ОСНОВИ DOCKER

1. **Що таке docker?**
2. **Принципи роботи**

1. Що таке Docker

Docker — інструмент з відкритим сирцевим кодом, який автоматизує розгортання застосунку у середовищах, що підтримують контейнеризацію. Для кращого розуміння суті Docker проведемо аналогію зі звичайними транспортними контейнерами.

Колись у транспортних компаній виникали труднощі із:

- транспортуванням різних (несумісних) типів товарів разом (наприклад, їжі та хімікатів або скла та цегли);
- транспортуванням пакувань різних розмірів у одному й тому ж транспорті.

Після появи контейнерів з'явилася можливість класти цеглу над склом, а хімікати — поруч з їжею. Стандартний контейнер можна наповнювати вантажем різного розміру і завантажувати й розвантажувати його тим самим транспортним засобом.

Повернемося до **контейнерів у розробці ПЗ.**

Під час розробки застосунку вам треба потурбуватися про всі необхідні залежності: на зразок бібліотек, веб-сервера, баз даних тощо. Інакше застосунок може працювати на вашому комп'ютері, але не запуститися на staging сервері чи комп'ютерах інших розробників та тестувальників.

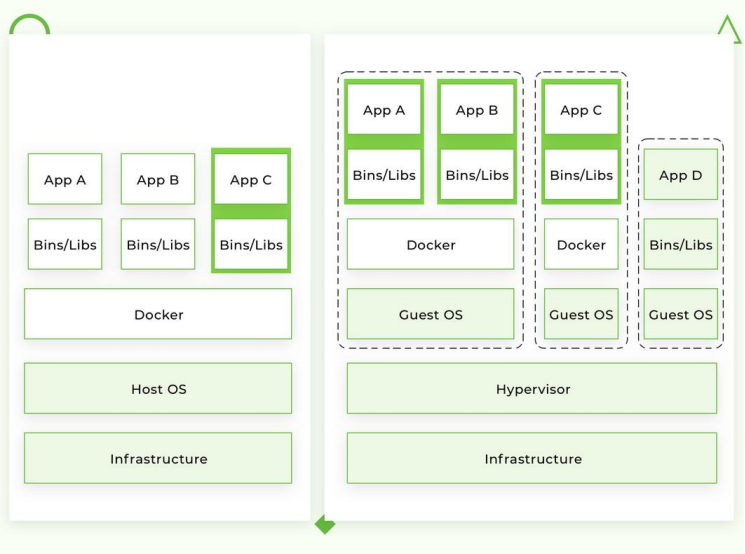
Таку проблему можна розв'язати, ізолювавши застосунок від системи.

Чим це відрізняється від віртуалізації

Зазвичай для уникнення неочікуваної поведінки застосунку використовувались віртуальні машини (VM). Основна проблема у тому, що VM — це *додаткова* ОС, що працює поверх хостової, а це додаткові гігабайти для проекту.

Часто ваш сервер обслуговує декілька VM, кожна з яких займає достатньо місця. А відомо, що більшість хмарних платформ стягує плату за використання додаткового простору. Ще один суттєвий недолік віртуальної машини — повільне завантаження.

Docker розв'язує усі перераховані проблеми, розділяючи ядро ОС між усіма контейнерами, що працюють як окремі процеси хостової ОС.



Варто пам'ятати, що Docker — не перша і не єдина платформа контейнеризації. Однак зараз це найпопулярніший та найпотужніший варіант серед аналогів.

Навіщо нам Docker

Через його переваги, серед яких:

- пришвидшення процесу розробки;
- зручна інкапсуляція застосунку;
- однакова поведінка на локальній машині, а також dev/staging/production серверах;
- простий і чіткий моніторинг;
- зручність масштабування.

Пришвидшення процесу розробки

Тепер у системі не потрібно встановлювати сторонні застосунки, на зразок PostgreSQL, Redis, Elasticsearch — запускайте їх в контейнерах. Docker також дозволяє запускати різні версії одного застосунку одночасно. Уявімо, що вам необхідно вручну перенести дані зі старої версії Postgres на нову. Також така ситуація виникає у мікросервісній архітектурі, коли ви хочете створити мікросервіс з новою версією стороннього ПЗ.

Завантаження двох різних версій того самого застосунку на хостову ОС викликає труднощі. За таких умов контейнери Docker будуть ідеальним рішенням: ви отримуєте ізольоване середовище для застосунку та сторонніх сервісів.

Зручна інкапсуляція застосунку

Більшість мов програмування, фреймворків та усі операційні системи мають свої пакетні менеджери. Якщо ваш застосунок використовує нативний пакетний менеджер, вам може бути складно створити порт для іншої системи.

Docker передбачає єдиний формат образу (image) для поширення застосунків між різними операційними системами та хмарними сервісами. Ваш застосунок тепер суцільний, має всі необхідні залежності та готовий до запуску.

Однакова поведінка на локальній машині, а також dev/staging/production серверах

Docker не може на 100% гарантувати однакову поведінку, адже завжди є людський фактор. Однак вірогідність помилки через різні версії операційних систем чи залежностей зводиться до мінімуму.

З правильним підходом до створення Docker-образу ваш застосунок використовуватиме його з відповідною версією ОС та необхідними залежностями.

Простий і чіткий моніторинг

Вже «з коробки» у вас є спосіб відстежувати логи усіх запущених контейнерів. Не треба запам’ятовувати, куди ваш застосунок та його залежності пишуть логи, та створювати кастомні хуки для налаштування цього процесу.

Ви можете інтегрувати зовнішній драйвер логування та відстежувати логи вашого застосунку в одному місці.

Зручність масштабування

Правильно огорнутий застосунок відповідає більшості критеріїв «Застосунку 12 факторів». За своїм задумом Docker змушує використовувати основні принципи: конфігурацію через змінні середовища, використання TCP/UDP тощо. Тобто якщо ваш застосунок правильно спроектований, він готовий до масштабування не лише у Docker.

Підтримка платформ

Linux — нативна платформа для Docker, тому він підтримує фічі ядра Linux. Однак Docker можна запускати на macOS і Windows. Різниця лише у тому, що на останніх ОС Docker інкапсульовано у крихітну віртуальну машину. Проте зараз Docker для macOS та Windows значно вдосконалився: він зручний для використання і дуже схожий на нативний інструмент.

Встановлення

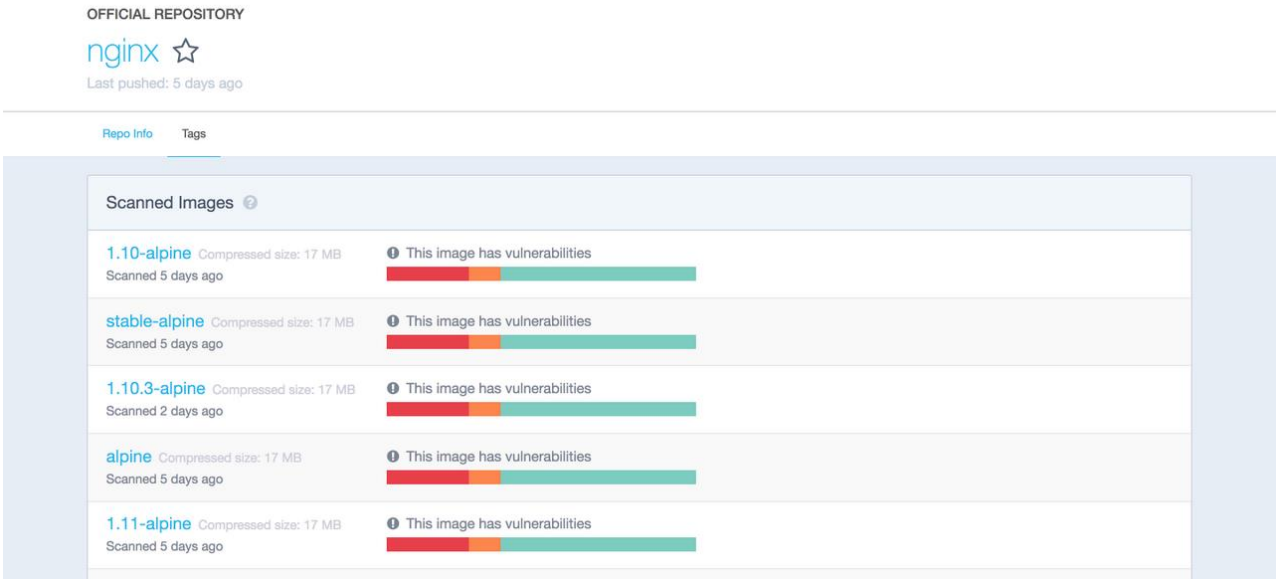
Інструкції зі встановлення Docker можна знайти за посиланням.

Якщо ви запускаєте Docker на Linux, вам треба виконати всі команди як `root` або додати вашого користувача до групи docker та повторно зайти.

```
sudo usermod -aG docker $(whoami)
```

Термінологія

- **Контейнер (Container)** — запущений екземпляр, що інкапсулює необхідне ПЗ. Контейнери завжди створюються з образу і можуть надавати порти та дисковий простір для взаємодії з іншими контейнерами чи/та зовнішнім ПЗ. Контейнери можна з легкістю знищити/видалити та створити знову. Контейнери не зберігають стан.
- **Образ (Image)** — базовий елемент кожного контейнеру. При створенні образу кожен крок кешується і може бути використаний повторно (копіювання під час запису). Час на збірку залежить від самого образу. З іншого боку: контейнери можна одразу запустити з образу.
- **Порт (Port)** — TCP/UDP порт у своєму звичному розумінні. Для спрощення припустимо, що порти можуть бути відкриті для зовнішнього ПЗ (доступні з хостової ОС) або підключатися до інших контейнерів (тобто доступні лише з цих контейнерів та невидимі для іншого ПЗ).
- **Volume** можна вважати спільною текою. Volume ініціалізується при створенні контейнеру і призначений для збереження даних, незалежно від життєвого циклу контейнера.
- **Registry (Сховище)** — сервер, що зберігає образи Docker. Ми можемо порівняти його з Github: витягуєте образ зі сховища, щоб розгорнути його локально, а потім відправляєте локально зібрані образи до віддаленого сховища.
- Docker Hub — сховище з веб-інтерфейсом від Docker Inc. Зберігає багато Docker-образів з різним ПЗ. Docker Hub — джерело «офіційних» образів Docker, створених їх командою або у співпраці з іншими компаніями (але це не обов'язково образи від офіційних виробників ПЗ). Якщо ви зареєстровані, можна переглянути перелік потенційних вразливостей таких образів. Доступні платні та безкоштовні облікові записи. Для безкоштовного облікового запису можна створювати один приватний образ та безліч публічних.
- Docker Store — сервіс дуже подібний на Docker Hub. Це майданчик з рейтингами, відгуками тощо.



Приклад 1: hello world

Час запустити ваш перший контейнер:

```
docker run ubuntu /bin/echo 'Hello world'
```

Результат в консолі:

```
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
6b98dfc16071: Pull complete
4001a1209541: Pull complete
```

```
6319fc68c576: Pull complete
b24603670dc3: Pull complete
97f170c87c6f: Pull complete
Digest:sha256:5f4bdc3467537cbbe563e80db2c3ec95d548a9145d64453b06939c4592d67b6d
Status: Downloaded newer image for ubuntu:latest
Hello world
```

- **docker run**— команда для запуску контейнера;
- **ubuntu** — образ, який ви запускаєте. Наприклад, образ ОС Ubuntu. Коли ви вказуєте образ, Docker спершу шукає його на вашому хості. Якщо образ там не знайдено, він витягується з публічного сховища — Docker Hub;
- **/bin/echo 'Hello world'**— команда, що виконується всередині нового контейнера і друкує вказаний текст.

Створимо інтерактивну командну оболонку всередині Docker-контейнера:

```
docker run -i -t --rm ubuntu /bin/bash
```

- -t прапор задає псевдо-tty або термінал всередині нового контейнера.
- -i прапор дозволяє здійснити інтерактивне з'єднання, перехопивши стандартний потік вводу (stdin) контейнера.
- --rm для автоматичного видалення контейнера при завершенні процесу. Зверніть увагу, що за замовчуванням контейнери не видаляються. Наш контейнер натомість існуватиме лише поки триватиме сеанс оболонки.

Якщо ви хочете, щоб контейнер продовжував виконання після завершення сеансу, треба зробити його демоном.

```
docker run --name daemon -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done
```

- --name daemon визначає назву нового контейнера. Якщо ви явно не вкажете назву, Docker згенерує її автоматично.
- -d прапор запускає контейнер як фоновий процес (тобто процес-демон).

Поглянемо, які контейнери ми вже створили:

```
docker ps -a
```

Результат в консолі:

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1fc8cee64ec2 ubuntu "/bin/sh -c 'while..." 32 seconds ago Up 30 seconds daemon
c006f1a02edf ubuntu "/bin/echo 'Hello ..." About a minute ago Exited (0) About a minute ago gifted_nobel
```

- docker ps — команда для відображення контейнерів;
- прапор -a потрібен, щоб показати всі контейнери (без нього отримаємо лише перелік запущених).

Тут ми бачимо два контейнери:

- gifted_nobel (назва згенерована автоматично, тобто у вас вона буде відрізнятися) — перший створений нами контейнер. Він друкує «Hello world».
- daemon — третій створений контейнер. Запускає фоновий процес.

Помітьте: тут немає другого контейнера (того, що з інтерактивною оболонкою), адже ми вказали прапор -rm. Тобто він автоматично видалився одразу після виконання.

Перевіримо логи, аби побачити, що робить зараз процес-демон:

```
docker logs -f daemon
```

В консолі маємо:

```
...
hello world
hello world
hello world
```

- docker logs — команда для отримання логів контейнера;
- прапор -f потрібен, щоб слідувати за результатом у консолі (працює так само, як tail -f).
  Зупинимо фоновий контейнер:
  ```
  docker stop daemon
  ```
  Переконаємось, що він дійсно зупинився:
  ```
  docker ps -a
  ```
  Результат у консолі:
  ```
  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
  1fc8cee64ec2 ubuntu "/bin/sh -c 'while..." 5 minutes ago Exited (137) 5 seconds ago daemon
  c006f1a02edf ubuntu "/bin/echo 'Hello ..." 6 minutes ago Exited (0) 6 minutes ago gifted_nobel
  ```
  Бачимо, що контейнер зупинено. Можемо запустити його ще раз:
  ```
  docker start daemon
  ```
  Переконаємось, що все зроблено правильно:
  ```
  docker ps -a
  ```
  Результат у консолі:
  ```
  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
  1fc8cee64ec2 ubuntu "/bin/sh -c 'while..." 5 minutes ago Up 3 seconds daemon
  c006f1a02edf ubuntu "/bin/echo 'Hello ..." 6 minutes ago Exited (0) 7 minutes ago gifted_nobel
  ```
  Тепер зупинимо його знову та видалимо усі контейнери вручну:
  ```
  docker stop daemon
  docker rm <your first container name>
  docker rm daemon
  ```
  Для цього можна скористатися ще й такою командою:
  ```
  docker rm -f $(docker ps -aq)
  ```
- docker rm — команда для видалення контейнера;
- прапор -f (для rm) зупиняє контейнер, якщо він у процесі виконання (тобто примусове видалення);
- прапор -q (для ps) потрібен, щоб надрукувати лише ідентифікатори контейнерів.

  Приклад 2: Змінні середовища та volume

  Від цього прикладу і далі вам знадобляться деякі додаткові файли з GitHub-репозиторію. Ви можете клонувати його або просто завантажити файли для прикладу тут.

  Час переходити до створення більш корисного контейнера, на зразок **Nginx**.

  Перейдіть у директорію examples/nginx:
  ```
  docker run -d --name "test-nginx" -p 8080:80 -v $(pwd):/usr/share/nginx/html:ro nginx:latest
  ```
  Застереження: команда може видатись громіздкою, але це лише приклад для пояснення volume та змінних середовища. У 99% реальних випадків ви не будете запускати контейнери Docker вручну, а використаєте сервіси оркестровки (оглянемо docker-compose у прикладі №4) або напишете свій власний скрипт для цього.

  Результат у консолі:
  ```
  Unable to find image 'nginx:latest' locally
  latest: Pulling from library/nginx
  683abbb4ea60: Pull complete
  a470862432e2: Pull complete
  977375e58a31: Pull complete
  Digest: sha256:a65beb8c90a08b22a9ff6a219c2f363e16c477b6d610da28fe9cba37c2c3a2ac
  Status: Downloaded newer image for nginx:latest
  afa095a8b81960241ee92ecb9aa689f78d201cff2469895674cec2c2acdcc61c
  ```
- -p прапор — перетворення порту HOST PORT:CONTAINER PORT.
- -v — точка монтування HOST DIRECTORY:CONTAINER DIRECTORY для volume.

Важливо: команда run приймає лише абсолютні шляхи. У нашому прикладі, ми використовували $(pwd) для встановлення абсолютного шляху поточної директорії. Тепер перейдіть за цим url у браузері: 127.0.0.1:8080.

Ми можемо спробувати змінити /example/nginx/index.html (який встановлено як volume для директорії /usr/share/nginx/html всередині контейнера) та перезавантажити сторінку.

Отримуємо інформацію про контейнер test-nginx:

```
docker inspect test-nginx
```

Така команда дозволяє отримати широку системну інформацію про встановлення Docker: версію ядра, кількість контейнерів та образів, відкриті порти, встановленні volume тощо.

Приклад 3: Створення першого Dockerfile

Для створення образу Docker вам потрібно створити файл з назвою Dockerfile. Це звичайний текстовий файл з інструкціями та аргументами. Перелічимо деякі команди з нашого прикладу:

- **FROM** — обирає базовий образ;
- **RUN** — виконує команду у контейнері;
- **ENV** — встановлює змінні середовища;
- **WORKDIR** — встановлює робочу директорію;
- **VOLUME** — встановлює точку монтування для volume;
- **CMD** — встановлює виконуваний файл для контейнера.

Більш детально про Dockerfile за посиланням.

Створимо образ, який отримуватиме вміст веб-сайту з curl та зберігатиме його у текстовий файл. Нам необхідно передати url веб-сайту через змінну середовища SITE_URL. Файл буде розміщено у директорії, змонтованій як volume.

Розмістіть назву **Dockerfile** у директорії examples/curl з таким вмістом:

```
FROM ubuntu:latest
RUN apt-get update
    && apt-get install --no-install-recommends --no-install-suggests -y curl
    && rm -rf /var/lib/apt/lists/*
ENV SITE_URL http://example.com/
WORKDIR /data
VOLUME /data
CMD sh -c "curl -Lk $SITE_URL > /data/results"
```

Dockerfile готовий. Можемо перейти до збірки реального образу.

Перейдіть до директорії examples/curl та виконайте таку команду для збірки:

```
docker build . -t test-curl
```

Результат у консолі:

```
Sending build context to Docker daemon  3.584kB
Step 1/6 : FROM ubuntu:latest
 ---> 113a43faa138
Step 2/6 : RUN apt-get update     && apt-get install --no-install-recommends --no-install-suggests -y
curl     && rm -rf /var/lib/apt/lists/*
 ---> Running in ccc047efe3c7
Get:1 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]
Get:2 http://security.ubuntu.com/ubuntu bionic-security InRelease [83.2 kB]
...
Removing intermediate container ccc047efe3c7
 ---> 8d10d8dd4e2d
Step 3/6 : ENV SITE_URL http://example.com/
 ---> Running in 7688364ef33f
Removing intermediate container 7688364ef33f
```

```
 ---> c71f04bdf39d
Step 4/6 : WORKDIR /data
Removing intermediate container 96b1b6817779
 ---> 1ee38cca19a5
Step 5/6 : VOLUME /data
 ---> Running in ce2c3f68dbbb
Removing intermediate container ce2c3f68dbbb
 ---> f499e78756be
Step 6/6 : CMD sh -c "curl -Lk $SITE_URL > /data/results"
 ---> Running in 834589c1ac03
Removing intermediate container 834589c1ac03
 ---> 4b79e12b5c1d
Successfully built 4b79e12b5c1d
Successfully tagged test-curl:latest
```

- команда docker build проводить збірку нового образу локально;
- прапор -t задає тег назви для образу.

Новий образ готовий і його можна побачити серед переліку вже наявних такою командою:

```
docker images
```

Результат у консолі:

```
REPOSITORY  TAG     IMAGE ID      CREATED         SIZE
test-curl   latest  5ebb2a65d771  37 minutes ago  180 MB
nginx       latest  6b914bbcb89e  7 days ago      182 MB
ubuntu      latest  0ef2e08ed3fa  8 days ago      130 MB
```

Ми можемо створити та запустити контейнер з образу. Спробуємо спочатку з параметрами за замовчуванням:

```
docker run --rm -v $(pwd)/vol:/data/:rw test-curl
```

Щоб побачити збережені до файлу результати:

```
cat ./vol/results
```

Протестуємо на прикладі facebook.com.

```
docker run --rm -e SITE_URL=https://facebook.com/ -v $(pwd)/vol:/data/:rw test-curl
```

Для результатів з файлу та сама команда:

```
cat ./vol/results
```

Найкращі практики створення образів

- Беріть лише необхідний **контекст** — використовуйте файл .dockerignore (як .gitignore у git).
- Уникайте встановлення **непотрібних пакетів** — це зайве місце на диску;
- **Використовуйте кеш**. Додайте контекст, який вносить багато змін (наприклад, сирцевий код вашого проекту) у кінець Dockerfile — так він використовуватиме кеш Docker ефективніше.
- **Будьте пильні з volume**. Необхідно пам'ятати, які дані є в цих сховищах, адже volume не знищується разом з контейнером, тому наступний контейнер використовуватиме попередньо збережені дані.
- Використовуйте **змінні середовища** (у RUN, EXPOSE, VOLUME). Так ваш Dockerfile буде гнучкішим.

Образи Alpine

Більшість Docker-образів створені на Alpine Linux — легкому дистрибутиві, який дозволяє зменшити загальний розмір образів.

Рекомендую використовувати образи на основі Alpine для сторонніх сервісів, на зразок Redis, Postgres тощо. Для образів вашого застосунку, використовуйте образи на основі buildpack — їх легко налагоджувати всередині контейнера. Ви також отримаєте багато попередньо встановлених системних вимог.

Ви самостійно вирішуєте, який базовий образ використовувати. Однак максимум користі можна отримати, застосовуючи один базовий образ — так кеш використовується найбільш ефективно.

Приклад 4: З'єднання між контейнерами

Docker compose — CLI-утиліта для з'єднання контейнерів.

Для її встановлення використовуємо pip:

```
sudo pip install docker-compose
```

У цьому прикладі з'єднаємо контейнери Python та Redis:

```
version: '3.6'
services:
  app:
    build:
      context: ./app
    depends_on:
      - redis
    environment:
      - REDIS_HOST=redis
    ports:
      - "5000:5000"
  redis:
    image: redis:3.2-alpine
    volumes:
      - redis_data:/data
volumes:
  redis_data:
```

Перейдіть до examples/compose та виконайте таку команду:

```
docker-compose up
```

У консолі побачимо:

```
Building app
Step 1/9 : FROM python:3.6.3
3.6.3: Pulling from library/python
f49cf87b52c1: Pull complete
7b491c575b06: Pull complete
b313b08bab3b: Pull complete
51d6678c3f0e: Pull complete
09f35bd58db2: Pull complete
1bda3d37eead: Pull complete
9f47966d4de2: Pull complete
9fd775bfe531: Pull complete
Digest: sha256:cdef88d8625cf50ca705b7abfe99e8eb33b889652a9389b017eb46a6d2f1aaf3
Status: Downloaded newer image for python:3.6.3
 ---> a8f7167de312
Step 2/9 : ENV BIND_PORT 5000
 ---> Running in 3b6fe5ca226d
Removing intermediate container 3b6fe5ca226d
 ---> 0b84340fa920
Step 3/9 : ENV REDIS_HOST localhost
 ---> Running in a4f9a1d6f541
Removing intermediate container a4f9a1d6f541
```

```
 ---> ebe63bf5959e
Step 4/9 : ENV REDIS_PORT 6379
 ---> Running in fd06aa65fd33
Removing intermediate container fd06aa65fd33
 ---> 2a581c31ff4f
Step 5/9 : COPY ./requirements.txt /requirements.txt
 ---> 671093a12829
Step 6/9 : RUN pip install -r /requirements.txt
 ---> Running in b8ea53bc6ba6
Collecting flask==1.0.2 (from -r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dacb4696386634607af824e
a997202cd0edb4b/Flask-1.0.2-py2.py3-none-any.whl (91kB)
Collecting redis==2.10.6 (from -r /requirements.txt (line 2))
  Downloading
https://files.pythonhosted.org/packages/3b/f6/7a76333cf0b9251ecf49efff635015171843d9b977e4ffc
f59f9c4428052/redis-2.10.6-py2.py3-none-any.whl (64kB)
Collecting click>=5.1 (from flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/34/c1/8806f99713ddb993c5366c362b2f908f18269f8d792aff
1abfd700775a77/click-6.7-py2.py3-none-any.whl (71kB)
Collecting Jinja2>=2.10 (from flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/7f/ff/ae64bacdfc95f27a016a7bed8e8686763ba4d277a78ca7
6f32659220a731/Jinja2-2.10-py2.py3-none-any.whl (126kB)
Collecting itsdangerous>=0.24 (from flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/dc/b4/a60bcdba945c00f6d608d8975131ab3f25b22f2bcfe1da
b221165194b2d4/itsdangerous-0.24.tar.gz (46kB)
Collecting Werkzeug>=0.14 (from flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/20/c4/12e3e56473e52375aa29c4764e70d1b8f3efa6682bef8
d0aae04fe335243/Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10->flask==1.0.2->-r /requirements.txt (line 1))
  Downloading
https://files.pythonhosted.org/packages/4d/de/32d741db316d8fdb7680822dd37001ef7a448255de96
99ab4bfcbdf4172b/MarkupSafe-1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous: started
  Running setup.py bdist_wheel for itsdangerous: finished with status 'done'
  Stored in directory:
/root/.cache/pip/wheels/2c/4a/61/5599631c1554768c6290b08c02c72d7317910374ca602ff1e5
  Running setup.py bdist_wheel for MarkupSafe: started
  Running setup.py bdist_wheel for MarkupSafe: finished with status 'done'
  Stored in directory:
/root/.cache/pip/wheels/33/56/20/ebe49a5c612fffe1c5a632146b16596f9e64676768661e4e46
Successfully built itsdangerous MarkupSafe
Installing collected packages: click, MarkupSafe, Jinja2, itsdangerous, Werkzeug, flask, redis
```

```
Successfully installed Jinja2-2.10 MarkupSafe-1.0 Werkzeug-0.14.1 click-6.7 flask-1.0.2
itsdangerous-0.24 redis-2.10.6
You are using pip version 9.0.1, however version 10.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Removing intermediate container b8ea53bc6ba6
 ---> 3117d3927951
Step 7/9 : COPY ./app.py /app.py
 ---> 84a82fa91773
Step 8/9 : EXPOSE $BIND_PORT
 ---> Running in 8e259617b7b5
Removing intermediate container 8e259617b7b5
 ---> 55f447f498dd
Step 9/9 : CMD [ "python", "/app.py" ]
 ---> Running in 2ade293ecb25
Removing intermediate container 2ade293ecb25
 ---> b85b4246e9f8

Successfully built b85b4246e9f8
Successfully tagged compose_app:latest
WARNING: Image for service app was built because it did not already exist. To rebuild this image
you must use `docker-compose build` or `docker-compose up --build`.
Creating compose_redis_1 ... done
Creating compose_app_1   ... done
Attaching to compose_redis_1, compose_app_1
redis_1  | 1:C 08 Jul 18:12:21.851 # Warning: no config file specified, using the default config. In
order to specify a config file use redis-server /path/to/redis.conf
redis_1  |                 _._
redis_1  |            _.-``__ ''-._
redis_1  |       _.-``    `.  `_.  ''-._           Redis 3.2.12 (00000000/0) 64 bit
redis_1  |   .-`` .-```.  ```/    _.,_ ''-._
redis_1  |  (    '      ,       .-`  | `,    )     Running in standalone mode
redis_1  |  |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
redis_1  |  |    `-._   `._    /     _.-'    |     PID: 1
redis_1  |   `-._    `-._  `-./  _.-'    _.-'
redis_1  |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1  |  |    `-._`-._        _.-'_.-'    |           http://redis.io
redis_1  |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1  |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1  |  |    `-._`-._        _.-'_.-'    |
redis_1  |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1  |       `-._    `-.__.-'    _.-'
redis_1  |           `-._        _.-'
redis_1  |               `-.__.-'
redis_1  |
redis_1  | 1:M 08 Jul 18:12:21.852 # WARNING: The TCP backlog setting of 511 cannot be enforced
because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1  | 1:M 08 Jul 18:12:21.852 # Server started, Redis version 3.2.12
redis_1  | 1:M 08 Jul 18:12:21.852 # WARNING overcommit_memory is set to 0! Background save
may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to
```

```
/etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
redis_1  | 1:M 08 Jul 18:12:21.852 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
redis_1  | 1:M 08 Jul 18:12:21.852 * The server is now ready to accept connections on port 6379
app_1    |  * Serving Flask app "app" (lazy loading)
app_1    |  * Environment: production
app_1    |    WARNING: Do not use the development server in a production environment.
app_1    |    Use a production WSGI server instead.
app_1    |  * Debug mode: on
app_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
app_1    |  * Restarting with stat
app_1    |  * Debugger is active!
app_1    |  * Debugger PIN: 170-528-240
```

Тут ми інкрементуємо лічильник переглядів у Redis. Щоб протестувати, перейдіть за адресою `127.0.0.1:5000`.

Використання docker-compose — тема для окремої статті. Для початку ви можете потренуватися з образами з Docker Hub. Якщо хочете створити власні образи, послуговуйтесь найкращими практиками, вказаними вище. Єдине, що можна додати з приводу використання docker-compose: ви завжди **повинні давати явні назви своїм volume** у `docker-compose.yml` (якщо, звичайно, образ має volume). Таке просте правило убезпечить вас від проблем з перевіркою volume у майбутньому.

```
version: '3.6'
services:
  ...
  redis:
    image: redis:3.2-alpine
    volumes:
      - redis_data:/data
volumes:
  redis_data:
```

У нашому прикладі `redis_data` буде назвою всередині файлу `docker-compose`. Реальна назва volume починається з назви проекту.

Для перегляду volume запустіть:

```
docker volume ls
```

Результат:

```
DRIVER              VOLUME NAME
local               apptest_redis_data
```

Якщо б ми явно не вказали назву volume, побачили б його UUID. Погляньте на приклад з мого комп’ютера:

```
DRIVER              VOLUME NAME
local               ec1a5ac0a2106963c2129151b27cb032ea5bb7c4bd6fe94d9dd22d3e72b2a41b
local               f3a664ce353ba24dd43d8f104871594de6024ed847054422bbdd362c5033fc4c
local               f81a397776458e62022610f38a1bfe50dd388628e2badc3d3a2553bb08a5467f
local               f84228acbf9c5c06da7be2197db37f2e3da34b7e8277942b10900f77f78c9e64
local               f9958475a011982b4dc8d8d8209899474ea4ec2c27f68d1a430c94bcc1eb0227
```

```
local     ff14e0e20d70aa57e62db0b813db08577703ff1405b2a90ec88f48eb4cdc7c19
local     polls_pg_data
local     polls_public_files
local     polls_redis_data
local     projectdev_pg_data
local     projectdev_redis_data
```

Деякі обмеження та вимоги

Залежно від архітектури вашої системи, обмеження і вимоги можуть відрізнятися. Ви можете ігнорувати їх або знайти обхідні шляхи, але так ви не отримаєте усі переваги від використання Docker. Тож скористайтеся кількома порадами:

- **1 застосунок = 1 контейнер**;
- НЕ запускайте процес як **фоновий** (не використовуйте `systemd`, `upstart` та подібні інструменти);
- **зберігайте дані за межами контейнерів** — у volume;
- **не використовуйте SSH** (краще послуговуйтесь командою `docker exec`);
- **не налаштовуйте конфіги всередині контейнера вручну**.