МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМЕНІ С.З. ҐЖИЦЬКОГО

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему: “**Дослідження ефективності розробки API додатків реалізованих технологіями REST, GraphQL та gRPC**”

Виконала:
здобувач 6 курсу групи ІТ-62
Спеціальності 126 – «Інформаційні системи та технології»
(шифр і назва)

Сеник Олена Юріївна
(Прізвище та ініціали)

Керівник: к.т.н., доц. Луб П.М.
(Прізвище та ініціали)

Рецензент: _____
(Прізвище та ініціали)

ДУБЛЯНИ-2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМЕНІ С.З. ҐЖИЦЬКОГО

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Другий (магістерський) рівень вищої освіти
Спеціальність 126 «Інформаційні системи та технології»

«ЗАТВЕРДЖУЮ»
Завідувач кафедри

_____
(підпис)
д.т.н., професор Тригуба А. М.
(вч. звання, прізвище, ініціали)
“____” _____2025 року

ЗАВДАННЯ

на кваліфікаційну роботу здобувачу

Сеник Олені Юріївній

1. Тема роботи: “Дослідження ефективності розробки API додатків реалізованих технологіями REST, GraphQL та gRPC”

Керівник роботи: к.т.н., доц. Луб П. М.

Затверджені наказом по університету №140/к-с від 28.02.2025.

2. Строк подання здобувачем роботи: 01.12. 2025 р.

3. Початкові дані: Архітектура API; Специфікація вимог до програмного продукту; Бібліотеки мов програмування; Науково-технічна і довідкова література.

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)

1 Аналіз сучасних підходів до розробки API та постановка завдання дослідження

2 Теоретичні підстави та середовище побудови API

3 Методика розробки проєкту

4 Результати тестування реалізованих технологій

5 Охорона праці та безпека в надзвичайних ситуаціях

Висновки

Бібліографічний список

Додатки

5. Перелік презентаційного матеріалу: *1, 2 – Вступ (тема, мета, завдання), 3 – Актуальність теми та проблематика, 4-6 – Теоретичний огляд, 7 – Критерії оцінювання ефективності, 8-9 – Опис реалізації, 10 – Сценарії тестування, 11-14 – Результати дослідження, 15 – Висновки тестування, 16 – Перспективи подальших досліджень, 17 – Загальні висновки*

6. Консультанти з розділів:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Луб П.М., доцент кафедри інформаційних технологій		
5	Городецький І.М., доцент кафедри інженерної механіки		

7. Дата видачі завдання 28.02.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Відмітка про виконання
1	Аналіз технологій і написання першого розділу	*01.03-01.05.25*	
2	Визначення плану і виконання другого розділу	*01.03-01.05.25*	
3	Реалізація системи і налаштування	*01.05-01.07.25*	
4	Написання третього розділу	*01.05-01.07.25*	
5	Тестування системи і написання четвертого розділу	*01.07-01.09.25*	
6	Написання розділу: «Охорона праці та безпека в надзвичайних ситуаціях»	*01.09-01.11.25*	
7	Завершення роботи вцілому	*01.11-01.12.25*	

Студентка Сеник Олена Юріївна .
(підпис) (Прізвище, ініціали)

Керівник роботи Луб Павло Миронович .
(підпис) (Прізвище, ініціали)

УДК 004.4:004.738.5

Кваліфікаційна робота: 82 с. текст. част., 42 рис., 21 табл., 17 арк. ілюстраційного матеріалу, 30 джерел, 9 додатків.

Дослідження ефективності розробки API додатків реалізованих технологіями REST, GraphQL та gRPC. Сеник О.Ю. Кафедра інформаційних технологій. Дубляни, Львівський НУВМБ, 2025.

Наведено практичне порівняння і оцінювання ефективності розробки API за допомогою технологій REST, GraphQL та gRPC. Представлено коротку реалізацію трьох API з використанням кожної технології. Сформульовано основну задачу, визначено мету дослідження і план роботи.

Об’єкт дослідження – ефективність роботи і розробки API реалізованих технологіями REST, GraphQL та gRPC.

Метою кваліфікаційної роботи є дослідження ефективності розробки API-додатків за допомогою практичної реалізації і аналізу, визначення переваг, недоліків та способів використання залежно від вимог системи.

Визначено показники оцінювання продуктивності системи, вигляд і структура програми. Проаналізовано технології REST, GraphQL та gRPC, представлено їх використання. Постановлено задачі розробки системи і вибрано засоби для реалізації.

Проведено проєктування і реалізацію системи з використанням обраних технологій. Виконано навантажувальне тестування і проаналізовано результати.

Описано висновки стосовно використання і ефективності REST, GraphQL та gRPC.

Ключові слова: API, REST, GraphQL, gRPC, ефективність, продуктивність, тестування.

ЗМІСТ

ВСТУП

Сьогодні розробка програмного забезпечення вимагає ефективних та масштабованих рішень для стабільної та продуктивної роботи. Однією з ключових частин багатьох додатків є API (Application Programming Interface), яка забезпечує взаємодію між клієнтськими додатками, сервісами і базами даних. Вибір технології для реалізації API впливає на продуктивність системи, зручність використання, підтримку тощо.

Через розвиток технологій з'явилось багато способів реалізації API. Одними з найпопулярніших технологій сьогодні є REST, GraphQL і gRPC, які відрізняються архітектурою, ефективністю та іншим.

REST (Representational State Transfer) – це архітектурний стиль на основі HTTP-протоколу, який широко використовується у веброзробці [1].

GraphQL – це мова запитів, в якій клієнт самостійно визначає структуру даних у запиті[2]

gRPC – це фреймворк від Google на основі HTTP/2 та Protocol Buffers, що є дуже ефективним у високонавантажених системах [3].

Ці технології широко застосовуються в багатьох сферах, але порівняння їхньої ефективності у різних умовах є актуальним питанням. Важливими критеріями є продуктивність, зручність використання, можливості масштабування та інше.

Метою роботи є дослідження ефективності розробки API-додатків з використанням технологій REST, GraphQL та gRPC за допомогою практичної реалізації і аналізу результатів. Необхідно визначити їхні переваги, недоліки і способи використання залежно від умов.

Для досягнення мети роботи були поставлені ***наступні завдання***:

- проаналізувати теоретичні основи побудови програмних інтерфейсів та сучасні технології розробки API, зокрема REST, GraphQL та gRPC.

- розкрити методи і середовища дослідження, формування вимог до програмного продукту й обґрунтувати вибір технологій і засобів для його реалізації.

- спроєктувати і створити три API – REST, GraphQL та gRPC – що реалізують однакову функціональність і можливість їх коректного порівняння.

- провести навантажувальне й функціональне тестування розроблених API за допомогою JMeter, Postman та інших засобів, а також виконати оцінку продуктивності та ефективності кожної технології.

Об'єкт дослідження: програмні інтерфейси взаємодії (API), що забезпечують обмін даними між клієнтськими і серверними компонентами програмних систем.

Предмет дослідження: методи, технології та інструменти розробки API, а також ефективність їх реалізації на основі підходів REST, GraphQL та gRPC в умовах практичного застосування і навантажувального тестування.

Новизна роботи полягає у тому, що:

- здійснено практичне порівняння REST, GraphQL та gRPC на основі створення трьох аналогічних API;

- розроблено експериментальну модель оцінювання ефективності API, яка включає критерії часу відгуку, пропускної здатності, стабільності під навантаженням та складності реалізації;

- обґрунтовано рекомендації щодо вибору технології API залежно від типу задачі, очікуваних навантажень та вимог до гнучкості й масштабованості.

Практичне значення: порівняння трьох технологій розробки API реалізується на основі виокремленої задачі. Отримані результати показують, які технології варто обирати за певних умов.

РОЗДІЛ 1
АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ДО РОЗРОБКИ API ТА ПОСТАНОВКА ЗАВДАННЯ ДОСЛІДЖЕННЯ

1.1. Головні поняття та основи розробки API

API (Application Programming Interface) – це набір правил і специфікацій, що дозволяє різним програмним забезпеченням взаємодіяти одне з одним [4]. Можна сказати, що API є посередником між клієнтом і сервером, при цьому жодному з них не потрібно знати будову іншого. Використання API спрощує розробку, тому що замість написання нового продукту можна підключити вже створене API, яке буде надавати необхідний функціонал і дані.

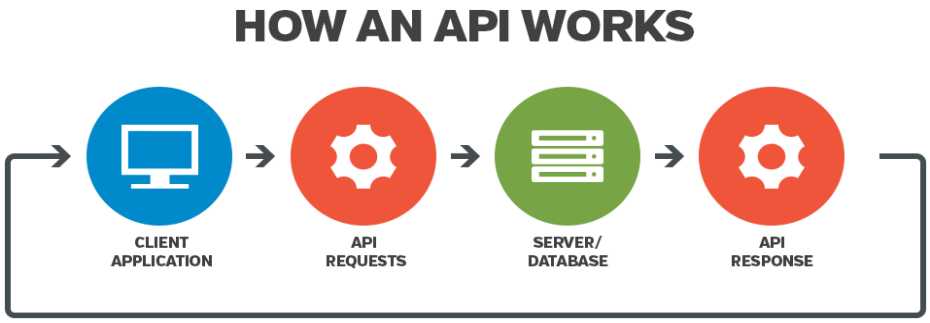

Рисунок 1.1 – Складові, що реалізовують роботу API

Сучасні системи є багатошаровими та складними, тому необхідне розподілення компонент, щоб досягти гнучкості і масштабованості. API можна повторно використовувати, що робить розробку ефективною [5]. Розробники можуть додавати і оновлювати код в API без втручання в інші частини програми. Деякі компанії продають доступ до свого API, щоб розробники могли використовувати дані і функціонал. Це корисно для співпраці компаній і монетизації технологій. Крім цього, API відіграє роль додатково шару для безпеки. Запити, HTTP-заголовки, механізми автентифікації та інші способи створюють додатковий захист для безпечної передачі даних. Також API містить дозволи, що дає можливість користувачам переглядати лише доступний їм контент.

Крім переваг, API має деякі недоліки. Якщо API раптово перестане працювати, наприклад компанії використовують стороннє API, а його закрили, тоді і програма буде неробочою. Також розробники не можуть додати новий функціонал в стороннє API, тому обмежені в роботі [6].

Сьогодні API оточують нас всюди. Поширеними прикладами є додатки з прогнозом погоди, здійснення оплати, швидкі реєстрації тощо, адже ці дії можна виконати з різних пристроїв і сайтів. Користувачеві не потрібно замислюватись як працює система, а можна просто користуватись.

1.2. Аналіз технологій REST, GraphQL та gRPC

Сьогодні існує багато способів створити системи API. Одними з найпоширеніших варіантів реалізації можна назвати REST, GraphQL та gRPC.

REST (Representational State Transfer) – це архітектурний стиль на основі HTTP протоколу для взаємодії систем [1]. REST API надсилає запит за допомогою веб-адреси.

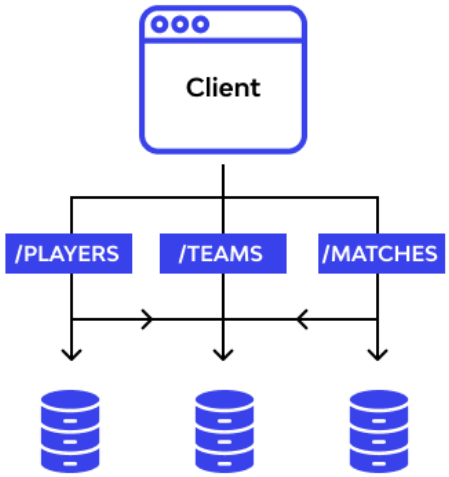

Рисунок 1.2 – Будова REST API

Для передачі даних використовується зазвичай формат JSON, але можливі й інші: HTML, XML тощо. REST API забезпечують єдиний інтерфейс і ідентифікують ресурси за допомогою URI, тобто однакові дані відповідають

лише одному ідентифікатору. Кожен запит має містити необхідну інформацію для його обробки. REST широко застосовується через свою простоту і підтримуваність, але має й недоліки з продуктивністю і гнучкістю.

В REST API використовують п’ять методів для обробки даних: GET, PUT, POST, PATCH, DELETE [7]. Кожен з них повертає статус відповіді і обмін даними відбувається у форматі JSON, але можливі й інші формати.

Метод GET використовують для отримання даних, наприклад списку користувачів або інформації про одного користувача. Щоб отримати інформацію про певне замовлення, код може виглядати так:

GET /orders/1

Дані, які будуть повертати у відповідь на запит можуть мати приблизно таку структуру:

```
{
 "id": 1,
 "name": "Order #1",
 "user": {
  "id": 3,
  "username": "Olena"
 }
}
```

Метод POST створює новий рядок з даними. Для того, щоб оновити дані використовують PUT або PATCH. Різниця між цими запитами полягає в кількості оновлених даних, тобто метод PUT передає відредагований об’єкт з усіма полями, при цьому PATCH може передати лише одне поле.

Останнім важливим методом є DELETE, який видаляє дані, наприклад усіх користувачів або за ідентифікатором.

GraphQL – це мова запитів для API. Вона вирішує проблему гнучкості REST, адже вона має строго типізовану схему для взаємозв’язку даних [2]. Цей інструмент дозволяє клієнту самостійно визначати, які дані отримати. GraphQL підтримує запити, мутації, тобто зміну даних, і підписки, що дозволяє отримувати дані в реальному часі. На стороні клієнта ця мова спрощує формування запитів, але при цьому є складною до реалізації на стороні сервера. Крім цього, можуть виникнути складнощі з безпекою і кешуванням даних.

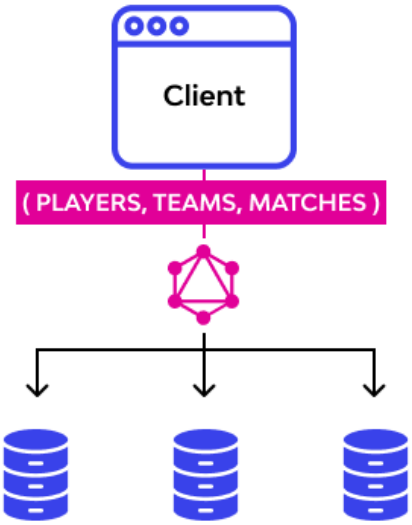

Рисунок 1.3 – Вигляд GraphQL API

GraphQL має схеми, які визначають структуру API, тобто в них описано типи і запити. Прикладом схеми може бути інтерфейс, клас чи тип:

```
type Order {
  id: ID!
  name: String!
  user: User!
}
```

Типи у GraphQL визначають, які поля користувач може обрати для отримання. У схемах описують запити, мутації і підписки. Наприклад, отримання списку замовлень або одного замовлення за ідентифікатором можна визначити наступним чином:

```
type Query {
  orders: [Order!]!
  order(id: ID!): Order
}
```

Для того, щоб створити замовлення, потрібно вказати схему мутації, так як саме вони відповідають за оновлення і створення даних:

```
type Mutation {
  createOrder(name: String!, userId: ID!): Order!
}
```

Підхід, в якому клієнт сам визначає, які дані отримати, може зменшити кількість зайвої інформації, а це пришвидшує відповіді сервера і збільшує ефективність API.

gRPC – це фреймворк для виклику віддалених процедур (RPC) на основі бінарного формату [3]. Він підтримує потокові запити, щоб клієнт і сервер могли швидко обмінюватись даними, тобто немає необхідності встановлювати зв'язок кожного разу. Це корисно для програм, які транслюють відео в режимі реального часу, містять чати, онлайн-ігри тощо. Опис схем схожий на GraphQL, але менш гнучкий, так як не можна вибрати поля. Також gRPC автоматично генерує код для різних мов програмування. На відміну від попередніх технологій, в gPRC клієнт і сервер мають мати доступ до .proto файлу. Тобто під час оновлення API потрібно оновлювати код і на сервері, і на клієнті. Цей фреймворк не дуже зручний і складний в роботі, але обіцяє високу продуктивність. gRPC поки не є дуже поширеним в порівнянні з REST і GraphQL.

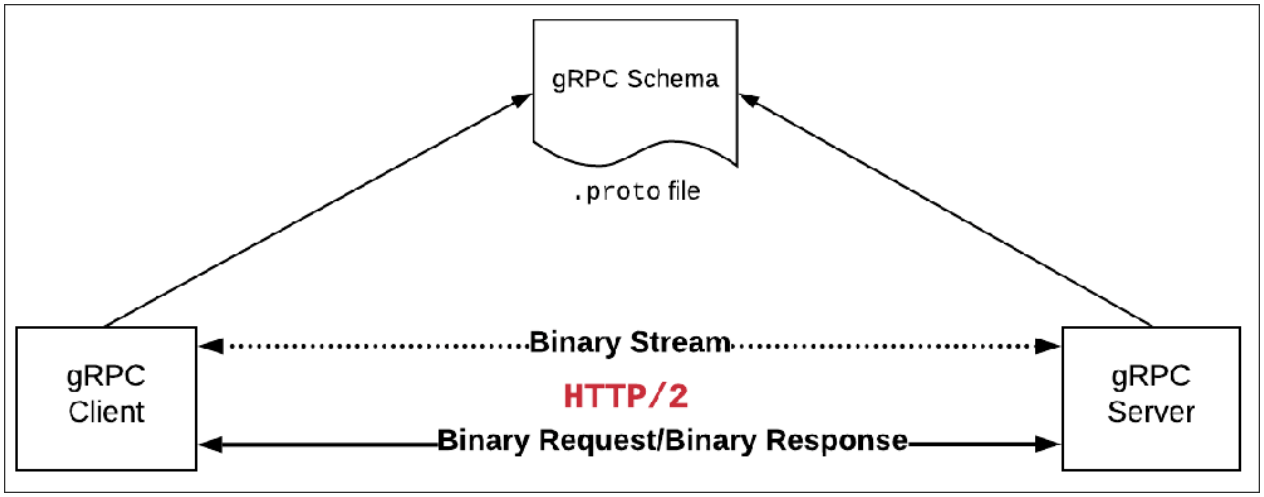

Рисунок 1.4 – Елементи будови gRPC API

Серед згаданих технологій gRPC є найскладнішим для розуміння. Усі методи і запити описуються в файлах ".proto". На основі цих файлів генерується код для сервера і клієнта. Сервіс для обробки замовлень можна описати наступним чином:

```
service OrderService {
  rpc GetOrder (OrderRequest) returns (OrderResponse);
  rpc CreateOrder (CreateOrderRequest) returns (OrderResponse);
}
```

В цьому прикладі є метод для отримання інформації про замовлення і створення нового.

gRPC підтримує різні види комунікації, тому його зручно використовувати в різних системах. Але через складну реалізацію він є менш поширеним.

1.3. Використання технологій для реальних практичних завдань

REST є дуже популярною архітектурою через простоту і сумісність. Часто цю архітектуру використовують для веб-сайтів і мобільних додатків. REST API підходить для більшості завдань, але все ж з обробкою великих і складних даних виникають труднощі. Архітектура використовується в таких відомих компаніях як Instagram, X, Amazon [8]. Instagram API дає доступ до профілів користувачів, фото і відео. AWS AI Services може додати функції AI в додатки, а API від компанії X спрощує реєстрацію, відображає пости та інше.

GraphQL орієнтована на сервіси, де потрібний гнучкий доступ до даних. Він також використовується для сайтів, мобільних додатків і систем, де є велика кількість пов’язаної інформації. Ця мова використовується в LinkedIn, Meta, Netflix [9]. Компанія Meta переписала додаток Facebook, щоб використовувати GraphQL замість REST. Це допомогло їм оптимізувати дані і пришвидшити роботу програми. Netflix також перейшли на цю мову, щоб завантаження даних проходило легше і спричиняло менше проблем із пропускною здатністю. GraphQL у LinkedIn об’єднує схеми сутностей з Rest.li (їх власний фреймворк для мікросервісів), а запити виконуються без центрального сервісу, завдяки чому підвищується продуктивність і безпека.

gRPC використовується в системах, де є акцент на продуктивність і швидкий обмін даними. Він підходить для мікросервісної архітектури, потокових сервісів і високонавантажених систем. gRPC широко використовується компанією Google, наприклад, в хмарних системах для швидкого отримання даних. Також фреймворк використовує Uber для обміну повідомленнями, тому що сповіщення зменшує затримку з’єднання [10].

1.4. Аналіз предметної області та проблематики

API є ключовим компонентом для взаємодії клієнтських додатків, серверів і сторонніх платформ, тому предметною областю дослідження є розробка та інтеграція API в системах. Серед основних проблем необхідно дослідити наступні:

– продуктивність. Вона впливає на швидкість обробки запитів і відповіді. При великій кількості даних, складних операціях або одночасних запитах може бути затримка у відповіді, тому необхідна оптимізація для забезпечення продуктивності.

– масштабованість. Вона визначає здатність системи ефективно обробляти більшу кількість запитів і сприймати навантаження.

– обробка великих даних. В більшості додатків використовується велика кількість даних. API має ефективно обробляти різні обсяги запитів і вкладені дані, щоб були мінімальні затримки.

– безпека. Багато API зберігають конфіденційні дані, тому важливо перевірити неможливість несанкціонованого використання, безпечну авторизацію і аутентифікацію.

– гнучкість. Часто виникає необхідність розширити систему, дописати нові запити й додати нову обробку даних. API має містити незалежні запити для зручного використання і розширення.

– зручність. Хороша, зрозуміла документація спрощує розробку API. Крім цього, важливий зрозумілий синтаксис без складних налаштувань. Простота розробки зменшує кількість помилок і час на написання.

1.5. Постановка задачі дослідження

З теоретичних відомостей можна зробити висновок, що необхідно порівняти ефективність різних технологій для створення API. Для того, щоб

створити якісні програми, застосунки чи інформаційні системи API має бути високопродуктивним, безпечним, зручним і гнучким. Кожна зі згаданих технологій (REST, GraphQL, gRPC) має свої переваги і недоліки, тому складно вибрати оптимальну технологію для розробки певного програмного рішення.

Метою дослідження є аналіз і практичне порівняння ефективності REST, GraphQL та gRPC при створенні API для обробки даних у додатках.

Серед поставлених задач можна виділити наступні:

1. Огляд основ API і способи реалізації.
2. Аналіз технологій REST, GraphQL та gRPC (особливості і сфери застосування).
3. Недоліки і проблеми API у сучасних інформаційних системах.
4. Визначення методів дослідження, метрик і показників для порівняння ефективності API.
5. Створення API за допомогою REST, GraphQL та gRPC.
6. Проведення експериментальних досліджень, порівняння результатів згідно метрик і показників.
7. Формулювання висновків щодо застосування REST, GraphQL та gRPC.

РОЗДІЛ 2
ТЕОРЕТИЧНІ ПІДСТАВИ ТА СЕРЕДОВИЩЕ ПОБУДОВИ API

2.1. Головні етапи побудови API

Для виконання дослідження з технологіями REST, GraphQL і gRPC потрібно зробити їх практичне порівняння. Першим кроком дослідження був аналіз технологій REST, GraphQL та gRPC, визначення переваг, недоліків, способів застосування тощо. Необхідно визначити метрики та показники, на основі яких буде сформовано висновки. До метрик можна віднести наступні:

– Response Time (час відповіді) – вимірює швидкість обробки запиту. За допомогою цього показника можна визначити продуктивність. В таблицю можна додати максимальний, мінімальний і середній час відповіді;

– Throughput (пропускна здатність) - вимірює кількість запитів за секунду. Ця метрика визначає масштабованість системи;

– Error Rate (кількість помилок) – вимірює кількість запитів, які повернули помилку;

– Development Time (час розробки) – показує скільки часу потрібно для створення API з однаковим функціоналом. Цей показник відображає зручність реалізації з використанням певної технології;

– Payload Size (розмір даних) – вимірює обсяг інформації, яку повертає запит. За допомогою цієї метрики можна побачити навантаженість на систему;

– Lines of Code (кількість коду) – показує скільки коду необхідно для реалізації однакової функції за допомогою різних технологій. Цей показник також визначає зручність розробки.

Також необхідно визначити, які показники можна отримати за допомогою різноманітних програм для тестування API, наприклад Postman або Apache JMeter. Вони також можуть відображати роботу API у вигляді графіків, що буде корисно для порівняння.

Для практичного порівняння спочатку необхідно розробити API на основі поданих технологій, яке буде містити однакову логіку, щоб умови роботи були однаковими. Для дослідження буде створено просте API для обробки замовлень користувачів. В ньому будуть запити для різного навантаження системи: створення замовлень і редагування (для перевірки роботи зі складених даних), отримання багатьох замовлень (для визначення продуктивності при великій кількості даних), видалення (для перевірки стабільності) тощо.

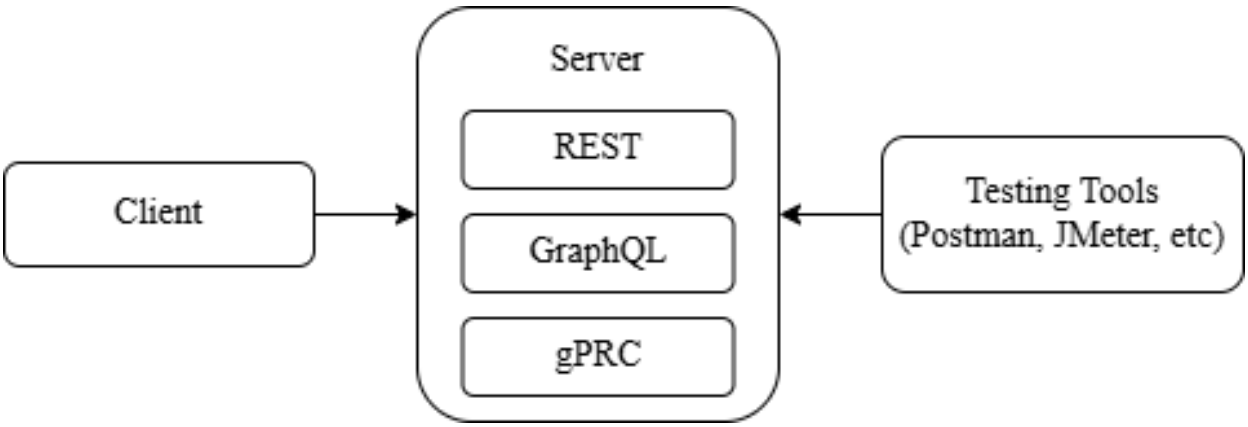

Рис 2.1 – Структура проєкту

Для кожної технології буде використано однакові тестові дані, щоб правильно оцінити результати. Після розробки буде проведено тестування у різних інструментах, щоб оцінити швидкість роботи. Всі результати будуть представлені у вигляді таблиць, графіків і діаграм, щоб візуально оцінити і проаналізувати роботу. Це допоможе сформулювати висновки на основі метрик щодо наступних критеріїв: продуктивність, масштабованість, гнучкість і зручність розробки.

Описаний метод дослідження допоможе реально оцінити ефективність розробки API з використанням REST, GraphQL, gRPC і зробити висновки щодо доцільності їх використання в різних умовах.

2.2. Специфікація вимог до програмного продукту

2.2.1. Призначення, мета

Ця система створена на основі технологій REST, GraphQL і gRPC для тестування ефективності API. Вона містить запити CRUD операцій для оцінювання технологій за певними критеріями. Система використовуватиметься для практичного порівняння сучасних технологій створення API.

2.2.2. Загальний опис

2.2.2.1 Характеристики продукту

Система надає можливість створювати, оновлювати, видаляти і переглядати замовлення користувачів. Для кожної технології REST, GraphQL та gRPC реалізовано однакову бізнес-логіку. Серед основних функцій також є вимірювання різних показників для оцінювання ефективності API. Система містить перелік функцій для тестування технологій.

2.2.2.2 Середовище функціонування

Операційні системи: Windows, Linux, Mac OS, IOS, Android.

Вебпереглядачі: Google Chrome, Mozilla Firefox, Opera, Safari, Microsoft Edge.

2.2.3. Характеристики системи

2.2.3.1 Керування замовленнями

2.2.3.1.1 Опис і пріоритет:

програмне забезпечення дає можливість створювати, редагувати, переглядати та видаляти замовлення користувачів.

Пріоритет: високий.

2.2.3.1.2 Послідовності дія/відгук:

користувач надсилає запит через API (REST, GraphQL або gRPC). При переході на відповідну сторінку користувач бачить перелік замовлень, де може створити нове, редагувати або видалити наявне.

2.2.3.1.3 Функціональні вимоги:

REQ-2.2.3.1.3.1: створення нового замовлення;

REQ-2.2.3.1.3.2: редагування замовлення;

REQ-2.2.3.1.3.3: видалення замовлення;

REQ-2.2.3.1.3.4: отримання списку всіх замовлень;

REQ-2.2.3.1.3.5: отримання інформації про конкретне замовлення;

REQ-2.2.3.1.3.6: отримання статистики замовлень.

2.2.3.2 Керування продуктами

2.2.3.2.1 Опис і пріоритет:

система дозволяє створювати продукти, керувати ними, зберігати.

Пріоритет: високий.

2.2.3.2.2 Послідовності дія/відгук:

при переході на відповідну сторінку адміністратор може створити продукт і переглядати його.

2.2.3.2.3 Функціональні вимоги:

REQ-2.2.3.2.3.1: створення нового продукта;

REQ-2.2.3.2.3.2: редагування даних продукта;

REQ-2.2.3.2.3.3: видалення продукта;

REQ-2.2.3.2.3.4: перегляд списку продуктів.

2.2.3.3 Аналітика продуктивності

2.2.3.3.1 Опис і пріоритет:

програма забезпечує збір даних для оцінки ефективності роботи API.

Пріоритет: високий.

2.2.3.3.2 Послідовності дія/відгук:

під час виконання запитів система фіксує час відповіді кожного з них, кількість успішних/помилкових запитів тощо. Результати тестів відображаються у вигляді діаграм.

2.2.3.3.3 Функціональні вимоги:

REQ-2.2.3.3.3.1: вимірювання часу відповіді кожного запиту;

REQ-2.2.3.3.3.2: фіксація навантаження на систему;

REQ-2.2.3.3.3.3: збір та виведення статистики у вигляді діаграм.

2.2.4. Вимоги зовнішніх інтерфейсів

2.2.4.1 Програмні інтерфейси

Програмні інтерфейси, які можуть бути застосовані:

– операційні системи: Windows, Mac OS, Android, iOS, Linux;

– вебпереглядачі: Google Chrome, Mozilla Firefox, Opera, Microsoft Edge, Safari.

2.2.4.2 Комунікаційні інтерфейси

Протоколи, які використовуватимуться:

– HTTP – для передачі даних через вебсторінки;

– FTP – для отримання файлів.

2.2.5. Інші нефункційні вимоги

2.2.5.1 Вимоги продуктивності

Часові рамки продуктивності продукту:

– час завантаження і відображення сторінки має займати максимум 5 секунд;

– час завантаження і відображення даних на запит користувача має займати максимум 5 секунд;

– збереження і оновлення інформації в базах даних має займати максимум 5 секунд.

Система має обробляти близько 100 одночасних запитів без сильної затримки.

2.2.5.2 Вимоги надійності

У випадку помилки сервер має повертати чітку і зрозумілу відповідь зі статусом та описом.

2.2.5.3 Вимоги гнучкості

Система має дозволяти легко додавати нові запити без зміни основної логіки і архітектури.

2.2.6. Інші вимоги

Програмне забезпечення повинне відповідати усім стандартам згідно чинного законодавства України станом на 15.10.2025.

Програмне забезпечення має бути реалізоване з використанням мови C# і технологій REST, GraphQL, gRPC.

2.3. Вибір засобів розробки і тестування

Для розробки API буде використано різні програми, бібліотеки і фреймворки для ефективної та надійної роботи:

– C# – об’єктно-орієнтована мова програмування, яка використовується для розробки серверних додатків на платформі .NET [11, 12];

– Apache JMeter – це програма для навантажувального тестування функціоналу і вимірювання продуктивності [13]. Apache дозволяє створювати плани тестування, отримувати звіт, відображати результати у вигляді графіків;

– Postman – це платформа, яка дозволяє проектувати, тестувати і керувати API [14]. На ній можна виконувати будь-які запити і складені також. Вона підтримує різні формати даних. Якщо встановити додаткові розширення, то можна візуалізовувати дані API у вигляді діаграм.

– Visual Studio – це платформа, яка використовується для написання програм [15]. В ній можна створювати і редагувати код. Visual Studio має багато різноманітного функціоналу для зручної розробки: редактори, компілятори, графічні дизайнери тощо.

РОЗДІЛ 3
МЕТОДИКА РОЗРОБКИ ПРОЄКТУ

3.1. Проєктування API

Перш ніж перейти до реалізації програмного рішення за допомогою заданих технологій, необхідно визначити структуру API. Порівняння відбуватиметься на основі простої логіки створення замовлення. Завдяки простим і класичним моделям можна легко зрозуміти бізнес-процеси проєкту. Для початку визначимо сутності для програми:

a) Base – це базова модель, яка містить поля, необхідні для кожної сутності: Id (ідентифікатор), CreatedAt (час створення), UpdatedAt (час оновлення);
b) User – це модель користувача, якому належатимуть замовлення. Міститиме найпростіші поля імені (FullName) та електронної пошти (Email);
c) User Address – це модель адреси користувача. Використовуватиметься для відображення роботи з вкладеними структурами. Вона буде складатись з міста (City) і вулиці (Street);
d) Product – це модель продукту, який можна замовити. Його основними полями буде назва (Name), опис (Description) і ціна (Price);
e) Category – це проста модель категорії продукта, яка теж показуватиме роботу з вкладеними даними;
f) Order Item – це модель рядка замовлення, в якій буде знаходитись кількість (Quantity), загальна ціна (TotalPrice) і посилання на продукт;
g) Order – це основна модель самого замовлення, яка міститиме згадані вище сутності.

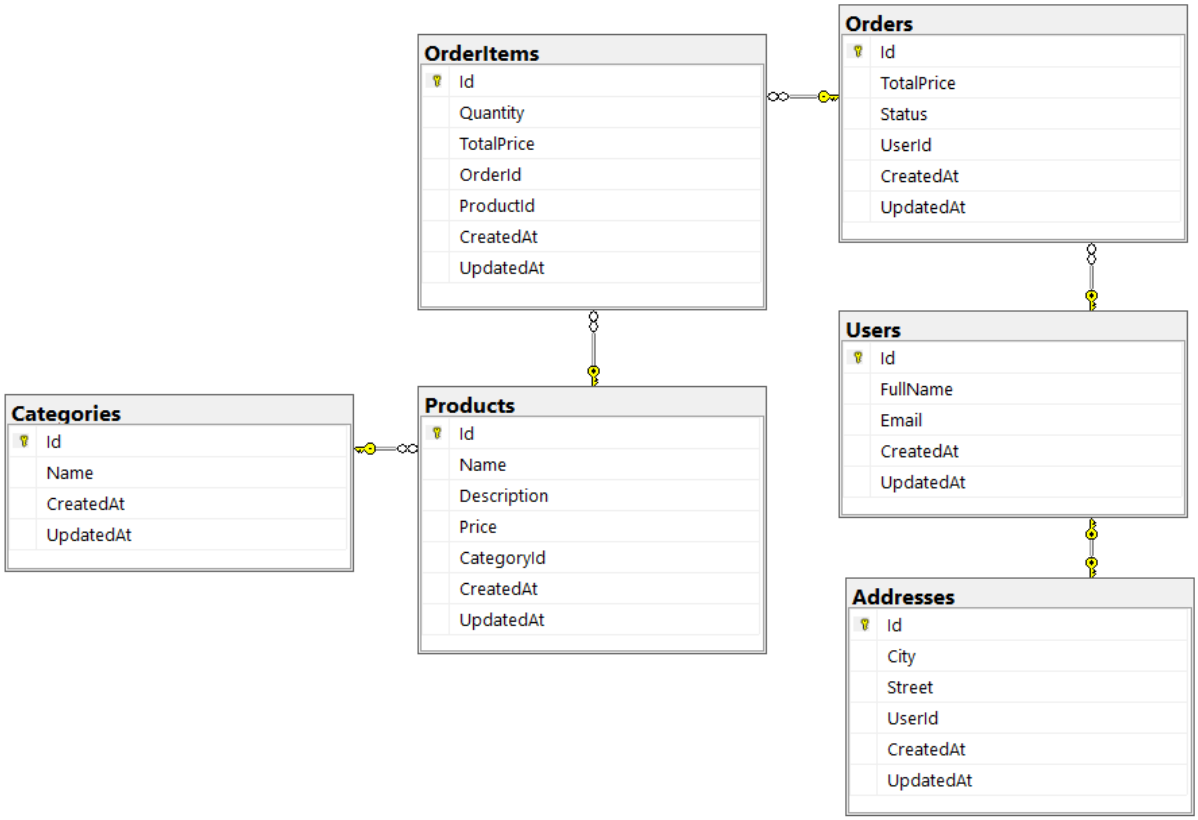

Рисунок 3.1 – Діаграма бази даних на основі сутностей

На рис. 3.1 зображено діаграму бази даних, яка демонструє сутності, згаданими вище. На ній присутні усі види зв’язків: один до одного (Addresses — Users), один до багатьох (Categories — Products, Products — OrderItems, Users — Orders, Orders — OrderItems), багато до багатьох (Products — Orders через проміжну таблицю OrderItems).

Наступні етапи проєктування: набір ендпоінтів, обробка помилок, формати даних – буде представлено в наступних підрозділах під час реалізації.

3.2. Методика створення REST API

Для виконання дослідження спочатку було реалізовано API за допомогою технології REST [16]. Цей архітектурний стиль на основі протоколу HTTP має певні правила та стандарти, які роблять систему масштабованою, гнучкою і

простою для використання. В REST API чітко розділяється сервер і клієнт, тому другому не потрібно знати внутрішнє наповнення першого.

Під час створення було дотримано основних принципів REST-архітектури: єдиний інтерфейс доступу, режим без стану, багатошаровість тощо.

Усі запити розміщені на своїх маршрутах, назви яких визначені за загальноприйнятими правилами. Наприклад, шляхи для запитів замовлень виглядають так:

– (GET) /api/orders – отримання всіх замовлень;

– (GET) /api/orders/{id} – отримання одного замовлення за його ідентифікатором;

– (GET) /api/orders/stats – отримання статистики замовлень;

– (POST) /api/orders – створення нового замовлення.

– (PATCH) /api/orders/{id} – часткове оновлення замовлення;

– (DELETE) /api/orders/{id} – видалення замовлення.

Інформація повертається у форматі JSON. Взаємодія відбувається через HTTP-методи. Серед найпоширеніших можна виділити наступні:

– GET – отримання даних;

– POST – створення нової моделі;

– PUT – оновлення всієї моделі;

– PATCH – оновлення частини моделі;

– DELETE – видалення.

Для зручності проєкт містить шари: Models (усі сутності, які визначають базу даних), Shared (моделі DTO для створення/оновлення, enums та інші допоміжні файли), Services (основна логіка взаємодії з базою даних і обробка інформації) і Controllers (API-шар з мінімальною бізнес-логікою, в якому отримуються запити і передаються дані до сервісів). Такий розподіл допомагає підтримувати систему, легко її розширювати і швидко знаходити необхідні частини коду. Вигляд структури створеної програми за допомогою REST API зображено на рис. 3.2.

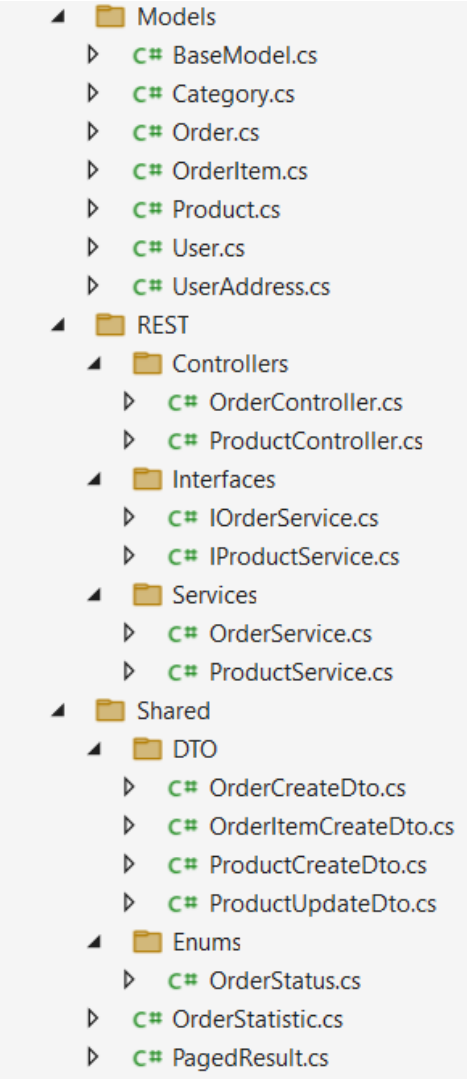

Рисунок 3.2 – Реалізована структура REST API

В цьому розділі реалізацію REST API буде зображено на основі моделі Orders (інші моделі знаходяться в Додатку А).

Для зручності я створила модель Base, в якій міститься інформація, яка стосується кожної моделі. Сюди також можна додати інформацію про користувача, який вносив зміни.

```
public abstract class BaseModel
{
   public int Id { get; set; }
   public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
   public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;
}
```

В моделі Order за допомогою Атрибутів валідації даних (Data Annotations) можна визначити поля і валідацію. Атрибут [Required] вказує, що відповідне

поле не може бути нульовим. Це допомагає перевіряти вхідні дані. Поля User і Items відіграють роль навігації, тобто встановлюють зв’язки між сутностями.

```
public class Order : BaseModel
{
   [Required]
   public decimal TotalPrice { get; set; }
   [Required]
   public required OrderStatus Status { get; set; } = OrderStatus.New;
   [Required]
   public int UserId { get; set; }

   public User? User { get; set; }
   public ICollection<OrderItem> Items { get; set; } = new List<OrderItem>();
}
```

Уся бізнес-логіка винесена в сервіси. Це допомагає зробити контролери зрозумілими і не перевантаженими. Також для сервісів визначено інтерфейси. Сервіси взаємодіють з контекстом через AppDbContext. Усі інтерфейси розміщено в Додатку Б, сервіси – в Додатку В.

Приклад інтерфейсу:

```
public interface IOrderService
{
   Task<IEnumerable<Order>> GetAll();
   Task<Order?> GetById(int id);
   Task<PagedResult<Order>> GetByFilter(OrderStatus? status, DateTime? dateFrom, DateTime? dateTo, string? sortBy, string? sortDir, int pageSize, int page);
   Task<OrderStatistic> GetStats();
   Task<Order> Create(OrderCreateDto order);
}
```

Приклад визначеного запиту з сервісу:

```
public async Task<IEnumerable<Order>> GetAll()
{
   return await _db.Orders
         .Include(o => o.User)
         .ToListAsync();
}
```

Для передачі даних між шарами часто використовують DTO (Data Transfer Object) для відокремлення моделей і безпечної передачі даних. В моїй програмі DTO використовуються для створення і оновлення сутностей, щоб користувач не зачепив дані, до яких не має мати доступ.

У сервісі відбувається валідація всередині коду (наприклад, перевірка на наявність моделі при знаходженні за ідентифікатором). Також проста валідація знаходиться всередині самих сутностей.

Для кожної сутності створено окремий контролер (Додаток Г) для кращого орієнтування. Вони обробляють запити і повертають результати.

Приклад оголошення контролера:

```
[ApiController]
[Route("api/orders")]
[Produces("application/json")]
public class OrderController : ControllerBase
```

Приклад використання функції GetAll з сервісу в контролері для отримання списку всіх замовлень:

```
[HttpGet]
[ProducesResponseType(StatusCodes.Status200OK,Type = typeof(IEnumerable<Order>))]
public async Task<ActionResult<IEnumerable<Order>>> GetAllOrders()
{
   var orders = await _orderService.GetAll();
   return Ok(orders);
}
```

В контролерах визначено відповіді, які повертатимуться залежно від результату обробки даних. Наприклад:

– 200 OK – успішний запит;

– 201 Created – ресурс створено;

– 400 Bad Request – помилка валідації, неправильний формат тощо;

– 404 Not Found – ресурс не знайдено;

– 500 Internal Server Error – помилки сервера.

Для зручного використання API було інтегровано Swagger, зображений на рис. 3.3, який показує JSON дані, допомагає тестувати програму, відображає всі запити в зручному вигляді тощо. Його підключення відбувається в конфігурація до проєкту (Додаток Д):

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
app.UseSwagger();
app.UseSwaggerUI();
```

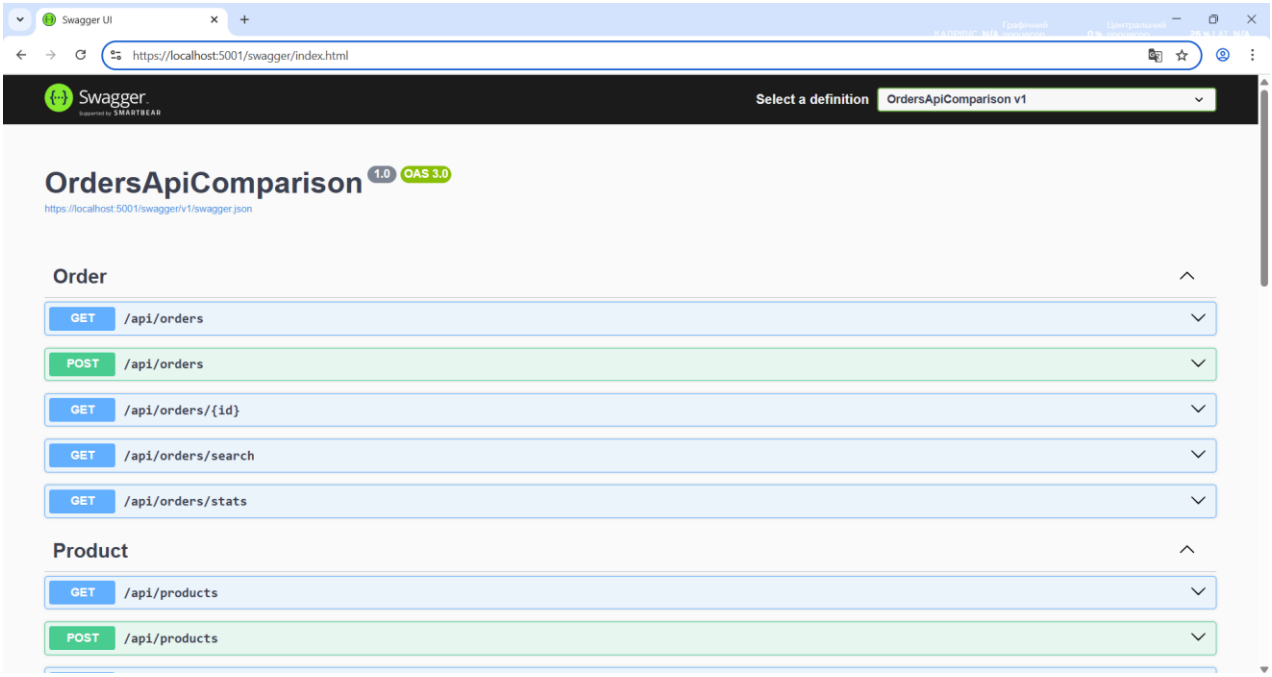

Рисунок 3.3 – Головне вікно Swagger

REST API є простим і зручним в реалізації. Документація зрозуміла і кількість статей і відео дуже велика, що значно спрощує розробку.

3.3. Методика створення GraphQL API

Наступною технологією для реалізації API є GraphQL. Вона є гнучкішою під час роботи з даними, зменшує кількість запитів, їх розмір і цим покращує продуктивність програми. GraphQL не має ендпоінти, а використовує один шлях, наприклад /graphql.

У проєкті було використано бібліотеку HotChocolate для реалізації GraphQL. Вона є найпоширенішою для створення GraphQL API у середовищі .NET[17].

Для початку необхідно описати моделі, поля, зв’язки тощо. У GraphQL не потрібно визначати структуру відповіді, адже користувач може сам обирати потрібні поля. Це допомагає уникнути зайвої великої кількості даних або нестачі даних (коли приходиться робити декілька запитів, щоб отримати всі дані). Ця ж

перевага може бути недоліком, адже користувач бачить всю схему моделі, що може впливати на безпеку.

В GraphQL існує три основних типи: Query (отримання даних), Mutation (зміна даних), Subscription (підписка). В межах дослідження розглянемо перші два.

Реалізацію API було здійснено на основі моделей, згаданих в REST API. В цій технології також використовується папка Shared з усіма enums, DTO тощо. Для справедливих результатів було реалізовано однаковий функціонал для усіх технологій.

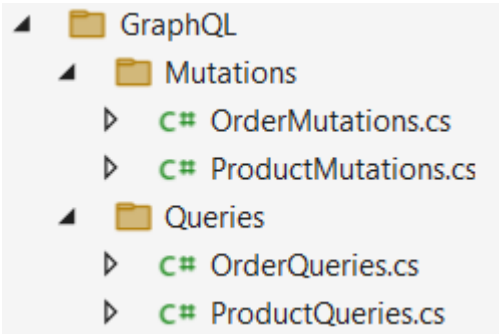

Рисунок 3.4 – Реалізована структура GraphQL

Підключення GraphQL відбувається у файлі Program.cs, де вказано запити, мутації, а також підтримку фільтрації, сортування, пагінації тощо (додаткові можливості HotChocolate) [18]. Окремі класи для запитів і мутацій дозволяють розділити логіку для зручності, але під час підключення їх об’єднано в одну схему.

```
builder.Services
    .AddGraphQLServer()
    .AddQueryType(d => d.Name("Query"))
        .AddTypeExtension<OrderQueries>()
        .AddTypeExtension<ProductQueries>()
    .AddMutationType(d => d.Name("Mutation"))
        .AddTypeExtension<OrderMutations>()
        .AddTypeExtension<ProductMutations>()
    .AddFiltering()
    .AddSorting()
    .AddProjections();
```

Отримання даних знаходиться в папці Queries (Додаток Е) і розбито на декілька класів. Перед кожним запитом використанні певні атрибути для розширення можливостей:

– UseFiltering – додає логіку для фільтрації даних на основі полів моделі;

– UseSorting – дозволяє сортувати дані за певним полем і напрямком;

– UseOffsetPaging – додає автоматичне підтримування пагінації запитів;

– UseProjection – гарантує, що повернуться лише запитувані поля.

Приклад класу з функціями для отримання усіх замовлень і одного:

```
[ExtendObjectType("Query")]
public class OrderQueries
{
    [UseDbContext(typeof(AppDbContext))]
    [UseOffsetPaging]
    [UseProjection]
    [UseFiltering]
    [UseSorting]
    public IQueryable<Order> GetOrders([ScopedService] AppDbContext context)
    {
        return context.Orders.AsNoTracking();
    }

    [UseDbContext(typeof(AppDbContext))]
    [UseProjection]
    public async Task<Order?> GetOrderById(int id, [ScopedService] AppDbContext context)
    {
        return await context.Orders.FindAsync(id);
    }
}
```

Під час роботи з запитом можна визначити поля для отримання. В запиті не буде вкладених структур, якщо ми цього не задамо власноруч.

```
query orders ($skip: Int, $take: Int, $where: OrderFilterInput) {
  orders(skip: $skip, take: $take, where: $where) {
    items {
      createdAt
      id
      status
      totalPrice
      updatedAt
      userId
    }
    pageInfo {
      hasNextPage
      hasPreviousPage
    }
```

```
    }
}
```

Мутації використовуються для створення, оновлення і видалення даних. У проєкті вони винесені в окрему папку Mutations (Додаток Ж). Приклад класу з функціями створення і видалення замовлення:

```
[ExtendObjectType("Mutation")]
 public class OrderMutations
 {
    [UseDbContext(typeof(AppDbContext))]
    public async Task<Order> CreateOrder(OrderCreateDto orderDto, [ScopedService] AppDbContext context)
    {
       var order = new Order
       {
          TotalPrice = orderDto.TotalPrice,
          Status = orderDto.Status,
          UserId = orderDto.UserId,
          Items = orderDto.Items.Select(
             i => new OrderItem
             {
                ProductId = i.ProductId,
                Quantity = i.Quantity,
                TotalPrice = i.TotalPrice
             }
          ).ToList()
       };

       context.Orders.Add(order);
       await context.SaveChangesAsync();
       return order;
    }

    [UseDbContext(typeof(AppDbContext))]
    public async Task<bool> DeleteOrder(int id, [ScopedService] AppDbContext context)
    {
       var order = await context.Orders.FindAsync(id);
       if (order == null)
       {
          return false;
       }

       context.Orders.Remove(order);
       await context.SaveChangesAsync();
       return true;
    }
 }
```

Завдяки Nitro інтерфейсу можна зручно керувати всіма запитами. Цей UI дозволяє створювати папки з запитами, додавати поля за допомогою Builder, визначати заголовки запитів. Також тут можна переглянути всі схеми і типи.

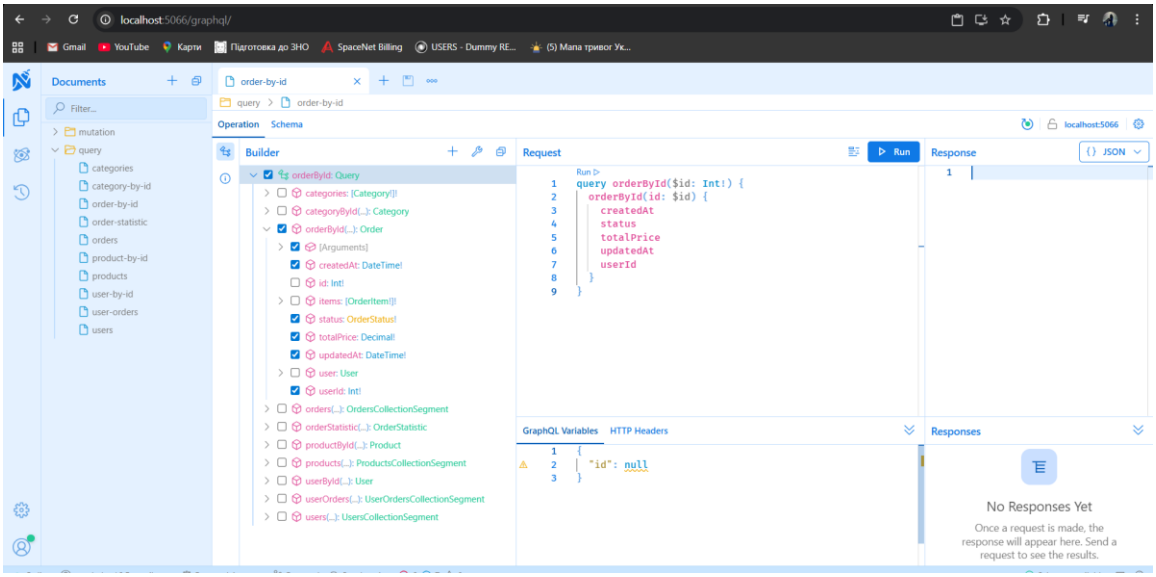

Рисунок 3.5 – Інтерфейс Nitro для керування запитами GraphQL

Керування запитами в GraphQL є досить зручним завдяки інтерфейсу. Об’ємна документація і бібліотеки пришвидшують розробку. Незважаючи на те, що REST є більш поширеним, GraphQL також може бути дуже зручним інструментом, якщо витратити час на його вивчення.

3.4. Методика створення gRPC API

Серед трьох технологій саме gRPC має зовсім інший стиль опису. Цей фреймворк є високопродуктивним, тому що використовує буфери протоколу. Завдяки ним запити передаються в малому за розміром форматі, що збільшує швидкість роботи. Цю технологію варто використовувати у мікросервісних архітектурах, де потрібна велика кількість викликів.

Для роботи з gRPC потрібно встановити пакети: Grpc.AspNetCore, Google.Protobuf, Grpc.Tools.

gRPC є зручним завдяки автоматичній генерації коду для сервера і клієнта. Уся структура коду (моделі, сервіси, DTO тощо) визначається в спеціальному файлі .proto [19] (Додаток И):

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
import "google/protobuf/empty.proto";

option csharp_namespace = "OrdersApiComparison.gRPC.Protos";

package orders;

service OrderService {
rpc GetOrders (google.protobuf.Empty) returns (GetOrdersResponse);
rpc GetOrderById (GetByIdRequest) returns (OrderModel);
rpc GetByFilter (GetByFilterRequest) returns (PagedOrderResult);
rpc GetStats (google.protobuf.Empty) returns(OrderStatisticModel);
rpc CreateOrder (CreateOrderRequest) returns (OrderModel);
}

message OrderModel {
   int32 id = 1;
   string total_price = 2;
   string status = 3;
   google.protobuf.Timestamp created_at = 4;
   google.protobuf.Timestamp updated_at = 5;
   UserModel user = 6;
}
```

Важливо, що файл .proto потрібно додати у сам проєкт (файл .csproj), щоб код генерувався автоматично:

```
<ItemGroup>
        <Protobuf Include="gRPC\Protos\order.proto" GrpcServices="Server" />
</ItemGroup>
```

Після визначення структури сервісів і моделей відбувається реалізація сервісу (Додаток К):

```
public class OrderGrpcService : OrderService.OrderServiceBase
{
   private readonly AppDbContext _db;
   public OrderGrpcService(AppDbContext db)
   {
      _db = db;
   }

   public override async Task<GetOrdersResponse> GetOrders(Empty request, ServerCallContext context)
   {
      var orders = await _db.Orders
         .Include(o => o.User)
```

```
            .ThenInclude(u => u.Address)
        .Include(o => o.Items)
            .ThenInclude(i => i.Product)
                .ThenInclude(p => p.Category)
        .ToListAsync();
    var response = new GetOrdersResponse();
    response.Orders.AddRange(orders.Select(MapOrderDetails));
    return response;
  }
}
```

Серед усіх технологій gRPC є найскладнішою для розуміння і має найменшу документацію. Її опис є досить довгим і мапінг виглядає неоптимізованим і непродуктивним.

3.5. Налаштування JMeter для навантажувального тестування

Для проведення навантажувального тестування зручним інструментом є JMeter, тому що він може показувати різні параметри (час відповіді, розмір, пропускну здатність) і будувати таблиці, графіки для відображення результатів у зрозумілому вигляді [20].

Thread Group (група потоків) визначає параметри для навантаження, крім цього він є контейнером і містить багато інших елементів: контролери, запити, налаштування.

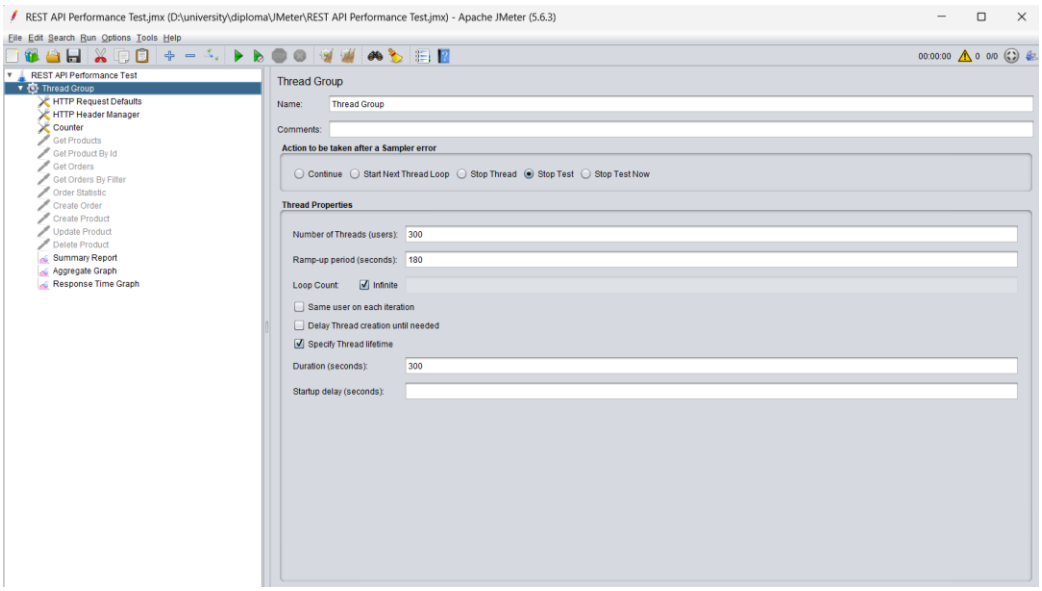

Рисунок 3.6 – Налаштування групи потоків

Thread Group робить симуляцію користувачів, які запускають запити. Потоки (користувачі) потрібно вказати у полі Number Of Threads. Ramp-Up Period (період нарощування) показує, як швидко користувачі будуть збільшуватись, щоб не було різкого навантаження, а Loop Count (кількість циклів) – скільки разів кожен користувач виконає запит. Останнім є Duration, який визначає час роботи тесту.

HTTP Request Defaults – це загальні параметри HTTP-запитів, тобто домен і порт можна ввести одразу для всіх запитів.

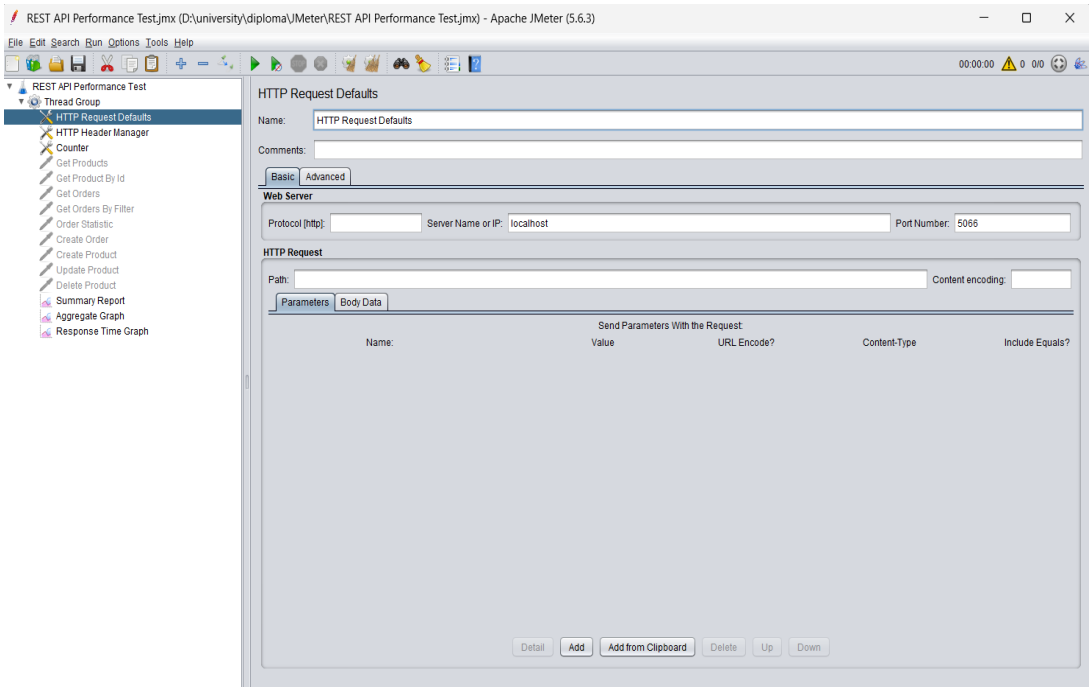

Рисунок 3.7 – Налаштування параметрів HTTP-запитів

Також є HTTP Header Manager, в якому задаються заголовки до запитів, наприклад тип вмісту (Content-Type), який показує формат надсилання даних (JSON в нашому випадку).

HTTP Request (запит) – це основний елемент, який надсилає запит. Тут також можна задати загальні налаштування запитів, але також вказати тип операції і шлях, для оперування даними. Внизу можна визначати параметри запиту і дані до нього (наприклад для створення або оновлення інформації).

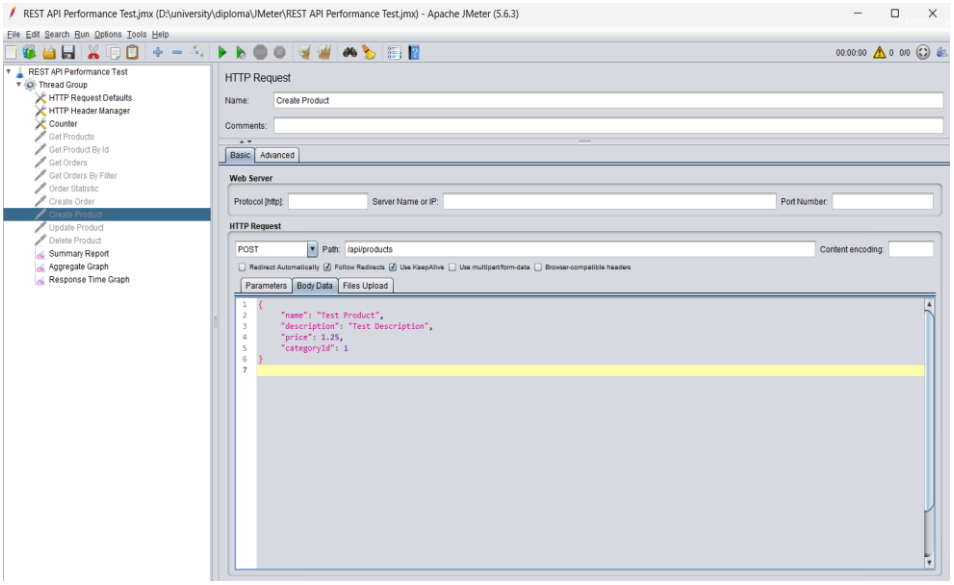

Рисунок 3.8 – Налаштування запиту створення продукту

Щоб налаштувати JMeter для використання з gRPC, потрібно встановити додатковий плагін JMeter gRPC Request. Після цього необхідно створити елемент gRPC Request, в якому обирається файл .proto, з нього підтягуються запити і після налаштувань розміру даних, часу очікування можна тестувати.

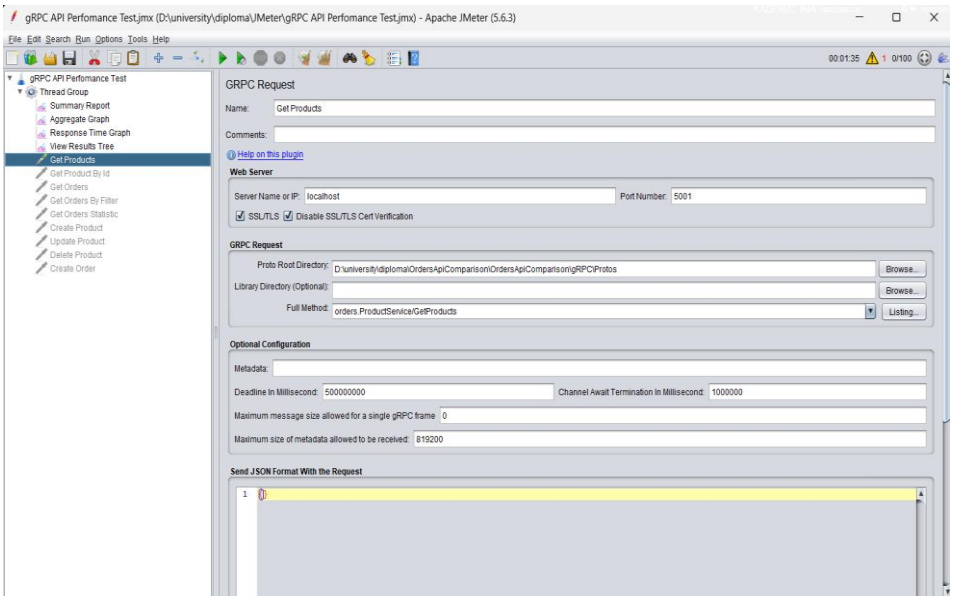

Рисунок 3.9 – Налаштування запиту gRPC

Крім цього інструменту, буде використовуватись Postman, в якому зручно створювати і тестувати запити. В ньому можна отримати розмір кожного запиту і визначити зручність користування.

РОЗДІЛ 4
РЕЗУЛЬТАТИ ТЕСТУВАННЯ РЕАЛІЗОВАНИХ ТЕХНОЛОГІЙ

4.1. Результати тестування за допомогою JMeter

Для початку було проведено навантажувальне тестування. Щоб охопити різні типи запитів, було обрано такі:

– Get All Products – отримання всіх продуктів;

– Get Product By Id – отримання одного продукта за ідентифікатором;

– Get All Orders – отримання всіх замовлень із вкладеними структурами (продукти, користувачі, адреси);

– Get Orders By Filter – отримання фільтрованих замовлень (пагінація, сортування, фільтри по даті чи статусах);

– Get Orders Statistic – обчислення статистики замовлень;

– Create Product – створення продукта;

– Update Product – оновлення частини продукта;

– Delete Product – видалення продукта;

– Create Order – створення замовлення з вкладеними об’єктами.

Тестування навантаження проводилось з налаштуваннями 300 користувачів, 180 секунд наростання і тривалістю 300 секунд. Ці параметри застосовувались для всіх запитів.

Для початку розглянемо значення кожного показника. Усі метрики використовуються для аналізу продуктивності програми. Результати кожного запиту в цьому підрозділі розділені по 2 таблиці: Час роботи, Масштабованість і стабільність.

Часові метрики показують час для виконання запитів. Average (sec) – це середнє значення виконання всіх запитів, який може показати відносний час відгуку. Median (sec) – це час виконання запиту, який перевищив або досягнув половини (50%) усіх запитів, тобто половина запитів виконалась швидше за цей час, а інша половина повільніше. Наступні показники 90% Line, 95% Line, 99%

Line схожі на Median, але замість половини підставляються їхні відсотки. Наприклад, 90% Line показує час виконання, який перевищили або досягнули 90% запитів і тільки 10% виконувались повільніше. Ці показники важливі для рівня обслуговування і визначення продуктивності для більшості користувачів. Min (sec) і Max (sec) – це відповідно найменший і найбільший час виконання запиту.

Наступні метрики з таблиці Масштабованість і Стабільність. Throughput (sec) – це пропускна здатність, яка показує кількість успішних запитів за 1 секунду. Error % показує запити, які закінчились помилкою. Він допомагає визначити надійність системи. Останнім є Standard Deviation (sec), який показує стандартне відхилення відносно середнього часу. Стабільним вважається низьке стандартне відхилення.

Для кожного запиту показано таблиці і графіки. У кожній таблиці зеленим маркером позначено найкращий результат і червоним – найгірший.

Першим запитом для порівняння є отримання всіх продуктів. Для коректних результатів будемо отримувати однакові поля в усіх технологіях.

У таблиці 4.1 наведено результати першого тесту щодо часу відповіді. За середнім часом (Average) gRPC є найшвидшим, що підтверджує переваги HTTP/2 протоколу і Protobuf. При цьому GraphQL виконав 50% запитів швидше (Median). На це може впливати час з’єднання, його встановлення в gRPC. Інші показники (90% Line, 95% Line, 99% Line і Max) є кращими у gRPC.

Щодо масштабованості і стабільності (таблиця 4.2) кращим також виявився gRPC. Він може обслуговувати найбільшу кількість користувачів за секунду і має найменше відхилення. Під цим рівнем навантаження технології не мають помилок.

Результати з таблиць можна побачити у вигляді графіків (рис. 4.1 – 4.3). На кожному графіку така відповідність кольорів і значень: червоний – Average, синій – Median, зелений – 90% Line, жовтий – 95% Line, фіолетовий – 99% Line, сірий – Min, чорний – Max.

Результати показують, що gRPC є найшвидшим і найбільш стабільним, що добре для високонавантажених систем. Наступним є GraphQL за результатами. Найповільнішим виявився REST через JSON і протокол.

Таблиця 4.1 – Час відповіді запиту Get All Products

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	13.03	10.75	26.15	27.87	29.68	0.33	35.7
GraphQL	7.21	5.23	14.16	15.32	18.43	0.08	21.4
gRPC	5.74	7.14	8.18	8.32	8.69	0.11	8.9

Таблиця 4.2 – Масштабованість і стабільність запиту Get All Products

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	15.9	0	8.54
GraphQL	28.9	0	4.55
gRPC	36.4	0	2.62

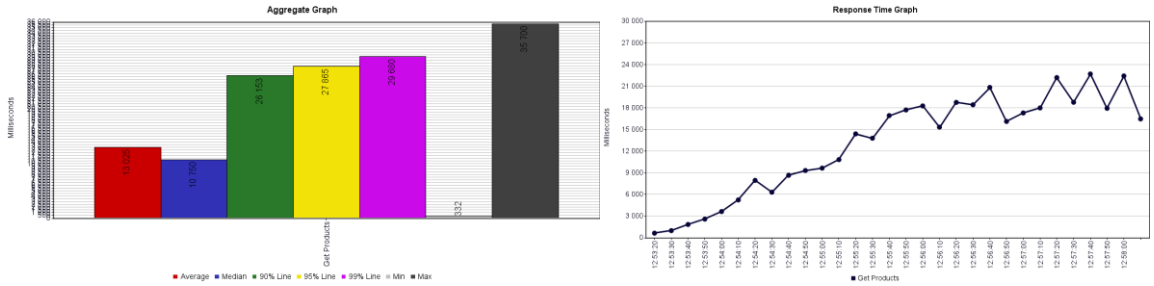

Рисунок 4.1 – Графіки роботи запиту Get All Products в REST API

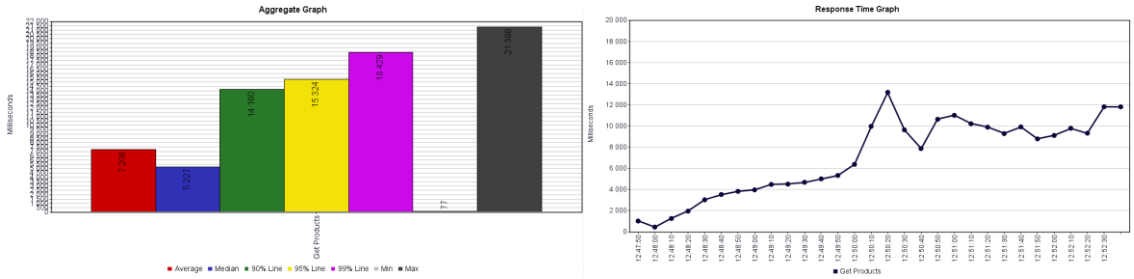

Рисунок 4.2 – Графіки роботи запиту Get All Products в GraphQL API

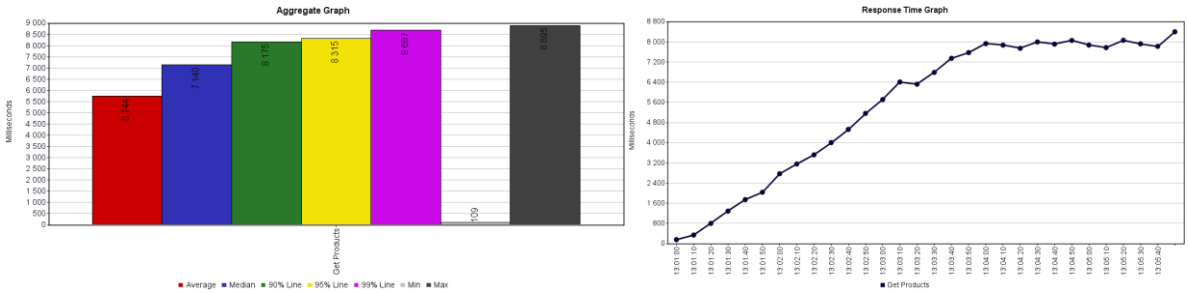

Рисунок 4.3 – Графіки роботи запиту Get All Products в gRPC API

Наступним запитом є отримання одного продукта за його ідентифікатором. У таблицях 4.3 і 4.4 можна побачити, що неочікувано технологія GraphQL виявилась найгіршою за всіма показниками. Ймовірно це пов'язано з складнішою обробкою запитів, хоча очікуваний результат мав бути іншим. Метрики представляють, що gRPC є найстабільнішою і найшвидшою.

Важливо, що час відповіді є досить довгим для такого запиту, що вказує на неоптимізованість системи. Варто використовувати сторонні засоби і бібліотеки для оптимізації.

Кількість помилок (відсутність) показують, що можна збільшити навантаження, але це також призведе до погіршення часових метрик.

Таблиця 4.3 – Час відповіді запиту Get Product By Id

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	4.42	3.62	9.26	9.98	11.38	0.07	13.13
GraphQL	5.19	4.66	10.86	11.84	13.16	0.05	14.38
gRPC	3.4	3.9	6.22	6.65	7.76	0.01	10.9

Таблиця 4.4 – Масштабованість і стабільність запиту Get Product By Id

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	47.2	0	3.15
GraphQL	40.2	0	3.7
gRPC	61.6	0	2.27

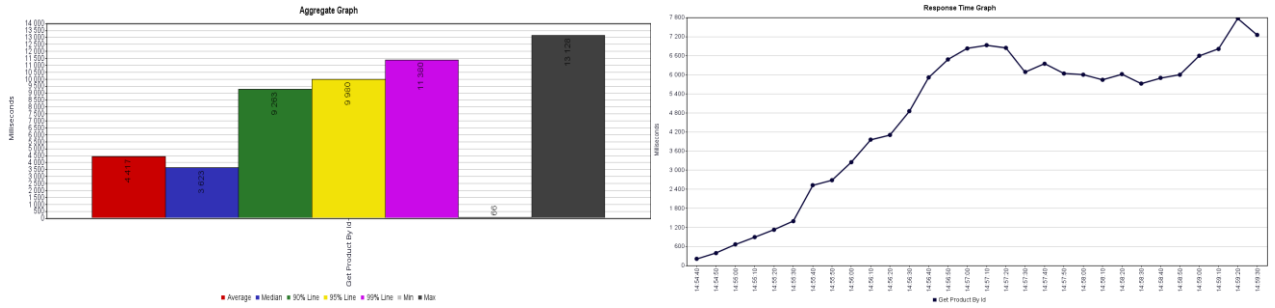

Рисунок 4.4 – Графіки роботи запиту Get Product By Id в REST API

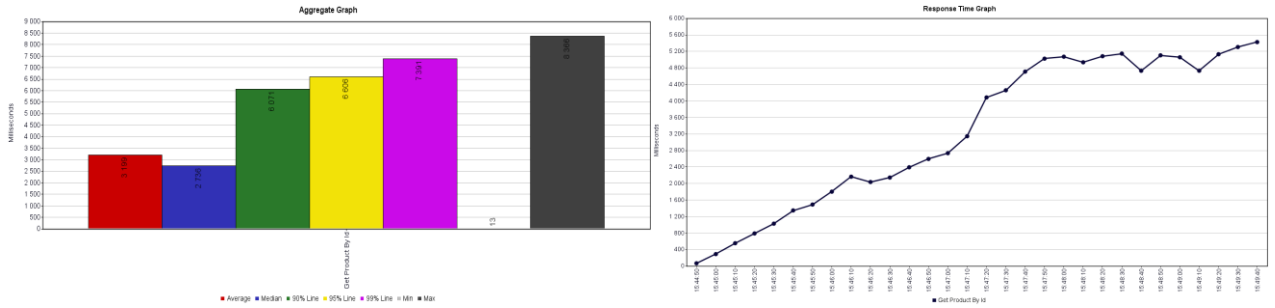

Рисунок 4.5 – Графіки роботи запиту Get Product By Id в GraphQL API

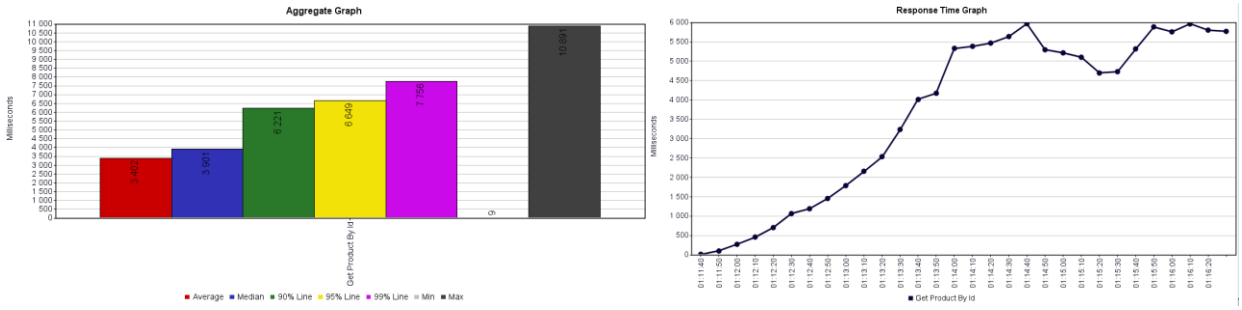

Рисунок 4.6 – Графіки роботи запиту Get Product By Id в gRPC API

Запит Get Orders представляє собою отримання 1000 замовлень з усіма вкладеними структурами (продукти, категорії, користувачі, адреси), тобто він є дуже складним і громіздким. Такі запити зазвичай не виконують, а використовують фільтрування і пагінацію для коректної роботи.

REST API проявив себе найгірше: довгий час роботи, погана продуктивність і наявність великого відсотку помилок. Він не призначений для обробки такої кількості складних запитів, адже тягне багато зайвої інформації.

З таблиць 4.5 і 4.6 робимо висновок, що в цьому запиті GraphQL є найпродуктивнішою технологією для такої роботи. Вона має найгнучкішу обробку частково через бібліотеку HotChocolate, саме тому для коректного

порівняння реалізацію цього запиту було погіршено.

Щодо gRPC бачимо, що результати тримаються на середині і теж є досить поганими. Оптимізувати цей запит досить складно і велику роль відіграє процес ручного мапінгу даних. REST і gRPC повністю провалили цей тест ймовірно через складність структури і неоптимізованість.

Таблиця 4.5 – Час відповіді запиту Get Orders

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	71.8	44.2	160.17	197.3	228.8	5.34	245.8
GraphQL	6.73	4.5	16.53	19.66	26.79	0.12	31.41
gRPC	42.61	47.76	67.96	72.09	80.18	2.35	89.09

Таблиця 4.6 – Масштабованість і стабільність запиту Get Orders

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	2.8	37.64	62.4
GraphQL	30.9	0	6.31
gRPC	4.2	0	22.05

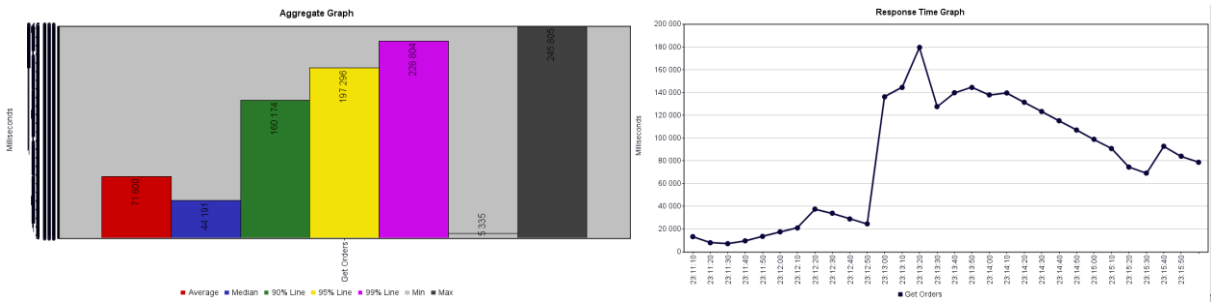

Рисунок 4.7 – Графіки роботи запиту Get All Orders в REST API

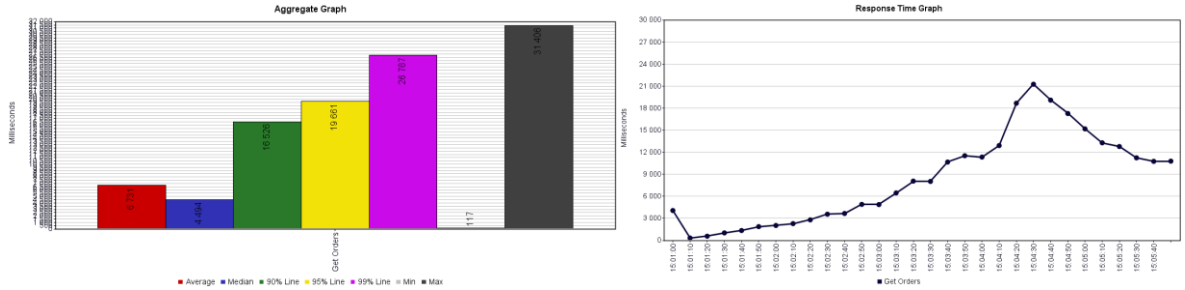

Рисунок 4.8 – Графіки роботи запиту Get All Orders в GraphQL API

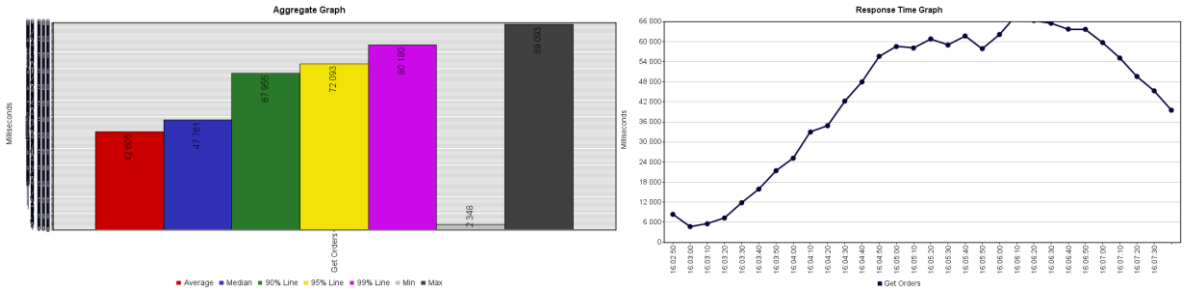

Рисунок 4.9 – Графіки роботи запиту Get All Orders в gRPC API

Запит Get Orders By Filter є досить поширеним в системах. Він представляє собою отримання оброблених і відфільтрованих даних. В цьому тестуванні використано сортування (за ціною), пагінацію (25 замовлень на 1 сторінці) і фільтрування (замовлення створенні від певної дати). Результати представлені у таблицях 4.7 і 4.8.

Технологія gRPC проявила найкращі і найстабільніші результати, але досить довгі. В неї досить мале відхилення і висока пропускна здатність. GraphQL теж має досить непогані результати, а якщо вибирати лише потрібні поля, то ймовірно перевершить показники gRPC. При цьому технологія REST досі залишається найгіршою.

Таблиця 4.7 – Час відповіді запиту Get Orders By Filter

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	5.48	3.66	11.3	15.6	31.87	0.035	64.03
GraphQL	4.78	3.96	10.22	10.88	12	0.058	13.82
gRPC	3.85	3.75	6.97	7.53	9.13	0.012	37.27

Таблиця 4.8 – Масштабованість і стабільність запиту Get Orders By Filter

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	37.9	0	5.96
GraphQL	43.7	0	3.25
gRPC	54.5	0	2.7

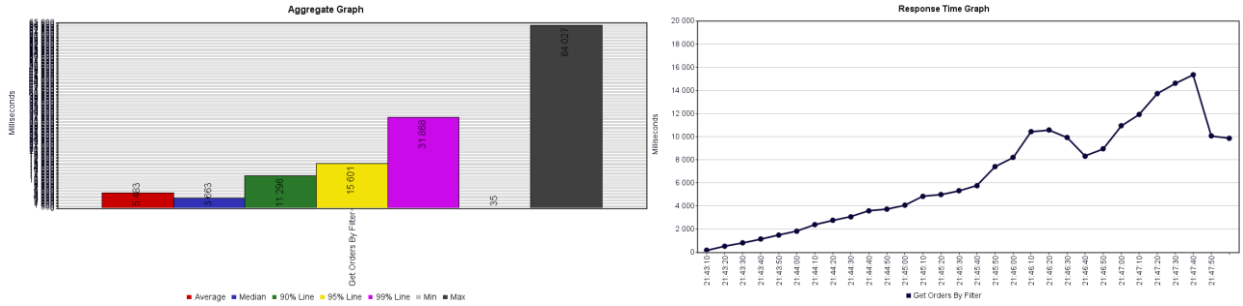

Рисунок 4.10 – Графіки роботи запиту Get Orders By Filter в REST API

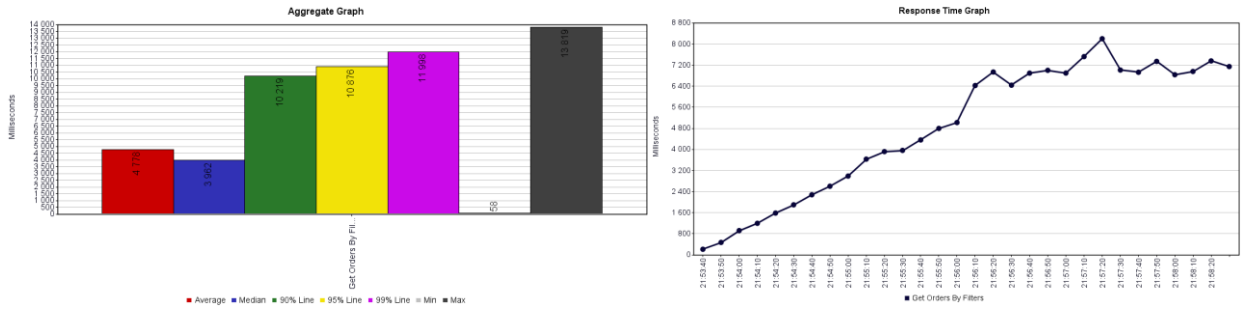

Рисунок 4.11 – Графіки роботи запиту Get Orders By Filter в GraphQL API

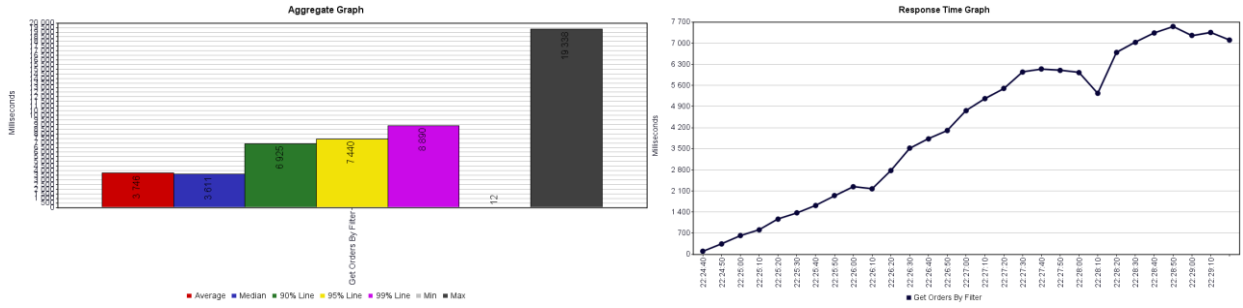

Рисунок 4.12 – Графіки роботи запиту Get Orders By Filter в gRPC API

Запит Get Orders Statistic представляє собою обчислення і аналіз. В ньому вираховується кількість замовлень, прибуток тощо. На відміну від інших запитів, які залежать від кількості даних, отримання статистики показує обчислювальні операції.

Таблиці 4.9 і 4.10 показують, що gRPC є лідером за часом відгуку, хоча деякі метрики є кращими в REST. Технологія gRPC знову проявляє стабільність і передбачуваність у результатах. В цих результатах GraphQL проявив себе найгірше. Можливо в ньому відбуваються додаткові витрати на обробку запиту на стороні сервера. В цьому запиті основна перевага GraphQL (гнучкість) взагалі ніяк не проявляється.

Таблиця 4.9 – Час відповіді запиту Get Orders Statistic

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	6.52	4.95	10.91	12.44	39.04	0.02	116.12
GraphQL	8.3	7.5	12.51	13.49	72.51	0.07	165.07
gRPC	6.38	6.21	9.97	10.75	17.37	0.09	117.59

Таблиця 4.10 – Масштабованість і стабільність запиту Get Orders Statistic

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	32.1	0	7.86
GraphQL	25.1	0	11.79
gRPC	35.6	0	6.96

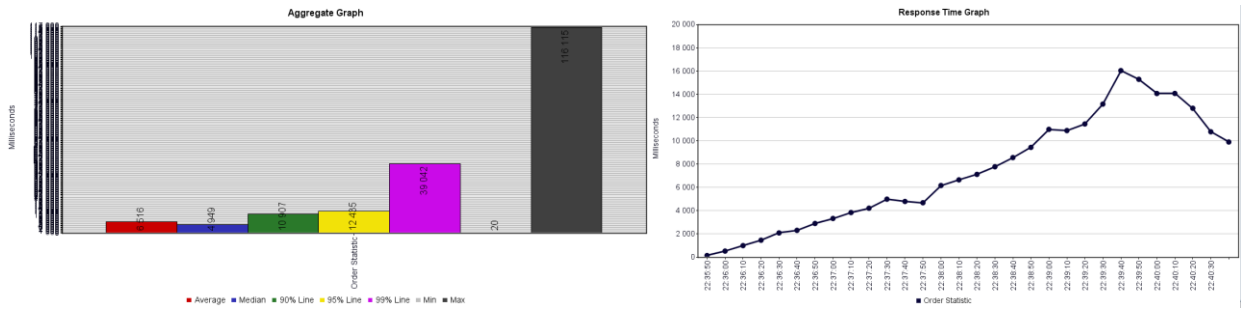

Рисунок 4.13 – Графіки роботи запиту Get Orders Statistic в REST API

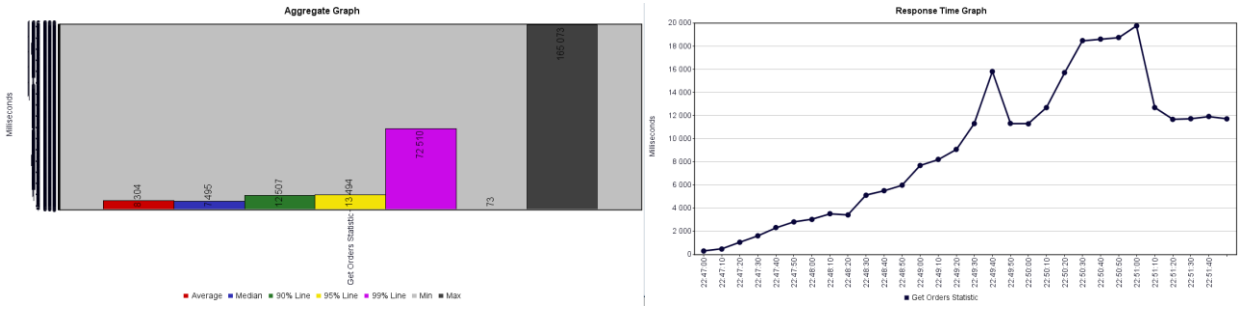

Рисунок 4.14 – Графіки роботи запиту Get Orders Statistic в GraphQL API

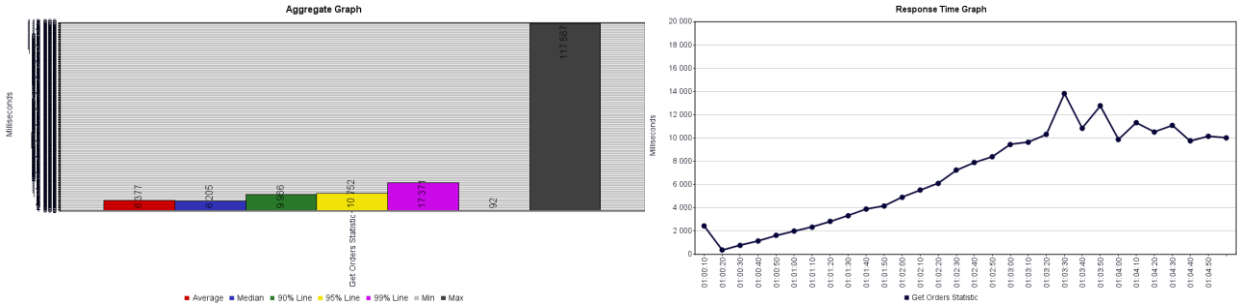

Рисунок 4.15 – Графіки роботи запиту Get Orders Statistic в gRPC API

Запит створення продукту є досить важливою операцією і може бути проблемним місцем у системі через зміну даних, тому що ці операції складніші під час виконання на стороні сервера.

В таблицях 4.11 і 4.12 видно, що gRPC і REST мають відносно схожі результати. Ефективність gRPC досі зберігається лише трохи випередивши REST в цьому тестування. Обидві технології мають дуже хорошу пропускну здатність і мале відхилення. Відсутність помилок також є дуже хорошим результатом в цьому тестуванні.

Знову технологія GraphQL проявляє себе найгірше. В ній операція створення реалізована за допомогою мутації, що створює додатковий шар обробки даних. Ймовірно це ускладнює і збільшує витрати в порівнянні з простими запитами REST і gRPC. В запитах мутацій GraphQL не має суттєвих переваг над іншими технологіями.

Таблиця 4.11 – Час відповіді запиту Create Product

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	2.93	2.41	6.43	7.05	7.75	0.03	8.68
GraphQL	4.74	3.98	11.1	11.92	12.94	0.02	13.93
gRPC	2.75	2.59	4.86	5.34	6.72	0.02	22.15

Таблиця 4.12 – Масштабованість і стабільність запиту Create Product

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	71.3	0	2.04
GraphQL	44.2	0	3.53
gRPC	76.2	0	1.76

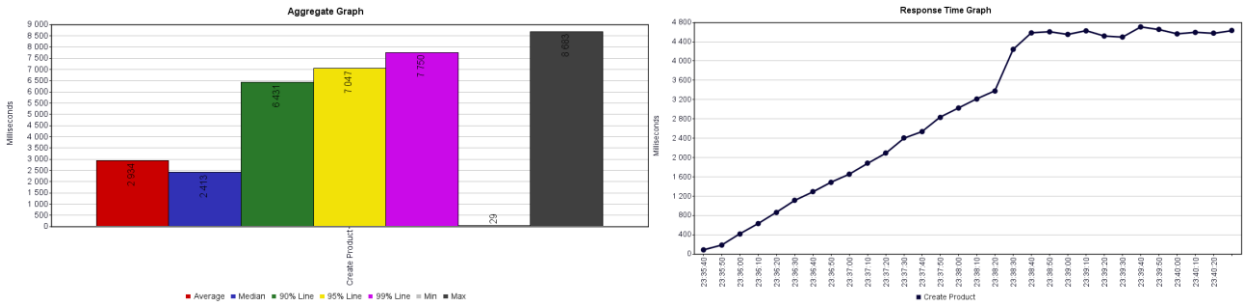

Рисунок 4.16 – Графіки роботи запиту Create Product в REST API

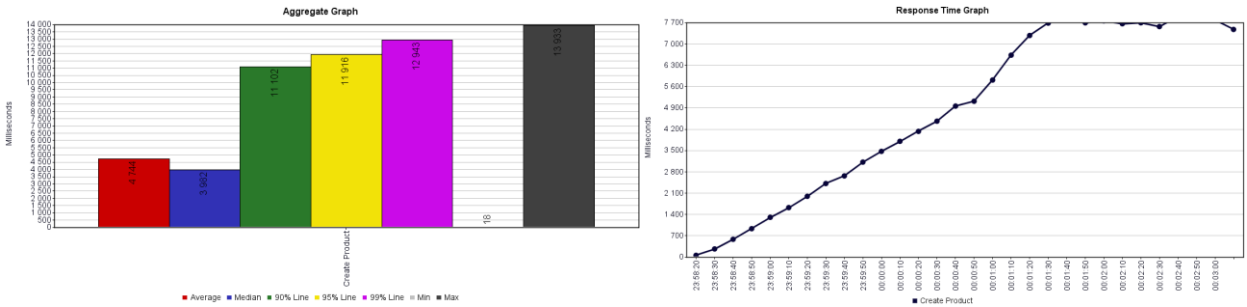

Рисунок 4.17 – Графіки роботи запиту Create Product в GraphQL API

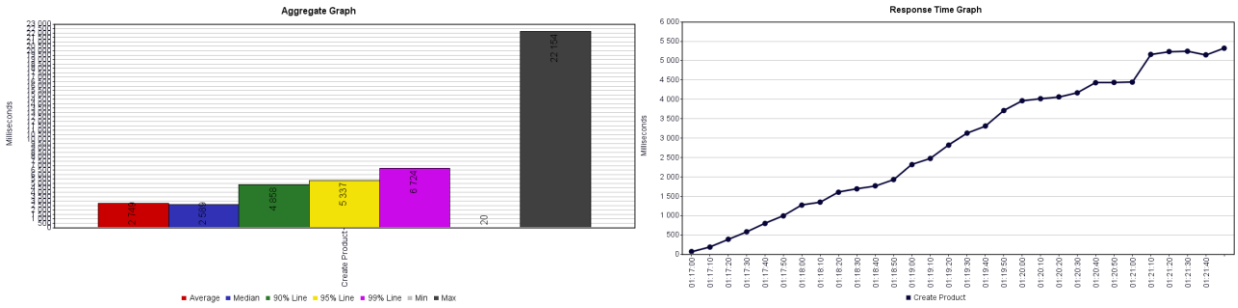

Рисунок 4.18 – Графіки роботи запиту Create Product в gRPC API

Запит оновлення продукту показує дуже схожі результати до попереднього (таблиці 4.13 і 4.14). gRPC є найкращою, але REST теж має досить хороші результати. Відносно масштабованості і стабільності технологія gRPC проявила найгірші результати, але можемо побачити, що вся таблиця має досить схожі значення.

Таблиця 4.13 – Час відповіді запиту Update Product

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	4.62	3.55	9.1	9.95	14.33	0.02	60.12
GraphQL	4.97	3.89	11.31	12.14	13.24	0.05	15.41
gRPC	4.05	3.21	7.27	7.97	9.88	0.05	53.87

Таблиця 4.14 – Масштабованість і стабільність запиту Update Product

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	45	0	3.8
GraphQL	42	0	3.74
gRPC	41.6	0	4.02

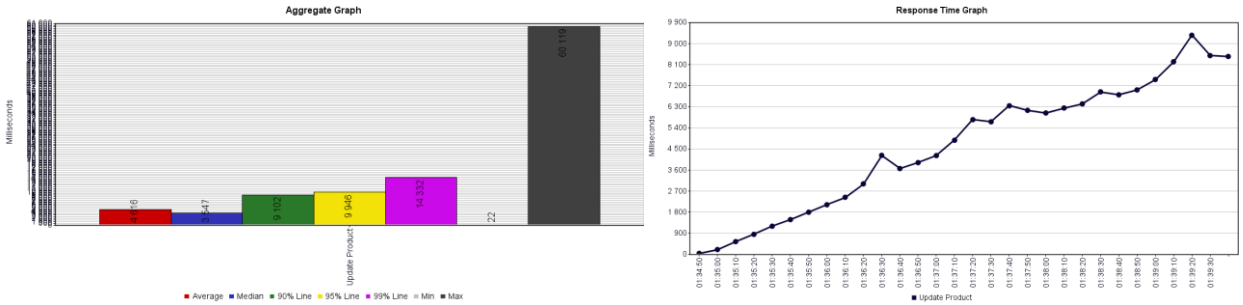

Рисунок 4.19 – Графіки роботи запиту Update Product в REST API

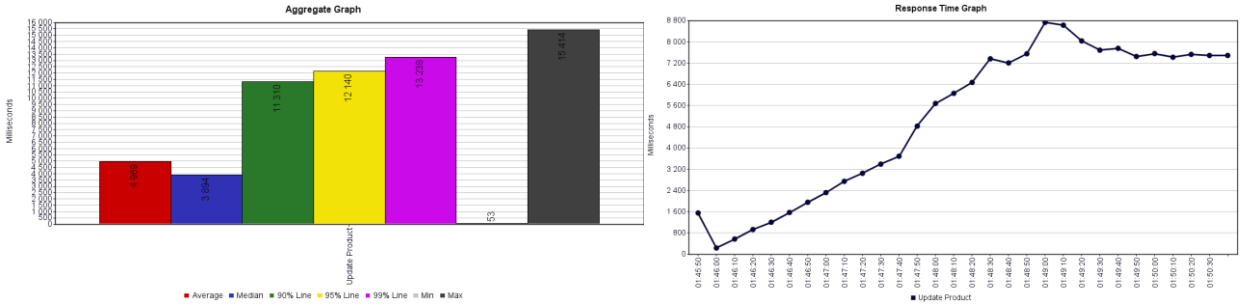

Рисунок 4.20 – Графіки роботи запиту Update Product в GraphQL API

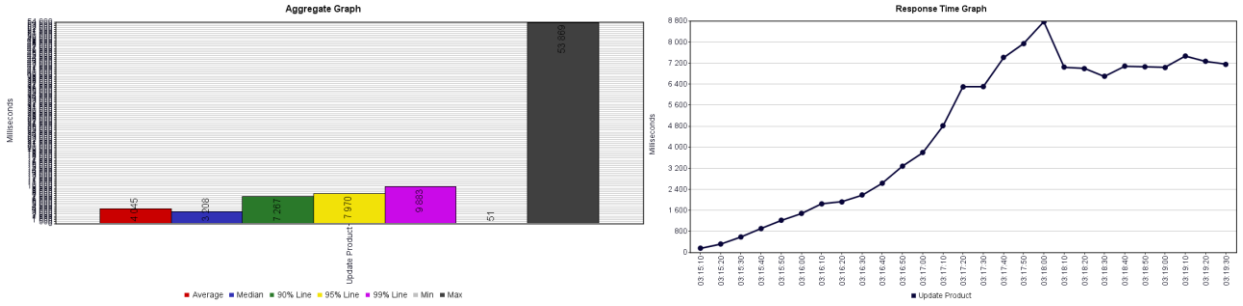

Рисунок 4.21 – Графіки роботи запиту Update Product в gRPC API

Далі розглянемо запит видалення продукта (таблиці 4.15 і 4.16). Як і в інших операціях модифікацій технологія GraphQL є найгіршою. Вона дуже нестабільна і деякі метрики сильно відрізняються від gRPC. Якщо видалення буде дуже інтенсивним, то GraphQL і REST проявлять себе з гіршої сторони. За результатами вони знаходяться близько одна від одної.

Технологія gRPC надалі залишається стабільною завдяки протоколу. Вона найкраще витримала навантаження, має найменше відхилення і є найбільш передбачуваною.

Таблиця 4.15 – Час відповіді запиту Delete Product

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	7.19	5.78	11.3	12.55	45.88	0.02	148.2
GraphQL	7.69	5.78	13.18	14.05	52.7	0.03	159
gRPC	4.7	4.51	7.76	8.6	11.98	0.01	118.1

Таблиця 4.16 – Масштабованість і стабільність запиту Delete Product

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	29	0	8.86
GraphQL	27.2	0	10.17
gRPC	44.5	0	5.59

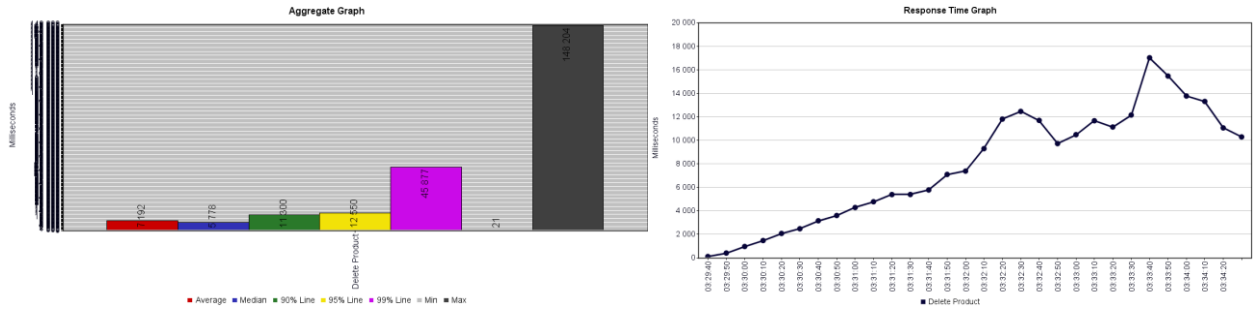

Рисунок 4.22 – Графіки роботи запиту Delete Product в REST API

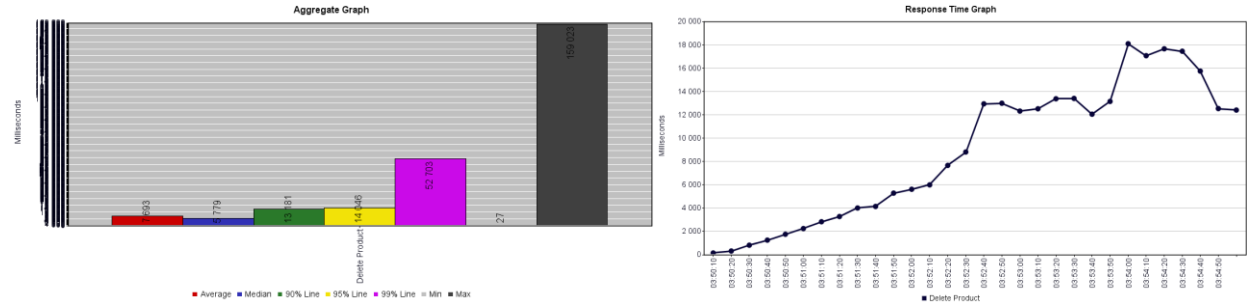

Рисунок 4.23 – Графіки роботи запиту Delete Product в GraphQL API

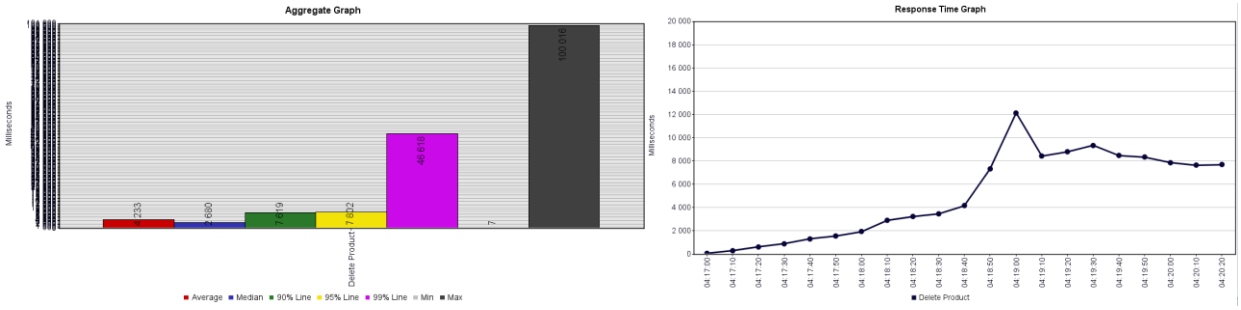

Рисунок 4.24 – Графіки роботи запиту Delete Product в gRPC API

Останнім запитом є створення замовлення. Воно відрізняється від запиту створення продукта, тому що тут є вкладені дані (рядки замовлення). Ці результати (таблиці 4.17 і 4.18) є нетиповими для технології gRPC. Ймовірно причина у складності серіалізації/десеріалізації великих структур даних і необхідно додатково оптимізувати запит.

В цьому запиті неочікувано REST проявила найкращі результати щодо часу, масштабованості, продуктивності. Запит створення виконувався відносно швидко в REST і GraphQL. Варто зазначити, що в технології REST з'явився незначний відсоток помилок.

JMeter показав логічно коректні, але нетипові результати для цього запиту.

Таблиця 4.17 – Час відповіді запиту Create Order

Технологія	Average (sec)	Median (sec)	90% Line (sec)	95% Line (sec)	99% Line (sec)	Min (sec)	Max (sec)
REST	5.42	3.62	11.2	12.98	21.21	0.02	86.69
GraphQL	6.53	4.49	12.95	14.81	30.29	0.095	101.6
gRPC	11.08	10.67	19.98	21.4	25.63	0.2	130.7

Таблиця 4.18 – Масштабованість і стабільність запиту Create Order

Технологія	Throughput (sec)	Error %	Standard Deviation (sec)
REST	38.3	0.01	6.31
GraphQL	32	0	7.44
gRPC	16.3	0	9.12

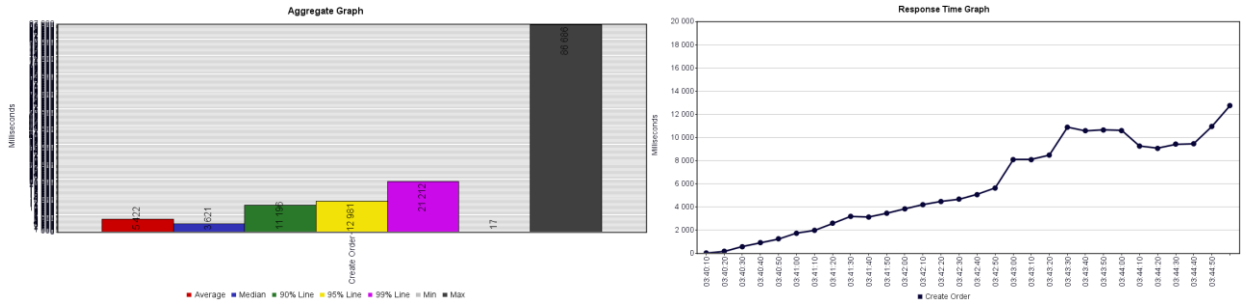

Рисунок 4.25 – Графіки роботи запиту Create Order в REST API

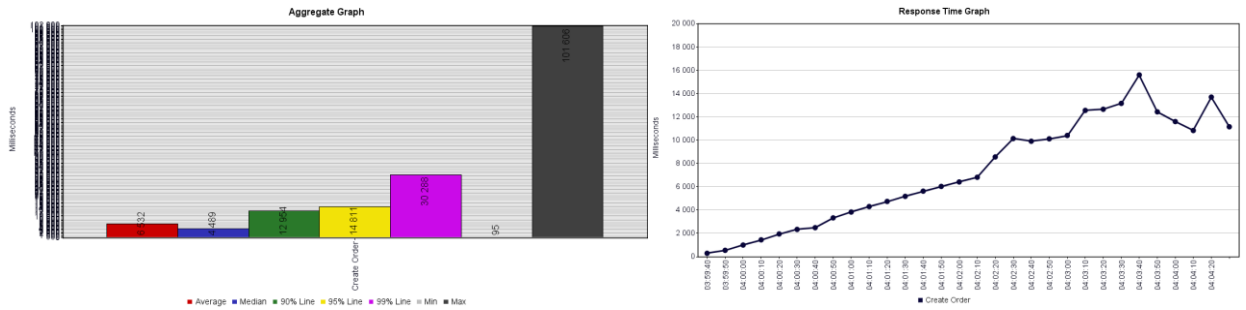

Рисунок 4.26 – Графіки роботи запиту Create Order в GraphQL API

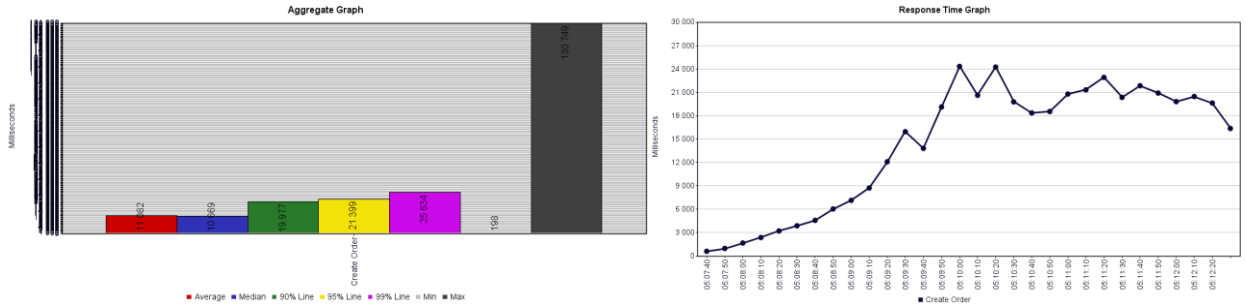

Рисунок 4.27 – Графіки роботи запиту Create Order в gRPC API

Отже, аналіз різних сценаріїв показав, що в більшості ситуацій технологія gRPC є найбільш вдалим рішенням. Оцінювання масштабованості, стабільності і часу відгуку за допомогою навантажувального тестування відображає, що вибір технології залежить від умов. gRPC має найкращу загальну продуктивність. GraphQL буде вдалим, коли обсяг даних дуже великий і потрібно діставати окремі поля. При цьому REST, незважаючи на гірші результати, є найуніверсальнішим і найпростішим рішенням.

4.2. Результати тестування за допомогою Postman та інших засобів

Наступним засобом для тестування є Postman. Для зручності було створено запити в різних папках. Postman дає можливість легко визначити розмір одного запиту для REST і GraphQL. На жаль, він не підтримує gRPC в повному обсязі, тому розмір запитів цієї технології можна отримати за допомогою попереднього засобу JMeter.

У таблиці 4.19 можна побачити розмір кожного запиту, згаданого в попередньому підрозділі. На основі результатів можна сказати, що gRPC найменш ефективний для операцій з отримання великої кількості даних (Get All Products, Get All Orders). Але саме ця технологія проявляє прекрасні результати на всіх запитах, пов'язаних з модифікаціями даних (Create Product, Update Product, Delete Product, Create Order).

REST і GraphQL мають схожі результати через однаковий формат даних (JSON), на відміну від gRPC (формат Protobuf). В операціях з великими структурами GraphQL показує хороший розмір даних завдяки своїй гнучкості. Він дає змогу отримувати лише потрібні поля, чим дуже відрізняється від інших технологій і може бути ще більш ефективним.

Останнім важливим кроком є визначення зручності розробки кожною технологією. В Інтернеті існує багато статей і відео про кожну технологію, але REST поки переважає за кількість інформації. Під час розробки було витрачено більше часу на ознайомлення з GraphQL і gRPC, але він не врахований у таблиці 4.20.

Таблиця 4.19 – Розмір усіх запитів залежно від технології

Запит	**Технологія**	**Response Size (KB)**
Get All Products	REST	312.33
	GraphQL	3.36
	gRPC	405.359
Get Product By Id	REST	0.46
	GraphQL	0.412

	gRPC	0.376
Get All Orders	REST	1790
	GraphQL	18.39
	gRPC	2550.857
Get Orders By Filter	REST	2.11
	GraphQL	61.08
	gRPC	4.865
Get Orders Statistic	REST	0.229
	GraphQL	0.3
	gRPC	0.112
Create Product	REST	0.398
	GraphQL	0.366
	gRPC	0.191
Update Product	REST	0.448
	GraphQL	0.38
	gRPC	0.204
Delete Product	REST	0.081
	GraphQL	0.194
	gRPC	0.021
Create Order	REST	0.885
	GraphQL	0.783
	gRPC	0.365

GraphQL за часом розробки є найшвидшим, в нього зручна структура і багато бібліотек, які економлять час. REST не сильно далеко відійшов, але зайняло більше часу на опис винятків, правил тощо. Також відіграє роль розбиття на сервіси і контролери.

Найважчою і найбільш громіздкою для реалізації є технологія gRPC. Вона займає більше часу щодо підключення і ознайомлення. Крім цього, опис файлу .proto є досить незручним, але генерація коду спрощує процес.

Таблиця 4.20 – Зручність розробки залежно від технологій

Технологія	Час розробки (год)	Кількість рядків коду
REST	8	307
GraphQL	6	150
gRPC	14	484

Отже, реалізація за допомогою gRPC є найбільш складною і довгою, але цей процес вартий того, щоб пришвидшити і оптимізувати систему.

Варто згадати, що кожна технологія має свої переваги, які неможливо або складно представити за допомогою інших технологій. GraphQL дозволяє легко вибирати поля для отримання, що неможливо зробити в REST і gRPC. Також він має режим підписок, щоб отримувати дані в реальному часі. У REST API це можна реалізувати за допомогою сокетів.

Технологія gRPC має потокову передачу, що є дуже зручною функцією для асинхронної обробки великих даних. Крім цього, вона містить двонаправлений зв’язок, щоб встановлювати зв’язок між сервісами.

REST є найпростішою і найбільш підтримуваною технологією. Її перевага в універсальності і зручності використання. REST також має потужне кешування на будь-якому рівні програми.

Залежно від пріоритетів програми, кожна технологія проявляє свої кращі сторони.

РОЗДІЛ 5
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

5.1. Розробка логіко-імітаційної моделі виникнення травм і аварій

Методикою оцінки рівня небезпеки робочих місць, машин, виробничих процесів та окремих виробництв передбачено пошук об'єктивного критерію рівня небезпеки для конкретного об'єкта. Таким показником вибрана ймовірність виникнення аварії, травми залежно від явища, що досліджується.

Для побудови логіко-імітаційної моделі процесу, формування і виникнення аварії та травми в процесі створення мікрокліматичних умов у приміщенні оцінюють відповідні небезпечні події. Кожній із них присвоїмо ймовірність виникнення:

Шифр	Назва події	Ймовірність
P_1	Відсутність захисного заземлення	0,02
P_2	Пошкодження захисного заземлення	0,04
P_3	Спрацювання складових захисту	0,1
P_4	Неправильна експлуатація захисту	0,02
P_5	Відсутність профілактичних заходів	0,2
P_6	Відсутність захисного щита	0,12
P_7	Недотримання правил вибору взуття	0,15
P_8	Незнання правил техніки безпеки	0,1
P_9	Відсутність засобів індивідуального захисту	0,2
P_{10}	Легковажність	0,08

На основі наведених подій будуємо матрицю логічних взаємозв'язків між окремими пунктами, графічна інтерпретація якої зображено на рис. 5.1.

Розрахуємо ймовірності виникнення подій, що формують логіко-імітаційну модель процесів створення мікрокліматичних умов. Розглянемо травмонебезпечну ситуацію, що виникає за умови роботи працівників із електронебезпекою.

Підставивши дані ймовірностей базових подій у формулу, отримаємо ймовірність події 13: $P_{13} = 0{,}2 + 0{,}4 - 0{,}2 \cdot 0{,}4 = 0{,}0592$.

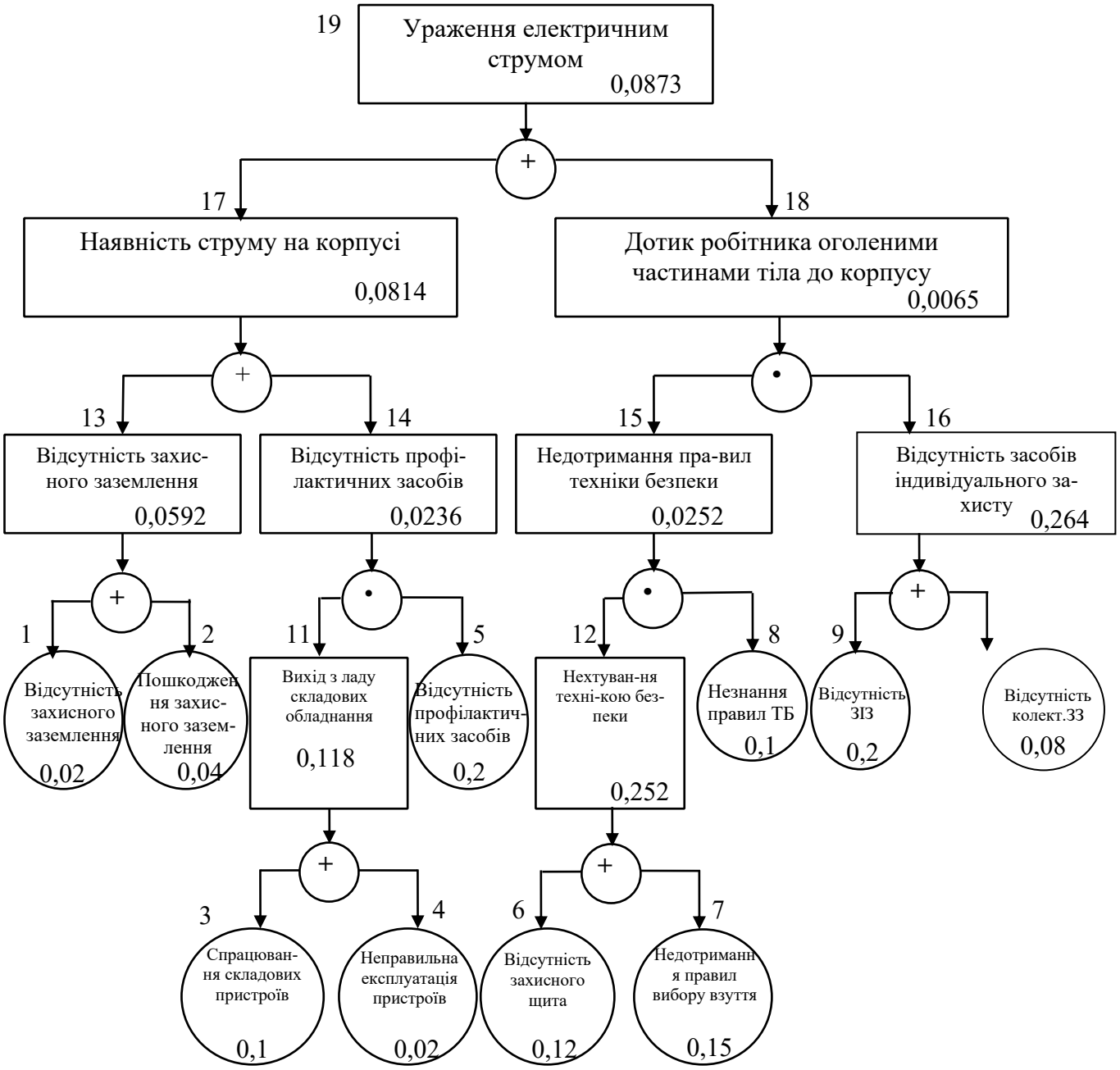

Рисунок 5.1. Матриця логічних взаємозв’язків між окремими подіями травмонебезпечної ситуації

Аналогічно визначаємо ймовірність інших подій:

$$P_{11} = P_4 + P_5 - P_4P_5 = 0,3 + 0,4 - 0,3 \cdot 0,4 = 0,118.$$

$$P_{12} = P_6 + P_7 - P_6P_7 = 0,3 + 0,5 - 0,3 \cdot 0,5 = 0,252.$$

$$P_{16} = P_9 + P_{10} - P_9P_{10} = 0,2 + 0,15 - 0,2 \cdot 0,15 = 0,264.$$

$$P_{14} = P_{11} \cdot P_5 = 0,118 \cdot 0,2 = 0,0236.$$

$$P_{15} = P_{12} \cdot P_8 = 0,252 \cdot 0,1 = 0,0252.$$

$$P_{17} = P_{13}+P_{14} - P_{13}\cdot P_{14} = 0,592+0,0236-0,0592\cdot 0,0236 = 0,0814.$$

$$P_{18} = P_{15}\cdot P_{16} = 0,264\cdot 0,0252 = 0,0065.$$

$$P_{19} = P_{17}+P_{18} - P_{17}\cdot P_{18} = 0,0065+0,0814 - 0,0065\cdot 0,0814 = 0,0873.$$

Таким чином, ймовірність перекидання машини та наслідкового виникнення травми працівника є досить мала і становить – $P_{19} = 0,0873$.

5.2. Планування заходів із покращення умов праці

До заходів щодо покращення умов праці належать всі види діяльності, спрямовані на попередження, нейтралізацію або зменшення негативної дії шкідливих і небезпечних виробничих факторів на працівників.

Рівень умов праці оцінюють порівнянням за фактичними і нормативними значеннями узагальнених (групових) показників.

Заходи щодо поліпшення умов праці здійснюють з метою створення безпечних умов праці шляхом:

- доведення до нормативного рівня показників виробничого середовища за елементами умов праці;
- захисту працівників від дії небезпечних і шкідливих виробничих факторів.

До показників ефективності заходів щодо поліпшення умов праці належать:

а) зміни стану умов праці:

- зміна кількості засобів виробництва, приведених у відповідність до вимог стандартів безпеки праці;
- покращення санітарно-гігієнічних показників;
- покращання психофізичних показників, зменшення фізичних і нервово-психічних навантажень, в т.ч. монотонних умов праці;

б) соціальні результати заходів:

- збільшення кількості робочих місць, що відповідають нормативним

вимогам;

- зниження рівня виробничого травматизму;
- престиж та задоволення працею.

Отже, на покращення охорони праці потрібно виділити кошти на відновлення вентиляційних систем у ремонтних майстернях, естетично оформити приміщення офісу, відновити кабінет з охорони праці, поновити протипожежний інвентар.

5.3. Безпека в надзвичайних ситуаціях

Актуальність проблеми природно-техногенної безпеки для населення і території, зумовлена зростанням втрат людей, що спричиняється небезпечними природними явищами, промисловими аваріями та катастрофами. У системі цивільної оборони окремого господарства необхідно забезпечити захист населення таким чином:

Укриття в захисних спорудах, якому підлягає усе населення відповідно до приналежності, досягається створенням фонду захисних споруд.

Евакуаційні заходи, які проводяться в містах та інших населених пунктах, які мають об’єкти підвищеної небезпеки, а також у воєнний час, основним способом захисту населення є евакуація і розміщення його у позаміській зоні.

Медичний захист проводиться для зменшення ступеня ураження людей, своєчасного надання допомоги постраждалим та їх лікування, забезпечення епідеміологічного благополуччя в районах надзвичайних ситуацій.

Радіаційний і хімічний захист включає заходи щодо виявлення і оцінки радіаційної та хімічної обстановки, організацію і здійснення дозиметричного та хімічного контролю, розроблення типових режимів радіаційного захисту, забезпечення засобами індивідуального захисту, організацію і проведення спеціальної обробки.

ВИСНОВКИ

У першому розділі розкрито головні відомості щодо розробки API і його роль в програмуванні. Для дослідження обрано технології REST, GraphQL та gRPC. Огляд документацій цих технологій допоміг зрозуміти їх приклади застосування, переваги, недоліки тощо. Також розглянуто деякі основні вимоги до реалізації API.

На сьогоднішній час REST є найпоширенішою архітектурою через її простоту і зрозумілість. Крім цього, вона є сумісною з багатьма системами, але погано працює з великими і складними даними. GraphQL також набрала великої популярності і є більш гнучкою, адже клієнти самі формують запити. Технологія gRPC є найменш відомою, але найбільш ефективною для передачі даних. Її недолік – це складне налаштування, але воно вартує того, щоб досягти високої продуктивності.

Головними показниками для оцінювання ефективності API є продуктивність, масштабованість, гнучкість, безпека і зручність використання. Для їх вимірювання необхідно скласти план тестування і визначити метрики, які в подальшому зобразити у вигляді діаграм для кращого розуміння.

Сформовано мету і завдання дослідження, які допоможуть практично порівняти REST, GraphQL і gRPC для визначення їх переваг і недоліків в реальних умовах роботи з даними в API.

У другому розділі розглянуто способи і методи дослідження ефективності реалізації API. Було обрано критерії та показники для порівняння продуктивності, гнучкості та масштабованості. Крім цього, було обрано основні технології для написання API і засоби тестування, такі як Apache JMeter та Postman. Важливо створити однакові умови для тестування кожної технології, щоб отримати об’єктивні результати.

Для дослідження необхідно розробити API з однаковою логікою обробки даних за допомогою згаданих технологій. Усі типи API мають виконувати однакові CRUD операції. Кожна технологія буде незалежною частиною, щоб

можна було легко змінювати систему. Для аналізу ефективності необхідно виміряти час відповіді, кількість запитів тощо.

В третьому розділі представлено процес проєктування, розробки та налаштування для здійснення дослідження. На етапі проєктування було створено моделі даних, визначено зв'язки між ними і розроблено загальну архітектуру системи. Це допомогло створити спільну логіку для всіх технологій, щоб їхнє порівняння було коректним.

У підрозділах можна побачити процес створення трьох API: REST, GraphQL, gRPC. Для усіх показано принципи взаємодії, структуру програми, методи тощо. Під час створення можна побачити різницю в реалізації, переваги та недоліки. До REST API і GraphQL API було підключено інтерфейс, щоб зручно використовувати всі запити, проглядати схеми.

Показано налаштування програми JMeter для навантажувального тестування. Підготовлено всі тестові сценарії, конфігурації, параметри, що допомагають вимірювати продуктивність кожного API і можуть формувати метрики для аналізу. Крім цього, для кожного запиту можна побачити додатково графіки і таблиці.

Останній розділ присвячений навантажувальному тестуванню на основі 9 різних сценаріїв: CRUD, фільтрація, обчислення, робота зі складними даними. Для всіх запитів було використано однакові параметри і дані, що дозволяє зробити коректний висновок щодо кожної технології. Також було проведено тестування у Postman, щоб визначити розмір кожного запиту.

Важливо, що визначити найкращу технологію неможливо, адже вибір залежить від умов і пріоритетів системи. Цей висновок було практично доведено за допомогою тестування.

Технологія gRPC є найпродуктивнішою у більшості випадків, де не немає вкладених і складних структур. Ця перевага проявляється завдяки протоколу HTTP/2 і формату Protobuffers. Він проявив чудові результати у функціях модифікацій даних. Незважаючи на переваги, на його реалізацію знадобиться

більше часу і коду. gRPC варто використовувати в програмах, де потрібна велика швидкість і ефективність.

GraphQL тримався посередині результатів, але його перевага проявляється в опрацюванні великої кількості даних. Зазвичай в системах необхідні певні поля, а не всі, з чим ця технологія справляється найкраще. Це дозволяє мінімізувати запити до бази даних і оптимізувати продукт. В решті запитів GraphQL проявляв себе гірше. Його розробка займає найменше часу і коду завдяки багатьом бібліотекам і фреймворкам, які ефективно опрацьовують дані.

Останньою є REST, яка залишається зручною завдяки своїй простоті і зрозумілості. В деяких запитах її метрики могли конкурувати з іншими технологіями, але все ж вона програє по багатьох часових показниках. При цьому вона залишається найлегшою для розуміння і найбільш підтримуваною. REST варто використовувати в простих і стандартних системах, де не потрібно великої гнучкості. Також її можна застосовувати для інтеграції з іншими системами.

Отже, технологію необхідно вибирати в залежності від вимог системи. Також можна спробувати їх комбінувати, щоб збільшити продуктивність всередині системи (gRPC) і забезпечити зручність для клієнтів (REST і GraphQL).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Codecademy. What is REST API (RESTful API). Codecademy. URL: https://www.codecademy.com/article/what-is-rest (дата звернення: 02.10.2025).

2. GraphQL | A query language for your API. GraphQL | A query language for your API. URL: https://graphql.org/ (дата звернення: 04.10.2025).

3. gRPC. URL: https://grpc.io/ (дата звернення: 04.10.2025).

4. What is an API (Application Programming Interface). GeeksforGeeks. URL: https://www.geeksforgeeks.org/software-testing/what-is-an-api/ (дата звернення: 02.10.2025).

5. What is an API? Use Cases and Benefits. Kong Inc. URL: https://konghq.com/blog/learning-center/what-is-api (дата звернення: 02.10.2025).

6. Навіщо потрібний API. ITSTEP Академія Online освіта. URL: https://cloud.itstep.org/blog/what-is-an-api-why-is-it-used-by-programmers-and-the-basics-of-working-with-it (дата звернення: 05.09.2025).

7. Що таке rest api: основні принципи та практики застосування. FoxmindEd. URL: https://foxminded.ua/shcho-take-rest-api/ (дата звернення: 05.09.2025).

8. The Postman Team. What Is a REST API? Examples, Uses, and Challenges. Postman. URL: https://blog.postman.com/rest-api-examples/ (дата звернення: 06.10.2025).

9. 8 examples of products transforming industries with GraphQL. Hygraph | The Best Headless CMS for Fast Implementation | Hygraph. URL: https://hygraph.com/blog/products-using-graphql (дата звернення: 04.10.2025).

10. Uber’s Next Gen Push Platform on gRPC. Uber Blog. URL: https://www.uber.com/en-PL/blog/ubers-next-gen-push-platform-on-grpc/ (дата звернення: 10.10.2025).

11. Nakov S. Fundamentals of computer programming with C#: the bulgarian C# book – Sofia, Bulgaria, 2013 – 1132 с.

12. C# Guide - .NET managed language. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 14.09.2025).

13. Apache JMeter. URL: https://jmeter.apache.org/ (дата звернення: 10.10.2025).

14. Postman. URL: https://www.postman.com/ (дата звернення: 04.10.2025).

15. Visual Studio. URL: https://visualstudio.microsoft.com/ (дата звернення: 05.08.2025).

16. Tutorial: Create a minimal API with ASP.NET Core. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api?view=aspnetcore-10.0&tabs=visual-studio (дата звернення: 02.10.2025).

17. ChilliCream. Get started with GraphQL in .NET Core - Hot Chocolate v13. ChilliCream GraphQL Platform. URL: https://chillicream.com/docs/hotchocolate/v13/get-started-with-graphql-in-net-core (дата звернення: 05.10.2025).

18. Maria. Building GraphQL APIs with C# and Hot Chocolate. DEV Community. URL: https://dev.to/chakewitz/building-graphql-apis-with-c-and-hot-chocolate-4n4g (дата звернення: 06.10.2025).

19. gRPC services with C#. Microsoft Learn. URL: https://learn.microsoft.com/en-gb/aspnet/core/grpc/basics?view=aspnetcore-10.0 (дата звернення: 15.10.2025).

20. JMeter в тестуванні. Онлайн-курси від компанії QATestLab. URL: https://training.qatestlab.com/blog/technical-articles/using-jmeter-in-testing/ (дата звернення: 02.11.2025).

21. Яковина В. С. Сенів М. М. Основи теорії надійності програмних систем : Навч. посіб. Львів : Вид-во Львів. політехніки, 2020. 248 с.

22. В. А. Павлиш, Л. К. Гліненко, Н. Б. Шаховська. Основи інформаційних технологій і систем : Підручник. Львів : Вид-во Львів. політехніки, 2018. 620 с.

23. Є. В. Левус, Т. А. Марусенкова, О. О. Нитребич. Життєвий цикл програмного забезпечення : Навч. посіб. Львів : Вид-во Львів. політехніки, 217. 208 с.

24. Білас О. Є. Якість програмного забезпечення та тестування : Навч. посіб. Львів : Вид-во Львів. політехніки, 2011. 216 с.

25. KMS Solutions. API Load Testing: Step-by-step Guide On How To Load Test Your APIs. Medium. URL: https://medium.com/@KMSSolutions/api-load-testing-step-by-step-guide-on-how-to-load-test-your-apis-decc27060cf2 (дата звернення: 02.11.2025).

26. Grafana Labs. API load testing: A beginner's guide. Grafana Labs. URL: https://grafana.com/blog/2024/01/30/api-load-testing/ (дата звернення: 07.11.2025).

27. Тестування api: основні типи та огляд головних інструментів. FoxmindEd. URL: https://foxminded.ua/testuvanniya-api/ (дата звернення: 03.11.2025).

28. Тестування API з Swagger: як перевірити функціональність API. Hillel Blog. URL: https://blog.ithillel.ua/articles/api-testing-with-swagger (дата звернення: 02.11.2025).

29. Top API Metrics You Should Monitor for Performance. DigitalAPI. URL: https://www.digitalapi.ai/blogs/api-metrics (дата звернення: 29.10.2025).

30. Understanding API Metrics and What to Monitor in an API. Middleware. URL: https://middleware.io/blog/api-metrics/ (дата звернення: 29.10.2025).

ДОДАТКИ

ДОДАТОК А

Код моделей для REST і GraphQL

BaseModel.cs:
```
public abstract class BaseModel
{
   public int Id { get; set; }
   public DateTime CreatedAt { get; set; } =
DateTime.UtcNow;
   public DateTime UpdatedAt { get; set; } =
DateTime.UtcNow;
}
```

Category.cs:
```
public class Category : BaseModel
{
   [Required]
   public required string Name { get; set; }
}
```

Order.cs:
```
public class Order : BaseModel
{
   [Required]
   public decimal TotalPrice { get; set; }
   [Required]
   public required OrderStatus Status { get; set; } =
OrderStatus.New;
   [Required]
   public int UserId { get; set; }
   public User? User { get; set; }
   public ICollection<OrderItem> Items { get; set; } =
new List<OrderItem>();
}
```

User.cs:
```
public class User : BaseModel
{
   [Required]
   public required string FullName { get; set; }
   public string? Email { get; set; }

   public UserAddress? Address { get; set; }
   public ICollection<Order>? Orders { get; set; }
}
```

UserAddress.cs:
```
public class UserAddress : BaseModel
{
   public string? City { get; set; }
   public string? Street { get; set; }

   public int UserId { get; set; }
}
```

Product.cs:
```
public class Product : BaseModel
{
   [Required]
   public required string Name { get; set; }
   public string? Description { get; set; }
   [Required]
   public required decimal Price { get; set; }

   [Required]
   public int CategoryId { get; set; }

   public Category? Category { get; set; }
}
```

OrderItem.cs:
```
public class OrderItem : BaseModel
{
   [Required]
   public int Quantity { get; set; }
   [Required]
   public decimal TotalPrice { get; set; }

   [Required]
   public int OrderId { get; set; }
   [Required]
   public int ProductId { get; set; }

   public Product? Product { get; set; }
   public Order? Order { get; set; }
}
```

ДОДАТОК Б

Код інтерфейсів сервісів для REST API

IOrderService.cs:

```
public interface IOrderService
{
    Task<IEnumerable<Order>> GetAll();
    Task<Order?> GetById(int id);
    Task<PagedResult<Order>> GetByFilter(OrderStatus? status, DateTime? dateFrom, DateTime? dateTo, string?
sortBy, string? sortDir, int pageSize, int page);
    Task<OrderStatistic> GetStats();
    Task<Order> Create(OrderCreateDto order);
}
```

IProductService.cs:

```
public interface IProductService
{
    Task<IEnumerable<Product>> GetAll();
    Task<Product?> GetById(int id);
    Task<Product> Create(ProductCreateDto product);
    Task<Product?> Update(int id, JsonPatchDocument<Product> patch);
    Task<bool> Delete(int id);
}
```

ДОДАТОК В

Код сервісів для REST API

OrderService.cs:

```
public class OrderService : IOrderService
{
   private readonly AppDbContext _db;

   public OrderService(AppDbContext db)
   {
      _db = db;
   }

   public async Task<IEnumerable<Order>> GetAll()
   {
      return await _db.Orders
         .Include(o => o.Items)
            .ThenInclude(i => i.Product)
               .ThenInclude(p => p.Category)
         .Include(o => o.User)
            .ThenInclude(u => u.Address)
         .ToListAsync();
   }

   public async Task<Order?> GetById(int id)
   {
      return await _db.Orders
         .Include(o => o.User)
         .FirstOrDefaultAsync(o => o.Id == id);
   }

   public async Task<PagedResult<Order>>
GetByFilter(OrderStatus? status, DateTime?
dateFrom, DateTime? dateTo, string? sortBy, string?
sortDir, int pageSize, int page)
   {
      var query = _db.Orders.AsQueryable();
      if (status.HasValue)
      {
         query = query.Where(o => o.Status == status);
      }
      if (dateFrom.HasValue)
      {
         query = query.Where(o => o.CreatedAt >=
dateFrom.Value);
      }
      if (dateTo.HasValue)
      {
         query = query.Where(o => o.CreatedAt <=
dateTo.Value);
      }
      var sortProperty = sortBy?.ToLowerInvariant()
?? "createdat";
      var isAscending = sortDir?.ToLowerInvariant()
== "asc";
      Expression<Func<Order, object>> keySelector =
sortProperty switch
      {
         "status" => o => o.Status,
         "totalprice" => o => o.TotalPrice,
         _ => o => o.CreatedAt
      };
      query = isAscending
       ? query.OrderBy(keySelector)
       : query.OrderByDescending(keySelector);
      var total = await query.CountAsync();

      var items = await query
         .Skip((page - 1) * pageSize)
         .Take(pageSize)
         .Include(o => o.Items)
            .ThenInclude(i => i.Product)
         .ToListAsync();
      return new PagedResult<Order>
      {
         Items = items,
         Page = page,
         PageSize = pageSize,
         TotalItems = total,
         TotalPages = (int)Math.Ceiling(total /
(double)pageSize)
      };
   }

   public async Task<OrderStatistic> GetStats()
   {
      var totalOrders = await
_db.Orders.CountAsync();
      var totalRevenue = await
_db.Orders.SumAsync(o => o.TotalPrice);
      var avgOrderAmount = totalOrders > 0 ?
totalRevenue / totalOrders : 0m;
      return new OrderStatistic
      {
         TotalOrders = totalOrders,
         TotalUsers = await _db.Users.CountAsync(),
         TotalRevenue = totalRevenue,
         AvgOrderAmount = avgOrderAmount
      };
   }

   public async Task<Order> Create(OrderCreateDto
orderDto)
   {
      var order = new Order
      {
         TotalPrice = orderDto.TotalPrice,
         Status = orderDto.Status,
         UserId = orderDto.UserId,
         Items = orderDto.Items.Select(
            i => new OrderItem
            {
               ProductId = i.ProductId,
               Quantity = i.Quantity,
               TotalPrice = i.TotalPrice
            }
         ).ToList()
      };
      _db.Orders.Add(order);
      await _db.SaveChangesAsync();
      return order;
```

```
    }
}
```

ProductService.cs:

```
public class ProductService : IProductService
{
    private readonly AppDbContext _db;
    public ProductService(AppDbContext db)
    {
        _db = db;
    }
    public async Task<IEnumerable<Product>>
GetAll()
    {
        return await _db.Products
            .Include(p => p.Category)
            .AsNoTracking()
            .ToListAsync();
    }
    public async Task<Product?> GetById(int id)
    {
        return await _db.Products
            .Include(p => p.Category)
            .FirstOrDefaultAsync(p => p.Id == id);
    }
    public async Task<Product>
Create(ProductCreateDto productDto)
    {
        var product = new Product
        {
            Name = productDto.Name,
            Price = productDto.Price,
            Description = productDto.Description,
            CategoryId = productDto.CategoryId,
        };
        _db.Products.Add(product);
        await _db.SaveChangesAsync();
        return product;
    }
    public async Task<Product?> Update(int id,
JsonPatchDocument<Product> patch)
    {
        var product = await _db.Products
                .Include(p => p.Category)
                .FirstOrDefaultAsync(p => p.Id == id);
        if (product == null)
        {
            return null;
        }
        var modelState = new ModelStateDictionary();
        patch.ApplyTo(product, modelState);
        if (!modelState.IsValid)
        {
            return null;
        }
        product.UpdatedAt = DateTime.UtcNow;
        await _db.SaveChangesAsync();
        return product;
    }

    public async Task<bool> Delete(int id)
    {
        var product = await _db.Products.FindAsync(id);
        if (product == null)
        {
            return false;
        }
        bool isUsedInOrders = await
_db.OrderItems.AnyAsync(oi => oi.ProductId == id);
        if (isUsedInOrders)
        {
            return false;
        }
        _db.Products.Remove(product);
        await _db.SaveChangesAsync();
        return true;
    }
}
```

ДОДАТОК Г

Код контролерів для REST API

OrderController.cs:

```
[ApiController]
[Route("api/orders")]
[Produces("application/json")]
public class OrderController : ControllerBase
{
   private readonly IOrderService _orderService;

   public OrderController(IOrderService orderService)
   {
      _orderService = orderService;
   }

   [HttpGet]

[ProducesResponseType(StatusCodes.Status200OK,
Type = typeof(IEnumerable<Order>))]
   public async
Task<ActionResult<IEnumerable<Order>>>
GetAllOrders()
   {
      var orders = await _orderService.GetAll();
      return Ok(orders);
   }

   [HttpGet("{id:int}", Name =
nameof(GetOrderById))]

[ProducesResponseType(StatusCodes.Status200OK,
Type = typeof(Order))]

[ProducesResponseType(StatusCodes.Status404NotFo
und)]
   public async Task<ActionResult<Order>>
GetOrderById(int id)
   {
      var order = await _orderService.GetById(id);
      return order == null ? NotFound() : Ok(order);
   }

   [HttpGet("search")]

[ProducesResponseType(StatusCodes.Status200OK,
Type = typeof(PagedResult<Order>))]
   public async
Task<ActionResult<PagedResult<Order>>>
GetFilteredOrders(
      [FromQuery] OrderStatus? status,
      [FromQuery] DateTime? dateFrom,
      [FromQuery] DateTime? dateTo,
      [FromQuery] string? sortBy,
      [FromQuery] string? sortDir = "desc",
      [FromQuery] int page = 1,
      [FromQuery] int pageSize = 10)
   {
      var orders = await
_orderService.GetByFilter(status, dateFrom, dateTo,
sortBy, sortDir, page, pageSize);
      return Ok(orders);
   }

   [HttpGet("stats")]

[ProducesResponseType(StatusCodes.Status200OK,
Type = typeof(OrderStatistic))]
   public async Task<ActionResult<OrderStatistic>>
GetOrdersStats()
   {
      var stats = await _orderService.GetStats();
      return Ok(stats);
   }

   [HttpPost]

[ProducesResponseType(StatusCodes.Status201Creat
ed, Type = typeof(Order))]

[ProducesResponseType(StatusCodes.Status400BadR
equest)]
   public async Task<IActionResult>
CreateOrder([FromBody] OrderCreateDto order)
   {
      if (order == null || !ModelState.IsValid)
      {
         return BadRequest(ModelState);
      }
      var createdOrder = await
_orderService.Create(order);
      return CreatedAtAction(nameof(GetOrderById),
new { id = createdOrder.Id }, createdOrder);
   }
}
```

ProductController.cs:

```
[ApiController]
[Route("api/products")]
[Produces("application/json")]
public class ProductController : ControllerBase
{
   private readonly IProductService _productService;

   public ProductController(IProductService
productService)
   {
      _productService = productService;
   }

   [HttpGet]

[ProducesResponseType(StatusCodes.Status200OK,
Type = typeof(IEnumerable<Product>))]
   public async
Task<ActionResult<IEnumerable<Product>>>
GetAllProducts()
   {
      var products = await _productService.GetAll();
      return Ok(products);
   }
```

```
    [HttpGet("{id:int}", Name =
nameof(GetProductById))]

[ProducesResponseType(StatusCodes.Status200OK,
Type = typeof(Product))]

[ProducesResponseType(StatusCodes.Status404NotFo
und)]
    public async Task<ActionResult<Product>>
GetProductById(int id)
    {
        var product = await
_productService.GetById(id);
        return product == null ? NotFound() :
Ok(product);
    }

    [HttpPost]

[ProducesResponseType(StatusCodes.Status201Creat
ed, Type = typeof(Product))]

[ProducesResponseType(StatusCodes.Status400BadR
equest)]
    public async Task<IActionResult>
CreateProduct([FromBody] ProductCreateDto
product)
    {
        if (product == null || !ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        var createdProduct = await
_productService.Create(product);
        return
CreatedAtAction(nameof(GetProductById), new { id
= createdProduct.Id }, createdProduct);
    }

    [HttpPatch("{id:int}")]

[ProducesResponseType(StatusCodes.Status200OK,
Type = typeof(Product))]

[ProducesResponseType(StatusCodes.Status404NotFo
und)]

[ProducesResponseType(StatusCodes.Status400BadR
equest)]
    public async Task<IActionResult>
UpdateProduct(int id, [FromBody]
JsonPatchDocument<Product> patch)
    {
        if (patch == null || !ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        var result = await _productService.Update(id,
patch);
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        return result == null ? NotFound() : Ok(result);
    }

    [HttpDelete("{id:int}")]

[ProducesResponseType(StatusCodes.Status204NoCo
ntent)]

[ProducesResponseType(StatusCodes.Status404NotFo
und)]
    public async Task<IActionResult>
DeleteProduct(int id)
    {
        var isDeleted = await _productService.Delete(id);
        return isDeleted ? NoContent() : NotFound();
    }
}
```

ДОДАТОК Д

Код конфігураційного файлу для старту програми

```
var builder = WebApplication.CreateBuilder(args);
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

// DB
builder.Services.AddDbContextFactory<AppDbContext>(options =>
    options.UseSqlServer(connectionString));

// REST
builder.Services.AddControllers().AddNewtonsoftJson();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddScoped<IOrderService, OrderService>();
builder.Services.AddScoped<IProductService, ProductService>();

// gRPC
builder.Services.AddGrpc();
builder.Services.AddGrpcReflection();

// GraphQL
builder.Services
    .AddGraphQLServer()
    .AddQueryType(d => d.Name("Query"))
        .AddTypeExtension<OrderQueries>()
        .AddTypeExtension<ProductQueries>()
    .AddMutationType(d => d.Name("Mutation"))
        .AddTypeExtension<OrderMutations>()
        .AddTypeExtension<ProductMutations>()
    .AddFiltering()
    .AddSorting()
    .AddProjections();

var app = builder.Build();

// Swagger
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
    app.MapGrpcReflectionService();      // gRPC Reflection
}

app.UseRouting();
app.UseHttpsRedirection();
app.UseGrpcWeb();

// ENDPOINTS
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers(); // REST
    endpoints.MapGraphQL();     // GraphQL
    endpoints.MapGrpcService<OrderGrpcService>().EnableGrpcWeb();    // gRPC
    endpoints.MapGrpcService<ProductGrpcService>().EnableGrpcWeb();    // gRPC
});

app.Run();
```

ДОДАТОК Е

Код запитів (Queries) для GraphQL API

OrderQueries.cs:

```
[ExtendObjectType("Query")]
public class OrderQueries
{
    [UseDbContext(typeof(AppDbContext))]
    [UseOffsetPaging]
    // [UseProjection]
    [UseFiltering]
    [UseSorting]
    public IQueryable<Order>
GetOrders([ScopedService] AppDbContext context)
    {
        return context.Orders
            .Include(o => o.User)
                .ThenInclude(u => u.Address)
            .Include(o => o.Items)
                .ThenInclude(i => i.Product)
                    .ThenInclude(p => p.Category)
            .AsNoTracking();
    }

    [UseDbContext(typeof(AppDbContext))]
    [UseProjection]
    public async Task<Order?> GetOrderById(int id,
[ScopedService] AppDbContext context)
    {
        return await context.Orders.FindAsync(id);
    }

    [UseDbContext(typeof(AppDbContext))]
    [UseProjection]
    public async Task<OrderStatistic?>
GetOrderStatistic([ScopedService] AppDbContext
context)
    {
        var totalOrders = await
context.Orders.CountAsync();
        var totalRevenue = await
context.Orders.SumAsync(o => o.TotalPrice);
        var avgOrderAmount = totalOrders > 0 ?
totalRevenue / totalOrders : 0m;
        return new OrderStatistic
        {
            TotalOrders = totalOrders,
            TotalUsers = await
context.Users.CountAsync(),
            TotalRevenue = totalRevenue,
            AvgOrderAmount = avgOrderAmount
        };
    }
}
```

ProductQueries.cs:

```
[ExtendObjectType("Query")]
public class ProductQueries
{
    [UseDbContext(typeof(AppDbContext))]
    [UseOffsetPaging]
    [UseProjection]
    [UseFiltering]
    [UseSorting]
    public IQueryable<Product>
GetProducts([ScopedService] AppDbContext context)
    {
        return context.Products.AsNoTracking();
    }

    [UseDbContext(typeof(AppDbContext))]
    [UseProjection]
    public async Task<Product?> GetProductById(int
id, [ScopedService] AppDbContext context)
    {
        return await context.Products.FindAsync(id);
    }
}
```

ДОДАТОК Ж

Код мутацій (Mutations) для GraphQL API

OrderMutations.cs:

```
[ExtendObjectType("Mutation")]
public class OrderMutations
{
   [UseDbContext(typeof(AppDbContext))]
   public async Task<Order>
CreateOrder(OrderCreateDto orderDto,
[ScopedService] AppDbContext context)
   {
      var order = new Order
      {
         TotalPrice = orderDto.TotalPrice,
         Status = orderDto.Status,
         UserId = orderDto.UserId,
         Items = orderDto.Items.Select(
            i => new OrderItem
            {
               ProductId = i.ProductId,
               Quantity = i.Quantity,
               TotalPrice = i.TotalPrice
            }
         ).ToList()
      };
      context.Orders.Add(order);
      await context.SaveChangesAsync();
      return order;
   }
}
```

ProductMutations.cs:

```
[ExtendObjectType("Mutation")]
public class ProductMutations
{
   [UseDbContext(typeof(AppDbContext))]
   public async Task<Product>
CreateProduct(ProductCreateDto productDto,
[ScopedService] AppDbContext context)
   {
      var product = new Product
      {
         Name = productDto.Name,
         Price = productDto.Price,
         Description = productDto.Description,
         CategoryId = productDto.CategoryId,
      };
      context.Products.Add(product);
      await context.SaveChangesAsync();
      return product;
   }

   [UseDbContext(typeof(AppDbContext))]
   public async Task<Product> UpdateProduct(int id,
ProductUpdateDto productDto, [ScopedService]
AppDbContext context)
   {
      var product = await
context.Products.FindAsync(id);
      if (product == null)
      {
         return null;
      }
      if (productDto.Name != null)
      {
         product.Name = productDto.Name;
      }
      if (productDto.Price.HasValue)
      {
         product.Price = productDto.Price.Value;
      }
      if (productDto.Description != null)
      {
         product.Description = productDto.Description;
      }
      if (productDto.CategoryId.HasValue)
      {
         var categoryExists = await
context.Categories.AnyAsync(c => c.Id ==
productDto.CategoryId.Value);
         if (categoryExists)
         {
            product.CategoryId =
productDto.CategoryId.Value;
         }
      }
      await context.SaveChangesAsync();
      return product;
   }

   [UseDbContext(typeof(AppDbContext))]
   public async Task<bool> DeleteProduct(int id,
[ScopedService] AppDbContext context)
   {
      var product = await
context.Products.FindAsync(id);
      if (product == null)
      {
         return false;
      }
      bool isUsedInOrders = await
context.OrderItems.AnyAsync(oi => oi.ProductId ==
id);
      if (isUsedInOrders)
      {
         return false;
      }
      context.Products.Remove(product);
      await context.SaveChangesAsync();
      return true;
   }
```

ДОДАТОК И

Код файлу .proto для gRPC API

```
syntax = "proto3";

import "google/protobuf/timestamp.proto";
import "google/protobuf/empty.proto";

option csharp_namespace =
"OrdersApiComparison.gRPC.Protos";

package orders;

service OrderService {
	rpc GetOrders (google.protobuf.Empty)
returns (GetOrdersResponse);
	rpc GetOrderById (GetByIdRequest) returns
(OrderModel);
	rpc GetByFilter (GetByFilterRequest) returns
(PagedOrderResult);
	rpc GetStats (google.protobuf.Empty)
returns(OrderStatisticModel);
	rpc CreateOrder (CreateOrderRequest)
returns (OrderModel);
}

service ProductService {
   rpc GetProducts (google.protobuf.Empty) returns
(GetProductsResponse);
   rpc GetProductById (GetByIdRequest) returns
(ProductModel);
   rpc CreateProduct (CreateProductRequest) returns
(ProductModel);
   rpc UpdateProduct (UpdateProductRequest) returns
(ProductModel);
   rpc DeleteProduct (GetByIdRequest) returns
(DeleteResponse);
}

message OrderModel {
   int32 id = 1;
   string total_price = 2;
   string status = 3;
   google.protobuf.Timestamp created_at = 4;
   google.protobuf.Timestamp updated_at = 5;

   UserModel user = 6;
}

message OrderDetailsModel {
   int32 id = 1;
   string total_price = 2;
   string status = 3;
   google.protobuf.Timestamp created_at = 4;
   google.protobuf.Timestamp updated_at = 5;

   UserDetailsModel user = 6;
   repeated OrderItemModel items = 7;
}

message OrderItemModel {
   int32 id = 1;
   int32 quantity = 2;
   string total_price = 3;
   google.protobuf.Timestamp created_at = 4;
   google.protobuf.Timestamp updated_at = 5;

   ProductModel product = 6;
}

message ProductModel {
   int32 id = 1;
   string name = 2;
   string description = 3;
   string price = 4;
   google.protobuf.Timestamp created_at = 5;
   google.protobuf.Timestamp updated_at = 6;

   CategoryModel category = 7;
}

message CreateProductRequest {
   string name = 1;
   string description = 2;
   string price = 3;

   int32 categoryId = 4;
}

message UpdateProductRequest {
   int32 id = 1;
   string name = 2;
   string description = 3;
   string price = 4;

   int32 categoryId = 5;
}

message CategoryModel {
   int32 id = 1;
   string name = 2;
   google.protobuf.Timestamp created_at = 3;
   google.protobuf.Timestamp updated_at = 4;
}

message UserModel {
   int32 id = 1;
   string full_name = 2;
   string email = 3;
   google.protobuf.Timestamp created_at = 4;
   google.protobuf.Timestamp updated_at = 5;
}

message UserDetailsModel {
   int32 id = 1;
   string full_name = 2;
   string email = 3;
   google.protobuf.Timestamp created_at = 4;
   google.protobuf.Timestamp updated_at = 5;
   AddressModel address = 6;
}
```

```
message AddressModel {
   int32 id = 1;
   string street = 2;
   string city = 3;
   google.protobuf.Timestamp created_at = 4;
   google.protobuf.Timestamp updated_at = 5;
}

message GetByIdRequest {
   int32 id = 1;
}

message GetOrdersResponse {
   repeated OrderDetailsModel orders = 1;
}

message GetProductsResponse {
   repeated ProductModel products = 1;
}

message GetByFilterRequest {
   string status = 1;
   google.protobuf.Timestamp date_from = 2;
   google.protobuf.Timestamp date_to = 3;

   string sort_by = 4;
   string sort_dir = 5;

   int32 page_size = 6;
   int32 page = 7;
}
```

```
message PagedOrderResult {
   int32 page = 1;
   int32 page_size = 2;
   int32 total_items = 3;
   int32 total_pages = 4;
   repeated OrderModel items = 5;
}

message OrderStatisticModel {
   int32 total_orders = 1;
   int32 total_users = 2;
   string total_revenue = 3;
   string avg_order_amount = 4;
}

message CreateOrderRequest {
   string total_price = 1;
   string status = 2;
   int32 user_id = 3;
   repeated CreateOrderItemRequest items = 4;
}

message CreateOrderItemRequest {
   int32 product_id = 1;
   int32 quantity = 2;
   string total_price = 3;
}

message DeleteResponse {
   bool success = 1;
}
```

ДОДАТОК К

Код сервісів для gRPC API

OrderGrpcService.cs:

```
public class OrderGrpcService :
OrderService.OrderServiceBase
{
   private readonly AppDbContext _db;

   public OrderGrpcService(AppDbContext db)
   {
      _db = db;
   }

   public override async Task<GetOrdersResponse>
GetOrders(Empty request, ServerCallContext context)
   {
      var orders = await _db.Orders
         .Include(o => o.User)
            .ThenInclude(u => u.Address)
         .Include(o => o.Items)
            .ThenInclude(i => i.Product)
               .ThenInclude(p => p.Category)
         .ToListAsync();
      var response = new GetOrdersResponse();

response.Orders.AddRange(orders.Select(MapOrderD
etails));
      return response;
   }

   public override async Task<OrderModel>
GetOrderById(GetByIdRequest request,
ServerCallContext context)
   {
      var order = await _db.Orders
         .Include(o => o.User)
         .FirstOrDefaultAsync(o => o.Id == request.Id);
      if (order == null)
      {
         throw new RpcException(new
Status(StatusCode.NotFound, "Order not found"));
      }
      return MapOrder(order);
   }

   public override async Task<PagedOrderResult>
GetByFilter(GetByFilterRequest request,
ServerCallContext context)
   {
      var query = _db.Orders.AsQueryable();
      if (!string.IsNullOrWhiteSpace(request.Status))
      {
         query = query.Where(o => o.Status.ToString()
== request.Status);
      }
      if (request.DateFrom != null)
      {
         query = query.Where(o => o.CreatedAt >=
request.DateFrom.ToDateTime());
      }
      if (request.DateTo != null)
      {
         query = query.Where(o => o.CreatedAt <=
request.DateTo.ToDateTime());
      }
      if (request.SortBy == "totalprice")
      {
         query = request.SortDir == "desc" ?
query.OrderByDescending(o => o.TotalPrice) :
query.OrderBy(o => o.TotalPrice);
      }
      else if (request.SortBy == "status")
      {
         query = request.SortDir == "desc" ?
query.OrderByDescending(o => o.Status) :
query.OrderBy(o => o.Status);
      }
      else
      {
         query = request.SortDir == "desc" ?
query.OrderByDescending(o => o.CreatedAt) :
query.OrderBy(o => o.CreatedAt);
      }
      var totalItems = await query.CountAsync();
      var totalPages = (int)Math.Ceiling(totalItems /
(double)request.PageSize);
      var items = await query
         .Include(o => o.User)
         .Select(o => new OrderModel
         {
            Id = o.Id,
            TotalPrice = o.TotalPrice.ToString(),
            Status = o.Status.ToString(),
            CreatedAt =
Timestamp.FromDateTime(o.CreatedAt.ToUniversal
Time()),
            UpdatedAt =
Timestamp.FromDateTime(o.UpdatedAt.ToUniversal
Time()),
            User = new UserModel
            {
               Id = o.User.Id,
               FullName = o.User.FullName,
               Email = o.User.Email,
               CreatedAt =
Timestamp.FromDateTime(o.User.CreatedAt.ToUniv
ersalTime()),
               UpdatedAt =
Timestamp.FromDateTime(o.User.UpdatedAt.ToUniv
ersalTime())
            }
         })
         .Skip((request.Page - 1) * request.PageSize)
         .Take(request.PageSize)
         .AsNoTracking()
         .ToListAsync();
      var response = new PagedOrderResult
      {
         Page = request.Page,
         PageSize = request.PageSize,
```

```
            TotalItems = totalItems,
            TotalPages = totalPages
        };

        response.Items.AddRange(items);
        return response;
    }

    public override async Task<OrderStatisticModel>
GetStats(Empty request, ServerCallContext context)
    {
        var totalOrders = await
_db.Orders.CountAsync();
        var totalUsers = await _db.Users.CountAsync();
        var revenue = await _db.Orders.SumAsync(o =>
o.TotalPrice);
        return new OrderStatisticModel
        {
            TotalOrders = totalOrders,
            TotalUsers = totalUsers,
            TotalRevenue = revenue.ToString("F2"),
            AvgOrderAmount = totalOrders > 0 ? (revenue
/ totalOrders).ToString("F2") : "0"
        };
    }

    public override async Task<OrderModel>
CreateOrder(CreateOrderRequest request,
ServerCallContext context)
    {
        var user = await
_db.Users.FindAsync(request.UserId);
        if (user == null)
        {
            throw new RpcException(new
Status(StatusCode.NotFound, "User not found"));
        }
        var order = new Models.Order
        {
            UserId = request.UserId,
            Status = OrderStatus.New,
            CreatedAt = DateTime.UtcNow,
            UpdatedAt = DateTime.UtcNow,
            Items = new List<OrderItem>()
        };
        decimal totalPrice = 0m;
        foreach (var item in request.Items)
        {
            var product = await
_db.Products.FindAsync(item.ProductId);
            if (product == null)
            {
                throw new RpcException(new
Status(StatusCode.NotFound, $"Product not found"));
            }
            var orderItem = new OrderItem
            {
                ProductId = item.ProductId,
                Quantity = item.Quantity,
                TotalPrice = decimal.Parse(item.TotalPrice,
CultureInfo.InvariantCulture)
            };
            order.Items.Add(orderItem);
        }
        order.TotalPrice = totalPrice;
        _db.Orders.Add(order);
        await _db.SaveChangesAsync();
        return MapOrder(order);
    }

    private static OrderModel MapOrder(Models.Order
o) =>
    new OrderModel
    {
        Id = o.Id,
        TotalPrice = o.TotalPrice.ToString(),
        Status = o.Status.ToString(),
        CreatedAt =
Timestamp.FromDateTime(o.CreatedAt.ToUniversal
Time()),
        UpdatedAt =
Timestamp.FromDateTime(o.UpdatedAt.ToUniversal
Time()),
        User = new UserModel
        {
            Id = o.User.Id,
            FullName = o.User.FullName,
            Email = o.User.Email,
            CreatedAt =
Timestamp.FromDateTime(o.User.CreatedAt.ToUniv
ersalTime()),
            UpdatedAt =
Timestamp.FromDateTime(o.User.UpdatedAt.ToUniv
ersalTime())
        }
    };

    private static OrderDetailsModel
MapOrderDetails(Models.Order o)
    {
        var model = new OrderDetailsModel
        {
            Id = o.Id,
            TotalPrice = o.TotalPrice.ToString(),
            Status = o.Status.ToString(),
            CreatedAt =
Timestamp.FromDateTime(o.CreatedAt.ToUniversal
Time()),
            UpdatedAt =
Timestamp.FromDateTime(o.UpdatedAt.ToUniversal
Time()),
            User = new UserDetailsModel
            {
                Id = o.User.Id,
                FullName = o.User.FullName,
                Email = o.User.Email,
                CreatedAt =
Timestamp.FromDateTime(o.User.CreatedAt.ToUniv
ersalTime()),
                UpdatedAt =
Timestamp.FromDateTime(o.User.UpdatedAt.ToUniv
ersalTime()),
                Address = o.User.Address != null
                    ? new AddressModel
                    {
                        Id = o.User.Address.Id,
```

```
                Street = o.User.Address.Street,
                City = o.User.Address.City,
                CreatedAt =
Timestamp.FromDateTime(o.User.Address.CreatedAt
.ToUniversalTime()),
                UpdatedAt =
Timestamp.FromDateTime(o.User.Address.UpdatedA
t.ToUniversalTime())
            }
            : null
        }
    };

    model.Items.AddRange(o.Items.Select(i =>
        new OrderItemModel
        {
            Id = i.Id,
            Quantity = i.Quantity,
            TotalPrice = i.TotalPrice.ToString(),
            CreatedAt =
Timestamp.FromDateTime(i.CreatedAt.ToUniversalT
ime()),
            UpdatedAt =
Timestamp.FromDateTime(i.UpdatedAt.ToUniversal
Time()),
            Product = new ProductModel
            {
                Id = i.Product.Id,
                Name = i.Product.Name,
                Description = i.Product.Description,
                Price = i.Product.Price.ToString(),
                CreatedAt =
Timestamp.FromDateTime(i.Product.CreatedAt.ToUn
iversalTime()),
                UpdatedAt =
Timestamp.FromDateTime(i.Product.UpdatedAt.ToU
niversalTime()),
                Category = new CategoryModel
                {
                    Id = i.Product.Category.Id,
                    Name = i.Product.Category.Name,
                    CreatedAt =
Timestamp.FromDateTime(i.Product.Category.Create
dAt.ToUniversalTime()),
                    UpdatedAt =
Timestamp.FromDateTime(i.Product.Category.Update
dAt.ToUniversalTime())
                }
            }
        }));

    return model;
  }
}

ProductGrpcService.cs:
public class ProductGrpcService :
ProductService.ProductServiceBase
{
  private readonly AppDbContext _db;
  public ProductGrpcService(AppDbContext db)
  {
    _db = db;
  }
  public override async Task<GetProductsResponse>
GetProducts(Empty request, ServerCallContext
context)
  {
    var products = await _db.Products
      .Include(p => p.Category)
      .Select(p => new ProductModel
      {
        Id = p.Id,
        Name = p.Name,
        Description = p.Description,
        Price = p.Price.ToString(),
        CreatedAt =
Timestamp.FromDateTime(p.CreatedAt.ToUniversal
Time()),
        UpdatedAt =
Timestamp.FromDateTime(p.UpdatedAt.ToUniversal
Time()),
        Category = new CategoryModel
        {
          Id = p.Category.Id,
          Name = p.Category.Name,
          CreatedAt =
Timestamp.FromDateTime(p.Category.CreatedAt.To
UniversalTime()),
          UpdatedAt =
Timestamp.FromDateTime(p.Category.UpdatedAt.To
UniversalTime())
        }
      })
      .ToListAsync();
    var response = new GetProductsResponse();
    response.Products.AddRange(products);
    return response;
  }

  public override async Task<ProductModel>
GetProductById(GetByIdRequest request,
ServerCallContext context)
  {
    var product = await _db.Products
      .Include(p => p.Category)
      .Select(p => new ProductModel
      {
        Id = p.Id,
        Name = p.Name,
        Description = p.Description,
        Price = p.Price.ToString(),
        CreatedAt =
Timestamp.FromDateTime(p.CreatedAt.ToUniversal
Time()),
        UpdatedAt =
Timestamp.FromDateTime(p.UpdatedAt.ToUniversal
Time()),
        Category = new CategoryModel
        {
          Id = p.Category.Id,
          Name = p.Category.Name,
          CreatedAt =
Timestamp.FromDateTime(p.Category.CreatedAt.To
UniversalTime()),
```

```
                UpdatedAt =
Timestamp.FromDateTime(p.Category.UpdatedAt.To
UniversalTime())
            }
        })
        .FirstOrDefaultAsync(p => p.Id == request.Id);
    if (product == null)
    {
        throw new RpcException(new
Status(StatusCode.NotFound, "Product not found"));
    }
    return product;
  }

  public override async Task<ProductModel>
CreateProduct(CreateProductRequest request,
ServerCallContext context)
  {
    var p = new Models.Product
    {
        Name = request.Name,
        Description = request.Description,
        Price = decimal.Parse(request.Price,
CultureInfo.InvariantCulture),
        CategoryId = request.CategoryId,
        CreatedAt = DateTime.UtcNow,
        UpdatedAt = DateTime.UtcNow
    };
    _db.Products.Add(p);
    await _db.SaveChangesAsync();
    return new ProductModel
    {
        Price = p.Price.ToString(),
        Name = p.Name,
        Description = p.Description,
        CreatedAt =
Timestamp.FromDateTime(p.CreatedAt.ToUniversal
Time()),
        UpdatedAt =
Timestamp.FromDateTime(p.UpdatedAt.ToUniversal
Time()),
        Category = new CategoryModel
        {
            Id = p.Category.Id,
            Name = p.Category.Name,
            CreatedAt =
Timestamp.FromDateTime(p.Category.CreatedAt.To
UniversalTime()),
            UpdatedAt =
Timestamp.FromDateTime(p.Category.UpdatedAt.To
UniversalTime()),
        }
    };
  }

  public override async Task<ProductModel>
UpdateProduct(UpdateProductRequest request,
ServerCallContext context)
  {
    var p = await
_db.Products.FindAsync(request.Id);
    if (p == null)
    {
        throw new RpcException(new
Status(StatusCode.NotFound, "Product not found"));
    }
    p.Name = request.Name;
    p.Description = request.Description;
    p.Price = decimal.Parse(request.Price,
CultureInfo.InvariantCulture);
    p.CategoryId = request.CategoryId;
    p.UpdatedAt = DateTime.UtcNow;
    await _db.SaveChangesAsync();
    return new ProductModel
    {
        Price = p.Price.ToString(),
        Name = p.Name,
        Description = p.Description,
        CreatedAt =
Timestamp.FromDateTime(p.CreatedAt.ToUniversal
Time()),
        UpdatedAt =
Timestamp.FromDateTime(p.UpdatedAt.ToUniversal
Time()),
        Category = new CategoryModel
        {
            Id = p.Category.Id,
            Name = p.Category.Name,
            CreatedAt =
Timestamp.FromDateTime(p.Category.CreatedAt.To
UniversalTime()),
            UpdatedAt =
Timestamp.FromDateTime(p.Category.UpdatedAt.To
UniversalTime()),
        }
    };
  }

  public override async Task<DeleteResponse>
DeleteProduct(GetByIdRequest request,
ServerCallContext context)
  {
    var p = await
_db.Products.FindAsync(request.Id);
    if (p == null)
    {
        return new DeleteResponse { Success = false
};
    }
    _db.Products.Remove(p);
    await _db.SaveChangesAsync();
    return new DeleteResponse { Success = true };
  }
}
```