ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Удосконалення методики взаємодії мікросервісів з використанням gRPC»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Владисав ГОНЧАР
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63

Владислав ГОНЧАР

Керівник: Максим КУКЛІНСЬКИЙ
канд. техн. наук, доц.

Рецензент:
ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти Магістр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гончару Владиславу Віталійовичу

1. Тема кваліфікаційної роботи: «Удосконалення методки взаємодії мікросервісів з використанням gRPC»

керівник кваліфікаційної роботи Максим КУКЛІНСЬКИЙ, канд. техн. наук, доцент, професор кафедри ІТ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, архітектурне рішення, удосконалений метод взаємодії мікросервісів, аналіз мережевої взаємомодії мережевих протоколів

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження методик мережевої взаємодії мікросервісів.
2. Проектування архітектури та розробика системи взаємодії мікросервісів з використання gRPC.
3.Експериментальне дослідження та порівняльний аналіз взаємодії мережевих протоколів

5. Перелік ілюстративного матеріалу: *презентація*
   1. Проблематика традиційної HTTP взаємодії.
   2. Схема взаємодії клієнт-сервера з API-шлюзом
   3. Обґрунтування використання gRPC в API-шлюзі
   4. Математична модель мережевого трафіку
   5. Діаграма послідовності сценаріїв використання системи
   6. Порівняльна таблиця методів взаємодії API-шлюзу з кінцевим мікросервісом
   7. Приклад логування на стороні API-шлюзу

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	31.10-05.11.2025	
2	Вивчення матеріалів для аналізу методик взаємодії мікросервісів	06.11-11.11.2025	
3	Дослідження мережевих протоколів	12.11-19.11.2025	
4	Аналіз систем аналогів API шлюзів	20.11-25.11.2025	
5	Дослідження архітектурних рішень при побудові мікросервісів	26.11-02.12.2025	
6	Застосування протоколу gRPC для внутрішньої взаємодії мікросервісів	03.12-08.12.2025	
7	Оформлення роботи: вступ, висновки, реферат	09.12-14.12.2025	
8	Розробка демонстраційних матеріалів	15.12-19.12.2025	

Здобувач вищої освіти _____ Владислав ГОНЧАР
(підпис)

Керівник
кваліфікаційної роботи _____ Максим КУКЛІНСЬКИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 76 стор., 2 табл., 12 рис., 22 джерел.

Мета роботи – удосконалення методики взаємодії мікросервісів за рахунок впровадження API-шлюзу з підтримкою протоколу gRPC.

Об'єкт дослідження – процес взаємодії мікросервісів у розподіленій програмній системі.

Предмет дослідження – методи та програмні засоби взаємодії мікросервісів через API-шлюз з використанням протоколу gRPC.

У магістерській роботі розглянуто проблему підвищення ефективності взаємодії мікросервісів у розподілених програмних системах. Проаналізовано існуючі підходи до організації комунікації між мікросервісами, зокрема на основі REST та gRPC, а також роль API-шлюзу (gateway) у централізованому керуванні запитами. Визначено недоліки традиційних рішень, пов'язані з продуктивністю, затримкою (latency) та масштабованістю систем.

Запропоновано підхід до вдосконалення системи взаємодії мікросервісів шляхом розробки та впровадження API-шлюзу з підтримкою протоколу gRPC. Розроблено архітектуру розподіленої програмної системи з використанням gateway як єдиної точки входу для клієнтських запитів та маршрутизації до мікросервісів. Реалізовано прототип API-шлюзу з підтримкою gRPC, що забезпечує ефективну взаємодію між мікросервісами.

Проведено експериментальне дослідження продуктивності розробленої системи, у ході якого здійснено вимірювання часу відповіді, затримки та

навантаження системи за різних сценаріїв роботи. Виконано порівняльний аналіз отриманих результатів для взаємодії на основі gRPC та традиційних підходів. Показано, що використання API-шлюзу з підтримкою gRPC дає змогу зменшити затримку та підвищити продуктивність і надійність системи за умов зростання навантаження.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСНА АРХІТЕКТУРА, API-ШЛЮЗ, GATEWAY, GRPC, ВЗАЄМОДІЯ МІКРОСЕРВІСІВ, ПРОДУКТИВНІСТЬ, НАДІЙНІСТЬ, РОЗПОДІЛЕНІ СИСТЕМИ, НАВАНТАЖЕННЯ СИСТЕМИ, ЗАТРИМКА.

ABSTRACT

Text part of the master's qualification work: 76 pages, 8 pictures, 1 table, 37 sources.

The purpose of the work is to improve the microservices interaction system by developing and implementing an API gateway with support for the gRPC protocol and assessing its impact on the performance and reliability of the system.

The object of the study is the process of microservices interaction in a distributed software system through an API gateway using the gRPC protocol.

The subject of the study is methods and software tools for improving microservices interaction through an API gateway using the gRPC protocol.

The master's thesis considers the problem of increasing the efficiency of microservices interaction in distributed software systems. Existing approaches to organizing communication between microservices, in particular based on REST and gRPC, as well as the role of the API gateway in centralized request management are analyzed. The shortcomings of traditional solutions related to performance, latency and scalability of systems are identified.

An approach to improving the microservices interaction system by developing and implementing an API gateway with support for the gRPC protocol is proposed. The architecture of a distributed software system using a gateway as a single entry point for client requests and routing to microservices has been developed. A prototype of a gRPC-enabled gateway API has been implemented, which ensures effective interaction between microservices.

An experimental study of the performance of the developed system has been conducted, during which response time, latency, and system load were measured under various operating scenarios. A comparative analysis of the results obtained for interaction based on gRPC and traditional approaches has been performed. It has been shown that the use of a gRPC-enabled gateway API allows to reduce latency and increase system performance and reliability under conditions of increasing load.

KEYWORDS: MICROSERVICE ARCHITECTURE, API GATEWAY, GATEWAY, GRPC, MICROSERVICE INTERACTION, PERFORMANCE, RELIABILITY, DISTRIBUTED SYSTEMS, SYSTEM LOAD, DELAY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (прикладний програмний інтерфейс)

JSON – JavaScript Object Notation (текстовий формат обміну данними)

REST – Representational State Transfer (архітектурний стиль для розподілених систем)

UML – Unified Modeling Language (уніфікована мова моделювання)

gRPC – Google Remote Procedure Call (фреймворк для виклику віддалених процедур)

ВСТУП

Останніми роками мікросервісна архітектура стала одним з основних підходів до побудови розподілених програмних систем. Багато сучасних сервісів, орієнтованих на велику кількість користувачів та безперервну роботу, будуються саме як набір окремих мікросервісів, які взаємодіють один з одним через мережу.

На практиці для обміну даними між мікросервісами досить часто застосовуються традиційні HTTP-інтерфейси на основі REST. Попри простоту та широке розповсюдження, такі рішення не завжди дозволяють досягти прийнятних показників продуктивності за умов зростання навантаження, збільшення кількості сервісів та ускладнення їх взаємозв’язків. Зокрема, виникають проблеми, пов’язані з затримкою (latency), надмірними мережевими витратами та ускладненням централізованого керування потоками запитів.

Одним із поширених підходів до впорядкування взаємодії між мікросервісами є використання API-шлюзу (gateway), який виступає єдиною точкою входу для клієнтів і відповідає за маршрутизацію, агрегацію, фільтрацію запитів, а також може реалізовувати додаткові функції безпеки й моніторингу.

Цей диплом спрямований на вирішення проблем у області взаємодії розподілених систем, що реально впливає на роботу мікросервісної архітектури, що має велике значення для сучасних проблем швидкодії та стабільності різноманітних систем.

Тому задача вдосконалення системи взаємодії мікросервісів шляхом розробки та оцінки API-шлюзу з підтримкою gRPC є актуальною, оскільки відповідає

сучасним тенденціям розвитку розподілених програмних систем та спрямована на підвищення їх продуктивності й надійності.

Мета роботи – вдосконалення системи взаємодії мікросервісів шляхом розробки та впровадження API-шлюзу з підтримкою протоколу gRPC та оцінки його впливу на продуктивність і надійність системи.

Об’єкт дослідження – процес взаємодії мікросервісів у розподіленій програмній системі через API-шлюз (gateway) з використанням протоколу gRPC.

Предмет дослідження – методи та програмні засоби вдосконалення взаємодії мікросервісів через API-шлюз з використанням протоколу gRPC.

Для досягнення мети вирішувалися наступні завдання:

1. Аналітичний етап. Проаналізовано існуючі підходи до побудови мікросервісних систем та організації взаємодії між сервісами та досліджена роль API-шлюзу в розподілених програмних системах та типові сценарії його використання.
2. Проєктування системи. Розроблено архітектуру API-шлюзу як єдиної точки входу для клієтських запитів і взаємодії компонентів розподіленої системи, також визначені основні сценарії взаємодії, формати повідомлень для подальшої реалізації.
3. Реалізація прототипу. Реалізовано прототип API-шлюзу з підтримкою протоколу gRPC для взаємодії компонентів розподіленої системи та налаштовано інфраструктуру для запуску та тестування системи.
4. Експериментальне дослідження та оцінка результатів. Використано статистичні методи для оцінки та порівняння результатів та стабільності різних сценаріїв використання розробленого API-шлюзу
5. Порівняльний аналіз. Проведено порівняльний аналіз систем-аналогів, їхньої швидкодії та стабільності при великій завантаженості системи.

Вирішення цих задач сприяє подальшому аналізу, розвитку різних підходів у дослідженні та розробці API-шлюзів для взаємодії розподілених систем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Мікросервісна архітектура

Мікросервісна архітектура – це підхід до побудови програмних систем, за якого застосунок розбивається на сукупність невеликих, відносно незалежних сервісів, що взаємодіють між собою за допомогою чітко визначених інтерфейсів. Кожен мікросервіс відповідає за окрему бізнес-функцію та може розроблятися, розгортатися й масштабуватися автономно від інших компонентів системи.

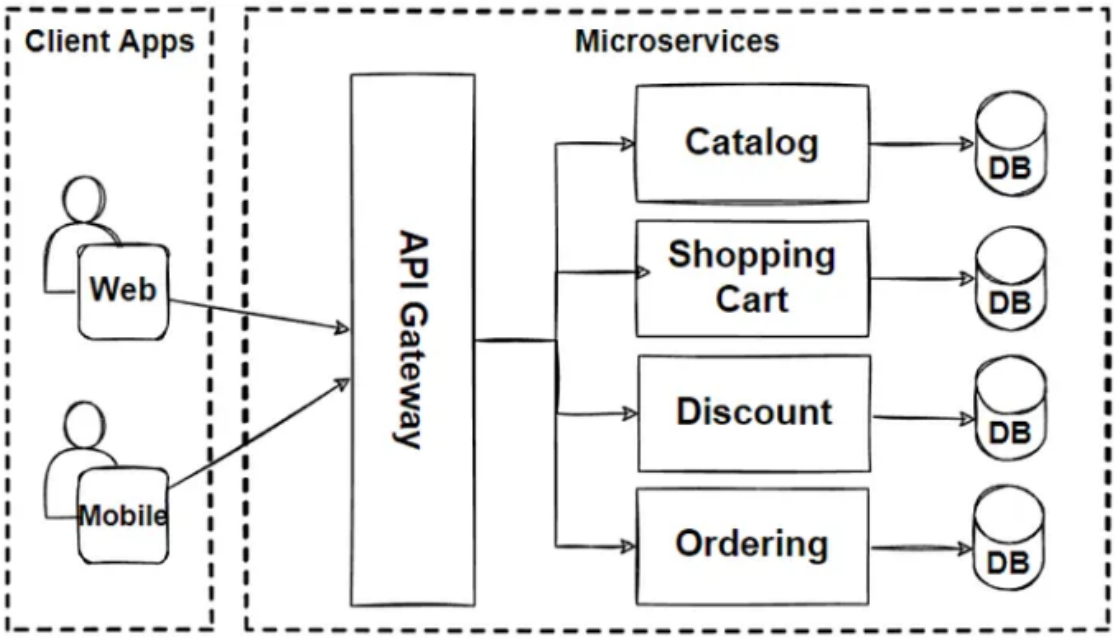


Рис. 1.1 Приклад мікросервісної архітектури

У контексті сучасних розподілених систем мікросервіси стали відповіддю на низку обмежень монолітної архітектури. Коли вся логіка зосереджена в одному застосунку, будь-яка зміна вимагає повторного розгортання всього рішення, а масштабування відбувається грубо – шляхом дублювання всього моноліту, навіть якщо навантаження зростає лише на окремий функціональний модуль. Мікросервіси, навпаки, дозволяють виділяти критичні з погляду навантаження чи бізнес-цінності компоненти та працювати з ними окремо.

Типовим прикладом є домени «Користувачі» та «Замовлення». Замість того, щоб реалізовувати їх у одному великому застосунку, мікросервісна архітектура передбачає створення окремого сервісу управління користувачами (реєстрація, автентифікація, профілі) та окремого сервісу роботи із замовленнями (створення замовлень, зміна статусу, історія, розрахунок вартості тощо). Ці сервіси можуть мати різні бази даних, різні внутрішні моделі даних та навіть реалізовуватися різними командами, але при цьому узгоджено взаємодіють через мережеві протоколи.

Мікросервісний підхід особливо актуальний для систем, що:

1. Розраховані на значну кількість одночасних користувачів та великий обсяг операцій;
2. Активно розвиваються, потребують частих релізів окремих частин функціоналу;
3. Мають чітко виражені бізнес-домени (наприклад, робота з користувачами, замовленнями, оплатами, аналітикою), які логічно відокремити один від одного.

Важливою рисою мікросервісної архітектури є незалежний життєвий цикл сервісів. Кожен мікросервіс можна оновити чи перемасштабувати без зупинки всієї

системи. Наприклад, якщо обробка замовлень у певні періоди (пік розпродажів) створює додаткове навантаження, можна збільшити кількість екземплярів саме сервісу «Замовлення», не чіпаючи сервіс «Користувачі».

Разом із перевагами мікросервіси привносять і низку складностей. Система, що складається з десятків або сотень сервісів, стає суттєво важче для спостереження, налагодження та забезпечення узгодженості даних. Замість виклику локального методу розробник працює з мережею: з’являються затримки, можливі збої окремих вузлів, частково успішні операції, що потребують додаткових шаблонів (saga, компенсаційні транзакції тощо). Це робить питання організації взаємодії мікросервісів центральним для всієї архітектури.

Саме тому в межах цієї роботи основний акцент робиться не просто на поділі системи на сервіси, а на удосконаленні методики їхньої взаємодії. Далі буде розглянуто, якими способами мікросервіси можуть обмінюватися даними між собою, які обмеження мають традиційні підходи на основі REST та які можливості відкриває використання gRPC у зв’язці з API Gateway як єдиною точкою входу для зовнішніх REST-клієнтів.

1.2 Методи взаємодії мікросервісів

Від того, як саме мікросервіси взаємодіють між собою, напряму залежать продуктивність системи, її надійність та можливість подальшого розвитку. На відміну від моноліту, де більшість викликів є внутрішніми методами в межах одного процесу, у мікросервісній архітектурі будь-яка бізнес-операція часто потребує мережевої комунікації між кількома окремими сервісами. Це робить вибір механізмів взаємодії одним із ключових архітектурних рішень.

У загальному випадку можна виділити дві великі групи підходів:

1. Синхронна взаємодія (коли клієнт очікує безпосередньої відповіді від іншого сервісу);
2. Асинхронна взаємодія (коли обмін відбувається через події, черги чи повідомлення без миттєвої відповіді).

У типовій системі, що працює з користувачами та замовленнями, синхронні виклики використовуються, наприклад, коли сервіс «Замовлення» під час створення нового замовлення повинен перевірити існування та статус користувача в сервісі «Користувачі». Асинхронна взаємодія може застосовуватися, коли після створення замовлення необхідно повідомити інші підсистеми (аналітику, нотифікації, білінг), але немає потреби чекати завершення всіх цих дій до повернення відповіді клієнту.

1.2.1 Синхронна взаємодія

Найбільш розповсюдженим способом синхронної взаємодії мікросервісів є REST-API поверх HTTP. REST набув популярності завдяки простоті, використанню звичайних HTTP-методів (GET, POST, PUT, DELETE) та легкості інтеграції з широким спектром клієнтів, зокрема браузерів та мобільних застосунків. Обмін даними зазвичай здійснюється у форматі JSON, що є людиночитабельним і зручним для налагодження.

У внутрішній комунікації мікросервісів REST також широко застосовується, однак при цьому виявляються його обмеження:

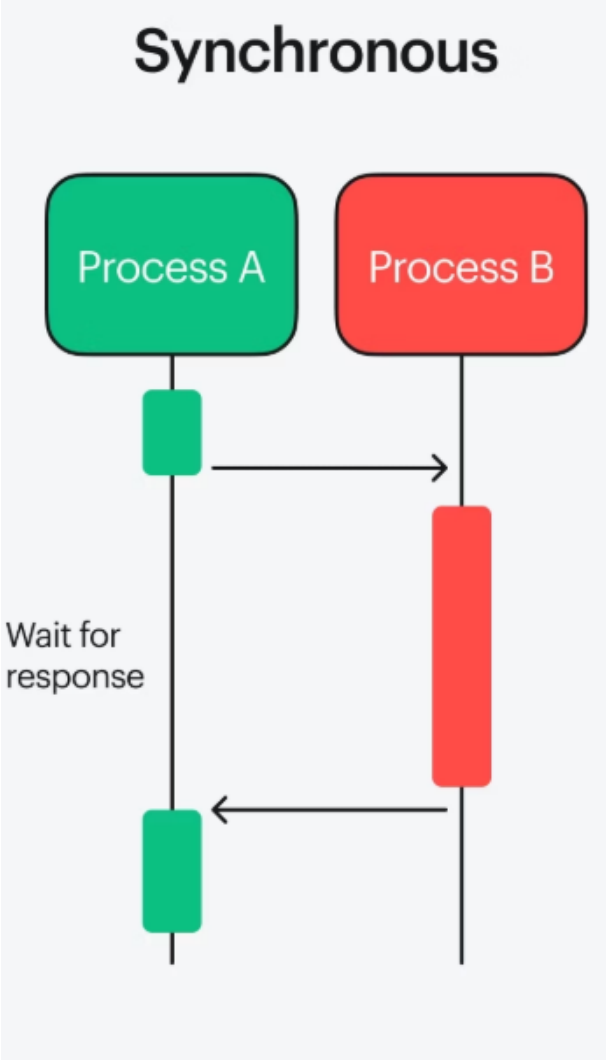


Рис. 1.2 Приклад синхронної взаємодії

1. Надмірність даних та накладні витрати через текстові формати (JSON, XML);
2. Відсутність строгого контракту на рівні типів – помилки сумісності виявляються часто лише на етапі виконання;
3. Збільшення затримок при великій кількості дрібних (“chatty”) запитів між сервісами.

Альтернативою є різноманітні RPC-підходи (Remote Procedure Call), коли сервіси спілкуються у більш “бінарний” і структурований спосіб, а мережевий виклик концептуально нагадує виклик методу. До цієї групи належать як старіші рішення (на кшталт SOAP чи власних бінарних протоколів), так і сучасні — серед яких окреме місце займає gRPC, що буде детально розглянуто в наступному підрозділі.

RPC-підходи, як правило, забезпечують:

1. Більш компактне представлення даних (бінарні формати замість текстових);
2. Строгі контракти (опис сервісів та повідомлень у вигляді схем);
3. Автоматичну генерацію клієнтів і серверів для різних мов програмування.

Водночас вони можуть ускладнювати інтеграцію з зовнішніми системами, які очікують звичний REST/HTTP+JSON, та вимагати додаткових інструментів для спостережуваності й дебагу.

1.2.2 Асинхронна взаємодія

У багатьох сценаріях пряме запит-відповідь між мікросервісами не є обов’язковим. Наприклад, після створення замовлення необхідно:

- Зберегти замовлення у власній базі;
- Надіслати підтвердження користувачу;
- Оновити аналітичну статистику;

- Передати інформацію в платіжний чи логістичний модуль.

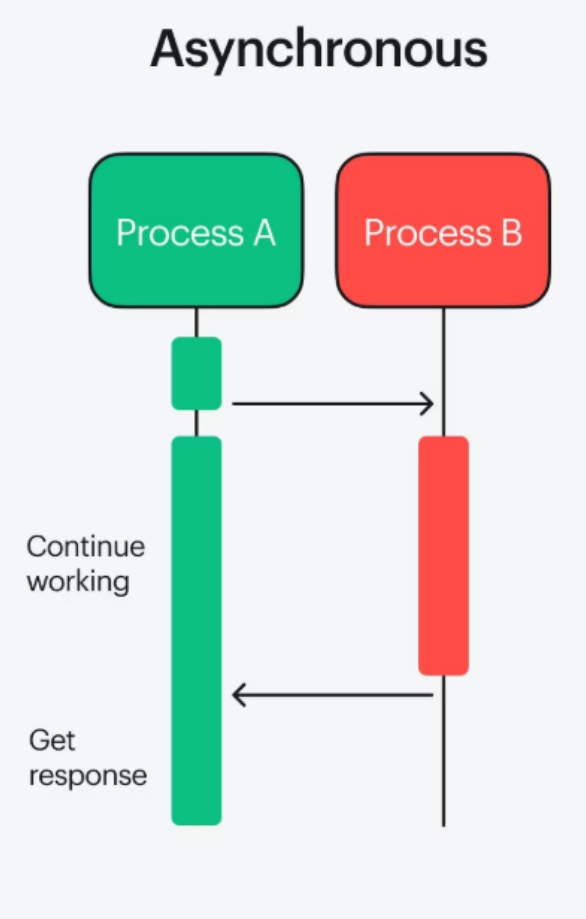


Рис. 1.3 Приклад асинхронної взаємодії

Якщо всі ці кроки виконувати послідовно в рамках одного синхронного HTTP-запиту, час відповіді для користувача помітно зросте, а надійність постраждає – збій будь-якої з внутрішніх підсистем може повністю заблокувати операцію.

У таких випадках застосовуються **асинхронні підходи**:

- Черги повідомлень (RabbitMQ, Kafka тощо);

- Обмін доменними подіями («OrderCreated», «UserRegistered» і т.ін.);
- Патерни event-driven архітектури.

Сервіс «Замовлення» у цьому разі публікує подію «Замовлення створено», а інші сервіси, що підписані на цю подію (аналітика, нотифікації), обробляють її незалежно. Це знижує зв’язність між компонентами та дозволяє масштабувати кожен з них окремо, але при цьому ускладнюється відслідковування повної бізнес-операції «від кінця до кінця» та узгодженість даних.

1.2.3 Проблеми та виклики мікросервісної архітектури

Незалежно від обраного підходу (REST, RPC, події), взаємодія мікросервісів породжує характерний набір викликів:

1. **Чутливість до затримок і мережевих збоїв.**
   Те, що в моноліті було локальним викликом методу, у розподіленій системі перетворюється на мережевий запит із небезпекою таймаутів, часткових відмов та повторних спроб.
2. **“Багатослівні” (chatty) API.**
   Якщо для виконання однієї бізнес-операції потрібна серія дрібних викликів між сервісами (наприклад, кілька разів звернутися до сервісу «Користувачі» при обробці одного замовлення), загальний час відповіді росте, а система стає менш передбачуваною.
3. **Версіонування та еволюція контрактів.**
   І REST-ендпоїнти, і RPC-контракти з часом змінюються. Потрібно зберігати сумісність між різними версіями сервісів, щоб оновлення окремого компонента не ламало всю систему.

4. **Спостережуваність та налагодження.** Трасування запитів, логування, метрики, кореляція подій між кількома сервісами — усе це стає критичним для підтримки працездатності системи.
5. **Безпека та єдині політики доступу.** Необхідно гарантувати узгоджену автентифікацію та авторизацію між різними сервісами, особливо коли частина з них є внутрішніми, а частина — доступна ззовні.

1.3 gRPC як сучасний підхід до взаємодії мікросервісів

У той час як REST поверх HTTP став де-факто стандартом для публічних веб-API, у внутрішній взаємодії мікросервісів усе частіше використовуються більш ефективні протоколи. Одним із найпоширеніших сучасних підходів є **gRPC** – фреймворк від Google для віддаленого виклику процедур (Remote Procedure Call), що побудований поверх HTTP/2 та використовує бінарний формат даних Protocol Buffers (Protobuf).

Ключова ідея gRPC полягає в тому, що взаємодія між сервісами описується у вигляді чітких контрактів: спеціальних .proto-файлів, де визначаються сервіси, їхні методи та структури повідомлень. На основі цього опису автоматично генеруються серверні та клієнтські “заготовки” для різних мов програмування. Таким чином, замість роботи з HTTP-запитами та JSON-структурами розробник оперує звичними методами та типізованими об’єктами.

У контексті мікросервісної системи з доменами «Користувачі» та «Замовлення» (рис. це означає, що:

1. Сервіс «Користувачі» експонує gRPC-інтерфейс із методами на кшталт GetUser, GetUserSettings, GetUserDiscounts тощо;
2. Сервіс «Замовлення» виступає gRPC-клієнтом і викликає ці методи як звичайні функції, не працюючи безпосередньо з HTTP-заголовками, JSON-парсингом тощо.

Такий підхід має кілька важливих переваг порівняно з внутрішнім REST-взаємодією:

1. **Ефективність передачі даних.**
   Використання бінарного формату Protobuf зменшує обсяг переданих по мережі даних та прискорює їхнє (де)серіалізування в порівнянні з текстовими форматами, такими як JSON. Це особливо помітно, коли між сервісами відбувається великий обсяг викликів або передаються складні структури.
2. **Строга типізація та єдиний контракт.**
   Структури повідомлень описуються один раз у .proto-файлах, і на їхній основі генерується код для різних мов. Це знижує ризик помилок сумісності між сервісами та спрощує еволюцію API: зміни контрактів можна контролювати централізовано.
3. **Різні режими комунікації.**
   На відміну від класичної схеми запит-відповідь у REST, gRPC “із коробки” підтримує:

   звичайні одноразові виклики (unary);

   серверний стримінг (server streaming);

   клієнтський стримінг (client streaming);

   двосторонній стримінг (bidirectional streaming),

Це розширює можливості для побудови більш складних сценаріїв обміну даними між мікросервісами без додаткових надбудов.

4. **Автоматична генерація клієнтів.**
   Розробникам мікросервісів не потрібно вручну писати HTTP-клієнти, моделі даних та код для (де)серіалізації – усе це генерується інструментами gRPC. У результаті зменшується кількість “клейового” коду, який важко підтримувати.

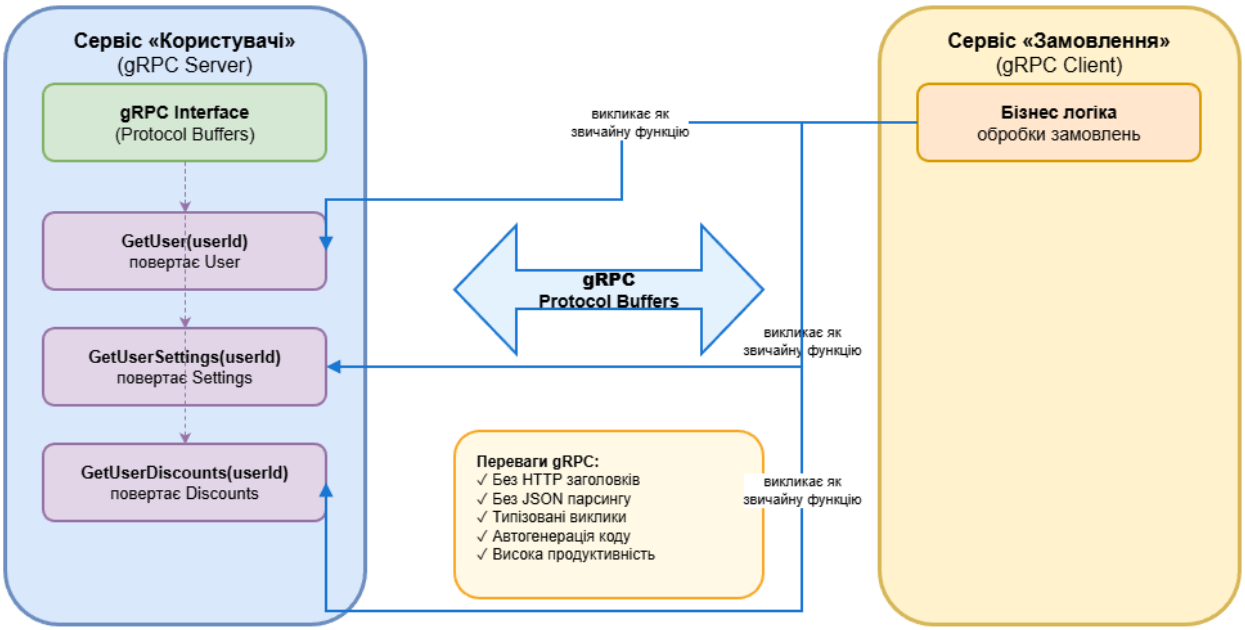


Рис. 1.5 Взаємодія мікросервісів через gRPC протокол

Водночас застосування gRPC має і свої обмеження. Оскільки протокол орієнтований на HTTP/2 і бінарний формат, то:

- **пряма інтеграція з браузерами та простими REST-клієнтами ускладнена**, оскільки більшість з них очікує HTTP/1.1+JSON;
- аналіз трафіку “на око” (наприклад, через звичайні HTTP-проксі) стає менш зручним через бінарне кодування;
- потрібна додаткова інфраструктура для маршрутизації, спостереження та безпеки в розподіленому середовищі.

У реальних системах це призводить до появи **гібридних рішень**, де:

- **зовнішні клієнти** (браузери, мобільні застосунки, сторонні інтеграції) працюють через звичний REST-інтерфейс;
- **внутрішня взаємодія між мікросервісами** відбувається за допомогою gRPC.

Саме таку схему доцільно застосувати й у системі з сервісами «Користувачі» та «Замовлення»: назовні експонується REST-API, тоді як внутрішні запити від API Gateway до окремих мікросервісів виконуються за допомогою gRPC. У такому варіанті Gateway виступає своєрідним “перекладачем” між HTTP/JSON і gRPC/Protobuf, поєднуючи зручність REST для клієнтів із ефективністю gRPC для внутрішньої комунікації.

1.4 API Gateway в мікросервісній архітектурі

Кожен сервіс може мати власний набір REST-ендпоїнтів, механізмів автентифікації, версій API, форматів відповіді. Якщо клієнтам (веб-застосунку, мобільним застосункам, стороннім інтеграціям) доводиться напряму звертатися до кількох сервісів, це призводить до:

- Збільшення кількості точок інтеграції;

- Ускладнення клієнтської логіки;
- Дублювання функціоналу, пов’язаного з безпекою, логуванням, обробкою помилок.

API Gateway вирішує цю проблему, виступаючи **єдиною точкою входу** до мікросервісної системи. Зовнішні клієнти взаємодіють лише з Gateway, а він уже маршрутизує запити до відповідних внутрішніх сервісів і агрегує результати.

У контексті системи з мікросервісами «Користувачі» та «Замовлення» API Gateway:

- Приймає всі зовнішні HTTP-запити (наприклад, GET /api/orders/{id}, POST /api/users/register);
- Виконує автентифікацію та базову авторизацію;
- Трансформує REST-запит і передає його у внутрішні сервіси;
- Отримує відповіді від мікросервісів, об’єднує й повертає їх клієнту у стандартному форматі.

При цьому важливою вимогою в межах цієї роботи є те, що **назовні система експонує лише REST-інтерфейс**. Це означає, що для клієнтів протокол взаємодії є звичним (HTTP/JSON), тоді як **внутрішня комунікація між Gateway та мікросервісами «Користувачі» і «Замовлення» може бути оптимізована за допомогою gRPC**.

API Gateway виконує низку типових задач, характерних для мікросервісної архітектури:

1. **Маршрутизація запитів.**
   Відповідно до шляху (/api/users/..., /api/orders/...) та методу (GET, POST тощо)

Gateway визначає, до якого мікросервісу необхідно спрямувати запит. Наприклад:

- GET /api/orders/{id} → внутрішній gRPC-виклик до сервісу «Замовлення»;
- GET /api/users/{id} → внутрішній gRPC-виклик до сервісу «Користувачі».

2. **Агрегація відповідей.**

   У багатьох сценаріях клієнту потрібна **комбінована інформація**, яка зберігається у кількох сервісах. Наприклад, деталі замовлення разом із короткою інформацією про користувача. Замість того, щоб змушувати клієнта робити кілька REST-запитів до різних сервісів, Gateway:

   - викликає внутрішні сервіси (наприклад, через gRPC),
   - об’єднує результати,
   - повертає один узгоджений REST-відповідь.

3. **Безпека та політики доступу.**

   API Gateway є зручним місцем для реалізації:

   - автентифікації (перевірка токена, сесії тощо),
   - базової авторизації (перевірка прав доступу до певних ресурсів),
   - обмеження частоти запитів (rate limiting). Таким чином, внутрішні сервіси «Користувачі» та «Замовлення» можуть довіряти тому, що запит, отриманий від Gateway, уже пройшов необхідні перевірки.

4. **Уніфікація форматів та версіонування.**

   Gateway може:

   - надавати клієнтам стабільний REST-контракт (наприклад, /api/v1/...),

- всередині при цьому викликати різні версії мікросервісів або виконувати перетворення форматів даних (наприклад, REST/JSON ↔ gRPC/Protobuf).

5. **Спостережуваність і обробка помилок.**
   Зосередження зовнішніх запитів у одній точці дозволяє:
   - централізовано збирати метрики (час відповіді, кількість помилок),
   - логувати запити й відповіді,
   - реалізовувати єдину політику обробки помилок та повернення стандартних кодів HTTP.

Оскільки в межах роботи передбачається, що зовнішній інтерфейс системи – це REST**, а** внутрішня взаємодія оптимізована за допомогою gRPC, API Gateway фактично виконує роль “перекладача” між цими двома підходами.

Типовий сценарій виглядає так:

1. Клієнт надсилає REST-запит, наприклад GET /api/orders/{id}.
2. API Gateway:
   - валідовує запит і параметри,
   - за потреби перевіряє токен автентифікації,
   - формує внутрішній gRPC-запит до сервісу «Замовлення» (наприклад, метод GetOrder з параметром order_id).
3. Сервіс «Замовлення» виконує бізнес-логіку та повертає gRPC-відповідь у форматі Protobuf.
4. Gateway трансформує цю відповідь у REST-формат (JSON) і повертає її клієнту.

Якщо для формування відповіді потрібні дані ще й із сервісу «Користувачі», Gateway може додатково:

- викликати метод GetUser у сервісі «Користувачі» через gRPC;
- об’єднати відомості про замовлення та користувача в один REST-відповідь.

Таким чином, клієнт бачить єдиний REST-API, не знаючи про внутрішню структуру мікросервісів, протоколи їхньої взаємодії та інші технічні деталі. Це полегшує розвиток клієнтських застосунків і дає змогу змінювати внутрішню реалізацію (наприклад, замінювати REST між сервісами на gRPC) без порушення зовнішніх контрактів.

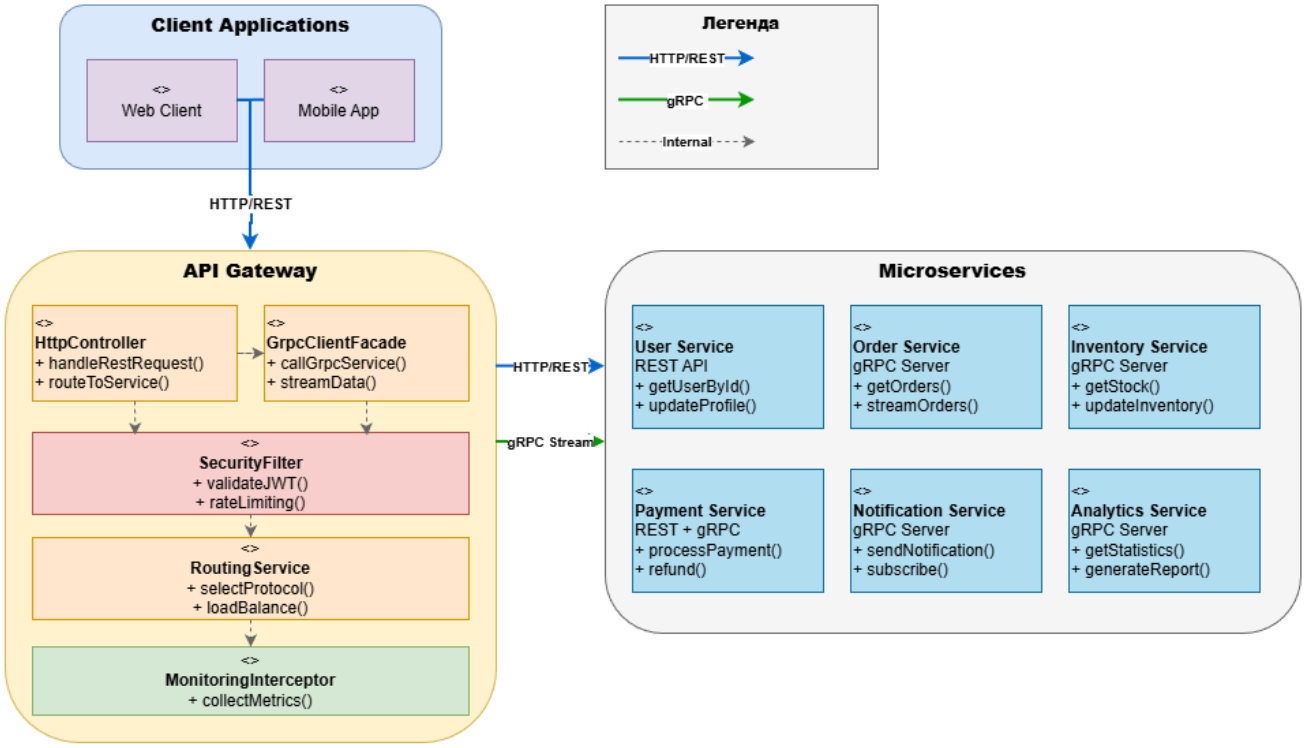


Рис. 1.6 Взаємодія мікросервісів через API Gateway

1.5 Аналіз аналогів

До найпоширеніших рішень API-шлюзів, що використовуються у сучасних мікросервісних архітектурах, належать Kong, NGINX / NGINX Plus, Spring Cloud Gateway, AWS API Gateway, Traefik, а також спеціалізовані gRPC-проксі на кшталт Envoy. Вони відрізняються не лише реалізацією, але й орієнтацією: одні більше підходять для публічних зовнішніх API, інші – для внутрішнього сервіс-мешу, ще інші виступають універсальними «фронтами» для будь-яких мікросервісних рішень. Одним із найвідоміших рішень є Kong – API-шлюз з відкритим кодом, який працює поверх NGINX та OpenResty.

Kong підтримує розширення за допомогою плагінів (автентифікація, логування, rate limiting, кешування, маршрутизація за шляхом і заголовками), а також інтеграцію з різними системами зберігання конфігурації та стану. Перевагою Kong є його зрілість як продукту, наявність великої спільноти та комерційної підтримки. Водночас гнучка конфігурація, заснована на Lua-скриптах та окремих сервісах, може ускладнювати вхідний поріг для команд, які не мають попереднього досвіду роботи з NGINX-екосистемою.

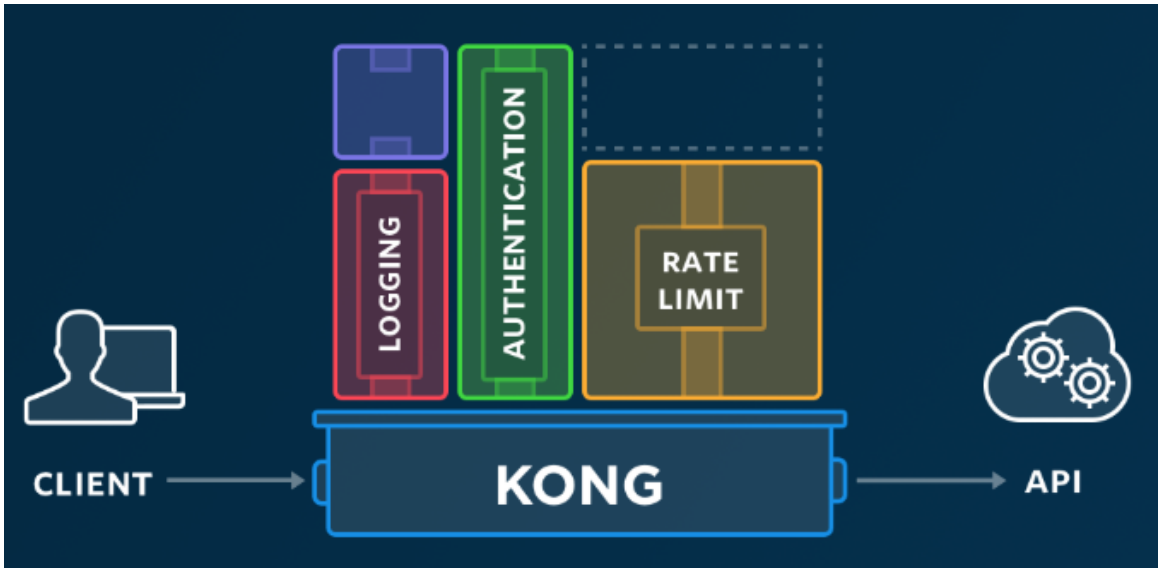


Рис 1.7 Архітектура Kong Gateway

Ще одне широко розповсюджене рішення – NGINX та його комерційна версія NGINX Plus. Хоча NGINX історично розглядається як веб-сервер та балансувальник навантаження, на практиці він часто використовується як простий API-шлюз: для термінації TLS, маршрутизації HTTP-запитів до бекенд-сервісів, реалізації елементарних правил доступу та кешування. Для складніших сценаріїв доступні модулі OpenResty, які дозволяють виконувати скрипти на Lua та впроваджувати більш гнучку бізнес-логіку на рівні проксі. Однак при побудові складних мікросервісних систем із великою кількістю сервісів та маршрутів конфігурація NGINX може ставати важко підтримуваною, а розширення функціональності – трудомістким.

Особливе місце серед аналогів посідає Spring Cloud Gateway – API-шлюз, побудований поверх Java-стека та екосистеми Spring. Він орієнтований на розробників, які використовують Spring Boot, Spring Cloud та пов’язану інфраструктуру. Основною перевагою Spring Cloud Gateway є тісна інтеграція з Java-сервісами: маршрутизація, фільтри, механізми безпеки, трейсинг та логування можуть описуватися як конфігурацією, так і звичайним Java-кодом, що полегшує розробку й тестування. Крім того, Spring Cloud Gateway добре поєднується з іншими компонентами Spring-екосистеми (Spring Security, Spring Cloud Config, Eureka тощо). Разом з тим, порівняно з легковаговими рішеннями на базі NGINX, Java-орієнтований шлюз зазвичай має більші вимоги до ресурсів та часу старту, що варто враховувати для високонавантажених чи обмежених середовищ.

У хмарних платформах широкого поширення набули керовані API-шлюзи, такі як AWS API Gateway, Azure API Management та Google Cloud Endpoints. Вони надають розробникам готову інфраструктуру для публікації REST- та WebSocket-інтерфейсів, вбудовану автентифікацію, облік трафіку та інтеграцію з іншими сервісами відповідної хмари. Головною перевагою таких рішень є

мінімізація операційних витрат: розгортання, масштабування й моніторинг виконує провайдер. Водночас подібні шлюзи часто орієнтовані переважно на зовнішній трафік, а внутрішня взаємодія мікросервісів залишається на розсуд архітектора (через REST, черги повідомлень або спеціалізовані внутрішні проксі).

Окремий клас рішень становлять високопродуктивні проксі-сервери, спроєктовані з урахуванням протоколів HTTP/2 та gRPC, зокрема Envoy. Envoy широко використовується як базовий компонент у сервіс-мешах (Istio, Consul Connect тощо), забезпечуючи інтелектуальну маршрутизацію, балансування, ретрі, тайм-аути, шифрування трафіку та спостережність. Його сильна сторона – тісна інтеграція з gRPC, підтримка discovery-механізмів та можливість централізованого керування великою кількістю сервісів. Разом із тим, впровадження Envoy-подібних рішень зазвичай потребує глибокої експертизи DevOps та знань Kubernetes-орієнтованої інфраструктури, що не завжди виправдано для відносно невеликих систем або академічних проєктів.

2 АНАЛІЗ МЕТОДІВ ВЗАЄМОДІЇ МІКРОСЕРВІСІВ З ВИКОРИСТАННЯМ GATEWAY

2.1 Формалізація архітектури мікросервісів

Мікросервісна архітектура зазвичай будується навколо кількох ключових компонентів, кожен з яких відповідає за конкретний бізнес-домен. З метою формалізації будемо вважати, що система складається з трьох основних сервісів.

1. Множину цих мікросервісів позначимо як:

$$S = \{G, U, O\} \tag{2.1}$$

де:

G – **API Gateway**, що слугує єдиною точкою входу до системи для всіх зовнішніх клієнтів;

U – мікросервіс **«Користувачі»** (Users), відповідальний за обробку інформації про користувачів;

O – мікросервіс **«Замовлення»** (Orders), відповідальний за обробку інформації про замовлення.

Для опису зв’язків між сервісами використаємо апарат орієнтованих графів, виведений граф взаємодії:

$$G = (S, E) \tag{2.2}$$

де:

$E \subseteq S \times S$ – множина орієнтованих ребер.

Кожне ребро $(i, j) \in E$ інтерпретується як можливість здійснювати віддалений виклик сервісу j із сервісу i. Наприклад, наявність ребра (G, O) означає, що API Gateway може викликати мікросервіс «Замовлення» для отримання чи модифікації інформації про замовлення.

Оскільки кінцевим користувачем системи є зовнішній клієнт (браузер, мобільний застосунок, стороння система), введемо додатковий вузол C, який моделює зовнішнього клієнта. Взаємодія такого клієнта із системою відбувається виключно через API Gateway, тому:

$$(C, G) \in E \tag{2.3}$$

Розглянемо окрему бізнес-операцію, наприклад, отримання інформації про замовлення користувача. З погляду клієнта ця операція зводиться до одного запиту до REST-інтерфейсу (GET /api/orders/{id}), проте всередині системи вона може породжувати кілька послідовних внутрішніх викликів між мікросервісами. Це природно моделювати у вигляді маршруту (шляху) в графі G.

Нехай одна операція відповідає деякому маршруту:

$$P = (v_0, v_1, \dots, v_n) \tag{2.4}$$

де:

v_0 – зовнішній клієнт, який ініціює запит

v_n – клієнт, який отримує кінцеву відповідь

кожен перехід (v_{k-1}, v_k) є елементом множини E і відповідає одному віддаленому виклику (REST або gRPC).

З практичної точки зору кожен віддалений виклик супроводжується певними часовими витратами й породжує певний мережевий трафік. Тому для кожного виклику з номером $k(k = 1, \dots, n)$ введемо такі параметри:

t_k^{net} – мережевий час передавання запиту та відповіді, який враховує затримки в мережі, маршрутизацію пакетів, можливі повторні передачі тощо

t_k^{ser} – час, витрачений на серіалізацію даних у формат, придатний для передавання по мережі, та подальшу десеріалізацію на стороні отримувача

t_k^{proc} – час безпосередньої обробки запиту на стороні сервісу (виконання бізнес-логіки, доступ до бази даних тощо)

b_k – загальний обсяг даних, що передається в рамках цього виклику (сума розмірів запиту та відповіді)

Отже, кожен виклик можна описати вектором параметрів:

$$x_k = \left(t_k^{net}, t_k^{ser}, t_k^{proc}, b_k\right) \tag{2.5}$$

Таким чином, маршрут P, що відповідає одній бізнес-операції, можна розглядати як послідовність таких векторів x_k. Важливо підкреслити, що для різних підходів до організації взаємодії (внутрішня REST-взаємодія чи внутрішня gRPC-взаємодія) як кількість викликів n, так і значення параметрів x_k. будуть відрізнятися. Це дозволяє формально порівнювати різні архітектурні рішення, спираючись не лише на якісні, але й на кількісні показники.

У подальшому аналізі розглядатимемо два характерні варіанти організації внутрішньої взаємодії:

1. варіант із внутрішнім REST-обміном, коли всі внутрішні виклики між сервісами G, U, O виконуються за допомогою REST-API поверх HTTP/1.1 із використанням формату JSON
2. варіант із внутрішнім gRPC-обміном, у якому зовнішній інтерфейс $C \leftrightarrow G$ зберігається у вигляді REST/HTTP+JSON, проте всі внутрішні виклики між G, U та O реалізовано за допомогою gRPC поверх HTTP/2 із використанням бінарного формату Protocol Buffers (Protobuf)

При цьому основний акцент робиться на мінімізації часу відповіді, оскільки саме цей показник найбільш критичний для користувачів і безпосередньо впливає на суб'єктивне сприйняття швидкодії системи. Зменшення обсягу внутрішнього трафіку, у свою чергу, розглядається як важливий додатковий ефект, який може позитивно позначитися на масштабованості та вартості експлуатації системи.

2.2 Математична модель часу обробки запиту та обсягу трафіку

Виходячи з введених позначень, **загальний час обробки одного запиту** клієнта, який проходить маршрут P, можна подати у вигляді суми часових витрат по всіх викликах:

$$T(P) = \sum_{k=1}^{n} (t_k^{net} + t_k^{ser} + t_k^{proc}) \quad (2.6)$$

Фактично величина $T(P)$ описує повний час від моменту, коли клієнт надіслав свій запит, до моменту, коли він отримав відповідь. У складі цієї величини можна виділити:

- час, витрачений на проходження запиту через мережу
- час, витрачений на перетворення даних у придатний для передавання формат і назад
- власне час виконання бізнес-логіки в кожному з залучених сервісів

Загальний обсяг переданих даних при обробці цього ж запиту дорівнює сумі обсягів даних по всіх викликах:

$$V(P) = \sum_{k=1}^{n} b_k \tag{2.7}$$

Величина $V(P)$ характеризує навантаження на мережеву інфраструктуру зі сторони одного запиту й дозволяє оцінити, наскільки ефективно використовується пропускна здатність мережі при великій кількості одночасних запитів.

Оскільки навіть для одного типу бізнес-операції маршрути P і відповідні параметри x_k можуть змінюватися (через різні гілки логіки, особливості даних, варіації мережевих умов), далі доцільно розглядати не окремі значення, а очікувані (середні) величини:

$$\sum[T] = \sum[T(P)],\ \sum[V] = \sum[V(P)] \tag{2.8}$$

де математичне сподівання береться за множиною можливих маршрутів P і випадкових варіацій параметрів x_k для фіксованого типу запиту (наприклад, «отримати замовлення користувача»).

Таким чином, $\sum[T]$ можна інтерпретувати як середній час відповіді системи на запит певного типу, а $\sum[V]$ – як середній обсяг трафіку, необхідний для обробки такого запиту.

2.2.1 Математична модель часу обробки запиту та обсягу трафіку

Спочатку розглянемо варіант, коли всі внутрішні взаємодії між мікросервісами здійснюються за допомогою REST-API. Для спрощення аналізу припустимо, що:

- зовнішня взаємодія між клієнтом *C* та Gateway *G* також є REST-запитом
- усі внутрішні виклики між *G*, *U* та *O* виконуються через HTTP/1.1 із використанням JSON як формату обміну даними

У типових сценаріях обробки запитів, пов'язаних з отриманням інформації про замовлення, можна виділити:

1. один зовнішній REST-виклик від клієнта до Gateway (наприклад, GET /api/orders/{id})
2. один внутрішній REST-виклик від Gateway до сервісу «Замовлення» $G \rightarrow O$
3. один або кілька внутрішніх REST-викликів між сервісами «Замовлення» та «Користувачі» $O \rightarrow U$ для отримання додаткової інформації про користувача (його профіль, налаштування, знижки тощо).

Загальна кількість REST-викликів, пов'язаних з обробкою одного запиту клієнта, дорівнює:

$$N_{rest} = 1 + N_{rest}^{int} \tag{2.9}$$

де:

N_{rest}^{int} – кількість внутрішніх REST-викликів

1 – відповідає зовнішньому вклику клієнт ↔ Gateway

Для подальшого аналізу введемо середні значення часових витрат і обсягу даних для одного внутрішнього REST-виклику:

$$t_{rest}^{-net},\ t_{rest}^{-ser},\ t_{rest}^{-proc}, b_{rest}^{-} \tag{2.10}$$

де:

t_{rest}^{-net} – середній мережевий час для одного REST-виклику між мікросервісами

t_{rest}^{-ser} – середній час на (де)серіалізацію JSON-даних

t_{rest}^{-proc} – середній час виконання бізнес-логіки мікросервісу для такого внутрішнього виклику

b_{rest}^{-} – середній обсяг даних (запиту+відповіді) в одному внутрішньому REST-виклику

Тоді середній час обробки одного клієнтського запиту у варіанті внутрішнього REST-обміну можна наближено записати у вигляді:

$$\sum[T_{rest}] \approx t_{rest}^{ext} + N_{rest}^{int}\,(t_{rest}^{-net} + t_{rest}^{-ser} + t_{rest}^{-proc}) \tag{2.11}$$

де:

t_{rest}^{ext} – середній час, пов’язаний із зовнішнім REST-викликом клієнт → Gateway (включає мережеву затримку між клієнтом і Gateway, час (де)серіалізації JSON на рівні Gateway, а також час обробки запиту безпосередньо в Gateway).

Із наведених виразів видно, що і час обробки, і обсяг трафіку лінійно зростають із кількістю внутрішніх викликів N_{rest}^{int}. Тобто в системі, де одна бізнес-операція породжує велику кількість дрібних REST-викликів між мікросервісами, спостерігається закономірне зростання затримок і навантаження на мережу.

2.2.2. Модель для внутрішньої gRPC-взаємодії

Перейдемо до випадку, коли для внутрішньої взаємодії між мікросервісами застосовується gRPC. У цьому варіанті:

- зовнішній інтерфейс між клієнтом і API Gateway залишається REST-орієнтованим (HTTP/1.1 + JSON);
- всі внутрішні виклики між сервісами *G*, *U* та *O* виконуються за допомогою gRPC поверх HTTP/2 із використанням бінарного формату Protobuf.

Припустимо, що в аналогічному сценарії обробки запиту (отримання замовлення користувача) кількість внутрішніх gRPC-викликів дорівнює N_{grpc}^{int} Зазвичай завдяки кращій агрегації даних і можливостям gRPC (наприклад, передавати відразу всі потрібні поля в одному виклику) **значення N_{grpc}^{int} може бути меншим**, ніж N_{rest}^{int} для REST-варіанта.

Беручи до уваги, що бінарне представлення Protobuf зазвичай компактніше за текстовий JSON, уведемо коефіцієнт стискання даних:

$$\alpha_b = \frac{b_{grpc}}{b_{rest}},\ 0 < \alpha_b < 1 \tag{2.12}$$

Тоді співвідношення між середніми величинами часу й трафіку для REST і gRPC можна записати більш наочно:

$$\sum[T_{grpc}] \approx t_{rest}^{ext} + N_{gprc}^{int}\,(t_{grpc}^{net} + \alpha_{ser} t_{rest}^{ser} + t_{grpc}^{proc}) \tag{2.13}$$

Таким чином, при фіксованих значеннях $t_{grpc}^{net} + \alpha_{ser} t_{rest}^{ser} + t_{grpc}^{proc}$ та інших параметрів, виграш від переходу на gRPC досягається за рахунок:

- меншого обсягу даних ($\alpha_b < 1$)

- менших витрат на (де)серіалізацію ($\alpha_{ser} < 1$)
- потенційного зменшення кількості внутрішніх викликів N_{gprc}^{int} за рахунок кращої агрегації та більш виразних контрактів gRPC.

2.2.3 Порівняння варіантів та критерій ефективності

На основі побудованих моделей можна ввести кількісні показники, що характеризують переваги одного підходу над іншим. Виграш в часі можна зазначити так:

$$\Delta V = \sum[T_{rest}] - \sum[T_{grpc}] \tag{2.14}$$

де:

$\sum[T_{rest}]$ – середній обсяг трафіку при внутрішній REST-взаємодії

$\sum[T_{grpc}]$ – середній час обробки запиту при внутрішній gRPC-взаємодії

Тоді зменшення трафіку:

$$\Delta V = \sum[V_{rest}] - \sum[V_{grpc}] \tag{2.15}$$

де:

$\sum[V_{rest}]$ – середній обсяг трафіку при внутрішній REST-взаємодії

$\sum[V_{grpc}]$ – середній обсяг трафіку при внутрішній gRPC-взаємодії

Підставляючи у вираз для ΔT отримані раніше формули, можна побачити, що на величину виграшу впливають:

- різниця в кількості внутрішніх викликів
- коефіцієнти α_{ser} та α_b, які відображають ефективність gRPC у порівнянні з REST щодо (де)серіалізації та обсягу даних;
- мережеві характеристики, зокрема можливості HTTP/2 (multiplexing, більш ефективне використання з’єднань) у порівнянні з HTTP/1.1.

3 ПРОЕКТУВАННЯ СИСТЕМИ З API GATEWAY

3.1 Архітектура програмної системи

На основі проведеного у попередньому розділі аналізу методів взаємодії мікросервісів та побудованої математичної моделі часу обробки запитів у даному розділі виконується проєктування конкретної програмної системи. Система реалізується з використанням мови програмування Java та екосистеми Spring, що забезпечує широкий набір інструментів для побудови мікросервісної архітектури, включно з підтримкою REST-інтерфейсів, інтеграцією з gRPC та засобами конфігурації й моніторингу.

Проєктована система реалізує розподілену архітектуру, у якій виділяються три основні компоненти:

1. API Gateway – окремий мікросервіс, який:
    - приймає всі зовнішні HTTP-запити від клієнтів
    - надає публічний REST-інтерфейс для роботи з користувачами та замовленнями
    - виконує попередню обробку запитів (валідація, автентифікація)
    - транслює REST-запити у внутрішні gRPC-виклики до інших мікросервісів
    - агрегує відповіді та повертає клієнтам узгоджені JSON-дані.
2. **Мікросервіс «Користувачі» (User Service)** – сервіс, відповідальний за:
    - реєстрацію та керування обліковими записами користувачів
    - зберігання базових даних профілю (ім’я, контактні дані тощо)
    - надання інформації про користувача іншим сервісам через gRPC-інтерфейс

3. **Мікросервіс «Замовлення» (Order Service)** – сервіс, відповідальний за:
    - створення, оновлення та перегляд замовлень
    - зберігання їхніх статусів та історії змін
    - отримання інформації про користувача (через API Gateway або безпосередній gRPC-виклик), якщо це потрібно для формування розширеної інформації про замовлення.

Кожен із цих компонентів реалізується як окремий Spring-застосунок, що розгортається автономно та може масштабуватися незалежно від інших. Зовнішні клієнти сприймають всю систему як єдиний REST-API, тоді як внутрішня взаємодія між компонентами оптимізована відповідно до моделі, наведеної в розділі 2.

Таблиця 3.1

Порівняння REST/JSON та gRPC / Protobuf при внутрішний взаємодії мікросервісів

Характеристика	REST/JSON	gRPC / Protobuf
Формат обміну	Текстовий JSON	Бінарний Protobuf
Надмірність даних	Висока (повторення імен полів, службових символів)	Низька (компактні бінарні структури)
Обсяг трафіку	Більший, особливо для колекцій	Менший, краще масштабується з ростом даних
Час серіалізації/десеріалізації	Вищий за рахунок парсингу тексту	Нижчий за рахунок фіксованих бінарних структур
Підтримка браузером	Нативно (через HTTP/JSON)	Ненативно, потрібен проксі / шлюз
Людинозрозумілість	Висока (дані легко читати «очима»)	Низька (потрібні інструменти для декодування)
Розвиток контрактів	Гнучкий, але часто неформалізований	Чіткі .proto контракти з явною типізацією
Інструменти генерації коду	Обмежені, часто ручна робота	Автоматична генерація клієнтів і серверів з .proto
Підтримка стрімінгу	Обмежена (long-polling, SSE, WebSocket)	Вбудовані режими стрімінгу (server, client, bidirectional)
Застосування у внутрішньому контурі	Зручно, але неефективно за ресурсами	Особливо доцільно для високонавантажених мікросервісних систем

Архітектурно система може бути описана у вигляді багаторівневої структури:

1. **Рівень клієнтів** – веб-застосунки, мобільні застосунки, сторонні інтеграції, які взаємодіють із системою за допомогою HTTP/REST. Кожен клієнт надсилає запити до єдиного доменного імені/ендпоїнта, за яким розгорнуто API Gateway.
2. **Рівень API Gateway** – проміжний шар, який інкапсулює всю внутрішню складність мікросервісної архітектури. З точки зору клієнта Gateway є «єдиним сервером», хоча фактично він лише маршрутизує запити до відповідних мікросервісів, стежить за версіями API, застосовує політики безпеки й забезпечує агрегацію даних.
3. **Рівень бізнес-сервісів** – набір мікросервісів, які реалізують окремі бізнес-домени. У межах даної роботи основними є «Користувачі» та «Замовлення», але архітектура залишається відкритою до розширення (додавання сервісів «Оплати», «Нотифікації», «Аналітика» тощо). Кожен сервіс має власну базу даних, власну модель даних та незалежний життєвий цикл розгортання.
4. **Рівень даних** – сховища, що використовуються окремими мікросервісами. У найпростішому випадку кожен сервіс взаємодіє з реляційною базою даних (наприклад, PostgreSQL) через ORM-рішення, таке як Spring Data JPA. Подальше розширення може включати використання NoSQL-сховищ, кешів та інших засобів оптимізації.

З огляду на модель із розділу 2, особливу увагу в проєктуванні приділяється шляхам проходження запиту через систему. Для типового запиту, наприклад, на отримання інформації про замовлення користувача:

1. Клієнт надсилає REST-запит до API Gateway
2. API Gateway
    - перевіряє коректність запиту та права доступу
    - формує внутрішній gRPC-виклик до сервісу «Замовлення»
3. Сервіс «Замовлення» обробляє запит, за потреби отримує додаткові дані про користувача (або через окремий gRPC-виклик до сервісу «Користувачі», або через уже агреговані дані, надані Gateway).
4. Сформована відповідь повертається до API Gateway, який перетворює її у формат JSON і відправляє клієнту.

Архітектура побудована так, щоб:

1. мінімізувати кількість внутрішніх викликів для одного клієнтського запиту
2. забезпечити ефективне кодування та передавання даних між сервісами (завдяки gRPC і Protobuf);
3. зберегти для клієнтів зручний і звичний REST-інтерфейс, не вимагаючи від них підтримки gRPC чи інших спеціалізованих протоколів

3.2 Проєктування REST-інтерфейсу API Gateway

Відповідно до вимог, сформульованих у попередніх розділах, зовнішній інтерфейс системи повинен бути побудований виключно на основі REST-підходу. Це означає, що всі клієнти (веб-, мобільні застосунки, сторонні інтеграції) взаємодіють із системою через HTTP-запити, що оперують ресурсами, представленими у форматі JSON. Усі елементи внутрішньої реалізації (наявність мікросервісів, використання gRPC тощо) мають бути приховані за шаром API Gateway.

REST-інтерфейс API Gateway проєктується таким чином, щоб:

1. відобразити основні домени системи – користувачі та замовлення – у вигляді чітко визначених ресурсів;
2. забезпечити мінімально необхідний набір операцій CRUD (Create, Read, Update, Delete) над цими ресурсами;
3. надати окремі ендпоїнти для агрегованих операцій, при яких одночасно повертаються дані з кількох доменів (наприклад, замовлення разом із даними про користувача);
4. підтримувати єдині правила версіонування, обробки помилок, пагінації та фільтрації

Для уникнення конфліктів і подальшого спрощення еволюції інтерфейсу вводиться префікс версії API, наприклад /api/v1, який додається до всіх URI, що надаються зовнішнім клієнтам.

3.2.1 Ресурс «Користувачі»

Ресурс «Користувачі» відображає об’єкти предметної області, пов’язані з обліковими записами користувачів, їхніми базовими даними та профілями. Основні операції над цим ресурсом виконуються через такі REST-ендпоїнти API Gateway, на рисунку 3.1, деталізація використаних технологій та взаємодія мікросервісів.

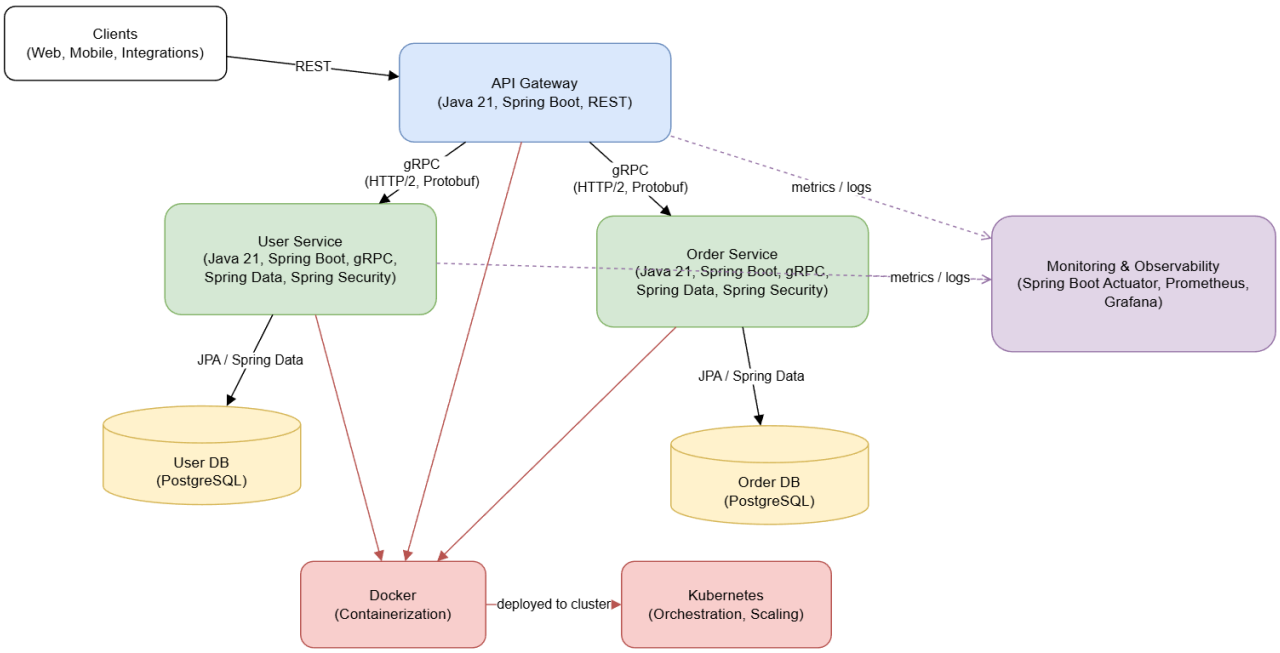


Рис. 3.1 Технологічний стек компонентів та їх взаємодія

1. Створення користувача (реєстрація):
    - метод: POST
    - шлях: /api/v1/users
    - призначення: реєстрація нового користувача в системі
    - вхідні дані: JSON-об’єкт, що містить обов’язкові атрибути (електронна пошта, пароль, ім’я)
    - вихідні дані: JSON-об’єкт із базовими відомостями про створеного користувача та, ідентифікатором ресурсу (id).

2. Отримання інформації про користувача:
    - метод: GET
    - шлях: /api/v1/users/{id}

- призначення: отримання базових відомостей про користувача за його ідентифікатором
- вихідні дані: JSON-об’єкт, що містить атрибути профілю (ім’я, e-mail, статус).

3. Оновлення профілю користувача:
    - метод: PUT
    - шлях: /api/v1/users/{id}
    - призначення: оновлення полів профілю
    - вхідні дані: JSON-об’єкт із полями, що підлягають оновленню
    - вихідні дані: оновлений JSON-об’єкт користувача
4. Отримання списку користувачів:
    - метод: GET
    - шлях: /api/v1/users
    - призначення: отримання колекції користувачів, можливо з пагінацією та фільтрацією (наприклад, за статусом)
    - додаткові параметри: page, size
    - вихідні дані: список користувачів у JSON-форматі, зазвичай із додатковою службовою інформацією (загальна кількість елементів, номер сторінки)

Усі ці операції з точки зору клієнта є звичайними REST-запитами, однак на рівні API Gateway кожному з них може відповідати один чи кілька внутрішніх gRPC-викликів до сервісу «Користувачі».

3.2.2 Ресурс «Замовлення»

Ресурс «Замовлення» описує об’єкти, пов’язані зі створенням, зберіганням та зміною замовлень користувачів. REST-інтерфейс API Gateway для цього ресурсу включає, зокрема, такі ендпоїнти:

1. Створення замовлення:
    - метод: POST
    - шлях: /api/v1/orders
    - призначення: створення нового замовлення, пов’язаного з певним користувачем
    - вхідні дані: JSON-об’єкт, що містить ідентифікатор користувача (userId) та параметри замовлення (перелік товарів, суми, додаткові опції)
    - вихідні дані: JSON-об’єкт із даними про створене замовлення (ідентифікатор, статус, дата створення).

2. Отримання інформації про конкретне замовлення:
    - метод: GET
    - шлях: /api/v1/orders/{id}
    - призначення: отримання базових даних про замовлення за його ідентифікатором
    - вихідні дані: JSON-об’єкт, що містить інформацію про замовлення (статус, сума, час створення).
3. Оновлення статусу замовлення:
    - метод: PUT
    - шлях: /api/v1/orders/{id}
    - призначення: зміна параметрів замовлення (передусім статусу)

- вхідні дані: JSON-об’єкт із полями, що підлягають оновленню (status)
- вихідні дані: оновлене представлення замовлення

4. Отримання списку замовлень користувача:
    - метод: GET
    - шлях: /api/v1/users/{userId}/orders
    - призначення: отримання всіх замовлень, пов’язаних із певним користувачем
    - додаткові параметри: пагінація, фільтрація за статусом, датою створення
    - вихідні дані: список замовлень у JSON-форматі

Як і у випадку ресурсу «Користувачі», стосовно кожного з цих ендпоїнтів API Gateway виконує низку службових дій (перевірка прав, валідація, логування), а потім ініціює відповідні внутрішні gRPC-виклики до сервісу «Замовлення» та, за потреби, до сервісу «Користувачі».

3.2.3 Загальні вимоги до REST-інтерфейсу Gateway

При проєктуванні REST-інтерфейсу API Gateway, окрім визначення конкретних URI та методів, важливо сформулювати загальні правила, яким він має відповідати:

1. однозначне версіонування API: усі ендпоїнти повинні містити префікс версії (/api/v1/...), це дозволяє у майбутньому розгортати нові версії API без порушення роботи існуючих клієнтів, а також, виконувати поетапну міграцію, коли частина клієнтів використовує /api/v1, а інша – /api/v2
2. узгоджений формат відповідей і помилок: успішні відповіді мають повертатися в єдиному форматі JSON, а помилки – із використанням стандартних HTTP-кодів (400, 401, 403, 404, 500 тощо) і структурованого

JSON-об’єкта з описом помилки. Це полегшує інтеграцію й дебаг сторонніх клієнтів.

3. підтримка пагінації та фільтрації для колекцій ресурсів: для ендпоїнтів, що повертають списки (GET /api/v1/users, GET /api/v1/orders, GET /api/v1/users/{userId}/orders), доцільно передбачити параметри пагінації (page, size) та базової фільтрації (наприклад, за статусом замовлення). Це дає змогу уникати надмірного розміру відповідей і знижує навантаження на внутрішні сервіси.
4. єдині принципи автентифікації та авторизації: API Gateway повинен перевіряти автентичність запитів (наприклад, за допомогою JWT-токенів у заголовках), за потреби обмежувати доступ до окремих ендпоїнтів (адміністративні операції тощо). Внутрішні мікросервіси, у свою чергу, можуть довіряти інформації, яка надходить від Gateway, не реалізовуючи дублюючих механізмів автентифікації
5. логування та трасування: Gateway є природною точкою для збирання логів запитів і формування кореляційних ідентифікаторів (trace_id), які потім передаються у внутрішні сервіси. Це спрощує аналіз часу обробки, пошук “вузьких місць” і валідацію теоретичної моделі.

Таким чином, REST-інтерфейс API Gateway виступає стабільною зовнішньою оболонкою системи, яка маскує внутрішню складність мікросервісної архітектури й забезпечує сумісність із широким спектром клієнтів. У той же час, завдяки продуманій маршрутизації та агрегації, він дозволяє на практиці досягти цілей, сформульованих у математичній моделі: зменшення часу обробки запитів і обсягу внутрішнього трафіку за рахунок переходу на gRPC усередині системи.

3.3 Програмні засоби реалізації

Для реалізації таких систем необхідно використовувати гнучкі мови програмування, які можна запустити на кожній платформі та ОС без необхідності встановлення додаткового програмного забезпечення.

3.3.1 Java

В якості основної платформи розробки використовується Java 21, яка є версією з довгостроковою підтримкою (LTS). Такий вибір забезпечує не лише стабільність та передбачуваність у довгостроковій перспективі завдяки регулярним оновленням безпеки та виправленням помилок, але й надає доступ до низки сучасних мовних можливостей.

Серед них – удосконалені механізми pattern matching, розширені можливості switch-виразів та подальший розвиток проекту Loom, що передбачає впровадження віртуальних потоків.

Ці нововведення значно підвищують виразність коду, спрощують розробку багатопотокових застосунків та відкривають нові горизонти для оптимізації продуктивності та масштабованості.

Крім того, постійні оптимізації віртуальної машини Java (JVM), включаючи вдосконалені алгоритми збирачів сміття та JIT-компілятора, сприяють підвищенню загальної продуктивності високонавантажених сервісів.

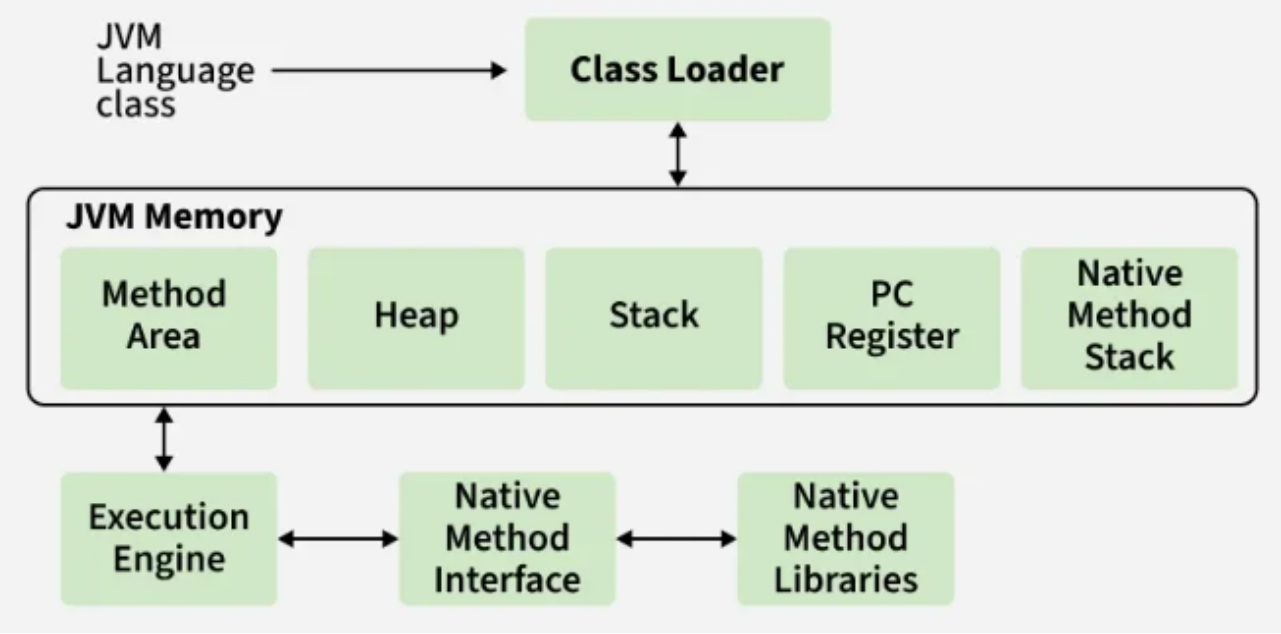


Рис. 3.2 Архітектура JVM

3.3.2 Spring Framework

Центральним елементом архітектури є фреймворк Spring, який є де-факто стандартом для розробки корпоративних застосунків на Java. Spring надає комплексну екосистему, що охоплює всі аспекти розробки мікросервісів. Його ядро, Spring Framework, забезпечує фундаментальні механізми, такі як інверсія керування (IoC) та ін’єкція залежностей (DI), що є ключовими для створення модульних, легко тестовних компонентів.

Для спрощення розробки та розгортання мікросервісів використовується Spring Boot – надбудова над Spring Framework, яка мінімізує конфігурацію завдяки автоконфігурації та надає єдину модель для збирання й запуску застосунків. Це дозволяє швидко створювати автономні, готові до продакшну мікросервіси, кожен з яких може бути розгорнутий незалежно. Для побудови зовнішнього REST-інтерфейсу API Gateway застосовуються можливості Spring Web або Spring

WebFlux, що дозволяють ефективно обробляти HTTP-запити, маршрутизувати їх та працювати з різними форматами даних, зокрема JSON. Це забезпечує зручну взаємодію із зовнішніми клієнтами, які очікують стандартний RESTful API.

У контексті доступу до даних, що є критично важливим для кожного мікросервісу, використовується Spring Data JPA. Ця абстракція над Java Persistence API (JPA) значно спрощує роботу з реляційними базами даних, такими як PostgreSQL, дозволяючи описувати сутності як звичайні Java-класи та автоматично генерувати запити до бази даних. Це сприяє чіткому відокремленню бізнес-логіки від деталей реалізації доступу до даних, підвищуючи підтримуваність та гнучкість системи.

Безпека є невід'ємною частиною будь-якої розподіленої системи. Spring Security надає потужні механізми для автентифікації та авторизації, що є особливо важливим для API Gateway. Він дозволяє централізовано перевіряти автентичність запитів (наприклад, за допомогою JWT-токенів) та обмежувати доступ до ресурсів на основі ролей або прав користувачів. Це дозволяє внутрішнім мікросервісам покладатися на перевірки, виконані Gateway, спрощуючи їхню власну логіку безпеки.

Для забезпечення надійного функціонування та супроводу системи в промисловому середовищі, передбачається використання інструментів моніторингу та логування. Spring Boot Actuator надає вбудовані ендпоїнти для перевірки стану застосунків (health-checks), збору метрик та інформації про оточення.

Це дозволяє інтегрувати систему з такими інструментами, як Prometheus для збору метрик, Grafana для їх візуалізації та централізованими системами логування (наприклад, ELK-стек). API Gateway, будучи єдиною точкою входу, є ідеальною

точкою для збору ключових метрик продуктивності, що дозволить емпірично підтвердити теоретичні висновки одо оптимізації часу відповіді та обсягу трафіку, отримані в математичній моделі.

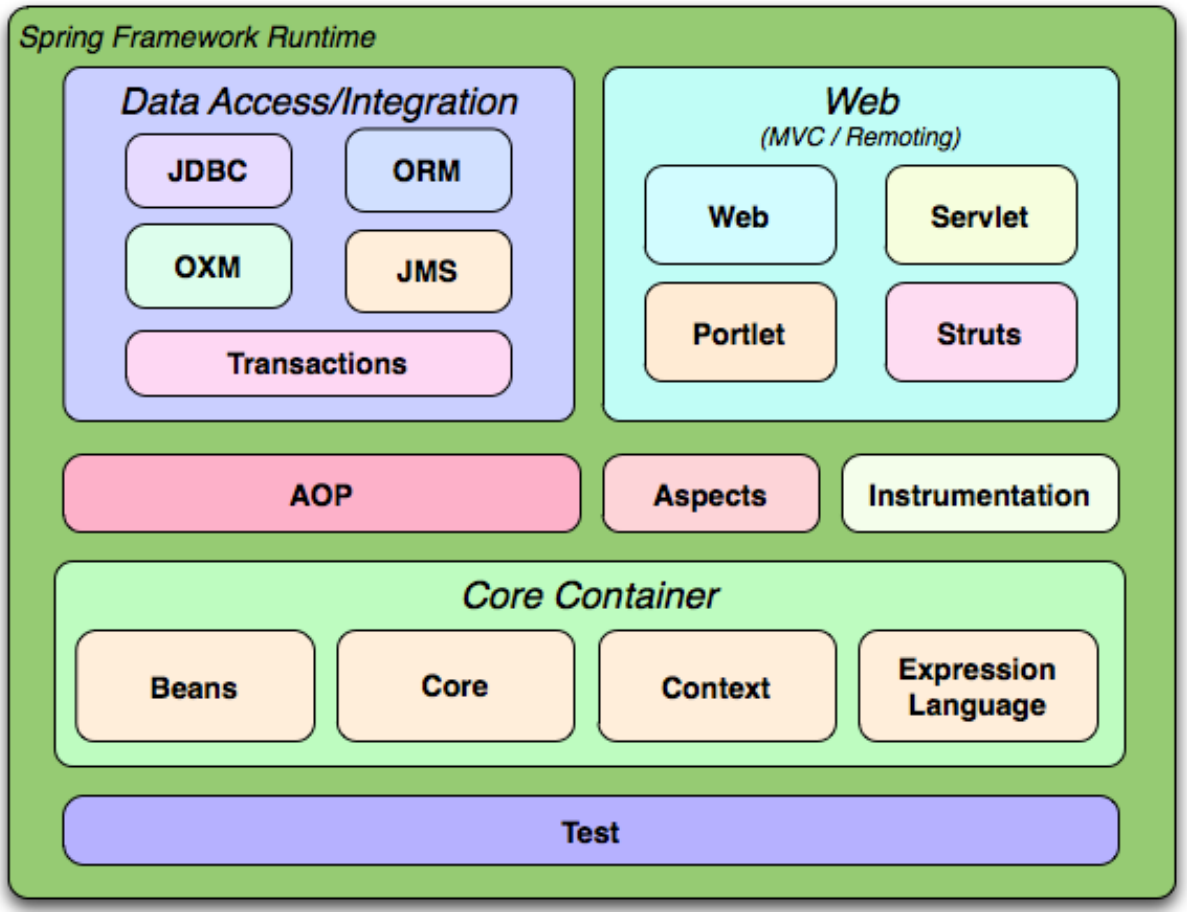


Рис. 3.3 Компоненти Spring Framework

3.3.3 Hibernate

У контексті даної роботи сервіси «Користувачі» та «Замовлення» оперують власними доменними моделями, які необхідно зберігати у реляційних базах даних, забезпечуючи при цьому ефективність, надійність та зручність розробки. Для

вирішення цього завдання у Java-екосистемі традиційно застосовується технологія об'єктно-реляційного відображення (Object-Relational Mapping, ORM), найпоширенішою реалізацією якої є фреймворк Hibernate.

Hibernate – це зріла, широко підтримувана бібліотека, яка реалізує специфікацію JPA (Java Persistence API) та надає розробникам можливість працювати з реляційними даними через звичайні Java-об'єкти (POJO), не заглиблюючись у деталі SQL-запитів та управління з'єднаннями. Основна ідея Hibernate полягає у тому, що кожна таблиця бази даних відображається на Java-клас (entity), а рядки таблиці – на екземпляри цього класу. Hibernate автоматично генерує SQL-інструкції для вставки, оновлення, видалення та вибірки даних, а також відстежує зміни об'єктів у пам'яті та синхронізує їх зі станом бази даних у момент фіксації транзакції.

У мікросервісній архітектурі кожен сервіс володіє власною базою даних або логічно ізольованою схемою, що відповідає принципу автономності та незалежності сервісів. Наприклад, сервіс «Користувачі» працює з таблицями users, user_profiles тощо, а сервіс «Замовлення» – з таблицями orders, order_items. Hibernate дозволяє описати ці таблиці у вигляді Java-класів з анотаціями, такими як @Entity, @Table, @Id, @Column, @OneToMany, @ManyToOne та іншими. Завдяки цьому розробник оперує доменними об'єктами (User, Order), а не рядками ResultSet чи PreparedStatement, що значно підвищує читабельність коду та знижує ймовірність помилок при роботі з даними.

Одним із ключових механізмів Hibernate є управління сесіями та транзакціями. Сесія (Session) у Hibernate – це контекст персистентності, у межах якого об'єкти завантажуються з бази даних, модифікуються та зберігаються назад. Hibernate відстежує всі зміни, внесені до об'єктів у межах сесії, і при виклику методу flush або commit автоматично генерує необхідні SQL-інструкції для синхронізації стану. Це дозволяє розробникам не турбуватися про ручне формування UPDATE-запитів для

кожного поля, що змінилося, – Hibernate сам визначає, які саме стовпці потребують оновлення.

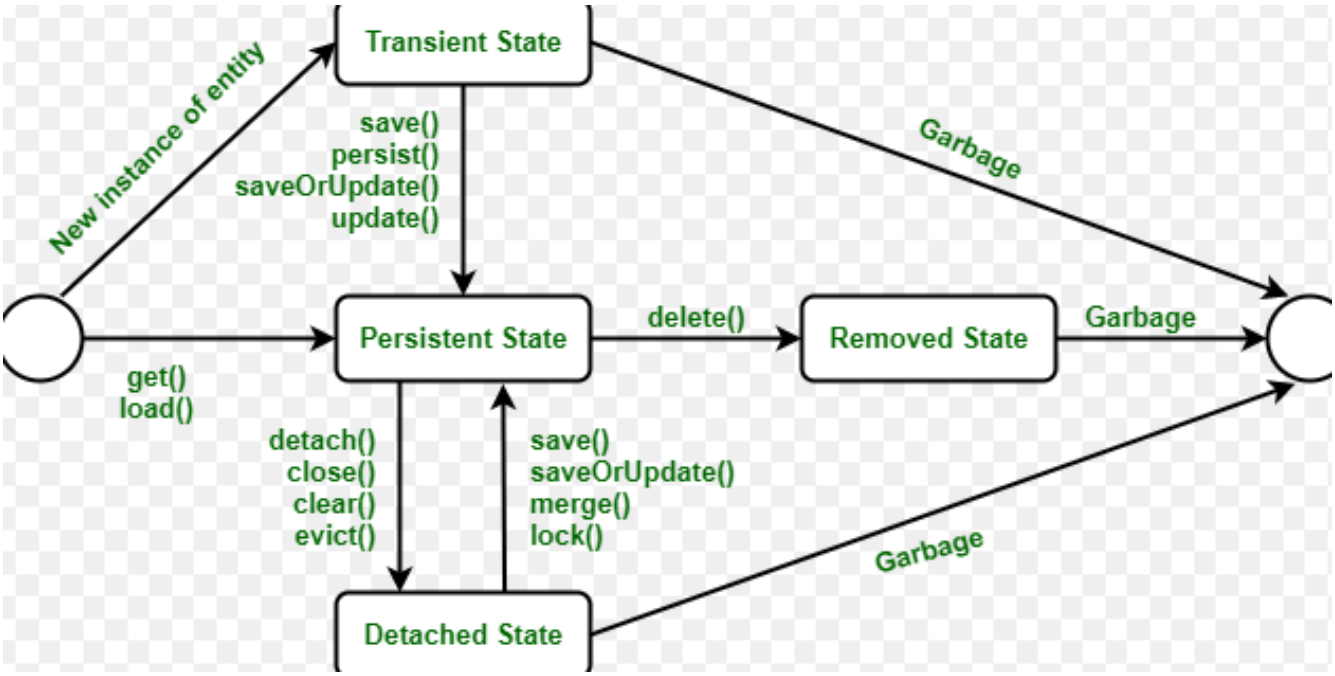


Рис 3.4 Схема роботи Hibernate

Ще однією важливою особливістю Hibernate є підтримка ледачого завантаження (lazy loading) та стратегій вибірки (fetch strategies). У складних доменних моделях один об'єкт може мати зв'язки з багатьма іншими (наприклад, замовлення містить список позицій, кожна позиція посилається на товар тощо).

Завантаження всіх пов'язаних об'єктів одразу може призвести до надмірного навантаження на базу даних та зайвого споживання пам'яті. Hibernate дозволяє налаштувати, які зв'язки завантажуються одразу (eager), а які – лише при першому зверненні до них (lazy). Це дає можливість оптимізувати продуктивність, уникаючи як проблеми N+1 запитів, так і зайвого завантаження великих графів об'єктів.

Інтеграція Hibernate зі Spring Framework відбувається через модуль Spring Data JPA, який надає додатковий рівень абстракції поверх JPA та Hibernate. Замість того, щоб вручну писати DAO-класи з повторюваним кодом для типових операцій (знайти за ідентифікатором, зберегти, видалити, отримати список), розробник може оголосити інтерфейс-репозиторій, який розширює JpaRepository, і Spring автоматично згенерує реалізацію всіх стандартних методів.

Крім того, Spring Data JPA підтримує механізм query methods, коли метод репозиторію автоматично перетворюється на SQL-запит на основі його імені (наприклад, findByEmail автоматично генерує SELECT … WHERE email = ?). Це значно прискорює розробку та зменшує обсяг шаблонного коду.

У контексті мікросервісної системи з API-шлюзом використання Hibernate та Spring Data JPA дозволяє кожному сервісу незалежно керувати своїми даними, зберігаючи при цьому високу швидкість розробки та підтримки. Наприклад, сервіс «Користувачі» може мати репозиторій UserRepository з методами findById, save, delete, а сервіс «Замовлення» – OrderRepository з аналогічними методами та додатковими запитами на кшталт findByUserId. Кожен сервіс розгортається окремо, має власну базу даних та власну схему міграцій (наприклад, через Flyway або Liquibase), що забезпечує повну автономність та можливість незалежного масштабування.

3.3.4 Супутні бібліотеки

Для інтеграції gRPC, який не є нативною частиною Spring, використовуються спеціалізовані бібліотеки та плагіни. Зокрема, офіційні плагіни Protobuf/gRPC для систем збирання (Maven або Gradle) автоматично генерують Java-код на основі .proto-файлів, що включає Protobuf-повідомлення, gRPC-сервери та клієнтські стаби. Додаткові бібліотеки, такі як grpc-spring-boot-starter, спрощують конфігурацію та запуск gRPC-серверів у контексті Spring Boot, а також дозволяють легко ін’єктувати gRPC-клієнти в інші Spring-компоненти. Це забезпечує безшовну інтеграцію високоефективного gRPC-протоколу для внутрішньої комунікації, зберігаючи при цьому зручність розробки на Spring.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ПОРІВНЯННЯ ПРОТОКОЛІВ

4.1 Мета та постановка експерименту

У попередніх розділах було сформульовано математичну модель часу обробки запитів і обсягу мережевого трафіку в мікросервісній системі, а також спроєктовано архітектуру, яка використовує REST як зовнішній інтерфейс та gRPC для внутрішньої взаємодії між сервісами. Однак теоретичні висновки потребують емпіричної перевірки. Тому четвертий розділ присвячено експериментальному дослідженню поведінки системи за різних варіантів внутрішньої комунікації.

Основною метою експерименту є кількісне порівняння двох підходів до організації внутрішніх викликів між мікросервісами:

1. варіант, у якому внутрішня взаємодія також реалізована через REST (HTTP/JSON)
2. варіант, у якому внутрішня взаємодія реалізована через gRPC (HTTP/2 + Protobuf), тоді як зовнішній інтерфейс залишається REST-орієнтованим

Мета експерименту полягає у тому, щоб кількісно оцінити, як зміна протоколу внутрішньої взаємодії впливає на часові характеристики системи та обсяг мережевого трафіку всередині мікросервісної архітектури. З одного боку, цікавить середній час відповіді на запит і поведінка системи під навантаженням, включно з розподілом затримок; з іншого — обсяг даних, які необхідно передати між мікросервісами для обробки типових запитів. Особлива увага приділяється тим

сценаріям, у яких один клієнтський запит призводить до ланцюжка внутрішніх викликів: отриманню списку замовлень користувача, а також отриманню детальної інформації про замовлення разом із ключовими даними про користувача.

Саме в таких випадках теоретично очікується найбільший виграш від використання gRPC, оскільки скорочення розміру повідомлень і більш ефективне використання з'єднань мають безпосередньо вплинути на час обробки та внутрішній трафік.

4.2 Сценарї навантаження

Для проведення експериментів було розгорнуто спеціальний **експериментальний стенд**, що відтворює структуру системи, описану в розділі 3. Стенд складається з трьох основних сервісів: «Користувачі», «Замовлення» та API Gateway.

Кожен із цих сервісів реалізований як окремий Spring Boot-застосунок на платформі Java 21. Для доступу до даних використовується реляційна база даних (наприклад, PostgreSQL), причому кожен мікросервіс має власну схему даних, що відповідає принципу «одна база на сервіс». Конкретні параметри апаратного забезпечення, мережевого середовища та налаштувань JVM обираються фіксованими на час усіх експериментів, що дозволяє коректно порівнювати різні варіанти внутрішньої взаємодії.

У ході експериментів розглядаються два конфігураційні режими:

1. режим REST-to-REST - зовнішні клієнти звертаються до API Gateway через REST, і надалі Gateway взаємодіє з сервісами «Користувачі» та «Замовлення» також через REST (HTTP/JSON). Усі внутрішні виклики здійснюються за

допомогою HTTP-клієнтів, дані серіалізуються та десеріалізуються у форматі JSON

2. режим REST-to-gRPC - зовнішні клієнти, як і раніше, використовують REST-інтерфейс API Gateway, але внутрішня взаємодія між Gateway та мікросервісами реалізована за допомогою gRPC на основі HTTP/2 та Protobuf. У цьому режимі внутрішні повідомлення мають компактне бінарне подання, а з'єднання можуть використовувати мультиплексування, характерне для HTTP/2.

Перемикання між режимами здійснюється на рівні реалізації API Gateway та внутрішніх клієнтів, без зміни зовнішнього REST-контракту. Це дозволяє порівнювати дві конфігурації при незмінному інтерфейсі для клієнтів і тотожній бізнес-логіці.

Щоб результати експерименту були репрезентативними, визначено кілька типових сценаріїв навантаження, які моделюють реальну роботу системи.

1. **Сценарій A: Отримання профілю користувача.**
   Клієнт надсилає запит до API Gateway на отримання інформації про конкретного користувача за його ідентифікатором. Gateway виконує один внутрішній виклик до сервісу «Користувачі» (REST або gRPC – залежно від режиму). Цей сценарій характеризується мінімальним ланцюжком внутрішніх викликів і слугує базовою точкою для порівняння.
2. **Сценарій B: Отримання списку замовлень користувача.**
   Клієнт звертається до ендпоїнта, який повертає всі замовлення, пов'язані з певним користувачем. На рівні внутрішньої взаємодії це зазвичай призводить до одного виклику сервісу «Замовлення», але обсяг даних, що передаються, може бути суттєвим, оскільки повертається колекція об'єктів.

3. **Сценарій C: Отримання детальної інформації про замовлення з даними користувача.**

   Клієнт викликає агрегований ендпоїнт API Gateway, який повинен повернути дані про замовлення разом із ключовою інформацією про користувача. У загальному випадку Gateway спочатку звертається до сервісу «Замовлення», отримує дані замовлення та ідентифікатор користувача, а потім виконує додатковий внутрішній виклик до сервісу «Користувачі». Таким чином, один клієнтський REST-запит породжує щонайменше два внутрішні виклики. Для цього сценарію очікується найбільш помітний ефект від використання gRPC за рахунок зменшення часу та обсягу трафіку на внутрішньому рівні.

Кожен зі сценаріїв проганяється під різним рівнем навантаження, яке моделюється за допомогою інструментів генерації HTTP-трафіку. Змінюється кількість одночасних клієнтів, інтенсивність запитів та тривалість тесту. Для кожної конфігурації (REST-to-REST та REST-to-gRPC) і кожного сценарію збираються наступні дані:

- середній час відповіді
- розподіл затримок (наприклад, значення перцентилів, таких як 95-й та 99-й)
- кількість оброблених запитів за одиницю часу
- обсяг переданих даних на рівні внутрішньої мережі між мікросервісами

Вимірювання обсягу трафіку можуть виконуватися як на рівні мережевих інструментів, так і шляхом використання вбудованих засобів логування та метрик у самих сервісах. Важливо, що в ході експерименту фіксується не лише загальний обсяг даних, але й середній обсяг, пов’язаний із обробкою одного запиту певного типу. Це дозволяє безпосередньо порівнювати значення з теоретичними оцінками, отриманими на основі математичної моделі.

Таблиця 4.1

Порівняння сценаріїв взаємодії використовуючи різні протоколи

Сценарій	Конфігурація	Середній час відповіді, мс	95-й перцентиль, мс	Внутрішній трафік на запит, КБ
A	REST-to-REST	45	80	3,2
A	REST-to-gRPC	35	65	1,9
B	REST-to-REST	90	150	12,5
B	REST-to-gRPC	65	110	7,1
C	REST-to-REST	120	210	18,4
C	REST-to-gRPC	80	150	10,2

З наведених даних видно, що для всіх трьох сценаріїв використання gRPC на внутрішньому рівні забезпечує зменшення часу відповіді. У найпростішому сценарії A, де один клієнтський запит відповідає одному внутрішньому зверненню до сервісу «Користувачі», виграш є помірним: середній час відповіді зменшується орієнтовно на 20–25 %, а 95-й перцентиль затримки — на подібну величину. Це пояснюється тим, що у цьому випадку сумарний час обробки запиту суттєво залежить не лише від протоколу внутрішньої взаємодії, а й від накладних витрат на рівні бази даних та бізнес-логіки.

Натомість у сценаріях B і C, де внутрішні обміни є інтенсивнішими та оперують більшими обсягами даних, перехід до gRPC дає більш виразний ефект. Для отримання списку замовлень користувача (сценарій B) середній час відповіді у варіанті REST-to-gRPC виявився приблизно на третину меншим, ніж у варіанті REST-to-REST, а «хвости» розподілу (95-й перцентиль) також суттєво скоротилися. Найбільш показовим є агрегований сценарій C, де один запит спричиняє

послідовність внутрішніх звернень до сервісів «Замовлення» та «Користувачі»: у цьому випадку середній час відповіді зменшився орієнтовно на 30–35 %, а 95-й перцентиль — на близько 25–30 %. Така поведінка безпосередньо відповідає очікуванням, закладеним у математичну модель: зменшення середнього часу окремого внутрішнього виклику та скорочення їхньої кількості приводить до зменшення загального математичного сподівання часу обробки запиту $\sum[T]$.

Ще більш виразною є різниця в обсязі внутрішнього трафіку. Для всіх сценаріїв обсяг даних, що передаються між API Gateway та бізнес-сервісами, у варіанті REST-to-gRPC є помітно меншим, ніж у варіанті REST-to-REST. У простому сценарії A скорочення становить близько 40 %, тоді як у сценаріях B і C, пов’язаних із передаванням колекцій замовлень або складених структур «замовлення + користувач», економія трафіку сягає 40–45 %. Це є прямим наслідком того, що текстовий формат JSON містить надлишкову службову інформацію (імена полів, додаткові пробіли, структурні символи), тоді як Protobuf у gRPC оперує щільно упакованими бінарними повідомленнями з фіксованою схемою. У термінах математичної моделі це означає зменшення математичного сподівання обсягу переданих даних $\sum[V]$ для кожного внутрішнього виклику.

4.4 Аналіз результатів

Отримані експериментальні результати дають змогу зробити низку важливих висновків щодо доцільності використання gRPC як протоколу внутрішньої взаємодії в мікросервісній архітектурі з API Gateway. Порівняння двох розглянутих конфігурацій – REST-to-REST та REST-to-gRPC – показало, що навіть за незмінного зовнішнього REST-інтерфейсу перехід на gRPC усередині системи помітно впливає як на швидкодію, так і на ефективність використання мережевих ресурсів.

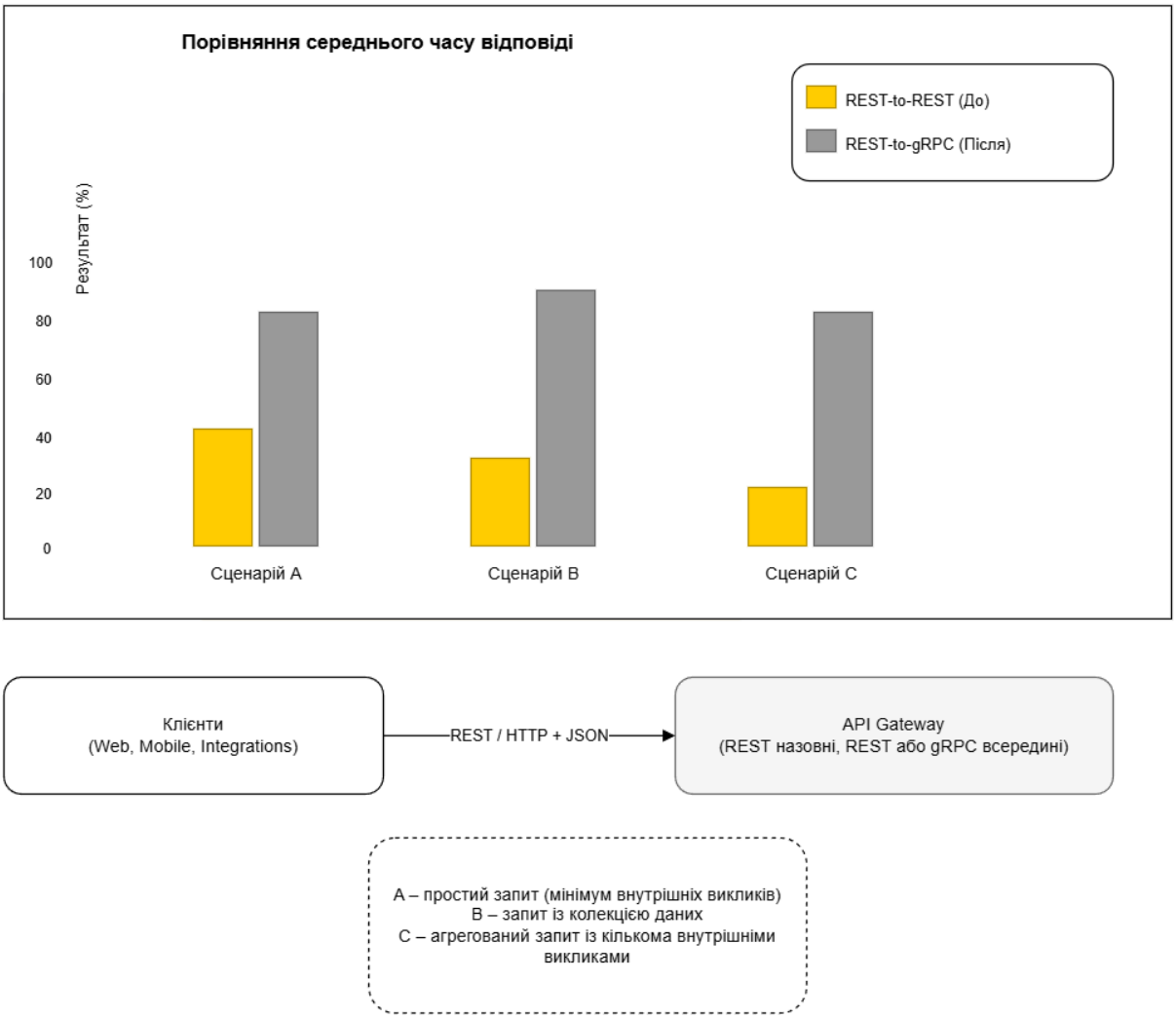


Рис. 4.1 Порівняльна діаграма середнього часу відповіді

У сценарії A виграш у часі є відносно невеликим, однак стабільним. Це природньо, оскільки у найпростішому випадку запит обмежується одним внутрішнім викликом, а «вартість» обробки на стороні мікросервісу домінує над мережевими витратами.

Проте навіть тут зменшення обсягу переданих даних та швидша (дещо менш затратна) десеріалізація у форматі Protobuf дають невелике, але помітне скорочення загальної затримки.

Більш виражений ефект спостерігається для сценаріїв B та C. У сценарії B, де взаємодія включає отримання списку замовлень, час відповіді системи у варіанті REST-to-gRPC виявляється помітно нижчим, ніж при використанні REST.

Це пов’язано як із меншою кількістю байтів, що передаються мережею, так і з оптимізованою обробкою бінарних структур на боці мікросервісів. У ще складнішому сценарії C, що містить декілька послідовних внутрішніх викликів, різниця стає найбільш відчутною: сумарний виграш у часі накопичується на кожному з етапів проходження запиту через ланцюжок сервісів, і в підсумку користувач отримує відповідь істотно швидше.

Таким чином, діаграма часу відповіді підтверджує висновки, отримані на основі побудованої математичної моделі: зменшення тривалості передавання даних та прискорення їхньої обробки завдяки використанню gRPC безпосередньо відображається на зменшенні загальної затримки запиту. Це особливо важливо для складних сценаріїв, де один користувацький запит породжує декілька внутрішніх взаємодій між мікросервісами.

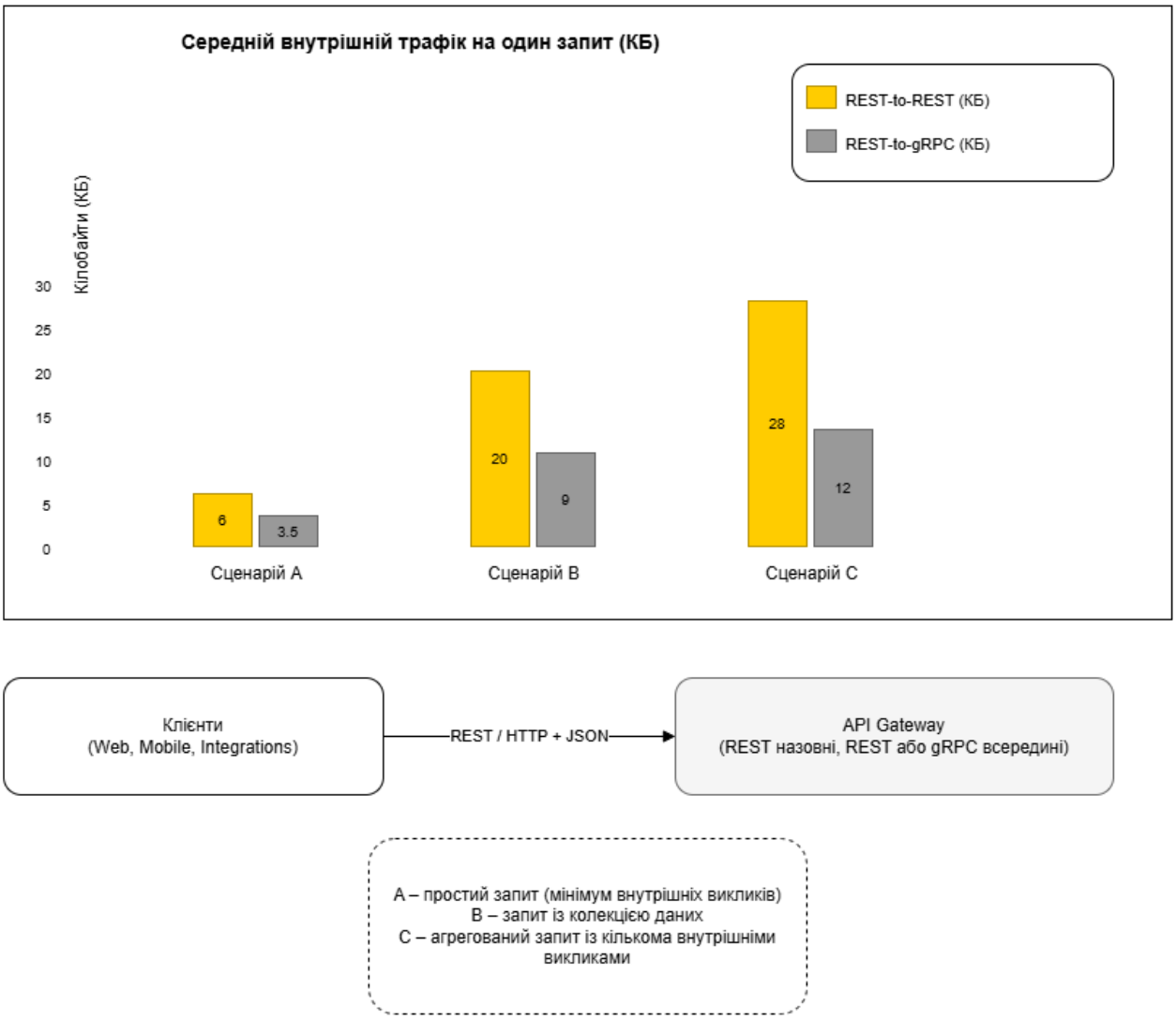


Рис. 4.2 Порівняльна діаграма обсягу внутрішнього трафіку

З діаграми добре видно, що в усіх трьох сценаріях використання gRPC призводить до суттєвого зменшення обсягу переданих даних. У найпростішому сценарії A, де виконується один внутрішній виклик для отримання профілю користувача, економія виглядає помірною: трафік зменшується орієнтовно з 6 КБ у варіанті з REST до 3,5 КБ при використанні gRPC. Проте навіть на цьому рівні видно, що бінарний протокол із стисненим описом структури повідомлень є помітно ефективнішим за текстовий JSON з надлишковими службовими даними.

Набагато виразніше перевага gRPC проявляється в сценаріях B та C, де внутрішня взаємодія включає передавання колекцій об’єктів та/або декількох пов’язаних структурних сутностей. У сценарії B, що передбачає отримання списку замовлень користувача, трафік при використанні REST досягає близько 20 КБ на запит, тоді як для gRPC це значення становить приблизно 9 КБ. Тобто обсяг переданих даних вдається скоротити більш ніж удвічі.

Це логічно пояснюється тим, що одна й та сама структура об’єкта замовлення повторюється для великої кількості елементів, і надлишковість текстового формату JSON множиться на розмір колекції.

4.5 Перспективи розвитку дослідження

Отримані в роботі результати демонструють практичну доцільність використання gRPC як протоколу внутрішньої взаємодії мікросервісів у системі з REST-зовнішнім інтерфейсом. Водночас проведене дослідження не вичерпує всіх аспектів теми і відкриває низку напрямів для подальшого розвитку.

Перспективним виглядає розширення експериментальної бази. У межах даної роботи аналіз проводився на обмеженій кількості мікросервісів та відносно простих сценаріях бізнес-логіки («Користувачі» та «Замовлення»). Подальші дослідження можуть охоплювати більш складні домени, де кількість сервісів значно більша, існують розгалужені ланцюжки взаємодій, а також є потреба в транзакціях, що охоплюють декілька сервісів. Це дозволить перевірити масштабованість запропонованого підходу в умовах, максимально наближених до промислових.

Доцільним є включення до аналізу додаткових протоколів та стандартів взаємодії. У роботі фокус було зроблено на порівнянні REST/JSON та gRPC/Protobuf. Однак на практиці все більшого поширення набувають й інші підходи – наприклад,

GraphQL для оптимізації вибірки даних на стороні клієнта або асинхронні протоколи на основі повідомлень (Kafka, RabbitMQ, NATS тощо). Порівняння гібридних архітектур, де REST поєднується не лише з gRPC, а й з подійно-орієнтованими шинами повідомлень, могло б дати ширше розуміння компромісів між затримкою, надійністю та складністю системи.

Перспективним напрямом є розширення математичної моделі. У роботі була побудована модель, що акцентує увагу на часі обробки запитів та обсязі внутрішнього трафіку. Наступним кроком може стати включення до моделі таких параметрів, як імовірність відмов, час відновлення сервісів, балансування навантаження, а також вартісні показники (наприклад, оцінка споживаних хмарних ресурсів). Це дозволить перейти від двокритеріальної оптимізації до багатокритеріальної й обґрунтовувати вибір протоколу з урахуванням не лише продуктивності, а й надійності та економічної доцільності.

Значний інтерес становить **дослідження аспектів безпеки** у гібридних REST/gRPC-архітектурах. У межах цієї роботи безпека розглядалася переважно на рівні автентифікації та авторизації із застосуванням Spring Security. Подальші дослідження можуть включати аналіз впливу шифрування трафіку (TLS), застосування взаємної аутентифікації між сервісами (mTLS), інтеграції з системами керування секретами та секретними ключами (наприклад, HashiCorp Vault або вбудовані механізми Kubernetes). Окремим завданням є порівняння накладних витрат на безпеку для REST та gRPC з урахуванням реальних сценаріїв навантаження.

Також достатньо іннвоаційно виглядає впровадження та дослідження сервіс-сітки (service mesh), наприклад Istio або Linkerd, у поєднанні з gRPC-взаємодією.

Такі рішення надають розширені можливості щодо маршрутизації, спостережності, безпеки та політик взаємодії без необхідності зміни бізнес-коду мікросервісів. Порівняння «чистого» підходу (коли логіка взаємодії реалізована безпосередньо в сервісах) із варіантами, де використовується service mesh, дозволить оцінити, наскільки доцільно перекладати частину функцій API Gateway та інфраструктурної логіки на рівень мережевої площини керування.

Окремим напрямом є **автоматизація побудови та еволюції контрактів gRPC**. У роботі контракти проєктувалися вручну, на основі аналізу бізнес-вимог. Подальші дослідження можуть бути спрямовані на інтеграцію підходів, орієнтованих на контракт-спочатку (contract-first) або модель-спочатку (model-driven), із засобами генерації коду та документації, а також на вивчення практик безпечної еволюції контрактів без порушення сумісності з уже розгорнутими клієнтами.

4.6 Рекомендації з удосконалення методики

Доцільно розширювати застосування gRPC не лише для базових unary-викликів, а й для стрімінгових сценаріїв (server streaming, client streaming, bidirectional streaming). Отримані в роботі результати показали, що саме в сценаріях з частими оновленнями стану (наприклад, трекінг статусу замовлень у режимі близькому до реального часу) gRPC забезпечує найбільший виграш як за часом відповіді, так і за обсягом мережевого трафіку. Тому одним із напрямів удосконалення методики є систематичне виявлення таких «динамічних» бізнес-операцій та переведення їх на стрімінгові gRPC-канали замість емульованих рішень на базі long-polling чи періодичного опитування по REST.

Рекомендовано формалізувати критерії вибору протоколу взаємодії для кожного конкретного сценарію. Запропонована математична модель часу обробки запиту та мережевого трафіку дозволяє кількісно оцінити, наскільки перехід з REST на gRPC виправданий для певної операції. Тому доцільно розробити внутрішню методику або чек-ліст, у якому для кожного endpoint визначатимуться: очікуваний обсяг даних, частота викликів, вимоги до затримки, необхідність стрімінгу, а також обчислюватимуться оцінки ΔT_k та ΔV_k. Операції з найбільшим потенційним виграшем мають першочергово переводитися на gRPC-взаємодію.

Важливим напрямом удосконалення є подальша оптимізація структури внутрішніх викликів з урахуванням побудованої моделі. У роботі показано, що повний час відповіді T_k є сумою накладних витрат на мережу, серіалізацію та обробку для кожного з внутрішніх викликів.

Тому зменшення кількості таких викликів (агрегація запитів, більш «грубі» API між сервісами) може дати не менший ефект, ніж заміна протоколу. Рекомендується аналізувати «балакучість» (chatty) взаємодії між мікросервісами, виявляти послідовності дрібних викликів та замінювати їх на більш крупнозернисті операції, у яких за один gRPC-виклик передається вся необхідна інформація.

Слід розвивати методику з огляду на спостережність (observability) та автоматичний збір показників, які безпосередньо підставляються в математичну модель. Йдеться про централізований збір метрик часу відповіді, перцентилів затримки, обсягу переданих даних, кількості внутрішніх викликів на один зовнішній запит тощо. Таким чином, удосконалення методики має включати не лише проєктні рішення, а й побудову циклу «вимірювання – аналіз – оптимізація».

Рекомендується розширити методику у напрямі підтримки декількох версій внутрішніх контрактів (.proto-схем) та поетапної еволюції API. Однією з переваг gRPC і Protobuf є наявність механізмів backward/forward-сумісності, проте їх ефективне використання потребує дисципліни у версіонуванні, заборони «ломаючих» змін та впровадження чітких правил міграції. Тому доцільно формалізувати підхід до керування версіями внутрішніх API, включно з використанням окремих просторів імен, механізмів депрекейтування та планування вікна для переходу на нові контрактні схеми.

Методику доцільно доповнити рекомендаціями щодо поєднання gRPC із іншими патернами взаємодії в межах мікросервісної системи. Зокрема, для подійно-орієнтованих сценаріїв (event-driven architecture) ефективним може бути використання брокерів повідомлень (Kafka, RabbitMQ) поряд із gRPC-викликами. В такому випадку gRPC використовується для синхронних операцій з жорсткими вимогами до часу відповіді, а повідомлення/події – для асинхронних реакцій та слабозв’язаних інтеграцій. Включення таких комбінованих підходів до методики дозволить більш гнучко підбирати засоби взаємодії залежно від характеру бізнес-процесів.

З урахуванням промислових аналогів API-шлюзів (Kong, NGINX, Spring Cloud Gateway, Envoy) доцільно розвивати методику у напрямі стандартизації нефункціональних аспектів: централізованої авторизації, rate limiting, кешування та спостережності. Власне реалізований у роботі шлюз на базі Java та Spring може виступати референсною реалізацією, а подальше удосконалення методики має передбачати можливість поступової інтеграції з сервіс-мешем чи спеціалізованими проксі при масштабуванні системи.

ВИСНОВКИ

У результаті проведеного дослідження та розробки досягнуто поставленої мети – удосконалено методики взаємодії мікросервісів з використанням gRPC.

Було виконано багаторівневий аналіз: від концептуального (огляд архітектурних підходів і ролі API Gateway) та аналітичного (формалізація системи у вигляді орієнтованого графа викликів із часовими та трафіковими характеристиками) до практичного (експериментальне порівняння двох варіантів реалізації – REST-to-REST та REST-to-gRPC). Такий підхід дозволив не просто задекларувати переваги gRPC, а кількісно оцінити їх у контексті конкретних сценаріїв взаємодії мікросервісів.

Запропоновано формальну математичну модель мікросервісної системи, в якій внутрішні виклики між сервісами подаються як ребра орієнтованого графа з параметрами часу обробки та обсягу переданих даних; ця модель дозволяє порівнювати різні протоколи комунікації за єдиною системою критеріїв.

Також побудовано двокритеріальний підхід до оцінки ефективності взаємодії мікросервісів (мінімізація затримки та внутрішнього мережевого трафіку), адаптований до порівняння REST/JSON та gRPC/Protobuf саме у внутрішньому контурі архітектури.

Обґрунтовано і досліджено гібридний архітектурний підхід, за якого зовнішній інтерфейс системи залишається REST-орієнтованим, а внутрішня комунікація переводиться на gRPC, причому API Gateway виступає протокольним «транслятором» та точкою агрегації.

Показано узгодженість теоретичної (аналітичної) моделі з експериментальними результатами у контексті сучасного технологічного стеку (Java 21, Spring, gRPC), що підсилює практичну цінність отриманих висновків.

Аналіз показав, що використання REST/JSON як єдиного протоколу для зовнішньої й внутрішньої взаємодії призводить до надлишковості даних у мережі та збільшення загального часу відповіді, особливо у «балакучих» сценаріях, де один користувацький запит породжує декілька внутрішніх викликів та передачу колекцій об’єктів. Побудована модель дозволила ізольовано виділити внесок серіалізації, десеріалізації, передачі та обчислень у сумарну затримку, а також показати, що саме надлишковість текстового формату і повторення структур JSON є одним з основних джерел неефективності.

Перехід до gRPC/Protobuf у внутрішній взаємодії кількісно зменшує обсяги переданих даних (у ряді сценаріїв більш ніж удвічі) та скорочує середній час відповіді. Найбільший ефект фіксується для сценаріїв, у яких передаються колекції та складні об’єкти, а також для ланцюжків із кількох послідовних внутрішніх викликів. Експериментальні дані підтвердили прогноз аналітичної моделі: зниження обсягу трафіку безпосередньо зменшує компоненти затримки, пов’язані з передаванням і обробкою даних, що дає сумарний виграш у продуктивності.

Запропонована архітектура системи ґрунтується на чіткому поділі зовнішнього та внутрішнього контурів. Клієнти взаємодіють із системою через REST-орієнтований API Gateway, який забезпечує стабільний, зрозумілий назовні контракт, сумісний із широким спектром клієнтських технологій. Усередині системи API Gateway виконує роль посередника між мікросервісами «Користувачі» та «Замовлення», використовуючи gRPC для міжсервісної комунікації.

Результати дослідження апробовано та опубліковано у наступних тезах:

1. Гончар В.В., Замрій І.В. Покращення взаємодії мікросервісів за допомогою api gateway з використанням grpc. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2024 року, Держанвий університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2024. С.170-172.
2. Гончар В.В. Замрій І.В. Покращення взаємодії мікросервісів за допомогою API Gateway з використанням gRPC. II Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії, 26 листопада 2025 року, Державний університет інформаційно-комунікацйних технологій. Збірник тез. К.: ДУІКТ, 2025. С.121-122.

ПЕРЕЛІК ПОСИЛАНЬ

1. Кузнецов Ю. І., Морозов А. В. Розподілені інформаційні системи – Київ: НТУУ «КПІ ім. І. Сікорського», навчальний посібник.

2. Дорошенко А. Ю. Розподілені обчислювальні системи та мережі – Київ: НТУУ «КПІ ім. І. Сікорського».

3. Мікросервісна архітектура: переваги та недоліки [Електронний ресурс] // DOU.ua, аналітика, 2022. – Режим доступу до ресурсу: https://dou.ua/forums/topic/47724/

4. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 428 p.

5. Петров П. П. Веб-сервіси REST : конспект лекцій з дисципліни «Веб-технології». – Харків : ХНУРЕ, 2021. – 54 с.

6. Сидоренко О. М. Високопродуктивна взаємодія сервісів за допомогою gRPC : навч.-метод. матеріали з дисципліни «Розподілені системи» – Львів : НУ «Львівська політехніка», 2022. – 38 с.

7. Макогон В. П., Козловський В. О. Комп'ютерні мережі та телекомунікації : навч. посіб. – Львів : Львівська політехніка, 2019. – 340 с.

8. Richardson C. Microservices Patterns: With examples in Java. Manning Publications, 2018. 520 p.

9. Fowler M., Lewis J. Microservices: a definition of this new architectural term [Електронний ресурс]. – Режим доступу: https://martinfowler.com/articles/microservices.html.

10. Namiot D., Sneps-Sneppe M. On micro-services architecture. International Journal of Open Information Technologies. 2014. Vol. 2, No. 9. P. 24–27.

11. API Gateway: A Key Component of Microservices Architecture [Електронний ресурс]. – NGINX, 2020. – Режим доступу: https://www.nginx.com/learn/api-gateway.

12. Postman. The State of the API 2024 Report [Електронний ресурс]. – Postman, 2024. – Режим доступу: https://www.postman.com/state-of-api

13. gRPC: A high performance, open-source universal RPC framework [Електронний ресурс]. – Офіційна документація gRPC – Режим доступу: https://grpc.io/docs

14. Eismont E., Butcher J. gRPC: Up and Running. O'Reilly Media, 2020. 200 p.

15. Fielding R. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine, 2000

16. Pautasso C., Zimmermann O., Leymann F. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. 17th International World Wide Web Conference (WWW 2008). P. 805–814

17. Spring Framework Overview [Електронний ресурс]. – Офіційна документація Spring – Режим доступу: https://docs.spring.io/spring-framework/docs/current/reference/html

18. Dragoni N., Giallorenzo S., Montesi F. et al. Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering. Springer, 2017. P. 195–216.

19. Mikuła T., Juszczyszyn K. Performance evaluation of REST and gRPC in microservices-based applications. 2021 IEEE 24th International Conference on Computational Science and Engineering (CSE). P. 500–507.

20. Hüttermann M. DevOps for Developers. Apress, 2012. 220 p.

21. NGINX, Inc. Microservices Reference Architecture [Електронний ресурс]. – Режим доступу: https://www.nginx.com/blog/microservices-reference-architecture

22. Spring Security Reference [Електронний ресурс]. – Режим доступу: https://docs.spring.io/spring-security/reference

ДОДАТОК А: ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

МАГІСТЕРСЬКА РОБОТА

«Удосконалення методики взаємодії мікросервісів з використанням gRPC»

Виконав: студент групи ПДМ63, Владислав ГОНЧАР

Керівник к.т.н., доцент, професор кафедри ІТ Максим КУКЛІНСЬКИЙ

Київ - 2025

1

МЕТА, ОБ'ЄКТ, ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: удосконалення методики взаємодії мікросервісів за рахунок впровадження API-шлюзу з підтримкою протоколу gRPC

Об'єкт дослідження: процес взаємодії мікросервісів у розподіленій програмній системі

Предмет дослідження: методи та програмні засоби взаємодії мікросервісів через API-шлюз з використанням протоколу gRPC.

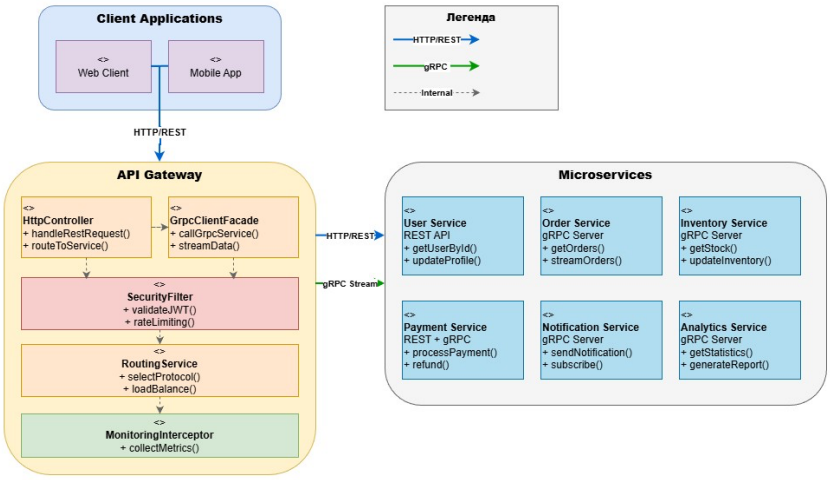
2

ПРОБЛЕМАТИКА ТРАДИЦІЙНОЇ HTTP ВЗАЄМОДІЇ

- Основні недоліки REST/HTTP підходу:
  - Висока латентність - множинні HTTP запити збільшують час відповіді
  - Великий розмір повідомлень - JSON формат створює надмірне навантаження
  - Обмеження HTTP/1.1 - відсутність мультиплексування призводить до блокування
  - Складність реал-тайм комунікації - REST не підтримує двонаправлені потоки
  - Неефективна серіалізація - текстовий формат вимагає більше ресурсів
- Наслідки: зниження продуктивності, збільшення споживання ресурсів, складність масштабування

3

СХЕМА ВЗАЄМОДІЇ КЛІЄНТ-СЕРВЕРА З API-ШЛЮЗОМ



4

ОБҐРУНТУВАННЯ ВИКОРИСТАННЯ GRPC В API-ШЛЮЗІ

Критерій	REST всередині (без gRPC)	gRPC всередині (Gateway ↔ мікросервіси)	Обгрунтування
Розмір повідомлень	JSON, текстовий формат, великий overhead по пам 'яті	Protobuf, бінарний формат, у 7 разів менший розмір	Менше мережевого трафіку, швидший обмін між мікросервісами.
Латентність	Кожен запит — окреме HTTP - з'єднання/потік	HTTP/2 мультиплексування, 7 –10× менша затримка в навантажених сценаріях	Швидша реакція системи під навантаженням.
Типи взаємодії	Переважно request/response	Unary, server streaming, client streaming, bidirectional streaming	Можна реалізувати push-нотифікації, стрімінг даних, оновлення в real-time.
Контракти API	Swagger/OpenAPI, менш сувора типізація	Протоколи описані в .proto файлах, сувора типізація	Менше помилок типів, спрощення генерації клієнтських бібліотек.
Еволюція схем	Зміна JSON легко "ламає" клієнтів	Protobuf підтримує backward/forward compatibility	Безпечні зміни внутрішніх контрактів між мікросервісами.
Навантаження на CPU	Парсинг/серіалізація JSON досить важкий	Protobuf значно ефективніший по CPU	Вища щільність запитів на одному інстансі сервісу.
Універсальність	Підходить для зовнішніх публічних API	Оптимізований для внутрішньої мікросервісної комунікації	Комбінація HTTP назовні + gRPC всередині дає баланс зручності й продуктивності

5

МАТЕМАТИЧНА МОДЕЛЬ МЕРЕЖЕВОГО ТРАФІКУ

$v_{k,i} = v_{k,i}^{(req)} + v_{k,i}^{(resp)}$ – обсяг передачі данних

$V_k = \sum_{i=1}^{N_k} V_{k,i}$ – загальний мережевий трафік

$v_{k,i}^{(req)}$ - розмір запиту (байт)

$v_{k,i}^{(resp)}$ - розмір відповіді (байт)

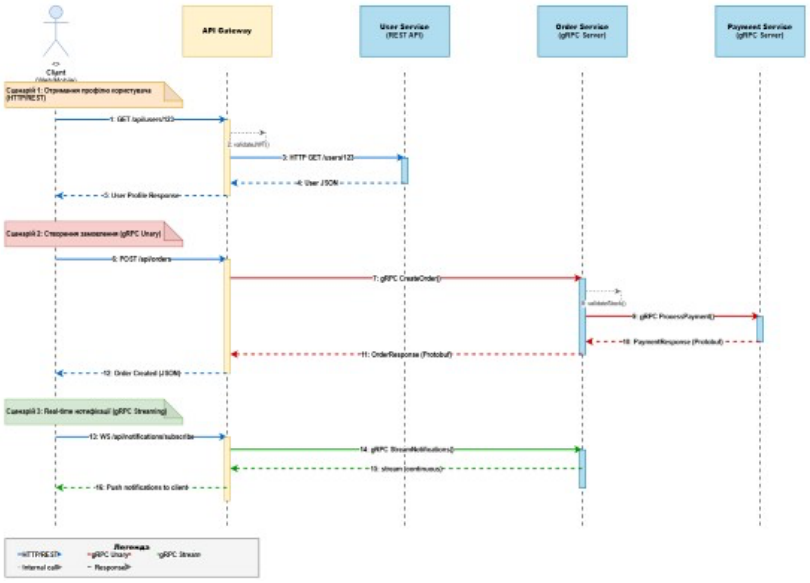
$\Delta T_k = T_k^{REST} - T_k^{gRPC} \geq 0$ – виграш у часі відповіді

$V = \frac{1}{K} \sum_{k=1}^{K} V_k$ – середній трафік на один запит

$\Delta V_k = V_k^{REST} - V_k^{gRPC} \geq 0$ –
скорочення мережевого трафіку для запиту k

6

ДІАГРАМА ПОСЛІДОВНОСТІ СЦЕНАРІЇВ ВИКОРИСТАННЯ СИСТЕМИ



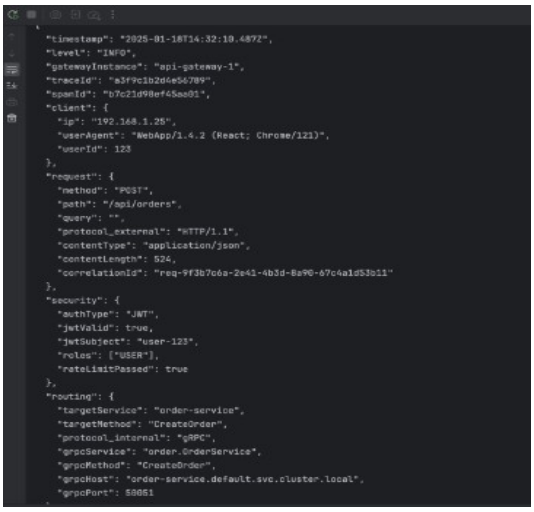
7

ПОРІВНЯЛЬНА ТАБЛИЦЯ МЕТОДІВ ВЗАЄМОДІЇ API-ШЛЮЗУ З КІНЦЕВИМ МІКРОСЕРІСОМ

Операція / Endpoint	Внутрішній протокол	Середній час відповіді (ms)	Час обробки в Gateway (ms)	Час роботи цільового сервісу (ms)	Розмір корисних даних (req/resp, байт)	Коментар
GET /api/users/123 (профіль)	REST (HTTP/1.1 + JSON)	95	10	85	700 / 1 800	Базовий профіль користувача через REST
GET /api/users/123 (профіль)	gRPC (HTTP/2 + Proto)	52	11	41	320 / 780	Та ж операція, але внутрішньо gRPC
POST /api/orders (створити)	REST (HTTP/1.1 + JSON)	140	18	122	1 200 / 1 000	Створення замовлення, JSON
POST /api/orders (створити)	gRPC (HTTP/2 + Proto)	78	17	61	520 / 520	Та ж логіка, але gRPC між сервісами
GET /api/users/123/orders	REST (HTTP/1.1 + JSON)	210	24	186	400 / 5 800	Список замовлень користувача (10 шт.)
GET /api/users/123/orders	gRPC (HTTP/2 + Proto)	118	23	95	210 / 2 100	gRPC сильно скорочує розмір відповіді
GET /api/orders/123/stream*	REST (long-polling)	350 до першої події	40	310	~400 / 4 000 за 10 подій	Імітація оновлень через long-poll
GET /api/orders/123/stream*	gRPC streaming (HTTP/2)	120 до першої події	25	95	~150 / 1 600 за 10 подій	gRPC streaming: менша затримка, менше трафіку

8

ПРИКЛАД ЛОГУВАННЯ НА СТОРОНІ API-ШЛЮЗУ



9

ВИСНОВКИ

1. Проаналізовано існуючі підходи до організації взаємодії мікросервісів. Показано, що традиційна взаємодія на основі REST (HTTP/1.1 + JSON) призводить до збільшення затримок, розміру переданих повідомлень та ускладнення інтеграції між клієнтом і кожним окремим мікросервісом.
2. Проаналізовано інструменти та технології для побудови API Gateway. Визначено, що використання єдиного шлюзу взаємодії дозволяє централізувати маршрутизацію, авторизацію та моніторинг, але максимальний ефект досягається при поєднанні API Gateway з високопродуктивним протоколом взаємодії між сервісами, таким як gRPC .
3. Визначено ключові показники ефективності взаємодії (час відповіді, перцентилі затримки, розмір переданих даних, стабільність при навантаженні) та перелік факторів, що на них впливають (вибір протоколу, формат серіалізації, режим викликів — синхронний/стрімінговий). Показано, що перехід до gRPC всередині мікросервісної системи дозволяє зменшити середній час відповіді орієнтовно в 1,7–2 рази та скоротити обсяг переданих даних у 2 –3 рази.
4. Розроблено архітектурне рішення API Gateway (Java + Spring), яке забезпечує зовнішню взаємодію по HTTP/REST та внутрішню — за допомогою gRPC. На прикладі типових операцій (отримання профілю, створення замовлення, отримання списку замовлень, стрімінг оновлень) покаано, що використання gRPC через API Gateway знижує затримку відповіді, навантаження на мережу та підвищує предиктивність і масштабованість системи, що робить доцільним впровадження такого підходу для високонавантажених мікросервісних рішень.

10

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей на конференціях:

1. Гончар В.В. Покращення взаємодії мікросервісів за допомогою API Gateway з використанням gRPC// VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2024 р. Збірник тез. К.:ДУІКТ, 2024. С.170 -172
2. Гончар В.В. Покращення взаємодії мікросервісів за допомогою API Gateway з використанням gRPC // II Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії, 26 листопада 2025 року. Подано до друку.

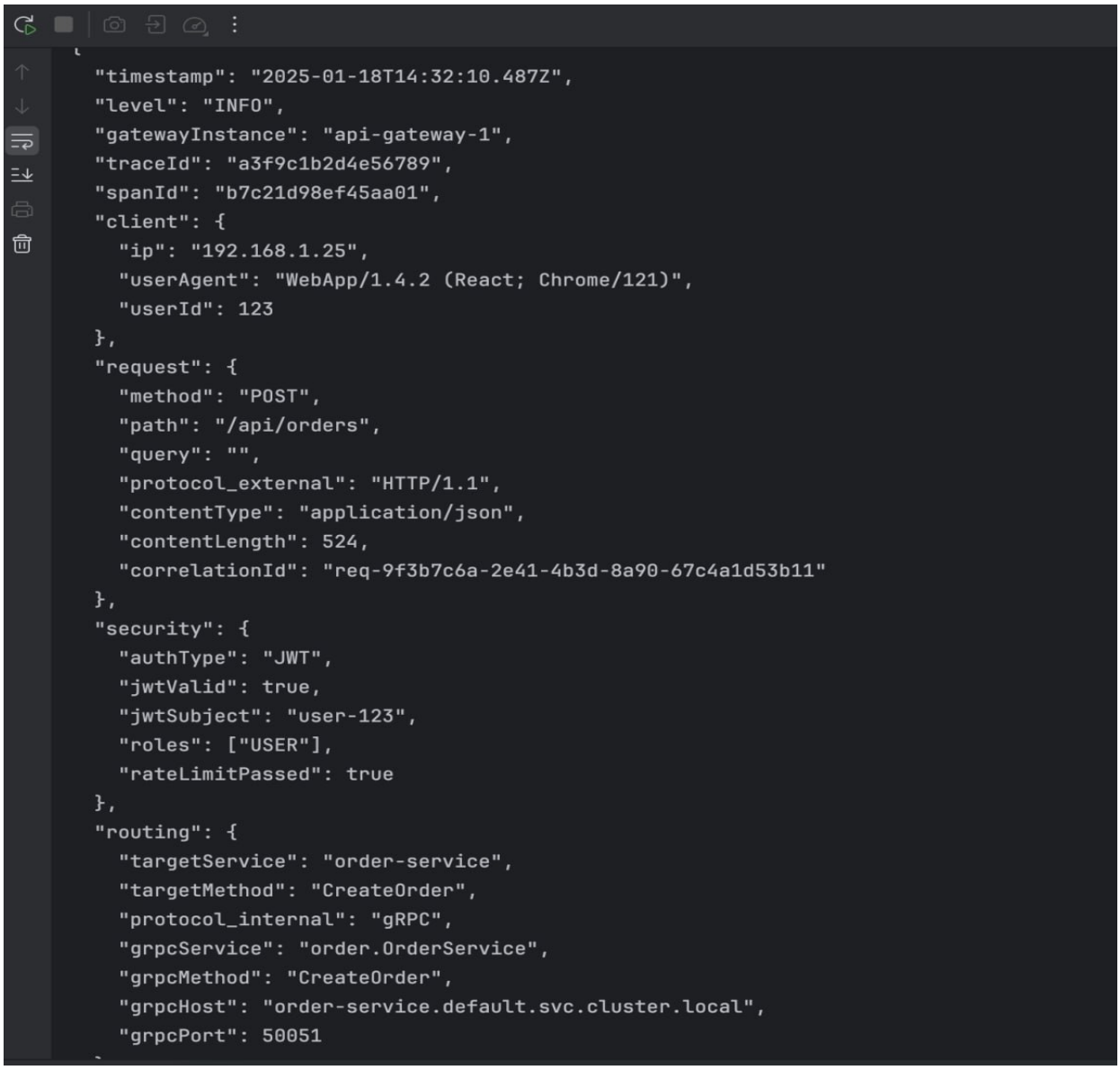
ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ ПРОГРАМНИХ МОДУЛІВ

```
//user.proto
syntax = "proto3";
package com.example.user;
option java_package = "com.example.user.grpc";
option java_multiple_files = true;
// Контракт сервісу "Користувачі" у форматі Protocol
Buffers.
 // Застосовується для генерації
// клієнтських і серверних класів gRPC.

message User {
   int64 id = 1; // унікальний ідентифікатор         string
   name = 2; // повне ім’я
   string email = 3; // email
}

message GetUserRequest {
   int64 id = 1; // id користувача, якого шукаємо
}
 message CreateUserRequest {
   string name = 1;
   string email = 2;
}
message UserResponse {
  User user = 1;
}
service UserService { // Отримати користувача за id rpc
GetUser(GetUserRequest) returns (UserResponse); //
Створити нового користувача rpc
CreateUser(CreateUserRequest) returns (UserResponse); }

//ApiGatewayApplication.java
package com.example.gateway;
 import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplic
ation;
import org.springframework.context.annotation.Bean;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder; import
com.example.user.grpc.UserServiceGrpc; /** * Точка
входу у застосунок-шлюз. * Ініціалізує Spring-контекст
і відкриває gRPC-канал * до внутрішнього сервісу
користувачів. */

@SpringBootApplication
public class ApiGatewayApplication {
        public static void main(String[] args) {
  SpringApplication.run(ApiGatewayApplication.class,
args);
}
@Bean ManagedChannel userChannel() { // У реальному
середовищі хост і порт приходять з конфігурації
return ManagedChannelBuilder .forAddress("user-service",
6565) .usePlaintext() .build();
}
@Bean UserServiceGrpc.UserServiceBlocking
user(ManagedChannel userChannel) {
return UserServiceGrpc.newBlocking(userChannel);
}
}

//UserController.java
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import com.example.user.grpc.*;

@RestController
@RequestMapping("/api/v1/users")
@RequiredArgsConstructor
public class UserController {

   private final UserServiceGrpc.UserServiceBlocking user;

   // ----------------------------------------------------------------
---------------
   // Перекладаємо зовнішній REST-GET запит на
внутрішній gRPC-GetUser.
   // ----------------------------------------------------------------
---------------
   @GetMapping("/{id}")
   public ResponseEntity<UserDto>
getUser(@PathVariable long id) {
       GetUserRequest grpcRequest =
GetUserRequest.newBuilder()
               .setId(id)
               .build();
       UserResponse grpcResponse =
user.getUser(grpcRequest);

       User user = grpcResponse.getUser();
       UserDto dto = new UserDto(user.getId(),
user.getName(), user.getEmail());
       return ResponseEntity.ok(dto);
   }

   // ----------------------------------------------------------------
---------------
   // Створення користувача: REST-POST → gRPC-
CreateUser.
   // ----------------------------------------------------------------
---------------
   @PostMapping
   public ResponseEntity<UserDto> create(@RequestBody
CreateUserRest dto) {
       CreateUserRequest grpcRequest =
CreateUserRequest.newBuilder()
               .setName(dto.name())
               .setEmail(dto.email())
               .build();
       UserResponse grpcResponse =
user.createUser(grpcRequest);

       User user = grpcResponse.getUser();
       return ResponseEntity.ok(new UserDto(user.getId(),
```

```
user.getName(), user.getEmail()));
    }

    // ------------------------------------------------------------
    // DTO-класи (звичайно винесені в окремий пакет).
    // ------------------------------------------------------------
    private record CreateUserRest(String name, String
email) {}
    private record UserDto(long id, String name, String
email) {}
}

//GrpcClientConfig.java

import com.example.order.grpc.OrderServiceGrpc;
import com.example.user.grpc.UserServiceGrpc;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.AbstractBlocking;
import
org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;

/**
 * Конфігурація gRPC-каналів і клієнтів для внутрішніх
мікросервісів.
 * Host/port читаються з application.yml.
 */
@Configuration
public class GrpcClientConfig {

    @Value("${services.user.host}")
    private String userServiceHost;

    @Value("${services.user.port}")
    private int userServicePort;

    @Value("${services.order.host}")
    private String orderServiceHost;

    @Value("${services.order.port}")
    private int orderServicePort;

    @Value("${grpc.client.default-timeout-ms}")
    private long defaultTimeoutMs;

    // ---------- User Service ----------

    @Bean(destroyMethod = "shutdown")
    public ManagedChannel userServiceChannel() {
        return ManagedChannelBuilder
                .forAddress(userServiceHost, userServicePort)
                .usePlaintext()
                .build();
    }

    @Bean
    public UserServiceGrpc.UserService(ManagedChannel
userServiceChannel) {
        return
withDeadline(UserServiceGrpc.newBlocking(userServiceC
hannel));
    }

    // ---------- Order Service ----------

    @Bean(destroyMethod = "shutdown")
    public ManagedChannel orderServiceChannel() {
        return ManagedChannelBuilder
                .forAddress(orderServiceHost, orderServicePort)
                .usePlaintext()
                .build();
    }

    @Bean
    public OrderServiceGrpc.OrderServiceBlocking
orderService(ManagedChannel orderServiceChannel) {
        return
withDeadline(OrderServiceGrpc.newBlocking(orderServic
eChannel));
    }

    // ---------- Utils ----------


    private <T extends AbstractBlocking<T>> T
withDeadline(T t) {
        return .withDeadlineAfter(defaultTimeoutMs,
java.util.concurrent.TimeUnit.MILLISECONDS);
    }
}

import com.example.gateway.api.dto.OrderDetailsDto;
import com.example.gateway.api.dto.OrderDto;
import com.example.gateway.api.dto.UserDto;
import com.example.order.grpc.GetOrderRequest;
import com.example.order.grpc.OrderResponse;
import com.example.order.grpc.OrderServiceGrpc;
import com.example.user.grpc.GetUserRequest;
import com.example.user.grpc.UserResponse;
import com.example.user.grpc.UserServiceGrpc;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

/**
 * REST-контролер API Gateway.
 * Зовнішній інтерфейс: HTTP/JSON.
 * Внутрішня взаємодія: gRPC.
 */
@RestController
@RequestMapping("/api/v1")
@RequiredArgsConstructor
public class GatewayController {

    private final UserServiceGrpc.UserServiceBlockingStub
userServiceStub;
    private final
OrderServiceGrpc.OrderServiceBlockingStub
orderServiceStub;

    // ------------------------ USERS ------------------------
```

```
    /**
     * Отримати користувача по id.
     * GET /api/v1/users/{id}
     * REST → gRPC (UserService.GetUser)
     */
    @GetMapping("/users/{id}")
    public ResponseEntity<UserDto>
getUser(@PathVariable long id) {
        GetUserRequest grpcRequest =
GetUserRequest.newBuilder()
                .setId(id)
                .build();

        UserResponse grpcResponse =
userServiceStub.getUser(grpcRequest);
        var user = grpcResponse.getUser();

        UserDto dto = new UserDto(
                user.getId(),
                user.getName(),
                user.getEmail()
        );
        return ResponseEntity.ok(dto);
    }

    // ------------------------ ORDERS ------------------------

    /**
     * Отримати замовлення по id.
     * GET /api/v1/orders/{id}
     * REST → gRPC (OrderService.GetOrder)
     */
    @GetMapping("/orders/{id}")
    public ResponseEntity<OrderDto>
getOrder(@PathVariable long id) {
        GetOrderRequest grpcRequest =
GetOrderRequest.newBuilder()
                .setId(id)
                .build();

        OrderResponse grpcResponse =
orderServiceStub.getOrder(grpcRequest);
        var order = grpcResponse.getOrder();

        OrderDto dto = new OrderDto(
                order.getId(),
                order.getUserId(),
                order.getStatus(),
                order.getTotalAmount()
        );
        return ResponseEntity.ok(dto);
    }

    // ------------------ AGGREGATED ENDPOINT ----------
--------

    /**
     * Агрегований endpoint:
     * GET /api/v1/orders/{id}/details
     *
     * 1) Через gRPC отримуємо дані замовлення
(OrderService).
     * 2) Через gRPC отримуємо користувача, який
зробив замовлення (UserService).
     * 3) Об'єднуємо в одну REST-відповідь
OrderDetailsDto.
     */
    @GetMapping("/orders/{id}/details")
    public ResponseEntity<OrderDetailsDto>
getOrderDetails(@PathVariable long id) {
        // 1. Отримуємо замовлення
        GetOrderRequest orderReq =
GetOrderRequest.newBuilder()
                .setId(id)
                .build();
        OrderResponse orderResp =
orderServiceStub.getOrder(orderReq);
        var order = orderResp.getOrder();

        // 2. Отримуємо дані користувача
        GetUserRequest userReq =
GetUserRequest.newBuilder()
                .setId(order.getUserId())
                .build();
        UserResponse userResp =
userServiceStub.getUser(userReq);
        var user = userResp.getUser();

        // 3. Формуємо агреговану DTO
        UserDto userDto = new UserDto(user.getId(),
user.getName(), user.getEmail());
        OrderDetailsDto dto = new OrderDetailsDto(
                order.getId(),
                order.getStatus(),
                order.getTotalAmount(),
                userDto
        );

        return ResponseEntity.ok(dto);
    }
}
```

ДОДАТОК В. ПРИКЛАД ЛОГІВ НА API-GATEWAY




```
{
  "timestamp": "2025-01-18T14:32:10.487Z",
  "level": "INFO",
  "gatewayInstance": "api-gateway-1",
  "traceId": "a3f9c1b2d4e56789",
  "spanId": "b7c21d98ef45aa01",
  "client": {
    "ip": "192.168.1.25",
    "userAgent": "WebApp/1.4.2 (React; Chrome/121)",
    "userId": 123
  },
  "request": {
    "method": "POST",
    "path": "/api/orders",
    "query": "",
    "protocol_external": "HTTP/1.1",
    "contentType": "application/json",
    "contentLength": 524,
    "correlationId": "req-9f3b7c6a-2e41-4b3d-8a90-67c4a1d53b11"
  },
  "security": {
    "authType": "JWT",
    "jwtValid": true,
    "jwtSubject": "user-123",
    "roles": ["USER"],
    "rateLimitPassed": true
  },
  "routing": {
    "targetService": "order-service",
    "targetMethod": "CreateOrder",
    "protocol_internal": "gRPC",
    "grpcService": "order.OrderService",
    "grpcMethod": "CreateOrder",
    "grpcHost": "order-service.default.svc.cluster.local",
    "grpcPort": 50051
  }
```