

Міністерство освіти і науки України
Національний авіаційний університет
Навчально-науковий інститут комп'ютерних інформаційних технологій
Кафедра комп'ютеризованих систем управління

Домашнє завдання
з дисципліни «Системне програмне забезпечення»
на тему «Платформа контейнеризації Docker»

Виконав:
студент ННІКІТ
групи СП-325
Клокун В. Д.
Перевірив:
Глазок О. М.

Київ 2019

ЗМІСТ

1	Завдання	3
2	Що таке платформа Docker	3
2.1	Системні засоби, які використовує платформа Docker	3
2.1.1	Простір ядра і користувача	3
2.1.2	Віртуалізація на рівні операційної системи	4
2.2	Основні поняття платформи Docker	4
2.3	Призначення платформи Docker	5
2.4	Компоненти платформи Docker	5
3	Практичне використання платформи Docker	6
3.1	Установка платформи Docker	6
3.2	Контейнери	8
3.2.1	Створення Docker-образів	8
3.2.2	Створення контейнерів з образів	11
3.3	Сервіси	14
3.3.1	Опис сервісів	14
3.3.2	Запуск і управління сервісами	15
4	Висновок	17

1. ЗАВДАННЯ

Описати платформу Docker і основні способи її використання.

2. ЩО ТАКЕ ПЛАТФОРМА DOCKER

2.1. Системні засоби, які використовує платформа Docker

Найважливіша функціональність платформи Docker побудована на основі засобів операційних систем, розроблених задовго до початку розробки платформи Docker. Інновація цієї платформи в тому, наскільки зручною вона робить використання цих засобів. Тому, щоб розібратись, що таке платформа Docker, спочатку розглянемо системні засоби, які зумовили її створення, на основі яких вона побудована і які вона використовує.

2.1.1. Простір ядра і користувача

Щоб захистити пам'ять і апаратне забезпечення від зловмисного використання або помилкової поведінки, сучасні операційні системи розділяють віртуальну пам'ять на два шари: простір ядра і простір користувача. *Простір ядра* — це область віртуальної пам'яті, в якій ядро операційної системи зберігається, виконується і надає свої сервіси.

Для правильної роботи ядру операційної системи необхідно багато функцій, що працюють на низькому рівні або виконують чутливі операції: прямий доступ до апаратного забезпечення, зберігання, зчитування і запис конфіденційних даних, виконання операцій, які при неправильному використанні можуть вивести систему з ладу. Ці функції є потенційно небезпечними, бо при неправильному використанні можуть вивести систему з ладу або порушити її безпеку. Крім того, більшість з них не потрібні користувачам напряму. Отже, в просторі ядра зберігається виключно ядро операційної системи і необхідні йому дані.

Щоб підвищити рівень стабільності і безпеки операційної системи, існує другий шар — *простір користувача*, тобто місце, де зберігається і виконується прикладний код і необхідні йому дані (користувацьких програм тощо). Інакше кажучи, у просторі користувача зберігається все, що не є ядром операційної системи: код бібліотек і прикладних виконуваних файлів, процеси користувачів тощо.

Порівняно із системним кодом, код, що виконується у користувацькому просторі, має менший доступ до ресурсів системи і менше прав з метою безпеки. Щоб отримати доступ до ресурсів ядра операційної системи, код з простору користувача викликає спеціальні функції — *системні виклики*.

2.1.2. Віртуалізація на рівні операційної системи

Такий підхід дозволяє використовувати технологію віртуалізації на рівні операційної системи — таку технологію віртуалізації, при якій ядро операційної системи надає можливість створювати декілька ізольованих інстанцій користувачього простору. При віртуалізації на рівні операційної системи ядро операційної системи залишається одним і тим же, змінюється лише простір користувача: системні бібліотеки, програми, файлова система, середовище виконання (англ. *runtime environment*) тощо.

2.2. Основні поняття платформи Docker

Docker — це програмна платформа для віртуалізації на рівні операційної системи. Вона дозволяє розробляти, розгортати і запускати прикладні програми за допомогою контейнерів. Використання контейнерів для розгортання прикладних програм називається *контейнеризацією*. Контейнеризація стала популярною технологією, бо контейнери:

1. Гнучкі. Навіть найскладніші прикладні програми можна контейнеризувати.
2. Легкі. Контейнери використовують ядро основної операційної системи.
3. Легкозамінні. Можна розгортати оновлення «на льоту», під час роботи існуючого контейнера.
4. Портативні. Можна побудувати контейнер на одному комп'ютері, завантажити його на віддалений сервер і запустити будь-де.
5. Масштабовані. Можна збільшувати і автоматично розподіляти контейнери-репліки.
6. Нагромаджувані (англ. *stackable*) — можна нагромаджувати сервіси вертикально і «на льоту».

ОБРАЗИ І КОНТЕЙНЕРИ У контексті платформи Docker, *образом* називається виконуваний пакет, який містить усі засоби, необхідні для запуску прикладної програми: її код, середовище виконання, бібліотеки, змінні середовища (англ. *environment variables*) і конфігураційні файли. Коли образ запускають на виконання, з'являється *контейнер* — конкретний запущений примірник певного образу.

КОНТЕЙНЕРИ І ВІРТУАЛЬНІ МАШИНИ На відміну від віртуальних машин, які запускають повноцінну гостьову операційну систему, щоб виконувати у ній потрібні програми, контейнери запускаються безпосередньо на операційній системі — хазяїні (англ. *host operating system*), використовують її ядро і ресурси на пряму. Тим не менш, ядро операційної системи повністю ізолює контейнери

одне від одного і операційної системи — хазяїна, якщо не вказано інакше.

2.3. Призначення платформи Docker

Багатьом прикладним програмам для роботи необхідні зовнішні бібліотеки або інші засоби на кшталт баз даних і черг задач. Під час розробки прикладних програм, вони часто тестуються на конкретних версіях цих зовнішніх бібліотек і засобів, тобто їх працездатність гарантується лише для певних версій. Однак, зазвичай на різних обчислювальних машинах встановлені різні операційні системи, різні версії необхідних бібліотек і засобів, або ці засоби взагалі відсутні. Часто це означає, що розроблена програма не зможе запуститись і правильно працювати на таких обчислювальних машинах. Ця проблема називається *проблемою управління конфігураціями*.

Щоб вирішити цю проблему і уникнути необхідності вручну переконуватись, що на усіх машинах, де повинна виконуватись програма, встановлені потрібні засоби правильної версії, використовують платформу Docker. Як було сказано у розділі 2.2, вона дозволяє розробляти, розгортати і запускати прикладні програми за допомогою контейнерів. Контейнери створюються за допомогою образів — виконуваних пакетів, в яких встановлені власні бібліотеки, налаштовані середовища виконання та інші необхідні засоби. Ці пакети незмінні, тобто якщо запустити на різних машинах різні контейнери, створені з однакового образу, всередині цих контейнерів буде однакове середовище.

Розробники використовують цю властивість, щоб з впевненістю запускати програму на різних обчислювальних машинах. Для цього програми розробляють так, щоб вони могли запускатись у контейнерах: коригують архітектуру, розбиваючи її на модулі, створюють образ, всередині якого має працювати програма, і коли потрібно запустити програму на іншій машині, її запускають за допомогою створеного образу. Таким чином, якщо програма протестована і налагоджена для роботи всередині описаного образу, можна впевнено стверджувати, що вона запрацює у будь-якому контейнері, створеному на основі описаного образу.

У платформи Docker є багато призначень, крім управління конфігураціями: поєднання контейнерів у багатоконтейнерні сервіси, балансування навантаження, реплікація і оркестрація (автоматичне розгортання) тощо, але всі вони базуються на основній ідеї контейнера, створеної для управління конфігурацією одного примірника.

2.4. Компоненти платформи Docker

Платформа Docker є «програмою як сервісом» (англ. *software-as-a-service, SAAS*), яка складається з декількох компонентів, що взаємодіють між собою. Такими

компонентами є:

1. Прикладні програми, до яких входять:
 - 1.1. Демон `dockerd` — постійний процес, який працює на фоні і керує Docker-контейнерами і об'єктами. Цей демон управляється запитами, відправленими за допомогою Docker Engine API.
 - 1.2. Програма `docker` — прикладна клієнтська програма, яка надає командний інтерфейс для взаємодії з демоном платформи Docker.
2. Об'єкти — різні сутності, на основі яких складаються різні модулі прикладних програм на платформі Docker. Такими сутностями є:
 - 2.1. Контейнер — стандартизоване ізольоване середовище, яке виконує прикладну програму або її окремий компонент. Контейнери управляються за допомогою командного інтерфейсу або Docker API.
 - 2.2. Образ — пакет, який містить середовище з усім необхідним для запуску контейнера і за допомогою якого вони будуються. Образи використовують, щоб зберігати і доставляти прикладні програми користувачам або іншим кінцевим точкам.
 - 2.3. Сервіс — набір контейнерів, які можна масштабувати і виконувати під управлінням декількох різних демонів Docker. Результат такого масштабування називають *роєм* (англ. *swarm*).
3. Регістри — репозиторії, в яких зберігаються Docker-образи. Клієнти підключаються до реєстрів, щоб завантажити («витягнути», англ. *pull*) образи на свої машини або помістити (англ. *push*) їх у реєстр.

Також платформа Docker надає утиліти, які спрощують її використання або розширити її функціональність, а саме:

1. Docker Compose — утиліта для визначення, запуску і управління прикладними програмами, побудованими на декількох контейнерах. Сервіси, з яких складається прикладна програма, визначаються у спеціальних файлах формату YAML під назвою `docker-compose.yml`.
2. Docker Swarm — утиліта, за допомогою якої Docker-контейнери об'єднують і організовують у кластери.

Як видно, платформа Docker надає повнофункціональний набір для розробки і розгортання прикладних програм різної складності у різних ситуаціях.

3. ПРАКТИЧНЕ ВИКОРИСТАННЯ ПЛАТФОРМИ DOCKER

3.1. Установка платформи Docker

Засоби платформи Docker розповсюджується у декількох варіантах: Docker Community Edition (CE) і Docker Enterprise Edition (EE). Компанія Docker, Inc. позиціонує варіант Community Edition як ідеальний для розробників і невеликих команд та експериментів з технологією контейнеризації, а Enterprise

Edition як ідеальний для компаній і широкомасштабних рішень. Тим не менш, інструменти Docker Community Edition є повнофункціональними і підходять для використання навіть для корпоративних клієнтів, тому використаємо саме цей варіант, щоб показати можливості платформи.

На момент виконання даного завдання продукт Docker Community Edition має дві редакції: Desktop і Server. Редакція Desktop призначена для запуску на персональних комп'ютерах, а редакція Server — на серверних. Ці версії доступні для багатьох важливих платформ (табл. 1).

Табл. 1: Апаратно-програмні платформи, що підтримуються продуктом Docker Community Edition редакцій Desktop і Server

Платформа	x86_64 / amd64	ARM	ARM64 / AArch64	IBM Power	IBM Z
Desktop					
Windows 10	+				
macOS	+				
Server					
CentOS	+				
Debian	+	+	+		
Fedora	+				
Ubuntu	+	+	+	+	+

Щоб встановити Docker на підтримуваній платформі, треба перейти за посиланням, де описана інформація про установку (<https://docs.docker.com/install/>), знайти потрібну платформу і слідувати наведеним вказівкам.

Після того, як продукт Docker буде встановлений, варто перевірити його працездатність. Для цього виконаємо команду (можливо, знадобиться запустити команду від імені суперкористувача за допомогою команди `sudo`):

```
1 docker --version
```

Якщо Docker встановлений правильно, результат виконання команди виглядатиме приблизно так:

```
1 Docker version 17.12.0-ce, build c97c6d6
```

3.2. Контейнери

Контейнери — це найменша структурна одиниця платформи Docker. Усі інші структурні одиниці будуються на основі контейнерів. Розглянемо основні задачі, які виникають під час роботи з ними.

3.2.1. Створення Docker-образів

Як було сказано у розділі 2.2, контейнери є конкретними примірниками образів, тобто щоб запустити контейнер, треба спочатку побудувати образ. Щоб створити Docker-образ, використовують спеціальні файли, які називаються «Docker-file». Ці файли містять інструкції для створення образів.

Розглядатимемо ці файли на прикладі прикладної програми — веб-додатку, який вітатиме користувача та виводитиме лічильник відвідувань. Веб-додаток напишемо на мові програмування Python, використовуючи базу даних Redis. Веб-додаток складатиметься з двох файлів: `requirements.txt` (лістинг 3.1) і `app.py` (лістинг 3.2).

Лістинг 3.1: Файл `requirements.txt`, який описує модулі, необхідні для роботи веб-додатку

```
1 Flask
2 Redis
```

Лістинг 3.2: Файл `app.py`, який описує веб-додаток

```
1 from flask import Flask
2 from redis import Redis, RedisError
3 import os
4 import socket
5
6 # Connect to Redis
7 redis = Redis(host="database", db=0, socket_connect_timeout=2,
8               socket_timeout=2)
9
10 app = Flask(__name__)
11
12 @app.route("/")
13 def hello():
14     try:
15         visits = redis.incr("counter")
16     except RedisError:
```



```

16         visits = "<i>cannot connect to Redis, counter  

    ↪ disabled</i>"
17
18     html = "<h3>Hello {name}!</h3>" \
19           "<b>Hostname:</b> {hostname}<br/>" \
20           "<b>Visits:</b> {visits}"
21     # Тут використовується змінна середовища "\allcaps{NAME}"
22     return html.format(name=os.getenv("\allcaps{NAME}", "world"),
    ↪ hostname=socket.gethostname(), visits=visits)
23
24 if __name__ == "__main__":
25     # Запустити веб-сервер на~порті 80
26     app.run(host='0.0.0.0', port=80)

```

Щоб створити контейнер, в якому можна запустити розроблений веб-додаток з базовою функціональністю (робота веб-сайту, привітання, крім лічильника), необхідно:

1. Скопіювати файли, необхідні для роботи: requirements.txt, app.py.
2. Встановити модулі, від яких залежить веб-додаток.
3. Відкрити порт, за яким буде доступний веб-додаток.
4. Визначити змінну середовища, яка використовується у веб-додатку.
5. Запустити сам додаток.

Отже, враховуючи вищеописані вимоги, створюємо відповідний Dockerfile. Для цього у тій же директорії, в якій знаходяться файли requirements.txt і app.py, створюємо файл під назвою Dockerfile (лістинг 3.3).

Лістинг 3.3: Dockerfile для створення образу веб-додатку

```

1  # Використовувати офіційний образ Python 3.6 під управлінням Alpine Linux
2  # як~основу для образу, що~буде створений
3  FROM python:2.7-slim
4
5  # Всередині контейнера перейти в~директорію "/app"
6  WORKDIR /app
7
8  # Скопіювати зміст директорії, в~якій знаходиться Dockerfile у~контейнер
9  # у~директорію "/app"
10 COPY . /app
11
12 # Запустити команду "pip install ...", щоб встановити модулі, від яких
13 # залежить веб-додаток
14 RUN pip install --trusted-host pypi.python.org -r requirements.txt
15
16 # Відкрити порт 80 для зовнішніх пристроїв

```

```

17 EXPOSE 80
18
19 # Визначити змінну середовища "NAME" і~присвоїти їй~значення "World"
20 ENV NAME World
21
22 # Запустити команду "python app.py", коли контейнер запуститься
23 CMD ["python", "app.py"]

```

Тепер необхідно створити образ. Для цього переходимо у директорію, яка містить створений Dockerfile і запускаємо побудову образу, описаного у файлі Dockerfile поточної директорії. Дамо цьому образу за допомогою параметру `-t` назву `webapp`. Ці дії виконуються за допомогою такої команди:

```
docker image build -t=webapp:0.1 .
```

В результаті був побудований необхідний образ (рис. 1).

```

2. bash
bash %1  bash %2
Build an image from a Dockerfile
vlads-mbp:docker-test myownreality$ docker image build -t=webapp:0.1 .
Sending build context to Docker daemon  5.12kB
Step 1/7 : FROM python:2.7-slim
--> eb40dcfc42
Step 2/7 : WORKDIR /app
--> Using cache
--> b31ebff8697c
Step 3/7 : COPY . /app
--> Using cache
--> 13aadcbd38e3
Step 4/7 : RUN pip install --trusted-host pypi.python.org -r requirements.txt
--> Using cache
--> fbe2457108f6
Step 5/7 : EXPOSE 80
--> Using cache
--> 7f5263c6f970
Step 6/7 : ENV NAME World
--> Using cache
--> ec1c06a7f2c7
Step 7/7 : CMD ["python", "app.py"]
--> Using cache
--> ce717b5605a1
Successfully built ce717b5605a1
Successfully tagged webapp:0.1
vlads-mbp:docker-test myownreality$

```

Рис. 1: Результат побудови образу

Переконаємось, що бажаний образ дійсно створений. Для цього запустимо таку команду:

```
docker image ls
```

Бачимо, що створений образ є у списку (рис. 1).



```
vlads-mbp:docker-test myownreality$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
webapp               0.1                ce717b5605a1        5 minutes ago
131MB
python              2.7-slim           eb40dcfc42          2 weeks ago
120MB
vlads-mbp:docker-test myownreality$
```

Рис. 2: Список наявних образів

3.2.2. Створення контейнерів з образів

Щоб запустити образ на виконання, тобто створити контейнер, використовують команду `docker run`. Наприклад, запустимо створений контейнер, пере-направивши відкритий порт 80 на порт 4000 за допомогою параметра `-p`; також за допомогою параметра `--rm` вкажемо, що після завершення роботи контейнер треба видалити. Це досягається такою командою:

```
docker run --rm -p 4000:80 webapp:0.1
```

В результаті контейнер запущений і працює на передньому плані (рис. 3а), а тому працює і веб-додаток (рис. 3б).

Найчастіше контейнери запускають у фоновому режимі, тобто на задньому плані. У такому випадку контейнер не займає термінал, а виконується у фоновому процесі. Тим не менш, він доступний для управління і відстеження за допомогою спеціальних команд.

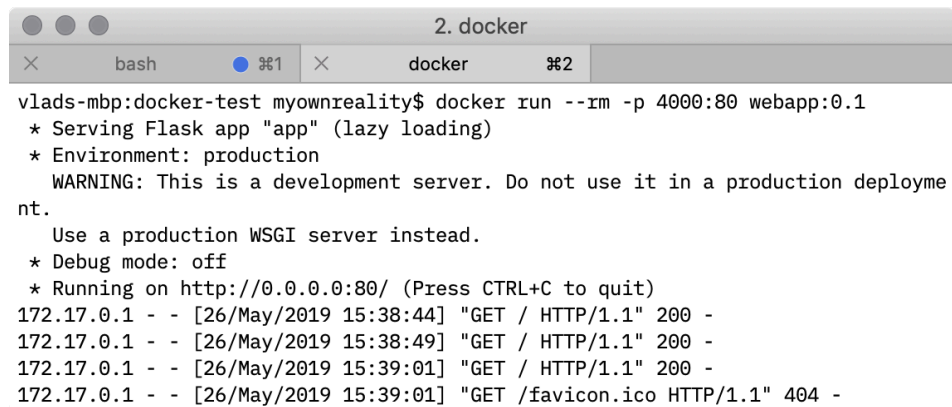
Отже, запустимо контейнер так само, як і у попередній раз, але тепер у фоновому режимі за допомогою параметра `-d`. Для цього спочатку перейдемо у активне вікно емулятора терміналу і завершимо виконання контейнера, який працює на передньому плані, натиснувши комбінацію клавіш `Ctrl + C`, а потім запустимо контейнер за допомогою відповідної команди:

```
docker run -d --rm -p 4000:80 webapp:0.1
```

В результаті побачимо, що у вікно емулятора терміналу був виведений ідентифікатор запущеного контейнера. Перевіримо, які команди зараз запущені. Для цього виконаємо таку команду:

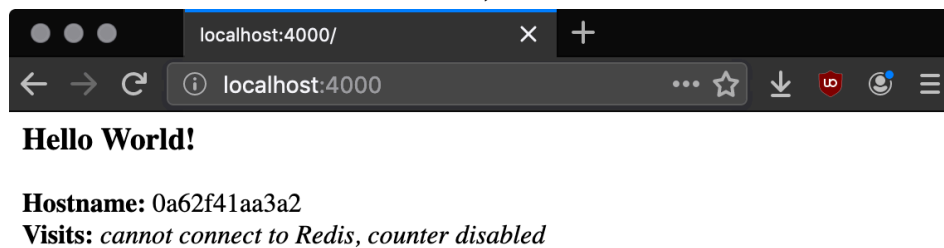
```
docker ps
```

Бачимо, що контейнер запущений на виконання і успішно працює (рис. 4а). Отже, працює і веб-додаток (рис. 4б).



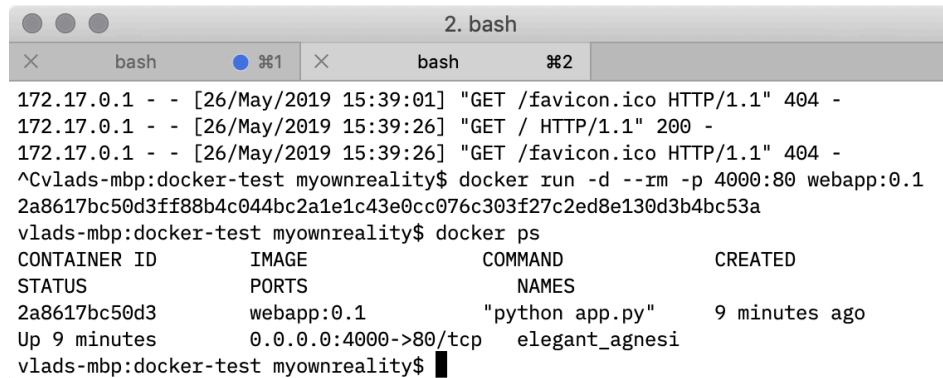
```
2. docker
bash %1 docker %2
vlads-mbp:docker-test myownreality$ docker run --rm -p 4000:80 webapp:0.1
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
172.17.0.1 - - [26/May/2019 15:38:44] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [26/May/2019 15:38:49] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [26/May/2019 15:39:01] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [26/May/2019 15:39:01] "GET /favicon.ico HTTP/1.1" 404 -
```

a)



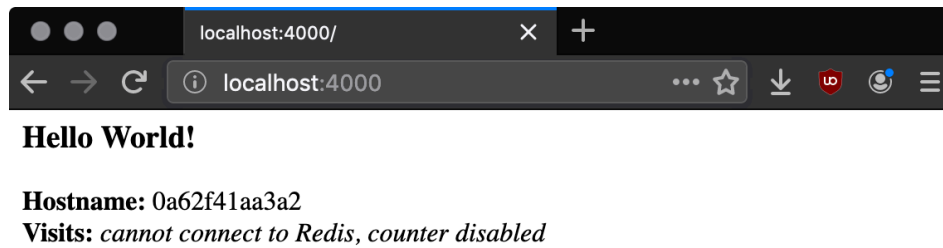
б)

Рис. 3: Результат створення і запуску контейнера: а — вікно терміналу, б — веб-додаток у браузері



```
172.17.0.1 - - [26/May/2019 15:39:01] "GET /favicon.ico HTTP/1.1" 404 -
172.17.0.1 - - [26/May/2019 15:39:26] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [26/May/2019 15:39:26] "GET /favicon.ico HTTP/1.1" 404 -
^Cvlads-mbp:docker-test myownreality$ docker run -d --rm -p 4000:80 webapp:0.1
2a8617bc50d3ff88b4c044bc2a1e1c43e0cc076c303f27c2ed8e130d3b4bc53a
vlads-mbp:docker-test myownreality$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS            PORTS              NAMES
2a8617bc50d3        webapp:0.1         "python app.py"     9 minutes ago
Up 9 minutes       0.0.0.0:4000->80/tcp elegant_agnesi
vlads-mbp:docker-test myownreality$
```

а)



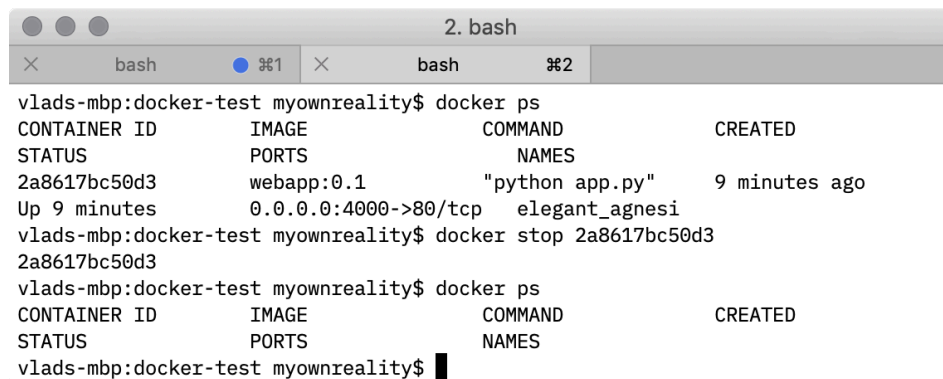
б)

Рис. 4: Результат запуску контейнера у фоновому режимі: а — вікно терміналу, б — веб-додаток у браузері

Тепер зупинимо працюючий контейнер. Для цього використовується команда `docker stop`. Як видно з вікна терміналу (рис. 4а), запущений контейнер має ідентифікатор `2a8617bc50d3`. Щоб зупинити цей контейнер, виконаємо таку команду:

```
docker stop 2a8617bc50d3
```

В результаті контейнер був зупинений (рис.5).



```
vlads-mbp:docker-test myownreality$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS        PORTS    NAMES
2a8617bc50d3   webapp:0.1 "python app.py"         9 minutes ago
Up 9 minutes   0.0.0.0:4000->80/tcp    elegant_agnesi
vlads-mbp:docker-test myownreality$ docker stop 2a8617bc50d3
2a8617bc50d3
vlads-mbp:docker-test myownreality$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS        PORTS    NAMES
vlads-mbp:docker-test myownreality$
```

Рис. 5: Зупинка контейнера, який працює у фоновому режимі

3.3. Сервіси

Сервіси — це різні частини розподіленого програмного забезпечення. Наприклад, у програмного забезпеченні, яке ми розглядаємо для прикладу, є два сервіси: веб-додаток і база даних. Взагалі, сервіси можна вважати «контейнерами у роботі» (англ. *containers in production*), тому що сервіс виконує лише один образ, але описує параметри, як цей образ повинен виконуватись: які порти використовувати, які мережі використовувати, при яких умовах перезавантажуватись тощо.

Використання сервісів дозволяє зручно визначати багатоконтейнерні інстанції для прикладних програм. Оскільки в програми, яка розглядається для прикладу, передбачено два сервіси, розглянемо її.

3.3.1. Опис сервісів

Зручну роботу з сервісами забезпечує утиліта Docker Compose, яка дозволяє створювати і управляти сервісами. При роботі з Docker Compose, сервіси описують за допомогою спеціальних файлів `docker-compose.yml`. Опишемо сервіси для нашої програми.

Як було сказано раніше, програма складається з двох сервісів: веб-додатку і бази даних. Щоб це описати, у директорії проекту створюємо відповідний файл `docker-compose.yml` (лістинг 3.4).

Лістинг 3.4: Файл `docker-compose.yml` для опису сервісів програми

```
1  # Використовувати Docker Compose версії 3
2  version: "3"
3  # Опис сервісів
4  services:
5    # Сервіс "webapp"
6    webapp:
7      # Використати образ "webapp:0.1"
8      image: "webapp"
9      # Перенаправити порти
10     ports:
11       # На~порт хоста 4000 перенаправити порт контейнера 80
12       - "4000:80"
13     # Сервіс "database"
14     database:
15       # Використати образ "redis"
16       image: "redis"
17       volumes:
18         # Прив'язати том "dbvolume" на~хості до~директорії "/data"
19         # всередині контейнера
20         - "dbvolume:/data"
21
22   # Опис томів
23   volumes:
24     # Оголосити "dbvolume", щоб дані сервісу "database" зберігались
25     # навіть після зупинки контейнера. Дані будуть збережені доти,
26     # доки том "dbvolume" не~буде видалений
27   dbvolume:
```

Описавши всі сервіси, необхідні для роботи програми, можна переходити до їх запуску.

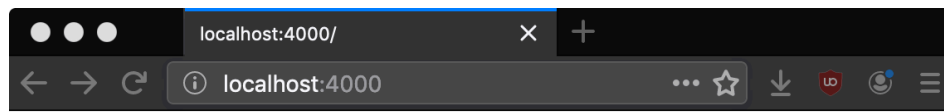
3.3.2. Запуск і управління сервісами

Тепер запусимо всі описані сервіси нашої програми за допомогою утиліти Docker Compose. Для цього виконаємо таку команду:

```
docker-compose up
```

В результаті бачимо, що запускаються два сервіси: база даних `database` і веб-додаток `webapp` (рис. 6а). Оскільки база даних запущена і працює, тепер лічильник справний (рис. 6б). Якщо ж сервіси треба запустити у фоновому режимі,

аналогічно до контейнерів, необхідно виконати вищезазначену команду з параметром `-d`.

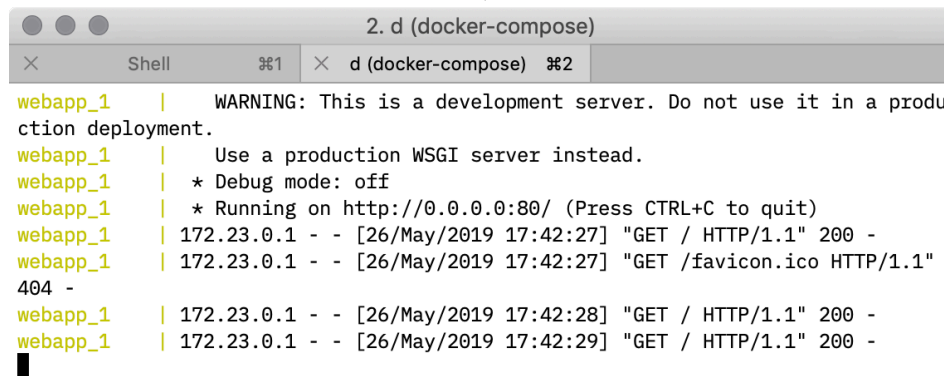


Hello World!

Hostname: 847e7e5003b2

Visits: 3

а)



б)

Рис. 6: Результат запуску мультисервісної програми: а — вікно терміналу, б — веб-додаток у браузері

Зупинимо запущені сервіси. Оскільки вони працюють на передньому плані, щоб їх зупинити достатньо натиснути комбінацію клавіш `Ctrl + C`. Після цього всі сервіси, запущені за описаним файлом `docker-compose.yml` будуть зупинені. Однак, це не означає, що була видалена вся інфраструктура, створена Docker Compose: ще залишились створена мережа, в якій знаходяться описані сервіси, а також том «`dbvolume`».

Щоб видалити створену інфраструктуру, тобто повністю зупинити сервіси і почистити результати їх виконання, треба виконати таку команду:

```
docker-compose down
```

В результаті її виконання будуть видалені усі мережі, томи та інші артефакти, описані у файлі `docker-compose.yml`.

4. ВИСНОВОК

Виконуючи дане домашнє завдання, ми ознайомились і описали платформу Docker. Ця платформа призначена для контейнеризації прикладних програм, що дозволяє зручно розповсюджувати програми, не переймаючись, що вони не запускатимуться на інших комп'ютерах.

Крім того, платформа Docker дозволяє ізолювати складові програмного продукту, не заважаючи обміну даних між ними, завдяки сервісам і спеціальним утилітам на кшталт Docker Compose, стеків і роїв. У рамках даного домашнього завдання ми розглянули розгортання багатосервісного програмного продукту за допомогою утиліти Docker Compose, яка дозволила зручно обмінюватись даними між описаними сервісами як за допомогою мережі, так і за допомогою файлової системи.

Загалом, завдяки відтворюваності об'єктів, які створюються інструментами платформи Docker, а також зручному управлінню за допомогою утиліт, які вона надає, ця платформа значно спрощує процес розробки, тестування і доставки розроблених програм кінцевим користувачам.