



Introduction to Docker

Presented by **Travis Holton**

Introduction to Docker



Administrative stuff

- Bathrooms
- Fire exits

About 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

Course Outline

- Setup
- Introduction to Containers
 - About containerisation
 - Benefits of
- Docker Platform
 - Workflow
 - Portability
- First Steps With Docker
 - Command line interaction
 - Basic commands

- **How Docker Works**
 - Components of Docker
 - Images, Containers, Registries
- **Images and Containers**
 - How images work
 - How containers work (cgroups, namespaces)

- Dockerising Applications
 - Building images
 - Dockerfile
 - Docker registry
- Dockerfile Best Practices
 - Images
 - Dockerfile
 - Multi-stage builds

- Docker Development Practices
 - Designing containerised applications
- Docker and Development
 - Microservices
 - Linking containers
 - docker-compose

- Local Development
 - Demo voting app
- Deploying Applications
 - Container orchestration

- **Kubernetes**
 - About Kubernetes
 - Architecture
 - Deploy demo

- Deploying Swarm Apps
- Wrap Up

Setup

Fetch course resources

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

```
cd ~/docker-introduction
```

- This folder contains:
 - Slides for reveal.js presentation
 - docker-introduction.pdf
 - Ansible setup playbook
 - Sample code for 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
 - Configuration management

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, you can 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
```

Setup Docker

- This playbook installs:
 - latest Docker *Community Edition*
 - docker-compose
- 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 the **course slides**

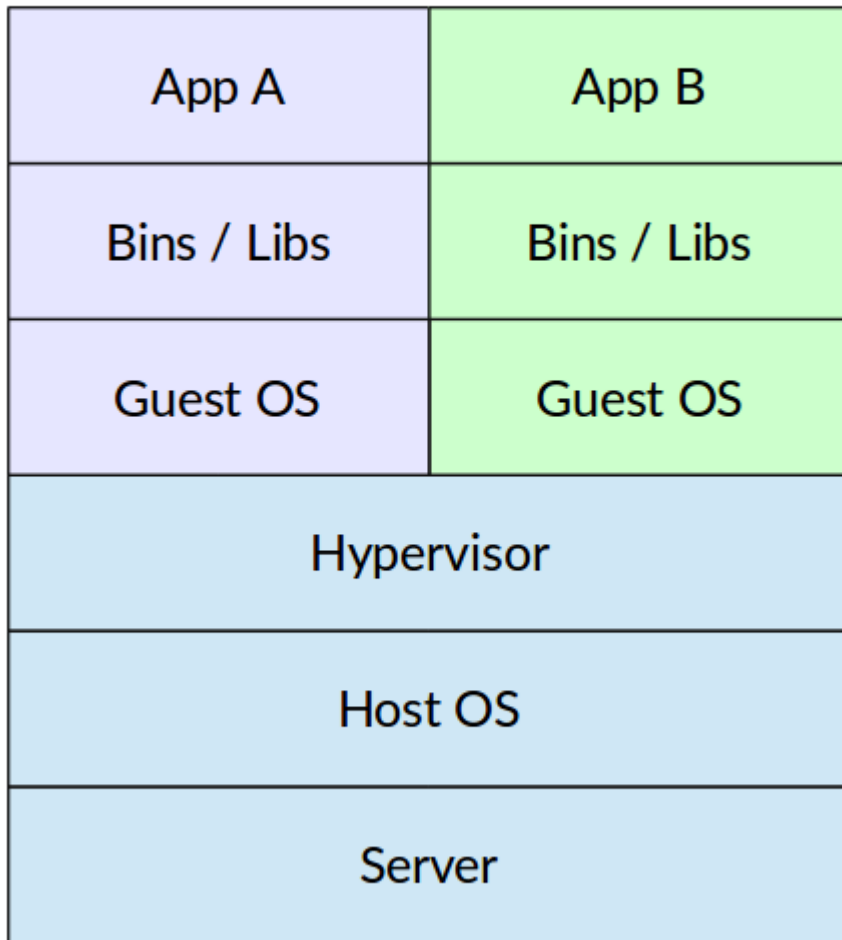
Introduction to Containers

What is containerisation?

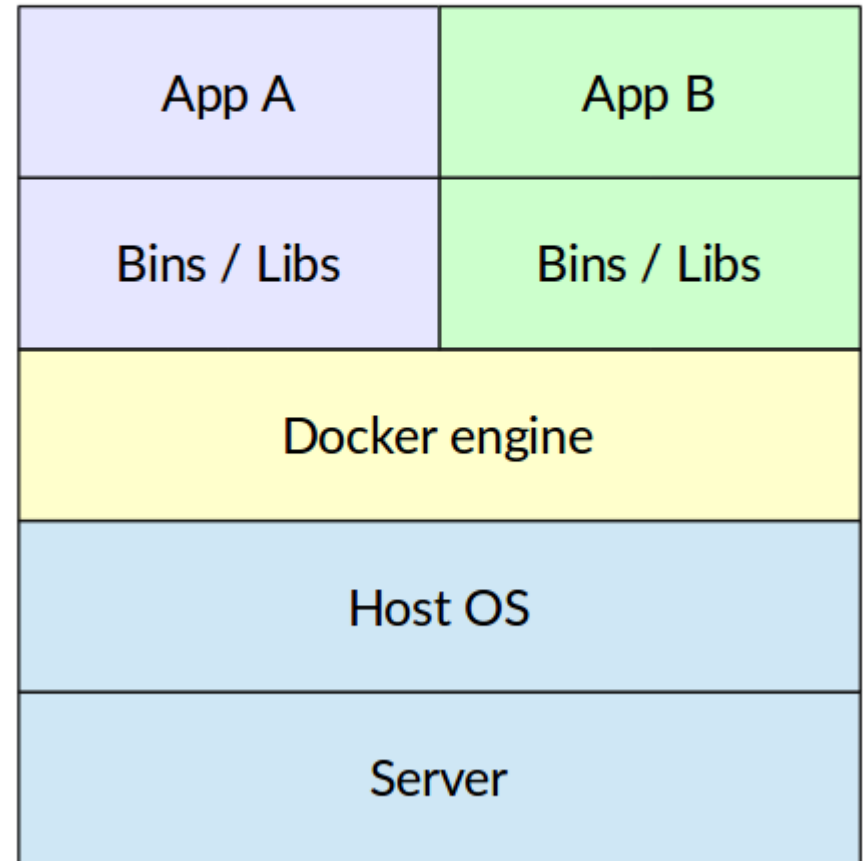
- A type of virtualization
- Differences from traditional VMs
 - Does not replicate entire OS, just bits needed for application
 - Run natively on host
- Key benefits:
 - More lightweight than VMs
 - Efficiency gains in storage, CPU
 - Portability

Lightweight

Virtualization



Docker



Benefits: Resources

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

Benefits: Decoupling

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

Benefits: Developer Workflow

- 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

Docker Platform

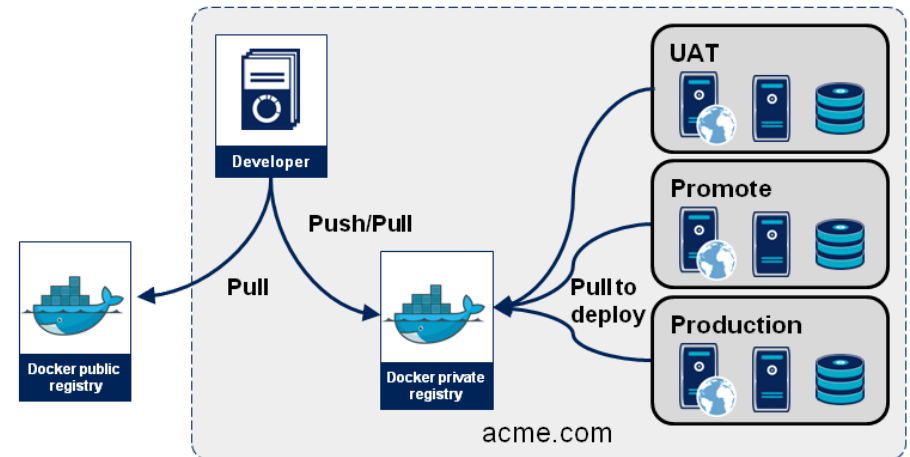
What is Docker?

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.



Portability

- Docker is supported on 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
```

- Version numbering scheme similar to Ubuntu versioning: YY.MM.#

In-line 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:
```

<code>--config string</code>	Location of client config files (default <code>"/U</code>
<code>-D, --debug</code>	Enable debug mode
<code>--help</code>	Print usage

Basic Client Usage

docker **command** *[options]* *[args]*

- Calling any command with **--help** displays documentation

Exercise: View documentation for docker run

```
$ docker run --help
```

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

Run a command **in** a **new** container

Options:

<code>--add-host list</code>	Add a custom host-to-IP mapping (host:ip)
<code>-a, --attach list</code>	Attach to STDIN, STDOUT or STDERR
<code>--blkio-weight uint16</code>	Block IO (relative weight), between 10 and 1000, or 0 to disable
<code>--blkio-weight-device list</code>	Block IO weight (relative device weight) (default [])
<code>.</code>	
<code>.</code>	

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

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

Run a Docker Container

`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

Run a Simple Container

```
docker run hello-world
```

- 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 **Alpine OS**, a very minimal Linux distribution.
- [command] argument is executed inside container
- The 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
```

Running an interactive container

- 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

docker ps

- List currently running containers

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

- Note the name assigned to the alpine container.
- By default docker assigns containers random names

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?

Removing containers

```
docker rm name|containerID
```

Exercise: Remove old *myalpine* container

```
docker rm myalpine
```

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

Exercise: Run a 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="YOURNAME" -p 8081:80 dockersamples/static-site
```


Run a website in a container

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

Stopping 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
```

- For the previous exercise, you'll need to be in another terminal

Exercise: Run 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
```

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](#)

Excercise: View container logs for *static-site* container

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

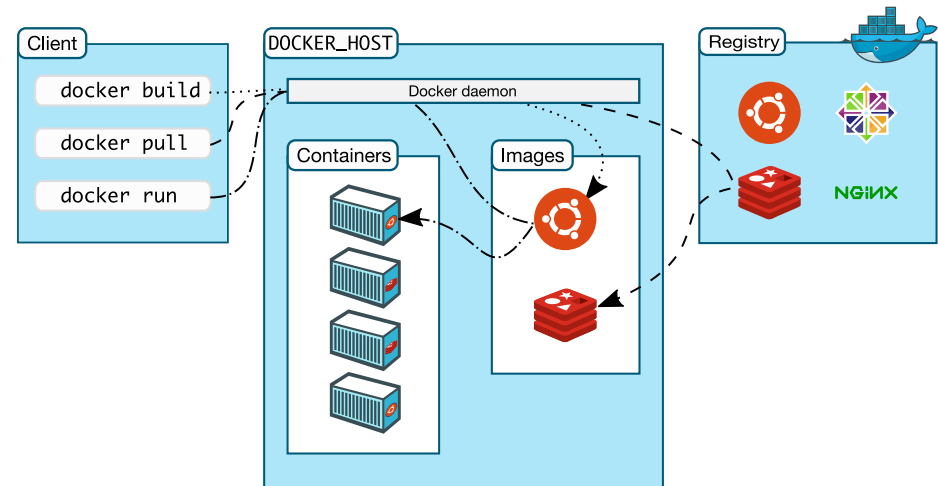
Exercise: Check process list in *static-site* container

List Local Images

```
docker image ls
```


Docker Behind the Scenes

- Docker application on your machine is a client-server application
 - Type commands to *docker client* on CLI
 - docker build
 - docker pull
 - docker run
 - 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



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



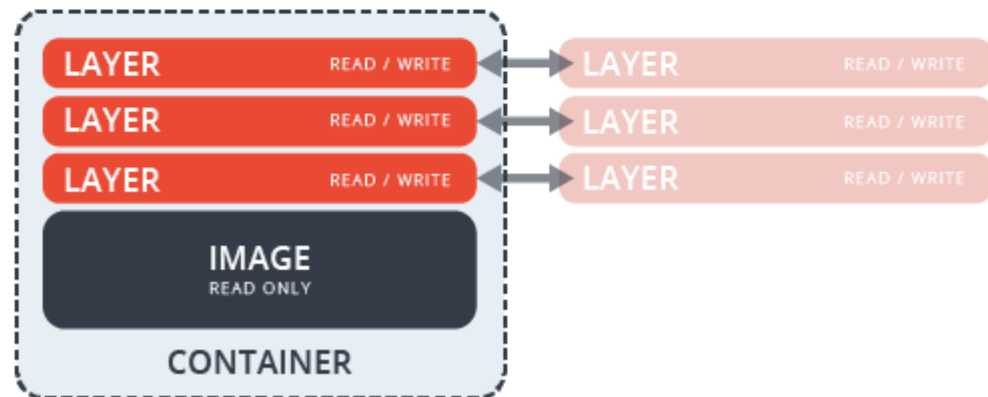
Docker Registry

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



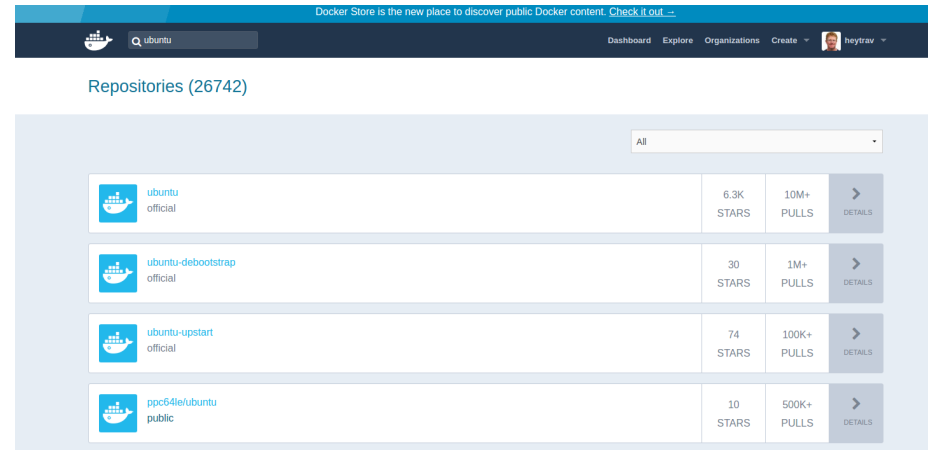
Docker Container

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**



The screenshot shows the Docker Store interface. At the top, there's a navigation bar with the Docker logo, a search bar containing 'ubuntu', and links for Dashboard, Explore, Organizations, Create, and a user profile. Below the navigation bar, the text 'Repositories (26742)' is displayed. A dropdown menu is set to 'All'. The main content area shows a table of repositories:

Repository	Stars	Pulls	Details
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*

Image Naming Semantics

- 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`
- The fully qualified image name identifies an image
 - `gitlab.catalyst.net.nz:4567/<group>/<project>:tag`
 - `quay.io/<username>/image-name:tag`

Images and Layering

- Images are a type of *layered* file system
- Each image is a type of archive file (eg. tar archive) containing
 - Additional archive files
 - Meta information
- Any child image built by adding layers on top of a base
- Each successive layer is set of differences to preceding layer

Images and Layering

Layer	Description
4	execute <code>myfile.sh</code>
3	make <code>myfile.sh</code> executable
2	copy <code>myfile.sh</code>
1	install libraries
0	Base Ubuntu OS

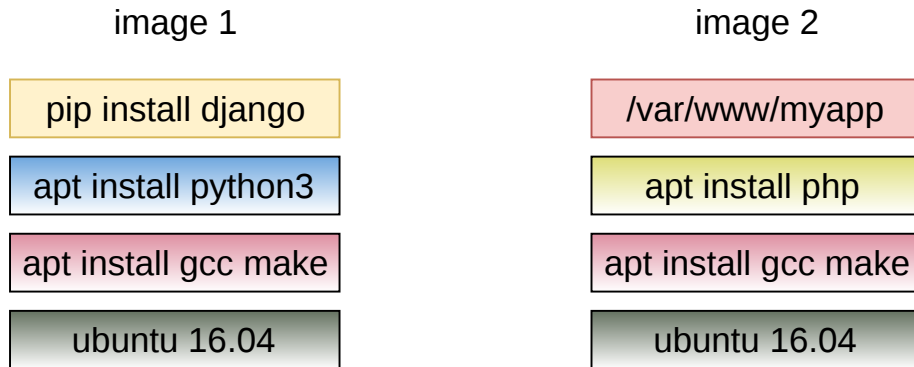
- A layer is an instruction that
 - changes the filesystem
 - tells Docker what to do when run

Sharing Image Layers

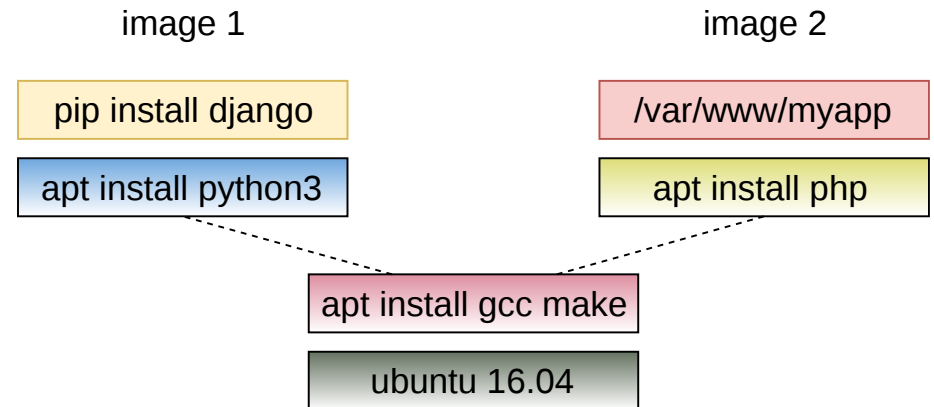
- Images will share any common layers
- Applies to
 - Images pulled from Docker
 - Images you build yourself

Sharing Image Layers

Two separate images



Reality: common layers shared



View Image Layers

`docker history <image>`

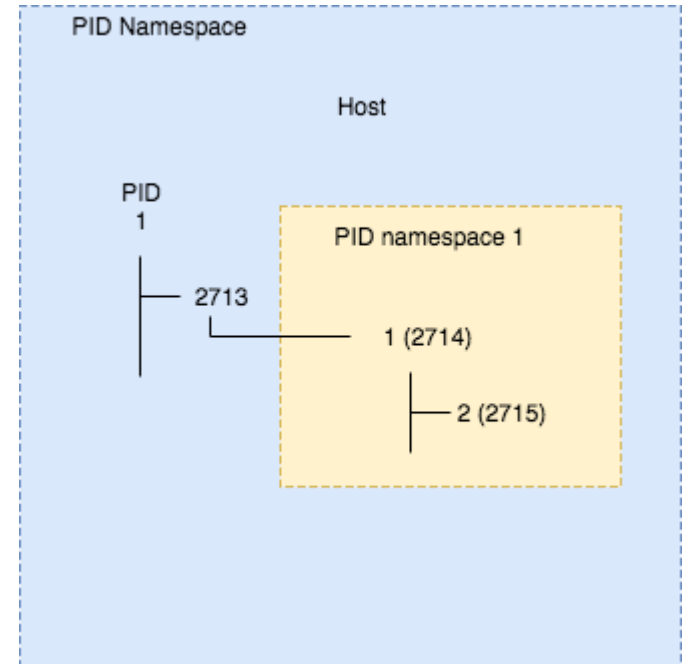
```
docker history heytrav/docker-introduction-slides
```

IMAGE	CREATED	CREATED BY	SIZE
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

Containers

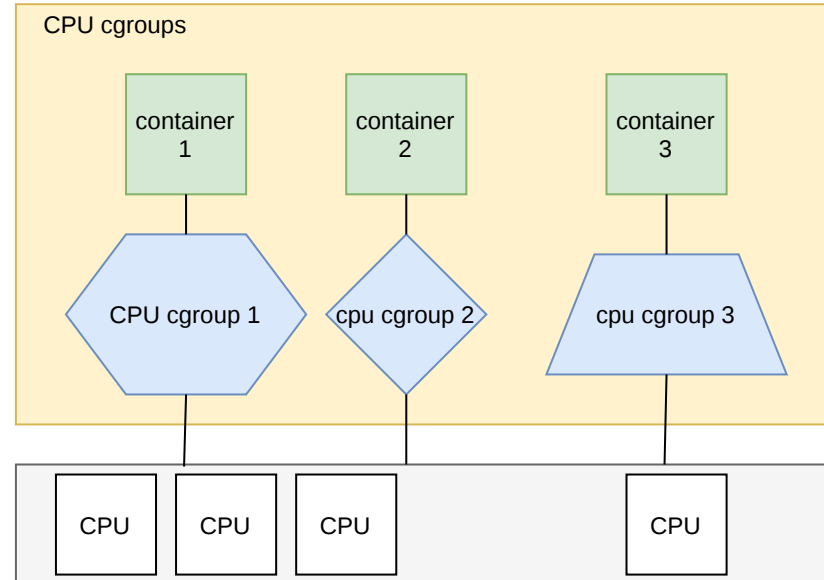
Namespaces

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



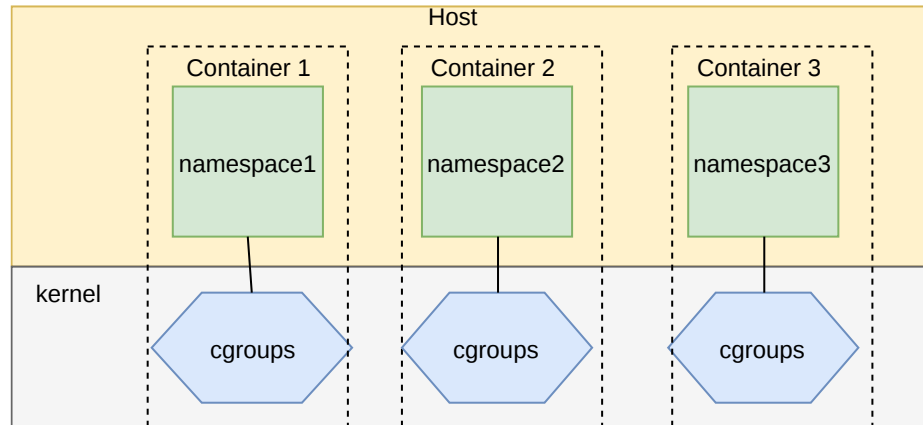
cgroups

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



Combining the Two

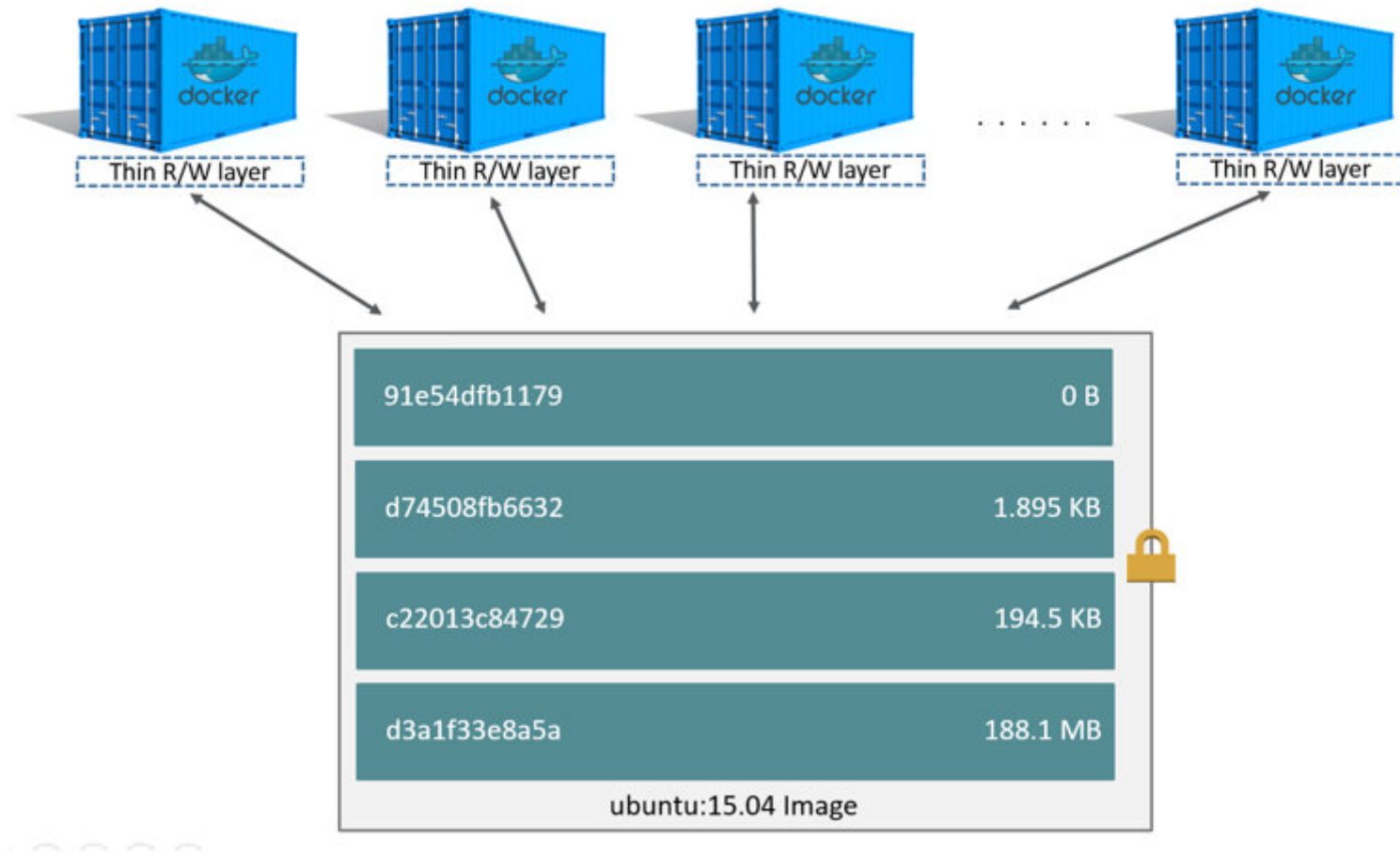
A running container represents a combination of layered file system, namespace and sets of cgroups



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

Container Layering



Create images, explore layers

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

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
```


Explore Image Layers

- 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

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
 - Another image you have created

RUN

RUN **command** **arg1** **arg2** ...

- Execute shell commands for building an 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 | /b
```

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
```

- 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"]
```

docker build

```
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

Exercise: Build an image using a Dockerfile

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

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

More Dockerfile Instructions

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 and 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

ENTRYPOINT & CMD

- Hypothetical application

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

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

```
docker run my-image
```

ENTRYPOINT & CMD

```
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

```
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

Creating Applications in Docker

- In this section we will:
 - Create a small web app based on Python Flask
 - Write a Dockerfile
 - Build an image
 - Run the image
 - Upload image to a Docker registry

Set up the the web page

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

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

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

```
docker build -t YOURNAME/myfirstapp .
```


Run the Container

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

...Now open **your test webapp**

Login to Docker Registry

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

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

`.dockerignore`

- A text file called `.dockerignore`
- 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

Sample .dockerignore File

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

Best Practices for Images

- 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

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
```

- 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

- 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

- 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 to use COPY

- 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

- 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

- 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 /root
ENTRYPOINT ['/root/myprogram']
```

Example: Multistage Build

```
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 revisited

- 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`

Example: Shell vs Exec

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

What happens when you want to stop container
topshell?

Example: Shell vs Exec

- 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**

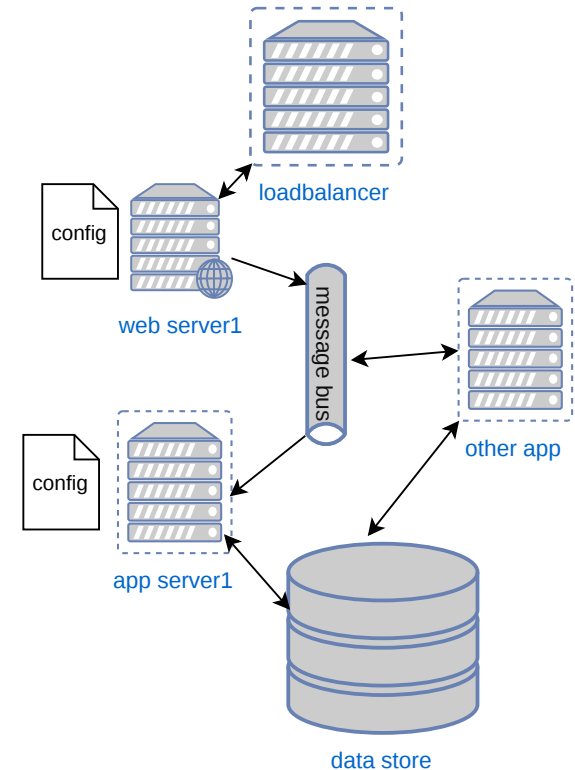
Summary

- 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**

Designing Containerised Applications

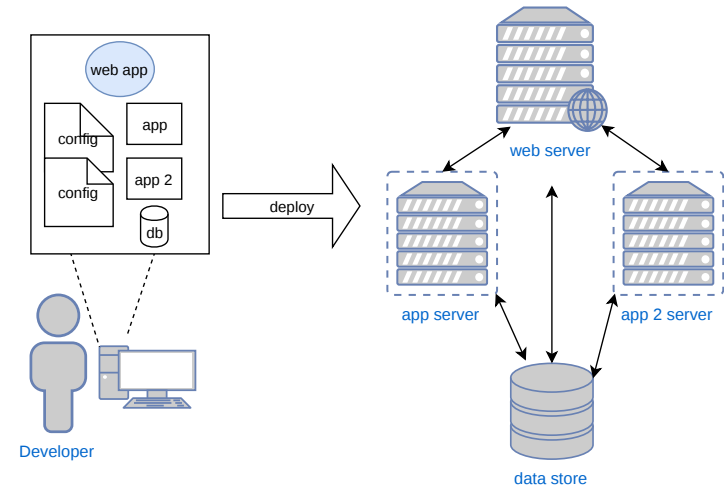
Developing Applications

- Applications can consist of many components
 - Web server (nginx, apache)
 - Database (sql, nosql)
 - Message Queue
 - Your application
- Typically spread across cluster of machines



Development vs Production

- Ideal scenario: Development environment identical to production
- In practice this is often difficult to achieve
 - Limited CPU of dev machines
 - Cost of machines
 - Compromise is to develop everything in single VM



Pitfalls of Single VM Development

- Single VM development creates blindspot
- Developers can make false assumptions about
 - Which config files on which machines
 - Which dependency libraries present on machines
- Difficult to scale individual services
- Applications components often tightly coupled
- Can lead to unpredictable behaviour when application is deployed to production

Misconception About Docker Containers

“ I'll just put my entire application into a Docker container and run it that way ”

Common mistake to try and treat Docker containers like traditional VMs

Designing Containerised Applications

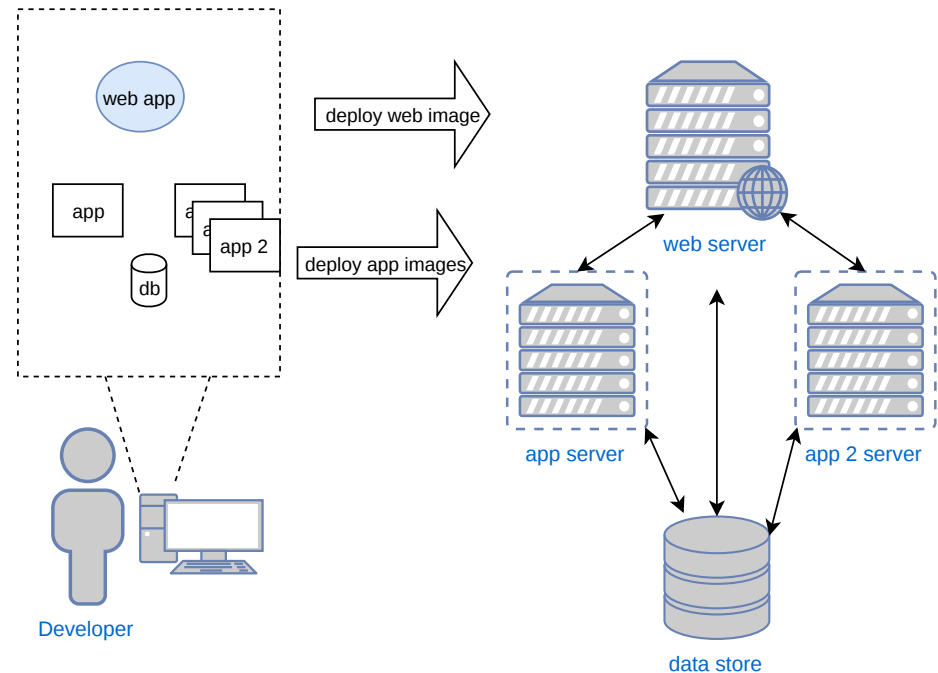
- Containerised application should *smallest executable unit*
 - Have a single application
 - Single executable runner
 - Single process per container
 - Single component of your application
- No stored *state*

Containerise Application Components

- Each component is self-contained

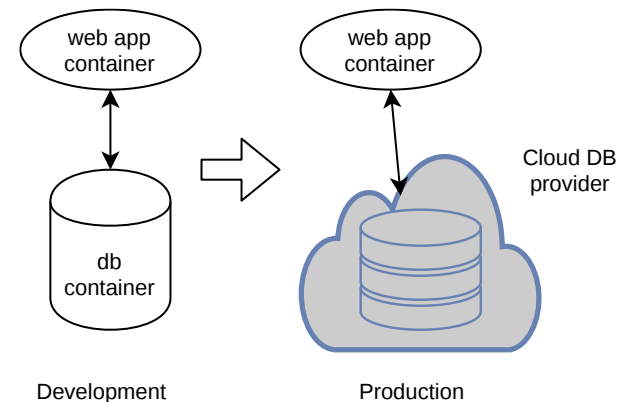
- dependencies
- configuration

- Better decoupling of components and dependencies
- Brings us closer to production environment



Designing Containerised Applications

- Containerised component(s) should be agnostic to other services/components
- Eg. application component may interact with
 - Containerised database in development
 - Cloud provider in production



Something still not right

*“ Your Docker thingy still isn't exactly
like production! ”*

*“ Our application(s) are not deployed as
Docker containers ”*

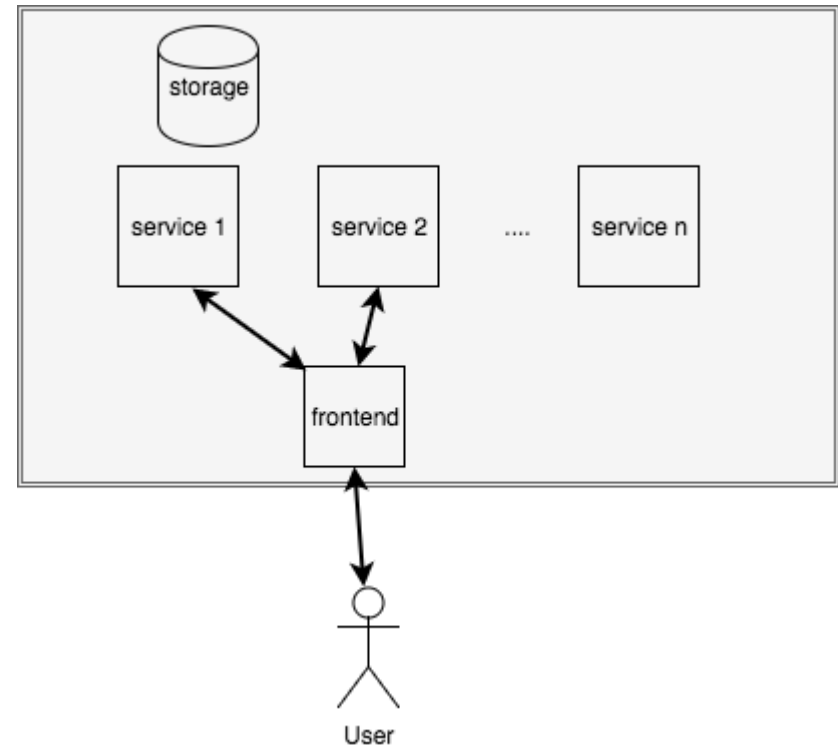
Docker Developer Workflow

- In the following sections we'll explore ways of making development environment similar/identical to production
- Development
 - docker-compose as a tool for managing complex microservice applications in development
- Production
 - Orchestration platforms for deploying container workloads in production environments
- Connecting the two with CI/CD workflows

Docker and Development

Docker and Microservices

- Containerised applications ideal for microservices
- Components ideally self contained and modular
 - deployed independently
 - scaled independently



Microservices in Docker

```
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
```

- Let's build a docker image for our app

Create Our App

- Fire up your favourite editor and create a Dockerfile

```
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 .
```

First Pass: Creating 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
```

- Try `docker ps` to see that *web* and *redis* containers are running
- Once you start the web container go to **web page** to see counter

Disadvantages of Command Line 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

- 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](#)

```
---
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
```

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
```

Interacting with Docker Compose

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

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

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

Exercise: Convert app to docker-compose

- In the 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

Stopping docker-compose

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

```
$ docker-compose stop
```

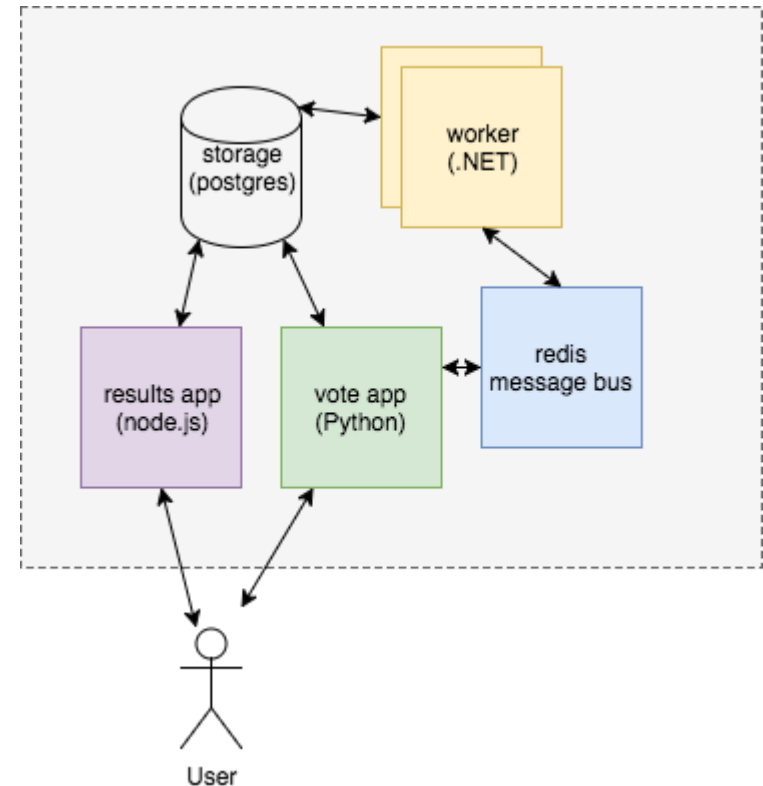
Summary

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

Docker Workflow Example

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



Start Application

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

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

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

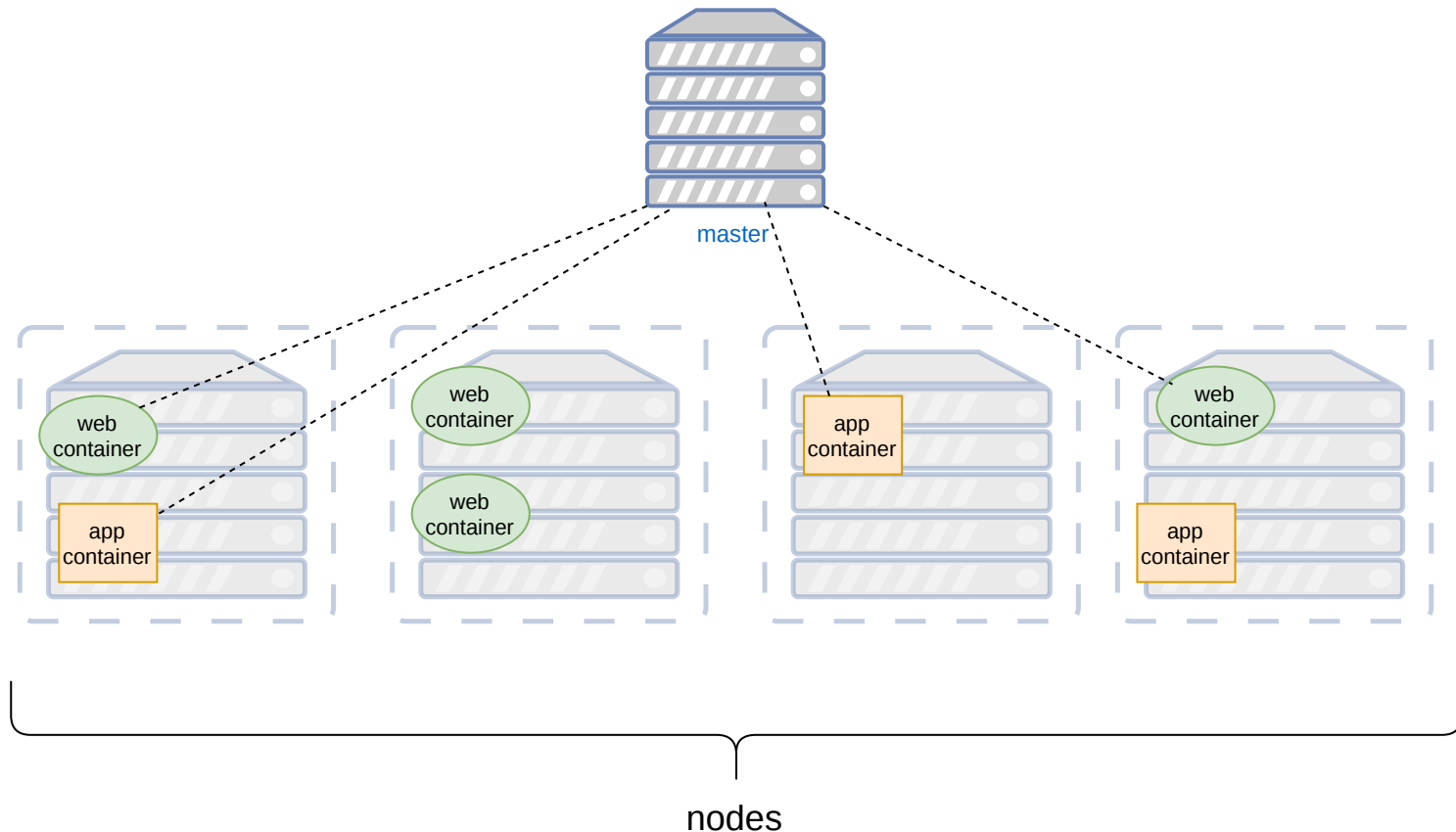
Deploying to Production

Container Orchestration

- Primary means of deploying containerised applications to production
- Provides tools for managing containers across a cluster
 - networking
 - scaling
 - monitoring
- Ideal for deploying containerised applications in production

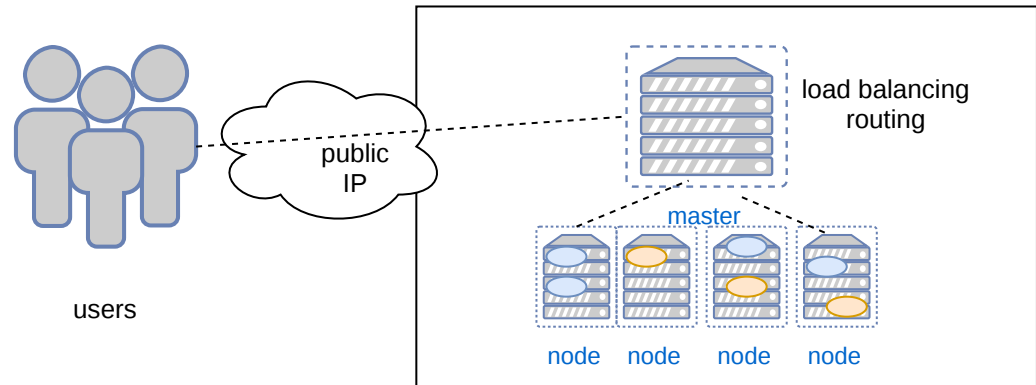
Server Architecture

- Machine designated the *master*
- Several machines designated *nodes*



Container Orchestration: User perspective

- Orchestration framework handles
 - Routing requests to containers
 - Load balancing between different containers
- From user perspective as if interacting with single application



Masters and Nodes

- The *master* is responsible for
 - scheduling containers to run across all *nodes*
 - managing the network interaction between nodes
 - monitoring container health
 - periodically kill/respawn containers
- The *nodes* or *workers*
 - Just run the containers

Container Lifecycle

- Containers are ephemeral
- The job of the *master* is to make sure containers are healthy
- It will periodically kill and respawn a container
- This is very similar to *phoenix* principle
 - Outdated or unhealthy containers *burned*
 - Fresh containers spawned in their place



Orchestration Platforms

- Apache Mesos
- AWS ECS
- Docker Swarm
 - Integrated into Docker since 17.03
- Kubernetes
 - Descends from *Borg*, (Google)
 - Joint project from Google, CoreOS, OpenShift
 - Can use other container platforms than Docker (eg. rkt)

Kubernetes

Kubernetes Facts

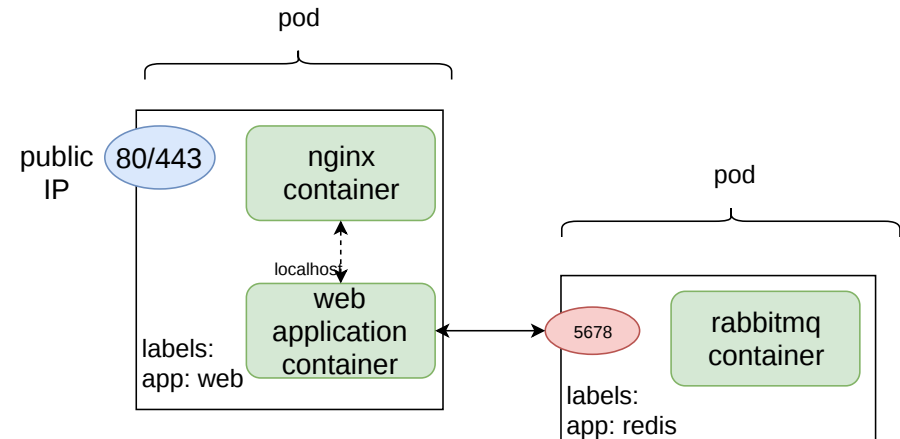
- Greek word for *helmsman* or *pilot*
- Also origin of words like *cybernetics* and *government*
- Inspired by *Borg*, Google's internal scheduling tool
- Play on *Borg cube*
- **Source**

Kubernetes Concepts

- Host types
 - master
 - performs *scheduling*
 - monitoring/healthchecks
 - node (formerly *minion*)
 - runs containers

Pods

- A *pod* is the unit of work
 - Consist of ≥ 1 containers
 - Always *scheduled* together
 - Have same IP
 - Communication via localhost

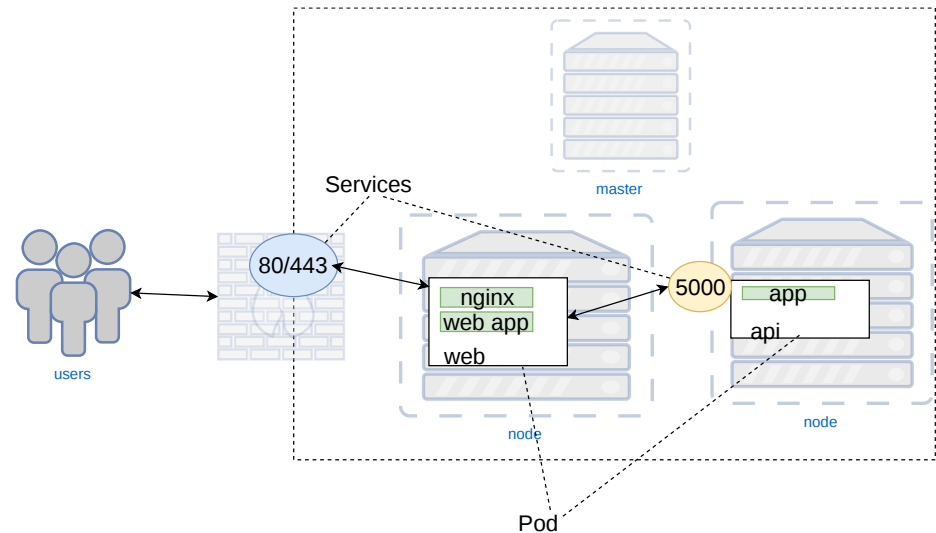


Deployments

- Provides declarative updates for Pods
- Describe *desired state* of an object and controller changes state at a controlled rate
- Functions
 - Create pods
 - Declare new state of pods
 - Scale deployment
 - Rollback to an earlier deployment revision

Services

- Exposes IP of Pod to
 - Other Pods
 - External ports (i.e. web, API ingress)



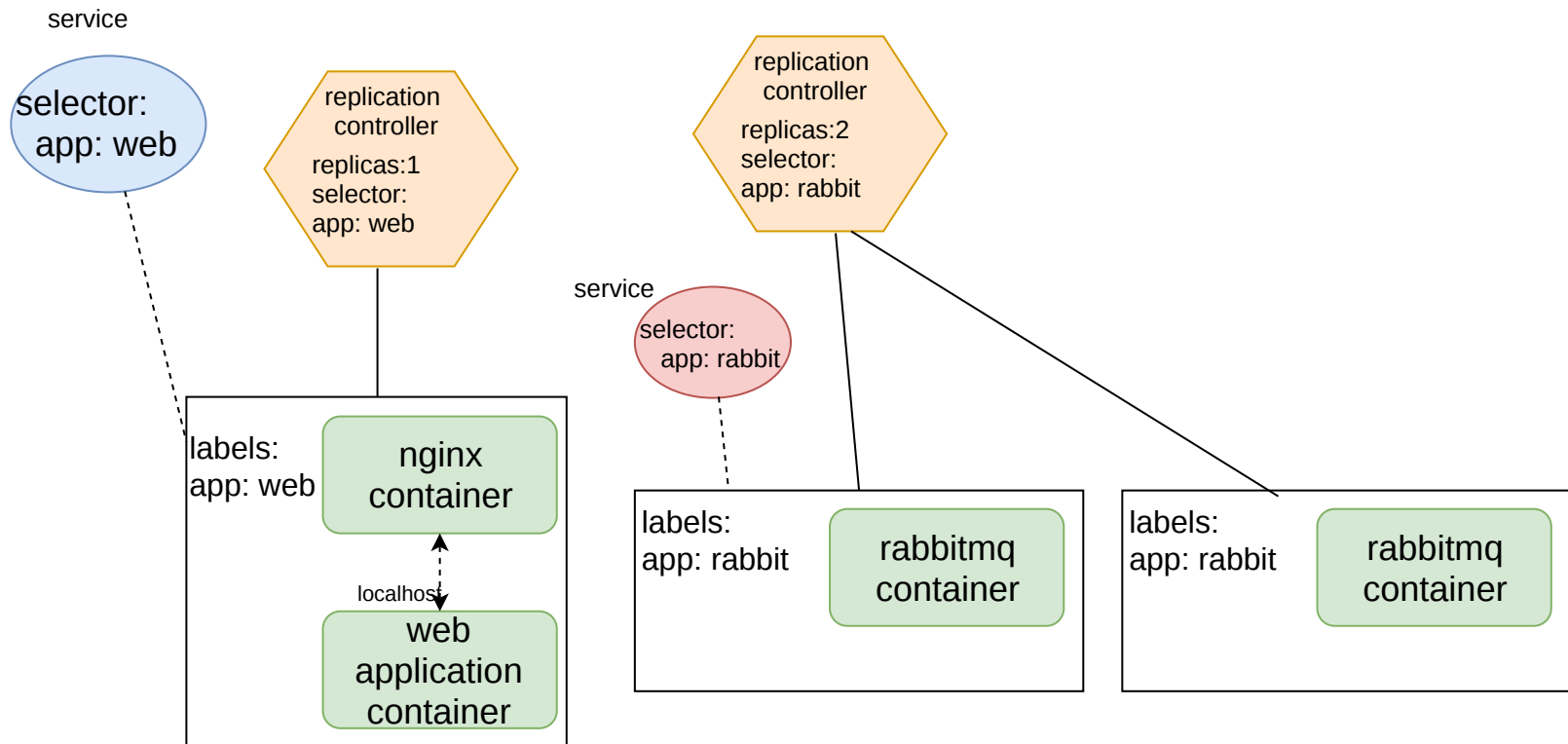
Labels & Selectors

- Label is a key: value pair used to group objects
 - replication controllers for scheduling pods
 - services
- Label Selectors
 - Select objects base on labels
 - Semantics:
 - `role = webserver`
 - `app != foo,`
 - `role in (webserver, backend)`

Namespaces

- Virtual cluster
- Isolate set of containers on same physical cluster

Kubernetes Labels & Deployments



Defining a Service

- Service spec defines
 - Type
 - NodePort | ClusterIP
 - Ports & protocol
 - Map to Replication Controller (Pod)

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  type: ClusterIP
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    app: redis
```

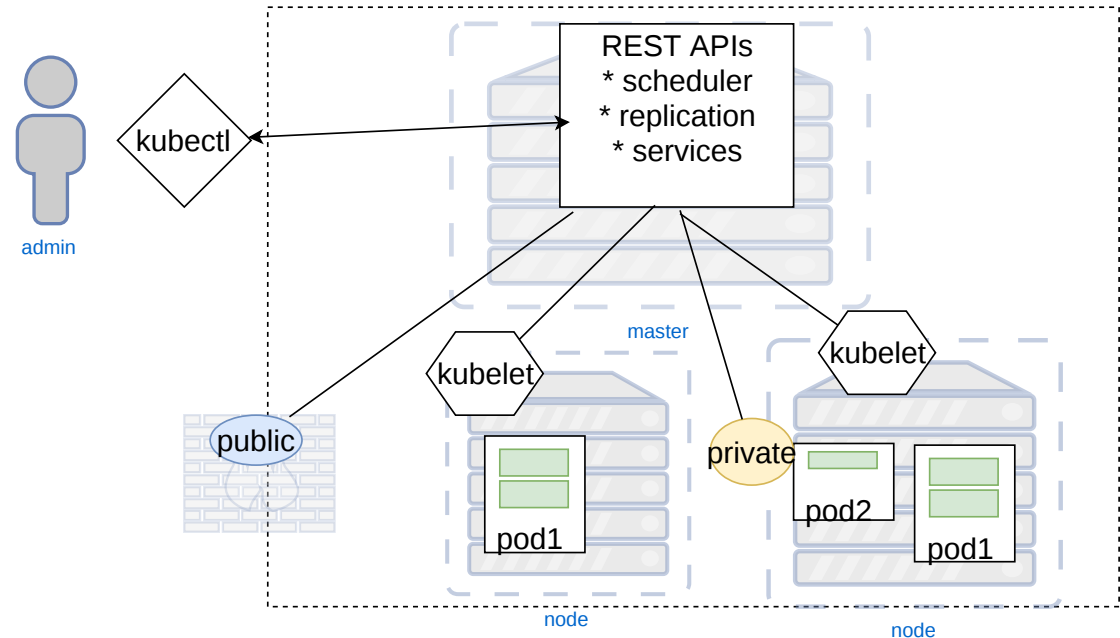
Defining a Deployment

- Specification deployment file
- Attributes define
 - How many instances to run at start
 - Label selectors
 - Images/containers in pod
 - Volumes mounted in pod

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - image: redis:alpine
          name: redis
          volumeMounts:
            - mountPath: /data
              name: redis-data
      volumes:
        - name: redis-data
          emptyDir: {}
```

Controlling Kubernetes

- Control plane of Kubernetes is a REST API
- Admin cluster using `kubectl` command line client



Demo: Set up Voting Application in Kubernetes

Setup

- Steps needed:
 - Create host machines in the cloud
 - Set up networking
 - Install Kubernetes dependencies
 - kubectl
 - kubeadm
 - kubelet
 - Join nodes to master
 - Deploy Kubernetes spec files

Create Hosts File

```
cd docker-introduction/ansible  
ansible-playbook local-setup.yml -e prefix=<username>
```


Create Kubernetes Cluster

```
ansible-playbook -K -i cloud-hosts \
  create-cluster-hosts.yml kubeadm-install.yml -e prefix=<userna
```

Setting up the Voting Application

- Have a look in the `example-voting-app/k8s-specifications`

Remotely Controlling Kubernetes

- Start kubectl proxy locally

```
kubectl --kubeconfig ~/k8s-admin.conf proxy  
Starting to serve on 127.0.0.1:8001
```

- Put this terminal aside and open a new one

Verify Kubernetes Cluster

```
kubectl --server=127.0.0.1:8001 get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
trainingpc-master	Ready	master	26m	v1.10.2
trainingpc-worker1	Ready	<none>	25m	v1.10.2
trainingpc-worker2	Ready	<none>	25m	v1.10.2

Create Namespace

- Create a namespace for our application

```
kubectl --server=127.0.0.1:8001 create namespace vote  
namespace "vote" created
```

Watch cluster

- In another terminal, run the following

```
watch -t -n1 'echo Vote Pods \  
  && kubectl --server=127.0.0.1:8001 get pods -n vote -o wide \  
  && echo && echo vote Services \  
  && kubectl --server=127.0.0.1:8001 get svc -n vote \  
  && echo && echo vote Deployments \  
  && kubectl --server=127.0.0.1:8001 get deployments -n vote \  
  && echo && echo Nodes \  
  && kubectl --server=127.0.0.1:8001 get nodes -o wide'
```

Load Specification Files

- The `apply` command loads a specification into kubernetes

```
kubectl apply <file>
```

- The entire vote app is specified in yaml files

```
cd ~/example-voting-app/k8s-specifications
for i in `ls *.yaml`; \
do kubectl --server=127.0.0.1:8001 apply -n vote -f $i; done
```

- This tells kubernetes to begin setting up containers
 - creates network endpoints
 - assigns Pods to replication controller
- When you run this, go back to the *watcher* terminal

View Website

- Once all containers are running you can visit your website
- You first need to find a couple ports:

```
vote Services
NAME      TYPE      CLUSTER-IP      ...    PORT(S)      AGE
db         ClusterIP  10.108.228.228   ...    5432/TCP      3h
redis      ClusterIP  10.107.101.100   ...    6379/TCP      3h
result     NodePort   10.107.43.36     ...    5001:31001/TCP 3h
vote       NodePort   10.104.244.69    ...    5000:31000/TCP 3h
```

- Navigate to the **voting app**. You may need to change the port

Scaling

- Orchestration platforms make it easy to scale your app up/down
 - Simply increase or decrease the number of containers
- Let's increase the number of vote containers

```
kubectl --server=127.0.0.1:8001 -n vote scale deployment vote --replicas=9
```

- Play with the scaled number; keep an eye on *watcher* terminal

Updating Our Application

- Update the `vote` application with your image

```
kubectl --server=127.0.0.1:8001 \
  -n vote set image deployment/vote \
  vote=YOURNAME/vote:v1
```

- Watch the *watcher* terminal
- Refresh the site several times while update is running

Kubernetes Dashboard

- Kubernetes provides a dashboard for monitoring purposes

```
kubectl --server=127.0.0.1:8001 -n kube-system apply -f \
https://raw.githubusercontent.com/kubernetes/dashboard/master
```

- Once you've activated it, go to the **dashboard page**

Clean up

```
ansible-playbook ansible/remove-cluster-hosts.yml -K -e prefix=<u
```

Deploying a Swarm Application

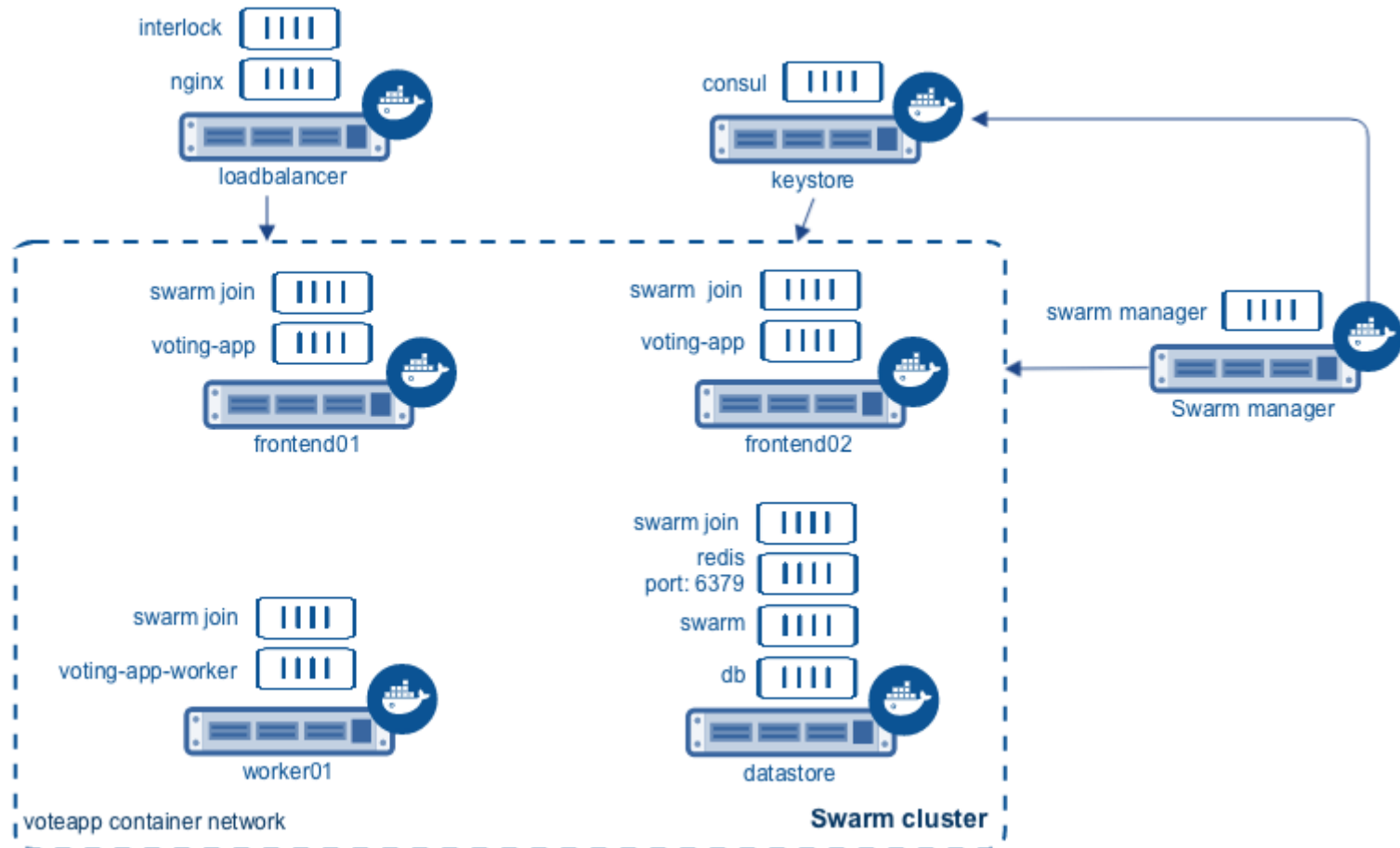
Docker Swarm

- 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



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
```

- Copy the `docker swarm join ...` command from the output

Join Worker Nodes

- Paste the command from the manager node into terminal on each worker node
- Repeat this for worker2

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

Check Nodes

```
docker node ls
```

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:
      replicas: 6
    update_config:
      delay: 5s
      parallelism: 1
```

Deploying the Voting App

- Upload `docker-stack.yml` 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
```

- 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

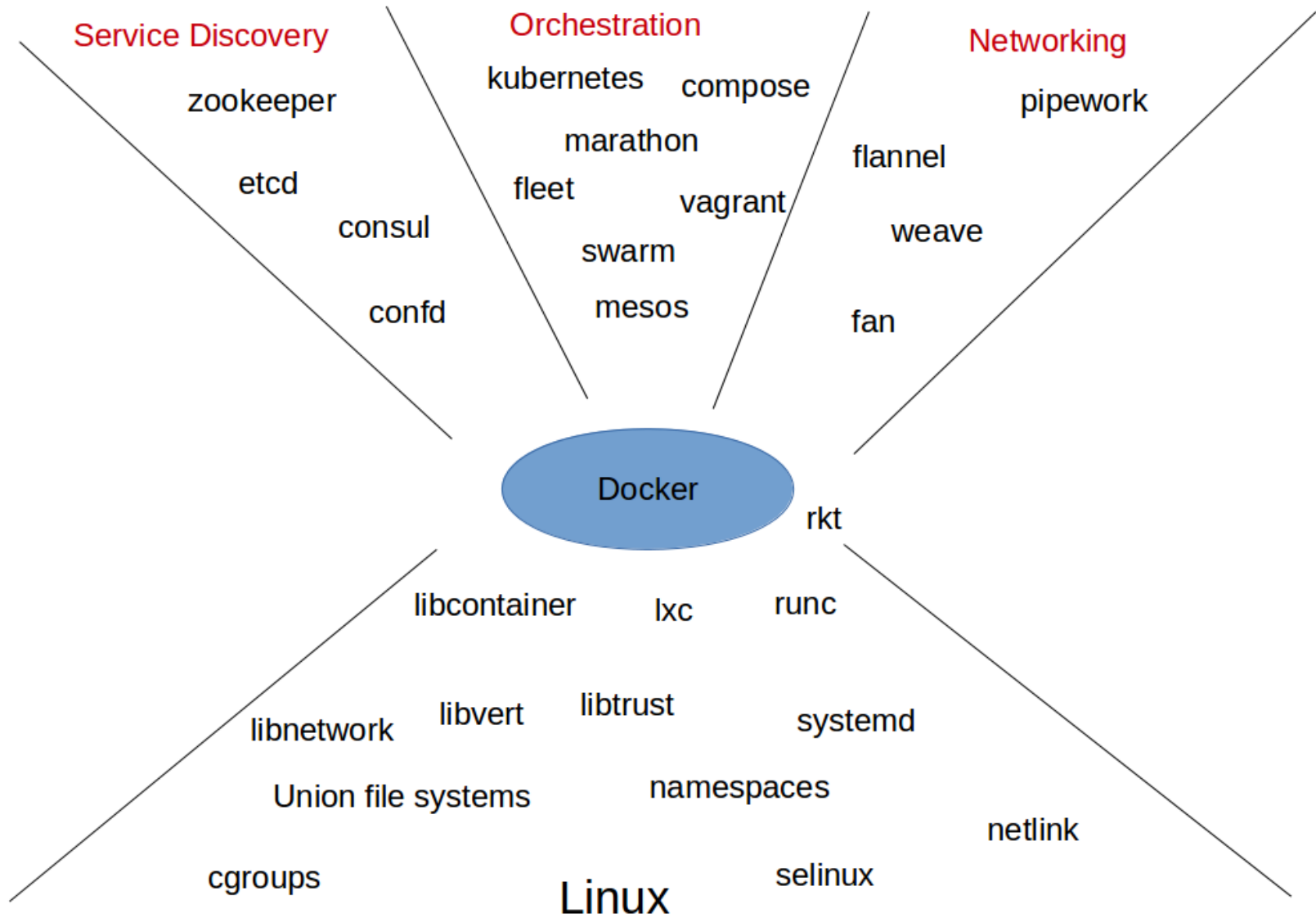
- Created a cluster with a cloud provider using ansible
 - 1 manager node
 - 2 worker nodes
- Deployed microservice for voting app in Docker Swarm
- Scaled service from 2 to 3 services
- Rolling-Updated image

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