МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Інститут прикладного системного аналізу
(назва факультету, інституту)

Кафедра системного проектування
(назва кафедри)

До захисту допущено

Завідувач кафедри

__________ А.І. Петренко
(підпис) (ініціали, прізвище)

“___”_____________2017 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломного проекту (роботи) освітньо-кваліфікаційного рівня “ **спеціаліст** ”
(назва ОКР)

з напряму підготовки (спеціальності) 7.05010103, «Системне проектування».

на тему: Встановлення та налаштування платформи Docker для створення, розгортання і запуску додатків на прикладі Python Flask application

Студент групи ДА-52с Мельничук Микола ___________
(шифр групи) (прізвище, ім’я, по батькові) (підпис)

Керівник проекту к.т.н., доц. Харченко К.В. ___________
(вчені ступінь та звання, прізвище, ініціали) (підпис)

Консультанти:

нормоконтроль к.т.н., доц. Стіканов В.Ю. __________
(назва розділу ДП (ДР)) (вчені ступінь та звання, прізвище, ініціали) (підпис)

рецензент проф. д.т.н., завідувач кафедри Обчислювальної техніки ФІОТ НТУУ «КПІ» Стіренко С.Г. __________
(назва розділу ДП (ДР)) (вчені ступінь та звання, прізвище, ініціали) (підпис)

Київ – 2017

Національний технічний університет України
“Київський політехнічний інститут”

Факультет (інститут) **Інститут прикладного системного аналізу**
(повна назва)

Кафедра **Системного проектування**
(повна назва)

Напрям підготовки **050101, Комп’ютерні науки**
(код, назва)

Спеціальність **7.05010103 «Системне проектування».**
(код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри

________ А.І. Петренко
(підпис) (ініціали, прізвище)

“___” ________ 2016р.

ЗАВДАННЯ

на дипломний проект (роботу) освітньо-кваліфікаційного рівня

“спеціаліст”
(назва рівня)

студенту Мельничук Миколі Михайловичу
(прізвище, ім’я, по батькові)

1. **Тема проекту** (роботи)

Встановлення та налаштування платформи Docker для створення, розгортання і запуску додатків на прикладі Python Flask application
затверджена наказом по університету від “______________”

2. **Термін здачі** студентом закінченого проекту (роботи) “15” 01 2016р.

3. **Вихідні дані до проекту** (роботи)

нефункціональні вимоги: платформа Docker, мова програмування Python, мікрофреймворк Flask, завдання та методичні вказівки для лабораторної роботи по практичному налаштуванню Docker на прикладі Python Flask додатків;
функціональні вимоги: базовий проект Python Flask додатку з налаштованою для нього Docker платформою.
Загальні технічні вимоги до програмного продукту (Додаток 1)

(визначаються кількісні або (та) якісні показники, яким повинен відповідати об’єкт проектування наукового дослідження)

4. **Перелік питань, які мають бути розроблені** *(формулюється у повному обсязі керівником ДП (ДР) із попереднім узгодженням (за необхідності) з консультантами з окремих питань і може бути структурований за розділами (частинами): основний (-а), економічний (техніко-економічний)(-а), охорона праці тощо); формулювання питань повинно починатися словами: "Розробити...", Обґрунтувати...", "Оптимізувати...", "Провести аналіз...", "Розрахувати..." тощо)*:

- Розглянути контейнеризацію та віртуалізацію
- Проаналізувати сучасний стан технічних рішень для управління ізольованими контейнерами з додатками;
- Розглянути параметри налаштування та запуску Docker контейнерів
- Встановити та налаштувати Flask додаток на Docker платформу;

5. **Перелік графічного (ілюстративного) матеріалу**

UML діаграма Python Flask додатку (креслення)
Діаграма зв'язку Python Flask додатку та Docker платформи (креслення)
Архітектура Docker (плакат)
Docker deployment diagram (Діаграма розгортання) (плакат)
Роль Docker подібних платформ в управлінні веб додатками (плакат)
Основні висновки по роботі (плакат)

7. **Дата видачі завдання** " 08 " 09 2016р.

Керівник дипломного проекту (роботи) _________ Харченко К.В
(підпис) (ініціали, прізвище)

Завдання прийняв до виконання _________ Мельничук М.М
(підпис) (ініціали, прізвище)

ЗАТВЕРДЖУЮ

Керівник
дипломного проекту (роботи)
__________ Харченко К.В
(підпис) (ініціали, прізвище)

“___”__________ 2016р.

КАЛЕНДАРНИЙ ПЛАН-ГРАФІК

виконання дипломного проекту (роботи)

студентом Мельничук Миколою Михайловичем
(прізвище, ініціали)

№ з/п	**Назва етапів роботи та питань, які повинні бути розроблені відповідно до завдання**	**Термін виконання**	**Позначки керівника про виконання завдань**
1	Ознайомлення з технічною літературою і підготовка теоретичної частини роботи	10.09.2016 – 25.09.2016	
2	Аналіз вимог завдання, вибір методів і засобів розв’язання поставленої задачі	15.10.2016	
3	Проектування модулів пристрою, розробка моделей.	1.11.2016	
4	Тестування розроблених моделей модулів. Перевірка відповідності завданню.	1.12.2016	
5	Підготовка графічного матеріалу, оформлення пояснювальної записки, підготовка до захисту	10.01.2016	
6	Проходження нормоконтролю, отримання відгуку, рецензії, передача роботи в ДЕК	28.12.2016	
7	Захист дипломної роботи		

Студент _______________
(підпис)

АНОТАЦІЯ

дипломної роботи спеціаліста Мельничука Миколи Михайловича

на тему: «Встановлення та налаштування платформи Docker для створення, розгортання і запуску додатків на прикладі Python Flask application»

Дипломна робота присвячена дослідженню віртуалізації на рівні операційної системи, так званої контейнерної віртуалізації. Основним предметом вивчення стала система віртуалізації Docker. Результатом роботи був створений показовий Flask додаток та налаштований Docker на монтування образу та запуск контейнерів. Також методичний матеріал роботи є досить детальною інструкцією з використання можливостей системи, зі всіма кроками, командами, їх тонкощами та знімками з екрану, де це потрібно. Тому ця робота рекомендується як навчальний матеріал при вивченні Docker.

Загальний обсяг роботи 79 сторінок, з них основна частина – 72 сторінки, 23 рисунка, 2 таблиці, 16 посилань.

Перелік ключових слів: докер, віртуалізація, контейнер, образ , Docker daemon, Flask, Python, Ubuntu.

АННОТАЦИЯ

дипломной работе специалиста Мельничука Миколи Михайловича

на тему: «Установка и настройка платформы Docker для создания, развертывания и запуска приложений на примере Python Flask application»

Дипломная работа посвящена исследованию виртуализации на уровне операционной системы, так называемой контейнерной виртуализации. Основным предметом изучения стала система виртуализации Docker. Результатом работы было создано показательное Flask приложение и настроен Docker на монтирования образа и запуск контейнеров. Также методический материал работы есть достаточно подробной инструкцией по использованию возможностей системы, со всеми шагами, командами, их тонкостями и снимками с экрана, где это нужно. Поэтому эта работа рекомендуется в качестве учебного материала при изучении Docker.

Общий объем работы 79 страниц, из них основная часть - 72 страницы, 23 рисунка, 2 таблицы, 16 ссылок.

Перечень ключевых слов: докер, виртуализация, контейнер, образ, Docker daemon, Flask, Python, Ubuntu.

ANNOTATION

for the specialists work of Melnychuk Mykola Mikhailovich

«Installing and configuring Docker platform for creating, deploying and running applications with the example of Python Flask application»

Diploma work is devoted to research of virtualization at the operating system level, so-called container virtualization. The main subject of study was the Docker virtualization. In result of work, Flask application was created and Docker platform was configured to mount the image and launch containers. Also methodical material work became a very detailed instruction for use of the system, with all the steps, commands, theirs's details and screenshots where it is needed. So the work recommended as a teaching material in the study of Docker.

The total amount of work 79 pages, the main part - 72 pages, 23 figures, 2 tables, 16 links.

Keyword list: Docker, virtualization, container, image, Docker daemon, Flask, Python, Ubuntu.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ 12

ВВЕДЕННЯ 13

1 КОНТЕЙНЕРИЗАЦІЯ ТА ВІРТУАЛІЗАЦІЯ.. 15

1.1 Порівняння контейнеризації та віртуалізації 15

1.2 Проблема контейнерів — безпека 16

1.3 Інші проблеми контейнерів 17

1.4 Висновок 18

2 СУЧАСНИЙ СТАН ТЕХНІЧНИХ РІШЕНЬ ДЛЯ УПРАВЛІННЯ ІЗОЛЬОВАНИМИ КОНТЕЙНЕРАМИ З ДОДАТКАМИ 19

2.1 LXD lightervisor 19

2.1.1 LXD та Docker 19

2.1.2 Переваги LXD.. 20

2.1.3 Основні компоненти LXD 21

2.1.3.1 Контейнери 21

2.1.3.2 Знімки... 21

2.1.3.3 Образи 21

2.1.3.4 Профілі 22

2.1.3.5 Віддалене керування 23

2.1.3.5 Безпека.. 23

2.1.3.6 REST API... 24

2.1.4 Висновок 24

2.2 CoreOS — rkt 25
2.2.1 Основні особливості rkt 26
2.2.2 Модель виконання 26
2.2.3 Архітектура запуску контейнера 26
2.3 Docker 27
2.3.1 Архітектура Docker 28
2.3.2 Головні компоненти Docker 29
2.3.3 Принцип роботи Docker 30
2.3.4 Принцип роботи контейнера 31
2.3.5 Технології, використані у Docker 33
2.4 Висновок 34
3 ПАРАМЕТРИ НАЛАШТУВАННЯ ТА ЗАПУСКУ DOCKER КОНТЕЙНЕРІВ 35
3.1 Файл .dockerignore 35
3.2 Інструкції файлу dockerfile 35
3.2.1 FROM 36
3.2.2 MAINTAINER 36
3.2.3 RUN 36
3.2.4 CMD 37
3.2.5 LABEL 38
3.2.6 EXPOSE 38
3.2.7 ENV 39

3.2.8 ADD 39
3.2.9 COPY 40
3.2.10 VOLUME 40
3.2.11 USER 41
3.2.12 WORKDIR 41
3.3 Робота з docker контейнерами 42
3.3.1 Команда docker build 42
3.3.1.1 Збірка з Git репозиторю 43
3.3.1.2 Збірка з текстового файлу 43
3.3.2 Команда docker run 44
3.3.2.1 Фоновий режим (-d) 45
3.3.2.2 Звичайний режим 45
3.3.2.3 Ідентифікація контейнера 45
3.3.2.4 Налаштування мережі 46
3.3.2.5 Перевизначення параметрів образу 47
3.3.3 Допоміжні команди 48
3.4 Висновок 49
4 ВСТАНОВЛЕННЯ ТА НАЛАШТУВАНЯ FLASK ДОДАТКУ НА DOCKER ПЛАТФОРМУ 50
4.1 Flask додаток 50
4.1.1 Встановлення Flask на Ubuntu 50
4.1.2 Модифікація базового Flask додатку 52

4.2 Встановлення та перший запуск Docker на Ubuntu 53
4.3 Налаштування Docker для запуску контейнерів Flask додатку. 55
4.3.1 Файл .dockerignore 55
4.3.2 Файл dockerfile 55
4.3.3 Створення образу 59
4.3.4 Запуск та управління контейнерами 60
4.3.4.1 Налаштування мережі 60
4.3.4.2 Запуск контейнера з перенаправленим портом 61
4.3.4.3 З'єднання двох контейнерів всередині Docker мережі 63
4.3.4.4 З'єднання контейнера з локальним хостом 66
4.4 Тестування Docker 68
4.4.1 План тестування 68
4.4.2 Тестування в Ubuntu 69
4.4.3 Тестування у Windows 10 70
4.5 Висновок 70
5 Управління термінами 72
5.1 Діаграма Ганта 72
5.2 Критичний шлях 73
5.3 Управління ризиками 74
5.4 Висновок 76
ВИСНОВКИ 77
ПЕРЕЛІК ПОСИЛАНЬ 79

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Docker - інструментарій для управління ізольованими Linux-контейнерами.

ОС – операційна система

vendor-lock-in – бізнес модель, в якій встановлюється залежність користувача від продукту однієї компанії

ПЗ – програмне забезпечення

CPU – центральний процесор

Машина – Електро́нна обчи́слювальна маши́на

VM (віртуальна машина) - модель обчислювальної машини, створеної шляхом віртуалізації обчислювальних ресурсів: процесора, оперативної пам'яті, пристроїв зберігання та вводу і виводу інформації.

Гіпервізор або Монітор віртуальних машин — комп'ютерна програма або обладнання, що забезпечує одночасне, паралельне виконання декількох операційних систем на одному і тому ж комп'ютері

Daemon (демон) – клієнт докер, що доступен по REST API.

Дистрибутив — форма розповсюдження програмного забезпечення. Дистрибутив зазвичай містить програми для початкової ініціалізації системи.

Nginx — вільний веб-сервер і проксі-сервер.

HTTP — протокол передачі даних, що використовується в комп'ютерних мережах. Назва скорочена від Hyper Text Transfer Protocol, протокол передачі гіпер-текстових документів.

Python — інтерпретована об'єктно-орієнтована мова програмування високого рівня з динамічною семантикою

Flask – мікрофреймворк для своренння веб-додатків на мові програмування Python.

ВВЕДЕННЯ

Сучасний стан інформаційних технологій дозволяє створювати додатки використовуючи і поєднуючи різні технології та бібліотеки. Часом їх стає настільки багато, що з’являються проблеми зв’язані з їх розповсюдженням на доставкою до замовника чи покупця.

Перша з них — це передача продукту клієнту. Припустимо у вас є серверний проект, який вже завершений і тепер його необхідно передати користувачеві. Для цього можна приготувати багато різних файлів, скриптів або ж написати інструкцію по установці. Після цього витрачається багато часу на вирішення проблем клієнта на зразок: «у мене нічого не працює», «ваш скрипт впав на середині - що тепер робити», «я переплутав порядок кроків в інструкції і тепер не можу йти далі» і т. п

Друга — тиражування. Якщо потрібно розгорнути для одного, або, навіть, декількох клієнтів не одну, не дві, а 50 — 100 машин. Затрачати час на їх налаштування та вирішувати проблеми окремо не є дуже вдалою практикою.

Третя — перевикористання. Наприклад, маємо 3 додатки які використовують однаковий стек технологій java-tomcat-nginx-postgre. Для того, щоб запустити новий додаток на новому сервері, потрібно з нуля встановити на налаштувати увесь цей стек.

Зазвичай ці проблеми вирішуються скриптом. Для цього у файл пишуться всі необхідні команди для встановлення ПЗ, які виконуються одна за одною. Це дещо вирішує проблеми, але на заздалегідь налаштованих серверах щось може піти не так або банально система застаріла та не буде підтримувати певні рядки скрипта, що вимагатиме додаткового втручання зі сторони розробника.

Хмарні сервіси також певною мірою вирішують проблеми, можна створити з нього образ і розповсюдити по іншим серверам. Але і тут є vendor-lock-in, що не дозволить запуск на інших сервісах окрім обраного, що може не

сподобатись деяким клієнтам. Також вони не надто швидкі. Навіть, найсучасніші сервіси набагато відстають у продуктивності від стаціонарних. [1]

Віртуальні машини — одне із найкращих рішень, але вони теж мають деякі недоліки. Розмір який має образ віртуальної машини є досить великим, що не дуже подобається користувачеві. Також цей образ потрібно скачувати кожен раз, коли робляться зміни у проекті з початку. До цього він складне управління спільним використанням серверних ресурсів - не всі віртуальні машини взагалі підтримують спільне використання пам'яті або CPU.

Останнє і найсучасніше вирішення цих проблем є контейнеризація в даному випадку — докер контейнеризація. Подібно віртуальній машині докер запускає свої процеси у власній, заздалегідь налаштованої операційній системі. Але при цьому всі процеси докера працюють на фізичному хост сервері ділячи всі процесори і всю доступну пам'ять з усіма іншими процесами, запущеними в хост системі. Підхід, який використовується докер знаходиться посередині між запуском всього на фізичному сервері та повної віртуалізації, пропонованої віртуальними машинами. Цей підхід називається контейнеризацією.

1 КОНТЕЙНЕРИЗАЦІЯ ТА ВІРТУАЛІЗАЦІЯ

1.1 Порівняння контейнеризації та віртуалізації

IT-ринок насичений багатьма новими і новаторськими технологіями, що використовуються не тільки для залучення більшої кількості автоматизації, а й для подолання існуючих труднощів. Віртуалізація поставила перед собою мету принести оптимізацію і портативність в IT-інфраструктуру. Проте, технологія віртуалізації має серйозні недоліки, такі як зниження продуктивності через надважку природу віртуальних машин (VM), відсутність переносимості додатків, сповільненість надання IT-ресурсів і так далі. Таким чином, IT-галузь стабільно приступають до ери контейнеризації, яку приніс Docker. Ініціативою Docker була спеціально розроблена парадигма контейнеризації для її легшого зрозуміння і використання. Докер дозволяє спростити та пришвидшити процес контейнеризації.

Назвіть будь-яку технологічну компанію, і можна буде з упевненістю сказати, що вона теж інвестує в контейнери. Google - звичайно. IBM - так. Microsoft - безумовно. Але той факт, що контейнери зараз неймовірно популярні, ще не робить віртуальні машини застарілими. Адже це зовсім не так. Так, контейнери дозволяють вмістити набагато більше додатків на одному фізичному сервері, ніж будь-яка віртуальна машина. Технології контейнеризації начебто Docker в цьому плані легко кладуть на лопатки VM. [2]

Віртуальні машини займають багато системних ресурсів. Кожна з них містить не тільки операційну систему, але і необхідне для роботи віртуальне обладнання. А це відразу забирає досить багато оперативної пам'яті і процесорних циклів. З контейнером ситуація зовсім інша - в ньому можна розмістити тільки додаток і необхідний мінімум системних бібліотек для виконання; на це піде мінімум ресурсів.

На практиці це означає, що при використанні контейнерів ви зможете розмістити в два-три рази більше додатків на одному сервері. Крім того, контейнеризація дозволяє створювати портативний і цілісне оточення для розробки, тестування та подальшого розгортання.

1.2 Проблема контейнерів — безпека

Це найсерйозніша проблема, яка, однак, часто не береться до уваги. Деніел Уолш, інженер з безпеки з Red Hat, опублікував статтю «Порожні контейнери». Там розглядається Докер, який використовує libcontainers в якості основи. Libcontainers взаємодіє з п'ятьма просторами - процес, Мережа, Mount, Hostname, Спільна пам'ять - при роботі з Linux. При цьому є безліч важливих підсистем ядра Linux поза контейнера .

До них відносяться всі пристрої, SELinux, контрольні групи і вся файлова система всередині / SYS. Це означає, що при наявності у користувача або додатки прав суперкористувача в контейнері операційна система може бути зламана.

Є багато способів убезпечити Докер і інші технології контейнеризації. Наприклад, можна змонтувати / SYS тільки для читання, змусити процеси контейнерів працювати з певними розділами файлової системи і налаштувати мережу так, щоб можна було підключатися лише до певних внутрішнім сегментам. Але нічого з цього за замовчуванням не зроблено, і вам доведеться додатково потурбуватися цими питаннями. [2]

Інша проблема полягає в тому, що контейнеризовано додатки випускає хто завгодно. І серед розробників завжди знайдуться «погані хлопці». Якщо, наприклад, ви або ваші співробітники досить ліниві, щоб запустити на сервері перший-ліпший контейнер, то цілком можете отримати трояна. Важливо донести до співробітників думці, що не можна просто завантажити будь-які додатки з Інтернету так само, як вони це роблять на своїх смартфонах.

1.3 Інші проблеми контейнерів

Роб Хиршфельд, генеральний директор RackN, зазначив наступне: «Ідея« ізольованою коробки» допомагає вирішити частину проблем з нижчими системами (ви знаєте, що у вас там є), але ніяк не з вищестоящими (ви не знаєте, від чого залежите)».

“Поділ системи на безліч дрібніших окремих частин - хороша ідея. Але це означає і те, що управляти доведеться ще більшим числом елементів. Завжди існує якась переломна точка між декомпозицією і неконтрольованим розростанням.”

Роб Хиршфельд, генеральний директор RackN [11]

Це не тільки проблема безпеки, але і проблема забезпечення якості. Можна сказати, що X контейнерів в змозі підтримувати веб-сервер NGINX, але чи достатньо цього? Розгорнути додаток в контейнері досить легко, але якщо помилитися з вибором компонентів, то ви просто витратите час даремно.

Крім того, контейнери зазвичай прив'язують до певної версії операційної системи. Це буває корисно: немає необхідності турбуватися про залежності, якщо додаток коректно працює в контейнері. Але натомість ви отримуєте додаткові обмеження. З віртуальними машинами не важливо, який гіпервізор використовується (KVM, Hyper-V, Vsphere, Xen), - швидше за все, ви зможете запустити там будь-яку ОС. Немає проблеми запустити в VM специфічний додаток, що працює тільки на QNX. Але реалізувати це з нинішнім поколінням контейнерів стає не просто.

1.4 Висновок

Якщо необхідно запускати максимум додатків на мінімальній кількості серверів, в такому випадку краще скористатися контейнерами. Але пам'ятайте про необхідність додатково подбати про безпеку.

Якщо ж потрібно виконувати безліч додатків і / або підтримувати різні ОС - краще підійдуть віртуальні машини. І якщо питання безпеки для стоїть на першому місці, то теж краще залишитися при VM.

Вважаю, більшість з нас будуть використовувати контейнери спільно з віртуальними машинами як в хмарах, так і у власних серверах. Адже можливості економії, що відкриваються нам контейнерами, на масштабі дуже великі, щоб їх ігнорувати. А у віртуальних машин, як і раніше є чимало переваг.

2 СУЧАСНИЙ СТАН ТЕХНІЧНИХ РІШЕНЬ ДЛЯ УПРАВЛІННЯ ІЗОЛЬОВАНИМИ КОНТЕЙНЕРАМИ З ДОДАТКАМИ

2.1 LXD lightervisor

LXD - це демон, який надає REST API для управління LXC контейнерами. Головна мета: зробити щось схоже на віртуальні машини під апаратною віртуалізацією, але з використанням Linux контейнерів.

На рисунку 2.1 зображено логотип LXD.

Рисунок 2.1 – Логотип LXD

2.1.1 LXD та Docker

LXD фокусується на системних контейнерах, які ще називають інфраструктурними контейнерами. Тобто LXD маніпулює контейнерами з повноцінною Linux системою всередині так само, як ви б працювали з системою, на фізичному обладнанні або в віртуальною машиною. Традиційні системи управління конфігураціями і розгортання додатків можуть використовувати LXD так само, як і в звичайному використанні на фізичному обладнанні, віртуальній машині або хмарних середовищах. [14]

На противагу цьому, Docker фокусується на мінімалістичний, легких контейнерах, які зазвичай не оновлюють або перелаштовують, а найімовірніше будуть замінені цілком. Це робить Docker і подібні платформи більше схожими до систем розгортання додатків, ніж до інструментів управління машинами. Ці дві моделі не є взаємовиключними, ви можете використовувати LXD для надання повноцінної Linux системи, а користувачі всередині контейнера можуть за допомогою Docker ставити потрібне їм ПЗ.

2.1.2 Переваги LXD

Робота з LXC показала, що він забезпечує користувача прекрасним набором низькорівневого інструментарію і бібліотеками для створення і управління контейнерами. Однак, такий інструмент не завжди дружній до користувача і вимагає від нього багато початкових знань для розуміння як це все працює. Підтримка зворотної сумісності зі старими контейнерами і методами розгортання так само не дає використовувати деякі налаштування безпеки LXC за замовчуванням і вимагає від користувача ручної правки.

Архітектура LXD зображена на рисунку 2.2 [14]

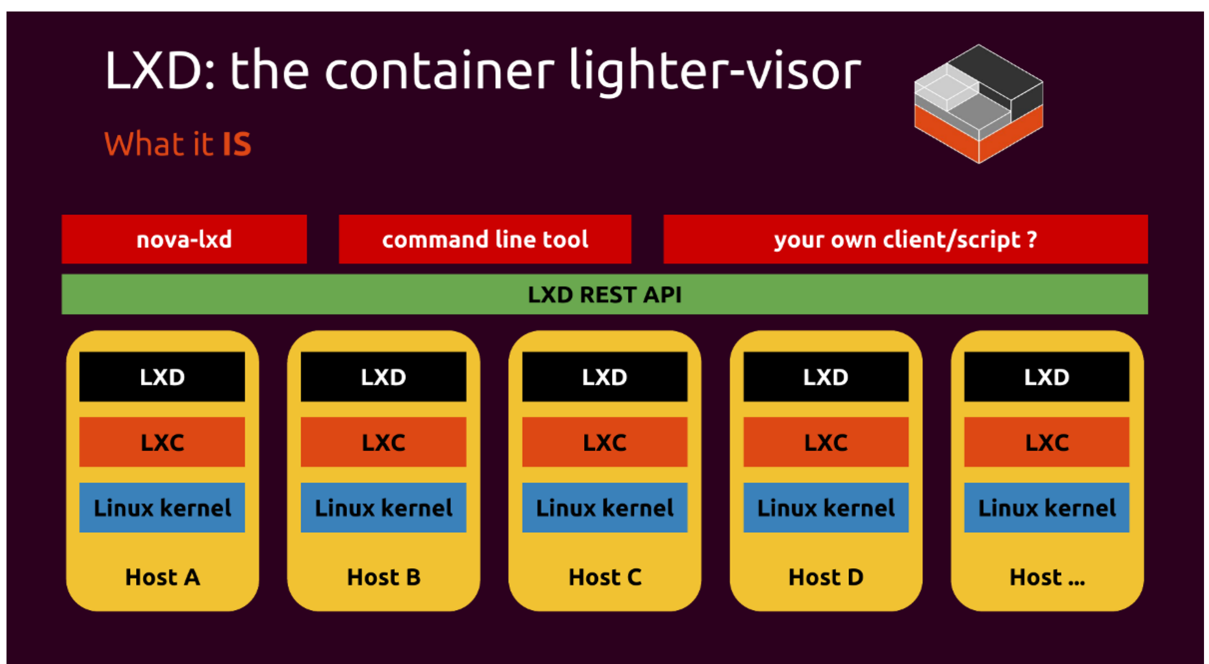

Рисунок 2.2 – Архітектура LXD

					ДА52с. 15. 0003. 001	Лист
Змн.	Арк.	№ докум.	Підпис	Дата		20

LXD - це спроба позбутися від перерахованих недоліків, до того ж LXD вирішує частину обмежень LXC в питаннях динамічного обмеження ресурсів, міграції контейнера і ефективної міграції в режимі реального часу. LXD безпечний за замовчуванням і більше орієнтований на користувача.

2.1.3 Основні компоненти LXD

2.1.3.1 Контейнери

Контейнери LXD містять:

- Файлова система (rootfs).
- Список опцій конфігурації, включаючи ліміти ресурсів. Опції безпеки, оточення і т.д.
- Перерахування пристроїв типу дисків, мережевих інтерфейсів.
- Набір профілів для контейнера звідки унаслідуються конфігурації.
- Деякі властивості даного контейнера: архітектура, ім'я
- Стан при використанні CRIU.

2.1.3.2 Знімки

Знімки контейнера незмінні в тому сенсі, що їх не можна модифікувати, крім перейменування, знищення або відновлення. Варто зазначити, що оскільки зберігається весь стан контейнера, то фактично ви маєте концепцією 'stateful' знімків. Це дає можливість відмінити зміни в контейнері, включаючи стан його CPU і пам'яті.

2.1.3.3 Образи

LXD заснований на використанні образів. Всі контейнери створені з образів. Образи зазвичай представляють собою чистий Linux дистрибутив, подібно до тих що використовується у віртуальній машині або в хмарних середовищах. Можливо публікувати (publish) контейнер, роблячи з нього образ, який вже використовується локальними або віддаленими LXD хостами.

Образи однозначно ідентифікуються своїми sha256 хешами і ви можете посилатися, використовуючи його цілком або тільки його частину. Так як набирати довгі хеші не зручно для користувача, то кожен образ має властивості, що допомагає його легко знайти в сховищі. Образу можна надати унікальне ім'я (alias).

LXD йде з трьома переднастроєними віддаленими сховищами образів:

- ubuntu: забезпечує стабільними образами Ubuntu.
- ubuntu-daily: забезпечує денними збірками Ubuntu.
- images: сервер від спільноти з образами різних дистрибутивів Linux, створені за upstream шаблонами.

Образи з віддалених серверів автоматично кешуються демоном LXD і зберігаються деякий час, за замовчуванням 10 днів. Додатково LXD автоматично оновлює образи образи, що ви використовуєте, якщо не вказано інше, так що найсвіжіша версія образу доступна локально.

2.1.3.4 Профілі

Профілі надають механізм задання конфігурації контейнера і доступних йому пристроїв в одному місці і можливість застосовувати профіль до будь-якої кількості контейнерів.

Контейнери можуть мати кілька профілів, що були примінені до нього. При побудові остаточної конфігурації, профілі застосовуються в порядку визначення, перекриваючи-перезаписуючи інформацію. Будь-яка локальна конфігурація що знаходиться в кінці, затирає абсолютно все, що йшло з профілів.

LXD йде з двома переднастроєними профілями:

- default. Автоматично застосовується до всіх контейнерів, якщо користувач не вкаже свій профіль. В даний час профіль робить тільки одну річ - визначає eth0 для контейнера.

- docker. Дозволяє вкладеність контейнерів (container nesting), робить запит на завантаження потрібних модулів ядра і кількох пристроїв в контейнері.

2.1.3.5 Віддалене керування

LXD - мережевий демон. Клієнт командного рядка дозволяє спілкуватися з декількома віддаленими LXD серверами та серверами образів.

За замовчуванням визначені:

- local: за замовчуванням зв'язок з локальним LXD демоном через unix сокет.
- ubuntu: сервер зі стабільними образами Ubuntu.
- ubuntu-daily: сервер з денними збірками Ubuntu.
- images: сервер images.linuxcontainers.org.

Можна додавати будь-яку кількість віддалених LXD хостів.

2.1.3.5 Безпека

При побудові LXD, одним з аспектів було створення безпечних контейнерів, що дозволяють запуск сучасних немодифікованих Linux.

Основні функції безпеки:

- Kernel namespaces – LXD використовує за замовчуванням більш безпечний для користувача простір імен (user namespace) на відміну від LXC (Непривілейовані контейнери). Це дозволяє гарантувати ізоляцію дій в контейнері від системи-хоста. Але залишається можливість зробити контейнер привілейованим, якщо це потрібно.
- Seccomp – фільтри для потенційно небезпечних та небажаних системних викликів.
- AppArmor – додаткові обмеження для mount, сокетов, ptrace і доступу до файлів. Обмеження міжконтейнерної взаємодії.
- Capabilities – запобігання завантаження модулів ядра, зміна часу хост-системи і т.д.

- CGroups – обмеження використання ресурсів і захист хоста від DoS з контейнера.

Замість того щоб змушувати користувача безпосередньо займатися безпекою через параметри, як це робиться в LXC, в LXD реалізована "конфігураційна мова", яка абстрагує більшість параметрів, роблячи крок до користувача. Для прикладу, можна легко попросити LXD «прокинути» пристрій хоста всередину контейнера, без ручного втручання при створенні major / minor номерів пристрої та оновленні політик cgroup. [16]

2.1.3.6 REST API

Все в LXD робиться через REST API. Немає інших способів взаємодії між клієнтом і демоном. REST API може бути доступний через локальний unix сокет, вимагаючи тільки членства в групі для перевірки автентичності, або через HTTPS, використовуючи сертифікат при автентифікації. Структура REST API, що відповідає описаним вище компонентам, проста і зрозуміла.

Коли потрібна складна схема взаємодії, LXD може спілкуватись через websockets. Це використовується в інтерактивній консолі, міграції контейнера і в повідомленнях про події. [16]

2.1.4 Висновок

LXD надає хороші інструменти командного рядка, але він не призначений для управління тисячами контейнерів на безлічі хостів. LXD базується на LXC і знаходиться на його вершині.

LXD фокусується на контейнерних системах (system container managers), тобто всередині працює повноцінна операційна система на базі Лінукс. З точки зору дизайну LXD не хвилює що працює в контейнері.

Це відрізняє LXD від Docker або Rocket, які позиціонують себе як менеджери додатків в контейнері (application container managers). Але немає нічого поганого у використанні LXD для створення повноцінного контейнера,

який буде обмежений ресурси та виконувати функції безпеки, а вже на ньому використовувати будь-який механізм дистрибуції додатків.

2.2 CoreOS — rkt

Проект CoreOS, що розвиває, засноване на ідеях контейнерної ізоляції, серверне оточення, представив випуск інструментарію управління контейнерами rkt (раніше відомий як Rocket), який позиціонується як більш безпечна, переносима і адаптована для серверного застосування альтернатива інструментарію Docker. Код rkt написаний на мові Go. [16]

Логотип rkt зображений на рисунку 2.3.

Рисунок 2.3 – Логотип rkt

2.2.1 Основні особливості rkt

- Розширені механізми забезпечення безпеки: ізоляція з використанням гіпервізора KVM, підтримка SELinux, SVirt і seccomp, інтеграція TPM (Trusted Platform Module), верифікація образів по цифровому підпису.
- Крім традиційних контейнерів на основі просторів імен (namespaces) і груп управління (cgroups) ізольовані оточення можуть створюватися і за допомогою інших технологій, включаючи звичайний chroot (контейнери fly) і віртуальне оточення Clear Linux (компактні і швидкі як звичайні контейнери, але надають більш високий рівень ізоляції за рахунок застосування гіпервізора KVM);

- Підтримка образів Docker і контейнерів, побудованих відповідно до специфікації App Container. Таким чином, rkt можна застосовувати в середовищах, які спочатку були розгорнуті за допомогою Docker. Також їх доцільно використовувати для побудови нової інфраструктури на базі формату App Container;

2.2.2 Модель виконання

Модель виконання не використовує фоновий керуючий процес, замість нього застосовуються методи поділу привілеїв для мінімізації коду, що виконується з правами root. Операції завантаження контейнера, перевірки його цілісності та верифікації по цифровому підпису виконуються під звичайним користувачем (в docker все виконується під root). [16]

Для управління контейнером використовуються засоби запуску ізольованих оточень, що надаються системами ініціалізації, такими як systemd, runit і upstart. Rkt дозволяє досягти значно вищого рівня захищеності в порівнянні з Docker, в якому всі операції проводяться за участю одного централізованого фонового процесу;

2.2.3 Архітектура запуску контейнера

Багаторівнева модульна архітектура rkt розділяє операції роботи з контейнером, окремі етапи налаштування файлової системи, підготовки виконуваного оточення і запуску додатків в контейнері.

Крім гарантування безпеки подібний підхід також дозволяє розширювати функціонал за рахунок реалізації додаткових функцій через підключення доповнень. Виділяються три основні стадії запуску контейнера:

- Нульова стадія - обробка проводиться силами утиліти rkt без залучення додаткових засобів. На даній стадії проводиться початкова підготовка контейнера: генерація UUID і маніфесту, створення файлової системи для контейнера, настройка директорій для виконання наступних стадій, копіювання виконуваного файлу першої стадії в файлову систему

контейнера, розпакування образів і копіювання додатків в директорії третьої стадії;

- Перша стадія - забезпечується окремим виконуваним файлом, що має повноваження настройки cgroups, запуску процесів і виконання операцій під користувачем root. На даній стадії здійснюється створення виконуваної групи на основі ФС, підготовленої в нульовій стадії, установка cgroups, просторів імен і точок монтування.
- Друга стадія, - проводиться безпосередній запуск програми в підготовленому контейнері. Зокрема, на даній стадії виконується процес ініціалізації вмісту контейнера, описаний в маніфесті запуску додатка (Application Manifest)

2.3 Docker

Докер - це відкрита платформа для розробки, доставки і експлуатації додатків. Docker розроблений для більш швидкого викладання додатків. За допомогою docker можна відокремити додаток від інфраструктури і звертатися з інфраструктурою як керованим додатком. Docker допомагає викладати код швидше, швидше тестувати, швидше викладати додатки і зменшити час між написанням і запуском коду. Docker робить це за допомогою легкої платформи контейнерної віртуалізації, використовуючи процеси і утиліти, які допомагають керувати і викладати програми.

Логотип Docker зображений на рисунку 2.4 [4].

Рисунок 2.4 – Логотип Docker

У своєму ядрі docker дозволяє запускати практично будь-який додаток, безпечно ізольований в контейнері. Безпечна ізоляція дозволяє запускати на одному хості багато контейнерів одночасно. Легка природа контейнера, який запускається без додаткового навантаження гіпервізора, дозволяє вам добиватися більше від заліза.

2.3.1 Архітектура Docker

Docker складається з двох головних компонентів:

- Docker: платформа віртуалізації з відкритим кодом;
- Docker Hub: платформа-як-сервіс для поширення і управління docker контейнерами.

Docker використовує архітектуру клієнт-сервер. Docker клієнт спілкується з демоном Docker, який бере на себе створення, запуск, розподіл контейнерів. Обидва, клієнт і сервер можуть працювати на одній системі, також можна підключити клієнт до віддаленого демона docker. Клієнт і сервер спілкуються через сокет або через RESTful API [5].

На рисунку 2.5 зображена архітектура Docker.

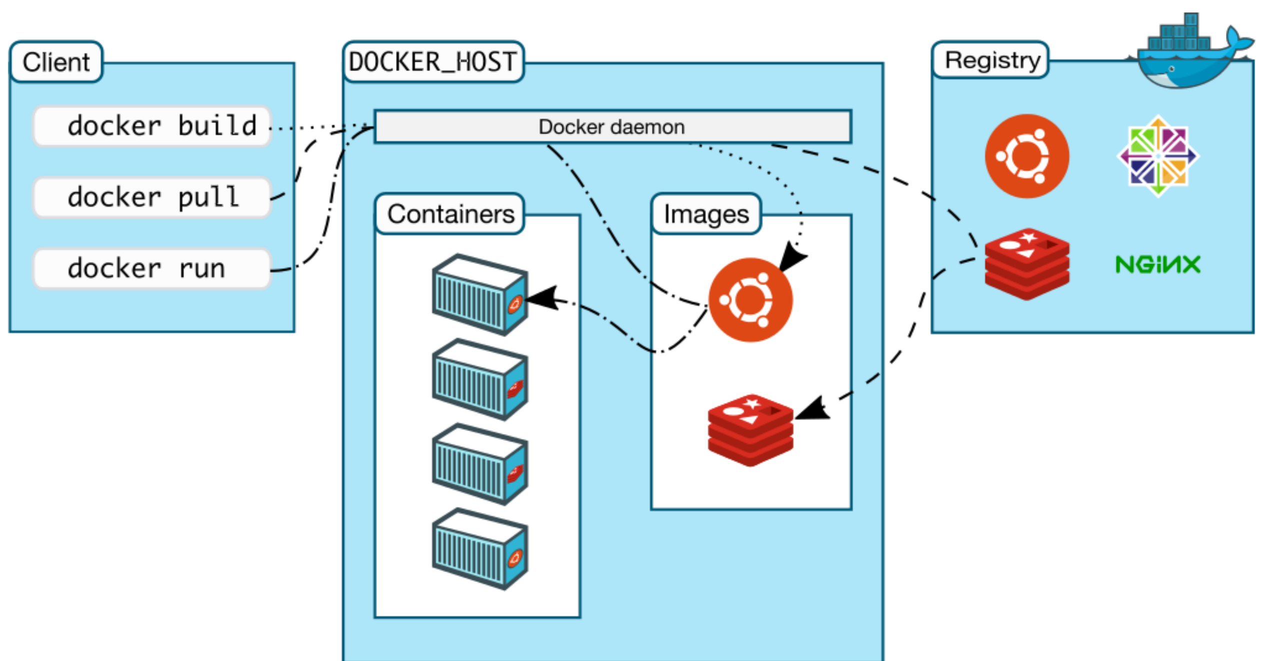

Рисунок 2.5 – Архітектура Docker

Як показано на діаграмі, демон запускається на хост-машині. Користувач не взаємодіє з сервером на пряму, а використовує для цього клієнт. Docker-клієнт - головний інтерфейс до Docker системи. Він отримує команди від користувача і взаємодіє з docker-демоном.

2.3.2 Головні компоненти Docker

Щоб розуміти, з чого складається docker, потрібно знати про три його компоненти:

- образи (images)
- реєстр (registries)
- контейнери

Docker-образ - це read-only шаблон. Наприклад, образ може містити операційну систему Ubuntu з Apache і додатком на ній. Образи використовуються для створення контейнерів. Docker дозволяє легко створювати нові образи, оновлювати існуючі, або можна завантажити образи створені іншими людьми. Образи - це компонента збірки docker-а.

Docker-реєстр зберігає образи. Є публічні і приватні реєстри, з яких можна скачати або завантажити образи. Публічний Docker-реєстр - це Docker

Hub. Там зберігається величезна колекція образів. Образи можуть бути створені вами або можна використовувати образи створені іншими користувачами. Реєстри - це компонента поширення.

Контейнери схожі на директорії. У контейнерах міститься все, що потрібно для роботи програми. Кожен контейнер створюється з образу. Контейнери можуть бути створені, запущені, зупинені, перенесені або видалені. Кожен контейнер ізольований і є безпечною платформою для додатка. Контейнери - це компонента роботи.

Виходячи з цих трьох компонентів в Docker можна:

- створювати образи, в яких знаходяться додатки;
- створювати контейнери з образів, для запуску додатків;
- розповсюдженню через Docker Hub або інший реєстр образів.

2.3.3 Принцип роботи Docker

Отже образ - це read-only шаблон, з якого створюється контейнер. Кожен образ складається з набору рівнів. Docker використовує union file system для поєднання цих рівнів в один образ. Union file system дозволяє файлам і директоріями з різних файлових систем (різних гілок) прозоро накладатися, створюючи когерентну файлову систему.

Одна з причин, по якій docker легкий - це використання таких рівнів. Коли змінюється образ, наприклад, проходить оновлення додатку, створюється новий рівень. Так, без заміни всього образу або його перезібрання, як вам можливо доведеться зробити з віртуальною машиною, тільки рівень додається або оновлюється. І вам не потрібно роздавати весь новий образ, публікується тільки оновлення, що дозволяє поширювати образи простіше і швидше.

В основі кожного образу знаходиться базовий образ. Наприклад, ubuntu, базовий образ Ubuntu, або fedora, базовий образ дистрибутива Fedora. Так само можна використовувати готові образи як базу для створення нових образів. Наприклад, образ apache можна використовувати як базовий образ для веб-додатків. Docker зазвичай бере образи з реєстру Docker Hub.[16]

Docker образи можуть створитися з цих базових образів, кроки опису для створення цих образів називаються інструкціями. Кожна інструкція створює новий образ або рівень. Інструкціями будуть наступні дії:

- запуск команди
- додавання файлу або директорії
- створення змінної оточення
- вказівки що запускати коли запускається контейнер цього способу

Ці інструкції зберігаються в файлі Dockerfile. Docker зчитує цей Dockerfile, коли збирається образ, виконує ці інструкції, і повертає кінцевий образ.

Реєстр - це сховище docker образів. Після створення образу ви можете опублікувати його на публічному реєстрі Docker Hub або на вашому особистому реєстрі. За допомогою docker клієнта ви можете шукати вже опубліковані образи і завантажувати їх на машину з docker для створення контейнерів.

Docker Hub надає публічні і приватні сховища образів. Пошук і скачування образів з публічних сховищ доступний для всіх. Вміст приватних сховищ не попадає в результат пошуку. І тільки ви і ваші користувачі можуть отримувати ці образи і створювати з них контейнери.

2.3.4 Принцип роботи контейнера

Контейнер складається з операційної системи, призначених для користувача файлів і метаданих. Відомо, що кожен контейнер створюється з образу. Цей образ говорить docker-у, що знаходиться в контейнері, який процес запустити, коли запускається контейнер та інші конфігураційні дані. Docker образ доступний тільки для читання. Коли docker запускає контейнер, він створює рівень для читання / запису зверху образу (використовуючи union file system, як було зазначено раніше), в якому може бути запущено додаток.

Або за допомогою програми docker, або за допомогою RESTful API, docker клієнт говорить docker-демону запустити контейнер.

$ sudo docker run -i -t ubuntu /bin/bash

Давайте розберемося з цією командою. Клієнт запускається за допомогою команди docker, з опцією run, яка говорить, що буде запущений новий контейнер. Мінімальними вимогами для запуску контейнера є такі атрибути:

- який образ використовувати для створення контейнера. У нашому випадку *ubuntu*
- команду яку ви хочете запустити коли контейнер буде запущений. У нашому випадку */bin/bash*

Після запуску цієї команди Docker, по порядку, робить наступне [16]:

- завантажує образ ubuntu: docker перевіряє наявність образу ubuntu на локальній машині, і якщо його немає - то викачує його з Docker Hub. Якщо ж образ є, то використовує його для створення контейнера;
- створює контейнер: коли образ отриманий, docker використовує його для створення контейнера;
- ініціалізує файлову систему і монтує read-only рівень: контейнер створений в файлової системі і read-only рівень доданий образ;
- ініціалізує мережу / міст: створює мережевий інтерфейс, який дозволяє docker-у спілкуватися хост машиною;
- Установка IP адреси: знаходить і задає адресу;
- Запускає вказаний процес: запускає програму;
- Обробляє та видає вихід додатку: підключається і залоговує стандартний вхід, вихід і потік помилок додатку, щоб можна було відслідковувати як працює програма.

Тепер у вас є робочий контейнер. Ви можете управляти своїм контейнером, взаємодіяти з вашим додатком. Коли вирішите зупинити додаток, видаліть контейнер.

2.3.5 Технології, використані у Docker

Докер написаний на мові Go і використовує деякі можливості ядра Linux, щоб реалізувати наведений вище функціонал.

Docker використовує технологію namespaces для організації ізольованих робочих просторів, які називаються контейнерами. Коли запускається контейнер, docker створює набір просторів імен для даного контейнера. Це створює ізольований рівень, кожен контейнер запущений в своєму просторі імен, і не має доступ до зовнішньої системи.

Список деяких просторів імен, які використовує docker:

- **pid**: для ізоляції процесу;
- **net**: для управління мережевими інтерфейсами;
- **ipc**: для управління IPC ресурсами. (ICP: InterProccess Communication);
- **mnt**: для управління точками монтування;
- **utc**: для ізолювання ядра і контролю генерації версій (UTC: Unix timesharing system).

Control groups (контрольні групи). Docker також використовує технологію cgroups або контрольні групи. Ключ до роботи додатка в ізоляції, надання додатку тільки тих ресурсів, які йому потрібно. Це гарантує, що контейнери будуть добре співіснувати. Контрольні групи дозволяють розділяти доступні ресурси заліза і якщо необхідно, встановлювати межі і обмеження. Наприклад, обмежити можливу кількість пам'яті, що використовується контейнером. [4]

Union File Sysem або UnionFS - це файлова система, яка працює створюючи рівні, що робить її дуже легкою і швидкою. Docker використовує UnionFS для створення блоків, з яких будується контейнер. Docker може використовувати кілька варіантів UnionFS включаючи: AUFS, btrfs, vfs і DeviceMapper [10].

Docker поєднує ці компоненти в обгортку, яку ми називаємо форматом контейнера. Формат, який використовується за умовчанням, називається libcontainer. Так само docker підтримує традиційний формат контейнерів в Linux

с допомогою LXC. В майбутньому Docker можливо буде підтримувати інші формати контейнерів. Наприклад, інтегруючись з BSD Jails або Solaris Zones.

2.4 Висновок

Зараз у світі IT існує досить багато технологій для роботи з контейнерною віртуалізацією. Спостерігається великий прогрес вже існуючих та нових платформ для контейнеризації. Найбільші компанії світу цікавляться ними та настроєні на подальшу роботу з ними. Але наразі залишаються відкритими деякі проблеми в даних технологіях, тому ще багато має бути покращено з часом.

Найпопулярніша контейнерна технологія — Docker. За останні роки її використання значно збільшилось і стало “мейнстрімом” у світі розробки та розгортання додатків. LXD не являється прямим конкурентом через різний підхід та мету цих технологій. А от платформа rkt може конкурувати з Docker, але вона відносно нова та не популярна. Та з часом вона можливо стане досить стабільною та інноваційною, щоб стати на озброєння у розробників та замінити Docker.

3 ПАРАМЕТРИ НАЛАШТУВАННЯ ТА ЗАПУСКУ DOCKER КОНТЕЙНЕРІВ

3.1 Файл .dockerignore

Перед докером CLI відправляє контекст в Docker демон, він шукає файл з ім'ям .dockerignore в кореневому каталозі контексту. Якщо цей файл існує, то CLI змінює контекст для виключення файлів і каталогів, що збігаються з шаблонами в ньому. Це допомагає уникнути відправки лишніх великих або конфіденційних файлів і каталогів до демона щоб вони не були випадково додані за допомогою ADD або COPY.

CLI інтерпретує файл *.dockerignore* як список шаблонів для ігнорування файлів та папок, який є досить подібним до файлу *.gitignore*. [4]

В даній роботі слід відразу помістити всі непотрібні докеру файли в цей файл, наприклад:

- якщо використовувати скриптові файли з розширенням sh, варто помістити рядок:

  **.sh*
- Якщо в проекті використовується git, можна помістити:

  /.gitignore

  /readme.md
- Також файли які вивористовує IDE, наприклад

  /.idea

3.2 Інструкції файлу dockerfile

Докер може будувати образ автоматично, читаючи інструкції з dockerfile. Dockerfile є текстовий документ, який містить всі команди які необхідні для запуску додатку, включаючи командний рядок. Всі команди з файлу, окрім команд запуску, відпрацьовують при створені образу [4].

3.2.1 FROM

FROM <образ>: <тег>

FROM інструкція встановлює базовий образ для наступних інструкцій. Таким чином, дійсний Dockerfile повинен мати від її першою. Базовим образом може бути будь-який інший. Найкраще його брати з публічного репозиторію такого як DockerHub. FROM повинна бути першою інструкцією, що не є коментарем в Dockerfile.

Інструкція FROM може з'являтися кілька разів в межах одного Dockerfile для створення декількох образів, які йдуть один за одним. Але потрібно записати ID образу з логів докеру перед наступним виконанням FROM.

В записі інструкції тег не є обов'язковими. Якщо опустити його, то підчас виконання файлу буде використаний останній. Білдер повертає помилку, якщо вона заданий образ чи тег не є дійсний.

Примітка! Використавши тег <latest> поверне останню версію

3.2.2 MAINTAINER

MAINTAINER <ім'я>

Інструкція MAINTAINER дозволяє залишити автора створеного файлу та образу.

3.2.3 RUN

Інструкція RUN має 2 форми:

- RUN <команда> (*shell* форма, команда запускається в оболонці за замовчуванням */bin/sh -c* на Linux)
- RUN ["виконуваний_файл", "параметер1", "параметер2"] (EXEC форма)

Команда RUN виконуватиме будь-які команди в новому *шарі* поверх поточного образу і закомітить результат. В результаті закомічений образ буде використовуватися для наступного кроку в Dockerfile.

Розшарування інструкції RUN і генеруючи коміти відповідає основним поняттям Докер, де коміти дешеві і контейнери можуть бути створені з будь-якої точки в історії образу, в так само, як в системах управління версіями як git чи svn.

Форма *exec* дозволяє уникнути зміну команди додаванням оболонки і виконувати команди, використовуючи базовий образ, який не містить команди оболонки.

В *shell* формі, можна використовувати символ \ (зворотний слеш), щоб продовжити одну команду RUN на наступний рядок.

Кеш-пам'ять для інструкцій RUN не видаляється автоматично при наступній збірці. Кеш для інструкції як *RUN apt-get dist-upgrade -y* буде повторно при наступній збірці. Кеш-пам'ять для інструкцій RUN може бути видалена за допомогою прапорця *—no-cache*.

Приклади використання інструкції RUN:

- *RUN apt-get update -y*
- *RUN apt-get install -y python3-pip*
- *RUN mkdir app*
- *RUN [“python3”, “script.py”]*

3.2.4 CMD

Інструкція CMD має три форми:

CMD ["виконуваний_файл", "параметр1", "param2"] (EXEC форма)

CMD ["param1", "param2"] (в якості стандартних для Entrypoint)

CMD команда param1 param2 (форма *shell*)

В dockerfile може бути тільки одна команда CMD. Якщо є більш однієї CMD тоді тільки остання CMD буде виконана. Основною метою CDM є забезпечення запуску основної програми додатку при запуску контейнера. Значення параметрів можуть включати в себе виконуваний файл.

Якщо використовувати *shell* форму, тоді команда буде виконана з додаванням у початок строки "*/bin/sh -c"*, щоб виконати команду без оболонки, потрібно використати *exec* форму.

Приклади використання:

- *FROM ubuntu*
  CMD echo "This is a test." | wc -
- *FROM ubuntu*
  CMD ["/usr/bin/wc","--help"]
- *CMD python3 hello.py*

3.2.5 LABEL

LABEL <ключ> = <значення> <ключ> = <значення>

Команда LABEL додає метадані до зображення які виглядають як пари ключ-значення. Приклади:

- *LABEL organization="KPI"*
- *LABEL version="1.0"*
- *LABEL name="Ivan" lastname="Ivanov"*

Переглянути метп-дані можна через команду *docker inspect id_контейнера в JSON форматі*

"Labels": {
"organization": "KPI",
"version"="1.0",
"name"="Ivan",
"lastname"="Ivanov"
}

3.2.6 EXPOSE

EXPOSE <порт> [<порт> ...]

Інструкція EXPOSE інформує Docker, що контейнер слухає на зазначених мережевих портах під час виконання. EXPOSE не робить порти контейнера

доступними для хоста. Для цього необхідно використовувати або прапорець -p, щоб відкрити діапазон портів або прапорець -P щоб відкрити всі порти. Можна виставити один номер порту і відкрити його зовні під іншим номером.

3.2.7 ENV

ENV <ключ> <значення>

ENV <ключ> = <значення> ...

Команда ENV встановлює змінну оточення <ключ> на значення <значення>. Це значення буде перебувати в середовищі всіх "нащадків" Dockerfile команд і можуть заміняти ключ на значення за тодомогою $*ключ*.

Інструкція ENV має дві форми. Перша форма, ENV <ключ> <значення>, буде встановлювати одну змінну. Весь рядок після першого пропуску буде розглядатися як <значення>.

Друга форма, ENV <ключ> = <значення> ..., дозволяє кілька змінних, які будуть встановлені в один час. Зверніть увагу на те, що друга форма використовує знак рівності (=) в синтаксисі, в той час як перша форма не робить.

Приклади використання:

- *ENV name = Ivan lastname = “Ivanov Ivanovych”*
- *ENV name Ivan*
  ENV lastname Ivan Ivanovych

Обидва варіанти дадуть одне і теж саме на виході.

3.2.8 ADD

ADD має дві форми:

ADD <sr > ... <dest>

ADD ["<src>", ... "<dest>"] (ця форма потрібна для шляхів, що містять пробіли)

Інструкція ADD копіює нові файли, каталоги або URL-адреси віддаленого файлу з <SRC> і додає їх в файлову систему образу по шляху <dest>. Також, якщо у нас є файл з розширенням *tar,* то ADD скопіює і розархівує його.

Декілька ресурсів <src> можуть йти одні за одним, і якщо це файли або каталоги, то вони повинні бути відносними до вихідного каталогу, який будується (контекст збірки). Кожен елемент <src> може містити групові символи і узгодження. Наприклад:

- *ADD hom* /mydir/*
- *ADD hom?.txt /mydir/*

3.2.9 COPY

COPY має дві форми:

COPY <src> ... <dest>

COPY ["<SRC>", ... "<Dest>"] (ця форма потрібна для шляхів, що містять пробіли)

Команда копіює нові файли або каталоги з <src> і додає їх в файлової системі контейнера по шляху <dest>.

ADD і COPY функціонально подібні, взагалі кажучи, є кращим COPY. Це тому, що це більш прозора інструкція, ніж ADD. COPY підтримує тільки основне копіювання локальних файлів в контейнер, в той час як ADD має деякі особливості (наприклад, розархівування *tar* і віддаленої підтримки URL), які не відразу очевидні.

3.2.10 VOLUME

VOLUME ["/ дані"]

Інструкція VOLUME створює точку монтування з вказаним ім'ям і позначає його як зовнішній змонтований том з хоста або інших контейнерів. Значення може бути в форматі JSON масиву,

VOLUME ["/var/db/"],

або простий рядок з декількома аргументами, такими як

VOLUME /var/db/

Команда *docker run* ініціалізує новостворений volume з будь-якими даними, яка існує в зазначеному місці в межах базового зображення.

Наприклад, розглянемо наступний фрагмент коду Dockerfile:

FROM ubuntu

RUN mkdir / myvol

RUN echo "hello world" > /myvol/greeting

VOLUME /myvol

В результаті цього докерфайлу в зображенні, яке викличе *docker run*, створиться нова точка монтування на /*myvol* і файл *greeting* скопіюється туди.

3.2.11 USER

USER admin

Інструкція USER встановлює ім'я користувача або UID використовувати при виконанні в образі будь-якого RUN, CMD і Entrypoint інструкцій, які йдуть нижче нього в dockerfile.

3.2.12 WORKDIR

WORKDIR /шлях_до_робочої_папки

Інструкція WORKDIR встановлює робочий каталог для будь-яких RUN, CMD, Entrypoint, COPY і ADD інструкцій, які слідують за нею в dockerfile. Якщо WORKDIR не існує, він буде створений за замовчуванням в кореневому каталозі, навіть якщо він ніде не використовується. Він може бути використаний кілька разів в одному dockerfile. Наприклад:

WORKDIR /

RUN mkdir app

WORKDIR /app

RUN touch readme.txt

3.3 Робота з docker контейнерами

3.3.1 Команда docker build

Дана команда існує для того, щоб будувати та перебудовувати образи. Загальний вигляд команди:

docker build [ОПЦІЇ] PATH | URL | -

Опції:

--build-arg value	Встановлює аргументи при збиранні образу
--cgroup-parent string	Змінює cgroup- parent для контейнера
--cpu-period int	Обмежує використання процесору в періодах
--cpu-quota int	Обмежує використання процесору в квотах
-c, --cpu-shares int	Вага сумісного використання процесору
--disable-content-trust	Пропускає верифікацію образу (по замовч. true)
-f, --file string	Назва докер файлу (по замов. 'PATH/Dockerfile')
--force-rm	Завжди видаляти проміжні образи.
--help	Допомога
--isolation string	Технологія ізолювання
--label value	Встановити мета дані (зо замовч. [])
-m, --memory string	Обмеження пам’яті
--no-cache	Не використовувати кеш при будуванні образу
--pull	Завжди витягувати найновіші образи
-q, --quiet	Не показувати логи будування
--rm	Видаляти проміжні контейнери (по замовч. true)
-t, --tag value	Назва образу та тег 'name:tag' формат
--ulimit value	Ulimit опція (по замовч [])

Docker будує зображення з Dockerfile і "контексту". Контекст - це файли, розташовані в зазначеному шляху (PATH) або URL. Параметр URL може посилатися на три види ресурсів: Git репозиторіїв, створені tarball і прості текстові файли [4].

3.3.1.1 Збірка з Git репозиторю

Коли параметр вказує URL на місце розташування сховища Git, репозиторій виступає в якості контексту збірки. Система рекурсивно клонує репозитарій і його підмодулів за допомогою git *clone --depth 1 --recursive* команди. Ця команда працює в тимчасовому каталозі на локальному хості. Після успішного виконання команди, каталог відправляється в daemon в якості контексту. Місцеві клони дають вам можливість отримати доступ до приватних сховищ за допомогою локальних облікових даних користувачів, VPN, і так далі.

Git URL-адреси приймають конфігурацію контексту в їх фрагмент секції, розділені двокрапкою :. Перша частина являє собою посилання, що Git буде перевірити, це може бути або гілку, тег, або комміт. Друга частина являє собою піддиректорію теки сховища, який буде використовуватися в якості контексту збірки.

Наприклад, запустіть цю команду, щоб використовувати каталог з ім'ям докер в контейнері:

$ docker build https://github.com/docker/rootfs.git#container:docker

3.3.1.2 Збірка з текстового файлу

Замість того, щоб вказати контекст, ви можете передати одну Dockerfile в URL або передати файла в через STDIN. Для запуску Dockerfile з STDIN потрібно:

$ docker build - < Dockerfile

Якщо ви використовуєте STDIN або вказуєте URL, що вказує на текстовий файл, система використовує вказаний dockerfile, і будь-яка -f, --file опція ігнорується.

За замовчуванням команда *docker build* буде шукати dockerfile в корені контексту збірки. -f, --file, Опція дозволяє вказати шлях до альтернативного файлу, щоб використовувати замість стандартного. Це корисно в тих випадках, коли той же набір файлів використовуються для кількох варіантів збірки. Шлях

до файлу повинен бути в контексті збірки. Якщо відносний шлях заданий, то він інтерпретується по відношенню до кореня контексту.

Якщо клієнт Docker втрачає зв'язок з daemon, збірка буде скасована. Це відбувається, якщо перервати клієнта docker з CTRL-C або, якщо клієнт Docker закривається з якої-небудь причини.

3.3.2 Команда docker run

Докер запускає процеси в ізольованих контейнерах. Контейнер являє собою процес, який працює на хості. Хост може бути локальним або віддаленим. Коли оператор виконує *docker run*, процес у контейнері ізольований і працює з власною файловою системою, мережею і його особисте дерево процесів ізольоване від хоста [4].

Основна команда *docker run* приймає таку форму:

$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

Команда *docker run* необхідно вказати образ з якого запустити контейнер. Розробник образу може по замовчуванню встановити такі параметри:

- запуск в фоновому чи звичайному режимі
- ідентифікація контейнера
- мережеві налаштування
- обмеження CPU і пам'яті

З *docker run [OPTIONS]* оператор може додати або перевизначити параметри за замовчуванням образу, встановлені розробником. І, крім того, оператори можуть перевизначати майже всі параметри за замовчуванням, встановлені самому середовищі виконання docker. Здатність оператора перевизначити образ і Docker run time параметри дає йому більше можливостей, ніж будь-які інші команди Docker.

При запуску контейнера Docker, ви повинні спочатку вирішити, якщо ви хочете запустити контейнер у фоновому режимі в "відключеному" режимі або в режимі переднього плану за замовчуванням:

3.3.2.1 Фоновий режим (-d)

Для запуску контейнера в фоновому режимі, можна використати -d=true або просто -d. За своєю конструкцією, контейнери, запущені у фоновому режимі зупиняються тільки тоді, коли процес зупиняється всередині контейнера. Контейнер в фоновому режим не може бути автоматично видаленим, коли він зупиняється, це означає, що ви не можете використовувати --rm опцію з опцією -d.

Для введення / виводу з фонового контейнера використовуйте мережеві з'єднання або загальних розділи (volumes). Це потрібно, оскільки контейнер більше не слухається командного рядка, де була запущена *docker run*.

Для того, щоб приєднатися до контейнера у фоновому режимі, використовуйте команду *docker attach*.

3.3.2.2 Звичайний режим

У звичайному режимі за замовчуванням, коли -d не вказано), *docker run* запускає процес в контейнері і прикріпляє консоль до стандартного вводу, виходу і виходу помилок (STDIN, STDOUT, STDERR). Він навіть може симулювати термінал і проводити сигнал [10]. Все це налаштовуєтьс:

-a = []: Приєднати до `` STDIN`, STDOUT` і / або `STDERR`

-t: симулювати термінал (TTY)

--sig-proxy Proxy всі отримані сигнали процесу (не в режимі TTY)

-i: залишати STDIN відкритим, навіть якщо не приєднаний до контейнера

Для інтерактивних процесів (наприклад, *shell*), ви повинні використовувати -i -t разом для того, щоб виділити TTY для процесу контейнера. -i -t часто пишеться -it.

3.3.2.3 Ідентифікація контейнера

Оператор може ідентифікувати контейнер трьома способами:

- UUID long ідентифікатор

- UUID short ідентифікатор
- Ім’я (name)

Ці ідентифікатори UUID приходять від daemon. Якщо ви не привласнити ім'я контейнера з опцією *--name*, то демон генерує випадкове ім'я для вас. Визначення імені може бути зручним способом, щоб додати значення контейнеру. Якщо ви вкажете ім'я, ви можете використовувати його при зверненні контейнера в межах мережі Docker. Це працює як для фонових так і звичайних контейнерів Docker.

І, нарешті, щоб допомогти з автоматизації, ви можете записати Docker ідентифікатор контейнера в файл за вашим вибором з *—cidfile=""*.

3.3.2.4 Налаштування мережі

--dns=[] Використати особливий DNS сервер

--network: Параметр мережі контейнера

--network-alias=[] : Дати скорочене ім’я контейнера в мережі

--add-host="": Додати рядок в /etc/hosts (host:IP)

--mac-address="": Встановити MAC адресу для контейнера

--ip="" : Встановити IPv4 адресу для контейнера

--ip6="" : Встановити IPv6 адресу для контейнера

За замовчуванням всім контейнери дозволена мережа, і вони можуть зробити будь-які вихідні з’єднання. Оператор може повністю відключити мережу використавши *docker run --network none*, що відключає всі вхідні і вихідні мережеві з’єднання [4].

Публікація портів і з’єднання з іншими контейнерами працює тільки з *`bridge`* мережею. Ви завжди повинні віддати їй перевагу.

За замовчуванням, MAC-адресу генерується з використанням IP-адреси, в контейнері. Ви можете встановити MAC-адресу контейнера в явному вигляді за допомогою параметра *—mac-address*. Але варто знати, що Docker не перевіряє задані вручну MAC-адреси на унікальність.

Підтримувані мережі:

- none - мережа відсутня.
- bridge (default) — з’єднання з контейнером через до моста через інтерфейси.
- host — використати мережевий стек хоста.
- container:<name|id> - використати мережевий стек іншого контейнера
- NETWORK — підключитись до створеної в докері мережі

3.3.2.5 Перевизначення параметрів образу

Коли розробник створює образ з dockerfile, розробник може встановити ряд параметрів за замовчуванням, які вступають в силу, коли зображення запускається в якості контейнера [16].

Чотири з команд dockerfile не можуть бути перевизначені під час виконання: FROM, MAINTAINER, RUN і ADD. Все інше має відповідне перевизначення в *docker run*.

CMD

Нагадаємо, додаткову команду в командному рядку Docker:

$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

Ця команда не є обов'язковим, тому що людина, яка створила образ, можливо, вже забезпечили *COMMAND* за замовчуванням з використанням команди dockerfile CMD. Оператор може перевизначити цю команду CMD просто вказавши нову команду *COMMAND*. В наступному прикладі заміть *CMD* замінить */bin/bash*:

docker run ubuntu /bin/bash

ENV

Крім того, оператор може встановити будь-яку змінну оточення в контейнері, використовуючи один або кілька *-e* прапорів, навіть відкидаючи ті, які згадані вище, або вже визначені розробником з dockerfile ENV:

docker run -e ”collor=red” ubuntu /bin/bash

USER

`root` (ID = 0) є користувачем за замовчуванням в контейнері. Розробник зображення може створювати додаткових користувачів. Розробник може встановити користувача за замовчуванням для запуску першого процесу з інструкцією USER в dockerfile. При запуску контейнера, оператор може перевизначити команду USER, передавши параметр -u.

-u = "", --user = "": Встановлює ім'я користувача або UID, і додатково ім’я_группи або GID для зазначеної команди.

Всі приклади наведені нижче дійсні:

--user=[user | user:group | uid | uid:gid | user:gid | uid:group]

WORKDIR

За замовчуванням робочий каталог для запуску виконуваних файлів в контейнері це кореневий каталог (/), але розробник може встановити інше значення за замовчуванням за допомогою команди dockerfile WORKDIR. Оператор може перевизначити з:

-w="": Робоча папка всередині контейнера

3.3.3 Допоміжні команди

1) Список всіх існуючих контейнерів (не тільки працючих):

   docker ps -a

2) Зупинити всі запущені контейнери:

   docker stop $(docker ps -a -q)

3) Видалити всі існуючі контейнери:

   docker rm $(docker ps -a -q)

   Якщо якісь контейнери як і раніше працює в якості daemon, використовуйте -f (примусово) після *rm*.

4) Видалити всі існуючі зображення

   docker rmi $(docker images -q -a)

5) Приєднати до працюючого контейнера

   docker attach назва_чи_UUID_контейнера

6) Використання логів Docker

docker logs -f назва_чи_UUID_контейнера

3.4 Висновок

Розглянуті файли — налаштування, та команди досить часто використовуються при роботі з Docker. Але їх налаштування вимагає багато зусиль, щодо тонкостей роботи кожної інструкції чи прапорця для команд терміналу. Функціонал досить обширний та дозволяє зручно налаштовувати, відлагоджувати та використовувати всю систему.

4 ВСТАНОВЛЕННЯ ТА НАЛАШТУВАНЯ FLASK ДОДАТКУ НА DOCKER ПЛАТФОРМУ

4.1 Flask додаток

4.1.1 Встановлення Flask на Ubuntu

Встановити мікрофреймворк Flask на Ubuntu не є складним, якщо python3 і pip вже встановленні. Якщо на машині ще немає python потрібно його встановити використавши наступні інструкції:

1. Перейти до суперкористувача та оновити індекси пакетів
   $ sudo apt-get update
2. Встановити пакет python3-pip
   $ sudo apt-get install python3-pip
3. Після цього *python* та *pip* вже встановленні, але версія *pip*, що надходить разом з *python* низької версії, тому його потрібно оновити за допомогою:
   $ sudo pip3 install --upgrade pip
4. Після цього пакети будуть готові до використання

Встановити Flask можна за допомогою команд:

$ sudo apt-get update

$ sudo pip3 install Flask

Для того, щоб перевірити роботу Flask, на офіційному сайті фреймворку є код пітонівського скрипту, який дозволяє запустити сервер, що повертає на сторінку «Hello World!» [16]. Для цього треба перейти в будь-яку робочу дерикторію та створити файл з назвою «*hello.py*». У файл варто помістити наступний код:

from flask import Flask

```
app = Flask(__name__)


@app.route("/")
def hello():
    return "Hello World!"


if __name__ == "__main__":
    app.run()
```

При цьому дуже важливо зберігати всі відступи, тому що мова Python чутлива до них. Саме тому тут немає фігурних дужок, і код завжди виглядає відформатованим.

Запусти даний скрипт можна за допомогою команди:

```
$ python hello.py
```

Після цього можна буде побачити вивід (рис. 4.1).

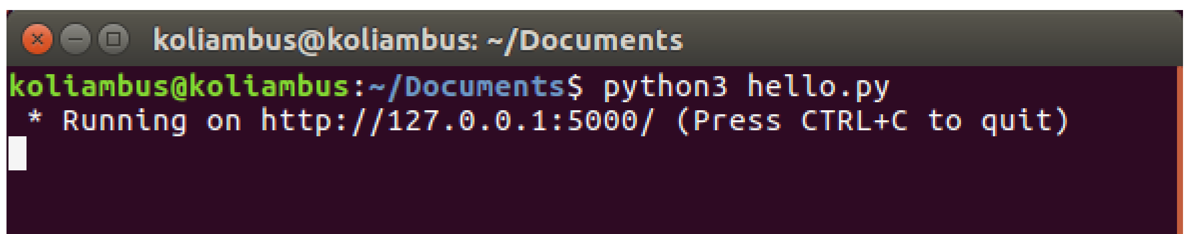

Рисунок 4.1 – Результат запуску Flask додатку

Перевірити роботу сервера можна перейшовши за посиланням (рис. 4.2).

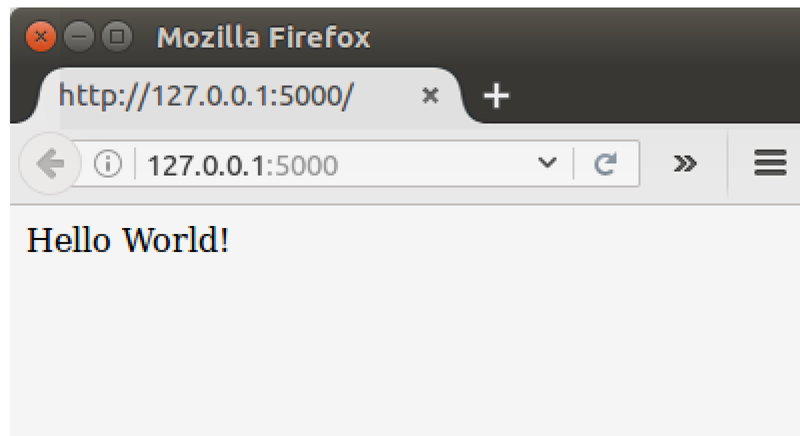

Рисунок 4.2 – Результат запуску Flask додатку в браузері

4.1.2 Модифікація базового Flask додатку

Вихідний код кінцевого варіанту додатку можна знайти за адресою https://github.com/koliambus/diploma . Базовий Flask додаток, описаний вище був перероблений для імплементації REST api методів. Також він був реструктуризований, щоб більше нагадувати ООП парадигму мови Python.

Клас *Config* містить змінні, що використовуються у багатьох місцях, тому вони є статичними для легкого доступу:

- container_number – номер контейнера в який поміщено додаток
- default_port – порт додатку за замовчуванням
- app – посилання на ядро Flask додатку.

Клас *Parser* – містить методи для зчитування парамерів запуску додатку в командному рядку:

- Метод *Parser::parse_container()* зчитує номер контейнера

Клас *Runner* – містить єдиний метод для запуску Flask додатку:

- Статичний метод *run(name, application)* – запускає *application*, якщо запущенно з головного рядка скрипту

Клас *RoutesConfigurator* є складовою частиною REST api, налаштовує реакцію додатку на перехід по посиланьнням. REST api:

- “/” - використовується для перевірки роботи. Повертає "Flask Dockerized, my container number = номер_контейнера"

- "*/hello/<int:container>*" - використовується для відповіді від про приєднання від іншого контейнера. Повертає "Hello from container #" + номер_вхідного_контейнера + " through container #" + номер_цього_контейнера
- *"/connect/<string:address>"* - приєднується до подібного додатку за адресою *address*. Повертає результат з'єднання.

Клас *HttpConnector* містить метод для з'єднання по HTTP протоколу.

В основнову тілі скрипту створюється Flask додаток, налаштовується REST та запускається додаток через вищезгадані класи.

Config.app = Flask(__name__)

RoutesConfigurator.configure(Config.app)

Runner.run(__name__, Config.app)

Цих модифікацій буде достатньо для того, щоб продемонструвати основні можливості Docker.

4.2 Встановлення та перший запуск Docker на Ubuntu

Для встановлення Docker на Ubuntu потрібно вміти користуватись терміналом та пройти усі кроки з офіційного документу за посиланням https://docs.docker.com/engine/installation/linux/ubuntulinux/ [4]. Сторінку також можна знайти загугливши "Docker Ubuntu"

Одразу після встановлення Docker на комп'ютер, варто налаштувати без рутовий доступ до docker, тому що його початкову версію потрібно запускати тільки під суперкористувачем. Для цього в терміналі:

1. Зайдіть під суперкористувачем
2. Створіть групу docker

   $ sudo groupadd docker
3. Додайте вашого локального користувача до docker групи

$ sudo usermod -aG docker ім'я_користувача

4. Тепер сервіс docker можна запустити з вашого локального користувача, для цього вийдіть з-під суперкористувача та зайдіть під локальним.

   exit

5. Для запуску docker сервісу пропишіть:

   service docker start

   Якщо ваш локальний користувач має пароль, ubuntu може його запросити двічі для запуску необхідних бібліотек *docker.socket, docker.service*. Для цього пароль суперкорисувача вже не потрібен.

6. Для перевірки запуску, docker має заздалегідь встановлені образи, один з яких ми запустимо.

   docker run hello-world

   Після запуску цей контейнер залишить по собі коротку лог-інструкцію з використання. Якщо ви побачили "Hello from Docker!" (рис. 4.3), значить docker готовий до використання.

```
koliambus@koliambus: ~
koliambus@koliambus:~$ su
Password:
root@koliambus:/home/koliambus# sudo groupadd docker
root@koliambus:/home/koliambus# sudo usermod -aG docker koliambus
root@koliambus:/home/koliambus# exit
exit
koliambus@koliambus:~$ service docker start
koliambus@koliambus:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

koliambus@koliambus:~$
```

Рисунок 4.3 – Запуск docker сервісу з-під локального користувача

7. Для зупинки docker сервісу введіть

   service docker stop

4.3 Налаштування Docker для запуску контейнерів Flask додатку.

Отже наразі у нас вже є готовий Flask додаток з мінімальною REST api та налаштованим на http:// GET запит до подібного додатку. До того, в ньому є змінна, яку потрібно передати на стадії запуску сервера. Також є встановлена та запущена Docker платформа, яку потрібно налаштувати на роботу з вищезгаданим додатком.

4.3.1 Файл .dockerignore

В даній роботі слід відразу помістити всі непотрібні докеру файли в цей файл, наприклад:

- якщо використовувати скриптові файли з розширенням sh, варто помістити рядок:
  **.sh*
- Якщо в проекті використовується git, можна помістити:
  /.gitignore
  /readme.md
- Також файли які використовує IDE, наприклад
  /.idea

4.3.2 Файл dockerfile

Для початку потрібно вибрати базовий образ, на основі якого буде працювати додаток Flask. Нехай в даній роботі ми будемо працювати на останній версії Ubuntu, наразі останньою версією являється 16.04 xenial. Для цього першим рядком dockerfile повинно бути:

FROM ubuntu:latest

Тут під тегом *latest* розуміється остання версія образу. Остання версія базового образу буде встановлена, навіть якщо не вказувати тег і просто написати

FROM ubuntu

Для того, щоб позначити автора чи того, хто буде підтримувати систему після інструкції *FROM* потрібно використати MAINTAINER. Нехай форматом вводу буде «*Ім'я Прізвище НомерГрупи email*». Наприклад:

MAINTAINER Mykola Melnychuk DA-52c "koliambus@ukr.net"

Наступним кроком буде інструкція *ENV,* яка буде використана, для того, щоб помітити кожен контейнер через окремий код для перевірки http з'єднання

між різними контейнерами. Для цього наддамо змінній *container_number* значення за замовчуванням «1»:

ENV container_number 1

Ця змінна буде використана для запуску Flask додатку.

Для роботи з Flask мікрофреймворком потрібно створити нову папку в контейнері, для того, щоб не розгортати всі файли та папки з вихідним кодом в корінь папки, де знаходяться системні папки Ubuntu. Для цього достатньо використати інструкцію RUN з параметрами:

RUN mkdir app

Папка, що міститиме всі файли додатку створена і тепер потрібно перенести всі файли додатку з поточної файлової системи до папки, що буде знаходиться в папці контейнера. Для цього варто використати інструкцію COPY вона просто прозоро скопіює файли не роблячи змін. Якби в нас був архівний файл з розширенням *tar* тоді для нього варто було б використати інструкцію ADD, як вміє розпаковувати такі архіви.

COPY . /app

Після назви інструкції стоїть «.» (крапка). Вона означає те, що всі файли з поточного місця розташування варто скопіювати. Але, звичайно, не будуть скопійовані ті файли, які входять до складу файлу *.dockerignore.* Файли будуть скопійовані до папки */app* яка знаходиться на образі контейнеру.

Для подальших дій варто змінити робочу папку на /*app* для більш зручної роботи з додатком. Для цього треба скористуватися інструкцією

WORKDIR /app

Для роботи Flask потрібно щоб на машині, як на тій, що розробляє додаток, так і віртуальній було встановлено пакети *python* та *pip.* Для цього використаємо інструкцію *RUN* та *shell* командний рядок:

RUN apt-get update -y

RUN apt-get install -y python3-pip

В цих рядках використано прапорець *-y* для програми *apt-get,* який автоматично дає згоду на встановлення пакетів.

Як і при встановленні на машину розробника *pip3* потрібно оновити до останньої версії:

RUN pip3 install --upgrade pip

На відміну від машини розробника, при встановленні Flask ми не будемо користуватись командним рядком для безпосереднього встановлення. Так як Flask сам по собі являється мікрофреймворком, то для його роботи потрібно буде встановлювати нові пакети при використанні нових функцій, якщо пакетів буде багато, то перерахувати їх в командному рядку не дуже практично. Для цього варто створити файл *requirements.txt* в якому будуть перечисленні всі пакети, розділені кінцем рядку. В даній роботі цей файл буде виглядати просто (рис. 4.4).

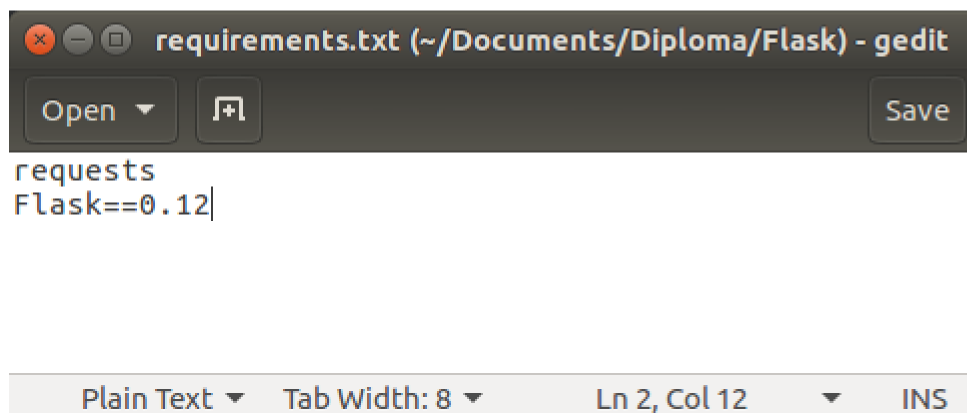

Рисунок 4.4 – Приклад файлу *requirements.txt*

Тепер цей файл можна використати при встановленні пакетів *pip.* Для цього знову ж таки використаємо інструкцію RUN.

RUN pip3 install -r requirements.txt

Де прапорець *-r* означає використання файлу з вимогами.

Для спілкування контейнерів між собою в середині Docker потрібно налаштувати номер порта, який буде експортуватися поза межі контейнера, щоб процес в середині контейнера було видно під цим портом. Для цього підійде

інструкція *EXPOSE.* Так як у Flask додатку порт по замовчуванню *5000,* то він і буде використаний:

EXPOSE 5000

На даний момент всі пакети встановленні, файли знаходяться у правильному місці, змінні готові до використання та робоча папка налаштована на корінь додатку. Тепер залишається запустити в роботу пітонівський скрипт. Для того щоб запускати основний процес під час запуску контейнера у докерфайлі передбачена інструкція *CMD*. Її і варто використати у даному випадку.

CMD python3 hello.py -C $container_number

В даному рядку встановлений python3 запускає скрипт під назвою hello.py . -C означає для додатку використання номеру контейнера. Після нього йде сам номер контейнера, який стоїть під назвою змінної навколишнього середовища. Тобто під час виконання цієї команди *$container_numbre* буде замінено на 1, і вийде: «*python3 hello.py -C 1»*

В загальному готовий файл dockerfile буде виглядати наступним чином:

```
FROM ubuntu:latest
MAINTAINER Mykola Melnychuk DA-52c "koliambus@ukr.net"
ENV container_port 1
RUN mkdir app
COPY . /app
WORKDIR /app
RUN apt-get update -y
RUN apt-get install -y python3-pip
RUN pip3 install --upgrade pip
RUN pip3 install -r requirements.txt
EXPOSE 5000
CMD python3 hello.py -C $container_number
```

4.3.3 Створення образу

На даному етапі роботи, файли, що потрібні для роботи docker та Flask готові. Подальшою дією потрібно створити образ з нашого *dockerfile* для того. Щоб його можна було в будь-який момент запустити на машині. Для цього потрібно використати команду

docker build -t diploma:latest .

Ця команда запускає створення образу (рис. 4.5) та надає йому, при успішному виконанні тег *diploma:latest*, що стоїть одразу після прапорця *-t*. В кінці команди стоїть шлях до основної папки з *dockerfile*, наразі це «.» (крапка) тому що командний рядок знаходиться в каталозі з вищезгаданим dockerfile.

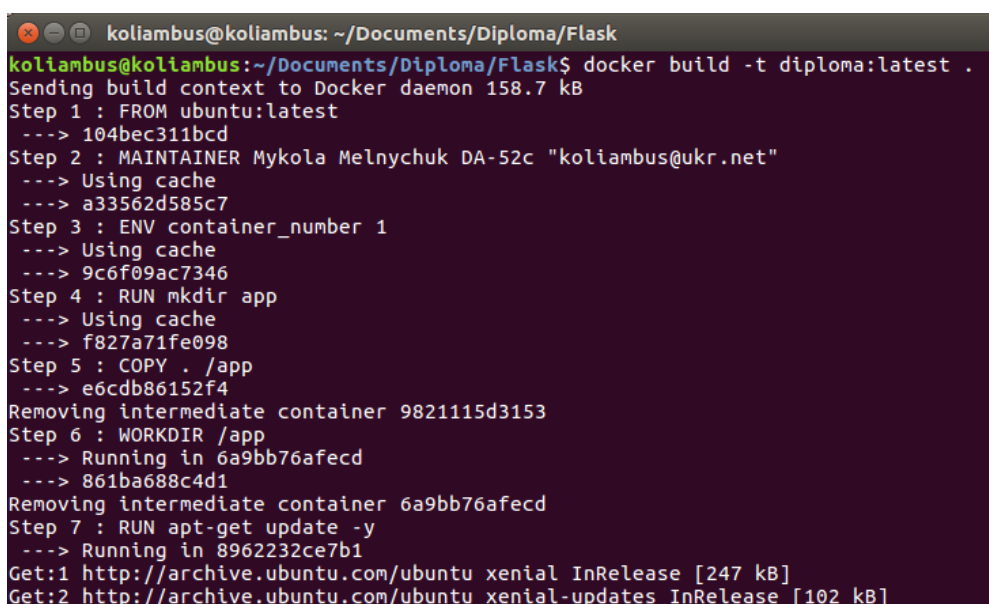

Рисунок 4.5 – Приклад виконання команди *docker build*

В логах даної команди (див. рис. 4.5) можна побачити шлях виконання команд з *dockerfile*, які позначені як “Step #”. Після деяких з них можна побачити повідомлення “Using cache”, що означає використання закешованого проміжного контейнеру. Під час роботи *docker build* кожен крок виконується в окремому контейнері, який після виконання інструкції видаляється з контексту

та може перейти у кеш. Після кожного запуску ці контейнери накопичуються та *docker build* команда проходить все швидше.

Після завершення останньої інструкції *dockerfile* команда *docker build,* при успішному виконанні, видає “Successful build UUID_образу” (рис. 4.6), де UUID_образу це номер образу, по якому можна запускати контейнер. Замість номеру образу також можна буде використати тег образу, що був встановлений під прапорцем *-t.*

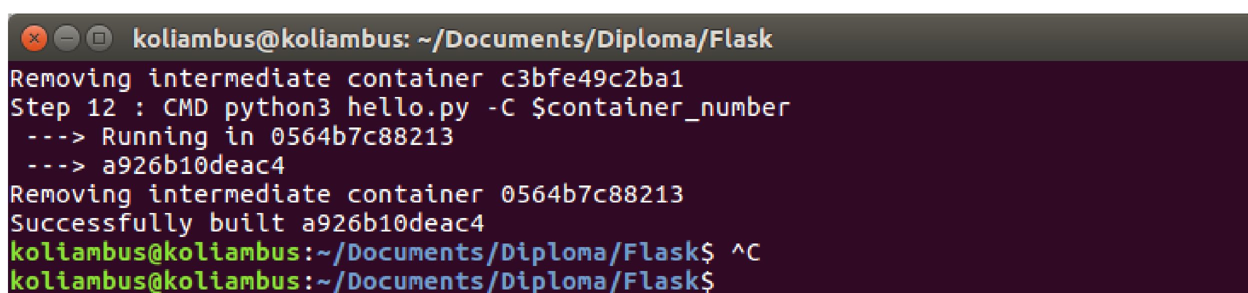

Рисунок 4.6 – Завершення виконання команди *docker build*

4.3.4 Запуск та управління контейнерами

4.3.4.1 Налаштування мережі

Так як в Docker буде виконано з’єднання одного контейнера з іншим, потрібно створити мережу всередині платформи (можна використати встановлену за замовчуванням, але в ній не можна вказати IP адресу контейнера). Для створення мережі використаймо команду *docker network*:

docker network create --subnet=172.18.0.0/16 flasknet

Команда *docker network create* створить окрему мережу в якій IP адреси будуть використовуватись з безкласової адресації підмережі 172.18.0.0/16 (рис. 4.7). В кінці команди встановлюється назва мережі *flasknet* для подальшого її використання при запуску контейнерів.

```
koliambus@koliambus: ~/Documents/Diploma/Flask
koliambus@koliambus:~/Documents/Diploma/Flask$ docker network create
 --subnet=172.18.0.0/16 flasknet
997a9023bf89e9bae4d2fafe1298985c142a1981928a9e6acb20ec98f38e150f
koliambus@koliambus:~/Documents/Diploma/Flask$
```

Рисунок 4.7 – Створення мережі в Docker

4.3.4.2 Запуск контейнера з перенаправленим портом

Для того щоб перевірити роботу створеного образу можна запустити досить легку команду *docker run,* без використання багатьох параметрів. Тоді для цього достатньо ввести в командний рядок:

docker run -t diploma:latest

Підчас виконання цієї команди образ під назвою та тегом *diploma:latest* розгорнеться в контейнер та запустить останню команду *CMD* з файлу *dockerfile* — *“python3 hello.py -C 1”*. Після “-C” стоїть одиниця, тому що змінна оточення *container_number,* що була вказана в *dockerfile* як 1 не була зміненна у команді *docker run*. В консолі можна буде побачити вивід запуску сервера (рис. 4.8).

```
koliambus@koliambus: ~/Documents/Diploma/Flask
koliambus@koliambus:~/Documents/Diploma/Flask$ docker run -t diploma:latest
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Рисунок 4.8 – Запуск простої команди docker run

Нажаль після запуску цієї команди підключитися до цього серверу з локального хоста неможливо. Намагаючись в браузері перейти по посиланню *http://127.0.0.1:5000/* буде невдалим (рис. 4.9).

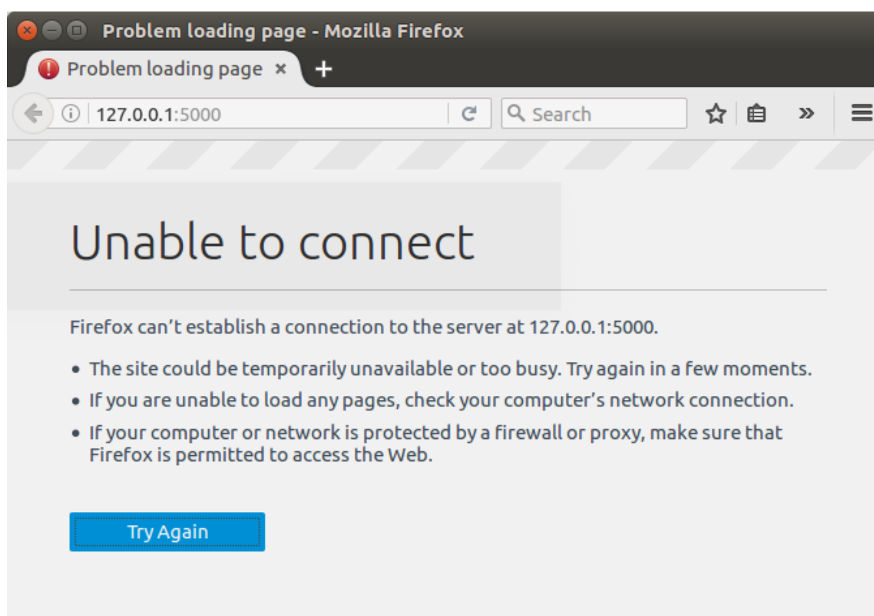

Рисунок 4.9 – Ілюстрація неможливості підключення до контейнера без мережевих налаштувань

Це відбувається через те, що порт 5000 не був опублікований на локальному хості. Для надання такої можливості потрібно додати параметр *-p* після якого вказати правило перенаправлення порту (рис. 4.10) як "порт_на_хості:порт_в_контейнері" :

docker run -t -p 5001:5000 diploma:latest

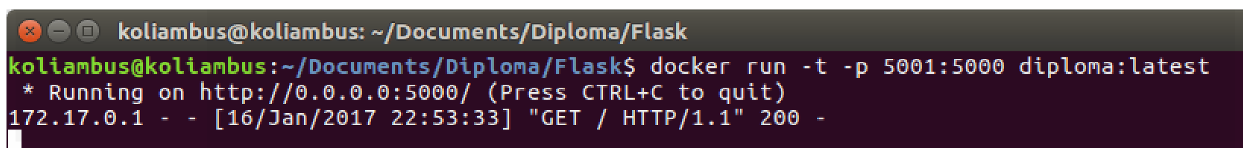

Рисунок 4.10 – Перенаправлення порту в команді docker run

В даному випадку внутрішній порт контейнера 5000 перенаправляється на зовнішній порт, тобто локального хоста 5001. Тепер до нього можна під ’єднатись по посиланню http://127.0.0.1:5001/ (рис. 4.11).

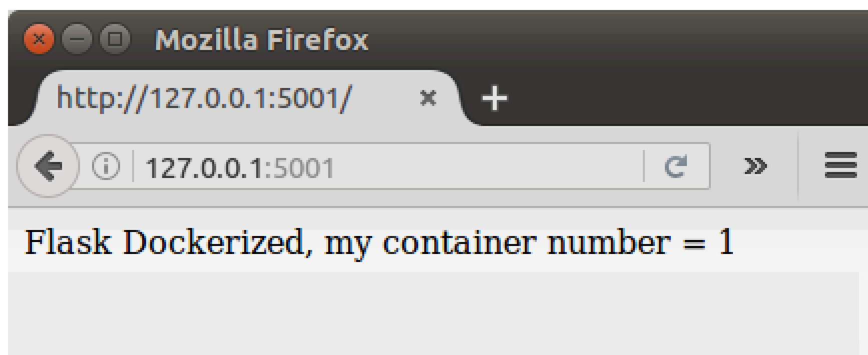

Рисунок 4.11 – Результат перенаправлення порту в команді docker run

4.3.4.3 З’єднання двох контейнерів всередині Docker мережі

В попередніх пунктах була створена та налаштовала підмережа *172.18.0.0/16* всередині Docker під назвою *flasknet*. Далі вона буде використана для з’єднання двох контейнерів без звернення до локального хоста. Для цього нам потрібно запустити два ідентичних контейнери нашого Flask додатку, включити їх в одну підмережу та роздати їм IP адреси. Для виконання цих завдань використаємо команду:

docker run -t -e "container_number=2" --net flasknet --ip 172.18.0.2 -p 5002:5000 -d diploma:latest

Розшифруємо дану команду:

- *-e "container_number=2"* — переписує команду dockerfile *ENV*, надаючи ключу “*container_number*” нового значення, для того, щоб контейнери можна було розрізнити.

- *--net flasknet* — задає контейнеру підмережу *flasknet*, що була створена раніше, для з'єднання контейнерів.
- *-ip* — задає контейнеру конкретну IP адресу, що повинна бути з підмережі *flasknet,* тобто *172.18.X.X.*
- *-d* — запуск контейнера у фоновому режимі.
- *-t, -p* — були згадані вище

Для кращого сприйняття тут краще, щоб *container_number (2)* збігався з останньою цифрою IP адреси (*172.18.0.2*) та останьою цифрою опублікованого порту (5002).

Далі варто запустити другий такий же самий контейнер, до якого і буде виконане з'єднання, використаємо в його основі *container_number=3.* В нас вийде:

docker run -t -e "container_number=3" --net flasknet --ip 172.18.0.3 -d diploma:latest

В даному випадку, прапорець *-p* не був використаний, так як ми не збираємось приєднуватись до нього з локального хоста. Але його все ж можна залишити. Командний рядок з цьому випадку буде виглядати як на рисунку 4.12.

```
koliambus@koliambus: ~/Documents/Diploma/Flask
koliambus@koliambus:~/Documents/Diploma/Flask$ docker run -t -e "container_numbe
r=2" --net flasknet --ip 172.18.0.2 -p 5002:5000 -d diploma:latest
6f3c83f522cb0b5228380f130a284d0a02a4c6c5acfbf22c637b31d3e419e75e
koliambus@koliambus:~/Documents/Diploma/Flask$
koliambus@koliambus:~/Documents/Diploma/Flask$
koliambus@koliambus:~/Documents/Diploma/Flask$ docker run -t -e "container_numbe
r=3" --net flasknet --ip 172.18.0.3 -d diploma:latest
772fe7186de14345a4fc975a4907932bc153a4cf4c791345584b2092d5a4490c
koliambus@koliambus:~/Documents/Diploma/Flask$
```

Рисунок 4.12 – Запуск контейнерів в одній підмережі з явними IP адресами

Тепер під 'єднаємося до Flask додатку контейнера №2, та викличемо у нього з допомогою REST api метод, */connect,* що з'єднає його з додатком в контейнері №3 (рис. 4.13).

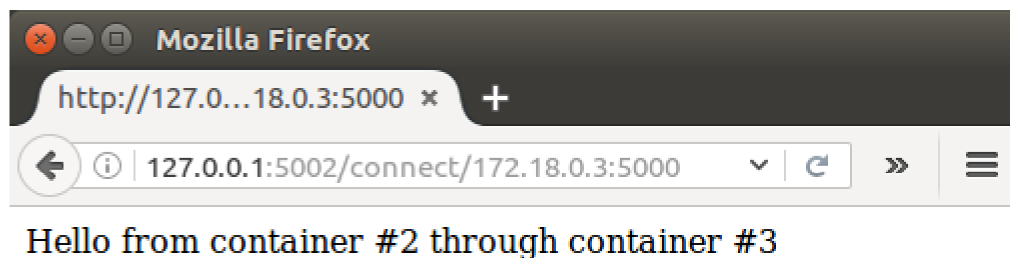

Рисунок 4.13 – Результат з'єднання двох контейнерів в Docker підмережі

Тут варто розшифрувати даний запит:

- *127.0.0.1:5002* — запустивши контейнер №2 ми опублікували його внутрішній порт 5000, як зовнішній порт 5002, тому ми і підключаємось до локального хоста і зовнішнього порту контейнера.
- */connect/* - згідно з REST api нашого Flask додатку це викличе команду з'єднання по HTTP протоколу, до IP адреси, вказаному після "/".
- *172.18.0.3:5000* — це адреса додатку в контейнері №3 по якій буде приєднуватись №2. Так як це з'єднання відбувається всередині Docker підмережі з назвою *flasknet,* то адресація буде вестися відносно внутрішніх адрес.

Загалом адресація проходить наступним чином:

1) Підключившись по адресу 127.0.0.1:5002 ми з'єднуємося з docker daemon, який у таблиці публічних портів знаходить підмережу та IP адресу

контейнера №2, який використовує цей порт та передає запит на його внутрішній порт 5000.

2) Знайдений контейнер зі своєї точки зору, тобто підмережі 172.18.0.0/16, під'єднується до внутрішньої IP адреси контейнеру №3 - 172.18.0.3 на порт 5000.

3) Далі контейнери по порядку виконують свої завдання та дають відповідь у зворотному порядку.

4.3.4.4 З'єднання контейнера з локальним хостом

Спочатку потрібно запустити контейнер з опублікованим портом, наприклад для контейнеру №2:

docker run -t -e "container_number=2" --net flasknet --ip 172.18.0.2 -p 5002:5000 -d diploma:latest

Для підключення контейнера до локального хоста, спочатку запустимо Flask додаток, де позначимо номер контейнера як №1, на локальній машині використавши командний рядок. Для цього треба виконати команду:

python3 hello.py -C 1

Підключитись до нього можна через браузер, ввівши там адресу 127.0.0.1:5000.

Тепер спробуємо з'єднатися додатком на локальному хості через контейнер №2. Для цього потрібно вказати контейнерному додатку, до якої адреси йому слід підключитись. Так як додаток знаходиться в контейнері, то він не бачить напряму локального хоста, але Docker встановлюючи підмережу 172.18.0.0/16 встановлює шлюх як локальний хост, тому адресою, до якої повинен підключитись додаток, це шлюз.

Для того, щоб дізнатися шлюз мережі контейнера можна скористатися командою *docker inspect UUID_контейнера*, що містить багато інформації про контейнер в JSON форматі. Так як *docker inspect* повертає багато інформації, варто скористатися командою *grep* для пошуку потрібної нам.

docker inspect 7c0e1b74cdc3 | grep Gateway

На виході цієї команди будемо мати щось схоже на рисунок 4.14:

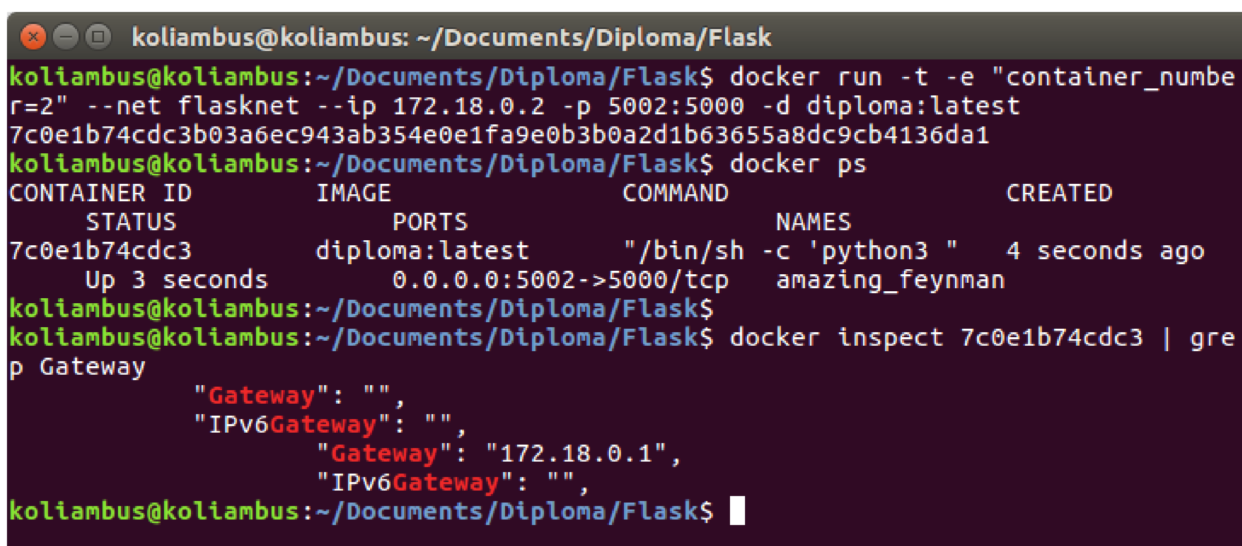

Рисунок 4.14 – Пошук шлюзу контейнера

Тут в полі “Gateway” вказана адреса шлюзу — 172.18.0.1 . Використаймо її для підключення з середини контейнера. Для цього в браузері під’єднаємось з адресою http://127.0.0.1:5002/connect/172.18.0.1:5000 (рис. 4.15).

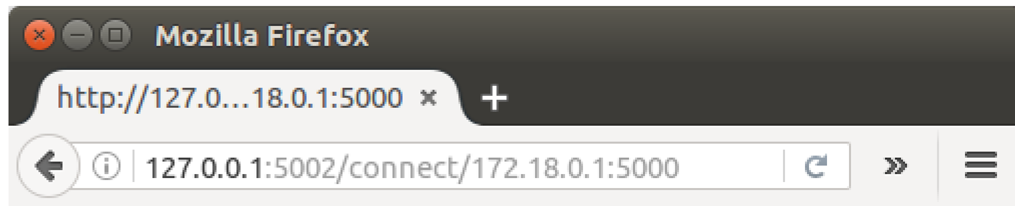

Рисунок 4.15 – Результат з’єднання контейнера з локальним хостом

4.4 Тестування Docker

Тестування Docker контейнера провести на операційних системах (ОС):

1) Ubuntu
2) Windows 10

Необхідними умовами для тестуванням є:

- Встановлення на ОС Docker

Перед тестуванням необхідно:

1) Запустити докер сервіс

4.4.1 План тестування

Тест кейс №1

1) Запуск в консолі команди:

   docker run -t -p 5001:5000 -d koliambus/diploma:latest

2) В браузері перейти по адресі:

   http://127.0.0.1:5001/

Очікуваний результат: Напис в браузері: “Flask Dockerized, my container number = 1”

Тест кейс №2

1) Запуск в консолі команди:

   docker network create --subnet=172.18.0.0/16 flasknet

2) Запуск в консолі команди:

docker run -t -e "container_number=2" --net flasknet --ip 172.18.0.2 -p 5002:5000 -d koliambus/diploma:latest

3) Запуск в консолі команди:

docker run -t -e "container_number=3" --net flasknet --ip 172.18.0.3 -p 5003:5000 -d koliambus/diploma:latest

4) В браузері перейти по адресі:

http://127.0.0.1:5002/connect/172.18.0.3:5000

Очікуваний результат: Напис в браузері: “Hello from container #2 through container #3”

4.4.2 Тестування в Ubuntu

Тест кейс №1

1) Добре
2) Добре

Очікуваний результат отримано. (рис. 4.16)

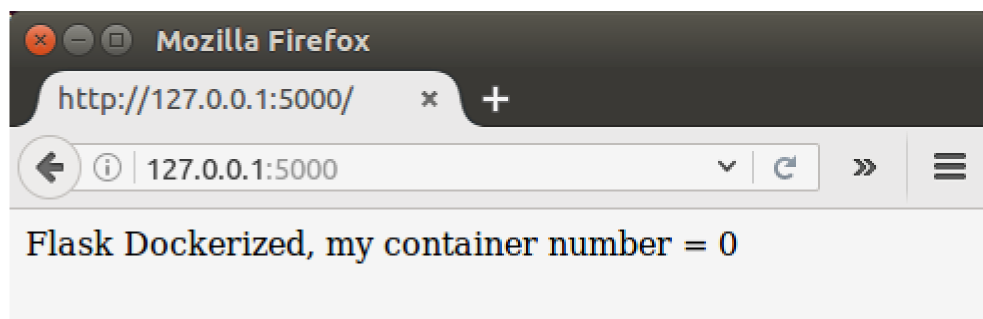

Рисунок 4.16 – Результат тест кейсу №1 на Ubuntu

Тест кейс №2

1) Добре
2) Добре
3) Добре

4) Добре

Очікуваний результат отримано (рис. 4.17).

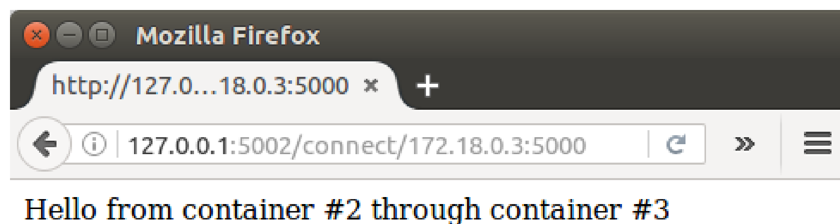

Рисунок 4.17 – Результат тест кейсу №2 на Ubuntu

4.4.3 Тестування у Windows 10

Тест кейс №1

1) Добре
2) Добре

Очікуваний результат отримано.

Тест кейс №2

1) Добре
2) Добре
3) Добре
4) Добре

Очікуваний результат отримано.

Всі тест кейси на ОС Ubuntu та Windows 10 пройдені успішно

4.5 Висновок

В даному розділі були продемонстровані кроки зі встановлення, налаштування та запуску контейнерів в системі Docker та роботи з мікрофреймворком Flask. Зокрема:

- Встановлення Flask на Ubuntu
- Модифікація базового Flask додатку
- Встановлення та перший запуск Docker на Ubuntu

- Налаштування Docker для запуску контейнерів Flask додатку.
- Створення образу
- Налаштування мережі
- Запуск контейнера з різними налаштуваннями мережі

Даний матеріал є досить детальною інструкцією з використання можливостей системи, зі всіма кроками, командами та їх тонкощами, який може бути досить корисним при вивчені Docker.

5 Управління термінами

5.1 Діаграма Ганта

Діаграма Ганта - це популярний тип діаграм, який використовується для ілюстрації плану, графіка робіт за будь-яким проектом. Є одним з методів планування та управління проектами. Перший формат діаграми був розроблений Генрі Л. Гантом у 1910 році [6].

Діаграма Ганта являє собою відрізки (графічні плашки), розміщені на горизонтальній шкалі часу. Кожен відрізок відповідає окремому завданню або підзадачі. Завдання і підзадачі, складові плану, розміщуються по вертикалі. Початок, кінець і довжина відрізка на шкалі часу відповідають початку, кінцю і тривалості завдання.

Діаграма може використовуватися для представлення поточного стану виконання робіт: частина прямокутника, що відповідає завданню, заштриховується, відзначаючи відсоток виконання завдання; показується вертикальна лінія, що відповідає моменту «сьогодні» (рис 5.1).

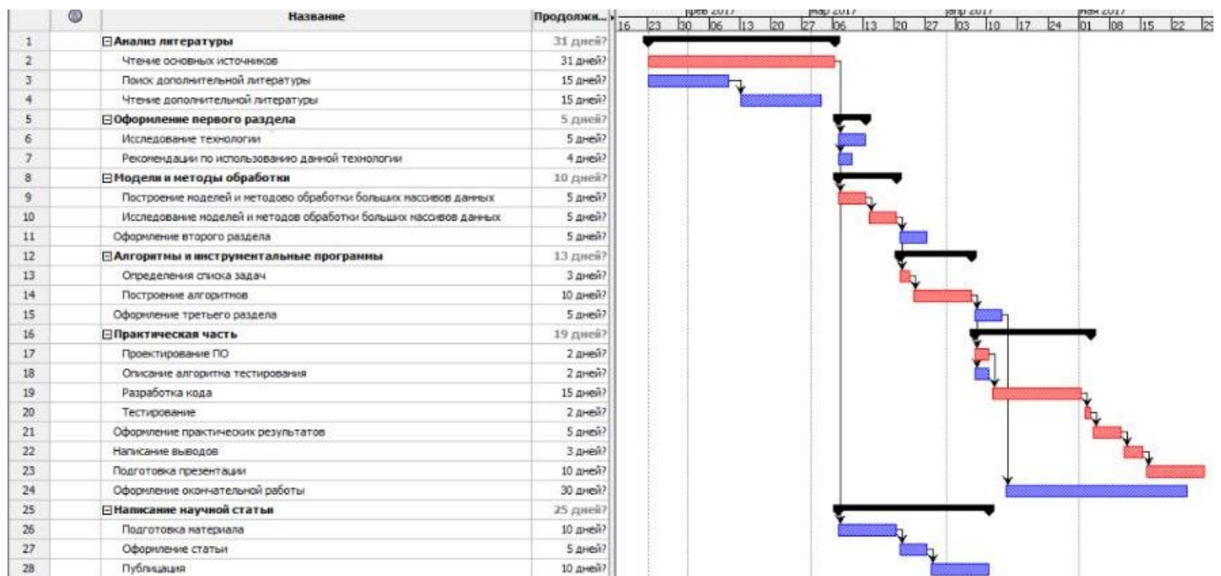

Рисунок 5.1 – Діаграма Гантта

5.2 Критичний шлях

Критичний шлях і час виконання (песимістичний - реалістичний — оптимістичний):

1) Читання літературних джерел - 41 - 31 - 26 днів
2) Аналіз предметної області - 30 - 25 - 20 днів
3) Вивчення існуючих робіт - 6 - 5 - 2 дні
4) Створення моделі алгоритму розв'язання задачі 23 - 20 - 15 днів
5) Проектування програмного рішення - 23 - 15 - 13 днів
6) Розробка коду - 18 - 11 - 9 днів
7) Тестування - 6 - 5 - 2 дні

Разом 112 днів (реалістична оцінка)

Відповідно до таблиці 5.1, з ймовірністю 0.4443 проект буде завершений за 112 днів.

Таблиця. 5.1 – Розрахунок ймовірності виконання проекту

№	Песимістич	Реально	Оптим	Дисперсія	Відхилення	Te
1	41	31	26	37.5	6.123724	31.83333
2	30	25	20	25	5	25
3	6	5	2	4.333333	2.081666	4.666667
4	23	20	15	16.33333	4.041452	19.66667
5	23	15	13	28	5.291503	16
6	18	11	9	22.33333	4.725816	11.83333
6	6	5	2	4.333333	2.081666	4.666667
Сума	147	112	87	137.832	22.54625	113.6667
		Z	-0.14196			
		F(Z)	0.4443			

5.3 Управління ризиками

Управління ризиками - це процес прийняття та виконання управлінських рішень, спрямованих на зниження ймовірності виникнення несприятливого результату і мінімізацію можливих втрат, викликаних його реалізацією. В рамках управління ризиками здійснюється кількісна та якісна оцінка ймовірності досягнення передбачуваного результату, невдачі і відхилення від мети.

Ризик являє собою можливу небезпеку несприятливого результату. Поняття ризику поєднує в собі оцінку ймовірності і наслідки настання несприятливої події [6].

У ситуації ризику відповідальна особа опиняється перед необхідністю розробки альтернативних варіантів рішення і подальшого вибору найбільш прийнятного з них. При цьому якщо дієслово "ризикувати" асоціюється з діями всупереч існуючим небезпекам, зневагою до них, то управління ризиком передбачає аналіз причин, джерел та факторів ризику, реалістичну оцінку небезпеки на шляху до наміченої мети, оцінку ефективності різних методів ризик-менеджменту і в той же час відхід від непотрібного ризику і невиправданих втрат. Вибір варіанту, максимально знижує ризик, часто веде до невисоких результатів.

Результати проведення класифікації ризиків продемонстровані в таблиці 5.2.

Таблиця 5.2 – Класифікація ризиків

Ризик	Заходи	Імовірн.	Вплив
Несподівана зміна умов використання використовуваного в роботі ПО	Розгляд альтернативних програмних рішень для виконання роботи заздалегідь	низька	сильний

Таблиця 5.2 (продовження)

Несподівана втрата продуктивності і ефективності в силу розчарування тим що нічого не працює.	Розгляд можливих поширених проблем Oracle SOA Suite, детальне вивчення документації.	висока	сильний
Вихід з ладу основного домену розгортки веб-сервісів - localhost	Збереження проміжних результатів, резервне розгортання на менш надійних ресурсах (Google App Engine, etc)	середня	сильний
Невиконання робіт в термін через несподівані додаткових завдань.	Більш якісне планування робіт по виконанню диплома.	середня	сильний
Труднощі в розумінні архітектури майбутньої системи і того як все це повинно бути влаштовано	Витратити додатковий час на вивчення документації і принципів роботи.	середня	середній
Неповний обсяг необхідної інформації в відібраних джерелах	Витратити додатковий час на вивчення всіх додаткових джерел	низька	середній
Втрата актуальності роботи	Постановка завдання та опис актуальності в універсальному вигляді.	низька	Вище серед-нього

5.4 Висновок

Методи управління термінами дозволяють краще зрозуміти часові рамки виконання проекту, його важливі задачі, які можуть зупинити процес при неправильному розподілені часу та розрахувати ризики та заздалегідь запобігти їх появі.

ВИСНОВКИ

В першому розділі було виконане порівняння віртуалізації та контейнеризації. В результаті було виявлено, що контейнери займають менше місця на системному диску, але не досконалі в плані безпеки. В свій час віртуальні машини краще підходять для ізольованих систем.

Якщо необхідно запускати максимум додатків на мінімальній кількості серверів, в такому випадку краще скористатися контейнерами. Але пам'ятайте про необхідність додатково подбати про безпеку. Якщо ж потрібно виконувати безліч додатків і / або підтримувати різні ОС - краще підійдуть віртуальні машини. І якщо питання безпеки для стоїть на першому місці, то теж краще залишитися при VM.

Вважаю, більшість з нас будуть використовувати контейнери спільно з віртуальними машинами як в хмарах, так і у власних серверах. Адже можливості економії, що відкриваються нам контейнерами, на масштабі дуже великі, щоб їх ігнорувати. А у віртуальних машин, як і раніше є чимало переваг.

У другому розділі було виконане порівняннь сучасних контейнерних систем, та виявленно, що найпопулярніша контейнерна технологія — Docker. За останні роки її використання значно збільшилось і стало “мейнстрімом” у світі розробки та розгортання додатків. LXD не являється прямим конкурентом через різний підхід та мету цих технологій. А от платформа rkt може конкурувати з Docker, але вона відносно нова та не популярна. Та з часом вона можливо стане досить стабільною та інноваційною, щоб стати на озброєння у розробників та замінити Docker.

При дедальному розгляді Docker у третьому розділі було виявлено, що користувалький інтерфейс містить досить багато команд, а основний файл налаштування dockerfile має багато інструкцій, та їх налаштування вимагає багато зусиль, щодо тонкостей роботи кожної інструкції чи прапорця для

команд терміналу. Функціонал досить обширний та дозволяє зручно налаштовувати, відлагоджувати та використовувати всю систему.

В четвертому розділі були продемонстровані кроки зі встановлення, налаштування та запуску контейнерів в системі Docker та роботи з мікрофреймворком Flask. Зокрема:

- Встановлення Flask на Ubuntu
- Модифікація базового Flask додатку
- Встановлення та перший запуск Docker на Ubuntu
- Налаштування Docker для запуску контейнерів Flask додатку.
  - Файл .dockerignore
  - Файл dockerfile
- Створення образу
- Налаштування мережі
- Запуск контейнера з різними налаштуваннями мережі

Матеріал четвертого розділу є досить детальною інструкцією з використання можливостей системи, зі всіма кроками, командами та їх тонкощами, який може бути досить корисним при вивчені Docker.

В останньому – п’ятому розділі, були розглянуті методи управління термінами та створені діаграма Ганта, розрахований критичний шлях та побудована таблиця класифікації ризиків в даній роботі.

Виявлено, що методи управління термінами дозволяють краще зрозуміти часові рамки виконання проекту, його важливі задачі, які можуть зупинити процес при неправильному розподілені часу та розрахувати ризики та заздалегідь запобігти їх появі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Офіційний сайт Docker. – Режим доступа: https://www.docker.com/. Дата доступа: 12.01.2017.
2. Хабр-Хабр. – Режим доступу: http://habrahabr.ru. - Дата доступа: 10.01.2017.
3. Client-Server Programming and Applications. — Department of Computer Sciences, Purdue University, West Lafayette, IN 47907: Prentice Hall, 1993.
4. Офіційний сайт документації Docker. – Режим доступа: https://docs.docker.com/ - Дата доступа: 08.01.2017.
5. Офіційний Git репозиторій Docker. – Режим доступа: https://github.com/docker/docker/ - Дата доступа: 13.01.2017.
6. Вікіпедія. – Режим доступа: https://uk.wikipedia.org - Дата доступа: 15.01.2017.
7. Записки программиста. Режим доступа: http://eax.me/docker/ - Дата доступа: 12.01.2017.
8. DOU. Режим доступа: https://dou.ua - Дата доступа: 07.01.2017.
9. Хакер ру. Режим доступа: https://xakep.ru - Дата доступа: 05.01.2017.
10. Docker: Creating Structured Containers. - Pethuru Raj et al, 2016.
11. The Docker Book. - James Turnbull, 2016.
12. Docker Cookbook. Solutions and Examples for Building Distributed Applications. - O'Reilly Media, 2015.
13. XGU ru. Режим доступа: http://xgu.ru/ - Дата доступа: 13.01.2017.
14. Ubuntu LXD. Режим доступа: https://www.ubuntu.com/cloud/lxd - Дата доступа: 3.01.2017.
15. Авторские статьи об OpenSource. Режим доступа: http://vasilisc.com/ - Дата доступа: 9.01.2017.
16. Flask. Режим доступа: http://flask.pocoo.org/ - Дата доступа: 5.01.2017.