МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Застосування технології GraphQL для створення програмного забезпечення мобільних пристроїв**

Виконав: студент 2 курсу, групи 8.1219-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного Забезпечення
(код і назва освітньої програми)
О.Б. Тішин
(ініціали та прізвище)
Керівник доцент, к.ф.-м.н, В.І. Попівщий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
Лютий П.О.
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кафедра програмного забезпечення автоматизованих систем
Рівень вищої освіти другий (магістерський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри В.Г. Вербицький
“ 01 ” вересня 2020 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Тішину Олексію Борисовичу
(прізвище, ім’я, по батькові)

1.Тема роботи Застосування технології GraphQL для створення програмного забезпечення мобільних пристроїв.

керівник роботи Попівщий В.І., доцент, к.ф.-м.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від “25” травня 2020 року № 600-с

2. Строк подання студентом кваліфікаційної роботи 30.11.2020
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.09.2020

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	02.09-10.09.20	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.09-12.09.20	виконано
3	Аналіз існуючих методів рішення	13.09-14.09.20	виконано
4	Дослідження проблеми передачі даних до мобільних пристроїв	15.09-20.09.20	виконано
5	Узгодження подальших дій з науковим керівником	21.09-26.09.20	виконано
6	Аналіз теоретичних відомостей	27.09-28.09.20	виконано
7	Проектування серверного додатка	29.09-13.10.20	виконано
8	Узгодження інтерфейсу клієнтського додатка з науковим керівником	14.10-16.10.20	виконано
9	Реалізація функціоналу серверного додатка	17.10-21.10.20	виконано
10	Представлення отриманих результатів науковому керівнику і узгодження плану подальшого дослідження	22.10-29.10.20	виконано
11	Реалізація функціоналу клієнтського додатка	30.10-17.11.20	виконано
12	Проведення аналізу можливостей розроблених програмних застосунків	18.11-23.11.20	виконано
13	Оформлення звіту	24.11-29.11.20	виконано

Студент _____ Тішин О.Б.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Попівщий В.І.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 101

Рисунків: 23

Таблиць: 1

Джерел: 34

Тішин О.Б. Застосування технології GraphQL для створення програмного забезпечення мобільних пристроїв.

Кваліфікаційна робота для здобуття ступеня вищої освіти магістра за спеціальністю 121 – Інженерія програмного забезпечення, науковий керівник доц. В.І. Попівщий. Інженерний навчально-науковий інститут ЗНУ. 2020.

Мета кваліфікаційної роботи полягає у вивченні технології GraphQL, проектуванні і розробці клієнт-серверної програмної системи, яка включає API для обробки REST і GraphQL запитів, і дослідженні за допомогою розробленої системи переваги і недоліки REST і GraphQL підходів.

Досліджено методи обміну даними в сучасних розподілених системах, їх проблематику і можливості використання в розроблюваній системі. Для реалізації запланованого функціоналу розроблено веб-сервер на базі Apollo Server, Nest.JS та Express.js та мобільний застосунок на платформі Android.

Виконано аналіз трафіку, часу відгуку, розмірів запитів і відповідей для REST і GraphQL підходів. Зроблено відповідні оцінки роботи технологій GraphQL та REST.

Ключові слова: GraphQL, JavaScript, HTTP Reactive, Serialization, REST, API, SOAP, синтаксис, лексема.

SUMMARY

Pages: 101

Figures: 23

Tables: 1

Sources: 34

Tishyn Oleksii. Application of GraphQL technology to create software for mobile devices.

Qualification work for obtaining a master's degree in specialty 121 - Software Engineering, research supervisor Vasiliy Popivshchiy. Engineering Educational Scientific Institute of Zaporizhia National University. 2020.

The purpose of the qualification work is to study GraphQL technology, design and develop a client-server software system that includes APIs for processing REST and GraphQL queries, and research using system advantages and disadvantages of REST and GraphQL approaches.

Methods of data exchange in modern distributed systems, their problems and possibilities of use in the developed system are investigated. A web server based on Apollo Server, Nest.JS and Express.js and a mobile application on the Android platform have been developed to implement the planned functionality.

The analysis of traffic, response time, query sizes and responses for REST and GraphQL approaches is performed. Appropriate evaluations of GraphQL and REST technologies have been made.

Keywords: GraphQL, JavaScript, HTTP Reactive, Serialization, REST, API, SOAP, syntax, token

ЗМІСТ

ВСТУП

Актуальність теми

На початку 2000-х років розробники багатьох фірм почали переосмислювати спосіб роботи з REST API. Їх непокоїла різниця між даними, які вони хотіли використовувати у своїх програмах, та запитами до сервера, які ці дані вимагали. Розробники не розглядали дані як URL-адреси ресурсів, вторинні ключі чи таблиці об’єднань; вони думали про це з точки зору графу об’єктів та моделей, таких як NSObjects або JSON, які в кінцевому рахунку використовувались в програмах,. Тому почались пошуки кращого вирішення цієї проблеми.

Netflix запропонував рішення, яке передбачало новий концептуальний рівень між типовим клієнтським та серверним рівнями, де на сервері розміщується специфічний для клієнта код.

Ще одна компанія — SoundCloud — зіткнулася з подібними проблемами. Під час переходу від монолітної архітектури до сервіс- орієнтованої, вони почали боротися зі своїм існуючим API. Їхнє рішення було досить цікавим: замість того, щоб включати розширені параметри налаштування до свого основного API, вони вирішили, що кожен випадок використання отримає власний сервер API. Це дозволило б розробникам ефективно оптимізувати кожен варіант використання.

У вересні 2015 року Facebook офіційно оголосив про випуск GraphQL, популярність якого з тих пір стрімко зросла. GraphQL — це специфікація мови запитів API та серверного механізму, здатного виконувати такі запити. В основі підходу GraphQL лежить наступний принцип: краще мати один «розумний» endpoint, який буде здатний працювати зі складними запитами і повертати дані саме в тій формі і в тому обсязі, які необхідні клієнту.

Дана робота присвячена дослідженню специфікації GraphQL, її переваг та недоліків при застосуванні з мобільними застосунками.

Мета і завдання дослідження

Мета роботи — спроектувати і розробити клієнт-серверну програмну систему, яка включає API для обробки REST і GraphQL запитів і дослідити за допомогою розробленої системи переваги і недоліки REST і GraphQL підходів. Для досягнення поставленої мети потрібно виконати ряд завдань:

1. Дослідити технології REST і GraphQL.

2. Спроектувати і розробити тестову програмну систему, яка включає сервер REST і GraphQL і мобільний клієнтський застосунок.

3. Виконати аналіз трафіку, часу відгуку, розмірів запитів і відповідей для REST і GraphQL підходів. Програмна система буде діагностувати відправлені пакети та виконувати аналіз надлишковості або нестачі даних при різних варіантах використання.

Об’єкт дослідження

Web API на основі кінцевих точок, побудовані на основі архітектури REST і GraphQL.

Предмет дослідження

Структуровані дані, що надходять до клієнтських додатків. Час та розмір відповідей при різній вкладеності запитів. Трафік між клієнтським та серверним застосунком.

Методи дослідження

Для розв’язання представлених завдань використовуються такі методи дослідження:

- Аналіз джерел з використання GraphQL та REST API.
- Проведення аналізу та різниці між методами отримання та зберігання інформації.
- Синтез отриманих результатів досліджень.
- Порівняльний аналіз програмних продуктів.

Наукова новизна одержаних результатів

Спроектована і розроблена оригінальна тестова програмна система, яка включає сервер REST і GraphQL і мобільний клієнтський застосунок. Система дозволяє досліджувати трафік, час відгуку, розміри запитів і відповідей для REST і GraphQL, наочно відображаючи переваги та недоліків обох підходів. Це буде сприяти побудові надійних систем API. Спроектовано алгоритм удосконалення системи пагінації.

Практичне значення одержаних результатів

Дані, отримані в результаті дослідження, показують, що GraphQL не є панацеєю для всіх типів вирішуваних у проектуванні API задач. Ретельне проектування API на основі архітектури REST може оптимізувати конкретний випадок використання і дати кращі результати.

Апробація результатів

Результати роботи, викладені у кваліфікаційній роботі магістра, були опубліковані в збірнику наукових праць студентів, аспірантів і молодих вчених «Молода наука-2020» і представлені на XXV науково-технічній конференції студентів, магістрантів, аспірантів, молодих вчених та викладачів [1].

Глосарій

Мобільний застосунок (англ. «Mobile app») — програмне забезпечення, призначене для роботи на смартфонах, планшетах та інших мобільних пристроях. Багато мобільних застосунків встановлені на самому пристрої або можуть бути завантажені на нього з онлайн магазинів мобільних застосунків, таких як App Store, Google Play, Windows Phone Store та інших, безкоштовно або за плату. Спочатку мобільні застосунки використовувалися для швидкої пере-

вірки електронної пошти, але їх високий попит призвів до розширення їх призначень і в інших областях, таких як ігри для мобільних телефонів, GPS, спілкування, перегляд відео та користування Інтернетом.

Фреймворк (англ. «*Framework*») — інфраструктура програмних рішень, що полегшує розробку складних систем. Спрощено дану інфраструктуру можна вважати своєрідною комплексною бібліотекою.

GraphQL — це мова запитів і маніпуляції даними з відкритим кодом для API і середовище виконання для обслуговування запитів з наявних даних.

REST — підхід до архітектури мережевих протоколів, які надають доступ до інформаційних ресурсів. Був описаний і популяризований 2000 року Роєм Філдінгом, одним із творців протоколу HTTP.

HTTP — протокол передачі даних, що використовується в комп'ютерних мережах.

Мутація (англ. *«Mutation»*) — зміна даних на сервері і отримання оновлених даних назад.

Підписка (англ. «*Subscription*») — підтримання з'єднання з сервером в режимі реального часу.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМ ПЕРЕДАЧІ ІНФОРМАЦІЇ ВІД СЕРВІСІВ НА ОСНОВІ КІНЦЕВИХ ТОЧОК

1.1 Загальні відомості про передачу інформації

Все в нашому світі містить інформацію, ми працюємо кожен день з інформацією. Ми створюємо алгоритми стиснення, щоб перекодувати дані з метою зменшення їхнього обсягу, розміру, об’єму.

Стиснення базується на усуненні надлишкової інформації, яка може міститись у вихідних даних, повторенні фрагментів. Існують види стиснення без втрат або зі втратами. Як правило стиснення може застосовуватися на протоколі HTTP. За специфікацією HTTP/1.1 (документ RFC 2616) передбачено три способи стиснення — deflate (RFC 1951), compress (аналог до unix-програми compress) і gzip (RFC 1952, аналог до unix-програми gzip). Найбільш практичним є метод gzip і його використовує більшість сучасних Web-клієнтів, таких як MS Internet Explorer, сімейство Mozilla, Konqueror та інші [1].

Стиснення на боці сервера потребує додаткових ресурсів.

До чого саме стиснення, а до того, що мобільні пристрої змушені отримувати надлишкову інформацію і це потрібно враховувати. GraphQL надає можливість отримувати ту інформацію, яка потрібна клієнту, без надлишковості.

1.2 Приклади задач з проблемами надлишковості

Приклад задач, для вирішення яких використовується GraphQL:

1. Деталі користувача які можна отримати на окремому маршруті.

2. Проблема безлічі запитів. Припустимо, з сервера необхідно отримати список всіх користувачів та інформацію про них. У REST-архітектурі для цього буде потрібно виконати один запит для отримання списку всіх користувачів і, в залежності від виконання API сервера, ще один або безліч запитів для отримання інформації про них [2].

3. Проблема зайвих даних. Припустимо, необхідно відобразити тільки ім'я та прізвище користувача, а в REST-архітектурі сервер повертає абсолютно всю інформацію про нього: прізвище, ім’я, по батькові, дату народження тощо [2].

4. Залежність архітектури від протоколу передачі даних. Архітектура REST прив’язана до HTTP і використання властивостей, характерних тільки для HTTP: методи, коди стану. Написаний додаток з архітектурою REST буде складно перенести на інший канал зв’язку, наприклад, WebSockets або SFTP [2].

5. Незалежність коду клієнта від сервера. Для усунення проблем безлічі запитів і зайвих даних можна створювати нові методи та логіку в програмі. Але для цього потрібно одночасно змінювати і клієнтський, і серверний код. При цьому виникає проблема версійності додатків [2].

1.3 Варіативність API

У цьому підрозділі пояснюється набір функцій або процедур, які дозволяють створювати додатки, що мають доступ до функцій або даних операційної системи, програми чи іншої послуги. Вони відомі як інтерфейси прикладного програмування (API).

В даний час API стали усюдисущими компонентами програмних інфраструктур. Їх можна знайти скрізь; від побутового обладнання, такого як холодильник, до складних технологій, таких як космічні станції. Крім того, вони є частиною мобільних пристроїв, що динамічно розвиваються, і де користувачі використовують пристрої у своєму повсякденному житті. Веб-додатки, бекенд системи та платформи для мобільних додатків зокрема надають API. API можна визначити як набір функцій або процедур, які використовуються комп’ютерними програмами для доступу до послуг операційної системи, бібліотек програмного забезпечення або інших систем. Так само, як користуваль-

ницький інтерфейс, який дозволяє взаємодії та зв’язок між програмним забезпеченням та особою, API полегшує спілкування між двома програмами, щоб функціональні можливості обмінювалися між ними [3].

Сучасні програми мають потребу в доступу до послуг (даних або функціональності) з віддаленої системи. Тут відповідальність API полягає в наданні інтерфейсу для збережених даних або функціональних можливостей, що відповідає потребам програми. У цьому випадку API являє собою договір між даними чи функціональними можливостями, що надаються постачальником послуг та споживачами, які хочуть взаємодіяти з ним. Отже, він визначає, як клієнт може отримувати послуги з віддаленої системи. Наприклад, клієнт використовує бібліотеку (набір функцій та процедур), яку пропонують дистанційні системи, як абстрактний шар поверх HTTP протоколу для доступу та обміну додатковою інформацією [3].

Таким чином, вони обидва використовують інформацію один одного, не порушуючи їх незалежності. Це основа для інтеграції та впровадження різних додатків та компонентів додатків, що дозволяє систематично розробити надійні операції з розподіленими додатками, програмами та мобільними додатками. В API можуть бути зацікавлені сторони, такі як дизайнери API, користувачі API та споживачі отриманого продукту. Окрім цього API використовуються для створення інтеграції додатків з діловими партнерами, постачальниками та замовниками, як показано на Рис. *1*. Як було описано, правильний дизайн API може покращити швидкість розробки, сприяти підвищенню якості та рівня використання програмного забезпечення [3].

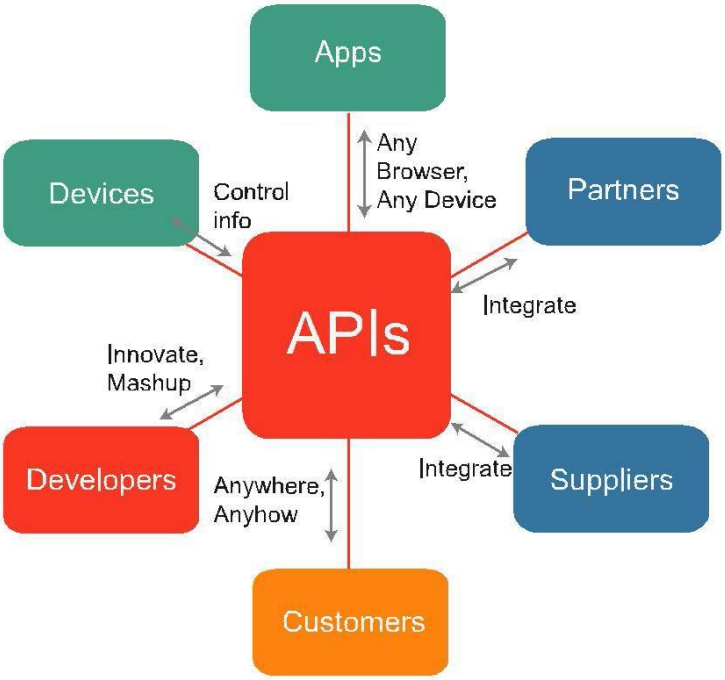

Рис. *1 Можливий стек роботи API*

API можна вважати такими, що мають три взаємодіючих шари; публічний інтерфейс, реальна реалізація функціональних можливостей та пересічний шар, який надає такі утиліти, як обробка помилок, бібліотеки сторонніх організацій та додаткові допоміжні функції. Зазвичай інтерфейси дають визначення функцій та структур даних, в той час як реалізація реалізує ці інтерфейси.

Технічно API можуть бути відкриті та використані у формах бібліотек, які пов'язані з певною мовою програмування або у формі мовно-агностичних веб-служб. Веб-інтерфейси API RESTful або сервіси на основі WSDL / SOAP є загальновідомими формами надання та використання API [4].

Ці API надають інтерфейс для веб-додатків або для додатків, які потребують з'єднання або спілкування один з одним через Інтернет. Кількість загальнодоступних веб-API постійно зростає, особливо із збільшенням інновацій у мобільних пристроях. Ці веб-API можна використовувати для того, щоб робити все, від перевірки трафіку та погоди, до оновлення статусу соціальних медіа чи для здійснення платежів.

Однією з найбільших проблем побудови API є побудова такої, яка працюватиме довго, і спільнота розробників програмного забезпечення завжди розглядає чотири основні особливості, щоб оцінити якість API; він повинен бути корисним і зрозумілим, стійким при удосконаленнях, а також забезпечити хорошу документацію. Для створення веб-API використовуються кілька стилів або протоколів, серед яких найпопулярнішим є REST [4].

1.3.1 REST

У цьому підрозділі йдеться про архітектурний стиль, який зазвичай використовується для розробки веб-API. Цей стиль відомий як *Representational State Transfer (REST).* Термін "REST" був введений у 2000 році в докторській дисертації Роя Філдінга, одного з головних авторів специфікації протоколу передачі гіпертексту (HTTP). REST – це архітектурний стиль для розподілених систем гіпермедіа і визначається на основі набору обмежень.

Відповідно до цього REST наголошує на наборі обмежень, таких як масштабованість взаємодій, загальність інтерфейсів, незалежне розгортання та посередники компонентів для зменшення затримки взаємодії, що забезпечують безпеку та інкапсуляцію застарілих систем [5].

Обмеження REST, що випливають із загальних архітектурних стилів, визначаються для властивостей, які вони викликають в архітектурах-кандидатах. Будь-яку архітектуру, сумісну з цими обмеженнями, можна назвати REST (або RESTful) архітектурою. Він використовує основні технології всесвітньої павутини (WWW), такі як HTTP разом з URI та MIME, щоб сприяти простоті, інтероперабельності на основі стандартів та повсюдній доступності на всіх видах платформ [5].

Простіше кажучи, REST – це будь-який інтерфейс між системами, що використовують транспортні протоколи на зразок HTTP для отримання сервісу та генерації операцій над ним у всіх можливих форматах, таких як розширювана мова розмітки (XML) та нотація об'єкта сценарію Java (JSON).

Філдінг дотримувався підходу, заснованого на обмеженнях, у процесі пошуку для REST. Тому він визначив деякі обмеження, і REST керується цими обмеженнями. Ці обмеження такі:

- Незалежність від стану. У парадигмі запит / відповідь між клієнтом і сервером необхідний стан для обробки запиту міститься в самому запиті.
- Уніфікований інтерфейс. У випадку використання протоколу HTTP для API REST, зв'язок ініціюється клієнтом і складається з запиту, за яким йде відповідь. Кожне повідомлення запиту разом із ідентифікатором ресурсу включає конкретні дії або дієслово HTTP (наприклад, GET, PUT, POST та DELETE), які визначають операцію, яка повинна виконуватися на ресурсі.
- Клієнт-сервер: Уніфікований інтерфейс, який відокремлює клієнтів від серверів, дозволяє клієнтам не турбуватися про внутрішні справи серверів і серверів, а також не хвилює інтерфейс користувача або стан клієнтів. Наприклад, клієнтів не цікавлять деталі зберігання даних кожного сервера, тому покращується продуктивність клієнтського коду, і сервери можуть стати простішими та масштабованими, щоб не турбуватися про користувальницький інтерфейс або стан користувача. Тому сервери та клієнти також можуть бути замінені та розроблені самостійно.
- Ідентифікація ресурсу за допомогою URI: Універсальний ідентифікатор ресурсу (URI) та жоден інший елемент не є єдиним ідентифікатором кожного ресурсу в цій системі REST. URI дозволяє нам отримувати доступ до інформації, щоб змінити або видалити її, або, наприклад, поділитися її точним розташуванням з третіми сторонами.
- Система шарів: ієрархічна архітектура між компонентами. Клієнт звичайно не може сказати, чи він підключений безпосередньо до кінцевого сервера чи до посередника. Посередницькі сервери можуть покращити масштабованість системи, включивши балансування навантаження та надаючи спільні кеші. Шари можуть також застосовувати політику безпеки. Кожен шар в системі REST має функціональні можливості.

- Кешування: Як і в WWW, клієнти можуть кешувати відповіді. Але відповіді повинні неявно або явно ідентифікувати себе як кешовані або не такі, щоб уникнути подальших запитів клієнтів від повторного використання черствих або невідповідних даних у відповіді. Якщо добре керувати, кешування може частково або повністю видалити взаємодію клієнт-сервер, що потім додатково покращує масштабованість та продуктивність

- Гіперлінки зі збереженим станом: Hypermedia дозволяє користувачеві переглядати набір об’єктів через гіпермедіа посилання. Що стосується API REST, то поняття гіпермедіа пояснює здатність інтерфейсу розробки додатків для надання клієнту та користувачу адекватних посилань для виконання конкретних дій над даними. Щоб зробити його справжнім, API REST повинні підтримувати Hypermedia як принцип двигуна стану заявки (HATEOAS). За даними цей принцип гарантує, що кожного разу, коли буде зроблено запит і повернуто відповідь з сервера, тоді частина інформації, що міститься у відповіді, буде переглядати гіперпосилання, пов'язані з іншими ресурсами клієнта. Ці гіперпосилання повідомляють клієнту, куди він може піти. далі, і що можливі дії в поточному стані його розмови з API. Він вимагає, щоб клієнти REST API орієнтувалися на відповіді, які вони отримують від API [6].

RESTful технологія базується на характерних елементах, відомих як ресурси, які є джерелами конкретної інформації. Щоб зрозуміти, ресурси – це будівельні блоки кожного RESTful Web API, і вони забезпечують єдиний інтерфейс, що дозволяє отримувати доступ та змінювати їх стан [6]. Кожен з них пов'язаний з глобальним ідентифікатором, наприклад URI.

Для взаємодії з ресурсом програма повинна мати ідентифікатор ресурсу, і необхідний метод. Навпаки, не потрібно знати впровадження служб та конфігурацію системи, тобто чи є кеш, проксі, шлюзи, міжмережеві стіни, тунелі чи щось інше між програмою та сервером, на якому розміщуються ресурси. Однак програма повинна бути здатна інтерпретувати формат даних (представлення), що повертається з ресурсу, який часто є документом HTML або XML, хоча це також може бути зображення, звичайний текст чи будь-який інший

вміст. Ці ресурси отримують доступ до компонентів мережі (користувальницькі агенти та сервери), які спілкуються через стандартизований протокол (наприклад, HTTP) та обмінюються вмістом (представленнями) цих ресурсів.

REST — все більш популярна альтернатива іншим стандартним протоколам обміну даними, наприклад, простий протокол доступу до об'єктів (SOAP), який має високу ємність, але також є дуже складним. Іноді бажано використовувати більш просте рішення для обробки даних, наприклад REST. У таблиці 1 наведені переваги та недоліки використання REST. REST був популярним способом викриття даних із сервера [1].

За часів, коли розроблялася концепція REST, клієнтські додатки були порівняно простими, а темпи розробки були майже не такими, якими вони є сьогодні. Таким чином, REST добре підходив для багатьох застосувань. Однак пейзаж API кардинально змінився за останні пару років. Зокрема, є три фактори, які складно підходили до розробки API-інтерфейсів: збільшене використання мобільних пристроїв, різноманітність фрейм-фреймів або платформ та швидка розробка. Ці фактори, в свою чергу, призводять до деяких інших проблем:

- Збільшення мобільного використання призводить до необхідності ефективного завантаження даних або мінімальної передачі даних
- Різноманітність різних фреймових фреймворків та платформ: неоднорідність фронтендних фреймворків та платформ, які запускають клієнтські додатки, ускладнює створення та підтримку одного API, який би відповідав усім вимогам.
- Швидкий розвиток та очікування швидкого розвитку функцій: Зміна REST API на стороні сервера призводить до змін на стороні клієнта.

Таблиця *1*

Переваги та недоліки REST

Переваги	Недоліки
✓ Простота: застосовує багато існуючих відомих стандартів (HTTP, XML, URI та MIME) ✓ Клієнти та сервери HTTP сумісні з усіма мовами програмування та операційною системою / апаратними платформами ✓ Невеликі зусилля потрібні для створення клієнта, і Служби можна перевірити, використовуючи просто веб-браузер	- Кодування великої кількості вхідних даних в ресурсі URI неможливо - Також може бути складним завдання кодування складних структур даних в URI - На даний момент веб-сервіси не мають стандартної граматики для опису веб-служб, як, наприклад, мова мови опису веб-служб (WSDL) у SOAP. - Немає стандартної лексики, яка б визначала інтерфейс веб-сервісів, і угода між споживачем послуги та виробником послуг не повинна встановлюватися.

1.3.2 GraphQL новий формат

Вважається, що це альтернативна REST. Ця нова технологія API називається Graph Query Language (GraphQL).

Це правда, що REST став стандартом для розробки веб-API вже більше десяти років. Однак він також виявився надто негнучким, щоб не відставати від швидко мінливих вимог клієнтів, які звертаються до них [7]. Зокрема, коли використовується REST, тоді відповідальність клієнтів за запит та відповідь функціональності API є номінальною. Отже, клієнт не має великого контролю над тим, яку саме функціональність потрібно запитувати чи яку функціональність отримати, оскільки майже все надається постачальником послуг.

GraphQL був розроблений, щоб впоратися з необхідністю приділяти більше відповідальності клієнтам для підвищення гнучкості та ефективності [7]. Як це було описано, GraphQL пропонує рішення для багатьох обмежень та неефективності досвідчених розробників, які взаємодіють з API REST. Наприклад, GraphQL дає користувачеві можливість запитувати будь-яку конкретну інформацію. Навпаки, користувач REST змушений робити додаткові запити, щоб отримати конкретну необхідну інформацію. Це також можливо з доброї волі постачальника послуг; якщо постачальник послуг не надає кінцеву точку

для цього запиту, тоді немає можливості отримати конкретну інформацію, необхідну користувачеві. Це було проілюстровано на рисунку 2. Тому GraphQL намагається покращити спосіб спілкування клієнтів з віддаленими системами.

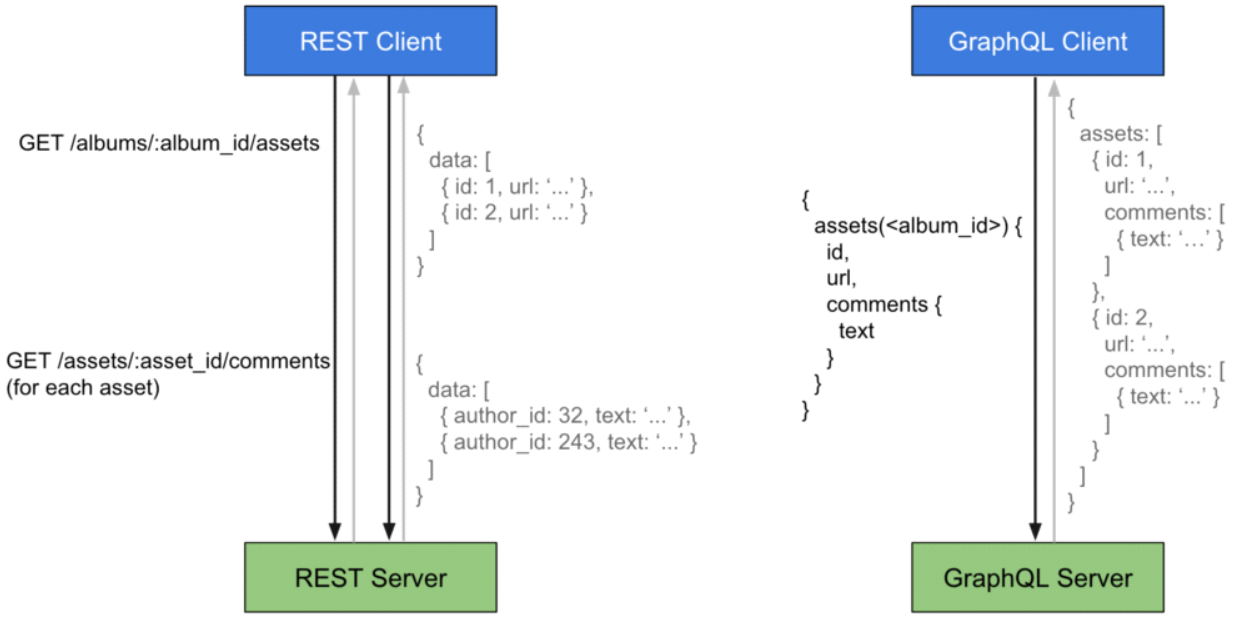

Рис. *2 Ілюстрація отримання даних. Зліва REST, справа GraphQL*

GraphQL часто плутають з технологією баз даних [8]. Це непорозуміння; GraphQL – мова запитів для API і навіть не для баз даних. Більше того, GraphQL є агностиком бази даних, і він може бути добре підходить в будь-якому контексті, де задіяний API. Подібно до сервера REST, сервер GraphQL не обмежений неспецифічною технологією чи мовою і може бути реалізований за допомогою будь-якої технології [8]. Потужність GraphQL походить від простої ідеї замість того, щоб визначати структуру відповідей на сервері; гнучкість надається клієнту. Як це наочно проілюстровано на рисунку 2, кожен запит визначає, які поля та відносини він хоче повернути, а GraphQL побудує відповідь на цей конкретний запит. Деякі переваги GraphQL порівняно з REST подано в [8].

1.4 Що таке GraphQL

Кілька років тому, ще до того, як хтось чув про GraphQL, інша архітектура API домінувала у галузі Web API: API на основі кінцевої точки. Будемо називати API на основі кінцевих точок будь-яку API, архітектура якої обертається навколо кінцевих точок HTTP. Це можуть бути API JSON через HTTP, кінцеві точки стилю RPC, REST тощо [9].

Ці API мали (і досі мають) кілька переваг. Насправді вони все ще домінують у галузі, коли справа стосується Web API. Для цього є причина. Ці кінцеві точки, як правило, досить прості у впровадженні і зазвичай можуть дуже відповідати на один варіант використання. Завдяки ретельному проектуванню API на основі кінцевої точки можна дуже оптимізувати для конкретного випадку використання. Вони легко кешуються, виявляються та прості у використанні клієнтами.

Останніми роками кількість різних типів споживачів Web API зросла. Хоча раніше веб-браузери були основними клієнтами для Web API, тепер нам доводиться створювати наші API для реагування на мобільні програми, інші сервери, які є частиною наших розподілених архітектур, ігрових консолей тощо. Навіть ваш холодильник може викликати Web API коли відчиняєш двері [9].

API на основі кінцевої точки чудово підходять для оптимізації обміну між клієнтом та сервером для однієї функціональності або випадку використання. Хитра річ у тому, що через вибух у типах клієнтів для певних API, які повинні обслуговувати багато випадків використання, створення хорошої кінцевої точки для обслуговування цих сценаріїв стало більш складним. Наприклад, якщо ви працювали на платформі електронної комерції та мусите навести випадок використання продуктів для сторінки товару, вам доведеться розглянути веб-браузери, які можуть надавати детальний огляд продуктів, мобільний додаток, який може відображати лише зображення товару на цій сторінці та ваш холодильник, який може мати дуже мінімальну версію даних, щоб

уникнути надсилання надто багато на дроті [9]. У підсумку в цих випадках відбувається те, що ми намагаємося створити універсальний API.

Універсальний API.

Що таке універсальний API? Це API, який намагається відповісти на занадто багато випадків використання. Це API, який почав оптимізуватися, як ми хотіли, і став дуже загальним через неможливість адаптації до багатьох різних способів використання загального випадку використання. Їм важко управляти розробникам API через те, наскільки вони пов'язані з різними клієнтами, а іноді і наскільки складним стає їх утримання на сервері [9].

Це стало досить поширеною проблемою з API на основі кінцевих точок, інколи звинувачували лише REST API. (Насправді REST не винен конкретно і пропонує способи уникнути цієї проблеми.) Web API, що стикаються з цією проблемою, реагували різними способами. Ми побачили, як деякі API відповідають найпростішим рішенням: додавання більшої кількості кінцевих точок, одна кінцева точка на варіацію. Наприклад, візьмемо API, який базується на кінцевій точці, який забезпечує спосіб отримання продуктів через products ресурс: GET / products.

Щоб надати версію ігрової консолі цього варіанту використання, певні API вирішили проблему таким чином:

```
GET api/playstation/products
GET api/mobile/products
```

Маючи досить великий Web API, ви можете здогадатися, що сталося з таким підходом. Кількість кінцевих точок, що використовуються для відповіді на варіанти тих самих випадків використання, вибухнула, що зробило API надзвичайно важким для розробників, дуже крихким до змін і, як правило, болем для підтримки та розвитку [9].

Не кожен обрав такий підхід. Деякі вирішили зберегти одну кінцеву точку для кожного випадку використання, але дозволяють використовувати певні параметри запиту. На найпростішому рівні це може бути дуже конкретний параметр запиту для вибору потрібної нам версії клієнта:

```
GET api/products?version=gaming
GET api/products?version=mobile
```

Деякі інші підходи були більш загальними, наприклад часткові:

```
GET api/products?partial=full
GET api/products?partial=minimal
```

А потім деякі інші обрали більш загальний підхід, дозволивши клієнтам вибрати те, що вони хочуть повернути із сервера. Специфікація JSON: API називає їх розрідженими наборами полів:

```
GET api/products?include=author&fields[products]=name,price
```

Деякі навіть доходили до створення мови запиту в параметрі запиту. Погляньте на цей приклад, натхненний Google Drive API:

```
GET api/products?fields=name,photos(title,metadata/height)
```

Усі розглянуті нами підходи роблять компроміси самі по собі. Більшість із цих компромісів знаходяться між оптимізацією (наскільки оптимізованою для одного випадку є кінцева точка) та настройкою (наскільки кінцева точка може адаптуватися до різних випадків використання або варіацій) [9].

Приблизно в 2012 році різні компанії стикалися з проблемою незрозумілого API, і багато з них почали думати про способи зробити більш настроюваний API для розробників з великим досвідом.

Розширений опис схеми роботи з GraphQL має на меті представити максимально детально роботу з GraphQL та її наявними аналогами. Для деталізації конструкції запитів можна звернутись до основного поняття запиту як такого.

GraphQL — це стандарт декларування структури даних і способів отримання даних, який виступає додатковим шаром між клієнтом і сервером. Однією з основних відмінних рис GraphQL є те, що структура і обсяг даних визначається клієнтським додатком [4].

На відміну від стандартного REST-підходу, що має кілька точок входу, що виконують прості завдання, GraphQL містить тільки одну точку входу, яка працює з усіма запитами. Фактично, GraphQL знаходиться між клієнтом і одним або декількома джерелами даних; він приймає запити клієнтів, перенаправляє ці запити відповідним обробникам і повертає необхідні дані відповідно до переданих інструкцій.

1.5 Аналіз SOA — Сервіс Орієнтована Архітектура

Служба (або сервіс, послуга) — це функція, яка є чітко визначеною, самодостатньою і не залежить від контексту чи стану інших служб [10]. Щоб зрозуміти, послуга — це діяльність або завдання, яке завжди стає доступним для її споживачів.

Споживачам послуги може не знадобитися впроваджувати та підтримувати її функціональність. Споживачі вживають її, не турбуючись про це, і вони трактують це як «чорну скриньку». Вона може бути запропонованою споживачеві через різні транспортні системи, якості та уявлення, які можуть допомогти споживачам визначити, що найкраще під їхні потреби.

Потреби споживача зазвичай не можуть бути задоволені лише однією службою. Тому ці послуги потрібно з'єднувати деякими способами, щоб спілкуватися між собою. Комунікація може включати в себе просту передачу даних або також може включати два або більше служб, що координують певну діяльність. З точки зору програмної архітектури, це відоме як сервісно-орієнтована архітектура (SOA). SOA – це в основному сукупність служб, які потребують зв'язку між собою. Щоб зробити його лаконічним, SOA можна описати як архітектурний стиль, який визначає, як можна будувати програми на основі сервісів які представляють компоненти програми.

Визначення SOA може бути більш детально розроблено за допомогою наступних трьох принципів:

- *Перевикористання компонентів:* Важливо розкласти бізнес-програми на бізнес-компоненти таким чином, щоб якомога більше компонентів були загального призначення (багаторазові) та якомога менше були спеціальні [10].
- *Підтримка веб-сервісів:* Компоненти повинні мати чітко визначені сервісні інтерфейси, які можна зберігати в каталозі, щоб споживач послуг міг запитувати каталог інтерфейсів для виявлення та виклику необхідних постачальників послуг. В даний час веб-сервіс (WS) – це розвинута технологія. WS забезпечує широко прийнятий механізм визначення послуги через WSDL, який можна визначити та виявити через універсальний каталог, опис, виявлення та інтеграцію (UDDI) шляхом обміну XML-повідомленнями за допомогою HTTP через Інтернет [11].
- *Enterprise Service Bus (ESB):* Замість точкових комунікацій між учасниками слід використовувати слабко пов'язану загальну інфраструктуру програмного забезпечення для комунікацій, посередництва, безпеки, служб каталогів та адміністрування, необхідних на всьому підприємстві. Хоча таку інфраструктуру можна забезпечити існуючими платформами інтеграції корпоративних прикладних програм (EAI), моделі SOA наполегливо пропонують WSenabled ESB для SOA [11].

З точки зору споживачів SOA справляє великий позитивний вплив завдяки особливостям та характеристикам, які він пропонує під час створення програм [11]. Наприклад, SOA може запропонувати зв'язане з'єднання, повторне використання послуг та неоднорідну сумісність. Враховуючи ці переваги, споживач може викликати функцію, не знаючи про місце, платформу чи рамки послуги. Цього можна досягти, використовуючи певне проміжне програмне забезпечення, яке приховує всі складності, необхідні для успішного завершення взаємодії. Технологія Web Service (WS) — приклад технології SOA, яка дозволяє будувати розподілені програми.

1.5.1 Протокол SOAP

SOAP — протокол обміну повідомленнями, широко розгорнутий за допомогою WS-технологій, а також є альтернативою REST та JSON [12]. Він не визначає стандартного транспортного протоколу для передачі повідомлень між постачальниками та споживачами, і він використовується поверх багатьох транспортних протоколів, але в основному використовується протокол HTTP.

Протокол простої передачі пошти (SMTP) також може використовуватися для передачі повідомлень SOAP. HTTP є ефективним транспортним протоколом для надсилання та прийому SOAP-повідомлень [13]. HTTP, своєю чергою, відомим чином використовується веб-браузерами для доступу до веб-ресурсів. Однак можуть використовуватися й інші протоколи, такі як SMTP або FTP. Компоненти розподілених додатків також можуть використовувати SOAP як опцію для обміну даними та інформацією по мережі. SOAP можна описати як архітектуру для обміну повідомленнями в розподілених середовищах. В основному використовується технологіями WS для полегшення взаємодії між постачальниками послуг та парадигмою споживачів. Структура повідомлення SOAP на рисунку 3.

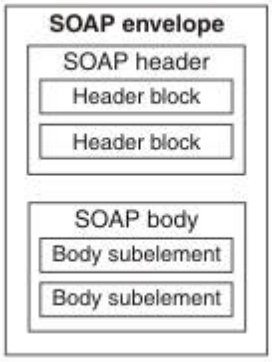

Рис. *3 Структура SOAP повідомлення*

1.5.2 Взаємодія з WSDL

WSDL – це мова опису веб-сервісів. Веб-сервіси повинні описуватися послідовно, щоб вони могли публікуватися постачальниками послуг. Опису-

ючи їх, вони можуть бути виявлені клієнтами служб та розробниками та зібрані в керованій ієрархії складених служб, які організовані для доставки службових рішень з доданою вартістю та складених програмних комплектацій [13]. Це дуже важливо для розробки прикладних програм та бізнес-процесів, які складають сервісні збірки. Для цього споживачі повинні точно визначити XML-інтерфейс веб-служби разом з іншими різними деталями повідомлення. Річ у тому, що схема XML є багатослівною, і це може частково допомогти, оскільки дозволяє розробникам описувати структуру повідомлень XML (див. рис. 4), зрозумілих веб-службам. На жаль, однієї схеми XML недостатньо, оскільки вона може не описувати важливих додаткових деталей, пов'язаних із спілкуванням із веб-службою, такими як функціональні та нефункціональні сервісні характеристики або сервісні політики [14].

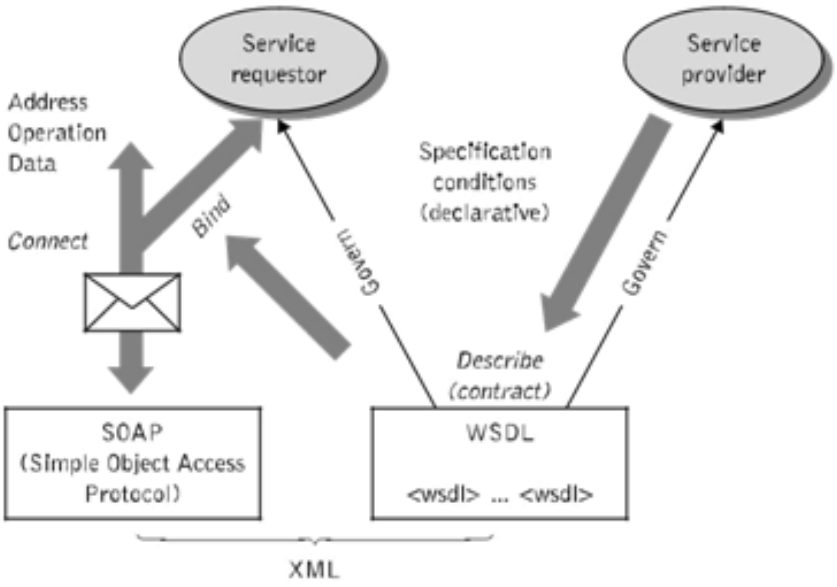

Рис. *4 WSDL регулювання взаємодії між споживачем послуги та постачальником послуг*

1.6 Open Data Protocol специфікація

OData (Open Data Protocol) — стандарт OASIS, який визначає кращі практики створення та використання API RESTful. OData допомагає вам зосередитись на вашій бізнес-логіці під час створення API RESTful, не турбуючись про підходи до визначення заголовків запитів і відповідей, кодів статусу, методів HTTP, конвенцій URL-адрес, типів медіа, форматів навантаження та варіантів запитів тощо. OData також керує вами щодо відстеження змін, визначення функцій / дій для процедур багаторазового використання та надсилання асинхронних / пакетних запитів тощо. Крім того, OData забезпечує можливість розширення для задоволення будь-яких спеціальних потреб ваших RESTful API [15].

OData RESTful API легко споживати. Метадані OData, машиночитаний опис моделі даних API, дозволяє створити потужні загальні клієнтські проксі та інструменти. Деякі з них можуть допомогти вам взаємодіяти з OData навіть не знаючи нічого про протокол. Наступні 6 кроків демонструють 6 цікавих сценаріїв споживання OData на різних платформах програмування.

З точки зору принципів REST "Все є ресурсом". Як ресурси можна отримати з API RESTful OData. Використовується зразкова послуга – ця послуга наприклад TripPin, яка використовує сервіс, що відкриває систему управління відвідуванням. Наш друг Максим, який зареєструвався в TripPin (див. лістинг 1, лістинг 2).

Лістинг 1 – *Формування запиту на список ресурсів*

```
GET https://services.odata.org/v4/TripPinServiceRW/People HTTP/1.1
OData-Version: 4.0
OData-MaxVersion: 4.0
```

Лістинг *2* – *Формування відповіді*

```
HTTP/1.1 200 OK
Content-Length: 1007
Content-Type: application/json; odata.metadata=minimal
OData-Version: 4.0
{
```

```
      '@odata.context': 'https://ser-
vices.odata.org/V4/(S(a2k31bgwiyejn2j2iiybvq4p))/TripPin-
ServiceRW/$metadata#People',
      '@odata.nextLink': 'https://ser-
vices.odata.org/V4/(S(a2k31bgwiyejn2j2iiybvq4p))/TripPinServiceRW/Peo-
ple?%24skiptoken=8',
      'value': [
          {
              '@odata.id': 'https://ser-
vices.odata.org/V4/(S(a2k31bgwiyejn2j2iiybvq4p))/TripPinServiceRW/Peo-
ple('russellwhyte')',
              '@odata.etag': 'W/'08D1D5BD423E5158'',
              '@odata.editLink': 'https://ser-
vices.odata.org/V4/(S(a2k31bgwiyejn2j2iiybvq4p))/TripPinServiceRW/Peo-
ple('russellwhyte')',
              'UserName': 'maxus',
              'FirstName': 'Max'
      …
```

Принципи REST також говорять про те, що кожен ресурс ідентифікується унікальним ідентифікатором. OData також дозволяє визначити ключові властивості ресурсу та отримати його за допомогою клавіш. На цьому кроці Максим хоче знайти інформацію про себе, вказавши своє ключове ім'я користувача [15].

Оскільки архітектура, побудована на основі поточних особливостей Інтернету, API RESTful також може підтримувати рядки запитів. Для цього OData визначає ряд параметрів системного запиту, які можуть допомогти вам побудувати складні запити для потрібних ресурсів. За допомогою цього наш друг Максим може дізнатися перших 2 осіб у системі, які зареєстрували принаймні одну поїздку, яка коштує більше 3000, і відобразити лише своє ім’я та прізвище.

Принципи REST вимагають використання простих та рівномірних інтерфейсів. З огляду на це, клієнти OData можуть очікувати уніфікованих інтерфейсів ресурсів. Передача без стану в REST здійснюється за допомогою різних методів HTTP у запитах. Пройшовши перші 3 кроки, Максим вважає систему корисною. Він хоче додати свого кращого друга Андрія до системи. Він з'ясовує, що все, що йому потрібно зробити, – це надіслати запит POST, що містить JSON-представлення інформації Андрія, на той самий інтерфейс, з якого він запитував інформацію людей [15].

У RESTful API зазвичай залежать один від одного. Для цього поняття взаємовідносин в OData можна визначити декілька ресурсів, щоб додати гнучкість та багатство моделі даних. Наприклад, у службі TripPin OData люди пов’язані з поїздками, які вони забронювали за допомогою системи. Знаючи це, Максим хотів би запросити Андрія до своєї існуючої поїздки в США, пов’язавши цю поїздку з Андрієм.

У RESTful API можуть бути деякі спеціальні операції, що містять складну логіку і часто їх можна використовувати. З цією метою OData підтримує визначення функцій та дій для представлення таких операцій. Вони також є самими ресурсами і можуть бути пов'язані з існуючими ресурсами. Розвівши службу TripPin OData, Максим виявляє, що у неї є функція під назвою GetInvolvedPeople, за допомогою якої він може дізнатися залучених людей до конкретної поїздки. Він використовує функцію, щоб дізнатися, хто ще, крім нього, і Льюїс вирушає в ту поїздку в США.

Open API не залежить від платформи та постачається з декількома інструментами, які можуть генерувати програмний код, документацію та тестові випадки із специфікацій Open API. Генератор коду може генерувати клієнтський та серверний код і доступний для різних мов програмування, наприклад, Java.

1.7 Можливості OData

Можливості та сильні сторони OData:

1. Серверна реалізація, яка може бути реалізована на більшості мов програмування, з великою кількістю публічно доступних бібліотек.
2. Суворий стандарт, який добре задокументований і зрозумілий для впровадження.
3. Хороші показники роботи, без дотримання принципів REST.
4. Більшість програмістів знайомі з REST, тому їм легше використовувати рамку на основі REST.

5. Дозволяє надсилати рядки запитів із спеціальними умовами та фільтрами, використовуючи задокументований синтаксис OData.

6. Підтримує Delta Feeds, задокументований стандартний спосіб надання специфічного для клієнта (або поблизу конкретного клієнта, не маючи загального стану) посилання, яке, якщо буде викликано, надаватиме лише нові записи в масиві.

7. Виявлення з URI з метаданими.

8. За допомогою HATEOAS можна вирішити часто цитовану скаргу за допомогою OData: він не підтримує версію. Аргумент OData, що не підтримує версію чи застаріліть поля, є суперечливим, оскільки розвинути навколо цієї проблеми надзвичайно просто.

9. З'єднання даних та зв'язки підтримуються концепцією "відносини", хоча відносини часто потребують декількох викликів, оскільки відносини визначені в URI у відповіді.

10. Суб’єкти визначаються як частина URI, що технічно призводить до декількох викликів / зворотних звертань до API, це більше робота для розробника, але може бути розроблена, оскільки статичний URI легше кешувати і, отже, масштабувати легше.

API OData добре підходить, коли ви знаєте, що споживачі вашого API походять із широкої групи розробників різних технологій, часто з нахилом для підприємства. OData – простіша концепція і має більш просте визначення для запитів, ніж GraphQL, які є достатньо потужними для задоволення більшості випадків використання. Комбінувати стандартні параметри REST з OData порівняно просто для підтримки розробників з різним рівнем складності запитів, від просто виклику URI, до вибору комплексу. Якщо я, як організація, хочу контролювати дані, отримані API, але не хочу необмежених запитів даних з міркувань продуктивності, OData представляє привабливий варіант як сильний стандарт RESTful з розширеним набором функцій [15, 16].

Переваги GraphQL:

- ✓ Як і OData, існує безліч бібліотек для впровадження, виявлення та запиту GraphQL.
- ✓ Єдина кінцева точка є великим плюсовим фактором для GraphQL в тому, що її не можна було б простіше визначити споживачам, що використовує її. Розробнику потрібні лише дві речі: URI та знання про те, що API відповідає стандарту GraphQL.
- ✓ Підтримує сортування, приєднання, відносини, вибір, зменшення даних, версію, депрекацію, розбиття на сторінки та інше з однієї поїздки на запит / відповідь.
- ✓ Підтримує "Підписки", подібні, але потужніші, ніж Delta канали в OData.
- ✓ Підтримує функції, аргументи, мутацію, псевдоніми, фрагменти, директиви, об'єднання, інтерфейси та всі статично введені мови. По суті, GraphQL — це автономна мова запитів та впровадження, яка, мабуть, є більш потужною, ніж OData.

1.8 Мікросервісна робота з GraphQL

GraphQL дозволяє отримувати дані з декількох джерел одночасно або навіть з різних мікросервісів, що працюють на різних серверах. структурна схема роботи GraphQL представлена на рисунку 5.

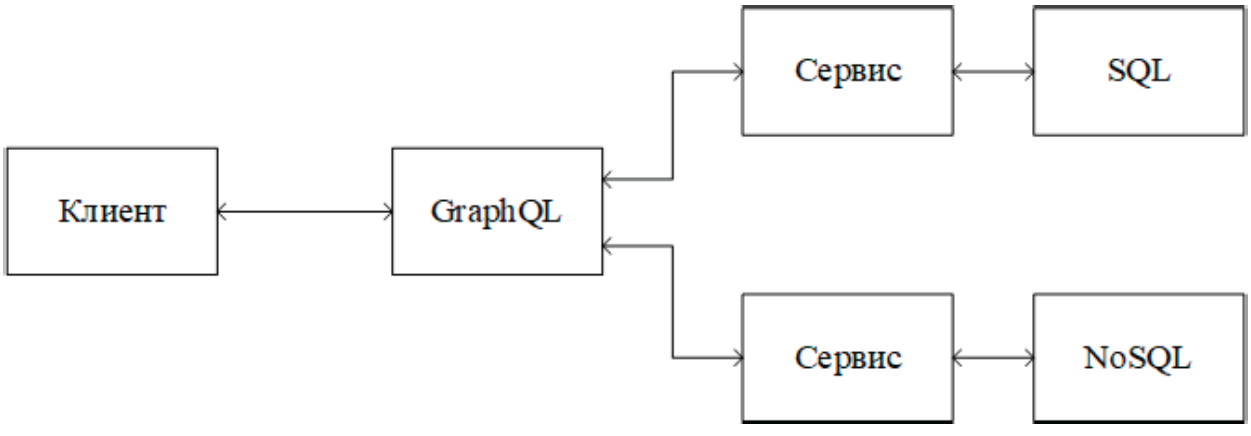

Рис. 5 *Блок-схема типових мікросервісів*

Технологія GraphQL сама не може обробити вхідний запит, тому необхідно вказати джерела даних для кожного об'єкта в графі. Для цього створені спеціальні обробники, базова структура яких представлена в лістингу 3.

Лістинг *3 Приклад обробника запиту GraphQL*

```
Query: {
      authors(args) {
      return authorsRepository.findById(args.id);
      }
}
```

При зустрічі в запиті поля authors, GraphQL знаходить необхідний оброблювач за відповідним ім’ям і повертає результат його виконання.

Крім отримання даних, GraphQL підтримує функції їх створення і зміни. Для цього існують обробники, звані мутаціями. Принцип їх роботи схожий на розглянутий вище, тільки дані не зчитуються, а оновлюються [16]. Приклад запиту на мутацію представлений в лістингу 4.

Лістинг *4 Приклад розпізнавача в GraphQL*

```
mutation {
      changeUserName(id: “1”,
      firstName: “Alex”,
      lastName: ”Bochovski”)
      {
      birthDay
}
 }
```

Дана мутація оновить користувача з ідентифікатором 1: встановить йому нове ім'я і прізвище, а також поверне дату народження цього користувача.

Для того, щоб технологія GraphQL розуміла, які є об'єкти в графі, як вони пов'язані і які є мутації, існує описовий файл під назвою схема. У схемі описані кожен об'єкт графа з типами даних полів, зв'язку з іншими об'єктами, сигнатури мутацій.

1.9 Архітектурні рішення

Архітектурне рішення поділу запитів і мутацій відмінно поєднується з принципом CQRS (Command-query responsibility segregation) винайдений Бертраном Мейером [5], який є одним з найважливіших принципів побудови великих і розширюваних систем. Оскільки поділ запитів на читання і команд на зміну даних дозволяє позбутися від безлічі проблем з неочікуваною зміною даних (див. рис. 6).

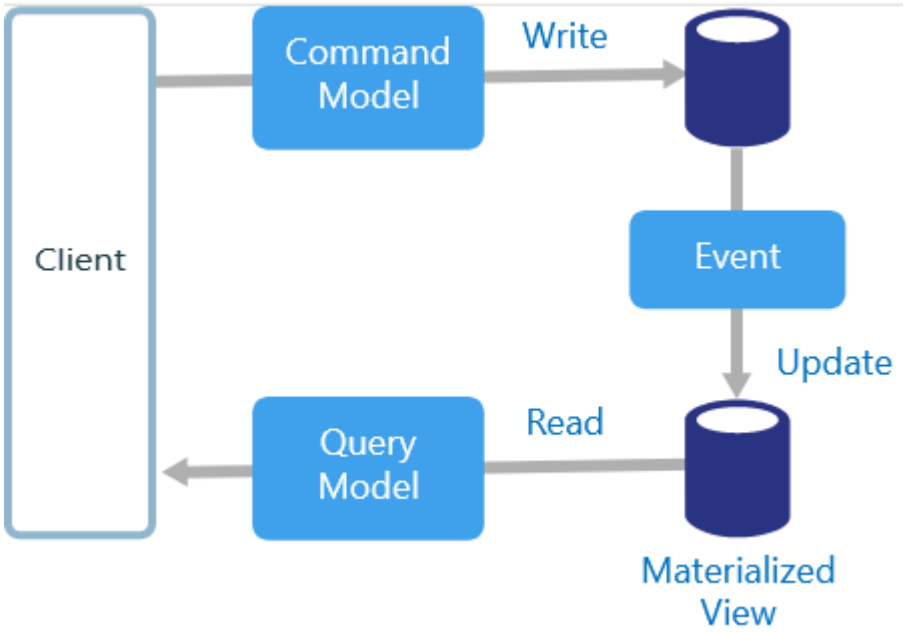

Рис. 6 *CQRS схема*

Технологія GraphQL крім запитів і мутацій містить інші можливості, що покращують швидкість і простоту написання запитів. До них відносяться:

- псевдоніми — дозволяють давати різні імена для об'єктів графа, що дозволяє отримувати кілька об'єктів одного типу в одному запиті;
- фрагменти — дозволяють винести в окрему змінну повторювані назви полів для різних об'єктів, що зменшує розмір запиту і підвищує його читаність;
- змінні — дозволяють винести значення будь-яких аргументів з тіла запиту GraphQL і передавати їх окремо, що дає можливість не змінювати тіло

запиту для різних аргументів, таких як, наприклад, ідентифікатор сутності [16].

1.10 Різноманітність і вибір баз даних

Реляційні бази даних були потужними програмними додатками з 80-х років і залишаються зараз. Вони зберігають високо структуровані дані в таблицях з зумовленими стовпцями певних типів і безліччю рядків одного і того ж типу інформації і, частково завдяки жорсткості їх організації, вимагають від розробників додатків строго структурувати дані [17].

У реляційних базах даних посилання на інші рядки і таблиці вказуються шляхом посилання на їх (первинні) атрибути ключів через стовпці зовнішнього ключа. Це можна виконати з обмеженнями, але тільки тоді, коли посилання ніколи не є обов'язковим. Вибірка обчислюється під час запиту, зіставляючи первинні і зовнішні ключі багатьох рядків таблиць, які підлягають об'єднанню. Ці операції розраховані на обчислення і пам'ять і мають експонентну вартість [17].

Якщо ви використовуєте зв’язки «багато-до-багатьох», вам необхідно представити таблицю JOIN (або таблицю об’єднань), яка містить зовнішні ключі обох таблиць-учасників, що ще більше збільшує витрати на операції об'єднння. Ці дорогі операції об'єднання зазвичай вирішуються шляхом денормалізації даних, щоб зменшити кількість необхідних вибірок [17].

Хоча не кожен варіант використання підходить для такого типу точних моделей даних, в минулому відсутність життєздатних альтернатив і велика підтримка реляційних баз даних ускладнювали створення альтернатив[17].

Вплив графових баз даних на архітектуру.

Зв’язки першокласних мереж у моделі графових баз даних, на відміну від інших систем управління базами даних, які вимагають від нас встановлення зв’язків між об'єктами, використовують спеціальні властивості, такі як

зовнішні ключі. Об'єднавши прості абстракції вузлів і зв'язків в пов'язаних структурах, графові бази даних дозволяють нам створювати складні моделі, які тісно пов'язані з нашою проблемною областю [17].

У деяких відносинах графові бази даних схожі на наступне покоління реляційних баз даних, але з підтримкою першого класу для «зв’язків» або з неявними сполуками, зазначеними за допомогою зовнішніх ключів в традиційних реляційних базах даних [17].

Кожен вузол (суб'єкт або атрибут) в моделі графових баз даних безпосередньо і фізично містить список взаємопов'язаних записів, які представляють його зв'язок з іншими вузлами. Ці взаємопов’язані записи організовані за типом і напрямком, і можуть містити додаткові атрибути. Всякий раз, коли ви запускаєте еквівалент операції JOIN, база даних просто використовує цей список і має безпосередній доступ до пов'язаних вузлів, що усуває необхідність в дорогому обчисленні пошуку / зіставлення [17].

Графова база даних — це база даних, призначена для обробки зв’язків між даними як першокласна мережа в моделі даних. Ми живемо в взаємозв’язаному світі. В ньому немає ізольованих частин інформації, але багато, пов’язаних структур навколо нас. Лише база даних, яка спочатку підтримує зв’язки, здатна ефективно зберігати, обробляти і запитувати з'єднання. У той час як інші бази даних обчислюють їх під час запиту через дорогі операції JOIN[17].

Доступ до вузлів і зв’язків у графовій базі даних — це ефективна неперервна операція, яка дозволяє швидко зчитати мільйони підключень в секунду на вузол. Незалежно від загального розміру вашого набору даних, графові бази даних перевершують управління високоповя’заними даними і складними запитами. Маючи тільки шаблон і набір вихідних точок, графові бази даних досліджують зв’язки навколо початкових точок, збір та його узагальнення інформації з мільйонів вузлів і зв’язків — залишаючи мільярди поза периметром пошуку недоторканими [17].

Модель графової бази даних.

Якщо працювати з об'єктною моделлю або діаграмою зв’язків сутностей, модель з мітками властивостей буде здаватися знайомою. Вузли — це об'єкти на графу. Вони можуть містити будь-яку кількість атрибутів (пари ключ-значення), так звані властивості. Вузли можуть бути позначені ярликами, які представляють їхні різні ролі. На додаток до контекстуалізації властивостей вузла і зв’язків, мітки можуть також служити для прикріплення інформації про індекс метаданих або обмеження до певних вузлів [17].

Зв’язки забезпечують направлення, іменовані, семантично релевантні для зв'язку між двома вузлами (наприклад, Employee WORKS_FOR Company). Зв’язок завжди має напрямок, тип, початковий вузол і кінцевий вузол. Як і вузли, зв’язки також можуть мати властивості.

У більшості випадків відносини мають кількісні характеристики, такі як ваги, витрати, відстані, рейтинги, інтервали часу або стійкість. Оскільки відносини зберігаються ефективно, два вузла можуть ділитися будь-яким числом або типом зв’язків, не жертвуючи продуктивністю. Зв’язки можуть бути задані в любому порядку[17]. Оскільки зв’язки завжди мають початковий і кінцевий вузол, не можна видалити вузол, не видаляючи також пов'язані з ним зв’язки [17].

1.11 Висновки до розділу 1

В цьому розділі було розглянуто основні технології і архітектури, які використовують при розробці API для взаємодії в розподіленому середовищі.

GraphQL у більшості обставин є більш потужною мовою запитів, але це буде важко реалізувати і не буде RESTful. OData є RESTful і має меншу криву навчання для споживачів вашого API, якщо вони не знають синтаксису запитів. Однак найважливіше, хто є вашим споживачем і яким найшвидшим, найвигіднішим способом ви зможете задовільнити потреби цього споживача використовувати ваше API. І OData, і GraphQL мають силу та простоту для використання в проектах.

Але є випадки, коли впровадження будь-якої технології було б нераціональним, коли ви намагаєтесь вирішити проблеми складними замість простих методами. Багато чого можна досягти, просто скориставшись універсальним інтерфейсом OpenAPI або дотримаючись RESTful стандарту вашого власного виготовлення.

РОЗДІЛ 2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ТЕХНОЛОГІЇ GRAPHQL

2.1 Оглядова оцінка GraphQL і REST

Для дослідження навантаження запитів GraphQL на мобільному застосунку став вибір між використанням технології для передачі даних. У цьому підрозділі будуть описані усі позитивні та негативні ключові особливості використання REST або GraphQL.

По-перше маючи досвід з REST можна стверджувати що REST має проблеми:

- Надлишковсть запитів
  - */products?field=name&field=description&field=variants[*].price*
- Нестача даних для отримання, необхідність уточнення на інший маршрут
  - */products?expand=productType&expand=variants[*].price.taxRate*
- API зміни та еволюція запитів
  - Версії (v1/v2/v3/…)
  - Застарілість запитів
  - Обслуговування кожного маршрута

Стосовно версіонності, не можна так легко взяти і змінити існуючий маршрут при активній розробці з клієнтом без обговорення з кінцевими користувачами. З мобільними пристроями це щє складніше, тому що кожна зміна версіонності стимулює переробляти моделі даних клієнтів. Контролювати версіонність на стороні бекенду є дуже затратним з точки зору часу і зусиль розробників. У GraphQL цієї проблеми немає.

Ключове для REST поняття — ресурс. Кожен ресурс ідентифікується за його URL-адресою, а для отримання ресурсу необхідно надіслати GET-запрошення до цього URL. Скоріше за все, відповідь прийде у форматі JSON, так як

саме цей формат використовується зараз у більшості API. Виглядати це буде приблизно так (див. лістинг 5).

Лістинг *5 Ресурс у формі REST*

```
GET /books/1

{
  "title": "John Ingram Adventures",
  "author": {
    "firstName": "Henry",
    "lastName": "Smith"
  }
  // ... інші поля
}
```

Зауваження: для розглянутого вище прикладу деякі REST API можуть повертати дані про автора (поле «author») як окремий ресурс. Одна з помітних особливостей REST полягає в тому, що тип, або форма ресурсу, і спосіб отримання ресурсу зчеплені воєдино. Говорячи про розглянутому вище прикладі в документації по REST API, ви можете послатися на нього як на «book endpoint» [18].

Лістинг *6 Підготовлені типи GraphQL*

```
type Book {
  id: ID
  title: String
  published: Date
  price: String
  author: Author
}
type Author {
  id: ID
  firstName: String
  lastName: String
  books: [Book]
}
type Query {
  book(id: ID!): Book
  author(id: ID!): Author
}
```

У лістингу 6 ми бачимо, що в GraphQL-запиті URL містить як потрібний нам ресурс, так і опис цікавих для нас полів. Крім того, вже не розробник сервера вирішує за нас, що потрібно включити у відповідь пов'язаний ресурс author, – це тепер рішення клієнта, що використовує API [18].

Але, що більш важливо, сутності ресурсів, поняття Books і Authors, не прив'язані до способу їх вилучення. Можна було б брати одну і ту ж книгу за допомогою запитів різного типу і з різним набором полів [18].

Висновки до розділу:

- *Подібність*: є поняття ресурсу, є можливість призначати ідентифікатори для ресурсів.
- *Подібність*: ресурси можуть бути вилучені за допомогою GET-запиту URL-адреси по HTTP.
- *Подібність*: відповідь на запит може повертати дані в форматі JSON.
- *Різниця*: в REST викликається вами кінцева точка (endpoint) — це і є сутність об'єкта. У GraphQL сутність об'єкта відокремлена від того, як саме ви його отримуєте.
- *Різниця*: в REST структура і обсяг ресурсу визначаються сервером. У GraphQL сервер визначає набір доступних ресурсів, а клієнт вказує необхідні йому дані прямо в запиті.

2.2 Клієнти GraphQL

Існує дуже багато клієнтів для роботи з GraphQL. Задачею даної кваліфікаційної роботи було побудувати можливість використати GraphQL на мобільному пристрої. Можна робити запити вручну, але це буде вимагати додаткових зусиль з обробки запитів і відповідей. Серед існуючих систем популярними є:

- graphql-request — він обгортає запити на вибірку в “Promise”, який може бути застосований для запитів на сервер GraphQL. Крім того, він обробляє деталі запиту і виконує аналіз даних [19].

- Apollo Client — це проект гнучкого клієнтського рішення GraphQL для виконання таких процедур, як кешування, поновлення користувальницького інтерфейсу і багато іншого. Команда розробників створила пакети, які надають зв'язки для React, Angular, Ember, Vue, iOS і Android [19].
- Relay — розроблено компанією Facebook, посередник між компонентами React та даними, які експортуються з серверу GraphQL [19].

Клієнт Apollo був розроблений компанією Meteor Development Group під ініціативою спільноти зі створення всеосяжного інструментарію GraphQL. Клієнт Apollo підтримує всі основні платформи фронтенд-розробки і незалежний від платформи. Apollo також реалізує інструменти, які допомагають у створенні сервісів GraphQL, підвищенні продуктивності бекенд-сервісів і впровадженні інструментів для моніторингу продуктивності API GraphQL.

Перевагу отримує Apollo Client за наявність максимальної підтримки з боку спільноти розробників.

2.3 Аналіз синтаксису формування запитів

На формування запитів до GraphQL повинні визначитись з простими та комплексними типами. GraphQL поділяє поля на скалярні типи або типи об’єктів. Скалярні типи схожі на примітиви в інших мовах. Це листя наших вибірок. У GraphQL є п'ять вбудованих скалярних типів:

- цілочисельні (Int),
- з плаваючою комою (Float),
- рядки (String),
- логічні (Boolean)
- унікальні ідентифікатори (ID).

Як цілі числа, так і числа з плаваючою комою повертають числа JSON, а рядки і ідентифікатори повертають рядки JSON. Логічні типи просто повертають логічні значення. Незважаючи на те що ID і String повернуть дані JSON

того ж типу, GraphQL все одно перевіряє, чи повертають ідентифікатори унікальні рядки. Типи об’єктів це групи одного або декількох полей [20].

Окрім типів спосіб отримання даних поділяються на «Запити», «Мутації», «Підписки». Перші потрібні для отримання даних, другі для зміни, треті для контролю змін (аналогічно вебсокетам для прослуховування змін даних).

Документ запиту GraphQL може містити визначення операцій і фрагментів. Фрагменти — це виборки, які можна використати повторно (див. лістинг 7).

Лістинг *7 Приклад фрагменту*

```
fragment liftInfo on Lift {
     name
     status
     capacity
     night
     elevationGain
}
```

Фрагменти створені за допомогою ключового ідентифікатора “fragment” і базуються на родительському типі. Є дуже багато варіативностей використання фрагментів, щоб не дублювати логіку запиту.

Окрім фрагментів є об’єднання, грубо кажучи це варіант поєднання фрагментів (див. лістинг 8).

Лістинг *8 Поєднання фрагментів*

```
query schedule {
     agenda {
     ...on Workout {
          name
          reps
     }...on StudyGroup {
          name
          subject
          students
          }
```

```
    }
}
```

Перерахування — спеціальний скалярний тип, який обмежений певним набором значень (див. лістинг 9).

Лістинг *9 Перерахування (enumerator)*

```
enum HAIR_COLOR {
    BLACK
    BLONDE
    BROWN
    GREY
}
```

Також в GraphQL є 2 абстрактних типу: union і interface. Вони можуть використовуватися тільки як повертаємий тип, тобто можна лише отримати дані такого типу, але не передати в запит в якості аргументу [21].

Інтерфейси — це абстрактний тип, що включає в себе набір обов'язкових полів, які повинні включати в себе типи, що успадковують цей інтерфейс. Якщо це не буде виконано, то станеться помилка валідації схеми. Приклад наведено у лістингу 10.

Лістинг *10 Інтерфейс GraphQL*

```
interface Character {
    id: ID!
    name: String!
}
type Droid implements Character {
    id: ID!
    name: String!
    function: String
}
type Human implements Character {
    id: ID!
    name: String!
    starships: [Starship]
}
```

Інший абстрактний тип, не включає обов'язкові поля. Може застосовуватися там, де необхідно використовувати сімейство з типів, у яких немає загальних полів, наприклад при реалізації пошуку або складних підписок [21]. Приклад наведено у лістингу 11.

Лістинг *11 Тип union*

```
union SearchResult = Human | Starship | Film
... on Human {
    name
    height
}
... on Starship {
    model
    capacity
    manufacturer
}
... on Film {
    title
    episode
}
```

Union-типи використовуються в тих місцях схеми, де можна сказати, що тут можна повернути один з перерахованих типів. У свою чергу, інтерфейси використовуються там, де про тип можна сказати, що він реалізує цей контракт [21].

Мутації — це такий вид запитів, що не обмежуються лише зчитуванням даних. Їх зазвичай використовують для зміни даних. Приклад мутації для створення даних показано у лістингу 12.

Лістинг *12 Приклад мутації*

```
mutation createSong {
 addSong(title:"No Scrubs", numberOne: true,
         performerName:"TLC") {
     id
     title
     numberOne
 }
}
```

Відповідь формується як для звичайного “Query” запиту [18].

Запити до мутацій можуть бути сформовані з використанням змінних, цей варіант можна вдало використовувати для передачі змінних як параметрів до запиту, щоб мати можливість використовувати одну і ту ж структуру, але для варіації даних (див. лістинг 13).

Лістинг *13 Приклад мутації зі змінними*

```
    mutation createSong($title:String! $numberOne:Int
$by:String!) {
      addSong(title:$title, numberOne:$number-
One, performerName:$by) {
          id
          title
          numberOne
      }
    }
```

Підписки — це ще один різновид отримання інформації, широко використовується коли клієнт може захотіти отримувати оновлення в реальному часі. Підписка перехоплює події API GraphQL для зміни даних в реальному часі. Приклад формування запиту на підписку у лістингу 14.

Лістинг *14 Приклад підписки*

```
subscription {
    liftStatusChange {
       name
       capacity
       status }}
```

2.3.1 Функції-розв’язувачі

Розв’язувач (Resolver) — це функція, яка повертає дані для певного поля. Функції “Resolver” повертають дані в типі і формі, заданих схемою. Розв’язувачі можуть бути асинхронними і можуть отримувати або оновлювати дані з REST API, бази даних або будь-якого іншого сервісу. Приклад розв’язувача наведено у лістингу 15.

Лістинг *15 Функція розв'язувач*

```
const typeDefs = `
     type Query {
          totalPhotos: Int!
     }
`

const resolvers = {
     Query: {
          totalPhotos: () => 42
     }
}
```

Змінна *typeDefs* визначає нашу схему. Це просто рядок. Всякий раз, коли ми створюємо такий запит, як *totalPhotos*, він повинен бути підкріплений функцією розв'язувача з тим же ім'ям. Визначення типу описує тип, який має повертати поле. Функція-розв'язувач повертає дані цього типу - в нашому випадку просто статичне значення 42 [19].

Також важливо відзначити, що розв'язувач повинен бути визначений як об'єкт з тим же ім'ям, що і об'єкт в схемі. Поле *totalPhotos* є частиною об'єкта запиту. Розв'язувач для цього поля також повинен бути частиною об'єкта *Query*.

2.3.2 GraphQL інтроспекція

Для отримання інформації про всі типи, які підтримує даний GraphQL API, можна виконати так званий introspection query. За замовчуванням на дебаг-версії сервера цей запит доступний, на production, природно, повинен бути відключений [21]. Також роботу з API спрощує використання GraphQL Playground — це графічна інтерактивна середовище розробки, що базується на GraphiQL, повноцінної IDE для роботи з GraphQL. При розробці і використанні Apollo Server, поширеного open source — сервера для розробки з вико-

ристанням GraphQL, Playground стає доступним на тому ж endpoint, що і сервер (наприклад, localhost: 4000 / playground). І, як і в випадку з introspection query, він доступний тільки в дебаг-версії.

2.4 Робота з вебсокетами

WebSocket — це комп'ютерний протокол зв'язку, що забезпечує повнодуплексні канали зв'язку через єдине TCP-з'єднання. У GraphQL роль вебсокетів виконують Subscriptions.

Підписки — це функція GraphQL, що дозволяє серверу надсилати дані своїм клієнтам, коли відбувається певна подія. Підписки зазвичай реалізуються за допомогою WebSockets, де сервер підтримує стабільне з'єднання з клієнтом. Це означає, що під час роботи з підписками ви порушуєте цикл запитів-відповідей, який використовувався для всіх попередніх взаємодій з API. Тепер клієнт ініціює стійке з'єднання із сервером, вказуючи, в якій події він зацікавлений. Кожного разу, коли відбувається ця конкретна подія, сервер використовує з'єднання для передачі очікуваних даних клієнту [22].

При використанні клієнту Apollo, це потребує додаткового створення нативного з’єднання з сокетом і використати обгортку у WebSocketLink. Кінцевий варіант як це працює на рисунку 7.

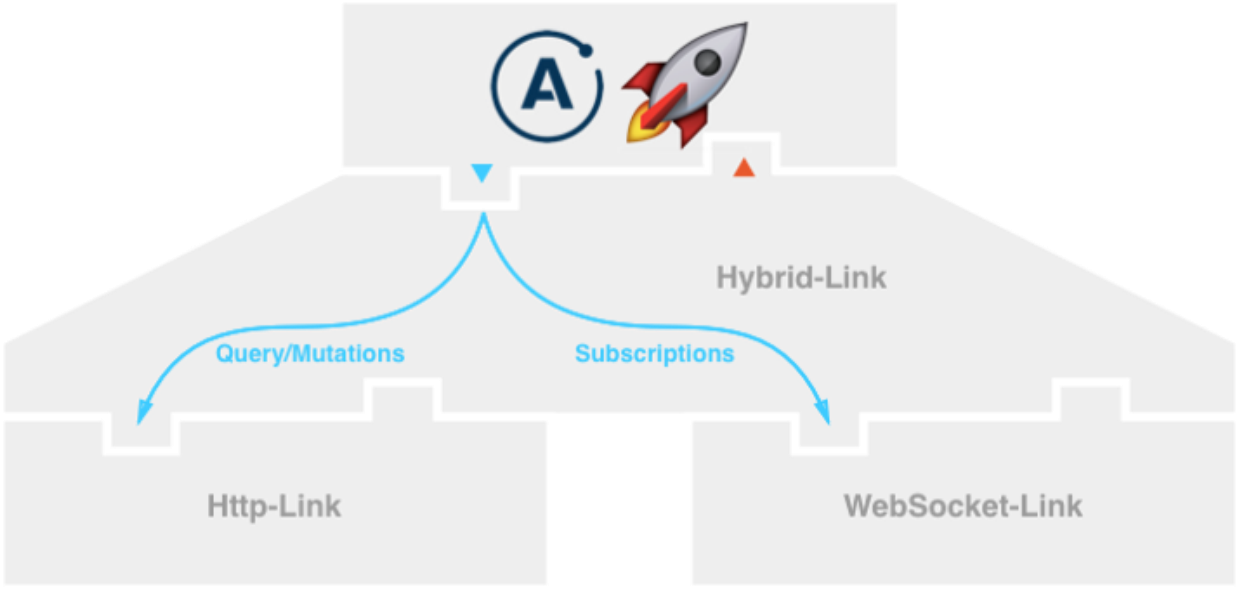

Рис. *7 Відмінність використання Query/Mutations і Subscriptions*

2.7 Експеримент порогу входження в технологію GraphQL

У цьому підрозділі описано контрольований експеримент проведений бразильськими дослідниками Gleison B., Marco T. V. на порівняння двох технологій впровадження веб-сервісів: REST та GraphQL [23]. Розкривається питання, яка технологія вимагає менших зусиль для впровадження запитів до веб-служб. Задаються наступні дослідницькі питання:

Питання 1: Скільки часу розробники витрачають на реалізацію запитів у REST та GraphQL? Насправді, щоб дати глибоке розуміння цього першого питання, досліджуєно три пов’язані питання:

1. Як цей час варіюється в залежності від типів запитів?
2. Як цей час змінюється серед студентів і аспірантів?
3. Як цей час змінюється залежно від досвіду учасників REST та GraphQL?

Експеримент проводився з використанням IDLE, що є простим IDE для програмування в Python.3. IDLE підходить для початківців, особливо в освітніх умовах. Як представлено на рисунку 8, учасникам було надано єдиний

файл вихідного коду, що містить опис завдань (у вигляді коментарів, див. Рядки 1–8) та конкретні змінні рядка для зберігання запитів (наприклад, рядок 13). Після виконання наданого коду він автоматично повідомляє, чи правильно виконаний запит чи ні [23].

Якщо це правильно, суб'єктам потрібно перейти до наступного завдання / запиту. В іншому випадку йому / їй було доручено переглянути та змінити імплементацію та спробувати ще раз. Крім того, кожне виконання формує журнал, що містить інформацію про запити (код, результат, час тощо). Цей журнал використано для обчислення часу, витраченого на кожне завдання і, яке названо T_i, для якого $1 \leq i \leq 8$. Журнал також передбачає час, коли кожне завдання було укладено (Fi).

Тому $Ti = F_i - F_i - 1$. Усі учасники експерименту розпочали одночасно, тобто F_0 відомо [23].

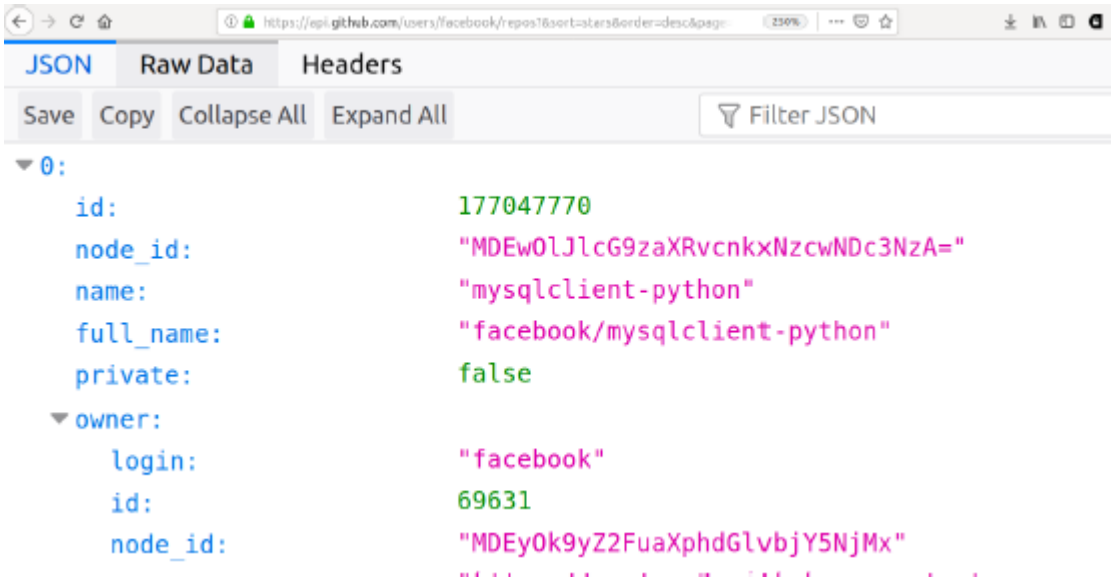

Рис. *8 Відповідь REST запиту з використанням веб-браузеру*

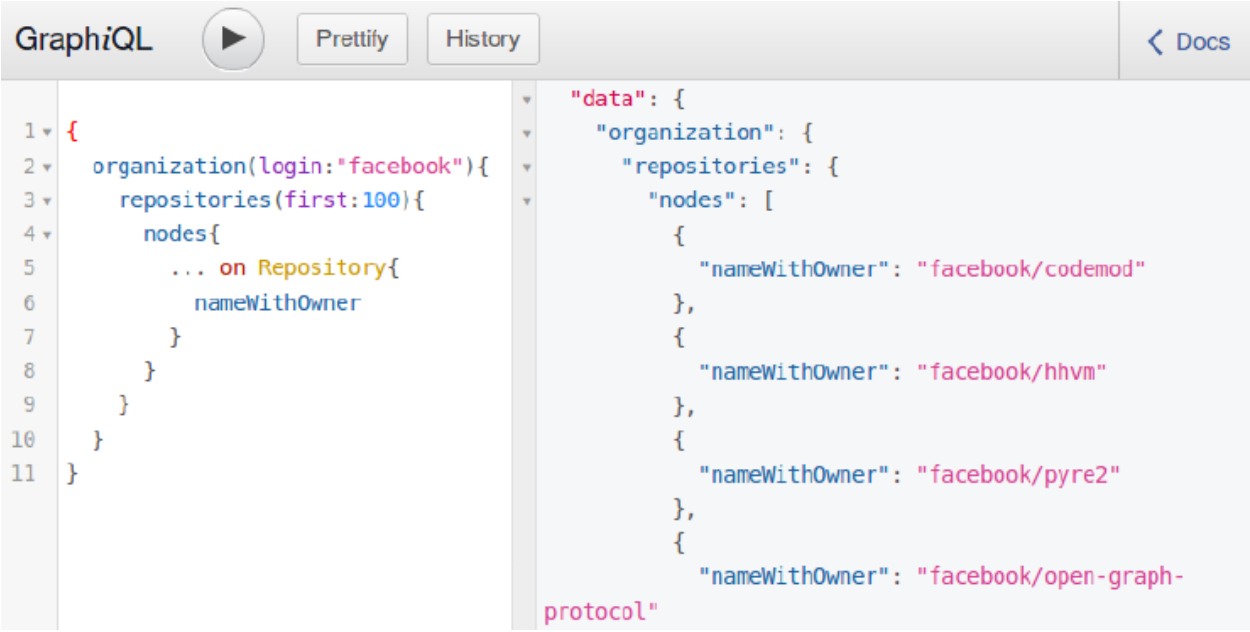

Рис. *9 Відповідь GraphQL запиту з використанням GraphQL*

GitHub надає веб-додаток під назвою GraphiQL для тестування запитів GraphQL (див. рис. 9). Цей додаток використовує функції GraphQL для підтримки, наприклад, автоматичного завершення. Стверджується, що якщо дозволити учасникам використовувати цю IDE – це не дає переваг GraphQL, оскільки його також використовують практикуючі в своєму щоденному досвіді роботи з мовою (просто для підкріплення, GraphiQL — офіційний додаток, що підтримується GitHub) [23].

Всі учасники уклали запропоновані завдання, тобто жоден учасник не покинув дослідження під час експерименту або не зміг реалізувати якийсь із запитів (див. рис. 10).

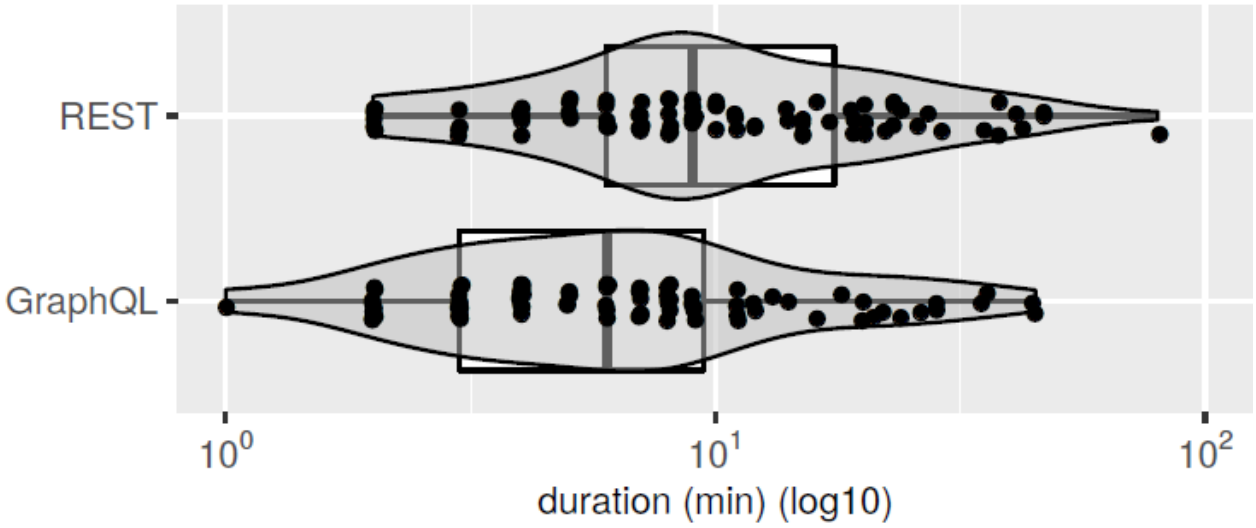

Рис. *10 Час на виконання завдань*

2.8 Логічна модель, як граф

Логічна модель даних, передбачена GraphQL, розглядає дані, які можуть бути представлені у графічній формі. Такий графік – це спрямований, позначений краями мультиграф у кожному вузлі має тип і властивості. Цей граф ми визначаємо, використовуючи домен (F, A, T). Потім кожен вузол на графі асоціюється з типом об’єкта T. Назва ребер, а також імена властивостей вузлів, складаються з імені поля F та набору аргументів, де таким аргументом є пара, що складається з окремого ім'я аргументу A та відповідне значення (зауважте, що набір аргументів може бути порожнім). Значення кожної властивості вузла є або одним скалярним значенням, або його послідовністю (див. рис. 11). Наступне визначення формально фіксує наше поняття графа GraphQL.

Визначення 1

GraphQL graphover(F,A,T) це кортеж G=(N,E,τ,λ) де:

– N набір вузлів,

– E набір ребер, які $(u, f[\alpha], v)$ де $u, v \in N, f \in F$, і α частка від A до Vals,

– τ : N → O_T функція визначення типу до кожного вузла

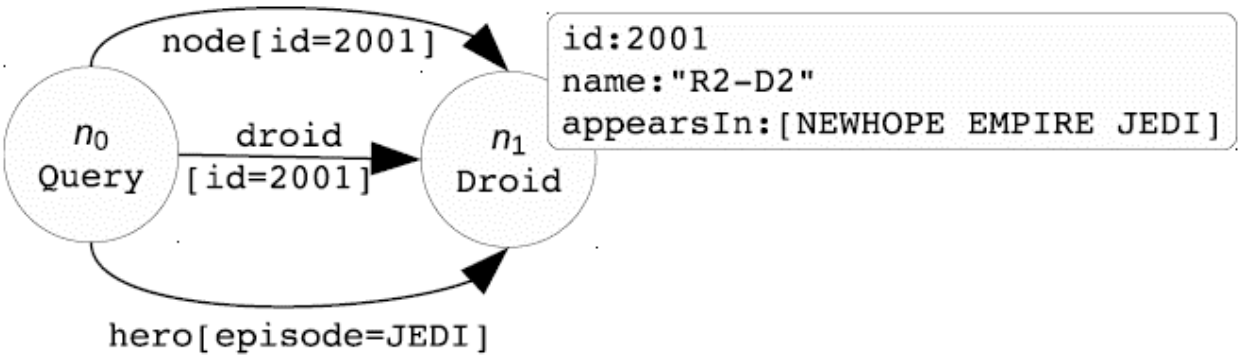

Рис. *11 Приклад GraphQL графа*

– ƛ часткова функція, що визначає $v \in Vals$ або послідовність $[v_1 \cdots v_n]$ скалярних значень, що $v_i \in Vals$ до деяких пар виду $(u, f[\alpha])$, де $u \in N, f \in F$, і часткове пов’язування A з Vals.

Вводиться поняття графа GraphQL, незалежного від будь-якої конкретної схеми GraphQL. Однак для цілі визначення запитів за таким графом передбачається, що граф відповідає заданій схемі. Умови, що відповідають схемі, накладеній на графік GraphQL, узагальнюються наступним чином.

Для кожного ребра, ім'я поля, яке позначає вузол, є серед імен полів, які схема визначає для типу вихідного вузла. Тип, який схема пов'язує з цим іменем поля, повинен відповідати типу цільового вузла, і якщо цей тип, пов'язаний з іменем поля, не є типом списку, то цільовий вузол – це єдиний вузол, підключений до вихідного вузла за допомогою вузла із заданим полем. Крім того, для кожного аргументу, асоційованого з ребром, ім'я аргументу повинно містити ім'я аргументу, яке схема асоціює з ім'ям поля, яке позначає вузол, а значення аргументу повинно мати тип, пов'язаний з ім’я аргументу [24].

На додаток до цих умов для міток вузлів існують подібні умови для властивостей вузла. Нарешті, граф повинен містити призначений вузол, тип якого є типом запиту схеми.

2.9 Математичне визначення мови запитів GraphQL

У цьому підрозділі наводиться офіційне визначення мови запитів GraphQL. Зокрема, спочатку визначається стислий синтаксис запитів GraphQL, який дуже нагадує синтаксис, подібний до JSON, введений у специфікації GraphQL. Після цього визначається формальна семантика цих запитів. Однак перед тим, як переходити до формальних визначень, надається деяка інтроспекція виразів, на основі яких можуть бути побудовані запити GraphQL і як ці вирази оцінюються графом GraphQL [24].

Найбільш базовою конструкцією в синтаксисі запитів GraphQL є вирази форми $f[\alpha]$. При обчисленні графа GraphQL такий вираз може бути використаний для збігу властивостей вузлів, ім'я яких має однакову форму. Тоді, припускаючи, що вартість властивості є скалярною величиною v або послідовність $[v_1 \cdots v_n]$ скалярних значень, то результатом оцінки є рядок форми $f: v$, або $f: [v_1 \cdots v_n]$. Альтернативна конструкція $f[\alpha]$ це $l: f[\alpha]$, що охоплює поняття "псевдоніми поля". Такі псевдоніми використовуються для перейменування назв полів, які надходять у запиті [24]. Тобто при використанні цієї альтернативної конструкції результатами будуть рядки форми $l: v$ або $l: [v_1 \cdots v_n]$.

Для узгодження ребер, може використовуватись форма $f[a]\{\varphi\}$, де φ це підзапит, що оцінюється в контексті цільових вузлів. Потім для випадку одного відповідного ребра результатом є рядок форми $f: \{\rho\}$, ρ – де рядок, отриманий в результаті оцінки підзапиту φ. З іншого боку, якщо кількість відповідних ребер може бути більше одиниці (що може бути, якщо тип, пов'язаний з полем f є типом списку), тоді рядок результату має форму $f: [\{\rho_1\} \cdots \{\rho_n\}]$. А вираз форми $f[\alpha]\{\varphi\}$ може мати префікс псевдоніма поля $l: f[\alpha]\{\varphi\}$.

Синтаксис запиту вводить ще дві конструкції: на $t\{\varphi\}$ та $\varphi_1 \cdots \varphi_n$. Хоча остання є просто переліком декількох підвиразів, результати яких мають бути об'єднані, перша фіксує поняття "умови типу", що дається тим, що специфікація GraphQL називає "вбудованим фрагментом". Отже, t є або типом об'єкта, і типом інтерфейсу, або типом об'єднання, а φ є підзапитом, який обчислюється

лише для нетермінальних вузлів, асоційований (об'єктний) тип сумісний з t (офіційне визначення цього поняття) [24].

2.10 Дослідження утиліт міграції REST до GraphQL

Вивчено роботу схем GraphQL. Хоча міграція не є цільовою задачею даної кваліфікаційної роботи, було цікаво побачити як GraphQL схема, побудована на основі існуючих провайдерів API виконує виклики до них. Хоча схему можна побудувати на стороні клієнта, краще перенести її на сторону сервера з міркувань продуктивності.

Є інструменти, які вбудовують API REST у GraphQL, але багато з них застосовують подібні підходи. Залежно від їхньої знайомості серед спільноти веб-розробників [25].

Загальна архітектура інструментів для обгортання REST показана на рисунку 12. Їх основна відмінність полягає в тому, як вони генерують схему GraphQL. Усі три інструменти реалізовані за допомогою Node.js та express для побудови сервера GraphQL. GraphiQL використовується як клієнтська програма на стороні споживача послуги. Як показано на рисунку 2.8.1 – споживач послуги (GraphiQL або клієнтська програма) надсилає свій запит на сервер GraphQL (1, 2 і 3). Потім сервер GraphQL отримує запит від клієнта (4). Для обробки цього запиту повинна бути сформована схема GraphQL під час виконання за допомогою одного з інструментів обгортання. Після генерування схеми graphql створює її примірник і сервер GraphQL викликає екземпляр згенерованої схеми (5,6). Після цього сервер будує виклики REST, використовуючи запит та роздільники полів схеми. Зауважте, що Base_URL постачальника послуг використовується для побудови роздільної здатності. Потім сервер надсилає REST-запити (наприклад, за допомогою HTTP) постачальнику послуг (7). Постачальник послуг обробляє REST-дзвінки та відправляє відповідь назад на сервер GraphQL (8). Після цього сервер GraphQL отримує відповідь від

постачальника послуг та налаштовує його відповідно до змісту запиту, отриманого проти згенерованого схемою [26]. Нарешті, індивідуальна відповідь відправляється назад клієнту (10). Відповідь, отримана клієнтом, може бути або даними, або повідомленням про помилку (11). Наприклад, повідомлення про помилку може бути отримане, якщо клієнт запитав неіснуючі поля, але клієнт добре поінформований (13) про помилку. Тоді як отримані дані мають аналогічний зразок із запитуваними даними, і клієнт може легко їх співставити (12) [27].

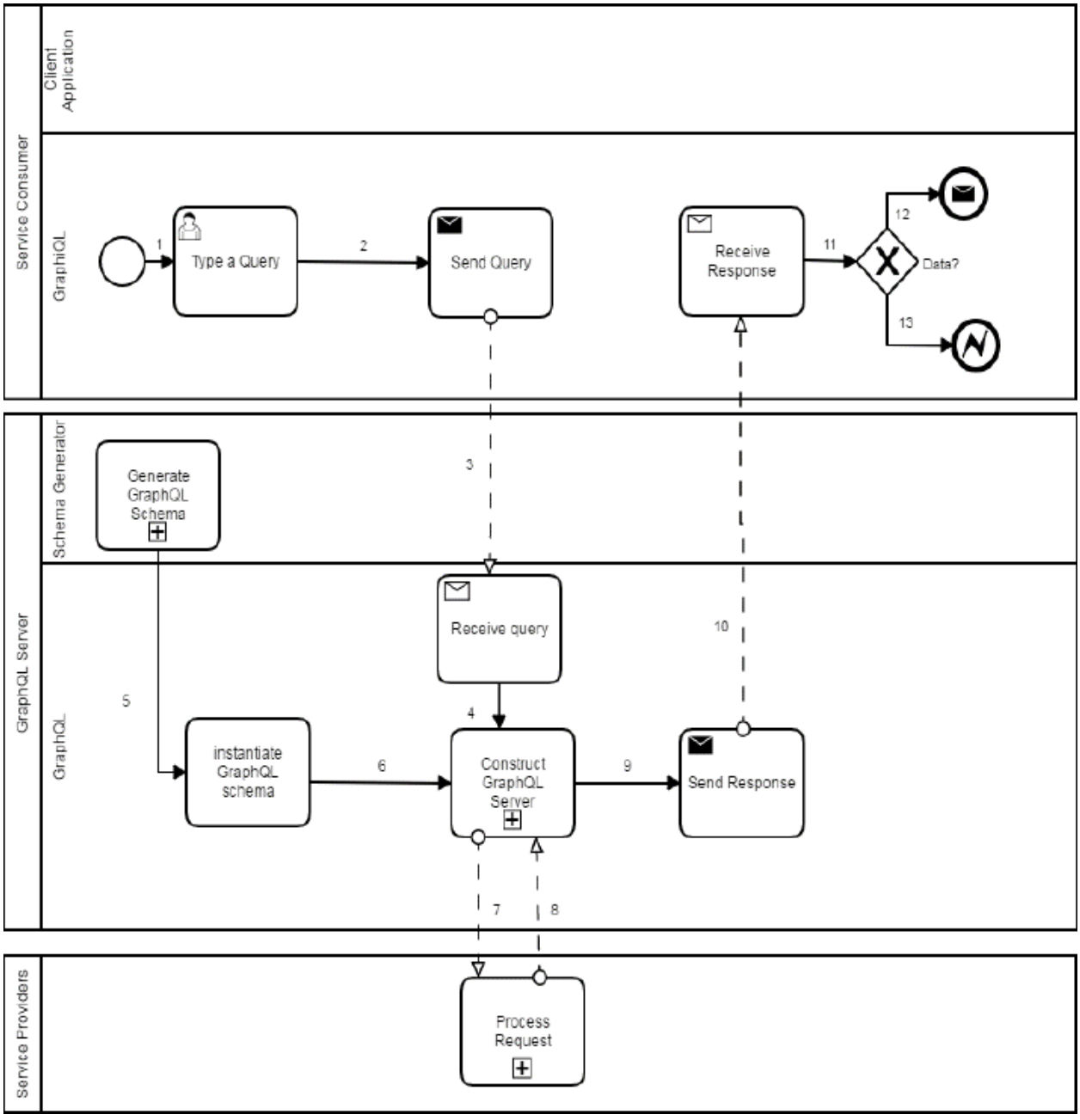

Рис. *12 Архітектура міграції*

2.11 Складність формування запитів

Запит GraphQL описує структуру даних відповідей, а також диктує функції обробника, до яких необхідно викликати його (які вирішувачі, в якому порядку та скільки разів). Пропонується дві метрики складності, призначені для вимірювання витрат з точки зору постачальника послуг GraphQL та клієнта: Складність вирішення відображає вартість виконання запиту сервера [28]. Складність типу відображає розмір даних, отриманих запитом. Постачальники послуг GraphQL отримають вигоду від будь-якого заходу, наприклад, залучивши їх для інформування про балансування навантаження, запобігання загрозам, розподіл ресурсів резолверів або встановлення цін на запит на основі вартості виконання або розміру відповіді. Клієнти GraphQL виграють від розуміння типової складності запиту, що може вплинути на їхні контракти зі службами GraphQL та мережевими провайдерами, або на політику кешування [29].

Метрики складності можна обчислити як для запиту, так і для його відповіді. Для запиту пропонується статичний аналіз для оцінки складності розв’язання та типу перед його виконанням з урахуванням мінімальних припущень на сервері GraphQL. Для відповіді розв'язок і складність типу визначаються аналогічно, але з точки зору полів та даних в об'єкті відповіді [30].

Інтуїція аналізу проста. Запит GraphQL описує розмір і форму відповіді. За допомогою відповідної формалізації семантики GraphQL — верхню межу складності розв’язання та складності типу можна обчислити, використовуючи зважені рекурсивні суми. Але якщо це не враховує типових практик проектування GraphQL, отримані обмеження можуть неправильно оцінювати складності [30].

У решті цього розділу описується два часто використовувані механізми пагінації GraphQL. Якщо схема та запит GraphQL використовують ці механізми, явно або неявно, ми можемо отримати більш жорстку і, таким чином, більш корисну межу складності. Дослідження повідомили, що обидві ці конвенції широко використовуються в реальних схемах GraphQL [30], тому підтримка їх також важлива для практичних цілей.

2.11.1 Конвенції пагінації GraphQL

У масштабі комерційних API-інтерфейсів GraphQL запитів до полів, що повертають списки об’єктів, можуть мати високу складність — наприклад, розглянемо (дуже великий) перехресний продукт усіх сховищ та користувачів GitHub. Офіційна документація GraphQL рекомендує розробникам схем обмежувати розміри відповідей за допомогою пагінації, використовуючи нарізування або шаблон з'єднань [27]. GraphQL не вказує семантику для таких аргументів, тому описуємо загальну конвенцію, за якою слідують комерційні [3, 6,29] та API з відкритим кодом [30] GraphQL.

Обробники можуть повертати списки об'єктів, що може призвести до довільно великих відповідей — обмежених лише розміром базових даних. Нарізка — це рішення, яке використовує обмежувальні аргументи для обмеження розміру повернутих списків (наприклад, products на рисунку 13).

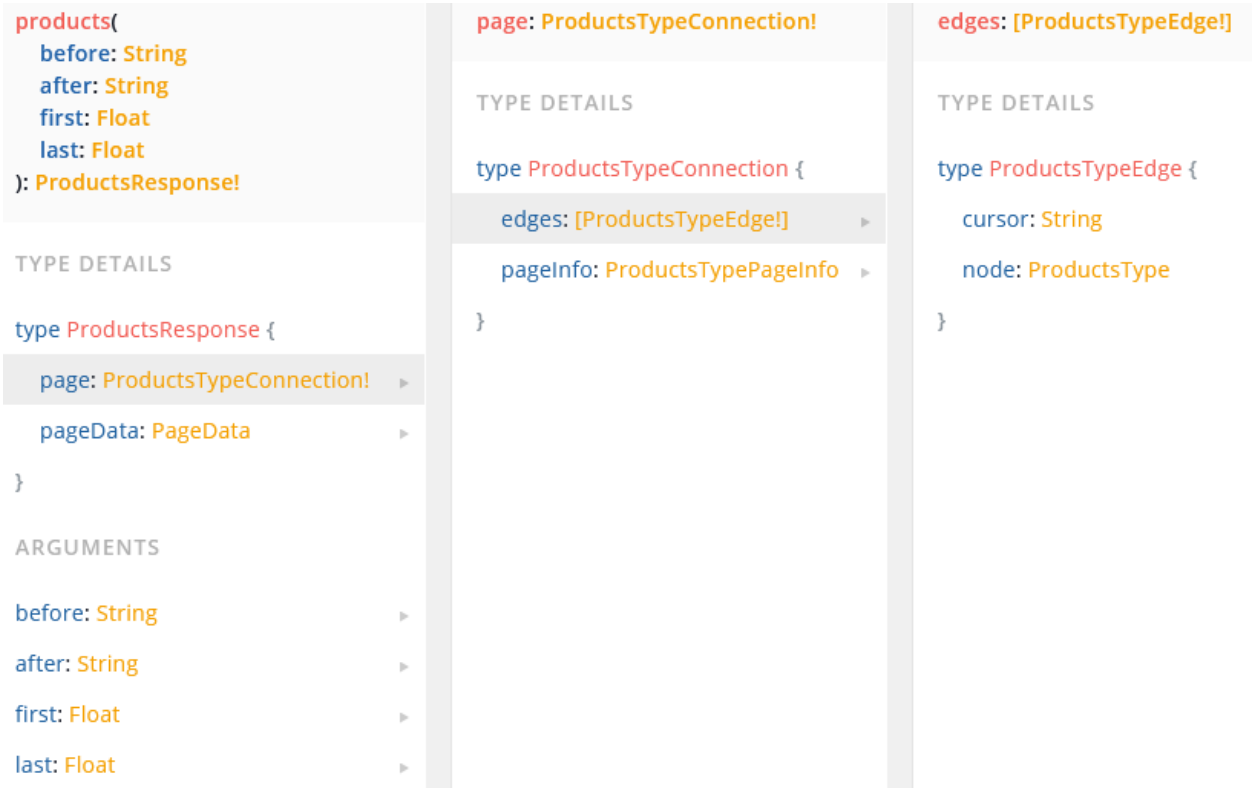

Рис. *13 Схема типу ProductsResponse*

Шаблон з'єднань вводить рівень опосередкованості для більш гнучкої пагінації з використанням віртуальних типів Edge і Connection [27, 15]. Наприклад, на рисунку 13 ліворуч поле page повертає єдиний ProductsTypeConnection, що дозволяє отримати доступ до загальної кількості ProductsType і поля країв, повертаючи список ProductsTypeEdge.

Розмір списку, що повертається розподільником, може, таким чином, залежати від поточних аргументів та аргументів батьків, що зберігаються в контексті. Забезпечення того, щоб обмежувальні аргументи насправді обмежували розмір поверненого списку, є відповідальністю розробників сервера [24]:

Пагінація — це не панацея. Незважаючи на те, що нарізка та шаблон з’єднань допомагають обмежити розмір відповіді запиту та кількість функцій вирішення, які викликає його виконання, ці шаблони не можуть заважати клієнтам формулювати складні запити, які можуть перевищувати обмеження швидкості користувача або перевантажувати серверні системи. Аналіз складності з урахуванням пагінації може статично ідентифікувати такі запити [30].

2.11.2 Конфігурація пагінації

Ігнорування аргументів нарізки або неправильна обробка шаблону з'єднань може призвести до недооцінки або завищеної оцінки вартості запиту. Розуміння семантики пагінації є, таким чином, важливим для точного статичного аналізу складності запиту. Оскільки пагінація GraphQL є домовленістю, а не специфікацією, пропонується доповнити схеми GraphQL конфігурацією, яка фіксує загальну семантику пагінації. Щоб уніфікувати цю конфігурацію з нашими визначеннями складності розв’язання та типу, включаються ваги, що представляють витрати вирішувача та типу [30]. Ось зразок конфігурації схеми на лістингу 16.

Лістинг *16 Зразок конфігурації схеми*

```
resolvers:
"Topic.relatedTopics":
limitArguments: [first]
defaultLimit: 10
resolverWeight: 1
"Topic.stargazers":
limitArguments: [first, last]
limitedFields: [edges, nodes]
defaultLimit: 10
resolverWeight: 1
```

```
types:
  Topic:
    typeWeight: 1
  Stargazer:
    typeWeight: 1
```

Ця конфігурація визначає поведінку пагінації для нарізки та шаблон з'єднань. У цій конфігурації розподільники ідентифікуються рядком "*type.field*" (наприклад, "*Topic.relatedTopics*"). Їх граничні аргументи визначаються полем *limitArguments*. Для нарізання аргумент limit застосовується безпосередньо до повернутого списку (див. "*Topic.relatedTopics*"). Для шаблону підключень, аргументи (обмеження) застосовуються до дочірніх елементів

повернутого об'єкта (див. обмежені поля для "*Topic.stargazers*") [30]. Поле *defaultLimit* вказує розмір списку, що повертається, якщо вирішувач викликається без обмежень аргументів.

Лаконічна конфігурація. Є інформація, що багато схем GraphQL відповідають узгодженим правилам іменування [30], тому вважається, що регулярні вирази та символи підстановки є природним способом зробити конфігурацію більш стислою. Наприклад, вираз "/.*Edge$/.nodes" може бути використаний для налаштування роздільників, пов'язаних з полями вузлів у всіх типах, імена яких закінчуються на Edge. Або вираз "User.*" Може бути використаний для налаштування роздільників для всіх полів типу User [30].

2.11.3 Аналіз вартості і складності запиту

У цьому підрозділі опис дослідження, проведеного дослідниками A. Cha, E. Wittern, G. BaudartA, J.C. Davis, L. Mandel, J. A. Laredo. У роботі формалізовано два аналізи запитів, щоб оцінити тип та вирішити складність запиту. Аналізи визначаються як індуктивні функції над структурою запиту, що відображає формалізацію семантики. Як і інші статичні аналізи вартості запитів, цей аналіз повертає верхні межі фактичних витрат на відповідь. Наприклад, якщо запит отримує 10 тем, цей аналіз поверне 10, як жорстку верхню межу розміру відповіді. Якщо на графіку даних є лише 3 теми, наш аналіз переоцінить фактичну вартість запиту [24].

Пропонується дві метрики складності: розв’язання складності та складність типу. Формалізується аналіз складності розв’язання, а потім пояснюється, як адаптувати підхід для обчислення складності типу.

Обчислюється оцінка складності розв’язання відповіді. Кожен виклик розподільника абстрагується відповідною вагою, а розміри списків обмежуються за допомогою функції обмеження.

Аналіз визначається індукцією структури запиту та відображає семантику, з однією основною відмінністю: аналіз складності працює виключно на типах, визначених у схемі, починаючи з типу запиту верхнього рівня [24].

Графіки складності укладання запитів по типу і розв'язувач сервісу Github (рис. 14, рис. 15) [24].

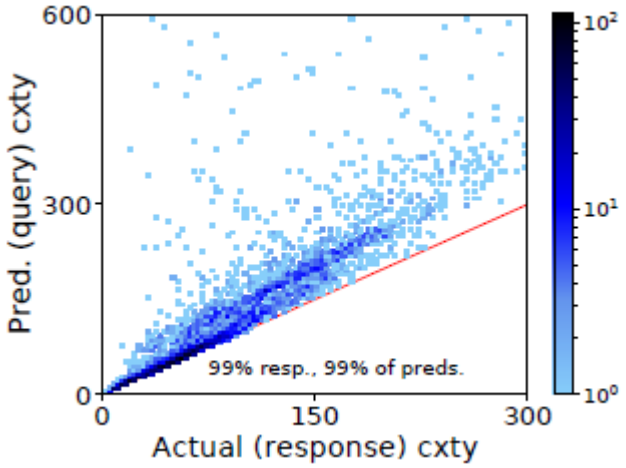

Рис. *14 Складність побудови типів Github*

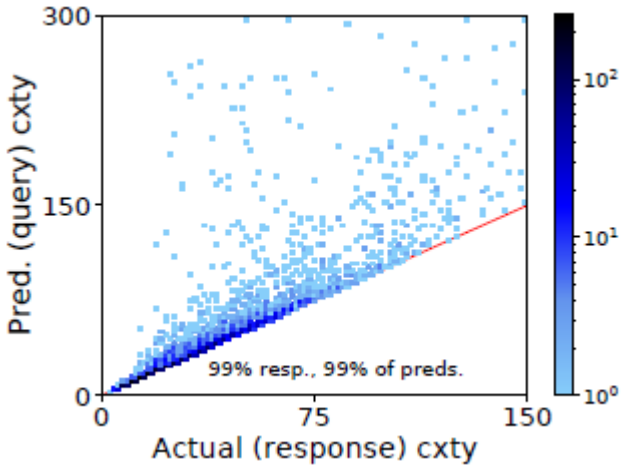

Рис. *15 Складність побудови розв'язувачів Github*

На кожному малюнку є текст, що вказує відсоток відображених відповідей (залишок перевищує вісь x) та відсоток відповідних передбачуваних прогнозів (залишок перевищує вісь y).

2.12 Швидкодія GraphQL

Продуктивність — ще одне питання, яке гостро обговорюється в контексті GraphQL проти REST. Послуги RESTful часто дискредитуються через схильність до недозавантаження та надмірного отримання.

Надмірне отримання відбувається, коли кінцева точка повертає зайві дані, ніж насправді потрібно. І навпаки, недозавантаження означає, що кінцева точка не може повернути достатню кількість необхідних даних [32].

Оскільки API REST за своєю суттю мають жорсткі структури даних, призначені для повернення обумовлених даних, коли вони потрапляють, ви можете отримати непотрібні дані або змусити зробити кілька дзвінків перед тим, як отримати відповідні дані. Ці недоліки також збільшують час, який потрібно серверу для повернення всієї запитуваної інформації [32].

Навпаки, GraphQL, замість використання фіксованих визначених сервером кінцевих точок, використовує гнучкий стиль, який дозволяє отримувати те, що ви хочете, в одному запиті API. Після визначення структури потрібної вам інформації ця сама структура буде повернута вам із сервера, що дозволяє уникнути недозавантаження та надмірного завантаження [32].

Хоча дискусія щодо продуктивності GraphQL проти REST може здатися на користь першої, є деякі випадки розробки, де RESTful API є кращим варіантом. Наприклад, у випадках, коли кешування потрібно для прискорення викликів API, REST API можуть працювати ефективніше [32].

API REST використовують вбудований механізм кешування HTTP, щоб швидше повертати кешовані відповіді. Поки GraphQL повинен повернутися до джерела, щоб отримати необхідні дані, REST може швидко отримати їх із браузера або мобільного кешу. Дійсно, GraphQL також має деякі варіанти кешування, але вони не досягли рівня REST.

Отже, в аналізі тесту продуктивності GraphQL проти REST переможець може бути не однозначним. Хоча GraphQL спрощує кількість запитів, необхідних середньому додатку, REST перевершує його завдяки надійності кешування.

2.13 Захищеність GraphQL

Що стосується безпеки GraphQL проти REST, довіра, схоже, схиляється до сторони останнього. REST пропонує декілька властивих способів забезпечити безпеку ваших API [32].

Наприклад, ви можете забезпечити безпеку REST API, застосувавши різні методи автентифікації API, наприклад, за допомогою аутентифікації HTTP, де конфіденційні дані надсилаються в заголовки HTTP, через веб-маркери JSON (JWT), де конфіденційні дані надсилаються як структури даних JSON, або за допомогою стандартних механізмів OAuth 2.0 [32].

Хоча GraphQL також пропонує деякі заходи для забезпечення безпеки ваших API, вони не настільки поширені серед розробників, як у REST. Наприклад, хоча GraphQL допомагає інтегрувати перевірку даних, користувачам залишається самостійно застосовувати заходи автентифікації та авторизації поверх запитів. Це часто призводить до непередбачуваних перевірок авторизації, що може поставити під загрозу безпеку програм на основі GraphQL [32].

Крім того, демонстраційний аналіз, який описано у підрозділі 2.13.1, виявив, що GraphQL ускладнює впровадження обмежувальних та інших заходів захисту від відмови в обслуговуванні, не має належної перевірки для самовизначених або спеціальних скалярних типів даних, а його функція самоаналізу може викрити внутрішні моделі даних [32].

2.13.1 Перевірки авторизації

Невідповідні перевірки авторизації. Розробникам API залишається самостійно застосовувати методи аутентифікації та авторизації. Гірше того, що "шари" резолверів, типові для API GraphQL, роблять це належним чином складнішим — перевірки авторизації повинні бути присутніми не тільки на розв’язувачах рівня запиту, але й для розв’язувачів, які завантажують додаткові дані (наприклад, щоб завантажити всі повідомлень для даного користувача) [33].

Існує 2 основні підходи авторизації у GraphQL. Перший, і більш поширений, виникає, коли функціональність авторизації обробляється безпосередньо розв’язувачами на рівні API GraphQL. Коли це зроблено, перевірки авторизації повинні виконуватися окремо в кожному місці, і будь-який випадок, коли це забуто, може призвести до зламу авторизації, який можна використати у своїх цілях. Це дозволяє зловмиснику виконати GraphQL, еквівалентний традиційній небезпечній атаці прямого посилання на об'єкт, і отримати будь-яку публікацію, яку вони хотіли б, загальнодоступну або приватну [33].

Документація GraphQL містить вказівки щодо безпечного виконання авторизації. Порада відносно проста — замість того, щоб виконувати логіку авторизації всередині функцій роз’язувача, вся логіка авторизації повинна виконуватися бізнес-логічним рівнем, що знаходиться в середині. Таким чином, усі необхідні перевірки авторизації можна виконувати в одному місці - це спрощує послідовне застосування обмежень [33].

2.14.2 Рекурсивні звертання та Rate Limit

Підвищена складність API GraphQL значно ускладнює реалізацію обмежень швидкості та інших засобів захисту від відмови в обслуговуванні. У той час як з REST API кожен HTTP-запит виконує рівно одну дію, запит GraphQL може виконувати довільно багато дій і, таким чином, виконувати довільно велику кількість ресурсів сервера. Через це ті самі стратегії обмеження швидкості, що використовуються для REST API — щоб просто обмежити кількість отриманих HTTP-запитів, як правило, не є достатніми для захисту API GraphQL [33].

Одним із загальних джерел запитів високої складності є графічна специфікація GraphQL. Якщо існує взаємозв'язок між двома типами об'єктів, зазвичай можна створити короткі запити, які швидко збільшують складність виконання. Наприклад, у нашому тестовому API користувач має набір повідом-

лень, який має автора, який має набір повідомлень, і так далі для скільки завгодно ітерацій. Невеликий, але дуже складний запит може бути сформований як такий (див. лістинг 17).

Лістинг *17 Рекурсивні звертання*

```
query Recurse {
  allUsers {
    posts {
      author {
        posts {
          author {
            posts {
              author {
                posts {
                  id
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Додавання додаткового шару рекурсії ще більше збільшує складність. Хоча цю проблему важко вирішити добре, як правило, існує дві стратегії, які використовуються для захисту від такого типу атак відмови в обслуговуванні. Простіше з них - просто встановити обмеження глибини рекурсії, щоб вкладені запити, що повертають такі масивні набори результатів, були відхилені. Однак це рішення ігнорує можливість дорогого запиту, який не вимагає великої глибини [33].

Інше популярне рішення, яке додає систему оцінки складності, вирішує ці проблеми. У системі оцінки складності кожній частині запиту присвоюється оцінка складності, і будь-який запит із сумарною складністю, що перевищує обране максимальне значення, буде відхилений. Наприклад, всім Користувачам може бути призначено 100 балів, тоді як полю дописів користувача може

бути призначено 10. У цій системі оцінки вкладених запитів будуть множитися, для комбінованого балу запитів 1000. У системі цього типу, точно призначаючи оцінки для кожного типу запиту важливі, але можуть бути складними завданнями для виконання на практиці [33].

2.14 Про Android і GraphQL

Використання GraphQL на Android не є неможливою задачею. Завдяки поширенню використання специфікації на мобільних пристроях було досліджено і проаналізовано алгоритм впровадження GraphQL на мобільні пристрої.

Взаємозв'язок між мобільним клієнтом та сервером можна побачити на рисунку 16.

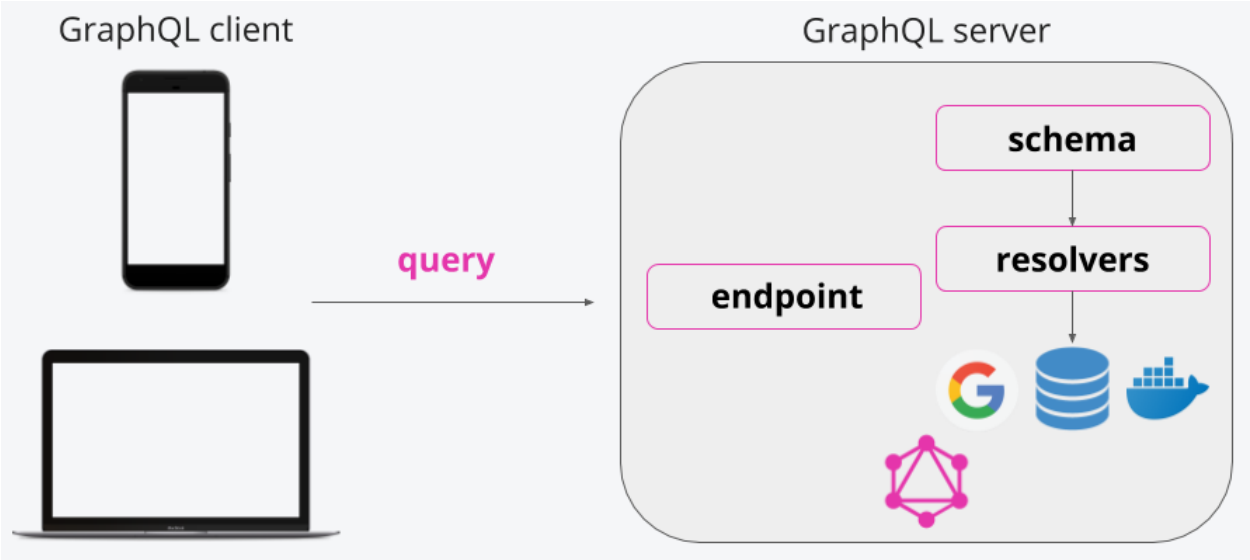

Рис. *16 Зв'язок між клієнтом на сервером*

Найбільш поширений сервіс для роботи з GraphQL — Apollo GraphQL Server. Це open source - проект, який активно розвивається Apollo GraphQL. Версії його існують для багатьох платформ, в тому числі для веб, iOS і Android. І саме про останній піде мова далі.

Apollo GraphQL клієнт для Android підтримує багато з основних функцій на даному етапі, в тому числі:

- queries, mutations, subscriptions;
- нормалізований кеш;
- завантаження файлів;

- власні скалярні типи.

В основі його роботи - кодогенерація сильно типізованих моделей за схемою GraphQL. За замовчуванням підтримується кодогенерація на Java, але можна в якості експериментальної можливості використовувати Kotlin-кодогенерацію.

Для полегшення роботи з GraphQL в Android Studio добре підходить плагін JS GraphQL. Основні його плюси:

- підсвічування синтаксису, автодоповнення, форматування коду;
- можливість виконання introspection query в Android Studio;
- підтримка виконання query, mutations в Android Studio та ін .;
- go-to-definition для GraphQL-файлів.

У build.gradle рівня додатку необхідно додати наступні рядки для роботи з Gradle-плагіном Apollo (див. лістинг 18).

Лістинг *18 build.gradle файл.*

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
       classpath("com.apollographql.apollo:apollo-
gradle-plugin:x.y.z")
    }
}
```

У в build.gradle рівнем модуля потрібно додати модуль (див. лістинг 19)

Лістинг *19 Модуль build.grade.*

```
apply plugin: 'com.apollographql.apollo'
repositories {
    jcenter()
```

```
    }
    dependencies {
        implementation("com.apollographql.apollo:apollo-
runtime:x.y.z")
    // If not already on your classpath, you might need the
 jetbrains annotations
        compileOnly("org.jetbrains:annotations:13.0")
        testCompileOnly("org.jetbrains:annotations:13.0")
    }
```

Далі виконуємо команду. */gradlew generateApolloSources*, щоб моделі за запитами з директорії src / main / graphql / com / my_package / згенерувались. При створенні моделей на основі GraphQL-типів будуть згенеровані Java-класи (або Kotlin-класи), типи полів класів відповідають скалярним типам GraphQL [34].

2.15 Висновки до розділу 2

На підставі отримання додаткових навичок роботи з REST та GraphQL можна зробити деякі висновки стосовно подальшого дослідження. А саме, технологія GraphQL дуже свіжа та доволі проста у виконанні для простих задач, але кожен раз коли проектуємо схеми, важно мати на увазі змогу орієнтуватись на зміни в базі даних. Нереляційна база, наприклад MongoDb має перевагу для роботи з GraphQL через синхронізовані моделі, у реляційних базах можна отримати схожий принцип роботи через наявність моделей.

Графова побудова схеми має на меті забезпечити максимальну стійкість системи до колізій. Структура графу побудована так, щоб не мати можливості отримувати вузли, не пов’язані з цією структурою.

Впровадження міграцій є трудомісткою роботою, яку порівняно складно охарактеризувати як гарне рішення, можливо так, для існуючих застосунків на

базі REST, але чим ширша API тим більше зусиль на вирішення конфліктів перекладу до типів GraphQL.

У кешуванні GraphQL програє через необхідність віддавати незакешовані сирі дані у вигляді графової схеми. У REST цієї проблеми не має через убудованість кешування у REST.

Обов'язковий перехід на GraphQL не має сенсу, все залежить тільки від бажання розробників. Але цей підхід дійсно спрощує роботу з великими різнорідними даними. Головне - пам'ятати про те, що GraphQL не припускає переробки всього того, що було раніше, і в підсумку кожен повинен вибирати той підхід, який йому ближче. На мою думку, GraphQL — це та технологія, з якою варто ознайомитися.

РОЗДІЛ 3 ОПИС ПРОГРАМНОЇ СИСТЕМИ

З метою дослідження використання технології GraphQL для створення програмного забезпечення для мобільних пристроїв в даній роботі розроблена тестова програмна система, яка складається з серверної і клієнтської частини.

Серверну частину представлено у вигляді хоста, який транслює GraphQL та REST запити. Створено сервіси, які використовують базу даних MongoDB, розгорнуту на віддаленому кластері серверів MongoDB Cloud. Сервіси спільні для використання REST'ом та GraphQL. Розроблено сервіс для впровадження пагінації на GraphQL.

Клієнтом є мобільний застосунок, на якому аналізується трафік та виконується обробка отриманих даних. Apollo Client робить десеріалізацію отриманої відповіді і поставляє до її відповідної моделі даних (див. рис. 17, рис. 18).

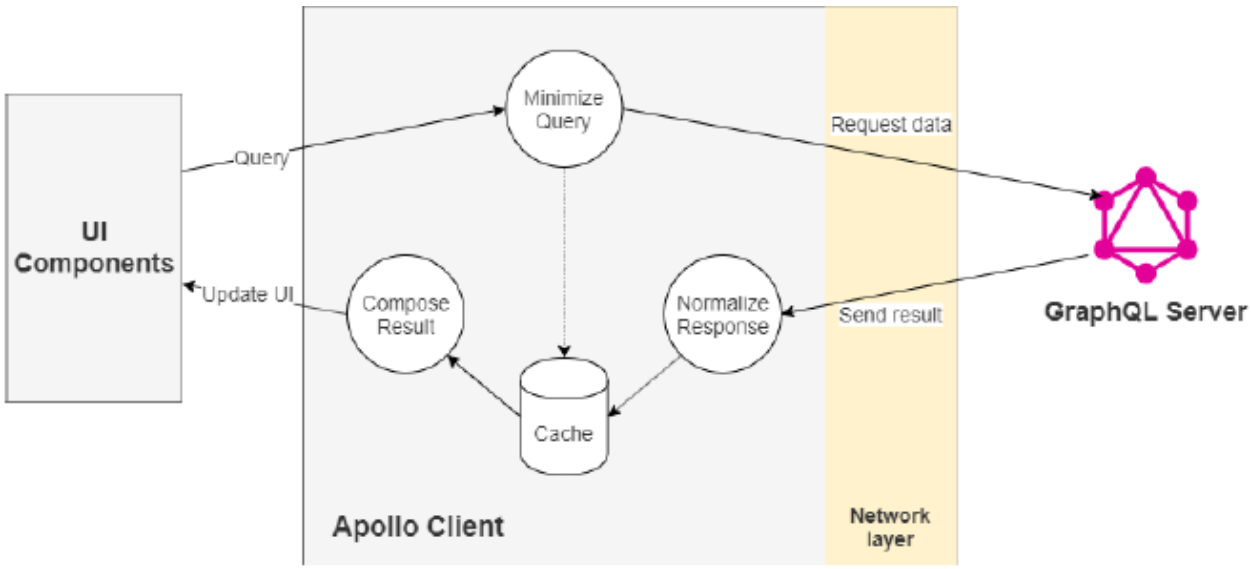

Рис. *17 Архітектура Apollo Client*

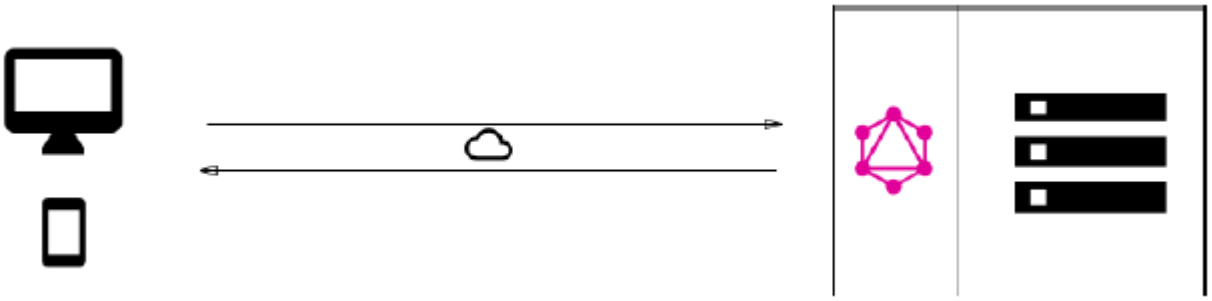

Рис. *18 GraphQL сервер з підключеною базою даних*

Використання технології ReactNative сводить діаграму роботи з сервером до схеми на рисунку 19. Для початку створення проекту потрібно визначитися з типом проекту. Є можливість створити його через Expo або React Native CLI.

Expo — інструмент розробника для створення проектів, перегляду журналів, відкриття на вашому пристрої, публікації тощо. Для роботи з Expo не потрібно мати XCode або Android Studio. Expo є надстройкою над React Native.

У React Native ми маємо можливість завжди повернутися до нативного коду. Це буває дуже важливо коли потрібно робити швидкі зміни і компілювати програму на своєму комп’ютері.

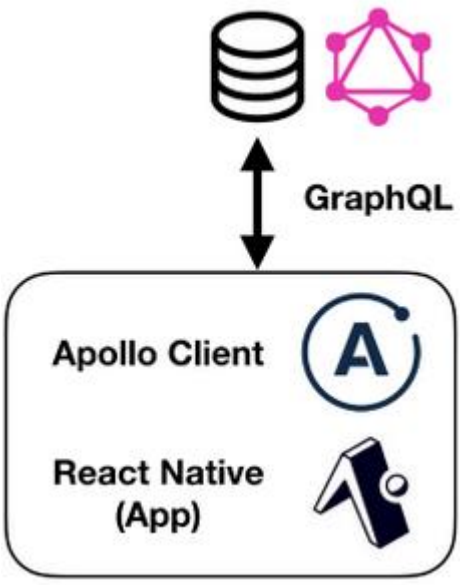

Рис. 19 GraphQL сервер і React Native

3.1 Архітектура бази даних

Архітектура бази даних складається з 11 нереляційних таблиць — колекцій. Деталі опису таблиць.

```
Categories
{
    "_id" : ObjectId,
    "productID" : NumberInt,
    "productName" : String,
    "supplierID" : NumberInt,
    "categoryID" : NumberInt,
    "quantityPerUnit" : String,
    "unitPrice" : Double,
    "unitsInStock" : NumberInt,
    "unitsOnOrder" : NumberInt,
    "reorderLevel" : NumberInt,
    "discontinued" : Boolean
}

Customers
{
    "_id" : ObjectId,
    "customerID" : String,
    "companyName" : String,
    "contactName" : String,
    "contactTitle" : String,
    "address" : String,
    "city" : String,
    "region" : null,
    "postalCode" : NumberInt,
    "country" : String,
    "phone" : String,
    "fax" : String,
}

Employee_Territories
{
    "_id" : ObjectId,
    "employeeID" : NumberInt,
    "territoryID" : String
}

Employees
{
    "_id" : ObjectId,
    "employeeID" : NumberInt,
```

```
    "lastName" : String,
    "firstName" : String,
    "title" : String,
    "titleOfCourtesy" : String,
    "birthDate" : ISODate,
    "hireDate" : ISODate,
    "address" : String,
    "city" : String,
    "region" : String,
    "postalCode" : NumberInt,
    "country" : String,
    "homePhone" : String,
    "extension" : NumberInt,
    "photo" : String
    "notes" : String,
    "reportsTo" : NumberInt,
    "photoPath" : String
}

Order_Details
{
    "_id" : ObjectId,
    "orderID" : NumberInt,
    "productID" : NumberInt,
    "unitPrice" : Double,
    "quantity" : NumberInt,
    "discount" : Double
}

Orders
{
    "_id" : ObjectId,
    "orderID" : NumberInt,
    "customerID" : String,
    "employeeID" : NumberInt(5),
    "orderDate" : ISODate,
    "requiredDate" : ISODate,
    "shippedDate" : ISODate,
    "shipVia" : NumberInt,
    "freight" : Double,
    "shipName" : String,
    "shipAddress" : String,
    "shipCity" : String,
    "shipRegion" : String,
    "shipPostalCode" : String,
    "shipCountry" : String
}

Products
```

```
{
    "_id" : ObjectId,
    "productID" : NumberInt,
    "productName" : String,
    "supplierID" : NumberInt,
    "categoryID" : NumberInt,
    "quantityPerUnit" : String,
    "unitPrice" : Double,
    "unitsInStock" : NumberInt,
    "unitsOnOrder" : NumberInt,
    "reorderLevel" : NumberInt,
    "discontinued" : Boolean
}

Regions
{
    "_id" : ObjectId,
    "regionID" : NumberInt,
    "regionDescription" : String
}

Shippers
{
    "_id" : ObjectId,
    "shipperID" : NumberInt,
    "companyName" : String,
    "phone" : String
}

Suppliers
{
    "_id" : ObjectId,
    "supplierID" : NumberInt,
    "companyName" : String,
    "contactName" : String,
    "contactTitle" : String,
    "address" : String,
    "city" : String,
    "region" : String,
    "postalCode" : String,
    "country" : String,
    "phone" : String,
    "fax" : String,
    "homePage" : String
}

Territories
{
    "_id" : ObjectId,
```

```
    "territoryID" : String,
    "territoryDescription" : String,
    "regionID" : NumberInt
}
```

Дані колекції зберігаються у сховищі колекцій серверу MongoDB.

Основні типи даних, це ObjectID — воно трансформується до типу ID у GraphQL, String, NumberInt, ISODate. Кожна з таблиць має з’єднання з залежними від них.

3.2 Засоби реалізації

3.2.1 Бібліотека Apollo Client

Apollo — це ціла екосистема, побудована розробниками як інфраструктура для додатків GraphQL. Бібліотеку можна використовувати на стороні клієнта для клієнтської програми GraphQL або на стороні сервера для сервера GraphQL [32].

Apollo Client використовується в даній роботі у комбінації з React Native. При виникненні помилок між запитами реалізовано обробник помилок від Apollo Client.

3.2.2 Бібліотека Mongoose

MongooseJS — це бібліотека, яка використовує картографічний документ об'єктів (ODM), який полегшує використання MongoDB, перекладаючи документи в базі даних MongoDB на об'єкти програми. Окрім MongooseJS, існує ще кілька ODM, які були розроблені для MongoDB, включаючи Doctrine, MongoLink та Mandango. Вибір пав на Mongoose через легкість і практичність у застосуванні.

Для побудування кожної моделі об’єктів потрібно було формувати імена полів, які повинні співпадати з іменами полів у документі бази даних. Кропітка робота і вимагає додаткових зусиль.

3.2.3 Технологія React Native

React Native — це фреймворк для розробки кроссплатформенних додатків для iOS і Android. Він з’явився на початку 2015 року, засновник – компанія Facebook, побудований на базі React, нативні компоненти мають зв'язку з Javascript і обернуті у React компоненти. Цей фреймворк надає можливість спростити розробку мобільного застосунку не переймаючись сильно конфігурацією та налаштуванням проекту. Все максимально автоматизоване, тому для дослідницької роботи підійде як найкраще, щоб не накоплювати увагу у проблемах самої мобільної розробки.

3.2.4 ExpressJS та Nest.JS

Для реалізації серверної частини потрібно побудувати сам сервер. Для кращої взаємодії з GraphQL було вибрано ExpressJS. Express JS — це фреймворк web-додатків для Node.js, реалізований як вільне і відкрите програмне забезпечення під ліцензією MIT. Він спроектований для створення веб-додатків і API. Щоб не ускладнювати реалізацію пишучи свої дескриптори та обробники відповідей краще використовувати комбінацію двох фреймворків — Express JS та Nest.JS.

Nest.JS представляє собою фреймворк для створення веб-застосунків Node.JS. Відмінною особливістю є те, що він вирішує проблему, яку не вирішує жоден інший фреймворк: структура проекту node.js. Якщо ви коли-небудь розробляли під node.js, ви знаєте, що можна багато чого зробити з допомогою одного модуля (наприклад, Express middleware може зробити все, від аутентифікації до валідації), що, в кінцевому підсумку, може призвести до важко підтримуваної "каші". Nest.js допомагає в цьому, надаючи класи, які спеціалізуються на різних проблемах.

Розробники Nest.js збирали натхнення з Angular. Наприклад, обидві платформи використовують guards для дозволу або запобігання доступу до деяких частин ваших додатків і обидві платформи надають інтерфейс CanActivate для реалізації цих guards. Тим не менш, важливо відзначити, що, незважаючи на

деякі подібні концепції, обидві структури незалежні один від одного. Тобто, можна легко реалізувати можливість розділення GraphQL і REST користуючись одним веб-сервером.

3.2.5 Візуалізація графу

Для візуалізації графу схеми GraphQL, було задіяно сервіс GraphQL Voyager. GraphQL Voyager представляє будь-який API GraphQL як інтерактивний візуальний графік (включаючи ці загальнодоступні API GraphQL). Він забезпечує швидку навігацію по графу, має ліву панель, яка надає більш детальну інформацію про кожен тип, опцію "пропустити Relay", яка спрощує графік, видаляючи класи обгортки Relay і можливість вибрати будь-який тип як корінь графіка Мається можливість захостити такий сервіс самому на окремому маршруті та приховувати деякі вузли для неавторизованих користувачів.

3.2.6 Тестування на GraphQL Playground

GraphQL Playground — це спосіб взаємодії з даними, які ваші джерела та плагіни додають як схеми. Коли потрібно багато взаємодіяти з цими даними, то Playground — саме те, що потрібно у вивченні цих даних. Запускається воно на маршруті *GET /graphql* і це дозволяє виконувати запити для тестування. Є можливість зберігати ці запити у подальше використання. Зберігаються вони у локальному сховищі браузера.

3.3 Модулі програмної системи

3.3.1 Серверна частина

При проектуванні серверної частини було приділено увагу принципу Database First. Спочатку було створено базу даних для дослідження, потім створено колекції на основі дампу *Northwind* реляційної бази даних.

База даних *Northwind* – це зразок бази даних, який спочатку був створений корпорацією Майкрософт і який протягом десятиліть використовувався як

основа для їхніх навчальних посібників у різних продуктах баз даних. База даних *Northwind* містить дані про продажі фіктивної компанії під назвою “Northwind Traders”, яка імпортує та експортує спеціальні продукти харчування з усього світу. База даних *Northwind* — це відмінна навчальна схема для ERP для малого бізнесу із замовниками, замовленнями, запасами, закупівлями, постачальниками, доставкою, працівниками та бухгалтерським обліком.

При створенні веб-сервера потрібно роз’яснити деякі проблеми, а саме. На якій архітектурі він буде запущений, яке навантаження планується. Для розробки веб-сервера підійшов фреймворк Nest.JS. NestJS — це відносно нове рішення в області бекенд-розробки, з великим набором функцій для швидкої побудови і розгортання корпоративних сервісів, які відповідають вимогам сучасних клієнтів додатків і дотримуються принципів SOLID і додатки дванадцяти факторів.

Дванадцятифакторний додаток — це методологія створення додатків програмного забезпечення як послуга, яка:

- Використовує декларативні формати для автоматизації налаштування, щоб мінімізувати час та витрати для нових розробників, які приєднуються до проекту;
- Має чітку взаємодію з базовою операційною системою, пропонуючи максимальну портативність між середовищами виконання;
- Придатна для розгортання на сучасних хмарних платформах, усуваючи потребу в адмініструванні серверів та систем;
- Мінімізує розбіжності між розробкою та виробництвом, забезпечуючи постійне розгортання для максимальної спритності;
- Може масштабуватися без суттєвих змін в оснащенні, архітектурі чи практиці розробки.

Дванадцятифакторну методологію можна застосовувати до програм, написаних будь-якою мовою програмування, і які використовують будь-яку комбінацію служб резервного копіювання (база даних, черга, кеш пам'яті тощо) [31].

«Принципи SOLID» є мнемонічною абревіатурою п'яти принципів проектування, спрямованих на те, щоб зробити програмне забезпечення більш зрозумілим, гнучким і ремонтопридатним.

Мовою розробки було вибрано TypeScript. TypeScript розширює JavaScript, додаючи типи. Розроблений компанією Microsoft, це строгий синтаксичний набір JavaScript і додає до мови необов’язкове статичне введення тексту. TypeScript призначений для розробки великих додатків та перекомпіляції в JavaScript. Окрім значних переваг строгої типізації проект має залежність використовувати Apollo Client, який підтримує TypeScript.

При старті серверу іде ініціалізація імпортованих модулів (див. лістинг 20). Ієрархічно пов’язаних між собою. Усі модулі які використовуються багаторазово визначені у підключенні через *forwardRef*.

Лістинг *20 Імпорт головного модуля серверу.*

```
@Module({
  imports: [
    ApiModule,
    RecipesModule,
    GraphQLModule.forRoot({
      debug: true,
      playground: true,
      installSubscriptionHandlers: true,
      autoSchemaFile: join(
        process.cwd(),
        'schema.gql',
      ),
      context: ({req}) => ({req})
    }),

MongooseModule.forRoot('mongodb://nwindport:nwindport@local
host:27017/Northwind'),
      // ThreeModule,
    ],
    controllers: [AppController],
    providers: [AppService],
})
export class AppModule {}
```

Кожен тип схеми повинен мати функції розв’язувачі для отримання, мутації даних. Приклад роз’язувача категорій серверної частини наочно відображено у лістингу 21.

Лістинг *21 Реалізація роз’язувача категорій.*

```
    import { Args, ArgsType, Field, ID, Int, Mutation,
Query, Resolver } from '@nestjs/graphql';
    import { CategoryService } from './category.service';
    import { CategoryResponse, CategoryType, UpCategory }
from './category.dto';
    import ConnectionArgs, { getPagingParameters } from
'~common/relay/connection.args';
    import { connectionFromArraySlice } from 'graphql-
relay';
    import { GraphQLBoolean } from 'graphql';

    // створення окремого класу для декоратору пошуку
    @ArgsType()
    class CategoryArgs extends ConnectionArgs {
      @Field(() => Int, {nullable: true, description: 'To
search by categoryID'})
      categoryID: number;
    }

    @Resolver()
    export class CategoryResolver {
      constructor(
        private readonly categoryService: CategoryService
      ) {}

      @Query(() => CategoryResponse, {description: 'List of
all categories'})
      async categories(@Args() args: CategoryArgs):
Promise<CategoryResponse> {
        const { limit, offset } =
getPagingParameters(args);
        const [data, count] = await
this.categoryService.findAll(limit, offset);

        const page = connectionFromArraySlice(
          data, args, { arrayLength: count, sliceStart:
offset || 0 },
        );

        return { page, pageData: { count, limit, offset }
};
```

```
    }

    @Query(() => CategoryType, {description: 'Category by
id'})
    async category(@Args('categoryID', {type: () => Int})
categoryID: number): Promise<any> {
      return await
this.categoryService.findOne({categoryID});
    }

    @Mutation(() => CategoryType, {nullable: true})
    async upCategory(@Args('data') data: UpCategory) {
      if (data.id) {
        return await this.categoryService.modify(data.id,
data);
      }
      return await this.categoryService.create(data);
    }

    @Mutation(() => GraphQLBoolean)
    async delCategory(@Args('id', {type: ()=> ID}) id:
string): Promise<any> {
      const doc = await this.categoryService.delete(id);
      return !!doc;
    }
  }
```

3.3.2 Клієнтська частина

Для клієнтської частини було побудовано застосунок на базі фреймворку React Native, складність створення була у конфігурації режиму «відладки», тому що для аналізу запитів немає стабільно працюючих застосунків.

Для отримання та відправлення запитів на GraphQL було створено об’єкт класу ApolloClient, і налаштовано з’єднання з локальним сервером. Проблема втрати з’єднання має місце бути, тому орієнтуючись на дослідження ми повинні мати стабільне з’єднання у локальній мережі Код налаштування можна побачити у лістингу 20.

Лістинг *22 Налаштування ApolloClient*

```
  const client = new ApolloClient({
    uri: 'http://192.168.88.5:3000/graphql',
```

```
    cache: new InMemoryCache(),
  });
```

Для отримання помилок створено окремий обробник помилок, він повідомляє помилки у консоль і у режимі «відладки» (див. лістинг 23).

Лістинг *23 Обробник помилок Apollo Client*

```
const errorLink = onError(({networkError}) => {
  if (networkError.statusCode === 401) {
    console.log(networkError);
  }
});
```

Через можливість створювати сторінки як компоненти у React Native, то основну логіку роботи з GraphQL поділено компоненти, що відповідають конкретній сторінці. Для використання мутацій було впроваджено lazyQuery від Apollo Client, щоб не визивати запит одразу від рендеру сторінки.

Відривок секції коду з компоненту категорій, який використовує GraphQL Apollo Client представлено у лістингу 24.

Лістинг *24 Секція списку категорій*

```
const PAGE_LIMIT = 10;

class CategoryList extends Component {
  state = {
    isLoading: true,
    responseData: [],
    hasNextPage: false,
    endCursor: null,
  };

  componentDidMount(): void {
    const {
      data: {loading, categories},
    } = this.props;
    if (categories) {
      const {hasNextPage, endCursor} =
categories.page.pageInfo;
```

```
            this.setState({
              isLoading: loading,
              responseData: categories.page.edges,
              hasNextPage,
              endCursor,
            });
          }
        }

        componentDidUpdate(
          prevProps: Readonly<P>,
          prevState: Readonly<S>,
          snapshot: SS,
        ): void {
          if (
            prevProps.data.loading &&
            !this.props.data.loading &&
            this.props.data.categories
          ) {
            const {
              data: {loading, categories},
            } = this.props;
            const {hasNextPage, endCursor} =
  categories.page.pageInfo;

            this.setState({
              isLoading: loading,
              responseData: categories.page.edges,
              hasNextPage,
              endCursor,
            });
          }
        }

        render() {
          const {
            data: {error, networkStatus, refetch, fetchMore},
            searchQuery,
            navigation: {navigate},
          } = this.props;
          const {responseData, hasNextPage, endCursor,
  isLoading} = this.state;
          console.log('render error', error);
          if (!error) {
            if (isLoading) {
              return <Text>fetching categories... </Text>;
            } else {
              return (
```

```
                <View>
                  <FlatList
                    data={responseData}
                    refreshing={networkStatus === 4}
                    onRefresh={async () => {
                      console.log('onRefresh', refetch);
                      const newFetchedData = await refetch();
                      const {
                        data: {
                          categories: {
                            page: {pageInfo, edges},
                          },
                        },
                      } = newFetchedData;
                      this.setState((prevState) => ({
                        responseData: [...edges],
                        hasNextPage: pageInfo.hasNextPage,
                        endCursor: pageInfo.endCursor,
                      }));
                    }}
                    onEndReachedThreshold={0.5}
                    onEndReached={async (info) => {
                      if (hasNextPage) {
                        const newFetchedData = await
fetchMore({
                          variables: {
                            limit: PAGE_LIMIT,
                            after: endCursor,
                          },
                        });
                        const {
                          data: {
                            categories: {
                              page: {pageInfo, edges},
                            },
                          },
                        } = newFetchedData;
                        this.setState((prevState) => ({
                          responseData:
[...prevState.responseData, ...edges],
                          hasNextPage: pageInfo.hasNextPage,
                          endCursor: pageInfo.endCursor,
                        }));
                      }
                    }}
                    keyExtractor={(item, index) =>
item.node.id}
                    renderItem={({item}) => (
```

```
                    <View style={{flex: 1, flexDirection:
'row', margin: 5}}>
                      <View style={{flex: 1}}>

<Text>{item.node.categoryName}</Text>
                      </View>
                      <View style={{flex: 1}}>

<Text>{item.node.description}</Text>
                      </View>

{/*<Text>{JSON.stringify(item)}</Text>*/}
                    </View>
                  )}
                />
              </View>
            );
          }
        } else {
          return <Text>Error Fetching categories</Text>;
        }
      }
    }

    export default graphql(queries.fetchCategories, {
      options: ({searchQuery}) => ({
        variables: {limit: PAGE_LIMIT, after: null},
      }), // compute query variable from prop
      notifyOnNetworkStatusChange: true,
    })(CategoryList);
```

Весь набір підготовлених запитів до GraphQL зберігається в окремому файлі. Виклик запитів реалізовано через посилання до об’єктів цього файлу. Вирізок змісту цього файлу представлено у лістингу 25.

Лістинг *25 Запит за деталями замовлення*

```
    const queries = {
    // All order details used in this app
    fetchOrderDetails: gql`
        query fetchOrderDetails($orderID: Int) {
          order_details(orderID: $orderID, first: 10) {
            page {
              edges {
                cursor
                node {
```

```
                ... on OrderDetailsType {
                  id
                  orderID
                  productID
                  unitPrice
                  quantity
                  discount
                }
              }
            }
          }
        }
      }
    `,
    …
}
```

Використання утиліт для інтроспекції дозволило зробити аналіз трафіку між клієнтським та серверним застосунком. Мається можливість робити секційний аналіз сегментів пакетів для отримання розміру відповіді, заголовків, тощо.

3.4 GraphQL обгортка пагінації типів

Для того, щоб не навантажувати клієнтський застосунок у даній роботі, використовується принцип побудування класу обгортки для впровадження пагінації. Пагінація надає можливість використовувати контрольні точки або маркери та лімітування отриманих даних. Окрім здатності порційно отримувати дані з бази, отримано можливість переносити цю логіку між різними серверними архітектурами. Можливість ходити за контрольними точками надається за допомогою формування унікальної стрічки закодованої на базі base64, на вхід подається унікальна локація курсору graphql. Приклад генерації обгортки представлено у лістингу 26.

Лістинг *26 Будування обгортки для пагінації GraphQL*

```
const meta = checkPagingSanity(args);
```

```
switch (meta.pagingType) {
  case 'forward': {
    return {
     limit: meta.first,
      offset: meta.after ? nextId(meta.after) : 0,
    };
  }
  case 'backward': {
    const { last, before } = meta;
    let limit = last;
    let offset = getId(before!) - last;

    if (offset < 0) {
      limit = Math.max(last + offset, 0);
      offset = 0;
    }

    return { offset, limit };
  }
  default:
    return {};
}
```

У лістингу 26 бачимо, як розраховується максимальна кількість елементів на сторінку на зсув. Зсув залежить від отриманого маркеру та вираховується за певним алгоритмом від залежної кількості елементів в колекції бази даних.

3.5 Вимоги до апаратної частини

Для розміщення серверної частини потрібно задовольняти наступним вимогам.

Мінімальні вимоги до апаратного забезпечення:

- Одноядерний процесор з частотою 2.0 GHz.
- Оперативна пам’ять ємністю не менше 512 Мб.

Рекомендовані вимоги до апаратного забезпечення:

- Двоядерний процесор з частотою 2.0 GHz.
- Оперативна пам’ять ємністю не менше 1024 Мб.

Для використання мобільного клієнтського застосунку потрібно задовольняти наступним вимогам.

Мінімальні вимоги до апаратного забезпечення:

- Наявність ОС Android 4 або вище.
- Наявність доступу до локальної або інтернет мережі для зв’язку з сервером.

3.6 Опис функціональних можливостей

Основна мета кваліфікаційної роботи є дослідження використання технології GraphQL для створення програмного забезпечення мобільних пристроїв, а не розробка кінцевого продукту, тому функціональні можливості системи було орієнтовано на проведення дослідження, а не в якості готової програмної системи. Застосунок поділено на 2 частини, які включають і не обмежуються використанням серверної частини, на якій впроваджено основну взаємодію, а саме сервер який надає обробку запитів GraphQL та REST, та мобільний застосунок на платформі React Native. Під час роботи мобільного застосунку встановлюється захищене з’єднання з сервером, всі налаштування доступні на етапі конфігурації проекту. Скомпільоване програмне забезпечення можливо використати для спілкування з віддаленим сервером. У результаті

аналізу запитів та відповідей формується підрахування розміру відповіді та фіксація часу відповіді.

3.7 Висновки до розділу 3

У результаті створених програмних застосунків можна побачити, що є необхідність у подальшому використанні комбінації GraphQL та мобільних застосунків.

1. Технологія дозволяє впровадити легкий інтерфейс для взаємодії з бекендом.
2. Строга типізація схем дозволяє уникнути помилок в майбутньому при зміні схем.
3. Створення обгорток для пагінації має свої недоліки в плані реалізації, але побудоване рішення у кваліфікаційній роботі може бути застосовано між будь-яких проектів на базі фреймворку NestJS.
4. Використання фреймворків, таких як NestJS полегшує роботу з серверною частиною при розділенні логічних компонентів між різними модулями.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ НАВАНТАЖЕННЯ ПРОГРАМНОЇ СИСТЕМИ

4.1 Порівняння запитів REST та GraphQL по кількості полів

Для виконання експерименту на розмір відповіді було впроваджено додаткові обчислення для побудування графіку.

Проведено експеримент для підрахунку кількості полів, що повертаються різними запитами. Результат можна побачити на рисунку 20.

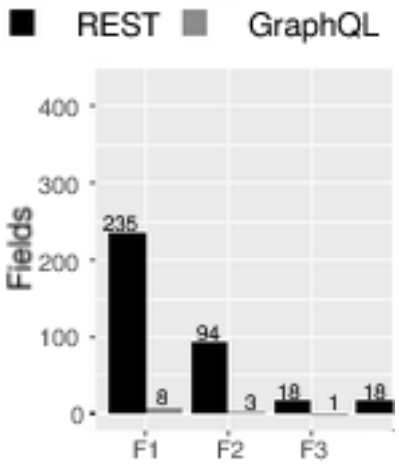

Рис. *20 Кількість полів, що повертаються на запити.*

Як можна помітити кількість полів, що повертаються на запитах REST значно вище ніж у GraphQL. Цей графік наочно демонструє недолік REST стосовно перевищення кількості отримуваних даних (Overfetching).

4.2 Аналіз розміру запитів та відповіді

Проведено додаткові заміри розміру відповіді у байтах. Згруповані дані отримані за допомогою аналізу розміру відповідей. Заміри використовувались на трьох таблицях, які можна побачити на рисунку 21.

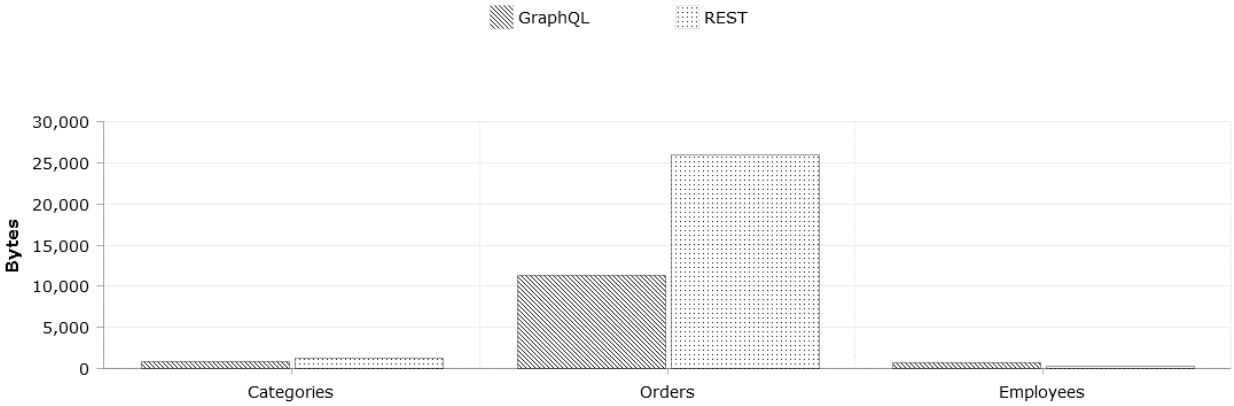

Рис. *21 Розмір відповіді у байтах для трьох колекцій*

Отримано додаткові заміри на розмір запиту з вимкненою компресією (див. рис. 22).

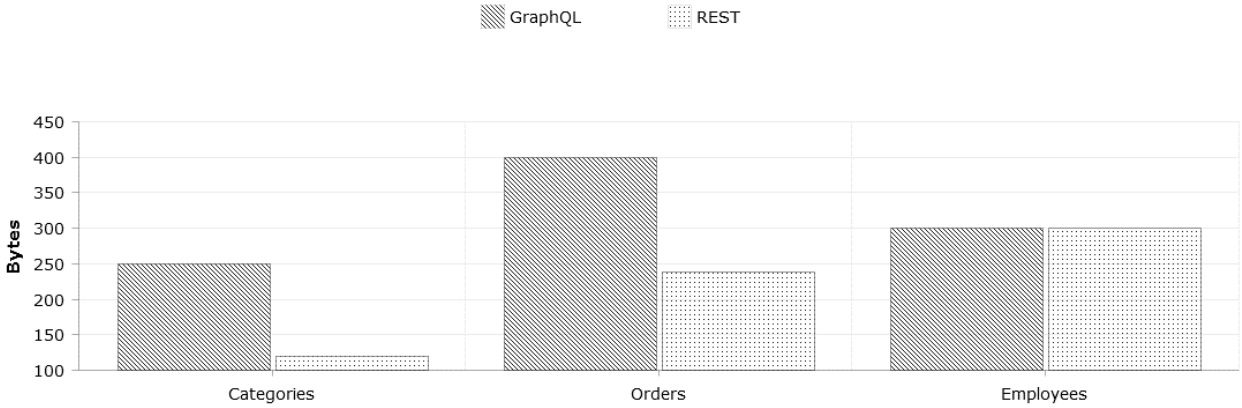

Рис. *22 Розмір запиту у байтах для трьох колекцій*.

В результаті порівняння можна зробити таке твердження, що GraphQL програє у розмірі запиту (для простих запитів, без багатої кількості параметрів), але це окупається невеликим розміром відповіді, майже як у 2 рази. Дані результати представлені при тестуванні маршрутів REST, які повертаюсь усі дані з колекцій. Більш детальний аналіз може бути зроблений за наявності поєднань у запитах сервісів REST.

4.3 Аналіз часу відгука

Дуже цікаво було проаналізувати час відповіді. Для тесту було створено 10000 даних на кожну колекцію в окремій базі. Отримано такі результати зважаючи на глибину вкладеності запиту, адже ми можемо отримувати інформацію між зв'язаними об'єктами (див. рис. 23).

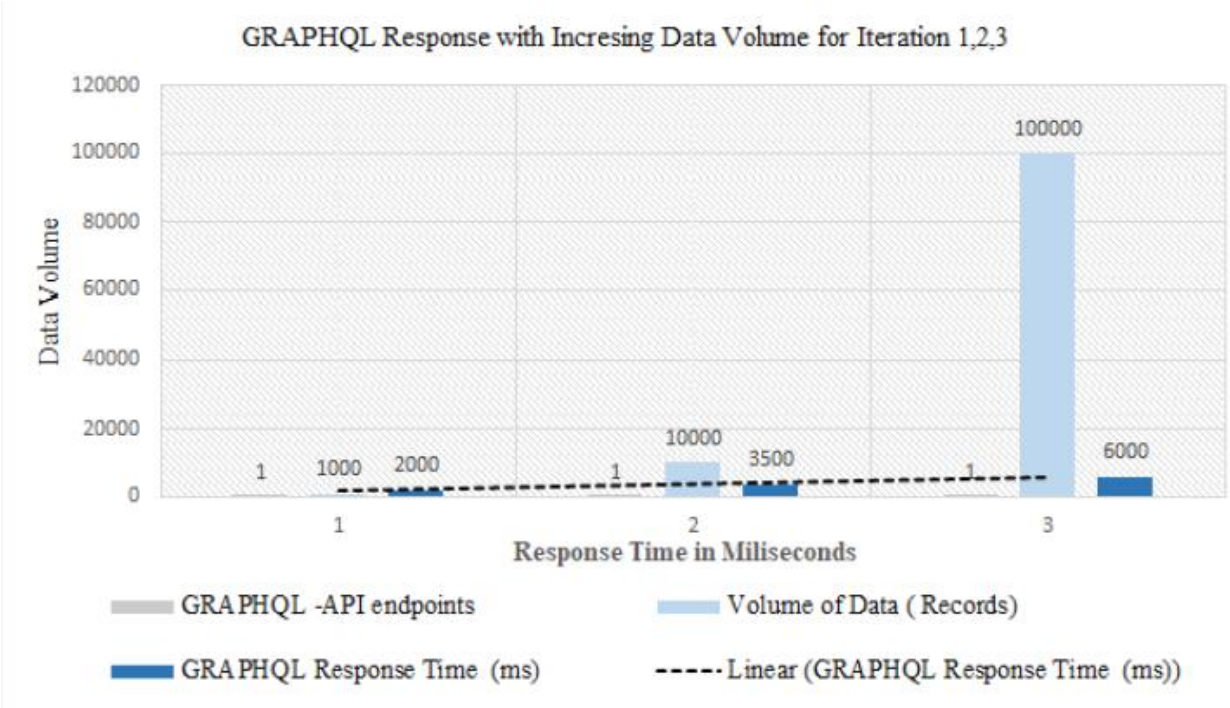

Рис. *23 Час відповіді згідно до розміру даних*

4.4 Висновки до розділу 4

Проаналізувавши складові відповідей до запитів можна виділити наступні ключові моменти:

1. GraphQL не справляється з великою вкладеність запитів.
2. Вкладеність між секторами схем може мати недоліки у обробці великої кількості даних.
3. Існує проблема REST при поверненні великої кількості даних.
4. Надається більше перевага використанню GraphQL для невеликих і нескладних систем.

На основі одержаних результатів можна зробити висновок, що використання GraphQL є оправданим для систем із гнучко спроектованою архітектурою.

GraphQL є новою моделлю веб-API. Його гнучкість може принести користь клієнтам, серверам та мережевим операторам. Але гнучкість також є загрозою: запити GraphQL можуть бути експоненціально складними, що може мати наслідки для постачальників послуг, включаючи обмеження тарифу та відмову в обслуговуванні. Основною вимогою до постачальників послуг є дешевий і практичний метод використання.

ВИСНОВКИ

У результаті виконання даної кваліфікаційної роботи було зроблено наступне:

1. Проведено аналіз сучасних технологій побудови Web API, зосередившись зокрема на REST і GraphQL.
2. Сформульовано вимоги, спроектовано і розроблено тестове середовище для дослідження цих технологій, яке складається з сервера, здатного обслуговувати запити як REST так і GraphQL і мобільного застосунку в якості клієнта.
3. На боці сервера реалізовано пагінацію, яка має надавати максимальну гнучкість та масштабованість.
4. Виконано аналіз трафіку, часу відгуку, розмірів запитів і відповідей для REST і GraphQL підходів.
5. Зроблено відповідні оцінки роботи технологій GraphQL та REST.

В якості зауважень до використання технології GraphQL як технології широкого використання можна назвати наступне:

Незважаючи на популярність цієї технології та активну рекламу вона все ще має нерозв’язані проблеми. Тому не можна вважати, що вона підійде при вирішенні будь-яких проблем у майбутньому. Ця технологія ще розвивається.

Так, в даній роботі при впровадженні у серверній частині GraphQL розв’язувачів було отримано проблеми з поверненням великої кількості даних. Існуючі варіанти пагінації можуть бути в graphql-relay пакеті, але через недостатню можливість оперувати секціями маршрутів було розроблено власний алгоритм пагінації для сервісів розв’язувачів GraphQL.

При побудові клієнтського застосунку значних проблем не виникало, але при відлагодженні програмної системи через нестабільність програм відладки інколи можуть виникати помилки в роботі аналізатора.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Тішин Олексій, студент магістратури ФЕЕІТ ІІ ЗНУ. Наук. кер.: к.ф.-м.н., доц. Попівщий В.І. «ВИКОРИСТАННЯ ТЕХНОЛОГІЇ GRAPHQL ДЛЯ СТВОРЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОБІЛЬНИХ ПРИСТРОЇВ». Збірник наукових праць студентів, аспірантів і молодих вчених «Молода наука-2020» : у 5 т. Запорізький національний університет. Запоріжжя: ЗНУ, 2020. Т.5. С. 169.
2. R. T. Fielding, R. N. Taylor, Principled design of the modern Web architecture, ACM Transactions on Internet Technology (TOIT), vol. 2, no. 2, 2002, pp. 115–150.
3. C. Jones, GraphQL with Oracle Database and node-oracledb, URL: https://blogs.oracle.com/opal/demo:-graphql-with-node-or acledb (дата звернення 12.10.2020).
4. R. Wieruch, The Road to GraphQL, 2018, 341 pp., https://www.academia.edu/40017073/The_road_to_graphql.
5. Магістерська робота (Професійна освіта. Комп’ютерні технології): Використання Neo4j і GraphQL на базі GraphDB для створення мікросервісу / Панкратова Н.Д. – Київ : 2018. – 268 с.
6. Игнатьев А. Ю. Обзор технологии GraphQL, Молодой ученый. – 2019. – №15. – С. 22-24. URL: https://moluch.ru/archive/253/57942/ (дата звернення 20.09.2020).
7. Principled design of the modern web architecture in 22nd International Conference on Software Engineering (ICSE), 2000, pp. 407–416.
8. R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, 2000.
9. M.A. Giroux, Production Ready GraphQL, 2020, p. 7.
10. R. Mizouni, M. A. Serhani, R. Dssouli, A. Benharref, and I. Taleb, Performance evaluation of mobile web services in 9th European Conference on Web Services (ECOWS), 2011, pp. 184–191.

11. O. Hartig, J. Pérez, An initial analysis of Facebook’s GraphQL language in 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web (AMW), 2017, pp. 1–10.
12. D.K. Barry, M. Kaufmann, Web services, service-oriented architectures, and cloud computing, 2003, pp. 54.
13. M. Papazoglou, Web services: principles and technology. Pearson Education. Petstore service provider based on swagger, 2008, pp. 68.
14. O. Fakorede, An investigation into the implementation issues and challenges of service oriented architecture, Bournemouth University, 2007, pp 82–90.
15. M. Pizzo, OData, OpenAPI Data providers. URL: https://www.odata.org/getting-started/understand-odata-in-6-steps/ (дата звернення 18.10.2020).
16. K. Patefield, OData or GraphQL The Best Tech for Developing an API Is Neither or Both!, URL: https://thenewstack.io/odata-or-graphql-the-best-tech-for-developing-an-api-is-neither-or-both/ (дата звернення 20.10.2020).
17. K. Ross, J. S. Schlipf, Thewell-founded semantics for general logic programs, ACM, 1991, pp. 620 – 650.
18. S. Stubailo, Порівняння REST та GraphQL, URL: https://habr.com/ru/post/335158/ (дата звернення 27.10.2020).
19. А. Бэнкс, Е. Порселло, GraphQL язык запросов для современных веб-приложений (Бестселлеры O’Reilly), 2019, 163 с.
20. А. Бэнкс, Е. Порселло, GraphQL язык запросов для современных веб-приложений (Бестселлеры O’Reilly), 2019, 60 с.
21. M. Ahieieva, Введение в GraphQL Subscriptions, URL: https://dou.ua/lenta/articles/working-with-qraphql/ (дата звернення 17.11.2020).
22. Realtime Updates with GraphQL Subscriptions, URL: https://www.howtographql.com/react-apollo/8-subscriptions/ (дата звернення 05.10.2020).

23. Gleison B., Marco T.V, REST vs GraphQL: Controlled Experiment, 2020, pp. 14-32.
24. A. Cha, E. Wittern, G. Baudart, J.C. Davis, L. Mandel, J. A. Laredo, Principled Approach to GraphQL Query Cost Analysis, 2020, pp. 38.
25. R. Krivtsov, Swagger to GraphQL, URL: https://github.com/yarax/swagger-to-graphql (дата звернення 26.10.2020).
26. G. Brito, T. Mombach, M. T. Valente, Migrating to GraphQL: A practical assessment in 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 140–150.
27. E. Wittern, A. Cha, and J. A. Laredo, Generating GraphQL-Wrappers for REST (-like) APIs," in International Conference on Web Engineering, 2018, pp. 65–83.
28. E.S. Ghebremicael, Transformation of REST API to GraphQL for OpenTOSCA, 2017 p.
29. Alain Armand, Graphql-REST-Wrapper 2016, URL: https://github.com/alonp99/graphql-rest-wrapper (дата звернення 21.10.2020).
30. GraphQL: The next generation of API design, URL: https://devblog.apollodata.com/graphql-the-next-generation-of-api-designf24b1689756a (дата звернення 24.10.2020).
31. A. Wiggins, The Twelve-Factor App, URL: https://12factor.net/ (дата звернення 25.10.2020).
32. A. Opidi, GraphQL vs. REST: A Comprehensive Comparison Subscriptions, URL: https://blog.api.rakuten.net/graphql-vs-rest/ (дата звернення 12.11.2020).
33. A. Noll, The 5 Most Common GraphQL Security Vulnerabilities, URL: https://carvesystems.com/news/the-5-most-common-graphql-security-vulnerabilities/ (дата звернення 18.11.2020).
34. R. Wieruch, The Road to GraphQ, 2018, p. 89.

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Тішин Олексій Борисович , студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти sp115-29@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Застосування технології GraphQL для створення програмного забезпечення мобільних пристроїв**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений/ознайомлена;

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 08.12.2019 Підпис _____ Тішин Олексій Борисович
(студент)

Дата 08.12.2020 Підпис _____ Попівщий Василій Іванович
(науковий керівник)