

docker

5 Day Training



Notes by
ROB

DECEMBER 13, 2019
HEWLETT PACKARD

Table of Contents

TABLE OF FIGURES	2
DOCKER	3
GENERAL NOTES	3
GOLDEN RULES OF DOCKER:.....	4
DOCKER OBJECTS	4
DOCKER IMAGES	4
<i>How Docker Images are made (Data layers)?.....</i>	<i>4</i>
USING DOCKER.....	5
<i>Containers</i>	<i>6</i>
<i>Copying container storage to path outside and vice-versa.....</i>	<i>7</i>
EXTERNAL STORAGE MOUNT	7
<i>Volume Mount</i>	<i>7</i>
<i>Bind Mount</i>	<i>7</i>
<i>tmpfs Mount</i>	<i>8</i>
CONTAINER NETWORKING	8
Port mapping	8
<i>To find one container from another using its name.....</i>	<i>8</i>
<i>Creating a new network.....</i>	<i>8</i>
<i>Create multiple containers on same network</i>	<i>9</i>
<i>Containers in one network to talk to container in another network.....</i>	<i>9</i>
ENVIRONMENT VARIABLES	9
TO SETUP PROXY AND CONNECT TO WIFI	9
DOCKER REALTIME APPLICATION EXAMPLE	9
DOCKER COMPOSE	10
YAML file	10
Steps to write yaml:	10
CREATE IMAGE FROM CONTAINERS	13
Options.....	13
BUILDING YOUR OWN IMAGES.....	14
<i>Let's build an image</i>	<i>14</i>
BUILDING IMAGE BY DOCKERFILE.....	14
<i>Running application in container example.....</i>	<i>15</i>
SHARING A DOCKER IMAGE	16
<i>Publishing an image in Dockerhub.....</i>	<i>16</i>
MULTI STAGE BUILDS	17
ORCHESTRATION	17
Docker swarm mode	17
Docker service	19
DOCKER SWARM EXAMPLE FOR POSTGRES.....	20
Second example	21
CGROUPS AND NAMESPACES	22
Exercise:	24
THROTTLING.....	25
FURTHER STEPS.....	26
USEFUL TOOLS TO USE WITH DOCKER	26

Table of Figures

Figure 1 Architecture of any application.....	4
Figure 2: Image layers	5
Figure 3: YAML file	12
Figure 4: docker-compose command	12
Figure 5: Dockerfile for application example.....	16
Figure 6: Docker Swarm (nodes).....	18
Figure 7: Docker swarm init command.....	18
Figure 8: Docker Swarm Example Screenshot1	20
Figure 9: Docker Swarm Example Screenshot2	21
Figure 10: Docker CGROUPS & NAMESPACES	23
Figure 11: Sharing PID Namespaces.....	24
Figure 12: Run container as non-root user	25
Figure 13: Traefik page1	27
Figure 14: Traefik 2	27

DOCKER

General Notes

- WHY? - Scaling larger application on same resource.
- In virtualization, hypervisors answers instead of BIOS to CPU.
- in virtualization, hyper V is the BIOS
- OVF- Open Virtualization File is standard of hyper V's
- OVA- Open Virtualization archive
- CGROUPS is used for keeping track of all the processes. It is used for accounting and throttling purposes.
- NAMESPACES – kernel maintains lists of things (processes, network interfaces etc.)
- A process inside a NAMESPACE can get info about things bounded only in that NAMESPACE and process which is not in any namespace can see all the processes.
- A container is an application isolated by OS. Each container has its own set of its NAMESPACES.
- LxC (Linux Container) library was created for creating and managing containers. It's a programmer's library.
- People started using toolkits (own libraries) for containerization.
- DotCloud company open sourced toolkit which became popular.
- They started new company called Docker. Separate toolkit for containerization.
- Docker and google started new toolkit 'Libcontainer' with additional features.
- OCI (Open Container Initiative) was started to standardize container by Docker.
- 'runC' library was the new standard library to be used.
- 'containerd' (pre-process stage) is used with 'runc' in the preparation for container to start the process.
- 'podman' is toolkit alternative to Docker to create and manage Container.
- Windows docker can be used from Windows 10 and Windows server 2019.
- Docker's main contribution:
 - Docker Image Format
 - Docker Registry Protocol – Registry is the location where images are stored.
 - Docker Engine (Docker Daemon/ Docker Service) – it runs on a host machine which talks through REST API.

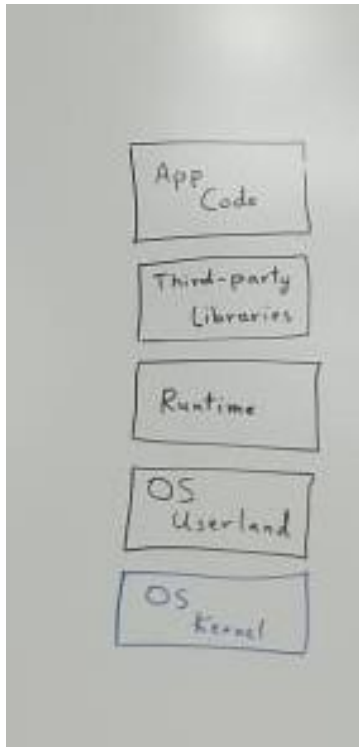


Figure 1 Architecture of any application

Golden Rules of Docker:

- Images are immutable
- Containers are disposable

Docker Objects

- Docker Image
- docker container
- docker volume
- docker networks

Docker Images

- Docker Image is the package using which containers are created.
- Docker Image has 2 things: Metadata and Data.

How Docker Images are made (Data layers)?

- It is made up of multiple layers
- Every layers have unique number called Hash (hexadecimal number Eg: 0xB01)

- Order of the layers → bottom to up.
- Images are union of multiple layers.

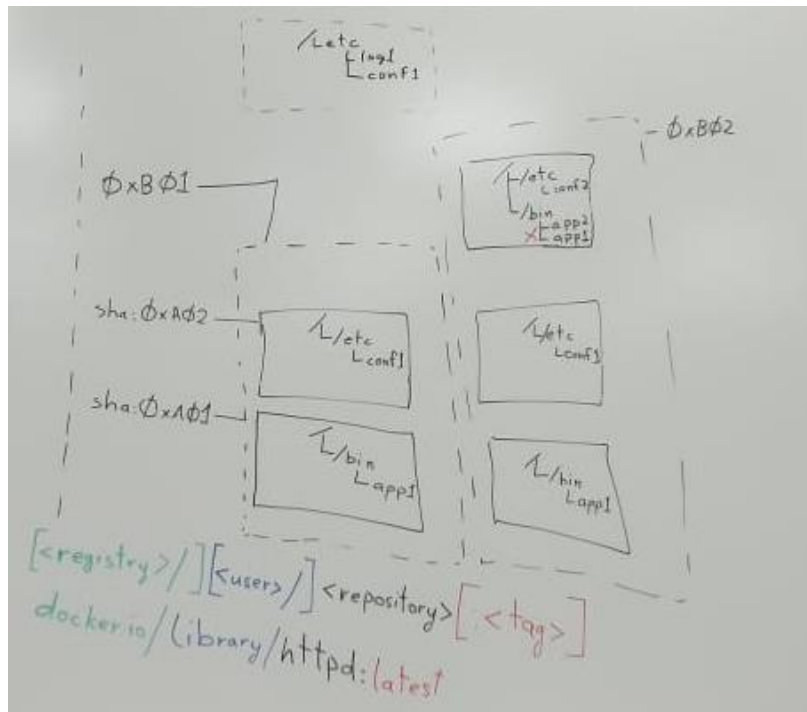


Figure 2: Image layers

- If some layers of image are already present then no need of downloading again.
- Each container will have only one layer for write option.
- If any edits are done to files, it gets added in write layer. It is called Copy On Write (COW).
- Images have names since hashes and ID's are not convenient.
- Image names in Docker has 4 parts:
 - Registry – [<registry>/]. It is optional. if not specified, default is dockerhub
 - Username – [<username>/]. optional. if not specified, default user
 - Repository – [<repository>/]. It is mandatory.
 - Tag – [:<tag>]. It is optional while writing. Default is 'latest'
 - i.e [<registry>/][<username>/][<repository>/][:<tag>]
 - Eg: docker.io/library/httpd:latest

Using Docker

Credentials – [user1,Pass@word1] //VM Credentials

- `docker <context> <command>` //syntax
- `docker version` //views the docker version
- `whomi` //to check the user
- user has to be a member of Docker group to talk to Daemon.
- `groups` //to view the groups user is a member of.
- if client are on different machine, it uses https. same machine – unix sockets.
- `docker system info` //To view info of system
- `docker image ls` //To view the images present
- `docker image inspect <imagename>` //To get more info of the image and it displays the metadata. under roots – layers --tells about data layers
- `docker image history <imagename>` //to get metadata and data layers. OB→metadata other is data layer.
- `docker system df` //to check the space df→diskfree
- Every image has command property. It can be checked with 'inspect' command in cmd.
- When we create a container, it just adds a metadata.

Containers

- `docker container create <image-name>` //To create a container out of a image
- `docker container ls -a` //To list all the container
- `docker container start <containername>` //to start the container
- `docker container top <containername>` //to see the process running by container
- `-t`→ is used for the process with terminal
- `-i`→ is used for container to be interactive
- `docker container attach <containername>` //to connect container to terminal of PID 1
- `ctrl p + ctrl q` //to detach the terminal
- `kill -9 <PID>` //kill process
- cannot kill PID 1 inside container.
- Responsibilities of PID 1:
 - Lifetime of PID 1 is the lifetime of container.
 - PID 1 is also responsible to produce logs
 - graceful shutdown of container.
- `docker container logs c1` //displays all commands run
- `docker container create <image-name> <arg>` //create container with command, this will override the image cmd.
- if container is stopped, it can be renamed.
- `docker container rename <oldname> <newname>` //change name
- `docker container exec <containername> <command>` //if its not interactive it run the new process and show the output, if its interactive it starts immediately and shows the output.
- `docker container stop <containername>`
- `docker container --restart <containername>` //restart container
- `docker container --restart always` // restart container always [stops and starts when machine reboots]
- `docker container --restart on-failure` // restart container when container crashes

- `docker container --restart unless-stopped` // restart container unless stopped manually
- `docker container diff <container-name>` //shows the changes done and path directory, D -deleted, A- added, C- change directory
- `docker container rm <container-name>` // Delete container. Use -f to delete forcefully.

Copying container storage to path outside and vice-versa

- `docker container cp <host-path> <container-name>:<container-path>`. //Copies from hostpath to container path.
- `docker container cp <container-name>:<container-path> <host-path>`. // Copies from hostpath to container path

External Storage Mount

- Data can be stored outside the container which is collectively known as Mounting.
 - Volume Mounts
 - Bind mounts
 - tmpfs mounts
- local volume drivers use docker hosts storage.

Volume Mount

- we use mount path in container, anything writing or reading in mount path is not included in union of layers.
- `docker volume ls` //to view volume present
- `docker container create --name <name> -v /vols/vol1 <image-name>` //create a container with volume
- `docker container create --name <name> --mount type=volume,target=/vols/vol1` //create a container with volume another way
- without name, volume created is anonymous
- `docker container create --name <name> --mount type=volume, target=<path inside container where stored is stored in xternal volume>,source= <name of external volume>`
- volume is not deleted when container is stopped.
- while deleting a container, using -volume and deletes only anonymous volumes.

`docker container create --name <name> --mount type=volume,target=/vols/vol1,readonly` //for containers to be readonly

Bind Mount

- Mount path is not managed by docker engine.
- a path on the host to path on the container.
- `docker container create --name <name> --mount type=bind,target=/vols/vol1` //creates container with bind mount
- we can give source path in bind mount but volume mount cannot.

- cannot unmount from container.

tmpfs Mount

- It is not available in windows.
- it does not support `-v` syntax.
- `--mount type=tmpfs,target=/vols/vol1` //Syntax – no source
- temporary file system.
- it gets deleted when container stops.
- it stores in RAM temporarily.

`docker run <container-name> <image-name>` //Create, start and attach the container. use `-d` to don't attach.

Container Networking

- `docker network ls` //to view networks
- bridge n/w is the default.
- The purpose of bridge is container to container communication in same host, container to outside communication
- `docker network inspect bridge` // to check the IP given to container
- dns of host is copied by docker. i.e resolv.conf is same inside and outside container.
- using bridge driver, all requests from container are NATted from host
- for outside world to reach container → port match, port forward, port map

Port mapping

- `docker container --name <cont-name> -p <hostIP>:<hostPort>:<container port>` //publish a network for outside world to communicate. → port mapping
- each container has its own hostname.
- `hostname` //check the hostname
- add '`--hostname <name>`' // to give a hostname
- we can ping hostname

To find one container from another using its name

- `docker container create --name <new-container-name> --link <container-to-link>:<alias>` //to link one container to another
- EG: `docker container create --name n5 --link n2:ws`
- `ping n2`
- Container to be linked should be running before starting new container.
- communication is only one way. i.e new container → container linked

Creating a new network

- `docker network create <option> <name>`
- Different <option> are:
 - `--driver <name>`

- `--subnet <IPsubnet>`

Create multiple containers on same network

- `docker container create <containername> --network <networkname> <imagename> //create container connected to specific network`
- multiple containers created with same network can communicate with each other.

Containers in one network to talk to container in another network

- `docker network connect <networkname to be connected> <containername which needs to connect to network>`
- EG: `docker network connect pnet1 n1`
- Container with host network will have hosts network properties → no NAT, port mapping is required.

`ping <containername> , <hostname>, <network-alias>` → all are same

Environment variables

- They are defined per process.
- It is available only to that process.
- In unix, we use 'export' to setup and make env var global.
- We put env var in container meta data while its creation.
- `docker container create <Cname> -e <variable> <value> <image-name> //create a env var → variable=value. also --env`

To setup proxy and connect to wifi

- we need two level of setup.
 - `proxy.sh`
 - Daemon service
- `sudo vi /etc/profile.d/proxy.sh`
 - `export http_proxy=http://web-proxy.in.hpecorp.net:8080`
 - `export https_proxy=http://web-proxy.in.hpecorp.net:8080`
 - `export no_proxy=localhost, rajini,kamal`
- logout & login the VM
- `sudo vi /etc/init.d/docker`
 - `start_stop_daemon → inbetween last "" quotes write "--env http_proxy=http://web-proxy.in.hpecorp.net:8080"`

Docker Realtime application example

- in dockerhub search for specific image
- CSCC – Command Storage Connectivity Configuration → always see this whenever pulling an image.
- search `postgres:alpine`
- in terminal → `docker pull postgres:alpine`
- `docker container create -- name db1 --mount type=volume,target=/var/lib/postgresql/data,source=db1vol1 --network pnet1 --env POSTGRES_USER=admin --env POSTGRES_PASSWORD=123 postgres:alpine`

- In docker hub, search for adminer.
- `docker container create --name fe1 --network pnet1 -e ADMINER_DEFAULT_SERVER=db1 adminer` //create container of adminer(front end application) on same network pnet1 as db1
- `docker container start db1 fe1`
- login with credentials, create database
- Now we have front end and back end.
- If container dies, application running also stops. If we create new container for same application and login with same credentials, all changes (Databases creates) still exists.

Docker Compose

- A stack is collection of docker objects which allows to run an application.
- docker has a tool which is known as “**docker compose**”.
- It provides a manifest file which has all the command to create stack.
- Manifest and docker-compose is written in YAML format.

YAML file

- It defines all the object with its properties.
- Properties are in Key:value pairs.
- A yaml key can have a single value which is scalar values.
- While writing a scalar value there should atleast one space after colon[:].
- Scalar value has a datatype. if it is a string enclose within quotes.
- A key can have multiple values known as list/array.

key:

- value1

- value2

- value3 //two spaces – one space value

- subkey1: value1

- docker compose file should begin with version [we are using version 3.7]
- version “3.7”
- We can have empty line and use # for comments.

Steps to write yaml:

- make a directory
- yaml file name is docker-manifest.yaml
- Inside yaml file.

version: "3.7"

networks:

dbappnet:

volumes:

dbappvol:

services:

backend:

image: postgres:alpine

volumes:

-dbappvol:/var/lib/postgresql/data

networks:

- dbappnet

environment

- POSTGRES_USER=admin

- POSTGRES_PASSWORD=something

```

version: "3.7"

networks:
  dbappnet:

volumes:
  dbappvol:

services:
  backend:
    image: postgres:alpine
    volumes:
      - dbappvol:/var/lib/postgresql/data
    networks:
      - dbappnet
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=something

```

Figure 3: YAML file

```

docker-compose
[-f <stack file name>]

<command>
[<command arguments>]

```

Figure 4: docker-compose command

- `docker-compose [-f <stack file name>] <command> [<command arguments>]`
- `docker-compose -f stack.yaml up -d` //to run docker compose and create container.
- `docker-compose [-p <instance name>]` //to give instance name
- `docker-compose -f stack.yaml -p dbapp2 up -d` //create docker container of another instance

- `docker-compose -f stack.yaml ps` //list the container created inside the instance
- `docker-compose -f stack.yaml logs` //shows the logs of services
- to add the one more service, open stack.yaml
- under service, add frontend and its configuration.
- `docker-compose -f stack.yaml -p dbapp1 logs <servicename>`
- `docker-compose -f stack.yaml down` //stops and removes containers created
- docker labels are used to map objects of particular application.
- docker compose is used when we have multiple components in applications.
- `docker-compose -d` is used to not show logs.

Create image from containers

```
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

- `docker commit <ContainerID or containerName> <newImageName>` //create an image from container

Options

Name, shorthand	Default	Description
<code>--author , -a</code>		Author (e.g., "John Hannibal Smith hannibal@a-team.com ")
<code>--change , -c</code>		Apply Dockerfile instruction to the created image
<code>--message , -m</code>		Commit message
<code>--pause , -p</code>	true	Pause container during commit

```
EG: docker commit c3f279d17e0a testimage
```

```
docker commit --change='CMD ["apachectl", "-DFOREGROUND"]' -c "EXPOSE 80"
c3f279d17e0a testimage
```

Building your own images

- It is better to use already present images since it might be already present.
- Find an image which has much as layers required by our application.
- Never start from scratch.
- There are 5 steps in planning of building images:
 1. Identify your CMD (PID 1)
 2. Based on (1), list your dependencies
 3. Divide (2) into
 - A. Things provided by you
 - B. Things not provided by you
 4. Based on (3B), Choose a good base image
 5. Copy everything in (3a) to a directory
- supervisor d and tini. //for multiple process

Let's build an image

- we use alpine base image but does not include vim and curl application so they have to be installed.
- scripts will be in /etc/os-release
- for my image, it should have vim, curl and os-release.
- among 5 steps:
 1. /bin/sh
 2. alpine userland, vim, curl, /etc/os-release
 3. .
 - a. /etc/os-release
 - b. alpine userland, vim, curl
 4. alpine
- mkdir myalpine , cd myalpine
- echo myalpine-1.0 >os-release
- docker container create --name bc1 --ti alpine:latest
- docker container cp os-release bc1:/etc/ //copy os-release file to container
- container start --ia bc1
 - o inside container
 - o apk add vim
 - o apk add curl //alpine package installs vim & curl
 - o ctrlP+ctrlQ //to detach from container
- docker container bc1 myalpine:1.0 //creates an image from the container. All the changes done in container will be in image

Building Image by Dockerfile

- Dockerfile specifies the requirements of the image required
- Format:

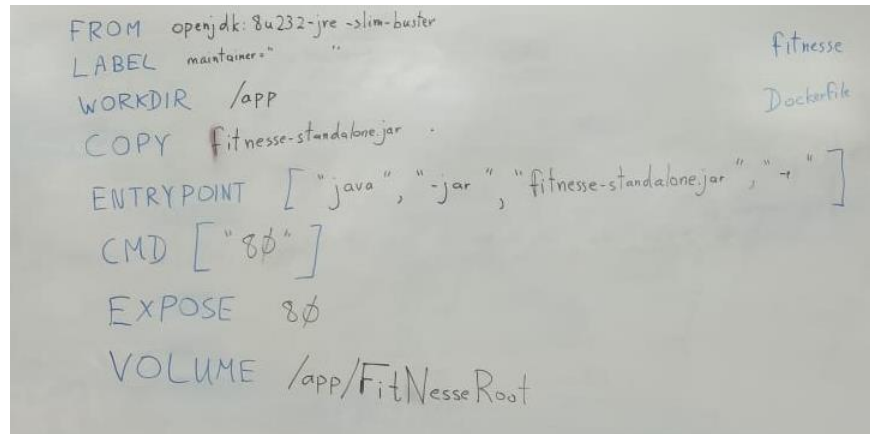
```
FROM <BASE-IMAGE-NAME>
LABEL <LABELS maintainer="<name> <email">
```

WORKDIR <working directory> //specifies the working directory for the container
 COPY <source—relative path to the build context> <destination-absolute_path>
 COPY os-release /etc/os-release //copies os-release to /etc/os-release
 RUN export http_proxy=http://web-proxy.in.hpecorp.net:8080 && \
 apk update && \
 apk add vim curl //they run only during build process
 && →used to combine run instructions
 \
 \ →used for instruction to continue in multiple line.
 CMD <command that begins when container start & starts PID1>
 CMD ["nginx", "-g", "daemon off"] //if command is big, we have to use square brackets and double quotes separated by comma.
 ENTRYPOINT ["nginx", "-g", "daemon off"] //same as cmd but it cannot be overridden when container is created.
 EXPOSE 8080 //expose port
 VOLUME <some-directory> //mounts to specified directory
 VOLUME /etc/share/data
 STOPSIGNAL SIGTERM //signal which sends when container stop.
 ADD <absolute-path or url> //similar to copy but can use URL links also and when file is tar, it gets untarred(unzipped)

- If there is both ENTRYPOINT and CMD, instruction in Entrypoint is concatenated with instruction in CMD. And if command is given while container creation it overrides CMD instruction concatenated with ENTRYPOINT instruction
- docker image build -f <Dockerfile-name> -t <new-image name> <parameter—i.e directory>
//image is built with Docker file

Running application in container example

- goto fitnesse.org and download //fitnesse is the application we will use.
- 5 Steps
 1. java -jar fitnesse-standalone.jar
 2. java 8, fitnesse-standalone.jar
 3. .
 - A. fitnesse-standalone.jar
 - B. java
 4. openjdk:8u232-jre-slim-buster //this is the image we use as base image
 5. copy fitnesse-standalone.jar to directory we create Dockerfile
- create directory fitnesse → create a Dockerfile
- vi Dockerfile



```

FROM openjdk:8u232-jre-slim-buster
LABEL maintainer=""
WORKDIR /app
COPY fitness-standalone.jar .
ENTRYPOINT ["java", "-jar", "fitness-standalone.jar", "-t"]
CMD ["80"]
EXPOSE 80
VOLUME /app/FitNesseRoot
  
```

Handwritten notes on the right side of the image: "fitness" and "Dockerfile".

Figure 5: Dockerfile for application example

- `docker image build --f Dockerfile -t myfitness:1.0 .` //create an image from the docker file.
- `docker container create --name testapp -p 9010:80 myfitness:1.0` //create a container "testapp" and port map hosts port 9010 to container's port 80 from image myfitness:1.0.
- Now docker container is running. Check the application in the browser
- In browser go to <http://localhost:9010> → fitness welcome page opens → it denotes container is working properly.
- Click on the edit option which opens an editable page.
- type "You can create a NewPage" → NewPage is wiki keyword which allows us to create a new page.
- click the plus symbol[+] → opens another editable page → type "This is a new page"

to use in house remove proxy.sh and docker.d file.

- `docker container create --name test3 -p 9010:8080 --mount type=volume,target=/app/FitNesseRoot/,source=ftvol myfitness:1.0 8080` // Creates a new container with volume mounted to the specified path by ftvol and exposing port 8080.
- goto browser localhost:9010 and do changes.
- create another container with same properties above. goto browser and changes is still existed.
- `docker image history myfitness:1.0` //shows all the layers of image.

Sharing a docker image

- Publish in dockerhub
- can save in host system
- `docker image save <image-name>` //saves image in tar format
- `docker image load <image-name.tar>` //loads image from host

Publishing an image in Dockerhub

- sign in to the profile.
- to publish to particular repository we should tag the image with username/repo:tag.
- In CLI, login to dockerhub
- `docker login <registry>` //login to registry.

- `docker login` //logins to dockerhub with username and password
- `docker image push <tagged-imagename>` //pushes the image to dockerhub

Multi Stage builds

- go to dockerhub and search voxel-dockerclient
- check the Dockerfile.
- There are multiple FROM →this is multi stage build.
- only final FROM steps will have layers in final image.
- if Dockerfile has multistage, while building image, use `--target <stage-name>` to create image out of only that stage.

Orchestration

- Orchestrator:
 - It is a software that manages large number (cluster) of services.
- It requires a manifest file to tell the details of different components of an application.
- It requires a physical hardware.
- Any kind of networks
- It needs an inventory which has requirements of server, storage etc.
- In a cluster, every node has an 'agent' whose one of the task is continuously push/tell the inventory.
- All the agents report to 'API' about the inventory.
- API stores the reports (inventories) to shared storage.
- Shared storage can be where API is place or outside the container or copy of shared storage is present on each node.
- User who wants the application to install has the manifest, manifest talks to API and check the status of nodes. API stores the manifest.
- Scheduler reads the shared storage and writes it decisions (which node to install which application) to shared storage.
- Agents continuously talk with shared storage and picks up the instructions from decisions of Scheduler and installs the application in the necessary node.
- Scheduler whenever wakes up, waits for all the reports of node. If node is down it doesn't give response to shared storage so Scheduler knows that node is down.
- Container in different nodes talk to eachother using **Overlay networking** software which is running on every node.
- There is a master node which runs extra application such as API, Scheduler and shared storage.
- Main orchestration softwares are **Docker swarm mode** and **Kubernetes**.

Docker swarm mode

- Play with docker enables to create 5 different machines and provide 4 hour session
- `labs.play-with-docker.com`
- login
- alt+enter to fullscreen
- A manager/master node has Scheduler, API etc.

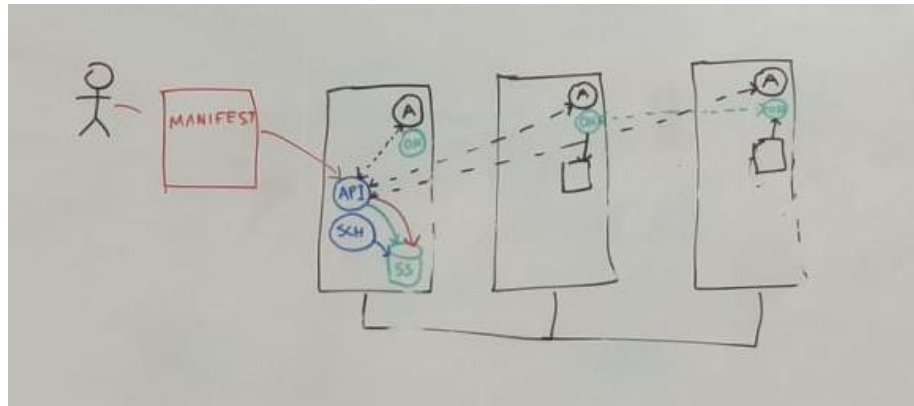


Figure 6: Docker Swarm (nodes)

- We can have multiple manager node.
- Split brain is when two manager gives different decisions.
- To eliminate the split brain condition, we should have odd number manager node.
- if manager node has multiple IPs, we have to use only one of them which is known as 'advertise'
- Click create two instances in left corner

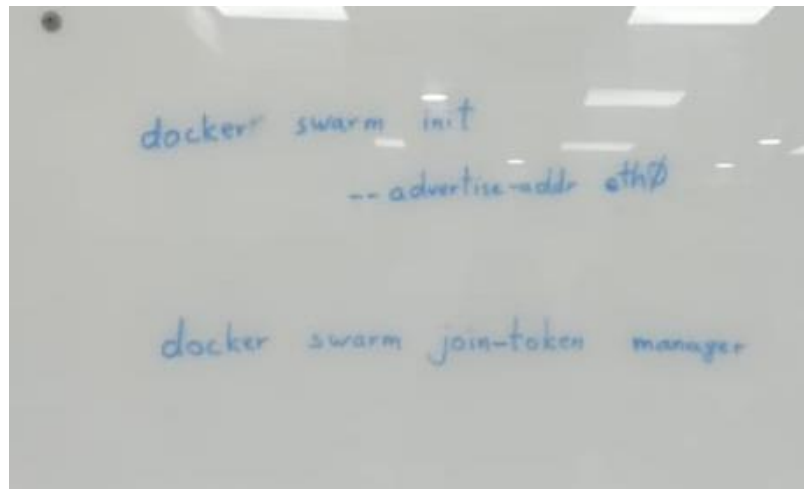


Figure 7: Docker swarm init command

- in first instance (node), type the following
- `docker swarm init --advertise-addr <ipaddress>` //initializes the cluster →provides shared storage.
- it gives the docker swarm join command, copy it and execute in another node
(`docker swarm join --token SWMTKN-1-0hflw3sgbf098quymjfj2h5w9ryihsub6euw5ilw1vxrt5js4z-5kaonoazxqfpl63mihxqokkcz192.168.0.13:2377`) //this makes node2 join the cluster as worker node
- A worker node cannot do operation which affects the entire cluster
- `docker network ls` //ingress the overlay network software.

- `docker swarm join-token worker` // it will give command for worker to join
- In docker, there is a new object called 'service'.

Docker service

- `docker service create <options> <image-name>`
`-d`
`-- name svc1` //name of the service
`--mode replicated` // replicate mode
`--replicas 1` // creates exact copy of service -- used for load balancing

replicas will be evenly distributed among nodes.

if `--mode global` //creates 1 replica in each node.

- This command stores service which will be stored in shared storage.
- `docker service create --name svc1 --mode replicated -d --replicas 1 nginx:alpine` //This is used for example
- `docker service ps svc1` //shows the containers created. it is a way to see what got done and where it gone done.
- `docker container rm -f <container-name>` //delete the container
- but docker immediately creates another container
- Therefore service is a contract, it always maintains the requirements. From service containers are created.
- `docker service update -d --image nginx:alpine svc1` //to update the service. here there's no image called nginx:alpine →therefore no replicas are created.. `docker service ps svc1`.
- Docker service can be updated except for mode.
- `docker service update -d --image chitradoc/ics --replicas 6 svc1` //update the service with new image.
- when we increase the replicas, one is already present remaining adds after the command. They get evenly distributed.
- Go to worker node → `docker swarm leave` //it leaves the swarm
- in manager node → `docker node ls` //worher node is down, its entries are not deleted
- `docker service create --name svc1 --mode replicated --replicas 1 -p 9000:8080 chitradoc/ics` //create new service
- when you publish (port map) to a service, it gets port is published to all the nodes.
- request to port 9000 will be load balanced by all the nodes → request can be given to any container.
- To have some control of where the service gets created we add "--constraint" property while updating.
- `docker service update -d --constraint node.id==<id>`
`node.hostname==<hostname>`
`node.labels.geography==asiapacific`
`node.labels.hpe.serverclass==ha`

- `docker service update -d --constraint-add node.labels.hpe.serverclass==ha svc1` //to give constraint/rule whose label hpe.serverclass's value is ha
- `docker node update --label-add hpe.serverclass=ha worker1` //adding label and giving the value ha
- in docker update, we can use `--env-add`, `--env-rm` {to add and delete the env var}, `--constraint-add`, `--constraint-rm`{add and delete constraints}

```

[manager3] (local) root@192.168.0.9 ~
$ docker service create --repl
--replicas
[manager3] (local) root@192.168.0.9 ~
$ docker service create --mode replicated --replicas 1 --name svc1 --network pnet1 nginx:alpine
ss2cdpxlpv6eudv6v6hb10uas
overall progress: 1 out of 1 tasks
I/1: running
verify: Service converged
[manager3] (local) root@192.168.0.9 ~
$ docker service ps
"docker service ps" requires at least 1 argument.
See 'docker service ps --help'.

Usage: docker service ps [OPTIONS] SERVICE [SERVICE...]

List the tasks of one or more services
[manager3] (local) root@192.168.0.9 ~
$ docker service ps svc1
ID                NAME                IMAGE                NODE                DESIRED STATE       CURRENT STATE       ERROR                PORTS
cq18fwtegtu      svc1.1              nginx:alpine         manager1            Running             Running 23 seconds ago
[manager3] (local) root@192.168.0.9 ~
$ docker service create --mode replicated --replicas 1 --name svc2 --constraint node.role==worker --network pnet1 nginx:alpine
j9tuwzix8dwl6jr01v59cil
overall progress: 1 out of 1 tasks
I/1: running
verify: Service converged
[manager3] (local) root@192.168.0.9 ~
$ docker service ls
ID                NAME                MODE                REPLICAS            IMAGE                PORTS
ss2cdpxlpv6e      svc1                replicated           1/1                 nginx:alpine
j9tuwzix8dwl      svc2                replicated           1/1                 nginx:alpine
[manager3] (local) root@192.168.0.9 ~
$ docker service ps svc2
docker: 'service' is not a docker command.
See 'docker --help'.
[manager3] (local) root@192.168.0.9 ~
$ docker service ps svc2
ID                NAME                IMAGE                NODE                DESIRED STATE       CURRENT STATE       ERROR                PORTS
6vvj5733626l      svc2.1              nginx:alpine         worker2            Running             Running about a minute ago
[manager3] (local) root@192.168.0.9 ~
$
[manager3] (local) root@192.168.0.9 ~

```

Figure 8: Docker Swarm Example Screenshot1

Docker Swarm example for postgres

- Docker template 3 managers and 2 workers.
- `docker service create --name frontend -d --network pnet1 -p 9000:8080 -e ADMINER_DEFAULT_SERVER=backend adminer` //no network pnet1, so service wont be created.
- `docker network create --driver overlay pnet1` //create network pnet1
- `docker service create --name frontend -d --network pnet1 -p 9000:8080 -e ADMINER_DEFAULT_SERVER=backend adminer` // now service is created
- port 900 is hyperlinked at top of the page.
- `docker service create --name backend -d --mount type=volume,target=/var/lib/postgresql/data,source=backendvol --network pnet1 --env POSTGRES_USER=admin --env POSTGRES_PASSWORD=something --constraint node.labels.hpe.nodetask==database postgres:alpine` //create a backend service for postgres application

```

docker-compose      docker-entrypoint.sh      docker-proxy      dockerd
[manager2] (local) root@192.168.0.48 -
$ docker service ls
ID                NAME                MODE                REPLICAS                IMAGE                PORTS
[manager2] (local) root@192.168.0.48 -
$ docker network create --driver overlay pnet1
wqgysw6l42b0z75sojxzqnyf
[manager2] (local) root@192.168.0.48 -
$ docker service create --name frontend -d --network pnet1 -p 9000:8080 -e ADMINER_DEFAULT_SERVER=backend adminer
22c1bf8b9uywdbbzfet8g89c
[manager2] (local) root@192.168.0.48 -
$
[manager2] (local) root@192.168.0.48 -
docker serviceC
[manager2] (local) root@192.168.0.48 -
$ docker service create --name backend -d --mo
--mode --mount
[manager2] (local) root@192.168.0.48 -
$ docker service create --name backend -d --mount type=volume,target=/var/lib/postgresql/data,source=backendvol --network pnet1 --en
--endpoint-mode --entrypoint --env
[manager2] (local) root@192.168.0.48 -
--env-file
OR
"docker service create" requires at least 1 argument.
See 'docker service create --help'.
Usage: docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]

Create a new service
[manager2] (local) root@192.168.0.48 -
$ docker service create --name backend -d --mount type=volume,target=/var/lib/postgresql/data,source=backendvol --network pnet1 --env POSTGRES_USER=admin --env POSTGRES_PASSWORD=something --con
--config --constraint --container-label
[manager2] (local) root@192.168.0.48 -
$ docker service create --name backend -d --mount type=volume,target=/var/lib/postgresql/data,source=backendvol --network pnet1 --env POSTGRES_USER=admin --env POSTGRES_PASSWORD=something --constraint node.labels
.hpe.nodetask==database postgres:alpine
0pz77bsmxymripndq3cak94b
[manager2] (local) root@192.168.0.48 -
$ docker node update --label-add node.labels.hpe.nodetask=database wor
worker1 worker2
[manager2] (local) root@192.168.0.48 -
$ docker node update --label-add node.labels.hpe.nodetask=database worker1
worker1
[manager2] (local) root@192.168.0.48 -

```

Figure 9: Docker Swarm Example Screenshot2

- `docker node update --label-add hpe.nodetask=database worker2` //update the node by adding the label to worker 2
- Then you will be able to login by clicking the hyperlink

Second example

- Drag and drop the stack.yaml to playbook in browser.
- `docker-compose -d stack.yaml up` //using docker-compose to create an application
- edit stack.yaml → under backend add
 deploy:
 placement:
 constraints:
 - node.labels.hpe.nodetask==database

version: "3.7"

networks:

dbappnet:

volumes:

dbappvol:

services:

backend:

image: postgres:alpine

volumes:

- dbappvol:/var/lib/postgresql/data

networks:

- dbappnet

environment:

- POSTGRES_USER=admin

- POSTGRES_PASSWORD=something

deploy:

placement:

constraints:

- node.labels.hpe.nodetask==database

frontend:

image: adminer

networks:

- dbappnet

ports:

- 8080

environment:

- ADMINER_DEFAULT_SERVER=backend

- ADMINER_DESIGN=pappu687

- `docker stack deploy --compose-file stack.yaml dbapp` //docker stack deploys similar to `docker-compose up`
- `docker stack ps dbapp` //it shows the container which is running

CGROUPS and NAMESPACES

- Normally two containers have different Namespaces. They cannot see each other
- **cgroup**: Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.

- **namespace:** wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

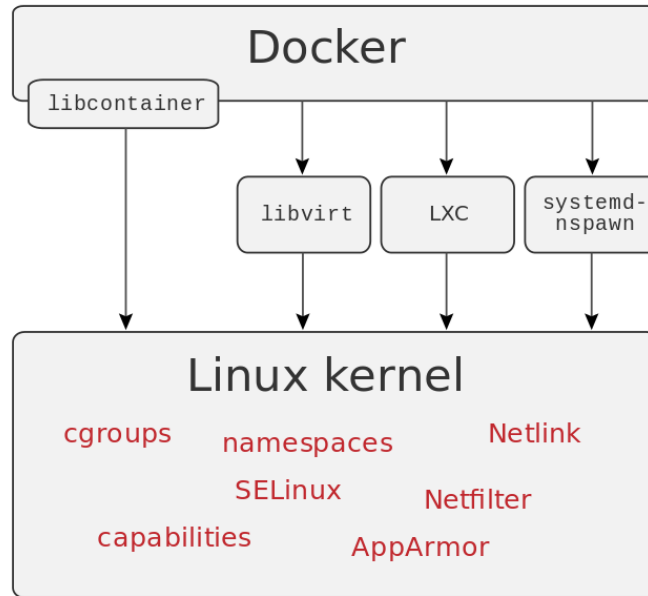


Figure 10: Docker CGROUPS & NAMESPACES

In short:

- **Cgroups** = limits how much you can use;
- **namespaces** = limits what you can see (and therefore use)

Docker Engine uses the following **namespaces** on Linux:

- **PID** namespace for process isolation.
- **NET** namespace for managing network interfaces.
- **IPC** namespace for managing access to IPC resources.
- **MNT** namespace for managing filesystem mount points.
- **UTS** namespace for isolating kernel and version identifiers.

- All namespaces can be shared but filesystem namespace cannot be shared.

Docker Engine uses the following cgroups:

- **Memory cgroup** for managing accounting, limits and notifications.
- **HugeTBL cgroup** for accounting usage of huge pages by process group.
- **CPU group** for managing user / system CPU time and usage.
- **CPUSet cgroup** for binding a group to specific CPU. Useful for real time applications and NUMA systems with localized memory per CPU.
- **BlkIO cgroup** for measuring & limiting amount of blkIO by group.
- **net_cls** and **net_prio cgroup** for tagging the traffic control.

- **Devices cgroup** for reading / writing access devices.
- **Freezer cgroup** for freezing a group. Useful for cluster batch scheduling, process migration and debugging without affecting prtrace.

Exercise:

```
matsya:~$ docker container create --name scooter nginx:alpine
affbabe91e472eda367e49d15de6e12eeb375362f5541594e21ea9d441d0c91e
matsya:~$ docker container start scooter
scooter
matsya:~$ docker container create --name sc1 -ti --pid container:scooter alpine:latest
beac33d6d9501a4b3e2f2dc0d715679e773eb8be94ff0beebe7395744d01701b
matsya:~$ docker container start sc1
sc1
matsya:~$ docker attach sc1
/ # ps aux
PID   USER     TIME   COMMAND
    1  root      0:00   nginx: master process nginx -g daemon off;
    8  101       0:00   nginx: worker process
    9  101       0:00   nginx: worker process
   10  root      0:00   /bin/sh
   15  root      0:00   ps aux
/ #
```

Figure 11: Sharing PID Namespaces

- Sharing the pid between the containers.
- we have to use `--pid container: <container-name>`
- When we stop, stop signal is sent to bin/sh even though it is not pid 1.
- container can share the namespace of different kind.

To share the network namespace between containers

- `docker container create --name sc2 -ti --network container:scooter alpine:latest //create container with IP address of scooter`
- `docker container attach sc2`
- `docker attach sc2 → ifconfig`
- IP addresses are same

Container to run other than root

- `cat /etc/passwd //it shows ther users in hosts`
- `docker container create --name sc4 --user user1 -ti alpine:latest // it will not start`
- `docker container create --name sc4 --user 1000:0 --mount type=bind,target=/data,source=/homw/user1/bindto -ti alpine:latest //it will start because containers recognize the user id and bind mount and it will start`

```

matsya:~/bindto$ cd ..
matsya:~$ docker container create --name sc5 -ti --mount type=bind,target=/data,source=/homw/user1/bindto --user 1000:0 alpine:latest
Error response from daemon: invalid mount config for type "bind": bind source path does not exist: /homw/user1/bindto
matsya:~$ docker container create --name sc5 -ti --mount type=bind,target=/data,source=/home/user1/bindto --user 1000:0 alpine:latest
1e059e744bcf03ad5f98cd93a813402eae46bded3caaa4747a58b38a1bdac52
matsya:~$ docker container start -ia sc5
/ $ ls
bin    data  dev   etc   home  lib   media mnt   opt   proc  root  run   sbin  srv   sys   tmp   usr   var
/ $ read escape sequence
matsya:~$ docker container top sc5
PID        USER      TIME      COMMAND
17956      user1     0:00      /bin/sh
matsya:~$ docker container start -ia sc5
ps aux
PID        USER      TIME      COMMAND
1 1000     0:00      /bin/sh
8 1000     0:00      ps aux
/ $ cd /data/
/data $ echo VIJAY>name.txt
/data $ ls
name.txt
/data $ ls -l
total 4
-rw-r--r-- 1 1000 1000 6 Dec 13 06:33 name.txt
/data $ read escape sequence
matsya:~$ ls
Desktop Downloads bindto dbapp1 fitness somedata
matsya:~$ cd bindto/
matsya:~/bindto$ ls
name.txt
matsya:~/bindto$ cat name.txt
VIJAY
matsya:~/bindto$ ls -l
total 4
-rw-r--r-- 1 user1 user1 6 Dec 13 12:03 name.txt
matsya:~/bindto$ █

```

Figure 12: Run container as non-root user

- we can change the user of container while creating
- `--user <username>` //This container will create but not start as the user is not present inside container/image
- `--user <id>` //This container will create and start, its permissions are same as permissions given to this user by host.
- It is better to give user-name as only user which is present in image will start the container.

Throttling

- We can control how much memory container can use.
- `docker container create --help` //it shows all the options, `--cpu-shares` & `--memory` to set the upper limit.
- Memory swap is the total memory (Physical + swap/virtual).
- `--memory-swap` //to control the swap memory.
- This is known Memory level throttling

exercise

- `docker pull polinux/stress`
- `docker container create --name s1 polinux/stress stress -v --vm 5 --vm-bytes 100M` //Start 5 processes of 100Mb ... allocate de allocate 5 times
- `docker container start s1` //start the container
- `docker container top s1` //shows the process running s1

- `docker container stats s1` //shows the stats of s1 container.. We didn't limit the memory. it uses the entire memory.
- `docker container logs s1` //shows the logs of the processes running inside the container
- `docker container create --name s2 --memory 400M --memory-swap 450M polinux/stress stress -v --vm 5 --vm-bytes 100M` //we provide upper limit
- `docker container start s2` //start the container
- `docker container ls` //container will be running
- `docker container ls -a`
- `docker container logs s2`
- `docker container stats s2`
- `docker container update --memory-swap 600M s2`
- `docker container start s2`
- `docker container ls`
- `docker container stats s2`

Throttling allows us to limit the memory used by the container.

Further steps

- Go through the docker documentation. <http://docs.docker.com>
- Guides section provide concept level understanding.
- references will give quick reference

Useful Tools to use with docker

- When there is a lot of application and we don't port forward every application, then we use reverse proxy.
- In computer networks, a reverse proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the proxy server itself.
- **Traefik:**
Traefik is the leading open source reverse proxy and load balancer for HTTP and TCP-based applications that is easy, dynamic, automatic, fast, full-featured, production proven, provides metrics, and integrates with every major cluster technology.
- Search Traefik in dockerhub
- `docker pull traefik:v1.7-alpine` //pull alpine flavor of traefik
- when containers and host are in same machine, we use unix socket
- so bind mount unix socket while creating the container.
- `docker container create --name tf1 --publish 80:80 --publish 9010:8080 --mount type=bind,source=/var/run/docker.sock,target=/var/run/docker.sock traefik:v1.7-alpine --docker --api` //traefik reverse proxy
- `docker container start tf1`
- goto browser in vm localhost and localhost:9010
- `docker container create --name ws1 --label traefik.frontend.rule=PathPrefixStrip:/app1 --label traefik.port=80 nginx:alpine`
- `docker container start ws1`

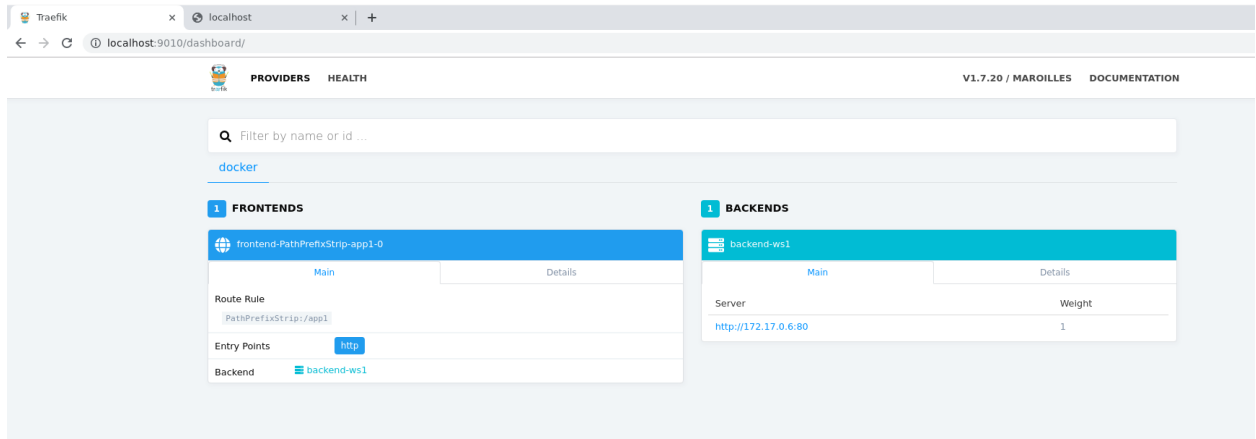


Figure 13: Traefik page1

- docker container create --name ws2 --label traefik.frontend.rule=PathPrefixStrip:/app2 --label traefik.port=8080 chitradoc/ics //creating a backend application
- goto browser localhost

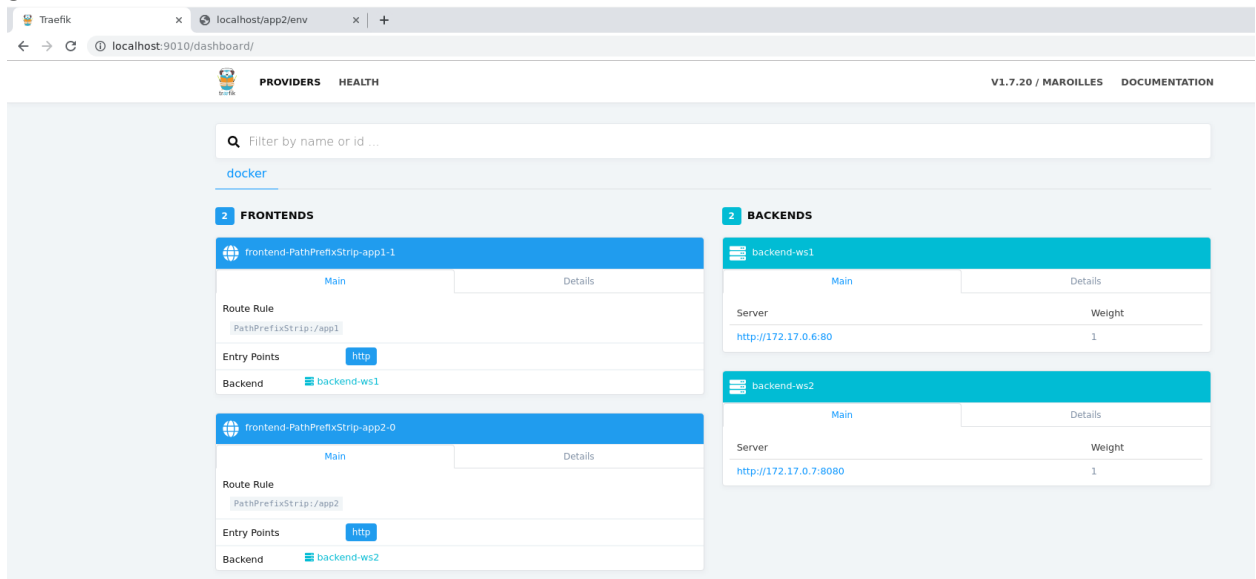


Figure 14: Traefik 2

- Traefik can run on docker swarm also.
- docker container create --name pt1 -p 9030:9000 --mount type=bind,source=/var/run/docker.sock,target=/var/run/docker.sock portainer/portainer
- goto browser localhost:9030
- **Portainer** is a lightweight management UI which allows you to easily manage your **Docker** host or Swarm cluster. **Portainer** is meant to be as simple to deploy as it is to use. It consists of a single container that can run on any **Docker** engine (**Docker** for Linux and **Docker** for Windows are supported)

- Portainer is GUI used to manage docker.
- In portainer, it talks to docker api → it runs as root

----- The End -----