Intro to Docker

Presented by Travis Holton

Administrivia

- Bathrooms
- Fire exits

This course

- Makes use of official Docker docs
- Based on latest Docker
- A mix of command line and theory
- Assumes no prior Docker knowledge
- Assumes familiarity with the linux shell
- Assumes we are using ubuntu 14.04 (trusty)

Aims

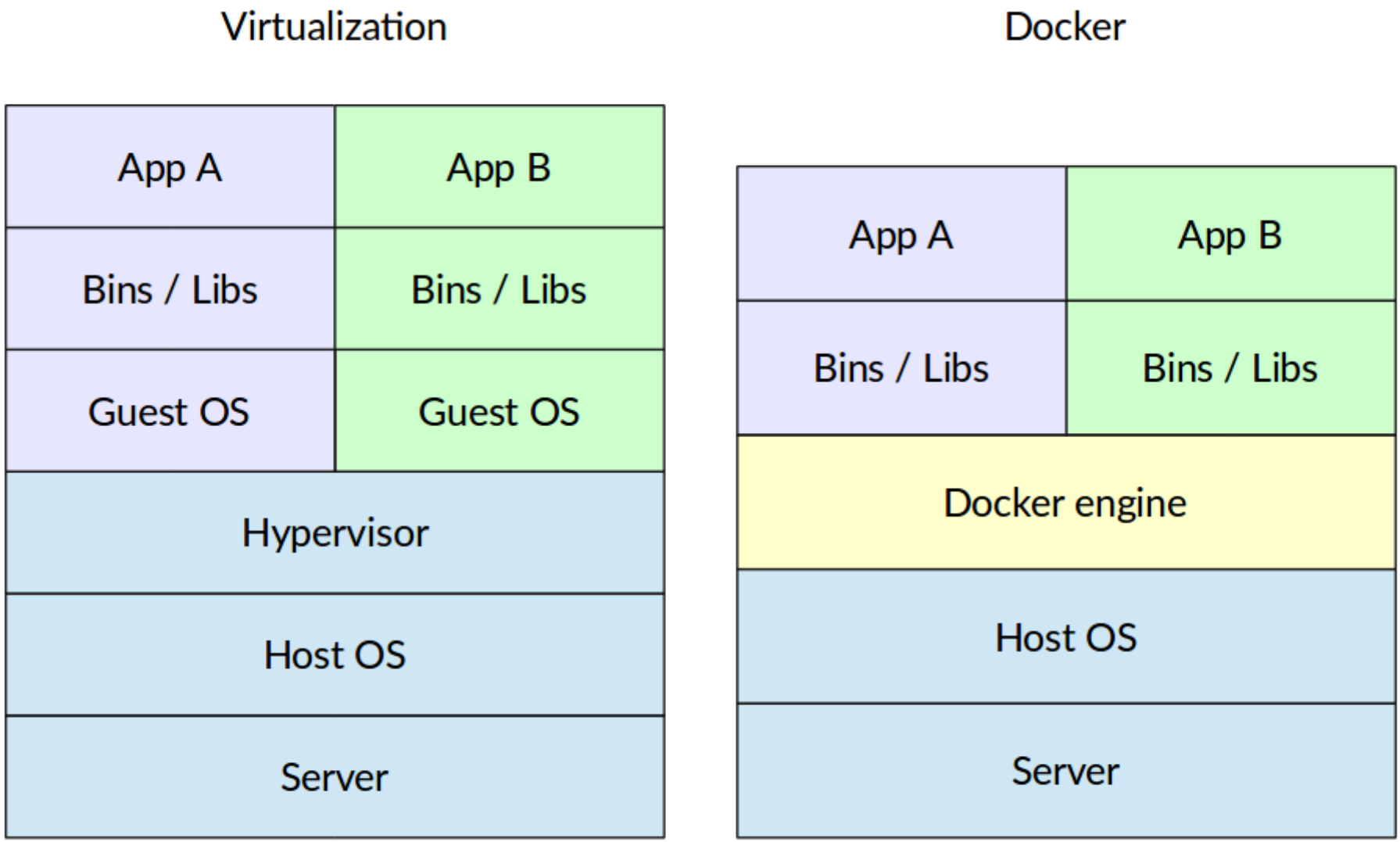
- Understand how to use Docker on the command line
- Understand how Docker works
- Learn how to integrate Docker with applications
- Learn ops and developers can use Docker to deploy applications
- Get people thinking about where they could use Docker

Introduction to containers

What is containerization?

- A type of virtualization
- Difference from traditional VMs
  - Don't replicate entire OS, just bits needed for application
  - Run natively on host
- Key benefits:
  - More lightweight than VMs
  - Efficiency gains in storage, CPU
  - Portability

Lightweight

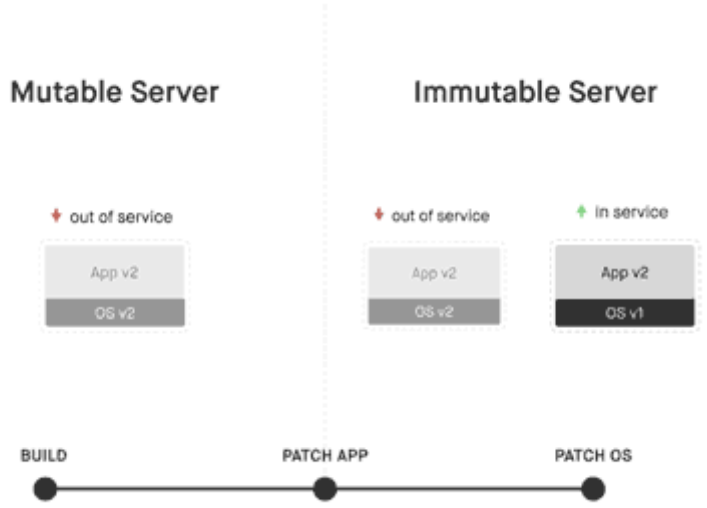


Benefits of Containers: Resources

- Containers share a kernel
- Use less CPU than VMs
- Less storage. Container image only contains:
  - executable
  - application dependencies

Benefits of Containers: Decoupling

- Application stack not coupled to host machine
- Scale and upgrade services independently
- Treat services like cattle instead of pets



Benefits of Containers: Workflows

- Easy to distribute
- Developers can wrap application with libs and dependencies as a single package
- Easy to move code from development environments to production in easy and replicable fashion

Introduction to Docker

The Docker Platform

What is Docker?

High level

An open-source platform for creating, running, and distributing software *containers* that bundle software applications with all of their dependencies.

Low level

A command-line tool for programmatically defining the contents of a Linux container in code, which can then be versioned, reproduced, shared, and modified easily just as if it were the source code to a program

Docker popularity

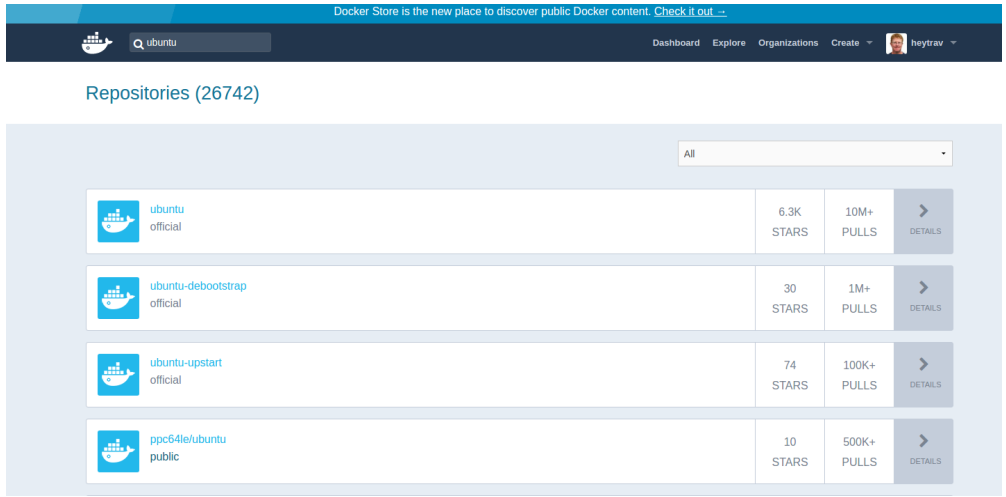
Docker workflow

Docker Portability

- Most modern operating systems
  - Linux (RHEL, CentOS, Ubuntu LTS, etc.)
  - OSX
  - Windows
- Lightweight Docker optimized Linux distributions (CoreOS, Project Atomic, RancherOS, etc.)
- Private clouds (OpenStack, Vmware)
- Public clouds (AWS, Azure, Rackspace, Google)

Docker Registries

- Public repositories for docker images
  - Docker Hub
  - Quay.io
  - GitLab ships with docker registry
- Create your own private registry docker/distribution



First Steps with Docker

Docker version

```
$ docker --version
Docker version 17.03.1-ce, build c6d412e
```

Current version scheme similar to Ubuntu versioning: `YY.MM.#`

Get command documentation

- Just typing `docker` returns list of commands
- Calling any command with `–h` flag displays some docs
- Comprehensive online docs on Docker website

Basic client usage

```
$ docker<ENTER>

    Usage:  docker COMMAND

    A self-sufficient runtime for containers

    Options:
          --config string      Location of client config files (default "/Users
      -D, --debug              Enable debug mode
          --help               Print usage
    .
    .
```

Exercise: Hello world

```
$ docker run hello-world
```

```
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/


   17s
```

Exercise: Pull and run course slides image

```
$ docker run --name docker-intro -d --rm \
        -p 8000:8000 heytrav/docker-introduction-slides:july-20
```

Follow along with course slides: http://localhost:8000

Exercise: Start a shell

```
$ docker run alpine /bin/sh
```

Exercise: Start an *interactive* shell

```
$ docker run -it alpine /bin/sh
```

```
➜  ~ docker run -
```

Exercise: Run detached container

```
$ docker run --rm --name static-site -e AUTHOR="YOUR NAME" \
    -d -p 8081:80 dockersamples/static-site
```

```
› docker run --rm --name static-site -e AUTHOR="YOUR NAME" \
    -d -p 8081:80 dockersamples/static-site
06ba2a841d43ad02a81e33f62561c87c5fb840aebcb0243e6b1a2c6d59a1e16d


› docker port static-site
docker80/tcp -> 0.0.0.0:8081
```

- `-d` creates container with process detached from terminal
- `-p` publish container with external port 8081 mapped to internal port 80
- `-e` pass environment variable into container
- `--name` the container "static-site"
- Go to http://localhost:8081 in your browser

List local images

```
$ docker image ls
```

```
➜  ~ docker image ls
REPOSITORY                              TAG                 IMAGE ID            CREATED
heytrav/docker-introduction-slides      latest              1cedfbdf2482        4 days ago
alpine                                  latest              4a415e366388        2 months ago
hello-world                             latest              48b5124b2768        3 months ago
➜  ~
```

List running containers

```
$ docker ps
```

```
➜  ~ docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED
25eff330a4e4        dockersamples/static-site            "/bin/sh -c 'cd /u..."   8 minutes ago
c5ddb8ebc26e        heytrav/docker-introduction-slides   "/usr/local/bin/du..."   4 hours ago
➜  ~
```

docker ps

```
Options:
-a, --all               Show all containers (default shows just running)
-f, --filter filter     Filter output based on conditions provided
    --format string     Pretty-print containers using a Go template
    --help              Print usage
-n, --last int          Show n last created containers (includes all states) (def
-l, --latest            Show the latest created container (includes all states)
    --no-trunc          Don't truncate output
-q, --quiet             Only display numeric IDs
-s, --size              Display total file sizes
```

More examples

`docker ps -a`

Show all containers (also not running)

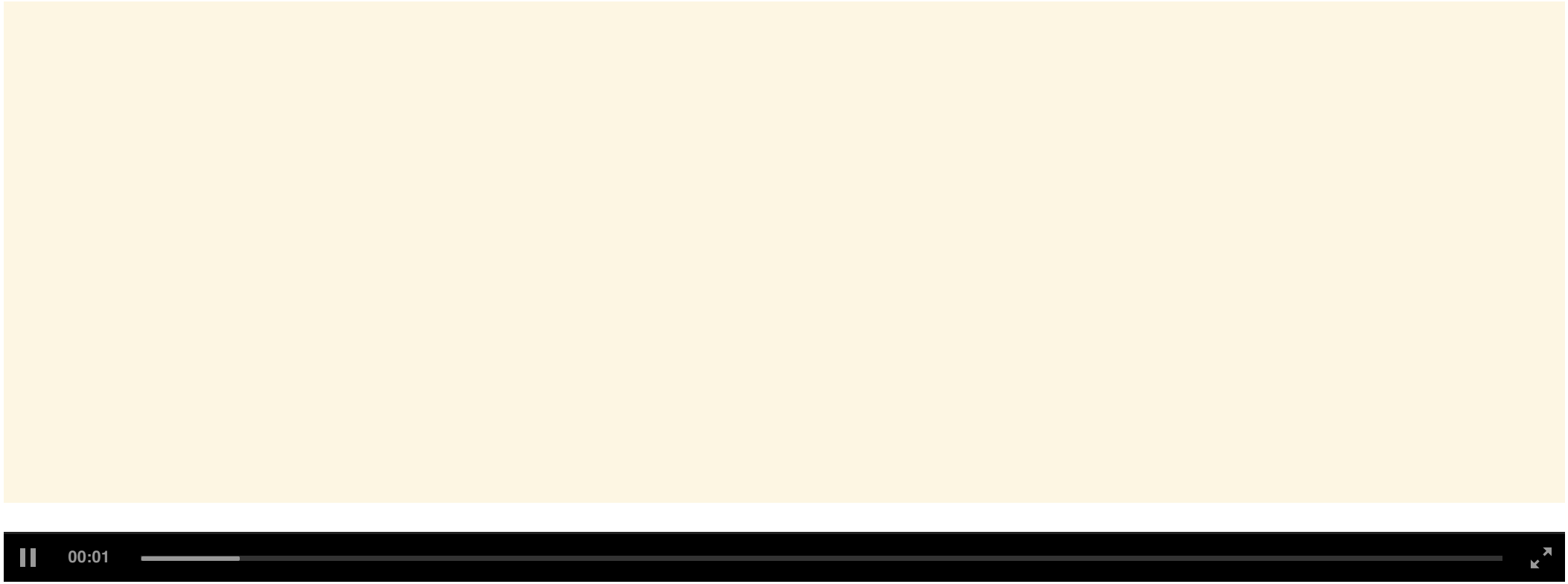
`docker ps -a --filter 'exited=0'`

Filter all containers by exit code

See online documentation

View container logs

```
$ docker logs
```



- `-f` flag to watch logs in realtime, like `tail -f file.log`

See online documentation

Enter a running container

```
$ docker exec OPTIONS <CONTAINER NAME>
```

```
› docker exec -it static-s
```

- Can be useful for debugging

See online documentation

Exercise: Stop a running container

```
$ docker stop <CONTAINER_ID>
```

```
➜  ~ docker stop 25eff330a4e4
```

Exercise: Clean up

```
$ docker stop $NAME
$ docker rm $NAME
```

```
➜  ~ docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED
S
d04e5d4049a4        dockersamples/static-site            "/bin/sh -c 'cd /u..."   13 seconds ago
ic-site
c5ddb8ebc26e        heytrav/docker-introduction-slides   "/usr/local/bin/du..."   5 hours ago
_jennings
➜  ~ docker stop static-site
static-site
➜  ~ docker rm sta
```

How Docker works

Components of Docker

Docker Image

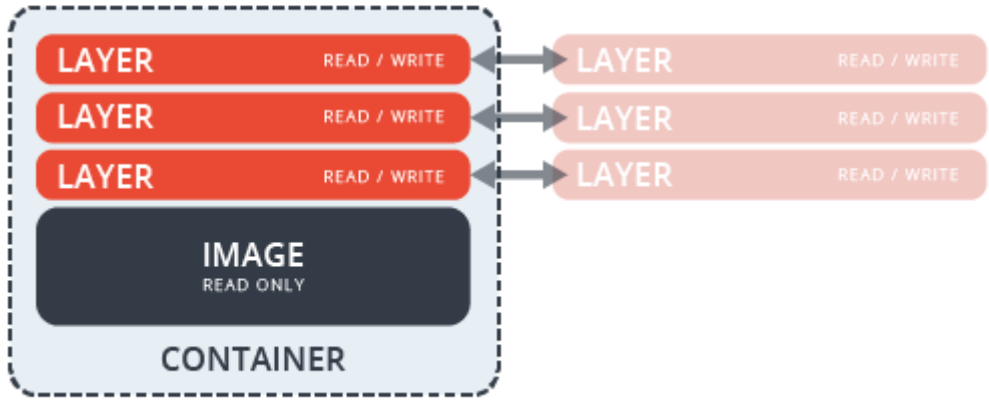
contains basic read-only image that forms the basis of container

Docker Registry

a repository of images which can be hosted publicly (like Docker Hub) or privately and behind a firewall

Docker Container

is comprised of a base image with layers that can be swapped out so it's not necessary to replace the entire VM when updating an application



Underlying technology

Go

Implementation language developed by Google

Namespaces

Provide isolated workspace, or *container*

cgroups

limit application to specific set of resources

UnionFS

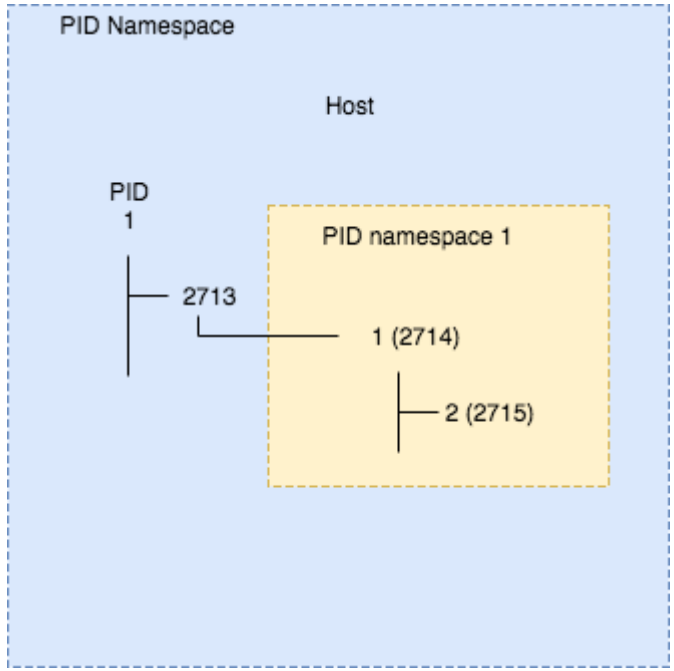
building blocks for containers

Container format

Combined namespaces, cgroups and UnionFS

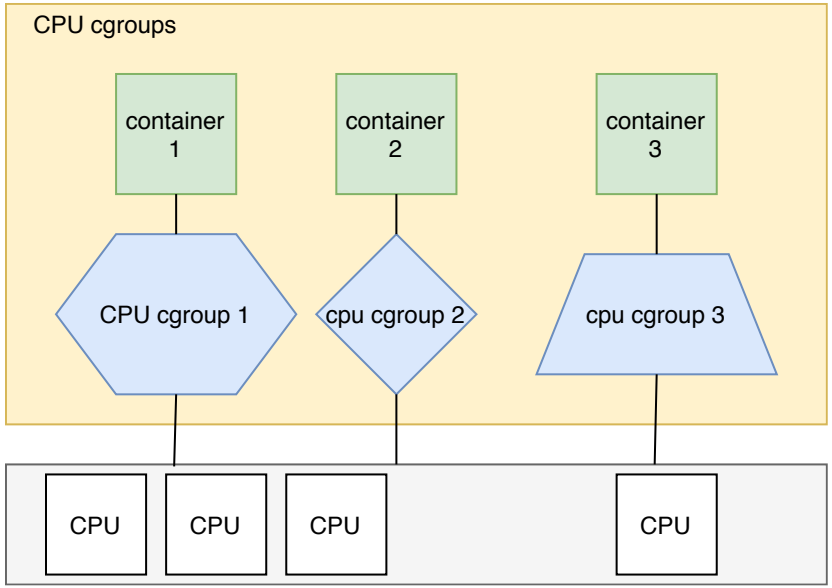
Namespaces

- Restrict visibility
- Processes inside a namespace should only see that namespace
- Namespaces:
  - pid
  - mnt
  - user
  - ipc



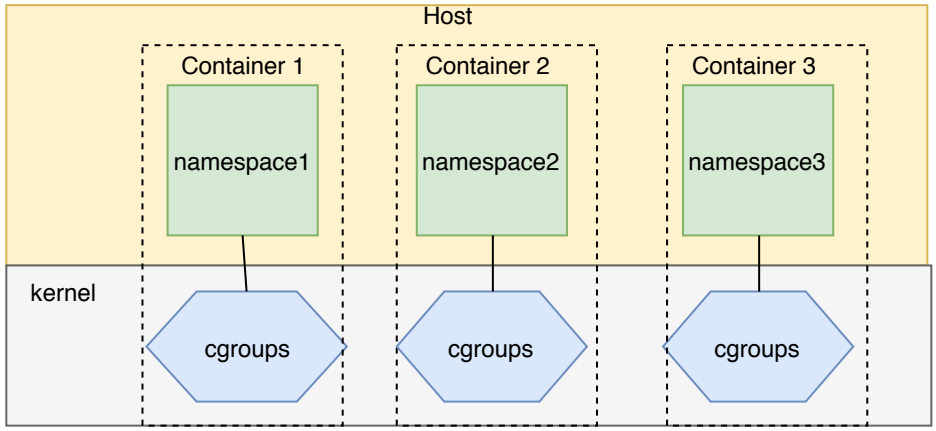
Cgroups

- Restrict usage
- Highly flexible; fine tuned
- Cgroups:
  - cpu
  - memory
  - devices
  - pids



Combining the two

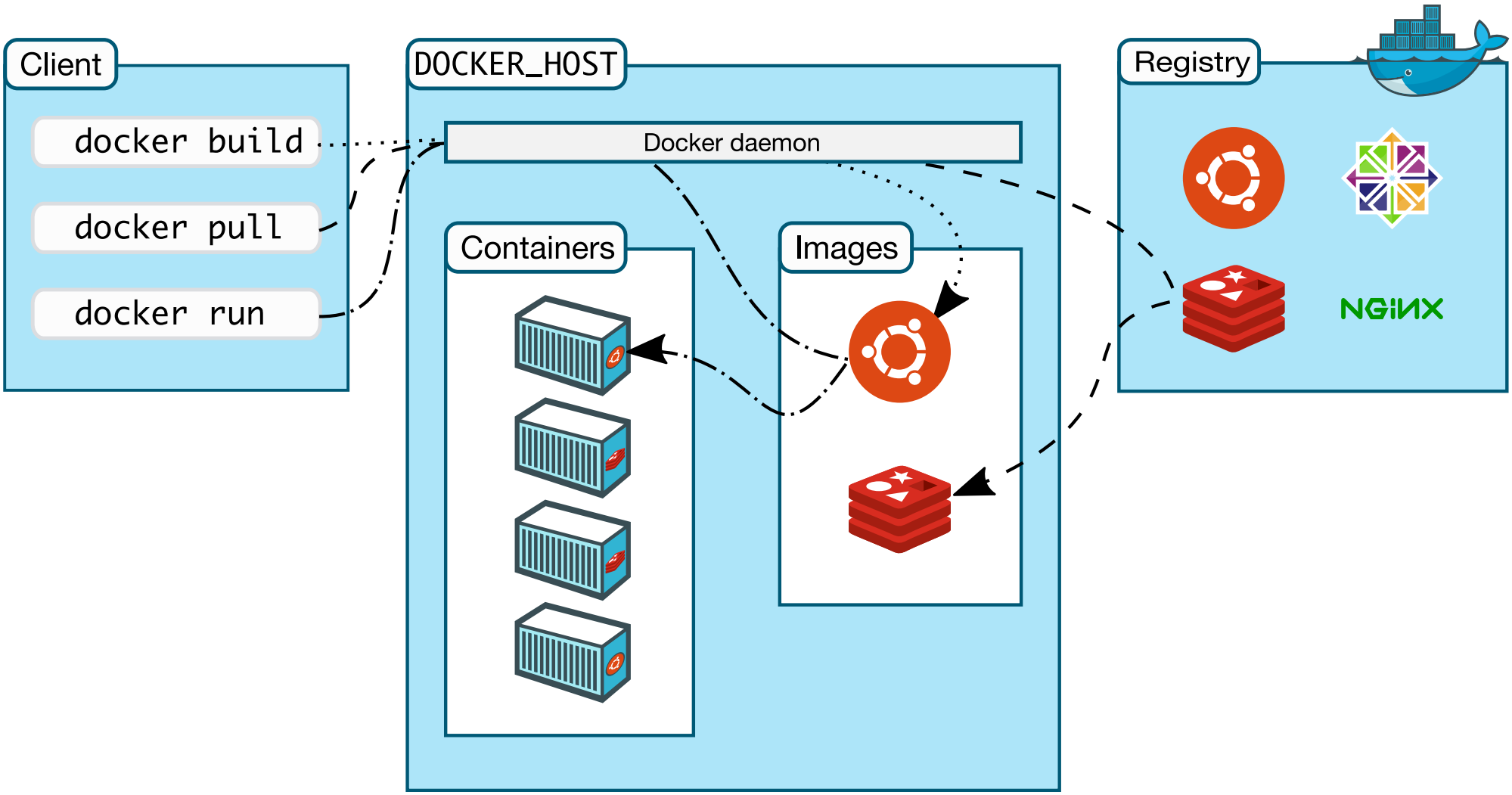
A running container represents a combination of namespace and sets of cgroups



Behind the scenes

- User types docker commands
- Docker client contacts docker daemon
- Docker daemon checks if image exists
- Docker daemon downloads image from docker registry if it does not exist
- Docker daemon runs container using image

Docker architecture



Images and Containers

Docker images

- Images are the basis of containers
- An image is a *readonly* file system similar to tar archive
- *Distributable* artefact of Docker

Types of images

Official Base Image

Images that have no parent (alpine, ubuntu, debian)

Base Image

Can be any image (official or otherwise) that is used to build a new image

Child Images

Build on base images and add functionality (this is the type you'll build)

Layering of images

- Images are *layered*
- Images always consist of an *official base image*
  - ubuntu:14.04
  - alpine:latest
- Any child image built by adding layers on top of base
- Each successive layer is set of differences to preceding layer

Exercise: Create a basic image

```
$ docker run -t -i ubuntu:16.04 /bin/bash

root@69079aaaaab1:/$ apt-get update
root@69079aaaaab1:/$ exit

$ docker commit 69079aaaaab1 ubuntu:update
13132d42da3cc40e8d8b4601a7e2f4dbf198e9d72e37e19ee1986c280ffcb97c

$ docker image ls
$ docker diff 69079aaaaab1
$ docker history ubuntu:16.04
$ docker history ubuntu:update
```

- Created a new layer (cache files added by apt)
- Not an ideal way to create images
- Better to create images using a Dockerfile

Create images with a *Dockerfile*

- A text file. Usually named `Dockerfile`
- Sequential instructions for building a Docker image
- Each instruction creates a layer on the previous
- A very simple Dockerfile with 4 layers:

```
FROM ubuntu:15.10
COPY . /app
RUN make /app
CMD ["python", "/app/app.py"]
```

Structure of a Dockerfile

- Tell Docker which base image to use

```
FROM ubuntu:15.10
```

- A number of commands telling docker how to build image

```
COPY . /app
RUN make /app
```

- Optionally tell Docker what command to run when the container is started

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

Common Dockerfile Instructions

...a non-exhaustive list

FROM

Define the base image for a new image

```
FROM ubuntu:17.04

FROM debian # :latest implicit

FROM my-custom-image:1.2.3
```

RUN

```
RUN apt-get update && apt-get install python3

RUN mkdir -p /usr/local/myapp && cd /usr/local/myapp

RUN make all

RUN curl https://domain.com/somebig.tar | tar -xv | /bin/sh
```

Execute shell commands for building image

WORKDIR

```
WORKDIR /usr/local/myapp
```

- Create a directory to start in when container runs
- Will be created if does not exist

COPY

```
COPY package.json /usr/local/myapp

COPY . /usr/share/www
```

Copy files from build directory into image

ENTRYPOINT

```
ENTRYPOINT ["node", "index.js"]

ENTRYPOINT ["python3", "app.py"]

ENTRYPOINT python3 app.py
```

- Configure container to run executable by default
- Preferred to use JSON array syntax (best practices)

CMD

```
CMD ["node", "index.js"]

ENTRYPOINT ["python3", "manage.py"]
CMD ["test"]
```

- Provide defaults to executable
- or provide executable
- Also, preferred to use JSON array syntax (best practices)
- Last argument to `docker run` overrides `CMD`

ENTRYPOINT & CMD

Hypothetical application

```
FROM ubuntu:latest
.
.
ENTRYPOINT ["./base-script"]
CMD ["test"]

$ docker run my-image
```

By default this image will just pass `test` as argument to `base-script` to run unit tests by default

```
$ docker run my-image server
```

Passing argument at the end tells it to override CMD and execute with `server` to run server feature

more Dockerfile instructions

EXPOSE

ports to expose when running

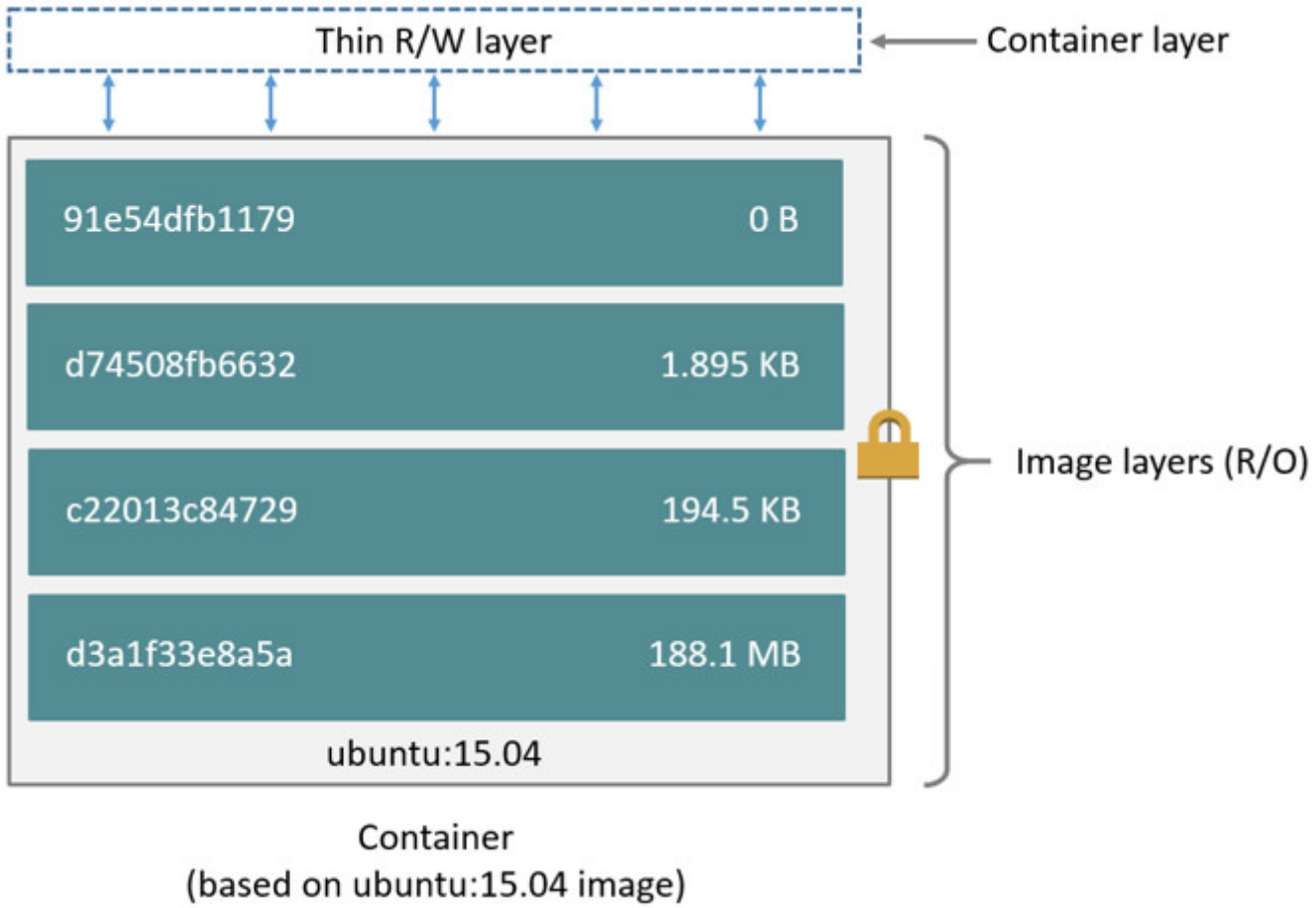
VOLUME

folders to expose when running

HEALTHCHECK CMD

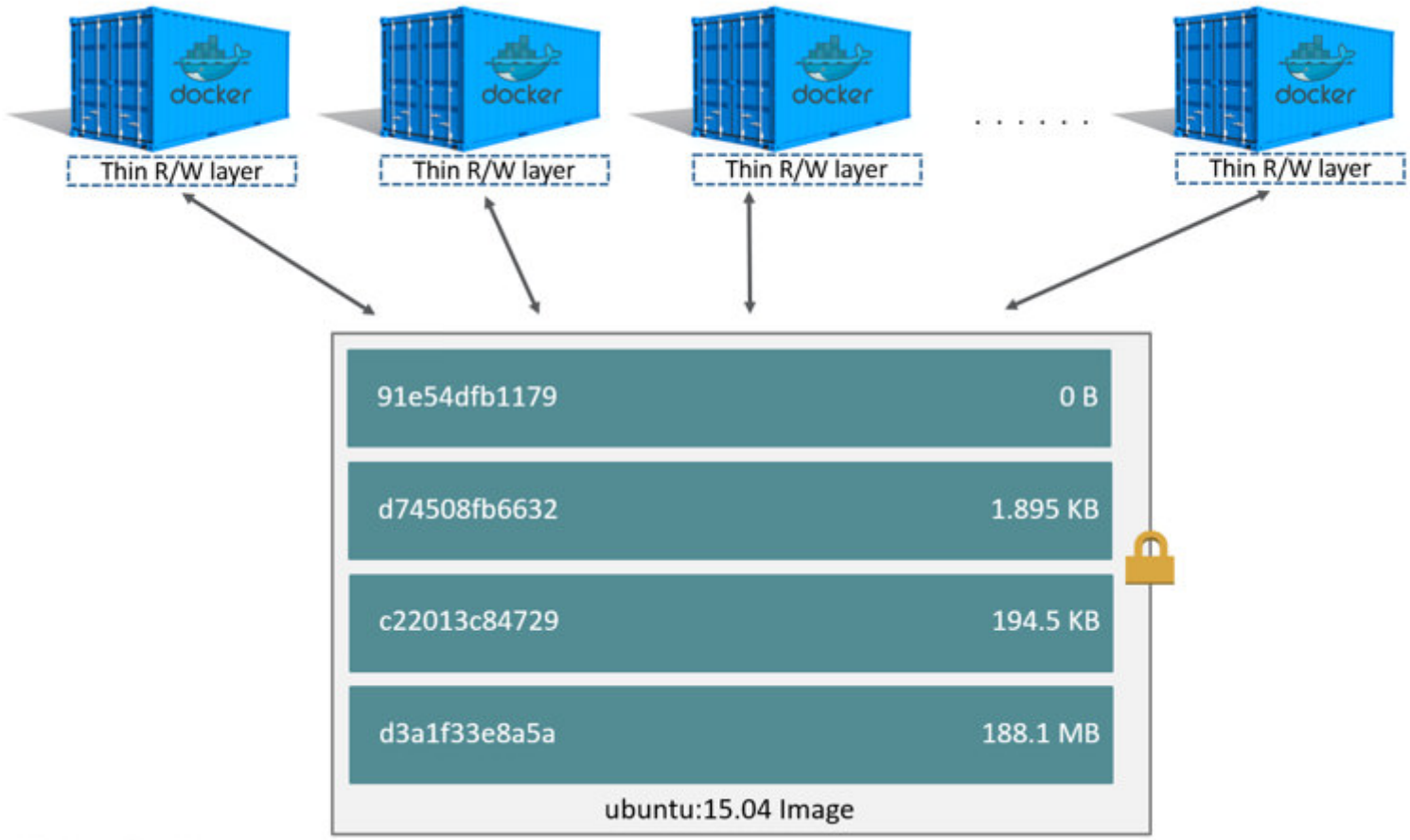
Check container health by running command at regular intervals inside container

Image layers



Container layering

- Container creates its own read/write layer on top of image
- Multiple containers each have own read/write layer, but can share the actual image



Sharing image layers

- Images will share any common layers
- Applies to
  - Images pulled from Docker
  - Images you build yourself

Exercise: Build images with common layers

~/docker-introduction/sample-code/layering

Dockerfile.base

```
FROM ubuntu:16.10
COPY . /app
```

Dockerfile

```
FROM acme/my-base-image:1.0
CMD /app/hello.sh
```

hello.sh

```
#!/bin/sh
echo "Hello world"
```

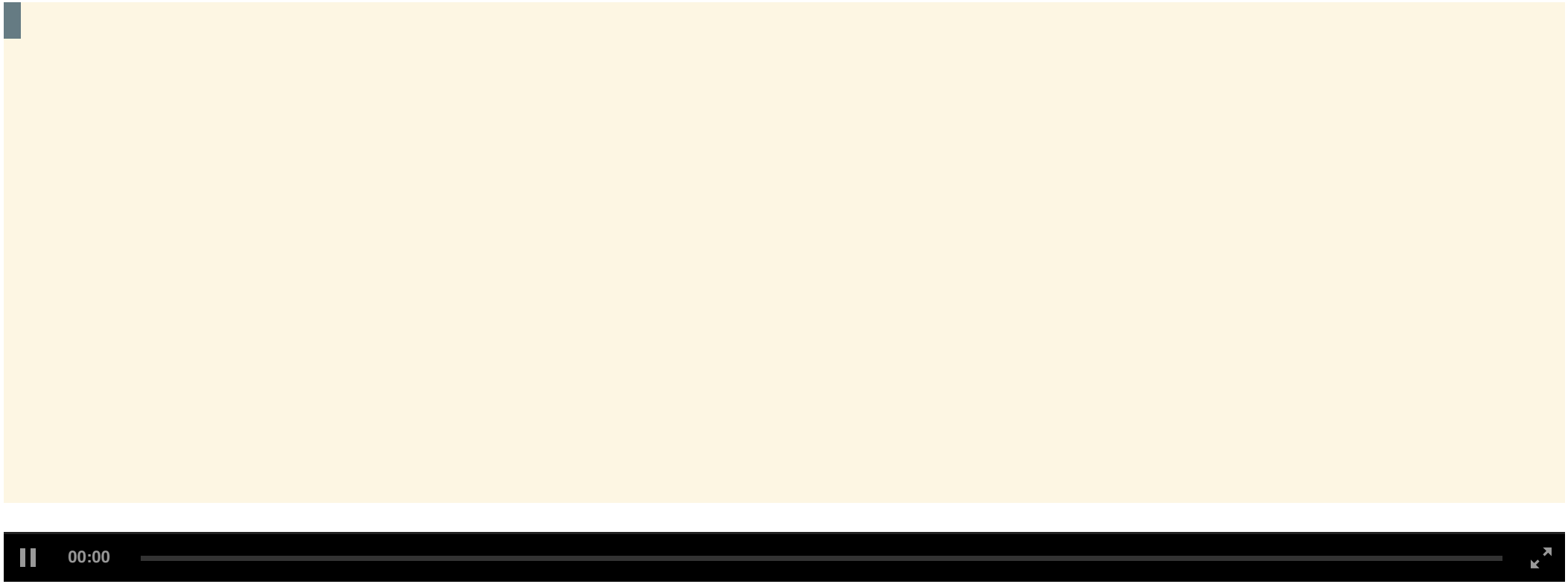
Build base image

```
$ docker build -t acme/my-base-image:1.0 -f Dockerfile.base .
```

```
docker-training
❯ docker build -t acme/my-base-image:1.0 -f Dockerfile.base .
Sending build context to Docker daemon 4.096 kB
Step 1/2 : FROM ubuntu:16.10
16.10: Pulling from library/ubuntu
869d7e479fb8: Downloading [=======================================>           ]  32.4 MB/42.59 MB
fcde8cc75da4: Download complete
b9d18efd03be: Download complete
95ed9114795e: Download complete
63ec97b2b19c: Download complete
```

Build child image

```
$ docker build -t acme/my-final-image:1.0 -f Dockerfile .
```



Compare base and final image

- The final image should contain all the same layers as the base image
- One additional layer: the last line of the Dockerfile

```
$ docker history acme/my-base-image:1.0
$ docker history acme/my-final-image:1.0
IMAGE          ...                                             SIZE
5932655b26aa   ...   #(nop)  CMD ["/bin/sh" "-c" "/a...       0 B<--new layer
2f723f94263a   ...   #(nop) COPY dir:dd75f285798cdc9...       106 B
8d4c9ae219d0   ...   #(nop)  CMD ["/bin/bash"]                0 B
<missing>      ...   mkdir -p /run/systemd && echo '...       7 B
<missing>      ...   sed -i 's/^#\s*\(deb.*universe\...       2.78 kB
<missing>      ...   rm -rf /var/lib/apt/lists/*              0 B
<missing>      ...   set -xe   && echo '#!/bin/sh' >...       745 B
<missing>      ...   #(nop) ADD file:9e2eabb7b05f940...       106 MB
```

Images and Tags

- Tags specify a particular version of an image

```
$ docker pull ubuntu:14.04
```

- Default to *latest*. In most cases this is a LTS version

```
$ docker pull ubuntu
```

- Registries like Docker Hub contain >> 100K images

```
$ docker search ubuntu
```

Dockerising applications

Create web application in Docker

- Create a small web app based on Python Flask
- Write a Dockerfile
- Build an image
- Run the image
- Upload image do Docker Registry

Step 1. Set up the web app

- Under `~/docker-introduction/sample-code/flask-app`

  app.py
  A simple flask application for displaying cat pictures

  requirements.txt
  list of dependencies for flask

  templates/index.html
  A jinja2 template

  Dockerfile
  Instructions for building a Docker image

Our Dockerfile

```
FROM alpine:3.5

# Install python and pip
RUN apk add --update py2-pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

CMD ["python", "/usr/src/app/app.py"]
```

Build the Docker image

```
$ cd ~/docker-introduction/sample-code/flask-app
$ docker build -t YOURNAME/myfirstapp .
```

```
➜  flask-app docker build -t heytrav/myfirstapp .
Sending build context to Docker daemon 8.192 kB
Step 1/8 : FROM alpine:3.5
3.5: Pulling from library/alpine
Digest: sha256:58e1a1bb75db1b5a24a462dd5e2915277ea06438c3f105138f97eb53149673c4
Status: Downloaded newer image for alpine:3.5
 ---> 4a415e366388
Step 2/8 : RUN apk add --update py2-pip
 ---> Running in a882d6e9cc6e
fetch http://dl-cdn.alpinelinux.org/alpine/v3.5/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.5/community/x86_64/APKINDEX.tar.gz
(1/12) Installing libbz2 (1.0.6-r5)
(2/12) Installing expat (2.2.0-r0)
(3/12) Installing libffi (3.2.1-r2)
(4/12) Installing gdbm (1.12-r0)
(5/12) Installing ncurses-terminfo-base (6.0-r7)
(6/12) Installing ncurses-terminfo (6.0-r7)
(7/12) Installing ncurses-libs (6.0-r7)
(8/12) Installing readline (6.3.008-r4)
(9/12) Installing sqlite-libs (3.15.2-r0)
```

Note: please replace YOURNAME with your Docker Hub username

Run the container

```
$ docker run -p 8888:5000 --name myfirstapp YOURNAME/myfirstapp
```

```
➜  flask-app docker run -p 8888:5000 --name myfirstapp --rm heytrav/myfirstapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
172.17.0.1 - - [07/May/2017 11:17:17] "GET / HTTP/1.1" 200 -
```

...Now open your test webapp

Login to a registry

```
$ docker login <registry url>
```

```
example-voting-app/vote
❯ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID,
reate one.
Username: heytrav
Password:
```

- If registry not specified, logs into hub.docker.com
- Can log in to multiple registries

Push image to registry

```
$ docker push YOURNAME/myfirstapp
```

```
➜  ~ docker push heytrav/myfirstapp
The push refers to a repository [docker.io/heytrav/myfirstapp]
e3289250a9d4: Pushed
716e864bdc5d: Pushed
7e0c0ae20bfa: Pushing [==================================================>] 5.055 MB
b106808228c0: Pushed
f3fd8cb51e02: Pushing [===========>                                       ] 10.92 MB/48.64 MB
23b9c7b43573: Pushing [>                                                  ]  68.1 kB/3.987 MB
```

Summary

- Wrote a small web application
- Used Dockerfile to create an image
- Pushed image to upstream registry

Dockerfile best practices

General guidelines

- Containers should be as ephemeral as possible
- Use a `.dockerignore` file
- Avoid installing unnecessary packages
- Minimise concerns
  - Avoid multiple processes/apps in one container

General guidelines

- Use current official repositories in FROM as base image
  - debian 124 MB
  - ubuntu 117 MB
  - alpine 3.99 MB
  - busybox 1.11 MB
- Minimise Layers
- Sort multiline arguments
- Split complex RUN statement on separate lines with backslashes
- Run apt-get update and apt-get install in same RUN
- Run clean up in same line whenever possible

Layer caching

```
$ cd ~/docker-introduction/sample-code/caching
$ docker build -t caching-example -f Dockerfile.layering .
```

Consequences of layer caching

```
# Example 1
FROM ubuntu:latest
RUN apt-get update
RUN apt-get install -y curl
#RUN apt-get install -y nginx
```

```
# Example 2
FROM ubuntu:latest
RUN apt-get update \
  && apt-get install -y curl #nginx
```

Minimise Layers

Remove non-essential files when possible.

Image size: 471 MB

```
FROM ubuntu:latest

RUN apt-get update \
 && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    mercurial \
    reprepro
```

Image size: 430 MB

```
FROM ubuntu:latest

RUN apt-get update \
  && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    mercurial \
    reprepro \
&& rm -rf /var/lib/apt/lists/*
```

ADD

ADD vs COPY

CMD

- Used to run software contained by image
- Should be run in form
  - `CMD ["executable", "param1", "param2", ..]`
- Or in form that creates interactive shell like
  - `CMD ["python"]`
  - `CMD ["/bin/bash"]`
- Avoid
  - `CMD "executable param1 param2 ..."`

ENTRYPOINT

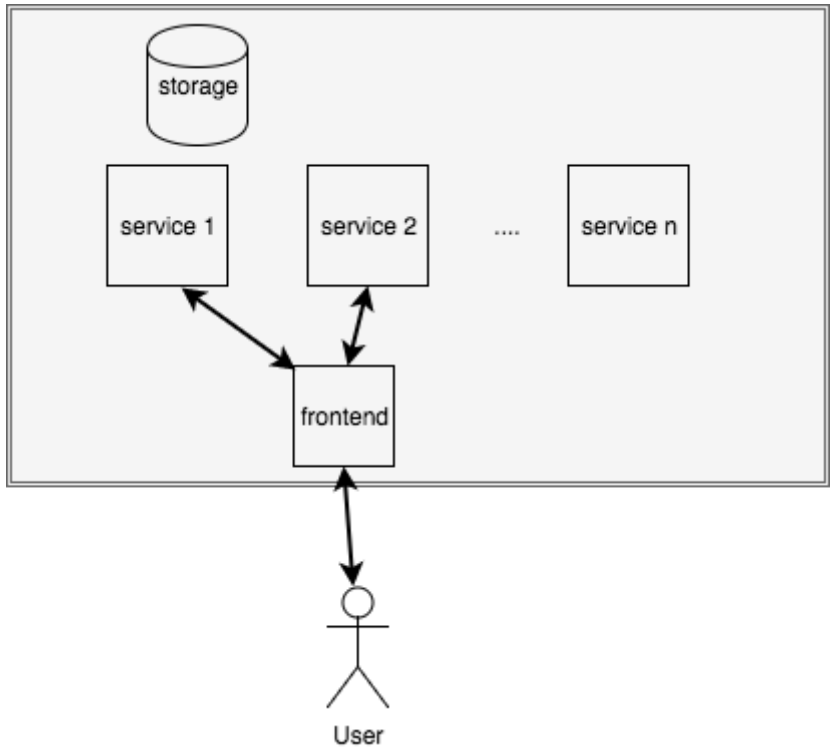
```
ENTRYPOINT ["python", "manage.py"]
CMD ["test"]
```

- When used in conjunction with CMD:
  - Set base command with ENTRYPOINT
  - Use CMD to set default argument
- Will just run tests when container is run with no params
  - `docker run myimage`
- Can override by passing argument to container
  - `docker run myimage runserver`
- For more see Dockerfile Best practices

Docker and Microservices

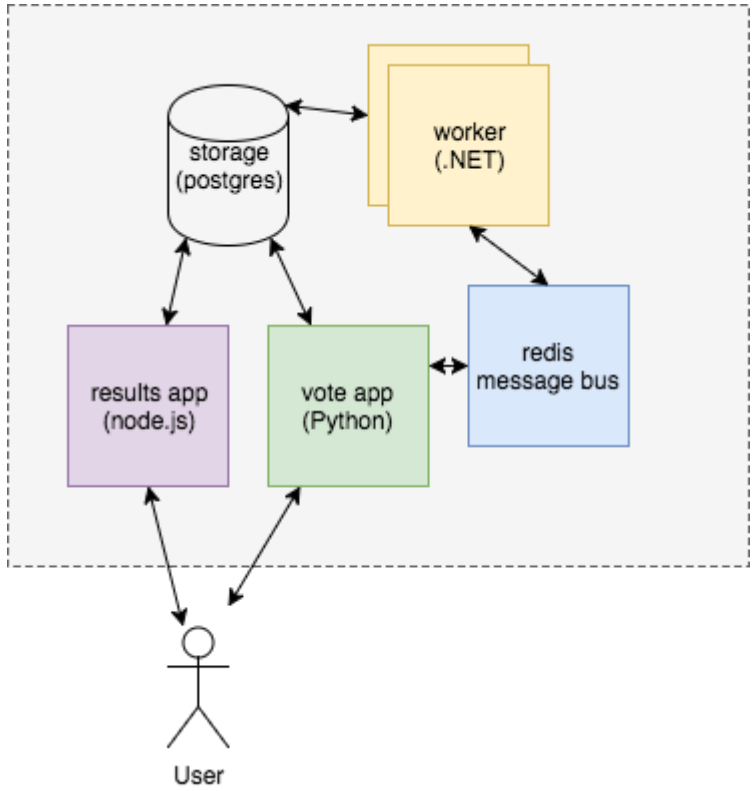
Microservices vs. Monoliths

- Small decoupled applications vs. one big app
- Developed independently
- Deployed and updated independently
- Scaled independently
- Better modularity



Build a voting app

- Python web application
- Redis queue
- .NET worker
- Postgres DB with a data volume
- Node.js app to show votes in real time



Build vote app components

```
$ cd ~/example-voting-app
$ docker build -t vote vote
$ docker build -t result result
$ docker build -t worker worker
```

Run service containers

```
$ docker run --rm -d -p 6379:6379 --name redis redis:alpine
$ docker run --rm -d --name db postgres:9.4
```

- Redis to act as message bus for microservices
- Postgres for storage of voting results

Run microservices

```
$ docker run --rm -d --name vote --link redis \
   --link db -v $PWD/vote:/app  -p 5000:80 vote
$ docker run --rm -d --name worker --link redis --link db worker
$ docker run --rm -d --name result -v $PWD/result:/app \
   --link db -p 5001:80 -p 5858:5858 result nodemon --debug server.js
```

- `--name` flag to specify name of container
- `--link` flag to tell docker to bridge two or more containers
- Voting app: http://localhost:5000
- Results app: http://localhost:5001

Disadvantages of this approach

- Tedious to type commands
- Can't scale individual services
- Better to use orchestration platforms

Container Orchestration

First, some more buzzwords

- Immutable infrastructure
- Cattle vs pets
- Snowflake Servers vs. Phoenix Servers

Immutable Architecture/Infrastructure

- Phoenix servers
- The environment is defined in code
- If you need to change *anything* you create a new instance and destroy the old one
- Docker makes it much more likely you will work in this way

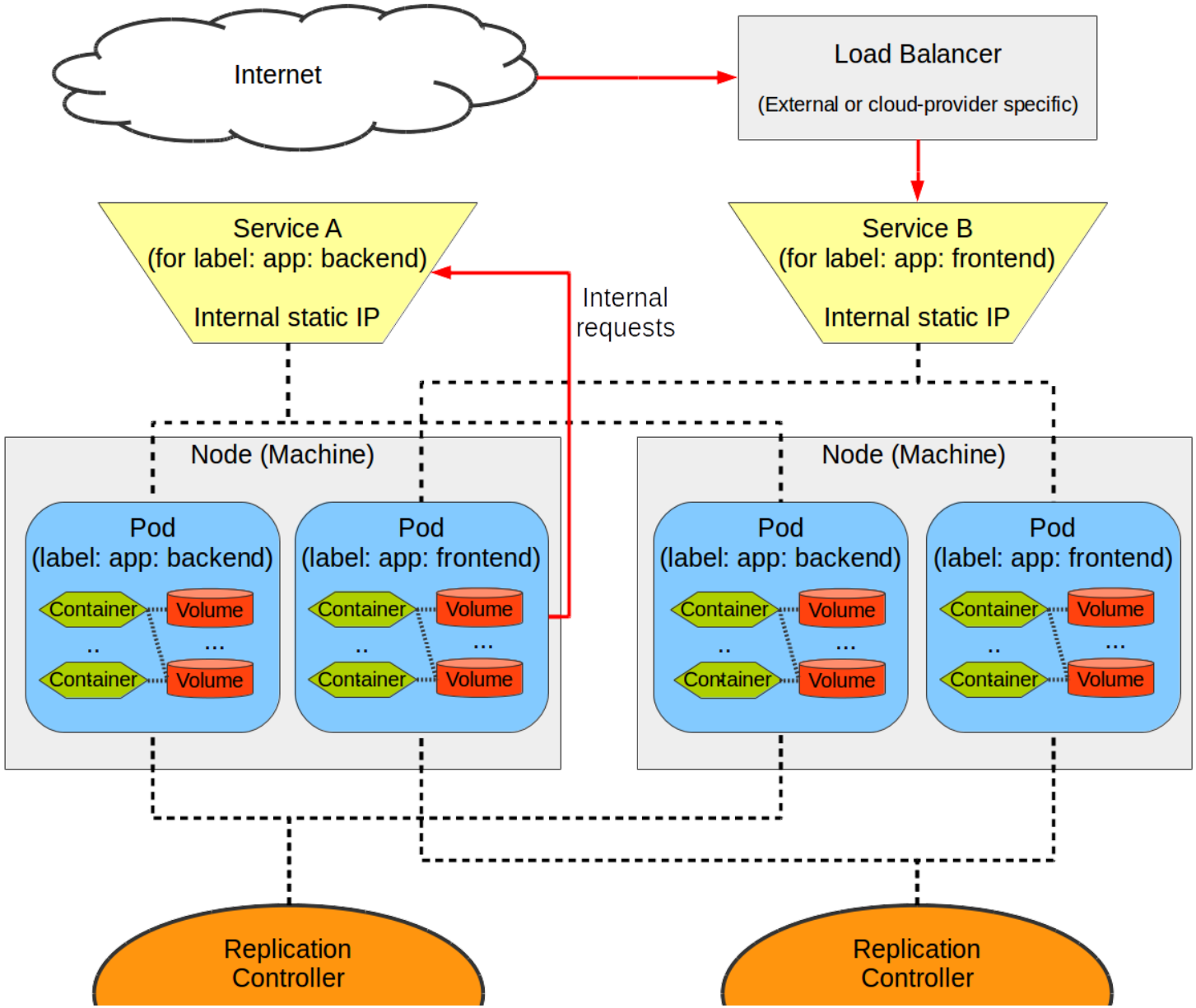
Container orchestration

- Frameworks for container orchestration
  - Docker Swarm
  - Kubernetes
- Manage deployment/restarting containers across clusters
- Networking between containers (microservices)
- Scaling microservices
- Fault tolerance

Kubernetes

- Container orchestrator
- Started by Google
- Inspired by Borg (Google's cluster management system)
- Open source project written in Go
- Cloud Native Computing Foundation
- Manage applications not machines

Kubernetes Overview




(for label: app: backend)
(for label: app: frontend)

Kubernetes Components

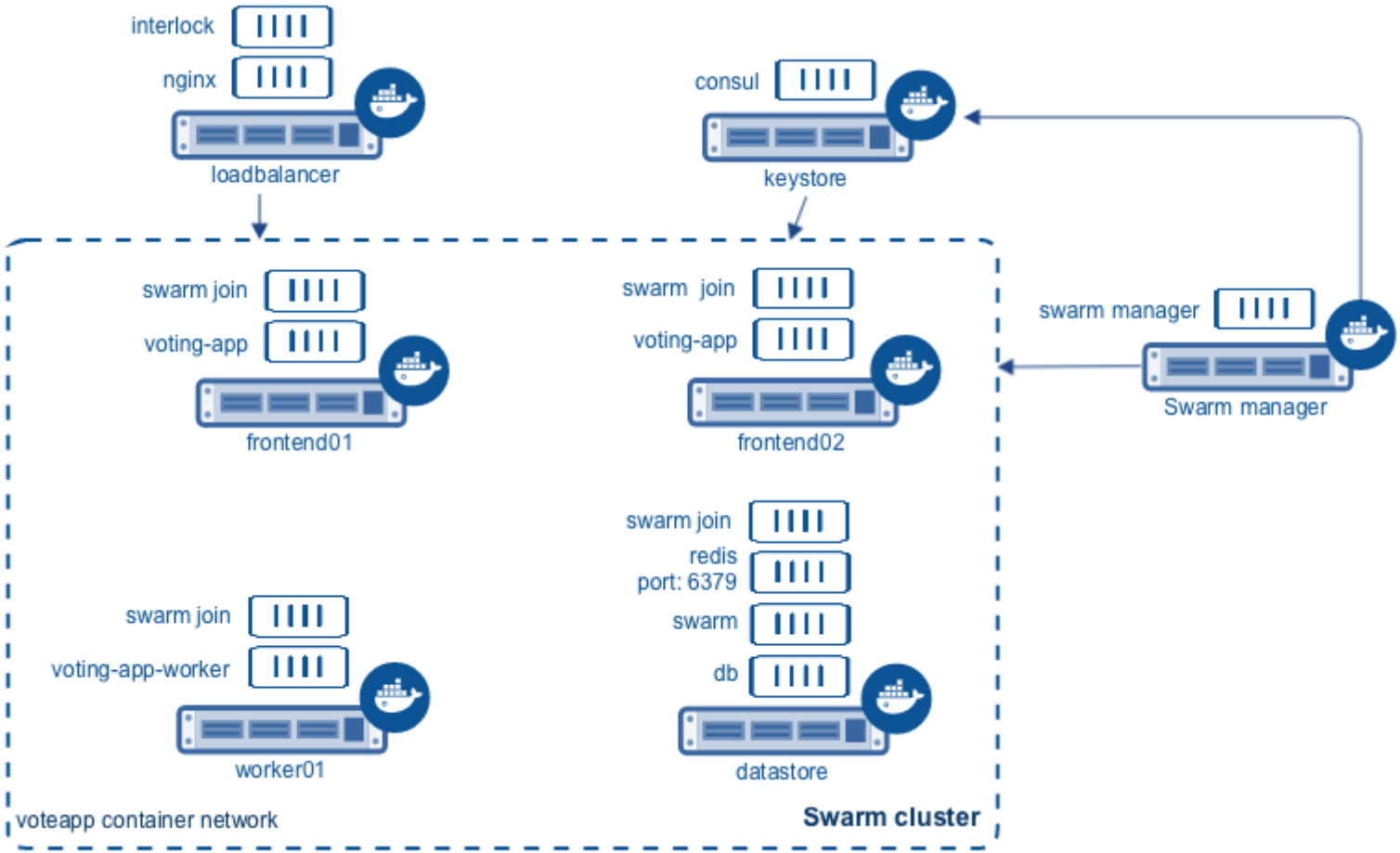
- Pods - an ephemeral group of co-scheduled containers that together provide a service
- Flat Networking Space - each pod has an IP and can talk to other pods, within a pod containers communicate via localhost (need to manage ports)
- Labels - Key value pairs, used to label pods and other objects so the scheduler can operate on them
- Services - stable endpoints comprised of one or more pods (external services are supported)
- Replication Controllers - the orchestrator that controls and monitors the pods within a service (known as replicas)

Docker Swarm

- Standard since Docker 1.12
- Manage containers across multiple machines
  - Scaling services
  - Healthchecks
  - Load balancing

Docker Swarm

- Two types of machines or *nodes*
  - 1 or more *manager* nodes
  - 0 or more *worker* nodes
- Managers control global state of cluster
  - Raft Consensus Algorithm
  - If one manager fails, any other should take over


interlock
nginx
loadbalancer
consul
keystore
swarm join
voting-app
frontend01
swarm join
voting-app
frontend02
swarm manager
Swarm manager
swarm join
redis
port: 6379
swarm
db
datastore
swarm join
voting-app-worker
worker01
voteapp container network
Swarm cluster

Docker Compose file

- Configure our services with a *compose file*
- A type of *service contract*
- yaml syntax
- Specifies
  - which services to run
  - scaling
  - network
  - mount file volumes
  - healthchecks
  - environment variables
  - secrets

```
# stack.yml
version: "3.3"
services:
  db:
    image: postgres:9.4
    .
    .
  redis:
    image: redis:latest
    deploy:
      replicas: 3

  vote:
    image: vote:latest
    depends_on:
      - redis
      - db
     deploy:
```

Initiate a Swarm

```
$ docker swarm init
$ cd ~/example-voting-app
```

- `docker swarm init` puts your machine in *swarm mode*
- Only need to do once to create manager node

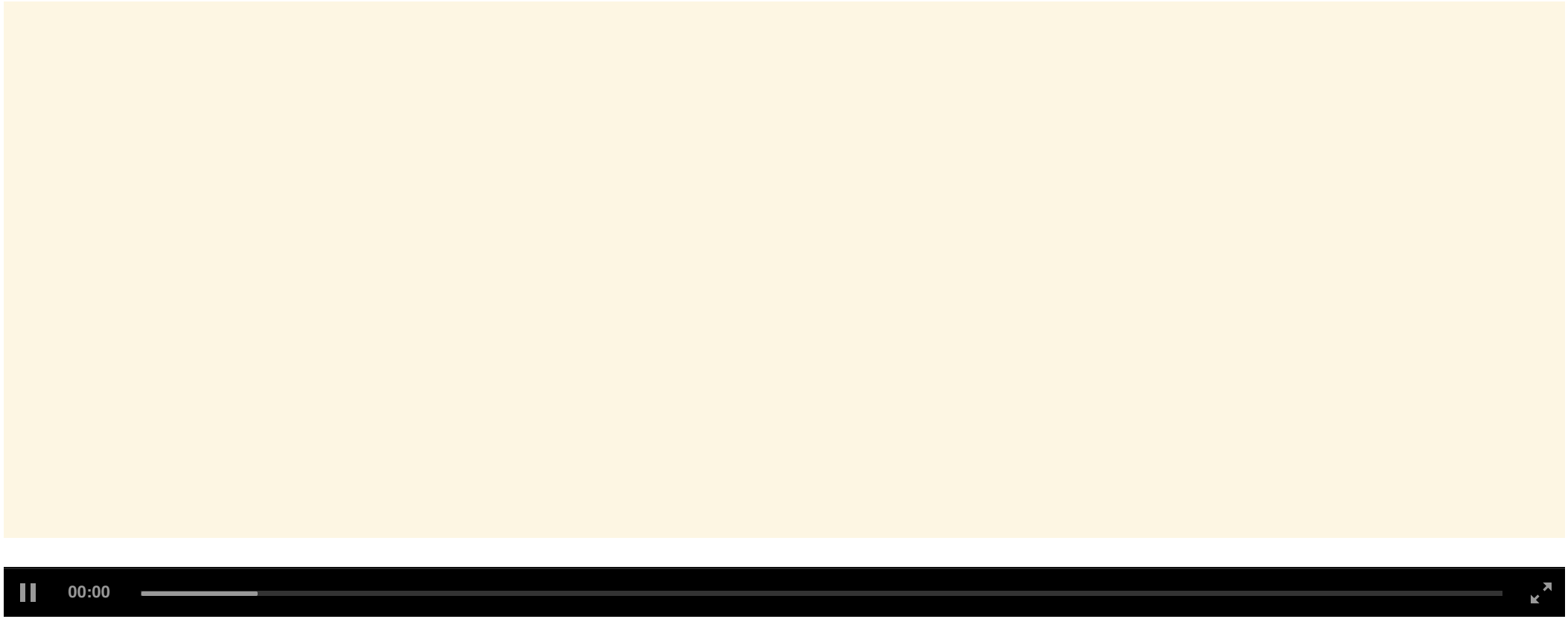
Deploy the stack

```
$ docker stack deploy --compose-file docker-stack.yml vote
```

```
master
❯ docker stack deploy --compose-file docker-stack.yml vote
Creating network vote_frontend
Creating network vote_backend
Creating network vote_default
Creating service vote_redis
Creating service vote_db
Creating service vote_vote
Creating service vote_result
```

Verify stack is running

```
$ watch docker stack ps vote
```



Now, let's go vote! When you're done, have a look at the results.

Modify vote app

- Open up `app.py`
- On lines 8 & 9, modify vote options
- Build image
- Push to Docker Hub (optional)

Change vote options

```
example-voting-app
❯ vim
```

Build image

In example-voting-app...

```
$ docker build -t vote:v2 vote
```

Note: please replace `yourname` with your docker hub username if you have one

```
Collecting itsdangerous>=0.21 (from Flask->-r requirements.txt (line 1))
  Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting click>=2.0 (from Flask->-r requirements.txt (line 1))
  Downloading click-6.7-py2.py3-none-any.whl (71kB)
Collecting Jinja2>=2.4 (from Flask->-r requirements.txt (line 1))
  Downloading Jinja2-2.9.6-py2.py3-none-any.whl (340kB)
Collecting Werkzeug>=0.7 (from Flask->-r requirements.txt (line 1))
  Downloading Werkzeug-0.12.1-py2.py3-none-any.whl (312kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->Flask->-r requirements.txt (line 1))
  Downloading MarkupSafe-1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous: started
  Running setup.py bdist_wheel for itsdangerous: finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/fc/a8/66/24d655233c757e178d45dea2de22a04c6d92766ab
  Running setup.py bdist_wheel for MarkupSafe: started
```

Update a service

```
$ docker service update --image vote:v2 vote_vote
```

Now go to the voting app and see what changed

Remove Swarm Stack

```
$ docker stack rm vote
```

```
example-voting-app git/master*
❯ docker stack rm vote
Removing service vote_redis
Removing service vote_result
Removing service vote_db
Removing service vote_vote
Removing service vote_worker
Removing service vote_visualizer
Removing network vote_backend
Removing network vote_frontend
Removing network vote_default

example-voting-app git/master*
❯
```

Summary

- Deployed a set of services on our local host
- Docker created a couple networks (front-tier, back-tier)
- Some services running multiple instances
- Next, we'll look at doing this across multiple machines

Running apps in the cloud

Goals

Setting up cluster

Ansible

- Python based tool set
- Automate devops tasks
  - server/cluster management
  - installing packages
  - deploying code
  - managing config

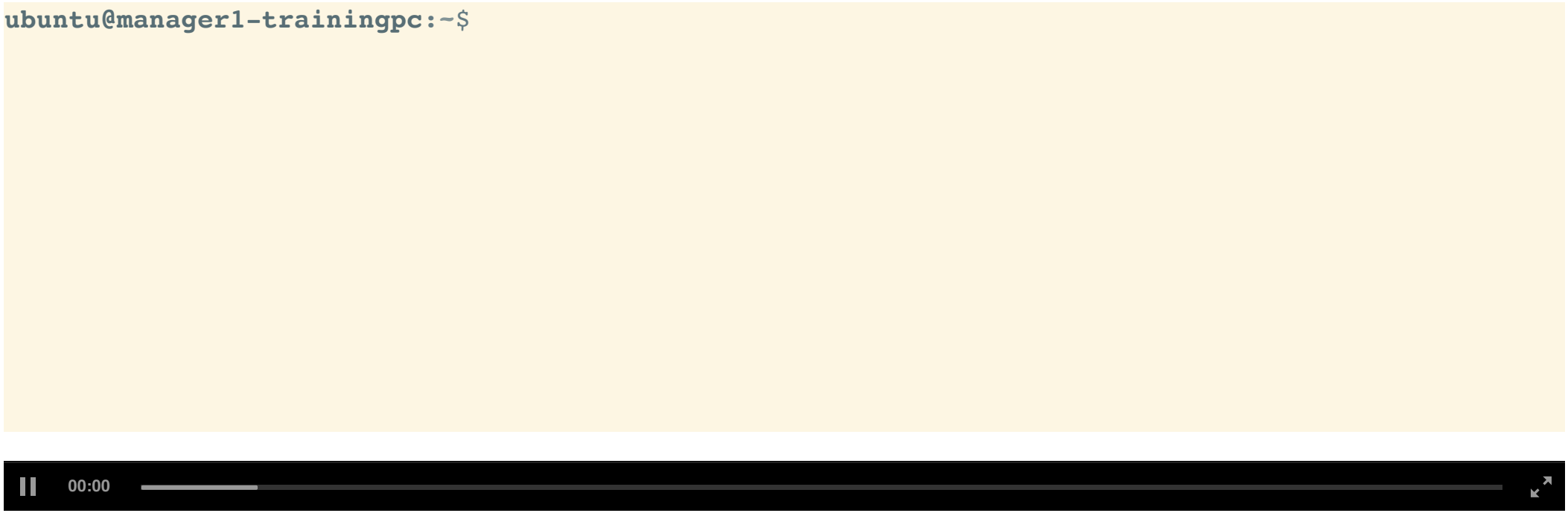
Setup steps

Create a cluster

```
$ cd ~/catalystcloud-ansible/example-playbooks/docker-swarm-mode
$ ansible-playbook -K create-swarm-hosts.yaml
```

Create Swarm

```
$ ssh manager<TAB><ENTER>
$ docker swarm init
```



Copy the `docker swarm join ...` command that is output

Join Worker Nodes

Paste the command from the manager node onto command line.

```
$ ssh worker1<TAB><ENTER>
$ docker swarm join --token $TOKEN  192.168.99.100:2377
```

```
ubuntu@worker1-trainingpc:~$
```

Repeat this for `worker2`

Check nodes

```
$ docker node ls
```

```
ubuntu@manager1-trainingpc:~$ docker node ls
ID                            HOSTNAME              STATUS              AVAILABILITY        MANAG
d1av3sf7qxmlbgtsc3jjpfw7b *   manager1-trainingpc   Ready               Active              Leade
i7jhcrv1ggamrbodueynlc2jh     worker2-trainingpc    Ready               Active
v4lo1ahpw89mmx042xzvotzt1     worker1-trainingpc    Ready               Active
ubuntu@manager1-trainingpc:~$
```

Deploying voting app

Upload docker-stack.yaml to manager node

```
$ cd ~/example-voting-app
$ scp docker-stack.yml manager1-TRAININGPC:~/
```

Deploy application

```
$ docker stack deploy -c docker-stack.yml vote
```

Monitor deploy progress

```
$ watch docker stack vote

$ watch docker service ls
```

Try out the voting app

http://voting.app:5000

To vote

http://voting.app:5001

To see results

http://voting.app:8080

To visualise running containers

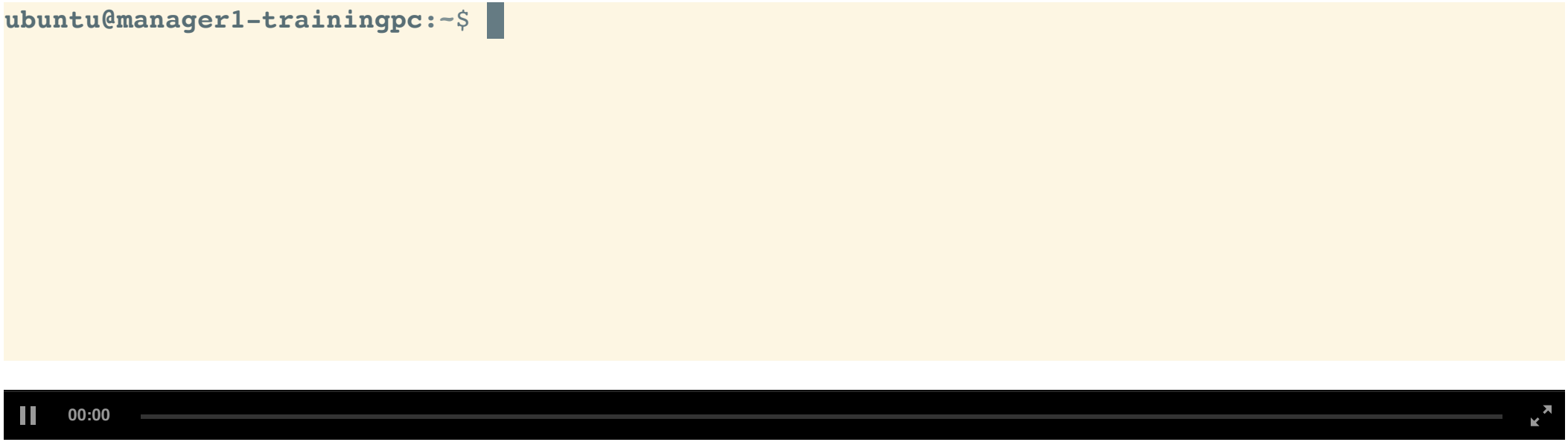
Scale services

```
$ docker service scale vote_vote=3
```

Look at the changes in the visualizer

Update a service

```
$ docker service update --image heytrav/vote vote_vote
```



Now go to the voting app and verify the change

Developer workflow

Drain a node

```
$ docker node update --availability drain worker1
```

- Sometimes necessary to take host offline
  - Planned maintenance
  - Patching vulnerabilities
  - Resizing host
- Prevents node from receiving new tasks
- Manager stops tasks running on node and launches replicas on active nodes

Return node to service

```
$ docker node update --availability active worker1
```

- during a service update to scale up
- during a rolling update
- when you set another node to Drain availability
- when a task fails on another active node

Summary

- Created a cluster with a cloud provider using ansible
  - 1 manager node
  - 2 worker nodes
- Deployed microservice for voting app in Docker Swarm
- Scaled service from 2 to 3 services
- Rolling-Updated image

Tear down your cluster

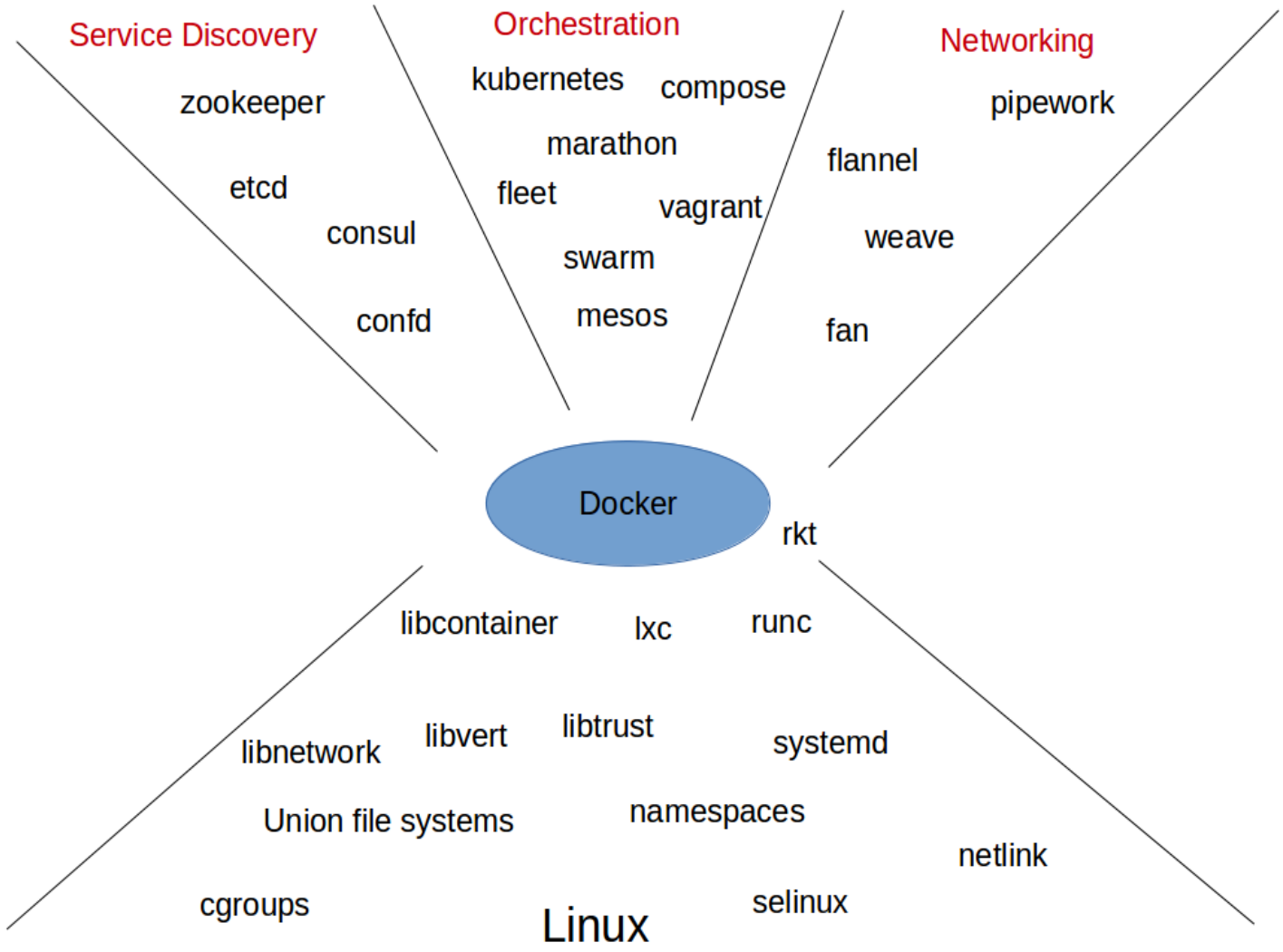
```
$ ansible-playbook -K remove-swarm-hosts.yaml
```

Wrap up

Docker ecosystem

- An explosion of tools
- Hard to keep up
- Lets have a quick look

Docker ecosystem



Competing technologies

- rkt (CoreOS)
- Serverless (FaaS)
  - Lambda (AWS)
  - Azure Functions (Microsoft)
  - Google cloud functions
  - iron.io

The end