

# Efficient distribution of processes over the network

## Containerization with Docker

Davide Quaglia (from a work by Alberto Tessari)

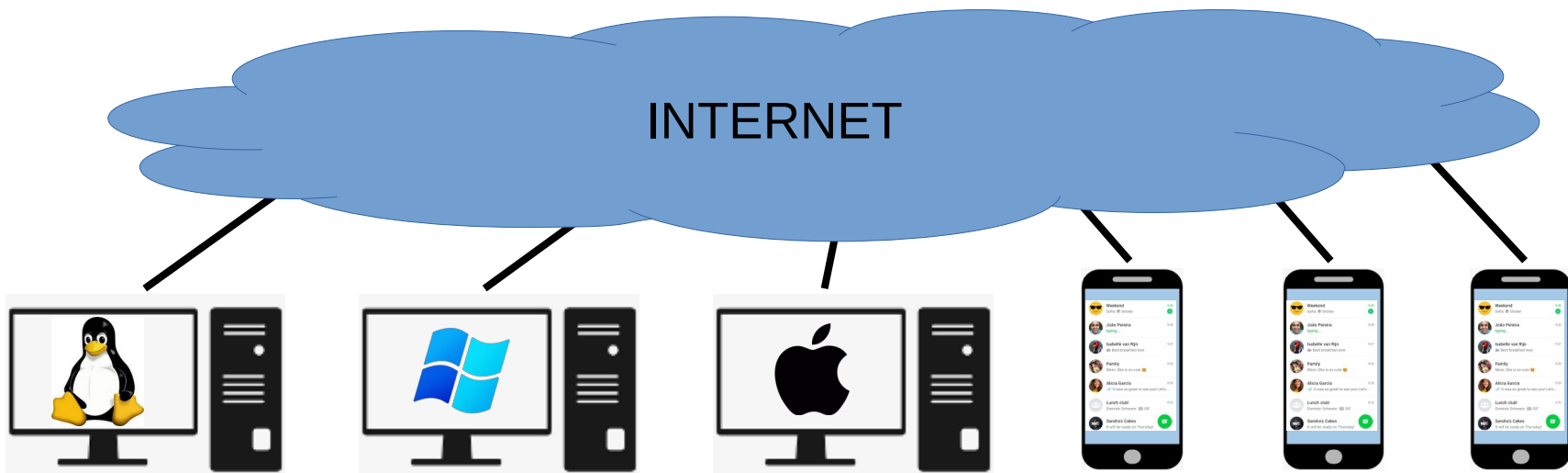


UNIVERSITÀ  
di **VERONA**  
Dipartimento  
di **INFORMATICA**

# Motivation 1

Hosts are always connected through the network which simplifies the distribution of software.

How to adapt to the heterogeneity of library versions, operating systems and CPUs?

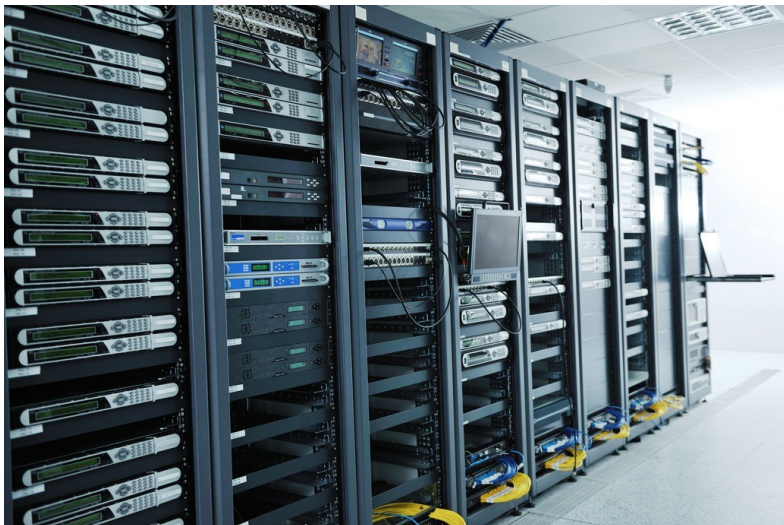


# Motivation 2

Well-known network applications are not hosted in isolated computers, but rather in:

- **Clusters:** groups of tens of computers connected by a high-performance LAN
- **Data centers:** groups of thousands of computers connected by a hierarchical set of high-performance LANs

cluster



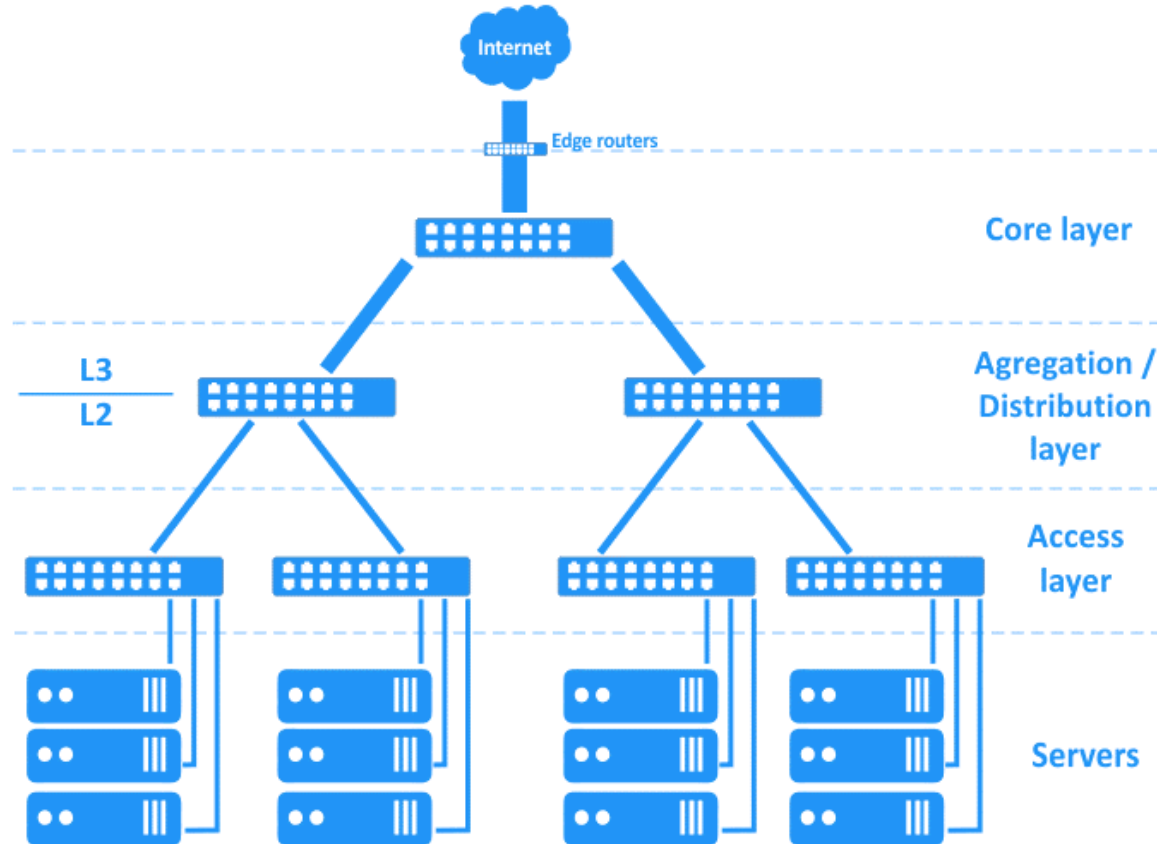
data center

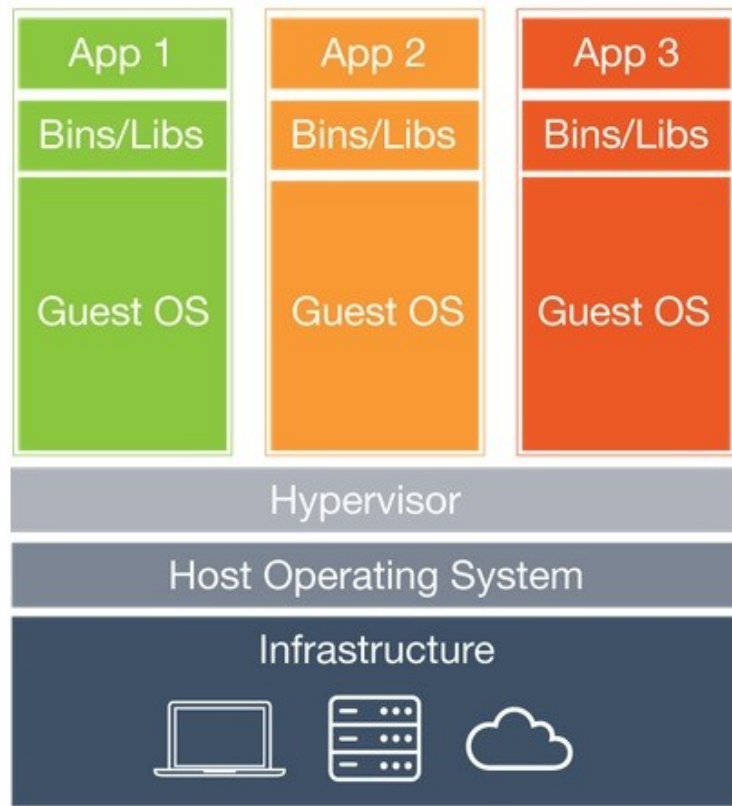


# Data center topology

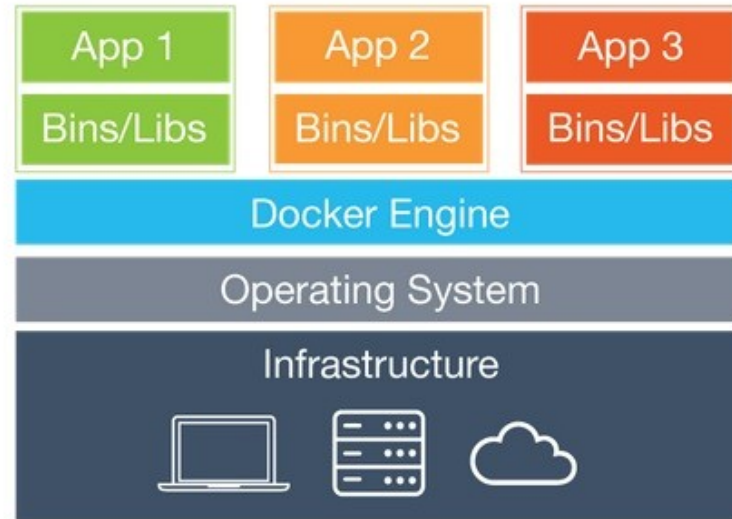
How to distribute efficiently many instances of server processes over the various computers?

- Load balancing
- No interference





**Virtual Machines**



**Containers**

## Virtualization versus Containerization

# What does “containerization” mean?

It is used to package, ship and deploy software so it can run on any hardware.

This concept is similar to the more traditional virtualization, but with less overhead to drastically increase scalability and performances.

# Why containers?

Using a container to ship an application, we are sure that our app will be executed with a specific set of software resources like: a specific OS distribution, a particular framework version and various dependencies such as libraries or modules.

Deploying on a container instead of bare metal increases:

- security through process-level isolation
- scalability since containers dynamically take and free resources
- portability through operating systems and hardware platforms
- stability avoiding conflicts between applications and preventing system updates to impact on the containerized application.

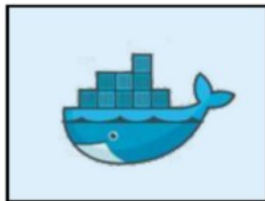
# Life cycle of a containerized application (with Docker)

Write a Dockerfile, build it into a Docker Image and run the image to deploy the actual Docker Container.



Dockerfile

Build



Docker  
Image

Run



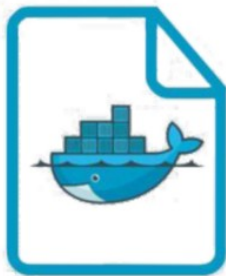
Docker  
Container



# Conceptualization

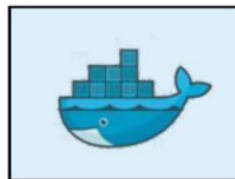
This is just a hint to help remembering! Not an official definition!

*Docker world*



Dockerfile

Build



Docker Image

Run



Docker Container

*Traditional  
single-  
platform  
world*

**MAKEFILE**

**EXECUTABLE**

**PROCESS**

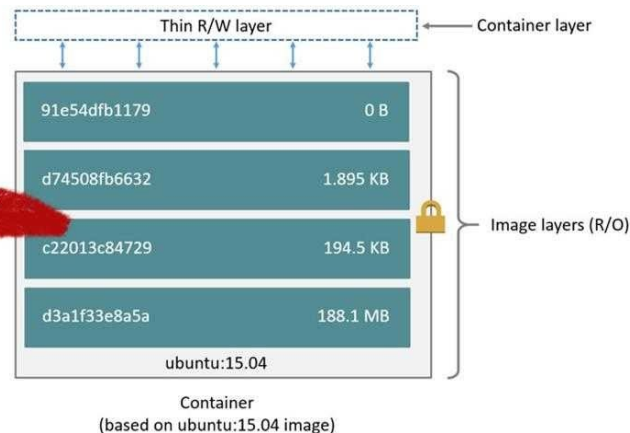
# DOCKERFILE

A Dockerfile is a list of instruction for the containerization engine to build a custom image that will contain the application we want to deploy.

In other words, it defines the specific environment we want for our app.

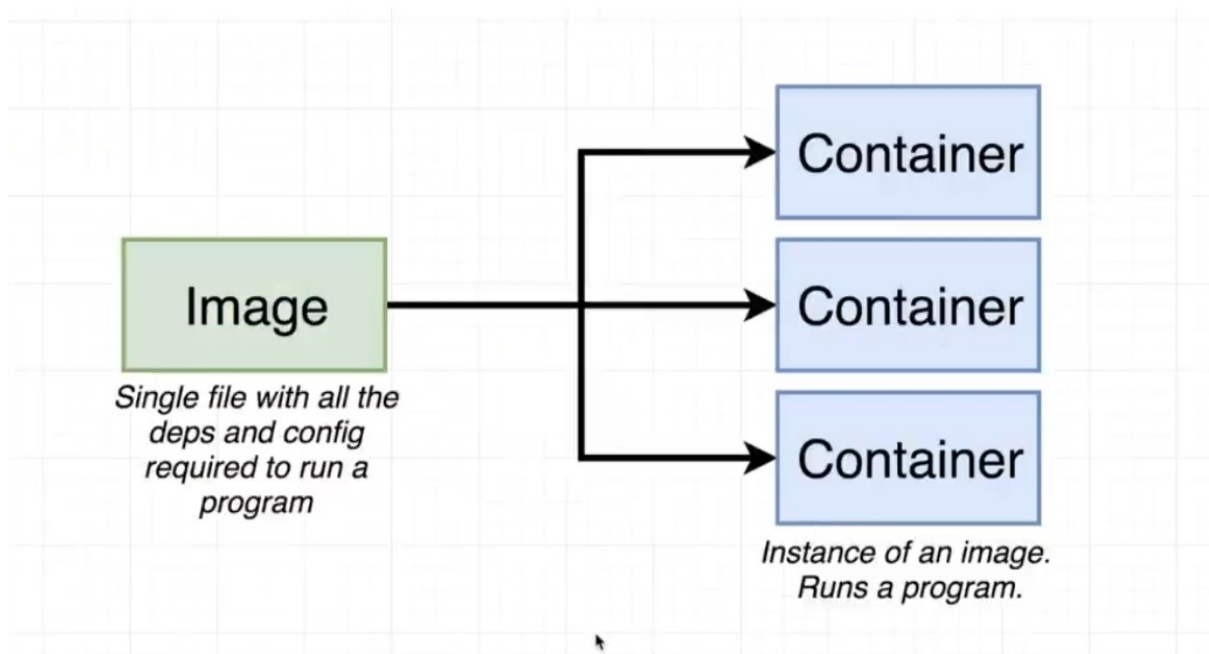
## Dockerfile

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```



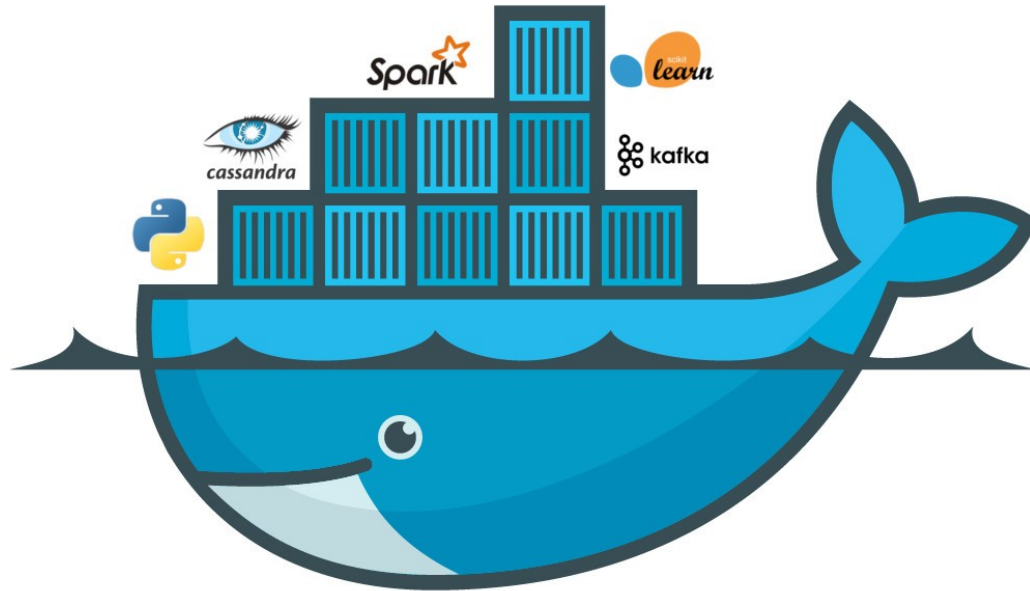
# Docker image

A Docker image is a template for running a Docker container.  
Once we build an image we can use it across different systems to run the same container.



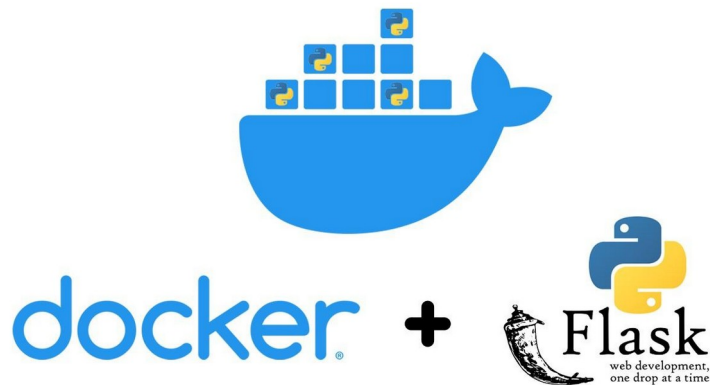
# Docker Container

A running process on top of the Host OS, isolated from other processes but it can provide services through open network ports.



A practical  
example:

# Flask server in a Docker Container



# Setting up Docker

## Installation on Windows/Mac

Docker Desktop on Windows/Mac: <https://www.docker.com/products/docker-desktop/>

## Linux installation

Docker on Ubuntu from the docker repository: [official documentation](#)  
(make sure to uninstall older docker versions)

or

Quick and easy Linux installation running the get-docker script:  
<https://get.docker.com/>

## Install Visual Studio Code

Download Visual studio code and it's Docker extension (test the integrity of the installations by running the command *docker* in the VSCode terminal).

# Useful docker commands

|                                |   |
|--------------------------------|---|
| docker ps                      | Give a list of all running containers on the system, every container has an id and a link to the image that generated it. Note: use option -a to list all the containers. |
| docker build                   | Build an image from a Dockerfile.   |
| docker pull                    | Download a pre-built image from the <b>Docker Hub</b>   |
| docker images                  | List available images.  |
| docker run                     | Create a container from an image and the start it.  |
| docker start/stop/restart/kill | Commands for starting, stopping, restarting and killing an existing container.  |
| docker rm                      | Delete a container.   |

# Write a simple web-app in Python using Flask

Create a directory (e.g. *flask\_web*) containing a python module (e.g. *app.py*) and a requirements file (*requirements.txt*).

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5
6  @app.route('/')
7  def docker_test():
8      return 'Docker and Flask are working!'
9
10
11  if __name__ == '__main__':
12      app.run(debug=True, host='0.0.0.0')
13
```

*app.py*

```
1
2  Flask==2.1.2
```

*requirements.txt*



# Writing a Dockerfile

When we write a Dockerfile we specify the layers our image will have, Dockerfiles usually start from a base layer containing a pre-built image of, for example, an operative system. There are all kind of pre-built images that can be found in the [docker-hub](https://hub.docker.com/).

Dockerfile flask\_web/Dockerfile/...

```
1 # Our image is built starting from the pre-built image Alpine:
2 # A minimal Docker image based on Alpine Linux
3 FROM alpine:latest
4
5 # installing Python and pip in Alpine
6 # with the keyword "ENV" we can set enviroment variables
7 ENV PYTHONBUFFERED=1
8 # using the command "RUN" we can execute shell commands while building an Image
9 RUN apk add --update --no-cache python3 && ln -sf python3 /usr/bin/python
10 RUN python3 -m ensurepip
11 RUN pip3 install --no-cache --upgrade pip setuptools
12
13 # Using the COPY command we can copy files from the local filesystem to
14 # the image's one and vice-versa
15 COPY ./requirements.txt /app/requirements.txt
16
```

```
16
17 # setting the working directory
18 WORKDIR /app
19
20 RUN pip install -r requirements.txt
21
22 COPY . /app
23
24 # With the keyword ENTRYPOINT we can set an executable to be run
25 # when the container starts up
26 ENTRYPOINT ["python"]
27 # we can use the CMD command to specify arguments for the executable
28 CMD ["app.py"]
29
30
31
```

# Building the Docker Image

After writing the dockerfile we can use it to build an image using the command *docker build* we can add the *-t* flag to specify the tag our container will have so we can identify the container easily.

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
~/D/u/t/d/flask_web >>> docker build -t docker-flask:1.0 .  
Sending build context to Docker daemon 4.608kB  
Step 1/11 : FROM alpine:latest  
latest: Pulling from library/alpine  
df9b9388f04a: Pull complete  
Digest: sha256:4edbd2beb5f78b1014028f4fbb99f3237d9561100b6881aaf  
Status: Downloaded newer image for alpine:latest
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
Removing intermediate container 3c7e8afe4706  
---> 87c627529d69  
Step 9/11 : COPY . /app  
---> d203fba74347  
Step 10/11 : ENTRYPOINT ["python"]  
---> Running in 0fe17533def8  
Removing intermediate container 0fe17533def8  
---> bef3703c568d  
Step 11/11 : CMD ["app.py"]  
---> Running in ffdc205e9982  
Removing intermediate container ffdc205e9982  
---> 77edaff8730d  
Successfully built 77edaff8730d  
Successfully tagged docker-flask:1.0  
~/D/u/t/d/flask_web >>> docker images  
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE  
docker-flask        1.0            77edaff8730d   52 seconds ago 87.5MB  
alpine              latest         0ac33e5f5afa   4 weeks ago   5.57MB  
~/D/u/t/d/flask_web >>>
```

# Creating and starting the container

After building the image we can start the container with the following command:

```
docker run -d -p 5000:5000 docker-flask:1.0
```

Options and arguments explanation:

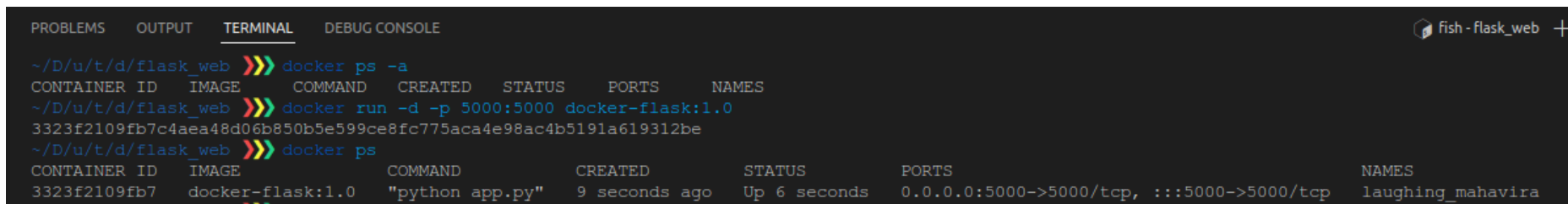
-d is used to run the image in a separated container

-p is used to map container's port(s) to host's port(s); port order is *host:container*

docker-flask:1.0 is the tag of the image we built with the previous command.

We can add the `-name='name'` option to give a custom name to the container, otherwise docker will generate a random one.

# Creating and starting the container



A terminal window titled 'fish - flask\_web' showing the following commands and output:

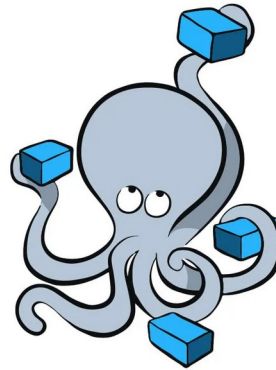
```
~/D/u/t/d/flask_web >>> docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS        NAMES
~/D/u/t/d/flask_web >>> docker run -d -p 5000:5000 docker-flask:1.0
3323f2109fb7c4aea48d06b850b5e599ce8fc775aca4e98ac4b5191a619312be
~/D/u/t/d/flask_web >>> docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS        NAMES
3323f2109fb7  docker-flask:1.0                    "python app.py"         9 seconds ago  Up 6 seconds  0.0.0.0:5000->5000/tcp, :::5000->5000/tcp  laughing_mahavira
```

Then open a web browser at <http://localhost:5000>

# Exercises

- 1) Stop, restart and kill the previously created container and see what happens by using the command “docker ps -a”
- 2) Remove the container and create a new one listening on the port 8000
- 3) Execute the same image in two different containers on the port 7000 and 8000, respectively.

# Multi-containers projects



docker  
Compose

# Multi-containers projects

When a project needs a Docker Image providing a variety of services, developers often rely on a tool called Docker Compose. Note: if you followed the linux installation using the `get-docker` script, you may need to run the command *`sudo apt install docker-compose`* to install this tool.

With Docker Compose, developers can write a YAML file describing all the application services, then with a single command the user can run all the containers providing those services needed to start the application. For example we can use Docker Compose to ship a project that provides two services: a front-end application and a database, running in two communicating containers, when we access the front-end service, it can interact with the database to reply using stored data.

Creating such complex projects is out of the scope of this overview, but, we can easily run this type of project using Docker Compose.

Once we have retrieved the image we want to start the container(s) from (for example using the command *`docker pull`* to download an image from docker hub) we should download the *`docker-compose.yml`* file, provided by the developers who made the image, and start all the containers at once with the command *`docker compose up`* (or *`docker-compose up`* for older versions).

We can add various flags/options when launching this command, for example we can start a container in detached mode with the `-d` option.

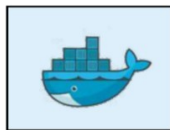
# Next step?

*Docker world*



Dockerfile

Build



Docker Image

Run



Docker Container



Kubernetes

---

*Traditional  
single-  
platform  
world*

**MAKEFILE**

**EXECUTABLE**

**PROCESS**

**OPERATING  
SYSTEM**