button>
              {(status === "workingRest" || status === "workingGraph") && (
                <CircularProgress
                  style={{
                    position: "absolute",
                    left: "50%",
                    top: "50%",
                    marginLeft: -12,
                    marginTop: -12,
                  }}
                  size={24}
                />
              )}
            </div>
          </Grid>
          <Results results={results} />
        </Grid>
      </Container>
  );
}

export default App;
```

Results.js

```
import React from "react";

import moment from "moment";
import numeral from "numeral";

import { Grid, Typography, Paper } from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";

const useStyles = makeStyles((theme) => ({
  container: {
    paddingLeft: theme.spacing(3),
    paddingRight: theme.spacing(3),
  },
  bar: {
    height: 15,
    borderRadius: 15,
  },
  title: {
    fontSize: "1rem",
  },
}));

export default (props) => {
  const { results } = props;
  const classes = useStyles();

  const bar = (size, color) => (
    <div
      className={classes.bar}
      style={{ width: size + "%", backgroundColor: color }}
    />
  );

  const displayResult = (value, unit, label, color, topRange) => (
    <Grid container justify="center" alignItems="center" spacing={4}>
      <Grid item xs={2}>
        <Typography>{label}</Typography>
```

```
         <Typography style={{ fontWeight: "bold" }}>
            {unit === "B"
               ? numeral(value).format("0.00b")
               : numeral(value).format("0") + " " + unit}
         </Typography>
      </Grid>
      <Grid item xs={9}>
         {bar((value / topRange) * 100, color)}
      </Grid>
   </Grid>
);

return (
   <>
      <Grid item xs={12}>
         <Typography variant="h5">Результати</Typography>
      </Grid>
      {results.length === 0 ? (
         <Grid item>
            <Typography variant="body1" color="textSecondary">
               Немає результатів
            </Typography>
         </Grid>
      ) : (
         results
            .sort((a, b) => (a.time - b.time) * -1)
            .map((result, key) => (
               <Grid key={key} item xs={12}>
                  <Paper elevation={3} square>
                     <Grid
                        container
                        direction="column"
                        justify="center"
                        spacing={4}
                        className={classes.container}
                     >
                        <Grid item>
                           <Typography variant="h6">
                              Результат #{key + 1} -{" "}
                              {moment(result.time).format("YYYY-MM-DD, H:mm:ss Z")}
                           </Typography>
                        </Grid>
                        <Grid item xs={12} style={{ maxHeight: 230 }}>
                           <Grid container>
                              <Grid item xs={6}>
                                 <Typography
                                    variant="overline"
                                    component="p"
                                    align="center"
                                    className={classes.title}
                                 >
                                    Час виконання
                                 </Typography>
                                 {displayResult(
                                    result.graphTime,
                                    "ms",
                                    "GraphQL",
                                    "#4fc3f7",
                                    result.graphTime > result.restTime
                                       ? result.graphTime
                                       : result.restTime
                                 )}
                                 {displayResult(
                                    result.restTime,
                                    "ms",
                                    "REST",
                                    "#01579b",
                                    result.graphTime > result.restTime
                                       ? result.graphTime
                                       : result.restTime
                                 )}
                                 <Typography
                                    variant="overline"
                                    component="p"
                                    align="center"
                                 >
                                    (менше - краще)
                                 </Typography>
                              </Grid>
                              <Grid item xs={6}>
```

```
                                <Typography
                                   variant="overline"
                                   component="p"
                                   align="center"
                                   className={classes.title}
                                >
                                   Розмір тіла відповіді
                                </Typography>
                                {displayResult(
                                   result.graphSize,
                                   "B",
                                   "GraphQL",
                                   "#4dd0e1",
                                   result.graphSize > result.restSize
                                      ? result.graphSize
                                      : result.restSize
                                )}
                                {displayResult(
                                   result.restSize,
                                   "B",
                                   "REST",
                                   "#006064",
                                   result.graphSize > result.restSize
                                      ? result.graphSize
                                      : result.restSize
                                )}
                                <Typography
                                   variant="overline"
                                   component="p"
                                   align="center"
                                >
                                   (менше - краще)
                                </Typography>
                             </Grid>
                          </Grid>
                       </Grid>
                       <Grid item>
                          <Typography align="center">
                             {result.graphTime > result.restTime ? "REST" : "GraphQL"}{" "}
                             був швидше на {" "}
                             <b>
                                {result.graphTime > result.restTime
                                   ? numeral(
                                         1 - result.restTime / result.graphTime
                                     ).format("0%")
                                   : numeral(
                                         1 - result.graphTime / result.restTime
                                     ).format("0%")}
                             </b>{" "}
                             ніж{" "}
                             {result.graphTime < result.restTime ? "REST" : "GraphQL"}{" "}
                             , та розмір тіла запита{" "}
                             {result.graphSize > result.restSize ? "REST" : "GraphQL"}
                             {" "}на{" "}
                             <b>
                                {result.graphSize > result.restSize
                                   ? numeral(
                                         1 - result.restSize / result.graphSize
                                     ).format("0%")
                                   : numeral(
                                         1 - result.graphSize / result.restSize
                                     ).format("0%")}
                             </b>{" "}
                             меньше, ніж у{" "}
                             {result.graphSize < result.restSize ? "REST" : "GraphQL"}.
                          </Typography>
                       </Grid>
                    </Grid>
                 </Paper>
              </Grid>
           ))
        )}
     </>
  );
};
```

Graphql.js

```
import React, { useState } from "react";

import { Typography, Paper, Grid, TextField, Button } from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";

const useStyles = makeStyles((theme) => ({
  container: {
     width: "80%",
  },
  field: {
     width: "100%",
  },
  button: {
     width: "100%",
     marginTop: 10,
  },
  paper: {
     justifyContent: "center",
     position: "relative",
     flexWrap: "wrap",
     marginTop: theme.spacing(3),
     padding: theme.spacing(2),
     backgroundColor: "transparent",
     border: "1px solid rgba(0, 0, 0, 0.23)",
  },
  paperTitle: {
     marginTop: -27,
     maxWidth: "fit-content",
     background: "white",
     paddingRight: 5,
     paddingLeft: 5,
  },
}));

const GraphQL = (props) => {
  const classes = useStyles();
  const [endpoint, setEndpoint] = useState("");


  const addQuery = (_) => {
     const queryArray = [...props.graphQL];
     queryArray.push({
        endpoint: props.graphQL[0].endpoint,
        query: "",
        iterations: "",
     });
     props.setGraphQL(queryArray);
  };


  const editQuery = (key) => (e) => {
     const queryArray = [...props.graphQL];
     const queryObject = { ...queryArray[key] };
     queryObject[e.target.name] = e.target.value;
     queryArray[key] = queryObject;
     props.setGraphQL(queryArray);
  };

  const editEndpoint = (e) => {
     setEndpoint(e.target.value);
     const queryArray = [...props.graphQL];
     queryArray.forEach((query, key) => {
        const queryObject = { ...query };
        queryObject.endpoint = e.target.value;
        queryArray[key] = queryObject;
     });
     props.setGraphQL(queryArray);
  };

  return (
     <Paper elevation={3} square>
        <Grid
           container
           direction="column"
           justify="center"
           alignItems="center"
           spacing={4}
        >
           <Grid item>
              <Typography variant="h5" align="center">
```

```
                GraphQL
             </Typography>
          </Grid>
          <Grid item className={classes.container}>
             <TextField
                className={classes.field}
                size="small"
                variant="filled"
                label="Ендпоінт"
                helperText="Eg. https://yourapi.app/graphql"
                value={endpoint}
                onChange={editEndpoint}
             />
          </Grid>
          <Grid item className={classes.container}>
             {props.graphQL.map((query, key) => (
                <Paper key={key} className={classes.paper} elevation={0}>
                   <Typography className={classes.paperTitle}>
                      Запит #{key + 1}
                   </Typography>
                   <TextField
                      className={classes.field}
                      variant="filled"
                      label="Запит"
                      rows={6}
                      size="small"
                      multiline
                      name="query"
                      value={props.graphQL[key].query}
                      onChange={editQuery(key)}
                   />
                   <TextField
                      style={{ marginTop: 10 }}
                      size="small"
                      variant="filled"
                      label="К-сть повторних запитів"
                      helperText="1-5"
                      name="iterations"
                      value={props.graphQL[key].iterations}
                      onChange={editQuery(key)}
                      type="number"
                   />
                </Paper>
             ))}
             <Button
                className={classes.button}
                variant="outlined"
                onClick={addQuery}
             >
                Додати запит
             </Button>
          </Grid>
       </Grid>
    </Paper>
  );
};

export default GraphQL;
```

graphFetcher.js

```
import { request } from "graphql-request";
import sizeof from "object-sizeof";

export default async (queries, config, setSize) => {
  let responseSize = 0;
  if (config.executionType === "parallel") {
     return new Promise(async (resolve, reject) => {
        const queryPromises = [];
        queries.forEach((query) => {
           for (
              let index = 0;
              index < Number.parseInt(query.iterations);
              index++
           ) {
              queryPromises.push(request(query.endpoint, query.query));
           }
        });
```

```
        const responses = await Promise.all(queryPromises);
        responses.forEach((data) => (responseSize += sizeof(data)));
        setSize(responseSize);
        resolve();
      });
  } else {
      return new Promise(async (resolve, reject) => {
        for (let queryIndex = 0; queryIndex < queries.length; queryIndex++) {
          const query = queries[queryIndex];
          for (
            let index = 0;
            index < Number.parseInt(query.iterations);
            index++
          ) {
            const response = await request(query.endpoint, query.query);
            responseSize += sizeof(response);
          }
        }
        resolve();
        setSize(responseSize);
      });
  }
};
```

Rest.js

```
import React, { useState } from "react";

import { Typography, Paper, Grid, TextField, Button } from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";

const useStyles = makeStyles((theme) => ({
  container: {
      width: "80%",
  },
  field: {
      width: "100%",
  },
  button: {
      width: "100%",
      marginTop: 10,
  },
  paper: {
      justifyContent: "center",
      position: "relative",
      flexWrap: "wrap",
      marginTop: theme.spacing(3),
      padding: theme.spacing(2),
      backgroundColor: "transparent",
      border: "1px solid rgba(0, 0, 0, 0.23)",
  },

  paperTitle: {
      marginTop: -27,
      maxWidth: "fit-content",
      background: "white",
      paddingRight: 5,
      paddingLeft: 5,
  },
}));

const GraphQL = (props) => {
  const classes = useStyles();
  const [endpoint, setEndpoint] = useState("");


  const addQuery = (_) => {
      const queryArray = [...props.graphQL];

   queryArray.push({
        endpoint: props.graphQL[0].endpoint,
        query: "",
        iterations: "",
      });

      props.setGraphQL(queryArray);
  };
```

```
const editQuery = (key) => (e) => {
   const queryArray = [...props.graphQL];
   const queryObject = { ...queryArray[key] };

   queryObject[e.target.name] = e.target.value;
   queryArray[key] = queryObject;
   props.setGraphQL(queryArray);
};

const editEndpoint = (e) => {
   setEndpoint(e.target.value);

   const queryArray = [...props.graphQL];

   queryArray.forEach((query, key) => {
      const queryObject = { ...query };
      queryObject.endpoint = e.target.value;
      queryArray[key] = queryObject;
   });

   props.setGraphQL(queryArray);
};

return (
   <Paper elevation={3} square>
      <Grid
         container
         direction="column"
         justify="center"
         alignItems="center"
         spacing={4}
      >
         <Grid item>
            <Typography variant="h5" align="center">
               GraphQL
            </Typography>
         </Grid>
         <Grid item className={classes.container}>
            <TextField
               className={classes.field}
               size="small"
               variant="filled"
               label="Ендпоінт"
               helperText="Eg. https://yourapi.app/graphql"
               value={endpoint}
               onChange={editEndpoint}
            />
         </Grid>
         <Grid item className={classes.container}>
            {props.graphQL.map((query, key) => (
               <Paper key={key} className={classes.paper} elevation={0}>
                  <Typography className={classes.paperTitle}>
                     Запит #{key + 1}
                  </Typography>
                  <TextField
                     className={classes.field}
                     variant="filled"
                     label="Запит"
                     rows={6}
                     size="small"
                     multiline
                     name="query"
                     value={props.graphQL[key].query}
                     onChange={editQuery(key)}
                  />
                  <TextField
                     style={{ marginTop: 10 }}
                     size="small"
                     variant="filled"
                     label="К-сть повторних запитів"
                     helperText="1-5"
                     name="iterations"
                     value={props.graphQL[key].iterations}
                     onChange={editQuery(key)}
                     type="number"
                  />
               </Paper>
            ))}
            <Button
```

```
                  className={classes.button}
                  variant="outlined"
                  onClick={addQuery}
               >
                  Додати запит
               </Button>
            </Grid>
         </Grid>
      </Paper>
   );
};

export default GraphQL;
```

ДОДАТОК В

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу магістра

на тему:

«Дослідження ефективності використання інтерфейсу прикладного програмування GraphQL у порівнянні з REST»

студента групи 121м-19-1 Машевського Андрія Миколайовича

Керівник економічного розділу

доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

ДОДАТОК Г

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Машевський.docx	Пояснювальна записка до магістерської роботи. Документ Word.
	Пояснювальна записка до до магістерської роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Машевський.ppt	Презентація до магістерської роботи