STEFANO MARIANI, UNIMORE

DOCKER FOR DEVS

ABSTRACT

- Docker is a platform for *developers* and sysadmins to build, share, and run applications
- The talk adopts *devs standpoint*:

"how can Docker improve my workflow?"

- We will discuss about development *in* Docker containers, deploying applications with Docker compose, and orchestrating services with Docker Swarm
- Goal is not to be exhaustive, but to understand if useful and for what

OUTLINE

- Docker overview
- Docker basics (plus extras)
- Docker-compose
- Docker Swarm
- Docker in Bitbucket, Gitlab, …

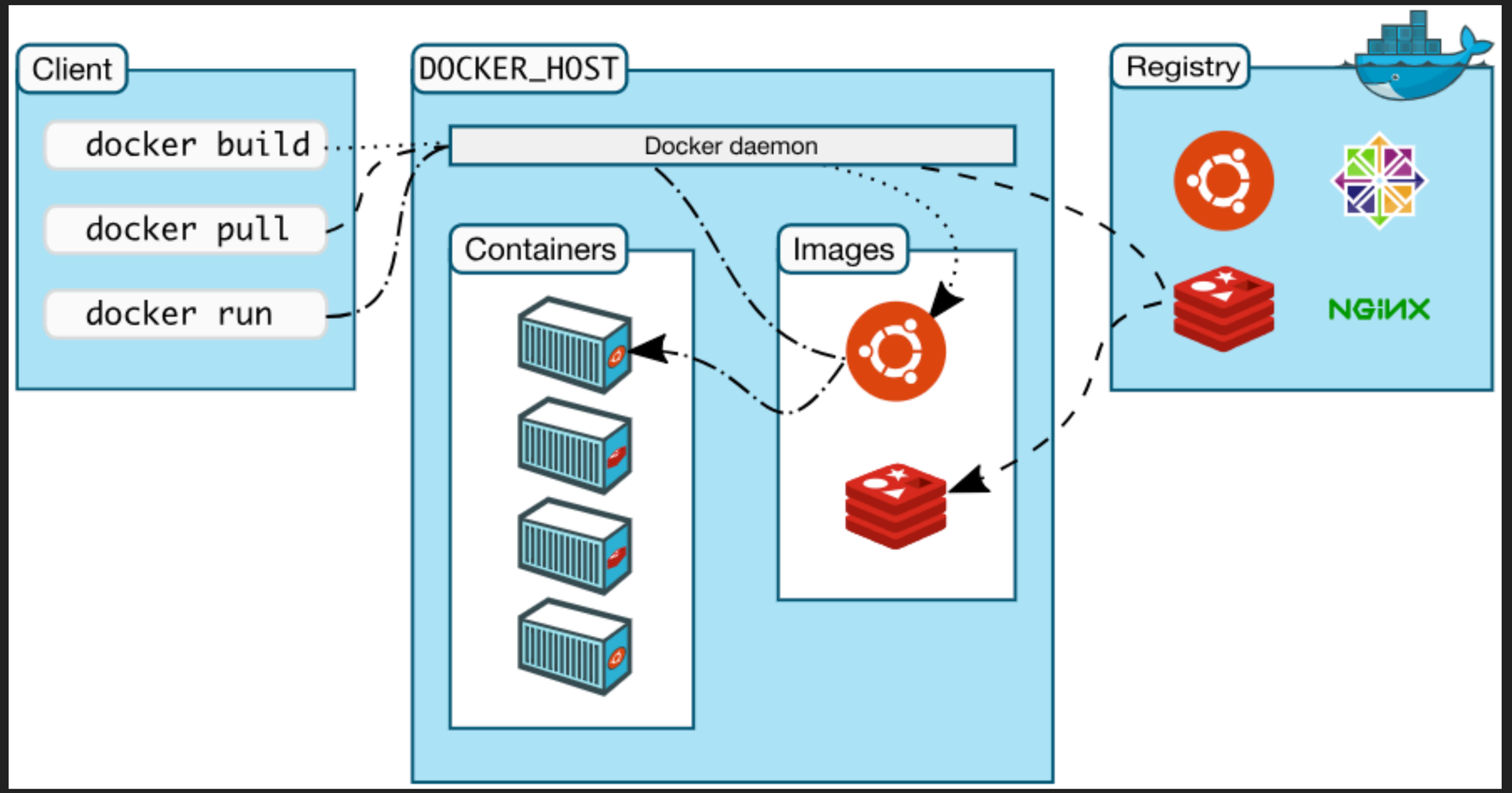
CONTAINERS

- Flexible: virtually any application can be containerised
- Lightweight: shared host kernel => more efficient than VMs
- Portable: e.g. build locally, deploy to the cloud
- Loosely coupled: encapsulation => replace / upgrade with no disruption to others (e.g. micro-services context)
- Scalable: add / remove / distribute app replicas
- Secure: isolation (e.g. private file system)

USE CASES

- Standardise dev environment (turn Docker containers into “development machines”)
  - no longer daunting set up (e.g. web service => language, libs, DB client, …)
  - 1-click setup (pull & run container)
  - no longer cross-platform compatibility issues
  - local, independent integration testing
- Support CI/CD pipeline
- Support run-time orchestration

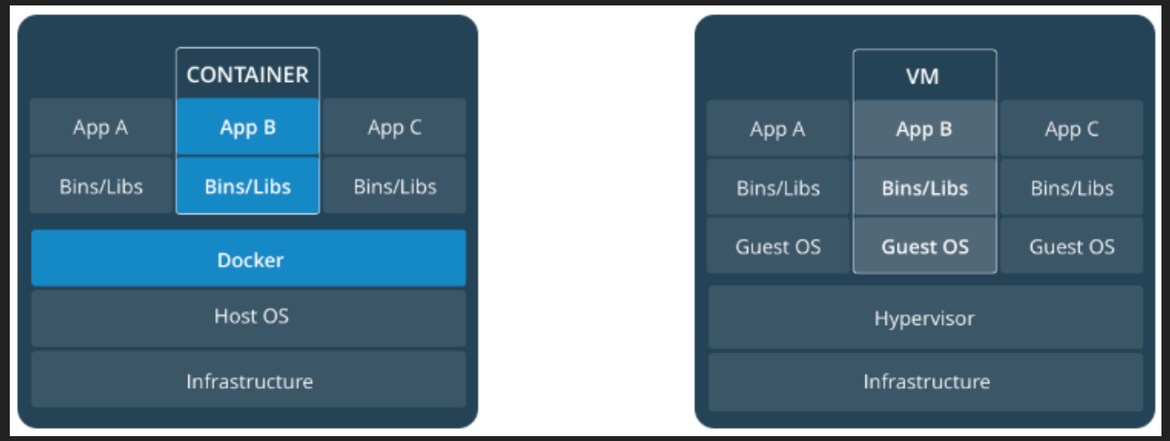
ARCHITECTURE



DOCKER BASICS

CORE CONCEPTS (1/2)

- **image** = *read-only* template with instructions for creating a Docker container (~ application environment = code + runtimes + deps + file system)
- **container** = runnable instance of an image (~ running process + encapsulation + configuration)



CORE CONCEPTS (2/2)

- **network**: pluggable networking subsystem, using drivers
  - bridge ~ containers on same host
  - host: directly use host network (remove isolation)
  - overlay ~ containers on different hosts, Swarm mode
- **data persistence**: containers have writable layer, *no persistence*
  - volumes: managed by Docker, preserve isolation, shareable (best choice in general)
  - bind mounts: bound to host dir structure, no isolation (to share source code)

TYPICAL WORKFLOW

1. Set up dev env
2. Build (& push) Docker image
3. Run image
   - singleton container
   - Swarm mode (later)

SET UP DEV ENV: SIMPLEST CASE

- Goal: dev Node.js app

1. install Node.js + npm locally
2. create Node.js project
3. write .dockerignore
4. write Dockerfile

2

```
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug

7 directories, 9 files
```

3

```
.git
.gitignore
node_modules
npm-debug.log
Dockerfile*
docker-compose*
README.md
LICENSE
.vscode
```

4

```
FROM node:10-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

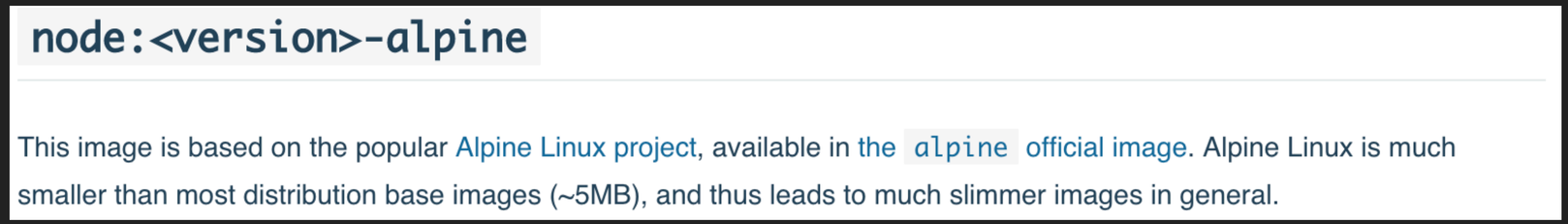
- **base image**
- in-container working dir
- host ==> container
- run command in container
- (copy Node.js project files)
- make port 3000 reachable
- tell Docker how to run app

BUILD DOCKER IMAGE

- (`pwd` = Dockerfile dir)
- `$> docker build -t [image_name].` (do not forget the dot!)
- `$> docker images` (to get image ID)
- `$> docker tag [image_id] [repo]/[image_name]:[image_version]`
- (opt) `$> docker push [repo]/[image_name]:[image_version]`
- (opt) `$> docker pull [repo]/[image_name]:[image_version]`

DOCKER BASE IMAGE

- Statement “`FROM node:10-alpine`”
- Image to use as first layer of image file system
- e.g.



- you always start off from some base image, just look for the best one ;)
  - memory footprint, up-to-date tooling, etc.

RUN IMAGE: SINGLETON CONTAINER

- `$> docker run -p80:3000 [repo]/[image_name]:[image_version]` (-d flag to release stdin)
  - map host port 80 to container port 3000 (exposed in Dockerfile!)
- `$> docker ps` (to get container ID)
- `$> docker logs [container_id]` (to see logs, follow with -f flag)
- `$> docker stop [container_id]` (to stop running)

SET UP DEV ENV: LIVE-RELOAD (1/2)

- Goal: dev Python/Java app
  1. same as previous but…
  2. …need extra work
     - in Dockerfile

A

```
FROM python:3.6-slim

COPY . /code
WORKDIR /code

RUN pip install --no-cache-dir -r requirements.txt
#RUN python3 -m pip install --no-cache-dir -r requirements.txt
RUN pip install -e .

ENTRYPOINT python ./mypackage/run.py
```

B

```
FROM maven:3.5-jdk-8

COPY . /usr/src/app
WORKDIR /usr/src/app

RUN apt-get update && apt-get install entr -y

RUN mvn clean package --batch-mode
ENTRYPOINT java -jar target/docker-compose-java-example-1.0-SNAPSHOT.jar
```

     - in **docker-compose.yml** (next slide + next section)

SET UP DEV ENV: LIVE-RELOAD (2/2)

A

```
services:
  python:
    image: registry.gitlab.com/mycompany/myproject/python:dev
    volumes:
      - ./python/:/code
    entrypoint: watchmedo auto-restart --recursive --pattern="*.py" --directory="." python mypackage/run.py
```

- Python watchmedo [...] is file system watchdog: file change ==> do something
- UNIX entr tool same functionality, used to trigger Maven build
  - as Java is *compiled*, we need an extra step w.r.t. Python (*interpreted*)
- Try chaning source code while container is running... ;)

B

```
  java:
    image: registry.gitlab.com/mycompany/myproject/java:dev
    volumes:
      - ./java/:/usr/src/app
    entrypoint: sh -c 'find src/ | entr mvn clean compile exec:java --batch-mode --quiet'
```

- Maven compiler plugin
- Maven JAR plugin
- Exec Maven plugin

SET UP DEV ENV: IDE ISSUE (1/2)

- In principle, we could develop a Dockerized app *without* having dependencies on the host machine
  - as it will run in the container, we need dependencies in the container as well
- In practice, our IDE would complain and our dev experience would degrade
  - no auto-completion, no design-time assistance, …
- Solution: bi-directional file synch ~ bind-mounts + shell script
  - need **docker-compose.yml**, again :)

SET UP DEV ENV: IDE ISSUE (2/2)

1. Bind-mount host dir
2. Install deps in container dir
3. Copy over
   - auto-install deps (npm)
   - deps in container
   - deps synch'd to host :)

- Trick is: *install in dir out of mount point*

```
version: "3"

# Define all the containers.
services:
  # Frontend Container.
  frontend:
    build: ./app/frontend
    volumes:
      - ./app/frontend:/usr/src/app
    ports:
     - 3000:3000
    environment:
      NODE_ENV: development
    command: /usr/src/app/entrypoint.sh
```

```
FROM node:10

# Create and define the node_modules's cache directory.
RUN mkdir /usr/src/cache
WORKDIR /usr/src/cache

# Install the application's dependencies into the node_modules's cache directory.
COPY package.json ./
COPY package-lock.json ./
RUN npm install

# Create and define the application's working directory.
RUN mkdir /usr/src/app
WORKDIR /usr/src/app
```

```
#!/bin/bash

cp -r /usr/src/cache/node_modules/. /usr/src/app/node_modules/
exec npm start
```

DOCKER-COMPOSE

OVERVIEW

- Tool for defining and running multi-container apps
  - YAML to configure app services
  - 1 command to create and launch
- Common use cases:
  - create and destroy isolated testing environments
  - ensure test => prod migration consistency

TYPICAL WORKFLOW

1. Create Dockerfile
2. Define app services in `docker-compose.yml`
3. Build & run app
   - singleton container
   - Swarm mode (later)
4. Manage services

DOCKERFILE

- Nothing really special except for…
- *…environment variables*
  - set with ENV keyword
  - used by Flask in example

```
FROM python:3.7-alpine

WORKDIR /code

ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0

RUN apk add --no-cache gcc musl-dev linux-headers

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["flask", "run"]
```

COMPOSE FILE

- Docker-compose version (syntax, capabilities)
- Services definition:
  - name (“web”, “redis”)
  - `build`: image of service (to build / pull / use)
  - `ports`: networking (host : container mapping)
  - `volumes`: data persistence (host : container bind mount)
  - `environment`: env variables (in example, activates Flask live-reload)

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

BUILD & RUN

- (`pwd` = docker-compose.yml dir)
- `$> docker-compose up` to build / pull service images + start containers
- ...that's it :)

- . . .

- Yes, as simple as that: all configuration is in docker-compose.yml!

MANAGE SERVICES

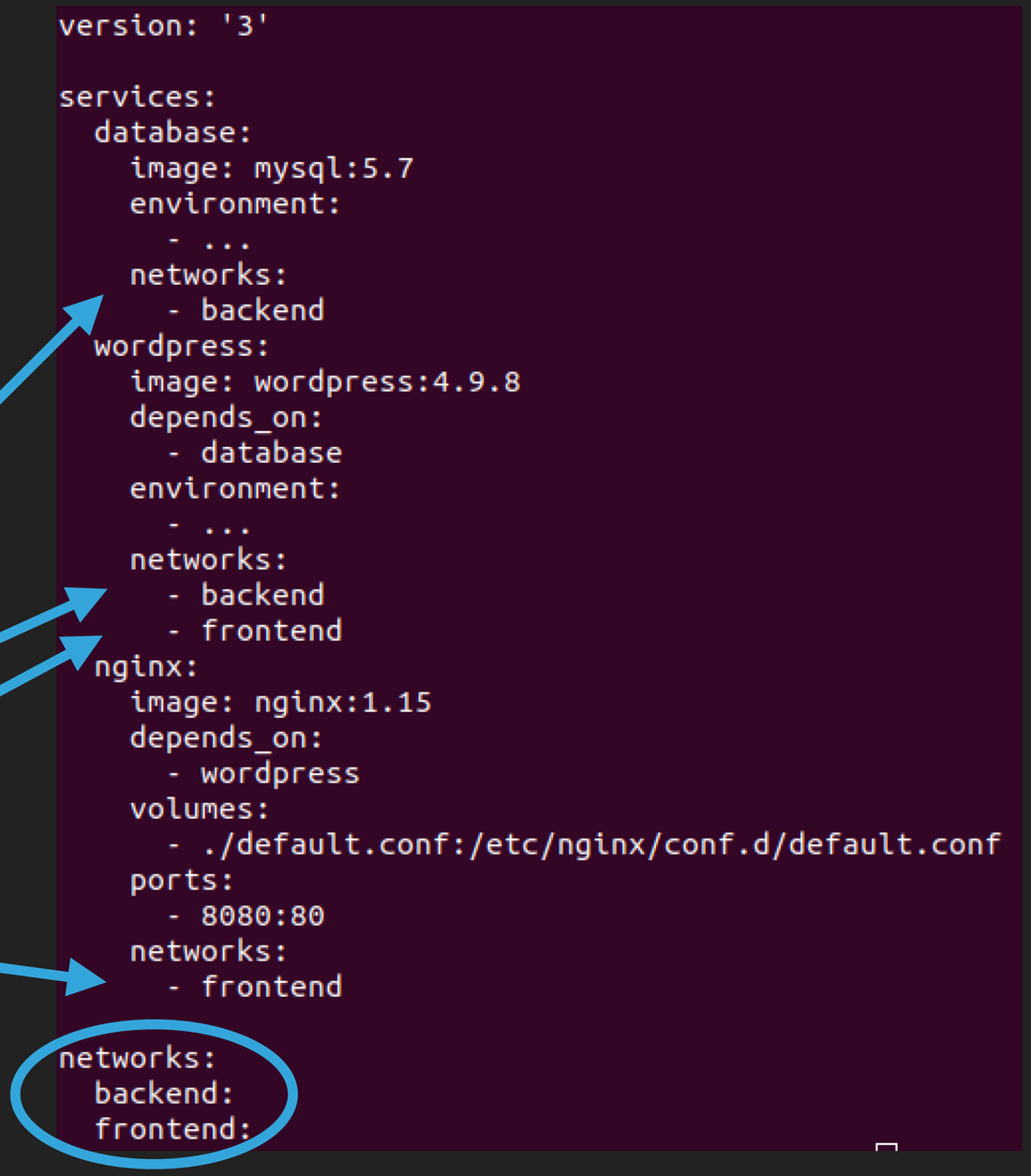
- `$> docker-compose up -d` for detached mode (release stdin)
- `$> docker-compose ps` to list running services
- `$> docker-compose stop` to stop app (crtl+c if not detached)
- `$> docker-compose down` to remove containers (not images)

EXTRAS

- Environment variables substitution:
  - Compose uses values from the shell environment in which docker-compose is run
  - can set default values for environment variables using a .env file (shell overrides)
- Same-host networking:
  - containers connected to the same bridge network automatically expose *all ports to each other*, and *no ports to the outside* world
  - (example in next slide)

EXTRAS: BRIDGE NETWORK

- Classic web app
  - frontend (nginx + wordpress)
  - backend (wordpress)
  - database (mysql)
- Two bridge networks:
  - backend: mysql – wordpress
  - frontend: nginx – wordpress
- nginx – mysql need not to know each other!



```
version: '3'

services:
  database:
    image: mysql:5.7
    environment:
      - ...
    networks:
      - backend
  wordpress:
    image: wordpress:4.9.8
    depends_on:
      - database
    environment:
      - ...
    networks:
      - backend
      - frontend
  nginx:
    image: nginx:1.15
    depends_on:
      - wordpress
    volumes:
      - ./default.conf:/etc/nginx/conf.d/default.conf
    ports:
      - 8080:80
    networks:
      - frontend

networks:
  backend:
  frontend:
```

TEST => PRODUCTION

- Modify docker-compose.yml
  - remove volume bindings for application code (no change from host)
  - update environment variables
  - specify a restart policy to avoid downtime
- Or better, override with prod.yml
  - `$> docker-compose -f docker-compose.yml -f prod.yml`
  - second file appended to first, overriding

SWARM MODE

OVERVIEW

- Cluster management integrated with Docker Engine
- Declarative service model to let devs define desired state of services in the application stack
- Auto-scaling / fault-tolerance by adding/removing tasks to match desired state
- Multi-host deployment through overlay networks
- Service discovery through embedded DNS service

KEY CONCEPTS

- **Swarm** = multiple Docker hosts running in swarm mode
  - managers = membership, task dispatching, load balancing, fault tolerance
  - workers = run swarm services
- **Service** = definition of tasks to execute on manager or workers
  - atomic scheduling unit of swarm
  - in docker-compose.yml define desired state (no. of replicas, network and storage resources, etc.)
- **Task** = container part of a swarm (vs. singleton container)
- **Node** = instance of the Docker engine participating in the swarm

TYPICAL WORKFLOW

- Create swarm on manager node
- Join worker nodes
- Deploy service / **stack** (later)
- Manage service / stack (monitor, scale, update)
- Remove service, shutdown swarm

CREATE SWARM

- `$> docker swarm init --advertise-addr [reachable_ip]`

```
Swarm initialized: current node (1o13vce2xrwo0c2dz87c1jv24) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4rv6qgx0psegyf3apb92sv92m94bkts6k324krgtoit96le8pw-ea8o3pbj2lh9m7gxtzk6vb2oz 192.168.227.129:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

- `$> docker info`

```
Swarm: active
 NodeID: 1o13vce2xrwo0c2dz87c1jv24
 Is Manager: true
 ClusterID: tjp40fggcbqs85iwjt7z2cm1q
 Managers: 1
 Nodes: 2
 Default Address Pool: 10.0.0.0/8
 SubnetSize: 24
 Data Path Port: 4789
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  Election Tick: 10
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
  Force Rotate: 0
 Autolock Managers: false
 Root Rotation In Progress: false
 Node Address: 192.168.227.129
 Manager Addresses:
  192.168.227.129:2377
```

JOIN NODE

- `$> sudo docker swarm join --token [token] [advertise-addr]`
- `$> docker info`

```
Swarm: active
 NodeID: ydq43qb655ywpqmo83frtd8i7
 Is Manager: false
 Node Address: 192.168.227.128
 Manager Addresses:
  192.168.227.129:2377
```

- `$> docker node ls` (on manager node!)

```
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
1o13vce2xrwo0c2dz87c1jv24 *   ubuntu              Ready               Active              Leader              19.03.2
ydq43qb655ywpqmo83frtd8i7     ubuntu              Ready               Active                                  19.03.2
```

DEPLOY TO SWARM

- From CLI
  - e.g. `$> docker service create --replicas 1 --name helloworld alpine ping docker.com`
  - e.g. `docker service create --replicas 3 --name redis --update-delay 10s redis:3.0.6`

```
oiv6112h8unhfgwbfvscni0qf
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
```

- With docker-compose.yml
  - (later)

MANAGE SWARM

- Monitor service `$> docker service ls`
  - check service details `$> docker service inspect --pretty [service-ID]`
  - check where service running `$> docker service ps [service-ID]`
- Scale service `$> docker service scale [service-ID]=[no_tasks]`
- Rolling update, e.g. `$> docker service update --image redis:3.0.7 redis`
  - check update `$> docker service ps [service-ID]`

```
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE                ERROR
                    PORTS
x58f73a0108r        redis.1             redis:3.0.7         ubuntu              Running             Running about a minute ago
e6kvkcdn3uvw         \_ redis.1         redis:3.0.6         ubuntu              Shutdown            Shutdown 2 minutes ago
e34h78zp7v2b        redis.2             redis:3.0.7         ubuntu              Running             Running about a minute ago
17vzenf2kjal         \_ redis.2         redis:3.0.6         ubuntu              Shutdown            Shutdown about a minute ago
wyzx7u42eghu        redis.3             redis:3.0.7         ubuntu              Running             Running about a minute ago
fqto15fbhrjl         \_ redis.3         redis:3.0.6         ubuntu              Shutdown            Shutdown about a minute ago
```

SHUTDOWN

- Remove service `$> docker service rm [service-ID]`
- Leave swarm `$> docker swarm leave (on worker)`
- Shutdown swarm `&> docker swarm leave --force` (on manager)

STACKS

- **Stack** = complete application stack, including multiple services
- When Docker in swarm mode, can use stacks to deploy apps with compose

```
FROM python:3.4-alpine
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

```
version: '3'

services:
  web:
    image: stackdemo
    build: .
    ports:
      - "8000:8000"
  redis:
    image: redis:alpine
```

- e.g. `$> docker stack deploy --compose-file docker-compose.yml stackdemo`

```
sm@ubuntu:~/Docker-playground/docker-stack$ sudo docker stack ls
NAME                SERVICES            ORCHESTRATOR
stackdemo           2                   Swarm
```

INTEGRATION

GITLAB, BITBUCKET, . . .

- Basically, regardless of provider, two ways to exploit Docker
  - use images as CI/CD environment

```
default:
  image: ruby:2.2

  services:
    - postgres:9.3

  before_script:
    - bundle install

test:
  script:
  - bundle exec rake spec
```

  - produce images as CI/CD output

```
before_script:
  - docker info

build_image:
  script:
    - docker build -t my-docker-image .
    - docker run my-docker-image /script/to/run/tests
```

THANKS
SORRY
NO QUESTIONS PLS

Stefano Mariani