

Міністерство освіти і науки України  
Державний заклад  
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

**Козаков Артем В'ячеславович**

**ДОСЛІДЖЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ З  
АСИНХРОННОЇ МОДЕЛЮ ВЗАЄМОДІЇ**

**кваліфікаційна робота  
здобувача вищої освіти другого (магістерського) рівня  
освітньої програми «Мультимедійні системи»  
за спеціальністю 121 „Інженерія програмного забезпечення”**

Особистий підпис \_\_\_\_\_ Козаков Артем

Науковий керівник \_\_\_\_\_ Світлана ПЕРЕЯСЛАВСЬКА,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій та  
систем

Завідувач кафедри \_\_\_\_\_ Микола СЕМЕНОВ,  
кандидат педагогічних наук,  
доцент кафедри інформаційних  
технологій та систем

## АНОТАЦІЯ

**Козаков А.В.**

**Тема:** Дослідження мікросервісної архітектури з асинхронної моделлю взаємодії.

**Спеціальність:** 121 "Інженерія програмного забезпечення".

**Установа:** ДЗ ЛНУ імені Тараса Шевченка, 2022р.

**Кваліфікаційна робота містить:** 110 стор., 28 рис., 55 джерел, 1 додаток.

**Об'єкт дослідження** – мікросервісна архітектура програмного додатку.

**Предмет дослідження** – асинхронна модель взаємодії між мікросервісами.

**Мета роботи** – дослідження технологій розробки мікросервісів із застосуванням асинхронної моделі взаємодії.

**Методи дослідження:** *теоретичні:* аналіз науково-технічних джерел з проблем дослідження; *емпіричні:* порівняний аналіз мікросервісів з монолітною архітектурою та порівняння асинхронної і синхронної моделі взаємодії; *експериментальні:* тестування розробленого додатку.

**Результати роботи.** Досліджено існуючі сучасні методи розробки мікросервісної архітектури з асинхронної моделлю взаємодії. Розроблено 3 мікросервіса які взаємодіють між собою за допомогою асинхронної моделі через брокер повідомлень.

**Ключові слова:** мікросервіси, асинхронна взаємодія, брокер повідомлень.

## Abstract

**Kazakov A.V.**

**Theme:** Research of micro-service architecture with an asynchronous interaction model.

**Specialty:** 121 "Software Engineering".

**Institution:** Taras Shevchenko National University of Luhansk, 2022.

**Qualification work contains:** 110 Pages., 28 fig., 55 sources, 1 appendix.

**Object of research:** microservice architecture of a software application.

**Subject of research:** asynchronous model of interaction between microservices.

**Purpose of research:** research of technologies for developing microservices using an asynchronous interaction model.

**Methods of research:** *theoretical*: analysis of scientific and technical sources on research problems; *empirical*: comparative analysis of microservices with monolithic architecture and comparison of asynchronous and synchronous interaction models; *experimental*: testing of the developed application.

**Methods of research.** The existing modern methods for developing a microservice architecture with an asynchronous interaction model are studied. 3 microservices have been developed that interact with each other through a message broker.

**Work results.** The existing modern methods for developing a microservice architecture with an asynchronous interaction model are studied. 3 microservices have been developed that interact with each other using an asynchronous model through a message broker

**Keywords:** microservices, asynchronous interaction, message broker.

## ЗМІСТ

Вступ.....	5
Розділ 1. ТЕОРЕТИЧНІ ОСНОВИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА МІЖПРОЦЕСНОЇ ВЗАЄМОДІЇ .....	7
1.1 Поняття мікросервісної архітектури .....	7
1.2 Форми взаємодії між службами .....	13
1.3 Асинхронна взаємодія між службами, види та особливості.....	19
Висновки до розділу 1 .....	27
Розділ 2. ПРОГРАМНА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ З АСИНХРОННОЮ МОДЕЛЛЮ ВЗАЄМОДІЇ .....	28
2.1 Програмні інструменти для реалізації мікросервісної архітектури з асинхронною моделлю взаємодії .....	28
2.2 Архітектурна модель мікросервісного додатку .....	31
2.3 Проектування бази даних для мікросервіса.....	32
2.4 Проектування мікросервісів та міжпроцесної взаємодії.....	35
2.4.1 WebService .....	35
2.4.2 WareHouseService .....	48
2.4.3 NotificationService.....	50
2.5. Тестування проекту.....	51
Висновки до розділу 2 .....	55
ВИСНОВКИ.....	56
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ .....	58
ДОДАТКИ.....	62

## ВСТУП

Сучасні програми повинні відповідати вимогам масштабованості і високого навантаження, розподілені або, принаймні, використовують деякі розподілені компоненти. Тому одним з найбільш актуальних архітектурних підходів до побудови системи є мікросервіси.

Архітектурою системи називають логічну структуру, що описує окремі компоненти, їх властивості і зв'язку як єдину систему.

Мікросервіси є більш гнучкою архітектурою ніж моноліт в якому архітектура будується як єдиний компонент, а не як набір декількох розподілених додатків.

Актуальність даної теми полягає у все більшій увазі до мікросервісів, масового переходу великих і високонавантажених сервісів від монолітної до мікросервісної архітектури.

**Мета роботи** – дослідження технологій розробки мікросервісів із застосуванням асинхронної моделі взаємодії.

**Об'єкт дослідження** – мікросервісна архітектура програмного додатку.

**Предмет дослідження** – асинхронна модель взаємодії між мікросервісами.

Для досягнення даної мети повинні бути вирішені наступні **завдання**:

1. Дослідити монолітну та мікросервісну архітектуру виділити плюси та мінуси кожної з них.
2. Дослідити методи взаємодії між мікросервісами.
3. Проаналізувати асинхронну модель взаємодії між мікросервісами.
4. Розробити додатки, які побудовані на мікросервісній архітектурі з асинхронною моделлю взаємодії використовуючи брокер повідомлень.

**Методи дослідження:**

*Теоретичні:* аналіз науково-технічних джерел з проблем дослідження;  
*емпіричні:* порівняний аналіз мікросервісів з монолітною архітектурою та

порівняння асинхронної і синхронної моделі взаємодії; *експериментальні*: тестування розробленого додатку.

### **Практичне завдання:**

Дослідити існуючі сучасні методи розробки мікросервісної архітектури з асинхронною моделлю взаємодії. Розробити 3 мікросервіса з асинхронною моделлю які взаємодіють між собою через брокер повідомлень.

**Структура дипломної роботи:** Робота складається з пояснювальної записки, списку використаних джерел, додатків. Обсяг роботи становить 66 сторінок, обсяг використаної літератури - 40 джерел.

В пояснювальній записці перший розділ містить теоретичні основи мікро сервісної архітектури, її характеристики особливості, виявлені її плюси та мінуси відносно монолітної архітектури, далі розглянути форми взаємодії між службами, асинхронну та синхронну модель, виявлені особливості кожного підходу та їх особливості, також у цьому розділі було проведено порівняння брокерів повідомлень RebbitMQ, Apache Kafka, Amazon Simple Queue Service (SQS) виділені їх плюси та мінуси розглянуто випадки в яких один брокер повідомлень має більше переваг перед іншими.

У другому розділі було встановлено вимоги до додатків, спроектована архітектура додатків та їх взаємодія, надані діаграми класів сервісів та загальна схема проекту. Представлено веб інтерфейс програми та описано функціональну складову програмних додатків. Представлена схема взаємодії сервісів через брокер повідомлень. Розглянуті можливості використання та подальший розвиток проекту.

Додаток містить в собі код програми.

# РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ МІКРО СЕРВІСНОЇ АРХІТЕКТУРИ ТА МІЖПРОЦЕСНОЇ ВЗАЄМОДІЇ

## 1.1 Поняття мікросервісної архітектури

З метою кращого розуміння сутності мікросервісної архітектури необхідно розглянути *монолітну архітектуру*, яка є традиційною моделлю програмного забезпечення, що реалізується єдиним модулем, який працює автономно та незалежно від інших програм [21].

Багато веб-застосунків невеликого та середнього розміру створюються з використанням монолітної архітектури. Монолітний додаток доставляється як єдиний програмний артефакт, що розгортається. Всі його компоненти – інтерфейс користувача, бізнес-логіка та логіка доступу до бази даних – об'єднані в єдину програму і розгортаються на сервері програм.

Монолітні програми змушують кілька груп розробників синхронізувати дату розгортання, тому що їх код необхідно збирати, тестувати та розгортати як єдине ціле [1].

### *Переваги монолітної архітектури*

При використанні монолітної архітектури зручно створювати програми на основі однієї бази коду, тому її основна перевага полягає у швидкості розробки. До переваг монолітної архітектури можна віднести такі особливості:

*Просте розгортання.* Використання одного файлу або каталогу, що виконується, спрощує розгортання.

*Розробка.* Програма легше розробляти, коли вона створена з використанням однієї бази коду.

*Продуктивність.* У централізованій базі коду та репозиторії один інтерфейс API часто може виконувати ту функцію, яку під час роботи з мікросервісами виконують численні API.

*Спрощене тестування.* Монолітне додаток є єдиним централізованим модулем, тому наскрізне тестування можна проводити швидше, ніж при використанні розподіленої програми.

*Зручне налагодження.* Весь код знаходиться в одному місці, завдяки чому стає легше виконувати запити та знаходити проблеми.

*Недоліки монолітної архітектури*

До недоліків монолітної архітектури можна віднести такі особливості.

*Зниження швидкості розробки.* Великий монолітний додаток ускладнює та уповільнює розробку.

*Масштабованість.* Неможливо масштабувати окремі компоненти.

*Надійність.* Помилка в одному модулі може вплинути на доступність програми.

*Перешкоди запровадження технологій.* Будь-які зміни в інфраструктурі чи мові розробки впливають на додаток цілком, що часто призводить до збільшення вартості та тимчасових витрат.

*Недостатня гнучкість.* Можливості монолітних додатків обмежені технологіями, що використовуються.

*Розгортання.* При внесенні невеликої зміни буде потрібно повторне розгортання всього монолітного додатку.

На противагу монолітній архітектурі ставиться ***архітектура мікросервісів***. Ідея мікросервісів спочатку виникла у спільноті розробників програмного забезпечення як пряма відповідь на багато проблем (як технічних, так і організаційних), пов'язаних зі спробами масштабування великих монолітних додатків. *Мікросервіс* - це невелика, слабко пов'язана розподілена служба. Мікросервіси дозволяють взяти додаток з великим набором функцій і розкласти його на прості в управлінні компоненти з певними обов'язками. Мікросервіси допомагають долати традиційні проблеми складності великої бази коду, розбиваючи її на невеликі чітко визначені частини. Ключові поняття, про які слід пам'ятати, розмірковуючи про



мікросервіси – це декомпозиція та розв'язка (unbunding). Функції додатків мають бути повністю незалежні друг від друга [1].

Термін "мікросервіси" існує вже дуже давно. На конференції Cloud Computing Expo в 2005 Пітер Роджерс використовує його у своїй презентації. Тоді він використовував термін "мікро-веб-сервіс" для іменування компонентів програмного забезпечення. Юваль Леві мав схожу думку щодо мікросервісів, він так-же вважав, що саме мікросервіси є наступним ступенем розвитку архітектури Microsoft. Він описав, як добре розроблена сервісна платформа застосовує основні архітектурні принципи веб-та веб-служб разом із Unix-подібними, щоб забезпечити радикальну гнучкість та простоту, надаючи платформу для застосування сервісно-орієнтованої архітектури у всьому середовищі додатків. У майстерні архітекторів програмного забезпечення, що проводилася недалеко від Венеції в 2011 році, використовувався термін «мікросервіс» для опис загального архітектурного стилю, який розвивали учасники останні роки. І, нарешті, в 2012 році, все та ж команда, що і в 2011, вирішила прийняти цей термін, як остаточний.

Одним з перших користувачів мікросервісів була компанія Netflix. У 2009 році компанія Netflix зіштовхнулася з проблемами зростання. Її інфраструктура не справлялася з попитом на послуги Netflix, що стрімко розвиваються, по потоковій передачі відео. Компанія вирішила перенести ІТ-інфраструктуру з приватних центрів обробки даних до публічної хмари, а також перейти від монолітної архітектури до мікросервісної. Єдина проблема полягала в тому, що терміна «мікросервіси» на той час не існувало, тому про структуру мало відомостей.

Компанія Netflix стала однією з перших великих організацій, які успішно перейшли з монолітної архітектури на хмарну архітектуру мікросервісів. Вона отримала спеціальний приз журі JAX у 2015 році — завдяки новій інфраструктурі, в яку вдалося впровадити методологію DevOps. Сьогодні компанія Netflix має понад тисячу мікросервісів. З їх допомогою здійснюється управління окремими частинами платформи та їх підтримка, тоді як

розробники регулярно розгортають код (кількість розгортань може досягати кількох тисяч разів на день).

Компанія Netflix однією з перших перейшла від монолітної архітектури до мікросервісної, яка стає все більш популярною на сучасному ринку.

### ***Характеристики мікросервісної архітектури***

- логіка застосування розбита на дрібні компоненти з чітко визначеними узгодженими межами відповідальності;
- кожен компонент відповідає за вузьке коло завдань та розгортається незалежно від інших; один мікросервіс відповідає за частину предметної області;
- для обміну даними між собою мікросервіси застосовують полегшені протоколи, такі як HTTP и JSON (JavaScript Object Notation – форма запису об'єктів JavaScript);
- програми на основі мікросервісів завжди обмінюються даними з використанням технологічно нейтрального формату (найчастіше використовується JSON), тому технічна реалізація служби не має значення; це означає, що додаток, що складається з мікросервісів, може бути написаний кількома мовами і з використанням кількох технологій;
- мікросервіси – завдяки невеликому розміру, незалежному та розподіленому характеру – дозволяють організаціям мати невеликі групи розробників із чітко визначеними сферами відповідальності. Ці групи можуть працювати над досягненням єдиної мети, наприклад над створенням програми, але кожна несе відповідальність лише за ті служби, над якими вони працюють.

### ***Переваги мікросервісів***

Оскільки архітектура мікросервісів складається з незалежно працюючих модулів, кожену службу можна розробляти, оновлювати, розгортати та масштабувати окремо від інших. Оновлення можна виконувати частіше, підвищуючи надійність, час безперебійної роботи та продуктивність програмного забезпечення.

*Гнучкість.* Просувайте гнучкі методи роботи серед невеликих команд, які регулярно розгортаються.

*Гнучке масштабування.* Коли мікросервіс досягає граничного навантаження, можна швидко виконати розгортання нових екземплярів цієї служби у супутньому кластері та знизити навантаження. Тепер ми працюємо з кількома власниками та без збереження стану, а клієнти розподілені за різними примірниками. З таким підходом ми можемо підтримувати екземпляри значно більшого розміру.

*Безперервне розгортання.* В мікросервісах є можливість регулярно та прискорено робити цикли релізу.

*Легкість обслуговування та тестування.* Команди можуть експериментувати з новими функціями та повертатися до попередньої версії, якщо щось не працює. Це спрощує оновлення коду та прискорює випуск нових функцій на ринок. Крім того, в окремих службах легко знаходити та виправляти помилки та баги.

*Незалежне розгортання.* Мікросервіси є окремими модулями, тому з ними можна легко і швидко виконувати незалежне розгортання окремих функцій.

*Гнучкість технологій.* У разі використання архітектури мікросервісів команди можуть вибирати інструменти з урахуванням своїх переваг.

*Висока надійність.* Розгортаючи зміни для конкретної служби, можна не боятися, що програма вийде повністю.

#### *Недоліки мікросервісів*

До недоліків мікросервісів можна віднести такі особливості.

*Розростання процесу розробки.* Мікросервіси ускладнюють роботу порівняно з монолітною архітектурою, оскільки у різних місцях виникає дедалі більше служб, створених кількома командами. Якщо розростання не контролюється належним чином, воно призводить до уповільнення розробки та зниження операційної ефективності.

*Експонентне зростання витрат на інфраструктуру.* Кожен новий мікросервіс може мати свою вартість комплексу тестів, інструкцій з розгортання, інфраструктури хостингу, інструментів моніторингу тощо.

*Додаткові організаційні витрати.* Командам потрібний додатковий рівень комунікації та співробітництва, щоб координувати роботу над оновленнями та інтерфейсами.

*Проблеми при налагодженні.* Кожен мікросервіс має свій набір журналів, що ускладнює налагодження. Крім того, додаткові труднощі можуть виникати у тому випадку, коли один бізнес-процес виконується на кількох машинах.

*Відсутність стандартизації.* Без загальної платформи може виникнути ситуація, де розширюється список мов, стандартів ведення журналів та засобів моніторингу.

*Відсутність ясності у питаннях володіння.* У міру появи нових служб збільшується і кількість команд, що працюють над ними. Згодом стає складніше визначити, які служби команда може використовувати і до кого слід звертатись за підтримкою.

### ***Перспективи використання мікросервісної архітектури***

*Великі колективи.* Групам розробників, які працюють над різними мікросервісами, не потрібно синхронізувати один з одним кожен крок, вибір інструментів та інші деталі. Нові функції можна розробляти паралельно та запускати у міру готовності.

*Об'ємні проекти зі складною архітектурою.* Оновлювати та підтримувати окремі модулі набагато простіше, ніж контролювати, як зміни позначаються на системі загалом.

*Продукти з різко змінним трафіком.* Якщо ваш продукт починає частіше користуватися в період свят або розпродажів, мікросервіси дозволять вам швидко масштабуватися і зменшити ризик відмови системи. До того ж, вам не доведеться платити за додаткову інфраструктуру, яка потрібна тільки в періоди пікових навантажень.

*Програми, які потребують частих оновлень. Достатньо змінити та налагодити лише той модуль, який ви хочете оновити. Це суттєво скорочує час розробки та наближає реліз.*

## **1.2. Форми взаємодії між службами**

У монолітному додатку, керованому єдиним процесом, компоненти викликають одне одного лише на рівні мови чи з допомогою викликів функції. Вони можуть бути тісно пов'язані, якщо створюються об'єкти з кодом (наприклад, `new ClassName()`), або можуть бути викликані незв'язано, якщо використовується впровадження залежності, посилаючись на абстракції, а не на конкретні екземпляри об'єкта. У будь-якому випадку об'єкти виконуються в одному процесі. Найскладніше завдання під час переходу від монолітного докладання до додатку з урахуванням мікрослужб полягає у зміні механізму взаємодії. Пряме перетворення внутрішньопроцесних викликів у виклики RPC до служб призведе до надмірної та неефективної взаємодії, що не підходить для розподілених середовищ. Труднощі розробки розподілених систем відомі так добре, що існує зведення принципів під назвою «Помилки про розподілені обчислення», де перераховані припущення, які часто роблять розробники при переході від монолітних до розподілених конструкцій [22].

Додаток на основі мікрослужб є розподіленою системою, що працює на декількох процесах або службах, іноді навіть не кількох серверах або вузлах. Зазвичай кожен екземпляр служби – це процес. Таким чином, служби повинні взаємодіяти за протоколом внутрішньопроцесної взаємодії, наприклад, HTTP, AMQP або двійковим протоколом, таким як TCP, залежно від характеру кожної служби.

*Типи зв'язку*

Клієнт та служби можуть взаємодіяти через різні типи зв'язку в залежності від сценарію та цілей. Ці типи зв'язку можна поділити на два напрямки [22].

Перша група визначає, чи є протокол синхронним або асинхронним:

- Синхронний протокол HTTP – це синхронний протокол. Клієнт відправляє запит і чекає на відповідь від служби. Це не залежить від виконання коду клієнта, яке може бути синхронним (потік заблокований) або асинхронним (потік не заблокований, відповідь зрештою буде відправлена). Тут важливо, що протокол (HTTP/HTTPS) є синхронним і код клієнта зможе продовжити виконання лише після отримання відповіді від HTTP-сервера.

- Асинхронний протокол. Інші протоколи, наприклад AMQP (протокол, підтримуваний багатьма операційними системами та хмарними середовищами), використовують асинхронні повідомлення. Код клієнта або відправник повідомлення зазвичай не очікує відповіді. Він просто надсилає повідомлення, як при надсиланні повідомлення в чергу RabbitMQ або іншого брокера повідомлень.

Друга група визначає, чи має запит одного або декількох одержувачів:

- Один отримувач. Кожен запит повинен оброблятися лише одним одержувачем чи службою. Наприклад, шаблон Command.

- Декілька одержувачів. Кожен запит може оброблятися різною кількістю одержувачів — від нуля до кількох. Такий тип взаємодії має бути асинхронним. Наприклад, механізм publish/subscribe, який використовується у таких шаблонах, як архітектура, керована подіями. Він ґрунтується на інтерфейсі шини подій або брокері повідомлень, коли події оновлюють дані у кількох мікрослужбах. Зазвичай це реалізується через службову шину або подібний об'єкт, наприклад, службову шину Azure, за допомогою тем і підписок.

Додаток з урахуванням мікрослужб часто використовує комбінацію цих стилів взаємодії. Найпоширеніший тип — це взаємодія з одним одержувачем за синхронним протоколом, наприклад HTTP або HTTPS, при виклику

звичайної служби веб-API HTTP. Для асинхронної взаємодії між мікрослужбами зазвичай використовуються протоколи повідомлень.

Синхронна та асинхронна взаємодія мікросервісів-це два різні підходи до обміну даними та виконання запитів між мікросервісами. Обидва підходи мають свої плюси і мінуси, і вибір підходу залежить від конкретних вимог і характеристик системи. Різниця між взаємодіями мікросервісів представлена на рис. 1.1.

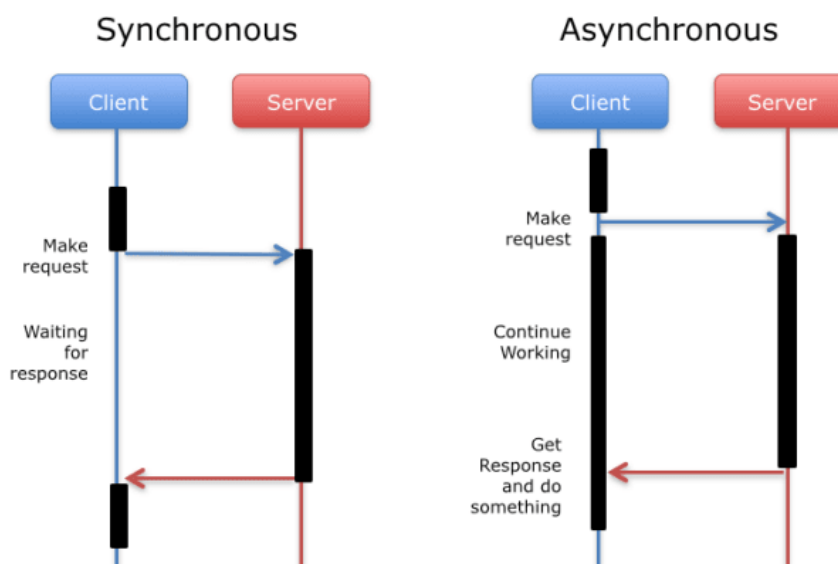


Рис. 1.1. Приклад взаємодій у мікросервісних архітектурах

### *Синхронна взаємодія*

У синхронній взаємодії клієнтська мікросервіс очікує відповіді від викликаного мікросервісу, перш ніж продовжувати свою роботу. Це означає, що клієнтський мікросервіс блокується і чекає відповіді, перш ніж перейти до наступного кроку.

*Плюси синхронної взаємодії:*

- Простота. Синхронна взаємодія простіша у реалізації та налагодженні.
- Прозорість. Синхронне взаємодія дозволяє легко відстежувати і управляти послідовністю виконання операцій.

*Мінуси синхронної взаємодії:*

- Залежність від доступності. Якщо мікросервіс, що викликається, недоступний або повільний, це може призвести до затримок і блокувань у клієнтському мікросервісі.
- Вузьке місце. Якщо синхронні дзвінки виконуються послідовно, це може стати вузьким місцем продуктивності.

Приклад використання. Додаток електронної комерції може синхронно викликати мікросервіс для перевірки наявності товару перед оформленням замовлення. Клієнтський мікросервіс блокується, поки не отримає відповідь про наявність товару.

*Асинхронна взаємодія*

В асинхронній взаємодії клієнтська мікросервіс надсилає запит до мікросервісу, що викликається, і продовжує свою роботу, не чекаючи відповіді. Відповідь може бути отримана пізніше, наприклад, через повідомлення або колбеки.

*Плюси асинхронної взаємодії:*

- Відмовостійкість. Асинхронна взаємодія дозволяє уникнути блокування клієнтського мікросервісу при недоступності викликається мікросервісу.
- Масштабованість. Асинхронна взаємодія може бути паралельною, що сприяє кращій масштабованості системи.

*Мінуси асинхронної взаємодії:*

- Складність. Асинхронна взаємодія вимагає більш складної реалізації, оскільки необхідно обробляти асинхронні відповіді та керувати станом запитів.



- Ускладнення налагодження. Відстеження та налагодження асинхронної взаємодії може бути складнішим через розподіл запитів та відповідей у часі.

Приклад використання. Система обробки платежів може асинхронно надсилати запит на обробку платежу, а потім отримувати відповідь про статус платежу через повідомлення. Такий підхід дозволяє клієнтському мікросервісу продовжувати роботу без очікування відповіді платіжного сервісу.

Правильний вибір підходу залежить від наступних факторів:

- Час відгуку. Якщо потрібна миттєва реакція, а затримки неприпустимі, синхронна взаємодія може бути кращою.
- Надійність. Якщо надійність і відмовостійкість важливі, асинхронна взаємодія може бути кращим, так як уникає блокувань і дозволяє більш гнучко обробляти помилки і відмови.
- Продуктивність. Якщо система вимагає високої продуктивності та паралельної обробки запитів, асинхронна взаємодія може бути більш ефективною.

На процес маршрутизації мікросервісів також має вплив такі компоненти як API Gateway та Service Discovery

*API Gateway* - це тип служби, яка знаходиться між Клієнтом та набором серверних служб. Коли клієнт робить запит до серверних служб, шлюз API направляє запит до служби та повертає відповідь клієнту. Це спрощує роботу клієнта завдяки переміщенню логіки виклику кількох мікросервісів до шлюзу API. Існуючих клієнтів не цікавлять зміни у внутрішній інфраструктурі, оскільки вони взаємодіють лише з рівнем шлюзу API[2].

Шлюз API може виконувати й інші завдання, такі як автентифікація, обмеження швидкості та моніторинг. Вивантажуючи ці завдання з серверних служб, шлюз API може підвищити продуктивність системи.

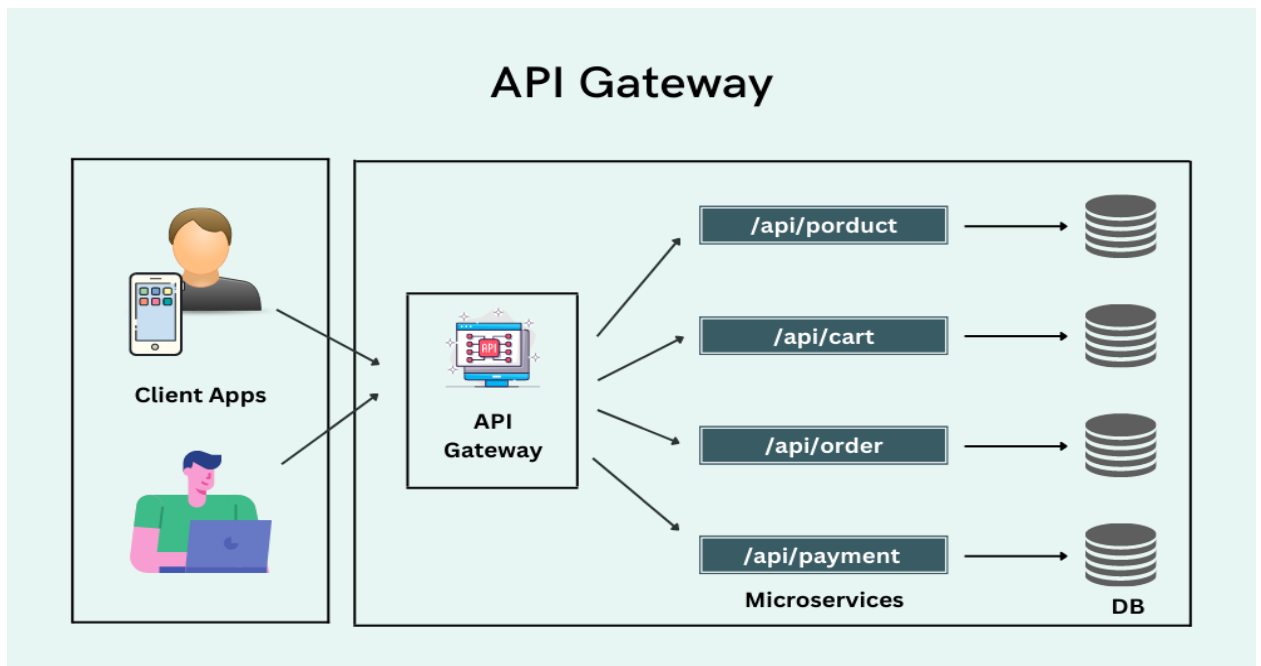


Рис. 1.2. Приклад роботи API Getway

Вони часто забезпечують єдину точку входу для всіх серверних служб компанії рисунок 1.2. Це може бути корисно для компаній, які мають багато різних послуг, або для компаній, які хочуть надати єдину точку входу для зовнішніх розробників.

Шлюзи API можуть забезпечити кілька переваг, включаючи:

- Зниження складності для клієнтів.
- Підвищена безпека.
- Підвищена продуктивність.
- Менша сукупна затримка.
- Більш швидкий час відгуку.
- Розширення застарілих додатків.

*Service Discovery* створений для того, щоб з мінімальними витратами можна підключити новий додаток в уже існуюче наше оточення. Використовуючи *Service Discovery*, ми можемо максимально розділити або контейнер у вигляді докера, або віртуальний сервіс від того оточення, в якому він запущений.

На класичному прикладі в web-це фронтенд, який приймає запит Користувача. Далі виконує маршрутизацію його на backend. На даному рисунку рис. 1.3 load-balancer балансує на два backend.

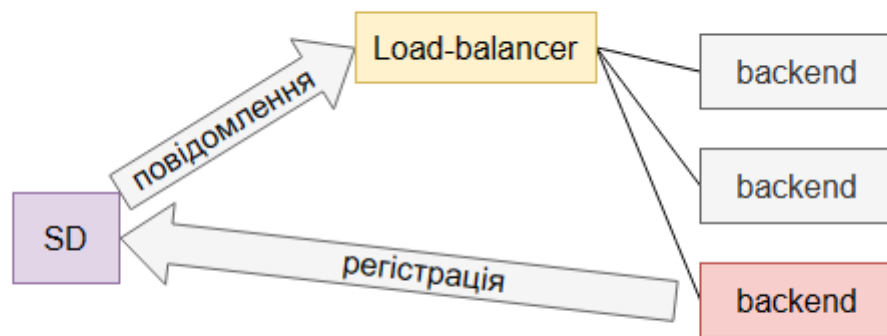


Рис. 1.3. Приклад роботи балансувальника

Спільнота мікрослужб сповідує філософію "розумні кінцеві точки та дурні канали". Цей слоган рекомендує створювати структуру, де окремі мікрослужби мінімально залежатимуть один від одного і матимуть максимальну внутрішню узгодженість. Як уже говорилося, кожна мікрослужба має власні дані та власну логіку предметної області. Але мікрослужби, що становлять комплексну програму, зазвичай просто створюються за допомогою взаємодії REST, а не складних протоколів, таких як webSockets, і гнучких комунікацій на основі подій замість централізованих оркестраторів бізнес-процесів.

## 1.2 Асинхронна взаємодія між службами, види та особливості

В рамках асинхронної взаємодії клієнт після відправки запиту серверу може продовжувати роботу, навіть якщо відповідь на запит ще не прийшов.

Прикладом асинхронної взаємодії є електронна пошта. Інший приклад-поширення повідомлень про новини різних видів відповідно до наявного на поточний момент реєстром передплатників, де кожен передплатник визначає теми, які його цікавлять.

Асинхронна взаємодія дозволяє отримати більш високу продуктивність системи за рахунок використання часу між відправкою запиту і отриманням відповіді на нього для виконання інших завдань. Інша важлива перевага асинхронної взаємодії-менша залежність клієнта від сервера, можливість продовжувати роботу, навіть якщо машина, на якій знаходиться сервер, стала недоступною. Ця властивість використовується для організації надійного зв'язку між компонентами, що працюють, навіть якщо і клієнт, і сервер не постійно працюють.

У той же час асинхронні взаємодії більш складно використовувати. Оскільки при такій взаємодії потрібно писати специфічний код для отримання і обробки результатів запитів, системи, засновані на асинхронних взаємодіях між своїми компонентами, значно важче розробляти і супроводжувати.

Найчастіше асинхронне взаємодія реалізується за допомогою черг повідомлень. При відправці повідомлення клієнт поміщає його у вхідну чергу сервера, а сам продовжує роботу. Після того, як сервер обробляє всі попередні повідомлення в черзі, він вибирає це повідомлення для обробки, видаляючи його з черги. Після обробки, якщо необхідна відповідь, сервер створює повідомлення, що містить результати обробки, і кладе його у вхідну чергу клієнта або в свою вихідну.

Черги повідомлень можуть бути налаштовані різними способами. У компонента може бути одна вхідна черга, а може — і кілька, для повідомлень від різних джерел або мають різний зміст. Крім того, компонент може мати вихідну чергу, або кілька, замість того, щоб класти повідомлення у вхідні черги інших компонентів. Черги повідомлень можуть зберігатися незалежно як від компонентів, які кладуть туди повідомлення, так і від тих, які забирають їх звідти. Повідомлення в чергах можуть мати пріоритети, а сама черга —

реалізовувати різні політики підтримки або зміни пріоритетів повідомлень в ході роботи.

Асинхронна взаємодія передбачає, що клієнт посилає повідомлення, яке буде оброблено коли-небудь пізніше. І тут виникають питання - як отримувати відповідь про обробку? І чи потрібно взагалі його отримувати? Тому що деякі системи залишають це користувачам, пропонуючи періодично оновлювати таблиці документів в інтерфейсі, щоб дізнатися статус. Але досить часто статус все-таки потрібен для того, щоб продовжити обробку — наприклад, після повного завершення резервування замовлення на складі передати його на оплату.

Тепер розглянемо ті самі черги про які йдеться вище, вони називаються брокерами повідомлень.

Брокер повідомлень-архітектурний патерн в розподілених системах. додаток, який перетворює повідомлення по одному протоколу від програми-джерела в повідомлення протоколу програми-приймача, тим самим виступаючи між ними посередником. Крім перетворення повідомлень з одного формату в інший, в завдання брокера повідомлень також входить: перевірка повідомлення на помилки;

- маршрутизація конкретного приймача (ам).
- розбиття повідомлення на кілька маленьких, а потім агрегування відповідей приймачів і відправка результату джерела.
- збереження повідомлень у базі даних.
- виклик веб-сервісів.
- поширення повідомлень передплатникам, якщо використовуються шаблони типу "видавець – підписник».

Уявімо, що у нас є сервіси А і В. А робить синхронний запит у сервіс В і чекає, поки В на своєму боці його обробить і поверне відповідь. Потім на основі цієї відповіді А вирішує, що саме робити далі.

Зазвичай чекати відповідь потрібно пару секунд, але є речі, які вимагають більше часу: наприклад, рендеринг відео або запит у велику базу. В такому

випадку клієнту доводиться чекати дуже довго. Уявіть, що відвідувач сайту натискає кнопку "завантажити» або "купити" і отримує у відповідь іконку завантаження на кілька хвилин.

Щоб уникнути подібного сценарію, синхронну взаємодію замінюють на асинхронну. Сервіс А відправляє великий запит сервісу В й не блокує свою роботу під час очікування відповіді. В такому випадку застосовується ще один вузол системи, який буде працювати як кур'єрська служба, — брокер повідомлень. Він бере на себе роль гаранта доставки даних — приймає запити від А і чекає відповіді від В, поки А продовжує роботу.

Ця модель передбачає, що асинхронна взаємодія здійснюється відповідно до наступної логіки двох ролей:

- Publishers публікують нову інформацію у вигляді згрупованих за деяким атрибутом повідомлень;
- Subscribers підписуються на потоки повідомлень з певними атрибутами та обробляють їх.

Наприклад, у нас є служба нотифікації, яка хоче читати лише дискретні потоки даних із повідомленнями. Він підписується на таку чергу і бере тільки повідомлення про те, що потрібно відправити СМС користувачеві, а повідомлення про рендеринг відео ігнорує.

Виходить, що брокер-це служба в інфраструктурі, до якої підключаються відправники і одержувачі даних. Брокер відповідає за групування потоків даних від відправників, створення черг і видачу даних одержувачам. А ось як саме він це робить, залежить від технології конкретного брокера.

### ***RabbitMQ***

RabbitMQ — це брокер повідомлень, заснований на протоколі AMQP, Advanced Message Queuing Protocol. Це відкритий протокол передачі повідомлень, який потрібен для спілкування різних частин системи між собою. Система складається з декількох компонентів:

- AMQP-брокер маршрутизує повідомлення і допомагає частинам системи спілкуватися між собою.
- Producer (відправник) відправляє повідомлення в брокер.
- Consumer (читач) отримує ці повідомлення.
- Exchange (обмінник) отримує повідомлення і розподіляє їх між чергами.
- Queue (черга) зберігає повідомлення і віддає їх підписаним одержувачам.
- Binding зберігає правила для обмінника.

### *Плюси RabbitMQ*

- Легкість розробки. RabbitMQ має клієнтські бібліотеки для більшості сучасних мов та відкритий код для розуміння.
- Просте адміністрування. У RabbitMQ зручна адмінка, де ви можете в режимі реального часу розбиратися з тим, що відбувається. Роутинги можна налаштовувати в процесі, перемикаючи навантаження і змінюючи правила обробки.
- Тонка настройка. Багато параметрів можна змінювати, щоб підлаштувати систему під свої потреби. Особливо це стосується черг.

### *Минуси RabbitMQ*

- Робота при високому навантаженні. RabbitMQ має ускладнення при горизонтальному масштабуванні в кластері. Доводиться додавати налаштування кластеризації над чергами, а вони складні і працюють погано. Доводиться вибирати, чим пожертвувати — гнучкістю або швидкістю.

### *Apache Kafka*

Kafka влаштований простіше RabbitMQ, і сутностей у нього менше. А ще він часто зустрічається у вакансіях: після 2020 року Kafka помітно потіснив RabbitMQ.

Концептуально в Kafka немає різноманіття конфігурацій, складних механізмів розподілу повідомлень по чергах і процесу доставки кожного окремого

повідомлення. Ядро його функціональності-запис даних, зберігання їх протягом заданого часу і видача цих даних за запитом.

Всі інші особливості можна розглядати як наслідок цього концепту. І в цьому його перевага перед іншими opensource-рішеннями, які працюють по пуш-механізму або протоколу AMQP

### *Плюси Kafka*

- Робить всього дві речі: записує і віддає. Якщо використовувати брокер разом з надкластером ZooKeeper, то можна налагодити кластерні трансфери — Kafka довго цього не вмів.
- Пропускна здатність-мільйони повідомлень в секунду. Причому її можна ефективно нарощувати: додати датчик в кластер і при переповненні просто створювати новий, налаштовуючи між ними реплікацію. Саме тому Kafka став промисловим стандартом.
- Дозволяє перечитувати повідомлення. Якщо ми прочитали повідомлення, а потім втратили зміну в базі, можна відкотити офсет назад і прочитати повідомлення знову.
- Дозволяє читати повідомлення пачками. Можна запросити відразу 1000 повідомлень, що знижує навантаження на мережу.

### *Мінуси Kafka*

- Проблеми з обробкою битих повідомлень. У RabbitMQ необроблене повідомлення ми заново закидаємо в чергу, і воно крутиться, поки його не вийде обробити. У Kafka для обробки наступного повідомлення нам потрібно обробити його або пропустити поточний. Бите ми просто втратимо назавжди.

У Kafka для цього застосовують концепцію dead letter queue — окремого місця для збереження битих повідомлень. Ми беремо щось бите, кладемо окремо і потім намагаємося обробити знову. Але це все одно додаткова складність.

- Потрібно вести облік останнього прочитаного повідомлення, причому для кожного читача. Тому що дані незмінні, і ми розкидаємо не їх, а



читачів по одному і тому ж масиву даних. Тобто зберігаємо ті точки, де вони зупинилися в читанні. Для цього є кілька рішень, але всі вони, звичайно, обтяжують систему.

### *Amazon Simple Queue Service (SQS)*

І ще один брокер-SQS від Amazon. Це керований сервіс черг повідомлень, за допомогою якого можна так само, як і за допомогою інших брокерів, ізолювати і масштабувати мікросервіси і безсерверні додатки. SQS пропонує два типи черг повідомлень:

- *Стандартні черги* забезпечують максимальну пропускну здатність і доставку повідомлень за принципом "хоча б один раз". Стандартна черга намагається дотримуватися порядку повідомлень, але він може бути порушений. Якщо система вимагає збереження порядку, то краще використовувати чергу FIFO.
- *Черги FIFO* з обмеженою пропускну здатністю гарантують, що повідомлення будуть оброблятися строго одноразово і виключно в порядку відправлення. Вони призначені для покращення обміну повідомленнями між додатками, коли порядок операцій та подій є критичним або коли дублювання неприпустимо.

SQS працює в зв'язці з SNS-сервісом обміну повідомленнями для зв'язку між додатками, а також між додатками і користувачами. Обмін відбувається за моделлю *pub — sub* («видавець-передплатник»): одержувачі підписуються на тему (топiк), а видавець публікує в цю тему повідомлення, які може зчитувати безліч одержувачів.

### *Плюси SQS*

- *Популярність за кордоном.* Практично всі великі компанії використовують інфраструктуру Amazon, тому і цей брокер актуальний. Якщо ви шукаєте роботу за кордоном, то знання сервісів Amazon буде великим плюсом.
- *Безпека.* Ретельне шифрування всіх повідомлень, які йдуть в брокера і від брокера.

- *Автоматичні бекапи.*
- Проста пряма відправка імейлів, СМС, http-запитів.
- *Вбудований DLQ.* Розбиті повідомлення можна зберігати та обробляти пізніше.
- *Інтеграція з усіма іншими сервісами Amazon.* Він має той самий тип авторизації, чітку структуру, дизайн та SDK.

### *Мінуси SQS*

- Vendor lock. Повна залежність від Amazon. Перейти на іншого постачальника, якщо щось трапиться, буде складно.

Вибір того чи іншого брокера залежить від конкретних завдань, якщо вам важлива гнучкість в маршрутизації повідомлень всередині системи. Інструментарій для побудови шляхів доставки даних в RabbitMQ здатний вирішити складні сценарії в організації потоків подій. А так само якщо вам важливий сам факт доставки повідомлень, але порядок доставки не принциповий. Так як налаштування RabbitMQ його не гарантують що на сервіс повідомлення прийдуть в тому ж порядку в якому вони були відправлені.

Apache Kafka однозначно підходить, якщо ви працюєте з big data. Реплікація та паралельна обробка допоможуть масштабувати систему до нескінченності. Також Kafka виграє по продуктивності: може досягти пропускну здатності в мільйони повідомлень в секунду навіть при обмежених ресурсах, так само якщо вам важливо що б порядок повідомлень виходив в тому ж порядку в якому він був відправлений. А так само він дає можливість переглядати історію повідомлень.

SQS в совою чергу розроблений спеціально для черг повідомлень і підтримує тільки обробку повідомлень за принципом FIFO (first in, first out). Так само він забезпечує високу доступність і відмовостійкість, але може мати більш високу затримку і меншу пропускну здатність. SQS може бути найкращим вибором для додатків, які надають пріоритет високій доступності та відмовостійкості з мінімальними вимогами до обслуговування та інфраструктури.

## Висновки до розділу 1

Перший розділ було присвячено теоретичному дослідженню мікросервісної архітектури, де особлива увага приділялася видам міжсервісної взаємодії. Було проаналізовано актуальність використання мікросервісної архітектури, виконано порівняння мікросервісів з монолітом, визначено плюси і мінуси архітектур, розглянуто їх характеристики, особливості та перспективи використання.

Розглянуто форми взаємодії між службами мікросервісної архітектури, а саме: синхронна та асинхронна модель взаємодії, виділено особливості кожної з них. Детальніше розглянута асинхронна модель взаємодії, виділені її і особливості.

Встановлено, що асинхронна взаємодія має гнучку архітектуру тому вона може продовжувати роботу після відправки запиту другого сервісу не зупиняючи повністю виконання коду при очікуванні відповіді, така система має більшу відмовостійкість і легше масштабірується однак використання асинхронної взаємодії вимагає більш складної реалізації так як необхідно обробляти асинхронні відповіді і управляти станом запитів, а так само налагодження асинхронної взаємодії може бути складніше через розподіл запитів і відповідей в часі.

Також було виконано порівняння брокерів повідомлень RabbitMQ, Apache Kafka та Amazon Simple Queue Service (SQS), виявлені плюси і мінуси кожного брокера, також було досліджено доцільність застосування кожного виду брокерів в залежності від умов та вимог.

## **РОЗДІЛ 2. ПРОГРАМНА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ З АСИНХРОННОЮ МОДЕЛЛЮ ВЗАЄМОДІЇ**

### **2.1 Програмні інструменти для реалізації мікросервісної архітектури з асинхронною моделлю взаємодії**

В дипломній роботі буде спроектована мікросервісна архітектура з асинхронною взаємодією на прикладі інтернет магазину.

В якості основних технологій при проектуванні та розгортанні мікросервісів будуть використовуватися ASP .NET CORE, RabbitMQ, Docker та СУБД PostgreSQL.

Docker – це платформа контейнеризації з відкритим кодом, за допомогою якої можна автоматизувати створення програм, доставку та управління ними. Платформа дозволяє швидше тестувати і викладати додатки, запускати на одній машині необхідну кількість контейнерів. Контейнеризація та віртуалізація мають схожість, але є і відмінності. Віртуалізація нагадує окремий комп'ютер зі своїм обладнанням і ОС, всередині якого можна запустити ще одну ОС. А контейнеризація передбачає, що віртуальне середовище запускається з ядра ОС, не передбачає віртуалізації обладнання і знижує споживання ресурсів.

Контейнер – це стандартна одиниця програмного забезпечення, яка упаковує код та всі його залежності, щоб програма швидко та надійно працювала з одного обчислювального середовища в інше. Зображення контейнера Docker-це легкий, автономний, виконуваний пакет програмного забезпечення, який включає все необхідне для запуску програми: код, час виконання, Системні інструменти, системні бібліотеки та налаштування.

Контейнером який буде використовуватися в проекті буде СУБД PostgreSQL.

PostgreSQL – це вільно поширювана об'єктно-реляційна система управління базами даних (СУБД) з відкритим вихідним кодом, написаному на мові C [5].

Відмінність PostgreSQL від другої популярної СУБД MySQL полягає в тому, що MySQL розрахована на проекти з інтенсивним читанням даних, для яких важлива швидкість і легкість управління, а PostgreSQL більш підходить для складних запитів і роботи з великими обсягами інформації (Big Data). У них різні підходи до зберігання даних і їх обробці, вони відрізняються продуктивністю і кількістю підтримуваних типів даних (в Postgre їх більше).

Переваги та недоліки PostgreSQL

#### *Переваги*

- *Розширюваність і багатий набір типів даних* – крім стандартних, в PostgreSQL є типи для геометричних розрахунків, мережевих адрес і повнотекстового пошуку. Потужна система розширень дозволяє додавати нові можливості і типи даних.
- *Масштабованість* – для підвищення продуктивності і масштабованості в PostgreSQL використовуються різні види блокувань на рівні таблиць і рядків, шість видів індексів, серед яких B-дерево і узагальнене дерево пошуку (GiST) для повнотекстового пошуку.
- *Крос-платформенність* – PostgreSQL підтримується всіма популярними операційними системами, серед яких різні дистрибутиви Linux і BSD, macOS, Windows, Solaris та інші.
- *Безпека* – PostgreSQL має безліч інструментів для захисту даних від злоумисників: пароль, Kerberos, LDAP, GSSAPI, SSPI, PAM та інші.
- *Можливості NoSQL* – крім стандартних форматів, PostgreSQL підтримує XML, а з дев'ятої версії — JSON і jsonb.
- *Вільне розповсюдження і відкритий код* – проект поширюється під ліцензією BSD, що дозволяє безкоштовно його використовувати, модифікувати і поширювати.

#### *Недоліки*

- *Складність Налаштування* – Налаштування бази даних вимагає глибокого розуміння архітектури та параметрів.
- *Високе споживання ресурсів* – PostgreSQL може споживати більше ресурсів (пам'яті та часу процесора) порівняно з деякими іншими СУБД.
- *Відсутність деяких функцій* – в порівнянні з деякими комерційними СУБД PostgreSQL може злегка відставати у функціональності.

Для перегляду та редагування бази даних буде використовуватися DBeaver.

DBeaver – це безкоштовна програма для роботи з СУБД. З її допомогою можна створювати нові бази, змінювати і видаляти дані в уже існуючих, виконувати SQL-запити.

Для написання мікросервісів я буду використовувати IDE Visual Studio та мову програмування C#.

Microsoft Visual Studio – лінійка продуктів компанії Microsoft, що включають інтегроване середовище розробки (IDE) програмного забезпечення і ряд інших інструментів. Дані продукти дозволяють розробляти як консольні додатки, так і ігри і додатки з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, UWP а також веб-сайти, веб-додатки, веб-служби як в рідному, так і в керованому кодах для всіх платформ, підтримуваних Windows, Windows Mobile, Windows CE, .NET Framework, .Net Core, .Net, MAUI, Xbox, Windows Phone. Net Compact Framework і Silverlight. Після покупки компанії Xamarin корпорацією Microsoft з'явилася можливість розробки IOS і Android програм.

Проект буде побудований на базі ASP.NET Core за допомогою шаблону програмування MVP.

ASP.NET Core надає технологію для створення веб-додатків на платформі .Net, що розвивається компанією Microsoft. В якості мов програмування для розробки додатків на ASP.NET Core використовуються C# і F#.

ASP.NET Core MVC представляє в загальному вигляді побудови програми навколо трьох основних компонентів - Model (моделі), View (уявлення) і Controller (контролери), де моделі відповідають за роботу з даними, контролери представляють логіку обробки запитів, а уявлення визначають візуальну складову. Основна мета застосування цієї концепції полягає у відділенні бізнес-логіки (моделі) від її візуалізації (уявлення, виду). За рахунок такого поділу підвищується можливість повторного використання коду. Найбільш корисно застосування даної концепції в тих випадках, коли користувач повинен бачити ті ж самі дані одночасно в різних контекстах і/або з різних точок зору.

#### *Асинхронність взаємодії*

Асинхронність в пректі зроблена завдяки брокеру повідомлень RabbitMQ, це дозволяє після відправки повідомлення сервісом WebService продовжувати його роботу без очікування відповіді, сервіс далі може обробляти запити користувача, редагувати базу даних, під час очікування відповіді.

## **2.2. Архітектурна модель мікросервісного додатку**

Мікросервісна архітектура з асинхронною взаємодією буде реалізована на прикладі інтернет магазину. Структура проекту складається з трьох мікросервісів: сервіс WEB, який відповідає за обробку веб інтерфейсу користувача, сервіс WarehouseService, який відповідає за роботу зі складом, і сервіс Notification для відправки повідомлень про статус замовлення покупцеві.

Також в рішення є проект ObjectModel де зберігаються класи які використовуються в кожному проекті свого роду бібліотека, це необхідно для того щоб позбутися від написання одного і того ж коду в різних ділянках проекту, а один раз написати потрібні класи і завдяки системі успадкування використовувати цей код там де це необхідно, цей бпроект написаний на

основі шаблону Веб-API ASP.NET Core. Структурна модель проекти виглядає наступним чином рис. 2.1.

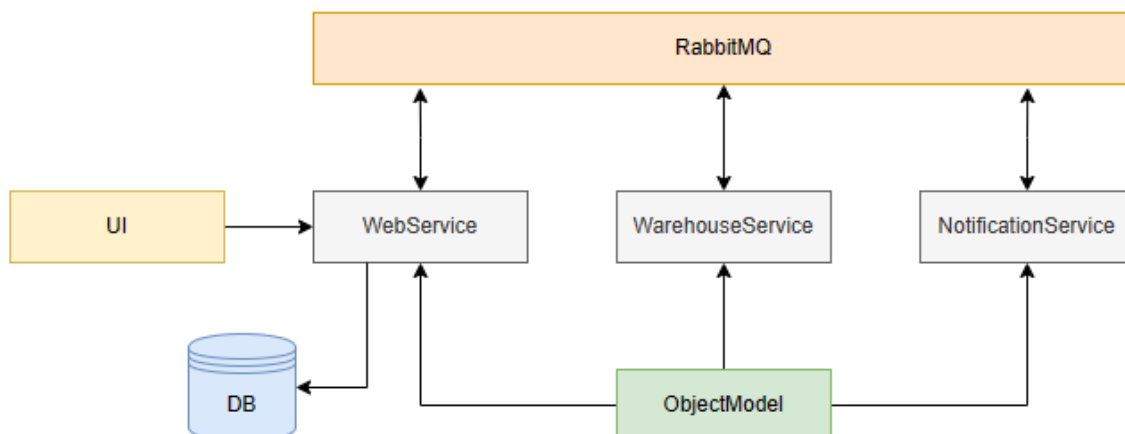


Рис. 2.1 Структурна модель проекту.

Сервіси взаємодіють через брокер повідомлень RabbitMQ, в якому є головний обмінник (Exchange) під назвою OrderExchange всередині нього 3 разв'язування перший amazon.orderService.order через який сервіс WebService передає моделі в сервіс WarehouseService, чергу amazon.notificationService.order через яку сервіс WarehouseService спілкується NotificationService і чергу amazon.webService.order через який NotificationService спілкується з WebService рис. 2.2.

Overview					Messages			Message rates				+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver	/ get	ack	
/	amazon.notificationService.order	classic	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	0.00/s	
/	amazon.orderService.order	classic	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	0.00/s	
/	amazon.webService.order	classic	D	idle	0	0	0					

Рис. 2.2 Черги повідомлень

## 2.2 Проектування бази даних для мікросервіса

База даних використовується в проекті буде використовувати реляційну модель.

Реляційна база даних – це тип бази даних, яка організовує дані в одну або кілька таблиць або відносин, кожна з яких має унікальну назву і



складається з набору рядків і стовпців. Дані в реляційній базі даних структуровані та організовані, що полегшує їх пошук, вилучення та управління.

У базі даних будуть оголошені такі таблиці ShopItems, ShoppingCarts, CartItems та CartItemShoppingCart, інфологічна модель бази даних виглядає даним чином рис. 2.3.

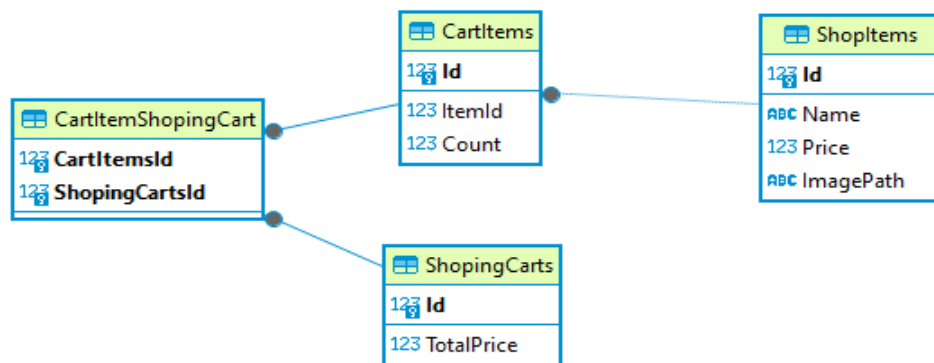


Рис. 2.3 Інфологічна модель бази даних

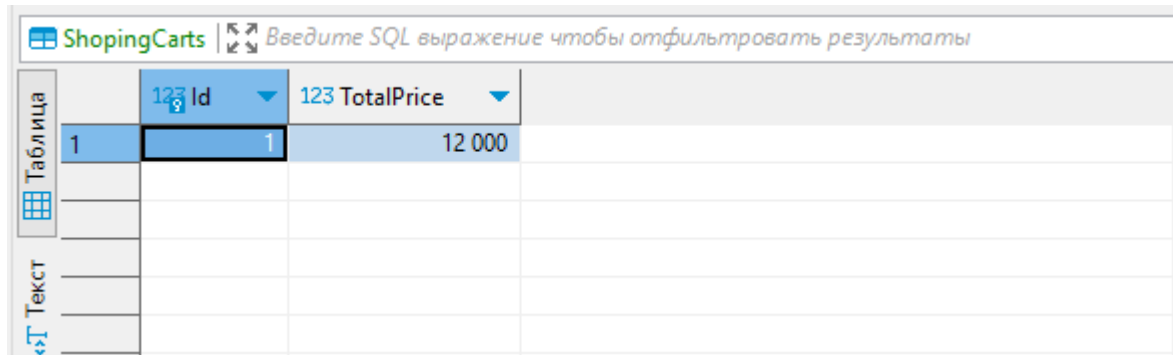
Розглянемо кожну таблицю та її призначення.

- ShopItems – таблиця в якій зберігаються всі товари які доступні для покупки на сайті, в таблиці є такі поля id який відповідає за унікальний номер за яким можна знайти товар в таблиці, Name Назва товару який буде видно користувачеві, Price ціна яка утановлена на конкретний товар і ImagePath зберігає в собі шлях до зображення на товар рис. 2.4.

ShopItems					
Введіть SQL вираження чтобы отфильтровать результаты					
Таблиця	123 Id	ABC Name	123 Price	ABC ImagePath	
1	1	I5-12400F	15 000	/images/i5-12400F.jpg	
2	2	R5-5600X	12 000	/images/5 5600X.jpg	
3	3	Athlon 200GE	6 000	/images/Athlon 200GE.png	
Текст					

Рис. 2.4 таблиця ShopItems

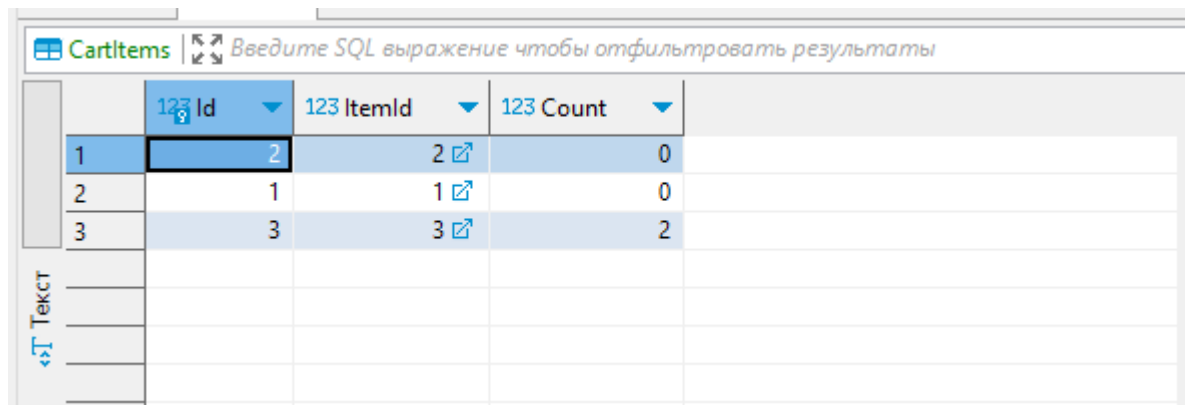
- ShoppingCarts – таблиця зберігає в собі кошика всіх користувачів в ній є поля id кошика і TotalPrice в якій показано на яку суму лежить товарів в кошику рис. 2.5.



	123 Id	123 TotalPrice
1	1	12 000

Рис. 2.5 таблиця ShoppingCarts

- CartItems – таблиця відповідає за кошик користувача в ній зберігаються всі товари який користувач відправив у кошик, в ній присутні такі поля, id унікальний ідентифікатор товару в кошику користувача, ItemId ідентифікатор який береться з таблиці ShopItems і Count кількість одиниць вибраного товару рис. 2.6.



	123 Id	123 ItemId	123 Count
1	2	2	0
2	1	1	0
3	3	3	2

Рис. 2.6 таблиця CartItems

- CartItemShoppingCart – зв'язкова таблиця яка створює сваязі між талицами CartItem і ShopingCart рис. 2.7.

Таблиця	123 CartItemsId		123 ShopingCartsId	
	1	1	1	1
	2	2	1	1
	3	3	1	1
Текст				

Рис. 2.7 таблиця CartItemShoppingCart

## 2.3 Проектування мікросервісів та міжпроцесної взаємодії

### 2.3.1 WebService

Тепер розглянемо кожен Сервіс більш детально почнемо з сервісу WebService який відповідає за обробку веб інтерфейсу користувачів, відправку потрібної інформації на сторінку, синхронізацію дій Користувача з базою даної, і відправкою іншим сервісам інформації про кошик.

Так як в проекті використовується патерн MVC структура сервісу буде виглядати даними чином рис. 2.8.

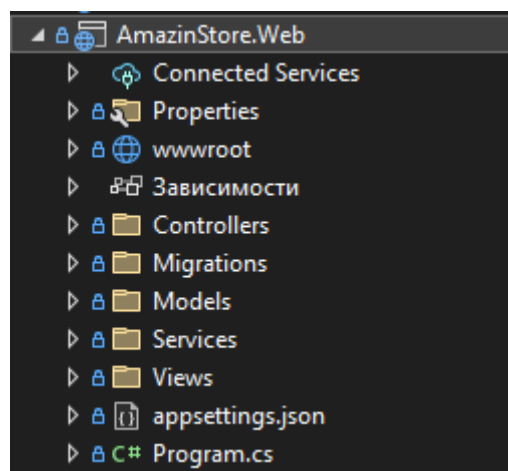


Рис. 2.8 структура сервісу WebService

структура сервісу Модель (Model)

Це основна логіка програми. Відповідає за дані, методи роботи з ними і структуру програми. Модель реагує на команди з контролера і видає інформацію та / або змінює свій стан. Вона передає дані в уявлення.

*Уявлення (View)*

Завдання компонента – візуалізація інформації, яку він отримує від моделі. View відображає дані на рівні інтерфейсу користувача. Наприклад, у вигляді таблиці або списку. Подання визначає зовнішній вигляд програми та способи взаємодії з ним.

### *Контролер (Controller)*

Забезпечує взаємодію з системою: обробляє дії користувача, перевіряє отриману інформацію і передає її моделі. Контролер визначає, як додаток буде реагувати на дії користувача. Також контролер може відповідати за фільтрацію даних і авторизацію.

Також в проєкті є ряд інших додаткових папок і файлів

`Program.CS` – цей файл є точкою входу в процес, він визначає клас `Program`, який ініціалізує і запускає хост з додатком.

`appsettings.json` – зберігає конфігурацію програми.

`Wwwroot` – вузол призначений для зберігання статичних файлів-зображень, скриптів javascript, файлів css і т. д., які використовуються додатком.

`Migrations` – папка в якій зберігаються міграції бази даних (процес зміни структури і вмісту бази даних з метою оновлення її версії, перенесення на іншу платформу або злиття з іншою базою даних).

`Services` – рівень послуг між контролером і моделлю, який служить для інкапсуляції бізнес-логіки, яка може бути в контролері. Сервіси покликані саме для того, щоб: приймати, завантажувати і зберігати будь-які типи даних з будь-якого типу зовнішнього джерела.

`Залежності` – папка в якій зберігаються всі залежності проєкту. Налаштування платформи, аналізатори і використовувані NuGet пакети.

*NuGet пакети* – це механізм, за допомогою якого розробники можуть створювати, передавати один одному та використовувати корисний код. Часто такий код розподілений по "пакетам", що включає скомпільований код (у вигляді бібліотек DLL) і інший вміст, необхідне використовувати ці пакети проєктам, на рисунку 2.9 оказані використовувані в проєкті NuGet пакети.








	<b>AutoMapper</b> автор: Jimmy Bogard A convention-based object-object mapper.	12.0.1
	<b>AutoMapper.Extensions.Microsoft.DependencyInjection</b> автор: Jimmy Bogard AutoMapper extensions for ASP.NET Core	12.0.1
	<b>Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation</b> автор: Microsoft Runtime compilation support for Razor views and Razor Pages in ASP.NET Core MVC.	7.0.13 8.0.0
	<b>Microsoft.EntityFrameworkCore</b> автор: Microsoft Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through...	7.0.14 8.0.0
	<b>Microsoft.EntityFrameworkCore.Design</b> автор: Microsoft Shared design-time components for Entity Framework Core tools.	7.0.14 8.0.0
	<b>Npgsql.EntityFrameworkCore.PostgreSQL</b> автор: Shay Rojansky, Austin Drenski, Yoh Deadfall PostgreSQL/Npgsql provider for Entity Framework Core.	7.0.11 8.0.0
	<b>RabbitMQ.Client</b> автор: VMware The RabbitMQ .NET client is the official client library for C# (and, implicitly, other .NET languages)	6.6.0 6.8.1

Рис. 2.9 Використовувані NuGet пакети

Діаграма класів сервісу WebService виглядає наступним чином рис. 2.10.

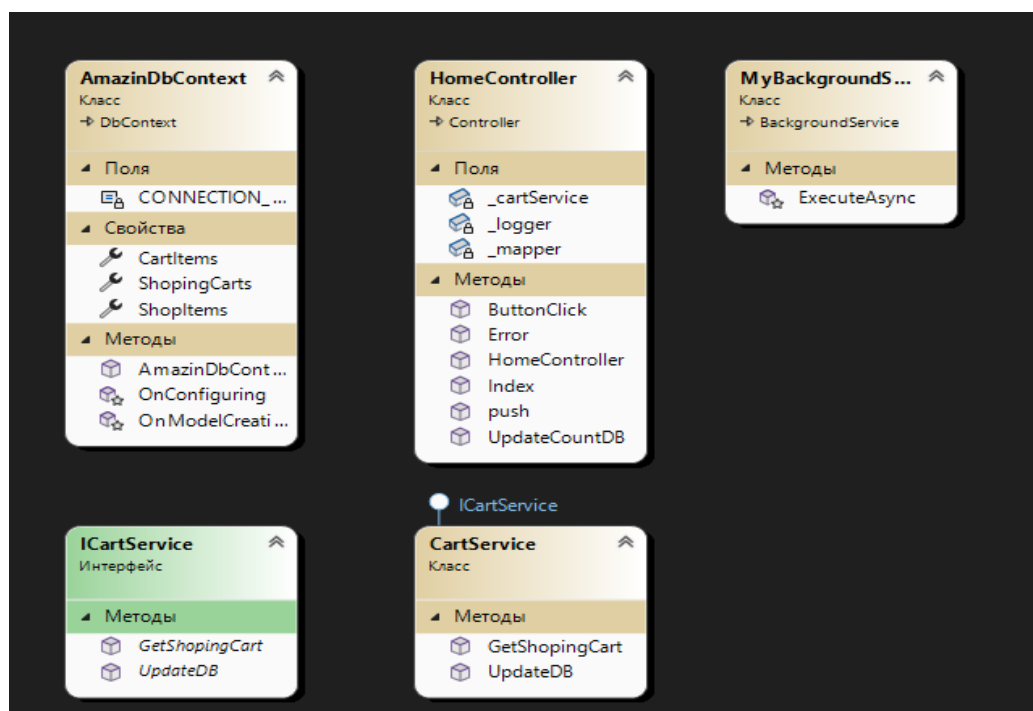


Рис 2.10 Діаграма класів сервісу WebService

Розглянемо кодову частину проекту.

Файл класу Program.cs є точкою входу в наш додаток і створює екземпляр IWebHost, на якому розміщується веб-додаток. Тут підключаються

всі основні класи. У проекті використовується компіляція Razor в реальному часі за це відповідає рядок

```
builder.Services.AddControllersWithViews().AddRazorRuntimeCompilation();
```

Підключається класи роботою з бд і її міграціями для цього використовується клас Background Service Для для реалізації довго виконується інтерфейсу `IHostedService`.

```
builder.Services.AddSingleton<ICartService, CartService>();  
builder.Services.AddHostedService<MyBackgroundService>();
```

А так само прописаний мапінг для роботи з DTO об'єктами

```
builder.Services.AddAutoMapper(typeof(AppMappingProfile));
```

мапінг – це процес відображення та перетворення даних з різних систем для їх інтеграції в цільову систему або базу.

`WebHost` використовується для створення екземплярів `IWebHost`, `IWebHostBuilder` та `IWebHostBuilder`, які мають попередньо налаштовані параметри. Метод `Create Builder ()` створює новий екземпляр `WebApplicationBuilder`.

Метод `Build()` повертає екземпляр `IWebHost`, а `Run ()` запускає веб-програму до її повної зупинки.

Контролер в мікро сервісі використовується один `HomeController` контролери успадковуються від класу `Controller`, так само він має методи для отримання параметрів і даних запиту, вони обробляють їх і формують відповіді у вигляді результату дії. Результат дії-це той об'єкт, який повертається методом після обробки запиту. У контролері проекту є такі методи.

Метод `index()` який отримує дані з бази даних і відправляє їх на сторінку користувачеві при відкритті сайту.

```
public IActionResult Index()  
{
```

```
    var cart = _cartService.GetShoppingCart();
```

```
return View(cart);
}
```

Button Click() метод який викликається httpPost запитом при оформленні товару і викликає метод Push().

[HttpPost]

```
public IActionResult ButtonClick()
{
    push();
    return Ok("12");
}
```

Update Count DB () Метод який викликається httpPost запитом при зміні кількості вибраних товарів на стрниці користувача, служить для синхронізації вибраних товарів Користувачем з базою даних.

Push () Метод роботи з RebbitMQ при натисканні користувачем кнопки оформити покупку бере з бази даних інформацію про покупки переносить їх в DTO об'єкт сереалізує в рядок переобразує в масив байтів і відправляє в брокер повідомлень.

```
public void Push()
{
    var factory = new ConnectionFactory() { HostName = "localhost" };
    using (var connection = factory.CreateConnection())
    {
        using (var chanel = connection.CreateModel())
        {
            ShopingCart model = _cartService.GetShopingCart();

            var modelDto = _mapper.Map<ShopingCartDTO>(model);

            var message = JsonSerializer.Serialize(modelDto, new
```

```
JsonSerializerOptions()
```

```

        {
            ReferenceHandler =
ReferenceHandler.IgnoreCycles
        });
        var body = Encoding.UTF8.GetBytes(message);

        chanel.BasicPublish("OrderExchange",
"amazon.orderService.order", mandatory: false, null, body);
    }
}
}

```

Розглянемо код більш детально

```
var factory = new ConnectionFactory() { HostName = "localhost" };
```

Тут прописуються параметри підключення до брокера повідомлень, так як брокер перебувати на тій же машині на якій працюють сервіси рдесь вказано localhost далі йде підключеніє до RabbitMQ

```
var connection = factory.CreateConnection()
```

```
var chanel = connection.CreateModel()
```

Потім використовуючи метод Get Shopping cart класу ShoppingCart йде отримання об'єкта в якому зберігатися вся інформація з бази даних про товари в кошику користувача

```
ShopingCart model = _cartService.GetShopingCart();
```

Далі цей об'єкт ми переводимо в модель DTO для відправки тільки потрібної нами інформації.

```
var modelDto = _mapper.Map<ShopingCartDTO>(model);
```

І після серіалізація цього об'єкта в рядок

```
var message = JsonSerializer.Serialize(modelDto, new
JsonSerializerOptions()
{

```



```
ReferenceHandler = ReferenceHandler.IgnoreCycles
```

```
});
```

Після переведення цього рядка в масив байтів

```
var body = Encoding.UTF8.GetBytes(message);
```

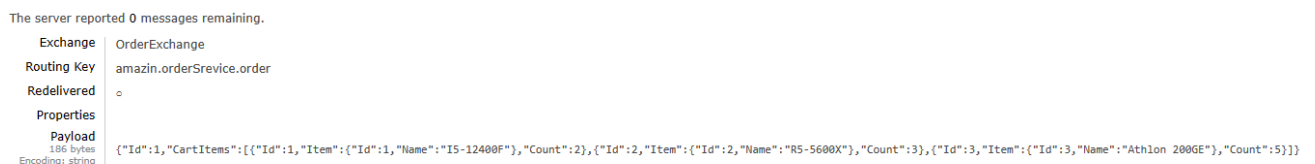
Та публікація інформації в брокер повідомлень

```
chanel.BasicPublish("OrderExchange", "amazin.orderSrevice.order",  
mandatory: false, null, body);
```

Де

- OrderExchange – черга в яку потрібно відправити повідомлення
- amazin.orderSrevice.order-гілка в яку потрібно покласти повідомлення.
- mandatory: false, null-параметри читання
- body-саме повідомлення яке потрібно оптравить

Повідомлення в брокері виглядає даним чином рисунок 2.9



The server reported 0 messages remaining.

Exchange	OrderExchange
Routing Key	amazin.orderSrevice.order
Redelivered	0
Properties	
Payload	{ "Id": 1, "CartItems": [ { "Id": 1, "Item": { "Id": 1, "Name": "IS-12400F", "Count": 2 }, { "Id": 2, "Item": { "Id": 2, "Name": "R5-5600X", "Count": 3 }, { "Id": 3, "Item": { "Id": 3, "Name": "Athlon 200GE", "Count": 5 } ] }
186 bytes	
Encoding: string	

Рис. 2.9

Також контролер має конструктор, через який за допомогою механізму dependency injection передається сервіс ILogger, який використовується для логгування.

```
private readonly ICartService _cartService;  
private readonly ILogger<HomeController> _logger;  
private readonly IMapper _mapper;
```

```

public HomeController(ILogger<HomeController> logger, ICartService
cartService, IMapper mapper)
{
    _logger = logger;
    _cartService = cartService;
    _mapper = mapper;
}

```

Як говорилося вище для передачі об'єктів використовується патерн DTO розглянемо що він із себе представляє.

Data Transfer Object – це об'єкт, який використовується для інкапсуляції даних та їх надсилання з однієї підсистеми програми в іншу [6].

DTO найчастіше використовуються рівнем послуг у n-рівневому додатку для передачі даних між ним та рівнем інтерфейсу користувача. Основна перевага тут полягає в тому, що він зменшує обсяг даних, які необхідно передавати по мережі в розподілених додатках. Вони також роблять чудові моделі в шаблоні MVC.

Іншим застосуванням DTO може бути інкапсуляція параметрів для викликів методів. Це може бути корисно, якщо метод приймає більше чотирьох або п'яти параметрів.

Класи DTO це ті ж класи для роботи і зберігання бд але тільки з потрібними для передачі полями вони мають таке ж ім'я але тільки з приставкой DTO рисунок 2.11.

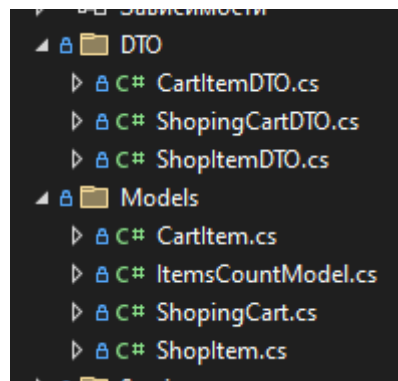
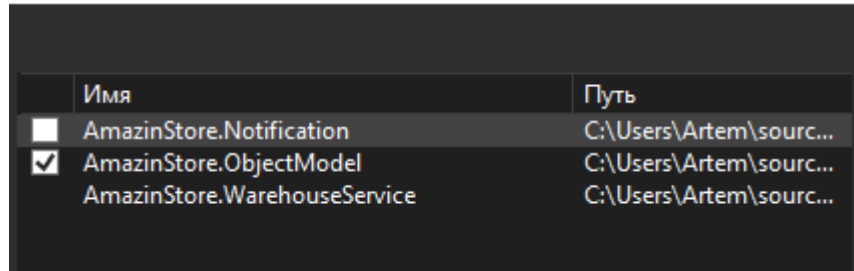


Рис. 2.11 DTO моделі для передачі даних

Так як DTO об'єкти і моделі бази даних однакові у всіх серисах він винесений в окремий проект ObjectModel до якого виставлені залежності для кожного сервісу тут ханяться класи DTO і класи для роботи з базою даних рисунок 2.12.



	Имя	Путь
<input type="checkbox"/>	AmazinStore.Notification	C:\Users\Artem\sourc...
<input checked="" type="checkbox"/>	AmazinStore.ObjectModel	C:\Users\Artem\sourc...
<input type="checkbox"/>	AmazinStore.WarehouseService	C:\Users\Artem\sourc...

Рис. 2.12 залежності сервісу WebService від ObjectModel

У цьому проекті так само лежать класи які взаємодіють з базою даних PostgreSQL. Класи мають такі ж назви як і таблиці в базі даних, Cart Items, ShopingCart, ShopItem, а так само клас ItemsCountModel для роботи з поточним екземпляром кошика.

У класі CartItems оголошені такі рядки

```
public int Id { get; set; }
public ShopItem Item { get; set; }
public int Count { get; set; }
public List<ShopingCart> ShopingCarts { get; set; }
```

У класі Shoppingcart оголошені такі рядки

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

public float TotalPrice { get; set; }
public List<CartItem> CartItems { get; set; }
```

У класі Shop Items оголошені такі рядки

```

public int Id { get; set; }
public string Name { get; set; }
public float Price { get; set; }
public string ImagePath { get; set; }

```

Особливістю PostgreSQL є в тому що при додаванні в ці класи нових рядків створиться міграція яка додати стовпець з такою ж назвою як ми вказали в новому полі, це дозволяє працювати з кожним рядком в базі даних як з об'єктом цього класу.

Тепер розглянемо папку Services тут об'явлені класи для роботи з базою даних, такі як AmazonDbContext Клас в которм прописані параметри для підключення з базою даних і параметри. BackgroundService клас для обробки міграцій з базою даних, CartService і ICartService клас і інтерфейс для отримання з бази даних кошика користувача і внесення змін в базу даних структуру папки показана на рис. 2.13.

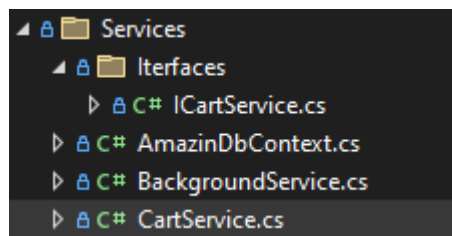


Рис. 2.13 структура папки Services

Migrations у папці міграцій лежать файли пов'язані з міграцією бази даних.

Міграція бази даних-це процес зміни структури та вмісту бази даних з метою оновлення її версії, перенесення на іншу платформу або злиття з іншою базою даних. Це може включати додавання нових таблиць, зміну існуючих таблиць, видалення таблиць або зміну типів даних. Міграція може знадобитися при оновленні версії СУБД або зміні платформи, на якій працює база даних. Структура папки виглядає даними чином рис. 2.14.

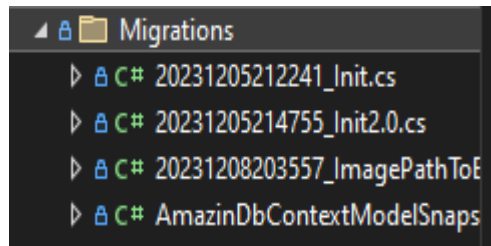


Рис 2.14 структура папки міграцій

Далі розглянемо папку Views тут знаходяться уявлення. Головним поданням являється файл `_Layout.cshtml` тут розписана розмітка яка буде повторюватися на всіх сторінках магазину в блоці `main` немає розмітки але є рядок `@RenderBody()`, `@` означає що код поле неї являється `c#` кодом це особливість Razor синтаксису.

Razor – це синтаксис розмітки для введення коду на основі .NET на веб-сторінки. Синтаксис Razor складається з розмітки Razor , `C#` та HTML. Файли, Razor містять зазвичай `.cshtml` розширення файлу.

У `RenderBody ()` буде повернуто унікальний для кожної сторінки код у мене використовується тільки одна сторінка з цього сюди завжди вставляється код з файлу `Index.cshtml` тут розмічена сторінка кошика магазину, в самому верху об'явлена рядок `@model ShoppingCart` вона позначає що уявлення очікує отримати з контролера модель класу `ShoppingCart` з якого сторінка потім буде брати потрібні дані для відображення їх користувачеві. Так як у користувача може перебувати Різне колісчество товарів то розмітка створюється динамічно на основі кількості товару в кошику користувача для цього в коді використовується Razor синтаксис `@ {for ()}` в цикл `for` виконується в залежності від кількості товару в кошику користувача, він динамічно створює блок товару і заповнює його інформацією взята з пердаваною моделі.

```
for (int i = 0; i < Model.CartItems.Count; i++)
```

```
{
```

```
    <div class="item">
```

```
        
```

```
        <p class="text">
```

```

                @Model.CartItems[i].Item.Name
            </p>
            <div class="counterBlock">
                <button class="buttonDown"
id="@Model.CartItems[i].Id" OnClick="down(this)">-</button>
                <div class="counterPanel"
name="productCount"
id="@Model.CartItems[i].Id">@Model.CartItems[i].Count</div>
                <button class="buttonUP"
id="@Model.CartItems[i].Id" OnClick="add(this)">+</button>
            </div>
            <div class="price">
                Ціна: @Model.CartItems[i].Item.Price
            </div>
        </div>
    }

```

Після виконання всього коду сторінка виглядає даним чином рисунок 2.15.

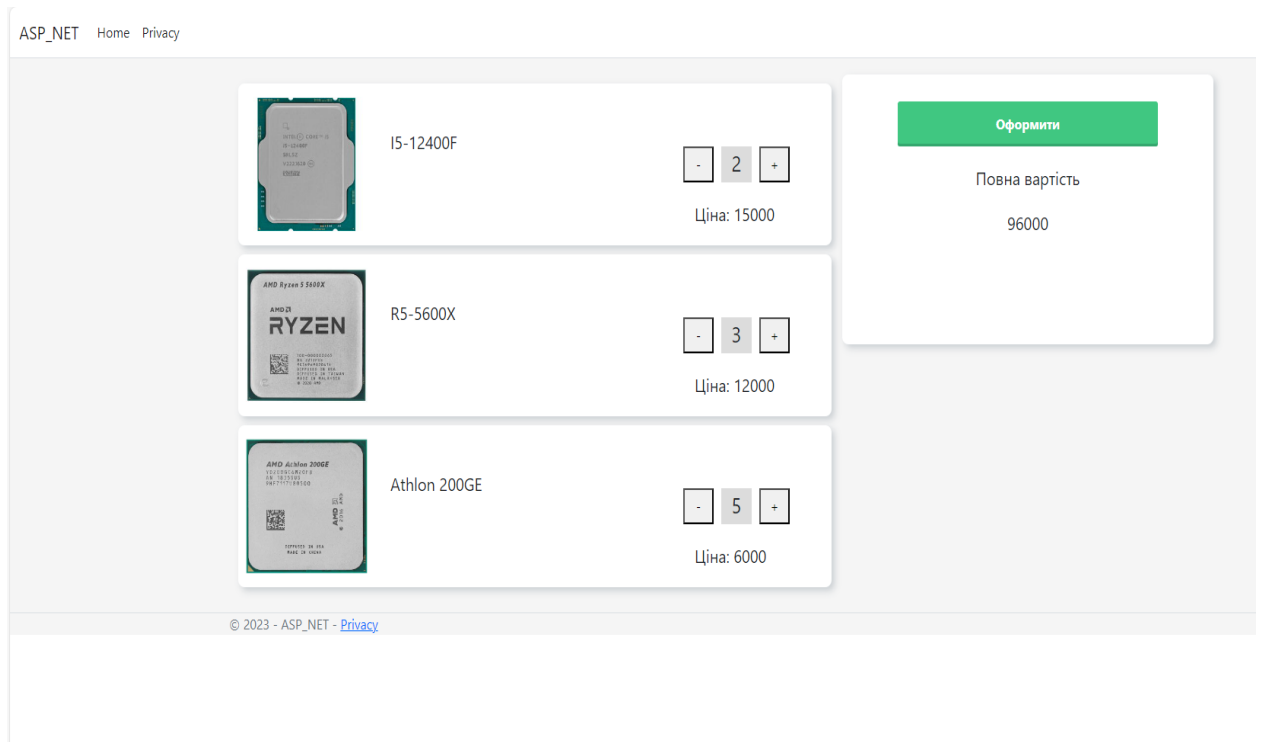


Рис 2.15 кошик користувача

Стилі які використовуються на сайт лежать в папці wwwroot в розділі CSS. Так само в цій папці є розділ js в якій лежать javascript файли які відповідають за обробку програмного доступу до об'єктів додатків. Файл Site.js обробляє натискання кнопок і відправку Post розглянемо його.

В даному javascript файлі оголошені такі функції OnLoad () функція яка потрібна для того щоб упевнитися що сторінка була повністю завантажена, це потрібно для правильного і безперебійного виконання скриптів на сайті.

Далі код для обробки натискання кнопки покупки яка відправляє post запит на сервер для виконання методу ButtonClick () в контролері для відправки інформації в брокер повідомлень.

```
let elem = document.getElementById('12');
```

```
elem.addEventListener("click", () => {  
    $.post("Home/ButtonClick", () => {  
  
    });  
});
```

Так само оголошені функції роботи роботи з базою даних, при зміні кількості товарів для покупки викликаються функції add і down, вони викликають функцію setCurrCount яка відправляє пост запит на сервер з інформацією про те на якому товарі було змінено кількість і поточне число товару для оновлення цієї інформації бази даних, цей запит викликає функцію UpdateCountDB в контролері та ж викликає метод UpdateDB в класі CartService і оновлює інформацію в базі, після цього повертається інформація про вдале изменени і викликається функція updateInfo в javascript коді яка змінює кількість товарів на сторінці користувача.

### 2.3.2 WareHouseService

WareHouseService - цей сервіс відповідає за прийняття замовлення від сервісу Web на склад, відстеження статусу замовлення, і відправкою її на сервіс повідомлень.

Цей сервіс не використовує патерн MVC, в ньому є файл Program.cs який є головною точкою входу і папка Services з класом StartupBackgroundService цей клас підключений в Program як клас фонового завдання який запускається разом з додатком. Тут і прописана вся логіка діаграма класу показана на рис 2.16.

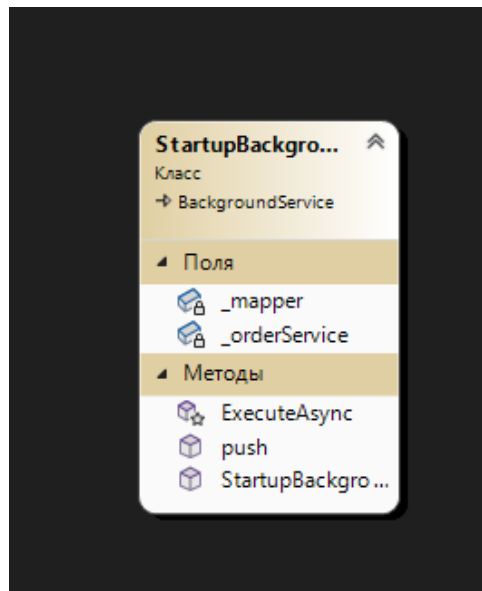


Рис 2.16. Діаграма сервісу WarehouseService

Розглянемо клас StartupBackgroundService

Тут прописані рядки логгирования

```
public StartupBackgroundService(IOrderService orderService, IMapper
mapper)
{
    _orderService = orderService;
    _mapper = mapper;
}
```



Функція `ExecuteAsync` в якій виконується вся логіка тут йде підключення `RebbitMQ` для отримання даних з брокера повідомлень, код шукає потрібну їй інформацію в черзі `amazon.orderService.order` якщо знаходить отримує її від туди, так як в першому сервісі ми переносили все це в масив байт, а перед цим переносили в модель `DTO` то тут нам потрібно зробити зворотну операцію, після отримання масиву байтів ми переводимо їх назад в рядок

```
var message = Encoding.UTF8.GetString(body.ToArray());
```

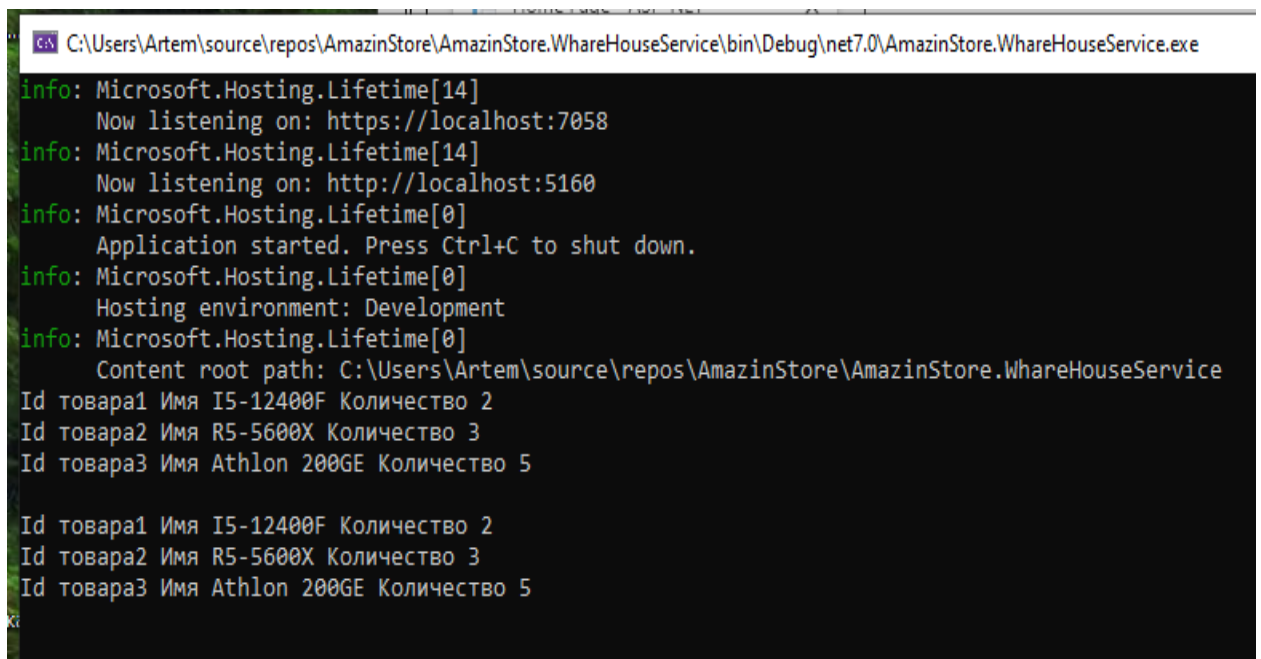
далі десериалізуємо назад в об'єкт класу `ShoppingCartDTO`

```
JsonConvert.DeserializeObject<ShoppingCartDTO>(message)
```

І після цього переносимо з об'єкта класу `DTO` в звичайний об'єкт класу `ShoppingCart`

```
ShoppingCart cart = _mapper.Map<ShoppingCart>();
```

Далі в циклі виводиться інформація про отримані товари на консоль додатка рис 2.17.



```
C:\Users\Artem\source\repos\AmazonStore\AmazonStore.WarehouseService\bin\Debug\net7.0\AmazonStore.WarehouseService.exe
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7058
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5160
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Artem\source\repos\AmazonStore\AmazonStore.WarehouseService
Id товара1 Имя I5-12400F Количество 2
Id товара2 Имя R5-5600X Количество 3
Id товара3 Имя Athlon 200GE Количество 5

Id товара1 Имя I5-12400F Количество 2
Id товара2 Имя R5-5600X Количество 3
Id товара3 Имя Athlon 200GE Количество 5
```

Рис 2.17 висновок отриманої інформації з брокера в консоль

Потім йде обробка статусів виконання складання замовлень, при зміні статусу замовлення викликається метод `Push()` в якому прописаний код підключення до брокера повідомлень і отпраки на сервіс повідомлень статусу

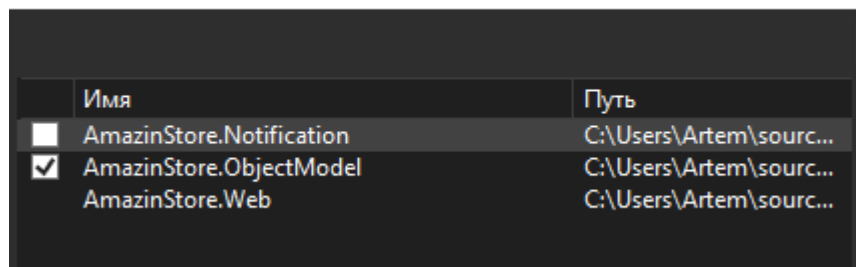
виконання замовлення, метод приймає текст який потрібно Відправити користувачеві в Особистий кабінет або на телефон.

```
push("Статус: Сборка");  
Thread.Sleep(1000);  
push("Статус: Отправлено");
```

далі відбувається переклад цього тексту в масив байтів і відправляється в чергу `amazon.notificationService.order` для отримання його в `NotificationService`.

Для цього сервісу так само виставлені залежності від проекту `ObjectModel` так як в проекті використовуються ті ж класи рисунок 2.18.

store.WarehouseService



	Имя	Путь
<input type="checkbox"/>	AmazonStore.Notification	C:\Users\Artem\sourc...
<input checked="" type="checkbox"/>	AmazonStore.ObjectModel	C:\Users\Artem\sourc...
	AmazonStore.Web	C:\Users\Artem\sourc...

Рис. 2.18 залежність Warehouse Service від ObjectModel

NuGet пакети в цьому Сервісі використовуються такі ж як в сервісі `WebService`.

### 2.3.3 NotificationService

Завдання `Notification Service` відправлення повідомлень Користувачеві про статус посилки.

Його структура аналогічна структурі `Warehouse Services`, а саме головний файл `Program.CS`, папка `Service` з класом `StartupBackgroundService` де прописана бізнес-логіка і папка з залежностями і NuGet пакетами, діаграма класів мікросервісу виглядає наступним чином рис 2.19.

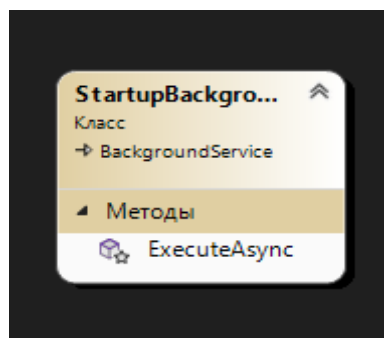


Рис 2.19 діаграма класів сервісу NotificationService

У цьому проєкті прописаний код отримання з брокера повідомлень інформації про статус посилки і відправки його користувачеві (так як проєкт створений для навчання у нас це просто висновок повідомлення на консоль).

Код отримання повідомлення з брокера аналогічний придедущим сервісів тільки так як ми з брокера отримуємо вже готову рядок яку потрібно послати користувачеві тут нам не потрібно передавати ніяких об'єктів, так що десеаріалізації тут ніякої немає і підключати цей сервіс до проєкту ObjectModel на поточний момент немає необхідності.

## 2.5. Тестування проєкту

Тепер після того як ми розібралися в структурі проєкту і того як він працює зсередини можна спробувати скористатися ним і протестувати.

Перед запуском проєкту потрібно запустити контейнер в Docker який відповідає за базу даних PostgreSQL інакше при спробі підключити і провести міграцію код видасть помилку рисунок 2.20.

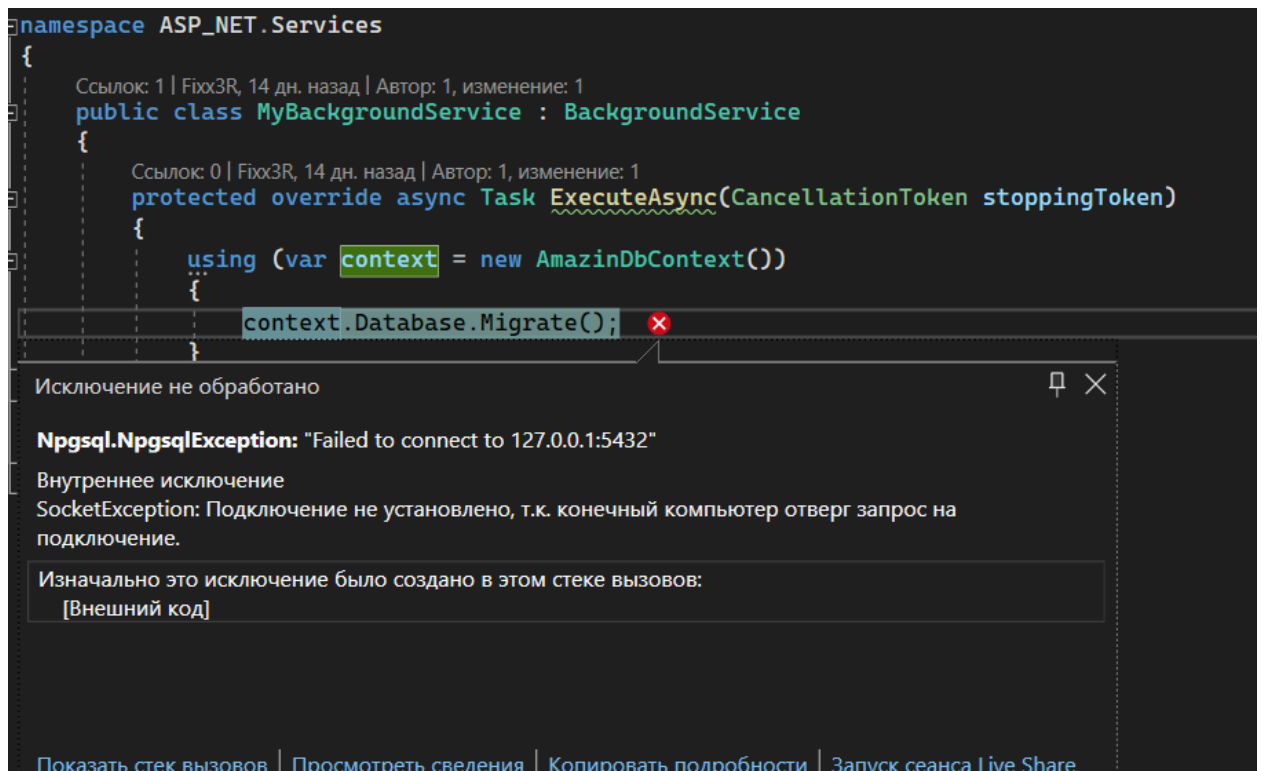


Рис. 2.20 Помилка неможливості провести міграцію з БД

Після запуску контейнера рисунок 2.21.

<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	<a href="#">amazon-postgres</a> 70ae00350923	<a href="#">postgres</a>	Running	N/A	<a href="#">5432:5432</a>	0 seconds ago	
	<a href="#">awesome-solomon</a>						

Рис. 2.21 запущений контейнер БД

Відкривається сайт з вікном кошика Користувача, а також 3 консольних вікна відповідають за свій сервіс рисунок 2.22.

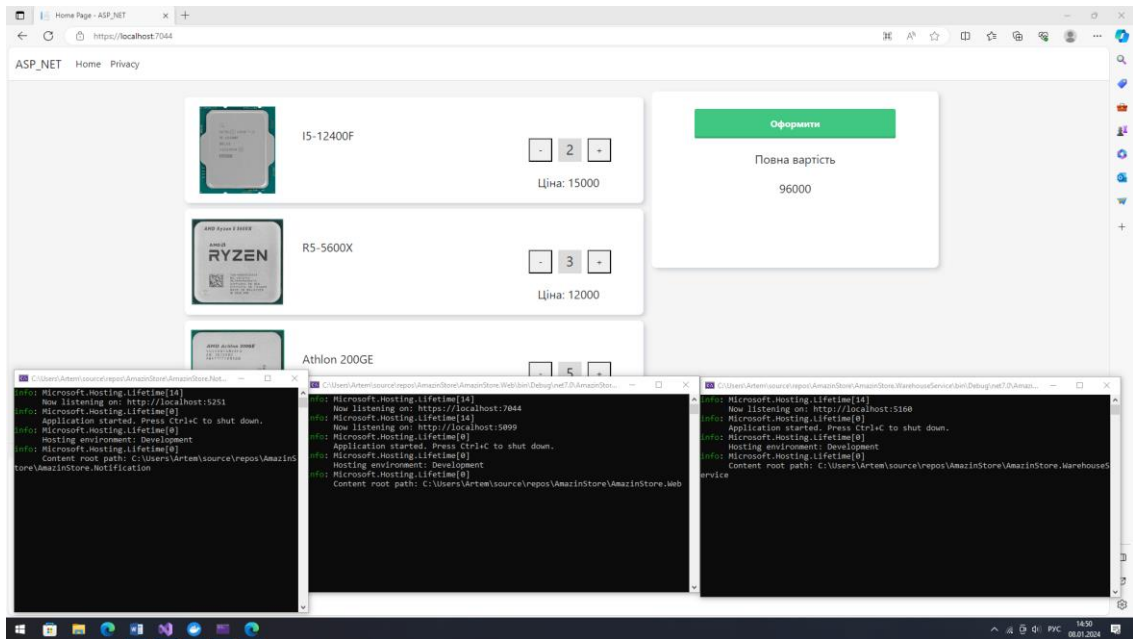


Рис. 2.22 Сторінка кошика користувача і консолі сервісів

Після відкриття сайту можна вибрати кількість потрібного товару, поки що немає можливості видаляти і додавати нові товари але є можливість вибрати їх кількість. Після того як користувач вибрав потрібну йому кількість товарів їх загальна ціна виводиться знизу під кнопкою оформлення товару.

Після натискання на кнопку оформити WebService відправляє інформацію на RabbitMQ звідки її бере WarehouseService і обробляє, виводить на консоль і емулює процес оформлення товару рисунок 2.23.

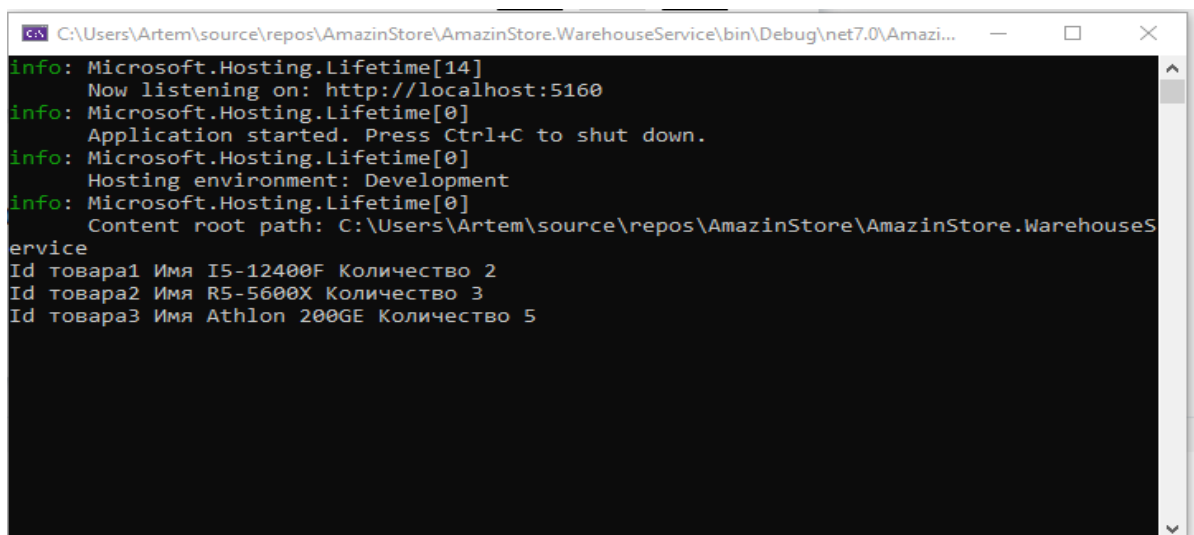
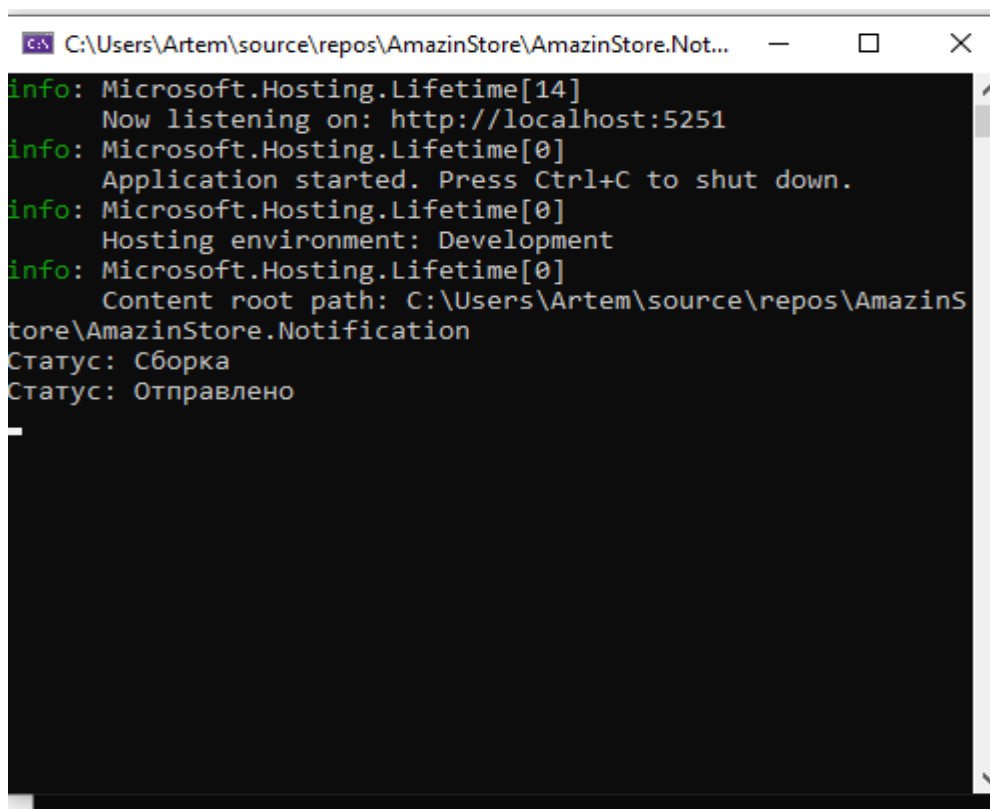


Рис. 2.23 виведення необхідних для складання товару в сервісі WarehouseService

Далі в міру зміни статусу посилки склад відправляє інформацію на NotificationService який відправляє повідомлення користувачеві рисунок 2.24.



```
C:\Users\Artem\source\repos\AmazinStore\AmazinStore.Not...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5251
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Artem\source\repos\AmazinStore\AmazinStore.Notification
Статус: Сборка
Статус: Отправлено
```

Рис 2.24 NotificationService

Таким чином завдяки реалізації мікро сервісної архітектури і використанням брокера повідомлень вдається швидко і надійно передавати інформацію між різними процесами і службами які можуть перебувати на різних серверах. Це дозволить розвивати ці сервіси різними командами незалежно один від одного.

Реалізація яка показана в даному проекті може бути використана при створенні будь-якого інтернет магазину, додати можливість вибирати товар, систему реєстрації, Можливість вибирати поравівшійся товар в закладки, для WarehouseService зробити зрозумілий веб інтерфейс, а NotificationService підключити до телефонії.

## Висновки до розділу 2

У другому розділі були розглянуті програмні інструменти та реалізація мікро сервісної архітектури проекту. Була розглянута платформа контейнеризації Docker, її роль в проекті інтернет магазину, розглянуто сутність контейнеру і контейнеризації. Розглянута СУБД PostgreSQL, виявлені її особливості переваги, та недоліки, а саме те що PostgreSQL добре підходить для складних запитів і роботи з великими обсягами інформації (Big Data), хоч і поступається в швидкості в роботі іншим СУБД. Під час дослідження було обрано архітектурні рішення проекту (за допомогою патерну Model-View-Controller), обрано середовище розробки на основі технології ASP.NET Core.

На підставі проведеного аналізу розроблена архітектурна модель проекту, до склау якої входить три сервіси: WebService, Warehouse, NotificationService. Також смодельована міжпроцесна взаємодія типу асинхронна черга (FIFO) що означає першим прийшов - першим вийшов, це було зроблено за допомогою брокера повідомлень RabbitMQ. Виконано моделювання та проектування бази даних на основі СУБД PostgreSQL.

На підставі проектних рішень було розроблено проект інтернет магазину , який реалізує асинхронну модель взаємодії між сервісами. Виконане тестування сервісів, розглянута їх взаємодія. Розібрані варіанти подальшого розвитку проекту.

## ВИСНОВКИ

В рамках даного дослідження було розглянуто архітектурну модель мікросервісів та форми міжпроцесної взаємодії.

У першій частині даної роботи було проведено теоретичне обґрунтування переваг мікросервісної архітектури в порівнянні з монолітами. Було проаналізовано актуальність використання мікросервісної архітектури, розглянуто їх характеристики, особливості та перспективи використання.

Під час дослідженні міжпроцесної взаємодії було проаналізовано синхронну та асинхронну модель, виділено особливості кожної з них.

Встановлено, що асинхронна взаємодія має більш гнучку архітектуру вона може продовжувати роботу після відправки запиту другого сервісу не зупиняючи повністю виконання коду при очікуванні відповіді, така система має більшу відмовостійкість і легше масштабується однак використання асинхронного взаємодії вимагає більш складної реалізації так як необхідно обробляти асинхронні відповіді і управляти станом запитів, а так само налагодження асинхронної взаємодії може бути складніше через розподіл запитів і відповідей в часі.

У синхронному взаємодії клієнтський мікросервіс блокується і призупиняє свою роботу при очікуванні відповіді від іншого сервісу, це може стати вузьким місцем якщо синхронні виклики виконуються послідовно, проте такий підхід являється більш простим в реалізації і дозволяє легко відстежувати і управляти послідовністю виконання операцій.

Але найчастіше грамотне поєднання синхронної та асинхронної взаємодії може бути найбільш ефективним.

Для подальшої реалізації асинхронної моделі взаємодії проведено аналіз брокерів повідомлень RabbitMQ, Apache Kafka та Amazon Simple Queue Service (SQS), виявлені плюси і мінуси кожного брокера, також було досліджено доцільність застосування кожного виду брокерів в залежності від умов та вимог.



На підставі проведеного аналізу розроблена архітектурна модель проекту, до складу якої входить три сервіси: WebService, WareHouse, NotificationService. Також смодельована міжпроцесна взаємодія типу асинхронна черга (FIFO) що означає першим прийшов - першим вийшов, це було зроблено за допомогою брокера повідомлень RabbitMQ. Виконано моделювання та проектування бази даних на основі СУБД PostgreSQL.

На підставі проектних рішень було розроблено проект з 3-х мікросервісів, який реалізує асинхронну модель взаємодії між сервісами завдяки брокеру повідомень. Виконане тестування сервісів, розглянута їх взаємодія.

Проведене дослідження не вичерпує всі поставлені питання, тому що як було виявлено у кожній моделі взаємодії є свої сильні та слабкі міста, тому в подальшому є можливість перейти на гібридну модель де використовуються обидві моделі взаємодії, грамотна реалізація такої моделі дозволить використовувати сильні сторони кожної моделі.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Марченко О.О Переваги та недоліки використання мікросервісної архітектури при роз-робці програмного забезпечення. URL: <https://conf.ztu.edu.ua/wp-content/uploads/2019/12/31-1.pdf> (дата звернення 20.10.23).
2. A Guide for Choosing the Best API Gateway. URL: <https://www.atatus.com/blog/a-guide-for-choosing-the-best-api-gateway/#what-is-an-api-gateway> (дата звернення 22.10.23).
3. An introduction to RabbitMQ – What is RabbitMQ? URL: <https://www.erlang-solutions.com/blog/an-introduction-to-rabbitmq-what-is-rabbitmq/> (дата звернення 06.11.23).
4. ASP.NET Core: What Is It? URL: <https://dev.to/thanhtr99270163/aspnet-core-what-is-it-11f0> (дата звернення 08.11.23).
5. ASP.NET overview URL: <https://learn.microsoft.com/en-us/aspnet/overview> (дата звернення 02.12.23)
6. Breaking With Filing Cabinets: The History of PostgreSQL. URL: <https://learnsql.com/blog/history-of-postgresql/> (дата звернення 25.12.23).
7. Communication in a microservice architecture. URL: <https://learn.microsoft.com/en-us/dotnet/architecture/>
8. CSS Tutorial URL: <https://www.w3schools.com/css/default.asp> (дата звернення 18.11.23).
9. DBeaver Documentation. URL: <https://dbeaver.com/docs/dbeaver/#about-dbeaver> (дата звернення 27.11.23)
10. Dependency Management in Visual Studio: NuGet and Beyond URL: <https://fossa.com/blog/dependency-management-visual-studio-nuget-beyond/> (дата звернення 03.12.23)
11. Getting started with PostgreSQL on Docker. URL: <https://www.sqlshack.com/getting-started-with-postgresql-on-docker/> (дата звернення 25.11.23)

12. Getting to Know CSS URL: <https://learn.shayhowe.com/html-css/getting-to-know-css/> (дата звернення 22.11.23).
13. How to open two Solution in one Visual Studio IDE? URL: <https://stackoverflow.com/questions/21643063/how-to-open-two-solution-in-one-visual-studio-ide> (дата звернення 05.12.24)
14. HTML For Beginners URL: <https://html.com/> (дата звернення 18.11.23).
15. HTTP Methods URL: <https://http.dev/methods> (дата звернення 23.11.23).
16. HTTP Protocol and HTTP Methods URL: <https://dotnettutorials.net/lesson/http-protocol/> (дата звернення 23.11.23).
17. HTTP Request Methods URL: [https://www.w3schools.com/TagS/ref\\_httpmethods.asp](https://www.w3schools.com/TagS/ref_httpmethods.asp) (дата звернення 23.11.23).
18. Install and manage packages in Visual Studio using the NuGet Package Manager URL: <https://learn.microsoft.com/en-us/nuget/consume-packages/install-use-packages-visual-studio> (дата звернення 03.12.23)
19. Manage PostgreSQL Database Server Using DBeaver. URL: <https://techviewleo.com/manage-postgresql-database-server-using-dbeaver/#:~:text=DBeaver%20is%20a%20database%20management,developers%2C%20SQL%20programmers%20and%20analysts> (дата звернення 28.11.23)
20. Microservices architecture design. URL: <https://learn.microsoft.com/en-us/azure/architecture/microservices/> (дата звернення 01.11.23).
21. Microservices vs. monolithic architecture URL: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith> (дата звернення 20.10.23).
22. Microservices, SOA, and APIs: Friends or enemies? URL: [https://developer.ibm.com/tutorials/1601\\_clark-trs/](https://developer.ibm.com/tutorials/1601_clark-trs/) (дата звернення 27.10.23).

- microservices/architect-microservice-container-applications/communication-in-microservice-architecture (дата звернення 22.10.23).
23. Microservices: understanding what it is and its benefits. URL: <https://www.atlassian.com/microservices> (дата звернення 01.11.23).
24. MVC Architecture – Detailed Explanation. URL: <https://www.interviewbit.com/blog/mvc-architecture/> (дата звернення 15.11.23).
25. MVC architecture: Explained with an example. URL: <https://dev.to/ridhikgovind/mvc-architecture-explained-with-an-example-9od> (дата звернення 13.11.23).
26. MVC Architecture: Model View Controller URL: <https://www.techgeekbuzz.com/blog/mvc-architecture/> (дата звернення 16.11.23).
27. RabbitMQ, get Started. URL: <https://www.rabbitmq.com/getstarted.html> (дата звернення 02.11.23).
28. Run PostgreSQL with Docker. URL: <https://fullstackcode.dev/2022/01/17/run-postgresql-with-docker/> (дата звернення 26.11.23)
29. The Model View Controller Pattern – MVC Architecture and Frameworks Explained. URL: <https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/> (дата звернення 15.11.23).
30. Web Services Architecture / W3C Working Group Note 1. URL: <http://www.w3.org/TR/ws-arch/> (дата звернення 27.10.23).
31. What are microservices ? URL: <https://microservices.io/> (дата звернення 28.10.23).
32. What is a Data Transfer Object (DTO) ? URL: <https://stackoverflow.com/questions/1051182/what-is-a-data-transfer-object-dto> (дата звернення 26.10.23).

33. What is a Data Transfer Object (DTO)? URL: <https://stackoverflow.com/questions/1051182/what-is-a-data-transfer-object-dto> (дата звернення 25.11.23)
34. What is ASP.NET Core? URL: <https://umbraco.com/knowledge-base/asp-dot-net-core/> (дата звернення 09.11.23).
35. What is DBeaver? URL: <https://database.guide/what-is-dbeaver/> (дата звернення 26.11.23)
36. What is Docker? URL: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined> (дата звернення 28.10.23).
37. What is Docker? Why is it important and necessary for developers? URL: <https://dev.to/amoniacou/what-is-docker-why-is-it-important-and-necessary-for-developers-part-i-39e5> (дата звернення 08.11.23).
38. What is JavaScript? A Definition of the JS Programming Language URL: <https://www.freecodecamp.org/news/what-is-javascript-definition-of-js/> (дата звернення 22.11.23).
39. What is PostgreSQL? URL: <https://www.postgresql.org/about/> (дата звернення 25.11.23)
40. What is RabbitMQ ? URL: <https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html>. (дата звернення 03.11.23).

## ДОДАТКИ

### 1. Webservice клас HomeController.cs

```
namespace ASP_NET.Controllers
{
    public class HomeController : Controller
    {
        private readonly ICartService _cartService;
        private readonly ILogger<HomeController> _logger;
        private readonly IMapper _mapper;

        public HomeController(ILogger<HomeController> logger, ICartService
cartService, IMapper mapper)
        {
            _logger = logger;
            _cartService = cartService;
            _mapper = mapper;
        }

        public IActionResult Index()
        {
            var cart = _cartService.GetShoppingCart();
            return View(cart);
        }

        [HttpPost]
        public IActionResult ButtonClick()
        {
            push();
            return Ok("12");
        }

        [HttpPost]
        public IActionResult UpdateCountDB([FromBody]ItemsCountModel
model)
        {
            Console.WriteLine(model.NewCount + " " + model.ItemId);
            model.FullPrice = _cartService.UpdateDB(model.ItemId,
model.NewCount);
            return Ok(JsonSerializer.Serialize(model));
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
```

```

        public IActionResult Error()
        {
            //return View(new ErrorViewModel { RequestId =
            Activity.Current?.Id ?? HttpContext.TraceIdentifier });
            return Ok();
        }

        public void push()
        {
            var factory = new ConnectionFactory() { HostName = "localhost"
};
            using (var connection = factory.CreateConnection())
            {
                using (var chanel = connection.CreateModel())
                {

                    ShoppingCart model =
                    _cartService.GetShoppingCart();

                    var modelDto =
                    _mapper.Map<ShoppingCartDTO>(model);

                    var message = JsonSerializer.Serialize(modelDto,
                    new JsonSerializerOptions()
                    {
                        ReferenceHandler =
                        ReferenceHandler.IgnoreCycles
                    });
                    var body = Encoding.UTF8.GetBytes(message);

                    chanel.BasicPublish("OrderExchange",
                    "amazon.orderService.order", mandatory: false, null, body);
                }
            }
        }
    }
}

```

## 2. WarehouseService клас StartupBackgroundService.cs

```

namespace WarehouseService.Services
{
    public class StartupBackgroundService : BackgroundService
    {

        private readonly IMapper _mapper;

        public StartupBackgroundService(IMapper mapper)
        {

```

```

        _mapper = mapper;
    }

protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    var factory = new ConnectionFactory() { HostName = "localhost" };
    var connection = factory.CreateConnection();

    var chanel = connection.CreateModel();

    var consumer = new EventingBasicConsumer(chanel);
    consumer.Received += (sender, args) =>
    {
        var body = args.Body;
        var message = Encoding.UTF8.GetString(body.ToArray());

        ShopingCart cart =
        _mapper.Map<ShopingCart>(JsonConvert.DeserializeObject<ShopingCartDTO>(message));

        //Console.WriteLine(message);

        for (int i = 0; i < cart.CartItems.Count; i++)
        {
            if (cart.CartItems[i].Count != 0)
            {
                Console.WriteLine("Id товара " +
                cart.CartItems[i].Item.Id + " Имя " + cart.CartItems[i].Item.Name + " Количество " +
                cart.CartItems[i].Count);
            }
        }
        Console.WriteLine("");

        push("Статус: Сборка");
        Thread.Sleep(1000);
        push("Статус: Отправлено");
    };
    chanel.BasicConsume(queue: "amazin.orderSrevice.order", autoAck: true,
    consumer: consumer);
}

public void push(string status)
{
    var factory = new ConnectionFactory() { HostName = "localhost" };
    using (var connection = factory.CreateConnection())
    {
        using (var chanel = connection.CreateModel())
        {
            var message = status;

```



```

        var body = Encoding.UTF8.GetBytes(message);

        chanel.BasicPublish("OrderExchange",
            "amazin.notificationService.order", mandatory: false, null, body);
    }
}
}
}
}

```

### 3. NotificationService клас StartupBackgroundService.cs

```

namespace AmazinStore.Notification.Services
{
    public class StartupBackgroundService : BackgroundService
    {
        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            var factory = new ConnectionFactory() { HostName = "localhost" };
            var connection = factory.CreateConnection();

            var chanel = connection.CreateModel();

            var consumer = new EventingBasicConsumer(chanel);
            consumer.Received += (sender, args) =>
            {
                var body = args.Body;
                var message = Encoding.UTF8.GetString(body.ToArray());

                Console.WriteLine(message);

            };
            chanel.BasicConsume(queue: "amazin.notificationService.order",
                autoAck: true, consumer: consumer);
        }
    }
}

```