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

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено:

Завідувач кафедри

_____ Олександр Коваль

«___»_____2020 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Програмне забезпечення веб-технологій та мобільних пристроїв»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Горизонтальне масштабування мікросервісів в хмарному Kubernetes кластері»

Виконала:
студентка IV курсу, групи ТІ-62
Войналович Валерія Анатоліївна _____

Керівник:
доцент, к.т.н.,
Смаковський Денис Сергійович _____

Рецензент:
доцент, к. ф.-м. н.,
Галкін Олександр Володимирович _____

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

Київ – 2020 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки: 121 Інженерія програмного забезпечення

Спеціалізація: Програмне забезпечення веб-технологій та мобільних пристроїв

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ
на дипломну роботу студенту

_Войналович Валерії Анатоліївні_____
(прізвище, ім’я, по батькові)

1. Тема роботи _Горизонтальне масштабування мікросервісів в хмарному Kubernetes кластері_____

керівник роботи_ к.т.н., доцент Смаковський Денис Сергійович_____
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”25” травня 2020р. **№1168-с**

2. Строк подання студентом роботи _____
3. Вихідні дані до роботи _мова програмування Java, фреймворки Spring, Spring Boot, Spring Integration, Spring Cloud GCP, платформа Kubernetes, платформа Google Cloud Platform._____
4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) _описати більш оптимальний алгоритм масштабування мікросервісів у хмарному кластері Kubernetes з використанням інформації про довжину черги, реалізувати систему з використанням описаного алгоритму: розробити програмне забезпечення веб-сервісів системи, налаштувати автоматичне розгортування та горизонтальне масштабування системи засобами Kuubernetes._____

5. Перелік ілюстративного матеріалу

_Актуальність, Мета та завдання, Стандартна архітектура мікросервісних систем, Опис використаних технологій, Опис системи без масштабування, Опис структури системи, Приклад роботи системи, Інструмент для тестування, Результати експерименту, Апробація роботи, Висновки_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання "_11_"___жовтня_____2019__ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи	15.10.2019	
2.	Вивчення та аналіз задачі	27.03.2020 – 03.04.2020	
3.	Розробка архітектури та загальної структури системи	09.04.2020 – 16.04.2020	
4.	Розробка структур окремих підсистем	17.04.2020 – 29.04.2020	
5.	Програмна реалізація системи	30.04.2020 – 12.05.2020	
6.	Оформлення пояснювальної записки	03.05.2020 – 05.06.2020	
7.	Захист програмного продукту	26.05.2020	
8.	Передзахист	10.06.2020	
9.	Захист	16.06.2020	

Студент _____ _Войналович В.А._____
(підпис) (прізвище та ініціали,)

Керівник роботи _____ _Смаковський Д.С._____
(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

Метою даної роботи є створення системи з реалізацією оптимального алгоритму горизонтального масштабування у хмарному кластері Kubernetes з використанням інформації про довжину черги, що дозволяє збільшити ефективність використовуваних ресурсів в системі.

Було реалізовано систему з більш оптимальним алгоритмом горизонтального масштабування на основі кількості повідомлень у черзі. Було виконано тестування системи, яке показало, що створена система здатна оброблювати більш, ніж у 2 рази більшу кількість запитів за однаковий проміжок часу порівняно з системою без масштабування.

Робота була надіслана на наступні конференції: XVIII міжнародна науково-практична конференція молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики» [1] та VI Міжнародної науково-практичної конференції «Сталий розвиток – XXI століття (наукові читання імені Ігоря Недіна)».

Дипломна робота містить: 61 сторінку, 18 рисунків, 1 таблицю, 3 додатки та 16 джерел.

Ключові слова: мікросервісна архітектура, Kubernetes, горизонтальне масштабування на основі довжини черги, Google Cloud Platform, Spring Boot, Java.

ABSTRACT

The purpose of this work is to create a system with the implementation of the optimal horizontal scaling algorithm in the Kubernetes cloud cluster using queue length information, which increases the efficiency of resources used in the system.

A system with a more optimal horizontal autoscaling algorithm based on the number of messages in the queue was implemented. System testing was performed, which showed that the created system is capable of processing more than 2 times more requests in the same period of time compared to the system without scaling.

The work was sent to the following conferences: XVIII International scientific and practical conference of young scientists and students "Modern problems of scientific support of power engineering" [1] and VI International Scientific and Practical Conference "Sustainable Development — XXI Century (scientific readings named to Igor Nedin)".

This diploma work contains: 61 pages, 18 figures, 1 table, 3 appendices and 16 sources.

Keywords: microservice architecture, Kubernetes, horizontal scaling based on queue length, Google Cloud Platform, Spring Boot, Java.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API — Application Programming Interface

GCP — Google Cloud Platform

REST — Representational State Transfer

PRI — Remote Procedure Invocation

Pub/Sub — Google Cloud Pub/Sub

HPA — Horizontal Pod Autoscaler

ВСТУП

У сучасному світі інформаційних технологій Інтернет займає вагоме місце у житті багатьох: більшість інформації та різних операцій відбуваються саме через Інтернет. Але із зростанням використання різних веб-сервісів, зростає навантаження на веб-сервіси, що може призвести до затримок у роботі або навіть виходу с ладу цих сервісів. А тому стають вкрай важливими питання створення надійних, відмовостійких та спроможних до масштабування систем.

Якщо навантаження є більшим, ніж система або сервіс може витримати, то це може спричинити відмову в обслуговуванні або припинення роботи сервісу. Також навантаження може нерівномірно розподілятися на сервіси протягом певного проміжку часу, а тому, навіть, якщо система матиме достатньо ресурсів, щоб витримувати високе навантаження, то в періоди низького навантаження ці ресурси не будуть використовуватися, а з цього випливають проблеми неефективного використання ресурсів, а також перевитрата коштів.

Запропонована до розробки система буде реалізована на основі мікросервісної архітектури, що дозволить масштабувати тільки окрему частину системи, на яку буде створюватись більше навантаження. Використання Kubernetes надає системі еластичності та стійкості до збоїв шляхом самовідновлення. Використання черги вирішує наступні проблеми: якщо сервіси не встигають обробляти всі запити, які надходять, або один з сервісів недоступний, черга виконує роль буфера і зберігає запити до тих пір, поки вони не будуть опрацьовані; також з використанням черги зменшується зв'язність сервісів і тим самим підвищується їх незалежність.

Оптимізований алгоритм горизонтального масштабування реалізований у роботі може бути використаний високонавантаженими веб-сервісами з майже з будь-якої сфери надання послуг, що потребують рішення для підвищення надійності системи та оптимізації використовуваних ресурсів. Це веб-сервіси, які

мають багато користувачів або отримують велику кількість запитів від користувачів або інших сервісів за короткі проміжки часу, наприклад, інтернет-магазини, соціальні мережі тощо.

1 ЗАДАЧА МАСШТАБУВАННЯ МІКРОСЕРВІСІВ У КЛАСТЕРІ KUBERNETES

У рамках цієї роботи поставлена задача створити систему на основі мікросервісної архіктектури у хмарному кластері Kubernetes [2] з використанням масштабування сервісів. Уявимо систему, що складається з двох сервісів: один приймає запити від користувачів і передає їх далі для обробки. Позначимо цей сервіс як видавець. Другий сервіс буде отримувати запити, надіслані видавцем, та оброблювати їх. Позначимо другий сервіс як споживач. Поміж двома сервісами розмістимо хмарний сервіс Google Cloud Pub/Sub [3] (надалі — Pub/Sub), який і буде виконувати роль черги. Отже, видавець буде отримувати запити з-поза меж системи, надсилати до Pub/Sub, споживач у свою чергу буде опитувати Pub/Sub щодо наявності неопрацьованих запитів, споживати ці запити і потім опрацьовувати їх. Дана система буде розвернута в Google Cloud Platform — запропонований компанією Google набір хмарних служб [3] — з використанням Kubernetes — відкритою системою автоматичного розгортання, масштабування та управління застосунками у контейнерах.

В роботі необхідно виконати наступні завдання:

— створити описану систему;

— налаштувати горизонтальне масштабування сервіса споживача на основі неопрацьованих запитів у черзі;

— провести тестування системи для виявлення якості роботи системи;

— провести обчислювальний експеримент шляхом тестування системи під великим навантаженням з використанням інструменту для тестування навантаженням — JMeter [5];

— в ході експерименту зробити заміри результатів обробки встановленої кількості запитів надісланих у встановлений проміжок часу з метою підтвердження ефективності запропонованої системи;

— проаналізувати отримані результати.

Опис розділів пояснювальної записки наступний:

— у першому розділі описано постановку задачі дипломної роботи;

— у другому розділі наведено огляд архітектурного підходу, який використовувався для реалізації дипломної роботи, а також огляд використаних технологій, опис їх компонентів, переваги та недоліки;

— у третьому розділі наведено опис засобів реалізацій застосунків з мікросервісною архітектурою, виконано їх порівняння, а також надано аргументацію, чому у ході дипломної роботи було використано обраний засіб;

— у четвертому розділі описана реалізація системи дипломної роботи: наведено опис створених сервісів, діаграми класів, взаємодія сервісів з іншими компонентами системи, послідовно викладено метод налаштування всіх компонентів системи, продемонстровано приклад роботи системи;

— у п’ятому розділі описано обчислювальний експеримент над системою, який був проведений методом тестування навантаженням, проаналізовано отримані результати.

2 ПОБУДОВА МІКРОСЕРВІСНИХ ЗАСТОСУНКІВ ТА ЇХ РОЗМІЩЕННЯ В ХМАРНОМУ KUBERNETES КЛАСТЕРІ

У даному розділі буде представлений опис архітектурного підходу мікросервісних систем, описано його переваги, а також надано огляд важливих функціональних компонентів таких систем. Для деяких компонентів буде наведено різні підходи реалізації та перераховано переваги та недоліки кожного підходу. Крім цього, буде подано опис платформи Kubernetes, за допомогою якої буде налаштоване розгортання мікросервісної системи. У розділі буде йтися про основні компоненти Kubernetes, їх взаємодію та функціональне призначення. У кінці буде наведено опис шаблону спілкування видавець-підписник.

2.1 Архітектура мікросервісів

Архітектурний стиль мікросервісів – це підхід, при якому єдиний застосунок будується як набір невеликих сервісів, кожний з яких працює у власному потоці та взаємодіє з іншими сервісами використовуючи легкі з точки зору використання ресурсів механізми, такі як HTTP — HyperText Transfer Protocol — протокол прикладного рівня передачі даних. Ці сервіси будуються на основі бізнес потреб застосунка та розгортаються незалежно з використанням повністю автоматизованого середовища. Ці сервіси можуть бути написані різними мовами та використовувати різні технології збереження даних.

Сервіси створені на основі мікросервісної архітектури зазвичай мають наступні властивості:

— їх роботу легко підтримувати та тестувати;

— слабко зв’язані;

— незалежно розгортуються;

— організовані навколо можливостей бізнесу;

— розроблюються маленькими командами.

Діаграма стандарної мікросервісної архітектури зображена на рисунку 2.1.

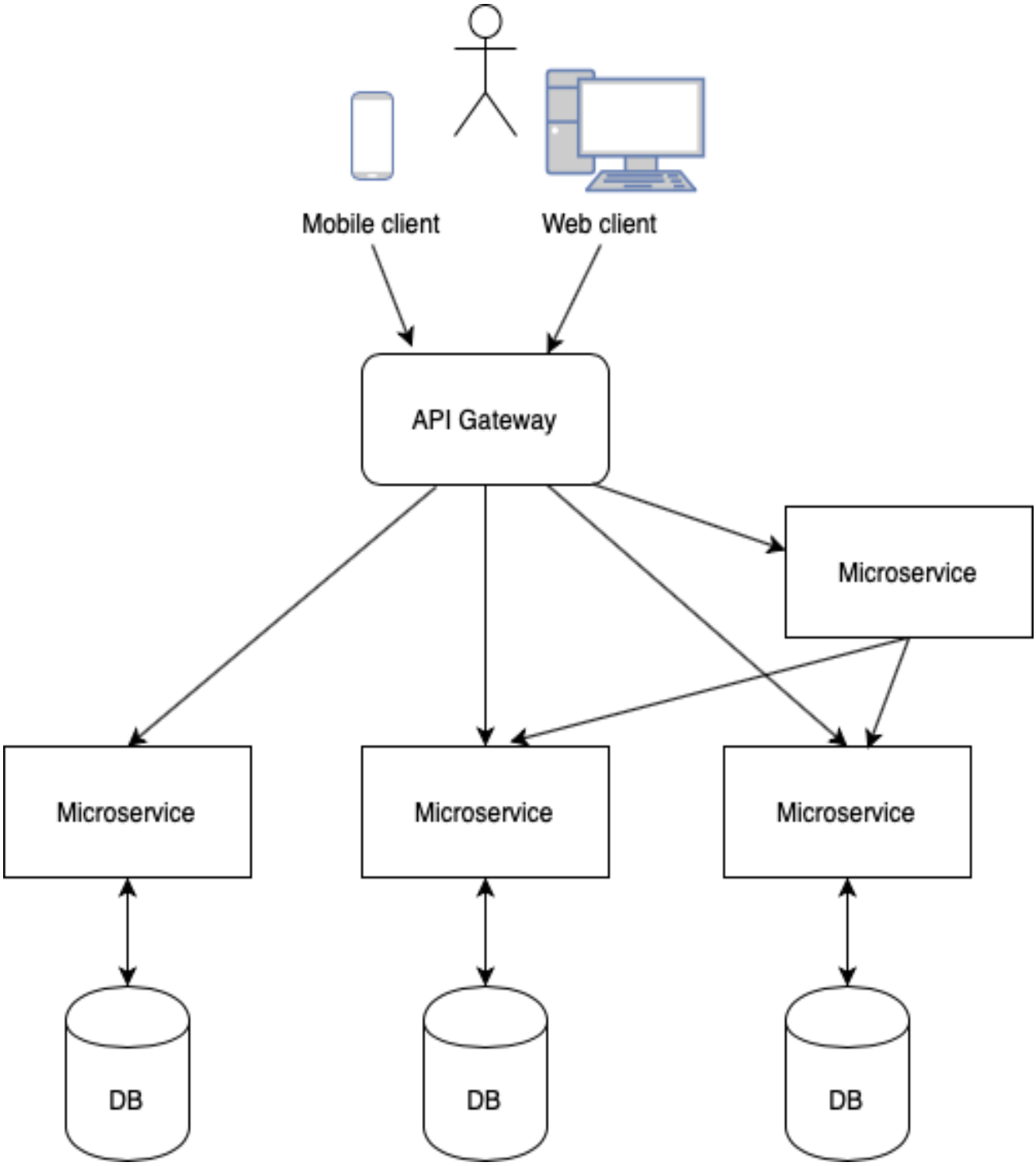


Рисунок 2.1 – Діаграма стандарної архітектури мікросервісної системи

Мікросервісна архітектура зумовлює швидку, часту та надійну доставку великих, складних застосунків та функціоналу. Вона також дозволяє організаціям переходити на більш новий технологічний стек.

Проте рішення з мікросервісною архітектурою має певну кількість недоліків:

— розробники повинні справлятися з додатковою складністю розробки розподілених систем. Розробники повинні реалізувати міжсервісну взаємодію та справлятися з частковим збоєм, реалізувати запити, які проходять через декілька сервісів, тестувати взаємодію між сервісами, що є значно складнішим, ніж у монолітній архітектурі;

— складність розгортання всієї системи. У виробничих умовах підвищується складність розгортання та управління системи, що складається з багатьох різних сервісів;

— збільшене використання пам’яті (конкретно для сервісів написаних мовою Java). З кожним сервісом запускається одна JVM, що значно підвищує надмірне використання ресурсів.

2.1.1 Service registry

При побудові мікросервісної архітектури очевидним фактом є те, що сервіси повинні взаємодіяти між собою певним чином. В монолітних застосунках сервіси взаємодіяли за допомогою мовних засобів або викликів процедур. В традиційних розподілених системах сервіс розгортається на фіксованому та відомому місцезнаходженні (хост та порт) таким чином, що інші сервіси можуть взаємодіяти через протокол HTTP та REST (Representational State Transfer – архітектурний стиль взаємодії компонентів розподіленого застосунку у мережі) або методом виклику віддалених процедур (RPC). Проте, сучасні застосунки мікросервісної архітектури здебільшого запускаються у віртуалізованих або контейнерізованих середовищах, що призводить до того, що кожен сервіс має

змінюване місцезнаходження — динамічну IP-адресу. Отже, постає проблема знаходження сервісів із динамічним IP-адресом іншими сервісами.

Для вирішення цієї проблеми необхідно запровадити механізм знаходження сервісів (Service Discovery) з використанням реєстру сервісів (Service Registry) [6]. Так, коли одному сервісу потрібно буде визвати інший сервіс, він звернеться до Service Registry, який знає актуальне місцезнаходження усіх сервісів і надасть адресу потрібного сервісу, і тоді перший сервіс звернеться по цій адресі. Механізм місцезнаходження сервісів може бути реалізований двома шляхами: знаходження сервісів на стороні клієнту (Client-side service discovery) [6] та на стороні серверу (Server-side service discovery) [6]. Знахождення сервісів на стороні клієнта реалізується шляхом додавання HTTP клієнта, який може взаємодіяти з Service Registry, на стороні клієнтського сервісу. Знаходження сервісів на стороні серверу реалізується за допомогою додавання та налаштування роутера (або балансувальника навантаження), який має відоме місцезнаходження. Запит потрапляє на роутер, який в свою чергу звертається до Service Registry, який також може бути вбудованим в роутер, після чого роутер перенаправляє запит за отриманою адресою доступного сервісу. Порівняно із знаходження сервісів на стороні сервера, знаходження на стороні клієнта має менше змінюваних складових, а також меншу кількість мережевих переходів.

Недоліки знаходження на стороні клієнта складаються з:

— зв'язування клієнта з Service Registry;

— необхідність реалізовувати логіку знаходження сервісів на стороні клієнта для кожної мови програмування або фреймворку у застосунку.

Переваги знаходження сервісів на стороні серверу:

— у порівнянні із знаходженням на стороні клієнта цей підхід має простіший код, адже не потрібно вирішувати задачу знаходження місцезнаходження, а просто зробити запит на роутер;

— деякі хмарні середовища забезпечують цей функціонал із коробки.

Знаходження на стороні серверу також має деякі недоліки:

— допоки це не є частиною хмарного середовища, роутер представляє собою ще одну частину системи, яка має бути сконфігурована та налаштована;

— роутер має підтримувати необхідні протоколи взаємодії (наприклад HTTP, gRPC, Thrift, тощо), допоки це не роутер на основі TCP (Transmission Control Protocol – один із основних протоколів передачі даних Інтернету, що призначається для управління передачі даних);

— необхідно більше мережевих переходів порівняно із знаходженням на стороні клієнта, що збільшує час необхідний для обробки запиту, що пов’язано із мережевими затримками.

При використанні шаблону знаходження сервісів виникає питання того, як сервіси будуть вноситися до регістру сервісів при створенні та видалятися при зупинці, а також, коли сервіс запущений, але не спроможний опрацьовувати запити. Для вирішення цієї проблеми використовують два підходи реєстрації та видалення сервісів: самореєстрація сервісу (Self Registration) або реєстрація сервісу за допомогою побічних сервісів (3rd Party Registration). Як можна зрозуміти з назв, у випадку самореєстрації сервіс сам відповідає за свою реєстрацію чи видалення з реєстру завдяки додатковій логіці у коді, у випадку реєстрації за допомогою побічних сервісів інший сервіс є відповідальним за реєстрацію та видалення сервісів з реєстру.

У системі із знаходженням місцерозташування та реєстрації сервісів важливою складовою є перевірка здоров’я сервісу — Health Check API. Уявимо ситуацію, коли певний сервіс запущений, але не може обробляти запити з певних причин, тоді найкращим виходом із даної ситуації буде сигналізування про те, що сервіс недієздатний, щоб роутер або балансувальник навантаження перестали надсилати запити на цей сервіс. Для створення механізму виявлення недієздатності сервісу у сервісі реалізують кінцеву точку API перевірки здоров’я, яка повертає інформацію про стан здоров’я сервісу. Сервіси, такі як: моніторингові сервіси, реєстр сервісів, балансувальник навантаження, для яких необхідно знати стан інших сервісів, періодично опитують цю кінцеву точку, щоб дізнатися стан сервісу.

2.1.2 API Gateway

Уявімо застосунок, серверна частина якого реалізована за допомогою мікросервісної архітектури, а інтерфейс користувача представлений за допомогою веб-додатку та мобільного додатку. Також цей застосунок можуть викликати інші побічні застосунки. Зазвичай, кожен мікросервіс надає певний API, який надає дані стосовно однієї певної предметної області, в той час як користувачу необхідні дані про декілька предметних областей і одразу. Наприклад, на сторінці інтернет-магазину користувач хоче бачити інформацію про товари, їх кількість, відгуки про товар, кабінет користувача. Зазвичай, дані про кожен з таких пунктів надає окремий мікросервіс, а це означає, що користувачеві потрібно викликати багато сервісів, щоб зібрати всю потрібну інформацію. Також, очевидним фактом є те, що для мобільної версії застосунку необхідно менше інформації, аніж для браузера на робочому столі комп'ютера, з міркувань максимально можливого обсягу інформації, який можливо одночасно вивести на екран. Також різні користувачі мають різну швидкість передачі по мережі, наприклад, мобільний або стаціонарний Інтернет-зв'язок. Мобільний зв'язок часто повільніший, ніж стаціонарний, а тому вимагає меншу кількість запитів до серверу, щоб не погіршити досвід користування застосунком. З огляду на всі описані пункти постає проблема, як тоді спроектувати систему, щоб клієнти могли оптимально доступатися до серверної частини. Вирішенням такої проблеми стала реалізація API Gateway [6] — єдиної точки входу до серверної частини для всіх клієнтів. Також схематичне розташування API Gateway можна знайти на рисунку 2.1. API Gateway опрацьовує запити двома шляхами: деякі запити просто перенаправляються до необхідного сервісу, а для інших виконується певна кількість запитів до декількох різних сервісів. Також можливе певне розподілення API Gateway: API Gateway може надавати окрему точку доступу для кожного клієнта (окремо для веб-застосунку, для мобільного додатку тощо). Така варіація API Gateway називається Backends for frontends.

API Gateway має наступні переваги:

— ізолює клієнтів від знання про те, як застосунок розбитий на мікросервіси;

— позбавляє клієнтів проблеми визначення місцезнаходження екземплярів сервісу;

— може забезпечувати оптимальний API для кожного клієнта;

— зменшує кількість запитів, а тому API Gateway необхідний для мобільних застосунків;

— спрощує клієнт шляхом перенесення логіки викликів багатьох сервісів з клієнта до API Gateway;

— перетворює стандартний публічний веб-протокол до будь-якого іншого, який застосовується поміж мікросервісами.

Недоліками API Gateway називають наступні пункти:

— збільшення складності інфраструктури, адже API Gateway — ще одна складова, яку потрібно розгортати, настроювати та керувати нею;

— збільшення часу відповіді через те, що API Gateway потребує додаткових викликів, проте це збільшення для більшості застосунків може бути мало відчутним.

2.1.3 Service communications

Нехай, маємо застосунок з мікросервісною архітектурою, який має обробляти запити від користувачів. Для цього сервіси повинні взаємодіяти між собою використовуючи певний протокол. Найчастіше вживаними є два підходи у вирішенні Цієї проблеми: віддалений виклик процедури RPI (Remote Procedure Invocation) та обмін повідомленнями (Messaging). При RPI клієнт застосовує протокол, що базується на виклику/відповіді, щоб зробити запити до сервісу [6]. Прикладами RPI є такі технології: REST, gRPC, Apache Thrift. На рисунку 2.1 спілкування такого виду між компонентами зображене за допомогою однонаправлених стрілок. Переваги такого підходу:

— будування запитів у форматі запит/відповідь, що є простим та знайомим підходом для більшості розробників;

— більш проста архітектура системи, адже не потрібно встановлювати проміжний брокер.

Недоліком RPI є зменшення доступності, оскільки і клієнт, і сервіс повинні бути доступними протягом часу всієї взаємодії.

Іншим підходом у вирішенні проблеми взаємодії сервісів є асинхронний обмін повідомленнями через певні канали зв’язку. Розрізняють деякі види асинхронного зв’язку [6]:

— запит/відповідь — сервіс відправляє запит отримувачу і очікує негайно отримати відповідь;

— повідомлення — сервіс відправляє запит отримувачу і не очікує відповіді;

— запит/асинхронна відповідь — сервіс відправляє запит отримувачу і очікує отримати відповідь в кінцевому рахунку;

— публікація/підписка (publish/subscribe) — сервіс публікує повідомлення для будь-якої кількості отримувачів;

— публікація/асинхронна відповідь — сервіс публікує повідомлення для одного або більше отримувачів, і деякі з них надсилають відповідь.

Відомі реалізації асинхронного обміну повідомленнями — Apache Kafka, RabbitMQ.

Цей підхід має наступні переваги:

— відв’язує відправника повідомлень від отримувача;

— покращує доступність системи, оскільки брокер буферизує повідомлення, допоки отримувач не зможе їх опрацювати;

— підтримує різні види зв’язку, що наведені вище.

Недоліком асинхронного спілкування є наявність брокера повідомлень, що підвищує складність системи, крім того, брокер повинен бути високодоступним.

Отже, у підрозділі 2.1 було зроблено загальний огляд мікросервісної архітектури, продемонстровано стандартну топологію розміщення компонентів у системі. Було наведено опис компонентів, які грають важливу роль у системі, наприклад, Service Discovery, API Gateway та Messaging. Було сказано про різні підходи для реалізації деяких компонентів, вказано їх переваги та недоліки.

2.2 Архітектура Kubernetes

Kubernetes — це портативна, розширювана платформа з відкритим вихідним кодом для управління контейнерізованними робочими елементами (workloads) та сервісами (Kubernetes Services), що полегшує і декларативну конфігурацію, і автоматизацію. В основі ідеї Kubernetes лежать контейнери. Контейнери схожі на віртуальні машини, але мають більш слабку ізоляції для спільного використання операційної системи між застосунками. Тому контейнери вважаються більш легкими. Контейнери — гарний спосіб зібрати докупи та запустити застосунок. На практиці необхідно керувати контейнерами, що запускають застосунок, та слідкувати, щоб не було перебоїв у роботі. Kubernetes допомагає впоратися з цими задачами: Kubernetes надає фреймворк для запуску розподілених систем без перебоїв. Kubernetes забезпечує наступні можливості [2]:

— місцезнахождення сервісів та балансування навантаження (Service Discovery та Load Balancing) — Kubernetes може виставляти контейнер використовуючи DNS ім’я або використовуючи його власну IP-адресу. Якщо трафік, що поступає до контейнера, має великий рівень, Kubernetes спроможний балансувати навантаження та розподілити трафік мережі таким чином, щоб розгортання (Deployment) був стабільний;

— оркестрація сховищ — Kubernetes дозволяє автоматично під’єднувати будь-яку систему сховища (локальну, хмарну, тощо);

— автоматичні розгортування та відкати — Kubernetes дозволяє описувати бажаний стан розгорнутих контейнерів і може змінювати поточний стан контейнерів для досягнення бажаного стану;

— automatic bin packing — можна надати Kubernetes кластер вузлів, який він буде використовувати для запуску контейнеризованих завдань. Можна задати Kubernetes, скільки пам’яті та CPU необхідно для кожного контейнера, і тоді Kubernetes буде розміщувати контейнери на вузлах з урахуваннями найбільш оптимального способу розміщення ресурсів;

— самовідновлення — Kubernetes може перезапускати контейнери, що вийшли з ладу, замінювати контейнери, прибирати контейнери, що не відповідають на задані перевірки здорового стану, і не пропонує їх використання, поки вони не здатні опрацьовувати запити;

— управління секретами та конфігурацією (Secret and configuration management) — Kubernetes дозволяє зберігати та управляти конфіденційною інформацією, такою як: паролі, токени OAuth, SSH ключі. Можна розгортати та оновлювати конфіденційну інформацію та конфігурації без перебудування образів контейнерів.

2.2.1 Компоненти Kubernetes

При розгортанні Kubernetes отримуєш кластер. Кластер — це набір комп’ютерів, сховищ даних і мережевих ресурсів, за допомогою яких Kubernetes виконує задачі у системі [7]. Також система може складатися з декількох кластерів.

Кластер складається з набору робочих машин, які називаються вузлами, на яких запускаються контейнеризовані застосунки. Кожен кластер складається хоча б з одного вузла. Вузлом є окремий комп’ютер, фізичний або віртуальний. Вузол складається з декількох компонентів, таких як kubelet і kube-proxy.

На робочих вузлах розміщуються поди (Pods), які є компонентами завантаженості програми. Под — це одиниця роботи в Kubernetes. Кожний под

має один або декілька контейнерів. Усі контейнери всередині подів мають одну й ту саму IP-адресу і простір портів, вони можуть спілкуватися між собою через локальний сервер або через міжпроцесорну взаємодію.

Керуючий вузол управляє робочими вузлами та подами у кластері. На практиці зазвичай керуючий вузол запускається на декількох комп'ютерах, забезпечуючи стійкість до збоїв та високу доступність.

2.2.2 Компоненти керуючого вузла

Компоненти керуючого вузла приймають глобальні рішення щодо кластеру, також знаходять та опрацьовують події кластеру [8]. Компоненти керуючого вузла можуть запускатися на будь-якій машині у кластері. Проте для простоти скрипти запуску зазвичай запускають усі компоненти на одній машині і не запускають контейнери користувачів на цій машині. Діаграма компонентів керуючого вузла та їх взаємодія зображені на рисунку 2.2.

kube-apiserver — це API сервер, який є складовою керуючого вузла, що надає Kubernetes API. Цей API сервер виступає у ролі інтерфейсу для взаємодії для керуючого вузла Kubernetes. Головною частиною реалізації серверу Kubernetes API є kube-apiserver. Він створений для того, щоб масштабувати горизонтально, тобто він масштабує розгортаючи більше екземплярів. Також можна запустити декілька екземплярів kube-apiserver і налаштувати балансування трафіку між цими екземплярами.

etcd — консистентне та високо доступне сховище у форматі ключ-значення, що використовується як резервна копія Kubernetes для усіх даних кластеру.

kube-scheduler — компонент керуючого вузла, що спостерігає за нещодавно утвореними подами, які ще не назначені на жоден з вузлів, та вибирає вузол, де под буде запускатись. Рішення щодо планування приймаються з урахуванням таких факторів: індивідуальні і колективні вимоги до ресурсів, обмеження щодо апаратного / програмного забезпечення / політики, специфікації спорідненості та

антиспорідненості, локальність даних, перешкоди між робочим навантаженням та кінцеві строки.

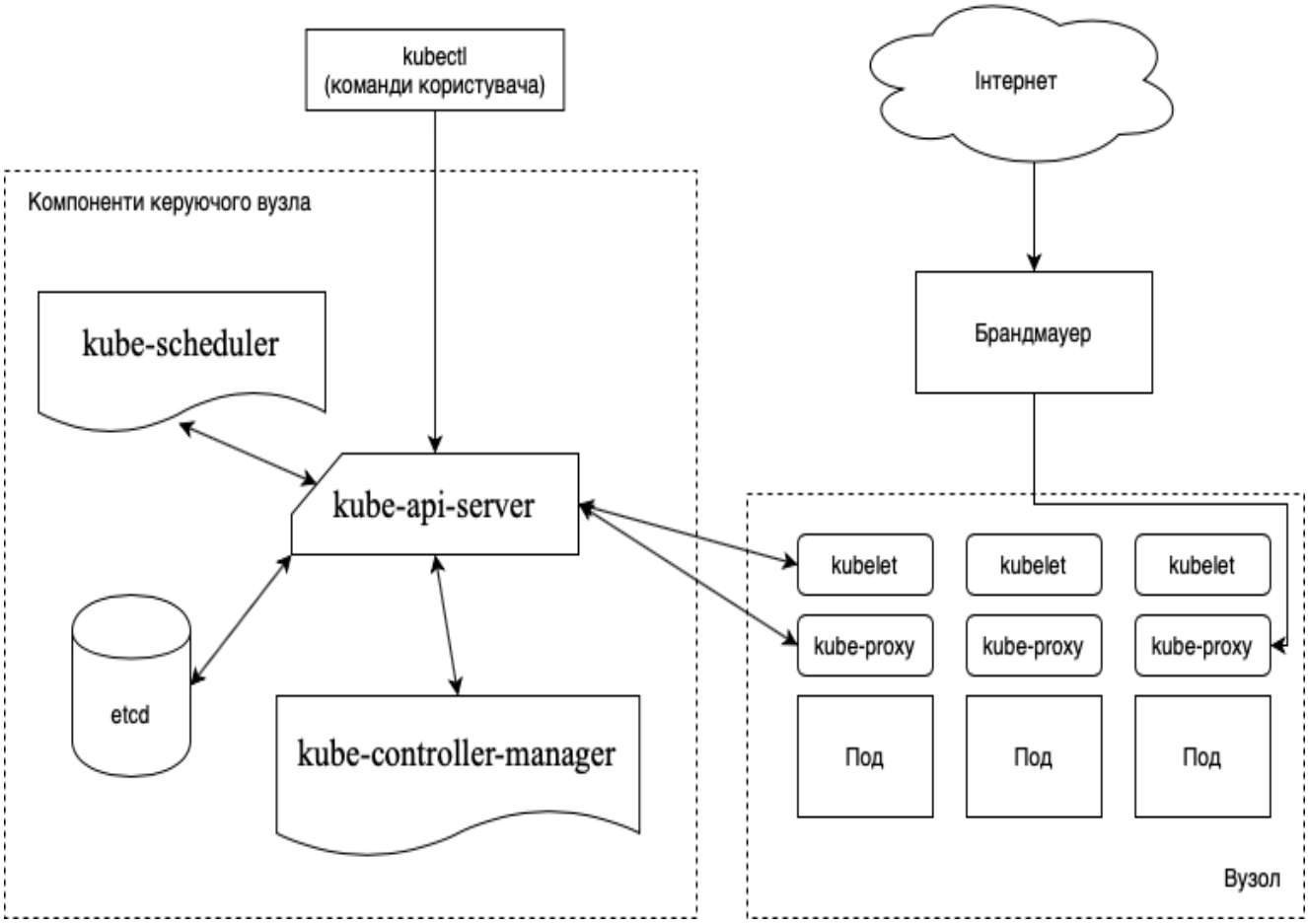


Рисунок 2.2 — Діаграма компонентів керуючого вузла

kube-controller-manager — компонент керуючого вузла, що запускає процеси контролера. Кожен контролер — це окремий процес, але для зменшення складності всі вони скомпоновані у єдиний двійковий файл та запускаються у одному процесі. Ці контролери включають: контролер вузлів, контролер реплікації, контролер кінцевих точок, контролер сервісних облікових записів і токенів.

2.2.3 Компоненти вузлів

Компоненти вузлів запускаються на кожному вузлі, підтримують запущені поди та забезпечують середовище виконання Kubernetes.

kubelet — агент, який запускається на кожному вузлі кластеру, він слідкує за тим, щоб контейнери знаходились у робочому стані у подах [8]. kubelet бере набір так званих PodSpecs, що надаються через декілька різних механізмів, і забезпечує, щоб поди описані у PodSpecs, були запущені та знаходились у здоровому стані.

kube-proxy — це мережевий проксі, який запускається на кожному вузлі кластеру, тим самим реалізуючи частину концепції Kubernetes Service. Він підтримує мережеві правила на вузлах. Ці правила дозволяють мережі спілкуватися з подами із сесіями мережі всередині або поза межами кластеру. kube-proxy використовує шар фільтрації пакетів операційної системи, якщо він один і доступний. В іншому випадку kube-proxy перенаправляє трафік до себе.

2.2.4 Addons

Addons (або доповнення) використовують ресурси Kubernetes (DaemonSet, Deployment тощо), щоб реалізувати властивості кластеру. Оскільки вони забезпечують властивості кластерного рівня, ресурси простору імен для доповнень належать до простору імен kube-system [8].

DNS (Domain Name System – комп’ютерна розподілена система для отримання інформації про домени) — в той час, коли інші доповнення не є безперечно необхідними, всі кластери Kubernetes повинні мати кластер DNS. Кластер DNS — це DNS сервер в додаток до інших DNS серверів у середовищі, який обслуговує DNS записи для сервісів Kubernetes. Контейнери, запущені Kubernetes, автоматично включать цей DNS сервер у свої DNS пошуки.

2.2.5 Контролери реплікації і набори реплік

Контролери реплікації (ReplicationController) і набори реплік (ReplicaSet) необхідні для управління групами подів [7], вибраних за допомогою селекторів міток. Вони гарантують, що визначена кількість екземплярів подів є завжди в

робочому стані. Головна відмінність між ними полягає у тому, що контролери реплікації перевіряють належність до групи по одному імені, а набори реплік дозволяють указувати декілька значень. Вважається, що краще надавати перевагу наборам реплік, адже контролери реплік — це їх підмножина.

Kubernetes гарантує, що кількість працюючих подів буде завжди відповідати значенню, що вказане в наборі реплік або в контролері реплікації. Кожен раз, коли кількість подів зменшується у зв’язку з якимось чинниками: проблемами з вузлами або ж самими подами, Kubernetes запускає нові екземпляри. В тому випадку, коли вручну було запущено більше подів, ніж указано у контролері реплікації або в наборі реплік, тоді Kubernetes сам видалить зайві поди.

2.2.6 Сервіси

Сервіси (Services в Kubernetes) використовуються для надання користувачам або іншим сервісам певних функцій [7]. Зазвичай вони складаються з групи подів, які ідентифікуються за допомогою міток. Сервіси можуть надавати доступ до зовнішніх ресурсів або екземплярів подів, якими можна керувати напряму на рівні віртуальних IP-адрес. В Kubernetes доступ до стандартних сервісів можна отримати за допомогою зручних кінцевих точок. Варто зазначити, що сервіси працюють на третьому мережевому рівні (TCP/UDP). Існує два механізми для публікування та пошуку сервісів: DNS та змінні середовища. Навантаження на сервіси здатна автоматично регулювати сама система Kubernetes, але якщо при цьому використовуються зовнішні ресурси або необхідне особливе ставлення, то розробники можуть виконувати балансування вручну.

2.2.7 Deployments

Deployment (або розгортання) надає можливість декларативного оновлення подів та наборів реплік [9]. Необхідно описати бажаний стан у розготці і контролер розгортання змінить поточний стан, щоб він відповідав бажаному. Можна визначити розгортання, щоб створити новий набір реплік, або щоб видалити існуюче розгортання і запозичити усі їх ресурси з новими розгортаннями.

2.2.8 Том

Місцеве сховище в подах — тимчасовий елемент, який видаляється разом з подом. Інколи цього достатньо, якщо потрібно лише передавати інформацію між контейнерами на одному вузлі. Але в певних випадках треба зберігати дані і після знищення пода або необхідно, щоб ці дані були доступними для декількох його екземплярів. Для цього передбачена концепція томів [7]. Існує багато видів томів або Volumes. Багато з них напряму підтримуються Kubernetes, але сучасний підхід передбачає додавання нових типів томів через інтерфейс CSI (Container Storage Interface — інтерфейс сховища контейнерів). Томи виду emptyDir підключаються до кожного контейнера і базуються на вмісті материнської системи. Також їх можна розмістити у пам’яті. Це сховище видаляється при знищенні пода.

2.2.9 Horizontal Pod Autoscaler

Horizontal Pod Autoscaler (або HPA) – ресурс Kubernetes, який реалізує механізм горизонтального масштабування подів. HPA може змінювати кількість екземплярів сервісів, якщо буде перевищено встановлений ліміт одного з заданих показників системи. HPA реалізований у вигляді нескінченого циклу, який періодично опитує задані показники системи на предмет їх перевищення

(рисунок 2.3).

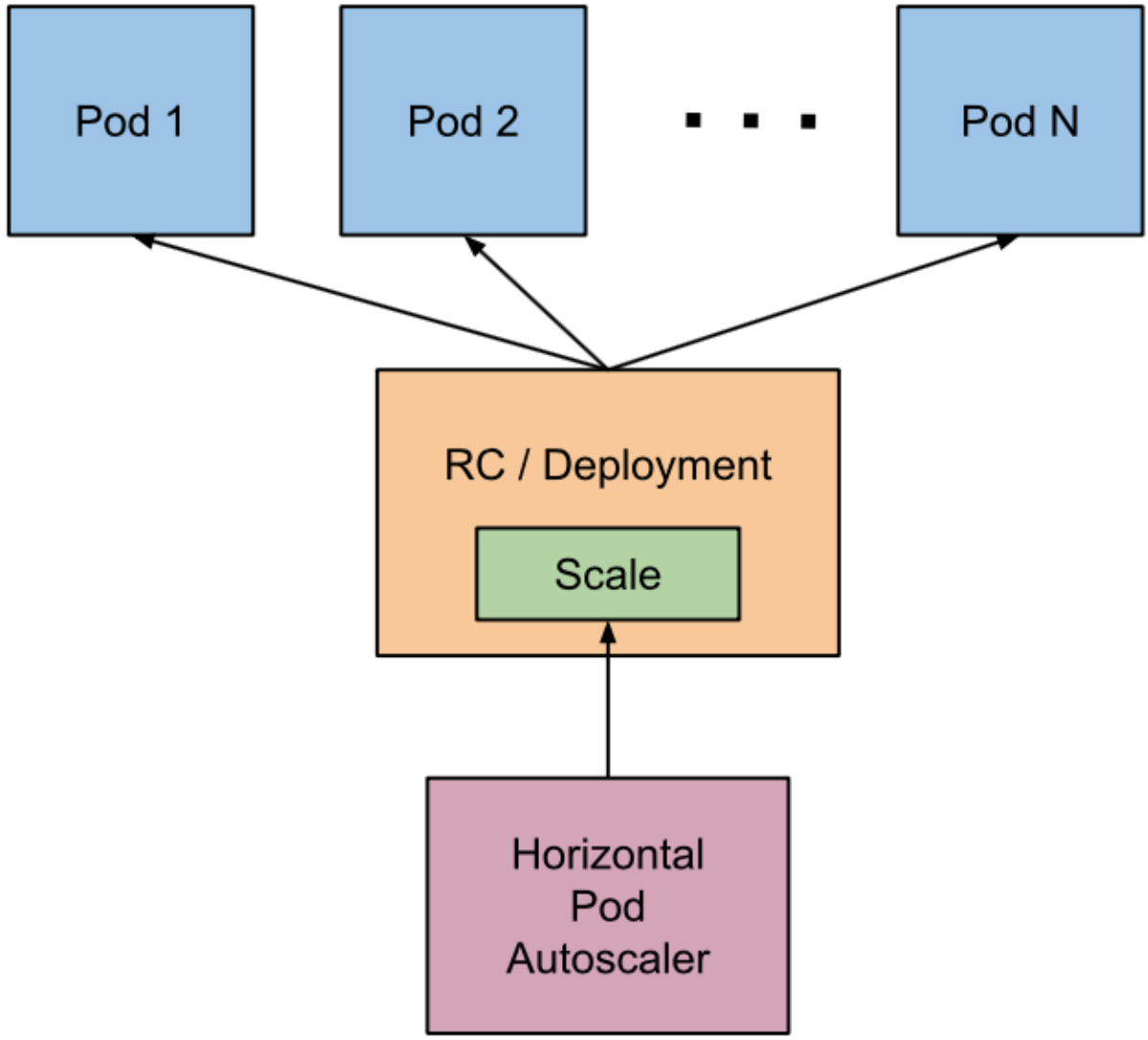


Рисунок 2.3 – Діаграма Horizontal Pod Autoscaler

HPA розраховує кількість необхідних екземплярів за формулою 2.1:

$$x = \left(c \times \left(\frac{m1}{m2}\right)\right), \tag{2.1}$$

де x — необхідна кількість екземплярів,

c — поточна кількість екземплярів,

m1 — поточне значення показника системи,

m2 — задане цільове значення показника системи.

Основна задача HPA визначити таку кількість екземплярів подів, яка забезпечить найближче поточне значення показника сервісу до цільового показника.

За замовчуванням Kubernetes підтримує автоматичне масштабування за показниками CPU або RAM, але цих показників не завжди достатньо. Іноді для більшої точності може знадобитися використовувати інші показники. Щоб надати таку можливість в Kubernetes 1.6 було додано підтримку використання користувальницьких показників у HPA.

Якщо глибше заглянути в те, як розроблений механізм автоматичного масштабування Kubernetes, виявиться, що HPA є не єдиною частиною цього механізму. HPA має отримувати показники ззовні, а компонентом, який відповідає за надання метрик HPA, є реєстр показників системи. Реєстр показників системи — це особлива частина кластеру, де будь-які показники надаються таким клієнтам, як HPA. API реєстру показників містить три окремих API:

— Resource Metrics API — забезпечує доступ до показники з подів (CPU та RAM);

— Custom Metrics API — забезпечує доступ до користувацьких показників, що відносяться до об’єктів Kubernetes;

— External Metrics API — забезпечує доступ до користувацьких показників, що не відносяться до об’єктів Kubernetes.

Кожен API показників вимагає відповідного сервера показників API, який потрібно налаштувати для викриття конкретних показників через API. Окрім сервера показників API, потрібен також колектор показників. Мета цього колектора — збирати конкретні показники з джерел та надавати їх на API-сервер показників (рисунок 2.4).

Для API різних показників можуть використовуватися різні колектори показників та різні API серверів показників. Стандартною конфігурацією API метричних ресурсів є cAdvisor як колектор показників, а сервер показників системи як офіційний сервер показників API.

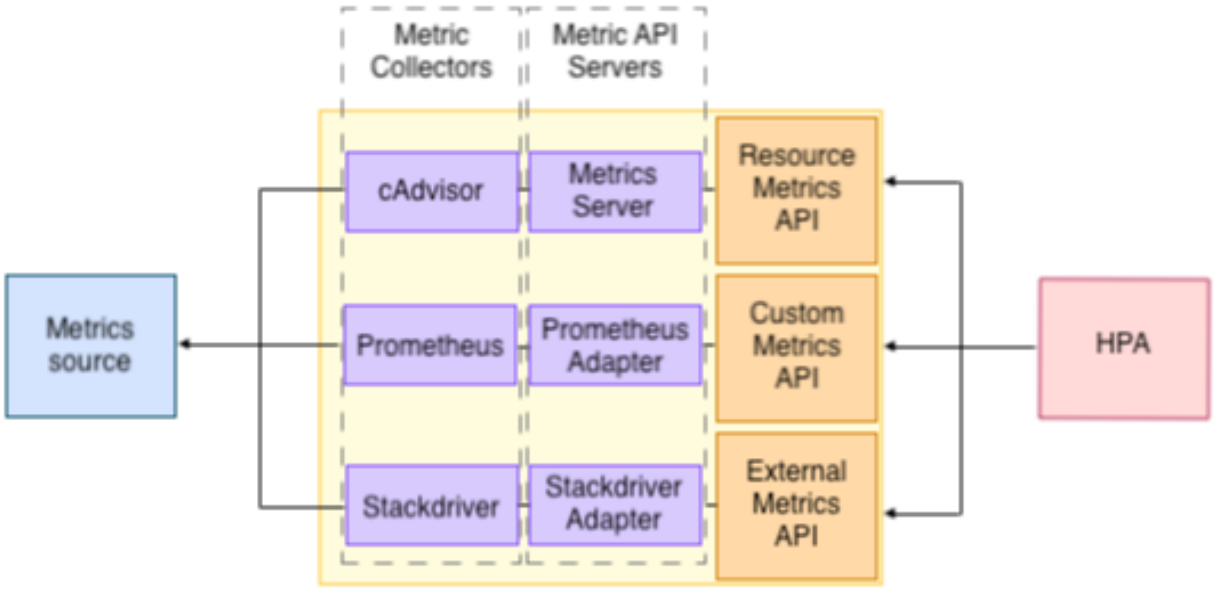


Рисунок 2.4 — Діаграма компонентів реєстру показників

Деякі з варіантів API для користувацьких та зовнішніх показників можуть бути Prometheus або Google Cloud Stackdriver як колектори показників та їх сервери показників API.

2.3 Спілкування виду видавець-підписник

Видавець-підписник — це шаблон спілкування у архітектурі програмного забезпечення [10], коли відправники повідомлень, видавці, не відправляють повідомлення напряму отримувачам, так званим підписникам, а натомість розподіляють опубліковані повідомлення по категоріями, не знаючи отримувача. Схожим чином, підписники також націлені на отримання повідомлень з однієї чи більше категорій, не знаючи видавця.

Шаблон видавець-підписник є спорідненим до шаблону черги повідомлень і зазвичай є складовою частиною системи проміжного програмного забезпечення орієнтованої на повідомлення. Більшість систем для обміну повідомленнями підтримують обидві моделі спілкування: чергу повідомлень та видавець-підписник.

У моделі видавець-підписник підписники зазвичай отримують лише підмножину з множини усіх виданих повідомлень. Процес вибору повідомлень для прийняття та обробки називається фільтрацією. Існує два види фільтрації: на основі теми та на основі вмісту повідомлення.

У системах на основі фільтрації по темі повідомлення публікуються до так званих “тем” або іменованих логічних каналів. Підписник у такій системі отримає всі повідомлення, що були опубліковані по темі, на яку вони підписані. Приклад такої системи зображений на рисунку 2.5. Також видавець відповідає за визначення тем, на які підписник може підписатися.

У системах з фільтрацією на основі вмісту повідомлень повідомлення надсилаються до підписника тільки у тому випадку, якщо атрибути або вміст цих повідомлень відповідають обмеженням, визначеним у підписнику. Підписник відповідає за класифікацію повідомлень.

Деякі системи підтримують змішану систему фільтрації: видавець публікує повідомлення до теми, в той час як підписник реєструє підписку на основі вмісту повідомлень до однієї чи більше тем.

У багатьох системах із моделлю спілкування видавець-підписник видавець надсилає повідомлення до посередника — брокера повідомлень або шини подій, а підписник реєструє підписки у того брокера, дозволяючи йому виконувати фільтрацію. Зазвичай брокер виконує функції зберігання на пересилки, щоб перенаправляти повідомлення від видавців до підписників. Також брокер може пріоритезувати повідомлення у черзі перед тим, як вони будуть надіслані далі.

Переваги шаблону видавець-підписник:

— слабке зв’язування — видавці слабко зв’язані з підписниками та навіть не знають про їх існування. Фокусуючись лише на темах, видавці та підписники можуть працювати абсолютно незалежно один від одного;

— масштабування — видавець-підписник забезпечує можливість кращого масштабування, ніж традиційні клієнт-серверні системи, шляхом паралельних операцій, кешування повідомлень, маршрутизації на основі дерев або мережі тощо. Проте видавець-підписник може втратити цю перевагу у системах з високим навантаженням.

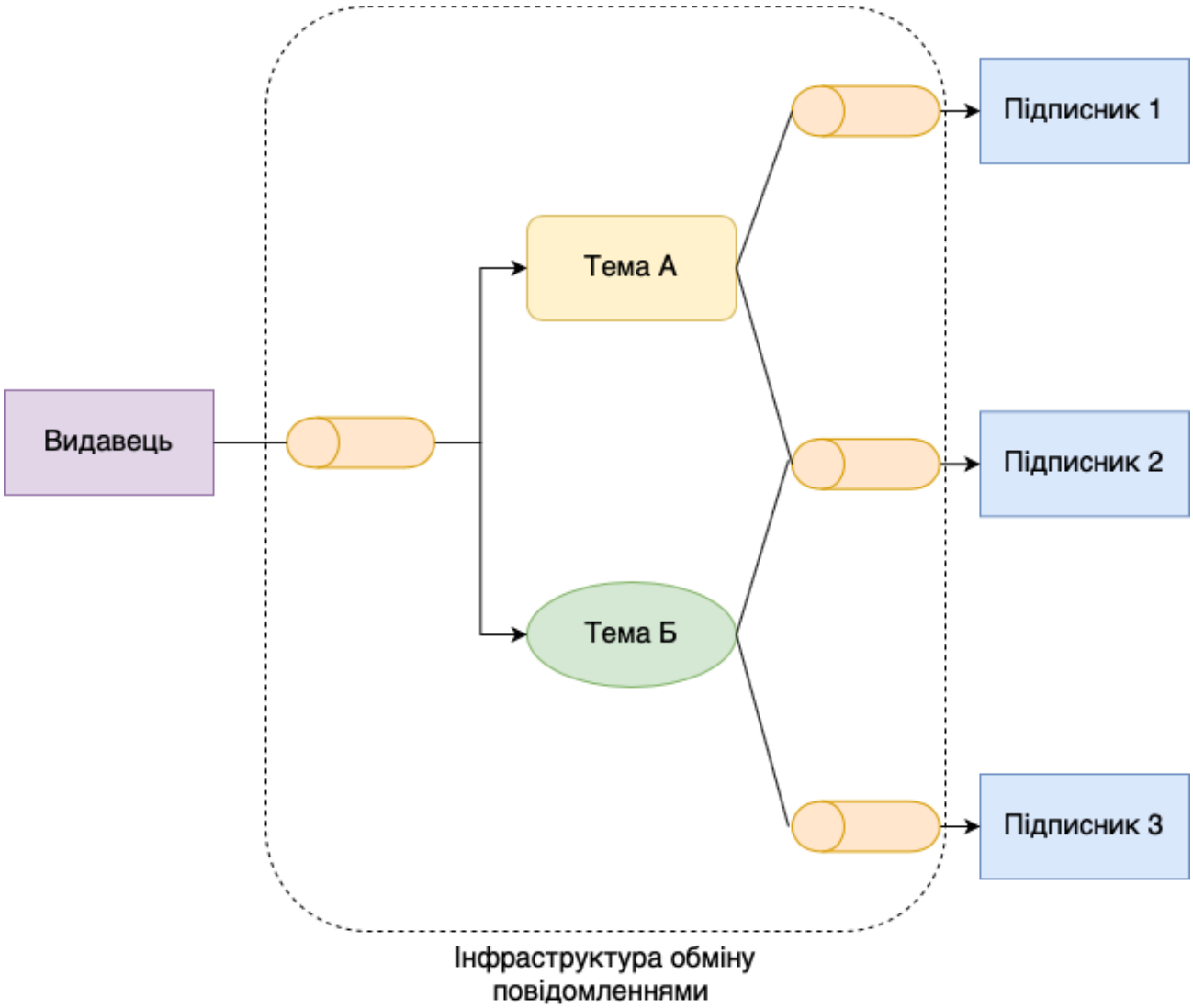


Рисунок 2.5 — діаграми системи з фільтрацією повідомлень по темі

Недоліки шаблону видавець-підписник:

— брокер у системах з видавцем-підписником може бути спроектований таким чином, щоб доставляти повідомлення протягом певного часу, а потім припинити спроби доставки, незважаючи на те, чи було отримане підтвердження про успішне отримання повідомлення усіма підписниками. Такі системи не можуть гарантувати стовідсоткову доставку повідомлень усім застосункам;

— при збільшенні кількості видавців та підписників, а також при зростанні кількості повідомлень зростає вірогідність нестабільності системи, що призводить до обмеження максимальної масштабованості системи з видавцем-підписником.

2.4 Висновки до розділу

У розділі 2.1 було описано стандартні підходи до побудови систем з мікросервісною архітектурою та компоненти, необхідні для реалізації таких систем.

Було описано переваги мікросервісної архітектури та її недоліки, було надано детальний огляд важливих функціональних компонентів таких систем. Для деяких компонентів було наведено різні підходи реалізації та перераховано переваги та недоліки кожного підходу.

Також у цьому розділі було наведено опис платформи, що відповідає за автоматизацію розгортання та масштабування мікросервісних систем, - Kubernetes. Було описано основні компоненти Kubernetes, їх взаємодію та функціональне призначення.

У кінці розділу йшлося про шаблон спілкування видавець-підписник, який може використовуватися для взаємодії компонентів системи.

3 ЗАСОБИ РЕАЛІЗАЦІЇ ЗАСТОСУНКІВ З ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

У цьому розділі будуть описані основні засоби реалізації застосунків з використанням мікросервісної архітектури для сервісів написаних мовою програмування Java. Буде наведено властивості та можливості кожного засобу, їх переваги та недоліки, а також буде виконано порівняння описаних підходів та наведена аргументація щодо вибору одного з них.

3.1 Аспекти реалізації застосунків з мікросервісною архітектурою

Розглянемо основний спектр проблем, пов’язаний із створенням мікросервісних архітектур, та подивимось, як Spring Cloud та Kubernetes вирішують ті чи інші проблеми. Мікросервісна архітектура гарна тим, що це архітектурний стиль з добре зрозумілими перевагами та компромісами. Мікросервіси забезпечують чіткі межі модулів, незалежне розгортання та різноманітність технологій. Але ціною цього є розробка розподілених систем та значні експлуатаційні витрати. Ключовим фактором успіху при створенні мікросервісної системи є зосередження уваги на виборі інструментів, які допоможуть вирішити якомога більше проблем мікросервісної архітектури. Швидкий і легкий процес запуску створення системи важливий, але шлях до фінального результату тривалий, і тому потрібно мати великий набір інструментів та рішень, щоб завершити створення системи успішно.

На рисунку 3.1 можна побачити важливі аспекти при розробці застосунків з мікросервісною архітектурою.

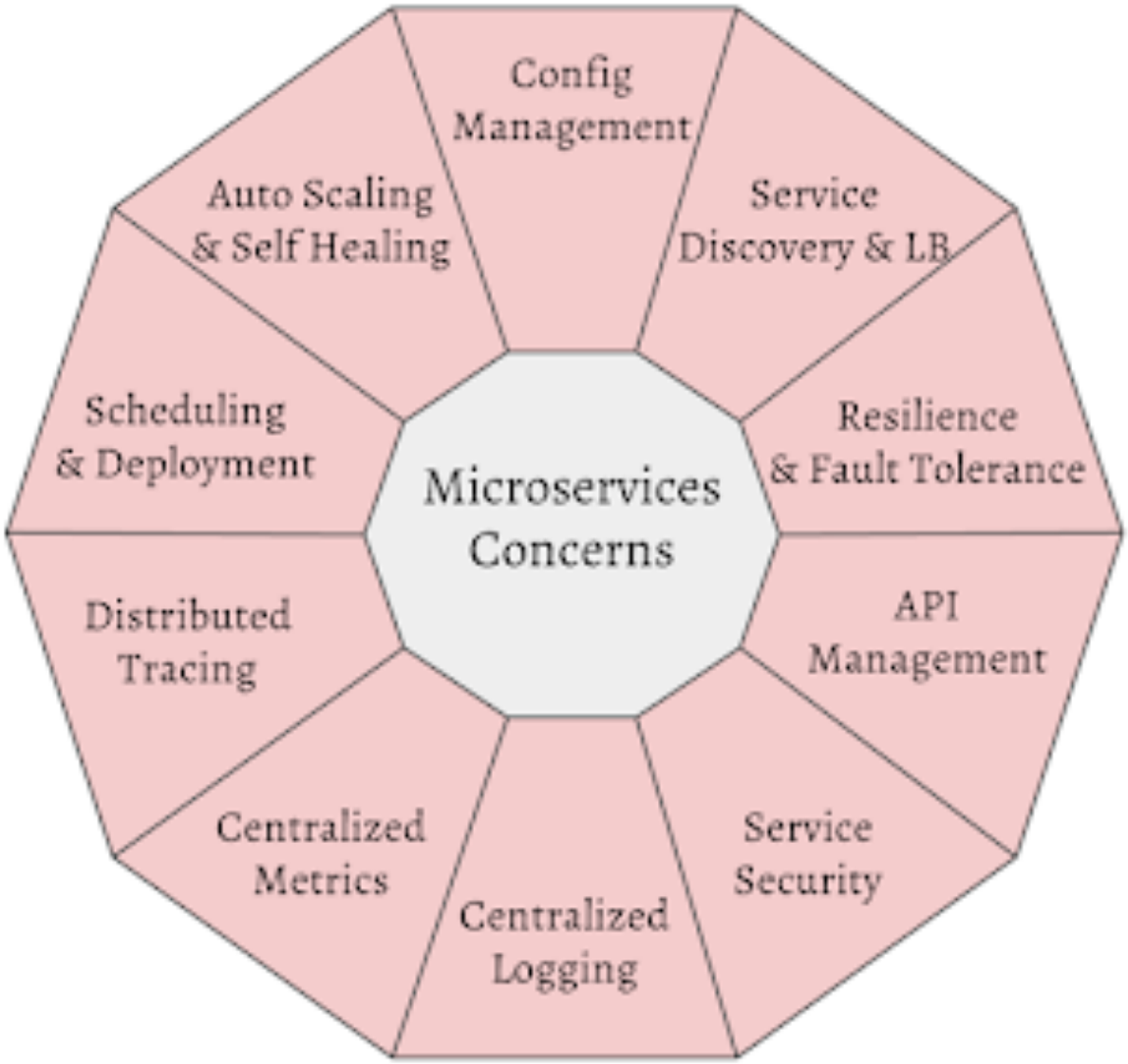


Рисунок 3.1 — Важливі аспекти для реалізації мікросервісних систем

На рисунку 3.1 можливо побачити список найпоширеніших технічних проблем, які мають бути вирішені при реалізації систем з мікросервісною архітектурою. На цьому рисунку не охоплено нетехнічні проблеми, такі як організаційна структура, культура тощо.

3.2 Spring Cloud як засіб реалізації застосунків з мікросервісною архітектурою

На сьогоднішній день існує два популярних засоби реалізації застосунків мовою Java на основі мікросервісної архітектури: Spring Cloud [11] та Kubernetes. Оскільки Kubernetes уже був описаний вище, приділимо більше уваги Spring Cloud.

Spring Cloud — це один із модулей Spring Framework, який дозволяє швидко будувати деякі поширені шаблони у розподілених системах. Spring Cloud має багатий набір інтегрованих бібліотек, версії яких узгоджені між собою і які забезпечують наступні функції: управління конфігураціями, знаходження сервісів, балансувальник навантаження, circuit breakers, маршрутизація тощо. Таким чином, Spring Cloud надає розробникам можливість більше сфокусуватися на бізнес-логіці застосунку, спрощуючи налаштування інфраструктури розподілених систем.

Побудова систем з використанням Spring Cloud має наступні переваги:

— уніфікована модель програмування, що забезпечується Spring Platform, і можливість швидкого створення застосунків використовуючи Spring Boot спрощують написання систем, а також зменшують час необхідний для початково налаштування проектів. Наприклад, за допомогою всього декілька анотацій можливо створити сервер для Service Discovery, ще декілька анотацій потрібні для налаштування клієнтів Service Discovery;

— багатий набір бібліотек, які дозволяють вирішити більшість проблем, що виникають під час виконання програми;

— різні бібліотеки Spring Cloud добре інтегровані одна з одною.

Недоліками Spring Cloud є:

— Spring Cloud обмежений тільки мовою розробки Java, що нівелює одну з переваг мікросервісних систем, а саме: заміна набору технологій, бібліотек або навіть мови написання, якщо це необхідно;

— на розробників та на самі застосунки покладається дуже багато відповідальності: кожен мікросервіс повинен підтримувати безліч клієнтів для отримання конфігурацій, знаходження локації сервіса та балансування навантаження. Такий підхід не приховує залежності часу будування проекту та часу виконання від середовища виконання;

— сам Spring Cloud покриває менший обсяг проблем мікросервісної архітектури. Розробникам необхідно потурбуватися про автоматичні розгортання, планування, управління ресурсами, ізоляцію процесів, самовідновлення тощо.

3.3 Kubernetes як засіб реалізації застосунків з мікросервісною архітектурою

Kubernetes може працювати з різними мовами, не обов’язково використовувати платформу Java. Kubernetes забезпечує наступні функції: управління конфігураціями, знаходження сервісів, балансування навантаження, відслідковування запитів, збір показників системи, планування робіт на рівні платформи та за межами застосунку. Застосунку не потрібні ніякі бібліотеки чи агенти на стороні клієнта, також застосунок може бути написаний будь-якою мовою.

Щодо переваг використання Kubernetes:

— Kubernetes підтримує багато мов та надає платформу управління контейнерами, що здатна запускати як хмарні, так і звичайні контейнеризовані застосунки. Сервіси, які надає платформа, такі як: управління конфігураціями, місцезнаходження сервісів, балансування навантаження, збирання показників системи, збирання та накопичення записів логування, можуть використовуватися різними мовами. Це дозволяє мати єдину платформу, яка буде використовуватися багатьма командами розробників і слугуватиме багатьом цілям;

— порівняно із Spring Cloud Kubernetes покриває більший спектр проблем застосунків з мікросервісною архітектурою;

— Kubernetes має велику спільноту, яка є однією з найактивніших спільнот відкритого коду на GitHub.

Проте Kubernetes має і свої недоліки:

— оскільки Kubernetes здатний працювати з багатьма мовами, то він не оптимізований для роботи з кожною платформою у тому ж ступені, як Spring Cloud для JVM;

— платформа Kubernetes більше спрямована для використання людьми із спеціальністю DevOps, ніж розробниками, тому його використання потребує вивчення нових концепцій;

— Kubernetes порівняно нова платформа, яка все же активно розробляється та росте, а тому можуть з’являтися нові функції, з якими буде складно працювати.

3.4 Порівняльна характеристика наведених засобів

На рисунку 3.2 зображена порівняльна таблиця [12], які вимоги мікросервісної архітектури можуть задовольнити Spring Cloud та Kubernetes. Як бачимо на рисунку 3.2, Kubernetes має більше вбудованих функцій, необхідних для побудови систем з мікросервісною архітектурою. До таких функцій належать: масштабування та самовідновлення, розгортання застосунків та планування, ізоляція процесів, управління середовищем та ресурсами.

Підсумовуючи усе наведене раніше, для виконання дипломної роботи була вибрана реалізація системи з мікросервісною архітектурою на основі Kubernetes, тому що він пропонує вирішення ширшого спектру проблем, а також дозволяє позбавити самі сервіси частини логіки необхідної для інфраструктурних задач, тим самим розмежовуючи відповідальність компонентів системи: сервіси відповідають за логіку застосунку, а Kubernetes відповідає за всі інфраструктурні питання.

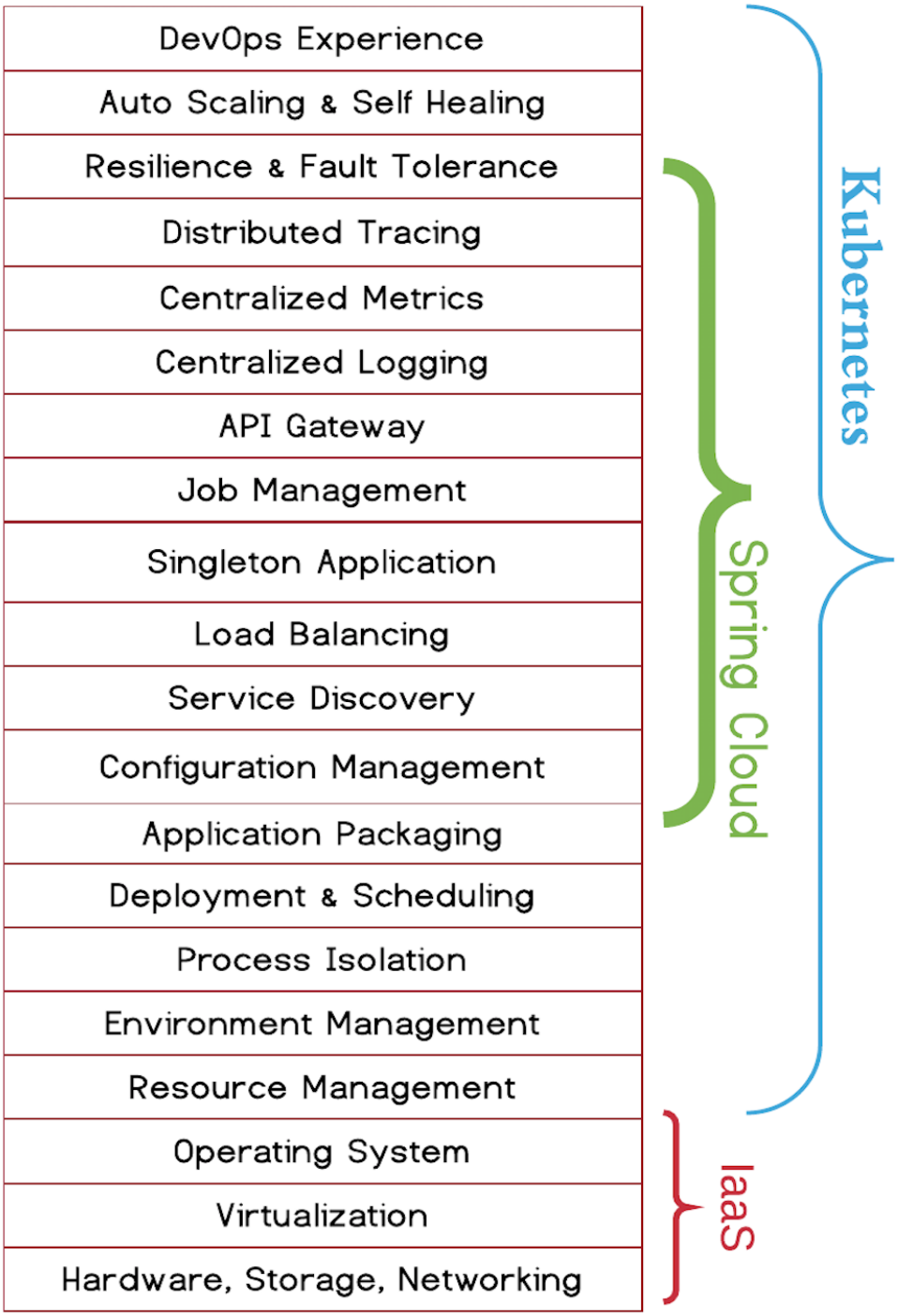


Рисунок 3.2 — Порівняльна таблиця Spring Cloud та Kubernetes

3.5 Висновки до розділу

У даному розділі були описані основні проблеми, які потрібно враховувати при розробці систем з мікросервісною архітектурою, та описані засоби, які

допомагають вирішити ці питання при реалізації систем із сервісами написаними мовою програмування Java. Найпоширеніші засоби є Spring Cloud та Kubernetes. Буде наведено основні властивості та можливості кожного засобу, їх переваги та недоліки, а також було виконано порівняння описаних засобів та наведена аргументація щодо вибору одного з них.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТЕСТОВОГО ЗАСТОСУНКА З ВИКОРИСТАННЯМ АСИНХРОННОЇ ВЗАЄМОДІЇ НА ОСНОВІ PUB/SUB У ХМАРНОМУ KUBERNETES КЛАСТЕРІ

Даний розділ містить опис реалізації системи: опис створених сервісів для виконання логіки застосунку, наведені їх діаграми класів, описано їх взаємодію з іншими компонентами системи. Також надана інформація про використовувані бібліотеки та фреймворки для написання сервісів. Викладено алгоритм налаштування системи та окремих її компонентів. Продемонстровано приклади роботи створеної системи.

4.1 Опис класів застосунків

Система складається з двох застосунків — сервіса-видавця та сервіса-підписника.

Діаграма класів сервіса-видавця зображена на рисунку 4.1. Сервіс складається з 5 класів.

1. Клас PublishingApp — точка входу в роботу сервіса, головний клас, в якому піднімається контекст застосунка Spring.

2. Клас PublishConfig — відповідальний за налаштування інтеграції сервіса з Google Cloud Pub/Sub. За допомогою шару абстракції наданого Spring Integration цей клас створює канал зв’язку, а також визначає тему створеного видавця, куди будуть надсилатися повідомлення у Cloud Pub/Sub.

3. Клас Controller — представляє собою шар контролю доступу до застосунка. Цей клас відповідає за взаємодію застосунка з користувачем або іншими системами, приймає від користувачів запити-повідомлення за

допомогою технології REST, виконує їх первинну перевірку та передає повідомлення далі для обробки.

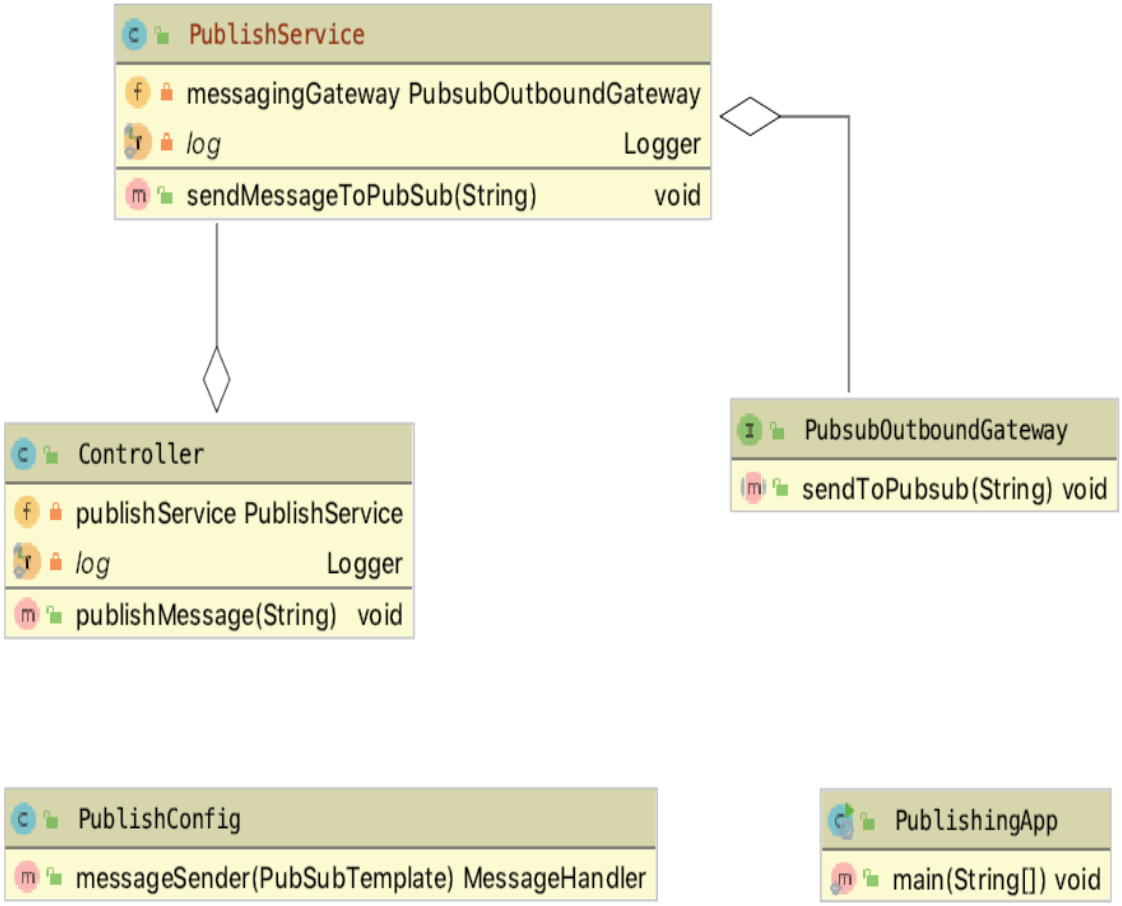


Рис. 4.1 — Діаграма класів сервіса-видавця

4. Клас PublishingService — відповідальний за надсилання повідомлення до Google Cloud Pub/Sub використовуючи PubsubOutboundGateway. Реєструє, що сервіс отримав повідомлення та надіслав його до Pub/Sub.

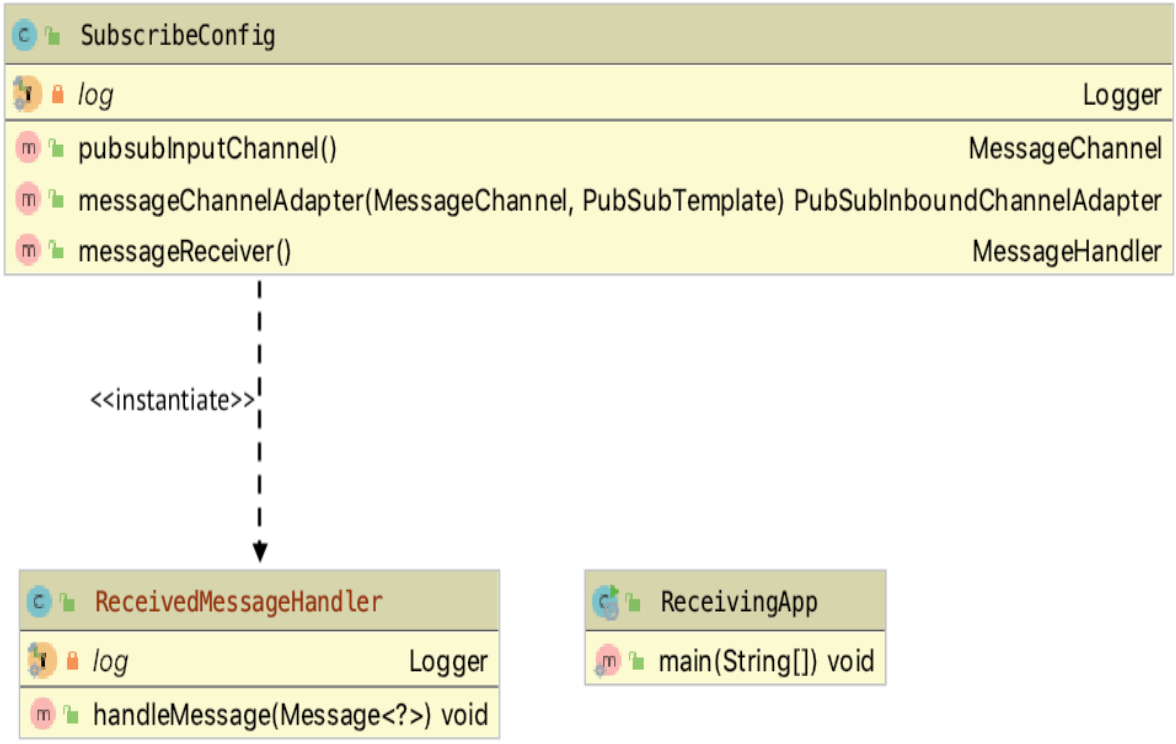
5. Інтерфейс PubsubOutboundGateway — спеціальний інтерфейс, який забезпечує роботу із Spring Integration. Дозволяє використовувати шлюз повідомлень для заданого каналу зв'язку — pubsubOutputChannel.

Діаграма класів сервіса-підписника зображена на рисунку 4.2. Сервіс містить 3 класи.

1. Клас ReceivingApp — точка входу в роботу сервіса, головний клас, в

якому піднімається контекст застосунка Spring.

2. Клас SubscribeConfig — відповідальний за налаштування каналу зв’язку із Pub/Sub. У цьому класі створюється канал зв’язку, який буде використовуватись Spring Integration, описується адаптер, який виступає посередником між Spring Integration та Google Cloud Pub/Sub. Також SubscribeConfig визначає загальну кінцеву точку для з’єднання сервісу з системою обміну повідомлень — ServiceActivator, який реагує на вхідне повідомлення та спонує подальшу його обробку шляхом виклику ReceivedMessageHandler.



4.2 — Діаграма класів сервіса-підписника

3. Клас ReceivedMessageHandler — відповідальний за обробку вхідного повідомлення. Реєструє, що сервіс отримав повідомлення.

4.2 Опис бібліотек та фреймворків, використаних у системі

Для реалізації програмного продукту використовувались наступні бібліотеки/модулі/фреймворки: Spring Framework, Spring Boot, Spring Web MVC, Spring Integration, Spring Cloud GCP.

Spring Boot — модуль Spring, який забезпечує швидке створення застосунків з нуля, надаючи певний набір конфігурацій за замовчуванням [13]. Spring Boot пропонує стартери — готові пакети зв’язних бібліотек із сумісними між собою версіями для конкретних цілей (веб, тестування, підключення бази даних тощо), а також надає вбудований сервер Tomcat.

Spring Web MVC — модуль Spring, що дозволяє створювати архітектурний шаблон Model — View — Controller за допомогою слабко зв’язаних готових компонентів [14]. Необхідним для дипломного продукту елементом модуля є RestController, який забезпечує користувачу можливість взаємодіяти із застосунком через REST.

Spring Integration — один із модулів Spring, який робить можливим полегшений обмін повідомленнями серед застосунків на основі Spring, а також підтримує інтеграцію з зовнішніми системами через декларативні адаптери [15]. Ці адаптери надають високорівневу абстракцію над підтримкою Spring віддалених викликів, обміном повідомлень та плануванням. Spring Integration пропонує наступні переваги:

— робить майже повністю незв’язними системи, що приймають участь у інтеграції;

— дозволяє учасникам інтеграції бути повністю незалежними від використовуваних протоколів, форматування або інших деталей реалізації інших учасників;

— сприяє розробці та повторному використанню компонентів, залучених до інтеграції.

Spring Cloud GCP — один із модулів Spring, який надає великий набір бібліотек для роботи з Google Cloud Platform, а також спрощує взаємодію Spring

застосунків з GCP [16]. Spring Cloud GCP забезпечує ще один рівень абстракції, надаючи користувачам можливість використовувати Google Cloud Pub/Sub та не залежати від конкретного Google Cloud Pub/Sub API. Spring Cloud GCP дозволяє публікувати або підписуватись на теми Google Cloud Pub/Sub, а також створювати, перелічувати та видаляти теми та підписки. Spring Boot стартер надає багато автоматичних конфігурацій за замовчуванням для різних компонентів Pub/Sub.

4.3 Опис системи

Загальна архітектура системи зображена на рисунку 4.3.

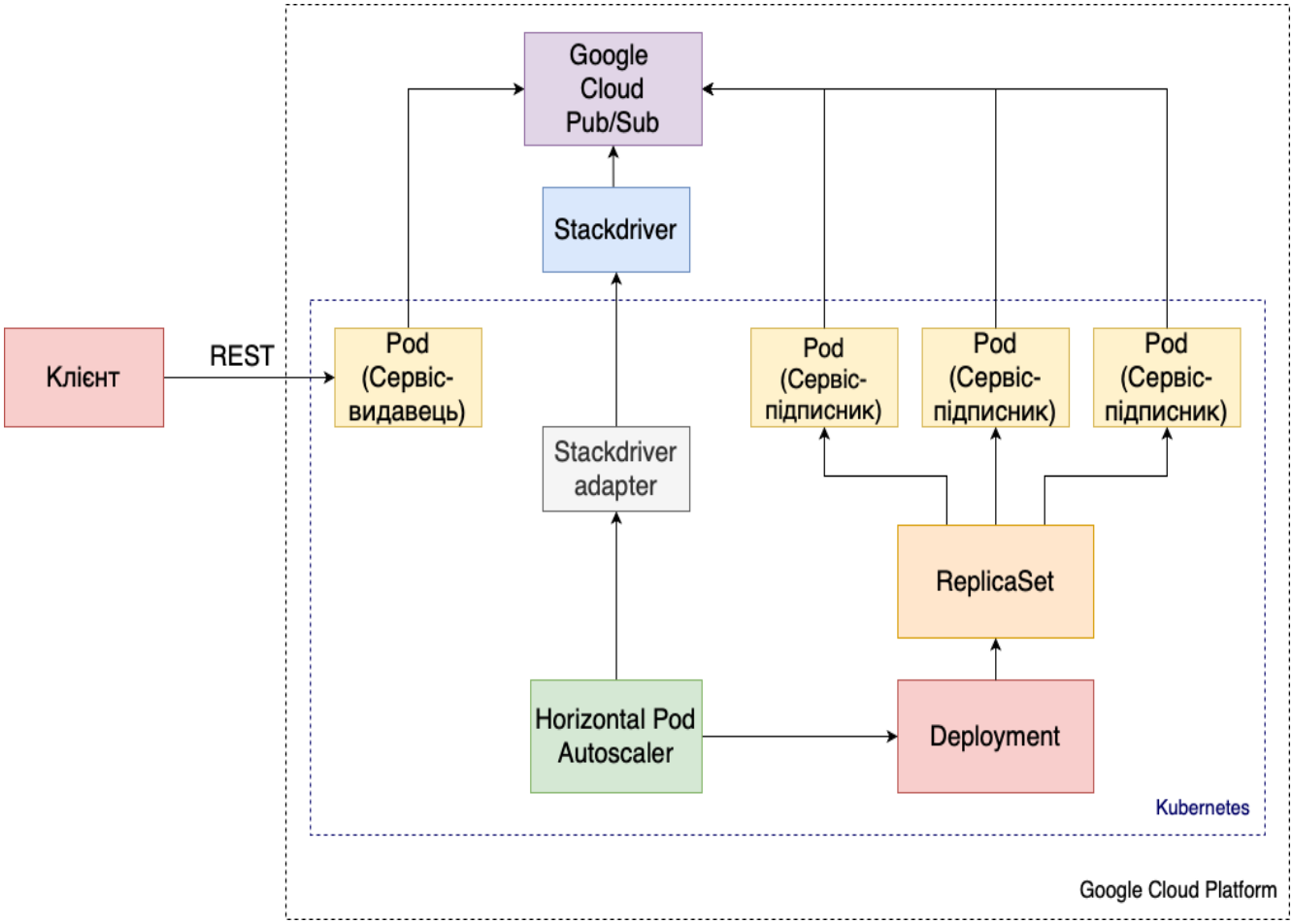


Рис 4.3 — Діаграма архітектури системи

Сервіс-видавець — сервіс описаний у розділі 4.1, відповідає за отримання повідомлень з-поза меж системи та пересилання їх до Google Cloud Pub/Sub. У

системі з використанням Kubernetes цей сервіс контейнеризується та розгортається як робоча одиниця Kubernetes — под.

Сервіс-підписник — сервіс описаний у розділі 4.1, відповідає за отримання повідомлень з Pub/Sub та їх обробку. У системі з використанням Kubernetes ці сервіси контейнеризуються та розгортаються як робоча одиниця Kubernetes — под.

Google Cloud Pub/Sub — рішення надане GCP, сервіс для асинхронного обміну повідомленнями, який виступає у даній роботі у ролі черги. Головні концепції Google Cloud Pub/Sub — це тема, підписка та повідомлення. Підписка у Pub/Sub налаштована у режимі “Pull”, тобто підписник відповідає за опитування та діставання повідомлень з черги.

Stackdriver – сервіс для збирання та збереження показників системи, що надається GCP.

Custom Metrics Stackdriver Adapter — адаптер, необхідний для налаштування взаємодії ресурсів Kubernetes, які відповідальні за збирання показників системи, із сервісом Stackdriver. Дозволяє ресурсам Kubernetes використовувати показники, надані Stackdriver, і на їх основі виконувати необхідні операції, такі як горизонтальне масштабування.

Horizontal Pod Autoscaler — ресурс Kubernetes, який реалізує механізм горизонтального масштабування сервісів. Відповідає за прийняття рішення про зменшення або збільшення кількості екземплярів сервісів та надсилає відповідну команду на Deployment.

Deployment — містить у собі певний бажаний стан подів та контролює його підтримання. При створенні Deployment неявно створюється і ReplicaSet, який належить Deployment. Deployment делегує команди щодо збільшення чи зменшення кількості екземплярів ReplicaSet.

ReplicaSet — безпосередньо відповідає за підтримання певної кількості подів.

Kubernetes — платформа для автоматичного розгортання, масштабування та управління контейнеризованними застосунками. Kubernetes відповідає за

вирішення більшості інфраструктурних питань даної роботи. Надає наступні ресурси, використані при створенні системи: Pods, Deployment, ReplicaSet, HPA.

Google Cloud Platform — набір хмарних сервісів, платформа для розгортання створеної системи.

4.4 Опис налаштування системи

Для того, щоб розгорнути систему у GCP, спочатку потрібно створити Docker image сервісів, описаних у розділі 4.1, після цього завантажити створені зображення до Google Container Registry - місце для збереження та управління Docker зображеннями, що надається GCP.

Також необхідно створити topic та subscription в Google Pub/Sub для використання черги.

Для використання Kubernetes, потрібно створити кластер на платформі GCP.

Наступним кроком потрібно описати налаштування розгортання кожного сервіса у спеціальному ресурсі Kubernetes — Deployment, а потім створити цей ресурс у GCP. Базуючись на даному описі Deployment, Kubernetes буде створювати поди, у яких буде міститись сервіс-видавець або сервіс-підписник, а також контролювати збереження бажаного стану подів, описаних у Deployment.

Щоб застосувати горизонтальне масштабування в Kubernetes, необхідно розгорнути Custom Metrics Stackdriver Adapter, щоб надати доступ Google Kubernetes Engine до показників системи у Stackdriver. Для запуску Custom Metrics Stackdriver Adapter потрібно надати користувачеві дозволи для створення необхідних ролей авторизації. Після цього потрібно розгорнути адаптер до кластеру.

Тепер необхідні показники можна збирати з Pub/Sub за допомогою колектора метрик, наданого Google — Stackdriver, і тоді ці показники можна отримати у Google Kubernetes Engine та Horizontal Pod Autoscaler, зокрема, через адаптер Stackdriver.

Для автоматичного масштабування потрібно розгорнути ресурс HPA в Kubernetes і вказати наступні параметри:

— мінімальна та максимальна кількість екземплярів;

— назва метрики для масштабування;

— цільове середнє значення для показника масштабування;

— ресурс для масштабування.

На рисунку 4.4 описані налаштування об’єкту HPA, який використовується у запропонованій системі.

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: pub-sub-hpa
spec:
  minReplicas: 1
  maxReplicas: 4
  metrics:
  - external:
      metricName: pubsub.googleapis.com|subscription|num_undelivered_messages
      metricSelector:
        matchLabels:
          resource.labels.subscription_id: testSubscription
      targetAverageValue: "2"
    type: External
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sub-app
```

Рисунок 4.4 — Налаштування об’єкту HPA для масштабування

Цей приклад визначає мінімальну та максимальну кількість екземплярів сервісів, а також середнє значення показника "кількість неопрацьованих повідомлень", яке буде використовуватись і при масштабуванні для збільшення екземплярів, і для масштабування для зменшення екземплярів.

Після створення HPA кількість екземплярів сервісу повинна дорівнювати мінімальному значенню, яке описане у файлі конфігурації. У даному випадку мінімальне значення дорівнює 1. Для отримання більш детальної інформації про

стан HPA, дізнатися, які події викликали масштабування, необхідно виконати команду kubectl describe hpa.

Якщо створити навантаження на систему, можна побачити, що кількість екземплярів сервісів зросла. Це означає, що автоматичне масштабування Kubernetes працює.

4.5 Приклад роботи системи

Для демонстрації роботи системи потрібно створити певне навантаження на систему шляхом надсилання великої кількості запитів за невеликий проміжок часу. У наступному розділі буде більш детально описано проведе тестування системи навантаженням, а у цьому розділі буде продемонстровано результат створеного навантаження та механізм горизонтального масштабування у дії.

Початковий стан системи, а саме кількість сервісів-підписників, до навантаження зображено на рисунку 4.5. На рисунку 4.6 зображено кількість сервісів-підписників, що були створені під дією навантаження.

Managed pods

Revision	Name	Status	Restarts	Created on
1	sub-app-59949f4b7b-sfjmm	Running	0	May 31, 2020, 2:48:16 PM

Рисунок 4.5 — Кількість екземплярів сервісів підписників перед створеним навантаженням у системі з автоматичним масштабуванням

Managed pods

Revision	Name	Status	Restarts	Created on
1	sub-app-59949f4b7b-sfjmm	Running	0	May 31, 2020, 2:48:16 PM
1	sub-app-59949f4b7b-fqnvr	Running	0	May 31, 2020, 7:02:48 PM
1	sub-app-59949f4b7b-7f67r	Running	0	May 31, 2020, 7:02:48 PM
1	sub-app-59949f4b7b-2wc2z	Running	0	May 31, 2020, 7:02:48 PM

Рисунок 4.6 — Кількість екземплярів сервісів підписників після створеного навантаження в системі з автоматичним масштабуванням

Як видно з рисунку 4.6, кількість екземплярів сервісів-підписників під час навантаження збільшилася до максимально можливої — 4 екземпляри, саме так, як це було сконфігуровано.

На рисунку 4.7 зображено результат команди kubectl describe hpa, де описано, коли та які саме події викликали масштабування сервісу, до якої кількості екземплярів.

```
Name:                pub-sub-hpa
Namespace:           default
Labels:              <none>
CreationTimestamp:   Sun, 31 May 2020 18:52:20 +0300
Reference:           Deployment/sub-app
Min replicas:        1
Max replicas:        4
Deployment pods:     1 current / 1 desired
Events:
  Type     Reason                        Age                From
Message
  ----     ------                        ----               ----
-------
  Normal   SuccessfulRescale             11m                horizontal-pod-autoscaler
New size: 4; reason: external metric pubsub.googleapis.com|subscription|num_undelivered_
messages(&LabelSelector{MatchLabels:map[string]string{resource.labels.subscription_id: t
estSubscription,},MatchExpressions:[],}) above target
  Normal   SuccessfulRescale             3m1s               horizontal-pod-autoscaler
New size: 1; reason: All metrics below target
```

Рис. 4.7 — Вивід у консоль опису поточного стану об’єкту HPA

Через певний проміжок часу — 5 хвилин за замовчуванням — за відсутності навантаження кількість екземплярів знову зменшилась до мінімальної.

4.6 Висновки до розділу

У цьому розділі було наведено детальний опис створеної системи. Були подані аспекти програмної реалізації сервісів, наведено їх діаграми класів та описано спосіб взаємодії. Також було описано бібліотеки та фреймворки, що використовувалися при створенні сервісів, а саме: Spring Boot, Spring Integration, Spring Cloud GCP, і вказано, яку функцію у системі вони виконують. Після цього було наведено опис всієї системи разом із компонентами платформи Kubernetes, описані особливості їх налаштування. У кінці було продемонстровано роботу системи.

5 ОБЧИСЛЮВАЛЬНИЙ ЕКСПЕРИМЕНТ З БАЛАНСУВАННЯМ НАВАНТАЖЕННЯ НА ОСНОВІ ГОРИЗОНТАЛЬНОГО МАСШТАБУВАННЯ

У даному розділі буде описано інструмент, за допомогою якого буде виконуватися генерація його навантаження, буде розглянуто його компоненти, буде наведений приклад налаштування. Також буде описано обчислювальний експеримент, його вхідні параметри та результати. Буде виконано аналіз отриманих даних.

5.1 Інструмент для генерації навантаження

Для перевірки ефективності та надійності створеної системи було проведено тестування навантаженням. У якості інструменту для генерації великої кількості запитів було використано JMeter — програмне забезпечення із відкритим програмним кодом, яке розроблюється Apache Software Foundation . На рисунку 5.1 зображено інтерфейс програми JMeter, а також налаштування, з якими він запускався.

Було створено спеціальний сценарій для виконання засобами JMeter, який складається із наступних елементів:

— лічильник — “Counter”;

— користувацькі змінні — “ User Defined Variables”;

— група потоків — “Diploma”;

— зразок http запиту — “Post a message”;

— таймер — “Sleep”;

— слухач запитів — “View Results Tree”.

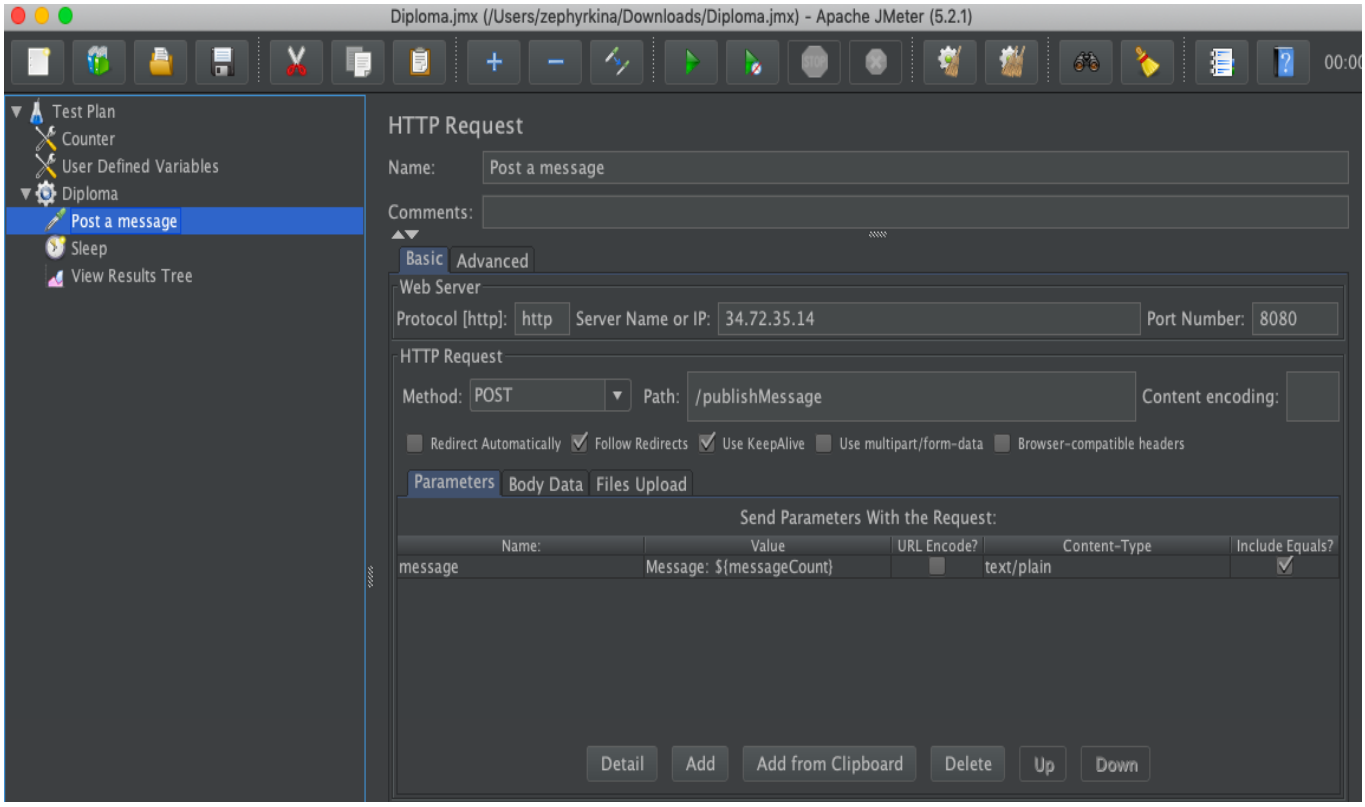


Рис. 5.1 — Інтерфейс JMeter

На рисунку ми бачимо наступні параметри запуску сценарію: використовуючи HTTP-метод POST на ip-адресу 34.72.35.14 (зовнішня ip-адреса сервісу-видавця), порт 8080, за відносним шляхом “/publishMessage” надсилається запит, який містить єдиний HTTP-параметр — “message”. Значення цього параметру з кожним запитом дорівнює слову ”Message” та значенню лічильника, який інкрементується з кожним запитом. Після виконання запиту спрацьовує таймер, який зупиняє виконання потоку на зазначений проміжок часу. Останнім кроком сценарію є слухач запитів, який реєструє дані запиту та дані відповіді, такі як: код відповіді, час виконання запиту, тіло відповіді тощо.

5.2 Результати експерименту

Тестування виконувалося на двох реалізаціях систем: з механізмом горизонтального масштабування та без. Система без масштабування має архітектуру зображену на рисунку 5.2. Архітектура створеної системи із

масштабуванням наведена на рисунку 4.3 у розділі 4. Ці системи відрізняються тим, що у системи на рисунку 5.2 відсутня низка компонентів, необхідних для виконання горизонтального масштабування системи.

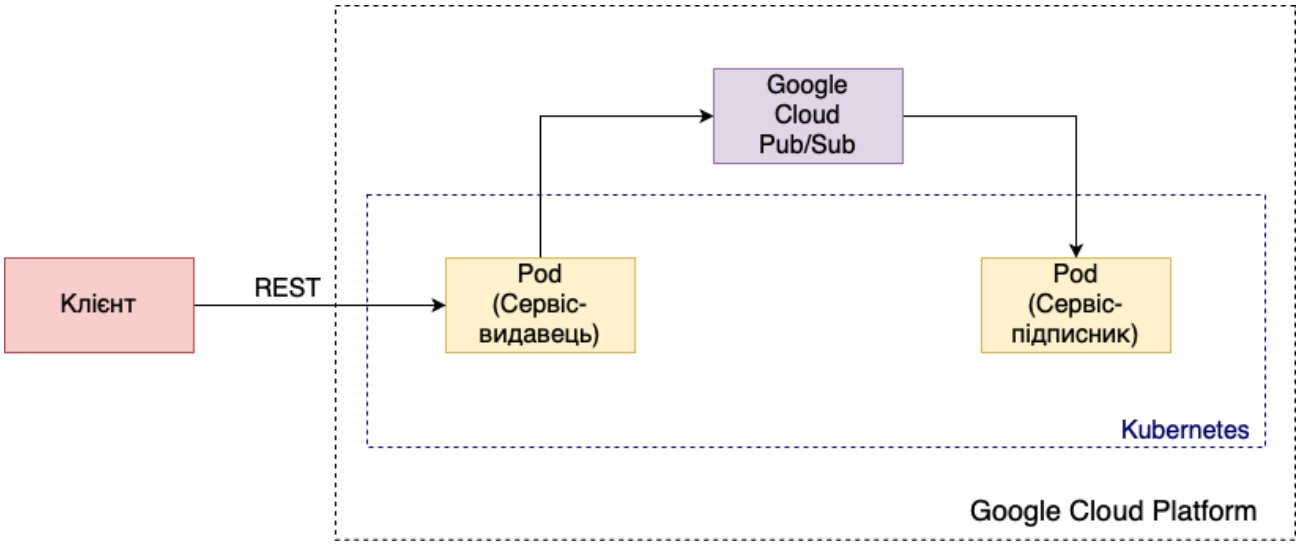


Рисунок 5.2 — діаграма архітектури системи без масштабування

На сервіс-видавець було надіслано 500 запитів із затримкою 500 мс для кожного тестового випадку.

За результатами тестів було отримано наступне. Було порівняно часові позначки, коли перше та останнє повідомлення публікувалось сервісом-видавцем та коли перші та останні повідомлення оброблялися сервісом-підписником, припускаючи, що на відправлення запитів у сервісі було витрачено відносно малий час, а на обробку кожного запиту виділилося 5 секунд для імітації реальних виробничих умов. Всі часові позначки отримані у ході експерименту наведені у таблиці 5.1.

Згідно з таблицею 5.1 затримка між публікацією останнього повідомлення та обробкою його сервісом підписником в системі без автоматичного масштабування становила 7 хвилин 14 секунд, тоді як у системі з автоматичним масштабуванням ця затримка становила 5 секунд.

Таблиця 5.1 — Отримані результати обробки запитів із автоматичним масштабуванням та без нього

	Без автомасштабування		З автомасштабуванням	
	Видавець	Підписник	Видавець	Підписник
Час першого запиту (xx:cc)	00:00	00:05	00:00	00:06
Час останнього запиту (xx:cc)	05:57	13:11	05:57	06:02

На рисунках 5.3 та 5.4 наведені графіки, що були отримані із дошки для моніторингу підписом у Google Cloud Pub/Sub. На графіках зображена динаміка кількості повідомлень у черзі під час тестування для систем без масштабуванням та з відповідно.

Як видно з даних рисунків, у системі без масштабування накопичувалося більше неопрацьованих повідомлень за певний проміжок часу, а також можливо візуально оцінити, наскільки більше часу система без масштабування опрацьовувала задану кількість повідомлень.

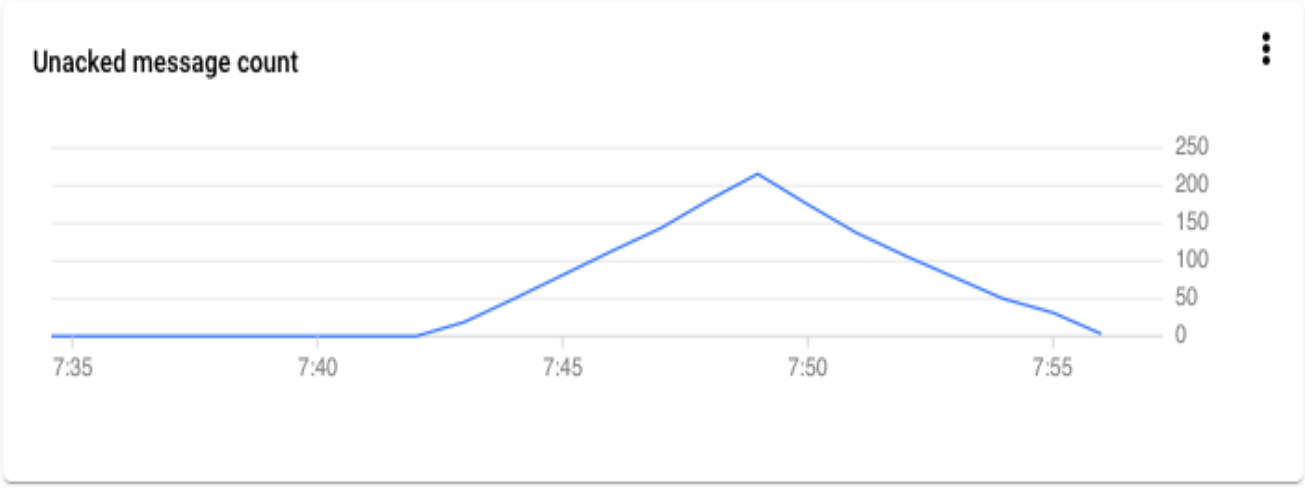


Рисунок 5.3 — Кількість повідомлень у черзі під час тестування у системі без масштабування

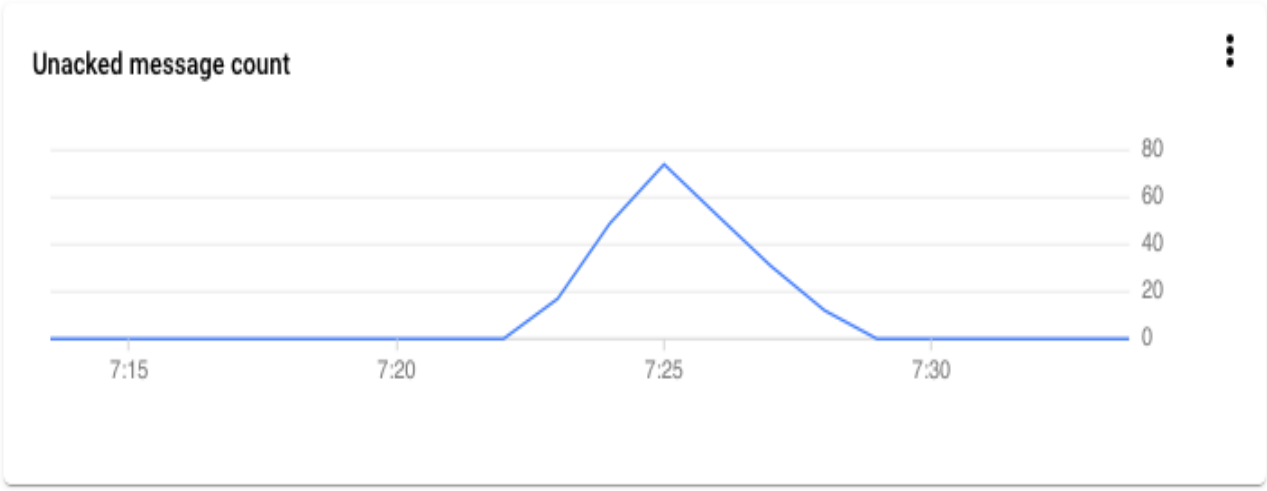


Рисунок 5.4 — Кількість повідомлень у черзі під час тестування у системі з масштабування

Базуючись на наведених даних можна зробити висновок, що система з механізмом горизонтального масштабування опрацьовувала задану кількість запитів більш ефективно.

Перетворюючи дані таблиці у формат кількості опрацьованих запитів за хвилину, можна отримати наступні пропускні здатності систем: у системі без автоматичного масштабування була зафіксована пропускна здатність у 38.17 запитів за хвилину, у системі з автоматичним масштабуванням — 83.3 запити за хвилину. Тобто система з горизонтальним масштабуванням здатна оброблювати більш ніж у два рази більше запитів за однаковий проміжок часу.

5.3 Висновки до розділу

У даному розділі було описано проведення обчислювального експерименту над реалізаціями двох систем: без масштабування та з масштабуванням. Спочатку було подано інформацію про інструмент для тестування систем навантаженням — JMeter та вказано налаштування, яке використовувалося при його запуску. Після цього наводився власне експеримент та його результати.

Було проаналізовано результати експерименту та встановлено, що створена система здатна оброблювати більш ніж у 2 рази більше запитів, ніж аналогічна система без горизонтального масштабування.

ВИСНОВКИ

Дана робота була присвячена створенню розподіленої системи з мікросервісною архітектурою з використанням автоматичного горизонтального масштабування на основі інформації про кількість повідомлень у черзі для того, щоб зробити створену систему більш надійною та відмовостійкою.

Було наведено опис загальних підходів для побудови мікросервісної архітектури, а також розглянуто принцип роботи та компоненти платформи Kubernetes. Було порівняно два набора технологій для реалізації мікровервісної архітектури та обґрунтовано вибір однієї з них для використання у даній роботі.

На основі описаних технологій було реалізовано систему з більш оптимізованим алгоритмом горизонтального масштабування на основі кількості повідомлень у черзі. Використання черги дозволило зробити систему більш надійною та стійкою до системних збоїв за рахунок буферизації отриманих повідомлень. Черга допомагає зберегти та в кінцевому рахунку забезпечити успішне виконання запити у випадку, коли один з сервісів вийшов з ладу або система отримує більше повідомлень, ніж сервіси здатні обробити. Використання інформації про кількість повідомлень у черзі дозволило більш точно налаштувати механізм горизонтального масштабування системи, що призвело до більш оптимального використання обчислювальних ресурсів.

Результати експерименту показали, що створена система здатна оброблювати більш ніж у 2 рази більшу кількість запитів за однаковий проміжок часу порівняно з системою без масштабування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Смаковський Д.С., Войналович В.А. Horizontal autoscaling of microservices in a cloud-based Kubernetes cluster // XVIII міжнародна науково-практична конференція молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики» — Київ. — квітень 2020. —том (2). — С. 122.

2. What is Kubernetes? [Електронний ресурс] — Режим доступу: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ .

3. What Is Pub/Sub? [Електронний ресурс] — Режим доступу: https://cloud.google.com/pubsub/docs/overview .

4. Google Cloud Platform Overview [Електронний ресурс] — Режим доступу: https://cloud.google.com/docs/overview.

5. JMeter [Електронний ресурс] — Режим доступу: https://jmeter.apache.org.

6. Chris Richardson. Microservices Patterns: With examples in Java – Manning, 2019.

7. Сафран Джиджи. Осваиваем Kubernetes. Оркестрами контейнерных архитектур. - СПб.: Питер, 2019.

8. Kubernetes Components [Електронний ресурс] — Режим доступу: https://kubernetes.io/docs/concepts/overview/components.

9. Deployments [Електронний ресурс] — Режим доступу: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ .

10. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. Pattern Oriented Software Architecture, Volume 1: A System of Patterns - John Wiley & Sons, 1996.

11. Spring Cloud [Електронний ресурс] — Режим доступу: https://spring.io/projects/spring-cloud .

12. Deploying Microservices: Spring Cloud vs. Kubernetes [Електронний ресурс] — Режим доступу: https://dzone.com/articles/deploying-microservices-spring-cloud-vs-kubernetes .

13. Spring Boot [Електронний ресурс] — Режим доступу: https://spring.io/projects/spring-boot#overview .

14. Spring Web MVC [Електронний ресурс] — Режим доступу: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc.

15. Spring Integration [Електронний ресурс] — Режим доступу: https://spring.io/projects/spring-integration .

16. Spring Cloud GCP [Електронний ресурс] — Режим доступу: https://spring.io/projects/spring-cloud-gcp .

ДОДАТОК А

Горизонтальне масштабування мікросервісів в хмарному Kubernetes кластері

Специфікація

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТІ6280_20Б

Аркушів 2

Київ – 2020

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ6280_20Б	Войналович-ТІ62-Записка.docx	Текстова частина дипломної роботи
Компоненти		
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ6280_20Б 12-1	PublishService.java Controller.java PublishConfig.java PubsubOutboundGateway.java	Компонент, що реалізує сервіс-видавець
УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ ТІ6280_20Б 12-2	ReceivedMessageHandler.java SubscribeConfig.java	Компонент, що реалізує сервіс-підписник

ДОДАТОК Б

Горизонтальне масштабування мікросервісів в хмарному Kubernetes кластері

Текст програмного модулю

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТІ6280_20Б

Аркушів 7

Київ – 2020

Клас PublishConfig.java

```
package ua.kpi.tef.publisher.config;

import org.springframework.cloud.gcp.pubsub.core.PubSubTemplate;
import
org.springframework.cloud.gcp.pubsub.integration.outbound.PubSubMessageHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.messaging.MessageHandler;

//Реалізує налаштування взаємодії застосунка із Cloud Pub/Sub
@Configuration
public class PublishConfig {

    /**
     * оголошує бін у контексті Spring, необхідний для інтеграції з Cloud Pub/Sub
     * @param pubsubTemplate - головний компонент для інтеграції з Cloud Pub/Sub
     *                       для відправки та отримання повідомлень
     */
    @Bean
    @ServiceActivator(inputChannel = "pubsubOutputChannel")
    public MessageHandler messageSender(PubSubTemplate pubsubTemplate) {

        // створює новий екземпляр адаптеру зовнішнього каналу для надсилання
повідомлень до Cloud Pub/Sub
        // передається назва теми, до якої потрібно публікувати повідомлення -
testTopic
        return new PubSubMessageHandler(pubsubTemplate, "testTopic");
    }
}
```

Клас Controller.java

```
package ua.kpi.tef.publisher.controller;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
```

```
import ua.kpi.tef.publisher.service.PublishService;

import static com.google.common.base.Preconditions.checkArgument;

//Реалізує точку доступу користувачів або ішних систем до застосунком
@Slf4j
@RestController
public class Controller {

    @Autowired
    private PublishService publishService;

    /**
     * приймає запити з-поза меж системи за допомогою технології REST
     * @param message - параметр вхідного запиту до застосунку, представляє собою
текстову строку
     */
    @PostMapping("/publishMessage")
    public void publishMessage(@RequestParam("message") String message) {
        //виконує валідацію вхідного повідомлення, щоб воно не було порожнім
        checkArgument(message != null && !message.isEmpty(), "Message shouldn't be
null or empty");

        //делегує надсилання відповідного повідомлення до сервісу, що відповідає за
взаємодію із Pub/Sub
        publishService.sendMessageToPubSub(message);
    }
}
```

Клас PublishService.java

```
package ua.kpi.tef.publisher.service;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import ua.kpi.tef.publisher.outbound.PubsubOutboundGateway;

//Реалізує сервіс, який надсилає повідомлення до Google Pub/Sub
@Slf4j
@Service
public class PublishService {
```

```
    @Autowired
    private PubsubOutboundGateway messagingGateway;

    /**
     * надсилання повідомлення до Google Pub/Sub використовуючи засоби Spring
Integration
     * @param message - вхідне повідомлення
     */
    public void sendMessageToPubSub(String message) {
        //виклакає messagingGateway Spring Integration для передачі повідомленняя
        messagingGateway.sendToPubsub(message);

        //вивід у консоль, що повідомлення було отримане сервісом
        log.info("Message was published: {}", message);
    }
}
```

Інтерфейс PubsubOutboundGateway.java

```
package ua.kpi.tef.publisher.outbound;

import org.springframework.integration.annotation.MessagingGateway;

//Надає проксі для інтеграційного шлюзу повідомлень,
//визначає назву каналу за замовчуванням - pubsubOutputChannel, куди будуть
надсилатися повідомлення
@MessagingGateway(defaultRequestChannel = "pubsubOutputChannel")
public interface PubsubOutboundGateway {
    void sendToPubsub(String text);
}
```

Клас SubscribeConfig.java

```
package ua.kpi.tef.subscriber.config;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cloud.gcp.pubsub.core.PubSubTemplate;
import org.springframework.cloud.gcp.pubsub.integration.AckMode;
```

```
import
org.springframework.cloud.gcp.pubsub.integration.inbound.PubSubInboundChannelAdapter
;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.channel.DirectChannel;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.MessageHandler;
import ua.kpi.tef.subscriber.service.ReceivedMessageHandler;

//Реалізовує налаштування взаємодії застосунка із Cloud Pub/Sub
@Slf4j
@Configuration
public class SubscribeConfig {

    /**
     * оголошує бін у контексті Spring, який відповідає за створення каналу,
     * який буде викликати одного підписника для кожного повідомлення
     * @return MessageChannel - створенний канал
     */
    @Bean
    public MessageChannel pubsubInputChannel() {
        return new DirectChannel();
    }

    /**
     * оголошує бін у контексті Spring, який реалізує адаптер до Cloud Pub/Sub,
     * що перетворює Cloud Pub/Sub повідом лення на Spring повідомлення і надсилає
їх до вказанного каналу
     * @param inputChannel - вхідний канал з назвою "pubsubInputChannel"
     * @param pubSubTemplate - головний компонент для інтеграції з Cloud Pub/Sub
     *                         для відправки та отримання повідомлень
     * @return PubSubInboundChannelAdapter - створений адаптер
     */
    @Bean
    public PubSubInboundChannelAdapter messageChannelAdapter(
            @Qualifier("pubsubInputChannel") MessageChannel inputChannel,
            PubSubTemplate pubSubTemplate) {

        //оголошення створюваного адаптера: ім'я підписки - testSubscription,
        //підключення вхідного каналу - pubsubInputChannel
        //налаштування автоматичного режиму підтвердження отримання повідомлення
        PubSubInboundChannelAdapter adapter =
                new PubSubInboundChannelAdapter(pubSubTemplate, "testSubscription");
```

```
        adapter.setOutputChannel(inputChannel);
        adapter.setAckMode(AckMode.MANUAL);

        return adapter;
    }

    /**
     * оголошує бін у контексті Spring, необхідний для інтеграції з Cloud Pub/Sub,
     * буде визваний, коли у канал надійде повідомлення,
     * pubsubInputChannel - назва каналу, звідки будуть отримуватися повідомлення
     * @return MessageHandler - обробник отриманого повідомлення
     */
    @Bean
    @ServiceActivator(inputChannel = "pubsubInputChannel")
    public MessageHandler messageReceiver() {
        // створює та повертає обробник повідомлень
        return new ReceivedMessageHandler();
    }
}
```

Клас ReceivedMessageHandler.java

```
package ua.kpi.tef.subscriber.service;

import lombok.extern.slf4j.Slf4j;
import
org.springframework.cloud.gcp.pubsub.support.BasicAcknowledgeablePubsubMessage;
import org.springframework.cloud.gcp.pubsub.support.GcpPubSubHeaders;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageHandler;

//Реалізує власний обробник отриманних повідомлень
@Slf4j
public class ReceivedMessageHandler implements MessageHandler {

    /**
     * примає повідомлення та виконує над ним певну логіку
     * @param message - перетворене Spring повідомлення
     */
    @Override
    public void handleMessage(Message<?> message) {
```

```
        //створює обгортку над Cloud Pub/Sub повідомленням,
        // щоб пізніше з його домогою надіслати підтвердження отримання повідомлення
        BasicAcknowledgeablePubsubMessage originalMessage =
                message.getHeaders().get(GcpPubSubHeaders.ORIGINAL_MESSAGE,
BasicAcknowledgeablePubsubMessage.class);

        try {
            // отримання тексту повідомлення
            String payload = new String((byte[]) message.getPayload());

            //зупинка потоку на 5 секунд для імітації реальних виробничих умов
            Thread.sleep(5000);

            //реєстрація, що сервіс отримав повідомлення
            log.info("Received message payload: {} ", payload);

            //надсилає підтвердження до Cloud Pub/Sub, що повідомлення було отримано
сервісом
            originalMessage.ack();
        } catch (Exception e) {

            // у випадку помилок під час обробки повідомлення надсилає підтвердження
до Cloud Pub/Sub,
            // що повідомлення не вдалося успішно виконати,
            // після чого повідомлення залишиться у черзі для наступних спроб
            originalMessage.nack();

            log.error("Exception was thrown during processing the message: {}",
e.getMessage());
            e.printStackTrace();
        }
    }
}
```

ДОДАТОК В

Горизонтальне масштабування мікросервісів в хмарному Kubernetes кластері

Опис програмного модулю

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТІ6280_20Б

Аркушів 8

Київ – 2020

АНОТАЦІЯ

Даний додаток містить стислий опис компонентів системи, що є реалізацією більш оптимального алгоритму для масштабування сервісів у кластері Kubernetes. Система призначений для забезпечення більш високого рівня надійності та відмовостійкості системи, а також для оптимізації використовуваних обчислювальних ресурсів.

Програмна частина системи була написана за допомогою мови програмування Java із використанням деяких модулів фреймворку Spring, розгортувалось за допомогою Kubernetes на хмарній платформі Google Cloud Platform.

ЗМІСТ

1 ЗАГАЛЬНІ ВІДОМОСТІ

У цьому додатку міститься опис основних компонентів системи з реалізацією оптимального алгоритму горизонтального масштабування, що виконує деякі із завдань, поставлених в розділі 1. У додатку Б міститься програмний код сервісів.

Система було розгорнута за допомогою фреймворку Spring Boot та платформи Kubernetes у хмарні обчислювальні ресурси Google Cloud Platflorm.

Сервіси були розроблені за допомогою мови програмування Java у середовищі розробки IntelliJ IDEA 2020.

2 ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Розроблена система виконує завдання горизонтального масштабування сервісів на основі інформації про довжину черги.

Створене рішення може бути використане майже у будь-якій високонавантаженій системі з мікросервісною архітектурою.

3 ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Створена система працює наступним чином: на сервіс-видавець надсилаються REST-запити від користувачів, потім ці запити надсилаються до черги. З черги запити-повідомлення отримуються сервісами-підписниками. Налаштований компонент платформи Kubernetes стежить за кількістю необроблених запитів у черзі і на основі цієї інформації робить висновок про збільшення чи зменшення кількості екземлярів сервісів-підписників у системі.

Сервіс-видавець та сервіс-підписник обмінюються повідомленнями за допомогою асинхронного виду зв’язку – черзі.

Для використання сервісів наданих GCP необхідно було підключати бібліотеки для інтеграції з ними.

4 ТЕХНІЧНІ ЗАСОБИ, ЩО ВИКОРИСТОВУЮТЬСЯ

Створену систему було протестовано у хмарному середовищі Google Cloud Platform на двох віртуальних машинах. Кожна з машин має 1 логічне ядро та 3.75 Гб оперативної пам’яті. Розроблена система не потребує значних обчислювальних ресурсів, а тому може бути розгорнута на комп’ютерах із подібною потужністю.

5 ВХІДНІ І ВИХІДНІ ДАНІ

Вхідними даними є REST-запити, що містять повідомлення у параметрі заголовку HTTP.

Вихідними даними є результат роботи HPA, а саме прийняття рішення про зменшення чи збільшення кількості екземплярів сервісу.