МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет інформаційних технологій
Кафедра мережевих та інтернет технологій

ЗАТВЕРДЖУЮ
завідувач кафедри
мережевих та інтернет технологій
_____ Ю.В. Кравченко
«_____»_____2021 року

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

галузі знань 17 «Електроніка та телекомунікації»
за спеціальністю 172 «Телекомунікації та радіотехніка»

на тему:

Архітектура, керована подіями, для розробки інформаційної системи, що забезпечує підтримання концепції "zero waste"

Виконав: студентка групи МІТ -41
Фекете Діана Михайлівна _____
(прізвище ім'я по-батькові) (підпис)

Керівник: доцент кафедри мережевих та інтернет технологій
к.т.н. Герасименко О.Ю. _____
(посада, прізвище ім'я по-батькові) (підпис)

Київ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет інформаційних технологій
Кафедра мережевих та інтернет технологій

ЗАТВЕРДЖУЮ
завідувач кафедри
мережевих та інтернет технологій

_____ Ю.В. Кравченко

«_____»_____2021 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ

Здобувачу вищої освіти Фекете Діані Михайлівні
(прізвище, ім'я, по батькові)

1. Тема роботи:
«Архітектура, керована подіями, для розробки інформаційної системи, яка забезпечує підтримання концепції "zero waste"»

затверджена на засіданні кафедри МІТ «4» грудня 2020 р. протокол № 8

2. Термін здачі закінченої роботи «31» травня 2021 р.

3. Вихідні дані до проекту (роботи)

Архітектура інформаційної системи, яка забезпечує підтримання концепції "zero waste", – мікросервісна архітектура, керована подіями

4. Зміст пояснювальної записки (перелік питань, що їх потрібно розробити, обсяг – 35-40 стор.)

1. Аналіз задачі розробки інформаційної системи, що забезпечує підтримання концепції "zero waste"
2. Проектування архітектури інформаційної системи
3. Розробка інформаційної системи

5. Перелік графічного матеріалу 8-10 слайдів

Постановка задачі, порівняльний аналіз та вибір засобів реалізації, шаблон проектування «публікація-підписка», принцип CQRS, схеми взаємодії компонентів системи при виконанні базових операцій, вибір засобів реалізації, приклад роботи інформаційної системи

Дата видачі завдання

Керівник роботи Доцент кафедри мережевих та інтернет технологій к.т.н. О.Ю. Герасименко
(підпис) (посада, прізвище, ім'я, по батькові)

Завдання прийняв до виконання
(підпис) (прізвище, ім'я, по батькові)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ

Номер	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Підготовчий	29.01.2020	
2	Розділ 1	01.03.2020	
3	Розділ 2	01.04.2020	
4	Розділ 3	01.05.2020	
5	Доповідь та слайди	27.05.2020	
6	Пояснювальна записка	31.05.2020	

Здобувач вищої освіти _____ Фекете Діана Михайлівна
(підпис) (прізвище, ім'я, по батькові)

Керівник _____Герасименко Оксана Юріївна
(підпис) (прізвище, ім'я, по батькові)

РЕФЕРАТ

Пояснювальна записка: 83 с., 48 рис., 9 додатків, 23 джерела.

Об’єкт дослідження: архітектура, керована подіями, для високонавантажених інформаційних систем.

Предмет дослідження: застосування шаблону «публікація-підписка» у проектуванні архітектури високонавантаженої інформаційної системи.

Мета роботи (проекту): розробити мікросервісну архітектуру, керовану подіями, для високонавантаженої інформаційної системи, яка підтримує концепцію "zero waste".

Методи дослідження: системний підхід, порівняльний аналіз.

В роботі проведено аналіз систем на основі подій, найпоширеніших шаблонів, що використовуються при їх проектуванні, та проведено порівняльну характеристику даного підходу з CRUD.

Запропоновано використати шаблон CQRS для проектування внутрішньої взаємодії та шаблон «публікація-підписка» як технологію обміну повідомленнями.

Спроектовано мікросервісну архітектуру, керовану подіями.

Розроблено код програми на мові програмування Go з використанням сховища даних MongoDB, систему обміну повідомленнями NATS, а також представлено приклад розгортання розробленої інформаційної системи з використанням технології контейнеризації додатків Docker.

Результати здійснених у дипломному проекті досліджень можуть бути використані для проектування реальної ефективної інформаційної системи, що забезпечує підтримання концепції «zero waste».

Галузь використання – соціальна сфера, ресторанний бізнес, торгівля.

Напрямки подальших досліджень полягають у вдосконаленні доступного функціоналу, розробки сервісів для блокування, реалізацію DLQ для ефективного репроцесингу помилок, що виникають.

Ключові слова: АРХІТЕКТУРА, ІНФОРМАЦІЙНА СИСТЕМА, ПОДІЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, API, СХЕМА, КОМПОНЕНТ, ПУБЛІКАЧ, ВИДАВЕЦЬ, АГРЕГАТ, СХОВИЩЕ ДАНИХ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface

CRUD – Create, Read, Update, Delete

CQS – Command Query Segmentation

DDD – Domain Driven Design

DLQ – Dead Letter Queue

JSON – JavaScript Object Notation

URL – Uniform Resource Locator

ІС – інформаційна система

ПЗ – програмне забезпечення

ВСТУП

У середньому близько 1.3 мільярдів тон їжі втрачається чи викидається кожен рік у світі – це одна третя від загальної кількості їжі, що призначена для людського споживання відповідно до даних від Food and Agriculture Organization of the United Nations (FAO) [1]. Загальна вартість втраченої їжі сягає до 2.6 трильйонів доларів у рік [2] та її обсяг є достатнім для того, щоб нагодувати всіх 815 мільйонів голодуючих людей у світі. Придатна до споживання їжа втрачається майже в кожній точці харчового ланцюга – починаючи від ферм та виробництв, та закінчуючи магазинами, ресторанами і людськими кухнями [3].

У сучасних реаліях цифрові технології мають невичерпний потенціал, який можна використати для вдосконалення чи повної трансформації вартості та структури харчових ланцюгів (FVCs), а також значним чином посприяти розробці більш продуктивних та стійких систем, що забезпечують обробку чи зберігання їжі [4]. Є сотні прикладів успішної діджиталізації невеликих фермерських угідь, які змогли зменшити харчові відходи чи не до 2% від загального обсягу, покращивши умови зберігання продуктів та забезпечивши пряму доставку на ринок тощо.

Використання технологій на фермах чи виробництвах – це лише невелика частина можливостей, які сучасний світ може запропонувати для повної трансформації культури відходів. Одну з ключових ролей у збільшенні обсягів втрати продуктів відіграють стандарти ринку, за якими смачні та свіжі продукти відправляються в смітник через зовнішні стандарти. Тобто картопля, яка «недостатньо гарна» – не відправиться на полиці супермаркету, а буде просто викинута. На щастя, уже існує сотня стартапів, які націлені на розробку програмного забезпечення, що допомагають зберегти ці продукти від викидання.

Мета цієї роботи полягає в знаходженні та демонстрації можливостей застосування архітектур, що керуються подіями, для розробки сучасної високонавантаженої інформаційної системи. Також у даній роботі розглянуто

переваги даного підходу над CRUD-орієнтованими системами, проаналізовано найпоширеніші проблеми та складнощі, які виникають при використанні цього патерну.

Соціальна цінність даної роботи полягає в створенні архітектури цифрового продукту, який може забезпечити зменшення втрати їжі, ще придатної до споживання. Для розробки високоефективного та легко масштабованого додатку використано сучасні технології та шаблони проектування програмного забезпечення.

РОЗДІЛ І. АНАЛІЗ ЗАДАЧІ РОЗРОБКИ ІНФОРМАЦІЙНОЇ СИСТЕМИ, ЩО ЗАБЕЗПЕЧУЄ ПІДТРИМАННЯ КОНЦЕПЦІЇ "ZERO WASTE"

1.1 Опис та аналіз предметної області

Концепція, що передбачає повну мінімізацію відходів для збереження навколишнього середовища та ресурсів планети, називається «zero waste». Це моральна, економічна і далекоглядна ціль, спрямована на зміну способу життя людей, де майже всі матеріали переробляються, а кількість речей, які викидаються, прямує до нуля.

Одна з найбільших перешкод на шляху до досягнення цілі «zero waste» (або ж нуль відходів) є колосальна втрата їжі на планеті («food waste» і «food lost»).

Історія терміну «food waste» тісно пов’язана з поняттям глобалізації. У теперішніх реаліях продукт з будь-якої точки світу можна легко придбати онлайн чи знайти в найближчому супермаркеті. Таким чином, мексиканські авокадо можна легко знайти на полицях українських магазинів, а наші вишні – на полицях холодної Фінляндії. Протягом цієї подорожі від ферми до споживача їжа втрачається майже чи не на кожному етапі – свіжі фрукти, овочі чи молоко можуть легко зіпсуватись та є дуже вразливими до умов перевезення.

Ресторани та харчові сервіси відіграють одну з ключових ролей в обсязі втрати їжі та відходів (food lost and waste – FWL) – від розробки меню до управління залишками з тарілок відвідувачів [5].

Враховуючи той факт, що близько 800 мільйонів людей у світі, більшість з них у країнах, що розвиваються, страждають від хронічної нестачі їжі [6], дуже прикро усвідомлювати, що одна третина їжі викидається.

Сучасні технології призначені для того, щоб вирішувати проблеми, з якими стикається людство. Відповідно існує безліч інструментів та способів для підвищення рівня людського життя чи вирішення світових проблем.

Задача даної роботи полягає в тому, щоб спроектувати архітектуру для застосунка, що допоможе боротись з світовим голодом та підтримувати загальну концепцію «zero waste». Ідея застосунку полягає в тому, щоб надати можливість для збуту харчових продуктів, які не продались через недосконалий зовнішній вигляд тощо.

1.2 Призначення архітектури, що забезпечує підтримання концепції «zero waste»

Як вже було зазначено, основна мета застосунку – сприяти зменшенню викидання продуктів, які придатні до споживання. Сучасні технології надають невичерпні можливості щодо усунення проблем з якими стикається людство.

Призначення даної архітектури полягає не лише у вирішеннях проблем пов’язаних з «food waste» чи «food loss», а також у вирішенні проблем з якими стикаються розробники при проектуванні сучасного ПЗ.

Визначальні критерії при проектуванні систем:

– кількість користувачів, що будуть використовувати спроектоване програмне забезпечення;
– кількість навантаження на одиницю часу;
– бюджет, доступний для програмування та підтримки системи;
– можливості для монетизації застосунку;

Проектування архітектури системи – це визначення основних вимог та критеріїв, яким має відповідати система, що розглядається. Побудова архітектури полягає у визначенні вхідних та вихідних даних, розділення системи на підсистеми, визначенні основних компонентів та технологій, що будуть застосовані.

Невідповідність архітектури – це ситуація, при якій розроблена система порушує принципи підходів, що були обрані для проектування або ж коли дана архітектура не підходить для конкретної ситуації.

1.3Аналіз поточної ситуації на ринку застосунків

Поточна ситуація на ринку еко-застосунків України залишає бажати кращого. У цілому світі тема програмного забезпечення, що допомагає зберегти природу, надзвичайно поширена. Останнє десятиліття інвестиції в застосунки подібного роду збільшитися в десятки, а то й сотні разів. Усвідомлення катастрофічної екологічної ситуації, що склалася на Землі, є рушійною силою, яка спонукає корпорації, великих та малих інвесторів вкладати кошти в розвиток технологій, які потенційно могли б допомогти уникнути глобальних катаклізмів.

У даній роботі для прикладу варто розглянути два найбільших відомих проекти, які зараз знаходяться на етапі стартапів, що дуже близькі до теми, яка розглядається: OLIO [7] та Full Harvest [8].

Olio – мобільний додаток, що забезпечує можливість обміну їжею з метою зменшення харчових відходів. Користувачі, які мають залишки їжі, можуть легко та безкоштовно поділитись нею з тими, хто цього потребує. Їжа має бути придатною для споживання, вона може бути як сирою, так і приготованою, закритою чи відкритою [9]. Перший реліз проекту був в 2015 році засновницями Тесою Кларк та Сашею Селестіал-Ван [10]. До початку жовтня 2017 року компанія зібрала близько 2.2 млн. доларів інвестицій [11]. До вересня 2020 року ним вже користувались близько 2.3 млн. користувачів.

Історія Olio ще раз підтверджує думку, що тема надзвичайно актуальна і зібрати необхідні інвестиції зараз дуже просто, адже ринок поки не перенасичений.

Оскільки, вибір продуктів на ринку занадто великий і з часом збільшується, відповідно і конкуренція між виробництвами зростає. Науковці переконані, що уникнення стандартів краси при виборі продуктів і зважання на те, що смак всередині жодним чином не залежить від зовнішніх факторів, може побороти світовий голод [12]. На щастя, вже з’явились застосунки, які

допомагають недосконалим продуктам потрапити на стіл споживачу чи то на годування тваринам, замість згнивання на звалищі.

Однією з таких компаній є Full Harvest – молодий американський стартап, що бореться з проблемою «food waste». Застосунок, що допомагає продавати чи купувати продукти харчування, які не потрапили на ринок в силу виключно зовнішніх факторів. Надзвичайно популярний стартап, який вже привернув увагу таких гігантів як: Forbes, Shape, Wall Street Journal. Ідеологія дуже проста: у користувачів є можливість як придбати товари, так і позбутися їх у сфері B2B [13].

1.4 Аналіз вимог до архітектури, керованої подіями, для розробки інформаційної системи. Постановка задачі

Виконання даної роботи передбачає вивчення основних переваг систем, що керуються подіями, а також ознайомлення з найпоширенішими технологіями, що використовуються для проектування таких систем. На основі проведених досліджень передбачається вибір найефективніших, дешевих та надійних інструментів для розробки та втілення проекту в життя з можливістю подальшого вдосконалення.

Даний проект передбачає такі технічні та експлуатаційні результати роботи, як: примітивна інформаційна система, що складається мінімум з двох мікросервісів, які обмінюються повідомленнями, повинен бути реалізований клієнтський інтерфейс, а також підключена база даних, для збереження інформації про користувачів, їх позиції на утриманні чи нові пропозиції.

Інформаційна система повинна забезпечувати можливість роботи у багатоклієнтському режимі роботи. Повинна забезпечуватися цілісність даних, висока доступність та продуктивність.

Обов’язковою умовою є визначення рекомендованих параметрів під можливими навантаженнями.

У результаті виконання даної роботи мають бути отримані:

– інфологічна модель предметної області;
– короткий опис можливих повідомлень, які передбачає система;
– програмний додаток, що складається з клієнтської та серверної частин;
– програмний код мікросервісів та клієнтської частини;
– логічна та фізична моделі бази даних.

Клієнтська частина повинна гарантувати та забезпечувати комфортну взаємодію користувача з додатком. Орієнтуючись та опираючись на права доступу (адміністратор чи клієнт) – система повинна забезпечувати можливість внесення даних, зміни, видалення чи редагування тощо.

З боку користувача необхідно забезпечити можливість створення покупки продуктів харчування, оплати, пошуку елементів та їх фільтр. Також необхідно розробити функціонал для введення пропозицій з продажу, можливість додати опис, ціну, місце доставки.

Система повинна передбачати, що один продукт може бути куплений тільки один раз, забезпечувати асинхронну поведінку, щоб уникнути ситуацій, коли користувач бронює продукт, якого вже немає в наявності. Цей елемент є ключовим в розробці даного продукту.

1.5 Аналіз підходів до проектування архітектури програмного забезпечення

1.5.1 Поняття архітектури програмного забезпечення. Основні види архітектур

Архітектура програмного забезпечення системи призначена для відображення того, як вона має бути організована інформаційна система (ІС), з яких компонентів буде складатися та як вони будуть взаємодіяти між собою, реалізуючи функціонал ІС. Дизайн архітектури описує високорівневу структуру

системи та її компонентів. Мета архітектури полягає в зображенні відношень між компонентами та їх базову взаємодію [14].

Вирішення питання вибору та проектування архітектури визначається на початковому етапі процесу розробки, що допомагає програмістам отримати детальний огляд системи. Правильно підібраний дизайн є рушійною силою в полегшенні комунікації між зацікавленими сторонами, суттєвому зниженні вартості обслуговування, спрощенню обслуговування системи та робить її більш гнучкою та легшою у масштабуванні.

У даному контексті варто розглянути різні види архітектури, які вважаються базовими, їх переваги та недоліки.

1. Монолітна архітектура.

Монолітний застосунок працює як єдина програма та запускається як один виконуваний файл. З ростом кодової бази, складність розробки та управління проектом виростає в геометричній прогресії. Проте, такий підхід дає деякі переваги у роботі. Можна спілкуватись з різними частинами проекту за допомогою прямих викликів методів, спільного використання пам’яті чи передачі повідомлень. Це дозволяє уникнути затримок в передачі повідомлень як, наприклад у HTTP та RPC.

Основні недоліки монолітної архітектури [15]:

– великі моноліти складно підтримувати та розвивати;
– будь-яка зміна в проекті вимагає повного перезапуску, що може стати причиною значного простою. Цей недолік також може стати причиною значного сповільнення розробки та тестування;
– моноліти мають обмеження в масштабуванні. Єдиний вихід, щоб легко обробляти нові запити, – це створити додатковий екземпляр моноліту, щоб розділили навантаження. Але трафік може тільки погіршити ситуацію з обробкою запитів у деяких частинах проекту, тому в цьому випадку створення нового екземпляру не буде ефективним;

– така архітектура повністю обмежує вибір технологій. Модулі мають бути написані однією мовою програмування та використовувати одні і ті ж фреймворки.

Оскільки, завдання дизайну системи, що розглядається у даній роботі, вимагає можливості швидкої розробки з мінімальними затратами, а також можливістю бути високоефективною навіть при величезних навантаженнях, з використанням різних технологій та розроблятись паралельно декількома командами у майбутньому – монолітна архітектура для розробки не підходить.

2. Мікросервісна архітектура.

Мікросервісна архітектура є повною протилежністю монолітної. Моноліт зберігає весь функціонал в одному виконуваному об’єкті, а мікросервіси розподіляють функціонал на маленькі частини. Ця архітектура допомагає вирішити деякі проблеми, які були розглянуті в попередньому пункті:

– задача кожного мікросервісу полягає в утриманні маленької частини функціоналу, що робить кодову базу простішою і набагато легшою в підтриманні та тестуванні;
– мікросервіси легко переводити на нову версію. У разі оновлення функціоналу, стара версія може працювати паралельно з новою, до тих пір, поки всі інші сервіси не почнуть використовувати нову версію;
– масштабування мікросервісної архітектури не вимагає дублікації усіх сервісів. Розробники можуть збільшувати чи зменшувати кількість екземплярів кожного сервісу в залежності від потреби;
– ця архітектура обмежує вибір технологій тільки в межах сервісу. Розробники різних сервісів можуть вибирати технології за власними бажанням та потребами сервісу.

Мікросервіс – це додаток, що може масштабуватись, запускатись та тестуватись повністю незалежно від усієї системи. Це вирішує питання єдиної відповідальності, коли сервіс відповідає лише за одну проблему, яку легко зрозуміти [16].

Застосунки, які використовують мікросервісну архітектуру, базуються на визначені кількості незалежно запущених частин, які легко масштабувати та тестувати.

Інша важлива перевага такого підходу – це надійність. Нехай, один з сервісів недоступний, тоді решта системи продовжить працювати, як і очікується.

Дизайн обмеження меж відповідальності сервісами може бути складним, але надзвичайно важливо розділити їх правильно, в іншому випадку вийде просто розподілений моноліт, який вмістить у собі проблеми як мікросервісів так і моноліту.

3. Архітектура, керована подіями.

При розробці мікросервісів та розподілених систем виникає потреба в комунікації та координації між сервісами та додатками. Це саме те, що допомагає вирішити архітектура, керована подіями.

Архітектура, керована подіями, – це архітектура програмного забезпечення, де застосунок публікує, визначає та відповідає на події. Подія репрезентує зміну в стані системи. Архітектура включає видавців (сутностей, які публікують події), процесори, відповідачів та посилання комунікацій [17].

Цей підхід застосовується для публікації подій при зміні стану системи. Коли стан визначеного компоненту застосунку оновлюється, відбувається публікація нової події, що свідчить про цю зміну. Ця подія може бути опублікована в «тему» (іменований логічний канал), підписники якої будуть оповіщені про цю зміну. Отримувач може викликати деякі функції після отримання події, після чого може опублікувати нове повідомлення для своїх підписників. Схема роботи шаблону «публікація-підписка» зображений на рисунку 1.1.

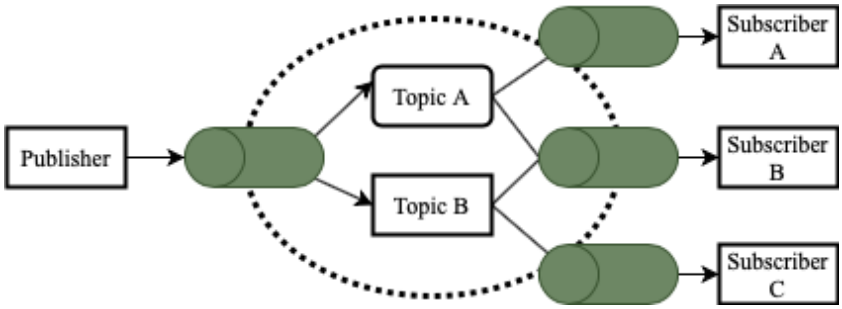


Рисунок 1.1 – Схема роботи шаблону проектування «публікація-підписка»

Така система може бути спроектована для вирішення питань узгодженості в розподілених та мікросервісних застосунках, де використання розподілених транзакцій є дуже складним чи не вистачає підтримки в базах даних. [18] Використовуючи таку архітектуру, додатки можуть використовувати події в якості сигналів для інших систем, про те, що вони виконали визначені дії. У разі, якщо виконання було перервано чи не виконано, додатки можуть створити подію, яка буде свідчити про помилку в системі.

Одна за найбільших переваг керованих подіями систем є те, що система буде слабо зв’язана. Це означає, що сервісам необхідно дуже мало інформації один про одного. Така система надзвичайно легко та швидко масштабується, що і є головною метою архітектури, яка розглядається у цій роботі.

1.5.2 Аналіз підходів до проектування програмного забезпечення

У цьому пункті буде розглянуто підходи до розробки програмного забезпечення, які будуть застосовані в рішенні для продукту, що розглядається в даній роботі. Підходи до проектування програмного забезпечення важливі для розуміння взаємозв'язку між модулями та функціональними можливостями кожного модуля.

1. Предметно-орієнтоване проектування

Ідея даного підходу полягає в проектуванні програмного забезпечення у такому вигляді, щоб вони імітували процес, який відбувається в реальному

житті. Команда розробників працює разом з предметним експертом, для того щоб отримати концептуальні рішення щодо системи, використовуючи єдину мову (ubiquitous language) [19].

Концептуальний опис – це модель, за якою розробники мають можливість спроектувати застосунок, не втрачаючи можливості співпрацювати з усіма, хто бере участь у розробці проекту та дизайні складних рішень.

2. Шаблон розділення відповідальності на команди та запити (CQRS)

CQRS (Command-Query Responsibility Segregation) – принцип імперативного програмування, шаблон розділення відповідальності на команди та запити. Розділяє операції зчитування та оновлення сховища даних. На рисунку 1.2 зображена узагальнена модель даного шаблону.

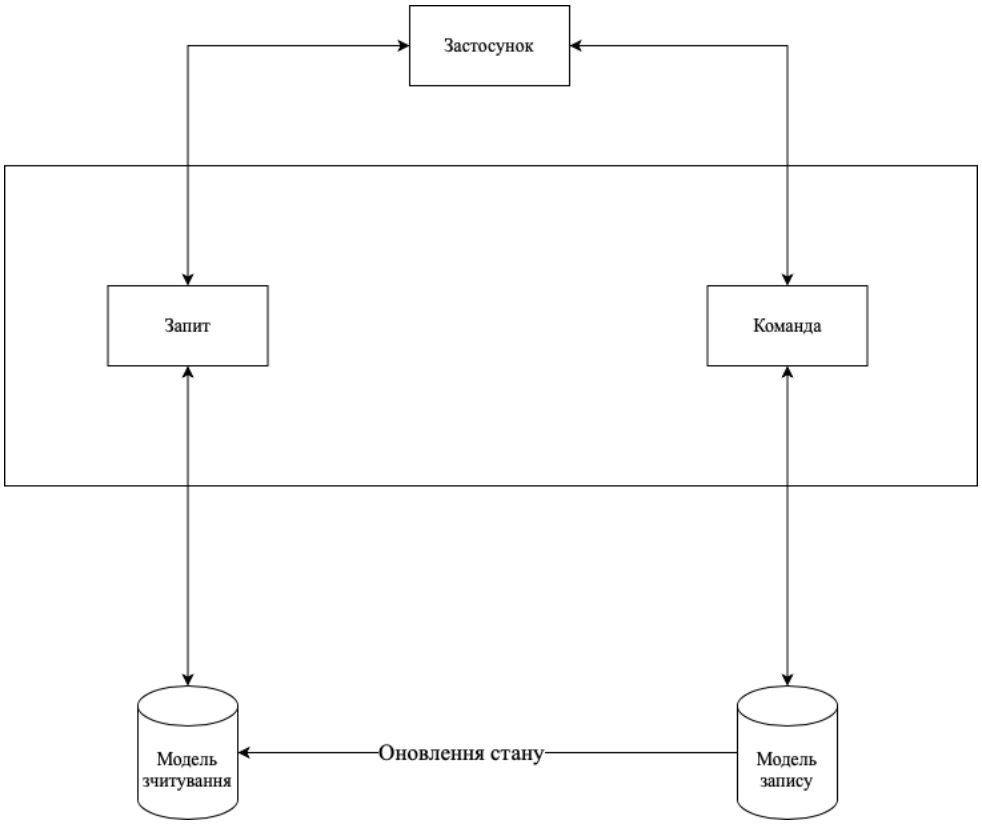


Рисунок 1.2 – Принцип програмування CQRS

Впровадження CQRS допомагає максимізувати ефективність, безпечність та масштабованість додатку, дозволяє побудувати різні моделі зчитування, які можуть бути оптимізовані під вимоги системи. Можна зберегти нормалізовану

модель запису для оптимізованих оновлень, в той же час тримаючи моделі зчитування денормалізованими для забезпечення ефективності [20].

Команди можуть оброблятись асинхронно. При помилках у сховищах, які унеможливлюють запис даних, можна зберегти команду в чергу та перезапустити виконання після виправлення помилок.

CQRS має багато переваг у деяких системах, але цей підхід є далеко не універсальним. Більшість систем працює добре, маючи лише одне сховище для зчитування та запису даних, у таких випадках впровадження CQRS тільки збільшить складність.

Висновки по розділу І

У даному розділі було розглянуто поняття концепції «zero waste» та її основні характеристики. Розглянуто два проекти на сучасному ринку: Olio та FullHarvest, їх основні характеристики та призначення.

Розглянуто призначення архітектур систем, на основі подій та їх застосування, а також призначення системи, що проектується. Визначено призначення даної роботи, основні характеристики та вимоги до системи, що проектується. Визначено основні елементи, які мають бути спроектовані в даній роботі та функції, які має підтримувати система.

Було визначено поняття мікросервісної та монолітної архітектур, проведено порівняльну характеристику та визначено недоліки й переваги для даних підходів.

Проаналізовано основні шаблони та підходи до проектування програмного забезпечення, що будуть використовуватись при проектуванні даної роботи та розглянули їх переваги та недоліки. Було розглянуто такі поняття як: CQRS та шаблон обміну повідомленнями «публікація-підписка».

РОЗДІЛ ІІ. ПРОЕКТУВАННЯ АРХІТЕКТУРИ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Поняття події. Джерела подій

Джерела подій є альтернативою до поширеної та відомої CRUD-архітектури. Найбільша різниця між системами, керованими подіями та системами, які базуються на запитах, у тому, що в першому варіанті сервіси жодним чином не взаємодіють, сервісам не треба чекати відповіді один від одного, вони працюють повністю незалежно.

Більшість застосунків оперують даними, і типовим підходом для таких застосунків є постійне оновлення поточного стану даних. Наприклад, в традиційній CRUD-моделі, обробка даних відбувається шляхом зчитування даних зі сховища, внесення деяких змін та оновлення поточного стану даних вже з новими значеннями, найчастіше використовуючи транзакції, щоб блокувати дані.

Підхід CRUD має такі обмеження:

– системи, що базуються на цьому підході, здійснюють операції оновлення напряму в сховищі, що може значно зменшити продуктивність, обмежити масштабування, зважаючи на процес виконання даних операцій;
– у спільній системі з багатьма одночасними користувачами виникає проблема конфліктів оновлень частіше, оскільки операції оновлення відбуваються з одним елементом даних;
– у разі, якщо немає додаткового механізму логування, історія буде втрачена.

Вирішити вище перелічені проблеми можна за допомогою, як вже було зазначено в першому розділі, системи, що керуються подіями – це потужний архітектурний шаблон, суть якого полягає в збереженні усіх станів продукту в

оригінальній послідовності. Ця послідовність слугує і як система подій, за допомогою якої можна отримати поточний стан системи, і як перелік записів про все, що відбулося протягом її життєвого циклу. Нижче представлено детальний опис того, що таке подія, за описом Беттс Д. [21].

1. Подія – це процес, який відбувся в минулому.
2. В патерні джерела подій, подія є незмінною (immutable), тобто вона не може бути змінена чи відмінена. Щоб обнулити подію, необхідно створити нову, яка нівелює попередню.
3. Подія має одного видавця та декількох слухачів, які можуть отримувати та реагувати на неї.
4. При використанні джерел подій, вона автоматично описує деякий бізнес-процес. Ім’я події зазвичай пишуть в минулому часі.

Використовуючи CQRS в системах, керованими подіями – подія послідовно перетворюється в команду. Термін команди в контексті CQRS шаблону описаний у першому розділі та визначений як операція, яка тригерить зміну в системі.

Використання CQRS шаблону не є обов’язковим в системах, що керуються подіями, але ці два патерни чудово поєднуються та доповнюють один одного. В даній роботі їх буде поєднано. Поєднання цих двох патернів зображено на рисунку 2.1.

У більшості систем, що керуються подіями без використання CQRS шаблону, можна стикнутися з проблемою при виконанні запиту до визначеного ресурсу, коли сховище подій зберігає зміни внесені в систему, які не являються її поточним станом [20]. Наприклад, щоб отримати інформацію про актуальний стан позиції «Яблука» від визначеного ресторану. Для того, щоб визначити, скільки кілограмів яблук ще доступно, необхідно отримати всі події по цій позиції та обробити їх (перезапустити), щоб перевірити – чи 5 доступних кілограмів яблук це актуальний стан.

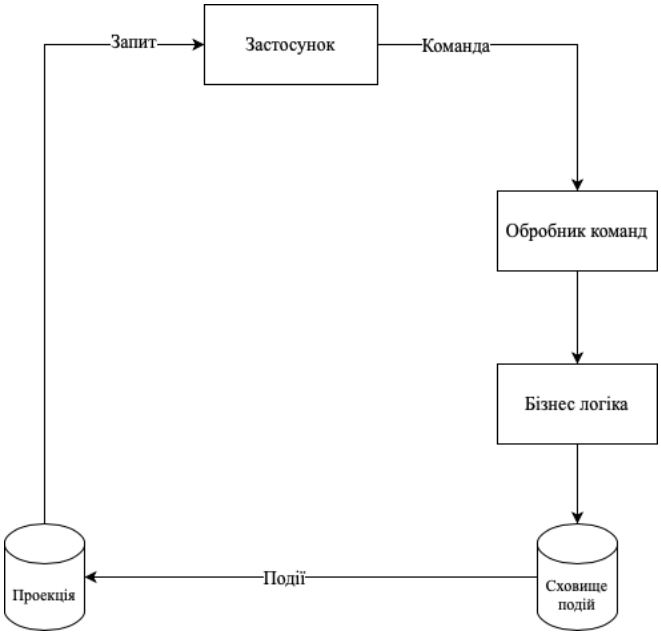


Рисунок 2.1 – Використання CQRS з джерелами подій

Разом з CQRS можна легко побудувати моделі для вичитування (агрегати), які зберігатимуть поточний стан при отриманні нових подій та використовувати це для схожих запитів. На рисунку 2.2 зображене застосування подій до агрегатів.

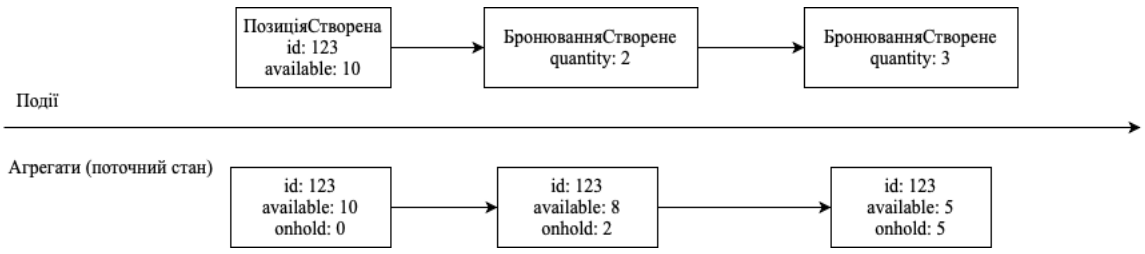


Рисунок 2.2 – Послідовне оновлення агрегатів під час виникнення нових подій

При отриманні системою, що керується подіями, команди, вона перевіряється на наявність необхідних значень та відповідність бізнес-вимогам. Якщо команда проходить валідацію, вона результує в створення однієї чи

декількох подій, які, в свою чергу, будуть застосовані до агрегату. В шаблоні джерел подій кожен агрегат будується на основі всіх подій, що йому належать.

Події, які належать агрегату та були збережені в сховище, ніколи не повинні оновлюватись чи видалятись. Замість видалення чи оновлення події, щоб виправити помилки, необхідно створити нову подію, яка усуне проблему [22]. Хоча є ситуації, коли оновлення подій може бути корисним, наприклад при переведенні подій на нову схему [23].

При валідації команди може бути необхідно отримати поточний стан агрегату. Наприклад, скоріш за все можливості створити бронювання продукту в пропозиції, яка була відмінена, не має бути. Для того, щоб перевірити чи пропозиція була відмінена, під час створення нового бронювання необхідно регідрувати агрегат. Процес регідрадації агрегату полягає в отриманні всіх подій, що належать агрегату, зі сховища, сортування їх від старішої до новішої. Після цього ці події застосовуються до порожнього агрегату (початковий стан).

Наведемо переваги запропонованого підходу. Найбільшою перевагою систем, що керуються подіями, є те, що поточний стан застосунку завжди буде побудований з послідовності подій. Це означає, що систему можна повернути до будь-якого моменту в часі, що допоможе тестувати та відлагоджувати системи швидко, адже завжди можна повернутись назад та відтворити те, що вже трапилось в минулому.

Одною з особливостей систем, що використовують цей шаблон, є можливість якісного смоук-тестування (smoke testing). Цей тип тестування може бути застосований для симуляції використання реальної системи, щоб переконатися, що вона буде працювати повністю правильно перед тип як використовувати цей код у виробництві. Якщо існує сховище подій з усіма змінами, що ведуть до поточного стану, є можливість перезапустити ці події під час тестування і отримати очікуваний стан, щоб бути впевненим у правильності кінцевого результату на виробництві.

Наступною великою перевагою є можливість детального огляду та відстеження даних у системі, що підтримує бізнес-аналітику. Системи, що керуються подіями, надають можливість для аналізу наявних даних, щоб виявити незвичні та цікаві поведінки у застосунку. Більше того, така перевага дозволяє удосконалити певні зони бізнесу чи виправити помилки, які до цього не були очевидними.

Проте запропонований підхід не позбавлений недоліків. Три головні проблеми, з якими можна стикнутися при розробці систем на основі подій, це:

1. події можуть повторюватись;
2. асинхронність отримання подій;
3. непослідовність подій.

Системи, що керуються подіями, за своєю будовою неконсистентні, що означає, що повідомлення можуть приходити непослідовно. Якщо два користувачі роблять замовлення одного й того ж продукту в системі, який залишився лише один, виникає необхідність відстежувати, чи дані оновлюються вчасно та повідомити одного з користувачів про те, що замовлення не може бути оброблено у зв’язку з нестачею продуктів.

Системи, що керуються подіями набагато складніші в своїй реалізації та більш нові, аніж CRUD.

Одним з найбільших недоліків є відсутність явної схеми подій. У SQL базах даних існує чітка схема обробки даних із визначеними значеннями та типами. Сховища подій в свою чергу зберігають усі сукупності подій. Проблема виникає, коли події відрізняються за вмістом та типами даних. Цей недолік ускладнює розвинення схеми та обробку даних.

2.2 Загальний опис архітектури та її компонентів

2.2.1 Високорівнена архітетура системи

Дана система спроектована як розподілена система з мікросервісною та керованою подіями архітектурою. Внутрішні сервіси спілкуються за допомогою NATS, а REST запити використовуються для комунікації між клієнтською та серверною частинами. Для збереження даних використовуються джерела подій. Застосунок розроблений таким чином, щоб в майбутньому його було легко масштабувати та розвивати. На початковому етапі було розроблено 3 мікросервіси.

На рисунку 2.3 зображена високорівнева архітектура мікросервісної частини.

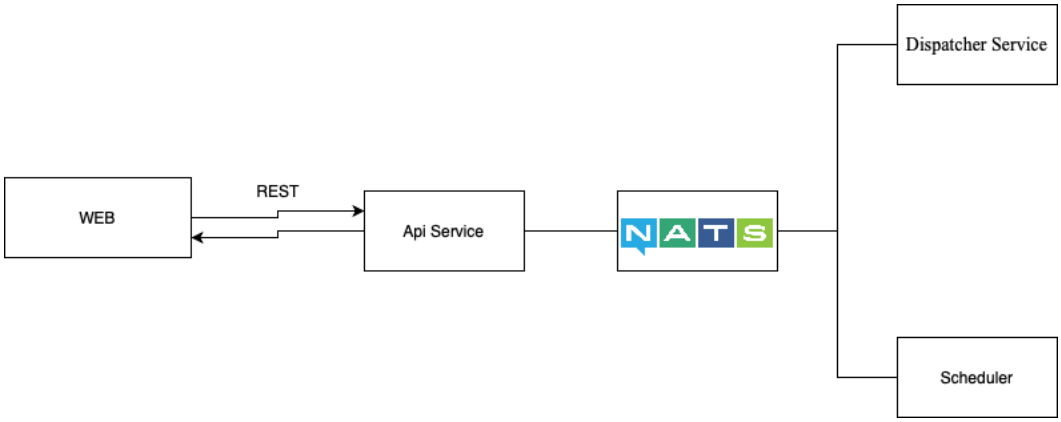


Рисунок 2.3– Високорівнева архітектура системи

Наслідуючи CQRS-шаблон, буде відділено моделі зчитування та запису. Таким чином, лише API-сервіс буде отримувати REST-запити з клієнтської частини та створювати команди для Dispatcher-сервісу, який, в свою чергу, буде записувати події та агрегати в сховище подій.

Спілкування між API, Dispatcher та Scheduler сервісами відбувається на основі обмінів повідомленнями, використовуючи відомий шаблон «публікація-підписка». Сервіси працюють незалежно один від одного, зникає необхідність в

очікуванні відповіді тощо. Основна ідея в тому, що API-сервіс публікує подію (повідомлення), яка свідчить про те, що відбулась зміна в процесі і будь-який сервіс, якому цікава ця подія, може її прочитати. Детальніше про цей шаблон обміну повідомленнями можна почитати в першому розділі.

API-сервіс відповідає за отримання та обробку запитів від клієнтської частини застосунку, використовуючи REST-методи. Після обробки запитів сервіс створює повідомлення, публікація яких відбувається за допомогою сучасної системи обміну повідомленнями з відкритим кодом NATS.

Dispatcher-сервіс виконує ролі сховища подій та часткового проектора. Для подальшого розвитку можна буде розділити цей сервіс на два компоненти: читач та видавець (reader and writer), як у класичному CQRS-шаблоні. Цей сервіс отримує команди та у разі успішної перевірки генерує подію та записує її у сховище. Також цей сервіс використовується для генерації агрегатів та збереження їх у сховище.

Scheduler – це сервіс, який виконує роль планувальника завдання за часом (cronjob). Він підписується на події та, у разі необхідності, звертається у сховище агрегатів, щоб оновити їх стан.

2.2.2 Основні компоненти та їх властивості

Система складається з 4 основних компонентів та 1 агрегату.

1. Продукт.

Продукт – це елемент, що доступний для збуту, який використовується при створенні нової позиції.

На рисунку 2.4 зображена структура об’єкту «Продукт» та належні йому поля:

– Id – унікальний номер продукту;
– UserId – унікальний номер користувача, який вніс продукт в сховищ даних;

– Unit – це об’єкт, що зберігає одиницю виміру продукту (наприклад, продукт «Яблука» вимірюється в кілограмах);
– Name – ім’я продукту.

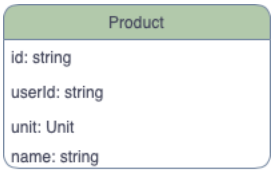


Рисунок 2.4 – Структура компоненту «Продукт»

2. Позиція.

Позиція – це доступний елемент для резервування та подальшого підтвердження замовлення. Може бути створена ресторанами, магазинами та іншими користувачами, які шукають можливості для збуту продуктів.

На рисунку 2.5 зображена структура «Позиції» та поля, які вона містить:

– Id – унікальний номер позиції;
– UserId – унікальний номер користувача, який створив позицію;
– ProductId – унікальний номер продукту, що публікується;
– Type – тип позиції (перелік);
– Quantity – кількість продукту;
– CreatedAt – час створення позиції (використовується для того, щоб відстежувати закінчення терміну придатності та закриття позиції за необхідності).
– ExpiresAt – час завершення терміну дії пропозиції.

Типи позиції використовуються як ідентифікатори для створення подій. Оскільки, мета полягає у детальному відстеженні історії по всіх позиціях, відстеження утримань тощо.

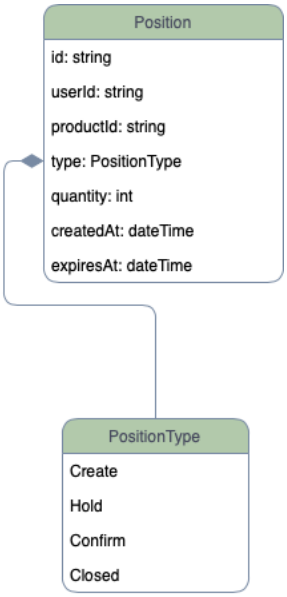


Рисунок 2.5 – Структура компоненту «Позиція»

3. Агрегат Позиції.

Внутрішній тип, який використовується для відстеження поточного стану компоненту «Позиції» і оновлюється з кожною новою подією над позицією, що належить цьому агрегатові.

На рисунку 2.6 зображена структура «Агрегат Позиції» та поля, які вона містить:

– Id – унікальний номер позиції;
– UserId – унікальний номер користувача, який створив позицію;
– ProductId – унікальний номер продукту, доступний в позиції;
– Available – кількість доступного продукту;
– OnHold – кількість продукту, що знаходиться на утриманні.

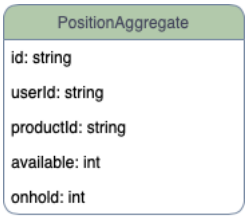


Рисунок 2.6 – Структура компоненту «Агрегат Позиції»

4. Користувач

На рисунку 2.7 зображена структура компоненту Користувач та поля, які вона містить:

– Id – унікальний номер користувача;
– FirstName – ім’я користувача;
– LastName – прізвище користувача.

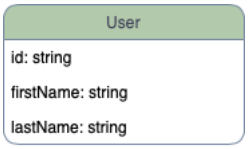


Рисунок 2.7 – Структура компоненту «Користувач»

5. Утримання

Тип, який використовується для відстеження позицій, які знаходяться на утриманні, та користувачів, які їх створили. Наприклад, Користувач1 створює утримання на Позиція1 на 10 одиниць продукту – це свідчить про те, що користувач має намір забрати якусь частину продуктів з позиції. Позиція може містити невизначену кількість утримань від невизначеної кількості користувачів (допоки не закінчиться доступна кількість продуктів). На рисунку 2.8 зображена структура даного компоненту з типами полів:

– Id – унікальний номер утримання;
– UserId – номер користувача, що створив утримання;
– PositionId – номер позиції, що утримується;
– Quantity – кількість елементів позиції, що утримується;
– CreatedAt – дата створення утримання.

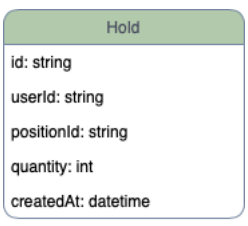


Рисунок 2.8 – Структура компоненту «Утримання»

2.3 Основні принципи функціоналу системи

Користувач взаємодіє з додатком за допомогою клієнтської частини, яка надсилає HTTP-запити до серверної частини. Кожна взаємодія користувача і додатку – це окремий виклик доступних REST-методів, які отримує API-сервіс, призначений для обробки запитів та створення команд для всієї системи. Такий підхід є високоефективним та легким у підтримці.

Базовий функціонал доступний для взаємодії користувача та додатку, що розглядається в даній роботі:

– Додати продукт;
– Створити позицію;
– Отримати всі актуальні позиції для окремого користувача;
– Отримати позиції доступні до утримання;
– Утримати визначену кількість продукту з позиції;
– Отримати всі утримання для окремого користувача;
– Відмінити створення позиції;
– Підтвердити утримання;
– Скасувати утримання;

Базовий функціонал, що відбувається без взаємодії з користувачем:

– Постійна перевірка поточного стану позицій і скасування позицій в разі закінчення терміну придатності;
– Оновлення поточного стану системи;
– Запис всіх змін станів позицій в сховище подій;

Розглянемо детальніше внутрішню архітектуру найважливіших та найскладніших методів.

Створення позиції відбувається за допомогою методу POST з такими параметрами:

– ProductId – продукт, що доступний в позиції;
– Quantity – кількість доступного продукту.

API-сервіс отримує параметричний HTTP-метод POST, валідує його та записує повідомлення для створення нової позиції. Dispatcher-сервіс «слухає» всі оновлення по позиціям та під час отримання нового повідомлення формує подію з типом «Create» і зберігає її в сховище подій.

Також на етапі створення події з типом «Create» ініціалізується порожній агрегат на цю позицію та також зберігається у сховище. За шаблоном CQRS – агрегати та події зберігаються в окремих сховищах. На рисунку 2.9 зображена візуалізація вищевказаного процесу з усіма переліченими елементами.

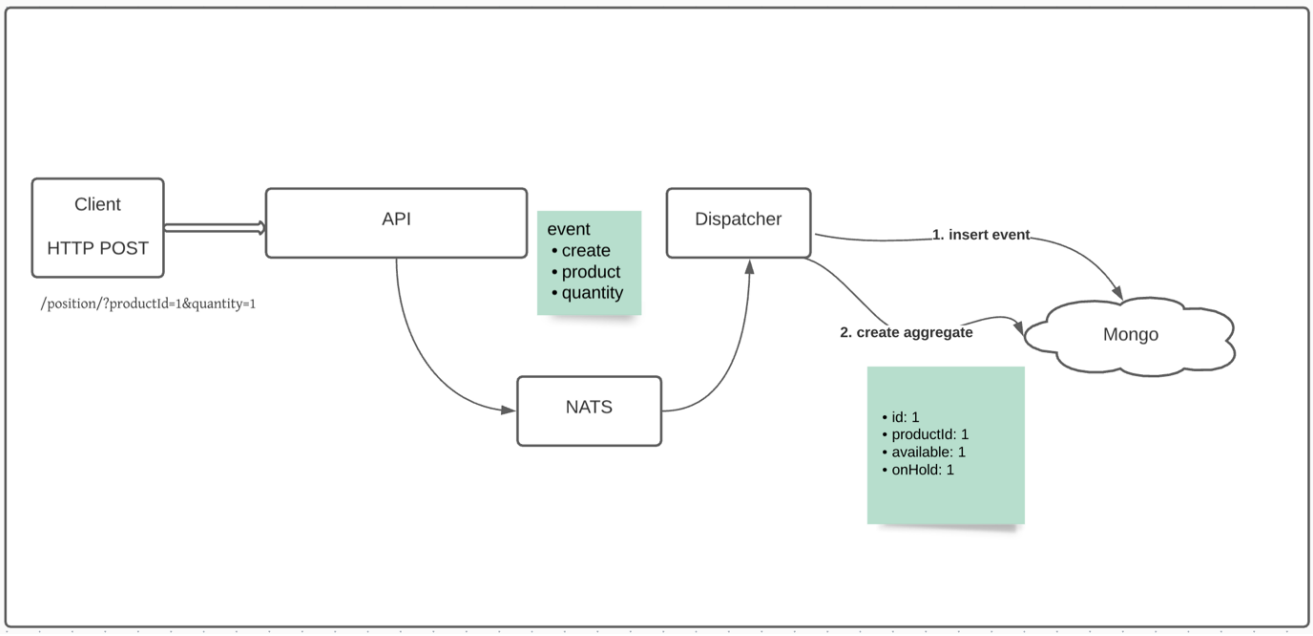


Рисунок 2.9 – Схематичне представлення взаємодії компонентів при створенні позиції

Отримання всіх позицій відбувається за допомогою HTTP-методу GET і, в залежності від вказаного шляху, можна обрати позиції, створені поточним користувачем, або ж всі доступні позиції до утримання. API-сервіс виконує функцію читача зі сховища агрегатів та повертає необхідні дані у форматі JSON. Даний підхід забезпечує миттєву обробку запитів, що є надзвичайно важливим

при розробці високонавантажених систем. На рисунку 2.10 зображена візуалізація вищеописаного процесу з усіма переліченими елементами.

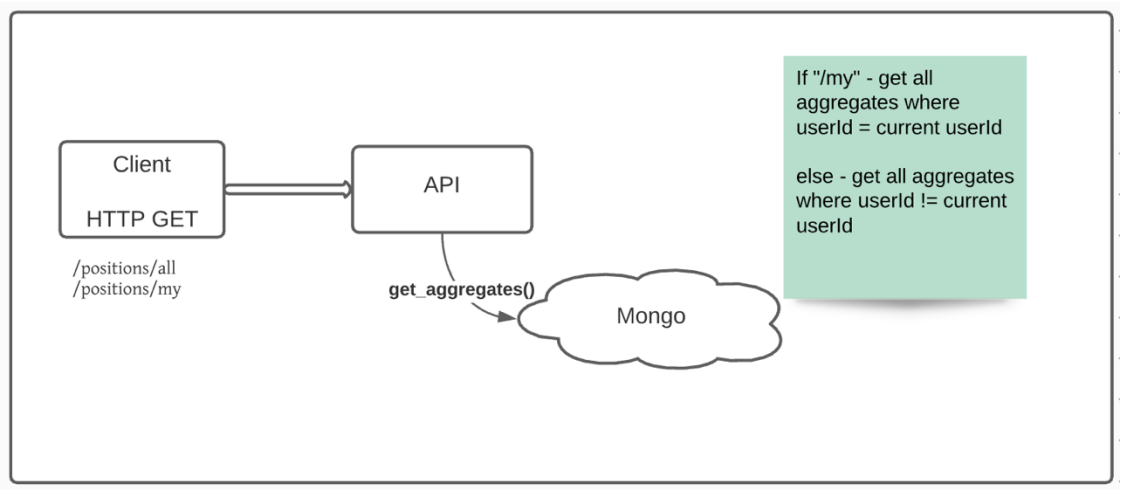


Рисунок 2.10 – Схематичне представлення взаємодії компонентів при отриманні позицій

Одним з найскладніших елементів системи, що керується подіями, є забезпечення узгодженості даних. Розподілені менеджери блокувань (lock-managers) використовуються для організації та серіалізації доступів до ресурсів. У нашому випадку ситуація неузгодженості даних може виникнути коли два користувачі одночасно створюють утримання на одну і ту ж позицію. У такому разі, при отриманні кожного запиту, необхідно «блокувати» оновлення позиції, поки попередні зміни не будуть збережені. На рисунку 2.11 можна побачити схему взаємодії всіх елементів, задіяних у процесі створення утримання.

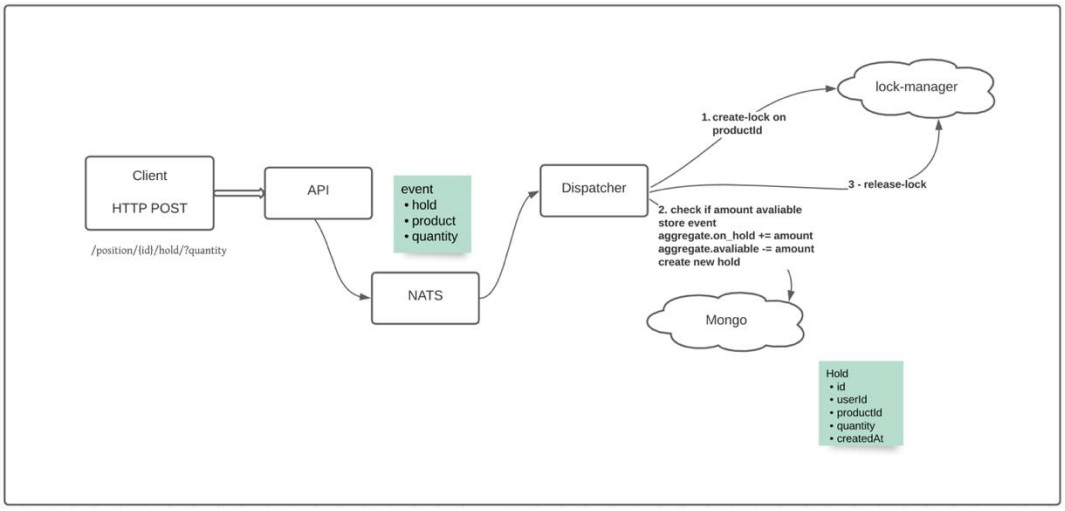


Рисунок 2.11 – Схема взаємодії компонентів при створенні утримання

Процес повернення усіх створених утримань відбувається схожим на процес отримання усіх позицій – за допомогою GET-методу. Є два різні методи: отримання всіх утримань користувача та отримати всі утримання по обраній позиції (доступно для користувачів, що мають відкриті позиції). На рисунку 2.12 зображена схема взаємодії компонентів при обробці отримання утримань.

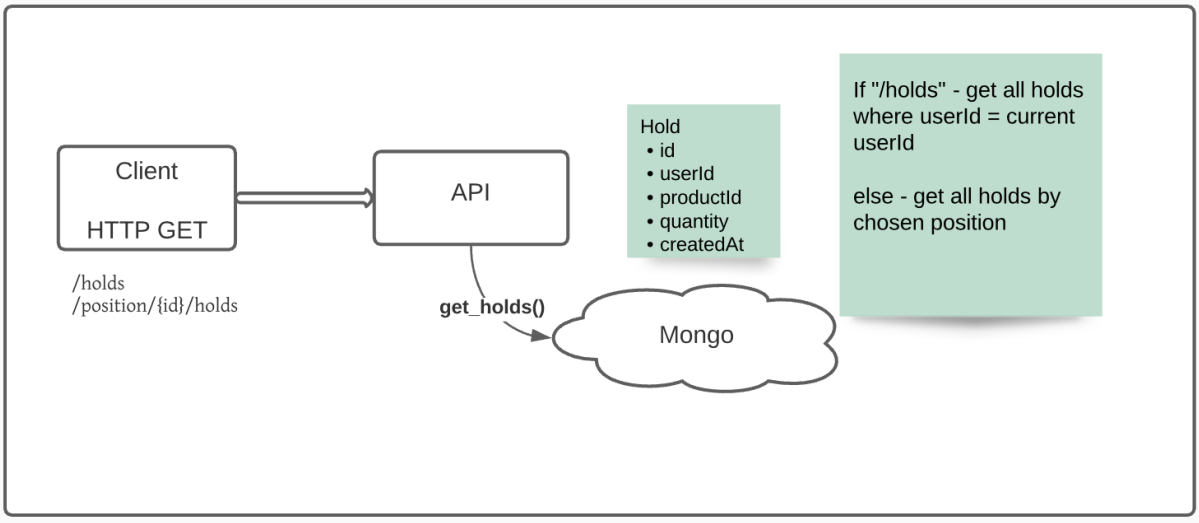


Рисунок 2.12 – Схема взаємодії компонентів при отриманні утримань

Таким чином, було детально розглянуто внутрішню архітектуру обробки запитів, створення команд та записів подій, якими оперує дана система.

2.4 Вибір технологій для розробки серверної частини

Найпоширенішою практикою для декомпозиції мікросервісів є проектування мікросервісу з обмеженим контекстом, що є ключовою ідеєю предметно-орієнтованого дизайну, який описаний у першому розділі. Цей принцип забезпечує логічне розділення бізнес-проблеми на різні піддомени шляхом розділення великого домену на обмежені контексти. Як правило для кожного обмеженого контексту розробляється мікросервіс.

Для того, щоб вирішити практичні завдання, пов’язані з мікросервісною архітектурою, необхідно визначитись із технологіями для розробки.

Технології, з якими необхідно визначитись при розробці системи, що керується подіями:

– мова програмування;
– технологія обміну повідомленнями;
– сховище (для подій);
– технології для швидкого розгортання;

2.4.1 Вибір мови програмування

Вибір мови програмування напряму залежить від дизайну архітектури системи. Необхідно звертати увагу на те, які функції має виконувати застосунок, архітектура буде монолітна чи мікросервісна, наскільки навантажена планується система та ресурси, які є доступними для розробки.

У даному випадку розробляється система, яка спрямована на вирішення завдань розподілених даних, асинхронних робочих процесів, а також високої продуктивності та масштабування.

Golang – це мова програмування з відкритим вихідним кодом, що дозволяє легко будувати просте, надійне та ефективне програмне забезпечення. Ця мова ідеально підходить для швидкої розробки мікросервісів.

Переваги мови програмування Golang:

– код легкий для сприйняття;
– ефективна робота програми в багатоядерних системах (багатопоточність);
– висока захищеність від помилок.

Дана мова програмування надає можливість створити легко масштабовані, високоефективні сервіси та швидко додавати нові елементи. При виборі мови програмування, порівняння здійснювалося між Go з Python, Ruby, Node, PHP та Java. Хоча кожна мова мала свої сильні сторони, Go найкраще відповідає задачам розробки архітектури даної системи. Крім того, порівняно простий синтаксис Go полегшив підбір і початок написання робочого коду.

У результаті аналізу було визначено, що Go – це найкраща мова для зручності та ефективності в розподіленій хмарній системі.

2.4.2 Вибір технології обміну повідомленнями

Для того, щоб опублікувати подію, яка сигналізує іншим сервісам, про зміну в стані системи, необхідно використовувати системи обміну повідомленнями. Існує велика кількість розподілених черг та систем обміну повідомленнями, які доступні для побудови сучасних розподілених систем.

Системи з відкритим кодом такі як Kafka, NATS, RabbitMQ, Google Cloud Pub/Sub, Amazon SQS та інші, забезпечують різні можливості та підходи в побудові розподілених систем. Оскільки, розподілені системи – дуже складні в проектуванні, сучасна система обміну повідомленнями має бути легкою у використанні та підходити для масштабування в таких середовищах як хмарні платформи, контейнери чи локальні сервіси.

NATS – це високоефективна система обміну повідомленнями з відкритим кодом. Вона імплементує систему «публікація-підписка» з підтримкою високого рівня масштабування. Незважаючи на те, що вона базується на розподіленій

моделі «публікація-підписка», даний інструмент чудово підходить також для розподіленого зчитування через групи черг підписників.

Сервер NATS є одним з найбільш продуктивних систем обміну повідомленнями, що надає можливість надсилати 15-18 мільйонів повідомлень в секунду. Платформа NATS проста у використанні та масштабуванні. Простота та ефективність робить NATS чудовим вибором у проектуванні розподілених мікросервісних систем як представлено у цій роботі. NATS написала на Golang, що надає їй багато переваг для використання в даному проекті.

Було вирішено обрати NATS як систему обміну повідомленнями для проектування та реалізації даного проекту.

2.4.3 Вибір сховища даних

MongoDB – це документо-орієнтована система керування базами даних з відкритим вихідним кодом, що не потребує схем таблиць. Вона підтримує зберігання в JSON-подібному форматі та має дуже гнучку мову для формування запитів, яка має можливість створювати індекси для різноманітних атрибутів, забезпечує ефективне збереження великих бінарний даних, підтримує парадигму Map/Reduce та багато іншого.

У MongoDB є вбудовані засоби із забезпечення шардінгу (розподіл набору даних по серверах на основі певного ключа), комбінуючи який з реплікацією даних можна побудувати горизонтально масштабований кластер зберігання (що є неймовірно важливим для розподілених систем).

Для легкої роботи з Mongo, можна використовувати MongoDB Compass – користувацький графічний інтерфейс для даної бази даних. Він дозволяє легко аналізувати та розуміти вміст даних, не використовуючи класичний MongoDB синтаксис. На додаток до цих переваг, Compass також надає можливість оптимізувати час виконання запитів, керувати індексами чи виконувати

валідування даних. На рисунку 2.13 зображений вигляд інтерфейсу MongoDB Compass.

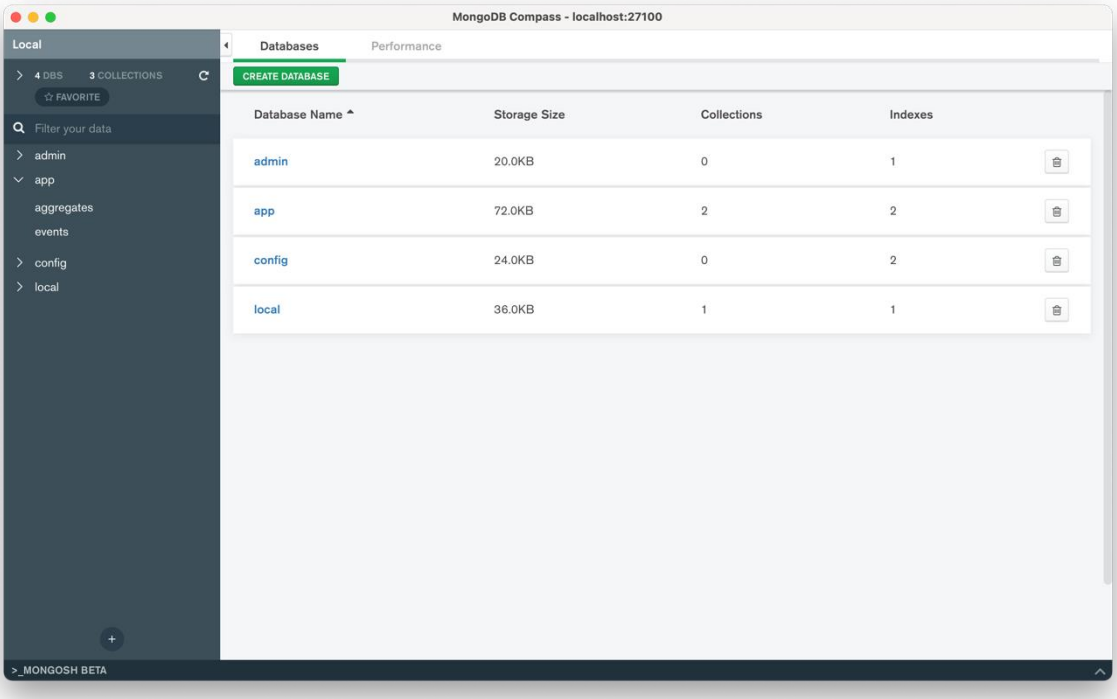


Рисунок 2.13 – Графічний інтерфейс MongoDB Compass

2.4.4 Допоміжні технології

1. Docker

Docker – ПЗ з відкритим вихідним кодом, що використовується для автоматичного розгортання застосунків всередині контейнерів.

У даній роботі було використано технологію Docker для швидкого розгортання проекту. Це один з найпоширеніший сучасних методів, що дозволяє легко пакувати, доставляти та запускати будь-які програми у вигляді легкого, портативного та незалежного контейнеру, який може працювати в будь-якому середовищі.

Docker Compose – це інструмент, для визначення та запуску мультиконтейнеризованого Docker застосунку. Використовуючи єдину команду,

можна створити та запустити всі сервіси з налаштувань. Це надзвичайно ефективний та зручний інструмент для швидкого розгортання, що ідеально підходить для поточної мікросервісної архітектури, по перевагам якому немає рівних на сучасному ринку.

2. Postman

Postman – це інструмент, для тестування RESTful APIs. Він пропонує елегантний користувальницький інтерфейс, за допомогою якого можна надсилати HTML-запити, щоб уникнути написання купи коду лише для того, щоб перевірити функціональність API.

В даній роботі цей інструмент використовується для тестування запитів до API-сервісу.

Висновки по розділу II

У даному розділі було проведено детальний аналіз особливостей систем, що керуються подіями, проведено порівняльну характеристику даних подій з системами, що CRUD для комунікації. Спроектовано високорівневу архітектуру системи, розглянуто її основні характеристики та елементи.

Розроблено схеми для кожного компоненту системи, розглянуто атрибути кожного компоненту, їх типи та властивості. Спроектовано схеми для основного функціоналу та проведено їх детальний аналіз з використанням графічних елементів.

Було проведено детальний аналіз існуючих технологій, для серверної частини, та обрано ті, які найкраще підходять для розробки даної системи.

РОЗДІЛ ІІІ. РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1 Розробка основних мікросервісів для серверної частини

Після визначення з технологіями, необхідними для розробки даного продукту, було складено план розробки, для пришвидшення написання коду.

Послідовність розробки:

– старт базових сервісів: API та Dispatcher;
– описання docker-compose.yaml файлу для зручного запуску;
– опис основних функцій для роботи з NATS;
– підключення NATS до основних сервісів;
– опис функцій для роботи з MongoDB;
– підключення MongoDB до основних сервісів;
– створення та вдосконалення REST методів.
– тестування базових функцій, використовуючи Postman.

Під час розробки мікросервісної архітектури, сервіси найчастіше розділяють на репозиторії для зручності управління кодовою базою, але існує й інший підхід, такий як монорепозиторій. Монорепозиторій – це один репозиторій, який зберігає весь код та додатки для проекту.

Переваги використання монорепозиторію:

– зручність в управлінні залежностями;
– пришвидшення розробки;
– покращення в співпраці.

Цей підхід було застосовано і в даній роботі, для пришвидшення розробки та зручності.

3.1.1 Запуск сервісів мовою програмування Golang

Як вже було зазначено в другому розділі, мова програмування Go є простою та зручною у використанні.

Програми, написані цією мовою, групуються в «пакети». Пакет – це колекція вихідних файлів в одній і тій же директорії, що компілюються разом. Функції, типи, змінні та константи, що визначені в одному файлі, будуть доступні у всіх файлах в межах одного пакету. Репозиторій може складатися з одного чи більше модулів. Модуль – це колекція суміжних пакетів, які випущені разом. Зазвичай один репозиторій вміщає один модуль, розташований у корені репозиторію. Кожен виконуваний файл мусить містити ім'я пакету. Усі виконувані команди завжди знаходяться в пакеті «main».

Розглянемо приклад запуску звичайної програми, що виводить на екран речення: «Привіт, Світ!». На рисунку 3.1 зображене створення нового проекту у середовищі розробки Goland від JetBrains.

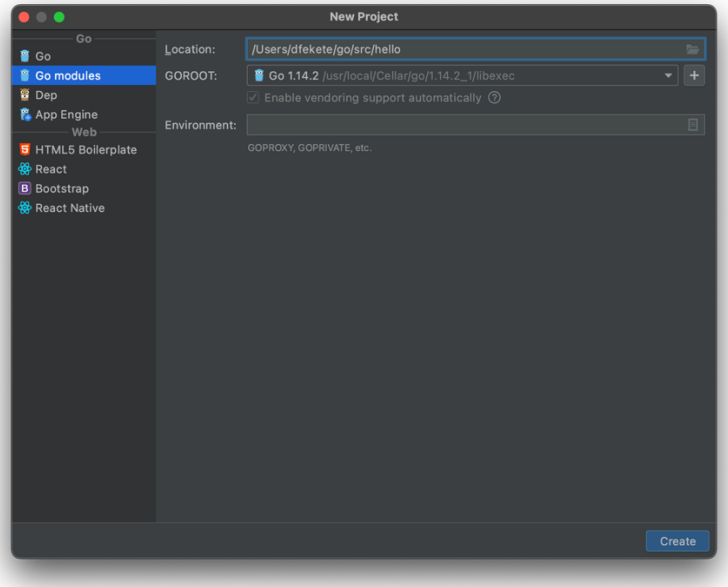


Рисунок 3.1 – Створення нового проекту в IDE Goland

Оскільки я обрала створення проекту на основі модулів, то в корені проекту у мене буде створений go.mod файл – рисунок 3.2. Цей файл описує шлях модуля – в ньому розташовані шляхи усіх пакеті, що будуть імпортовані в модулі.

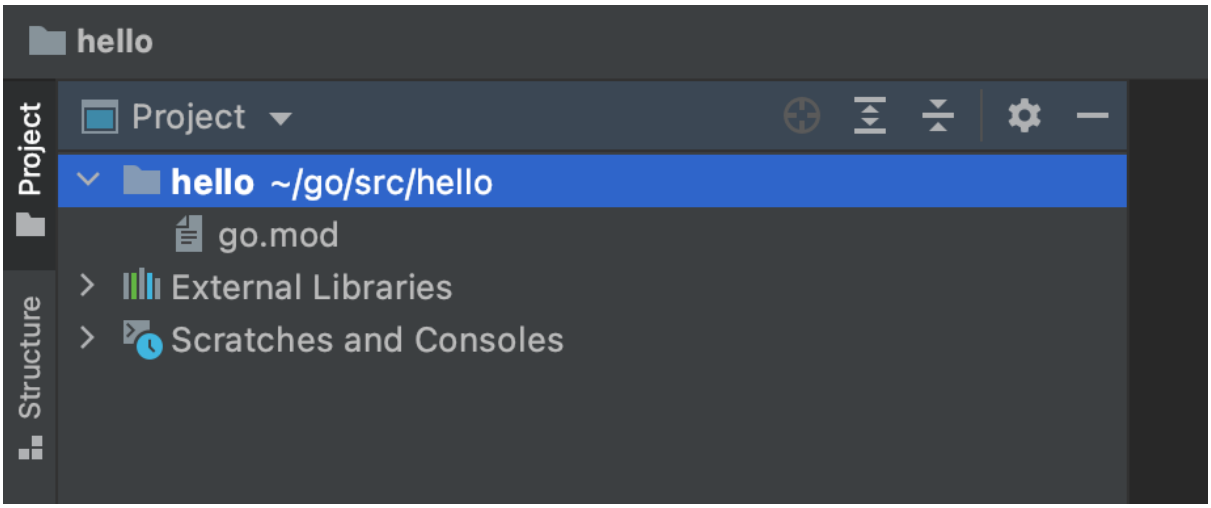


Рисунок 3.2 – Автоматично створений новий проект з файлом go.mod

Початковий вміст файлу go.mod зображений на рисунку 3.3. Він зберігає ім’я модуля (нашого проекту) та версію мови програмування, що використовується.

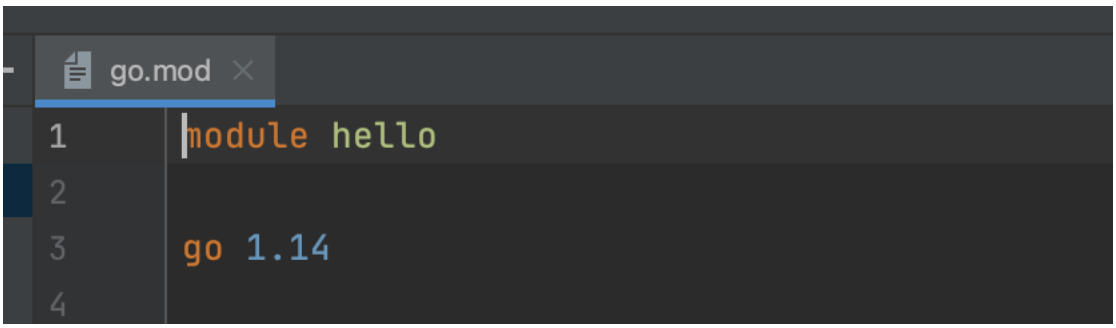


Рисунок 3.3 – Початковий вміст файлу go.mod

Наступний крок – створення програми для виведення тексту в консоль. Як вже було зазначено всі виконувані команди знаходяться в пакеті «main». На рисунку 3.4 зображений код основної програми.

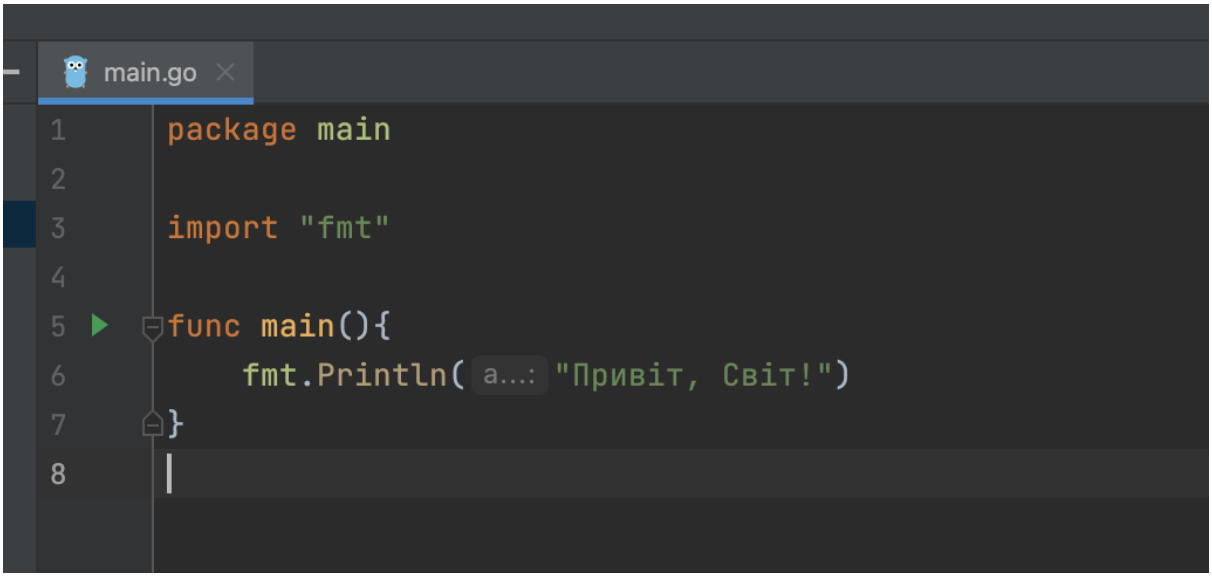


Рисунок 3.4 – Основний код програми

Щоб запустити програму необхідно запустити дві команди (рисунок 3.5):

– «go install ./…», яка створює скомпільований файл у папці pkg та виконуваний файл з розширенням .exec у каталозі bin;
– «go run main.go», яка компілює та запускає код (але не створює двійкові файли в папці pkg та bin, як це робить попередня команда).

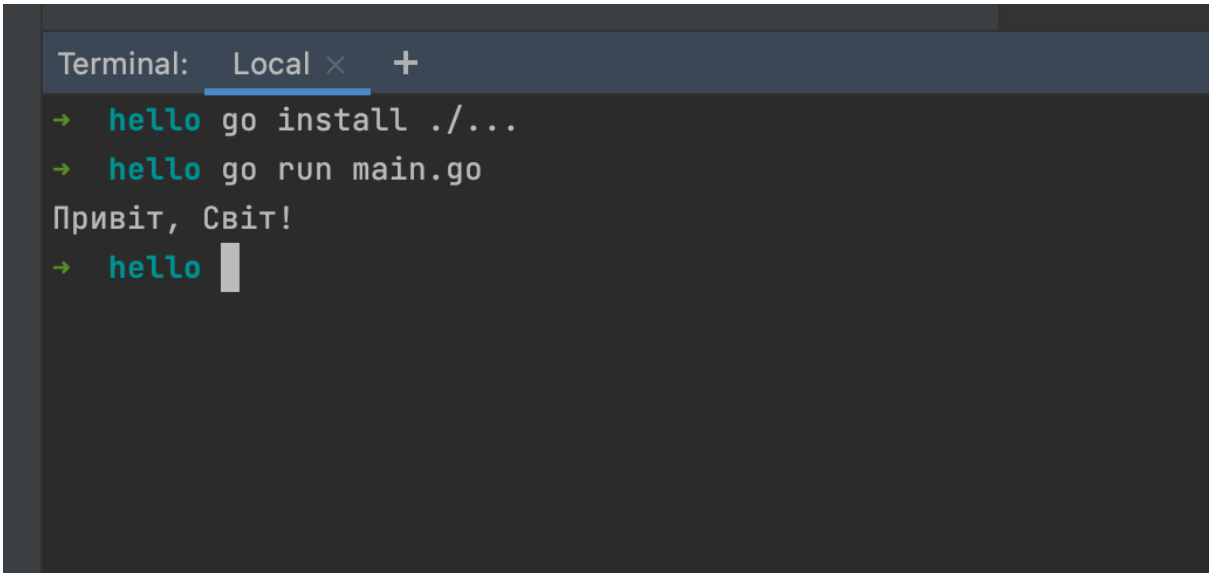


Рисунок 3.5 – Запуск базової програми мовою програмування Go

3.1.2 Базові підключення. Налаштування NATS та бази даних MongoDB

В основі структури даної роботи лежать два головні сервіси: API та Dispatcher. Кожен з них розташований в однойменному пакеті основного монорепозиторію. Для швидкого запуску було обрано технологію docker compose.

На основі дослідження обов'язків та зв'язків між сервісами з другого розділу – можна сформувати файл docker-compose.yaml. Цей файл використовується для визначення сервісів, з яких складається застосунок та які будуть запускатись в ізольованому середовищі.

Для даної роботи необхідно підключити NATS та MongoDB. Використовуючи доступні докер образи, опишемо підключення необхідних сервісів на рисунку 3.6.

Рисунок 3.6 – Підключення сервісів Nats та MongoDB в docker-compose.yaml

Для Nats необхідно вказати назву образу та його версію, а також частоту перезапуску. Для бази даних необхідно вказати назву образу та його версію (використовуємо останню), ім'я контейнеру, в якому вона стартує, та порт, на якому буде розгорнута.

У структурі проекту, спільний код для роботи з Nats знаходиться в папці event. Дана папка вміщає в собі три файли: event.go, nats.go та position.go.

У файлі nats.go описано основні функції для роботи з цією технологією. Функція NewNats(), приймає посилання на розгорнутий NATS сервер та використовується для підключення до сервісу та повертає помилку, якщо під'єднання невдале. На рисунку 3.7 зображений опис даної функції.

```go
func NewNats(url string) (*NatsEventStore, error) {
    nc, err := nats.Connect(url)
    if err != nil {
        return nil, err
    }
    return &NatsEventStore{nc: nc}, nil
}
```

Рисунок 3.7 – Реалізація функції для підключення до Nats

NatsEventStore – це структура, зображена на рисунку 3.8, яка зберігає підключення до серверу, вказівник на підписку Позицій та канал, для збереження опублікованих Позицій. Канал в мові програмування Go – структура створена для обміну інформацією між потоками виконання програми, та куди також можна поміщати дані та отримувати їх, використовуючи оператор «<–».

```go
import ...

type NatsEventStore struct {
    nc                            *nats.Conn
    publishedPositionSubscription *nats.Subscription
    publishedPositionChan         chan PublishedPosition
}
```

Рисунок 3.8 – Структура NatsEventStore

Над поданою структурою можна виконувати різноманітні функції. Розглянемо деякі з них. Функція Close(), реалізація якої зображена на рисунку

3.9, була створена для закриття підключення за потреби. Вона закриває підключення, відписується від підписок та закриває канал, який зберігав опубліковані повідомлення.

```
func (e *NatsEventStore) Close() {
    if e.nc != nil {
        e.nc.Close()
    }
    if e.publishedPositionSubscription != nil {
        e.publishedPositionSubscription.Unsubscribe()
    }
    close(e.publishedPositionChan)
}
```

Рисунок 3.9 – Реалізація функції Close()

Функція PublishPositionsUpdates(), що зображена на рисунку 3.10, приймає позицію та створена для публікації повідомлення, у даному випадку публікації оновлення Позиції. Також на цьому ж рисунку зображена допоміжна функція writeMessage(), що використовується для перетворення вхідних даних в байти (для передачі повідомлення).

```
func (e *NatsEventStore) PublishPositionUpdates(pos schema.Position) error {
    m := PublishedPosition{ ID: pos.ID, ProductId: pos.ProductId, Type: pos.Type.String(), Quantity: pos.Quantity, CreatedAt: pos.CreatedAt}
    data, err := e.writeMessage(&m)
    if err != nil {
        return err
    }
    return e.nc.Publish(m.Key(), data)
}

func (mg *NatsEventStore) writeMessage(p Position) ([]byte, error) {
    b := bytes.Buffer{}
    err := gob.NewEncoder(&b).Encode(p)
    if err != nil { nil, err }
    return b.Bytes(), nil
}
```

Рисунок 3.10 – Реалізація глобальної функції PublishPositionsUpdates та допоміжної writeMessage

Функція SubscribePositionUpdates() дозволяє слухати підписку, отримувати повідомлення, декодувати їх та повертати отримані повідомлення в каналі (чи

помилку, якщо така виникла). Реалізація даної функції в коді зображена на рисунку 3.11.

```
func (es *NatsEventStore) SubscribePositionUpdates() (<-chan PublishedPosition, error) {
    m := PublishedPosition{}
    es.publishedPositionChan = make(chan PublishedPosition, 64)
    ch := make(chan *nats.Msg, 64)
    var err error
    es.publishedPositionSubscription, err = es.nc.ChanSubscribe(m.Key(), ch)
    if err != nil { return nil, err }
    // Decode message
    go func() {
        for {
            select {
            case msg := <-ch:
                if err := es.readMessage(msg.Data, &m); err != nil {
                    log.Fatal(err)
                }
                es.publishedPositionChan <- m
            }
        }
    }()
    return es.publishedPositionChan, nil
}
```

Рисунок 3.11 – Реалізація функції SubscribePositionUpdates()

Як можна помітити, попередня функція викликає ще одну допоміжну функцію readMessage(), реалізація якої зображена на рисунку 3.12. Вона приймає дані в байтах та декодує їх, повертаючи при цьому помилку, якщо така виникла.

```
func (mq *NatsEventStore) readMessage(data []byte, m interface{}) error {
    b := bytes.Buffer{}
    b.Write(data)
    return gob.NewDecoder(&b).Decode(m)
}
```

Рисунок 3.12 – Реалізація допоміжної функції readMessage()

Вище були зазначені основні функції для роботи з шаблоном «публікація-підписка» Nats, використаний для операцій над Позиціями. Аналогічним чином розробляються й функції для отримання Утримань тощо. В майбутньому можна

універсалізувати такі функції, які будуть приймати й повертати повідомлення в залежності від вказаного типу.

Файли для роботи з базою знаходяться в пакеті mongo. Необхідно описати структуру моделей, які будуть зберігатись, та саму структуру сховища. Файл storage.go містить функції необхідні для підключення та закриття підключення з базою даних. На рисунку 3.13 зображена структура Storage, яка містить в собі клієнт та саму базу даних, яка використовується.

Рисунок 3.13 – Структура Сховище (Storage)

На рисунку 3.14 зображена функція NewMongo(), що використовується для створення нового підключення та приймає посилання для підключення та ім’я бази даних, до якої необхідно підключитись. Також там відбувається виклик клієнту, щоб перевірити, що підключення вдале.

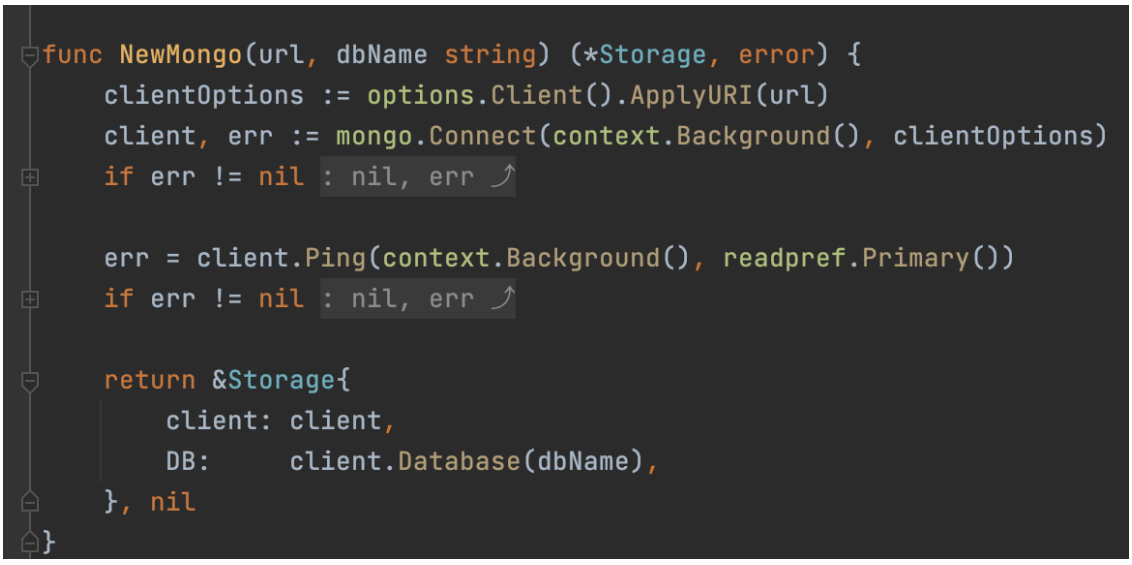


Рисунок 3.14 – Функція для підключення до MongoDB

Аналогічно необхідно запрограмувати функцію Cancel(), яка надасть можливість швидкого відключення від бази у разі необхідності.

Моделі, які використовуються для збереження записів, знаходяться в пакеті model та однойменному файлі model.go. На рисунку 3.15 зображена структура події, яка буде збережена.

```
type Event struct {
    ID        primitive.ObjectID `bson:"_id,omitempty"`
    UserID    string             `bson:"userId"`
    ProductId string             `bson:"productId,omitempty"`
    Type      string             `bson:"type,omitempty"`
    Quantity  int                `bson:"quantity,omitempty"`
    CreatedAt primitive.DateTime `bson:"createdAt,omitempty"`
}
```

Рисунок 3.15 – Опис структури «Подія» (Event)

3.1.3 Розробка API сервісу

API-сервіс – сервіс для комунікації з клієнтською частиною та відсилання команд для Dispatcher-сервісу. Розгортається на порті 8181 та залежить від сервісів Nats та Mongo. На рисунку 3.16 зображений опис сервісу в docker-compose.yaml, окрім вищеописаних правил також передаються дві змінні: адреса Nats та MongoDB, що будуть використовуватись для підключення.

Рисунок 3.16 – Опис сервісу в docker-compose.yaml з необхідними залежностями

Розглянемо основну структуру головного файлу сервісу main.go. На рисунку 3.17 зображена структура функції main() – функція, що містить всі головні виконувані команди.

1. зчитування конфігурації (в якій містяться змінні для Nats та Mongo);
2. підключення до MongoDB, використовуючи функції з пакету mongo, які були розглянуті розділі 3.1.2;
3. підключення до Nats, використовуючи функції з пакету event, які були розглянуті розділі 3.1.2;
4. старт http серверу та створення маршрутизатора для запитів.

```
func main() {
    var cfg Config
    err := envconfig.Process( prefix: "", &cfg)
    if err != nil {
        log.Fatal(err)
    }

    storage, err := mongo.NewMongo(fmt.Sprintf("mongodb://#{cfg.MongoDbAddress}"), dbName: "app")
    if err != nil {
        log.Fatal(err)
    }
    defer storage.Cancel()

    // connects to NATS
    retry.ForeverSleep(2*time.Second, func(_ int) error {
        es, err := event.NewNats(fmt.Sprintf( format: "nats://%s", cfg.NatsAddress))
        if err != nil {
            log.Println(err)
            return err
        }
        event.SetEventStore(es)
        return nil
    })
    defer event.Close()

    router := newRouter(storage)
    if err := http.ListenAndServe( addr: ":8181", router); err != nil {
        log.Fatal(err)
    }
}
```

Рисунок 3.17 – Головна функція сервісу API

HTTP Router – це процесор, що передає дані, базуючись на HTTP методі та URL шляху в записі заголовку атрибутів.

Основні функції для роботи з HTTP розташовані в файлі handler, а на рисунку 3.18 зображена функція з файлу main.go, що реєструє шлях та вказує йому у відповідність обробник запиту.

```
router.HandleFunc( path: "/position", createPositionHandler).
    Methods( methods...: "POST").
    Queries( pairs...: "productId", "{productId}", "quantity", "{quantity}")
```

Рисунок 3.18 – Реєстрація шляху для створення Позиції

Обробник запитів – це функція, яка містить бізнес-логіку, що має виконатись при запиті з клієнта. У ній описується парсинг атрибутів, маніпуляції з даними тощо. На рисунку 3.19 зображений обробник створення нової Позиції. Логіка, що лежить в основі функції:

– зчитати параметри та провалідувати їх;

– сформувати нову Позицію;

– опублікувати нову Позицію, використовуючи Nats;

– повернути ID створеної позиції.

В обов'язковому порядку, кожна потенційна помилка має бути оброблена та збережена.

```go
// positions
func createPositionHandler(w http.ResponseWriter, r *http.Request) {
    type response struct {
        ID string `json:"id"`
    }

    // Read parameters
    productId := template.HTMLEscapeString(r.FormValue( key: "productId"))
    if len(productId) < 1 || len(productId) > 140 {
        util.ResponseError(w, http.StatusBadRequest,  message: "Invalid productId")
        return
    }

    quantity := template.HTMLEscapeString(r.FormValue( key: "quantity"))
    quantityInt, err := strconv.Atoi(quantity)
    if err != nil {
        util.ResponseError(w, http.StatusBadRequest,  message: "Invalid quantity")
        return
    }

    createdAt := time.Now().UTC()
    id := primitive.NewObjectID()

    position := schema.Position{
        ID:        id.Hex(),
        ProductId: productId,
        Quantity:  quantityInt,
        Type:      schema.Create,
        CreatedAt: createdAt,
    }

    // Publish event
    if err := event.PublishPositionUpdates(position); err != nil {
        log.Println(err)
    }
    log.Printf("Published new position: #{position}")

    // Return new position
    util.ResponseOk(w, response{ID: position.ID})
}
```

Рисунок 3.19 – Обробник запиту для створення нової Позиції

Обробники інших методів працюють за аналогією до вищевказаного. Наприклад, функція для отримання усіх поточних позицій та реєстрація шляху для даного методу зображені на рисунку 3.20 та 3.21 відповідно.

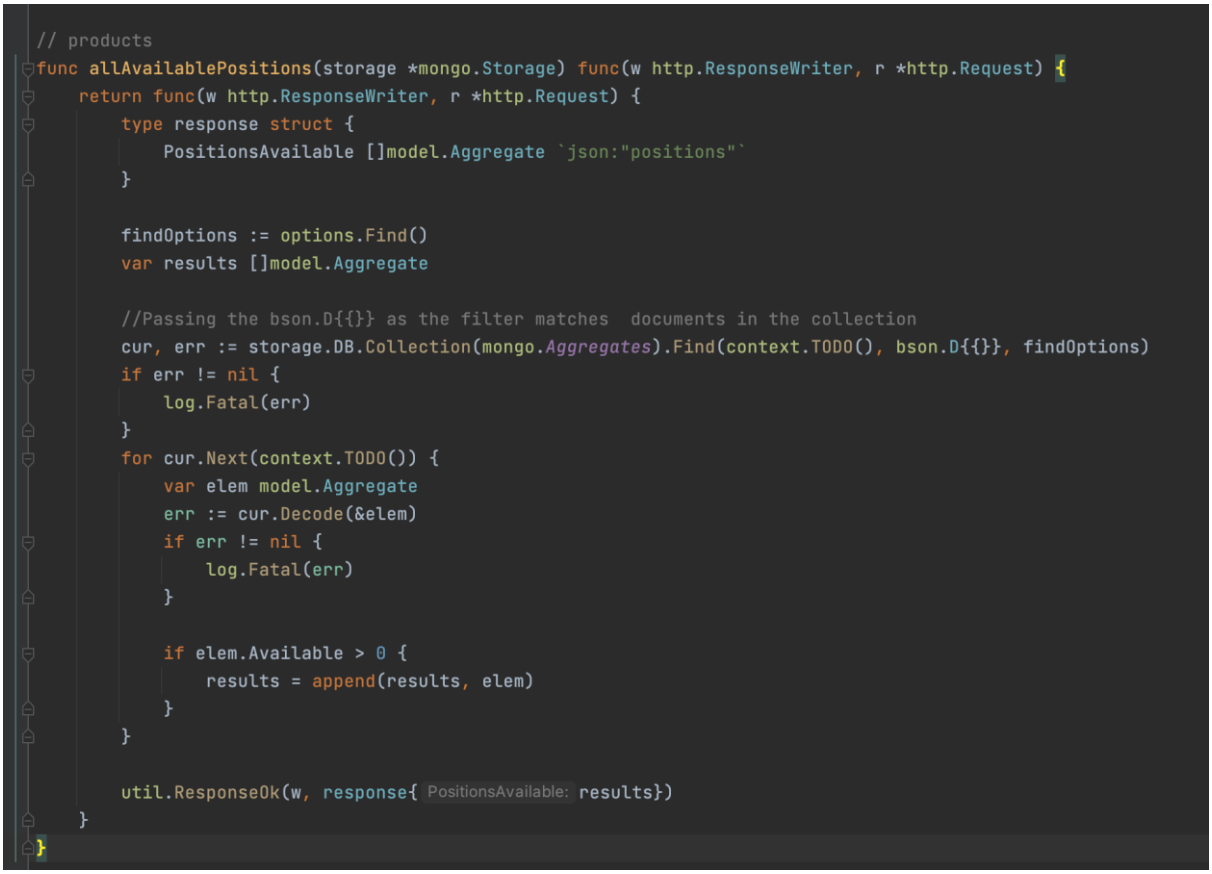


Рисунок 3.20 – Обробник запиту для отримання усіх доступних Позицій

Рисунок 3.21 – Реєстрація шляху для отримання усіх Позицій

3.1.4 Розробка Dispatcher сервісу

Dispatcher – сервіс для обробки команд від API-сервісу, створення подій і їх агрегатів та запис їх в сховище. Очевидно, що сервіс залежить від Nats та MongoDB, саме тому при описі сервісу в docker compose файлі, додаємо

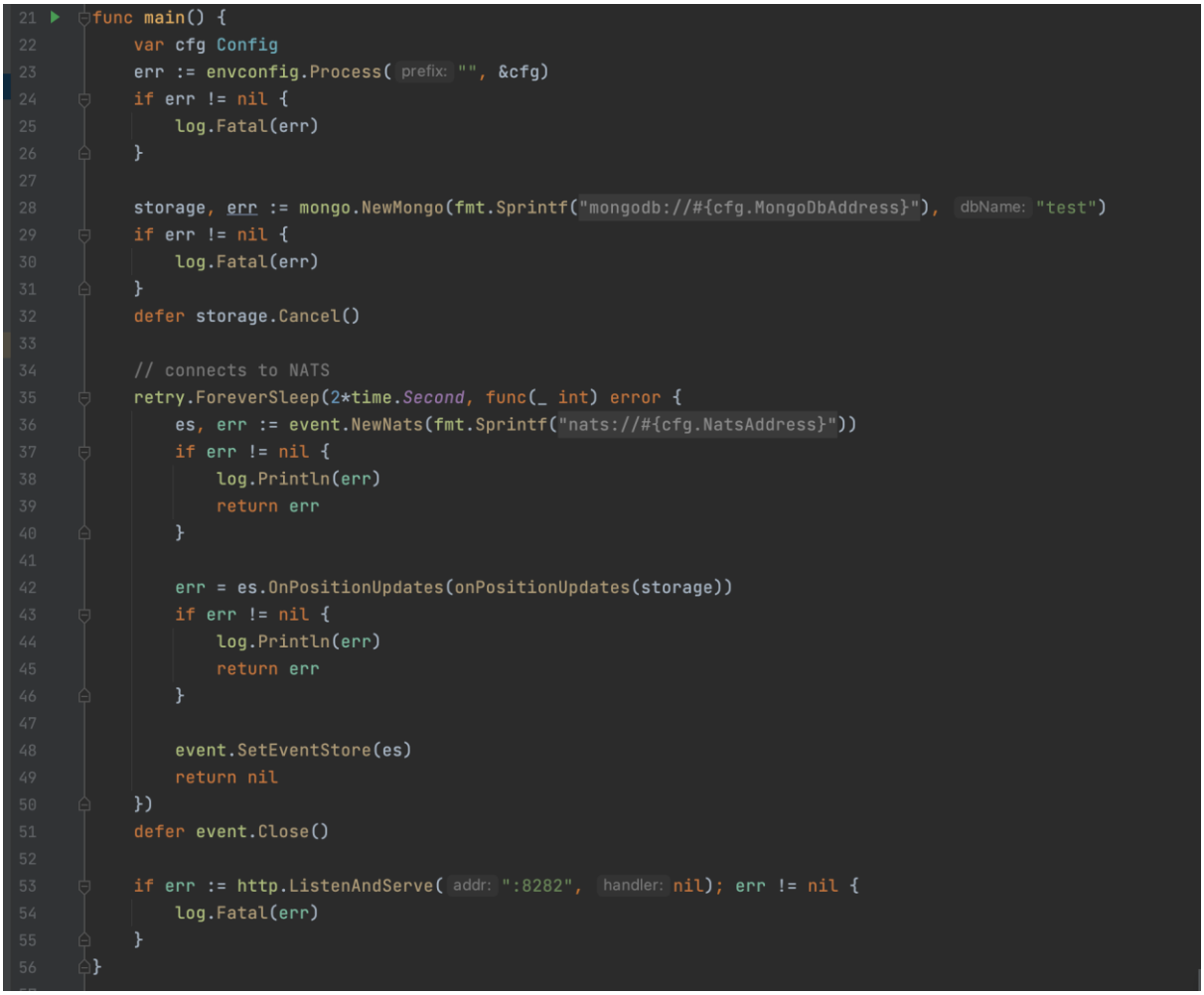
залежності на ці інструменти. Dispatcher буде розгорнутий на порті 8282. На рисунку 3.22 зображений опис даного сервісу в docker compose файлі.

```yaml
dispatcher:
  build: "."
  command: "dispatcher-service"
  ports:
    - "8282:8282"
  depends_on:
    - "nats"
    - "mongo"
  environment:
    NATS_ADDRESS: "nats:4222"
    MONGODB_ADDRESS: "mongo:27017"
```

Рисунок 3.22 – Опис сервісу в docker-compose.yaml з необхідними залежностями

Розглянемо основну структуру головного файлу сервісу main.go. На рисунку 3.23 зображена структура функції main() – функція, що містить всі головні виконувані команди.

1. зчитування конфігурації (в якій містяться змінні для Nats та Mongo);
2. підключення до MongoDB, використовуючи функції з пакету mongo, які були розглянуті розділі 3.1.2;
3. підключення до Nats, використовуючи функції з пакету event, які були розглянуті розділі 3.1.2 – хочу звернути увагу, що всередині даної функції викликається OnPositionUpdates(), яка слухає «тему» з оновленнями по Позиціям).



```go
func main() {
    var cfg Config
    err := envconfig.Process( prefix: "", &cfg)
    if err != nil {
        log.Fatal(err)
    }

    storage, err := mongo.NewMongo(fmt.Sprintf("mongodb://#{cfg.MongoDbAddress}"), dbName: "test")
    if err != nil {
        log.Fatal(err)
    }
    defer storage.Cancel()

    // connects to NATS
    retry.ForeverSleep(2*time.Second, func(_ int) error {
        es, err := event.NewNats(fmt.Sprintf("nats://#{cfg.NatsAddress}"))
        if err != nil {
            log.Println(err)
            return err
        }

        err = es.OnPositionUpdates(onPositionUpdates(storage))
        if err != nil {
            log.Println(err)
            return err
        }

        event.SetEventStore(es)
        return nil
    })
    defer event.Close()

    if err := http.ListenAndServe( addr: ":8282", handler: nil); err != nil {
        log.Fatal(err)
    }
}
```

Рисунок 3.23 – Головна функція сервісу Dispatcher

Під час того, як сервіс отримує повідомлення з «теми», необхідно обробити інформацію та створити необхідні події та їх агрегати. На рисунку 3.24 зображена перша частина функції, що обробляє отримані команди, для створення чи оновлення Позицій. Логіка, в основі цієї частини функції:

– згенерувати ID для нової події (на основі ID отриманої позиції);
– записати час на основі отриманого часу з Позиції;
– вивести на екран зчитане повідомлення;
– сформувати нову Подію;
– записати подію в колекцію Події в сховищі.

```go
func onPositionUpdates(storage *mongo.Storage) func(position event.PublishedPosition) {
    return func(p event.PublishedPosition) {
        id, err := primitive.ObjectIDFromHex(p.ID)
        if err != nil {
            log.Println(err)
        }
        createdAt := primitive.NewDateTimeFromTime(p.CreatedAt)

        log.Printf( format: "Readed new message in positions.updates: %v", p)

        evnt := model.Event{
            ID:        id,
            ProductId: p.ProductId,
            Type:      p.Type,
            Quantity:  p.Quantity,
            CreatedAt: createdAt,
        }

        // write new event in event store
        _, err = storage.DB.Collection(mongo.Events).InsertOne(context.Background(), evnt)
        if err != nil {
            log.Println(err)
        }
```

Рисунок 3.24 – Обробник отриманих команд на управління Позиціями (ч. 1)

Наступна частина функції зображена на рисунку 3.25 Логіка, в основі цієї частини функції:

– якщо це Позиція з типом «Create» – створити новий агрегат;
– записати в базу даних;
– перевірка записаних даних (логування).

```go
        // for new created position write an aggregate -
        if evnt.Type == schema.Create.String() {
            aggregate := model.Aggregate{
                ID:        id,
                ProductId: p.ProductId,
                Available: p.Quantity,
                OnHold:    0, // when position is created - 0 quantity is on hold
            }

            _, err = storage.DB.Collection(mongo.Aggregates).InsertOne(context.Background(), aggregate)
            if err != nil {
                log.Println(err)
            }
        }

        // CHECKERS!
        var data interface{}
        var data1 interface{}
        err = storage.DB.Collection(mongo.Events).FindOne(context.Background(), bson.M{"_id": id}).Decode(&data)
        if err != nil {
            log.Println(err)
        }

        err = storage.DB.Collection(mongo.Aggregates).FindOne(context.Background(), bson.M{"_id": id}).Decode(&data1)
        if err != nil {
            log.Println(err)
        }
        log.Println(data)
        log.Println(data1)
    }
}
```

Рисунок 3.25 – Обробник отриманих команд на управління Позиціями (ч. 2)

3.2 Запуск та тестування проекту на локальній машині

При проектуванні поточної системи, враховувалося те, що може виникнути необхідність розгортати даний проект на машинах з різними технічними характеристиками та з різними операційними системами. Саме тому, Docker став чудовим рішенням для запуску всієї системи.

У попередніх розділах було представлено структуру docker compose файлу, але для того, щоб запустити систему, використовуючи Docker необхідно також описати Dockerfile, який розміщується в корені проекту та структура якого зображена на рисунку 3.26.

```
FROM golang:1.10.2-alpine3.7 AS build
RUN apk --no-cache add gcc g++ make ca-certificates
WORKDIR /go/src/github.com/feketiko/services

COPY vendor vendor
COPY event event
COPY schema schema
COPY mongo mongo
COPY api-service api-service
COPY dispatcher-service dispatcher-service

RUN go install ./...

FROM alpine:3.7
WORKDIR /usr/bin
COPY --from=build /go/bin .
```

Рисунок 3.26 – структура Dockerfile

Команди, які використовуються для написання:

– FROM – визначає батьківський образ, на основі якого все будується (у цьому випадку це go образ);

– RUN – виконує команди в новому шарі поверх поточного образу та комітить зміни;
– WORKDIR – визначає головну директорію;
– COPY – копіює файли системи в файлову систему контейнера.

Перше, що необхідно зробити, – запустити Docker на локальній машині, як це зображено на рисунку 3.27.

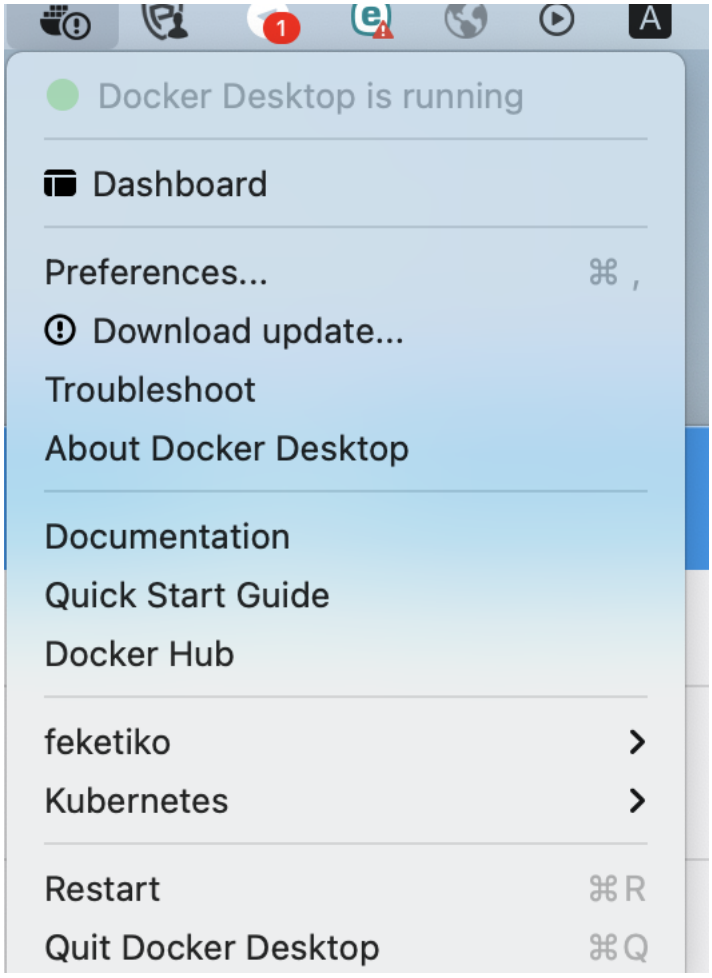


Рисунок 3.27 – Запущений Docker на локальній машині

Після цього необхідно відкрити термінал, перейти у корінь проекту та запустити наступну команду: docker compose up --build. Дана команда будує образи перед стартом контейнерів та агрегує вивід кожного контейнеру на екран. Як можна помітити на рисунку 3.28, спочатку виконується Dockerfile.

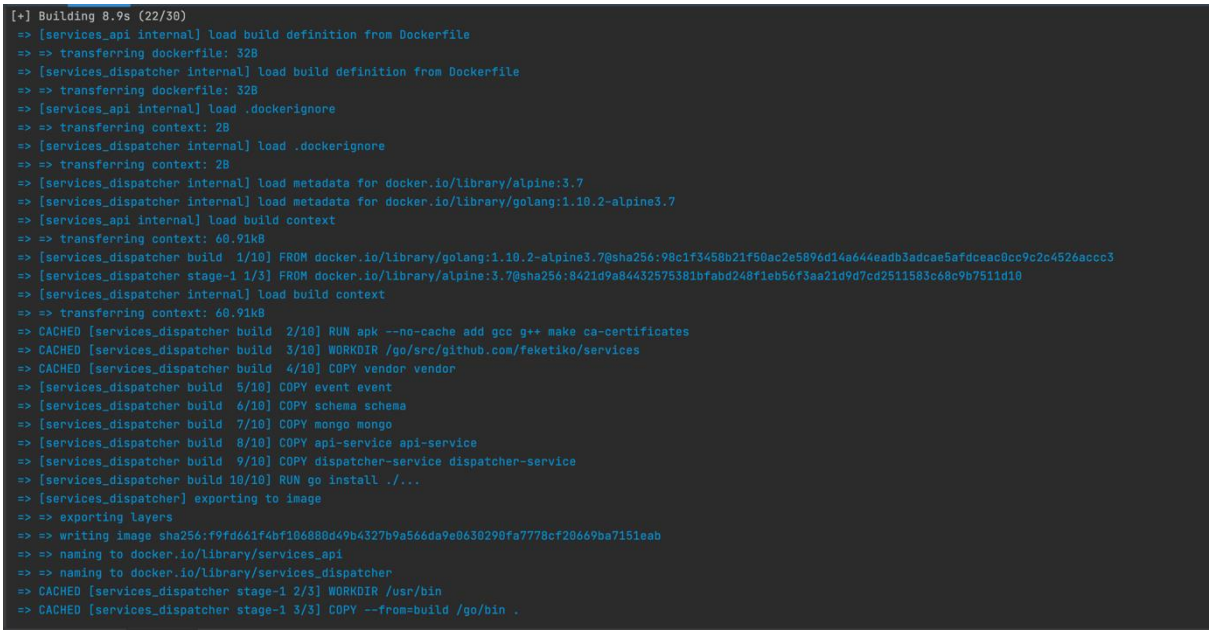


Рисунок 3.28 – Виконання Dockerfile

Відстежувати поведінку та логи всієї системи можна або в терміналі, чи через GUI Docker-a, як це зображено на рисунку 3.29.

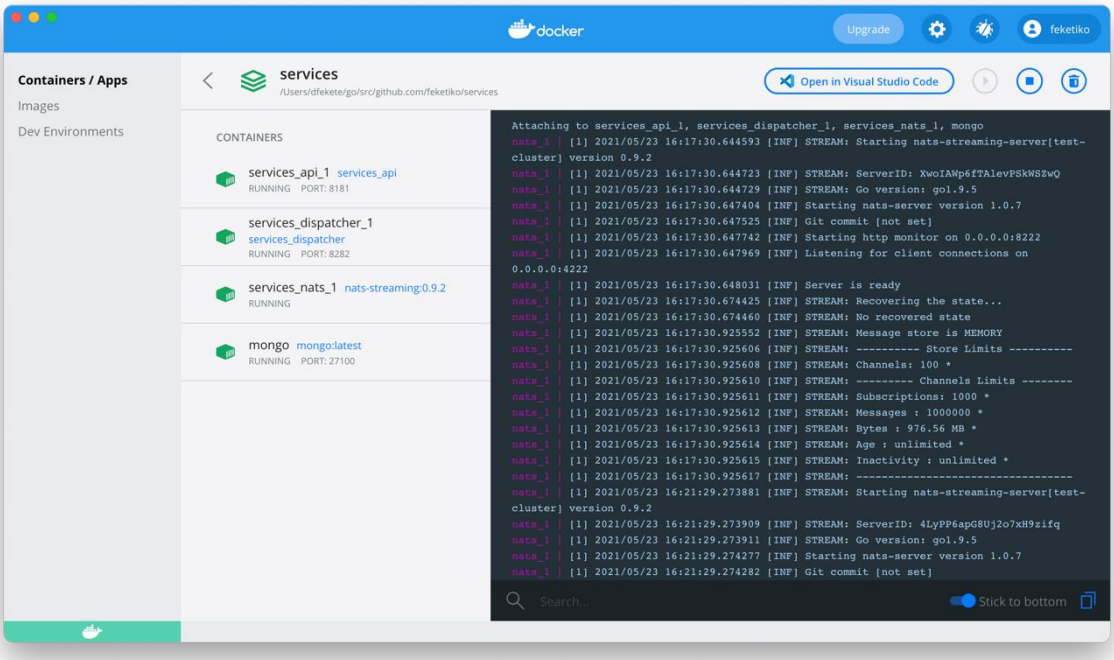


Рисунок 3.29 – використання GUI Docker-a для моніторингу системи

Для того, щоб протестувати систему, було використано Postman для створення REST-запитів до API-сервісу. На рисунку 3.30 зображена відправка параметричного POST-методу для створення нової Позиції. Як можна зазначити, при відправці сервіс повернув ID створеної Позиції.

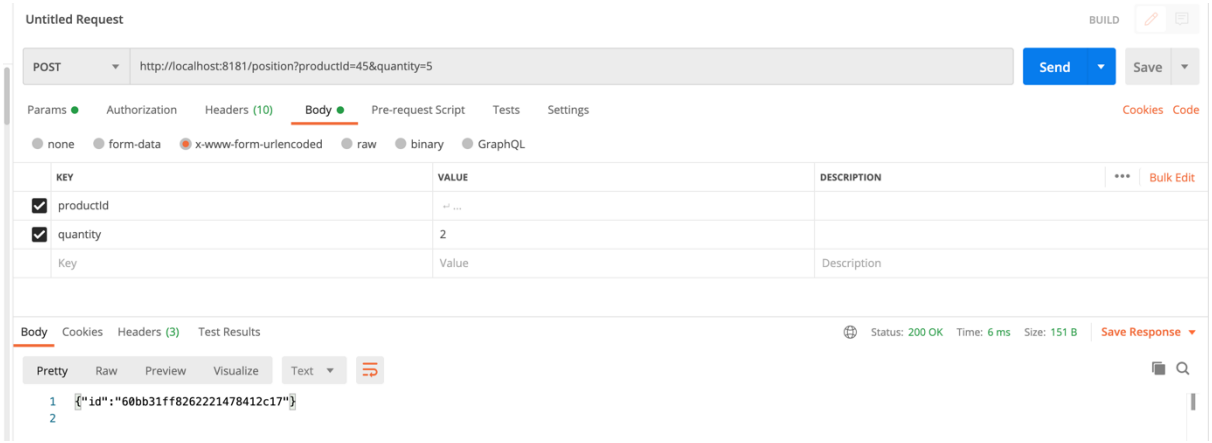


Рисунок 3.30 – POST-метод для створення нової Позиції

Після цього необхідно перевірити логи, щоб переконатись, що система відпрацювала як очікувалось. На рисунку 3.31 можна помітити таку послідовність виконаних дій:

– API прийняв повідомлення про Позицію та створив повідомлення для Dispatcher-сервісу;
– Dispatcher вичитав повідомлення та створив подію та агрегат.

```
api_1 | 2021/06/05 08:12:47 Published new position: {60bb31ff8262221478412c17
api_1 | 1 create 2 2021-06-05 08:12:47.900216924 +0000 UTC 0001-01-01 00:00:00 +0000 UTC}
dispatcher_1 | 2021/06/05 08:12:47 Readed new message in positions.updates:
{60bb31ff8262221478412c17
dispatcher_1 | 1 create 2 2021-06-05 08:12:47.900216924 +0000 UTC}
dispatcher_1 | 2021/06/05 08:12:47 [{_id ObjectID("60bb31ff8262221478412c17")} {userId }
{productId
dispatcher_1 | 1} {type create} {quantity 2} {createdAt 1622880767900}]
dispatcher_1 | 2021/06/05 08:12:47 [{_id ObjectID("60bb31ff8262221478412c17")} {productId
dispatcher_1 | 1} {available 2}]
```

Рисунок 3.31 – Логування процесу створення позиції

Наступним кроком необхідно переконатись, що Подія та Агрегат були збережені в сховище. На рисунку 3.32 зображений вміст сховища Подій та на рисунку 3.33 зображений вміст сховища Агрегатів.

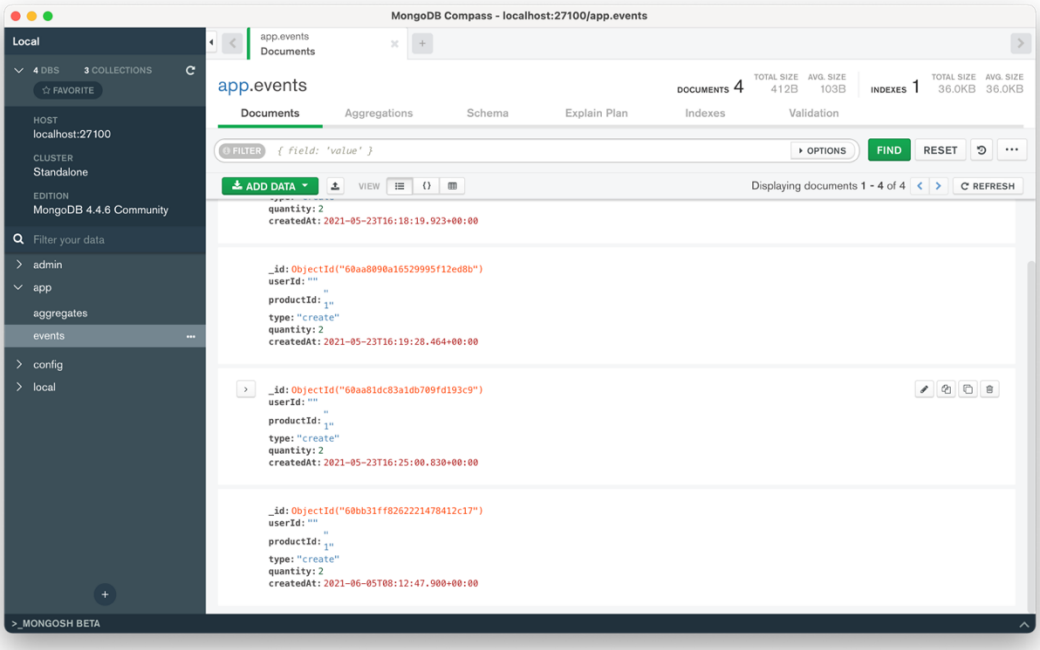


Рисунок 3.32 – Сховище Подій в MongoDB

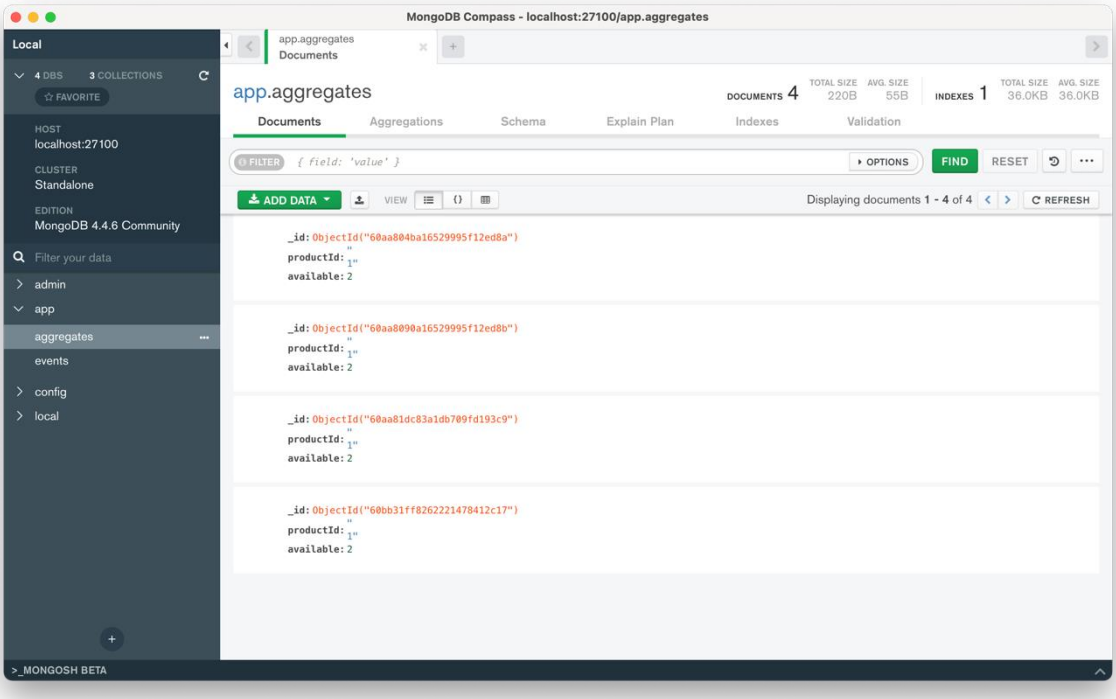


Рисунок 3.33 – Сховище Агрегатів в MongoDB

Отже, таким чином було протестовано працездатність системи і визначено – всі сервіси системи працюють справно.

Висновки по розділу ІІІ

У даному розділі було розглянуто послідовність проектування мікросервісної архітектури системи, керованої подіями.

Розглянуто підключення NATS та MongoDB до сервісів, використовуючи мову програмування Go та відповідні фреймворки. Розглянуто особливості та імплементацію основних функцій API та Dispatcher сервісів та їх взаємодію.

Було виконано запуск проекту на локальній машині з використанням Docker та протестовано роботу однієї з функцій, використовуючи Postman для створення параметричних REST-методів, та перевірено збереження подій у сховище, використовуючи GUI MongoDB Compass.

ВИСНОВОК

Мета даної роботи полягала в проектуванні архітектури інформаційної системи, керованої подіями, для забезпечення підтримки концепції «zero waste». Було спроектовано архітектуру цифрового продукту, який зможе забезпечити зменшення втрати їжі, придатної до споживання. Визначено головні проблеми та можливості їх усунення, пов’язані з проблемою «food waste». Було проведено аналіз ситуації на ринку схожих застосунків та наведено приклад вже існуючих систем. Було визначено основні функції та критерії, яким має відповідати поточна система.

У результаті виконання даної роботи були отримані:

– інфологічна модель предметної області;
– короткий опис можливих повідомлень, які передбачає система;
– схеми основного функціоналу;
– програмний додаток;
– програмний код мікросервісів;
– логічна та фізична моделі бази даних.

У ході виконання даної роботи було проведено детальний аналіз існуючих шаблонів проектування та технологій, що можуть використовуватись для розробки схожих систем. Також у даній роботі було розглянуто переваги систем, що керуються подіями, над CRUD-орієнтованими системами, проаналізовано переваги та недоліки обидвох підходів, визначено характеристики мікросервісної та монолітної архітектур та проаналізовано їх різницю.

Розроблено схеми для кожного компоненту системи, розглянуто атрибути компонентів, їх типи та властивості. Спроектовано схеми для основного функціоналу та проведено їх детальний аналіз, представлено їх опис з використанням графічних елементів.

Додаток, що був спроектований, легко масштабується, використовує сучасні технології та підходи до проектування програмного забезпечення. Було

детально описано роботу основних програмних компонентів та їх характеристики. Сервіси, що були спроектовані, готові до застосування на сучасному ринку застосунків, що допомагають підтримувати концепції «zero waste». Подальший розвиток роботи можливий у напрямку вдосконалення функціоналу існуючих сервісів та розробки нового функціоналу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Technical Platform on the Measurement and Reduction of Food Loss and Waste. [Електронне джерело]. Режим доступу: http://www.fao.org/platform-food-loss-waste/en/.
2. Food Wastage Footprint. [Електронне джерело]. Режим доступу:http://www.fao.org/nr/sustainability/food-loss-and-waste/en/
3. The Estimated Amount, Value, and Calories of Postharvest Food Losses at the Retail and Consumer Levels in the United States [Електронне джерело]. Режим доступу: https://www.ers.usda.gov/webdocs/publications/43833/43680_eib121.pdf?v=0
4. Đuric'. I. 2020. Digital technology and agricultural markets – Background paper for The State of Agricultural Commodity Markets (SOCO) 2020. Rome, FAO. [Електронне джерело]. Режим доступу. https://doi.org/10.4060/cb0701en
5. [Електронне джерело]. Режим доступу: http://www.fao.org/flw-in-fish-value-chains/value-chain/retail/restaurants-and-catering/en/.
6. Food Loss and Waste in Fish Value Chains. [Електронне джерело]. Режим доступу: https://www.worldvision.ca/stories/food/world-hunger-facts-how-to-help
7. Застосунок Olio. [Електронне джерело]. Режим доступу: https://en.wikipedia.org/wiki/Olio_(app)
8. Застосунок FullHarvest. [Електронне джерело]. Режим доступу: https://www.fullharvest.com
9. Food Waste App OLIO Has Become A Lifeline For Those Who Can't Afford To Feed Themselves. [Електронне джерело]. Режим доступу: https://www.huffingtonpost.co.uk/entry/food-waste-app-olio-hidden-hunger_uk_595f4212e4b0d5b458e97c36

10. Food for London: Olio, the app matching surplus food to hungry Londoners. [Електронне джерело]. Режим доступу: https://www.standard.co.uk/news/foodforlondon/food-for-london-the-app-matching-surplus-food-to-hungry-londoners-a3387641.html
11. Don’t Toss That Lettuce — Share It. [Електронне джерело]. Режим доступу: https://www.gsb.stanford.edu/insights/dont-toss-lettuce-share-it
12. Beauty (and taste!) are on the inside. FAO. [Електронне джерело]. Режим доступу:http://www.fao.org/fao-stories/article/en/c/1100391/
13. How Full Harvest is Using Technology to Connect the Dots in the B2B Food Waste Space. [Електронне джерело]. Режим доступу: https://agfundernews.com/how-full-harvest-is-using-technology-to-connect-the-dots-in-the-b2b-food-waste-space.html
14. Sommerville I. Software engineering. Tenth edition, global edition. Boston, Mass. Amsterdam Cape Town: Pearson Education Limited; 2016. 810 p. (Always learning).
15. Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, et al. Microservices: yesterday, today, and tomorrow. arXiv:160604036 [cs] [Електронне джерело]. 2016 Jun 13 [cited 2018 May 15]; Режим доступу: http://arxiv.org/abs/1606.04036
16. Thönes J. Microservices. IEEE Software. 2015 Jan;32(1):116–116.
17. Liu L, Özsu MT, editors. Encyclopedia of database systems. New York: Springer; 2009. 5 p. (Springer reference).
18. De La Torre C. Challenges and solutions for distributed data management [Електронне джерело]. [cited 2018 May 20]. Доступно з: https://docs.microsoft.com/en- us/dotnet/standard/microservices-architecture/architect-microservice-container- applications/distributed-data-management
19. Evans E. Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional; 2004. 563 p.

20. Young G. Command Query Separation? | Greg Young [Електронне джерело]. [cited 2018 May 20]. Режим доступу: http://codebetter.com/gregyoung/2009/08/13/command- query-separation/
21. Betts D, Domínguez J, Melnik G, Simonazzi F, Subramanian M. Exploring CQRS and Event Sourcing. 1st ed. Microsoft patterns & practices; 2013. 376 p.
22. GOTO Conferences. GOTO 2014 • Event Sourcing • Greg Young [Електронне джерело]. [cited 2018 Apr 25]. Режим доступу: https://www.youtube.com/watch?v=8JKjvY4etTY&t=
23. Overeem M, Spoor M, Jansen S. The dark side of event sourcing: Managing data conversion. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2017. p. 193–204.

ДОДАТОК А

Код програмних модулів та конфігураційних файлів

1. Код файлу Docker-compose.yaml

```
version: "3.6"

services:
  nats:
    image: "nats-streaming:0.9.2"
    restart: "always"
  mongo:
    image: 'mongo:latest'
    container_name: 'mongo'
    ports:
      - '27100:27017'
  api:
    build: "."
    command: "api-service"
    depends_on:
      - "nats"
      - "mongo"
    ports:
      - "8181:8181"
    environment:
      NATS_ADDRESS: "nats:4222"
      MONGODB_ADDRESS: "mongo:27017"
  dispatcher:
    build: "."
    command: "dispatcher-service"
    ports:
      - "8282:8282"
    depends_on:
      - "nats"
      - "mongo"
    environment:
      NATS_ADDRESS: "nats:4222"
      MONGODB_ADDRESS: "mongo:27017"
```

2. Код файлу Dockerfile

```
FROM golang:1.10.2-alpine3.7 AS build
RUN apk --no-cache add gcc g++ make ca-certificates
WORKDIR /go/src/github.com/feketiko/services

COPY vendor vendor
COPY event event
COPY schema schema
COPY mongo mongo
COPY api-service api-service
COPY dispatcher-service dispatcher-service

RUN go install ./...

FROM alpine:3.7
WORKDIR /usr/bin
COPY --from=build /go/bin .
```

3. Код файлу go.mod

```
module github.com/feketiko/services

go 1.14

require (
	github.com/golang/protobuf v1.5.2 // indirect
	github.com/gorilla/handlers v1.5.1 // indirect
	github.com/gorilla/mux v1.8.0
	github.com/kelseyhightower/envconfig v1.4.0
	github.com/nats-io/nats-server/v2 v2.2.4 // indirect
	github.com/nats-io/nats.go v1.11.0
	github.com/segmentio/ksuid v1.0.3
	github.com/tinrab/retry v1.0.0
	go.mongodb.org/mongo-driver v1.5.2
	gopkg.in/mgo.v2 v2.0.0-20190816093944-a6b53ec6cb22
)
```

4. Код файлу go.sum

```
github.com/BurntSushi/toml v0.3.1/go.mod h1:xHWCNGjB5oqiDr8zfno3MHue2Ht5sIBksp03qcyfWMU=
github.com/aws/aws-sdk-go v1.34.28 h1:sscPpn/Ns3i0F4HPEWAVcwdIRaZZCuL7llJ2/60yPIk=
github.com/aws/aws-sdk-go v1.34.28/go.mod h1:H7NKnBqNVzoTJpGfLrQkkD+ytBA93eiDYi/+8rV9s48=
github.com/davecgh/go-spew v1.1.0/go.mod h1:J7Y8YcW2NihsgmVo/mv3lAwl/skON4iLHjSsI+c5H38=
github.com/davecgh/go-spew v1.1.1/go.mod h1:J7Y8YcW2NihsgmVo/mv3lAwl/skON4iLHjSsI+c5H38=
github.com/felixge/httpsnoop v1.0.1 h1:lvB5Jl89CsZtGIWuTcDM1E/vkVs49/Ml7JJe07l8SPQ=
github.com/felixge/httpsnoop v1.0.1/go.mod h1:m8KPJKqk1gH5J9DgRY2ASl2lWCfGKXixSwevea8zH2U=
github.com/go-sql-driver/mysql v1.5.0/go.mod h1:DCzpHaOWr8IXmIStZouvnhqoel9Qv2LBy8hT2VhHyBg=
github.com/go-stack/stack v1.8.0 h1:5SgMzNM5HxrEjV0ww2lTmX6E2Izsfxas4+YHWRs3Lsk=
github.com/go-stack/stack v1.8.0/go.mod h1:v0f6uXyyMGvRgIKkXu+yp6POWl0qKG85gN/melR3HDY=
github.com/gobuffalo/attrs v0.0.0-20190224210810-a9411de4debd/go.mod h1:4duuawTqi2wkkpB4ePgWMaai6/Kc6WEz83bhFwpHzj0=
github.com/gobuffalo/depgen v0.0.0-20190329151759-d478694a28d3/go.mod h1:3STtPUQYuzV0gBVOY3vy6CfMm/ljR4pABfrTeHNLHUY=
github.com/gobuffalo/depgen v0.1.0/go.mod h1:+ifsuy7fhi15RWncXQQKjWS9JPkdah5sZvtHc2RXGlg=
github.com/gobuffalo/envy v1.6.15/go.mod h1:n7DRkBerg/aorDM8kbduw5dN3oXGswK5liaSCx4T5NI=
github.com/gobuffalo/envy v1.7.0/go.mod h1:n7DRkBerg/aorDM8kbduw5dN3oXGswK5liaSCx4T5NI=
github.com/gobuffalo/flect v0.1.0/go.mod h1:d2ehjJqGOH/Kjqcoz+F7jHTBbmDb38yXA598Hb50EGs=
github.com/gobuffalo/flect v0.1.1/go.mod h1:8JCgGVbRjJhVgD6399mQr4fx5rRfGKVzFjbj6RE/9UI=
github.com/gobuffalo/flect v0.1.3/go.mod h1:8JCgGVbRjJhVgD6399mQr4fx5rRfGKVzFjbj6RE/9UI=
github.com/gobuffalo/genny v0.0.0-20190329151137-27723ad26ef9/go.mod h1:rWs4Z12d1Zbf19rlsn0nurr75KqhYp52EAGGxTbBhNk=
github.com/gobuffalo/genny v0.0.0-20190403191548-3ca520ef0d9e/go.mod h1:80lIj3kVJWwOrXWWMRzzdhW3DsrdjILVil/SFKBzF28=
github.com/gobuffalo/genny v0.1.0/go.mod h1:XidbUqzak3lHdS//TPu2OgiFB+51Ur5f7CSnXZ/JDvo=
github.com/gobuffalo/genny v0.1.1/go.mod h1:5TExbEyY48pfunL4QSXxlDOmdsD44RRq4mVZ0Ex28Xk=
github.com/gobuffalo/gitgen v0.0.0-20190315122116-cc086187d211/go.mod h1:vEHJk/E9DmhejeLeNt7UVvlSGv3ziL+djtTr3yyzcOw=
github.com/gobuffalo/gogen v0.0.0-20190315121717-8f38393713f5/go.mod h1:V9QVDIxsgKNZs6L2IYiGR8datgMhB577vzTDqypH360=
github.com/gobuffalo/gogen v0.1.0/go.mod h1:8NTelM5qd8RZ15VjQTFkAW6qOMx5wBbW4dSCS3BY8gg=
github.com/gobuffalo/gogen v0.1.1/go.mod h1:y8iBtmHmGc4qa3urIyo1shvOD8JftTtfcKi+71xfDNE=
github.com/gobuffalo/logger v0.0.0-20190315122211-86e12af44bc2/go.mod h1:QdxcLw541hSGtBnhUc4gaNIXRjiDppFGaDqzbrBd3v8=
github.com/gobuffalo/mapi v1.0.1/go.mod h1:4VAGh89y6rVOvm5A8fKFxYG+wIW6LO1FMTG9hnKStFc=
github.com/gobuffalo/mapi v1.0.2/go.mod h1:4VAGh89y6rVOvm5A8fKFxYG+wIW6LO1FMTG9hnKStFc=
github.com/gobuffalo/packd v0.0.0-20190315124812-a385830c7fc0/go.mod h1:M2Juc+hhDXf/PnmBANFCqx4DM3wRbgDvnVWeG2RIxq4=
github.com/gobuffalo/packd v0.1.0/go.mod h1:M2Juc+hhDXf/PnmBANFCqx4DM3wRbgDvnVWeG2RIxq4=
github.com/gobuffalo/packr/v2 v2.0.9/go.mod h1:emmyGweYTm6Kdper+iywB6YK5YzuKchGtJQZ0Odn4pQ=
github.com/gobuffalo/packr/v2 v2.2.0/go.mod h1:CaAwI0GPIAv+5wKLtv8Afwl+Cm78K/I/VCm/3ptBN+0=
```

github.com/gobuffalo/syncx v0.0.0-20190224160051-33c29581e754/go.mod h1:HhnNqWY95UYwwW3uSASeV7vtgYkT2t16hJgV3AEPUpw=
github.com/golang/protobuf v1.4.0-rc.1/go.mod h1:ceaxUfeHdC40wWswd/P6IGgMaK3YpKi5j83Wpe3EHw8=
github.com/golang/protobuf v1.4.0-rc.1.0.20200221234624-67d41d38c208/go.mod h1:xKAWHe0F5eneWXFV3EuXVDTCmh+JuBKY0li0aMyXATA=
github.com/golang/protobuf v1.4.0-rc.2/go.mod h1:LlEzMj4AhA7rCAGe4KMBDvJI+AwstrUpVNzEA03Pprs=
github.com/golang/protobuf v1.4.0-rc.4.0.20200313231945-b860323f09d0/go.mod h1:WU3c8KckQ9AFe+yFwt9sWVRKCVIyN9cPHBJSNnbL67w=
github.com/golang/protobuf v1.4.0/go.mod h1:jodUvKwWbYaEsadDk5Fwe5c77LiNKVO9IDvqG2KuDX0=
github.com/golang/protobuf v1.4.2/go.mod h1:oDoupMAO8OvCJWAcko0GGGIgR6R6ocIYbsSw735rRwI=
github.com/golang/protobuf v1.5.0/go.mod h1:FsONVRAS9T7sI+LIUmWTfcYkHO4aIWwzhcaSAoJOfIk=
github.com/golang/protobuf v1.5.2 h1:ROPKBNFfQgOUMifHyP+KYbvpjbdoFNs+aK7DXlji0Tw=
github.com/golang/protobuf v1.5.2/go.mod h1:XVQd3VNwM+JqD3oG2Ue2ip4fOMUkwXdXDdiuN0vRsmY=
github.com/golang/snappy v0.0.1 h1:Qgr9rKW7uDUkrbSmQeiDsGa8SjGyCOGtuasMWwvp2P4=
github.com/golang/snappy v0.0.1/go.mod h1:/XxbfmMg8lxefKM7IXC3fBNl/7bRcc72aCRzEWrmP2Q=
github.com/google/go-cmp v0.3.0/go.mod h1:8QqcDgzrUqlUb/G2PQTWiueGozuR1884gddMywk6iLU=
github.com/google/go-cmp v0.3.1/go.mod h1:8QqcDgzrUqlUb/G2PQTWiueGozuR1884gddMywk6iLU=
github.com/google/go-cmp v0.4.0/go.mod h1:v8dTdLbMG2kIc/vJvl+f65V22dbkXbowE6jgT/gNBxE=
github.com/google/go-cmp v0.5.2/go.mod h1:v8dTdLbMG2kIc/vJvl+f65V22dbkXbowE6jgT/gNBxE=
github.com/google/go-cmp v0.5.5 h1:Khx7svrCpmxxtHBq5j2mp/xVjsi8hQMfNLvJFAlrGgU=
github.com/google/go-cmp v0.5.5/go.mod h1:v8dTdLbMG2kIc/vJvl+f65V22dbkXbowE6jgT/gNBxE=
github.com/gorilla/handlers v1.5.1 h1:9lRY6j8DEeeBT10CvO9hGW0gmky0BprnvDI5vfhUHH4=
github.com/gorilla/handlers v1.5.1/go.mod h1:t8XrUpc4KVXb7HGyJ4/cEnwQiaxrX/hz1Zv/4g96P1Q=
github.com/gorilla/mux v1.8.0 h1:i40aqfkR1h2SlN9hojwV5ZA91wcXFOvkdNIeFDP5koI=
github.com/gorilla/mux v1.8.0/go.mod h1:DVbg23sWSpFRCP0SfiEN6jmj59UnW/n46BH5rLB71So=
github.com/inconshreveable/mousetrap v1.0.0/go.mod h1:PxqpIevigyE2G7u3NXJIT2ANytuPF1OarO4DADm73n8=
github.com/jmespath/go-jmespath v0.4.0 h1:BEgLn5cpjn8UN1mAw4NjwDrS35OdebyEtFe+9YPoQUg=
github.com/jmespath/go-jmespath v0.4.0/go.mod h1:T8mJZnbsbmF+m6zOOFylbeCJqk5+pHWvzYPziyZiYoo=
github.com/jmespath/go-jmespath/internal/testify v1.5.1/go.mod h1:L3OGu8Wl2/fWfCI6z80xFu9LTZmf1ZRjMHUOPmWr69U=
github.com/joho/godotenv v1.3.0/go.mod h1:7hK45KPybAkOC6peb+G5yklZfMxEjkZhHbwpqxOKXbg=
github.com/karrick/godirwalk v1.8.0/go.mod h1:H5KPZjojv4lE+QYImBI8xVtrBRgYrIVsaRPx4tDPEn4=
github.com/karrick/godirwalk v1.10.3/go.mod h1:RoGL9dQei4vP9ilrpETWE8CLOZ1kiN0LhBygSwrAsHA=
github.com/kelseyhightower/envconfig v1.4.0 h1:Im6hONhd3pLkfDFsbRgu68RDNkGF1r3dvMUtDTo2cv8=
github.com/kelseyhightower/envconfig v1.4.0/go.mod h1:cccZRl6mQpaq41TPp5QxidR+Sa3axMbJDNb//FQX6Gg=
github.com/klauspost/compress v1.9.5/go.mod h1:RyIbtBH6LamlWaDj8nUwkbUhJ87Yi3uG0guNDohfE1A=
github.com/klauspost/compress v1.11.12 h1:famVnQVu7QwryBN4jNseQdUKES71ZAOnB6UQQJPZvqk=
github.com/klauspost/compress v1.11.12/go.mod h1:aoV0uJVorq1K+umq18yTdKaF57EivdYsUV+/s2qKfXs=
github.com/konsorten/go-windows-terminal-sequences v1.0.1/go.mod h1:T0+1ngSBFLxvqU3pZ+m/2kptfBszLMUkC4ZK/EgS/cQ=
github.com/konsorten/go-windows-terminal-sequences v1.0.2/go.mod h1:T0+1ngSBFLxvqU3pZ+m/2kptfBszLMUkC4ZK/EgS/cQ=
github.com/kr/pretty v0.1.0/go.mod h1:dAy3ld7l9f0ibDNOQOHHMYYIIbhfbHSm3C4ZsoJORNo=
github.com/kr/pty v1.1.1/go.mod h1:pFQYn66WHrOpPYNljwOMqo10TkYh1fy3cYio2l3bCsQ=
github.com/kr/text v0.1.0/go.mod h1:4Jbv+DJW3UT/LiOwJeYQe1efqtUx/iVham/4vfdArNI=
github.com/markbates/oncer v0.0.0-20181203154359-bf2de49a0be2/go.mod h1:Ld9puTsIW75CHf65OeIOkyKbteujpZVXDpWK6YGZbxE=
github.com/markbates/safe v1.0.1/go.mod h1:nAqgmRi7cY2nqMc92/bSEeQA+R4OheNU2T1kNSCBdG0=
github.com/minio/highwayhash v1.0.1 h1:dZ6IIu8Z14VlC0VpfKofAhCy74wu/Qb5gcn52yWoz/0=
github.com/minio/highwayhash v1.0.1/go.mod h1:BQskDq+xkJ12lmlUUi7U0M5Swg3EWR+dLTk+kldvVxY=
github.com/montanaflynn/stats v0.0.0-20171201202039-1bf9dbcd8cbe/go.mod h1:wL8QJuTMNUDYhXwkmfOly8iTdp5TEcJFWZD2D7SIkUc=
github.com/nats-io/jwt v1.2.2 h1:w3GMTO969dFg+UOKTmmyuu7IGdusK+7Ytlt//OYH/uU=
github.com/nats-io/jwt v1.2.2/go.mod h1:/xX356yQA6LuXI9xWW7mZNpxgF2mBmGecH+Fj34sP5Q=
github.com/nats-io/jwt/v2 v2.0.2 h1:ejVCLO8gu6/4bOKIHQpmB5UhhUJfAQw55yvLWpfmKjI=
github.com/nats-io/jwt/v2 v2.0.2/go.mod h1:VRP+deawSXyhNjXmxPCHskrR6Mq50BqpEI5SEcNiGlY=
github.com/nats-io/nats-server/v2 v2.2.4 h1:yonz5m/M4C4uqDbOCMuSintpvILI0UjdBZ7SuhKhCns=

github.com/nats-io/nats-server/v2 v2.2.4/go.mod h1:sEnFaxqe09cDmfMgACxZbziXnhQFhwk+aKkZjBBRYrI=
github.com/nats-io/nats.go v1.11.0 h1:L263PZkrmkRJRJT2YHU8GwWWvEvmr9/LUKuJTXsF32k=
github.com/nats-io/nats.go v1.11.0/go.mod h1:BPko4oXsySz4aSWeFgOHLZs3G4Jq4ZAyE6/zMCxRT6w=
github.com/nats-io/nkeys v0.2.0/go.mod h1:XdZpAbhgyyODYqjTawOnIOI7VlbKSarI9Gfy1tqEu/s=
github.com/nats-io/nkeys v0.3.0 h1:cgM5tL53EvYRU+2YLXIK0G2mJtK12Ft9oeooSZMA2G8=
github.com/nats-io/nkeys v0.3.0/go.mod h1:gvUNGjVcM2IPr5rCsRsC6Wb3Hr2CQAm08dsxtV6A5y4=
github.com/nats-io/nuid v1.0.1 h1:5iA8DT8V7q8WK2EScv2padNa/rTESc1KdnPw4TC2paw=
github.com/nats-io/nuid v1.0.1/go.mod h1:19wcPz3Ph3q0Jbyiqsd0kePYG7A95tJPxeL+1OSON2c=
github.com/pelletier/go-toml v1.7.0/go.mod h1:vwGMzjaWMwyfHwgIBhI2YUM4fB6nL6lVAvS1LBMMhTE=
github.com/pkg/errors v0.8.0/go.mod h1:bwawxfHBFNV+L2hUp1rHADufV3IMtnDRdf1r5NINEl0=
github.com/pkg/errors v0.8.1/go.mod h1:bwawxfHBFNV+L2hUp1rHADufV3IMtnDRdf1r5NINEl0=
github.com/pkg/errors v0.9.1 h1:FEBLx1zS214owpjy7qsBeixbURkuhQAwrK5UwLGTwt4=
github.com/pkg/errors v0.9.1/go.mod h1:bwawxfHBFNV+L2hUp1rHADufV3IMtnDRdf1r5NINEl0=
github.com/pmezard/go-difflib v1.0.0/go.mod h1:iKH77koFhYxTK1pcRnkKkqfTogsbg7gZNVY4sRDYZ/4=
github.com/rogpeppe/go-internal v1.1.0/go.mod h1:M8bDsm7K2OlrFYOpmOWEs/qY81heoFRclV5y23lUDJ4=
github.com/rogpeppe/go-internal v1.2.2/go.mod h1:M8bDsm7K2OlrFYOpmOWEs/qY81heoFRclV5y23lUDJ4=
github.com/rogpeppe/go-internal v1.3.0/go.mod h1:M8bDsm7K2OlrFYOpmOWEs/qY81heoFRclV5y23lUDJ4=
github.com/segmentio/ksuid v1.0.3 h1:FoResxvleQwYiPAVKe1tMUlEirodZqlqglIuFsdDntY=
github.com/segmentio/ksuid v1.0.3/go.mod h1:/XUiZBD3kVx5SmUOl55voK5yeAbBNNIed+2O73XgrPE=
github.com/sirupsen/logrus v1.4.0/go.mod h1:LxeOpSwHxABJmUn/MG1IvRgCAasNZTLOkJPxbbu5VWo=
github.com/sirupsen/logrus v1.4.1/go.mod h1:ni0Sbl8bgC9z8RoU9G6nDWqqs/fq4eDPysMBDgk/93Q=
github.com/sirupsen/logrus v1.4.2/go.mod h1:tLMulIdttU9McNUspp0xgXVQah82FyeX6MwdIuYE2rE=
github.com/spf13/cobra v0.0.3/go.mod h1:1l0Ry5zgKvJasoi3XT1TypsSe7PqH0Sj9dhYf7v3XqQ=
github.com/spf13/pflag v1.0.3/go.mod h1:DYY7MBk1bdzusC3SYhjObp+wFpr4gzcvqqNjLnInEg4=
github.com/stretchr/objx v0.1.0/go.mod h1:HFkY916IF+rwdDfMAkV7OtwuqBVzrE8GR6GFx+wExME=
github.com/stretchr/objx v0.1.1/go.mod h1:HFkY916IF+rwdDfMAkV7OtwuqBVzrE8GR6GFx+wExME=
github.com/stretchr/testify v1.2.2/go.mod h1:a8OnRcib4nhh0OaRAV+Yts87kKdq0PP7pXfy6kDkUVs=
github.com/stretchr/testify v1.3.0/go.mod h1:M5WIy9Dh21IEIfnGCwXGc5bZfKNJtfHm1UVUgZn+9EI=
github.com/stretchr/testify v1.6.1/go.mod h1:6Fq8oRcR53rry900zMqJjRRixrwX3KX962/h/Wwjteg=
github.com/tidwall/pretty v1.0.0/go.mod h1:XNkn88O1ChpSDQmQeStsy+sBenx6DDtFZJxhVysOjyk=
github.com/tinrab/retry v1.0.0 h1:u1x0cMZszwG44AaEeH8xx3Z1guNt8syzULeOsDhzg9s=
github.com/tinrab/retry v1.0.0/go.mod h1:PWRlqYOz5dCyuZbxKhtQ60GN6OwSLwMxnjMqof4LIso=
github.com/xdg-go/pbkdf2 v1.0.0 h1:Su7DPu48wXMwC3bs7MCNG+z4FhcyEuz5dlvchbq0B0c=
github.com/xdg-go/pbkdf2 v1.0.0/go.mod h1:jrpuAogTd400dnrH08LKmI/xc1MbPOebTwRqcT5RDeI=
github.com/xdg-go/scram v1.0.2 h1:akYIkZ28e6A96dkWNJQu3nmCzH3YfwMPQExUYDaRv7w=
github.com/xdg-go/scram v1.0.2/go.mod h1:1WAq6h33pAW+iRreB34OORO2Nf7qel3VV3fjBj+hCSs=
github.com/xdg-go/stringprep v1.0.2 h1:6iq84/ryjjeRmMJwxutI51F2GIPlP5BfTvXHeYjyhBc=
github.com/xdg-go/stringprep v1.0.2/go.mod h1:8F9zXuvzgwmyT5DUm4GUfZGDdT3W+LCvS6+da4O5kxM=
github.com/youmark/pkcs8 v0.0.0-20181117223130-1be2e3e5546d h1:splanxYIlg+5LfHAM6xpdFEAYOk8iySO56hMFq6uLyA=
github.com/youmark/pkcs8 v0.0.0-20181117223130-1be2e3e5546d/go.mod h1:rHwXgn7JulP+udvsHwJoVG1YGAP6VLg4y9I5dyZdqmA=
go.mongodb.org/mongo-driver v1.5.2 h1:AsxOLoJTgP6YNM0fXWw4OjdluYmWzQYp+lFJL7xu9fU=
go.mongodb.org/mongo-driver v1.5.2/go.mod h1:gRXCHX4Jo7J0IJ1oDQyUxF7jfy19UfxniMS4xxMmUqw=
golang.org/x/crypto v0.0.0-20180904163835-0709b304e793/go.mod h1:6SG95UA2DQfeDnfUPMdvaQW0Q7yPrPDi9nlGo2tz2b4=
golang.org/x/crypto v0.0.0-20190308221718-c2843e01d9a2/go.mod h1:djNgcEr1/C05ACkg1iLfiJU5Ep61QUkGW8qpdssI0+w=
golang.org/x/crypto v0.0.0-20190422162423-af44ce270edf/go.mod h1:WFFai1msRO1wXaEeE5yQxYXgSfI8pQAWXbQop6sCtWE=
golang.org/x/crypto v0.0.0-20200302210943-78000ba7a073/go.mod h1:LzIPMQfyMNhhGPhUkYOs5KpL4U8rLKemX1yGLhDgUto=
golang.org/x/crypto v0.0.0-20200323165209-0ec3e9974c59/go.mod h1:LzIPMQfyMNhhGPhUkYOs5KpL4U8rLKemX1yGLhDgUto=

```
        golang.org/x/crypto                                v0.0.0-20210314154223-e6e6c4f2bb5b
h1:wSOdpTq0/eI46Ez/LkDwIsAKA71YP2SRKBODiRWM0as=
        golang.org/x/crypto                                v0.0.0-20210314154223-e6e6c4f2bb5b/go.mod
h1:T9bdIzuCu7OtxOm1hfPfRQxPLYneinmdGuTeoZ9dtd4=
        golang.org/x/net                                   v0.0.0-20190311183353-d8887717615a/go.mod
h1:t9HGtf8HONx5eT2rtn7q6eTqICYqUVnKs3thJo3Qplg=
        golang.org/x/net                                   v0.0.0-20190404232315-eb5bcb51f2a3/go.mod
h1:t9HGtf8HONx5eT2rtn7q6eTqICYqUVnKs3thJo3Qplg=
        golang.org/x/net                                   v0.0.0-20200202094626-16171245cfb2/go.mod
h1:z5CRVTTTmAJ677TzLLGU+0bjPO0LkuOLi4/5GtJWs/s=
        golang.org/x/net                                   v0.0.0-20210226172049-e18ecbb05110
h1:qWPm9rbaAMKs8Bq/9LRpbMqxWRVUAQwMI9fVrssnTfw=
        golang.org/x/net                                   v0.0.0-20210226172049-e18ecbb05110/go.mod
h1:m0MpNAwzfU5UDzcl9v0D8zg8gWTRqZa9RBIspLL5mdg=
        golang.org/x/sync                                  v0.0.0-20190227155943-e225da77a7e6/go.mod
h1:RxMgew5VJxzue5/jJTE5uejpjVlOe/izrB70Jof72aM=
        golang.org/x/sync                                  v0.0.0-20190412183630-56d357773e84/go.mod
h1:RxMgew5VJxzue5/jJTE5uejpjVlOe/izrB70Jof72aM=
        golang.org/x/sync                                  v0.0.0-20190423024810-112230192c58/go.mod
h1:RxMgew5VJxzue5/jJTE5uejpjVlOe/izrB70Jof72aM=
        golang.org/x/sync                                  v0.0.0-20190911185100-cd5d95a43a6e
h1:vcxGaoTs7kV8m5Np9uUNQin4BrLOthgV7252N8V+FwY=
        golang.org/x/sync                                  v0.0.0-20190911185100-cd5d95a43a6e/go.mod
h1:RxMgew5VJxzue5/jJTE5uejpjVlOe/izrB70Jof72aM=
        golang.org/x/sys                                   v0.0.0-20180905080454-ebe1bf3edb33/go.mod
h1:STP8DvDyc/dI5b8T5hshtkjS+E42TnysNCUPdjciGhY=
        golang.org/x/sys                                   v0.0.0-20190130150945-aca44879d564/go.mod
h1:STP8DvDyc/dI5b8T5hshtkjS+E42TnysNCUPdjciGhY=
        golang.org/x/sys                                   v0.0.0-20190215142949-d0b11bdaac8a/go.mod
h1:STP8DvDyc/dI5b8T5hshtkjS+E42TnysNCUPdjciGhY=
        golang.org/x/sys                                   v0.0.0-20190403152447-81d4e9dc473e/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
        golang.org/x/sys                                   v0.0.0-20190412213103-97732733099d/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
        golang.org/x/sys                                   v0.0.0-20190419153524-e8e3143a4f4a/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
        golang.org/x/sys                                   v0.0.0-20190422165155-953cdadca894/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
        golang.org/x/sys                                   v0.0.0-20190531175056-4c3a928424d2/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
        golang.org/x/sys                                   v0.0.0-20201119102817-f84b799fce68
h1:nxC68pudNYkKU6jWhgrqdreuFiOQWj1Fs7T3VrH4Pjw=
        golang.org/x/sys                                   v0.0.0-20201119102817-f84b799fce68/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
        golang.org/x/term                                  v0.0.0-20201126162022-7de9c90e9dd1/go.mod
h1:bj7SfCRtBDWHUb9snDiAeCFNEtKQo2Wmx5Cou7ajbmo=
        golang.org/x/text v0.3.0/go.mod h1:NqM8EUOU14njkJ3fqMW+pc6Ldnwhi/IjpwHt7yyuwOQ=
        golang.org/x/text v0.3.3/go.mod h1:5Zoc/QRtKVWzQhOtBMvqHzDpF6irO9z98xDceosuGiQ=
        golang.org/x/text v0.3.5 h1:i6eZZ+zk0SOf0xgBpEpPD18qWcJda6q1sxt3S0kzyUQ=
        golang.org/x/text v0.3.5/go.mod h1:5Zoc/QRtKVWzQhOtBMvqHzDpF6irO9z98xDceosuGiQ=
        golang.org/x/time                                  v0.0.0-20200416051211-89c76fbcd5d1
h1:NusfzzA6yGQ+ua51ck7E3omNUX/JuqbFSaRGqU8CcLI=
        golang.org/x/time                                  v0.0.0-20200416051211-89c76fbcd5d1/go.mod
h1:tRJNPiyCQ0inRvYxbN9jk5I+vvW/OXSQhTDSoE431IQ=
        golang.org/x/tools                                 v0.0.0-20180917221912-90fa682c2a6e/go.mod
h1:n7NCudcB/nEzxVGmLbDWY5pfWTLqBcC2KZ6jyYvM4mQ=
        golang.org/x/tools                                 v0.0.0-20190329151228-23e29df326fe/go.mod
h1:LCzVGOaR6xXOjkQ3onu1FJEFr0SW1gC7cKk1uF8kGRs=
        golang.org/x/tools                                 v0.0.0-20190416151739-9c9e1878f421/go.mod
h1:LCzVGOaR6xXOjkQ3onu1FJEFr0SW1gC7cKk1uF8kGRs=
```

```
	golang.org/x/tools v0.0.0-20190420181800-aa740d480789/go.mod
h1:LCzVGOaR6xXOjkQ3onu1FJEFr0SW1gC7cKk1uF8kGRs=
	golang.org/x/tools v0.0.0-20190531172133-b3315ee88b7d/go.mod
h1:/rFqwRUd4F7ZHNgwSSTFct+R/Kf4OFW1sUzUTQQTgfc=
	golang.org/x/xerrors v0.0.0-20191204190536-9bdfabe68543
h1:E7g+9GITq07hpfrRu66IVDexMakfv52eLZ2CXBWiKr4=
	golang.org/x/xerrors v0.0.0-20191204190536-9bdfabe68543/go.mod
h1:I/5z698sn9Ka8TeJc9MKroUUfqBBauWjQqLJ2OPfmY0=
	google.golang.org/protobuf v0.0.0-20200109180630-ec00e32a8dfd/go.mod
h1:DFci5gLYBciE7Vtevhsrf46CRTquxDuWsQurQQe4oz8=
	google.golang.org/protobuf v0.0.0-20200221191635-4d8936d0db64/go.mod
h1:kwYJMbMJ01Woi6D6+Kah6886xMZcty6N08ah7+eCXa0=
	google.golang.org/protobuf v0.0.0-20200228230310-ab0ca4ff8a60/go.mod
h1:cfTl7dwQJ+fmap5saPgwCLgHXTUD7jkjRqWcaiX5VyM=
	google.golang.org/protobuf v1.20.1-0.20200309200217-e05f789c0967/go.mod
h1:A+miEFZTKqfCUM6K7xSMQL9OKL/b6hQv+e19PK+JZNE=
	google.golang.org/protobuf v1.21.0/go.mod h1:47Nbq4nVaFHyn7ilMalzfO3qCViNmqZ2kzikPIcrTAo=
	google.golang.org/protobuf v1.23.0/go.mod h1:EGpADcykh3NcUnDUJcl1+ZksZNG86OlYog2l/sGQquU=
	google.golang.org/protobuf v1.26.0-rc.1/go.mod
h1:jlhhOSvTdKEhbULTjvd4ARK9grFBp09yW+WbY/TyQbw=
	google.golang.org/protobuf v1.26.0 h1:bxAC2xTBsZGibn2RTntX0oH50xLsqy1OxA9tTL3p/lk=
	google.golang.org/protobuf v1.26.0/go.mod h1:9q0QmTI4eRPtz6boOQmLYwt+qCgq0jsYwAQnmE0givc=
	gopkg.in/check.v1 v0.0.0-20161208181325-20d25e280405/go.mod
h1:Co6ibVJAznAaIkqp8huTwlJQCZ016jof/cbN4VW5Yz0=
	gopkg.in/check.v1 v1.0.0-20180628173108-788fd7840127/go.mod
h1:Co6ibVJAznAaIkqp8huTwlJQCZ016jof/cbN4VW5Yz0=
	gopkg.in/errgo.v2 v2.1.0/go.mod h1:hNsd1EY+bozCKY1Ytp96fpM3vjJbqLJn88ws8XvfDNI=
	gopkg.in/mgo.v2 v2.0.0-20190816093944-a6b53ec6cb22
h1:VpOs+IwYnYBaFnrNAeB8UUWtL3vEUnzSCL1nVjPhqrw=
	gopkg.in/mgo.v2 v2.0.0-20190816093944-a6b53ec6cb22/go.mod
h1:yeKp02qBN3iKW1OzL3MGk2IdtZzaj7SFntXj72NppTA=
	gopkg.in/yaml.v2 v2.2.8/go.mod h1:hI93XBmqTisBFMUTm0b8Fm+jr3Dg1NNxqwp+5A1VGuI=
	gopkg.in/yaml.v3 v3.0.0-20200313102051-9f266ea9e77c/go.mod
h1:K4uyk7z7BCEPqu6E+C64Yfv1cQ7kz7rIZviUmN+EgEM=
```

5. Код схеми Позицій з файлу schema/model.go

```
package schema

import (
	"time"
)

type Type int

const (
	Create Type = iota
	Hold
	Confirm
	Closed
)

func (e Type) String() string {
	switch e {
	case Create:
		return "create"
	case Hold:
		return "hold"
	case Confirm:
		return "confirm"
	case Closed:
```

```
                return "closed"
        default:
                return ""
        }
}

type Position struct {
        ID       string    `json:"id"`
        UserID   string    `json:"userId"`
        ProductId string   `json:"productId"`
        Type     Type      `json:"type"`
        Quantity int       `json:"quantity"`
        CreatedAt time.Time `json:"createdAt"`
        ExpiresAt time.Time `json:"expiresAt"`
}
```

6. Код з пакету event (підключення Nats)
    a. Файл event.go

```
package event

import "github.com/feketiko/services/schema"

type Store interface {
    Close()
    PublishPositionUpdates(position schema.Position) error
    SubscribePositionUpdates() (<-chan PublishedPosition, error)
    OnPositionUpdates(f func(PublishedPosition)) error
}

var impl Store

func SetEventStore(es Store) {
    impl = es
}

func Close() {
    impl.Close()
}

func PublishPositionUpdates(position schema.Position) error {
    return impl.PublishPositionUpdates(position)
}

func SubscribePositionUpdates() (<-chan PublishedPosition, error) {
    return impl.SubscribePositionUpdates()
}

func OnPositionUpdates(f func(PublishedPosition)) error {
    return impl.OnPositionUpdates(f)
}
```

    b. Файл nats.go

```
package event

import (
    "bytes"
    "encoding/gob"
    "log"

    "github.com/feketiko/services/schema"
    "github.com/nats-io/nats.go"
```

```
)

type NatsEventStore struct {
	nc                        *nats.Conn
	publishedPositionSubscription *nats.Subscription
	publishedPositionChan        chan PublishedPosition
}

func NewNats(url string) (*NatsEventStore, error) {
	nc, err := nats.Connect(url)
	if err != nil {
		return nil, err
	}
	return &NatsEventStore{nc: nc}, nil
}

func (e *NatsEventStore) Close() {
	if e.nc != nil {
		e.nc.Close()
	}
	if e.publishedPositionSubscription != nil {
		e.publishedPositionSubscription.Unsubscribe()
	}
	close(e.publishedPositionChan)
}

func (e *NatsEventStore) PublishPositionUpdates(pos schema.Position) error {
	m := PublishedPosition{pos.ID, pos.ProductId, pos.Type.String(), pos.Quantity,
pos.CreatedAt}
	data, err := e.writeMessage(&m)
	if err != nil {
		return err
	}
	return e.nc.Publish(m.Key(), data)
}

func (mq *NatsEventStore) writeMessage(p Position) ([]byte, error) {
	b := bytes.Buffer{}
	err := gob.NewEncoder(&b).Encode(p)
	if err != nil {
		return nil, err
	}
	return b.Bytes(), nil
}

func (es *NatsEventStore) SubscribePositionUpdates() (<-chan PublishedPosition, error) {
	m := PublishedPosition{}
	es.publishedPositionChan = make(chan PublishedPosition, 64)
	ch := make(chan *nats.Msg, 64)
	var err error
	es.publishedPositionSubscription, err = es.nc.ChanSubscribe(m.Key(), ch)
	if err != nil {
		return nil, err
	}
	// Decode message
	go func() {
		for {
			select {
			case msg := <-ch:
				if err := es.readMessage(msg.Data, &m); err != nil {
```

```
                                        log.Fatal(err)
                                    }
                                    es.publishedPositionChan <- m
                                }
                            }
                        }()
                        return es.publishedPositionChan, nil
                    }

                    func (e *NatsEventStore) OnPositionUpdates(f func(PublishedPosition)) (err error) {
                        m := PublishedPosition{}
                        e.publishedPositionSubscription, err = e.nc.Subscribe(m.Key(), func(msg *nats.Msg) {
                            e.readMessage(msg.Data, &m)
                            f(m)
                        })
                        return
                    }

                    func (mq *NatsEventStore) readMessage(data []byte, m interface{}) error {
                        b := bytes.Buffer{}
                        b.Write(data)
                        return gob.NewDecoder(&b).Decode(m)
                    }
```

c. Файл position.go

```
package event

import (
    "time"
)

type Position interface {
    Key() string
}

type PublishedPosition struct {
    ID       string
    ProductId string
    Type     string
    Quantity  int
    CreatedAt time.Time
}

func (p *PublishedPosition) Key() string {
    return "positions.updates"
}
```

7. Код з пакету mongo
    a. Файл model.go

```
package model

import "go.mongodb.org/mongo-driver/bson/primitive"

type Event struct {
    ID       primitive.ObjectID `bson:"_id,omitempty"`
    UserID    string            `bson:"userId"`
    ProductId string            `bson:"productId,omitempty"`
    Type      string            `bson:"type,omitempty"`
    Quantity  int               `bson:"quantity,omitempty"`
    CreatedAt primitive.DateTime `bson:"createdAt,omitempty"`
}
```

```
type Aggregate struct {
    ID primitive.ObjectID `bson:"_id,omitempty"`
    ProductId string `bson:"productId,omitempty"`
    Available int    `bson:"available,omitempty"`
    OnHold    int    `bson:"onHold,omitempty"`
}
```

b. Файл storage.go

```
package mongo

import (
    "context"
    "log"

    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
    "go.mongodb.org/mongo-driver/mongo/readpref"
)

const (
    Events     = "events"
    Aggregates = "aggregates"
    Holds      = "holds"
)

type Storage struct {
    client *mongo.Client
    DB     *mongo.Database
}

func (s *Storage) Cancel() {
    err := s.client.Disconnect(context.Background())
    if err != nil {
        log.Fatal(err)
    }
}

func NewMongo(url, dbName string) (*Storage, error) {
    clientOptions := options.Client().ApplyURI(url)
    client, err := mongo.Connect(context.Background(), clientOptions)
    if err != nil {
        return nil, err
    }

    err = client.Ping(context.Background(), readpref.Primary())
    if err != nil {
        return nil, err
    }

    return &Storage{
        client: client,
        DB:     client.Database(dbName),
    }, nil
}
```

8. API сервіс

a. Util.go

```
package util

import (
    "encoding/json"
```

```
    "net/http"
)

func ResponseOk(w http.ResponseWriter, body interface{}) {
    w.WriteHeader(http.StatusOK)
    w.Header().Set("Content-Type", "application/json")

    json.NewEncoder(w).Encode(body)
}

func ResponseError(w http.ResponseWriter, code int, message string) {
    w.WriteHeader(code)
    w.Header().Set("Content-Type", "application/json")

    body := map[string]string{
        "error": message,
    }
    json.NewEncoder(w).Encode(body)
}
```

b. Handler.go

```
package main

import (
    "context"
    "html/template"
    "log"
    "net/http"
    "strconv"
    "time"

    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/bson/primitive"
    "go.mongodb.org/mongo-driver/mongo/options"

    "github.com/feketiko/services/api-service/util"
    "github.com/feketiko/services/event"
    "github.com/feketiko/services/mongo"
    "github.com/feketiko/services/mongo/model"
    "github.com/feketiko/services/schema"
)

// positions
func createPositionHandler(w http.ResponseWriter, r *http.Request) {
    type response struct {
        ID string `json:"id"`
    }

    // Read parameters
    productId := template.HTMLEscapeString(r.FormValue("productId"))
    if len(productId) < 1 || len(productId) > 140 {
        util.ResponseError(w, http.StatusBadRequest, "Invalid productId")
        return
    }

    quantity := template.HTMLEscapeString(r.FormValue("quantity"))
    quantityInt, err := strconv.Atoi(quantity)
    if err != nil {
        util.ResponseError(w, http.StatusBadRequest, "Invalid quantity")
        return
    }
```

```
	createdAt := time.Now().UTC()
	id := primitive.NewObjectID()

	position := schema.Position{
		ID:	id.Hex(),
		ProductId: productId,
		Quantity:  quantityInt,
		Type:	schema.Create,
		CreatedAt: createdAt,
	}

	// Publish event
	if err := event.PublishPositionUpdates(position); err != nil {
		log.Println(err)
	}
	log.Printf("Published new position: %v", position)

	// Return new position
	util.ResponseOk(w, response{ID: position.ID})
}

// products
func allAvailablePositions(storage *mongo.Storage) func(w http.ResponseWriter, r *http.Request)
{
	return func(w http.ResponseWriter, r *http.Request) {
		type response struct {
			PositionsAvailable []model.Aggregate `json:"positions"`
		}

		findOptions := options.Find()
		var results []model.Aggregate

		//Passing the bson.D{{}} as the filter matches  documents in the collection
		cur,	err	:=	storage.DB.Collection(mongo.Aggregates).Find(context.TODO(),
bson.D{{}}, findOptions)
		if err != nil {
			log.Fatal(err)
		}
		for cur.Next(context.TODO()) {
			var elem model.Aggregate
			err := cur.Decode(&elem)
			if err != nil {
				log.Fatal(err)
			}

			if elem.Available > 0 {
				results = append(results, elem)
			}
		}

		util.ResponseOk(w, response{results})
	}
}

// holds
func createHoldHandler(w http.ResponseWriter, r *http.Request) {
	type response struct {
		ID string `json:"id"`
	}
```

```
        // Read parameters
        positionId := template.HTMLEscapeString(r.FormValue("positionId"))
        if len(positionId) < 1 || len(positionId) > 140 {
                util.ResponseError(w, http.StatusBadRequest, "Invalid positionId")
                return
        }

        quantity := template.HTMLEscapeString(r.FormValue("quantity"))
        quantityInt, err := strconv.Atoi(quantity)
        if err != nil {
                util.ResponseError(w, http.StatusBadRequest, "Invalid quantity")
                return
        }

        position := schema.Position{
                ID:       positionId,
                Quantity:  quantityInt,
                Type:      schema.Hold,
        }

        // Publish event
        if err := event.PublishPositionUpdates(position); err != nil {
                log.Println(err)
        }
        log.Printf("Published new position: %v", position)

        // Return new position
        util.ResponseOk(w, response{ID: position.ID})
}
```

c. Main.go

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "time"

        "github.com/gorilla/mux"
        "github.com/kelseyhightower/envconfig"
        "github.com/tinrab/retry"

        "github.com/feketiko/services/event"
        "github.com/feketiko/services/mongo"
)

type Config struct {
        NatsAddress    string `envconfig:"NATS_ADDRESS"`
        MongoDbAddress string `envconfig:"MONGODB_ADDRESS"`
}

func newRouter(storage *mongo.Storage) (router *mux.Router) {
        router = mux.NewRouter()
        router.HandleFunc("/positions/all", allAvailablePositions(storage)).Methods("GET")

        router.HandleFunc("/position", createPositionHandler).
                Methods("POST").
                Queries("productId", "{productId}", "quantity", "{quantity}")
```

```
	//creates a hold for position - {id} is positionId
	router.HandleFunc("/positions/{id}/hold", createHoldHandler).
		Methods("POST").
		Queries( "quantity", "{quantity}")
	return
}

func main() {
	var cfg Config
	err := envconfig.Process("", &cfg)
	if err != nil {
		log.Fatal(err)
	}

	storage, err := mongo.NewMongo(fmt.Sprintf("mongodb://%s", cfg.MongoDbAddress),
"app")
	if err != nil {
		log.Fatal(err)
	}
	defer storage.Cancel()

	// connects to NATS
	retry.ForeverSleep(2*time.Second, func(_ int) error {
		es, err := event.NewNats(fmt.Sprintf("nats://%s", cfg.NatsAddress))
		if err != nil {
			log.Println(err)
			return err
		}
		event.SetEventStore(es)
		return nil
	})
	defer event.Close()

	router := newRouter(storage)
	if err := http.ListenAndServe(":8181", router); err != nil {
		log.Fatal(err)
	}
}
```

9. Dispatcher
   a. Handler.go

```
package main

import (
	"context"
	"log"

	"go.mongodb.org/mongo-driver/bson"
	"go.mongodb.org/mongo-driver/bson/primitive"

	"github.com/feketiko/services/event"
	"github.com/feketiko/services/mongo"
	"github.com/feketiko/services/mongo/model"
	"github.com/feketiko/services/schema"
)

func onPositionUpdates(storage *mongo.Storage) func(position event.PublishedPosition) {
	return func(p event.PublishedPosition) {
		id, err := primitive.ObjectIDFromHex(p.ID)
		if err != nil {
```

```
                log.Println(err)
        }
        createdAt := primitive.NewDateTimeFromTime(p.CreatedAt)

        log.Printf("Readed new message in positions.updates: %v", p)

        evnt := model.Event{
                ID:       id,
                ProductId: p.ProductId,
                Type:     p.Type,
                Quantity:  p.Quantity,
                CreatedAt: createdAt,
        }

        // write new event in event store
        _, err = storage.DB.Collection(mongo.Events).InsertOne(context.Background(), evnt)
        if err != nil {
                log.Println(err)
        }

        // for new created position write an aggregate -
        if evnt.Type == schema.Create.String() {
                aggregate := model.Aggregate{
                        ID:       id,
                        ProductId: p.ProductId,
                        Available: p.Quantity,
                        OnHold:    0, // when position is created - 0 quantity is on hold
                }

                _, err = storage.DB.Collection(mongo.Aggregates).InsertOne(context.Background(), aggregate)
                if err != nil {
                        log.Println(err)
                }
        }

        // CHECKERS!
        var data interface{}
        var data1 interface{}
        err = storage.DB.Collection(mongo.Events).FindOne(context.Background(), bson.M{"_id": id}).Decode(&data)
        if err != nil {
                log.Println(err)
        }

        err = storage.DB.Collection(mongo.Aggregates).FindOne(context.Background(), bson.M{"_id": id}).Decode(&data1)
        if err != nil {
                log.Println(err)
        }
        log.Println(data)
        log.Println(data1)
    }
}
```

b. Main.go

```
package main

import (
    "fmt"
    "log"
```

```
	"net/http"
	"time"

	"github.com/kelseyhightower/envconfig"
	"github.com/tinrab/retry"

	"github.com/feketiko/services/event"
	"github.com/feketiko/services/mongo"
)

type Config struct {
	NatsAddress    string `envconfig:"NATS_ADDRESS"`
	MongoDbAddress string `envconfig:"MONGODB_ADDRESS"`
}

func main() {
	var cfg Config
	err := envconfig.Process("", &cfg)
	if err != nil {
		log.Fatal(err)
	}

	storage, err := mongo.NewMongo(fmt.Sprintf("mongodb://%s", cfg.MongoDbAddress),
"test")
	if err != nil {
		log.Fatal(err)
	}
	defer storage.Cancel()

	// connects to NATS
	retry.ForeverSleep(2*time.Second, func(_ int) error {
		es, err := event.NewNats(fmt.Sprintf("nats://%s", cfg.NatsAddress))
		if err != nil {
			log.Println(err)
			return err
		}

		err = es.OnPositionUpdates(onPositionUpdates(storage))
		if err != nil {
			log.Println(err)
			return err
		}

		event.SetEventStore(es)
		return nil
	})
	defer event.Close()

	if err := http.ListenAndServe(":8282", nil); err != nil {
		log.Fatal(err)
	}
}
```