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

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

До захисту допущено:

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Програмне забезпечення інформаційно-комунікаційних систем»
спеціальності 121 «Інженерія програмного забезпечення»
на тему: «Розподілена стримінгова система подій за шаблоном публікація-підписка»

Виконав (-ла):
студент (-ка) IV курсу, групи ІТ-61
Конорін Богдан Вікторович _____

Керівник:
асистент кафедри АУТС
Дорога-Іванюк Олена Олександрівна _____

Рецензент:
доцент кафедри ПЗКС ФПМ, к.т.н
Цуркан Василь Васильович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – перший (бакалаврський)

Спеціальність - 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Програмне забезпечення інформаційно-комунікаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту

Коноріну Богдану Вікторовичу

1. Тема проєкту «Розподілена стримінгова система подій за шаблоном публікація-підписка», керівник проєкту Дорога-Іванюк Олена Олександрівна, асистент кафедри АУТС, затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом проєкту: 9 червня 2020 р.

3. Вихідні дані до проєкту: мова програмування C#, середовище програмування Visual Studio, фреймворк .NET Core 3.1, Docker-контейнеризація, брокер повідомлень Apache Kafka, Elastic стек технологій

4. Зміст пояснювальної записки: постановка задачі, вибір та обґрунтування компонентів системи, опис програмного забезпечення, структура даних і ресурсів програми, тестування програмного забезпечення

5. Перелік графічного матеріалу: діаграма прецедентів, діаграма розгортання, структура проєкту, діаграма компонентів

6. Дата видачі завдання: 3 березня 2020 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Аналіз предметної області	10.04.2020	
2.	Опис функціональної схеми	13.04.2020	
3.	Проєктування системи	20.04.2020	
4.	Розробка макету проєктованої системи	24.04.2020	
5.	Розроблення функціональної схеми системи	26.04.2020	
6.	Реалізація програмної частини системи та подальше тестування	29.04.2020	
7.	Тестування програми	14.05.2020	
8.	Виконання графічних документів	17.05.2020	
9.	Оформлення пояснювальної записки	18.05.2020	
10.	Подання проєкту на основний захист	15.06.2020	

Студент Богдан КОНОРІН

Керівник Олена ДОРОГА-ІВАНЮК

АНОТАЦІЯ

Конорін Б.В. Розподілена стримінгова система подій за шаблоном публікація-підписка. КПІ ім. Ігоря Сікорського, Київ, 2020.

Проєкт містить 5 розділів, 70 сторінок, 30 рисунків, посилання на 17 літературних джерел і 4 кресленики.

Ключові слова: стримінг, події, слухач, потік даних, продюсер, консюмер, джерело даних, мікросервіси, .NET, Apache Kafka.

Об’єктом розробки є стримінгова система подій різного типу за форматом публікація-підписка.

Мета роботи – розробити платформу, котра буде мати широкий набір програмних інтерфейсів для можливості передачі даних від джерел до клієнтів з забезпеченням інфраструктурних рішень для концентрації на правилах бізнесу.

Під час виконання дипломного проєкту було розроблено систему для стримінгу даних в реальному часі з можливістю візуалізації зібраних даних. Було проведено детальний аналіз існуючих підходів до розробки та моделювання, що виконували б поставлені функціональні та нефункціональні вимоги до системи. Платформа розроблена мовою програмування C# на базі середовища виконання .NET Core. Компоненти системи розгорнуті у Docker-контейнерах для подальшої міграції в хмарне середовище.

Отримана платформа є основою для систем типу стримінгу даних з забезпеченням усіх необхідних складових на базовому рівні з можливістю розширення в майбутньому. Дана робота зможе зацікавити компанії, котрі зацікавлені в передачі даних між своїми додатками або у інтеграціях з іншими бізнес рішеннями.

SUMMARY

Konorin B.V. Distributed publish-subscribe event streaming system. Igor Sikorsky KPI, Kyiv, 2020.

The project contains 5 sections, 70 pages, 30 figures, links to 17 literary sources and 4 design documents.

Keywords: streaming, events, data stream, producer, consumer, data source, microservices, .NET, Apache Kafka.

The object of the development is the streaming system of events of different types in the format “publication-subscription”.

The aim of the development is to develop a platform, which will have a wide range of software interfaces to transfer data from sources to clients, providing infrastructure solutions to focus on the business rules.

During the diploma project, a system for real-time data streaming with the ability of collected data visualization was developed. The detailed analysis of existing approaches for development and modelling, which would meet the functional and non-functional system requirements, was conducted. The platform is developed with the help of C# programming language based on the .NET Core runtime. The components of the system are deployed in Docker containers for further migration to the cloud environment.

The gained platform is the basis for the systems of data streaming types, providing all necessary components at the basic level with the possibility of further expansion. This work can be of interest to the companies, which are interested in transferring data between their applications or in integrations with other business solutions.

Поз.	Формат	Позначення	Найменування	Кількість аркушів	№ прим.	Примітки
1			Документація загальна			
2						
3			Знову розроблена			
4						
5	А4	ІТ61.090БАК.004 ПЗ	Розподілена стримінгова система	70		
6			подій за шаблоном публікація-			
7			підписка. Пояснювальна записка			
8	А3	ІТ61.090БАК.004 Д1	Розподілена стримінгова система	1		
9			подій за шаблоном публікація-			
10			підписка. Діаграма прецедентів			
11	А3	ІТ61.090БАК.004 Д2	Розподілена стримінгова система	1		
12			подій за шаблоном публікація-			
13			підписка. Діаграма розгортання			
14	А3	ІТ61.090БАК.004 Д3	Розподілена стримінгова система	1		
15			подій за шаблоном публікація-			
16			підписка. Структура проєкту			
17	А3	ІТ61.090БАК.004 Д4	Розподілена стримінгова система	1		
18			подій за шаблоном публікація-			
19			підписка. Діаграма компонентів			
20						
21						
22						
23						
24						

					ІТ61.090БАК.004 ТП		
Змн.	Лист	№ докум.	Підпис	Дата			
Розроб.		Конорін Б.В.			Розподілена стримінгова система подій за шаблоном публікація-підписка. Відомість технічного проєкту	Літ.	Лист 1 / Листів 1
Перевір.		Дорога-Іванюк О.О.					
Н. Контр.						КПІ ім. Ігоря Сікорського кафедра АУТС гр. ІТ-61	
Затверд.		Ролік О.І.					

Пояснювальна записка
до дипломного проєкту
на тему: «Розподілена стримінгова система подій за шаблоном публікація-підписка»

Київ – 2020 року

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Брокер повідомлень – проміжний комп'ютерний програмний модуль, який переводить повідомлення з формального протоколу обміну повідомленнями відправника в формальний протокол обміну повідомленнями одержувача;

Дуплексний зв’язок – можливість підтримувати зв’язок між двома учасниками для обміну між ними повідомленнями у обох напрямках;

Консюмер (англ. Consumer) – об’єкт, котрий приймає повідомлення;

Контейнеризація (англ. Containerization) – спосіб віртуалізації операційної системи, в якій запускаються додатки для їх ізоляції від зовнішнього впливу;

Кросплатформність (англ. Cross-platform) – здатність програмного забезпечення працювати з двома і більше операційними системами;

Логи (англ. Log) – записи про події або зміни стану системи у системних журналах;

Меппінг даних (англ. Data Mapping) – це процес створення відображень елементів даних між двома різними моделями даних;

Партиція (англ. Partition) – частина у топіку, яка виділена за певним критерієм;

Продюсер (англ. Producer) – об’єкт, який створює повідомлення;

Сокет – програмний інтерфейс для впровадження інформаційного обміну між процесами;

Стримінг (англ. Streaming) – потокова передача будь-якого об’єму даних малими порціями;

Топік (англ. Topic) – черга повідомлень брокеру повідомлень;

Хостинг (англ. Hosting) – можливість зробити веб-додатки людей та організацій доступними через мережу Інтернет;

API (англ. Application Programming Interface) – програмний інтерфейс, який представляє набір функціональності доступний до використання від сервісу;

DevOps – методологія активної взаємодії фахівців з розробки з фахівцями з інформаційно-технологічного обслуговування і взаємна інтеграція їх робочих процесів один в одного для забезпечення якості продукту;

ETL (Extract, Transform, Load) – загальна процедура копіювання даних з одного або декількох джерел у систему призначення, яка представляє дані інакше, ніж джерела, або в іншому контексті, ніж джерела;

HTTP (англ. HyperText Transfer Protocol) – протокол передачі даних через мережу Інтернет;

Internet of Things (IoT) – система взаємопов’язаних обчислювальних пристроїв, механічних і цифрових машин, предметів, тварин або людей, яким надаються унікальні ідентифікатори та можливість передачі даних по мережі без необхідності взаємодії у форматі людина-людина чи людина-комп'ютер;

REST (англ. Representational State Transfer) – архітектурний стиль взаємодії компонентів розподілених додатків у мережі;

SQL (англ. Structured query language) – декларативна мова програмування для роботи з реляційними базами даних.

ВСТУП

Щодня у світі відбуваються сотні подій у різних сферах бізнесу та життєдіяльності суспільства, за котрими слідкують сотні тисяч глядачів по всьому світу та беруть у цьому активну участь. Глядачами можуть бути як люди, так і цифрові пристрої. Події можуть бути різних масштабів та стосуватися широкого спектру джерел інформації від трансляції відео-контенту через цифрові кабелі до інформації зі спеціальних пристроїв, що транслюється через мережу Інтернет за допомогою різноманітних протоколів.

Реальність сьогодення показує, що джерел інформації та подій, котрі оточують нас, велика кількість - набагато більше, ніж нам потрібно, та яку ми маємо можливість обробити. З кожним днем все більше компаній та інженерів шукають способи передачі, обробки та фільтрації даних для використання у своїх цілях. Колосальну кількість інформації виробляють пристрої «Інтернету речей», які на сьогоднішній день є одним з основних сфер для аналізу науки даних. Також, більшість компаній зацікавлена в телеметрії та обробці даних для подальшого впровадження рекламних компаній та для кращого розуміння способів задоволення потреб клієнтів.

Розробників таких систем у світі не багато, і вони є монополістами на цьому ринку, проте бізнес-користувачів цих систем дуже багато. Подібний клас систем відсутній на ринку України, оскільки його розробка є трудомістким процесом і вимагає великої кількості інвестицій.

Предметна область розробки таких систем досліджена поверхнево, оскільки вона тільки набирає популярність у світі та кожна сфера трансляції подій має власні бізнес-правила, закони та інженерні рішення, які більшість компаній захищає, як комерційну таємницю, оскільки вони дають значну перевагу над конкурентами. Також епідемії хвороби у світі активізують трансформацію контенту у цифрову сферу, у тому числі і у стримінгових платформ.

Завдання дипломного проєкту полягає в дослідженні специфіки систем передачі даних від джерел до клієнтів, а також у побудові такої системи, яка стане

основою для подальшого збільшення бізнес-правил та підтримки передачі подій з різних сфер життя. У ході роботи події будуть розглядатись у контексті спортивних подій реального часу та контексті зміни стану предметів або явищ таких, як температура або файл на диску.

Для того, щоб глядачі мали можливість отримати актуальну інформацію про всі деталі ситуації того чи іншого явища, бізнес повинен отримувати та обробляти інформацію з низькою затримкою, високою надійністю та доступністю, так як від цього можуть залежати клієнти цього бізнесу. Для забезпечення цих правил потрібно використовувати такі підходи, як шардинг або реплікація сервісів обробки та трансляції. Також критичним може стати доступ до спільних ресурсів, тому обробка повинна бути налаштована на неблокуючі операції з ними та використовувати кеш для швидкого доступу до них. Такі системи чудово пристосовані для хмарних обчислень та розгортання в такому середовищі для можливості контролювання навантаження.

Серцем подібних систем є хаб обміну повідомленнями, який повинен витримувати велике навантаження, мати високу стійкість до відмов, виконувати агрегацію та проксіювання подій, забезпечувати маршрутизацію та фільтрацію подій по різним каналам. Створення подібного програмного рішення дозволило би значно полегшити розробку подібних систем та збільшити кількість вітчизняних рішень у сфері трансляцій у реальному часі.

Клієнтами системи можуть бути як веб-додатки, так і інші сервіси зацікавлених бізнес-партнерів. Для цього потрібно додати можливість підтримки декількох типів підключення через різні варіанти каналів, щоб клієнти могли вибирати, залежно від того, що має більший пріоритет для бізнесу: швидкість, надійність, можливість мати способи для комунікації з цільовою системою в дуплексному форматі або в одноканальному режимі.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Функціональні вимоги до продукту

Функціональні вимоги для програмного забезпечення складають собою основу, котра повинна бути задовільнена під час розробки платформи, тому займає найбільшу частину часу розробника. Функціонал системи повинен мати найкращі характеристики існуючих рішень та пропонувати покращення чи аналоги функціоналу з інших типів систем, тим самим бути агрегатом найкращих рис існуючих комерційних рішень у суміжних сферах.

Стримінгова платформа подій являє собою систему по трансляції даних від джерела до сервісів-слухачів. З визначення виходить, що система повинна складатися з таких компонентів: сервіси, які створюють різноманітні події, головний хаб, який відповідає за розсилку подій до систем-підписників, клієнти-сервіси системи, котрі слухають події та описують правила, за якими відбувається фільтрація, та веб-сервіс, котрий відповідає за авторизацію у системі та можливість підписуватись на події або переглядати інформацію про можливі трансляції.

Всі частини системи повинні мати можливість розгортання на UNIX системах, Windows та підтримувати можливість до контейнеризації для подальшого розгортання у хмарних провайдерів.

Основний функціонал сервісів-джерел полягає у тому, щоб генерувати дані та відправляти їх у “сирому” вигляді до головного хабу, котрий відповідає, в свою чергу, за обробку цих даних та подальшої трансляції до клієнтів. У ході роботи потрібно покрити генерацію подій футбольних матчів та емуляцію даних з датчиків, наприклад таких, як температурний датчик або датчик вологи.

Дані про футбольні матчі потрібно знайти у відкритому доступі, проте деталі їх перебігу, а саме таких подій, як гол, пенальті та інше, можуть бути приватними, тому слід реалізувати функціонал, який буде імітувати події, які відбуваються під час матчу для забезпечення більшої варіативності даних для подальшої обробки та фільтрації. Екземпляри сервісів-джерел можуть обробляти

декілька спортивних заходів за раз або бути поділені на категорії залежно від ресурсу, з якого беруться відповідні дані.

Сервіси, які будуть імітувати роботу датчиків, можуть брати свої дані з відкритих джерел або генерувати їх відповідно до поставлених правил. Ці дані повинні зазначити стан вимірюваних величин протягом дня в певному інтервалі, а також покривати декілька міст України.

Екземплярів сервісів-джерел може бути будь-яка кількість, яка задовольняє ресурси платформи, і головна функція кожного з них - це створення даних без додаткової обробки та пересилання їх через брокер повідомлень у будь-якій кількості, але не більше, ніж може витримати машина, на якій буде налаштований та розгорнений цей брокер.

Головною частиною системи виступає сервіс-хаб, який відповідає за впровадження бізнес-правил до даних, які транслюються, і подальшої передачі їх до сервісів клієнтів. Хаб працює за шаблоном публікація-підписка, що в свою чергу означає, що сервіс-слухач повинен відправити HTTP запит на сервіс підписок, щоб підписатись на потрібну подію. Перед тим, як підписатись до відповідної події, сервіс-слухач повинен отримати ключ авторизації від сервісу підписок. Завдяки отриманому ключу, клієнт може дізнатись про заплановані події та підписатись на певні з них. Сервіс підписок повертає результат на запит у вигляді таблиці подій з полями, по яким може відбуватися фільтрація, тому що слухачі можуть бути не зацікавлені у всіх даних, що транслюються від джерел. Після підписки сервіс підписок повинен відправити нотифікацію до хабу про підписника та його бізнес-правила, адресу і тип зв'язку. Хаб повинен створювати можливість логування транзакцій у середині системи та надавати можливість для подальшої візуалізації змін системи. Для покращення кібер-захисту додатку можна створити “білий список” доменів, які можуть бути клієнтами платформи.

Клієнтська частина повинна представляти собою публічний програмний інтерфейс, який потрібно буде в подальшому підключити до сервісів, які будуть виступати слухачами. Клієнтська програмна бібліотека системи складає собою набір контрактів, які використовуються хабом і клієнтами для обміну даними між

ними за відповідними протоколами. Бібліотека повинна реалізовувати декілька найпопулярніших підходів передачі даних:

— формат пуш-повідомлень по HTTP на веб-сервер слухача (не вимагає додаткових дій зі сторони клієнта, окрім підписки з таким типом протоколу комунікації);

— формат консюмера логів стримінгової платформи або брокера повідомлень;

— формат обміну повідомлень за допомогою дуплексного каналу зв'язку на основі сокетів через UDP протокол.

Кожний з підходів відрізняється за своєю можливістю надавати гарантію про те, що повідомлення буде доставлено клієнту. З цього виходить, що хаб може нотифікувати клієнтів у форматі «run and forget», що означає: якщо повідомлення не буде доставлено, то повторного відправлення даних не відбудеться.

1.2 Нефункціональні вимоги до програмного продукту

Вимоги до системи, які не стосуються її функціоналу, проте є вимогами до інфраструктури програмного забезпечення, називають нефункціональними вимогами. Такі вимоги описують характеристики, обмеження та специфіку використання ресурсів для можливості її правильного функціонування. Для даної категорії систем виділяють такі нефункціональні вимоги:

— масштабованість - платформа поділяється на групи сервісів за призначенням, тому важливо додавати нові екземпляри додатків до груп, без переривань роботи існуючих, для підвищення пропускної можливості системи, збільшення кількості даних для трансляції або пришвидшення обробки цих даних;

— кросплатформність - сервіси повинні масштабуватись у хмарному середовищі, на різних віртуальних машинах на основі різних операційних систем, тому система повинна працювати як на Windows, так і UNIX системах;

— стійкість до відмов - додаток повинен обробляти велику кількість калькуляцій та обробку даних під навантаженням, що потенційно може викликати

помилки в процесі. Система повинна працювати, якщо процес обробки неможливий та генерувати логи про некоректну роботу відповідних транзакцій;

— стійкість до навантажень - платформа буде транслювати дані в реальному часі, що створить навантаження на систему. Для можливості витримати велике навантаження потрібно використати технології та структури даних, щоб зменшити кількість операцій, пов'язаних з процесами ввід/вивід, та мати можливість масштабувати систему для розподілення центрів обробки даних.

1.3 Огляд та аналіз аналогів продукту розробки

Подібний клас систем є специфічним для кожної предметної області трансляції подій та даних, тому у відкритому доступі не велика кількість інформації про такі бізнес рішення. Отже, найближчими до розглядання можна вважати системи, які являють собою хаби або брокери повідомлень, що транслюють дані.

Дані потокової передачі відносяться до даних, які генеруються безперервно, зазвичай у великих обсягах і з високою швидкістю. Джерело потоку даних зазвичай складається з потоку логів, які записують події у міру їх виникнення - наприклад, користувач, який натискає посилання на веб-сторінці, або датчик, що повідомляє про поточну температуру.

Загальні приклади потокових даних включають:

— датчики IoT;

— логи сервера та логи центрів безпеки;

— рекламу в режимі реального часу;

— дані, що передаються в клік, із додатків та веб-сайтів.

Такий клас систем поділяють на декілька типів і найчастіше використовують разом, як конвеєр, через який проходять дані від джерела до клієнта, тим самим забезпечуючи фільтрацію і аналіз.

1.3.1 Брокер повідомлень

Такий тип систем бере дані з джерела, який називається продюсером, переводить їх у стандартний формат повідомлень і потоково передає їх. Інші компоненти можуть потім слухати та споживати повідомлення, передані брокером.

На відміну від старих брокерів, парадигми орієнтованої на повідомлення, потокові брокери підтримують дуже високу продуктивність із можливістю зберігати невідправлені повідомлення на диску машини, мають величезну потужність гігабайт в секунду або більше трафіку повідомлень, і тісно зосереджені на потоковому режимі з невеликою підтримкою перетворень даних або планування завдань[1]. На рисунку 1.1 видно структуру таких систем на прикладі RabbitMQ. Також прикладом таких систем є Apache Kafka, ZeroMQ, ActiveMQ та інші.

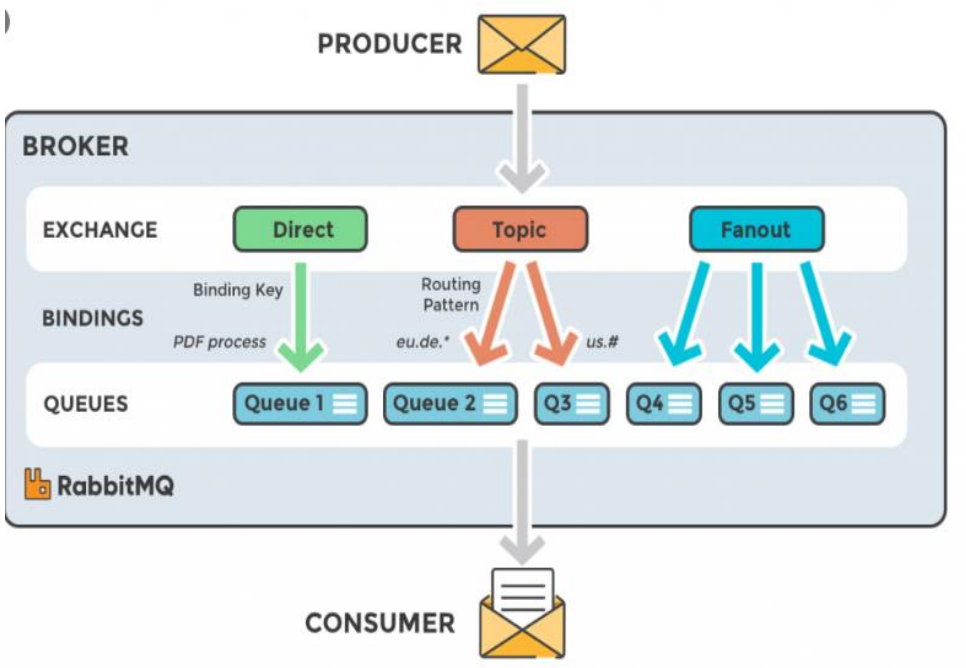

Рисунок 1.1 – Схема структури роботи брокера повідомлень

1.3.2 Інструменти ETL у режимі реального часу

Потоки даних від одного або декількох брокерів повідомлень потрібно агрегувати, трансформувати та структурувати до того, як дані можуть бути

проаналізовані за допомогою інструментів аналітики на основі мов запитів. Це робиться інструментом або платформою ETL, який отримує запити від користувачів, отримує події з черг повідомлень і застосовує запит, щоб створити результат, часто виконуючи додаткові приєднання, перетворення на агрегації даних[1]. Результатом може бути виклик API, дія, візуалізація, попередження або в деяких випадках новий потік даних. Приклад схематичної роботи та зміни стану такої системи зображено на рисунку 1.2.

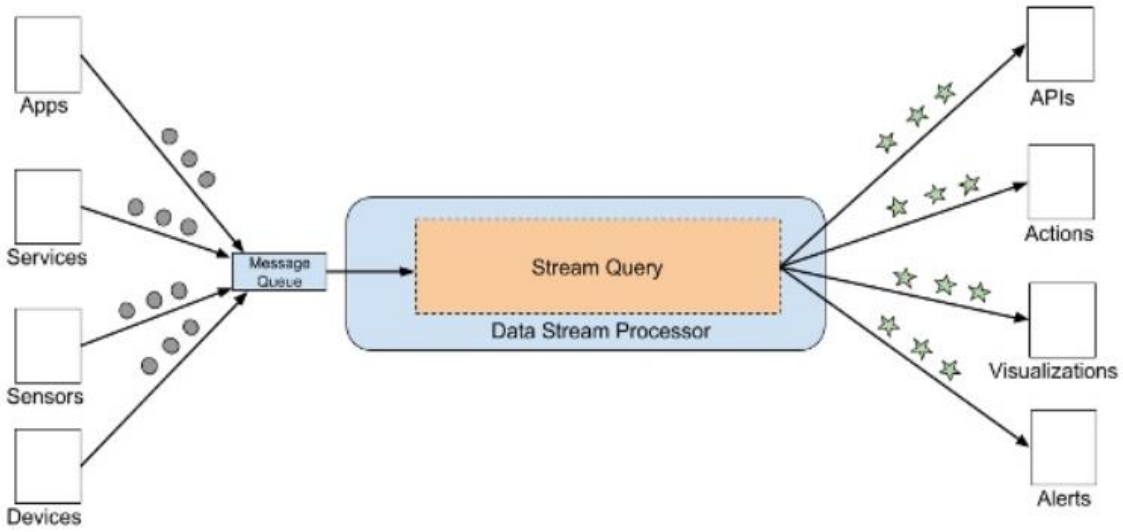

Рисунок 1.2 – Схема роботи ETL платформ

Кілька прикладів інструментів ETL з відкритим кодом для передачі даних - Apache Storm, Spark Streaming та WSO2 Stream Processor. Хоча ці програмні продукти працюють по-різному, всі вони здатні прослуховувати потоки повідомлень, обробляти дані та зберегти їх у сховищі.

1.3.3 Платформи аналітики даних

Після того, як потокові дані підготовлені до споживання процесором потоку, їх необхідно проаналізувати, щоб забезпечити цінність цих даних. Існує

багато різних підходів до потокової аналітики даних. Найчастіше використовуються для потокової аналітики даних:

— Elasticsearch, що використовується, якщо потрібно виконувати швидкий пошук по тексту та виконувати аналіз тексту;

— Apache Cassandra, що використовується, якщо потрібно транслювати потокові дані у додатки з низькою затримкою. Часто використовується для застосунків з необхідністю прийняття рішення в реальному часі;

— Amazon Redshift, що дозволяє майже в режимі реального часу здійснювати аналітику з інструментами business intelligence та інформаційною панеллю, які інтегруються з Redshift сервісами.

1.3.4 Потокове сховище даних

Системи, які не обробляють дані у реальному часі, повинні записувати їх у сховища даних для подальшого аналізу та обробки. З появою недорогих технологій зберігання даних більшість організацій сьогодні зберігають свої потокові дані про події. Варіанти зберігання мають свої недоліки та переваги, проте їх поділяють на три типи: зберігання в базі даних, зберігання у брокері повідомлень, зберігання у сховищах, які не мають структури. Останній тип набуває більшої популярності через те, що головною перевагою є можливість записувати дані будь-якого типу для відкладеної обробки. З етапом розвитку таких систем з'явились сервіси «озера даних», структуру яких зображено на рисунку 1.3. Озеро даних - це централізоване сховище, яке дозволяє зберігати структуровані і неструктуровані дані в будь-якому масштабі. На таких системах запускають різні види аналітики - від панелей моніторингу та візуалізації до обробки великих даних, аналітики в реальному часі і машинного навчання для прийняття правильних рішень.

Проаналізувавши аналоги системи, їх переваги та недоліки, стало можливим визначити ключові особливості, котрим повинен слідувати додаток, що розробляється.

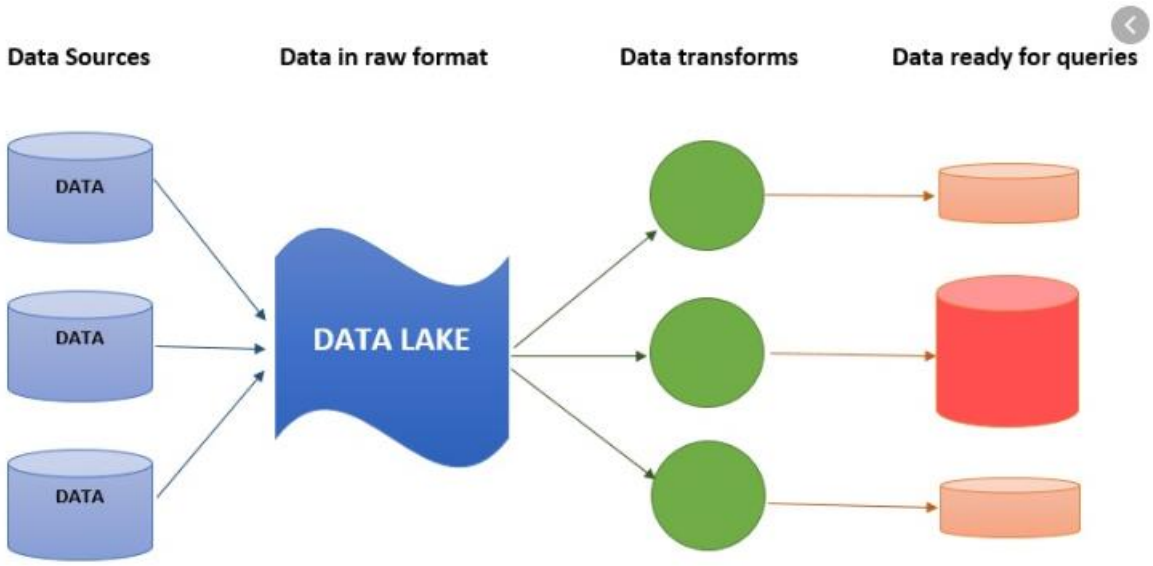

Рисунок 1.3 – Схема обробки даних при роботі з «озером даних»

Серед них можна виділити: система повинна працювати за шаблоном публікація-підписка, що дозволяє зменшити зв'язність між програмними компонентами продюсерів та консюмерів, тим самим надаючи гнучкість стійкості до відмов та можливості горизонтального масштабування системи при великому навантаженні[1]. Важливим аспектом системи, котра розробляється, є те, що вона складає собою готовий базис для потокового стримінгу даних, беручи основну функціональність з усіх вище описаних типів систем, які найчастіше використовуються разом, так як кожна з них має великий функціонал, який підходить для вирішення специфічних задач, поставлених бізнесом.

Висновки до розділу

У ході створення цього розділу було проаналізовано існуючі аналоги систем, рекомендації до побудови стримінгових платформ даних та на основі них було виведено функціональні та нефункціональні вимоги до проєктованої системи. Використовуючи отримані дані, стало можливим виокремити можливі переваги та недоліки системи, що розробляється.

Стисло можна виділити такі переваги сучасної архітектури платформи потокової передачі даних від джерела до клієнта:

— може усунути потребу у великих інженерних проєктах;

— продуктивність, висока доступність та вбудована стійкість до помилок;

— новіші платформи базуються на хмарній інфраструктурі, і вони можуть бути розгорнуті дуже швидко, без попередніх інвестицій;

— гнучкість та підтримка випадків багаторазового використання.

2 ВИБІР ТА ОБҐРУНТУВАННЯ КОМПОНЕНТІВ СИСТЕМИ

2.1 Вибір мови програмування та платформи реалізації

При проєктуванні будь-якої системи потрібно порівняти переваги та недоліки існуючих інструментів та вірно підібрати під задачі, які буде вирішувати команда розробників.

При розробці хмарних додатків потрібно вибирати інструмент, який підтримує кросплатформність. Найпопулярнішими інструментами є мови програмування на базі Java Virtual Machine, технологія для розробки серверних додатків Node.js на мові програмування JavaScript, PHP та декілька інших.

Компанія Microsoft після створення операційної системи Windows в більшості інвестувала ресурси в розробки технологій, які будуть працювати на Windows OS. Так було створено .NET Framework та мову програмування C#, яка взяла найкращі риси тогочасних мов JVM. Проте націленість на Windows OS не дозволяла створювати додатки на Linux, так як віртуальна машина Common Runtime Language не могла виконувати C# код, що в свою чергу було головною перешкодою виходу Microsoft на ринок хмарних обчислень.

У 2016 році було представлено новий програмний фреймворк .NET Core, який спроєктований з нуля для подальшої заміни .NET Framework, так як в ньому було виправлено всі недоліки минулого фреймворку[2]. Зараз ця технологія є дуже перспективною та на синтетичних тестах має більшу швидкість ніж Node.js та Spring.

Саме ця технологія була вибрана основною для розробленої платформи, так як вона має чудову масштабованість, пропускну здатність та велику кількість бібліотек з реалізацією доступу до програмних компонентів, баз даних, брокерів повідомлень та інше.

Мова програмування C# має найбільш зручний синтаксис, типізацію, що надає надійності, та велику підтримку зі сторони розробників з усього світу.

2.2 Архітектурні рішення системи

2.2.1 Модель акторів

Однією з головних переваг платформ стримінгу даних є їх здатність до незалежного збільшення фунціоналу та розширюваності, тому було вирішено будувати програмне забезпечення у об’єктно-орієнтованій парадигмі, так як можливо зробити архітектуру компонентів, як зв’язаних між собою об’єктів. Об'єктно-орієнтоване програмування - це парадигма програмування, ідея котрої полягає в представлені явищ та сутностей у вигляді програмних об’єктів, котрі інкапсулюють в собі характеристики та поведінку, тим самим дозволяють забезпечити програмі модульність та можливість повторного використання[3]. Об'єкти являють собою екземпляри класів (виступають шаблоном з описом характеристик), що використовуються для взаємодії один з одним для проєктування додатків та комп'ютерних програм.

Проте існує декілька представлень цієї парадигми. Методологія, що найчастіше використовується має такі недоліки:

— традиційні мови ООП не розроблялися з можливістю паралельних обчислень. Проблемою постає стан невизначеності результату «гонки» двох калькуляцій, які виконуються паралельно;

— для роботи з розшареними ресурсами, програмістам доводиться явно контролювати проблемні місця за допомогою блокування критичних секцій та її механізмів;

— блокування коду легко здійснити для простих програм. Але оскільки програми стали складними, реалізація блокування також стала складною.

Для усунення цих проблем існує інша методологія, яка має назву модель акторів. Модель актора в інформатиці - це математична модель паралельних обчислень, яка розглядає актора, як універсальний примітив паралельних обчислень. У відповідь на повідомлення, яке воно отримує, актор може: приймати місцеві рішення, створювати більше акторів, надсилати більше повідомлень та визначати, як відповісти на наступне отримане повідомлення. Актори можуть

змінювати свій власний приватний стан, але можуть впливати один на одного опосередковано через обмін повідомленнями (забороняючи синхронізацію на основі блокування).

Модель акторів переймає філософію, що все - це актор. Це схоже на «все є об'єкт» філософію, яку використовують об'єктно-орієнтовані мови програмування. Вважається, що модель акторів є справжнім підходом до об’єктно-орієнтованої парадигми програмування тому, що він був описаний в джерелі, з якого вперше з’явилась назва цієї парадигми[4].

Головним шляхом комунікації між сервісами є асинхронні повідомлення між акторами. Через повідомлення інкапсулюються команди та дані для актора, який є адресатом цього повідомлення. Схематично систему акторів та взаємодію між ними зображено на рисунку 2.1. Серед команд акторів може бути створення нового актору або розпаралелювання роботи з ресурсами на іншу систему акторів на другому сервері.

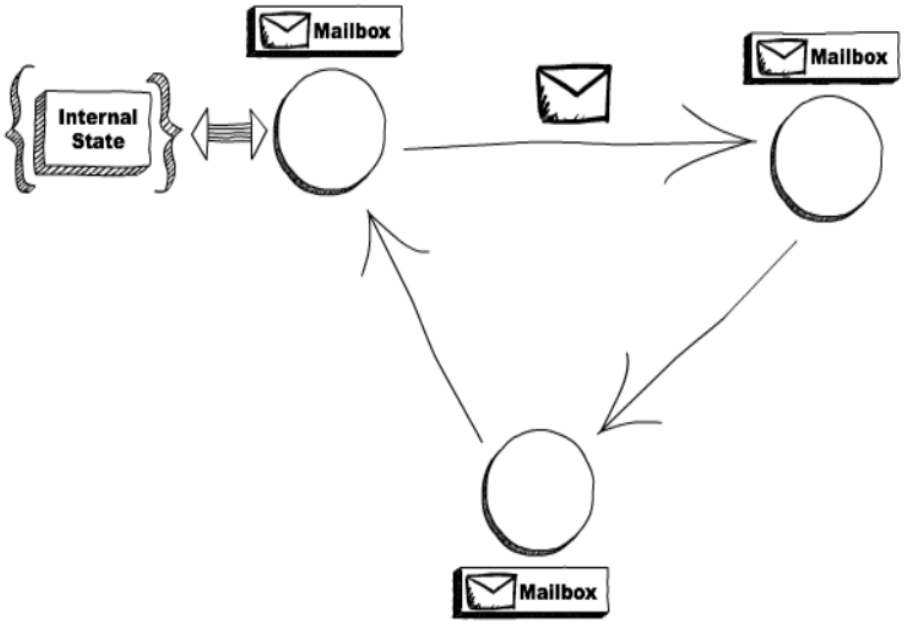

Рисунок 2.1 - Актори спілкуються один з одним, надсилаючи асинхронні повідомлення

Це забезпечує не блокуючі (ситуація, коли один сервіс чекає виконання іншого) виконання обчислень, а також слабку зв'язність сервісів, що дозволяє легко масштабувати систему. Актори можуть ділитись на ізольовані системи акторів та запускатись на різних машинах, при тому залишати можливість комунікації один з одним. На принципах моделі акторів будується архітектура реактивних систем.

Для комунікації між сервісами використовуються брокери повідомлень, такі як RabbitMQ та Apache Kafka. В реалізації програмного продукту було використано Apache Kafka, тому що він має такі переваги[5]:

— висока пропускна здатність. Не маючи настільки великого обладнання, Kafka здатний обробляти дані з високою швидкістю та великим обсягом. Також здатний підтримувати пропускну здатність тисяч повідомлень в секунду;

— низька затримка. Він здатний обробляти ці повідомлення з дуже низькою затримкою діапазону в мілісекунди, яку вимагає більшість нових випадків використання;

— толерантність до помилок. Однією з найкращих переваг є допущення відмов. У Kafka є властива здатність бути стійким до відмови вузла / машини в кластері;

— довговічність. Тут довговічність стосується збереження даних або повідомлень на диску. Крім того, реплікація повідомлень є однією з причин довговічності, отже, повідомлення ніколи не втрачаються.

2.2.2 Мікросервісна архітектура

Вибір архітектури є дуже важливим рішенням на початку розробки, так як в подальшому це матиме пряму залежність на витрати нового функціоналу. На початку розробки будь-який тип архітектури підходить, так як функціоналу не вистачає для того, щоб зіткнутись з проблемами, проте чим більший час проєкту, тим ціна за помилку на початку проєктування вища. Тому більшість бізнесів, які починають створювати свої продукти консультуються з експертами предметної області та архітекторами програмного забезпечення, щоб мінімізувати ризики та

зекономити час розробників на подальшу підтримку неправильних рішень та мінімізацію технічного боргу.

Під час підбору архітектурного рішення для системи було переглянуто чотири типи побудови архітектури: моноліт, сервісно-орієнтована, мікросервісна та безсерверна.

Монолітна архітектура є найрозповсюдженішим типом у світі програмних додатків, так як вимагає найменше зусиль на етапі розробки, чудово підходить для демонстрації прототипу та потребує набагато менше навичок проєктування зі сторони команди розробників. Більшість веб-додатків мають монолітну структуру через особливості інструментів для вирішення тих чи інших проблем. Не дивлячись на те, що моноліти набагато дешевші у підтримці під час розробки та чудово підходять до розробки у невеликій команді розробників, вони мають свої значні мінуси.

Більшість великих корпорацій мають додатки з даним типом, тому що зв'язаність компонентів та складність бізнес логіки не дозволяють масштабувати такі додатки, і витрати на перебудову або створення нового продукту дуже великі, порівняно з підтримкою існуючого додатку монолітного типу.

Вважається, що цей тип архітектури не справляється з сучасними проблемами, перед якими бізнес повинен шукати рішення, через те, що такі системи важко та дорого масштабувати, і вони можуть вести себе непередбачувано при високому навантаженні та при великій кількості активних користувачів, що призводить до того, що проблеми в додатку дуже важко знайти та відтворити, що в свою чергу значно збільшує час розробника на пошук правильного підходу до тої чи іншої проблеми[8].

Серед переваг виділяють:

— простота розробки: монолітні архітектури прості в створенні, тестуванні та розгортанні. Такі характеристики надають можливість для швидкої розробки та швидкого розгортання окремої однієї версії, що дуже скорочує затрачений час на розробку та на навчання складним підходам. Ці програми можуть

масштабувати горизонтально в одному напрямку, виконавши кілька копій програми за балансиром навантаження;

— простота у підтримці інфраструктури: маючи одну велику базу коду, команда має можливість вирішувати такі складні проблеми розподілених систем, як: моніторинг стану системи, логування транзакцій та операцій всередині системи та налаштування ключів, фунціоналу та адрес підключення. Маючи все в одному додатку, стає легко повторно використовувати конфігурації та модулі, які застосовуються в інших місцях, які потрібно відслідковувати;

— продуктивність: якщо система не є високо інтенсивною у відношенні до калькуляцій та обчислень, монолітна архітектура дозволяє зменшити кількість запитів на інші сервіси, що в свою чергу усуває затримку на мережевий запит, що виступає однією з переваг над розподіленими системами. Комунікація відбувається на рівні процесу, що зі свого боку набагато швидше та може бути оптимізованою можливостями операційної системи. Також відпадає необхідність використання розподілених кешів для швидкого доступу до даних, тому що можна зберігати дані в пам’яті додатку, проте існує шанс втратити дані при вимиканні системи;

Серед важливих мінусів можна виділити:

— відсутність повторного використання: монолітні програми не використовуються повторно, оскільки вони використовують інструменти з цільової платформи та інструменти, які вона надає специфічно до підходу. Не надається можливості повторного використання функції та інтегрувати її в інший проєкт, оскільки це залежить від загальної розробленої екосистеми. Це означає, що фунціонал не буде працювати поза розробленим додатком;

— проблеми при масштабуванні: так як моноліт виконує усі бізнес операції, зберігає дані та утримує сесії обробки, то проблеми починаються при масштабуванні таких систем, тому що горизонтальне масштабування не може гарантувати цілісність даних та стійкість до їх втрати, а вертикальне масштабування в своє чергу може бути дуже дорогим для великих компаній та їх продуктів, так як вимагає час для повторного налаштування або покупки нового обладнання, якщо компанія не використовує послуги хмарних провайдерів;

— представлення системи, як чорний ящик: з часом росту кодової бази та зав’язки її на екосистему використаної технології, компоненти починають бути складними та зав’язаними один на одного, тому при розробці нового незалежного фунціоналу потрібно брати до уваги існуючі модулі та процеси, що в свою чергу може бути складним та недокументованим. В таких випадках найчастіше виходом є розширення існуючого або перевизначення програмних інтерфейсів, що не дозволяє використовувати платформу за прямим призначенням для вирішення проблем бізнесу;

— неможливість використання відповідних технологій: велика кількість інструментів та платформ дають різноманітний фунціонал, тому вибрати початкову платформу для розробки може не вистачити в подальшому для специфічних задач. Додання нових технологій змушує розробників переписувати модулі системи або приймати архітектурні рішення для розширення системи, а найгіршим варіантом буде написання свого пропрієтарного варіанту існуючого інструменту, що може бути необхідним для специфічних цілей, проте дорогим для підтримки у часі;

— версіювання системи: розробляючи продукт у великій команді, впровадження нового фунціоналу може бути складним, тому що важко тримати актуальну версію у кожного розробника. Для клієнтів ця проблема постає в тому, щоб випустити новий фунціонал в експлуатацію, потрібно перезапускати систему та заливати її на хостинг, що для систем, які повинні бути 99% часу у робочому стані, дуже критично.

Монолітну архітектуру слід розглядати, якщо вам не потрібно працювати з паралельними обчисленнями великих масивів даних, великим навантаженням, або проєкт не буде мати велику та складку логіку, котра буде підтримуватись великою кількістю розробників протягом років. Простоту такої архітектури зазвичай зображають, як на рисунку 2.2. Про успіх під час розробки системи невідомо, тому можна починати з монолітної архітектури, проте всередині програмні модулі варто розділяти на шари та окремі бібліотеки для створення слабкої зв’язності всередині моноліту для подальшої можливості мігрувати до інших типів архітектури шляхом створення нових сервісів з окремих модулів.

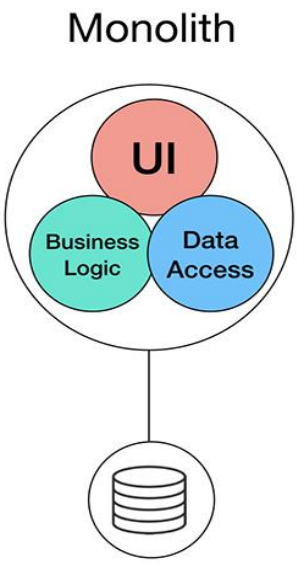

Рисунок 2.2 – Приклад монолітної архітектури

Далі було розглянуто сервісно-орієнтовану архітектуру додатків, ідея котрої була розв’язати монолітний застосунок на декілька зв’язаних між собою сервісів. Цей тип архітектури активно використовується у великих промислових та корпоративних додатках, тому що дає можливість забезпечити покриття недоліків монолітної архітектури. Виводити окремі функціональні компоненти в сервіси дозволяє вирішити проблему серійності продукту та розділити роботу на окремі незалежні команди розробників. Така архітектура проєктується так, що кожен сервіс знає один про одного найчастіше через центральний сервіс, проте може бути стійким до відмови одного з них під час транзакції, тим самим забезпечуючи можливість для інтегрування нового фунціоналу та розширювання старого при тому, що вартість на підтримку інфраструктури не значно збільшилась порівняно з монолітною архітектурою. Такий тип архітектури забезпечує слабку зв’язність між компонентами, що дозволяє розробникам сконцентруватись на розробці окремого бізнес-фунціоналу.

Розроблений у 2009 році маніфест сервісно-орієнтованої архітектури описує шість цінностей при побудові систем з таким типом[7]:

— бізнес цінності мають більше значення, ніж технічна стратегія;

— стратегічні цілям надається більше значення, ніж конкретним перевагам проєкту;

— внутрішній сумісності надається більше значення, ніж спеціальній інтеграції;

— спільним службам надається більше значення, ніж реалізація конкретних цілей;

— гнучкість системи має важливішу перевагу, ніж оптимізація додатку;

— еволюційному вдосконаленню надається більше значення, ніж прагнення до початкової досконалості.

Схематично ідеї та приклад архітектури зображають приблизно до тої, що на рисунку 2.3.

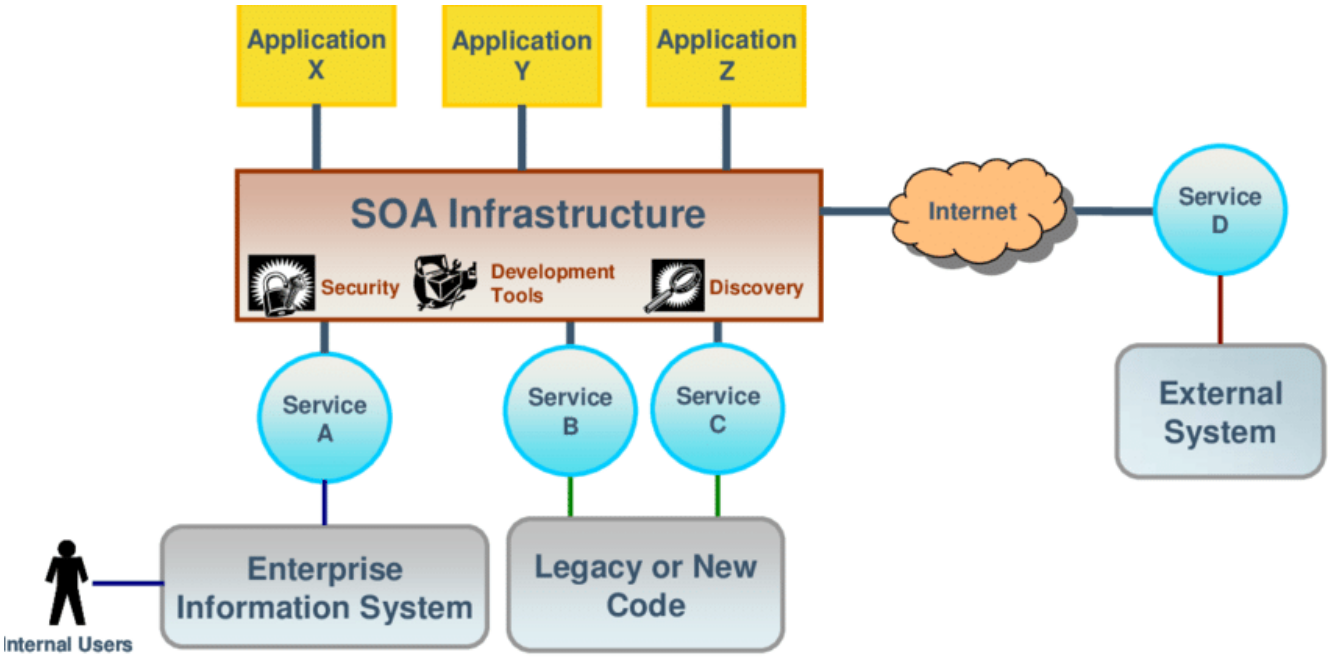

Рисунок 2.3 – Компоненти СОА системи

Саме через такі ідеї сервісно-орієнтована архітектура вважається найкращою на сьогодення, та впроваджується в більшість сфер від банківської до стримінгу.

Підсумовуючи, можна виділити такі переваги:

— стійкість до відмов: завдяки ізоляції кожного функціонального сервісу системи, відмова одного сервісу не буде критично впливати на роботу інших, що в свою чергу дає більше гнучкості для підтримки;

— ціна розробки та підтримки: кожен модуль може розроблятися паралельно різними командами, що в свою чергу підвищує швидкість створення нового функціоналу та можливість для підтримки різних версій продукту одночасно. Також такі системи легко тестуюся, що дозволяє знаходити помилки до того, як продукт буде офіційно випущено;

— повторне використання фунціоналу: кожен окремий модуль виступає незалежним сервісом, що в свою чергу дозволяє його повторне використання в різних частинах продукту, що розробляється, зміна такого сервісу без змін контракту між сервісами, призводить до можливості написання нового фунціоналу без створення нового коду на стороні сервісів-клієнтів.

Проте такий тип архітектури також має свої недоліки:

— початкова складність розробки: починаючи розробляти додаток за такою архітектурою, команда розробників повинна мати певні навички у проєктуванні даного типу систем. Також, деякий час потрібно буде відвести на налаштування інфраструктурної частини додатку, таку як: моніторинг, логування, конфігурація та інше. Також вибір протоколу комунікації меж сервісами може в майбутньому давати затримку, яка може стати критичною для тої чи іншої частини системи, тому це також важливо ураховувати на початку створення додатку;

— управління системою: під час розгортання системи у реліз режимі потрібно переконатись, що кожен сервіс працює коректно та має можливість обмінюватись повідомленнями з іншими. Спосіб комунікації повинен також контролюватись, щоб не перевищувати допустиму пропускну здібність системи, щоб запобігти виникненню відмов при проведені транзакцій;

— затримка відгуку: так як комунікація найчастіше виконується через HTTP або протоколи обміну через брокери повідомлень, то вони мають свою затримку у мережі, що може бути критичним для швидкої обробки даних, також сервіси повинні мати можливість обробити проблеми зі зв’язком або слабким підключенням при обміні повідомлень через інтернет.

Розроблений додаток дипломного проєкту взяв найкращі риси такого типу архітектури. Підсумовуючи, можна сказати, що сервісно-орієнтована архітектура -

це найкращий вибір, якщо застосунок буде розроблятись та використовуватись протягом років.

На даний момент найбільш спірною у спільноті розробників є саме мікросервісна архітектура, яка в свою чергу є різновидом сервісно-орієнтованої архітектури. Мікросервіси - це сучасна інтерпретація сервісно-орієнтованої архітектури, яка використовується для побудови розподілених програмних систем. Сервіси в архітектурі мікросервісів - це процеси, які спілкуються один з одним по мережі з метою досягнення поставленої мети[6]. Ці сервіси використовують технологічні агностичні протоколи, які допомагають укладати вибір мови та рамки, роблячи їхній вибір внутрішньою службою, що дозволяє в свою чергу вирішувати специфічні задачі для тої чи іншої сфери. Цей підхід чудово працює з гнучкою методологією розробки та DevOps процесами, які створились після того, як мікросервіси набрали популярність у світі серед корпорацій.

Не існує єдиного загальновизнаного визначення мікросервісів. У літературі можна знайти такі характеристики та принципи:

— добре розділені інтерфейси, що надають можливість створювати незалежні від розгортання сервіси;

— бізнес-орієнтована розробка (наприклад, орієнтована на предметну область);

— просунутий підхід для побудови хмарних додатків;

— можливість використовувати різні мови програмування та сховища даних;

— легке розгортання додатків у контейнерах;

— децентралізована можливість версіонування та розгортання;

— DevOps з цілісним моніторингом сервісів.

До значних переваг мікросервісної архітектури виділяють:

— поліпшену ізоляцію несправностей: більші програми можуть залишатися в робочому стані критично, не реагуючи на несправності одного модуля, що в свій час надає змогу і далі працювати системі, окрім деякого фунціоналу, що тимчасово задовольняє потреби бізнесу;

— свобода у виборі технологій: мікросервіси забезпечують гнучкість випробувати новий стек технологій на індивідуальному сервісі за потребою. Не буде такої кількості проблем, що залежать, і відміна змін стає набагато простішою. З меншим кодом у кожному сервісі є більша гнучкість. Через можливість вибору технологій відповідно до потреб, великі компанії мають команди розробників у різних сферах, що вирішують специфічні проблеми з відповідними інструментами;

— простоту розуміння: розробники мають можливість виносити функціональність в окремі мікросервіси, що покривають певну задачу з малою кількістю кодової бази, що в свою чергу легко підтримувати та використовувати;

— менші та швидші розгортання: менші бази коду та область застосування пропорційні більш швидким розгортанням системи, що дозволяє використовувати переваги безперервного розгортання та мати декілька різних версій одного і того самого сервісу та балансувати трафік між ними для детального тестування нового фунціоналу;

— масштабованість: оскільки сервіси є окремими, при такій архітектурі легше масштабувати найбільш потрібні сервіси у відповідний час, для можливості утримати та обробити велике навантаження на систему, на відміну від усієї програми. Якщо зробити все правильно, це може вплинути на економію коштів. Тому великі компанії мають окремих спеціалістів, котрі займаються автоматизацією розгортання системи або слідкують за тим, що все працює вірно.

Проте також присутні недоліки, які не дозволяють використовувати такі системи малими компаніями та продуктами, які не розроблюються з підтримкою великої кількості фунціоналу та великого навантаження:

— складність розробки: найбільший недолік мікросервісної архітектури виступає в проєктуванні та підтримці, як одного цілого продукту. Великі затрати на проєктування та вирішення проблем, можуть бути критичними для невеликих команд. Також готові рішення інфраструктурних проблем від хмарних провайдерів вимагають додаткових витрат та експертизу зі сторони команди;

— швидкість розробки: підтримка інфраструктурної частини системи та способи комунікації можуть забрати велику частину часу розробників при старті

нового сервісу, також складність розгортання системи може бути великою, якщо немає спеціалістів, які оптимізували цю роботу;

— безпека системи: так як комунікація відбувається через повідомлення, то цей трафік може бути перехоплений та замінений з метою викрадення даних. Питання безпеки також вимагає додаткових інфраструктурних вкладень в систему та експертизу команди розробників.

Будуючи систему на основі даної архітектури, прийнято розділяти сховища даних до відповідних сервісів, тим самим на зав'язуючись на одну базу, через яку працюють всі сервіси, також, не дивлячись на складну систему зі сторони серверу, найчастіше такі системи мають лише один клієнтській застосунок, котрий запускається у браузері. На рисунку 2.4 видно, як зазвичай виглядають такі системи.

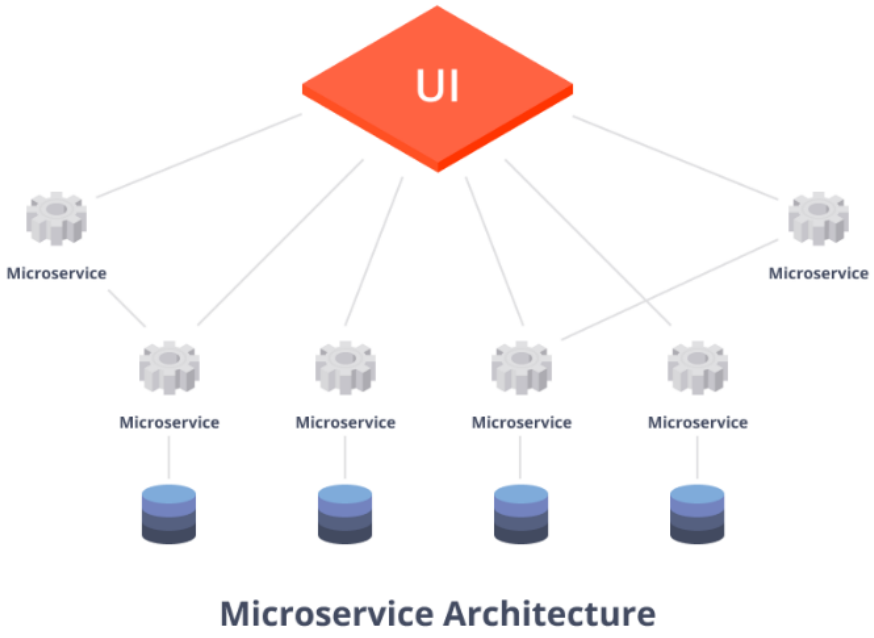

Рисунок 2.4 – Схема мікросервісної архітектури

Підсумовуючи, потрібно сказати, що такий тип архітектури є дуже гнучким у розробці та балансуванні трафіку або навантаження, що є найбільшою перевагою для великих корпорацій, таких як Apple, LinkedIn, Google та інші. Даний тип

архітектури було частково використано у розробці дипломного проєкту, що більше схоже на сервісно-орієнтовану архітектуру, так як система не має багатого фунціоналу, поділеного на граничні контексти предметної області, що притаманно архітектурі мікросервісів.

З активним розвитком хмарної інфраструктури та можливостей хмарних провайдерів з'явився такий тип архітектури, як безсерверний. Безсерверна архітектура (зустрічається в зарубіжних джерелах, як абревіатура FaaS, що означає Function As A Service) - це шаблон дизайну програмного забезпечення, де програми розміщуються у сторонніх сервісах, що виключає необхідність розробника у серверному програмному забезпеченні та обкладенні. Програми розбиті на окремі функції, які можна викликати та масштабувати окремо. Приклад системи, побудованої на безсерверних застосунках, зображено на рисунку 2.5.

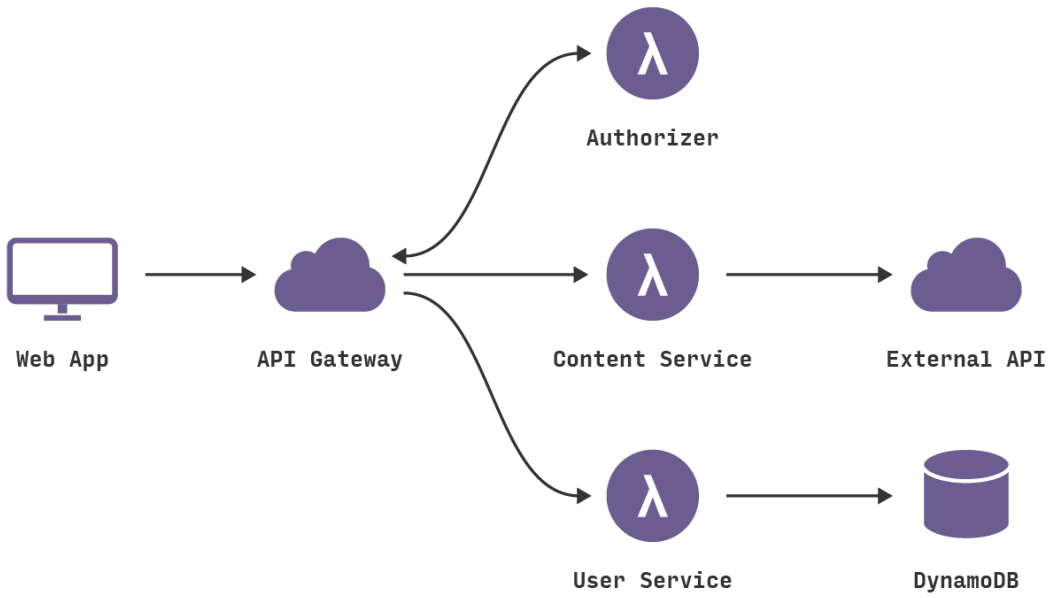

Рисунок 2.5 – Архітектура з використанням безсерверних сервісів

Розміщення програмного забезпечення в Інтернеті зазвичай передбачає управління якоюсь інфраструктурою сервера. Зазвичай, це означає віртуальний або фізичний сервер, яким потрібно керувати, а також операційну систему та інші процеси розміщення веб-серверів, необхідні для запуску вашої програми. Використання віртуального сервера від хмарного постачальника, таких як Amazon або Microsoft, означає усунення проблем з фізичним обладнанням, але все ж

вимагає певного рівня управління операційною системою та процесами програмного забезпечення веб-сервера[9].

Головною перевагою такого підходу є те, що команді розробників не потрібно витрачати час на підтримку інфраструктури, на налаштування масштабування та розгортання, а можна більше сфокусуватись на розробці бізнес фунціоналу.

2.3 Реактивні системи

На основі ідей моделей акторів та архітектури мікросервісів для досягнення стійкості систем та можливості обробки великого навантаження було створено підхід – реактивні системи. Ідея цих систем є основою для усіх стримінгових платформ даних.

Системи, побудовані, як реактивні системи, є більш гнучкими, нещільно пов'язаними та масштабованими. Це полегшує їх розробку і гнучкість змінам. Вони значно стійкіші до помилок, і коли трапляються помилки, вони зустрічають це за просунутими підходами та ідеями, а не викликають критичне завершення програми або виконання. Реактивні системи відрізняються високою здатністю відповідати на запити при будь-яких ситуація, що дає користувачам ефективний інтерактивний зворотний зв'язок. Синергія усіх компонентів реактивних систем зображена на рисунку 2.6.

У маніфесті реактивних систем описуються такі 4 принципи, як: готовність відповідати на запити в найкритичніших випадках, відновлюваність, гнучкість, керованість повідомленнями[10].

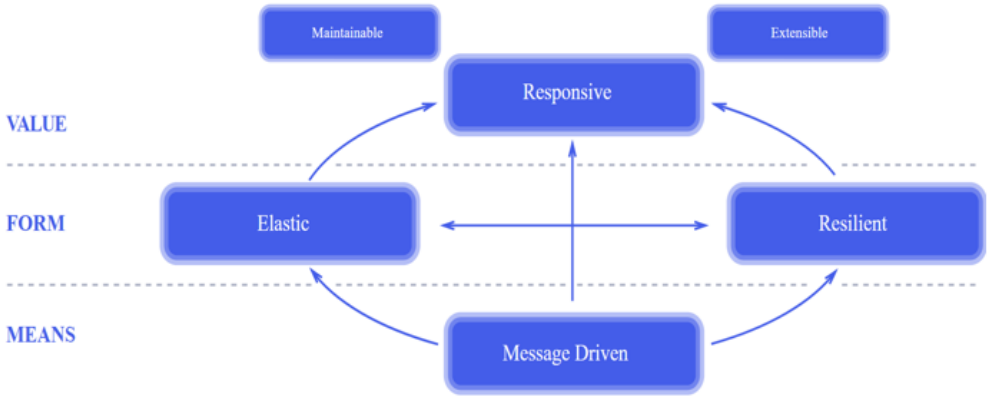

Рисунок 2.6 – Схема концептів реактивних систем

— готовність відповідати: система реагує своєчасно, якщо це можливо. Відповідальність є наріжним каменем зручності використання та корисності, але, більше того, готовність до відповідей означає, що проблеми можна швидко виявити та ефективно вирішити. Чутливі системи фокусуються на забезпеченні швидкого та послідовного часу реагування, встановлення надійних верхніх меж, щоб вони забезпечували стабільну якість обслуговування. Ця послідовна поведінка в свою чергу спрощує обробку помилок, формує довіру кінцевих користувачів та заохочує подальшу взаємодію;

— відновлюваність: система залишається чутливою перед збоями. Це стосується не лише високодоступних, критично важливих систем - будь-яка система, яка не є стійкою, буде не реагувати після відмови. Стійкість досягається шляхом тиражування, стримування, ізоляції та делегування. Поломки містяться всередині кожного компонента, ізолюючи компоненти один від одного, і тим самим забезпечуючи, що частини системи можуть вийти з ладу та відновитись без шкоди для системи в цілому. Відновлення кожного компонента делеговано іншому (зовнішньому) компоненту, а висока доступність забезпечується шляхом реплікації, де це необхідно. Клієнт компонента не обтяжений поводженням з його відмовами;

— гнучкість: система залишається чутливою при різному навантаженні. Реактивні системи можуть реагувати на зміни швидкості введення, збільшуючи або зменшуючи ресурси, виділені для обслуговування цих входів. Це передбачає конструкції, які не мають суперечливих питань або центральних вузьких місць, в результаті чого можливе шматування або копіювання компонентів та розподілення вхідних даних між ними. Реактивні системи підтримують прогнози, а також алгоритми реактивного масштабування, забезпечуючи відповідні заходи в реальному часі. Вони економічно досягають еластичності на товарних і програмних платформах;

— керованість повідомленнями: реактивні системи покладаються на асинхронну передачу повідомлень для встановлення межі між компонентами, що забезпечує вільне з'єднання, ізоляцію та прозорість розташування. Цей кордон також забезпечує засоби для делегування відмов, як повідомлень. Використання явної передачі повідомлень дозволяє керувати навантаженням, еластичністю та контролем потоку, формуючи та контролюючи черги повідомлень у системі та застосовуючи зворотний тиск, коли це необхідно. Розташування прозорого обміну повідомленнями, як засобу комунікації дозволяє керувати непрацездатністю однакових конструкцій та семантикою в кластері або в межах одного хосту. Неблокуючий зв’язок дозволяє одержувачам лише споживати ресурси під час активних дій, що призводить до менших витрат системи.

Великі системи складаються з менших, і тому залежать від реакційних властивостей їх складових. Це означає, що реактивні системи застосовують принципи проєктування, тому ці властивості застосовуються на всіх рівнях масштабу, що робить їх зручними[10]. Найбільші в світі системи покладаються на архітектури, що базуються на цих властивостях і обслуговують потреби мільярдів людей щодня. Настав час застосовувати ці принципи дизайну свідомо від самого початку, замість того, щоб їх знову відкривати.

2.4 Сховища даних

2.4.1 NoSQL бази даних

Сучасними стандартами для сховищ даних виступають реляційні бази даних. Реляційна база даних - це тип бази даних, який використовує структуру, яка дозволяє ідентифікувати та отримувати доступ до даних стосовно іншого фрагмента даних у базі даних. Часто дані в реляційній базі даних організовуються в таблиці. Проте такий тип баз даних має критичні недоліки для роботи з платформами стримінгу даних[11]:

— модель реляційних даних не відповідає кожному домену;

— еволюція складної схеми через негнучку модель даних;

— слабка розподілена доступність через погану горизонтальну масштабованість;

— показники продуктивності через з'єднання, транзакцій ACID та суворих обмежень послідовності (особливо в розподілених середовищах).

Швидкість обробки запитів, структура даних та масштабованість - головні критерії, за якими було обрано NoSQL базу даних.

Бази даних NoSQL використовують різноманітні моделі даних для доступу до даних і управління ними. Бази даних таких типів оптимізовані для додатків, які працюють з великим обсягом даних, потребують низької затримки і гнучких моделей даних. Все це досягається шляхом пом'якшення жорстких вимог до несуперечності даних, характерних для інших типів БД.

Для вирішення проблем з масштабованістю платформа зберігає дані у Apache Cassandra. Apache Cassandra - розподілена система управління базами даних, що відноситься до класу NoSQL-систем і розрахована на створення високомасштабованих і надійних сховищ величезних масивів даних, представлених у вигляді хешу[12]. Також використовується у процесі платформ ETL для прийняття рішень у реальному часі.

2.4.2 Розподілений кеш

Розподілений кеш - це розширення традиційної концепції кешу, що використовується в одному локалі. Розподілений кеш може охоплювати кілька серверів, щоб він міг збільшуватися в розмірах і в транзакційній ємності. В основному використовується для зберігання даних додатків, що знаходяться в даних бази даних та веб-сеансів.

Кеши мають велику швидкість доступу до ресурсів, тому використання їх зменшує час обробки подій. Для розподіленого кешу використовується Redis.

Redis - це проєкт структури даних в пам'яті, що реалізує розподілену базу даних зі значеннями ключів у пам'яті з необов'язковою тривалістю[13].

2.5 Комунікація

При проєктуванні розподілених систем дуже важливо враховувати, як сервіси будуть обмінюватись інформацією один з одним, так як на цьому будується вся система.

Сучасним лідером при виборі того, як служби спілкуватимуться між собою, як правило, є HTTP. Популярність цього підходу забезпечена тим, що за цим протоколом працюють більшість веб-сервісів та браузерів. Цей протокол має розширення HTTPS, котрий надає додаткової безпеки при обміні повідомленнями, що в свою чергу покриває єдиний недолік з проблемами безпеки, та виділяє такі сильні сторони як: запити без стану, гарантія того, що дані будуть цілісні, так як цей протокол працює поверх TCP протоколу, який гарантує гарний зв’язок між клієнтом та слухачем.

Дані через HTTP передаються у форматі JSON або XML, та робота з такими запитами до даних обробляється за архітектурними підходами проєктування програмного інтерфейсу як REST чи GraphQL[14].

Обмін через цей спосіб може відбуватись в синхронному варіанті та асинхронному. Синхронний підхід має на увазі, що той сервіс, що відправив

повідомлення буде чекати відповіді, та не може займатись іншою роботою до поки не отримає. Найчастіше такі запити є передбачуваними та добре піддаються пошукам помилок, проте вони є поганими зі сторони продуктивності сервісу. Проблему блокування вирішують через запуск таких запитів в окремому потоці процесу. Асинхронний підхід дозволяє обробляти інші процеси у системі, поки сервіс чекає відповідь. При такому підході сервіси ізольовано одне від одного, і зв'язок стає слабо зв’язаним. Мінусом є те, що він створює додаткові запити HTTP на другому сервісі. Тепер це буде запитуватися ззовні, поки запит не буде завершений. Це вводить і складність для клієнта, оскільки тепер він повинен перевіряти хід запиту. Але асинхронний зв’язок дозволяє сервісам залишатися вільно пов'язаними один з одним.

Ще одна схема комунікації, яку можемо використовувати в архітектурі мікросервісів - це комунікація на основі повідомлень[14]. Такий тип було вибрано, як основний при розробці проєкту.

На відміну від HTTP-зв’язку, залучені служби безпосередньо не спілкуються між собою. Натомість служби надсилають повідомлення брокеру повідомлень, на які підписалися інші служби. Це виключає велику складність, пов’язану з HTTP-зв’язком. Не потрібні служби, щоб знати, як спілкуватися між собою. Це усуває необхідність послуг безпосередньо комунікувати один з одним. Натомість усі служби знають про брокера повідомлень, і вони надсилають повідомлення цьому брокеру. Інші служби можуть вибрати підписку на повідомлення брокера, які їх цікавлять.

Існує багато брокерів повідомлень з різним фунціоналом та підходами, у роботі було вибрано Apache Kafka, тому що він пропонує чудову масштабованість та надійність.

Останній шаблон спілкування - це модель, що керується подією. Це ще один асинхронний підхід, і він повністю усуває зв’язок між сервісами[14].

На відміну від схеми обміну повідомленнями, де служби повинні знати загальну структуру повідомлень, підхід, орієнтований на події, не потребує цього.

Спілкування між службами відбувається через події, які виробляють окремі сервіси.

Тут ще потрібен брокер повідомлень, оскільки окремі служби записують на нього свої події. Але, на відміну від підходу до повідомлення, споживчим службам не потрібно знати деталі події; вони реагують на виникнення події, а не повідомлення, яке подія може або не може доставити.

Ця схема підтримує сервіси слабо зв'язаними, оскільки даних з описом не включаються. Кожен сервіс при такому підході реагує на виникнення події, щоб керуватися її бізнес логікою. Тут ми надсилаємо події через тему SNS (Amazon Simple Notification Service). Можуть бути використані й інші події, наприклад завантаження файлів або оновлення рядків бази даних.

2.6 Логування

Журнали подій фіксують події, що відбуваються при виконанні системи, щоб забезпечити аудиторський слід, який можна використовувати для розуміння діяльності системи та діагностики проблем. Вони мають важливе значення для розуміння діяльності складних систем, особливо у випадку програм із малою взаємодією користувачів (наприклад, серверних додатків). Також може бути корисно поєднувати записи файлів журналу з декількох джерел. Цей підхід, у поєднанні зі статистичним аналізом, може призвести до кореляції між, здавалося б, неспорідненими подіями на різних серверах. Інші рішення використовують запити та звіт про всі мережі.

Логування – це процес збирання для логів для створення журналів подій систем. Зазвичай логи пишуться в консоль, виводяться на екран, пишуться в файл на диску або у базу даних. Сервіси стримінгової платформи, що розробляється, пишуть логи у Elasticsearch. Elasticsearch - це пошукова система, що зберігає свої дані у вигляді документів, як деякі види NoSQL баз даних. Вона надає повнотекстовий пошуковий механізм, що підтримує багатосторонні дані, із веб-інтерфейсом HTTP та документами JSON з довільними схемами[15].

Головною особливістю використання цієї технології у даному продукті є те, що Elasticsearch дозволяє виконувати швидкий пошук по журналам та мати просунуті критерії фільтрації записів.

Налаштування сервісів логування вимагає багато зусиль та навичок зі сторони розробників, хоча логування подій у журнали є однією з базових функцій операційної системи. Виходячи з цього, було створено таких підхід, у якому сервіси збирають логи та пишуть його у системний вивід або просто зберігають, а зовнішні сервіси роблять запит на отримання даних. В свою чергу це зменшує навантаження на сервіси, що дуже добре в розподілених високонавантажених системах. Далі такі дані акумулюються та відправляються туди, де дані зберігаються, наприклад на сервер Elasticsearch.

2.7 Візуалізація

Візуалізація даних - це графічне зображення інформації та даних. Використовуючи візуальні елементи, такі як діаграми, графіки та карти, інструменти візуалізації даних забезпечують доступний спосіб бачити та розуміти тенденції, формати та структури даних. Важко аналізувати дані та статистику у вигляді великих масивів чисел та записів у таблицю, тому вирішено було використовувати візуалізацію для розуміння статистики статусів роботи сервісів, зміни значень датчиків та результатів спортивних подій.

Для візуалізації було вибрано технологію Kibana. Технологія включає в себе широку бібліотеку готових рішень для візуалізації даних різного типу складності. Прикладом роботи такої системи виступає набір графів та діаграм, як зображено не рисунку 2.7.

Kibana - інформаційна панель візуалізації даних із відкритим кодом для Elasticsearch. Вона надає можливості візуалізації поверх вмісту, індексованого на кластері Elasticsearch.

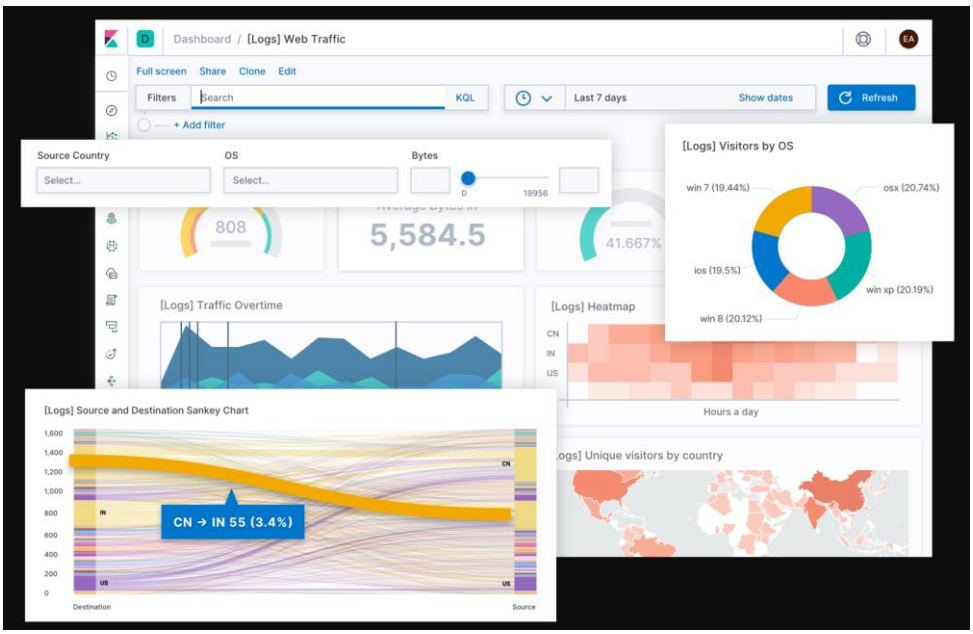

Рисунок 2.7 – Приклад візуалізації даних з Kibana

Користувачі можуть створювати графіки смуг, ліній та розсипань, або кругові діаграми та карти на великих обсягах даних. Kibana також пропонує інструмент для презентації, іменований Canvas, який дозволяє користувачам створювати слайд-колоди, які витягують живі дані безпосередньо з Elasticsearch[16].

2.8 Розгортання продукту в хмарній інфраструктурі

Хмарна інфраструктура стосується апаратних та програмних компонентів, таких як сервери, сховище даних, мережеве та програмне забезпечення для віртуалізації, які необхідні для підтримки вимог обчислення моделі хмарних обчислень. Зазвичай, такі моделі складаються з багатьох інфраструктурних шарів та залежать від обладнання, мережі та програмного забезпечення, як це зображено на рисунку 2.8.

Хмарна інфраструктура характеризується своєю масштабованістю та ізольованістю. Незалежні компоненти її можуть складатись з віртуальних машин або з програмних контейнерів.

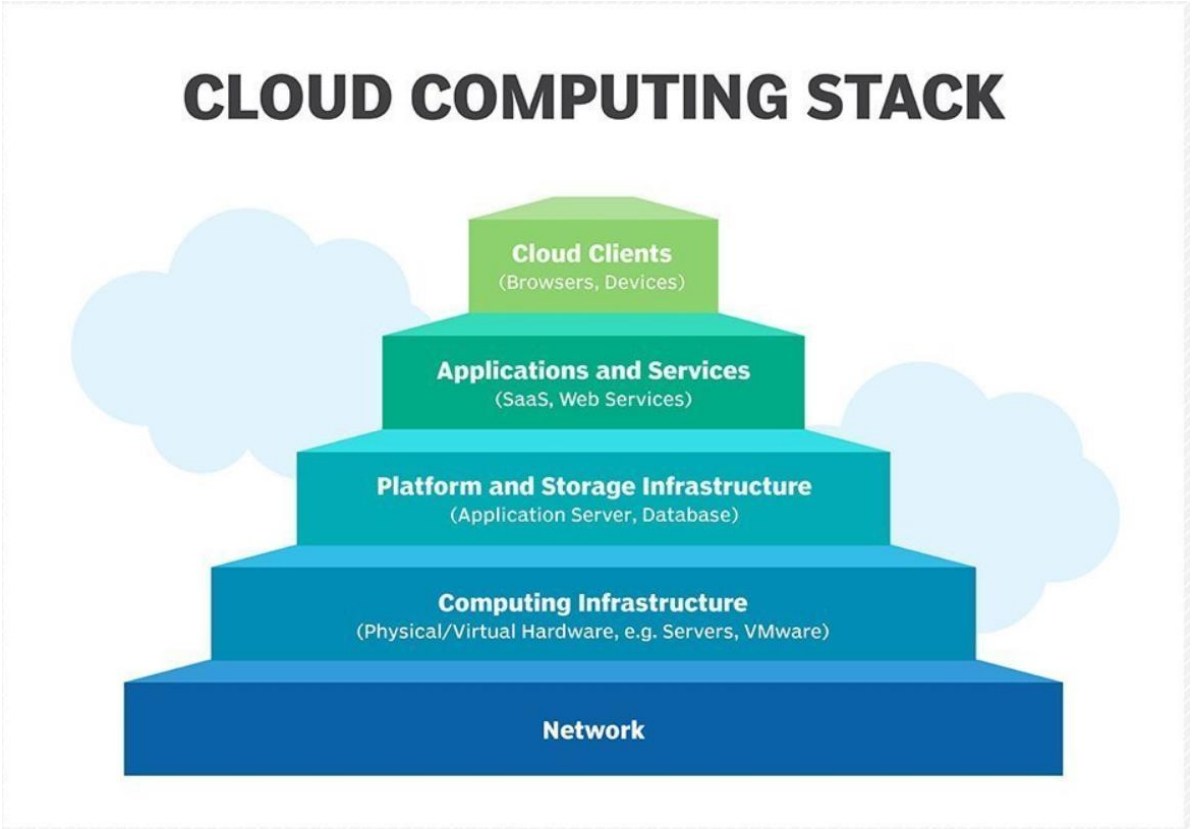

Рисунок 2.8 – Ієрархія складових хмарної інфраструктури

Кожен сервіс системи, включаючи сховища даних та брокер повідомлень, запускається в Docker контейнерах.

Docker - це програмна платформа для швидкої розробки, тестування і розгортання додатків. Docker упаковує ПО в стандартизовані блоки, які називаються контейнерами[17]. Кожен контейнер включає все необхідне для роботи програми: бібліотеки, системні інструменти, код і середовище виконання. Завдяки Docker можна швидко розгортати і масштабувати додатки в будь-якому середовищі і зберігати впевненість у тому, що код буде працювати.

З розвитком контейнеризації у розробників з’явилась можливість приділяти увагу більше розробці нового бізнес фунціоналу замість хвилювання об інфраструктурі систем. Також контейнеризація дозволяє мати одночасно декілька варіантів однієї і тої системи, проте у різних версіях, що значно підвищує здатність системи до тестування.

Розвиток контейнеризації створив такий тип систем, як оркестратори, наприклад Kubernetes, що став еталоном у своїй сфері. Якщо сервіси, які

запускаються у контейнерах спроєктовані, як незалежні процеси та є ізольованими від інших, оркестратори можуть повністю керувати життєвим циклом таких сервісів. Керуючи ними, можливо досягти тонкого налаштування реагування на навантаження, балансування трафіку, моніторинг працездатності, реагування на помилки, що у сукупності дає можливість мати систему, яка відповідає на запити при будь-яких ситуаціях і є дуже стійкою до любих змін та проблем навіть, якщо вони критичні для того чи іншого сервісу.

Висновки до розділу

У даному розділі було проведення підбір та аналіз технологій для розробленої платформи стримінгу даних. Розділ містить опис переваг та недоліків цільових технологій, на основі яких було підібрано рішення саме для такого типу систем.

Для розробки усіх компонентів системи було вибрано .NET Core, так як платформа була спроєктована саме для хмарних обчислень та має чудовий потенціал до розвитку та масштабування. Використовуючи мову C# було створено клієнтські бібліотеки з програмним контрактом, який дозволяє використовувати фунціонал протоколів передачі даних.

Для асинхронного обміну повідомлень використовується брокер повідомлень Apache Kafka, що чудово підходить для поставлених цілей своєю масштабованістю та надійністю.

Для зберігання даних використовувалась NoSQL Apache Cassandra через свою високу здатність до масштабованості та сховище даних, яке тримає дані в пам’яті комп’ютера Redis, тому що він має велику швидкість доступу до даних, що грає велику роль при обробці великого обсягу даних.

Логування системи відбувається за допомогою вбудованих інструментів .NET платформи та зберігає журнали подій у Elasticsearch для подальшої візуалізації у сервісі Kibana.

3 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Діаграма прецедентів

Діаграму, котра описує які дії можуть виконувати елементи або користувачі системи, називають діаграмою прецедентів.

Така діаграма представляє собою граф, який представляє собою залежність між актором та його діями, як один до багатьох. В свою чергу кожен прецедент може мати інші прецеденти, які є детальною частиною в нього або є розширенням, пояснюючи деталі реалізації дії, тим самим можна покрити специфічні випадки на етапі проєктування. Також, важливо відмітити, що в даному типі діаграм не вказуються методи та способи реалізації тих чи інших дій.

Під час проєктування та розробки було створено діаграму даного типу, яка зображена на кресленика ІТ61.090БАК.004 Д1. Діаграма складається с двох частин у яких описуються діаграми акторів для джерела подій (рис 3.2) та слухача цих подій (рис 3.1). Роботу сервісу підписок було включено в частину роботи клієнта системи, а усі процеси та прецеденти хабу системи до категорії джерел.

Рисунок 3.1 – Діаграма прецедентів клієнта

Клієнт системи може виконувати дії з чотирьох груп:

— керувати підписками на трансляції подій;

— дізнаватись інформацію про можливі трансляції;

— налаштовувати правила роботи з фільтрацією для пониження кількості нерелевантної інформації;

— використовувати логи даних для подальшої візуалізації з метою отримання статистики.

Кернування підписок включає весь фунціонал та спектр дій, який може стосуватись підписок в даному випадку, а саме: отримання ключа після авторизації у системі, як слухач, використання цього ключа в подальшому для реєстрації підписок на відповідні події, видаляти підписки від подій, змінювати їх. Важливим моментом є вибір типу підключення серед можливих:

— push формат повідомлення;

— дуплексний канал зв’язку через сокети за UDP протоколом;

— використання інтерфейсу консюмера Apache Kafka.

Вибраний тип підписки буде впливати на подальше навантаження на систему та можливість розширюваності. Основним з параметрів підписки є вказання IP адресу слухача. При зміні існуючих параметрів хаб почне транслювати на потрібну адресу щойно отримає повідомлення від сервісу підписок.

Після створення ключа слухача клієнт зможе запросити у сервера підписок список усіх подій на сьогоднішній день або вибрану дату, що потім дозволить оформити підписку з правилами фільтрації. Список буде повертати детальну інформацію про події, стан їх та можливі дані, які будуть з них транслюватись. Після отримання даних клієнт створює новий запит на підписку з вказанням ідентифікатору події та полями, в яких він зацікавлений.

Правила фільтрації в подальшому можна створювати додаткові, змінювати існуючі та видаляти, тим самим клієнт має змогу специфічно та тонко налаштовувати систему до поставлених цілей. Також присутня можливість автопідписки на усі події з найбільшою можливою кількістю варіантів даних для

самостійної обробки в подальшому засобами клієнтського програмного забезпечення.

Також для клієнта системи є можливість підключення до системи логування, для подальшої візуалізації даних та стану системи. В такому випадку дані доступні лише у режимі перегляду та мають повний набір полів, тому клієнт не може впливати на роботу системи візуалізації та налаштовувати кастомізацію.

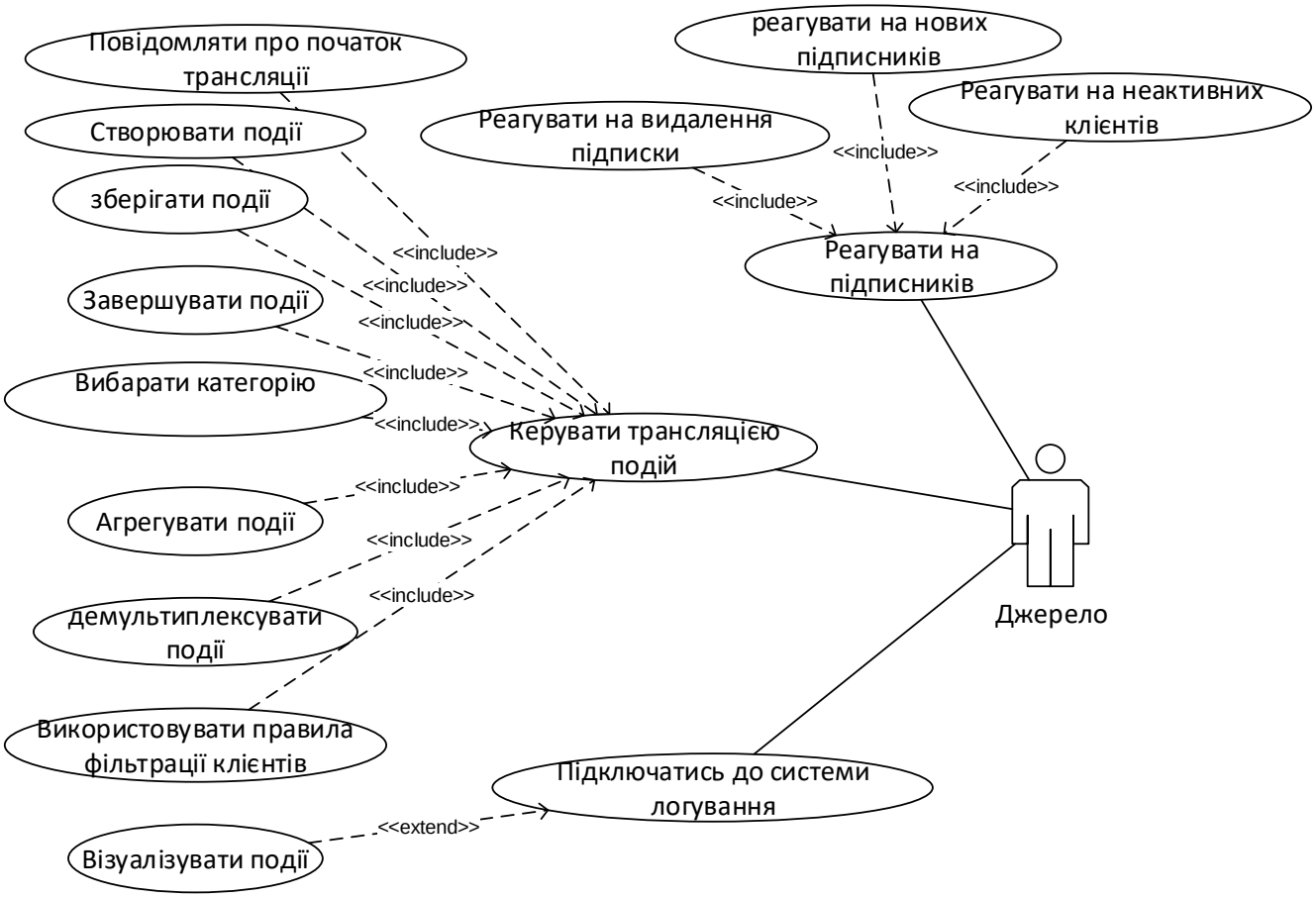

Рисунок 3.2 - Діаграма прецедентів джерела подій

На рисунку 3.1.2 зображено детальну діаграму прецедентів джерела подій. Функціонал хабу було включено в фунціонал актору джерела, так як воно виступає продюсером даних та опрацьовує кластер роботи для підготовки даних для клієнтів системи.

Джерело даних має 3 групи прецедентів:

— робота з логуванням та візуалізацією;

— трансляція даних та все, що до неї належить;

— реагування та зміни у системі підписок.

Відносно логування та візуалізації фунціонал такий самий, як і для клієнта проте доступне додаткове налаштування та більш гнучке логування.

Широкий спектр дій по відношенню до створення подій включається в модуль зв'язаний з трансляцією. Засобами брокеру повідомлень Apache Kafka джерело має змогу гнучко налаштовувати трансляцію даних до хабу з подальшою розсилкою до клієнтів. Налаштовуючи брокер повідомлень, джерело має змогу впливали на екземпляри хабу в кластері для розподілення навантаження за різними розділами, топіками та партиціями. Представляється асинхронний обмін повідомленнями, що є частиною трансляції даних. Для покращення пропускної можливості системи використовується агрегація та мультиплексування даних та подій у одне повідомлення.

Джерело повідомляє про деталі події та її статус перед початком трансляції, що в подальшому дозволяє додавати, видаляти та змінювати дані. Також використовуються правила описані клієнтами системи для фільтрації та модифікації даних під час їх обробки. Дані зберігаються у форматі, як вони були створені, проте транслюються у форматі, який потрібен клієнтам.

Важливою частиною є робота з підписками та реагування на їх зміни, так як від цього хаб дізнається про клієнтів системи, які правила використовувати та куди транслювати дані. Також потрібно мати функціональність реагування на збій клієнта або його відсутню активність для зменшення непотрібного навантаження на систему та нотифікувати клієнта про проблеми.

3.2 Діаграма розгортання

На діаграмі розгортання можна побачити, як частини системи взаємодіють один з одним. З діаграми, зображеної на кресленику ІТ61.090БАК.004 Д2, можна побачити, що серцем системи виступає група сервісів, які утворюють хаб, через які

проходять усі дані під час калькуляції, до яких потім застосовуються бізнес правила, то далі транслюються до сервісів-слухачів.

Компоненти діаграми складаються з модулів та опису, які елементи системи повинні бути включені до цього модулю. Як видно з рисунку 3.3, модуль хабу, котрий виступає шиною, вимагає середовище запуску .NET Core, котре повинне бути розгорнуте на машині запуску хабу, а також налаштоване сховище Redis, так як це є необхідною частиною для роботи хабу.

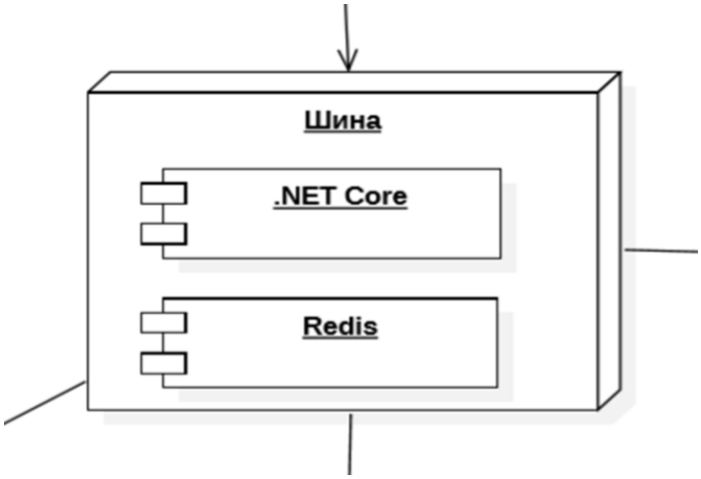

Рисунок 3.3 – Структура модуля хабу та його компонент

Також на діаграмі видно, що клієнти системи працюють тільки з сервісом підписок, що дозволить хабу дізнаватись інформацію про них та транслювати дані, тим самим знижуючи зв’язність компонентів, що чудово підходить до горизонтального масштабування у хмарному середовищі.

Сервіси та клієнтські бібліотеки написані на мові програмування C# та використовують віртуальну машину .NET Core, тому на схемі їх виділено, як компоненти. Це додає свої обмеження зі сторони вибору провайдера або інструментів при налаштуванні цільової платформи при розгортанні.

Зі схеми видно, що сховища логів та сервера візуалізації використовують готові інструменти Elastic і Kibana відповідно, що позначає, що зі сторони розробників не потрібно писати додаткового програмного забезпечення, тільки

шар програмного інтерфейсу для зв’язку з програмною частиною існуючих компонентів.

Зі схем даного типу можна виділити, які модулі повинні використовуватись та що повинно бути налаштовано для розгортання системи, тим самим полегшуючи роботу розробників та адміністраторів баз даних або серверів.

3.3 Структура проєкту

Структура проєкту використовується для опису частин системи та фунціоналу. Такі частини позначаються блоками та, створюючи зв’язком між ними, можна отримати загальну архітектуру системи та як компоненти взаємодіють один з одним.

Такий тип систем є основним при проєктуванні архітектури програмного забезпечення.

На кресленику ІТ61.090БАК.004 Д3 зображено структуру проєкту до розробленої платформи. Діаграму можна поділити на 4 частин:

— частина клієнта;

— шар доступу до даних;

— група консюмерів;

— група продюсерів.

На рисунку 3.4 зображено клієнтську частину, яка включає в себе сервер підписок, клієнтську бібліотека та сервер візуалізації Kibana.

Даний тип діаграм описує також методи взаємодії між компонентами з описом технічних деталей.

Також такий тип системи вказує як компоненти зв’язані між собою та який тип взаємодії.

Даний клас діаграм може бути схожим з ідеєю до діаграми прецедентів, проте з технічної частини з описом модулів.

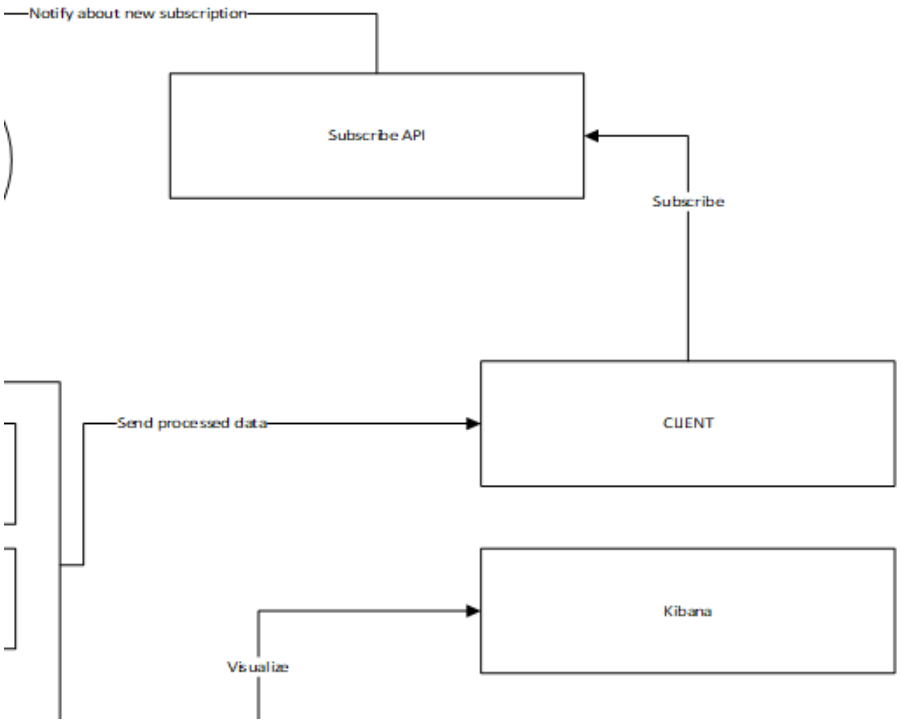

Рисунок 3.4 – Клієнтська частина системи

Сервіси, які слугують джерелами даних про події, групуються в групу продюсерів. Така групу джерел зображена на рисунку 3.5. Група може мати декілька версій одного і того самого джерела та мати декілька версій. На основі цих сервісів можна налаштовувати балансування між ними та впроваджувати новий фунціонал. Кожен з сервісів використовує програмну бібліотеку для роботи з фунціоналом продюсер брокеру повідомлень Apache Kafka. До цієї групи входять сервіси, котрі генерують дані про футбольні події, про події перебігу кінних перегонів або емуляції даних з датчиків, таких як датчик температури або вологи.

До групи джерел відносяться сервіси, котрі генерують дані про такі події, як:

— перебіг подій під час футбольного матчу;
— зміна температури або вологи у приміщені;
— зміна позицій гонщиків під час кінних перегонів.

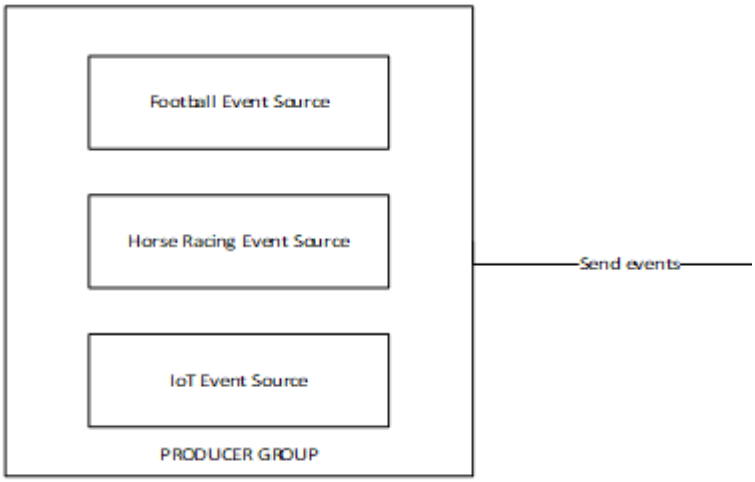

Рисунок 3.5 – Група продюсерів

Група консюмерів зображена на рисунку 3.6. Група складається з декількох екземплярів хабу для гнучкого налаштування при різному навантажені. Так, як і з групою продюсерів, група консюмерів використовує програмну бібліотеку до доступу фунціоналу костюмера Apache Kafka. Виконуючи бізнес дії та проводячи транзакції, дані передаються до клієнтів через хаб. Група продюсерів працює з шаром доступу до даних для зберігання результатів обчислень та записує логи у журнали Elasticsearch.

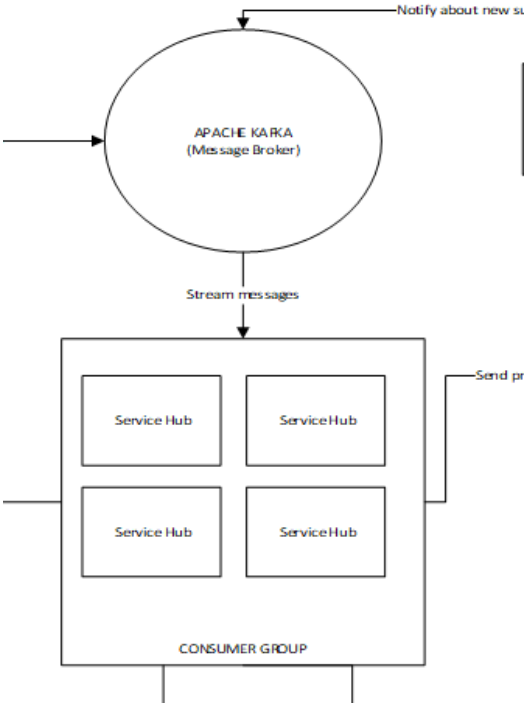

Рисунок 3.6 – Група консюмерів

Шар доступу до даних представлений на рисунку 3.7 зображає елементи системи, які відповідають за програмний інтерфейс підключення до публічного програмного інтерфейсу систем управління баз даних, таких як Apache Cassandra. Також хаб взаємодіє з Redis, і при масштабуванні він є основним тимчасовим сховищем даних, в якому поточні дані зберігаються для обміну між всіма сервісами допоки вони не будуть збережені на диск. Також цей шар має доступ до сховища логів Elasticsearch і дозволяє інтегрувати цей фунціонал як для клієнтів, так і для джерел.

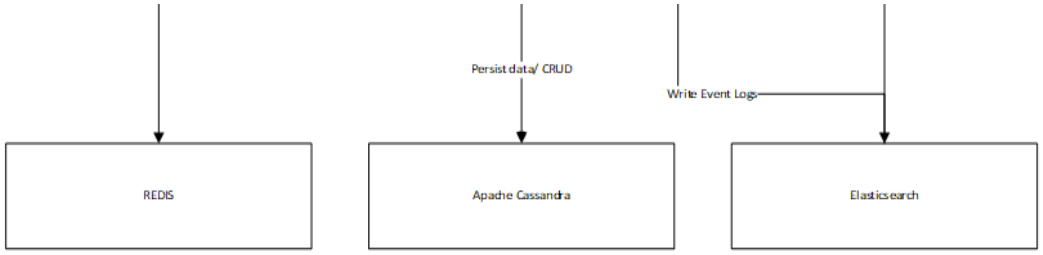

Рисунок 3.7 – Шар доступу до даних

3.4 Діаграма компонентів

Діаграма компонентів описує конкретні частини модулів та компонентів. Модуль комунікації використовується в усіх частинах системи, в котрих відбувається комунікація з іншими частинами. На рисунку 3.8 зображено детальні компоненти цього модуля та назви інтерфейсів, які вони викривають, як публічні програмні інтерфейси для подальшого використання.

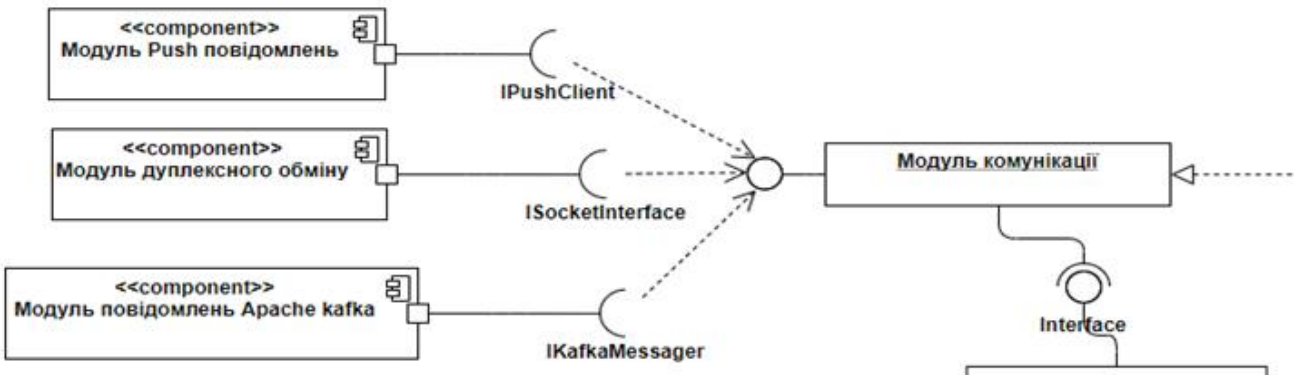

Рисунок 3.8 – Компоненти модуля комунікації

Через шаблон фасад, який інкапсулює деталі реалізації, користувачі цього модуля можуть описувати, що вони хочуть використовувати та деталі про адресу сервісів.

Модуль використовується як джерелами для трансляції даних, так і клієнтами. Також цей модуль використовуються у хабі для передачі від джерела до клієнта та у сервері підписок для нотифікації хабу про зміни.

Рисунок 3.9 зображає, як програмні модулі взаємодіють один з одним з описом інтерфейсу зв'язку.

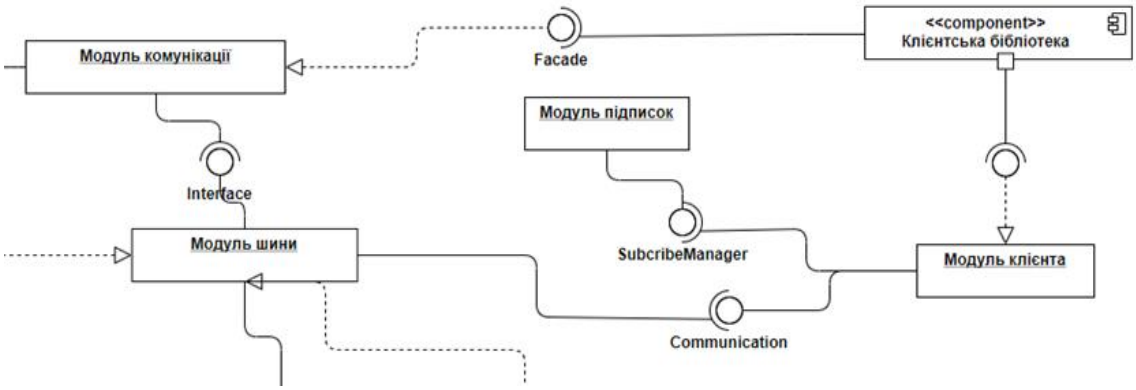

Рисунок 3.9 – Взаємодія модулів публікації-підписок

З рисунку видно, що модуль клієнта має власну клієнтську бібліотеку, котра використовує фасад інтерфейс з модуля комунікації для обміну через програмний інтерфейс комунікації.

API комунікації використовується у модулі хабу та має власного менеджера з конкретними налаштуваннями та обгортками навколо існуючого фунціоналу, специфічного для хабу та розподілених систем.

На рисунку 3.10 описано, що шар доступу до даних має свої публічні API, з котрими в подальшому взаємодіє хаб.

Кожна з реалізацій інтерфейсу являє собою обгортками над існуючими бібліотеками доступу до конкретний програмний реалізацій.

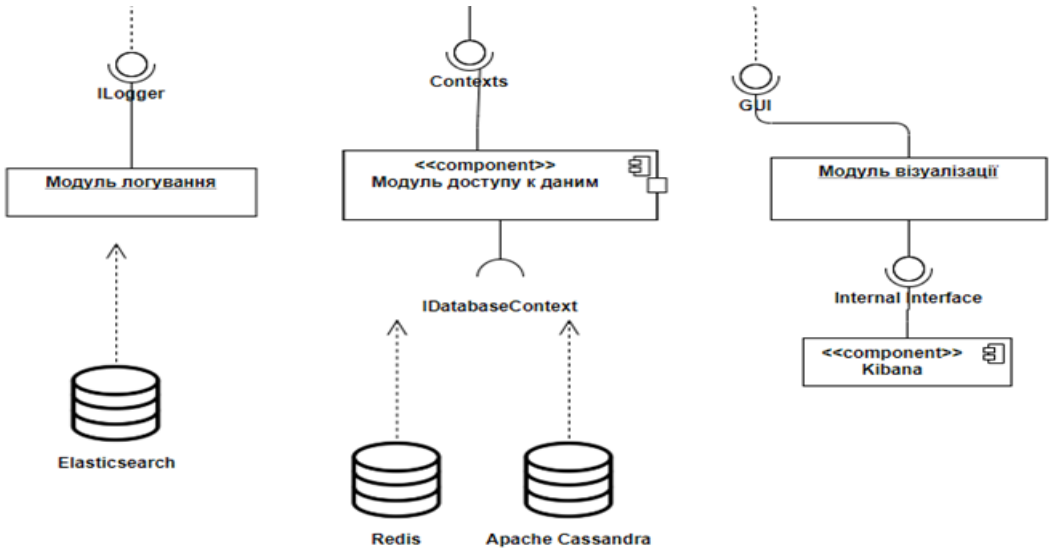

Рисунок 3.10 – Модулі доступу до даних

Компонент доступу до даних включає в себе реалізації та програмні менеджери у вигляді контекстів для роботи з Apache Cassandra та можливостями сховища Redis.

Також до цього шару відноситься інтерфейс для роботи з логами, котрий має налаштування для роботи з процесами пошуку та запису Elasticsearch.

Модуль візуалізації використовує компонент Kibana для візуалізації даних, який отримує від модуля логування в результаті роботи сервісів хабу.

Висновки до розділу

У ході даного розділу було проведено аналіз архітектурних компонентів системи, та прийняті рішення було представлено на таких типах діаграм, як:

— прецедентів;
— розгортання;
— структура проєкту;
— компонентів.

Кожен тип діаграми описує розроблену систему з різних аспектів, що дозволить виявити проблеми ще на етапі проєктування. Такий підхід використовується архітекторами програмного забезпечення для схематичного

пояснення для команди розробників, про деталі реалізації та прогнозовану поведінку системи.

Діаграма прецедентів, котра була створена під час роботи, описує можливу множину прецедентів та конкретних дій, які можуть виконувати актори системи. Схема розглядає поведінку зі сторони джерела даних про події та зі сторони клієнта цих даних з можливістю гнучкого налаштування аспектів фільтрації.

Діаграма розгортання, котра була створена під час роботи, описує взаємодію компонентів та основні складові технологічного стеку, які потрібно налаштувати та розгорнути для подальшої роботи системи.

Структурна діаграма описує загальну архітектуру системи, як окремі програмні компоненти, які взаємодіють один з одним. Ця діаграма представляє основну ідею та елементи розробленої системи, що слугує ілюстрацією для розробника.

Діаграма компонентів, розроблена з описом конкретних елементів системи, які повинні бути спроєктовані для кожного програмного модуля, що є більш конкретною реалізацією механізму роботи елементів системи.

4 СТРУКТУРА ДАНИХ І РЕСУРСІВ ПРОГРАМИ

Програмна реалізація застосунку складається з такого набору бібліотек та програмних реалізацій:

— програмні модулі джерел даних;

— консольний застосунок хабу системи;

— веб-сервер підписок;

— програмний модуль клієнтського фунціоналу;

— розшарені модулі, які являють собою ядро системи, які відповідають за роботу з доступом до даних, логування та іншими.

Усі частини платформи написані на мові програмування C#, котрі запускаються на віртуальній машині .NET Core. Для середовища виконання платформи було використано останню стабільну версію .NET Core 3.1. Цільовою системою для розгортання системи слугують Linux машини. Додатки розроблялись на MacOS. Програмні компоненти, включаючи бази даних та системи логування, розгортаються у контейнерах Docker.

4.1 Ядро системи

Ядром системи є набір програмних інтерфейсів та бібліотек, які використовуються повністю або частково кожним компонентом системи. Ядро системи розбито на два програмні модуля EventStreamingService.Core та EventStreamingService.DAL, які відповідають за інфраструктурну частину систему та доступ до шару даних відповідно.

Модуль EventStreamingService.Core включає в себе набір програмних інтерфейсів до інфраструктурних компонентів системи:

— логування;

— обміну повідомлень;

— конфігурації;

— моделі предметної області платформи;

— меппінг даних;

Модуль EventStreamingService.DAL включає в себе набір програмних інтерфейсів для роботи з шаром доступу до даних:

— доступ до взаємодії з Apache Cassandra;

— доступ до взаємодії з Redis;

— доступ до взаємодії з Elasticsearch.

Усі компоненти, які відносяться до логування знаходяться за простором імен EventStreamingService.Core.Logging. Логування забезпечується реалізацією інтерфейсу ILogger та впровадженням його в модулі, де необхідно працювати з логуванням, наприклад як: в обробниках хабу, менеджерах джерел або клієнтській бібліотеці.

За впровадження реалізації та вибір її типу використовується компонент LoggerManager, котрий в свою чергу бере інформацію з файлів конфігурації про тип логування. Типи логування можуть бути:

— в консольний вивід програми;

— в журнал подій операційної системи;

— в текстовий файл;

— в сховище даних Elasticsearch.

ElasticLogger реалізує ILogger інтерфейс для впровадження логування різного типу критичності, як: інформація, попередження, помилка, та записує усю інформацію в відповідну колекцію у Elasticsearch для подальшого гнучкого пошуку через Kibana або через шар доступу до даних. Усі типи логування базуються на використані безкоштовної відкритої бібліотеки Serilog, котра дає можливість використовувати наведені вище типи логування.

Конфігурація сервісів є дуже важливим аспектом для хмарних застосунків. За конфігурацію відповідає компонент ConfigManager. Це глобальний модуль, до котрого має доступ кожний компонент систем. Конфігураційні параметри він отримує з переданих параметрів середовища від Docker контейнера або цільової операційної системи, на якій розгортається той чи інший сервіс.

Простір імен EventStreamingService.Core.Domain включає в себе моделі даних з описом характеристик та структури. Ці моделі стосуються таких сутностей платформи:

— подія;
— футбольна подія;
— подія кінних скачок;
— подія зчитування датчику;
— розклад подій;
— модель логування;
— опис підписника системи.

Конкретні події наслідують характеристики базової моделі подій з описом основних характеристик, таких як час події, тип даних, короткий опис та ідентифікатор.

Інші моделі описують специфічні характеристики тої чи іншої бізнес сутності або являють собою агрегацією інших моделей, таких як події у модель розкладу подій по часу.

Меппінг моделей вважається однією з кращих практик розробки програмного забезпечення, що дозволяє розділити представлення даних, які отримує система, та дані, з якими працює система.

Меппінг профілі описують, як дані з джерел даних та серверу підписок повинні бути представленні для уніфікованої роботи з ними на рівні хабу, представлених для клієнтів та логування.

Важливим для системи є компонент, котрий знаходиться у просторі імен EventStreamingService.Core.Messaging та відповідає за комунікацію між сервісами. Комунікація між сервісами побудована на тому, що один сервіс виступає консюмером, інший – продюсером. Наприклад, джерело подій виступає продюсером даних, а хаб системи – консюмером. Для брокера повідомлень використовується Apache Kafka, тому простір імен має два компоненти KafkaConsumer та KafkaProducer для цих цілей на основі безкоштовної бібліотеки Confuent.Kafka, котра написана під середовище виконання .NET Core.

До ядра системи відноситься також шар доступу до баз даних. Шар включає в себе об’єкти, які називаються контекстами, котрі відтворюють сесію роботи з тою чи іншою базою даних при ініціалізації. Також можуть впливати на підключення та транзакції під час цих сесій, що дозволяє розробнику використовувати зручний програмний інтерфейс. Цей модуль складається з трьох просторів імен Cassandra, ELK та Redis.

Для роботи з Redis використовується RedisContext, котрий працює з сервером та виконує роботу зі зчитуванням та записом даних. Для роботи використовується бібліотека StackExchange.Redis. Швидкість для роботи з розподіленим кешем є найбільш важливим критерієм, тому ця бібліотека надає найбільш швидкий доступ до роботи з цими даними. Для представлення колекцій використовуються масиви з моделями, представленими у ядрі платформи, та описують події, що нещодавно відбулись для можливості контролювати збій у системі, дублікати даних та напряму під час розробки, аналізувати та агрегувати дані. Також використовується колекція підписників для моніторингу активних підключень та утримання інформації про публічні адреси, куди потрібно надіслати дані.

Для роботи з Elasticsearch використовується ELKConxtext. ELK абревіатура означає Elasticsearch, Logstash, Kibana, що показує, що цей контекст працює як з самими Elasticsearch, так може в подальшому бути розширений для роботи з сервісом збирання та трансформації даних Logstash, та для роботи з Kibana програмно з коду. Для роботи з контекстом було вибрано високорівневу бібліотеку NEST, так як вона має типізовані конструкції мови програмування для роботи з базою, що підвищує її надійність на рівні типів системи. Контекст під час сесії працює лише з колекцією логів, що використовує модель Log з ядра системи, та може поширюватись більш складною структурую даних, з якими вона працює для подальшого налаштування відображення на сервері візуалізації Kibana.

Для роботи з Apache Cassandra використовується CassandraContext побудований, як обгортка навколо низькорівневого драйвера для роботи з базою на мові C# - CassandraCsharpDriver.

Колекції записів у сховищі даних представляються у вигляді набору колонок, тому, як правило, дані представляються у ненормалізованому стані для швидкості доступу до них. Контекст використовує мову схожу на SQL – CQL (Cassandra Query Language), тим самим забезпечуючи роботу зі зчитуванням, редагуванням, створенням та видаленням даних. Також контекст має можливість налаштування роботи з відповідним кластером бази даних.

Колекція підписників має такий набір полів при зберіганні в колекції:

— id - унікальний ідентифікатор у форматі UUID;
— ip - адрес серверу підписника;
— name - назва підписника;
— details - інформація про підписника;
— type - тип підключення;
— latency - значення можливої затримки;
— time_zone - часова зона;
— created_at - дата створення підписки;
— last_modified - дата останньої зміни;
— subscribed_types - колекція типів подій, на котрі підписаний клієнт;
— rules - правила фільтрації полів.

4.2 Джерела даних

Джерела даних це група сервісів-продюсерів, котрі поділяються на три типи:

— джерело даних з датчиків;
— джерело подій футбольного матчу;
— джерело перебігу подій кінних гонок.

Усі вище перераховані типи реалізацій знаходяться у просторі імен EventStreamingService.Source. Кожен тип використовує свій компонент-менеджер, котрий відповідає за генерацію та цілісність даних про події, та може виступати, як окремим сервісом, так і частиною існуючого сервісу.

За допомогою вбудованої бібліотеки класів в платформу .NET Core використовуються засоби для контролювання подій по часу та паралельних обчислень.

Якщо декілька екземплярів сервісів запущено, то кожен сервіс генерує події ізольовано від інших сервісів, тому кожен сервіс генерує події тільки до відповідної категорії або відповідного спортивного заходу окремо.

Для передачі даних з групи продюсерів до групи консюмерів у вигляді хабу використовується модуль KafkaProducer з ядра системи. Обробка та посилання цих даних відбувається в асинхронному та паралельному вигляді, що породжує велику кількість даних, що транслюються через Apache Kafka. Для брокеру повідомлень таке навантаження є припустимим, проте для клієнтів та обробників у вигляді хабу – не завжди. Для запобігання перебільшення пропускної здатності системи також використовується таймер на певну затримку на стороні джерел даних. Інформацію про затримки також можна отримати у сервера підписок під час перегляду списку доступних подій на певну дату.

4.3 Клієнтська бібліотека

Клієнтська бібліотека використовується як клієнтами, так і хабом, так як включає в себе необхідний фунціонал для обміну повідомленнями у зв’язку продюсер-консюмер між хабом та клієнтами. Компоненти клієнтської бібліотеки знаходяться у просторі імен EventStreamingService.Client та складаються з трьох програмних інтерфейсів:

— споживання даних від джерела;

— формат push комунікації;

— дуплексний формат комунікації через сокети.

Кожен з вище перечисленних API працює з відповідними типом підключення та надає змогу переслати дані завдяки ньому. Ці інтерфейси використовуються хабом для подальшої передачі даних до клієнтів після калькуляції, та у ситуації коли тип підключення являється дуплексним, то

використовується ця бібліотека стороні клієнта для можливості обміну повідомлень у зворотному напрямку.

4.4 Хаб

Хаб системи виступає групою консюмерів відповідно до групи продюсерів джерел. Хаб є основною частиною системи, яка з'єднує процеси створення та отримання даних. Хаб використовує клієнтську бібліотеку для надсилання даних відповідно до підписок за різними типами підключень. Програмний інтерфейс хабу знаходиться у просторі імен EventStreamingService.Bus. У просторі імен знаходяться програмні модулі, котрі працюють окремо з кожним типом джерел. До роботи з джерелами входять агрегації та фільтрації, якими займаються відповідні програмні модулі. На рівні хаба відбуваються агрегація та меппінг даних у формат логів для подальшої обробки у Elasticsearch та відображання на Kibana. Хаб слухає брокер повідомлень на наявність подій зі сторони сервісу підписок та має відповідний компонент-менеджер, який реагує на зміни у списку підписників та зміни підключення.

Хаб - єдиний сервіс, котрий напряму працює з шаром доступу до даних. Для підвищення пропускної здібності та швидкості обробки даних використовується Redis. Це є основне сховище даних під час роботи системи, проте дані існують в пам'яті і можуть бути втрачені при збоях хабу, тому у відкладеній манері дані зберігаються у Cassandra, що створює неповну цілісність даних під час обробки, проте дані будуть у потрібній формі у постійному місці зберігання, коли обробки відповідної групи подій закінчиться.

4.5 Сервер підписок

Сервер підписок являє собою REST WebAPI та знаходиться в просторі імен EventStreamingService.SubscribeApi. Веб-сервер реагує на HTTP запити від потенційних клієнтів системи та повертає дані у форматі JSON.

Архітектура сервісу побудована за шаблоном CQRS (Command-Query Responsibility Segregation). Ідея такої архітектури полягає в розділені системи на дві частини. Одна відповідає за роботу з даними, тобто: видалення, створення редагування. Інша за запити, тим самим дозволяє зменшити зв'язність між представленням даних та роботи з ними. Цей шаблон підходить, якщо потрібно створювати складні запити до бази даних та мати велику швидкість доступу до даних тому, що надається можливість контролювати, з якими даними та базами працює система. Для реалізації цього шаблону було використано MediatoR бібліотеку. Тому проєкт складається з модулів Queries, Commands, Handlers для запитів, команд та обробників відповідно.

Система побудована на основі платформи ASP.NET Core та використовує усі можливості бібліотек платформи для запуску такого веб-серверу.

Сервер відповідає за такі процеси у системі, як:

— робота з підписниками;

— перегляд подій за датами;

— робота з правилами підписок та фільтрації.

Для роботи з переліченими вище процесами використовується ScheduleHandler та SubscriberHandler програмні компоненти.

4.6 Розгортання у хмарній інфраструктурі

Найбільш сучасною тенденцію до розгортання застосунків у хмарному середовищі є контейнеризація. Усі сервіси системи, включаючи сервер візуалізації та сервери баз даних, розгорнуті в контейнерах Docker. Для запуска усіх сервісів використовується Docker-compose, у котрому описуються усі необхідні вимоги та налаштування до контейнерів у форматі YAML файлу. Для зберігання даних з контейнера налаштовано volume. Volume - це привязка дискового простору контейнеру до простору на машині, де запущено цей контейнер. Для додаткового налаштування Redis та Elasticsearch в volume можна передати файли конфігурації.

Висновки до розділу

У даному розділі було розглянуто головні складові розробленого продукту та деталі їх реалізації з описом використаних бібліотек, створених модулів та специфічні моменти реалізації.

Головними частинами системи є:

— ядро системи;

— шар доступу до даних;

— група продюсерів, а саме джерел даних;

— група консюмерів, а саме хаб – головна частина обробки системи;

— REST веб-сервер підписок;

— програмний модуль клієнтського фунціоналу.

Перераховані вище компоненти складаються з:

— веб-серверів;

— консольних застосунків;

— воркерів та системних сервісів;

— програмних бібліотек.

Під час реалізації було використано діаграми та схеми з розділу про опис програмного забезпечення.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Під час розробки важливо притримуватись поставленим вимогам, так як в них описують специфічні проблеми, котрі важливі для бізнесу. Для тестування відповідності функціональних та нефункціональних вимог, поставлених раніше використовуються такі підходи:

— мануальне тестування;

— інтеграційне тестування;

— тестування навантаженням.

Мануальне тестування, в контексті систем стримінгу даних має на увазі, що потрібно створити слухача та підписатись на отримання подій, тим самим зрозумівши, чи працює логіка по стримінгу даних від джерел до клієнтів.

Для тестування отримання даних від джерел даних про футбольні події було створено веб-сервер, котрий буде утримувати push-повідомлення від хабу платформи. Для візуалізації цілісності даних, було накладено отримані дані на відеозапис футбольного матчу, як це зображено на рисунку 5.1.

Рисунок 5.1 – Результат роботи з джерелом футбольних подій

Для роботи з джерелом даних про події перебігу кінного забігу використовується підключення через сокети для виводу контенту в реальному часі. Такі дані накладуються на відеозапис та відображають позиції спортсменів за кольором їх одягу. Результат тестування зображено на рисунку 5.2.

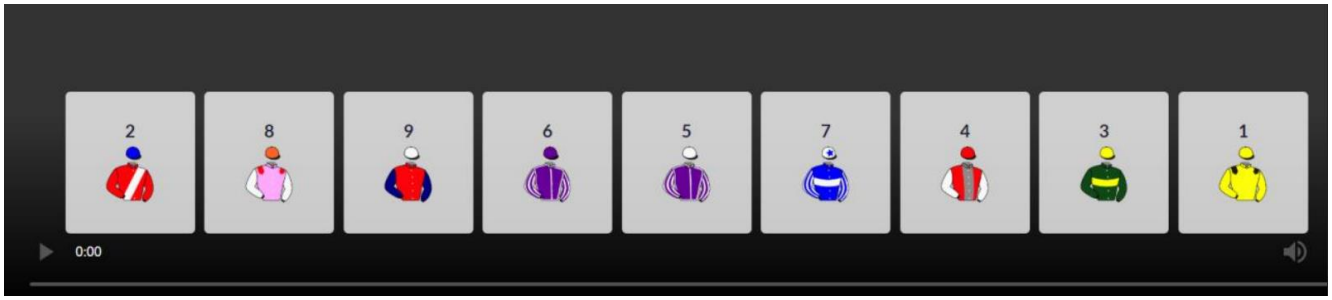

Рисунок 5.2 – Результат роботи з джерелом кінних подій

Для тестування роботи генерації даних від сервісу датчику температур використовується підписка на роботу з сервером візуалізації, тим самим надається можливість перевірити працездібність системи візуалізації та вживання даних від одного із джерел даних. На рисунку 5.3 зображено результат генерації даних на тиждень у вигляді графіку зі зміною значень температури.

Рисунок 5.3 – Результат роботи з джерелом генерації температури

Сервер візуалізації надає можливість детально масштабувати дані на графіку, тим самим можна отримати дані за певний період часу. На рисунку 5.4 зображено графік зміни температури протягом дня.

Рисунок 5.4 – Зміна подій зчитаних з датчика температури протягом дня

Інтеграційне тестування проводиться методом отримання метрик про роботу частин системи та отримання результату при мануальному тестуванні. Якщо існує збій у системі, то це не дозволить отримати повний набір даних до відповідного типу подій. Результат роботи системи видно на рисунку 5.5, на якому зображено, що для 1000 подій 71% транзакцій мають підтвердження інформаційного характеру, 15 – були оброблені успішно, проте існують деякі неточності та можливі затримки, та 14 відсотків про те, що дані є невалідними або було отримано збій при роботі з цими даними.

Візуалізуючи цю статистику, стало можливим протестувати сервер візуалізації для іншого набору даних та типу діаграми.

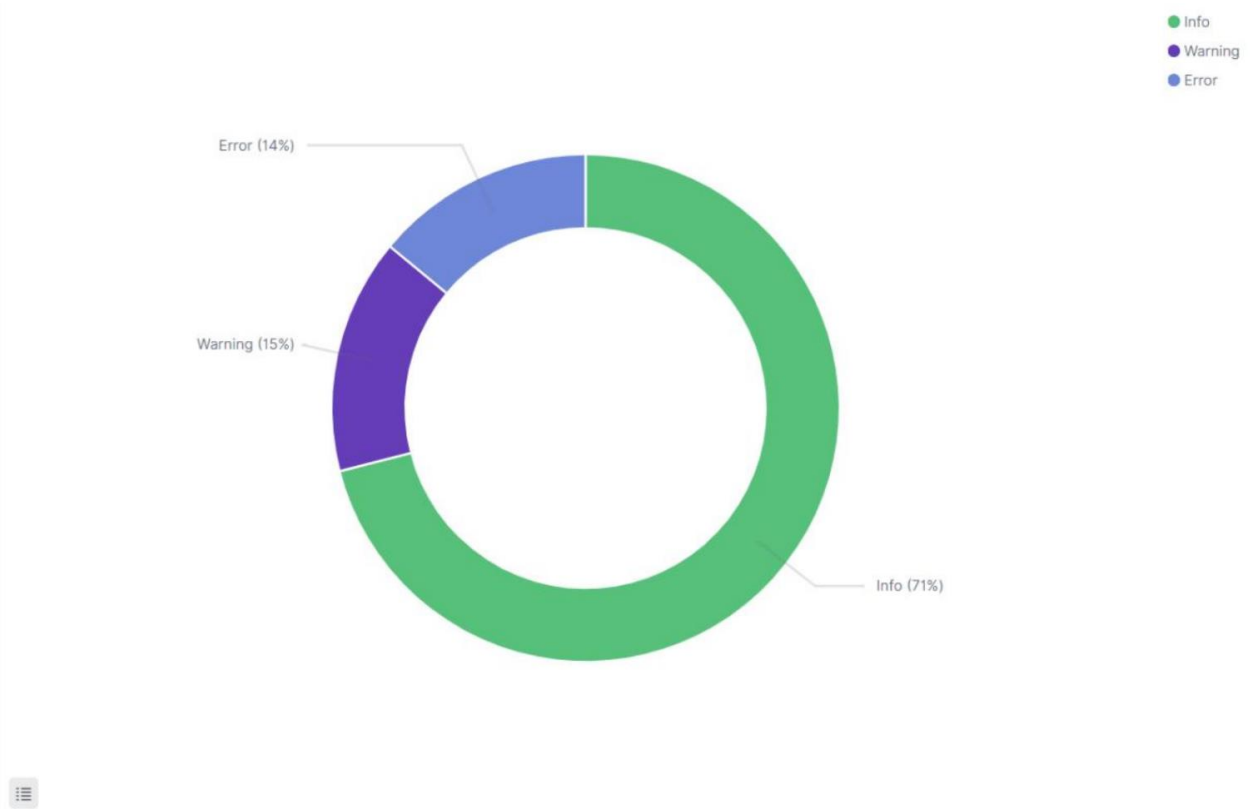

Рисунок 5.5 – Кільцева діаграма статистики результатів обробки системи

Для інтеграційних тестів важливо, щоб інфраструктурні компоненти також працювали відповідно, тому потрібно протестувати, що розгортання їх у контейнерах також працює справно. На рисунку 5.6 видно, що контейнери Redis, Cassandra та ELK Stack працюють справно.

```
Bohdan@DESKTOP-O6GFSJK MINGW64 ~/source/repos/bogdaner2/EventStreamingService/docker (master)
$ docker ps -a
CONTAINER ID   IMAGE                                                 COMMAND                  CREATED          STATUS          PORTS                                                NAMES
f3d222af3b37   redis                                                 "docker-entrypoint.s…"   3 hours ago      Up 5 minutes    0.0.0.0:6379->6379/tcp                               redis
2749103f9210   docker.elastic.co/kibana/kibana:7.2.0                 "/usr/local/bin/kiba…"   3 hours ago      Up 5 minutes    0.0.0.0:5601->5601/tcp                               kibana
2ff12ded293d   docker.elastic.co/elasticsearch/elasticsearch:7.2.0   "/usr/local/bin/dock…"   3 hours ago      Up 5 minutes    0.0.0.0:9200->9200/tcp, 9300/tcp                     elasticsear
ch
2a3693c12b6d   cassandra                                             "docker-entrypoint.s…"   3 hours ago      Up 5 minutes    7000-7001/tcp, 7199/tcp, 9160/tcp, 0.0.0.0:9042->9042/tcp   cassandra
```

Рисунок 5.6 – Список запущених контейнерів з інфраструктурною частиною

Важливим для тестування платформ стримінгу даних є тестування навантаженням. Такий тип тестування має на увазі, що одночасно до системи інтенсивно звертаються протягом певного проміжку часу багато клієнтів. Для даного типу тестування було створено окремний консольний додаток, котрий використовує безкоштовну бібліотеку NBobmber для навантаження на систему у прописаних сценаріях. Додаток тестує час на генерацію даних на день та

надсилання цих даних до хабу з метою зберігання їх у Elasticsearch сховищі. Тестування проводилось за умови одночасної генерації даних від 50 джерел по 5 днів протягом однієї хвилини на операційній системі Windows. Як зображено на рисунку 5.7 у результаті тестування було створено 32000 записів у сховище даних.

Name	Health	Status	Primaries	Replicas	Docs count	Storage size
temperature	yellow	open	1	1	32140	14.3mb

Рисунок 4.7 – Результат створених даних під час тестування

Результати роботи тестування показали, що на генерацію та зберігання даних під великим навантаженням температурних показників на день в середньому потрібно від 1 до 3 секунд.

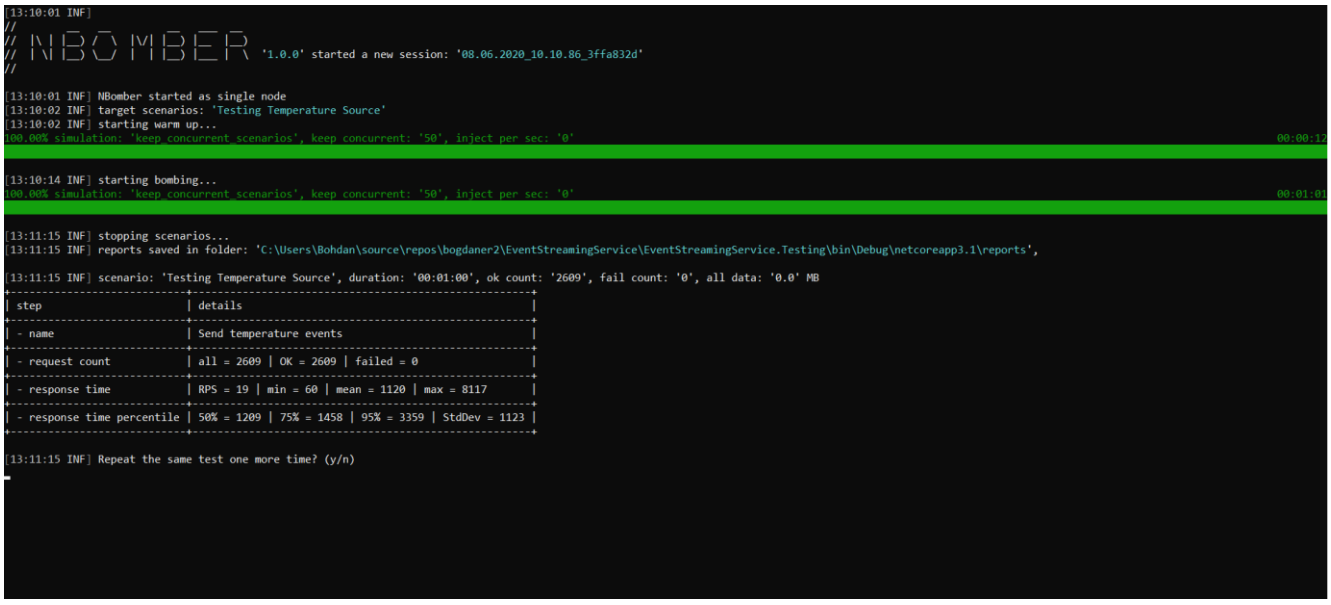

Рисунок 5.8 – Результат тестування навантаженням

Висновки до розділу

У ході даного розділу була протестована система на відповідність функціональним та нефункціональним вимогам за допомогою підходів тестувань: мануальних, інтеграційних та навантаженням.

ВИСНОВКИ

У ході розробки дипломного проєкту було розроблено систему для трансляції даних різного характеру від джерел до клієнтів за шаблоном публікація-підписка та розгонено її у хмарному середовищі за допомогою контейнеризації.

Під час виконання роботи було покращено навички проєктування та аналізу систем стримінгу даних у реальному часі. Були проаналізовані різновиди таких систем та їх спільні характеристики, переваги та недоліки.

Розробка даного продукту була розбита на декілька етапів. Для постановки задачі потрібно було проаналізувати аналоги системи та виокремити функціональні та нефункціональні вимоги до платформи. Після отримання вимог, було проведено аналіз підходів до вирішення поставлених проблем.

Для детального проєктування системи було створено діаграму діяльності, компонентів та розгортання, що було об’єднано в структурну діаграму з загальною архітектурою системи для розуміння шляхів подальшої підтримки та розширення системи під нові бізнес-правила.

У ході розробки було створено документацію проєкту з описом тонкостей розробленої платформи та можливими підходами до подальшої підтримки та розширення.

Під час розробки прототипу було ознайомлено з такими тенденціями сучасної розробки, як: NoSQL бази даних, системи асинхронного обміну повідомленнями та використання контейнеризації.

Головною перевагою розробленого додатку є те, що він включає в себе всі основні аспекти для трансляції, обробки та аналізу даних від джерела до клієнтів.

Отриманий продукт підходить для подальших наукових досліджень та розширення фунціоналу відповідно до нових сфер подій та бізнес-правил.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 4 Key components of a Streaming Data Architecture [Електронний ресурс]. – режим доступу: https://www.upsolver.com/blog/streaming-data-architecture-key-components

2. .NET Core [Електронний ресурс]. – режим доступу: https://ru.wikipedia.org/wiki/.NET_Core

3. Об’єктно орієнтоване програмування [Електронний ресурс]. – режим доступу: https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%27%D1%94%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D1%96%D1%94%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F

4. Actor Model [Електронний ресурс]. – режим доступу: https://en.wikipedia.org/wiki/Actor_model

5. Apache Kafka Official Documentation [Електронний ресурс]. – режим доступу: https://kafka.apache.org/

6. Microservices Guide [Електронний ресурс]. – режим доступу: https://martinfowler.com/microservices/

7. Service-oriented architecture [Електронний ресурс]. – режим доступу: https://en.wikipedia.org/wiki/Service-oriented_architecture

8. MonolythFirst [Електронний ресурс]. – режим доступу: https://martinfowler.com/bliki/MonolithFirst.html

9. Serverless Architecture [Електронний ресурс]. – режим доступу: https://martinfowler.com/articles/serverless.html

10. Reactive Manifesto [Електронний ресурс]. – режим доступу: https://www.reactivemanifesto.org/

11. Why Relational Databases are not the Cure-All [Електронний ресурс]. – режим доступу: https://phauer.com/2015/relational-databases-strength-weaknesses-mongodb

12. Apache Cassandra [Електронний ресурс]. – режим доступу: https://cassandra.apache.org/

13. Redis [Електронний ресурс]. – режим доступу: https://en.wikipedia.org/wiki/Redis

14. 3 methods for microservice communication [Електронний ресурс]. – режим доступу: https://blog.logrocket.com/methods-for-microservice-communication/

15. Elasticsearch [Електронний ресурс]. – режим доступу: https://en.wikipedia.org/wiki/Elasticsearch

16. Kibana [Електронний ресурс]. – режим доступу: https://en.wikipedia.org/wiki/Kibana

17. Productivity with Docker [Електронний ресурс]. – режим доступу: https://www.docker.com/use-cases/productivity