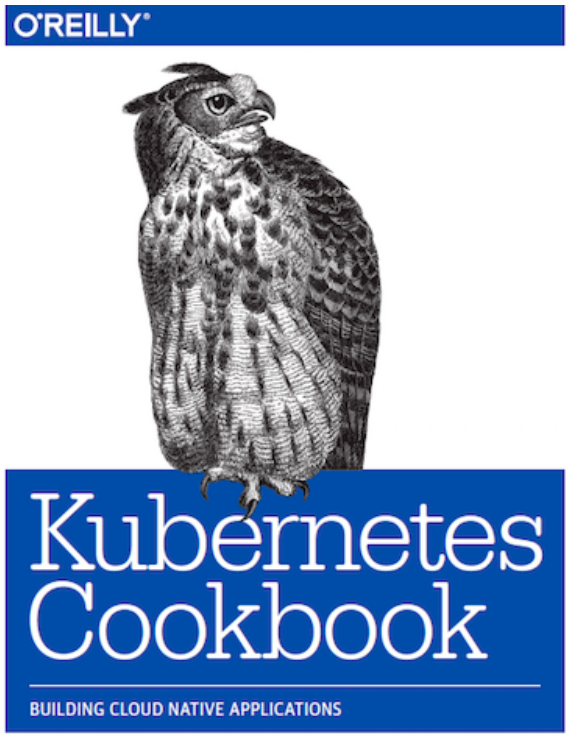
Velocity Kubernetes Training - O'REILLY

By Sebastien Goasguen, author of the Docker cookbook and upcoming Kubernetes cookbook. Senior Director of Cloud Technologies, Bitnami.
@sebgoa https://github.com/kubeless/kubeless



Pre-requisities

- minikube , https://github.com/kubernetes/minikube
- kubectl , https://kubernetes.io/docs/user-guide/prereqs/
- git

Manifests here:

https://github.com/sebgoa/oreilly-kubernetes

News of 2016

Kubernetes Training

- Two days
- We break every 90 minutes.
- 15 minutes break.
- One hour lunch break

Day 1

- Introduction and Kubernetes architecture
- Installation options (Minikube and kubeadm)
- API objects (metadata, annotation, labels, specs, and schemas)
- `kubectl` (tips and tricks)
- Services and ingress (service types, controllers, and ingress rules)
- Deployments (rolling updates and rollbacks)
- Advanced Pods (probe, init-containers…)

Day 2

- Custom Resource Definitions (extended Kubernetes API, etc.)
- The Kubernetes ecosystem (Helm, Python client)
- Volumes, DaemonSets
- Scheduling (Node/Pode affinity and custom schedulers)
- Security (RBAC, and network policies)
- Logging and monitoring (Heapster, Prometheus, and Fluentd)

Are you READY ?

All set ....

Part I: Introduction

Kubernetes is an open-source software for automating deployment, scaling, and management of containerized applications.

Builds on 15 years of experience at Google.

Google infrastructure started reaching high scale before virtual machines and containers provided a fine grain solution to pack clusters efficiently.

- Open Source and available on GitHub
- Apache Software License
- Now governed by the Cloud Native Computing Foundation at the Linux Foundation
- Several Special Interest Groups (SIG)
- Open to everyone
- Weekly Hangouts

Other Solutions

Containers have seen a huge rejuvenation in the last 3 years. They provide a great way to package, ship and run applications (Docker moto).

- Building containers on developers's machine is easy
- Sharing images is easy via Docker registries

But managing containers at scale and architecting a distributed applications based on microservices principles is still challenging.

Kubernetes provides a powerful API to manage distributed applications.

Other solutions:

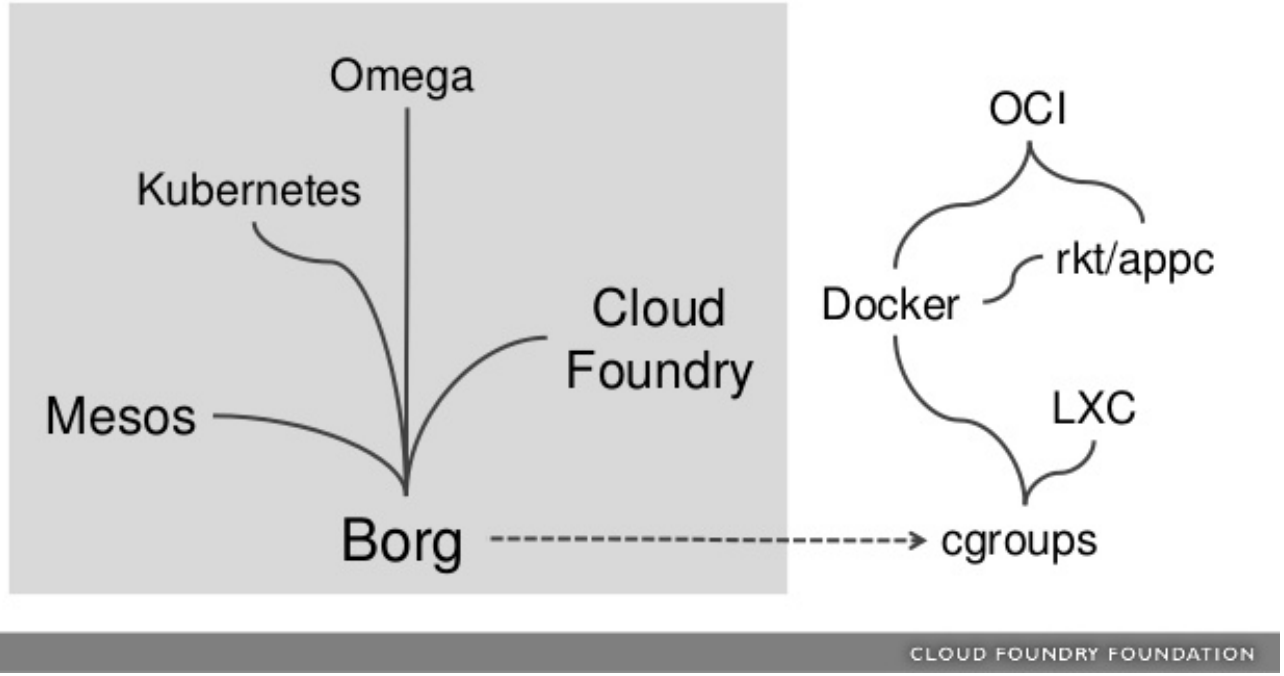
- Docker Swarm
- Hashicorp Nomad
- Apache Mesos
- Rancher

Borg Heritage

- Borg was a Google secret for a long time.
- Orchestration system to manage all Google applications at scale
- Finally described publicly in 2015
- Paper explains ideas behind Kubernetes

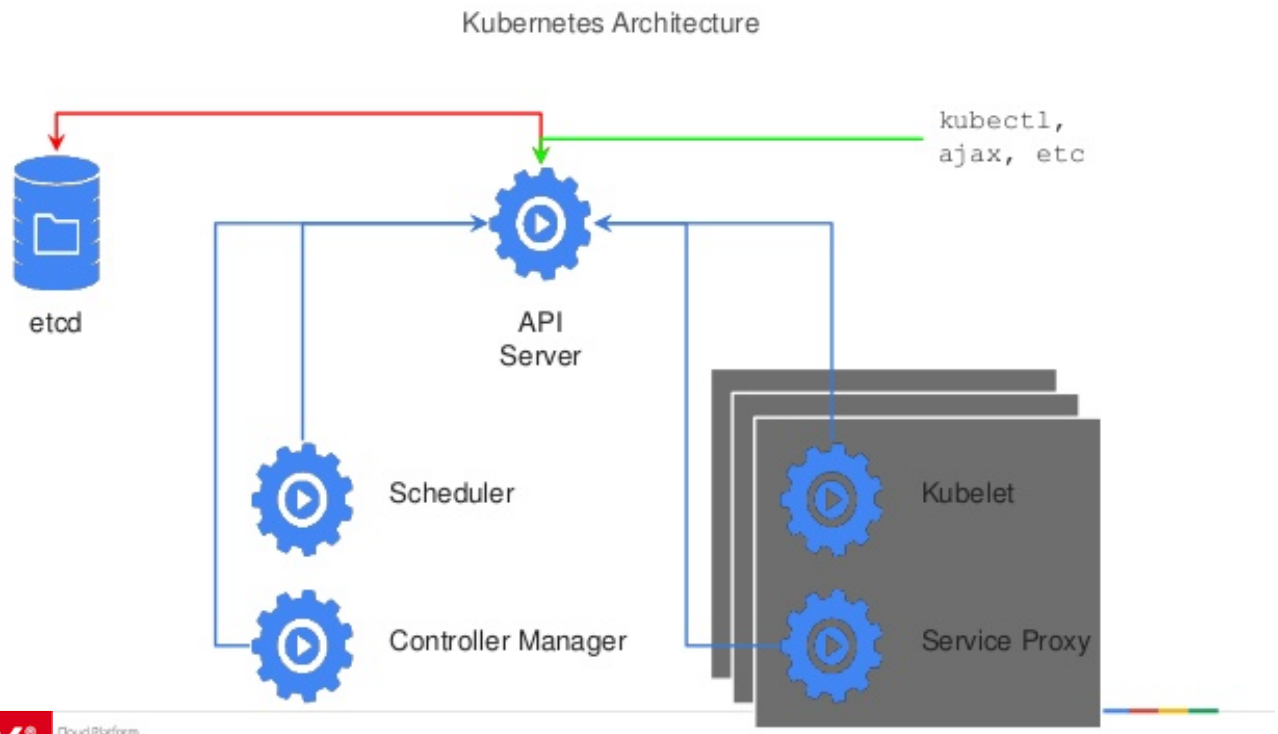
Kubernetes Lineage

- Google contributed `cgroups` to the Linux kernel
- `cgroups` and linux `namespaces` at the heart of containers
- Mesos was inspired by discussions with Google when Borg was still secret
- Cloud Foundry implements 12 factor apps principles for microservices apps.



What is it really ?

- A cluster manager
- A scheduler to place containers in a cluster
- Lots of HA features
- Geared towards managing apps at scale
- Deployed as services on VMs or Bare-metal machines



How is it doing ?

- Open Source in June 2014 (2 years old)
- 1200 + contributors
- 50k commits
- Second Biggest Golang project on GitHub (Docker #1)
- Google and Red Hat lead contributors
- Meetups in +100 cities worldwide
- +20,000 people on Slack
- 1 release every ~3 months

A Tour of Web Resources

Let's get a browser and tour the various key resources ...

- kubernetes.io
- Documentation
- CNCF
- GitHub
- YouTube Channels

Part I: Installation and Discovery

A look at:

- `gcloud` to use GKE
- `minikube` for local development
- `kubeadm` to build your own cluster

Getting Started with Kubernetes Easily

To get started without having to dive right away into configuring a cluster, there are two choices:

- Google Container Engine (GKE)

```
$ gcloud container cluster create ricardo
```

Needs an account on Google cloud, create a Kubernetes cluster in the Cloud using GCE instances.

- Minikube

```
$ minikube start
```

Install `minikube` locally and discover Kubernetes on your local machine

Minikube

Minikube is open source and available on GitHub.

Install the latest release. e.g on OSX:

```
$ curl -Lo minikube
https://storage.googleapis.com/minikube/releases/v0.20.0/minikube-darwin-
amd64
$ chmod +x minikube
$ sudo mv minikube /usr/local/bin/
```

You will need an "Hypervisor" on your local machine, e.g VirtualBox, KVM, Fusion

```
$ minikube start
```

Minikube Usage

You now have `minikube` installed on your machine, you can use it for development and learning the API. Kubernetes runs inside a VM and is setup using a single binary called `localkube`. You cannot use minkube to learn how to operate and configure the system internally. We will use it to learn the API and use the `kubectl` client.

```
Usage:
minikube [command]

Available Commands:
dashboard Opens/displays the kubernetes dashboard URL
delete Deletes a local kubernetes cluster.
docker-env sets up docker env variables; similar
get-k8s-versions Gets the list of available kubernetes
ip Retrieve the IP address of the running cluster.
logs Gets the logs of the running localkube instance,
service Gets the kubernetes URL for the specified
ssh Log into or run a command on a machine with SS
start Starts a local kubernetes cluster.
status Gets the status of a local kubernetes cluster.
stop Stops a running local kubernetes cluster.
version Print the version of minikube.
```

kubeadm

`kubeadm` is a new CLI tool to ease deployment of Kubernetes.

- `kubeadm init` on the master node
- `kubeadm join` on the worker nodes

`kubeadm` runs the `kubelet` via systemd, and manifests for the other Kubernetes components are run via manifests.

Demo of `kubeadm` on DigitalOcean.

Documentation

Part II: The API and `kubectl`

- `kubectl`
- Introduction to key primitives
- The Kubernetes API

Become Friends with Pods

Pods are the lowest compute unit in Kubernetes. Group of containers and volumes.

Simplest form, single container

```
apiVersion: v1
kind: Pod
metadata:
  name: foobar
spec:
  containers:
  - name: ghost
    image: ghost
```

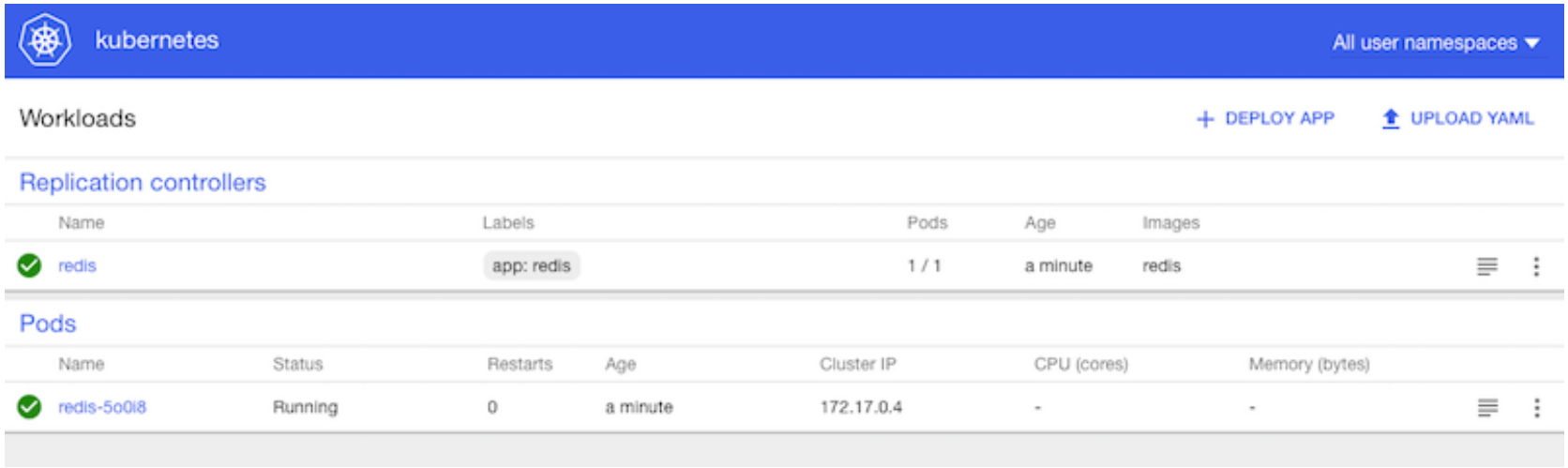
And launch it with:

```
kubectl create -f pod.yml
```

Starting Your First Kubernetes Application via the Dashboard

```
$ minikube dashboard
```

- Click on `+Deploy App`
- Specify `redis` as a container image
- What is happening ?
- Check the logs ?
- `minikube ssh` is the container running ?



Using the Kubernetes CLI

The Dashboard is nice to use but the Kubernetes CLI is extremely powerful.
The k8s CLI is called `kubectl`

- Install `kubectl`

```
$ wget https://storage.googleapis.com/kubernetes-
release/release/v1.6.4/bin/darwin/amd64/kubectl
$ wget https://storage.googleapis.com/kubernetes-
release/release/v1.6.4/bin/linux/amd64/kubectl
```

- Access your Redis *application*

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
redis-5o0i8 1/1 Running 0 14m
$ kubectl logs
$ kubectl logs redis-5o0i8
_.-_
_.-``__ ''-._
_.-``    `.  `_.  ''-._ Redis 3.2.3 (00000000/0) 64 bit
.-`` .-```.  ```\/ _.,_ ''-._
...
$ kubectl exec -ti redis-5o0i8 bash
root@redis-5o0i8:/data# redis-cli
127.0.0.1:6379>
```

Access the Application with `kubectl port-forward`

Accessing an application running in a container via `exec` is for debugging.

To access the containerized app from your local machine you can use a `port-forward` technique.

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
redis-aoo4c 1/1 Running 0 3m

$ kubectl port-forward redis-aoo4c 6379:6379
I0816 11:00:54.773757 16518 portforward.go:213] Forwarding from
127.0.0.1:6379 -> 6379
I0816 11:00:54.774560 16518 portforward.go:213] Forwarding from [::1]:6379
-> 6379
I0816 11:00:58.193456 16518 portforward.go:247] Handling connection for
6379
```

Locally you can use a Redis client to connect to the Redis running in a container started by Kubernetes.

```
$ redis-cli
127.0.0.1:6379>
```

What k8s Objects Do We See in the Dashboard ?

Toggle the resources listing

Check API Resources with `kubectl`

Check it with `kubectl`:

```
$ kubectl get pods
$ kubectl get rs
$ kubectl get ns
```

But there is much more

```
$ kubect proxy &
$ curl http://127.0.0.1:8080
{
"paths": [
"/api",
"/api/v1",
"/apis",
...
$ curl http://127.0.0.1:8080/api
```

Powerful REST based API

YAML or JSON definitions for objects

```
$ kubectl --v=9 get pods
...
I0816 11:20:40.722829 16899 round_trippers.go:286] GET
https://192.168.99.100:8443/api/v1/namespaces/default/pods 200 OK in 2
milliseconds
I0816 11:20:40.722867 16899 round_trippers.go:292] Response Headers:
I0816 11:20:40.722880 16899 round_trippers.go:295] Content-Type:
application/json
I0816 11:20:40.722891 16899 round_trippers.go:295] Date: Tue, 16 Aug 2016
09:20:40 GMT
I0816 11:20:40.722958 16899 request.go:870] Response Body:
{"kind":"PodList","apiVersion":"v1","metadata":
{"selfLink":"/api/v1/namespaces/default/pods","resourceVersion":"722"},"items":
[{"metadata":{"name":"nginx-n0bla","generateName":"nginx-
","namespace":"default","selfLink":"/api/v1/namespaces/default/pods/nginx-
n0bla","uid":"3ad10ac5-638e-11e6-bf50-cec7655de670"
...
```

You can `get` every object, as well as `delete` them:

```
$ kubectl get pods redis-aoo4c -o yaml
apiVersion: v1
kind: Pod
metadata:

$ kubectl delete pod/redis-aoo4c
```

Pods

Represents a group of collocated containers and associated volumes. Top level API object to run containers, smallest compute unit in k8s.

Full specifications well documented. See also the user-guide

```
$ cat redis.yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - image: redis:3.2
    imagePullPolicy: IfNotPresent
    name: mysql
  restartPolicy: Always

$ kubectl create -f redis.yaml
```

Namespaces

Every request is namespaced e.g `GET`
`https://192.168.99.100:8443/api/v1/namespaces/default/pods`

Namespaces are intended to isolate multiple groups/teams and give them access to a set of resources. Each namespace can have quotas. In future releases, we will have RBAC policies in namespaces (i.e define read/write access for users within/between namespaces)

```
$ kubectl get ns
$ kubectl create ns oreilly
$ kubectl get ns/oreilly -o yaml
$ kubectl delete ns/oreilly
```

Create a Pod in *kubecon* namespace

```
$ cat redis.yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
  namespace: oreilly
```

ResourceQuota Object

A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created, as well as the total amount of compute resources that may be consumed by resources in that project.

Create a *oreilly* ns from a file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: oreilly
```

Then create a *ResourceQuota* to limit the number of Pods

```
$ cat rq.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    pods: "1"
...
$ kubectl create -f rq.yaml --namespace=oreilly
```

Then test !

Note: Limits

In a namespace you can also define ranges for resource limits. This is achievable through a *limitRange* object.

```
$ kubectl get limits
```

Pods can define resource requests and limits. To dive deeper into limits and quotas check this walkthrough.

Replica Sets

But this is supposed to be all about microservices and scaling. How does Kubernetes scales containers ? Remember that the other Object that we saw in the Dashboard was *replication set*. A replica set ensures that a specified number of pod “replicas” are running at any one time. In other words, a replica set makes sure that a pod or homogeneous set of pods are always up and available.

Well documented

Inspect the *redis* RS:

```
$ kubectl get rs redis -o yaml
```

Try to scale it and watch.

```
$ kubectl scale rs redis --replicas=5
replicaset "redis" scaled
$ kubectl get pods --watch
```

RS Objects

Same as all Objects. Contains *apiVersion*, *kind*, *metadata*

But also a *spec* which sets the number of replicas, and the selector. An RC insures that the matching number of pods is running at all time. The *template* section is a Pod definition.

```
apiVersion: extensions/v1beta1
kind: ReplicaSEt
metadata:
  name: redis
  namespace: default
spec:
  replicas: 2
  selector:
    app: redis
  template:
    metadata:
      name: redis
      label:
        app: redis
    spec:
      containers:
      - image: redis:3.2
```

Demo: Guestbook

Let's run the Guestbook app !

- Find it on GitHub

All-in-one

- Download the yaml or json description
- Run it
- Access it...

kubectl tips and tricks

A few things to remember about *kubectl*. And if you don't, check the cheat sheet.

```
$ kubect config view
$ kubectl config use-context
$ kubectl annotate
$ kubectl label
$ kubectl create -f ./<DIR>
$ kubectl create -f <URL>
$ kubectl edit ...
$ kubectl proxy ...
$ kubectl exec ...
$ kubectl logs ...
$ kubectl get pods,svc,deployments
$ kubectl --v=99 ...
```

You can cat all your objects in one file, and *create* that file.

If things fail:

```
$ kubectl describe ...
```

Part III: Labels and Services

- Labels to select objects
- Services to expose your applications

Labels

You will have noticed that every resource can contain labels in its metadata. By default creating a deployment with `kubectl run` adds a label to the pods.

```
apiVersion: v1
kind: Pod
metadata:
...
  labels:
    pod-template-hash: "3378155678"
    run: ghost
```

You can then query by label and display labels in new columns:

```
$ kubectl get pods -l run=ghost
NAME READY STATUS RESTARTS AGE
ghost-3378155678-eq5i6 1/1 Running 0 10m
$ kubectl get pods -Lrun
NAME READY STATUS RESTARTS AGE RUN
ghost-3378155678-eq5i6 1/1 Running 0 10m ghost
nginx-3771699605-4v27e 1/1 Running 1 1h nginx
```

Labels

While you define labels in Pod templates in specifications of deployments (typically), you can also add labels on the fly:

```
$ kubectl label pods ghost-3378155678-eq5i6 foo=bar
$ kubectl get pods --show-labels
NAME READY STATUS RESTARTS AGE LABELS
ghost-3378155678-eq5i6 1/1 Running 0 11m foo=bar,pod-template-
hash=3378155678,run=ghost
```

Why use labels ?

Because they are a great way to query and select resources. For example, if you want to force the scheduling of a Pod on a specific node. you can use a `nodeSelector` in a Pod definition. Add specific labels to certain nodes in your cluster and use that labels in the Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
  nodeSelector:
    disktype: ssd
```

LUNCH TIME

Accessing *Services*

Now that we have a good handle on creating resources, managing and inspecting them with `kubectl`. The elephant in the room is how do you access your applications ?

The answer is Services, another Kubernetes object. Let's try it:

```
$ kubectl expose deployment/nginx --port=80 --type=NodePort
$ kubectl get svc
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes 10.0.0.1 <none> 443/TCP 18h
nginx 10.0.0.112 nodes 80/TCP 5s
$ kubectl get svc nginx -o yaml
```

```
apiVersion: v1
kind: Service
...
spec:
  clusterIP: 10.0.0.112
  ports:
  - nodePort: 31230
...
```

```
$ minikube ip
192.168.99.100
```

Open your browser at `http://192.168.99.100:<nodePort>`

Discuss.

Services Abstractions

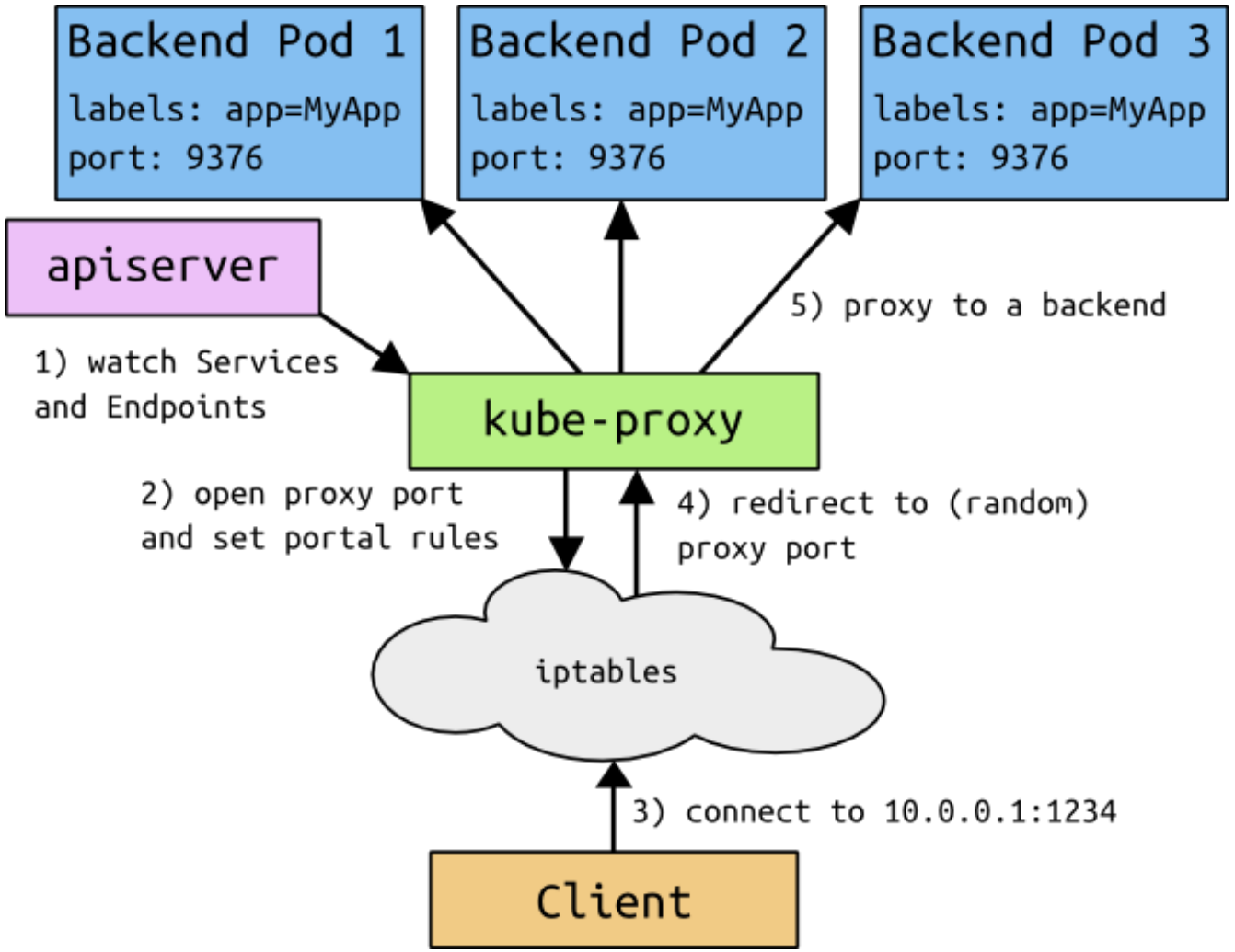
Services are abstractions that define a set of pods that provide a micro-service. They are first class citizen in Kubernetes and key to linking applications together.

They provide a stable virtual endpoint for ephemeral Pods in your cluster. Other services can target them and will be redirected to the endpoints that match the service Pod selection.

In addition you can use the Service abstraction to expose external resources into your k8s clusters (e.g bring in legacy databases), or point to another k8s cluster or namespace.

Used to be implemented in userspace, now implemented via iptables. `kube-proxy` agent running on all Kubernetes nodes, watches Kubernetes API for new Services and Endpoints being created. It opens random ports on nodes to listen to traffic to the ClusterIP:Port, and redirects to a random service endpoints (used to be round-robin in userspace implementation).

Services Diagram



Service Types

Services can be of three types:

- ClusterIP
- NodePort
- LoadBalancer

LoadBalancer services are currently only implemented on public cloud providers like GKE and AWS. Private cloud solutions also may implement this service type if there is a Cloud provider plugin for them in Kubernetes (e.g CloudStack, OpenStack)

ClusterIP service type is the default and only provides access internally (except if manually creating an external endpoint).

NodePort type is great for debugging, but you need to open your firewall on that port (NodePort range defined in Cluster configuration). Not recommended for public access.

Note that you can also run a `kubectl proxy` locally to access a ClusterIP service. Great for development.

Let's try it !

Labels Use Case: Canary Deployments

Labels, Services and Deployments are key to Canary deployments scenarios.

Typical steps:

- Create two deployments
- Add labels that differentiate the canary plus a common label
- Create a service that selects all Pods from the two deployments

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

DNS

A DNS service is provided as a Kubernetes add-on in clusters. On GKE and minikube this DNS service is provided by default. A service gets registered in DNS and DNS lookup will further direct traffic to one of the matching Pods via the ClusterIP of the service.

```
$ kubectl create -f busybox.yaml
$ kubectl exec -ti busybox -- nslookup nginx
Server: 10.0.0.10
Address 1: 10.0.0.10

Name: nginx
Address 1: 10.0.0.112
$ kubectl get svc
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes 10.0.0.1 <none> 443/TCP 19h
nginx 10.0.0.112 nodes 80/TCP 36m
$ kubectl exec -ti busybox -- wget http://nginx
Connecting to nginx (10.0.0.112:80)
index.html 100% |*******************************| 612 0:00:00 ETA
```

This DNS functionality is provided by SkyDNS and kube2sky. Services definitions are stored in the etcd of Kubernetes. kube2sky listens to those entries and feeds them to skyDNS for name resolution. Note that the etcd of skyDNS is a separate container started in the DNS add-on RC.

Towards Deployments

Deployments allow server side updates of Pods at specified rate. Used for canary and other deployments patterns. Deployments generates ReplicaSets. ReplicaSets brings set-base pods selector (e.g tier notin (frontend, backend))

Deployments are *extensions* API

```
$ curl http://127.0.0.1:8080/apis/extensions/v1beta1
{
  "kind": "APIResourceList",
  "groupVersion": "extensions/v1beta1",
  "resources": [
...
    {
      "name": "deployments",
      "namespaced": true,
      "kind": "Deployment"
    },
```

Try:

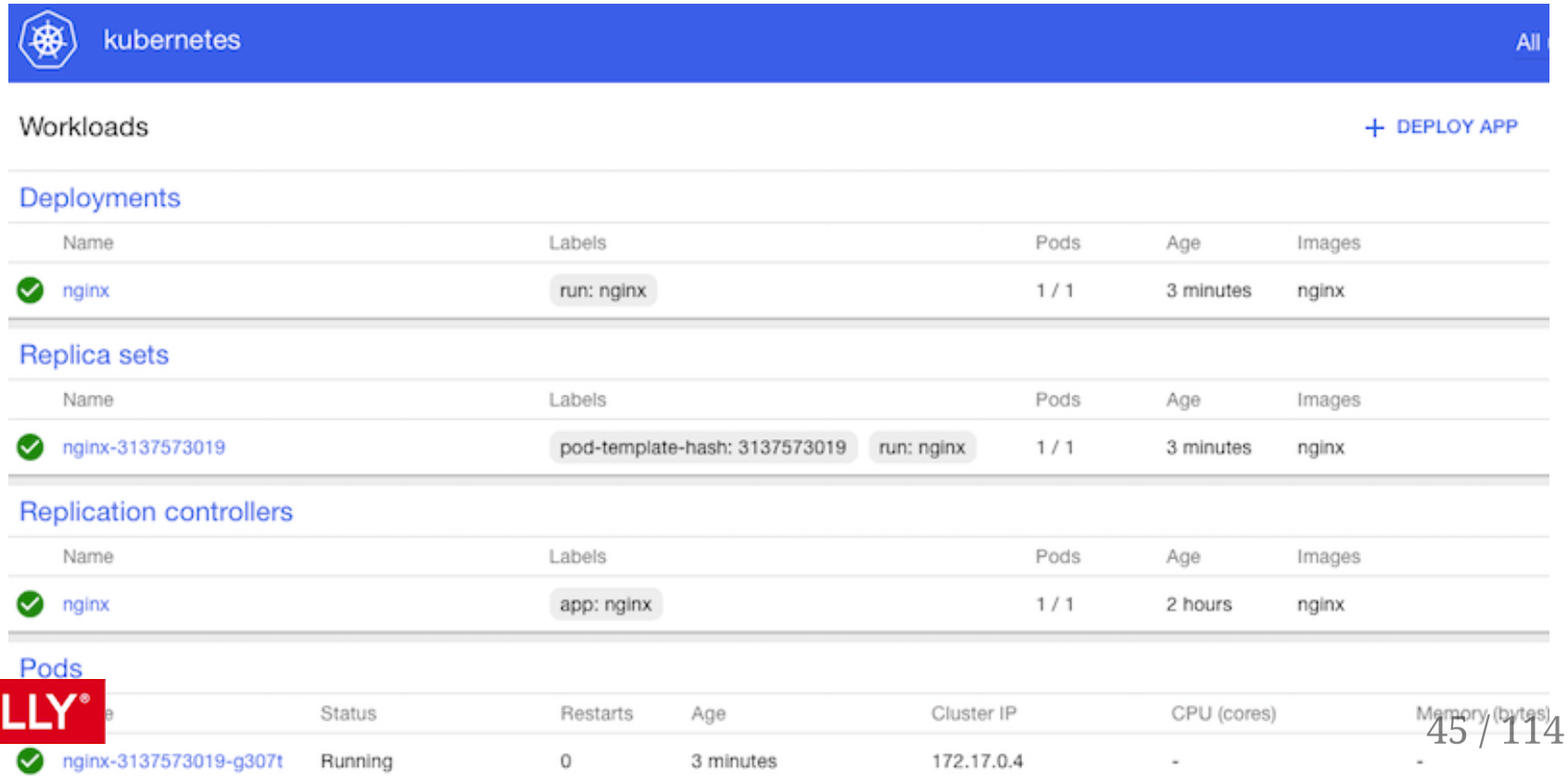
```
$ kubectl run nginx --image=nginx
```

Deployments

What does it do ?

```
$ kubectl get deployments
$ kubectl get rs
$ kubectl get pods
```

Check the dashboard



Scaling and Rolling update of Deployments

Just like RC, Deployments can be scaled.

```
$ kubectl scale deployment/nginx --replicas=4
deployment "nginx" scaled
$ kubectl get deployments
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
nginx 4 4 4 1 12m
```

What if you want to update all your Pods to a specific image version. *latest* is not a version number...

```
$ kubectl set image deployment/nginx nginx=nginx:1.10 --all
```

What the RS and the Pods.

```
$ kubectl get rs --watch
NAME DESIRED CURRENT AGE
nginx-2529595191 0 0 3m
nginx-3771699605 4 4 46s
```

You can also use `kubectl edit deployment/nginx`

Deployments Roll Back

When you create a deployment you can record your changes in an annotations

```
$ kubectl run ghost --image=ghost --record
$ kubectl get deployments ghost -o yaml
```

```
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
    kubernetes.io/change-cause: kubectl run ghost --image=ghost --record
```

Now do an update and check the status. You will see that the Pod failed. You can now roll back.

```
$ kubectl set image deployment/ghost ghost=ghost:09 --all
$ kubectl rollout history deployment/ghost
deployments "ghost":
REVISION CHANGE-CAUSE
1 kubectl run ghost --image=ghost --record
2 kubectl set image deployment/ghost ghost=ghost:09 --all
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
ghost-2141819201-tcths 0/1 ImagePullBackOff 0 1m
$ kubectl rollout undo deployment/ghost
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
ghost-3378155678-eq5i6 1/1 Running 0 7s
```

Deployments

You could roll back to a specific revision with `--to-revision=2`

You can also edit a deploy with `kubectl edit`

You can pause a deployment and resume

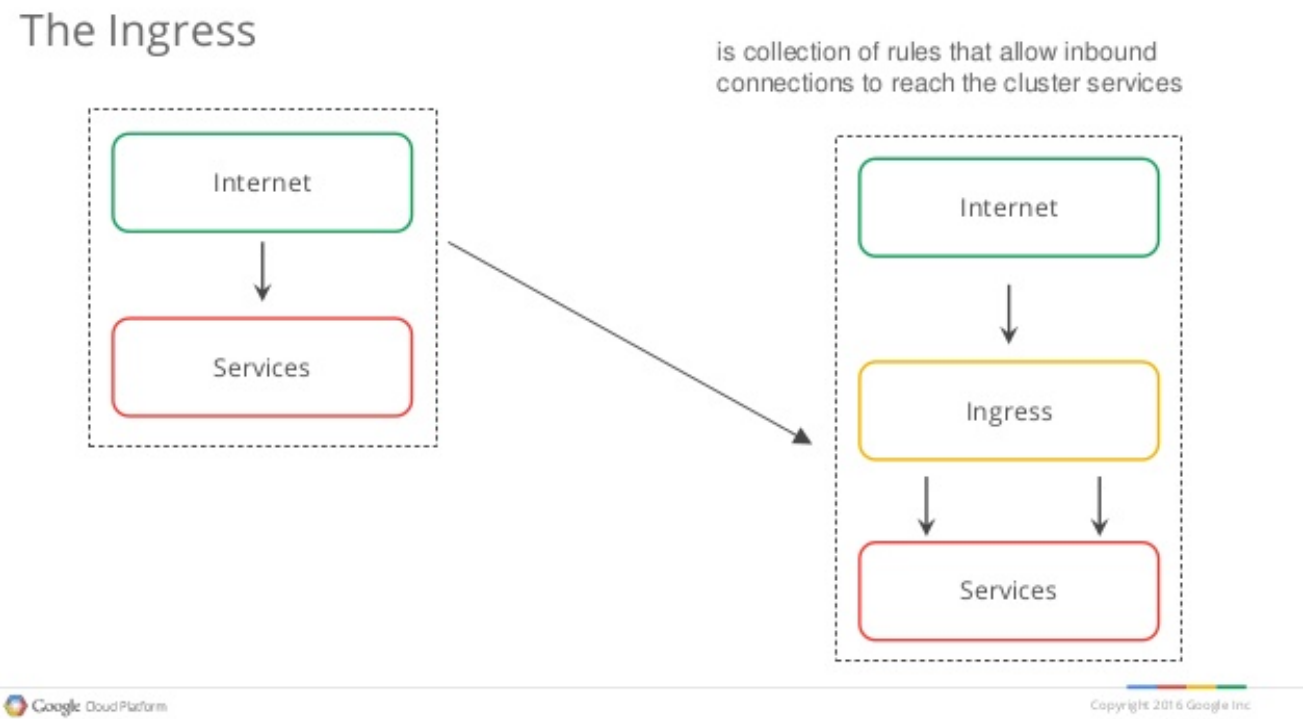
```
$ kubectl rollout pause deployment/ghost
$ kubectl rollout resume deployment/ghost
```

Note that you can still do a rolling update on replication controllers with the `kubectl rolling-update` command, but this is client side. Hence if you close your client, the rolling update will stop.

Ingress Controller

If you are not on GCE or AWS and do not have a LoadBalancer type service available, and you do not want to use a NodePort type service. How do you let inbound traffic reach your services ? **You use an Ingress Controller.**

A proxy (e.g HAproxy, nginx) that gets reconfigured based on rules that you create via the Kubernetes API.



Ingress API Resource

Ingress objects still an extension API like deployments, replicasets etc... A typical Ingress object that you can POST to the API server is:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ghost
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: ghost
          servicePort: 2368
```

Ingress Controller

You will need to have an Ingress controller for this rule to take effect. In our example, we will run an *nginx* based ingress controller, running as an RC and binding its port 80/443 to the same host ports.

Note that you could implement your own Ingress Controller.

- Deploy the Ingress controller.

```
$ cd ingress-controller
$ kubectl create -f backend.yaml
```

Ingress Exercise

- Create a Ghost deployment and Service.
- Create an Ingress rule to allow inbound traffic to your Ghost blog.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ghost
spec:
  rules:
  - host: ghost.192.168.99.100.nip.io
    http:
      paths:
      - backend:
          serviceName: ghost
          servicePort: 2368
```

Exercise

Create a deployment to run a MySQL Pod.

```
$ kubectl run mysql --image=mysql:5.5 --env=MYSQL_ROOT_PASSWORD=root
$ kubectl logs mysql-2595205605-1onu7
Initializing database
160816 17:31:28 [Note] /usr/local/mysql/bin/mysqld (mysqld 5.5.51)
starting as process 56 ...
$ kubectl exec -ti mysql-2595205605-1onu7 -- mysql -p -uroot
```

Exercise WordPress

There are several ways to do this:

- Wordpress container with embedded MySQL DB
- Wordpress Pod with two containers (i.e one wordpress, one mysql)
- Two Pods, one Wordpress, one Mysql plus a MySQL service.

All Pods should be created via a deployment.

MySQL DB needs an environment variable set. This can be done via specifying an env var directly in the spec, or using a secret.

```
    - image: mysql:5.5
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: root
```

Using Secrets

To avoid passing secrets directly in a Pod definition, Kubernetes has an API object called *secrets*. You can create, get, delete secrets. They can be used in Pod templates.

```
$ kubectl get secrets
$ kubectl create secret generic --help
$ kubectl create secret generic mysql --from-literal=password=root
```

And a Pod will look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql:5.5
    env:
    - name: MYSQL_ROOT_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysql
          key: password
    imagePullPolicy: IfNotPresent
    name: mysql
  restartPolicy: Always
```

ConfigMap

To store a configuration file made of key value pairs, or simply to store a generic file you can use a so-called config map and mount it inside a Pod

```
$ kubectl create configmap velocity --from-file=index.html
```

The mount looks like this:

```
...
spec:
  containers:
  - image: busybox
...
    volumeMounts:
    - mountPath: /velocity
      name: test
    name: busybox
  volumes:
  - name: test
    configMap:
      name: velocity
```

Everything together

Let's put it all together

- Create a namespace
- Create a Quota
- Create all objects via deployment
- Create an Ingress controller and rule
- Access it

Everything in a single file or directory.

Day 2

Morning

- Volumes, PersistentVolumes and Claims
- Custom Resource Definitions (extending Kubernetes API)
- The Kubernetes ecosystem (Helm, Python client)

Afternoon

- Security (RBAC, and network policies)
- Logging and monitoring (Fluentd and Prometheus)
- Scheduling (Node/Pode affinity and custom schedulers)
- Upgrades

A Quick Refresher

```
kubectl run ghost --image=ghost
kubectl expose deployment ghost --port 2368 --type NodePort
```

- Pods, ReplicaSets, Deployments, Services
- Volumes, PVs, PVCs, Secrets, ConfigMaps
- Namespace, Quotas
- API Server, Scheduler, Controller, Kubelet Agent with "Proxy"

Part I

- Volumes

Volumes

Look at the Mysql Pod that was started with a secret, what do you see ? How is the secret made available in the Pod.

```
spec:
  containers:
...
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-a9l9m
      readOnly: true
...
  volumes:
  - name: default-token-a9l9m
    secret:
      secretName: default-token-a9l9m
```

A Pod spec contains a *volumes* sections in addition to the *containers* sections. Then each container in the Pod can contain a *volumeMounts* key.

Volumes

In GCE or AWS you can use Volumes of type *GCEpersistenDisk* or *awsElasticBlockStore* which allows you to mount GCE and EBS disks in your Pods.

emptyDir and *hostPath* volumes are extremely easy to use (and understand). emptyDir is an empty directory that gets erased when the Pod dies (but survives container restarts), HostPath volumes survive Pod deletion.

NFS and *iSCSI* are straightforward choices for multiple readers scenarios.

Ceph, GlusterFS ...

Volumes Exercise

Create a Pod with two containers and one volumes shared. Experiment with *emptyDir* and *hostPath*

```
    containers:
  - image: busybox

    volumeMounts:
    - mountPath: /busy
      name: test
    name: busy

  - image: busybox

    volumeMounts:
    - mountPath: /box
      name: test
    name: box

  volumes:
  - name: test
    emptyDir: {}
```

Persistent Volumes and Claims

Persistent Volumes are a storage abstraction, which provides a standard volume type for Pod: Claims. You define PersistentVolumes Objects backed by an underlying storage provider. Pods mount volumes based on claims they make on the persistent storage.

```
$ kubectl get pv
$ kubectl get pvc
```

PV can be of type: NFS, iSCSI, RBD, CephF, GlusterFS, Cinder (OpenStack), HostPath for testing only.

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv0001
  labels:
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/somepath/data01"
```

Exercise

Re-write your Wordpress examples, to use PV/PVC for the Mysql volume (/var/lib/mysql)

Extending the Kubernetes API with CRD

Custom Resource Defintions

Kubernetes lets you add your own API objects. Kubernetes can create a new custom API endpoint and provide CRUD operations as well as watch API.

This is great to extend the k8s API server with your own API.

Check the Custom Resource Definition documentation

The first public use of this was at Pearson, where they used Third Party Resources to create AWS relational databases on the fly.

A more recent use case is the etcd Operator which lets you create etcd clusters using the Kubernetes API.

CRD Example

```
$ cat db.yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
name: databases.foo.bar
spec:
group: foo.bar
version: v1
scope: Namespaced
names:
plural: databases
singular: database
kind: DataBase
shortNames:
- db
```

Let's create this new resource and check that it was indeed created.

```
$ kubectl create -f database.yml
$ kubectl get customresourcedefinition
```

Custom Resources

You are now free to create a *customresource*. Just like Deployments, Pods, or Services, you need to write a manifest for it and you can use `kubectl` to create it.

```
$ cat db.yml
apiVersion: foo.bar/v1
kind: DataBase
metadata:
name: my-new-db
spec:
type: mysql
$ kubectl create -f db.yml
```

And dynamically `kubectl` is now aware of the *customresource* you created.

```
$ kubectl get db
```

And now you *just* need to write a controller.

Helm, The Kubernetes Package Manager

Helm

The package manager for Kubernetes. Open Source, created by Deis, available on GitHub.

An application is packaged in a Chart and published in a repository as a tarball (e.g HTTP server).

```
$ helm create oreilly
Creating oreilly
$ cd oreilly/
$ tree
.
├── Chart.yaml
├── charts
├── templates
└── values.yaml
```

Helm deploy Charts as releases. The templates are evaluated based on the `values.yaml` content. This results in deployments, services etc being created. *Helm* can delete a complete release, upgrade.

Should work on Windows :)

Note: Helm just graduated from the Kubernetes Incubator

Helm Example

Helm is a client that runs on your machine. You need to deploy the server side called `tiller` and use `helm` to communicate with it. Then browse Chart repositories, pick a Chart to install and create a *release*.

- Install Helm
- Deploy `tiller`
- Install application

```
$ helm init
$ helm repo list
$ helm install stable/minio
```

Showing Python Some Love Because There is no Just Golang :)

Python Client

Kubernetes now has a Python Client available via Pypi.

It is a Kubernetes incubator project.

```
$ pip install kubernetes
```

And then:

```
$ python
Python 2.7.12 (default, Oct 11 2016, 14:42:23)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import kubernetes
```

Using Python client

Instantiate a client to the API group you want to use.

```
>>> from kubernetes import client,config
>>> config.load_kube_config()
>>> v1=client.CoreV1Api()
>>> v1.list_node()
...
>>> v1.list_node().items[0].metadata.name
minikube
```

Starting a Pod in Python

No high level classes…

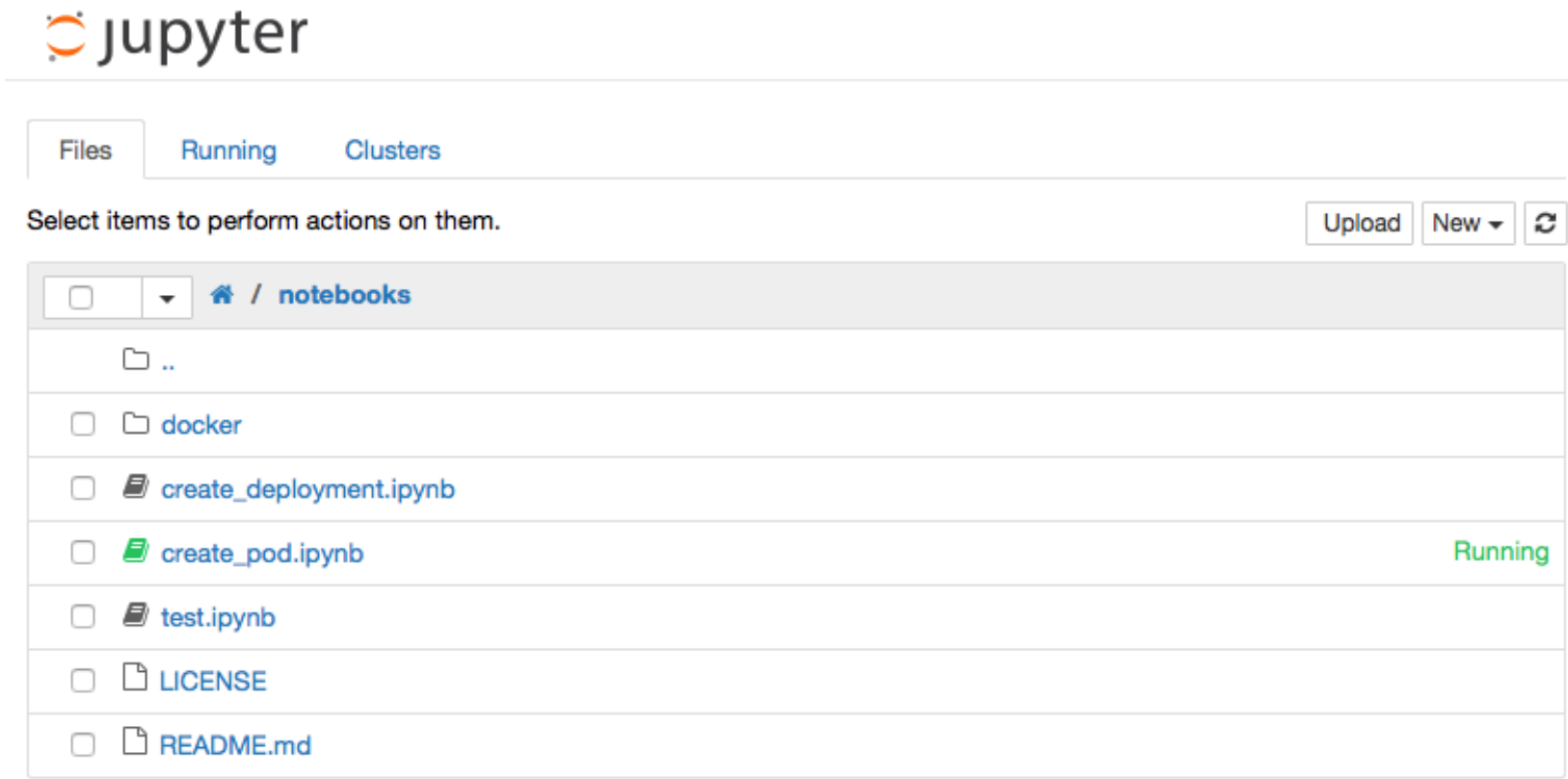
```
>>> container = client.V1Container()
>>> container.image = "busybox"
>>> container.args = ["sleep", "3600"]
>>> container.name = "busybox"
```

Using Jupyter Notebooks

Check these introductory notebooks

```
kubectl create -f docker/jupyter.yml
```

Then open *Jupyter* and start playing interactively.



LUNCH TIME

Afternoon Agenda

- Security (RBAC, and network policies)
- Logging and monitoring (Fluentd and Prometheus)
- Scheduling (Node/Pode affinity and custom schedulers)
- Upgrades

A Few Misconceptions: Namespaces

```
kubectl get pods
kubectl get pods --all-namespaces
```

Create namespaces

```
kubectl get ns
kubectl create ns foobar
kubectl run bob --image=bob
kubectl run bob --image=bob -n foobar
```

Namespaces do not give network isolations, they just give you naming isolation for resources.

A Few Misconceptions: Container Isolation

Check this https://cloud.weave.works/k8s/net?k8s-version=1.6.0

PS: Absolutely not a negative criticism.

```
  - apiVersion: extensions/v1beta1
    kind: DaemonSet
    metadata:
      name: weave-net
...
    spec:
      template:
        ...
        spec:
          containers:
            - name: weave
...
              securityContext:
                privileged: true
...
            - name: weave-npc
...
              securityContext:
                privileged: true
          hostNetwork: true
          hostPID: true
          securityContext:
            seLinuxOptions: {}
```

And Then Secrets !

```
$ kubectl create secret generic foobar --from-literal=password=root
secret "foobar" created
$ kubectl get secrets
NAME TYPE DATA AGE
foobar Opaque 1 3s
```

```
$ kubectl get secrets foobar -o yaml
apiVersion: v1
data:
password: cm9vdA==
kind: Secret
...
$ echo "cm9vdA==" | base64 -D
root
```

Answers to misconceptions

- Network Policies
- Pod Security Policies
- Secret encryption at rest in 1.7 + sealed-secrets

But first let's talk about RBAC

Life of An API Request

1/ API request is received on TLS secured API Server

2/ Authentication (e.g certs, tokens, basic auth, web hooks, openid)

3/ Authorization (e.g ABAC, now RBAC)

4/ Admission Control (e.g PSP) -- let's hear from Stefan --

5/ Do your thing

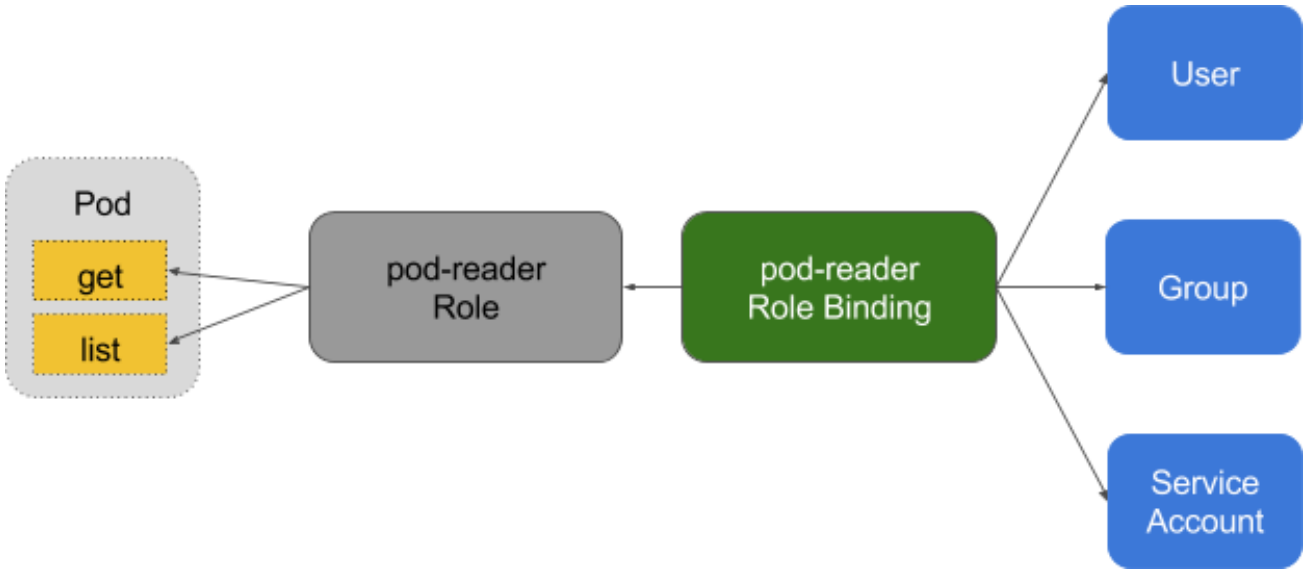
Role Based Access Control (RBAC)

For a while every user had the same privileges in a default setup.

Since 1.6.0, in `kubeadm` RBAC is the default authorization process.

Check Bitnami's docs https://docs.bitnami.com/kubernetes/how-to/configure-rbac-in-your-kubernetes-cluster/

Or the official doc of course
https://kubernetes.io/docs/admin/authorization/rbac/



concepts in RBAC: Role and Role Binding.

RBAC is 100% API Driven

New Beta API resource

```
curl localhost:8080//apis/rbac.authorization.k8s.io/v1beta1
{
"kind": "APIResourceList",
"apiVersion": "v1",
"groupVersion": "rbac.authorization.k8s.io/v1beta1",
"resources": [
{
"name": "clusterrolebindings",
"singularName": "",
"namespaced": false,
"kind": "ClusterRoleBinding",
"verbs": [
"create",
"delete",
"deletecollection",
"get",
"list",
"patch",
"update",
"watch"
]
...
```

Which means that you control your RBAC configurations via the API, using rnetes manifests...

Role and Binding

A Role

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
namespace: default
name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
resources: ["pods"]
verbs: ["get", "watch", "list"]
```

And a binding:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
name: read-pods
namespace: default
subjects:
- kind: User
name: jane
apiGroup: rbac.authorization.k8s.io
roleRef:
kind: Role
name: pod-reader
apiGroup: rbac.authorization.k8s.io
```

Test it on Minikube

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

- Create a user (i.e create a user certificate using the CA of minikube)

```
$ openssl genrsa -out employee.key 2048
$ openssl req -new -key employee.key -out employee.csr -subj
"/CN=employee/O=bitnami"
```

Locate CA, replace CA_LOCATION with it (`/etc/kubernetes/pki`, `~/.minikube`)

```
$ openssl x509 -req -in employee.csr -CA CA_LOCATION/ca.crt -CAkey
CA_LOCATION/ca.key -CAcreateserial -out employee.crt -days 500
```

Create a user with a context:

```
$ kubectl config set-credentials employee --client-
certificate=/home/employee/.certs/employee.crt --client-
key=/home/employee/.certs/employee.key
$ kubectl config set-context employee-context --cluster=minikube --
namespace=office --user=employee
```

Test it

- Set a `kubectl` context

```
kubectl --context=fanboy get pods
Error from server (Forbidden): User "system:anonymous" cannot list pods in
the namespace "meetup". (get pods)
```

Create the Role and binding

Then check its impact

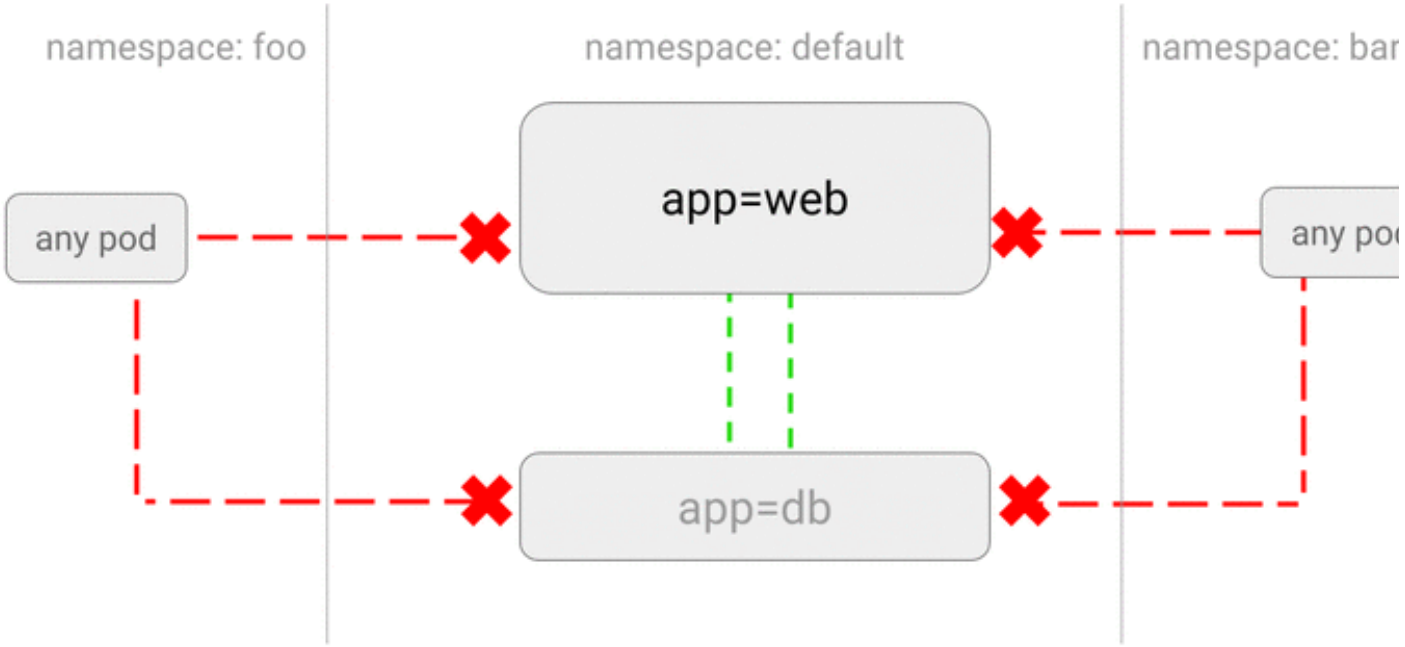
```
$ kubectl --context=fanboy get pods
No resources found.
$ kubectl --context=fanboy -n default get pods
Error from server (Forbidden): User "fanboy" cannot list pods in the
namespace "default". (get pods)
sebgoa@foobar meetup $
```

Network Policies

You need a Networking add-on that has a network policy controller.

Check Ahmet's tutorial https://ahmet.im/blog/kubernetes-network-policy/

And his repo https://github.com/ahmetb/kubernetes-networkpolicy-tutorial



Deny All

Do not write spec.ingress

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
      env: prod
```

Network Policy Example (thanks Ahmet)

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: bookstore
      role: api
  ingress:
  - from:
      - podSelector:
          matchLabels:
            app: bookstore
  - from:
      - podSelector:
          matchLabels:
            app: inventory
```

Pod Security Policies

Check Bitnami's docs https://docs.bitnami.com/kubernetes/how-to/secure-kubernetes-cluster-psp/

```
kubectl get psp
```

Write your own policies, check OpenShit for instance.

Below, privileged containers not allowed. Containers cannot run processes as `root`.

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  fsGroup:
    rule: RunAsAny
  runAsUser:
    rule: MustRunAsNonRoot
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - '*'
```

Testing PSP in minikube

You need to configure the admission controller

```
minikube start --extra-
config=apiserver.GenericServerRunOptions.AdmissionControl=NamespaceLifecycle,Li
```

Sealed Secrets

Problem: "I can manage all my K8s config in git, except Secrets."

Solution: Encrypt your Secret into a SealedSecret, which is safe to store - even to a public repository. The SealedSecret can be decrypted only by the controller running in the target cluster and nobody else (not even the original author) is able to obtain the original Secret from the SealedSecret.

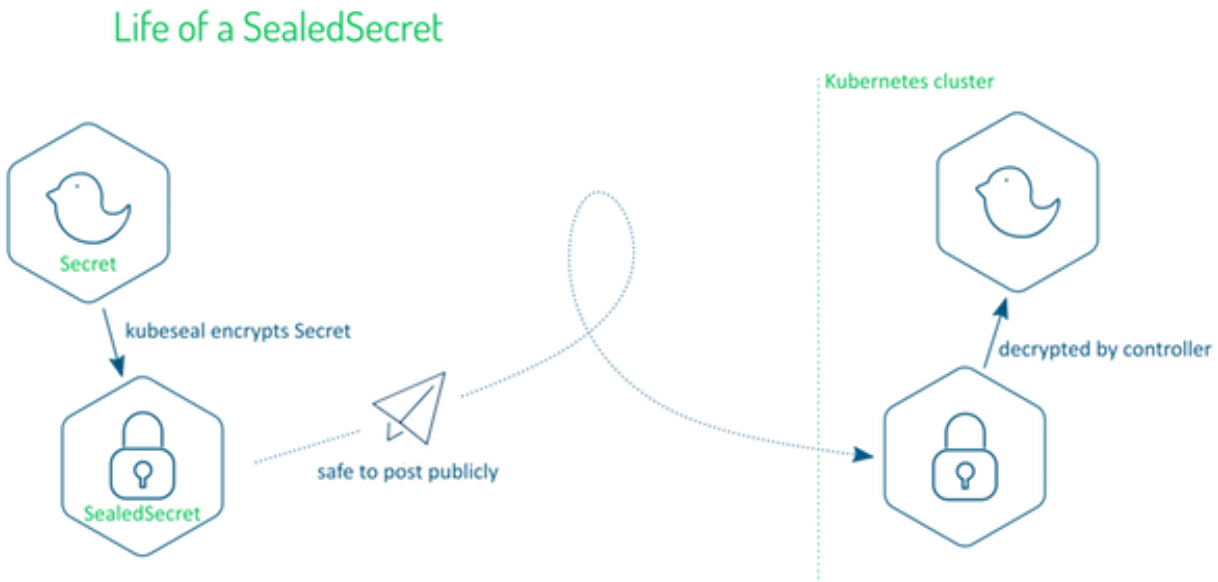
Open Source

- https://github.com/bitnami/sealed-secrets

Sealed Secrets

- Kubernetes extension
- TPR (CRD) for `SealedSecrets`
- A controller

Blog from Gus: https://engineering.bitnami.com/articles/sealed-secrets.html



How does it work

- Controller starts
- Generates a certificate
- `kubeseal` cli retrieves the public cert to encrypt
- Seal secret and create `SealedSecret` object
- Store `SealedSecret` in git
- `kubectl apply SealedSecret`
- Controller decrypts and creates corresponding `Secret` object in-cluster

Create Secret

Use `kubectl` and the `--dry-run` option to generate a JSON manifest of a secret.

```
$ kubectl create secret generic mysecret --dry-run --from-literal=foo=bar -o json
$ more mysecret.json
{
    "kind": "Secret",
    "apiVersion": "v1",
    "metadata": {
        "name": "mysecret",
        "creationTimestamp": null
    },
    "data": {
        "foo": "YmFy"
    }
}
```

Generate Sealed Secret

Download and use the `kubeseal` CLI

```
$ kubeseal < mysecret.json >mysealedsecret.json
$ more mysealedsecret.json
{
  "kind": "SealedSecret",
  "apiVersion": "bitnami.com/v1alpha1",
  "metadata": {
    "name": "mysecret",
    "namespace": "default",
    "creationTimestamp": null
  },
  "spec": {
    "data": "AgBAqtda+GOTFbjmrrsg8A8ojKvX0msQBaW99WHkcsabFiN6RR5rLI7WDBdNvz78Q3r+
```

Scheduling Policies

The default scheduler contains default predicates and priorities, however this can be changed via a scheduler policy file. This file is formatted in JSON and lists the policies that you want the scheduler to apply. You can check an example policy file. A short version is shown below, it uses a single filter and a single priority function. It has the effect of spreading the pods across nodes that can fit its resources.

```
{
"kind" : "Policy",
"apiVersion" : "v1",
"predicates" : [
{"name" : "PodFitsResources"},
],
"priorities" : [
{"name" : "LeastRequestedPriority", "weight" : 1}
]
}
```

Typically you will configure a scheduler with this policy using the `--policy-config-file` parameter and define a name for this scheduler using the `--scheduler-name` parameter. You will then have two schedulers running and will be able to specify which scheduler to use in the Pod specification.

Scheduling via Pod Specification

In most cases, the default scheduler will be suitable for your needs and you will not need to worry about setting up a customer scheduler with different policies. You will be able to influence the scheduling through the Pod specification.

A Pod specification contains several field that informs scheduling, namely:

- `nodeName`
- `nodeSelector`
- `affinity`
- `schedulerName`
- `Tolerations`

Dive into the details of the specification via the API reference docs.

And check the scheduling documentation.

Specifying a Node via `nodeName` or `nodeSelector`

The `nodeName` and `nodeSelector` field in a Pod specification provide a straightforward way to target a node or a set of nodes.

The `nodeName` tells the scheduler to place the Pod on a node with the specific name, while the `nodeSelector` tells the scheduler to place the Pod on a node that matches the labels defined under the `nodeSelector` field.

For instance, say that you want a Pod to be placed on a node with label `foo=bar` you would write the following specification (e.g for a Pod running `redis`).

```
    apiVersion: v1
    kind: Pod
    metadata:
    name: foobar
    spec:
    containers:
    - name: redis
    image: redis
    nodeSelector:
    foo: bar
```

The Pod would remain pending until a node is found with the matching labels. You can test this on `minikube`. After seeing the Pod remaining in *Pending* state you can label the `minikube` node with `foo=bar` and the Pod will start running.

Affinity Rules

A more advanced way to control the scheduling via a Pod specification is to use *Affinity* rules.

Affinity rules are beta since Kubernetes v1.6. There are two types of affinit rules: node affinity rules and pod affinity rules. Node affinity refers to more advanced way to express node preferences for a pod, while Pod affinity referes to advanced ways to express the placement of Pods in relation to other Pods.

Rules can be "soft" or "hard" thereby expressing a preference or a requirement.

Below is an example Pod specification that expresses hard requirement to place the Pod on a node with label `foo=bar`:

```
apiVersion: v1
kind: Pod
metadata:
name: ghost
spec:
affinity:
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: foo
```

Pod Affinity/Anti-affinity

Similarly to Node affinity rules, a Pod specification can contain *podAffinity* or *podAntiAffinity* rules. These rules inform the scheduler on how to place a Pod relative to other Pods.

A typical example is if you want two pods to be co-located on the same node. You would write a *podAffinity* rule. Typical example:

```
    apiVersion: v1
    kind: Pod
    metadata:
    name: ghost
    spec:
    affinity:
    podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
    matchExpressions:
    - key: app
    operator: In
    values:
    - frontend
    topologyKey: failure-domain.beta.kubernetes.io/zone
    containers:
    - name: ghost
    image: gchost:0.9
```


The rule below will make sure the *ghost* Pod is scheduled on the same node

Taints and Tolerations

While node affinity rules, allow Pods to be scheduled on specific nodes. Taints and tolerations allow you to do the opposite. A node with a particular *taint* will repel Pods that do not tolerate that taint.

This mechanism is a way to keep Pods off of a set of nodes. For example you could taint the Kubernetes master nodes to make sure only pods with label `foo=bar` can run on them. To do so, you would *taint* the node and add a *tolerations* field in the Pod specification. The taint would be applied like so:

```
kubectl taint node master foo=bar:NoSchedule
```

And the Pod specification would contain a *toleration*:

```
tolerations:
- key: "foo"
operator: "Equal"
value: "bar"
effect: "NoSchedule"
```

This feature is beta as of Kubernetes v1.6, check the full Documentation for more examples and use cases.

Node Selector

Create a Pod which uses a `nodeSelector`:

```
apiVersion: v1
kind: Pod
metadata:
  name: foobar
spec:
  containers:


   • image: redis
     name: redis
     nodeSelector:
     foo: bar
```

Label `minikube`

```
kubectl label node minikube foo=bar
```

Custom Scheduler

Create a Pod which uses a custom scheduler:

```
apiVersion: v1
kind: Pod
metadata:
  name: foobar
spec:
  containers:
  - image: redis
    name: redis
  schedulerName: foobar
```

And now create a _binding_ to place this pod on a a node

```
{
  "apiVersion": "v1",
  "kind": "Binding",
  "metadata": {
    "name": "foobar-sched"
  },
  "target": {
    "apiVersion": "v1",
    "kind": "Node",
    "name": "minikube"
  }
}
```

```
curl -H "Content-Type:application/json" -X POST --data @binding.json
http://localhost:8080/api/v1/namespaces/default/pods/foobar-sched/binding/
```

Monitoring

Deploy `prometheus` with:

```
kubectl apply -f monitoring-namespace.yaml
kubelet apply -f prometheus-rbac.yaml
kubelet apply -f prometheus-config.yaml
kubelet apply -f prometheus-statefulset.yaml
kubelet apply -f prometheus-svc.yaml
kubelet apply -f node-exporter-daemonset.yaml
kubelet apply -f node-exporter-svc.yaml
```

Deploy `grafana` with:

```
kubelet apply -f grafana-statefulset.yaml # (1)
kubelet apply -f grafana-svc.yaml # (2)
```

Logging

Fluentd is a log aggregator part of CNCF

Fluentd deployment

```
kubectl label node minikube beta.kubernetes.io/fluentd-ds-ready=true

kubectl create -f fluentd-es-configmap.yaml
kubectl create -f fluentd-es-ds.yaml
```

Elasticsearch Deployment

```
kubectl create -f es-statefulset.yaml
kubectl create -f es-service.yaml
```

Kibana Deployment

```
kubectl create -f kibana-deployment.yaml
kubectl create -f kibana-service.yaml
```

Upgrade

As usual, you should read upgrading kubernetes upstream documentation, in particular noting specific upgrading requirements that each release-step may need.

And read the release notes :)

Upgrade Steps

The upgrade needs to tackle *three* main core k8s type of services:

1. Kubernetes state, typically running alongside in master nodes::

   a. `etcd` upgrade see https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/

2. Kubernetes control plane, running on master nodes:

   a. `kube-apiserver`

   b. `kube-controller-manager`

   c. `kube-scheduler`

   Upgrade control plane before the nodes

3. Kubernetes nodes services, running on worker nodes:

   a. `kubelet`

   b. `kube-proxy`

Drain nodes

Don't forget to drain your nodes and hopefully you started your Pods via deployment...Right ?

```
$ kubectl drain node node-1
$ kubectl cordon --help
$ kubectl uncordon --help
```

Thank You

Stay in touch **@sebgoa**

File issues on https://github.com/sebgoa/oreilly-kubernetes

I hope you enjoyed this crash training.

And Enjoy Kubernetes