**Модуль 2: Python Development**

**Заняття 2: Docker**

**Docker**

Наступний рівень створення віртуальних середовищ — це зімітувати всю операційну систему, скористатися "віртуальною машиною" для запуску вашого застосунку. Так ви зможете бути впевненими на 100%, що жодні сторонні залежності системного рівня не втрутяться в роботу вашого застосунку. Звичайно, ви можете використовувати повноцінні віртуальні машини на зразок VirtualBox або аналоги для цього. Але є зручніше, а головне економічне рішення — це Docker.

Docker – це платформа, яка дозволяє «створювати, постачати та запускати будь-який застосунок всюди». За неймовірно короткий час вона пройшла великий шлях і тепер вважається стандартним способом вирішення одного з найдорожчих аспектів програмного забезпечення – розгортання.

Docker дозволяє запустити віртуальну машину за мінімальних накладних витрат. Практично ви нічого не втрачаєте, запускаючи застосунок в Docker, навантаження на процесор або пам'ять залишається на тому ж рівні, що й при використанні рішень без віртуалізації. Але сам застосунок працює на "окремій" контрольованій ізольованій операційній системі зі своїми застосунками, налаштуваннями та іншим.

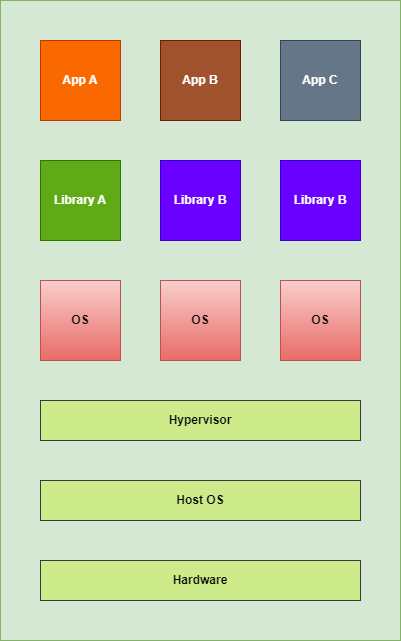
**Контейнери**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/intro#%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%B9%D0%BD%D0%B5%D1%80%D0%B8)Контейнери докорінно змінюють спосіб розробки, розповсюдження та функціонування програмного забезпечення. Розробники можуть створювати програмне забезпечення на локальній системі, точно знаючи, що воно працюватиме однаково у будь-якому операційному середовищі. Контейнери (containers) – це засоби інкапсуляції застосунку разом з його залежностями.

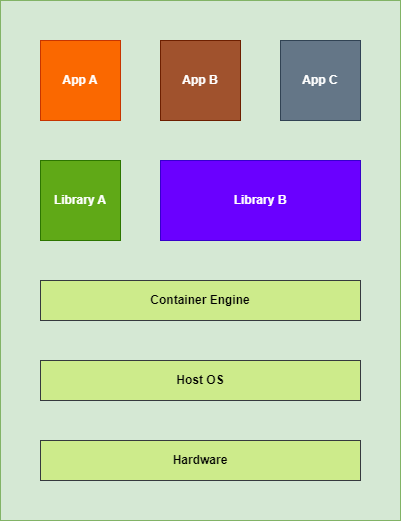
В основу Docker було закладено існуючу технологію Linux-контейнерів з різноманітними обгортками та розширеннями – здебільшого, використовуючи образи, що переносяться, та зручний для користувача інтерфейс – для створення повністю готового до застосування рішення, що забезпечує створення та розповсюдження контейнерів.

Платформа Docker складається із двох окремих компонентів: Docker Engine — механізму, що відповідає за створення та функціонування контейнерів, та Docker Hub — хмарного сервісу для розповсюдження контейнерів (такий собі аналог GitHub).

На перший погляд контейнери можуть здатися лише спрощеною формою віртуальних машин, але, незважаючи на те, що контейнери та віртуальні машини на перший погляд здаються схожими, між ними існують важливі відмінності, які простіше всього продемонструвати на графічних схемах.



Три віртуальні машини, що працюють на одному хості



Три контейнери, що працюють на одному хості

На відміну від віртуальних машин, ядро хоста (Host OS) спільно використовується (розділяється) працюючими контейнерами. Це означає, що контейнери завжди обмежуються використанням того самого ядра, яке функціонує на хості.

**Встановлення**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/intro#%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F)Докер постачається у вигляді Community Edition (CE) та Enterprise Edition (EE), ми будемо використовувати CE. На головній сторінці доступні посилання для завантаження [Docker Desktop](https://www.docker.com/products/docker-desktop) під усі популярні платформи. Виберіть вашу та встановіть Докер.

Щоб переконатися в правильності встановлення та працездатності програмного середовища Docker, варто виконати команду docker version. Ця команда повинна видати інформацію, схожу на наведену нижче:

Client: Docker Engine - Community

Cloud integration: 1.0.12

Version: 20.10.5

API version: 1.41

Go version: go1.13.15

Git commit: 55c4c88

Built: Tue Mar 2 20:14:53 2021

OS/Arch: windows/amd64

Context: default

Experimental: true

Server: Docker Engine - Community

Engine:

Version: 20.10.5

API version: 1.41 (minimum version 1.12)

Go version: go1.13.15

Git commit: 363e9a8

Built: Tue Mar 2 20:15:47 2021

OS/Arch: linux/amd64

Experimental: false

containerd:

Version: 1.4.4

GitCommit: 05f951a3781f4f2c1911b05e61c160e9c30eaa8e

runc:

Version: 1.0.0-rc93

GitCommit: 12644e614e25b05da6fd08a38ffa0cfe1903fdec

docker-init:

Version: 0.19.0

GitCommit: de40ad0

**Навіщо потрібен Docker**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/intro#%D0%BD%D0%B0%D0%B2%D1%96%D1%89%D0%BE-%D0%BF%D0%BE%D1%82%D1%80%D1%96%D0%B1%D0%B5%D0%BD-docker)Головна причина — Docker заощаджує бізнесу багато грошей. Але перерахуємо все ж таки докладніше основні причини використання:

* Заміна віртуальних машин
* Прототипування програмного забезпечення
* Упаковка програмного забезпечення
* Можливість для архітектури мікросервісів
* Моделювання мереж
* Можливість продуктивності повного стеку в автономному режимі
* Скорочення неминучих витрат на відлагодження
* Документування залежностей програмного забезпечення
* Можливість безперервної доставки

**Архітектура та компоненти Docker**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/intro#%D0%B0%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0-%D1%82%D0%B0-%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%B8-docker)У Docker використовується архітектура клієнт/сервер, відповідно до якої клієнт взаємодіє з демоном Docker, а той надає всі необхідні клієнту послуги.

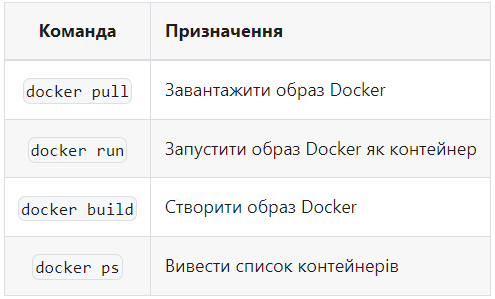
Розглянемо компоненти, що складають екосистему Docker:

* *сервер* або *демон Docker*. Виконується в хост-системі та керує всіма запущеними контейнерами;
* *контейнер Docker*. Автономна віртуальна система, що містить процес, що виконується, всі файли, залежності, адресний простір процесу та мережеві порти, необхідні застосунку. Оскільки кожен контейнер має свій простір портів, варто організувати їх відображення у фактичні порти на рівні Docker;
* *клієнт Docker*. Інтерфейс користувача або командний рядок для взаємодії з демоном Docker;
* *образи Docker*. Шаблонні файли, доступні лише для читання, із контейнером Docker. Їх можна переміщати та передавати. На відміну від віртуальних машин, ці файли можна зберігати в системі управління версіями;
* *реєстр Docker*. Репозиторій для зберігання та розповсюдження образів контейнерів Docker. Приклад [Docker Hub](https://hub.docker.com/) (аналог GitHub), куди можна поміщати та звідки можна витягувати образи. Організації можуть організувати свій реєстр;
* файл Dockerfile. Це дуже простий текстовий файл, що містить команди, які виконують збирання образів Docker;
* *Docker Machine*. Встановлює та конфігурує Docker-хости на локальних та віддалених ресурсах. Крім того, Machine конфігурує клієнта Docker, спрощуючи процедуру перемикання між середовищами. За найсвіжішою інформацією з цієї теми звертайтесь до офіційної [онлайн-документації Docker](https://docs.docker.com/);
* *Docker Swarm*. По суті, це готовий до використання механізм кластеризації, що дозволяє об'єднати кілька вузлів Docker в один великий хост Docker.
* *Docker Compose*. Інструмент для створення та виконання застосунків, скомпонованих з кількох Docker-контейнерів. Такі компонування використовуються головним чином при розробці та тестуванні, але набагато рідше у виробничому середовищі.

**Ключові команди Docker**

Основна функція Docker – створювати, відправляти та запускати програмне забезпечення у будь-якому місці, де є Docker. Для користувача, найчастіше Docker – це програма з командним рядком, який вони запускають. Як і будь-який інший інструмент управління вихідним кодом на зразок git, ця програма має підкоманди, які виконують різні операції. Перелічимо їх.

Основні підкоманди Docker.



**docker pull**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/command#docker-pull)Команда docker pull завантажує вказаний образ із реєстру Docker на локальний комп'ютер:

docker pull image:tag

Наприклад, щоб завантажити образ MySQL, потрібно виконати наступну команду.

docker pull MySQL

Примітка: Якщо тег (tag, що позначає версію) не вказано, команда підставить тег "latest" і завантажить лише останню версію образу MySQL. Фактично виконалася команда docker pull MySQL:latest.

**docker run**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/command#docker-run)Після завантаження образу (командою pull) необхідно виконати його запуск командою docker run:

docker run [options] image: tag [command, args]

Ця команда розгортає контейнер у його власній файловій системі, що має свій набір портів та IP-адресу. Крім назви образу, команді run можна також передати додаткові ключі та аргументи. Ті, що найчастіше використовуються з них:

* i перемикає команду в інтерактивний режим та відкриває STDIN;
* t створює псевдотермінал tty
* d запускає процес у фоновому режимі (режим демона), коли контейнер запускається без підтримки командного рядка

**docker ps**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/command#docker-ps)Команда docker ps виводить список усіх контейнерів, запущених до цього моменту:

docker ps [options]

Якщо виконати команду ps із ключем –a, ми побачимо список усіх контейнерів, навіть не запущених

**docker build**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/command#docker-build)Команда дозволяє створити новий образ з усіма змінами, виконаними у контейнері:

docker build [OPTIONS] PATH | URL | -

Команда збірки docker створює образи Docker із файлу Dockerfile та «контексту». Контекст збірки - це набір файлів, розташованих за вказаним шляхом або URL-адресою. Процес збірки може посилатися на будь-який із файлів у контексті. Наприклад, ваша збірка може використовувати інструкцію COPY для посилання на файл у контексті. Параметр URL може належати до трьох типів ресурсів: репозиторіїв Git, попередньо упакованих контекстів tarball та файлів із простим текстом.

З повним переліком команд завжди можна познайомитись у [документації](https://docs.docker.com/engine/reference/commandline/docker/).

Давайте познайомимося з цими і не лише командами з практичного боку.

**Робота з Docker**

Ми розберемо практичне застосування основних команди, які найчастіше використовують програмісти у роботі з Docker.

Щоб завантажити з[**DockerHUB**](https://hub.docker.com/) образ і запустити контейнер з [**hello-world**](https://hub.docker.com/_/hello-world)**.**

Потрібно виконати команду:

docker run -it hello-world

Команда повинна повернути наступне повідомлення.

PS E:\WebDir> docker run -it 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.

(amd64)

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 ID:

https://hub.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/get-started/

**INFO**

Якщо образ hello-world був відсутній у системі, то автоматично вні буде завантажений з репозиторію DockerHUB командою docker pull hello-world, а потім буде запущений контейнер.

PS E:\WebDir> docker run -it hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

2db29710123e: Pull complete

Digest: sha256:18a657d0cc1c7d0678a3fbea8b7eb4918bba25968d3e1b0adebfa71caddbc346

Status: Downloaded newer image for hello-world:latest

...

Щоб подивитися на запущені контейнери, потрібно виконати команду.

docker ps

Результатом буде таблиця з основною інформацією про запущений контейнер.

Якщо додати необов'язковий прапорець -a, то побачимо ще й не запущені контейнери системи

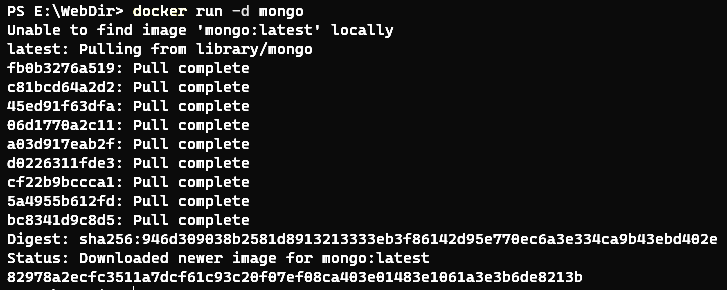
docker ps -a



У нашому випадку контейнер hello-world буде у списку не запущених. По-перше, ми не запускали його в режимі демона, а по-друге він виконався і відразу завершив свою роботу. Щоб контейнер працював у фоні, нам потрібен сервіс – наприклад образ з базою даних.

Давайте запустимо контейнер з базою даних [MongoDB](https://hub.docker.com/_/mongo), виконавши команду

docker run -d mongo



Якщо контейнер вже працює з прапором -d, то зайти в нього можна командою exec, головне вказати id hash контейнера. Цей hash ми можемо дізнатися двома способами, коли запускали контейнер або за допомогою команди docker ps. У моєму випадку це 82978a2ecfc3511a7dcf61c93c20f07ef08ca403e01483e1061a3e3b6de8213b. Не обов'язково вводити всі значення хешу, часто достатньо ввести перші 4-5 символів.

docker exec -it 8297 bin/sh

У команді ми додали bin/sh. Це говорить, що коли ми зайшли в контейнер, потрібно виконати команду і запустити термінал. З'явиться запрошення терміналу на початку із символом #. Виконаємо там команду mongod. Якщо сервер з базою даних запустився правильно, ми побачимо логи. Введіть команду exit, щоб вийти з контейнера.

Зупинити виконання контейнера можна командою stop, потім потрібно вказати або ім'я контейнера, або його hash (можна ввести перші цифри hash, головне щоб докер однозначно зрозумів, що потрібно зупинити)

docker stop <name|hash>

Повторний запуск контейнера виконується командою start

docker start <name|hash>

Видалення контейнера команда

docker rm <name|hash>

Щоб контейнер був нам "корисний", необхідно прокинути порти назовні. Сервер MongoDb працює всередині контейнера на порту 27017, нам потрібно зв'язати порт всередині контейнера та зовні за допомогою параметра -p 27017:27017. Ліворуч це наш порт 27017, який буде видно на комп'ютері для нашої програми. Праворуч порт 27017, який використовує контейнер у собі. Лівий порт ми можемо вказувати під час запуску будь-якого, головне, щоб він був вільним у системі. Але прийнято, що MongoDb працює на порту 27017.

docker run -p 27017:27017 -d mongo

Тепер ми можемо отримати доступ до бази даних всередині контейнера на порту 27017, що ми й робитимемо, коли почнемо вивчати MongoDb

Для того, щоб вивести збереження даних зовні контейнера, можна використовувати voluemes. Це дозволяє зберігати дані не всередині контейнера, а локально на диску, наприклад, у папці e/dbstorage.

docker run -p 27017:27017 -v e/dbstorage:/data/db -d mongo

Це дозволяє навіть у разі видалення поточного контейнера не втратити актуальну базу даних. І ми можемо запустити новий контейнер, вказавши при цьому voluemes, і дані будуть ті самі.

Щоб побачити всі образи, які ми завантажили в систему, виконайте команду:

docker images

Наступним кроком ми створимо свій образ з **Dockerfile.**

**Докерезація застосунку**

**Створення базового застосунку Flask**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B1%D0%B0%D0%B7%D0%BE%D0%B2%D0%BE%D0%B3%D0%BE-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BA%D1%83-flask)Для прикладу ми зробимо базовий застосунок Flask з виведенням «Hello World!» у браузері. По-перше, створіть каталог для нашого проекту

mkdir flask-docker-app

cd flask-docker-app

Відкрийте редактор коду та створіть 2 файли.

* app.py - Наш основний скрипт
* requirements.txt — тут вказуються всі пакети, що використовуються у вашому проекті.

Налаштуйте віртуальне середовище для нашого застосунку:

python -m venv env

source env/bin/activate

Встановлюємо **Flask**

pip install Flask

Коли Flask встановлений, він завантажує інші пакети, які йому необхідні для ефективної роботи. Щоб додати ці пакети у наш файл requirements.txt. Ми виконуємо команду:

python -m pip freeze > requirements.txt

Ви матимете схожий вміст файлу requirements.txt, який відрізняється тільки версіями.

click==8.1.3

colorama==0.4.5

Flask==2.2.2

itsdangerous==2.1.2

Jinja2==3.1.2

MarkupSafe==2.1.1

Werkzeug==2.2.2

**INFO**

Якщо ми використовуємо pipenv як віртуальне середовище, то команда створення файлу requirements.txt:

pipenv lock --keep-outdated --requirements > requirements.txt

Для poetry це команда:

poetry export --without-hashes --format=requirements.txt > requirements.txt

Потім ми додамо наступний код в app.py.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello():

return "Hello World!"

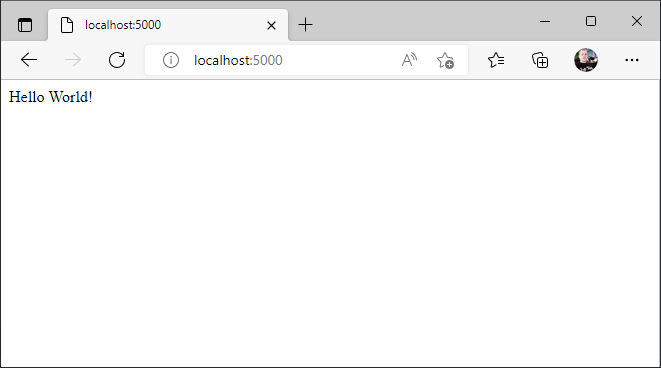
if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Тут ми для головного роутера @app.route('/') виводимо повідомлення "Hello World!". Запустимо app.py

python app.py

Програма **Flask** працюватиме на наступному URL <http://localhost:5000/>.



Завершіть роботу сервера, натиснувши «Ctrl+C». І почнемо докерезацію нашого застосунку.

**Створюємо Dockerfile.**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-dockerfile)У редакторі коду створіть файл Dockerfile (без розширень) всередині папки проекту. Потім додайте до нього наступний код:

Dockerfile

# Docker-команда FROM вказує базовий образ контейнера

# Наш базовий образ - це Linux з попередньо встановленим python-3.10

FROM python:3.10

# Встановимо змінну середовища

ENV APP\_HOME /app

# Встановимо робочу директорію всередині контейнера

WORKDIR $APP\_HOME

# Скопіюємо інші файли в робочу директорію контейнера

COPY . .

# Встановимо залежності всередині контейнера

RUN pip install -r requirements.txt

# Позначимо порт, де працює застосунок всередині контейнера

EXPOSE 5000

# Запустимо наш застосунок всередині контейнера

ENTRYPOINT ["python", "app.py"]

Коротко розглянемо різні інструкції, призначені для використання у файлах Dockerfile. Ми не заглиблюватимемося у подробиці, а повна та точна документація з інструкцій доступна на [сайті Docker](http://docs.docker.com/reference/builder/). Коментарі у файлах Dockerfile записуються за допомогою символу # на самому початку рядка.

**FROM**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#from)

Визначає основний образ для файлу Dockerfile. У нашому випадку це образ [python](https://hub.docker.com/_/python). Всі наступні інструкції виконують операції створення поверх заданого образу. Основний образ визначається в формі IMAGE:TAG (наприклад, python:3). За відсутності тега за замовчуванням належить "latest", але це погана практика вказувати цей тег. Ця інструкція обов’язково повинна бути найпершою в Dockerfile.

**ENV**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#env)

Визначає змінні середовища всередині образу. На ці змінні можна посилатися у наступних інструкціях. Наприклад: ... ENV MY\_VERSION 1.3 RUN apt-get install -y mypackage=$MY\_VERSION ... Визначені в цій інструкції змінні будуть доступними також всередині образу.

**WORKDIR**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#workdir)

Визначає робочий каталог для всіх наступних інструкцій RUN, CMD, ENTRYPOINT, ADD, COPY. Інструкцію можна використовувати кілька разів. Допускається зазначення відносних шляхів, причому підсумковий шлях визначається щодо раніше вказаного робочого каталогу WORKDIR.

**COPY**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#copy)Використовується для копіювання файлів із контексту створення в образ. Має два формати:

COPY "джерело" "мета"

і

COPY ["джерело", "мета"]

Обидва варіанти копіюють файл або каталог з "джерело", у контексті створення, в "мета" всередині контейнера. Формат не передбачає зазначення шляху "джерело", розташованих поза межами контексту створення — не можна вказати файл для копіювання ../some\_folder/some\_file.ext.

**RUN**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#run)Запускає задану інструкцію всередині контейнера та зберігає результат.

**EXPOSE**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#expose)Повідомляє Docker, що в цьому контейнері існуватиме процес, який прослуховує заданий порт або кілька портів. Docker використовує цю інформацію при встановленні з’єднання між контейнерами або під час відкриття портів для спільного доступу за допомогою аргументу -p у команді docker run.

**ENTRYPOINT**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#entrypoint)Визначає файл, що виконується, і аргументи за замовчуванням, який запускається під час ініціалізації контейнера. Також у цю програму, що виконується, передаються як аргументи будь-які інструкції CMD або аргументи команди docker run, записані після імені образу.

**ADD**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#add)Копіює файли з контексту створення або з URL-посилань в образ, що створюється. Якщо архівний файл додається з локального шляху, то він буде автоматично розпакований. Оскільки діапазон функціональності інструкції ADD дуже великий, у загальному випадку краще скористатися простішою командою COPY для копіювання файлів та каталогів.

**CMD**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#cmd)Запускає інструкцію під час ініціалізації контейнера. Інструкція CMD заміщається будь-якими аргументами, вказаними у команді docker run після імені образу. Насправді виконується лише остання інструкція CMD, а всі попередні інструкції CMD будуть скасовані.

**MAINTAINER**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#maintainer)

Визначає метадані про автора Author для образу, що створюється в заданому рядку. Витягти ці метадані можна за допомогою команди docker inspect -f {{.Author}} IMAGE. Зазвичай використовується для запису імені автора образу та його контактних даних.

**VOLUME**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#volume)

Оголошує заданий файл або каталог як том. Якщо такий файл або каталог вже існує в образі, він копіюється у том під час запуску контейнера. Якщо задано кілька аргументів, то вони інтерпретуються як визначення кількох томів.

**Створюємо образ**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-%D0%BE%D0%B1%D1%80%D0%B0%D0%B7)Тепер перед створенням образу необхідно відредагувати файл app.py. Вказуємо, що

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=False, host='0.0.0.0')

Потім створіть образ докеру наступною командою. При виконанні команди, ви повинні перебувати в корені проекту.

У команді замість krabaton повинно бути ім’я вашого користувача, яке ви зареєстрували на сайті [https://hub.docker.com](https://hub.docker.com/) (*прим*. Автор використав своє ім’я користувача). Замість flask-docker можна задати своє ім’я образу

docker build . -t krabaton/flask-docker

Запустіть процес створення образу і у вас почнеться виведення процесу у консолі:

[+] Building 5.1s (10/10) FINISHED

=> [internal] load build definition from Dockerfile 0.0s

=> => transferring dockerfile: 32B 0.0s

=> [internal] load .dockerignore 0.0s

=> => transferring context: 2B 0.0s

=> [internal] load metadata for docker.io/library/python:3.10 1.5s

=> [auth] library/python:pull token for registry-1.docker.io 0.0s

=> [1/4] FROM docker.io/library/python:3.10@sha256:e9c35537103a2801a30b15a77d4a56b35532c964489b125ec1ff24f3d5b53409 0.0s

=> [internal] load build context 0.0s

=> => transferring context: 711B 0.0s

=> CACHED [2/4] WORKDIR /app 0.0s

=> [3/4] COPY . . 0.0s

=> [4/4] RUN pip install -r requirements.txt 3.3s

=> exporting to image 0.1s

=> => exporting layers 0.1s

=> => writing image sha256:2fe6ac3db17346e9d6ef833fff86f9041dada3849c1cd8fa53b084dafe94609b 0.0s

=> => naming to docker.io/krabaton/flask-docker

Як бачимо, образ був створений за 10 кроків та зайняло це 5.1 секунди. Ваш час може відрізнятися у більший чи менший бік, все залежить від продуктивності машини.

Запустимо контейнер зі створеного образу.

docker run -itd -p 3000:5000 krabaton/flask-docker

Після запуску контейнера наш застосунок доступний на наступному URL <http://localhost:3000/>. Це дозволяє протестувати наш застосунок локально.

**Висновок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-02/module-02-02/dockerfile#%D0%B2%D0%B8%D1%81%D0%BD%D0%BE%D0%B2%D0%BE%D0%BA)

Ми створили застосунок, поклали його в образ і запустили як контейнер на нашій машині. Ви можете приступати до виконання домашнього завдання.

Також для подальшого поглиблення у тему використовуйте офіційний [Playground](https://www.docker.com/play-with-docker/)