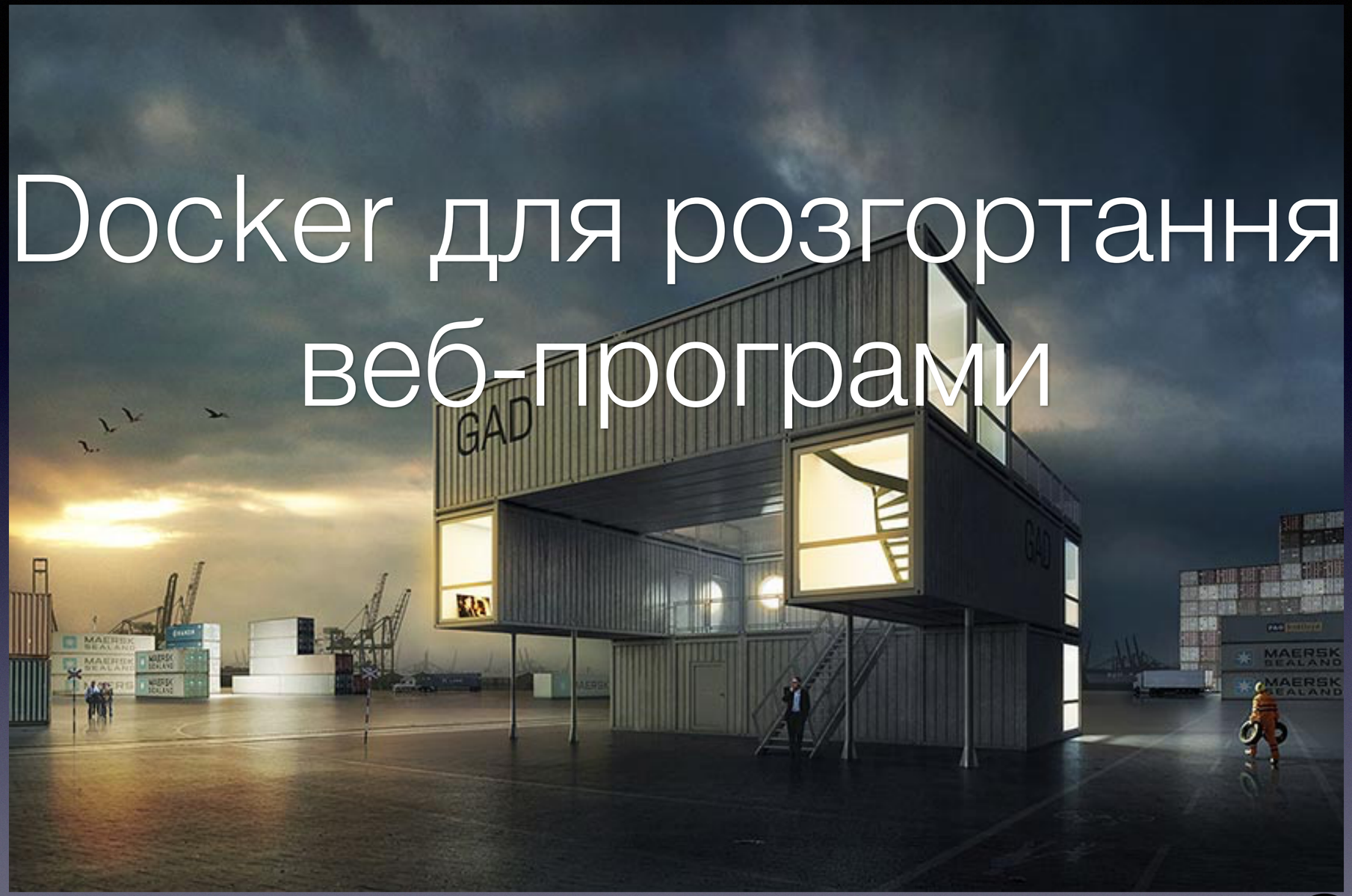
Docker для розгортання веб-програми



GENIUS space

Технології контейнеризації додатків знайшли широке застосування у сферах розробки ПЗ та аналізу даних. Ці технології допомагають зробити додатки безпечнішими, полегшують їхнє розгортання та покращують можливості з їхнього масштабування. Зростання і розвиток технологій контейнеризації можна вважати одним із найважливіших трендів сучасності.

Docker - це платформа, яка призначена для розробки, розгортання та запуску застосунків у контейнерах. Слово "Docker" останнім часом стало чимось на зразок синоніма слова "контейнеризація". І якщо ви ще не користуєтеся Docker, але водночас працюєте або збираєтеся працювати у сферах розроблення застосунків чи аналізу даних, то Docker - це те, з чим ви неодмінно зустрінетеся в майбутньому.

GENIUS space

Контейнер

- У ньому можна щось зберігати. Щось може перебувати або в контейнері,
або за його межами.

- Його можна переносити. Контейнер Docker можна використовувати на локальному комп'ютері, на комп'ютері колеги, на сервері постачальника хмарних послуг (на кшталт AWS). Це споріднює контейнери Docker зі звичайними контейнерами, в яких, наприклад, перевозять різні милі серцю дрібнички під час переїзду в новий будинок.

- У контейнер зручно щось класти і зручно щось із нього виймати. У звичайного контейнера є кришка на засувках, яку треба зняти для того, щоб щось покласти в контейнер або щось із нього вийняти. У контейнерів Docker є щось подібне, що представляє їхній інтерфейс, тобто - механізми, які дають їм змогу взаємодіяти із зовнішнім світом. Наприклад, у контейнера є порти, які можна відкривати для того, щоб до додатка, що працює в контейнері, можна було б звертатися з браузера. Працювати з контейнером можна і засобами командного рядка.

- Якщо вам потрібен контейнер, його можна замовити в інтернет-магазині. Порожній контейнер можна купити, наприклад, на сайті Amazon. У цей магазин контейнери потрапляють від виробників, які роблять їх у величезних кількостях, використовуючи прес-форми. У випадку з контейнерами Docker те, що можна порівняти з прес-формою, а саме - образ контейнера, зберігається в спеціальному репозиторії. Якщо вам потрібен якийсь контейнер, ви можете завантажити з репозиторію відповідний образ, і, використовуючи його, цей контейнер створити.

Програмне забезпечення

- Контейнери Docker можна порівнювати не тільки зі звичайними контейнерами або з живими організмами. Їх можна порівняти і з програмами.
Зрештою, контейнери - це програми. І, на фундаментальному рівні, контейнер являє собою набір інструкцій, який виконується на якомусь процесорі, обробляючи якісь дані.

- Контейнер - це програма

- Під час виконання контейнера Docker всередині нього зазвичай виконується якась програма. Вона виконує в контейнері якісь дії, тобто - робить щось корисне.

- Наприклад, код, який працює в контейнері Docker, можливо, відправив на ваш комп'ютер той текст, який ви зараз читаєте. Цілком можливо й те, що саме код, який виконується в контейнері Docker, приймає голосові команди, які ви даєте Amazon Alexa, і перетворює їх на інструкції для ще якихось програм, що працюють в інших контейнерах.

- Завдяки використанню Docker можна, на одному і тому ж комп'ютері, одночасно запускати безліч контейнерів. І, як і будь-які інші програми, контейнери Docker можна запускати, зупиняти, видаляти. Можна досліджувати їхній вміст і створювати їх.

Файл Dockerfile

- Файл Dockerfile містить набір інструкцій, дотримуючись яких Docker збиратиме образ контейнера. Цей файл містить опис базового образу, який буде являти собою вихідний шар образу. Серед популярних офіційних базових образів можна відзначити python, ubuntu, alpine.

- В образ контейнера, поверх базового образу, можна додавати додаткові шари. Робиться це відповідно до інструкцій з Dockerfile. Наприклад, якщо Dockerfile описує образ, який планують використовувати для розв'язання задач машинного навчання, то в ньому можуть бути інструкції для включення в проміжний шар такого образу бібліотек NumPy, Pandas і Scikit-learn.

- І, нарешті, в образі може міститися, поверх усіх інших, ще один тонкий шар, дані, що зберігаються, піддаються зміні. Це - невеликий за обсягом шар, що містить програму, яку планується запускати в контейнері.

Контейнер Docker

- Для того щоб запустити контейнер, нам потрібен, по-перше, образ контейнера, по-друге - середовище, в якому встановлено Docker, здатне зрозуміти команду виду **docker run image_name**. Ця команда створює контейнер з образу і запускає його.

Репозиторій контейнерів

- Якщо ви хочете дати можливість іншим людям створювати контейнери на основі вашого образу, ви можете відправити цей образ у хмарне сховище. Найбільшим подібним сховищем є репозиторій Docker Hub. Він використовується під час роботи з Docker за замовчуванням.

Платформа Docker

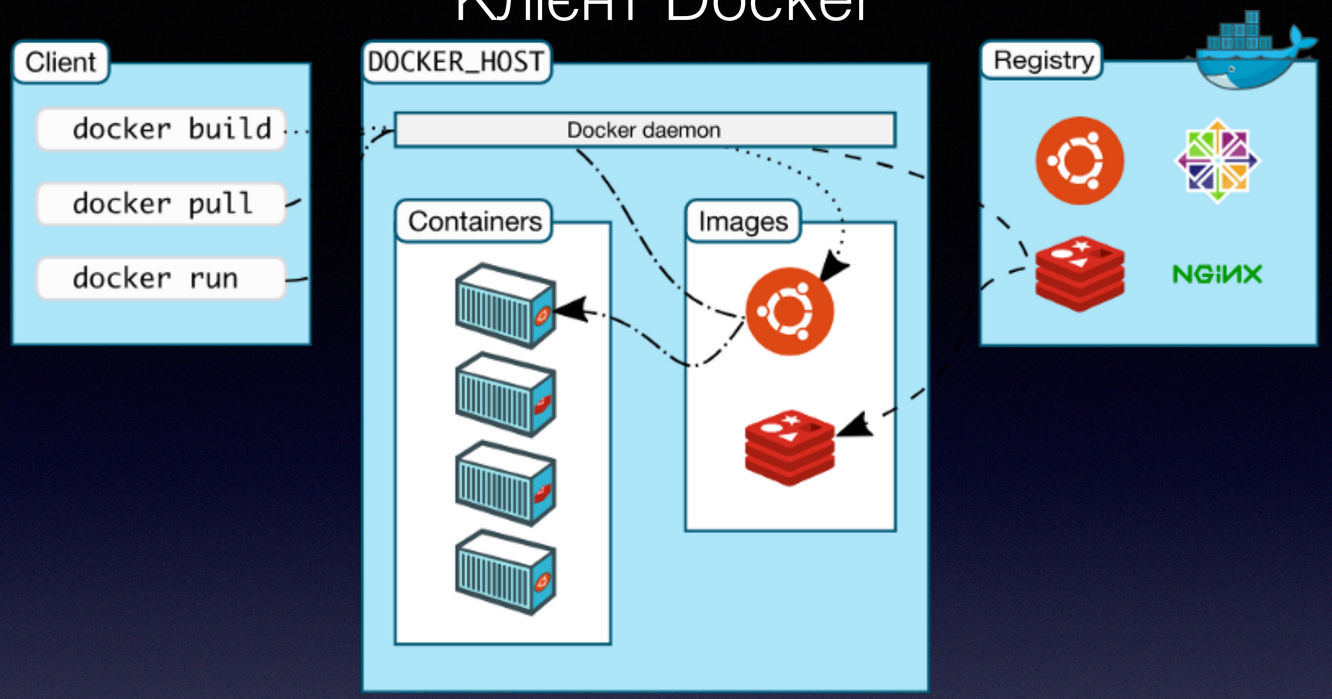
- Платформа Docker (Docker Platform) - це програма, яка дає нам змогу упаковувати додатки в контейнери та запускати їх на серверах. Платформа Docker дає змогу поміщати в контейнери код і його залежності. Як результат, системи, засновані на контейнерах, легко масштабувати, оскільки контейнери можна переносити і відтворювати.

Движок Docker

- Движок Docker (Docker Engine) - це клієнт-серверний застосунок. Компанія Docker розділила рушій Docker на два продукти. Docker Community Edition (CE) - це безкоштовне ПЗ, багато в чому засноване на опенсорсних інструментах.

- Ймовірно, ви будете користуватися саме цією версією Docker. Docker Enterprise - це платна версія системи, що дає користувачам додаткові можливості в галузі підтримки систем, управління ними та безпеки. Платна версія Docker дає компанії кошти, необхідні для її існування.

Клієнт Docker



- Клієнт Docker (Docker Client) - це основний засіб, який використовують для взаємодії з Docker. Так, під час роботи з інтерфейсом командного рядка Docker (Docker Command Line Interface, CLI), у термінал вводять команди, що починаються з ключового слова docker, звертаючись до клієнта. Потім клієнт використовує API Docker для надсилання команд демону Docker.

Демон Docker

- Демон Docker (Docker Daemon) - це сервер Docker, який очікує запитів до API Docker. Демон Docker керує образами, контейнерами, мережами та томами.

Тома Docker

- Томи Docker (Docker Volumes) являють собою найкращий механізм постійного зберігання даних, які споживаються або виробляються додатками.

Реєстр Docker

- Реєстр Docker (Docker Registry) являє собою віддалену платформу, яка використовується для зберігання образів Docker. Під час роботи з Docker образи надсилають до реєстру і завантажують із нього. Подібний реєстр може бути організований тим, хто користується Docker. Крім того, постачальники хмарних послуг можуть підтримувати і власні реєстри. Наприклад, це стосується AWS і Google Cloud.

Хаб Docker

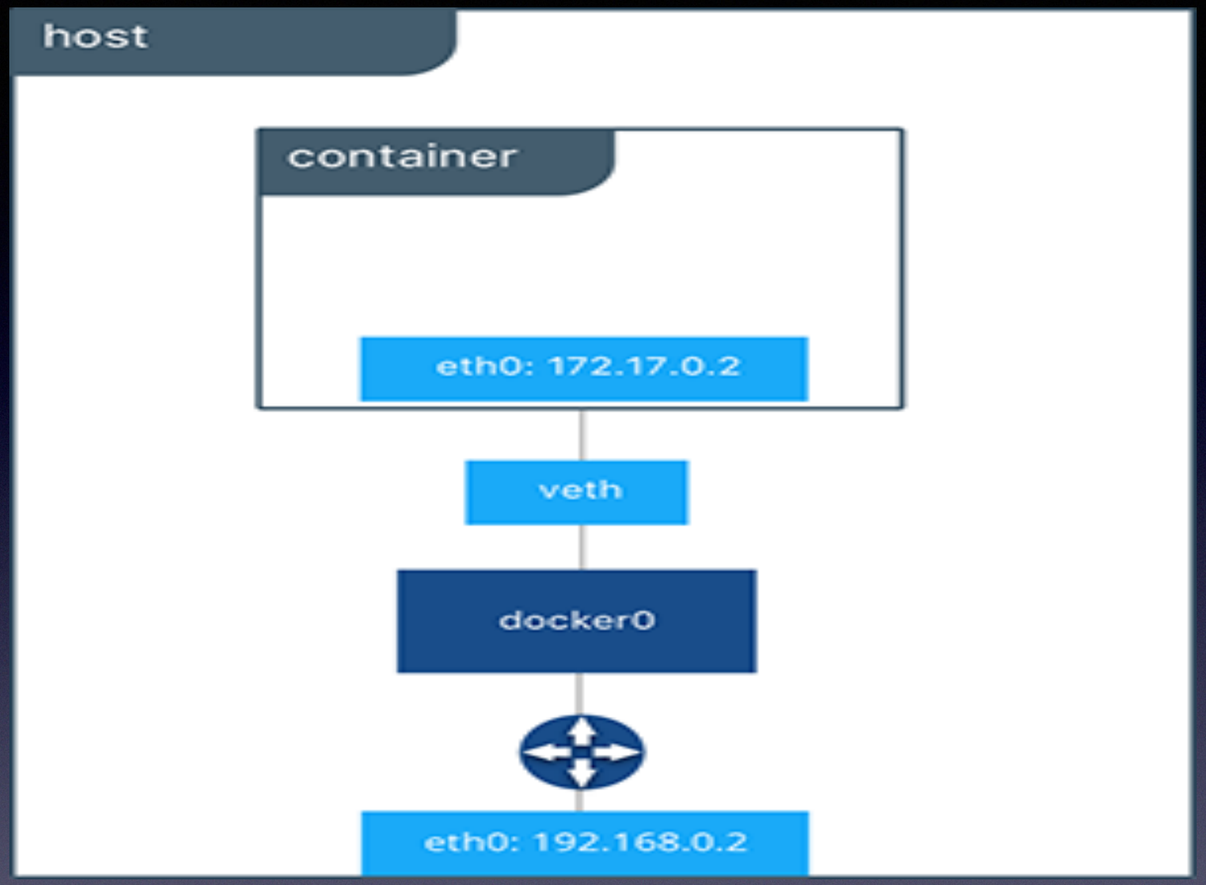
- Хаб Docker (Docker Hub) - це найбільший реєстр образів Docker. Крім того, саме цей реєстр використовується під час роботи з Docker за замовчуванням. Користуватися хабом Docker можна безкоштовно.

Репозиторій Docker

- Репозиторієм Docker (Docker Repository) називають набір образів Docker, що володіють однаковими іменами і різними тегами. Теги - це ідентифікатори образів.

- Зазвичай у репозиторіях зберігаються різні версії одних і тих самих образів. Наприклад, Python - це ім'я найпопулярнішого офіційного репозиторію Docker на хабі Docker. А ось Python:3.7-slim - це версія образу з тегом 3.7-slim у репозиторії Python. До реєстру можна відправити як цілий репозиторій, так і окремий образ.

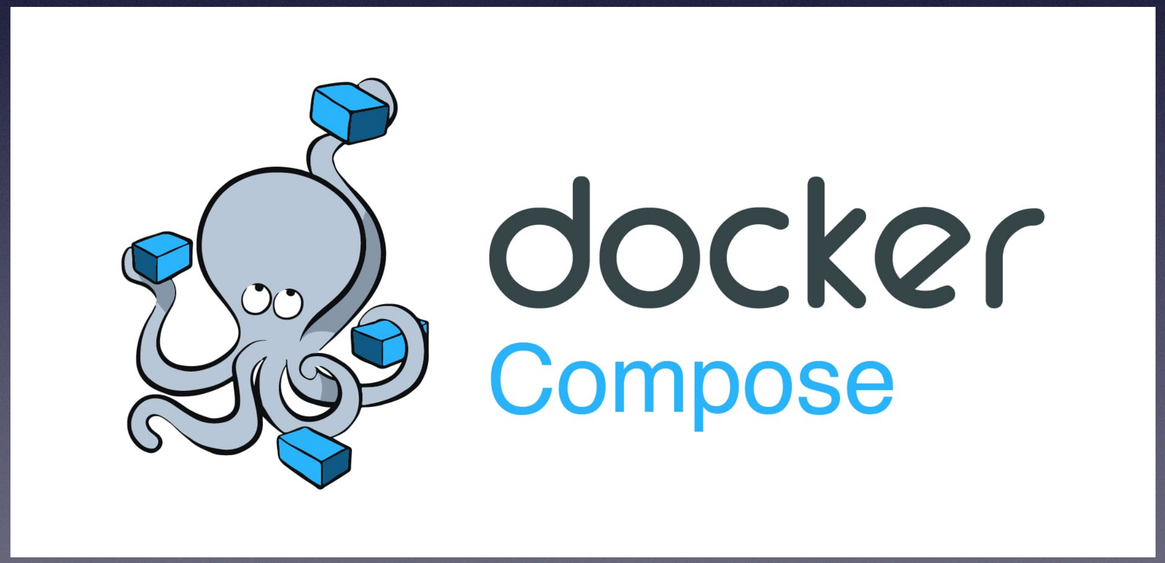
Мережа Docker



- Мережеві механізми Docker (Docker Networking) дають змогу організовувати зв'язок між контейнерами Docker. З'єднані за допомогою мережі контейнери можуть виконуватися на одному і тому ж хості або на різних хостах.

Docker Compose

Docker Compose - це інструмент, який спрощує розгортання додатків, для роботи яких потрібно кілька контейнерів Docker. Docker Compose дає змогу виконувати команди, що описуються у файлі docker-compose.yml. Ці команди можна виконувати стільки разів, скільки буде потрібно. Інтерфейс командного рядка Docker Compose спрощує взаємодію з багатоконтейнерними додатками. Цей інструмент встановлюється під час встановлення Docker.



Файли Dockerfile

- У файлах Dockerfile містяться інструкції зі створення образу. З них, набраних великими літерами, починаються рядки цього файлу. Після інструкцій йдуть їхні аргументи. Інструкції, під час складання образу, обробляються зверху вниз. Ось як це виглядає:

FROM ubuntu:18.04

COPY . /app

- Шари в підсумковому образі створюють тільки інструкції FROM, RUN, COPY, і ADD. Інші інструкції щось налаштовують, описують метадані, або повідомляють Docker про те, що під час виконання контейнера потрібно щось зробити, наприклад, відкрити якийсь порт або виконати якусь команду.

- Тут ми виходимо з припущення, відповідно до якого використовується образ Docker, заснований на Unix-подібній ОС. Звичайно, тут можна скористатися і образом, заснованим на Windows, але використання Windows - це менш поширена практика, працювати з такими образами складніше. У результаті, якщо у вас є така можливість, користуйтеся Unix.

Ось декілька інструкцій Dockerfile

- **FROM** - задає базовий (батьківський) образ.
- **LABEL** - описує метадані. Наприклад - відомості про те, хто створив і підтримує образ.
- **ENV** - встановлює постійні змінні середовища.
- **RUN** - виконує команду і створює шар образу. Використовується для встановлення в контейнер пакетів.
- **COPY** - копіює в контейнер файли та папки.
- **ADD** - копіює файли і папки в контейнер, може розпаковувати локальні .tar-файли.
- **CMD** - описує команду з аргументами, яку потрібно виконати, коли контейнер буде запущено. Аргументи можуть бути перевизначені під час запуску контейнера. У файлі може бути присутня лише одна інструкція CMD.
- **WORKDIR** - задає робочу директорію для наступної інструкції.
- **ARG** - задає змінні для передачі Docker під час складання образу.
- **ENTRYPOINT** - надає команду з аргументами для виклику під час виконання контейнера. Аргументи не перевизначаються.
- **EXPOSE** - вказує на необхідність відкрити порт.
- **VOLUME** - створює точку монтування для роботи з постійним сховищем.

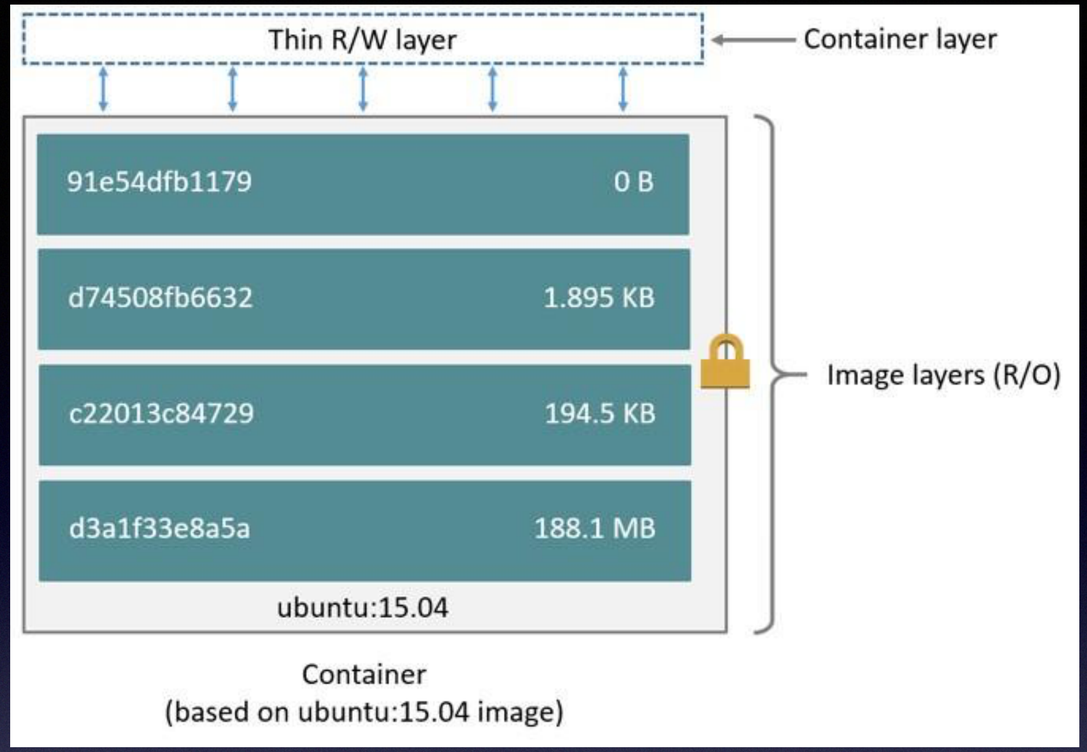
Простий Dockerfile

- Dockerfile може бути надзвичайно простим і коротким. Наприклад - таким:

FROM ubuntu:18.04

Інструкція FROM

- Файл Dockerfile має починатися з інструкції FROM, або з інструкції ARG, за якою йде інструкція FROM.
- Ключове слово FROM повідомляє Docker про те, щоб під час складання образу використовувався б базовий образ, який відповідає наданому імені та тегу. Базовий образ, крім того, ще називають батьківським образом.
- У цьому прикладі базовий образ зберігається в репозиторії ubuntu. Ubuntu - це назва офіційного репозиторію Docker, що надає базову версію популярної ОС сімейства Linux, яка називається Ubuntu.
- Зверніть увагу на те, що Dockerfile, який ми розглядаємо, містить тег 18.04, який уточнює те, який саме базовий образ нам потрібен. Саме цей образ і буде завантажено під час складання нашого образу. Якщо тег в інструкцію не включено, тоді Docker виходить із припущення про те, що потрібен найсвіжіший образ із репозиторію. Для того щоб ясніше висловити свої наміри, автору Dockerfile рекомендується вказувати те, який саме образ йому потрібен.
- Коли вищеописаний Dockerfile використовується на локальній машині для складання образу вперше, Docker завантажить шари, які визначаються образом ubuntu. Їх можна уявити накладеними один на одного. Кожен наступний шар являє собою файл, що описує відмінності образу порівняно з тим його станом, у якому він був після додавання в нього попереднього шару.
- Під час створення контейнера шар, у який можна вносити зміни, додається поверх усіх інших шарів. Дані, що знаходяться в інших шарах, можна тільки читати.



- Docker, заради ефективності, використовує стратегію копіювання під час запису. Якщо шар в образі існує на попередньому рівні і якомусь шару потрібно зробити читання даних з нього, Docker використовує наявний файл. При цьому нічого завантажувати не потрібно.

- Коли образ виконується, якщо шар потрібно модифікувати засобами контейнера, то відповідний файл копіюється в самий верхній, змінюваний шар.

- Продовжимо розгляд інструкцій, які використовуються в Dockerfile, навівши приклад такого файлу зі складнішою структурою.

Складніший Dockerfile

- Хоча файл Dockerfile, який ми щойно розглянули, вийшов акуратним і зрозумілим, він влаштований надто просто, у ньому використовується лише одна інструкція. Крім того, там немає інструкцій, що викликаються під час виконання контейнера. Поглянемо на ще один файл, який збирає маленький образ. У ньому є механізми, що визначають команди, які викликаються під час виконання контейнера.

```
FROM python:3.7.2-alpine3.8

LABEL maintainer="genius@gmail.com"

ENV ADMIN=“dima"

RUN apk update && apk upgrade && apk add bash

COPY . ./app

ADD https://raw.githubusercontent.com/discdiver/pachy-vid/master/sample_vids/vid1.mp4 \

/my_app_directory

RUN ["mkdir", "/a_directory"]

CMD ["python", “./my_script.py"]
```

Базою цього образу є офіційний образ Python з тегом 3.7.2-alpine3.8. Проаналізувавши цей код, можна побачити, що цей базовий образ містить у собі Linux, Python, і, за великим рахунком, цим його склад і обмежується. Образи ОС Alpine вельми популярні у світі Docker. Річ у тім, що вони вирізняються маленькими розмірами, високою швидкістю роботи і безпекою. Однак образи Alpine не відрізняються широкими можливостями, характерними для звичайних операційних систем.
Тому для того, щоб зібрати на основі такого образу щось корисне, творцеві образу потрібно встановити в нього необхідні йому пакети.

Інструкція LABEL

- Інструкція LABEL (мітка) дозволяє додавати в образ метадані. У випадку з файлом, який ми зараз розглядаємо, вона містить контактні відомості творця образу.
Оголошення міток не уповільнює процес складання образу і не збільшує його розмір.
Вони лише містять у собі корисну інформацію про образ Docker, тому їх рекомендується включати у файл.

Інструкція ENV

- Навколишнє середовище Інструкція ENV дозволяє задавати постійні змінні середовища, які будуть доступні в контейнері під час його виконання.У попередньому прикладі після створення контейнера можна користуватися змінною ADMIN.

- Інструкція ENV добре підходить для завдання констант.Якщо ви використовуєте якесь значення в Dockerfile кілька разів, скажімо, під час опису команд, що виконуються в контейнері, і підозрюєте, що, можливо, вам колись доведеться змінити його на інше, його має сенс записати в подібну константу.

- Треба зазначити, що у файлах Dockerfile часто існують різні способи вирішення одних і тих самих завдань. Що саме використовувати - це питання, на вирішення якого впливає прагнення до дотримання прийнятих у середовищі Docker методів роботи, до забезпечення прозорості рішення і його високої продуктивності. Наприклад, інструкції RUN, CMD і ENTRYPOINT служать різним цілям, але всі вони використовуються для виконання команд.

Інструкція RUN

- Інструкція RUN дозволяє створити шар під час складання образу.

Після її виконання в образ додається новий шар, його стан фіксується. Інструкція RUN часто використовується для встановлення в образи додаткових пакетів. У попередньому прикладі інструкція RUN apk update && apk upgrade повідомляє Docker про те, що системі потрібно оновити пакети з базового образу. Слідом за цими двома командами йде команда && apk add bash, яка вказує на те, що в образ потрібно встановити bash.

- Те, що в командах має вигляд apk - це скорочення від Alpine Linux package manager (менеджер пакетів Alpine Linux). Якщо ви використовуєте базовий образ якоїсь іншої ОС сімейства Linux, тоді вам, наприклад, під час використання Ubuntu, для встановлення пакетів може знадобитися команда виду RUN apt-get. Пізніше ми поговоримо про інші способи встановлення пакетів.

- Інструкція RUN і схожі з нею інструкції - такі, як CMD і ENTRYPOINT, можуть бути використані або в exec-формі, або в shell-формі. Exec-форма використовує синтаксис, що нагадує опис JSON-масиву. Наприклад, це може виглядати так: RUN ["my_executable", "my_first_param1", "my_second_param2"].

- У попередньому прикладі ми використовували shell-форму інструкції RUN у такому вигляді: RUN apk update && apk upgrade && apk add bash.

- Пізніше в нашому Dockerfile використано exec-форму інструкції RUN, у вигляді RUN ["mkdir", "/a_directory"] для створення директорії. При цьому, використовуючи інструкцію в такій формі, потрібно пам'ятати про необхідність оформлення рядків за допомогою подвійних лапок, як це прийнято у форматі JSON.

Інструкція COPY

- Інструкція COPY представлена в нашому файлі так: COPY . ./app.

Вона повідомляє Docker про те, що потрібно взяти файли і папки з локального контексту збірки і додати їх у поточну робочу директорію образу. Якщо цільова директорія не існує, ця інструкція її створить.

Інструкція ADD

- Інструкція ADD дає змогу вирішувати ті самі завдання, що й COPY, але з нею пов'язано ще кілька варіантів використання. Так, за допомогою цієї інструкції можна додавати в контейнер файли, завантажені з віддалених джерел, а також розпаковувати локальні .tar-файли.

- У цьому прикладі інструкцію ADD було використано для копіювання файлу, доступного за URL, у директорію контейнера my_app_directory.Треба зазначити, однак, що документація Docker не рекомендує використання подібних файлів, отриманих за URL, оскільки видалити їх не можна, і оскільки вони збільшують розмір образу.

- Крім того, документація пропонує скрізь, де це можливо, замість інструкції ADD використовувати інструкцію COPY для того, щоб зробити файли Dockerfile зрозумілішими. Вважаю, команді розробників Docker варто було б об'єднати ADD і COPY в одну інструкцію для того, щоб тим, хто створює образи, не доводилося б пам'ятати занадто багато інструкцій.

- Зверніть увагу на те, що інструкція ADD містить символ розриву рядка - \. Такі символи використовуються для поліпшення читабельності довгих команд шляхом розбиття їх на кілька рядків.

Інструкція CMD

- Інструкція CMD надає Docker команду, яку потрібно виконати під час запуску контейнера. Результати виконання цієї команди не додаються в образ під час його складання.

- У нашому прикладі за допомогою цієї команди запускається скрипт my_script.py під час виконання контейнера.

Ось ще дещо, що потрібно знати про інструкцію CMD:

- В одному файлі Dockerfile може бути присутня лише одна інструкція CMD. Якщо у файлі є кілька таких інструкцій, система проігнорує всі, крім останньої.

- Інструкція CMD може мати exec-форму. Якщо в цю інструкцію не входить згадка виконуваного файлу, тоді у файлі має бути присутня інструкція ENTRYPOINT. У такому разі обидві ці інструкції мають бути представлені у форматі JSON.

- Аргументи командного рядка, що передаються docker run, перевизначають аргументи, надані інструкції CMD в Dockerfile.

Розглянемо ще один файл Dockerfile, у якому будуть використані деякі нові команди.

```
FROM python:3.7.2-alpine3.8

LABEL maintainer="genius@gmail.com"

# Встановлюємо залежності

RUN apk add --update git

# Задаємо поточну робочу директорію

WORKDIR /usr/src/my_app_directory

# Копіюємо код із локального контексту в робочу директорію образу

COPY . .

# Задаємо значення за замовчуванням для змінної

ARG my_var=my_default_value

# Налаштовуємо команду, яка має бути запущена в контейнері під час його виконання

ENTRYPOINT ["python", "./app/my_script.py", "my_var"]

# Відкриваємо порти

EXPOSE 8000

# Створюємо том для зберігання даних

VOLUME /my_volume
```

У цьому прикладі, крім іншого, ви можете бачити коментарі, які починаються з символу #.

- Одна з основних дій, що виконуються засобами Dockerfile, - це встановлення пакетів.

Як уже було сказано, існують різні способи встановлення пакетів за допомогою інструкції RUN.

- Пакети в образ Alpine Docker можна встановлювати за допомогою apk. Для цього, як ми вже говорили, застосовується команда виду RUN apk update && apk upgrade && apk add bash.

- Крім того, пакети Python в образ можна встановлювати за допомогою pip, wheel і conda. Якщо йдеться не про Python, а про інші мови програмування, то під час підготовки відповідних образів можуть використовуватися й інші менеджери пакетів.

- При цьому для того, щоб установлення було б можливим, нижчий шар повинен надати шару, в який виконується встановлення пакетів, відповідний менеджер пакетів. Тому якщо ви зіткнулися з проблемами під час інсталяції пакетів, переконайтеся в тому, що менеджер пакетів встановлений до того, як ви спробуєте ним скористатися.

- Наприклад, інструкцію RUN у Dockerfile можна використовувати для встановлення списку пакетів за допомогою pip. Якщо ви так робите - об'єднайте всі команди в одну інструкцію і розділіть її символами розриву рядка за допомогою символу \. Завдяки такому підходу файли будуть виглядати акуратно і це призведе до додавання в образ меншої кількості шарів, ніж було б додано за використання декількох інструкцій RUN.

- Крім того, для встановлення кількох пакетів можна вчинити і по-іншому. Їх можна перерахувати у файлі та передати менеджеру пакетів цей файл за допомогою RUN. Зазвичай таким файлам дають ім'я requirements.txt

Інструкція WORKDIR

- Інструкція WORKDIR дозволяє змінити робочу директорію контейнера.

З цією директорією працюють інструкції COPY, ADD, RUN, CMD і ENTRYPOINT, що йдуть за WORKDIR. Ось деякі особливості, що стосуються цієї інструкції:

- Краще встановлювати за допомогою WORKDIR абсолютні шляхи до папок, а не переміщатися по файловій системі за допомогою команд cd у Dockerfile.

- Інструкція WORKDIR автоматично створює директорію в тому випадку, якщо вона не існує.

- Можна використовувати кілька інструкцій WORKDIR. Якщо таким інструкціям надаються відносні шляхи, то кожна з них змінює поточну робочу директорію.

Інструкція ARG

- Інструкція ARG дає змогу задати змінну, значення якої можна передати з командного рядка в образ під час його складання. Значення для змінної за замовчуванням можна представити в Dockerfile. Наприклад: ARG my_var=my_default_value.

- На відміну від ENV-змінних, ARG-змінні недоступні під час виконання контейнера. Однак ARG-змінні можна використовувати для задавання значень за замовчуванням для ENV-змінних з командного рядка в процесі складання образу. А ENV-змінні вже будуть доступні в контейнері під час його виконання.

інструкція ENTRYPOINT

- Пункт переходу в якесь місце

- Інструкція ENTRYPOINT дає змогу задавати команду з аргументами, яка має виконуватися
під час запуску контейнера. Вона схожа на команду CMD, але параметри, що задаються в
ENTRYPOINT, не перезаписуються в тому випадку, якщо контейнер запускають з параметрами командного рядка.

- Натомість аргументи командного рядка, що передаються в конструкції виду docker run my_image_name, додаються до аргументів, що задаються інструкцією ENTRYPOINT. Наприклад, після виконання команди виду docker run my_image bash аргумент bash додасться в кінець списку аргументів, заданих за допомогою ENTRYPOINT. Готуючи Dockerfile, не забудьте про інструкції CMD або ENTRYPOINT.

- У документації до Docker є кілька рекомендацій, що стосуються того, яку інструкцію, CMD або ENTRYPOINT, варто вибрати як інструмент для виконання команд під час запуску контейнера:

- Якщо під час кожного запуску контейнера потрібно виконувати одну й ту саму команду - використовуйте ENTRYPOINT.

- Якщо контейнер буде використовуватися в ролі програми - використовуйте ENTRYPOINT.

- Якщо ви знаєте, що під час запуску контейнера вам знадобиться передавати йому аргументи, які можуть перезаписувати аргументи, зазначені в Dockerfile, використовуйте CMD.

- У нашому прикладі використання інструкції ENTRYPOINT ["python", "my_script.py", "my_var"] призводить до того, що контейнер, під час запуску, запускає Python-скрипт my_script.py з аргументом my_var. Значення, представлене my_var, потім можна використовувати в скрипті за допомогою argparse. Зверніть увагу на те, що в Dockerfile змінній my_var, до її використання, призначено значення за замовчуванням за допомогою ARG. У результаті, якщо під час запуску контейнера йому не передали відповідне значення, буде застосовано значення за замовчуванням.

- Документація Docker рекомендує використовувати exec-форму:

- ENTRYPOINT: ENTRYPOINT ["executable", "param1", "param2"].

Інструкція EXPOSE

- Інструкція EXPOSE вказує на те, які порти планується відкрити для того, щоб через них можна було б зв'язатися з працюючим контейнером. Ця інструкція не відкриває порти. Вона, скоріше, відіграє роль документації до образу, засобом спілкування того, хто збирає образ, і того, хто запускає контейнер.

- Для того щоб відкрити порт (або порти) і налаштувати перенаправлення портів, потрібно виконати команду docker run з ключем -p. Якщо використовувати ключ у вигляді -P (із великою літерою P), то відкрито буде всі порти, зазначені в інструкції EXPOSE.

Інструкція VOLUME

- Інструкція VOLUME дає змогу вказати місце, яке контейнер використовуватиме для постійного зберігання файлів і для роботи з такими файлами.

Команди для керування контейнерами

- Загальна схема команд для управління контейнерами виглядає так:

docker container my_command

- Ось команди, які можуть бути підставлені туди, де ми використовували my_command:
- **create** - створення контейнера з образу.
- **start** - запуск наявного контейнера.
- **run** - створення контейнера і його запуск.
- **ls** - виведення списку працюючих контейнерів.
- **inspect** - виведення докладної інформації про контейнер.
- **logs** - виведення логів.
- **stop** - зупинка працюючого контейнера з відправленням головному процесу контейнера сигналу SIGTERM, і, через деякий час, SIGKILL.
- **kill** - зупинка працюючого контейнера з відправленням головному процесу контейнера сигналу SIGKILL.
- **rm** - видалення зупиненого контейнера.

Команди для керування образами

docker image my_command

Ось деякі з команд цієї групи:

- **build** - збірка образу.
- **push** - надсилання образу у віддалений реєстр.
- **ls** - виведення списку образів.
- **history** - виведення відомостей про шари образу.
- **inspect** - виведення докладної інформації про образ, зокрема - відомостей про шари.
- **rm** - видалення образу.

Різні команди

- **docker version** - виведення відомостей про версії клієнта і сервера Docker.
- **docker login** - вхід до реєстру Docker.
- **docker system prune** - видалення невикористовуваних контейнерів, мереж і образів, яким не призначено ім'я і тег.

Початок існування контейнера

- На початковому етапі роботи з контейнерами використовуються команди create, start і run.

Вони застосовуються, відповідно, для створення контейнера, для його запуску, і для його створення та запуску.

- Ось команда для створення контейнера з образу:

docker container create my_repo/my_image:my_tag

- У наступних прикладах конструкція **my_repo/my_image:my_tag** буде скорочена до my_image.

- Команда create приймає безліч прапорів. Наприклад, її можна записати в такому вигляді:

docker container create -a STDIN my_image

- Прапор **-a** являє собою коротку форму прапора **--attach**. Цей прапор дозволяє підключити контейнер до STDIN, STDOUT або STDERR.

- Після того, як контейнер створено, його можна запустити такою командою:

docker container start my_container

- Зверніть увагу на те, що послатися на контейнер у команді можна або використовуючи його ID, або ім'я.

- Тепер поглянемо на команду, яка дає змогу створити і запустити контейнер:

docker container run my_image

- Ця команда теж здатна приймати безліч аргументів командного рядка.

Розглянемо деякі з них на прикладі такої конструкції:

docker container run -i -t -p 1000:8000 --rm my_image

- Прапор **-i** - це скорочення для **--interactive**. Завдяки цьому прапору потік STDIN підтримується у відкритому стані, навіть якщо контейнер до STDIN не підключений.

- Прапор **-t** - це скорочення для **--tty**. Завдяки цьому прапору виділяється псевдотермінал, який з'єднує термінал, що використовується, з потоками STDIN і STDOUT контейнера.

- Для того щоб отримати можливість взаємодії з контейнером через термінал, потрібно спільно використовувати прапори **-i** і **-t**.

- Прапор **-p** являє собою скорочення для **--port**. Порт - це інтерфейс, завдяки якому контейнер взаємодіє із зовнішнім світом. Конструкція 1000:8000 перенаправляє порт Docker 8000 на порт 1000 комп'ютера, на якому виконується контейнер. Якщо в контейнері працює якийсь застосунок, здатний виводити щось у браузер, то для того, щоб до нього звернутися, у нашому випадку можна перейти в браузері за адресою localhost:1000.

- Прапор **--rm** автоматично видаляє контейнер після того, як його виконання завершиться.

- Розглянемо ще деякі приклади команди run:

docker container run -it my_image my_command

- У подібній конструкції може застосовуватися команда sh, яка створить сесію терміналу в контейнері, з якою можна взаємодіяти через ваш термінал. Під час роботи з образами, заснованими на Alpine, краще орієнтуватися на використання sh, а не bash, тому що в цих образах, за замовчуванням, оболонку bash не встановлено. Для виходу з інтерактивної сесії скористайтеся командою exit.

- Зверніть увагу на те, що тут ми скомбінували прапори **-i** і **-t** в **-it**.

- Ось ще один приклад роботи з командою run:

- **docker container run -d my_image**

- Прапор **-d** - це скорочення для **--detach**. Ця команда запускає контейнер у фоновому режимі. Це дає змогу використовувати термінал, з якого запущено контейнер, для виконання інших команд під час роботи контейнера.

Перевірка стану контейнера

- Якщо у вас є запущені контейнери Docker і ви хочете дізнатися про те, що це за контейнери, вам знадобиться вивести їхній список. Зробити це можна такою командою:

docker container ls

- Ця команда виводить список контейнерів, що виконуються, і постачає цей список деякими корисними відомостями про них. Ось ще один приклад цієї команди:

docker container ls -a -s

- Ключ -a цієї команди - це скорочення для --all. Завдяки використанню цього ключа можна вивести відомості про всі контейнери, а не тільки про ті, що виконуються.

- Ключ **-s** - це скорочення для **--size**. Він дозволяє вивести розміри контейнерів.

- Ось команда, яка виводить докладні відомості про контейнер:

docker container inspect my_container

- Ось команда, що виводить логи контейнера:

docker container logs my_container

Завершення роботи контейнера

- Іноді працюючий контейнер треба зупинити. Для цього використовується така команда:

docker container stop my_container

- Вона дає змогу зупиняти працюючі контейнери, дозволяючи їм коректно завершити роботу. У контейнера є, за замовчуванням, 10 секунд, на те, щоб завершити роботу.

- Якщо ж контейнер потрібно зупинити швидко, не піклуючись про коректне завершення його роботи, можна скористатися такою командою:

docker container kill my_container

- Команда **kill**, якщо порівняти працюючий контейнер з увімкненим телевізором, нагадує вимкнення телевізора шляхом відключення його від електрики. Тому, в більшості ситуацій, для зупинки контейнерів рекомендується використовувати команду stop.

- Ось команда, яка дозволяє швидко зупинити всі контейнери, що працюють:

docker container kill $(docker ps -q)

- Для видалення зупиненого контейнера можна скористатися такою командою:

docker container rm my_container

- Ось команда, яка дозволяє видалити всі контейнери, які на момент виклику цієї команди не виконуються:

docker container rm $(docker ps -a -q)

Створення образів

- Ось команда, яка дозволяє збирати образи Docker:

docker image build -t my_repo/my_image:my_tag .

- У цьому разі створюється образ з ім'ям **my_image**, під час його збирання використовується файл Dockerfile, що знаходиться за вказаним шляхом або URL.

- Прапор **-t** - це скорочення для **--tag**. Він вказує Docker на те, що створюваному образу треба призначити наданий у команді тег. У цьому випадку це **my_tag**.

- Крапка наприкінці команди вказує на те, що образ треба зібрати з використанням файлу Dockerfile, що знаходиться в поточній робочій директорії.

- Після того, як образ зібрано, його можна відправити у віддалений реєстр. Завдяки цьому ним зможуть скористатися інші люди, його можна буде завантажити і запустити на іншому комп'ютері. Припустимо, ви хочете використовувати Docker Hub. Якщо так - вам знадобиться завести там обліковий запис. Користуватися цим ресурсом можна безкоштовно.

- Після того, як ви зареєструєтеся на Docker Hub, вам потрібно увійти в систему. І хоча команда, яка для цього використовується, безпосередньо до команд, призначених для роботи з образами, не належить, її корисно буде розглянути саме тут. Йдеться про таку команду:

docker login

- Вона дає змогу увійти в обліковий запис на Docker Hub. Для входу в систему вам знадобиться ввести ім'я користувача та пароль.

- Після входу в систему можна буде відправляти образи в реєстр. Робиться це так:

docker image push my_repo/my_image:my_tag

- Тепер, коли у вас набереться кілька образів, ви можете їх досліджувати за допомогою спеціальних команд.

Дослідження образів

- Ось команда, яка виводить список образів, виводячи, зокрема, і відомості про їхній розмір:

docker image ls

- Наступна команда дає змогу вивести відомості про проміжні образи, що входять до складу образу, зокрема - дані про їхні розміри і про те, як вони були створені:

docker image history my_image

- Ось команда, яка виводить докладні відомості про образ, зокрема - дані про шари, з яких складається образ:

docker image inspect my_image

- Якщо ви створите дуже багато образів, може трапитися так, що деякі з них знадобиться видалити.

Видалення образів

- Ось команда, яка дає змогу видалити вказаний образ:

docker image rm my_image

- Якщо образ зберігається у віддаленому репозиторії, його звідти видалено не буде.

- Ось команда, яка дозволяє видалити всі локальні образи:

docker image rm $(docker images -a -q)

- Користуватися цією командою варто з обережністю, але треба зауважити, що під час її використання образи, що зберігаються у віддаленому сховищі, видалено не буде. У цьому полягає одна з переваг зберігання образів у репозиторіях.