Міністерство освіти і науки України
Український державний університет науки і технологій

Комп'ютерних технологій і систем
(назва факультету)

Комп'ютерні інформаційні технології
(повна назва кафедри)

Пояснювальна записка

до кваліфікаційної роботи
ОС магістра
(ступінь вищої освіти)

на тему: Дослідження способів зменшення часу виконання запитів у RESTful API
за освітньою програмою Інженерія програмного забезпечення
зі спеціальності: 121 Інженерія програмного забезпечення

Виконав: студент групи: ІІ32321 Олексій ПОДЕДВОРНИЙ

Керівник: доц. Вадим АНДРЮЩЕНКО

Нормоконтролер: доц. Світлана ВОЛКОВА

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

Студент
(підпис)

Дніпро – 2025 рік

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

Computer Technologies and Systems
(faculty)

Computer information technology
(department)

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

on the topic: Research on Methods to Reduce Query Execution Time in RESTful APIs.
according to educational curriculum Software engineering»
in the Specialty: 121 Software engineering
(specialty and its code)

Done by the student of the group: PZ2321 Oleksii PODEDVORNYI
(name, surname)

Scientific Supervisor: Vadym ANDRIUSHCHENKO
(position, name, surname)

Normative controller: Svitlana VOLKOVA
(position, name, surname)

Supervisors

(Chapter title heading)	(position, name, surname)
(Chapter title heading)	(position, name, surname)
(Chapter title heading)	(position, name, surname)
(Chapter title heading)	(position, name, surname)

Dnipro – 2025

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: «Комп’ютерні технології і системи»
Кафедра: «Комп'ютерні інформаційні технології»
Рівень вищої освіти: магістр
Освітня програма: «12 Інженерія програмного забезпечення»
Спеціальність: «121 Інженерія програмного забезпечення»
(шифр та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КІТ
_____ Вадим ГОРЯЧКІН
(підпис) (Ім’я ПРІЗВИЩЕ)
_____ _____ 202__р

З А В Д А Н Н Я

на кваліфікаційну роботу магістра
(ступінь вищої освіти)

студенту Поддедворному Олексію Едуардовичу
(Прізвище, Ім’я По батькові)

1. Тема роботи: Дослідження способів зменшення часу виконання запитів у RESTful API.

Керівник роботи: Андрющенко Вадим Олександрович, доцент
(Прізвище, Ім’я, По батькові, науковий ступінь, вчене звання)

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

2. Строк подання студентом роботи: __.__.2024 р.
3. Вихідні дані до роботи: архітектура монолітного застосунку, побудованого на основі RESTful API; тестове середовище для вимірювання продуктивності; набір тестових сценаріїв; інструменти для збору та обробки даних
4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати):
4.1 Формулювання завдання дослідження
4.2 Методологія дослідження
4.3 Розробка програмного забезпечення
4.4 Тестування програмного забезпечення та початкові результати
4.5 Обробка та аналіз зібраних даних

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень):
5.1 Презентація
5.2 Відео роботи програми

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада консультанта	Завдання видав: (підпис консультанта, дата)	Завдання прийняв: (підпис студента, дата)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ		
2	Аналіз предметної області та постановка задачі		
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження		
4	Постановка задачі, технічне завдання		30%
5	Розробка інструментальних засобів дослідження		
6	Виконання досліджень		60%
7	Оформлення тез доповідей		
8	Оформлення пояснювальної записки		
9	Розробка демонстраційних матеріалів		100%
10	Подання кваліфікаційної роботи до кафедри		
11	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії		

Студент _____ Олексій ПОДЕДВОРНИЙ
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник роботи _____ Вадим АНДРЮЩЕНКО
(підпис) (Ім’я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра: 136 с., 37 рис., 7 джерел та 3 додатки.

Магістерська робота на тему *«Дослідження способів зменшення часу виконання запитів у RESTful API»* присвячена дослідженню та реалізації способів прискорення веб-додатків на базі монолітної архітектури з використанням ASP.NET Core.

Об’єктом дослідження є RESTful API монолітного додатка з Onion-архітектурою.

Метою роботи є зменшення часу виконання запитів у RESTful API шляхом впровадження ефективних комбінації способів прискор для підвищення продуктивності додатка.

Методи дослідження: теоретичний аналіз, експериментальний метод, статистичний аналіз (середні значення, стандартне відхилення), порівняльний аналіз продуктивності API.

Наукова новизна: досліджено та проаналізовано вплив способів прискорення на продуктивність RESTful API монолітного додатка, визначено ефективні рішення для покращення швидкодії без переходу на мікросервіси.

Результати роботи можуть бути застосовані для оптимізації продуктивності веб-додатків на базі RESTful API, що дозволить знизити час відгуку серверів та покращити користувацький досвід.

Результати дослідження: розроблено інструмент для збору метрик продуктивності, проведено експериментальний аналіз методів прискорення (кешування, стиснення даних, пагінація), сформульовано рекомендації щодо впровадження прискорення для підвищення продуктивності RESTful API.

Ключові слова: RESTFUL API, ASP.NET CORE, ONION-АРХІТЕКТУРА, КЕШУВАННЯ, СТИСНЕННЯ ДАНИХ, ПАГІНАЦІЯ, ПРОДУКТИВНІСТЬ, ЧАС ВИКОНАННЯ ЗАПИТІВ, ПРИСКОРЕННЯ.

ЗМІСТ

ВСТУП

RESTful API є одним із найпопулярніших підходів до створення веб-сервісів, забезпечуючи гнучку та масштабовану архітектуру для взаємодії між клієнтами та серверами. Однак зі зростанням обсягу даних і збільшенням кількості користувачів веб-додатків виникають проблеми, пов'язані з продуктивністю API, зокрема, збільшується час відгуку серверів на запити.

Проблема зменшення часу виконання запитів є важливим аспектом у контексті розроблення сучасних веб-додатків, де користувачі очікують на швидкий і стабільний відгук. Підвищення продуктивності RESTful API безпосередньо впливає на якість роботи додатків, покращує користувацький досвід і знижує навантаження на серверні ресурси.

Об'єктом дослідження є RESTful API для монолітного додатку, розробленого на базі ASP.NET Core з використанням Onion-архітектури. Головна мета — створення інструменту для збору метрик продуктивності запитів, які дозволять оцінити ефективність різних способів прискорення, таких як кешування, стиснення даних та пагінація. Реалізація включає компоненти для зберігання й обробки даних у базі MySQL, механізми вимірювання часу виконання запитів, а також налаштування глобальних параметрів для контролю параметрів прискорення у API.

1 ФОРМУЛЮВАННЯ ЗАВДАННЯ ДОСЛІДЖЕННЯ

1.1 Актуальність дослідження

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

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

1.2 Мета дослідження

Головною метою цього дослідження є розробка методології та програмного інструментарію для оцінки ефективності різних способів зменшення часу виконання запитів у RESTful API. Це передбачає створення механізмів для збору та аналізу метрик продуктивності запитів, визначення ключових факторів, що впливають на швидкість виконання запитів, та дослідження впливу технік прискорення, таких як кешування, стиснення даних та пагінація, на продуктивність додатків.

Для досягнення цієї мети необхідно впровадити системний підхід до збору метрик, розробити алгоритми вимірювання часу виконання запитів та проаналізувати отримані дані з використанням статистичних методів. Результати дослідження дадуть можливість запропонувати конкретні рекомендації щодо впровадження найбільш ефективних способів або комбінацій способів прискорення монолітних додатків.

1.3 Завдання дослідження

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати основні причини сповільнення роботи RESTful API у контексті монолітного додатка, дослідити чинники, що впливають на продуктивність;

– дослідити наявні способи зменшення часу виконання запитів в API, такі як кешування, стиснення даних, асинхронні операції, оптимізація запитів бази даних та ефективне керування потоками;

– розробити архітектуру системи для збору даних про продуктивність запитів, включно з методами моніторингу та аналізу метрик на рівні програми;

– реалізувати програмні засоби для збору даних про час виконання запитів;

– провести експериментальний аналіз продуктивності API до та після впровадження рішень щодо прискорення;

– сформулювати рекомендації щодо впровадження найбільш ефективних способів підвищення продуктивності.

1.5 Методи дослідження

Для проведення дослідження буде використано такі методи:

– теоретичний аналіз. Аналіз наукових праць та наявних рішень, пов'язаних з підвищенням продуктивності RESTful API у монолітних додатках;

– експериментальний метод. Розробка програмного інструментарію для збору та аналізу метрик продуктивності;

– статистичний метод. Обчислення середнього часу виконання запитів, стандартного відхилення та мінімальних/максимальних значень для оцінки стабільності та ефективності API;

– порівняльний аналіз. Оцінка ефективності підходів до прискорення до та після впровадження на основі експериментальних даних.

1.6 Очікувані результати

Очікуваними результатами дослідження є:

– аналіз продуктивності RESTful API та ідентифікація ключових вузьких місць;

– реалізація інструментарію для вимірювання часу виконання запитів у монолітних додатках;

– впровадження та тестування способів прискорення (кешування, стиснення, пагінація тощо);

– формулювання рекомендацій щодо підвищення продуктивності RESTful API.

1.7 Наукова новизна та практична значущість

Наукова новизна полягає у дослідженні впливу сучасних способів прискорення на продуктивність RESTful API монолітного додатка. Особливу увагу приділено використанню Onion-архітектури для впровадження прискорення.

Практична значущість полягає у можливості застосування результатів роботи для оптимізації продуктивності веб-додатків, що дозволяє підвищити швидкодію API без необхідності переходу до мікросервісної архітектури.

2 МЕТОДОЛОГІЯ ДОСЛІДЖЕННЯ

2.1 Теоретичний аналіз існуючих рішень

Підвищення продуктивності RESTful API є одним з найважливіших аспектів у сучасній розробці програмного забезпечення. Ця задача особливо актуальна для великих монолітних додатків, де від продуктивності API залежить стабільність та швидкодія системи в цілому. Існує декілька основних способів, які застосовуються для підвищення ефективності API, включаючи кешування, стиснення даних, пагінацію, асинхронну обробку та оптимізацію запитів до бази даних.

2.1.1 Кешування

Кешування є одним із найбільш ефективних способів покращення продуктивності API, оскільки дозволяє уникнути повторного звернення до бази даних або обробки складних запитів при кожному виклику. Існують різні стратегії кешування, включаючи кешування на стороні сервера та використання HTTP-заголовків (Cache-Control, ETag). Інший варіант — використання серверного кешування за допомогою таких інструментів, як Redis або Memcached. Це дозволяє зменшити навантаження на базу даних та підвищити швидкість обробки запитів.

До переваг кешування можна віднести:

- зменшення кількості звернень до бази даних;
- прискорення обробки повторних запитів;
- підвищення продуктивності API в умовах великої кількості запитів.

Серед недоліків можна визначити наступні:

- потребує налаштування та ретельного контролю оновлень кешу;
- може призвести до проблем з консистентністю даних, якщо оновлення не виконується належним чином.

2.1.2 Стиснення даних

Стиснення даних дозволяє зменшити об'єм переданих даних між клієнтом та сервером. Зменшення об'єму даних безпосередньо впливає на швидкість відповіді API. Одним із найбільш популярних методів стиснення є використання алгоритму Gzip. Також існує альтернативний алгоритм Brotli, який може бути ефективнішим за Gzip для певних типів даних. Застосування стиснення дозволяє підвищити швидкість передачі даних і зменшити затримки.

До переваг можна віднести наступне:

- зменшення розміру переданих даних;
- підвищення швидкості передачі даних у мережі;
- ефективне використання мережевих ресурсів.

Мінус ж наступні:

- додаткові витрати процесорного часу на стиснення та розпакування даних;
- менша ефективність при використанні для вже стиснених даних, таких як зображення чи відео.

2.1.3 Пагінація

При роботі з великими наборами даних важливо використовувати пагінацію для зменшення навантаження на сервер. Пагінація дозволяє користувачам отримувати дані частинами, зменшуючи обсяг кожного окремого запиту. Цей метод знижує кількість оброблюваних записів за раз та дозволяє уникнути проблем із продуктивністю. Пагінацію можна реалізувати за допомогою параметрів запиту, таких як ?limit=10&page=2, для контролю кількості записів, які повертаються на кожній сторінці.

Серед переваг можна виділити наступні:

- зниження кількості оброблюваних записів за раз;
- зменшення часу відповіді на запити.

До недоліків же відносять наступне:

– потребує додаткової логіки у клієнтському коді для обробки сторінок;
– може призвести до проблем з консистентністю, якщо дані оновлюються у реальному часі.

2.1.4 Оптимізація бази даних

Одним із основних джерел проблем з продуктивністю API є затримки при зверненні до бази даних. Для оптимізації цього аспекту рекомендується застосовувати індексацію полів, які часто використовуються у запитах, а також оптимізувати SQL-запити. Крім того, для великих додатків можна розглянути такі методи, як шардінг бази даних, що дозволяє розподілити навантаження на кілька серверів.

Переваги:

– підвищення швидкості виконання запитів;
– зниження навантаження на сервер бази даних.

Недоліки:

– складність налаштування та потреба у додаткових ресурсах.

2.1.5 Асинхронна обробка

Асинхронна обробка дозволяє виконувати операції у фоновому режимі без блокування основного потоку запиту, що підвищує швидкість обробки. Наприклад, при створенні складних операцій можна запустити їх у фоновому режимі та повернути клієнту попередню відповідь, сповіщаючи його про завершення задачі пізніше.

До переваг входить наступне:

– підвищення продуктивності за рахунок розподілу обчислень;
– зменшення затримок для користувачів.

Серед недоліків можна зазначити, що:

– більш складна логіка обробки запитів та викликів;
– потребує додаткової обробки асинхронних операцій і помилок.

2.1.6 Висновки щодо аналізу існуючих рішень

Проаналізовані способи прискорення RESTful API показують, що комбінування кількох підходів — кешування, стиснення даних та пагінації — може значно покращити продуктивність системи. Однак необхідно враховувати специфіку застосування кожного методу та можливі побічні ефекти. Для досягнення найкращих результатів потрібно проводити моніторинг і тестування під час кожного етапу прискорення.

Для подальших досліджень прискорення обраного API було обрано поєднання кешування, стиснення даних та пагінації.

Це рішення підкріплене результатами попередніх досліджень у галузі проектування та підвищення продуктивності веб-сервісів наведених у таких англомовних підручниках як "Designing Web APIs" (Brenda Jin, Saurabh Sahni & Amir Shevat) [1] та "RESTful Web APIs" (Leonard Richardson & Mike Amundsen) [2]. Ця література випускається видавництвом O’Reilly, що спеціалізується на виданні підручників пов’язаних з комп’ютерним ПЗ та технічною стороною.

2.2 Науковий метод дослідження

2.2.1 Формулювання гіпотези

Формулювання гіпотези є фундаментальним етапом будь-якого наукового дослідження. Гіпотеза є припущенням або теоретичним передбаченням, яке потребує перевірки за допомогою експериментів. У рамках даного дослідження основна гіпотеза може бути сформульована таким чином: «*Параметри прискорення, такі як кешування, стиснення даних (Gzip) та пагінація, за умов*

достатньої кількості ресурсів знижують час виконання запитів у середньому на X%».

Це припущення базується на тому, що зазначені засоби прискорення, перевірені в численних наукових роботах та підручниках, покращують роботу серверів та ефективність роботи веб-додатків. Наприклад, згідно з підручниками «Designing Web APIs» (Brenda Jin, Saurabh Sahni & Amir Shevat) [1] та «RESTful Web APIs» (Leonard Richardson & Mike Amundsen) [2], згадані методи прискорення вважаються найефективнішими для підвищення швидкості та ефективності API.

2.2.2 Вибір критеріїв оцінки та параметрів для аналізу

Основним критерієм, за яким оцінюється ефективність впроваджених рішень стосовно прискорення є *час виконання запиту (Execution Time)*. Цей параметр визначає швидкість відповіді API на запити клієнта і є ключовим показником продуктивності веб-додатків. Однак, для більш комплексного аналізу дослідження включає також декілька додаткових параметрів, які потенційно можуть впливати на час виконання та загальну продуктивність системи. Додаткові параметри включають:

- розмір респонсу (Response Size) — розмір даних, які передаються сервером клієнту. Прискорення шляхом стиснення даних (наприклад, Gzip) спрямована на зменшення цього параметра, що може позитивно вплинути на швидкість передачі;
- кількість оброблених записів (Record Count) — цей показник відображає, наскільки великі обсяги даних обробляються під час запиту. Чим більше оброблених записів, тим більше навантаження на сервер і можливі затримки;
- складність запиту (Query Complexity) — рівень вкладеності даних у респонсі та кількість зв’язків між об’єктами. Запити з високою складністю можуть вимагати більшого часу на обробку та передачу;

– стабільність результатів — визначається шляхом розрахунку стандартного відхилення часу виконання запитів, що дозволяє оцінити рівномірність продуктивності та виявити можливі коливання або нестабільність.

Отже, основний критерій один та акцент робиться на ньому - часі виконання запиту, але до уваги також беруться додаткові параметрах, які прямо чи опосередковано впливають на час виконання запиту, для глибшого розуміння впливу прискорення на продуктивність API.

2.2.3 Планування експерименту для перевірки гіпотези

Планування експерименту — це ключовий етап наукового методу, який передбачає визначення методології та підходів до перевірки гіпотези. Для даного дослідження було розроблено спеціальний ендпоінт, який дозволяє вмикати та вимикати певні способи прискорення. Цей підхід забезпечує можливість тестування з різними комбінаціями способів. У ході дослідження тестування проводиться у восьми комбінаціях, тобто різні варіації взаємодії таких способів прискорення між собою:

– без використання жодного способу;
– тільки кешування;
– тільки пагінація;
– тільки компресія;
– кешування та пагінація;
– кешування та компресія;
– пагінація та компресія;
– всі способи разом.

Кожен експеримент повторюється кілька разів для забезпечення точності та стабільності результатів. Параметри запитів та налаштувань прискорення фіксуються для подальшого аналізу.

Окрім основного тестування в стандартних умовах, експеримент також проводиться в ізольованому середовищі з обмеженими ресурсами (докеризована база даних та бекенд), щоб оцінити вплив параметрів прискорення у різних сценаріях. Це дозволяє проаналізувати контекстуальну залежність прискорення і їх ефективність за умов дефіциту ресурсів.

2.2.4 Висновки щодо наукового методу дослідження

Науковий метод дослідження дозволяє систематично підходити до вивчення впливу параметрів прискорення на продуктивність RESTful API. Він забезпечує не лише об’єктивність та повторюваність результатів, але й можливість отримання якісних даних, які дозволяють зробити достовірні висновки щодо ефективності різних підходів до прискорення.

Додаткове тестування в ізольованому середовищі з обмеженими ресурсами розширює аналіз та допомагає визначити, за яких умов параметри прискорення можуть бути менш ефективними або навіть контрпродуктивними. Це підкреслює важливість врахування контексту при впровадженні певних способів прискорення у реальних системах.

2.3 Експериментальний метод

2.3.1 Підготовка середовища та умов для експерименту

Для проведення експерименту було підготовлено середовище, що забезпечує точність і повторюваність результатів.

База даних MySQL розміщена у Docker-контейнері для ізоляції від інших процесів системи та контролю над ресурсами. Docker дозволяє створити стабільне середовище з можливістю обмеження ресурсів, таких як оперативна пам’ять та процесорний час. Конфігурація контейнера зображена на рисунку нижче (див. рис. 2.1).

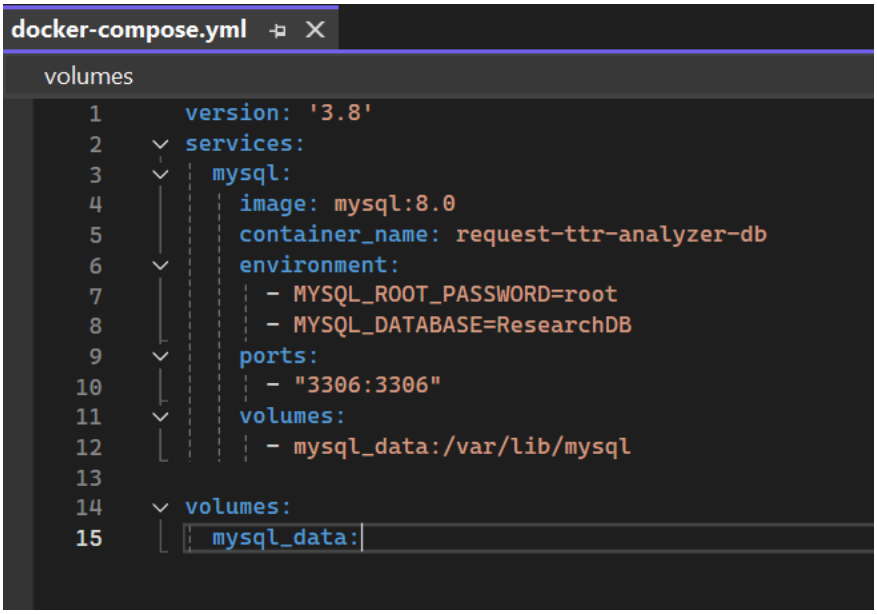


Рисунок 2.1 – Docker-композитний файл, використаний для налаштування бази даних

Контейнеризація бази даних дозволяє ізолювати та керувати ресурсами на рівні контейнерів, що забезпечує стабільні умови виконання тестів, незалежно від зовнішніх впливів. Ця можливість буде використана пізніше при додатково експерименті.

Додаток реалізовано на базі ASP.NET Core із використанням Onion-архітектури для забезпечення чіткого розподілу відповідальності між рівнями програми. База даних MySQL використовується для зберігання даних про продуктивність, що надає змогу зберігати великі обсяги інформації та проводити подальший аналіз.

У якості тестового середовища буде використано консольний додаток для автоматизованої відправки запитів до API, що забезпечує повторюваність результатів.

2.3.2 Планування та проведення експерименту

Виділено 5 основних CRUD-методів для кожного контролера, які реалізуються однаково та відрізняються лише кількістю вкладеностей та обсягом респонсу для сутностей «Products» та «Orders».

Ці методи включають:

– створення записів (Create);

– отримання всіх записів (Get All) з можливістю пагінації;

– отримання конкретного запису (Get by ID);

– редагування записів (Update);

– видалення записів (Delete).

Було визначено 8 основних комбінацій параметрів прискорення, які включають кешування, стиснення даних та пагінацію (див. рис. 2.2.).

```
16      // 8 possible combinations of optimization flag
17      public static readonly List<GlobalConfigurationFlags?> CombinationsOfParameters = new ()
18      {
19          null,                                                                   // params are not set
20          GlobalConfigurationFlags.UseCache,                                      // only cashing
21          GlobalConfigurationFlags.EnablePagination,                              // only pagination
22          GlobalConfigurationFlags.CompressData,                                  // only compression
23          GlobalConfigurationFlags.UseCache | GlobalConfigurationFlags.CompressData,          // caching and compression
24          GlobalConfigurationFlags.UseCache | GlobalConfigurationFlags.EnablePagination,      // caching and pagination
25          GlobalConfigurationFlags.CompressData | GlobalConfigurationFlags.EnablePagination,  // compressing and pagination
26          GlobalConfigurationFlags.EnablePagination | GlobalConfigurationFlags.CompressData | GlobalConfigurationFlags.UseCache // all parameters
27      };
```

Рисунок 2.2 – Визначені комбінації параметрів

Для кожної CRUD операції було заплановано відправку 30 запитів з кожною комбінацією параметрів. Загальна кількість запитів визначається наступним чином: 8 комбінацій * 10 ендпоінтів * 30 запитів на кожний ендпоінт. Загалом виходить 2400 запитів до сервера та відповідна кількість записів з деталями кожного запиту у БД.

Важно підкреслити, що запити були розподілені рівномірно між методами для обох контролерів. Це забезпечує об’єктивність результатів і знижує вплив зовнішніх факторів.

2.3.3 Збір та фіксація метрик

Метрики збираються за допомогою middleware, яке перехоплює кожен запит і фіксує такі дані:

– час виконання запиту. Фіксується початок та кінець обробки запиту за допомогою таймера. Час виконання записується в базу даних;

– розмір респонсу. Обчислюється обсяг даних, переданих клієнту;

– кількість оброблених записів. Реєструється кількість записів, повернених у відповіді;

– складність запиту. Визначається на основі глибини вкладеності об’єктів у відповіді (використовується кастомний алгоритм обчислення складності);

– HTTP метод. Для подальшого групування даних для аналізу.

Кожен запит отримує унікальний GUID та фіксується стан флагу з активними способами прискорення, що забезпечує точність ідентифікації та можливість зв’язування метрик з відповідним запитом.

2.3.4 Висновки щодо експериментального методу дослідження

Експериментальний метод забезпечує чітку процедуру проведення дослідження з використанням автоматизованого відправлення запитів та збору метрик. Це дозволяє систематично оцінити вплив різних способів прискорення на продуктивність RESTful API та забезпечити достовірність отриманих результатів.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір технологій та архітектурний підхід

3.1.1 Використання ASP.NET Core, Entity Framework Core, MySQL

Для розробки програмного забезпечення, призначеного для збору та аналізу метрик продуктивності API, було обрано технології, які забезпечують гнучкість, продуктивність та простоту розширення.

ASP.NET Core обрано як основний веб-фреймворк для розробки RESTful API завдяки його легкій структурі, швидкій обробці запитів та можливості гнучко налаштовувати середовище виконання. Це дозволяє ефективно обробляти HTTP-запити та легко впроваджувати middleware для збору метрик продуктивності.

Для взаємодії з базою даних було обрано ORM (Object-Relational Mapping) фреймворк Entity Framework Core [5]. Він дозволяє легко працювати з реляційними даними, автоматизуючи процеси створення, читання, оновлення та видалення (CRUD) записів у базі даних. Вибір EF Core обумовлений його підтримкою різних баз даних, зручним API для маніпуляції даними та можливістю швидкої інтеграції з ASP.NET Core.

Як основну реляційну базу даних обрано MySQL через її високу продуктивність, гнучкість і поширеність у веб-розробці. Використання MySQL у поєднанні з Entity Framework Core дозволяє легко зберігати дані про продуктивність запитів і забезпечувати їх швидке зчитування для подальшого аналізу.

У розробці програмного забезпечення було застосовано Onion-архітектуру, яка забезпечує чітке розділення логіки на шари (Domain, Application, Infrastructure, Presentation) та дотримується принципу залежностей, що сприяє легкій підтримці й розширенню функціональності програми.

3.1.2 Опис Onion-архітектури та її впровадження в проекті

Onion-архітектура — це підхід до проєктування програмного забезпечення, який дозволяє розділити логіку додатку на декілька шарів із чіткими зонами відповідальності [4]. Основні принципи Onion-архітектури, застосовані у проєкті:

– domain Layer (доменний шар): Основний шар, що містить доменні моделі, інтерфейси та логіку, яка є незалежною від зовнішніх технологій. У контексті проєкту це класи для продуктів та замовлень, а також інтерфейси сервісів, які взаємодіють з ними;

– application Layer (шар застосунків): Цей шар відповідає за бізнес-логіку додатку, тобто за обробку запитів до бази даних і взаємодію з іншими компонентами системи. У проекті це реалізовано за допомогою сервісів, таких як ProductService, які відповідають за CRUD-операції над продуктами та іншими доменними об'єктами;

– infrastructure Layer (інфраструктурний шар): У цьому шарі реалізовано взаємодію з зовнішніми технологіями — базою даних, middleware, кешами тощо. Саме тут підключено Entity Framework Core для роботи з MySQL та налаштовано логіку збереження метрик про продуктивність запитів;

– presentation Layer (шар презентації): Це шар для роботи з HTTP-запитами, тобто контролери та кінцеві точки API. Контролери приймають запити, передають їх для обробки в сервіси (Application Layer) та повертають відповіді клієнту. Також тут розміщуються middleware для перехоплення та логування запитів.

Onion-архітектура дозволяє забезпечити незалежність від зовнішніх сервісів і гнучкість під час розробки. Наприклад, заміна бази даних або підключення нового типу кешування вимагає мінімальних змін у коді.

Розроблюване рішення структуровано відповідно до цих принципів (див. рис 3.1).

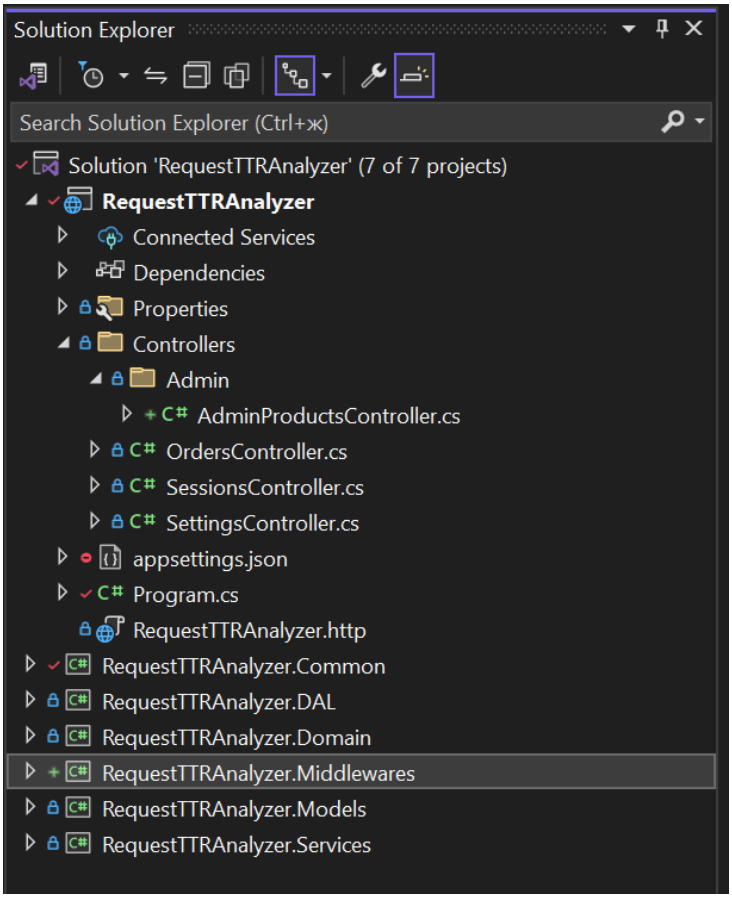


Рисунок 3.1 – Структура проєкту в середовищі розробки, що відображає застосування Onion-архітектури

На скріншоті видно, що рішення розділено на окремі проєкти/шари: Common, DAL, Domain, Middlewares, Models, Services, що чітко відповідає принципам Onion-архітектури. Головний проєкт містить контролери та базову конфігурацію ASP.NET Core, включно з appsettings.json та контролери для різних частин функціоналу (OrdersController, SessionsController, SettingsController та AdminProductsController).

Таке розділення на шари також спрощує тестування та підтримку коду, дозволяючи легко додавати нові оптимізації чи проводити рефакторинг програми.

3.2 Реалізація збору даних про продуктивність

У даному розділі буде розглянуто реалізацію збору метрик про продуктивність запитів під час виконання запитів до RESTful API. Для досягнення цієї мети було розроблено middleware-компонент RequestTimingMiddleware для збору метрик часу виконання та інших параметрів запитів.

3.2.1 Опис ролі middleware в програмі

Middleware у ASP.NET Core — це проміжні компоненти, які перехоплюють, обробляють або модифікують вхідні запити та вихідні відповіді. У цьому проекті middleware відповідають за обробку винятків та забезпечення єдиної структури помилок у відповідях API, та вимірювання часу виконання запитів, а також збір та збереження метрик продуктивності в базі даних для подальшого аналізу [3].

Реалізовані middleware забезпечують логування та обробку всіх виняткових ситуацій у запитах, а також зберігають час виконання, розмір відповідей, кількість записів та інші параметри у вигляді логів.

3.2.2 Реалізація middleware для вимірювання часу виконання запитів

RequestTimingMiddleware (для збору метрик продуктивності) - відповідає за збір часу виконання кожного запиту, а також метрик, пов'язаних із продуктивністю, таких як розмір відповіді, кількість записів у відповіді та складність запиту (див. лістинг 3.1).

Лістинг 3.1 – Код RequestTimingMiddleware -у

```
public class RequestTimingMiddleware
{
    private readonly RequestDelegate _next;
```

```
        public RequestTimingMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            var unitOfWork =
context.RequestServices.GetRequiredService<IUnitOfWork>();

            // Start the stopwatch to measure execution time
            var stopwatch = Stopwatch.StartNew();

            var requestLog = new RequestLog
            {
                Endpoint = context.Request.Path,
                ExecutedAt = DateTime.UtcNow
            };

            var originalBodyStream = context.Response.Body;

            try
            {
                using (var memoryStream = new MemoryStream())
                {
                    // Set a new `MemoryStream` to capture the
response
                    context.Response.Body = memoryStream;

                    await _next(context);

                    // Read the captured response
                    memoryStream.Seek(0, SeekOrigin.Begin);
```

```
                    string responseBody = await new
StreamReader(memoryStream).ReadToEndAsync();

                    // Counting metrics
                    CalculateMetrics(requestLog, responseBody);

                    // Set the response size in bytes
                    requestLog.ResponseSizeBytes =
memoryStream.Length;

                    // Move the stream pointer to the beginning and
copy the contents to the source `Response.Body`
                    memoryStream.Seek(0, SeekOrigin.Begin);
                    await
memoryStream.CopyToAsync(originalBodyStream);
                }

                requestLog.IsSuccess = context.Response.StatusCode <
400;
            }
            catch (Exception ex)
            {
                // If an exception occurs, mark request as
unsuccessful
                requestLog.IsSuccess = false;

                throw;
            }
            finally
            {
                stopwatch.Stop();

                // Gather information for logging
                requestLog.ExecutionTimeMs =
stopwatch.ElapsedMilliseconds;
```

```
                requestLog.ConfigFlags =
SettingsService.OptimizationConfig;

                // Save the log to the database
                unitOfWork.Repository<RequestLog>().Add(requestLog);
                unitOfWork.SaveChanges();

                // Always set `Response.Body` back to the original
stream
                context.Response.Body = originalBodyStream;
            }
        }

        // Приватні методи для розрахунку метрик (CalculateMetrics,
GetDepth тощо)
    }
```

Функціонал RequestTimingMiddleware:

– за допомогою Stopwatch вимірюється час виконання запиту від його отримання до відправки відповіді;

– визначається кількість записів у відповіді, розмір у байтах та складність запиту;

– зібрані дані зберігаються у вигляді об’єктів RequestLog у базі даних за допомогою IUnitOfWork.

Дані, зібрані за допомогою RequestTimingMiddleware, можуть бути використані для подальшого аналізу продуктивності, включаючи визначення, які запити є більш ресурсоємними та які комбінації параметрів прискорення можуть бути ефективними.

3.3 Реалізація сервісу продуктів (ProductsService)

3.3.1 Опис функціональності та методів CRUD

ProductsService — це сервіс, що відповідає за всі операції, пов'язані з продуктами в додатку, включно зі створенням, оновленням, видаленням і отриманням інформації про продукти (CRUD).

Цей сервіс забезпечує виконання таких основних завдань:

– створення продукту (CreateProduct): Метод приймає дані продукту у вигляді ProductRequestModel, мапить їх на доменну сутність Product та зберігає в базі даних. Після збереження даних створений продукт мапиться назад у ProductResponseModel для відповіді клієнту. Якщо глобальна конфігурація параметрів прискорення дозволяє, дані продукту можуть бути стиснуті;

– редагування продукту (EditProduct): Метод дозволяє оновлювати існуючий продукт за допомогою переданих даних. Після оновлення продукт також оновлюється в кеші для підтримки актуальності даних між кешем та базою при наступних запитах;

– видалення продукту (DeleteProduct): Метод видаляє продукт із бази даних за його ідентифікатором (ID) та очищує кеш;

– отримання продукту за ідентифікатором (GetProductById): Метод повертає дані про конкретний продукт. Якщо параметри прискорення дозволяють використання кешу, продукт буде взятий з кешу (якщо є);

– отримання списку продуктів (GetProducts): Метод дозволяє отримати список продуктів з бази даних, з можливістю використання пагінації та інших способів прискорення (наприклад, стискання даних).

3.3.2 Взаємодія з базою даних та використання патерну Repository

ProductsService активно використовує IUnitOfWork, який виступає в ролі централізованого інтерфейсу для доступу до репозиторіїв, що дозволяє організувати всі взаємодії з базою даних через єдиний об'єкт [6]. Це відповідає принципам Onion-архітектури, де інфраструктурний шар взаємодіє з базою даних, а бізнес-логіка сервісу залишається незалежною від специфіки бази даних.

Методи CreateProduct та EditProduct використовують IUnitOfWork для додавання або оновлення записів у таблиці Products.

Метод GetProductById перевіряє, чи дозволено використовувати кеш, і намагається отримати продукт з кешу перед тим, як виконати запит до бази даних. При відсутності продукту в кеші дані витягуються з бази та додаються в кеш для майбутніх запитів.

Метод GetProducts підтримує пагінацію для прискорення відправки великих обсягів даних та зменшення часу виконання запиту.

3.3.3 Глобальна конфігурація параметрів прискорення

ProductsService використовує об'єкт OptimizationConfig, який зберігає глобальні налаштування параметрів прискорення для запитів. Конфігурація включає такі флаги:

– UseCache – дозволяє або забороняє використання кешу для отримання продуктів;

– CompressData – вказує, чи слід стискати дані про продукти перед поверненням їх клієнту (якщо текстовий опис досить довгий);

– EnablePagination – дозволяє використання пагінації для прискорення відправки великих обсягів даних.

Ці параметри конфігурації контролюються централізовано через глобальні флаги та можуть бути змінені за допомогою запиту до API Settings контролеру. ProductsService динамічно перевіряє значення цих флагів перед кожною операцією, забезпечуючи застосування прискорення, коли увімкнені параметри прискорення (див. лістинг 3.2).

Лістинг 3.2 – Приклад коду взаємодії з кешем

```
private Product GetProductFromCacheById(int id)
{
    string cacheKey = $"Product_{id}";

    // Try to get the product from cache
    if (_cache.TryGetValue(cacheKey, out Product cachedProduct))
        return cachedProduct;

    // If not found in cache, get the product from database
    var product = _unitOfWork.Repository<Product>().Select(x =>
x.Id == id)
        .Include(x => x.Brand.Logo)
        .FirstOrDefault();

    if (product != null)
    {
        var cacheOptions = new MemoryCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow =
TimeSpan.FromMinutes(10),
            SlidingExpiration = TimeSpan.FromMinutes(5)
        };
```

```
            // Store the product in cache
            _cache.Set(cacheKey, product, cacheOptions);
        }

        return product;
    }
```

При зверненні до сервісу перевіряється наявність продукту в кеші за допомогою IMemoryCache, якщо продукт знайдено в кеші, він повертається, а якщо ні — здійснюється запит до бази даних, після чого дані додаються в кеш для подальшого використання.

Кешування контролюється конфігурацією MemoryCacheEntryOptions, яка дозволяє налаштувати абсолютний час життя об'єкта в кеші та ковзне (sliding) оновлення терміну дії при зверненнях.

Якщо опис продукту досить великий, сервіс може стискати ці дані за допомогою алгоритму GZip перед відправкою їх клієнту. Це зменшує розмір переданої інформації та оптимізує використання мережі (див. лістинг 3.3).

Лістинг 3.3 – Приклад коду стискання даних

```
    private void CompressProductData(ProductResponseModel product)
    {
        if       (string.IsNullOrEmpty(product.Description)       ||
product.Description.Length                                          <
_optimizationConfig.MinLengthForCompression)
            return;

        product.Description = CompressString(product.Description);
    }

    private string CompressString(string data)
    {
        byte[] dataBytes = Encoding.UTF8.GetBytes(data);

        using (var outputStream = new MemoryStream())
```

```
        {
            using (var gzipStream = new GZipStream(outputStream, 
CompressionLevel.Optimal))
            {
                gzipStream.Write(dataBytes, 0, dataBytes.Length);
            }

            return Convert.ToBase64String(outputStream.ToArray());
        }
    }
```

Стискання використовується лише тоді, коли розмір тексту перевищує мінімальну довжину, зазначену у глобальній конфігурації. Для експерименту було встановлено довжину в 300 символів.

ProductsService є ключовим елементом програми для обробки CRUD-операцій з продуктами та впровадження підходів прискорення, таких як кешування, пагінація та стискання даних. Гнучке використання глобальних флагів прискорення дозволяє динамічно змінювати поведінку сервісу, забезпечуючи підвищення продуктивності запитів RESTful API.

Повний текст програми представлений в документі Додаток Б «Текст програми».

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ПОЧАТКОВІ РЕЗУЛЬТАТИ

Для демонстрації дієздатності програми буде використано декілька запитів до ProductController-а: на створення товару та на отримання інформації про товари. Це допоможе охопити всі досліджувані параметри прискорення та побачити час виконання запиту при наявності прискорення та без нього.

Для тестування буде використано Swagger, бо зазвичай у проектах на ASP.NET Core Swagger інтегрується за допомогою бібліотеки Swashbuckle, яка генерує OpenAPI-документацію та створює інтерактивну сторінку, де можна побачити всі доступні методи API та спробувати їх виконати.

Swagger — це інструмент, який дозволяє автоматично створювати документацію для RESTful API та переглядати ендпоінти та виконувати запити.

4.1 Тестування роботи ProductService та middleware

Для взаємодії з більшістю ендпоінтів за допомогою інтерфейсу Swagger необхідно відкрити сторінку Swagger та отримати access token для виконання запитів (див. рис. 4.1). Отримати його можна у апі, що пов’язана з входом до облікового запису: PUT api/sessions/ (див. рис. 4.2 та рис. 4.3).

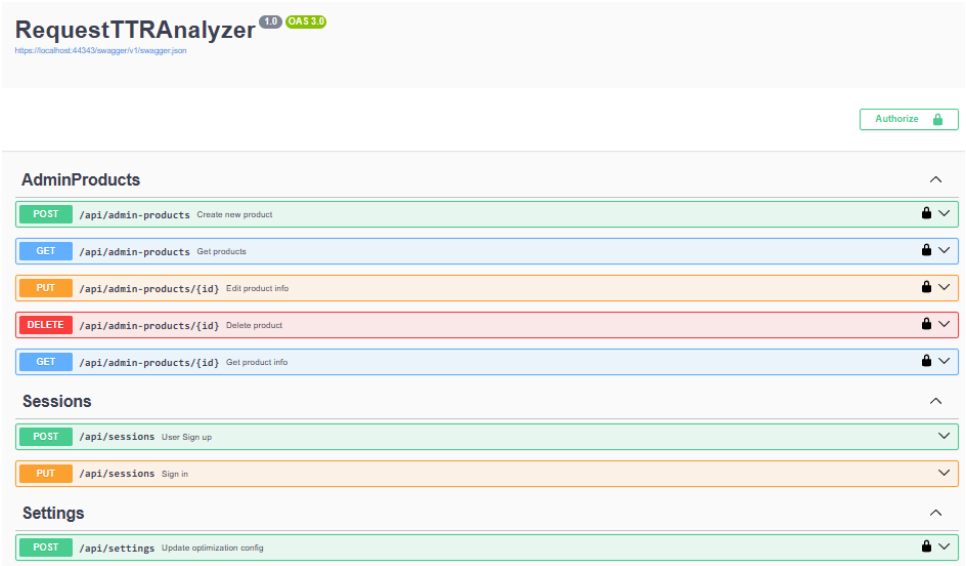


Рисунок 4.1 – Інтерфейс Swagger з доступними ендпоінтами

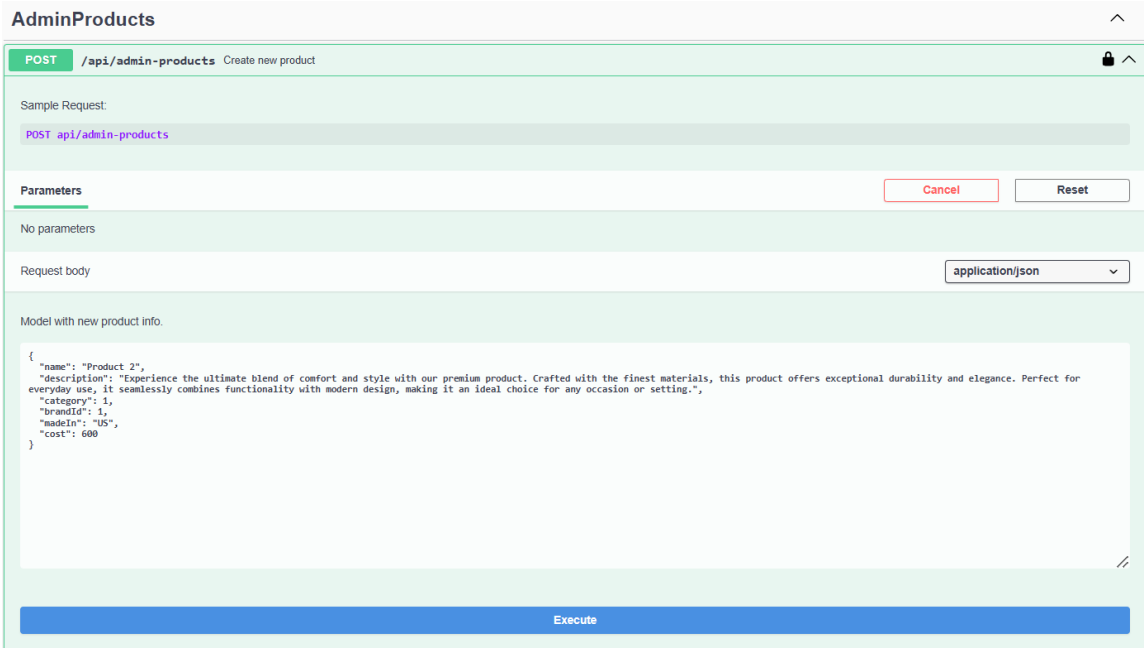


Рисунок 4.2 – Приклад запиту на створення товару

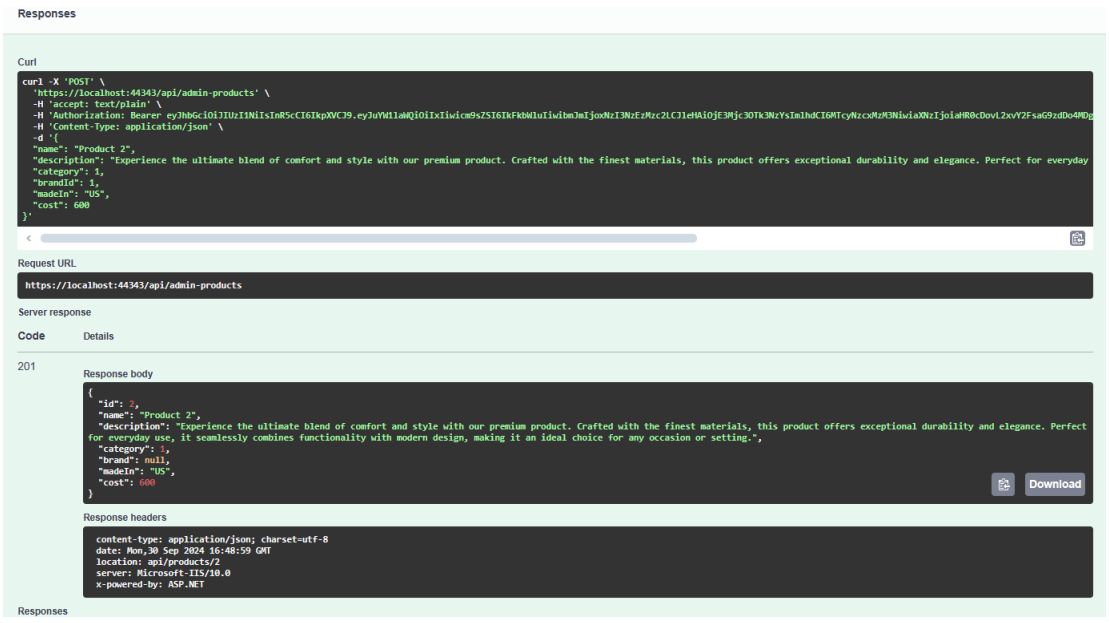


Рисунок 4.3 – Response сервера на створеня сутності товару

Створена сутність товару вноситься до БД разом із інформацією про запит (див. рис. 4.4 та рис.4.5)

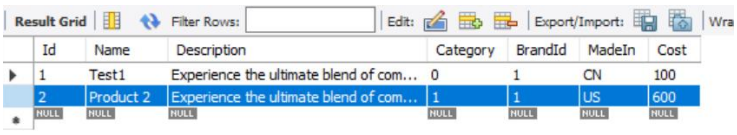


Рисунок 4.4 – Створений товар у БД

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Id	Endpoint	ExecutionTimeMs	ExecutedAt	RecordCount	FieldCount	ResponseSizeBytes	QueryComplexity	ConfigFlags	IsSuccess
15	/api/admin-products	90	2024-09-30 16:48:59.436352	1	7	401	0	NULL	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 4.5 – Запис з інформацією про виконаний запит

Далі виконаємо запит на отримання списку товарів (див. рис. 4.6 та рис. 4.7). Інформація про новий новий запит записується у БД для подальшої обробки (див. рис. 4.8).

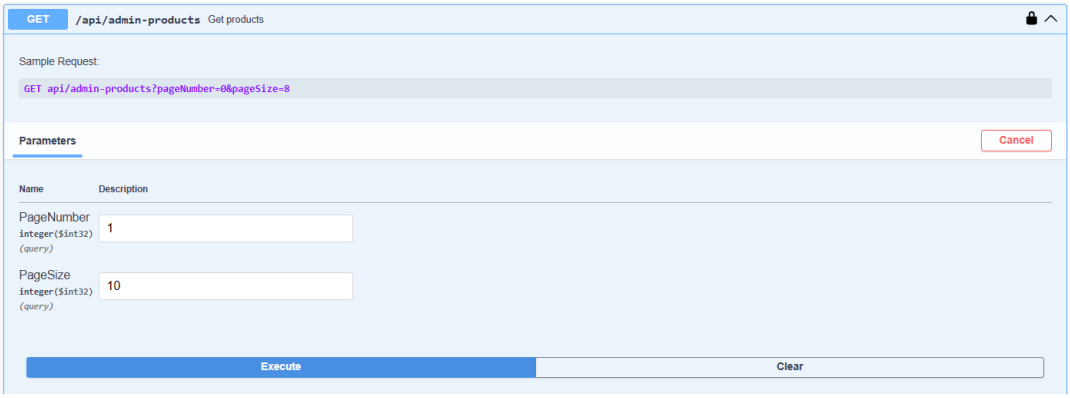


Рисунок 4.6 – Запит на отримання товарів (без прискорення, тому пагінація відключена)

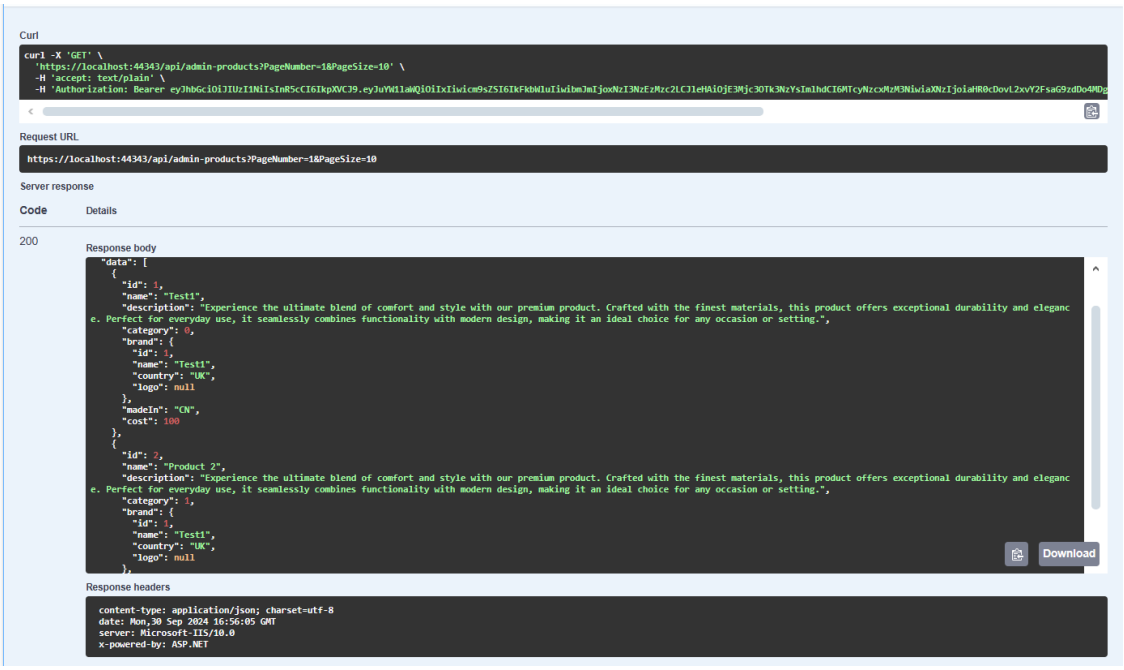


Рисунок 4.7 – Результат виконання запиту на отримання товарів(без параметрів прискорення)

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Id	Endpoint	ExecutionTimeMs	ExecutedAt	RecordCount	FieldCount	ResponseSizeBytes	QueryComplexity	ConfigFlags	IsSuccess
15	/api/admin-products	90	2024-09-30 16:48:59.436352	1	7	401	0	NULL	1
16	/api/admin-products	66	2024-09-30 16:56:05.922459	2	7	937	2	NULL	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 4.8 – Запис з інформацією про виконаний запит

Аналогічним чином протестуємо api на отримання інформації про товар за його id (див. рис. 4.9 та рис. 4.10). Інформація про деталі запиту вноситься до БД (див. рис. 4.11).

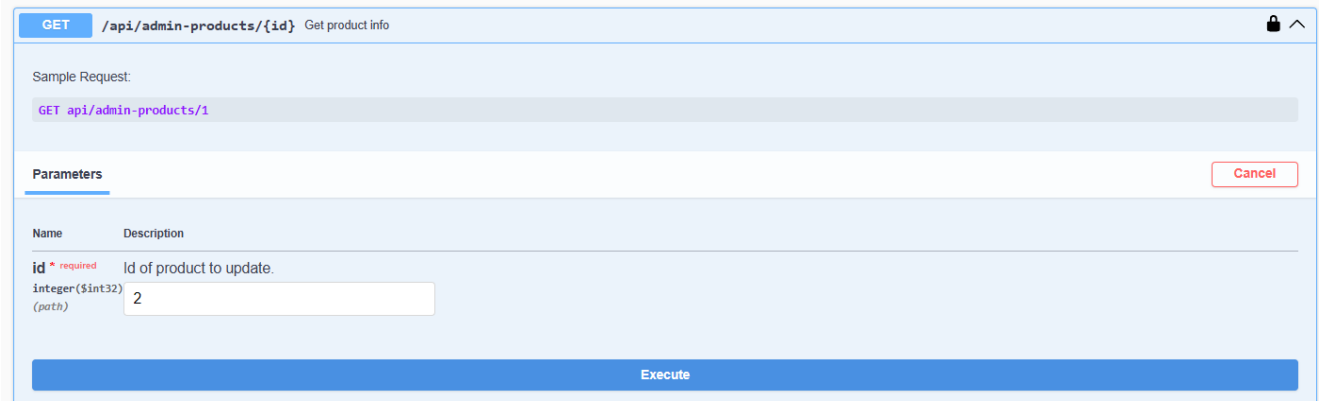


Рисунок 4.9 – Запит на отримання інформації про товар

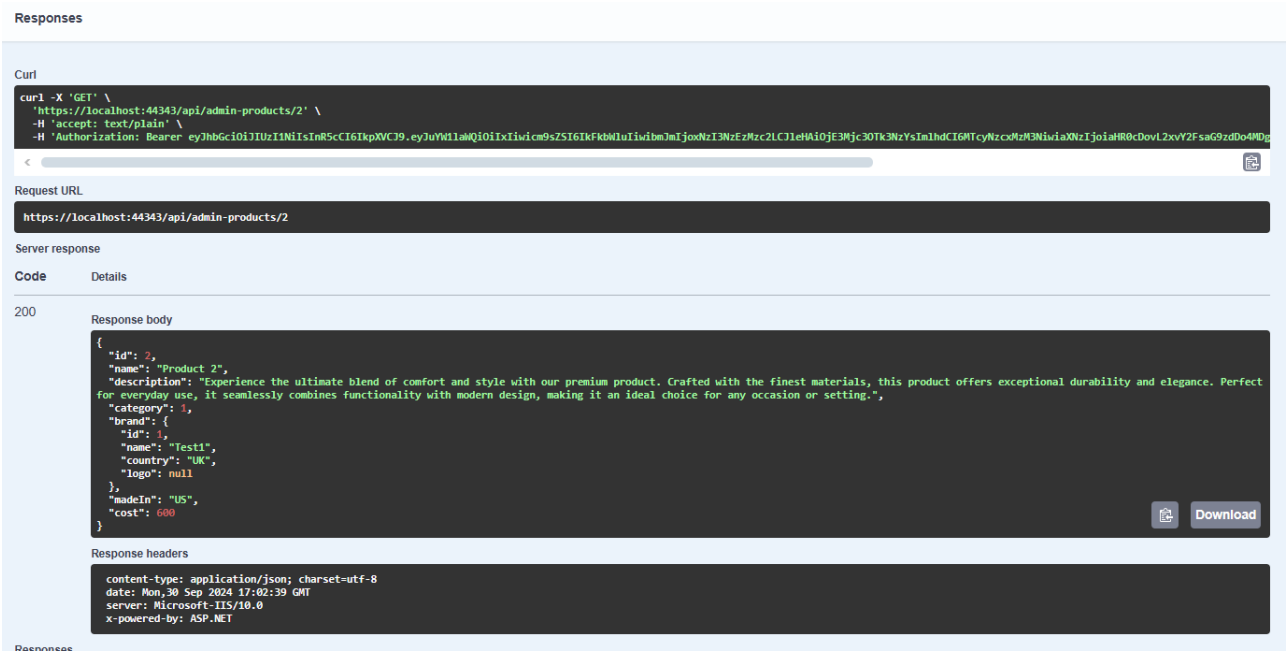


Рисунок 4.10 – Відповідь сервера

Id	Endpoint	ExecutionTimeMs	ExecutedAt	RecordCount	FieldCount	ResponseSizeBytes	QueryComplexity	ConfigFlags	IsSuccess
15	/api/admin-products	90	2024-09-30 16:48:59.436352	1	7	401	0	NULL	1
16	/api/admin-products	66	2024-09-30 16:56:05.922459	2	7	937	2	NULL	1
17	/api/admin-products/2	49	2024-09-30 17:02:39.762561	1	7	447	2	NULL	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 4.11 – Запис з інформацією про виконаний запит

Далі необхідно протестувати зміну статусу флагів налаштувань які відповідають за прискорення (див. рис. 4.12 та рис. 4.13).

Інформація про цей запит також внесеться до БД але вона по суті надлишкова, оскільки параметри прискорення не впливають на конфігураційні api.

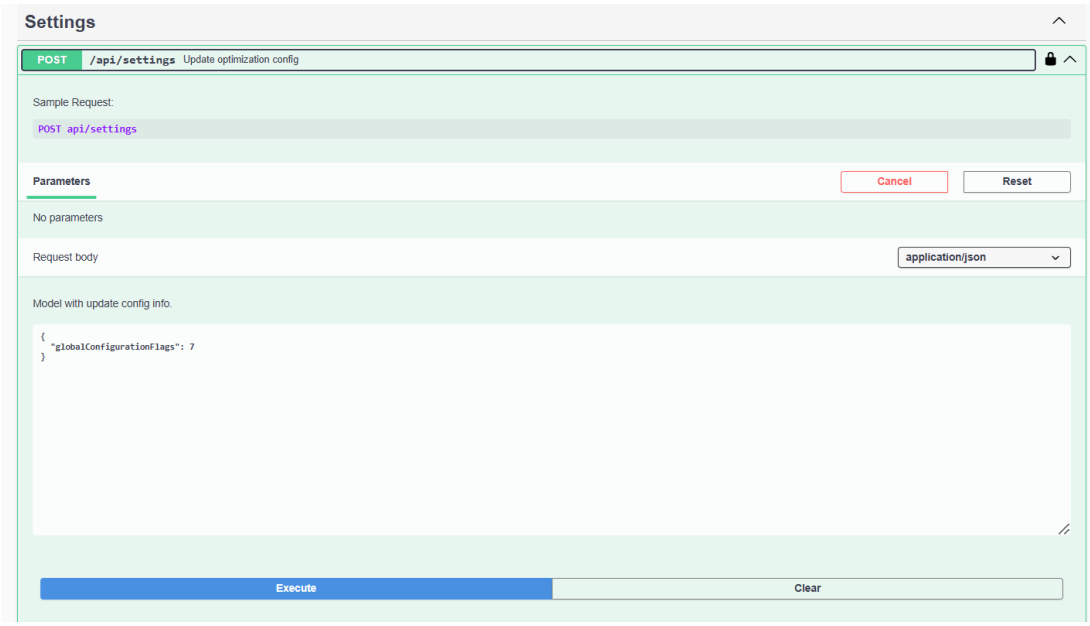


Рисунок 4.12 – Запит на ввімкнення флагів прискорення

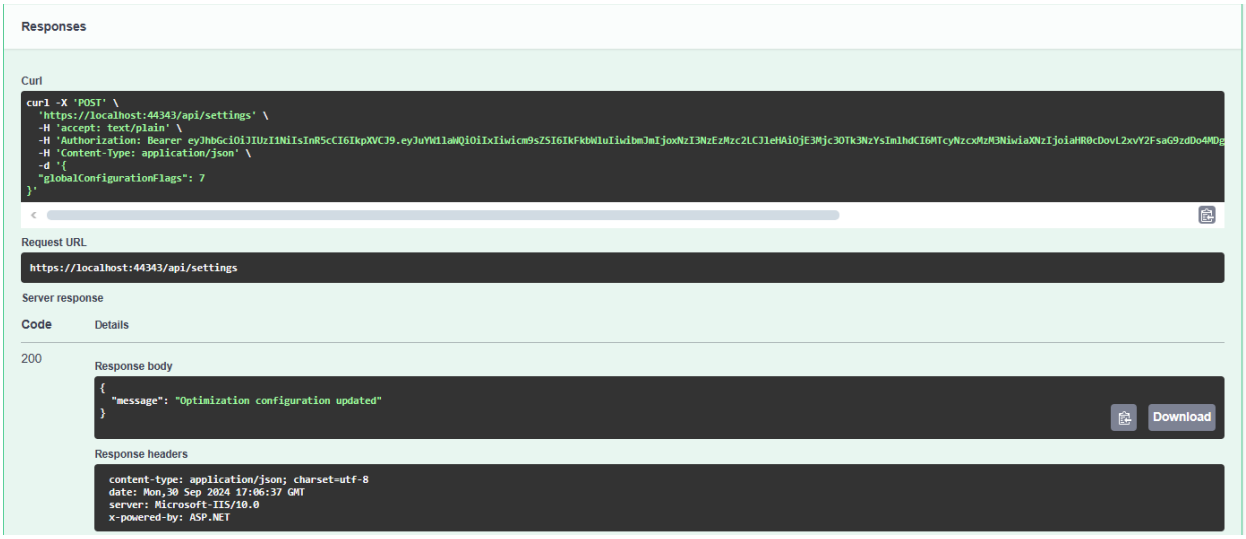


Рисунок 4.13 – Результат ввімкнення флагів

Для порівняння виконаємо два запити: на отримання списку товарів з пагінацією (див. рис. 4.14 та рис. 4.15) та отримання товару за id (див. рис. 4.16 та рис. 4.17). Інформацію стосовно час виконання та деталі запитів було записано до БД (див. рис. 4.18).

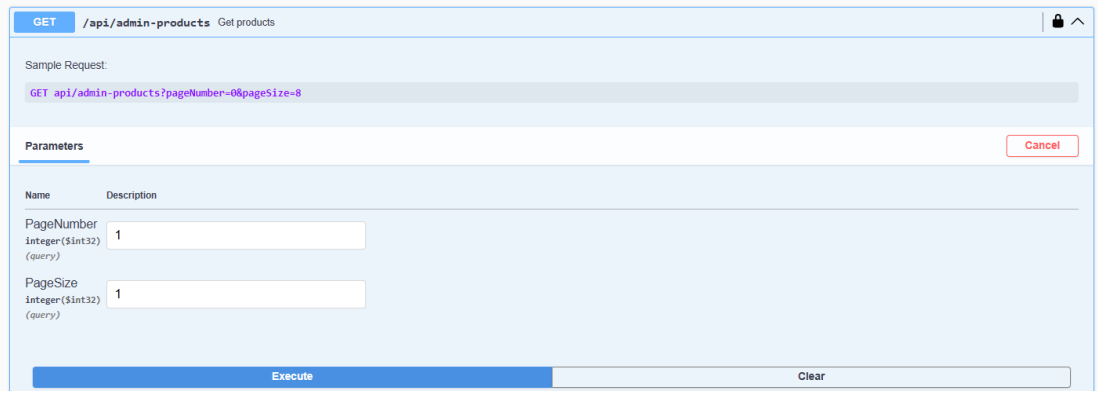


Рисунок 4.14 - Запит на отримання продуктів з пагінацією

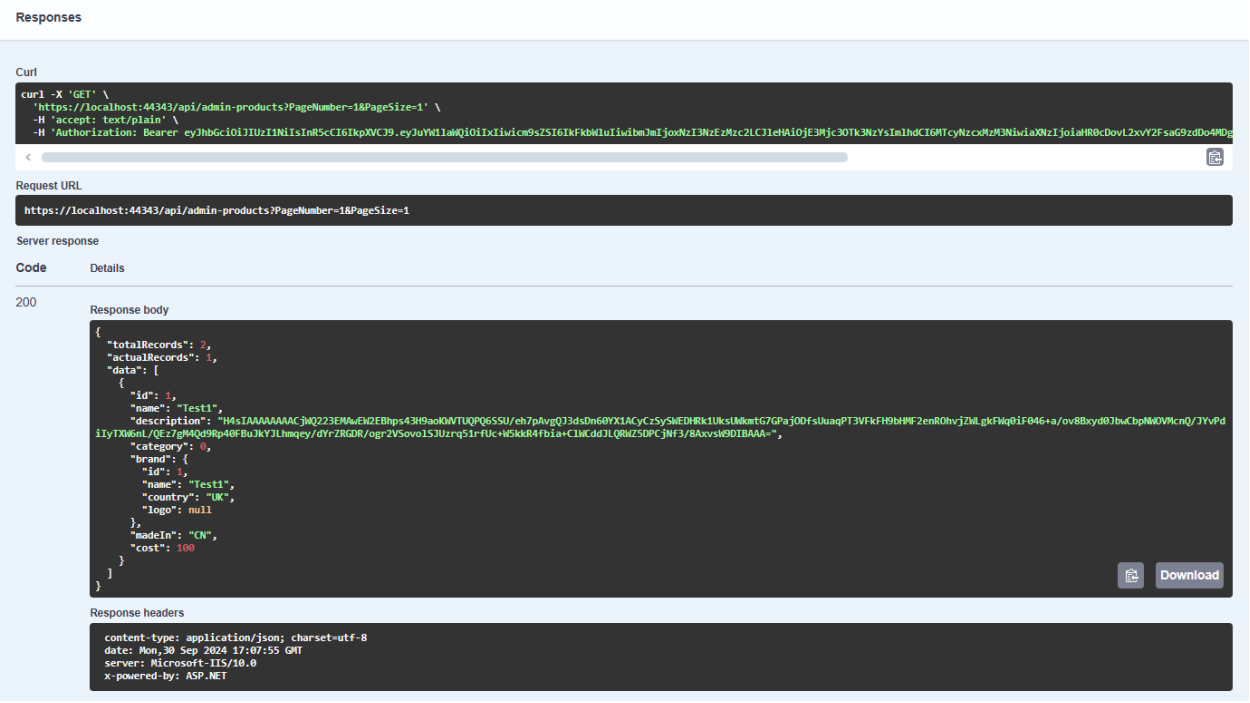


Рисунок 4.15 – Результат виконання

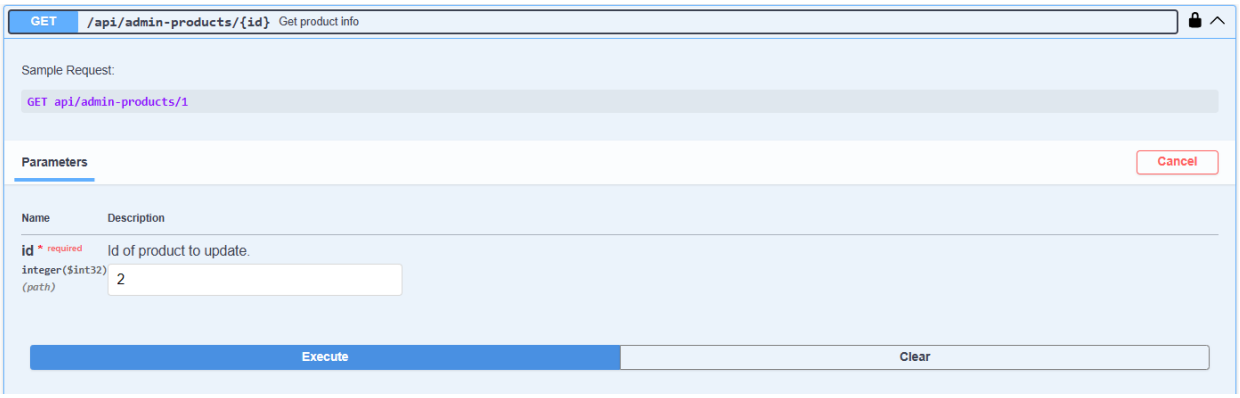


Рисунок 4.16 – Запит на отримання інформації про продукт (з кешуванням та стисканням)

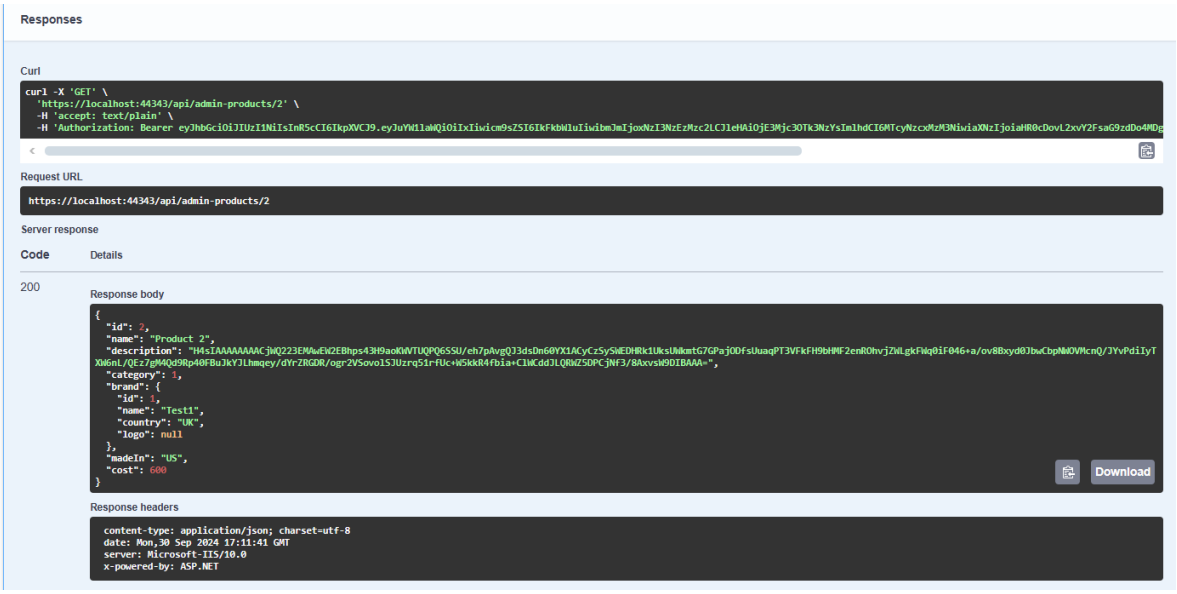


Рисунок 4.17 – Результат виконання

Id	Endpoint	ExecutionTimeMs	ExecutedAt	RecordCount	FieldCount	ResponseSizeBytes	QueryComplexity	ConfigFlags	IsSuccess
15	/api/admin-products	90	2024-09-30 16:48:59.436352	1	7	401	0	NULL	1
16	/api/admin-products	66	2024-09-30 16:56:05.922459	2	7	937	2	NULL	1
17	/api/admin-products/2	49	2024-09-30 17:02:39.762561	1	7	447	2	NULL	1
18	/api/settings	13	2024-09-30 17:06:37.345357	1	1	48	0	7	1
19	/api/admin-products	60	2024-09-30 17:07:55.935014	1	7	471	2	7	1
20	/api/admin-products/2	11	2024-09-30 17:11:41.922332	1	7	429	2	7	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 4.18 – Інформація про виконані запити у БД (сині з використанням параметрів прискорення)

На рисунку 4.18 видно, що виконання запитів на отримання товарів без ввімкнених параметрів прискорення (id 16 та 17) зайняло 66 та 49 мс відповідно, також важливо звернути увагу на розмір відповіді, що сягає 937 байт та 447 байт.

Записи з інформацією про ті ж самі запити але з ввімкнутими параметрами прискорення виділено на рисунку (id 19 та 20). Важливо відмітити, що час виконання запиту суттєво змінився, особливо для апі на отримання товару по id, а розмір відповіді зменшився вдвічі завдяки пагінації. Подальші розрахунки та порівняння ефективності будуть проводитися на дослідницькій практиці.

4.2 Аналіз коректності збору метрик

У цьому розділі розглянемо, як було реалізовано middleware для збору метрик про виконання запитів у системі. Middleware перехоплює кожен HTTP-запит, вимірює час виконання, збирає інформацію про відповідь, обчислює складність запиту та зберігає результати в базу даних для подальшого аналізу.

RequestTimingMiddleware починає роботу з кожним HTTP-запитом, що надходить до сервера. Одразу після отримання запиту запускається Stopwatch, щоб почати вимірювати час виконання. Крім того, створюється об'єкт RequestLog, який збирає основну інформацію про запит, таку як шлях (Endpoint) та час отримання (ExecutedAt).

Middleware використовує MemoryStream для перехоплення відповіді, що надсилається клієнту. Це дозволяє аналізувати дані відповіді перед тим, як передати їх клієнту. Після завершення обробки запиту MemoryStream зчитує дані відповіді, обчислює розмір у байтах (ResponseSizeBytes) та виконує розрахунок метрик через метод CalculateMetrics.

Для кожного запиту збираються такі метрики:

– час виконання запиту (ExecutionTimeMs): Загальний час обробки запиту в мілісекундах. Цей час обраховується за допомогою Stopwatch і вимірюється від початку запиту до моменту формування відповіді;

– кількість записів у відповіді (RecordCount): Підрахунок кількості об'єктів, які містяться у відповіді, здійснюється через парсинг JSON-відповіді. Якщо це пагінована відповідь, підраховується кількість елементів у полі Data;

– кількість полів у кожному записі (FieldCount): Обчислюється кількість ключів (полів) у основній сутності об'єкта відповіді, щоб оцінити ширину даних;

– складність запиту (QueryComplexity): Визначається на основі вкладеності об'єктів у відповіді. Якщо глибина вкладеності перевищує певні пороги, запит може бути позначений як Simple, Moderate або Complex.

Якщо під час обробки запиту виникає виняток (Exception), middleware відмічає цей запит як невдалий (IsSuccess = false) і передає помилку для подальшої обробки. Після цього Stopwatch зупиняється, а всі зібрані метрики записуються в базу даних через IUnitOfWork.

У процесі тестування було перевірено:

– повнота метрик: Всі необхідні поля (Endpoint, ExecutionTimeMs, ResponseSizeBytes, RecordCount, FieldCount, QueryComplexity) правильно записуються в таблицю RequestLog в базі даних після кожного запиту;

– коректність часу виконання: Час виконання запиту відповідає реальному часу обробки запиту в системі, що підтверджує коректну роботу Stopwatch;

– відповідність метрик прискорення: Метрики зберігаються відповідно до глобальних флагів прискорення (кешування, стискання, пагінація).

Наприклад, час виконання запиту значно скорочується при увімкненому кешуванні, що можна побачити на рисунку 3.18.

Під час тестування middleware виявлено, що:

– він стабільно працює з декількома типами запитів, незалежно від їх складності;

– метрики надійно збираються навіть у випадках, коли запит завершується винятком;

– операції з перехопленням та збереженням метрик не створюють додаткового значного навантаження на систему.

Таке тестування підтвердило коректність збору метрик та готовність до проведення подальших експериментів із продуктивністю запитів.

5 ОБРОБКА ТА АНАЛІЗ ЗІБРАНИХ ДАНИХ

5.1 Підготовка даних до аналізу

Після збору даних у рамках експериментального дослідження дані слід агрегувати для подальшого аналізу. Агрегація передбачає структурування та групування метрик відповідно до основних характеристик та комбінацій параметрів прискорення, які були встановлені для кожного запиту. Це дозволить наочно порівняти результати й виявити ключові закономірності. Нижче зображено частину даних без агрегації (див. рис. 5.1).

Id	Endpoint	ExecutionTimeMs	ExecutedAt	RecordCount	FieldCount	ResponseSizeBytes	QueryComplexity	ConfigFlags	IsSuccess	RequestMethod
3	/api/products	69	2024-10-28 00:32:16.543609	1	7	461	0	NULL	1	POST
4	/api/products	11	2024-10-28 00:32:16.617906	1	7	297	0	NULL	1	POST
5	/api/products	10	2024-10-28 00:32:16.631750	1	7	533	0	NULL	1	POST
6	/api/products	10	2024-10-28 00:32:16.644752	1	7	361	0	NULL	1	POST
7	/api/products	7	2024-10-28 00:32:16.659686	1	7	432	0	NULL	1	POST
8	/api/products	9	2024-10-28 00:32:16.671395	1	7	361	0	NULL	1	POST
9	/api/products	8	2024-10-28 00:32:16.683126	1	7	137	0	NULL	1	POST
10	/api/products	8	2024-10-28 00:32:16.694062	1	7	111	0	NULL	1	POST
11	/api/products	8	2024-10-28 00:32:16.704245	1	7	413	0	NULL	1	POST
12	/api/products	8	2024-10-28 00:32:16.715089	1	7	298	0	NULL	1	POST
13	/api/products	8	2024-10-28 00:32:16.726681	1	7	112	0	NULL	1	POST
14	/api/products	9	2024-10-28 00:32:16.738601	1	7	319	0	NULL	1	POST
15	/api/products	8	2024-10-28 00:32:16.750035	1	7	112	0	NULL	1	POST
16	/api/products	8	2024-10-28 00:32:16.761354	1	7	162	0	NULL	1	POST
17	/api/products	9	2024-10-28 00:32:16.771793	1	7	534	0	NULL	1	POST
18	/api/products	16	2024-10-28 00:32:16.788068	1	7	158	0	NULL	1	POST
19	/api/products	9	2024-10-28 00:32:16.806934	1	7	294	0	NULL	1	POST
20	/api/products	9	2024-10-28 00:32:16.818217	1	7	162	0	NULL	1	POST
21	/api/products	7	2024-10-28 00:32:16.830545	1	7	168	0	NULL	1	POST
22	/api/products	8	2024-10-28 00:32:16.840341	1	7	161	0	NULL	1	POST
23	/api/products	13	2024-10-28 00:32:16.851126	1	7	319	0	NULL	1	POST

Рисунок 5.1 – Частина зібраних даних у чистому вигляді без агрегації

Для спрощення пронумеруємо кожну комбінацію способів прискорення:

– комбінація 1. Без прискорення;
– комбінація 2. Кешування;
– комбінація 3. Пагінація;
– комбінація 4. Компресія;
– комбінація 5. Кешування + компресія;
– комбінація 6. Кешування + пагінація;
– комбінація 7. Компресія + пагінація;
– комбінація 8. Кешування + компресія + пагінація.

5.1.1 Групування за комбінаціями способів прискорення

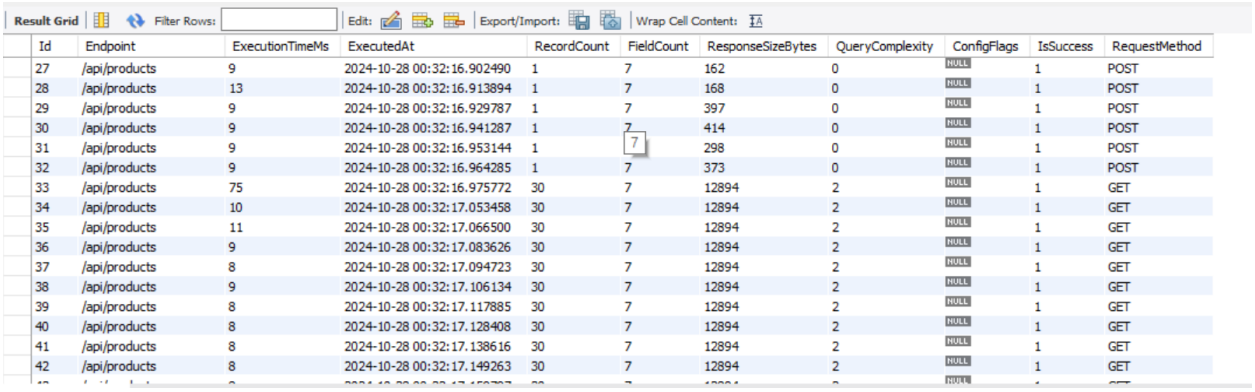
Дані будуть згруповані відповідно до різних комбінацій активних параметрів прискорення (кешування, стиснення даних, пагінація). Це дозволить оцінити ефективність кожної комбінації в контексті впливу на час виконання запитів та інші метрики.

Усього було визначено 8 основних комбінацій способів прискорення, що охоплюють усі можливі стани включення та виключення кешування, стиснення та пагінації (див. рис. 5.3). Для кожної комбінації буде створена окрема група, яка містить усі виконані запити з відповідними параметрами (див. рис. 5.2).

```
1  SELECT * FROM ResearchDB.RequestLogs
2  WHERE Endpoint != "/api/test" AND Endpoint != "/api/settings"
3  AND ConfigFlags IS NULL
4  ORDER BY Id
5  LIMIT 3000
```

Рисунок 5.2 – Приклад запиту на отримання метрик по першій комбінації

Запити на отримання інших комбінацій повністю аналогічні за виключенням значення перевірки поля ConfigFlags яке буде змінюватися залежно від комбінації.



Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Id	Endpoint	ExecutionTimeMs	ExecutedAt	RecordCount	FieldCount	ResponseSizeBytes	QueryComplexity	ConfigFlags	IsSuccess	RequestMethod
27	/api/products	9	2024-10-28 00:32:16.902490	1	7	162	0	NULL	1	POST
28	/api/products	13	2024-10-28 00:32:16.913894	1	7	168	0	NULL	1	POST
29	/api/products	9	2024-10-28 00:32:16.929787	1	7	397	0	NULL	1	POST
30	/api/products	9	2024-10-28 00:32:16.941287	1	7	414	0	NULL	1	POST
31	/api/products	9	2024-10-28 00:32:16.953144	1	7	298	0	NULL	1	POST
32	/api/products	9	2024-10-28 00:32:16.964285	1	7	373	0	NULL	1	POST
33	/api/products	75	2024-10-28 00:32:16.975772	30	7	12894	2	NULL	1	GET
34	/api/products	10	2024-10-28 00:32:17.053458	30	7	12894	2	NULL	1	GET
35	/api/products	11	2024-10-28 00:32:17.066500	30	7	12894	2	NULL	1	GET
36	/api/products	9	2024-10-28 00:32:17.083626	30	7	12894	2	NULL	1	GET
37	/api/products	8	2024-10-28 00:32:17.094723	30	7	12894	2	NULL	1	GET
38	/api/products	9	2024-10-28 00:32:17.106134	30	7	12894	2	NULL	1	GET
39	/api/products	8	2024-10-28 00:32:17.117885	30	7	12894	2	NULL	1	GET
40	/api/products	8	2024-10-28 00:32:17.128408	30	7	12894	2	NULL	1	GET
41	/api/products	8	2024-10-28 00:32:17.138616	30	7	12894	2	NULL	1	GET
42	/api/products	8	2024-10-28 00:32:17.149263	30	7	12894	2	NULL	1	GET

Рисунок 5.3 – Приклад отриманих даних для першої комбінації

Результати виборки по кожній окремій комбінації будуть збережені у окремі CSV файли для спрощення подальшого аналізу та побудови графіків.

5.1.2 Групування за методами API

Дані також будуть згруповані відповідно до API методів: створення (Create), отримання всіх записів (Get All), отримання запису за ID (Get by ID), редагування (Update) та видалення (Delete). Оскільки однакові методи реалізовані як для сутності «Products», так і для «Orders», дані можна об’єднати в групи за аналогічними методами, щоб виявити загальні закономірності незалежно від конкретного контролера.

Для кожного методу необхідно провести аналіз впливу комбінацій способів прискорення, що дозволить оцінити ефективність прискорення у контексті різних типів операцій. Для прикладу виконаємо групування за GET методов (див. рис. 5.4 та рис. 5.5)

```
1 ●  SELECT * FROM ResearchDB.RequestLogs
2    WHERE Endpoint != "/api/test" AND Endpoint != "/api/settings"
3    AND RequestMethod = "GET"
4    ORDER BY Id
5    LIMIT 3000
```

Рисунок 5.4 – Приклад запиту для отримання метрик незалежно від комбінації для GET запитів

Інші запити виглядають аналогічно лише зі зміною типу HTTP методу на POST, PUT та DELETE.

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

Id	Endpoint	ExecutionTimeMs	ExecutedAt	RecordCount	FieldCount	ResponseSizeBytes	QueryComplexity	ConfigFlags	IsSuccess	RequestMethod
60	/api/products	13	2024-10-28 00:32:17.389958	30	7	12894	2	NULL	1	GET
61	/api/products	13	2024-10-28 00:32:17.406009	30	7	12894	2	NULL	1	GET
62	/api/products	36	2024-10-28 00:32:17.422049	30	7	12894	2	NULL	1	GET
93	/api/products/17	13	2024-10-28 00:32:19.197359	1	7	564	2	NULL	1	GET
94	/api/products/20	5	2024-10-28 00:32:19.213132	1	7	448	2	NULL	1	GET
95	/api/products/14	5	2024-10-28 00:32:19.220697	1	7	668	2	NULL	1	GET
96	/api/products/12	5	2024-10-28 00:32:19.227745	1	7	470	2	NULL	1	GET
97	/api/products/27	4	2024-10-28 00:32:19.234724	1	7	448	2	NULL	1	GET
98	/api/products/8	5	2024-10-28 00:32:19.241240	1	7	425	2	NULL	1	GET
99	/api/products/16	4	2024-10-28 00:32:19.248252	1	7	360	2	NULL	1	GET
100	/api/products/26	4	2024-10-28 00:32:19.254717	1	7	554	2	NULL	1	GET
101	/api/products/7	4	2024-10-28 00:32:19.261382	1	7	422	2	NULL	1	GET
102	/api/products/3	5	2024-10-28 00:32:19.267855	1	7	704	2	NULL	1	GET
103	/api/products/11	5	2024-10-28 00:32:19.274642	1	7	616	2	NULL	1	GET
104	/api/products/28	5	2024-10-28 00:32:19.281740	1	7	616	2	NULL	1	GET
105	/api/products/22	5	2024-10-28 00:32:19.288922	1	7	439	2	NULL	1	GET

Рисунок 5.5 – Частина зібраних метрик за допомогою цього запиту

5.2 Обчислення основних статистичних показників

5.2.1 Середнє значення (Mean)

Середнє значення є ключовим статистичним показником, що дозволяє визначити загальну тенденцію у вибірці [7]. В контексті даного дослідження середнє значення часу виконання запиту та розміру респонсу надає загальну оцінку ефективності кожної комбінації способів прискорення.

– середній час виконання запиту (avg_execution_time). Цей показник обчислюється для кожної комбінації і дозволяє порівняти швидкість обробки запитів у різних умовах. Високе значення середнього часу може свідчити про те, що поточна комбінація параметрів менш ефективна, тоді як низьке значення вказує на кращу продуктивність. Середній час виконання є основним критерієм для визначення ефективності способів прискорення;

– середній розмір респонсу (avg_response_size). Цей показник показує, як різні комбінації способів прискорення впливають на обсяг переданих даних. Наприклад, активація параметру стиснення (compression) повинна зменшити середній розмір респонсу. Зменшення розміру відповіді сприяє швидшій передачі даних через мережу, що є особливо важливим для додатків з великим трафіком або обмеженою пропускною здатністю.

Дані збираються за допомогою SQL запиту (див. рис. 5.6). Результати обчислень середнього значення представлені у таблиці нижче для кожної комбінації параметрів, що дозволить легко порівняти ефективність різних конфігурацій (див. табл. 5.1).

```sql
SELECT
    ROUND(AVG(ExecutionTimeMs), 2) AS avg_execution_time,
    ROUND(AVG(ResponseSizeBytes), 2)AS avg_response_size,
    COUNT(*) AS request_count,
    ConfigFlags AS config_flags,
    RequestMethod AS request_type
FROM
    ResearchDB.RequestLogs
WHERE
    Endpoint != "/api/test" AND Endpoint != "/api/settings"
    AND ConfigFlags = {номер комбінації}
GROUP BY
    config_flags, request_type;
```

Рисунок 5.6 – SQL запит для розрахунку середнього значення для кожного параметра

Таблиця 5.1 – Середні значення за типом реквесту та комбінацією

Комбінація	**avg_execution_time(мс)**	**avg_response_size(байт)**	**request_count(шт)**	**request_type**
1	14.17	357.02	60	POST
1	14.13	16905.41	120	GET
1	33.22	370.73	60	PUT
1	10.32	50.00	60	DELETE
2	10.40	372.08	60	POST
2	8.79	16121.25	120	GET
2	10.95	371.13	60	PUT
2	9.93	50.00	60	DELETE
3	8.90	382.53	60	POST
3	5.83	5127.67	120	GET
3	10.50	371.17	60	PUT
3	9.17	50.00	60	DELETE
4	8.60	360.75	60	POST
4	7.48	14677.72	120	GET
4	9.82	366.77	60	PUT
4	8.95	50.00	60	DELETE
5	8.58	379.92	60	POST
5	6.11	16664.58	120	GET
5	10.08	367.17	60	PUT
5	8.82	50.00	60	DELETE
6	8.32	342.67	60	POST
6	3.59	3607.55	120	GET
6	18.32	372.12	60	PUT
6	29.07	50.00	60	DELETE
7	8.58	359.03	60	POST
7	5.26	4617.45	120	GET
7	9.23	367.17	60	PUT
7	8.68	50.00	60	DELETE

Продовження Таблиці 5.1 – Середні значення за типом реквесту та комбінацією

Комбінація	avg_execution_time(мс)	avg_response_size(байт)	request_count(шт)	request_type
8	8.38	363.98	60	POST
8	3.83	4180.96	120	GET
8	9.55	367.10	60	PUT
8	8.92	50.00	60	DELETE

5.2.2 Стандартне відхилення (Standard Deviation)

Стандартне відхилення показує ступінь розсіювання значень навколо середнього [7]. У нашому випадку, цей показник важливий для оцінки стабільності результатів:

– стандартне відхилення для часу виконання запиту. Якщо значення стандартного відхилення для часу виконання є низьким, це вказує на стабільну роботу системи в рамках певної комбінації способів прискорення. Високе стандартне відхилення може свідчити про нестабільну поведінку, що, в свою чергу, може бути наслідком навантаження або інших непередбачуваних факторів. Для додатків з високими вимогами до стабільності важливо, щоб стандартне відхилення було якнайменшим;

– стандартне відхилення для розміру респонсу. Це значення дозволяє визначити, наскільки переданий обсяг даних змінюється між запитами. Якщо стандартне відхилення розміру відповіді низьке, це означає, що обсяг даних залишається відносно стабільним для певної комбінації параметрів. Низьке значення стандартного відхилення для розміру відповіді важливе для мереж з обмеженою пропускною здатністю, оскільки це дозволяє більш точно планувати передачу даних.

Для розрахунку стандартного відхилення було використано SQL запит що наведено нижче (див. рис. 5.7). Результати розрахунків було винесено у окрему таблицю (див. табл. 5.2).

```
SELECT
    ROUND(STDDEV(ExecutionTimeMs), 2) AS stddev_execution_time,
    ROUND(STDDEV(ResponseSizeBytes), 2) AS stddev_response_size,
    ConfigFlags AS config_flags
FROM
    ResearchDB.RequestLogs
WHERE
    Endpoint != "/api/test" AND Endpoint != "/api/settings"
    AND ConfigFlags = {номер комбінації}
GROUP BY
    config_flags;
```

Рисунок 5.7 – SQL-запит для обчислення стандартного відхилення

Таблиця 5.2 – Розраховане стандартне відхилення

№ Комбінації	stddev_execution_time(мс)	stddev_response_size(мс)
1	18.37	15056.42
2	5.62	14701.14
3	3.01	4396.67
4	3.28	12524.53
5	4.34	15143.45
6	18.10	3653.72
7	2.16	4095.32
8	3.30	4088.86

5.2.3 Мінімальне та максимальне значення (Min/Max)

Розрахунок мінімальних та максимальних значень допомагає зрозуміти діапазон змін часу виконання та розміру відповіді при різних комбінаціях параметрів [7]:

– мінімальний та максимальний час виконання запиту. Мінімальне значення часу виконання показує, наскільки швидко система може обробляти запити в оптимальних умовах. Максимальне значення вказує на час обробки запиту в найгірших умовах. Порівняння цих значень для різних комбінацій дозволяє оцінити, як сильно параметри прискорення зменшують час виконання запиту в складних умовах;

– мінімальний та максимальний розмір респонсу. Ці показники дозволяють визначити, наскільки сильно змінюється обсяг даних при різних налаштуваннях прискорення. Наприклад, при активації стиснення очікується, що максимальний розмір респонсу значно зменшиться, що є показником ефективності цього методу.

Дані про мінімальне та максимальне значення збираються за допомогою SQL запиту (див. рис. 5.8). Результати обчислень представлені у таблиці нижче (див. табл. 5.3).

```
SELECT
    MIN(ExecutionTimeMs) AS min_execution_time,
    MAX(ExecutionTimeMs) AS max_execution_time,
    MIN(ResponseSizeBytes) AS min_response_size,
    MAX(ResponseSizeBytes) AS max_response_size,
    ConfigFlags AS config_flags
FROM
    ResearchDB.RequestLogs
WHERE
    Endpoint != "/api/test" AND Endpoint != "/api/settings"
AND
    ConfigFlags = {номер комбінації}
GROUP BY
    config_flags;
```

Рисунок 5.8 – SQL-запит для обчислення мінімального та максимального значень

Таблиця 5.3 – Розраховані мінімальні та максимальні значення

№ Комбінації	min_execution_time(мс)	max_execution_time(мс)	min_response_size(байт)	max_response_size(байт)
1	4	163	49	50633
2	0	24	49	48922
3	3	34	47	16234
4	3	16	49	41577
5	0	27	49	50440
6	0	216	47	17365
7	3	12	47	17704
8	0	15	47	17911

5.2.4 Результати обчислень та їх аналіз

5.2.4.1 Аналіз середніх значень для кожної комбінації способів прискорення

Проведене дослідження середніх значень дозволяє отримати загальне уявлення про ефективність використання різних комбінацій способів прискорення. Наведені середні значення виконання запитів (avg_execution_time) та розміру відповіді (avg_response_size) для кожної комбінації показують значні зміни, що дозволяє оцінити вплив кожного з способів та їх комбінацій на загальну продуктивність API.

Без прискорення (Комбінація 1): Середній час виконання запитів та розмір відповіді досить високі, що підтверджує початкове припущення про потребу в прискоренні. Висока варіативність значень вказує на відсутність стабільності без додаткових параметрів прискорення.

Кешування (Комбінація 2): Кешування забезпечує стабільне зниження часу виконання запитів, проте середній розмір відповіді залишається на високому рівні. Це свідчить про те, що кешування сприяє зниженню часу доступу до раніше збережених даних, але не оптимізує розмір переданих даних.

Пагінація (Комбінація 3): Застосування пагінації значно знижує розмір відповіді та забезпечує стабільність часу виконання запитів. Це рішення є ефективним для зменшення навантаження на систему при отриманні великої кількості записів, знижуючи обсяг переданих даних.

Компресія (Комбінація 4): Компресія знижує розмір відповіді, але має обмежений вплив на середній час виконання запитів. Це пояснюється тим, що компресія працює на рівні передачі даних, не прискорюючи обробку запитів на сервері.

Кешування + Компресія (Комбінація 5): Комбінація кешування та компресії забезпечує зниження часу виконання запитів та розміру відповіді, що

підтверджує їх позитивний взаємний вплив. Проте стандартне відхилення вказує на можливу варіативність результатів залежно від специфіки запитів.

Кешування + Пагінація (Комбінація 6): Ця комбінація дає суттєве зниження часу виконання запитів та розміру відповіді. Висока стабільність часу виконання є додатковою перевагою, що робить цю комбінацію ефективною для покращення продуктивності.

Компресія + Пагінація (Комбінація 7): Використання компресії разом з пагінацією знижує розмір відповіді, при цьому стабільність часу виконання покращується. Дана комбінація прискорює передачу та зменшує обсяг даних.

Кешування + Компресія + Пагінація (Комбінація 8): Комбінація усіх трьох параметрів забезпечує максимальне зниження часу виконання та розміру відповіді. Це оптимальне рішення, яке дозволяє досягти балансу між стабільністю результатів та їх продуктивністю.

На основі проведеного аналізу можна зробити висновок, що кожен спосіб прискорення робить свій внесок у зниження часу виконання або зменшення обсягу переданих даних. Однак найбільш ефективною є комбінація всіх трьох способів (кешування, компресія, пагінація), яка забезпечує максимальну ефективність і стабільність роботи API.

5.2.4.2 Аналіз стандартного відхилення

Середнє стандартне відхилення часу виконання запитів для різних комбінацій показує, наскільки стабільні результати для кожного набору параметрів. Найвищі значення стандартного відхилення спостерігаються у комбінаціях без прискорення (Комбінація 1) та з кешуванням і пагінацією (Комбінація 6). Це означає, що у цих випадках час виконання запитів є менш стабільним, що може свідчити про значні коливання у навантаженні системи або зміну швидкості залежно від розміру оброблюваних даних.

Натомість, комбінації з прискоренням через кешування і компресію (Комбінація 5) та через пагінацію і компресію (Комбінація 7) показують

найнижчі стандартні відхилення, що свідчить про стабільність часу виконання. Це вказує на те, що застосування цих способів прискорення сприяє більш передбачуваному і стабільному виконанню запитів.

5.2.4.3 Аналіз максимальних та мінімальних значень

Максимальні значення часу виконання запитів також значно варіюються залежно від комбінації способів прискорення. Найвищі максимальні значення часу виконання зафіксовані в комбінації без прискорення(Комбінація 1), де час виконання сягає до 163 мс, що є показником найбільшого навантаження на систему без прискорення.

З іншого боку, при використанні комбінацій кешування з компресією (Комбінація 5) та всіх трьох способів прискорення (Комбінація 8), максимальний час виконання значно знижується, що свідчить про ефективне зниження навантаження завдяки таким способам прискорення. Мінімальні значення часу виконання вказують на те, що навіть при найкращих умовах запит займає певний час, але його зменшення спостерігається при включенні прискорення параметрів.

5.2.4.4 Висновки

На основі аналізу статистичних показників можна зробити висновок, що:

– кешування, компресія та їх комбінація з іншими методами значно знижують час виконання запитів та стабілізують його, що видно з низьких значень стандартного відхилення для цих комбінацій;

– використання лише одного способу прискорення (наприклад, кешування або пагінації) зменшує час виконання, але не так ефективно, як комбінації методів. Комбінація компресії та пагінації, а також кешування з компресією, демонструють найкращі результати як за часом виконання, так і за стабільністю;

– комбінація всіх трьох способів (кешування, пагінація, компресія) є найбільш оптимальною за багатьма показниками, забезпечуючи найкращі результати для великого обсягу даних, але не завжди стабільну ефективність при малих обсягах даних.

Цей аналіз демонструє значення правильного вибору комбінацій способів прискорення, а також необхідність балансування між кількістю методів і їхньою взаємодією для досягнення найкращого результату.

5.3 Візуалізація результатів

Для аналізу зібраних даних і наочності проведених експериментів було побудовано чотири графіки, що відображають залежність часу виконання запитів та розміру респонсу від різних комбінацій параметрів прискорення.

Графік оцінки стабільності часу виконання для кожної комбінації наведено нижче (див. рис. 5.9).

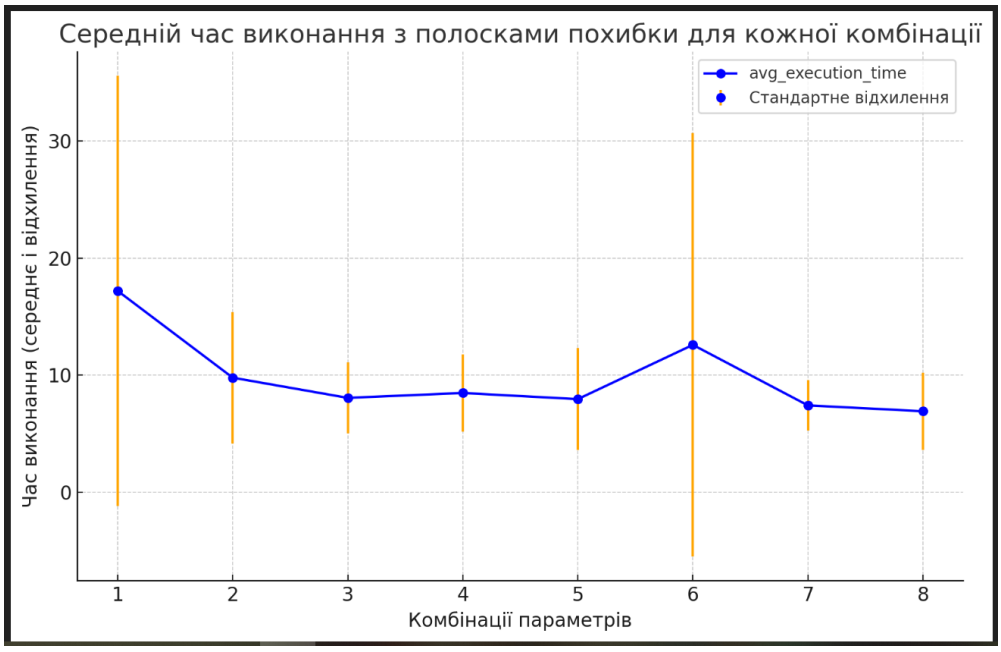


Рисунок 5.9 – Графік оцінки стабільності часу виконання для кожної комбінації

На цьому графіку представлено середній час виконання для кожної комбінації з доданими полосками похибки, які представляють стандартне відхилення. Графік показує, що комбінації з меншим стандартним відхиленням мають більш стабільний час виконання. Найбільше коливання спостерігається в комбінації 1 (без прискорення) та комбінації 6 (кешування + пагінація), що свідчить про нестабільність часу виконання у відсутності прискорення або з використанням певних поєднань

Залежність середнього часу виконання та середнього розміру респонсу від комбінацій способів прискорення зображено далі (див. рис. 5.10).

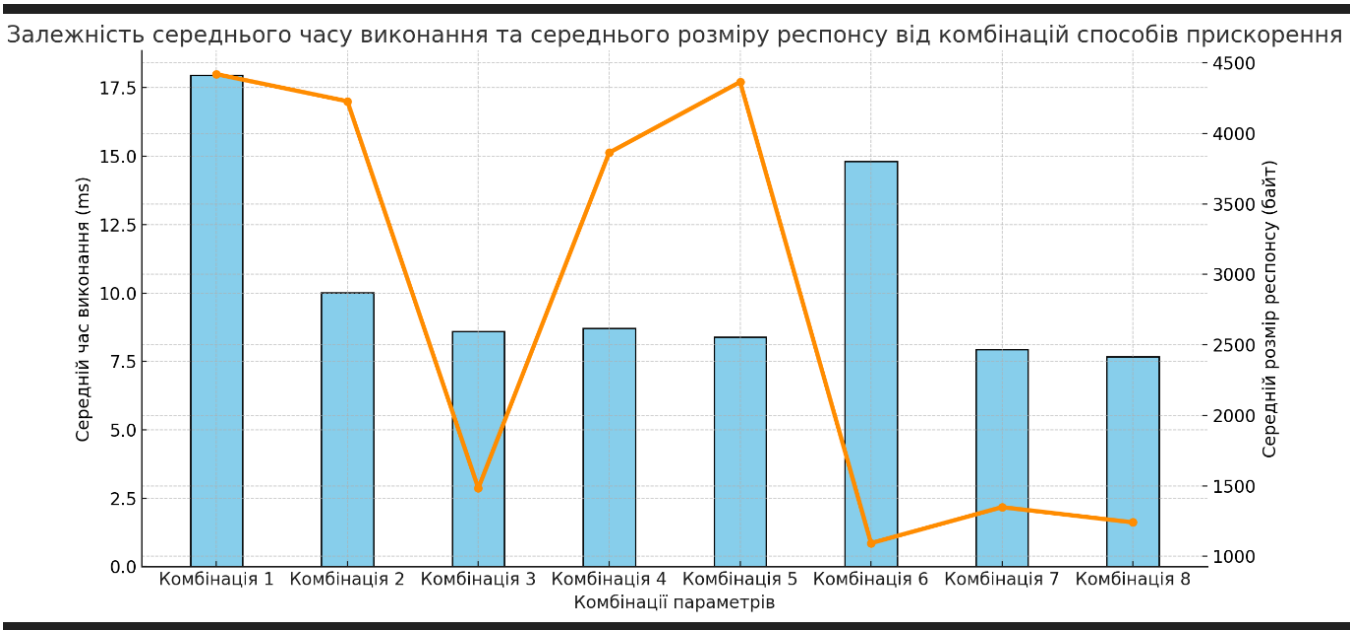


Рисунок 5.10 – Залежність середнього часу виконання та середнього розміру респонсу від комбінацій способів прискорення

Цей графік демонструє, як різні комбінації впливають на середній час виконання та середній розмір респонсу. Видно, що прискорення із застосуванням кешування або пагінації значно зменшує середній час виконання. Комбінація 3 (пагінація) та комбінація 8 (кешування + компресія + пагінація) показали найнижчий середній час виконання, тоді як розмір респонсу залишається на приблизно однаковому рівні.

Наступним є графік мінімального, максимального та середнього часу виконання за комбінаціями (див. рис. 5.11).

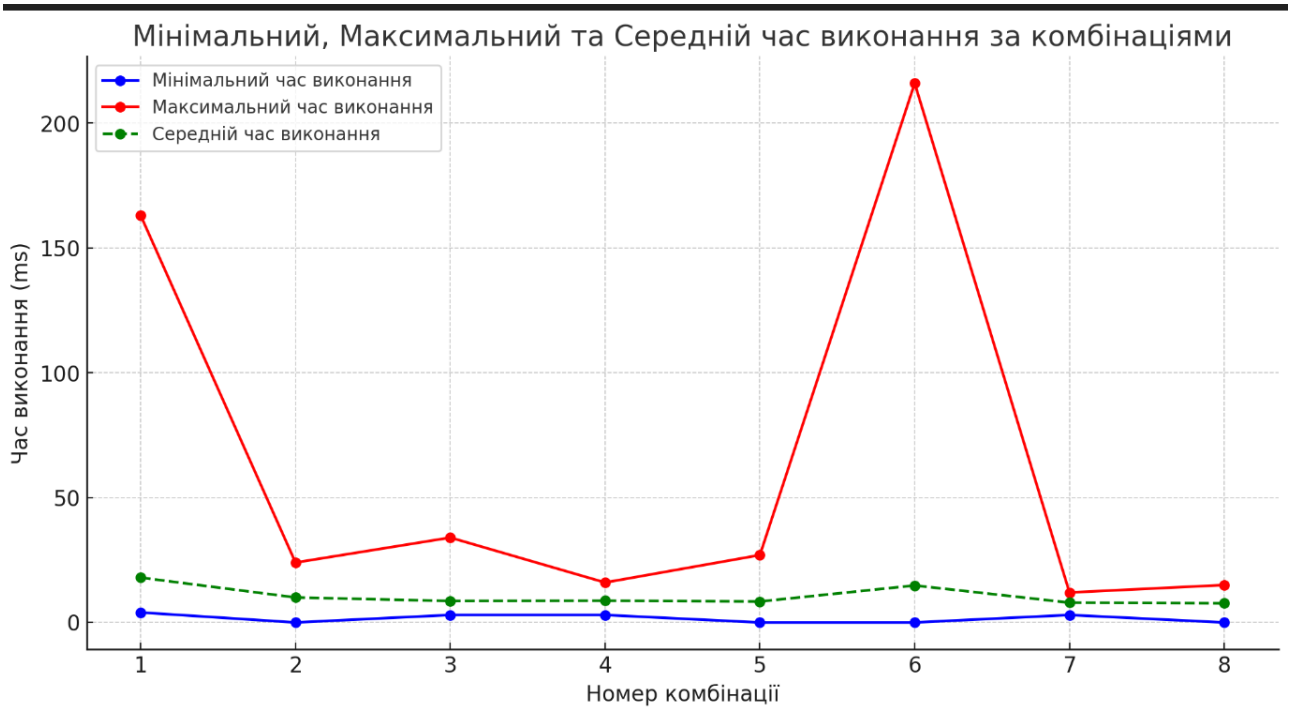


Рисунок 5.11 – Графік з відображенням мінімального та максимального часу виконання

На цьому графіку відображено мінімальний, максимальний та середній час виконання для кожної комбінації. Комбінації 1 та 6 виділяються значним збільшенням максимального часу виконання, що свідчить про можливі пікові затримки під час обробки запитів без прискорення або за певних умов. Мінімальний час виконання відносно стабільний для всіх комбінацій, що підтверджує ефективність прискорення в забезпеченні мінімального часу на обробку запиту.

І останнім є графік мінімального, максимального та середнього розміру респонсу за комбінаціями (див. рис. 5.12).

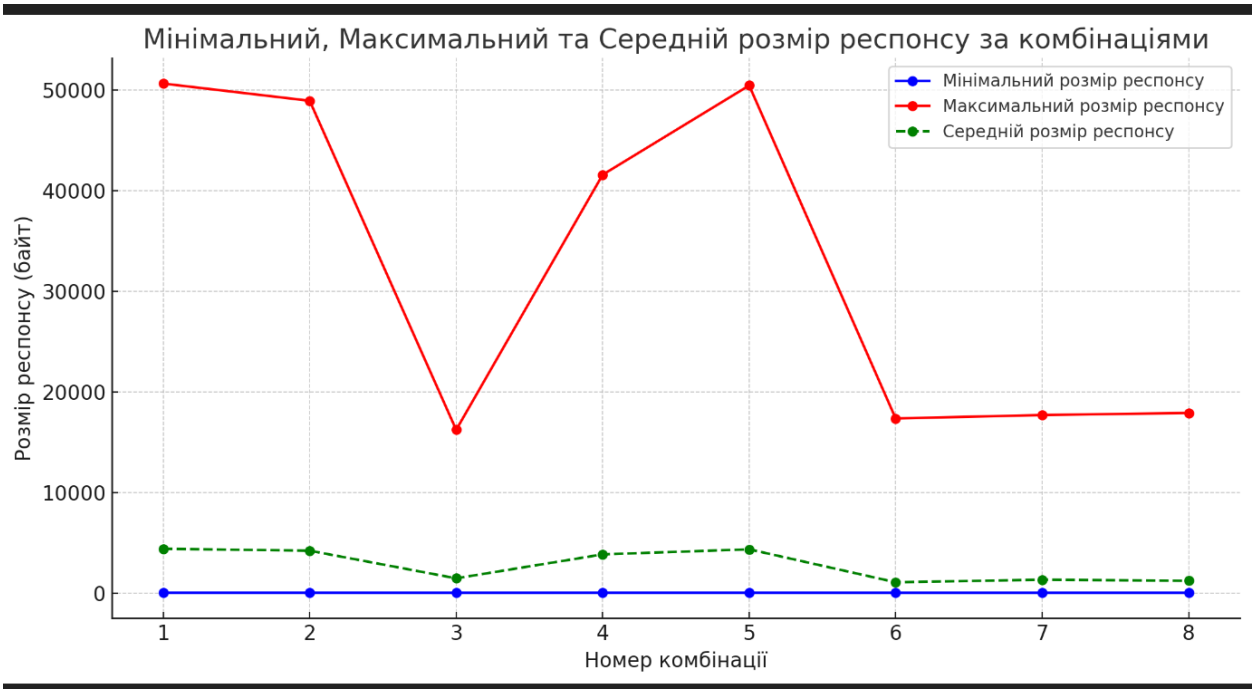


Рисунок 5.12 – Графік з відображенням мінімального та максимального розміру респонсу

Цей графік дозволяє оцінити вплив прискорення на розмір респонсу. Найбільші розміри спостерігаються у комбінації 1 (без прискорення) та комбінації 2 (тільки кешування). Комбінації 3, 7 та 8 значно зменшують розмір респонсу завдяки пагінації та компресії, що підтверджує ефективність цих підходів у зменшенні навантаження на канал передачі даних.

На основі представлених графіків можна зробити наступні висновки:

– комбінації прискорення значно впливають на стабільність та швидкість виконання запитів. Комбінації з кешуванням і пагінацією забезпечують найкращі показники, тоді як відсутність прискорення або використання лише кешування призводить до високих коливань у часі виконання;

– зменшення розміру респонсу відбувається при використанні пагінації та компресії, що дозволяє знизити навантаження на мережу та підвищити швидкість завантаження даних для кінцевого користувача;

– максимальні показники часу виконання та розміру респонсу вказують на можливі точки покращення, де слід розглянути додаткові способи прискорення для зменшення пікових значень.

Загалом, результати підтверджують ефективність обраних методів прискорення для підвищення продуктивності веб-сервісів і надають рекомендації для подальшого покращення архітектури API.

5.4 Інтерпретація результатів

Після проведення експериментів та аналізу отриманих результатів можена зробити детальні висновки щодо ефективності різних способів прискорення роботи запитів у RESTful Web API. У ході дослідження були протестовані різні комбінації способів прискорення (кешування, компресія та пагінація), щоб визначити їхній вплив на час виконання запитів та розмір респонсу.

5.4.1 Порівняння результатів з початковою гіпотезою

На початку дослідження була висунута гіпотеза, що застосування певних способів прискорення зменшить середній час виконання запитів на значний відсоток, що в результаті приведе до більш ефективної роботи системи. Після аналізу середніх значень часу виконання, стандартного відхилення та розміру респонсу для кожної комбінації ми можемо сказати, що гіпотеза здебільшого підтвердилася. Деякі комбінації дійсно показали помітне зменшення часу виконання, особливо в поєднанні кешування з іншими способами прискорення.

Аналіз середнього часу виконання показав, що комбінація з усіма способами прискорення (кешування, компресія та пагінація) виявилася однією з найбільш ефективних. Вона дозволила значно знизити середній час виконання запитів порівняно з комбінаціями, де використовувалися лише окремі способи прискорення. Однак, слід зазначити, що поєднання кешування та пагінації, без компресії, також дало позитивний ефект, особливо у випадках із запитами, що обробляють великі обсяги даних. Компресія, у свою чергу, показала себе як ефективний спосіб зменшення розміру респонсу, але її вплив на час виконання був менш значним у порівнянні з іншими способами прискорення.

5.4.2 Стабільність результатів та аналіз крайніх значень

Вивчення стандартного відхилення показало, що різні комбінації способів прискорення мають різну стабільність. Комбінації з високим стандартним відхиленням, такі як кешування у поєднанні з компресією, показують, що результати можуть варіюватися залежно від конкретних умов запиту або структури даних. З іншого боку, комбінації з меншою кількістю способів прискорення мали значно менше стандартне відхилення, що свідчить про стабільнішу поведінку. Це може бути важливим фактором для систем, де стабільність часу виконання запитів є критичною.

Результати мінімального та максимального часу виконання для різних комбінацій демонструють, як різні способи прискорення впливають на пікове навантаження. Максимальні значення часу виконання, наприклад, були найвищими для комбінацій без кешування, що свідчить про важливість кешування для зниження пікових затримок. Мінімальні значення часу були подібними для всіх комбінацій, що свідчить про те, що при мінімальних навантаженнях система може працювати ефективно навіть без додаткових способів прискорення. Але здебільш це може свідчити про залежність від серверних ресурсів, та як я вважаю потребую детальнішого розглядання при обмеженні ресурсів сервера.

Компресія показала свою ефективність у зниженні розміру респонсу, що було особливо помітно для запитів із великими обсягами даних. Поєднання компресії з кешуванням дозволило досягти найменшого розміру респонсу. Це свідчить про те, що для систем, де збереження трафіку або прискорення передачі даних є пріоритетом, компресія є необхідною.

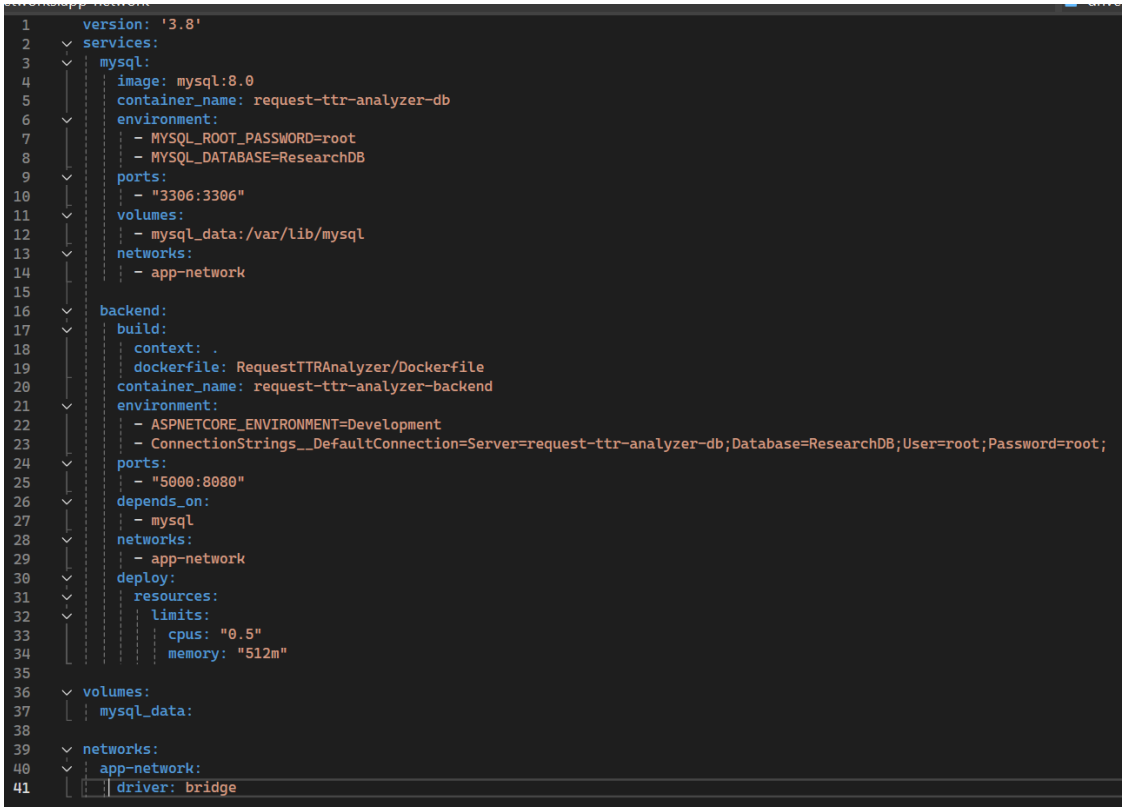
5.5 Проведення дослідження в ізольованому середовищі

Для більш комплексної оцінки впливу параметрів прискорення (кешування, компресії та пагінації) слід провести додатковий експеримент за умов обмежених

ресурсів. Таке середовище моделює реальні сценарії, де серверна інфраструктура може бути обмежена, наприклад, через використання контейнеризації або недостатню обчислювальну потужність.

Оскільки БД уже ізольоване ще на початку дослідження то для цього експерименту бекенд частину проекту було поміщено у Docker контейнер.

Змінений Docker compose файл можно побачити нижче (див. рис. 5.13). В нього було добавлено сервіс backend для якого було встановлено обмеження по доступним для контейнера ресурсам. Контейнеру с бекендом було виділено 0.5 ядра ЦП та 512 мб оперативної пам’яті.



```yaml
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    container_name: request-ttr-analyzer-db
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=ResearchDB
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
    networks:
      - app-network

  backend:
    build:
      context: .
      dockerfile: RequestTTRAnalyzer/Dockerfile
    container_name: request-ttr-analyzer-backend
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ConnectionStrings__DefaultConnection=Server=request-ttr-analyzer-db;Database=ResearchDB;User=root;Password=root;
    ports:
      - "5000:8080"
    depends_on:
      - mysql
    networks:
      - app-network
    deploy:
      resources:
        limits:
          cpus: "0.5"
          memory: "512m"

volumes:
  mysql_data:

networks:
  app-network:
    driver: bridge
```

Рисунок 5.13 – Оновлена версія Docker-compose файлу

Для забезпечення коректного зв’язку між сервісом БД та сервісом бекенда було прописано конфігурацію для створення нової віртуальної мережі для цих двох сервісів.

5.5.1 Обчислення основних статистичних показників

Для порівняння результатів у стандартному середовищі та у ізольованому з обмеженими ресурсами було проведено повторний збір даних за умови обмеження ресурсів бекенду. Формування та вибірка результатів виконувалися за тими самими методами та скриптами, що були використані у попередньому підрозділі (див. підрозділ 5.2).

Розраховане середні значення можна побачити нижче у таблиці (див. табл. 5.4).

Таблиця 5.4 – Середні значення за типом реквесту та комбінацією

Комбінація	avg_execution_time(мс)	avg_response_size(байт)	request_count(шт)	request_type
1	12.37	370.78	60	POST
1	12.04	16647.64	120	GET
1	10.10	371.37	60	PUT
1	8.07	50.00	60	DELETE
2	8.58	375.07	60	POST
2	7.08	15814.03	120	GET
2	9.58	371.17	60	PUT
2	9.08	50.00	60	DELETE
3	9.17	384.53	60	POST
3	9.18	4583.58	120	GET
3	9.50	371.15	60	PUT
3	8.03	50.00	60	DELETE
4	8.53	348.58	60	POST
4	9.19	14628.03	120	GET
4	8.88	366.72	60	PUT
4	6.17	50.00	60	DELETE
5	14.13	352.80	60	POST
5	6.57	15184.19	120	GET
5	8.30	367.08	60	PUT
5	5.43	50.00	60	DELETE
6	5.82	349.25	60	POST
6	2.70	3968.85	120	GET
6	7.32	372.13	60	PUT
6	4.92	50.00	60	DELETE
7	5.65	364.87	60	POST
7	6.44	4939.40	120	GET
7	8.52	367.10	60	PUT
7	5.40	50.00	60	DELETE

Продовження Таблиці 5.4 – Середні значення за типом реквесту та комбінацією

Комбінація	avg_execution_time(мс)	avg_response_size(байт)	request_count(шт)	request_type
8	9.75	372.30	60	POST
8	5.32	4097.21	120	GET
8	7.72	367.02	60	PUT
8	5.55	50.00	60	DELETE

Можна помітити, що середні показники часу почали значно відрізнятися, особливо для запитів типу GET та POST.

Засновуючись на проміжному результаті випливає висновок, що за умови обмеження серверних ресурсів деякі способи прискорення будуть менш ефективними. Наприклад компресія, вона напряму залежить від ресурсів процесора та при нестачі ресурсу виконання операцію буде сповільнено.

Наступним статистичним параметром для ізольованого середовища при обмеженні ресурсів буде стандартне відхилення (див. табл 5.5).

Таблиця 5.5 – Розраховане стандартне відхилення

№ Комбінації	stddev_execution_time(мс)	stddev_response_size(байт)
1	14.38	14240.71
2	10.94	14215.77
3	11.78	3970.25
4	11.39	12446.47
5	13.69	13671.09
6	4.89	4133.4
7	8.83	4253.1
8	11.93	4032.74

У стандартному середовищі комбінації 3, 4, 5, 7, 8 демонструють менше стандартне відхилення часу виконання запитів, тобто більш стабільну продуктивність. Це вказує на те що зазначені комбінації залежать від ресурсів сервера та обмеженість цих ресурсів буде впливати на час виконання запиту.

Проте для комбінації 6, стандартне відхилення зменшилося на 73%, що свідчить про позитивний вплив ізоляції на цю комбінацію (див. табл. 5.7).

Останнім статистичним показником є максимальні та мінімальні значення (див. табл. 5.6).

Таблиця 5.6 – Розраховані мінімальні та максимальні значення

№ Комбінації	min_execution_time(мс)	max_execution_time(мс)	min_response_size(байт)	max_response_size(байт)
1	2	132	49	47452
2	0	73	49	47027
3	2	58	47	16105
4	1	68	49	41342
5	0	80	49	45365
6	0	40	47	17653
7	1	78	47	18076
8	0	76	47	15869

В ізольованій середовищі максимальний час виконання запитів значно зріс для більшості комбінацій (2, 3, 4, 5, 7, 8), що вказує на погіршення стабільності API під час обробки важких запитів.

Лише для комбінацій 1 та 6 максимальний час виконання зменшився, що свідчить про позитивний вплив ізоляції для цих сценаріїв. Це може свідчити про те що ці комбінації менше залежать від ресурсів сервера.

5.5.2 Візуалізація результатів та висновки

Помітно, що для деяких комбінацій (наприклад, 1 та 2) середній час виконання в ізольованому середовищі суттєво знизився, тоді як для інших (наприклад, 3 та 5) спостерігається незначне збільшення (див. табл. 5.7). Це свідчить про різний вплив ізоляції на ефективність виконання запитів залежно від налаштувань параметрів (див. рис. 5.14).

Від'ємні значення в полі "Відмінність (%)" означають, що середній час виконання запитів у ізольованому середовищі зменшився порівняно зі стандартним середовищем.

Таблиця 5.7 – Порівняння середнього часу виконання між середовищами

Комбінація	Середнє (Стандартне середовище, мс)	Середнє (Ізольоване середовище, мс)	Відмінність (%)
1	17.96	10.65	-40.73
2	10.02	8.58	-14.35
3	8.60	8.97	4.30
4	8.71	8.19	-5.97
5	8.40	8.61	2.50
6	14.83	5.19	-64.99
7	7.94	6.50	-18.08
8	7.67	7.09	-7.63

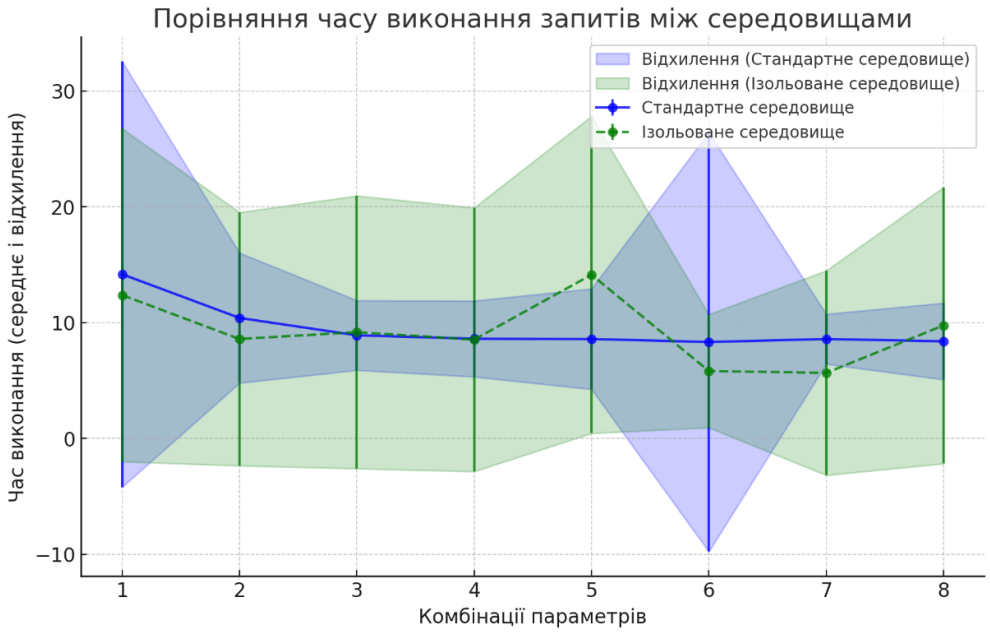


Рисунок 5.14 – Порівняння середнього часу виконання запитів між середовищами

Таблиця 5.8 – Аналіз стандартного відхилення часу виконання запитів

Комбінація	Стандартне середовище (Стандартне відхилення , мс)	Ізольоване середовище (Стандартне відхилення , мс)	Відмінність (%)
1	18.37	14.38	-21.7
2	5.62	10.94	94.7
3	3.01	11.78	291.3
4	3.28	11.39	247.1
5	4.34	13.69	215.3
6	18.10	4.89	-73.0
7	2.16	8.83	308.8
8	3.30	11.93	261.8

Аналіз даних таблиці показує суттєві відмінності в стабільності виконання запитів між стандартним та ізольованим середовищами (див. рис. 5.15). Зменшення стандартного відхилення спостерігається лише для комбінацій 1 (-21.7%) та 6 (-73.0%), що свідчить про покращення стабільності в ізольованому середовищі, ймовірно, через оптимізацію ресурсів або ефекти ізоляції. Для решти комбінацій (2, 3, 4, 5, 7, 8) стандартне відхилення значно зростає, досягаючи максимуму для комбінацій 7 (308.8%) та 8 (261.8%) (див. табл. 5.8). Це свідчить про суттєву варіативність у часі виконання запитів в ізольованому середовищі, що, ймовірно, пов’язано з обмеженнями ресурсів.

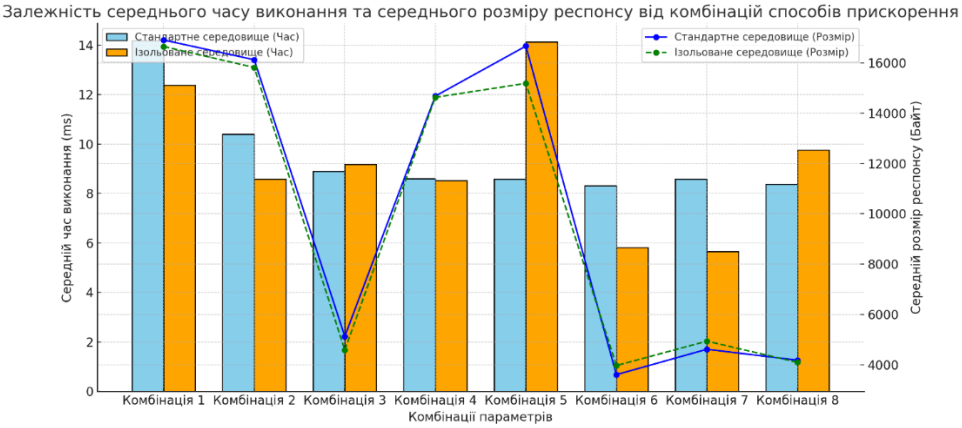


Рисунок 5.15 – Порівняння залежності різних комбінацій від середнього часу виконання та розміру респонсу між середовищами

Таблиця 5.9 – Аналіз мінімальних і максимальних значень між стандартною та ізольованою середами

Комбінація	**Стандартне середовище (мін. час , мс)**	**Ізольоване середовище (мін. час , мс)**	**Відмінність (%)**	**Стандартне середовище (макс. час , мс)**	**Ізольоване середовище (макс. час , мс)**	**Відмінність (%)**
1	4	2	-50.0	163	132	-19.0
2	0	0	0.0	24	73	204.2
3	3	2	-33.3	34	58	70.6
4	3	1	-66.7	16	68	325.0
5	0	0	0.0	27	80	196.3
6	0	0	0.0	216	40	-81.5
7	3	1	-66.7	12	78	550.0
8	0	0	0.0	15	76	406.7

Аналіз таблиці 5.9 демонструє суттєві відмінності між мінімальними та максимальними значеннями часу виконання запитів у стандартному та ізольованому середовищах (див. табл. 5.9). Для мінімального часу ізольоване середовище у більшості комбінацій (1, 3, 4, 7) показує зменшення значень, зокрема найзначніше зниження спостерігається для комбінацій 4 (-66.7%) і 7 (-66.7%). Це свідчить про потенційне підвищення ефективності ізоляції у випадках з невеликими обсягами операцій. У той же час для комбінацій 2, 5, 6 та 8 мінімальний час залишається незмінним, що може вказувати на незначний вплив ізоляції з обмеженням ресурсів у цих випадках.

Щодо максимального часу, ізольоване середовище демонструє значне збільшення у більшості комбінацій, зокрема у комбінаціях 4 (325.0%), 7 (550.0%) та 8 (406.7%). Це свідчить про суттєву нестабільність в умовах високого навантаження або складних операцій в ізольованому середовищі. Виняток становлять комбінації 1 (-19.0%) і 6 (-81.5%), де максимальний час у ізольованому середовищі зменшується, що може вказувати на покращення обробки запитів за цих конфігурацій.

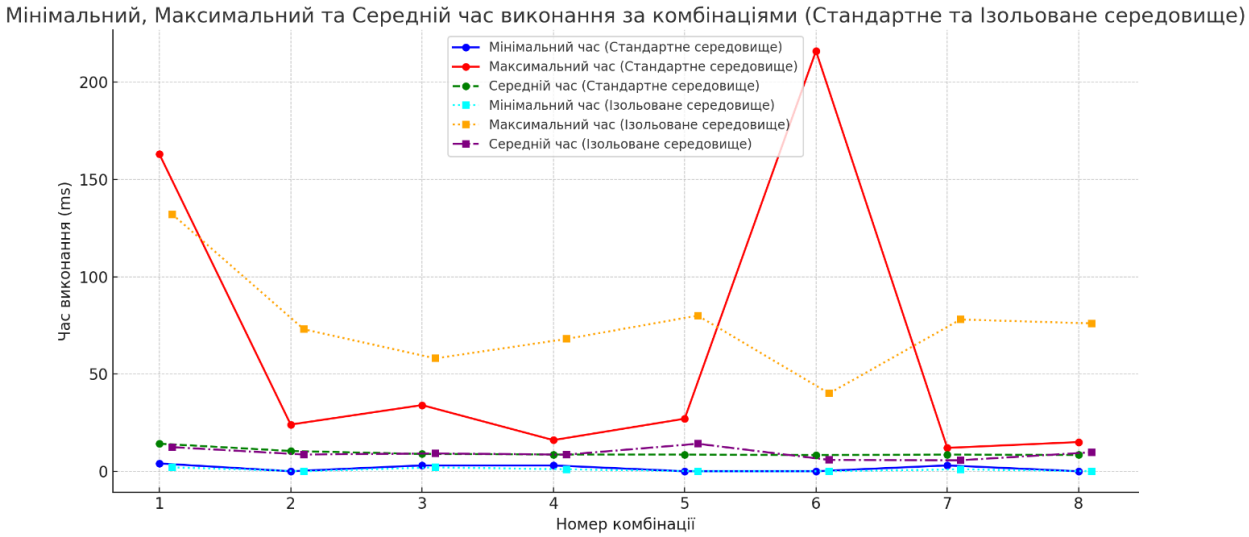


Рисунок 5.16 – Порівняння мінімального та максимального часу виконання між середовищами

На основі проведеного аналізу можна зробити висновок, що обмеження ресурсів бекенда в ізольованому середовищі має неоднозначний вплив на час виконання запитів для різних комбінацій параметрів.

У випадках із невеликими навантаженнями або простими операціями (комбінації з мінімальними значеннями часу) ізоляція демонструє позитивний вплив, зменшуючи мінімальний час виконання запитів. Це свідчить про те, що ізольоване середовище здатне прискорювати обробку простих операцій. Водночас для складніших сценаріїв або великих навантажень (максимальні значення часу) ізоляція може призводити до значного зростання часу виконання через обмеження доступних ресурсів, особливо для комбінацій 7 та 8.

Крім того, аналіз середнього та стандартного відхилення показав, що стабільність виконання запитів також залежить від комбінації параметрів.

ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ

У ході роботи було досліджено методи зменшення часу виконання запитів у RESTful API на прикладі монолітного додатка, побудованого за Onion-архітектурою на базі ASP.NET Core. Результатом стало впровадження ефективних комбінацій способів прискорення часу виконання запитів, які дозволили досягти покращення продуктивності API. Головні досягнення роботи включають:

– розробка програмного забезпечення для збору метрик продуктивності (середній час виконання, стандартне відхилення, мінімальні та максимальні значення);

– впровадження способів прискорення: кешування, стиснення даних, пагінація;

– проведення експериментів із застосуванням різних комбінацій методів, що дозволило оцінити їхній вплив на швидкість виконання запитів та стабільність роботи системи;

– проведення додатково експерименту у ізольованому середовищі з обмеженими ресурсами.

Результати дослідження підтвердили ефективність комбінованих способів прискорення для монолітних додатків, що дозволяє зменшити час відгуку серверів без переходу на складніші архітектури.

Дослідження показало, що:

– кешування забезпечує стабільний час виконання запитів, але його ефективність залежить від частоти оновлення даних;

– пагінація зменшує обсяг даних у відповіді та покращує швидкість роботи при обробці великих обсягів інформації;

– стиснення даних (компресія) дозволяє суттєво зменшити розмір відповіді API, що є критичним для роботи у високонавантажених мережах;

– найбільш ефективною виявилася комбінація кешування, пагінації та стиснення, що дозволила досягти як високої швидкодії, так і стабільності результатів;

– у деяких випадках (комбінація 6) ізоляція значно покращує стабільність, тоді як для більшості інших спостерігається погіршення. Таким чином, обмеження ресурсів у бекенді в ізольованому середовищі може як прискорювати, так і погіршувати продуктивність API залежно від характеру операцій.

Рекомендації:

– для додатків, що потребують стабільності, доцільно поєднувати кешування з пагінацією;

– у високонавантажених системах слід додатково використовувати стиснення даних для мінімізації мережевого трафіку;

– подальше тестування необхідно проводити з урахуванням різних обсягів даних та сценаріїв навантаження.

Для подальшого розвитку програми пропонується:

– візуалізація метрик: створення графічного інтерфейсу для відображення часу виконання запитів, розміру відповідей та інших показників у реальному часі;

– дослідження додаткових оптимізацій: впровадження CDN, оптимізація запитів до бази даних та управління ресурсами серверів;

– розширення тестових сценаріїв: проведення тестування системи під різними умовами навантаження та для різних типів запитів.

Отримані результати та запропоновані рекомендації дозволять підвищити продуктивність RESTful API у монолітних додатках, а також створити основу для подальших досліджень у цій галузі.

Результати дослідження були представлені на конференції «Сучасні Інформаційні Та Комунікаційні Технології На Транспорті, В Промисловості І Освіті» та наведені у документі Додаток В «Тези доповіді для конференції».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jin, B., Sahni, S., Shevat, A. Designing Web APIs [Текст] / B. Jin, S. Sahni, A. Shevat. – Sebastopol : O'Reilly Media, 2018. – 204 с.
2. Richardson, L., Amundsen, M. RESTful Web APIs [Текст] / L. Richardson, M. Amundsen. – Sebastopol : O'Reilly Media, 2013. – 406 с.
3. Fundamentals of Middleware in ASP.NET Core [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0. – Дата доступу: 08.01.2025. – Назва з екрану.
4. Common Web Application Architectures [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/ru-ru/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures. – Дата доступу: 08.01.2025. – Назва з екрану.
5. Entity Framework Core [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/ef/core/. – Дата доступу: 08.01.2025. – Назва з екрану.
6. ASP.NET Identity Using MySQL Storage with an EntityFramework MySQL Provider [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/ru-ru/aspnet/identity/overview/getting-started/aspnet-identity-using-mysql-storage-with-an-entityframework-mysql-provider. – Дата доступу: 08.01.2025. – Назва з екрану.
7. Hastie, T., Tibshirani, R., Friedman, J. The Elements of Statistical Learning [Текст] / T. Hastie, R. Tibshirani, J. Friedman. – New York : Springer, 2009. – 745 с.

ДОДАТОК А

Технічне завдання

ЗАТВЕРДЖУЮ

Проректор
Українського державного університету науки і технології
Анатолій РАДКЕВИЧ

RequestTTRAnalyzer

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ

1116130.1431-01-ЛЗ

Завідувач кафедри КІТ

_____Вадим ГОРЯЧКІН

Керівник розробки

_____Вадим АНДРЮЩЕНКО

Виконавець

_____Олексій ПОДЕДВОРНИЙ

Нормоконтролер

_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
1116130.1431-01-ЛЗ

RequestTTRAnalyzer

Технічне завдання

1116130.1431-01

Листів 12

2025

ЗМІСТ

А.1 ВВЕДЕННЯ

В умовах сучасного розвитку інформаційних технологій ефективність роботи веб-додатків та швидкість обробки запитів стають ключовими факторами для забезпечення позитивного користувацького досвіду. Величезна кількість веб-сервісів та RESTful API працюють у режимі реального часу, обслуговуючи мільйони користувачів щоденно. Це призводить до зростання вимог до продуктивності та швидкості доступу до даних.

Підвищення продуктивності API є складним і багатогранним завданням, яке охоплює не тільки оптимізацію коду, але й налаштування параметрів кешування, стиснення даних, масштабування бази даних та інші фактори. Ефективне керування цими параметрами здатне значно покращити швидкість виконання запитів та знизити навантаження на сервери.

У даній роботі проводиться розробка та дослідження системи збору метрик про продуктивність запитів до RESTful API. Основна мета роботи полягає у створенні інструментарію для збору та аналізу метрик часу виконання запитів, кількості переданих даних та інших показників, що впливають на продуктивність API. Реалізація системи включає використання сучасних технологій, таких як ASP.NET Core, MySQL та Onion-архітектура, що дозволяє досягти високої гнучкості та масштабованості додатку.

Зібрані дані дадуть можливість оцінити ефективність різних методів прискорення та знайти шляхи покращення продуктивності, що є актуальним для будь-якої системи, яка працює з великими обсягами даних та має вимоги до швидкості обробки запитів.

А.2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

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

Тема дипломної роботи - “ДОСЛІДЖЕННЯ СПОСОБІВ ЗМЕНШЕННЯ ЧАСУ ВИКОНАННЯ ЗАПИТІВ У RESTFUL API”. Керівник - доцент Андрющенко В.О.

А.3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка системи спрямована на створення інструменту для збору, обробки та аналізу метрик продуктивності RESTful API, що використовуються у монолітних додатках. Основне призначення цього інструментарію полягає у можливості вимірювання часу виконання запитів, визначення ефективності різних підходів прискорення та моніторингу поведінки API в різних умовах навантаження.

Система повинна надати користувачам можливість отримувати детальну інформацію про кожен запит, зокрема: час обробки, розмір відповіді, кількість записів, складність запиту та використані параметри прискорення. Ці дані будуть використовуватися для порівняльного аналізу та визначення факторів, які найбільше впливають на продуктивність додатку.

Розробка дозволить підвищити ефективність роботи веб-сервісів за рахунок вибору оптимальних параметрів конфігурації та забезпечення швидкої та стабільної обробки запитів до API.

А.4 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

А.4.1 Вимоги до функціональних характеристик

Збір метрик продуктивності. Програмне забезпечення повинно збирати метрики про кожен запит до RESTful API, включаючи час виконання, розмір відповіді, кількість записів, складність запиту та інші параметри.

Гнучка конфігурація параметрів прискорення. Система повинна дозволяти динамічно змінювати налаштування глобальних параметрів прискорення, таких як кешування, стиснення даних, пагінація, та відображати їх вплив на продуктивність.

Зберігання та обробка зібраних даних. Програма повинна зберігати всі метрики у базі даних для подальшого аналізу та надавати засоби для їх обробки і візуалізації.

Кросплатформенність. Програмний продукт має бути сумісним із різними операційними системами та працювати незалежно від середовища виконання.

А.4.2 Вимоги до надійності

Стійкість до помилок. Програма повинна коректно обробляти виняткові ситуації (Exceptions) та реєструвати всі помилки у логах.

Стабільність роботи під навантаженням. Система повинна витримувати високе навантаження запитами (тестування до 1000 запитів на хвилину) без збоїв або помітної деградації продуктивності.

Збереження цілісності даних. Всі дані, зібрані під час збору метрик, мають бути захищеними від втрати або некоректного збереження у разі раптових збоїв.

А.4.3 Вимоги експлуатації

Зручність конфігурації. Програмне забезпечення має надавати простий інтерфейс (UI чи API) для налаштування параметрів прискорення та моніторингу стану системи.

Можливість розширення. Система повинна підтримувати просте розширення функціональності, наприклад, додавання нових типів метрик або прискорення без необхідності повного рефакторингу існуючого коду.

Автоматизоване розгортання. Забезпечення можливості швидкого розгортання та налаштування програми на різних середовищах (наприклад, тестове та продакшн-середовище).

А.4.4 Вимоги до складу та параметрів технічних засобів

База даних. Система повинна підтримувати реляційну базу даних MySQL (версія 5.7 або вище).

Серверна частина. Використання сервера з можливістю розміщення ASP.NET Core додатків (наприклад, Windows Server або Linux із .NET Core Runtime).

Обсяг пам'яті. Мінімум 2 ГБ оперативної пам'яті для коректної роботи сервісу та бази даних при середньому навантаженні.

А.4.5 Вимоги до інформаційної та програмної сумісності

Сумісність з ORM (Entity Framework Core). Програма має використовувати EF Core для взаємодії з базою даних, що забезпечує абстрагування від конкретної СУБД.

Підтримка сучасних стандартів JSON та HTTP. Програмне забезпечення повинно працювати з сучасними стандартами обміну даними (HTTP 1.1/2, JSON) для максимальної інтеграції з іншими системами.

Інтеграція з інструментами моніторингу та логування. Система має бути сумісною з інструментами для моніторингу (наприклад, Serilog, Elasticsearch) та можливістю підключення до зовнішніх систем збору логів.

А.4.6 Вимоги до маркування і упаковки

Візуальні елементи. У разі розробки інтерфейсу користувача (UI) всі елементи дизайну повинні бути чітко позначені й організовані для зручності користувача.

А.4.7 Вимоги до транспортування та зберігання

Дистрибутив. Програмне забезпечення повинно постачатися у вигляді архіву або контейнера (Docker), який містить всі необхідні компоненти для розгортання.

Зберігання резервних копій. Рекомендовано регулярне створення резервних копій бази даних та конфігураційних файлів для захисту від втрати даних у разі збою.

Захист даних. Забезпечення безпечного зберігання даних, що включає як метрики запитів, так і лог-файли.

А.5 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Таблиця А.1 – Стадії та етапи розробки

Етап розробки	**Завдання**	**Строки виконання**
1. Планування та аналіз вимог	Вивчення предметної області, формулювання вимог до системи, розробка технічного завдання (ТЗ).	03.09.24
2. Проектування системи	Розробка архітектури програми на основі Onion-підходу, проектування бази даних, підготовка специфікацій для сервісів та API.	10.09.24
3. Розробка основної функціональності	Реалізація API та сервісів для CRUD-операцій (напр., ProductsService), інтеграція бази даних, впровадження middleware для збору метрик.	17.09.24
4. Тестування та налагодження	Проведення модульних або інтеграційних тестів, перевірка коректності збору метрик, верифікація роботи параметрів прискорення.	28.09.24
5. Налаштування та тестування ізольованого середовища	Підготовка ізольованого середовища до експлуатації програми. Тестування в умовах обмежених системних ресурсів.	01.12.24
6. Вдосконалення функціоналу	Покращення продуктивності системи, внесення змін на основі тестування, розширення функціональності (додавання нового контролера).	02.01.25

А.6 ПОРЯДОК І КОНТРОЛЬ ПРИЙМАННЯ

Контроль за виконанням роботи здійснює керівник розробки доц. Андрющенко В.О.

Прийом здійснюється комісією у складі:

- Горячкін В. М. (керівник підрозділу);
- Андрющенко В.О. (керівник розробки).

А.7 БІБЛІОГРАФІЧНИЙ СПИСОК

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

ДОДАТОК Б

Текст програми

ЗАТВЕРДЖУЮ

Проректор
Українського державного університету науки і технології
Анатолій РАДКЕВИЧ

RequestTTRAnalyzer

Текст програми

ЛИСТ ЗАТВЕРДЖЕННЯ

1116130.1431-01 12 01-ЛЗ

Завідувач кафедри КІТ

_____Вадим ГОРЯЧКІН

Керівник розробки

_____Вадим АНДРЮЩЕНКО

Виконавець

_____Олексій ПОДЕДВОРНИЙ

Нормоконтролер

_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
1116130.1431-01 12 01-ЛЗ

RequestTTRAnalyzer

Текст програми

1116130.1431-01 12 01-ЛЗ

Листів 48

2025

АНОТАЦІЯ

Документ 1116130.1431-01 12 01 «RequestTTRAnalyzer. Текст програми» У рамках магістерської роботи розроблено програму для дослідження продуктивності RESTful API, побудовану на основі ASP.NET Core із застосуванням Onion-архітектури. Програма включає механізми збору метрик продуктивності, такі як середній час виконання запитів, стандартне відхилення, мінімальні та максимальні значення, а також розмір респонсу. Для вимірювання метрик реалізовано middleware, яке інтегрується в процес обробки HTTP-запитів, і базу даних MySQL для збереження результатів. Розроблене програмне забезпечення також містить інструменти для налаштування параметрів прискорення, зокрема кешування, пагінації та компресії, що дозволяє проводити експериментальний аналіз впливу різних способів прискорення на продуктивність API.

ЗМІСТ

Б.1 ТЕКСТ ПРОГРАМИ

User.cs

```
using Microsoft.AspNetCore.Identity;
using RequestTTRAnalyzer.Domain.Entities.Interfaces;
using RequestTTRAnalyzer.Domain.Entities.Orders;
using System.ComponentModel.DataAnnotations.Schema;
using File = RequestTTRAnalyzer.Domain.Entities.Files.File;

namespace RequestTTRAnalyzer.Domain.Entities.Identity
{
    public class User : IdentityUser<int>, IEntity<int>
    {
        #region Properties

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public int? AvatarId { get;set; }

        #endregion

        #region Navigation Properties

        [InverseProperty("Creator")]
        public virtual ICollection<File> UploadedFiles { get;set;
}

        [ForeignKey("AvatarId")]
        public virtual File Avatar { get; set; }

        [InverseProperty("User")]
        public virtual ICollection<Order> Orders { get; set; }

        #endregion

        public User()
        {
            UploadedFiles = new List<File>();
        }
    }
}
```

Order.cs

```
using RequestTTRAnalyzer.Domain.Entities.Identity;
using RequestTTRAnalyzer.Domain.Entities.Interfaces;
using RequestTTRAnalyzer.Models.Enums;
using System.ComponentModel.DataAnnotations.Schema;

namespace RequestTTRAnalyzer.Domain.Entities.Orders
```

```
{
    public class Order : IEntity<int>
    {
        #region Properties

        public int Id { get; set; }

        public int UserId { get; set; }

        public DateTime CreatedAt { get; set; }

        public DateTime? DeliveredAt { get; set; }

        public OrderStatus Status { get; set; }

        public string UserComment { get; set; }

        #endregion

        #region Navigation Properties

        [ForeignKey("UserId")]
        [InverseProperty("Orders")]
        public virtual User User { get; set; }

        [InverseProperty("Order")]
        public virtual ICollection<OrderProduct> Products { get;
set; }

        #endregion

        public Order()
        {
            Products = new List<OrderProduct>();
        }
    }
}
```

OrderProduct.cs

```
using RequestTTRAnalyzer.Domain.Entities.Interfaces;
using RequestTTRAnalyzer.Domain.Entities.Products;
using System.ComponentModel.DataAnnotations.Schema;

namespace RequestTTRAnalyzer.Domain.Entities.Orders
{
    public class OrderProduct : IEntity<int>
    {
        #region Properties

        public int Id { get; set; }

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

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

        public int Amount { get; set; }

        #endregion

        #region Navigation Properties

        [ForeignKey("OrderId")]
        [InverseProperty("Products")]
        public virtual Order Order { get; set; }

        [ForeignKey("ProductId")]
        [InverseProperty("Orders")]
        public virtual Product Product { get; set; }

        #endregion
    }
}
```

Product.cs

```
using RequestTTRAnalyzer.Domain.Entities.Interfaces;
using RequestTTRAnalyzer.Domain.Entities.Orders;
using RequestTTRAnalyzer.Models.Enums;
using System.ComponentModel.DataAnnotations.Schema;

namespace RequestTTRAnalyzer.Domain.Entities.Products
{
    public class Product : IEntity<int>
    {
        #region Properties

        public int Id { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

        public ProductCategory Category { get; set; }

        public int BrandId { get; set; }

        /// <summary>
        /// country of manufacture
        /// </summary>
        public string MadeIn { get; set; }

        public double Cost { get; set; }

        #endregion
```

```
        #region Navigation properties

        [ForeignKey("BrandId")]
        [InverseProperty("Products")]
        public virtual Brand Brand { get; set; }

        [InverseProperty("Product")]
        public virtual ICollection<OrderProduct> Orders { get;
set; }

        #endregion

        public Product()
        {
            Orders = new List<OrderProduct>();
        }
    }
}
```

Brand.cs

```
using RequestTTRAnalyzer.Domain.Entities.Files;
using RequestTTRAnalyzer.Domain.Entities.Interfaces;
using System.ComponentModel.DataAnnotations.Schema;
using File = RequestTTRAnalyzer.Domain.Entities.Files.File;

namespace RequestTTRAnalyzer.Domain.Entities.Products
{
    public class Brand : IEntity<int>
    {
        #region Properties

        public int Id { get; set; }

        public string Name { get; set; }

        /// <summary>
        /// country of brand registration
        /// </summary>
        public string Country { get; set; }

        /// <summary>
        /// Brand logo
        /// </summary>
        public int? ImageId { get;set; }

        #endregion

        #region Navigation Properties

        [InverseProperty("Brand")]
        public virtual ICollection<Product> Products { get; set; }
```

```
        [ForeignKey("ImageId")]
        public virtual File Logo { get; set; }

        #endregion

        public Brand()
        {
            Products = new List<Product>();
        }
    }
}
```

RequestLog.cs

```
using RequestTTRAnalyzer.Domain.Entities.Interfaces;
using RequestTTRAnalyzer.Models.Enums;

namespace RequestTTRAnalyzer.Domain.Entities.Research
{
    public class RequestLog : IEntity<int>
    {
        #region Properties

        public int Id { get; set; }

        public string Endpoint { get; set; }

        public long ExecutionTimeMs { get; set; }

        public DateTime ExecutedAt { get; set; }

        public int? RecordCount { get; set; }

        public int? FieldCount { get; set; }

        public long ResponseSizeBytes { get; set; }

        public string RequestMethod { get; set; }

        /// <summary>
        /// depth props
        /// </summary>
        public QueryComplexity QueryComplexity { get; set; }

        /// <summary>
        /// UseCache, CompressData, etc.
        /// It`s a bit sum of flag values
        /// </summary>
        public GlobalConfigurationFlags? ConfigFlags { get; set; }

        public bool IsSuccess { get; set; }

        #endregion
```

```
    }
}

RequestTimingMiddleware.cs
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using RequestTTRAnalyzer.DAL.Data.Interfaces;
using RequestTTRAnalyzer.Domain.Entities.Research;
using RequestTTRAnalyzer.Models.Enums;
using RequestTTRAnalyzer.Models.ResponseModels.Pagination;
using RequestTTRAnalyzer.Services.Services;
using System.Diagnostics;

namespace RequestTTRAnalyzer.Middlewares.Middlewares
{
    public class RequestTimingMiddleware
    {
        private readonly RequestDelegate _next;

        public RequestTimingMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            var unitOfWork =
context.RequestServices.GetRequiredService<IUnitOfWork>();

            // Start the stopwatch to measure execution time
            var stopwatch = Stopwatch.StartNew();

            var requestLog = new RequestLog
            {
                Endpoint = context.Request.Path,
                ExecutedAt = DateTime.UtcNow,
                RequestMethod = context.Request.Method,
            };

            var originalBodyStream = context.Response.Body;

            try
            {
                using (var memoryStream = new MemoryStream())
                {
                    // Set a new `MemoryStream` to capture the
response
                    context.Response.Body = memoryStream;

                    await _next(context);
```

```
                    // Read the captured response
                    memoryStream.Seek(0, SeekOrigin.Begin);
                    string responseBody = await new
StreamReader(memoryStream).ReadToEndAsync();

                    // Counting metrics
                    CalculateMetrics(requestLog, responseBody);

                    // Set the response size in bytes
                    requestLog.ResponseSizeBytes =
memoryStream.Length;

                    // Move the stream pointer to the beginning
and copy the contents to the source `Response.Body`
                    memoryStream.Seek(0, SeekOrigin.Begin);
                    await
memoryStream.CopyToAsync(originalBodyStream);
                }

                requestLog.IsSuccess = context.Response.StatusCode
< 400;
            }
            catch (Exception ex)
            {
                // If an exception occurs, mark request as
unsuccessful
                requestLog.IsSuccess = false;

                throw;
            }
            finally
            {
                stopwatch.Stop();

                // Gather information for logging
                requestLog.ExecutionTimeMs =
stopwatch.ElapsedMilliseconds;
                requestLog.ConfigFlags =
SettingsService.OptimizationConfig;

                // Save the log to the database

unitOfWork.Repository<RequestLog>().Add(requestLog);
                unitOfWork.SaveChanges();

                // Always set `Response.Body` back to the original
stream
                context.Response.Body = originalBodyStream;
            }
        }
```

```
        #region private

        private QueryComplexity CalculateComplexity(object item)
        {
            if (item == null)
                return QueryComplexity.Simple;

            int depth = GetDepth(item);

            if (depth > 3)
                return QueryComplexity.Complex;
            else if (depth > 1)
                return QueryComplexity.Moderate;
            else
                return QueryComplexity.Simple;
        }

        private int GetFieldCount(object item)
        {
            if (item == null)
                return 0;

            var dict =
JsonConvert.DeserializeObject<Dictionary<string,
object>>(JsonConvert.SerializeObject(item));
            return dict.Count;
        }

        private void CalculateMetrics(RequestLog requestLog,
string responseBody)
        {
            if (string.IsNullOrWhiteSpace(responseBody))
            {
                requestLog.RecordCount = 0;
                requestLog.FieldCount = 0;
                return;
            }

            var jsonResponse =
JsonConvert.DeserializeObject<PaginationResponseModel<object>>(res
ponseBody);

            // If `Data` is not `null`, this is the result of
pagination
            if (jsonResponse?.Data != null)
            {
                requestLog.RecordCount =
jsonResponse.ActualRecords;

                var firstItem =
jsonResponse.Data.FirstOrDefault();
                if (firstItem != null)
```

```
                {
                    requestLog.FieldCount =
GetFieldCount(firstItem);
                    requestLog.QueryComplexity =
CalculateComplexity(firstItem);
                }
            }
            // process the response as an ordinary object.
            else
            {
                var singleResponse =
JsonConvert.DeserializeObject<object>(responseBody);
                requestLog.RecordCount = 1;
                requestLog.FieldCount =
GetFieldCount(singleResponse);
                requestLog.QueryComplexity =
CalculateComplexity(singleResponse);
            }
        }

        private int GetDepth(object item)
        {
            if (item == null)
                return 0;

            var dict =
JsonConvert.DeserializeObject<Dictionary<string,
object>>(JsonConvert.SerializeObject(item));
            int maxDepth = 1;

            foreach (var keyValue in dict)
            {
                if (keyValue.Value is JObject nestedObject)
                {
                    var nestedDict =
nestedObject.ToObject<Dictionary<string, object>>();
                    maxDepth = Math.Max(maxDepth, 1 +
GetDepth(nestedDict));
                }
                else if (keyValue.Value is JArray array)
                {
                    foreach (var arrayItem in array)
                    {
                        if (arrayItem is JObject arrayObject)
                        {
                            var nestedDict =
arrayObject.ToObject<Dictionary<string, object>>();
                            maxDepth = Math.Max(maxDepth, 1 +
GetDepth(nestedDict));
                        }
                    }
                }
```

```
            }

            return maxDepth;
        }

        #endregion
    }
}
```

OrderService.cs

```
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.Options;
using RequestTTRAnalyzer.Common.Helpers;
using RequestTTRAnalyzer.DAL.Data.Interfaces;
using RequestTTRAnalyzer.Domain.Entities.Orders;
using RequestTTRAnalyzer.Models.Enums;
using RequestTTRAnalyzer.Models.Internal;
using RequestTTRAnalyzer.Models.RequestModels.Orders;
using RequestTTRAnalyzer.Models.RequestModels.Pagination;
using RequestTTRAnalyzer.Models.ResponseModels.Orders;
using RequestTTRAnalyzer.Models.ResponseModels.Pagination;
using RequestTTRAnalyzer.Services.Interfaces;
using System.Security.Claims;

namespace RequestTTRAnalyzer.Services.Services
{
    public class OrderService : IOrderService
    {
        private readonly IUnitOfWork _unitOfWork;
        private readonly IMapper _mapper;
        private readonly IMemoryCache _cache;
        private readonly OptimizationConfig _optimizationConfig;

        private readonly int? _userId;

        public OrderService(IUnitOfWork unitOfWork,
            IMapper mapper,
            IHttpContextAccessor httpContextAccessor,
            IMemoryCache cache,
            IOptions<OptimizationConfig> optimizationOptions)
        {
            _unitOfWork = unitOfWork;
            _mapper = mapper;
            _cache = cache;
            _optimizationConfig = optimizationOptions.Value;

            string claim =
httpContextAccessor.HttpContext.User.Claims.FirstOrDefault(w =>
w.Type == ClaimTypes.NameIdentifier)?.Value;
```

```
            _userId = Convert.ToInt32(claim);
        }

        public OrderResponseModel CreateNewOrder(OrderRequestModel
model)
        {
            var order = new Order
            {
                UserId = _userId.Value,
                CreatedAt = DateTime.UtcNow,
                Status = OrderStatus.Created,
                UserComment = model.UserComment,
                Products =
_mapper.Map<List<OrderProduct>>(model.Products)
            };

            _unitOfWork.Repository<Order>().Add(order);
            _unitOfWork.SaveChanges();

            var response = _mapper.Map<OrderResponseModel>(order);

            // Check if compression is enabled
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
                CompressOrderData(response);

            return response;
        }

        public OrderResponseModel EditOrder(int id,
OrderRequestModel model)
        {
            var order = _unitOfWork.Repository<Order>().Select(x
=> x.Id == id)
                .FirstOrDefault() ?? throw new
CustomException(404, "Order is not found");

            _mapper.Map(model, order);

            _unitOfWork.Repository<Order>().Update(order);
            _unitOfWork.SaveChanges();

            // Update cache after successful database save
(always, regardless of flags)
            string cacheKey = $"Order_{id}";
            _cache.Set(cacheKey, order, new
MemoryCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow =
TimeSpan.FromMinutes(_optimizationConfig.CacheExpirationMinutes),
```

```
                SlidingExpiration =
TimeSpan.FromMinutes(_optimizationConfig.CacheSlidingExpirationMin
utes)
            });

            var response = _mapper.Map<OrderResponseModel>(order);

            // Check if compression is enabled
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
                CompressOrderData(response);

            return response;
        }

        // Delete a order by ID
        public void DeleteOrder(int id)
        {
            var order = _unitOfWork.Repository<Order>().Select(x
=> x.Id == id)
                .FirstOrDefault() ?? throw new
CustomException(404, "Order is not found");

            _unitOfWork.Repository<Order>().Remove(order);
            _unitOfWork.SaveChanges();

            // Remove product from cache after deletion (always,
regardless of flags)
            string cacheKey = $"Order_{id}";
            _cache.Remove(cacheKey);
        }

        public OrderResponseModel GetOrderById(int id)
        {
            Order order;

            // Check if cache is enabled
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Us
eCache))
                // Retrieve order from cache
                order = GetOrderFromCacheById(id);
            else
                order = _unitOfWork.Repository<Order>().Select(x
=> x.Id == id)
                    .Include(x => x.Products)
                        .ThenInclude(x => x.Product.Brand.Logo)
                    .FirstOrDefault();

            if (order == null)
```

```
                throw new CustomException(404, "Order is not
found");

            var response = _mapper.Map<OrderResponseModel>(order);

            // Check if compression is enabled
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
                CompressOrderData(response);

            return response;
        }

        public PaginationResponseModel<OrderResponseModel>
GetOrders(PaginationRequestModel model = null)
        {
            var orders =
_unitOfWork.Repository<Order>().Select(null, selectAll: true)
                .Include(x => x.Products)
                    .ThenInclude(a => a.Product.Brand);

            var response = new
PaginationResponseModel<OrderResponseModel> { TotalRecords =
orders.Count() };

            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.En
ablePagination))
            {
                if (model == null)
                    throw new CustomException(400, "Model cannot
be null when pagination is enabled");

                response.Data =
_mapper.Map<List<OrderResponseModel>>(orders.Skip((model.PageNumbe
r - 1) * model.PageSize)
                    .Take(model.PageSize));
            }
            else
                response.Data =
_mapper.Map<List<OrderResponseModel>>(orders);

            response.ActualRecords = response.Data.Count;

            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
            {
                foreach (var product in response.Data)
                    CompressOrderData(product);
            }
```

```
            return response;
        }

        #region private

        private Order GetOrderFromCacheById(int id)
        {
            // Define the cache key based on order ID
            string cacheKey = $"Order_{id}";

            // Try to get the product from cache
            if (_cache.TryGetValue(cacheKey, out Order
cachedOrder))
                return cachedOrder;

            // If not found in cache, get the order from database
            var order = _unitOfWork.Repository<Order>().Select(x
=> x.Id == id)
                .Include(x => x.Products)
                    .ThenInclude(a => a.Product.Brand.Logo)
                .FirstOrDefault();

            if (order != null)
            {
                // Set cache options (e.g., cache for 10 minutes)
                var cacheOptions = new MemoryCacheEntryOptions
                {
                    AbsoluteExpirationRelativeToNow =
TimeSpan.FromMinutes(10),
                    SlidingExpiration = TimeSpan.FromMinutes(5)
                };

                // Store the order in cache
                _cache.Set(cacheKey, order, cacheOptions);
            }

            return order;
        }

        private void CompressOrderData(OrderResponseModel order)
        {
            if (string.IsNullOrEmpty(order.UserComment) ||
order.UserComment.Length <
_optimizationConfig.MinLengthForCompression)
                return;

            // compress data
            order.UserComment =
CompressionUtility.CompressString(order.UserComment);
        }
```

```
        #endregion
    }
}

ProductService.cs
using AutoMapper;
using Microsoft.AspNetCore.Http;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.Options;
using RequestTTRAnalyzer.Common.Helpers;
using RequestTTRAnalyzer.DAL.Data.Interfaces;
using RequestTTRAnalyzer.Domain.Entities.Products;
using RequestTTRAnalyzer.Models.Enums;
using RequestTTRAnalyzer.Models.Internal;
using RequestTTRAnalyzer.Models.RequestModels.Pagination;
using RequestTTRAnalyzer.Models.RequestModels.Products;
using RequestTTRAnalyzer.Models.ResponseModels.Pagination;
using RequestTTRAnalyzer.Models.ResponseModels.Products;
using RequestTTRAnalyzer.Services.Interfaces;
using System.Security.Claims;

namespace RequestTTRAnalyzer.Services.Services
{
    public class ProductsService : IProductsService
    {
        private readonly IUnitOfWork _unitOfWork;
        private readonly IMapper _mapper;
        private readonly IMemoryCache _cache;
        private readonly OptimizationConfig _optimizationConfig;

        private readonly int? _userId;

        public ProductsService(IUnitOfWork unitOfWork,
            IMapper mapper,
            IHttpContextAccessor httpContextAccessor,
            IMemoryCache cache,
            IOptions<OptimizationConfig> optimizationOptions)
        {
            _unitOfWork = unitOfWork;
            _mapper = mapper;
            _cache = cache;
            _optimizationConfig = optimizationOptions.Value;

            string claim =
httpContextAccessor.HttpContext.User.Claims.FirstOrDefault(w =>
w.Type == ClaimTypes.NameIdentifier)?.Value;

            _userId = Convert.ToInt32(claim);
        }
```

```
        public ProductResponseModel
CreateProduct(ProductRequestModel model)
        {
            if (model == null)
                throw new CustomException(400, "Model is
invalid");

            var product = _mapper.Map<Product>(model);

            _unitOfWork.Repository<Product>().Add(product);
            _unitOfWork.SaveChanges();

            var response =
_mapper.Map<ProductResponseModel>(product);

            // Check if compression is enabled
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
                CompressProductData(response);

            return response;
        }

        public ProductResponseModel EditProduct(int id,
ProductRequestModel model)
        {
            var product =
_unitOfWork.Repository<Product>().Select(x => x.Id == id)
                .FirstOrDefault() ?? throw new
CustomException(404, "Product is not found");

            _mapper.Map(model, product);

            _unitOfWork.Repository<Product>().Update(product);
            _unitOfWork.SaveChanges();

            // Update cache after successful database save
(always, regardless of flags)
            string cacheKey = $"Product_{id}";
            _cache.Set(cacheKey, product, new
MemoryCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow =
TimeSpan.FromMinutes(_optimizationConfig.CacheExpirationMinutes),
                SlidingExpiration =
TimeSpan.FromMinutes(_optimizationConfig.CacheSlidingExpirationMin
utes)
            });

            var response =
_mapper.Map<ProductResponseModel>(product);
```

```
            // Check if compression is enabled
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
                CompressProductData(response);

            return response;
        }

        // Delete a product by ID
        public void DeleteProduct(int id)
        {
            var product =
_unitOfWork.Repository<Product>().Select(x => x.Id == id)
                .FirstOrDefault() ?? throw new
CustomException(404, "Product is not found");

            _unitOfWork.Repository<Product>().Remove(product);
            _unitOfWork.SaveChanges();

            // Remove product from cache after deletion (always,
regardless of flags)
            string cacheKey = $"Product_{id}";
            _cache.Remove(cacheKey);
        }

        public ProductResponseModel GetProductById(int id)
        {
            Product product;

            // Check if cache is enabled
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Us
eCache))
                // Retrieve product from cache
                product = GetProductFromCacheById(id);
            else
                product =
_unitOfWork.Repository<Product>().Select(x => x.Id == id)
                    .Include(x => x.Brand.Logo)
                    .FirstOrDefault();

            if(product == null)
                throw new CustomException(404, "Product is not
found");

            var response =
_mapper.Map<ProductResponseModel>(product);

            // Check if compression is enabled
```

```
            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
                CompressProductData(response);

            return response;
        }

        public PaginationResponseModel<ProductResponseModel>
GetProducts(PaginationRequestModel model = null)
        {
            var products =
_unitOfWork.Repository<Product>().Select(null, selectAll: true)
                .Include(x => x.Brand.Logo);

            var response = new
PaginationResponseModel<ProductResponseModel> { TotalRecords =
products.Count() };

            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.En
ablePagination))
            {
                if (model == null)
                    throw new CustomException(400, "Model cannot
be null when pagination is enabled");

                response.Data =
_mapper.Map<List<ProductResponseModel>>(products.Skip((model.PageN
umber - 1) * model.PageSize)
                    .Take(model.PageSize));
            }
            else
                response.Data =
_mapper.Map<List<ProductResponseModel>>(products);

            response.ActualRecords = response.Data.Count;

            if
(SettingsService.IsOptimizationEnabled(GlobalConfigurationFlags.Co
mpressData))
            {
                foreach (var product in response.Data)
                    CompressProductData(product);
            }

            return response;
        }


        #region private
```

```
        private Product GetProductFromCacheById(int id)
        {
            // Define the cache key based on product ID
            string cacheKey = $"Product_{id}";

            // Try to get the product from cache
            if (_cache.TryGetValue(cacheKey, out Product
cachedProduct))
                return cachedProduct;

            // If not found in cache, get the product from
database
            var product =
_unitOfWork.Repository<Product>().Select(x => x.Id == id)
                .Include(x => x.Brand.Logo)
                .FirstOrDefault();

            if (product != null)
            {
                // Set cache options (e.g., cache for 10 minutes)
                var cacheOptions = new MemoryCacheEntryOptions
                {
                    AbsoluteExpirationRelativeToNow =
TimeSpan.FromMinutes(10),
                    SlidingExpiration = TimeSpan.FromMinutes(5)
                };

                // Store the product in cache
                _cache.Set(cacheKey, product, cacheOptions);
            }

            return product;
        }

        private void CompressProductData(ProductResponseModel
product)
        {
            if (string.IsNullOrEmpty(product.Description) ||
product.Description.Length <
_optimizationConfig.MinLengthForCompression)
                return;

            // compress data
            product.Description =
CompressionUtility.CompressString(product.Description);
        }

        #endregion
    }
}
```

SettingsService.cs

```
using RequestTTRAnalyzer.Models.Enums;
using RequestTTRAnalyzer.Models.Internal;
using RequestTTRAnalyzer.Models.RequestModels;
using RequestTTRAnalyzer.Services.Interfaces;

namespace RequestTTRAnalyzer.Services.Services
{
    public class SettingsService : ISettingsService
    {
        private static GlobalConfigurationFlags?
_optimizationConfig;

        public static GlobalConfigurationFlags? OptimizationConfig
{ get => _optimizationConfig; }

        public static bool
IsOptimizationEnabled(GlobalConfigurationFlags flag)
        {
            return OptimizationConfig.HasValue &&
                   OptimizationConfig.Value.HasFlag(flag);
        }

        public void
UpdateOptimizationConfig(OptimizationConfigRequestModel model)
        {
            if (model == null)
                throw new CustomException(400, "Model is
invalid");

            if(model.GlobalConfigurationFlags.HasValue)

ValidateFlag(model.GlobalConfigurationFlags.Value);

            _optimizationConfig = model.GlobalConfigurationFlags;
        }

        #region private

        public void ValidateFlag(GlobalConfigurationFlags
inputFlag)
        {
            var allValidFlags = GlobalConfigurationFlags.UseCache
|

GlobalConfigurationFlags.CompressData |

GlobalConfigurationFlags.EnablePagination;

            if ((inputFlag & ~allValidFlags) != 0)
                throw new CustomException(400, "Invalid flag
value");
```

```
        }

        #endregion
    }
}

OrdersController.cs
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using RequestTTRAnalyzer.Common.Attributes;
using RequestTTRAnalyzer.Models.RequestModels.Orders;
using RequestTTRAnalyzer.Models.RequestModels.Pagination;
using RequestTTRAnalyzer.Models.ResponseModels;
using RequestTTRAnalyzer.Models.ResponseModels.Errors;
using RequestTTRAnalyzer.Models.ResponseModels.Orders;
using RequestTTRAnalyzer.Models.ResponseModels.Pagination;
using RequestTTRAnalyzer.Services.Interfaces;

namespace RequestTTRAnalyzer.Controllers
{
    [Authorize(AuthenticationSchemes =
JwtBearerDefaults.AuthenticationScheme)]
    [ApiController]
    [Route("api/orders")]
    public class OrdersController : Controller
    {
        private readonly IOrderService _orderService;

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

        //POST api/orders
        /// <summary>
        /// Create new order
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     POST api/orders
        ///
        /// </remarks>
        /// <param name="model"> Model with new order info.
</param>
        /// <returns> 201 created order</returns>
        /// <response code="201">Returns created order</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
```

```
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpPost]
        [ProducesResponseType(typeof(OrderResponseModel), 201)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult CreateOrder([FromBody]
OrderRequestModel model)
        {
            var response = _orderService.CreateNewOrder(model);

            return Created(new Uri($"api/orders/{response.Id}",
UriKind.Relative), response);
        }

        //PUT api/orders/{id}
        /// <summary>
        /// Edit order info
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     PUT api/orders/1
        ///
        /// </remarks>
        /// <param name="model"> Model with new order info.
</param>
        /// <param name="id"> Id of order to update. </param>
        /// <returns> 200 updated order</returns>
        /// <response code="200">Returns updated order</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="404">Order is not found</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpPut("{id}")]
        [ProducesResponseType(typeof(OrderResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 404)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult EditOrder([FromRoute, ValidateId] int
id, [FromBody] OrderRequestModel model)
        {
            var response = _orderService.EditOrder(id, model);

            return Ok(response);
        }
```

```
        //DELETE api/orders/{id}
        /// <summary>
        /// Delete order
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     DELETE api/orders/1
        ///
        /// </remarks>
        /// <param name="id"> Id of order to delete. </param>
        /// <returns> 200 with message</returns>
        /// <response code="200">Returns message</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="404">Order is not found</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpDelete("{id}")]
        [ProducesResponseType(typeof(MessageResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 404)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult DeleteOrder([FromRoute, ValidateId]
int id)
        {
            _orderService.DeleteOrder(id);

            return Ok(new MessageResponseModel { Message = "Order
has been deleted successfully" });
        }

        //GET api/orders/{id}
        /// <summary>
        /// Get order info
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     GET api/orders/1
        ///
        /// </remarks>
        /// <param name="id"> Id of order to get info about.
</param>
        /// <returns> 200 with order info</returns>
        /// <response code="200">Returns order info</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
```

```
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="404">Order is not found</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpGet("{id}")]
        [ProducesResponseType(typeof(OrderResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 404)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult GetOrderById([FromRoute, ValidateId]
int id)
        {
            var response = _orderService.GetOrderById(id);

            return Ok(response);
        }

        //GET api/orders
        /// <summary>
        /// Get orders
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     GET api/orders?pageNumber=0&amp;pageSize=8
        ///
        /// </remarks>
        /// <param name="model"> Model pagination info. </param>
        /// <returns> 200 list of orders</returns>
        /// <response code="200">Returns orders</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpGet]

[ProducesResponseType(typeof(PaginationResponseModel<OrderResponse
Model>), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult GetOrders([FromQuery]
PaginationRequestModel model)
        {
            var response = _orderService.GetOrders(model);

            return Ok(response);
        }
```

```
    }
}

ProductsController.cs
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using RequestTTRAnalyzer.Common.Attributes;
using RequestTTRAnalyzer.Common.Constants;
using RequestTTRAnalyzer.Models.RequestModels.Pagination;
using RequestTTRAnalyzer.Models.RequestModels.Products;
using RequestTTRAnalyzer.Models.ResponseModels;
using RequestTTRAnalyzer.Models.ResponseModels.Errors;
using RequestTTRAnalyzer.Models.ResponseModels.Pagination;
using RequestTTRAnalyzer.Models.ResponseModels.Products;
using RequestTTRAnalyzer.Services.Interfaces;

namespace RequestTTRAnalyzer.Controllers
{
    [Authorize(AuthenticationSchemes =
JwtBearerDefaults.AuthenticationScheme/*, Policy =
AuthPolicy.RequireAdminRole*/)]
    [ApiController]
    [Route("api/products")]
    public class ProductsController : Controller
    {
        private readonly IProductsService _productsService;

        public ProductsController(IProductsService
productsService)
        {
            _productsService = productsService;
        }

        //POST api/products
        /// <summary>
        /// Create new product
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     POST api/products
        ///
        /// </remarks>
        /// <param name="model"> Model with new product info.
</param>
        /// <returns> 201 created product</returns>
        /// <response code="201">Returns created
product</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
```

```
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpPost]
        [ProducesResponseType(typeof(ProductResponseModel), 201)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult CreateProduct([FromBody]
ProductRequestModel model)
        {
            var response = _productsService.CreateProduct(model);

            return Created(new Uri($"api/products/{response.Id}",
UriKind.Relative), response);
        }

        //PUT api/products/{id}
        /// <summary>
        /// Edit product info
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     PUT api/products/1
        ///
        /// </remarks>
        /// <param name="model"> Model with new product info.
</param>
        /// <param name="id"> Id of product to update. </param>
        /// <returns> 200 updated product</returns>
        /// <response code="200">Returns updated
product</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="404">Product is not found</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpPut("{id}")]
        [ProducesResponseType(typeof(ProductResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 404)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult EditProduct([FromRoute, ValidateId]
int id, [FromBody] ProductRequestModel model)
        {
            var response = _productsService.EditProduct(id,
model);
```

```
            return Ok(response);
        }

        //DELETE api/products/{id}
        /// <summary>
        /// Delete product
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     DELETE api/products/1
        ///
        /// </remarks>
        /// <param name="id"> Id of product to delete. </param>
        /// <returns> 200 with message</returns>
        /// <response code="200">Returns message</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="404">Product is not found</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpDelete("{id}")]
        [ProducesResponseType(typeof(MessageResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 404)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult DeleteProduct([FromRoute, ValidateId]
int id)
        {
            _productsService.DeleteProduct(id);

            return Ok(new MessageResponseModel { Message =
"Product has been deleted successfully" });
        }

        //GET api/products/{id}
        /// <summary>
        /// Get product info
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     GET api/products/1
        ///
        /// </remarks>
        /// <param name="id"> Id of product to update. </param>
        /// <returns> 200 with product info</returns>
        /// <response code="200">Returns product info</response>
        /// <response code="400">Invalid request data</response>
```

```
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="404">Product is not found</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpGet("{id}")]
        [ProducesResponseType(typeof(ProductResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 404)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult GetProductById([FromRoute,
ValidateId] int id)
        {
            var response = _productsService.GetProductById(id);

            return Ok(response);
        }

        //GET api/products
        /// <summary>
        /// Get products
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     GET api/products?pageNumber=0&amp;pageSize=8
        ///
        /// </remarks>
        /// <param name="model"> Model pagination info. </param>
        /// <returns> 200 list of products</returns>
        /// <response code="200">Returns products</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpGet]

[ProducesResponseType(typeof(PaginationResponseModel<ProductRespon
seModel>), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public IActionResult GetProducts([FromQuery]
PaginationRequestModel model)
        {
            var response = _productsService.GetProducts(model);

            return Ok(response);
```

```
        }
    }
}

SessionsController.cs
using Microsoft.AspNetCore.Mvc;
using RequestTTRAnalyzer.Models.RequestModels;
using RequestTTRAnalyzer.Models.RequestModels.Auth;
using RequestTTRAnalyzer.Models.ResponseModels.Auth;
using RequestTTRAnalyzer.Models.ResponseModels.Errors;
using RequestTTRAnalyzer.Models.ResponseModels.Users;
using RequestTTRAnalyzer.Services.Interfaces.Auth;

namespace RequestTTRAnalyzer.Controllers
{
    [ApiController]
    [Route("api/sessions")]
    public class SessionsController : Controller
    {
        private readonly ISessionsService _sessionsService;

        public SessionsController(ISessionsService
sessionsService)
        {
            _sessionsService = sessionsService;
        }

        //POST api/sessions
        /// <summary>
        /// User Sign up
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     POST api/settings
        ///     {
        ///         "firstName": "user",
        ///         "lastName": "userovich",
        ///         "phoneNumber": "+111231231",
        ///         "email": "user@mail.com",
        ///         "password": "userpass",
        ///     }
        ///
        /// </remarks>
        /// <param name="model"> Model with user info. </param>
        /// <returns> 201 with created user</returns>
        /// <response code="201">Created user</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
```

```
        [HttpPost]
        [ProducesResponseType(typeof(UserResponseModel), 201)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public async Task<IActionResult> SignUp([FromBody]
SignUpRequestModel model)
        {
            var response = await _sessionsService.SignUp(model);

            return Created(new Uri($"api/users/{response.Id}",
UriKind.Relative), response);
        }

        //PUT api/sessions
        /// <summary>
        /// Sign in
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     PUT api/settings
        ///     {
        ///         "email": "user@mail.com",
        ///         "password": "userpass",
        ///     }
        ///
        /// </remarks>
        /// <param name="model"> Model with auth info. </param>
        /// <returns> 201 with user tokens</returns>
        /// <response code="201">User tokens</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpPut]
        [ProducesResponseType(typeof(TokensResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public async Task<IActionResult> SignIn([FromBody]
SignInRequestModel model)
        {
            var response = await _sessionsService.SignIn(model);

            return Ok(response);
        }
    }
}
```

SettingsController.cs

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using RequestTTRAnalyzer.Models.RequestModels;
using RequestTTRAnalyzer.Models.ResponseModels;
using RequestTTRAnalyzer.Models.ResponseModels.Errors;
using RequestTTRAnalyzer.Services.Interfaces;

namespace RequestTTRAnalyzer.Controllers
{
    [Authorize(AuthenticationSchemes =
JwtBearerDefaults.AuthenticationScheme)]
    [ApiController]
    [Route("api/settings")]
    public class SettingsController : Controller
    {
        private readonly ISettingsService _settingsService;

        public SettingsController(ISettingsService
settingsService)
        {
            _settingsService = settingsService;
        }

        //POST api/settings
        /// <summary>
        /// Update optimization config
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     POST api/settings
        ///
        /// </remarks>
        /// <param name="model"> Model with update config info.
</param>
        /// <returns> 200 with message</returns>
        /// <response code="200">Returns message</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpPost]
        [AllowAnonymous]
        [ProducesResponseType(typeof(MessageResponseModel), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
```

```
        public IActionResult UpdateOptimizationConfig([FromBody]
OptimizationConfigRequestModel model)
        {
            _settingsService.UpdateOptimizationConfig(model);

            return Ok(new MessageResponseModel{ Message =
"Optimization configuration updated" });
        }
    }
}

TestController.cs
using Microsoft.AspNetCore.Mvc;
using RequestTTRAnalyzer.Models.RequestModels.Auth;
using RequestTTRAnalyzer.Models.ResponseModels.Auth;
using RequestTTRAnalyzer.Models.ResponseModels.Errors;
using RequestTTRAnalyzer.Models.ResponseModels.Pagination;
using RequestTTRAnalyzer.Models.ResponseModels.Users;
using RequestTTRAnalyzer.Services.Interfaces.Auth;

namespace RequestTTRAnalyzer.Controllers
{
    [ApiController]
    [Route("api/test")]
    public class TestController : Controller
    {
        private readonly ISessionsService _sessionsService;

        public TestController(ISessionsService sessionsService)
        {
            _sessionsService = sessionsService;
        }

        //GET api/test
        /// <summary>
        /// Get test users tokens to make a test request
        /// </summary>
        /// <remarks>
        /// Sample Request:
        ///
        ///     GET api/test
        ///
        /// </remarks>
        /// <param name="model"> Model with tokens info. </param>
        /// <returns> 200 with received tokens</returns>
        /// <response code="200">Users tokens</response>
        /// <response code="400">Invalid request data</response>
        /// <response code="401">Invalid token</response>
        /// <response code="403">Resourse isn`t allowed</response>
        /// <response code="500">If an unhandled exception has
been occured</response>
        [HttpGet]
```

```
[ProducesResponseType(typeof(PaginationResponseModel<BareTokenResp
onseModel>), 200)]
        [ProducesResponseType(typeof(ErrorResponseModel), 400)]
        [ProducesResponseType(typeof(ErrorResponseModel), 401)]
        [ProducesResponseType(typeof(ErrorResponseModel), 403)]
        [ProducesResponseType(typeof(ErrorResponseModel), 500)]
        public async Task<IActionResult> GetTestTokens()
        {
            var response = await
_sessionsService.GetUsersTokens();

            return Ok(response);
        }
    }
}
```

RequestTTRAnalyzer::Program.cs

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using Microsoft.OpenApi.Models;
using Newtonsoft.Json.Converters;
using RequestTTRAnalyzer.Common.Attributes;
using RequestTTRAnalyzer.Common.Constants;
using RequestTTRAnalyzer.DAL;
using RequestTTRAnalyzer.DAL.Data;
using RequestTTRAnalyzer.DAL.Data.Interfaces;
using RequestTTRAnalyzer.Domain.Entities.Identity;
using RequestTTRAnalyzer.Middlewares.Middlewares;
using RequestTTRAnalyzer.Models.Internal;
using RequestTTRAnalyzer.Services.AutoMapper;
using RequestTTRAnalyzer.Services.Interfaces;
using RequestTTRAnalyzer.Services.Interfaces.Auth;
using RequestTTRAnalyzer.Services.Services;
using RequestTTRAnalyzer.Services.Services.Auth;
using System.Text;

namespace RequestTTRAnalyzer
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);


builder.Configuration.SetBasePath(AppDomain.CurrentDomain.BaseDire
ctory)
              .AddJsonFile("appsettings.json")
```

```
            .Build();

        // Add services to the container.

builder.Services.AddDbContext<ApplicationDbContext>(options =>
            options.UseMySql(

builder.Configuration.GetConnectionString("DockerConnection"),
                new MySqlServerVersion(new Version(8, 0))
        ));

        builder.Services.AddIdentity<User,
IdentityRole<int>>(options =>
        {
            options.Password.RequireDigit = false;
            options.Password.RequireLowercase = false;
            options.Password.RequireUppercase = false;
            options.Password.RequiredLength = 8;
            options.User.RequireUniqueEmail = true;
        })
       .AddEntityFrameworkStores<ApplicationDbContext>()
       .AddDefaultTokenProviders();

        builder.Services.AddHttpContextAccessor();
        builder.Services.AddSingleton<IConfiguration>(provider
=> builder.Configuration);
        builder.Services.AddScoped<DbContext>(serviceProvider
=>
        {
            var dbContextOptions =
serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbC
ontext>>();
            return new ApplicationDbContext(dbContextOptions);
        });


builder.Services.AddAutoMapper(typeof(MapperProfile).Assembly);
        builder.Services.AddScoped<ITokenService,
TokenService>();
        builder.Services.AddScoped<IAesEncryptionService,
AesEncryptionService>();
        builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
        builder.Services.AddScoped<ISessionsService,
SessionsService>();
        builder.Services.AddScoped<ISettingsService,
SettingsService>();
        builder.Services.AddScoped<IProductsService,
ProductsService>();
        builder.Services.AddScoped<IOrderService,
OrderService>();

        builder.Services.AddCors(options =>
```

```
            {
                options.AddPolicy("AllowAll",
                    builder =>
                    {
                        builder.AllowAnyOrigin()
                               .AllowAnyMethod()
                               .AllowAnyHeader();
                    });
            });

            var serviceProvider =
builder.Services.BuildServiceProvider();
            var serviceScopeFactory =
serviceProvider.GetRequiredService<IServiceScopeFactory>();

            var jwtSettings =
builder.Configuration.GetSection("JWT");


builder.Services.AddAuthentication(JwtBearerDefaults.Authenticatio
nScheme)
            .AddJwtBearer(options =>
            {
                options.TokenValidationParameters = new
TokenValidationParameters
                {
                    ValidateIssuer = true,
                    ValidateAudience = true,
                    ValidateLifetime = true,
                    ValidateIssuerSigningKey = true,
                    ValidIssuer = jwtSettings["Issuer"],
                    ValidAudience = jwtSettings["Audience"],
                    IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtSettings["Key"])),
                };
            });

            builder.Services.AddAuthorization(options =>
            {
                options.AddPolicy(AuthPolicy.RequireAdminRole,
policy => policy.RequireRole(UserRole.Admin));
                options.AddPolicy(AuthPolicy.RequireUserRole,
policy => policy.RequireRole(UserRole.User));
                options.AddPolicy(AuthPolicy.RequireAdminUser,
policy => policy.RequireRole(UserRole.AdminUser));
            });

            builder.Services.AddControllers()
                .AddNewtonsoftJson(options =>
                {
                    options.SerializerSettings.Converters.Add(new
StringEnumConverter());
```

```
            });
        // Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen(c =>
        {
            var xmlFile =
$"{System.Reflection.Assembly.GetExecutingAssembly().GetName().Nam
e}.xml";
            var xmlPath =
Path.Combine(AppContext.BaseDirectory, xmlFile);
            c.IncludeXmlComments(xmlPath);


c.AddSecurityDefinition(JwtBearerDefaults.AuthenticationScheme,
new OpenApiSecurityScheme
            {
                In = ParameterLocation.Header,
                Description = "Enter JWT token 'Bearer
{token}'",
                Name = "Authorization",
                Type = SecuritySchemeType.ApiKey
            });


c.OperationFilter<SecurityRequirementsOperationFilter>();
        });

        builder.Services.AddMemoryCache();

builder.Services.Configure<OptimizationConfig>(builder.Configurati
on.GetSection("OptimizationConfig"));

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        //if (app.Environment.IsDevelopment())
        //{
            app.UseSwagger();
            app.UseSwaggerUI();
        //}

        #region middlewares

        app.UseMiddleware<ExceptionMiddleware>();
        app.UseMiddleware<RequestTimingMiddleware>();

        #endregion

        //app.UseHttpsRedirection();

        app.UseRouting();
```

```
            app.UseCors("AllowAll");

            app.UseAuthentication();
            app.UseAuthorization();

            app.MapControllers();

            // Init db roles and users
            using (var scope = app.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                var userManager =
services.GetRequiredService<UserManager<User>>();
                DbInitializer.Initialize(services,
userManager).Wait();
            }

            app.Run();
        }
    }
}
```

RequestTTRAnalyzer.RequestEmulator::RequestConstants

```
using RequestTTRAnalyzer.Models.Enums;

namespace RequestTTRAnalyzer.RequestEmulator.Constants
{
    public class RequestConstants
    {
        // base api url
        //public const string BASE_API_URL =
"http://localhost:53497";
        public const string BASE_API_URL =
"http://localhost:5000";

        // url to get tokens list to perform a request for
research
        public const string TOKEN_REQUEST_URL = "api/test";

        // url to change state of global configuration flags
        public const string CHANGE_FLAGS_URL = "api/settings";

        // 8 possible combinations of optimization flag
        public static readonly List<GlobalConfigurationFlags?>
CombinationsOfParameters = new ()
        {
            null,
// params are not set
            GlobalConfigurationFlags.UseCache,
// only cashing
```

```
            GlobalConfigurationFlags.EnablePagination,
// only pagination
            GlobalConfigurationFlags.CompressData,
// only compression
            GlobalConfigurationFlags.UseCache |
GlobalConfigurationFlags.CompressData,          // caching and
compression
            GlobalConfigurationFlags.UseCache |
GlobalConfigurationFlags.EnablePagination,      // caching and
pagination
            GlobalConfigurationFlags.CompressData |
GlobalConfigurationFlags.EnablePagination,  // compressing and
pagination
            GlobalConfigurationFlags.EnablePagination |
GlobalConfigurationFlags.CompressData |
GlobalConfigurationFlags.UseCache // all parameters
        };

        public static readonly List<RequestRoute> ApiRoutes =
new()
        {
            // Products controller api methods
            new RequestRoute("api/products", HttpMethod.Post),
            new
RequestRoute("api/products?pageNumber={0}&amp;pageSize={1}",
HttpMethod.Get),
            new RequestRoute("api/products/{0}", HttpMethod.Put),
            new RequestRoute("api/products/{0}", HttpMethod.Get),

            // Orders controller api methods
            new RequestRoute("api/orders", HttpMethod.Post),
            new
RequestRoute("api/orders?pageNumber={0}&amp;pageSize={1}",
HttpMethod.Get),
            new RequestRoute("api/orders/{0}", HttpMethod.Put),
            new RequestRoute("api/orders/{0}", HttpMethod.Get),
            new RequestRoute("api/orders/{0}", HttpMethod.Delete),

            new RequestRoute("api/products/{0}",
HttpMethod.Delete),
        };
    }

    public class RequestRoute
    {
        public string Url { get; set; }

        public HttpMethod Method { get; set; }

        public RequestRoute(string url, HttpMethod method)
        {
            Url = url;
```

```
            Method = method;
        }
    }
}

RequestTTRAnalyzer.RequestEmulator::Program.cs
using Newtonsoft.Json;
using RequestTTRAnalyzer.Models.Internal;
using RequestTTRAnalyzer.Models.RequestModels;
using RequestTTRAnalyzer.Models.RequestModels.Orders;
using RequestTTRAnalyzer.Models.RequestModels.Products;
using RequestTTRAnalyzer.Models.ResponseModels.Auth;
using RequestTTRAnalyzer.Models.ResponseModels.Pagination;
using RequestTTRAnalyzer.RequestEmulator.Constants;
using System.Net.Http.Headers;
using System.Text;

namespace RequestTTRAnalyzer.RequestEmulator
{
    internal class Program
    {
        private const int COUNT_OF_REQUESTS = 30; //TODO: After
testing change to 30 for research
        private const int BASE_PAGINATION_PAGE = 1;
        private const int BASE_PAGINATION_LIMIT = 8;
        private const int CURRENT_MAX_PAGE = 4;

        static async Task Main(string[] args)
        {
            Console.WriteLine("#INFO_Hit any key to starty
processing\n");
            Console.ReadKey();

            //for local usage
            string jsonFilePath =
"../../../../test_data_varying.json";

            //for docker usage
            //string jsonFilePath = "/app/test_data_varying.json";

            #region Extract test data from file

            var jsonData = File.ReadAllText(jsonFilePath);

            var rootData =
JsonConvert.DeserializeObject<RootTestData>(jsonData);

            if(rootData == null)
            {
                Console.WriteLine("#ERROR_Unable to extract data
from file\n");
                return;
```

```
            }

            var products = rootData.Products;
            var orders = rootData.Orders;

            #endregion

            #region get tokens

            var tokens = (await
SendApiRequestAsync<PaginationResponseModel<BareTokenResponseModel
>>(RequestConstants.TOKEN_REQUEST_URL,
                    HttpMethod.Get, noReturn: false))
                .Data
                .Select(x => x.Token)
                .ToList();

            #endregion

            #region requests sending

            // for tokens, to send request as random authorized
user
            Random random = new Random();
            int index = 0;

            foreach (var combination in
RequestConstants.CombinationsOfParameters)
            {
                // chage optimization config to iterated
combination
                await SendApiRequestAsync<object>(
                    RequestConstants.CHANGE_FLAGS_URL,
                    HttpMethod.Post,
                    new OptimizationConfigRequestModel {
GlobalConfigurationFlags = combination}
                    );

                // smth like log
                Console.WriteLine($"#INFO_Updated flags to:
{combination}");

                foreach (var route in RequestConstants.ApiRoutes)
                {
                    var isProductRequest =
route.Url.Contains("products");

                    for (int i = 0; i < COUNT_OF_REQUESTS; i++)
                    {
                        int randomIndex =
random.Next(tokens.Count);
                        string randomToken = tokens[randomIndex];
```

```
                        int? routeParam = default;
                        (int page, int limit) pagination =
(BASE_PAGINATION_PAGE, BASE_PAGINATION_LIMIT);
                        ProductRequestModel productBody = default;
                        OrderRequestModel orderBody = default;

                        // for routes with route param and delete
method, to prevent random deleting
                        if (route.Url.Contains('{') &&
!route.Url.Contains('?'))
                        {
                            if (route.Method == HttpMethod.Delete)
                                routeParam = index + i + 1;
                            else
                                routeParam = index > i ?
random.Next(index + 1, index + COUNT_OF_REQUESTS + 1) :
random.Next(tokens.Count); // random value for other requests
                        }
                        if (route.Url.Contains('?'))
                            pagination = (random.Next(1,
CURRENT_MAX_PAGE + 1), random.Next(1, BASE_PAGINATION_LIMIT + 1));
// get randow values for pagination (in allowed range)

                        if(route.Method == HttpMethod.Post)
                        {
                            if (isProductRequest)
                            {
                                var randomDataIndex =
random.Next(products.Count);
                                productBody =
products[randomDataIndex];
                            }
                            else
                            {
                                var randomDataIndex =
random.Next(orders.Count);
                                orderBody =
orders[randomDataIndex];

                                if(index > i)
                                {
                                    var newBody = new
OrderRequestModel();

                                    newBody.UserComment =
orderBody.UserComment;
                                    newBody.Products =
orderBody.Products.Select(x => new OrderProductRequestModel {
ProductId = x.ProductId + index, Amount = x.Amount })
                                        .ToList();

                                    orderBody = newBody;
```

```
                                }
                            }
                        }
                        if (route.Method == HttpMethod.Put)
                        {
                            if (isProductRequest)
                                productBody =
products[COUNT_OF_REQUESTS - 1 - i];
                            else
                            {
                                orderBody =
orders[COUNT_OF_REQUESTS - 1 - i];

                                if (index > i)
                                {
                                    var newBody = new
OrderRequestModel();

                                    newBody.UserComment =
orderBody.UserComment;
                                    newBody.Products =
orderBody.Products.Select(x => new OrderProductRequestModel {
ProductId = x.ProductId + index, Amount = x.Amount })
                                        .ToList();

                                    orderBody = newBody;
                                }
                            }
                        }

                        await SendApiRequestAsync<object>(
                            route.Url,
                            route.Method,
                            body: isProductRequest ? productBody :
orderBody,
                            noReturn: true,
                            bearerToken: randomToken,
                            urlParams: routeParam.HasValue ?
[routeParam.Value] : [pagination.page, pagination.limit]);
                    }
                }

                index += COUNT_OF_REQUESTS;
            }

            // finish log
            Console.WriteLine("#INFO_FINISH_All tasks are
done\n");

            #endregion
        }
```

```
        public static async Task<T> SendApiRequestAsync<T>(string
routeTemplate, HttpMethod method, object body = null, string
bearerToken = null, bool noReturn = true, params object[]
urlParams)
        {
            string url = string.Format(routeTemplate, urlParams);

            url = RequestConstants.BASE_API_URL + '/' + url;

            var request = new HttpRequestMessage(method, url);

            if (body != null && (method == HttpMethod.Post ||
method == HttpMethod.Put))
            {
                var json = JsonConvert.SerializeObject(body);
                request.Content = new StringContent(json,
Encoding.UTF8, "application/json");
            }

            if (!string.IsNullOrEmpty(bearerToken))
                request.Headers.Authorization = new
AuthenticationHeaderValue("Bearer", bearerToken);

            using var httpClient = new HttpClient();
            var response = await httpClient.SendAsync(request);

            if (!response.IsSuccessStatusCode || noReturn)
                return default;

            var responseContent = await
response.Content.ReadAsStringAsync();

            T? result =
JsonConvert.DeserializeObject<T>(responseContent);

            return result;
        }
    }
}

docker-compose.yaml
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    container_name: request-ttr-analyzer-db
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=ResearchDB
    ports:
      - "3306:3306"
    volumes:
```

```
      - mysql_data:/var/lib/mysql
    networks:
      - app-network

  backend:
    build:
      context: .
      dockerfile: RequestTTRAnalyzer/Dockerfile
    container_name: request-ttr-analyzer-backend
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ConnectionStrings__DefaultConnection=Server=request-ttr-
analyzer-db;Database=ResearchDB;User=root;Password=root;
    ports:
      - "5000:8080"
    depends_on:
      - mysql
    networks:
      - app-network
    deploy:
      resources:
        limits:
          cpus: "0.5"
          memory: "512m"

volumes:
  mysql_data:

networks:
  app-network:
    driver: bridge

RequestTTRAnalyzer::Dockerfile
#
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 5000
EXPOSE 5001

#
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src

#
COPY ["RequestTTRAnalyzer/RequestTTRAnalyzer.csproj",
"RequestTTRAnalyzer/"]
RUN dotnet restore "RequestTTRAnalyzer/RequestTTRAnalyzer.csproj"

#
COPY . .
WORKDIR "/src/RequestTTRAnalyzer"
RUN dotnet publish -c Release -o /app/publish
```

```
#
FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "RequestTTRAnalyzer.dll"]
```

ДОДАТОК В

Тези доповіді для конференції

ЗАТВЕРДЖЕНО
1116130.1431-01 90 01-ЛЗ

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

1116130.1431-01 90 01-ЛЗ

Листів 5

2025

ЗМІСТ

В.1 ТЕЗИ ДОПОВІДІ НА КОНФЕРЕНЦІЇ

Міністерство освіти і науки України

Український державний університет науки і технологій

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

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

12.12.2024 – 13.12.2024

Дніпро
2024

Дослідження способів зменшення часу виконання запитів у RESTful API

Подедворний О.Е., Андрющенко В.О.,
Український державний університет науки і технологій, Україна

Дослідження присвячене вивченню способів зменшення часу виконання запитів у RESTful API на прикладі монолітного додатку, розробленого на базі ASP.NET Core з використанням Onion-архітектури. Сучасні веб-додатки часто стикаються з проблемами продуктивності, особливо при обробці великої кількості запитів та передачі значних обсягів даних. У цьому контексті важливим є пошук ефективних способів прискорення API, що сприятиме покращенню швидкості обробки запитів та зменшенню розміру респонсу.

Основною метою роботи є розробка інструментів для збору та аналізу метрик продуктивності запитів з можливістю оцінити ефективність різних способів прискорення, таких як кешування, стиснення даних (Gzip) та пагінація. Було розроблено програмний продукт, який включає глобальний сервіс конфігурації параметрів прискорення, дозволяючи активувати чи деактивувати певні способи прискорення для запитів RESTful API. У межах цієї архітектури виділяється сервіс роботи з товарами – ProductService, який виконує основні CRUD операції та застосовує обрані способи прискорення на рівні кожного запиту.

Для збору даних про час виконання запитів, розмір респонсу та інші метрики було впроваджено middleware, що дозволяє перехоплювати HTTP-запити, запускати таймер на час виконання та відслідковувати інші параметри, такі як кількість оброблених записів, кількість полів у респонсі та складність запиту. Отримані метрики зберігаються у базі даних MySQL та аналізуються для оцінки ефективності застосованих способів.

У ході дослідження було виявлено, що використання способів прискорення, таких як кешування та компресія, суттєво зменшує час виконання запитів та розмір переданих даних. Пагінація дозволяє обмежити обсяг переданих записів, що також позитивно впливає на швидкість роботи API. Проведені експерименти показали, що сукупне застосування цих методів значно покращує продуктивність монолітного додатку.

Отримані результати можуть бути використані для подальшого розвитку програмних систем, де швидкість виконання запитів є критичною, а також для реалізації більш ефективної взаємодії між сервером та клієнтом. Запропоновані підходи можуть бути адаптовані для інших систем та вдосконалені в контексті їхнього впровадження у великих проектах.