catalyst

Intro to Docker

Presented by Travis Holton

Administrivia

- Bathrooms
- Fire exits

This course

- Makes use of official Docker docs
- Based on Docker version 1.13.1
- 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

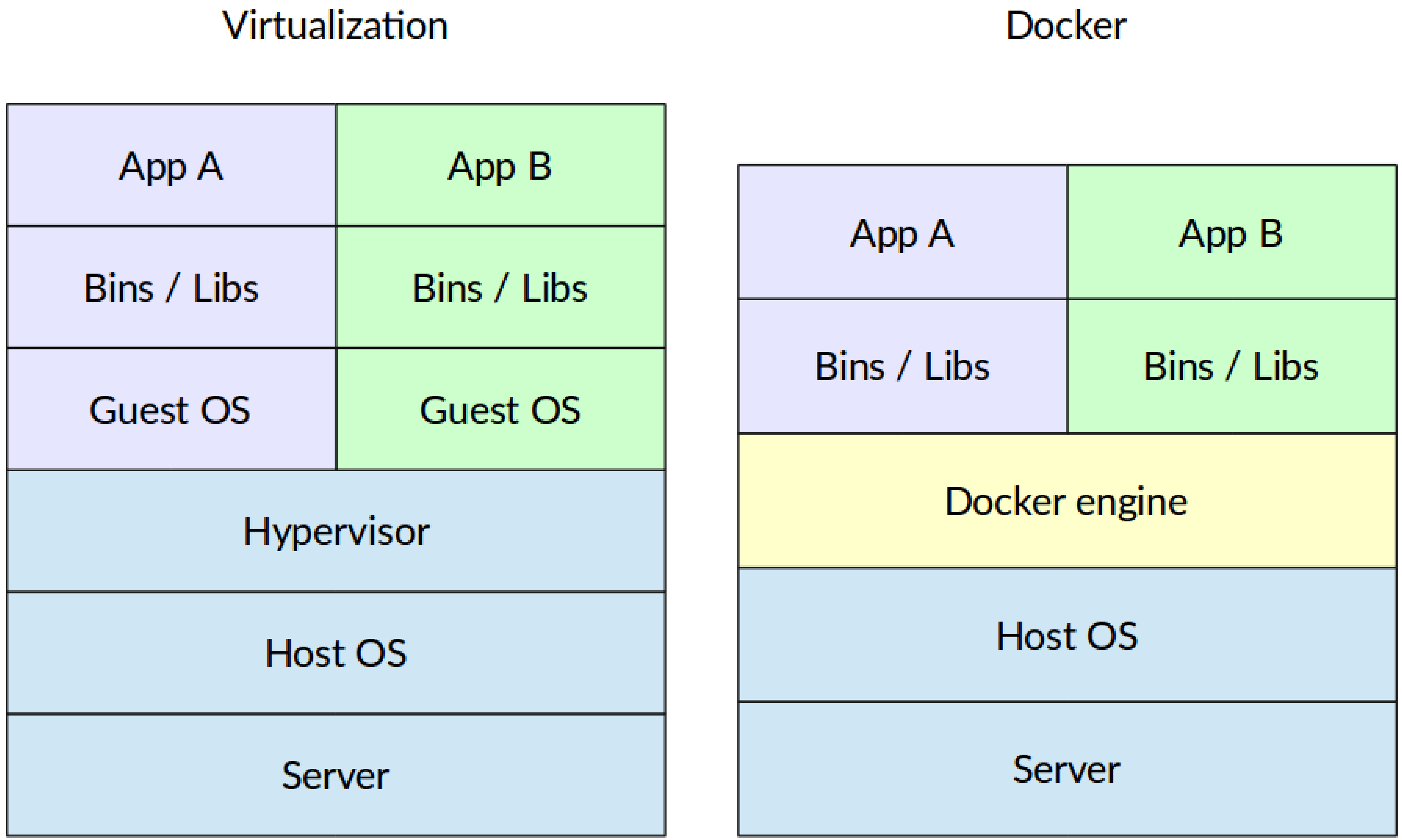
- Understand how to use Docker on the command line
- Understand where Docker can be used
- Appreciation of the larger ecosystem
- Get people thinking about where they could use Docker

Introduction to containers

What is containerization?

- An operating system level virtualization method for running distributed applications without launching an entire VM.
- Multiple isolated systems run on a single host and access a single kernel
- Key benefits:
  - Lightweight - Places less strain on overall resources.
  - Efficiency gains in storage, CPU
  - Portability

Lightweight



Virtualization

App A | App B

Bins / Libs | Bins / Libs

Guest OS | Guest OS

Hypervisor

Host OS

Server

Docker

App A | App B

Bins / Libs | Bins / Libs

Docker engine

Host OS

Server

Benefits of Containers: Resources

- Containers share a kernel
- Container image only contains
  - executable
  - application dependencies

Benefits of Containers: Decoupling

- Application stack not coupled to host machine
- Scale and upgrade services independently
- Configuration is coupled with the application
- Treat services like cattle instead of pets

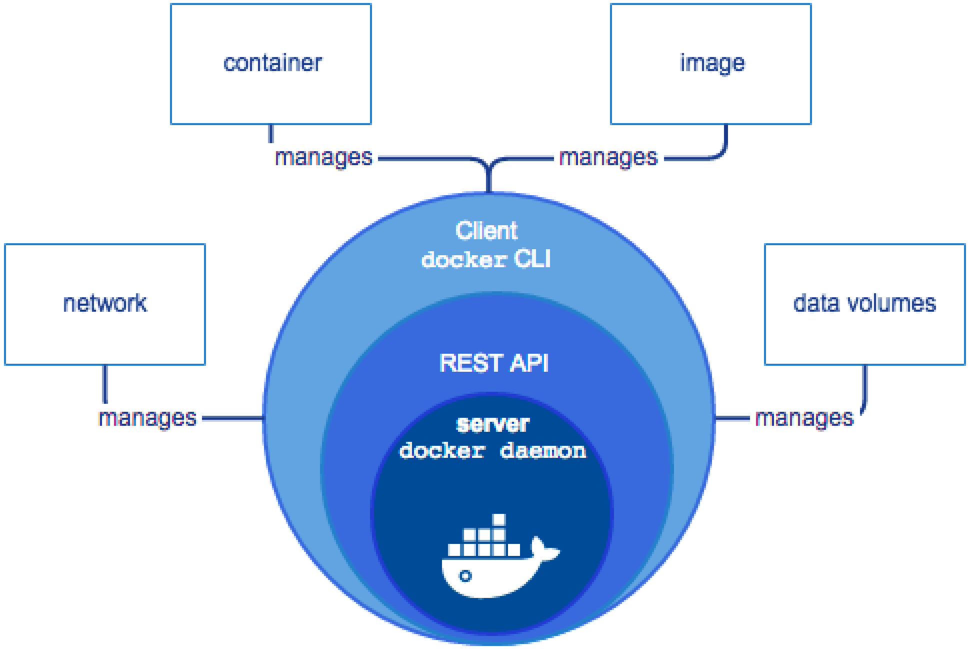
Introduction to Docker

The Docker Platform

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- Deploy application to staging/production environments

The Docker Engine

- A server which is a type of long-running program called a daemon process (the dockerd command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the docker command).



Uses of Docker

- Fast consistent delivery of applications
- Responsive deployment and scaling
- Running more workloads on same hardware

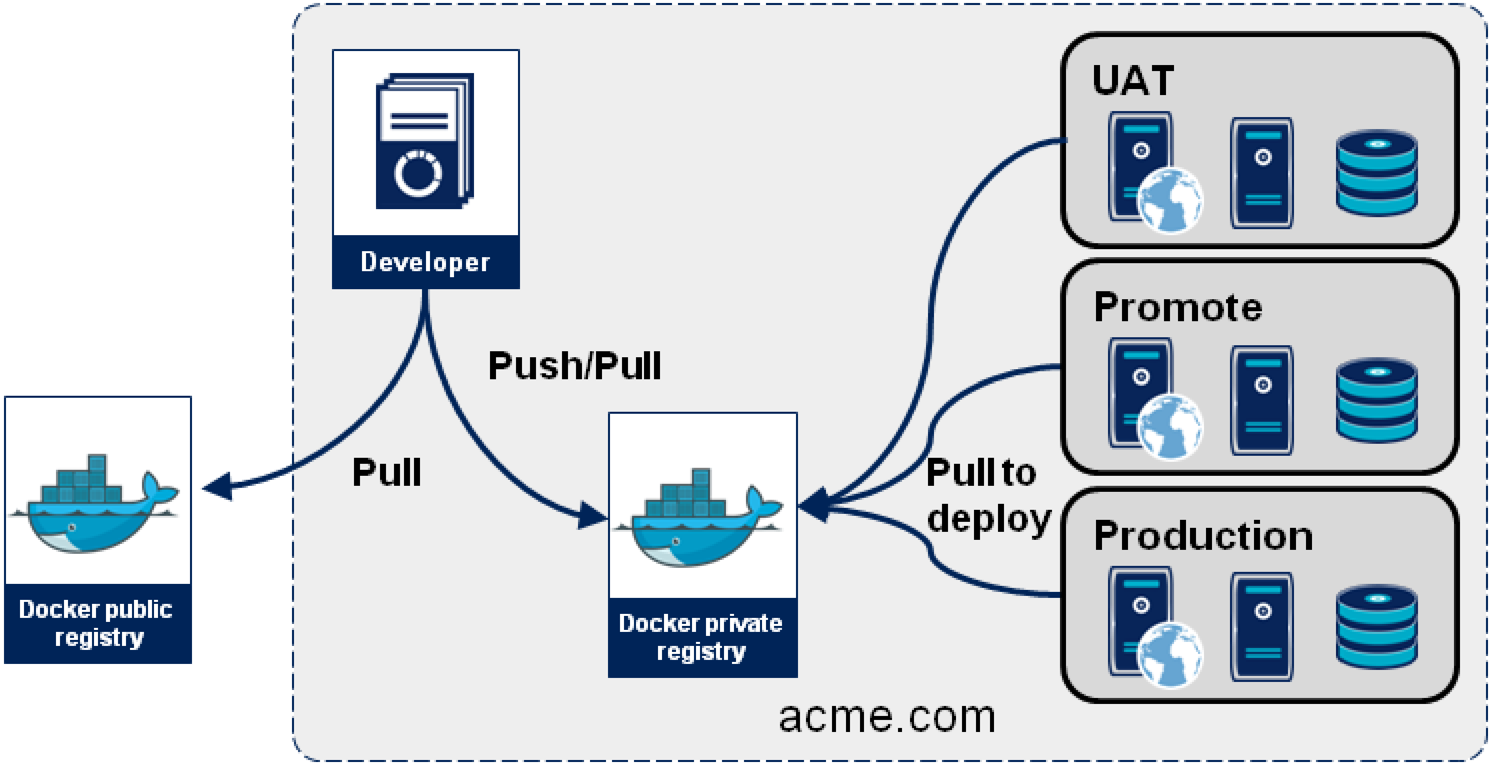
Docker Portability

- 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)

Why Docker?

- Containers generally lightweight compared to traditional virtualization
- Consistency across dev/test/prod
- Easy to modify, update and scale
- Dependency encapsulation and decoupling
- Enables new workflows

Developer workflow



Docker Hub

- Canonical home of official docker images
- Contains >100K docker images (private, public, official)
- Free, so let's sign up

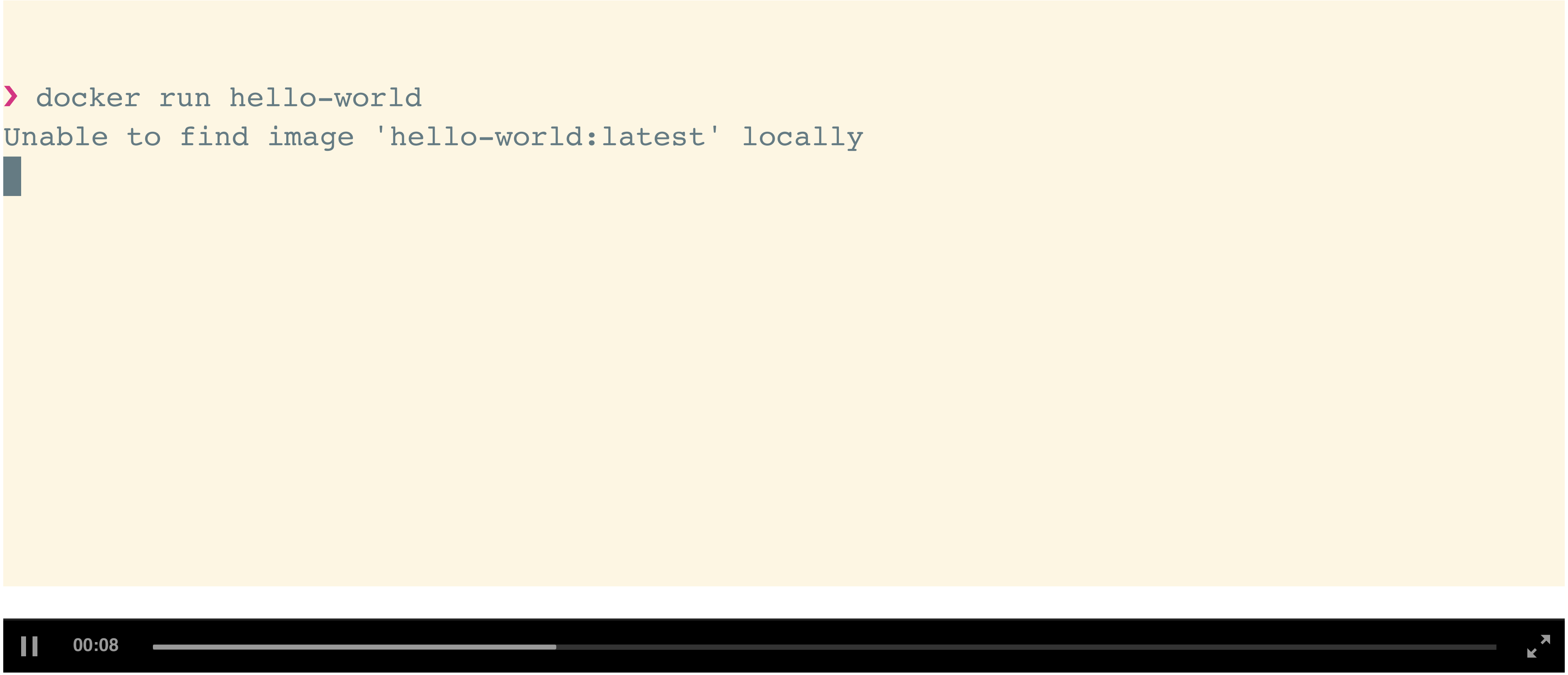
Running Docker

Run the `docker` client

```
$ docker<ENTER>

    Usage:  docker COMMAND

    A self-sufficient runtime for containers

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

Exercise: Hello world

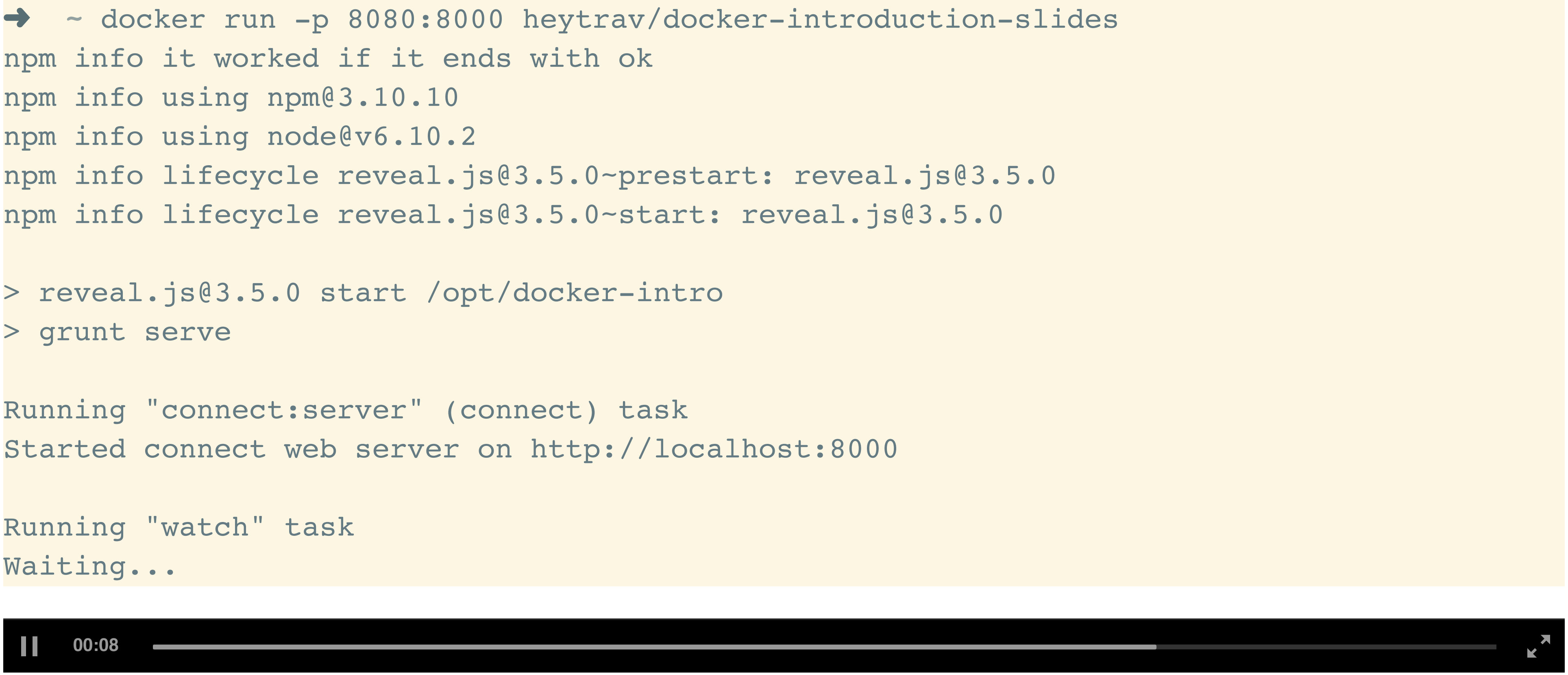
```
$ docker run hello-world
```



```
❯ docker run hello-world
Unable to find image 'hello-world:latest' locally
```

Exercise: Pull and run course slides image

```
$ docker run --name docker-intro --rm \
        -p 8000:8000 heytrav/docker-introduction-slides:may-30
```

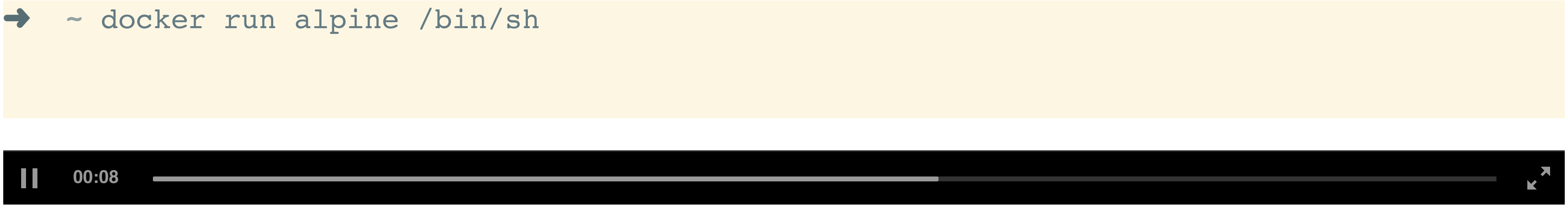


```
➜  ~ docker run -p 8080:8000 heytrav/docker-introduction-slides
npm info it worked if it ends with ok
npm info using npm@3.10.10
npm info using node@v6.10.2
npm info lifecycle reveal.js@3.5.0~prestart: reveal.js@3.5.0
npm info lifecycle reveal.js@3.5.0~start: reveal.js@3.5.0

> reveal.js@3.5.0 start /opt/docker-intro
> grunt serve

Running "connect:server" (connect) task
Started connect web server on http://localhost:8000

Running "watch" task
Waiting...
```

View course slides

Exercise: Start a shell

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

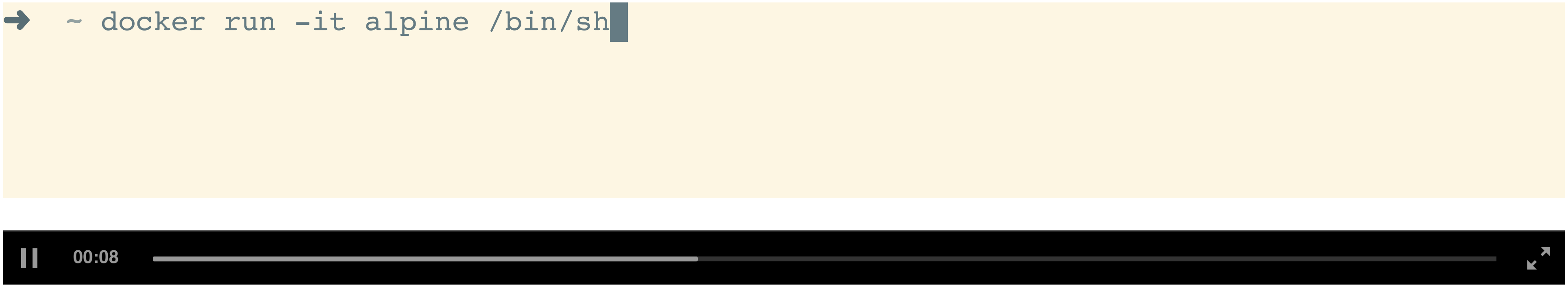


```
➜  ~ docker run alpine /bin/sh
```

- Docker starts alpine image
- Runs shell command
- Exits immediately
- Require `-i -t` flag

Exercise: Start an *interactive* shell

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

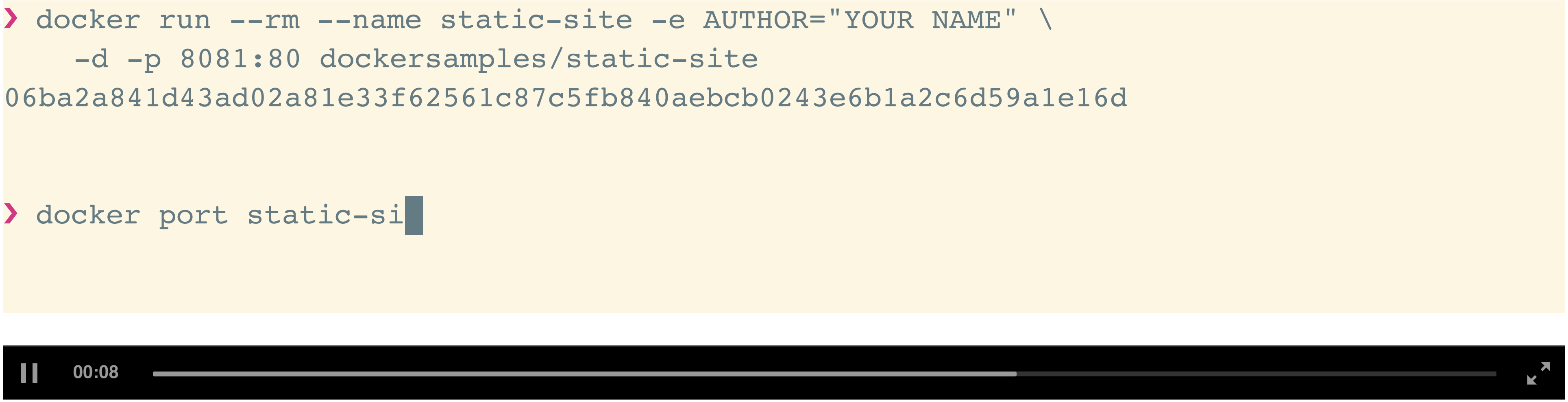


```
➜  ~ docker run -it alpine /bin/sh
```

- Docker starts alpine image
- Runs shell command
- Execute commands inside container

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-si
```

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

General purpose commands

Get command documentation

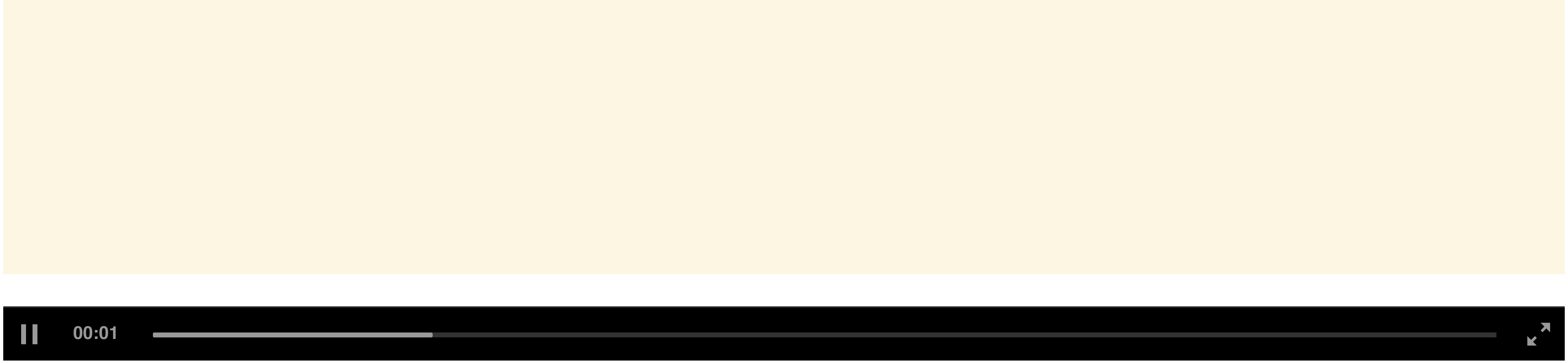
- Just typing `docker` returns list of commands
- Calling any command with `-h` flag displays some docs
- Comprehensive online docs on Docker website

Docker version

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

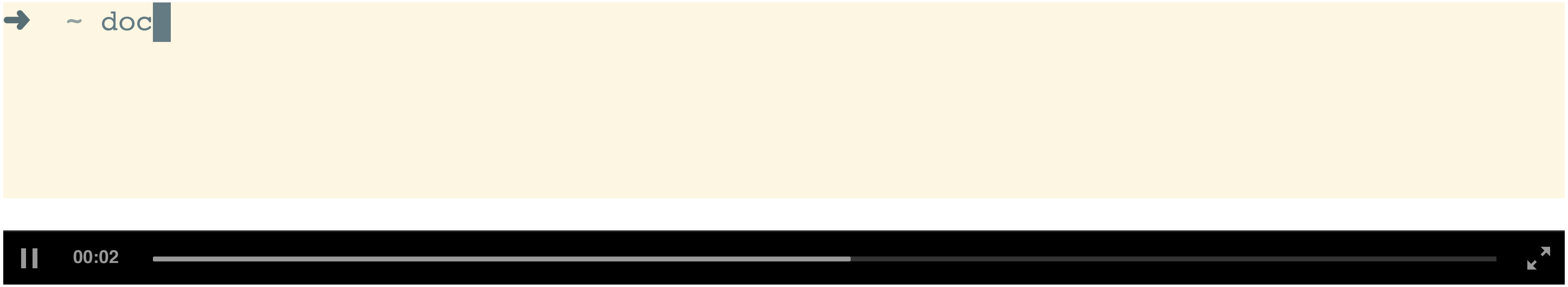
List local images

```
$ docker image ls
```



List running containers

```
$ docker ps
```



```
➜  ~ doc
```

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
-l, --latest           Show the latest created container (includes all sta
    --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)

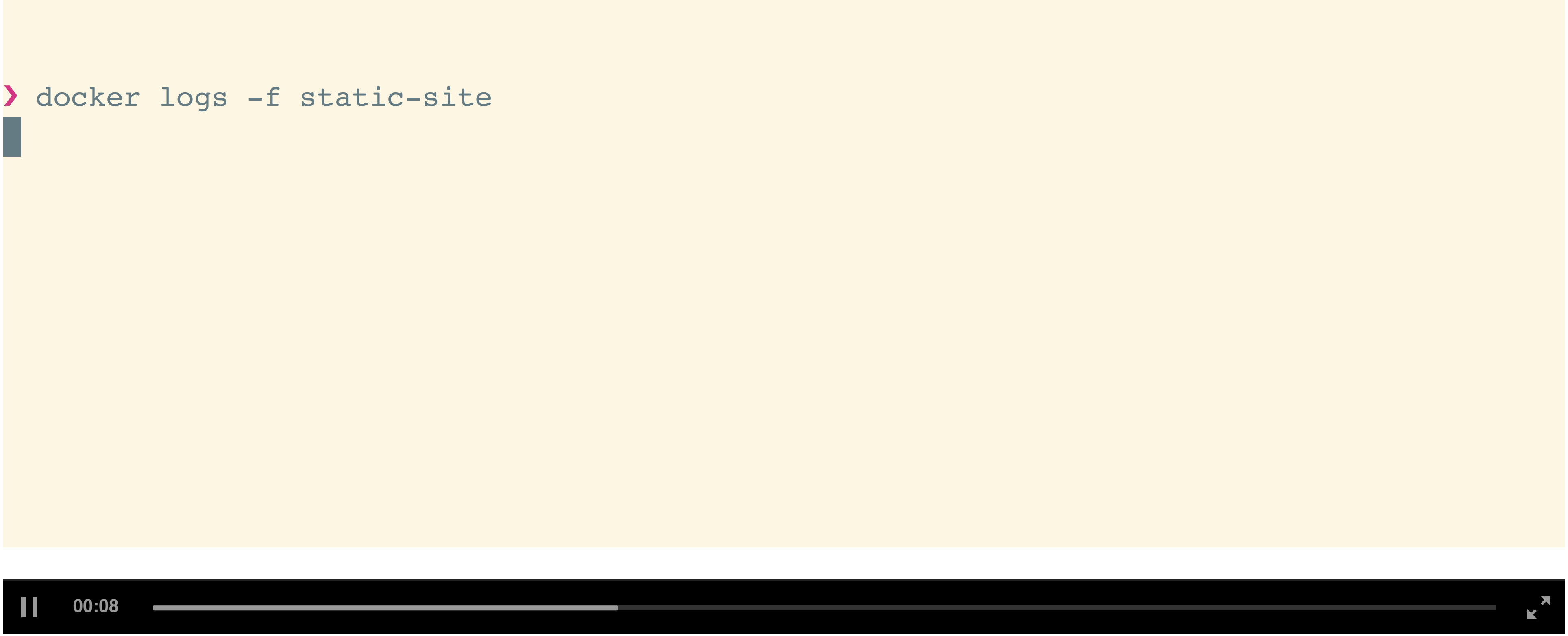
`docker ps -a --filter 'exited=0'`

Filter all containers by exit code

See online documentation

View container logs

```
$ docker logs
```



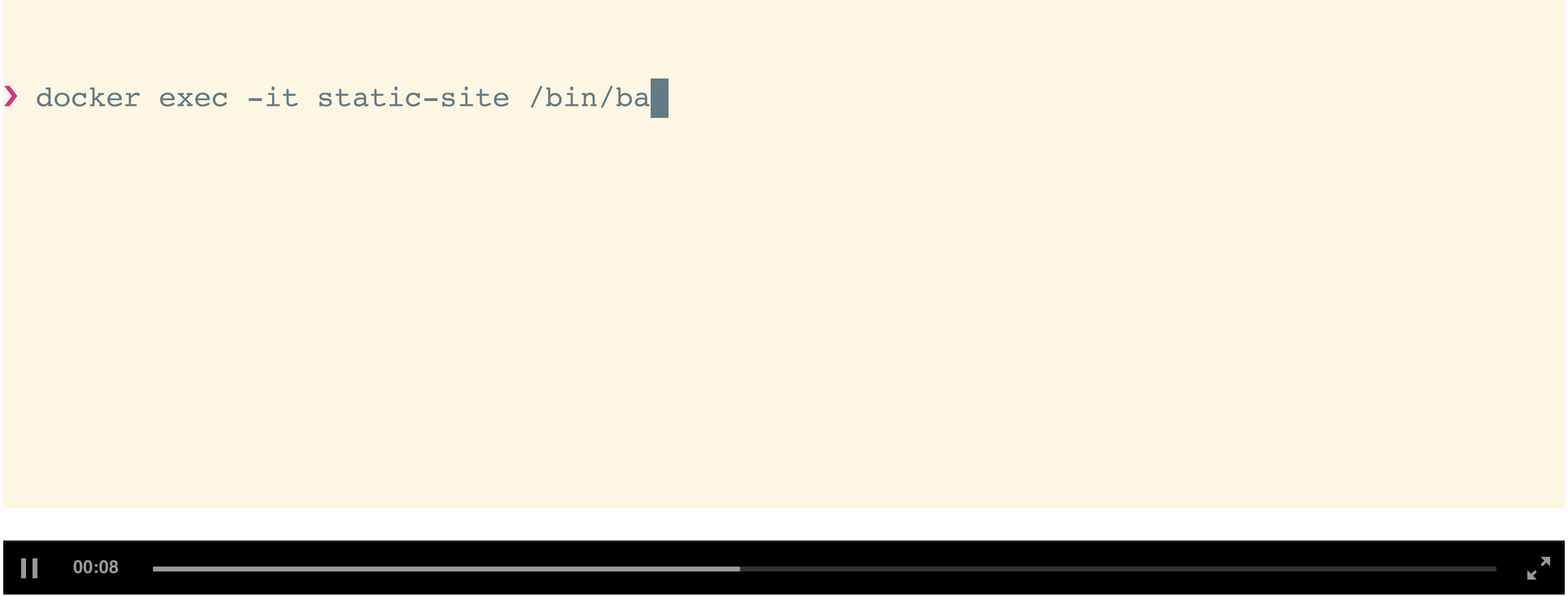
```
❯ docker logs -f static-site
```

- `-f` flag to watch logs in realtime

See online documentation

Enter a running container

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



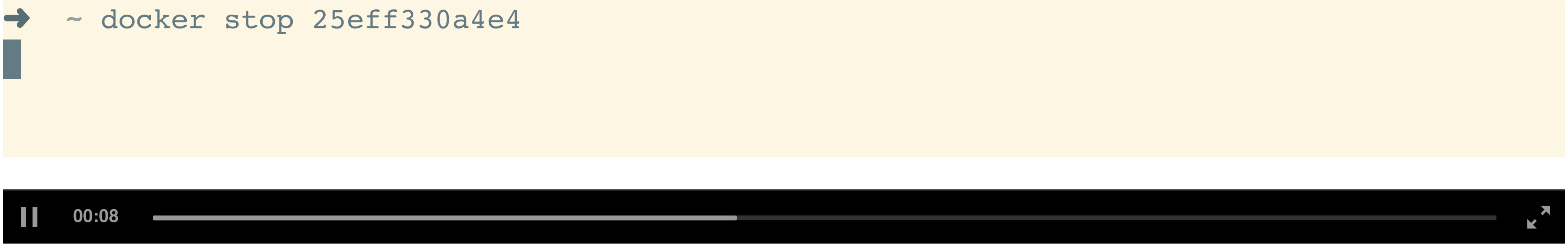
```
❯ docker exec -it static-site /bin/ba
```

- Can be useful for debugging

See online documentation

Exercise: Stop a running container

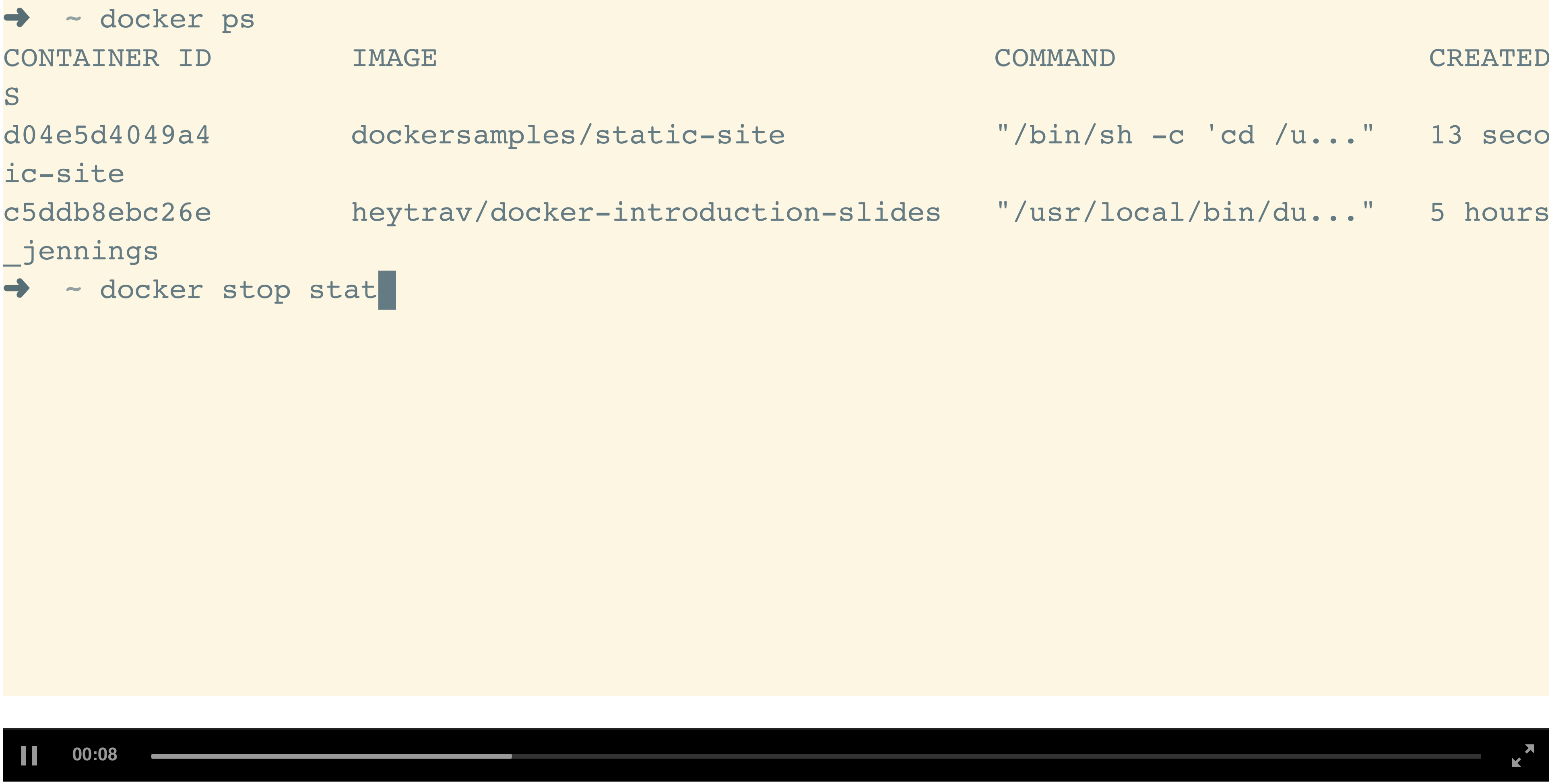
```
$ docker stop <CONTAINER_ID>
```



```
➜  ~ 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 seco
ic-site
c5ddb8ebc26e        heytrav/docker-introduction-slides "/usr/local/bin/du..."   5 hours
_jennings
➜  ~ docker stop stat
```

Docker in a nutshell

Components of Docker

Docker Image

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

IMAGE
READ ONLY

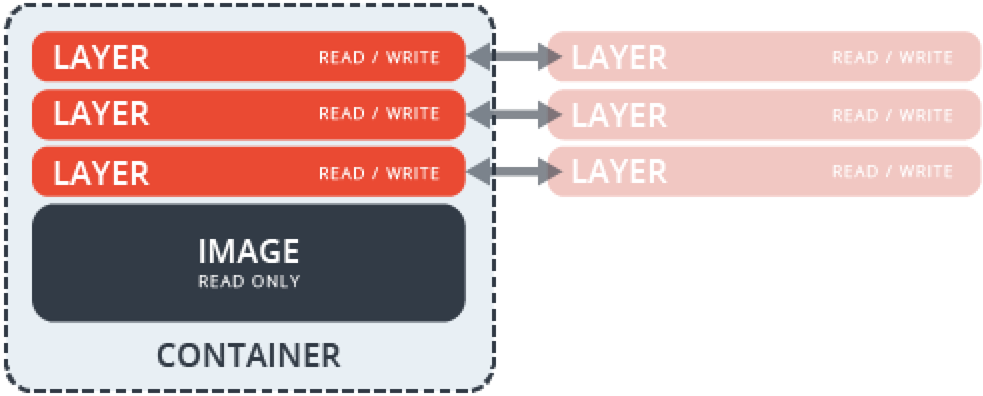
Docker Registry

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

IMAGE READ ONLY	IMAGE READ ONLY	IMAGE READ ONLY	IMAGE READ ONLY
IMAGE READ ONLY	IMAGE READ ONLY	IMAGE READ ONLY	IMAGE READ ONLY
IMAGE READ ONLY	IMAGE READ ONLY	IMAGE READ ONLY	IMAGE READ ONLY

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



LAYER READ / WRITE ⟷ LAYER READ / WRITE

LAYER READ / WRITE ⟷ LAYER READ / WRITE

LAYER READ / WRITE ⟷ LAYER READ / WRITE

IMAGE
READ ONLY

CONTAINER

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

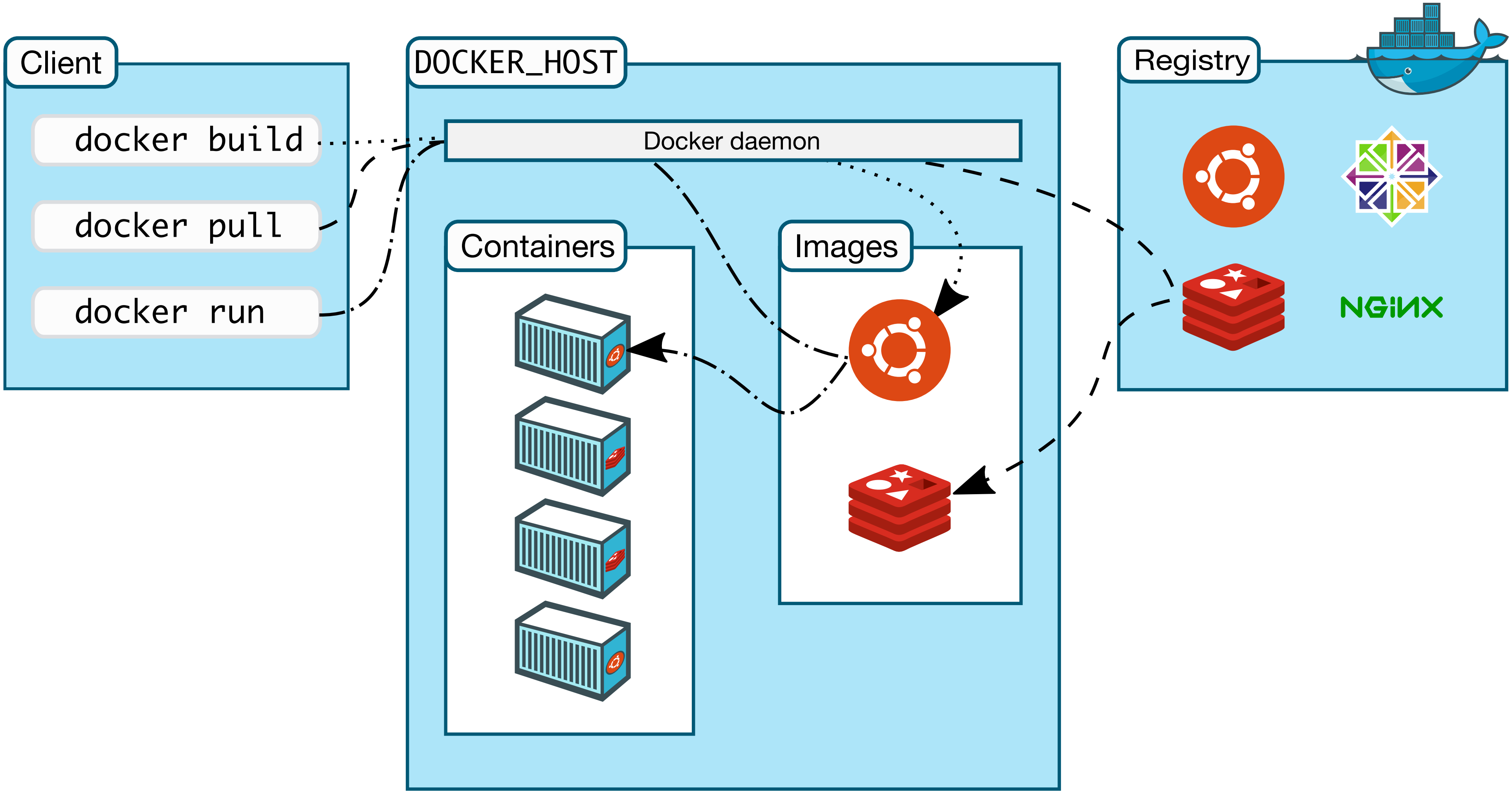
Container format

Combined namespaces, cgroups and UnionFS

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

Create images with a *Dockerfile*

- A text file. Usually named `Dockerfile`
- Sequential instructions for building a Docker image
- From top to bottom, each instruction creates a layer on the previous
- A very simple Dockerfile:

```
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
```

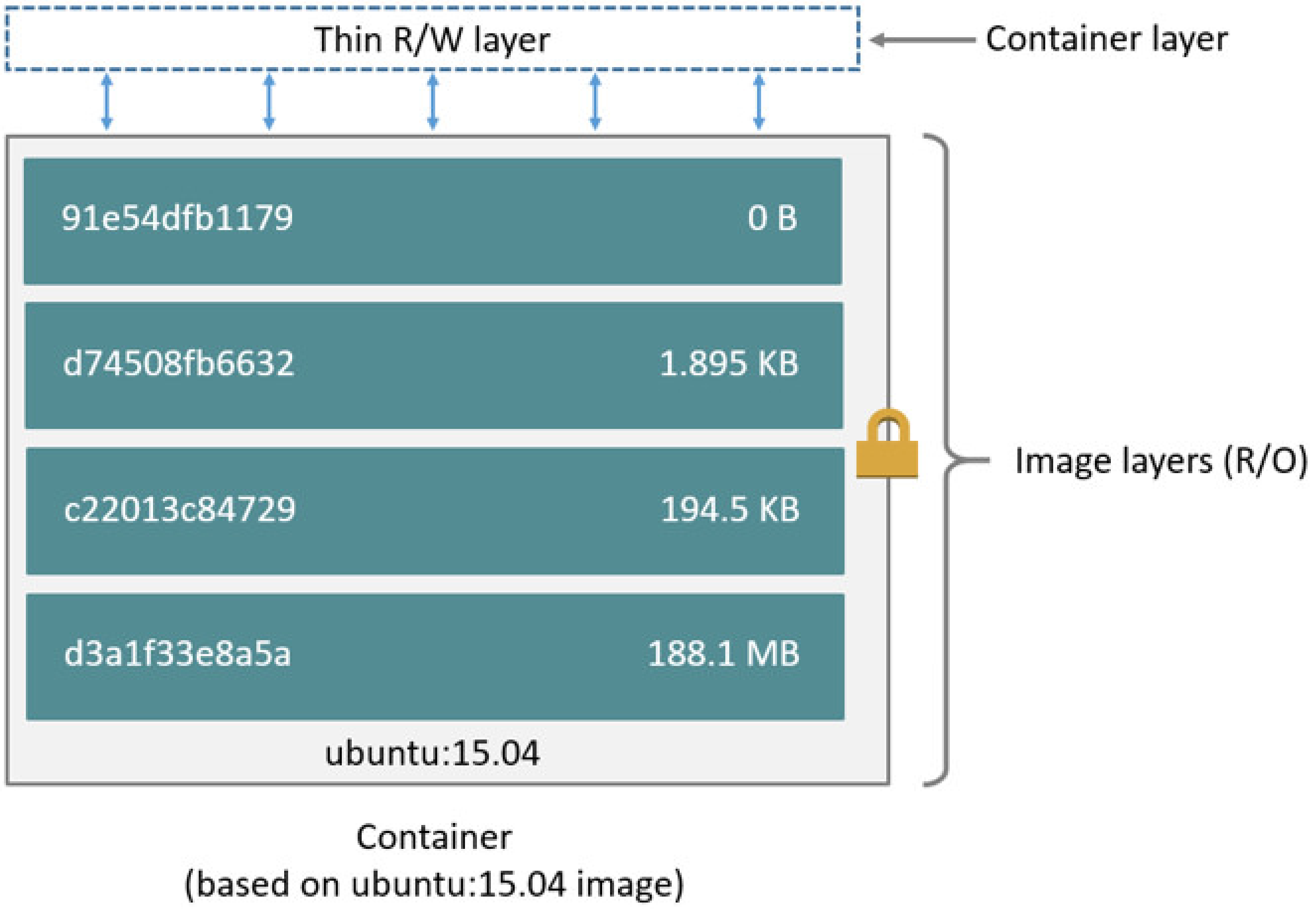
- A number of commands telling docker how to build image

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

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

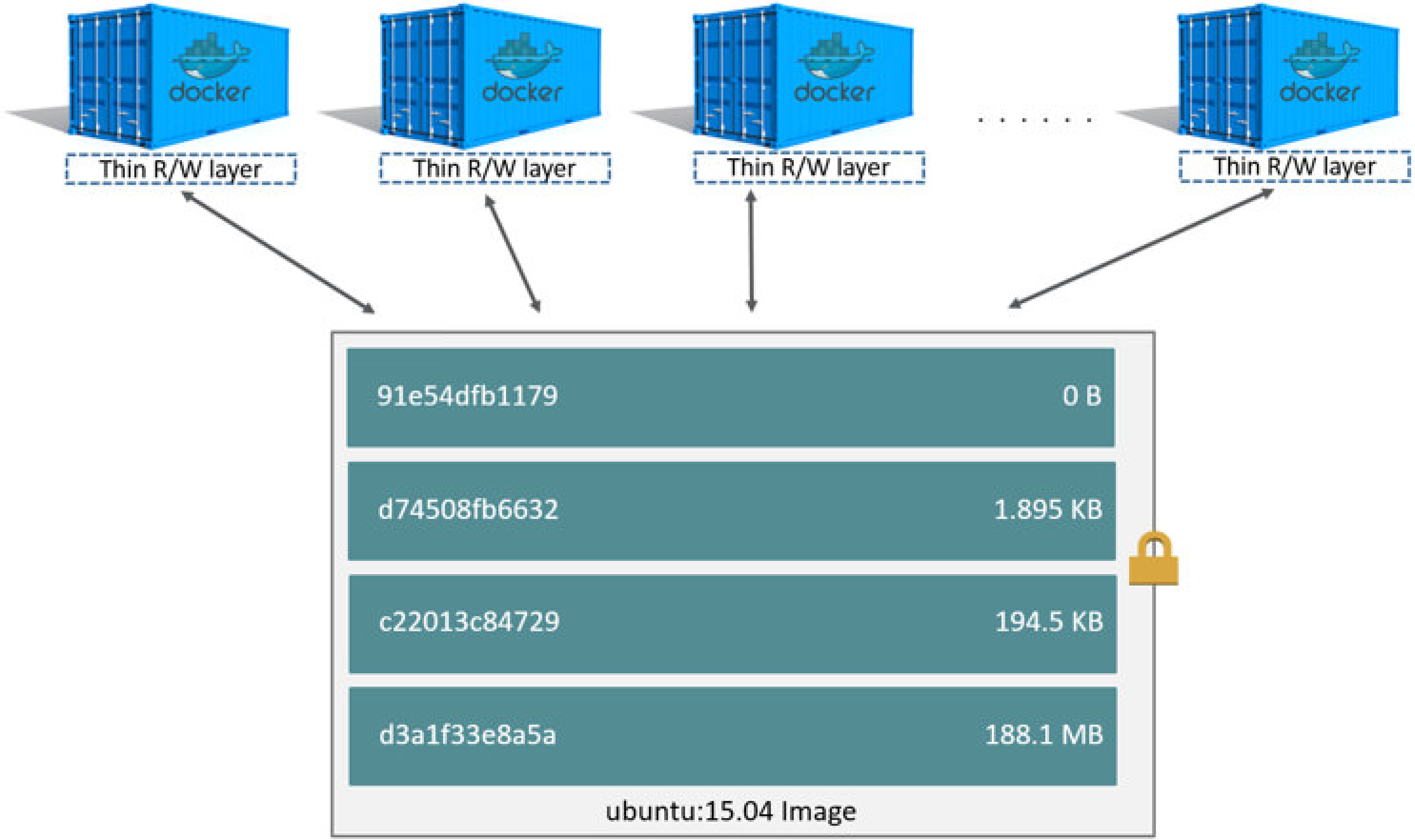
```
CMD python /app/app.py
```

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

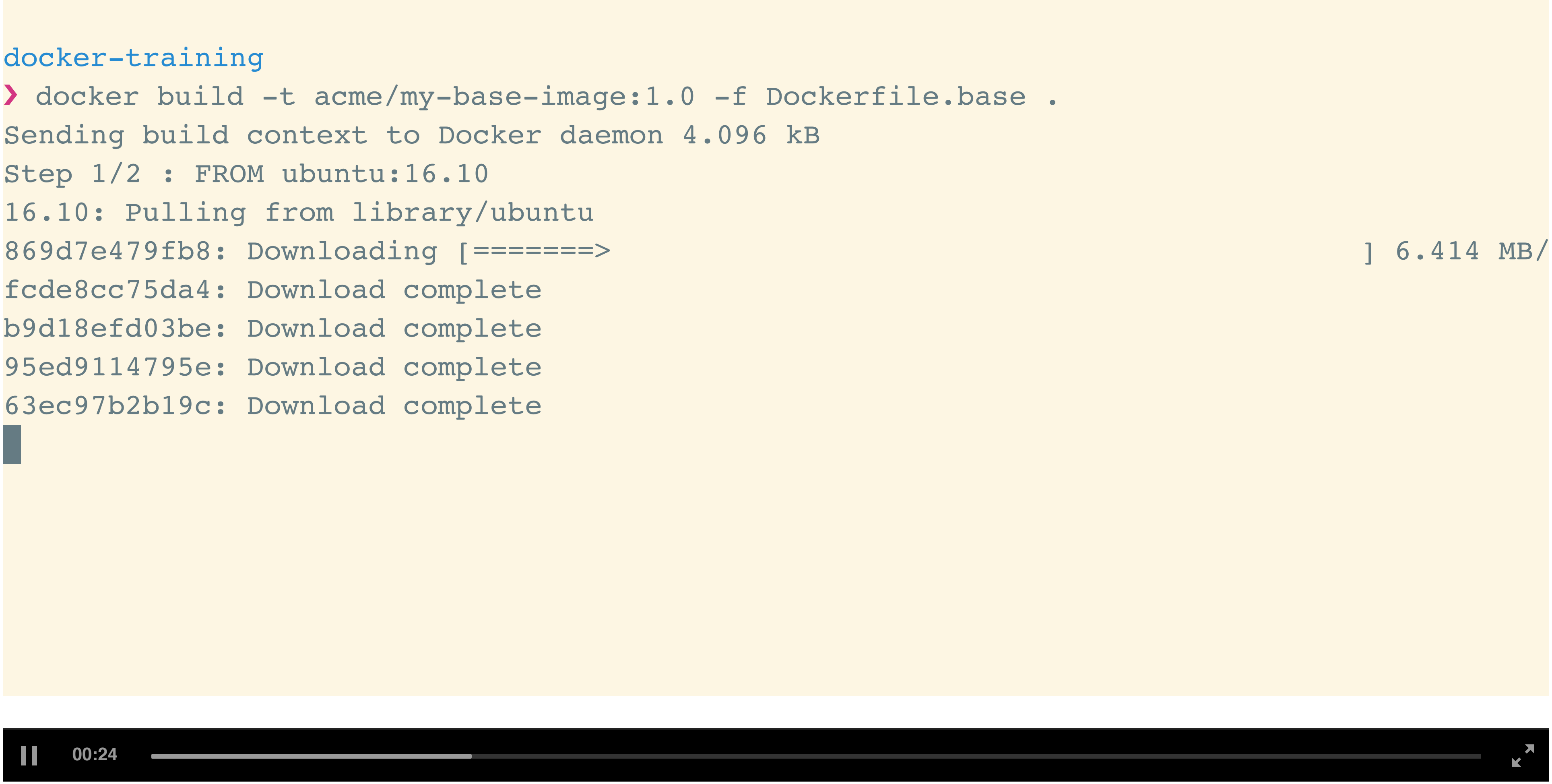
```
FROM acme/my-base-image:1.0
CMD /app/hello.sh
```

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 [=======>                                           ] 6.414 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 .
```

```
docker-training
❯
```

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

Dockerfile directives

FROM

Tell Docker which base image to use (alpine, ubuntu)

WORKDIR

set the working directory (will be created if doesn't exist)

COPY

copy files from build environment into image

Dockerfile directives (continued)

RUN

execute a command (i.e. bash command)

EXPOSE

ports to expose when running

VOLUME

folders to expose when running

CMD/ENTRYPOINT

command to execute when container starts

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

Writing a Dockerfile

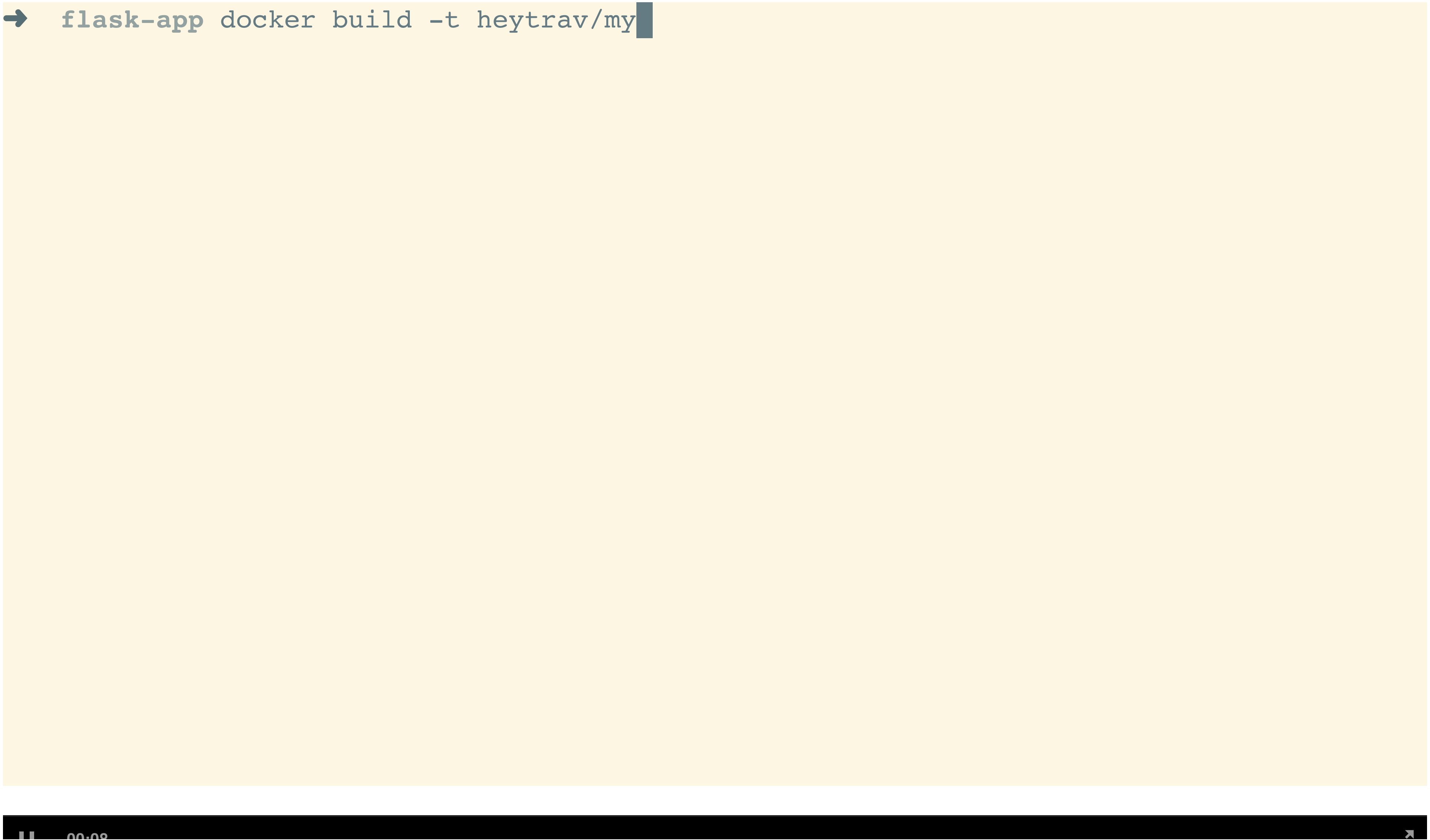
- In the sample-code/flask-app folder
- Start your favourite editor (gedit, vi, emacs, etc.)
- Create a file called Dockerfile

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 a Docker image

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

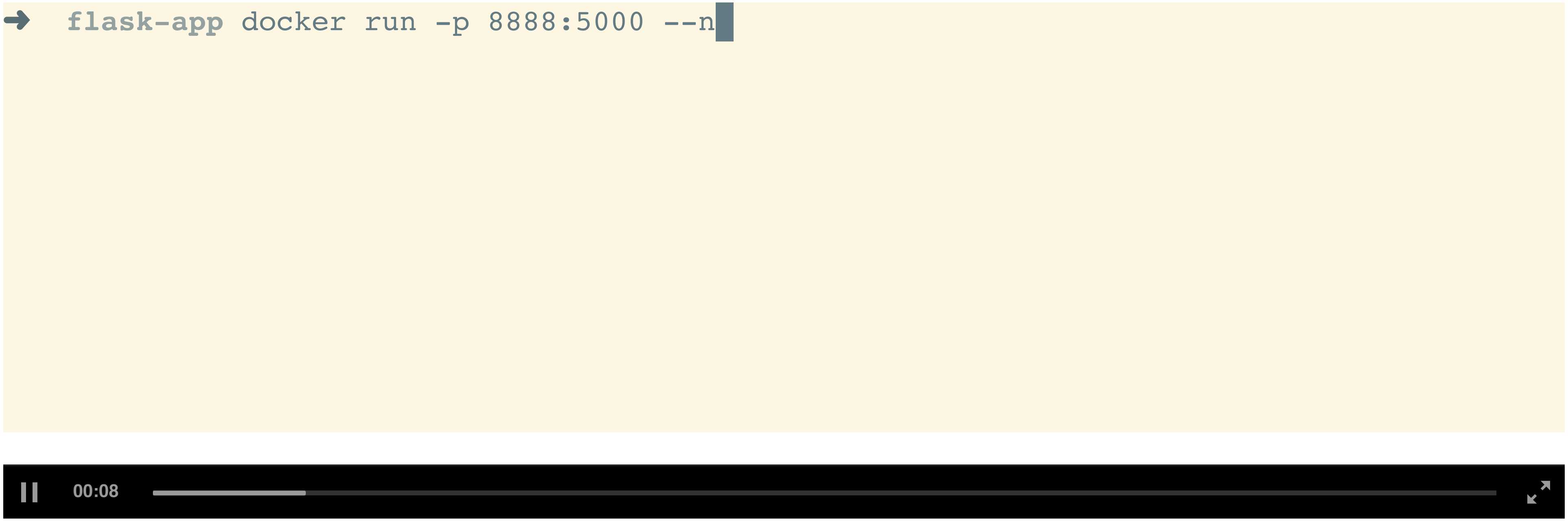


```
➜  flask-app docker build -t heytrav/my
```

Note: please replace YOURNAME with your Docker Hub username

Run your image

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

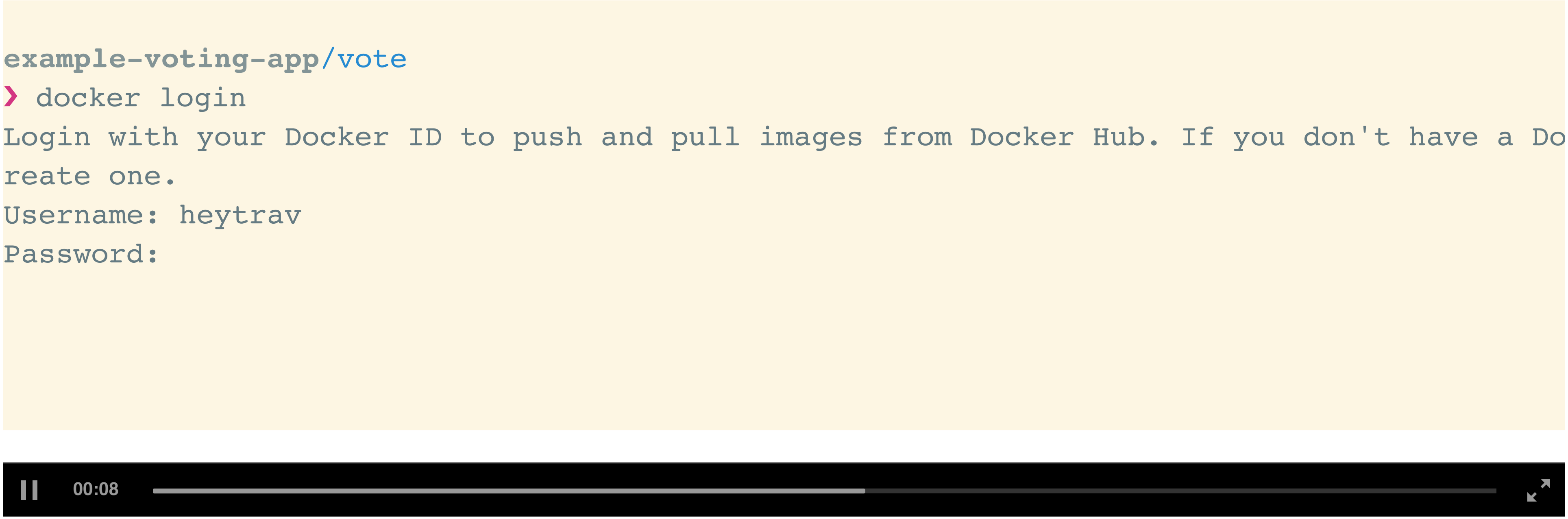


```
➜  flask-app docker run -p 8888:5000 --n
```

...Now open your test webapp

Login to docker

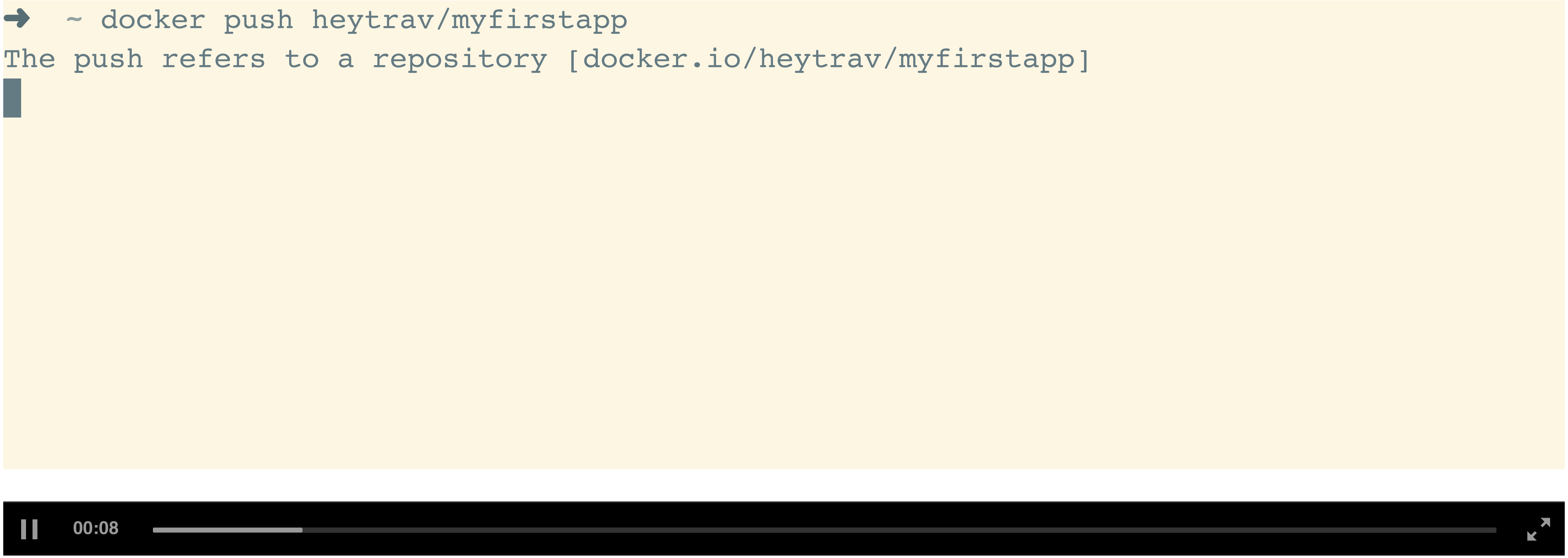
```
$ docker login
```



```
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 Do
reate one.
Username: heytrav
Password:
```

Push our first image

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



```
➜  ~ docker push heytrav/myfirstapp
The push refers to a repository [docker.io/heytrav/myfirstapp]
```

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
- Minimize 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

Minimising layers

```
$ docker pull heytrav/example-1
$ docker pull heytrav/example-2
$ docker pull heytrav/example-3
$ docker image ls | grep example
$ docker history heytrav/example-3
$ docker history heytrav/example-2
$ docker history heytrav/example-1
```

example-1

Remove cache files in same layer

example-2

Remove cache files in next layer

example-3

Install libs without removing cache

ADD vs COPY

```
ADD somefile.tar.gz /                      # add tar into directory
ADD . /usr/path/                           # copy file to directory
ADD http://domain.com/some/file.txt .      # download file from web
ADD http://domain.com/some/file.tar.gz . # same thing (doesn't unpack tar
COPY . /usr/local/bin                      # copy files to /usr/local/bin
```

- Both *copy* things into image
- ADD does a bit of magic:
  - unpacks tars
  - downloads files
  - copies files
- ADD considered inconsistent and not transparent
- Recommend to only use COPY and never ADD

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

Multi-container Applications as Microservices

Microservices

- A suite of small services, each running in its own process and communicating with lightweight mechanisms
- Services can be implemented with entirely different stacks
- Services can be developed, maintained, and updated independently
- Often use well defined REST APIs to communicate
- If a message bus/queue is used then its likely to be lightweight (ZeroMQ vs ESB)

Build a voting app

```
$ cd ~/example-voting-app
```

- Python web application
- Redis queue
- .NET worker
- Postgres DB with a data volume
- Node.js app to show votes in real time
- Clone the repo and change directories into it

Defining multi container apps

- Microservice architecture encourages apps run in isolation
  - Webapp
  - Database
  - Nginx
  - Backend code
- Easier to scale components
- Easier to replace components

Running *linked* containers.

```
$ docker run -d --name="database" -v ~/data:/var/lib/data postgres:9.3
$ docker run -d --name="redis" redis
$ docker run -d --link database:database --link redis:redis \
      -p 8000:8000 YOURNAME/pythonwebapp
```

- `--name` flag to specify name of container
- `--link` flag to tell docker to bridge two or more containers
- This works, but gets very unwieldy as number of apps and interactions increases.

Docker Compose file

- See example from repo
- A `yaml` file that specifies
  - which services to run
  - how to network them
  - mount file volumes
  - ..and a lot more

Running services in Swarm Mode

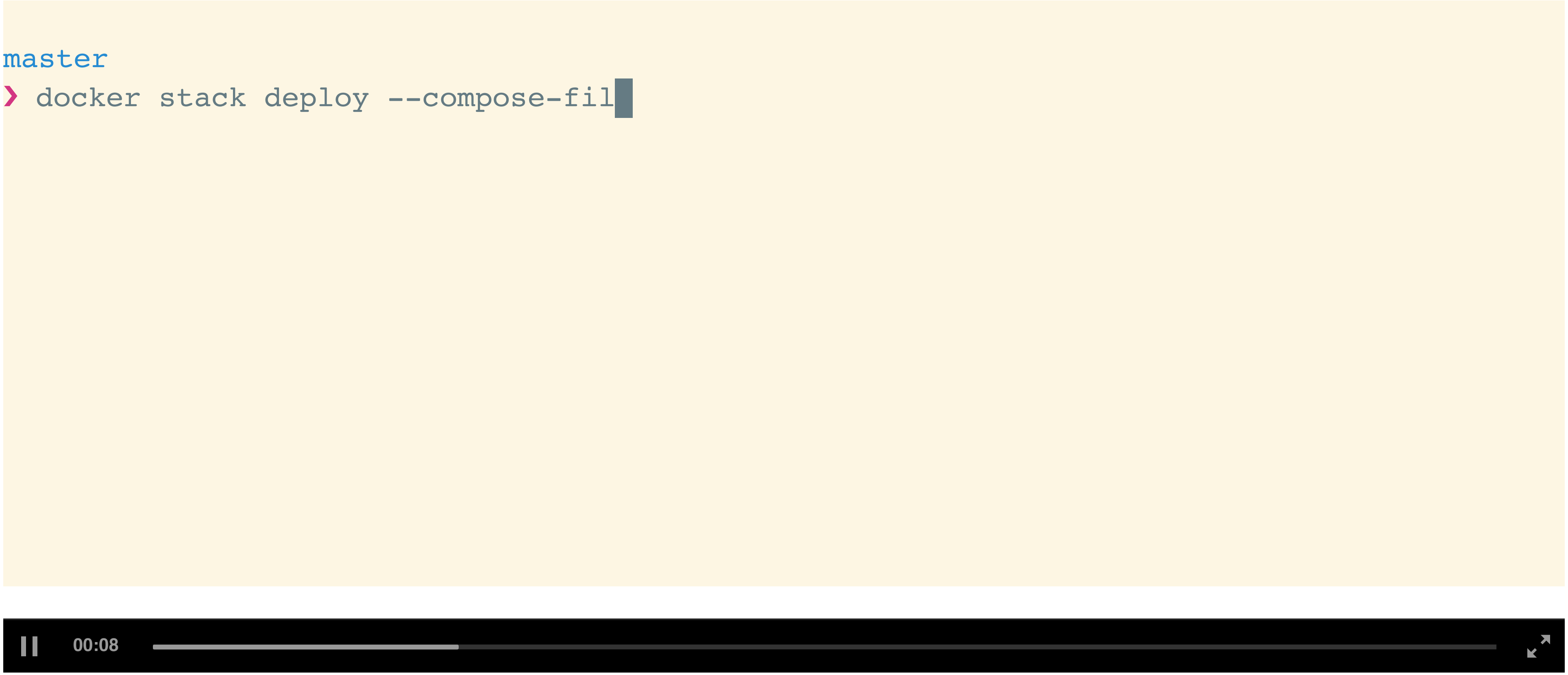
- Swarm Mode added to Docker in 1.12
- Cluster management and orchestration features embedded in Docker
- Can run applications on one or more machines

Initiate a Swarm

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

Deploy the stack

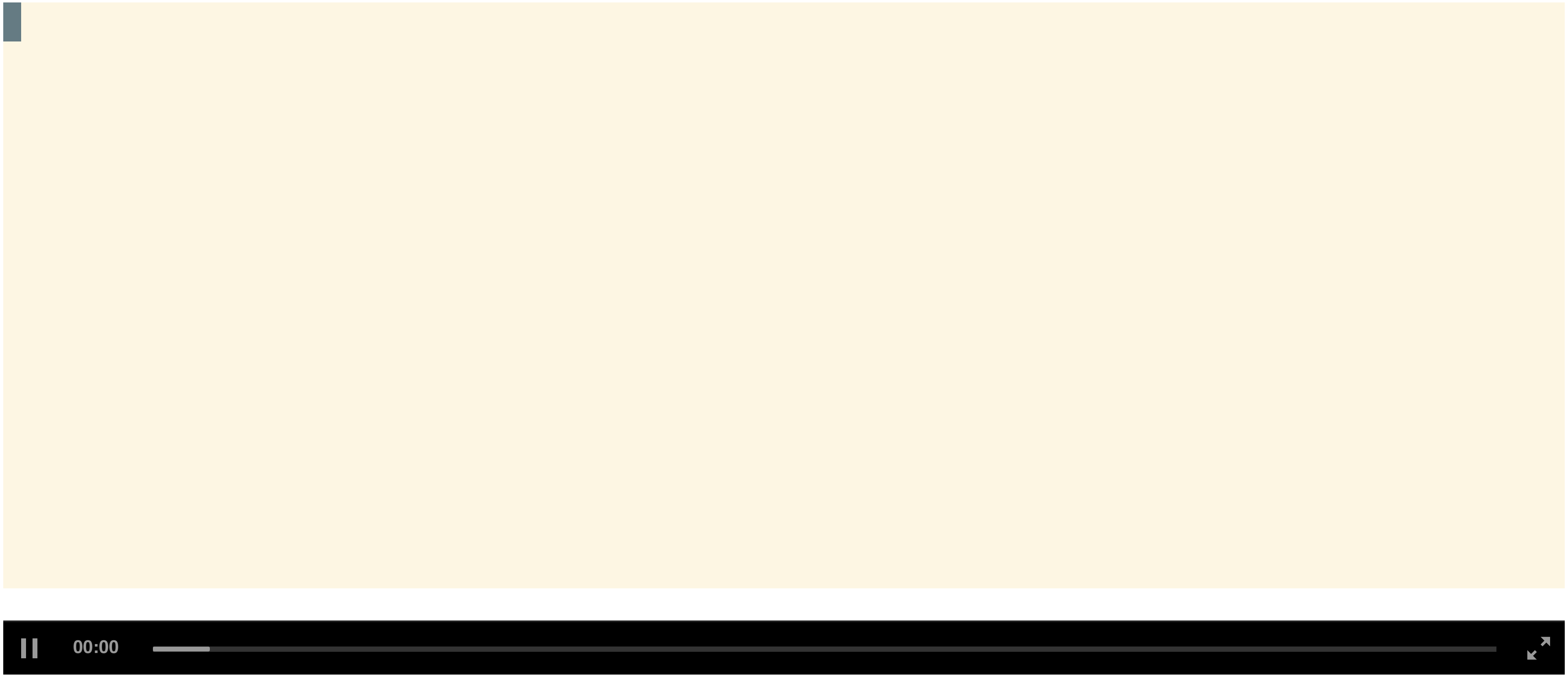
```
$ docker stack deploy --compose-file docker-stack.yml vote
```



```
master
❯ docker stack deploy --compose-fil
```

Verify stack is running

```
$ docker stack ps vote
```

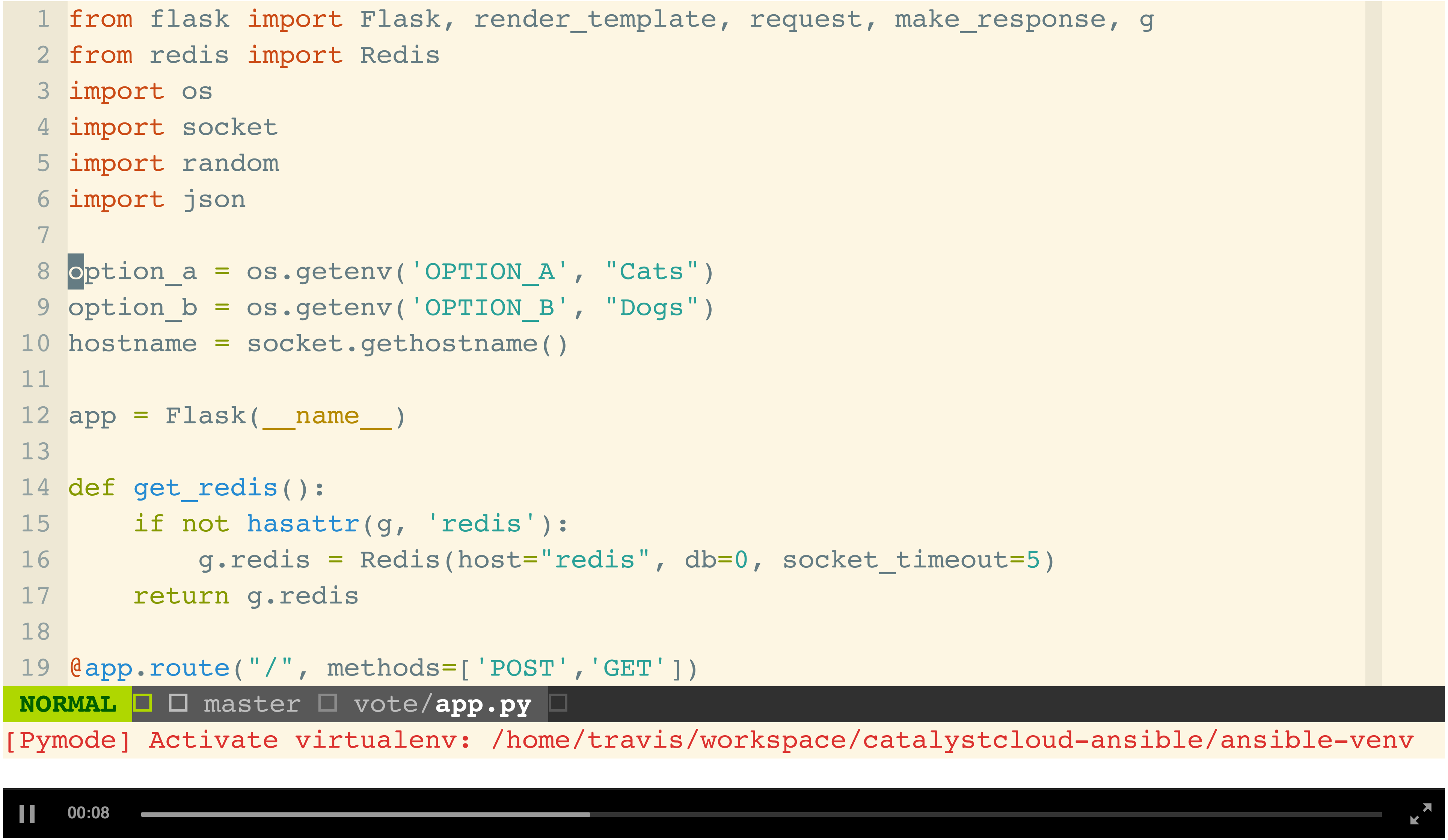


Now, let's go vote! When you're done, have a look at the results.

Modify vote app

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

Change vote options



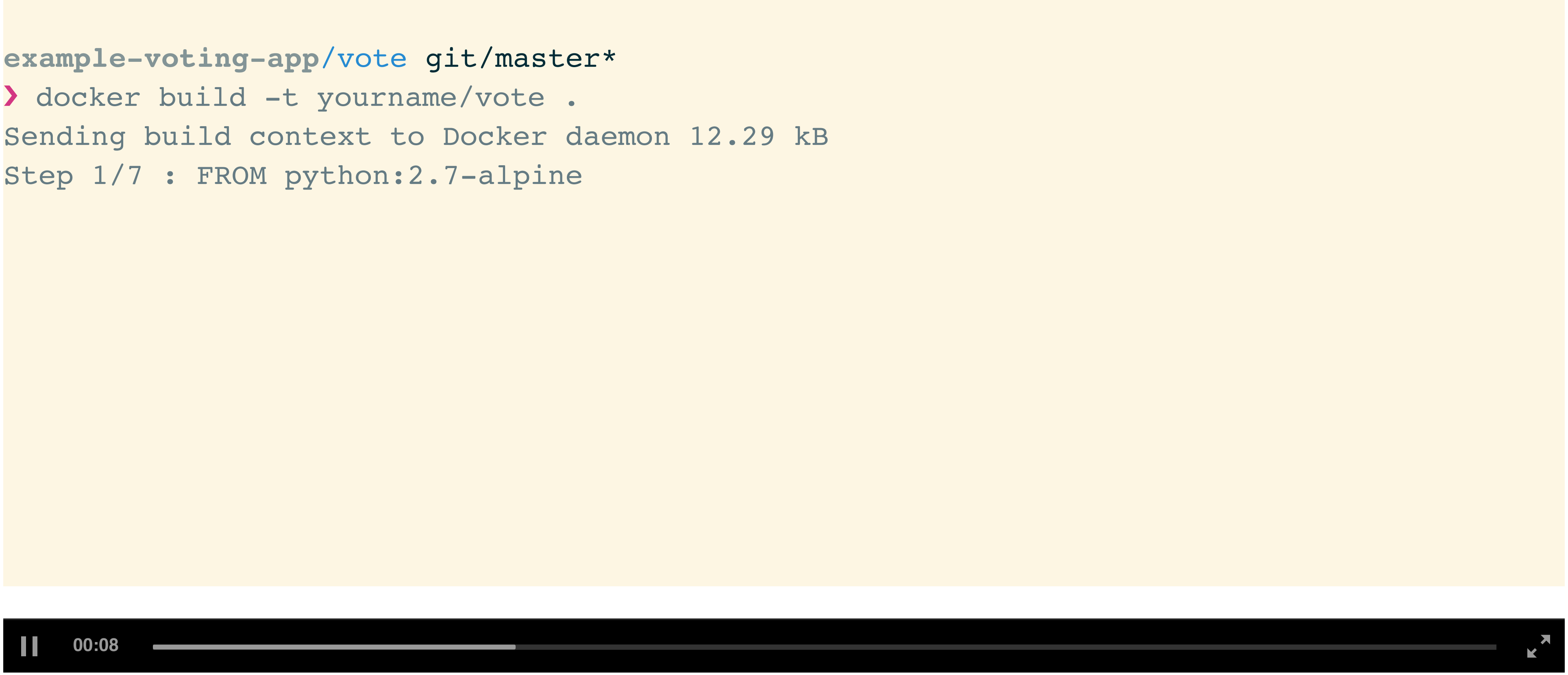
```
from flask import Flask, render_template, request, make_response, g
from redis import Redis
import os
import socket
import random
import json

option_a = os.getenv('OPTION_A', "Cats")
option_b = os.getenv('OPTION_B', "Dogs")
hostname = socket.gethostname()

app = Flask(__name__)

def get_redis():
    if not hasattr(g, 'redis'):
        g.redis = Redis(host="redis", db=0, socket_timeout=5)
    return g.redis

@app.route("/", methods=['POST','GET'])
```

```
NORMAL □ □ master □ vote/app.py □
[Pymode] Activate virtualenv: /home/travis/workspace/catalystcloud-ansible/ansible-venv
```

Build image

In example-voting-app...

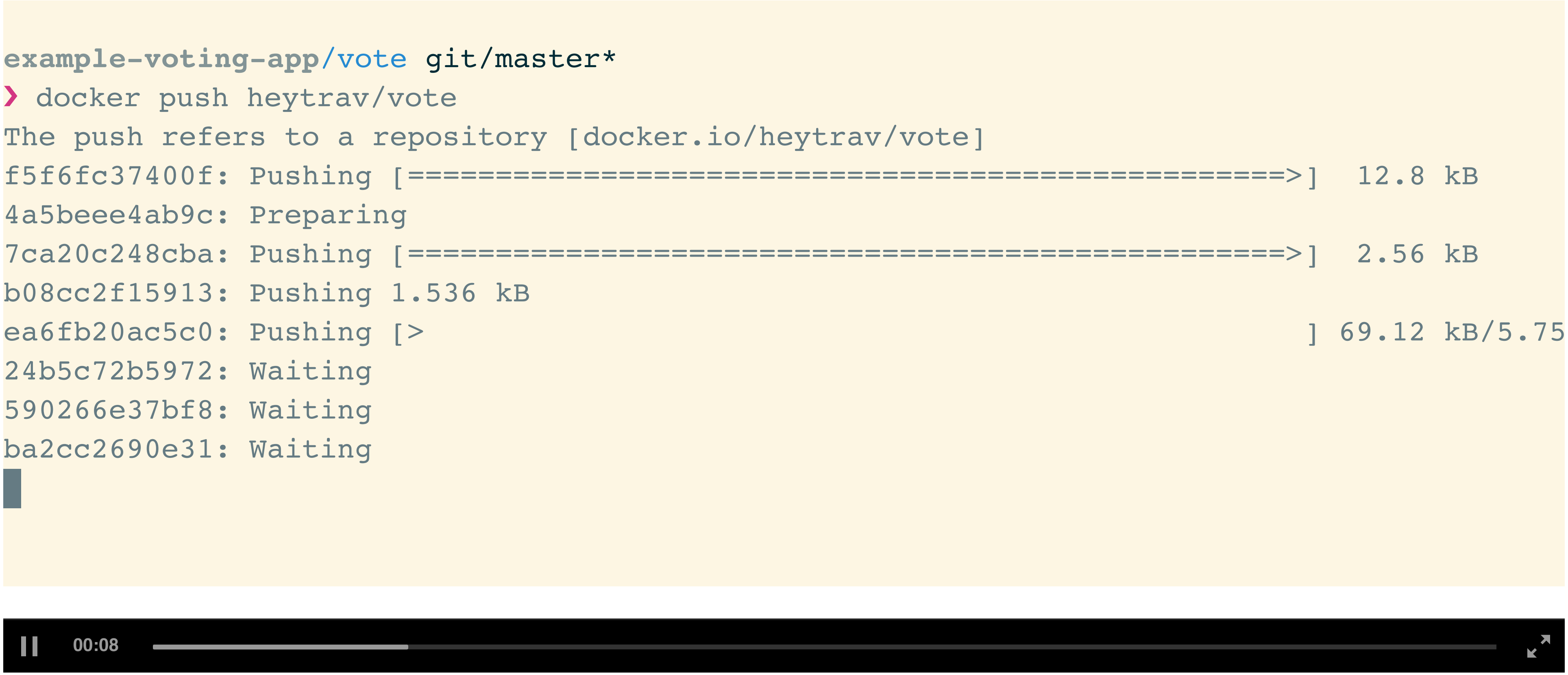
```
$ docker build -t YOURNAME/vote vote
```



```
example-voting-app/vote git/master*
❯ docker build -t yourname/vote .
Sending build context to Docker daemon 12.29 kB
Step 1/7 : FROM python:2.7-alpine
```

Push changes to Docker Hub

```
$ docker push YOURNAME/vote
```



```
example-voting-app/vote git/master*
❯ docker push heytrav/vote
The push refers to a repository [docker.io/heytrav/vote]
f5f6fc37400f: Pushing [=================================================>]  12.8 kB
4a5beee4ab9c: Preparing
7ca20c248cba: Pushing [=================================================>]  2.56 kB
b08cc2f15913: Pushing 1.536 kB
ea6fb20ac5c0: Pushing [>                                                  ] 69.12 kB/5.75
24b5c72b5972: Waiting
590266e37bf8: Waiting
ba2cc2690e31: Waiting
```

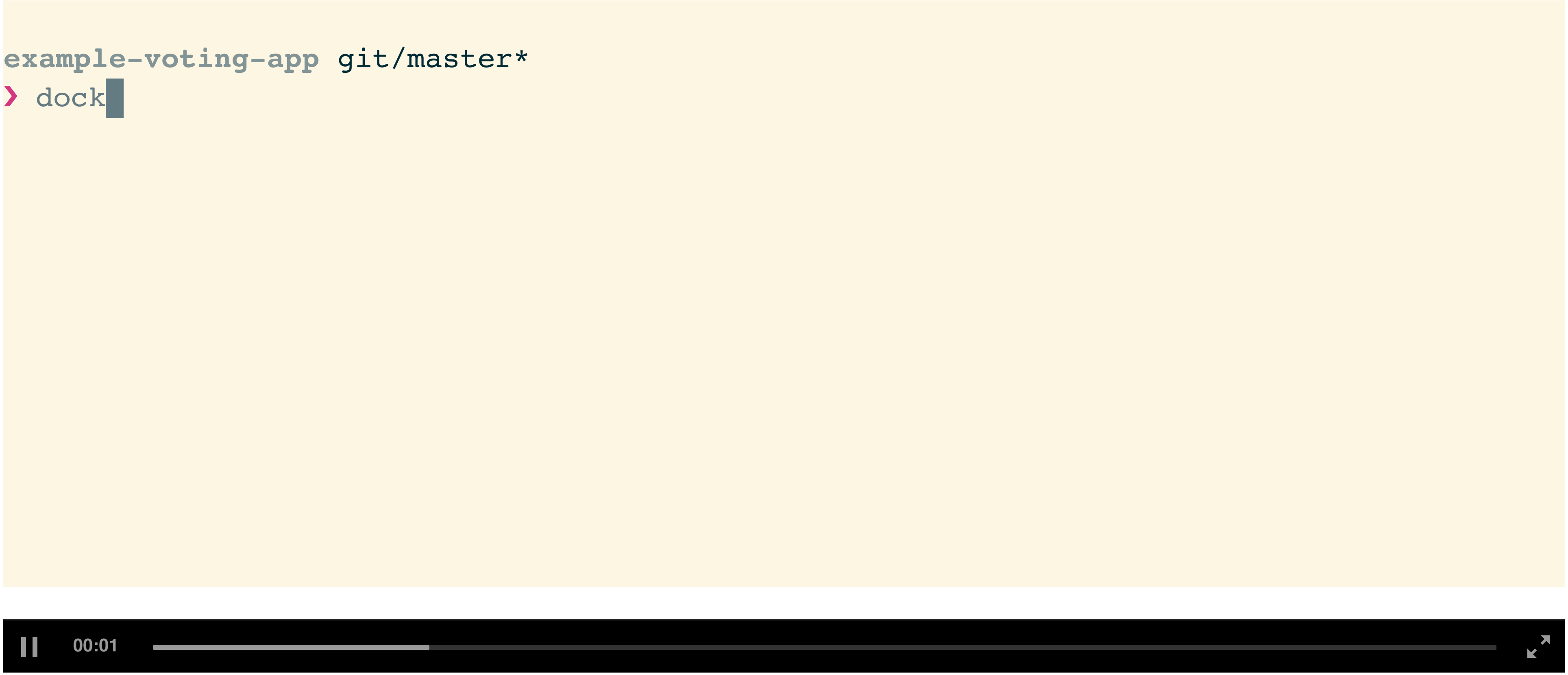
Update a service

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

Now go to the voting app and see what changed

Remove Swarm Stack

```
$ docker stack rm vote
```



```
example-voting-app git/master*
❯ dock
```

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

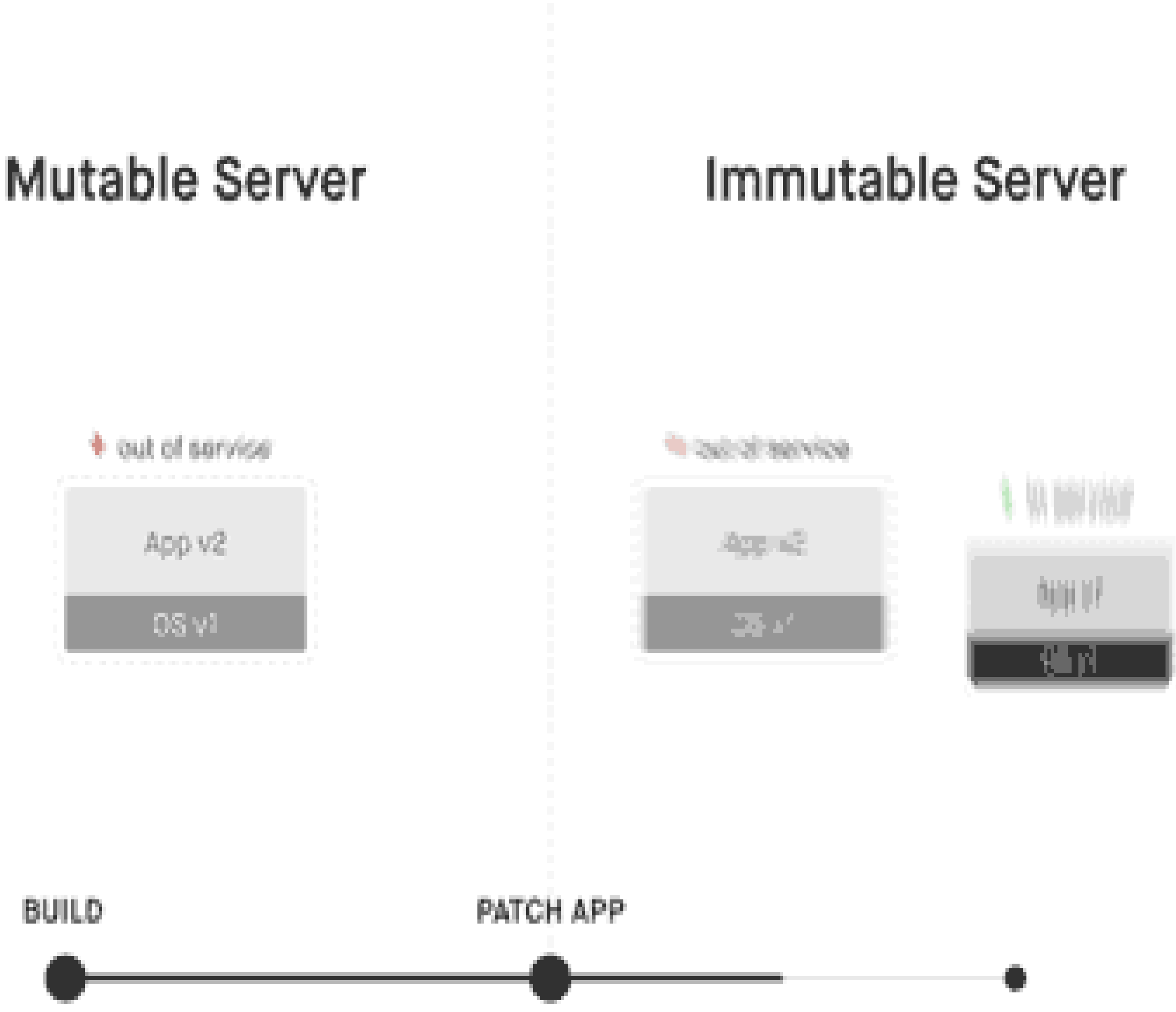
Some Concepts

- Buzzwords ahead!
- 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
- Procedural vs Declarative

Immutable Architecture



Deploying to Catalyst Cloud

- Deploy the voting app to Catalyst Cloud
- Source the RC file

```
$ source ~/os-training.catalyst.net.nz-openrc.sh
```

Introducing Ansible

- DevOps swiss army knife
- Python based tool set
- Lots of uses:
  - server/cluster management
  - deploy code
  - install packages
- Follow instructions for
  - installing ansible
  - activating python venv

Where we should be now..

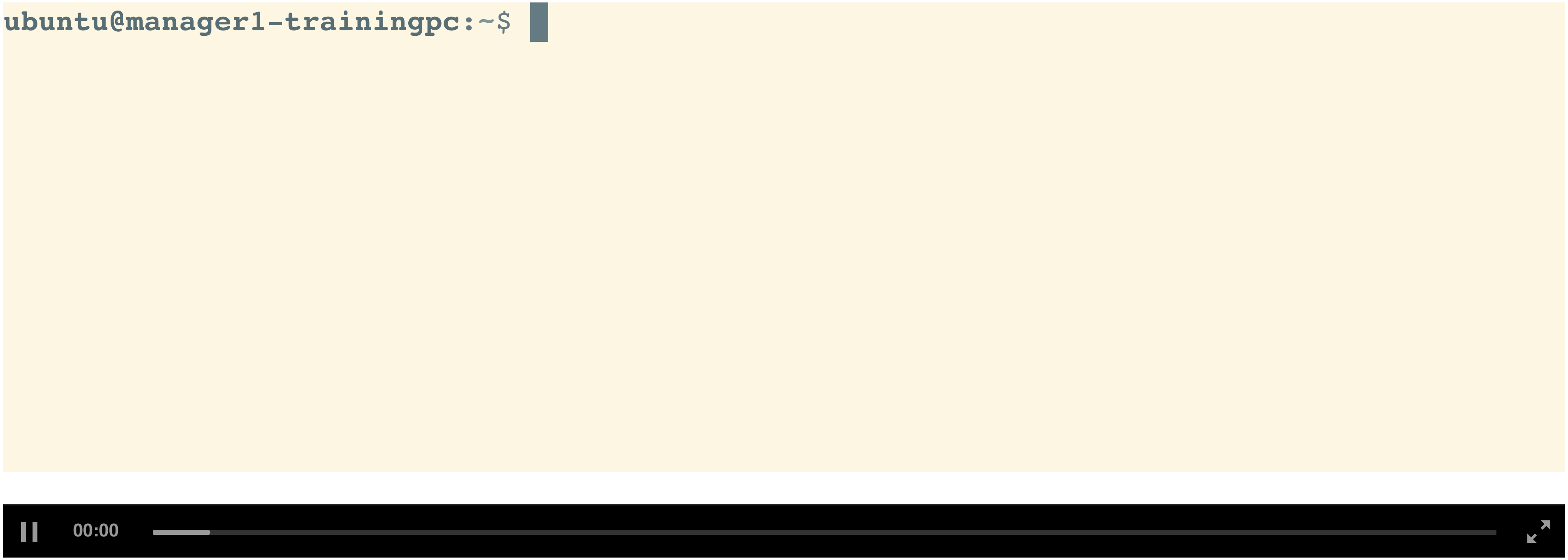
- In your home directory you should have
  - a folder named *docker-introduction*
  - a folder named *example-voting-app*
  - a folder named *catalystcloud-ansible*
- Should have downloaded `os-training.catalyst.net.nz-openrc.sh` (somewhere)
- Should have ansible installed
- Should have activated and sourced a python virtualenv
  - `(ansible-env)` should appear at beginning of line in your shell

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
```



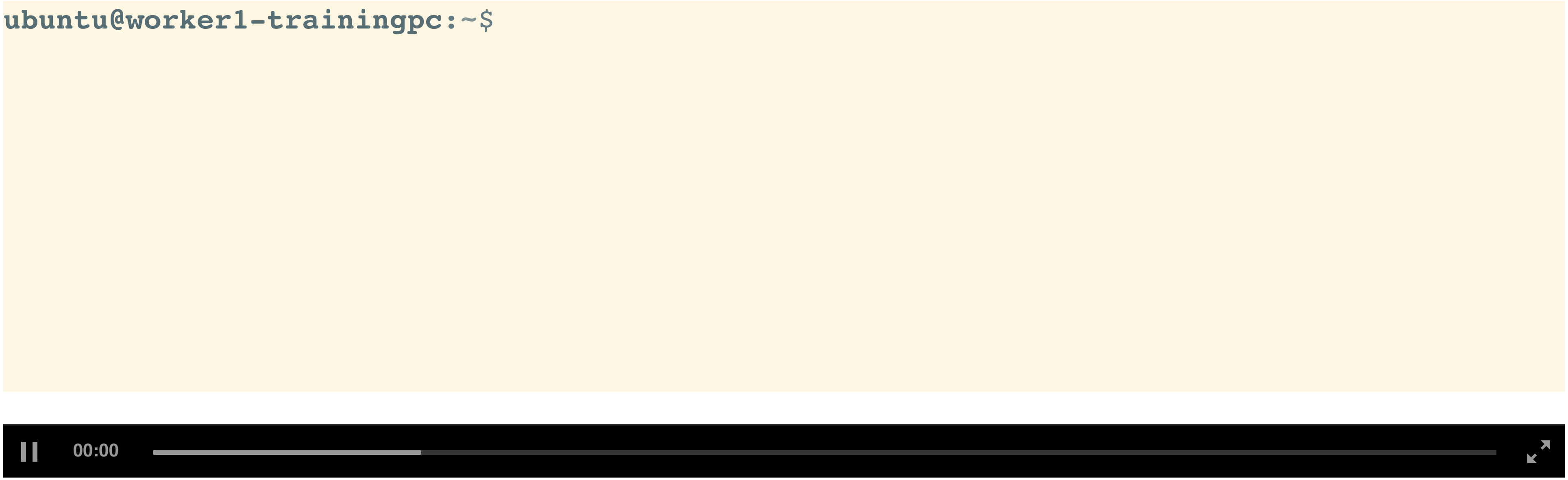
```
ubuntu@manager1-trainingpc:~$
```

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
```

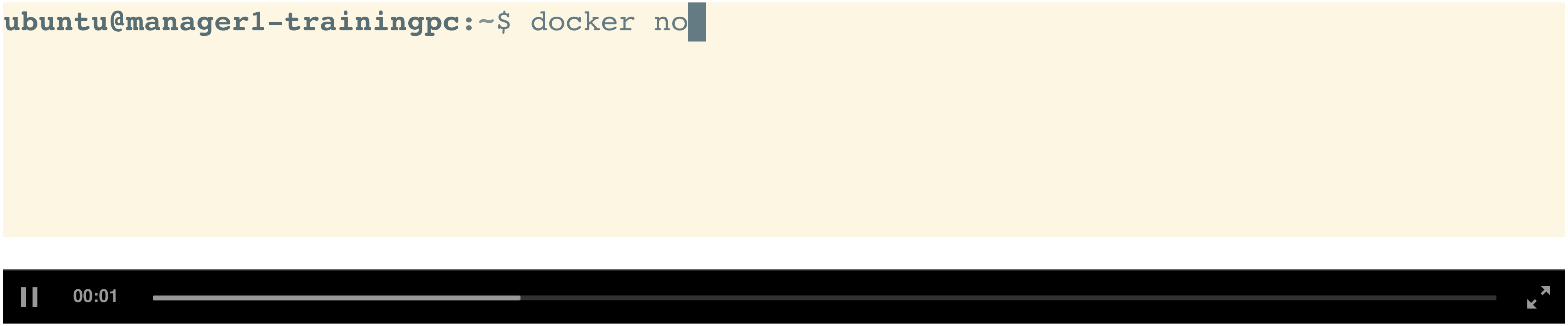


```
ubuntu@worker1-trainingpc:~$
```

Repeat this for `worker2`

Check nodes

```
$ docker node ls
```



```
ubuntu@manager1-trainingpc:~$ docker no
```

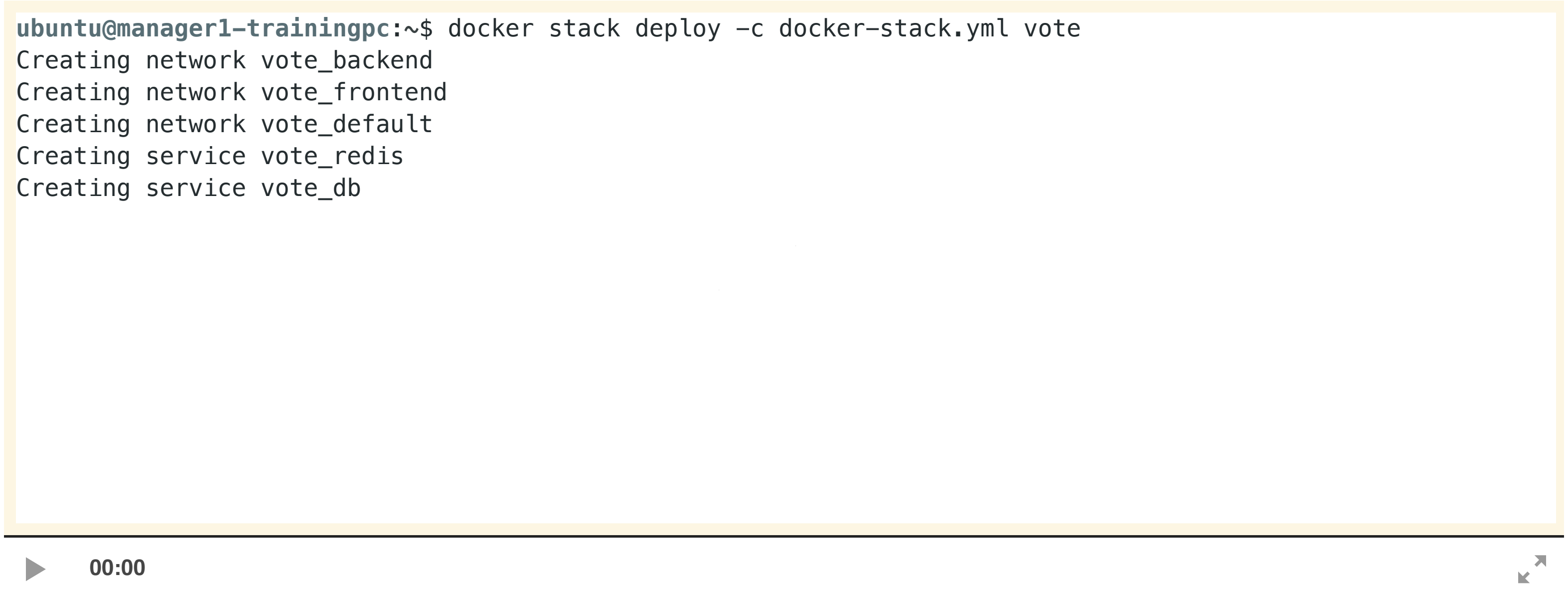
Voting app (again)

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
```



```
ubuntu@manager1-trainingpc:~$ docker stack deploy -c docker-stack.yml vote
Creating network vote_backend
Creating network vote_frontend
Creating network vote_default
Creating service vote_redis
Creating service vote_db
```

Powered by asciinema

Monitor deploy progress

```
$ docker stack ps vote
```

```
ubuntu@manager1-trainingpc:~$
```

```
$ docker service ls
```

Let's Vote!

Vote

To vote

Results

To see results

Visualizer

To visualise running containers

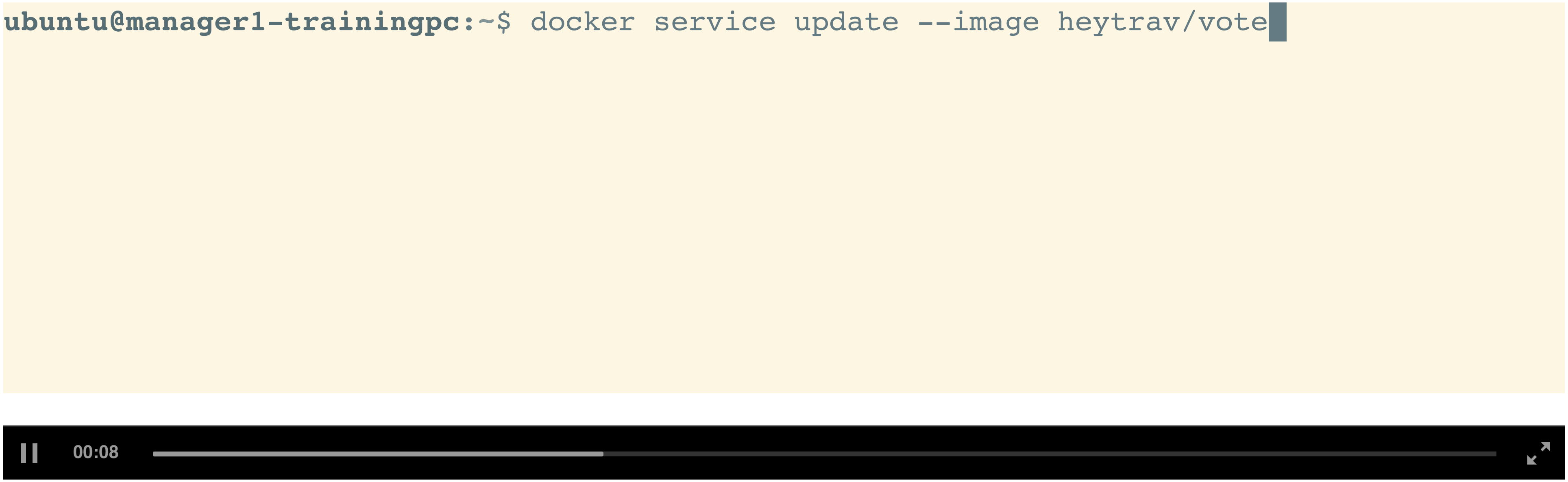
Scale services

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

Look at the changes in the visualizer

Update a service

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



```
ubuntu@manager1-trainingpc:~$ docker service update --image heytrav/vote
```

Now go to the voting app and verify the change

Drain a node

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

- Sometimes necessary to take host offline
  - Planned maintenance
- 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

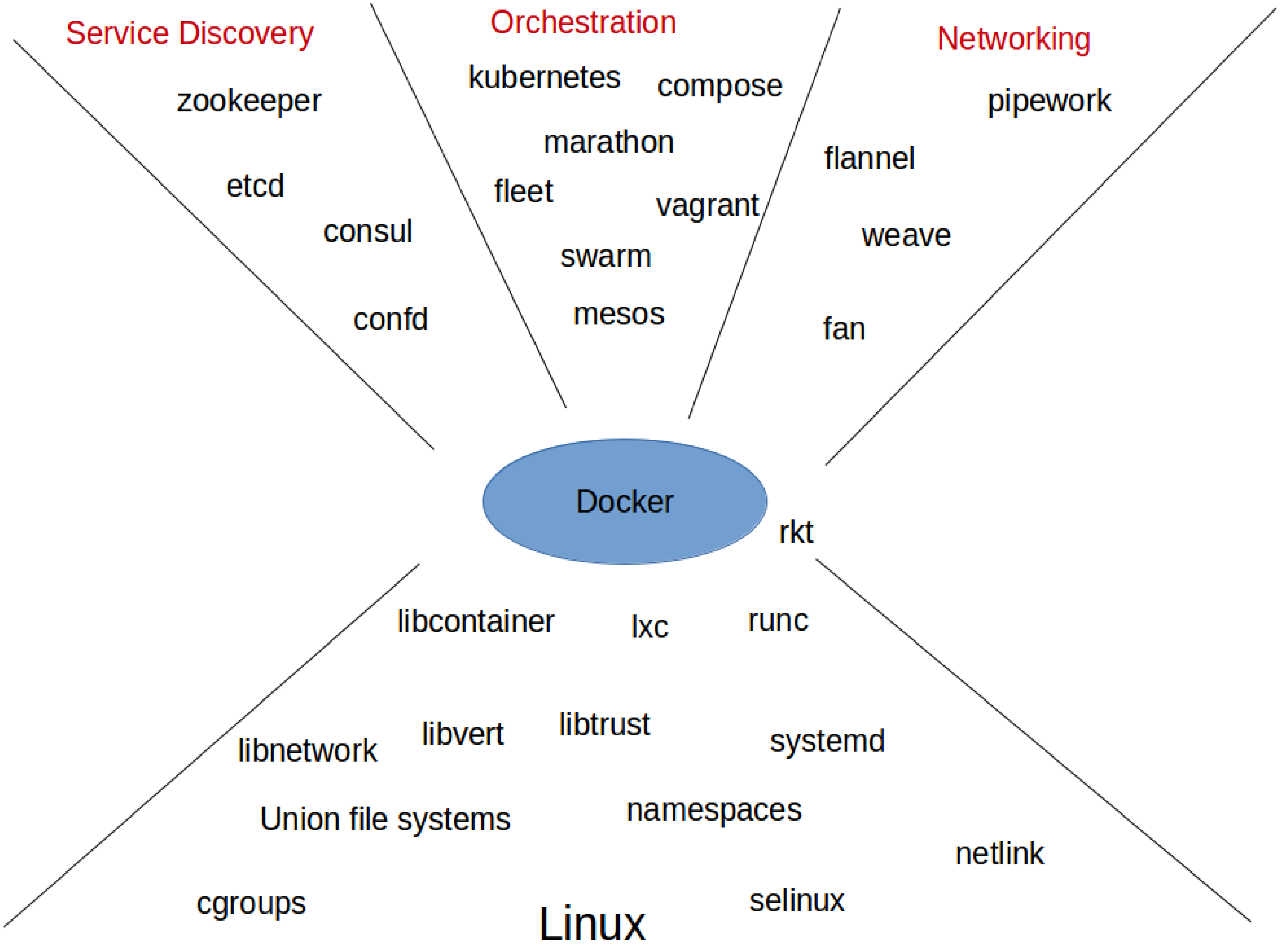
```
$ ansible-playbook -K remove-swarm-hosts.yaml
```

Wrap up

Docker ecosystem

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

Docker ecosystem



Competing technologies

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

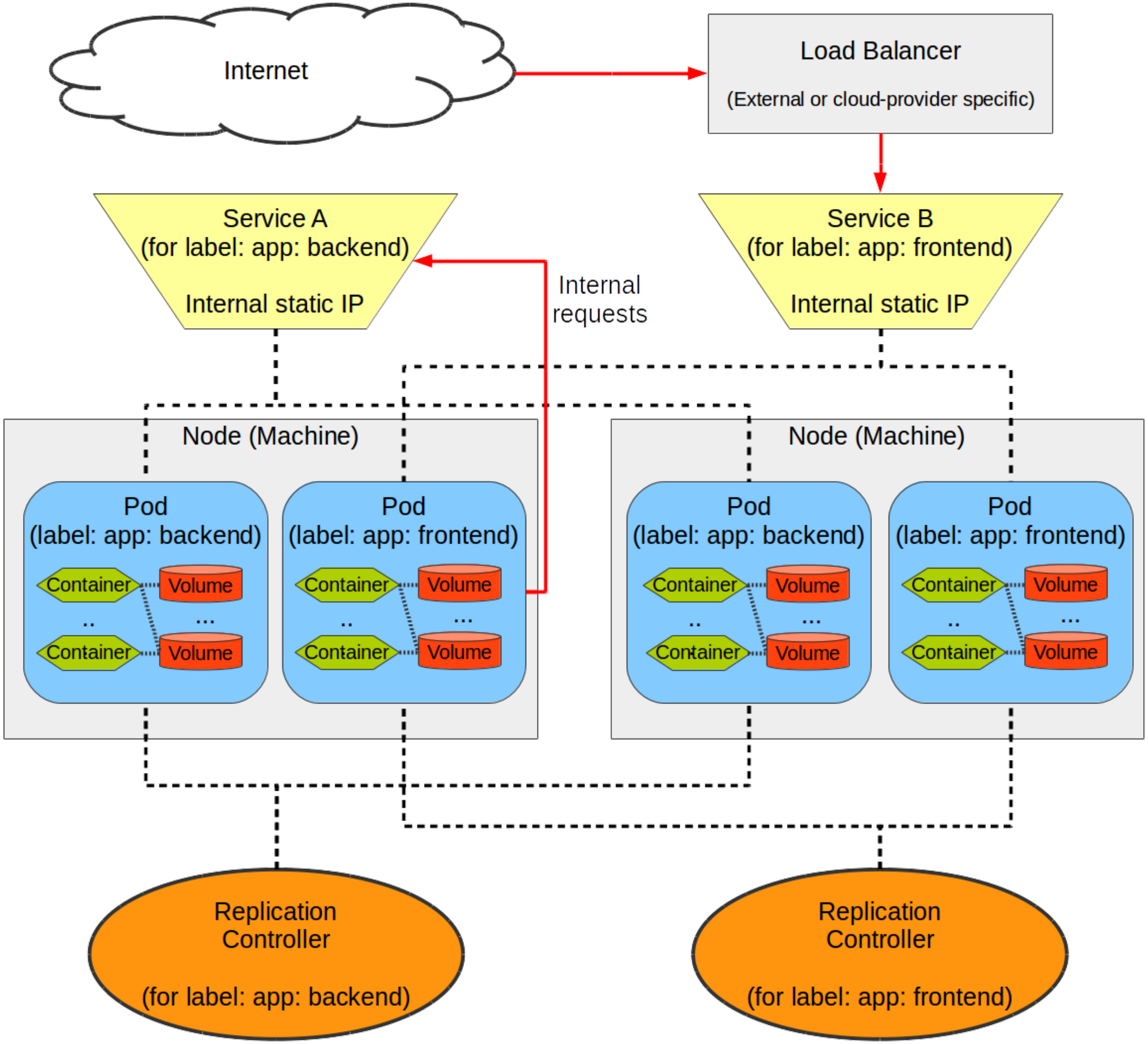
Kubernetes

- 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 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)

Kubernetes Overview


Internet
Load Balancer
(External or cloud-provider specific)
Service A
(for label: app: backend)
Internal static IP
Internal
requests
Service B
(for label: app: frontend)
Internal static IP
Node (Machine)
Pod
(label: app: backend)
Container
Volume
..
...
Container
Volume
Pod
(label: app: frontend)
Container
Volume
..
...
Container
Volume
Node (Machine)
Pod
(label: app: backend)
Container
Volume
..
...
Container
Volume
Pod
(label: app: frontend)
Container
Volume
..
...
Container
Volume
Replication
Controller
(for label: app: backend)
Replication
Controller
(for label: app: frontend)

Pods/Services

- Co-locate containers
- Shared volumes
- IP address (important for port space and migration)
- Unit of deployment and migration
- Easy migration = high utilisation
- Scale service by scaling pods

The end