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
$ git checkout oct-10
```

- 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
```

- Ansible installed
- Activated and sourced a python virtualenv

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

```
$ cd docker-introduction
$ ansible-playbook -K ansible/docker-install.yml
```

Fetch and run slides

```
$ docker run --name docker-intro -d --rm \
        -p 8000:8000 heytrav/docker-introduction-slides:oct-10
```

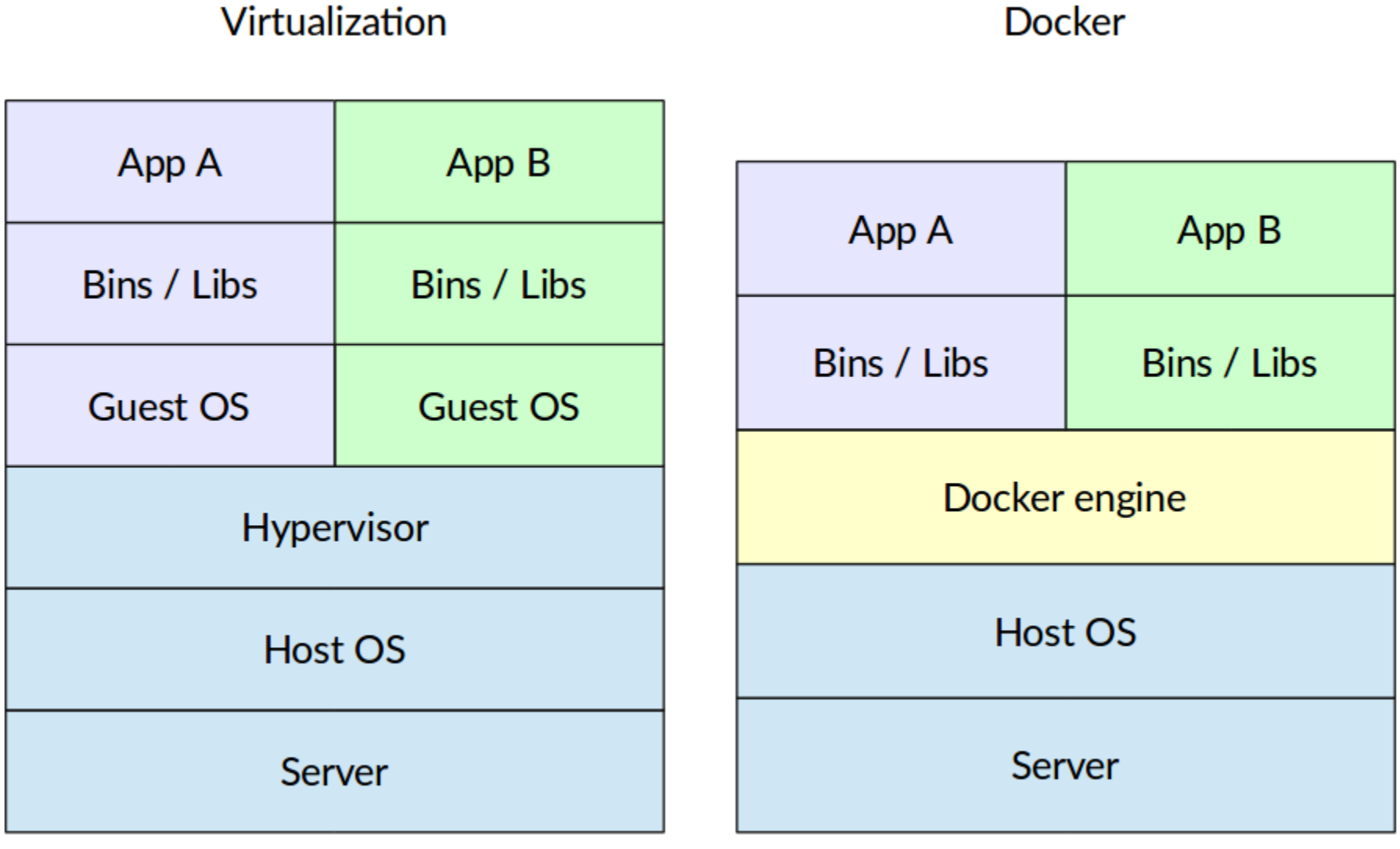
Follow along with course slides: http://localhost:8000

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



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

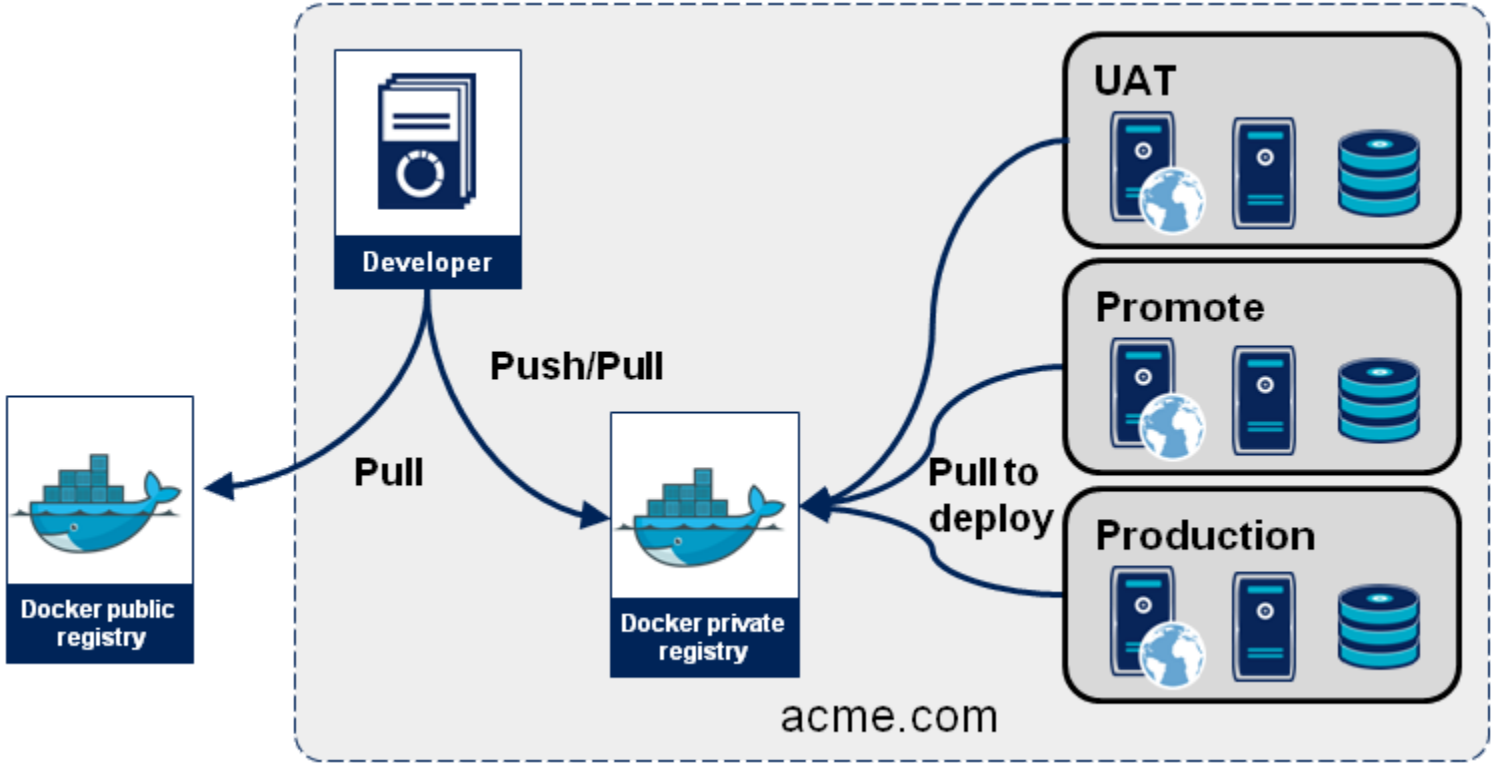
A command-line tool for programmatically defining the contents of a Linux container in code, which can then be versioned, reproduced, shared, and modified easily just as if it were the source code to a program

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.

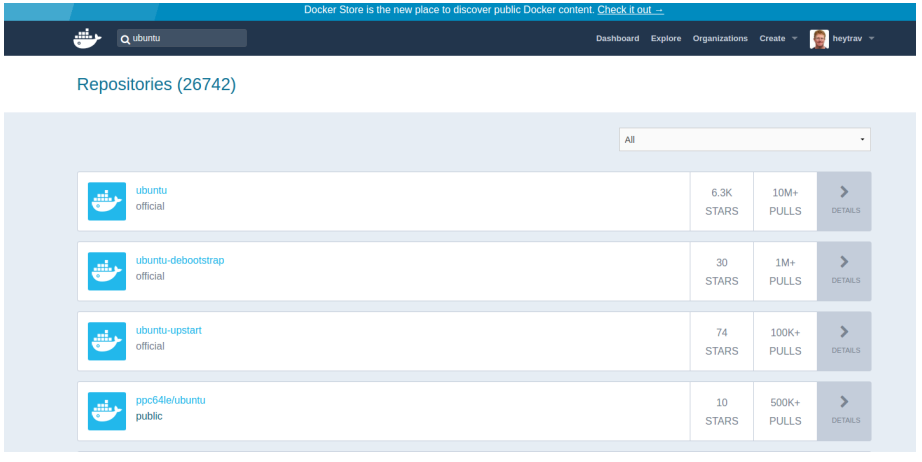


Docker Portability

- 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)

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` or `--help` displays some docs
- Comprehensive online docs on Docker website

Basic client usage

`docker command [options] [args]`

```
$ 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: 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
      .
      .
```

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.

Start a shell

`docker run image [command]`

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

- Docker starts container using alpine image
- `[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

```
➜  ~ docker run alpine
```

Exercise: Start an *interactive* shell

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

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

- 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

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 a name to the running alpine container

```
$ 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 m
```

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

- Run the image: `dockersamples/static-site`
- Name it *static-site*
- Pass your name to the AUTHOR environment variable
- Map port 8081 to 80 internally (hint `8081:80`)
- Container terminates when stopped
- Go to localhost:8081
- CTRL-C to exit

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

```
❯ docker run --name static-site --rm \
        -e AUTHOR="Trav" -p 8081:80 dockersamples/static-site
Unable to find image 'dockersamples/static-site:latest' locally
```

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 p
```

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 stati
```

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 stat
```

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

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

```
$ docker stop 25eff330a4e4
```

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
➜  ~
```

How Docker works

Components of Docker

Docker Image

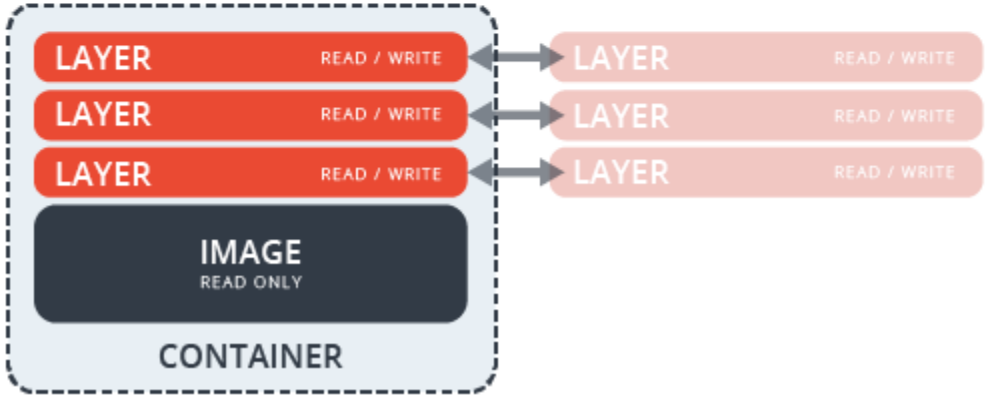
contains basic read-only image that forms the basis of container

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

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
- Image must have a name in lower case letters
- Tag is optional. Implicitly *:latest* if not specified
  - `postgres:9.4`
  - `ubuntu == ubuntu:latest == ubuntu:16.04`
- Url and username if pushing to registry
  - `docker.io/username/my-image`
  - `my.reg.com/my-image:1.2.3`

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

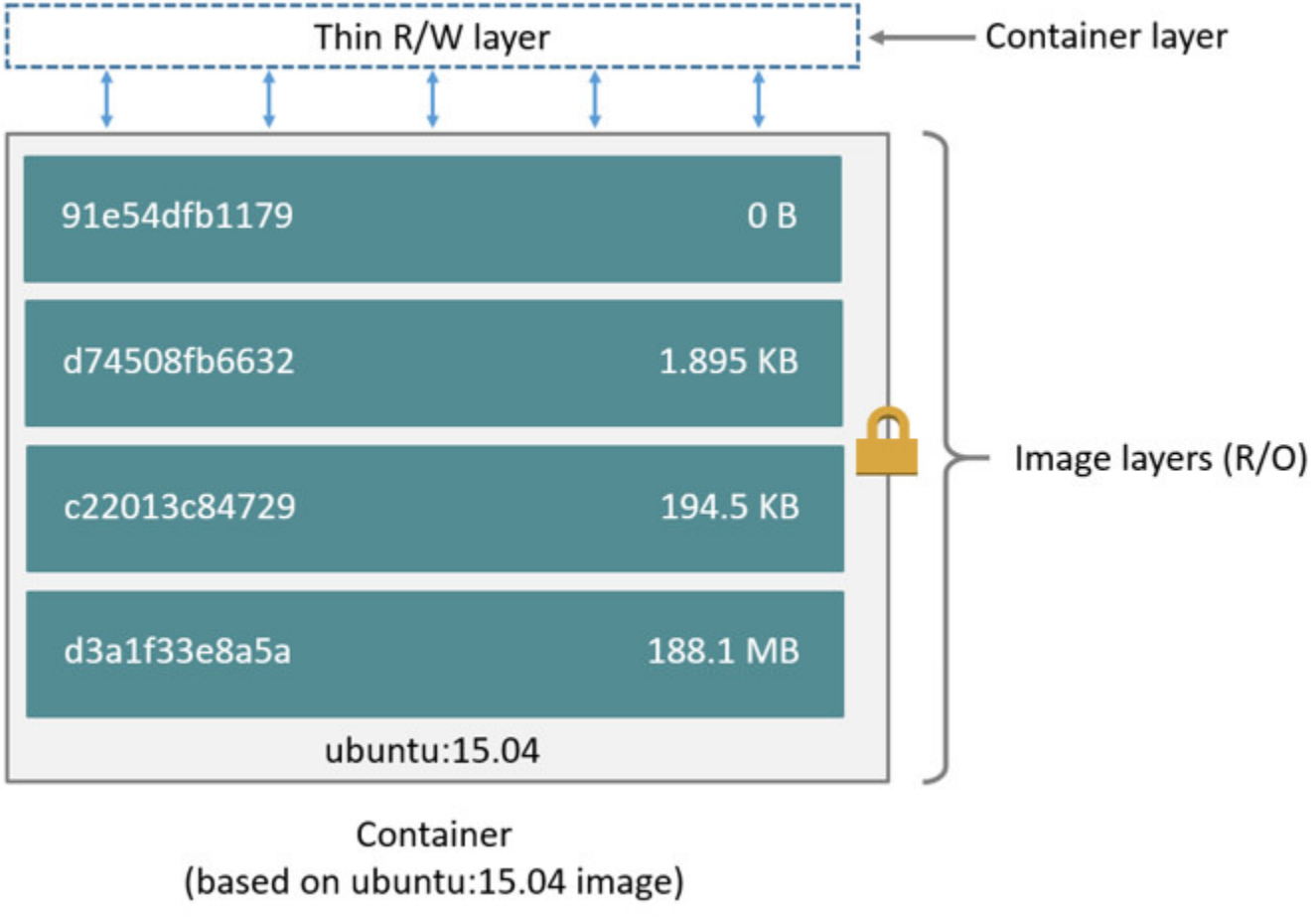
Child Images

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

Image layers



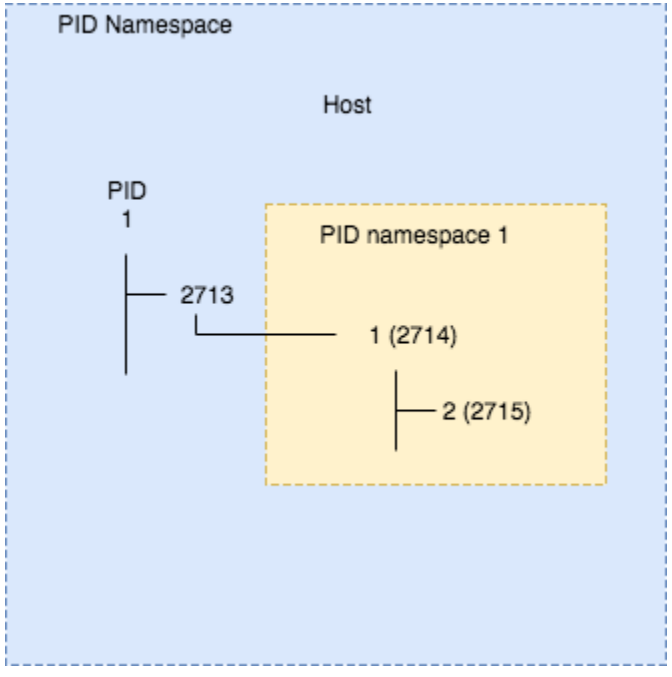
Sharing image layers

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

Container basics

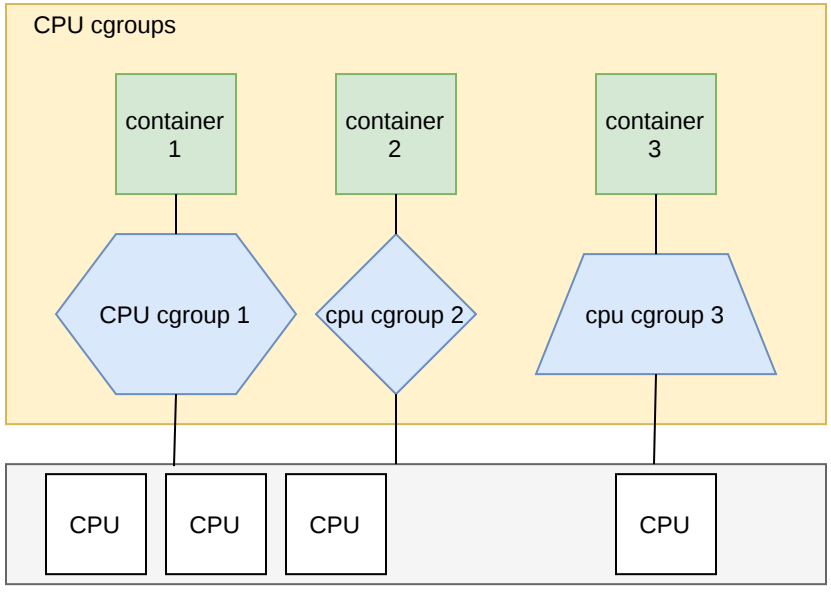
Namespaces

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



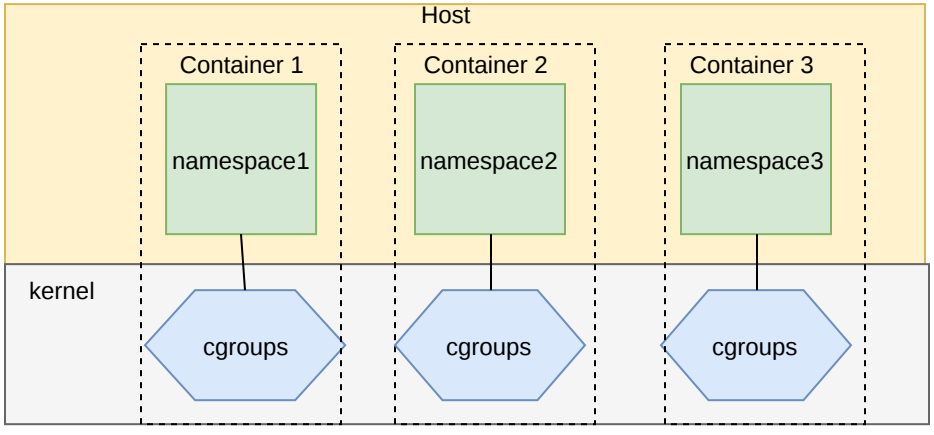
Cgroups

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



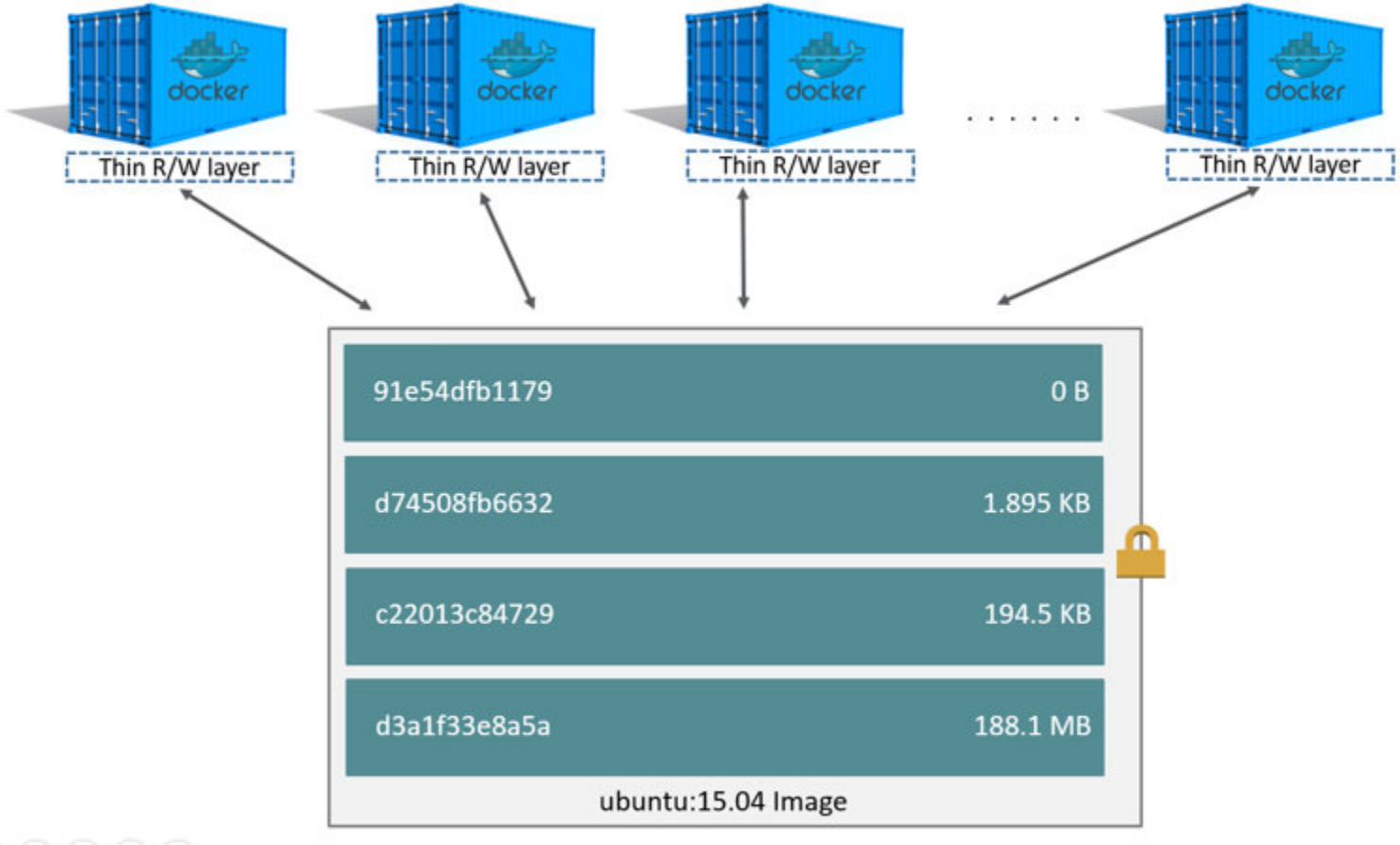
Combining the two

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



Container layering

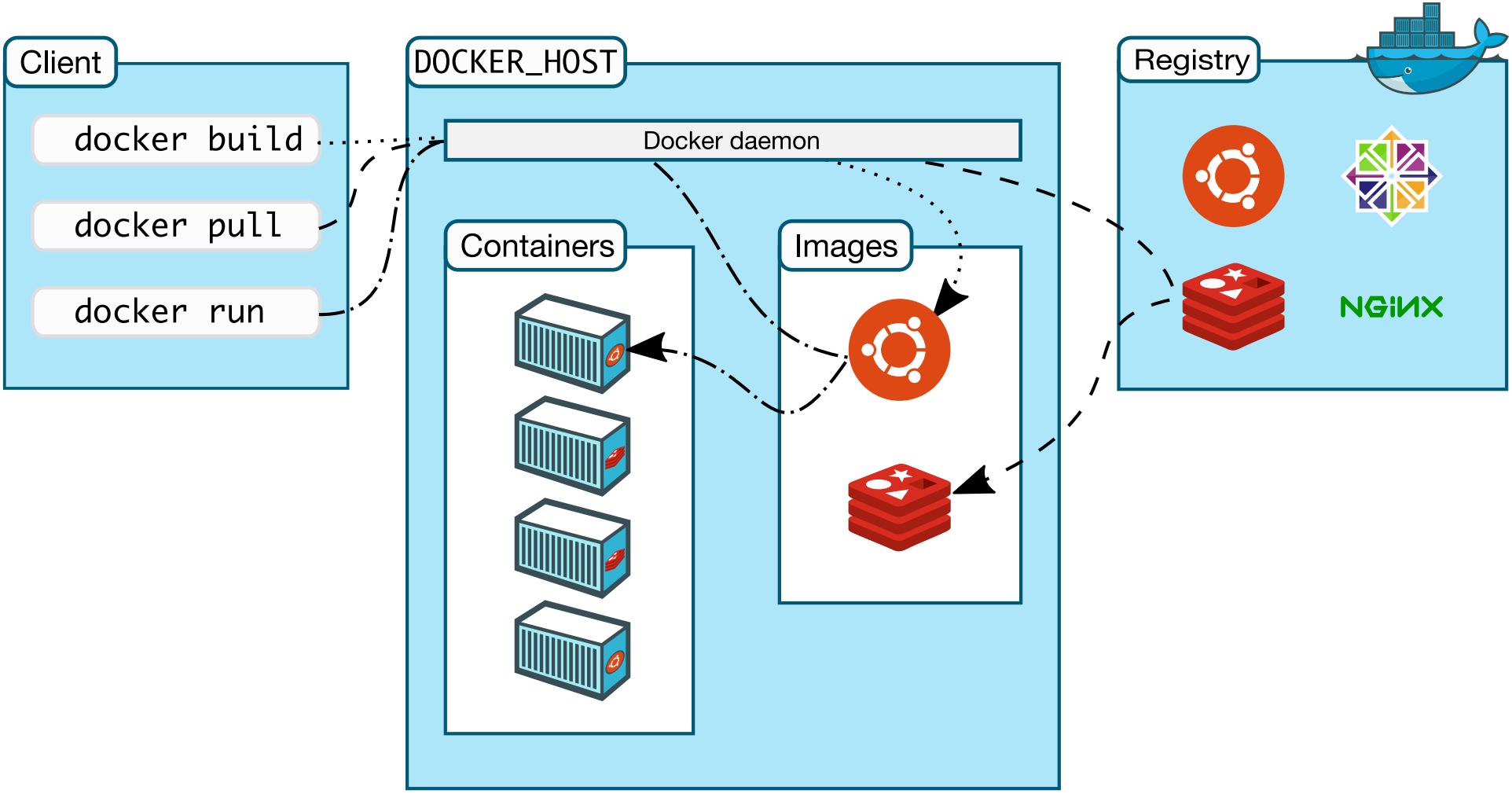
- Container creates its own read/write layer on top of image
- Multiple containers each have own read/write layer, but can share the actual image



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



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

docker build

Build Docker images

```
docker build [options] image[:tag] path
```

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

Exercise: build docker image using Dockerfile

- Call image `acme/my-base-image`
- Tag it `1.0`

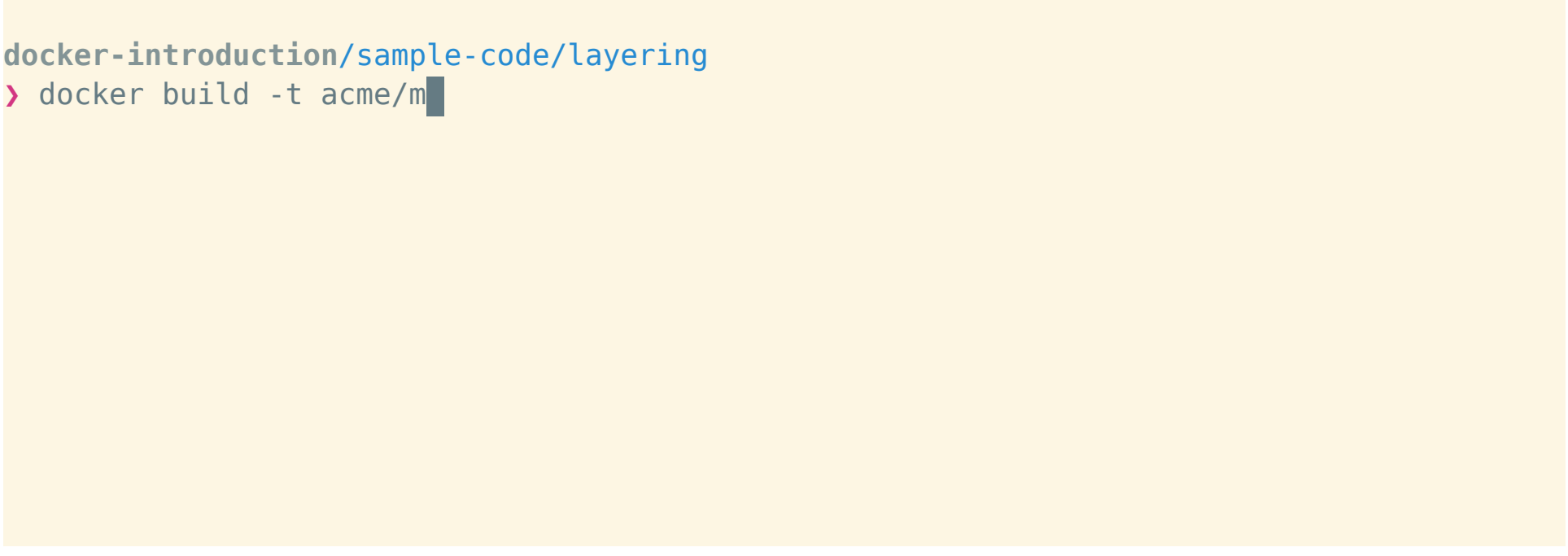
```
$ docker build -t acme/my-base-image:1.0  .
```

```
docker-introduction/sample-code/layering
❯ docker build -t acme/my-base-image:1.0 .
Sending build context to Docker daemon  4.096kB
Step 1/2 : FROM ubuntu:16.10
```

Exercise: Build child image

- Create `acme/my-final-image`
- Tag `1.0`
- Use `Dockerfile.child` to build image

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



Exercise: Compare base and child image layers

- Use: `docker history 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 .
```

```
➜   docker build -t
```

Note: please replace YOURNAME with your Docker Hub username

Run the container

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

```
➜   docker run -p 8888
```

...Now open your test webapp

Login to a registry

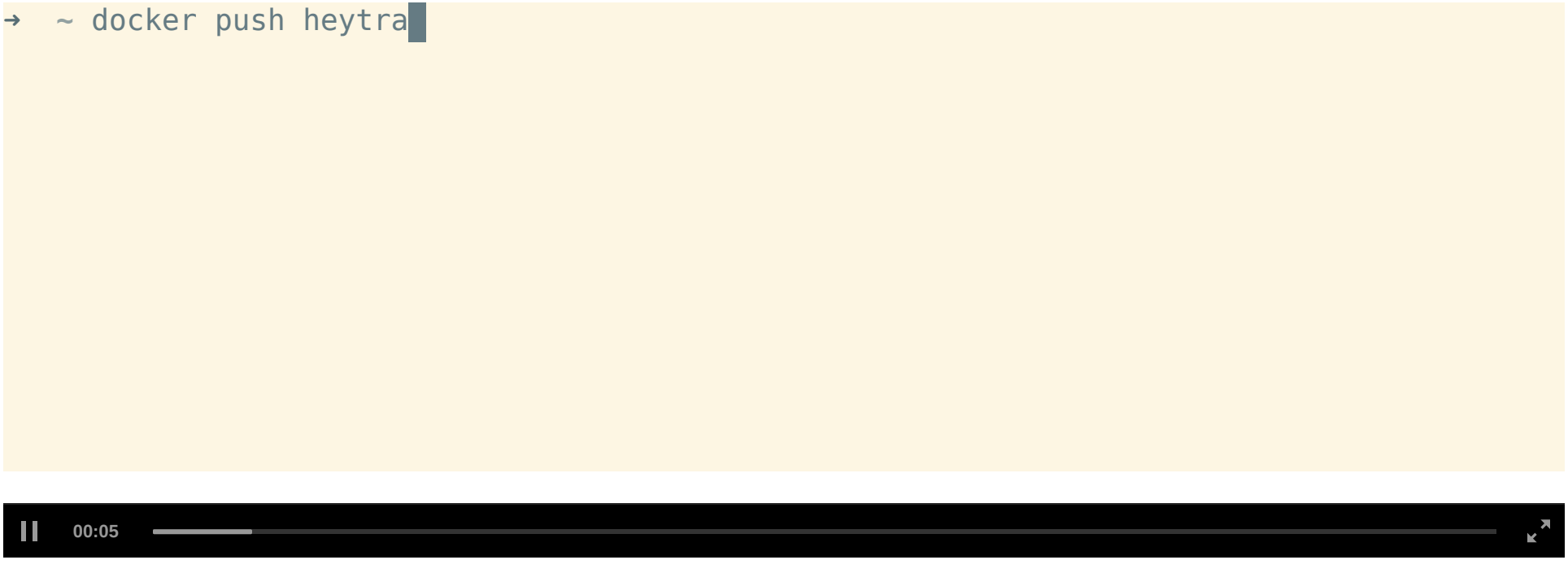
```
$ docker login <registry url>
```

```
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
```



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 .
```

- 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)

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

- 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/
```

ADD vs COPY

- Problem with ADD

```
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

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

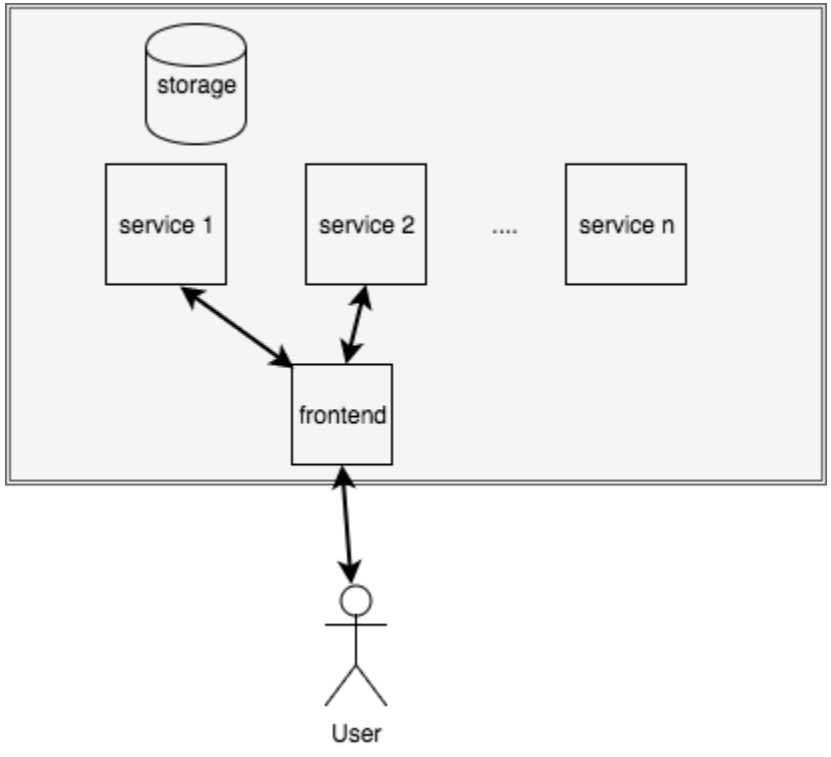
```
ENTRYPOINT ["python", "manage.py"]
CMD ["test"]
```

- 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 Development

Microservices vs. Monoliths

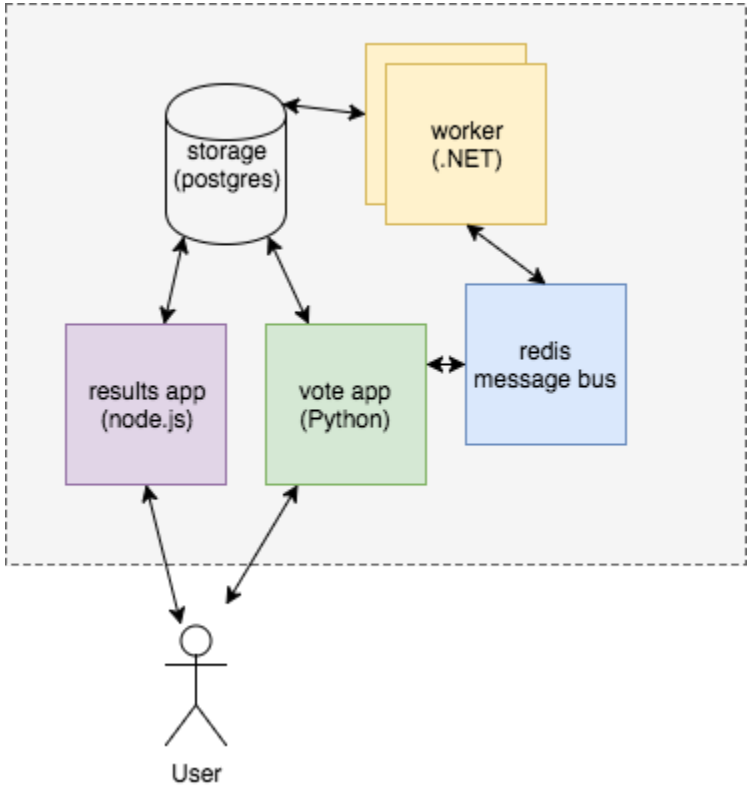
- Small decoupled applications vs. one big app
- Developed independently
- Deployed and updated independently
- Scaled independently
- Better modularity
- Docker containers fit with microservice architecture



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
```

Build vote app components

```
$ docker build -t vote vote
```

```
$ docker build -t result result
```

```
$ docker build -t worker worker
```

```
$ docker image ls
```

Run microservices

```
#! /bin/bash
# Helper services
docker run --rm -d -p 6379:6379 --name redis redis:alpine
docker run --rm -d --name db postgres:9.4

# Application
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
```

`--link a:b`

Link container *a* to container *b*

`-v ./path:/var/lib/path`

Mount a directory as a volume

`-p 8080:80`

Map port in container

Disadvantages of this approach

- Complicated with shell/script commands
  - Managing service interactions
  - Adding/managing services
- Can't scale services
- Better tools exist..

Docker Compose

Docker as a dev environment

- Declarative YAML syntax
- Specifies
  - Services
  - image or build
  - volumes
  - environment variables
- Interactive development
- Can be used for staging/production environments

```
---
# docker-compose.yml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
    volumes:
      logvolume01: {}
```

Docker Compose

Basic commands

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

scale

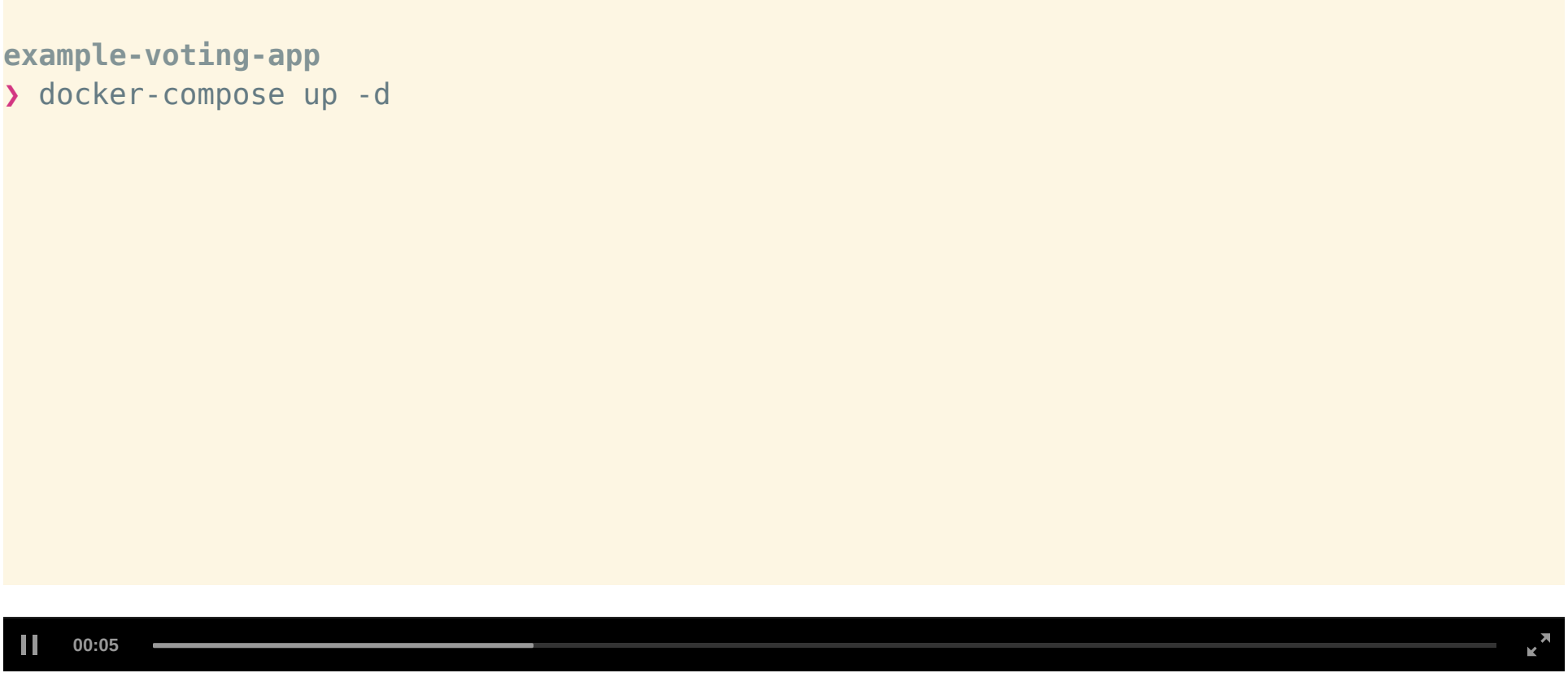
Command	Description
`up`	Start compose
`down`	Stop & tear down containers/networks
`restart` <service name>	Restart a service

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

Have a look at the documentation

Docker Compose Example

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



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

```
example-voting-app
❯ vim vote/app.py
```

Scaling services

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

```
example-voting-app
❯ docker ps
dockerCONTAINER ID        IMAGE                                  COMMAND                  C
215ebf4f9d52        postgres:9.4                           "docker-entrypoint..."   4 minut
8b60a02a898f        examplevotingapp_vote                  "python app.py"          4 minut
bc4de04d5264        redis:alpine                           "docker-entrypoint..."   4 minut
3e264f1af5a9        examplevotingapp_worker                "/bin/sh -c 'dotne..."   4 minut
00802d4f0393        examplevotingapp_result                "nodemon --debug s..."   4 minut
05e6a293c67b        heytrav/docker-introduction-slides     "/usr/local/bin/du..."   15 hour

example-voting-app
❯ docker-
```

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

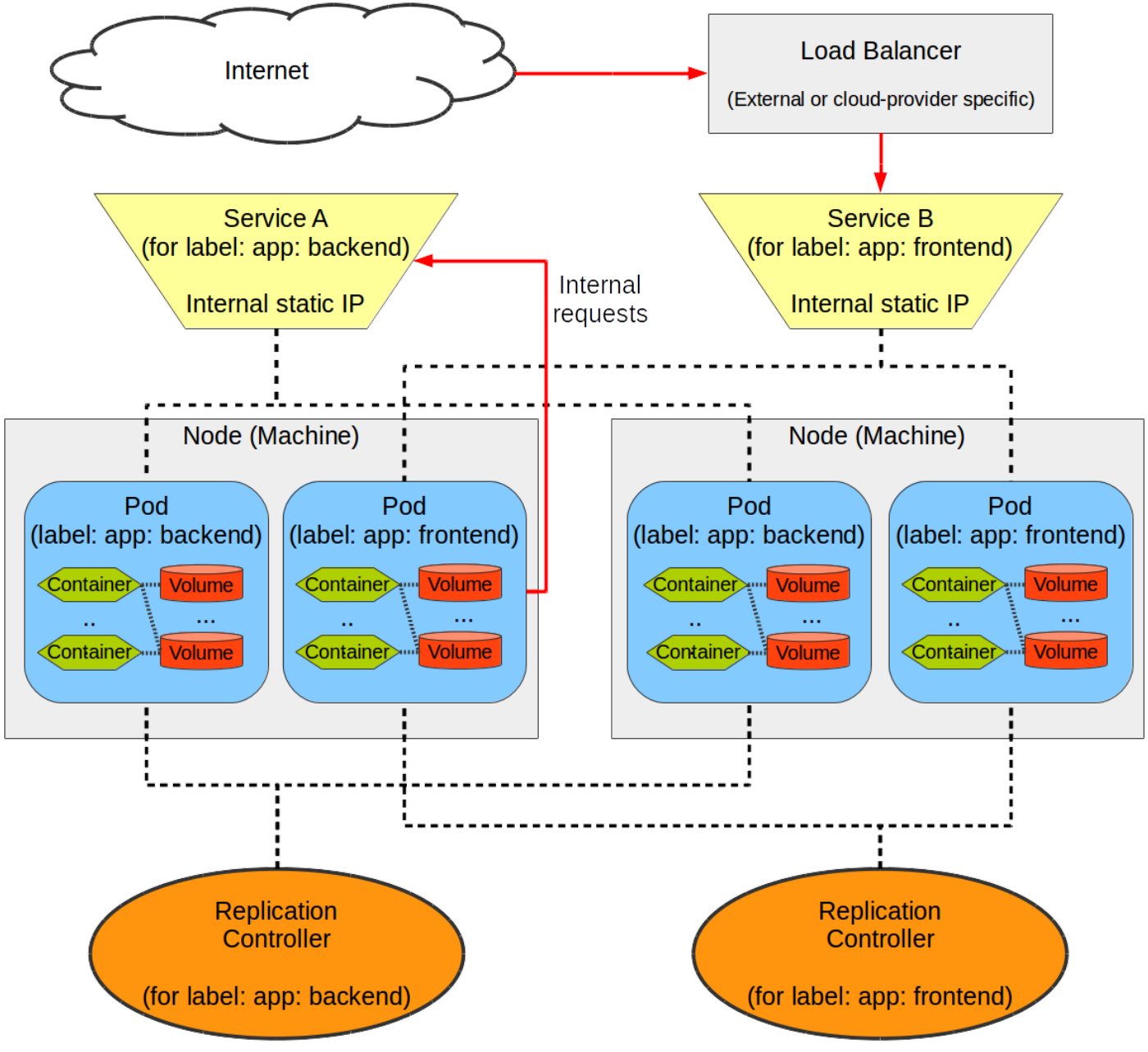
Container orchestration

- 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


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)

Kubernetes Components

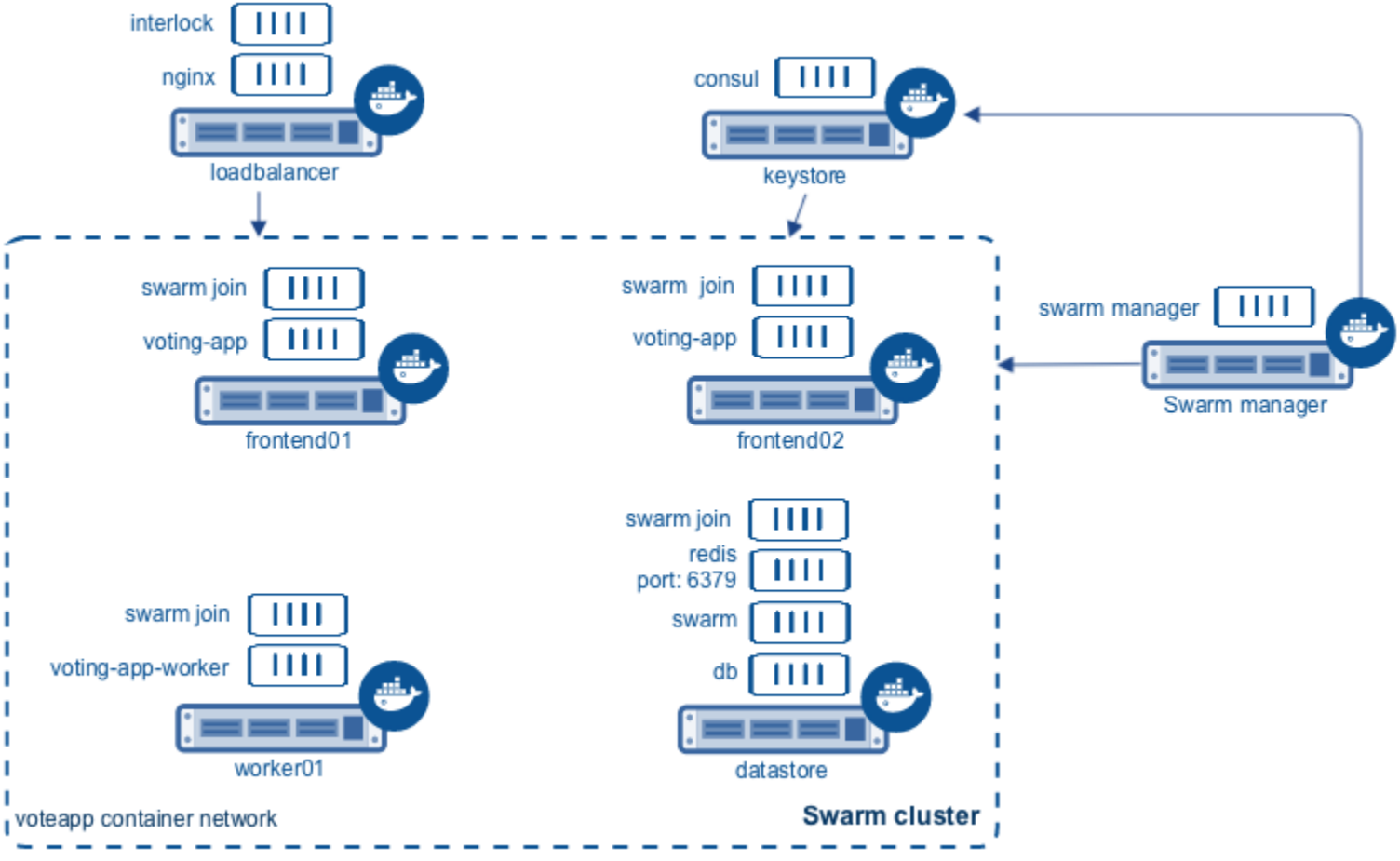
- Pods - an ephemeral group of co-scheduled containers that together provide a service
- Flat Networking Space - each pod has an IP and can talk to other pods, within a pod containers communicate via localhost (need to manage ports)
- Labels - Key value pairs, used to label pods and other objects so 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

Swarm Stack File

- 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:
```

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
```

Verify stack is running

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

```
master
❯ docker stack ps
```

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

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

```
example-voting-app
❯ docker build -t yourname/vote
```

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
❯ 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
❯
```

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

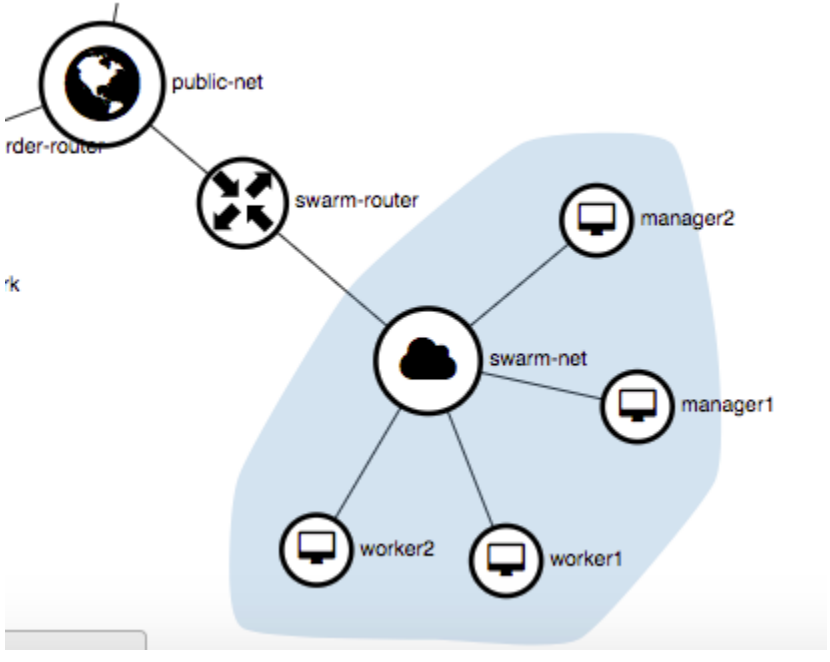
Running apps in the cloud

Goals

- Set up cluster of multiple machines
  - Catalyst Cloud (OpenStack)
- Install Docker on each machine
- Initialise a swarm
- Deploy our voting app
- Run through a few typical scenarios
  - Rolling update with *vote:v2*
  - Drain node for maintenance

Setting up 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

```
$ cd ~/catalystcloud-ansible/example-playbooks/docker-swarm-mode
$ ansible-playbook --ask-sudo-pass \
    --extra-vars "suffix=-$( hostname )" \
    create-swarm-hosts.yaml
```

Create Swarm

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

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

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
```

Repeat this for `worker2`

Check nodes

```
$ docker node ls
```

```
ubuntu@manager1-trainingpc:~$ docker node ls
ID                            HOSTNAME              STATUS              AVAILABILITY
d1av3sf7qxmlbgtsc3jjpfw7b *   manager1-trainingpc   Ready               Active
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
```

```
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
```

00:00

Powered by asciinema

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 heytrav/vote vote_vote
```

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

Now go to the voting app and verify the change

Developer workflow

- Push code to repository
- Continuous Integration (CI) system runs tests
- If tests successful, automate image build & push to a docker registry
- Manually/automatically run `docker service update`
- Easy to setup with existing services and automation tools like Ansible
  - DockerHub (eg. these slides)
  - GitHub
  - CircleCI
  - GitLab
  - Quay.io

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

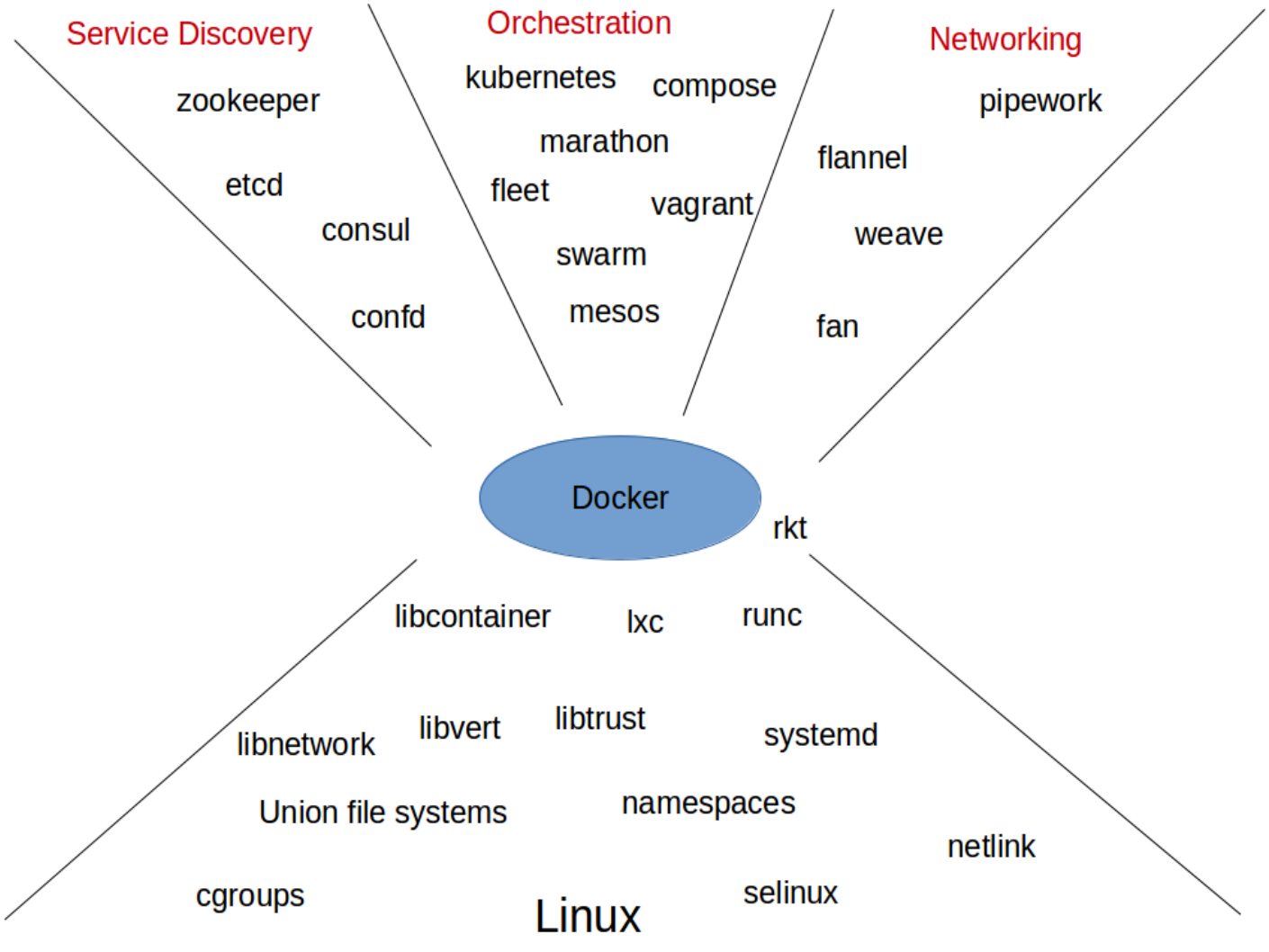
```
$ ansible-playbook -K  --extra-vars "suffix=-$( hostname )" remove-swarm
```

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