catalyst

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

Setup

Fetch course resources

```
$ git clone \
   https://github.com/catalyst-training/docker-introduction.git
$ cd docker-introduction
```

```
$ cd ~/docker-introduction
$ ls
```

- Slides for Reveal.js presentation
- docker-introduction.pdf
- Ansible setup playbook
- Sample code for some exercises

Ansible

- Some of the features we will be exploring require setup. We'll use ansible for that.
- Python based tool set
- Automate devops tasks
  - server/cluster management
  - installing packages
  - deploying code
  - managing config

Setup Ansible

```
$ git clone https://github.com/catalyst/catalystcloud-ansible.git
```

```
$ cd ~/catalystcloud-ansible
$ ./install-ansible.sh
.
. <stuff happens>
.
$ source $CC_ANSIBLE_DIR/ansible-venv/bin/activate
```

- Installs python virtualenv with latest ansible libraries
- We'll be using this virtualenv for tasks throughout the course.

Setup Docker

- Follow instructions on website for installing
  - Docker Community Edition
  - docker-compose
- If you are using Ubuntu, use the ansible playbook included in course repo

```
$ cd docker-introduction
$ ansible-playbook -K ansible/docker-install.yml \
      -e ansible_python_interpreter=/usr/bin/python
```

- This playbook installs:
  - latest Docker *Community Edition*
  - `docker-compose`
  - Note: you might need to logout and login again

Fetch and run slides

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

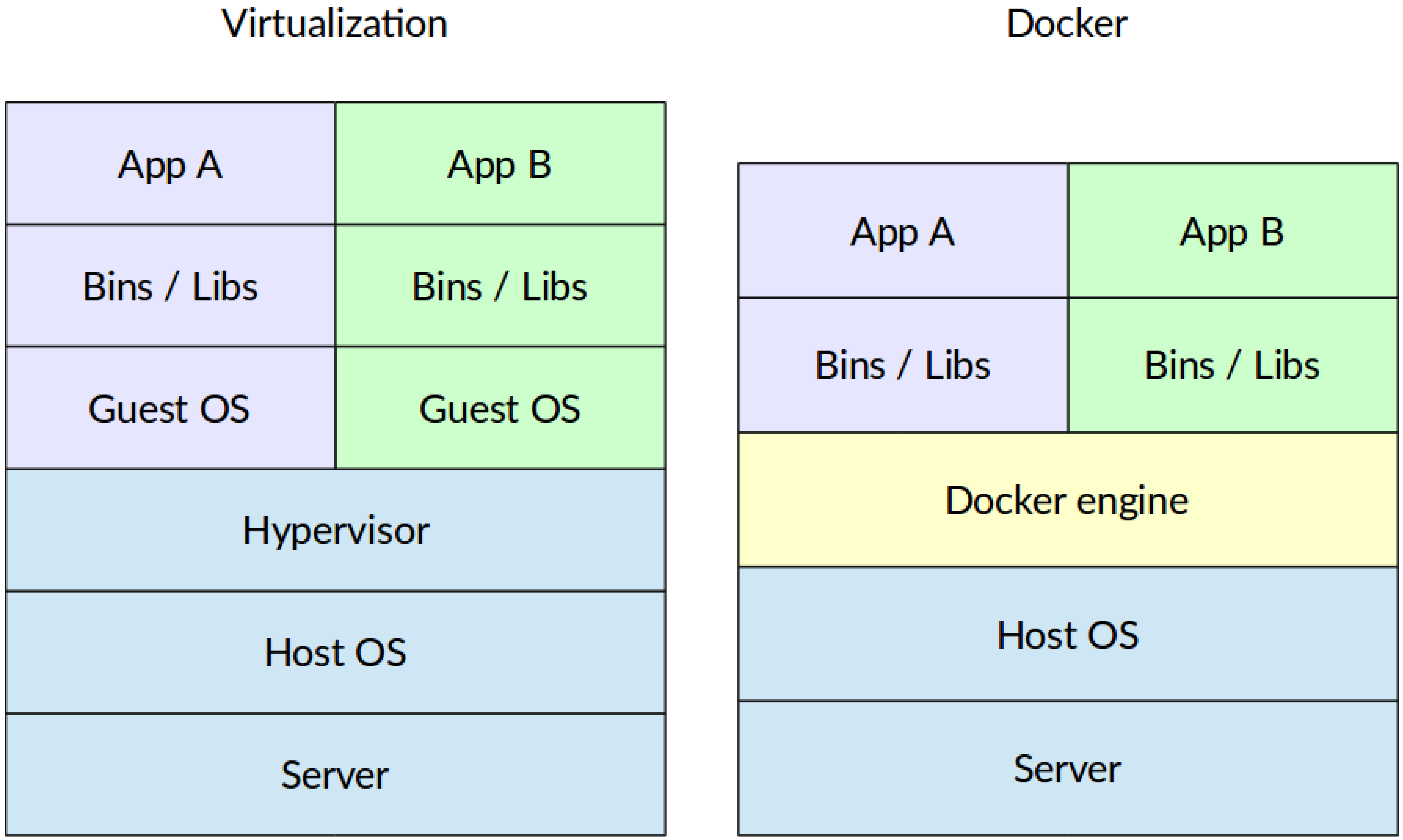
Follow along with course slides: http://localhost:8000

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



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
- 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

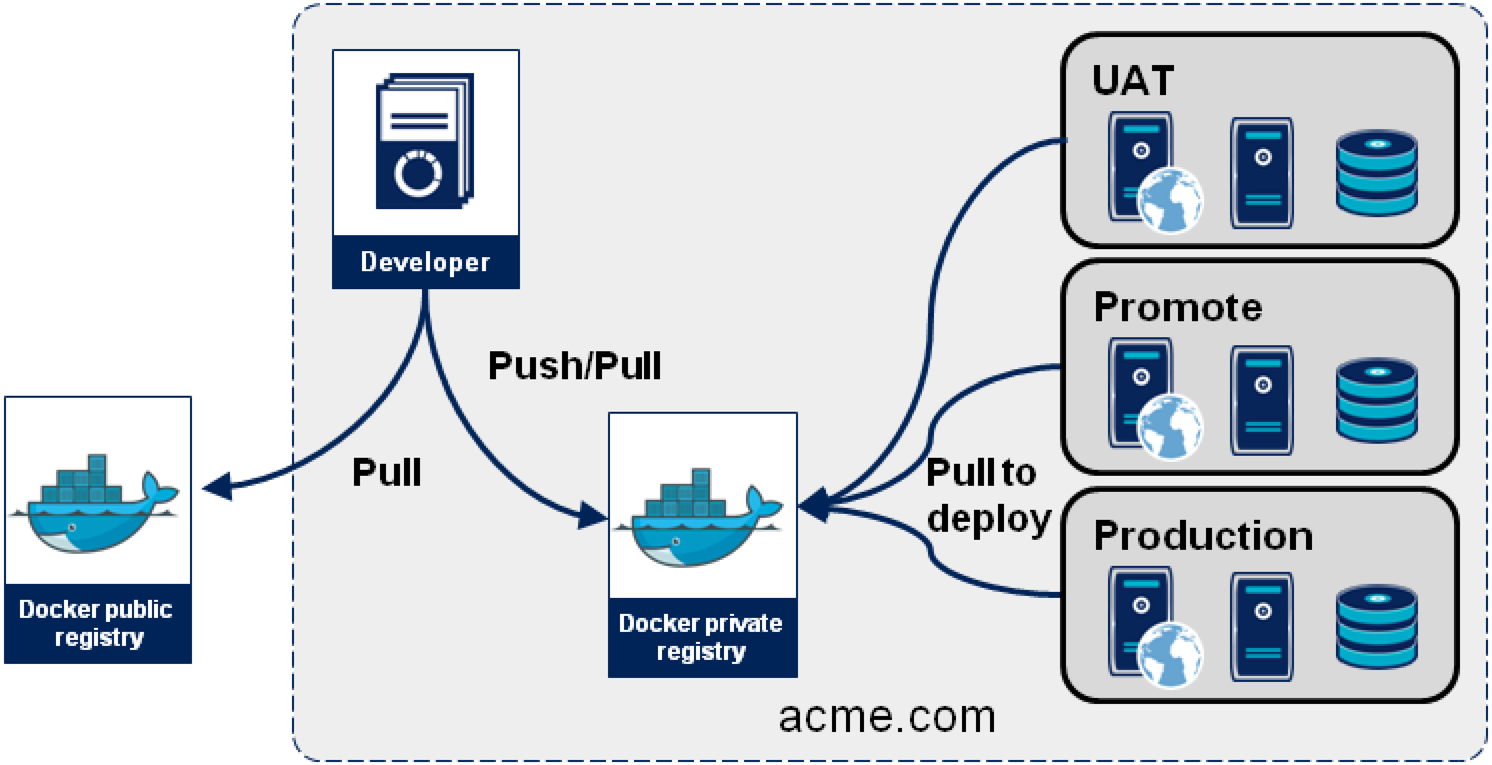
A command-line tool for programmatically defining the contents of a Linux container in code, which can then be versioned, reproduced, shared, and modified easily just as if it were the source code to a program

Docker popularity

- Linux containers are not new
  - FreeBSD Jails
  - LXC containers
  - Solaris Zones
- Docker is doing for containers what Vagrant did for virtual machines
  - Easy to create
  - Easy to distribute

Docker workflow

- Developer packages application and supporting components into image
- Developer/CI pushes image to private or public registry
- The image becomes the unit for distributing and testing your application.



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)

First Steps with Docker

Docker version

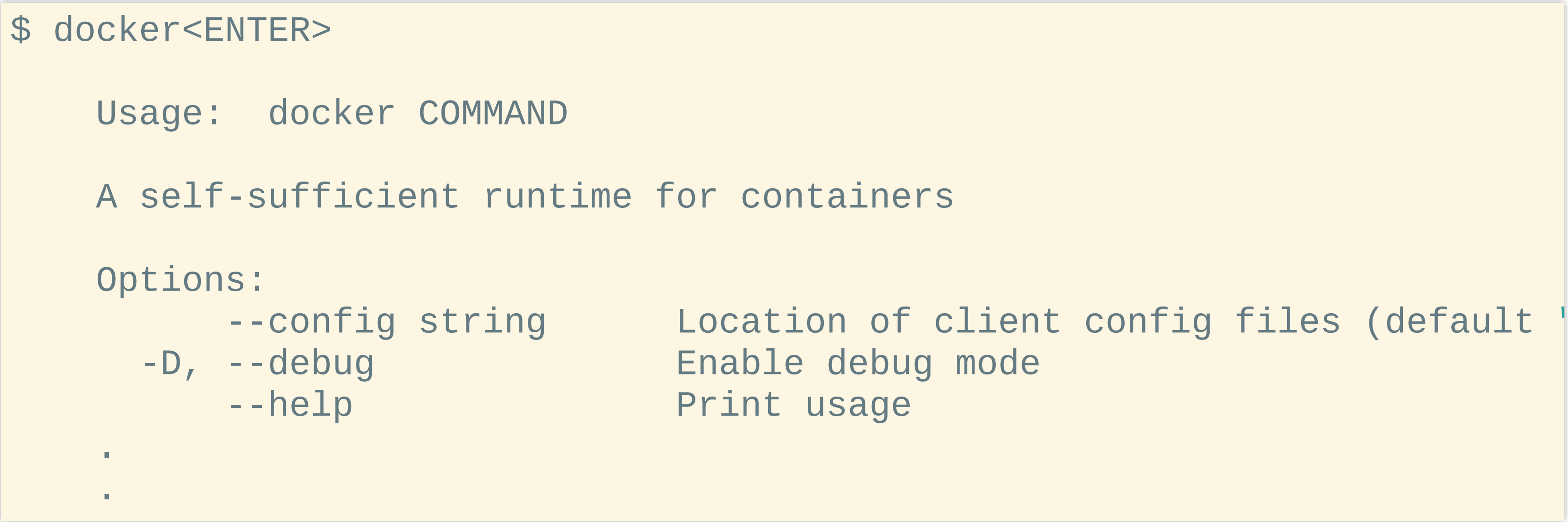
```
$ docker --version
Docker version 17.12.0-ce, build c6d4123
```

Current version scheme similar to Ubuntu versioning:

`YY.MM.#`

Get command documentation

- Just typing `docker` returns list of commands
- Comprehensive online docs on Docker website



```
$ 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
    .
    .
```

Basic client usage

```
docker command [options] [args]
```

- Calling any command with `--help` displays some docs

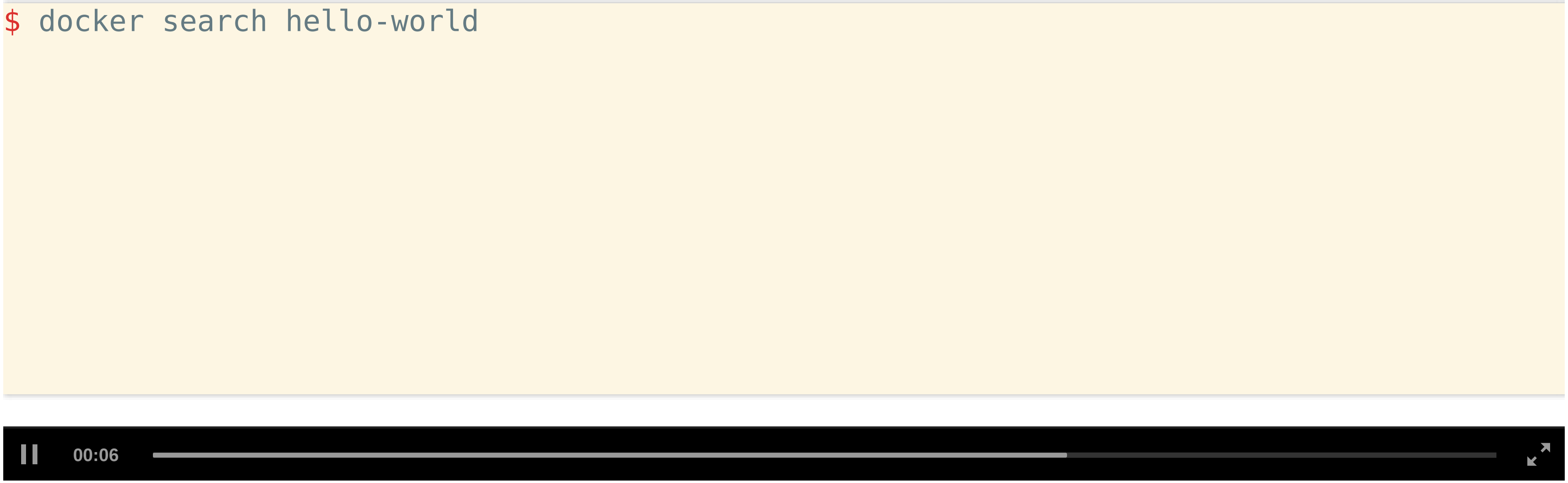
Exercise: View documentation for `docker run`

```
$ docker run --help
Usage:  docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

Options:
      --add-host list                  Add a custom host-to-IP mapping (h
  -a, --attach list                    Attach to STDIN, STDOUT or STDERR
      --blkio-weight uint16            Block IO (relative weight), betwee
      --blkio-weight-device list       Block IO weight (relative device w
      .
      .
```

Search for images

```
docker search [OPTIONS] TERM
```

Option	Argument	Description
-f, --filter	filter	Filter output based on conditions provided
--format	string	Pretty-print search using a Go template
--help		Print usage
--limit	int	Max number of search results (default 25)
--no-trunc		Don't truncate output

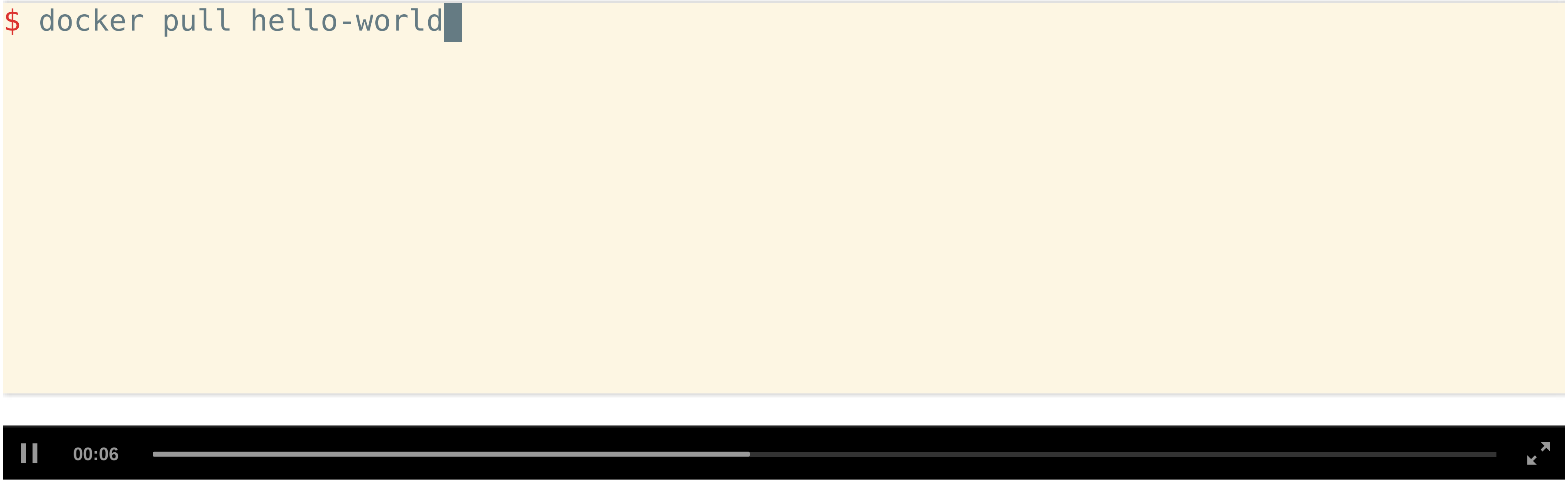


```
$ docker search hello-world
```

Pull an image from a registry

```
docker pull [OPTIONS] NAME[:TAG]
```

Option	Argument	Description
-a, --all-tags		Download all tagged images in the repository
--disable-content-trust		Skip image verification (default true)
--help		Print usage



```
$ docker pull hello-world
```

`docker run`

Run a command in a container from an image

`docker run [options] image [command]`

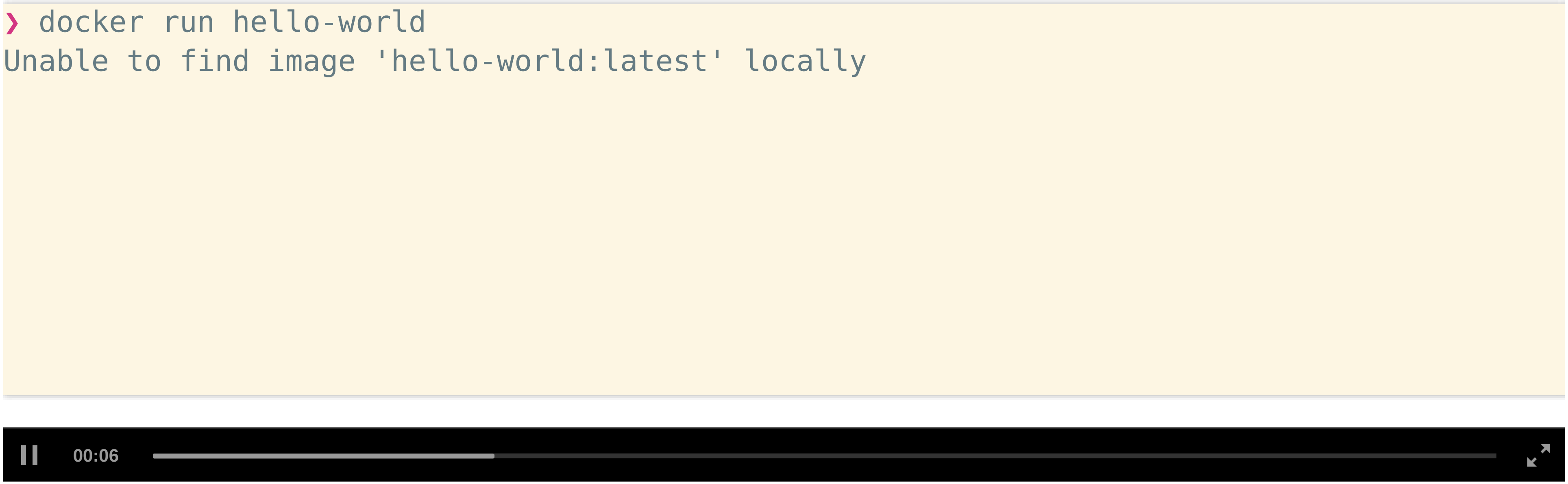
- `docker run` requires an `image` argument

Option	Argument	Description
-i		Keep STDIN open
-t		Allocate a tty
--rm		Automatically remove container on exit
-v	**list**	Mount a volume
-p	**list**	List of port mappings
-e, --env	**list**	Set environment variables
-d, --detach		Run container in background and print container ID
--link	**list**	Add link to another container
--name	**string**	Name for the container

These are just examples that we'll use in the course. See complete list with `docker run --help`

Run a simple container

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



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

- The hello-world image was created by docker for instructional purposes. It just outputs a *hello world*-like message and exits.

Execute command in a container

`docker run image [command]`

```
$ docker run alpine ls
bin
dev
.
.
usr
var
$
```

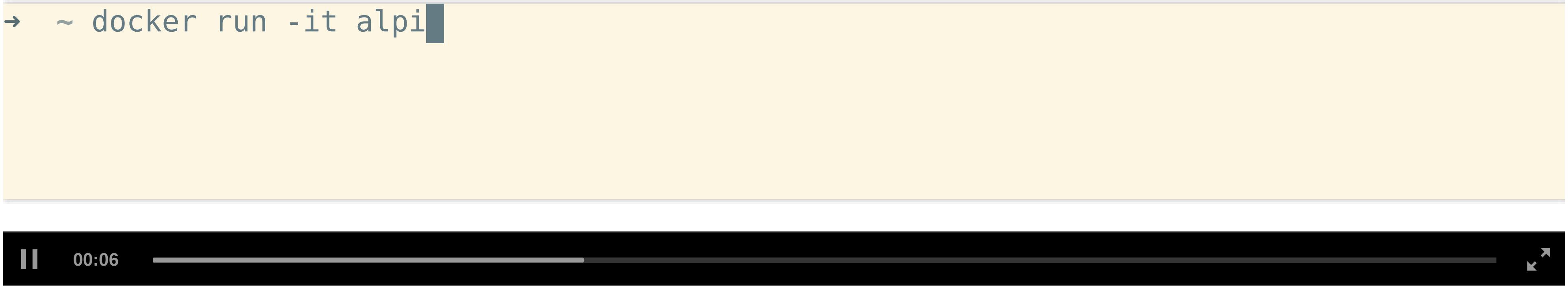
- Docker starts container using alpine image
- The *alpine* image contains the Alpine OS, a very minimal Linux distribution.
- `[command]` argument is executed inside container
- Exits immediately
- A docker container only runs as long as it has a process (eg. a shell terminal or program) to run

Exercise: Start an *interactive* shell

`docker run [options] alpine /bin/sh`

Find `[options]` to make container run interactively

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



```
→  ~ docker run -it alpi
```

- Docker starts alpine image
  - `-i` interactively
  - `-t` allocate a pseudo-TTY
- Runs shell command
- Execute commands inside container
- Exiting the shell stops the process and the container

container

List running containers

`docker ps`

```
$ docker ps

CONTAINER ID  IMAGE                               ... NAMES
b3169acf49f8  alpine                              ... adoring_edison
02aa3e50580c  heytrav/docker-introduction-slides  ... docker-intro
```

Note: by default docker will assign a random name to each container (i.e. *adoring_edison*).

Option	Argument	Description
`-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
`--no-trunc`		Don't truncate output

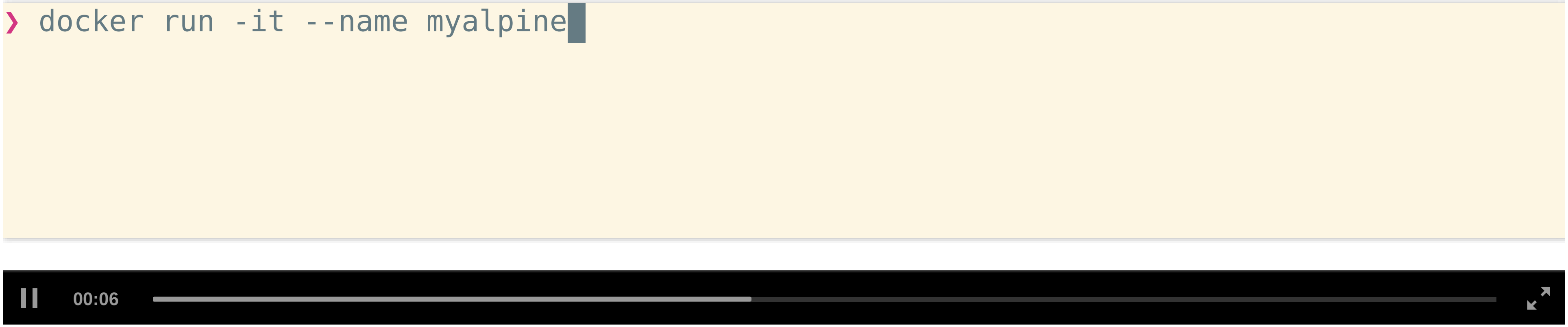
Exercise: Assign the name *myalpine* when running previous example container

Hint: `docker run -it <option> alpine`

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

```
❯ docker ps
CONTAINER ID        IMAGE                                 ...    NAMES
db1faf244e7a        alpine                                ...    myalpine
02aa3e50580c        heytrav/docker-introduction-slides    ...    docker-intro
```

- Exit the shell
- Repeat using same name. What happens?



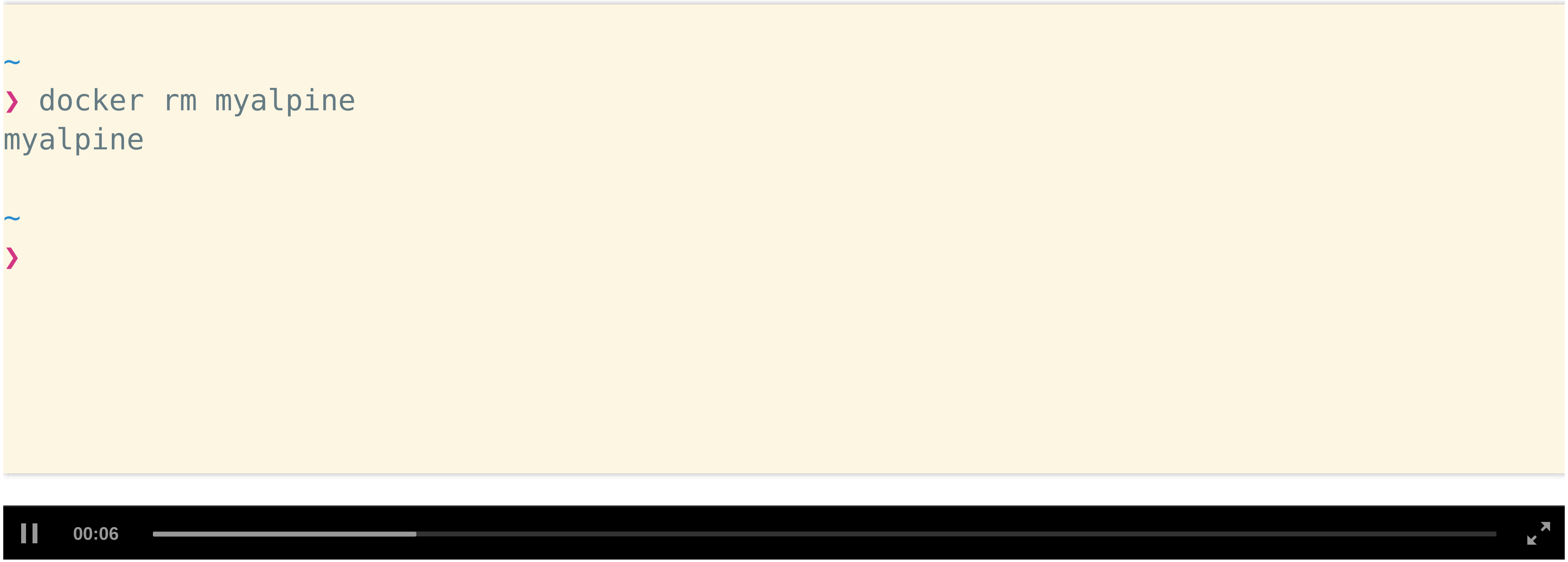
```
❯ docker run -it --name myalpine
```

Removing containers

```
docker rm name|containerID
```

Exercise: remove old `myalpine` container

```
$ docker rm myalpine
```



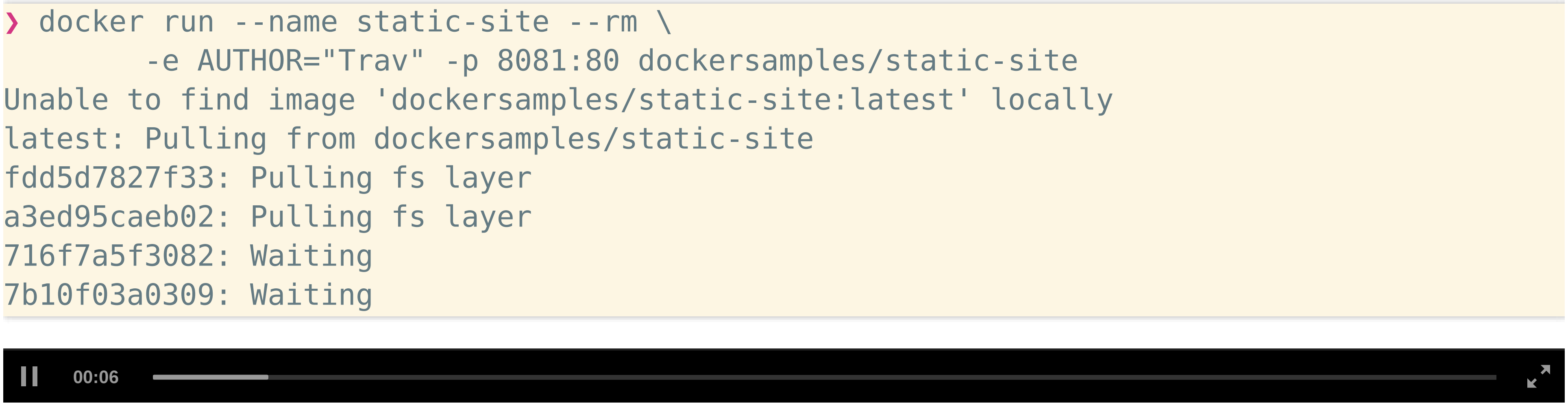
```
~
❯ docker rm myalpine
myalpine

~
❯
```

If you pass the `--rm` flag to `docker run`, containers will be cleaned up when stopped.

Exercise: Run website in a container

```
$ docker run [OPTIONS] dockersamples/static-site
```

- Find values for [OPTIONS]:
  - Give it the name: *static-site*
  - Pass `AUTHOR="YOURNAME"` as environment variable
  - Map port 8081 to 80 internally (hint `8081:80`)
  - Cleans up container on exit



```
❯ docker run --name static-site --rm \
      -e AUTHOR="Trav" -p 8081:80 dockersamples/static-site
Unable to find image 'dockersamples/static-site:latest' locally
latest: Pulling from dockersamples/static-site
fdd5d7827f33: Pulling fs layer
a3ed95caeb02: Pulling fs layer
716f7a5f3082: Waiting
7b10f03a0309: Waiting
```

- Note: `docker run` implicitly pulls image if not available
- Try to exit using CTRL-C. What happens?

Stop a running container

```
docker stop name|containerID
```

Exercise: Stop the `static-site` container

- You actually have a couple options:
  - use the name you gave to the container

    ```
    $ docker stop static-site
    ```

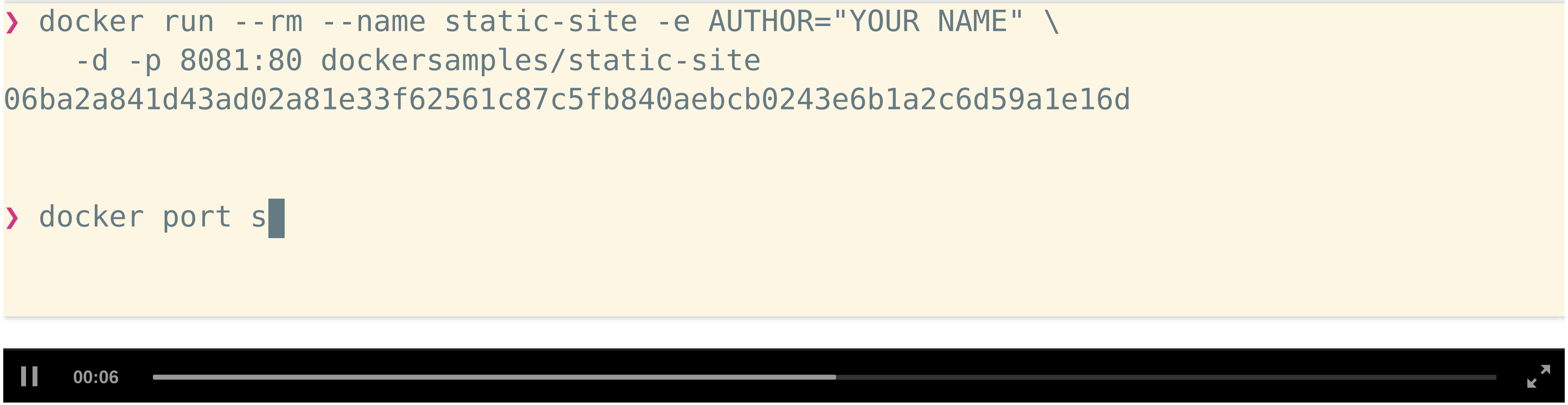
  - use the `CONTAINERID` from `docker ps` output (will depend on your environment)

    ```
    $ docker stop 25eff330a4e4
    ```

Exercise: Running a detached container

- Run static-site container like you did before, but add option to run in the background (i.e. *detached* state).

```
$ 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 s
```

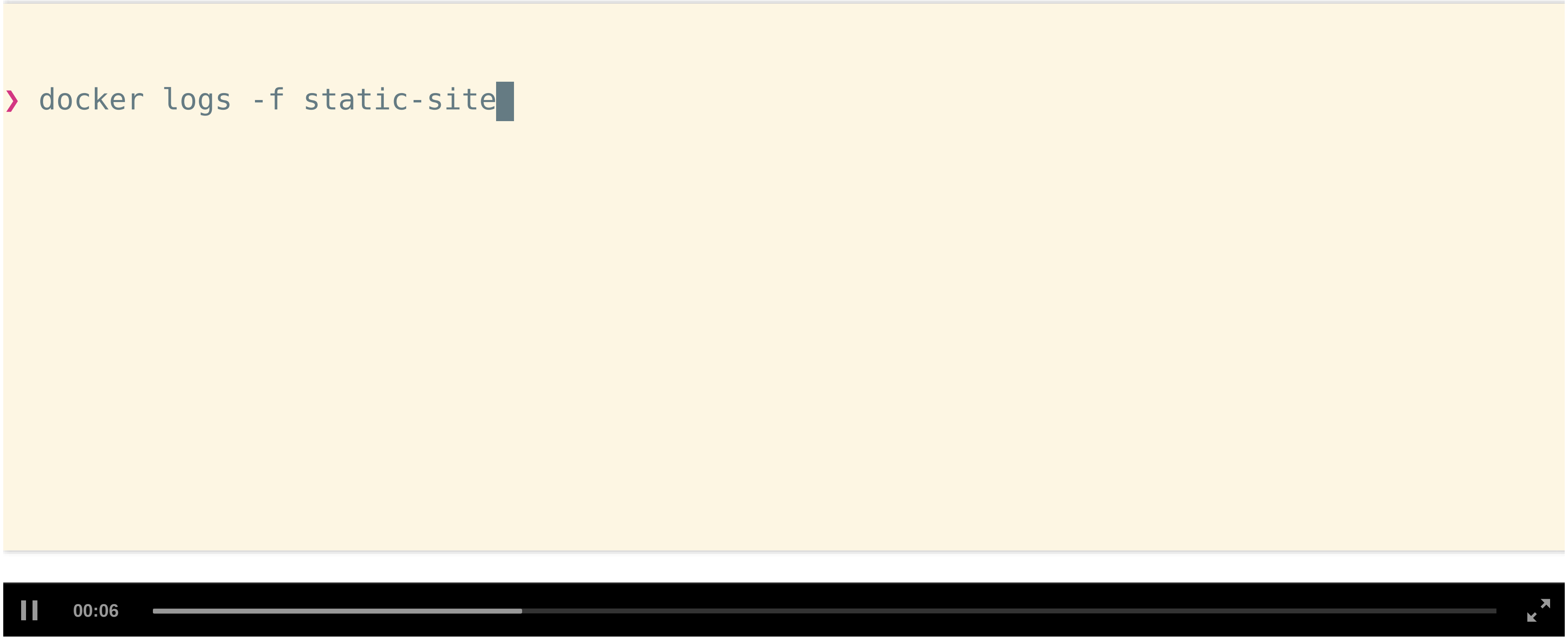
View container logs

```
docker logs [options] CONTAINER
```

Option	Argument	Description
`--details`		Show extra details provided to logs
`-f, --follow`		Follow log output
`--help`		Print usage
`--since`	string	Show logs since timestamp (e.g. 2013-01-02T13:23:37) or relative (e.g. 42m for 42 minutes)
`--tail`	string	Number of lines to show from the end of the logs (default "all")
`-t, --timestamps`		Show timestamps

See online documentation

Exercise: view container logs for `static-site` container



```
❯ docker logs -f static-site
```

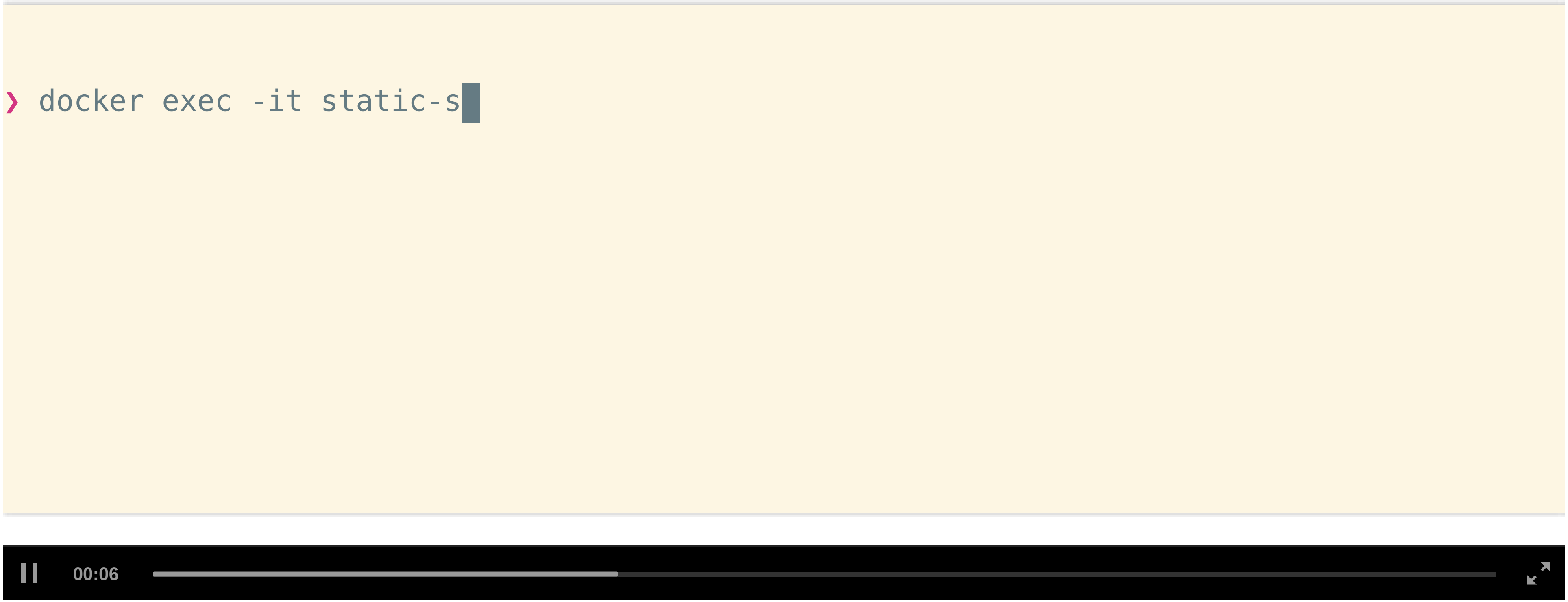
Note: Go to localhost:8081 and refresh a few times

docker exec

```
docker exec [options] CONTAINERID [command]
```

- A way to interact with a running container
- Open a shell inside a running container.
- A bit like ssh'ing into a machine
- Can be useful for debugging
- See online documentation

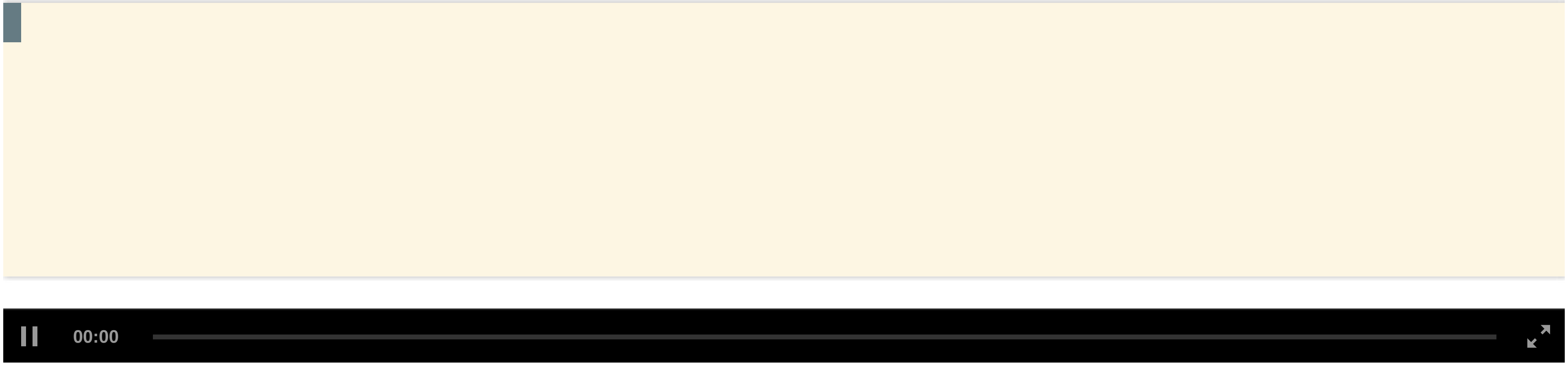
Exercise: Check process list in `static-site` container



```
❯ docker exec -it static-s
```

List local images

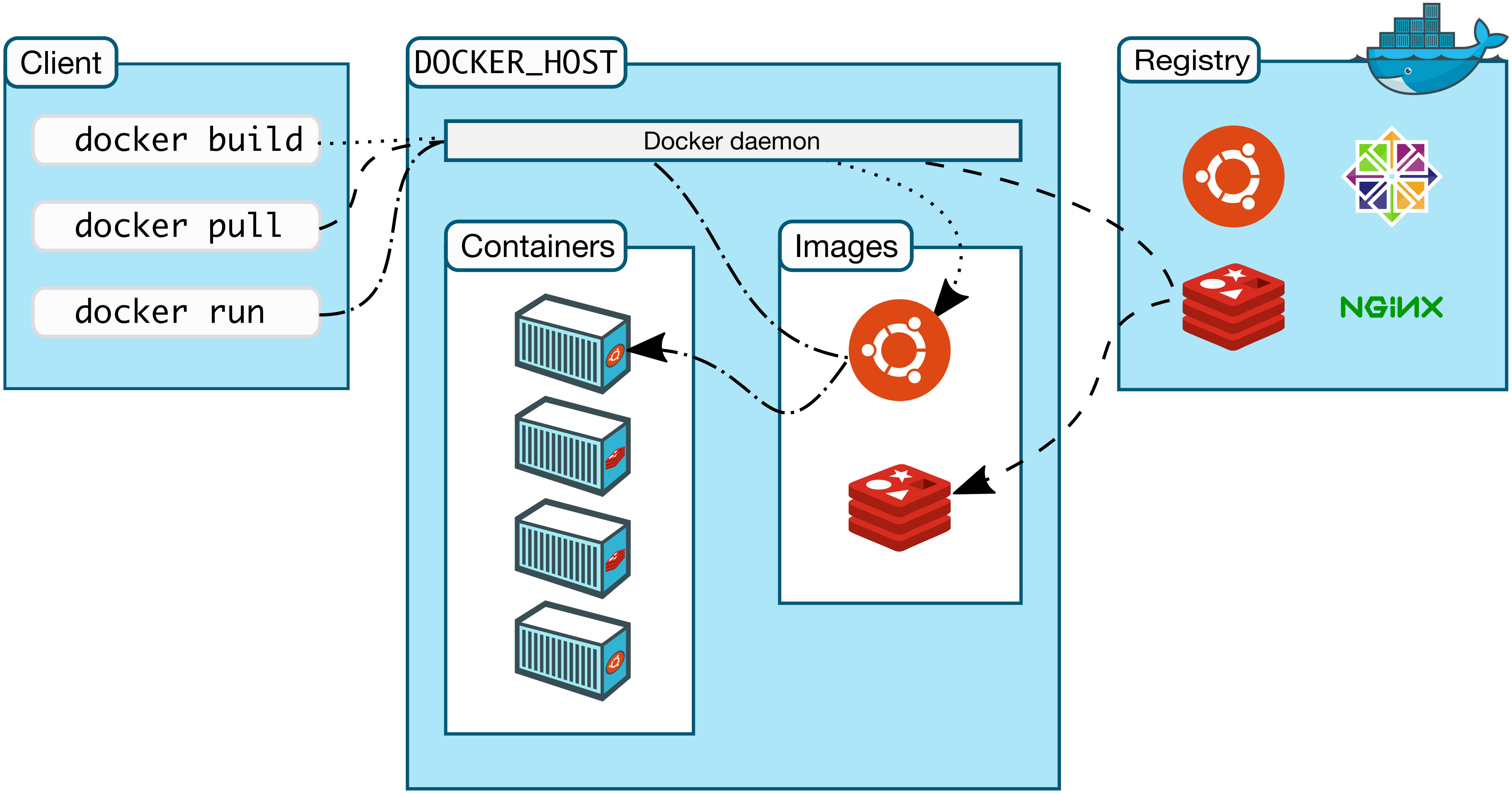
```
$ docker image ls
```



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



How Docker works

Components of Docker

- Images
  - The build component
  - Distributable *artefact*
- Containers
  - The run component
- Registries
  - The distribution component

Components of Docker

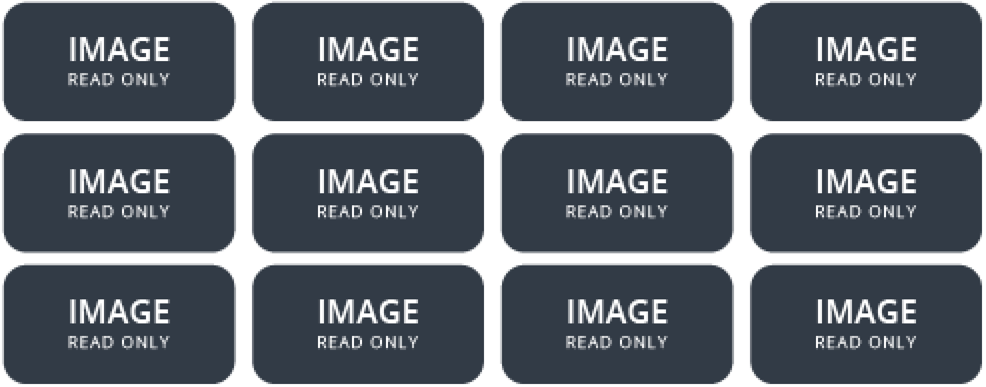
Docker Image

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

IMAGE
READ ONLY

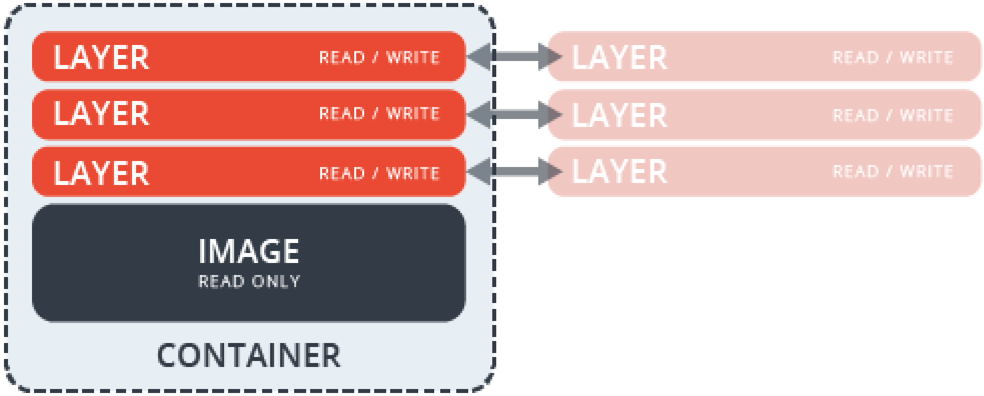
Docker Registry

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



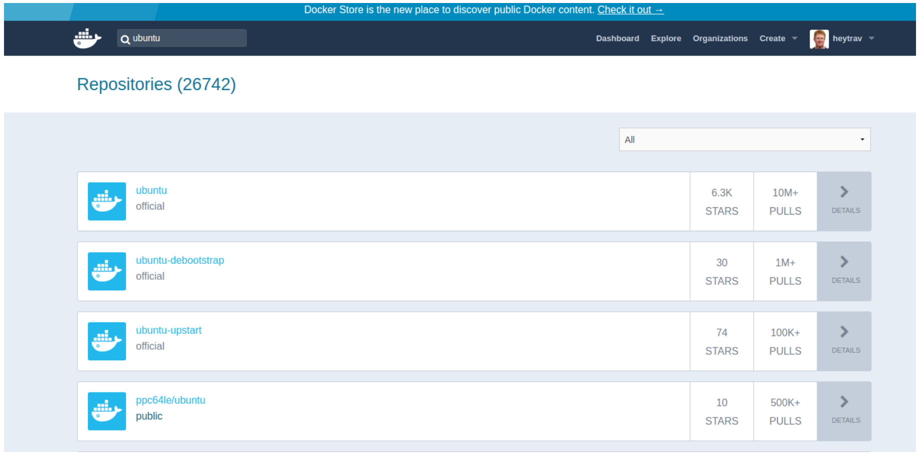
Docker Container

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



Docker Registries

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



Docker Store is the new place to discover public Docker content. Check it out →

Repositories (26742)

ubuntu official	6.3K STARS	10M+ PULLS	DETAILS
ubuntu-debootstrap official	30 STARS	1M+ PULLS	DETAILS
ubuntu-upstart official	74 STARS	100K+ PULLS	DETAILS
ppc64le/ubuntu public	10 STARS	500K+ PULLS	DETAILS

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

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

Created by single authority (OS, packages):

- ubuntu:16.04
- centos:7.3.1611
- postgres

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)

Image naming semantics

- No upper-case letters
- Tag is optional. Implicitly *:latest* if not specified
  - `postgres:9.4`
  - `ubuntu == ubuntu:latest == ubuntu:16.04`
- If pushing to a registry, need url and username
  - If registry not specified, docker.io is default:
    - `docker.io/username/my-image == username/my-image`
  - `my.reg.com/my-image:1.2.3`
  - GitLab registry accept several variants:
    - gitlab.catalyst.net.nz:4567/<group>/<project>:tag
    - gitlab.catalyst.net.nz:4567/<group>/<project>/optional-image-name:tag
    - gitlab.catalyst.net.nz:4567/<group>/<project>/optional-name/optional-image-name:tag

Layering of images

- Images are *layered*
- Any child image built by adding layers on top of base
- Each successive layer is set of differences to preceding layer
- A layer is an instruction that
  - change filesystem
  - tells Docker what to do when run

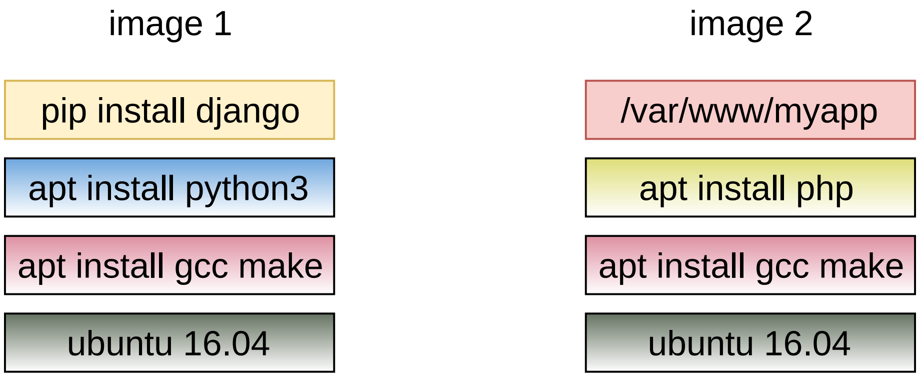
Layer	**Description**
4	execute `myfile.sh`
3	make myfile.sh executable
2	copy myfile.sh to working directory
1	install libs
0	Base Ubuntu OS

Sharing image layers

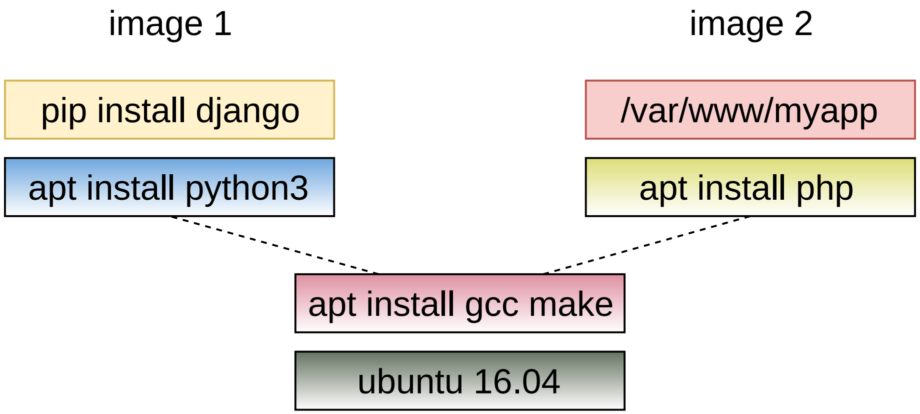
- Images will share any common layers
- Applies to
  - Images pulled from Docker
  - Images you build yourself

Sharing image layers

Two separate images



Reality: common layers shared



View image layers

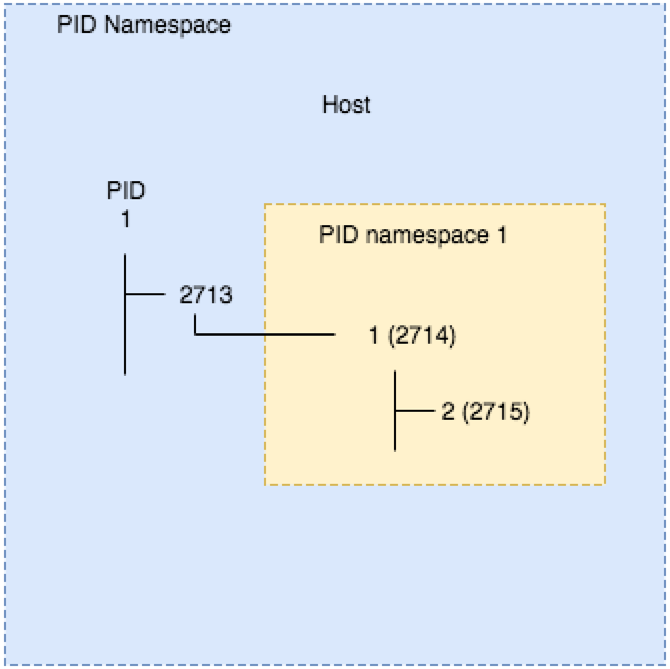
`docker history <image>`

```
$ docker history heytrav/docker-introduction-slides

IMAGE          CREATED        CREATED BY              SIZE                COMMENT
e72084f25e08   2 months ago   /bin/sh -c #(nop)       0B
<missing>      2 months ago   /bin/sh -c #(no         0B
   .
   .
<missing>      9 months ago   /bin/sh -c #(n          0B
<missing>      9 months ago   /bin/sh -c #(n          3.97MB
```

Container basics

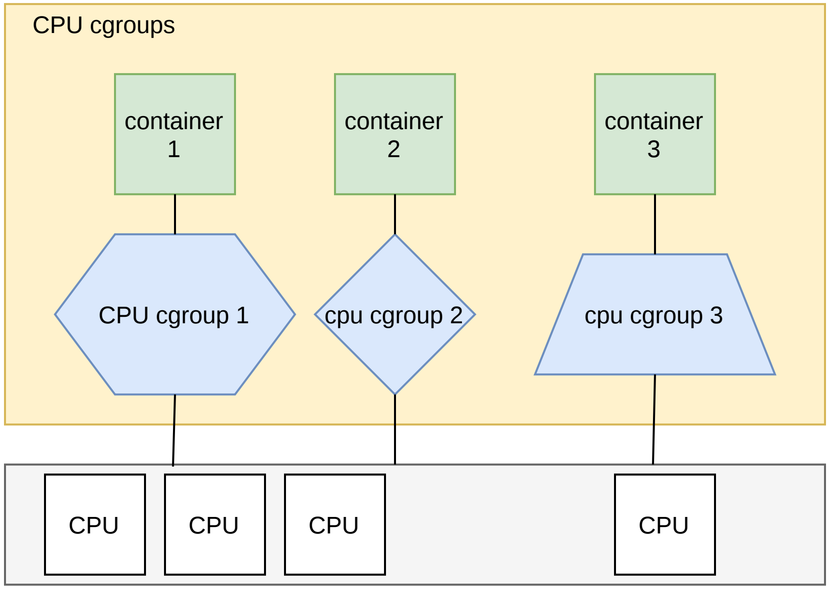
Namespaces

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



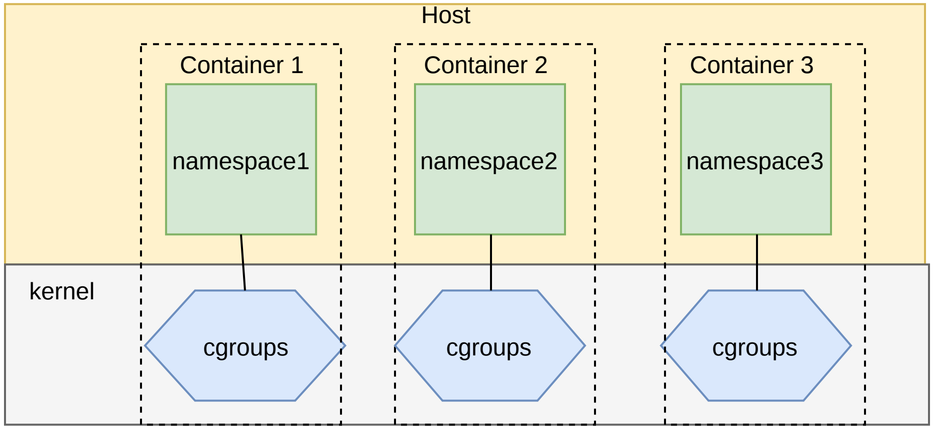
Cgroups

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



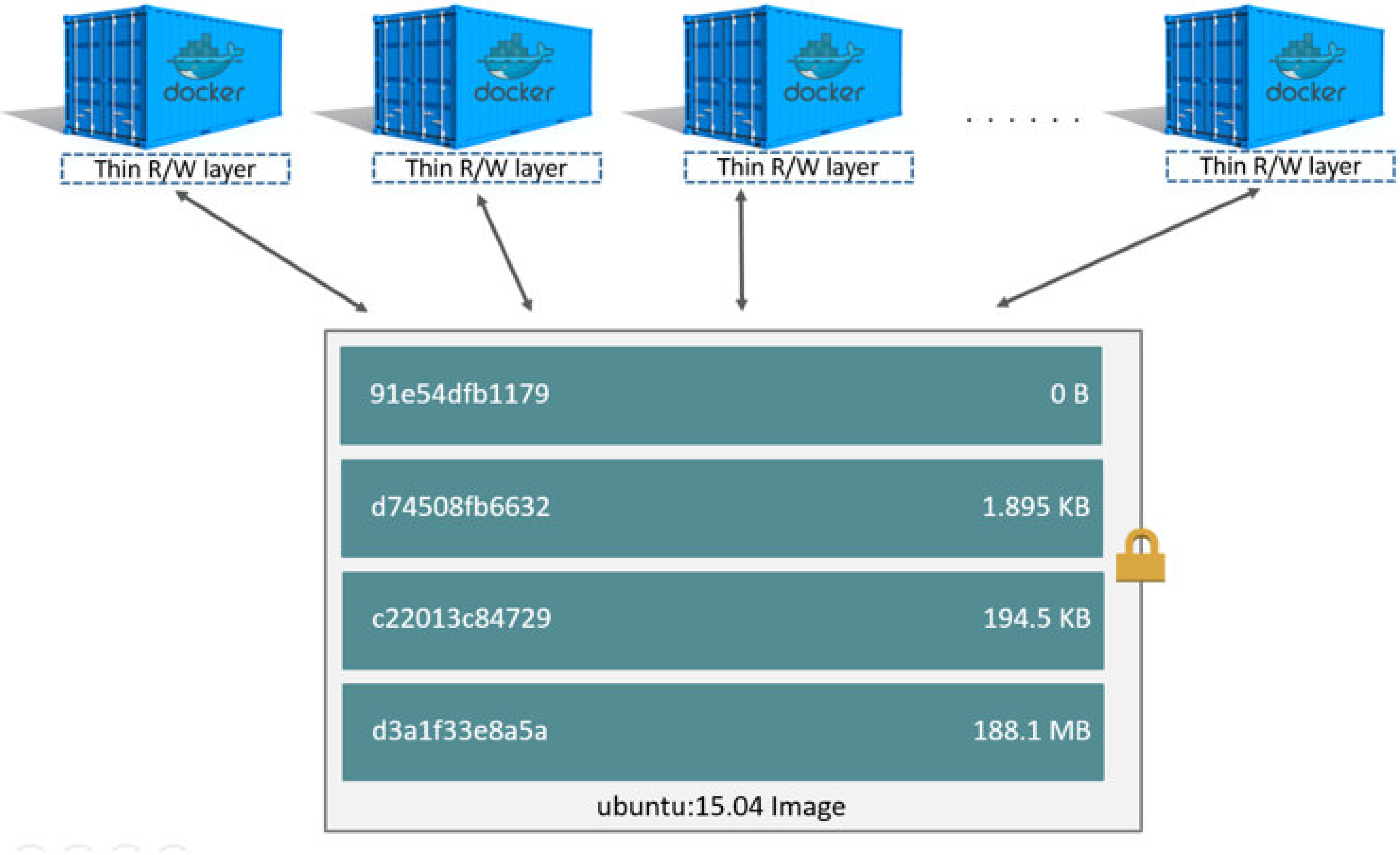
Combining the two

A running container represents a combination of layered file system, namespace and sets of cgroups



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



Create images, explore layers

Docker command	Description	Syntax
`diff`	Inspect changes to files on a container's filesystem	`docker diff [options]` *`CONTAINERID`*
`commit`	Create a new image from a container's changes	`docker commit [options]` *`CONTAINER [IMAGE[:TAG]]`*
`history`	Show history of an image	`docker history [options]` *`image:tag`*

Exercise: Explore image layers

```
$ docker run -it ubuntu:16.04 /bin/bash
root@CONTAINERID:/$ apt-get update
root@CONTAINERID:/$ exit
$ docker ps -a
$ docker diff CONTAINERID
```

```
$ docker commit CONTAINERID ubuntu:update
13132d42da3cc40e8d8b4601a7e2f4dbf198e9d72e37e19ee1986c280ffcb97c
```

```
$ docker image ls
$ docker history ubuntu:16.04
$ docker history ubuntu:update
```

- Created an image by committing changes in a container
- Now have two separate images
- Share common layers; only difference is new layer on ubuntu:update

Creating Docker Images

Introducing the *Dockerfile*

- A text file
- Usually named `Dockerfile`
- Sequential instructions for building a Docker image
- Each instruction creates a layer on the previous

Structure of a Dockerfile

- Start by telling Docker which base image to use

```
FROM <base image>
```

- 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

FROM

```
FROM image:tag
```

Define the base image for a new image

```
FROM ubuntu:17.04
```

```
FROM debian # :latest implicit
```

```
FROM my-custom-image:1.2.3
```

- Image can be
  - An official base image
    - ubuntu:16.04
    - alpine
    - postgres:9.4
  - Another image you have created

RUN

```
RUN command arg1 arg2 ...
```

Execute shell commands for building image

```
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
```

COPY

`COPY src dest`

Copy files from build directory into image

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

```
COPY . /usr/share/www
```

WORKDIR

```
WORKDIR path
```

- Create a directory in the image
- Container will run relative to this directory

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

CMD

- Provide defaults to executable
- or provide executable
- Two ways to execute a command:
  - shell form: `CMD command param1 param2 ...`
  - exec form: `CMD ["command", "param1", "param2"]`

Exercise: Write a basic Dockerfile

```
$ cd ~/docker-introduction/sample-code/first-docker-file
$ ls
hello.sh
```

- Write a `Dockerfile`:
  - Named `Dockerfile`
  - Based on alpine
  - Set working directory to `/app`
  - Copy hello.sh into working directory
  - make hello.sh executable
  - tell docker to run hello.sh on docker run

```
FROM alpine
WORKDIR /app
COPY hello.sh .
RUN chmod +x hello.sh
CMD ["./hello.sh"]
```

Now that we have a Dockerfile, we can make an image

Building Docker images

```
docker build [options] image:[tag] ./path/to/Dockerfile
```

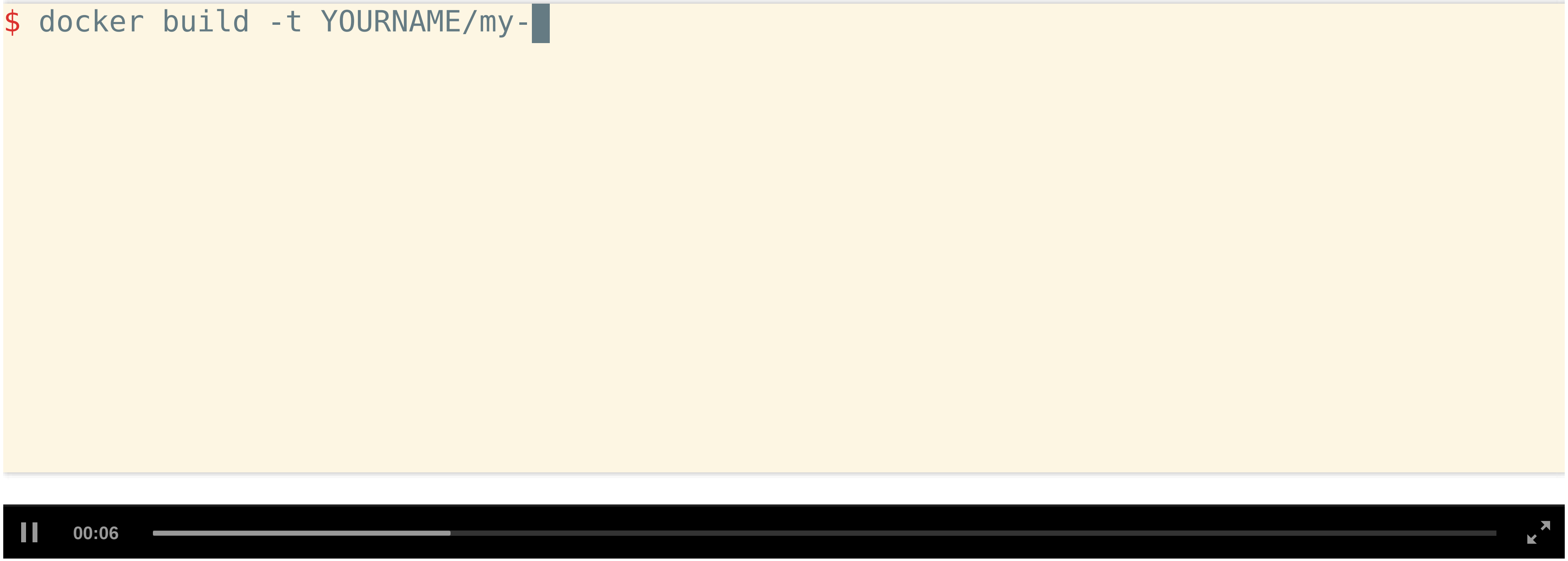
Options	Arguments	Description
`--compress`		Compress the build context using gzip
`-c, --cpu-shares`	int	CPU shares (relative weight)
`--cpuset-cpus`	string	CPUs in which to allow execution (0-3, 0,1)
`--cpuset-mems`	string	MEMs in which to allow execution (0-3, 0,1)
`--disable-content-trust`		Skip image verification (default true)
`-f, --file string`		Name of the Dockerfile (Default is 'PATH/Dockerfile')
`--pull`		Always attempt to pull a newer version of the image
`-t, --tag`	list	Name and optionally a tag in the 'name:tag' format

Note that a path to folder with Dockerfile is always required.
When in same directory, use "."

Exercise: build image using Dockerfile

- Build a Docker image:
  - Use Dockerfile from earlier example
  - Name image YOURNAME/my-first-image

```
$ docker build -t YOURNAME/my-first-image .
```



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

```
$ docker run YOURNAME/my-first-image .
```

A few more Dockerfile directives

ENTRYPOINT

- Docker images need not be executable by default
- ENTRYPOINT configures executable behaviour of container
- *shell* and *exec* forms just like CMD

```
$ cd ~/docker-introduction/sample-code/entrypoint_cmd_examples
$ docker build -t not-executable -f Dockerfile.notexecutable .
$ docker run not-executable # does nothing
```

```
$ docker build -t executable -f Dockerfile.executable .
$ docker run executable
```

Combining ENTRYPOINT & CMD

- Arguments following the image for `docker run image` overrides CMD
- Use exec form of ENTRYPOINT and CMD together to set base command and default arguments
- 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

Exploring ENTRYPOINT & CMD behaviour

-
  ```
  $ cd sample-code/entrypoint_cmd_examples
  ```

- Compare Dockerfiles:
  - Dockerfile.cmd_only
  - Dockerfile.cmd_and_entrypoint
- Build images:

```
$ docker build -t cmd_only -f Dockerfile.cmd_only .
$ docker build -t cmd_and_entrypoint -f Dockerfile.cmd_and_entrypoint .
```

- Run both the images with or without an additional argument to see what happens

More Dockerfile instructions

EXPOSE

ports to expose when running

VOLUME

folders to expose when running

ENV

Set an environment variable

See official reference documentation for more

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 to a 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

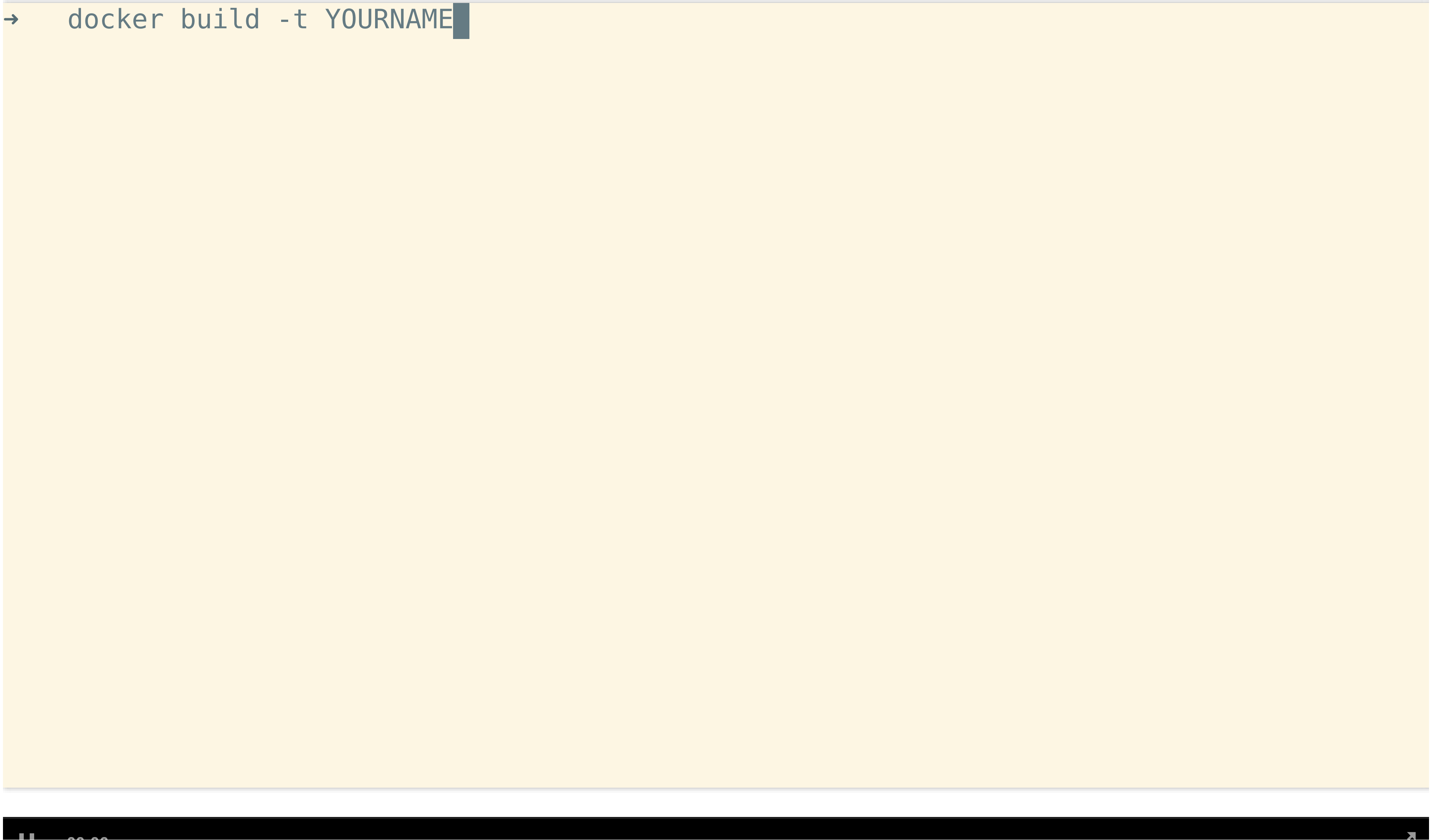
  Instructions for building a Docker image

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 .
```

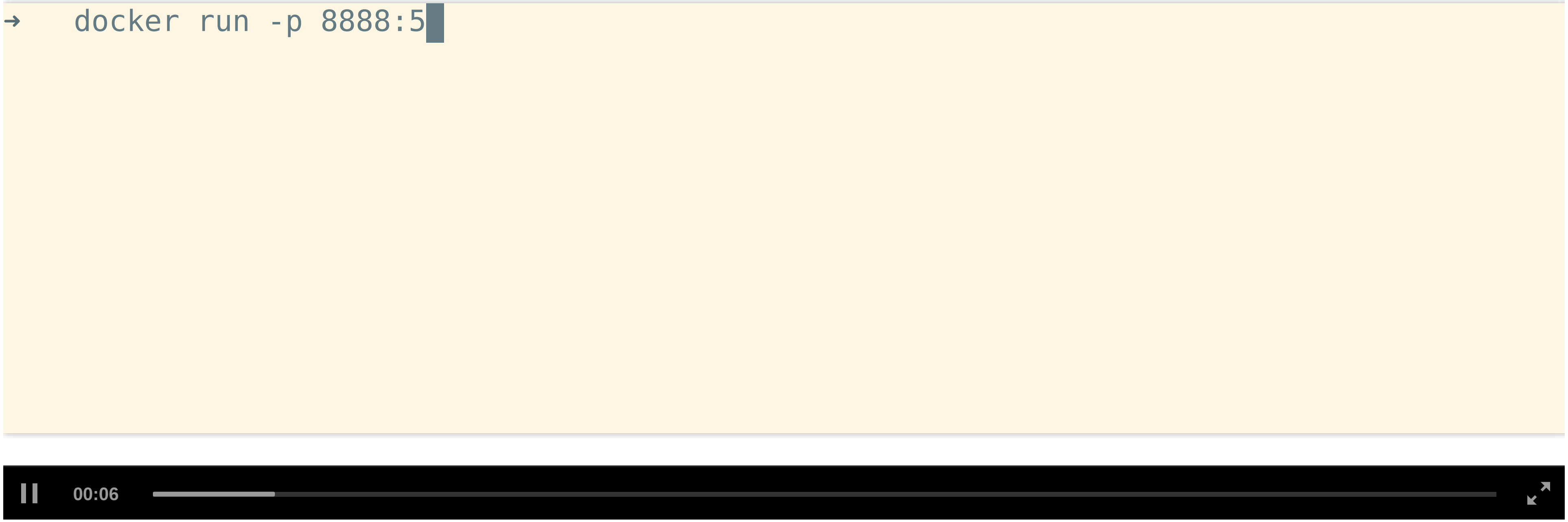


```
➜    docker build -t YOURNAME
```

Note: please replace YOURNAME with your Docker Hub username

Run the container

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

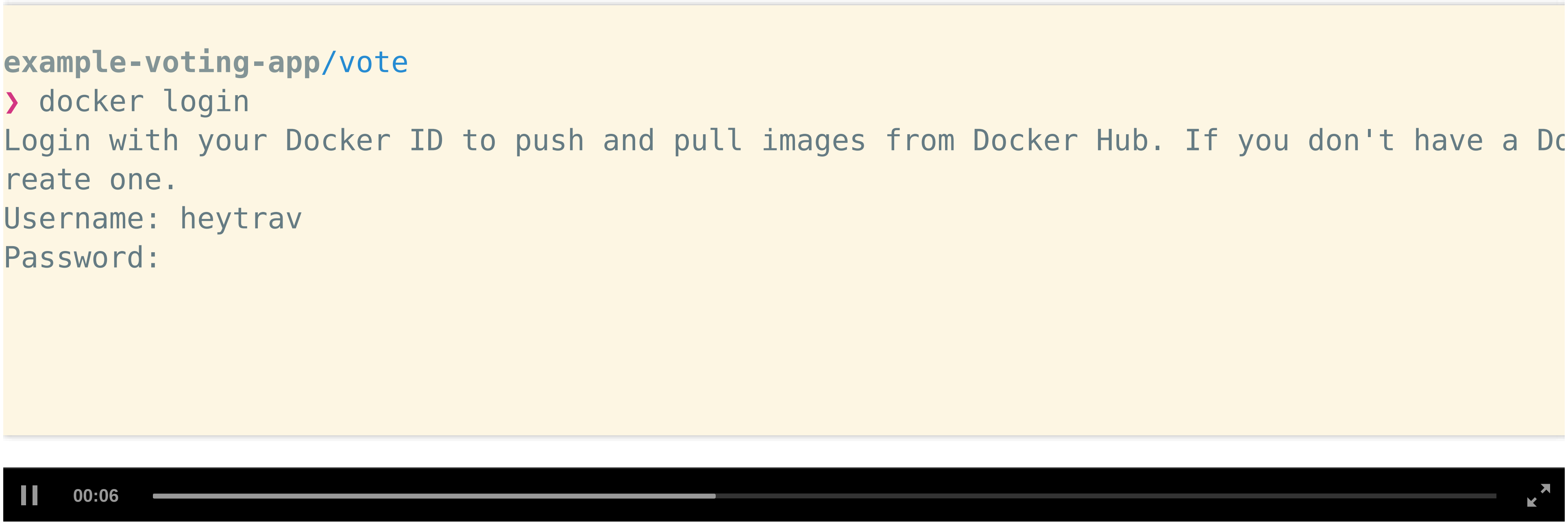


```
➜   docker run -p 8888:5
```

...Now open your test webapp

Login to a registry

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

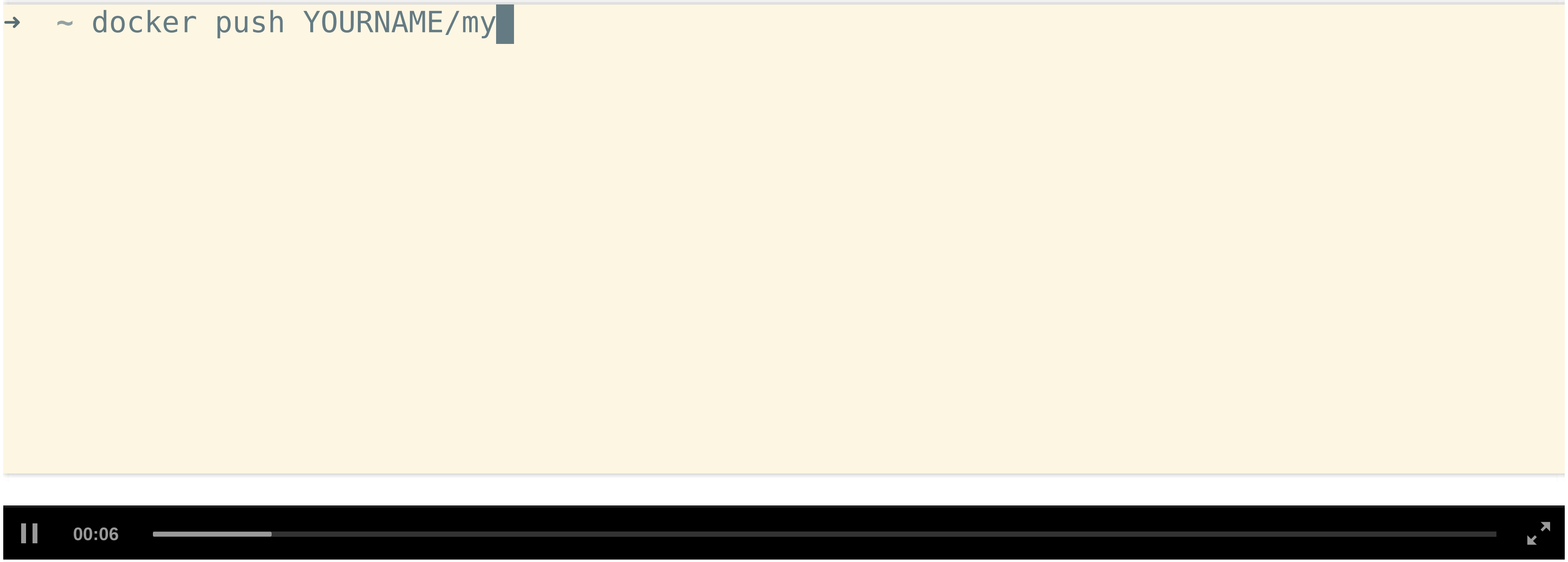


```
example-voting-app/vote
❯ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Do
reate one.
Username: heytrav
Password:
```

- 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 YOURNAME/my
```

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
- Avoid installing unnecessary packages
- Minimise concerns
    - Avoid multiple processes/apps in one container

Use a `.dockerignore` file

```
# .dockerignore
.git*
.dockerignore
Dockerfile
README*
# don't import python virtualenv
.venv
```

- Top level of your project
- Very similar to `.gitignore`
- `COPY . dest/` will not copy files ignored in `.dockerignore`
- Include things you don't want in your image:
  - `.git` directory
  - `node_modules`, `virtualenv` directories

General guidelines

- Use current official repositories in FROM as base image
- Image size may be a factor on cloud hosts where space is limited
  - debian 124 MB
  - ubuntu 117 MB
  - alpine 3.99 MB
  - busybox 1.11 MB
- Choice of image depends on other factors

Layer caching

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

- Build image in `sample-code/caching` directory
- Run build a second time. What happens?
- Change line with Change me! and run again
- Each instruction creates a layer in an image
- Docker caches layers when building
- When a layer is changed Docker rebuilds from changed layer

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
```

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

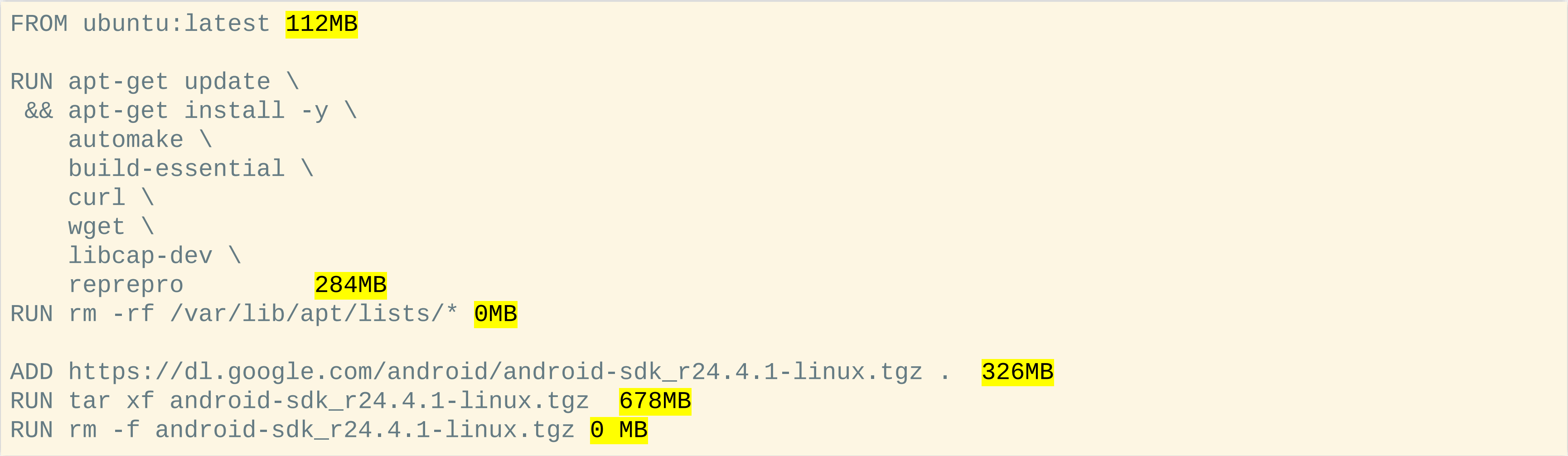
- Uncomment nginx line and run `docker build` again
- Only rebuilds from layer that was *changed*
- Example 1: `apt-get update` does not refresh index
    - apt repos might change
- Best to combine apt-get update and install packages to force apt to refresh index (Example 2)

Optimising image size

Intermediate layers

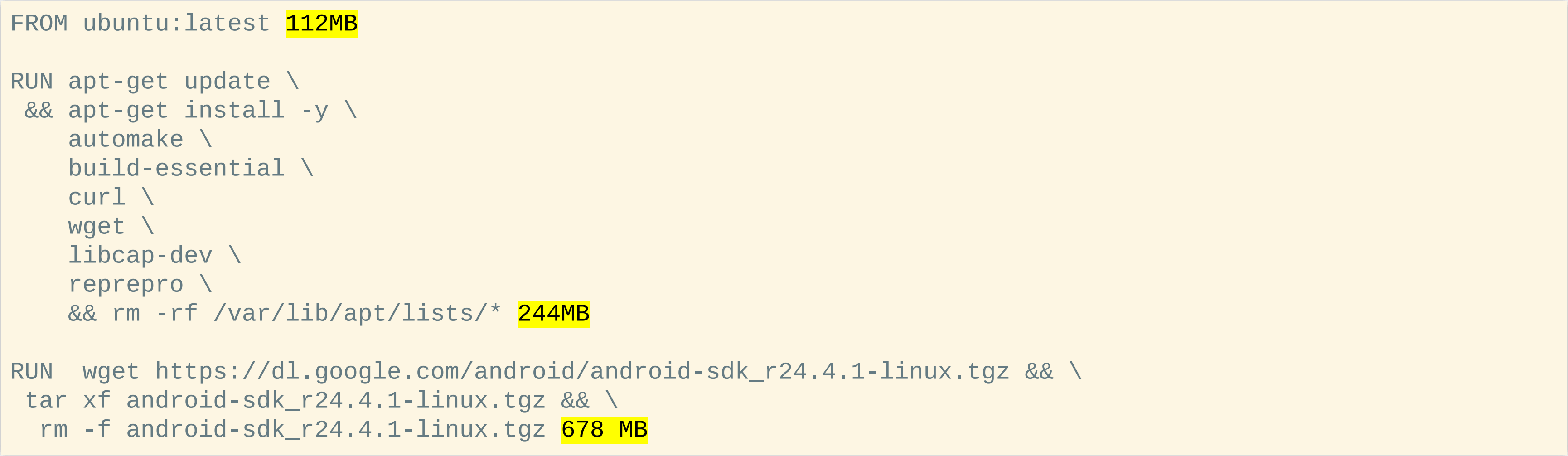
- Image size is sum of intermediate layers
- Even if you remove something it exists as a diff on previous layer
- Run clean up in same layer whenever possible

Example: Optimising image size



```
FROM ubuntu:latest 112MB

RUN apt-get update \
 && apt-get install -y \
    automake \
    build-essential \
    curl \
    wget \
    libcap-dev \
    reprepro         284MB
RUN rm -rf /var/lib/apt/lists/* 0MB

ADD https://dl.google.com/android/android-sdk_r24.4.1-linux.tgz .  326MB
RUN tar xf android-sdk_r24.4.1-linux.tgz  678MB
RUN rm -f android-sdk_r24.4.1-linux.tgz 0 MB
```

Image size: 1.4 GB



```
FROM ubuntu:latest 112MB

RUN apt-get update \
 && apt-get install -y \
    automake \
    build-essential \
    curl \
    wget \
    libcap-dev \
    reprepro \
    && rm -rf /var/lib/apt/lists/* 244MB

RUN  wget https://dl.google.com/android/android-sdk_r24.4.1-linux.tgz && \
 tar xf android-sdk_r24.4.1-linux.tgz && \
  rm -f android-sdk_r24.4.1-linux.tgz 678 MB
```

Image size: 1 GB

ADD

- Copies files to a directory

```
ADD . /usr/path/
```

- Downloads file from web

```
ADD http://domain.com/file.txt /usr/path/
```

- Unpack archives into directory

```
ADD file.tar /usr/path/
```

- However, does not unpack remote archives. This will just put `file.tar` in `/usr/path/`

```
ADD http://domain.com/file.tar /usr/path/
```

Problems with ADD

- Large intermediate layers

```
ADD http://domain.com/big.tar.gz /usr/path/ # large intermediate layer
RUN cd /usr/path && tar -xvf big.tar.gz \
   && rm big.tar.gz
```

  - Increased overall image size
- Better solution:

```
RUN curl -SL http://domain.com/big.tar.gz  \
   | tar -xJC /usr/path
```

  - Smaller image size
- COPY only copies files

```
COPY . /usr/path/
```

- Recommend to only use COPY and never ADD

Multistage builds

Optimise image buiilds

- Best practices intended to optimise image size by keeping them small
- Come at the expense of readability
    - Layers with long complicated commands
- Multistage builds
    - Introduced with Docker 17.05
    - Enable optimised image size
    - maintain readability

Multistage builds

How they work

- Multiple FROM directives in a Dockerfile
- Each FROM represents a new build
- Selectively copy artifacts from one of the previous builds
- Leave behind what is not needed

```
FROM ubuntu:16.04 as builder
WORKDIR /bin
COPY . /bin/
RUN make install


FROM alpine
COPY --from=builder /bin/myprogram /
ENTRYPOINT ['/root/myprogram']
```

Multistage builds example

```
$ cd ~/href-counter
$ docker build -t href-counter -f Dockerfile.build .
$ docker image ls | grep href
REPOSITORY    TAG     IMAGE ID          SIZE
href-counter  latest  b0eb64a75c55      687MB
```

```
$ docker build -t href-counter-multi -f Dockerfile.multi .
$ docker image ls | grep href
REPOSITORY          TAG           SIZE
href-counter-multi  latest        10.3MB
```

CMD & ENTRYPOINT

General best practices

- Avoid using *shell* form
  - `ENTRYPOINT "executable param1 param2 ..."`
- Docker directs POSIX commands at process with PID 1
- Using *shell* form, process is run internally using `/bin/sh -c` and do not have PID 1
- It can be difficult to stop container since process does not receive SIGTERM from `docker stop container`

```
$ cd ~/docker-introduction/sample-code/entrypoint_cmd_examples
$ docker build -t runtop-shell -f Dockerfile.top_shell .
$ docker run --rm --name topshell runtop-shell
```

What happens when you want to stop container *topshell*?

CMD & ENTRYPOINT

General best practices

- Best practice to use **exec** form:
  - `CMD ["executable", "param1", "param2", ..]`
- Or in form that creates interactive shell like
  - `ENTRYPOINT ["python"]`
  - `CMD ["/bin/bash"]`

```
$ docker build -t runtop-exec -f Dockerfile.top_exec .
$ docker run runtop-exec
```

- Sometimes app constraints don't allow single process on PID1
- For this purpose recommended to use dumb-init

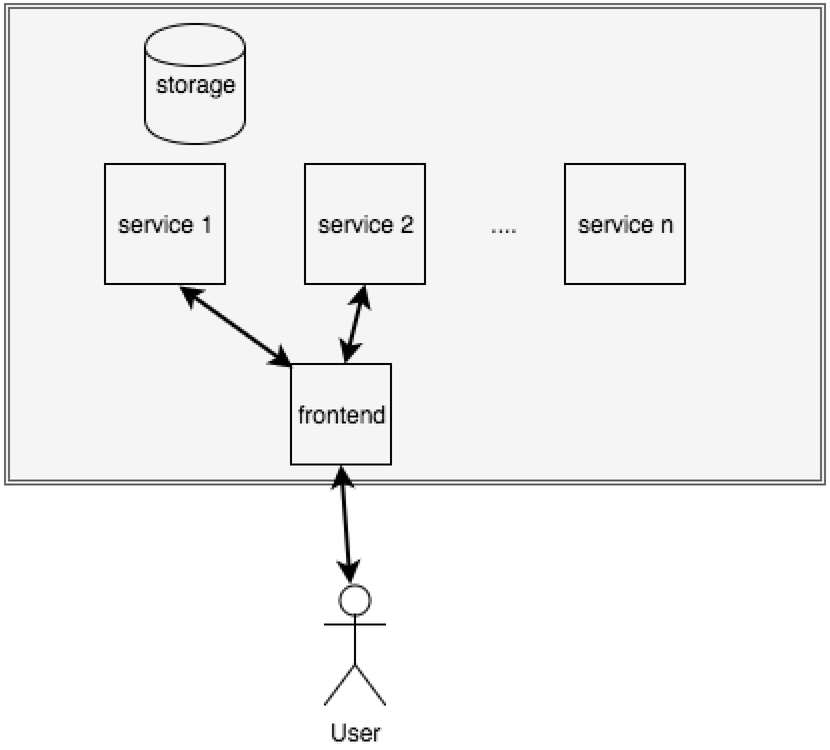
Summary

- Dockerfile best practices aim to
  - Keep image footprint small
    - Sometimes at expense readability
    - Multistage builds are a good compromise
  - Maintain clean control over containers
    - Easy to top and start
    - Flexible in the way they are executed
- Official Dockerfile best practices

Docker and Development

Microservices vs. Monoliths

- Small decoupled applications vs. one big app
- Developed independently
- Deployed and updated independently
- Scaled independently
- Better modularity
- Docker containers fit with microservice architecture



Microservices in Docker

Linking multiple containers

```
$ cd ~/docker-introduction/sample-code/mycomposeapp
```

- Let's build a simple application with two components
  - Web application using Python Flask
  - Redis message queue
- The app is already in mycomposeapp/app.py
- We want to run the app and redis as separate microservices
- Redis is already available as a docker image
  - `$ docker pull redis:alpine`
- We're going to have build a docker image for our app

Create our app

- Go into the mycomposeapp directory

```
$ cd ~/docker-introduction/sample-code/mycomposeapp
$ ls
$ gedit Dockerfile
```

- Contents of `Dockerfile`

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

- Build Docker image for app

```
$ docker build -t web .
```

Run containers as microservices

- First let's start our redis container

```
$ docker run -d --rm --name redis redis:alpine
```

- For our web container, we need a specific option to connect it to redis
  - --link <name of container>

```
$ docker run -d --rm --name web --link redis -p 5000:5000 web
```

- Once you start the web container go to web page to see counter

Disadvantages of this approach

- Complicated with shell/script commands
  - Managing service interactions
  - Adding/managing services
- Can't scale services
- Stopping and cleaning up services can be tedious
  - BTW, you'll need to stop each of those containers

```
$ docker stop web
$ docker stop redis
```

- Better tools exist..

Docker Compose

Docker as a dev environment

- A tool that let's you easily bootstrap complex microservice apps
- Allows interactive development
  - you can work on the code while the container is running
- Can be used for staging/production environments
- Uses a YAML based config file called the *docker-compose file*

The docker-compose file

- Service description file
- YAML
- By default: `docker-compose.yml`
- Specifies
    - Services
        - effectively containers that you will run
    - Volumes
        - filesystem mounts for containers
    - Networks
        - to be created and used by containers
- Have a look at the compose file reference

```yaml
---
version: "3"
services:
  db:
   image: db
   volumes:
     - data-volume:/var/lib/db
  redis:
    image: redis:alpine
  webapp:
    build: .
    ports:
      - 80:80
    networks:
        hostnet: {}

volumes:
  data-volume:

networks:
  hostnet:
    external:
      name: host
```

docker-compose services

- Each key in *services* dictionary represents a base name for a container
- Attributes of a service include
  - build
    - path or dictionary pointing to Dockerfile to build for container
  - image
    - Use a particular image for container
  - ports
    - expose ports for accessing application
  - volumes
    - mount into container

```
---
version: "3"
services:

  webapp:
    build: .
    ports:
      - 80:80
      - 443:443
    networks:
        hostnet: {}
  db:
   image: db
   volumes:
     - data-volume:/var/lib/db

  redis:
    image: redis:alpine
```

Docker Compose

Basic commands

```
docker-compose COMMAND [options] [args]
```

scale

Command	Description
`up`	Start compose
`down`	Stop & tear down containers/networks
`restart` <service name>	Restart a service

- Use `docker-compose -h` to view inline documentation

Exercise: convert our app to use `docker-compose`

- In same directory as previous example
- Create a file called `docker-compose.yml`
- Add our service definition:

```
---
version: "3"
services:
  web:
    build: .
    ports:
        - "5000:5000"
  redis:
    image: redis:alpine
```

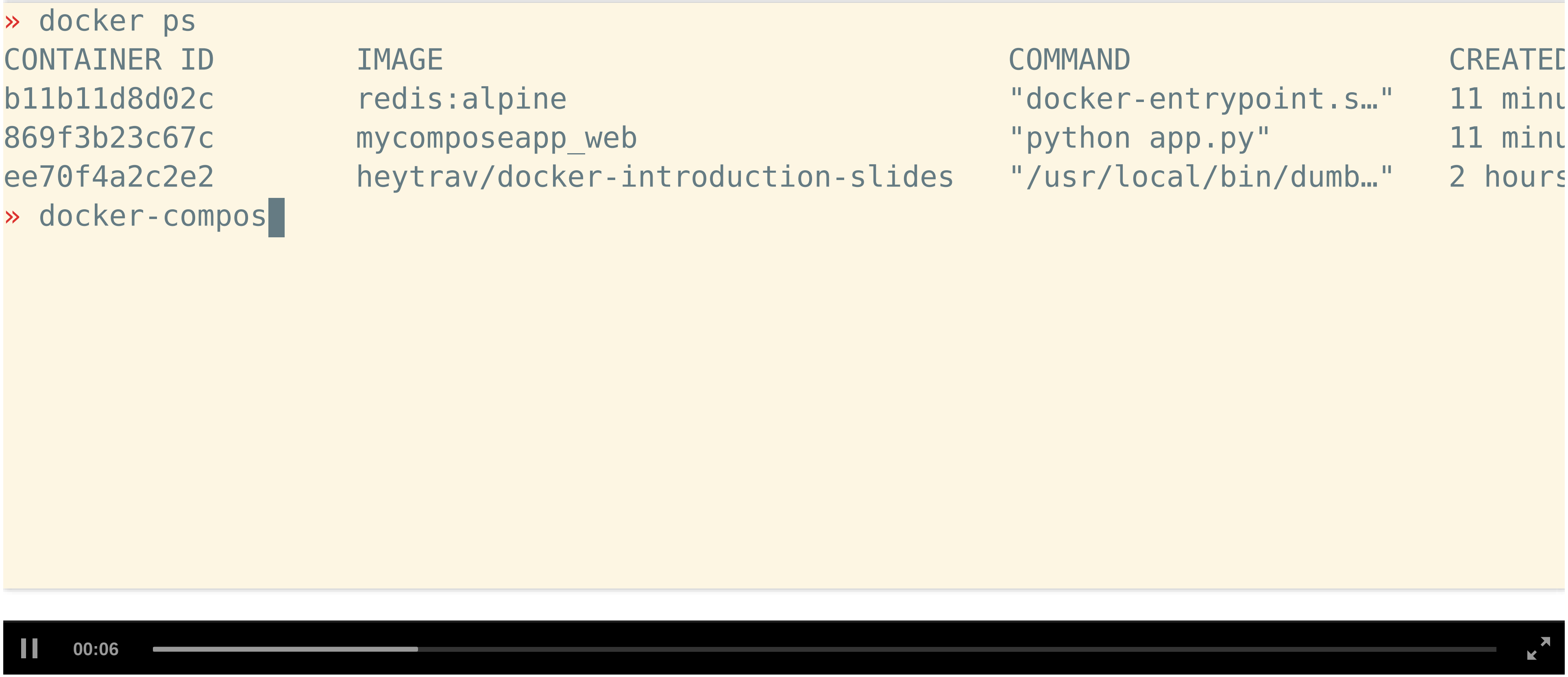
- Start our microservices

```
$ docker-compose up [-d]
```

Scaling services

```
$ docker-compose up -d --scale SERVICE=<number>
```

- Try scaling the redis service to 4 instances



```
» docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED
b11b11d8d02c        redis:alpine                           "docker-entrypoint.s…"   11 minu
869f3b23c67c        mycomposeapp_web                       "python app.py"          11 minu
ee70f4a2c2e2        heytrav/docker-introduction-slides     "/usr/local/bin/dumb…"   2 hours
» docker-compos
```

Stopping docker-compose

- In the directory where your `docker-compose.yml` file is:

-
  ```
  $ docker-compose stop
  ```

Summary

- `docker-compose` provides useful way to setup development environments
- Takes care of
  - networking
  - linking containers
  - scaling services

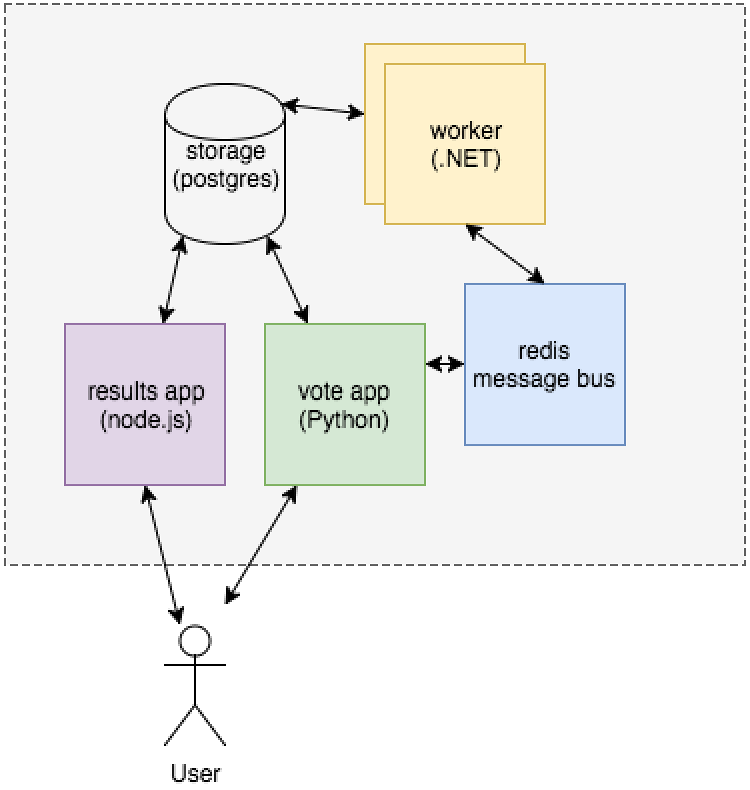
Docker Workflow Example

Taking apps from desktop to deployment

Example voting application

Microservice application consisting of 5 components

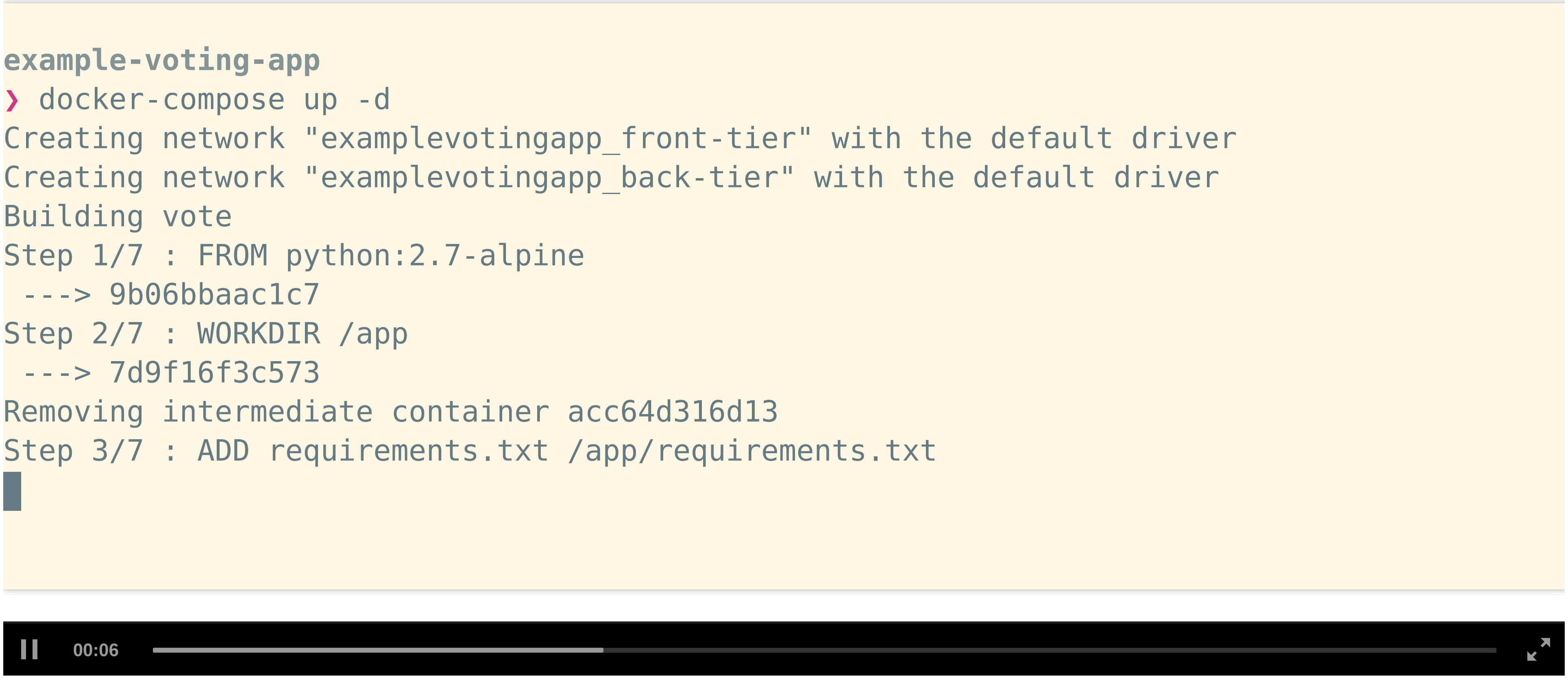
- Python web application
- Redis queue
- .NET worker
- Postgres DB with a data volume
- Node.js app to show votes in real time



```
$ git clone https://github.com/dockersamples/example-voting-app.git
$ cd example-voting-app
```

Start Application

```
$ cd example-voting-app
$ docker-compose up -d
```



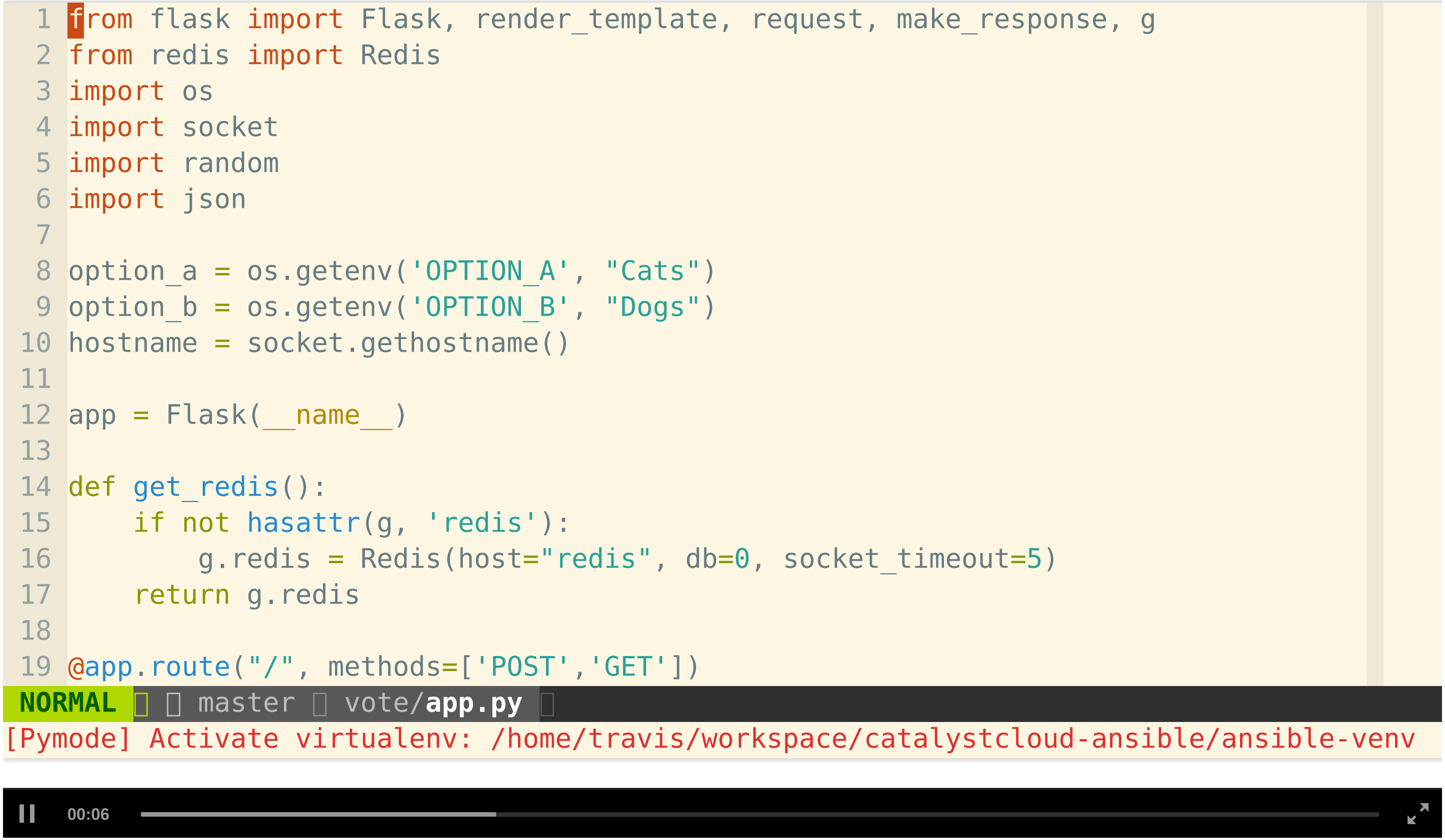
```
example-voting-app
❯ docker-compose up -d
Creating network "examplevotingapp_front-tier" with the default driver
Creating network "examplevotingapp_back-tier" with the default driver
Building vote
Step 1/7 : FROM python:2.7-alpine
 ---> 9b06bbaac1c7
Step 2/7 : WORKDIR /app
 ---> 7d9f16f3c573
Removing intermediate container acc64d316d13
Step 3/7 : ADD requirements.txt /app/requirements.txt
```

Vote and view results

Interactive development

- Open up `vote/app.py`
- On lines 8 & 9, modify vote options
- View change in voting application

Change vote options



```python
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
```

Developer workflow

- Push code to repository
- Continuous Integration (CI) system runs tests
- If tests successful, automate image build & push to a docker registry
- Easy to setup with existing services
  - DockerHub (eg. these slides)
  - GitHub
  - CircleCI
  - GitLab
  - Quay.io

Developer workflow

- Ship your artefact directly using docker-compose
  - Useful if you want to test an image immediately
- Tell docker-compose to rebuild the image
- Let's build and tag the image as YOURNAME/vote:v2 and push to hub.docker.com
- This will come in handy in an example we're doing later

```
$ docker-compose build vote
$ docker tag examplevotingapp_vote:latest YOURNAME/vote:v2
$ docker push YOURNAME/vote:v2
```

Summary

- With docker-compose it's relatively easy to develop on a microservice application
- Changes visible in real time
- Can easily package and distribute images for others to use

Deploying Applications

Deploying Applications

- Inevitable goal of developing apps is to deploy them somewhere
- Typically some kind of hosting provider
  - Bare metal
  - Cloud Provider
- We'll use Catalyst Cloud OpenStack

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

Orchestration

- Hosting may consist of multiple machines
- Once infrastructure is in place, need way to manage containers
  - networking
  - volume mounts
  - linking between containers
  - general monitoring, healthchecks
  - Deploying new images
- This is where orchestration tools come in

Container orchestration

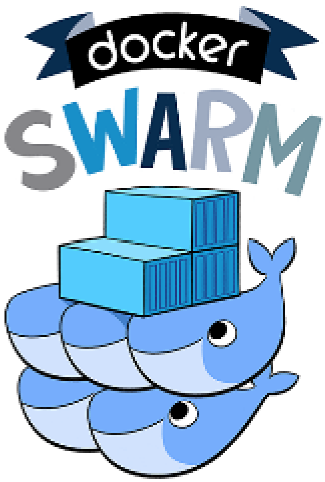
- Frameworks for container orchestration
  - Docker Swarm
  - Kubernetes
- Manage deployment/restarting containers across clusters
- Networking between containers (microservices)
- Scaling microservices
- Fault tolerance

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

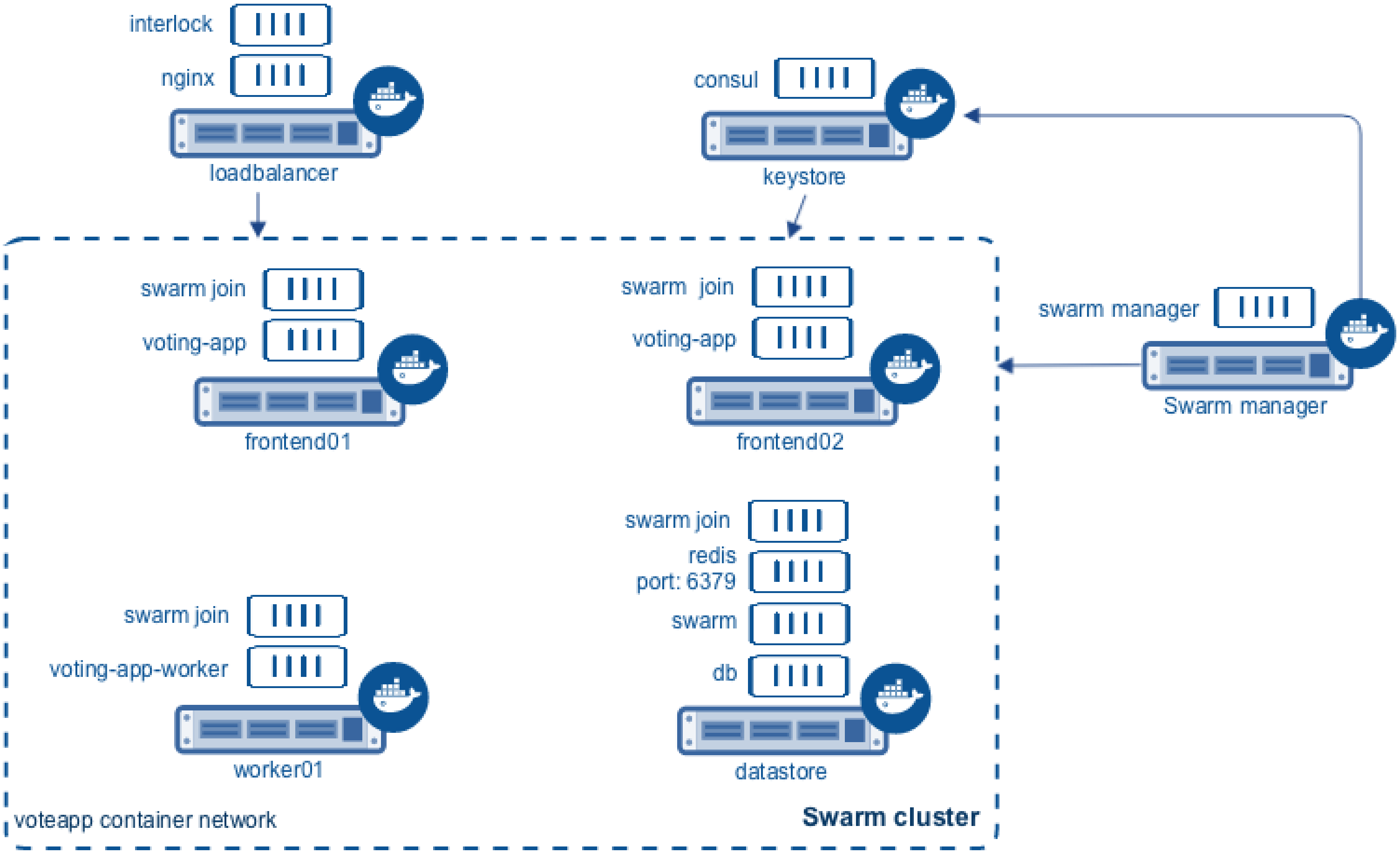
Docker Swarm

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



Docker Swarm

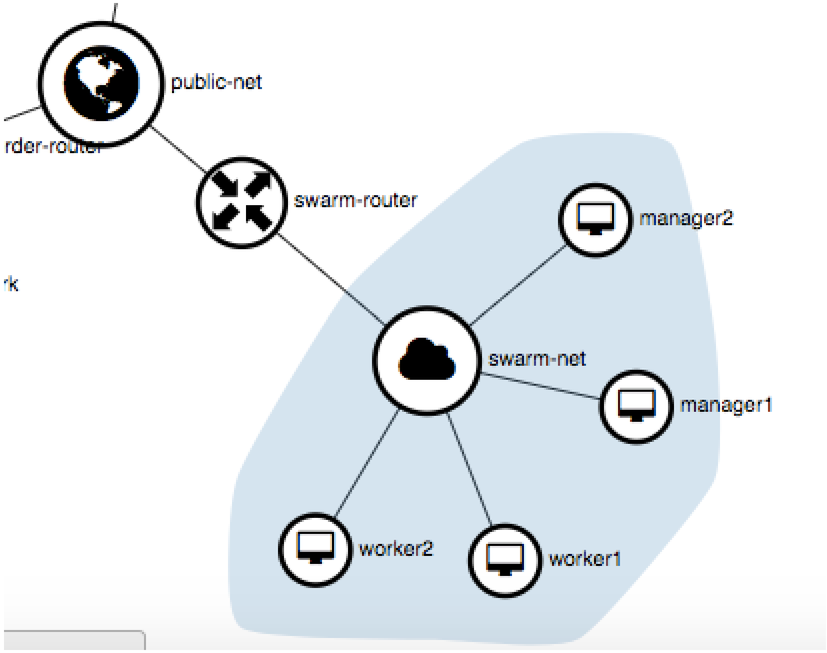
- Two types of machines or *nodes*
  - 1 or more *manager* nodes
  - 0 or more *worker* nodes
- Managers control global state of cluster
  - Raft Consensus Algorithm
  - If one manager fails, any other should take over


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

Deploying a Swarm Application

Setting up a cluster

- Need to:
  - provision machines
  - set up router(s)
  - set up security groups
- Preferable to use automation tools:
  - Chef
  - Puppet
  - Terraform
  - Ansible



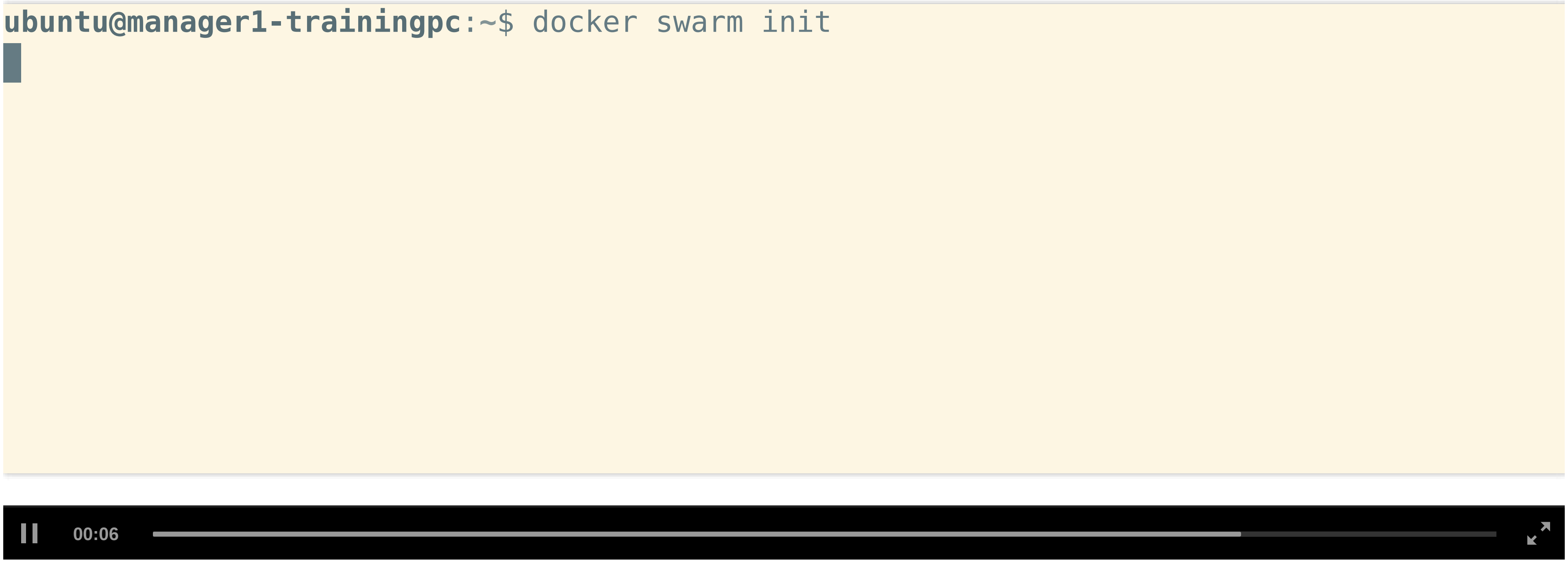
Create a cluster

```
$ cat ~/credentials.txt
$ source ~/os-training.catalyst.net.nz-openrc.sh
<enter os training password from ~/credentials.txt>

$ cd ~/docker-introduction/ansible
$ ansible-playbook -i cloud-hosts -K -e suffix=-$( hostname ) \
    create-swarm-hosts.yml
```

This will do stuff for while, good time for some coffee

Create Swarm

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



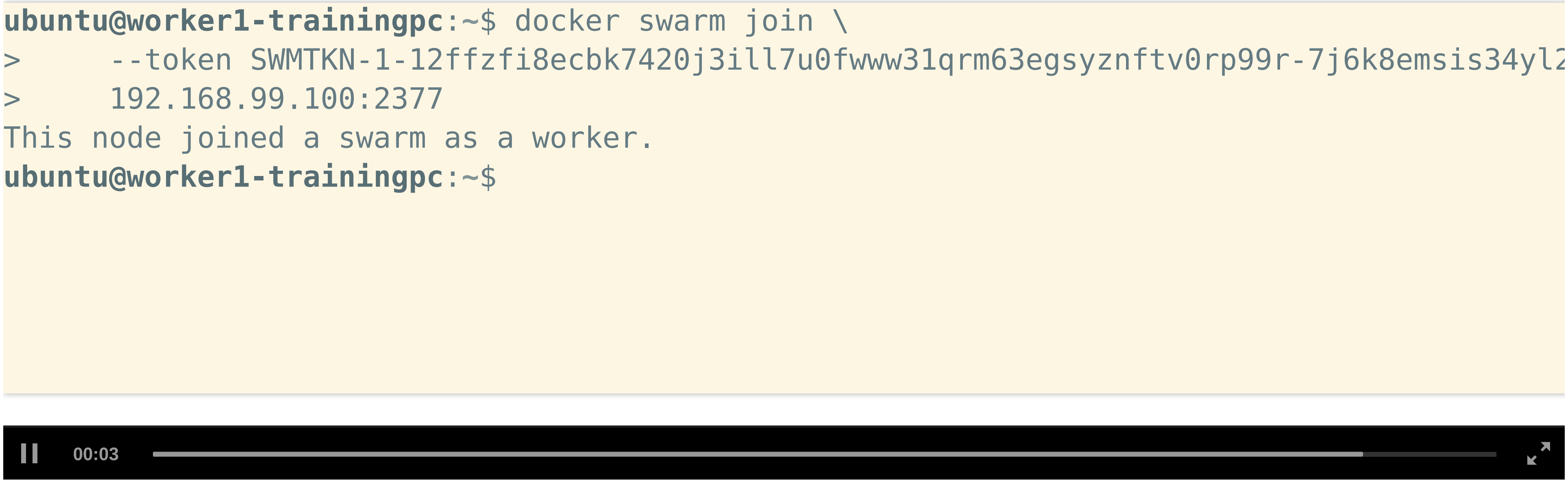
```
ubuntu@manager1-trainingpc:~$ docker swarm init
```

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
```

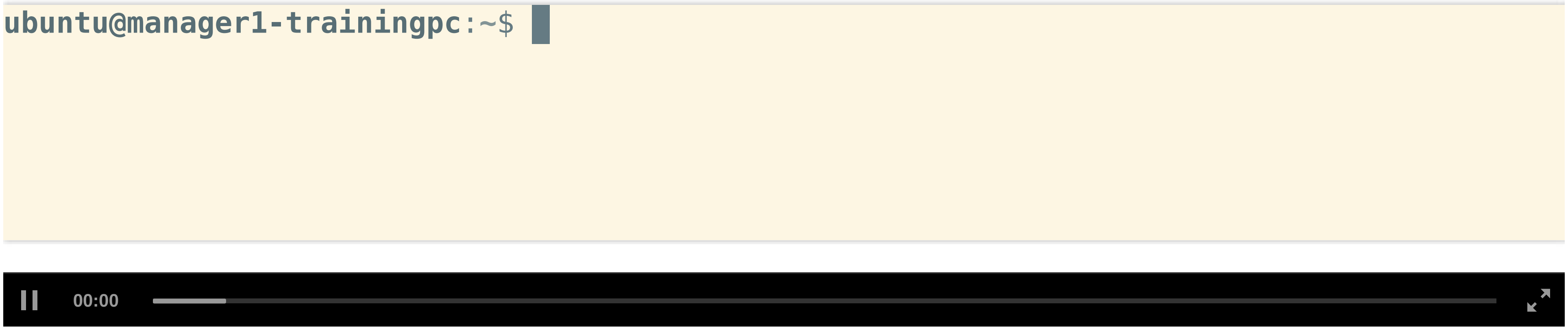


```
ubuntu@worker1-trainingpc:~$ docker swarm join \
>     --token SWMTKN-1-12ffzfi8ecbk7420j3ill7u0fwww31qrm63egsyznftv0rp99r-7j6k8emsis34yl2
>     192.168.99.100:2377
This node joined a swarm as a worker.
ubuntu@worker1-trainingpc:~$
```

Repeat this for `worker2`

Check nodes

```
$ docker node ls
```



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

Swarm Stack File

- Service description
- YAML format
- Similar to file used for `docker-compose`
- A few differences
  - No `build` option
  - No shared volumes

```
# 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:
```

Deploying voting app

Upload docker-stack.yaml to manager node

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

Deploy application

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

Monitor deploy progress

```
$ watch docker stack ps 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

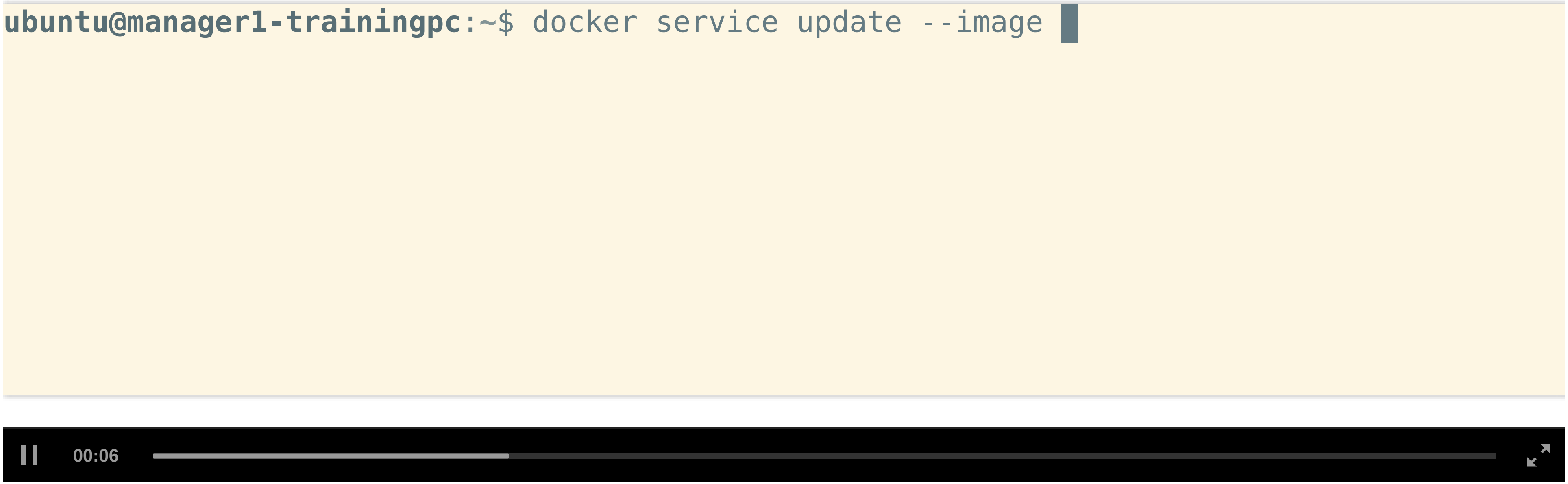
Scale services

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

Look at the changes in the visualizer

Update a service

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



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

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
  - 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

Tear down your cluster

When you're done playing around with the voting app, please run the following

```
$ ansible-playbook -i cloud-hosts -K  -e suffix=-$( hostname ) \
   remove-swarm-hosts.yaml
```

Summary

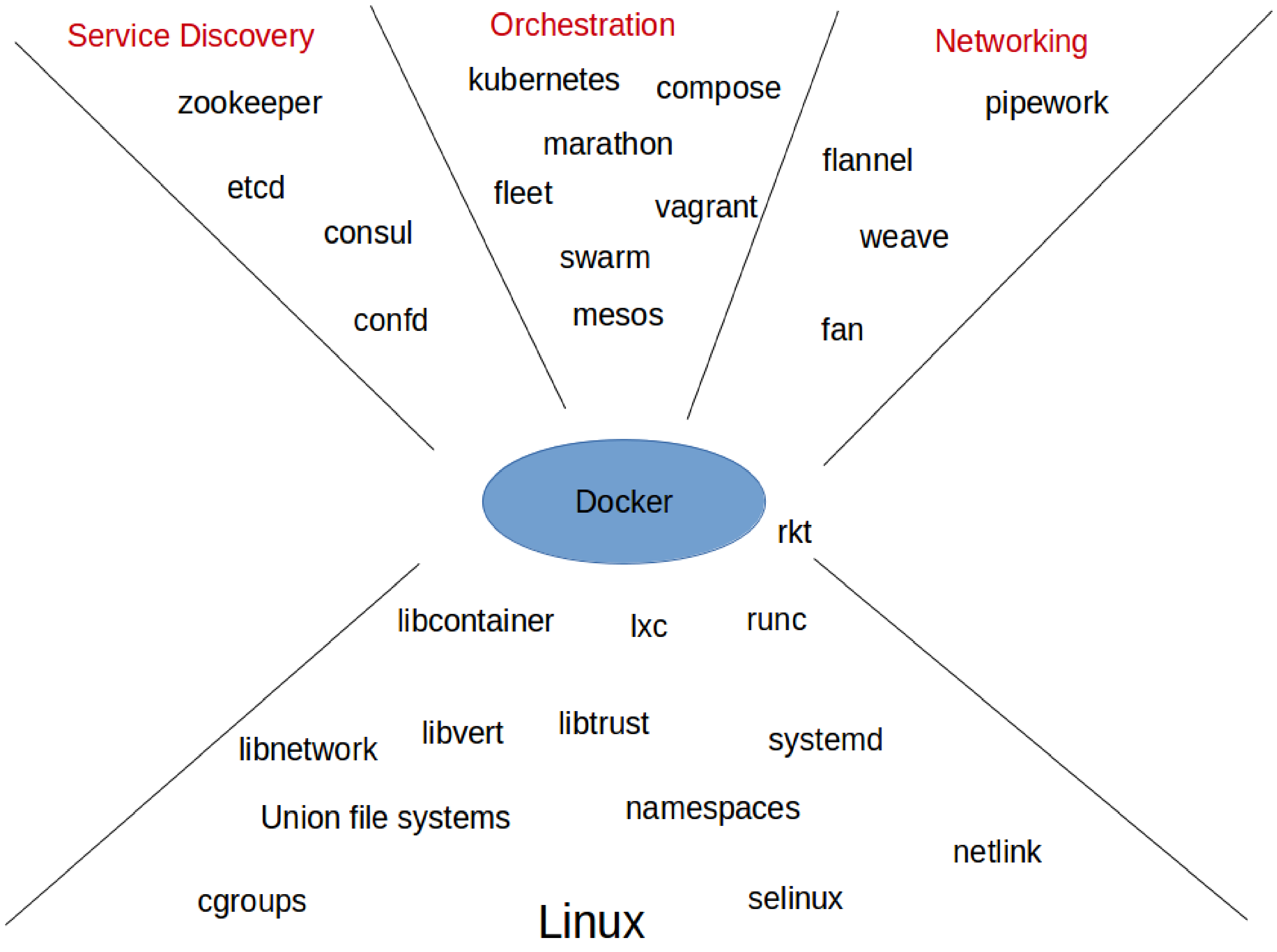
- Created a cluster with a cloud provider using ansible
  - 1 manager node
  - 2 worker nodes
- Deployed microservice for voting app in Docker Swarm
- Scaled service from 2 to 3 services
- Rolling-Updated image

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