

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій**

«На правах рукопису»
УДК 004

До захисту допущено:
Завідувач кафедри
 Олександр РОЛІК
« » 2023 р.

**Магістерська дисертація
на здобуття ступеня магістра
за освітньо-професійною програмою «Інформаційні управляючі системи та
технології»
зі спеціальності 126 «Інформаційні системи та технології»
на тему: «Система керування повідомленнями при асинхронній комунікації
мікросервісів»**

Виконав:
студент 2 курсу, групи ІС-23мп
Луценко Олександр Андрійович

Керівник:
ст. викладач, к.т.н
Галушко Дмитро Олександрович

Рецензент:
доцент, к.т.н
Волокита Артем Миколайович

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

Студент _____

Київ – 2024 року

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма

«Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр РОЛІК

«___» _____ 2023 р.

**ЗАВДАННЯ
на магістерську дисертацію студенту
Луценко Олександр Андрійович**

1. Тема дисертації «Система керування повідомленнями при асинхронній комунікації мікросервісів», науковий керівник дисертації Галушко Дмитро Олександрович, к.т.н, затверджені наказом по університету від «07» 11 2023 р. № 5168-с
2. Термін подання студентом дисертації «12» 01 2024 р.
3. Об'єкт дослідження: асинхронна комунікація мікросервісів та потреби користувачів цих систем.
4. Вихідні дані: технічне завдання
5. Перелік завдань, які потрібно розробити: огляд та аналіз сфери застосувань та існуючих рішень, формулювання вимог до системи, вибір технологій розробки системи, розробка системи, публікація системи, тестування системи, розробка стартап-проекту
6. Орієнтовний перелік графічного (ілюстративного) матеріалу: діаграма варіантів використання, ER-діаграма, діаграма класів, діаграма публікації системи, діаграма діяльності, схема структури системи, діаграма компонентів, діаграма послідовностей.

7. Дата видачі завдання 01.09.2023 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Огляд та аналіз сфери застосувань	14.09.2023	
2	Огляд та аналіз існуючих рішень	20.09.2023	
3	Формулювання вимог до системи	27.09.2023	
4	Вибір інструментів реалізації системи	04.10.2023	
5	Проектування системи	16.10.2023	
6	Розробка системи	11.11.2023	
7	Публікація та тестування системи	29.11.2023	
8	Розробка стартап-проекту	10.12.2023	
9	Оформлення текстової документації	18.12.2023	

Студент

Олександр ЛУЦЕНКО

Науковий керівник

Дмитро ГАЛУШКО

РЕФЕРАТ

Луценко О.А. Система керування повідомленнями при асинхронній комунікації мікросервісів. КПІ ім. І. Сікорського, Київ, 2023.

Дисертація містить 125 с. тексту, 59 рисунків, 24 таблиці, 30 джерел, 9 додатків.

Асинхронна комунікація мікросервісів є ключовим механізмом в роботі мікросервісів, але при використанні цієї комунікації виникає багато проблем з повідомленнями. Тема дисертації є актуальною, адже вона допомагає вирішити частину цих проблем.

Метою магістерської дисертації є підвищення ефективності обробки повідомлень асинхронної комунікації мікросервісів. Задачі магістерської дисертації включають в себе є аналіз існуючих рішень, проектування, розробка, тестування та інтеграція системи.

Об'єктом дослідження є асинхронна комунікація мікросервісів та потреби користувачів цих систем.

Предметом дослідження є просте в налаштуванні та використанні збереження повідомлень асинхронної комунікації мікросервісів від втрат та моніторинг цих повідомлень для подальшого повторного відправлення в залежності від їх стану та вимог користувача.

Програмний продукт був написаний мовою програмування C# на платформі .NET7 за архітектурою Clean Architecture. Для зберігання даних була використана СУБД MySQL, для роботи з даними був використаний фреймворк Entity Framework Core. Результати отримані при розробці системи були використані для розробки стартап-проекту.

Ключові слова: МІКРОСЕРВІСИ, C#, .NET7, MYSQL, АСИНХРОННА КОМУНІКАЦІЯ, RABITMQ

ABSTRACT

Lutsenko O.A. Message management system for microservices asynchronous communication. Igor Sikorskyi, KPI, Kyiv, 2023.

The dissertation contains 125 pages, 59 figures, 24 tables, 30 sources, 9 appendices.

Asynchronous communication of microservices is a key mechanism in the operation of microservices, but there are many problems with messages when using this communication. The topic of the dissertation is relevant, because it helps to solve some of these problems.

The goal of the master's thesis is to improve the efficiency of processing messages of asynchronous communication of microservices. The tasks of the master's thesis include the analysis of existing solutions, design, development, testing and integration of the system.

The object of research is asynchronous communication of microservices and the needs of users of these systems.

The subject of the research is to easily configure and save microservices asynchronous communication messages from loss and monitor these messages for further resending depending on their status and user requirements.

The software product was written in the C# programming language on the .NET7 platform based on the Clean Architecture. The MySQL DBMS was used for data storage, and the Entity Framework Core framework was used for working with data. The results obtained during the development of the system were used for the development of a startup project.

Keywords: MICROSERVICES, C#, .NET7, MYSQL, ASYNCHRONOUS COMMUNICATION, RABITTMQ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ	8
ВСТУП	9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	11
1.1 Мікросервісна архітектура	11
1.2 Асинхронна комунікація між мікросервісами	12
1.3 Проблеми асинхронної комунікації	14
Висновок до розділу 1	15
2 ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ	17
2.1 Datadog	17
2.2 RabbitMQ Sensor	18
2.3 Confluent Control Center	19
2.4 Hangfire	21
2.5 Elastic Stack (ELK)	22
2.6 Prometheus та Grafana	23
2.7 Аналіз існуючих рішень	25
Висновок до розділу 2	26
3 ФОРМУВАННЯ ВИМОГ СИСТЕМИ	27
3.1 Опис вимог до системи	27
3.2 Діаграма варіантів використання	28
Висновок до розділу 3	29
4 ВИБІР ТЕХНОЛОГІЙ РЕАЛІЗАЦІЇ СИСТЕМИ	30
4.1 Вибір технологій	30
4.2 Бази даних	32
4.3 Брокер повідомлень	34
Висновок до розділу 4	37
5 РОЗРОБЛЕННЯ СИСТЕМИ	39
5.1 Основні сутності	39
5.2 Інтеграція з базою даних	42
5.3 Логіка зберігання повідомлень	43
5.4 Механізм повторного відправлення повідомлень	48
5.5 Реєстрація компонентів системи	53
Висновок до розділу 5	55
6 ПУБЛІКАЦІЯ СИСТЕМИ	57
Висновок до розділу 6	59
7 ІНТЕГРАЦІЯ СИСТЕМИ КЕРУВАННЯ ПОВІДОМЛЕННЯМИ	60
7.1 Domain рівень	62

7.2 Infrastructure рівень	62
7.3 API рівень	69
7.4 Message consumer	74
7.5 Механізм повторного відправлення повідомлень	79
Висновок до розділу 7	85
8 ТЕСТУВАННЯ СИСТЕМИ	87
Висновок до розділу 8	94
9 РОЗРОБКА СТАРТАП ПРОЕКТУ	95
9.1 Опис ідеї проекту	95
9.2 Технологічний аудит ідеї проекту	98
9.3 Аналіз ринкових можливостей запуску стартап-проекту	100
9.4 Розроблення ринкової стратегії проекту	113
9.5 Розроблення маркетингової програми стартап-проекту	116
Висновок до розділу 9	121
ВИСНОВКИ	122
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ	123
Додаток А	126
Додаток Б	126
Додаток В	126
Додаток Г	126
Додаток Ґ	126
Додаток Д	126
Додаток Е	126
Додаток Є	126
Додаток Ж	126

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ

БД – База даних

СУБД – Система управління базами даних

JSON – JavaScript Object Notation

HTTP – Hypertext Transfer Protocol

AMQP – Advanced Message Queuing Protocol

ORM – Object Relational Mapper

API – Application Programming Interface

Pub – Publisher

Sub – Subscriber

LINQ - Language Integrated Query

ВСТУП

У відповідь на виклики сучасного ІТ-світу, архітектура мікросервісів стала широко використовуватися, оскільки вона пропонує значну гнучкість, можливість масштабування та відмінну надійність. Мікросервіси відповідають на виклики, які ставлять перед собою монолітні системи, розбиваючи програмне забезпечення на взаємопов'язані, але незалежні модулі, забезпечуючи таким чином більш ефективну взаємодію та управління.

Асинхронна комунікація між мікросервісами є ключовою для підтримки високої продуктивності та масштабованості, але це також створює проблеми, такі як проблеми з отриманням повідомлень, дублювання або найбільша проблема - втрата повідомлень. Щоб гарантувати надійність, послідовність та доставку повідомлень в таких системах, необхідна ефективна система керування повідомленнями.

Існує багато систем для моніторингу даних, в тому числі асинхронних повідомлень, наприклад такі як Datadog, RabbitMQ Sensor, Confluent Control center тощо.

Але, нажаль, систем для керування повідомленнями в залежності від їх стану майже немає, в більшості використовуються системи, в яких користувачам потрібно самим описати та імплементувати усі необхідні дії, наприклад Hangfire, в якому можливо описати дії для механізму повторюваного відправлення повідомлення.

Ці системи не є простими в використанні або в налаштуванні. Користувачі потребують систему, яка б брала на себе усі складнощі реалізації та роботи механізму для повторюваного відправлення повідомлення, ціль якого забезпечити відправлення повідомлень за умов визначених користувачем, для підвищення шансу їх опрацювання та збереження цих повідомлень від втрат.

За умови, якщо певне повідомлення не було успішно опрацьоване після повторного відправлення, користувач має мати змогу легко знайти та мануально провести певні дії в залежності від стану цього повідомлення.

Саме тому в рамках магістерської дисертації буде розроблений продукт, який вирішує 2 основні проблеми - збереження повідомлень асинхронної комунікації мікросервісів для їх моніторингу та система для повторюваного відправлення повідомлення.

Об'єктом дослідження є асинхронна комунікація мікросервісів та потреби користувачів цих систем.

Предметом дослідження є просте в налаштуванні та використанні збереження повідомлень асинхронної комунікації мікросервісів від втрат та моніторинг цих повідомлень для подальшого повторного відправлення в залежності від їх стану та вимог користувача.

Мета проекту - підвищення ефективності обробки повідомлень асинхронної комунікації мікросервісів.

Практичне значення полягає в тому, щоб надати можливість користувачам зберігати повідомлення асинхронної комунікації мікросервісів та надати можливість повторної відправки цих повідомлень в залежності від їх стану та вимог користувача.

Програмний продукт був написаний мовою програмування C# на платформі .NET7. Для зберігання даних була використана СУБД MySQL, для роботи з даними був використаний фреймворк Entity Framework Core. Магістерська дисертація складається з наступних розділів: вступ, аналіз предметної області, огляд та аналіз існуючих рішень, формування вимог до системи, вибір технологій реалізації системи, розроблення системи, публікація системи, інтеграція системи керування повідомленнями, тестування системи, розробка стартап проекту, висновки, перелік використаних джерел.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Мікросервісна архітектура

Мікросервісна архітектура є сучасним архітектурним підходом до розробки програмного забезпечення, який спрямований на розділення великої і складної системи на менші, автономні компоненти. Ці компоненти, відомі як мікросервіси, можуть бути розроблені, розгорнуті та масштабовані незалежно один від одного.

Рисунок 1.1 відображає приклад мікросервісної архітектури.

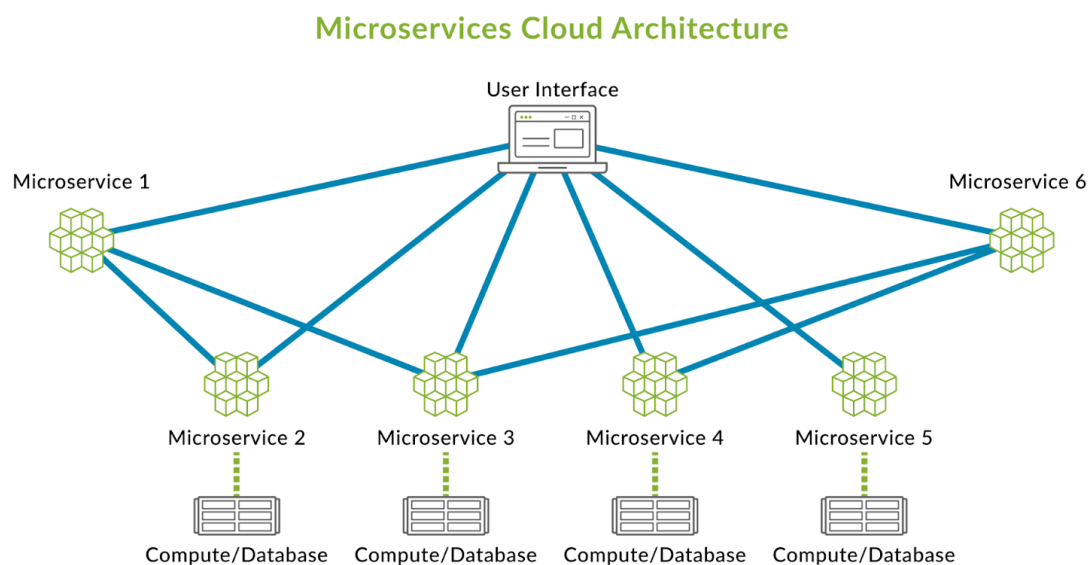


Рисунок 1.1 – Приклад мікросервісної архітектури

Основні принципи мікросервісної архітектури перелічені нижче.

Декомпозиція. Розділення системи на менші компоненти, кожен з яких відповідає за конкретний аспект функціональності.

Автономність. Кожний мікросервіс повинен бути незалежним, мати власну базу даних і не залежати від інших сервісів у своїй роботі.

Границі сервісів. Чітке визначення границь та відповідальності кожного мікросервісу, що дозволяє розробникам зосередитися на конкретних задачах.

Технологічна гетерогенність: З можливістю вибору технології для кожного мікросервісу залежно від конкретних потреб.

Переваги мікросервісної архітектури перелічені нижче.

Масштабованість. Можливість масштабування окремих частин системи залежно від потреб, без необхідності масштабування всієї системи.

Стійкість до відмов. Якщо один сервіс зазнає збою, це не вплине на роботу інших.

Швидкість розробки. Команди можуть працювати паралельно над різними мікросервісами, що прискорює процес розробки.

Легкість оновлення. Окремі мікросервіси можна оновлювати незалежно, без впливу на інші частини системи.

Однак мікросервісна архітектура також приносить власний набір викликів. Це може включати складності в управлінні, моніторингу та безпеці. Також може бути потреба в розробці специфічних інструментів для забезпечення спілкування між сервісами, а також в гарантії якості кожного окремого мікросервісу. Незважаючи на ці виклики, правильно реалізована мікросервісна архітектура може принести значні переваги і допомогти забезпечити швидку, гнучку та надійну систему.

1.2 Асинхронна комунікація між мікросервісами

Асинхронна комунікація стає ключовою у мікросервісних архітектурах, особливо коли необхідна висока доступність, масштабованість та відмовостійкість. У такому підході сервіси обмінюються даними не очікуючи відповіді від одного до іншого у реальному часі.

Рисунок 1.2 відображає різницю синхронної та асинхронної комунікації сервісів.

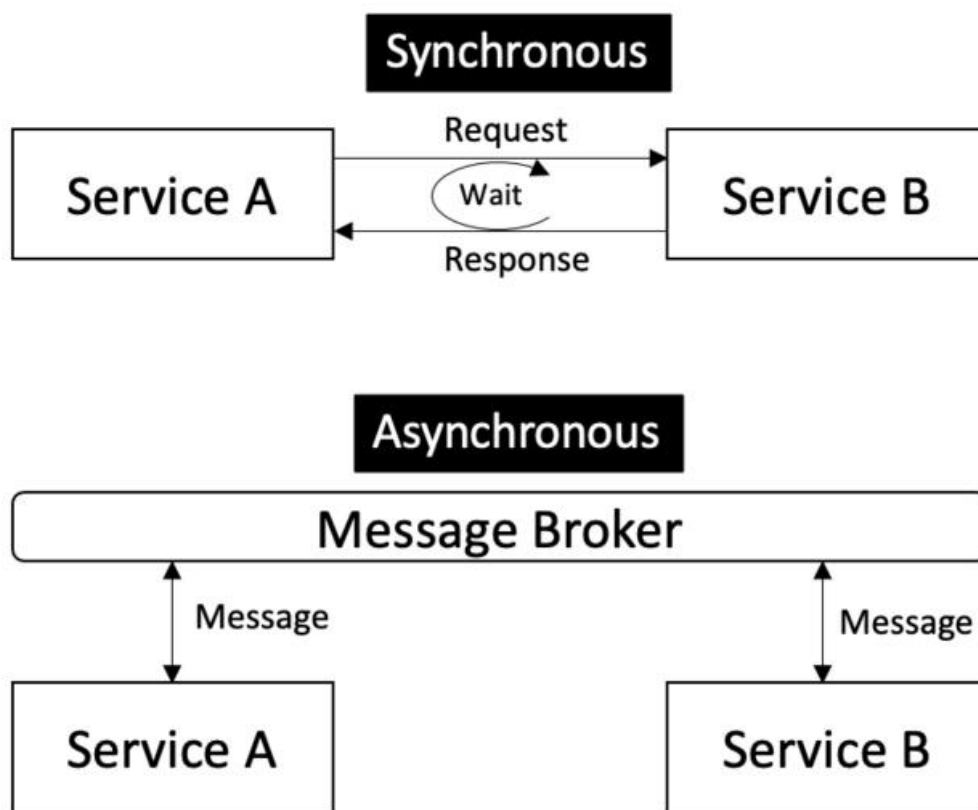


Рисунок 1.2 – Різниця синхронної та асинхронної комунікації сервісів.

Особливості асинхронної комунікації:

- відсутність прямої взаємодії: Сервіси не залежать від відгуку інших сервісів для продовження своєї роботи. Це може підвищити продуктивність і знизити час затримки;
- повідомлення: Комунікація здійснюється за допомогою відправлення та отримання повідомлень, які можуть бути поставлені в чергу та оброблені пізніше;
- буферизація: Завдяки використанню черг повідомлень, асинхронна комунікація може витримувати великі навантаження, шляхом буферизації запитів;

Переваги асинхронної комунікації:

- відмовостійкість: Якщо один сервіс збоїть, інші можуть продовжувати роботу без перебоїв, тому що вони не чекають відповіді в реальному часі;
- масштабованість: Сервіси можуть обробляти запити у різний час, дозволяючи системі ефективно масштабуватися під збільшене навантаження;
- оптимальне використання ресурсів: Завдяки відсутності необхідності в постійних відповідях, ресурси можуть бути використані більш ефективно;

Інструменти та підходи:

- черги повідомлень: Як RabbitMQ, Kafka та інші, які дозволяють організувати надійну асинхронну передачу повідомлень;
- event-driven підхід: Де сервіси реагують на події, що виникають у системі, замість того, щоб надсилати прямі запити;
- callback'и: Метод, що дозволяє отримати відповідь після обробки запиту без блокування поточного процесу;

Асинхронна комунікація між мікросервісами дозволяє створювати системи, які є більш масштабованими, гнучкими та стійкими до різних видів відмов. Однак, як і будь-який підхід, він має свої виклики, які потребують уважного планування та управління.

1.3 Проблеми асинхронної комунікації

Асинхронна комунікація має безліч переваг, але як і будь-який технологічний підхід, вона не є вільною від викликів та проблем.

Основні проблеми, які можуть виникнути:

- втрата повідомлень: Це може статися через ряд причин, включаючи мережеві збої, відмови обладнання, або помилки в самому механізмі передачі повідомлень. Наприклад, якщо система, що відповідає за зберігання повідомлень у черзі, зазнає відмови, повідомлення може бути втрачено;

- дублювання повідомлень: Іноді повідомлення може бути доставлено кілька разів через мережеві збої або збої в системах передачі повідомлень. Це може призвести до непередбачуваних наслідків, якщо система не розроблена так, щоб враховувати такі ситуації;
- проблеми з впорядкуванням: В асинхронних системах дуже важко гарантувати, що повідомлення буде оброблено в правильному порядку, особливо при високому навантаженні;
- складність відслідковування: Через асинхронний характер комунікації може бути важко відслідковувати, які повідомлення були оброблені і коли;
- таймаути: Якщо обробка асинхронного запиту займає занадто багато часу, це може викликати затримки або відмови в інших частинах системи;

Способи рішення:

- підтвердження доставки: Використання механізмів підтвердження доставки для гарантії того, що повідомлення було оброблено;
- ідемпотентність: Проектування системи таким чином, щоб вона могла безпечно обробляти одне й те саме повідомлення кілька разів без негативних наслідків;
- системи відслідковування: Інтеграція засобів відслідковування, які допомагають моніторити і аналізувати потік повідомлень;
- використання шаблонів проектування: Таких як Saga для управління довгими операціями та уникнення проблем з узгодженістю;

На завершення, важливо розуміти, що асинхронна комунікація – це потужний інструмент, але він вимагає уважного проектування та розуміння можливих викликів та рішень для їх подолання.

Висновок до розділу 1

У цьому розділі було проведено ґрунтовний аналіз мікросервісної архітектури, її основних принципів, переваг та викликів. Значна увага приділена

асинхронній комунікації у мікросервісних системах, її ролі у підвищенні стійкості та ефективності роботи.

Детально розглянуто проблеми, які можуть виникати при асинхронній комунікації, включаючи втрату, дублювання повідомлень, складнощі з впорядкуванням та відслідковуванням. Пропонуються конкретні рішення цих проблем, такі як підтвердження доставки, ідемпотентність, використання систем відслідковування, а також застосування шаблонів проектування, що дозволяють оптимізувати асинхронну комунікацію та зробити її більш надійною та ефективною. Важливість цих заходів підкреслюється необхідністю уважного планування та управління для успішної реалізації асинхронних комунікацій у складних мікросервісних системах.

2 ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

2.1 Datadog

DataDog [1] є великою платформою моніторингу, яка надає гнучкі інструменти для відстеження асинхронних повідомлень, логів, метрик серверів чи баз даних та іншого.

Рисунок 2.1 відображає приклад інтерфейсу DataDog.

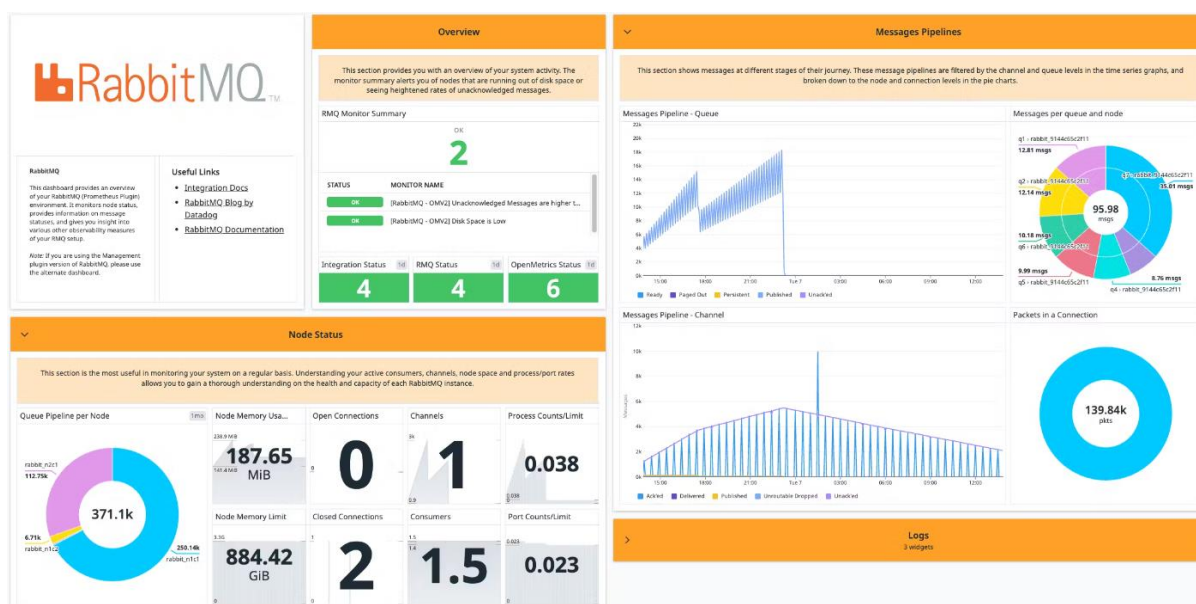


Рисунок 2.1 – Приклад інтерфейсу DataDog

Переваги DataDog:

- широкі можливості інтеграції з іншими платформами та сервісами;
- гнучкість і масштабованість;
- інтуїтивний користувацький інтерфейс;

Недоліки DataDog:

- неспеціалізований на асинхронних повідомленнях, може бути відсутня деяка специфічна функціональність;
- велика вартість за користування;
- використовуються лише для моніторингу повідомлень, не надає можливості керування ними;

- зберігання повідомлень на довготривалі терміни коштує багато;

2.2 RabbitMQ Sensor

RabbitMQ Sensor [2] - це спеціалізований інструмент моніторингу для RabbitMQ, призначений для відстеження активності в чергах.

Рисунок 2.2 відображає приклад інтерфейсу RabbitMQ Management Plugin.

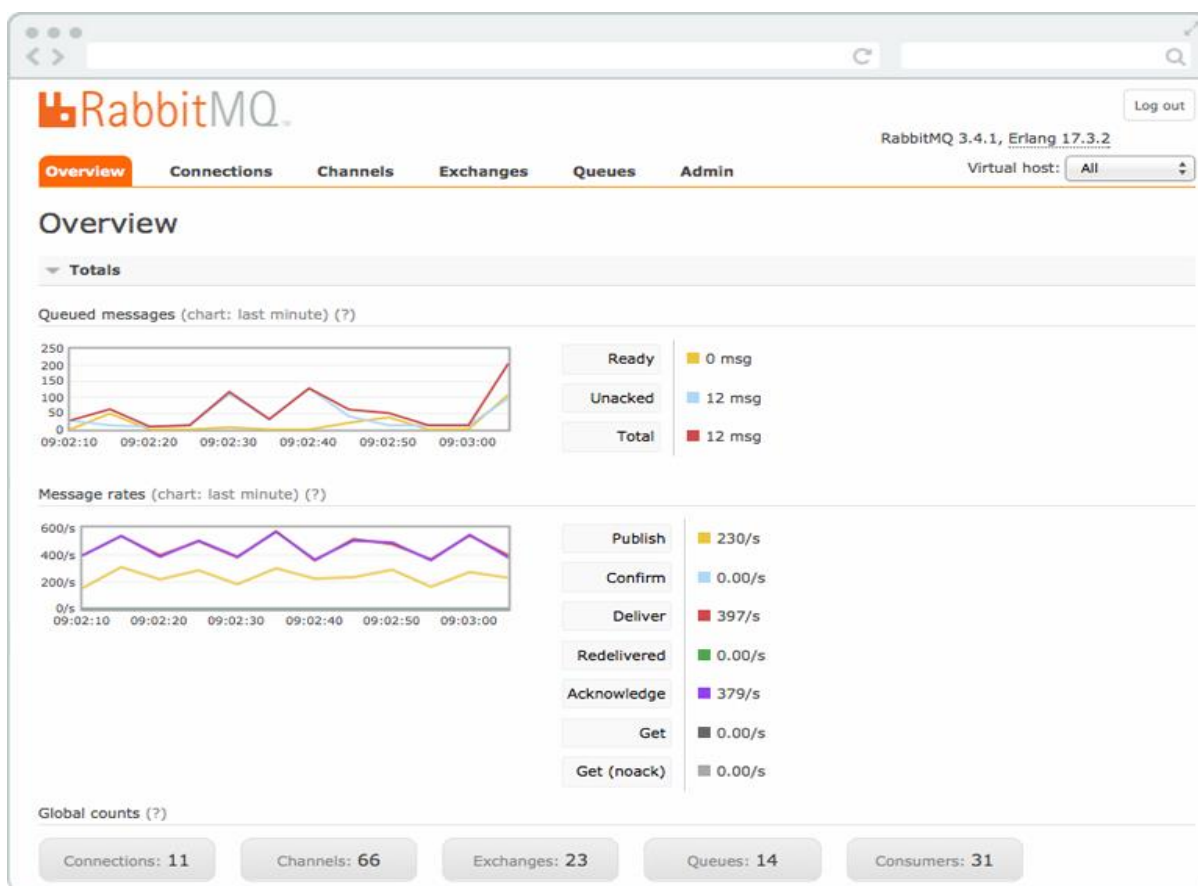


Рисунок 2.2 – Приклад інтерфейсу RabbitMQ Management Plugin

Переваги RabbitMQ Sensor:

- глибокий аналіз активності в RabbitMQ [3];
- виявлення і виправлення проблем у реальному часі;
- оптимізований для високої продуктивності RabbitMQ;

Недоліки RabbitMQ Sensor:

- обмежена застосовність за межами RabbitMQ;

- вимагає знань з RabbitMQ для ефективного використання;
- використовуються лише для моніторингу повідомлень, не надає можливості керування ними;

Також, RabbitMQ Management Plugin надає можливість керувати повідомленнями, але не на автоматизованому рівні.

2.3 Confluent Control Center

Confluent Control Center [4] - це інструмент для комплексного моніторингу, керування та налаштування Kafka [5]. Він є частиною платформи Confluent і дозволяє користувачам забезпечити безперебійну роботу Kafka, а також глибоко вивчати роботу своєї Kafka інфраструктури.

Рисунок 2.3 відображає приклад інтерфейсу Confluent Control Center.

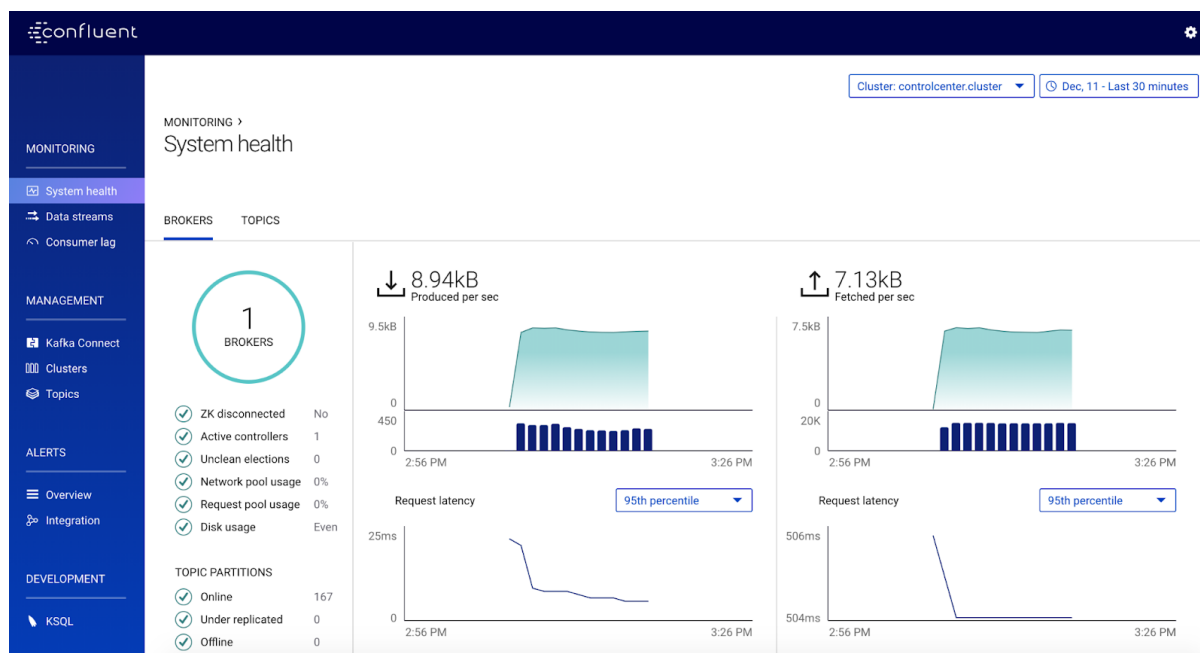


Рисунок 2.3 – Приклад інтерфейсу Confluent Control Center

Переваги Confluent Control Center:

- візуалізація роботи Kafka: стріми даних, топіки, споживачі та брокери;
- алертинг та сповіщення про можливі проблеми в реальному часі;

- глибокий моніторинг продуктивності, латентності, пропущених повідомлень та інших ключових показників;
- інтуїтивний інтерфейс для керування і налаштування Kafka;

Недоліки Confluent Control Center:

- в основному працює з Confluent Platform, яка є комерційним продуктом;
- велика вартість користування;
- вимагає певного рівня знань Kafka для ефективного використання;
- використовуються лише для моніторингу повідомлень, не надає можливості керування ними;

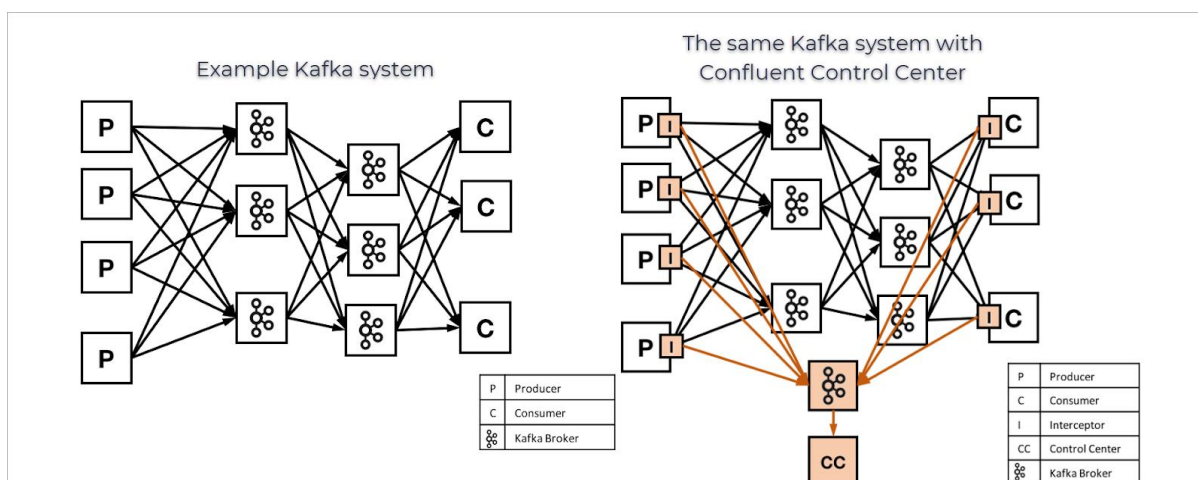


Рисунок 2.4 – Різниця Kafka та Kafka з Confluent Control Center

Confluent Control Center надає додаткові можливості для моніторингу Kafka, яких може бракувати в базовій версії. Це робить його корисним інструментом для компаній, які використовують Kafka на великому масштабі та прагнуть глибокого розуміння його роботи.

Рисунок 2.4 відображає різницю Kafka та Kafka з Confluent Control Center.

2.4 Hangfire

Hangfire [6] - це фреймворк для управління фоновими завданнями у .NET-додатках. Він дозволяє запускати довгі операції асинхронно, в тому числі механізм для повторного відправлення асинхронних повідомлень, забезпечуючи їх виконання в тлі, що дозволяє основному потоку програми не чекати завершення цих завдань.

Рисунок 2.5 відображає дашборд Hangfire.

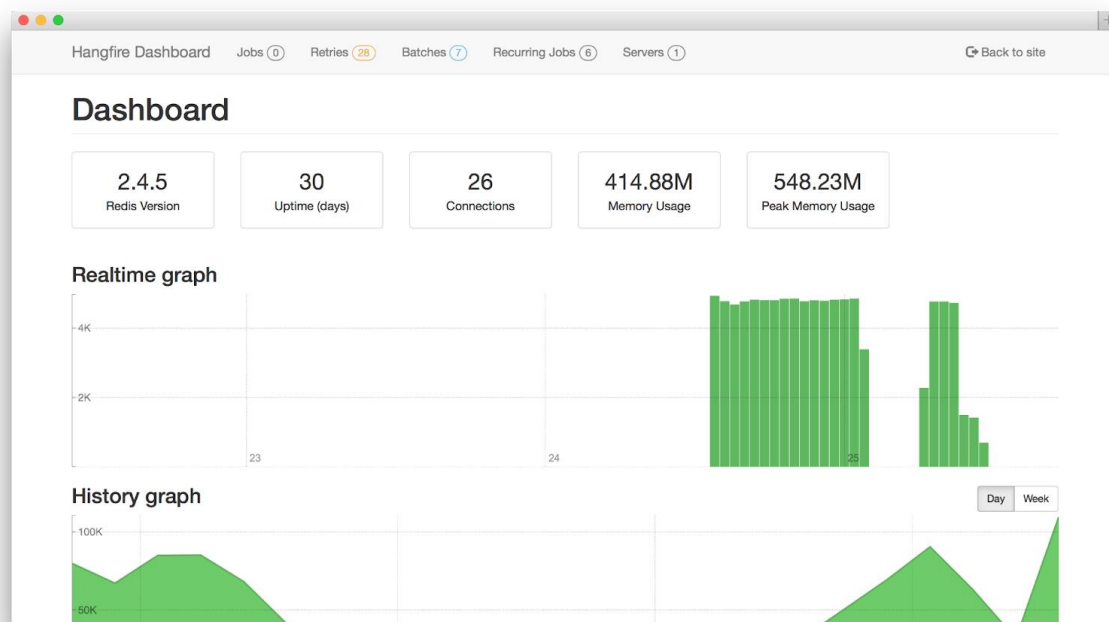


Рисунок 2.5 – Приклад дашборду Hangfire.

Переваги Hangfire:

- інтеграція безпосередньо з .NET, що робить його особливо корисним для додатків, що використовують цей стек;
- автоматичне планування, відновлення та повторне виконання завдань;
- гнучкість у налаштуванні: можливість задати певні дії на випадок невдачі, встановити максимальну кількість спроб та інше;
- можливість налаштувати механізм повторного відправлення асинхронних повідомлень;

Недоліки Hangfire:

- обмежена лише екосистемою .NET;

- хоча Hangfire може забезпечувати повторне виконання завдань, він не є спеціалізованим інструментом для асинхронної комунікації між мікросервісами;
- потреба в глибокому розумінні .NET для оптимальної настройки та використання;
- не використовується для зберігання повідомлень;

2.5 Elastic Stack (ELK)

ELK Stack [7] є комбінацією трьох продуктів: Elasticsearch, Logstash та Kibana. У контексті моніторингу повідомлень асинхронної комунікації між мікросервісами, ELK Stack використовується для агрегації, індексації та візуалізації логів повідомлень, дозволяючи слідкувати за їхнім життєвим циклом.

Рисунок 2.6 відображає приклад інтерфейсу ELK Stack.

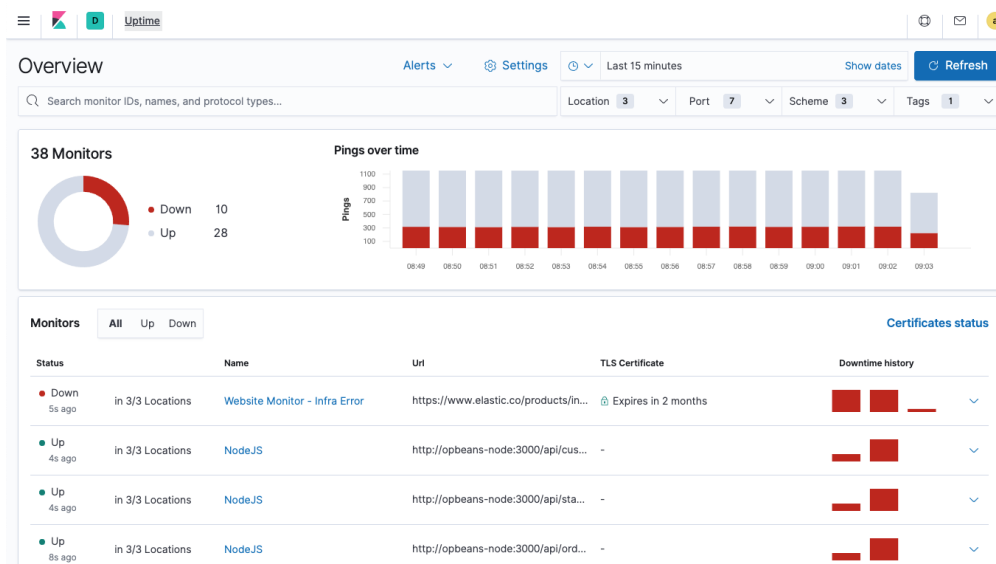


Рисунок 2.6 – Приклад інтерфейсу ELK Stack.

Переваги ELK Stack:

- з допомогою ELK можна відстежувати життєвий цикл кожного повідомлення в асинхронній системі;
- дозволяє аналізувати та з'ясовувати, чи було оброблено повідомлення, чи зазнало воно затримки або ж втрати;

- Kibana дозволяє відображати динаміку потоку повідомлень в реальному часі; Недоліки ELK Stack:
- оскільки ELK не було спеціально розроблено для моніторингу асинхронних повідомлень, його налаштування під цю конкретну задачу може вимагати додаткових зусиль;
- необхідність обробки великої кількості асинхронних повідомлень може збільшити вимоги до ресурсів;
- зберігання повідомлень на довготривалі терміни коштує багато;

Elastic Stack є дуже корисним інструментом для моніторингу асинхронних повідомлень у системах мікросервісів. Він допомагає командам ідентифікувати та аналізувати потенційні проблеми або затримки в обробці повідомлень.

2.6 Prometheus та Grafana

Prometheus [8] є системним моніторингом, який активно використовується в контексті моніторингу мікросервісів. Коли асинхронна комунікація між мікросервісами відбувається через брокери повідомлень або подібні системи, метрики про ці повідомлення можна збирати та відображати в Grafana [9].

Рисунок 2.7 відображає приклад інтерфейсу Grafana.



Рисунок 2.7 – Приклад інтерфейсу Grafana.

Переваги Prometheus та Grafana:

- Prometheus дає змогу визначити власні метрики, що дозволяє моніторити асинхронні повідомлення з довільним рівнем деталізації;
- завдяки високій продуктивності Prometheus можна моніторити асинхронні повідомлення в реальному часі;
- з Grafana користувачі можуть створювати налаштовувані графіки та дошки, що забезпечують глибокий аналіз повідомлень;

Недоліки Prometheus та Grafana:

- необхідно витратити час на розробку та налаштування специфічних метрик для моніторингу асинхронних повідомлень;
- інтеграція Prometheus з брокерами повідомлень може вимагати певних технічних знань;
- Prometheus не надає довгострокового зберігання даних "з коробки";

Використання Prometheus та Grafana для моніторингу асинхронних повідомлень дозволяє отримати глибокий інсайт в роботу системи комунікації. За допомогою цих інструментів можна виявляти затримки, втрати або інші аномалії в процесі обміну повідомленнями між мікросервісами.

2.7 Аналіз існуючих рішень

Проаналізувавши існуючі рішення були виявлені їхні функціональні переваги та недоліки. Ці дані відображені в таблиці 2.1.

Таблиця 2.1 - Підсумкові дані аналізу існуючих рішень

Критерій	DataDog	RabbitMQ Sensor	Confluent Control Center	Hangfire	Elastic Stack (ELK)	Prometheus та Grafana
Моніторинг повідомлень	+	+	+	-	+	+
Технічні метрики	+(не тільки для повідомлень)	+	+	-	+(не тільки для повідомлен ь)	+(не тільки для повідомлен ь)
Зберігання повідомлень	+-(довготривал е зберігання коштує багато)	+-(напряму в RabbitMQ)	+-(напряму в Kafka)	-	+-(довготрива ле зберігання коштує багато)	+-(довготрива ле зберігання коштує багато)
Можливість керування повідомлення м	-	+-(напряму в RabbitMQ)	+-(напряму в Kafka)	+(надає можливість імплементувати будь-які дії для керування повідомленнями)	-	-

Ці рішення представляють собою лише частину доступних на ринку засобів управління та моніторингу повідомлень. Кожне з них має свої особливості, що робить його гарним кандидатом для певних сценаріїв.

Можна побачити, що майже всі рішення надають інформативні технічні метрики, що є само по собі великим плюсом, але поставлена задача не потребує цих деталей.

Усі рішення не надають можливості доступного зберігання повідомлень на довготривали терміни, що є великим недоліком.

Лише Hangfire надає можливість автоматизованого керування повідомленнями, але це рішення потребує самостійної реалізації керування, що є недоліком.

Великою перевагою є то, що майже всі рішення надають можливість слідкувати за станами повідомлень певний час.

Отже, не одне з існуючих рішень не досягає мети поставлених в магістерській дисертації - збереження повідомлень асинхронної комунікації мікросервісів від втрат та моніторинг цих повідомлень.

Висновок до розділу 2

Аналізуючи різні інструменти управління та моніторингу повідомлень, як DataDog, RabbitMQ Sensor, Confluent Control Center, Hangfire, Elastic Stack та Prometheus з Grafana, можна зробити висновок про їхні переваги та обмеження. Хоча більшість з них надають детальні технічні метрики і дозволяють відслідковувати стан повідомлень, жоден з них повністю не задовольняє вимоги магістерської дисертації щодо довгострокового зберігання асинхронних повідомлень. Особливо важливою є відсутність функціоналу автоматизованого управління повідомленнями. Таким чином, існує потреба в розробці нового інноваційного рішення, яке б задовольнило ці специфічні потреби.

3 ФОРМУВАННЯ ВИМОГ СИСТЕМИ

3.1 Опис вимог до системи

Система керування повідомленнями асинхронної комунікації повинна задовольняти наступні вимоги.

Бути простою в використанні та налаштуванні.

Ця вимога є основоположною для забезпечення швидкої адаптації та впровадження системи у реальних умовах. Користувачі повинні мати змогу без труднощів зрозуміти функціонал системи та адаптувати її до своїх потреб.

Зберігати повідомлення на довготривалі терміни.

Це є важливою вимогою, оскільки це дозволяє забезпечити тривалу історію комунікації та можливість аналізу даних у майбутньому. Також це гарантує, що інформація не буде втрачена в разі технічних збоїв чи інших випадкових обставин.

Інтеграція з різноманітними брокерами повідомлень.

Інтеграція з різноманітними брокерами повідомлень забезпечує гнучкість системи та можливість використання її в різних інфраструктурних середовищах. Це відкриває можливість для безшовної міграції між різними платформами та забезпеченням сумісності з різними технологіями.

Мати змогу налаштувати механізм повторного відправлення повідомлень за критеріями, налаштованими користувачем.

Механізм повторного відправлення повідомлень є критичним для забезпечення надійності асинхронної комунікації. Зокрема, у випадках, коли початкове повідомлення не доходить до адресата через мережеві збої або інші проблеми. Можливість налаштовувати цей механізм дозволяє користувачам визначати, як і коли повідомлення повинно бути відправлено знову, гарантуючи при цьому ефективну комунікації.

3.2 Діаграма варіантів використання

Діаграма використання — це візуальне представлення взаємодій між користувачами (або "акторами") та системою. Вона є частиною методології UML (Unified Modeling Language) та використовується для моделювання функціональності системи з точки зору її користувачів. Основними елементами діаграми використання є актори та випадки використання.

Актори на такій діаграмі представляють ролі людей, систем або інших сутностей, які взаємодіють із системою. Вони можуть бути як зовнішніми (наприклад, клієнти або зовнішні сервіси), так і внутрішніми (наприклад, працівники компанії).

Випадки використання — це функції або задачі, які система може виконувати. Кожен випадок використання описує певну ціль, яку користувач або актор прагне досягти за допомогою системи.

Діаграма використання допомагає зрозуміти, як користувачі будуть взаємодіяти із системою, і які функції їм будуть необхідні. Вона також важлива для виявлення різних сценаріїв використання системи, що допомагає при проектуванні та розробці програмного забезпечення.

Діаграма варіантів використання відображена в додатку А. На ній відображено 2 актори - сама система, та користувач інтегруючий систему.

Система має можливість обробити повідомлення та провести певні дії в залежності від результату обробки, якщо під час обробки виникає проблема - дані повідомлення оновлюються за описаною політикою повторних спроб, для того щоб механізм повторної відправки повідомлень мав змогу знайти це повідомлення та повторно відправити його. Якщо повідомлення було успішно оброблено, статус цього повідомлення оновлюється та зберігається в базі даних.

Користувач, інтегруючий систему, має змогу підключити NuGet пакет системи, підключити систему в класі *Startup* чи *Program*, описати кастомний клас *Message*, щоб за бажанням додати певні дані до повідомлення, викликати сервіс для

запису повідомлень до бази при надсиланні повідомлення до брокера та описати політику повторних спроб. Ця політика включає в себе вказання назви черги для повторної відправки повідомлення, вказання максимальної кількості спроб повторної відправки повідомлення, опис статусів повідомлення, при яких повідомлення може бути повторно відправлене та опис часового інтервалу повторної відправки повідомлення.

Висновок до розділу 3

В цьому розділі було ретельно визначено ключові характеристики та функціональність, необхідні для ефективної системи керування повідомленнями в контексті асинхронної комунікації мікросервісів. Основна увага була приділена простоті використання та налаштування системи, що забезпечує легку адаптацію та впровадження в різних робочих середовищах.

Інтеграція з різноманітними брокерами повідомлень підкреслює гнучкість системи та її сумісність з широким спектром існуючих технологій.

Ключовим аспектом системи є можливість користувача налаштовувати механізм повторного відправлення повідомлень, що значно підвищує надійність комунікації в мікросервісних архітектурах. Це забезпечує ефективну обробку повідомлень навіть у разі збоїв чи втрати даних.

Також була описана та розроблена діаграма варіантів використання.

Таким чином, сформульовані вимоги до системи відображають необхідний баланс між функціональністю, надійністю, гнучкістю та зручністю користування, створюючи міцну основу для розробки ефективного рішення, призначеного для керування асинхронними повідомленнями мікросервісів.

4 ВИБІР ТЕХНОЛОГІЙ РЕАЛІЗАЦІЇ СИСТЕМИ

Одними з найбільш використовуваними платформами для розробки застосунків є .NET, Java, Python, Node.js, Ruby on Rails, та інші.

.NET [10] від Microsoft — це потужний фреймворк, що використовується для розробки веб, настільних, мобільних та хмарних застосунків.

Java [11] також є дуже популярною платформою, особливо для великих корпоративних застосунків та мобільних додатків на базі Android [25].

Python [12] є широко використовуваною мовою для веб-розробки завдяки фреймворкам, таким як Django [26] та Flask [27], а також відомий своєю ефективністю в аналітиці даних, штучному інтелекті та наукових дослідженнях.

Node.js [13], що працює на JavaScript [28], дозволяє розробникам створювати швидкі та масштабовані веб-застосунки.

Ruby on Rails [14] — це інший популярний фреймворк для веб-розробки, який відомий своєю простотою та швидкістю розробки.

Кожна з цих платформ має свої переваги та особливості, і вибір платформи зазвичай залежить від конкретних потреб проекту, вимог до продуктивності, доступних ресурсів, а також особистих уподобань розробника.

4.1 Вибір технологій

Найбільш привабливою з оглянутих технологій є платформа .NET, а саме тому, що:

Мова C#: C# [15] є сучасною, об'єктно-орієнтованою мовою програмування з чіткою синтаксичною структурою, що робить розробку продуктивною та приємною.

Висока продуктивність: Завдяки оптимізаціям JIT-компіляції та сучасному управлінню пам'яттю, .NET-застосунки можуть працювати дуже швидко.

Кросплатформеність: З випуском .NET Core, застосунки можна розгортати й розробляти не тільки на Windows, але й на macOS та Linux.

Єдина платформа для різних рішень: З .NET можна розробляти веб-застосунки, настільні програми, мобільні застосунки, мікросервіси, хмарні рішення та ігри.

Активна спільнота: Є велика кількість ресурсів, бібліотек, інструментів та фреймворків від спільноти розробників, що допомагає підтримувати та розширювати платформу.

Підтримка: Оскільки це продукт від Microsoft, є велика кількість офіційних ресурсів, документації та підтримки для розробників.

Також, платформа .NET має потужний фреймворк для роботи з базами даних - Entity Framework Core (EF Core) [16]. Ось декілька ключових аспектів, що роблять його важливим елементом в екосистемі .NET:

Об'єктно-реляційне відображення: EF Core дозволяє розробникам працювати з базами даних, використовуючи .NET об'єкти, значно знижуючи кількість коду, необхідного для роботи з даними.

Підтримка різних СУБД: Як і в Entity Framework, EF Core підтримує роботу з різними системами управління базами даних, включаючи SQL Server [17], MySQL [18], SQLite [19], PostgreSQL [20] та інші.

LINQ-запити: EF Core використовує LINQ [29] (Language Integrated Query) для формування запитів до бази даних, що робить код більш читабельним і знижує ймовірність помилок.

Міграції: Функціонал міграцій дозволяє легко управляти змінами у структурі бази даних, що є надзвичайно зручним у розробці та розгортанні застосунків.

Продуктивність та оптимізація: EF Core розроблений з урахуванням продуктивності, пропонуючи поліпшену обробку та оптимізацію запитів.

Гнучкість та розширюваність: EF Core може бути розширений та налаштований для задоволення конкретних вимог проекту.

Підтримка асинхронної обробки: EF Core підтримує асинхронні запити, що є ключовим аспектом для підвищення продуктивності веб-застосунків.

Підтримка розробників та спільноти: Як частина екосистеми .NET, EF Core має сильну підтримку як від Microsoft, так і від активної спільноти розробників.

Отже, завдяки своїм потужним можливостям та вбудованій підтримці, Entity Framework Core є ключовим компонентом для розробників, що використовують платформу .NET для створення сучасних, ефективних та масштабованих застосунків. Він не лише спрощує взаємодію з базами даних через ORM, але й пропонує гнучкість та розширені можливості, які відповідають сучасним вимогам програмування.

4.2 Бази даних

Розроблювана система має змогу працювати з будь-якими СУБД, будь-то релятивні чи NoSql [21], але для кожної використаної СУБД необхідно імплементувати окремого провайдера. Така методологія є досить розповсюдженою, та, наприклад, використовується в ORM фреймворці Entity Framework Core, який надає можливість обрати будь-якого провайдера для роботи з даними.

Але, в контексті магістерської дисертації як основний інструмент для зберігання та обробки даних було визначено систему управління базами даних MySQL. Даний вибір обумовлений рядом важливих переваг, які притаманні цій СУБД.

Висока продуктивність: Однією з ключових особливостей MySQL є її здатність швидко обробляти великі обсяги даних. Це досягається завдяки ефективним алгоритмам оптимізації та механізмам кешування, які гарантують миттєвий доступ до запитаних даних, що є критично важливим у наукових дослідженнях, де швидкодія може впливати на результати.

Надійність: MySQL відома своєю стабільністю і надійністю. Завдяки розробленим механізмам транзакцій, журналюванням та відновленню після збоїв,

дослідники можуть бути впевнені у збереженості своїх даних навіть у випадку технічних проблем.

Кросплатформеність: MySQL працює на множині операційних систем, що дозволяє дослідникам вибирати оптимальне середовище для своїх досліджень, незалежно від того, чи це Windows, Linux або macOS. Ця гнучкість спрощує інтеграцію та масштабування дослідницьких проектів.

Масштабованість: MySQL підтримує як невеликі додатки, так і системи з величезним обсягом даних. Завдяки її масштабованості, дослідники можуть починати з малих дослідницьких проектів та розширювати їх з ростом потреб, без необхідності зміни платформи.

Гнучкість: MySQL надає широкий спектр функцій для обробки даних. Підтримка численних типів даних, індексів, тригерів та збережених процедур робить її зручною для реалізації складних аналітичних завдань і витончених наукових алгоритмів.

Безпека: В умовах зростання загроз інформаційній безпеці, MySQL пропонує розширений набір механізмів безпеки, включаючи авторизацію на різних рівнях, шифрування даних та аудит.

Відкритий код: Перевагою MySQL є її відкритий вихідний код. Активна спільнота постійно працює над поліпшеннями, що гарантує постійне оновлення та вдосконалення системи, а також забезпечує доступ до найновіших технологічних рішень.

Інтеграція з іншими технологіями: MySQL може легко співпрацювати з різними програмними мовами та платформами. Ця інтеграційна здатність дозволяє дослідникам використовувати її в комбінації з іншими інструментами, що оптимізує процес наукового аналізу.

Економічна вигода: MySQL є доступною з економічної точки зору, що робить її ідеальним рішенням для наукових інституцій з обмеженими фінансовими ресурсами.

Широкий спектр інструментів: СУБД супроводжується багатим набором утиліт для адміністрування, резервного копіювання, оптимізації та моніторингу, що полегшує управління даними і забезпечує стабільність роботи системи.

У підсумку, вибір MySQL для магістерської дисертації є обґрунтованим рішенням, яке забезпечує потужний інструмент для роботи з даними.

4.3 Брокер повідомлень

Розроблювана система може бути сумісна з будь-якими брокерами повідомлень, проте вона не інтегрується з ними безпосередньо. Натомість, використання брокерів повідомлень відбувається на стороні користувача системи.

Існує багато різних брокерів повідомлень, наприклад:

Apache Kafka: Це розподілена потокова платформа, яка широко використовується для ефективної обробки великих потоків даних у реальному часі. Kafka особливо цінується за свою здатність до швидкої обробки та зберігання великих обсягів даних, а також за масштабування і надійність в розподілених системах.

ActiveMQ: Це ще один популярний брокер повідомлень, розроблений Apache. Він підтримує різноманітні протоколи обміну повідомленнями та відрізняється гнучкістю та легкістю у конфігурації.

Amazon SQS (Simple Queue Service) [30]: Це повністю керована чергова служба, яка надається Amazon Web Services (AWS). SQS дозволяє легко масштабувати та відправляти повідомлення між різними компонентами системи.

Microsoft Azure Service Bus: Це інтегрована платформа для обміну повідомленнями та інтеграції додатків, що надається у рамках хмарного сервісу Microsoft Azure.

Google Cloud Pub/Sub: Це хмарний сервіс від Google, який дозволяє обмінюватися повідомленнями між розподіленими додатками та службами.

IBM MQ (раніше WebSphere MQ): Це система обміну повідомленнями, яка забезпечує надійний і безпечний обмін даними між додатками та системами.

ZeroMQ: Це високопродуктивний асинхронний брокер повідомлень, який підтримує різні шаблони комунікації та легко інтегрується з різними мовами програмування.

NATS: Це легка та високопродуктивна система обміну повідомленнями, яка підтримує як синхронні, так і асинхронні комунікаційні шаблони.

Але в рамках магістерської дисертації буде використовуватись інший брокер повідомлень - RabbitMQ. Він є популярним брокером повідомлень, що використовується для обміну повідомленнями між різними компонентами або системами в розподілених архітектурах. Він забезпечує надійний та гнучкий спосіб обміну повідомленнями, який підтримує різноманітні сценарії обробки повідомлень. Далі наведені деякі ключові аспекти та особливості RabbitMQ.

Основні особливості:

Модель обміну повідомленнями AMQP: RabbitMQ використовує протокол Advanced Message Queuing Protocol (AMQP) для обміну повідомленнями, що забезпечує гнучкість та надійність.

Черги: Повідомлення зберігаються в чергах до того, як їх буде оброблено. Черги забезпечують буферизацію та забезпечують стійкість до збоїв у системі.

Обмінники (Exchanges): Обмінники приймають повідомлення від виробників та маршрутизують їх до однієї або декількох черг, засновуючись на правилах маршрутизації.

Зв'язування (Bindings): Зв'язування визначають правила, за якими повідомлення з обмінників маршрутизуються до черг.

Підтримка різних клієнтів та бібліотек: RabbitMQ може взаємодіяти з багатьма мовами програмування та платформами завдяки широкій підтримці клієнтських бібліотек.

Архітектурні особливості:

Масштабування та надійність: RabbitMQ підтримує кластеризацію для масштабування та високу доступність.

Транзакції: RabbitMQ підтримує транзакції, що дозволяє гарантувати доставку повідомлень.

Підтримка відкатування повідомлень (Message Acknowledgment): RabbitMQ дозволяє споживачам підтверджувати обробку повідомлень, що забезпечує їх надійну доставку.

Гнучкість у маршрутизації: RabbitMQ пропонує різні типи обмінників, такі як прямі, тематичні, заголовні та розподілені, для різних сценаріїв маршрутизації повідомлень.

Додаткові Аспекти та Функціональність:

Підтримка Різних Протоколів: Крім AMQP, RabbitMQ підтримує інші протоколи, включаючи MQTT та STOMP, що робить його гнучким рішенням для різних типів додатків.

Менеджмент і Моніторинг: RabbitMQ пропонує розширені можливості для моніторингу та управління чергами, обмінниками, зв'язуваннями та повідомленнями через веб-інтерфейс та API.

Безпека: RabbitMQ підтримує SSL/TLS для шифрування даних та механізми аутентифікації, що забезпечує безпечну передачу повідомлень.

Плагіни: Система підтримує різноманітні плагіни, які дозволяють розширювати її функціональність, включаючи інтеграцію з іншими системами та покращення моніторингу.

RabbitMQ ефективно використовується у широкому спектрі застосувань, включаючи:

Розподілені системи: Для забезпечення взаємодії між різними компонентами в розподілених середовищах.

Мікросервісні архітектури: Як засіб комунікації між мікросервісами.

Асинхронна обробка: Для відкладеної або фонової обробки завдань.

Інтеграція систем: Як міст для обміну даними між різними додатками та сервісами.

RabbitMQ є надійним, масштабованим і гнучким рішенням для обміну повідомленнями, яке може бути адаптоване під різні потреби від простих до дуже складних систем обміну повідомленнями. Його здатність до інтеграції з різноманітними мовами програмування та середовищами робить його ідеальним вибором для багатьох організацій та проектів, в тому числі для магістерської дисертації.

Висновок до розділу 4

Цей розділ підкреслює критичне значення правильного вибору технологічних інструментів для створення ефективної системи керування повідомленнями в асинхронній комунікації мікросервісів. Визначення .NET як основної платформи для розробки було зроблено на основі її високої продуктивності, кросплатформеності, єдиної платформи для різних рішень та широкої підтримки з боку спільноти та Microsoft.

Зокрема, мова програмування C# та фреймворк Entity Framework Core (EF Core) відіграють ключові ролі в розробці завдяки їх продуктивності, гнучкості та інтеграційним можливостям. EF Core особливо цінний через його об'єктно-реляційне відображення, підтримку різних СУБД, LINQ-запитів та зручності в управлінні змінами бази даних через міграції.

Для зберігання та обробки даних обрано MySQL, який відповідає вимогам високої продуктивності, надійності, кросплатформеності, масштабованості та безпеки. Його вибір обґрунтовано також відкритим кодом, інтеграційною здатністю та економічною доступністю.

У контексті брокерів повідомлень було вирішено використовувати RabbitMQ, що підкреслює його універсальність, масштабованість, надійність і гнучкість у маршрутизації повідомлень. RabbitMQ вибрано за його спроможність

працювати в складних розподілених системах і мікросервісних архітектурах, що є ключовим для успішної реалізації проекту.

Отже, вибір технологій реалізації системи, включаючи .NET, MySQL, та RabbitMQ, є добре обгрунтованим і відповідає вимогам проекту. Важливо відзначити, що кожен з цих інструментів має свою роль у вирішенні специфічних задач проекту, забезпечуючи потрібний баланс між продуктивністю, гнучкістю, та зручністю розробки. Це дає змогу створити систему, яка не тільки ефективно вирішує поставлені задачі, але й може легко адаптуватися до майбутніх вимог та розширень, забезпечуючи тривалу стійкість та ефективність використання.

5 РОЗРОБЛЕННЯ СИСТЕМИ

5.1 Основні сутності

В розроблюваній системі є лише одна сутність для збереження в базі даних, це сутність Message, яка описує стан відправленого повідомлення та описує необхідні поля для роботи механізму повторного відправлення повідомлення.

Схема таблиці Messages репрезентує дані про повідомлення. Схема відображена на таблиці 5.1

Таблиця 5.1 - Схема таблиці Messages.

Стовпець	Тип даних	Пояснення
Id	int	унікальний ідентифікатор повідомлення
Content	longtext	контент відправленого повідомлення, репрезентований як строка
ExecutionStatus	int	статус повідомлення
CreatedOn	datetime	репрезентує коли повідомлення було створено в системі, іншими словами, коли вони було відправлене
ModifiedOn	datetime?	репрезентує коли повідомлення було оновлено в системі
Queue	longtext	назва черги до якої публікується відправлене повідомлення
ExecuteAt	datetime?	репрезентує коли потрібно повторно відправити повідомлення, використовується механізмом повторного відправлення повідомлень
MaxRetryAttempts	int	репрезентує максимальну кількість спроб

		повторного опрацювання повідомлення
TotalRetryAttempts	int	репрезентує поточну кількість спроб, які було проведено для повторного опрацювання повідомлення

Може виникнути питання, чи не треба винести дані щодо механізму повторюваного відправлення повідомлення в окрему таблицю. Це не було зроблено за наступних причин.

Тісний зв'язок з повідомленням: Ці поля напряду пов'язані з конкретним повідомленням. Вони не мають самостійного значення поза контекстом цього повідомлення, тому логічно зберігати їх у тій самій таблиці.

Ефективність запитів: Інтеграція цих полів у основну таблицю полегшує виконання запитів, оскільки не потрібно здійснювати з'єднання таблиць для отримання повної інформації про повідомлення та його статус повторного відправлення.

Нормалізація даних: Немає необхідності у створенні окремої таблиці для полів, які відносяться до одного повідомлення і не дублюються в інших місцях бази даних. Розбиття їх на окремі таблиці було би надмірною нормалізацією.

Спрощене управління: Зберігання цих даних в одній таблиці спрощує управління, оновлення та видалення інформації. Якщо ці поля були б у окремій таблиці, додаткові операції з'єднання таблиць стали б більш складними і ресурсомісткими.

Отже, на основі вищезазначеного, можна зробити висновок, що ExecuteAt, MaxRetryAttempts та TotalRetryAttempts повинні залишитися в таблиці Messages з іншою інформацією про повідомлення для оптимальної організації даних та ефективності роботи системи.

Сутність Message репрезентується в кодовій базі системи як клас Message, що відображений на рисунку 5.1.

```

public class Message
{
    [1 usage]
    public int Id { get; set; }
    [2 usages]
    public string Content { get; set; }
    [5 usages]
    public ExecutionStatus Status { get; set; }
    [1 usage]
    public DateTime CreatedOn { get; set; }
    [1 usage]
    public DateTime? ModifiedOn { get; set; }
    [3 usages]
    public string Queue { get; set; }
    [2 usages]
    public DateTime? ExecuteAt { get; set; }
    [2 usages]
    public int MaxRetryAttempts { get; set; }
    [2 usages]
    public int TotalRetryAttempts { get; set; }

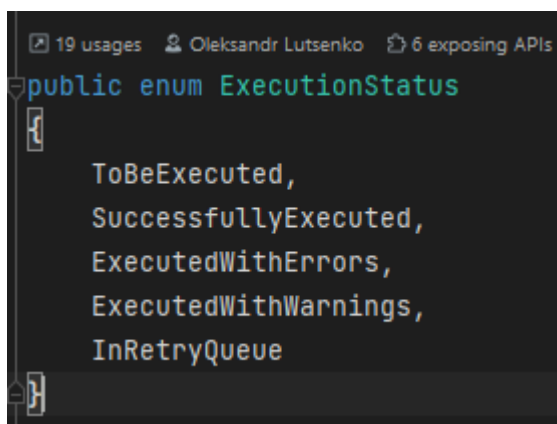
    [2 usages] Oleksandr Lutsenko
    public void Fill(object content, string queue, ExecutionStatus status = ExecutionStatus.ToBeExecuted)
    {
        Content = JsonSerializer.Serialize(content);
        Status = status;
        Queue = queue;
    }
}

```

Рисунок 5.1 – Клас Message.

ExecutionStatus репрезентує можливі статуси повідомлення, такі як: “До Виконання”, “Успішно Виконано”, “Виконано З Помилками”, “Виконано З Попередженнями”, “У Черзі Повторного Відправлення”.

ExecutionStatus представляє собою тип даних Enum, який гарно підходить для роботи зі статусами чи інших зв’язаних між собою констант. Цей тип відображений на рисунку 5.2.



```

19 usages  Oleksandr Lutsenko  6 exposing APIs
public enum ExecutionStatus
{
    ToBeExecuted,
    SuccessfullyExecuted,
    ExecutedWithErrors,
    ExecutedWithWarnings,
    InRetryQueue
}

```

Рисунок 5.2 – Enum ExecutionStatus

Користувачі системи можуть вказувати статус повідомлення при його опрацюванні, та приймати дії ті рішення в залежності від цього статусу в майбутньому.

ER діаграма відображена в додатку Б.

Діаграма класів відображена в додатку В.

5.2 Інтеграція з базою даних

Як було описано в розділі 4.2 - для роботи з базою даних буде використовуватись фреймворк Entity Framework Core.

Для його установки потрібно знайти пакет Microsoft.EntityFrameworkCore в бібліотеці пакетів NuGet та встановити його.

Далі буде описано клас *MessagesContext*, який репрезентує сесію з базою даних і може використовуватися для запитів і збереження даних, лістинг коду цього класу відображено на рисунку 5.3.

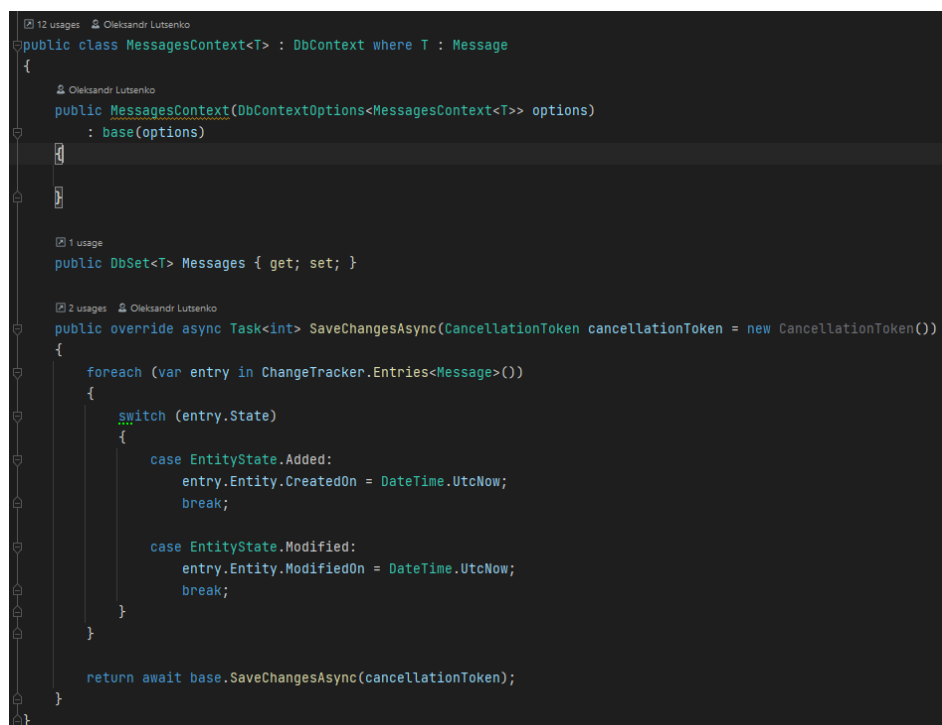
Цей клас повинен наслідувати клас *DbContext* з пакету Entity Framework Core, щоб встановити зв'язок з базою даних.

Конструкція *where T : Message* визначає обмеження для узагальненого типу *T*, яке гарантує, що *T* може бути лише типом *Message* або його нащадком. Це означає, що *MessagesContext<T>* може бути інстанційований з будь-яким типом, що наслідується від класу *Message*.

В контексті Entity Framework, це означає, що *MessagesContext<T>* може використовуватися для створення, читання, оновлення та видалення екземплярів *Message* (або його нащадків) в базі даних.

Також клас *MessagesContext* повинен мати в собі посилання та реєстрацію таблиці *Message*, а саме - *public DbSet<T> Messages { get; set; }*

Крім того, цей клас може перевизначити логіку збереження даних, для цього треба додати override метод *SaveChangesAsync*, який крім збереження даних даних також записує чи оновлює дані полей *CreatedOn* та *ModifiedOn* в залежності від стану зберезуванного об'єкту.



```

12 usages Oleksandr Lutsenko
public class MessagesContext<T> : DbContext where T : Message
{
    Oleksandr Lutsenko
    public MessagesContext(DbContextOptions<MessagesContext<T>> options)
        : base(options)
    {
    }

    1 usage
    public DbSet<T> Messages { get; set; }

    2 usages Oleksandr Lutsenko
    public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken = new CancellationToken())
    {
        foreach (var entry in ChangeTracker.Entries<Message>())
        {
            switch (entry.State)
            {
                case EntityState.Added:
                    entry.Entity.CreatedOn = DateTime.UtcNow;
                    break;

                case EntityState.Modified:
                    entry.Entity.ModifiedOn = DateTime.UtcNow;
                    break;
            }
        }

        return await base.SaveChangesAsync(cancellationToken);
    }
}

```

Рисунок 5.3 – Клас MessagesContext

5.3 Логіка зберігання повідомлень

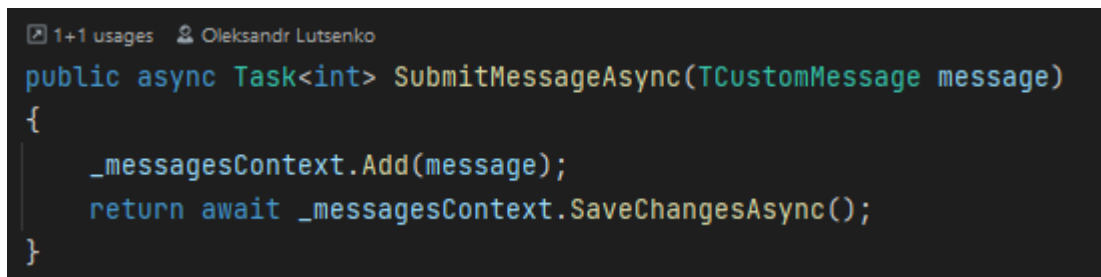
Далі потрібно розробити сервіс для зберігання і оновлення даних повідомлення в базі даних.

Для цього треба об'явити клас *MessageService<TCustomMessage> : IMessageService<TCustomMessage> where TCustomMessage : Message, new()*. Знову

конструкція *where TCustomMessage : Message* визначає обмеження для узагальненого типу *TCustomMessage*, яке гарантує, що *TCustomMessage* може бути лише типом *Message* або його нащадком. Це означає, що *MessageService<T>* може бути інстанційований з будь-яким типом, що наслідується від класу *Message*. Також тут визначається *new()*, що надає можливість створення об'єкту типу *TCustomMessage* в цьому класі.

Клас *MessageService* використовує імплементований раніше клас *MessagesContext<TCustomMessage>* для операцій з базою даних.

Далі імплементується метод *SubmitMessageAsync(TCustomMessage)* для збереження повідомлення в базі даних. Цей метод має в параметрах тип *TCustomMessage*, який відображає, що як аргумент можна передати лише клас, який був наслідований від типу *Message*. Лістинг цього методу відображено на рисунку 5.4.



```

1+1 usages Oleksandr Lutsenko
public async Task<int> SubmitMessageAsync(TCustomMessage message)
{
    _messagesContext.Add(message);
    return await _messagesContext.SaveChangesAsync();
}

```

Рисунок 5.4 – Метод *SubmitMessageAsync*

Далі буде описаний клас

CreateAndSubmitMessageAsync<TMessageContent>(TMessageContent content, string queue, ExecutionStatus status = ExecutionStatus.ToBeExecuted).

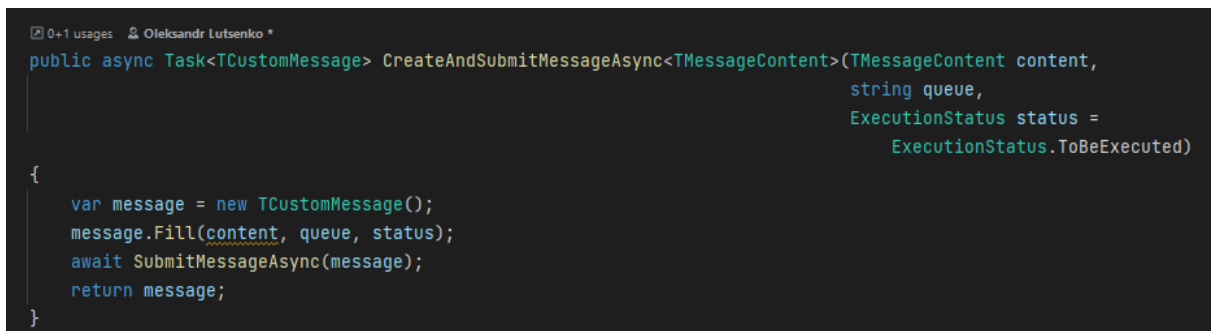
Цей метод, на відміну від методу *SubmitMessageAsync(TCustomMessage)*, потрібен не лише для запису повідомлення до бази даних, але і для створення об'єкту повідомлення для запису.

TMessageContent являє собою контент повідомлення для відправки, *queue* відображає назву черги, до якої було відправлене повідомлення, *status* відображає статус повідомлення при записі його до бази даних.

Як можна побачити на рисунку 5.5, спочатку створюється об'єкт повідомлення для запису, потім цей об'єкт заповнюється даними за допомогою методу *Fill* класу *Message*.

Код цього методу відображений на рисунку 5.6, на ньому можна побачити, запис полів повідомлення, в тому числі, що контент повідомлення переводиться в json.

В методу *CreateAndSubmitMessageAsync*, після запису даних повідомлення, в об'єкт повідомлення зберігається в базі та повертається з методу за допомогою *return*.

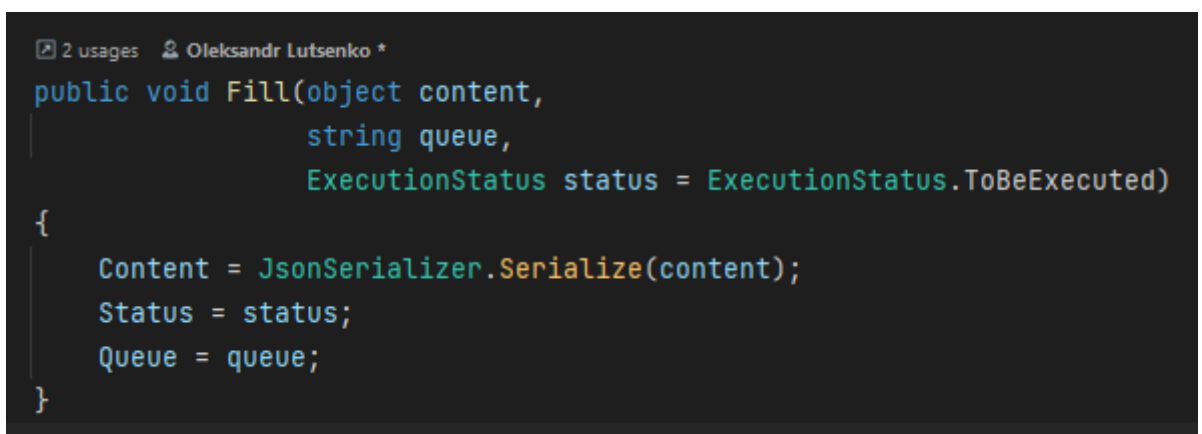


```

0+1 usages Oleksandr Lutsenko *
public async Task<TCustomMessage> CreateAndSubmitMessageAsync<TMessageContent>(TMessageContent content,
                                                                              string queue,
                                                                              ExecutionStatus status =
                                                                              ExecutionStatus.ToBeExecuted)
{
    var message = new TCustomMessage();
    message.Fill(content, queue, status);
    await SubmitMessageAsync(message);
    return message;
}

```

Рисунок 5.5 – Метод *CreateAndSubmitMessageAsync*



```

2 usages Oleksandr Lutsenko *
public void Fill(object content,
                string queue,
                ExecutionStatus status = ExecutionStatus.ToBeExecuted)
{
    Content = JsonSerializer.Serialize(content);
    Status = status;
    Queue = queue;
}

```

Рисунок 5.6 – Метод *Fill* класу *Message*

Далі буде описаний метод *UpdateMessageAsync(TCustomMessage message, RetryPolicy retryPolicy = null)* для оновлення даних повідомлення, який треба

використовувати після опрацювання повідомлення, для оновлення його статусу. Лістинг цього методу відображено на рисунку 5.7.

Цей метод, як і метод *SubmitMessageAsync*, має в параметрах *TCustomMessage*, тип, який наслідується від типу *Message*.

Далі метод оновлює дані та зберігає зміни в базі даних, також, він, як і метод *SubmitMessageAsync*, повертає кількість змінених повідомлень, щоб користувач цього методу мав змогу перевірити, чи збереглися його зміни.

```
public async Task<int> UpdateMessageAsync(TCustomMessage message, RetryPolicy retryPolicy = null)
{
    if (retryPolicy != null)
    {
        message.ExecuteAt = DateTime.UtcNow.Add(retryPolicy.RetryInterval);
        message.MaxRetryAttempts = retryPolicy.MaxRetryAttempts;
    }
    _messagesContext.Update(message);
    return await _messagesContext.SaveChangesAsync();
}
```

Рисунок 5.7 – Метод *UpdateMessageAsync*

Лістинг коду всього сервісу представлено на рисунку 5.8.

```

1 usage Oleksandr Lutsenko *
public class MessageService<TCustomMessage> : IMessageService<TCustomMessage> where TCustomMessage : Message, new()
{
    private readonly MessagesContext<TCustomMessage> _messagesContext;

    Oleksandr Lutsenko
    public MessageService(MessagesContext<TCustomMessage> messagesContext)
    {
        _messagesContext = messagesContext;
    }
}

0+3 usages Oleksandr Lutsenko
public async Task<int> UpdateMessageAsync(TCustomMessage message, RetryPolicy retryPolicy = null)
{
    if (retryPolicy != null)
    {
        message.ExecuteAt = DateTime.UtcNow.Add(retryPolicy.RetryInterval);
        message.MaxRetryAttempts = retryPolicy.MaxRetryAttempts;
    }
    _messagesContext.Update(message);
    return await _messagesContext.SaveChangesAsync();
}

1+1 usages Oleksandr Lutsenko
public async Task<int> SubmitMessageAsync(TCustomMessage message)
{
    _messagesContext.Add(message);
    return await _messagesContext.SaveChangesAsync();
}

0+1 usages Oleksandr Lutsenko *
public async Task<TCustomMessage> CreateAndSubmitMessageAsync<TMessageContent>(TMessageContent content,
                                                                              string queue,
                                                                              ExecutionStatus status =
                                                                              ExecutionStatus.ToBeExecuted)
{
    var message = new TCustomMessage();
    message.Fill(content, queue, status);
    await SubmitMessageAsync(message);
    return message;
}

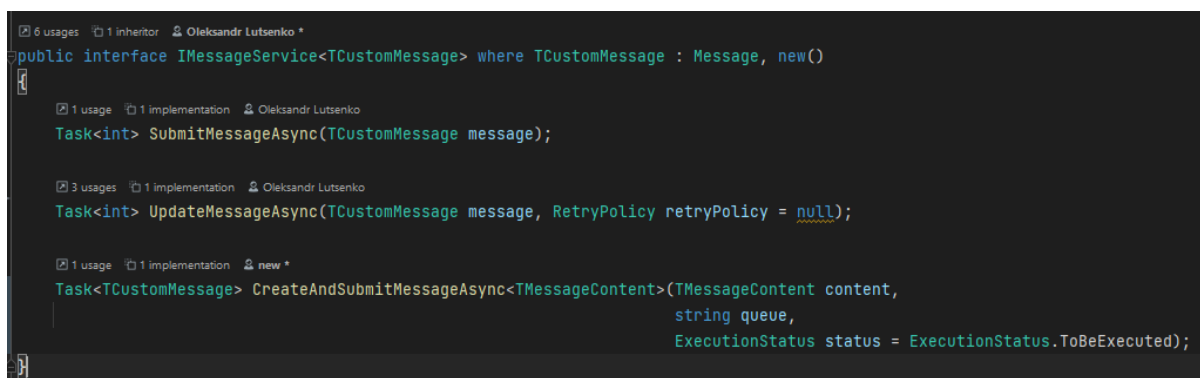
```

Рисунок 5.8 – Клас *MessageService<TCustomMessage>*

Також, треба обов'язково описати інтерфейс *IMessageService<TCustomMessage>*, який визначає контракт для класу *MessageService<TCustomMessage>*. Одна з причин використання інтерфейсів, є те, що користувачам не потрібно думати про реалізацію цих методів та шукати певний клас, який наслідує цей інтерфейс та реалізує необхідну логіку, усю відповідальність на бере на себе інтерфейс.

Крім того, цей інтерфейс повинен мати ті ж самі узагальнені конструкції як і клас, який наслідує інтерфейс.

Лістинг коду інтерфейса *IMessageService<TCustomMessage>* відображається на рисунку 5.9.



```

6 usages 1 inheritor Oleksandr Lutsenko *
public interface IMessageService<TCustomMessage> where TCustomMessage : Message, new()
{
    1 usage 1 implementation Oleksandr Lutsenko
    Task<int> SubmitMessageAsync(TCustomMessage message);

    3 usages 1 implementation Oleksandr Lutsenko
    Task<int> UpdateMessageAsync(TCustomMessage message, RetryPolicy retryPolicy = null);

    1 usage 1 implementation new *
    Task<TCustomMessage> CreateAndSubmitMessageAsync<TMessageContent>(TMessageContent content,
                                                                    string queue,
                                                                    ExecutionStatus status = ExecutionStatus.ToBeExecuted);
}

```

Рисунок 5.9 – Інтерфейс *IMessageService<TCustomMessage>*

5.4 Механізм повторного відправлення повідомлень

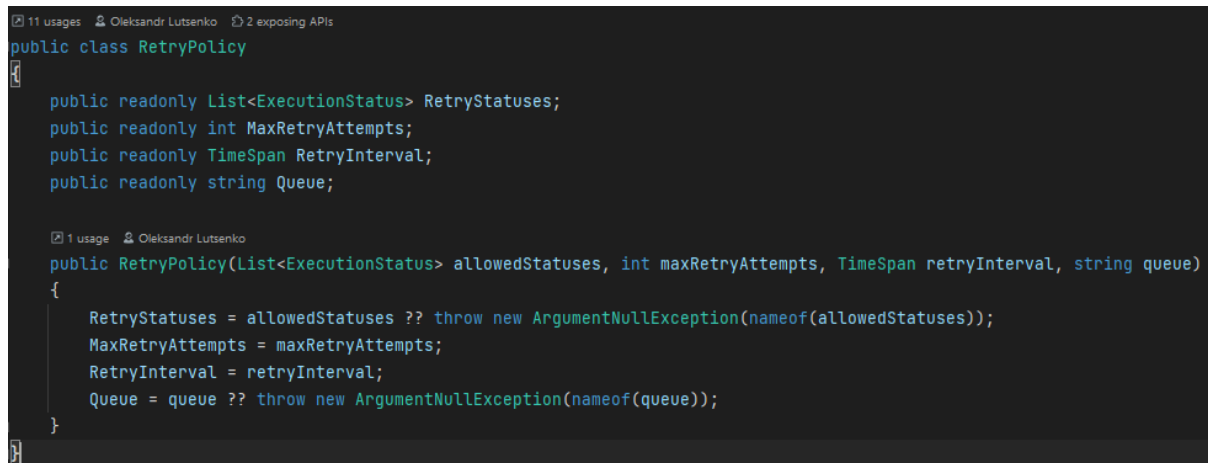
Далі буде описано типи даних та логіку роботи механізму повторного відправлення повідомлень.

Для початку буде описано модель політики клас *RetryPolicy*, яку повинні використовувати користувачі механізму повторного відправлення повідомлень, щоб обозначити як і коли повідомлення повинно бути повторно відправлене.

Як видно на рисунку 5.10 цей клас має 4 поля:

- список *ExecutionStatus*, щоб користувач мав змогу перелічити статуси повідомлення, при яких повідомлення повинно бути повторного відправлене;
- *MaxRetryAttempts*, репрезентує максимум спроб повторного відправлення повідомлення, якщо повідомлення буде повторного відправлене максимальну кількість разів - воно більше не буде повторного надсилатись;
- *RetryInterval* репрезентує інтервал з яким повідомлення повинно бути повторно відправлене;
- *Queue* слугує для того, щоб обозначити в яку чергу треба відправляти повідомлення;

Також цей клас має конструктор, який записує дані в об'єкт та не дозволяє не вказувати можливі статуси на назву черги.



```
public class RetryPolicy
{
    public readonly List<ExecutionStatus> RetryStatuses;
    public readonly int MaxRetryAttempts;
    public readonly TimeSpan RetryInterval;
    public readonly string Queue;

    public RetryPolicy(List<ExecutionStatus> allowedStatuses, int maxRetryAttempts, TimeSpan retryInterval, string queue)
    {
        RetryStatuses = allowedStatuses ?? throw new ArgumentNullException(nameof(allowedStatuses));
        MaxRetryAttempts = maxRetryAttempts;
        RetryInterval = retryInterval;
        Queue = queue ?? throw new ArgumentNullException(nameof(queue));
    }
}
```

Рисунок 5.10 – Клас RetryPolicy

Далі буде описано клас *RetryPolicyBuilder*, ціль якого забезпечити зручний інтерфейс для створення об'єктів класу *RetryPolicy*.

Builder - паттерн, який використовується в *RetryPolicyBuilder*, дозволяє крок за кроком створювати складні об'єкти. Він дозволяє користувачам зручно встановлювати параметри, які їм потрібні, в тому числі забезпечуючи для інших параметрів значення за замовчуванням.

Клас *RetryPolicyBuilder* містить методи, що повертають екземпляр самого себе, це дозволяє використовувати ланцюжковий виклик методів, відомий як *Fluent Interface*. Цей підхід робить код, який використовує білдер, більш читабельним і легким для розуміння.

Метод *WithExecutionStatus* дозволяє встановити конкретні статуси повідомлень, для яких потрібно застосувати політику повторних спроб. Якщо метод не викликається, використовується список за замовчуванням, який містить статуси *ToBeExecuted*, *ExecutedWithWarnings* та *ExecutedWithErrors*.

Метод *WithMaxRetryAttempts* дозволяє задати максимальну кількість спроб повторного відправлення повідомлення.

Метод *WithRetryInterval* встановлює часовий інтервал між повторними спробами відправлення повідомлення.

Метод *WithQueue* вказує, до якої черги будуть направлені повідомлення, які потребують повторної відправки.

Нарешті, метод *Build* використовується для створення екземпляра *RetryPolicy*, ініціалізованого відповідно до заданих параметрів.

Цей механізм будівника дозволяє легко розширювати і змінювати логіку створення об'єктів, не змінюючи код, який ці об'єкти використовує. Якщо в майбутньому з'являться нові поля або концепції для *RetryPolicy*, до білдера можна буде легко додати нові методи, не порушуючи існуючі інтерфейси.

Для ілюстрації, як може використовуватися клас *RetryPolicyBuilder*, наводиться приклад на рисунку 5.11.

```
public static RetryPolicy RetryPolicy = new RetryPolicyBuilder()
    .WithQueue("products")
    .WithExecutionStatus(ExecutionStatus.ExecutedWithErrors)
    .WithMaxRetryAttempts(3)
    .WithRetryInterval(TimeSpan.FromSeconds(30)) // RetryPolicyBuilder
    .Build();
```

Рисунок 5.11 – Приклад використання *RetryPolicyBuilder*

У цьому прикладі створюється політика повторних спроб, яка буде застосовуватися до повідомлень зі статусом *ExecutedWithErrors*, з максимум трьома спробами повторного відправлення, з інтервалом в 30 секунд, і повідомлення будуть відправлені до черги "products".

Лістинг коду класа *RetryPolicyBuilder* відображено на рисунку 5.12.

```

5 usages Oleksandr Lutsenko 4 exposing APIs
public class RetryPolicyBuilder
{
    public readonly List<ExecutionStatus> _defaultRetryStatuses = new ()
    {
        ExecutionStatus.ToBeExecuted, ExecutionStatus.ExecutedWithWarnings, ExecutionStatus.ExecutedWithErrors
    };
    private List<ExecutionStatus> _retryStatuses = new();
    private int _maxRetryAttempts = 5;
    private TimeSpan _retryInterval = TimeSpan.FromMinutes(2);
    private string _queue;

    1 usage Oleksandr Lutsenko
    public RetryPolicyBuilder WithExecutionStatus(params ExecutionStatus[] statuses)
    {
        _retryStatuses.AddRange(statuses);
        return this;
    }

    1 usage Oleksandr Lutsenko
    public RetryPolicyBuilder WithMaxRetryAttempts(int maxAttempts)
    {
        _maxRetryAttempts = maxAttempts;
        return this;
    }

    1 usage Oleksandr Lutsenko
    public RetryPolicyBuilder WithRetryInterval(TimeSpan interval)
    {
        _retryInterval = interval;
        return this;
    }

    1 usage Oleksandr Lutsenko
    public RetryPolicyBuilder WithQueue(string queue)
    {
        _queue = queue;
        return this;
    }

    1 usage Oleksandr Lutsenko
    public RetryPolicy Build()
    {
        return new RetryPolicy(_retryStatuses.Count > 0
            ? _retryStatuses
            : _defaultRetryStatuses,
            _maxRetryAttempts,
            _retryInterval,
            _queue);
    }
}

```

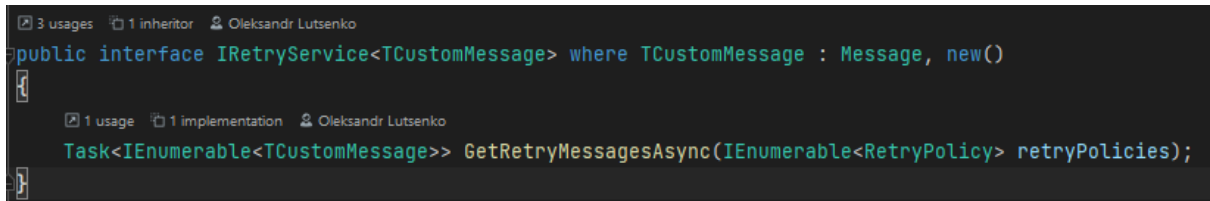
Рисунок 5.12 – Клас *RetryPolicyBuilder*

Далі буде описано сервіс для роботи з механізмом повторного відправлення повідомлень.

Сервіс *RetryService<TCustomMessage>* реалізовано як загальний клас, що впроваджує інтерфейс *IRetryService<TCustomMessage>*, де *TCustomMessage* є нащадком класу *Message*. Цей сервіс використовує контекст

MessagesContext<TCustomMessage>, щоб управляти повідомленнями, які можуть потребувати повторної відправки.

Лістинг цього інтерфейсу відображено на рисунку 5.13.



```

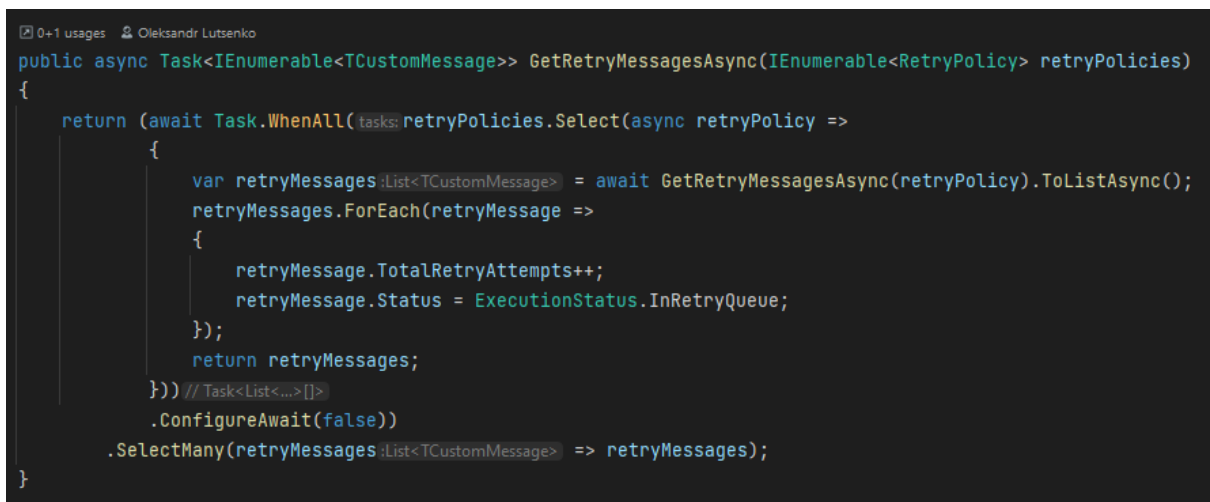
3 usages 1 inheritor Oleksandr Lutsenko
public interface IRetryService<TCustomMessage> where TCustomMessage : Message, new()
{
    1 usage 1 implementation Oleksandr Lutsenko
    Task<IEnumerable<TCustomMessage>> GetRetryMessagesAsync(IEnumerable<RetryPolicy> retryPolicies);
}

```

Рисунок 5.13 – Інтерфейс *IRetryService*

Конструктор класу приймає контекст повідомлень як залежність і ініціалізує приватне поле *_messagesContext*, яке потім використовується для взаємодії з повідомленнями в базі даних.

Метод *GetRetryMessagesAsync* приймає колекцію об'єктів *RetryPolicy* і асинхронно обробляє кожну політику, використовуючи лінію задач для одночасного отримання повідомлень, які відповідають кожній політиці повторних спроб. Для кожного повідомлення, яке підлягає повторній відправці, він збільшує лічильник спроб і змінює статус на *ExecutionStatus.InRetryQueue*, готуючи його до повторної відправки. Цей метод відображено на рисунку 5.14.



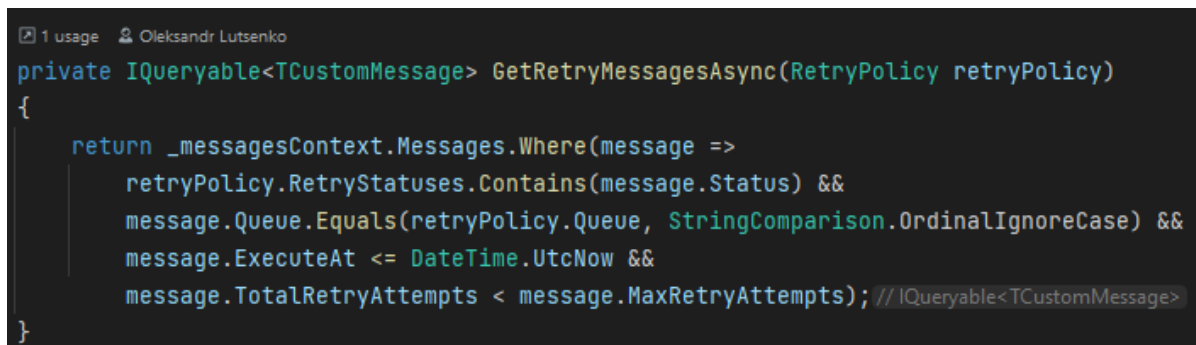
```

0+1 usages Oleksandr Lutsenko
public async Task<IEnumerable<TCustomMessage>> GetRetryMessagesAsync(IEnumerable<RetryPolicy> retryPolicies)
{
    return (await Task.WhenAll(tasks: retryPolicies.Select(async retryPolicy =>
    {
        var retryMessages:List<TCustomMessage> = await GetRetryMessagesAsync(retryPolicy).ToListAsync();
        retryMessages.ForEach(retryMessage =>
        {
            retryMessage.TotalRetryAttempts++;
            retryMessage.Status = ExecutionStatus.InRetryQueue;
        });
        return retryMessages;
    }))) // Task<List<...>[]>
        .ConfigureAwait(false))
        .SelectMany(retryMessages:List<TCustomMessage> => retryMessages);
}

```

Рисунок 5.14 – Публічний метод *GetRetryMessagesAsync*

GetRetryMessagesAsync є приватним допоміжним методом, який виконує фільтрацію повідомлень з контексту згідно із заданою політикою повторних спроб. Він перевіряє статус повідомлення, відповідність черги, час виконання та кількість попередніх спроб відправки. Цей метод відображено на рисунку 5.15.



```
1 usage  Oleksandr Lutsenko
private IQueryable<TCustomMessage> GetRetryMessagesAsync(RetryPolicy retryPolicy)
{
    return _messagesContext.Messages.Where(message =>
        retryPolicy.RetryStatuses.Contains(message.Status) &&
        message.Queue.Equals(retryPolicy.Queue, StringComparison.OrdinalIgnoreCase) &&
        message.ExecuteAt <= DateTime.UtcNow &&
        message.TotalRetryAttempts < message.MaxRetryAttempts); // IQueryable<TCustomMessage>
}
```

Рисунок 5.15 – Приватний метод *GetRetryMessagesAsync*

Цей сервіс є ключовим компонентом у механізмі повторного відправлення повідомлень. Він забезпечує абстракцію над нижчим рівнем взаємодії з базою даних, що дозволяє легко інтегрувати логіку повторного відправлення повідомлень в інші частини системи. Якщо повідомлення не вдалося відправити з першого разу, сервіс допомагає автоматично повторити спробу на основі визначеної політики, тим самим підвищуючи надійність доставки повідомлень у системі.

Лістинг коду сервісу представлено в додатку Ж.

5.5 Реєстрація компонентів системи

У процесі створення масштабованих і легко підтримуваних застосунків важливу роль відіграє архітектурний компонент, знаний як "Inversion of Control" (IoC) контейнер, який відповідає за реєстрацію залежностей та їхнє впровадження у різні частини програми. В .NET екосистемі, IoC контейнер зазвичай реалізується

за допомогою класу *IServiceCollection*, який дозволяє конфігурувати та реєструвати сервіси, що можуть бути запитані в інших частинах застосунку.

Розширюючи *IServiceCollection* за допомогою класу *ServiceCollectionExtensions*, створюється централізований механізм для реєстрації компонентів, які використовуються в системі асинхронного спілкування.

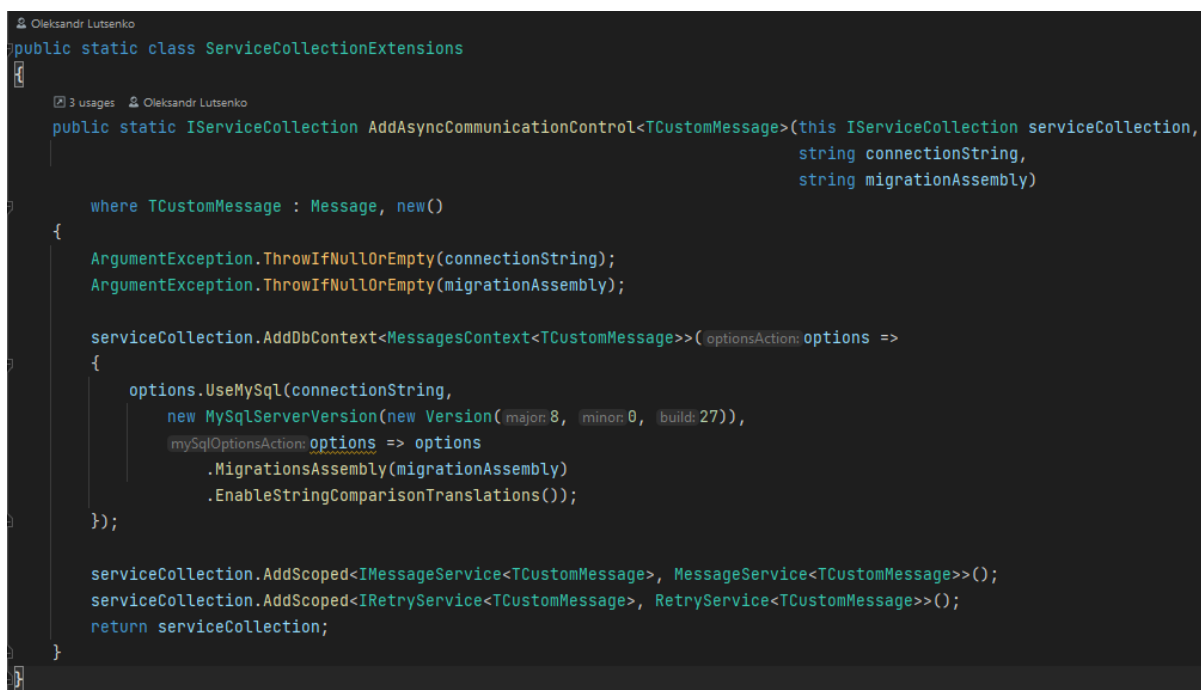
Метод *AddAsyncCommunicationControl* служить для реєстрації необхідних сервісів та налаштувань контексту бази даних. Цей метод використовує загальний тип *TCustomMessage*, який повинен походити від базового класу *Message*.

Ось основні кроки, які виконує метод *AddAsyncCommunicationControl*:

- перевірка параметрів: Метод першим ділом перевіряє, чи не пусті строки *connectionString* та *migrationAssembly*, які є необхідними для конфігурації бази даних і міграцій;
- конфігурація контексту бази даних: За допомогою *AddDbContext*, сервіс реєструє контекст *MessagesContext<TCustomMessage>*, який конфігурується для використання MySQL з вказаним рядком підключення та асамблеєю для міграцій;
- реєстрація сервісів: Метод додає до контейнера залежностей сервіси *IMessageService<TCustomMessage>* та *IRetryService<TCustomMessage>*, вказуючи їхні реалізації. Це забезпечує те, що коли будь-який компонент у системі запитує ці інтерфейси, ІоС контейнер зможе надати відповідні реалізації;
- завершення реєстрації: На завершення, *IServiceCollection* повертається, що дозволяє подальше ланцюгове додавання залежностей, якщо це необхідно;

Цей підхід не лише спрощує процес реєстрації та управління залежностями в .NET застосунках, але й сприяє більшій гнучкості та тестованості компонентів системи, оскільки сервіси можна легко замінити мок-об'єктами або іншими реалізаціями під час тестування.

Лістинг коду реєстрації компонентів відображено на рисунку 5.16.



```

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddAsyncCommunicationControl<TCustomMessage>(this IServiceCollection serviceCollection,
        string connectionString,
        string migrationAssembly)
    {
        where TCustomMessage : Message, new()
        {
            ArgumentException.ThrowIfNullOrEmpty(connectionString);
            ArgumentException.ThrowIfNullOrEmpty(migrationAssembly);

            serviceCollection.AddDbContext<MessagesContext<TCustomMessage>>((optionsAction: options =>
            {
                options.UseMySQL(connectionString,
                    new MySQLServerVersion(new Version(major: 8, minor: 0, build: 27)),
                    mySqlOptionsAction: options => options
                        .MigrationsAssembly(migrationAssembly)
                        .EnableStringComparisonTranslations());
            }));

            serviceCollection.AddScoped<IMessageService<TCustomMessage>, MessageService<TCustomMessage>>();
            serviceCollection.AddScoped<IRetryService<TCustomMessage>, RetryService<TCustomMessage>>();
            return serviceCollection;
        }
    }
}

```

Рисунок 5.16 – Реєстрації компонентів системи

Схема структури системи, в тому числі з компонентами прототипу, відображена в додатку Д.

Діаграма компонентів системи з компонентами прототипу застосунку, який використовує побудовану систему, відображена в додатку Е.

Діаграма послідовності відображена в додатку Є.

Висновок до розділу 5

У цьому розділі представлена детальна розробка системи управління асинхронними повідомленнями, яка включає створення сутностей, інтеграцію з базою даних та реалізацію механізму повторного відправлення повідомлень. Розробка системи базується на ефективному використанні політик повторних спроб та гнучкому управлінні повідомленнями.

Основна увага приділяється створенню надійної та масштабованої архітектури, що сприяє вдосконаленню комунікації в системах з використанням мікросервісної архітектури. Оптимізація процесів та реалізація гнучкої логіки

повторної відправки відіграють ключову роль у підвищенні ефективності та надійності системи.

6 ПУБЛІКАЦІЯ СИСТЕМИ

Для поширення системи для користувачів вона буде опублікована до NuGet - популярного менеджера пакетів для платформи розробки .NET. Він використовується для керування залежностями в проектах на платформі .NET і спрощує процес встановлення, оновлення та видалення бібліотек та інших компонентів, які використовуються в програмному забезпеченні.

Для цього буде згенеровано *.nupkg* файл, використовуючи технології IDE розробки JetBrains Rider:

- обрати потрібний проект - *AsyncCommunicationControl*;
- перейти в *Properties - Nuget*;
- вказати, що *.nupkg* файл повинен генеруватись при побудові проекту;
- вказати назву, опис та автора пакету;

Рисунок 6.1 репрезентує вікно для налаштування для генерування *.nupkg* файлу.

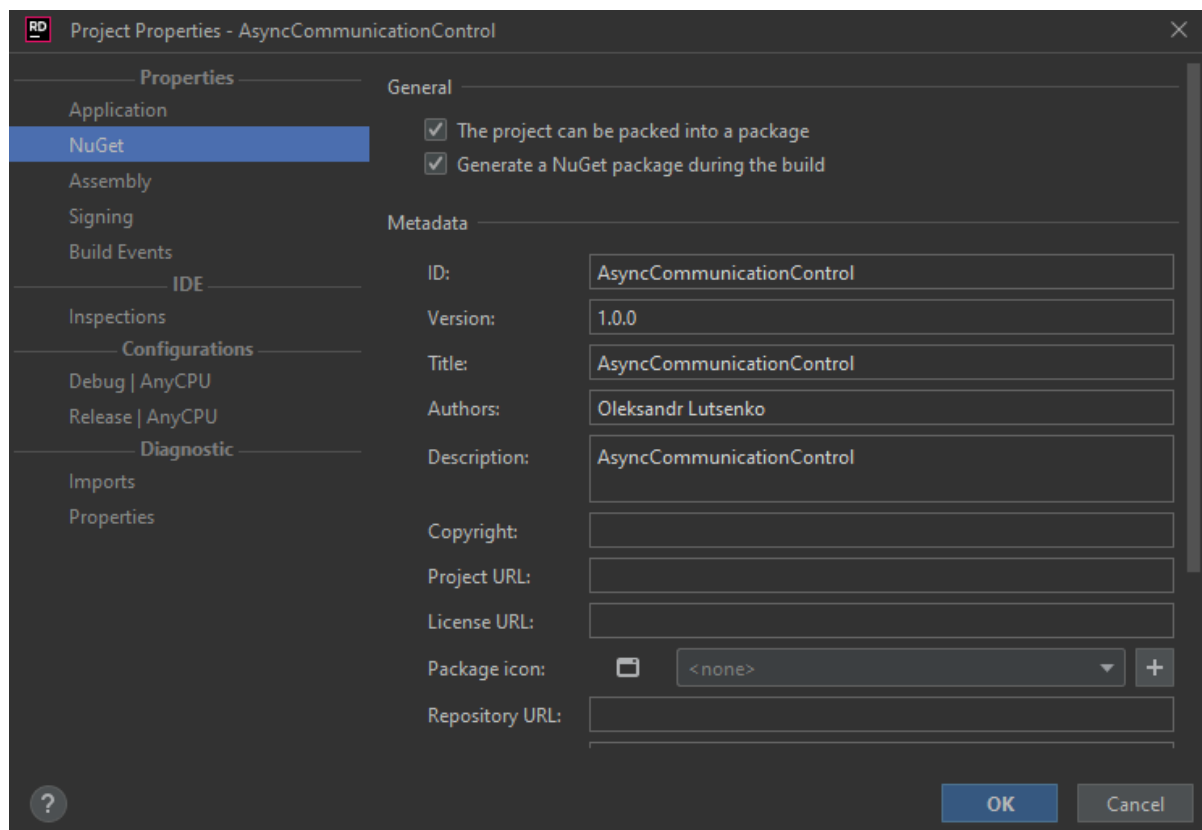


Рисунок 6.1 – Налаштування для генерування *.nupkg* файлу.

Далі буде побудований проект, та *.nupkg* файлу з'явиться в Debug директорії.

Після цього потрібно створити API ключ для публікації пакету до менеджера пакетів Nuget. Для цього потрібно:

- перейти на офіційний сайт <https://www.nuget.org>;
- авторизуватись в аккаунт;
- перейти до сторінки API Keys, натиснути на Create та заповнити всі потрібні поля;

Рисунок 6.2 відображає налаштування для генерування API ключа Nuget.

– Create

The screenshot shows the 'Create' API Key form in the NuGet.org interface. The form is titled 'Create' and contains several sections:

- Key Name ***: A text input field containing 'AsyncCommunicationControl'.
- Expires In**: A dropdown menu showing '365 days'.
- Package Owner ***: A dropdown menu showing 'roomey12'.
- Select Scopes**: A section with three radio buttons: 'Push' (checked), 'Push new packages and package versions', and 'Push only new package versions'. There is also an unchecked checkbox for 'Unlist package'.
- Select Packages**: A section with a text input field for 'Glob Pattern' containing an asterisk (*). Below the input field is a note: 'To select which packages to associate with a key, use a glob pattern, select individual packages, or both.' and a tooltip: 'A glob pattern allows you to replace any sequence of'.

Рисунок 6.2 – Налаштування для генерування API ключа Nuget.

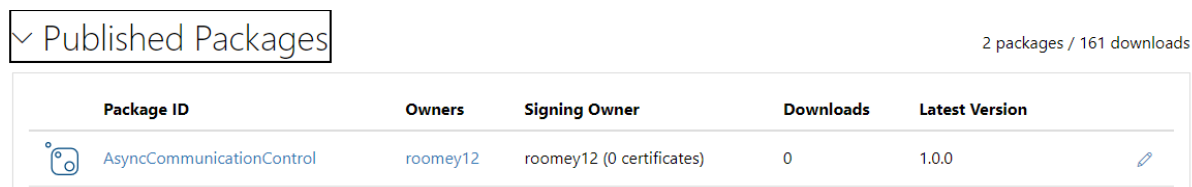
Після цього ключ буде створений і пакет можна публікувати.

Для цього потрібно відкрити термінал в директорії *.nupkg* файлу та використати команду -

```
dotnet nuget push <Назва nupkg файлу>.nupkg --api-key <API ключ> --source
https://api.nuget.org/v3/index.json
```

Після використання цієї команди пакет був опублікований, далі треба зачекати декілька хвилин для індексації та інших задач та пакет стає доступним для завантаження.

Рисунок 6.3 відображає готовий до використання пакет.
AsyncCommunicationControl.



The screenshot shows the NuGet Package Manager interface. At the top, there is a search bar with the text 'Published Packages' and a dropdown arrow. To the right of the search bar, it says '2 packages / 161 downloads'. Below the search bar is a table with the following columns: Package ID, Owners, Signing Owner, Downloads, and Latest Version. The table contains one row for the package 'AsyncCommunicationControl'.

Package ID	Owners	Signing Owner	Downloads	Latest Version
 AsyncCommunicationControl	roomey12	roomey12 (0 certificates)	0	1.0.0

Рисунок 6.3 – Готовий до використання пакет AsyncCommunicationControl.

Діаграма публікації системи відображена в додатку Г.

Висновок до розділу 6

У цьому розділі детально описано процес публікації розробленої системи на NuGet, що є ключовим етапом для забезпечення доступності системи для широкого кола користувачів.

Процес включає генерацію .nupkg файлу за допомогою JetBrains Rider, створення API-ключа на сайті NuGet, і публікацію пакету через командний рядок. Ці дії дозволяють опублікувати пакет на платформі, де він стає доступним для інших розробників. Така публікація сприяє поширенню та використанню розробленої системи, забезпечуючи її інтеграцію в різноманітні проекти та середовища.

7 ІНТЕГРАЦІЯ СИСТЕМИ КЕРУВАННЯ ПОВІДОМЛЕННЯМИ

Для інтеграції системи буде створено простий застосунок, ціль якого надіслати повідомлення до консьюмера.

Для наближення до робочих реалій застосунків буде реалізовано правильно, не дивлячись на те що він виконує просту задачу. Застосунок буде побудований на популярній архітектурі - Clean Architecture.

Clean Architecture [22] — це концепція у програмній інженерії, яка полягає в організації коду таким чином, щоб він був незалежним від фреймворків, UI, бази даних або будь-яких інших зовнішніх агентів. Головна ідея полягає в тому, щоб розробляти систему за принципом, де бізнес-логіка або доменний шар є центральним, а всі інші зовнішні шари навколо нього змінюються і адаптуються, не впливаючи на бізнес-правила.

Clean Architecture складається з таких основних рівнів, які перераховані нижче.

Domain [23]: Цей рівень формує ядро архітектури і включає сутності, які є високорівневими правилами бізнесу та бізнес-логікою. Вони представляють основні концепції, на яких базується система, і мають залишатися стабільними, незалежно від змін у зовнішніх шарах.

Application [24]: Шар, де розміщуються випадки використання, які оркеструють потік даних між зовнішніми шарами та доменом, реалізуючи конкретні функції, необхідні для кінцевих користувачів.

Infrastructure: На цьому рівні розміщуються елементи, які безпосередньо взаємодіють із зовнішнім світом, наприклад, з базами даних, файловими системами, веб-сервісами та брокерами повідомлень. Цей шар реалізує деталі, необхідні для виконання випадків використання.

Presentation: Цей рівень відповідає за відображення даних користувачу та інтерпретацію його дій. Це може бути веб-інтерфейс, API чи інший тип користувацького інтерфейсу.

Описані рівні відображені на рисунку 7.1.

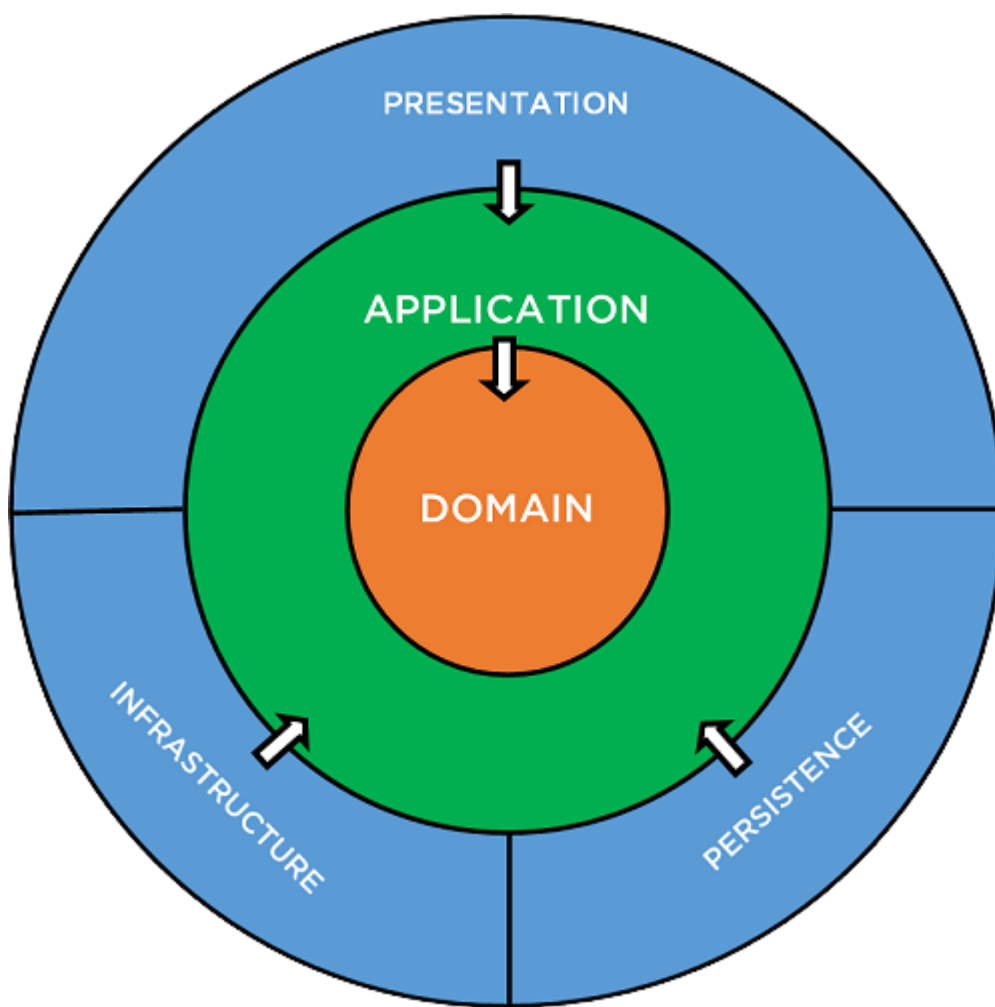


Рисунок 7.1 – Рівні Clean Architecture.

Розробка простого застосунку для відправлення повідомлень до консьюмера, відповідно до цих рівнів, дозволить нам забезпечити правильну організацію коду і легкість управління залежностями. Наш застосунок зможе ефективно комунікувати з консьюмером, при цьому зберігаючи гнучкість для подальших розширень та адаптації до змінних вимог бізнесу.

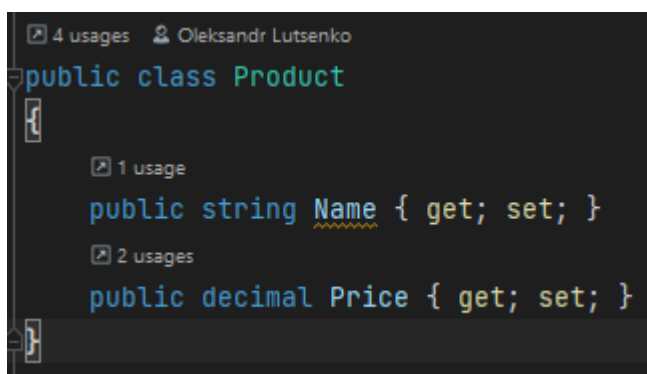
Схема структури системи відображена в додатку Д.

Діаграма компонентів відображена в додатку Е.

Діаграма послідовності відображена в додатку Є.

7.1 Domain рівень

Так як застосунок, що розробляється, є простим та існує для демонстрації розробленої системи, нам не потрібно багато логіки, саме тому був реалізований лише 1 клас - модель *Product*, з двома полями “Назва” та “Ціна”. Цей клас відображений на рисунку 7.2.



```
4 usages Oleksandr Lutsenko
public class Product
{
    1 usage
    public string Name { get; set; }
    2 usages
    public decimal Price { get; set; }
}
```

Рисунок 7.2 – Клас *Product*.

7.2 Infrastructure рівень

На цьому рівні знаходяться класи для роботи з зовнішніми сервісами - такими як бази даних, брокери повідомлень чи інше.

Отже, на цьому рівні повинна бути описана модель, яка наслідується з класу *Message* з розробленої у розділі 5 системи.

Буде імплементований свій тип *MyMessage* на основі вище вказаного типу *Message*, додавши нове поле *Description*, для демонстрації того, що користувачі системи можуть додавати свої поля, та вводити додаткову логіку відносно цих полей.

Також, будуть реалізовані два конструктори - один за замовчуванням, та один, який має в параметрах вказане раніше поле *Description*.

Описаний тип відображено на рисунку 7.3.



```

22 usages Oleksandr Lutsenko
public class MyMessage : Message
{
    1 usage
    public string? Description { get; set; }

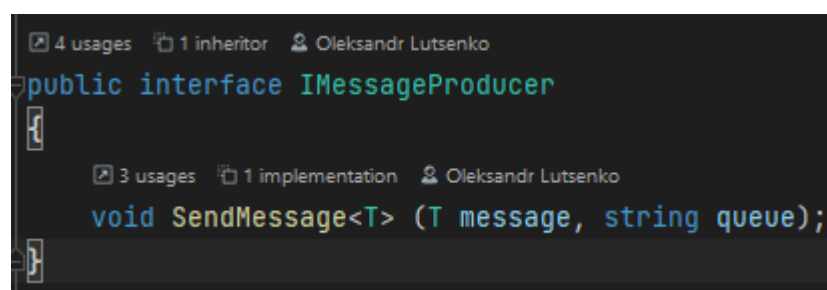
    1 usage Oleksandr Lutsenko
    public MyMessage(string description)
    {
        Description = description;
    }

    8 usages Oleksandr Lutsenko
    public MyMessage()
    {
    }
}

```

Рисунок 7.3 – Клас *MyMessage*, наслідований від типу *Message*.

Далі потрібно описати сервіс, для відправлення повідомлень до брокерів, для цього спочатку буде об'явлений інтерфейс *IMessageProducer*, що відображено на рисунку 7.4.



```

4 usages 1 inheritor Oleksandr Lutsenko
public interface IMessageProducer
{
    3 usages 1 implementation Oleksandr Lutsenko
    void SendMessage<T> (T message, string queue);
}

```

Рисунок 7.4 – Інтерфейс *IMessageProducer*.

Цей інтерфейс визначає один метод:

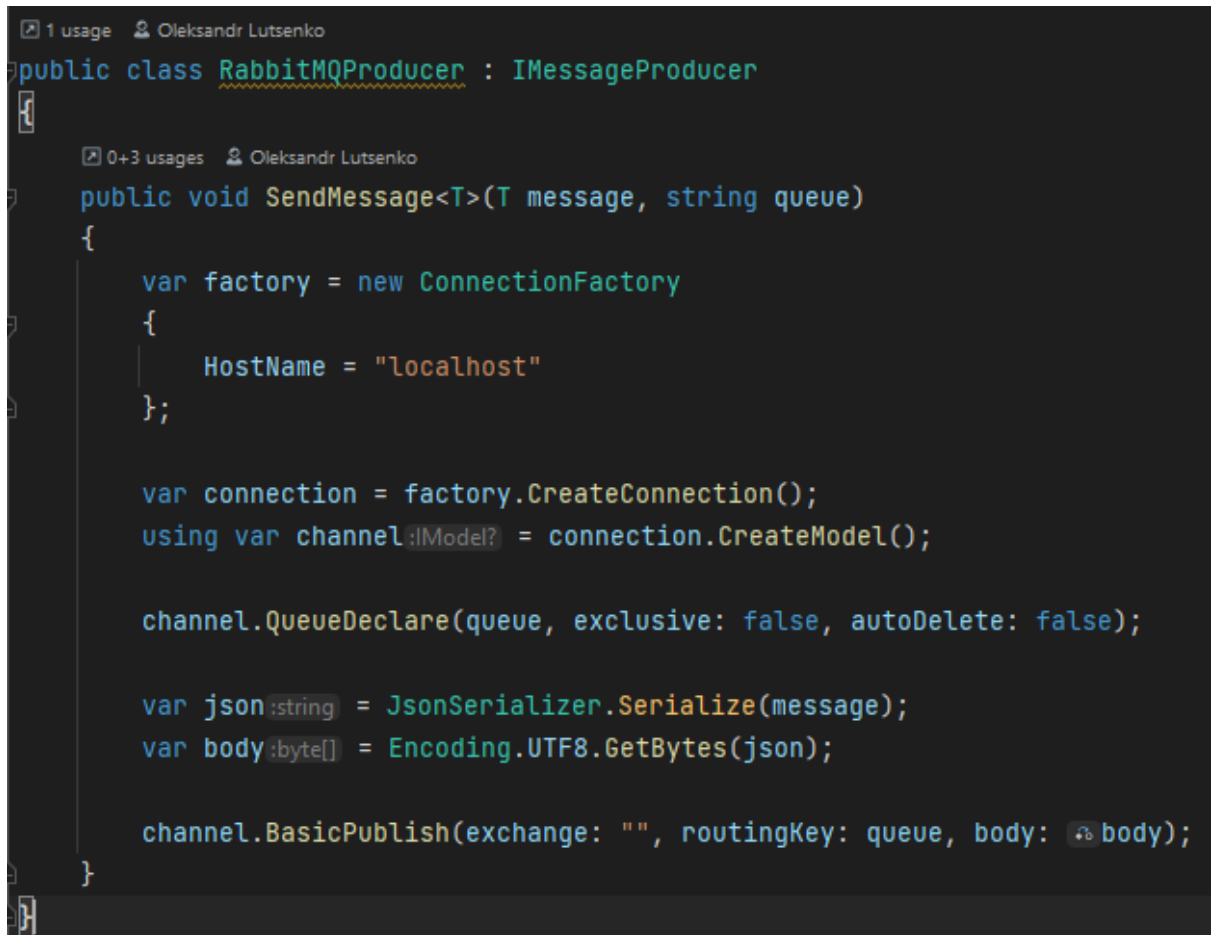
SendMessage<T>(T message, string queue): Цей метод призначений для відправлення повідомлення в задану чергу. Тип повідомлення є загальним (*T*), що означає, що метод може приймати будь-який тип даних як повідомлення.

Далі буде описано імплементацію цього інтерфейсу.

В випадку магістерської дисертації, як описувалось в розділі 4.3, як брокер повідомлень буде використаний RabbitMQ.

Клас *RabbitMQProducer* реалізує інтерфейс *IMessageProducer* та використовує RabbitMQ для відправлення повідомлень. Буде розглянуто його основні компоненти, які можна побачити на рисунку 7.5:

- конструктор з'єднання (ConnectionFactory): Створює новий екземпляр ConnectionFactory, який використовується для створення з'єднання з RabbitMQ сервером. У цьому випадку використовується локальний хост ("localhost");
- створення з'єднання (CreateConnection): Використовуючи ConnectionFactory, клас встановлює з'єднання з RabbitMQ сервером;
- створення каналу (CreateModel): Після встановлення з'єднання створюється канал, через який будуть відправлятися повідомлення;
- оголошення черги (QueueDeclare): Визначається черга, до якої будуть відправлятися повідомлення. Параметри вказують, що черга не є ексклюзивною та не буде автоматично видалятися;
- серіалізація повідомлення (JsonSerializer.Serialize): Повідомлення серіалізується у формат JSON;
- кодування та відправлення (BasicPublish): Серіалізоване повідомлення кодується в UTF-8 та відправляється у вказану чергу за допомогою RabbitMQ;



```

1 usage  Oleksandr Lutsenko
public class RabbitMQProducer : IMessageProducer
{
    0+3 usages  Oleksandr Lutsenko
    public void SendMessage<T>(T message, string queue)
    {
        var factory = new ConnectionFactory
        {
            HostName = "localhost"
        };

        var connection = factory.CreateConnection();
        using var channel : IModel? = connection.CreateModel();

        channel.QueueDeclare(queue, exclusive: false, autoDelete: false);

        var json : string = JsonSerializer.Serialize(message);
        var body : byte[] = Encoding.UTF8.GetBytes(json);

        channel.BasicPublish(exchange: "", routingKey: queue, body: body);
    }
}

```

Рисунок 7.5 – Клас RabbitMQProducer.

Цей клас та інтерфейс спрощують відправлення повідомлень через RabbitMQ, надаючи абстракцію для відправлення будь-якого типу даних.

Після цього буде описано сервіс обгортку над *IMessageProducer*, який надсилає повідомлення до черги та записує його до бази, для моніторингу та майбутніх дій.

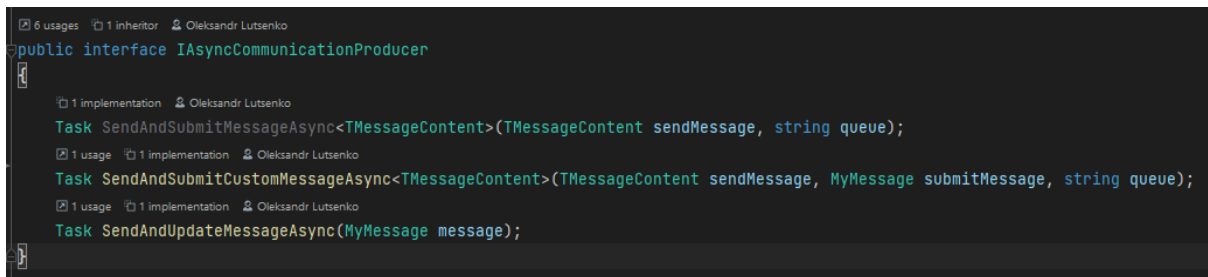
Для початку буде описано інтерфейс *IAsyncCommunicationProducer*, що відображений на рисунку 7.6.

Цей інтерфейс визначає наступні методи:

- `SendAndSubmitMessageAsync`: Відправляє повідомлення до черги. В тому числі записує його до бази даних. Вхідні параметри включають зміст повідомлення (*TMessageContent*) та назву черги (*string queue*);
- `SendAndSubmitCustomMessageAsync`: Відправляє повідомлення до черги. Надає можливість користувачу вказати повідомлення для запису в базу даних

на свій розсуд. Вхідні параметри включають основне повідомлення для відправки (TMessageContent), налаштоване повідомлення для запису в базу даних (MyMessage) та назву черги (string queue);

- SendAndUpdateMessageAsync: Оновлює вже існуюче повідомлення та відправляє його до черги. Вхідні параметри включають повідомлення для оновлення та відправлення (MyMessage);



```

6 usages 1 inheritor Oleksandr Lutsenko
public interface IAsyncCommunicationProducer
{
    1 implementation Oleksandr Lutsenko
    Task SendAndSubmitMessageAsync<TMessageContent>(TMessageContent sendMessage, string queue);
    1 usage 1 implementation Oleksandr Lutsenko
    Task SendAndSubmitCustomMessageAsync<TMessageContent>(TMessageContent sendMessage, MyMessage submitMessage, string queue);
    1 usage 1 implementation Oleksandr Lutsenko
    Task SendAndUpdateMessageAsync(MyMessage message);
}
  
```

Рисунок 7.6 – Інтерфейс *IAsyncCommunicationProducer*.

Далі буде описано клас *AsyncCommunicationProducer*. Цей клас реалізує інтерфейс *IAsyncCommunicationProducer* та окрім описаних раніше методів має наступні компоненти:

- залежності: Клас залежить від *IMessageProducer* для відправлення повідомлень та від *IServiceProvider* для створення та управління залежністю *IMessageService*, яка потрібна для маніпуляцій з даними повідомлення;
- конструктор: Ініціалізує залежності;

```

1 usage Oleksandr Lutsenko *
public class AsyncCommunicationProducer : IAsyncCommunicationProducer
{
    private readonly IMessageProducer _messageProducer;
    private readonly IServiceProvider _serviceProvider;

    Oleksandr Lutsenko
    public AsyncCommunicationProducer(IMessageProducer messageProducer, IServiceProvider serviceProvider)
    {
        _messageProducer = messageProducer;
        _serviceProvider = serviceProvider;
    }

    0+1 usages Oleksandr Lutsenko
    public async Task SendAndUpdateMessageAsync(MyMessage message)
    {
        using var scope = _serviceProvider.CreateScope();
        var messageService = scope.ServiceProvider.GetRequiredService<IMessageService<MyMessage>>();
        await messageService.UpdateMessageAsync(message);
        _messageProducer.SendMessage(message, message.Queue);
    }

    Oleksandr Lutsenko
    public async Task SendAndSubmitMessageAsync<TMessageContent>(TMessageContent messageContent, string queue)
    {
        using var scope = _serviceProvider.CreateScope();
        var messageService = scope.ServiceProvider.GetRequiredService<IMessageService<MyMessage>>();
        var message = await messageService.CreateAndSubmitMessageAsync(messageContent, queue);
        _messageProducer.SendMessage(message, queue);
    }

    0+1 usages Oleksandr Lutsenko *
    public async Task SendAndSubmitCustomMessageAsync<TMessageContent>(TMessageContent messageContent,
                                                                           MyMessage customMessage,
                                                                           string queue)
    {
        using var scope = _serviceProvider.CreateScope();
        var messageService = scope.ServiceProvider.GetRequiredService<IMessageService<MyMessage>>();
        customMessage.Fill(messageContent, queue);
        await messageService.SubmitMessageAsync(customMessage);
        _messageProducer.SendMessage(customMessage, queue);
    }
}

```

Рисунок 7.7 - Клас *AsyncCommunicationProducer*.

Клас *AsyncCommunicationProducer*, відображений на рисунку 7.7, забезпечує гнучкість у відправці та обробці повідомлень, інтегруючись з системою черг та використовуючи сервіси для маніпуляцій повідомленнями в базі даних. Це дає можливість розробляти складні асинхронні рішення для обміну повідомленнями в розподілених системах.

Далі потрібно описати клас *ServiceCollectionExtensions* з розширюючим методом для *IServiceCollection* в .NET, який використовується для реєстрації залежностей у контейнері *DI* (Dependency Injection). Цей клас відображено на рисунку 7.8.

Буде розглянуто його деталі:

- призначення: Метод *AddRabbitMQ* призначений для додавання специфічних для *RabbitMQ* сервісів до *IServiceCollection*, що дозволяє використовувати ці сервіси в інших частинах застосунку через механізм впровадження залежностей;
- параметри: *serviceCollection*: Це екземпляр *IServiceCollection*, до якого додаються сервіси;
- функціональність:

serviceCollection.AddSingleton<IMessageProducer, RabbitMQProducer>():

Реєструє *RabbitMQProducer* як однокземплярний (singleton) сервіс для інтерфейсу *IMessageProducer*. Це означає, що при кожному запиті на *IMessageProducer*, *DI* контейнер надаватиме той самий екземпляр *RabbitMQProducer*.

serviceCollection.AddSingleton<IAsyncCommunicationProducer,

AsyncCommunicationProducer>(): Аналогічно, реєструє *AsyncCommunicationProducer* як однокземплярний сервіс для інтерфейсу *IAsyncCommunicationProducer*;

- повернення даних: Метод повертає *IServiceCollection*, що дозволяє використовувати ланцюжкові виклики при реєстрації сервісів;



```
public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddRabbitMQ(this IServiceCollection serviceCollection)
    {
        serviceCollection.AddSingleton<IMessageProducer, RabbitMQProducer>();
        serviceCollection.AddSingleton<IAsyncCommunicationProducer, AsyncCommunicationProducer>();
        return serviceCollection;
    }
}
```

Рисунок 7.8 - Статичний клас *ServiceCollectionExtensions* з методом *AddRabbitMQ(this IServiceCollection serviceCollection)*.

Цей метод є частиною паттерну "Fluent Interface", який використовується у багатьох .NET бібліотеках для спрощення конфігурації. Завдяки реєстрації сервісів у контейнері DI, розробники можуть легко впроваджувати залежності в свої компоненти та сервіси, що підвищує модульність і тестованість коду.

7.3 API рівень

На цьому рівні буде об'явлений контролер, який має ендпоінт для відправлення повідомлення до черги в залежності від переданого тіла запиту.

Клас *ProductController* є контролером у веб-застосунку, розробленому з використанням ASP.NET Core. Нижче наведений опис цього класу:

Атрибути класу:

- *[ApiController]*: Атрибут, який вказує, що клас є контролером API. Він дозволяє використовувати декілька функцій ASP.NET Core, таких як автоматичне виведення моделі запитів та валідація;
- *[Route("[controller]")]*: Визначає маршрут для контролера. *[controller]* автоматично замінюється назвою контролера без суфікса "Controller", тобто "Product" у цьому випадку;

Властивості та поля:

- *private readonly IAsyncCommunicationProducer _asyncCommunicationProducer*: Це поле використовується для зберігання посилання на екземпляр сервісу *IAsyncCommunicationProducer*;
- *private const string ProductsQueue = "products"*: Константа, що визначає назву черги, яка використовується для повідомлень про продукти;

Конструктор

- *public ProductController(IAsyncCommunicationProducer asyncCommunicationProducer): Конструктор, який ін'єктує залежність IAyncCommunicationProducer;*

Методи

[HttpPost]: Атрибут, що вказує, що метод CreateProduct обробляє HTTP POST запити.

- *public async Task<IActionResult> CreateProduct(Product product):*
Асинхронний метод, який приймає об'єкт Product як параметр та відповідає за обробку створення нового продукту. Він використовує *_asyncCommunicationProducer* для надсилання повідомлення в чергу *ProductsQueue*. Після виконання операції, метод повертає HTTP відповідь *Ok()*, що відповідає статусу 200. Цей метод має атрибут *[HttpPost]*, що вказує, що метод *CreateProduct* обробляє HTTP POST запити.
Також, варто зазначити, що тут в конструктор типу *MyMessage* передається строка “fromProductController”, яка стане значенням поля *Description*;

Цей клас є типовим прикладом контролера у веб-API застосунку, розробленому з використанням ASP.NET Core. Він демонструє, як можна інтегрувати асинхронну комунікацію та роботу з чергами повідомлень у веб-застосунку. Лістинг коду контролера наведено на рисунку 7.9.

```

[ApiController]
[Route("[controller]")]
public class ProductController : ControllerBase
{
    private readonly IAsyncCommunicationProducer _asyncCommunicationProducer;
    private const string ProductsQueue = "products";

    public ProductController(IAsyncCommunicationProducer asyncCommunicationProducer)
    {
        _asyncCommunicationProducer = asyncCommunicationProducer;
    }

    [HttpPost]
    public async Task<IActionResult> CreateProduct(Product product)
    {
        var message = new MyMessage("fromProductController");
        await _asyncCommunicationProducer.SendAndSubmitCustomMessageAsync(product, message, ProductsQueue);
        return Ok();
    }
}

```

Рисунок 7.9 - Клас *ProductController*.

Далі буде описано клас *Program*, який в ASP.NET Core використовується для конфігурації та запуску додатку. Опис цього класу:

Створення WebApplication Builder:

- `var builder = WebApplication.CreateBuilder(args);` Цей рядок ініціює створення будівника веб-додатку, який використовується для налаштування сервісів та конфігурації;

Конфігурація Сервісів:

- `builder.Services.AddControllers();` Додає контролери до сервісів додатку;
- `builder.Services.AddEndpointsApiExplorer()` та `builder.Services.AddSwaggerGen();` Конфігурує Swagger для генерації документації API;
- `builder.Services.AddAsyncCommunicationControl<MyMessage>();` Додає та конфігурує сервіси, пов'язані з асинхронним управлінням повідомленнями, описані у розділі 5;
- `builder.Services.AddRabbitMQ();` Викликає розширюючий метод, об'явлений в розділі 7.2, для додавання *RabbitMQ* як системи повідомлень;

Побудова та Конфігурація Додатку:

- `var app = builder.Build()`: Створює екземпляр веб-додатку з налаштуваннями, визначеними раніше.

Конфігурація HTTP Request Pipeline:

- `app.UseSwagger()` та `app.UseSwaggerUI()`: Включає Swagger UI в режимі розробки для тестування API;
- `app.UseHttpsRedirection()`: Примушує HTTP запити перенаправлятися на HTTPS;
- `app.MapControllers()`: Вказує на використання контролерів;

Запуск Додатку:

- `app.Run()`: Запускає додаток та слухає вхідні HTTP запити;

Цей клас є вхідною точкою для ASP.NET Core веб-додатку та використовується для налаштування та запуску веб-сервера, сервісів, конфігурації маршрутів та міدلварів. Лістинг цього класу представлено на рисунку 7.10.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddAsyncCommunicationControl<MyMessage>(
    connectionString: builder.Configuration["Messages:ConnectionString"],
    migrationAssembly: builder.Configuration["Messages:MigrationAssembly"]);
builder.Services.AddRabbitMQ();
var app:WebApplication = builder.Build();

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

app.UseHttpsRedirection();

app.MapControllers();

app.Run();
```

Рисунок 7.10 - Клас *Program*.

Крім того, на цьому рівні знаходяться файли міграцій бази даних, які використовуються для оновлення схеми бази.

Також, цей рівень має файл конфігурації *appsettings.json*, відображений на рисунку 7.11, який містить connection string для підключення до бази даних, назву збірки для побудови міграцій, та конфігурацію за замовчуванням для роботи логів.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Messages": {
    "ConnectionString": "Server=localhost;Database=AsyncCommunicationControl;User=
    "MigrationAssembly": "MyWebApp"
  }
}
```

Рисунок 7.11 - Файл конфігурації *appsettings.json*.

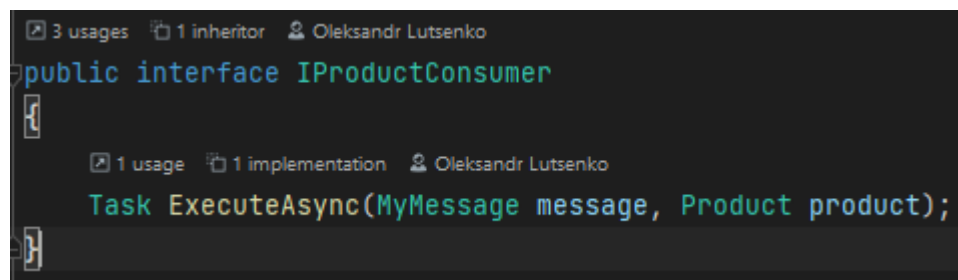
7.4 Message consumer

Далі буде описано застосунок для слухання та обробки повідомлень асинхронної комунікації.

Для початку буде описаний інтерфейс *IProductConsumer*. Це інтерфейс, який визначає контракт для класів, що обробляють повідомлення про продукти.

Метод:

- *Task ExecuteAsync(MyMessage message, Product product)*: Асинхронний метод, який приймає повідомлення (*MyMessage*) та продукт (*Product*). Задача цього методу - виконати певну логіку обробки повідомлення про продукт; Лістинг цього інтерфейсу наведено на рисунку 7.12.



```

3 usages 1 inheritor Oleksandr Lutsenko
public interface IProductConsumer
{
    1 usage 1 implementation Oleksandr Lutsenko
    Task ExecuteAsync(MyMessage message, Product product);
}

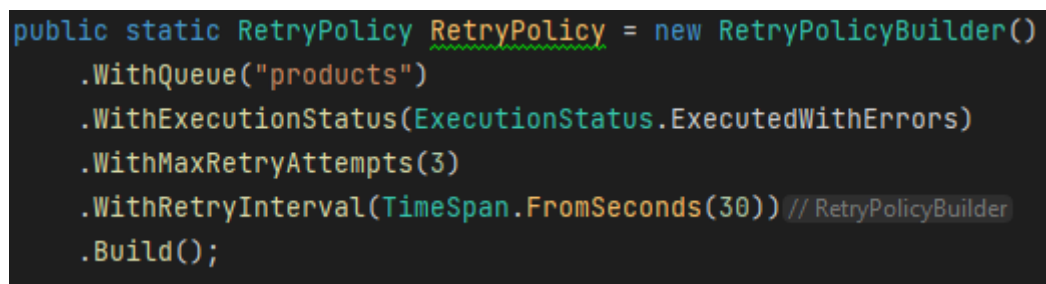
```

Рисунок 7.12 - Інтерфейс *IProductConsumer*.

ProductConsumer - це клас, що реалізує інтерфейс *IProductConsumer*. Він використовує внедрення залежностей (*IServiceProvider*) для доступу до інших сервісів.

Поля:

- *IServiceProvider _serviceProvider*: Використовується для створення області залежностей та доступу до інших сервісів;
- *RetryPolicy*: Статичне поле, яке визначає політику повторних спроб для обробки повідомлень. Ця політика включає інформацію про чергу, з якої повинно братися повторне повідомлення, статус виконання, який повинно мати повідомлення для її повторної обробки, максимальну кількість спроб для повторного обробки повідомлення та інтервал між спробами обробки;



```

public static RetryPolicy RetryPolicy = new RetryPolicyBuilder()
    .WithQueue("products")
    .WithExecutionStatus(ExecutionStatus.ExecutedWithErrors)
    .WithMaxRetryAttempts(3)
    .WithRetryInterval(TimeSpan.FromSeconds(30)) // RetryPolicyBuilder
    .Build();

```

Рисунок 7.13 - Опис політики повторних спроб для *ProductConsumer*.

Конструктор:

- *ProductConsumer(IServiceProvider serviceProvider)*: Ініціалізує клас, встановлюючи залежність *IServiceProvider*;

Метод *ExecuteAsync*:

- цей метод виконує основну логіку обробки продукту та повідомлення. Він використовує *IServiceProvider* для доступу до *IMessageService<MyMessage>* та виконує наступні кроки:
 - виводить інформацію про продукт;
 - перевіряє ціну продукту. Якщо ціна вища за певне значення, викидає виключення;
 - при успішній обробці оновлює статус повідомлення на *SuccessfullyExecuted*;
 - в разі помилки оновлює статус повідомлення на *ExecutedWithErrors*;

Як видно на рисунку 7.13, в політиці повторних спроб указаний статус *ExecutedWithErrors*, це означає, що тільки повідомлення з таким статусом будуть повторно оброблені, саме тому в разі помилки при обробці повідомлення, вказується цей статус, а в випадку успішної обробки вказується статус *SuccessfullyExecuted*.

Цей клас відповідає за обробку повідомлень про продукти, з можливістю обробки помилок та повторних спроб. Лістинг цього класу вказано на рисунку 7.14.

```

public class ProductConsumer : IProductConsumer
{
    private readonly IServiceProvider _serviceProvider;

    public static RetryPolicy RetryPolicy = new RetryPolicyBuilder()
        .WithQueue("products")
        .WithExecutionStatus(ExecutionStatus.ExecutedWithErrors)
        .WithMaxRetryAttempts(3)
        .WithRetryInterval(TimeSpan.FromSeconds(30)) // RetryPolicyBuilder
        .Build();

    Oleksandr Lutsenko
    public ProductConsumer(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    0+1 usages Oleksandr Lutsenko
    public async Task ExecuteAsync(MyMessage message, Product product)
    {
        using var scope = _serviceProvider.CreateScope();
        var messageService = scope.ServiceProvider.GetRequiredService<IMessageService<MyMessage>>();
        try
        {
            Console.WriteLine($"Product: {product.Name}|{product.Price}");
            if (product.Price > 10)
            {
                throw new Exception(message: "wrong price");
            }
            message.Status = ExecutionStatus.SuccessfullyExecuted;
            await messageService.UpdateMessageAsync(message);
        }
        catch (Exception ex)
        {
            message.Status = ExecutionStatus.ExecutedWithErrors;
            try
            {
                await messageService.UpdateMessageAsync(message, RetryPolicy);
            }
            catch (Exception ex1)
            {
                Console.WriteLine(ex1.Message);
            }
        }
    }
}

```

Рисунок 7.14 - Клас *ProductConsumer*.

Далі буде описано клас *Program* цього застосунку.

Як було вказано раніше, клас *Program* є точкою входу в застосунок. Він відповідає за ініціалізацію конфігурації, налаштування залежностей, та в випадку

консьюмера повідомлень, відповідає за запуск основної логіки обробки повідомлень.

Методи:

Main():

- ініціалізує конфігурацію;
- налаштовує контейнер впровадження залежностей (*IServiceCollection*), додаючи сервіси, зокрема *IProductConsumer* та конфігурацію для асинхронної комунікації, з розділу 5;
- створює *IServiceProvider* та запускає основну логіку в методі *Run*;

Run(IServiceProvider serviceProvider):

- створює з'єднання з RabbitMQ, використовуючи *ConnectionFactory*;
- оголошує чергу "products" для обробки повідомлень;
- створює об'єкт *EventingBasicConsumer* для асинхронного споживання повідомлень з черги;
- підписується на подію *Received*, де відбувається обробка отриманих повідомлень:
 - десеріалізує повідомлення та вміст продукту;
 - викликає *ExecuteAsync* у *IProductConsumer* для обробки продукту;
 - обробляє можливі виключення та виводить повідомлення про помилку в консоль;
- оозпочинає споживання повідомлень з черги;
- запускає нескінченний цикл, що чекає на введення з клавіатури для завершення програми;

Цей клас реєструє компоненти системи та прослуховує асинхронні повідомлення. Його лістинг відображено на рисунку 7.15.

```

class Program
{
    Oleksandr Lutsenko
    static void Main()
    {
        var configuration = new ConfigurationBuilder()
            .AddJsonFile(path: "appsettings.json", optional: true)
            .AddEnvironmentVariables() // IConfigurationBuilder
            .Build();

        var serviceProvider = new ServiceCollection()
            .AddScoped<IProductConsumer, ProductConsumer>()
            .AddAsyncCommunicationControl<MyMessage>(
                connectionString: configuration["Messages:ConnectionString"],
                migrationAssembly: configuration["Messages:MigrationAssembly"]) // IServiceCollection
            .BuildServiceProvider();

        Run(serviceProvider);
    }

    1 usage Oleksandr Lutsenko
    static void Run(IServiceProvider serviceProvider)
    {
        var factory = new ConnectionFactory { HostName = "localhost" };
        using var connection = factory.CreateConnection();
        using var channel<Model?> = connection.CreateModel();
        channel.QueueDeclare("products", exclusive: false, autoDelete: false);

        var consumer = new EventingBasicConsumer(channel);
        consumer.Received += (model: object?, eventArgs) =>
        {
            try
            {
                var body: byte[] = eventArgs.Body.ToArray();
                var message = JsonSerializer.Deserialize<MyMessage>(* body);
                var product = JsonSerializer.Deserialize<Product>(message.Content);
                var productConsumer = serviceProvider.GetRequiredService<IProductConsumer>();
                productConsumer.ExecuteAsync(message, product);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        };
        channel.BasicConsume(queue: "products", autoAck: true, consumer: consumer);

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }
}

```

Рисунок 7.15 - Клас *Program* консьюмера повідомлень.

В тому числі, цей застосунок має такий самий файл конфігурації *appsettings.json*, як і API додаток.

7.5 Механізм повторного відправлення повідомлень

Далі буде описано механізм повторного відправлень повідомлень, основні методи цього механізму були реалізовані в пакеті розробленого в розділі 5, зараз буде описано їх використання.

Для початку буде описаний воркер `RetryWorker`:

`RetryWorker` - це клас, який успадковує клас `BackgroundService`. Це означає, що цей клас призначений для використання як фоновий сервіс, який може виконувати роботу в фоновому режимі під час роботи додатка. Задача цього класу обирати повідомлення, які потрібно повторного надіслати, з бази даних, та надсилати їх до брокера повідомлень.

Конструктор `RetryWorker` приймає кілька параметрів, які необхідні для його коректної роботи:

- *`ILogger<RetryWorker> logger`*: Логгер, який дозволяє реєструвати події та повідомлення під час роботи сервісу;
- *`IServiceProvider serviceProvider`*: Постачальник служб, який використовується для створення об'єктів та доступу до служб в іншому місці коду;
- *`IAsyncCommunicationProducer asyncCommunicationProducer`*: Інтерфейс описаний у розділі 7.2, використовується для відправки повідомлень та запису їх до бази даних;

Крім цього, конструктор створює список `_retryPolicies`, який містить об'єкт `RetryPolicy`. Він ініціалізується з політики `ProductConsumer.RetryPolicy`, описаний у розділі 7.4.

Метод `ExecuteAsync(CancellationToken stoppingToken)` є реалізацією методу з класу `BackgroundService`. Цей метод виконується в фоновому режимі. В цьому методі відбувається основна робота:

- виводиться інформаційне повідомлення про поточний час, щоб було зрозуміло коли воркер працює;
- викликається метод `ExecuteRetryAsync(_retryPolicies)`, який відповідає за спроби відправити повторне повідомлення;

- викликається метод `Task.Delay(5000, stoppingToken)`, який призупиняє виконання роботи на 5 секунд або до тих пір, поки не надійде сигнал для завершення (`stoppingToken`);

Це робиться для введення затримки між ітераціями циклу у фоновому робочому потоці. Важливо обрати збалансоване значення для такої затримки, і значення 5000 мілісекунд може бути прийнятним в даному контексті.

Нижче наведені обґрунтування для використання такої затримки:

Згладження навантаження: Використання невеликого проміжку між ітераціями циклу допомагає згладити навантаження на систему. Це особливо корисно, якщо фоновий процес виконує якусь роботу, яка може бути ресурсоємкою, наприклад запити до бази даних для отримання повідомлень.

Збалансована відзивчивість: Занадто велика затримка може зробити програму менш відзивчивою. Якщо фоновий метод повинен реагувати на події або виконувати певні дії з регулярністю, як в даному випадку, великий проміжок часу може призвести до погіршення відзивчивості додатка.

Втрата актуальності даних: Якщо фоновий сервіс занадто рідко оновлює дані або виконує певні операції, то це може призвести до втрати актуальності даних. Наприклад, якщо ви використовуєте фоновий сервіс для синхронізації даних і він викликається надто рідко, то користувач може побачити застарілі дані.

Неефективне використання ресурсів: Великий проміжок часу між викликами методу може призвести до неефективного використання ресурсів. Наприклад, якщо фоновий сервіс викликається раз на годину і весь цей час він очікує, то це може виглядати як надмірне споживання процесорного часу або пам'яті.

Використання надто малого часу, наприклад, 5 мілісекунд, в `Task.Delay(5, stoppingToken)` може мати кілька негативних наслідків, що перераховані нижче.

Велике навантаження на процесор: Занадто короткий інтервал затримки призведе до надмірного виконання ітерацій у фоновому циклі. Якщо цикл виконується настільки швидко, це може призвести до великого навантаження на

процесор, особливо на многоядерних системах, що може займати ресурси та призвести до підвищеного споживання електроенергії.

Можливе перевантаження системи: Використання надто малих затримок може призвести до перевантаження системи, особливо якщо фоновий сервіс виконується на декількох інстансів або одночасно з іншими фоновими задачами.

Велика активність в мережі: Якщо фоновий процес включає в себе мережеву активність, наприклад, відправку запитів на сервер чи до бази даних, використання надто малих затримок може призвести до великої кількості запитів на сервер за короткий проміжок часу, що може перевантажити мережу або сервер.

Велика кількість контекстних перемикань: Маленькі затримки можуть призводити до великої кількості контекстних перемикань між потоками, що витрачають дорогоцінні ресурси процесора та можуть призвести до збільшення накладних витрат.

Загалом, важливо обирати затримку, яка є збалансованою і відповідає потребам додатку.

Опис класа *RetryWorker*

Метод `ExecuteRetryAsync(IEnumerable<RetryPolicy> retryPolicies)` викликається з методу `ExecuteAsync` і відповідає за спроби відправити повідомлення з помилковими спробами. Він робить наступні дії:

- створює область видимості (scope) за допомогою *IServiceProvider*, щоб отримати доступ до сервісів;
- отримує сервіс *IRetryService<MyMessage>* зі створеної області видимості;
- отримує список повідомлень для повторної відправки, використовуючи інтерфейс з розділу 5;
- для кожного повідомлення виводиться інформаційне повідомлення про відправку і викликається метод `SendAndUpdateMessageAsync`, з інтерфейса *IAsyncCommunicationProducer*, описаного у розділі 7.2, для відправлення повідомлення;

Лістинг коду цього класу наведено на рисунку 7.16.

```

public class RetryWorker : BackgroundService
{
    private readonly ILogger<RetryWorker> _logger;
    private readonly IServiceProvider _serviceProvider;
    private readonly IAsyncCommunicationProducer _asyncCommunicationProducer;
    private readonly IEnumerable<RetryPolicy> _retryPolicies;

    [Oleksandr Lutsenko]
    public RetryWorker(ILogger<RetryWorker> logger,
        IServiceProvider serviceProvider,
        IAsyncCommunicationProducer asyncCommunicationProducer)
    {
        _logger = logger;
        _serviceProvider = serviceProvider;
        _asyncCommunicationProducer = asyncCommunicationProducer;
        _retryPolicies = new List<RetryPolicy> { ProductConsumer.RetryPolicy };
    }

    [Oleksandr Lutsenko]
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation(message: "Worker running at: {time}", DateTimeOffset.Now);
            await ExecuteRetryAsync(_retryPolicies);
            await Task.Delay(5000, stoppingToken);
        }
    }

    [1 usage] [Oleksandr Lutsenko]
    private async Task ExecuteRetryAsync(IEnumerable<RetryPolicy> retryPolicies)
    {
        using var scope = _serviceProvider.CreateScope();
        var retryService = scope.ServiceProvider.GetRequiredService<IRetryService<MyMessage>>();
        var retryMessages<List<MyMessage>> = (await retryService.GetRetryMessagesAsync(retryPolicies)).ToList();
        foreach (var retryMessage in retryMessages)
        {
            _logger.LogInformation(message: "Sending retry message: {retryMessageId} at {time}", params: args: retryMessage.Id, DateTimeOffset.Now);
            await _asyncCommunicationProducer.SendAndUpdateMessageAsync(retryMessage);
        }
    }
}

```

Рисунок 7.16 - Клас RetryWorker.

Далі буде описаний клас Program цього компонента:

В даному випадку цей клас створює та налаштовує Host для додатка, використовуючи ASP.NET Core. Цей хост дозволяє запускати фонові служби, які можуть працювати у фоновому режимі або налагоджувати конфігурацію додатка. Буде розглянуто кожен частину коду:

- *Host.CreateDefaultBuilder(args)* - цей метод створює новий екземпляр хоста за допомогою типових налаштувань для ASP.NET Core додатків;
- *ConfigureAppConfiguration((hostContext, config) => { ... })* - ця частина налаштовує конфігурацію додатка. Вона додає різні джерела конфігурації, такі як *appsettings.json* та змінні середовища, які можуть бути використані для налаштування додатка;

- *ConfigureServices((hostContext, services) => { ... })* - ця частина налаштовує служби, які будуть використовуватися в додатку. Вона отримує доступ до конфігурації та додає різні служби до контейнера інверсії управління (DI container). Служби, які додаються:
 - *RetryWorker*: Це фонові служба, яка була згадана раніше в цьому розділі. Вона додається як фонові служба для обробки певних завдань у фоновому режимі;
 - *AddAsyncCommunicationControl<MyMessage>(...)*: Цей метод, описаний у розділі 5, додає можливість маніпулювання даними повідомлень в базі даних;
 - *AddRabbitMQ()*: Цей метод, описаний у розділі 7.2, надає сервіси для відправки повідомлень до RabbitMQ;
- *Build()* - ця операція завершує налаштування хоста і повертає готовий до запуску екземпляр хоста;
- *host.Run()* - цей метод запускає хост, що дозволяє фоновим службам, доданим у попередніх кроках, почати виконувати свою роботу. Фонові служби будуть виконувати свої завдання в фоновому режимі із використанням налаштувань і служб, які були встановлені в попередніх кроках;

Загалом, цей клас допомагає створити та налаштувати механізм повторного відправлення повідомлень для роботи у фоновому режимі.

Лістинг цього класу наведено на рисунку 7.17.

```

var host = Host.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((hostContext, config) =>
    {
        config.AddJsonFile(path: "appsettings.json", optional: true);
        config.AddJsonFile(path: "appsettings.Development.json", optional: true);
        config.AddEnvironmentVariables();
    })
    .ConfigureServices((hostContext, services) =>
    {
        var configuration = hostContext.Configuration;

        services.AddHostedService<RetryWorker>();
        services.AddAsyncCommunicationControl<MyMessage>(
            connectionString: configuration["Messages:ConnectionString"],
            migrationAssembly: configuration["Messages:MigrationAssembly"]);
        services.AddRabbitMQ();
    }) // IHostBuilder
    .Build();

host.Run();

```

Рисунок 7.18 - Клас Program механізму повторного відправлення повідомлень.

Діаграма діяльності механізму повторного відправлення повідомлень асинхронної комунікації відображена в додатку Г.

Висновок до розділу 7

У цьому розділі було детально описано процес інтеграції системи керування повідомленнями. Ця інтеграція виконується через розробку простого додатка з використанням архітектури Clean Architecture. Основними компонентами є Domain, Infrastructure, API, та Consumer рівні. Розглядалися важливі аспекти, такі як моделювання даних, створення та конфігурація сервісів для обробки та відправлення повідомлень. Значну увагу було приділено механізму повторного відправлення повідомлень, що включає реалізацію фонових сервісів та ефективно

управління повідомленнями. Такий підхід демонструє гнучкість та ефективність розробленої системи в контексті реальних застосувань.

8 ТЕСТУВАННЯ СИСТЕМИ

Тестування системи буде проведено за двома випадками:

- перевірити, що підготовлене повідомлення, при успішній обробці, не надсилається повторно;
- перевірити, що підготовлене повідомлення, при невдалій обробці, повторно відправляється механізмом повторного відправлення повідомлень;

Для початку буде протестований перший випадок. Для цього буде запущено три застосунки:

- *MyWebApp* - для відправки запиту про створення продукту, ендпоінт з цього застосунку надішле повідомлення до черги;
- *MyConsumer* - для обробки повідомлення, відправленого з *MyWebApp*;
- *MyRetrySystem* - для повторного відправлення невдало опрацьованого повідомлення;

Рисунок 8.1 демонструє 3 запущених застосунка.

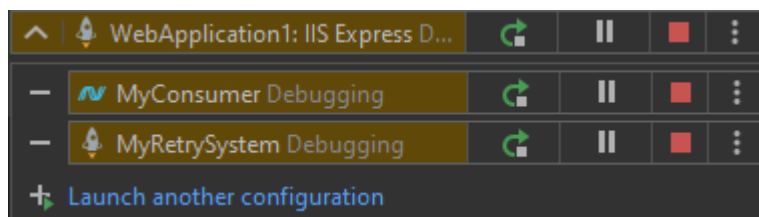


Рисунок 8.1 - Запущені застосунки *MyWebApp*, *MyConsumer*, *MyRetrySystem*.

В розробленому, в розділі 7.4, *MyConsumer* при обробці повідомлення існує логіка - якщо ціна продукту більше десяти, відмітити повідомлення як невдало оброблене, щоб воно було заново відправлене механізмом повторного відправлення повідомлень. Якщо ціна дорівнює десяти або менше, то повідомлення вважається успішно обробленим. Код обробки повідомлення відображений на рисунку 8.2

```

public async Task ExecuteAsync(MyMessage message, Product product)
{
    using var scope = _serviceProvider.CreateScope();
    var messageService = scope.ServiceProvider.GetRequiredService<IMessageService<MyMessage>>();
    try
    {
        Console.WriteLine($"Product: {product.Name}|{product.Price}");
        if (product.Price > 10)
        {
            throw new Exception(message: "wrong price");
        }
        message.Status = ExecutionStatus.SuccessfullyExecuted;
        await messageService.UpdateMessageAsync(message);
    }
    catch (Exception ex)
    {
        message.Status = ExecutionStatus.ExecutedWithErrors;
        try
        {
            await messageService.UpdateMessageAsync(message, RetryPolicy);
        }
        catch (Exception ex1)
        {
            Console.WriteLine(ex1.Message);
        }
    }
}

```

Рисунок 8.2 - Код обробки повідомлення.

Отже, буде відправлено запит до *MyWebApp* з тілом, де продукт має ціну менше за 10, наприклад 5. Рисунок 8.3 відображає цей запит та його результат.

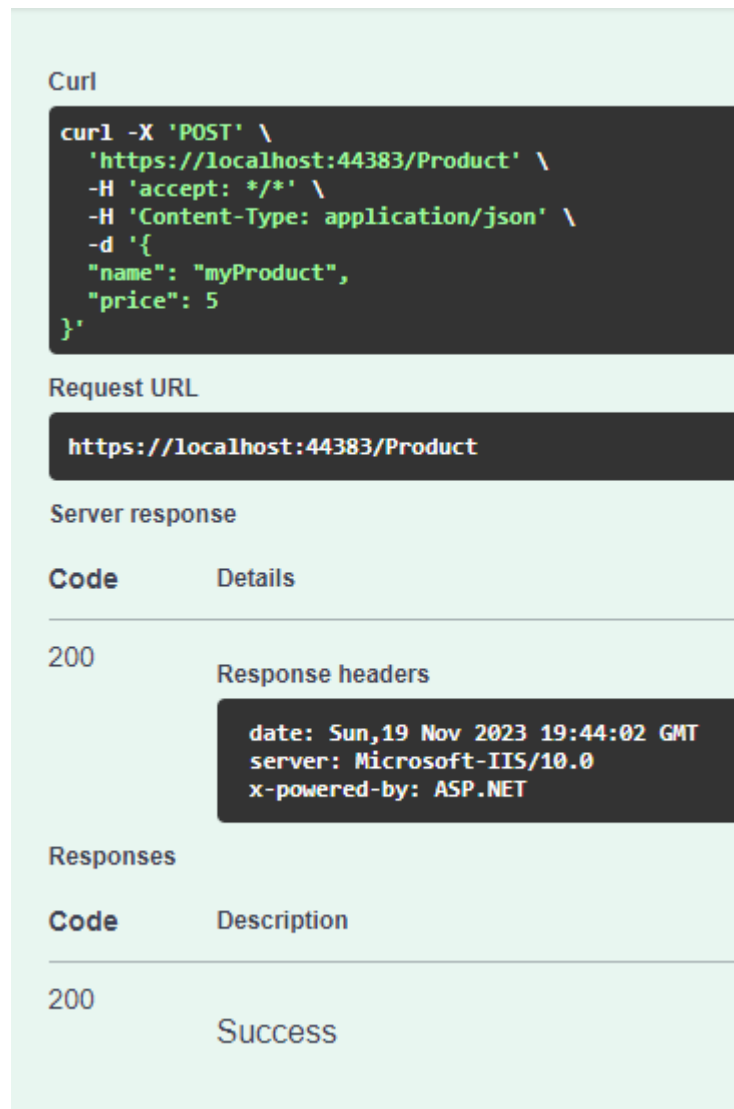


Рисунок 8.3 - Запит з ціною продукту 5 та результат цього запиту.

Як видно на рисунку 8.3, запит отримав відповідь HTTP Status Code 200, що відображає успіх.

Після цього запиту, в базі даних було створено нове повідомлення, як видно, для поля *Description* воно має значення “fromProductController”, як і було вказано в розділі 7.3. Поле *Content* має значення тіла повідомлення в форматі JSON, а саме назва продукту “myProduct”, та ціна 5. Поле *Status* дорівнює 0, що відображає значення *ToBeExecuted*, як було вказано в розділі 5. Поле *CreatedOn* містить значення дати та часу, коли повідомлення було створено. Поле *Queue* вказує значення “products”, що репрезентує назву черги, до якої було відправлено

повідомлення. Всі поля відносно повторної відправки повідомлення містять значення за замовчуванням.

Цей запис в базі даних відображений на рисунку 8.4.

	Id	Description	Content	Status	CreatedOn	ModifiedOn	Queue	ExecuteAt	MaxRetryAttempts	TotalRetryAttempts
▶	35	fromProductController	{"Name":"myProduct","Price":5}	0	2023-11-19 ...	NULL	products	NULL	0	0
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 8.4 - Дані надісланого повідомлення в базі даних.

Після цього, повідомлення надходить до консьюмера повідомлень *MyConsumer*.

Як видно з рисунку 8.5, що репрезентує логи *MyConsumer*, дані повідомлення були виведені в консоль, що відображає, що повідомлення було опрацьоване.

```
C:/Users/OleksandrLutsenko/
Press [enter] to exit.
Product: myProduct|5
```

Рисунок 8.5 - Логи *MyConsumer*.

Щоб переконатись, що повідомлення було опрацьоване успішно, будуть перевірені оновлені дані повідомлення в базі даних.

	Id	Description	Content	Status	CreatedOn	ModifiedOn	Queue	ExecuteAt	MaxRetryAttempts	TotalRetryAttempts
▶	35	fromProductController	{"Name":"myProduct","Price":5}	1	2023-11-19 ...	2023-11-1...	products	NULL	0	0
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 8.6 - Оновлені дані повідомлення в базі даних.

Як видно на рисунку 8.6, оновився *Status* повідомлення до значення 1, що відображає значення *SuccessfullyExecuted*. Також було оновлено значення *ModifiedOn*, тепер воно відображає дату та час, коли повідомлення було опрацьоване.

Отже, повідомлення було успішно опрацьоване, як і очікувалось.

Далі буде перевірено другий випадок, а саме що при невдалій обробці, повідомлення повторно відправляється механізмом повторного відправлення повідомлень.

Для цього потрібно відправити запит до *MyWebApp*, де в тілі запиту вказано ціну 100.

The screenshot displays a web client interface with the following sections:

- Curl**: A dark box containing the command:

```
curl -X 'POST' \  
'https://localhost:44383/Product' \  
-H 'accept: */*' \  
-H 'Content-Type: application/json' \  
-d '{  
  "name": "myInvalidProduct",  
  "price": 100  
'
```
- Request URL**: A dark box containing the URL `https://localhost:44383/Product`.
- Server response**: A table with two columns: **Code** and **Details**.

Code	Details
200	Response headers date: Sun, 19 Nov 2023 20:11:45 GMT server: Microsoft-IIS/10.0 x-powered-by: ASP.NET
- Responses**: A table with two columns: **Code** and **Description**.

Code	Description
200	Success

Рисунок 8.7 - Запит до *MyWebApp*, який має в тілі запиту ціну продукту 100, та результат цього запиту.

Як можна побачити на рисунку 8.7, запит був успішно опрацьований.

	Id	Description	Content	Status	CreatedOn	ModifiedOn	Queue	ExecuteAt	MaxRetryAttempts	TotalRetryAttempts
▶	36	fromProductController	{"Name":"myInvalidProduct",...	0	2023-11-19 ...	NULL	products	NULL	0	0
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 8.8 - Повідомлення з ціною продукту 5 в базі даних.

Як видно з рисунку 8.8, повідомлення було створено в базі даних.

Далі, *MyConsumer* повинен опрацювати повідомлення. На рисунку 8.9 можна побачити що повідомлення було опрацьоване.

```
C:/Users/OleksandrLutsenko/Desktop
Press [enter] to exit.
Product: myProduct|5
Product: myProduct|5
Product: myInvalidProduct|100
□
```

Рисунок 8.9 - Логи *MyConsumer*, що мають текст відносно повідомлення з ціною 100.

Буде перевірено, як саме було опрацьоване повідомлення, для цього потрібно знайти його в базі даних.

	Id	Description	Content	Status	CreatedOn	ModifiedOn	Queue	ExecuteAt	MaxRetryAttempts	TotalRetryAttempts
▶	36	fromProductController	{"Name":"myInvalidProduct",...	2	2023-11-19 ...	2023-11-1...	products	2023-11-19 20:18:37.430395	3	0
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 8.10 - Дані повідомлення після невдалої обробки.

Як можна побачити на рисунку 8.10, *Status* повідомлення змінив значення на 2, що відображає *ExecutedWithErrors*. Також з'явилися значення в трьох інших полях - *ModifiedOn*, адже дані повідомлення були оновлені, *ExecuteAt*, відображає коли наступний раз повідомлення повинно бути повторного відправлене, та *MaxRetryAttempts*, репрезентує максимальну кількість спроб повторної обробки цього повідомлення. Значення *ExecuteAt* та *MaxRetryAttempts* отримані та побудовані з політики повторних спроб цього консьюмера, ця політика була описана у розділі 7.4.

Отже, повідомлення було невдало оброблене, тому це повідомлення повинно бути заново відправлене механізмом повторного відправлення повідомлень, не раніше часу вказаного в полі *ExecuteAt*. Також, варто зазначити, що механізм повторного відправлення повідомлень шукає повідомлення для відправки, як було описано в розділі 7.5, кожні 5 секунд та пошук повідомлень займає деякий час, тому повідомлення може бути повторно відправлене на декілька секунд пізніше очікуваного.

Далі треба перейти до механізму повторного відправлення повідомлень, як видно з рисунку 8.11, повідомлення було знайдено в базі даних та повторно відправлене.

```
info: MyRetrySystem.RetryWorker[0]
      Sending retry message: 36 at 11/19/2023 22:29:49 +02:00
```

Рисунок 8.11 - Логи механізму повторного відправлення повідомлень, що репрезентують ID повідомлення, та дату та час його повторного відправлення.

Перш ніж повідомлення буде знову відпрацьовано *MyConsumer*, треба перевірити його дані в базі даних.

	Id	Description	Content	Status	CreatedOn	ModifiedOn	Queue	ExecuteAt	MaxRetryAttempts	TotalRetryAttempts
►	36	fromProductController	("Name": "myInvalidProduct", ...	4	2023-11-19 ...	2023-11-19 ...	products	2023-11-19 20:18:37.430395	3	1
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 8.12 - Оновлені дані повідомлення, після його повторної відправки.

Як видно з рисунку 8.12, *Status* повідомлення змінився на 2, що репрезентує *InRetryQueue*, отже повідомлення було опрацьоване механізмом повторного відправлення повідомлень. Також значення *TotalRetryAttempts* збільшилось на 1, що відображає що почалась перша спроба повторної обробки повідомлення.

Далі потрібно перейти до *MyConsumer*, так як тіло повідомлення містить ті самі дані, ціна продукту залишилась 100, отже повідомлення знову і надалі буде невдало опрацьоване, і знов і знов буде повторного відправлятися механізмом

повторного відправлення повідомлень, до тих пір, поки значення *TotalRetryAttempts* не дійде до значення *MaxRetryAttempts*.

Будуть перевірені дані повідомлення в базі даних, після максимальної кількості спроб повторної відправки повідомлення.

Id	Description	Content	Status	CreatedOn	ModifiedOn	Queue	ExecuteAt	MaxRetryAttempts	TotalRetryAttempts
36	fromProductController	{"Name":"myInvalidProduct",...	2	2023-11-19 ...	2023-11-1...	products	2023-11-19 20:38:47.348005	3	3

Рисунок 8.13 - Оновлені дані повідомлення, після максимальної кількості спроб повторної відправки повідомлення.

Як можна побачити на рисунку 8.13, *Status* повідомлення змінився на 2, що відображає *ExecutedWithErrors*, також значення *TotalRetryAttempts* досягло значення *MaxRetryAttempts*.

Оновлені дані означають, що повідомлення було опрацьоване неуспішно та спроби повторної відправки більше не будуть проводитись.

Отже, усі необхідні випадки були перевірені та тестування системи пройшло успішно. Повідомлення повторного відправляються в залежності від описаної політики спроб повторного відправлення повідомлень, як і було заплановано.

Висновок

до

розділу

8

У цьому розділі було проведено тестування системи керування повідомленнями, з фокусом на два ключові аспекти: перевірка успішної обробки повідомлень та тестування механізму повторного відправлення при невдалій обробці.

Для цього були задіяні три застосунки: *MyWebApp*, *MyConsumer* і *MyRetrySystem*. Тестування включало в себе - відправлення повідомлень з різними умовами та аналіз реакції системи. Результати показують, що система ефективно опрацьовує повідомлення згідно з очікуваннями, демонструючи надійність механізмів обробки та повторного відправлення в різних сценаріях.

9 РОЗРОБКА СТАРТАП ПРОЕКТУ

У сфері стартапів, що перебувають на початковому етапі свого розвитку, ключовою задачею є створення інноваційного продукту, який вносить новизну на ринок та відповідає невирішеним потребам відповідної галузі.

Статистика свідчить, що значна частина стартапів зустрічається з труднощами, що часто призводять до невдач. Основні виклики, з якими стикаються стартапи, включають недостатність попиту на ринку для нового продукту, обмежений доступ до фінансування, неефективний підбір команди, конкуренцію, що випереджає на ринку, та недоліки у стратегії ціноутворення та вартості продукту.

Вирішальним для успіху стартапу є глибоке розуміння ринкових потреб і тенденцій. Це створює основу для залучення інвестицій, які є критично важливими для розвитку стартапу, забезпечуючи його зростання та приносячи вигоду інвесторам.

9.1 Опис ідеї проекту

Зміст ідеї, напрямки застосування та вигоди для користувача наведені в таблиці 9.1.

Таблиця 9.1 - Опис ідеї стартап проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Система керування повідомленнями при асинхронній комунікації мікросервісів	ІТ-інфраструктури, хмарні сервіси, фінансові установи	Точне відстеження статусу повідомлень, поліпшення аналітики даних

	Електронна комерція, стартапи, великі корпорації	Забезпечення надійного зберігання даних, підвищення безпеки інформації
	Логістика, телекомунікації, охорона здоров'я	Гнучке управління потокami даних, підвищення ефективності комунікацій

Таблиця 9.1 підсумовує ключові аспекти ідеї стартапу, який фокусується на системі керування асинхронними повідомленнями в мікросервісних архітектурах. Вона демонструє, як система може бути застосована в різних галузях, включаючи ІТ-інфраструктури, хмарні сервіси, фінансові установи, електронну комерцію, логістику та охорону здоров'я.

Основні переваги для користувачів полягають у точному відстеженні статусу повідомлень, поліпшенні аналітики даних, надійному зберіганні даних, підвищенні безпеки інформації та ефективності комунікацій.

Буде проведений техніко-економічний аналіз переваг, він здійснюється шляхом порівняння основних характеристик ідеї проекту із властивостями існуючих рішень, таких як DataDog, RabbitMQ Sensor, Confluent Control Center, Hangfire, Elastic Stack (ELK), Prometheus & Grafana. Цей аналіз оцінює слабкі (W), нейтральні (N) та сильні (S) сторони проекту у порівнянні з конкурентами. Результати аналізу відображені на таблиці 9.2.

Таблиця 9.2 - Порівняння техніко-економічних характеристик ідеї проекту з конкурентами

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів							W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проєкт	Data Dog	RabbitMQ Sensor	Confluent Control Center	Hangfire	Elastic Stack (ELK)	Prometheus та Grafana			
1.	Моніторинг повідомлень	+/-	+	+	+	-	+	+		X	
2.	Технічні метрики	-	+(не тільки для повідомлень)	+	+	-	+(не тільки для повідомлень)	+(не тільки для повідомлень)	X		
3.	Зберігання повідомлень	+	+-(довготривале зберігання коштує багато)	+-(напряму в RabbitMQ)	+-(напряму в Kafka)	-	+-(довготривале зберігання коштує багато)	+-(довготривале зберігання коштує багато)			X

4.	Можливість керування повідомленнями	+	-	+-(напряму в RabbitMQ)	+-(напряму в Kafka)	+(надає можливість імплементувати будь-які дії для керування повідомленнями, в тому числі механізм повторного відправлення повідомлень)	-	-			X
----	-------------------------------------	---	---	------------------------	---------------------	---	---	---	--	--	---

З таблиці 9.2 можна побачити, що розроблювана система хоч і має недоліки, порівнюючи з іншими, але в критичних для системах характеристиках, таких як - зберігання повідомлень та можливість керування повідомленнями, вона є найкращою.

9.2 Технологічний аудит ідеї проекту

Технологічний аудит ідеї проекту зосереджується на детальному аналізі технічних аспектів, що становлять основу для розробки продукту. Важливо забезпечити, щоб технологічна основа була міцною, гнучкою та здатною адаптуватися до майбутніх змін та вимог ринку.

Технологічна здійсненність ідеї проекту описана в таблиці 9.3.

Таблиця 9.3 - Технологічна здійсненність ідеї проекту.

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Відправлення повідомлень асинхронної комунікації	Будь-який брокер повідомлень, в випадку магістерської дисертації - RabbitMQ	Наявні у вільному доступі.	Доступні, розповсюджені безкоштовно.
2	Зберігання даних	Будь-яка СУБД, в випадку магістерської дисертації - MySQL	Наявні у вільному доступі.	Доступні, розповсюджені безкоштовно.
3	Запис відправлених повідомлень асинхронної комунікації до бази даних	C#, .NET7, EntityFramework Core	Наявні у вільному доступі.	Доступні, розповсюджені безкоштовно.
4	Механізм	C#, .NET7,	Наявні у	Доступні,

	повторного відправлення повідомлень	Background Service	вільному доступі.	розповсюджу ються безкоштовно.
Обрана технологія реалізації ідеї проекту: зкомбіноване використання RabbitMQ, MySQL, C#, .NET7, та EntityFramework Core дозволяє ефективно реалізувати всі необхідні аспекти системи асинхронного обміну повідомленнями. RabbitMQ забезпечує надійну та гнучку систему брокера повідомлень, MySQL слугує ефективним рішенням для зберігання даних, тоді як C# і .NET7 з EntityFramework Core є потужними інструментами для розробки бекенд-логіки, включаючи запис повідомлень та механізм повторного відправлення. Ці технології є вільно доступними та широко використовуються у розробці сучасних програмних продуктів.				

Проаналізувавши таблицю 9.3, стає зрозуміло, що технологічна реалізація проекту є можливою, підходи реалізації описані у розділі 4 є актуальними.

9.3 Аналіз ринкових можливостей запуску стартап-проекту

У цьому розділі проводиться глибокий аналіз ринкових можливостей для запуску стартапу, спрямованого на управління асинхронними повідомленнями у мікросервісних архітектурах. Аналіз включає оцінку основних гравців на ринку, поточний обсяг продажів, динаміку ринку, наявність обмежень для входу на ринок та специфічні вимоги до стандартизації та сертифікації. Також розглядаються потенційні ринкові можливості та загрози, а також вимоги споживачів до продукту. Цей аналіз є важливим для визначення потенціалу проекту на ринку та розробки стратегії його впровадження.

Таблиця 9.4 - Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку	Характеристика
1	Кількість головних гравців	Невелика кількість великих компаній та численні малі та середні підприємства
2	Загальний обсяг продаж	Високий, з огляду на зростаючу потребу в мікросервісних рішеннях
3	Динаміка ринку	Зростаюча, зважаючи на активну цифровізацію бізнес-процесів
4	Наявність обмежень для входу	Технологічні бар'єри, потреба у високій кваліфікації розробників
5	Специфічні вимоги до стандартизації та сертифікації	Відповідність ІТ стандартам безпеки та конфіденційності
6	Середня норма рентабельності в галузі, %	Вища за середній банківський відсоток

Аналіз таблиці 9.4 вказує на значний потенціал ринку для стартапу, з огляду на високий попит та зростання сфери мікросервісних технологій. Незважаючи на високі технологічні бар'єри та потребу в спеціалізованих знаннях, рентабельність у цій галузі обіцяє бути високою.

Таблиця 9.5 - Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці клієнтів	Вимоги споживачів до товару
1	Моніторинг і управління асинхронними повідомленнями	ІТ компанії, розробники ПЗ, компанії з великими ІТ інфраструктурами	Шукають надійні, гнучкі рішення з простим інтерфейсом	Висока надійність, гнучкість, простота в налаштуванні
2	Підвищення ефективності комунікації в мікросервісних архітектурах	Хмарні провайдери, компанії, що займаються цифровою трансформацією	Потребують рішень, що легко інтегруються з існуючими системами	Інтеграція з різними брокерами повідомлень, масштабованість
3	Забезпечення відмовостійкості ІТ-систем	Середні та великі бізнеси з критичними ІТ-системами	Зосереджені на мінімізації ризиків та підвищенні стійкості	Висока надійність, можливість швидкого відновлення після збоїв

На основі даних з таблиці 9.5, можна зробити висновок, що проект має великий потенціал, оскільки він пропонує рішення, які зустрічають важливі потреби різних груп клієнтів, зокрема в області ефективності, надійності та інтеграції в ІТ-інфраструктуру.

Таблиця 9.6 - Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Швидка зміна технологій	Втрата актуальності продукту через нові технологічні рішення	Інвестування в R&D, адаптація до нових технологій
2	Конкуренція	Поява більш інноваційних або дешевших рішень від конкурентів	Розробка унікальних функцій, зниження витрат
3	Зміни в законодавстві	Нові вимоги до безпеки даних можуть вплинути на функціональність	Моніторинг законодавчих змін, своєчасне впровадження змін

Таблиця 9.7 - Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Розширення ринку IT-послуг	Збільшення потенційної клієнтської бази	Активний маркетинг, розширення мережі продажів

2	Цифрова трансформація бізнесу	Зростання попиту на ефективні ІТ-рішення	Налаштування продукту під потреби цифрової трансформації
3	Технологічні інновації	Використання нових технологій для покращення продукту	Інвестування в новітні технології та їх інтеграція в продукт

Аналіз факторів загроз і можливостей демонструє, що розроблюваний проект має значний потенціал на ринку, незважаючи на виклики, пов'язані зі швидким розвитком технологій і конкуренцією. Потрібно зосередитися на розробці унікальних функцій та адаптації до змін у галузі для забезпечення тривалого успіху.

Таблиця 9.8 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства
Тип конкуренції: монополістична	Невелика кількість гравців зі значною ринковою часткою	Розробка унікальних рішень для вирізнення на ринку
Рівень конкурентної боротьби: національний	Конкуренція між великими ІТ компаніями на рівні країни	Позиціонування продукту, акцент на місцевих особливостях

Галузева ознака: внутрішньогалузева	Конкуренція між компаніями, що пропонують схожі ІТ рішення	Спеціалізація на нішевих рішеннях, інновації
Конкуренція за видами товарів: товарно-видова	Різні рішення в межах однієї категорії продуктів	Диференціація продукту, підкреслення унікальності
Характер конкурентних переваг: нецінова	Якість, інновації, технічна підтримка	Вклад у якість та інновації, зосередження на сервісі
Інтенсивність: марочна	Бренди з високою впізнаваністю	Розробка сильного бренду, маркетингові кампанії

Аналіз таблиці 9.8 вказує на необхідність розробки унікальних рішень та вкладення в бренд і якість продукту для досягнення успіху на конкурентному ринку ІТ-рішень. Важливо зосередитися на інноваціях і відмінностях продукту, щоб вирізнитися серед конкурентів.

Таблиця 9.9 - Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
------------------	------------------	-----------------------	---------------	---------	------------------

	Datadog, RabbitMQ Sensor, Confluent Control Center, Hangfire, Elastic Stack (ELK)	Висока технологі чна складність потребує спеціалізо ваних знань. Середні до високих капіталов кладення для розробки та маркетинг у. Регулятор ні вимоги можуть бути складним и залежно від регіону	Обмежена кількість провайдерів брокерів повідомлен ь.	Зростаюча потреба в ефективни х рішеннях для керування асинхронн ими повідомле ннями. Висока конкурент ність через існуючи аналоги.	Висока конкуренція від спеціалізован их систем.
--	---	---	--	--	---

Висновки	Інтенсивність конкурентної боротьби з боку прямих конкурентів, є високою. Ці компанії мають значний вплив на ринку і пропонують різноманітні продукти, що задовольняють потреби користувачів. Це створює конкурентне середовище, де новим гравцям потрібно пропонувати інноваційні рішення та унікальні переваги для	Можливість входу в ринок: Є, але обмежені через високі бар'єри. Потенційні конкуренти: Є, здебільшого стартапи та нові технологічні компанії. Строки виходу на ринок: Непередбачувані, залежать від ринкових	Вплив на ціноутворення: Можуть впливати на ціни через обмежену конкуренцію. Вимоги до якості та надійності: Високі стандарти якості, які можуть вимагати значних інвестицій у розробку технологічних обмеження: Вимоги до сумісності з існуючими технологіями можуть впливати на	Клієнти частково диктують умови роботи на ринку: Потреба в ефективно сті: Клієнти шукають рішення, які підвищують ефективність керування асинхронними повідомленнями. Висока конкурентність: Через наявність аналогів, клієнти	Висока конкуренція: Існують спеціалізовані системи, які можуть задовольняти подібні потреби клієнтів. Вибір альтернатив: Клієнти можуть віддавати перевагу вже відомим, перевіреним рішенням, якщо вони вважають їх більш надійними або вигідними. Необхідність диференціації: Щоб бути
----------	--	--	--	--	---

	успішного входження на ринок і конкуренції з існуючими великими гравцям.	умов та здатності долати бар'єри.	дизайн та функціонал ьність продукту	можуть вимагати більшій функціона льності, кращої якості послуг або нижчих цін.	конкурентосп роможними, потрібно відмінні особливості продукту, які відзначають його серед аналогів.
--	--	--	---	--	--

Аналіз таблиці 9.9 показує, що ринок систем керування асинхронними повідомленнями є конкурентним, з високою інтенсивністю конкурентної боротьби серед існуючих гравців. Можливості входу в ринок існують, але обмежені через високі технологічні бар'єри та потребу в спеціалізованих знаннях. Постачальники мають вплив через обмежену кількість, але не диктують умови. Клієнти мають значний вибір, вимагаючи високої якості та ефективності. Товари-замінники створюють високу конкуренцію, вимагаючи диференціації продукту для успіху на ринку.

Таблиця 9.10 - Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування
1	Інноваційність та Унікальність	Проект вводить інноваційні рішення в управлінні асинхронними повідомленнями, що виділяє його серед конкурентів.
2	Якість та Надійність	Високі стандарти якості та надійності відповідають вимогам клієнтів до міцних та ефективних ІТ-систем.
3	Легкість Інтеграції	Сумісність проекту з існуючими ІТ-інфраструктурами робить його універсальним рішенням для різних бізнесів.
4	Підтримка та Обслуговування Клієнтів	Надання відмінної підтримки та обслуговування клієнтів є ключовим відмінником на ринку ІТ-рішень.
5	Вартісна Ефективність	Конкурентоспроможні ціни без компромісів щодо якості відповідають фінансовим

		обмеженням та очікуванням клієнтів.
--	--	-------------------------------------

Таблиця 9.10 ілюструє фактори конкурентоспроможності на основі аналізу конкуренції, характеристик ідеї проекту, вимог споживачів до товару та факторів маркетингового середовища.

Буде проведений порівняльний аналіз сильних та слабких сторін розроблюваної системи керування повідомленнями при асинхронній комунікації мікросервісів.

Таблиця 9.11 - Порівняльний аналіз сильних та слабких сторін системи керування повідомлень асинхронної комунікації мікросервісів

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з Snus Limited						
			-3	-2	-1	0	+1	+2	+3
1	Інноваційність та Унікальність	18					X		
2	Якість та Надійність	15				X		X	
3	Легкість Інтеграції	17				X			
4	Підтримка та Обслуговування Клієнтів	16				X			

5	Вартісна Ефективність	19							X
---	--------------------------	----	--	--	--	--	--	--	---

Таблиця 9.12 - SWOT- аналіз стартап-проекту

Сильні сторони: - інноваційність та унікальність продукту, що відрізняє його на ринку. - висока якість та надійність рішень; - легкість інтеграції з існуючими ІТ-інфраструктурами; - вартісна ефективність продукту;	Слабкі сторони: - можливі складнощі з постійним технічним оновленням продукту; - обмежені можливості для швидкого масштабування; - недостатня впізнаваність бренду на початкових етапах;
Можливості: - зростання попиту на керування асинхронними повідомленнями; - потреба в ІТ-інтеграціях через цифрову трансформацію бізнесів; - можливість формування партнерств з іншими ІТ-компаніями.;	Загрози: - швидкі зміни технологій можуть зробити продукт неконкурентоспроможним; - висока конкуренція на ринку ІТ-рішень; - економічні коливання можуть вплинути на інвестиційну привабливість;

Таблиця 9.12, побудована за допомогою SWOT-аналізу, показує, що проект має сильні сторони, такі як інноваційність, якість, легкість інтеграції та вартісну ефективність продукту. Водночас, існують слабкі моменти, такі як потенційні виклики у технічному оновленні, обмежені можливості для масштабування та

початкова невпізнаваність бренду. Загрози включають швидкі технологічні зміни та високу конкуренцію. Проте, є значні можливості, зумовлені зростанням попиту на управління асинхронними повідомленнями та потребами цифрової трансформації, що відкривають перспективи для розвитку та партнерства з іншими ІТ-компаніями.

Таблиця 9.13 - Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Розробка та тестування продукту з власними ресурсами	Висока	6-12 місяців
2	Пошук інвестицій для більш швидкого розвитку та маркетингу	Середня	3-6 місяців
3	Стратегічне партнерство з великими ІТ-компаніями	Низька	12-18 місяців
4	Концентрація на ринку малого та середнього бізнесу	Висока	6-9 місяців
5	Розвиток open-source компонентів для залучення спільноти	Середня	9-12 місяців

На основі проведеного аналізу, відображеного в таблиці 9.13, найбільш підходящою альтернативою для виведення стартап-проекту на ринок є "Пошук інвестицій для швидшого розвитку та маркетингу". Ця альтернатива забезпечує

відносно високу ймовірність отримання ресурсів та пропонує більш стислі строки реалізації (3-6 місяців). Це дозволяє ефективно конкурувати з потенційними проектами конкурентів, швидко відповідаючи на ринкові вимоги та зміни.

9.4 Розроблення ринкової стратегії проекту

Таблиця 9.14 - Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	ІТ-компанії, що шукають рішення для асинхронної комунікації	Висока	Середній	Висока	Середня
2	Стартапи, які потребують масштабовані ІТ-інфраструктури	Середня	Низький	Середня	Висока

3	Великі корпорації з розвиненими ІТ-системами	Низька	Високий	Низька	Низька
Які цільові групи обрано: 1					

На основі аналізу та ринкових можливостей, було обрано стратегію концентрованого маркетингу, для системи керування повідомленнями, з особливим фокусом на першу групу споживачів - ІТ-компанії, які шукають рішення для асинхронної комунікації. Ця група показала високий рівень готовності до сприйняття нашого продукту та має значний орієнтовний попит, що забезпечує міцну основу для успішного входу в ринок та зростання. Визначена стратегія дозволить нам ефективно розподілити ресурси та швидко адаптуватися до ринкових умов.

Таблиця 9.15 - Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспромо жні позиції відповідно до обраної альтернативи	Базова стратегія розвитку*
1	Концентрація на ІТ-компаніях	Концентрован ий маркетинг	Інноваційність, якість	Стратегія лідерства по витратах

2	Розширення на ринок стартапів	Диференційований маркетинг	Адаптивність, модульність	Стратегія диференціації
3	Вхід на масовий ринок	Масовий маркетинг	Ціноутворення, масштаб	Стратегія спеціалізації

В таблиці 9.15 вибір базової стратегії залежить від обраної альтернативи розвитку, стратегії охоплення ринку, та ключових конкурентних переваг. Зазначені стратегії розвитку відображають різні напрямки, за якими можна досягти конкурентних переваг, залежно від внутрішніх ресурсів та зовнішніх умов ринку.

Таблиця 9.16 - Визначення базової стратегії конкурентної поведінки

Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки*
Ні	Забирати споживачів у конкурентів	Так, інноваційність та висока якість продукту	Стратегія виклику лідера

Таблиця 9.17 - Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції	Вибір асоціацій, які мають сформувати комплексну
-------------------------------------	---------------------------	-------------------------------------	--

		власного стартап-проекту	позицію власного проекту (три ключових)
Висока якість, інноваційність	Стратегія лідера	Інноваційність, технічна перевага	Новаторство, надійність, експертиза

У таблицях 9.16 та 9.17 враховуються ключові вимоги цільової аудиторії, обрана стратегія розвитку та конкурентні переваги стартапу, що дозволяють сформувати комплексну позицію на ринку. Основна мета - створити унікальний образ торговельної марки, який буде відповідати запитам та сприйняттю потенційних клієнтів.

9.5 Розроблення маркетингової програми стартап-проекту

Для формування маркетингової концепції системи керування повідомленнями в таблиці 9.18 потрібно узагальнити результати аналізу конкурентоспроможності, визначивши ключові переваги потенційного товару:
Таблиця 9.18 - Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Ефективне управління асинхронним і повідомленнями	Висока продуктивність і надійність	Інноваційність, адаптивність до змін технологій, ефективне ціноутворення

2	Підвищення ефективності ІТ-інфраструктури	Зниження витрат на обслуговування	Гнучкість у використанні, висока інтеграція з існуючими системами
3	Забезпечення безперебійної роботи систем	Підвищення стійкості до відмов	Висока надійність, швидке відновлення після збоїв, оптимізація процесів

В таблиці 9.19 буде описано три рівні моделі товару.

Таблиця 9.19 - Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
I. Товар за задумом	Система керування повідомленнями при асинхронній комунікації мікросервісів		
II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх /Тл/Е/Ор
	1.Збереження повідомлень	М	Тх
	2.Механізм повторної відправки повідомлень	М	Тх
	Якість: ручне тестування та інтеграційні тести		
	Постачання: NuGet пакет		

	Марка: Snus Limited - Async Control System
III. Товар із підкріпленням	До продажу: тестовий період
	Після продажу: знижки при купівлі продукту на довготривалий період
За рахунок чого потенційний товар буде захищено від копіювання: ліцензія, патентування, торгова марка.	

В таблиці 9.20 наведені ціни послуг конкурентів та обрана ціна послуг розроблюваної системи керування повідомленнями.

Таблиця 9.20 - Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	\$200 - \$300	\$250 - \$350	\$20000 - \$40000 щомісяця	\$100 - \$200

Таблиця 9.21 - Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Угоди безпосередньо з розробником,	Постачальник повинен	Однорівневи й або	Змішана система:

	розрахунки через банківські або електронні системи за умовами угоди	забезпечити технічну підтримку, вирішення проблем інсталяції та обслуговування, обробку платежів	багаторівневий, залежно від ринку реалізації	пряма та через посередників у складних ринкових умовах для зменшення оперативних витрат
--	---	--	--	---

Таблиця 9.21 визначає потенційну систему збуту системи керування повідомленнями, з урахуванням поведінки клієнтів та вимог до функцій, які має виконувати постачальник. Оптимальна система збуту рекомендується на основі аналізу витрат і ефективності взаємодії з клієнтами.

Таблиця 9.22 - Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Цільові клієнти - це технічно освічені особи, які	Основними каналами комунікації є професійні соціальні	Продукт позиціонується як інноваційне рішення, яке	Головна мета рекламного повідомлення -	Рекламне звернення будується навколо концепції

шукають інноваційні та ефективні рішення для оптимізації своїх процесів. Вони цінують детальну технічну інформацію, випадки використання та відгуки.	мережі (наприклад, LinkedIn), тематичні блоги, спеціалізовані форуми та вебсайти, де фахівці діляться знаннями та відгуками. Додатково, ефективними каналами можуть бути вебінари та конференції.	пропонує високу якість та надійність у сфері асинхронної комунікації мікросервісів. Основні акценти - технологічна перевага, модульність, легкість інтеграції.	продемонструвати унікальність продукту та його технічні переваги. Зосередження на історіях успіху використання продукту, підкреслення його здатності вирішувати складні завдання.	"технологічного лідера", з акцентом на новаторстві та доведень ефективності продукту. Використовується комбінація візуальних зображень, відгуків клієнтів, а також детальних технічних описів, щоб залучити увагу цільової аудиторії.
--	---	--	---	---

Таблиця 9.22 охоплює всі аспекти маркетингових комунікацій, включаючи детальний аналіз специфіки поведінки цільових клієнтів, вибір найбільш ефективних каналів комунікацій, стратегії позиціонування продукту та розробку завдань для рекламних повідомлень. Цей підхід дозволяє не лише забезпечити високий рівень взаємодії з потенційними клієнтами, але й створити стійке уявлення про бренд та продукт у свідомості цільової аудиторії.

Ключовим елементом є розробка концепції рекламного звернення, яке ґрунтується на розумінні унікальних потреб і вподобань клієнтів, а також на акцентуванні особливостей продукту, які відповідають цим потребам. Ефективне

використання обраних каналів комунікації та змістовного меседжу допоможе залучити цільову аудиторію та сприятиме створенню позитивного іміджу бренду системи керування повідомленнями.

Висновок до розділу 9

Проведений аналіз демонструє, що стартап-проект системи керування повідомленнями при асинхронній комунікації мікросервісів має значний потенціал для комерціалізації на ринку. Основні показники, такі як наявність попиту, позитивна динаміка ринку, і, найголовніше, можливість досягнення високої рентабельності, підтверджують цю перспективу. Аналіз конкурентного середовища та потенційних груп клієнтів свідчить про цілком реалістичну можливість успішного входу на ринок, незважаючи на наявність певних бар'єрів входу.

Для ефективного виведення стартап-проекту системи керування повідомленнями на ринок було вирішено зосередитися на специфічному сегменті цільової аудиторії, який включає IT-інфраструктури, хмарні сервіси та фінансові установи. Такий підхід дозволяє максимально використовувати ресурси проекту, адаптуватися до специфічних потреб цієї групи клієнтів та ефективно розгортати маркетингові зусилля. З часом, після забезпечення стабільної присутності в обраному сегменті, планується розширення на інші ринкові сегменти, що дозволить збільшити ринкову частку та конкурентоспроможність проекту.

Враховуючи вищенаведені фактори, подальша імплементація проекту є вкрай доцільною. Це не лише дозволить компанії закріпитися на ринку з новим інноваційним продуктом, але й відкриває шлях для подальшого розвитку та розширення діяльності.

ВИСНОВКИ

У рамках магістерської дисертації була успішно розроблена та реалізована система керування повідомленнями при асинхронній комунікації мікросервісів. Ретельний аналіз існуючих технологій та ринкових потреб підтвердив актуальність та важливість проекту в контексті сучасних ІТ-рішень.

Розроблений продукт вирішує проблеми загублених повідомлень асинхронної комунікації мікросервісів, за допомогою запису їх до бази даних та механізму повторної відправки повідомлень.

У дисертації також було проведено детальний аналіз сучасних методів управління асинхронними повідомленнями в мікросервісних архітектурах. Була оцінена низка існуючих інструментів, таких як DataDog, RabbitMQ Sensor, Confluent Control Center, Hangfire, ELK Stack, Prometheus і Grafana. Аналіз виявив, що жоден з цих інструментів не відповідає всім вимогам проекту, особливо щодо довготривалого зберігання повідомлень та їхнього автоматизованого управління.

Реалізація дисертації включає використання передових технологій для розробки системи управління асинхронними повідомленнями у мікросервісних архітектурах, а саме, RabbitMQ для брокеражу повідомлень, MySQL для бази даних, а також .NET7 і C# для розробки основної логіки системи. Це поєднання технологій забезпечило високу ефективність, надійність та гнучкість системи, дозволяючи їй ефективно вирішувати проблеми, пов'язані з асинхронними мікросервісами, такі як відстеження та управління повідомленнями.

Важливою складовою роботи стала розробка стартап-проекту, що включала аналіз ринку та розробку стратегій маркетингу. Дослідження показало, що, незважаючи на високу конкуренцію, існують значні перспективи для успішного входу на ринок та подальшого розвитку проекту.

У підсумку, магістерська дисертація демонструє комплексний підхід до розробки та впровадження інноваційного ІТ-проекту, від ідеї до реалізації та аналізу ринкових можливостей. Результати дослідження свідчать про великий потенціал проекту та його здатність внести значний вклад у сферу асинхронної комунікації мікросервісів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Datadog documentation. URL: <https://docs.datadoghq.com/> (дата звернення: 21.09.2023)
2. RabbitMQ Sensor - Monitoring RabbitMQ. URL: <https://www.ibm.com/docs/en/instana-observability/current?topic=technologies-monitoring-rabbitmq> (дата звернення: 21.09.2023)
3. Understanding RabbitMQ Queue & Messaging. URL: <https://hevodata.com/learn/rabbitmq-queue/> (дата звернення: 18.09.2023)
4. Confluent Control Center documentation. URL: <https://docs.confluent.io/platform/current/control-center/index.html> (дата звернення: 22.09.2023)
5. Apache Kafka description. URL: <https://kafka.apache.org/> (дата звернення: 18.09.2023)
6. Hangfire description. URL: <https://www.hangfire.io/> (дата звернення: 22.09.2023)
7. Elastic Stack features. URL: <https://www.elastic.co/elastic-stack/features> (дата звернення: 23.09.2023)
8. Prometheus introduction. URL: <https://prometheus.io/docs/introduction/overview/> (дата звернення: 24.09.2023)
9. Graphana decription. URL: <https://grafana.com/docs/> (дата звернення: 24.09.2023)
10. What is .NET? URL: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet> (дата звернення: 01.10.2023)
11. Java description. URL: <https://www.java.com/en/download/help/index.html> (дата звернення: 02.10.2023)
12. What is Python? URL: <https://www.infoworld.com/article/3204016/what-is-python-powerful-intuitive-programming.html> (дата звернення: 02.10.2023)
13. What is NodeJS? URL: <https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs> (дата звернення: 03.10.2023)

14. Why to use Ruby on Rails? URL: <https://www.toptal.com/ruby-on-rails/after-two-decades-of-programming-i-use-rails> (дата звернення: 03.10.2023)
15. C# documentation. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/> (дата звернення: 05.10.2023)
16. Overview of EntityFramework Core. URL: <https://learn.microsoft.com/en-us/ef/core/> (дата звернення: 07.10.2023)
17. What is SQL Server? URL: <https://www.sqlservertutorial.net/getting-started/what-is-sql-server/> (дата звернення: 19.10.2023)
18. MySQL overview. URL: <https://www.openlogic.com/blog/mysql-overview> (дата звернення: 19.10.2023)
19. About SQLite. URL: <https://www.sqlite.org/about.html> (дата звернення: 20.10.2023)
20. What is PostgreSQL? URL: <https://aws.amazon.com/rds/postgresql/what-is-postgresql/> (дата звернення: 21.10.2023)
21. What are NoSQL databases? URL: <https://www.ibm.com/topics/nosql-databases> (дата звернення: 24.10.2023)
22. The Clean Architecture. URL: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html> (дата звернення: 10.10.2023)
23. The Clean Architecture. Domain Layer. URL: <https://dev.to/julianlasso/clean-architecture-domain-layer-3bdd> (дата звернення: 10.10.2023)
24. The Clean Architecture. Application Layer. URL: <https://dev.to/julianlasso/clean-architecture-application-layer-3n1e> (дата звернення: 10.10.2023)
25. What is Android OS? URL: <https://www.techtarget.com/searchmobilecomputing/definition/Android-OS> (дата звернення: 14.10.2023)
26. Django at glance. URL: <https://docs.djangoproject.com/en/5.0/intro/overview/> (дата звернення: 24.10.2023)

27. Introduction to Flask. URL: <https://opensource.com/article/18/4/flask> (дата звернення: 15.10.2023)
28. What is JavaScript. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript (дата звернення: 16.10.2023)
29. LINQ. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/linq/> (дата звернення: 18.10.2023)
30. Amazon Simple Queue Service. URL: <https://aws.amazon.com/sqs/> (дата звернення: 20.10.2023)