

Systems of Systems (SoS)

M.Sc. In Critical Computing Systems Engineering

ISEP/IPP – 2021/22, 2nd semester

Assignment 3:

Docker Cluster

Pedro Santos



Introduction to Containers/Docker

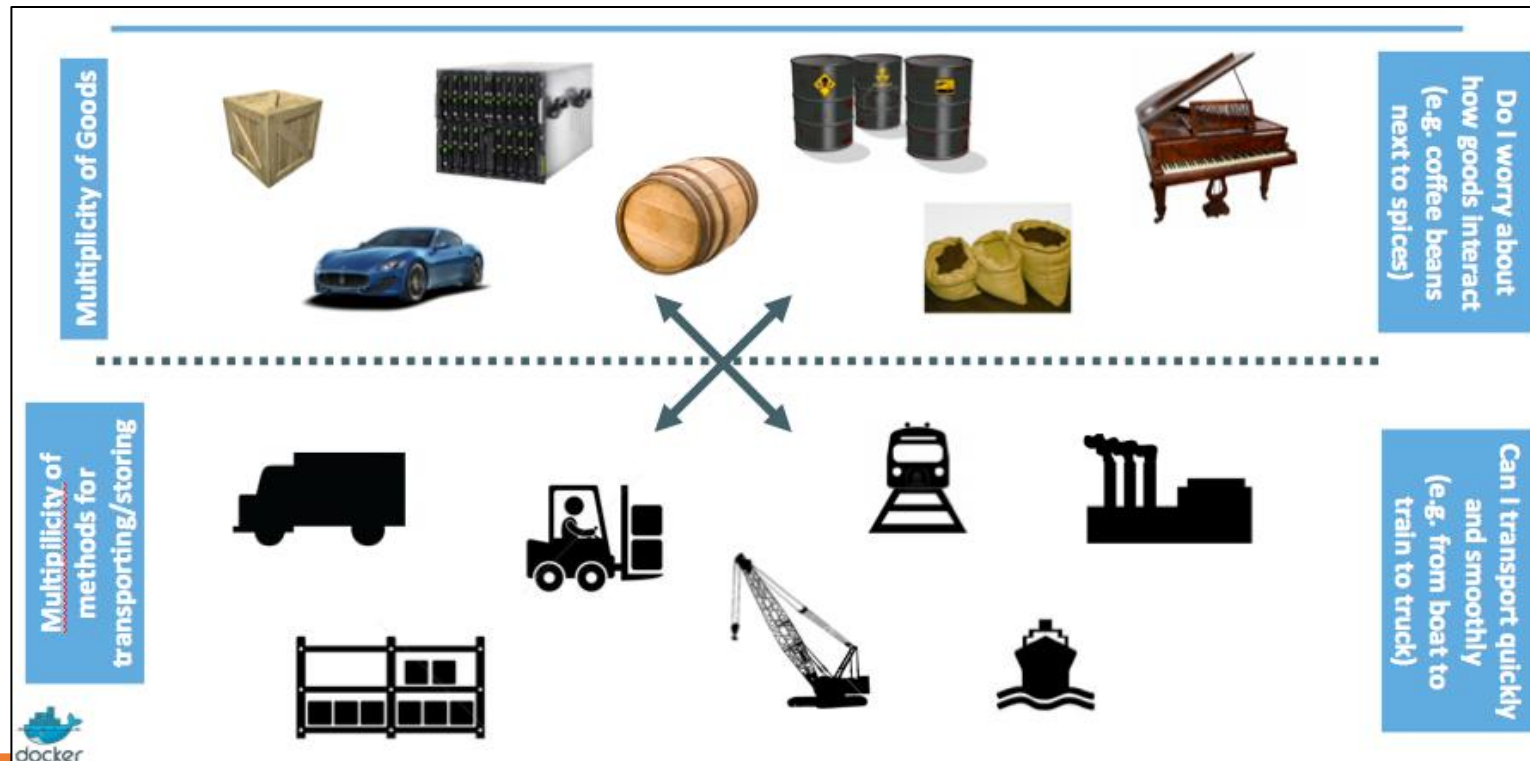


What is a Container?

- Containerization is an approach to software development in which an **application or service, its dependencies, and its configuration (abstracted as deployment manifest files)** are packaged together as a container image.
- **What?**
Simply put, a container is a sandboxed process on your machine that is isolated from all other processes on the host machine. Each container can run a whole web application or a service.
- **How?**
That isolation leverages kernel namespaces and cgroups. These features that have been in Linux for a long time, but Docker has worked to make these capabilities approachable and easy to use.
- **What do I need to run a container?**
Containerized applications run on top of a **container host** that in turn runs on the OS (Linux or Windows), therefore have a significantly smaller footprint than virtual machine (VM) images.
- To summarize, a container:
 - is a runnable instance of an image. You can create, start, stop, move, or delete a container using the DockerAPI or CLI.
 - can be run on local machines, virtual machines or deployed to the cloud.
 - is portable (can be run on any OS).
 - is isolated from another containers and run their own software, binaries, and configurations.

Why “Container”?

- Just as shipping containers allow goods to be transported by ship, train, or truck regardless of the cargo inside, software containers act as a standard unit of software deployment that can contain different code and dependencies.
- Containerizing software this way enables developers and IT professionals to deploy them across environments with little or no modification.

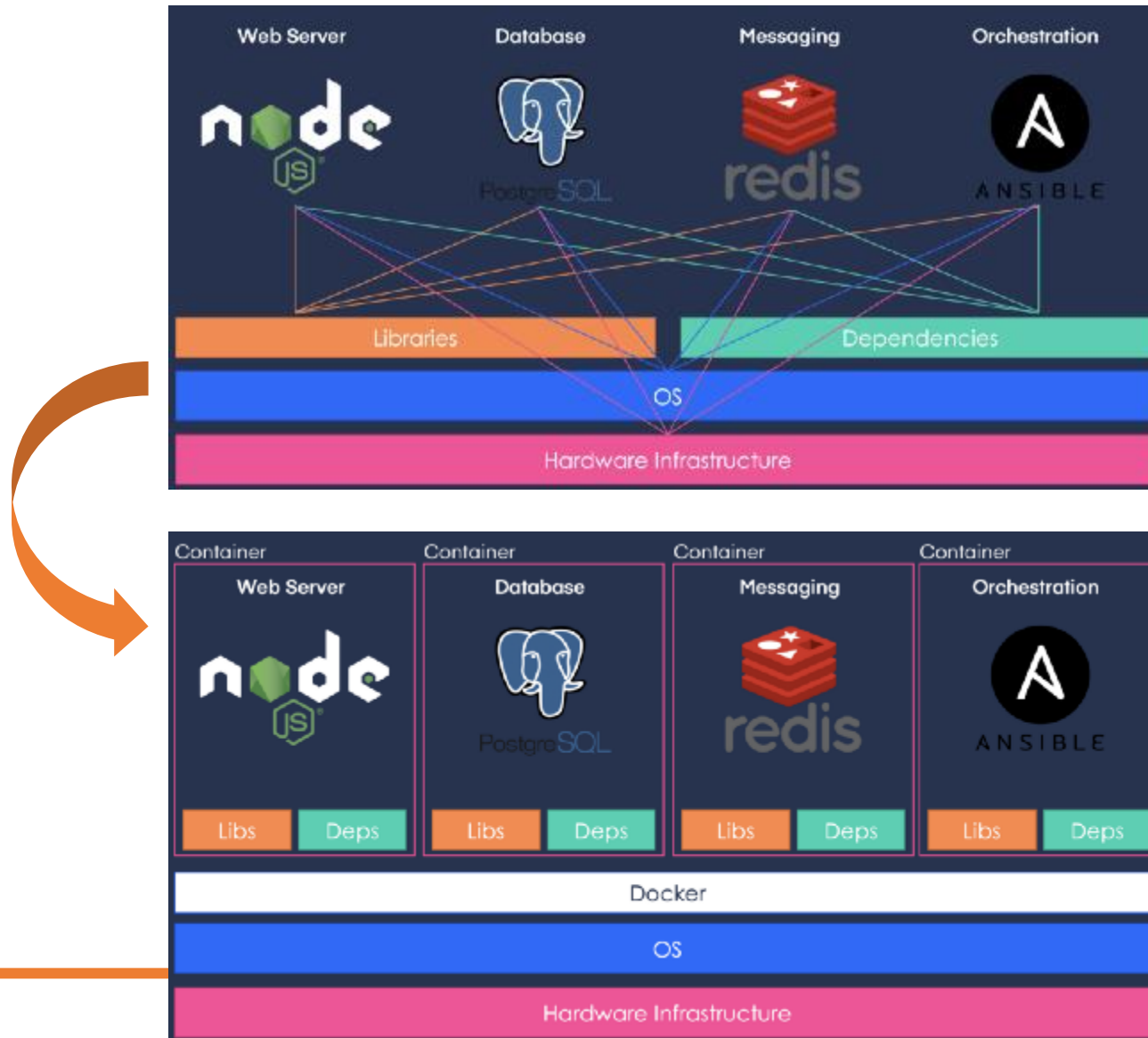


Why “Container”?

- Just as shipping containers allow goods to be transported by ship, train, or truck regardless of the cargo inside, software containers act as a standard unit of software deployment that can contain different code and dependencies.
- Containerizing software this way enables developers and IT professionals to deploy them across environments with little or no modification.



Why “Container”?



Docker & Containers

- Why is Docker synonymous with containers?
 - Docker set the industry standard, to a point where the terms '*Docker*' and '*Containers*' are used interchangeably
 - However, there are other tools for containerization
- What other alternatives are there?
 - [Artifactory Docker Registry](#)
 - [LXC/LXD](#) (Linux)
 - [Hyper-V](#) and Windows Containers
 - [rkt](#) (works with Kubernetes)
 - [Podman](#) (open-source container engine)
 - [runC](#) (portability solution)
 - [containerd](#) (a container runtime)
 - [Buildah](#)
 - [BuildKit](#)
 - [Kaniko](#)

Goal & Parts of Assignment 3

Goal:

- Become capable of using containers, a virtualization technique to conveniently deploy, scale and manage services
- Learn how to set up simple services (single container)
- Learn how to set up complex services (multi-container)
- Learn how to deploy and manage containers over multiple devices

Parts:

1. Part1: Setting up and Running a Container
 2. Part 2: Multi-container Applications
 3. Part 3: Set up a Docker Swarm
-

Installing Docker

A solid orange horizontal bar spanning the width of the slide, located at the bottom.

Installing Docker

- Linux or VM machine necessary in principle (for Windows, WSL may work, although no guarantees)
 - If using VM but SSH'ing from native OS, you need to port forward
- Installing Docker:
 - Convenience script: <https://docs.docker.com/engine/install/ubuntu/#install-using-the-convenience-script>
 - Run docker without *sudo* (strongly recommended): <https://docs.docker.com/engine/install/linux-postinstall/>
 - <https://docs.docker.com/engine/security/rootless/>
- Installing Docker Compose:
 - There are two versions of Docker, v1 or v2
 - If using v1: when running the tutorial scripts, do: “./services create **legacy**; ./services start **legacy**”
 - If using v2
 - Make sure to uninstall Compose v1: <https://docs.docker.com/compose/install/#uninstallation>
 - Follow these instructions: <https://docs.docker.com/compose/cli-command/#install-on-linux>

Part 1: Setting Up and Running Containers



Setting Up and Running Containers

Overview

- We will set up a container that provides a basic website.
- We will use NGINX, a web server (can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache) in its containerized mode.

Topics addressed & Learning Objectives:

- We will discuss the difference between a **container image** (the blueprint of containerized application) and a **container** (a running instance of an image)
- We will learn about *Dockerfile*, a text document that contains all the commands a user could call on the command line to assemble an image.
- Finally, we will learn how to launch the container

Tutorial adapted from:

<https://training.play-with-docker.com/beginner-linux/>

Container & Images

Images

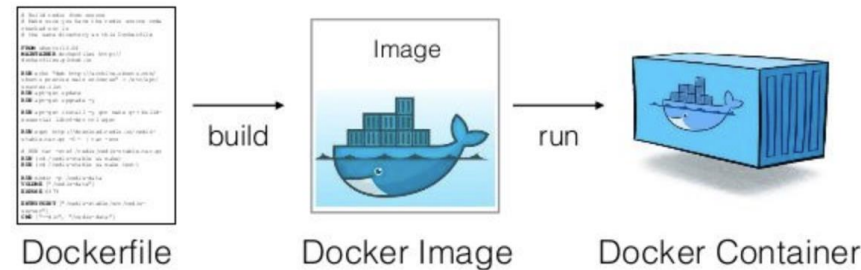
- When running a container, it uses an isolated filesystem. This custom filesystem is provided by a container image.
- Since the image contains the container's filesystem, it must contain everything needed to run an application - all dependencies, configuration, scripts, binaries, etc.
- The image also contains other configurations for the container, such as environment variables, a default command to run, and other metadata.
- In a sense, **an image is a read-only template with instructions for creating a Docker container.**

Containers

- A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
- A container is a standard Linux process typically created through a clone() system call instead of fork() or exec().
- By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Dockerfile

- Dockerfiles are used by Docker to build images automatically by reading the instructions, thus offering flexibility, readability, accountability and helps in easy versioning of the project.



- Looking into the Dockerfile:
 - FROM** specifies the base image to use as the starting point for this new image you're creating. For this example we're starting from nginx:latest.
 - COPY** copies files from the Docker host into the image, at a known location. In this example, COPY is used to copy two files into the image: index.html and a graphic that will be used on our webpage.
 - EXPOSE** documents which ports the application uses.
 - CMD** specifies what command to run when a container is started from the image. Notice that we can specify the command, as well as run-time arguments.

```
FROM nginx:latest

COPY index.html /usr/share/nginx/html
COPY linux.png /usr/share/nginx/html

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

Tutorial adapted from:

<https://training.play-with-docker.com/beginner-linux/>

Setting up a Website using Docker

1. `git clone https://github.com/dockeramples/linux_tweet_app`

2. `cd ~/linux_tweet_app`

3. `docker image build -t linux_tweet_app:1.0 .`

4. `docker container run \`
`--detach \`
`--publish 80:80 \`
`--name linux_tweet_app \`
`$DOCKERID/linux_tweet_app:1.0`

5. Access <http://<RPI IP>:80>. What do you observe?

- List running containers

```
docker ps
```

- List of images

```
docker image ls
```

Tutorial adapted from:

<https://training.play-with-docker.com/beginner-linux/>

Part 2:

Multi-container Apps

A solid orange horizontal bar spanning the width of the slide, located at the bottom.

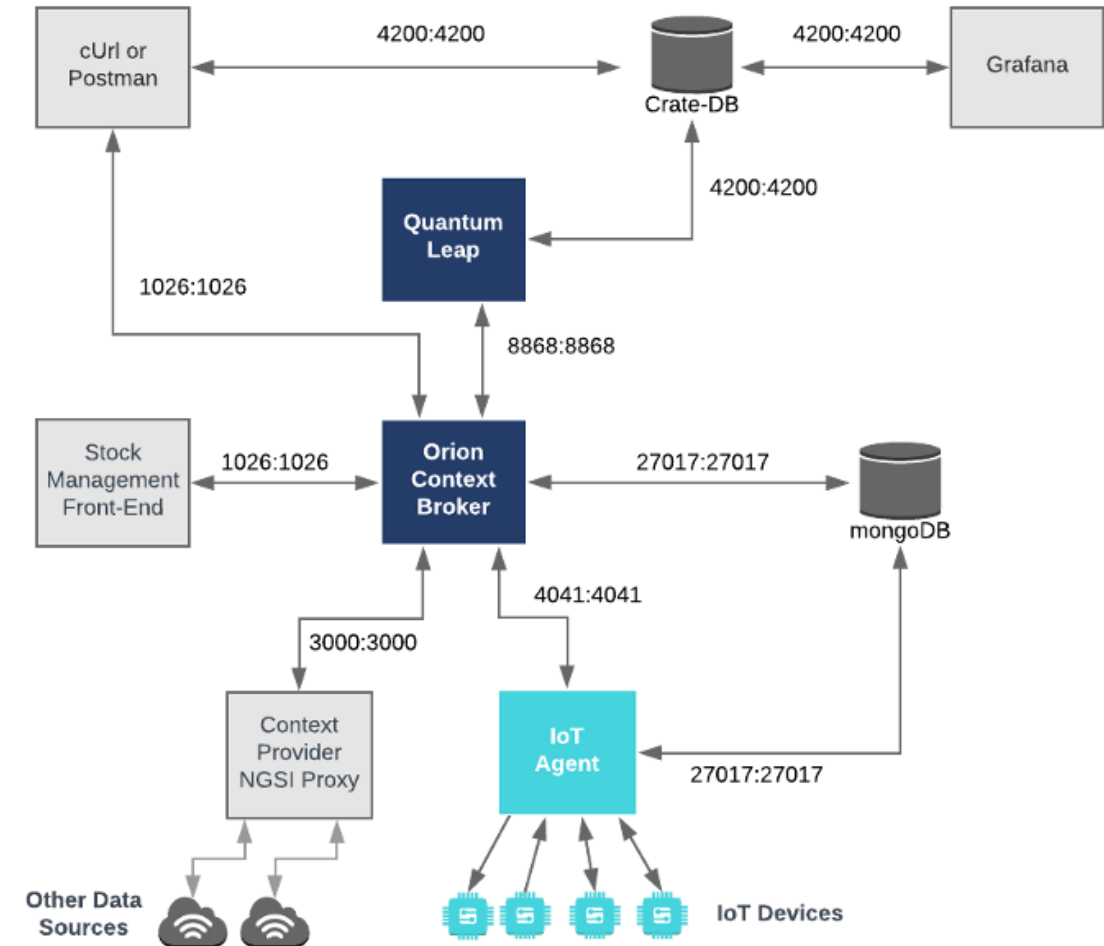
Multi-container Applications

Motivation

- Often you will need to set up multiple containers cooperating among themselves.
- Example: a FIWARE ecosystem that can be fully deployed in containers:
 - <https://fiware-tutorials.readthedocs.io/en/latest/time-series-data.html>

Overview

- In this tutorial we will learn about basic application containerization using Docker and running various components of an application as microservices.
- We will utilize Docker Compose for orchestration during the development.



Example of a multi-container application

Overview

- We will start with a Python script that scrapes links from a web page, and gradually evolve it into a multi-service application stack.
- After completion, the application stack will contain the following microservices:
 1. A web application written in PHP and hosted in Apache that takes a URL as the input and summarizes extracted links from it
 2. The web application talks to an API server written in Python that takes care of the link extraction and returns a JSON response
 3. A Redis cache that is used by the API server to avoid repeated fetch and link extraction for pages that are already scraped. The API server will only load the page of the input link from the web if it is not in the cache.



Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

Micro-Service Orchestration

- Step 1: Containerized Link Extractor Script
- Step 2: Link Extractor API Service
- Step 3: Link Extractor API and Web Front End Services

Optional:

- Step 4: Redis Service for Caching
- Step 5: Swap Python API Service with Ruby

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

Background: Link Extractor (LE) Script

- Link Extractor Python Script

```
#!/usr/bin/env python

import sys
import requests
from bs4 import BeautifulSoup

res = requests.get(sys.argv[-1])
soup = BeautifulSoup(res.text, "html.parser")
for link in soup.find_all("a"):
    print(link.get("href"))
```

- See it in operation:

```
git clone https://github.com/ibnesayed/linkextractor.git
cd linkextractor
git checkout demo
git checkout step0
python3 linkextractor.py https://www.jn.pt
```

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

1. Building the Link Extractor Container Image

1. Pre-requisites:

Dockerfile

```
FROM      python:3
LABEL     maintainer="Sawood Alam <@ibnesayeed>"

RUN       pip install beautifulsoup4
RUN       pip install requests

WORKDIR   /app
COPY      linkextractor.py /app/
RUN       chmod a+x linkextractor.py

ENTRYPOINT ["./linkextractor.py"]
```

Filetree:

```
.
├── Dockerfile
├── README.md
└── linkextractor.py
```

2. Building the image

```
git checkout step1
docker image build -t linkextractor:step1 .
```

3. Test it. What's the outcome?

```
docker container run -it --rm linkextractor:step1 http://jn.pt/
```

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

2. Link Extractor API Service

Motivation

- Suppose we wish to make our Link Extractor service accessible to other services.
- We will set up our server accessible as a WEB API at `http://<hostname>[:<prt>]/api/<url>`
- Output is provided in JSON format

Overview

- Main server script: `main.py`
- When API is accessed,
 1. URL is decoded
 2. Links are extracted
 3. Results are formatted in JSON and returned

```
#!/usr/bin/env python

from flask import Flask
from flask import request
from flask import jsonify
from linkextractor import extract_links

app = Flask(__name__)

@app.route("/")
def index():
    return "Usage: http://<hostname>[:<prt>]/api/<url>"

@app.route("/api/<path:url>")
def api(url):
    qs = request.query_string.decode("utf-8")
    if qs != "":
        url += "?" + qs
    links = extract_links(url)
    return jsonify(links)

app.run(host="0.0.0.0")
```

main.py

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

2. Link Extractor API Service

Other changes

- Dependencies are moved to the requirements.txt file
- Server script ./linkextractor.py → main.py (and dockerfile is updated to refer to the main.py file)

Dockerfile

```
FROM      python:3
LABEL     maintainer="Sawood Alam <@ibnesayeed>"

WORKDIR   /app
COPY      requirements.txt /app/
RUN       pip install -r requirements.txt

COPY      *.py /app/
RUN       chmod a+x *.py

CMD       ["./main.py"]
```

requirements.txt

```
beautifulsoup4
flask
requests
```

Filetree

```
.
├── Dockerfile
├── README.md
├── linkextractor.py
├── main.py
└── requirements.txt
```

Previous Dockerfile

```
FROM      python:3
LABEL     maintainer="Sawood Alam <@ibnesayeed>"

RUN       pip install beautifulsoup4
RUN       pip install requests

WORKDIR   /app
COPY      linkextractor.py /app/
RUN       chmod a+x linkextractor.py

ENTRYPOINT ["./linkextractor.py"]
```

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

2. Link Extractor API Service

1.

```
git checkout step3  
docker image build -t linkextractor:step3 .
```
2. We need to set up port mapping to make the service accessible outside of the container (the Flask server used here listens on port 5000 by default).

```
docker container run -d -p 5000:5000 --name=linkextractor linkextractor:step3
```

3. Test it. What's the outcome?

```
curl -i http://localhost:5000/api/http://jn.pt/  
docker container logs linkextractor
```

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

3. Link Extractor API and Web Front End Services

Motivation

- Suppose now that we wish to have a web-based graphical interface.
- The web application is made accessible at `http://<hostname>[:<prt>]/?url=<url-encoded-url>`

Docker Compose

- We will use **Docker Compose** to set up the multi-container application.
- A `docker-compose.yml` file is written to build various components and glue them together

```
version: '3'

services:
  api:
    image: linkextractor-api:step4-python
    build: ./api
    ports:
      - "5000:5000"
  web:
    image: php:7-apache
    ports:
      - "80:80"
    environment:
      - API_ENDPOINT=http://api:5000/api/
    volumes:
      - ./www:/var/www/html
```

docker-compose.yml

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

Micro-Service Orchestration



Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

3. Link Extractor API and Web Front End Services

File structure:

- The link extractor JSON API service (written in Python) is moved in a separate `./api` folder that has the exact same code as in the previous step
- A web front-end application in PHP, provided by `index.php` under `./www` folder, talks to the JSON API

Filetree

```
.  
├── README.md  
├── api  
│   ├── Dockerfile  
│   ├── linkextractor.py  
│   ├── main.py  
│   └── requirements.txt  
├── docker-compose.yml  
└── www  
    └── index.php
```

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

3. Link Extractor API and Web Front End Services

File structure:

- The link extractor JSON API service (written in Python) is moved in a separate `./api` folder that has the exact same code as in the previous step
- A web front-end application in PHP, provided by `index.php` under `./www` folder, talks to the JSON API

Filetree

```
.
├── README.md
├── api
│   ├── Dockerfile
│   ├── linkextractor.py
│   ├── main.py
│   └── requirements.txt
├── docker-compose.yml
└── www
    └── index.php
```

```
$api_endpoint = $_ENV["API_ENDPOINT"] ?:
"http://localhost:5000/api/";
$url = "";
if(isset($_GET["url"]) && $_GET["url"] != "") {
    $url = $_GET["url"];
    $json = @file_get_contents($api_endpoint . $url);
    if($json == false) {
        $err = "Something is wrong with the URL: " . $url;
    } else {
        $links = json_decode($json, true);
        $domains = [];
        foreach($links as $link) {
            array_push($domains, parse_url($link["href"],
PHP_URL_HOST));
        }
        $domainct = @array_count_values($domains);
        arsort($domainct);
    }
}
```

index.php

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

3. Link Extractor API and Web Front End Services

Notes on Docker Compose

- The PHP application is mounted inside the official php:7-apache Docker image for easier modification during the development
- An environment variable API_ENDPOINT is used inside the PHP application to configure it to talk to the JSON API server

Filetree

```
.
├── README.md
├── api
│   ├── Dockerfile
│   ├── linkextractor.py
│   ├── main.py
│   └── requirements.txt
├── docker-compose.yml
└── www
    └── index.php
```

```
version: '3'

services:
  api:
    image: linkextractor-api:step4-python
    build: ./api
    ports:
      - "5000:5000"
  web:
    image: php:7-apache
    ports:
      - "80:80"
    environment:
      - API_ENDPOINT=http://api:5000/api/
    volumes:
      - ./www:/var/www/html
```

docker-compose.yml

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

3. Link Extractor API and Web Front End Services

1.

```
git checkout step4  
docker-compose up -d --build
```

2. Run the list of containers. What do you observe?

```
docker container ls
```


3. Access <http://<RPI IP>:80>. Insert a URL. What do you observe?

Tutorial adapted from:

<https://training.play-with-docker.com/microservice-orchestration/>

Part 3:

Docker Swarm

A solid orange horizontal bar spanning the width of the slide, located at the bottom.

What is a Swarm?

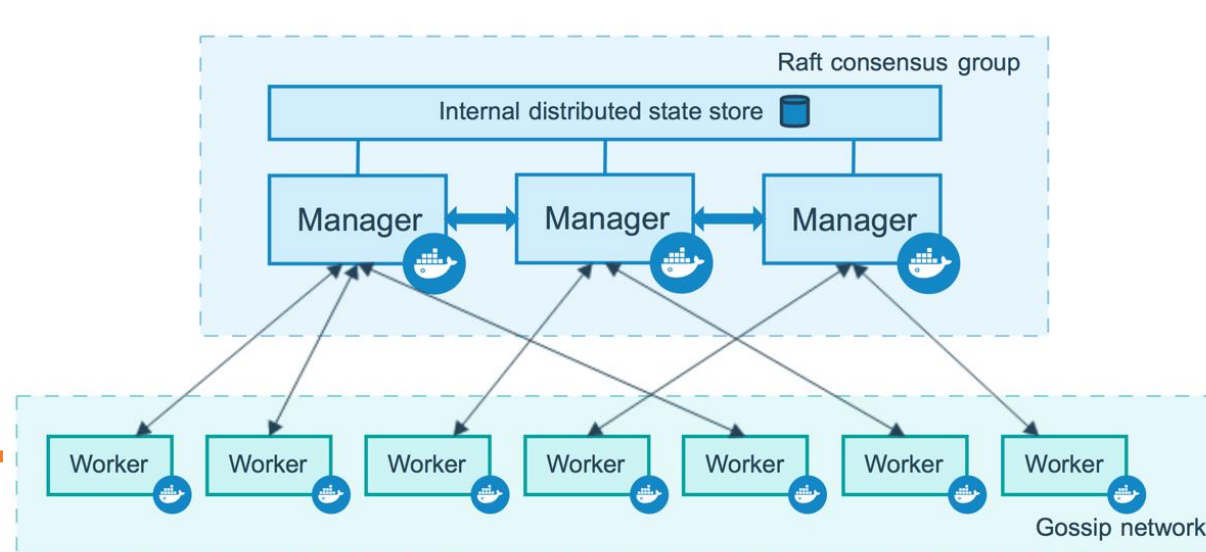
- **What is a Docker Swarm?**

- A swarm consists of multiple Docker hosts which run in swarm mode and act as **managers** (to manage membership and delegation) and **workers** (which run swarm services).
- A given Docker host can be a manager, a worker, or perform both roles.

- **Why:**

- When you create a service, you define its optimal state (number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more).
- Docker works to maintain that desired state. For instance, if a worker node becomes unavailable, Docker schedules that node's running containers on other nodes.

- **Analogy:** In the same way that you can use Docker Compose to define and run containers, you can define and run Swarm service stacks.



Docker Swarm

Create a swarm

1. To start a swarm, we run the following command in the first node.
We just need to do this in one node, and this node will be a manager.

```
ssh pi@docker1  
sudo docker swarm init --advertise-addr 192.168.1.181
```

2. The previous command outputs a command that can be used to join other nodes to the swarm as a worker.



```
docker swarm join \  
  --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxxv8rssmk743ojnwacrr2e7c \  
  192.168.1.181:2377
```

3. Check nodes currently in Swarm:

```
sudo docker node ls
```

Add another Manager

1. In Node 1, let's create a token for adding managers to the swarm.

```
sudo docker swarm join-token manager
```

2. The previous command outputs a command that can be used to join other nodes to the swarm as a manager.



```
docker swarm join --token SWMTKN-abc...manager...xyz 192.168.1.181:2377
```

Tutorial adapted from:

<https://howchoo.com/g/njy4zdm3mwy/how-to-run-a-raspberry-pi-cluster-with-docker-swarm>

Docker Swarm

3. Log in to Node 2, run the command previously produced.

```
ssh pi@docker2
sudo docker swarm join --token SWMTKN-abc...manager...xyz 192.168.1.181:2377
```

Add a Worker

```
ssh pi@docker3
sudo docker swarm join --token SWMTKN-abc...worker...xyz 192.168.1.181:2377
```

Run the visualization tool

```
ssh pi@docker1
sudo docker service create \
  --name viz \
  --publish 8080:8080/tcp \
  --constraint node.role==manager \
  --mount
type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
  alexellis2/visualizer-arm:latest
```

Tutorial adapted from:

<https://howchoo.com/g/njy4zdm3mwy/how-to-run-a-raspberry-pi-cluster-with-docker-swarm>

Thank you