**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**
**ІМЕНІ ТАРАСА ШЕВЧЕНКА**
Факультет комп’ютерних наук та кібернетики
Кафедра теорії та технології програмування

**Кваліфікаційна робота**
**На здобуття ступеня магістра**
за спеціальністю 122 «Комп’ютерні науки»
на тему:

# КОНТЕЙНЕРИЗАЦІЯ ВЕБ-ЗАСТОСУНКІВ

Виконала студентка 2-го курсу магістратури
Журавель Надія Володимирівна

_______________________
(підпис)

Науковий керівник:
к.ф.-м.н., доцент
Кузенко Володимир Федорович

_______________________
(підпис)

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

Студент ___________

Роботу розглянуто і допущено до захисту на засіданні кафедри теорії та технології програмування
«____» ________________ 201_ р.,
протокол №____

Завідувач кафедри
М. С. Нікітченко __________

Київ – 2021

# РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел (9 найменувань) та 3 додатків.

Робота містить 33 рисунки. Загальний обсяг роботи становить 61 сторінок, основний текст роботи викладено на 52 сторінках.

DOCKER, KUBERNETES, CLOUD, ВЕБ-ЗАСТОСУНОК, МІГРАЦІЯ В ХМАРУ, КОНТЕЙНЕРИЗАЦІЯ, CONTAINERS.

Об'єктом роботи є контейнеризація веб-застосунків, способи створення і міграції застосунку в хмару. Предметом роботи є створений кластер Kubernetes у платформі Google Cloud, на якому поміщений Docker контейнер для платформи перевезення посилок.

**Метою роботи** є дослідження контейнеризації веб-застосунків, безпосередньо технології Docker, застосовуючи її, на практиці створити контейнер веб-додатку і мігрувати його у Google Cloud на кластер Kubernetes Engine. При цьому систематизувати, закріпити та розширити теоретичні та практичні знання з інформаційних технологій та застосувати їх при розв'язанні задачі розробки програми, використовуючи мову Java разом з фреймворком Spring Boot.

Веб-застосунок написаний об'єктно-орієнтованою мовою програмування – Java. Інструменти розроблення: фреймворк Spring, Hibernate, об'єктно-реляційна система керування базами даних PostgreSQL, система управління міграціями баз даних Liquibase. Для міграції застосунку в хмару було створено Docker контейнер і розміщено його у створеному кластері Kubernetes Engine.

Результати роботи: опис, створення та демонстрація використання контейнерів застосунків на прикладі проекту перевезення посилок.

# ЗМІСТ

РЕФЕРАТ ..... 2
ЗМІСТ ..... 3
ПЕРЕЛІК ВИКОРИСТАНИХ СКОРОЧЕНЬ ..... 5
ВСТУП ..... 6
1 КОНТЕЙНЕРИЗАЦІЯ ..... 9
1.1 Віртуалізація на рівні операційної системи ..... 9
1.2 Порівняння контейнерів та віртуальних машин ..... 12
1.3 Контейнери веб-застосунків та системні контейнери ..... 15
2 ВИБІР ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ ..... 18
2.1 Docker ..... 18
2.1.1 Історія Docker ..... 19
2.1.2 Архітектура Docker ..... 20
2.2 Kubernetes ..... 23
2.2.1 Еволюція розгортання застосунків ..... 24
2.2.2 Особливості Kubernetes ..... 26
2.2.3 Архітектура Kubernetes ..... 28
2.3 Платформа Google Cloud ..... 30
2.3.1 Google Kubernetes Engine ..... 30
2.3.2 Google Container Registry ..... 31
3 РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ ..... 33
3.1 Опис архітектури додатку ..... 34
3.2 Підключення неперевної інтеграції ..... 36
3.2.1 Створення контейнерів для тестування ..... 37
3.2.2 Визначення конфігурацій збірки ..... 39
3.2.3 Публікація нової версії у GitHub ..... 40
3.3 Міграція веб-застосунку у хмарну платформу Heroku ..... 41
4 МІГРАЦІЯ ЗАСТОСУНКУ У GOOGLE CLOUD ..... 45
4.1 Створення контейнера для всього застосунку ..... 45
4.2 Розгортання контейнера ..... 45
4.3 Створення кластера Kubernetes ..... 46
4.3 Розгортання застосунку на кластері ..... 48
4.4 Встановлення доступу до застосунку ..... 48
ВИСНОВКИ ..... 51
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ..... 52

ДОДАТОК А Файл pom.xml ........................................................................ 53
ДОДАТОК Б ER-діаграма проекту ........................................................................ 59
ДОДАТОК В Фрагмент файлу changelog-master.yaml ........................................................................ 60

# ПЕРЕЛІК ВИКОРИСТАНИХ СКОРОЧЕНЬ

**API** – Application programming interface (Прикладний програмний інтерфейс);

**CI/CD** – continuous integration / continuous delivery (Неперевна інтеграція, неперевне розгортання);

**CLI** – command-line interface (Інтерфейс командного рядка);

**DevOps** – Development і operations (Низка практик);

**REST** – Representational State Transfer (Передача репрезентативного стану);

**URL** – Uniform Resource Locator (Уніфікована адреса ресурсу);

**XML** – Extensible Markup Language (Розширювана мова розмітки);

**ВМ** – Віртуальна машина;

**ОС** – Операційна система.

# ВСТУП

Сьогодні все більшої актуальності набувають хмарні технології і можливість мігрувати власний продукт у хмару, оскільки використання хмарних сервісів – це перш за все значна економія власної IT-інфраструктури, скорочення витрат на її адміністрування і обслуговування, гнучкість і здатність справлятися з піковими навантаженнями. Ринок все більше створює запити на хмарні рішення і вважається, що за хмарними технологіями майбутнє.

За даними звіту Flexera «Хмарні рішення: поточний стан» (Rightscale 2019 State of the Cloud Report from Flexera) [1], 94% респондентів вже використовують те чи інше хмарне рішення, а витрати компаній на хмарні технології продовжують швидко зростати. Їх плановані витрати на публічні хмари в 2019 році на 24% перевищують показники 2018 року. В середньому 38% робочого навантаження припадає у респондентів на публічні хмари, а 41% відноситься до приватних. У малому і середньому бізнесі пропорція інша: 43% – публічні хмари, 35% – приватні. Відділи IT в компаніях і на підприємствах фокусуються на управлінні хмарними ресурсами та оптимізації витрат на хмарні технології. 66% підприємств вже мають виділену групу фахівців з хмарних технологій, або центр компетенцій і передового досвіду, 21% компаній планує його організувати. Пріоритетним в діяльності відділів IT вважається оптимізація витрат і грамотне управління хмарними ресурсами (68%), підготовка аргументації і прийняття рішень про те, які програми і в яких хмарних середовищах використовувати (62%), продумування і настройка політик для хмарних додатків (59%).

За прогнозом Oracle, в найближчі десять років в хмари будуть перенесені до 80% бізнес-додатків, туди ж мігрують практично всі корпоративні дані, в хмарному середовищі буде виконуватися вся розробка і тестування додатків.

Одним із ключових рішень при міграції в хмару є створення контейнерів для власних сервісів. Контейнери дають розробникам можливість створювати передбачувані середовища, ізольовані від інших додатків. Вони також можуть включати в себе програмні залежності, необхідні додатком, такі як певні версії

середовищ виконання мов програмування і інші бібліотеки програмного забезпечення. З точки зору розробника, все це гарантовано буде узгодженим незалежно від того, де в кінцевому підсумку розгорнуто додаток. Все це сприяє підвищенню продуктивності: розробники і групи IT-фахівців витрачають менше часу на налагодження і діагностику відмінностей в середовищах і більше часу на розробку нових функцій для користувачів. З іншого боку, це також збільшує гарантію якості продукту, оскільки полегшує його тестування завдяки тому, що розробники можуть робити припущення в середовищах розробки і тестування, що збільшує рівень надійності у виробничому середовищі.

Контейнери можуть працювати практично де завгодно, що значно спрощує розробку і розгортання: в операційних системах Linux, Windows і Mac; на віртуальних машинах, на машині розробника або в локальних дата-центрах; і, звичайно ж, в хмарі. Широка популярність формату образів Docker для контейнерів ще більше сприяє його переносимості.

Таким чином, контейнеризація веб-застосунків є **актуальною** темою.

**Об'єкт дослідження** – контейнеризація веб-застосунків, способи створення і міграції застосунку в хмару, **предмет дослідження** – створений кластер Kubernetes у платформі Google Cloud, на якому поміщений Docker контейнер для платформи перевезення посилок.

В роботі використано такі **методи дослідження**, як спостереження, порівняння, аналіз та синтез.

**Мета і задачі дослідження.** В рамках роботи над кваліфікаційною роботою перед автором була поставлена мета дослідити контейнеризацію веб-застосунків, безпосередньо технологію Docker, застосовуючи її, на практиці створити контейнер веб-додатку і мігрувати його у Google Cloud на кластер Kubernetes Engine. При цьому систематизувати, закріпити та розширити теоретичні та практичні знання з інформаційних технологій та застосувати їх при розв’язанні задачі розробки програми, використовуючи мову Java разом з фреймворком Spring Boot.

Поставлена мета передбачає вирішення таких **основних задач**:

- дослідження контейнеризації веб-застосунків;
- вибір засобів реалізації завдання;
- вивчення технології Docker та Kubernetes;
- демонстрація контейнеризації веб-застосунку та його міграції у хмарне сховище на прикладі розробленого проекту.

**Отримані результати:** кластер Kubernetes Engine, на яккому розміщений образ Docker контейнера проекту «Evopost».

У роботі використано такі технології як: Spring Framework [3] – універсальний фреймворк для написання бек-енду на мові Java, Hibernate – ORM фреймворк для створення відображення між об'єктами та реляційними структурами, реляційну базу даних PostgreSQL для збереження всієї інформації, технологію Docker для створення контейнерів, а також Kubernetes для міграції застосунку в хмару.

**Можливі сфери застосування.** Платформа для мандрівників «Evopost» може застосовуватися в повсякденному житті осіб, які подорожують і бажають заробляти на подорожах, а також користувачів, які бажають доставити свою посилку швидко та дешево.

# 1 КОНТЕЙНЕРИЗАЦІЯ

## 1.1 Віртуалізація на рівні операційної системи

Віртуалізація на рівні операційної системи (ОС) [2] – метод віртуалізації, при якому ядро операційної системи підтримує декілька ізольованих примірників простору користувача, замість одного. Ці примірники (часто звані контейнерами або зонами) з точки зору користувача повністю ідентичні реальному серверові. Ядро забезпечує повну ізольованість контейнерів, тому програми з різних контейнерів не можуть впливати одна на одну.

У Unix-подібних операційних системах цю функцію можна розглядати як вдосконалену реалізацію стандартного механізму chroot, який змінює видиму кореневу папку для поточного запущеного процесу та його дочірніх систем. На додаток до механізмів ізоляції, ядро часто надає функції управління ресурсами для обмеження впливу діяльності одного контейнера на інші контейнери.

Термін контейнер, хоча найпопулярніший посилається на системи віртуалізації на рівні ОС, іноді неоднозначно використовується для позначення повніших середовищ віртуальних машин, що працюють в різному ступені узгодженості з основною ОС, наприклад контейнерів Microsoft Hyper-V.

У звичайних операційних системах для персональних комп’ютерів комп’ютерна програма може бачити (хоча й не матиме доступу) всі ресурси системи. Операційна система може мати можливість дозволити або заборонити доступ до таких ресурсів, виходячи з того, яка програма їх запитує та облікового запису користувача, в контексті якого вона працює. Операційна система може також приховувати ці ресурси, так що коли комп'ютерна програма їх перераховує, вони не відображаються в результатах переліку. Проте, з точки зору програмування, комп'ютерна програма взаємодіяла з цими ресурсами, а операційна система керувала актом взаємодії.

За допомогою віртуалізації операційної системи або контейнеризації можна запускати програми в контейнерах, яким виділено лише частини цих ресурсів. Програма, яка очікує побачити весь комп’ютер після запуску всередині контейнера,

може бачити лише виділені ресурси і вважає, що це все, що доступно. У кожній операційній системі можна створити кілька контейнерів, кожному з яких виділено підмножину ресурсів комп’ютера. Кожен контейнер може містити будь-яку кількість комп'ютерних програм. Ці програми можуть працювати одночасно або окремо, і навіть можуть взаємодіяти між собою.

Контейнеризація має схожість з віртуалізацією додатків: в останньому лише одна комп'ютерна програма розміщується в ізольованому контейнері, а ізоляція застосовується лише до файлової системи.

Віртуалізація на рівні операційної системи зазвичай використовується у віртуальних хостингових середовищах, де вона корисна для надійного розподілу кінцевих апаратних ресурсів серед великої кількості користувачів. Системні адміністратори можуть також використовувати її для консолідації серверного обладнання, переміщуючи служби на окремих хостах у контейнери на одному сервері.

Інші типові сценарії включають відокремлення декількох програм для розділення контейнерів для покращення безпеки, незалежності обладнання та доданих функцій управління ресурсами. Покращена безпека забезпечується використанням механізму chroot. Реалізації віртуалізації на рівні операційної системи, здатні до реальної міграції, також можуть бути використані для динамічного балансування навантаження контейнерів між вузлами в кластері.

Віртуалізація на рівні операційної системи зазвичай вимагає менше накладних витрат, ніж повна віртуалізація, оскільки програми у віртуальних розділах на рівні ОС використовують звичайний інтерфейс системного виклику операційної системи і не потребують емуляції або запуску на проміжній віртуальній машині, як це відбувається з повною віртуалізацією (наприклад, VMware ESXi, QEMU або Hyper-V) та паравіртуалізацією (наприклад, Xen або User-mode Linux ). Ця форма віртуалізації також не вимагає апаратної підтримки для ефективної роботи.

Віртуалізація на рівні операційної системи не така гнучка, як інші підходи до віртуалізації, оскільки вона не може розміщувати гостьову операційну систему,

відмінну від хост-системи, або іншого гостьового ядра. Наприклад, у Linux хороші дистрибутиви, але інші операційні системи, такі як Windows, не можуть бути розміщені. Операційні системи, що використовують систематику змінних входів, мають обмеження у віртуалізованій архітектурі. Методи адаптації, включаючи аналіз ретрансляції хмарних серверів, підтримують віртуальне середовище на рівні ОС у цих додатках.

Solaris частково долає обмеження, описані вище, завдяки своїй типізованій зоні (branded zone), яка забезпечує можливість запуску середовища в контейнері, що емулює стару версію Solaris 8 або 9 у хості Solaris 10. Типізовані зони Linux (звані типізованими зонами "lx") також доступні в системах Solaris на базі x86, забезпечуючи повний простір користувачів Linux та підтримку виконання додатків Linux; крім того, Solaris надає утиліти, необхідні для встановлення дистрибутивів Red Hat Enterprise Linux 3.x або CentOS 3.x Linux всередині зон "lx". Однак у 2010 році типізовані зони Linux були видалені з Solaris; у 2014 році вони були повторно впроваджені в Illumos, який є гілкою Solaris з відкритим кодом, підтримуючи 32-розрядні ядра Linux.

| Механізм | Операційна система | Ліцензія | Дата випуску |
|---|---|---|---|
| chroot | більшість Юнікс-подібних операційних систем | власницька<br>BSD<br>GNU GPL<br>CDDL | 1982 |
| Solaris Containers | OpenSolaris | CDDL | 05/2008 |
| Solaris Containers | Solaris | CDDL | 01/2005 |
| FreeVPS | Linux | GNU GPL | - |
| iCore Virtual Accounts | Windows XP | власницька | 06/2008 |
| Linux-VServer | Linux | GNU GPL v.2 | - |
| LXC | Linux | GNU GPL v.2 | 2008 |
| OpenVZ | Linux | GNU GPL v.2 | - |
| Parallels Virtuozzo Containers | Linux, Microsoft Windows | власницька | - |
| FreeBSD Jail | FreeBSD | BSD | 03/2000 |
| sysjail | OpenBSD, NetBSD | BSD | - |
| WPARs | AIX | власницька | 10/2007 |
| Docker | Linux (використовуючи LXC) | Apache License 2.0 | 2013 |

Рисунок 1.1 – Приклади реалізації віртуалізації

Деякі реалізації забезпечують механізми копіювання і запису (CoW) на рівні файлу. Найчастіше стандартна файлова система використовується спільно між розділами, а ті розділи, які змінюють файли, автоматично створюють власні копії. Цей підхід простіший для створювання резервних копій, більш ефективніший за розпоряджанням простору та простіший для кешування, ніж копіювання на рівні блоку схеми копіювання та написання, що є популярним для цілісних системних віртуалізаторів. Однак віртуалізатори цілої системи можуть працювати з нетиповими файловими системами та створювати знімки всього стану системи.

### 1.2 Порівняння контейнерів та віртуальних машин

Контейнери Linux і віртуальні машини – це упаковані обчислювальні середовища, які поєднують різні IT-компоненти та ізолюють їх від решти системи. Вони відрізняються головним чином своєю масштабованістю та портативністю.

Контейнери зазвичай вимірюються в мегабайтах, оскільки тільки програма та всі файли, необхідні для її запуску, упаковані в контейнер. Часто в нього також упаковуються окремі функції, які виконують певні завдання (так звані мікросервіси). Контейнери дуже легко переміщати в різних середовищах через їх невеликі розміри та спільну операційну систему.

З іншої сторони, віртуальні машини зазвичай вимірюється в гігабайтах. Вони переважно містять власну операційну систему, на якій вони можуть виконувати одночасно кілька ресурсомістких функцій. Завдяки численним доступним для них ресурсам, віртуальні машини можуть абстрагувати, розділяти, дублювати та емулювати цілі сервери, операційні системи, робочі столи, бази даних та мережі.

На додаток до технологічних відмінностей, порівняння контейнерів з віртуальними машинами є репрезентативним порівнянням між новими IT-практиками та традиційними IT-архітектурами.

В наш час стали можливими нові IT-практики такі як, власна розробка в хмарі, неперевна інтеграція (CI), неперервне розгортання (CD) та DevOps, де робочі навантаження можна розділити на найменші одиниці, що обслуговуються –

зазвичай на одну функцію або мікросервіс. Ці невеликі одиниці найкраще упаковувати в контейнери, щоб кілька команд могли працювати над окремими частинами програми або послуги, не викликаючи конфліктів чи змінюючи код, упакований в інших контейнерах. Традиційні IT-архітектури (монолітні та застарілі програми) використовують один великий тип файлу для всіх аспектів робочого навантаження. Цей тип файлів не можна розділити, і тому він повинен бути упакований як ціла одиниця у більшому середовищі, часто саме у віртуальній машині. Раніше звичайною практикою було створення та запуск цілого додатка у віртуальній машині. Однак весь код та залежності спричиняли великі об'єм віртуальної машини та каскадні помилки навіть під час оновлення.

Завдяки своїм невеликим розмірам, контейнери можуть бути легко переміщені в оголені металеві системи, а також у публічні, приватні, гібридні та багатохмарні середовища. Вони також є ідеальним середовищем для сучасних хмарних додатків, які складаються з декількох мікросервісів, які забезпечують незалежну розробку і автоматизоване управління, зокрема розміщення у хмарі.

Програми, що працюють у хмарі, допомагають пришвидшити створення нових програм, оптимізацію існуючих програм та зв'язування всіх програм. Однак для цього потрібно, щоб контейнери були сумісні з базовою операційною системою. Порівняно з віртуальною машиною, контейнери краще використовувати, коли на меті є:

- розробка застосунку у хмарі;
- мікросервісна архітектура;
- застосування практики DevOps або CI/CD;
- переміщення масштабованих IT-проектів у суперечливе IT-середовище, яке працює під загальною операційною системою.

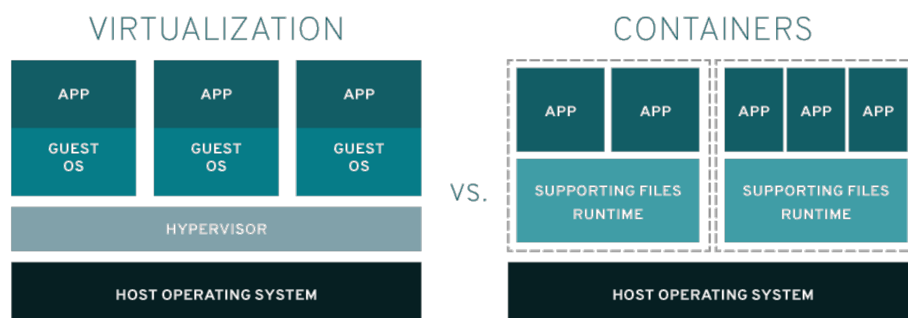
Віртуальні машини можуть виконувати набагато більше операцій, ніж один контейнер. З цієї причини монолітні навантаження все ще упаковуються у віртуальні машини. Однак ця розширена функціональність робить віртуальну машину набагато менш портативною через її залежність від операційної системи,

програми та бібліотек. Порівняно з контейнерами, віртуальні машини краще підходять для випадку, де необхідно:

- маніпулювати традиційними, застарілими та монолітними навантаженнями;
- ізолювати ризикові цикли розробки;
- надавати ресурси інфраструктури (таких як мережі, сервери та дані);
- запустити операційну систему під іншою операційною системою (наприклад, Unix під Linux).

У випадку віртуалізації програмне забезпечення під назвою "гіпервізор" відокремлює ресурси від їх фізичних пристроїв, щоб їх можна було розділити та зарезервувати для віртуальної машини. Коли користувач видає інструкцію віртуальній машині, яка вимагає додаткових ресурсів з фізичного середовища, гіпервізор спрямовує запит, який фізична система продовжує, і кешує віртуальну машину виглядають і поводяться як фізичні сервери. Це може призвести до недоліків численних операційних систем, а залежності додатків, які, як правило, навіть для одного додатка або мікросервісу не потрібні, можуть множитися.

Контейнери містять мікросервіс або програму та все, що потрібно для її запуску. Весь вміст контейнера показано на зображенні (Рис. 1.2). Це файл на основі коду, який містить усі бібліотеки та залежності. Можна вважати ці файли інсталяцією дистрибутива Linux, оскільки зображення містить усі пакети RPM та файли конфігурації. Оскільки контейнери дуже малі, часто сотні контейнерів, як правило, нещільно об’єднуються між собою. Ось чому платформи для організації контейнерів (такі як Red Hat OpenShift та Kubernetes ) використовуються для їх створення та управління.




Зображення 1.2 – Порівняння віртуалізацій та контейнерів

### 1.3 Контейнери веб-застосунків та системні контейнери

**Контейнер веб-застосунку** являє собою відносно новий тип контейнера. Це додаток, послуга або навіть рішення, орієнтоване на мікросервіс, яке зазвичай запускає лише один процес всередині. Як результат, контейнери додатків сприяють створенню незмінної та ефемерної інфраструктури. Якщо додаток або послугу потрібно оновити, з відповідного зображення будується цілий новий контейнер з необхідними налаштуваннями. Наступним кроком передбачається заміна існуючого екземпляра запущеного контейнера. У перші дні така складність, а також відсутність належної ізоляції та несумісність контейнерів додатків з деякими технологіями (такими як Java та середовищами виконання баз даних) суттєво вплинули на швидкість прийняття контейнерів програм. Розробникам довелося внести суттєві корективи, щоб гарантувати, що їхні технології та додаткові функціональні можливості можуть бути запущені всередині контейнерів правильно. Крім того, як рішення без стану, контейнери програм не можуть спочатку зберігати інформацію про стан всередині, тому такі операції делегуються зовнішнім постійним системам зберігання. Однак, на думку розробників суто контейнерів без стану, приведення стану в розгортання є застарілим способом архітектурного проектування. Вони стверджують, що найкращим і найчистішим підходом до контейнерів є виконання операцій, які не вимагають збереження стану. На ринку доступно кілька реалізацій контейнерів програм: Docker, containerd, CRI-O та деякі інші.

У наш час впроваджено багато додаткових інструментів для зручного використання служб, що використовують контейнери додатків. Як результат, розробники можуть отримати вигоду від вузькоспеціалізованих підрозділів контейнерів додатків без недоліків складного управління.

Однак деякі програми та технології все ще можуть мати проблеми під час переходу до контейнерів програм через відсутність належної ізоляції, відсутность статусу та вимог до деякого процесу. Це особливо актуально для складних інструментів оркестрації контейнерів додатків, таких як Kubernetes. У разі

труднощів з міграцією системні контейнери можна вважати найбільш підходящою альтернативою контейнеризації.

**Системні Контейнери (**також відома як операційна система контейнер) є найстарішим типом контейнерів [3]. Це рішення, орієнтоване на операційну систему, яке поводиться як автономна система, яка не потребує спеціалізованого програмного забезпечення або спеціальних зображень, таких як Docker. Системні контейнери досить схожі на віртуальні машини, але мають дуже низькі накладні витрати та просте управління. Вони зазвичай використовуються для традиційних або монолітних додатків, оскільки вони дозволяють розміщувати архітектури, інструменти та конфігурації, реалізовані для віртуальних машин. Існують різні реалізації системних контейнерів: LXC/LXD, OpenVZ/Virtuozzo, контейнери BSD, Linux vServer, контейнери Solaris та деякі інші. Системні контейнери запускають повнофункціональні системи init (systemd, SysVinit, Upstart, OpenRC), що дозволяють породити кілька процесів (наприклад, OpenSSH, crond або syslogd) всередині одного контейнера під тією самою ОС. Системні контейнери дозволяють повторно використовувати архітектури, інструменти та конфігурації, реалізовані для віртуальних машин.

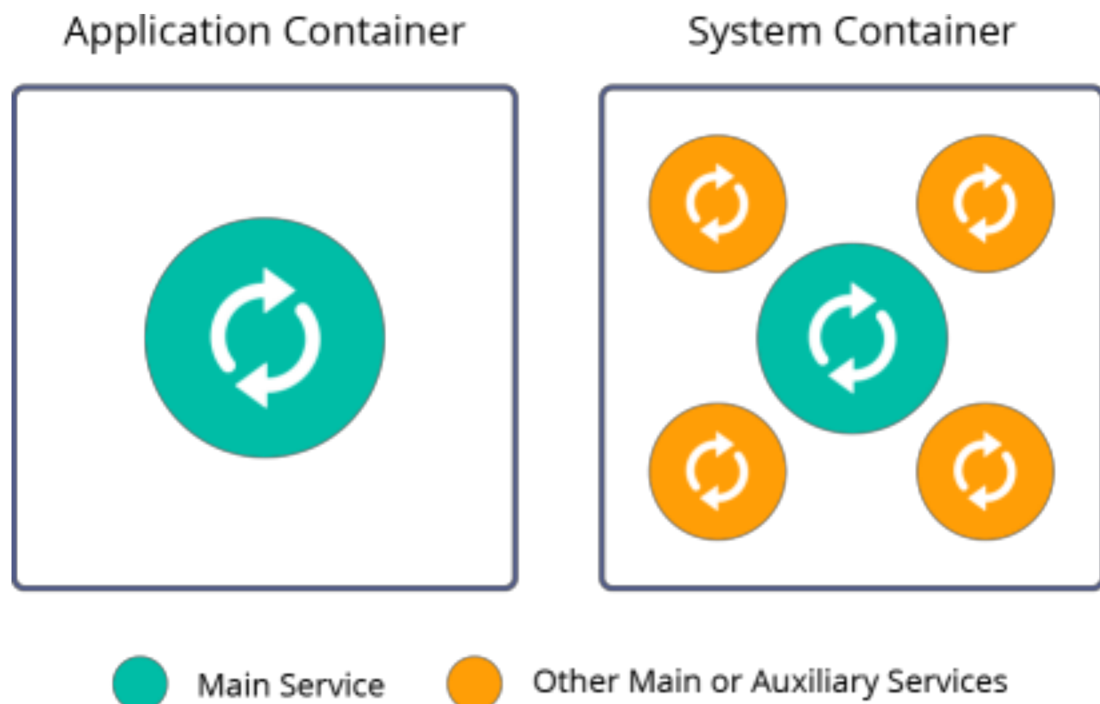



Рисунок 1.3 – Контейнер додатків та системний контейнер

Дані контейнери можна розглядати як рішення зі збереженням стану. Вони підтримують реальну міграцію (через вузли хоста, центри обробки даних або навіть хмари) і не втрачають дані чи стан після перезавантаження. Така стійкість даних ідеально підходить для запуску довготривалих додатків та служб із підтримкою стану (включаючи екземпляри баз даних SQL, NoSQL та в пам'яті).

Також системні контейнери витончено співіснують з екосистемою Java і не потребують спеціальних налаштувань. Системні контейнери підтримують існуючі рішення для здійснення гарячого перерозподілу без необхідності перезапуску контейнера або середовища виконання Java. Крім того, вони спрощують та пришвидшують кластеризацію серверів додатків Java EE / Jakarta EE. Також контейнери дозволяють запускати кілька процесів одночасно, усі під однією операційною системою, а не окремими гостьовими операційними системами. Це знижує вплив на продуктивність та забезпечує переваги віртуальних машин, таких як запуск декількох процесів, а також нові переваги контейнерів, такі як краща портативність та швидкий час запуску.

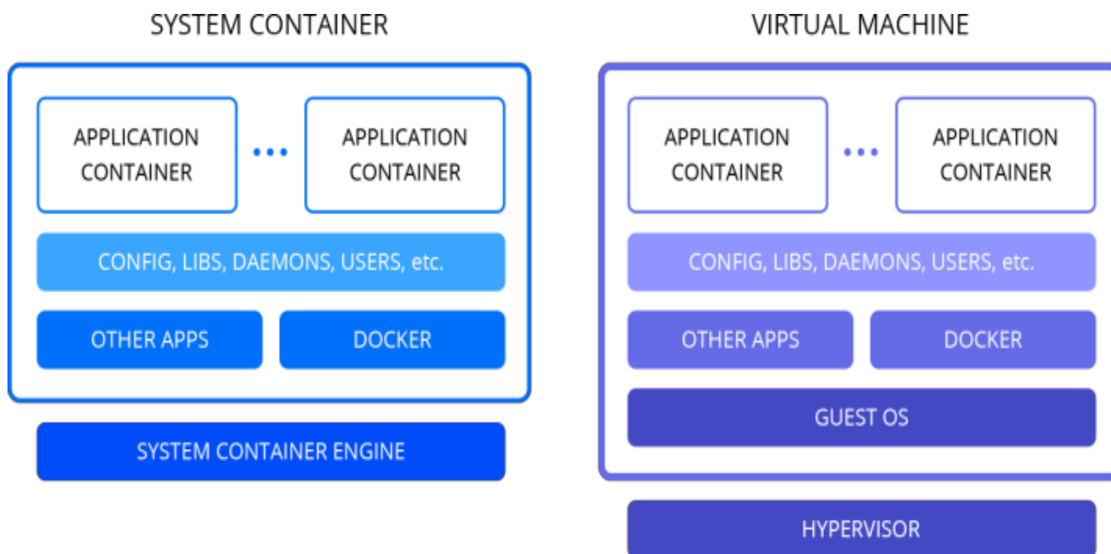



Рисунок 1.4 – Архітектура системного контейнера та віртуальної машини

# 2 ВИБІР ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ

## 2.1 Docker

Docker [4] – це набір продуктів платформи як послуги (PaaS), які використовують віртуалізацію на рівні ОС для доставки програмного забезпечення в так званих контейнерах. Контейнери ізольовані один від одного і об’єднують власне програмне забезпечення, бібліотеки та конфігураційні файли; вони можуть спілкуватися між собою за чітко визначеними каналами. Оскільки всі контейнери користуються послугами одного ядра операційної системи, вони використовують менше ресурсів, ніж віртуальні машини.

Програмне забезпечення, яке розміщує контейнери, називається Docker Engine. Вперше було розроблене в 2013 році. Docker може використовувати різні інтерфейси для доступу до функцій віртуалізації ядра Linux.

Docker може упакувати програму та її залежності у віртуальний контейнер, який може працювати на будь-якому комп’ютері Linux, Windows або macOS. Це дозволяє додатку запускатись у різних місцях, наприклад, локально, у загальнодоступній хмарі або у приватній хмарі. При запусканні в Linux Docker використовує функції ізоляції ресурсів ядра Linux (такі як cgroups та простори імен ядра) та об'єднану файлову систему (наприклад, OverlayFS), щоб дозволити контейнерам працювати, наприклад в одному Linux, уникаючи накладних витрат на запуск та обслуговування віртуальних машин. На операційній системі macOS Docker використовує віртуальну машину Linux для запускання контейнерів.

Оскільки контейнери Docker мають невелику вагу, на одному сервері або віртуальній машині може одночасно працювати кілька контейнерів. Аналіз 2018 року показав, що типовий випадок використання Docker передбачає запуск восьми контейнерів на хост і що чверть аналізованих організацій запускає 18 і більше контейнерів на хост.

Підтримка простору імен ядра Linux в основному ізолює погляд програми на операційне середовище, включаючи дерева процесів, мережу, ідентифікатори користувачів та змонтовані файлові системи, тоді як групи ядра забезпечують

обмеження ресурсів для пам'яті та центрального процесора. Починаючи з версії 0.9, Docker включає власний компонент libcontainer для безпосереднього використання засобів віртуалізації, що надаються ядром Linux, крім використання абстрагованих інтерфейсів віртуалізації через libvirt, LXC та systemd-nspawn.

Docker створив галузевий стандарт для контейнерів, щоб їх можна було переносити де завгодно. Контейнери мають спільне ядро операційної системи машини, тому не потрібна операційна система для кожного додатка, що сприяє підвищенню ефективності роботи сервера та зменшує витрати на ліцензування сервера. Docker забезпечує безпеку додатків в контейнерах і найсильніші можливості ізоляції.

Технологія Docker унікальна, оскільки вона фокусується на вимогах розробників та системних операторів, щоб відокремити залежності додатків від інфраструктури. Успіх у світі Linux зумовив партнерство з Microsoft, яке перенесло контейнери Docker та їх функціональність на Windows Server (іноді їх називають контейнерами Docker Windows).

### 2.1.1 Історія Docker

Компанія Docker Inc. була заснована Камелем Фунаді, Соломоном Хайксом та Себастьяном Пелом під час американського бізнес інкубатора Y Combinator Summer 2010 та була запущена в 2011 році. Стартап також був одним із 12 стартапів у першій групі Founder's Den. Хайкс розпочав проект Docker у Франції як внутрішній проект в рамках dotCloud, компанії з надання платформи як послуги.

Docker дебютував для публіки в Санта-Кларі на конференції Python PyCon у 2013 році. Він був випущений як відкритий код у березні 2013 року. На той час він використовував систему віртуалізації на рівні операційної системи Linux Containers (LXC) як середовище виконання за замовчуванням. Через рік, з виходом версії 0.9, Docker замінив LXC на власний компонент libcontainer, який був написаний мовою програмування Go. У 2017 році Докер створив проект Moby для відкритих досліджень та розробок.

### 2.1.2 Архітектура Docker

Docker використовує клієнт-сервер архітектуру [5]. Клієнт Docker комунікує з демоном Docker, який здійснює команди *build, run* та *pull* контейнерів Docker. Клієнт Docker і демон можуть працювати в одній системі, або можна підключити клієнт Docker до віддаленого демона Docker. Клієнт Docker і демон взаємодіють за допомогою REST API, через сокети UNIX або мережевий інтерфейс. Іншим клієнтом Docker є Docker Compose, який дозволяє працювати з програмами, що складаються з набору контейнерів.

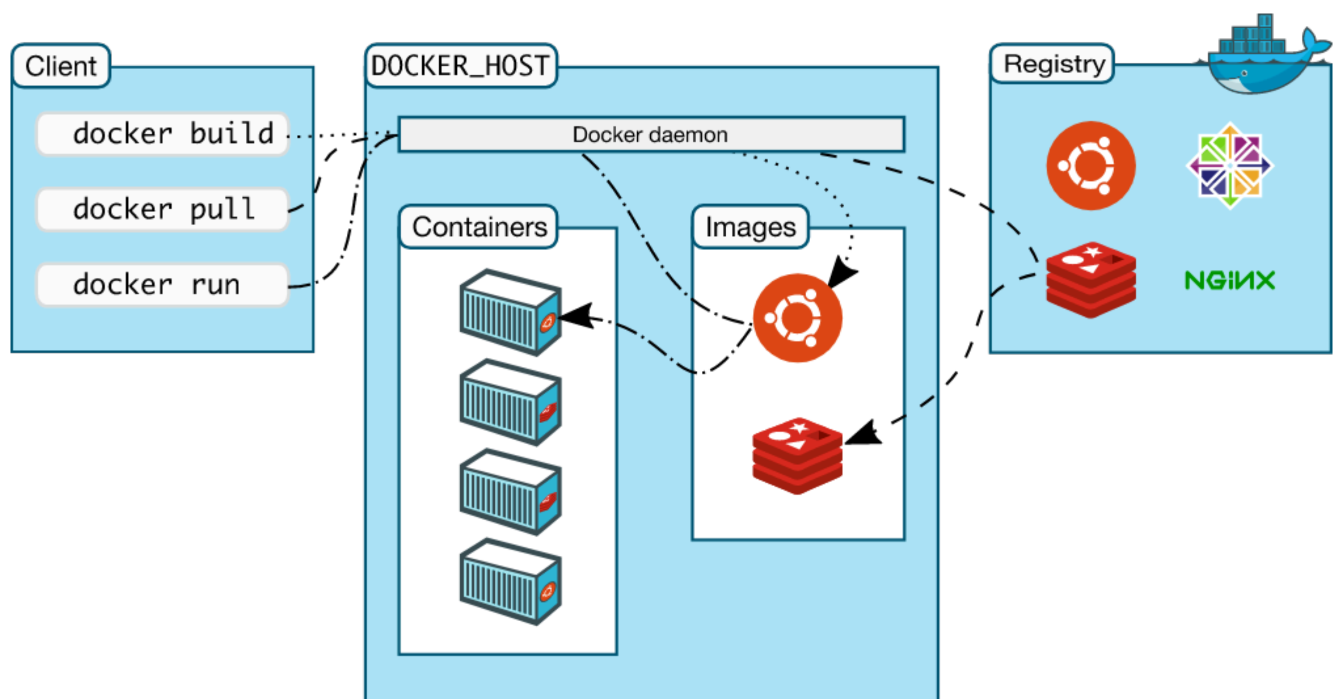



Рисунок 2.1– Архітектура Docker

**Демон Docker (Docker daemon)** (*dockerd*) прослуховує запити API Docker та керує об’єктами Docker, такими як зображення, контейнери, мережі та томи. Демон також може спілкуватися з іншими демонами для управління послугами Docker.

**Клієнт Docker (Docker client)** (*docker*) – основний спосіб взаємодії багатьох користувачів Docker з Docker. При використанні таких команд, як docker run, клієнт надсилає ці команди *dockerd*, які їх виконують. Команда *docker*

використовує API Docker, при цьому клієнт Docker може спілкуватися з декількома демонами.

**Контейнер реєстр (Docker registry)** – це сховище зображень Docker. Docker Hub – це загальнодоступний реєстр, який доступний для будь-яких користувачів, і Docker налаштований на пошук зображень у Docker Hub за замовчуванням.

При використанні команди *docker pull* або *docker run*, потрібні зображення витягуються із налаштованого реєстру. При використанні команди docker push зображення надсилається до налаштованого реєстру.

При використанні Docker, створюються та використовуються зображення, контейнери, мережі, томи, плагіни та інші об’єкти.

**Зображення (Docker images)** являється певним шаблоном доступним лише для читання з інструкціями по створенню контейнера Docker. Часто зображення базується на іншому зображенні, з деякими додатковими налаштуваннями. Наприклад, можна створити зображення, яке базується на зображенні ubuntu, але встановлює веб-сервер Apache та персональну програму, а також деталі конфігурації, необхідні для запуску цієї програми.

Можна створювати власні зображення або використовувати лише ті, що створені іншими особами та опубліковані в реєстрі. Щоб створити власний образ, необхідно створити файл Docker з простим синтаксисом для визначення кроків, необхідних для створення образу та його запуску. Кожна інструкція у файлі Docker створює шар у зображенні. Коли змінюється файл Docker і оновлюється зображення, оновлюються лише ті шари, які змінилися. Це одна з причин того, що робить зображення настільки легкими, малим та швидкими у порівнянні з іншими технологіями віртуалізації.

**Контейнер (Container)** – це запущений екземпляр зображення. Можна створювати, запускати, зупиняти, переміщувати або видаляти контейнер за допомогою Docker API або CLI. Дозволяється підключити контейнер до однієї або декількох мереж, приєднати до нього сховище або навіть створити нове зображення на основі його поточного стану.

За замовчуванням контейнер порівняно добре ізольований від інших контейнерів та його хост-машини. Також можна контролювати, наскільки мережа контейнера, сховище чи інші базові підсистеми ізольовані від інших контейнерів або від хост-машини.

Контейнер визначається його зображенням, а також будь-якими параметрами конфігурації, які надаються йому під час створення або запуску. Коли контейнер видаляється, будь-які зміни його стану, які не зберігаються в постійному сховищі, зникають.

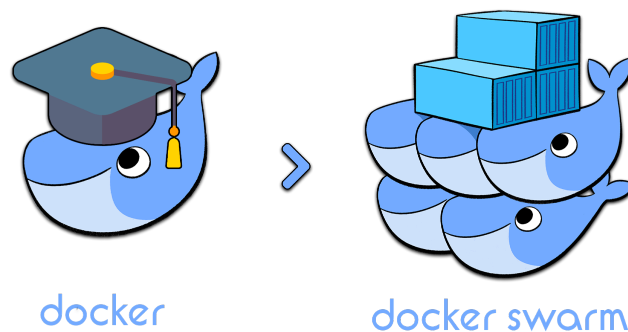
Docker написаний мовою програмування Go і використовує кілька функцій ядра Linux для забезпечення його функціональності. Docker використовує технологію, що називається набір просторів імен (namespaces), яка забезпечує ізольовану робочу область, що називається контейнером. Коли запускається контейнер, Docker створює набір просторів імен для цього контейнера. Ці простори імен забезпечують шар ізоляції. Кожен аспект контейнера працює в окремому просторі імен, і його доступ обмежений цим простором імен.

Служба Docker дозволяє масштабувати контейнери за допомогою декількох демонів Docker. Результатом є набір взаємодіючих демонів, які спілкуються через Docker API.

**Docker Compose** – це інструмент для визначення та запуску багатоконтейнерних програм Docker. Він використовує файли формату YAML для налаштування служб програми та виконує процес створення та запуску всіх контейнерів однією командою. *Docker-compose* – утиліта для командного рядка, що дозволяє виконувати команди на кількох контейнерах одночасно, наприклад, створення зображень, масштабування контейнерів, запуск контейнерів, які були зупинені та інші. Команди, пов’язані з маніпулюванням зображень або опціями інтерфейсу користувача, не є актуальними в Docker Compose, оскільки вони стосуються одного контейнера. Файл docker-compose.yml використовується для визначення служб програми та включає різні параметри конфігурації. Наприклад, build параметр визначає параметри конфігурації, такі як шлях до файлу Docker, command параметр дозволяє замінити команди Docker за замовчуванням тощо.

Перша публічна бета-версія Docker Compose (версія 0.0.1) була випущена 21 грудня 2013 р. Перша готова до виробництва версія (1.0) була доступна 16 жовтня 2014 р.

**Docker Swarm** забезпечує вбудовану функціональну кластеризацію контейнерів Docker, яка перетворює групу двигунів Docker в єдиний віртуальний механізм Docker. У Docker версії 1.12 і вище, режим Swarm інтегрований з Docker Engine. Утиліта командного рядка *docker swarm* дозволяє користувачам запускати групу контейнерів, створювати виявлення маркерів, список вузлів в кластері, і багато іншого. Утиліта командного рядка *docker node* дозволяє користувачам виконувати різні команди для керування вузлами в групі, наприклад, взяти список вузлів, оновлення вузлів і видалення вузлів з групи.




Зображення 2.2 – Зображення docker swarm

## 2.2 Kubernetes

Kubernetes (також відома як K8s) [6] – це система з відкритим кодом для автоматизації розгортання, масштабування та управління контейнерними програм. Вона групує контейнери, з яких складається додаток, у логічні блоки для зручного управління та пошуку. Kubernetes базується на 15-річному досвіді запуску виробничих робочих навантажень у Google, поєднаному з найкращими в світі ідеями та практиками спільноти. Іншими словами, Kubernetes – це портативна, розширювана платформа з відкритим кодом для управління контейнерними робочими навантаженнями та послугами, що полегшує як декларативну

конфігурацію, так і автоматизацію. Вона має велику, швидко зростаючу екосистему.

Компанія Google відкрила проект Kubernetes у 2014 році. Kubernetes поєднує в собі 15-річний досвід роботи у виробництві масштабних виробничих навантажень із найкращими в світі ідеями та практикою спільноти.

### 2.2.1 Еволюція розгортання застосунків

**Традиційна ера розгортання:** напочатку організації запускали програми на фізичних серверах. Не було можливості визначити межі ресурсів для додатків на фізичному сервері, і це спричинило проблеми з розподілом ресурсів. Наприклад, якщо на фізичному сервері працює кілька програм, можуть бути випадки, коли одна програма забирала б більшу частину ресурсів, і як результат інші програми мали б недостатню ефективність. Рішенням для цього було б запустити кожну програму на іншому фізичному сервері. Проте це рішення не вплине позитивно на результат, оскільки ресурси були недостатньо використані, і організаціям було дорого підтримувати багато фізичних серверів.

**Ера віртуалізованого розгортання:** як рішення було запроваджено віртуалізацію. Це дозволяє запускати кілька віртуальних машин на одному фізичному центральному процесорі. Віртуалізація дозволяє ізолювати програми між віртуальними машинами та забезпечує рівень захисту, оскільки інша програма не може вільно отримати доступ до інформації однієї програми.

Віртуалізація дозволяє краще використовувати ресурси на фізичному сервері та забезпечує кращу масштабованість, оскільки додаток можна легко додавати або оновлювати, зменшує апаратні витрати та багато іншого. За допомогою віртуалізації можна представити набір фізичних ресурсів як кластер одноразових віртуальних машин.

Кожна віртуальна машина є повноцінною машиною, що працює над усіма віртуалізованими апаратними засобами, включаючи власну операційну систему.

**Ера розгортання контейнерів:** контейнери схожі на віртуальні машини, але вони мають розслаблені властивості ізоляції, щоб спільно використовувати операційну систему серед програм, саме тому контейнери вважаються відносно легкими. Подібно до віртуальної машини, контейнер має власну файлову систему, частку центрального процесора, пам'ять, простір процесів тощо. Оскільки вони відокремлені від базової інфраструктури, вони переносяться між хмарами та дистрибутивами операційної системи.

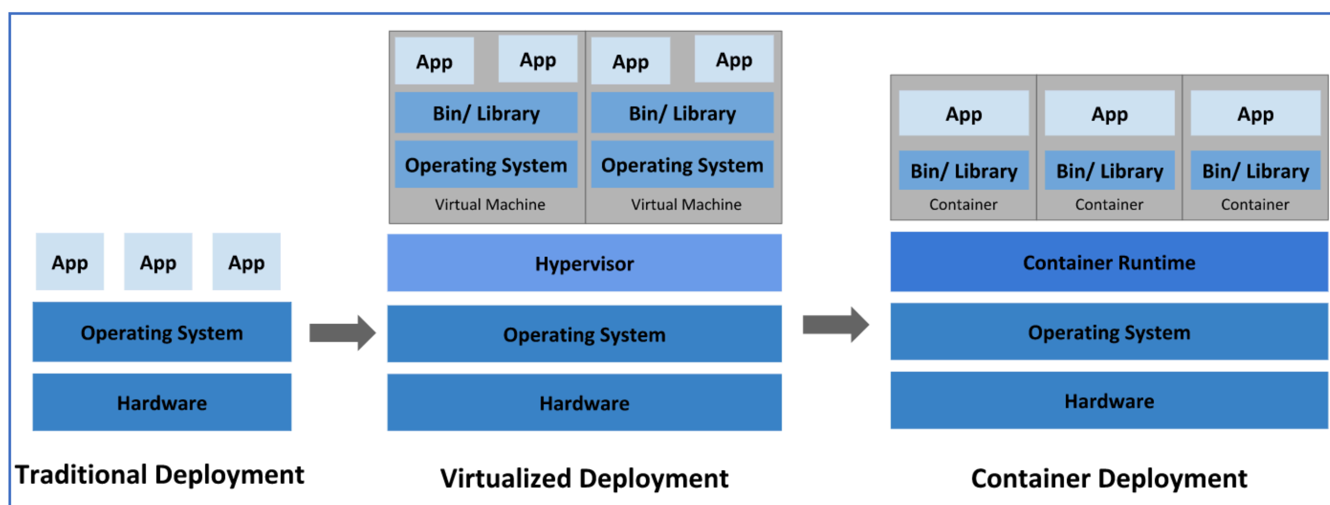



Рисунок 2.3 – Еволюція розгортання застосунків

Контейнери стали популярними, оскільки вони надають додаткові переваги, такі як:

- швидке створення та розгортання додатків: підвищена простота та ефективність створення образу контейнера порівняно з використанням зображень віртуальних машин;
- постійний розвиток, інтеграція та розгортання: забезпечує надійне та часте створення та розгортання зображень контейнерів із швидкими та ефективними відкатами (завдяки незмінності зображення);
- розділення проблем Dev та Ops: дозволяє створювати образи контейнера додатків під час збирання чи випуску, а не під час розгортання, тим самим відокремлюючи програми від інфраструктури;

• спостережливість не тільки відображає інформацію та метрики на рівні операційної системи, але також стан роботи програми та інші сигнали;

• екологічна узгодженість у розробці, тестуванні та виробництві: контейнер працює на ноутбуці так само, як у хмарі;

• переносимість хмарних та дистрибутивів ОС: працює на Ubuntu, RHEL, CoreOS, локально, на великих загальнодоступних хмарах та де-небудь ще;

• управління, орієнтоване на програми: підвищує рівень абстракції від запуску ОС на віртуальному обладнанні до запуску програми в ОС за допомогою логічних ресурсів;

• слабко пов'язані, розподілені, еластичні мікросервіси: додатки розбиті на менші, незалежні частини, їх можна розгортати та керувати динамічно – не монолітний стек, що працює на одній великій одноцільовій машині;

• ізоляція ресурсів: передбачувана продуктивність програми;

• використання ресурсів: висока ефективність та щільність.

### 2.2.2 Особливості Kubernetes

Контейнери – це хороший спосіб об’єднати та запустити необхідні програми. У виробничому середовищі потрібно керувати контейнерами, в яких запущені програми, і переконуватися, що немає простоїв. Наприклад, якщо контейнер зупиняється, потрібно запускати інший контейнер. Хорошим рішенням для цього було використання певної системи.

Kubernetes надає структуру для стабільного запуску розподілених систем, забезпечує масштабування та відновлення програми після відмови, надає схеми розгортання тощо. Kubernetes не є традиційною системою PaaS (платформа як послуга), що включає все. Оскільки Kubernetes працює на рівні контейнера, а не на апаратному рівні, він надає деякі загальноприйнятні функції, загальні для пропозицій PaaS, такі як розгортання, масштабування, балансування навантаження та дозволяє користувачам інтегрувати свої рішення для ведення журналу,

моніторингу та попередження. Однак Kubernetes не є монолітним, і ці рішення за замовчуванням є необов’язковими та підключаються за бажанням.

Kubernetes забезпечує такі послуги [7], як:

• Виявлення послуги та балансування навантаження Kubernetes може розмістити контейнер, використовуючи ім'я DNS або використовуючи власну IP-адресу. Якщо трафік до контейнера високий, Kubernetes може балансувати навантаження і розподіляти мережевий трафік таким чином, щоб розгортання було стабільним.

• Організація зберігання Kubernetes дозволяє автоматично змонтувати обрану систему зберігання, наприклад, локальні сховища, загальнодоступні хмарні провайдери тощо.

• Автоматизовані розгортання та повернення. Можна описати бажаний стан для розгорнутих контейнерів за допомогою Kubernetes, і він може змінити фактичний стан до бажаного стану з контрольованою швидкістю. Наприклад, можна автоматизувати Kubernetes для створення нових контейнерів для певного розгортання, видалення існуючих контейнерів та прийняття всіх їх ресурсів до нового контейнера.

• Автоматична упаковка сміття. Необхідно визначити для Kubernetes кластер вузлів, який він може використовувати для запуску контейнерних завдань. Для цього повинно бути повідомляно Kubernetes, скільки ЦП та пам'яті (ОЗУ) потрібно кожному контейнеру. Kubernetes може помістити контейнери на задані вузли, щоб найкраще використовувати ресурси.

• Самовідновлення. Kubernetes перезапускає контейнери, які не замінює контейнери, вбиває контейнери, які не відповідають на ваш певний користувачем перевірки здоров'я, і не рекламує їх клієнтів , поки вони не будуть готові служити.

• Управління змінними середовища та конфігурацією. Kubernetes дозволяє зберігати та керувати конфіденційною інформацією, такою як паролі, маркери OAuth та ключі SSH. Можна розгортати та оновлювати секретні змінні та

конфігурацію програми, не відновлюючи образи контейнера та не відкриваючи секретні змінні у конфігурації стека.

### 2.2.3 Архітектура Kubernetes

При розгортанні Kubernetes створюється кластер (cluster). Кластер Kubernetes складається з набору робочих машин, що називається вузлами (nodes), які запускають контейнерні програми. Кожен кластер має принаймні один робочий вузол. Робочий вузол розміщує поди (pods), що є компонентами робочого навантаження програми. Контрольна площина керує робочими вузлами та подами в кластері. У виробничих середовищах площина управління, як правило, працює на декількох комп'ютерах, а кластер, як правило, працює на декількох вузлах, забезпечуючи відмовостійкість і високу доступність.

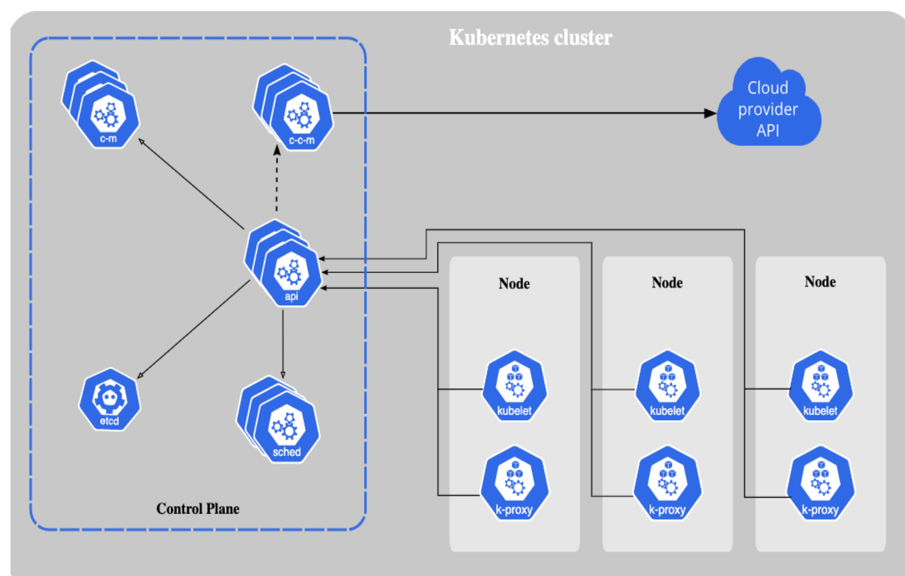



Рисунок 2.4 – Архітектура Kubernetes

Компоненти площини управління приймають глобальні рішення щодо кластера (наприклад, планування), а також виявляють і реагують на події кластера (наприклад, запускають нову поду, коли поле replicas розгортання не задовольняє необхідній кількості реплік). Компоненти площини управління можна запускати на будь-якій машині кластера. Однак для простоти сценарії налаштування зазвичай

запускають усі компоненти площини управління на одній машині і не запускають на цій машині контейнери користувачів.

Сервер API є компонентом Kubernetes панелі управління, який предстваляє API Kubernetes. Сервер API – це інтерфейс для площини управління Kubernetes. Основною реалізацією сервера Kubernetes API є kube-apiserver, який призначений для горизонтального масштабування, тобто масштабується шляхом розгортання більшої кількості екземплярів. Можна запустити кілька екземплярів kube-apiserver і збалансувати трафік між цими екземплярами.

Розподілене та надійне сховище даних в формі “ключ-значень" etcd, що використовується як резервне сховище Kubernetes для всіх даних кластера.

При використанні etcd як основно сховища, потрібно обов’язково настроїти резервне копіювання цих даних.

Компонент площини управління, який стежить за створеними подами без призначення вузола kube-scheduler вибирає вузол для запуску. До факторів, що враховуються при прийнятті рішень щодо планування, належать: індивідуальні та колективні вимоги до ресурсів, обмеження на апаратне чи програмне забезпечення, специфікації належності та неналежності вуздів, локалізація даних, терміни.

Cloud-controller-manager запускає контролери, які взаємодіють з основними хмарними провайдерами. Логічно, кожен контролер є окремим процесом, але для зменшення складності всі вони компілюються в один двійковий файл і виконуються в одному процесі.

Компоненти вузла працюють на кожному вузлі, підтримуючи запущені вузли та забезпечуючи середовище виконання Kubernetes.

Kuberlet агент, який працює на кожному вузолі кластері, запезпечує роботу контейнерів в поді. Kubelet приймає набір PodSpecs, які надаються за допомогою різних механізмів, і гарантує роботу контейнерів, описаних в цих PodSpecs. Kubelet не керує контейнерами, які не були створені Kubernetes.

Kube-proxy - це мережевий проксі, який працює на кожному вузол у вашому кластері, реалізуючи частину концепції Kubernetes сервіс. kube-proxy підтримує мережеві правила на вузлах. Ці правила мережі дозволяють здійснювати

мережевий зв'язок з підсистемами з мережевих сеансів всередині або поза кластером. kube-proxy використовує рівень фільтрації пакетів операційної системи, якщо такий є, і він доступний. В іншому випадку kube-proxy сам перенаправляє трафік.

Інформаційна панель – це веб-інтерфейс загального призначення для кластерів Kubernetes. Це дозволяє користувачам керувати програмами, що працюють у кластері, та усунути неполадки, а також самим кластером.

### 2.3 Платформа Google Cloud

Google Cloud Platform [8] – запропонований компанією Google набір хмарних служб, які виконуються на тій же самій інфраструктурі, яку Google використовує для своїх продуктів призначених для кінцевих споживачів. Google Cloud Platform надає своїм клієнтам платформу як послугу, інфраструктуру як послугу та безсерверні обчислювальні середовища.

У квітні 2008 року Google оголосив про App Engine, платформу для розробки та розміщення веб-додатків у керованих Google центрах обробки даних, що стало першою послугою хмарних обчислень від компанії. Послуга стала загальнодоступною в листопаді 2011 року. З моменту оголошення App Engine, Google додав до платформи кілька хмарних сервісів.

Google Cloud Platform є частиною Google Cloud, яка включає в себе суспільну хмарну інфраструктуру Google Cloud Platform, а також Google Workspace (G Suite), корпоративні версії Android і Chrome ОС, а також інтерфейси прикладного програмування (API).

#### 2.3.1 Google Kubernetes Engine

Захищений і повністю керований сервіс Kubernetes з революційним режимом роботи автопілота забезпечує швидкий запуск кластерів, включаючи мультизональні та регіональні, масштабування до 15000 вузлів і безпеку, включаючи сканування вразливостей образів контейнерів і шифрування даних, а

також інтегрований хмарний моніторинг з уявленнями інфраструктури, додатків і Kubernetes [9].

Особливостями Kubernetes Engine є:

- Режим роботи автопілота

Оптимізований кластер з попередньо налаштованими параметрами робочого навантаження, що забезпечує роботу без вузлів. Google може керувати інфраструктурою всього кластера, включаючи вузли. Максимально збільшує операційну ефективність і безпеку додатків, обмеживши доступ тільки до Kubernetes API і захистивши від мутації вузлів.

- Автомасштабування пода і кластера

Горизонтальне автомасштабирование подів на основі використання центрального процесора або настроююванні показників, автоматичне масштабування кластера і вертикальне автомасштабування подів, яке безперервно аналізує використання центрального процесора і пам'яті модулів. Автоматично встановлює масштабування збірки вузлів і кластерів з кількох збірок вузлів в залежності від мінливих вимог до робочого навантаження.

- Додатки Kubernetes

Готові до роботи контейнерні рішення з готовими шаблонами розгортання, що забезпечують перенесення, спрощене.

### 2.3.2 Google Container Registry

Реєстр контейнерів – це сервіс, який забезпечує контроль над образами Docker, виконувати аналіз вразливостей і встановлювати доступ за допомогою детального контролю доступу. Існуючі інтеграції CI/CD дозволяють налаштовувати повністю автоматизовані конвеєри Docker для швидкого зворотного зв’язку. Сервіс дозволяє визначати хто може отримувати доступ, переглядати або завантажувати зображення, а також забезпечує стабільний час безвідмовної роботи в інфраструктурі, захищеної системою безпеки Google. Він надає можливість автоматично створювати та надсилати образи в приватний реєстр

при фіксації коду в хмарних репозиторіях вихідного коду, GitHub або Bitbucket. Можна легко налаштовувати конвеєри CI/CD з інтеграцією в Cloud Build або розгортати безпосередньо в Google Kubernetes Engine, App Engine, Cloud Functions або Firebase. Постійно оновлювана база даних допомагає своєчасно оновлювати сканування вразливостей і виявляти нові шкідливі програми.

Двійкова авторизація для визначення політик і запобіганню розгортання образів, що суперечать встановленим політикам. Дозволяється вімкнути автоматичне блокування образів контейнерів, щоб заборонити розгортання небезпечних образів в Google Kubernetes Engine.

Надається можливість встановити кількість реєстрів залежно від необхідності, а також надсилати образи Docker у свій приватний реєстр контейнерів за допомогою стандартного інтерфейсу командного рядка Docker.

Завдяки можливості використання регіональних приватних репозиторіїв по всьому світу, можна отримати оптимальний час відгуку з будь-якого розташування.

# 3 РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ

Додаток «Evopost» призначений для передачі посилок користувачам, де один з користувачів створює запит про послугу, а інший приймає його. Він дозволяє користувачу створювати, зберігати та вилучати як посилки, так і поїздки. Окрім того, користувач має створити запит про купівлю певного товару і його доставку.

Дана робота передбачає демонстрування міграції даного застосунку у хмару, а також створення Docker контейнерів для тестування, що в одночас забезпечить перевірку нової версії перед розгортанням.

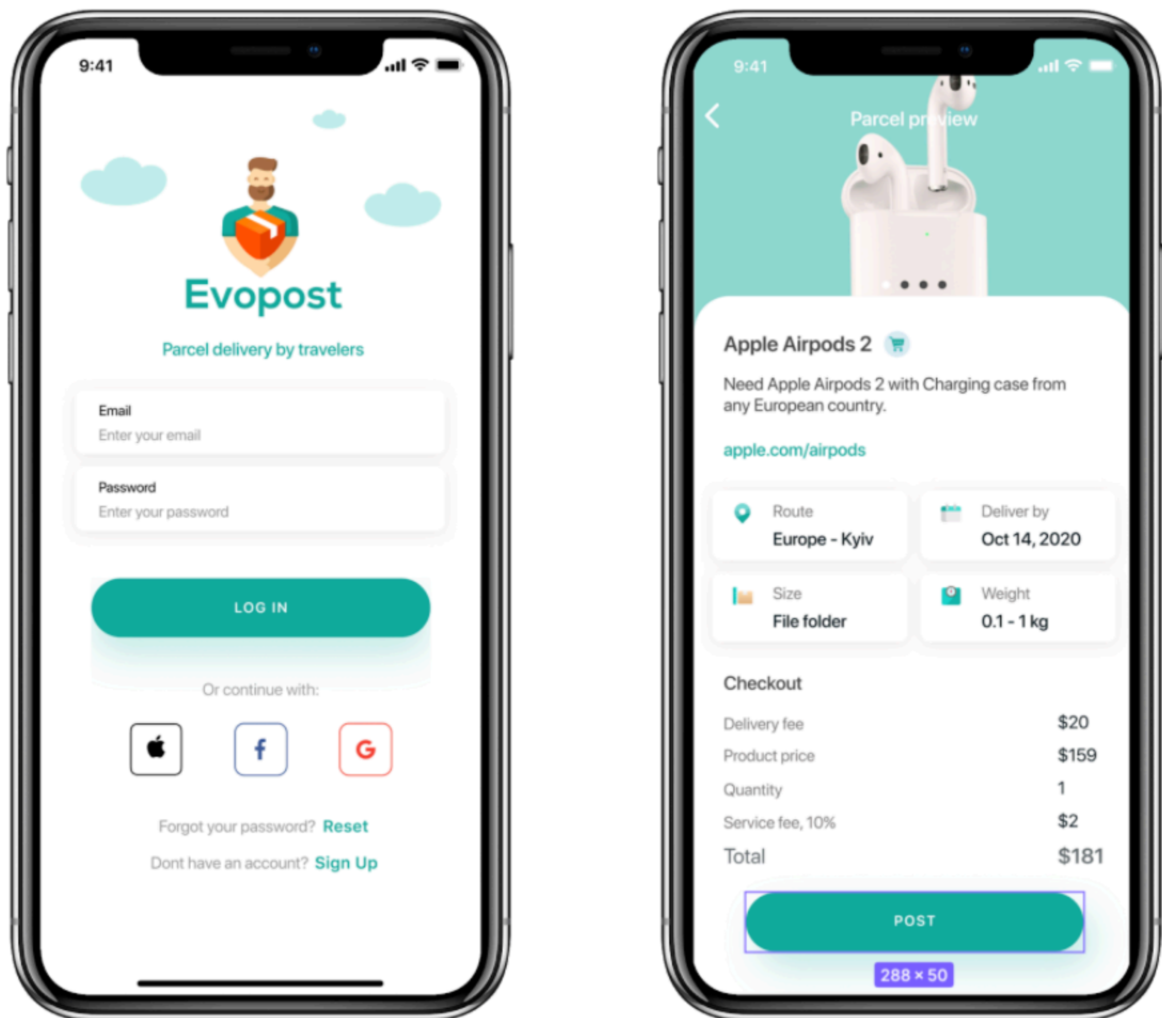



Рисунок 3.1 – Прототип додатку Evopost

### 3.1 Опис архітектури додатку

В основі проекту лежать фреймворк Spring та засіб автоматизації Maven. Всі дані зберігаються у системі керування базами даних PostgreSQL, зокрема була підключена система управління міграціями бази даних Liquibase.

Hibernate забезпечує відображення об'єктів у базу даних (ORM). Стандартизованим інтерфейсом для фреймворку Hibernate виступає Java Persistence API, а Entity являється безпосереднім об'єктом, для якого забезпечується ORM. Тому було створено пакет entity, в якому поміщено всі Entities. ER-діаграму наведено у додатку Б. Spring Data надає набір готових реалізацій для створення DAO (об'єктів, що надають абстрактний інтерфейс до певного типу бази даних), але Spring вирішили називати їх не DAO, а Repository. Тому для кожного Entity було створено Repository, який дозволяє керувати об'єктом у базі даних і вони поміщені у пакеті repository. Всі Repository наслідують JpaRepository – інтерфейс фреймворка Spring Data, який забезпечує набір стандартних методів JPA для роботи з базою даних.

Напряму використовувати Repository для отримання даних на інтерфейсі користувача не прийнято, тому використовують Services – Java-класи, які забезпечують основну бізнес-логіку проекту. Було створено пакет service, у якому поміщено всі сервіси. Кожен сервіс представлений інтерфейсом, в якому вказані необхідні методи для написання бізнес-логіки, та його реалізацією. Всі реалізації поміщені у підпакет impl.

Також було додано Lombok плагін, який завдяки анотаціям дозволив позбутися всіх конструкторів та методів getter, setter, equals і toString.

Для можливості слідкувати за виконанням бізнес-логіки проекту було додано Slf4J – бібліотеку логування. Було створено пакет config, у ньому клас AppConfig.java, у якому містився Logger Bean.

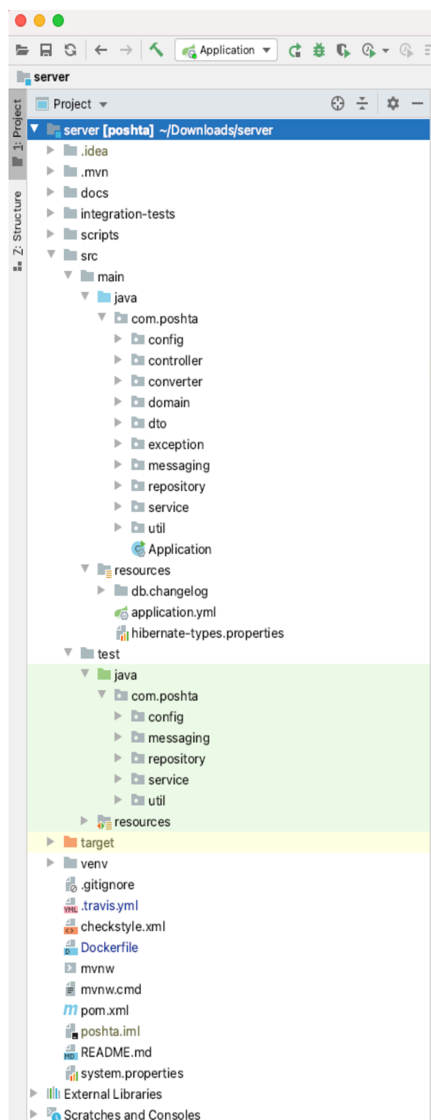
Щоб забезпечити обробку виключень було створено пакет exceptions, у який поміщено всі класи Exceptions.

У Spring Security є кілька методів шифрування. Для шифрування паролів користувачів було використано BCrypt, оскільки це найкраще рішення. Було визначено Bean BCryptPasswordEncoder у відповідній конфігурації, а також підключено шифрування до реєстрації користувача і аутентифікації.

Для Google та Facebook Sign-in авторизації було підключено протокол OAuth2. Для цього було створено директорію з назвою oauth2, у яку було поміщено всі необхідні класи.

Для оплати онлайн було підключено платіжну систему Stripe, яка дозволяє користувачам здійснювати оплату послуг.

Файл pom.xml з усіма залежностями наведено у додатку А.



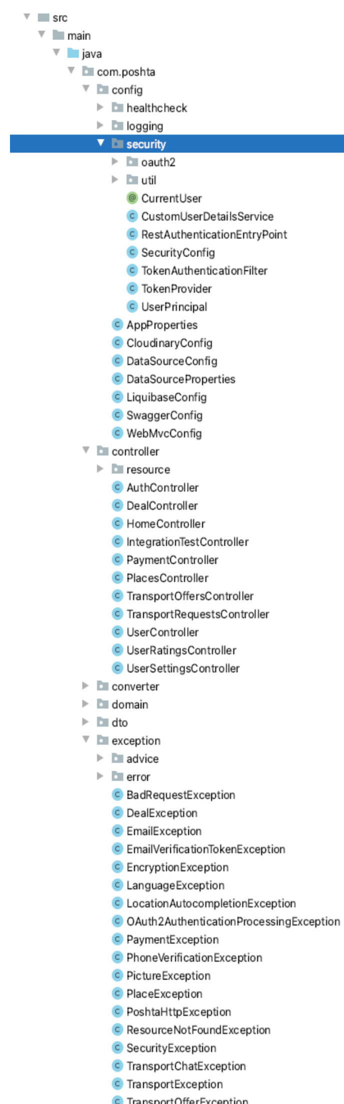




Рисунок 3.2, Рисунок 3.3 – Структура проекту

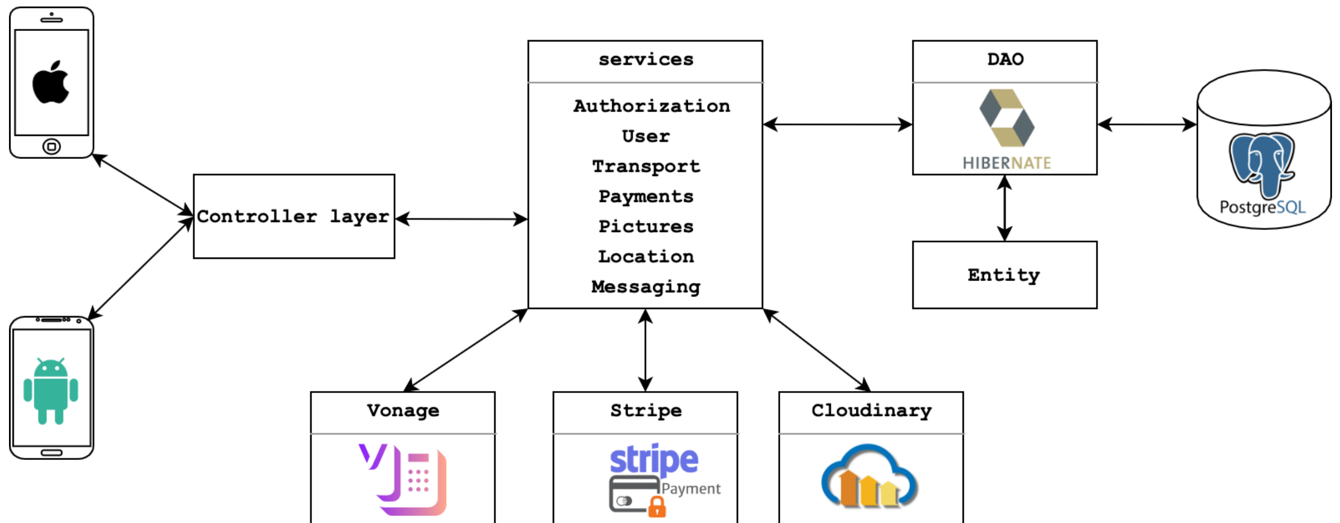



Рисунок 3.4 – Архітектура проекту

## 3.2 Підключення неперевної інтеграції

Неперервна інтеграція – це практика частого злиття дрібних змін коду, а не злиття великих змін в кінці циклу розробки. Метою є створення більш здорового програмного забезпечення шляхом розробки та тестування з меншими кроками.

Як платформа безперервної інтеграції, Travis CI підтримує процес розробки шляхом автоматичного створення та тестування змін коду, забезпечуючи негайний зворотний зв’язок про успіх змін. Travis CI також може автоматизувати інші частини процесу розробки, керуючи розгортаннями та сповіщеннями.

Оскільки Evopost є командним проектом, для зручності його реалізації було використано один з найбільших веб-сервісів для спільної розробки програмного забезпечення GitHub, що базується на системі керування версіями Git. Travis CI надає можливість інтеграції разом з GitHub. Під час запуску збірки Travis CI клонує сховище GitHub у абсолютно нове віртуальне середовище та виконує ряд завдань для побудови та тестування коду. Якщо одне або кілька з цих завдань не завершується успішно, збірка вважається зламаною. Якщо всі завдання завершуються успішно, збірка вважається пройденою, і Travis CI може розгорнути код на веб-сервері або хості програми. Збірки CI також можуть автоматизувати

інші частини робочого процесу доставки. Це означає, що є можливіть мати завдання, які залежать одне від одного за допомогою етапів збірки, налаштовувати сповіщення, готувати розгортання після збірки та інші.

### 3.2.1 Створення контейнерів для тестування

Compose – це інструмент для визначення та запуску багатоконтейнерних програм Docker. За допомогою Compose використовується файл розширення .yaml для налаштування сервісів програми. Потім за допомогою однієї команди створюються та запускаються всі сервіси з конфігурації.

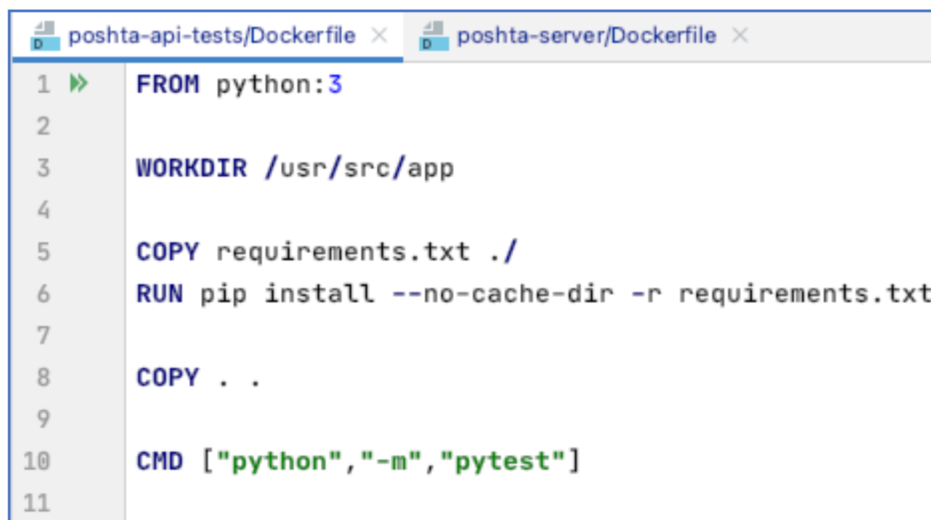
Використання Compose – це в основному триетапний процес:

- Було визначено середовище програми у Dockerfile, щоб його можна було відтворювати де завгодно.

Для тестування було створено два Docker контейнери, перший містить команду для запускання веб-застосунка, а другий для тестів.

- Було створено docker-compose.yml файл для можливості запуску контейнерів в ізольованому середовищі.

- Після запуску команди *docker compose up* запускається команда Docker compose та запускає весь додаток.



poshta-api-tests/Dockerfile | poshta-server/Dockerfile

```
FROM openjdk:11

RUN mkdir -p /usr/src/server/

COPY ./poshta-0.0.1-SNAPSHOT.jar /usr/src/server/poshta-server.jar

CMD ["java","-jar","/usr/src/server/poshta-server.jar"]
```

Рисунок 3.5, Рисунок 3.6 – Файли Dockerfile для тестування

.travis.yml | docker-compose.yml

```
version: '3'

services:
  poshta-db:
    image: postgres:11.1
    ports:
      - 5428:5432
#    volumes:
#      - ./postgres-data:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=ituser
      - POSTGRES_PASSWORD=itpassword
      - POSTGRES_DB=poshtadbit
    healthcheck:
      test: ["CMD", "pg_isready"]
      timeout: 5s
      retries: 3
  poshta-server:
    build: ./poshta-server
    image: poshta-server:1.0
    ports:
      - 8079:8080
    environment:
      - SPRING_ACTIVE_PROFILES=default,integration-tests
      - DB_DRIVER=org.postgresql.Driver
      - DB_URL=jdbc:postgresql://poshta-db:5432/poshtadbit
      - DB_USERNAME=ituser
      - DB_PASSWORD=itpassword
      - GOOGLE_CLIENT_ID=it-google-client-id
      - GOOGLE_CLIENT_SECRET=it-google-client-secret
      - GOOGLE_CLIENT_ID_IOS=it-google-client-ios
      - GOOGLE_CLIENT_ID_ANDROID=it-google-client-android
```

```
33             - FB_CLIENT_ID=it-fb-client-id
34             - FB_CLIENT_SECRET=it-fb-client-secret
35             - OAUTH_TOKEN_SECRET=it-oauth-token-secret
36             - MAIL_PASSWORD=it-mail-password
37             - APP_URL=http://localhost:8080
38             - CLOUDINARY_CLOUD_NAME=it-evopost
39             - CLOUDINARY_API_KEY=it-cloudinary-apy-key
40             - CLOUDINARY_API_SECRET=it-cloudinary-api-secret
41             - NEXMO_API_KEY=it-nexmo-api-key
42             - NEXMO_API_SECRET=it-nexmo-api-secret
43             - STRIPE_API_SECRET=it-stripe-api-secret
44             - GOOGLE_PLACES_API_KEY=it-google-places-api-key-secret
45           healthcheck:
46             test: ["CMD", "curl", "-f", "http://poshta-server:8080/health"]
47             interval: 8s
48             retries: 10
49           depends_on:
50             - poshta-db
51         poshta-api-tests:
52           build: ./poshta-api-tests
53           image: poshta-api-tests:1.0
54           environment:
55             - API_BASE_URL=http://poshta-server:8080
56             - DB_NAME=poshtadbit
57             - DB_USERNAME=ituser
58             - DB_PASSWORD=itpassword
59           depends_on:
60             - poshta-server
```

Рисунок 3.7, Рисунок 3.8 – Файл docker-compose.yml

### 3.2.2 Визначення конфігурацій збірки

Збірки на Travis CI налаштовуються переважно за допомогою конфігурації збірки, що зберігається у файлі .travis.yml у сховищі. Це дозволяє конфігурації керувати версіями та забезпечує гнучкість.

Для випадків розширеного використання основний файл конфігурації збірки .travis.yml може імпортувати інші спільні джерела конфігурації за допомогою функції Build Config Imports. Наступним кроком було саме створення даного файлу.

.travis.yml | docker-compose.yml

```
sudo: required

cache:
  directories:
    - .autoconf
    - $HOME/.m2
    - $HOME/docker

#before_cache:
# Save tagged docker images
#- >
#  mkdir -p $HOME/docker && docker images -a --filter='reference=*postgre*' --filter='reference=*python*' --filter='reference=*openjdk*' --form
#  | xargs -n 2 -t sh -c 'test -e $HOME/docker/$1.tar.gz || docker save $0 | gzip -2 > $HOME/docker/$1.tar.gz'
services:
  - docker

env:
  DOCKER_COMPOSE_VERSION: 1.23.2

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VERSION}/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin
  # Load cached docker images
  #- if [[ -d $HOME/docker ]]; then ls $HOME/docker/*.tar.gz | xargs -I {file} sh -c "zcat {file} | docker load"; fi

before_script:
  - mvn -B clean install -DskipTests
  - docker-compose -f integration-tests/docker-compose.yml build

script:
  - mvn -B test
  - docker-compose -f integration-tests/docker-compose.yml up --abort-on-container-exit --exit-code-from poshta-api-tests

after_script:
  - docker-compose -f integration-tests/docker-compose.yml rm -fsv
```

Рисунок 3.9 – Файл .travis.yml

### 3.2.3 Публікація нової версії у GitHub

Після створення Pull Request у GitHub автматично запускаються дві збірки, які провіряють зміни на валідність і дозволяють з’єднати дану гілку з основною.

Рисунок 3.10 – Приклад роботи Travis CI у GitHub

## 3.3 Міграція веб-застосунку у хмарну платформу Heroku

Heroku – це одна з перших хмарних платформ як сервіс (PaaS), що підтримує кілька мов програмування. Програми, які запускаються на Heroku, як правило, мають унікальний домен, який використовується для маршрутизації HTTP-запитів до правильного контейнера програм. Кожен з контейнерів розподілений по сітці, яка складається з декількох серверів. Сервер Heroku Git обробляє сховища додатків, що надходять від дозволених користувачів. Всі послуги Heroku розміщені на платформі хмарних обчислень Amazon EC2 .

З програм Heroku було використано два сервіси.

- Платформа Heroku

Мережа Heroku запускає програми замовника у віртуальних контейнерах, які виконуються в надійному середовищі виконання. Heroku називає ці контейнери "Dynos". Ці Dynos можуть запускати код, написаний на Node, Ruby, PHP, Go, Scala, Python, Java або Clojure. Heroku також пропонує власні пакети збірки, за допомогою яких розробник може розгортати програми на будь-якій іншій мові. Heroku дозволяє розробнику миттєво масштабувати програму, просто збільшуючи кількість динозаврів або змінюючи тип динозаврів, в яких працює програма.

- Heroku Postgres

Heroku Postgres – це служба хмарних баз даних (DBaaS) для Heroku, заснована на PostgreSQL . Heroku Postgres надає такі функції, як постійний захист, відкат та висока доступність; також гілки, фоловери та кліпи.

В Heroku було створено акаунт і підключено наші сервіси. Основний сервер називається poshta. Було вставлено автоматичний деплоймент з GitHub репозиторію, а також додано змінні конфігурації, оскільки задля безпеки продукту їх варто не поміщати разом з кодом у GitHub.

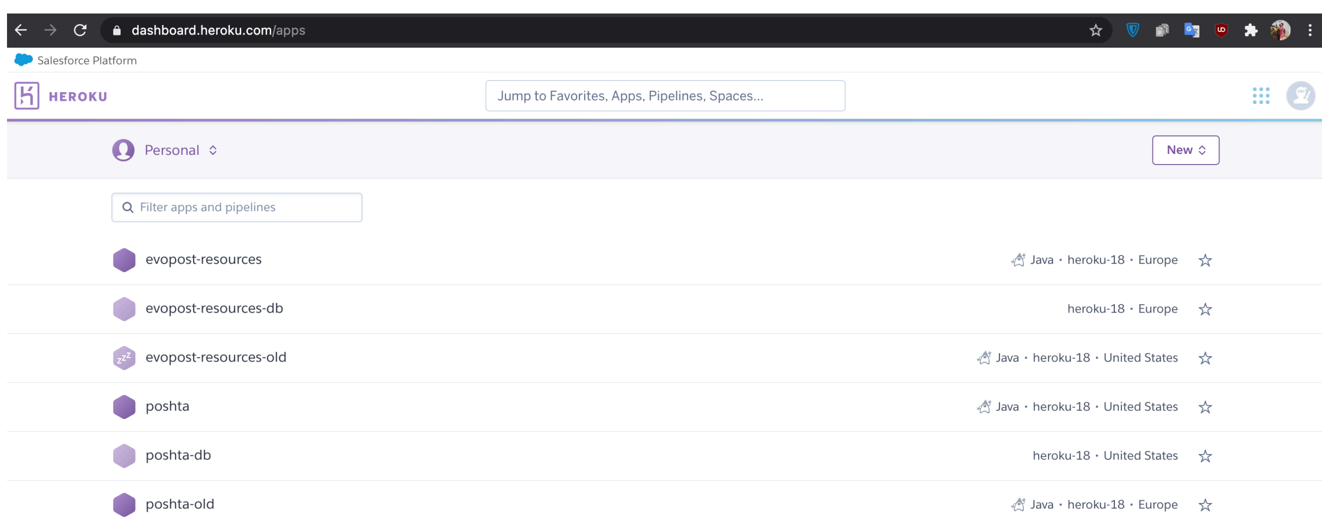



Рисунок 3.11 – Створені застосунки у Heroku

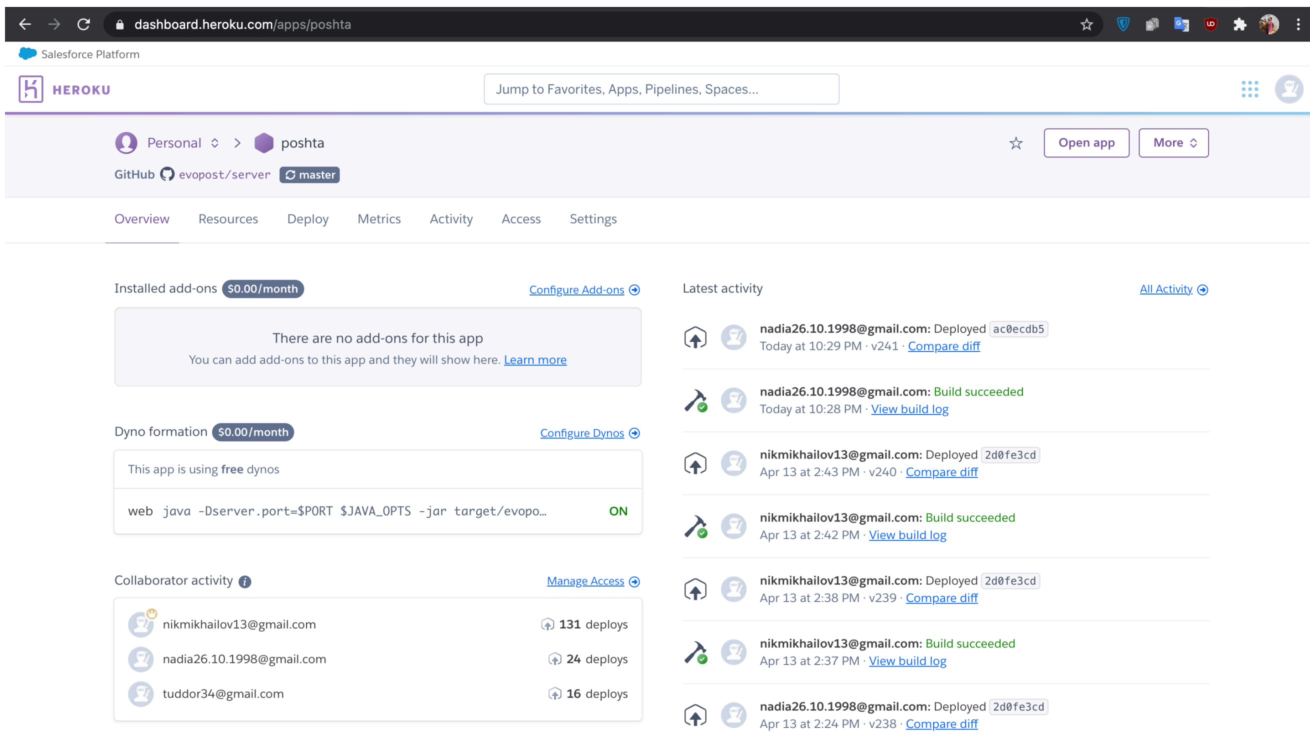



Рисунок 3.12 – Вигляд серверу Evopost у Heroku

Також було створено новий сервіс для бази даних, у якому підключено базу даних Postgres. За допомогою Liquibase міграційних схем у коді, відбуваються всі маніпуляції зі схемою даних цієї бази даних. У додатку В було наведено основний файл, де зібрані усі міграції. Сервер підключається до бази даних за допомогою конфігураційних змінних DB_USER, DB_PASSWORD, DB_DRIVER, DB_URL.

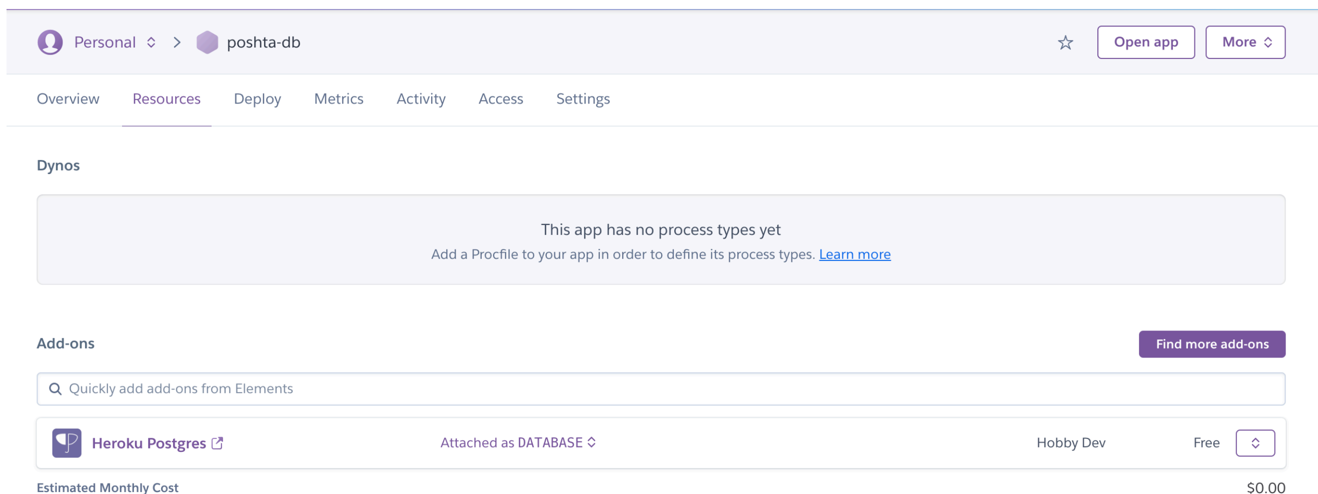


Рисунок 3.13, Рисунок 3.14 – Вигляд сервісу для бази даних у Heroku

Рисунок 3.15 – Один із файлів змін для бази даних

# 4 МІГРАЦІЯ ЗАСТОСУНКУ У GOOGLE CLOUD

## 4.1 Створення контейнера для всього застосунку

Було створено Dockerfile для міграції застосунку у хмару, у якому зазначена команда мавен mvn clean package та build. Для перевірки контейнер спершу було запущено локально за допомогою команд:

docker build -t evopost-server . та docker run -d -p 8080:8080 evopost-server.

Даний Docker файл було поміщено у репозиторій GitHub для можливості звернення до нього з платформи Google Cloud.

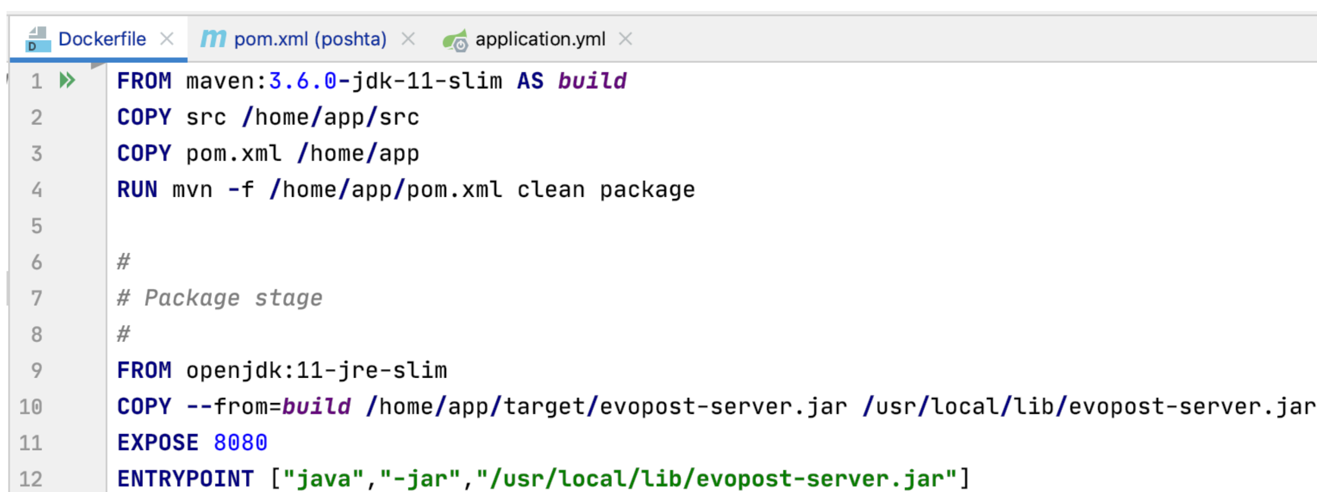



Рисунок 4.1 – Dockerfile для застосунку

## 4.2 Розгортання контейнера

Всі команди були виконані у Cloud Shell за допомогою Cloud Console. Спершу необхідно склонувати репозиторії з GitHub за допомогою команди:

git clone https://github.com/evopost/server

cd server.

Наступним кроком було створено Docker образ застосунку: docker build -t gcr.io/evopost-resources-310313/evopost-server:v1 .

Префікс gcr.io відноситься до Реєстратора контейнерів, де буде зберігатися даний образ і не завантажує образ локально.

Наступним кроком потрібно загрузити контейнер у реєстр що Kubernetes Engine зміг його загрузити і запустити :

docker push gcr.io/evopost-resources-310313/evopost-server:v1

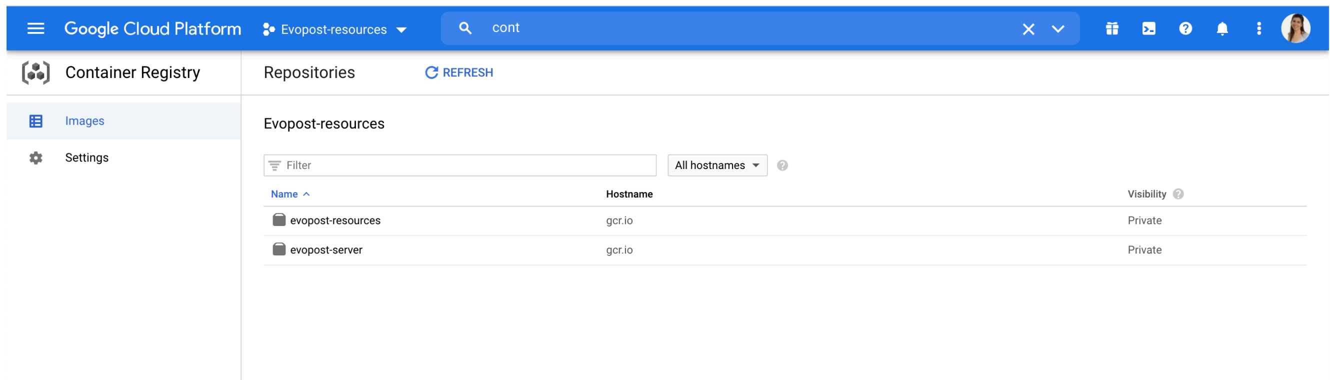



Рисунок 4.2 – вигляд контейнерів у Container Registry

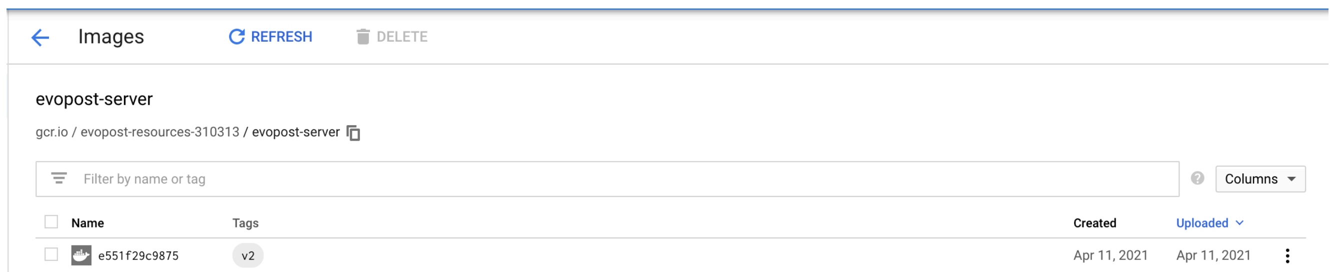



Рисунок 4.3 – Контейнер evopost-server

### 4.3 Створення кластера Kubernetes

Спершу необхідно встановити регіон, де буде розміщуватись кластер: gcloud config set compute/region europe-west-3. Далі було безпосередньо створено кластер: gcloud container clusters create-auto evopost-cluster.

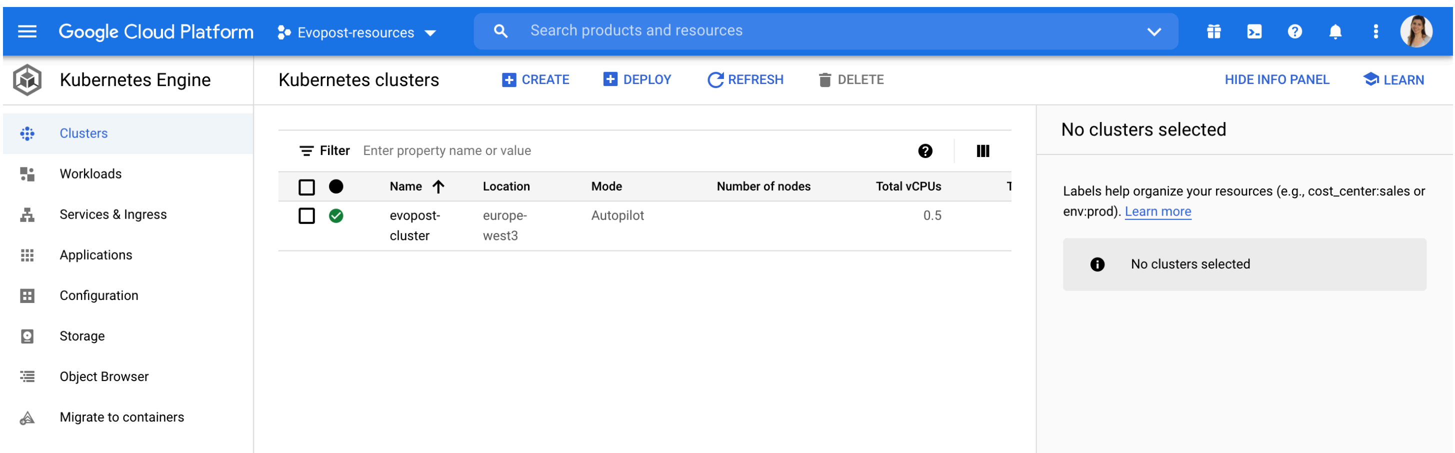



Рисунок 4.4 – Створений кластер evopost-cluster

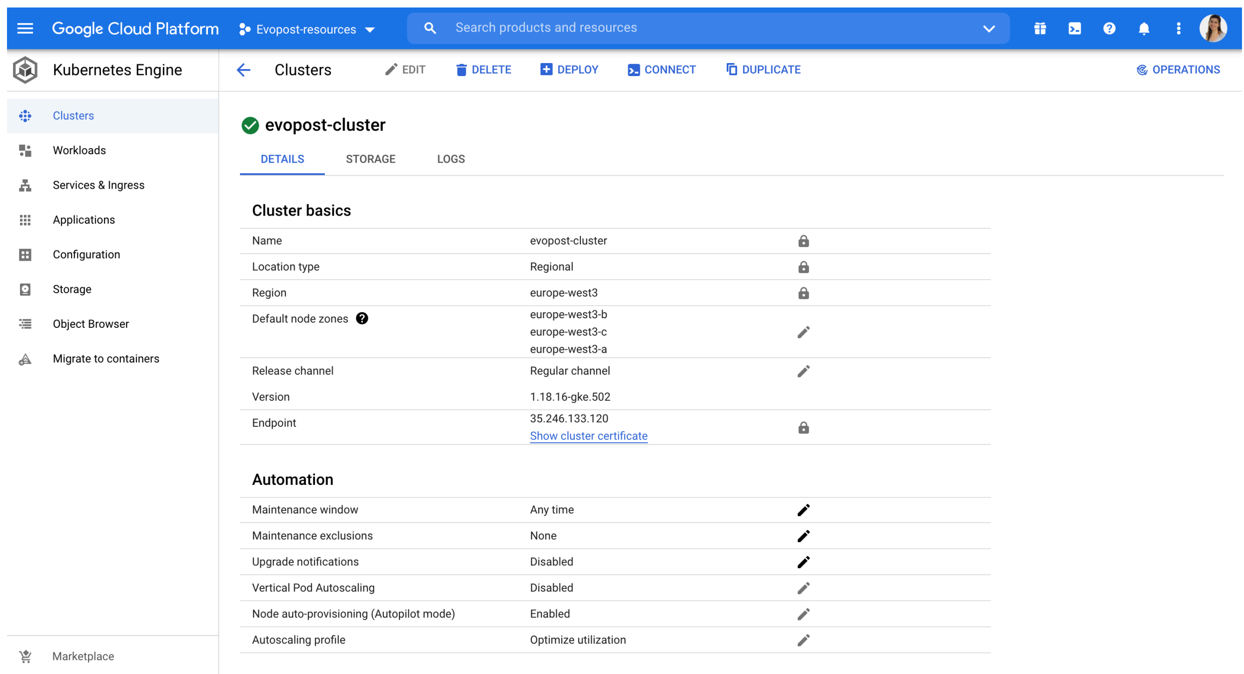



Рисунок 4.5 – Вигляд кластеру

Після завершення створення кластера можна за допомогою команди kubectl get nodes переглянути список вузлів.

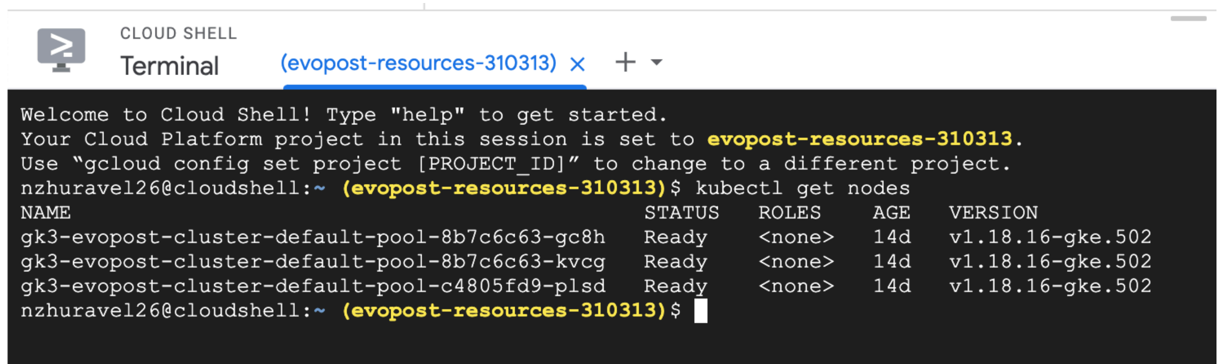



Рисунок 4.6 – Список доступних вузлів у кластері

### 4.3 Розгортання застосунку на кластері

Необхідно створити розгортання для Docker образу веб-застосунку:

kubectl create deployment evopost-resources —image=gcr.io/evopost-resources-310313/evopost-server:v1.

Наступним кроком було встановлення ресурсу для автоматичного масштабування реплік застосунку:

kubectl autoscale deployment evopost-resources --cpu-percent=80 --min=1 —max=5.

За допомогою команди kubectl get pods можна переглянути список активний под.

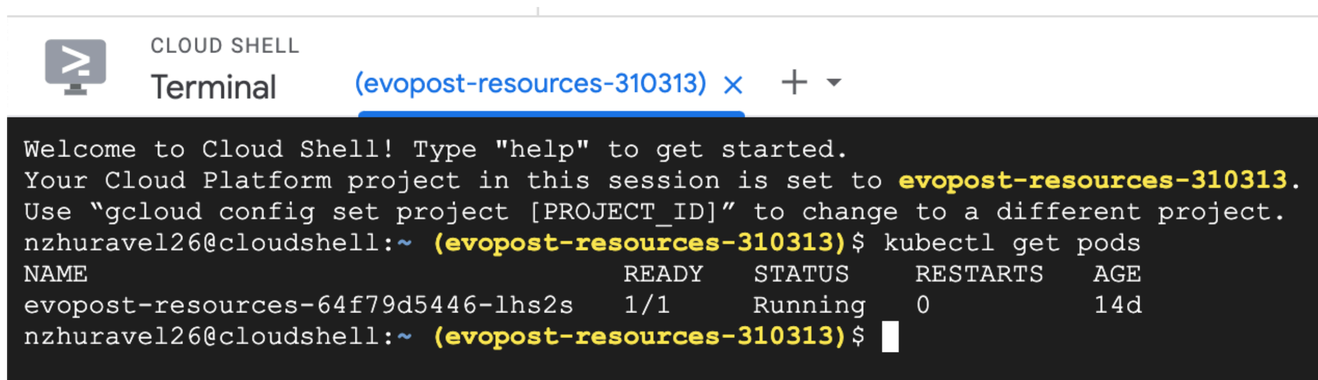



Рисунок 4.7 – Список активних под

### 4.4 Встановлення доступу до застосунку

Хоча поди мають індивідуально призначені IP-адреси, ці IP-адреси можна отримати лише зсередини кластера. Крім того, поди GKE розроблені для ефемерності, для запуску або зупинки на основі потреб масштабування. Коли пода падає через помилку, GKE автоматично перерозподіляє її, призначаючи щоразу нову IP-адресу. Це означає, що для будь-якого розгортання набір IP-адрес, що відповідають активному набору подів, є динамічним. Необхідне рішення для грування подів разом в одне статичне ім'я хосту, і виставити групу подів поза кластером в Інтернет. Сервіси Kubernetes вирішують обидві ці проблеми. Вони групують поди в одну статичну IP-адресу, доступну з будь-якого пода всередині

кластера. GKE також призначає ім'я хосту DNS цьому статичному IP. Тип сервісу за замовчуванням у GKE називається ClusterIP, де сервіс отримує IP-адресу, доступну лише зсередини кластера. Щоб виставити сервіс Kubernetes поза кластером, необхідно створити сервіс типу LoadBalancer. Цей тип сервісу створює зовнішній IP балансу навантаження для набору подів, доступних через Інтернет.

Тому за допомогою команди kubectl expose deployment evopost-resources --name= evopost-resources-service --type=LoadBalancer --port 80 --target-port 8080 було згенеровано Kubernetes сервіс. За допомогою IP-адреси можна отримати доступ до сервісу.

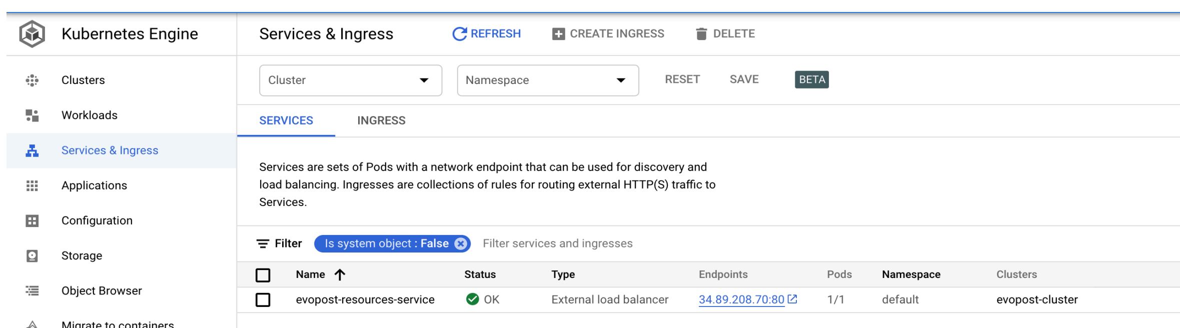



Рисунок 4.8 – Вигляд сервісу

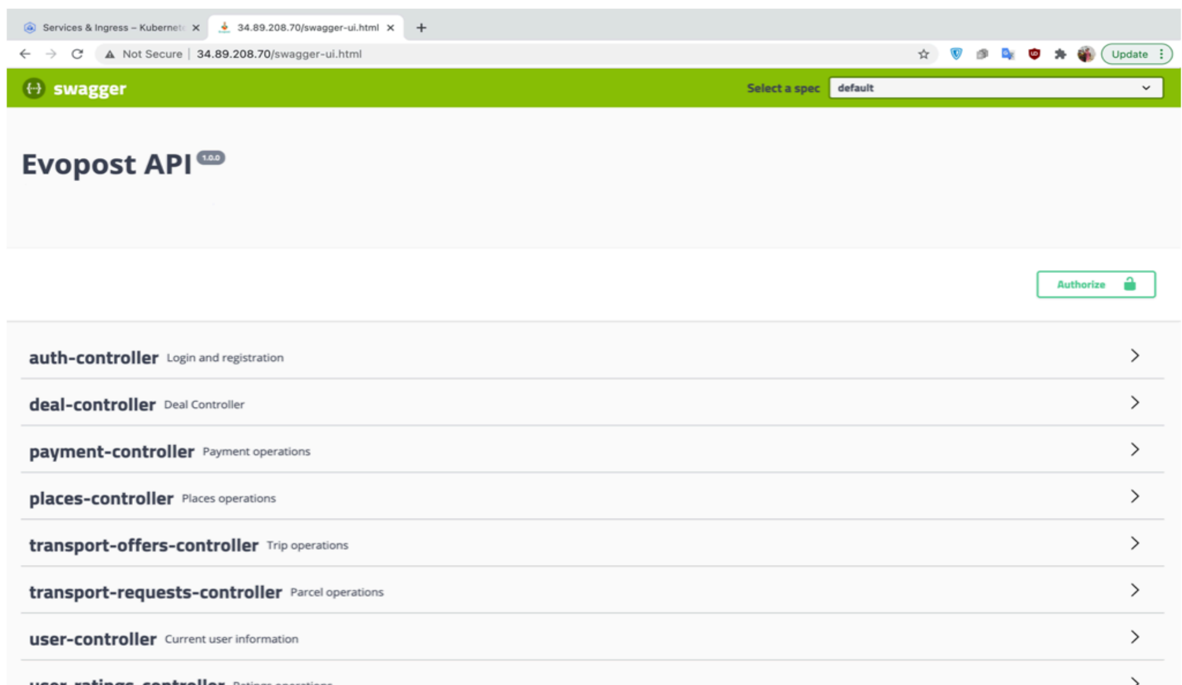



Рисунок 4.9 – Сторінка swagger-ui веб-застосунку

Рисунок 4.10 – Вигляд логів кластеру

# ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було досліджено контейнеризацію веб-застосунків, а також проаналізовано переваги застосування контейнерів, а саме Docker контейнерів. Було досліджено такі технології, як Docker та Kubernetes, які на даний момент є найактуальнішими технологіями для міграції застосунку у хмару.

У ході виконання кваліфікаційної роботи було систематизовано та поглиблено набуті протягом навчання знання в галузі веб-програмування, застосовуючи основні концепції фреймворка Spring (зокрема Spring Boot, Spring Data, Spring Security). На основі розробленого веб-застосунку для перевезення посилок, у якому збереження даних відбувається у базі даних PostgreSQL та найдійна аутентифікація користувачів забезпечена за допомогою стандарту авторизації OAuth2, було створено два докер контейнера для тестування застосунку, поміщено застусунок у GitHub репозиторії, а також була успішно виконана міграція застосунку на кластер Kubernetes у Google Cloud.

В результаті виконання поставлених задач було досліджено предметну область, набуто практичні навички програмування, застосовано теоретичні знання на практиці. При цьому було освоєно навички міграції застосунку у хмару на базі сучасних веб-технологій.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Отчет Flexera: состояние облачной отрасли в 2019 году [Електронний ресурс] // IBS DataFort. – 2019. – Режим доступу до ресурсу: https://www.datafort.ru/reports/flexera/.

2. OS-level virtualization [Електронний ресурс] // Wikipedia. – 2019. – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/OS-level_virtualization.

3. Офіційний сайт проекту Jelastic [Електронний ресурс] // Jelastic Documentation. – 2021. – Режим доступу до ресурсу: https://docs.jelastic.com/.

4. Docker (software) [Електронний ресурс] // Wikipedia. – 2021. – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Docker_(software).

5. Офіційний сайт проекту Docker [Електронний ресурс] // docker docs. – 2021 – Режим доступу до ресурсу: https://docs.docker.com/.

6. Kubernetes [Електронний ресурс] // Wikipedia. – 2021. – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Kubernetes.

7. Офіційний сайт проекту Kubernetes [Електронний ресурс] // Kubernetes Documentation. – 2021. – Режим доступу до ресурсу: Kubernetes https://kubernetes.io/docs/home/.

8. Google Cloud Platform [Електронний ресурс] // Wikipedia. – 2021. – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Google_Cloud_Platform

9. Офіційний сайт проекту Google Cloud Platform [Електронний ресурс] // Google Kubernetes Engine documentation. – 2021. – https://cloud.google.com/kubernetes-engine/docs.

# ДОДАТОК А

## Файл pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/
maven-4.0.0.xsd">


  <description>The greatest server for delivery and travelling app</description>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.13.RELEASE</version>
    <relativePath/>
  </parent>
  <groupId>com.poshta</groupId>
  <artifactId>poshta</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>poshta</name>

  <properties>
    <java.version>8</java.version>
    <swagger.version>2.9.2</swagger.version>
    <lombok.version>1.18.10</lombok.version>
    <jjwt.version>0.9.1</jjwt.version>
    <h2.version>1.4.200</h2.version>
    <postgresql.version>42.2.8</postgresql.version>
    <postgresql-testcontainers.version>1.12.3</postgresql-testcontainers.version>
    <liquibase-core.version>3.4.1</liquibase-core.version>
    <maven-checkstyle-plugin.version>3.1.0</maven-checkstyle-plugin.version>
    <assertj.version>3.15.0</assertj.version>
    <hibernate.types.vmihalcea>2.9.4</hibernate.types.vmihalcea>
    <google-api-client.version>1.30.8</google-api-client.version>
    <restfb.version>3.4.0</restfb.version>
    <client.version>5.2.1</client.version>
    <cloudinary-http44.version>1.25.0</cloudinary-http44.version>
    <cloudinary-taglib.version>1.25.0</cloudinary-taglib.version>
    <query.dsl.version>4.2.2</query.dsl.version>
    <apache.collection-utils.version>4.0</apache.collection-utils.version>
    <apache.commons.text>1.8</apache.commons.text>
    <stripe-java.version>20.41.0</stripe-java.version>
    <maven-resources-plugin.version>3.0.2</maven-resources-plugin.version>
    <mapstruct.version>1.3.1.Final</mapstruct.version>
    <request.logging.zalando>1.5.0</request.logging.zalando>
    <google-maps-services.version>0.15.0</google-maps-services.version>
  </properties>

  <dependencies>

    <!--Spring-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
```

```xml
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-websocket</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-oauth2-client</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>

    <!--Third party-->
    <dependency>
      <groupId>org.mapstruct</groupId>
      <artifactId>mapstruct</artifactId>
      <version>${mapstruct.version}</version>
    </dependency>
    <dependency>
      <groupId>org.mapstruct</groupId>
      <artifactId>mapstruct-processor</artifactId>
      <version>${mapstruct.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>${h2.version}</version>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>${postgresql.version}</version>
    </dependency>

    <dependency>
      <groupId>com.cloudinary</groupId>
      <artifactId>cloudinary-http44</artifactId>
```

```
    <version>${cloudinary-http44.version}</version>
  </dependency>

  <dependency>
    <groupId>com.cloudinary</groupId>
    <artifactId>cloudinary-taglib</artifactId>
    <version>${cloudinary-taglib.version}</version>
  </dependency>

  <!-- custom types hibernate-->
  <dependency>
    <groupId>com.vladmihalcea</groupId>
    <artifactId>hibernate-types-52</artifactId>
    <version>${hibernate.types.vmihalcea}</version>
  </dependency>

  <dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>${postgresql-testcontainers.version}</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
    <version>${liquibase-core.version}</version>
  </dependency>

  <dependency>
    <groupId>com.stripe</groupId>
    <artifactId>stripe-java</artifactId>
    <version>${stripe-java.version}</version>
  </dependency>


  <!--Authentication-->
  <dependency>
    <groupId>com.google.api-client</groupId>
    <artifactId>google-api-client</artifactId>
    <version>${google-api-client.version}</version>
  </dependency>

  <dependency>
    <groupId>com.restfb</groupId>
    <artifactId>restfb</artifactId>
    <version>${restfb.version}</version>
  </dependency>

  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>${jjwt.version}</version>
  </dependency>

  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>${swagger.version}</version>
  </dependency>

  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>${swagger.version}</version>
  </dependency>
```

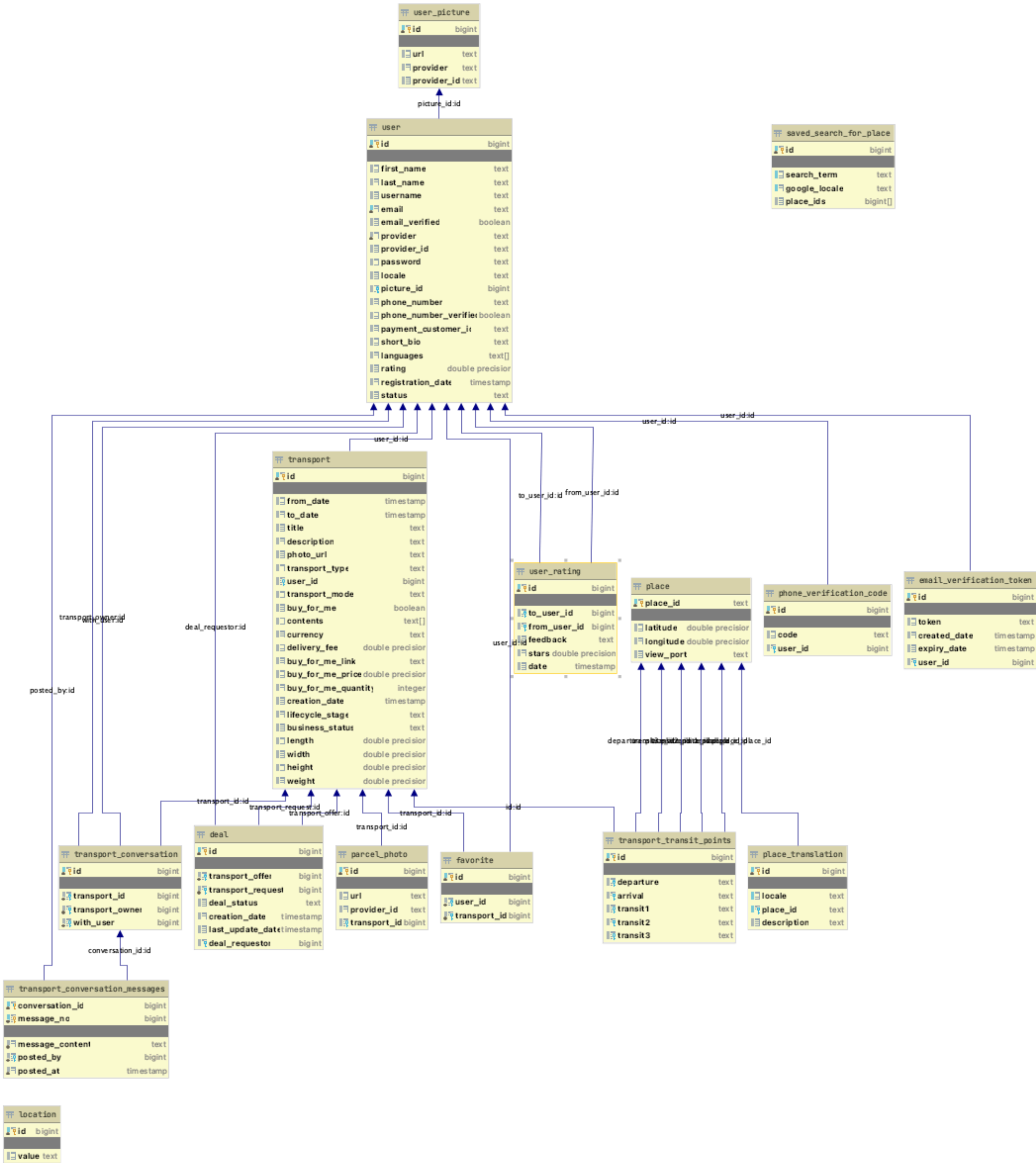
```
    <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-bean-validators</artifactId>
      <version>${swagger.version}</version>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>${lombok.version}</version>
      <scope>provided</scope>
    </dependency>

    <!-- https://mvnrepository.com/artifact/com.nexmo/client -->
    <dependency>
      <groupId>com.nexmo</groupId>
      <artifactId>client</artifactId>
      <version>${client.version}</version>
    </dependency>

    <!-- log requests and responses -->
    <dependency>
      <groupId>org.zalando</groupId>
      <artifactId>logbook-spring-boot-starter</artifactId>
      <version>${request.logging.zalando}</version>
    </dependency>

    <!-- testing dependencies -->
    <dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>${assertj.version}</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>com.querydsl</groupId>
      <artifactId>querydsl-apt</artifactId>
      <version>${query.dsl.version}</version>
    </dependency>
    <dependency>
      <groupId>com.querydsl</groupId>
      <artifactId>querydsl-jpa</artifactId>
      <version>${query.dsl.version}</version>
    </dependency>
    <!-- Collection utils -->
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-collections4</artifactId>
      <version>${apache.collection-utils.version}</version>
    </dependency>

    <dependency>
      <groupId>com.google.maps</groupId>
      <artifactId>google-maps-services</artifactId>
      <version>${google-maps-services.version}</version>
    </dependency>

    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.datatype</groupId>
      <artifactId>jackson-datatype-jsr310</artifactId>
```

```
    </dependency>

    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-text</artifactId>
      <version>${apache.commons.text}</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>${maven-checkstyle-plugin.version}</version>
        <configuration>
          <sourceDirectories>
            <sourceDirectory>${project.build.sourceDirectory}</sourceDirectory>
            <sourceDirectory>${project.build.testSourceDirectory}</sourceDirectory>
          </sourceDirectories>
          <configLocation>checkstyle.xml</configLocation>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>check</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>com.mysema.maven</groupId>
        <artifactId>apt-maven-plugin</artifactId>
        <version>1.1.3</version>
        <executions>
          <execution>
            <goals>
              <goal>process</goal>
            </goals>
            <configuration>
              <outputDirectory>target/generated-sources/java</outputDirectory>
              <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
            </configuration>
          </execution>
        </executions>
      </plugin>
<!--      copy jar for integration testing in docker-->
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>${maven-resources-plugin.version}</version>
        <executions>
          <execution>
            <id>copy-files-on-build</id>
            <phase>install</phase>
            <goals>
              <goal>copy-resources</goal>
            </goals>
            <configuration>
              <outputDirectory>${basedir}/integration-tests/poshta-server</outputDirectory>
              <resources>
                <resource>
                  <directory>target</directory>
```

```
                <includes>
                  <include>*.jar</include>
                </includes>
                <filtering>false</filtering>
              </resource>
            </resources>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
  <finalName>evopost-server</finalName>
 </build>
 <profiles>
  <profile>
    <id>use-google-repo</id>
    <repositories>
      <repository>
        <id>First thirdparty repository</id>
<url>https://maven.google.com/</ url>
      </repository>
    </repositories>
  </profile>
 </profiles>
</project>
```

# ДОДАТОК Б

## ER-діаграма проекту



# ДОДАТОК В

## Фрагмент файлу changelog-master.yaml

```yaml
- include:
   file: 001-create-schema.sql
   relativeToChangelogFile: true
- include:
   file: 002-create-domain-tables.yaml
   relativeToChangelogFile: true
- include:
   file: 003-create-hibernate-sequence.sql
   relativeToChangelogFile: true
- include:
   file: 004-create-test-user.sql
   relativeToChangelogFile: true
- include:
   file: 005-add-new-columns-to-user.yaml
   relativeToChangelogFile: true
- include:
   file: 006-drop-domain-tables.yaml
   relativeToChangelogFile: true
- include:
   file: 007-create-domain-tables.yaml
   relativeToChangelogFile: true
- include:
   file: 008-add-constraint-user.yaml
   relativeToChangelogFile: true
- include:
   file: 009-add-column-transport-mode.yaml
   relativeToChangelogFile: true
- include:
   file: 010-add-column-buy-for-me.yaml
   relativeToChangelogFile: true
- include:
   file: 011-add-column-content-types.yaml
   relativeToChangelogFile: true
- include:
   file: 012-drop-columns-from-transport-content.yaml
   relativeToChangelogFile: true
- include:
   file: 013-add-foreign-key-transport-content.yaml
   relativeToChangelogFile: true
- include:
   file: 014-add-email-verification-token-table.yaml
```

```yaml
      relativeToChangelogFile: true
  - include:
      file: 015-add-locale-to-user.yaml
      relativeToChangelogFile: true
  - include:
      file: 016-add-picture-table.yaml
      relativeToChangelogFile: true
  - include:
      file: 017-drop-column-image-url-from-user.yaml
      relativeToChangelogFile: true
  - include:
      file: 018-add-column-picture-id-to-user.yaml
      relativeToChangelogFile: true
  - include:
      file: 019-drop-columns-startingpoint-destination.yaml
      relativeToChangelogFile: true
  - include:
      file: 020-add-column-order-for-transport-transit-points.yaml
      relativeToChangelogFile: true
  - include:
      file: 021-add-phone-number-to-user.yaml
      relativeToChangelogFile: true
  - include:
      file: 022-add-phone-verification-code-table.yaml
      relativeToChangelogFile: true
  - include:
      file: 023-add-payment-customer-id-user.yaml
      relativeToChangelogFile: true
  - include:
      file: 024-short-bio-user.yaml
      relativeToChangelogFile: true
  - include:
      file: 025-add-user-languages.yaml
      relativeToChangelogFile: true
  - include:
      file: 026-add-user-rating.yaml
      relativeToChangelogFile: true
  - include:
      file: 027-add-ratings-table.yaml
      relativeToChangelogFile: true
  - include:
      file: 028-add-conversation-table.yaml
      relativeToChangelogFile: true
```