

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”**

Створення та запуск мікросервісу в Docker контейнері

**МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи з курсу
“Операційні системи”
для студентів базового напрямку 6.050101
“Комп’ютерні науки”**

**Затверджено
на засіданні кафедри
“Системи автоматизації проектування”**

Львів – 2020

Мета: Ознайомитись із особливостями створення docker іміджі та запуску мікросервісів в контейнері.

Короткі теоретичні відомості:

1. Файл `.dockerignore`

CLI інтерпретує файл `.dockerignore` як список шаблонів для ігнорування файлів та папок, який є досить подібним до файлу `.gitignore`. 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 проєкті використовується `git`, можна помістити:
`/.gitignore`
`/readme.md`
- Також файли які вивористовує IDE, наприклад
`/.idea`

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.

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 під час виконання файлу буде використаний останній. Білдер повертає помилку, якщо вона заданий образ чи тег не є дійсний.

Примітка! Використавши тег <latest> поверне останню версію

2.2. MAINTAINER

MAINTAINER <ім'я>

Інструкція MAINTAINER дозволяє залишити автора створеного файлу та образу.

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* -у 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"]*

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*

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"  
  
}
```

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.

2.7. ENV

ENV <ключ> <значення>

ENV <ключ> = <значення> ...

Команда ENV встановлює змінну оточення на значення . Це значення буде перебувати в середовищі всіх "нащадків" Dockerfile команд і можуть замінювати ключ на значення за допомогою \$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.

2.8. ADD

ADD має дві форми:

ADD <src> ... <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/*

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.

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.

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`.

2.12. WORKDIR

WORKDIR /шлях_до_робочої_папки

Інструкція WORKDIR встановлює робочий каталог для будь-яких RUN, CMD, Entrypoint, COPY і ADD інструкцій, які слідує 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. Робота з docker контейнерами

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`
образу

Встановлює аргументи при збиранні

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

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.

3.1.1. Збірка з Git репозиторію

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
```

3.1.2. Збірка з текстового файлу

Замість того, щоб вказати контекст, ви можете передати одну Dockerfile в URL або передати файлу 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.

3.2. 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.

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:

3.2.1. Фоновий режим (-d)

Для запуску контейнера в фоновому режимі, можна використати `-d=true` або просто `-d`. За своєю конструкцією, контейнери, запуснені у фоновому режимі зупиняються тільки тоді, коли процес зупиняється всередині контейнера. Контейнер в фоновому режимі 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*.

3.2.2. Звичайний режим

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

- `-a = []`: Приєднати до ``STDIN`, STDOUT` і / або `STDERR`
- `-t`: симулювати термінал (TTY)
- `--sig-proxy` Проху 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`.

3.2.3. 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=""`.

3.2.4. Налаштування мережі

<code>--dns=[]</code>	Використати особливий DNS сервер
<code>--network:</code>	Параметр мережі контейнера
<code>--network-alias=[] :</code>	Дати скорочене ім'я контейнера в мережі
<code>--add-host=""</code> :	Додати рядок в <code>/etc/hosts</code> (host:IP)
<code>--mac-address=""</code> :	Встановити MAC адресу для контейнера
<code>--ip=""</code> :	Встановити IPv4 адресу для контейнера
<code>--ip6=""</code> :	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.

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

3.2.5. 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.

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 замість *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_групи 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. Допоміжні команди

- 1) Список всіх існуючих контейнерів (не тільки працюючих):

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
```

Хід роботи

1. Виконуємо наступні команди в терміналі Ubuntu:

```
sudo apt-get update
```

```
sudo apt-get install python3-pip
```

```
sudo pip3 install --upgrade pip
```

```
sudo pip3 install Flask
```

Ми встановили Python 3, pip і мікрофреймворк Flask. Для того, щоб перевірити роботу Flask, на офіційному сайті фреймворку є скрипт, який дозволяє запустити сервер, що повертає на сторінку «Hello World!». Для цього треба перейти в будь-яку робочу директорію 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 $First_Name $Second_Name"
```

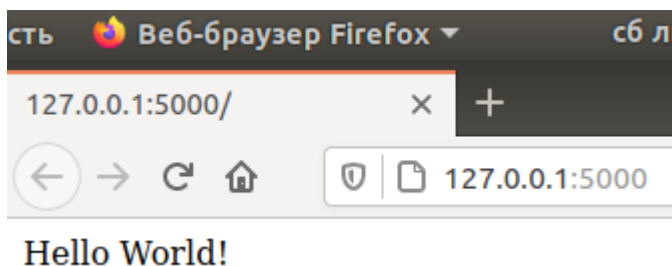
```
if __name__ == "__main__":  
    app.run()
```

Запусти даний скрипт можна за допомогою команди:

python3 hello.py

```
darkor@darkor-VirtualBox: ~/test
Файл Зміни Перегляд Пошук Термінал Довідка
darkor@darkor-VirtualBox:~/test$ python3 hello.py
* Serving Flask app "hello" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Ввівши адресу <http://127.0.0.1:5000/> у браузері, можна побачити вивід.



2. Налаштування Docker для запуску контейнерів 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 парадигми мови Python.

Переходимо в папку із Flask скриптом. Створюємо файл **.dockerignore**, і записуємо в нього наступне:

```
*.sh
/.gitignore
/readme.md
/.idea
```

Створюємо **dockerfile**, записуємо:

```
FROM ubuntu:latest
MAINTAINER First_Name Sec_Name kn-N "personal@mail"
ENV container_number 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
```

requirements.txt

```
requests
Flask==0.12
```

hello.py

```
# -*- coding: utf-8 -*-
import optparse
import requests
from flask import Flask

class Parser:
    @staticmethod
    def parse_container():
        parser = optparse.OptionParser()
        parser.add_option("-C", "--container", dest="container", default=0)
        options, _ = parser.parse_args()
        return options.container

class Runner:
    @staticmethod
    def run(name, application):
        if name == "__main__":
            Config.container_number = Parser.parse_container()
            application.run(host="0.0.0.0", port=Config.default_port)

class Config:
    container_number = 0
    default_port = 5000
    app = None

class RoutesConfigurator:
    @staticmethod
    def configure(application):
        @application.route("/")
        def hello():
            return "Flask Dockerized, my container number = " +
str(Config.container_number)

        @application.route("/hello/<int:container>")
        def hello_container(container):
            return "Hello from container #" + str(container) + " through
container #" + str(Config.container_number)

        @application.route("/connect/<string:address>")
        def connect(address):
            connector = HttpConnector()
            return connector.connect("http://" + address + "/hello/" +
str(Config.container_number)).content
```

```
class HttpConnector:
    def connect(self, address):
        return requests.get(address)

Config.app = Flask(__name__)

RoutesConfigurator.configure(Config.app)

Runner.run(__name__, Config.app)
```

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*.

```
darkor@darkor-VirtualBox: ~/test
Файл Зміни Перегляд Пошук Термінал Довідка
Collecting itsdangerous>=0.21
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting Jinja2>=2.4
  Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp38-cp38-manylinux1_x86_64.whl (32 kB)
Collecting urllib3<1.27,>=1.21.1
  Downloading urllib3-1.26.2-py2.py3-none-any.whl (136 kB)
Collecting Werkzeug>=0.7
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Installing collected packages: MarkupSafe, Werkzeug, urllib3, Jinja2, itsdangerous, idna, click, chardet, certifi, requests, Flask
Successfully installed Flask-0.12 Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 certifi-2020.11.8 chardet-3.0.4 click-7.1.2 idna-2.10 itsdangerous-1.1.0 requests-2.25.0 urllib3-1.26.2
Removing intermediate container dff4c09330a8
---> 28b2cab3b6bf
Step 11/12 : EXPOSE 5000
---> Running in 79531cc1d8d7
Removing intermediate container 79531cc1d8d7
---> 0eeef42f026f
Step 12/12 : CMD python3 hello.py -C $container_number
---> Running in b9852a1f481f
Removing intermediate container b9852a1f481f
---> 6db837cc87c9
Successfully built 6db837cc87c9
Successfully tagged diploma:latest
darkor@darkor-VirtualBox:~/test$
```

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* (*diploma:latest*).

4. Запуск та управління контейнерами

4.1. Налаштування мережі

Для створення мережі використаємо 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.

```
darkor@darkor-VirtualBox: ~/test
Файл Зміни Перегляд Пошук Термінал Довідка
darkor@darkor-VirtualBox:~/test$ sudo docker network create --subnet=172.18.0.0/16 flasknet
[sudo] пароль до darkor:
f8cefdeee5c52c54923e57efa54030167e9579ceceb650504d9818f5669a57c7
darkor@darkor-VirtualBox:~/test$
```

4.2. Запуск контейнера з перенаправленням портом

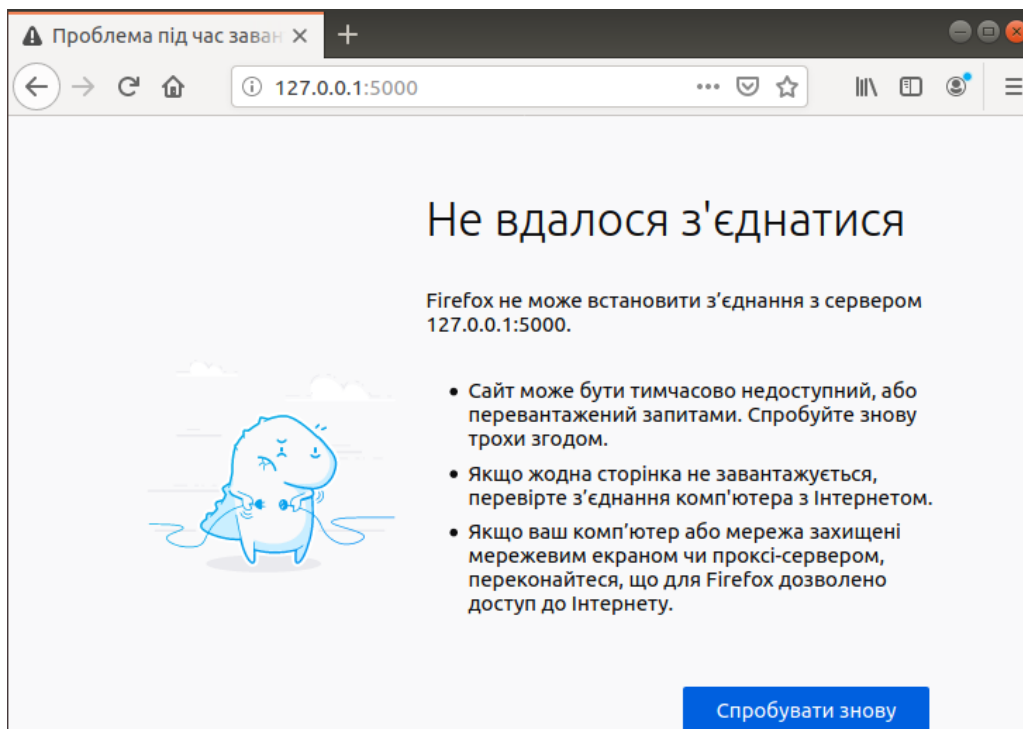
Для того щоб перевірити роботу створеного образу можна запустити досить легку команду *docker run*, без використання багатьох параметрів. Тоді для цього достатньо ввести в командний рядок:

```
docker run -t diploma:latest
```

Якщо після запуску ви не можете зупинити виконання за допомогою *Ctrl + C* то перезавантажте ОС і запустіть з параметром *-i* (*docker run -t -i diploma:latest*). Під час 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 змінена у команді *docker run*. В консолі можна буде побачити вивід запуску сервера.

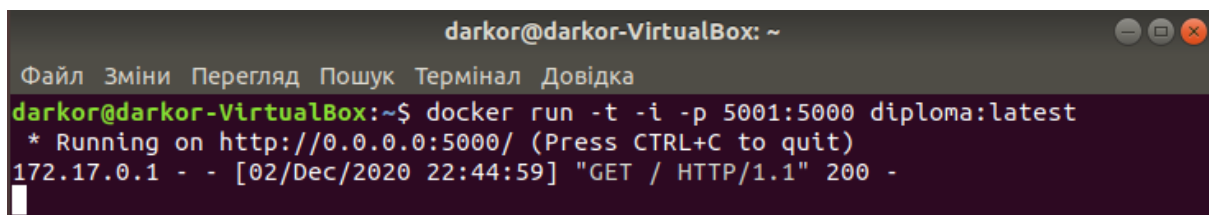
```
darkor@darkor-VirtualBox: ~/test
Файл Зміни Перегляд Пошук Термінал Довідка
darkor@darkor-VirtualBox:~$ cd test
darkor@darkor-VirtualBox:~/test$ docker run -t diploma:latest
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

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.

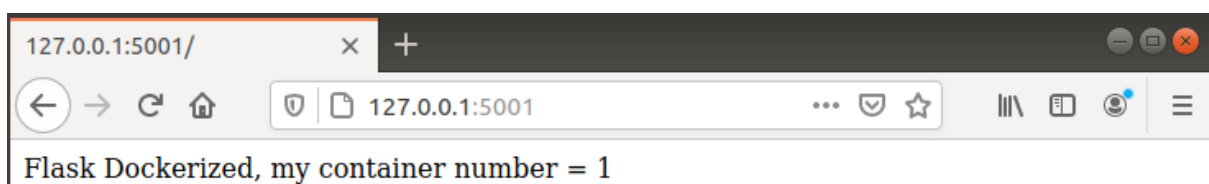


Це відбувається через те, що порт 5000 не був опублікований на локальному хості. Для надання такої можливості потрібно додати параметр `-p` після якого вказати правило перенаправлення порту як “порт_на_хості:порт_в_контейнері” :

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



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/>.



4.3. З'єднання двох контейнерів всередині Docker мережі

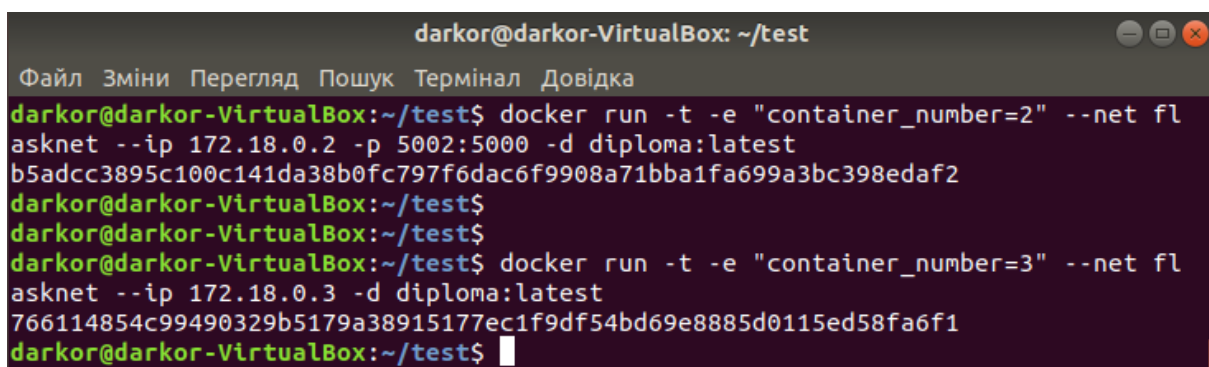
В попередніх пунктах була створена та налаштувала підмережа `172.18.0.0/16` всередині Docker під назвою `flasknet`. Далі вона буде

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
```

```
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 цього випадку буде виглядати як на рисунку.



```
darkor@darkor-VirtualBox: ~/test
Файл Зміни Перегляд Пошук Термінал Довідка
darkor@darkor-VirtualBox:~/test$ docker run -t -e "container_number=2" --net flasknet --ip 172.18.0.2 -p 5002:5000 -d diploma:latest
b5adcc3895c100c141da38b0fc797f6dac6f9908a71bba1fa699a3bc398edaf2
darkor@darkor-VirtualBox:~/test$
darkor@darkor-VirtualBox:~/test$
darkor@darkor-VirtualBox:~/test$ docker run -t -e "container_number=3" --net flasknet --ip 172.18.0.3 -d diploma:latest
766114854c99490329b5179a38915177ec1f9df54bd69e8885d0115ed58fa6f1
darkor@darkor-VirtualBox:~/test$
```

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.



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.

4.4. 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 (в іншому терміналі):

```
python3 hello.py -C 1
```



```

darkor@darkor-VirtualBox:~/test$ python3 hello.py -C 1
* Serving Flask app "hello" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
172.18.0.2 - - [03/Dec/2020 01:32:15] "GET /hello/2 HTTP/1.1" 200 -

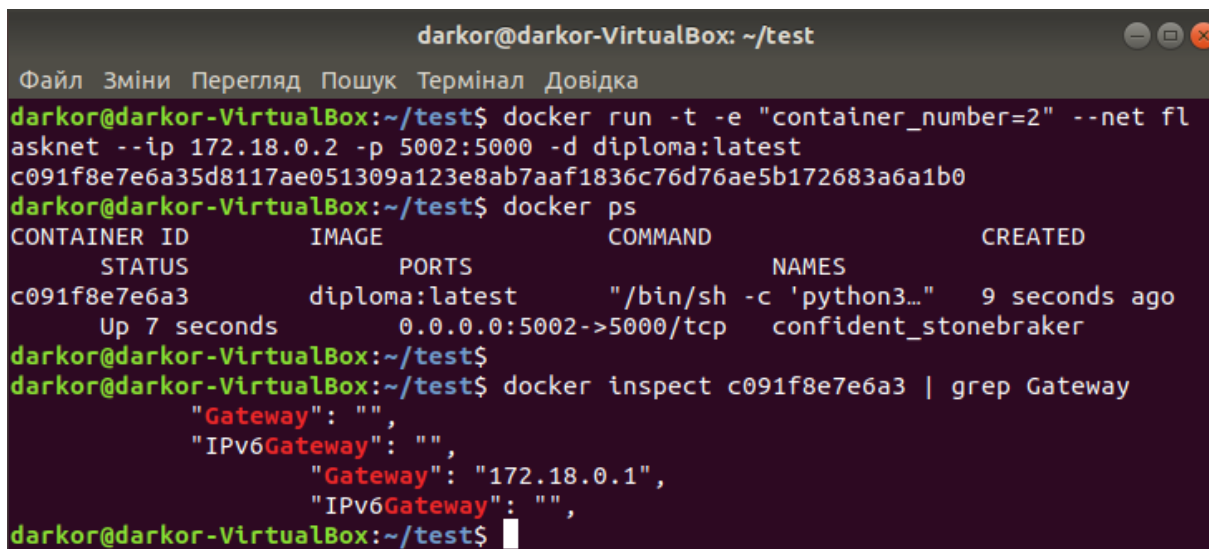
```

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 напряду локального хоста, але Docker встановлюючи підмережу 172.18.0.0/16 встановлює шлях як локальний хост, тому адресою, до якої повинен підключитись додаток, це шлях.

Для того, щоб дізнатися шлях 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`



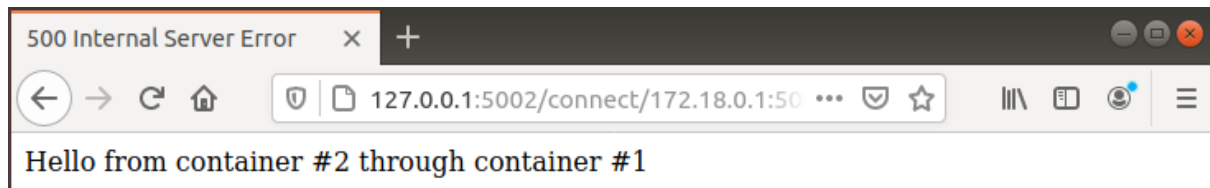
```

darkor@darkor-VirtualBox: ~/test
Файл Зміни Перегляд Пошук Термінал Довідка
darkor@darkor-VirtualBox:~/test$ docker run -t -e "container_number=2" --net fl
asknet --ip 172.18.0.2 -p 5002:5000 -d diploma:latest
c091f8e7e6a35d8117ae051309a123e8ab7aaf1836c76d76ae5b172683a6a1b0
darkor@darkor-VirtualBox:~/test$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS            PORTS              NAMES
c091f8e7e6a3       diploma:latest     "/bin/sh -c 'python3..." 9 seconds ago
Up 7 seconds      0.0.0.0:5002->5000/tcp    confident_stonebraker
darkor@darkor-VirtualBox:~/test$
darkor@darkor-VirtualBox:~/test$ docker inspect c091f8e7e6a3 | grep Gateway
    "Gateway": "",
    "IPv6Gateway": "",
    "Gateway": "172.18.0.1",
    "IPv6Gateway": "",
darkor@darkor-VirtualBox:~/test$

```

Тут в полі “Gateway” вказана адреса шлязу — 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> .



Ми з'єднали контейнер з локальним хостом.