Kubernetes

Starting with Version 1.2

William Stewart
zatech
zoidbergwill basically everywhere

Intro

- Who am I?
- What do I do?

Overview

Docker

Kubernetes

Demo

Kubernetes local development

War stories

What this talk isn't

- An advanced / in-depth look at Kubernetes

  (Unfortunately we don't have the time)

  Hopefully this will be a decent foundation

- A comparison of Kubernetes vs:

  - Mesos/Marathon
  - ECS
  - Helios
  - Swarmkit

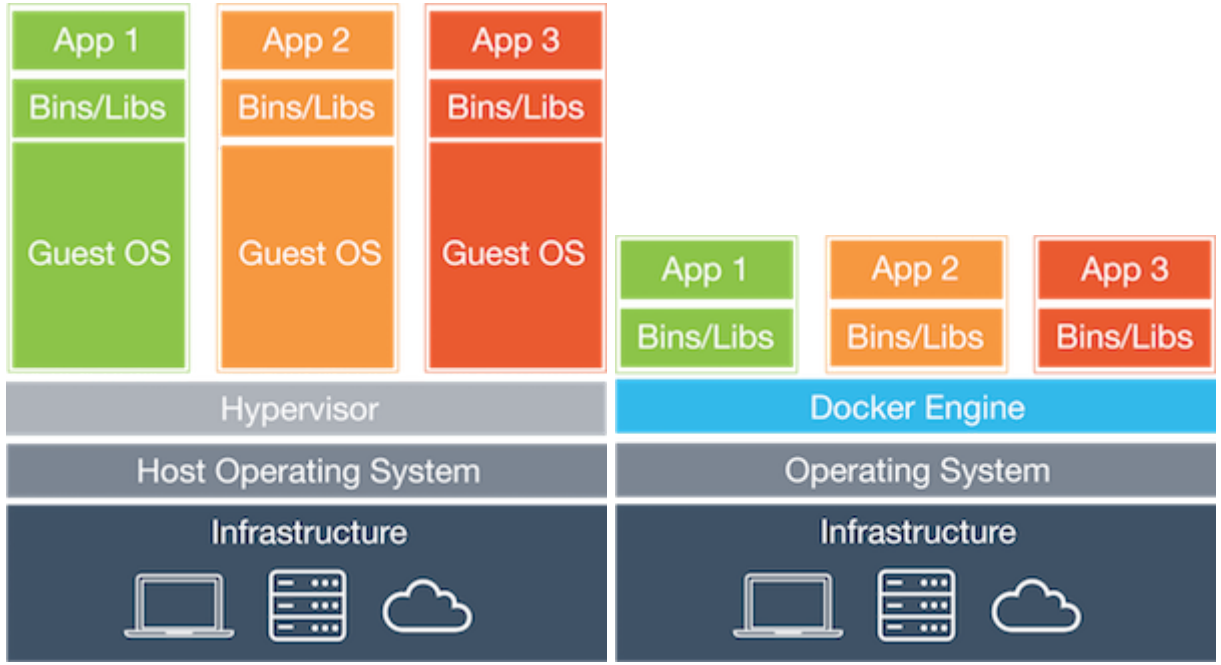
- This is not a zero sum game. People will be using Kubernetes, Swarm, and Mesos for time to come. Don't be afraid of competition, embrace it.

  — Kelsey Hightower (@kelseyhightower) June 20, 2016

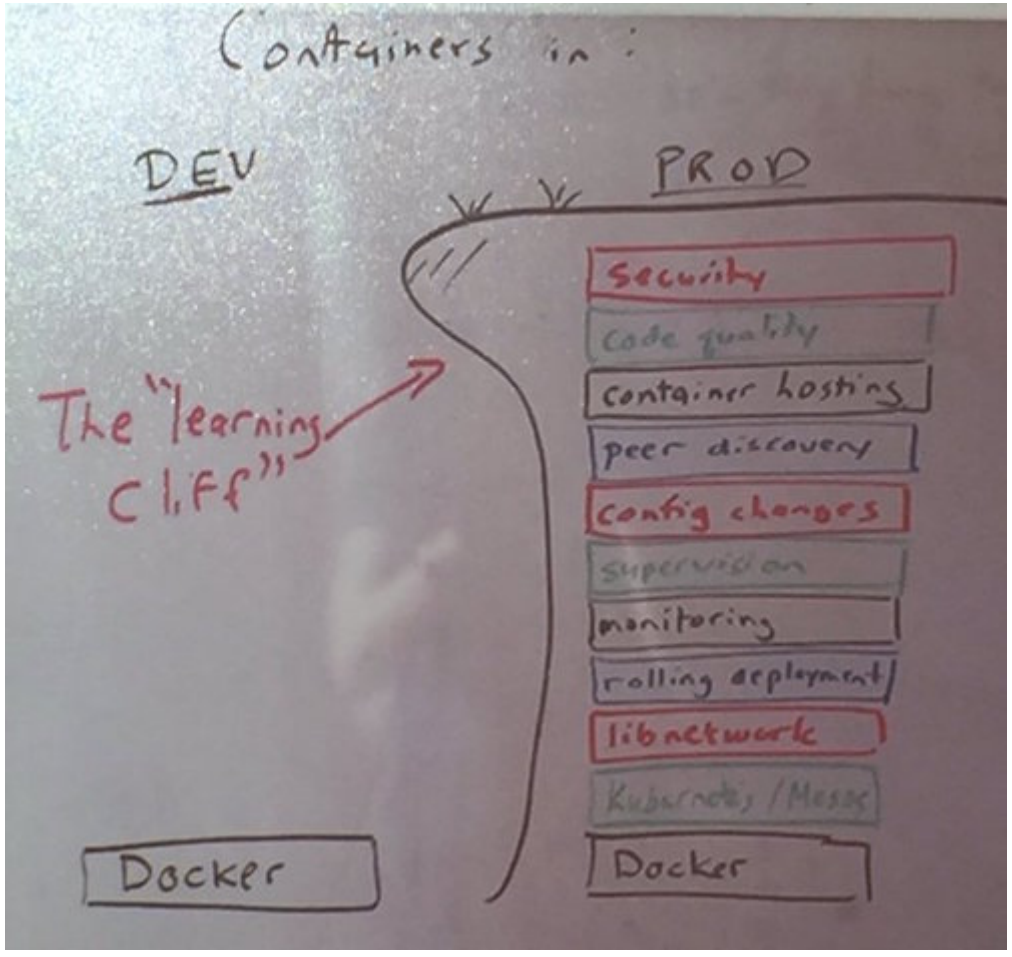
What is Docker?

Docker is a simple standard way of building applications into containers, which means we can build and test the same Docker containers that we are running in production.

Containers are a lighter and more portable version of virtualisation compared to virtual machines.



Running Docker in production



Container cluster managers
can help

Automate the boring stuff

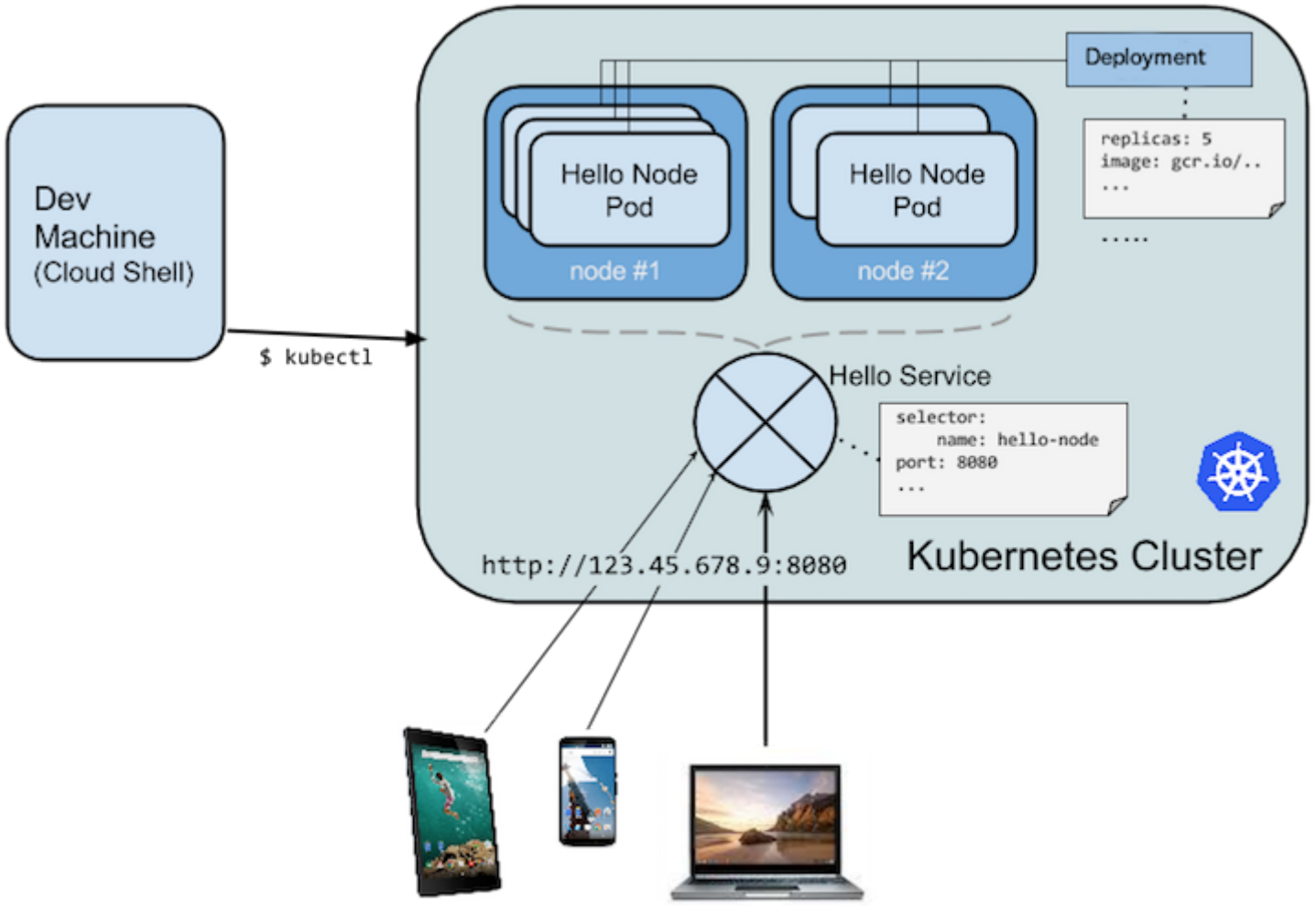
> “Automation is a force multiplier, not a panacea”

Value of automation

- Consistency
- Extensibility
- MTTR (Mean Time To Repair)
- Faster non-repair actions
- Time savings

Dan Luu's Notes on Google's Site Reliability Engineering book

Enter Kubernetes



How does it help?

- Container hosting
- Config changes
- Supervision
- Monitoring
- Rolling deployments
- Networking
- and more...

kubectl

Let's set up a basic environment

```
$ hub clone --depth 1 zoidbergwill/docker-django-migrations-example
$ docker build -t web:1 .
$ kubectl run db --image=postgres --env="POSTGRES_PASSWORD=my-secret-pw" \
  --port 5432
deployment "db" created
$ kubectl expose deployment db
service "db" exposed
$ kubectl run web --image=web:1 --port 80 --env="POSTGRES_PASSWORD=my-secret-pw" \
  --replicas 2  # Or kubectl create -f k8s/web-v1-deployment.yml
deployment "web" created
$ kubectl expose deployment web --type=LoadBalancer
service "web" exposed
$ kubectl get deployments,services
NAME         DESIRED      CURRENT       UP-TO-DATE   AVAILABLE   AGE
db           1            1             1            1           1h
web          2            2             2            2           1h
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
db           10.0.0.100   <none>        5432/TCP     1h
kubernetes   10.0.0.1     <none>        443/TCP      2h
web          10.0.0.65                  80/TCP       1h
```

The building blocks

`Node`

- Containers have to run somwehere
- All machines that talk to the Kubernetes API Server, and can have pods scheduled on them
- They can have unique `labels` which can be useful, for different sized boxes, or guaranteeing `Pods` run on certain `Nodes`.
- Whether it's AWS, DigitalOcean, GCP, or your own tin, they're destined to die some day.

The building blocks

Pod

- The base resource that is scheduled
- It is destined to be re-scheduled, updated, or destroyed.

We're gonna touch more on them in a bit, because `Pods` and `Services` are the main power of Kubernetes.

The building blocks

`Service`

- Simple load balancers that use a selection of labels to route traffic to pods.
- They all have the following:
  - `selector` for finding pods to forward the traffic.
  - `clusterIP` since we have to hit the load balancer somehow
  - `ports` to send the traffic from and to.
  - Potentially more...

Scheduling Pods

`Deployment`

The default way to schedule a pod

e.g. API, DB, Frontend, Workers

`DaemonSet`

Making sure an instance of this pod runs on every node, or every node of a certain type

e.g. Logging agents, Monitoring agents, Cluster storage nodes

`Job`

These are for once off pods.

e.g. Migrations, Batch jobs,

`ReplicationController`

The old default way of scheduling pods...

Basic Resource template

```
$ kubectl get deployment/web -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: "2016-06-19T19:31:33Z"
  generation: 2
  labels:
    run: web
  name: web
  namespace: default
  resourceVersion: "2445"
  selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/web
  uid: "6dfc09fe-3654-11e6-929f-9a4437171650"
spec:
  replicas: 2
  selector:
    matchLabels:
      run: web
  template: ...
status:
  availableReplicas: 2
  observedGeneration: 2
  replicas: 2
  updatedReplicas: 2
```

Pods

What are they?

- The smallest deployable unit
- One or more containers to be scheduled together
- Each pod gets a unique internal IP

Why more than one container?

- Management (e.g. shared fate, horizontal scaling)
- Resource sharing and communication (e.g. sharing volumes, speaking on `localhost`)

Uses

- Log ingestion
- Separating nginx from the webserver
- Local cache management

Fun with labels on Pods

Canary deployments / AB testing

Deployments / Pods

```
name: frontend
replicas: 3
...
labels:
  app: guestbook
  tier: frontend
  track: stable
...
image: gb-frontend:v3
---
name: frontend-canary
replicas: 1
...
labels:
  app: guestbook
  tier: frontend
  track: canary
...
image: gb-frontend:v4
```

Service

```
selector:
  app: guestbook
  tier: frontend
```

More Fun with Pods

Orphan'ing a pod

Deployments / Pods

The canary is acting up.

Let's make sure it doesn't get scaled down in an update:

```
name: frontend-canary
...
labels:
  track: canary
...
image: gb-frontend:v4
```

Service

```
selector:
  app: guestbook
  tier: frontend
```

How we got to Kubernetes 1.2

- `ReplicationControllers` `kubectl rolling-update` to new image tags, which directly created `Pods`
- `Deployments` creating `ReplicaSets` with hashes of the pod spec which go on to create `Pods`, and we can `kubectl rollout undo`
- Factor config out into `Secrets` and `ConfigMaps`, and load values from them into `Deployments`

A Simple Pod Spec

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  ...
  name: web
spec:
  ...
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: web
    spec:
      containers:
      - env:
        - name: POSTGRES_PASSWORD
          value: my-secret-pw
        - name: QOTD
          value: Nostalgia isn't what it used to be.
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        image: web:1
        name: web
        ports:
        - containerPort: 80
        resources: {}
status: {}
```

ConfigMap & Secret

`qotd-configmap.yml`:

```
apiVersion: v1
data:
  january: "Nostalgia isn't what it used to be."
  february: Richard Stallman exists because he compiled himself into being.
kind: ConfigMap
metadata:
  name: qotd
  namespace: default
```

`postgres-secret.yml`:

```
apiVersion: v1
data:
  # echo "my-secret-pw" | base64
  password: bXktc2VjcmV0LXB3Cg==
kind: Secret
metadata:
  name: postgres
type: Opaque
```

A More Fancy Pod Spec

```
spec:
  ...
  template:
      ...
    spec:
      containers:
      - env:
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres
              key: password
        - name: QOTD
          valueFrom:
            configMapKeyRef:
              name: qotd
              key: january
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        livenessProbe:
          httpGet:
            path: /healthz
            port: 80
            scheme: HTTP
          initialDelaySeconds: 30
          timeoutSeconds: 5
        image: web:1
        name: web
        ports:
        - containerPort: 80
```

Resources I'm gonna ignore

horizontalpodautoscalers (aka 'hpa')

```
$ kubectl autoscale deployment web --min=2 --max=10  --cpu-percent=80
```

ingress (aka 'ing')

```
Ingress is a collection of rules that allow inbound connections to reach
the endpoints defined by a backend. An Ingress can be configured to give
services externally-reachable urls, load balance traffic, terminate SSL,
offer name based virtual hosting etc.
```

limitranges (aka 'limits')

These allow setting namespace-wide resource-specific limit ranges.

persistentvolumeclaims (aka 'pvc')

These are an interesting abstraction on top of peristent volumes.

Kubernetes local development

- docker-compose (Not really accurate)
- hyperkube (Running a single-node cluster in docker-machine/locally)
- localkube / minikube (Officlal github.com/kubernetes/minikube) There was:
  - boot2kube/kmachine/kcompose
  - monokube
  - localkube

```
## Primary Goals

From a high level the goal is to make it easy for a new user to run a
Kubernetes cluster and play with curated examples that require least
amount of knowledge about Kubernetes.
These examples will only use kubectl and only a subset of Kubernetes
features that are available will be exposed.

- Works across multiple OSes - OS X, Linux and Windows primarily.
- Single command setup and teardown UX.
- Unified UX across OSes
- Minimal dependencies on third party software.
- Minimal resource overhead.
- Eliminate any other alternatives to local cluster deployment.
```

[local cluster UX proposal]

War stories

Memcached

Kafka

Migrations

Accurate local dev

Memcached and Me being dumb

Kafka: Persistent Storage

- tutorials make running zookeeper/kafka easy
- they cheat.

Lots of storage works

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs
- iscsi
- flocker
- glusterfs
- rbd
- gitRepo
- secret
- persistentVolumeClaim
- downwardAPI
- FlexVolume
- AzureFileVolume
- vsphereVirtualDisk

Sneaky Examples

Most Kubernetes examples use replicas with volumes using `emptyDir` or `hostPath` are easy, `gcePersistenDisk` less so

A Kafka example:

```
...
spec:
  replicas: 3
  template:
    spec:
      volumes:
        - name: data
          emptyDir: {}
      containers:
        - name: server
          image: elevy/kafka:latest
          ...
          volumeMounts:
          - name: data
            mountPath: /kafka/data
```

GCP showing off running exhibitor in Kubernetes

```
 replicas: 3
 template:
    volumes:
    - name: nfs
      nfs:
        server: singlefs-1-vm
        path: /data
```

gcePersistentDisk

They only allow a single pod binding a disk in read-write mode at a time, so as soon as you scale beyond one replica it cries.

So you have to make `Deployments` for each individual pod for now.

```
volumes:
  - name: data
    gcePersistentDisk:
      pdName: kafka-1
      fsType: ext4
containers:
  - name: server
    ...
    volumeMounts:
      - mountPath: /kafka
        name: data
nodeSelector:
  custom/node-id: "1"
```

Kafka: Pods are disposable

- Kafka v.0.9 can auto assign broker IDs, which seems useful for disposable pods, but it breaks when using another broker IDs storage volume.
- Peer discovery is also hard, but unique deployments make it possible, with unique services:

```
spec:
replicas: 1
metadata:
  name: kafka-3
  labels:
    app: kafka
    server-id: "3"
```

Migrations

- Distributed systems are hard.
- Schema changes are hard.
- Rolling updates to distributed systems with schema changes are hard

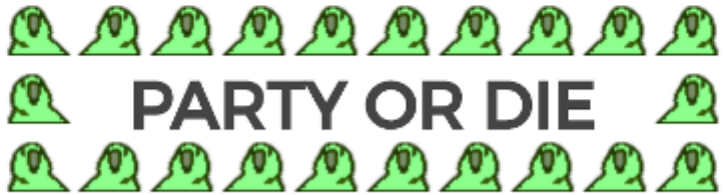
Docker sucks

Everything sucks

I've whined about local development and it's getting better

Superbalist.com and Takealot.com are hiring!!!

Cult of the Party Parrot



Thanks, Questions, and Sauce

Code

https://zoidbergwill.github.io/presentations/2016/kubernetes-1.2-and-spread/

Things you should read

- O'Reilly's Site Reliability Engineering: How Google Runs Production Systems Amazon
- Dan Luu's notes are good too
- Borg, Omega, and Kubernetes Lessons learned from three container-management systems over a decade. Essay
- Running a single node Kubernetes cluster with Docker
- `kubernetes-dashboard`
- Cloud Native Computing Foundation
- Kubernetes-Anywhere: An official Kubernetes repo with some documentation on running Kubernetes with Docker for Mac beta
- `minikube`: The official Go binary for running a simpler local cluster.
- `awesome-kubernetes` list on GitHub, cuz it has some neat things.