Eötvös Loránd University (ELTE)
Faculty of Informatics (IK)
Pázmány Péter sétány 1/c
1117 Budapest, Hungary

DOCKER TUTORIAL

Open Source Technologies for Real-Time Data Analytics

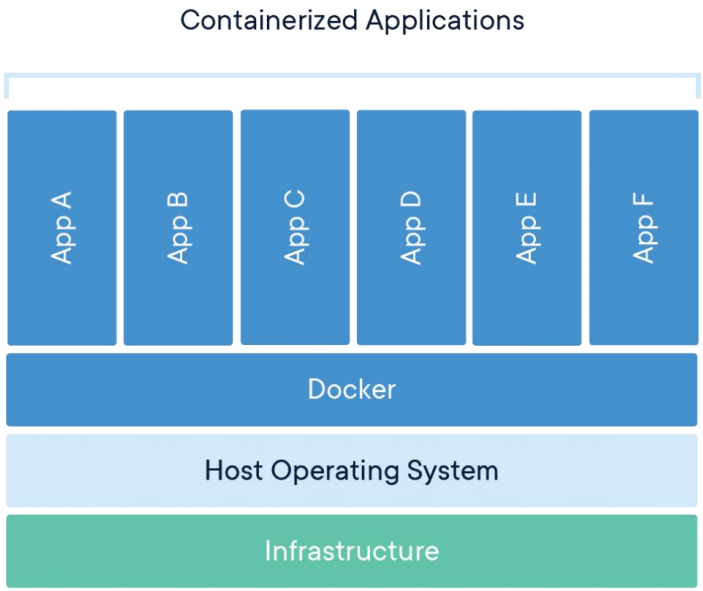
Imre Lendák, PhD, Associate Professor

2020
Budapest, Hungary

Introduction

- Introduction
- Working with Docker
  - Basic commands
  - Images
  - Networking
- Docker ecosystem
  - Docker Hub
  - Docker Compose
  - Kubernetes
- Security

Container intro

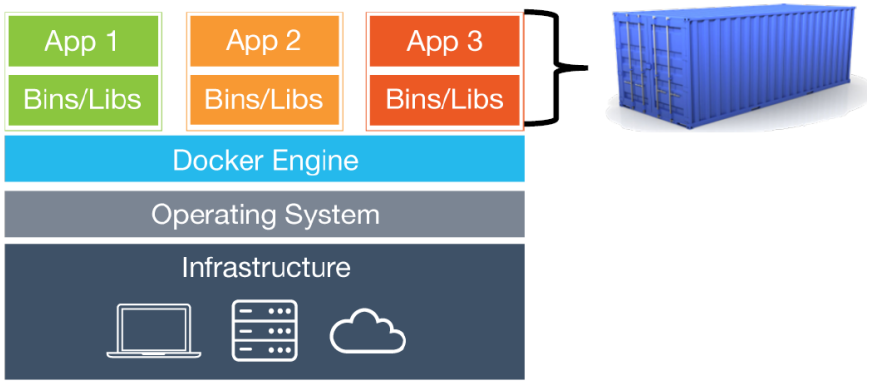


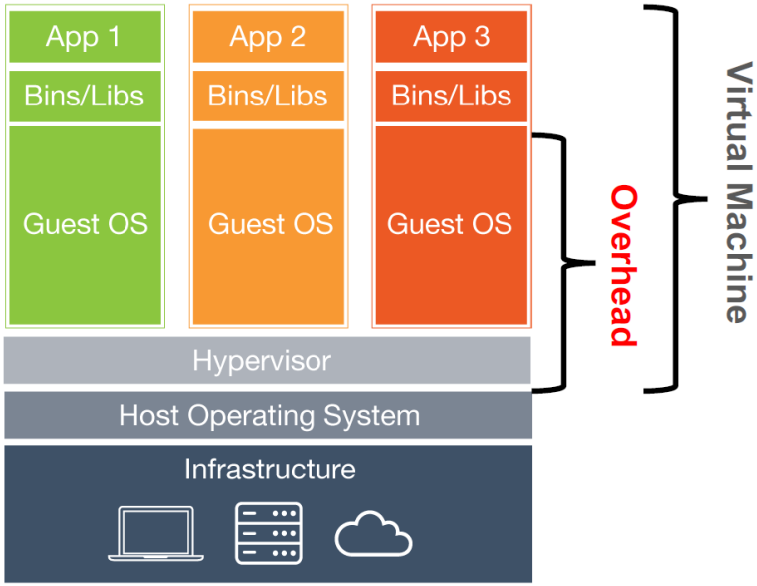
- **DEF:** A **container** is a lightweight, standalone, executable package of software that includes everything needed to run an application:
  - code,
  - runtime,
  - system tools,
  - system libraries and
  - settings.

https://www.docker.com/resources/what-container

Docker intro

- **DEF:** Docker is an open-source project that automates the deployment of software applications inside containers
  - Virtualization of application (i.e. process) instead of hardware
  - Runs on top of the core OS (Linux or Windows)
  - No dedicated CPU, memory, network requirements as in VMs
- Trivia:
  - **Initial release:** March 20th, 2013
  - **Latest stable release:** 19.03.13 (Sept 16, 2020)
  - **Original author(s):** Solomon Hykes
  - **License:** Apache License 2.0





Motivation

	Bare-metal	Virtual machine	Container
Underlying platform	N/A	Hypervisor on bare-metal	OS on VM or bare-metal
Performance	**Best**	Average	Average
Provisioning time	At least hours	Minutes	**Seconds**
Tenant isolation	Lowest (wo host OS)	Average (?)	Maximum (?)
Configuration flexibility	None	Average	**Best**
App portability	Backup & restore, ISO images	VM image, VM tools	**Best**
Granularity	Large	Average	Small

https://www.snia.org/sites/default/files/CSI/SNIA_Intro_to_Containers_Container_Storage_and_Docker_Final.pdf

Terminology

- **Images** are application blueprints which form the basis of containers
- **Containers** are created from Docker images and run the actual application. We create a container using ‘docker run’
- The **Docker Daemon** is the background service running on the host that manages building, running and distributing Docker containers. The daemon is the (only) process that is run by the core OS
  - The daemon is part of the Docker Engine
- **Docker Client** is the command line tool that allows the user to interact with the daemon. There can be other forms of clients apart from Docker Client
- **Docker Hub** is a registry of Docker images

https://docker-curriculum.com/#terminology

Docker installation

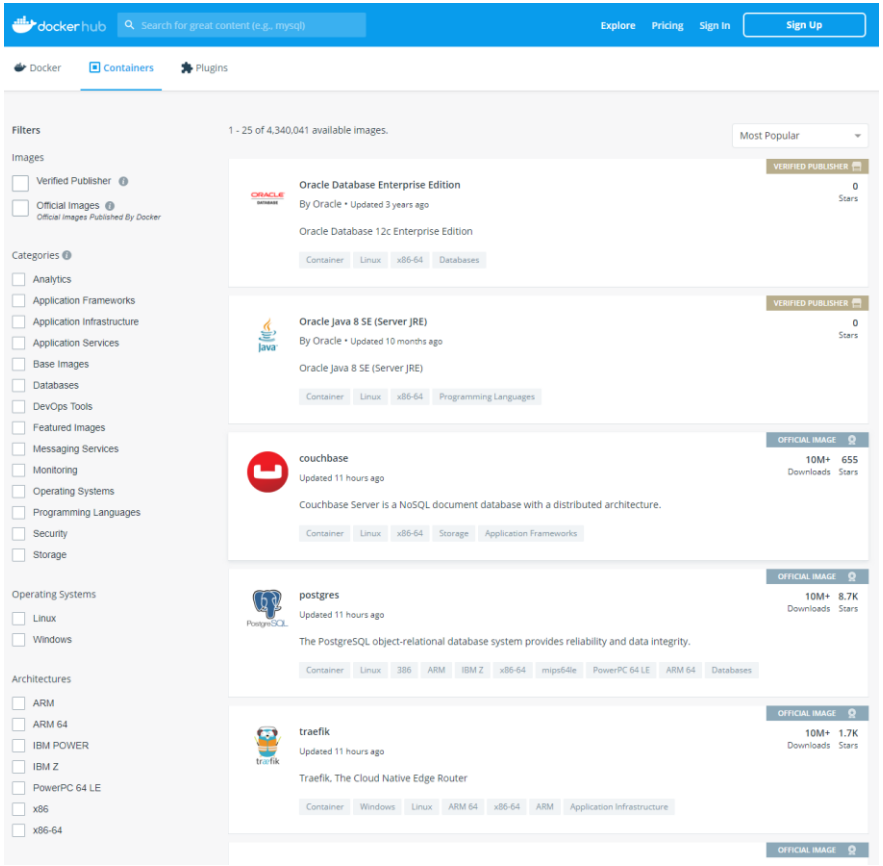
- The Docker platform consists (at least) of the following components
  - Docker Engine – the service capable to run processes in sandboxes inside a core OS
  - Docker Client – the client app used to manage images
  - Docker was originally built to run on Linux
- The Docker Desktop package is today (December 2020) available for Windows and Mac
  - URL: https://www.docker.com/products/docker-desktop
  - Docker Desktop contains both the Engine and Client
  - Installation steps: download, install & run the Engine

Docker Hub

Docker Hub

- **DEF: Docker Hub** is a registry of Docker images
  - Official Docker Hub URL: https://hub.docker.com/search?q=&type=image

- If required anybody can set up his/her Docker registry to store & pull images
  - Software producers often publish their own images



Basic commands

Basic commands

- $ docker version – check Docker version
- $ docker search – search the Docker Hub (DH) for images
- $ docker pull – pull image(s) from a repository (e.g. DH)
- $ docker images – command for working with images
- $ docker container – command family for working with containers

- $ docker run – run a container from an image
- $ docker ps – list active containers
- $ docker attach – attach to a container
- $ docker exec – execute a command in a container
- $ docker kill – kill one or more docker containers

Images

Working with images – 1

- $ docker pull elasticsearch:7.9.0
  - Pull the elasticsearch image from Docker Hub
- $ docker pull docker.elastic.co/elasticsearch/elasticsearch:6.3.2
  - Pull from Elastic's image registry
- $ docker images
  - List locally available images
- $ docker build
  - Build docker image
- $ docker run
  - Run an image in a local sandbox → results in a container (!)

Working with images – 2

- $ docker ps
  - Show all containers that are currently running on a core OS
- $ docker ps –a
  - Show all containers that we ran

- $ docker container ls
  - List running containers
- $ docker container prune
  - Remove containers

Hands-on #1: Elasticsearch

- $ docker pull elasticsearch:7.9.0
  - Note: pull ES does not work without specifying the required version
- $ docker run --rm -d --name es -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" elasticsearch:7.9.0
  - Note: --name allows us to name (i.e. tag) the container
  - Note: -p allows us to create port mappings between the container and the core OS
  - Note: --rm allows us to remove the named container after use → easier to re-run it afterwards, i.e. no need to remove it before re-running
- $ docker container logs es
  - Check the logs of the container
- $ docker ps
- $ curl -X GET "localhost:9200“
  - Check whether it works → should respond with elasticsearch cluster info (in JSON)
  - Note: curl is an open-source, command-line tool for transferring data with URLs
  - Note: curl should be installed prior to issuing the above command
- $ docker container stop es
- $ docker ps

https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html

Hands-on 2 & 3 directory structure

Directory structure

python_client

- Dockerfile
- python_client.py
- requirements.txt

python_server

- Dockerfile
- python_server.py
- requirements.txt

Comments

- Requirements installed via dockerfiles:
  - Linux packages
  - Python libraries
- File types usually copied to containers:
  - Configuration
  - Binary
  - Source code, e.g. Python code in our case
- Usually a single command is run in the container with CMD

Hands-on #2: Python service

Python server

```
import datetime
import socketserver
import time

class MyTCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        # self.request is the TCP socket connected to the client
        has_error = False
        try:
            self.data = self.request.recv(1024).strip()
            self.data = bytes(self.data)
            print(datetime.datetime.now(), " - {} wrote:".format(self.client_address[0]))
            print(' data: ', str(self.data))

            server_hello = "Python Server is up and running."
            send_data = server_hello.to_bytes()
            self.request.sendall(send_data)
        except:
            print('Exception occurred in handle.')

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        server.serve_forever()
```

Dockerfile

```
FROM python:3

RUN apt-get update && apt-get upgrade -y
RUN apt-get install -y sudo

WORKDIR /usr/src/app

COPY requirements.txt ./requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Secure against running as root, but allow sudo (for tcpdump)
RUN adduser --disabled-password --gecos '' python_server
RUN adduser python_server sudo
RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
WORKDIR /home/python_server

COPY python_server.py /python_server.py

EXPOSE 9999

CMD [ "python", "/python_server.py" ]
```

Hands-on #3: Python client

Python client

```
import socket
import os

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    HOST, PORT = "192.168.10.10", 9999

    binary_message = bytes([0, 0, 0])

    # Create a socket (SOCK_STREAM means a TCP socket)
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        # Connect to server and send data
        sock.connect((HOST, PORT))
        sock.sendall(binary_message)
        # Receive data from the server and shut down
        received = sock.recv(1024)
        print(received)
d
```

Dockerfile

```
FROM python:3

RUN apt-get update && apt-get upgrade -y
RUN apt-get install -y sudo
RUN apt-get install -y openssh-server nmap iputils-ping

COPY requirements.txt ./requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Secure against running as root
RUN useradd -rm -d /home/python_client -s /bin/bash -g root -G
sudo -u 1000 python_client
RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
RUN echo 'python_client:python_client' | chpasswd
RUN /usr/bin/ssh-keygen -A
RUN service ssh start

EXPOSE 22

WORKDIR /home/python_client
COPY python_client.py ./python_client.py

#CMD [ "python", "python_client.py" ]
CMD ["/usr/sbin/sshd","-D"]
```

Build & run both

Server

- $ docker build -t python_server .
  - Builds the container and tags it as ‘python_server’
- $ docker run -d --name python_server --rm -p 9999:9999 python_server
  - Run in detached mod
  - Set container name
  - Remove after use
  - Map ports to host ports

Client

- $ docker build -t python_server .
  - Note: the dot is for the local folder, where build will look for the dockerfile
- $ docker run -d --name python_client --rm python_client
- $ docker attach -it python_client sh
  - Run a single command in the container

Networking

Docker networks

- Q: Can the Python client and server communicate out-of-the-box with their listed implementations?
- A: Not really, it is necessary to properly set up a network for them.
  - **Note:** the containers will be assigned to different networks by default
- Common Docker network types:
  - **bridge**: the network in which containers are run by default – note: limited to a single host running the Docker engine
  - **overlay**: the network type used for multi-host network communication
  - **macvlan**: the network type used to connect Docker containers directly to the host network interface(s)

Setting up a network

- $ docker network ls
  - List docker networks
- $ docker network inspect
  - Inspect networks
- $ docker network create python_net
  - Create a new Docker network with default type: bridge

- $ docker run -d --name python_server --net python_net
- $ docker run -d --name python_client --net python_net
  - Both will be accessible via symbolic names, i.e. python_server and python_client on the python_net network

Docker Compose

Docker Compose

- **DEF:** Docker Compose is a tool for defining and running multi-container applications with Docker.
  - Comes pre-installed with Docker Desktop on Windows and Mac
  - Allows app developers to define multi-container applications in a single YAML configuration file
- Trivia:
  - Docker Compose was originally named Fig
  - Fig lead developer: Aanand Prasad
  - Fig was acquired by Docker in 2014 and renamed to Docker Compose

Compose configuration

```
version: "3.8"
services:
    python_server:
        build: ./python_server/.
        networks:
            python_net:
                ipv4_address: 192.168.10.1
        ports:
            - "9999:9999"
    python_client:
        build: ./python_client/.
        tty: true
        depends_on:
            - python_server
        networks:
            python_net:
                ipv4_address: 192.168.10.100
        ports:
            - 22:22
networks:
    python_net:
        ipam:
            driver: default
            config:
                - subnet: 192.168.10.0/24
```

- The YAML file is positioned outside the folders of the Docker containers
- Config file contents:
  - Version number (Docker Compose)
  - Services, i.e. containers
  - Networks → can be more than one

Compose commands

- Docker Compose command are (usually) issued via the command line (at least during the testing phase)
- $ docker-compose --version
- $ docker-compose up
  - Run a ‘composed’ app, i.e. cluster of containers & networks
  - Should be issued in the folder of the YAML file
- $ docker-compose ps
  - List containers
- $ docker network ls
  - List Docker networks → notice the network create by Compose
- $ docker-compose down
  - Destroy the cluster, i.e. containers & networks

Directories and files


```
python_app
        docker-compose.yml
        python_client
                Dockerfile
                python_client.py
                requirements.txt
        python_server
                Dockerfile
                python_server.py
                requirements.txt
```

Docker security

Docker security

1. Only use images from trusted vendors to avoid malware
2. Use thin, short-lived containers to reduce your attack surface
3. Avoid mixing containers to protect critical data
4. Use Docker Bench for Security to analyze your settings
5. Configure your containers properly to protect the host
6. Harden the host to safeguard all other containers in the environment
7. Employ secure computing mode (seccomp) to filter your system calls, e.g. filter out all audio-related api calls
8. Enable troubleshooting without logging in to limit SSH access

https://www.uptycs.com/blog/docker-security-best-practices

Summary

- Introduction
- Docker Hub
- Basic commands
- Images
- Networking
- Docker Compose
- Security

Thank you for your attention!