

Containerisation with Docker & Kubernetes

Hello !



- I'm Dan !
- Senior Field Engineer at Heptio
VMware
- Ex:
 - Heptio
 - Docker
 - Hewlett-Packard Enterprise
 - SkyBet
 - European Space Agency
 - ...
- Still a maintainer and contributor to various Docker and Moby projects (mainly GO, but I sometimes write C (if need be))
- @thebsdbox – for random nonsense

Agenda



- Where did Docker come from?
- Docker Under the Hood
- Using Docker
- Questions?

Where did Docker come from?



In the beginning ...



Where did Docker come from?

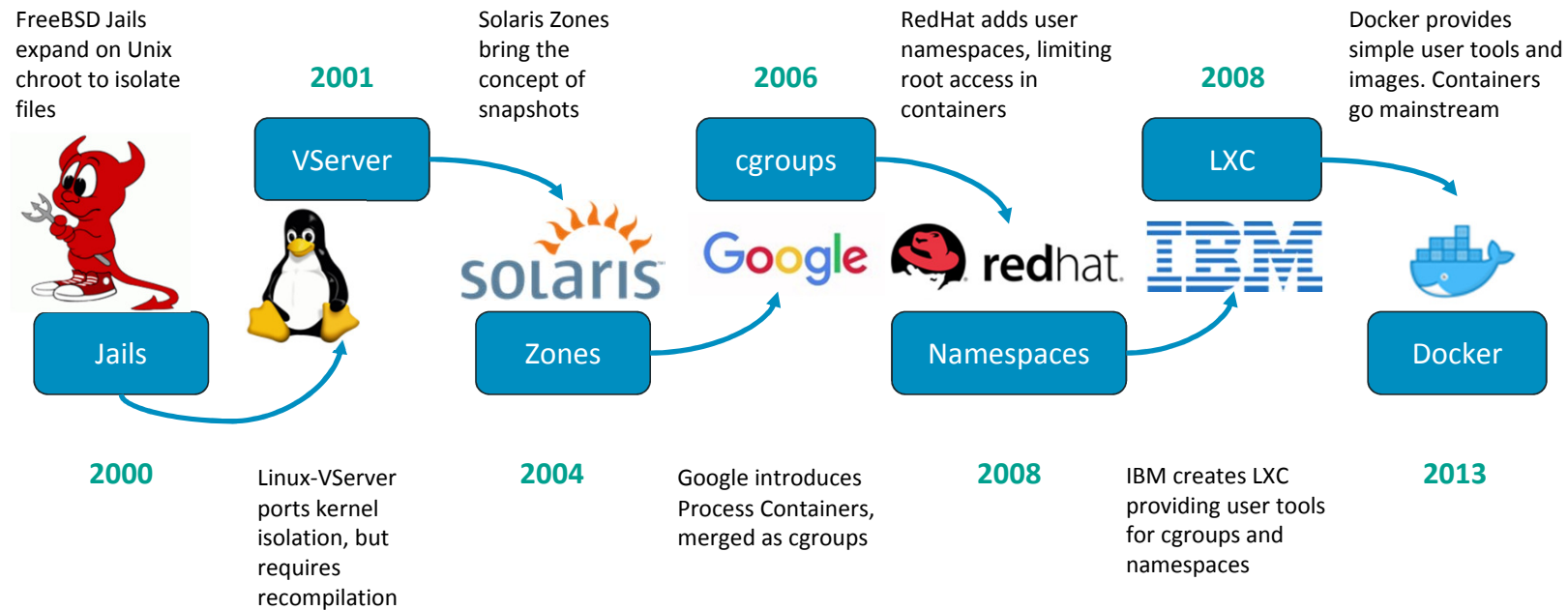


Things broke a lot ...

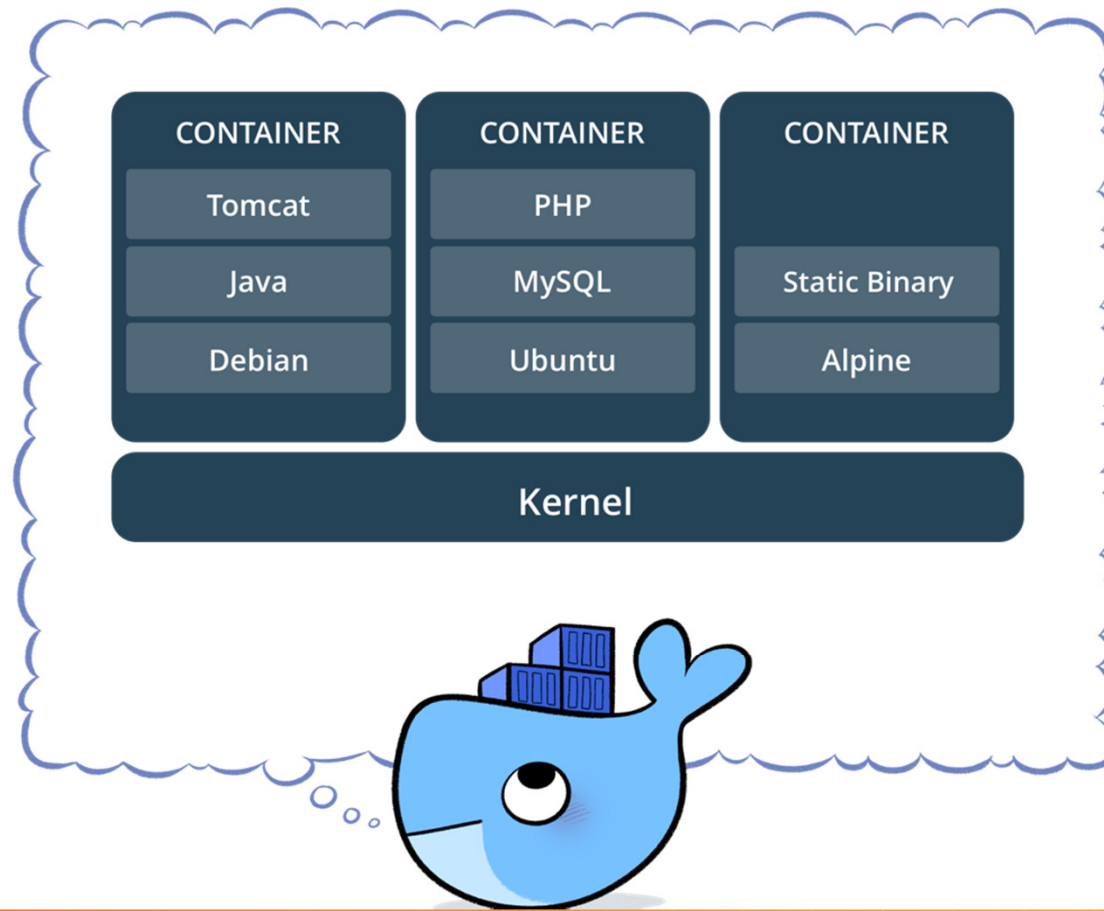
A screenshot of a PuTTY terminal window titled '192.168.137.101 - PuTTY'. The terminal displays the output of a 'yum update' command. It shows several packages being updated, including 'libdaemon', 'dbus-glib', 'java-1.4.2-gcj-compat', and 'libxml2-python'. However, the process fails with an error: 'Error: Missing Dependency: /usr/bin/rebuild-security-providers is needed by package java-1.4.2-gcj-compat-1.4.2.0-40jpp.115.i386 (base)'. The terminal also suggests using '--skip-broken' or running 'package-cleanup --problems' and 'package-cleanup --dupes'. The prompt 'console-3.2#' is visible at the bottom, followed by the command 'yum install yum-utils' which is partially entered.

```
192.168.137.101 - PuTTY
--> Processing Dependency: /usr/bin/rebuild-security-providers for package: java
--> Package libdaemon.i386 0:0.10-5.el5 set to be updated
--> Running transaction check
--> Package dbus-glib.i386 0:0.73-8.el5 set to be updated
--> Package java-1.4.2-gcj-compat.i386 0:1.4.2.0-40jpp.115 set to be updated
--> Processing Dependency: /usr/bin/rebuild-security-providers for package: java
--> Processing Dependency: /usr/bin/rebuild-security-providers for package: java
--> Package libxml2-python.i386 0:2.6.26-2.1.2.8 set to be updated
--> Processing Dependency: /usr/bin/rebuild-security-providers for package: java
--> Processing Dependency: /usr/bin/rebuild-security-providers for package: java
--> Finished Dependency Resolution
java-1.4.2-gcj-compat-1.4.2.0-40jpp.115.i386 from base has depsolving problems
--> Missing Dependency: /usr/bin/rebuild-security-providers is needed by packa
86 (base)
java-1.4.2-gcj-compat-1.4.2.0-40jpp.115.i386 from base has depsolving problems
--> Missing Dependency: /usr/bin/rebuild-security-providers is needed by packa
86 (base)
Error: Missing Dependency: /usr/bin/rebuild-security-providers is needed by pack
386 (base)
You could try using --skip-broken to work around the problem
You could try running: package-cleanup --problems
                        package-cleanup --dupes
                        rpm -Va --nofiles --nodigest
console-3.2# yum install yum-utils
```

Where did Docker come from?



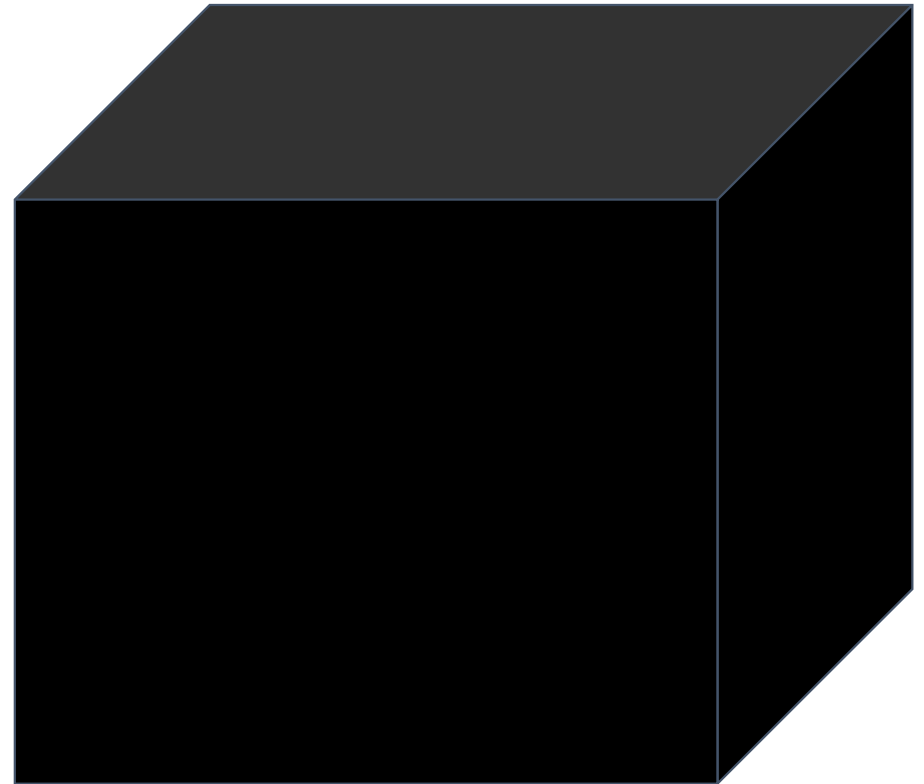
Docker under the hood



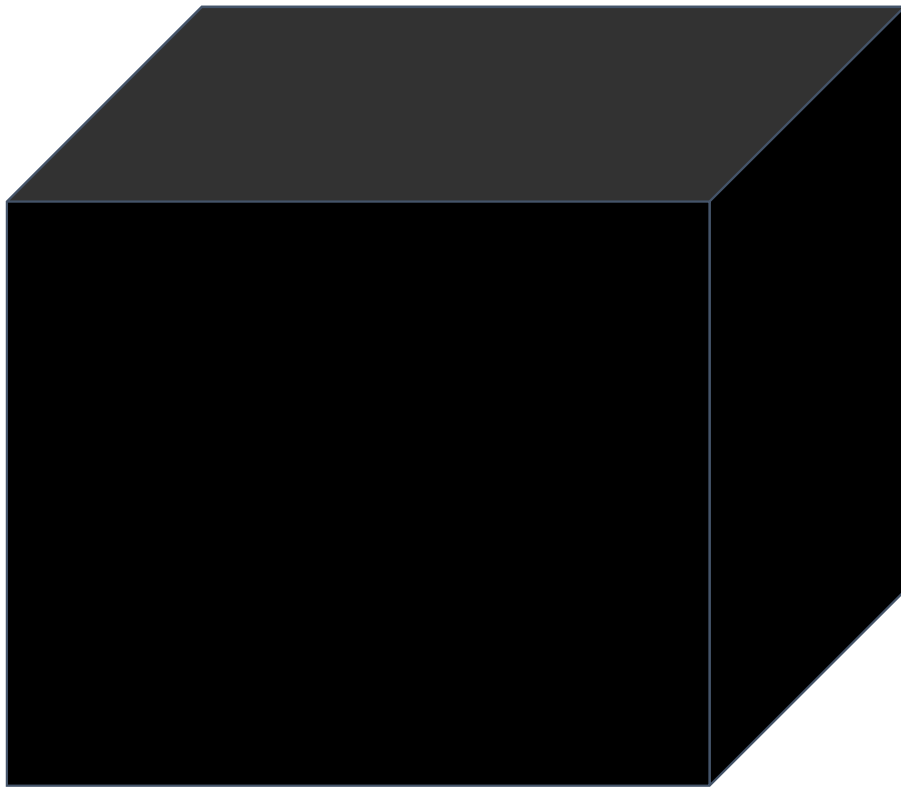
Docker Installation



- Linux installation(s)
 - `apt-get install -y docker`
 - `yum install -y docker`
- Docker for Mac
- Docker for Windows
- Docker on Windows 10 / 2016 / 2019



Docker Components



Docker CLI

dockerd

containerd

runc

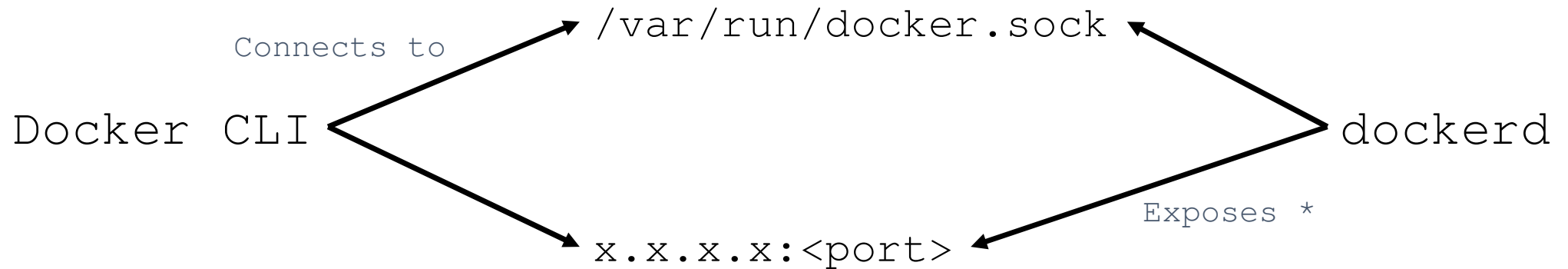
Docker Command Line Interface

- `docker build` build docker image from Dockerfile
- `docker run` run docker image
- `docker logs` show log data for a running or stopped container
- `docker ps` list running containers, `-a` includes stopped containers
- `docker images` list all images on the local volume
- `docker rm` remove a container | `docker rmi` : remove an image
- `docker tag` name a docker image
- `docker login` login to registry
- `docker push/pull` push or pull volumes to/from Docker Registries
- `docker inspect` return info/metadata about a Docker object

Docker Command Line Interface

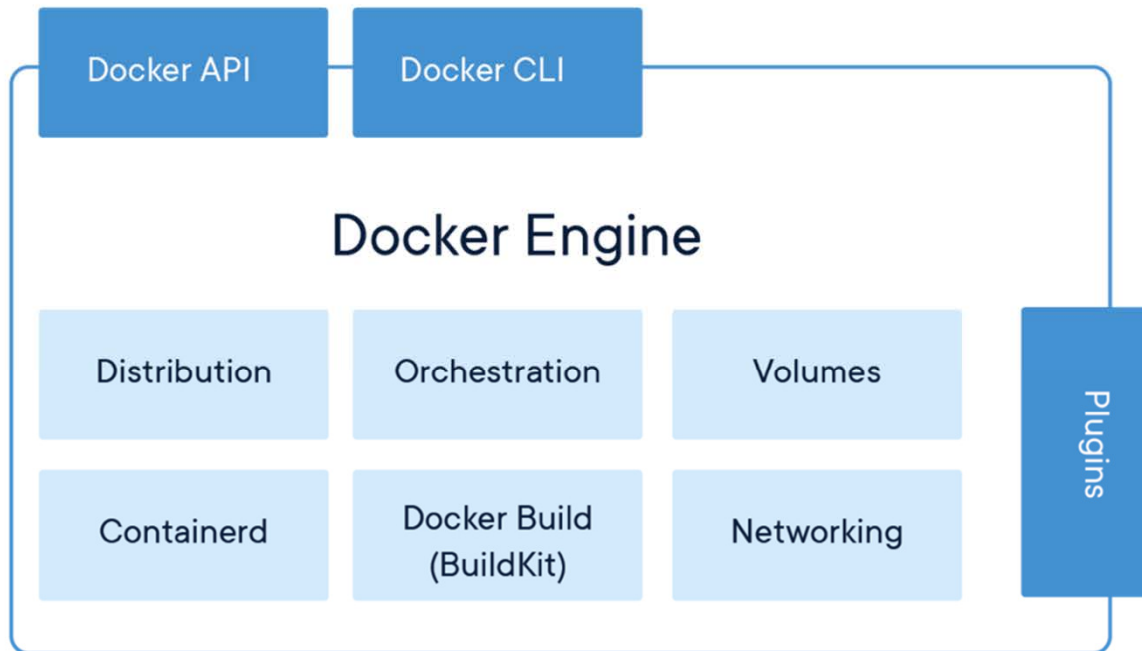


```
[dan@docker ~]$ docker run
```



* Configured in `/etc/docker/json`

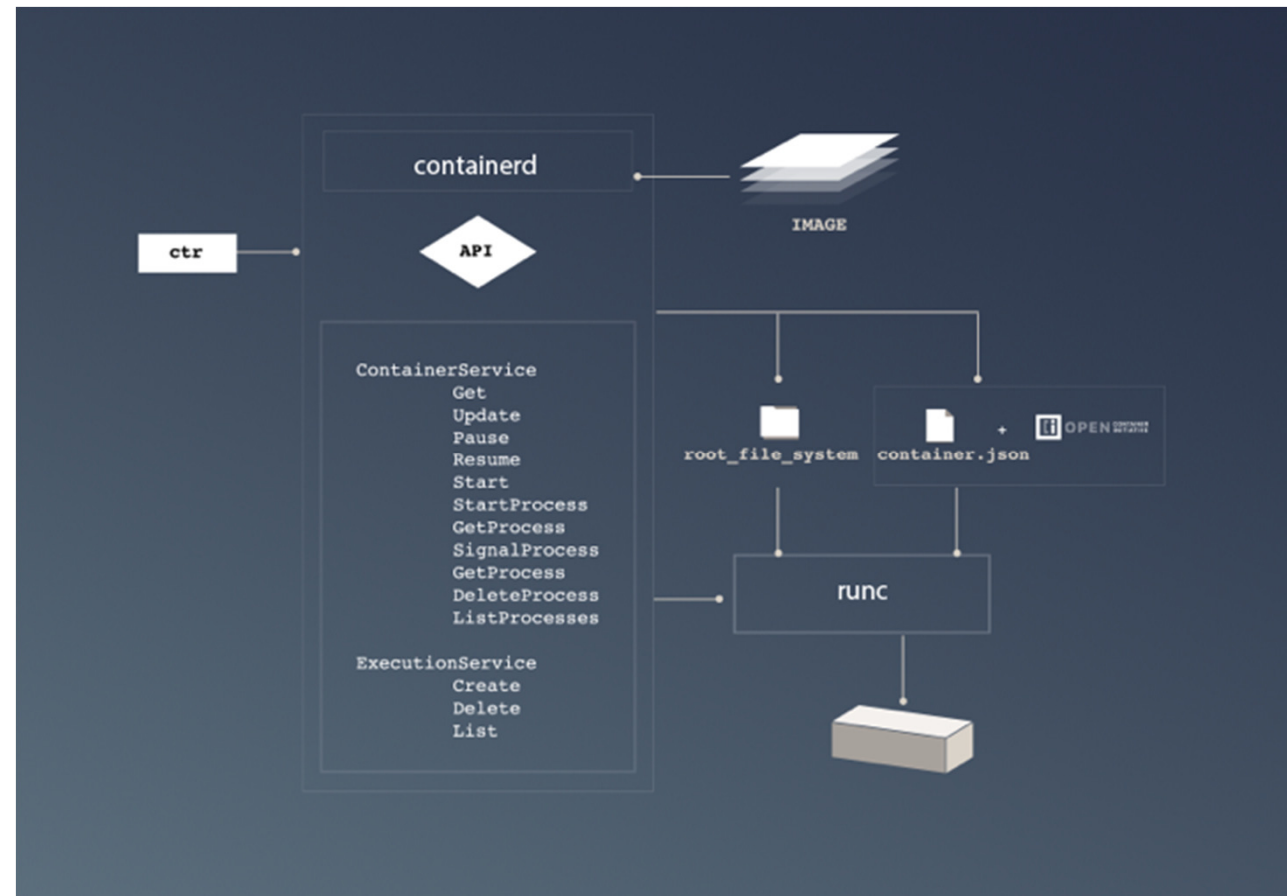
Docker Daemon



- Provides the "standard" interaction with a container platform
 - Image download from registries
 - Plugin features to extend the container platform for other vendors
 - In-built orchestration for multiple docker Engines
 - Container build function

containerd

- Manages the full container lifecycle:
 - Container execution
 - Image transfer/storage
 - Presentation of images to the OCI runtime
 - Networking
 - Container supervision



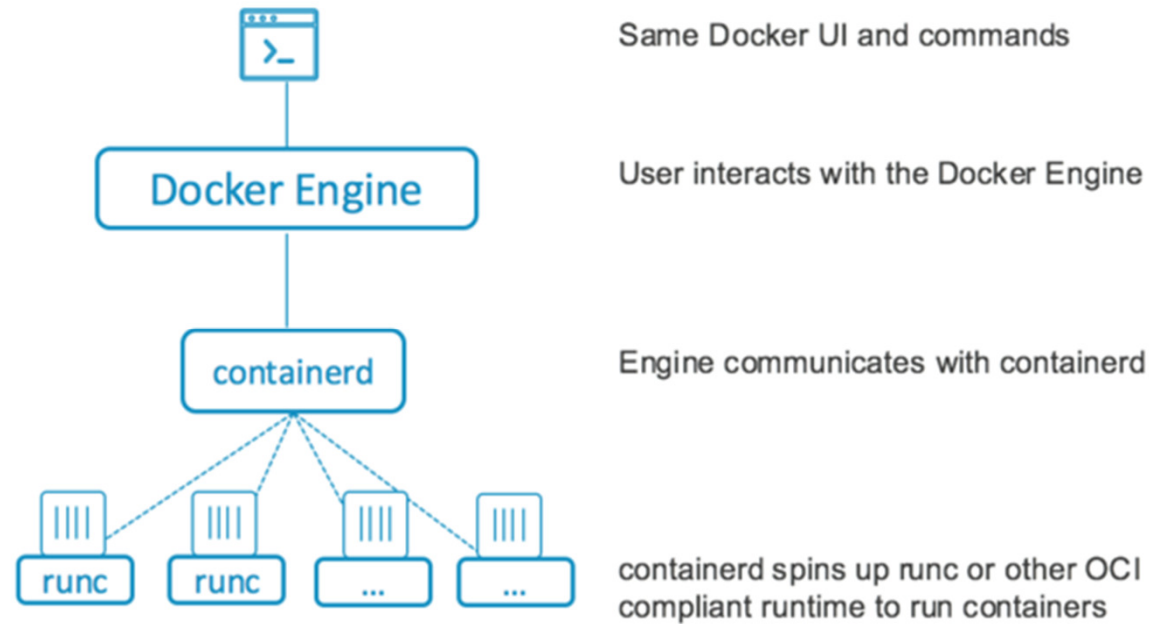
runc (or any OCI specific runtime)



- Lowest level component
- Implements an OCI compatible interface
- Implements the namespace isolation that “defines” a container
- Handles the starting, monitoring and stopping of a container



All tied together



Docker Image



```
/
/bin
/boot
/dev
/etc
/home
/lib64
/mnt
/opt
/var
/usr
```

aabbccdde

```
/
/bin
/boot
/dev
/etc
/home
/lib64
/mnt
/opt
/var
/usr
```

Manifest.json

Docker Image



```
/
/bin
/boot
/dev
/etc
/home
/lib64
/mnt
/opt
/var
/usr
/usr/myapp
```

aabbccdee

aabbccdee

```
/
/bin
/boot
/dev
/etc
/home
/lib64
/mnt
/opt
/var
/usr
/usr/myapp
```

Manifest.json

Docker Image



/	/	/
/bin	/usr/myapp	/etc/ssl.keys
/boot		
/dev		
/etc		
/home		
/lib64		
/mnt		
/opt		
/var		
/usr		

aabbccdde

aabbccdde

aabbccdde

/
/bin
/boot
/dev
/etc
/etc/ssl.keys
/home
/lib64
/mnt
/opt
/var
/usr
/usr/myapp

Manifest.json

Docker Image

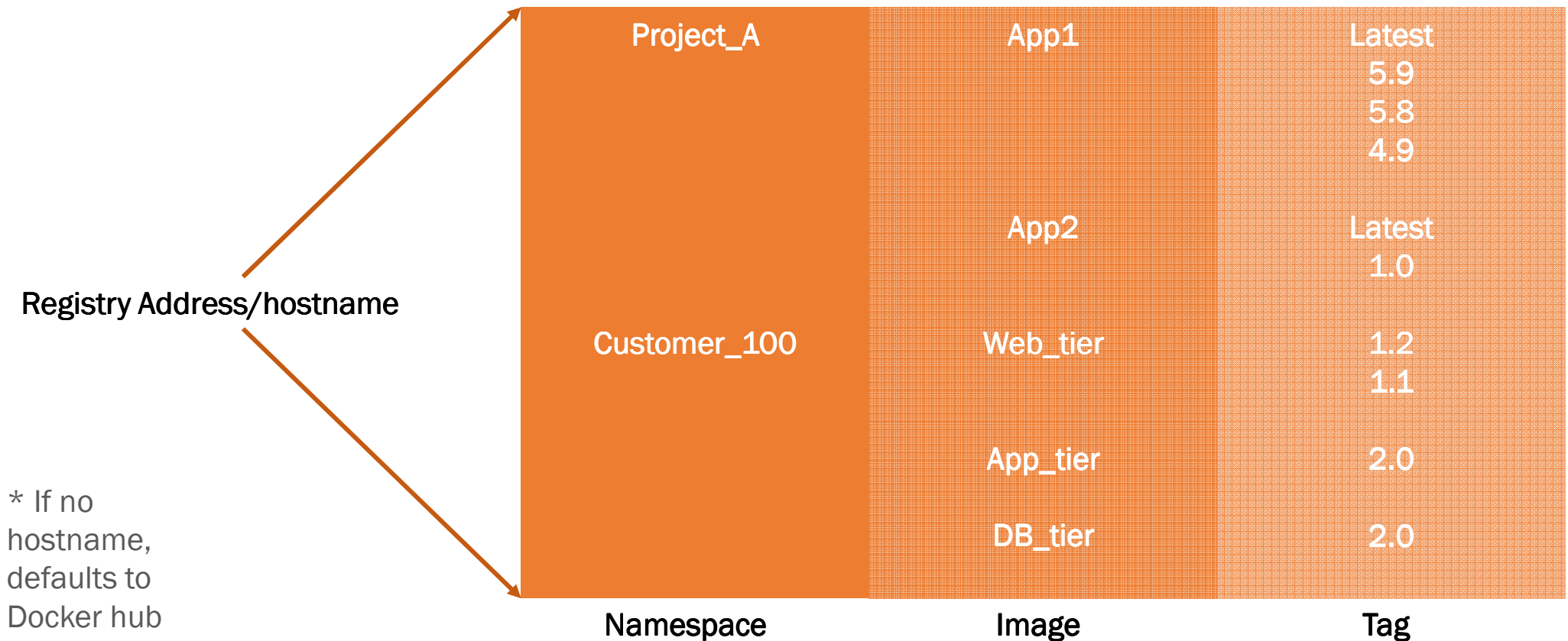


<pre>/ /bin /boot /dev /etc /home /lib64 /mnt /opt /var /usr</pre>	<pre>/ /usr/myapp</pre>	<pre>/ /etc/ssl.keys</pre>	<pre>/ /etc/ssl.keys</pre>	<pre>/ /bin /boot /dev /etc /home /lib64 /mnt /opt /var /usr /usr/myapp</pre>
aabbccdee	aabbccdee	aabbccdee	aabbccdee	Manifest.json

Container Registry



* Container registries are now standardized through the Open Containers Initiative.



Pulling an image from a Registry

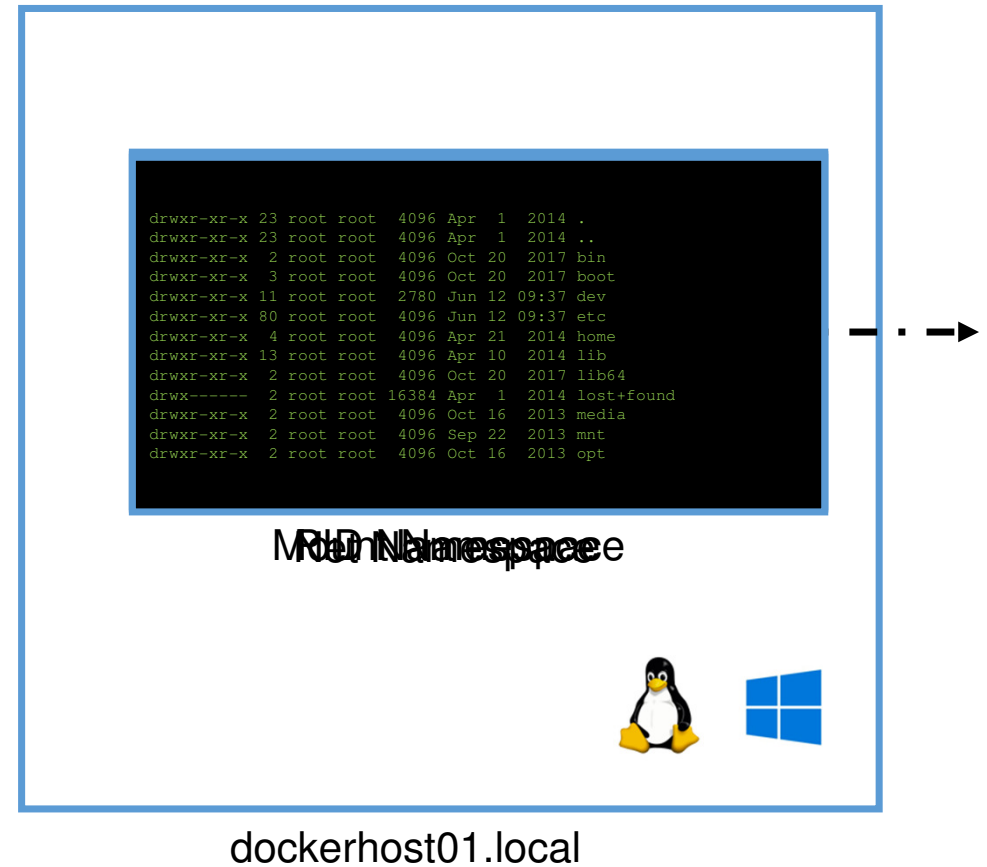


```
[dan@docker ~]$ docker pull quay.io/customer_a/web_tier:1.0  
  
[dan@docker ~]$ docker pull customer_b/app_tier:2.0
```

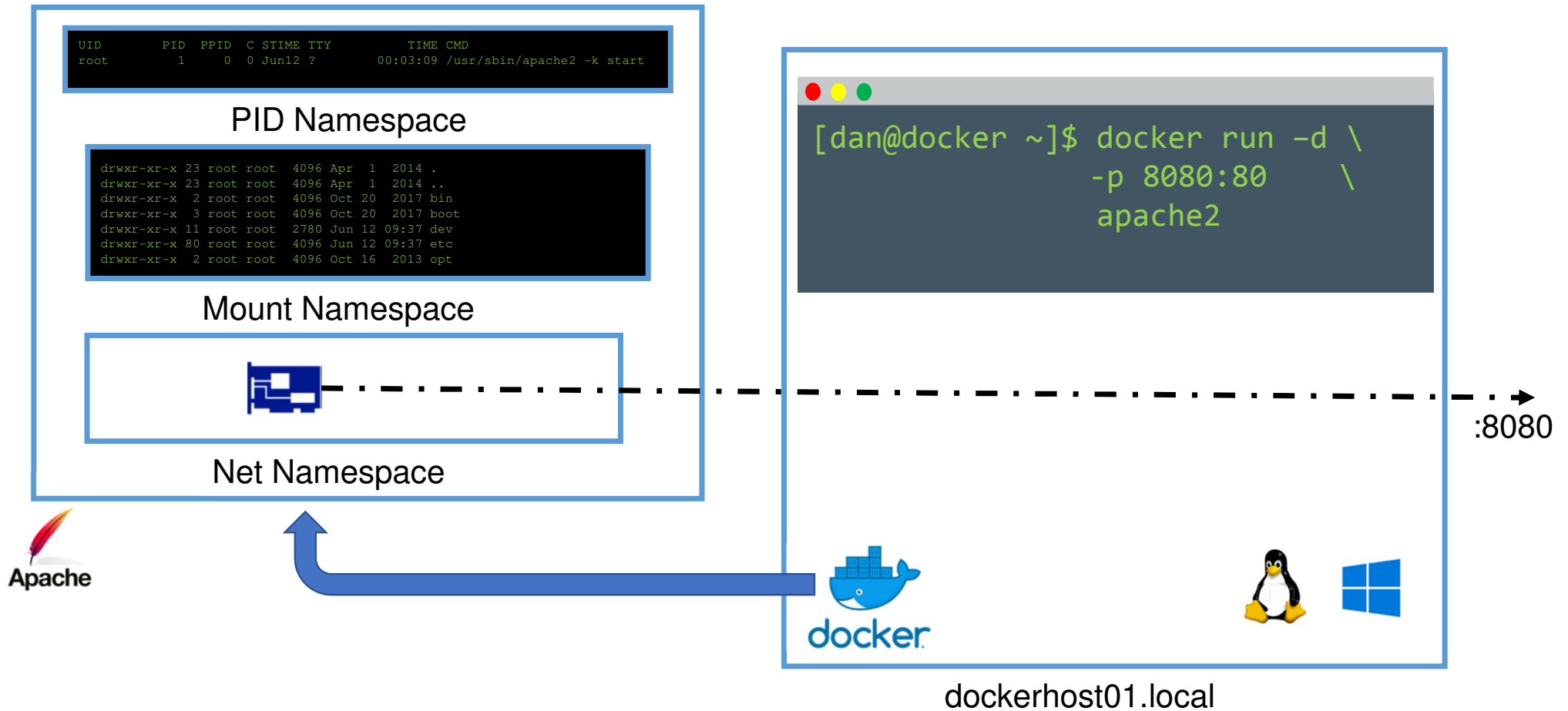
* Above specifies the project(namespace), the image and the specific tag (version) of the image to be pulled.

What happens if no tag is specified?

What is a container?



What is a container?



Using Docker



Using Docker



Docker File

Docker CLI

Docker Compose

Container Orchestration

Dockerfile



FROM defines the base image that the subsequent layers will build on top of.

The base image **scratch** is the smallest base image and allows (as the name suggests) starting from scratch.

```
FROM golang:1.9.2-alpine3.6

RUN go get github.com/thebsdbox/klippy

WORKDIR /go/src/github.com/thebsdbox/klippy

RUN go build -o /bin/klippy

ENTRYPOINT ["/bin/klippy"]
CMD ["--help"]
```

Dockerfile



RUN Creates a new empty layer and will execute a command upon that layer, any filesystem changes will be then stored in the new layer.

In this example **go get** will pull go source code from the internet, which will persist in this new layer

```
FROM golang:1.9.2-alpine3.6

RUN go get github.com/thebsdbox/klippy

WORKDIR /go/src/github.com/thebsdbox/klippy

RUN go build -o /bin/klippy

ENTRYPOINT ["/bin/klippy"]
CMD ["--help"]
```

Dockerfile



WORKDIR sets the current working directory for any subsequent commands

```
FROM golang:1.9.2-alpine3.6

RUN go get github.com/thebsdbox/klippy

WORKDIR /go/src/github.com/thebsdbox/klippy

RUN go build -o /bin/klippy

ENTRYPOINT ["/bin/klippy"]
CMD ["--help"]
```

Dockerfile



ENTRYPOINT specifies the command that will be ran by default when the container is ran.

CMD specifies a command that is ran when no arguments are passed to a container.

```
FROM golang:1.9.2-alpine3.6

RUN go get github.com/thebsdbox/klippy

WORKDIR /go/src/github.com/thebsdbox/klippy

RUN go build -o /bin/klippy

ENTRYPOINT ["/bin/klippy"]
CMD ["--help"]
```

Efficient Dockerfile(s)



```
FROM golang:1.9.2-alpine3.6 AS build

RUN go get github.com/thebsdbox/klippy
WORKDIR /go/src/github.com/thebsdbox/klippy
RUN go build -o /bin/klippy

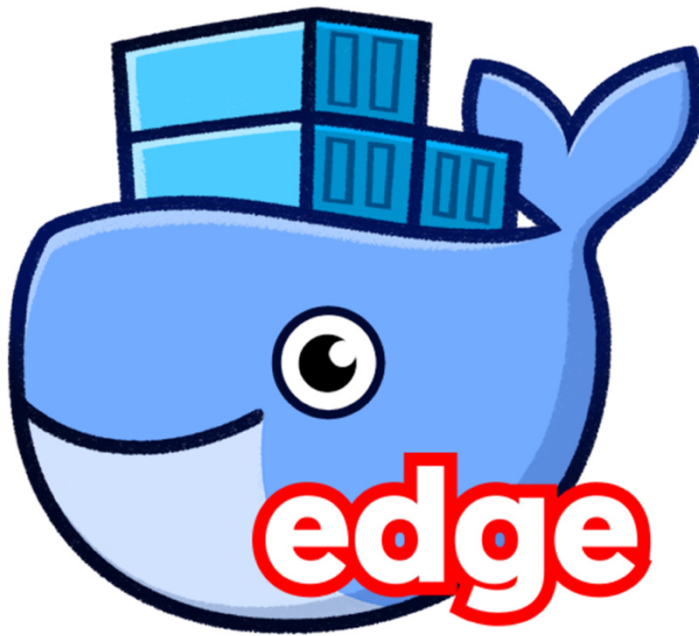
# This results in a single layer image
FROM scratch
COPY --from=build /bin/klippy /bin/klippy

ENTRYPOINT /bin/klippy"
CMD ["--help"]
```

A **MultiPart** docker file allows building your image from a number of other images.

Allows the separation of building/compiling images to the final application image, creating smaller images that only have the assets that are needed.

Docker CLI



The **CLI** is standard across all platforms, I'll be using Docker for Mac to demonstrate some of the more useful commands.

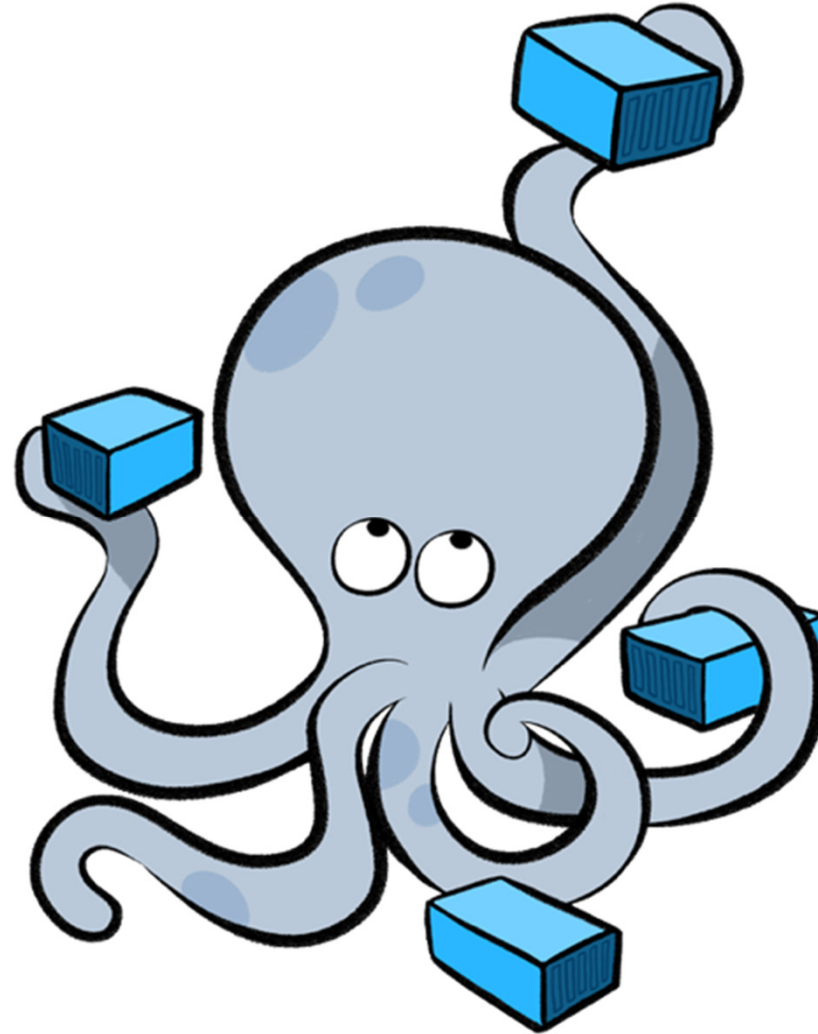
Some **CLI** commands, such as the **stacks** and **service** commands will only appear once a node or multiple nodes are in a cluster (also only master nodes will be able to manage the cluster)

Docker Compose



Docker compose provides the capability to orchestrate, build and deploy an application that is built from multiple containers.

A **compose** file is a `yaml` file that specifies the various containers that make up an application, and how the various containers should interact.



Docker Compose



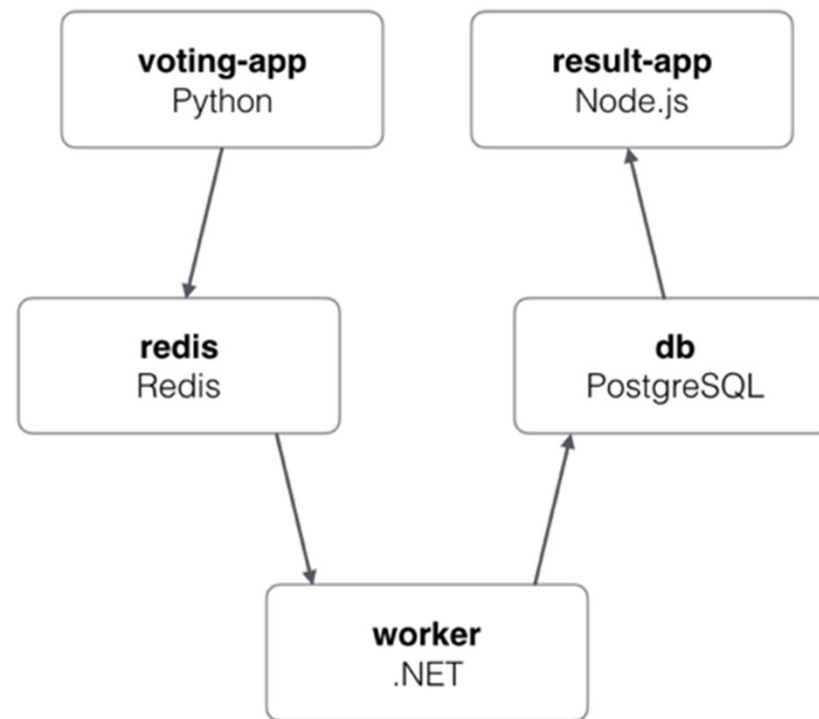
```
version: "3"

services:
  result:
    build: ./result
    command: nodemon server.js
    volumes:
      - ./result:/app
    ports:
      - "5001:80"
      - "5858:5858"

  redis:
    image: redis:alpine
    container_name: redis
    ports: ["6379"]

  db:
    image: postgres:9.4
    container_name: db
    volumes:
      - "db-data:/var/lib/postgresql/data"
```

Docker Compose example



<https://github.com/docker-samples/example-voting-app>

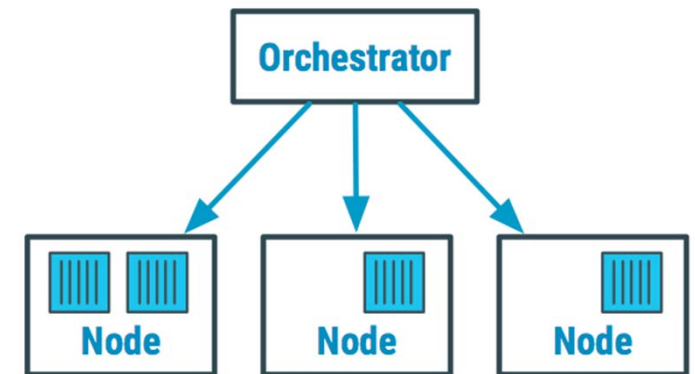
Container Orchestration



Container orchestrators provide the functionality to manage containers over one or more hosts. When a **scheduling** request is made (i.e. asking the orchestrator to deploy an application) the orchestrator will be able to examine the environment and handle the deployment and placement of containers within the cluster.

Additional features (may) include:

- Provisioning hosts
- Instantiating a set of containers
- Rescheduling failed containers
- Linking containers together through agreed interfaces
- Exposing services to machines outside of the cluster
- Scaling out or down the cluster by adding or removing containers



Container Orchestrators



Nomad



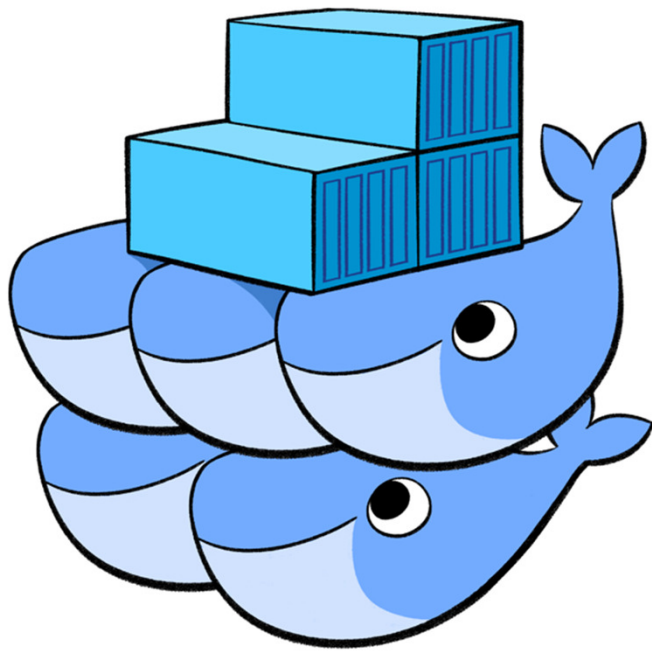
MESOS



kubernetes



Docker Swarm



Docker Swarm: Produces a single, virtual Docker host by clustering multiple Docker hosts together.

It presents the same Docker API; allowing it to integrate with any tool that works with a single Docker host.

On a **Manager** node:

```
docker swarm init
...
    docker swarm join --token aabbcc 192.168.0.1:2377
...
```

On a **Worker** node:

```
docker swarm join --token aabbcc 192.168.0.1:2377
```

Kubernetes



Kubernetes was created by Google and is one of the most feature-rich and widely used orchestration frameworks; its key features include:

- Automated deployment and replication of containers
- Online scale-in or scale-out of container clusters
- Load balancing over groups of containers
- Rolling upgrades of application containers
- Resilience, with automated rescheduling of failed containers
- Controlled exposure of network ports to systems outside of the cluster

Kubernetes is designed to work in multiple environments, including bare metal, on-premises VMs, and public clouds.

Most public clouds now have a managed Kubernetes offering.



Questions?

Agenda



- What is Kubernetes
- Kubernetes Objects
- Kubernetes Networking
- Kubernetes in Action

What is Kubernetes?

- Container Orchestrator
 - Provision, manage, scale applications
- Manage infrastructure resources needed by applications
 - Volumes
 - Networks
 - Secrets
 - And much much more ...
- Declarative model
 - Provide the "desired state" and Kubernetes will make it happen
- What's in a name?
 - Kubernetes (K8s/Kube): 'helmsman' in ancient Greek

How was Kubernetes created?



How was Kubernetes created?

- Based on Google's Borg & Omega
- Open Governance
 - Cloud Native Compute Foundation
- Adoption by Enterprise
 - RedHat, Microsoft, VMware, IBM, and Amazon

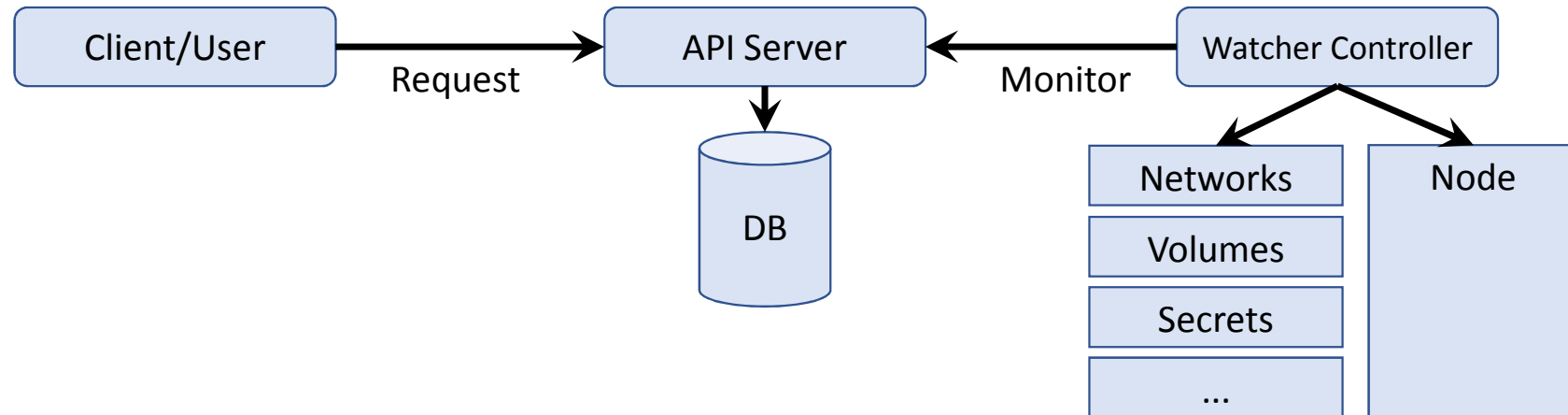
Kubernetes Architecture



At its core, Kubernetes is a database (*etcd*).

- The DB represents the user's desired state
- Watchers & controllers react to changes in the DB.
- Watchers attempt to make reality match the desired state

Kubernetes Architecture



The API Server is the HTTP/REST frontend to the DB

(more on controllers later ...)

Kubernetes Architecture



- To work with Kubernetes, you use *Kubernetes API objects* to describe your cluster's *desired state*: what applications or other workloads you want to run, what container images they use, the number of replicas, what network and disk resources you want to make available, and more.
- Once you've set your desired state, the *Kubernetes Control Plane* works to make the cluster's current state match the desired state. (Source: kubernetes.io)

Kubernetes Resource Model

A resource for every purpose

- Config Maps
- Daemon Sets
- **Deployments**
- Events
- Endpoints
- Ingress
- Jobs
- Nodes
- Namespaces
- **Pods**
- Persistent Volumes
- Replica Sets
- Secrets
- Service Accounts
- **Services**
- Stateful Sets, and more...

- Kubernetes aims to have the building blocks on which you build a cloud native platform.
- Therefore, the internal resource model **is** the same as the end user resource model.

Key Resources

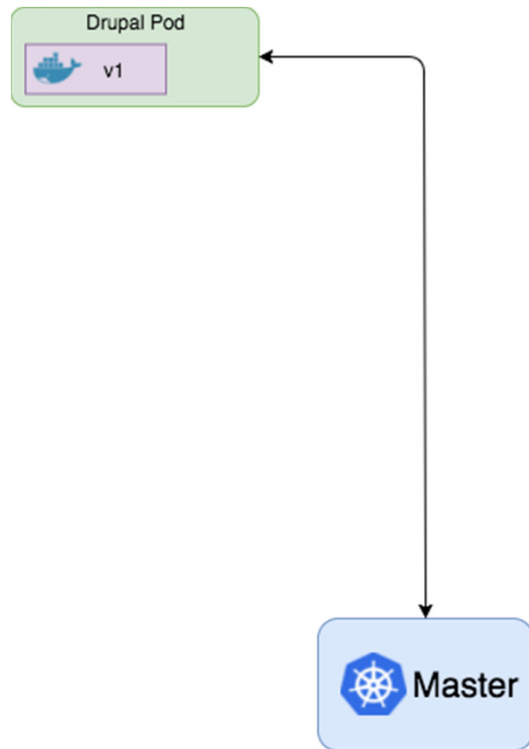
- Pod: set of co-located containers
 - Smallest unit of deployment
 - Several types of resources to help manage them
 - Replica Sets, Deployments, Stateful Sets, ...
- Services
 - Define how to expose your app as a DNS entry
 - Query based selector to choose which pods apply

Kubernetes Objects

The big ones:

- Pod
- Service
- Volume
- Namespace

Kubernetes Objects: pods



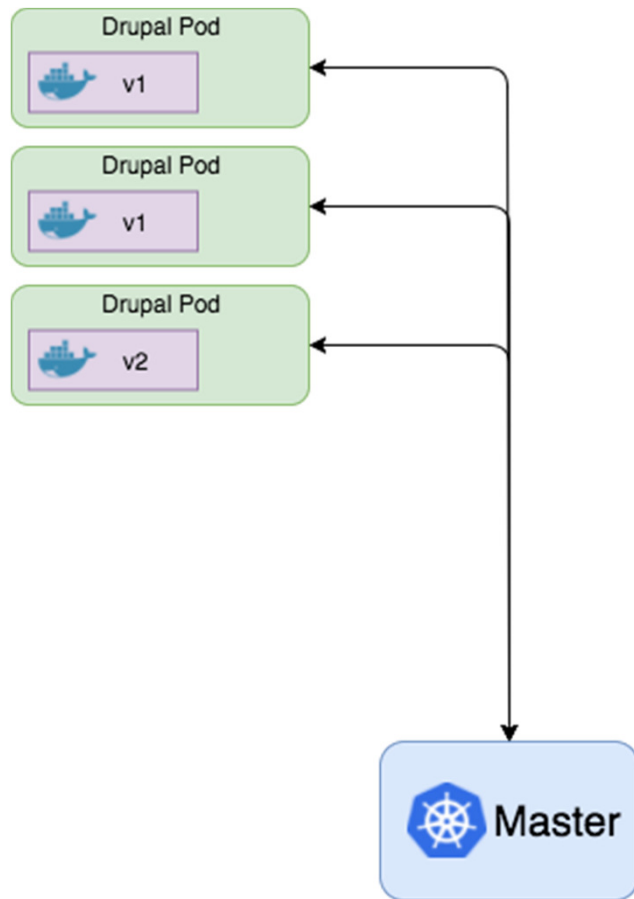
- The Pod is the core component of Kubernetes
- Collection of 1 or more containers
- Each pod should focus on one container, however sidecar containers can be added to enhance features of the core container

```
spec:
  template:
    spec:
      containers:
      - name: drupal
        image: cr.io/repo/mydrupal:v1
```

Kubernetes Objects: pods

- A pod is a running process on your cluster.
 - the smallest, simplest unit
 - may be one container or multiple (tightly-coupled) containers
 - often Docker containers, but k8s supports other container runtimes
- Each pod:
 - has a unique IP
 - can have storage volumes shared with the whole pod
 - is managed by controllers
 - controllers use pod templates to make pods

Kubernetes Objects: pods

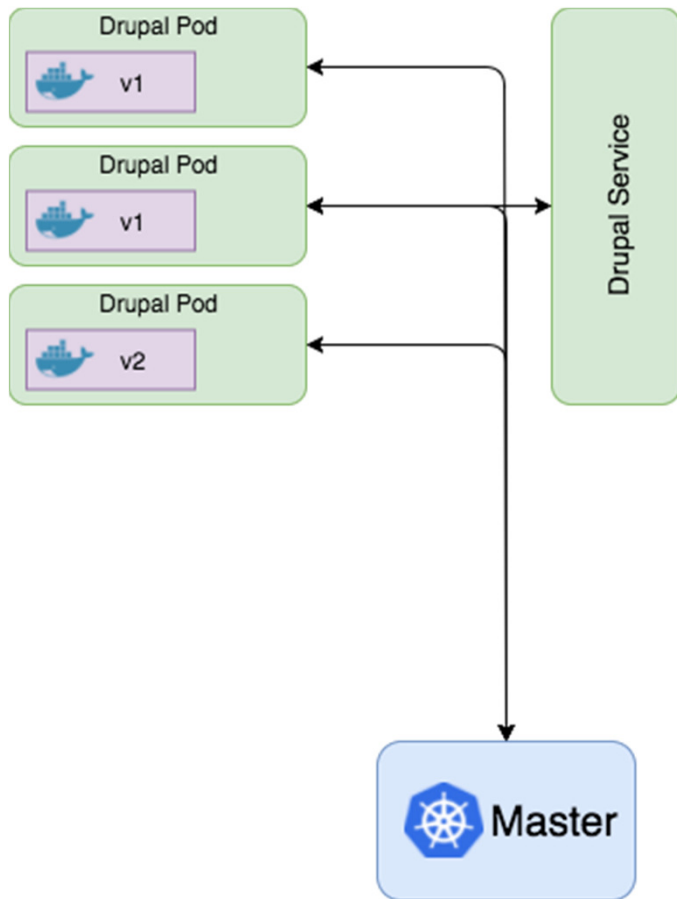


- Once Kubernetes understands what is in a pod, multiple management features are available:
- System Performance
 - Scale up/down the number of pods based on CPU load or other criteria
- System Monitoring
 - Probes to check the health of each pod
 - Any unhealthy ones get killed and new pod is put into service
- Deployments
 - Deploy new versions of the container
 - Control traffic to the new pods to test the new version
 - Blue/Green deployments
 - Rolling deployments

Kubernetes Objects: services

- A logical set of pods
 - like a microservice
 - how groups of pods find each other (e.g., how frontends find backends)
- Services:
 - have their IPs (sometimes called cluster-IP) managed by Kubernetes (although you can manage it yourself)
 - can provide access to pods, but also to outside services
 - are discoverable, and create environment variables

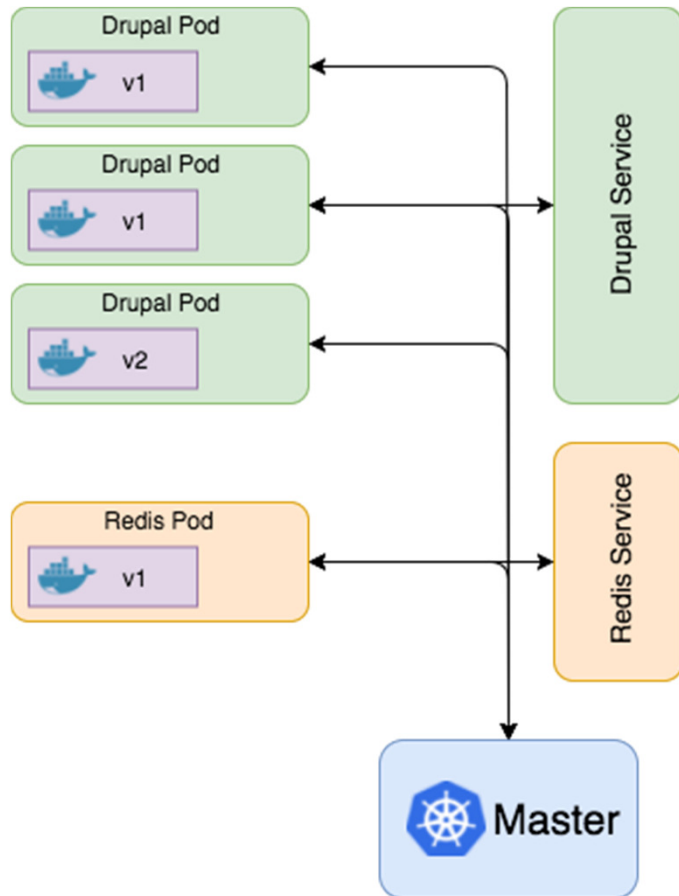
Kubernetes Objects: services



- Kubernetes Services are used to control communications with the pods
 - Load balance the requests
 - Don't send traffic to the unhealthy ones
 - Only talk to the correct version

```
apiVersion: v1
kind: Service
metadata:
  name: drupal
spec:
  selector:
    app: drupal
  ports:
    - name: http-port
      port: 80
  type: LoadBalancer
```

Kubernetes Objects: services



- With the Service architecture Kubernetes handles things that you often might have to worry about
 - Service discovery
 - Load balancing
 - Scaling
- Service discovery allows each pod just needs to call the name of the service it wants to talk to
- Services have multiple options
 - Session based load balancing
 - Single port based services
 - External Services
- The Service architecture of Kubernetes can be scaled up to handle as many services as you would like for your system

Kubernetes Objects: volumes



- a directory, usually with data in it, accessible to containers in a pod
- each container in the pod must specify what volumes and where to mount them
- lots of different kinds of volumes, some provider-specific!

Kubernetes Objects: namespaces



- multiple virtual clusters in the same physical cluster
- provide scope for names: names in each namespace are unique
- can be used to divide resources across users
- you don't need namespaces just to differentiate different versions of the same software: use labels instead

Kubernetes Controllers

- **Deployments**
 - use instead of ReplicaSets
 - describe the desired state, and the controller makes it happen in a controlled way
 - can be versioned for easy rollback
 - give status updates for monitoring (*progressing, complete, fail to progress*)
- **StatefulSets**
 - manages the deployment and scaling of a set of pods, *and provides guarantees about the ordering and uniqueness* of these pods
 - use when you need stable ids, persistent storage, or ordered creation or deletion
- **DaemonSets**
 - runs a copy of a pod in all nodes
 - used for logging, etc.
- **Jobs**
 - creates one or more pods, that run and terminate

Kubernetes Client

- CLI tool to interact with Kubernetes cluster
- Platform specific binary available to download
 - <https://kubernetes.io/docs/tasks/tools/install-kubectl>
- The user directly manipulates resources via json/yaml

```
$ kubectl (create|get|apply|delete) -f myResource.yaml
```

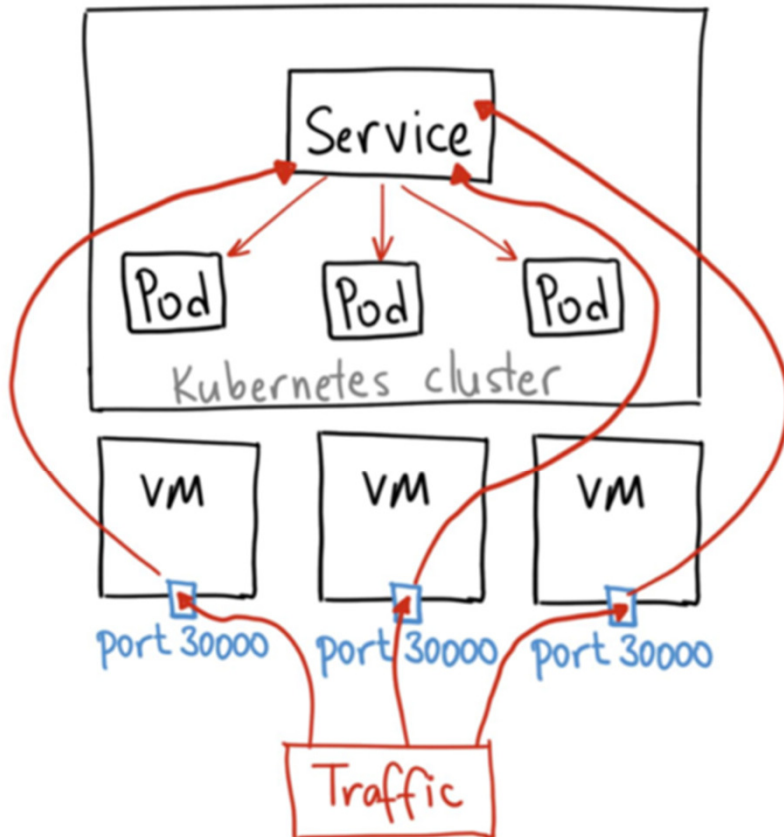
Kubernetes Networking

Type/Features	L2	L3	Overlay	Cloud
Summary	Pods communicate using Layer 2	Pod traffic is routed over underlay network	Pod traffic is encapsulated in an overlay network and uses the underlay network	Pod traffic is routed in a cloud virtual network
Underlying Technology	L2 ARP, Broadcast	Routing protocols such as BGP	VXLAN	Cloud fabric
Encapsulation	None	None	Overlay	None
Examples		Calico	Flannel, Weave	GKS, ACS, AKS

Kubernetes Networking

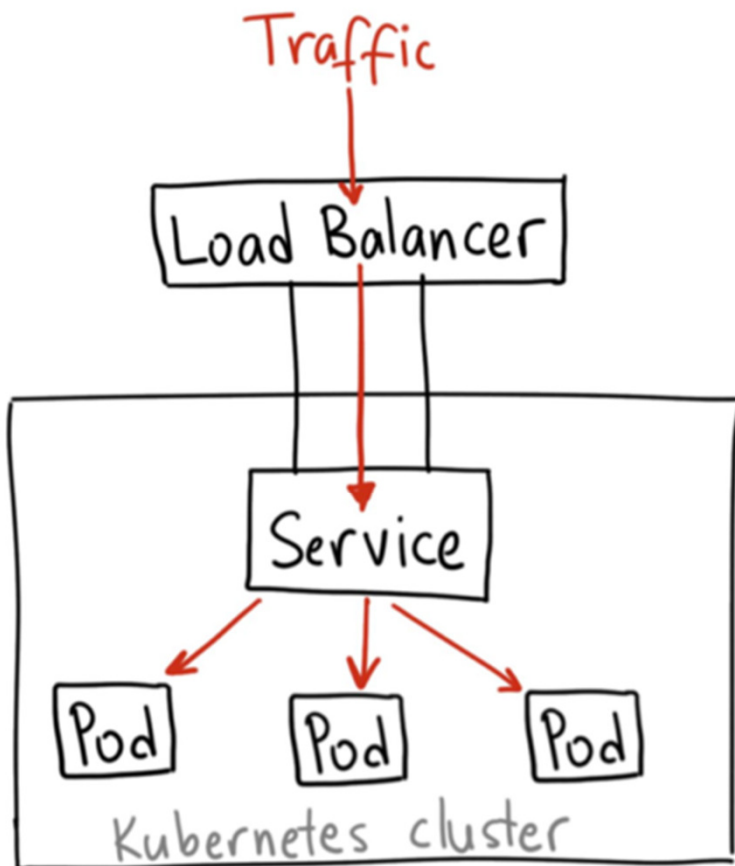
Features	NodePort	Load Balancer	Ingress
Summary	Service is exposed using a reserved port in all nodes of cluster(Default: 32000-32767)	Typically implemented as network load balancer	Typically implemented as http load balancer
IP Address	Node IP is used for external communication	Each service needs to have own external IP	Many services can share same external IP, uses path based demux
Use Case	Testing	L3 Services	L7 Services
Examples		GKE Network Load balancer	Nginx, Contour

NodePort



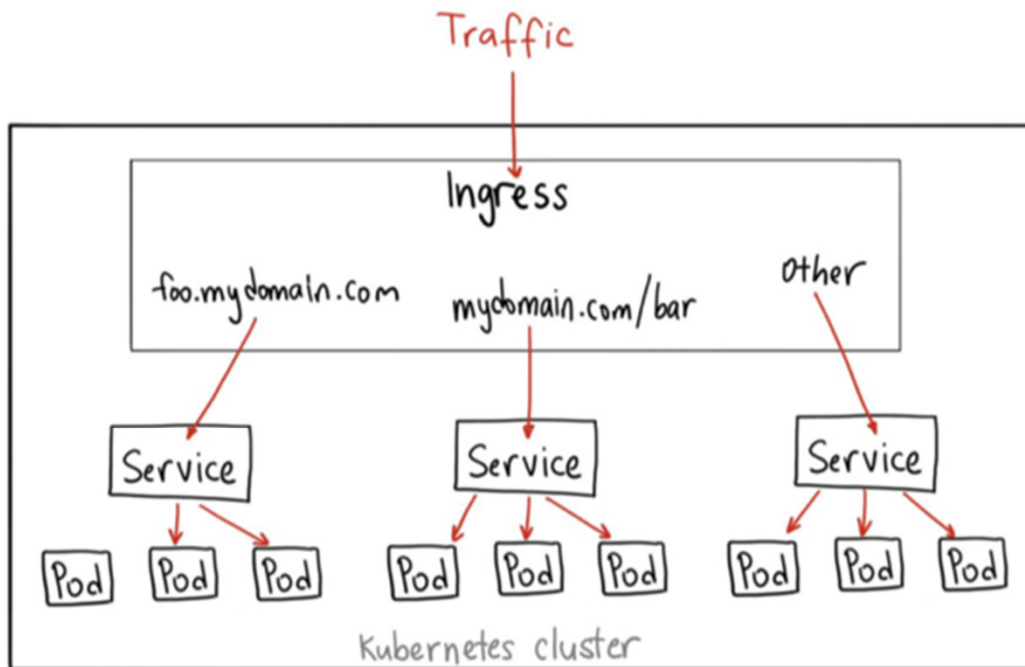
```
apiVersion: v1
kind: Service
metadata:
  name: productpage
  labels:
    app: productpage
spec:
  type: NodePort
  ports:
    - port: 30000
      targetPort: 9080
  selector:
    app: productpage
```

Load Balancer



```
apiVersion: v1
kind: Service
metadata:
  name: productpage
labels:
  app: productpage
spec:
  type: LoadBalancer
  ports:
    - port: 80
    targetPort: 9080
  selector:
    app: productpage
```

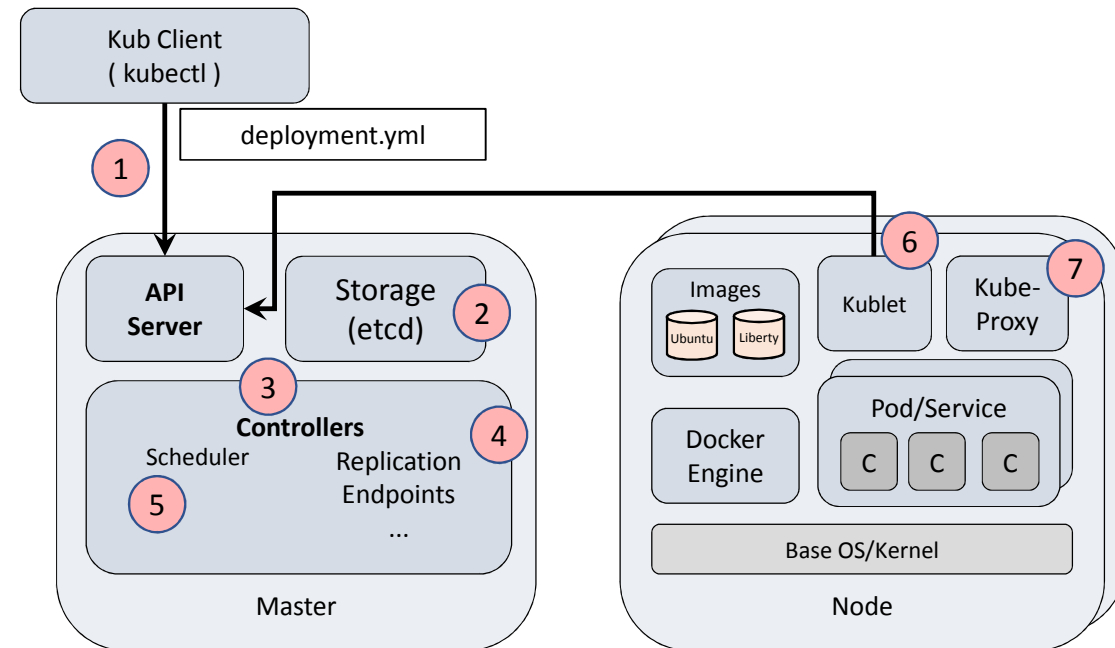
Load Balancer



```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: gateway
spec:
  backend:
    serviceName: productpage
    servicePort: 9080
rules:
  - host: mydomain.com
    http:
      paths:
        - path: /productpage
          backend:
            serviceName: productpage
            servicePort: 9080
```

Under the covers

1. User via "**kubectl**" deploys a new application
2. API server receives the request and stores it in the DB (**etcd**)
3. Watchers/controllers detect the resource changes and act upon it
4. **ReplicaSet** watcher/controller detects the new app and creates new pods to match the desired # of instances
5. Scheduler assigns new pods to a **kubelet**
6. **Kubelet** detects pods and deploys them via the container running (e.g. Docker)
7. **Kubeproxy** manages network traffic for the pods – including service discovery and load-balancing

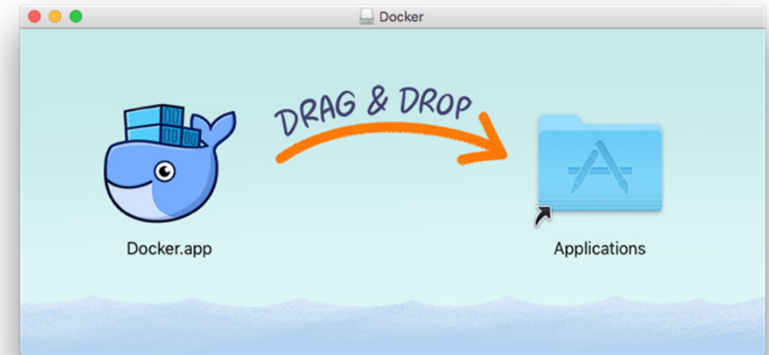


Kubernetes in Action!

Where is Kubernetes?

- Main Website - <http://kubernetes.io>
- Source Code - <https://github.com/kubernetes>
- YouTube - https://www.youtube.com/channel/UCZ2bu0qutTOM0tHYa_jklwg

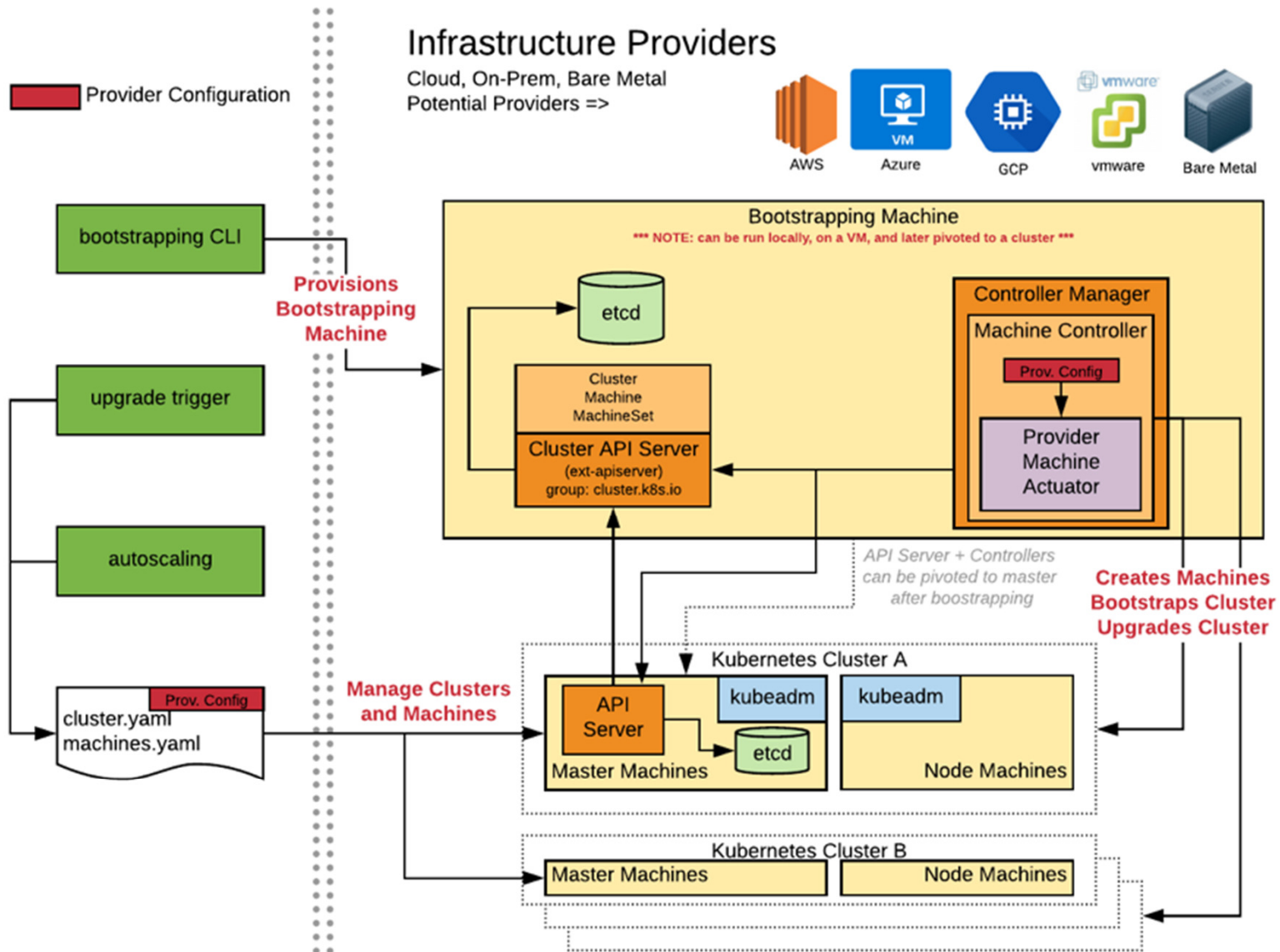
Getting Kubernetes



- Azure Kubernetes Service
- Amazon Elastic Container Service for Kubernetes
- Google Kubernetes Engine
- Digital Ocean Kubernetes

Cluster-API

- A management framework to handle day 1 and 2 operations for kubernetes cluster
 - Day 1) Bringing up a cluster
 - Solves from 0 to Kubernetes
 - Day 2) Managing a cluster
 - Managing in an idiomatic kubernetes way
 - Upgrades
 - Scaling
 - Standardizing a fragmented ecosystem
 - Many tools, all with varying scope and user experience
- Experimenting still!



Kubevirt



Leverage KubeVirt and Kubernetes to manage virtual machines for impractical-to-containerize apps.



Combine existing virtualized workloads with new container workloads on the one platform.



Support development of new microservice applications in containers that interact with existing virtualized applications.

CNCF

* Cloud Native Computing Foundation

CNCF Cloud Native Landscape

2019-02-26T06:38:23Z b4139b2

See the interactive landscape at l.cncf.io

Greyed logos are not open source

App Definition and Development

Database

Streaming & Messaging

Application Definition & Image Build

Continuous Integration & Delivery

Orchestration & Management

Scheduling & Orchestration

Coordination & Service Discovery

Remote Procedure Call

Service Proxy

API Gateway

Service Mesh

Runtime

Cloud-Native Storage

Container Runtime

Cloud-Native Network

Provisioning

Automation & Configuration

Container Registry

Security & Compliance

Key Management

Cloud

Public



This landscape is intended as a map through the previously uncharted terrain of cloud native technologies. There are many routes to deploying a cloud native application, with CNCF Projects representing a particularly well-traveled path

CLOUD NATIVE COMPUTING FOUNDATION

CLOUD NATIVE Landscape

Redpoint Amplify

l.cncf.io

Special

Kubernetes Certified Service Provider

Kubernetes Training Partner

Platform

Certified Kubernetes - Distribution

Certified Kubernetes - Hosted

Certified Kubernetes - Installer

PaaS/Container Service

Observability and Analysis

Monitoring

Logging

Tracing

Chaos Engineering

Serverless

Questions?

Thankyou !