ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «МОДУЛЬ МАСШТАБОВАНИХ ВЕБ-ЗАСТОСУВАНЬ НА ХМАРНІЙ ІНФРАСТРУКТУРІ GOOGLE CLOUD»

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерної інженерії
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні системи та мережі
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ *Богдан СТЕЛЬМАХ*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач(ка) вищої освіти гр. КСДМ-61
Богдан СТЕЛЬМАХ
Ім'я, ПРІЗВИЩЕ
Керівник: Анастасія ВЄЧЕРКОВСЬКА
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ
Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2025
ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Магістр

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Комп'ютерної інженерії

_____ Наталія ЛАЩЕВСЬКА

«_____» _____ 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Стельмаху Богдану Олександровичу

1. Тема кваліфікаційної роботи: « Модуль масштабованих веб-застосувань на хмарній інфраструктурі Google Cloud »
керівник кваліфікаційної роботи Анастасія ВЄЧЕРКОВСЬКА, *к.т.н.,*доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «20» грудня 2024 р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, характеристики хмарних сервісів, модулі створення веб архітектури, масштабованість і надійність системи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Теоретичні основи побудови масштабованих веб-застосувань на хмарній інфраструктурі Google Cloud.
2. Розробка модуля масштабованих веб-застосувань.
3. Розгортання та експлуатація модуля на хмарній інфраструктурі Google Cloud.

5. Перелік ілюстративного матеріалу: *презентація,додаток Б*
    1. Огляд платформи GCP.
    2. Вибір основних сервісів Googlce Cloud для модуля.
    3. Огляд основних методів і характеристик модуля.
    4. Загальний огляд модуля.
    5. Розробка архітектури модуля.
    6. Реалізація модуля.
    7. Створення безпечної мережі.
    8. Розгортання застосунку в GKE.
    9. Тестування застосунку.
    10. Моніторинг і логування модуля.
6. Дата видачі завдання «16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	02.09-10.09.2024	
2	Огляд платформи Google Cloud		
3	Аналіз методів та засобів розгортання веб-застосунків		
4	Розробка архітектури модуля		
5	Розгортання і тестування модуля		
7	Оформлення роботи: вступ, висновки, реферат		
8	Розробка демонстраційних матеріалів		
9	Попередній захист роботи		

Здобувач вищої освіти _____ Богдан СТЕЛЬМАХ

(підпис)

Керівник

кваліфікаційної роботи _____ Анастасія ВЄЧЕРКОВСЬКА

(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 79 стор.,6 табл., 33 рис., 15 джерел.

Мета роботи – розробка модуля масштабованих веб-застосунків на хмарній інфраструктурі Google Cloud та дослідження його ефективності.

Об'єкт дослідження – процес розробки та розгортання масштабованих веб-застосунків.

Предмет дослідження – модуль масштабованих веб-застосунків на хмарній інфраструктурі Google Cloud, його архітектура, функціональні можливості та ефективність.

У роботі використано різноманітні методи, такі як:

1. Аналіз літератури.
2. Проектування архітектури.
3. Експериментальне розгортання.

Проведено аналіз сучасних методів розгортання веб-застосунків на Google Cloud Platform.

Розроблено та оптимізовано модуоб масштабованих веб-застосувань на Google Cloud Platform

Проведено експерименти для розгортання модуля і тестування навантаження на застосунок.

КЛЮЧОВІ СЛОВА: GOOCLE CLOUD PLATFORM, АХІТЕКТУРА ВЕБ-ЗАСТОСУНКІВ, GOOGLE KUBERNETES ENGINE, НАДІЙНІСТЬ СИСТЕМИ, ТЕСТУВАННЯ НАВАНТАЖЕННЯ

ABSTRACT

Text part of the master's qualification work: 79 pages, 33 pictures, 6 table, 15 sources.

The purpose of the work is a development of a scalable web application module on the Google Cloud cloud infrastructure and research into its effectiveness.

Object of research – the process of developing and deploying scalable web applications.

Subject of research – Google Cloud scalable web application module, its architecture, functionality and performance.

Summary of the work:

1. Analysis of the literature.
2. Architecture design.
3. Experimental deployment.

An analysis of modern methods of deploying web applications on Google Cloud Platform was carried out.

Developed and optimized the mod about scalable web applications on Google Cloud Platform

Experiments were conducted to deploy the module and test the load on the application.

KEYWORDS: GOOCLE CLOUD PLATFORM, WEB APPLICATION ARCHITECTURE, GOOGLE KUBERNETES ENGINE, SYSTEM RELIABILITY, LOAD TESTING

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

ПОДАННЯ
ГОЛОВІ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня магістра

Направляється здобувач(ка) СТЕЛЬМАХА Б.О. до захисту кваліфікаційної роботи за спеціальністю 123 Комп'ютерна інженерія
освітньо-професійної програми «Комп'ютерні системи та мережі»

на тему: «Модуль масштабованих веб-застосувань на хмарній інфраструктурі Google Cloud ».

Кваліфікаційна робота і рецензія додаються.

Директор ННІІТ _____ Андрій БОНДАРЧУК
(підпис)

Висновок керівника кваліфікаційної роботи

Кваліфікаційна робота розглянута. Здобувач(ка) СТЕЛЬМАХА Б.О. допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедри КІ _____ Наталія ЛАЩЕВСЬКА
(підпис)

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну магістерську роботу

здобувача(ки) вищої освіти Стельмаха Богдана Стельмаха

на тему « Модуль масштабованих веб-застосувань на хмарній інфраструктурі Google Cloud »

Актуальність.

…

Позитивні сторони.

1.

2.

3.

Недоліки.

1.

2.

Відзначені зауваження не впливають на загальну позитивну оцінку кваліфікаційної магістерської роботи.

Висновок: *кваліфікаційна робота на здобуття ступеня магістра заслуговує оцінку "_____", а здобувач(ка)* ***Стельмах Богдан Стельмах*** *заслуговує присвоєння кваліфікації магістр з комп'ютерної інженерії.*

Рецензент: _____ _____

науковий ступінь, вчене звання *підпис* *Ім'я, ПРІЗВИЩЕ*

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 12
ВСТУП 13
1 ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ МАСШТАБОВАНИХ ВЕБ-ЗАСТОСУНКІВ НА ХМАРНІЙ ІНФРАСТРУКТУРІ GOOGLE CLOUD 15
1.1 Хмарні технології та їх переваги для веб-застосунків 15
1.2 Архітектура мережевих застосувань 23
1.2.1 Мікросервісна архітектура 23
1.2.2 Безсерверна архітектура 24
1.2.3 Розподілені системи та їх особливості 25
1.3 Огляд платформи Google Cloud Platform (GCP) та її сервісів 26
1.3.1 Google Compute Engine 27
1.3.2 Google App Engine 28
1.3.3 Google Kubernetes Engine 29
1.3.4 Google Cloud Storage 30
1.3.5 Google Cloud SQL 31
1.3.6 BigQuery 32
1.3.7 Інші сервіси GCP 32
1.4 Архітектурні патерни масштабованих веб-застосувань 34
1.4.1 Шарувата архітектура (Layered Architecture) 34
1.4.2 Архітектура, керована подіями (Event-Driven Architecture) 35
1.4.3 Патерн "Команда-запит поділу відповідальності" (CQRS) 36
1.5 Методи забезпечення масштабованості веб-застосувань 36
1.5.1 Горизонтальне та вертикальне масштабування 37
1.5.2 Балансування навантаження 39
1.5.3 Кешування 40
1.5.4 Оптимізація бази даних 40
РОЗДІЛ 2. РОЗРОБКА МОДУЛЯ МАСШТАБОВАНИХ ВЕБ-ЗАСТОСУВАНЬ 43
2.1 Аналіз вимог до модуля 43
2.2 Вибір сервісів GCP 45
2.3 Архітектура модуля 49
2.3.1 Методологія дванадцяти факторів 52
2.3.2 Методологія Continuous Integration в Google Cloud 54

2.4 Вибір технологій та інструментів розробки 56
2.4.1 Визначення SLI та SLO 57
2.4.2 Створення мікросервіса для програми 60
2.4.3 Проектування REST API 61
2.4.4 Вибір сервісів Google Cloud для зберігання даних 62
2.4.5 Проектування мережевої інфраструктури та балансування навантаження 63
2.4.6 Забезпечення надійності, масштабованості та безпеки 64
РОЗДІЛ 3. РОЗГОРТАННЯ ТА ЕКСПЛУАТАЦІЯ МОДУЛЯ НА ХМАРНІЙ ІНФРАСТРУКТУРІ GOOGLE CLOUD 69
3.1 Розгортання застосунку 69
3.1.1 Створення безпечної мережі 77
3.1.2 Створення кластера Kubernetes в GKE 80
3.2 Демонстрація масштабування застосунку в дії 86
ВИСНОВКИ 91
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 93
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 95
ДОДАТОК Б. БЕКЕНД ЗАСТОСУНКУ 102

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

GCP – Google Cloud Platform

GKE - Google Kubernetes Engine

SQL - Structured query language

HPA - Horizontal Pod autoscaling

VPA - Vertical Pod autoscaling

REST - Representational State Transfer

API - Application programming interface

CQRS - Command-query responsibility segregation

CPU - Central processing unit

ВСТУП

Актуальність теми. У сучасному світі, де веб-застосунки відіграють ключову роль у бізнесі, комунікації та розвагах, забезпечення їх масштабованості та надійності є критично важливим. Зростання обсягів даних, кількості користувачів та вимог до продуктивності вимагає нових підходів до розробки та розгортання веб-застосунків. Хмарні технології, зокрема Google Cloud Platform, надають потужні інструменти для вирішення цих задач, дозволяючи створювати гнучкі, масштабовані та економічно ефективні рішення.

Мета роботи. Метою магістерської роботи є розробка модуля масштабованих веб-застосунків на хмарній інфраструктурі Google Cloud та дослідження його ефективності.

Об'єкт дослідження. Об'єктом дослідження є процес розробки та розгортання масштабованих веб-застосунків.

Предмет дослідження. Предметом дослідження є модуль масштабованих веб-застосунків на хмарній інфраструктурі Google Cloud, його архітектура, функціональні можливості та ефективність.

Методи дослідження. Для досягнення мети дослідження будуть використані наступні методи:

- Аналіз літератури та документації Google Cloud з питань хмарних технологій, масштабування та розробки веб-застосунків.
- Проектування архітектури модуля з використанням сучасних підходів та best practices.
- Експериментальне розгортання та тестування модуля на Google Cloud Platform.
- Аналіз результатів тестування та оцінка ефективності модуля за визначеними критеріями.

Наукова новизна одержаних результатів. Наукова новизна роботи полягає у:

- Розробці комплексного модуля масштабованих веб-застосунків, що враховує специфіку Google Cloud Platform.
- Дослідженні та порівнянні різних методів забезпечення масштабованості та відмовостійкості.
- Експериментальному підтвердженні ефективності запропонованих рішень.

Практична значущість одержаних результатів. Практична значущість роботи полягає у можливості використання розробленого модуля для створення масштабованих та надійних веб-застосунків на Google Cloud Platform. Результати дослідження можуть бути використані розробниками та архітекторами для проектування та розгортання хмарних рішень.

1 ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ МАСШТАБОВАНИХ ВЕБ-ЗАСТОСУНКІВ НА ХМАРНІЙ ІНФРАСТРУКТУРІ GOOGLE CLOUD

1.1 Хмарні технології та їх переваги для веб-застосунків

У сучасному світі, де обсяги даних та кількість користувачів веб-застосунків постійно зростають, традиційні підходи до розробки та розгортання програмного забезпечення часто виявляються неефективними. Виникає потреба в нових рішеннях, які дозволять створювати гнучкі, масштабовані та економічно вигідні системи. Саме такими рішеннями є хмарні технології. Хмарні обчислення – це модель надання обчислювальних ресурсів, таких як сервери, сховища даних, мережі, програмне забезпечення, через Інтернет за запитом. Користувачі отримують доступ до цих ресурсів віддалено, не потребуючи власної інфраструктури.

Хмарні обчислення - це спосіб організації комп'ютерних мереж, де величезна кількість розподілених комп'ютерів об'єднані через Інтернет та доступні користувачам через мережеві протоколи, такі як IP. По суті, це надання послуг через Інтернет: обчислювальні потужності, розподілені системи та обробка даних за запитом, з можливістю масштабування, швидкого розширення ресурсів та відновлення після збоїв.[]

Публічна хмара - це вид хмарних обчислень, де інфраструктура та послуги належать і управляються стороннім провайдером та доступні широкому загалу через Інтернет. Користувачі можуть отримати доступ до спільних ресурсів, таких як сервери, сховища даних та програми, сплачуючи лише за те, що вони використовують. Прикладами провайдерів публічних хмар є Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform.

До переваг публічної хмари можна віднести:

Економія коштів, бо Ви платите лише за використані ресурси, що робить публічну хмару більш економічно вигідною, ніж утримання власних фізичних серверів та інфраструктури.

Автоматичне оновлення програмного забезпечення,бо Вам не потрібно турбуватися про оновлення програмного забезпечення, оскільки воно відбувається автоматично.

Доступність, бо публічні хмари дозволяють користувачам отримувати доступ до своїх ресурсів та програм з будь-якої точки світу, де є підключення до Інтернету.

Звичайно,що є й недоліки публічної хмари:

Безпека та конфіденційність, бо публічні хмари можуть бути вразливими до витоків даних, кібератак та інших ризиків безпеки. Оскільки дані зберігаються на серверах, що належать сторонньому провайдеру, завжди існує ризик витоку або компрометації конфіденційної інформації.

Обмежений контроль,бо користувачі публічних хмар мають обмежений контроль над інфраструктурою та ресурсами, що використовуються для запуску їхніх застосунків. Це може ускладнити налаштування середовища відповідно до специфічних вимог.

Залежність від інтернет-з'єднання, бо для доступу до ресурсів та застосунків, розміщених у публічній хмарі, потрібне надійне та стабільне інтернет-з'єднання. Повільне або нестабільне з'єднання може вплинути на продуктивність та доступність послуг.

Простої в роботі, бо провайдери публічних хмар можуть стикатися з простоями в роботі через апаратні збої, проблеми з програмним забезпеченням або технічне обслуговування. Це може призвести до тимчасової втрати доступу до застосунків та даних.

Відповідність вимогам та нормативним актак,бо публічні хмарні послуги можуть не відповідати певним вимогам щодо відповідності або нормативним актам, наприклад, щодо конфіденційності або безпеки даних. Це може створити юридичні або контрактні проблеми для бізнесу.

Перевищення витрат,бо оплата за публічні хмарні послуги зазвичай здійснюється за принципом "плати за використання", що може призвести до непередбачених перевитрат, якщо використання перевищує очікуваний рівень. Крім того, вартість використання публічних хмарних послуг може зростати з часом, оскільки провайдери коригують свої цінові моделі або додають нові функції та послуги.

Приватна хмара - це хмарне середовище, де інфраструктура та послуги належать і управляються однією організацією, наприклад, компанією або урядом, і доступ до них мають лише авторизовані користувачі цієї організації. Організації, що використовують приватну хмару, мають власний центр обробки даних. Приватна хмара забезпечує вищий рівень безпеки. Приклади - HPE, Dell, VMware тощо.Переваги приватної хмари включають в себе:

Приватні хмари забезпечують вищий рівень безпеки, оскільки організація має повний контроль над хмарними послугами. Вони можуть налаштовувати сервери для управління своєю безпекою.

Приватні хмари дозволяють організаціям налаштовувати інфраструктуру та послуги відповідно до своїх специфічних вимог, а також налаштовувати безпеку.

Приватні хмари забезпечують підвищену конфіденційність, оскільки організація (компанія або уряд) має більше контролю над тим, хто має доступ до їхніх даних та ресурсів.

До недоліків приватної хмари можна віднести:

Висока вартість,бо приватні хмари потребують виділеного обладнання, програмного забезпечення та мережевої інфраструктури, придбання та обслуговування яких може бути дорогим. Це може ускладнити впровадження приватної хмари для малого бізнесу або організацій з обмеженим бюджетом.

Обмежена масштабованість, бо приватні хмари призначені для обслуговування конкретної організації, що означає, що вони можуть бути не такими масштабованими, як публічні хмарні послуги. Це може ускладнити швидке додавання або видалення ресурсів у відповідь на зміни попиту.

Технічна складність,бо налаштування та управління інфраструктурою приватної хмари потребує технічних знань та спеціалізованих навичок. Це може бути проблемою для організацій, які не мають власних IT-ресурсів або експертизи.

Ризики безпеки,бо приватні хмари зазвичай вважаються більш безпечними, ніж публічні, оскільки вони працюють в рамках власної інфраструктури організації. Однак, вони все ще можуть бути вразливими до ризиків безпеки, таких як витоки даних або кібератаки.

Відсутність стандартизації,бо приватні хмари часто будуються з використанням власного обладнання та програмного забезпечення, що може ускладнити інтеграцію з іншими хмарними послугами або міграцію до іншого постачальника хмарних послуг у майбутньому.

Обслуговування та оновлення інфраструктури приватної хмари може бути трудомістким та ресурсоємним. Це може бути проблемою для організацій, яким потрібно зосередитися на інших основних видах діяльності.

Гібридна хмара - це поєднання публічних та приватних хмарних середовищ, що дозволяє організаціям скористатися перевагами обох типів хмар. Вона ефективно керує рівнями трафіку в періоди пікового навантаження. Гібридна хмара може забезпечити більшу гнучкість, масштабованість та економічну ефективність, ніж використання одного хмарного середовища. Приклади - IBM, DataCore Software, Rackspace, Threat Stack, Infinidat тощо.

Переваги гібридної хмари:

Гібридна хмара зберігає дані (включаючи конфіденційні) на приватному хмарному сервері, тоді як публічний сервер забезпечує гнучкість та масштабованість.

Гібридна хмара дозволяє організаціям переміщувати робочі навантаження між приватними та публічними хмарами залежно від їхніх потреб.

Гібридна хмара забезпечує контроль над конфіденційними даними та високий рівень безпеки, одночасно використовуючи переваги економії коштів публічної хмари.

Недоліки гібридної хмари:

Гібридні хмари складні в налаштуванні та управлінні, оскільки вони потребують інтеграції між різними хмарними середовищами. Це може вимагати спеціалізованих технічних знань та ресурсів.

Гібридні хмари можуть бути дорожчими у впровадженні та управлінні, ніж публічні або приватні хмари окремо, через необхідність додаткового обладнання, програмного забезпечення та мережевої інфраструктури.

Гібридні хмари вразливі до ризиків безпеки, таких як витоки даних або кібератаки, особливо за відсутності стандартизації та узгодженості між різними хмарними середовищами.

Управління даними в різних хмарних середовищах може бути складним завданням, особливо коли йдеться про забезпечення відповідності таким нормативним актам, як GDPR або HIPAA.

Гібридні хмари залежать від зв'язку між різними хмарними середовищами, що може призвести до затримки мережі та проблем з продуктивністю.

Інтеграція різних хмарних середовищ може бути складною, особливо коли йдеться про забезпечення сумісності між різними застосунками та послугами.

Гібридні хмари можуть вимагати від організацій співпраці з кількома постачальниками хмарних послуг, що може призвести до прив'язки до постачальника та обмежити можливість переходу до іншого постачальника в майбутньому.[3]

Розуміння різниці між публічними, приватними та гібридними хмарами є важливим для правильного вибору інфраструктури. Тому нижче наведено таблицю з порівнянням:

Таблиця 1.1

Порівняння хмарних моделей

Фактор	Публічна хмара	Приватна хмара	Гібридна хмара
Ресурси	Спільне використання ресурсів між кількома клієнтами	Ресурси використовуються однією організацією	Поєднання публічної та приватної хмар, залежно від потреб
Оренда	Дані кількох організацій зберігаються в публічній хмарі	Дані однієї організації зберігаються в хмарі	Дані зберігаються в публічній хмарі, але з забезпеченням безпеки
Модель оплати	Платіть за використання	Різноманітні цінові моделі	Може включати комбінацію оплати за використання в публічній хмарі та фіксованої ціни в приватній хмарі. Також доступні інші моделі, такі як оплата за споживання, підписка тощо
Керування	Сторонній постачальник послуг	Конкретна організація	Може бути комбінацією обох
Масштабованість та гнучкість	Висока масштабованість та гнучкість	Передбачуваність та стабільність	Масштабованість та гнучкість завдяки поєднанню публічних та приватних хмарних послуг
Вартість	Найменш дорога	Найдорожча	Може бути дорожчою або дешевшою, залежно від потреб та вимог організації
Доступність	Загальнодоступна (через Інтернет)	Обмежена для конкретної організації	Може бути комбінацією обох

Хмарні технології характеризуються рядом ключових особливостей[9]:

1. Масштабованість - можливість швидко та легко збільшувати або зменшувати обсяг ресурсів в залежності від потреб.
2. Еластичність - автоматичне адаптування до змін навантаження, забезпечуючи оптимальне використання ресурсів.
3. Доступність - високий рівень доступності застосунків завдяки резервуванню та географічному розподілу ресурсів.
4. Pay-as-you-go - оплата лише за фактично використані ресурси, що дозволяє знизити витрати.

Розглянемо різновиди хмарних сервісів[2,9]:

— IaaS (Інфраструктура як сервіс) надає доступ до базових обчислювальних ресурсів, таких як віртуальні машини, мережі та сховища даних. Користувачі самостійно встановлюють та налаштовують операційні системи та програмне забезпечення. Прикладами IaaS є Google Compute Engine, Amazon EC2;

— PaaS (Платформа як сервіс) надає платформу для розробки, тестування та розгортання застосунків, включаючи операційну систему, середовище виконання, бази даних та інші інструменти. Прикладами PaaS є Google App Engine, Heroku;

— SaaS (Програмне забезпечення як сервіс) надає готове програмне забезпечення через Інтернет. Користувачі отримують доступ до застосунків через веб-браузер або API, не потребуючи встановлення та налаштування програмного забезпечення на своїх пристроях. Прикладами SaaS є Google Workspace, Salesforce.

Використання хмарних технологій для веб-застосунків надає ряд суттєвих переваг[6]:

1. Масштабованість - хмарні платформи дозволяють швидко масштабувати веб-застосунки в залежності від змін навантаження. Це особливо важливо для застосунків з мінливою кількістю користувачів, наприклад, для

інтернет-магазинів або соціальних мереж. Замість інвестування в додаткове обладнання, можна просто збільшити обсяг ресурсів в хмарі.

2. Еластичність - хмарні платформи автоматично адаптуються до змін навантаження, забезпечуючи оптимальне використання ресурсів. Наприклад, якщо кількість запитів до веб-застосунку зростає, хмарна платформа автоматично збільшить кількість серверів, що обробляють ці запити. Це дозволяє уникнути перевантажень та забезпечити стабільну роботу застосунку.
3. Доступність - хмарні провайдери забезпечують високий рівень доступності веб-застосунків завдяки резервуванню та географічному розподілу ресурсів. Навіть у разі виходу з ладу одного з дата-центрів, застосунок продовжуватиме працювати завдяки резервним копіям в інших дата-центрах.
4. Економічна ефективність - використання хмарних технологій дозволяє знизити витрати на розробку та розгортання веб-застосунків. Оплата здійснюється лише за фактично використані ресурси, що дозволяє уникнути капітальних витрат на обладнання та його обслуговування.
5. Гнучкість - хмарні платформи пропонують широкий вибір сервісів та конфігурацій, що дозволяє вибрати оптимальне рішення для кожного конкретного веб-застосунку. Наприклад, можна вибрати різні типи баз даних, сервіси кешування, інструменти моніторингу тощо.
6. Безпека - хмарні провайдери інвестують значні кошти в забезпечення безпеки даних. Вони використовують сучасні технології захисту інформації, такі як шифрування, фаєрволи, системи виявлення вторгнень тощо.
7. Швидкість розгортання - хмарні платформи дозволяють швидко розгорнути нові веб-застосунки та оновлення. Це значно скорочує час виведення продукту на ринок.

Поряд з перевагами, хмарні технології мають і певні обмеження:

1. Залежність від хмарного провайдера,бо користувачі хмарних сервісів залежать від політики та рішень хмарного провайдера.

2. Питання безпеки та конфіденційності даних, бо зберігання даних в хмарі може викликати побоювання щодо їх безпеки та конфіденційності.
3. Проблеми з латентністю та доступністю мережі,бо доступ до хмарних сервісів залежить від якості інтернет-з'єднання.
4. Складність міграції до хмари, бо перенесення існуючих застосунків до хмари може бути складним та вимагати значних зусиль.
5. Враховуючи всі переваги та обмеження, можна зробити висновок, що хмарні технології є перспективним напрямком розвитку IT-індустрії та надають широкі можливості для його розвитку.

1.2 Архітектура мережевих застосувань

Сучасні веб-застосунки все частіше будуються як складні розподілені системи, що вимагають ретельного вибору архітектурних підходів для забезпечення масштабованості, надійності та ефективності.[4] У цьому розділі розглянемо ключові архітектурні патерни, що застосовуються для побудови масштабованих веб-застосунків, з акцентом на їх переваги та недоліки в контексті хмарного середовища Google Cloud Platform.[6]

1.2.1 Мікросервісна архітектура

Мікросервісна архітектура - це сучасний підхід до розробки веб-застосунків, де велика програма розбивається на маленькі, незалежні модулі, які називаються мікросервісами. Кожен мікросервіс виконує свою окрему функцію, наче маленький самостійний застосунок, і спілкується з іншими мікросервісами через API. Це як оркестр, де кожен музикант грає свою партію, але разом вони створюють гармонійну мелодію.

Завдяки такому підходу, кожен мікросервіс можна розробляти, оновлювати та масштабувати незалежно від інших. Це дозволяє швидше реагувати на зміни,

ефективніше використовувати ресурси та підвищувати надійність системи. Уявіть, що один музикант в оркестрі помилився. В мікросервісній архітектурі це не зупинить всю музику, а лише вплине на окрему партію.
Звичайно, мікросервіси мають і свої складнощі. Управління великою кількістю мікросервісів може бути непростим завданням. Також потрібно забезпечити узгодженість даних між ними та ретельно налаштувати моніторинг для відстеження роботи кожного сервісу.

Попри ці складнощі, мікросервіси є популярним вибором для багатьох великих компаній, таких як Netflix, Amazon, Uber та Spotify, оскільки вони дозволяють створювати гнучкі, масштабовані та надійні застосунки. Мікросервіси часто використовуються разом з контейнеризацією (наприклад, Docker) та оркестрацією контейнерів (наприклад, Kubernetes).

Для комунікації між мікросервісами можуть використовуватись різні протоколи, такі як HTTP, gRPC та message queues (наприклад, RabbitMQ, Kafka).

Існують різні патерни проектування мікросервісів, такі як API Gateway, Service Mesh та Circuit Breaker, які допомагають вирішувати типові проблеми, пов'язані з мікросервісною архітектурою.

1.2.2 Безсерверна архітектура

Якщо уявити, що ви пишете програму, але не турбуєтесь про сервери, на яких вона буде працювати. Вам не потрібно їх купувати, налаштовувати, оновлювати чи стежити за їхньою роботою. Це і є суть безсерверної архітектури. Ваш код просто виконується у відповідь на певні події, наприклад, коли користувач надсилає запит на ваш сайт або коли змінюються дані в базі даних.

Хмарний провайдер, такий як Google Cloud, AWS чи Azure, бере на себе всю роботу з управління інфраструктурою. Він автоматично масштабує ресурси залежно від навантаження, тому вам не потрібно турбуватися про те, чи витримає

ваш застосунок наплив користувачів. Ви платите лише за фактично використані ресурси, що може значно знизити витрати.

Безсерверна архітектура дозволяє розробникам зосередитися на написанні коду та бізнес-логіці, а не на налаштуванні серверів. Це прискорює розробку та дозволяє швидше виводити нові продукти на ринок.

Звичайно, безсерверна архітектура має і свої обмеження. Наприклад, хмарні провайдери можуть встановлювати обмеження на час виконання коду, а відлагодження застосунків може бути складнішим через розподілену природу виконання. Також існує ризик залежності від конкретного хмарного провайдера (vendor lock-in).

Популярні приклади безсерверних платформ: AWS Lambda, Google Cloud Functions та Azure Functions. Вони дозволяють запускати код на різних мовах програмування та інтегруються з іншими хмарними сервісами:

— безсерверна архітектура добре підходить для невеликих застосунків, мікросервісів та завдань, які виконуються нерегулярно;
— вона часто використовується для обробки подій, створення API та розробки бекенду для мобільних застосунків;
— безсерверні платформи зазвичай надають інструменти для моніторингу, логування та відлагодження застосунків.

1.2.3 Розподілені системи та їх особливості

Розподілені системи - це системи, де обчислення та дані розподілені між кількома комп'ютерами, об'єднаними мережею. Уявіть собі велику компанію з багатьма відділами, які працюють разом для досягнення спільної мети. Кожен відділ має свої завдання та ресурси, але вони постійно обмінюються інформацією та співпрацюють один з одним.

Так само і в розподілених системах: кожен комп'ютер виконує свою частину роботи, але разом вони утворюють єдину систему, здатну вирішувати складні

завдання. Це дозволяє досягти високої продуктивності, масштабованості та надійності. Якщо один комп'ютер виходить з ладу, інші можуть продовжувати роботу, забезпечуючи безперервність сервісу.

Розподілені системи є основою для багатьох сучасних технологій, таких як хмарні обчислення, великі дані та Інтернет речей. Вони дозволяють нам користуватися пошуковими системами, соціальними мережами, онлайн-магазинами та багатьма іншими сервісами, які стали невід'ємною частиною нашого життя:

— розподілені системи можуть бути різних типів: клієнт-серверні, peer-to-peer, хмарні тощо;
— для розробки розподілених систем використовуються різні технології та підходи, такі як мікросервіси, безсерверна архітектура та розподілені бази даних;
— важливими аспектами розподілених систем є узгодженість даних, обробка конкурентних запитів, затримки в мережі та забезпечення безпеки;
— розподілені системи постійно розвиваються та вдосконалюються, що веде до появи нових технологій та можливостей.

1.3 Огляд платформи Google Cloud Platform (GCP) та її сервісів

Google Cloud Platform (GCP) - це комплексна хмарна платформа, що пропонує широкий спектр сервісів для розробки, розгортання та масштабування застосунків. GCP працює на тій самій інфраструктурі, що й пошукова система Google, YouTube та інші продукти Google, забезпечуючи високу надійність, продуктивність та безпеку.[7]

У цьому розділі розглянемо ключові сервіси GCP, які є особливо корисними для розробки та розгортання масштабованих веб-застосунків.

Рис. 1.1 Огляд сервісів Google Cloud

1.3.1 Google Compute Engine

Google Compute Engine - це як конструктор для створення віртуальних комп'ютерів у хмарі Google. Ви можете обрати потрібну вам конфігурацію, встановити будь-яку операційну систему та легко змінювати потужність вашого віртуального комп'ютера залежно від потреб. Це наче мати власний комп'ютер, але він знаходиться в дата-центрі Google і ви можете до нього підключитися з будь-якої точки світу. Compute Engine надає велику гнучкість та контроль над вашими віртуальними машинами. Ви можете використовувати його для різних завдань, таких як розміщення веб-сайтів, запуск баз даних, обробка даних, машинне навчання та багато іншого. Він також легко інтегрується з іншими сервісами Google Cloud, такими як Cloud Storage для зберігання даних та Kubernetes Engine для управління контейнерами.

— Compute Engine пропонує широкий вибір віртуальних машин з різними характеристиками та цінами;

— ви можете створювати власні образи віртуальних машин або використовувати готові образи з публічної бібліотеки Google;
— Compute Engine забезпечує високу доступність та надійність ваших віртуальних машин;
— ви можете автоматизувати управління віртуальними машинами за допомогою інструментів командного рядка або API;
— Compute Engine є частиною Google Cloud Platform, що надає широкий спектр хмарних сервісів для різних потреб.

1.3.2 Google App Engine

Google App Engine - це як чарівна платформа, де ваші веб-застосунки та API оживають без зайвого клопоту. Вам не потрібно турбуватися про сервери, їх налаштування та обслуговування - App Engine зробить все за вас. Він автоматично підлаштовується під кількість відвідувачів, збільшуючи або зменшуючи потужність, щоб ваш сайт завжди працював швидко та безперебійно.

App Engine підтримує найпопулярніші мови програмування, такі як Java, Python, PHP, Go та Node.js. Він також надає доступ до вбудованих сервісів, таких як бази даних, кешування та черги завдань, що значно спрощує розробку. Крім того, App Engine легко інтегрується з іншими сервісами Google Cloud, такими як Cloud Storage та Cloud SQL.

Якщо ви розробляєте веб-застосунок, мобільний бекенд або API, який повинен бути швидким, надійним та доступним для великої кількості користувачів, Google App Engine - це відмінний вибір. Він дозволить вам зосередитися на розробці вашого продукту, а не на управлінні інфраструктурою.

— App Engine має гнучку цінову політику, де ви платите лише за фактично використані ресурси;
— App Engine забезпечує високий рівень безпеки та захисту даних;
— App Engine має простий та зручний інтерфейс управління;

— App Engine підтримує різні середовища виконання, включаючи стандартне середовище та гнучке середовище;
— App Engine є частиною Google Cloud Platform, що надає широкий спектр хмарних сервісів для різних потреб.

1.3.3 Google Kubernetes Engine

Google Kubernetes Engine (GKE) - це як диригент оркестру для ваших застосунків, упакованих в контейнери. Kubernetes, система, на якій базується GKE, допомагає автоматично розподіляти "музикантів" (контейнери) по "сцені" (серверах), слідкувати за їх роботою та забезпечувати гармонійне звучання всього "оркестру" (застосунку).

GKE бере на себе всі рутинні завдання з управління контейнерами, такі як розгортання, масштабування, оновлення та моніторинг. Він автоматично додає або видаляє контейнери залежно від навантаження, розподіляє трафік між ними та перезапускає їх у разі збоїв. Це дозволяє вам зосередитися на розробці застосунку, а не на управлінні інфраструктурою.

GKE ідеально підходить для розгортання мікросервісних застосунків, де кожен мікросервіс працює в окремому контейнері. Це забезпечує високу гнучкість та масштабованість, оскільки ви можете незалежно масштабувати кожен мікросервіс залежно від його потреб:

— GKE інтегрується з іншими сервісами Google Cloud, такими як Cloud SQL, Cloud Storage та Cloud Monitoring, що спрощує розробку та управління застосунками;
— GKE підтримує різні мережеві конфігурації та забезпечує високий рівень безпеки;
— GKE має гнучку цінову політику та простий інтерфейс управління;
— GKE є одним з найпопулярніших керованих сервісів Kubernetes у світі.

1.3.4 Google Cloud Storage

Google Cloud Storage - це як бездонна скриня для ваших даних, де можна зберігати все, що завгодно: від фотографій та відео до важливих документів та резервних копій. Цей сервіс надійно зберігає ваші файли та забезпечує до них швидкий доступ з будь-якої точки світу.

Ви можете думати про Cloud Storage як про величезний жорсткий диск в Інтернеті, який ніколи не заповнюється і завжди доступний. Він автоматично створює копії ваших файлів в різних дата-центрах, тому навіть якщо один дата-центр виходить з ладу, ваші дані залишаються в безпеці.

Cloud Storage пропонує різні варіанти зберігання даних залежно від ваших потреб. Якщо вам потрібен швидкий доступ до файлів, ви можете використовувати стандартний клас сховища. Якщо ж ви зберігаєте дані, до яких рідко звертаєтесь, ви можете обрати дешевший клас сховища для архівування.

Cloud Storage легко інтегрується з іншими сервісами Google Cloud, такими як Compute Engine та App Engine. Ви можете використовувати його для зберігання медіафайлів для вашого веб-сайту, резервних копій ваших баз даних або логів ваших застосунків:

— Cloud Storage має гнучку цінову політику та простий інтерфейс управління;
— Cloud Storage підтримує різні інструменти та API для доступу до даних;
— Cloud Storage забезпечує високий рівень безпеки та захисту даних за допомогою шифрування та контролю доступу;
— Cloud Storage є частиною Google Cloud Platform, що надає широкий спектр хмарних сервісів для різних потреб.

1.3.5 Google Cloud SQL

Google Cloud SQL - це зручний та надійний сервіс для роботи з базами даних в хмарі. Уявіть собі, що вам не потрібно встановлювати та налаштовувати базу даних на власному сервері, а також турбуватися про її обслуговування, резервне копіювання та оновлення. Cloud SQL бере на себе всі ці завдання, дозволяючи вам зосередитися на розробці вашого застосунку.

Cloud SQL підтримує популярні реляційні бази даних, такі як MySQL, PostgreSQL та SQL Server. Він автоматично створює резервні копії, захищаючи ваші дані від втрат, та забезпечує високу доступність, щоб ваш застосунок завжди мав доступ до даних. Ви також можете легко масштабувати вашу базу даних залежно від потреб, додаючи ресурси або створюючи репліки.

Cloud SQL інтегрується з іншими сервісами Google Cloud, такими як Compute Engine та Kubernetes Engine, що дозволяє легко розгортати та управляти вашими застосунками. Він також надає інструменти для моніторингу та оптимізації продуктивності баз даних.

Якщо вам потрібна надійна, масштабована та доступна база даних для вашого застосунку, Google Cloud SQL - це відмінний вибір. Він знімає з вас тягар управління базами даних та дозволяє зосередитися на розробці вашого продукту;

- Cloud SQL має гнучку цінову політику та простий інтерфейс управління.
- Cloud SQL забезпечує високий рівень безпеки та захисту даних за допомогою шифрування та контролю доступу;
- Cloud SQL підтримує різні версії баз даних та дозволяє налаштовувати їх відповідно до ваших потреб;
- Cloud SQL є частиною Google Cloud Platform, що надає широкий спектр хмарних сервісів для різних завдань.

1.3.6 BigQuery

BigQuery - це як гігантський суперкомп'ютер для аналізу даних, який живе в хмарі Google. Уявіть собі, що вам потрібно проаналізувати величезну кількість інформації, наприклад, всі транзакції в інтернет-магазині за останні декілька років. На звичайному комп'ютері це може зайняти дні або навіть тижні. BigQuery ж здатний виконати такий аналіз за лічені секунди!

Секрет BigQuery полягає в його унікальній архітектурі. Він розподіляє дані між тисячами серверів та використовує спеціальні алгоритми для швидкої обробки запитів. Вам не потрібно турбуватися про налаштування або обслуговування серверів, BigQuery робить все це автоматично. Ви просто завантажуєте свої дані та запускаєте SQL-запити, а BigQuery миттєво надає вам результати.

BigQuery використовується в різних сферах, від бізнес-аналітики до наукових досліджень. Він допомагає компаніям аналізувати дані про клієнтів, продажі, маркетинг та фінанси. Вчені використовують BigQuery для обробки великих наборів даних в галузі геноміки, астрономії та інших наук. BigQuery також може бути використаний для аналізу логів, виявлення аномалій та прогнозування майбутніх подій.

BigQuery підтримує різні формати даних, включаючи CSV, JSON, Avro та Parquet.

BigQuery має вбудовані функції машинного навчання, які дозволяють створювати та розгортати моделі безпосередньо в BigQuery.
BigQuery має інтеграцію з іншими сервісами Google Cloud, такими як Cloud Storage, Dataflow та Dataproc.

BigQuery має гнучку цінову політику, де ви платите лише за обсяг даних, які ви зберігаєте та аналізуєте.

1.3.7 Інші сервіси GCP

Google Cloud Platform (GCP) надає широкий спектр сервісів, що виходять за рамки базових обчислювальних ресурсів, сховищ даних та баз даних. Ці сервіси відіграють важливу роль у забезпеченні масштабованості, надійності, безпеки та ефективності веб-застосунків.

В цьому підрозділі ми розглянемо деякі з ключових сервісів GCP, які можуть бути використані для розробки та розгортання сучасних веб-застосунків.

Cloud Load Balancing - це керований сервіс балансування навантаження, який розподіляє трафік між кількома інстансами застосунку. Це дозволяє забезпечити високу доступність, масштабованість та ефективне використання ресурсів. Cloud Load Balancing підтримує різні типи балансувальників навантаження, включаючи HTTP(S), TCP та UDP балансувальники.

Cloud CDN (Content Delivery Network) - це глобально розподілена мережа серверів, яка кешує статичний контент (зображення, відео, скрипти) ближче до користувачів. Це дозволяє прискорити завантаження веб-сторінок та зменшити затримки для користувачів в різних частинах світу.

Cloud Monitoring - це сервіс для моніторингу та спостереження за станом застосунків та інфраструктури. Він збирає метрики, логи та події з різних джерел та надає інструменти для їх візуалізації, аналізу та сповіщення.

Cloud Logging - це керований сервіс для збору та зберігання логів застосунків та систем. Він дозволяє централізовано збирати логи з різних джерел, включаючи віртуальні машини, Kubernetes та інші сервіси GCP. Cloud Logging надає інструменти для пошуку, аналізу та експорту логів.

Cloud DNS - це керований сервіс DNS (Domain Name System), який дозволяє вам легко управляти DNS-записами вашого домену. Cloud DNS забезпечує високу доступність, масштабованість та безпеку.

Cloud Armor - це веб-застосунок брандмауер (WAF), який захищає ваші застосунки від DDoS-атак, SQL-ін'єкцій та інших веб-загроз. Cloud Armor дозволяє налаштовувати правила для фільтрації трафіку та блокування шкідливих запитів.

Google Cloud Platform надає широкий спектр сервісів, які можуть бути використані для розробки та розгортання масштабованих, надійних та безпечних веб-застосунків. Вибір конкретних сервісів залежить від вимог до застосунку та його архітектури. В цілому, GCP надає все необхідне для створення сучасних хмарних рішень.

1.4 Архітектурні патерни масштабованих веб-застосувань

Архітектурні патерни відіграють важливу роль у проектуванні масштабованих веб-застосувань. Вони надають перевірені рішення для типових проблем, що виникають при розробці систем, які повинні обробляти великі навантаження та забезпечувати високу доступність.

У цьому розділі розглянемо деякі ключові архітектурні патерни, що застосовуються для створення масштабованих веб-застосувань.[12]

1.4.1 Шарувата архітектура (Layered Architecture)

Шарувата архітектура - це популярний спосіб організації програмного забезпечення, який можна порівняти з багатоповерховим будинком. Кожен поверх має своє призначення: на першому поверсі знаходиться рецепція та магазини (інтерфейс користувача), на верхніх поверхах - офіси з працівниками, які виконують різні завдання (бізнес-логіка), а в підвалі - серверна кімната з усіма важливими системами (доступ до даних).

Такий поділ на шари дозволяє спростити розробку та підтримку застосунку. Кожен шар можна розробляти та тестувати незалежно від інших, що робить код більш модульним та гнучким. Якщо потрібно внести зміни до інтерфейсу користувача, це не вплине на роботу бізнес-логіки або доступу до даних.

Однак, важливо пам'ятати, що шари не повинні бути занадто тісно пов'язані між собою. Інакше застосунок може перетворитися на моноліт, який важко

змінювати та масштабувати. Уявіть собі, що в нашому будинку всі комунікації проходять через один єдиний канал в підвалі. Якщо цей канал пошкодиться, весь будинок залишиться без електрики, води та інтернету.[12]

— шарувата архітектура може мати різну кількість шарів залежно від складності застосунку;
— існують різні варіації шаруватої архітектури, такі як MVC (Model-View-Controller), MVP (Model-View-Presenter) та MVVM (Model-View-ViewModel);
— шарувата архітектура є одним з найпоширеніших патернів проектування і використовується в багатьох різних типах застосунків.

1.4.2 Архітектура, керована подіями (Event-Driven Architecture)

Подійно-орієнтована архітектура (EDA) - це спосіб побудови програм, де різні частини програми спілкуються між собою, обмінюючись повідомленнями про події. Уявіть собі групу людей, які працюють над спільним проектом. Кожна людина відповідає за свою частину роботи, але коли хтось завершує своє завдання, він повідомляє про це інших. Так само і в EDA: різні частини програми "слухають" події та реагують на них.[12]

Цей підхід дозволяє створювати дуже гнучкі програми, де можна легко додавати нові функції або змінювати існуючі, не впливаючи на інші частини програми. EDA також допомагає підвищити продуктивність та надійність програм, оскільки різні частини можуть працювати паралельно та незалежно одна від одної.

EDA широко використовується в різних сферах, наприклад, в веб-застосунках, мобільних додатках, іграх та системах Інтернету речей. Вона дозволяє створювати складні та динамічні системи, які можуть адаптуватися до змін та обробляти великі обсяги даних.

— EDA часто використовується разом з іншими архітектурними патернами, такими як мікросервіси та безсерверна архітектура;

— для обробки подій можуть використовуватися різні технології, такі як message queues (черги повідомлень) та event buses (шини подій);

— EDA є важливим інструментом для створення сучасних розподілених систем.

1.4.3 Патерн "Команда-запит поділу відповідальності" (CQRS)

CQRS (Command Query Responsibility Segregation) - це підхід до розробки програмного забезпечення, який розділяє операції читання та запису даних. Уявіть собі бібліотеку, де є окремі зали для читання та для видачі книг. Це дозволяє оптимізувати кожен процес окремо: в залі для читання можна створити тиху та спокійну атмосферу, а в залі для видачі - організувати швидку та ефективну роботу з книгами.[12]

Так само і в CQRS: операції читання та запису даних обробляються окремо, що дозволяє покращити продуктивність та масштабованість застосунку. Наприклад, для читання даних можна використовувати просту та швидку базу даних, а для запису - більш складну та надійну.

CQRS також надає більше гнучкості, оскільки дозволяє використовувати різні моделі даних для читання та запису. Це може бути корисним в ситуаціях, коли дані для читання та запису мають різну структуру або вимоги до обробки. Однак, CQRS додає складності до проектування та розробки застосунку, тому його варто використовувати лише тоді, коли це дійсно необхідно.

1.5 Методи забезпечення масштабованості веб-застосувань

Масштабованість веб-застосунків - це їхня здатність адаптуватися до зростання, немов дитина, яка з часом стає дорослою. Спочатку застосунок може бути невеликим та простим, але з розвитком бізнесу та збільшенням кількості користувачів він повинен "вирости" та навчитися обробляти більше даних і запитів, не втрачаючи при цьому своєї швидкості та стабільності.

Щоб застосунок міг успішно "дорослішати", потрібно застосовувати різні методи та техніки масштабування. Це як правильне харчування та фізичні вправи для дитини: вони допомагають їй рости здоровою та сильною. Масштабування дозволяє застосунку впоратися з навантаженням, забезпечити його безперебійну роботу та позитивний досвід для користувачів.[12,14]

Основні аспекти масштабованості:

1. Застосунок повинен швидко обробляти запити та надавати відповіді, навіть якщо одночасно звертається багато користувачів.
2. Застосунок повинен бути доступним завжди, навіть під час пікових навантажень або технічних збоїв.
3. Застосунок повинен ефективно використовувати ресурси (сервери, пам'ять, мережу), щоб мінімізувати витрати.
4. Застосунок повинен легко адаптуватися до змін вимог та навантаження, немов дитина, яка швидко вчаться новому.

1.5.1 Горизонтальне та вертикальне масштабування

Коли веб-застосунок починає зростати та обслуговувати більше користувачів, виникає потреба в масштабуванні, тобто в збільшенні його потужності для обробки зростаючого навантаження. Існує два основних підходи до масштабування: вертикальне та горизонтальне[10,11].

Вертикальне масштабування - це як "апгрейд" комп'ютера. Ви збільшуєте потужність існуючого сервера, на якому працює ваш застосунок, додаючи більше ресурсів,[10] таких як:

— центральний процесор (CPU) - збільшення кількості ядер або тактової частоти процесора;

— оперативна пам'ять (RAM) - збільшення обсягу оперативної пам'яті;

— дисковий простір - використання швидших та місткіших дисків, наприклад, SSD замість HDD.

Вертикальне масштабування зазвичай легше реалізувати, оскільки воно не вимагає змін в архітектурі застосунку. Збільшення ресурсів на одному сервері може зменшити затримку в порівнянні з розподілом навантаження на кілька серверів.До мінусів,існує фізичний ліміт на кількість ресурсів, які можна додати до одного сервера. Якщо сервер виходить з ладу, весь застосунок стає недоступним.Вертикальне масштабування може бути дорожчим, особливо при використанні хмарних сервісів, де оплата нараховується за обсяг використаних ресурсів.

Горизонтальне масштабування - це як створення команди з кількох людей для виконання одного завдання. Ви додаєте більше серверів до інфраструктури застосунку та розподіляєте навантаження між ними.Ви можете додавати скільки завгодно серверів для обробки зростаючого навантаження.Якщо один сервер виходить з ладу, інші сервери продовжують працювати, забезпечуючи безперервність роботи застосунку. Ви можете легко додавати або видаляти сервери залежно від потреб.[11]

Недоліки горизонтального масштабування:

— горизонтальне масштабування вимагає більш складної інфраструктури та налаштування;

— хоча горизонтальне масштабування може бути більш економічним в довгостроковій перспективі, воно все одно вимагає інвестицій в додаткові сервери.

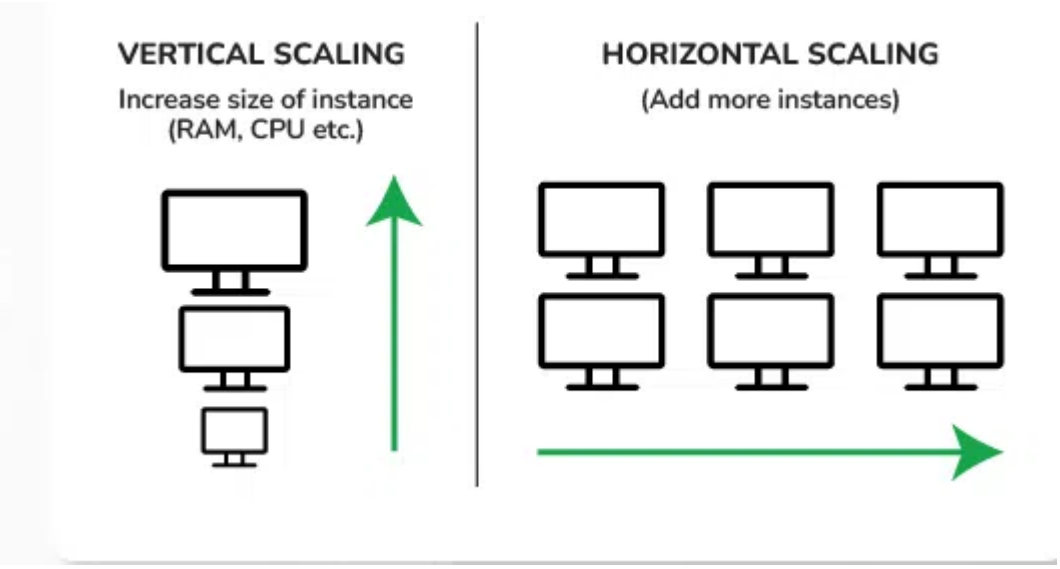


Рис. 1.2 – Горизонтальне і вертикальне масштабування

1.5.2 Балансування навантаження

Балансування навантаження - це як розумний диспетчер у великому офісі, який розподіляє вхідні дзвінки між вільними операторами. Замість того, щоб всі дзвінки надходили на один номер і створювали чергу, балансувальник навантаження спрямовує їх на різні сервери, забезпечуючи швидку та ефективну обробку запитів.[14]

У контексті веб-застосунків, балансувальник навантаження розподіляє вхідний трафік між кількома серверами, запобігаючи перевантаженню одного сервера та забезпечуючи високу доступність застосунку. Навіть якщо один сервер виходить з ладу, балансувальник навантаження автоматично перенаправляє трафік на інші сервери, і користувачі навіть не помічають проблеми.

Існують різні алгоритми, які використовуються для визначення, на який сервер спрямувати запит. Деякі з найпоширеніших алгоритмів:

— Round Robin - запити розподіляються по черзі між серверами;
— Least Connections - запити спрямовуються на сервер з найменшою кількістю активних з'єднань;

— IP Hash - запити від одного IP-адреси завжди спрямовуються на той самий сервер.

1.5.3 Кешування

Кешування - це як запам'ятовування важливої інформації, щоб не шукати її знову і знову. Уявіть, що вам потрібно часто користуватися довідником. Замість того, щоб щоразу гортати його з початку, ви можете запам'ятати найважливіші сторінки або зробити закладки. Це значно прискорить пошук потрібної інформації.

Так само і в веб-застосунках: кешування дозволяє зберігати часто використовувані дані в швидкій пам'яті (кеші), щоб не обробляти однакові запити знову і знову. Це зменшує навантаження на сервери, прискорює завантаження сторінок та знижує витрати на інфраструктуру.

Типи кешування:

1. Дані зберігаються в кеші веб-браузера користувача. Наприклад, браузер може зберігати зображення та скрипти, щоб не завантажувати їх з сервера при кожному відвідуванні сайту.
2. Дані зберігаються в кеші на сервері застосунку. Наприклад, сервер може кешувати результати запитів до бази даних, щоб не виконувати однакові запити знову і знову.
3. Дані зберігаються в розподіленій системі кешування, такій як Redis або Memcached. Це дозволяє розподілити навантаження на кеш між кількома серверами та забезпечити високу доступність.[13]

1.5.4 Оптимізація бази даних

База даних - це як серце веб-застосунку, де зберігається вся важлива інформація. Коли застосунок зростає, обсяг даних збільшується, і база даних може

стати "вузьким місцем", сповільнюючи роботу всього застосунку. Це як спробувати пропустити великий натовп через вузькі двері - виникне тиснява та затримки.[13]

Щоб запобігти цьому, потрібно оптимізувати базу даних, тобто налаштувати її для швидкої та ефективної роботи з великими обсягами даних. Це як розширити двері та організувати рух натовпу, щоб уникнути тисняви. Методи:

1. Індексація - створення індексів для прискорення пошуку даних.
2. Оптимізація запитів - написання ефективних SQL-запитів.
3. Реплікація - сворення копій бази даних для розподілу навантаження.
4. Шардування - розподіл даних між кількома серверами баз даних.

Крім перерахованих методів, існує багато інших способів оптимізації бази даних, таких як оптимізація запитів, використання процедур та функцій, налаштування параметрів сервера баз даних тощо.

Оптимізація бази даних є важливим аспектом забезпечення масштабованості веб-застосунків. Вона дозволяє підвищити продуктивність, зменшити час відгуку та підвищити доступність даних

1.6 Моніторинг та управління масштабованими веб-застосунками

У світі сучасних веб-застосунків, де величезні обсяги даних та запитів обробляються щосекунди, моніторинг та управління стають надзвичайно важливими. Ефективний моніторинг дозволяє нам зазирнути "під капот" застосунку та зрозуміти, як він працює, де є проблеми та як його покращити. Це як лікар, який стежить за життєво важливими показаннями пацієнта та вчасно реагує на будь-які відхилення.[13]

Моніторинг масштабованих веб-застосунків - це не лише відстеження завантаження сервера та часу відповіді. Це також аналіз логів, трасування запитів, виявлення помилок та забезпечення доступності застосунку. Це як комплексне

медичне обстеження, яке дозволяє отримати повну картину стану здоров'я пацієнта.

Для моніторингу використовуються різні інструменти, як вбудовані в хмарні платформи (наприклад, Google Cloud Monitoring), так і сторонні рішення (Datadog, New Relic). Ці інструменти збирають та аналізують дані, візуалізують метрики та надсилають сповіщення про потенційні проблеми.

Управління масштабованими веб-застосунками - це як керування великим оркестром. Диригент повинен забезпечити, щоб всі музиканти грали синхронно та гармонійно, а оркестр в цілому звучав чудово. Так само і адміністратор застосунку повинен управляти ресурсами, балансувати навантаження, налаштовувати кешування та оптимізувати базу даних.Для автоматизації управління можуть використовуватися різні інструменти, такі як системи конфігурації (Ansible, Puppet), системи оркестрації (Kubernetes) та системи CI/CD. Це як нотні записи для оркестру, які допомагають музикантам грати синхронно.

Ефективний моніторинг та управління - це ключ до успіху будь-якого масштабованого веб-застосунку. Вони дозволяють забезпечити високу продуктивність, доступність та надійність системи, а також швидко реагувати на зміни та проблеми.

РОЗДІЛ 2. РОЗРОБКА МОДУЛЯ МАСШТАБОВАНИХ ВЕБ-ЗАСТОСУВАНЬ

2.1 Аналіз вимог до модуля

Цей розділ є ключовим етапом у розробці модуля масштабованих веб-застосувань, оскільки тут визначається, що саме модуль повинен робити та які його характеристики. Чітке формулювання вимог є основою для подальшого проектування, реалізації та тестування. Враховуючи специфіку мого проекту, я визначив наступні ключові вимоги:

1. Функціональні вимоги (*обов'язкові*):

1. Модуль повинен забезпечувати автоматичне масштабування застосунку в Kubernetes, використовуючи Vertical Pod Autoscaler. Масштабування має відбуватися динамічно, на основі метрик, таких як використання CPU, пам'яті або кількість запитів. Важливою є підтримка різних стратегій масштабування для гнучкого налаштування.
2. Для розподілу трафіку між Pod'ами я планую використовувати Google Cloud Load Balancing. Модуль повинен інтегруватися з цим сервісом та підтримувати різні алгоритми балансування для оптимального розподілу навантаження.
3. Модуль повинен забезпечувати безперервну роботу застосунку навіть у разі збоїв. Для цього будуть використані механізми Kubernetes, такі як Liveness та Readiness probes, а також реплікація критичних компонентів. Моніторинг стану системи та сповіщення про помилки також є важливими аспектами відмовостійкості.
4. Модуль повинен легко інтегруватися з іншими сервісами Google Cloud, такими як Cloud SQL та Cloud Storage, а також надавати API для взаємодії з іншими системами.

5. Безпека є пріоритетом. Модуль повинен забезпечувати аутентифікацію та авторизацію користувачів, захист даних від несанкціонованого доступу та захист від поширених веб-атак.

Нефункціональні вимоги:

1. Модуль повинен забезпечувати високу продуктивність застосунку, мінімізуючи час відгуку та максимізуючи пропускну здатність.
2. Висока надійність та мінімальний час простою є ключовими вимогами.
3. Модуль повинен забезпечувати ефективне та швидке масштабування застосунку залежно від навантаження.
4. Модуль повинен відповідати стандартам безпеки та підтримувати аудит безпеки.
5. Код модуля повинен бути зрозумілим, добре документованим та легким для підтримки.
6. Модуль повинен мати можливість розгортання на різних кластерах Kubernetes.

Вимоги до інтерфейсу:

— модуль повинен надавати REST API для взаємодії з іншими системами. API повинні бути добре документовані та підтримувати версіонування;
— за необхідності, модуль може надавати CLI для керування та налаштування;
— за необхідності, модуль може мати веб-інтерфейс для моніторингу та управління.

Усі вимоги до модуля були чітко задокументовані у вигляді текстового документа та таблиці вимог. Враховуючи специфіку моєї роботи, я також визначив метрики, за якими буде оцінюватися ефективність модуля, такі як час відгуку, пропускна здатність, час безвідмовної роботи та використання ресурсів. Цей детальний аналіз вимог став основою для подальшого проектування та реалізації модуля масштабованих веб-застосувань.

2.2 Вибір сервісів GCP

У цьому розділі я зосередився на виборі доступних сервісів GCP для масштабованих веб-застосувань на Google Cloud Platform, використовуючи мою просту програму як практичний приклад. Основний акцент було зроблено на вибір та інтеграцію сервісів GCP для забезпечення масштабованості, надійності та ефективності. Однак, для того, щоб правильно обрати сервіси Google Cloud Platform (GCP), необхідно розуміти різницю між ними та їхні можливості. Найкраще у цьому може допомогти таблиця, яка дозволить порівняти ключові характеристики кожного сервісу.

Таблиця 2.1

Порівняння сервісів обчислення Google Cloud

Опція	**Підтримка мов**	**Модель використання**	**Масштабування**	**Основний сценарій використання**
Compute Engine	Будь-яка	IaaS	Автомасштабування сервера	Загальні робочі навантаження
GKE (Google Kubernetes Engine)	Будь-яка	IaaS, PaaS	Кластерне масштабування	Контейнерні робочі навантаження
App Engine (стандартне середовище)	Python, Node.js, Go, Java, Ruby, PHP	PaaS	Автоматичне масштабування керованих серверів	Масштабовані веб-застосунки, мобільні бекенд-застосунки
App Engine (гнучке середовище)	Python, Node.js, Go, Java, PHP, Ruby, .NET, власні середовища виконання	PaaS	Автоматичне масштабування керованих серверів	Масштабовані веб-застосунки, мобільні бекенд-застосунки

Продовження таблиці 2.1

Порівняння сервісів обчислення Google Cloud

Опція	**Підтримка мов**	**Модель використання**	**Масштабування**	**Основний сценарій використання**
Cloud Functions	Python, Node.js, Go, Java	Мікросервісна архітектура	Безсерверне	Легкі події, дії
Cloud Run	Будь-яка	PaaS	Безсерверне	Розгортання та масштабування контейнеризованих застосунків

Google Cloud пропонує широкий вибір сервісів для зберігання та управління даними, кожен з яких має свої унікальні переваги та призначений для вирішення конкретних завдань. Важливо правильно обрати сховище, де правильно буде зберігатися інформація.

Таблиця 2.2

Порівняння сервісів збереження даних Google Cloud

Тип	**Сервіс**	**Добре підходить для**	**Приклади використання**
Об'єктне сховище	Cloud Storage	Бінарних або об'єктних даних	Зображення, медіа-контент, резервні копії
Реляційні бази даних	Cloud SQL	Веб-фреймворків	CMS, електронна комерція
	Cloud Spanner	RDBMS + масштабування, висока доступність, сильна узгодженість	Метадані користувачів, рекламні/фінансові/маркетингові технології
Нереляційні бази даних	Cloud Firestore	Ієрархічних даних, мобільних та веб-застосунків	Профілі користувачів, стан гри
	Cloud Bigtable	Великих обсягів читання та запису, подій	Рекламні технології, фінанси, Інтернет речей
Хранилище даних	BigQuery	Корпоративних даних	Аналітика, інформаційні панелі

Враховуючи, що моя програма є відносно простою, я обрав наступні сервіси GCP для її розгортання та масштабування:

— Google Kubernetes Engine (GKE) для оркестрації та управління контейнеризованою програмою. GKE дозволить мені легко масштабувати застосунок, розподіляти навантаження та забезпечувати відмовостійкість;
— Cloud SQL для зберігання даних застосунку. Я обрав Cloud SQL for PostgreSQL, враховуючи потреби моєї програми;
— Cloud Storage для зберігання статичного контенту, такого як HTML, CSS та JavaScript файли. Cloud Storage забезпечить швидкий доступ до контенту та його ефективну доставку користувачам;
— Cloud Load Balancing для розподілу трафіку між інстансами застосунку. Cloud Load Balancing дозволить забезпечити високу доступність та ефективне використання ресурсів.

Вибір конкретного сервісу залежить від вимог вашого застосунку та ваших уподобань. Наприклад, якщо вам потрібна повна гнучкість та контроль, Compute Engine може бути хорошим вибором. Якщо ж вам потрібна простота та автоматичне масштабування, Cloud Run або App Engine можуть бути кращими варіантами.

Але іноді обрати потрібні сервіси стає складною роботою і щоб спростити це завдання я створив 2 блок схеми (використовуючи інформацію з Таблиці 2.1 і Таблиці 2.2) для швидкого і правильного вибору:

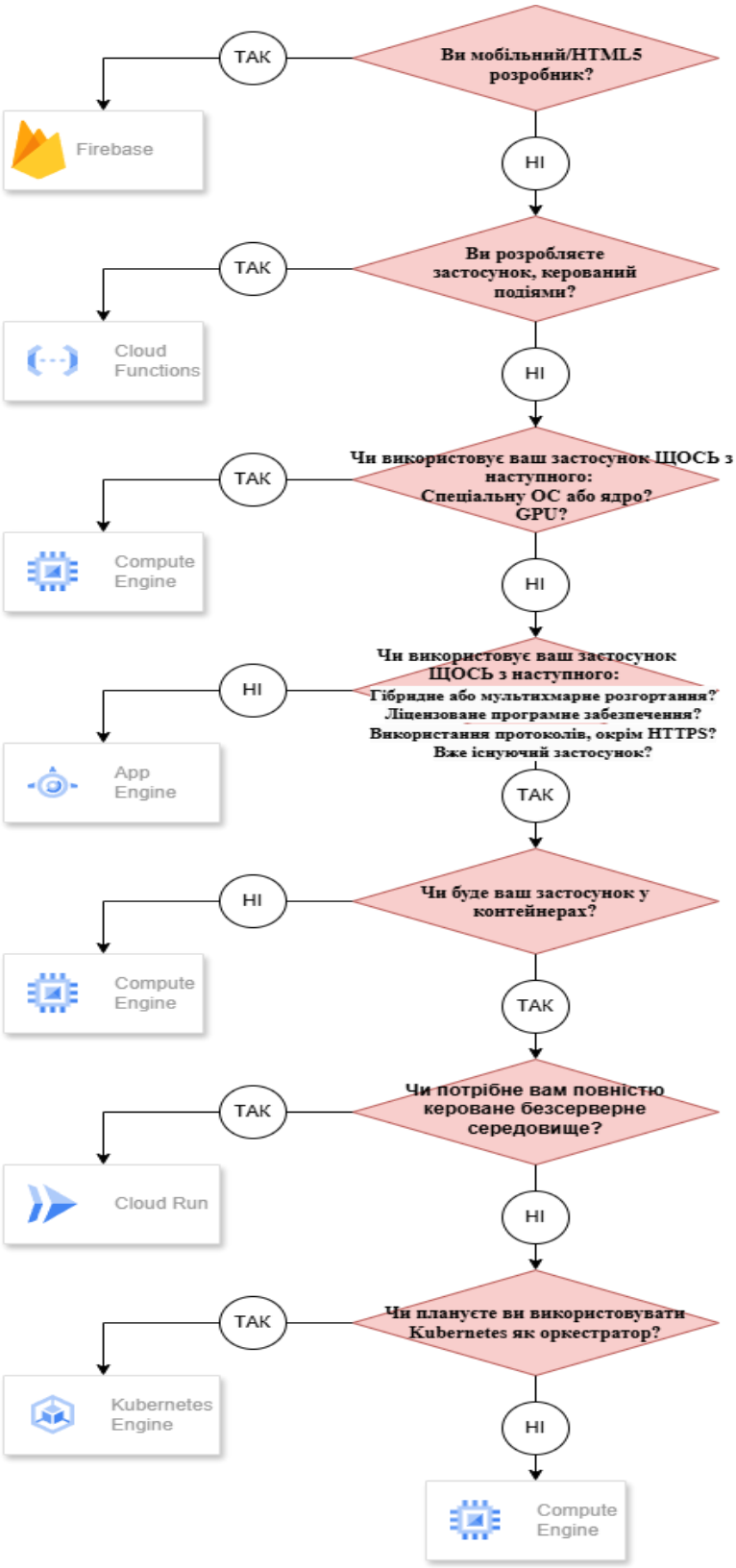


Рис 2.1 Блок-схема для вибору сервісу обчислення

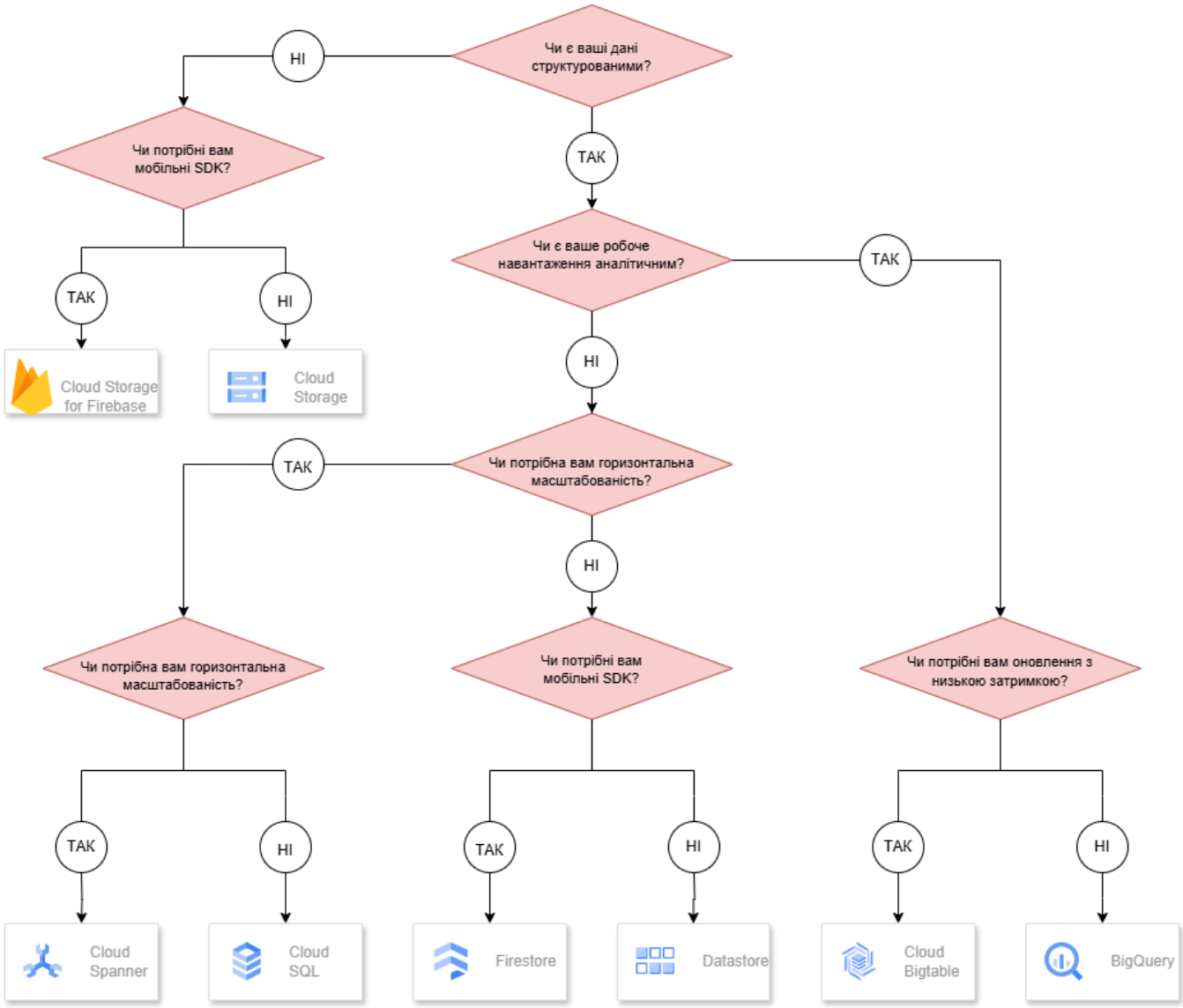


Рис 2.2 Блок-схема для вибору сервісу зберігання даних

2.3 Архітектура модуля

Архітектура мого модуля виглядає наступним чином:

1. Моя програма на Java/HTML/CSS (вихідний код занесений у Додаток Б) була упакована в Docker контейнер. Це забезпечило портативність та ізоляцію застосунку.
2. Docker контейнер було розгорнуто в кластері GKE. GKE управляє Pod'ами, які містять контейнери мого застосунку.
3. Дані застосунку зберігаються в базі даних Cloud SQL for PostgreSQL.

4. Зберігання статичного контенту: HTML, CSS та JavaScript файли зберігаються в Cloud Artifact Resistry.
5. Балансування навантаження: Вхідний трафік розподіляється між Pod'ами застосунку за допомогою Cloud Load Balancing.

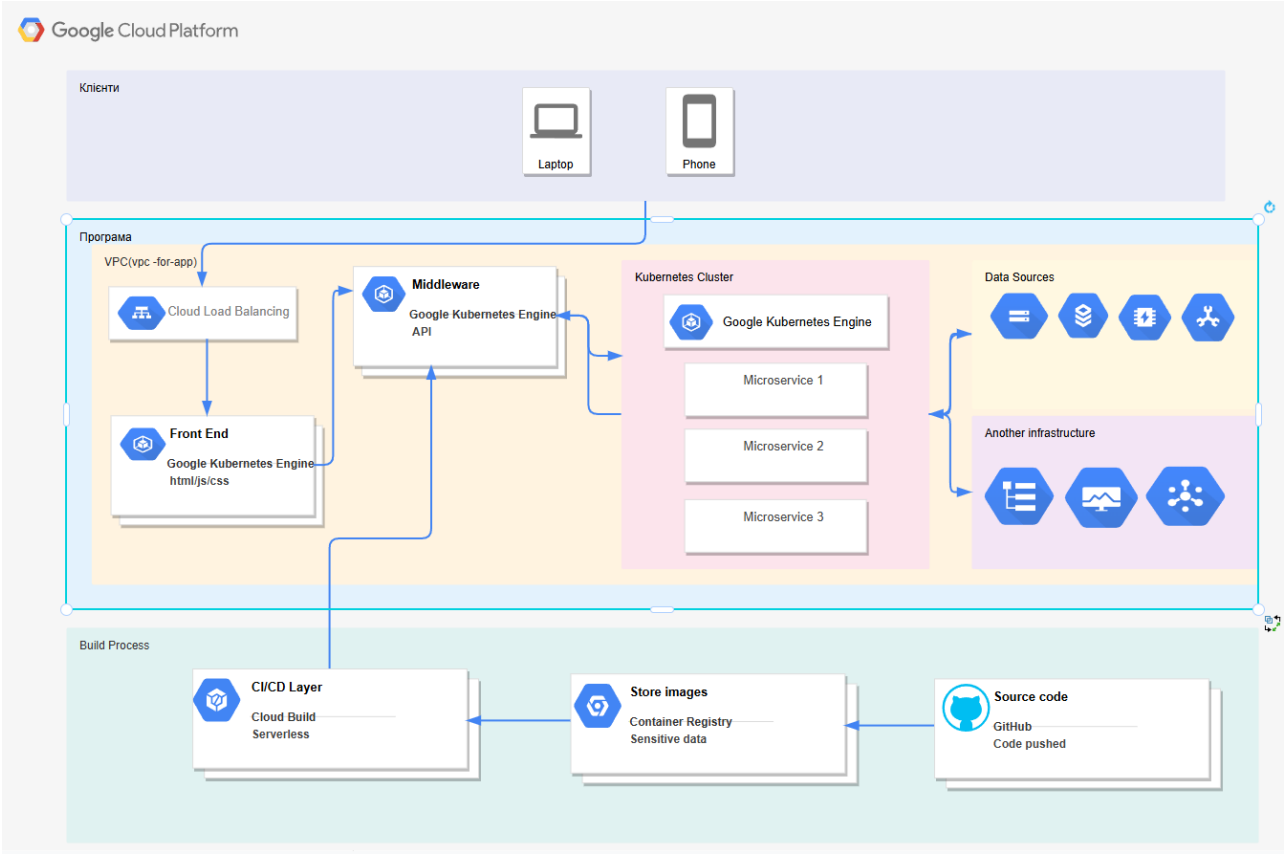


Рис 2.3 Загальна схема архітектури

На цьому рисунку зображено архітектуру мого застосунку, розгорнутого на Google Cloud Platform. Вона базується на мікросервісному підході та використовує різні сервіси GCP для забезпечення масштабованості, надійності та безпеки.

Компоненти архітектури:

1. Клієнти: Користувачі, які взаємодіють з застосунком через веб-браузер (Laptop) або мобільний застосунок (Phone).
2. Front End (Google Kubernetes Engine): Розміщений на Google Kubernetes Engine (GKE) для забезпечення масштабованості та доступності. Відповідає

за відображення користувацького інтерфейсу та обробку взаємодії з користувачем (HTML, CSS, JavaScript).

3. Middleware (Google Kubernetes Engine): Також розміщений на GKE. Реалізує бізнес-логіку застосунку та надає API (Application Programming Interface) для взаємодії з Front End та іншими мікросервісами.
4. Kubernetes Cluster (Google Kubernetes Engine): Містить мікросервіси застосунку (Microservice 1, Microservice 2, Microservice 3), які взаємодіють між собою та з іншими компонентами системи.
5. Data Sources: Різні джерела даних, які можуть використовуватися застосунком, включаючи бази даних Cloud SQL, Cloud Spanner, Cloud Datastore тощо. Також може включати інші інфраструктурні компоненти, такі як message queues (наприклад, Cloud Pub/Sub).
6. Cloud Load Balancing: Розподіляє трафік між інстансами Front End та Middleware, забезпечуючи високу доступність та ефективність.
7. CI/CD Layer (Cloud Build): Cloud Build використовується для автоматизації збірки, тестування та розгортання застосунку в Kubernetes Engine.
8. Store images (Container Registry): Container Registry використовується для зберігання образів Docker, які використовуються для розгортання застосунку в Kubernetes Engine.
9. Source code (GitHub): GitHub використовується для зберігання вихідного коду застосунку та управління версіями.

Взаємодія компонентів:

1. Клієнти надсилають запити до застосунку через HTTPS.
2. Cloud Load Balancing розподіляє трафік між інстансами Front End.
3. Front End взаємодіє з Middleware через API.
4. Middleware взаємодіє з мікросервісами в Kubernetes Cluster та з Data Sources.
5. CI/CD пайплайн автоматизує процес збірки, тестування та розгортання застосунку в Kubernetes Engine.

Переваги цієї архітектури:

1. Масштабованість та надійність: Kubernetes Engine та Cloud Load Balancing забезпечують масштабованість та високу доступність застосунку.
2. Гнучкість: Мікросервісний підхід дозволяє розробляти та розгортати компоненти застосунку незалежно один від одного.
3. Ефективність: Використання безсерверних технологій (Cloud Functions, Cloud Run) дозволяє оптимізувати витрати на інфраструктуру.
4. Безпека: GCP надає різні механізми безпеки для захисту даних та застосунків.

2.3.1 Методологія дванадцяти факторів

При проектуванні архітектури модуля, я буду керувався методологією "Дванадцять факторів" (The Twelve-Factor App), яка є набором рекомендацій для створення сучасних хмарних застосунків. Ця методологія допомагає забезпечити портативність, масштабованість та автоматизацію розгортання застосунків.

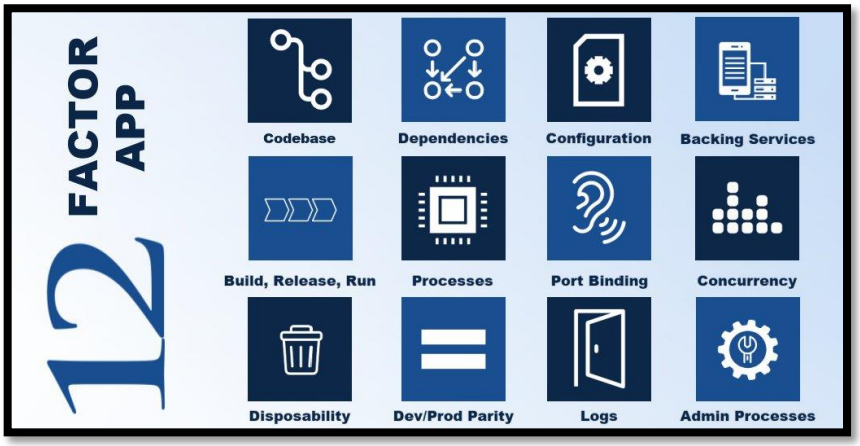


Рис 2.4 The Twelve-Factor App

Ось як я буду застосувавати принципи "Дванадцяти факторів" у своєму проекті:

1. Codebase: Код мого застосунку зберігається в системі контролю версій (GitHub) і є єдиним для всіх середовищ розгортання.
2. Dependencies: Всі залежності мого застосунку чітко визначені в файлі pom.xml (для Maven) або build.gradle (для Gradle).
3. Config: Конфігурація застосунку (наприклад, параметри підключення до бази даних) зберігається в змінних середовища, а не в коді.

4. Backing services: Сервіси, від яких залежить мій застосунок (наприклад, база даних Cloud SQL), розглядаються як приєднані ресурси і можуть бути легко замінені.
5. Build, release, run: Етапи збірки, випуску та запуску застосунку чітко розділені.
6. Processes: Застосунок працює як один або кілька безстатусних процесів.
7. Port binding: Застосунок експонує свої сервіси через порт.
8. Concurrency: Масштабування застосунку здійснюється шляхом запуску кількох інстансів процесів.
9. Disposability: Процеси застосунку можуть бути швидко запущені та зупинені.
10. Dev/prod parity: Середовища розробки, тестування та продакшену максимально наближені одне до одного.
11. Logs: Логи застосунку обробляються як потоки подій.
12. Admin processes: Адміністративні завдання виконуються як одноразові процеси.

Застосування методології "Дванадцять факторів" дозволило мені створити модуль, який є:

— портативним, бо модуль може бути легко розгорнутий на різних платформах;

— масштабованим, бо модуль може бути легко масштабований залежно від навантаження;

— надійним,бо модуль є стійким до збоїв та забезпечує високу доступність;

— у наступних підрозділах я розгляну деталі реалізації модуля, використовуючи обрані сервіси GCP та керуючись принципами "Дванадцяти факторів".

2.3.2 Методологія Continuous Integration в Google Cloud

У сучасному світі розробки програмного забезпечення, де швидкість та якість є ключовими факторами успіху, методологія Continuous Integration (CI) стала невід'ємною частиною процесу. CI передбачає часту інтеграцію змін коду в центральний репозиторій та автоматичний запуск збірки та тестування, що дозволяє виявляти та виправляти помилки на ранніх етапах, покращувати якість коду та прискорювати процес розробки.

Google Cloud Platform (GCP) надає потужний набір інструментів та сервісів для впровадження CI/CD, які легко інтегруються та забезпечують масштабованість, надійність і безпеку.

Основні принципи CI в Google Cloud:

1. Часта інтеграція: Розробники інтегрують свій код до спільного репозиторію кілька разів на день, що дозволяє уникнути конфліктів та складних зливань коду.
2. Автоматизована збірка: Процес збірки проекту повністю автоматизований за допомогою Cloud Build або інших інструментів, таких як Jenkins, налаштованих в GCP. Це включає компіляцію коду, запуск скриптів, упакування та інші необхідні кроки.
3. Автоматизоване тестування: Автоматизовані тести запускаються після кожної збірки для перевірки коректності коду та виявлення помилок на ранній стадії. Google Cloud пропонує різні сервіси для тестування, такі як Cloud Test Lab.
4. Швидкий зворотний зв'язок: Розробники отримують швидкий зворотний зв'язок про результати збірки та тестування за допомогою інструментів сповіщення та моніторингу, таких як Cloud Monitoring та Cloud Logging.

Переваги CI в Google Cloud:

1. Масштабованість та надійність: GCP надає масштабовану та надійну інфраструктуру для запуску вашого пайплайну CI/CD. Ви можете легко масштабувати ресурси залежно від потреб вашого проекту.
2. Інтеграція з іншими сервісами GCP: CI/CD легко інтегрується з іншими сервісами GCP, такими як Cloud Source Repositories, Cloud Build, Container Registry та Kubernetes Engine. Це дозволяє створити єдиний та ефективний процес розробки та розгортання.
3. Безпека: Google Cloud надає безпечне середовище для запуску вашого пайплайну CI/CD. Ви можете використовувати різні механізми безпеки, такі як IAM (Identity and Access Management) та VPC Service Controls, для захисту ваших даних та застосунків.
4. Ефективність: Google Cloud дозволяє оптимізувати витрати на CI/CD. Ви платите лише за ті ресурси, які використовуєте, і можете використовувати різні опції ціноутворення для оптимізації витрат.

Інструменти CI/CD в Google Cloud:

1. Cloud Build: Повністю керований сервіс CI/CD, який дозволяє створювати, тестувати та розгортати застосунки в Google Cloud. Cloud Build може автоматично збирати та тестувати ваш код з різних джерел, включаючи GitHub, Bitbucket та Cloud Source Repositories.
2. Cloud Source Repositories: Приватні репозиторії Git, які розміщені в Google Cloud та інтегруються з іншими сервісами GCP, такими як Cloud Build та Cloud Logging.
3. Container Registry: Приватний реєстр образів Docker, який дозволяє безпечно зберігати та управляти вашими образами Docker. Container Registry інтегрується з Kubernetes Engine та іншими сервісами GCP.

4. Kubernetes Engine: Керований сервіс Kubernetes, який дозволяє розгортати, управляти та масштабувати контейнеризовані застосунки в Google Cloud. Kubernetes Engine інтегрується з Cloud Build та іншими сервісами CI/CD.
5. Jenkins в Google Cloud: Ви можете розгорнути Jenkins в Google Cloud та використовувати його для CI/CD. Jenkins надає широкий вибір плагінів та інтеграцій з різними інструментами та сервісами.

2.4 Вибір технологій та інструментів розробки

У цьому розділі я опишу свій вибір технологій та інструментів для розробки модуля масштабованих веб-застосувань. Враховуючи вимоги до модуля, а також специфіку Google Cloud Platform, я обрав наступний стек технологій.
При розробці модуля масштабованих веб-застосувань я обрав Java як основну мову програмування. Java є популярною та зрілою мовою з великою спільнотою та безліччю бібліотек та фреймворків. Вона кросплатформена та об'єктно-орієнтована, що сприяє створенню модульного та легко підтримуваного коду. Крім того, Java добре підтримується на Google Cloud Platform.

Як основний фреймворк я обрав Spring Boot, який спрощує розробку веб-застосунків, надаючи готові компоненти та автоматичну конфігурацію. Spring Boot має вбудовану підтримку мікросервісів та добре інтегрується з Kubernetes. Широка екосистема Spring Boot дозволяє використовувати різноманітні бібліотеки та інструменти для розробки.

Для управління залежностями та збірки проекту я використав Maven. Контроль версій здійснювався за допомогою Git, що дозволило ефективно керувати кодом та співпрацювати з іншими розробниками. Для контейнеризації застосунку було обрано Docker, а для оркестрації та управління контейнерами - Kubernetes. Для налаштування CI/CD пайплайну було використано Jenkins. Такий вибір технологій та інструментів дозволив мені ефективно розробляти, тестувати та розгортати модуль масштабованих веб-застосувань на Google Cloud

Platform. Застосування Java та Spring Boot забезпечило швидку та ефективну розробку, а Docker та Kubernetes - портативність, масштабованість та відмовостійкість. Terraform спростив та прискорив процес розгортання, а Jenkins забезпечив автоматизацію збірки, тестування та розгортання застосунку. В цілому, обраний стек технологій забезпечив високу продуктивність, надійність, масштабованість та гнучкість модуля.

2.4.1 Визначення SLI та SLO

На етапі реалізації модуля масштабованих веб-застосувань я визначив індикатори рівня обслуговування (SLI) та цілі рівня обслуговування (SLO). SLI та SLO є важливими для моніторингу та забезпечення надійності мого застосунку. Вони допоможуть мені зрозуміти, наскільки добре мій застосунок працює з точки зору користувача, та вжити заходів для покращення його продуктивності та доступності.

Як SLI для свого веб-застосунку я обрав наступні метрики:

— час відгуку (latency) - час, необхідний для обробки запиту до API;

— помилки (error rate) - відсоток запитів до API, що завершилися помилкою;

— доступність (availability) - відсоток часу, протягом якого застосунок доступний для користувачів;

— пропускна здатність (throughput) - кількість запитів, оброблених застосунком за одиницю часу.

Для кожного SLI я визначив цільові значення (SLO), які вважаю прийнятними для мого застосунку:

— SLI: Час відгуку API;

— SLO: 95% запитів повинні оброблятися менше ніж за 100 мс;

— SLI: Помилки API;

— SLO: Рівень помилок не повинен перевищувати 1%;

— SLI: Доступність застосунку;

— SLO: Застосунок повинен бути доступним 99.9% часу.

Визначення SLI та SLO дозволить мені:

— стежити за здоров'ям застосунку: SLI, як термометр, показують "температуру" вашого застосунку, а SLO визначають, яка "температура" є нормальною. Ви зможете постійно відстежувати ці показники та вчасно помічати будь-які проблеми;

— знаходити слабкі місця: Якщо застосунок "хворіє", SLI допоможуть вам визначити, де саме проблема. Наприклад, якщо час завантаження сторінки перевищує SLO, ви зможете зосередити свої зусилля на оптимізації цього аспекту;

— покращувати якість обслуговування: Використовуючи SLI та SLO, ви можете ставити перед собою конкретні цілі щодо покращення продуктивності та надійності застосунку та відстежувати прогрес у досягненні цих цілей;

— задовольняти потреби користувачів: Визначаючи SLO на основі очікувань користувачів, ви можете забезпечити, що ваш застосунок відповідає їх потребам та надає високий рівень обслуговування.У наступних підрозділах я детальніше розгляну реалізацію модуля, враховуючи визначені SLI та SLO.

Але найкраще допомогти оцінити SLI та SLO можуть допомогти приклади історій користувачів:

“Як користувач поточного аккаунту, я хочу перевірити свій список збережень у будь-який час доби і у будь якому місці.”

“Як менеджер, я хочу проаналізувати дані про ефективність сховища усіх наших користувачів, щоб я міг визначити обсяг і допомогти їм покращитися.”

Виходячи з вимог прикладів, потрібно заповнити таблицю SLO та SLI:

Таблиця 2.3

Огляд очікуваних SLO і SLI

Приклад історії	SLO	SLI
Успішний логін	> 99.9% успішних спроб логіну	Кількість успішних логінів / Загальна к-ть спроб логіну
Час відповіді авторизації	< 200 мс	Час, витрачений на обробку запиту авторизації (від запиту до відповіді)
Створення/редагування /видалення нотаток	> 99.9% успішних операцій	Кількість успішних операцій з нотатками / Загальна кількість спроб
Час відповіді	< 1 секунда	Час, витрачений на обробку запиту (створення, редагування, видалення)
Збереження/видалення посилань	> 99.9% успішних операцій	Кількість успішних / Загальна к-ть спроб
Час відповіді	< 1 секунда	Час, витрачений на обробку запиту (збереження, видалення)

2.4.2 Створення мікросервіса для програми

На етапі створення мікросервіса для моєї програми на Java/HTML/CSS, я врахував обрані технології (Java, Spring Boot) та принципи "Дванадцяти факторів". Спочатку я створив новий проект Spring Boot, використовуючи Spring Initializr. Проект було структуровано відповідно до рекомендацій Spring Boot, з окремими пакетами для контролерів, сервісів та репозиторіїв. Потім, використовуючи Spring Web, я розробив REST API для мого застосунку. API містить ендпоінти для основних операцій з даними, таких як створення, читання, оновлення та видалення. Для зручності використання API було задокументовано за допомогою Swagger. Для взаємодії з базою даних Cloud SQL я використав Spring Data JPA, що спростило роботу з базою даних та дозволило зосередитися на бізнес-логіці застосунку. Нарешті, я реалізував логіку обробки запитів до API в контролерах та сервісах, приділяючи особливу увагу продуктивності та масштабованості коду.

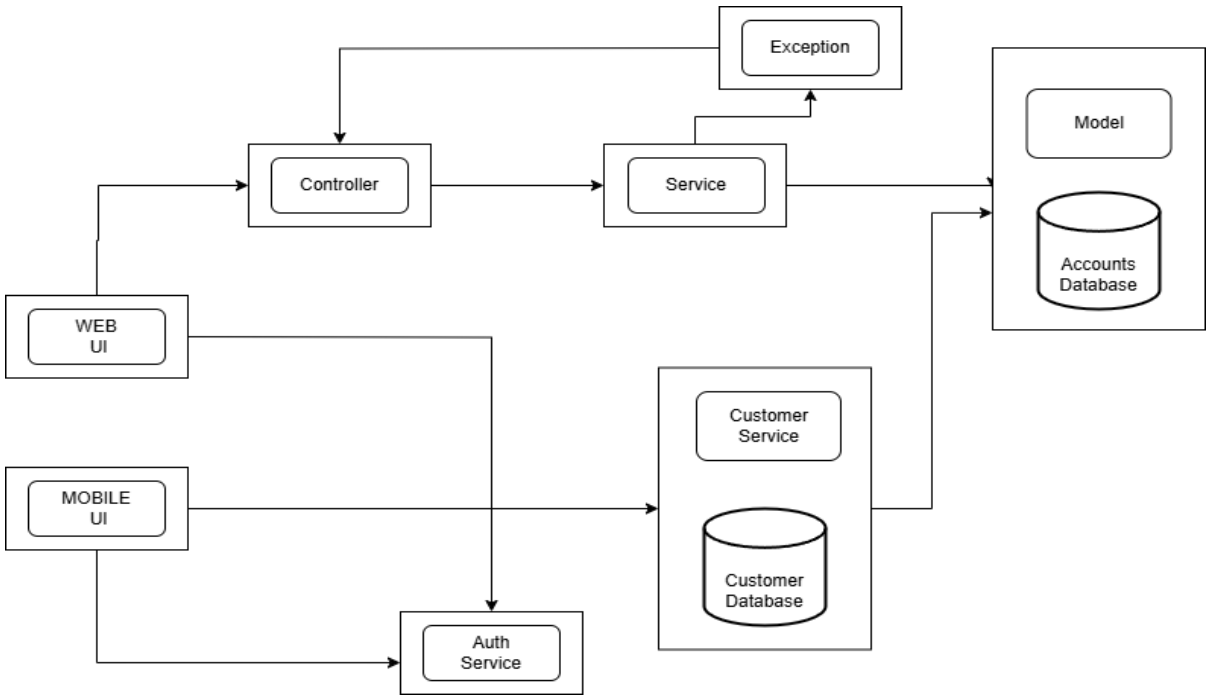


Рис 2.5 Очікувана архітектура мікросервісів

2.4.3 Проектування REST API

У цьому підрозділі я детально опишу процес проектування REST API для мого модуля масштабованих веб-застосувань. REST API є ключовим компонентом, що забезпечує взаємодію між клієнтськими застосунками (веб-інтерфейс, мобільний додаток) та серверною частиною, дозволяючи користувачам ефективно керувати своїми даними.

Оскільки код не є основною ідеєю роботи, тому деяку інформацію щодо коду буде винесено в додатки. Використовуючи Додаток Б я описав REST API. Метою цієї діяльності є отримання досвіду розробки API та врахування таких аспектів, як структура URL-адреси API, формати відповіді на запит повідомлення та керування версіями.

Таблиця 2.4

Визначення REST API додатку

Метот HTTP	Ендпоінт	Операція
GET	/home	Головна сторінка
POST	/api/users	Створити користувача
GET	/api/users/check-username/{username}	Перевірити ім'я
POST	/api/files	Завантажити файл
GET	/api/files/{id}	Отримати метадані
DELETE	/api/files/{id}	Видалити файл
POST	/api/notes	Створити нотатку
GET	/api/notes	Отримати нотатки
GET	/api/notes/{id}	Редагувати нотатку
DELETE	/api/notes/{id}	Видалити нотатку
POST	/api/credentials	Створити облікові дані
PUT	/api/credentials/{id}	Оновити облікові дані

2.4.4 Вибір сервісів Google Cloud для зберігання даних

У цьому розділі я опишу, як я вибирав сервіси Google Cloud для зберігання та обробки даних у моєму модулі масштабованих веб-застосувань.Використовув для цього блок-схему Рис. 2.2 для вибору сервісів.

Спочатку я проаналізував потреби мого застосунку щодо зберігання та обробки даних. Враховуючи, що мій застосунок є простою програмою нотаток, мені потрібні наступні можливості:

Для зберігання нотаток, які мають певну структуру (заголовок, текст, дата створення тощо).

- база даних повинна мати можливість масштабуватися для обробки зростаючого обсягу даних та кількості користувачів;
- дані повинні бути надійно захищені від втрат та доступні в будь-який час;
- база даних повинна легко інтегруватися з моїм застосунком на Java та Spring Boot;
- для зберігання структурованих даних нотаток. Cloud SQL є керованим сервісом баз даних, який забезпечує масштабованість, надійність та доступність. PostgreSQL є потужною реляційною базою даних з відкритим кодом, яка добре підходить для мого застосунку. Cloud SQL дозволяє легко масштабувати базу даних залежно від потреб застосунку і забезпечує високу доступність та автоматичне резервне копіювання даних. Spring Boot має вбудовану підтримку для роботи з PostgreSQL через Spring Data JPA;
- для зберігання файлів, прикріплених до нотаток (якщо така функціональність передбачена). Cloud Storage є масштабованим та надійним сервісом для зберігання об'єктів. Має гнучку цінову політику, що дозволяє оптимізувати витрати на зберігання даних і Google Cloud надає API та бібліотеки для роботи з Cloud Storage з Java.

Я також розглядав інші сервіси Google Cloud для зберігання даних, такі як:

1. Cloud Firestore – це NoSQL база даних для мобільних та веб-застосунків.

2. Cloud Spanner – це глобально розподілена база даних для застосунків, що потребують високої узгодженості даних.

Однак, враховуючи специфіку мого застосунку, Cloud SQL for PostgreSQL та Cloud Storage є найбільш підходящими сервісами.

2.4.5 Проектування мережевої інфраструктури та балансування навантаження

У цьому розділі я опишу, як було спроектовано мережеву інфраструктуру та налаштовано балансування навантаження для мого модуля масштабованих веб-застосувань на Google Cloud Platform (GCP), з урахуванням наданої вами діаграми архітектури.

Для створення ізольованого та безпечного мережевого середовища для мого застосунку я використав VPC. Як видно з діаграми, я створив VPC з кількома підмережами в різних зонах доступності (*europe-central2-a*, *europe-central2-b*, *europe-central2-c*) в регіоні *europe-central2*. Це забезпечує високу доступність та відмовостійкість, оскільки застосунок розподілено між кількома зонами.

Для розподілу трафіку між інстансами застосунку я використав два типи балансувальників навантаження:

— HTTP Global Load Balancer - балансувальник розподіляє трафік між різними регіонами Google Cloud. На діаграмі видно, що він розташований в регіоні europe-central та спрямовує трафік до UI в регіоні europe-central2;

— TCP Load Balancer - балансувальник розподіляє трафік між інстансами мікросервісів UserService та NoteService в регіоні europe-central2. Він забезпечує ефективний розподіл навантаження та високу доступність.

Діаграма демонструє наступну мережеву топологію:

1. Користувачі з Інтернету надсилають запити до HTTP Global Load Balancer.
2. HTTP Global Load Balancer спрямовує запити до UI в регіоні europe-central2.

3. UI взаємодіє з мікросервісами UserService та NoteService через TCP Load Balancer.
4. Мікросервіси обробляють запити та взаємодіють з базами даних.

Додаткові аспекти,які врахував при створенні мережової інфраструктури застосунку:

— Firewall: Я налаштував правила брандмауера для обмеження доступу до ресурсів в моїй VPC;
— DNS: Я використав Cloud DNS для управління DNS-записами мого застосунку;
— VPN: Я налаштував VPN-з'єднання для забезпечення безпечного доступу до ресурсів в приватних підмережах.

Проектування мережевої інфраструктури та налаштування балансування навантаження є важливими етапами в розробці масштабованих веб-застосунків. Використання сервісів Google Cloud, таких як VPC та Cloud Load Balancing.

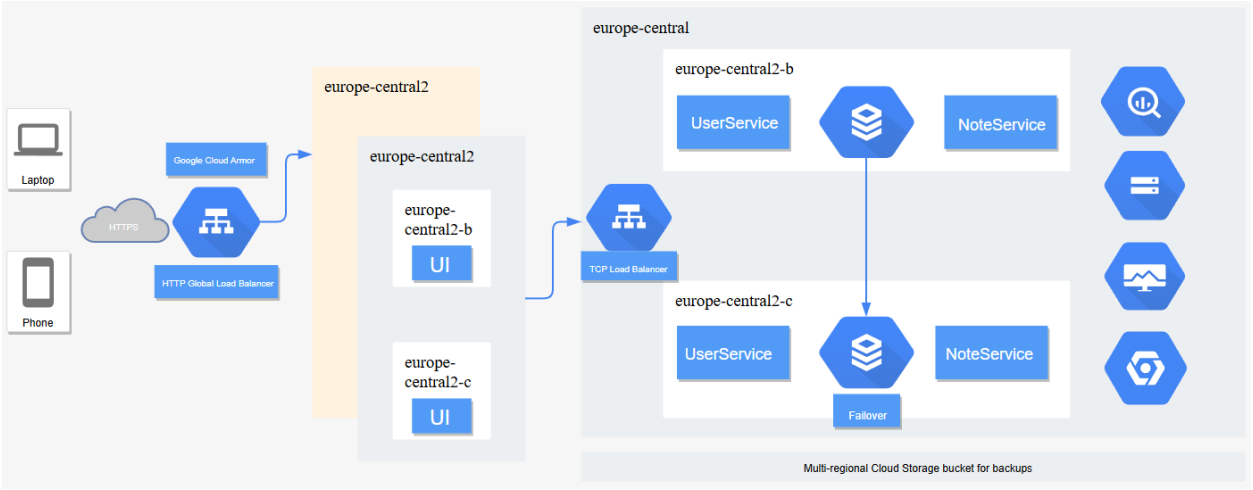


Рис 2.6 Схема мережевої інфраструктури

2.4.6 Забезпечення надійності, масштабованості та безпеки

У цьому розділі я опишу, як я забезпечив надійність, масштабованість та безпеку мого модуля веб-застосувань на Google Cloud Platform (GCP), враховуючи

надану вами схему архітектури. Перший важливий пункт – це надійність системи. Для забезпечення відмовостійкості я використав наступні механізми:

1. Розгорнув мікросервіси у кількох репліках (Pod'ах) в кластері Kubernetes. Це дозволяє системі продовжувати роботу, навіть якщо один з Pod'ів виходить з ладу.
2. Використовував Cloud SQL з опцією високої доступності для автоматичного резервування бази даних.
3. Розглянув можливість розгортання застосунку в кількох регіонах Google Cloud для захисту від збоїв в одному регіоні (як показано на схемі).
4. Налаштував health checks в Cloud Load Balancing для моніторингу стану Pod'ів та автоматичного виведення з ладу непрацюючих інстансів.
5. Реалізував обробку помилок на всіх рівнях застосунку, включаючи механізми повторних спроб та fallback-сценарії.
6. Використовував Cloud Monitoring та Cloud Logging для моніторингу стану застосунку та виявлення потенційних проблем.

Однією з переваг GKE є те, що його було створено з урахуванням масштабування і він додає ще більше можливостей для масштабування. Безумовно, GKE може досить добре працювати і без особливої конфігурації, але є великий потенціал зменшення витрат якщо розглянути глибше. Засіб автомасштабування кластерів GKE автоматично змінює кількість вузлів у певному пулі вузлів відповідно до вимог робочих навантажень. Коли попит низький, засіб автоматичного масштабування кластера зменшується до мінімального розміру, який визначається завчасно.

Horizontal Pod Autoscaler (HPA) - це потужний інструмент Kubernetes, який дозволяє автоматично масштабувати кількість pod'ів у вашому застосунку залежно від споживання ресурсів (CPU, пам'ять) або інших метрик. Це дозволяє ефективно використовувати ресурси та забезпечувати необхідну продуктивність застосунку, навіть при змінних навантаженнях.Особливості HPA:

— HPA постійно моніторить метрики, які ви визначили (наприклад, використання CPU);

— HPA порівнює поточне значення метрики з цільовим значенням, яке ви встановили;

— якщо поточне значення метрики перевищує цільове, HPA збільшує кількість pod'ів. Якщо поточне значення нижче цільового, HPA зменшує кількість pod'ів.

Переваги використання HPA:

— HPA дозволяє автоматично масштабувати застосунок, використовуючи лише необхідну кількість ресурсів;

— HPA забезпечує, що ваш застосунок може обробляти пікові навантаження, запобігаючи перевантаженню та простою;

— HPA допомагає оптимізувати витрати на хмарні ресурси, оскільки ви платите лише за фактично використані ресурси;

— HPA автоматизує процес масштабування, звільняючи вас від необхідності ручного втручання.

Вертикальне масштабування - це метод збільшення потужності веб-застосунку шляхом додавання ресурсів до існуючого сервера, на якому він працює. Це означає збільшення таких параметрів, як:

— CPU: Додавання ядер процесора або збільшення їх тактової частоти.

— RAM: Збільшення обсягу оперативної пам'яті.

— Storage: Використання швидших та більших дисків, наприклад, SSD замість HDD.

— Network: Збільшення пропускної здатності мережі.

Переваги вертикального масштабування:

1. Простота: Зазвичай легше додати ресурси до існуючого сервера, ніж налаштовувати новий.
2. Немає змін в архітектурі: Не потребує змін в коді або архітектурі застосунку.
3. Збільшення ресурсів на одному сервері може зменшити затримку в порівнянні з розподілом навантаження на кілька серверів.

Недоліки вертикального масштабування:

1. Існує фізичний ліміт на кількість ресурсів, які можна додати до одного сервера.
2. Якщо сервер виходить з ладу, весь застосунок стає недоступним.
3. Вертикальне масштабування може бути дорожчим, ніж горизонтальне, особливо при використанні хмарних сервісів, де оплата нараховується за обсяг використаних ресурсів.
4. Навіть при збільшенні ресурсів, можуть виникнути проблеми з продуктивністю, якщо застосунок не оптимізовано для використання додаткових ресурсів.

Оскільки мій застосунок має невелике навантаження, вертикальне масштабування має бути достатнім.Також вертикальне масштабування може бути хорошим варіантом на початкових етапах розвитку застосунку, коли потреби в ресурсах ще не великі.У майбутньому вертикальне масштабування може бути використано в комбінації з горизонтальним для досягнення оптимальної продуктивності та ефективності.

Для забезпечення комплексного захисту мого застосунку та даних користувачів, я реалізував багаторівневу систему безпеки, використовуючи різноманітні сервіси та механізми Google Cloud Platform, що відображено на Схемі .

Для контролю доступу до застосунку я використовую OAuth 2.0 та JWT (JSON Web Token). Це дозволяє надійно ідентифікувати користувачів та надавати їм доступ лише до дозволених ресурсів.

Контроль доступу на основі ролей.RBAC в Kubernetes та IAM в Google Cloud дозволяють мені гнучко налаштовувати права доступу до ресурсів, визначаючи ролі та прив'язуючи їх до користувачів та груп.

Для захисту даних під час передачі використовується HTTPS. Додатково, я розглядаю можливість шифрування даних в спокої за допомогою Cloud Key Management Service (KMS).

Для захисту від DDoS-атак та інших зовнішніх загроз використовується Cloud Armor, який діє як веб-застосунок брандмауер (WAF).

Для створення ізольованого мережевого середовища та обмеження доступу до ресурсів застосунку використовується Virtual Private Cloud (VPC) та налаштовані правила брандмауера.

Таким чином, завдяки комплексному підходу до безпеки, я забезпечив надійний захист застосунку та даних користувачів від різноманітних загроз.

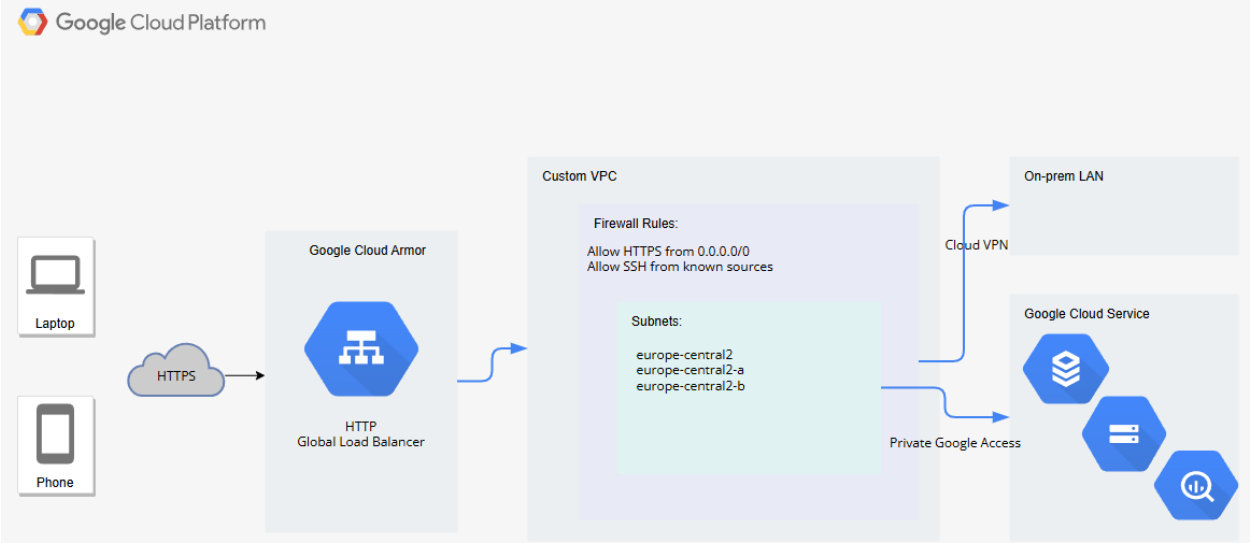


Рис 2.6 Моделювання безпечних служб Google Cloud

РОЗДІЛ 3. РОЗГОРТАННЯ ТА ЕКСПЛУАТАЦІЯ МОДУЛЯ НА ХМАРНІЙ ІНФРАСТРУКТУРІ GOOGLE CLOUD

3.1 Розгортання застосунку

У рамках моєї роботи, для створення та управління ресурсами в Google Cloud Platform, я обрав консоль gcloud, а саме Cloud Shell. Цей вибір обумовлений кількома факторами:

— gcloud надає потужний інтерфейс командного рядка, який дозволяє швидко та ефективно створювати, налаштовувати та управляти ресурсами GCP. Це особливо корисно при роботі з великою кількістю ресурсів або при автоматизації завдань;

— gcloud надає широкий спектр команд та опцій для налаштування ресурсів GCP, що дозволяє точно контролювати всі аспекти їх роботи;

— gcloud дозволяє легко автоматизувати створення та управління ресурсами за допомогою скриптів та інших інструментів автоматизації;

— gcloud інтегрується з іншими інструментами та сервісами GCP, такими як Cloud Build, Cloud Run та Kubernetes.

Процес створення ресурсів:

1. Створення проекту: Першим кроком є створення проекту в Google Cloud. Проект — це контейнер, в якому зберігаються всі ваші ресурси GCP. Я створив проект з назвою thesis2025-443618.
2. Активація Cloud Shell: Cloud Shell — це веб-браузерна оболонка командного рядка, яка надає доступ до ресурсів GCP. Cloud Shell вже містить встановлений gcloud CLI.
3. Авторизація: Після активації Cloud Shell я пройшов авторизацію в Google Cloud, щоб gcloud CLI міг отримати доступ до моїх ресурсів.

4. Вибір проекту: Я використав команду *gcloud config set project* thesis2025-443618 для вибору проекту, в якому буду створювати ресурси.

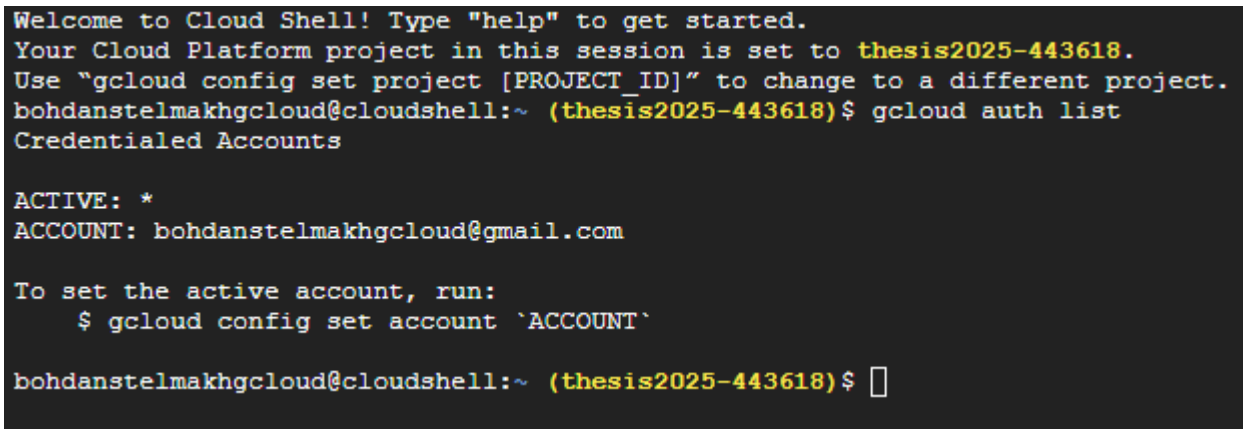


Рис. 3.1 Розгортання Cloud Shell і авторизація

Для демонстрації модуля веб застосувань у Google Cloud я використаю зразок своєї програми REST створеною Spring Boot,яку я ще створював і зберігав у 2020 році. Її я успішно вигружу зі свого репозиторію Github - https://github.com/stelmakhbohdan/super-driver-java-app.git

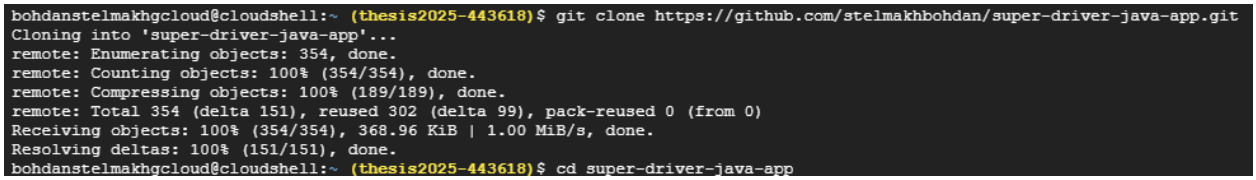


Рис. 3.2 Копіювання коду з мого Github

Перед розгортанням застосунку на Google Cloud Platform потрібно переконатися, що мій код успішно компілюється та виконується локально. Для цього запускаю застосунок локально, використовуючи наступну команду (3.1):

./mvnw -DskipTests spring-boot:run. (3.1)

Ця команда запустить мій застосунок Spring Boot, пропускаючи тести. Це дозволить мені переконатися, що в коді немає помилок компіляції або виконання, які можуть перешкодити розгортанню на GCP.Але під час тестування Maven-

проекту в середовищі розробки, я зіткнувся з наступною помилкою: “ */mvnw: Permission denied “*

Ця помилка сигналізує про те, що конве’єр не має достатніх прав для виконання файлу ./mvnw. Щоб вирішити цю проблему, необхідно змінити права доступу до файлу, надавши йому право на виконання.

Для цього в терміналі gписую наступну команду(3.2):

$$chmod +x\ mvnw\ . \tag{3.2}$$

Після цього повторно запускаю тест. Зміна прав доступу дозволить конвеєру виконати файл mvnw, і тестування мого Maven-проекту проходить успішно.

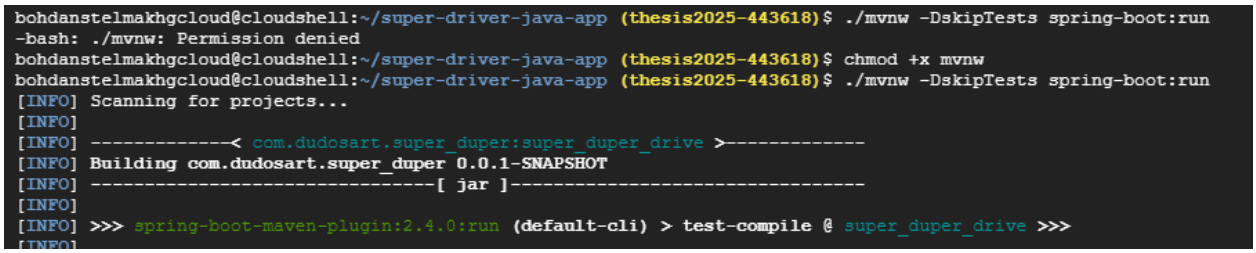


Рис. 3.3 Запуск тестування

І через декілька секунд – програму успішно протестовано.

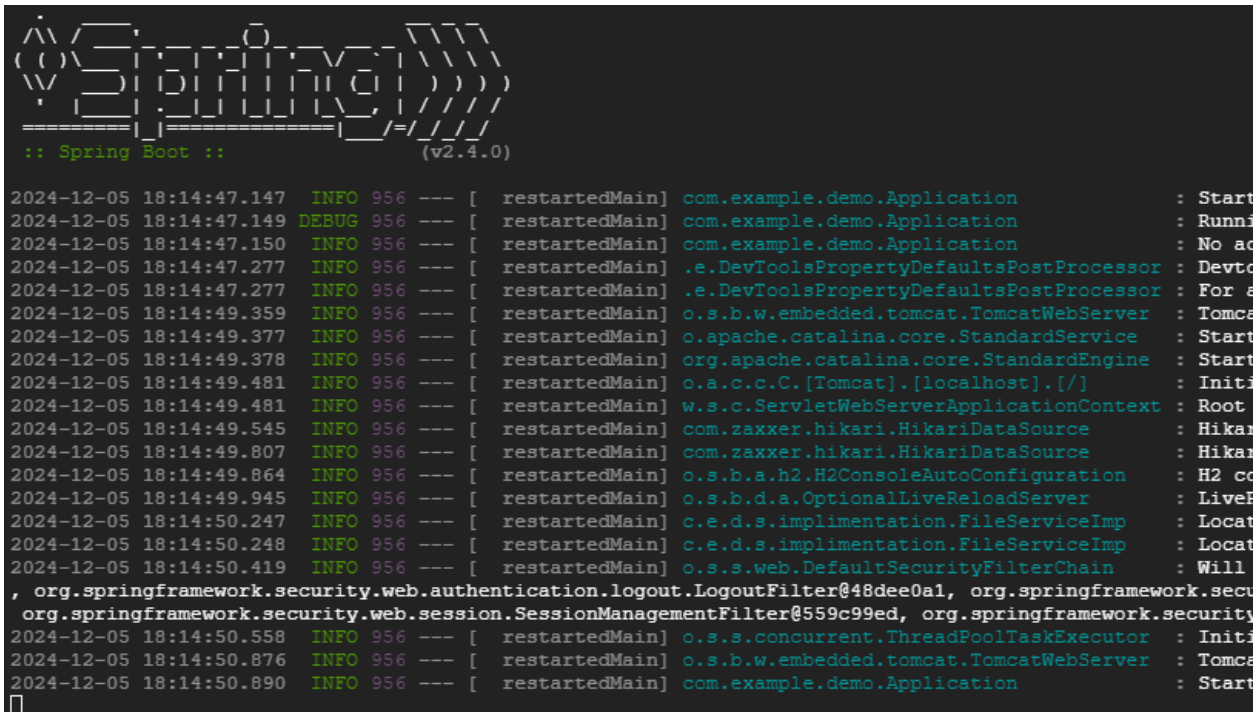


Рис. 3.4 Успішне тестування

Для успішного запуску мого Spring Boot застосунку, необхідно переконатися в наявності коректно налаштованого Maven Wrapper. Maven Wrapper - це зручний інструмент, який дозволяє запускати Maven проекти без необхідності попереднього встановлення Maven на системі. Він забезпечує консистентність збірки проекту на різних середовищах, гарантуючи, що всі розробники використовують однакову версію Maven.

Для створення Maven Wrapper виконую наступну команду(3.3):

./mvn -N io.takari:maven:wrapper . (3.3)

Ця команда згенерує файли Maven Wrapper (mvnw та mvnw.cmd) у кореневому каталозі мого проекту.

Після створення Maven Wrapper, запускаю застосунок за допомогою команди(3.4):

./mvnw -DskipTests spring-boot:run . (3.4)

DskipTests дозволяє пропустити виконання тестів під час запуску.
Maven Wrapper дуже сильпо спрощує розгортання:

- Maven Wrapper гарантує, що всі розробники використовують однакову версію Maven, що запобігає проблемам сумісності;
- він спрощує процес налаштування середовища розробки, оскільки не вимагає ручного встановлення Maven;
- Maven Wrapper автоматично завантажує та налаштовує необхідну версію Maven, якщо вона ще не встановлена.

```
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ mvn -N io.takari:maven:wrapper
[INFO] Scanning for projects...
[INFO]
[INFO] ------------< com.dudosart.super_duper:super_duper_drive >-------------
[INFO] Building com.dudosart.super_duper 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- io.takari:maven:0.7.7:wrapper (default-cli) @ super_duper_drive ---
[INFO]
[INFO] Maven Wrapper version 0.5.6 has been successfully set up for your project.
[INFO] Using Apache Maven: 3.6.3
[INFO] Repo URL in properties file: https://repo.maven.apache.org/maven2
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.255 s
[INFO] Finished at: 2024-12-05T18:17:59Z
[INFO] ------------------------------------------------------------------------
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ ./mvnw -DskipTests spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] ------------< com.dudosart.super_duper:super_duper_drive >-------------
[INFO] Building com.dudosart.super_duper 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
```

Рис. 3.5 Успішний запуск Maven Wrapper

І відразу я можу запустити завантажувальну програму Spring і перевірити кінцеву точку за допомогою опції веб-перегляду та доступу до API.

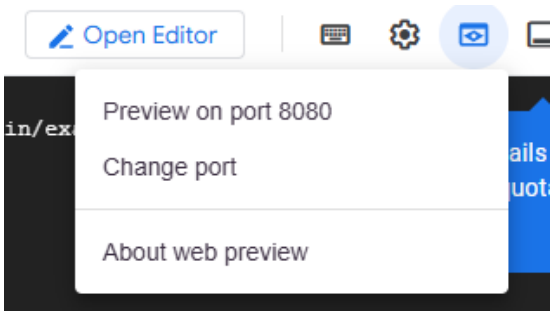


Рис. 3.6 Перевірка програми

Це відкриє нову вкладку у браузері, і я зможу отримати доступ до API, як показано нижче:

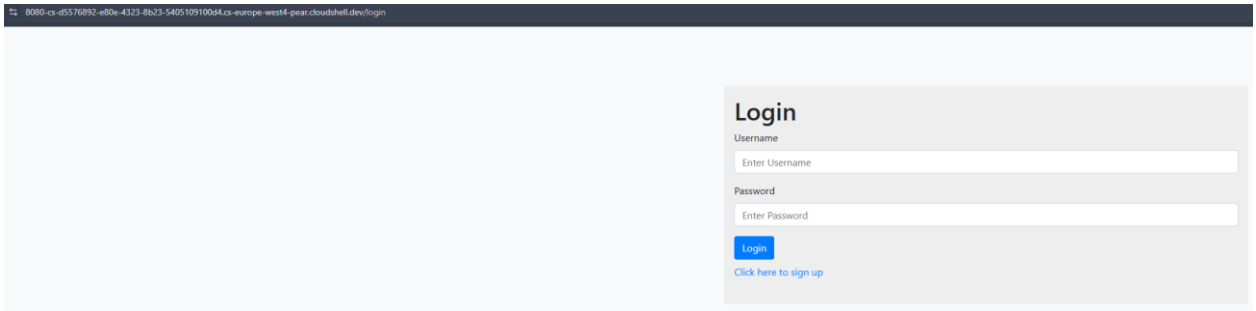


Рисунок 3.7 –Попередня перевірка програми

На цьому етапі я контейнеризував свій застосунок, тобто упакував його в образ Docker. Контейнеризація дозволяє ізолювати застосунок від операційної системи, забезпечуючи його портативність та незалежність від середовища виконання. Це важливий крок для розгортання застосунку в хмарному середовищі, такому як Google Kubernetes Engine (GKE).

Для контейнеризації я обрав плагін Jib Maven. Jib - це інструмент з відкритим кодом від Google, який спрощує процес створення образів Docker для Java-застосунків. Він інтегрується з Maven та Gradle і дозволяє створювати образи без необхідності написання Dockerfile. Вибір Jib був обумовлений його перевагами:

— Jib автоматично визначає залежності застосунку та створює оптимізовані образи;
— Jib створює образи швидше, ніж традиційний процес збірки Docker;
— Jib інтегрується з Google Container Registry (GCR) та підтримує найкращі практики безпеки.

Як альтернативу Jib, можна було б створити Dockerfile та зібрати образ Docker за допомогою команди docker build. Однак, Jib забезпечує більш простий та ефективний спосіб контейнеризації Java-застосунків.

Процес контейнеризації:

1. Першим кроком я зібрав свій застосунок за допомогою команди(3.5):

./mvnw -DskipTests package . (3.5)

2. Jib автоматично створює образ Docker під час збірки застосунку. Jib визначає залежності застосунку, копіює необхідні файли та налаштовує середовище виконання в контейнері.
3. Після створення образу я відправив його в Google Container Registry (GCR). GCR - це приватний реєстр образів Docker в Google Cloud.

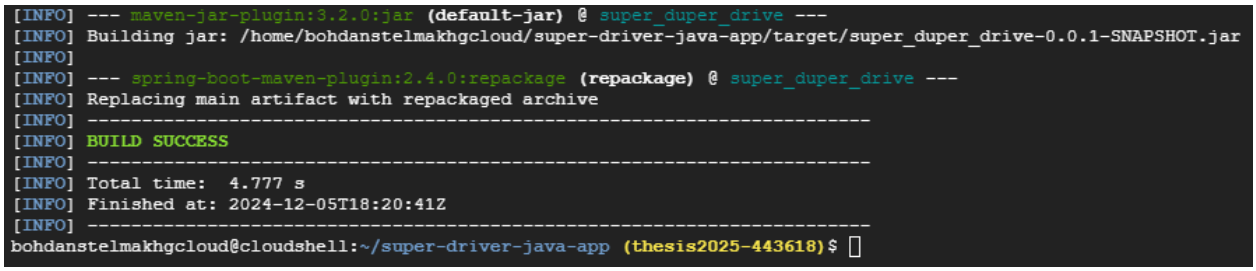


Рис. 3.8 Створення докеру

Далі потрібно ввімкнути реєстр контейнерів google (gcr), куди буде вставлено зображення докера. Образ, надісланий до реєстру, буде використано під час створення розгортання в GKE. Щоб увімкнути реєстр контейнерів, виконую таку команду(3.6):

gcloud services enable containerregistry.googleapis.com . (3.6)

Після успішної збірки застосунку я перейшов до створення образу Docker та його відправки в Google Container Registry (GCR). Для цього я використав наступні кроки:

Для початку я експортував ідентифікатор мого проекту GCP в змінну середовища GOOGLE_CLOUD_PROJECT. Це дозволило мені використовувати цю змінну в подальших командах (3.7):

export GOOGLE_CLOUD_PROJECT=`gcloud config list --format="value(core.project)"` . (3.7)

Для створення образу Docker я використав плагін Jib Maven. За допомогою наступної команди я запустив Jib та вказав тег образу, який включає ідентифікатор мого проекту GCP та версію образу (v1).

./mvnw -DskipTests com.google.cloud.tools:jib-maven-plugin:build \ -Dimage=gcr.io/$GOOGLE_CLOUD_PROJECT/super-driver-java-app:v1 . (3.8)

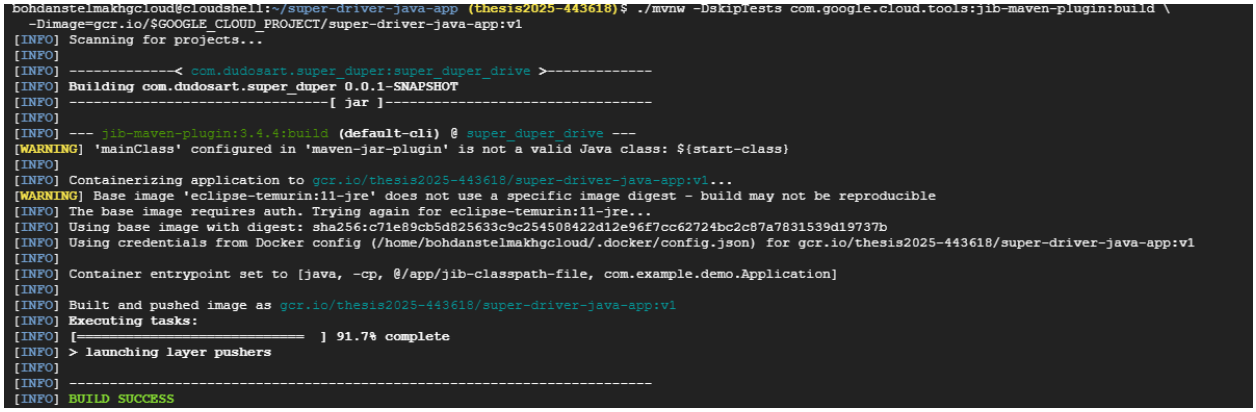


Рис. 3.9 Імпорт в Google Container Registry

Перевіряю свій образ докера, створений у реєстрі контейнерів через gcloud командою(3.9):

gcloud container images list --repository=gcr.io/$GOOGLE_CLOUD_PROJECT . (3.9)

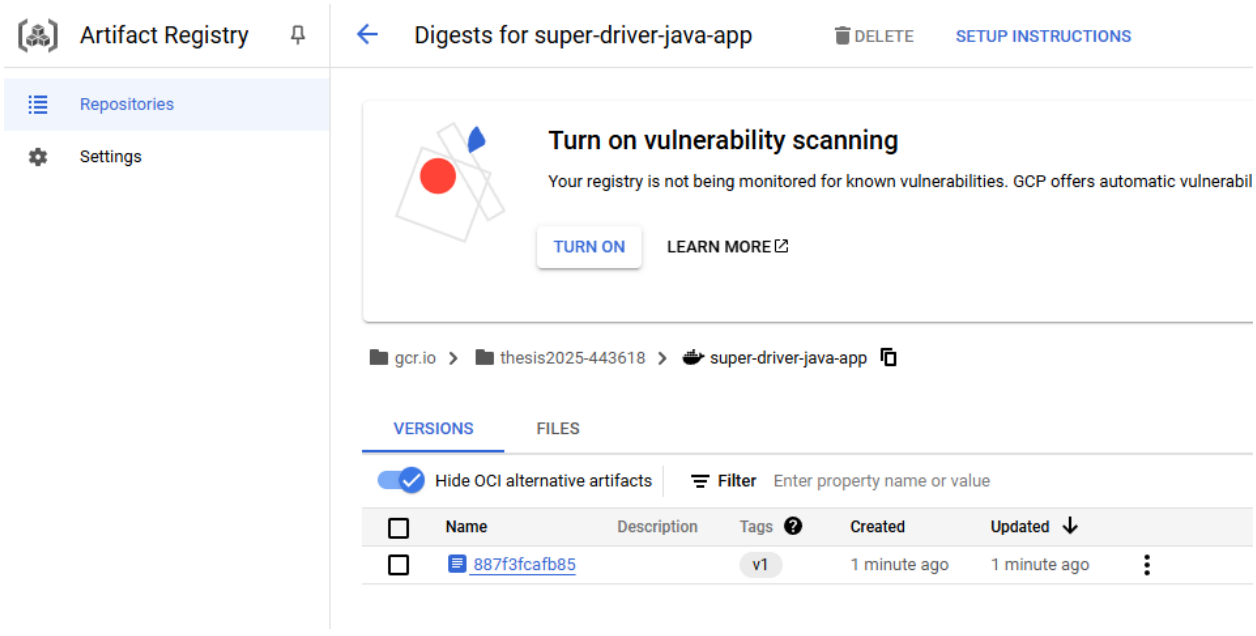


Рис. 3.10 Наявність Docker в Google Container Registry

3.1.1 Створення безпечної мережі

Перед тим як розгорнути програму в GKE потрібно створити окрему мережу VPC яка показана на Схема 3.На діаграмі показано Cloud VPN, який з'єднує вашу локальну мережу (On-prem LAN) з вашою Virtual Private Cloud (VPC) в Google Cloud. Це означає, що потрібно налаштувати VPN-з'єднання для доступу до ресурсів у вашому кластері Kubernetes з локальної мережі.
Тому перехожу до створення своєї VPC:

```
bohdanstelmakhgcloud@cloudshell:~ (thesis2025-443618)$ gcloud compute networks create vpc-for-app \
    --subnet-mode custom
Created [https://www.googleapis.com/compute/v1/projects/thesis2025-443618/global/networks/vpc-for-app].
NAME: vpc-for-app
SUBNET_MODE: CUSTOM
BGP_ROUTING_MODE: REGIONAL
IPV4_RANGE:
GATEWAY_IPV4:
```

Рис. 3.10 Створення VPC

Далі створюю дві підмережі в рамках VPC (Virtual Private Cloud) з назвою vpc-for-app:

— europe-central2-a з діапазоном IP-адрес 10.128.0.0/20;

— europe-central2-b з діапазоном IP-адрес 10.128.16.0/20.

Обидві підмережі знаходяться в регіоні europe-central2. Підмережі дозволяють розділити VPC на логічні сегменти та контролювати доступ до ресурсів в кожному сегменті.

```
bohdanstelmakhgcloud@cloudshell:~ (thesis2025-443618)$ gcloud compute networks subnets create europe-central2-a \
    --network vpc-for-app \
    --region europe-central2 \
    --range 10.128.0.0/20
Created [https://www.googleapis.com/compute/v1/projects/thesis2025-443618/regions/europe-central2/subnetworks/europe-central2-a].
NAME: europe-central2-a
REGION: europe-central2
NETWORK: vpc-for-app
RANGE: 10.128.0.0/20
STACK_TYPE: IPV4_ONLY
IPV6_ACCESS_TYPE:
INTERNAL_IPV6_PREFIX:
EXTERNAL_IPV6_PREFIX:
bohdanstelmakhgcloud@cloudshell:~ (thesis2025-443618)$ gcloud compute networks subnets create europe-central2-b \
    --network vpc-for-app \
    --region europe-central2 \
    --range 10.128.16.0/20
Created [https://www.googleapis.com/compute/v1/projects/thesis2025-443618/regions/europe-central2/subnetworks/europe-central2-b].
NAME: europe-central2-b
REGION: europe-central2
NETWORK: vpc-for-app
RANGE: 10.128.16.0/20
STACK_TYPE: IPV4_ONLY
IPV6_ACCESS_TYPE:
INTERNAL_IPV6_PREFIX:
EXTERNAL_IPV6_PREFIX:
```

Рис. 3.11 Створення підмереж

Після створюються два правила брандмауера:

— allow-https: дозволяє вхідний трафік на порт 443 (HTTPS) з будь-якої IP-адреси (0.0.0.0/0);

— allow-ssh-from-known-sources: дозволяє вхідний трафік на порт 6000 (SSH) з певних IP-адрес, які, ймовірно, вказуються в параметрі --source-ranges.

Правила брандмауера дозволяють контролювати вхідний та вихідний трафік до VPC, забезпечуючи безпеку ресурсів.

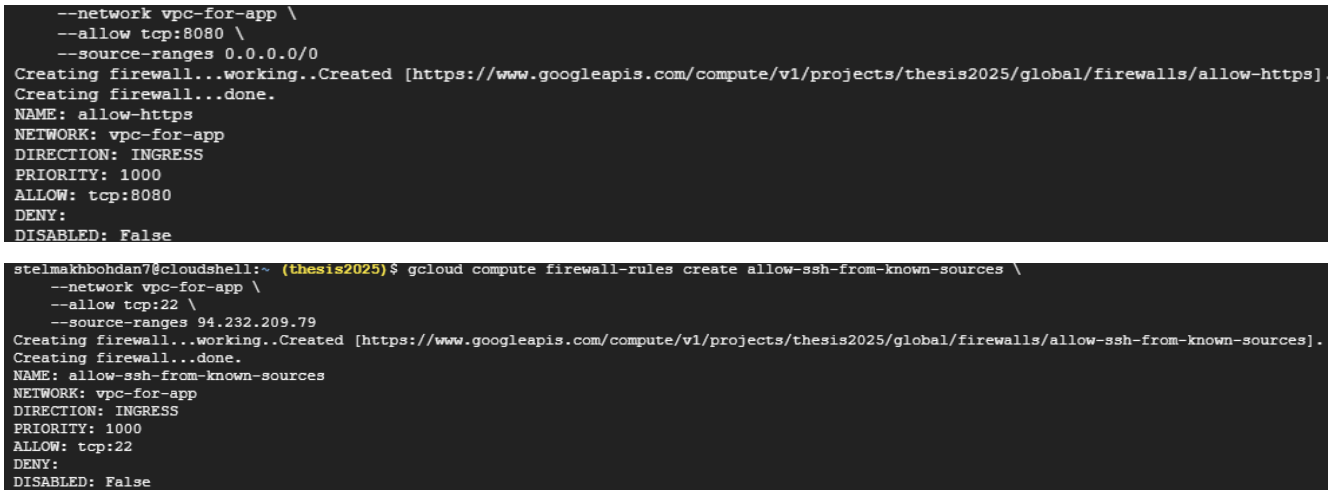


Рис. 3.12 Створення правил брандмауера

Перехожу до Cloud VPN, який дозволяє з'єднати VPC з локальною мережею (On-prem LAN):

— спочатку створюється шлюз Cloud VPN з назвою my-vpn-gateway в регіоні europe-central2;

— потім створюється VPN-тунель, який з'єднує цей шлюз з локальною мережею.

Cloud VPN забезпечує безпечне з'єднання між вашою хмарною інфраструктурою та вашою локальною мережею.

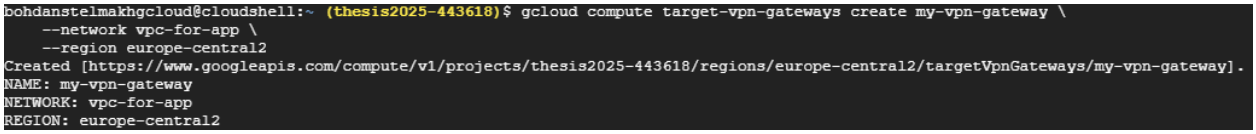


Рис. 3.13 Створення Cloud VPN

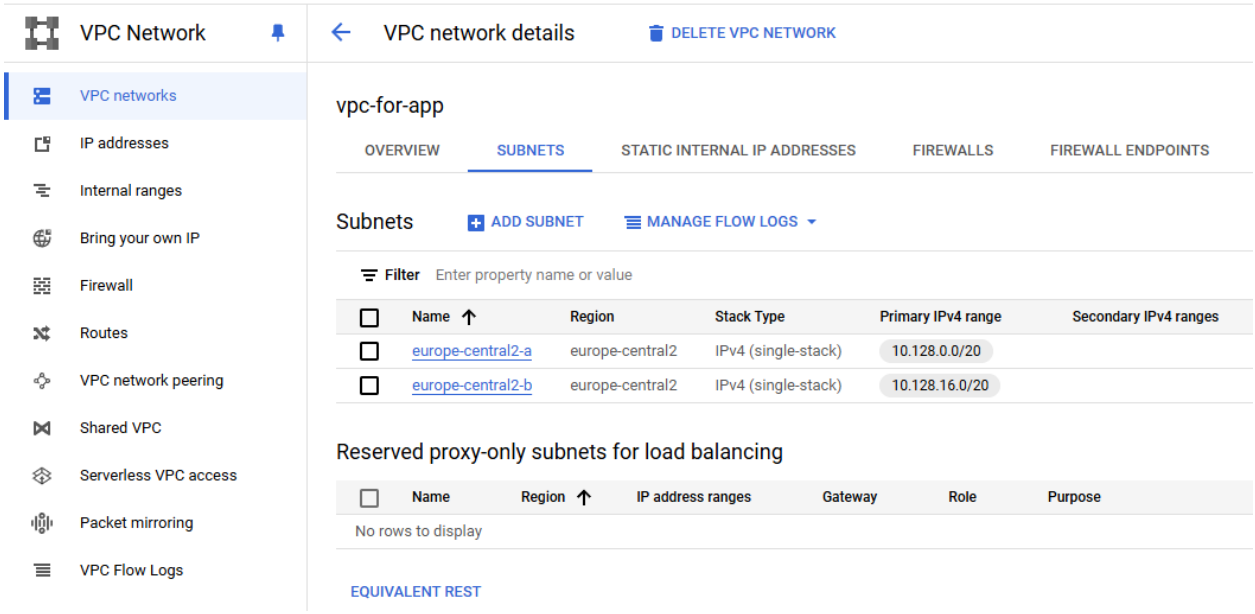


Рис. 3.14 Загальний вигляд VPC з консолі Google Cloud

Для захисту мого застосунку від несанкціонованого доступу та шкідливого трафіку я налаштував політику безпеки Cloud Armor. Політика має наступні характеристики:

Правило 1 - це правило з високим пріоритетом (999), яке забороняє трафік з двох конкретних IP-адрес (192.168.1.1 та 192.168.1.2). Я додав опис до цього правила: "Block traffic from specific IP addresses".

Правило 2 - це правило за замовчуванням з низьким пріоритетом (2,147,483,647), яке дозволяє весь трафік.

Оскільки правило 1 має вищий пріоритет, воно перевизначає правило за замовчуванням. Таким чином, Cloud Armor блокує трафік з двох IP-адрес, а весь інший трафік дозволяє.

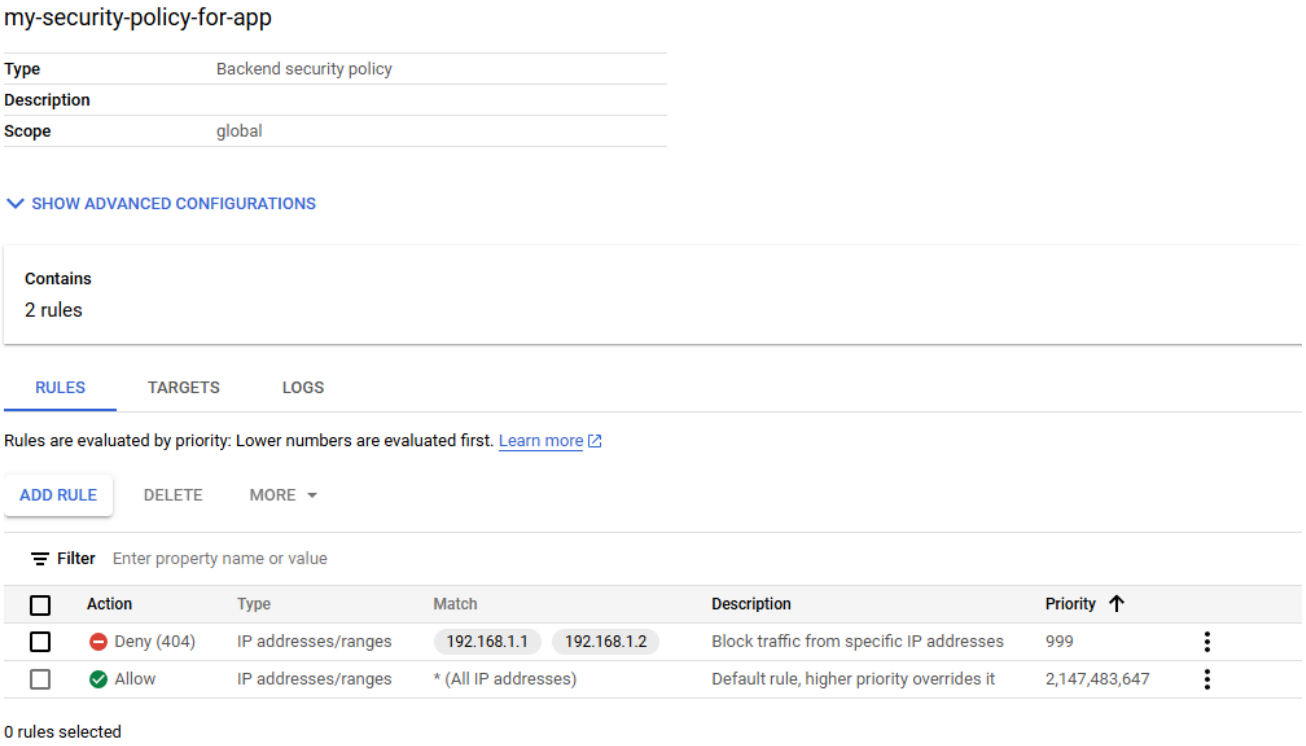


Рис. 3.15 Створення Cloud Armour

Ця конфігурація демонструє, як Cloud Armor може бути використаний для створення "чорного списку" IP-адрес, з яких заборонено доступ до застосунку. Це дозволяє захистити застосунок від небажаного трафіку та потенційних DDoS-атак.

3.1.2 Створення кластера Kubernetes в GKE

Для розгортання мого контейнеризованого застосунку мені знадобився кластер Kubernetes, керований сервісом Google Kubernetes Engine (GKE).

Перш ніж створювати кластер, я переконався, що необхідні служби Google Cloud API увімкнено. Це можна зробити за допомогою команди gcloud(3.10):

gcloud services enable compute.googleapis.com container.googleapis.com . (3.10)

Для створення кластера Kubernetes я використав наступну команду gcloud (3.11):

*gcloud container clusters create super-driver-java-app *
*--num-nodes 3 *
*--machine-type n1-standard-1 *
*--zone europe-central2-a *
*--enable-autoscaling *
*--min-nodes 1 *
*--max-nodes 10 *
*--enable-ip-alias *
*--enable-network-policy *
*--enable-shielded-nodes *
*--release-channel regular *
*--network vpc-for-app *
--subnetwork subnet-for-app , (3.11)

Пояснення параметрів:

- num-nodes 3: Створити кластер з трьома вузлами за замовчуванням;
- machine-type n1-standard-1: Використовувати віртуальні машини типу n1-standard-1 для вузлів кластера;
- zone europe-central2-a: Створити кластер в зоні доступності europe-central2-a;
- enable-autoscaling: Ввімкнути автоматичне масштабування кластера;
- min-nodes 1: Мінімальна кількість вузлів в кластері – 1;
- max-nodes 10: Максимальна кількість вузлів в кластері – 10;
- enable-ip-alias: Ввімкнути IP-псевдоніми для Pod'ів, що покращує мережеву продуктивність;
- enable-network-policy: Ввімкнути мережеві політики Kubernetes для контролю трафіку між Pod'ами;
- enable-shielded-nodes: Ввімкнути захищені вузли для підвищення безпеки;

— release-channel regular: Використовувати стабільний канал випуску Kubernetes;

— network vpc-for-app: вказує назву VPC, яку потрібно використовувати для кластера;

— subnetwork subnet-for-app: вказує назву підмережі в VPC, яку потрібно використовувати для кластера.

Рис. 3.16 Створення кластера GKE

Після створення кластера Kubernetes в GKE, я перейшов до розгортання мого контейнеризованого застосунку. Для цього я використав команду kubectl(3.12) create deployment, яка дозволяє створити Deployment - об'єкт Kubernetes, що описує бажаний стан застосунку.

$$kubectl\ create\ deployment\ app\text{-}demo\ \backslash \quad (3.12)$$
$$--image=gcr.io/\$GOOGLE\_CLOUD\_PROJECT/super\text{-}driver\text{-}java\text{-}app:v1\ ,$$

Пояснення параметрів:

— app-demo: Назва Deployment;

— image=gcr.io/$GOOGLE_CLOUD_PROJECT/super-driver-java-app:v1: Шлях до образу Docker в Google Container Registry (GCR). Змінна $GOOGLE_CLOUD_PROJECT містить ідентифікатор мого проекту GCP.

У результаті виконання цієї команди було створено Deployment з назвою app-demo. Kubernetes автоматично завантажив образ Docker з GCR та запустив мій застосунок в Pod'і.

```
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl create deployment app-demo
deployment.apps/app-demo created
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
app-demo   1/1     1            1           37s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
app-demo   1/1     1            1           53s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl describe deployments
Name:                   app-demo
Namespace:              default
CreationTimestamp:      Thu, 05 Dec 2024 19:02:50 +0000
Labels:                 app=app-demo
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=app-demo
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=app-demo
```

Рис. 3.17 Запуск застосунка у Pod’і

Для подальшого і більш детального налаштування розгортання можна використовувати YAML-файл конфігурації Deployment. В YAML-файлі можна вказати такі параметри, як:

— кількість реплік Pod'ів;
— ресурси, які потрібні застосунку (CPU, пам'ять);
— змінні середовища;
— порти, на яких прослуховує застосунок;
— Health checks для моніторингу стану застосунку.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-demo
  template:
    metadata:
      labels:
        app: app-demo
    spec:
      containers:
      - name: my-app
        image: gcr.io/$GOOGLE_CLOUD_PROJECT/super-driver-java-app:v1
        ports:
        - containerPort: 8080
```

Рис. 3.18 Конфігурація YAML файлу

Розгортання застосунку в GKE за допомогою команди *kubectl create deployment* є простим та ефективним способом запуску контейнеризованих застосунків в хмарному середовищі. Kubernetes автоматично управляє Pod'ами, забезпечуючи масштабованість, надійність та доступність застосунку.

Після розгортання застосунку в кластері Kubernetes, мені потрібно було зробити його доступним для зовнішніх користувачів. Оскільки Pod'и доступні лише через внутрішні IP-адреси кластера, я використав Service та Load Balancer для експонування застосунку.

Спочатку я створив Service типу LoadBalancer, який надає доступ до мого застосунку через стабільну IP-адресу. Для цього я використав наступну команду kubectl(3.13):

kubectl create service loadbalancer app-demo --tcp=8080:8080 , (3.13)

Пояснення параметрів:

— rest-demo: Назва Service;

— tcp=8080:8080: Вказує, що Service прослуховує порт 8080 і перенаправляє трафік на порт 8080 контейнера.

Створення Service типу LoadBalancer автоматично запускає процес створення HTTP(S) Load Balancer в Google Cloud. Load Balancer розподіляє вхідний трафік між Pod'ами, на яких працює мій застосунок, забезпечуючи високу доступність та масштабованість. Після створення Load Balancer я отримав зовнішню IP-адресу, за допомогою якої можна отримати доступ до мого застосунку з Інтернету. Використання Service типу LoadBalancer в Kubernetes дозволяє легко експонувати застосунки для зовнішніх користувачів. Load Balancer в Google Cloud забезпечує високу доступність, масштабованість та надійність застосунку.

```
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl create service loadbalancer app-demo --tcp=8080:8080
service/app-demo created
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get services
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
app-demo     LoadBalancer   34.118.231.225   <pending>     8080:31667/TCP   7s
kubernetes   ClusterIP      34.118.224.1     <none>        443/TCP          15m
rest-demo    LoadBalancer   34.118.227.58    34.118.9.37   8080:31250/TCP   4m10s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get services
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)          AGE
app-demo     LoadBalancer   34.118.231.225   34.118.44.77   8080:31667/TCP   72s
kubernetes   ClusterIP      34.118.224.1     <none>         443/TCP          16m
rest-demo    LoadBalancer   34.118.227.58    34.118.9.37    8080:31250/TCP   5m15s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$
```

Рис. 3.19 Створення Public IP для доступа до програми

Отримавши зовнішній IP 34.118.44.77 я спробую зробити підключення через інший пристрій для тесту доступності програми:

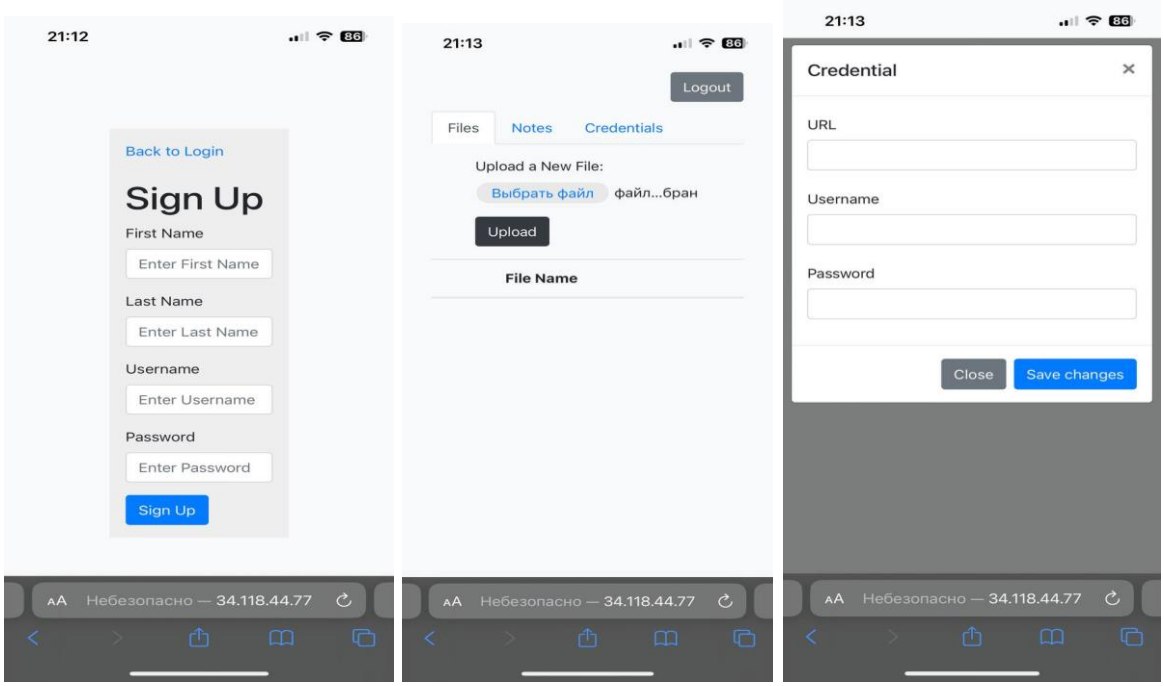


Рис. 3.20 Демонстрація застосунку в дії

3.2 Демонстрація масштабування застосунку в дії

Для демонстрації масштабованості мого застосунку, розгорнутого в кластері Kubernetes на Google Cloud Platform, я використав Cloud Monitoring для візуалізації метрик продуктивності до і під час навантажувального тестування.

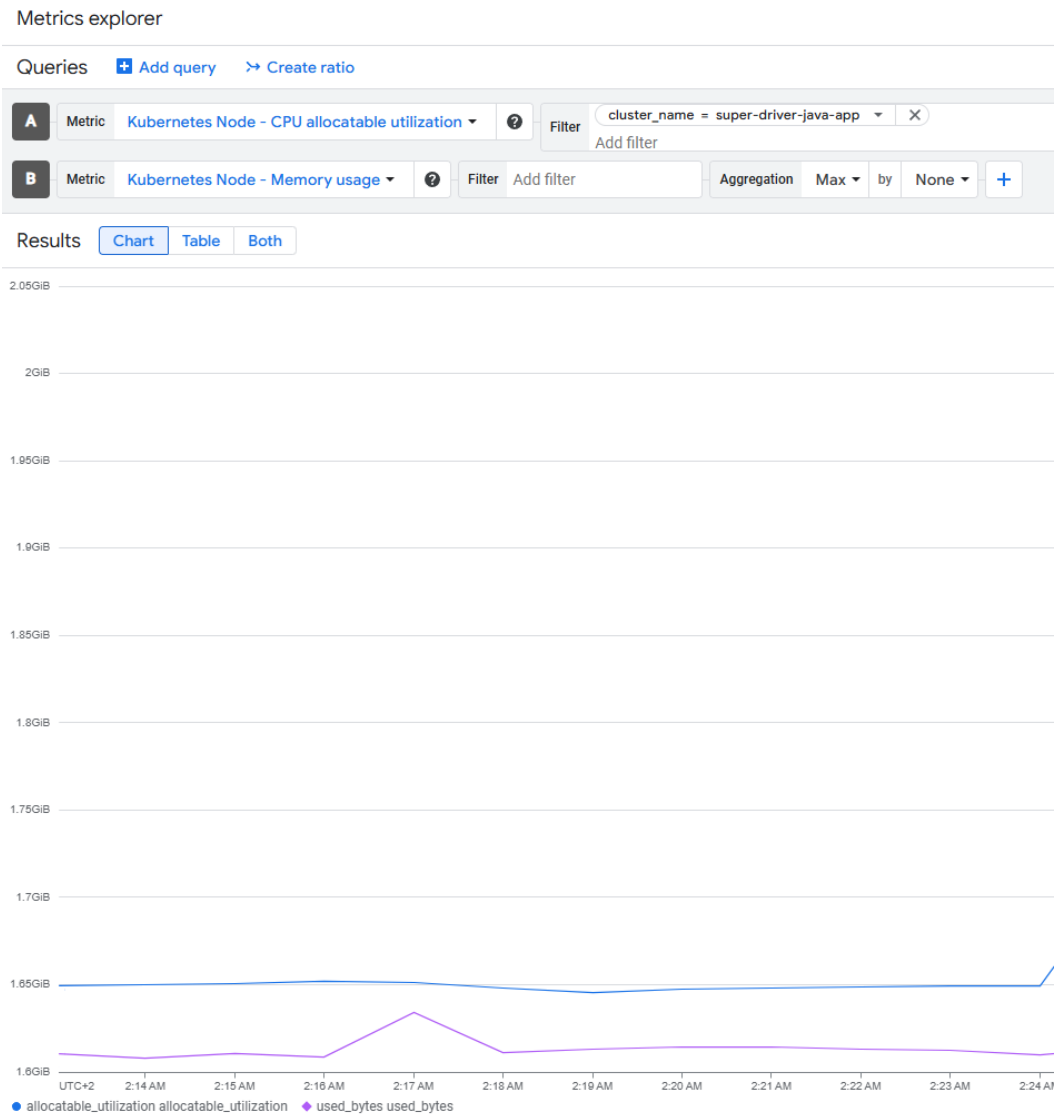


Рис. 3.21 Моніторинг до навантаження

На Рисунку 3.21 зображено два графіки в Metrics Explorer в Cloud Monitoring, які показують використання ресурсів кластера Kubernetes super-driver-java-app. Верхній графік (синій) :

1. Метрика: Kubernetes Node - CPU allocatable utilization - Показує відсоток використання доступних ресурсів CPU на вузлах вашого кластера.

2. Фільтр: cluster_name = super-driver-java-app - Фільтрує метрики за назвою вашого кластера.
3. Агрегація: Max by None - Показує максимальне значення метрики за вибраний період часу.

Верхній графік показує, що використання CPU на вузлах вашого кластера є відносно низьким та стабільним протягом останньої години.

Нижній графік (фіолетовий):

1. Метрика: Kubernetes Node - Memory usage - Показує обсяг використаної пам'яті на вузлах вашого кластера.
2. Фільтр: cluster_name = super-driver-java-app - Фільтрує метрики за назвою вашого кластера.
3. Агрегація: Max by None - Показує максимальне значення метрики за вибраний період часу.

Нижній графік показує, що використання пам'яті також є стабільним, але дещо вищим, ніж використання CPU.

Перед тим як перейти до тестування навантаження перевіряю наявність наявності вертикального і горизонтального масштабування:

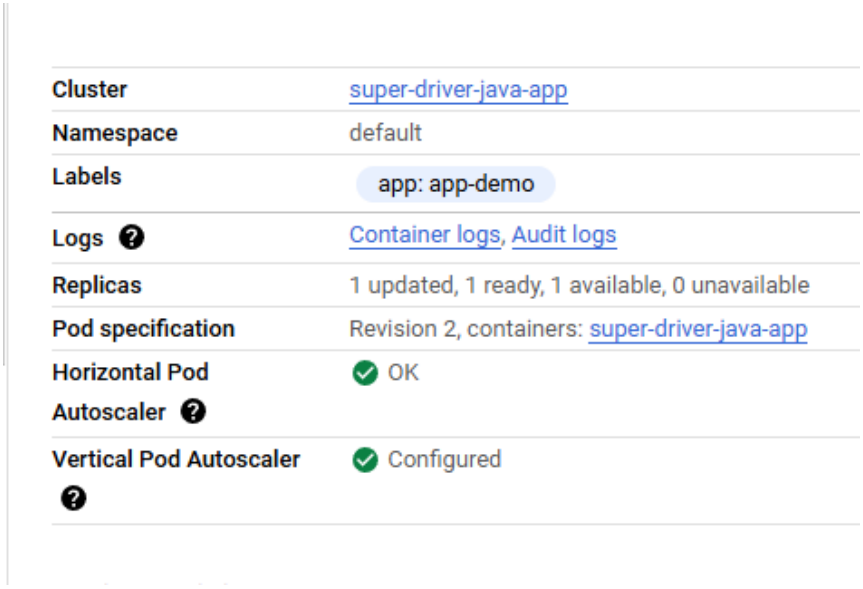


Рис 3.22 Демонстрація наявності HPA і VPA

Для найкращої демонстрації масштабованості мого застосунку в Google Kubernetes Engine (GKE) я провів навантажувальне тестування. Використовуючи

інструмент ab (Apache Benchmark), я згенерував велику кількість запитів до мого застосунку та спостерігав, як GKE автоматично масштабує кількість Pod'ів залежно від навантаження.

Запустив ab з параметрами, які визначають кількість запитів та кількість одночасних з'єднань. Приклад команди (3.14) :

ab -n 100 -c 10 http://34.118.95.49:8080/home . (3.14)

```
C:\Users\stelm>ab -n 100 -c 10 http://34.118.95.49:8080/home
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 34.118.95.49 (be patient).....done


Server Software:
Server Hostname:        34.118.95.49
Server Port:            8080

Document Path:          /home
Document Length:        0 bytes

Concurrency Level:      10
Time taken for tests:   1.682 seconds
Complete requests:      100
Failed requests:        0
Non-2xx responses:      100
Total transferred:      39100 bytes
HTML transferred:       0 bytes
Requests per second:    59.47 [#/sec] (mean)
Time per request:       168.159 [ms] (mean)
Time per request:       16.816 [ms] (mean, across all concurrent requests)
Transfer rate:          22.71 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       10   16   3.4     16      20
Processing:    20  139  25.6    145     180
Waiting:       19   99  37.2    100     152
Total:         40  155  25.7    160     198

Percentage of the requests served within a certain time (ms)
  50%    160
  66%    162
  75%    163
  80%    165
  90%    167
  95%    170
  98%    170
  99%    198
 100%    198 (longest request)

C:\Users\stelm>
```

Рис 3.23 Тестування навантаження за допомогою ab

Параметри тестування:

— кількість запитів: 100;

— кількість одночасних з'єднань: 10.

Результати тестування:

— час виконання: 1.682 секунди;

— успішні запити: 100;

— невдалі запити: 0;

— запити за секунду: 59.47;

— середній час відповіді: 168.159 мс.

Аналіз результатів:

— застосунок демонструє прийнятну продуктивність, обробляючи близько 59 запитів за секунду;

— під час тестування не було виявлено жодних невдалих запитів, що свідчить про стабільну роботу застосунку.

Для оцінки масштабованості потрібно провести додаткові тести з більшим навантаженням та проаналізувати, як змінюються показники продуктивності.

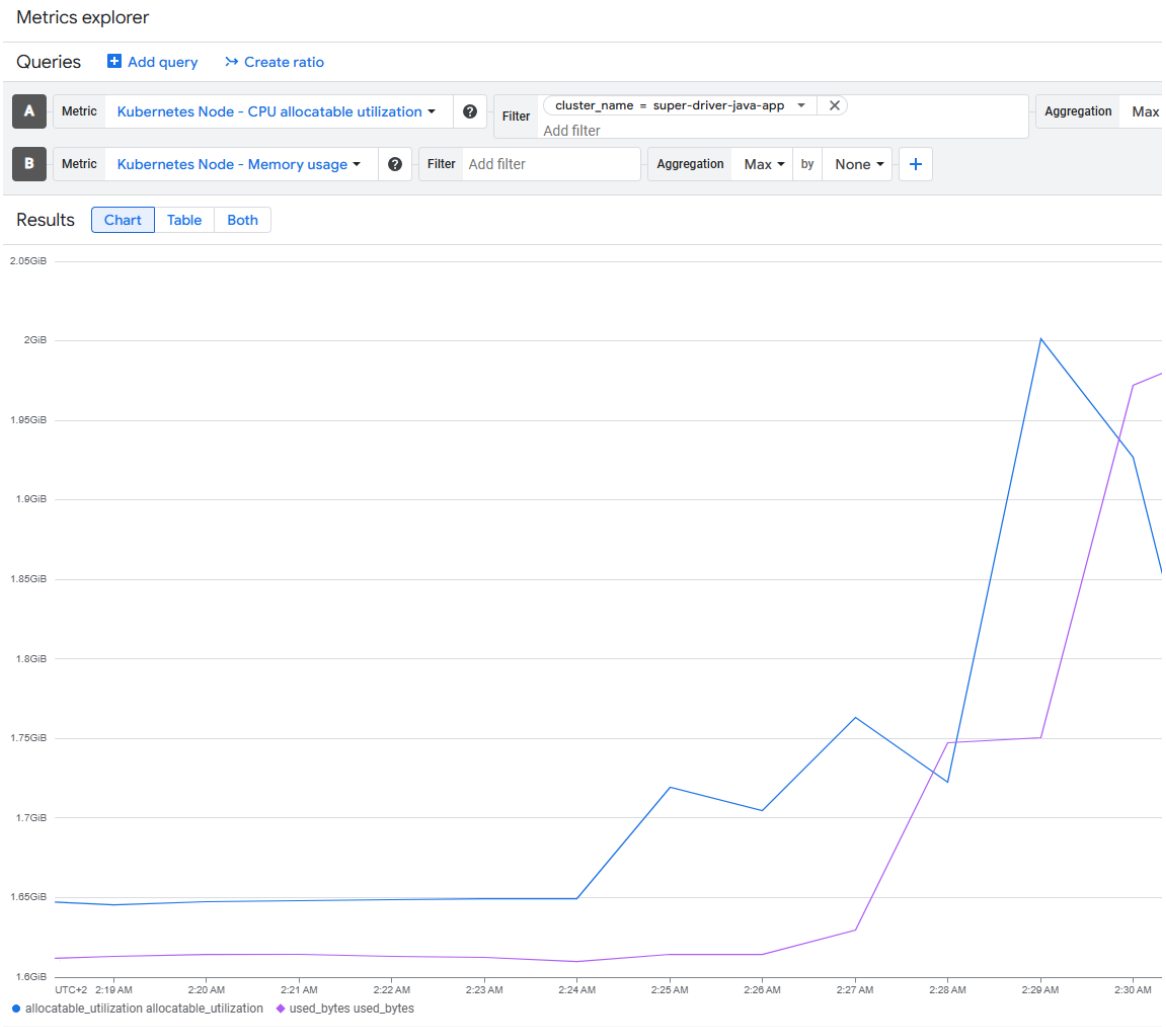


Рис 3.24 Моніторинг після навантаження

На Рисунку 3.24 показано два графіки, що відображають динаміку використання CPU та пам'яті вузлами мого кластера Kubernetes протягом останньої години.

Верхній графік (синій):

— показує відсоток використання доступних ресурсів CPU (Kubernetes Node - CPU allocatable utilization);

— чітко видно значне збільшення використання CPU під час навантажувального тестування, що свідчить про зростання потреби в обчислювальних ресурсах.

Нижній графік (фіолетовий):

— показує обсяг використаної пам'яті (Kubernetes Node - Memory usage) на вузлах кластера;

— спостерігається незначне збільшення використання пам'яті, що корелює зі збільшенням використання CPU.

Під час тестування Horizontal Pod Autoscaler (HPA) реагував на зростання використання CPU, автоматично збільшуючи кількість Pod'ів мого застосунку. Це дозволило динамічно адаптуватися до збільшеного навантаження та підтримувати продуктивність застосунку на належному рівні.

Наведені графіки наочно демонструють ефективність масштабування мого застосунку в Kubernetes за допомогою HPA. Збільшення використання ресурсів чітко корелює з періодом навантажувального тестування, а здатність HPA динамічно масштабувати кількість Pod'ів підтверджує правильність обраної стратегії масштабування.

ВИСНОВКИ

У даній роботі було проведено дослідження та розробку модуля масштабованих веб-застосувань на хмарній інфраструктурі Google Cloud Platform. В результаті виконання роботи було досягнуто поставленої мети та виконано всі поставлені завдання.

У першому розділі було проведено аналіз хмарних технологій та їх переваг для веб-застосувань. Було розглянуто різні архітектурні підходи до побудови масштабованих систем, включаючи мікросервісну архітектуру та безсерверну архітектуру. Також було детально описано платформу Google Cloud Platform та її сервіси, релевантні для розробки масштабованих веб-застосувань, такі як, Kubernetes Engine, Artifact Registry, Cloud Storage, Cloud Load Balancing та інші. Було розглянуто архітектурні патерни та методи забезпечення масштабованості, включаючи горизонтальне та вертикальне масштабування, балансування навантаження.

У другому розділі було проведено аналіз вимог до модуля масштабованих веб-застосувань, спроектовано його архітектуру та описано процес реалізації.. Було створено мікросервіс для програми з використанням Java та Spring Boot, спроектовано REST API для взаємодії з застосунком та обрано сервіси Google Cloud для зберігання даних та роботи з ними. Було також спроектовано мережеву інфраструктуру та налаштовано балансування навантаження. Особлива увага була приділена забезпеченню надійності, масштабованості та безпеки модуля.

У третьому розділі було описано процес розгортання застосунку на Google Cloud Platform з використанням Docker та Kubernetes. Було продемонстровано масштабування застосунку в дії та описано методи забезпечення безпеки модуля. Також було проведено моніторинг та аналіз роботи модуля, а також аналіз метрик продуктивності та використання ресурсів.

У цілому, робота демонструє практичне застосування хмарних технологій Google Cloud Platform для створення масштабованих та надійних веб-застосунків. Розроблений модуль може бути використаний як основа для розробки

різноманітних веб-застосунків, а отримані результати дослідження можуть бути корисними для розробників та архітекторів програмного забезпечення.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Thomas Erl, Ricardo Puttini, Zaigham Mahmood. Cloud Computing: Concepts, Technology & Architecture. Prentice Hall, 2013. 528 p.

2. Peter Mell, Timothy GranceNational NIST Definition of Cloud Computing *Institute of Standards and Technology (NIST).* 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

3. Evie Harrison, Reasons Why You Should Choose Google Cloud Over Others JULY 17, 2020, URL: https://geekflare.com/reasons-to-choosegoogle-cloud/

4. Sam Newman. Building Microservices: Designing Fine-Grained Systems, наук. посіб., 1-е вид, 2015 с. 64 – 98 .

5. Google Cloud Products. Google Cloud Documentation. URL: https://cloud.google.com/docs/overview/cloud-platform-services(date of access: 09.08.2024).

6. Site Reliability Engineering: How Google Runs Production Systems \ Betsy Beyer et al. London: O'Reilly Media, Inc., 2018. 550 p.

7. Навчальний курс Google Cloud Platform Fundamentals: Core Infrastructure. Coursea lerning. URL: https://www.coursera.org/learn/gcp-fundamentals (дата звернення 02.09.2024).

8. Aaron Keeports. Knowledge Base On Premise vs. Cloud. URL: https://www.cleo.com/blog/knowledge-base-onpremise-vs-cloud (date of access: 21.10.2024).

9. Čandrlić Ganesh. Types of cloud computing *GlobalDots Cloud Innovation Hunters*. Cloud computing. P. 4.

10 .Vertical Pod autoscaling. *Google Cloud*. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler(date of access: 11.09.2024)

11. Horizontal Pod autoscaling. *Google Cloud.* URL: https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler(date of access: 13.09.2024)

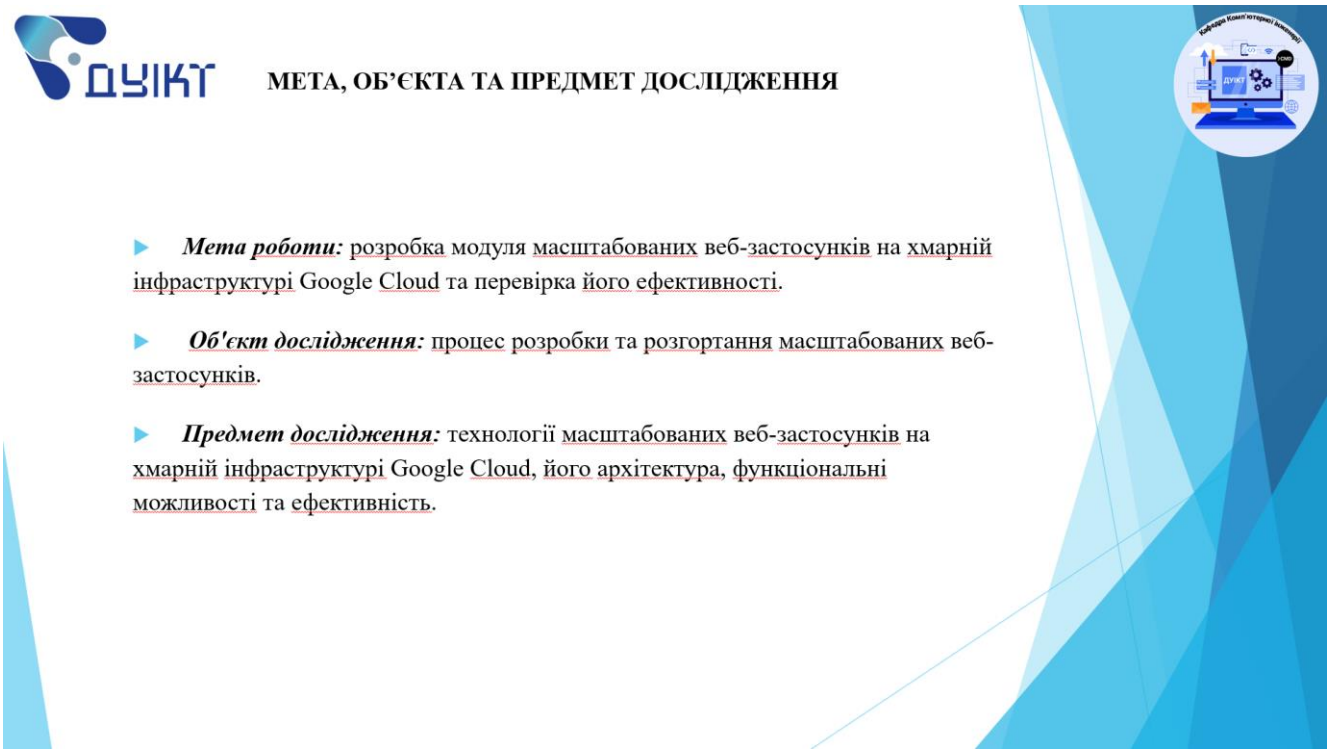
12. Adam Bellemare. Building Event-Driven Microservices: Leveraging Organizational Data at Scale 1st Edition. Toronto: O'Reilly . 2020.322 p.

13. Professional Cloud Architect for Associate Cloud Engineers. *Google Cloud.* URL: https://partner.cloudskillsboost.google/paths/117 (date of access: 25.08.2024 - 29.11.2024)

14. Load Balancing Analysis Using Round-Robin and Least-Connection Algorithms for Server Service Response Time / Applied Technology and Computing Science Journal. Indonesia:Jakarta, 2022. 54 p.

15. Google Cloud Platform Services Summary. *Google Cloud.* URL: https://cloud.google.com/terms/services?hl=en (date of access: 20.09.2024)

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



АКТУАЛЬНІСТЬ РОБОТИ

Хмарні обчислення стали невід'ємною частиною сучасного IT світу. Вони надають бізнесу та розробникам гнучкість, масштабованість та економічну ефективність, дозволяючи швидко реагувати на зміни ринку та зосередитись на основній діяльності.

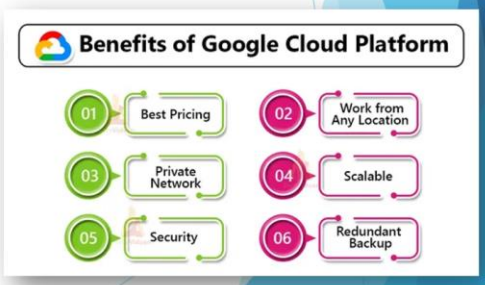
Сучасні веб-застосунки повинні обробляти величезні обсяги даних та велику кількість користувачів. Масштабованість є критично важливою для забезпечення безперебійної роботи та високої доступності застосунків, особливо в умовах пікових навантажень.

Google Cloud Platform (GCP) є одним з провідних хмарних провайдерів, що пропонує широкий спектр сервісів та інструментів для розробки, розгортання та масштабування застосунків. GCP поєднує в собі високу надійність, продуктивність, безпеку та інноваційні технології.

ОГЛЯД ПЛАТФОРМИ GCP

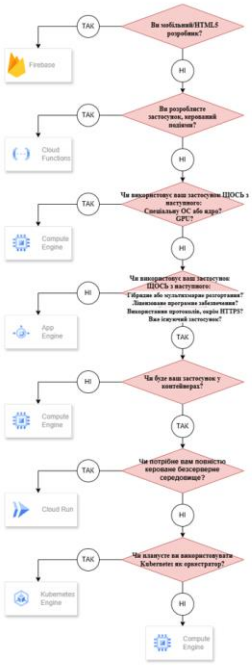
Google Cloud Platform не обмежується одним типом архітектури, а надає можливості для реалізації різних архітектурних підходів залежно від потреб вашого застосунку.

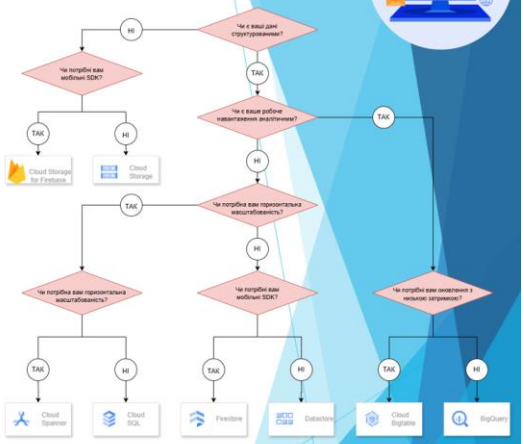
- Мікросервісна архітектура
- Безсерверна архітектура
- Багаторівнева архітектура
- Гібридна та багатохмарна архітектура



ВИБІР ОСНОВНИХ СЕРВІСІВ GOOGLE CLOUD ДЛЯ МОДУЛЯ

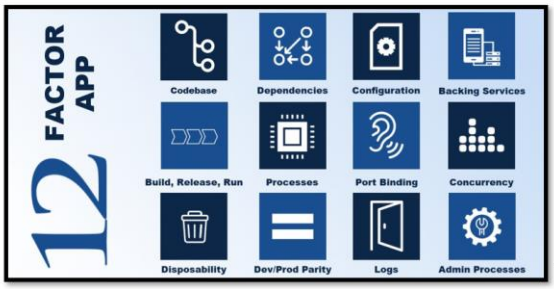
ОПЦІЯ	ПІДТРИМКА МОВ	МОДЕЛЬ ВИКОРИСТАННЯ	МАСШТАБУВАННЯ	ОСНОВНИЙ СЦЕНАРІЙ ВИКОРИСТАННЯ
COMPUTE ENGINE	Будь-яка	IaaS	Автомасштабування сервера	Загальні робочі навантаження
GKE (GOOGLE KUBERNETES ENGINE)	Будь-яка	IaaS, PaaS	Кластерне масштабування	Контейнерні робочі навантаження
APP ENGINE (СТАНДАРТНЕ СЕРЕДОВИЩЕ)	Python, Node.js, Go, Java, Ruby, PHP	PaaS	Автоматичне масштабування керованих серверів	Масштабовані веб-застосунки, мобільні бекенд-застосунки
APP ENGINE (ГНУЧКЕ СЕРЕДОВИЩЕ)	Python, Node.js, Go, Java, PHP, Ruby, .NET, власні середовища виконання	PaaS	Автоматичне масштабування керованих серверів	Масштабовані веб-застосунки, мобільні бекенд-застосунки
CLOUD FUNCTIONS	Python, Node.js, Go, Java	Мікросервісна архітектура	Безсерверне	Легкі події, дії
CLOUD RUN	Будь-яка	PaaS	Безсерверне	Розгортання та масштабування контейнеризованих застосунків



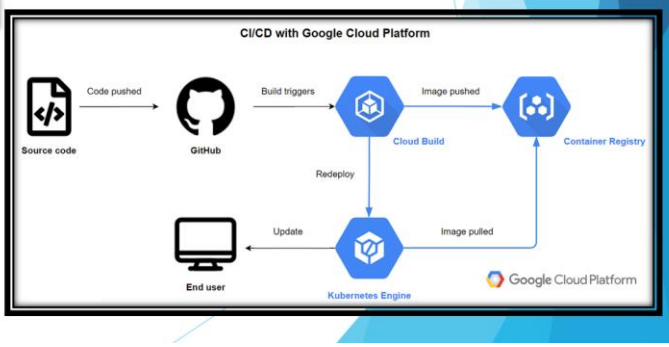


ОГЛЯД ОСНОВНИХ МЕТОДІВ І ХАРАКТЕРИСТИК МОДУЛЯ

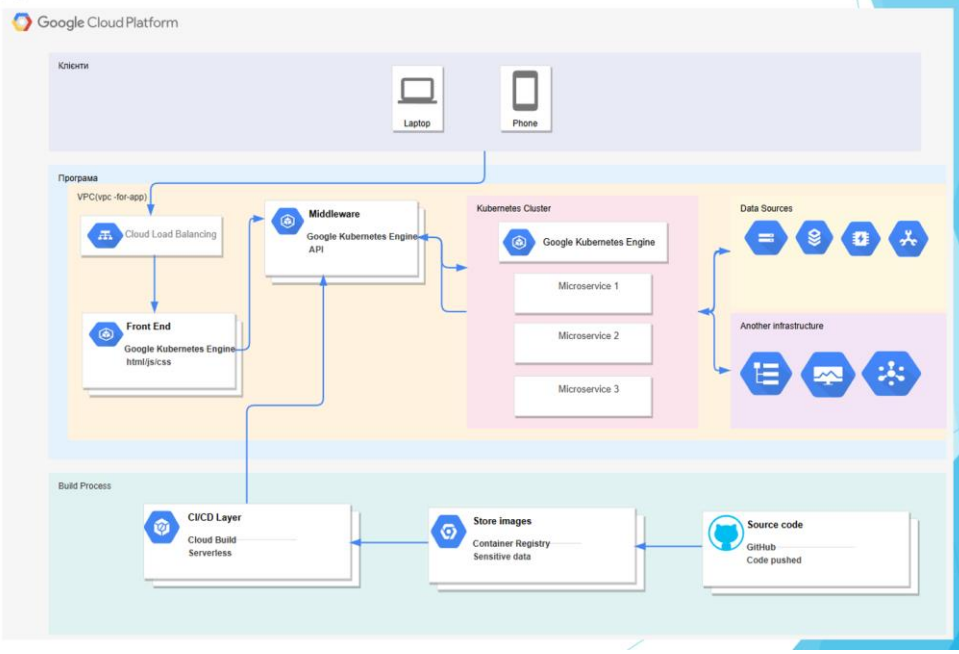
Проектування архітектури модуля методологією (The Twelve-Factor App)



Безперервна інтеграція в Google Cloud

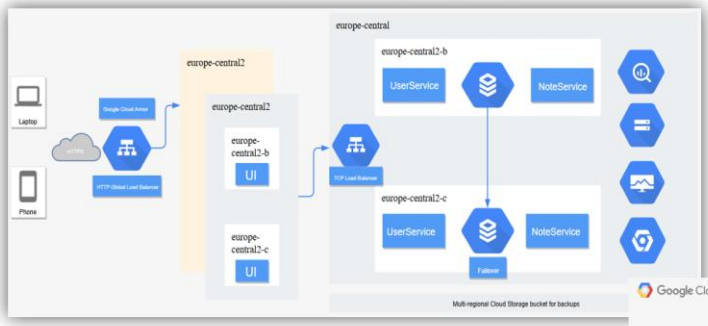


ЗАГАЛЬНИЙ ОГЛЯД МОДУЛЯ



РОЗРОБКА АРХІТЕКТУРИ МОДУЛЯ

Схема мережевої інфраструктури



Моделювання безпечних служб Google Cloud

РЕАЛІЗАЦІЯ МОДУЛЯ

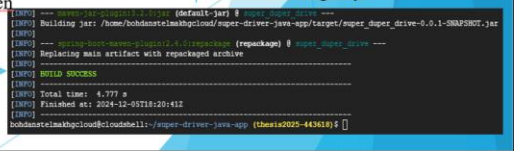
1.Копіювання

2.Тестування

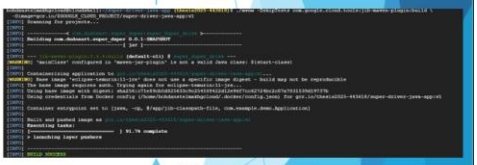
3.Запуск Maven Wrapper

4.Перевірка застосунка

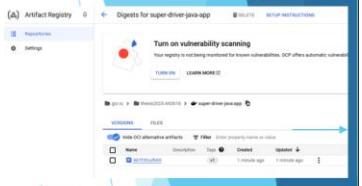
5.Контернізація Jib Maven



6. Імпорт в Google Container Registry



7.Docker в Google Container Registry

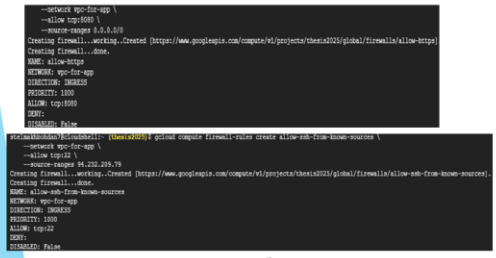


СТВОРЕННЯ БЕЗПЕЧНОЇ МЕРЕЖІ

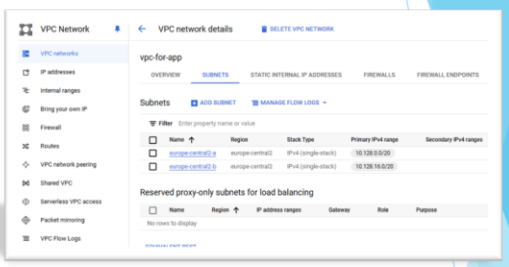
Створення VPC

Створення підмереж

Створення правил брандмауера



Загальний вигляд VPC з консолі Google Cloud



Створення Cloud Armour

РОЗГОРТАННЯ ЗАСТОСУНКУ В GKE

```
Creating cluster super-driver-java-app in europe-central2-a... Cluster is being health-checked (Kubernetes Control Plane is healthy)...done.
Created [https://container.googleapis.com/v1/projects/thesis2025-443618/zones/europe-central2-a/clusters/super-driver-java-app].
To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload_/gcloud/europe-central2-a/super-driver-java-app?project=thesis2025-443618
kubeconfig entry generated for super-driver-java-app.
NAME: super-driver-java-app
LOCATION: europe-central2-a
MASTER_VERSION: 1.30.5-gke.1443001
MASTER_IP: 34.118.101.14
MACHINE_TYPE: n1-standard-1
NODE_VERSION: 1.30.5-gke.1443001
NUM_NODES: 3
STATUS: RUNNING
```

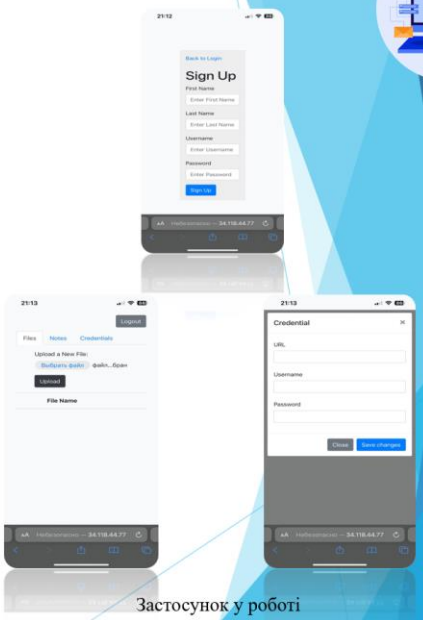
Створення кластера GKE

```
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl create deployment app-demo
deployment.apps/app-demo created
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
app-demo   1/1     1            1           37s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
app-demo   1/1     1            1           53s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl describe deployments
Name:                   app-demo
Namespace:              default
CreationTimestamp:      Thu, 05 Dec 2024 19:02:50 +0000
Labels:                 app=app-demo
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=app-demo
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=app-demo
```

Запуск застосунка у Pod'і

```
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl create service loadbalancer app-demo --tcp=8080:8080
service/app-demo created
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)          AGE
app-demo     LoadBalancer   34.118.231.225  <pending>      8080:31467/TCP   7s
kubernetes   ClusterIP      34.118.224.1    <none>         443/TCP          15m
rest-demo    LoadBalancer   34.118.227.58   34.118.9.37    8080:31250/TCP   4m10s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)          AGE
app-demo     LoadBalancer   34.118.231.225  34.118.46.77   8080:31467/TCP   72s
kubernetes   ClusterIP      34.118.224.1    <none>         443/TCP          16m
rest-demo    LoadBalancer   34.118.227.58   34.118.9.37    8080:31250/TCP   5m15s
bohdanstelmakhgcloud@cloudshell:~/super-driver-java-app (thesis2025-443618)$
```

Створення LoadBalancer і Public IP для доступа до програми



Застосунок у роботі

ТЕСТУВАННЯ ЗАСТОСУНКУ

Apache Benchmark

```
C:\Users\stelm>ab -n 100 -c 10 http://34.118.95.49:8080/home
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 34.118.95.49 (be patient).....done


Server Software:
Server Hostname:        34.118.95.49
Server Port:            8080

Document Path:          /home
Document Length:        0 bytes

Concurrency Level:      10
Time taken for tests:   1.682 seconds
Complete requests:      100
Failed requests:        0
Non-2xx responses:      100
Total transferred:      39100 bytes
HTML transferred:       0 bytes
Requests per second:    59.47 [#/sec] (mean)
Time per request:       168.159 [ms] (mean)
Time per request:       16.816 [ms] (mean, across all concurrent requests)
Transfer rate:          22.71 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       10   16   3.4     16      20
Processing:    20  139  25.6    165     180
Waiting:       19   99  37.2    100     152
Total:         40  155  25.7    160     198

Percentage of the requests served within a certain time (ms)
  50%    160
  66%    162
  75%    163
  80%    165
  90%    167
  95%    170
  98%    170
  99%    198
 100%    198 (Longest request)

C:\Users\stelm>
```

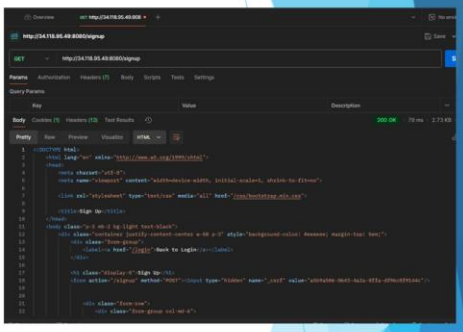
Час виконання: 1.682 секунди

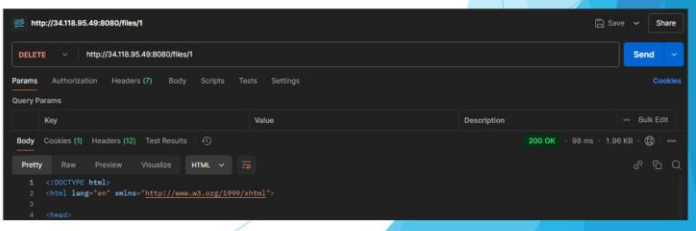
Успішні/невдалі запити: 100/0

Запити за секунду: 59.47

Середній час відповіді: 168.159 мс

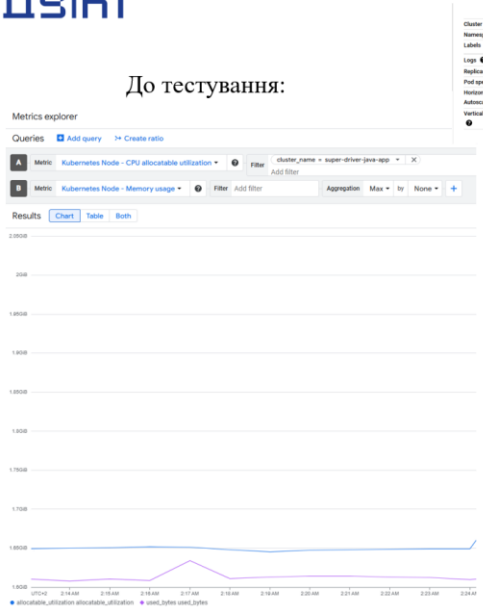
Postman



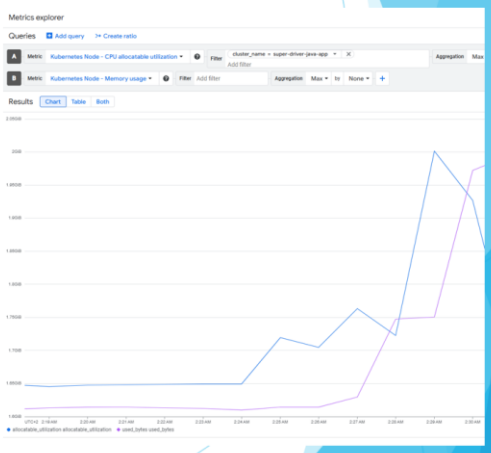


МОНІТОРИНГ І ЛОГУВАННЯ МОДУЛЯ

До тестування:



Після тестування:



ВИСНОВКИ

У даній магістерській роботі було проведено дослідження та розробку модуля масштабованих веб-застосувань на хмарній інфраструктурі Google Cloud Platform. В результаті виконання роботи було досягнуто поставленої мети та виконано всі поставлені завдання.

У першому розділі було проведено аналіз хмарних технологій та їх переваг для веб-застосувань. Було розглянуто різні архітектурні підходи до побудови масштабованих систем, включаючи мікросервісну архітектуру та безсерверну архітектуру. Також було детально описано платформу Google Cloud Platform.

У другому розділі було проведено аналіз вимог до модуля масштабованих веб-застосувань, спроектовано його архітектуру та описано процес реалізації. Було створено мікросервіс для програми з використанням Java та Spring Boot, спроектовано REST API для взаємодії з застосунком та обрано сервіси Google Cloud. Було також спроектовано мережеву інфраструктуру та налаштовано балансування навантаження.

У третьому розділі було описано процес розгортання застосунку на Google Cloud Platform з використанням Docker та Kubernetes. Було продемонстровано масштабування застосунку в дії та описано методи забезпечення безпеки модуля. Також було проведено моніторинг та аналіз роботи модуля, а також аналіз метрик продуктивності та використання ресурсів.

ДОДАТОК Б. БЕКЕНД ЗАСТОСУНКУ