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

Інститут Прикладного Системного Аналізу
Кафедра Системного Проєктування

До захисту допущено:
Завідувач кафедри
Вадим МУХІН
«___»_____2023 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Інтелектуальні сервіс-орієнтовані розподілені обчислення»
спеціальності Комп'ютерні науки на тему:

«Застосування хмарної платформи Firebase для розробки та розгортання мікросервісів»

Виконав:
студент IV курсу, групи ДА-91
Нужний Микита Олександрович _____

Керівник:
доцент, к.т.н. Булах Богдан Вікторович _____

Консультант з нормоконтролю:
доцент, к.т.н. Кирюша Богдан Анатолійович_____

Рецензент:
Зав. каф. ОТ ФІОТ КПІ ім. Ігоря Сікорського, д.т.н., проф. Стіренко Сергій Григорович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Київ – 2023

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

Навчально-науковий інститут прикладного системного аналізу

Кафедра системного проєктування

Рівень вищої освіти: перший (бакалаврський)

Спеціальність: 122 «Комп'ютерні науки»

Освітньо-професійна програма:

«Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Вадим МУХІН

«___»_____2023 р.

ЗАВДАННЯ
на дипломну роботу студенту
Нужного Микити Олександровича

1. Тема роботи «Застосування хмарної платформи Firebase для розробки та розгортання мікросервісів», керівник роботи Булах Богдан Вікторович, доцент, затверджені наказом по університету від «_30_»____05_____ 2023 р. № 2065-с
2. Термін подання студентом роботи – «_15_»____06_____ 2023 р.
3. Вихідні дані до роботи
4. Зміст роботи:
    1. Вступ.
    2. Мікросервісна архітектура застосунків
    3. Хмарна платформа Firebase
    4. Використання Firebase Emulators для розробки мікросервісів

5. Реалізація використання інфраструктури Firebase для розгортки мікросервісів для агрегатора сайту медіаресурсів

6. Оцінка ефективності

7. Приклади використання

8. Висновки

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

1. Презентація до захисту роботи.

6. Дата видачі завдання «___»_____ 20__ р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	02.12.2022	
2	Дослідження наявних рішень для створення мікросервісної архітектури	01.04.2023	
3	Дослідження хмарної платформи Firebase	04.04.2023	
4	Аналіз можливостей щодо використання Firebase Emulators у мікросервісній архітектурі	05.04.2023	
5	Реалізація використання інфраструктури Firebase для розгортки мікросервісів	08.04.2023	
6	Бенчмаркінг та аналіз отриманих результатів	19.04.2023	
7	Розгляд поточних проєктів, які використовують Firebase як основне середовище розробки та розгортки	27.04.2023	
8	Порівняння та підбивання підсумків	05.04.2023	

Студент Нужний М. О.

Керівник Булах Б. В.

АНОТАЦІЯ

до дипломної роботи Нужного Микити Олександровича на тему «Застосування хмарної платформи Firebase для розробки та розгортання мікросервісів»

Структура дипломної роботи: Загальний обсяг пояснювальної записки: 115 сторінок, 19 рисунків, 7 таблиці, 15 посилань.

Актуальність теми визначається тим, що мікросервісна архітектура стає все популярнішою у сфері програмного забезпечення. Мікросервіси дозволяють розділити додаток на невеликі, незалежні компоненти, що полегшує розробку, тестування та масштабування програмного забезпечення. Перший розділі даної роботи присвячений історії виникнення та причинам використання мікросервісів. У другому розділі розглянуто хмарну платформу Firebase. Третій розділ містить у собі опис поетапного процесу розробки та розгортки мікросервісного застосунку для платформи агрегатора медіа. У четвертому розділі проведено функціонально вартісний аналіз попередньо розробленого застосунку.

Метою даної роботи визначено дослідити та проаналізувати функціональні можливості хмарної платформи Firebase для розробки та розгортання мікросервісів. Також провести порівняння Firebase з альтернативними хмарними платформами для виявлення їхніх переваг і недоліків.

Об’єкт дослідження: хмарна платформа Firebase та мікросервісна архітектура

Ключові слова: “Firebase”, “Мікросервісна архітектура”, “REST”, “Хмарні обчислення”, “TypeScript”, “Python”,

ABSTRACT

bachelor's thesis of Nuzhnyi Mykyta Oleksandrovich on “Application of the Firebase cloud platform for microservices development and deploy”

The structure of the thesis: Total volume of the explanatory note: 155 pages, 19 figures, 7 tables, 15 references.

The relevance of the topic is determined by the fact that microservice architecture is becoming increasingly popular in the software industry. Microservices allow you to divide an application into small, independent components, which makes it easier to develop, test, and scale software. The first section of this paper is devoted to the history of microservices and the reasons for their use. The second section describes the Firebase cloud platform. The third section describes the step-by-step process of developing and deploying a microservice application for a media aggregator platform. The fourth section provides a functional and cost analysis of the previously developed application.

The purpose of this study is to investigate and analyse the functionality of the Firebase cloud platform for developing and deploying microservices. Also, to compare Firebase with alternative cloud platforms to identify their advantages and disadvantages.

Subject of study: Firebase cloud platform and microservice architecture

Keywords: “Firebase”, “Microservice architecture”, “REST”, “Cloud computing”, “TypeScript”, “Python”,

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

API — Application Programming Interface (Прикладний Програмний Інтерфейс)

AWS — Amazon Web Services (Веб Сервіси Амазон)

REST — Representational State Transfer (Передача Репрезентативного Стану)

HTTP — HyperText Transfer Protocol (Протокол Передачі Гіпертекстових Документів)

SOA — Service-Oriented Architecture (Сервісно-Орієнтована Архітектура)

США — Сполучені Штати Америки

ВСТУП

У сучасному інформаційному суспільстві, де темпи розвитку технологій надзвичайно високі, розробка та розгортання мікросервісних архітектур стає дедалі важливішою умовою успішного впровадження програмного забезпечення.

Архітектура мікросервісів є однією з найперспективніших та найінноваційніших парадигм розробки програмного забезпечення і набирає популярності в індустрії розробки. Замість монолітної системи, де вся функціональність зосереджена в одному великому застосунку, архітектура мікросервісів сприяє декомпозиції системи на невеликі, незалежні компоненти, що взаємодіють між собою, які називаються мікросервісами.

Ця архітектурна парадигма дає змогу розробникам створювати складні системи з набору невеликих сервісів, які можна розробляти, розгортати та масштабувати незалежно один від одного. Кожен мікросервіс має свою власну функціональність і може бути розроблений з використанням різних технологій і мов програмування. Це дає змогу розробникам створювати окремі сервіси незалежно один від одного, прискорюючи розробку, спрощуючи тестування та підвищуючи гнучкість системи загалом.

Вивчення мікросервісних архітектур має особливе значення в сучасній розробці програмного забезпечення. Переваги цієї архітектури визначають ефективність розробки, масштабованість, простоту обслуговування та безпеку програмних продуктів.

Однак у міру збільшення розміру проєктів і складності мереж мікросервісів виникають нові проблеми. Управління, моніторинг і захист цих мікросервісів стає складним завданням для розробників. Саме тут на допомогу приходять хмарні платформи, що надають розробникам інструменти та сервіси для швидкого розгортання, управління та масштабування мікросервісів.

На цьому тлі тема використання хмарних платформ для спрощення та покращення процесу розробки та розгортання мікросервісів є актуальною, оскільки Firebase, розроблена компанією Google, є однією з провідних хмарних платформ, Firebase надає корисні інструменти для роботи з базами даних, автентифікацією користувачів, зберіганням файлів тощо, даючи змогу розробникам зосередитися на функціональності мікросервісів замість того, щоб витрачати час на ускладнення інфраструктури.

Одним з основних аналогів хмарної платформи Firebase є Amazon Web Services (AWS), яка пропонує широкий спектр хмарних послуг і продуктів, які можна використовувати для розробки та розгортання мікросервісів.

AWS пропонує Amazon Elastic Compute Cloud (EC2) для віртуалізації серверів, Amazon Simple Storage Service (S3) для зберігання файлів, Amazon Relational Database Service (RDS) для управління базами даних, Amazon API Gateway для управління API та багато інших сервісів. AWS також надає розширені можливості моніторингу, безпеки та масштабування, даючи змогу розробникам ефективно розгортати та керувати мікросервісами.

Іншим відомим аналогом Firebase є Microsoft Azure, який також пропонує низку віртуальних машин Azure Virtual Machines, Azure Storage, Azure SQL Database, Azure Functions тощо. Крім потужних інструментів для розробки, управління та моніторингу мікросервісів, Azure також пропонує підтримку низки мов програмування та інтеграцію з іншими інструментами розробки Microsoft.

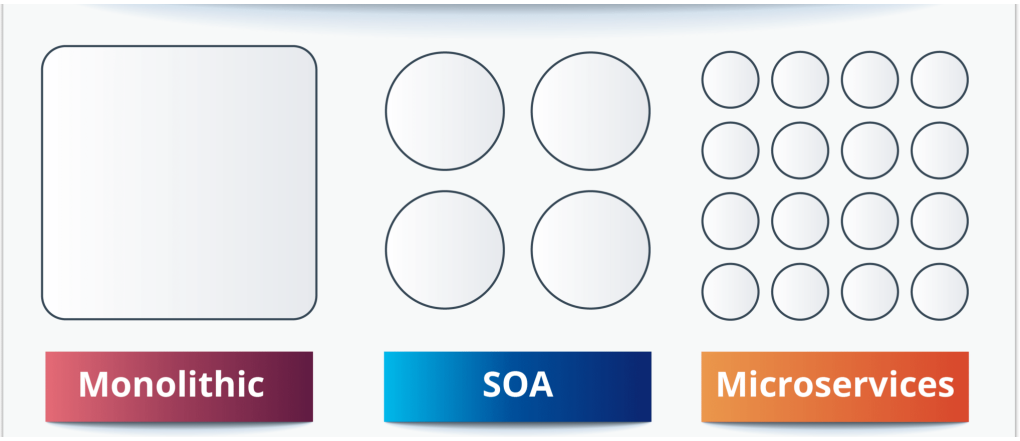
Як і Firebase, AWS і Azure пропонують розробникам можливість працювати з хмарною інфраструктурою для розгортання, масштабування та управління мікросервісами. Кожна з цих платформ має свій унікальний набір функцій і переваг, тому вибір залежить від конкретних потреб проєкту та вподобань розробника.

1. МІКРОСЕРВІСНА АРХІТЕКТУРА ЗАСТОСУНКІВ

1.1 Поняття та історія розробки мікросервісної архітектури

Історія та походження мікросервісів — це постійна спроба забезпечити кращий зв'язок між різними платформами та зробити системи простішими та зручнішими. Мікросервіси зазвичай розглядають як методологію розроблення програмного забезпечення, яка організовує додатки у вигляді слабко пов'язаних груп сервісів. Однак мікросервіси — це будь-який тип невеликих сервісів, які взаємодіють з іншими сервісами для забезпечення роботи програмного додатка. Архітектура мікросервісів використовує дрібнозернисті сервіси та легковагі протоколи.

Доктор Пітер Роджерс запровадив термін "мікровебсервіси" в презентації з хмарних обчислень у 2005 р. [3] Роджерс кинув виклик традиційному мисленню і просував програмні компоненти, що підтримують мікровебсервіси. Він також представив презентацію про концепцію "мікровебсервісів". У своїй презентації він створив функціональну модель мікросервісів, яка згодом стала реальністю. Він описав, як складні збірки сервісів працюють за простим інтерфейсом URI, і запропонував: "Будь-який сервіс, з будь-яким ступенем деталізації, може бути опублікований". Далі було пояснено, що добре налаштована платформа мікровебсервісів "застосовує основні архітектурні принципи вебсервісів і REST-сервісів, поряд з Unix-подібним плануванням і конвеєризацією, щоб забезпечити радикальну гнучкість і поліпшену простоту для сервіс-орієнтованих архітектур".



1.1 — Схематичне порівняння монолітної (зліва), сервіс-орієнтовної (в центрі) та мікросервісної архітектур(справа) [2]

Enterprise Java Beans для сервіс-орієнтованих архітектур

IBM випустила Enterprise Java Beans (EJB) 1997 року. Це була одна з перших спроб створити "маленькі" сервіси та мікросервіси, які могли б працювати з програмними компонентами, пов'язаними з Інтернетом. EJB було розроблено, щоб дозволити розробникам писати код стандартизованим чином і розв'язувати багато загальних проблем автоматично. Це була перша специфікація, що забезпечує простий спосіб "інкапсуляції та повторного використання" бізнес-логіки в корпоративних Java-додатках.

Однак із використанням корпоративних Java-бобів усе ще були проблеми. Вони могли використовуватися тільки під час роботи в Java і не могли взаємодіяти з іншими системами. Обмеження, пов'язані з роботою лише з Java та неможливістю розширення на інші платформи, призвели до появи рішення під назвою сервіс-орієнтовна архітектура (SOA), яка стала наступною еволюцією мікросервісів. Це була наступна еволюція мікросервісів. [2]

Сервіси в контексті SOA - це "самодостатні" додатки, що виконують певні завдання. SOA дає змогу цим сервісам взаємодіяти один з одним, використовуючи вільне сполучення між мовами й платформами (вільне

сполучення і сьогодні використовується для вебсервісів). Вільне з'єднання означає, що клієнт може залишатися незалежним від необхідних сервісів: дві платформи можуть взаємодіяти, навіть якщо вони не пов'язані. Це досягається шляхом використання спеціальних інтерфейсів, які можуть переводити дані. (Дехто вважає мікросервіси підмножиною SOA, але SOA і мікросервіси мають свої сильні та слабкі сторони та можуть бути класифіковані як окремі системи). Сервіси SOA в основному побудовані на простих протоколах об'єктного доступу. [5]

1.1.1 Простий протокол доступу до об'єктів (SOAP)

SOAP відіграє важливу роль у розвитку мікросервісів, випущений Microsoft 1999 року, SOAP заснований на філософії "Робіть найпростіші речі", SOAP використовує HTTP (протокол передачі гіпертексту) для використання об'єктних методів. Протокол) для використання методів об'єктів. Це спосіб передачі невеликих обсягів інформації, або повідомлень, через Інтернет. Ці повідомлення використовують формат XML і зазвичай відправляються за допомогою HTTP; використання HTTP, як для вебсторінки, має ту перевагу, що воно може обійти більшість мережевих брандмауерів (оскільки брандмауери зазвичай не блокують HTTP-трафік; див. нижче), більшість повідомлень SOAP можуть проходити через них без проблем). Це рішення використовує переваги двох тенденцій у світі комп'ютерних технологій на початку 2000-х років:

— Дедалі ширше використання HTTP у корпоративних мережах.

— Підтримка, що надається механізмами автоматичної реєстрації та налаштування текстового Інтернет-спілкування.

Повідомлення SOAP укладені в "конверт", що містить тіло і заголовок. Заголовок може містити ідентифікатор повідомлення та дату передачі, а тіло - власне повідомлення. Усі повідомлення SOAP використовують один і той

самий формат і тому сумісні з різними протоколами та операційними системами. Наприклад, надсилання SOAP-повідомлення з комп'ютера під управлінням Windows XP на вебсервер на базі Unix не призведе до пошкодження повідомлення; Unix-сервер може відкрити файл або переспрямувати повідомлення в найбільш відповідне місце.

Хоча SOAP має свої переваги, він не масштабується і не підтримує обробку помилок, тому дві функції мікросервісу виконуються автоматично. Крім того, SOA і SOAP спочатку були дуже простими, але спроби розширити їхню корисність шляхом додавання шарів зробили їх незграбними і громіздкими.

SOA продовжує використовуватися як базовий інструмент, тоді як зусилля з підвищення ефективності SOAP тривають.

1.1.2 REST.

Загалом, обчислювальна спільнота почала відмовлятися від концепцій, необхідних SOAP, приблизно у 2005-2007 роках; у період із 2008 до 2010 року популярності набуло передання репрезентативного стану (REST), яке просунулося в цьому напрямі. Цей архітектурний стиль визначає обмеження, які використовуються для побудови вебсервісів, і підтримує використання хмари для розроблення програмних застосунків; вебсервіси, які відповідають архітектурному стилю REST, називаються RESTful вебсервісами, а такі сервіси підтримують сумісність між комп'ютерними системами, що взаємодіють через Інтернет.

REST з використанням HTTP є дуже поширеною практикою в сучасній розробці мікросервісів; RESTful API описують інтерфейси прикладного програмування (API), які POST, GET, PUT і DELETE дані за допомогою HTTP-запитів. API портативні, масштабовані, прості у використанні та легко

інтегруються. API вебсайтів використовують код для підтримки зв'язку між двома додатками.

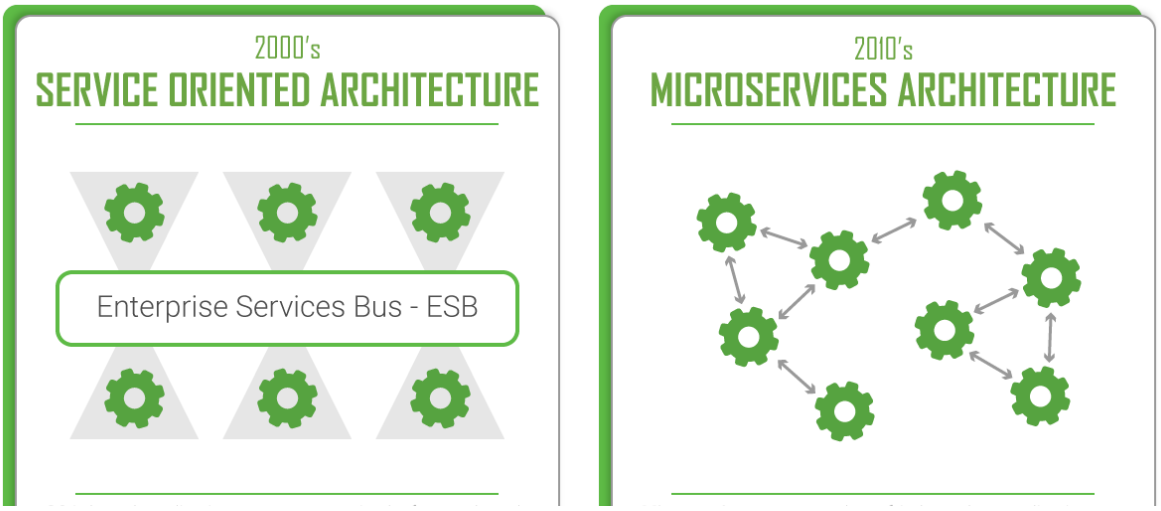


Рисунок 1.2 — Схематичне зображення сервісно-орієнтованої та мікросерверної архітектур [13]

Рой Філдінг працював над створенням архітектурного стилю REST з 1996 до 1999 року і 2000 року опублікував свою докторську дисертацію "Архітектурні стилі та розробка мережевого програмного забезпечення". Філдінґ обговорив архітектурний стиль REST у книзі "Архітектури". Філдінґ виступав за гнучкість, а не за жорсткість; основна ідея REST полягала в тому, що для кожної частини даних URL залишається незмінним, але те, як він працює, залежить від того, як він використовується.

"Протягом усього процесу стандартизації HTTP мене просили захистити вибір дизайну вебу. Це надзвичайно важко зробити в процесі, який приймає пропозиції від будь-якої людини щодо питання, яке швидко ставало центральним для всієї індустрії... Я отримав коментарі від більш ніж 500 розробників, багато з яких були блискучими інженерами з десятиліттями досвіду, і мене попросили захистити найабстрактніші вебвзаємодії. Мені довелося пояснювати все — від найабстрактніших понять до найтонших деталей синтаксису HTTP. Цей процес допоміг мені доопрацювати мою

модель до основного набору принципів, властивостей і обмежень, які ми зараз називаємо REST". [1]

REST часто використовується в поєднанні з мікросервісами.

Сучасний стан мікросервісів

У травні 2011 року на семінарі архітекторів програмного забезпечення, що проходив недалеко від Венеції, термін "мікросервіси" було використано для опису архітектурного стилю, який деякі учасники нещодавно вивчали. У травні 2012 року вони вирішили, що "мікросервіси" - це саме те. Вони вирішили, що "мікросервіси" - це найбільш відповідна назва для їхньої роботи, і офіційно прийняли її. Вони експериментували зі створенням безперервно розгортаємої системи, застосовуючи на практиці філософію DevOps. Ця форма архітектури швидко завоювала популярність.

DevOps - це набір методів розробки програмного забезпечення, які об'єднують розробку (Dev) і експлуатацію (Ops) для скорочення життєвого циклу розробки. Архітектура мікросервісів ефективно об'єднує обидва відділи для створення більш надійних, більш продуктивних і більш стійких додатків. У результаті розробники можуть частіше вносити оновлення, гарантуючи, що програмне забезпечення відповідає бізнес-цілям. Архітектури мікросервісів швидко стали стандартом для DevOps. Вони особливо корисні для безперервного випуску програмного забезпечення.

1.2 Навіщо використовувати мікросервісну архітектуру

Переваги архітектури мікросервісів включають:

— підтримка: мікросервіси набирають популярність і підтримуються багатьма постачальниками технологій і хмарних послуг. Поєднання ефективності та підтримки в поєднанні з простотою використання змусило багато компаній перейти на мікросервіси;

— мовна незалежність: мікросервіси можуть працювати практично з будь-якою технологією з незначними модифікаціями;

— DevOps: мікросервісні архітектури розроблені для підтримки філософії DevOps. У результаті DevOps допомагає керувати розробкою додатків на основі мікросервісів;

— ресурси не розділяються: ранні сервісні архітектури були розроблені з акцентом на поділ ресурсів. Загальні ресурси, як правило, були надмірно складними. Архітектури мікросервісів дозволяють інкапсулювати кожен сервіс. Тому розробники можуть налаштовувати мікросервіси, не зачіпаючи інші сервіси.

Мікросервіси відрізняються від традиційних монолітних систем, у яких компоненти розробляються окремо. Архітектури мікросервісів вимагають мінімального управління кількома або більше вторинними сервісами, побудованими на різних мовах програмування і з використанням різних технологій зберігання даних. Мікросервісні команди дають змогу іншим розробникам використовувати різні мови програмування і технології без шкоди для загальної архітектури розроблення програмного забезпечення. Складність монолітних архітектур нелегко зрозуміти, а робота з великим кодом додатків може виявитися складною. Мікросервіси та хмарні архітектури об'єднують функціональність програми, розділяючи її на низку незалежних сервісів.

Мікросервіси знижують складність самих сервісів і підвищують масштабованість під час побудови розподілених систем; вони автоматично надають сервіси на вимогу, які можуть реплікуватися десятки разів на хвилину; їх можна використовувати для надання єдиного сервісу великій кількості користувачів; їх можна використовувати як єдиний сервіс для великої кількості додатків. Мікросервіси ідеально підходять для розподілених хмарних середовищ, оскільки їх можна використовувати як хмарні сервіси,

які запускають нову машину за лічені секунди. Таким чином, хмарні ресурси мікросервісів можуть використовуватися більш ефективно, знижуючи щомісячний рахунок хмарному провайдеру. Стандартні монолітні архітектури (серверні додатки й бази даних) засновані на одній і тій самій величезній кодовій базі.

Мікросервіси — це комплексна концепція, яку можна застосовувати на гранульованому або модульному рівні для розроблення додатків, продуктів і рішень. Архітектури мікросервісів додатків ще недостатньо зрілі, щоб проводити A/B-тестування цілих додатків і бізнес-процесів. Майбутнє мікросервісів веде до повністю без серверної архітектури, яка приваблива через обіцяну економію в плані кількості використовуваної обчислювальної потужності. Архітектура мікросервісів належить до архітектурного стилю, що використовується під час розроблення застосунків. Вона належить до архітектурного стилю розроблення застосунків.

З поширенням мікросервісів підприємства прагнуть використовувати підвищену гнучкість і масштабованість, і Market Forecast [4] прогнозує зростання світового ринку хмарних мікросервісів на 22,5%, при цьому на ринку США очікується зростання на 27,4%. Майбутнє мікросервісів розгортатиметься різними та складними шляхами.

У дослідницькому звіті Cloud Microservices Market Research Report [4], опублікованому в лютому 2020 року, прогнозується, що обсяг світового ринку архітектури мікросервісів зростатиме з 2019 до 2026 року з темпом зростання у 21,37%, сягнувши до 2026 року значення в 3,1 млрд доларів США.

Розробники можуть об'єднати мікросервіси з подієво-орієнтованими архітектурами для побудови розподілених, масштабованих, відмовостійких і масштабованих систем, які споживають і обробляють великі обсяги подієвої інформації в реальному часі. Мікросервіси дають змогу командам розробників впроваджувати нові функціональні можливості без необхідності

змінювати або переписувати великі частини наявного коду на спеціальній основі. Моніторинг мікросервісів дає змогу тестувати архітектуру та продуктивність сервісів і виявляти майбутні проблеми для налагодження. Підприємства визнають переваги мікросервісних архітектур. Прийняття гібридних хмар різними кінцевими користувачами та галузями є ключовим фактором, що стимулює зростання ринку мікросервісів.

Без серверні архітектури доводять модель центральної хмари та мікросервісів до логічного завершення. Збалансований, органічний підхід допомагає оптимізувати процеси, необхідні для створення IT-хмари з мікросервісами без втрати контролю або повернення до монолітного минулого.

Мікросервіси вимагають специфічних функцій, які залежать від ОС і машин. Необхідно відстежувати та керувати ризиками безпеки, швидким масштабуванням і динамічними хмарними архітектурами. Зміни та оновлення вимагають від розробників розробки та розгортання всього стека. Коли компанії розробляють і впроваджують архітектуру мікросервісів, їм необхідно мати схеми, які можна повторно використовувати і розширювати для майбутніх проєктів і послуг.

2. ХМАРНА ПЛАТФОРМА FIREBASE

2.1 Ключові інструменти Firebase

Firebase - це платформа для розробки мобільних і вебдодатків, яка надає широкий спектр інструментів і сервісів для управління хмарними обчисленнями та зберіганням даних. Придбана у 2014 році компанією Google, платформа з того часу стала одним з найпопулярніших інструментів для розробників в секторі мобільних додатків індустрії мобільних додатків.

Рисунок. 2.1 — Основні компоненти Firebase [12]

На рис. 2.1 зображено основні інструменти яки надає Firebase, а саме:

— Firebase Authentication: сервіс для аутентифікації користувачів. Надає можливість використовувати різні методи аутентифікації, включаючи електронну пошту/пароль та соціальні мережі (наприклад, Google, Facebook, Twitter);

— Firestore: база даних NoSQL, що забезпечує зберігання даних у вигляді колекцій і документів; Firestore забезпечує швидкий доступ до даних і синхронізацію в режимі реального часу між різними пристроями;

— Realtime Database: Інструмент для зберігання даних, який надає схожу функціональність з Firestore, але базується на моделі ключ-значення. Дозволяє створювати додатки в режимі реального часу, які автоматично оновлюються при зміні даних;
— Firebase Storage: сервіс для зберігання та управління медіафайлами, такими як зображення, відео та аудіо. Файли легко доступні й можуть бути використані для зберігання аватарів користувачів, зображень продуктів тощо;
— Firebase Functions: сервіс, який дозволяє створювати та запускати серверні функції в хмарі. За допомогою нього реалізується серверна логіка, яка реагує на певні події, такі як створення нових користувачів або оновлення даних, але також подією може бути виклик функції іззовні, про що буде розглянуто більш детально;
— Firebase Hosting: послуга для розгортання і розміщення статичних файлів, вебсторінок і вебдодатків, використовуючи глобальну мережу CDN (Content Delivery Network);
— Firebase Cloud Messaging (FCM): сервіс для надсилання повідомлень на мобільні пристрої та веббраузери. Розробники можуть надсилати повідомлення користувачам, які встановили додаток, та реалізувати push повідомлення;
— Firebase Analytics: аналітичний інструмент, який надає інформацію про використання додатків, поведінку користувачів та ефективність маркетингових кампаній.

Також сервіс має багато інших корисних функцій, і кожен компонент можна використовувати окремо або в поєднанні з іншими компонентами. Розробникам також доступні SDK (набори для розробки програмного забезпечення) для різних платформ, включаючи Android, iOS, веб та Unity. [12]

2.2 Firebase Emulators

Зважаючи на потреби під час розробки програмного забезпечення та вибір Firebase бази даних, одним з корисних інструментів, які можна використовувати під час розробки, є Firebase Emulators. Firebase Emulators - це локальні середовища для емуляції різних Firebase сервісів, що дозволяють розробникам працювати з цими сервісами на своєму власному комп'ютері, спрощуючи процес розробки та тестування.

Firebase Emulators надають можливість емулювати Firebase Realtime Database, Firestore, Authentication, Cloud Functions, Hosting та інші сервіси. Це дозволяє розробникам працювати з даними та функціональністю Firebase без необхідності підключення до хмарної інфраструктури, що забезпечує більшу швидкість та зручність у процесі розробки.

З використанням Firebase Emulators, можливо емулювати базу даних та проводити тестування функціонала без необхідності завантаження даних на хмару Firebase. Це особливо корисно під час розробки, коли ви хочете швидко та ефективно перевірити роботу своєї програми, не витрачаючи ресурси на кожне збереження або отримання даних з хмари.

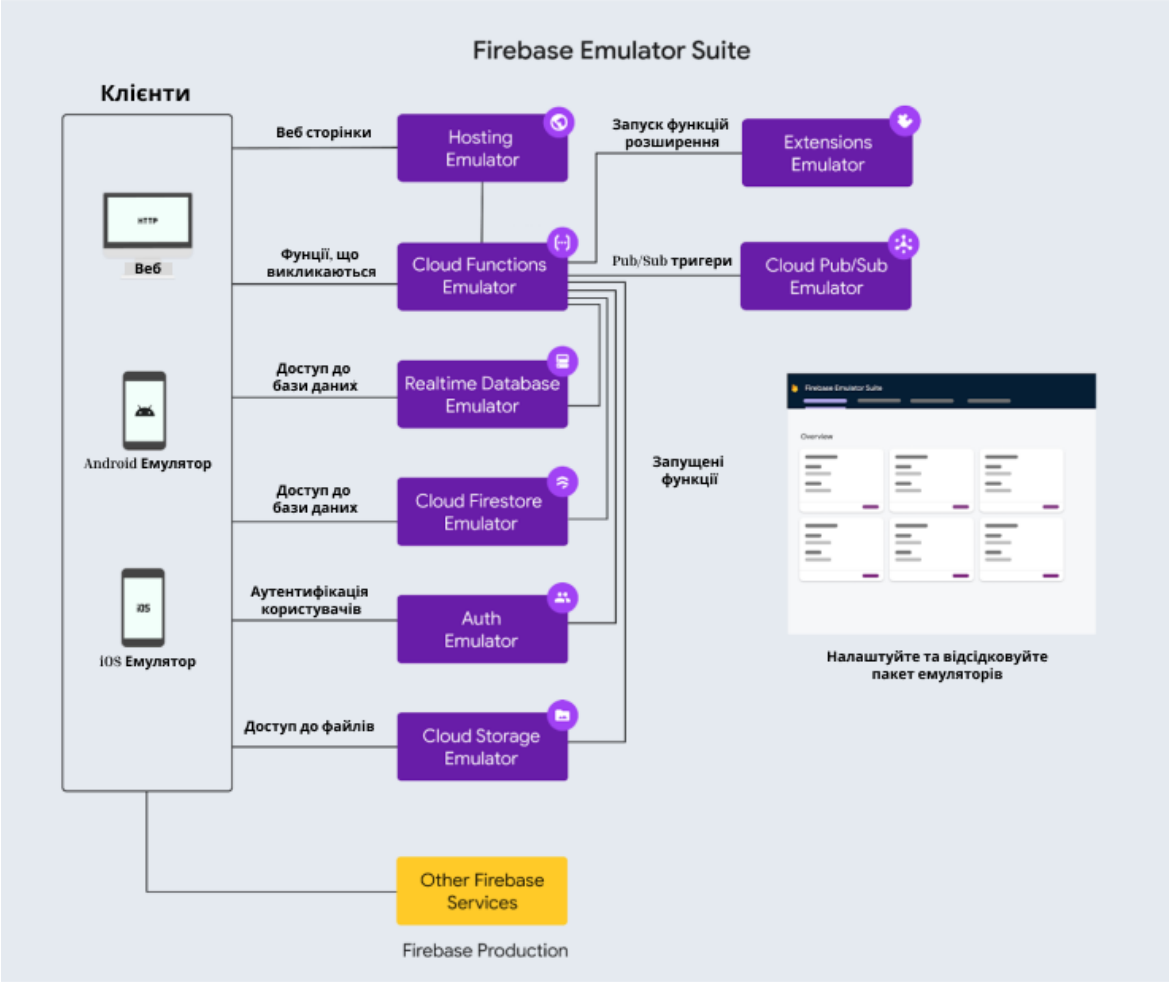


Рисунок 2.2 — Схема поєднання емуляторів Firebase [12]

Firebase Emulators також дозволяють налаштовувати різні сценарії та умови для тестування додатка. Емулятори надають можливість симулювати різні стани бази даних, автентифікацію користувачів та взаємодію з іншими сервісами Firebase. Це допомагає виявити та виправити можливі помилки та недоліки в вашому програмному забезпеченні ще до його релізу.

З використанням локального інтерфейсу користувача Firebase Emulators, простіше та більш ефективно розробляти та тестувати програмне забезпечення, знижуючи залежність від хмарної інфраструктури та збільшуючи продуктивність розробки. Цей інструмент допоможе економити час, ресурси та забезпечити якісний контроль над розробкою програмного продукту.

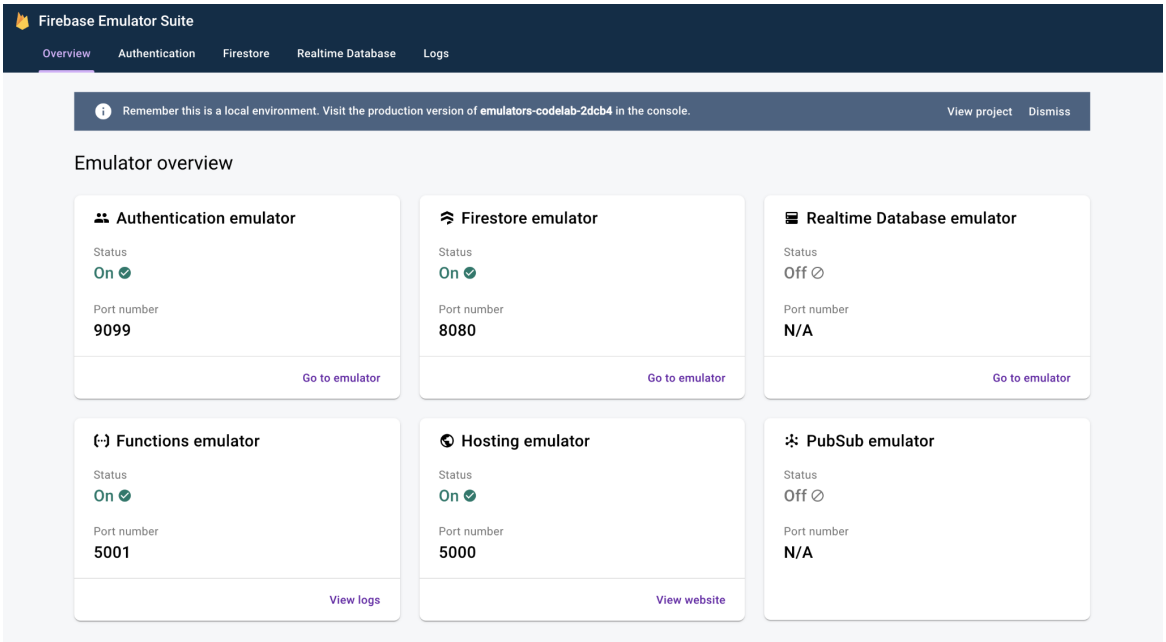


Рисунок 2.3 — Firebase Emulators додатковий локальний інтерфейс користувача

Загалом, Firebase Emulators є потужним інструментом для розробки програмного забезпечення, що дозволяє ефективно використовувати Firebase базу даних та інші сервіси, забезпечуючи зручне та продуктивне середовище для розробки та тестування додатка.

На жаль, емулятори також мають певні недоліки. Не зважаючи на свою зручність, вони забирають певні потужності персонального комп’ютера розробника, тож якщо мова йде про розробку великого застосунку та запуск усіх емуляторів одразу, особливо на не дуже сучасних системах, це може призвести до проблем з функціонуванням комп'ютера.

Але локальна емуляція хмарного середовища також містить у собі іншу проблему, особливо якщо архітектура була спроєктований не найкращим способом, через що можуть виникати конфлікти у проєкт. Наприклад при використанні декількох Firebase застосунків, під час запуску емуляторів та розгортки можуть виникати конфлікти пов’язані з файлами які містять облікові дані для різних проєктів.

Таким чином, під час створення нового програмного продукту, необхідно одразу брати до уваги, що процес розробки охоплювати проєктування для Firebase. Звісно, розробники з Google зробили усе можливо, для того, щоб міграція вже наявного проєкту на їх сервіс пройшла якомога безболісно. Firebase пропонує доволі багато розширень та інтеграцій, але найголовніше, не стримує користувачів у своєму середовищі.

Припустімо, що додаток вже існує та використовує технологію яку Firebase не надає, не зважаючи на це, існує дуже велика ймовірність, що дану технологію можна використовувати з підключенням іззовні. [12]

2.3 Підтримка мікросервісної архітектури платформою Firebase

Архітектура мікросервісів з використанням Firebase Functions надає розробникам можливість створювати та керувати незалежними функціями, які працюють в хмарному середовищі Firebase. Це дозволяє вам розбити додаток на окремі компоненти, які виконують конкретні функції, та розгорнути їх без зайвих зусиль.

Використовуючи модуль функцій у Firebase можливо створювати незалежні функції, які відповідають за окремі функціональні аспекти додатка. Наприклад, мати окремі функції для обробки платежів, надсилання повідомлень, обробки запитів користувачів тощо. Кожна функція працює незалежно від інших, що забезпечує гнучкість та масштабованість додатка.

Firebase Functions має функціонал під написання функцій мовою програмування JavaScript, TypeScript, Python, Java та Go. Тобто розробник має змогу використовувати знайомий синтаксис та інструменти розробки для створення та налагодження функцій, або поєднувати декілька мов між собою під конкретні задачі.

Крім того, Firebase Functions інтегрується з іншими сервісами Firebase, такими як Firebase Firestore, Firebase Authentication та Firebase Cloud Messaging для усіх перелічених мов програмування. Це дозволяє вам використовувати ці сервіси для зберігання даних, автентифікації користувачів та спілкування з вашими мікросервісами. Наприклад, ви можете створити функцію, яка реагує на зміни в базі даних Firestore та автоматично відправляє повідомлення користувачеві за допомогою Firebase Cloud Messaging.

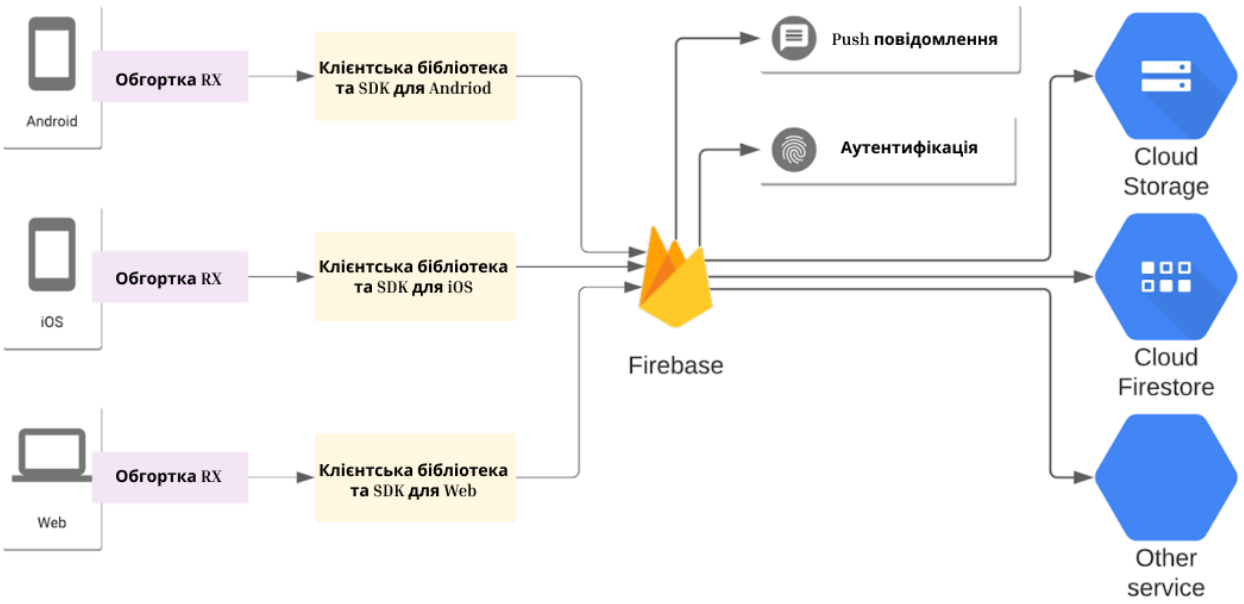


Рисунок 2.4 — Приклад мікросервісної архітектури, створеної за допомогою Firebase [12]

Один з головних переваг використання Firebase Functions для архітектури мікросервісів — це масштабованість. Firebase автоматично масштабує ваші функції відповідно до потреб додатка. Це означає, що дуже легко розгортати нові екземпляри функцій, якщо виникає потреба обробляти більше запитів або забезпечувати більшу продуктивність. Firebase динамічно збільшує або зменшує кількість екземплярів функцій в залежності від навантаження, що дозволяє ефективно використовувати ресурси та забезпечує швидку відповідь користувачам.

В цілому, використання Firebase Functions для архітектури мікросервісів дозволяє створити гнучку, масштабовану та ефективну систему, яка розбиває ваш додаток на незалежні компоненти. За допомогою даного хмарного сервісу легко розгортати та керувати функціями, використовуючи інструменти Firebase, та інтегрувати їх з іншими сервісами для досягнення більшої функціональності та продуктивності додатка.

2.4 Огляд сучасних проєктів які використовують Firebase

Firebase є дуже популярною платформою для розробки сучасних веб та мобільних додатків. Багато відомих компаній та продуктів використовують Firebase як свою основну технологію. Ось декілька прикладів сучасних застосунків, які використовують Firebase:

— Twitter, соціальна мережа для надсилань текстових повідомлень, висловлення думок, використовує Firebase для зберігання і синхронізації даних у режимі реального часу, реалізації аутентифікації та забезпечення масштабованої функціональності сервера;

— The New York Times у своїй онлайн газеті використовує Firebase для управління і синхронізації контенту, доставлення сповіщень користувачам і аналізу даних;

— Instacart: популярна служба доставлення харчових продуктів, використовує Firebase для зберігання даних про замовлення, управління користувачами та обробки платежів;

— Shazam: додаток для розпізнавання музики, використовує Firebase для зберігання музичних профілів користувачів, синхронізації даних і надання персоналізованих рекомендацій;

— Duolingo: популярний застосунок для вивчення мов, використовує Firebase для зберігання даних про навчання користувачів, відстеження прогресу та надання попереджень;

— TikTok: популярна платформа для створення та обміну короткими відео, використовує Firebase для зберігання відео, опрацювання коментарів і взаємодії з користувачами;

— Airbnb: сервіс спільного бронювання помешкань, використовує Firebase для опрацювання платежів, управління бронюваннями та зворотного зв'язку з користувачами;

— Discord: платформа для спілкування геймерів, використовує Firebase для реалізації групових чатів, забезпечення голосового і відеозв'язку та зберігання уподобань користувачів;

— Strava: застосунок для відстеження фітнесу, використовує Firebase для зберігання даних про тренування користувачів, надання статистики та підтримки соціальної взаємодії;

— Headspace: популярний застосунок для медитації та підтримки психічного здоров'я Headspace використовує Firebase для зберігання станів користувачів, підтримки платежів і надання персональних рекомендацій;

Ці приклади показують різноманітність застосувань Firebase в різних галузях, від соціальних мереж до фітнесу та добробуту. Firebase забезпечує гнучкі інструменти та інфраструктуру, які допомагають розробникам створювати інноваційні та потужні додатки для користувачів. Також важливо зазначити, що дані сервіси та додатки, використовують Firebase саме для розгортання власних мікросервісів, та використання мікросервісів запропонованих самою платформою.

2.5 Порівняння платформи Firebase з конкурентами

Одним з головних конкурентів Firebase є сервіс від компанії Amazon, а саме Amazon Web Services Amplify, скорочено AWS Amplify. Розглянемо наступні переваги AWS, щоб краще зрозуміти ситуацію.

AWS забезпечує більшу гнучкість, ніж Firebase. Крім того, Amazon надає свободу встановлювати елементи з відкритим кодом і переходити на інший сервіс. Подібно до інших сервісів, таких як Netlify, AWS пропонує постійне рішення для безперервного розгортання. Крім того, Google Cloud (частково сервіси якого використовує Firebase) дозволяє безперервне розгортання. Однак потребує додаткових налаштувань.

AWS надає різні налаштування для різних завдань, таких як тестування, виробництво та розробка. Однак, з Firebase є необхідним створення багатьох проєктів, що є тривалою процедурою. Також AWS надає надійні API на різних мовах програмування, які допоможуть оптимізувати інфраструктуру. Будь то створення резервних копій або розгортання нового компонента, API AWS, швидше за все, впораються з усім необхідним.

AWS був розроблений і створений, щоб забезпечити найбільш універсальне, безпечне і надійне середовище хмарних обчислень з усіх можливих. Основна інфраструктура AWS була створена, щоб задовольнити потреби в безпеці навіть найбільш вразливих підприємств по всьому світу. AWS підтримується комплексними та складними засобами захисту, які легко задовольняють вимоги безпеки.

Менша вартість. У багатьох випадках AWS буде дешевше, ніж Firebase, оскільки він не є керованим сервісом. Крім того, AWS пропонує довгострокові контракти, які дозволяють заощадити до 75% від звичайних цін.

Модель ціноутворення Pay-As-You-Go. AWS надає високомасштабовані та універсальні послуги хмарних обчислень за доступною ціною. З AWS оплачуються лише ті послуги, які потрібні організації. Дуже зручною є можливість легко розширювати послуги за потреби. Наприклад отримати нескінченне сховище для архівів і резервних копій, а також створювати нові сервери, зменшувати або збільшувати розмір наявних серверів і багато іншого, залежно від наявних потреб.

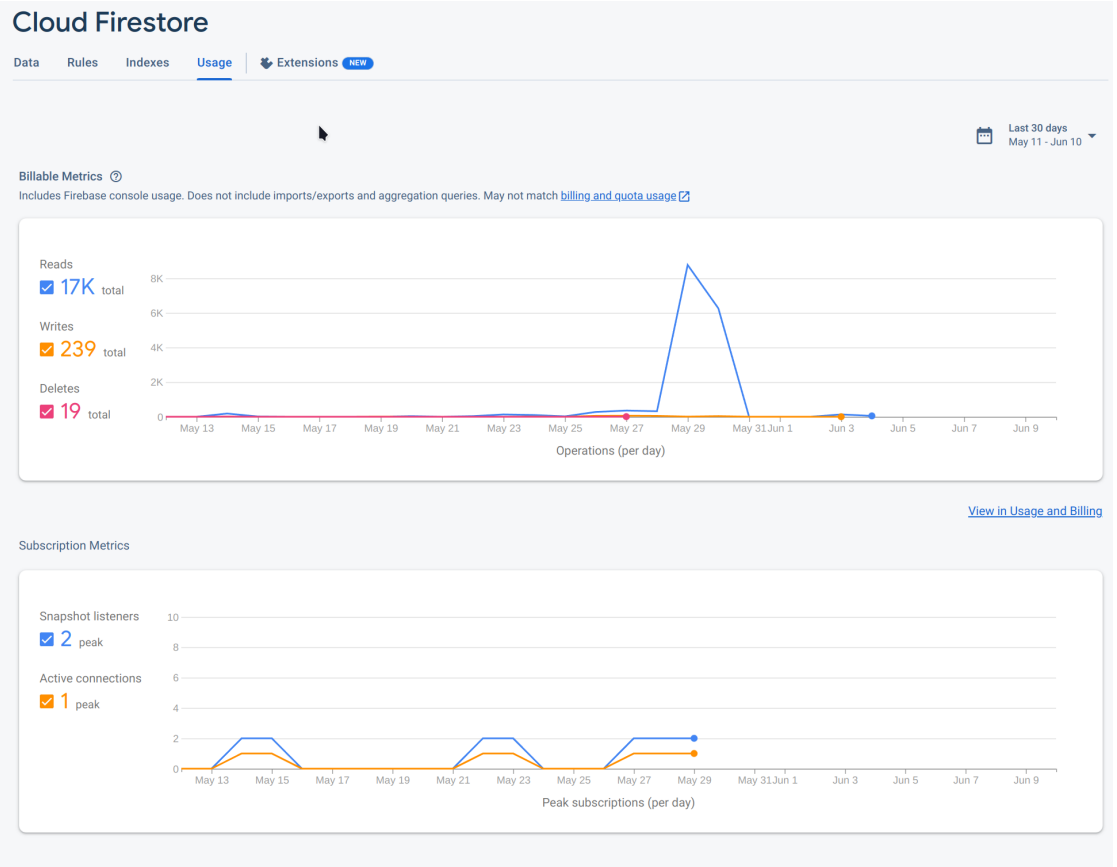


Рисунок 2.5 — Статистка використання бази даних з розробляємого проєкту (синім — читання з БД, жовтим — запити, червоним — видалення)

В остаточному порівнянні Firebase проти AWS, Firebase від Google - це Back-end як послуга (BaaS). Хостинг, керовані сервіси та платформа для розробки додатків — все це надається цим сервісом. Хоча AWS є постачальником інфраструктури як послуги, це не єдиний варіант. Це

найвідоміша у світі компанія з надання хмарних послуг. Вона надає доступ до серверів, сховищ і віртуальних машин за нижчою ціною, ніж Firebase

Таблиця 2.1 - Порівняння Firebase та AWS Amplify

Сервіс	Firebase	AWS Amplify
	Присутність функціоналу	
Авторизація користувачів	так	так
Надсилання повідомлень	так	так
Аналітика	так	так
Хостинг	так	так
Безсервісні функції	так	так
Сховище файлів	так	так
База даних	Realtime, Firebase	graphQl
Пошук	ні	так
Мобільна аутентифікація	так	так
Аналіз продуктивності	crashlytics performance test lab	ні
Інше	Віддалене налаштування	-
Машинне навчання	Повноцінна підтримка машинного навчання (ML kit)	Deep Learning AMIs
Тестове середовище	Необхідне використання декількох Firebase проєктів	Відповідні середовища
Безперервне розгортання	ні	GitHub комміт розгортання
CLI	так	так

Що до аутентифікації, обидві системи забезпечують безпечну автентифікацію користувачів у вашому додатку. Наявна інтеграція з відомими постачальниками ідентифікаційних даних, такими як Facebook та Apple, або створення власного провайдера ідентифікації користувачів. Найбільш суттєва різниця між системами полягає у складності їхніх зв'язків з об'єднаними постачальниками ідентифікаційних даних, причому Firebase є простішою у використанні та обслуговуванні.

Замість того, щоб зберігати дані користувачів на своїх серверах, наявна можливість передати це завдання стороннім рішенням для зберігання об'єктів від Google або Amazon. Хмарне сховище Google - це швидкий і потужний вибір, тоді як Amazon S3 надає більш надійне рішення для масового завантаження, керування версіями об'єктів і тегування метаданих.

За потреби гнучкого хостингу вебдодатків, обидва сервіси надають глобально розподілені хмарні мережі, оптимізовані для фреймворків React та Angular (серед інших). І Google Firebase, і AWS Amplify дозволяють витрачати більше часу на розробку додатку і менше часу на інтеграцію з бекендом, пропонуючи багатофункціональні набори інструментів, які спрощують процес реалізації критично важливих, але монотонних бекенд-інтерфейсів.

У порівнянні з Firebase, обидві платформи надають багатий набір можливостей сповіщень для онлайн та мобільних додатків. Знову ж таки, рішення Google, Firebase Cloud Messaging, було створено спеціально для випадку використання Back-end як сервісу. Водночас AWS використовує Pinpoint для маркетингового дизайну і Simple Notification Service (SNS) для доставляння повідомлень клієнтам. Таке розділення від імені AWS приносить користь клієнтам, які інтегруються з існуючим інфраструктурним середовищем AWS, але додає непотрібної складності невеликим проєктом.

Кожна платформа має безплатний рівень, а також варіант з оплатою в міру використання. Багато сервісів Firebase є повністю безплатний. Уся аналітика, комунікації та прогнозування, а також інші функції є абсолютно безплатними. А з тарифним планом Blaze оплата знімається лише за те, що не входить до безплатного рівня. Firebase використовує переваги Google Cloud Pricing для обчислень та зберігання даних. Якщо вимоги до додатка вже визначені (або визначитеся), існує можете скористатися їхнім інтерактивним калькулятором витрат. Аналогічно користувач самостійно вирішує, скільки послуг AWS необхідно використовувати для оплати Amplify. Хоча Amazon надає кілька оцінок вартості на своїй сторінці з цінами на Amplify, їх загальний калькулятор цін AWS, ймовірно, буде більш корисним.

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОГО ЗАСТОСУНКУ НА ПЛАТФОРМІ FIREBASE

У даному розділі буде розглянуто проєктування та створення серверного застосунку для агрегації медіа, що включатиме у собі:

— захищену авторизацію користувачів за допомогою Firebase Authentication;

— кінцеві точки для керування профілем користувача;

— можливість додавати медіа та знаходити медіа по ключовим словам;

— можливість для користувачів підписуватись на медіаресурс та отримувати оновлення;

— незалежні між собою мікросервіси для необхідних задач

Також під час розробки необхідно розв'язати проблему, яка пов’язана з відсутністю пошуку у базі даних Firebase.

3.1 Проєктування мікросервісної архітектури за допомогою Firebase

При проєктуванні мікросервісної архітектури додаток розбивається на окремі сервіси, кожен з яких виконує певну функцію. Наприклад, у типовому застосунку можна виділити такі сервіси:

— сервіс для транзакцій користувачів;

— сервіс для управління контентом;

— сервіс для надсилання SMS і так далі.

Кожен сервіс може мати власну колекцію документів у Firestore, що дозволяє керувати та розширювати кожну функціональну частину додатка незалежно.

Також при проєктуванні архітектури мікросервісів для платформи Firebase важливо враховувати такі вимоги до додатка, як масштабованість, безпека і продуктивність. Firebase надає потужні інструменти для проєктування такої архітектури й дозволяє розбити додаток на незалежні сервіси, та під'єднати до них вже наявні сервіси, також управляти доступом до них і забезпечити масштабованість і продуктивність.

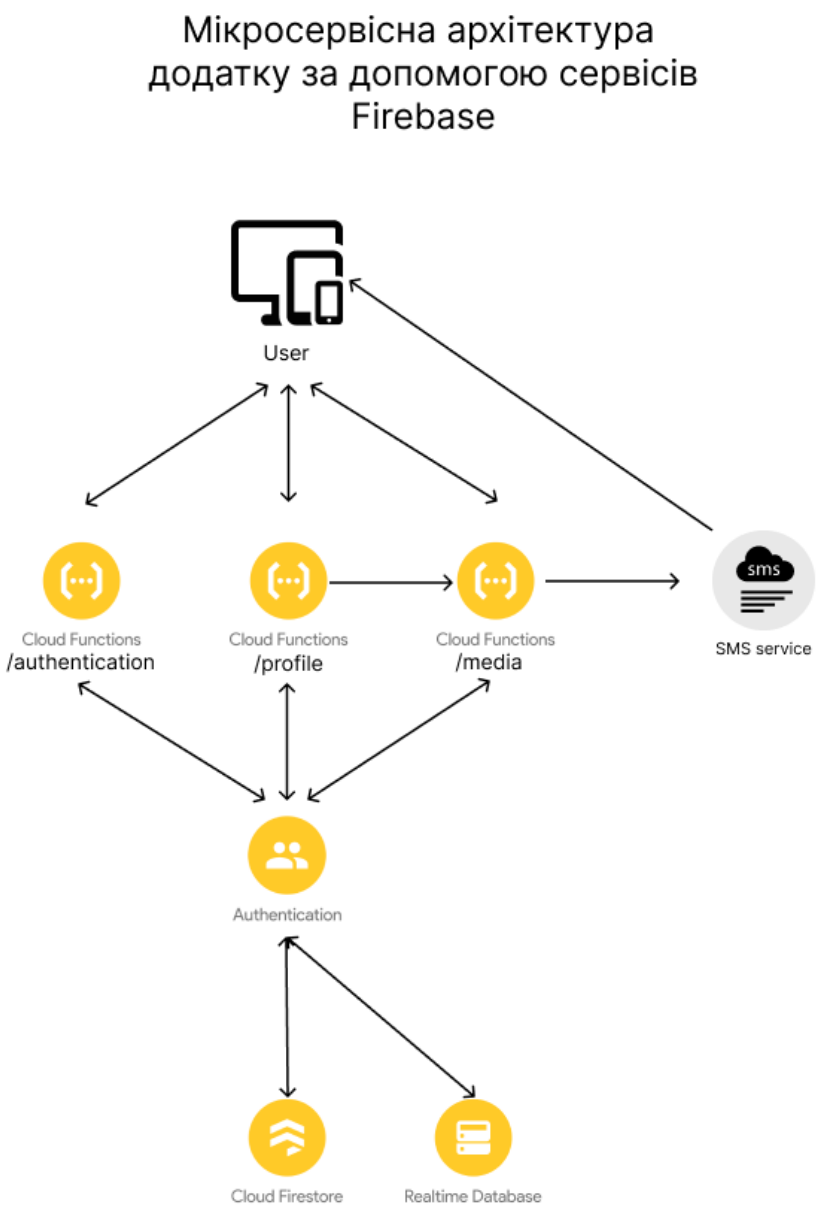


Рисунок 3.1 — Архітектура мікросервісного додатка використовуючи Firebase

Розроблена архітектура наслідує патерн проєктування мікросервісів “Агрегатор”, де у ролі самого агрегатору використовується клієнт вебсторінки. Даний патерн є однією з найпоширеніших реалізацій мікросервісної архітектури, також легко масштабується та швидко розробляється.

Мікросервіс “authentication”

Сервіс відповідає за автентифікацію користувачів додатка за допомогою Firebase Authentication, щоб забезпечити безпечний процес входу та реєстрації користувачів. Після успішної автентифікації служба отримує токен, який може бути використаний для автентифікації користувача при доступі до інших мікросервісів.

Мікросервіс “profile”

Даний сервіс використовує Firebase Authentication для перевірки токена і надання доступу до профілю користувача. Він також використовує Firebase Firestore для зберігання оновлень та інформації про профіль користувача. Користувачі можуть вносити зміни до свого профілю, і ці зміни зберігаються у Firestore для подальшого використання.

Мікросервіс “media”

Наступний сервіс виконує операції CRUD (створення, отримання, оновлення та видалення) над медіаресурсами. Він також дозволяє користувачам підписуватись і відписуватись від медіаресурсів. Firebase Firestore використовується для зберігання даних про медіа та Realtime Database для зберігання інформації про підписки.

Сервіс надсилання SMS-повідомлень

Програма використовує мікросервіс для надсилання SMS-повідомлень, підключений до мікросервісу "media". Ця служба дозволяє надсилати

SMS-повідомлення користувачам у відповідь на певні дії або події, пов'язані з медіаресурсом.

Таким чином в архітектурі наведеної на рис. 3.1 кожен мікросервіс має свою зону відповідальності та дозволяє розробникам без зайвих перешкод розширяти застосунок та масштабувати його в залежності від потреб, додаючи нові мікросервіси, які стабільно взаємодіють з середовищем Firebase.

3.2 Розробка за допомогою Firebase Emulators

Розробляти функції Firebase за допомогою емуляторів Firebase - це потужний інструмент, який дозволяє тестувати та налагоджувати ваші функції локально, перш ніж розгортати їх на хмарних серверах Firebase. Ця комбінація інструментів робить розробку більш ефективною та швидкою.

Для налаштування середовища розробки необхідно виконати наступні дії:

— Налаштування Firebase CLI;

— Ініціалізація проєкту Firebase;

— Створення та конфігурація Firebase Functions;

— Розробка програмного коду.

Далі буде розглянуто дані пункти більш детально та розроблено додаток, який буде розгорнуто.

3.2.1 Налаштування Firebase CLI

Firebase CLI (Command Line Interface) - це інструмент командного рядка, який використовується для доступу до функціональності Firebase з робочого терміналу. Firebase CLI надає можливість керувати проєктами Firebase, розгортати функції, налаштовувати хостинг, керувати базами даних та багато іншого.

Залежно від обраної операційної системи, на який відбувається розробка, необхідно належним методом встановити npm (node packet manager) пакет “firebase-tools”, та провести його налаштування у глобальній середі операційної системи за допомогою команди “npm install -g firebase-tools”

Наступною дією буде виконання аутентифікації за допомогою облікового запису Google та команди “firebase login”

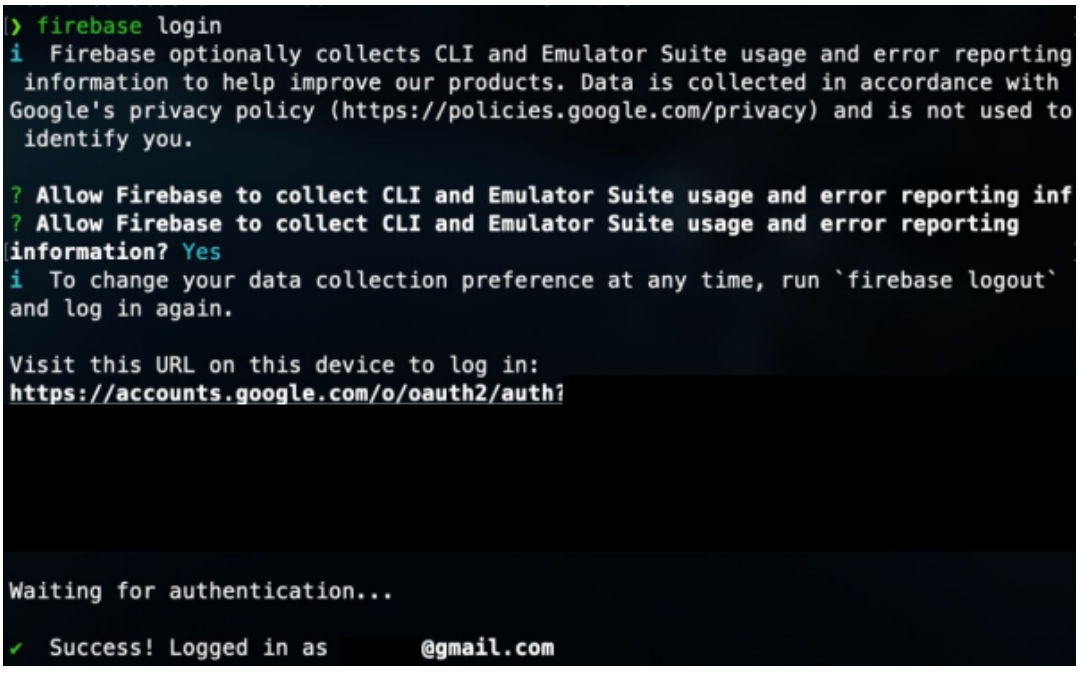


Рисунок 3.2 — Аутентифікація у firebase-tools

Якщо проєкт на вебсайті Firebase було створено попередньо до ініціалізації проєкту локально, його можна використовувати за допомогою

команди “firebase use <ідентифікатор проєкту>”, яка призначає обраний додаток до поточної теки у якій виконано команду.

3.2.2 Ініціалізація проєкту Firebase

Перш ніж використовувати Firebase CLI для роботи з проєктом, необхідно ініціалізувати Firebase проєкт на робочому пристрої. Для цього потрібно запустити термінал, перейти до теки проєкту і ввести команду “firebase init”. Ця команда почне процес налаштування Firebase проєкту та дозволить вам вибрати потрібні сервіси Firebase, такі як хостинг, база даних, аутентифікація тощо.

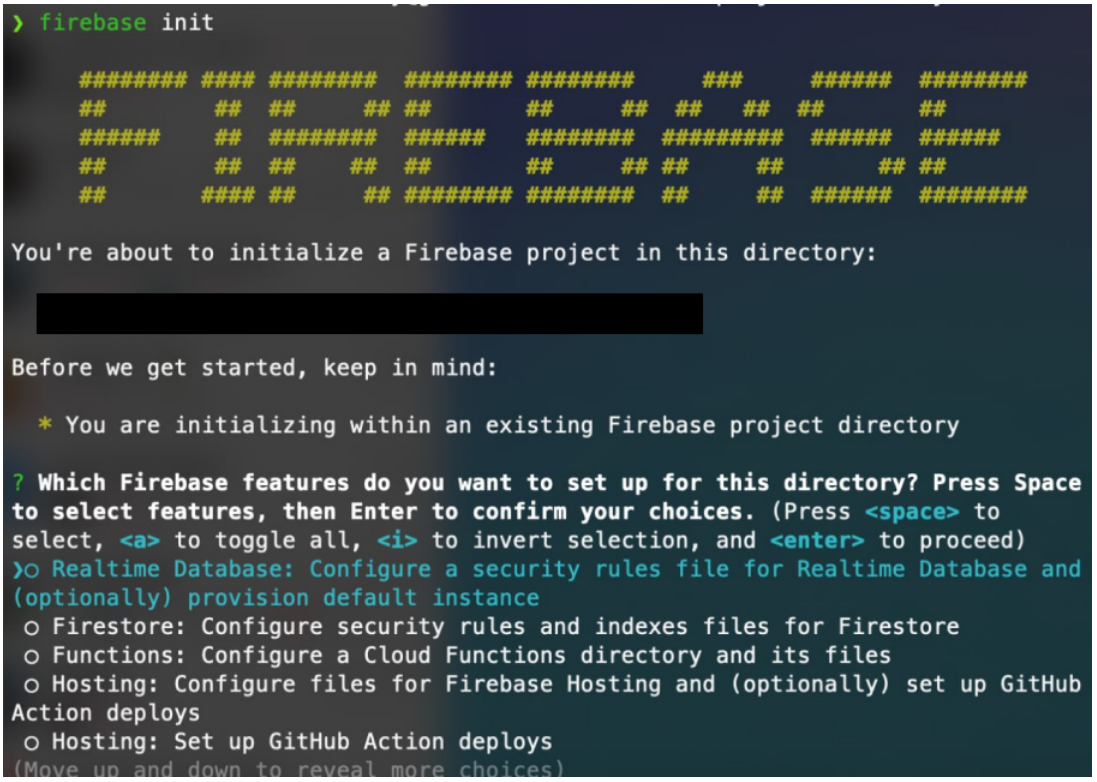


Рисунок 3.3 — Налаштування проєкту Firebase

Після вибору необхідних сервісів для додатка, створити новий проєкт, та провести повне налаштування необхідних сервісів. Дане налаштування містить у собі вибір з декількох регіонів, де можна ініціалізувати сервіси, оскільки це напряму впливає на швидкодію застосунку, то необхідно обирати найближче до потенційних користувачів.

Далі відбувається налаштування сервісів Firebase, а саме, правил приватності Firestore та Realtime баз даних, провести обрання мови програмування з доступних опцій та обрати які саме емулятори треба запускати. Станом на 05.28.2023 Firebase пропонує емулятори таких сервісів як Authentication, Functions, Firestore, Realtime Database, Hosting, Pub/Sub та Storage.

Оскільки необхідно налаштувати три мікросервіси (profile, media, authentication), то дану процедуру необхідно виконати тричі, але створювати проєкт у системі Firebase доведеться лише при першому налаштуванні. Також для реалізації мікросервісу authentication було обрано мову програмування Python версії “3.10”, а для двох інших мікросервісів - TypeScript версії “4.9.5”. Firebast-tools автоматично проведе налаштування необхідних пакетів для початку розробки, що пришвидшує даний процес, та є зручнішим для розробника.

3.2.3 Створення та конфігурація Firebase Functions

Firebase підтримує одразу дві варіації конфігурації проєкту з мікросервісами. Перший спосіб це класичний варіант розподілу мікросервісів, де розробник матиме окремий репозиторій пов’язаний з кожним мікросервісом, а другий це налаштування одного репозиторію, використовуючи декілька мікросервісів, тобто другий спосіб більше схожий на монолітну архітектуру.

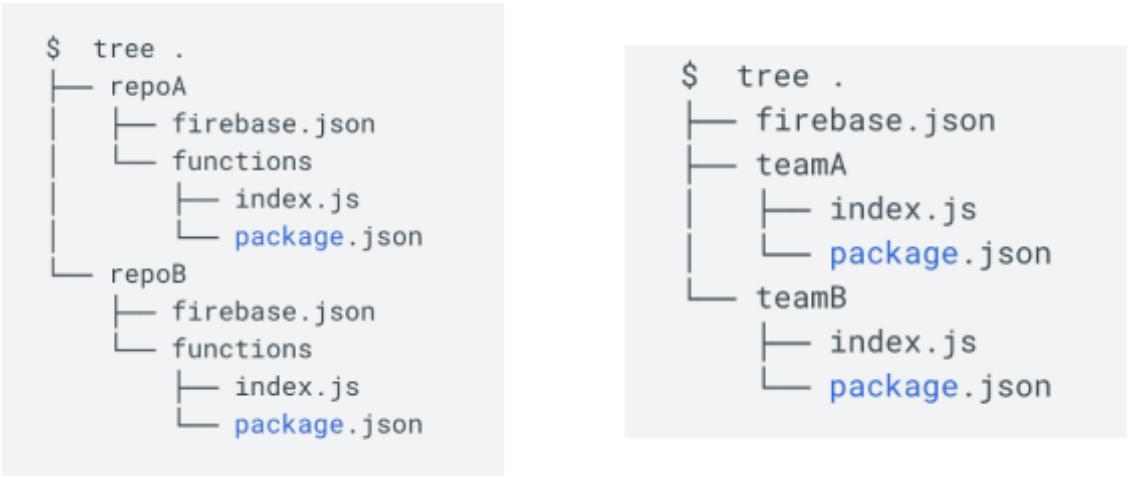


Рисунок 3.4 — Порівняння двох можливих варіантів налаштування мікросервісів

Файл firebase.json, у якому зберігається конфігурація повинен бути розміщений в кореневому каталозі проєкту Firebase. Даний файл можливо створити власноруч або використовувати команду “firebase init” для ініціалізації проєкту та автоматичного створення файлу firebase.json з деякими базовими налаштуваннями.

Звісно при виконанні вищезазначеної команди тричі у трьох різних мікросервісах, буде створено три файли firebase.json, що є більш сприятливим для мікросервісної архітектури. Також у даних файлах зберігаються параметри для запуску необхідних емуляторів, а саме порт до якого вони під’єднуються, версію середовища виконання та інші додаткові параметри.

Для завершення ініціалізації та доступу до комплекту для розробки програмного забезпечення Firebase admin, необхідно провести налаштування сервісного акаунту проєкту. Це можна виконати у вебдодатку Firebase console, у налаштуваннях проєкту де необхідно перейти до сторінки “Service accounts” та згенерувати приватний ключ доступу натиснувши “Generate new private key”. Бажано зробити окремий ключ для кожного мікросервісу. Без даних конфігурацій Firebase відхилятиме запити, які надсилатимуть розроблені функції.

3.2.4 Розробка програмного коду

Оскільки у додатку заплановано реалізувати три мікросервіси, необхідно визначити архітектуру для їх реалізації. Усі сервіси базуватимуться на архітектурі REST (передача репрезентативного стану), яка є стилем розробки веб додатків на основі протоколу HTTP. За необхідністю, дані мікросервіси зможуть взаємодіяти як між собою, так і з іншими сервісами за допомогою HTTP запитів. REST є простішим у використанні, та більш сучасним варіантом ніж архітектурний стиль SOAP та дозволяє робити більш гнучкі та масштабовані додатки.

Мікросервіс authentication має виконувати лише дві дії, а саме реєстрацію нового користувача та аутентифікацію вже створеного. Ці дії винесені до окремого сервісу, оскільки типово є тісно пов’язані в інтерфейсі користувача. Для того, щоб Firebase розпізнав функції що розробляються, необхідно об’єднати їх у кінцеві точки за допомогою мікрофрейморку Flask. Ось як це виглядатиме:

```
import firebase_admin
from firebase_functions import https_fn
from flask import Flask, request, jsonify, make_response
app = Flask(__name__)
cred = credentials.Certificate('./configs/service_account.json')
firebase_admin.initialize_app(cred)
@app.post('/login')
def login():
 ...
@app.post('/register')
def register():
 ...

@https_fn.on_request(region='europe-west1')
def authentication(req: https_fn.Request) -> https_fn.Response:
   with app.request_context(req.environ):
      return app.full_dispatch_request()
```

У даному прикладі, Firebase лише є надбудовою до вже функціонального мікрофреймворку, що дуже спрощує розробку на дану систему.

Під час розробки даного мікросервісу, було визначено суттєві недоліки сервісу Firebase які стосуються підтримки мови програмування Python. Головною проблемою виявилась неможливість імпортування локально створених модулів, тобто реалізацію усього мікросервісу довелося виконувати в одному файлі, що унеможливлює використання типових архітектурних патернів, але у документації зазначено, що така підтримка має бути реалізована у майбутньому командою Google. Ця проблема не є значною, адже даний мікросервіс має лише дві основні функції, але потенційно може призвести до зменшення можливості масштабувати даний сервіс.

З процесом розробки двох інших мікросервісів не виникло даної проблеми, адже Firebase однозначно має кращу підтримку мов програмування JavaScript та Typescript. Для реалізації даних мікросервісів було використано архітектурний шаблон MVC (Модель, вигляд, контролер), який розділяє код на три загальні частини зі своїми окремими функціями, а саме:

— View (вигляд) - будь-яке представлення інформації, яке отримає користувач на виході. Вигляд може мати кілька взаємопов'язаних областей, наприклад, різні таблиці та поля форми, в яких відображаються дані.

— Controller (контролер) - інтерфейс взаємодії та обробки даних між View (виглядом) та Model (моделлю). Функціональність контролерів охоплює відстеження певних подій, які відбуваються внаслідок дій користувача. Контролери дають змогу структурувати код, групуючи пов'язані дії в окремі класи. Наприклад, типовий MVC проєкт може мати контролер користувача, що містить групу методів, пов'язаних з управлінням обліковим записом

користувача, таких як реєстрація, аутентифікація, редагування профілю, зміна пароля тощо.

— Model - модель даних яка є центральним компонентом шаблону та є незалежною від інтерфейсу користувача. Модель містить у собі ядро даних та основні функції їх обробки й не залежить від процесів введення та виведення даних.

Аналогічно до Flask (у випадку з Python), з TypeScript необхідно використати модуль express, якій впорається з налаштуванням кінцевих точок у даній архітектурній реалізації. Його налаштування відбувається наступним чином:

```
import express from 'express';
import config from './config/config';
import media from './routes/media';
import * as functions from 'firebase-functions';

const admin = require('firebase-admin');
// express app
const router = express();
// Parse the request
router.use(cookieParser());
router.use(express.json());
router.use(express.urlencoded({ extended: true }));
router.use('/health', health);
router.use('', media);
exports.media = functions.region('europe-west1').https.onRequest(router);
```

Таким чином, щоб Firebase Functions розпізнав весь мікросервіс у вигляді exress api, все що необхідно зробити для доступу до його кінцевих точок - це додати експортування функції, яка безпосередньо реагує на HTTP запити.

Використання REST у поєднанні з архітектурою MVC дає змогу розділити логіку сервера на окремі компоненти, що спрощує розробку та супровід системи. Це забезпечує гнучкість, масштабованість і повторне

використання коду. REST спрощує комунікацію між клієнтом і сервером, даючи їм змогу взаємодіяти з ресурсами за допомогою стандартних методів HTTP. Використовуючи MVC, можна чітко розділити бізнес-логіку, уявлення та управління даними. Поділ бізнес-логіки, представлення та управління даними. Разом вони створюють ефективну та легко масштабовану систему.

Функціональні вимоги до мікросервісів:

а) мікросервіс “authentication”:

1) реєстрація користувача:

1.1) можливість створення нового користувача;

1.2) HTTP POST запит який приймає у тілі json з параметрами email, first_name, last_name, password, repeat_password, is_author;

1.3) валідація отриманих параметрів, перевірка збігу password та repeat_password;

1.4) створення запису користувача у сервісі Firebase Authentication;

1.5) створення запису користувача у базі даних Firebase Authentication у колекції “users”;

2) аутентифікація користувача

2.1) HTTP GET запит, який приймає у параметрах зміну authorization, яка являється Firebase idToken;

2.2) валідація отриманого Firebase idToken;

2.3) генерація токена для аутентифікації за допомогою firebase admin;

б) мікросервіс “profile”:

1) отримання даних облікового запису;
2) оновлення даних облікового запису;

3) видалення даних облікового запису;
4) зміна пароля облікового запису;

в) мікросервіс “media”

1) отримання усіх медіа, отримання медіа за id, отримання медіа на які користувач має підписку;
2) створення нового медіа;
3) підписка та відписка на оновлення медіа, з відправленням повідомлення на електронну скриньку;

Запустити та протестувати функціонал у реальному часі допоможе команда “firebase emulators:start”, після чого, за умови правильного налаштування, Firebase має запустити усі попередньо налаштовані емулятори, які доступні у відповідних до вказаних у файлі firebase.json портах.

Відсутність пошуку у базі даних Firebase, не є остаточно вірним формуванням проблеми, оскільки вона все ж має такі запити як orderBy() - впорядкування результатів по параметру, та where() - пошук по параметру, або декільком параметрам. Але згідно з документацією Firebase [12], даний запит є дуже обмеженим у можливостях та потребує додаткової побудови індексів для оптимізації. Метод where() приймає три параметри: поле для фільтрації, оператор порівняння та значення. Cloud Firestore підтримує наступні оператори порівняння:

- **<** — менше ніж
- **<=** — менше або дорівнює
- **==** — дорівнює
- **>** — більше ніж
- **>=** — більше або дорівнює
- **!=** — не дорівнює
- **array-contains** — масив містить

- **array-contains-any** — масив містить будь-що з
- **in** — попарне порівняння між параметром за набором значень за оператором “==”
- **not-in** — попарне порівняння між параметром за набором значень за оператором “**!**=”

Слід також зазначити, що дані оператори мають обмеження у тому, як їх можна між собою поєднувати. Наприклад, за один запит, можна зробити максимум 30 порівнянь. Також, з даними операціями, через обмеження неможливо виконати мультитекстовий пошук, це означає що за один запит можна використати оператори “<=” та “>=” лише на одному параметрі. Додаючи до вище оговореного, дані оператори не дозволяють виконати пошук підстроки з текстового рядка, тож “шукати” таким запитом, можна лише з початку рядка. Було досліджено три можливі підходи даної проблеми. Перший з них, та найефективніший — це під'єднати одне з розширень для повноцінного пошуку, але вони є платними. Другий підхід, це побудова окремої колекції для пошуку, колекція містить у собі документи, у яких полями виступають розбиті посимвольно рядки, даний підхід також значно підвищує витрати на підтримку такої бази даних. Останній варіант, який саме був використаний у роботі, піти на компроміс, але понизити витрати на підтримку бази даних. Перше, що треба було зробити, це продублювати параметри, по яких необхідно виконувати пошук, у нижньому регістрі. Друге — розробити функцію, яка дозволяє одночасно пошук по одному обраному полю за допомогою операторів “<=” та “>=”, та використання усіх інших операторів та розбиття отриманих результатів на сторінки (як з пошуком, так і без нього). Розглянемо створену функцію:

```
export const getCollectionWithPagination = async (
    collection: string,
    pageSize: number | undefined,
    lastKey: string | undefined,
    whereEqualQueries: WhereQuery[],
```

```
    orderBy?: OrderBy[],
    whereSearchQuery?: WhereSearchQuery | undefined
)
```

Функція прийматиме такі параметри: назва колекції, розмір однієї сторінки, id останнього документу з попередньої сторінки (невизначений, якщо це перша сторінка), масив з операціями, які не містять у собі оператори “<=” та “>=”, порядок сортування по певним параметрам та об’єкт для пошуку, який будується використовуючи наступну функцію:

```
export const whereSearchQueryConstructor = (field: string, value: any):
WhereSearchQuery => {
    return {
        less: {
            field: field + 'Search',
            operator: '<=',
            value: `${value}~`
        },
        greater: {
            field: field + 'Search',
            operator: '>=',
            value: `${value}`
        }
    };
};
```

У функції getCollectionWithPagination() формується запит на отримання даних з колекції, використовуючи усі вищезазначені фільтри та пошук.

Також, кожен мікросервіс має документуватись за допомогою OpenApi 3.0, даний інструмент є дуже зручним для документування REST додатків та пришвидшує процес розробки для людей, які працюватимуть з даними мікросервісами. Ось приклад документування таким чином кінцевої точки для отримання медіа, у якому описані параметри, які приймає точка, та відповіді які вона потенційно може надіслати:

```
paths:
  /media:
    get:
      summary: return all media
      operationId: getAllMedia
```

```yaml
parameters:
  - in: query
    name: specialization
    required: false
  - in: query
    name: type
    required: false
  - in: query
    name: region
    required: false
  - in: query
    name: rate
    required: false
  - in: query
    name: title
    required: false
responses:
  200:
    description: success get all media
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/AllMediaGetSuccessResponse'
  500:
    description: unknown error
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/CommonErrorResponse'
```

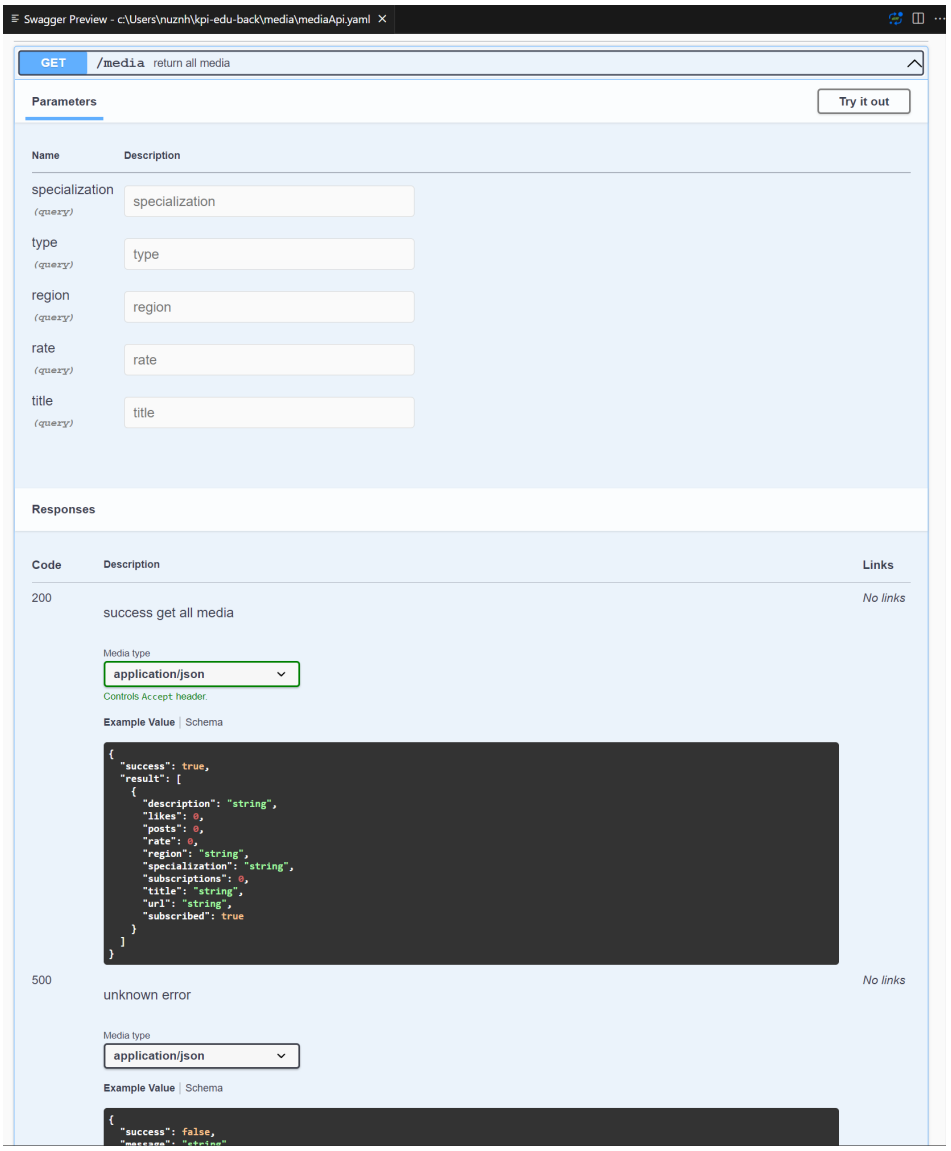


Рисунок 3.6 — OpenApi 3.0 вигляд документації

3.3 Розгортання додатка

Процес розгортання додатка у Firebase є швидким та зручним засобом розміщення проєкт в мережу Інтернет. Виконується розгортання командою “firebase deploy”, що публікує на домен розроблений код, правила для баз даних та файлосховища.

Після розгортання застосунку на платформі Firebase існує кілька можливих інструментів налаштування та моніторингу для забезпечення оптимальної продуктивності та стабільності застосунку.

Одним із перших кроків після розгортання є ввімкнення логування Firebase. Ведення журналу дає змогу отримати детальну інформацію про продуктивність застосунку та виявити можливі помилки. Повідомлення про помилки, діагностичні журнали та інші корисні дані можна переглядати, щоб допомогти виявити й усунути проблеми з додатком.

Ще одним важливим аспектом є моніторинг продуктивності - Firebase надає можливість відстежувати показники CPU, RAM та інших ресурсів програми. Це допомагає визначити ефективність застосунку та ухвалити своєчасні рішення щодо оптимізації або масштабування. Firebase Performance Monitoring також може бути використаний для вимірювання продуктивності окремих компонентів програми. Цей інструмент дає змогу відстежувати час відгуку, завантаження сторінок та інші показники продуктивності. Збираючи дані про продуктивність застосунку, можна виявити проблемні області та вжити відповідних заходів. Firebase також дає змогу налаштовувати сповіщення про помилки та незвичайні події. Налаштувавши правила сповіщень, ви можете отримувати сповіщення у разі виникнення певних типів помилок або несподіваних подій. Це дасть змогу знати, що відбувається у застосунку в режимі реального часу.

Ще одна важлива функція — моніторинг бази даних Firebase: моніторинг баз даних Firebase Realtime Database або Firestore. За допомогою моніторингу є змога зібрати статистику використання бази даних, перевірити наявність непотрібних запитів і оптимізувати обробку даних.

Тож, засоби розгортання застосунку на Firebase пропонують широкий спектр можливостей налаштування та моніторингу. Журнали, показники продуктивності, сповіщення та моніторинг бази даних — усі ці інструменти

дають змогу вам контролювати та покращувати продуктивність програми що розробляється. Ці інструменти забезпечують безперебійну та стабільну роботу програмного забезпечення на платформі Firebase, а у випадку виникнення проблем, допомагають якомога швидше їх позбутися.

3.4 Тестування отриманого застосунку

3.4.1 Ручне тестування

Ручне тестування REST API мікросервісу Firebase Emulators - це ефективний спосіб перевірити поведінку трьох мікросервісів: медіа, аутентифікації та профілів. Для цього використовуйте інструмент Postman, який дає змогу надсилати запити й отримувати відповіді від кожного з цих сервісів.

Наступним кроком буде тестування мікросервісу authentication, який відповідає за автентифікацію користувачів - Postman дає змогу користувачам надсилати запити на реєстрацію нових користувачів та вхід до системи. Тестування має гарантувати, що нові користувачі створюються успішно, що вони отримують правильний токен доступу після входу в систему, і що процес виходу з системи працює правильно.

Спочатку перевіримо мікросервіс media. Даний сервіс відповідає за роботу з медіаресурсами; Postman дає змогу надсилати запити на завантаження, отримання, видалення та зміну статусу підписки на джерела інформації. Тестування має перевірити, що файли можуть бути успішно завантажені на сервер, що відповіді, які містять посилання на завантажені файли, повертаються правильно, і що видалення файлів відбувається без проблем.

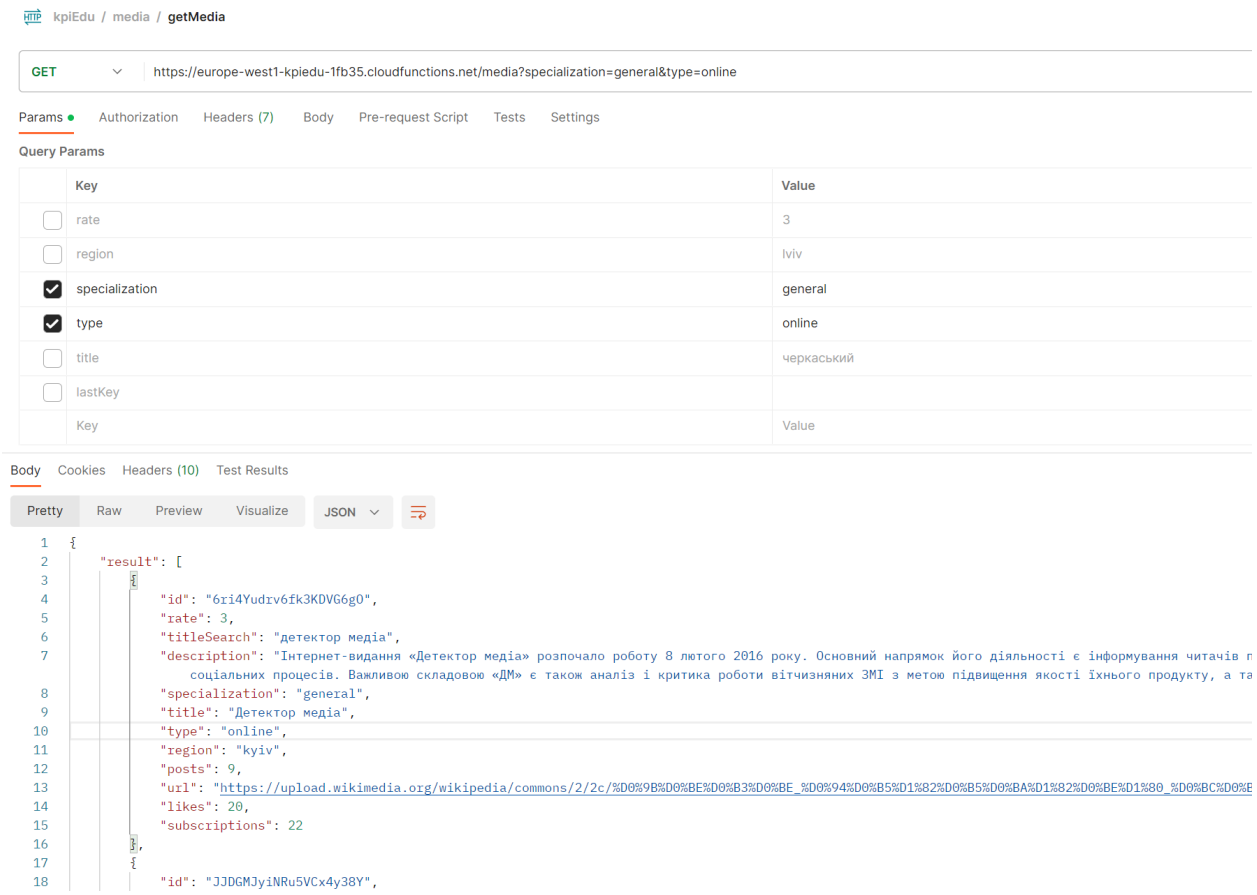


Рисунок 3.6 — Приклад GET запиту для отримання медіа з фільтрами

Нарешті, протестуймо мікросервіс “profile”, що відповідає за профілі користувачів - Postman може використовуватися для надсилання запитів на отримання, оновлення або видалення профілів користувачів. Тест має підтвердити, що інформацію про профіль користувача успішно отримано, профіль коректно оновлено і профіль видалено без проблем. Також слід обробляти можливі помилки, наприклад про не знайдений обліковий запис користувача

Також перевірено сервіс на стійкість до спамових атак, такі атаки були особливо популярні на початку активної фази війни між Україною та Росією, тож на думку авторів дослідження, стійкість до них є важливим критерієм вибору сервісу для будь-якої сфери розробки. Тестування проводилось наступним чином, розроблено програму, яка надсилає певну кількість запитів на сервер та зазначає час відповіді.

```
import axios from "axios"
let results = [];
const testDDOSAttack = async () => {
    const reqStart = performance.now()
    return
axios.get('https://europe-west1-kpiedu-1fb35.cloudfunctions.net/media?specializ
ation=general&type=online').then((res) => {
        const reqEnd = performance.now()
        results.push(reqEnd-reqStart)
        return reqEnd-reqStart
    })
}

// Call the async functions concurrently
const functionCalls = Array.from({ length: 5000 }, () => testDDOSAttack());
const start = async () => {

    await Promise.all(functionCalls)
    .then((time) => {

      console.log('All async functions completed.');
    })
    .catch((error) => {
      console.error('An error occurred:', error);
    });
}

await start()
for (let index = 0; index < results.length; index++) {
    const element = results[index];
    console.log(element)
}
console.log(results.length)
```

За даними такого тесту було отримано наступні результати:

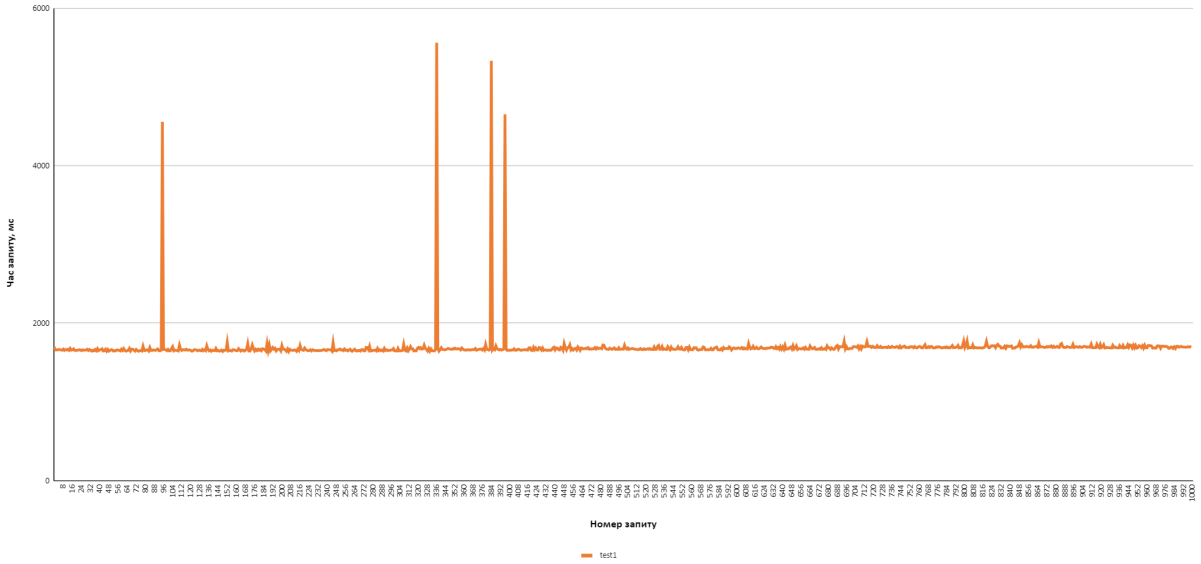


Рисунок 3.7 — Тестування 1000 запитів

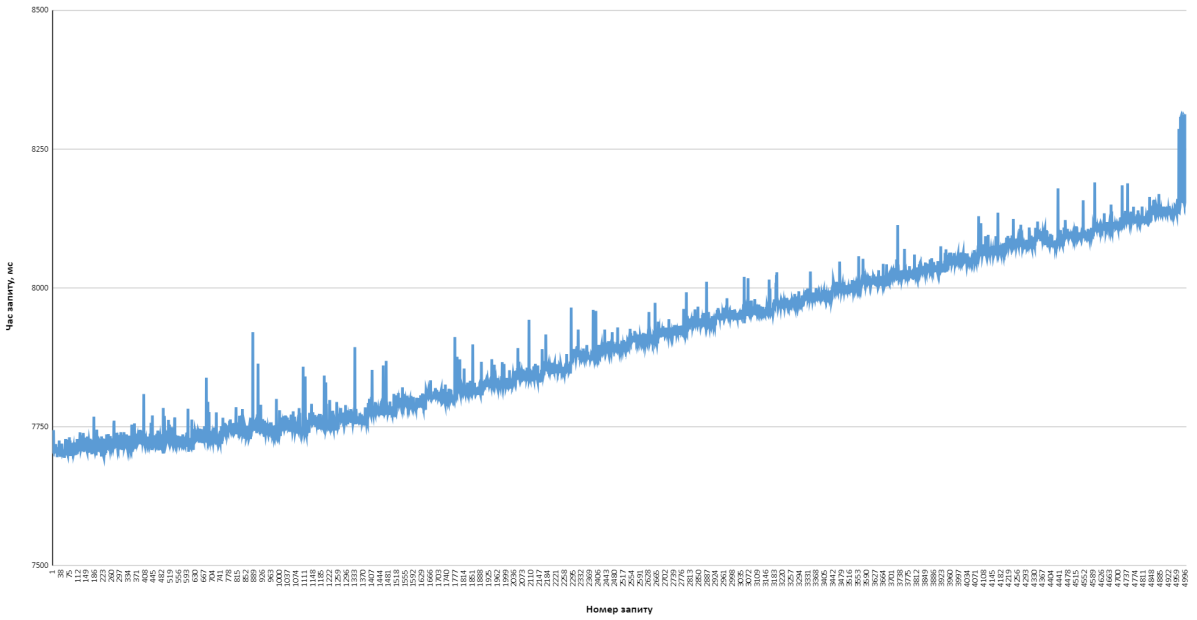


Рисунок 3.8 — Тестування 5000 запитів

Як видно з наведених вище тестів, під час великої кількості запитів з одного клієнту — час на відповідь з нього значно зростає, також, тест на 10000 запитів провалився з помилкою, закрив з’єднання. За наслідками такого тестування, усі сервіси Google помістили IP системи у низьку пріоритетність,

через що навіть однойменний пошуковик перестав відповідати. Тестування на захищеність від DDoS атак можна вважати пройденим.

Загалом, ручне тестування REST API мікросервісів Firebase Emulators за допомогою Postman дає змогу перевірити поведінку усіх трьох мікросервісів окремо. Тести мають перевірити різні функції кожного сервісу та переконатися, що вони працюють правильно. Такий підхід допомагає забезпечити якість і надійність мікросервісів що розробляються.

3.4.2 Тестування за допомогою Firebase

Для тестування мікросервісних додатків Firebase має ще один сервіс, який зветься A\B Testing. A/B-тестування — це саме те, як воно звучить: припустимо існує дві версії елемента (A і B) і метрика, яка визначає успіх. Щоб визначити, яка версія краща, необхідно піддати обидві версії одночасному експериментуванню. Зрештою, відбувається вимірювання, яка версія була більш успішною, і обирається для використання в реальному світі.

Під час тестування REST API мікросервісу застосунку Firebase за допомогою Firebase A/B тестування необхідно ефективно визначити найкращу реалізацію функціонала кожного з трьох мікросервісів: media, authentication та profile.

Firebase A/B тестування дає змогу одночасно експериментувати з різними варіантами реалізації для певних груп користувачів. Це дає змогу отримати значущі дані про те, які варіанти є більш привабливими та ефективними для користувачів.

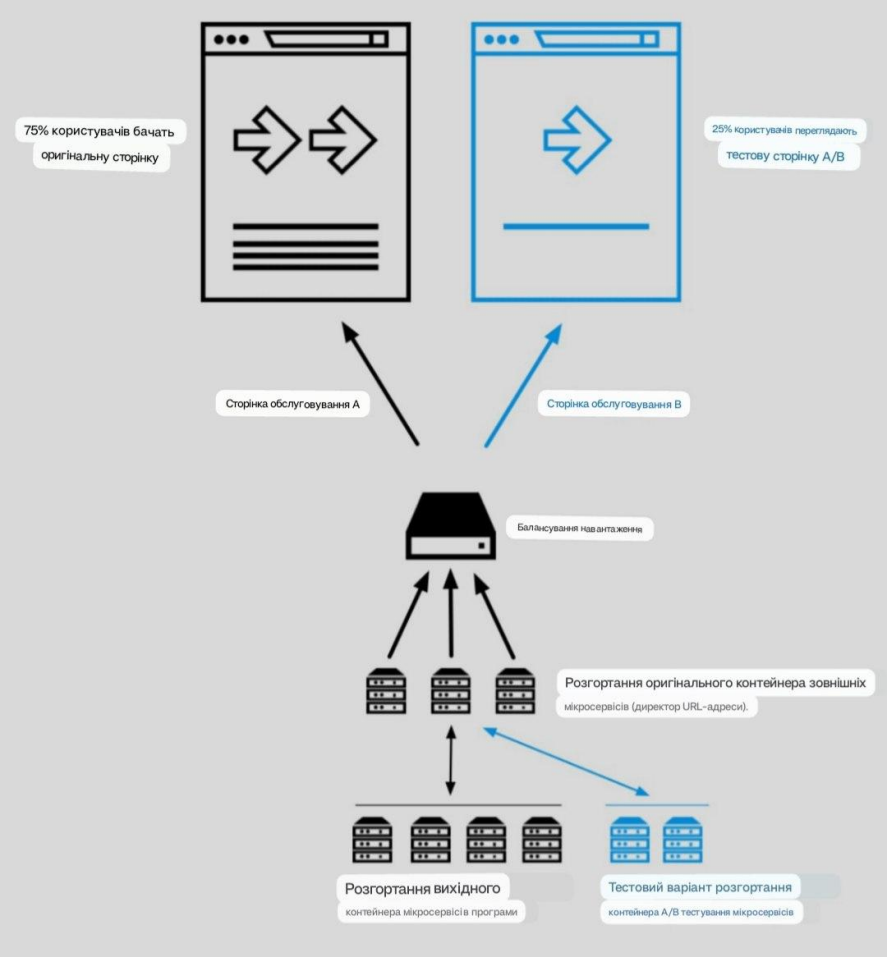


Рисунок 3.9 — A/B тестування мікросервісного додатка [14]

Firebase A/B тестування дозволяє встановлювати різні функціональні опції для кожного мікросервісу. Після початку експерименту Firebase A/B Testing автоматично розподіляє користувачів між різними варіантами та збирає дані про їхню поведінку та реакцію на кожен варіант. Потім зібрані дані можна проаналізувати за допомогою Firebase Analytics, щоб з'ясувати, який варіант найкраще відповідає потребам користувачів.

Firebase A/B Testing пропонує можливість проводити A/B тестування за допомогою Firebase Remote Config. Це дає змогу динамічно змінювати конфігурацію вашого мікросервісного застосунку REST API без необхідності випускати нову версію застосунку. Можна випробувати різні конфігурації, увімкнути або вимкнути певні функції та проаналізувати їхній вплив на користувачів.

Firebase A/B тестування надає можливість об'єктивно протестувати різні варіанти реалізації REST API мікросервісу вашого застосунку на платформі Firebase. Такий підхід дає змогу підвищити якість і ефективність мікросервісів, враховуючи реакцію користувачів і забезпечуючи більш зручний і комфортний досвід.

4. ФУНКЦІОНАЛЬНО ВАРТІСНИЙ АНАЛІЗ РОЗРОБЛЕНОГО ДОДАТКА

На даному етапі дипломної роботи, необхідно розглянути основні характеристики майбутнього програмного продукту, призначеного для проведення демографічних опитувань. Основною метою поставлено забезпечення якісної реалізації всіх необхідних опитувань для висвітлення тем, що становлять інтерес не лише для України, але й для всього світу.

У дослідженні також розглядаються різні варіанти реалізації для забезпечення найбільш точної та оптимальної стратегії вибору, що впливає на економічні фактори та сумісність з майбутніми програмними продуктами. Для досягнення цих цілей використовується метод функціонально-вартісного аналізу (скорочено — ФВА).

Функціонально-вартісний аналіз (ФВА) - це метод, який дозволяє оцінити справжню вартість продукту або послуги незалежно від організаційної структури компанії; основною метою ФВА є виявлення можливостей для економії витрат за допомогою підвищення ефективності виробництва та реалізації оптимального співвідношення між споживчою вартістю продукту та витратами на його виробництво. Аналіз базується на економічній, технічній та проєктний інформації.

Алгоритм ФВА включає наступні етапи: визначення послідовності етапів розробки продукту, визначення загальних річних витрат і трудомісткості, виявлення джерел витрат і розрахунок кінцевого кошторису програмного продукту.

4.1 Постанова задачі проєктування

На даному етапі методологія ФВА використовується за для технічного та економічного аналізів розробки системи, що прогнозує та передбачає

показник стійкості фінансових показників. Рішення щодо проєктування та впровадження компонентів, які розробляються, впливають на систему в цілому, і окремі підсистеми повинні їм відповідати. Тому основним аналізом є аналіз функціональності програмних продуктів, призначених для збору, обробки та аналізу даних підприємства.

Список технічних вимог до розробляємого продукту:

— функціонування на сервісах Google Firebase;

— достатня швидкодія розробляємо мікросервісів;

— здібність до простого масштабування та обслуговування;

— мінімізація витрат на розробку та впровадження розробляємого продукту.

4.2 Обґрунтування функцій програмного продукту

Основна функція F_0 — розробка ряду мікросервісів, які дозволяють підтримувати агрегатор медіаресурсів базуючись на базових потребах типових сайтів. При позначенні даної функції як фундаментальної, можливо виділити інші:

— F_1 - обрання архітектури мікросервісів

— F_2 - обрання тарифного плану Firebase

— F_3 - обрання мов програмування мікросервісів та середовища розробки

Можливі реалізації вищезазначених функцій (наведено лише декілька для кожної):

Функція F_1:

1. Мікросервісний агрегатор.
2. Гілкова архітектура.

Функція F_2:

1. Тарифний план “No-cost”.
2. Тарифний план “Blaze”.

Функція F_3:

1. Python та TypeScript, Pycharm та Webstorm.
2. Python та JavaScript, Visual Studio Code.
3. Java, Intellij idea.
4. Python та TypeScript, JetBrains Fleet.

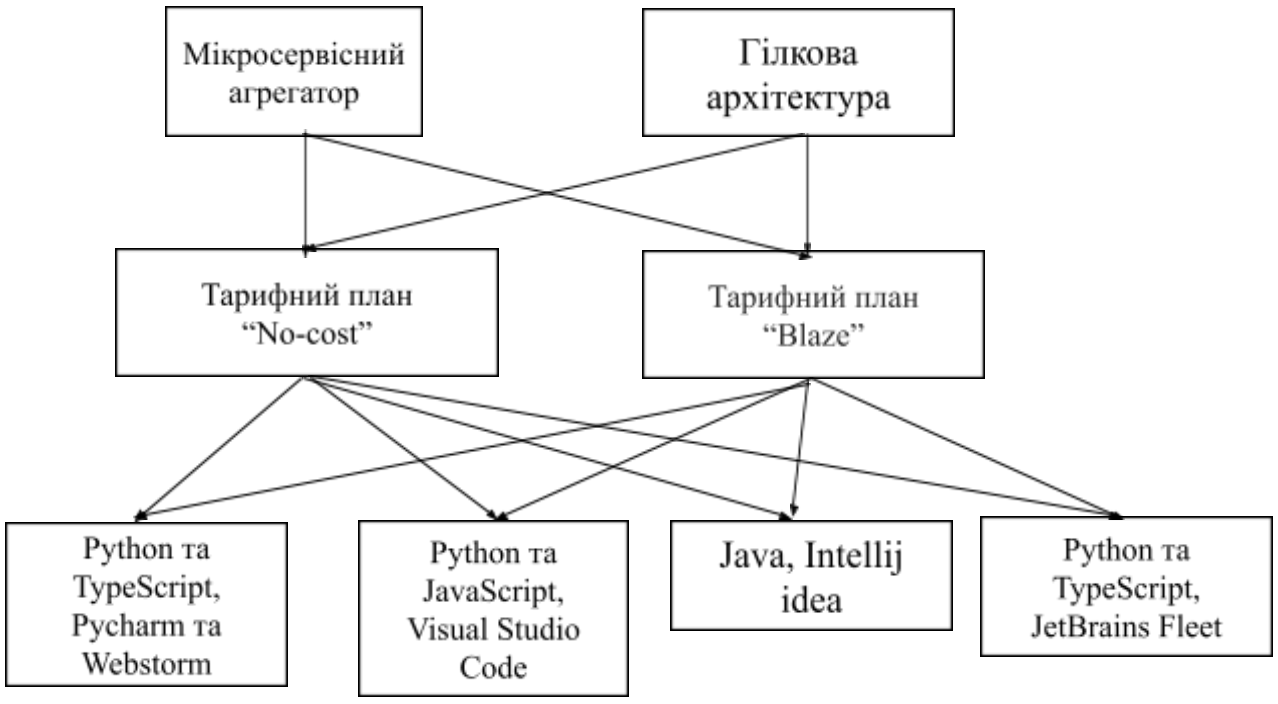


Рисунок 4.1 — Морфологічна мапа

Дана морфологічна карта показує сукупність усіх можливих базових функцій. У наступній таблиці наведено позитивно-негативну матрицю.

Таблиця 4.1 — Позитивно негативна матриця

Функції	Варіанти реалізації	Переваги	Недоліки
F_1	1	Проста реалізація та масштабованість	Відсутність взаємозв'язку між сервісами
	2	Наявність головного мікроконтролеру, який оброблює дані інших	Складніша масштабованість, менша швидкодія
F_2	1	Повністю безплатний	Неможлива розгортка Cloud Functions, але наявний доступ до усіх інших сервісів
	2	Доступ до усього функціоналу Firebase	Щомісячна оплата в залежності від рівня використання
F_3	1	Використання за потреби двох мов програмування, оптимізованих на роботу з вебпроєктами	Використання двох середовищ розробки, одне з яких є платним.
	2	Побудова мікросервісів двома мовами програмування, безплатне середовище розробки	VS code не є повноцінним середовищем розробки, та має менше функціоналу від повноцінних IDE. JavaScript не є типізованою мовою програмування

	3	Java є типізованою мовою програмування, та типовим рішенням для створення мікросервісів	Проблеми з розгортанням Java застосунків на платформі Firebase Functions
	4	JetBrains Fleet підтримує одразу дві мови програмування та є безплатним	JetBrains Fleet також все ще має лише публічну пробну версію та знаходиться у розробці

Беручі за основу аналіз позитивно-негативної матриці, можливо зробити такі висновки, що під час розробки продукту, де які опції необхідно лишити, через те що, вони не відповідають поставленим перед даним продуктом вимогам. Дані варіанти відзначені у морфологічній карті.

Функція F_1:

Необхідно надати перевагу архітектурі, яка має більший потенціал до масштабування та швидкості розробки. Обираємо перший варіант, а другий відкидаємо.

Функція F_2:

У пріоритеті для даного застосунку, використовувати весь можливий потенціал середовища Firebase та сервіси, які він пропонує. Обираємо другий варіант, а перший відкидаємо.

Функція F_3:

Реалізація програми за допомогою двох мов є гарною ідеєю, оскільки окремі мікросервіси не взаємодіють між собою, а деякі сервіси мовою Pyhton

розробити простіше, тож відкидаємо третій варіант. Також надамо перевагу середовищу, яке підтримує одночасно дві мови, тобто відкидаємо перший варіант.

Таким чином, розглянемо наступні варіації реалізації програмного продукту:

$$F_1 1 - F_2 2 - F_3 2,$$

$$F_1 1 - F_2 2 - F_3 4$$

Для того, щоб оцінити якість цієї функції, було обрано систему параметризації, як описано нижче.

4.3 Обґрунтування систем параметрів програмного продукту

На основі даних, описаних вище, було визначено основні параметри відбору, які використовуються для розрахунку коефіцієнтів технологічного рівня.

Для характеристики програмних продуктів використовуються наступні параметри:

— X1 - час обробки даних мовою програмування;

— X2 - час відповіді на REST запит;

— X3 - кількість оперативної пам’яті необхідний для збереження та обробки даних;

— X4 - час побудови проєкту включаючи встановлення залежностей.

Таблиця 4.2 — Параметри програмного продукту

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Час обробки даних мовою програмування	X1	мс	500	350	200
Час відповіді на REST запит	X2	мс	1000	700	400
Кількість оперативної пам’яті необхідний для збереження та обробки даних	X3	мб	256	192	156
Час побудови проєкту включаючи встановлення залежностей	X4	с	85	70	50

За даними таблиці 4.2 будуються наступні графічні характеристики параметрів зображені на рисунках 4.2 — 4.5

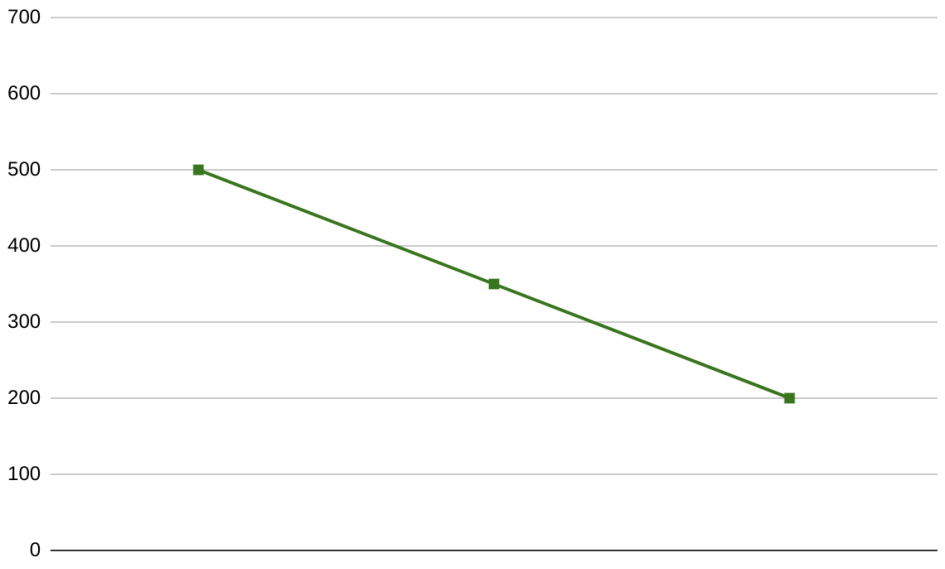


Рисунок 4.2 — X1, час обробки даних мовою програмування

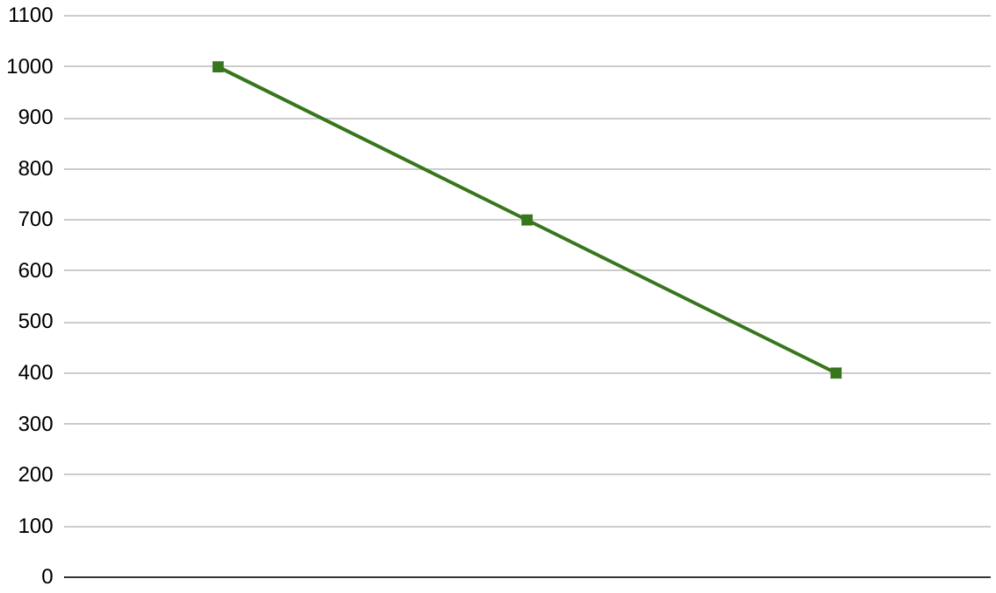


Рисунок 4.3 — X2, час відповіді на REST запит

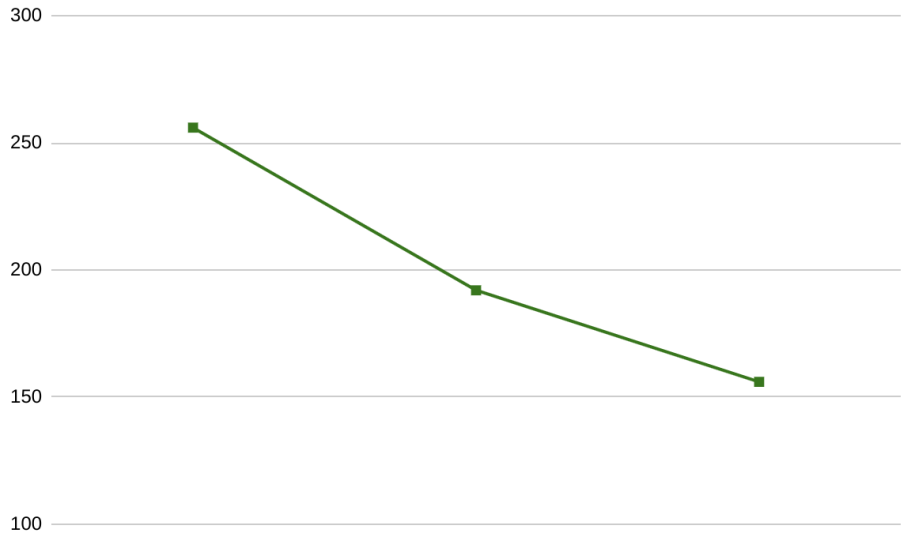


Рисунок 4.4 — X3, кількість оперативної пам'яті необхідний для збереження та обробки даних

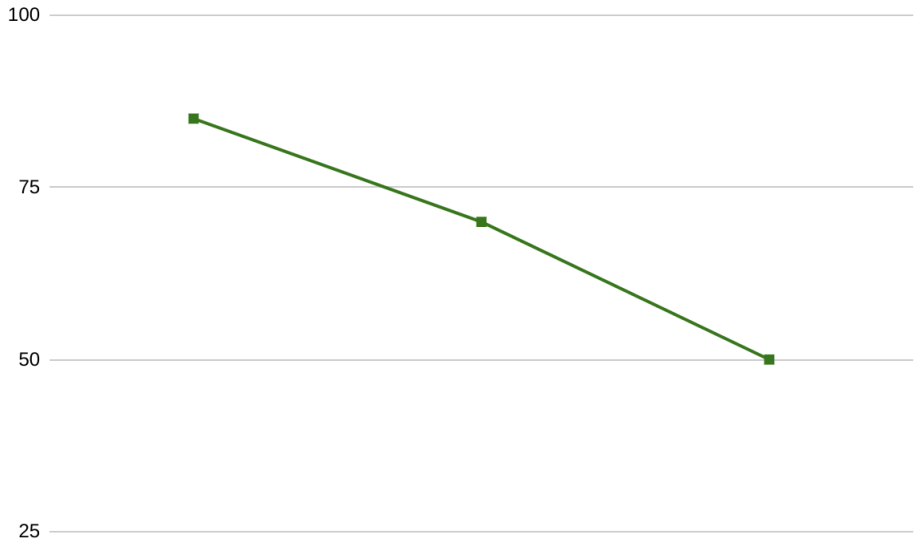


Рисунок 4.5 — X4, час побудови проєкту включаючи встановлення залежностей

4.4 Аналіз експертного оцінювання параметрів

Коли процес детального аналізу буде завершений, експерти мають оцінити ступінь значущості кожного з параметрів для конкретно поставленої цілі, а саме для розробки програмного продукту, який надає найбільш швидкодійні та зручні для обробки результати при REST запитах які мають обробляти мікросервіси що розробляються.

Важливість (значущість) кожного з параметрів визначається методом попарного порівняння. Дану оцінку проводить комісія експертів з 7 людей. Визначення коефіцієнтів важливості (або значущість) передбачає у собі:

— Визначити рівень значущості параметрів шляхом присвоєння різних ступенів;

— Перевірити придатність для подальшого використання експертної думки;

— Визначити бінарну пріоритетність параметрів;

— Обробка результатів та визначення коефіцієнтів значущості.

Таблиця 4.3 — Результати експертного ранжування

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Час обробки даних мовою програмування	мс	3	3	4	4	4	3	3	24	-6.5	42.25
X2	Час відповіді на REST запит	мс	4	4	3	3	3	4	4	25	-7.5	56.25
X3	Кількість оперативної пам’яті необхідний для збереження та обробки даних	мб	2	1	2	2	2	1	1	11	6.5	42.25
X4	Час побудови проєкту включаючи встановлення залежностей	с	1	2	1	1	1	2	2	10	7.5	56.25
	Разом		10	10	10	10	10	10	10	70	0	197

Для того, щоб визначити ступінь надійності експертної думки, були визначені наступні параметри

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70, \ \#(4.1)$$

де N — число експертів та n – кількість параметрів;

б) середнє арифметичне рангів:

$$T = \frac{1}{n} R_{ij} = 17,5. \ \#(4.2)$$

в) відхилення суми рангів кожного параметра від середньо арифметичного по рангах:

$$\Delta_i = R_i - T. \ \#(4.3)$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г) загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 197. \ \#(4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2\left(n^3 - n\right)} = \frac{12 \cdot 197}{7^2\left(4^3 - 4\right)} = 0,804 > W_k = 0,67. \ \#(4.5)$$

Отримані коефіцієнти узгодженості вищі за стандартний коефіцієнт 0,67, що означає, що цей рейтинг можна вважати надійним.

За допомогою отриманих результатів ранжування, можна провести необхідне попарне порівняння усіх параметрів та занести їх у таблицю 4.4.

Таблиця 4.4 — Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	<	<	>	>	>	<	<	<	0,5
X1 і X3	>	>	>	>	>	>	>	>	1,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	>	>	>	>	>	>	>	>	1,5
X2 і X4	>	>	>	>	>	>	>	>	1,5
X3 і X4	<	>	<	<	<	>	>	<	0,5

Таблиця 4.4 – Попарне порівняння параметрів.

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається за формулою:

$$a_{ij} = \{1.5 \text{ при } X_i > X_j \; 1.0 \text{ при } X_i = X_j \; 0.5 \text{ при } X_i < X_j . \#(4.6)$$

З отриманих числових оцінок переваги складемо матрицю A=‖aij‖.

Для кожного параметра необхідно провести розрахунок вагомості $K_{ві}$ за наступними формулами:

$$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \#(4.7)$$

$$b_i = \sum_{i=1}^{N} a_{ij} \#(4.8)$$

Відносна оцінка розраховується кілька разів, поки наступне значення не буде незначно відрізнятися (<2%) від попереднього; на другому і наступних кроках відносна оцінка розраховується за такими формулами:

$$K_{\text{ві}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \#(4.9)$$

$$b_i^{'} = \sum_{i=1}^{N} a_{ij} b_j \#(4.10)$$

Виходячи з таблиці 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тож більшої кількості ітерації проводити немає потреби

Таблиця 4.5 – Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1	0,5	1,5	1,5	4,5	0,281	16,25	0,275	59,125	0,274
X2	1,5	1	1,5	1,5	5,5	0,344	21,25	0,360	77,875	0,361
X3	0,5	0,5	1	0,5	2,5	0,156	9,25	0,157	34,125	0,158
X4	0,5	0,5	1,5	1	3,5	0,219	12,25	0,208	44,875	0,208
Всього:					16	1,000	59	1,000	216	1,000

4.5 Аналіз рівня якості варіантів реалізації функцій

Рівень якості кожного варіанту основної функціональності оцінюється окремо.

Абсолютні значення параметрів, X1 (Час обробки даних мовою програмування), X3 (кількість оперативної пам’яті необхідний для збереження та обробки даних), та X4 (час побудови проєкту включаючи встановлення залежностей) відповідають технічним вимогам до функціональності програмного забезпечення. Абсолютне значення параметру X2 (час відповіді на REST запит) не є найгіршим.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується таким чином (таблиця 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{\text{в}i,j} B_{i,j}, \ \#(4.11)$$

де n – кількість параметрів;

$K_{\text{в}i}$ – коефіцієнт вагомості i–го параметра;

B_i – оцінка i–го параметра в балах.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП.

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	1	X3	168	5	0,158	0,79
F2	2	X4	65	5	0,208	1,04
F3	2	X1	373	3	0,274	0,822
	2	X2	451	4	0,361	1,444
	4	X1	347	3	0,274	0,822
	4	X2	420	3	0,361	1,083

За даними з таблиці 4.6 за формулою:

$$K_K = K_{\text{ТУ}}\left[F_{1k}\right] + K_{\text{ТУ}}\left[F_{2k}\right] + \ldots + K_{\text{ТУ}}\left[F_{zk}\right], \ \#(4.12)$$

Визначимо рівень якості кожного з варіантів:

$$K_{K2} = 0{,}79 + 1{,}04 + 0{,}822 + 1{,}444 = 4{,}096,$$

$$K_{K4} = 0{,}79 + 1{,}04 + 0{,}822 + 1{,}083 = 3{,}735$$

З даних розрахунків, можна дійти висновку, що кращим є другий варіант, у якого коефіцієнт технічного рівня є більшим.

4.6 Економічний аналіз варіантів розробки ПП

Щоб визначити вартість розробки програмного додатка, спочатку розраховується трудомісткість.

Усі варіанти містять два окремих завдання:

1. розробка проєкту програмного продукту;

2. розробка програмної оболонки;

Завдання 1 належить до групи А через свою новизну; завдання 2 належить до групи Б.

У завданні 1 використовується довідкова інформація, у завданні 2 - інформація у форматі даних. Розрахуємо критерії часу розробки та програмування для кожного завдання.

Загальна трудомісткість розраховується наступним чином:

$$T_О = T_Р \cdot К_П \cdot К_{СК} \cdot К_М \cdot К_{СТ} \cdot К_{СТ.М}, \#(4.13)$$

де $T_Р$ – трудомісткість розробки ПП;

$К_П$ – коригувальний коефіцієнт;

$К_{СК}$ – коефіцієнт на складність вхідної інформації;

$К_М$ – коефіцієнт рівня мови програмування;

$К_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$К_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першої задачі, виходячи зі ступеня новизни алгоритму А та часових критеріїв для обчислювальних задач групи складності 1, трудомісткість $T_Р = 55$ людиноднів. Коригувальний коефіцієнт, що враховує тип нормативно-довідкової інформації в першому завданні: $К_П = 1,65$. Коригувальний коефіцієнт, що враховує складність перевірки вхідної та вихідної інформації для всіх семи завдань, виглядає наступним чином $К_{СК} = 1$. Оскільки при розробці перших завдань використовувалися стандартизовані модулі, то коефіцієнти, які це враховують $К_{СТ} = 0,88$. Тоді загальна трудомісткість програмування першої задачі дорівнює:

$$T_1 = 55 \cdot 1,65 \cdot 0,88 = 79,86 \text{ людинодня.}$$

Аналогічні розрахунки слід зробити для наступних завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $T_Р = 27$ людиноднів, $К_П = 0,86$, $К_{СК} = 1$, $К_{СТ} = 0,68$:

$$T_2 = 27 \cdot 0,86 \cdot 0,68 = 15,7896 \text{ людиноднів}$$

Для кожного з обраних варіантів реалізації програми трудомісткість визначається шляхом розрахунку трудомісткості відповідного завдання.

$$T_I = (79,86 + 15,7896 + 4,8 + 15,7896) \cdot 8 = 929,91 \text{ людиногодини.},$$
$$T_{II} = (79,86 + 15,7896 + 6,91 + 15,7896) \cdot 8 = 946,8 \text{ людиногодини.}$$

Найвищу трудомісткість має варіант II.

У процесі розробки беруть участь два розробники, а саме один DevOps інженер з окладом у 25000 гривень, один архітектор з окладом у 21800 гривень, та один backend програміст з окладом у 27500 гривень.

Використаємо наступну формулу для визначення середньої заробітної плати за годину:

$$СЧ = \frac{М}{T_m \cdot t} грн., \#(4.14),$$

де М – місячний оклад працівників;

T_m – кількість робочих днів тиждень;

t – кількість робочих годин в день.

$$СЧ = \frac{25000+21800+37000}{3 \cdot 21 \cdot 8} = 166,27 \text{ грн. } \#(4.15)$$

Також, розрахуємо заробітну плату за наступною формулою:

$$СЗП = С_ч \cdot Т_i \cdot КД, \#(4.16),$$

де $С_ч$– величина погодинної оплати праці програміста;

$Т_i$ – трудомісткість відповідного завдання;

$К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 166,27 \cdot 929,91 \cdot 1,2 = 185539,36$ грн,

II. $С_{ЗП} = 166,27 \cdot 946,8 \cdot 1,2 = 188909,32$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 185539,36 \cdot 0,22 = 40818,66$ грн,

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 188909,32 \cdot 0,22 = 41560,05$ грн.

Визначимо витрати на оплату однієї машино-години. ($С_М$)

Оскільки одна ЕОМ обслуговує одного DevOps інженер з окладом у 25000 гривень., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 25000 \cdot 0,2 = 60000 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 60000 \cdot (1 + 0,2) = 72000 \text{ грн.}$$

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0,22 = 72000 \cdot 0,22 = 15840 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 30% та вартості ЕОМ – 48600 грн:

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1,3 \cdot 0,3 \cdot 48600 = 18954 \text{ грн.},$$

де $К_{ТМ}$ – коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$ – річна норма амортизації;

$Ц_{ПР}$ – договірна ціна приладу.

Витрати на профілактику та ремонт розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1,3 \cdot 48600 \cdot 0,04 = 2527,2 \text{ грн.},$$

де $К_Р$ – відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 12 – 15) \cdot 8 \cdot 0,5 =$$

$$= 936 \text{ години},$$

де $Д_К$ – календарна кількість днів у році;

$Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t – кількість робочих годин в день;

$К_В$ – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 936 \cdot 0,3 \cdot 0,5 \cdot 4,79 = 672,52 \text{ грн},$$

де $N_С$ – середньо-споживча потужність приладу;

$К_З$ – коефіцієнтом зайнятості приладу;

$Ц_{ЕН}$ – тариф за 1 кВт-год електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0,67 = 48600 \cdot 0,67 = 32562 \text{ грн.}$$

Отже, річні експлуатаційні витрати будуть становити:

$$C_{ЕКС} = C_{ЗП} + C_{ВІД} + C_{А} + C_{Р} + C_{ЕЛ} + C_{Н}, \#(4.17),$$

$C_{ЕКС}$ = 72000 + 15840 + 18954 + 2527,2 + 672,52 + 32562 = 142555,72 грн.

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{М\text{-}Г} = C_{ЕКС} / T_{ЕФ} = 142555,72 / 936 = 152,3 \text{ грн/год.}$$

У цьому випадку всі роботи з розробки програмного продукту виконуються на комп'ютері, тому, в залежності від обраного варіанту застосування, витрати на оплату машинного часу:

$$C_{М} = C_{М-Г} \cdot T, \#(4.18),$$

I. $C_{М} = 152,3 \cdot 929,91 = 141625,29$ грн.,

II. $C_{М} = 152,3 \cdot 946,8 = 144197,64$ грн.

Накладні витрати складають 67% від заробітної плати:

$$C_{Н} = C_{ЗП} \cdot 0,67, \#(4.19),$$

I. $C_{Н} = 141625,29 \cdot 0,67 = 94888,94$ грн.,

II. $C_{Н} = 144197,64 \cdot 0,67 = 96612,42$ грн.

Отже, вартість розробки ПП за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{ВІД} + С_{М} + С_{Н}, \#(4.20),$$

I. $С_{ПП}$ = 185539,36 + 40818,66 + 141625,29 + 94888,94 = 462872,25 грн.,

II. $С_{ПП}$ = 188909,32 + 41560,05 + 144197,64 + 96612,42 = 471279,43 грн.

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕР j} = К_{К j} / С_{Ф j}, \#(4.21),$$

$$К_{ТЕР1} = 4,096 / 462872,25 = 0,885 \cdot 10^{-5},$$

$$К_{ТЕР2} = 3,735 / 471279,43 = 0,792 \cdot 10^{-5}$$

З даних коефіцієнтів можна побачити що, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1}$= 0,885 ·10^{-5}.

В результаті проведеного функціонально-вартісного аналізу програмного комплексу, що розробляється, можна зробити висновок, що перший варіант реалізації програмного комплексу є найкращим з тих, що залишилися перший з двох варіантів.

Цей варіант реалізації програмного продукту має такі параметри:

- обрання архітектури мікросервісів: мікросервісний агрегатор;
- обрання тарифного плану Firebase: тарифний план “Blaze”;
- обрання мов програмування мікросервісів та середовища розробки: Python та JavaScript, Visual Studio Code.

4.8 Висновки

У цьому розділі виконано повний функціональний та вартісний аналіз програмного продукту. Також була проведена оцінка основних функцій програмного продукту.

В результаті функціонально-вартісного аналізу програмного комплексу, що розробляється, були визначені основні функції програмного продукту, проведена їх оцінка та знайдені параметри, що їх характеризують. На основі цього аналізу були обрані варіанти реалізації програмного продукту.

ВИСНОВКИ

У даній дипломній роботі, яку було присвячено використанню хмарної платформи Firebase для розробки та розгортання мікросервісів та розробці мікросервісних додатків було проведено аналіз конкурентів Firebase та дослідженню архітектури мікросервісів, розроблено функціональний додаток, що демонструє переваги використання Firebase для побудови інфраструктури мікросервісів.

Дослідження показало, що Firebase є потужною та ефективною хмарною платформою, яка надає широкий спектр інструментів та сервісів для розробки та розгортання мікросервісів. До них відносяться база даних в режимі реального часу, автентифікація користувачів, зберігання файлів, хостинг та сповіщення. Firebase також має високу масштабованість і доступність, що робить її ідеальним вибором для розгортання додатків, заснованих на архітектурі мікросервісів

У першому розділі роботи, було розглянуто фундаментальні поняття мікросервісної архітектури, наведено теоретичні відомості що до сервісних та мікросервісних застосунків, та розглянуто причини їх використання у комерційній розробці програмного забезпечення. Другий розділ повністю присвячено самій платформі Firebase. У ньому продемонстровано широкі можливості даної платформи, доступні сервіси, які можна використовувати під час розробки, тестування, моніторингу та розгортки застосунку. Третій розділ описує процес розробки програмного забезпечення для самої платформи Firebase, налаштування проєкту, баз даних, конфігурації для розгортки, та запуску інструменту, який має назву Firebase Emulators та використовується для розробки додатку на ЕОМ розробника, без використання хмарних технологій. Четвертий розділ містить у собі функціонально-вартісний аналіз розробленого продукту, та описує потенційні витрати на розробку повноцінною командою розробників.

Недоліки використання лише Firebase під час розробки, були розглянуті під час порівняння його з іншим сервісом, а саме з Amazon Web Services Amplify. Головним недоліком було визначено відсутність реалізованого пошуку у базах даних Firestore та Realtime Database, що є доволі дивним, оскільки розробкою хмарного середовища займається команда Google, яка створила найбільшу на дату написання роботи однойменну пошукову систему.

Розроблений додаток для агрегації медіаресурсів показує переваги мікросервісної архітектури, а його розгортка на серверах Firebase, доводить що викладати свої продукти у мережу Інтернет за допомогою даної системи може понизити витрати та оптимізувати процес розробки застосунків, або навіть використовуватись у навчальних цілях.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Amirhosein Taherkordi, Frank Eliassen, Daniel Romero, and Romain Rouvoy RESTful Service Development for Resource-Constrained Environments. University of Oslo, 2011.

2. Daniel R.F. Apolinário, Breno B.N. de França A method for monitoring the coupling evolution of microservice-based architectures. *Journal of the Brazilian Computer Society*, 2021.

3. Keith D.Foote, Dataversity : A Brief History of Microservices URL :

https://www.dataversity.net/a-brief-history-of-microservices (дата звернення 24.05.2023).

4. Global cloud microservices industry market research URL: https://www.industryresearch.biz/global-cloud-microservices-industry-market-15068201 (дата звернення: 27.05.2023)

5. Nicolai M. Josuttis SOA in Practice: The Art of Distributed System Design, 2007. P 107 – 137

6. Pankaj Chougale, Vaibhav Yadav, Dr. Anil Gaikwad Firebase – overview and usage, *International Research Journal of Modernization in Engineering Technology and Science*, 2021.

7. Parth Ghiya TypeScript Microservices: Build, deploy, and secure Microservices using TypeScript combined with Node.js. P 228 – 256

8. Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, C. Mohan Firestore: The NoSQL Serverless Database for the Application Developer *IEEE 39th International Conference on Data Engineering*, 2023

9. Ranganath Manohar Rane, Sandesh Suresh Kadam A Research Paper on Firebase Authentication. *International Journal For Scientific Research & Development*, 2021.

10. Tarek Ziadé Python Microservices Development: Build, test, deploy, and scale microservices in Python. P156 – 185

11. Офіційна документація сервісів Amazon Web Serviced від компанії Amazon URL: https://docs.aws.amazon.com/ (дата звернення 08.06.2023)

12. Офіційна документація сервісів Firebase від компанії Google URL: https://firebase.google.com/docs (дата звернення 01.06.2023)

13. Порівняння REST та SOAP архітектур URL: https://www.dineshonjava.com/soap-vs-restful-microservices/ (дата звернення 29.05.2023)

14. Про A/B тестування мікросервісних додатків URL: https://melv1n.com/ab-testing-guide-product-managers/

15. Про збір та облік єдиного внеску на загальнообов'язкове державне соціальне страхування: Закон Україні від 01.05.2023 р. № 2464-VI. *Відомості Верховної Ради України*, 2011, № 2-3, ст.11

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ (МІКРОСЕРВІС AUTHENTICATION)

./functions/main.py:

```
from functools import wraps
import firebase_admin
from firebase_admin import credentials, auth, firestore
from firebase_admin._auth_utils import EmailAlreadyExistsError
from firebase_functions import https_fn

from flask import Flask, request, jsonify, make_response

import re
from flask_cors import CORS

app = Flask(__name__)
cors = CORS(app)

cred = credentials.Certificate('./configs/service_account.json')
firebase_admin.initialize_app(cred)



def required_params(required):
    def decorator(fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            _json = request.get_json()
            missing = [f'{r}' for r in required.keys()
                       if r not in _json]
            if missing:
                response = {
                    'general_msg': 'missing_params',
```

```
                        'errors': {'missing': missing}
                    }
                    return custom_response(False, response, 400)
                wrong_types = [r for r in required.keys()
                               if not isinstance(_json[r], required[r])]
                if wrong_types:
                    response = {
                        'general_msg': 'wrong_data_types',
                        'errors': {'required_types': {k: str(v) for k, v in
required.items() if k in wrong_types}}
                    }
                    return custom_response(False, response, 400)
                return fn(*args, **kwargs)


            return wrapper


        return decorator



    def custom_response(status, data, status_code):
        response = make_response(jsonify({"status": status, "result": data}),
status_code)
        response.headers["Content-Type"] = "application/json"
        return response



    def validations_register(email: str, password: str, repeat_password: str):
        regex = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b'
        errors = []
        if not re.fullmatch(regex, email):
            errors.append('email_validation_failed')
        if password != repeat_password:
            errors.append('passwords_not_match')
```

```
            if len(password) < 8 or re.search('[0-9]', password) is None or
re.search('[A-Z]', password) is None:

                errors.append('password_validation_failed')

            return errors



        @app.post('/login')

        def login():

            id_token = request.headers.get('authorization')

            try:

                user = auth.verify_id_token(id_token)

                db = firestore.client()

                user_record =
db.collection(u'users').document(f'{user["uid"]}').get()

                return custom_response(False, user_record.to_dict(), 200)

            except Exception as e:

                print(e)

                return custom_response(False, {'errors':
['failed_to_verify_token']}, 401)



        @app.post('/register')

        @required_params({

            'email': str,

            'first_name': str,

            'last_name': str,

            'password': str,

            'repeat_password': str,

            'is_author': bool

        })

        def register():

            req = request.get_json()

            email = req['email']
```

```
            password = req['password']
            repeat_password = req['repeat_password']
            first_name = req['first_name']
            last_name = req['last_name']
            is_author = req['is_author']
            avatar = None
            try:
                if req['avatar'] is not None or req['avatar'] != '':
                    avatar = req['avatar']
            except KeyError:
                avatar = None

            errors = validations_register(email, password, repeat_password)
            if len(errors) > 0:
                return custom_response(False, {'errors': errors}, 400)
            try:
                user = auth.create_user(
                    email=email,
                    email_verified=False,
                    password=req['password'],
                    display_name=f'{first_name} {last_name}',
                    disabled=False,
                )
            except EmailAlreadyExistsError:
                return custom_response(False, {'errors':
['email_already_exists']}, 409)
            except:
                return custom_response(False, {'errors': ['unknown_error']}, 500)

            db = firestore.client()
            db.collection(u'users').document(f'{user.uid}').set({
                'email': email,
```

```
            'first_name': first_name,
            'last_name': last_name,
            'is_author': is_author,
            'avatar': avatar
        })
        return custom_response(True, None, 200)

@https_fn.on_request(region='europe-west1')
def authentication(req: https_fn.Request) -> https_fn.Response:
    with app.request_context(req.environ):
        return app.full_dispatch_request()
```

ДОДАТОК Б

ЛІСТИНГ ПРОГРАМИ (МІКРОСЕРВІС PROFILE)

./functions/src/index.ts:

```
import express from 'express';
import config from './config/config';
import logging from './config/logging';
import * as functions from 'firebase-functions';

// Routes
import health from './routes/health';
import profile from './routes/profile';

const cors = require('cors');
const cookieParser = require('cookie-parser');
const admin = require('firebase-admin');

const NAMESPACE = 'Index';
```

```
admin.initializeApp({
    credential: admin.credential.cert(config.firebaseAdminConfig)
});

// express app
const router = express();

router.use(
    cors({
        origin: config.allowedOrigins,
        methods: config.allowedMethods,
        allowedHeaders: config.allowedHeaders,
        maxAge: config.maxAge,
        optionsSuccessStatus: 200
    })
);

// Logging the request
router.use((req, res, next) => {
    logging.info(NAMESPACE, `METHOD - [${req.method}, URL - [${req.url}], IP - [${req.socket.remoteAddress}]]`);

    res.on('finish', () => {
        logging.info(
            NAMESPACE,
            `METHOD - [${req.method}, URL - [${req.url}], IP - [${req.socket.remoteAddress}]
        STATUS - [${res.statusCode}]]`
        );
    });

    next();
});
```

```
// Parse the request
router.use(cookieParser());
router.use(express.json());
router.use(express.urlencoded({ extended: true }));
router.use('/health', health);
router.use('', profile);
exports.profile = functions.region('europe-west1').https.onRequest(router);

router.use((req, res) => {
    const error = new Error('Route was not found on server');
    return res.status(404).json({
        message: error.message
    });
});
```

./functions/routes/profile.ts:

```
import express from 'express';
import controller from '../controllers/profile';
import { verifySession } from '../middleware/verifySession';

const router = express.Router();

router.get('', [verifySession], controller.getProfile);
router.patch('', [verifySession], controller.updateProfile);
router.patch('/change-password', [verifySession], controller.changePassword);
router.delete('', [verifySession], controller.deleteProfile);

export default router;
```

./functions/routes/health.ts:

```
import express from 'express';
```

```
import controller from '../controllers/health';

const router = express.Router();


router.get('/ping', controller.healthCheck);


export = router;
```

./functions/controllers/health.ts:

```
import { Request, Response } from 'express';

import logging from '../config/logging';

import { successResponse } from '../utils/customResponses';


const NAMESPACE = 'Health check Controller';


const healthCheck = (req: Request, res: Response) => {

    logging.info(NAMESPACE, `Health check route called.`);

    return successResponse(res, { ping: 'pong' }, 200);

};


export default { healthCheck };
```

./functions/controllers/profile.ts:

```
import { Request, Response } from 'express';

import { deleteDoc, getDocById, updateDoc } from '../utils/firestoreOperations';

import { failResponse, successResponse } from '../utils/customResponses';

import { ChangePasswordRequestBody, UpdateUserProfileRequestBody } from
'../types/profile';

import { profileUpdateRecord } from '../utils/createRecord';

import logging from '../config/logging';

import admin, { FirebaseError } from 'firebase-admin';


const NAMESPACE = 'Profile controller';
```

```
const getProfile = async (req: Request<any>, res: Response) => {

    await getDocById('users', res.locals.decodedClaims.uid)

        .then((profile) => {

            if (!profile) return failResponse(res, NAMESPACE, 'Profile not
found', ['not_found'], 404);

            return successResponse(res, profile, 200);

        })

        .catch((err) => {

            logging.error(NAMESPACE, 'Unknown error', err);

            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);

        });

};


const updateProfile = async (req: Request<any, unknown,
UpdateUserProfileRequestBody>, res: Response) => {

    const newProfileData = profileUpdateRecord(req.body);

    await admin.auth().updateUser(res.locals.decodedClaims.uid, {

        displayName: `${req.body.first_name} ${req.body.last_name}`

    });

    await updateDoc('users', res.locals.decodedClaims.uid, newProfileData)

        .then((updateResult) => {

            if (!updateResult) return failResponse(res, NAMESPACE, 'User not
found', ['not_found'], 404);

            return successResponse(res, newProfileData, 200);

        })

        .catch((err) => {

            logging.error(NAMESPACE, 'Unknown error', err);

            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);

        });

};
```

```
const changePassword = async (req: Request<any, unknown,
ChangePasswordRequestBody>, res: Response) => {

    if (req.body.newPassword !== req.body.repeatNewPassword)

        return failResponse(res, NAMESPACE, 'Passwords mismatch',
['update_failed'], 400);

    await admin

        .auth()

        .updateUser(res.locals.decodedClaims.uid, {

            password: req.body.newPassword

        })

        .then((result) => {

            return successResponse(res);

        })

        .catch((err: FirebaseError) => {

            if (err.code === 'auth/invalid-password') {

                logging.error(NAMESPACE, 'Failed to update user password', err);

                return failResponse(res, NAMESPACE, 'Invalid new password',
['invalid_password'], 400);

            }


            if (err.code === 'auth/user-not-found') {

                logging.error(NAMESPACE, `User ${res.locals.decodedClaims.uid}
was not found`, err);

                return failResponse(res, NAMESPACE, 'User not found',
['not_found'], 404);

            }


            logging.error(NAMESPACE, 'Unknown error', err);

            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);

        });

};


const deleteProfile = async (req: Request<any>, res: Response) => {

    return await admin
```

```
        .auth()
        .deleteUser(res.locals.decodedClaims.uid)
        .then(() => {
            deleteDoc('users', res.locals.decodedClaims.uid).catch((err) => {
                logging.error(NAMESPACE, 'Unknown error', err);
                return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);
            });
            return successResponse(res);
        })
        .catch((err: FirebaseError) => {
            if (err.code === 'auth/user-not-found') {
                logging.error(NAMESPACE, `User ${res.locals.decodedClaims.uid}
was not found`, err);
                return failResponse(res, NAMESPACE, 'User not found',
['not_found'], 404);
            }
            logging.error(NAMESPACE, 'Unknown error', err);
            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);
        });
};

export default { getProfile, updateProfile, changePassword, deleteProfile };
```

./functions/verifySession.ts:

```
import { NextFunction, Request, Response } from 'express';
import { failResponse } from '../utils/customResponses';
import admin from 'firebase-admin';

const NAMESPACE = 'Verify Session';
```

```
export const verifySession = (req: Request, res: Response, next: NextFunction) =>
{
    const token = req.headers.authorization || '';
    admin
        .auth()
        .verifyIdToken(token, true)
        .then((decodedClaims) => {
            res.locals.decodedClaims = decodedClaims;
            next();
        })
        .catch((error) => {
            return failResponse(res, NAMESPACE, 'Unauthorized', ['unauthorized'],
401);
        });
};
```

./fuctions/types/profile.ts:

```
export interface UpdateUserProfileRequestBody {
    first_name: string;
    last_name: string;
    avatar: string;
    is_author: boolean;
}

export interface ChangePasswordRequestBody {
    newPassword: string;
    repeatNewPassword: string;
}
```

./functions/utils/clearUndefinedValuesInObject.ts:

```
export const clearUndefinedValuesInObject = (obj: any) => {
    Object.keys(obj).forEach((key) => {
        if (obj[key] === undefined) {
```

```
            delete obj[key];
        }
    });
    return obj;
};
```

./functions/utils/createRecord.ts:

```
export const checkUndef = (value: any, def: any) => {
    return value === undefined ? def : value;
};

export const deleteUndef = (obj: any) => {
    Object.keys(obj).map((key) => {
        if (obj[key] === undefined) delete obj[key];
    });
    return obj;
};

export const profileUpdateRecord = (data: any) => {
    const result = {
        first_name: checkUndef(data.first_name, undefined),
        last_name: checkUndef(data.last_name, undefined),
        avatar: checkUndef(data.avatar, undefined),
        is_author: checkUndef(data.is_author, undefined)
    };
    return deleteUndef(result);
};
```

./functions/utils/customResponses.ts:

```
import { Response } from 'express';
import logging from '../config/logging';
```

```
interface successResponse {
    result: any;
    success: boolean;
}

export const successResponse = (res: Response, data?: any, status: number
= 200): Response => {
    const body: successResponse = {
        result: data,
        success: true
    };
    return res.status(status).json(body);
};

interface failedResponse {
    errors: any;
    message: string;
    success: boolean;
}

export const failResponse = (
    res: Response,
    namespace: string,
    message: string = 'Server failure',
    errors: string[] = [],
    status: number = 400
): Response => {
    logging.error(namespace, message);
    const body: failedResponse = {
        errors: errors,
        message: message,
```

```
            success: false
        };
        return res.status(status).json(body);
    };

    export const permissionsForbiddenResponse = (res: Response, namespace:
    string) => {
        return failResponse(res, namespace, 'Not enough permissions to
    perform', ['forbidden'], 403);
    };
```

./functions/utils/firestoreOperations.ts:

```
    import { Response } from 'express';
    import logging from '../config/logging';

    interface successResponse {
        result: any;
        success: boolean;
    }

    export const successResponse = (res: Response, data?: any, status: number
    = 200): Response => {
        const body: successResponse = {
            result: data,
            success: true
        };
        return res.status(status).json(body);
    };

    interface failedResponse {
        errors: any;
        message: string;
```

```
    success: boolean;
}


export const failResponse = (
    res: Response,
    namespace: string,
    message: string = 'Server failure',
    errors: string[] = [],
    status: number = 400
): Response => {
    logging.error(namespace, message);
    const body: failedResponse = {
        errors: errors,
        message: message,
        success: false
    };
    return res.status(status).json(body);
};


export const permissionsForbiddenResponse = (res: Response, namespace:
string) => {
    return failResponse(res, namespace, 'Not enough permissions to
perform', ['forbidden'], 403);
};
```

ДОДАТОК В
ЛІСТИНГ ПРОГРАМИ (МІКРОСЕРВІС MEDIA)

./functions/src/index.ts:

```
import express from 'express';
import config from './config/config';
import logging from './config/logging';
import * as functions from 'firebase-functions';

// Routes
import health from './routes/health';
import media from './routes/media';

const cors = require('cors');
const cookieParser = require('cookie-parser');
const admin = require('firebase-admin');

const NAMESPACE = 'Index';

admin.initializeApp({
    credential: admin.credential.cert(config.firebaseAdminConfig),
    databaseURL: 'https://kpiedu-1fb35.europe-west1.firebasedatabase.app/'
});

// express app
const router = express();

router.use(
    cors({
        origin: config.allowedOrigins,
        methods: config.allowedMethods,
```

```
            allowedHeaders: config.allowedHeaders,
            maxAge: config.maxAge,
            optionsSuccessStatus: 200
        })
    );

    // Logging the request
    router.use((req, res, next) => {
        logging.info(NAMESPACE, `METHOD - [${req.method}, URL - [${req.url}], IP - [${req.socket.remoteAddress}]]`);

        res.on('finish', () => {
            logging.info(
                NAMESPACE,
                `METHOD - [${req.method}, URL - [${req.url}], IP - [${req.socket.remoteAddress}]
            STATUS - [${res.statusCode}]]`
            );
        });

        next();
    });

    // Parse the request
    router.use(cookieParser());
    router.use(express.json());
    router.use(express.urlencoded({ extended: true }));

    router.use('/health', health);
    router.use('', media);
    exports.media = functions.region('europe-west1').https.onRequest(router);

    router.use((req, res) => {
```

```
        const error = new Error('Route was not found on server');
        return res.status(404).json({
            message: error.message
        });
    });
```

./functions/config/config.ts:

```
    import firebaseAdminConfig from '../../configs/firebaseConfig.json';
    import generalConfig from '../../configs/generalConfig.json';

    const SERVER_HOSTNAME = process.env.SERVER_HOSTNAME || 'localhost';
    const SERVER_PORT = process.env.SERVER_PORT || 8002;

    const SERVER = {
        hostname: SERVER_HOSTNAME,
        port: SERVER_PORT
    };
    const config = {
        server: SERVER,
        firebaseAdminConfig: firebaseAdminConfig.adminConfig,
        allowedOrigins: generalConfig.allowedOrigins,
        allowedMethods: generalConfig.allowedOptions,
        allowedHeaders: generalConfig.allowedHeaders,
        maxAge: generalConfig.maxAge,
        allowCreds: generalConfig.allowCreds,
        pinEncryptionKey: generalConfig.pinEncryptionKey,
        iv: generalConfig.iv,
        mailjet: generalConfig.mailjet
    };

    export default config;
```

./functions/config/logging.ts:

```
	const getTimeStamp = (): string => {
	    return new Date().toISOString();
	};

	const info = (namespace: string, message: string, object?: any): void => {
	    if (object) {
	        console.info(`[${getTimeStamp()}] [INFO] [${namespace}]
	${message},`, object);
	    } else {
	        console.info(`[${getTimeStamp()}] [INFO] [${namespace}]
	${message},`);
	    }
	};

	const warn = (namespace: string, message: string, object?: any): void => {
	    if (object) {
	        console.warn(`[${getTimeStamp()}] [WARN] [${namespace}]
	${message},`, object);
	    } else {
	        console.warn(`[${getTimeStamp()}] [WARN] [${namespace}]
	${message},`);
	    }
	};

	const error = (namespace: string, message: string, object?: any): void =>
	{
	    if (object) {
	        console.error(`[${getTimeStamp()}] [ERROR] [${namespace}]
	${message},`, object);
	    } else {
	        console.error(`[${getTimeStamp()}] [ERROR] [${namespace}]
	${message},`);
	    }
	};
```

```
    const debug = (namespace: string, message: string, object?: any): void =>
    {
        if (object) {
            console.debug(`[${getTimeStamp()}] [DEBUG] [${namespace}]
    ${message},`, object);
        } else {
            console.debug(`[${getTimeStamp()}] [DEBUG] [${namespace}]
    ${message},`);
        }
    };


    export default {
        info,
        warn,
        error,
        debug
    };
```

./functions/controllers/media.ts:

```
    import { Request, Response } from 'express';
    import { CreateNewMediaRequestBody, GetAllMediaRequestQuery } from
    '../types/media';
    import {
        createDocWithoutDocId,
        deleteDoc,
        getCollectionWithPagination,
        getDocById,
        updateDoc,
        WhereQuery,
        WhereSearchQuery,
        whereSearchQueryConstructor
    } from '../utils/firestoreOperations';
    import { failResponse, successResponse } from '../utils/customResponses';
    import logging from '../config/logging';
```

```
import { validateMediaCreation } from '../utils/validations';

import { createMediaRecord } from '../utils/createRecord';

import { getRecord, setRecord } from '../utils/realtimeDBOperations';

import { emailSend } from '../utils/emailSender';

const utf8 = require('utf8');

const NAMESPACE = 'media controller';

const getAllMedia = async (req: Request<any, unknown, unknown,
GetAllMediaRequestQuery>, res: Response) => {

    const whereQueries: WhereQuery[] = [];

    if (req.query.region) {

        whereQueries.push({

            field: 'region',

            operator: '==',

            value: req.query.region

        });

    }

    if (req.query.type) {

        whereQueries.push({

            field: 'type',

            operator: '==',

            value: req.query.type

        });

    }

    if (req.query.specialization) {

        whereQueries.push({

            field: 'specialization',

            operator: '==',

            value: req.query.specialization

        });

    }

    if (req.query.rate) {

        whereQueries.push({
```

```
            field: 'rate',
            operator: '==',
            value: parseInt(req.query.rate)
        });
    }
    let whereSearch: undefined | WhereSearchQuery = undefined;
    if (req.query.title) whereSearch =
whereSearchQueryConstructor('title', decodeURI(req.query.title));
    return await getCollectionWithPagination('media', undefined,
undefined, whereQueries, whereSearch)
        .then((media) => {
            return successResponse(res, media, 200);
        })
        .catch((err) => {
            logging.error(NAMESPACE, 'Error sending media', err);
            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);
        });
};
const getMediaById = async (req: Request<any>, res: Response) => {
    return await getDocById('media', req.params.mediaId)
        .then((media: any) => {
            if (!media) return failResponse(res, NAMESPACE, 'Media not
found', ['not_found'], 404);
            successResponse(res, media, 200);
        })
        .catch((err) => {
            logging.error(NAMESPACE, 'Error sending media', err);
            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);
        });
};
const createMedia = async (req: Request<any, unknown,
CreateNewMediaRequestBody>, res: Response) => {
    const validation = await validateMediaCreation(req.body);
```

```
    if (validation.length > 0) return failResponse(res, NAMESPACE, 'Failed
to create a media', validation, 400);

    const newMedia = createMediaRecord(req.body);

    await createDocWithoutDocId('media', newMedia, true)

        .then((result) => {

            return successResponse(res, result, 200);

        })

        .catch((err) => {

            logging.error(NAMESPACE, 'Error creating media', err);

            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);

        });

};

const deleteMedia = async (req: Request<any>, res: Response) => {

    await deleteDoc('media', req.params.mediaId)

        .then((result) => {

            if (!result) return failResponse(res, NAMESPACE, 'Media not
found', ['not_found'], 404);

        })

        .catch((err) => {

            logging.error(NAMESPACE, 'Error deleting media', err);

            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);

        });

};

const changeMediaSubscriptionStatus = async (req: Request<any>, res:
Response) => {

    const media = await getDocById('media', req.params.mediaId);

    if (!media) return failResponse(res, NAMESPACE, 'Media not found',
['not_found'], 404);



    let subStatus = true;

    if (req.route.path.startsWith('/un')) {

        subStatus = false;
```

```
        const subscribed = await
getRecord(`mediasSubscriptions/${res.locals.decodedClaims.uid}/${media.id}
`);

        try {

            if (!subscribed.val().subscribed) {

                return failResponse(res, NAMESPACE, 'Already
unsubscribed', ['already_unsubscribed'], 409);

            }

        } catch (err) {

            logging.error(NAMESPACE, '', err);

        }

    } else {

        const subscribed = await
getRecord(`mediasSubscriptions/${res.locals.decodedClaims.uid}/${media.id}
`);

        try {

            if (subscribed.val().subscribed) {

                return failResponse(res, NAMESPACE, 'Already subscribed',
['already_subscribed'], 409);

            }

        } catch (err) {

            logging.error(NAMESPACE, '', err);

        }

    }



    return await
setRecord(`mediasSubscriptions/${res.locals.decodedClaims.uid}/${media.id}
`, {

        subscribed: subStatus

    })

        .then(() => {

            if (subStatus) {

                emailSend(

                    res.locals.decodedClaims.uid,

                    res.locals.decodedClaims.name,

                    'Media Subscription',
```

```
                    `${res.locals.decodedClaims.name}, thank you for
subscribing to ${media.title}`

                );

                updateDoc('media', media.id, { subscriptions:
media.subscriptions + 1 }).catch((err) => {

                    logging.error(NAMESPACE, 'Update media failed', err);

                });

            } else {

                emailSend(

                    res.locals.decodedClaims.uid,

                    res.locals.decodedClaims.name,

                    'Media Subscription',

                    `${res.locals.decodedClaims.name}, we sorry to hear
about yours unsubscribing from ${media.title}`

                );

                updateDoc('media', media.id, { subscriptions:
media.subscriptions - 1 }).catch((err) => {

                    logging.error(NAMESPACE, 'Update media failed', err);

                });

            }

            return successResponse(res, { subscribed: subStatus }, 200);

        })

        .catch((err) => {

            logging.error(NAMESPACE, 'Error subscribing to media', err);

            return failResponse(res, NAMESPACE, 'Unknown error',
['unknown_error'], 500);

        });

};


const showSubscriptions = async (req: Request<any>, res: Response) => {

    const subscriptions = await
getRecord(`mediasSubscriptions/${res.locals.decodedClaims.uid}/`).then((sn
ap) =>

        snap.val()

    );
```

```
            let result = subscriptions
                ? await Promise.all(
                      Object.keys(subscriptions).map(async (mediaId) => {
                          if (subscriptions[mediaId].subscribed) {
                              const media = await getDocById('media', mediaId);
                              if (media) return media;
                          }
                          return;
                      })
                  )
                : [];
            result = result.filter((value: any) => value);
            return successResponse(res, result, 200);
        };
        export default {
            getAllMedia,
            getMediaById,
            createMedia,
            deleteMedia,
            changeMediaSubscriptionStatus,
            showSubscriptions
        };
```

./functions/routes/media.ts:

```
        export type MediaSpecialization = 'politics' | 'culture' | 'general';
        export type MediaType = 'agency' | 'newspaper' | 'online';
        export type MediaRegion = 'kyiv' | 'lviv' | 'cherkasy';
        export interface GetAllMediaRequestQuery {
            specialization: MediaSpecialization;
            type: MediaType;
            region: MediaRegion;
            rate: string;
            title: string;
```

```
    }
    export interface CreateNewMediaRequestBody {
        description: string;
        url: string;
        title: string;
        specialization: MediaSpecialization;
        type: MediaType;
        region: MediaRegion;
    }
```

./functions/types/media.ts:

```
    export type MediaSpecialization = 'politics' | 'culture' | 'general';
    export type MediaType = 'agency' | 'newspaper' | 'online';
    export type MediaRegion = 'kyiv' | 'lviv' | 'cherkasy';
    export interface GetAllMediaRequestQuery {
        specialization: MediaSpecialization;
        type: MediaType;
        region: MediaRegion;
        rate: string;
        title: string;
    }
    export interface CreateNewMediaRequestBody {
        description: string;
        url: string;
        title: string;
        specialization: MediaSpecialization;
        type: MediaType;
        region: MediaRegion;
    }
```

./functions/utils/realtimeDBOperations.ts:

```
    import admin, { database } from 'firebase-admin';
    import logging from '../config/logging';
```

```
import Query = database.Query;
const NAMESPACE = 'ReltimeDBoperations';
export const getRecord = async (
    path: string,
    limitNumber?: number | undefined,
    orderByChildPath?: string,
    childRangeQuery?: any,
    equalTo?: any
) => {
    let dbRef: Query = admin.database().ref(path);

    if (orderByChildPath) {
        dbRef = dbRef.orderByChild(orderByChildPath);
    }

    if (childRangeQuery) {
        if (childRangeQuery.start) {
            dbRef = dbRef.startAfter(childRangeQuery.start);
        }
        if (childRangeQuery.end) {
            dbRef = dbRef.endBefore(childRangeQuery.end);
        }
    }

    if (limitNumber) {
        dbRef = dbRef.limitToLast(limitNumber);
    }

    if (equalTo) {
        dbRef = dbRef.equalTo(equalTo);
    }
```

```
    return await dbRef.once(
        'value',
        async (data) => {
            return data;
        },
        (err: Error) => {
            logging.error(NAMESPACE, err.name, err);
            throw err;
        }
    );
};

export const pushRecord = async (path: string, data: any) => {
    let dbRef = admin.database().ref(path);
    return await dbRef
        .push(data)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            logging.error(NAMESPACE, err.name, err);
            throw err;
        });
};

export const setRecord = async (path: string, data: any) => {
    let dbRef = admin.database().ref(path);
    return await dbRef
        .set(data)
        .then(() => {
            return true;
        })
```

```
        .catch((err) => {
            logging.error(NAMESPACE, err.name, err);
            throw err;
        });
};

export const updateRecord = async (path: string, data: any) => {
    let dbRef = admin.database().ref(path);
    return await dbRef
        .update(data)
        .then(() => {
            return true;
        })
        .catch((err) => {
            logging.error(NAMESPACE, err.name, err);
            throw err;
        });
};

export const removeRecord = async (path: string) => {
    let dbRef = admin.database().ref(path);
    return await dbRef
        .remove()
        .then(() => {
            return true;
        })
        .catch((err) => {
            logging.error(NAMESPACE, err.name, err);
            throw err;
        });
};
```