Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут
комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№ ___ від ____.____.2025 р.
Завідувач кафедри _____

_____ _____
(підпис) (прізвище та ініціали)

Пояснювальна записка

до дипломної роботи бакалавра

на тему «Реалізація підходу CQRS у розподіленій програмній системі з використанням брокеру повідомлень Kafka»

Захищено на засіданні ЕК № _____
протокол № ___ від ___.___.2025 р.
Оцінка _____ / _____
Голова ЕК

_____ _____
(підпис) (прізвище та ініціали)

Виконав:
студент 4 курсу, групи КС-41
Спеціальності:
122 Комп'ютерні науки
(шифр і назва спеціальності підготовки)
Іващенко Дмитро Володимирович
(прізвище, ім'я та по батькові, підпис)
Керівник ст. викл. Паршенцев Б. В.
(ступень, звання, прізвище та ініціали, підпис)
Рецензент к.т.н, доцент Лисицький К.Є.
(ступень, звання, прізвище та ініціали, підпис)

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра кібербезпеки інформаційних систем, мереж і технологій
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Галузь знань: 12 Інформаційні технології
Спеціальність: 122 Комп'ютерні науки
Освітня програма: Комп'ютерні науки

ЗАТВЕРДЖУЮ
завідувач кафедри ІПСіТ
Олег ОЛЕШКО
«____» _____20___ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЕКТ)

Іващенка Дмитра Володимировича
(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Реалізація підходу CQRS у розподіленій програмній системі з використанням брокеру повідомлень Kafka».

керівник роботи: ст. викл. каф. ІПСіТ Паршенцев Богдан Володимирович,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "16" квітня 2025року №4101-5/962

2. Строк подання студентом роботи 02 червня 2025 року

3. Перелік питань, які потрібно розробити:
дослідити сутність та базові принципи мікросервісної архітектури й CQRS-шаблону з інтеграцією Apache Kafka, визначити вимоги до міжпроцесної синхронної й асинхронної комунікації, проаналізувати предметну область розроблюваного застосунку, провести Event Storming, спроєктувати структуру команд і подій, реалізувати сховище подій з Event Sourcing, імплементувати сервіси для запису та читання даних на платформі Spring Boot, забезпечити автоматизоване й ручне тестування сервісів за підходами TDD та BDD, а також зробити висновки щодо доцільності реалізації CQRS у мікросервісній архітектурі.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Аналіз відкритих джерел, книжних видань, наукових праць та дисертацій.	18.04.2025-26.04.2025
2	Дослідження предметної області, що буде суттю мікросервісного застосунку.	27.04.2025-29.04.2025
3	Визначення особливостей реалізації мікросервісного застосунку у порівняння з монолітною архітектурою.	30.04.2025-03.05.2025
4	Проведення сесії Event Storming щодо проєктування архітектури спілкування у подальшому розроблюваному застосунку.	04.05.2025-05.05.2025
5	Програмна реалізація повідомлень: команд та подій, що використовуються для спілкування між мікросервісами.	06.05.2025-08.05.2025
6	Налаштування мікросервісної інфраструктури для забезпечення можливості подальшої розробки сервісів.	09.05.2025-15.05.2025
7	Програмна реалізація сервісів, що визначено під час Event Storming та аналізу предметної області, за шаблоном CQRS.	16.05.2025-27.05.2025
8	Написання автотизованих тестів, проведення ручного тестування застосунку.	28.05.2025-29.05.2025

5. Дата видачі завдання 10 жовтня 2025 року

Студент Іващенко Д.В. _____
ініціали, прізвище підпис

Керівник роботи Паршенцев Б.В. _____
ініціали, прізвище підпис

АНОТАЦІЯ

Пояснювальна записка містить 76 сторінок, 4 розділи, 21 рисунки, 1 таблицю, 19 джерел посилання.

Робота присвячена імплементації шаблону отримання даних Command and Query Responsibility Segregation (CQRS) у межах розробки застосунку з мікросервісної архітектури із застосуванням брокеру повідомлень Apache Kafka, що надає покращену масштабованість, доступність та спрощення підтримки застосунку. Проведено аналіз існуючих рішень, опрацьовано найкращі практики розробки мікросервісних застосунків з акцентом на коректну реалізацію CQRS з інтеграцією Apache Kafka для реалізації подієво-орієнтованої архітектури у мікросервісному застосунку. Результати роботи показали можливості реалізації сервісів у такий спосіб, що надають шляхи ефективного використання сучасних технологій, у тому числі нереляційних сховищ даних зі збереженням консистентності застосунку.

Методами дослідження є аналіз відкритих джерел: Інтернет-порталів, книжних видань та дисертацій; проведення процесу Event Storming задля створення архітектури застосунку; програмна реалізація мікросервісного застосунку та його окремих частин за допомогою платформи Spring Boot.

У результаті реалізації програмного коду сервісів отримано розподілену мікросервісну архітектуру користувацької платформи для роботи з сутностями кінофільмів. Такий застосунок включає в себе набір сервісів, що спілкуються між собою за допомогою у синхронний та асинхронний спосіб. Новизною такої реалізації є поєднання використання підходу CQRS та брокеру подій Apache Kafka у межах подієво-орієнтованої архітектури.

Ключові слова: АСИНХРОННА КОМУНІКАЦІЯ, МІКРОСЕРВІСИ, РОЗПОДІЛЕНА АРХІТЕКТУРА, СИНХРОННА КОМУНІКАЦІЯ, ШЕСТИКУТНА АРХІТЕКТУРА, APACHE KAFKA, CQRS, DDD, DOCKER, EVENT SOURCING, EVENT STORMING, HTTP, REDIS, REST, SPRING BOOT.

ABSTRACT

The explanatory note comprises 76 pages, 4 chapters, 21 figures, 1 table, and 19 references.

This work focuses on the implementation of the Command and Query Responsibility Segregation (CQRS) pattern within the development of a microservice-based application utilizing the Apache Kafka message broker. The chosen approach enhances scalability, availability, and simplifies maintenance. A comprehensive analysis of existing solutions was conducted, alongside a review of best practices in microservice development. Particular attention was given to the correct application of CQRS and its integration with Apache Kafka to enable an event-driven architecture. The study demonstrates how services can be implemented in a manner that leverages modern technologies—such as non-relational databases—while preserving overall system consistency.

The research methodology includes the analysis of open sources: online portals, book publications, and dissertations; conducting Event Storming sessions to design the application architecture; and the software development of the microservice application and its individual components using the Spring Boot platform.

As a result of the software implementation, a distributed microservice architecture was developed for a user platform centered around managing movie-related entities. The application consists of a suite of services that interact through both synchronous and asynchronous communication. The novelty of this solution lies in the integration of the CQRS pattern with the Apache Kafka event broker within an event-driven architectural framework.

Keywords: APACHE KAFKA, ASYNCHRONOUS COMMUNICATION, CQRS, DDD, DISTRIBUTED ARCHITECTURE, DOCKER, EVENT SOURCING, EVENT STORMING, HEXAGONAL ARCHITECTURE, HTTP, MICROSERVICES, REDIS, REST, SPRING BOOT, SYNCHRONOUS COMMUNICATION.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

ACID (Atomicity, Consistency, Isolation, Durability) – набір властивостей, що представляють транзакції у базах даних: атомарність, консистентність, ізоляція, довговічність.

API (Application Programming Interface) – інтерфейс взаємодії програмних компонентів.

BDD (Behavior-Driven Development) – методологія розробки з описом поведінки системи, що зрозуміла особам, що не є розробниками застосунку.

CQRS (Command and Query Responsibility Segregation) – шаблон розділення сервісу на окремі модель для читання та модель для запису даних.

CRUD (Create, Read, Update, Delete) – базові операції в базах даних.

DI (Dependency Injection) – інверсія використання залежностей.

DDD (Domain-Driven Design) – підхід до проєктування програмного забезпечення, орієнтований на предметну область.

HTTP (Hypertext Transfer Protocol) – текстовий формат передачі інформації між клієнтом та сервером.

MVP (Minimal Viable Product) – мінімальний життєздатний продукт; перша версія застосунку, що придатна до розгляду як повноцінний застосунок.

ORM (Object-Relational Mapping) – принцип зв'язку об'єктно-орієнтованого коду з реляційною базою даних.

RDBMS (Relational Database Management System) – система управління реляційними базами даних.

REST (Representational State Transfer) – набір рекомендацій передачі інформації між клієнтом та сервером.

SLO (Service Level Objective) – метрика очікуваної якості обслуговування.

TDD (Test-Driven Development) – методологія розробки, що представляє написання тестів перед розробкою сервісу.

ВСТУП

У сучасному світі інформаційних технологій зі збільшенням кількості клієнтів різних сервісів постають нові виклики щодо підтримки високого рівня продуктивності, масштабованості та раціонального використання програмних рішень, що дозволяють обмінюватися інформацією між застосунками, що розміщені у межах одного дата-центру для певного програмного продукту, так і за випадку, коли між різними сервісами стоять великі відстані, що можуть бути представлені тисячами кілометрів [1].

У випадку, коли всі компоненти спілкуються між собою через комп'ютерну мережу, існує проблема затримки передачі даних. Зі збільшенням розміру відповіді запиту також збільшується час відповіді. Ці два показники продуктивності передачі даних між мережею є критичними та надають пряму залежність загальній продуктивності спілкування компонентів однієї або декількох відповідних систем. Дана проблема неможливості надалі ефективно масштабувати програмні системи поступово з'являється у класичних монолітних застосунках, де вся бізнес-логіка працює в одному процесі. Вони з початку свого життєвого циклу за декілька років часто стають дуже складними для підтримки та втрачають можливість ефективного масштабування [1].

Наприклад, у випадку вертикального масштабування маємо проблему обмеження максимальної кількості користувачів, що можуть спілкуватися з певним застосунком за одиницю часу. Це стається найчастіше через те, що бази даних, що виступають у ролі сховища стану застосунку на рівні персистентності, мають обмежену кількість максимальної кількості з'єднань, що може робити застосунок з рівня бізнес-логіки [1]. Додатково, зазвичай у монолітних застосунках також використовуються реляційні бази даних, де хоча наявна підтримка виконання запитів різної складності, але саме ця особливість – виконання складних запитів, може також тримати відкритим підключення, тому що чим складніший запит, тим він може виконуватися

довше, навіть секунди або десятки секунд, що є критичним показником у сучасному світі, де клієнти бажають отримувати відповідь за невелику прийнятну кількість часу, наприклад, за десяток мілісекунд. Також повільні відповіді загалом знижують загальну продуктивність системи [2].

Щодо горизонтального масштабування у застосунках великого масштабу з великою кількість підкомпонентів, то даний варіант здається більш логічним та раціональним: можна клонувати монолітний застосунок у декілька екземплярів, якими управляє балансувальник навантаження. У такому випадку використовують шардінг, техніку розділення навантаження на екземпляри за певною ознакою, наприклад, за ідентифікатором користувача, часом запиту або за іншими інформаційними відомостями. У такому випадку для більшості проєктів мету ефективного масштабування застосунку може бути успішно досягнуто. Такий підхід є прийнятним у більшості випадків, він добре працює з часом. Однак у деяких проєктах клієнти надсилають запити нерівномірно по кількості на кожен маршрут. Наприклад, може бути ситуація, що отримання інформації про профілі користувачів не має широкого попиту запитувачів, а робота із замовленнями, зі створенням та їх отриманням, набуває надзвичайної популярності. У такому випадку, у класичному проєкті для кожного компоненту піднімають відповідну інфраструктуру, наприклад, кеші в ORM або інших інструментах для тимчасового збереження стану застосунку. Звідси маємо проблему, що монолітний застосунок завжди потребує роботи всієї інфрастуктури застосунку: навіть якщо до одного сервісу не роблять запити, задля створення екземпляру для обробки навантаження з іншим сервісом, все одно потрібно запускати весь застосунок одразу і для підтримки роботи і з першим сервісом, і з другим що може коштувати додаткових ресурів на хостингу через те, що відповідні програмні компоненти для такої інфрастуктури коштують відповідно до потребуваної продуктивності системи. Саме тут така система зустрічає межі ефективного масштабування, що надалі робить невигідною роботу системи [1].

У найгіршому випадку така система може почати отримувати збитки, що перевищуватимуть доходи від роботи такого застосунку.

Актуальність даної роботи полягає у впровадженні сучасного підходу проєктування застосунку так, щоби швидкість передачі даних від клієнту до серверу була найбільш можливою у межах прикладного рівня структури застосунків, а також була наявність можливості точкового масштабування саме тих компонентів, які потрібні у певний момент часу. Ключовим архітектурним шаблоном проєктування такої системи є мікросервісна архітектура, де бізнес-логіка застосунку розміщена на декількох незалежних компонентах, сервісах, що представляють собою незалежно впроваджувані, низько зв'язні компоненти [1]. Використання такого підходу надає можливість подальшого включення використання шаблону CQRS. Даний шаблон проєктування рівня застосунку надає можливості розділення моделі роботи із застосунком на дві частини. Перша частина – модель для читання. Вона здійснює агрегування всіх потрібних даних, що може потребувати клієнт. Така модель також використовує оптимізації, які спеціально налаштовані для швидкого та ефективного читання інформації, яка завжди готова до отримання у будь-який момент часу за запитом клієнта. Друга частина – модель для запису. У ній здійснюється лише запис інформації, де клієнт напряму не має можливості робити запити на вибірку даних. Тут можливе використання підходів, що забезпечують зберігання інформації у вигляді, що оптимізований саме для обробки даних у сервісі так, що надає можливість отримувати клієнтом потрібну інформацію швидше [2].

Існуючими рішеннями реалізації CQRS та мікросервісної архітектури є сучасні системи великого масштабу, що підійшли до лімітів ефективності роботи монолітних застосунків. Одним з прикладів є Netflix. У даній компанії мікросервіси використовуються повсюди. Наприклад, під час отримання клієнтом користувача ліцензії на програвання або завантаження певного відеоматеріалу, використовується даний шаблон, де на сервісі з інформацією про доступні активні ліцензії, користувач може запитати статус такої ліцензії

з тієї моделі сервісу, що оптимізована саме для швидкого читання [3]. Звідси, користувач швидше отримає відповідь та почне відповідне програвання відео замість довгого очікування на виконання складного запиту, що виконувався би довше на сервісі загального призначення без додаткових оптимізацій.

Об’єктом дослідження є мікросервісний застосунок у вигляді незалежного Application Programming Interface (API) із застосуванням шаблону CQRS за мотивами сервісу Netflix, де центральною моделлю є сутності кінострічок.

Предметом дослідження є імплементація сучасних найкращих практик проєктування рішення у на основі класичної мікросервісної архітектури застосунку із застосуванням шаблонів отримання даних CQRS та допоміжного шаблону Event Sourcing із застосуванням брокеру повідомлень Apache Kafka.

Метою реалізації такої системи є поліпшення навичок проєктування та створення архітектури типового мікросервісного застосунку, що створено за сучасними трендами та найкращими практиками із акцентом на реалізацію шаблону CQRS з іншими допоміжними шаблонами, що надають можливість ефективно імплементувати дане рішення через включення властивостей високої доступності застосунку, його незалежного розгортання, а також мінімізація часу відповіді від серверу до клієнту.

Новизною даної роботи є практична реалізація мікросервісної архітектури з акцентом на імплементацію шаблону CQRS у поєднанні із шаблонами консистентності Event Sourcing, що покращує можливості масштабування сервісів у такій архітектурі через можливість споживання всіх попередніх подій, що надає безпроблемне рішення створення реплік та незалежного деплою сервісів з малим рівнем зчеплення між ними [4].

Результати даної роботи будуть корисні розробникам програмного забезпечення, що хочуть дізнатися важливі деталі та особливості реалізації шаблонів отримання даних та відмовостійкості у межах розгляду архітектурного шаблону мікросервісної архітектури.

1 АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1 Основні положення

1.1.1 Визначення мікросервісної архітектури

Microservices (мікросервіси) – це архітектурний стиль розробки програмного забезпечення, де застосунок декомпозується на декілька низько зв’язних між собою сервісів. Сервіс – це окремий застосунок, що є складовою мікросервісної архітектури; такий сервіс має вузьку спеціалізацію та невелику кількість бізнес-логіки відносно загальної кількості логіки застосунку [4]. Окремим сервісом не обов’язково мати один принцип відповідальності. Розділення відповідальності між окремим частинами застосунку можна виконувати вільно відповідно до бізнес-інваріантів. У межах мікросервісної архітектури існують наступні рекомендації, що у сучасному їх представленні є конвенціями до розробки:

- кожному сервісу потрібно надати окремий сховище для даних. Важливо зазначити, що хоча дане твердження зазвичай має на увазі саме окремий екземпляр застосунку бази даних, не обов’язково виділяти сховище саме у такий спосіб. Достатньо створити у вже існуючій базі даних окремий простір. У такий спосіб маємо простоту подальшої масштабованості системи, якщо оригінальна версія застосунку представлена у вигляді класичного монолітного рішення, а поступова міграція до мікросервісів не може бути виконання повністю одразу. У випадку, якщо застосунок створюється одразу у вигляді розподіленої архітектури, краще використовувати саме окремі екземпляри баз даних;
- сховища даних кожного сервісу мають бути приватними. Жоден інший сервіс потрібен не мати можливості напряму отримувати інформацію з певного сервісу через сховище. Це зумовлено тим, що типове сховище, наприклад, база даних, може мати застарілі дані, які ще не оновилися сервісом. Наприклад, часто інструменти Object-

Relational Mapping (ORM) відкладають збереження у сховище даних на певний час через оптимізацію використання мережевих ресурсів або у інший спосіб тощо;

- на кожному сервісі має бути присутнє API, яке може бути представлене або у вигляді можливості робити запити напряму на сервіс, або опосередковано через публікацію повідомлень: команд та подій. Тут важливо зазначити, що у загальному випадку задля виконання цієї умови можна спростити дану вимогу до наступного твердження: сервіс має брати участь у міжсервісній взаємодії або бути доступним для клієнта напряму [4].

У теорії системного дизайну існує поняття Scale Cube (куб масштабованості). Він представляє собою трьохосьове представлення можливості масштабування застосунку. Нехай маємо приклад такого кубу на рисунку 1.1.1.1.

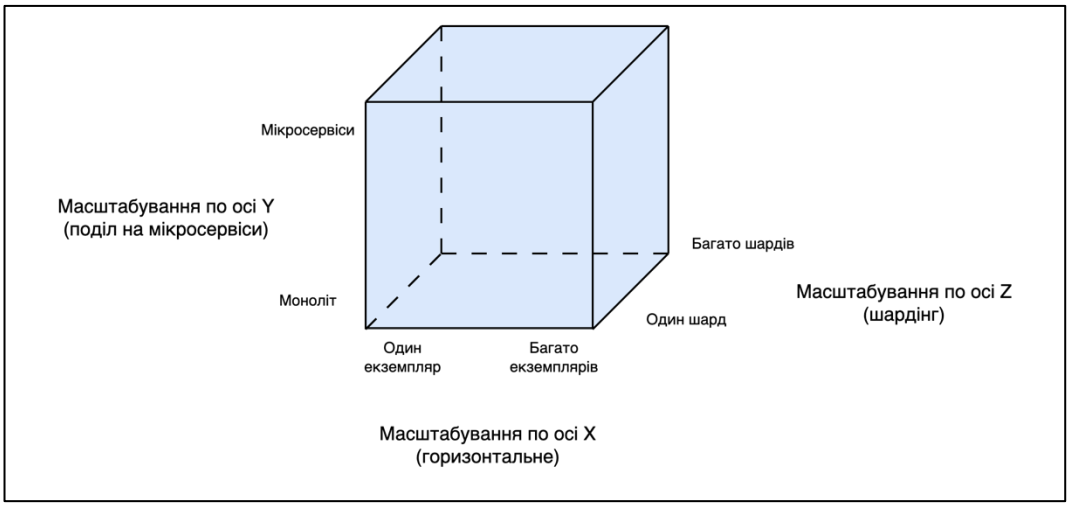

Рисунок 1.1.1.1 – Зовнішній вигляд кубу масштабованості

З даного рисунку маємо, що у сучасному технологічному розвитку побудови застосунків маємо три види масштабованості: за кількістю екземплярів, за поділом на мікросервіси та за шардінгом, процесом розділення відповідальності даних між різними екземплярами або сервісами за певною ознакою даних, з якими відбувається робота (за ідентифікатором користувача,

номером замовлення тощо) [5]. У межах мікросервісної архітектури масштабування може бути успішно виконане за всіма трьома осями. Важливо зазначити, що доцільність обов'язкового масштабування в усіх трьох напрямках не є гарантованою завжди, тому що все залежить від структури застосунку та цілей, що ставляться у межах його розробки та підтримки. За маленького проєкту навряд чи є потреба здійснювати масштабування за шардінгом, якщо ми маємо на меті створення Minimum Viable Product (MVP). Додатково, під час створення сервісів саме розробником не завжди наявна потреба створювати систему та одразу тестувати її у випадку роботи за горизонтального масштабування. Такий варіант раціонально використовувати у лише у великих проєктах, де потреби виходять на такий рівень, де кількість користувачів може сильно варіюватися з часом та потрібно запускати додаткові екземпляри задля продовження коректної роботи. Поточне дослідження створення застосунку за допомогою мікросервісів має на увазі створення застосунку із використанням масштабування по осі Y. Ігнорування інших двох осей є доречним у цьому випадку, тому що шардінг та горизонтальне масштабування – це рівень інфраструктури системи, що наразі не потрібен для демонстрації можливостей архітектурного підходу та шаблонів проєктування. У даній роботі пропонуватиметься розглянути рівень застосунку та інфраструктури застосунку відповідно як допоміжний рівень для вирішення проблем, що будуть описані далі.

1.1.2 Структура сервісу

Класичний сервіс – це окремий застосунок, як було зазначено у підрозділі 1.1.1, обов'язково потрібен мати певне API для забезпечення взаємодії з іншими компонентами системи. Нехай ми маємо застосунок, що представляє агрегат кінофільмів та відповідає за операції Create, Read, Update, Delete (CRUD) над даною сутністю. Візуалізацію прикладу абстрактної структури сервісу маємо на рисунку 1.1.2.1.

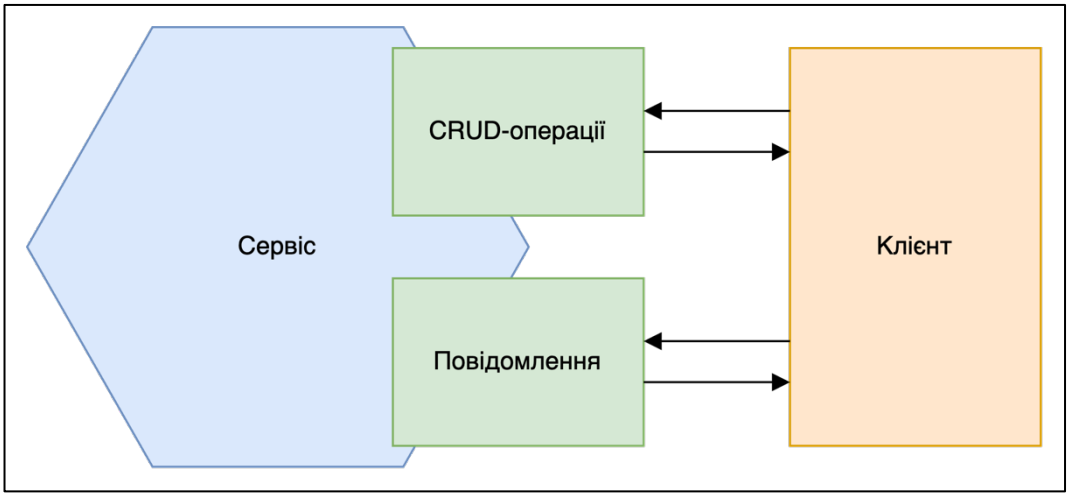

Рисунок 2.1.2.1 – Абстрактне представлення
спілкування клієнту та серверу

З рисунку 1.1.2.1 отримуємо, що сервіси можуть спілкуватися одне з одним, з іншими клієнтами через певний набір контрактів API. У такому випадку клієнт визначає якою частиною контракту йому користуватися за відповідних потреб. Саме сервіс у своїй будові відповідає за певний аспект бізнес-логіки. У даному прикладі наведено приклад з CRUD-сервісом фільмів. Важливо відмітити, що у даному випадку ми маємо відображення «один до одного» у сенсі відповідності мікросервісу та його відповідника визначенню «сервіс» у трьохшаровій архітектурі застосунку, де бізнес-логіка представлення на рівні сервісу, має у собі всю потрібну логіку для відповідності бізнес-логіці та інваріантам [6].

У випадку мікросервісної архітектури, типовий сервіс рекомендовано будувати за гексагональною (шестикутною) архітектурою через спрощення розуміння як працює сервіс. У такому випадку, коли у системі буде наявно багато сервісів, у кожному з них можна буде простіше розібратися з їх способами спілкування між собою та з клієнтами. Гексагональна архітектура – це архітектурний шаблон на рівні застосунку, що має на увазі представлення програмного рішення із використанням у собі низькозв’язних компонентів у вигляді портів та адаптерів до бізнес-логіки, що представлена у центрі. Порт – це певний інтерфейс задля спілкування з певною бізнес-логікою, що

представляє собою API-контракт, який має реалізувати адаптер. У свою чергу, адаптер – це реалізація програмного компоненту, що спілкується із зовнішнім світом, поза сервісом, та здійснює спілкування із бізнес-логікою через визначений порт. У класичному випадку адаптери підключаються до бізнес-логіки через механізм Dependency Injection (DI), що доступний у більшості сучасних веб-фреймворків. У загальній ідеї такої архітектури лежать чотири види портів, що поділяються на два загальні типи (вхідні та вихідні) [7]. Приклад такої класифікації портів наведено на таблиці 1.1.2.1.

Таблиця 1.1.2.1 – Класифікація портів у гексагональній архітектурі

Вид	**Тип**	**Опис**
Вхідний	Вхідні події	Представляють асинхронні події, що стосуються зміни станів бізнес-сутностей.
	Спілкування з клієнтом	Описують контракти для синхронного спілкування з клієнтом.
Вихідний	Вихідні події	Містять події будь-якого характеру, що застосунок виконує у результаті виконання бізнес-логіки
	Збереження даних	Характеризують класичне з’єднання зі сховищем стану застосунку.

Власне реалізація такої архітектури не вносить нових програмних компонентів, а лише здійснює новий підхід до розуміння компонування вже існуючих компонентів або тих, що планується написати у майбутньому. Наприклад, у якості порту для збереження даних можна взяти програмний інтерфейс репозиторію сутності, а у свою чергу такий інтерфейс буде використовувати адаптер у вигляді Data-Access Object (DAO), де вже є імплементація конкретна імплементація логіки збереження у потрібне місце (у базу даних тощо). У стандартному варіанті реалізації гексагональної архітектури допускається просте групування програмних класів та інтерфейсів за пакетами, що мають назви типів портів, що наведено у таблиці 1.1.2.1. При розробці застосунку у такий спосіб отримуємо прямолінійну можливість

проаналізувати можливості застосунку та його способи взаємодії із іншими об’єктами системи або із зовнішніми клієнтами [7]. Приклад представлення структури сервісу, що реалізує гексагональну архітектуру, наведено на рисунку 1.1.2.2.

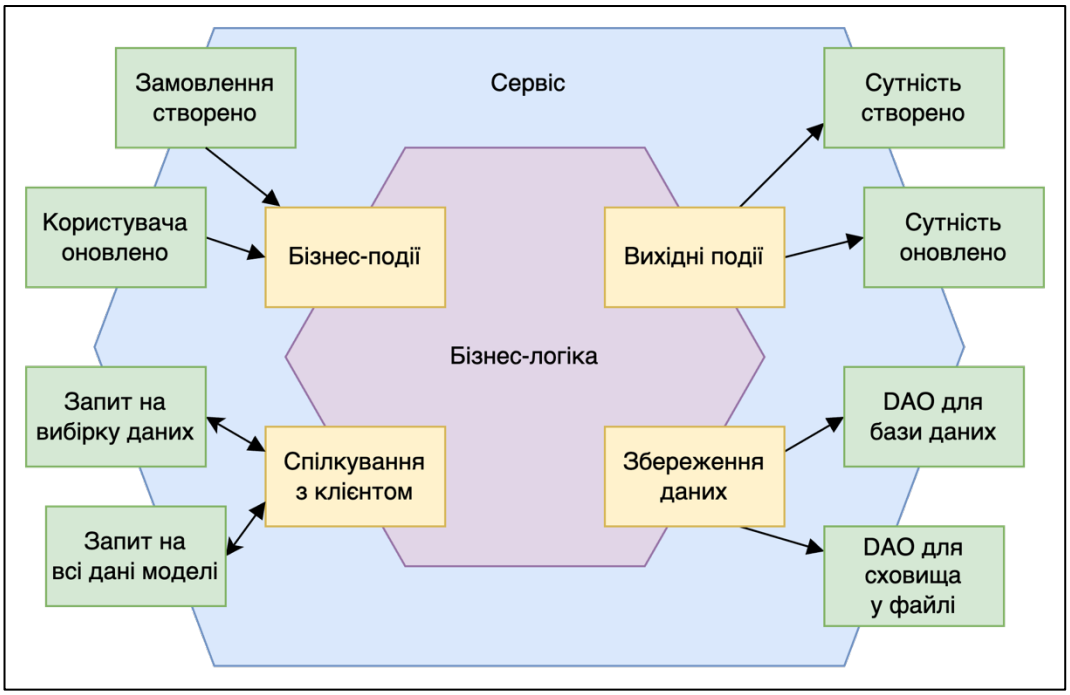

Рисунок 1.1.2.2 –Візуалізація гексагональної архітектури сервісу

Маємо наступні переваги гексагональної архітектури. По-перше, отримуємо спрощення розуміння залежностей компонентів між собою у наданому сервісі, підтримка чистоти коду задля можливості нескладного масштабування сервісу у майбутньому, а також можливість тестування бізнес-логіки в ізольованому варіанті. По-друге, у мікросервісній архітектурі застосунку, де зазвичай у наявності велика кількість сервісів, які нагадують такий, що показано на рисунку 1.1.2.2. Через це, якщо будувати застосунки за гексагональною архітектурою, то надається спрощення розуміння як та у який спосіб певний сервіс має залежності з іншим сервісом щодо аспекту спілкування одне з одним.

З іншого боку, у такій архітектурі є недоліки. Наприклад, цей підхід заставляє розуміти розробку сервісів трохи інакше, що може сповільнити

розробку. Також тут можливе дублювання коду у межах створення додаткових абстракцій [7]. Хоча дана архітектура не є обов’язковою у жодному з варіантів розробки застосунків, однак переваги, що отримуються, є набагато суттєвішими саме у межах мікросервісів, тому у даній роботі більшість сервісів буде розроблено саме за такою архітектурою.

2.2 Міжпроцесна комунікація

2.2.1 Синхронне спілкування

У класичному монолітному застосунку зазвичай існує один центральний програмний процес, до якого надходять всі запити від клієнтів. Тут клієнт надсилає запит, який проходить мережевий шлях від себе до серверу, далі сервер обробляє відповідь, надсилає відповідь назад та клієнт наприкінці її отримує. Це класичний приклад блокуючої спілкування, де одна сторона спілкування очікує на відповідь іншої [8]. Приклад діаграми послідовності виклику віддаленої процедури за допомогою синхронного спілкування між двома сервісами наведено на рисунку 1.2.1.1.

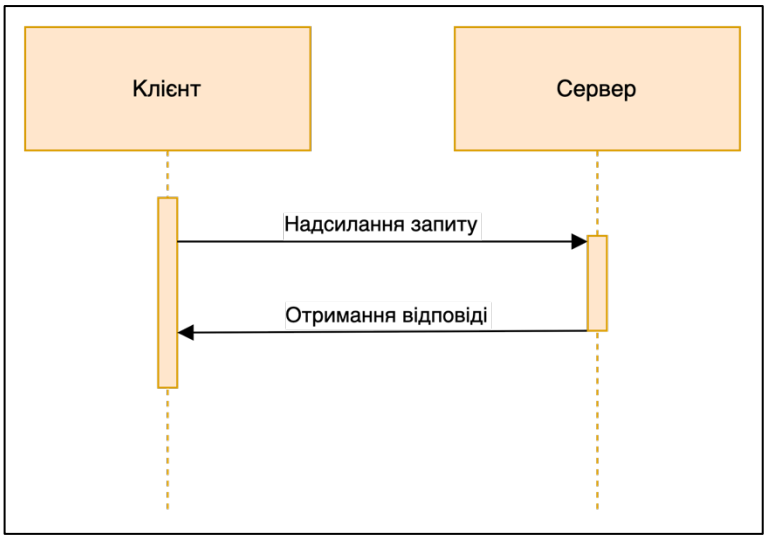

Рисунок 1.2.1.1 – Діаграма послідовності синхронного спілкування

Розглянемо варіант синхронного спілкування у мікросервісній архітектурі. Тут ми маємо два варіанти для отримання певних даних, віддаленого виклику процедури тощо:

- комунікація із зовнішнім клієнтом;
- діалог з іншими сервісами у межах одного застосунку.

У першому варіанті нема нічого критичного у межах того, що клієнт хоче запросити певну інформацію синхронно. Така поведінка бажана у більшості варіантів, особливо за звичайних CRUD-операцій, коли інший зовнішній клієнт бажає викликати маршрути нашого застосунку так, що не потрібно впроваджувати додаткову інфраструктуру для очікування відповіді. Тому даний варіант у межах даної роботи підійде для створення маршрутів для зовнішніх клієнтів.

Щодо другого варіанту, у мікросервісній архітектурі, якщо кожен сервіс буде використовувати такий варіант такий варіант призведе до високої зв’язності між сервісами та потребуватиме мати всі сервіси, що виконують певну трансзакцію, перебутаи у режимі онлайн. Так можна отримати розподілений моноліт (distributed monolith), що матиме всі недоліки як мікросервісної архітектури, так і монолітної: складне масштабування сервісів, важке розгортання тощо. Звідси, синхронна комунікація у міжсервісній взаємодії має бути за можливістю мінімальною [8].

1.2.2 Асинхронна комунікація

Асинхронна комунікація – це вид міжпроцесної взаємодії, за якого клієнт використовує неблокуюче очікування відповіді від серверу. Під час отримання відповіді клієнт зазвичай запускає окрему функцію зворотнього виклику [8].

Зазвичай у випадку асинхронної комунікації використовують обмін повідомленнями, що до виконання та результат виконання відповідно.

Такі повідомлення поділяються на наступні види:

- команда (command) – представляє запит, його параметри;
- подія (event) – демонструє результат виконання задачі, зміну стану доменної моделі тощо.

Нехай надсилання події відбувається за допомогою виклику віддаленої процедури на клієнті [8]. Тоді приклад асинхронної взаємодії виду «запит-відповідь», що є аналогією синхронного спілкування, у вигляді діаграми послідовностей представлено на рисунку 1.2.2.1.

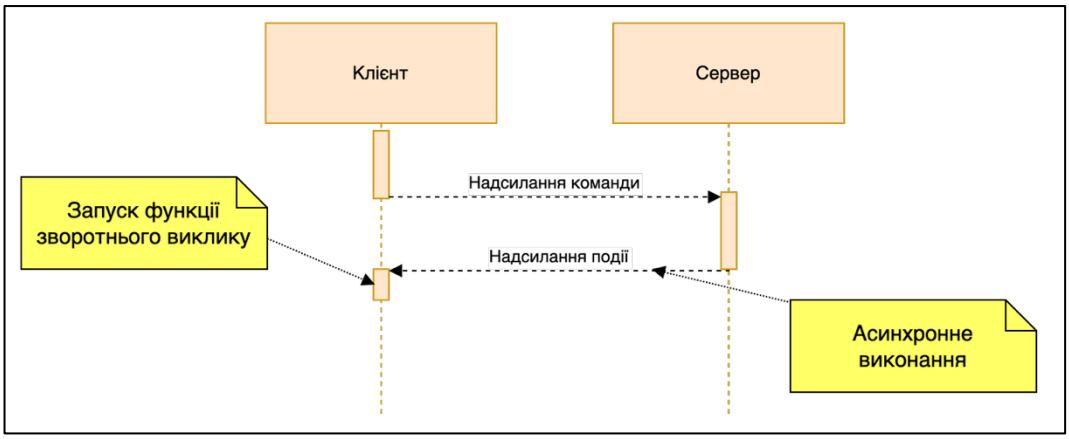

Рисунок 1.2.2.1 – Приклад асинхронного спілкування «запит-відповідь»

Такий представлений варіант виглядає надійніше з боку відсутності потреби клієнта здійснювати блокування власного потоку виконання. Однак варіант «запит-відповідь» має значний недолік: у такому варіанті, за відсутності додаткової інфраструктури, виникає проблема, коли клієнт недоступний. Сервер захоче відправити повідомлення назад до клієнта, але це буде неможливо.

У цьому варіанті можна припустити створення кешу адрес клієнтів та пробувати надсилати відповідь з певним проміжком часу зі сподіванням, що клієнт перейде у режим онлайн та буде доступним. Але так збільшується зв’язність інфраструктурної логіки із логікою застосунку. Додатково, у мікросервісній архітектурі зазвичай більшість сервісів мають динамічні адреси, також нові екземпляри сервісів постійно запускаються та зникають в

залежності від потреб застосунку. Приклад такої архітектури наведено на рисунку 1.2.2.2, пункті а.

В іншому випадку маємо варіант з публікацією повідомлень через брокер повідомлень з комунікацією «один до багатьох». Даний варіант представляє собою створення повідомлень та розміщення їх у брокері повідомлень так, що у будь-який момент інший сервіс може забрати дане повідомлення. Через це немає потреби всі сервіси, що залучені до певної транзакції, мати у стані онлайн одночасно. У загальному випадку, маємо наступні переваги спілкування через брокери повідомлення:

- низька зв’язність між сервісами, що підвищує продуктивність розробки через спрощення архітектури комунікацій;
- можливість кешування попередніх повідомлень для подальшого перечитування.

Однак також маємо і недоліки:

- потреба у додатковій інфраструктурі, брокері повідомлень;
- єдина точка відмови. Виправляється реплікацією брокерів.

Приклад такої низькозв’язної архітектури наведено на рисунку 1.2.2.2, пункті б.

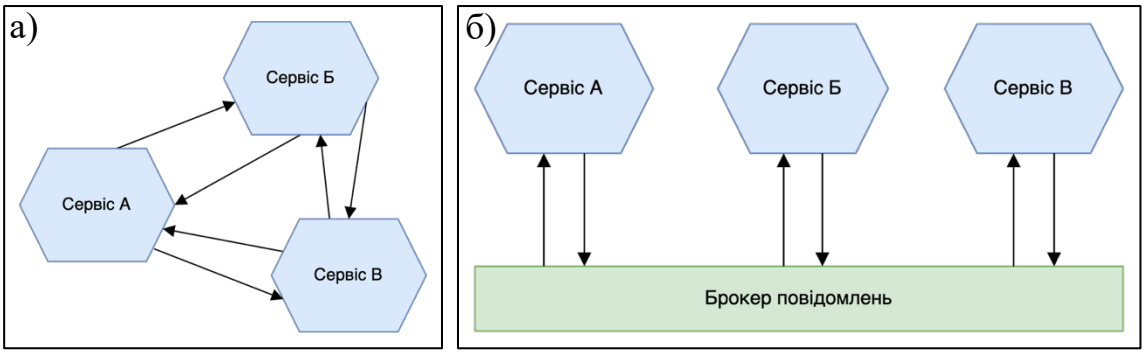

Рисунок 1.2.2.2 – Наочна різниця зв’язності сервісів одне з одним без використання брокеру повідомлень та з його використанням

Брокери повідомлень поділяються на два типи: такі, що засновані та чергах та такі, що базуються на журналюванні [8].

Засновані на чергах брокери містять у собі повідомлення, що читаються клієнтами один раз, а потім видаляються з черги. Такий варіант підходить для архітектури, де у пріоритеті наявна потреба передавати повідомлення до конкретного одержувача. Найпоширенішими застосуваннями черг є сервіси для комплексної обробки наданої інформації, наприклад, обробка відео або математичні обчислення тощо, де брокер обирає до якого клієнта надати завдання так, щоби обчислення були швидшими та ефективнішими, наприклад, в залежності від потужності кожного з екземплярів сервісів. Хоча такий варіант є надійним та прямолінійним, але він більш складний для налаштування. Додатково, у межах даної роботи є акцент на створенні однакових за «рангом» сервісів, що не відрізняються одне від одного. Додатково, саме масштабування сервісів на інфраструктурному рівні, щоби розкрити питання ефективності черг, також не є ключовою темою даної роботи [8].

У свою чергу, журнальні брокери містять такі записи, що можна читати багато разів кожному з клієнтів такого брокеру за потреби у будь-який момент часу. Такі брокери також називають «append-only logs». У такому випадку один сервіс публікує повідомлення без вказання до кого він має потрапити, а всі зацікавлені інші сервіси можуть отримати таке повідомлення. Додатково, тут підтримуються читання того самого потоку новими сервісами з початку або починаючи із зазначеного запису, тому тут присутня можливість динамічно додавати сервіси у процесі роботи. Такий варіант добре підходить для застосунку, що планується розробити у даній роботі, тому що тут відсутня потреба налаштовувати які сервіси мають отримати повідомлення: кожен вирішує чи потрібно слухати певний канал з повідомленнями окремо [8].

1.2.3 Apache Kafka

Одним з лідерів у сфері журнальних брокерів є Apache Kafka. Даний інструмент має ідею тем (топіків), створювачів (продюсерів) та споживачів (консюмерів). Топік – це категорія, до якої організуються повідомлення. Це може бути певна бізнес-сутність, наприклад, замовлення або користувач. Консюмери та продюсери – це ті, хто публікують повідомлення та ті, хто їх споживають відповідно. Додатково, кожен топік може буде поділений на розділи, що нагадують шардінг. У такому випадку певний набір консюмерів збираються в одну групу для читання з певного топіку за розділами. Kafka використовує ідею pull-моделі, де споживачі власноруч перевіряють наявність нових повідомлень. У такий спосіб можна знизити навантаження на мережу, тому що якщо певні споживачі офлайн, вони ніяк не впливають на продуктивність мережі [9].

Як було зазначено вище, у журнальних брокерах повідомлень, яким є Kafka, головними особливостями є низька затримка отримання та надсилання повідомлень через оптимізовані мережеві комунікації, а також можливість перечитання попередніх повідомлень за потреби у будь-який час. Через це Kafka гарантує, що у межах одного розділу всі повідомлення будуть збережені та надалі прочитані у коректному порядку. Додатково, гарантується читання повідомлень за моделлю «як мінімум один раз» («at least once») [9], що добре підходить для мікросервісів, тому що пропуск певного повідомлення може бути критичним та призведе до неконсистентності застосунку.

Більш детальний приклад комунікації сервісів з брокером повідомлень наведено на рисунку 1.2.3.1. У випадку мікросервісної архітектури такий підхід є ефективним до використання через вищезазначені переваги брокерів, особливо таких, що є журнального типу, як Kafka. У межах даної роботи також буде зосереджено увагу на нерозділенні споживачів на групи через те, що матимемо акцент на масштабування саме за віссю Y, відповідно до кубу масштабованості у пункті 1.1.1.

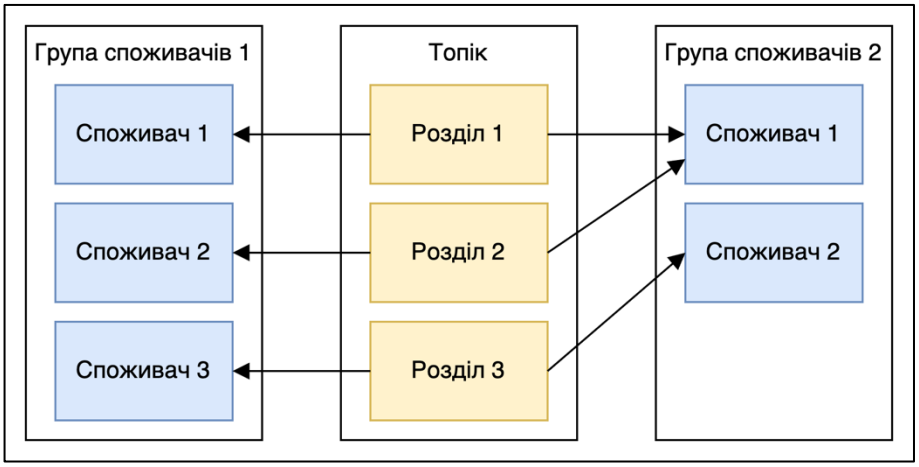

Рисунок 1.2.3.1 – Споживання повідомлень групами споживачів з топіку, що поділений на розділи

1.3 Шаблон Command and Query Responsibility Segregation

1.3.1 Постанова проблеми

Під час побудови мікросервісної архітектури наявні наступні проблеми щодо структури сервісів щодо варіанту виконання запитів до даних сервісів.

По-перше, так як у мікросервісній архітектурі зазвичай кожен сервіс має власне приватне сховище інформації, тут також є задача отримання даних поміж сервісів. Якщо один сервіс має інформацію щодо однієї сутності, то інший потрібен мати можливість отримати інформацію щодо неї задля різних цілей. Зазвичай, головна мета отримання інформації про сутності між різними сервісами – це виконання бізнес-інваріантів. Одним з рішень є отримання інформації з іншого сервісу напряму, наприклад, через синхронне спілкування [2].

Однак у такому варіанті, якщо в Service Level Objective (SLO) проєкту маємо гарантію режиму роботи 99.5% часу на рік, то якщо певний сервіс також потребуватиме запиту з інших сервісів для надання представлення клієнту,

то маємо наступну формулу доступності такого сервісу з отриманням всієї потрібної інформації на певний момент часу:

$$A = 99.5\%^{S}, \quad (1.3.1.1)$$

де S – це кількість сервісів, до яких потрібно звернутися задля отримання всієї потрібної інформації;

A – це доступність такого сервісу, що здійснює запити до інших сервісів.

Звідси, якщо $S = 5$, то $A = 99.5\%^{5} \approx 97.5\%$, що є критичним показником у багатьох сервісах. Більш того, у такий спосіб неможливо буде виконати SLO [8].

Саме така проблема також не є притаманною лише синхронному спілкуванню поміж сервісами, бо у разі асинхронного спілкування за будь-якого варіанту потрібний сервіс має чекати на отримання всієї інформації. Дану проблему з вибіркою даних з інших сервісів можна вирішити за допомогою CQRS, хоча даний шаблон не є головним призначенням для цього.

По-друге, кінцевим клієнтам потрібно швидко та зручно робити запити до застосунку задля отримання певних даних. Нехай ми маємо класичний сервіс, який наведено у якості прикладу на рисунку 1.1.2.1. Звідси, клієнт робить синхронний запит до такого сервісу та очікує відповідь. Але більшість мікросервісів використовують рішення сховища даних у вигляді реляційних систем управління базами даних (Relational Database Management System, RDBMS), у яких присутня нормалізація даних по різним таблицям. Тому для вибірки даних сервісу потрібно звертатися до бази даних, де виконується об’єднання певних таблиць у такий вигляд, що задовольняє API-контракт. Однак така операція потребує певного часу. Припустимо, що можна обрати певне документоорієнтоване рішення Not Only SQL (NoSQL), де дані можуть бути денормалізовані та оптимізовані для читання, тому дана проблема може

вирішитися. Однак у такому випадку, як відомо, NoSQL-рішення зазвичай не підтримують транзакційність, тобто з властивостей баз даних Atomicity, Consistency, Isolation, Durability (ACID) зазвичай не мають властивість ізоляції. Така особливість є критичною для високонавантажених застосунків, тому що без транзакційності застосунок може набути неконсистентного стану [8].

По-третє, іноді постає задача повнотекстового пошуку по певним даним з бази даних. Деякі реляційні рішення, наприклад, PostgreSQL, мають підтримку повнотекстового пошуку [10]. Однак такі можливості є базовими та дещо обмеженими щодо NoSQL-рішень, які мають більш потужні варіанти аналізу тексту та пошуку по ньому, наприклад, Elasticsearch [11]. Тому якщо додавати іншу базу даних в застосунок, є нова проблема синхронізації результатів. Такий варіант потрібно окремо підтримувати, налаштовувати в застосунку тощо, що може бути складним за часом виконання та за підтримкою.

По-четверте, з другого пункту також виходить інша проблема: коли сервіс стає більшим за масштабом можливостей, маємо проблему підтримки такого сервісу, тому що в ньому можна просто складно розібратися, особливо, якщо над ним працює велика кількість розробників.

1.3.2 Опис шаблону

CQRS має два основні види реалізації:

- розділення у межах одного сервісу;
- окремий сервіс з публічним читанням записів із синхронним API.

У випадку CQRS на рівні одного сервісу маємо впровадження додаткової бази даних, що наповнюється через події та доступна клієнтам для виконання запитів. У свою чергу, оригінальна частина сервісу матиме лише модифікуючі можливості Create, Update, Delete (CUD). Додатково, за власної моделі для читання у тому самому сервісі, маємо наступну ідею: нехай сервіси А та Б мають певні доменні повідомлення щодо зміни стану власних бізнес-

сутностей або інші сервісні повідомлення, та публікують їх як події до брокеру повідомлень. Звідси, задля вирішення такої проблеми інший сервіс В підписується на потрібні йому події з сервісів А та Б. У такий спосіб цей сервіс створює та накопичує певний набір інформації з інших сервісів, що доступний вже локально та незалежно від інших сервісів. Так, навіть якщо через деякий час сервіси А та/або Б перестануть коректно функціонувати, у сервісу В все одно буде потрібна інформація у наявності. Це один із способів вирішити проблему з SLO, тому що зв'язність між сервісами тепер відсутня [8].

З одного боку, такий варіант виглядає доречним, однак у цьому випадку проблема складності підтримки сервісу залишається актуальною. Тепер без додавання нових бізнес-функцій підтримка такого сервісу стане складнішою, що може бути критичним за великої кількості сервісів у застосунку. Приклад CQRS у межах одного сервісу, де для збереження операцій для виконання запитів над даними використовується оптимізоване для читання NoSQL-рішення, наведено на рисунку 1.3.2.1.

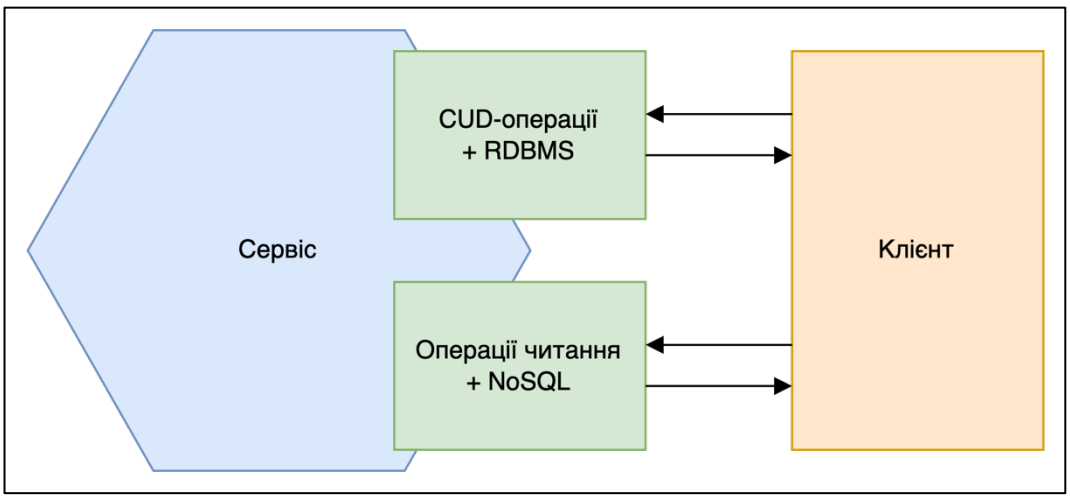

Рисунок 1.3.2.1 – Приклад схеми роботи сервісу за CQRS

1.3.3 Варіанти реалізації

Відповідно, нехай на прикладі сервісу з фільмом маємо, що CUD-операції представлені CUD-моделлю у застосунку, а операції читання – R-моделлю від слова «read» [8]. Також нехай сервіс для фільмів має

інформаційну залежність з акторами та режисерами, тому що це зазвичай потрібно для отримання повної інформації про фільм. Тоді діаграму послідовностей для оновлення моделі для читання наведено на рисунку 1.3.3.1.

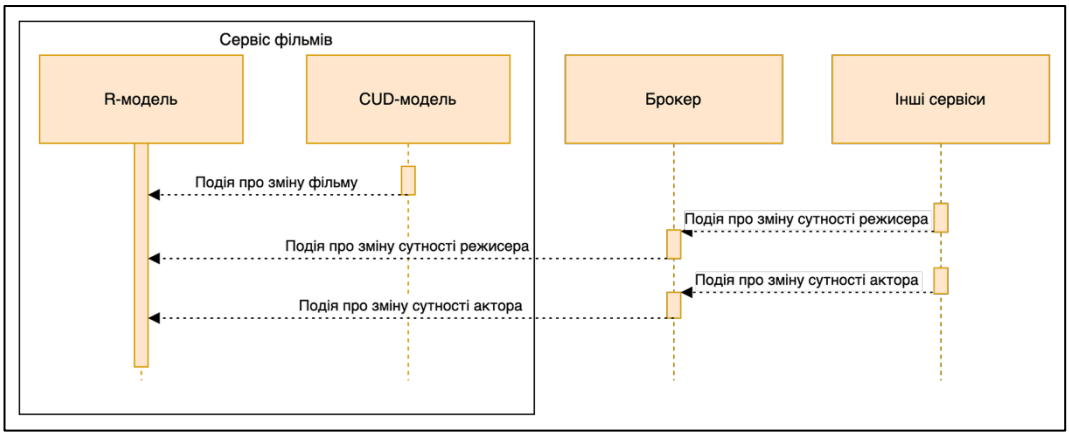

Рисунок 1.3.3.1 – Приклад оновлення моделі для читання у межах того самого одного сервісу

Інший варіант реалізації CQRS – створення окремого сервісу, що підписується на доменні події. У такий спосіб можливе створення незалежного сервісу, який лише надає вигляд інформації через читання R-моделі [8]. Приклад такого варіанту наведено на рисунку 1.3.3.2.

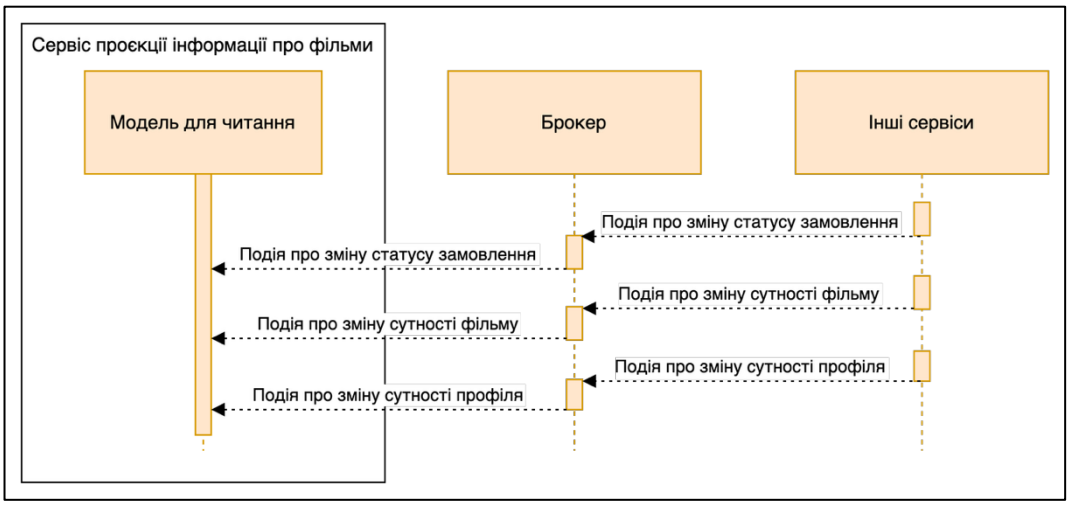

Рисунок 1.3.3.2 – Приклад оновлення моделі для читання у межах окремого сервісу

Звідси, постає питання який варіант використовувати під час розробки сервісів. Тут існує одна головна різниця між такими підходами – можливість масштабування системи. По-перше, різні частини сервісів (частина для читання та для запису інформації відповідно) можуть мати власні вимоги до потреб серверів за оперативною пам’яттю (Random Access Memory, RAM), за потужністю центрального процесора (Central Processing Unit, CPU) та пам’ять постійного зберігання [8].

Якщо модель для читання має значущі потреби до RAM у вигляді X, а модель для обробки та збереження інформації має значущу потребу до CPU у вигляді Y, то якщо обирати перший варіант, там где R-модель та CUD-модель знаходяться в одному сервісі, то коли такий сервіс розгортатимуть у хмарі, потрібно буде орендувати сервіс з характеристиками, що одразу задовольнятимуть вимоги і до RAM, і до CPU, тобто потрібно буде платити відповідні кошти і за змінну X, і за змінну Y, де не можна буде обрати щось одне [5].

У випадку, коли ми маємо відповідні R- та CUD-моделі у вигляді різних незалежних сервісів, то за аналогічних вимог до заліза можливо орендувати сервери, що окремо задовольнятимуть потреби RAM для моделі для читання, і окремо потреби до CPU для моделі для запису інформації [5].

У підсумку, варіант з окремим сервісом для R-моделі виглядає більш вигідно, тому що масштабувати сервіси можна відповідно до поточних вимог користувачів. Наприклад, у певний момент часу можна запустити дуже багато моделей для читання, яким потрібна тільки RAM, якщо на такий функціонал наразі є попит. У такий спосіб можна економити кошти та не витрачати зайві кошти на CPU. Однак важливо зазначити, що створення додаткових сервісів та їх подальша підтримка – це нагальне питання, яким не можна нехтувати. Чим більше сервісів у системі, тим більше людських ресурсів потрібно залучувати до підтримки таких систем. У такому випадку потрібно знаходити баланс між потребами у масштабуванні сервісів та їх питанням до подальшої підтримки [5][8]. Візуалізацію прикладу відношення потреби у простоті

підтримки сервіс проти його потреб до масштабування наведено на рисунку 1.3.3.3.

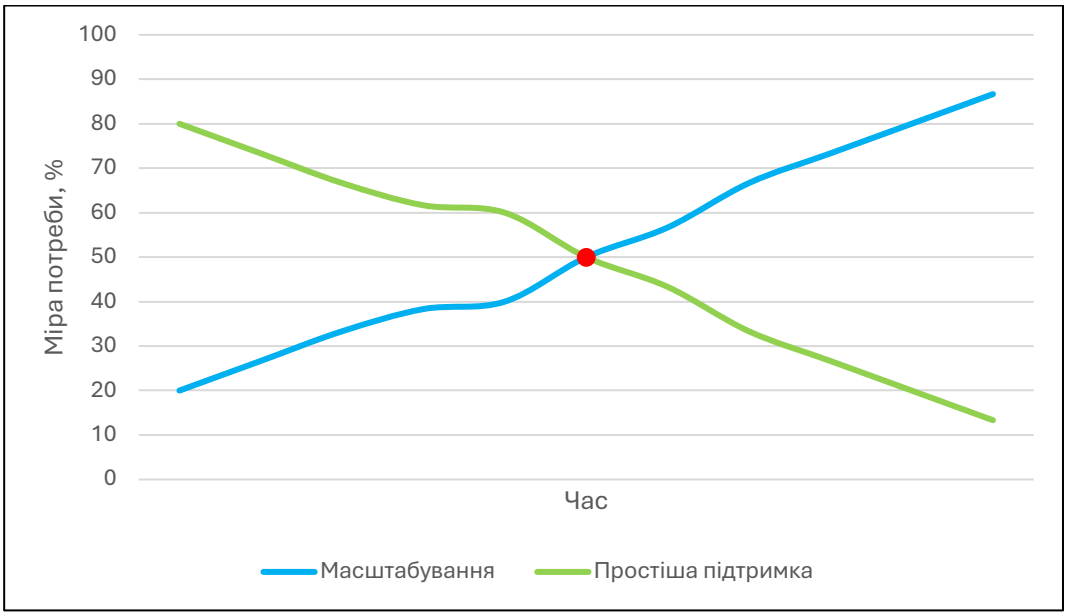

Рисунок 1.3.3.3 – Приклад можливих потреб проєкту з плином часу

З рисунку 1.3.3.3 маємо, що за певного моменту, що позначено червоною точкою, потреби до масштабування сервісу перевищують потреби до простої підтримки. Якщо потреби до підтримки є головнішими на даний момент, то вигіднішим буде будувати CQRS у межах одного сервісу. Інакше, якщо потрібна максимальна можливість масштабування сервісів, то краще розділяти моделі на різні сервіси. У випадку даного рисунку, червона точка означає точку зміну переважання трендів щодо потреб проєкту. Якщо CQRS було побудовано як один сервіс, то це може призвести до певного технічного боргу (technical debt) у майбутньому, тому при побудові лінії трендів підтримки сервісів проти їх масштабованості варто зважати на можливий подальший розвиток проєкту у наступний рік та три роки та обирати варіант CQRS, що пасує потребам та нефункціональним вимогам проєкту [8].

У межах даної роботи буде розглянуто побудову CQRS у вигляді окремих сервісів для можливості навести приклад архітектури, що всюди реалізується у великих проєктах та використовується найчастіше. Надалі у

даній роботі мова буде йтися лише про реалізацію з відокремленням R- та CUD-моделей в окремі сервіси.

1.3.4 Приклади використання

CQRS має широке використання у мікросервісній архітектурі та є одним з головним шаблонів проєктування сервісів. Головними прикладами використання цього шаблону є наступні варіанти:

- наявність складної логіки отримання інформації, що включає у себе складні запити до бази даних, які краще виокремити в окремий аналог матеріалізованих виглядів (materialized view) з баз даних у вигляді окремого сервісу, що оптимізований для швидкого отримання такої інформації. Тут можна застосовувати сховища даних, що більше оптимізовані для отримання інформації, аніж для його запису. Сюди можна віднести NoSQL-рішення, наприклад, Redis, що зберігає інформацію пари ключів-значень в RAM для пришвидшення доступу до таких даних, або MongoDB, що є документоорієнтованим сховищем інформації у вигляді JSON, де для створення виглядів (view) інформації зі складною структурою та багатою кількістю зв’язків для оптимізації її отримання можна виконати денормалізацію даних;
- потреба у масштабуванні сервісів, коли існує великий попит на читання даних, але немає потреби також їх записувати у великому масштабі кількості запитів;
- розділення принципів відповідальності сервісів. У такий спосіб хоча для певної доменної сутності існують більші вимоги для підтримки таких сервісів, але у загальному випадку систему, що складається з атомарних сервісів, таких, що відповідають за один тип операцій, працюють лише над однією доменною проблемою, то у перспективі існує можливість сумарного спрощення підтримки та розгортання таких систем тощо [8].

Але існують і випадки, коли краще утриматися від реалізації такого шаблону. Основними варіантами є наступні ситуації:

- створення MVP застосунку. У цьому випадку можна не розділяти моделі для читання та для запису на різні сервіси. Взагалі, коли будують першу версію продукту, зазвичай проєкти, особливо стартапи, будують за принципом монолітної архітектури замість мікросервісної. Тому якщо ціллю є створити якусь систему швидко та випустити її на ринок, краще взагалі відмовитися від мікросервісної архітектури;
- відсутність потреби у підвищені продуктивності або масштабованості системи. У багатьох проєктах достатньо використовувати CRUD-моделі у межах одного сервісу, що є поширеною практикою для невеликих проєктів;
- брак досвіду у команди розробників. Це дуже важливий пункт, який варто враховувати у будь-якій ситуації. Якщо розробники не знайомі з певними архітектурними шаблонами, краще відмовитися від їх використання. У поточний час потрібно провести тренінги для навчання розробників новим шаблонам. Але навіть тоді команда може бути ще не готовою для викликів нових технологій у справжньому великому проєкті. Можна почати розробку певного сервісу або його наборів класичною CRUD-моделлю, а після того, як у команді буде більше досвіду та розуміння проблем, які можуть виникнути у подальшому, тоді можна імплементувати CQRS: спочатку вибірково, у некритичних сервісах, що не мають високого навантаження, а після того, як досвіду стане ще більше, тоді можна переходити до більш критичних сервісів. Алгоритм саме такий, тому що якщо розробляти CQRS без досвіду одразу, весь застосунок можна зламати, а продуктивність такої системи з CQRS може бути ще меншою, ніж за звичайного CRUD-сервісу [8].

1.3.5 Переваги та недоліки

Кожен архітектурний шаблон не є ідеальним. Будь-які підходи у сучасній розробці програмного забезпечення мають власні переваги та недоліки. Задача архітектора – проаналізувати ризики та технічний борг щодо даних недоліків та порівняти їх з перевагами, що пропонує підхід. Також беруться до уваги бізнес-вимоги, а особливо нефункціональні вимоги до проєкту. У сумі даних показників обирається підхід, що завдасть найменшої шкоди у наступні декілька років, а його переваги будуть значимими для проєкту під час його поточного розвитку. CQRS не є виключенням: цей шаблон має недоліки, що потрібно проаналізувати перед його імплементацією.

Щодо переваг, CQRS має наступні пункти, що варті уваги:

- можливість імплементації складних запитів до бази даних сервісу так, що таких запитів насправді не буде відбуватися, а вся інформація такого сервісу будуватиметься на принципі споживання подій щодо змін станів доменних сутностей або на основі інших повідомлень. У такий спосіб вигляд певної інформації створюється інкрементально, і навіть найскладніші варіанти наповнення сховищ з оптимізацією для читання зазвичай проходять швидко та легко, а отримання такої інформації надалі з сервісу для читання відбуватиметься без додаткових затримок на опрацювання даних, тому що вони завжди готові для читання;
- можливість створення виглядів даних, що мають у собі інформацію з різних сервісів без потреби мати негайний доступ до таких сервісів, що підвищує доступність такого сервісу для читання та зменшує кількість мережевих стрибків для отримання певної інформації. У такий спосіб модель для читання в CQRS може накопичувати інформацію з інших сервісів та негайно надавати її клієнту;
- забезпечення єдиного принципу відповідальності, коли кожен сервіс відповідає за одну проблему: або лише читання даних клієнтом, або лише їх запис ним. Завдяки цьому маємо спрощення загальної

підтримки застосунку та забезпечення точкового масштабування саме потрібних сервісів, як було зазначено у підпункті 1.3.2 [5][8].

Даний шаблон має також і недоліки. Головними з них є:

- підвищення складності архітектури застосунку через те, що у сервісі лінійно збільшується кількість сервісів, що потрібно реалізувати задля виконання тієї самої бізнес-логіки, що можна було би вирішити без впровадження додаткових рішень, але з більшим технічним боргом у майбутньому. У такому випадку створюють нові команди розробників, що відповають за окремі домени проблем у застосунку. Хоча це добрий варіант, але варто враховувати, що на кожного розробника та команду відповідно потрібно проводити додаткові тренінги для їх навчання бізнес-моделі поточного застосунку та ознайомлення з процесами у поточному проєкті. Це може бути дорого. Особливо, якщо розробники є професіоналами високого рівня та потребуватимуть відповідного гідного рівня оплати праці;
- підвищення часу очікування у межах кінцевої узгодженості, де після того, як CUD-модель оновила дані та створила подію, для R-моделі потрібен певний час, щоби отримати дану подію, обробити її та зберегти у власне сховище. На показник затримки оновлення моделі для читання впливають пропускна здатність мережі, її навантаження, географічне розташування між сервісами, потужність сервісів тощо.

Даний сервіс має недоліки, що можуть бути критичними у багатьох випадках. Однак якщо маємо великий проєкт на мікросервісах, що потребує точкового масштабування та оптимізації швидкості отримання клієнтом відповіді на взяття даних, то переваги CQRS значно перевищують його недоліки, тому у загальному випадку цей шаблон можна рекомендувати до імплементації [5].

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Аналіз доменної області

У даній роботі взято до уваги розробку API-застосунку на тему кінофільмів із можливістю їх придбання різними способами: напряму за фіксовану визначену суму або безлімітно без додаткової плати після оформлення підписки. Задля проєктування такої системи пропонується розглянути підхід Domain-Driven Development (DDD) щодо створення архітектури такого застосунку. Основна ідея – це використання термінів корінь агрегату та його значення. Агрегат – це певний набір сутностей, що можна розглядати як єдине ціле. По-перше маємо аналіз аналіз предметної області та визначення ключових бізнес-моделей [12]. Задля даного процесу проаналізовано типові варіанти використання аналогічних сервісів на прикладі Netflix. Надалі створено високорівневу модель сутностей. Маємо наступні моделі:

- фільм – має основні типові властивості (назва, рік тощо), а також ціну, що включені в кореня агрегату. Додатково включено також акторів та команду зйомки, що відносяться до значень агрегату;
- профіль користувача – включає базові властивості, наприклад, електронну пошту;
- замовлення – це певний запис із фільмом та профілем користувача, статусом (на розгляді, прийнято, відхилено, відмінено);
- підписка – вид сучасного квитка-перепустки до безлімітного замовлення фільмів на певний строк, має назву та ціну;
- членство – представляє пропуск користувача до безоплатного замовлення фільмів, поки воно активне. Це особливий вид замовлення, що має у собі користувача, тип підписки та дату наступного платежу. Може автоматично поновлюватися за потреби.
- платіжна операція – використовується при оплаті замовлень та підписок, має у собі користувача та поточний баланс.

Маємо приблизну UML-діаграму класів моделей застосунку на рисунку 2.1.1.

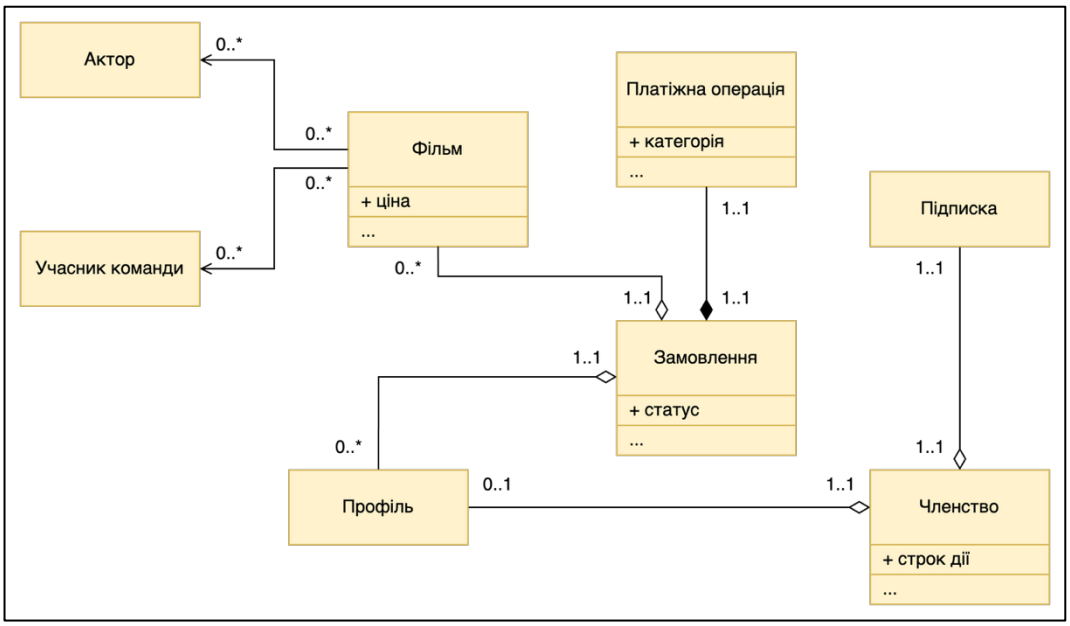

Рисунок 2.1.1 – Початкова UML-діаграма класів моделей

З рисунку 2.1.1 маємо наступне: наприклад, фільм може містити будь-яку кількість акторів та учасників команди, і навпаки, учасники команди та актори можуть бути призначені до будь-якої кількості фільмів, включно з нульовою кількістю. Так само описуються інші зв’язки.

Однак на даному етапі маємо недостатню абстракцію між сервісами для створення мікросервісів. Для цього потрібно виконати інкапсуляцію певних моделей частин доменної моделі так задля виділення агрегатів. Маємо наступну доменну модель у вигляді агрегатів на рисунку 2.1.2. Звідси отримано відповідні корені агрегатів та їх значення. Наприклад, в агрегаті фільмів коренем є саме сутність фільм, а одним з його значень полів є актори та команда учасників. У такий спосіб реалізовано укрупнення сутностей у такі, що можна представити у вигляді саме агрегатів, тому що надалі буде розглянуто мікросервіси, що будуються на принципі єдиної відповідальності, де один сервіс буде представляти нуль: якщо сервіс відповідає за роботу з бізнес-моделлю, він має один агрегат. У випадку, якщо матимемо утилітарний

сервіс, то він може не представляти жоден з агрегатів, а лише його використовувати при певних запитах до такого сервісу. Надалі одним з таких сервісів буде поштовий сервіс, що у собі не має моделей, але за запитом він надсилатиме листи.

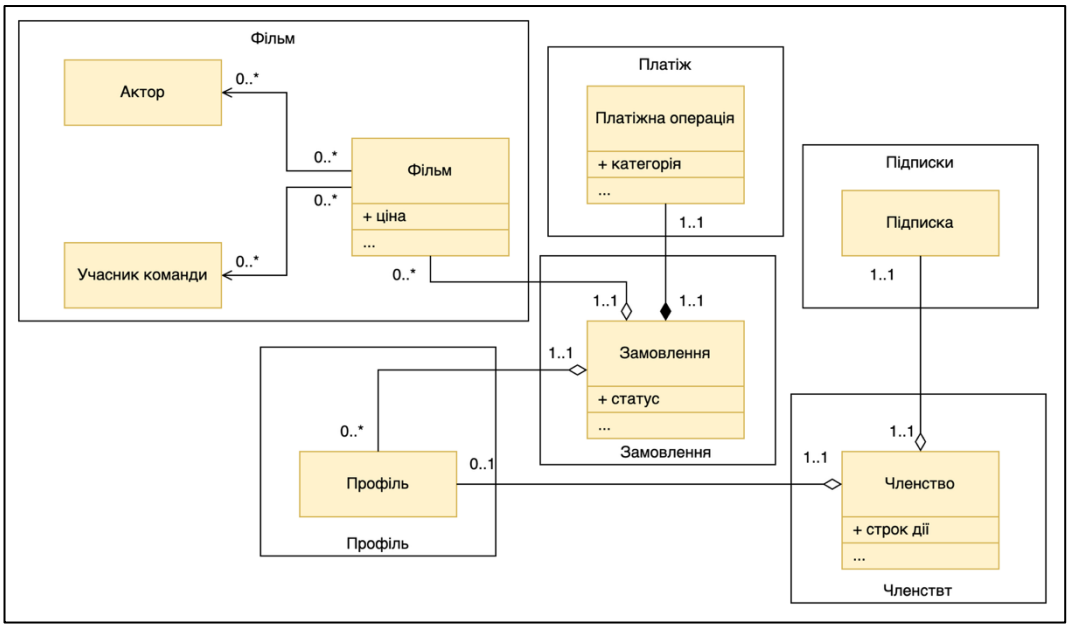

Рисунок 2.1.2 – Результат виділення агрегатів

Після виділення агрегатів маємо питання: наприклад, у замовлення є більше складових, ніж власне модель замовлення: у нього є і профіль, і фільм, і платіж тощо. У такому випадку є бажання переробити аграгат так, щоби все включити у відповідний агрегат замовлення. Але за правилами DDD в агрегат потрібно включати лише такі частини, що є складовими саме до певної одиниці моделі: наприклад, до фільмів входять актори та команда, вони відносяться саме до однієї сутності: фільмів. А у випадку із замовленням воно лише має різні види зв’язків: слабкий, тобто агрегацію у випадку UML-моделювання, та сильний, композицію [12].

У наступному підрозділі буде розглянуто комплексне проєктування сервісів для мікросервісної архітектури з відповідними зв’язками на рівні інфраструктури та додатковими сервісами, що будуть використані для реалізації бізнес-логіки.

2.2 Event Storming

Після визначення ключових агрегатів визначено, що наступним етапом проєктування міжсервісного спілкування та архітектури застосунку взагалі є етап створення моделі повідомлень: команд та подій.

Одним з підходів є Event Storming («шторм подій»), що визначається як процес більш детального командного аналізу можливостей застосунку та предметної області, що має на меті представлення шляхів спілкування сервісів між собою, а також визначення загальних можливостей застосунку. Цей процес є здебільшого неформальним, до якого залучаються програмні розробники та експерти доменної області. Вони разом вирішують як має працювати застосунок та які можливості він матиме [13].

Під час моделювання системи буде взято до уваги наступні компоненти:

- агрегати – набір сутностей, що можна розглядати як єдине ціле. У даному випадку вони представлятимуть у більшості CUD-моделі, що оновлюватимуть стан агрегатів та будуть представлені у вигляді джерел зміни стану агрегату;
- команди – певні повідомлення, що означають задачу, що потрібно виконати агрегатом;
- подія – результат зміни стану агрегату;
- модель для читання – друга частина шаблону CQRS, яка відповідатиме за представлення даних клієнту в оптимізованому представленні;
- зовнішній сервіс – інший сервіс у межах поточного застосунку, який може підписатися на певну подію та викликати певну бізнес-логіку, за яку відповідає даний сервіс. У такому випадку подія є каталізатором щодо старту події [13].

У даній роботі реалізовано процес Event Storming за наступною схемою, що попередньо обговорено з науковим керівником та створено, на рисунку 2.2.1.

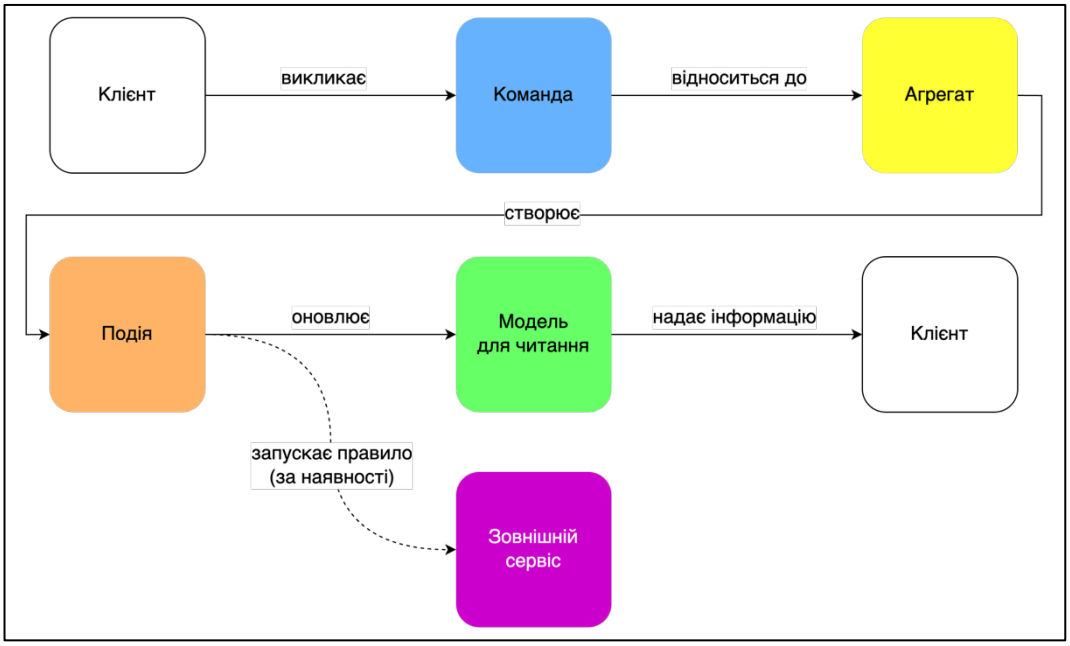

Рисунок 2.2.1 – Відношення сутностей в Event Storming

Головною перевагою даного підходу є глибше розуміння предметної області, а також скорочення часу проєктування системи з днів та тижнів до годин [13]. У межах даної роботи обрано саме такий підхід через потребу змоделювати які міжпроцесні взаємодії можливі у даній системі, тому що, як було зазначено раніше, у випадку CQRS така асинхронна взаємодія у вигляді обміну повідомленнями між сервісами – це ключова особливість можливості масштабованості мікросервісних систем.

Після визначення основних елементів, що будуть братися до уваги під час Event Storming, створено неформальну сесію обговорення структури комунікації у додатку, яка тривала дві години.

У наступних посиланнях на рисунки додатку А буде наведено приклади візуалізації даного процесу англійською мовою для спрощення перекладання бізнес-термінів на мови програмування, де найкращими практиками є написання програмного коду саме англійськими термінами. Додатково, англійська мова є досить поширеною у світі, і тому якщо до проєкту залучаться розробники з інших країн, результати процесу Event Storming можна використати для подальшого прикладу архітектури спілкування у даному проєкті, що спростить процес адаптації нових розробників до нового проєкту.

На початковому етапі визначено агрегати, які виділено раніше, та команди, тобто функціонал, який потрібно реалізовувати у даних агрегатах. На цьому етапі ще нема конкретної моделі системи. Тут визначаються лише до вимоги виду «що агрегат має вміти». Результат першого етапу моделювання розміщено на рисунку А.1 додатку А.

На другому етапі до схеми додано події, що мають статися на відповідь на команди. Тут головний аспект – це визначення такої події, що показує однозначне успіх, невдачу або інформаційне повідомлення, яке вже сталося, тому тут зазначаються дієслова у минулому часі. Інакше не можна зрозуміти: чи команда виконається, чи не виконається у підсумку, тому що на обіцяння певної команди виконатися не можна брати до уваги, на неї не можна розраховувати, тому що тут є велика вірогідність призвести систему до неконсистентного стану, який потім складно обробляти та виправляти [8].

Використовуються саме події минулого часу через те, що коли подія вже сталася, можна розраховувати на те, що всі сервіси однозначно зрозуміють надану інформацію, а також можуть розраховувати на неї. Додатково, у такому разі можна всім сервісам, що зацікавлені у повідомленні, одразу реагувати на надану інформацію події. У такий спосіб підвищується консистентність та загальна продуктивність системи [13].

Результат додавання подій до команд агрегатів наведено на рисунку А.2 додатку А. Додатково варто відмітити, що на цьому рисунку розміщено події, які запускаються на запит іншої події (команди). У такому випадку можна буде такі дії виокремити в окремі сервіси, що будуть виконувати задану дію, щоби не перевантажувати логікою основний сервіс агрегату задля зниження складності підтримки такого застосунку.

У кінцевому етапі моделі для читання винесено в окремі сервіси, які будуть насичуватися подіями підписавшись на відповідні оновлення агрегатів. Додатково, важливо зазначити, що такі моделі додано не тільки для читання одного агрегату, а й для декількох, що визначають композицію декількох елементів, що представляють єдину сутність. У такому випадку, наприклад, у

клієнта наявна можливість в оптимізованому вигляді отримувати інформацію одразу з декількох агрегатів різних сервісів без прив’язки до їх статусу онлайн [8]. Також додано спеціальний, більш адміністративний сервіс, який представляє собою аудит-логування подій у застосунку, що надає наступні переваги:

- аналіз аномалій та підозрілих дій у застосунку акумулювання подій;
- перевірка минулих дій користувача для аналізу безпеки;
- відстеження помилок у застосунку, особливо у продакшені тощо.

Це напряму показує ключову перевагу CQRS та Event Sourcing – масштабування з низьким зчепленням між частинами застосунку, бо у такому випадку додається окремий сервіс, що підпишеться на майже всі події застосунку без додавання інформації про своє існування в інші сервіси, що займаються бізнес-логікою [2][5][8].

Також додано саги, що займаються обробкою замовлень. Сага – це серія локальних транзакцій між сервісами, що координуються асинхронним спілкуванням. У такому випадку, одним з можливих варіантів є використання центрального координатору саги у вигляді оркестратору – це окремий сервіс, що має певний стан та відстежує етапи виконання команд через отримання відповідей від сервісів, що виконують такі команди. Напряму цей шаблон консистентності даних не впливає на CQRS, однак це також ще один популярний підхід до розробки масштабованих систем у мікросервісній архітектурі. Лише важливо зазначити, що наприкінці своєї діяльності саги публікують нові повідомлення, якими можуть бути зацікавлені моделі для читання. Тобто це ще один спосіб керування командами та подіями, але у контексті міжсервісної взаємодії [8].

Додатково, важливо обґрунтувати рішення в одному випадку винести сервіс для надсилання електронної пошти в окрему частину застосунку, де мається на увазі відправка листів щодо підтвердження нових профілів в застосунку, а в іншому – додати функціональність надсилання повідомлення поштою прямо в сервіс замовлень. У мікросервісній архітектурі важливо не

перевантажувати один сервіс бізнес-логікою, тому що на певному етапі розробки може статися ситуація, коли підтримка командою такого сервісу стане складнішою. Особливо це стане очевидним, якщо у сервіс вже одразу вкладається можливість додавання складної логіки.

Тому у першому варіанті з підтвердженням профілю даний функціонал винесено в окремий сервіс через показ приклад того, що підтвердження користувача – це не критичний функціонал, його можна без проблем винести в окремий сервіс.

Однак в аналогічній ситуації з замовленнями, де після підтвердження успіху замовлення маємо негайну відправку повідомлення із резюме замовлення, можна взяти варіант, коли за законами певних країн після створення та підтвердження замовлення є потреба надсилати результат поштою. У випадку, коли надсиланням листів займається той самий сервіс, є більша вірогідність, що у той самий час, як подія дістанеться цього сервісу замовлень, цей сервіс продовжить роботу та успішно надішле повідомлення. Тобто у такому випадку не матимемо високого зчеплення з іншим сервісом, же не потрібно сподіватися на коректну роботу іншого сервісу, що відповідає за критичний функціонал [8].

Результат фінального етапу моделювання системи за Event Storming розміщено на рисунку А.1 додатку А.

2.3 Проєктування повідомлень

Після виконання Event Storming тепер ми маємо неформальний список бізнес-операцій, що будуть виконуватися під час міжсервісної взаємодії. Вони представлені у вигляді команд та повідомлень. Важливо зазначити, що розмір повідомлень напряму визначає продуктивність системи, тому що за великої кількості повідомлень завеликого розміру може зрости затримка оновлення інформації у сервісах через те, що надана кількість інформації має проходити через комп’ютерну мережу, щоби дістатися до брокера повідомлень, а звідти до всіх зацікавлених клієнтів.

Для всіх змодельованих повідомлень за допомогою Event Storming у програмному коді реалізовано команди та події, що надалі будуть використані як складова для міжсервісного спілкування та подальшого розширення функціоналу застосунку.

2.3.1 Структура команд

Команда – це запит на виконання певної дії. Зазвичай сюди додають тіло запиту, що представляє певну роботу, яку потрібно виконати. Команди називають у такий спосіб: фраза з дієсловом та слово «command». Приклад: UpdateProfileCommand. Такі повідомлення публікуються у брокер повідомлень та розповсюджуються на всі зацікавлені сервіси [8].

Приклад команди наведено на лістингу 2.3.1.1.

Лістинг 2.3.1.1 – Приклад команди UpdateProfileCommand

```
public record UpdateProfileCommand(
  String profileId,
  String email,
  String name) {
}
```

Додатково варто зазначити, що у межах команди сервіс, що буде створювати таку команду, має додати туди всю інформацію, що потрібна іншим зацікавленим сервісам. Інакше цим сервісам потрібно буде здійснюватися блокуючу взаємодію з сервісом, що має потрібну інформацію, що знизить загальний рівень доступності сервісів.

2.3.2 Структура подій

Подія – це результат виконання певної праці або повідомлення про її закінчення. Таке повідомлення має схожу структуру на команду, але вже показує її результат виконання. Зазвичай у межах Event Sourcing використовується підхід, що команди та події відносяться до агрегатів, тому якщо маємо команду UpdateProfileCommand, то після обробки даної команди

сервісом, що відповідає за профілі, створить фактичний доказ оновлення профілю, що буде відображений подією ProfileUpdatedEvent. У такий спосіб подія є також єдиним джерелом правди, на який можуть опиратися інші сервіси, що зацікавлені у даних агрегатах [8]. Приклад події ProfileUpdatedEvent наведено на лістингу 2.3.2.1.

Лістинг 2.3.2.1 – Приклад події ProfileUpdatedEvent

```
public record ProfileUpdatedEvent (
  String profileId,
  String email,
  String name) {
}
```

Звідси маємо, що подія повністю копіює структуру команди, тому що команда – це лише запит на виконання дії, а подія – це її результат, тому такий результат може дублювати інформацію, особливо у межах класичних CRUD-сервісів. Хоча це може бути трохи незручно, але у такий спосіб спрощується підтримка сервісів, що використовують асинхронну взаємодію через повідомлення, тому що їм потрібно лише підписатися на отримання таких подій без влаштування додаткової інфраструктури.

2.4 Сховище подій

2.4.1 Event Sourcing

Під час розробки застосунку постало питання консистентності системи у випадку обміну подіями між сервісами. Якщо виконувати асинхронний міжсервісний обмін повідомленнями, то тут потрібна інфраструктура для збереження таких повідомлень, тому що від надсилання повідомлення до його отримання одержувачем може пройти велика кількість часу, а одночасно тримати онлайн ті самі екземпляри сервісів не завжди є можливість [4].

На вирішення цієї проблеми вирішено використовувати шаблон консистентності мікросервісної архітектури Event Sourcing, що надає можливість збереження зміни станів агрегатів у вигляді серії подій, що може

споживатися іншими сервісами. Додатково, тут також має місце певне сховище подій, що іноді називають ланцюжком подій, яке є єдиним джерелом правди (single source of truth), де лише звідси всі сервіси отримують повідомлення [4]. Це продовжує загальну ідею подієво-орієнтованої архітектури мікросервісів, але стандартизує процес обміну такими повідомленнями у такий спосіб, що надають відповідні переваги.

Основними перевагами такого шаблону є наступні пункти:

- надійна публікація подій – під час збереження повідомлень у ланцюжок подій, то це означає, що певна подія дійсно відбувалася, і надалі всі сервіси, що зацікавлені у зміні станів агрегатів можуть отримувати такі зміни і не піддавати сумніву чи дійсно відбулася подія, чи відбувалася вона один раз чи більше разів. Зазвичай події публікуються один раз, і повторні публікації того самого повідомлення відкидаються ланцюжком подій;
- зберігання історії подій – якщо сервіс публікує подію про зміну стану агрегату, то така подія запам’ятовується системою назавжди. Після цього навіть за відмови всіх сервісів опубліковані події можна перечитати з ланцюжка подій зі збереженням всіх змін, що були впроваджені до агрегатів впродовж всього часу роботи сервісів;
- можливість аудиту системи – через те, що в ланцюжок подій зберігаються всі повідомлення, наявна можливість імплементувати додатковий сервіс, що буде споживати такі події та робити відповідні висновки, що означають статистичну інформацію щодо користування такою системою. Так як Event Sourcing працює безперервно, то і аудит можна робити інкрементально, тобто із застосуванням CQRS, безперервно та вчасно повідомляти аудиторів у разі виникнення помилок у системі, аномалій щодо поведінки користувачів тощо [4][8].

З іншого боку, у такого шаблону є недоліки:

- потреба змінювати модель програмування – у такому випадку неможливо здійснювати класичні CRUD-операції над сутностями. Тут потрібно інкрементально наповнювати стан агрегатів відповідно до подій. Такий варіант є трохи складнішим для розробки, а також потребує відповідного досвіду у команди розробки;
- потреба мати додаткову інфраструктуру – для збереження повідомлень потрібний інший спеціальний сервіс, що буде виступати у ролі Event Store, сховища для збереження подій, а всім клієнтам використовувати відповідне API для підписки на нові повідомлення з такого сховища [8].

У такому випадку переваги перевищують недоліки, хоча складність налаштування та проблема додаткової інфраструктури потребують вирішення. Додатково, визначено, що такий шаблон вигідно використовувати разом з CQRS, тому що він надає можливість більшого масштабування системи через те, що Event Store – це єдине джерело правди подій, звідки маємо, що отримуємо мінімальне зчеплення між сервісами, тому що кожен сервіс з моделі CQRS потребуватиме лише спілкування з одним сервісом: із сховищем подій [8].

2.4.2 Імплементація сховища подій

Event Store – це гібрид одночасно бази даних та брокеру повідомлень [8]. У такому випадку постає питання вибору інструмента для сховища подій.

2.4.2.1 Apache Kafka як брокер повідомлень

Спершу можна розглянути випадок з Apache Kafka. Це брокер, що має широкі можливості підтримки роботи з високонавантаженими системами.

З одного боку, Kafka – це інструмент для надсилання та отримання повідомлень, що зберігаються у пам’яті на визначений строк. Event Sourcing вимагає, щоби події зберігалися нескінченно велику кількість часу, тому що

коли розгортаються нові сервіси, вони мають прочитати всі повідомлення зі сховища: від найпершого до останнього. Додатково, якщо клієнт у вигляді сервісу підпишеться на певний топік, то йому прийдеться продивлятися вміст кожного з повідомлень задля аналізу чи не належить певна подія поточному агрегату з сервісу [9].

З іншого боку, Kafka має наступні переваги:

- гарантія доставки повідомлень. За випадку, якщо повідомлення досягає Kafka, воно тут гарантовано залишається на визначений строк;
- можливість перечитання історії подій сервісом, де Kafka представляє собою потік повідомлень, які можна прочитати визначивши потрібний індекс для початку читання повідомлень;
- підтримка зовнішніх розширень – Kafka є популярним брокером, яким користуються по всьому світу. Через це він сьогодні є майже стандартом у сфері журнальних брокерів. Звідси, за потреби до Kafka можна також підключити інші зовнішні системи, якщо є потреба, а також додавати інші розширення до системи [8][9].

2.4.2.2 Axon Framework як зручне API для Event Sourcing

Одним з прикладів справжнього прикладу Event Store є технологія Axon Framework, що надає можливість одразу мати і брокер подій, і сховище у вигляді бази даних. Він нагадує Kafka, але є більш спеціалізованим інструментом, який можна використовувати у застосунках, що реалізовують Event Sourcing у межах мікросервісної архітектури.

Додатково, Axon надає можливості використання спеціалізованого API задля написання застосунків, що акцентують увагу на розробці за підходом DDD. У такому випадку, можливо є вигідним використовувати одразу Axon замість Kafka, але у якості прикладу та спрощення інфраструктури розробки застосунку надалі буде використано саме Kafka для доставки повідомлень до клієнтів, однак задля інтеграції зручного API для

виконання збагачення агрегатів та написання зручних слухачів подій додатково буде використано Axon Framework.

У загальному випадку маємо наступну архітектуру роботи з даним фреймворком, що можна використовувати як надбудову над Kafka [14], на рисунку 2.4.2.2.1.

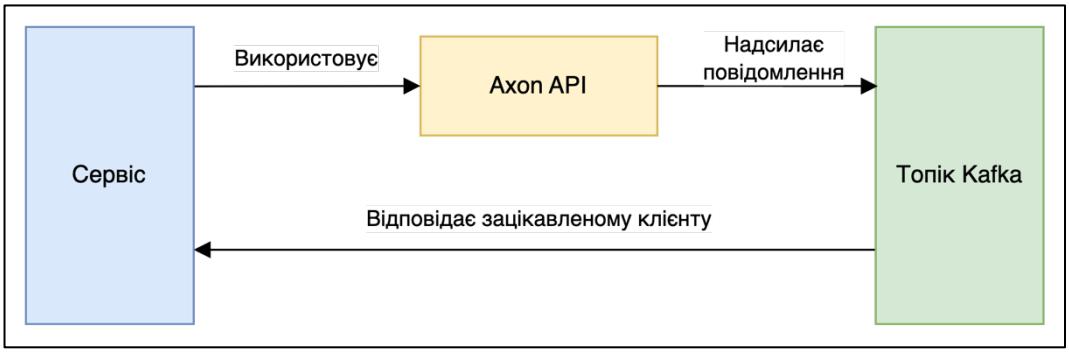

Рисунок 2.4.2.2.1 – Архітектура надсилання та отримання повідомлень

Під час створення інфраструктури застосунку для мікросервісів обрано платформу контейнеризації Docker, що надає можливість розгортання програмних засобів у контейнерах. У варіанті піднімання Event Store та подальших сервісів також застосовано таку платформу. Приклад налаштування Axon наведено у лістингу Б.1 додатку Б. Варто зазначити, що у цьому та наступних проєктах Docker Compose використано наступні ключові слова, що описують правила розгортання контейнерів:

- image – визначає базовий образ сервісу;
- environment – визначає налаштування через змінні оточення;
- volumes – створює посилання від шляху з контейнера до реального шляху операційної системи (ОС), що надасть змогу зберегти дані між перезапусками контейнеру;
- ports – відображення портів з контейнеру до ОС [14].

2.5 CQRS

Під час розробки застосунку написано 12 сервісів за шаблоном CQRS. Всі застосунки розроблено за допомогою Java-фреймворку Spring Boot, що має можливості до підключення потрібних модулів для створення сучасних застосунків, у тому числі мікросервісів. Надалі представлено деякі найпоказовіші приклади реалізації сервісів за даним шаблоном. Всі моделі (для читання та для запису) розділено на окремі сервіси задля показу можливостей незалежного масштабування сервісів, а також можливість написання більш складних сервісів [8].

2.5.1 Сервіс фільмів

2.5.1.1 Модель для запису даних

У найпростішому варіанті маємо реалізацію CRUD-сервісу, але у вигляді двох окремих сервісів: один з них реалізовує модель для запису інформації, інший – лише для читання. Так як використовується підхід Event Sourcing, то всі асинхронні операції у сервісах поділяються на два типи:

- створення, оновлення та видалення профілю – ініціюється клієнтом через синхронну взаємодію; знаходиться на сервісі для запису даних;
- читання сутностей профілів – також ініціюється клієнтом, але насичення історії змін профілів виокнується через повідомлення.

Під час реалізації такого сервісу використано наступні залежності з Spring Boot, що розміщено на лістинг Б.2 додатку Б. Синхронну взаємодію з клієнтом реалізовано через фреймворк Spring Web, що застосовує спілкування через HTTP. Після цього створено модель фільмів, що представлено сутністю, яка обробляється через API Axon, що успішно застосовує Event Sourcing разом з Kafka. Приклад такої моделі розміщено на лістингу 2.5.1.1.1. У такому варіанті модель є центральним джерелом стану агрегатів, які обробляють вхідні команди та вирішують чи приймати пропоновану зміну через створення події. У загальному випадку у класичному CRUD-сервісі немає складної

бізнес-логіки, тому під час отримання вхідної команди маємо відразу створення події, у якій надалі всі зміни зберігаються у сутність.

Лістинг 2.5.1.1.1 – Фрагмент агрегату фільмів MovieController

```
@Aggregate
public class Movie {
    @AggregateIdentifier
    private String id;
    private String title;
    private Integer releaseYear;
    private BigDecimal price;

    // ...

    @CommandHandler
    private void process(UpdateMovieCommand command) {
        apply(new MovieUpdatedEvent(command.movieId(),
command.movie()));
    }

    @EventSourcingHandler
    private void handle(MovieUpdatedEvent event) {
        this.id = event.movieId();
        this.title = event.movie().title();
        this.releaseYear = event.movie().releaseYear();
    }

    // ...
}
```

Відповідно, для спілкування з клієнтом написано веб-контролер з шаблону проєктування веб-застосунків Model-View-Controller (MVC), що наведено у якості прикладу на лістингу 2.5.1.1.2.

Лістинг 2.5.1.1.2 – Фрагмент контролеру MovieController

```
@RestController
public class MovieController {
    private final CommandGateway commandGateway;

    @PutMapping("/{id}")
   CompletableFuture<Movie> update(String id, MovieDto movie) {
        var updateCommand = new UpdateMovieCommand(id, movie);
        return commandGateway.send(updateCommand);
    }
}
```

2.5.1.2 Модель для читання даних

Як було зазначено раніше, модель для читання використовує доменні події для насичення власної колекції агрегатів через підписання на такі повідомлення. У випадку сервісу для сутності фільмів, створено окремий сервіс, що має схожу структуру класичного застосунку Spring Boot, але тут наявна важлива відмінність – для збереження та подальшого відображення екземплярів агрегатів клієнту потрібно їх десь зберігати. У випадку Axon їх менеджмент у межах сервісу відбувається у RAM в оптимізованому вигляді для роботи у JVM [14], однак модель для читання має зберігати агрегати в оптимізованому вигляді саме для отримання їх клієнтом [2][8]. Тому найпростішим варіантом було вирішено підключити до проєкту Redis – це просте NoSQL-сховище, що також зберігається у RAM, але працює як окремий сервіс, що може зберігати нескладні структури даних у форматі «ключ-значення». Таке сховище підходить для збереження сутностей фільмів, тому обрано саме його через просту підтримку, розгортання та швидку роботу через те, що всі дані зберігаються в оперативній пам’яті. Підключення Redis є вкрай простим: у Spring Boot є окремий фреймворк Data Redis, де потрібно лише створити контейнер у Docker та надати адресу такого сервісу [15]. Приклад Compose-проєкту наведено на лістингу Б.3 додатку Б. Додатково, маємо приклад обробника подій, що наповнює колекцію фільмів, на лістингу 2.5.1.2.1.

Лістинг 2.5.1.2.1 – Фрагмент обробника доменних подій агрегату фільмів

```
@Component
public class MovieProjectionHandler {
    private final MovieNameRepository repository;
    private final MovieMapper mapper;

    @EventHandler
    public void on(MovieUpdatedEvent event) {
        var movie =
repository.findById(event.movieId()).orElseThrow();
        movie.setTitle(event.movie().title());
        repository.save(movie);
    }
```

```
}
```

У такий самий спосіб за допомогою Spring Web MVC [16], але тепер задля отримання даних, виконання GET-запитів до сервісу, написано наступний MVC-контролер, що розміщено на лістингу 2.5.1.2.2.

Лістинг 2.5.1.2.2 – Фрагмент контролеру отримання фільмів

```
@RestController
public class MovieNameController {
    private final MovieService movieService;

    @GetMapping
    public List<Movie> findAll() {
        return movieService.findAll();
    }
}
```

У такий спосіб реалізовано сервіс для роботи з фільмами, який клієнт бачитиме як класичний CRUD-сервіс, але розділений на два незалежні компоненти.

Архітектуру таких двох сервісів для запису та читання даних наведено на рисунку 2.5.1.2.1.

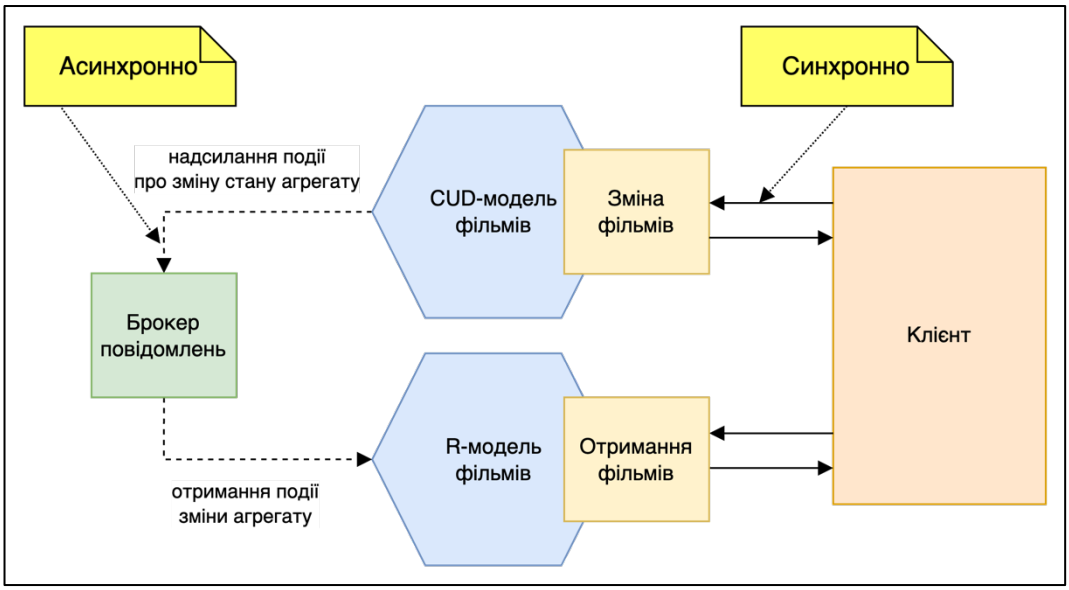

Рисунок 2.5.1.2.1 – Приклад спілкування CQRS-сервісів для агрегату фільмів

2.5.2 Сервіс замовлень

2.5.2.1 Модель для запису даних

В іншому випадку також створено відповідний агрегат для замовлень, який розміщено у лістингах Б.4 та Б.5 додатку Б відповідно. Структура такого сервісу відповідає аналогічній, що представлена у випадку агрегату фільму. Однак у даному випадку маємо більш складний варіант для обміну повідомленнями через те, що замовлення може змінювати статус.

По-перше, нехай замовлення має свій унікальний ідентифікатор, профіль користувача та фільм з ціною, що було замовлено. У такому випадку, якщо повністю перекласти модель розмірковування структури повідомлень у код, то отримаємо наступну подію щодо створення та оновлення замовлення відповідно на лістингу 2.5.2.1.1.

Лістинг 2.5.2.1.1 – Приклад подій замовлення та оновлення статусу замовлення

```
public record OrderCreatedEvent(String orderId,
String profileId, String profileName, String profileEmail,
String movieId, String movieName, String movieYear,
BigDecimal moviePrice) {
}

public record OrderConfirmedEvent(String orderId,
String profileId, String profileName, String profileEmail,
String movieId, String movieName, String movieYear,
BigDecimal moviePrice) {
}
```

За такого випадку маємо дублювання інформації. Однак такий процес можна оптимізувати так: під час створення замовлення дійсно потрібно передавати якомога більше інформацію у події, щоби всі зацікавлені сервіси отримали інформацію щодо замовлення. Але у разі мікросервісів, де існує шаблон CQRS, а також його похідний варіант – приватні проєкції агрегатів у сервісах, де кожен сервіс може зберегти для себе потрібну інформацію, можна не передавати кожен раз всі дані щодо оновлення замовлення. Якщо існує сервіс для обробки замовлень, при його створенні збирається вся потрібна

інформація, а при аналізі зміни статусу потрібно буде лише ідентифікатор замовлення, якого стосується така зміна, тому що на попередньому етапі сервіс вже запам’ятав всі поля замовлення [8][12].

Маємо новий варіант повідомлень на лістингу 2.5.2.1.2.

Лістинг 2.5.2.1.2 – Оновлений варіант подій щодо замовлення

```
public record OrderCreatedEvent(String orderId,
String profileId, String profileName, String profileEmail,
String movieId, String movieName, String movieYear,
BigDecimal moviePrice) {
}

public record OrderConfirmedEvent(String orderId) {
}
```

Однак тут є ще один варіант для зменшення розміру такого повідомлення: сюди потрібно додавати лише ту інформацію, що потрібна його споживачам [8][12]. Наприклад, за поточного варіанту бізнес-логіки, наприклад, замовленню не потрібно мати інформацію про користувача та фільм, окрім ідентифікатора користувача та назви та ціни фільму. Звідси, якщо прибрати зайві поля з події, можна отримати фінальний варіант події створення замовлення на лістингу 2.5.2.1.3.

Лістинг 2.5.2.1.3 – Фінальний варіант події щодо замовлення

```
public record OrderCreatedEvent(String orderId,
String profileId, String movieId, String movieName,
BigDecimal moviePrice) {
}
```

Саме такий варіант є найкращим щодо можливості пропускної здатності обробки повідомлень у межах застосунку, тому що навіть прибирання одного поля у перспективі може економити терабайти інформації, що передається мережею [8].

Щодо практичної реалізації сервісу він є аналогічним сервісу фільмів. Приклади програмного коду контролеру розміщено на лістингу Б.6 додатку Б.

2.5.2.2 Модель для читання даних

У даному варіанті отримання інформації про замовлення реалізовано аналогічно до сервісу з фільмами, але з відмінністю, що клієнт буде отримувати більшу кількість інформації, ніж передається у межах повідомлень щодо замовлень.

Звідси, CQRS-модель для читання агрегату замовлення виступає у ролі API-композитора, що надає доступ одразу до декількох агрегатів у межах отримання колекції [8]. Через це наявна можливість перевірки

Приклад реалізації такого сервісу наведено на лістингах Б.7-Б.9 додатку Б, який показує процес насичення моделей для читання через створення обробників подій для обробки змін стану агрегату фільмів, замовлення та профілю. У такий спосіб клієнт отримує можливість отримати повну інформацію щодо замовлення разом з іншими агрегатами. Звідси, маємо таку сутність замовлень у моделі для читання на лістингу 2.5.2.2.1.

Лістинг 2.5.2.2.1 – Модель замовлень

```
@RedisHash
public class Order {
  @Id
  private String id;
  private OrderStatus1 status = OrderStatus1.PENDING;
  private Movie movie;
  private Profile profile;

  public Order complete() {
    status = OrderStatus1.COMPLETED;
    return this;
  }

  public Order cancel() {
    status = OrderStatus1.CANCELED;
    return this;
  }

  // ...
}
```

У такому випадку маємо аналогічний MVC-контролер, що надає клієнтам повну інформацію про замовлення на лістингу 2.5.2.2.2.

Лістинг 2.5.2.2.2 – Фрагмент контролеру для отримання замовлень

```
@RestController
public class OrderController {
  private final OrderService service;

  @GetMapping("/{id}")
  public Iterable<Order> findSettledOrdersFor(String profileId){
    return service.findSettledOrdersFor(id);
  }
}
```

Звідси, коли клієнт захоче отримати інформацію щодо своїх замовлень, у той же момент можна буде надіслати йому відповідь без додаткових затримок щодо отримання інформації з різних сервісів, тому що всі агрегати вже збережено в одному сервісі-моделі для читання. Візуалізацію такого процесу у вигляді діаграми послідовності наведено на рисунку 2.5.2.2.1.

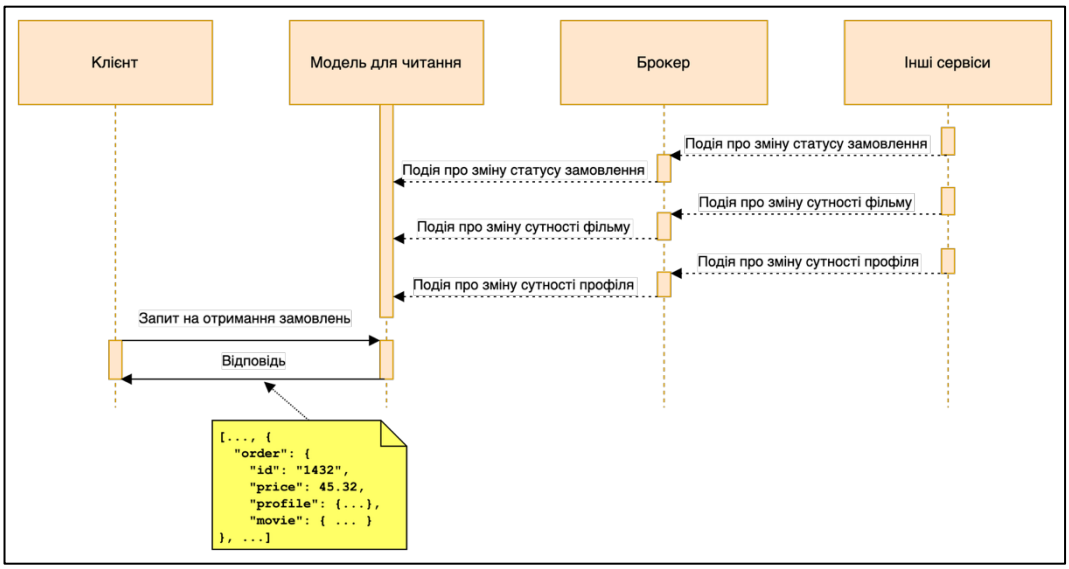

Рисунок 2.5.2.2.1 – Приклад насичення моделі для читання та отримання клієнтом відповіді

2.5.3 Сервіс аудит-логування

Аудит-логування – це підхід до збирання, групування, каталогізації та обробки подій, що стаються у застосунку з метою їх подальшого збереження у форматі, що підходить для подальшого аналізу аудиторами [17].

Так як у застосунку використовуються події для інформування щодо зміни станів агрегатів, то за такого випадку можна додати сторонній сервіс, що не бере участь у реалізації бізнес-логіки, але йому потрібно отримувати інформацію про події, що відбуваються у застосунку. Такий підхід показує можливість додавання нових сервісів різної структури, що мають на меті виконання задач різних рівнів у такий спосіб, що інші сервіси не мають знати про те, що інший сервіс читає певну інформацію з подій. Це є одною з головних переваг у мікросервісній архітектурі, що реалізується за допомогою подієво-орієнтованого підходу та напряму підтримується CQRS [8].

У такий спосіб реалізовано аналогічний сервіс, що має у собі лише модель для читання та не передбачає створення сервісу для запису даних, тому що аудит-логування не передбачає виконання жодної ролі щодо виконання бізнес-логіки у застосунку. Звідси, такий сервіс має у собі обробники подій, які отримують інформацію щодо всіх подій у застосунку. Приклад обробника наведено на лістингу Б.10 додатку Б. Аналогічно, задля отримання потрібних логів клієнтом також використовується MVC-контролер, що є аналогічним до попередніх, його приклад показано у лістингу Б.11 додатку Б.

У такому випадку розкривається основна перевага можливостей мікросервісної архітектури: під час масштабування проєкту у рамках розширення кількості бізнес-логіки, той самий сервіс може незалежно додати власний обробник подій для кожного їх типу без потреби інші сервіси дізнаватися про те, що даний сервіс хоче брати до уваги більшу кількість подій. Наприкінці успішно отримується сервіс пласкої структури, де складність його розширення є горизонтальною, тому що всі обробники подій знаходяться на одному рівні у сервісі. Звідси, CQRS надає можливості та перспективи для безболісного розширення сервісів та їх можливостей зі збереженням простоти підтримки та масштабування проєкту, особливо коли над проєктом працює велика кількість команд розробників.

3 ТЕСТУВАННЯ СЕРВІСІВ

Після написання сервісів для даного застосунку постало питання їх тестування, особливо у контексті шаблону CQRS із імплементацією обміну повідомленнями через брокер подій. Подальше тестування передбачається у двох форматах: за допомогою ручного тестування та автоматизованого тестування відповідно.

Основна перевага тестування – узгодження очікувань та фактичних результатів виконання операцій у сервісі з метою запевнення виконання функціональних та нефункціональних вимог проєкту. У рамках даного розділу передбачається написання тестів саме для перевірки функціональних вимог, а саме для перевірки коректності функціонування сервісів у сенсі реалізації бізнес-логіки.

3.1 Автоматизоване тестування за допомогою TDD

Test-Driven Development (TDD) – це методологія написання тестів для застосунку з метою упередження можливих помилок ще до початку розробки через написання тестів, що напочатку не будуть проходити. Надалі буде написано бізнес-логіку застосунку, що підлаштовуватиметься під написані тести, після чого буде зроблено загальний рефакторинг для приведення його до принципів чистого коду та дотримання найкращих практик написання сервісів [18]. У загальному випадку етапи TDD представлені на рисунку 3.1.1.

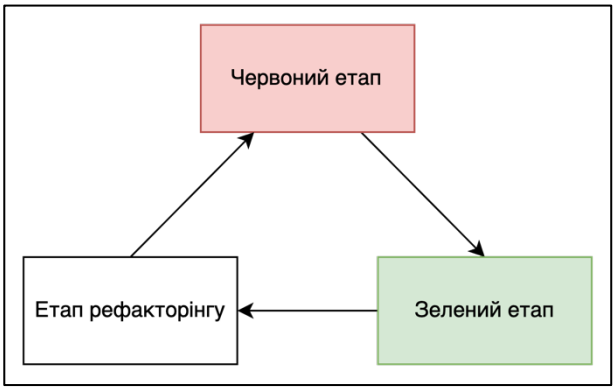

Рисунок 3.1.1 – Етапи TDD

Під час написання автоматизованих тестів застосовано принцип чорного ящика («black box testing»), коли тестувальник не знає як працює система зсередини [18]. Такий підхід підходить саме для TDD, тому що сервіс ще не написано на етапі створення тестів, тому за такої комбінації TDD надає такі переваги:

- написання набору тестів, що покриватимуть якомога більше варіантів використання застосунку;
- обов’язкова підтримка високого рівня покриття тестами, що збільшує загальну надійність застосунку, бо тут з’являється можливість подальшого регресійного тестування [18].

3.1.1 Асинхронна взаємодія

У випадку мікросервісів маємо підвищену складність тестування, тому що за подієво-орієнтованого спілкування потрібно використовувати інакший спосіб тестування, аніж для класичного веб-застосунку на HTTP [19], а також через те, що наразі сервіси реалізовано через CQRS, отримано більшу кількість сервісів, які потрібно протестувати.

Під час написання тестів для асинхронної взаємодії взято до уваги обробку повідомлень агрегатами, де за даної команди надається відповідь. Таке тестування варто проводити саме у моделі для запису інформації, тому що саме такий серіс є центральним та єдиним джерелом правди щодо стану агрегатів. Власне ці та наступні тести написано за методологією Behavior-Driven Development (BDD), де є три основні етапи оформлення тесту [18]:

- етап Given («дано») – містить надану інформацію, що буде використано для аналізу поведінки відного таких даних;
- етап When («коли») – визначає подію, що буде виконуватися за наданих умов;
- етап Then («тоді») – означає результат виконання певної операції, який буде проходити подальшу валідацію на відповідність очікуванням.

У такий спосіб тести можна краще зрозуміти, а тестувальник отримує можливість дізнатися принцип роботи інших програмних модулів лише подивившись на структуру або назву такого тесту. Також для тестів обрано спеціальні набори бібліотек, які допоможуть у зручний та надійний спосіб виконати написання автоматизованих тестів. У даний набір входять інструменти JUnit та Axon Testing API. Приклад написання тесту, що перевіряє асинхронну взаємодію на прикладі сервісу замовлень, наведено на лістингу В.1 додатку В.

В аналогічний спосіб написано тести для обробників подій моделей для читання. Основна відмінність полягає у зміні акценту зі зміни станів агрегату на стан репозиторію сутностей.

3.1.2 Синхронна взаємодія

Так як всі сервіси використовують протокол HTTP для синхронного обміну інформацією з клієнтами, у такому випадку використано фреймворк MockMvc, що також за методологією BDD надає можливість написання підтримуваних та зрозумілих тестів. Він також надає можливість запуску тестів на рівні застосунку без потреби запуску окремого веб-серверу, що підвищує ізольованість тестів та спрощує їх написання.

На кожний метод контролеру зазвичай достатньо написати як мінімум два набори тестів: в одному наборі є тести, що успішно проходять; в іншому – тести, що видають помилки з метою перевірки коректності опрацювання валідації наданого запиту.

Приклад набору тестів для перевірки коректності відпрацювання контролеру для отримання колекції даних з сервісу моделі для читання фільмів наведено на лістингу В.2 додатку В.

3.2 Ручне тестування

Після написання автоматизованих тестів, що показують успішність відповідей застосунку на відповідні запити та події, маємо фінальний етап тестування: з боку користувача, вже з повністю запущеними сервісами, що готові до роботи. Задля тестування окремих сервісів написано шаблони HTTP-запитів, що запускаються через Integrated Development Environment (IDE), що має такі можливості для виконання запитів. Запити виконуватимуться саме у такий спосіб, тому що цей інструмент надає простий інтерфейс щодо виконання запитів, де не потрібно використовувати інші зовнішні клієнти, що потрібно налаштовувати. Додатково, так як всі операції, розробка та тестування, у такому випадку виконуються у межах однієї IDE, там також можна створити запити з методів контролерів автоматично, що також економить час розробки. Звідси, маємо приклад тестування моделі для читання історії запитів.

Нехай у застосунку створено користувача, додано фільми, оновлено баланс користувача, тоді при виконанні запиту до моделі для читання виконується такий запит на лістингу 3.2.1. Важливо зазначити, що за замовчуванням мікросервіс використовує випадковий порт, тому за такого варіанту складно його тестувати, тому що кожного разу порт змінюється. Такі сервіси

Лістинг 3.2.1 – Запит для отримання історії замовлень користувача

```
@profileId = c0357db1-dcdb-485d-9f78-52ac891dac70
GET http://localhost:62500/{{profileId}}
Content-Type: application/json
```

Маємо результат такого запиту на лістингу 3.2.2.

Лістинг 3.2.2 – Відповідь щодо історії замовлень користувача

```
[
  {
    "id": "fd0f1338-be84-400a-81ee-30be58ff248d",
    "status": "PENDING",
    "movie": {
      "id": "d676b092-0cfa-48a7-b9a3-60fedc5f18d0",
      "title": "La La Land",
      "releaseYear": 2016
    },
    "price": "220.59"
  },
  {
    "id": "0ae9d575-6bb0-4a4c-b1be-93def31a99d3",
    "status": "COMPLETED",
    "movie": {
      "id": "d676b092-0cfa-48a7-b9a3-60fedc5f18d0",
      "title": "The Breakfast Club",
      "releaseYear": 1985
    },
    "price": "119.99"
  },
  // ...
]
```

В аналогічний спосіб відбувається тестування всіх інших сервсів

ВИСНОВКИ

Під час реалізації застосунку мікросервісної архітектури проаналізовано та розроблено оптимальний спосіб впровадження шаблону CQRS задля забезпечення підвищення доступності сервісів, що реалізують такий підхід, а також спрощення їх підтримки командою розробників [8]. Додатково, за допомогою імплементації брокеру повідомлень Kafka та використання подієво-орієнтованої підходу розробки отримано можливість реалізації такого шаблону розробки із можливістю незалежного масштабування сервісів за кількістю бізнес-логіки на сервіс. У такий спосіб кожен з сервісів підписується на потрібні йому події, де наприкінці отримуємо низьке счеплення між сервісами через те, що сервісам не потрібно знати навіть про існування одне одного: всі міжсервісні комунікації відбуваються лише за допомогою обміну повідомленнями: командами та подіями відповідно [5][8][9].

У ході роботи визначено, що оптимальною архітектурою мікросервісу є його представлення у вигляді шестикутної архітектури, яка має у собі вхідні та вихідні точки комунікації з іншими сервісами, що підвищує розуміння логіку роботи сервісу та знижує затрати на його підтримку через прямолінійність структури такого сервісу. Підвищення доступності сервісів вдалося отримати із впровадженням CQRS через розділення принципів відповідальності: якщо сервіс з моделлю для запису не є доступним, окремий сервіс для читання даних продовжує працювати незалежно. Також моделі для читання можуть обирати власне сховище даних, тому що кожен сервіс має право на власну базу даних [5][8]. Наприклад, у більшості сервісів для зберігання даних обрано Redis, що також підвищує швидкість отримання даних. З іншого боку, у нього є недоліки у вигляді неможливості впровадження складної структури моделі даних, але як приклад у даній роботі він спрацьовує коректно. У реальних проєктах Redis також часто використовується як сховище для кешів, де головним пріоритетом є швидкість отримання даних, що попередньо денормалізовано [8].

Розроблений застосунок успішно протестовано за двома підходами до тестування: автоматизованого та ручного. У варіанті автоматизованого тестування написано програмні тест-кейси для асинхронної та синхронної взаємодії сервісів з клієнтом. У загальному випадку, підходи тестування мікросервісних застосунків є такими самими, якщо розглядати рівень юніт- та інтеграційних тестів. Тут так само можна використовувати як вже відомі фреймворки, такі як MockMVC, так і популярні підходи до тестування у вигляді TDD та BDD [18].

У подальшому, розроблений застосунок можна масштабувати у двох векторах розвитку. По-перше, хоча початкова розробка сервісу була дещо складнішою, ніж у монолітному застосунку, особливо з провадженням CQRS та подієво-орієнтованого обміну повідомленнями, однак надалі можна без труднощів додавати нову бізнес-логіку, нові сервіси, де складність їх додавання вже буде не такою складною, а в деяких випадках і простішою, ніж у моноліті. Це можливо через те, що більшість сервісів є невеликого розміру за кількістю в них логіки, тому їх зручно, швидко та просто розробляти та підтримувати у подальшому, особливо з можливістю написання тестів. По-друге, написаний застосунок можна розгорнути у хмарі, що надасть можливість масштабування сервісів за віссю X кубу масштабування сервісів. Отримані сервіси є класичними застосунками, що легко розгортати у різних середовищах. Хмарна архітектура надає переваги розміщення застосунку у різних зонах доступності по всьому світу, що підвищує загальну відмовостійкість та доступність сервісів [1][8].

Щодо інфраструктурного рівня, можливо у подальшому реалізувати шаблони API Gateway (API-шлюз), що надасть можливість звертатися до всіх сервісів застосунку клієнтом через єдину точку входу, що спростить для клієнта спосіб доступу до застосунку. Додатково, у такому вигляді можна реалізувати спілкування між шлюзом та сервісами через протокол gRPC замість HTTP, який хоча складніше налаштувати, але він підвищить швидкість передачі відповіді, що проходить через бізнес-сервіс до шлюзу [8].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Microservices Pattern Language // Microservices.io. URL: https://microservices.io/patterns/microservices.html. Дата звернення: 23.05.2025.
2. CQRS Pattern // Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs.Дата звернення: 23.05.2025.
3. Scaling Event Sourcing for Netflix Downloads, Episode 2 // Netflix Tech Blog. URL: https://netflixtechblog.com/scaling-event-sourcing-for-netflix-downloads-episode-2-ce1b54d46eec. Дата звернення: 23.05.2025.
4. Event Sourcing Pattern // Microservices.io. URL: https://microservices.io/patterns/data/event-sourcing.html. Дата звернення: 23.05.2025.
5. Abbott M. Fisher M. Art of Scalability, The: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Crawfordswille : Addison-Wesley, 2015, 624 с.
6. Fowler M. Patterns of Enterprise Application Architecture. Upper Saddle River: Pearson Education, 2003, 560 с.
7. Cockburn A, Juan M. de Paz. Hexagonal Architecture Explained. Salt Lake City : Humans and Technology Inc, 2024, 202 с.
8. Richardson C. Microservices Patterns. Shelter Island : Manning Publications Co., 2019, 522 с.
9. Zelenin A., Kropp A. Apache Kafka in Action: From basics to production ^ Manning, 2025, 368 с.
10. Chapter 12. Full Text Search // PostgreSQL Documentation. URL: https://www.postgresql.org/docs/17/textsearch.html. Дата звернення: 23.05.2025.
11. Elasticsearch features // Elastic. URL: https://www.elastic.co/elasticsearch/features. Дата звернення: 23.05.2025.

12. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software : Addison-Wesley, 2003, 560 с.
13. Brandolini A. Introducing EventStorming. Faenza : Avanscoperta, 2015, 345 с.
14. AxonIQ Docs // AxonIQ. URL: https://docs.axoniq.io/home/. Дата звернення: 23.05.2025.
15. Spring Data Redis // Spring Docs. URL: https://spring.io/projects/spring-data-redis. Дата звернення: 23.05.2025.
16. Spring Web // Spring Docs. URL: https://docs.spring.io/spring-boot/reference/web/index.html. Дата звернення: 23.05.2025.
17. National Information Assurance (IA) Glossary, USA. URL: https://web.archive.org/web/20120227163121/http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf. Дата звернення: 23.05.2025.
18. TDD Made Easy: An introduction to Test-Driven Development // Symflower. URL: https://symflower.com/en/company/blog/2023/how-to-get-started-with-tdd/downloads/Symflower_Whitepaper_TDD-Made-Easy.pdf. Дата звернення: 23.05.2025.
19. Fielding R. Architectural Styles and the Design of Network-based Software Architectures, дис. PhD, Irvine, 2000, 180 с. URL: https://web.archive.org/web/20120124022936/https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Дата звернення: 23.05.2025.

ДОДАТОК А. ЕТАПИ EVENT STORMING

Рисунок А.1 – Перший етап Event Storming, де розміщено агрегати та команди

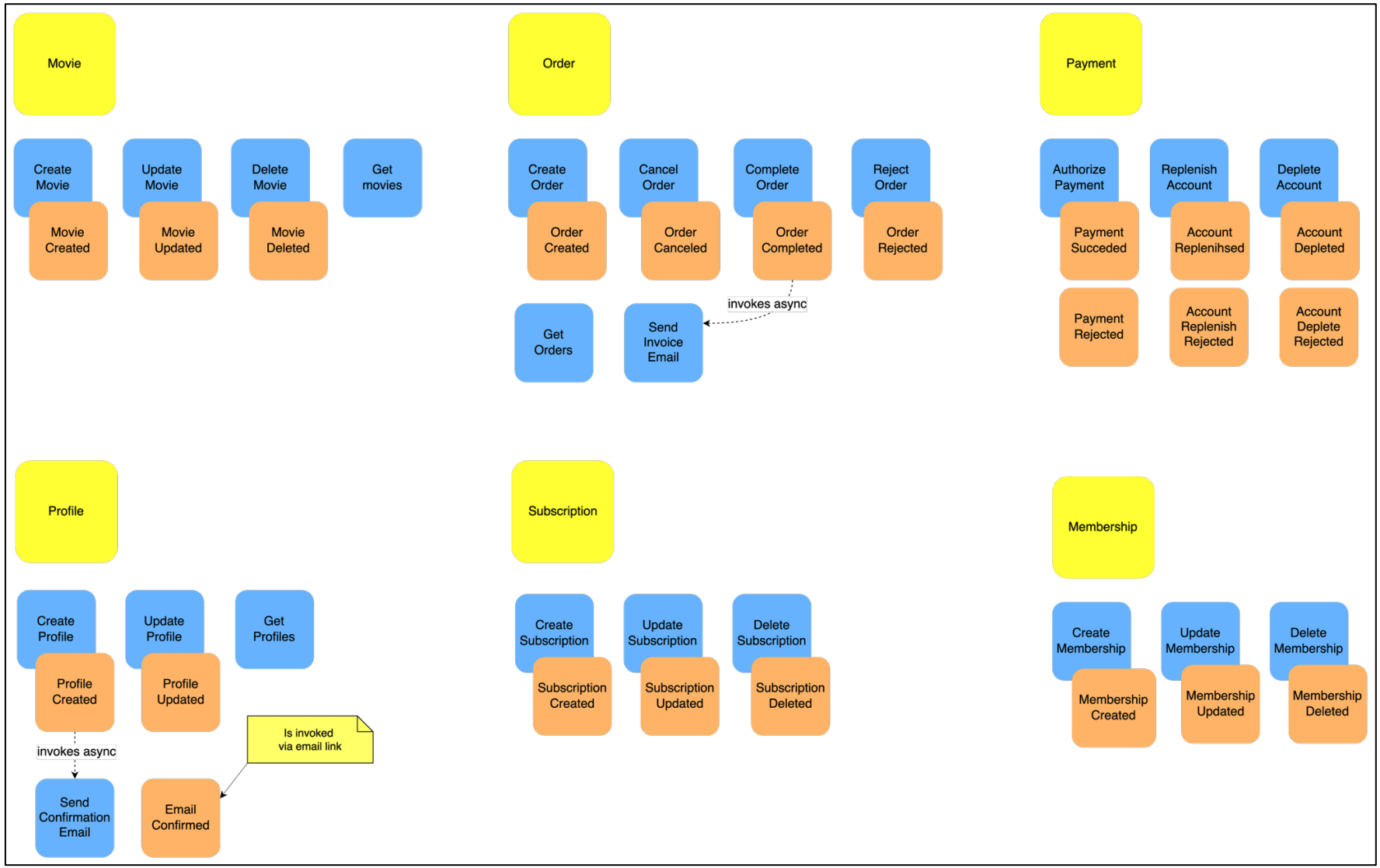

Рисунок А.2 – Другий етап Event Storming, де додано події

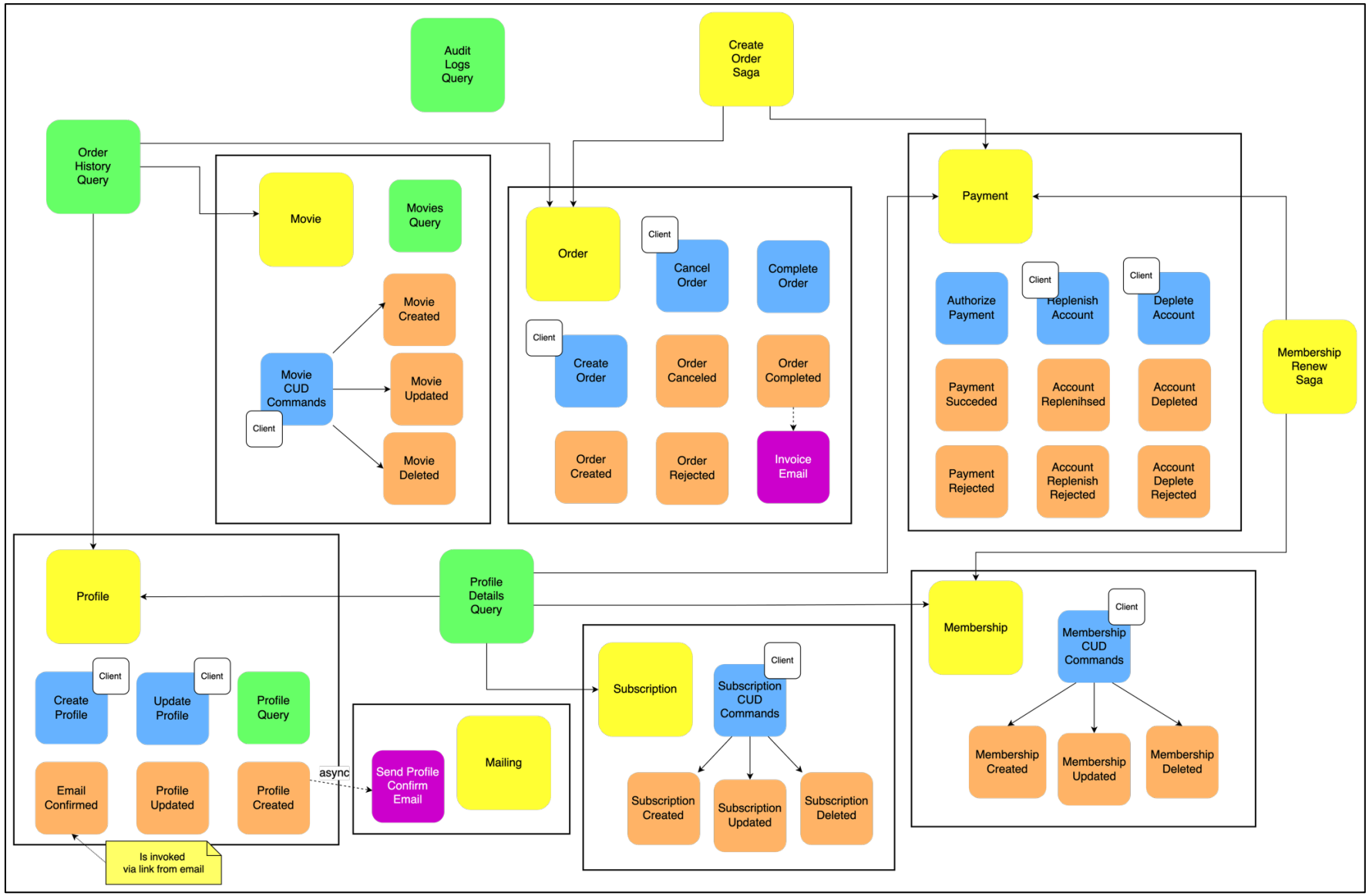

Рисунок А.3 – Фінальний етап Event Storming, де додано CQRS, саги та окремий сервіс для пошти

ДОДАТОК Б. ПРИКЛАДИ ВИХІДНОГО КОДУ

Лістинг Б.1 – Проєкт Docker Compose для розгортання Axon

```
services:
  axonserver:
    image: 'axoniq/axonserver:2024.2.4-jdk-17'
    environment:
      - 'AXONIQ_AXONSERVER_STANDALONE=TRUE'
    volumes:
      - "./axon/config:/axonserver/config"
      - "./axon/data:/axonserver/data"
      - "./axon/events:/axonserver/events"
      - "./axon/exts:/axonserver/exts"
      - "./axon/log:/axonserver/log"
      - "./axon/plugins:/axonserver/plugins"
    ports:
      - '8024:8024'
      - '8124:8124'
```

Лістинг Б.2 – Вихідний код залежностей сервісу фільмів моделі запису даних

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>ua.karazin</groupId>
        <artifactId>movies-base-dependencies</artifactId>
        <version>1.0-SNAPSHOT</version>
        <relativePath/>
    </parent>

    <groupId>ua.karazin</groupId>
    <artifactId>movies-content-service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>movies-content-service</name>
    <description>movies-content-service</description>

    <properties>
        <java.version>21</java.version>
    </properties>
```

```
    <dependencies>
        <dependency>
            <groupId>ua.karazin</groupId>
            <artifactId>movies-base-events</artifactId>
            <version>1.3</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-
jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-
validation</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-docker-compose</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Лістинг Б.3 – Проєкт Docker Compose для розгортання Redis

```
services:
  redis:
    image: redis:7.4.2
    ports:
      - "9920:6379"
    environment:
      - REDIS_PASSWORD=pass
      - REDIS_USER=user
      - REDIS_USER_PASSWORD=pass
```

Лістинг Б.4 – Модель Order

```
@Aggregate
public class Order {
    @AggregateIdentifier
    private String orderId;
    @AggregateMember
    private Movie movie;
    private String profileId;
    private OrderStatus1 status;

    @CommandHandler
    private Order(CreateOrderCommand command) {
        apply(new OrderCreatedEvent(command.orderId(),
command.orderDto()));
    }

    @CommandHandler
    private void handle(CompleteOrderCommand command) {
        apply(new OrderCompletedEvent(command.orderId(),
command.finalPrice()));
    }

    @EventSourcingHandler
    private void on(OrderCreatedEvent event) {
        this.orderId = event.orderId();
        this.movie = new Movie();
        this.profileId = event.orderDto().profileId();
        this.status = OrderStatus1.PENDING;
    }

    @EventSourcingHandler
    private void on(OrderCompletedEvent event) {
        throwIfIsSettled();
        this.status = OrderStatus1.COMPLETED;
    }

    @EventSourcingHandler
    private void on(OrderRejectedByPaymentEvent event) {
        throwIfIsSettled();
        this.status = OrderStatus1.REJECTED_BY_PAYMENT;
    }
}
```

Лістинг Б.5 – Проєкція фільму для агрегату замовлення

```
public class Movie {
    @EntityId
    private String movieId;
    private String title;
    private BigDecimal price;
```

```
    @EventSourcingHandler
    private void on(OrderCreatedEvent event) {
        var dto = event.orderDto();
        this.movieId = dto.movieId();
        this.title = event.orderDto().title();
        this.price = event.orderDto().price();
    }
}
```

Лістинг Б.6 – Фрагмент контролеру для моделі запису замовлень

```
@RestController
public class OrderController {
  private final OrderService service;

  @PostMapping
  public CompletableFuture<Order> createOrder(@Valid
@RequestBody OrderCreateRequest request) {
    return service.createOrderFrom(request);
  }
}
```

Лістинг Б.7 – Обробник подій замовлень для насичення проєкції фільмів

```
@Component
@RequiredArgsConstructor
public class MovieProjectionHandler {
  private final MovieService service;

  @EventHandler
  private void on(MovieCreatedEvent event) {
    service.saveFrom(event);
  }

  @EventHandler
  private void on(MovieUpdatedEvent event) {
    service.updateFrom(event);
  }

  @EventHandler
  private void on(MovieDeletedEvent event) {
    service.deleteById(event.movieId());
  }
}
```

Лістинг Б.8 – Обробник подій замовлень для насичення проєкції профілей

```
@Component
@RequiredArgsConstructor
public class ProfileProjectionHandler {
  private final ProfileService service;

  @EventHandler
  private void on(ProfileCreatedEvent event) {
    service.saveFrom(event);
  }

  @EventHandler
  private void on(ProfileUpdatedEvent event) {
    service.updateFrom(event);
  }
}
```

Лістинг Б.9 – Обробник подій замовлень для насичення проєкції замовлень

```
@Component
@RequiredArgsConstructor
public class OrderProjectionHandler {
  private final OrderService service;

  @EventHandler
  private void on(OrderCreatedEvent event) {
    service.saveFrom(event);
  }

  @EventHandler
  private void on(OrderRejectedByPaymentEvent event) {
    service.rejectById(event.orderId());
  }

  @EventHandler
  private void on(OrderCompletedEvent event) {
    service.completeById(event.orderId(), event.finalPrice());
  }
}
```

Лістинг Б.10 – Фрагмент обробника подій для аудит-логування

```
@Component
public class LogMembershipHandler {
  private final LogService service;

  @EventHandler
  private void on(ProfileCreatedEvent event, Instant instant) {
    service.save(Log.builder()
        .action("Profile create")
        .profileId(event.profileId())
        .level(LogLevel.INFO)
        .createdAt(instant)
        .build());
  }

  @EventHandler
  private void on(MemRejectedEven event, Instant instant) {
    service.save(Log.builder()
        .action("Membership rejected")
        .profileId(event.profileId())
        .level(LogLevel.WARNING)
        .createdAt(instant)
        .build());
  }
}
```

Лістинг Б.11 – Фрагмент контролеру для отримання логів клієнтом

```
@RestController
@RequiredArgsConstructor
public class LogController {
  private final LogService service;

  @GetMapping("/{id}")
  public Iterable<Log> getLogsForProfile(@PathVariable
String id) {
    return logService.findByProfileId(id);
  }
}
```

ДОДАТОК В. ПРИКЛАД ПРОГРАМНИХ ТЕСТІВ

Лістинг В.1 – Приклад тестування асинхронної взаємодії сервісу замовлень

```
class OrderTest {
  private final AggregateTestFixture<Order> fixture =
      new AggregateTestFixture<>(Order.class);

  @Test
  void whenCreateOrderCommand_thenOrderIsCreated() {
    var orderDto = OrderDto1.builder()
        .orderId(UUID.randomUUID())
        .profileId(UUID.randomUUID())
        .movieId(UUID.randomUUID())
        .movieName("Movie name")
        .moviePrice(new BigDecimal("120.50"))
        .build();

    fixture.givenNoPriorActivity()
        .when(new CreateOrderCommand(orderDto.getOrderId(),
orderDto))
        .expectEvents(new
OrderCreatedEvent(orderDto.getOrderId(), orderDto));
  }

  @Test
  void
givenPendingOrder_whenCompleteOrderCommand_thenOrderIsCreated()
{
    var orderDto = OrderDto1.builder()
        .orderId(UUID.randomUUID())
        .profileId(UUID.randomUUID())
        .movieId(UUID.randomUUID())
        .movieName("Movie name")
        .moviePrice(new BigDecimal("120.50"))
        .build();

    fixture.given(new OrderCreatedEvent(orderDto.getOrderId(),
orderDto))
        .when(new CompleteOrderCommand(orderDto.getOrderId()))
        .expectEvents(new
OrderCompletedEvent(orderDto.getOrderId()));
  }

  @Test
  void
givenCompletedOrder_whenRejectOrderCommand_thenExceptionIsThrown
() {
    var orderDto = OrderDto1.builder()
        .orderId(UUID.randomUUID())
        .profileId(UUID.randomUUID())
        .movieId(UUID.randomUUID())
```

```
        .movieName("Movie name")
        .moviePrice(new BigDecimal("120.50"))
        .build();

    fixture.given(new OrderCreatedEvent(orderDto.getOrderId(),
orderDto))
        .andGiven(new
OrderCompletedEvent(orderDto.getOrderId()))
        .when(new
RejectOrderByPaymentCommand(orderDto.getOrderId()))
        .expectException(IllegalStateException.class);
  }
}
```

Лістинг В.2 – Приклад тестування MVC-контролеру сервісу історії замовлень

```
@WebMvcTest
@RequiredArgsConstructor
class OrderControllerTest {
  private static final String PROFILE_ID = "123";
  private static final Order ORDER =
TestDataFactory.buildOrder(PROFILE);

  @Autowired
  private MockMvc mockMvc;

  @MockitoBean
  private OrderService orderService;

  @BeforeEach
  void setUp() {
    var orderHistory = List.of(ORDER);
    Mockito.when(orderService.findSettledOrdersFor(PROFILE_ID)
       .thenReturn(orderHistory);
  }

  @Test
  void givenInvalidId_whenGetOrders_thenThrowAnException()
throws Exception {
    mockMvc.perform(get("/12345"))
        .andExpect(status().is4xxClientError());
  }

  @Test
  void givenValidId_whenGetOrders_thenReturnOrders() throws
Exception {
    mockMvc.perform(get("/" + PROFILE_ID))
        .andExpect(content().json(new
ObjectMapper().writeValueAsString(List.of(ORDER))))
        .andExpect(status().is2xxSuccessful());
  }
}
```