

АНОТАЦІЯ

Курсова робота присвячена дослідженню технології GraphQL та розробці тестового сервісу для демонстрації її можливостей і переваг над технологією REST API.

ЗМІСТ

ВСТУП.....	4
РОЗДІЛ 1	6
ОСНОВНІ ХАРАКТЕРИСТИКИ ТЕХНОЛОГІЇ	6
1.1. Загальна характеристика мови програмування JavaScript.....	6
1.2. JavaScript як серверна мова програмування. Node.JS	8
1.3. Загальна характеристика REST API та її призначення в сучасному програмному забезпеченні.....	9
1.4. Загальна характеристика GraphQL, його призначення у сучасному програмному забезпеченні.....	11
РОЗДІЛ 2	15
ВИЗНАЧЕННЯ ТЕХНОЛОГІЇ REST API	15
2.1. Основні поняття REST API.....	15
2.2. Семантика протоколу HTTP	19
РОЗДІЛ 3	23
ВИЗНАЧЕННЯ ТЕХНОЛОГІЇ GRAPHQL.....	23
3.1. Основні поняття GraphQL	23
3.2. Порівняльна характеристика REST технології з GraphQL, їх переваги та недоліки	26
РОЗДІЛ 4	29
РЕАЛІЗАЦІЯ ТЕХНОЛОГІЇ GRAPHQL ЯК АЛЬТЕРНАТИВА REST API. 29	
4.1. Постановка задачі	29
4.2. Реалізація поставленого завдання та демонстрація роботи додатку.....	29
ВИСНОВКИ	38
СПИСОК ДЖЕРЕЛ ТА ЛІТЕРАТУРИ	39
ДОДАТКИ.....	40

ВСТУП

У 2012 році всередині компанії Facebook розробники під час створення мобільного додатку Facebook для iOS та Android користувачів зіштовхнулись з новою для них проблемою. Проблема полягала в тому, що чим складнішими ставали їхні додатки, тим частіше вони виходили з ладу, також значно сповільнювалась робота самого додатку. Їм був потрібен достатньо потужний API для отримання інформації, який би дав змогу описати весь Facebook, але в той же час достатньо простий для вивчення та використання їхніми розробниками. Основною причиною, чому REST технологія не давала змоги вирішити їхні проблеми це те, що виникала занадто велика відмінність між даними, які вони хотіли використати в їхніх додатках, та запитам до сервера яких потребувала дана технологія. Крім цього виникали мережеві проблеми стосовно отримання даних такі, як: переповнення даними та недовантаження даних. Таке розчарування надихнуло трьох розробників цієї компанії Лі Байрона, Ніка Шрока та Дена Шафера на створення нового проекту, яким в решті і став GraphQL [1].

GraphQL був створений в 2012 році і протягом трьох років активно використовувався компанією Facebook, а вже в 2015 році відбувся його перший публічний реліз. Дана технологія була розроблена, як альтернатива REST API, та іншим спеціальним веб-сервісним архітектурам. Основними причинами, чому на даний момент багато компаній відмовляються від REST технології, є наступні [1]:

- REST API став складним в процесі розробки громіздких додатків;
- виникла потреба відокремити Front-End та Back-End для пришвидшення розробки додатку;
- у випадку наявності мобільного клієнта потрібно забезпечити швидкість відповіді клієнту на його запит;
- чіткість формування запиту з метою одержання клієнтом лише потрібної йому інформації.

В даний момент існує декілька різних реалізацій даної технології. Головними конкурентами в цій сфері є Facebook та Apollo [1].

РОЗДІЛ 1

ОСНОВНІ ХАРАКТЕРИСТИКИ ТЕХНОЛОГІЙ

1.1. Загальна характеристика мови програмування JavaScript

Javascript (JS) – це динамічна, об’єктно-зорієнтована, прототипна мова програмування, яка приносить інтерактивність веб-сайтам. Дана мова найчастіше використовується для створення сценаріїв веб-сторінок, що дає можливість на стороні клієнта взаємодіяти з користувачем. Також за допомогою javascript можна виконувати такі дії, як: керування браузером, асинхронний (паралельний) обмін інформацією з сервером, зміна структури та зовнішнього вигляду веб-сторінок. Класифікують javascript, як прототипну, скриптову мову програмування з динамічною типізацією. Також дана мова, окрім прототипної, підтримує також і інші парадигми програмування, а саме: імперативну та частково функціональну. Основними напрямками, в яких використовується javascript, є наступні [2]:

- написання сценаріїв веб-сторінок для надання їм інтерактивності;
- створення односторінкових веб-додатків;
- створення мобільних додатків;
- програмування на стороні сервера за допомогою Node.js.

На даний момент javascript є однією з найпопулярніших мов програмування в інтернеті. Це пояснюється зручним та багатим функціоналом, якими володіє дана мова, а саме [2]:

- об’єкти, з можливістю динамічної зміни типу завдяки механізму прототипів;
- функції, як об’єкти першого класу;
- обробка винятків;
- автоматичне приведення типів;
- автоматичне прибирання сміття;
- анонімні функції.

Основну гнучкість даній мові надають такі властивості, як: виринання, замикання, контекст(контекст виконання), та делегування подій.

Всі ці властивості перетворюють дану мову на потужний механізм для розробки веб-застосувань.

Для зручності javascript містить декілька вбудованих об'єктів. До них належать: Global, Object, Error, Function, Array, String, Boolean, Number, Math, Date, RegExp. Також дана мова містить набір вбудованих операцій, які не обов'язково є функціями або методами, а також набір вбудованих операторів, що керують логікою виконання програм [2].

Оскільки javascript надає програмісту досить великі можливості, то при розробці великих веб-додатків з використанням даної технології критично важливим є доступ до інструментів налагодження програми. Ця необхідність потрібна тому, що браузері від різних виробників, дещо відрізняються у поведінці javascript коду і реалізації DOM (Document Object Model) моделі. Сьогодні майже всі браузері підтримують інструменти налагодження програм, користуючись якими відстеження помилок при написанні коду значно спрощується. Також оскільки javascript є дуже популярною існують такі зручні інструменти, як [2]:

- 1) **ESLint** – перевірка якості коду. Користуючись даним інструментом можна здійснити сканування коду, та знайти деякі проблеми в ньому.
- 2) **Prettier** – даний інструмент дозволяє програмістам забути про погано відформатований нечитабельний код. Prettier зробить все форматування автоматично, потрібно лише надати ресурс з кодом.
- 3) **Babel** – це перекладач javascript коду до більш старих версій. Завдяки цьому застосуванню у розробників є можливість користуватись найновішим функціоналом мови, не турбуючись про те, що оточення не встигло реалізувати найновіший стандарт [2].

Оскільки js є інтерпретатором, без строгої типізації, то кожен блок сценарію написаний даною мовою інтерпретатор розбиває окремо. На веб-сторінках, коли

треба комбінувати блоки js та HTML(Hypertext Markup Language) коду, то синтаксичні помилки знайти легше, якщо тримати увесь javascript код не в одному файлі, розбити його на багато малих пов'язаних між собою js файлів. Користуючись таким способом синтаксична помилка не спричиняє падіння всієї сторінки, а також при такій структуризації коду будь-які помилки відстежуються набагато простіше [2].

1.2.JavaScript як серверна мова програмування. Node.JS

Node.js – це програмна платформа з відкритим кодом, для створення високопродуктивних мережевих додатків написаних мовою js. Дана технологія заснована на платформі V8, який був розроблений компанією GOOGLE для трансляції javascript-коду в машинний код [3].

Node.js додає javascript можливість взаємодіяти з пристроями вводу-виводу через власний API, підключати інші зовнішні бібліотеки, що були написані різними мовами програмування та забезпечувати звернення до них безпосередньо з js-коду. Найчастіше Node.js використовується для написання серверної частини додатку, виконуючи роль веб-сервера, але за допомогою даної технології можна створювати також і віконні додатки. [3]

Завдяки Node.js додаток легше масштабується. Це все досягається завдяки тому, що при одночасному підключенню великої кількості користувачів до серверу, Node.js працює асинхронно, тобто дана технологія розставляє пріоритети стосовно обробки запитів користувачів, завдяки чому ресурси розподіляються грамотніше. Така ідея розділу пріоритетів прийшла в голову розробнику Раяну Далу, оскільки виділення окремого потоку під кожне під'єднання було досить затратним відносно ресурсів. Був винайдений підхід, при якому використовується система, яка орієнтована на події. Інакше кажучи, дана система повинна реагувати на дію або бездіяльність користувача і виділяти під це ресурс. [3]

На даний момент дана технологія розвивається в дуже швидкому темпі, нараховуючи близько 200 000 пакетів. [3]

Ще однією з причин популярності Node.js є реалізація дуже зручного пакетного менеджера npm, завдяки якому (прописавши лише один рядок) можна довантажити всі необхідні для проекту пакети. Даний менеджер є настільки гнучким, що дозволяє керувати як пакетами, які є локальними залежностями певного проекту, так і глобально встановленими інструментами js. Всі залежності, що інсталювані для локального проекту, зберігаються у файлі package.json. У цьому файлі кожна залежність (dependencies) може визначити діапазон дійсних версій, використовуючи схему семантичної версії, що дозволяє уникнути небажаних змін, а також надає розробникам змогу автоматично оновлювати пакети. [3]

Отже, Node.js є хорошою альтернативою іншим серверним мовам програмування, оскільки асинхронне подієве оточення дає змогу зменшити затрати ресурсів на обслуговування великої кількості клієнтів. Також дана технологія дозволяє створювати масштабовані системи, завдяки тому, що жодна з функцій Node.js не працює напряду з потоками вводу-виводу. [3]

1.3. Загальна характеристика REST API та її призначення в сучасному програмному забезпеченні

REST API (Representational state transfer) – це підхід до архітектури мережеских протоколів, що здійснюють доступ до інформаційних ресурсів. Даний підхід був описаний Роєм Філдінгом (одним з творців протоколу HTTP) у 2000 році. В своїй дисертації “Архітектурні стилі та дизайн мережеских програмних архітектур” він визначив теоретичну основу, під спосіб взаємодії клієнтів та серверів у Всесвітній павутині. Філдінг описав концепцію побудови розподіленого додатка, при якій кожен запит клієнта до сервера містить в собі вичерпну інформацію щодо бажаної відповіді сервера, а сервер в свою чергу не зобов'язаний зберігати інформацію про стан клієнта. Користуючись даним архітектурним стилем, дані повинні передаватися у вигляді невеликої кількості стандартних форматів, таких як: JSON, XML, HTML та ін. Але як і будь-який архітектурний стиль REST API також має список власних обмежень. За

Філдінгом є п'ять обов'язкових обмежень для побудови розподілених REST - додатків [4]:

- a) модель клієнт-сервер;
- b) відсутність стану;
- c) кешування;
- d) однорідність інтерфейсу;
- e) шари абстракції;
- f) код на запит.

Останнє обмеження в списку не є обов'язковим, і якщо воно не надає переваг для конкретного застосування, то немає необхідності його реалізовувати. До прикладу, дозвіл на завантаження стороннього коду може бути небажаним з точки зору безпеки. Якщо ж не дотриматись хоча б одного з інших обмежень, то дану систему вже не можна вважати RESTful системою. В іншому випадку, якщо будуть дотримані всі обов'язкові обмеження, то система отримає ряд наступних переваг [4]:

- a) надійність (за рахунок відсутності необхідності зберігати інформацію про стан клієнта);
- b) продуктивність (завдяки кешуванню даних);
- c) простота уніфікованого інтерфейсу;
- d) портативність компонентів системи шляхом перенесення програмного коду разом з даними;
- e) простота внесення змін (навіть при працюючому додатку);
- f) прозорість системи взаємодій між компонентами системи для сервісних служб;
- g) масштабованість для великої кількості компонентів та взаємодій між компонентами;
- h) здатність еволюціонувати, пристосовуючись до нових вимог (на прикладі Всесвітньої павутини).

REST технологія набула популярності не лише завдяки вищевказаним можливостям, а ще й тому, що є дуже простим інтерфейсом управління інформацією без використання яких-небудь додаткових внутрішніх прошарків. Тобто дані не загортаються в XML, як це відбувається в SOAP та XML-RPC, а передаються в тому вигляді, в якому вони знаходяться [4].

Кожна одиниця інформації визначається глобальним ідентифікатором, таким як URL-адреса. В свою чергу кожна URL-адреса має суворо заданий формат. Наприклад, якщо потрібно взяти книгу, яка знаходиться на 3-ій позиції на книжній полиці, тоді ключ до неї, тобто її URL матиме наступний вигляд /book/3, а якщо потрібно відкрити дану книгу на 41 сторінці, тоді URL виглядатиме так /book/3/page/41. Завдяки уніфікованим глобальним ідентифікаторам доступ до даних здійснюється неймовірно просто. Причому абсолютно не важливо в якому форматі знаходяться дані за адресою /book/3/page/41 – це може бути як відсканована сторінка даної книги в форматі jpeg, так і текстовий документ Microsoft Word [4].

Управління інформацією сервісу цілком ґрунтується на протоколі передачі даних. Найбільш розповсюдженим протоколом звичайно ж є HTTP протокол. Для даного протоколу дії над даними здійснюється за допомогою наступних методів POST, GET, PUT, DELETE [4].

Отже, основною причиною чому технологія REST є такою популярною – це простота її використання. Адже по запиту, що надійшов на сервер можна одразу зрозуміти, що він робить. Також REST вважається менш ресурсо-затратним ніж вищезгадані SOAP та XML-RPC, оскільки не потрібно здійснювати попередній розбір запитів для того, щоб зрозуміти що саме вони повинні робити та не потрібно переводити дані з одного формату в інший [4].

1.4. Загальна характеристика GraphQL, його призначення у сучасному програмному забезпеченні

GraphQL - це мова запитів, яка призначена для побудови клієнтських додатків. Синтаксис даної технології інтуїтивно зрозумілий, та гнучкий у

розумінні опису своїх вимог до даних та їх взаємодій.

GraphQL має три основні характеристики, які використовуються клієнтом для завантаження даних з сервера [5]:

- GraphQL дозволяє клієнту чітко вказати, які дані йому потрібні;
- полегшує агрегацію даних взятих з декількох джерел;
- використовує систему типів для опису даних.

Основними причинами, чому дана технологія є хорошою альтернативою REST – це те що, використовуючи GraphQL, клієнтам більше не має потреби декілька разів звертатись до сервера за різними даними, оскільки можна отримати всю необхідну для клієнта інформацію за один запит. Також, використовуючи GraphQL, клієнт спілкується з сервером за допомогою унікальної мови запитів, яка в свою чергу відміняє необхідність для сервера жорстко задавати структуру або склад даних, що повертаються сервером, а також не прив'язує клієнта до конкретного сервера (рис.1). Фактично GraphQL знаходиться “в середині” між клієнтом і сервером. Оскільки клієнт запрошує в сервера дані, а той в свою чергу повинен відповісти на цей запит актуальними даними [5].

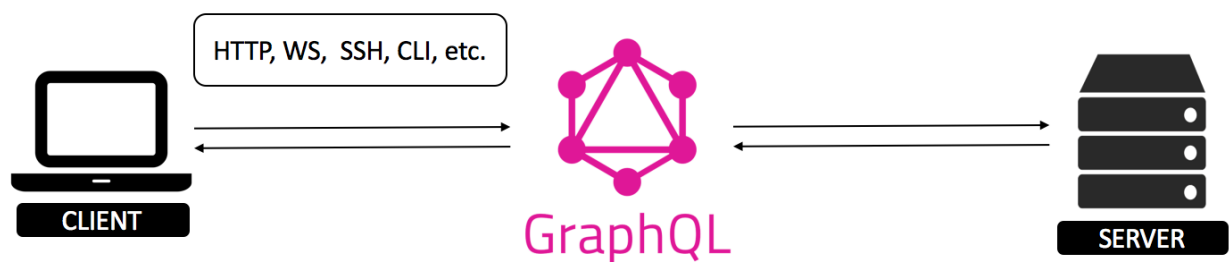


Рис.1. Клієнт-серверна архітектура з використанням GraphQL

Тобто GraphQL в цьому випадку виступає посередником між клієнтом і сервером або навіть особистим помічником клієнта стосовно запитів до сервера.

Саме в даному випадку у більшості виникає запитання, чому клієнт не може спілкуватися з сервером на пряму без посередників (рис.2). [5]

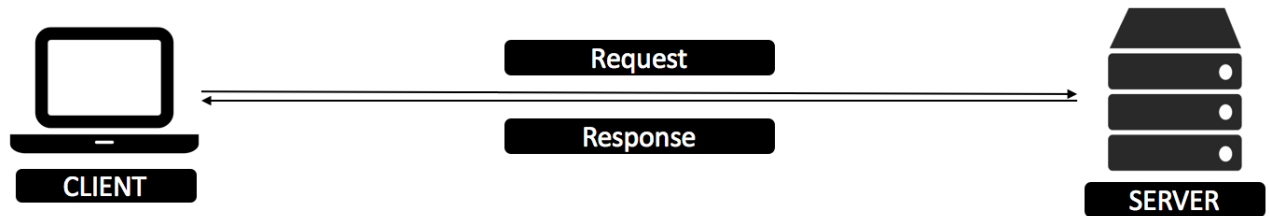


Рис.2. Клієнт-серверна архітектура з використанням REST API

Оскільки клієнт із сервера зазвичай потребує значної кількості інформації, а сервер може надати тільки інформацію згідно одного запиту за один раз. Саме тому клієнт повинен багаторазово звертатись до сервера, щоб отримати всі необхідні йому дані. Користуючись GraphQL такої проблеми не виникає, оскільки в цьому випадку він виступає особистим помічником клієнта. Наприклад, потрібно вирішити наступну задачу – “клієнту потрібно замовити піцу, доставку продуктів та забрати речі з хімчистки”. Користуючись старим REST подібним підходом клієнту довелося б зробити три окремі запити по кожній з позицій і це виглядало б наступним чином (рис.3).

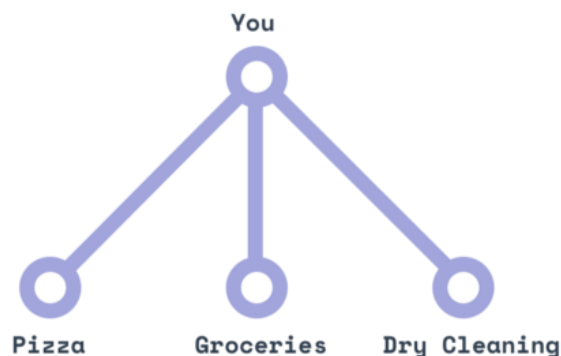


Рис.3. Принцип роботи запитів в REST API

При використанні, наприклад GraphQL, потрібно передати адреси місць, де знаходяться дані, а потім можна здійснити запит на одержання всіх даних одночасно. Даний процес чітко відображає рис.4.

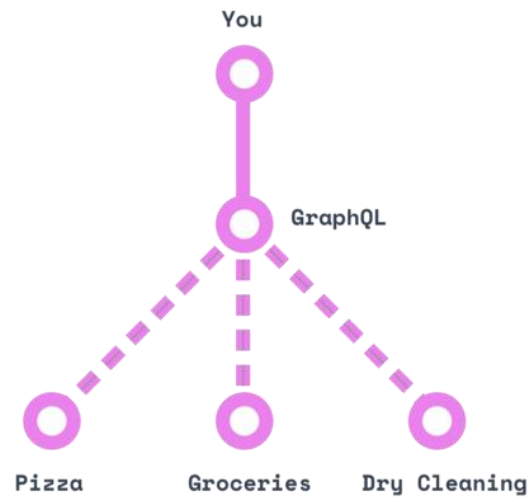


Рис.4. Принцип роботи запитів в GraphQL

Саме завдяки вище вказаним можливостям GraphQL витісняє REST підхід.

РОЗДІЛ 2

ВИЗНАЧЕННЯ ТЕХНОЛОГІЇ REST API

2.1. Основні поняття REST API

Як вже зазначалось, щоб додаток вважався RESTful-додатком, то в ньому повинні бути дотримані певні обмеження. Визначимо сутність кожного з цих обмежень [6].

Модель клієнт-сервер

Оскільки першою архітектурою, від якої REST успадковує обмеження, є клієнт-серверна архітектура, то основним принципом, що лежить в основі даного обмеження є розмежування потреб. Суть полягає у відмежуванні потреб інтерфейсу клієнта від потреб сервера, на якому зберігаються дані. Саме принцип розмежування підвищує переносимість клієнтського коду на інші платформи, в той час як спрощення серверної частини покращує масштабованість. Отже, розмежування дає змогу окремим частинам розвиватися незалежно одна від одної [6].

Відсутність стану

Принцип даного обмеження полягає в тому, що взаємодії між сервером і клієнтом не мають стану. Кожен запит містить в собі всю необхідну інформацію для його обробки і не покладається на те, що сервер знає щось з попереднього запиту. Поняття відсутності стану не означає, що його немає, а означає лише те, що сервер не знає про стан клієнта. Саме тому в проміжок часу між запитами немає жодної інформації про стан клієнта, що зберігається на сервері. Наприклад клієнт може завантажити сторінку сайту, тоді сервер обробить цей його запит і “забуде” про клієнта. Через декілька хвилин клієнт може натиснути на інше посилання і тоді сервер знову обробить його запит і знову “забуде” про клієнта. В цей час сервер може обробляти запити інших клієнтів, але для нашого клієнта це немає ніякого значення. Завдяки даному обмеженню дані про стан сесії зберігаються на стороні клієнта, і передаються з кожним запитом. Сервер в свою чергу після завершення обробки запиту може звільнити всі ресурси, що були задіяні для цієї операції, без жодного ризику втратити цінну інформацію. Спрощується моніторинг так, як для того щоб розібратись, що відбувається в певному запиті, достатньо лише подивитись на нього. Також збільшується

надійність, оскільки помилка в одному запиті не зачіпає інші. Окрім плюсів дане обмеження має великий мінус, який полягає в тому, що в кожен запит доводиться додавати дані сесії з клієнта, що значно знижує продуктивність. Також збереження стану на різних клієнтах складно підтримувати, бо реалізації клієнтів можуть відрізнятись, в той час як середовище сервера цілком контролює розробник [6].

Кешування

Це обмеження полягає в тому, що дані, які передаються сервером, повинні містити інформацію про те, чи можна їх кешувати і якщо можна, то як довго. Завдяки кешуванню збільшується продуктивність, оскільки відбувається уникання зайвих запитів до сервера, але це в свою чергу зменшує надійність того, що дані в кеші можуть бути застарілими [6].

Однорідність інтерфейсу

Наявність уніфікованого інтерфейсу є фундаментальною вимогою REST-сервісів. Дане обмеження дозволяє кожному з сервісів розвиватися незалежно, саме тому їх легко модифікувати при потребі. Також на однорідні інтерфейси накладається ще чотири обмеження [6]:

- Ідентифікація ресурсів.

Це означає, що всі ресурси ідентифікуються в запитах. Наприклад, з використанням URL в інтернет-системах.

- Маніпуляція ресурсами через представлення.

Суть полягає в тому, що якщо клієнт зберігає представлення ресурсу з метаданими включно, то він має достатньо інформації для того, щоб модифікувати або видалити ресурс.

- Самоописові повідомлення.

Кожне повідомлення містить достатньо інформації, щоб зрозуміти яким чином його можна обробити.

- Гіпермедіа як засіб зміни стану додатку.

Клієнти змінюють стан системи тільки через дії, які динамічно визначені в гіпермедіа на сервері. Самостійно клієнт не може визначити, що доступна якась

операція над якимось ресурсом, якщо він не отримав інформацію про це в попередніх запитах до сервера [6].

Шари абстракції

Зазвичай клієнт не може визначити чи взаємодіє він напряму з сервером, чи з проміжним вузлом у зв'язку з ієрархічною структурою мереж. Мається на увазі, що така структура утворює шари. Кожен компонент потрапляє в якийсь шар і спілкується лише з компонентами в шарі над ним або в шарі під ним. Дане обмеження покращує масштабованість за рахунок збалансованого навантаження та розподіленого кешування. Обмеження знання системи одним шаром зменшує складність компонентів [6].

Запитування коду

Це обмеження дозволяє розширити функціональність клієнта за рахунок завантаження коду з серверу у вигляді аплетів або сценаріїв. Філдінг стверджує, що дане обмеження не є обов'язковим, але дотримуючись його, можна спроектувати архітектуру, що підтримує бажану функціональність [6].

Оскільки REST є архітектурним стилем, то він має власні архітектурні елементи [6].

Одним з цих понять є елементи даних. Компоненти REST системи спілкуються, передаючи один одному представлення ресурсу в форматі, що вибирається з набору стандартних форматів даних. Даний формат обирається динамічно відповідно до бажань клієнта та можливостей сервера [6].

Ключовим елементом даних в REST є ресурс. Ресурс може бути представлений чим завгодно, це може бути текстовий файл, зображення, якесь динамічне значення, навіть щось з реального світу [6].

Для того, щоб мати змогу посилатись на ресурси, існує поняття ідентифікатора ресурсів. Компонент, що надав ресурсу ідентифікатор та дозволяє доступитись до даного ресурсу за даним ідентифікатором, відповідає за збереження незмінного стану функції приналежності. Якщо подивитись на ресурс з іншого боку, то він і є цією функцією приналежності, що відображає

моменти в часі на множину однотипних сутностей [6].

Ще одним елементом даних є представлення.

Представлення – це послідовність байтів, та метадані представлення, які потрібні для опису цих байтів. Часто представлення знаходяться у вигляді документів, файлів, HTTP-повідомлень тощо. Також досить часто трапляється така ситуація, що метадані є не лише в представлення ресурсу, а й в самого ресурсу. Прикладом метаданих ресурсу може бути посилання на джерело в html тезі [6].

Контрольні дані – це дані, які описують ціль повідомлення між компонентами в представленні. Наприклад, це може бути запит щодо виконання дії такої як створення, зміна або видалення ресурсу. Також контрольні дані можуть виступати, як значення відповіді, тобто можуть містити інформацію про стан ресурсу, або ж описувати помилку. Ті контрольні дані, які включені в запити чи відповіді, можуть керувати поведінкою кеша [6].

Також в елементах даних існує таке поняття, як медіа-типи. Медіа-тип – це формат даних представлення. Одні медіа-типи краще підходять для автоматичної обробки, інші – для того, щоб відображатись користувачеві. Для того щоб поєднати кілька видів представлення в одному, можна скористатись композитними медіа типами. Від формату даних залежить прихованість застосунку, яку сприймає безпосередньо користувач. До прикладу браузер може почати відображення веб-сторінки, ще до того, як HTML код повністю завантажиться, це збільшує видиму швидкість роботи додатку [6].

Конектори в REST надають інтерфейс для комунікації між компонентами в даній системі. Завдяки конекторам є можливість приховати реалізацію ресурсів, та механізм комунікації. За своєю сутністю конектори подібні, до виклику віддалених процедур (RPC – Remote Procedur Call). RPC – полягає в тому, що дозволяє програмі запусненій на одному комп'ютері, звертатись до функцій програми, що виконується на іншому комп'ютері. Головний нюанс в якому полягає відмінність між конекторами та RPC – це передача параметрів та результат виклику. Параметри в свою чергу складаються з ідентифікатора

ресурсу, контрольних даних та представлення (не є обов'язковим), а результат складається з контрольних даних відповіді та безпосередньо представлення. Найважливішими типами конекторів є клієнт та сервер. Їхня відмінність полягає в тому, що клієнт створює запит, в той час як сервер очікує на запити і відповідає на них відкриваючи доступ до своїх сервісів. Існує також додатковий тип конектора – кеш. Кеш може знаходитись як і на клієнтській стороні для уникнення створення зайвих запитів, так і на серверній – для уникнення зайвого обчислення відповіді на певний запит. Оскільки одним з обов'язкових обмежень в REST є однорідність інтерфейсу, то кеш може легко дізнатись чи можна кешувати запит. За замовченням, відповідь на запит отримання ресурсу кешувати можна, а запити зміни ресурсу кешувати заборонено. Проте всі ці налаштування можна змінити за допомогою контрольних даних [6].

Резолвер – це ще один тип конектора. Резолвер перетворює ідентифікатори ресурсів в інформацію про мережеві адреси, яка є необхідною для компонентів щоб отримати цей ресурс. Наприклад можна розглянути URL в якому міститься доменне ім'я, і для доступу до відповідного домена, потрібно дізнатись адресу в DNS-сервера. В цьому конкретному випадку DNS грає роль резолвера. Використання одного або декількох резолверів здатне збільшити життєздатність ідентифікатора ресурсу, оскільки він не вказує на фізичне розташування ресурсу, яке може змінитися. Останньою формою конектора в REST є тунель. Основним завданням тунеля є проведення запитів через межу системи, наприклад через фаєрвол. Головною причиною чому тунелі включені до REST архітектури є те, що певні компоненти системи можуть перетворюватися в тунелі по запиті. Це чітко можна побачити на прикладі HTTP-тунелю, який активується при отриманні запиту з методом CONNECT [6].

2.2. Семантика протоколу HTTP

HTTP (Hyper Text Transfer Protocol) – це протокол передачі даних, що використовується в комп'ютерних мережах. Даний протокол належить до протоколів моделі OSI 7-го прикладного рівня [7].

В REST архітектурі ресурс може бути представлений чим завгодно, але дії,

які можна виконувати над ресурсом, визначаються в повідомленнях, які описуються стандартним протоколом. В системі WWW цей протокол – HTTP, хоча існують REST архітектури на основі інших мережевих протоколів [7].

Стандарт HTTP визначає 8 типів повідомлень [7]:

GET – це тип повідомлення, який запрошує вміст вказаного ресурсу. Ресурс, що запитується може приймати параметри. Усі параметри передаються за допомогою URL. Згідно з HTTP стандартом запити типу GET вважаються ідемпотентними. Це означає, що багаторазове виконання одного й того самого запиту повинно вертати однаковий результат, за умови, що в проміжок часу між запитами ресурс не змінювався. Завдяки даній властивості дозволяється кешувати відповіді на запити типу GET.

OPTIONS – повертає HTTP методи, які підтримуються веб-сервером. Зазвичай цей метод використовується для визначення можливостей веб-сервера.

HEAD – це метод який за своїм призначенням аналогічний до методу GET, за винятком того, що у відповіді сервера на даний метод відсутнє тіло. Цей метод корисний для витягнення мета-інформації, що задана в заголовках відповіді, без пересилання всієї інформації. Тобто завдяки цьому методу, клієнт може отримати лише заголовки які б відсилалися разом з представленням ресурсу, але не саме представлення ресурсу.

POST – цей метод передає призначені для користувача дані заданому ресурсу. Тобто даний метод створює новий ресурс, використовуючи передане в тілі представлення.

PUT – цей метод змінює стан поточного ресурсу представленням, що передається.

DELETE – цей метод видаляє вказаний ресурс. Також належить до ідемпотентних методів. Хоча після другого виклику даний метод поверне статус 204 No Content, а потім і 404 Not Found, але ресурсу не буде, як після першого видалення так і після 10-го.

TRACE – повертає отриманий запит таким чином, що клієнт може подивитися, що саме проміжні сервери додають в запит або змінюють в ньому.

CONNECT – це метод, який призначений для використання разом з проксі-серверами, які мають змогу динамічно перемикатися в тунельний режим.

Існує також ще 9-ий метод, який щоправда не описаний в HTTP, але описаний в його додатку RFC 5789.

PATCH – це метод, який змінює лише частину ресурсу на основі переданого представлення. Якщо якась частина ресурсу не згадується в переданому представленні, то це означає що її змінювати не потрібно. Завдяки даному методу зменшується кількість інформації, яку потрібно передати для зміни ресурсу.

В протоколі HTTP використовуються такі коди статусів [8]:

- **1xx** – інформаційний. Запит прийнято, продовжуй процес. Сервери не повинні відсилати клієнтам 1xx відповіді, за виключенням експериментальних умов.
- **2xx** – успіх. Цей клас кодів стану вказує на те, що запит клієнта було успішно прийнято і зрозуміло.
- **3xx** – перенаправлення. Даний клас кодів вказує на те, що майбутні дії повинні бути виконані клієнтом, для успішного завершення запиту.
- **4xx** – помилка на стороні клієнта. Цей клас кодів призначений для визначення помилок на стороні клієнта (окрім випадку коли методом запиту був HEAD).
- **5xx** – помилка на стороні сервера. Даний клас кодів вказує на те, що сервер знає про те, що виникла помилка, або у випадку коли сервер не має змоги опрацювати запит клієнта.

Основними статусами, які використовуються даним протоколом є [8]:

- **200 (OK)** – запит виконано успішно. Інформація, яка вертається у відповіді напряму залежить від методу, що використовувався в запиті.
- **301 (Moved Permanently)** – ресурс переміщено. Даний код вказує на те, що ресурсу було назначено нову URL-адресу, і всі наступні звернення до цього ресурсу повинні здійснюватися по новій адресі. Новий URL вказується в полі Location заголовка.
- **403 (Forbidden)** – доступ до запитуваного ресурсу заборонений. Сервер зрозумів запит, але відмовляється його виконувати через обмеженість доступу для клієнта, який звертається до даного ресурсу.
- **404 (Not Found)** – ресурс не знайдено. Сервер не знайшов жодних ресурсів по вказаному URL, також немає жодних вказівок про те, чи це постійний стан, чи ні. Слід використати статус 410 (**Gone**), якщо серверу відомо, що старий ресурс недоступний постійно і сервер немає адреси для пересилання.

- 503 (**Service Unavailable**) – немає доступу до сервісу. Сервер не може опрацювати запит через тимчасове перевантаження, або ж з приводу технічних робіт. Це тимчасовий стан з якого сервер вийде через деякий проміжок часу. Якщо цей час відомий, то його можна вказати в заголовку **Retry-After**.

РОЗДІЛ 3

ВИЗНАЧЕННЯ ТЕХНОЛОГІЇ GRAPHQL

3.1. Основні поняття GraphQL

GraphQL – це мова запитів для API. Дана технологія надає повний і зрозумілий опис даним у вашому API. Також GraphQL надає можливість клієнтам запитувати лише ті дані, які їм потрібні та не загромождає відповідь надлишковою інформацією [9].

На практиці GraphQL побудований на трьох основних блоках, якими є: схема (Schema), запит (Query), розпізнавач (Resolver) [9].

Запит в GraphQL складається з ключового слова `query` та фігурних дужок, в середині яких перераховуються всі необхідні дані, які називаються полями запиту. Надсилаючи запит до API, можна отримати точно те, що і було написано у запиті. Відповідь на запит в даній технології ізоморфна до самого запиту, тобто має ту саму структуру, що і сам запит. Додатки, які використовують GraphQL швидкі і стабільні, оскільки вони самі визначають дані, які хочуть отримати в запиті, а не сервер це робить за них [9].

Також, оскільки GraphQL має одну “розумну” кінцеву точку (endpoint), то клієнтам більше не потрібно багаторазово звертатись до сервера за різними ресурсами. Користуючись структурою запитів, які підтримують вкладеність полів і також дозволяють, щоб деякі поля були цілими масивами даних, GraphQL дає можливість клієнту отримати всю необхідну інформацію за один запит. Оскільки підтримується вкладеність полів, то дані повинні бути чітко структуровані, саме тому дана технологія найкраще підходить для даних, які організовані ієрархічно. Для цього в GraphQL і визначено поняття схеми даних. Дана технологія строго типізована, саме тому всі дані організовані в термінах типів і полів, а не кінцевих точок, як це відбувається в REST API. Типи використовуються для того, аби було чітке розуміння, яку інформацію можна запитувати. Якщо ж користувач помилився, запрошуючи певну інформацію, то завдяки системі типів буде виведено чітку і зрозумілу інформацію про помилку, яку буде досить просто знайти та усунути. Отже, з попереднього опису стає зрозумілим, що використовуючи GraphQL клієнт спілкується не на пряму з сервером, а з проміжною ланкою, якою і є наш GraphQL. Ще одним з головних плюсів даної технології, який надає їй неймовірної гнучкості це те, що схема

запиту і структура бази даних, що знаходиться на сервері – жодним чином не пов’язані. Всі запити які надходять GraphQL оброблює завдяки розпізнавачам, які чітко вказують як і де отримати дані, щоб вони чітко відповідали полю яке запрошує клієнт [9].

Для того аби детальніше розібратись в роботі запитів давайте зупинимось на їхній структурі. Введемо ще декілька допоміжних понять.

Документ GraphQL – це стрічка на мові GraphQL, яка описує одну або декілька операцій. Операція в свою чергу – це одиничний запит даних або мутація (зміна даних), які інтерпретують виконуючий модуль GraphQL [9].

Розглянемо деякий документ:

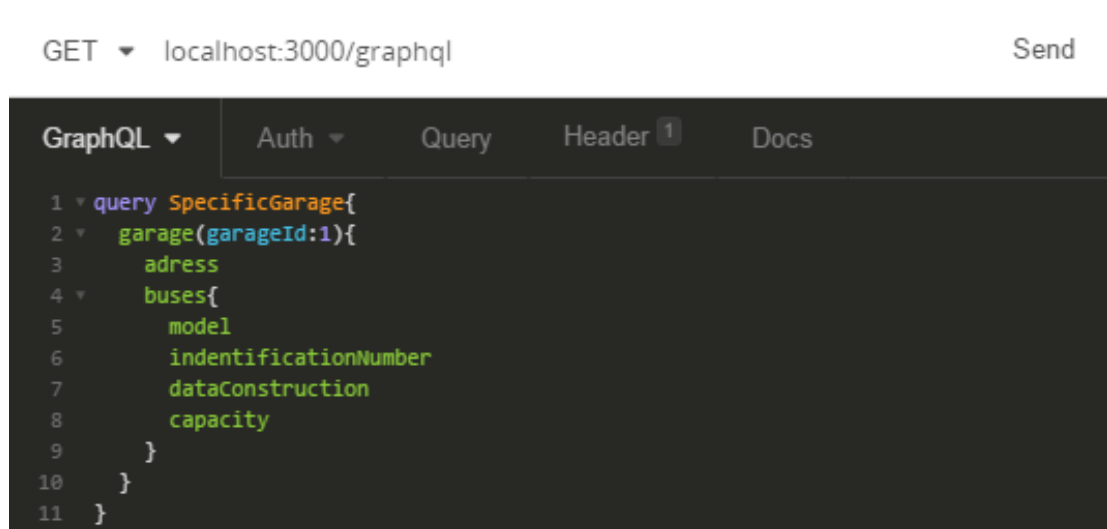


Рис.5. Приклад GraphQL запиту

На даному рисунку (рис.5) показані основні конструкції, за допомогою яких описуються запрошувані дані.

Поле(Field). Це одиниця запрошуваних даних, яка стає полем відповіді у форматі JSON. Зверніть увагу всі частини є полями, як би глибоко вони не знаходились в структурі запиту. Поле на вершині запиту діє так само, як і поле, що знаходиться на рівні глибше [9].

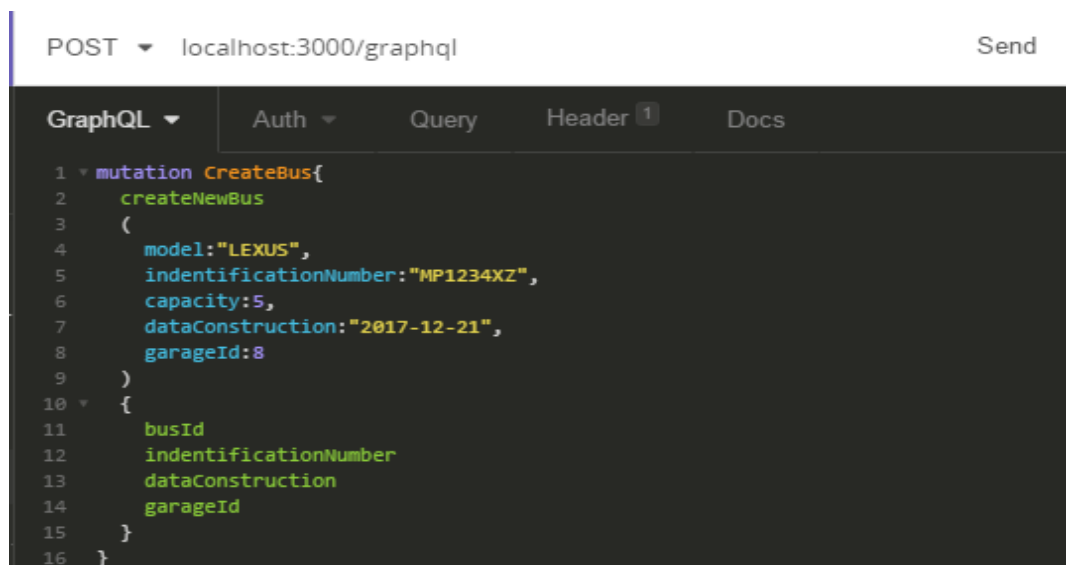
Аргументи (Arguments). Набір пар ключ-значення, пов’язаних з конкретним полем. Вони передаються на сервер обробнику полів і впливають на отримання даних. Аргументи можуть бути літералами, або змінними. Важливо відмітити те, що аргументи можуть бути в будь-якого поля незалежно від його рівня вкладеності [9].

Тип операції (Operation type). Можливо вибрати одне з трьох значень: query, mutation, subscription, яке вказує на тип виконуваної операції. Хоча мовні конструкції з різними операціями виглядають досить схоже, проте специфікація GraphQL передбачає для них різні режими виконання на сервері [9].

Ім'я операції (Operation name). Присвоювати імена операціям зручно для відлагодження і надання логічного навантаження на сервері. Легше знайти проблемний запит в проекті по імені, ніж розбирати вміст запиту. Ім'я операції по своєму смислового навантаженню дуже схоже на ім'я функції в будь-якій мові програмування [9].

Визначення змінних (Variable definitions). Запит GraphQL може мати динамічну частину, яка змінюється при різних зверненнях до сервера, в той час як текст запиту залишається незмінним [9].

Запит може звертатись до сервера не лише для того аби отримати дані, а й для того аби додати, редагувати або видалити їх. Саме для цього в GraphQL слугують мутації (рис.6).

A screenshot of a GraphQL client interface. At the top, it shows 'POST' and 'localhost:3000/graphql' with a 'Send' button. Below this is a tabbed interface with 'GraphQL' selected. The main area displays a GraphQL mutation query for creating a new bus. The query is as follows:

```
1 mutation CreateBus{
2   createNewBus
3   (
4     model:"LEXUS",
5     indentificationNumber:"MP1234XZ",
6     capacity:5,
7     dataConstruction:"2017-12-21",
8     garageId:8
9   )
10  {
11    busId
12    indentificationNumber
13    dataConstruction
14    garageId
15  }
16 }
```

Рис.6. Приклад GraphQL мутації

Для того аби додати об'єкт(елемент) до бази даних необхідно визначити потрібну мутацію в GraphQL схемі. Якщо поле визначене, як GraphQLNonNull, то це означає що поле є обов'язковим і виклик мутації без вказання такого поля призведе до виникнення помилки. Але дану помилку буде не складно відстежити завдяки все тій же строгій типізації та схемі [9].

З даного рисунку видно, що для того аби виконати мутацію необхідно

використати ключове слово `mutation` після чого можна вказати назву даній операції, але це не є обов'язковою опцією. Після потрібно викликати мутацію, яка відповідаю за створення нового елемента бази. За допомогою пар ключ-значення визначити, які саме поля будуть мати значення. В наступних фігурних дужках ми можемо запросити поля елемента, який був доданий до бази, щоб переконатись у тому, що запис було здійснено успішно [9].

Ще досить важливим плюсом даної технології є те, що вона самостійно перевіряє вхідні дані на валідність. Взагалі кажучи, GraphQL перевіряє кожен запит або мутацію на схему. Це відіграє важливу роль при складній формі вхідних даних, адже нам не потрібно писати крихкий код валідації цих даних. GraphQL потурбується про це замість нас [9].

Досить зручним помічником під час створення запитів є поняття інтроспекції схеми. Завдяки йому ви можете запросити в GraphQL поточну схему. Дана перевага надає нам мета-повноваження для динамічного виявлення схеми.

Запит, який повертає всі імена типів та їх опис (рис.7):

```
1 query SchemaType {
2   __schema {
3     types {
4       name
5       description
6     }
7   }
8 }
```

Рис.7. Запит на отримання поточної схеми GraphQL

3.2. Порівняльна характеристика REST технології з GraphQL, їх переваги та недоліки

Вже знаючи, дещо про сильні і слабкі сторони обох технологій давайте підведемо підсумок і проаналізуємо основні відмінності та схожості між цими технологіями. [10]

Характеристики, які є спільними для обох підходів [10]:

1. Обидві технології мають поняття ресурсу і дають можливість надати ідентифікатор для ресурсу.
2. Ресурси можуть бути отримані за допомогою **GET**-запиту **URL**-адреси по **HTTP** протоколу.
3. Відповідь на запит може повертати дані представлені у JSON форматі.

4. Список кінцевих точок (endpoint) в REST API схожий на список полів в типах query та mutation, що використовуються в GraphQL. Обидва поняття є точками входу для доступу до даних.
5. Обидві технології дають можливість розрізняти запити до API на читання і на запис.
6. Кінцеві точки (endpoint) REST API та поля в GraphQL в решті решт функції на сервері.

Характеристики, згідно яких технології відрізняються [10]:

1. В REST API кінцева точка, що викликається – це і є сутність об'єкта. На відміну від REST в GraphQL сутність об'єкта чітко відділена від поняття самого отримання цього об'єкта.
2. В REST API структура і об'єм ресурсу визначаються сервером. В GraphQL сервер визначає набір доступних ресурсів, а клієнт запрошує необхідні йому дані безпосередньо в запиті.
3. За допомогою GraphQL клієнт може одним запитом дістати всю необхідну йому інформацію, завдяки зв'язкам між сутностями, які визначені в схемі. В REST для отримання зв'язаних ресурсів прийдесться зробити декілька запитів до різних кінцевих точок для отримання всієї необхідної інформації.
4. В REST запис даних визначається зміною HTTP-методу запиту з **GET** наприклад на **POST** або **PUT**. Використовуючи GraphQL відбувається зміна ключового слова з **query** на **mutation**.
5. В REST кожен запит зазвичай викликає рівно одну функцію-обробник, в той час як запит в GraphQL може викликати багато функцій-розпізнавачів для побудови складної відповіді з багатьма вкладеними ресурсами.

Різницю між цими двома технологіями наглядно демонструє наступна схема:

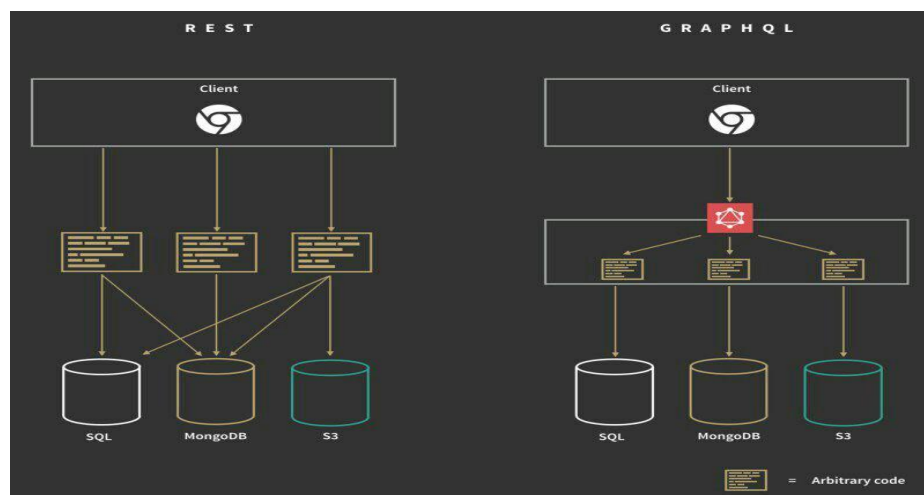


Рис.8. Схема порівняння REST API та GraphQL

Також варто розглянути чим відрізняються життєвий цикл HTTP-запиту в REST і в GraphQL.

Життєвий цикл HTTP-запиту в REST складається з 4 кроків:

1. Сервер отримує запит і витягує HTTP-метод та URL-адресу.
2. API-фреймворк зіставляє метод і шлях з функцією, що зареєстрована в серверному коді.
3. Функція виконується один раз і повертає результат.
4. API-фреймворк перетворює результат, додає відповідний код відповіді та заголовки відповіді, і відправляє це все клієнту.

Життєвий цикл HTTP-запиту в GraphQL також складається з 4 кроків:

1. Сервер отримує HTTP-запит і витягує з нього GraphQL-запит.
2. Запит проходить наскрізь, і для кожного поля викликається відповідна функція-розпізнавач.
3. Функція викликається і повертається результат.
4. Бібліотека GraphQL і сервер прикріплюють отриманий результат до відповіді, яка відповідає формі запиту, що був надісланий клієнтом [10].

РОЗДІЛ 4

РЕАЛІЗАЦІЯ ТЕХНОЛОГІЇ GRAPHQL ЯК АЛЬТЕРНАТИВА REST API

4.1. Постановка задачі

Оскільки ІТ сфера розвивається дуже швидко, відповідно деякі компанії мають дуже великі проекти на яких використовуються застарілі технології. І досить часто виникають ситуації коли старенька технологія вже не може коректно підтримуватися розробниками і функціонувати на високому рівні. Саме з такою проблемою і зіштовхнулись розробники компанії Facebook. Їхній API став настільки складним що їх мобільний додаток не міг нормально функціонувати, а користувачі в свою чергу були змушені терпіти недоліки роботи такого додатку, головним з яких була дуже повільна відповідь сервера на запити користувачів.

Демонстраційний додаток, який був розроблений в ході даної курсової роботи, повинен продемонструвати основні переваги GraphQL для роботи з API. Для написання додатку було вибрано мову JavaScript на платформі Node.JS. Застосувався підхід розбиття додатку на окремі модулі, кожен з яких мав би відповідати за свій базовий функціонал. В ході розробки був використаний популярний фреймворк для роботи з сервером express. Було реалізовано зв'язок з реляційною базою MySQL звідки і брались дані для демонстрації роботи додатку. Оскільки дані на сервері отримувались з бази асинхронно, то було використано фреймворк bluebird, для того аби вирішити цю проблему. Завдяки використанню даного програмного забезпечення стало можливим повернення промісу (Promise), який і допоміг вирішити проблеми з асинхронністю.

4.2. Реалізація поставленого завдання та демонстрація роботи додатку

Для реалізації додатку в першу чергу було створено тестову БД, в якій є лише 2 таблиці, які пов'язані за зовнішнім ключем. Далі було прописано з'єднання з базою з використанням mysql фреймворку для роботи з базою в Node.JS. Після

чого було описано сутності використовуваних об'єктів АВТОБУС та ГАРАЖ в GraphQL.

Опис сутності автобус (див. Додаток 1. schema.js):

```
const busType = new GraphQLObjectType({
  name: 'bus',
  description: 'Describing bus type!',
  fields: () => ({
    busId: {
      type: GraphQLInt
    },
    model: {
      type: GraphQLString
    },
    indentificationNumber: {
      type: GraphQLString
    },
    capacity: {
      type: GraphQLInt
    },
    dataConstruction: {
      type: GraphQLDate
    },
    garageId: {
      type: GraphQLInt,
    }
  })
});
```

Опис сутності гараж (див. Додаток 1. schema.js):

```
const garageType = new GraphQLObjectType({
  name: 'garage',
  description: 'Describing garage type!',
  fields: () => ({
    garageId: {
```

```

        type: GraphQLInt
      },
      adress: {
        type: GraphQLString
      },
      owner: {
        type: GraphQLString
      },
      buses: {
        type: new GraphQLList(busType),
        description: 'Select all the buses of the specific garage.',
        resolve(root, args){
          return new Promises(function(resolve, reject) {
            mc.query('SELECT * FROM bus WHERE garageId = ?',
            [root.garageId], function(err, result) {
              return (err ? reject(err) : resolve(result));
            });
          });
        }
      }
    })
  });

```

Як видно з опису гаража вже присутня вкладеність полів. Сутність гараж буде містити в собі поле, яке буде масивом тих автобусів, що знаходяться в даному гаражі.

Було реалізовано запити та мутації для:

- Створення автобуса і/або гаража.
- Редагування автобуса і/або гаража.
- Перегляду інформації про поточні автобуси і/або гаражі.
- Видалення автобуса і/або гаража.

З наступного запиту видно, як ми можемо маніпулювати полями в запиті для того аби не отримувати надлишкової інформації у відповіді. Саме завдяки цій

перевазі GraphQL і здобув такої популярності, адже сервер не перенавантажується, а клієнт отримує лише те, що йому потрібно.

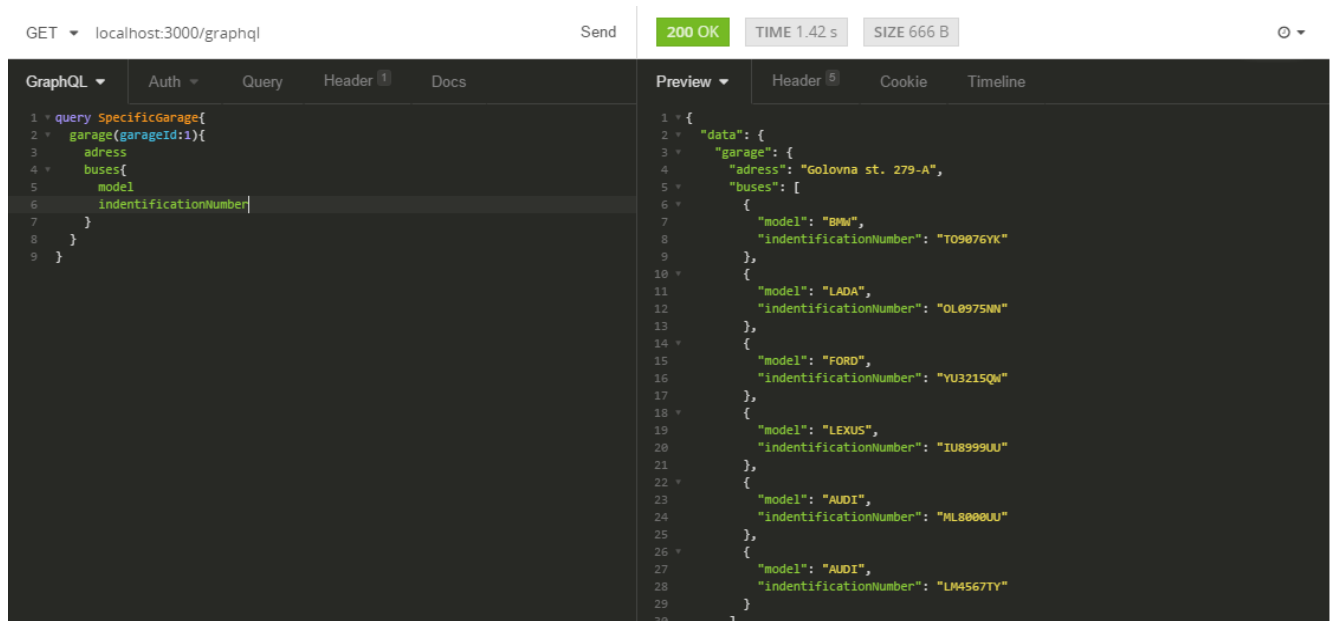


Рис.9. Приклад GraphQL запиту із вкладеними полями

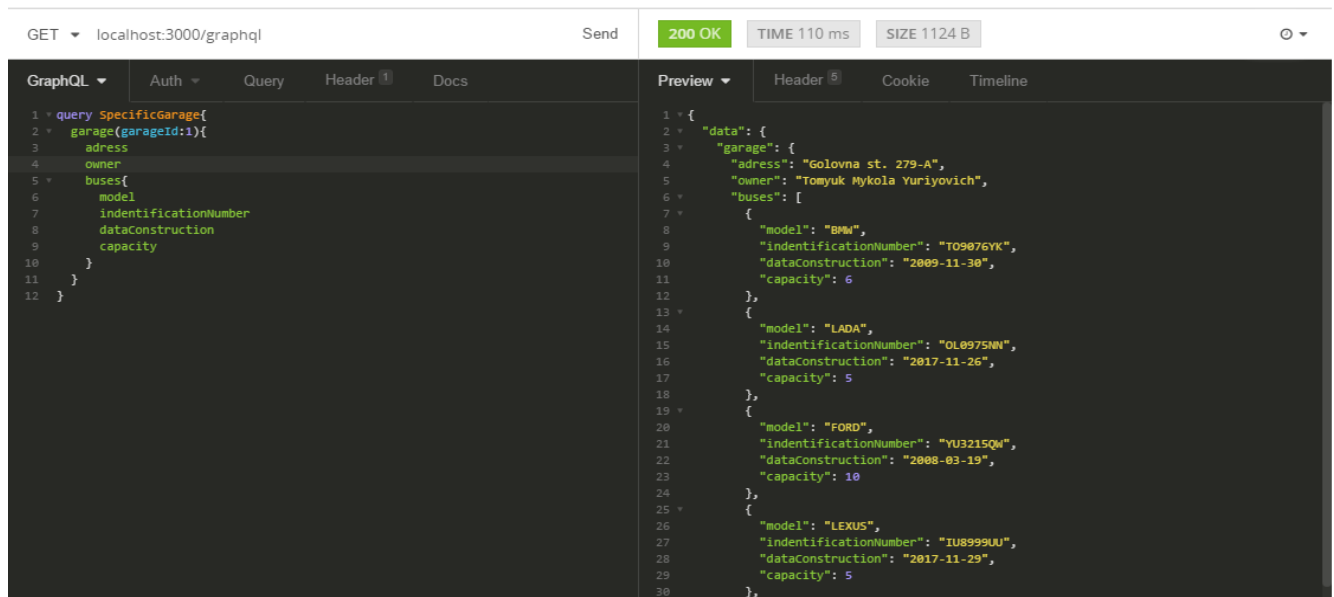


Рис.10. Приклад GraphQL запиту з великою кількістю полів і аргументом

Також порівнюючи ці два запити чітко видно, що вони є ізоморфними відносно даних, які повернулись у відповідь з сервера.

Для виконання запису даних, їх редагування або видалення використовуються мутації, які також описані в схемі GraphQL.

Мутації (див. Додаток 1. schema.js):


```

const rootMutation = new GraphQLObjectType({
  name: "rootMutation",
  description: 'Mutating information about buses and garages!',
  fields: {
    createNewBus: {
      type: busType,
      args: {
        model: {
          type: new GraphQLNonNull(GraphQLString)
        },
        indentificationNumber: {
          type: new GraphQLNonNull(GraphQLString)
        },
        capacity: {
          type: new GraphQLNonNull(GraphQLInt)
        },
        dataConstruction: {
          type: new GraphQLNonNull(GraphQLDate)
        },
        garageId: {
          type: new GraphQLNonNull(GraphQLInt)
        }
      },
      resolve(root, args){
        return new Promises(function(resolve, reject) {
          mc.query('INSERT INTO bus SET ?', args ,function(err, result) {
            return (err ? reject(err) : resolve(
              new Promises(function(resolve, reject){
                mc.query('SELECT * FROM bus WHERE indentificationNumber = ? AND garageId =
?', [args.indentificationNumber, args.garageId], function(err, result) {
                  return (err ? reject(err) : resolve(result[0]));
                });
              })
            ));
          });
        });
      }
    }
  }
});

```

```

    });
  }
}
}

```

Даний код демонструє мутацію (рис.11), яка додає новий автобус до бази даних. Як видно зі схеми під час вказання аргументів ми використовували тип GraphQLNonNull, який означає, що дане поле є обов'язковим. Саме тому під час створення мутація буде виглядати наступним чином:

The screenshot shows the GraphQL Playground interface. On the left, the 'Query' tab is active, displaying a mutation query: `mutation CreateBus{ createNewBus { model:"LEXUS", indentificationNumber:"MP1234XZ", capacity:5, dataConstruction:"2017-12-21", garageId:8 } }`. On the right, the 'Preview' tab shows the JSON response: `{ "data": { "createNewBus": { "busId": 54, "indentificationNumber": "MP1234XZ", "dataConstruction": "2017-12-20", "garageId": 8 } } }`.

Рис.11. Приклад створення об'єкта за допомогою мутації

Відсутність якогось з полів в мутації призведе до помилки (рис.12), адже всі поля під час створення автобуса зазначені, як обов'язкові. Але в помилці буде чітко вказано, що і чому саме її спричинило.

The screenshot shows the GraphQL Playground interface with an error. The 'Query' tab on the left contains the same mutation query as in Figure 11. The 'Preview' tab on the right displays an error response: `{ "errors": [{ "message": "Field \"createNewBus\" argument \"capacity\" of type \"Int!\" is required but not provided.", "locations": [{ "line": 2, "column": 2 }] }] }`. The error message indicates that the 'capacity' field is required but not provided.

Рис.12. Приклад помилки при відсутності обов'язкового поля

З даного скріншота видно, що не вистачає аргументу `capacity` типу `int`. В помилці говориться що даний аргумент обов'язковий але не вказаний в мутації. Завдяки такому зручному сервісу повідомлень про помилки ми легко можемо її відстежити та усунути.

Порівнюючи отриману інформацію у відповіді з використанням технології GraphQL, з відповіддю яку ми отримуємо за допомогою REST, ми чітко бачимо, що для того аби не отримувати надлишкової інформації в REST потрібно додати цілу купу різних кінцевих точок, і все одно можна не врахувати якусь комбінацію полів. Додавання великої кількості кінцевих точок сильно загромождає код, але надає клієнту більше можливостей отримати лише потрібну інформацію, в той час як в GraphQL клієнт в запиті сам визначає, які саме поля він хоче отримати. Таким чином GraphQL має лише одну “розумну” кінцеву точку через яку клієнт і отримує необхідну інформацію.

GET запит в GraphQL (рис.13):

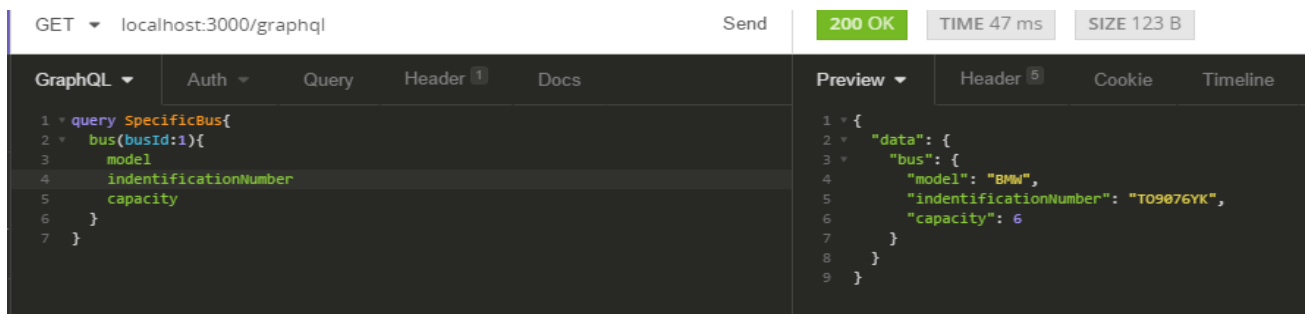


Рис.13. Приклад GET запиту GraphQL

GET запит в REST API (рис.14):

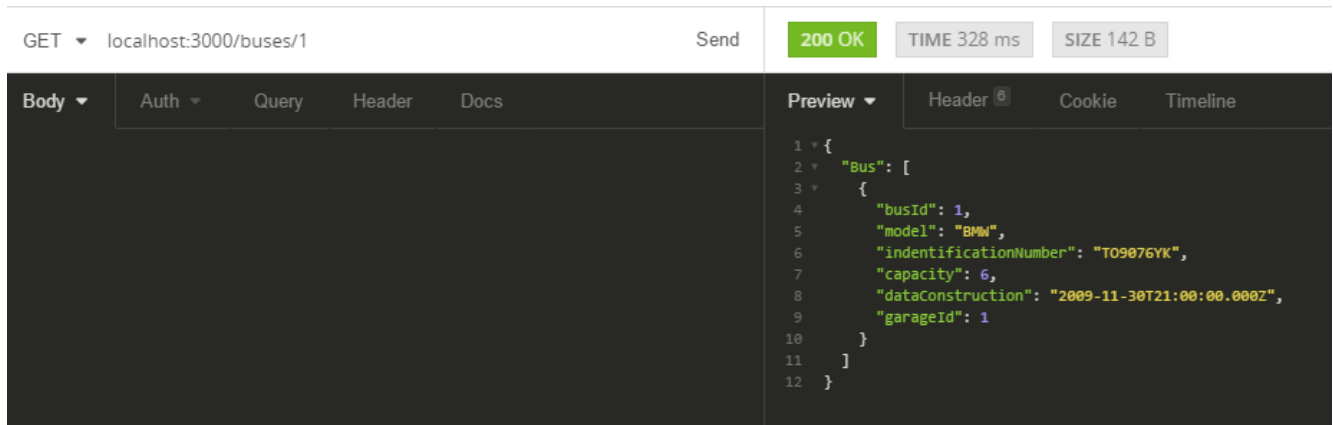


Рис.14. Приклад GET запиту REST API

Подивившись уважно ще раз на ці скріншоти легко можна зрозуміти, що інформацію яка повернеться на запит в REST визначена на сервері, а відповідь сервера у випадку з GraphQL буде на 100% відображати запрошувані дані.

Ще одною відмінністю між REST API та GraphQL є те як за допомогою їх використання оновлювати дані. Використовуючи REST технологію є два способи оновлювати дані, один з них підходить в тому випадку коли потрібно оновити повністю всі дані, а інший лише коли частину даних так, щоб частина яку не оновлювали залишилась без змін. Це HTTP-методи PUT і PATCH відповідно. Для цих самих дій в GraphQL достатньо написати одну мутацію в якій вказати, що поля не є обов'язковими для оновлення. Це значно спрощує життя клієнту, адже він самостійно може визначати варто йому оновлювати певні поля чи ні, і для цього не потрібно прописувати низку зайвого коду, як у випадку з REST API. Ці всі відмінності чітко видно з наступних скріншотів.

Виконання PUT методу (рис.15):

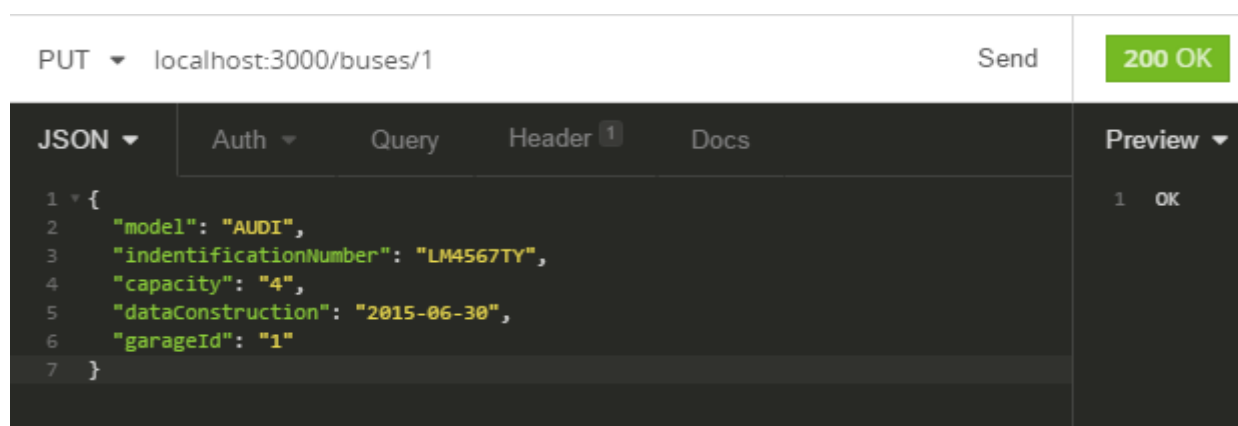


Рис.15. Виконання HTTP- метода PUT

Виконання PATCH методу (рис.16):

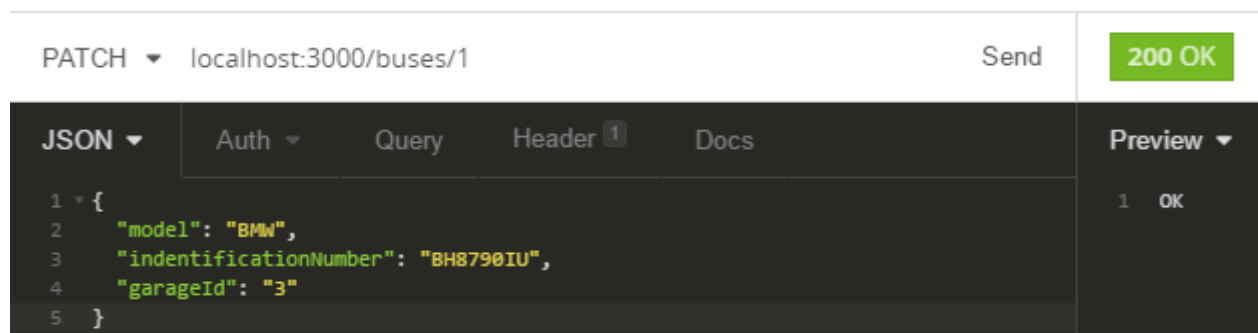
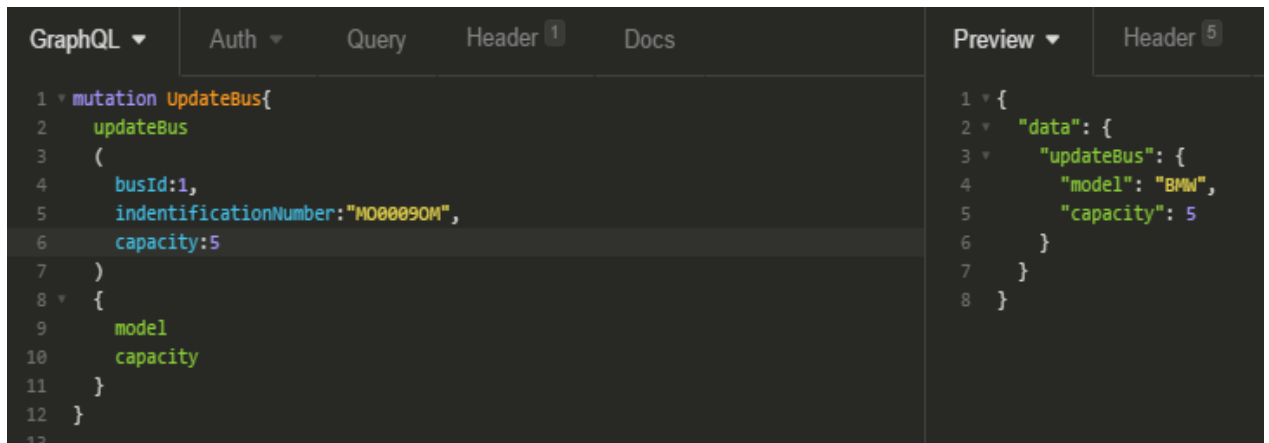


Рис.16. Виконання HTTP- метода PATCH

GraphQL мутація (рис.17):



The screenshot shows a GraphQL IDE interface with two main panels. The left panel, titled 'GraphQL', contains a mutation query. The right panel, titled 'Preview', shows the JSON response.

```
1 mutation UpdateBus{
2   updateBus
3   (
4     busId:1,
5     indentificationNumber:"MO00090M",
6     capacity:5
7   )
8   {
9     model
10    capacity
11  }
12 }
```

```
1 {
2   "data": {
3     "updateBus": {
4       "model": "BMW",
5       "capacity": 5
6     }
7   }
8 }
```

Рис.17. GraphQL мутація для оновлення об'єкта

Реалізації цих двох технологій значною мірою відрізняються одна від одної. Використовуючи REST клієнт на пряму спілкується з сервером, в той час як GraphQL є проміжною ланкою між клієнтом і сервером, яка забирає запити у клієнта і передає їх серверу.

ВИСНОВКИ

Отже, можна зробити висновок, що GraphQL це сучасний метод для роботи з API, який не просто так завойовує популярність на IT ринку, а завдяки своїй гнучкості та універсальності під час розробки додатків з його використанням. Порівнюючи його з REST технологією ми визначили, що GraphQL вирішує такі нагальні проблеми, як недовантаження даних та надлишкову їх завантаження. Звичайно можна говорити про те, що можна зробити велику кількість кінцевих точок і тоді ця проблема вирішиться, але таким чином код стає дуже складно відлагоджувати і масштабувати. GraphQL – це інноваційний підхід, який дає змогу клієнту отримувати лише те, чого він потребує. Використовуючи GraphQL розробники нарешті можуть роз'єднати роботу Front-End від Back-End, що значно пришвидшить роботу розробки. Окрім цього, гнучкість даної технології полягає ще в тому, що вона не залежить від джерела даних, а це означає, що не прийдеться створювати безліч кінцевих точок, як би це прийшлося б робити з використанням REST, якби дані знаходились в різних джерелах, як от в SQL та NoSQL. Саме тому використання однієї “розумної” кінцевої точки і спрощує життя клієнту. Адже тепер не потрібно чекати доки дані будуть довантажуватись з різних точок.

У ході даної курсової роботи було розглянуто дві найпопулярніші на сьогоднішній день технології для роботи з API. Було продемонстровано роботу двох різних додатків, які були створені з використанням цих технологій. Було проведено порівняльну характеристику REST API та GraphQL, у ході якої чітко видно, що на сьогоднішній день GraphQL переважає свого опонента за багатьма аспектами.

СПИСОК ДЖЕРЕЛ ТА ЛІТЕРАТУРИ

1. Вступ до GraphQL [Електронний ресурс] – Режим доступу до ресурсу: <http://graphql.org/learn/>
2. С. Zakas N. Розуміння ECMAScript 6 / Nicholas C. Zakas., 2015. – 363 с.
3. Brown E. Web Development with Node.JS and Express / Ethan Brown., 2014. – 291 с.
4. REST [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/REST#CITEREFFielding2000>
5. GraphQL: все, що вам потрібно знати [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@weblab_tech/graphql-everything-you-need-to-know-58756ff253d8
6. Підручник з REST API [Електронний ресурс] – Режим доступу до ресурсу: <https://knpuniversity.com/screencast/rest/intro>
7. HTTP [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.wikipedia.org/wiki/HTTP>
8. Список кодів стану HTTP [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D1%96%D0%B2_%D1%81%D1%82%D0%B0%D0%BD%D1%83_HTTP
9. Що таке GraphQL? [Електронний ресурс] – Режим доступу до ресурсу: <https://code.tutsplus.com/ru/tutorials/what-is-graphql--cms-29271>
10. Порівняння REST API і GraphQL [Електронний ресурс] – Режим доступу до ресурсу: <https://habrahabr.ru/post/335158/>

ДОДАТКИ

Додаток 1. Лістинг коду schema.js

```
const Promises = require('bluebird');
const mc = require('../dbConnection');
const {GraphQLDate} = require('graphql-iso-date');
const {
  GraphQLObjectType,
  GraphQLString,
  GraphQLInt,
  GraphQLSchema,
  GraphQLList,
  GraphQLNonNull
} = require('graphql');

// Bus type specification!
const busType = new GraphQLObjectType({
  name: 'bus',
  description: 'Describing bus type!',
  fields: () => ({
    busId: {
      type: GraphQLInt
    },
    model: {
      type: GraphQLString
    },
    indentificationNumber: {
      type: GraphQLString
    },
    capacity: {
      type: GraphQLInt
    },
    dataConstruction: {
      type: GraphQLDate
    },
  })
});
```



```

garage: {
  type: garageType,
  description: 'Select garage where the bus is situated.',
  resolve(root, args){
    return new Promises(function(resolve, reject) {
      mc.query('SELECT garage.* FROM (garage INNER JOIN bus ON
garage.garageId = bus.garageId) WHERE bus.busId = ?', [root.busId]
,function(err, result){
        return (err ? reject(err) : resolve(result[0]));
      });
    });
  }
}
})
});

```

//Garage type specification!

```

const garageType = new GraphQLObjectType({
  name: 'garage',
  description: 'Describing garage type!',
  fields: () => ({
    garageId: {
      type: GraphQLInt
    },
    adress: {
      type: GraphQLString
    },
    owner: {
      type: GraphQLString
    },
    buses: {
      type: new GraphQLList(busType),
      description: 'Select all the buses of the specific garage.',
      resolve(root, args){
        return new Promises(function(resolve, reject) {

```

```

        mc.query('SELECT * FROM bus WHERE garageId = ?',
[root.garageId], function(err, result) {
            return (err ? reject(err) : resolve(result));
        });
    });
}
}
})
});

```

//Root query, where all the get queries are described!

```

const rootQueryType = new GraphQLObjectType({
  name: 'rootQuery',
  description: 'Fetching information about buses and garages!',
  fields: {
    bus: {
      type: busType,
      args: {
        busId: {
          type: GraphQLInt
        }
      },
      resolve(root, args) {
        return new Promises(function(resolve, reject) {
          mc.query('SELECT * FROM bus WHERE busId = ?', [args.busId],
function(err, result) {
            return (err ? reject(err) : resolve(result[0]));
          });
        });
      }
    },
    buses: {
      type: new GraphQLList(busType),
      resolve(root, args){
        return new Promises(function(resolve, reject) {

```

```

        mc.query('SELECT * FROM bus', function(err, result) {
            return (err ? reject(err) : resolve(result));
        });
    });
}
},
garage: {
    type: garageType,
    args: {
        garageId: {
            type: GraphQLInt
        }
    },
    resolve(root, args) {
        return new Promises(function(resolve, reject) {
            mc.query('SELECT * FROM garage WHERE garageId = ?',
[args.garageId], function(err, result) {
                return (err ? reject(err) : resolve(result[0]));
            });
        });
    }
},
garages: {
    type: new GraphQLList(garageType),
    resolve(root, args) {
        return new Promises(function(resolve, reject){
            mc.query('SELECT * FROM garage', function(err, result) {
                return (err ? reject(err) : resolve(result));
            });
        })
    }
}
}
});

```

```

//Mutations
const rootMutation = new GraphQLObjectType({
  name: "rootMutation",
  description: 'Mutating information about buses and garages!',
  fields:{
    createNewBus: {
      type: busType,
      args: {
        model: {
          type: new GraphQLNonNull(GraphQLString)
        },
        indentificationNumber: {
          type: new GraphQLNonNull(GraphQLString)
        },
        capacity: {
          type: new GraphQLNonNull(GraphQLInt)
        },
        dataConstruction: {
          type: new GraphQLNonNull(GraphQLDate)
        },
        garageId: {
          type: new GraphQLNonNull(GraphQLInt)
        }
      },
      resolve(root, args){
        return new Promises(function(resolve, reject) {
          mc.query('INSERT INTO bus SET ?', args ,function(err,
result) {
            return (err ? reject(err) : resolve(
              new Promises(function(resolve, reject){
                mc.query('SELECT * FROM bus WHERE
indentificationNumber = ? AND garageId = ?', [args.indentificationNumber,
args.garageId], function(err, result) {
                  return (err ? reject(err) :
resolve(result[0]));
                });
              });
            });
          });
        });
      }
    }
  }
});

```

```

        })
    ));
    });
    });
}
},
updateBus: {
    type: busType,
    args:{
        busId: {
            type: new GraphQLNonNull(GraphQLInt)
        },
        indentificationNumber: {
            type: GraphQLString
        },
        capacity: {
            type: GraphQLInt
        },
        dataConstruction: {
            type: GraphQLDate
        },
        garageId: {
            type: GraphQLInt
        }
    },
    resolve(root, args){
        return new Promises(function(resolve, reject) {
            mc.query('UPDATE bus SET ? WHERE busId = ?', [args,
args.busId] ,function(err, result) {
                return (err ? reject(err) : resolve(
                    new Promises(function(resolve, reject){
                        mc.query('SELECT * FROM bus WHERE busId = ?',
[args.busId], function(err, result) {
                            return (err ? reject(err) :
resolve(result[0]));
                        });
                    });
            });
        });
    });

```

```

        })
    ));
    });
    });
}
},
deleteBus: {
    type: busType,
    args: {
        busId: {
            type: new GraphQLNonNull(GraphQLInt)
        }
    },
    resolve(root, args) {
        return new Promises(function(resolve, reject) {
            mc.query('DELETE FROM bus WHERE busId = ?', [args.busId]
,function(err, result) {
                return (err ? reject(err) : resolve(result));
            });
        });
    }
},
createNewGarage: {
    type: garageType,
    args: {
        adress: {
            type: new GraphQLNonNull(GraphQLString)
        },
        owner: {
            type: new GraphQLNonNull(GraphQLString)
        }
    },
    resolve(root, args) {
        return new Promises(function(resolve, reject) {

```

```

        mc.query('INSERT INTO garage SET ?', args, function(err,
result) {

            return (err ? reject(err) : resolve(
                new Promises(function(resolve, reject){
                    mc.query('SELECT * FROM garage WHERE owner = ?
AND adress = ?', [args.owner, args.adress], function(err, result) {
                        return (err ? reject(err) :
resolve(result[0]));
                    });
                })
            ));
        });
    });
    });
    });
    });
    });
    },
    },

    updateGarage: {
        type: garageType,
        args: {
            garageId: {
                type: new GraphQLNonNull(GraphQLInt)
            },
            adress: {
                type: GraphQLString
            },
            owner: {
                type: GraphQLString
            }
        },
        resolve(root, args){
            return new Promises(function(resolve, reject) {
                mc.query('UPDATE garage SET ? WHERE garageId = ?', [args,
args.garageId] ,function(err, result) {
                    return (err ? reject(err) : resolve(
                        new Promises(function(resolve, reject){
                            mc.query('SELECT * FROM garage WHERE garageId =
?', [args.garageId], function(err, result) {

```

```

                                return (err ? reject(err) :
resolve(result[0]));
                                });
                                })
                                ));
                                });
                                });
                                }
                                },
                                deleteGarage: {
                                    type: garageType,
                                    args: {
                                        garageId: {
                                            type: new GraphQLNonNull(GraphQLInt)
                                        }
                                    },
                                    resolve(root, args) {
                                        return new Promises(function(resolve, reject) {
                                            mc.query('DELETE FROM garage WHERE garageId = ?',
[args.garageId] ,function(err, result) {
                                                return (err ? reject(err) : resolve(result));
                                            });
                                        });
                                    }
                                }
                            }
                        });
                        module.exports = new GraphQLSchema({
                            query: rootQueryType,
                            mutation: rootMutation
                        });

```

Додаток 2. Лістинг коду dbConnection.js

//Data base connection configurations.

```
const mysql = require('mysql');
```

```
const dbConnection = mysql.createConnection({
```



```

    host: 'localhost',
    user: 'andrew',
    password: 'andrew1997',
    database: 'bus_app'
  });
module.exports = dbConnection;

```

Додаток 3. Лістинг коду server.js

```

const express = require('express');
const graphql = require('express-graphql');
const schema = require('./data/schema');
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
  extended: true
}));
app.use('/graphql', graphql({
  schema: schema,
  pretty: true,
  graphiql: true
}));
app.listen(5000, function(){
  console.log('Connected to the server via the port 5000!!!');
});

```

Додаток 4. Лістинг коду dbConnection.js

```

//Data base connection configurations.
const mysql = require('mysql');
const dbConnection = mysql.createConnection({
  host: 'localhost',
  user: 'andrew',
  password: 'andrew1997',
  database: 'bus_app'
});
module.exports = dbConnection;

```

Додаток 5. Лістинг коду server.js

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const mc = require('./dbConnection.js');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));
/* Starting our server with port 3000 */
app.listen(3000, function () {
    console.log('Connected to the server via the port 3000!!!');
});
/* Default route with / url */
app.get('/', function (request, response) {
    return response.send({ message: 'Connected to the server via the port 3000!!!' });
});
/* Get buses app */
app.get('/buses', function (request, response) {
    mc.query('SELECT * FROM bus;', function (error, results, fields) {
        if (error) throw error;
        return response.send({ Buses: results });
    });
});
app.get('/buses/:id', function(request, response){
    let id = request.params.id;
    if(!isFinite(id)) {
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id must contain only numbers!' });
    }
    mc.query('SELECT busId FROM bus WHERE busId = ?', [id], function(error, results, fields){
        if(Object.keys(results).length == 0){
            return response.sendStatus(404);
        }
    })
}
```

```

        mc.query('SELECT * FROM bus WHERE busId = ?', [id] , function(error,
results, fields){
            if(error) throw error;
            return response.send({ Bus: results});
        });
    });
});
/* Post bus app */
app.post('/buses', function(request, response){
    let newBus = request.body;
    if (Object.keys(newBus).length < 5) {
        return response.status(400).send({ error: true, message: 'Please
provide information about bus in json format.' });
    }
    mc.query('INSERT INTO bus SET ? ', newBus , function (error, results,
fields) {
        if (error) throw error;
        return response.sendStatus(200);
    });
});
/* Put bus app */
app.put('/buses/:id', function(request, response){
    let id = request.params.id;
    let updateBus = request.body;
    if(!isFinite(id)){
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id
must contain only numbers!' });
    }
    if(Object.keys(updateBus).length < 5){
        return response.status(400).send({ error: true, message: 'Please
provide information about bus in json format.' });
    }
    mc.query('SELECT busId FROM bus WHERE busId = ?', [id], function(error,
results, fields){
        if(Object.keys(results).length == 0){
            return response.status(404).send({error: 'Not Found', message:
'There is no bus with id = ' + id});
        }
    });
});

```

```

    }
    mc.query('UPDATE bus SET ? WHERE busId = ?', [updateBus, id] ,
function(error, results, fields){
    if(error) throw error;
    return response.sendStatus(200);
});
});
});
/* Patch specific bus */
app.patch('/buses/:id', function(request, response){
    let id = request.params.id;
    let updateBus = request.body;
    if(!isFinite(id)){
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id
must contain only numbers!' });
    }
    if(Object.keys(updateBus).length < 1){
        return response.status(400).send({ error: true, message: 'Please
provide information about bus in json format.' });
    }
    mc.query('SELECT busId FROM bus WHERE busId = ?', [id], function(error,
results, fields){
        if(Object.keys(results).length == 0){
            return response.status(404).send({error: 'Not Found', message:
'There is no bus with id = ' + id});
        }
        mc.query('UPDATE bus SET ? WHERE busId = ?', [updateBus, id] ,
function(error, results, fields){
            if(error) throw error;
            return response.sendStatus(200);
        });
    });
});
app.patch('/buses', function(request, response){
    return response.sendStatus(400);
});

```

```

/* Delete bus app */
app.delete('/buses/:id', function(request, response){
    let id = request.params.id;
    if(!isFinite(id)) {
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id
must contain only numbers!' });
    }
    mc.query('SELECT busId FROM bus WHERE busId = ?', [id], function(error,
results, fields){
        if(Object.keys(results).length == 0){
            return response.sendStatus(404);
        }
        mc.query('DELETE FROM bus WHERE busId = ?', [id], function (error,
results, fields) {
            if (error) throw error;
            return response.sendStatus(200);
        });
    });
});
app.delete('/buses', function(request, response){
    return response.sendStatus(400);
});
/* Get garages app */
app.get('/garages', function (request, response) {
    mc.query('SELECT * FROM garage', function (error, results, fields) {
        if (error) throw error;
        return response.send({ Garages: results });
    });
});
app.get('/garages/:id', function(request, response){
    let id = request.params.id;
    if(!isFinite(id)){
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id
must contain only numbers!' });
    }
    mc.query('SELECT garageId FROM garage WHERE garageId = ?', [id],
function(error, results, fields){

```

```

        if(Object.keys(results).length == 0){
            return response.sendStatus(404);
        }
        mc.query('SELECT * FROM garage WHERE garageId = ?', [id] ,
function(error, results, fields){
            if(error) throw error;
            return response.send({ Garage: results});
        });
    });
});
/* Post garage */
app.post('/garages', function(request, response){
    let newGarage = request.body;
    if (Object.keys(newGarage).length < 2) {
        return response.status(400).send({ error: true, message: 'Please
provide information about bus in json format.' });
    }
    mc.query('INSERT INTO garage SET ? ', newGarage , function (error, results,
fields) {
        if (error) throw error;
        return response.sendStatus(200);
    });
});
/* Put garage */
app.put('/garages/:id', function(request, response){
    let id = request.params.id;
    let updateGarage = request.body;
    if(!isFinite(id)){
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id
must contain only numbers!' });
    }
    if(Object.keys(updateGarage).length < 2){
        return response.status(400).send({ error: true, message: 'Please
provide information about garage in json format.' });
    }
    mc.query('SELECT garageId FROM garage WHERE garageId = ?', [id],
function(error, results, fields){

```

```

        if(Object.keys(results).length == 0){
            return response.status(404).send({error: 'Not Found', message:
'There is no garage with id = ' + id});
        }
        mc.query('UPDATE garage SET ? WHERE garageId = ?', [updateGarage, id] ,
function(error, results, fields){
            if(error) throw error;
            return response.sendStatus(200);
        });
    });
});
/* Patch garage */
app.patch('/garages/:id', function(request, response){
    let id = request.params.id;
    let updateGarage = request.body;
    if(!isFinite(id)){
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id
must contain only numbers!' });
    }
    if(Object.keys(updateGarage).length < 1){
        return response.status(400).send({ error: true, message: 'Please
provide information about garage in json format.' });
    }
    mc.query('SELECT garageId FROM garage WHERE garageId = ?', [id],
function(error, results, fields){
        if(Object.keys(results).length == 0){
            return response.status(404).send({error: 'Not Found', message:
'There is no garage with id = ' + id});
        }
        mc.query('UPDATE garage SET ? WHERE garageId = ?', [updateGarage, id] ,
function(error, results, fields){
            if(error) throw error;
            return response.sendStatus(200);
        });
    });
});
});

```

```

/* Delete garage app */
app.delete('/garages/:id', function(request, response){
    let id = request.params.id;
    if(!isFinite(id)) {
        return response.status(400).send({ error: 'Incorrect id!', message: 'Id
must contain only numbers!' });
    }
    mc.query('SELECT garageId FROM garage WHERE garageId = ?', [id],
function(error, results, fields){
    if(Object.keys(results).length == 0){
        return response.sendStatus(404);
    }
    mc.query('DELETE FROM garage WHERE garageId = ?', [id], function
(error, results, fields) {
        if (error) throw error;
        return response.sendStatus(200);
    });
});
});
app.delete('/garages', function(request, response){
    return response.sendStatus(400);
});

```