

Online Kubernetes Training - O'REILLY

By Sebastien Goasguen, author of the Docker Cookbook and co-author of the Kubernetes cookbook.

Independent / Innovator. Currently available for private trainings !

@sebgoa

O'REILLY®



Kubernetes
Cookbook

BUILDING CLOUD NATIVE APPLICATIONS

O'REILLY®

Sébastien Goasguen

Pre-requisites

- minikube , <https://github.com/kubernetes/minikube>
- **or** Dockert for Desktop with kubernetes enabled
- kubectl , <https://kubernetes.io/docs/user-guide/prereqs/>

Manifests here:

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

```
git clone https://github.com/sebgoa/oreilly-kubernetes
cd oreilly-kubernetes/manifests
```

Kubernetes Training

- Two days, three parts each.
- We break every 50 minutes.
- 10 minutes break.

Day 1

- Introduction and Installation
- Getting started with `kubectl` (Pods and ReplicaSets)
- Labels and Services

Day 2

- Deployments and Ingress
- Configuration with ConfigMaps, Secrets and Volumes
- Helm, Custom Resource Definition and Python client.

Kubernetes Training

Goal: Getting to know Kubernetes

- Introduction to Kubernetes
- Get our hands dirty using `minikube`
- Basics of the API, API objects and usage with `kubectl`
- Using `kubectl` to manage a distributed applications

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).

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

 Research at Google

[Home](#) [Publications](#) [People](#) [Teams](#) [Outreach](#) [Blog](#) [Work at Google](#)

Large-scale cluster management at Google with Borg

Venue

Proceedings of the European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France (2015)

Publication Year

2015

Authors

Abhishek Verma, Luis Pedrosa, [Madhukar R. Korupolu](#), David Oppenheimer, Eric Tune, John Wilkes

BibTeX

Abstract

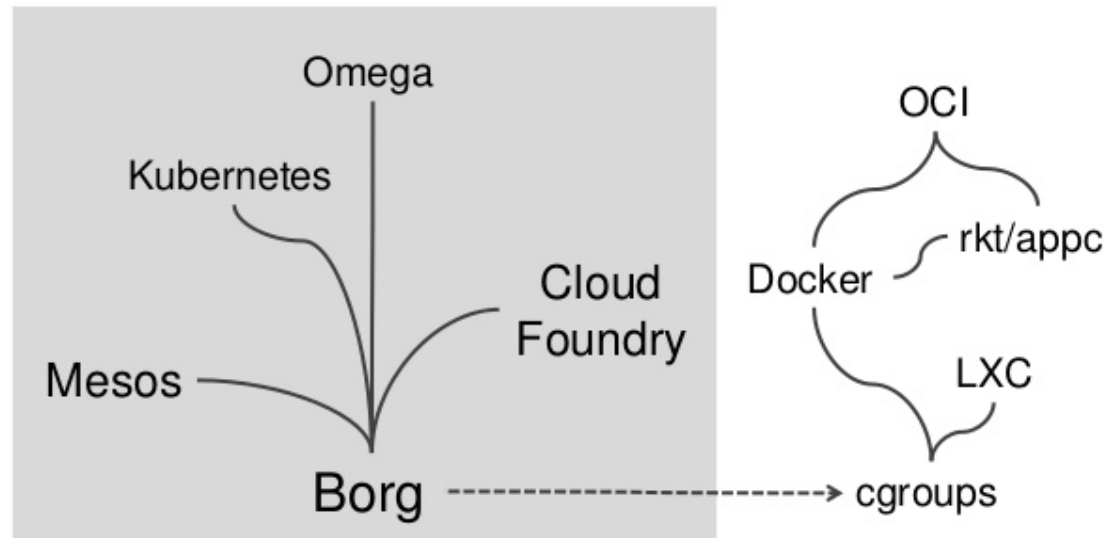
Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines. It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.



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.

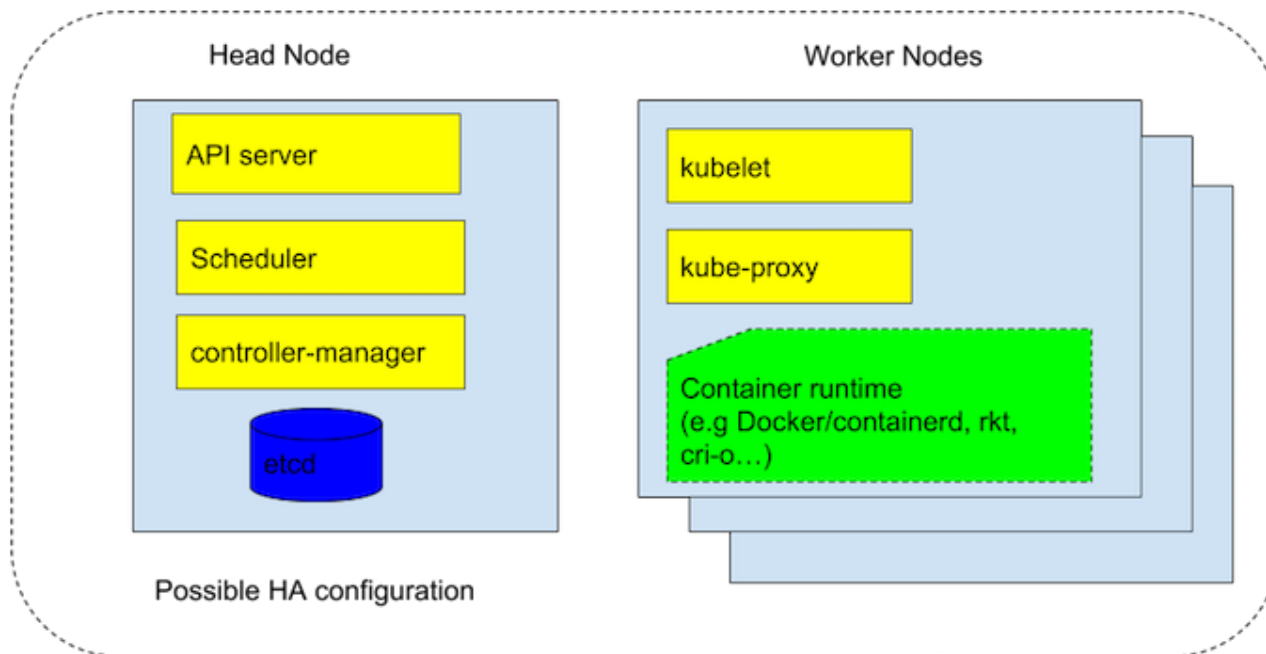


CLOUD FOUNDRY FOUNDATION

What is it really ?

- A resource manager with lots of HA features
- A scheduler to place containers in a cluster
- Deployed as services on VMs or Bare-metal machines

Kubernetes Cluster



How is it doing ?

- Open Source in June 2014 (4 years old)
- 1500 + contributors
- 60k commits
- Second Biggest Golang project on GitHub (Docker #1)
- Google and Red Hat lead contributors
- Meetups in +100 cities worldwide
- +30,000 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 clusters create oreilly
```

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

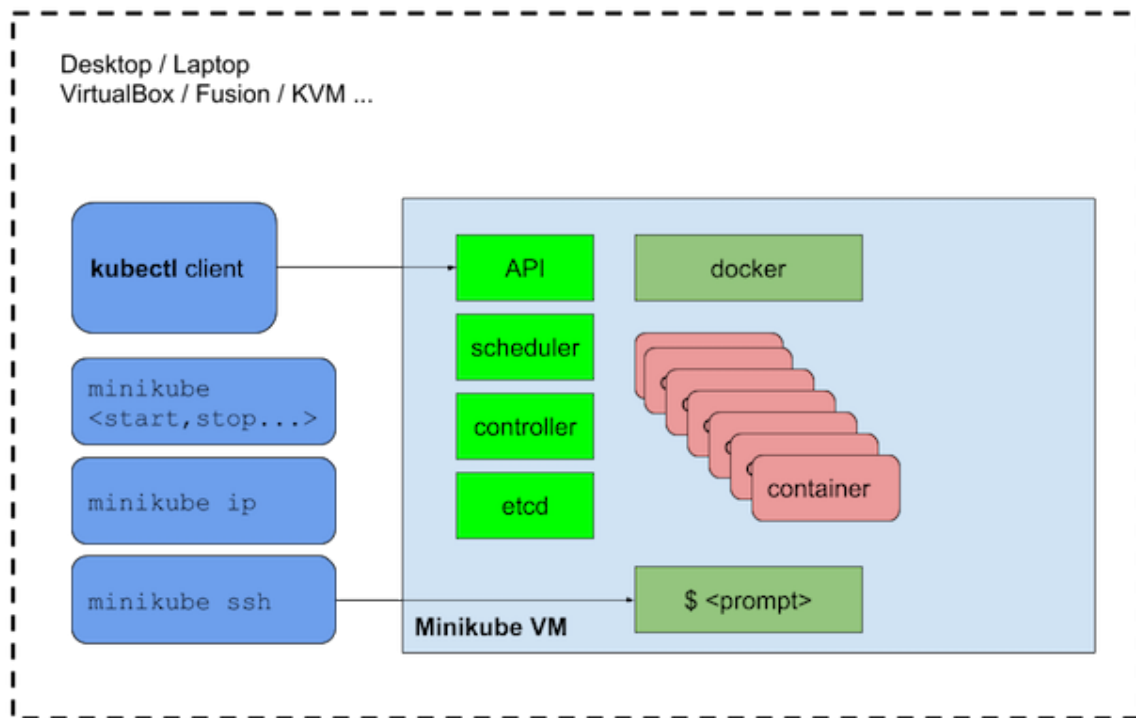
- Minikube

```
$ minikube start
```

Install minikube locally and discover Kubernetes on your local machine

Minikube

[Minikube](#) is open source and available on GitHub.



Minikube Installation/Start

Install the latest [release](#). e.g on OSX:

```
$ curl -Lo minikube  
https://storage.googleapis.com/minikube/releases/v0.23.0/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 minikube 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 in [Safari](#)

[Documentation](#)

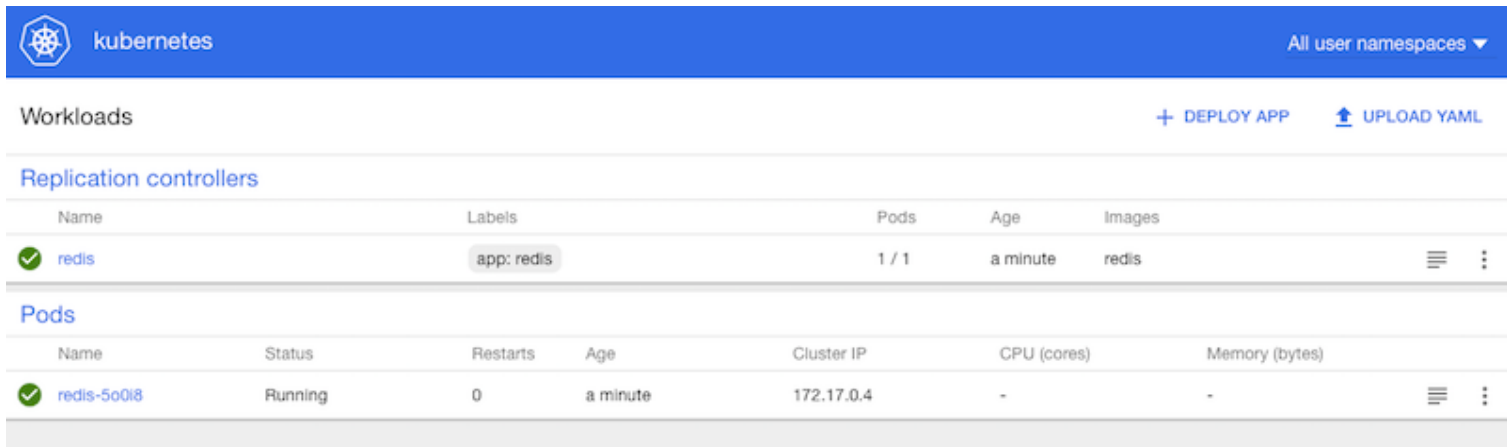
Part II: The API and kubectl

- kubectl
- Introduction to key primitives
- The Kubernetes API

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 ?



The screenshot shows the Kubernetes Dashboard interface. At the top, there's a blue header with the Kubernetes logo and the word "kubernetes". On the right, it says "All user namespaces ▼". Below the header, there's a section for "Workloads" with two buttons: "+ DEPLOY APP" and "UPLOAD YAML". Under "Workloads", there's a subsection for "Replication controllers". It contains a table with one entry: "redis". The table has columns for Name, Labels, Pods, Age, and Images. The "redis" entry shows a green checkmark, the label "app: redis", 1 / 1 pods, an age of "a minute", and the image "redis". Below this, there's a subsection for "Pods". It contains a table with one entry: "redis-5o0i8". The table has columns for Name, Status, Restarts, Age, Cluster IP, CPU (cores), and Memory (bytes). The "redis-5o0i8" entry shows a green checkmark, status "Running", 0 restarts, an age of "a minute", cluster IP "172.17.0.4", and dashes for CPU and memory.

Name	Labels	Pods	Age	Images
✓ redis	app: redis	1 / 1	a minute	redis

Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
✓ redis-5o0i8	Running	0	a minute	172.17.0.4	-	-

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/v1.8.1/bin/darwin/amd64/kubectl  
$ wget https://storage.googleapis.com/kubernetes-  
release/release/v1.8.1/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>
```

Explore `kubectl` commands

```
$ kubectl explain pods  
$ kubectl explain pods.metadata  
$ kubectl describe pods redis-7f5f77dc44-mpjvc
```

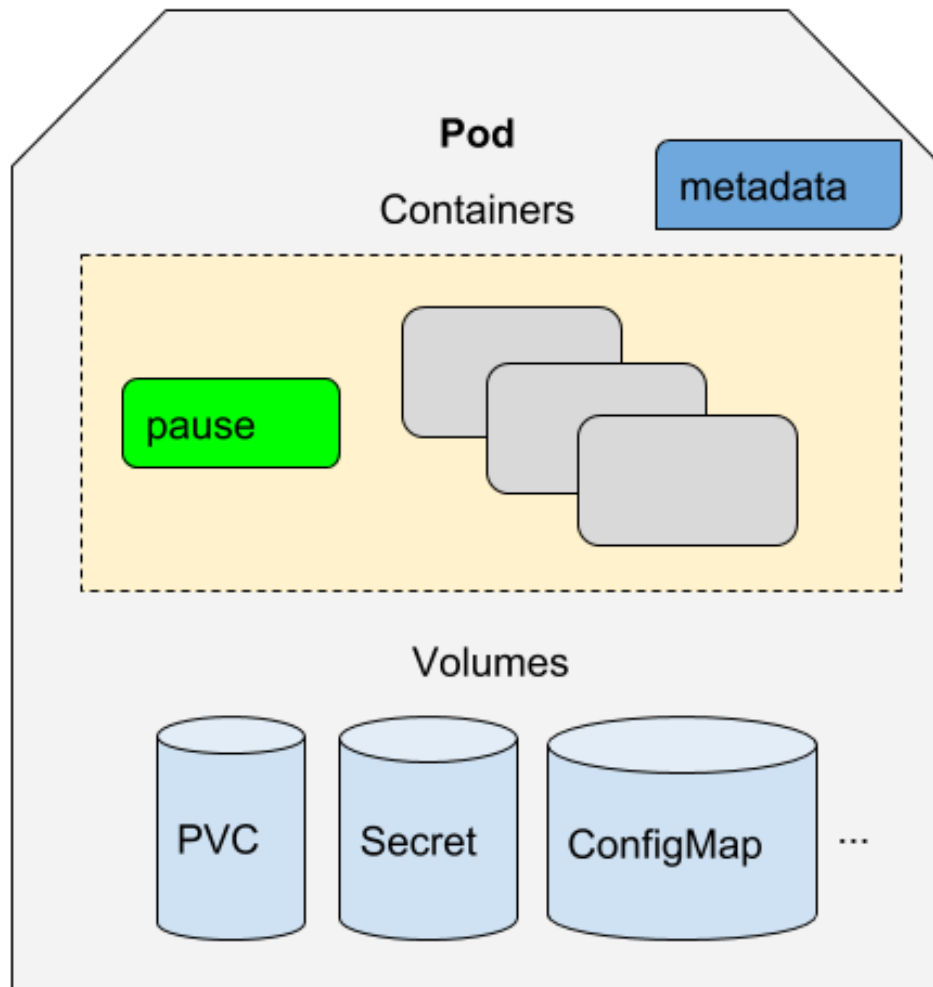
This object is stored in etcd and represented by a manifest:

```
$ kubectl get pods redis-7f5f77dc44-mpjvc -o <json,yaml>
```

Check the manifest, explore it using `explain` and cross-reference with the [documentation](#)

A Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: oreilly
spec:
  containers:
  - name: ghost
    image: ghost
  volumes:
  - name: test
    hostPath:
      path: /tmp
```



Become Friends with Pods

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

Simplest form, single container

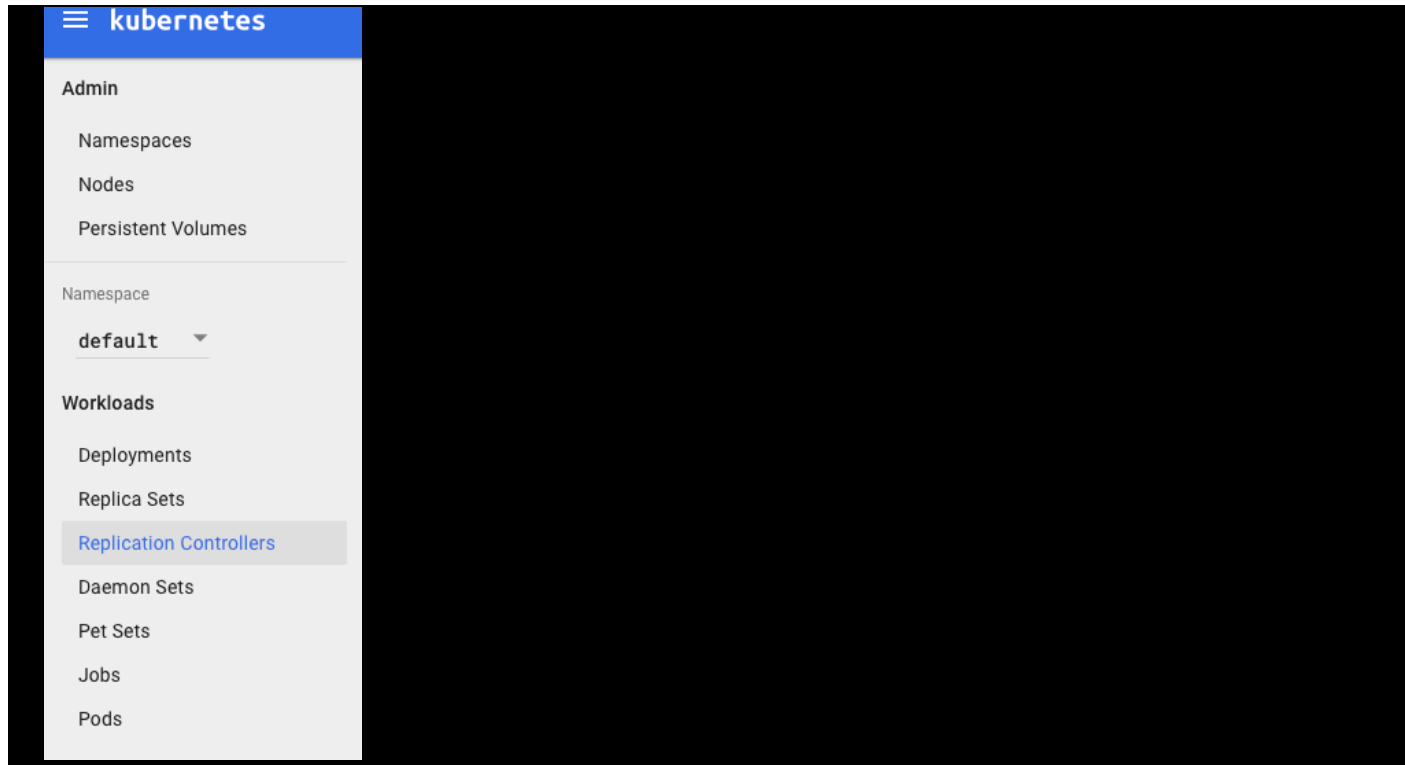
```
apiVersion: v1
kind: Pod
metadata:
  name: foobar
spec:
  containers:
  - name: ghost
    image: ghost
```

And launch it with:

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

What k8s Objects Do We See in the Dashboard ?

Check the resources listing on the left gutter of the dashboard.



Check API Resources with kubectl

Check it with kubectl:

```
$ kubectl get pods
$ kubectl get rc
$ kubectl get ns
```

But there is much more ... explore it old style !

```
$ kubectl proxy &
$ curl http://127.0.0.1:8001
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    ...
  ]
}
$ curl http://127.0.0.1:8001/api
```

Powerful REST based API

YAML or JSON definitions for objects

```
$ kubectl --v=9 get pods
...
I0816 11:20:40.722829 16899 round_tripper.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_tripper.go:292] Response Headers:
I0816 11:20:40.722880 16899 round_tripper.go:295] Content-Type:
application/json
I0816 11:20:40.722891 16899 round_tripper.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
```

Namespaces

Every request is namespaced e.g GET

`https://192.168.99.100:8443/api/v1/namespaces/default/pods`

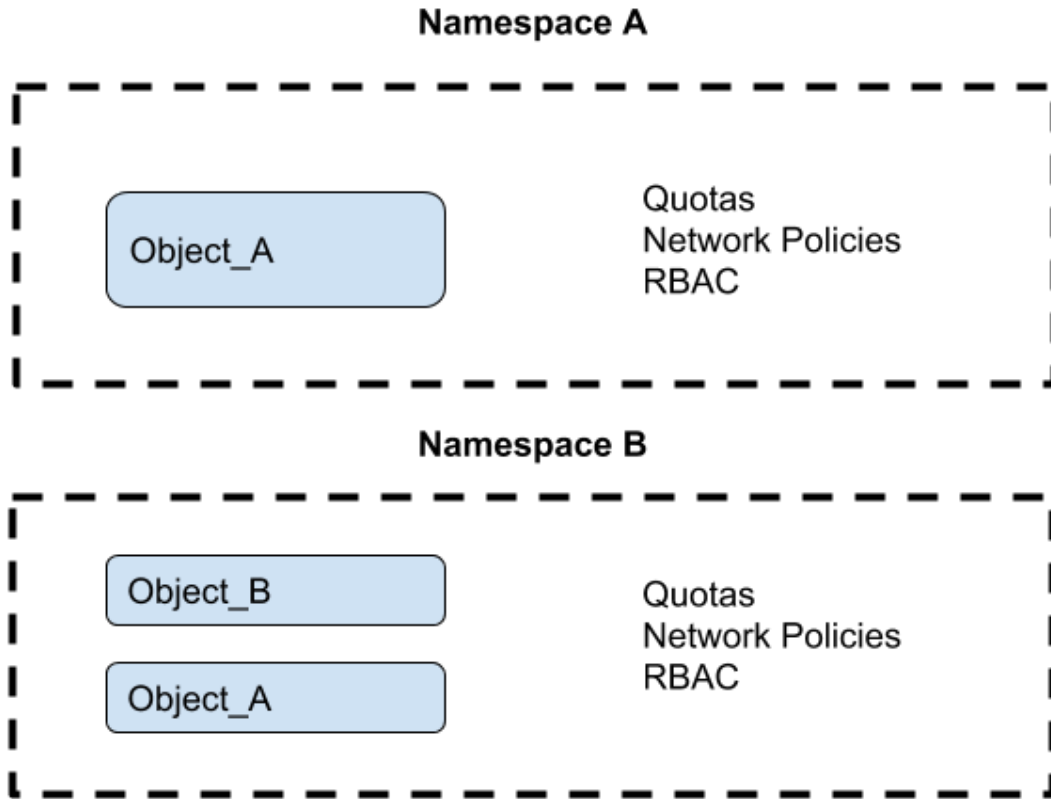
Namespaces are intended to isolate multiple groups/teams and give them access to a set of resources. Each namespace can have quotas. In future releases, we will have RBAC policies in namespaces (i.e define read/write access for users within/between namespaces)

```
$ 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
```

Namespaces



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 !

Pods

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

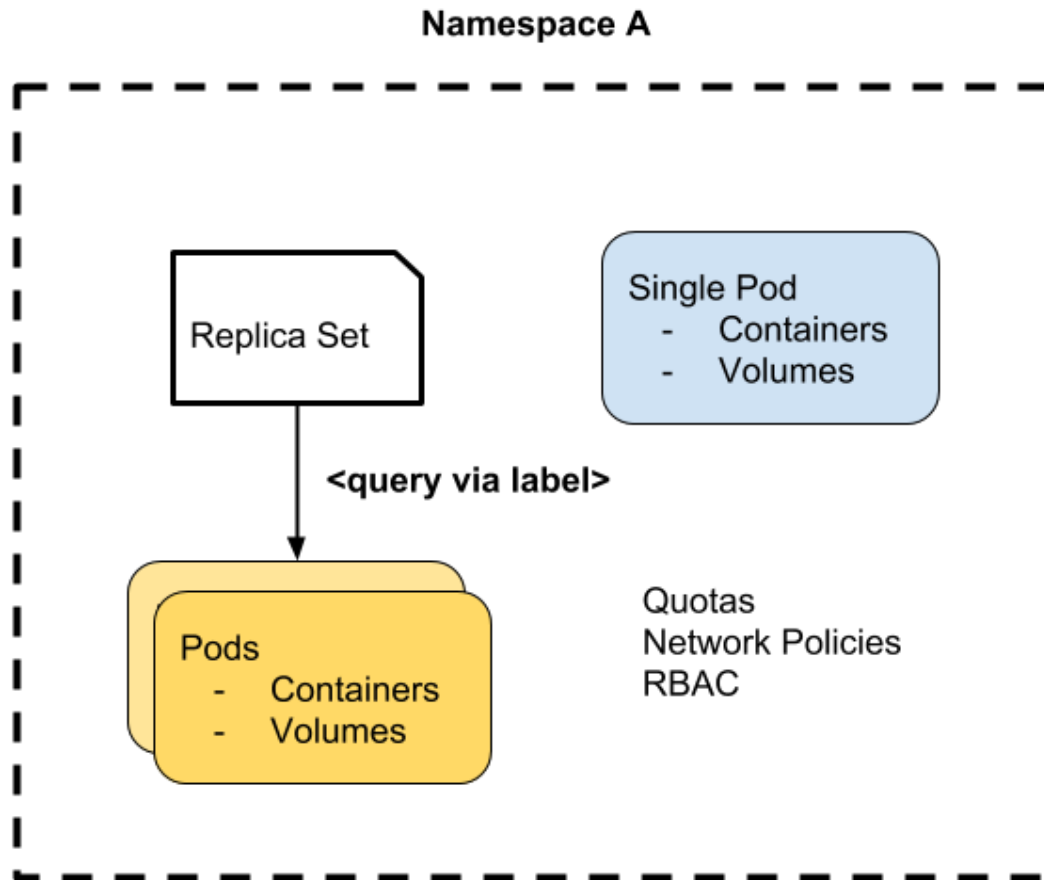
Full specifications well [documented](#). See also the [user-guide](#)

```
$ cat redis.yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - image: redis:3.2
    name: redis

$ kubectl create -f redis.yaml
```

What happens if you delete this Pod ?

Replica Set



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 *replica 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
```

Now write a RS from scratch !

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
    spec:
      containers:
      - image: redis:3.2
```

kubectl tips and tricks

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

```
$ kubectl 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=9 ...
```

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
```

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>`

Services Abstractions

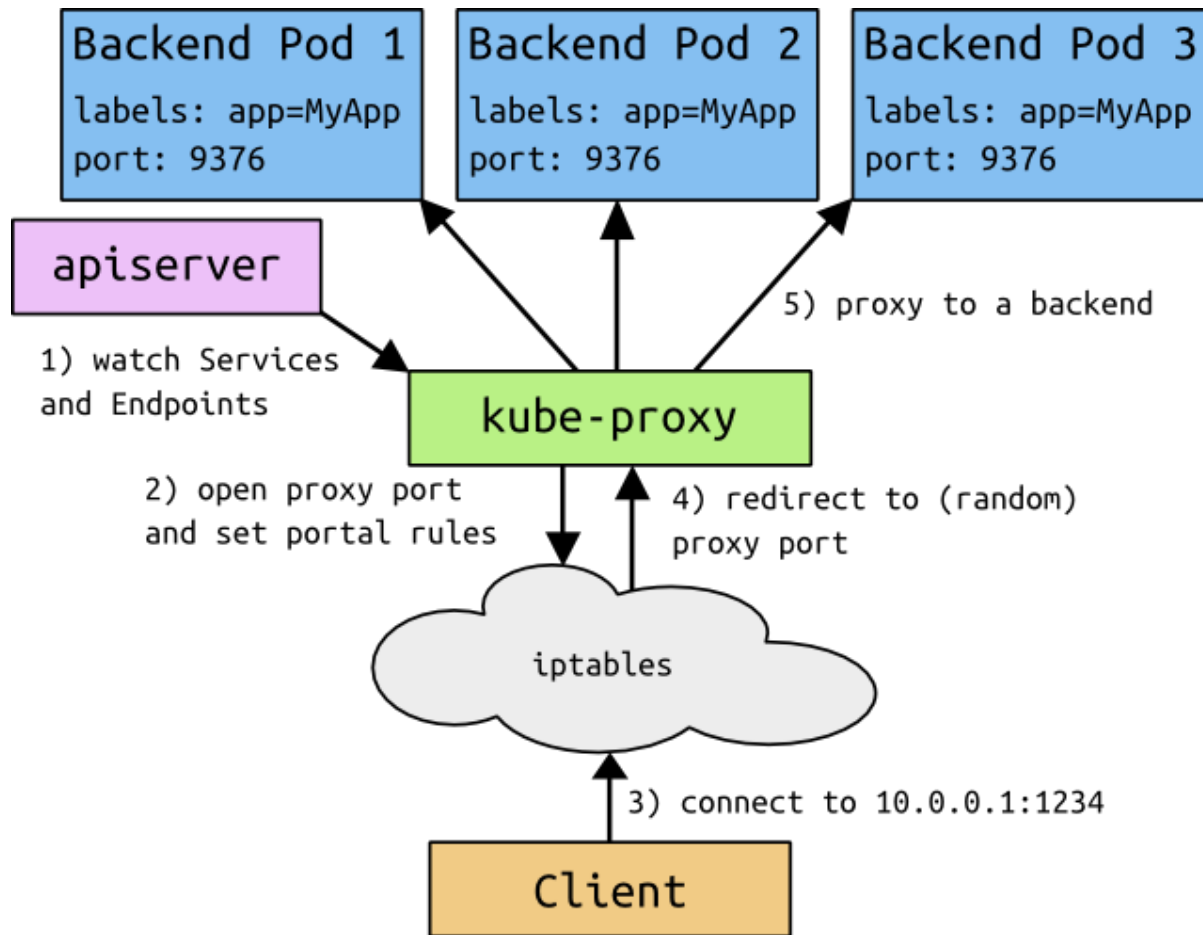
[Services](#) are abstractions that define a set of pods that provide a micro-service. They are first class citizen in Kubernetes and key to linking applications together.

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 !

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.

Day 2

- Part I: Deployments, Ingress
- Part II: Secrets, ConfigMaps, Volumes
- Part III: Helm, Custom Resource Definitions and Python client

Towards Deployments

Deployments are *apps* API

```
$ curl http://127.0.0.1:8080/apis/apps/v1
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "apps/v1",
  "resources": [
    ...
    {
      "name": "deployments",
      "namespaced": true,
      "kind": "Deployment"
    },
  ],
}
```

Try:

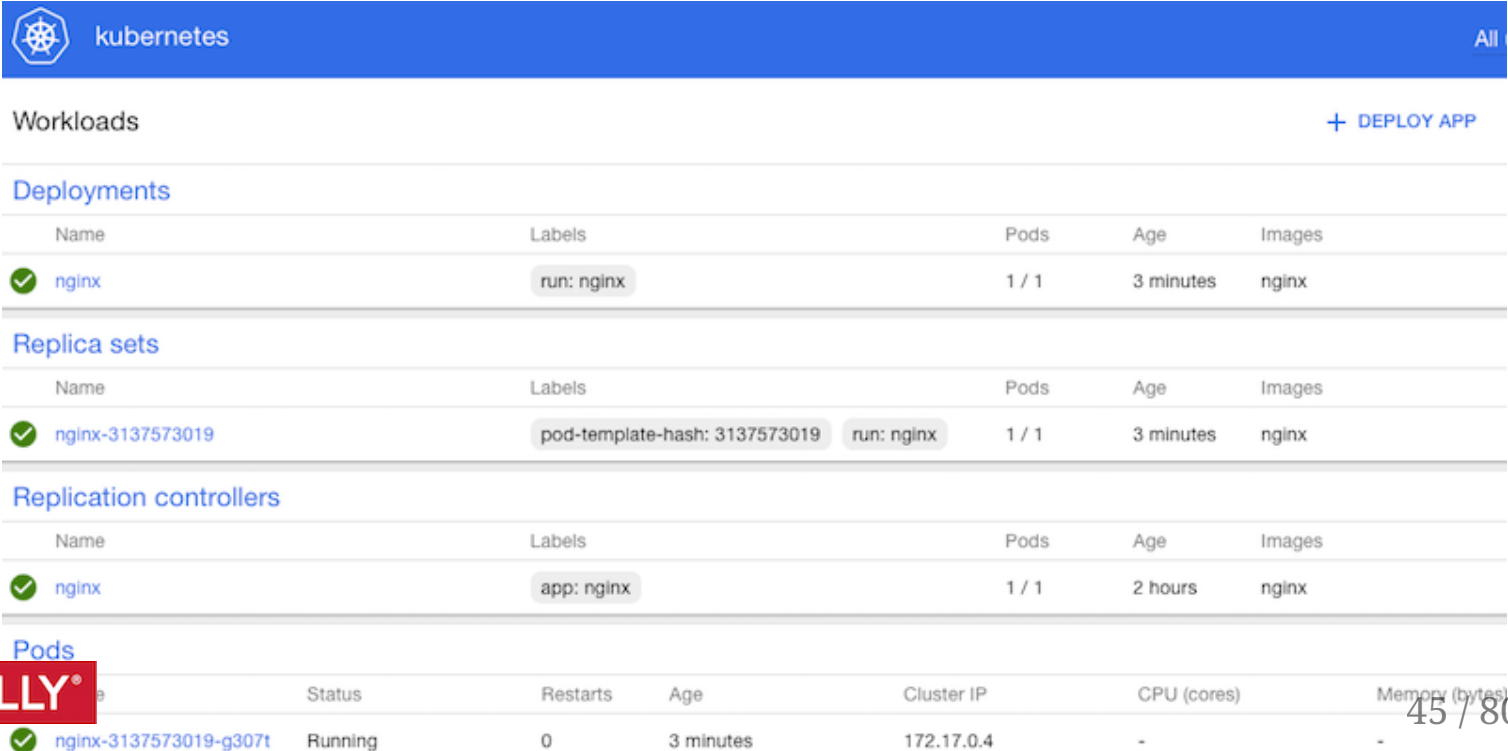
```
$ kubectl run nginx --image=nginx
```

Deployments

What does it do ?

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

Check the dashboard



The screenshot shows the Kubernetes Dashboard interface. At the top is a blue header with the Kubernetes logo and the word "kubernetes". On the right side of the header is a link that says "All". Below the header, there's a section titled "Workloads" with a "+ DEPLOY APP" button on the right. Under "Workloads", there are three expandable sections: "Deployments", "Replica sets", and "Replication controllers". Each section contains a table with columns: Name, Labels, Pods, Age, and Images. The "Deployments" section shows one deployment named "nginx" with 1 pod, 3 minutes age, and the nginx image. The "Replica sets" section shows one replica set named "nginx-3137573019" with 1 pod, 3 minutes age, and the nginx image. The "Replication controllers" section shows one replication controller named "nginx" with 1 pod, 2 hours age, and the nginx image. At the bottom, there's a "Pods" section which is partially visible, showing a table with columns: Name, Status, Restarts, Age, Cluster IP, CPU (cores), and Memory (bytes). The first pod shown is "nginx-3137573019-g307t" with a status of "Running", 0 restarts, 3 minutes age, Cluster IP 172.17.0.4, 0 CPU cores, and 0 memory bytes.

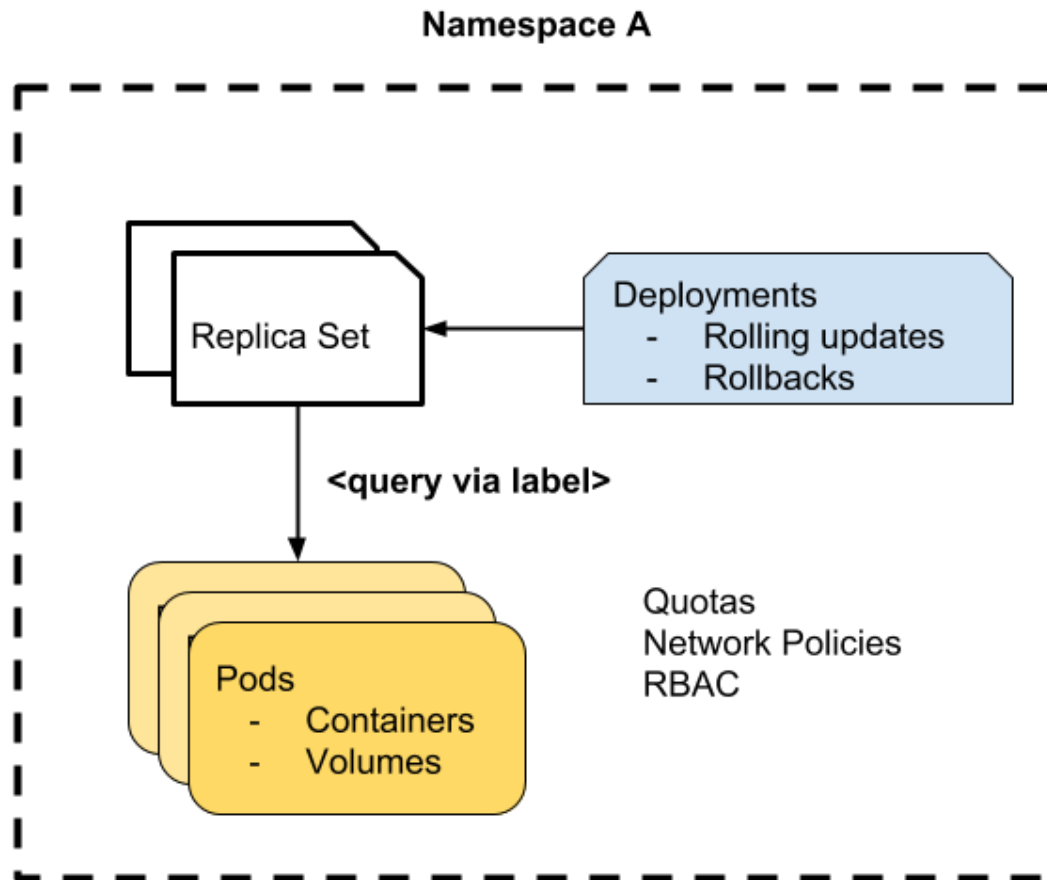
Name	Labels	Pods	Age	Images
✓ nginx	run: nginx	1 / 1	3 minutes	nginx

Name	Labels	Pods	Age	Images
✓ nginx-3137573019	pod-template-hash: 3137573019 run: nginx	1 / 1	3 minutes	nginx

Name	Labels	Pods	Age	Images
✓ nginx	app: nginx	1 / 1	2 hours	nginx

Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
✓ nginx-3137573019-g307t	Running	0	3 minutes	172.17.0.4	-	-

Deployments



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`

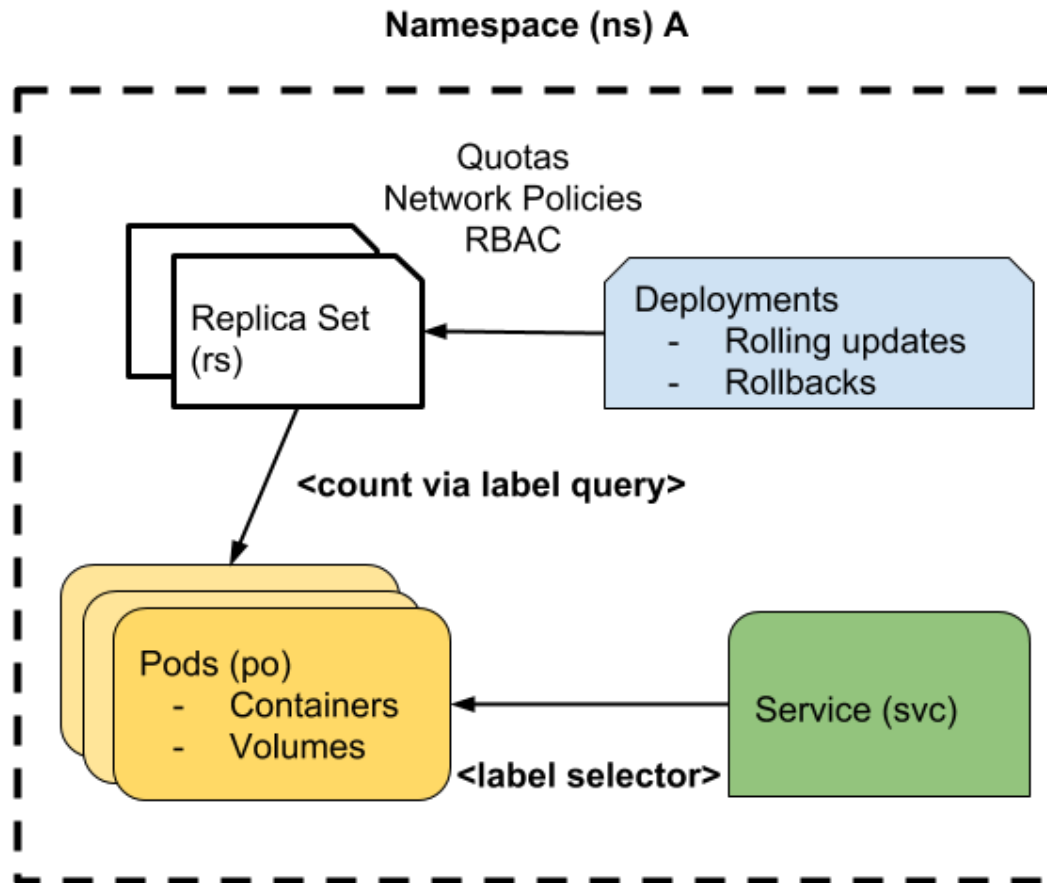
You can also edit a deploy with `kubectl edit`

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.

All Core Objects Together



Demo: Guestbook

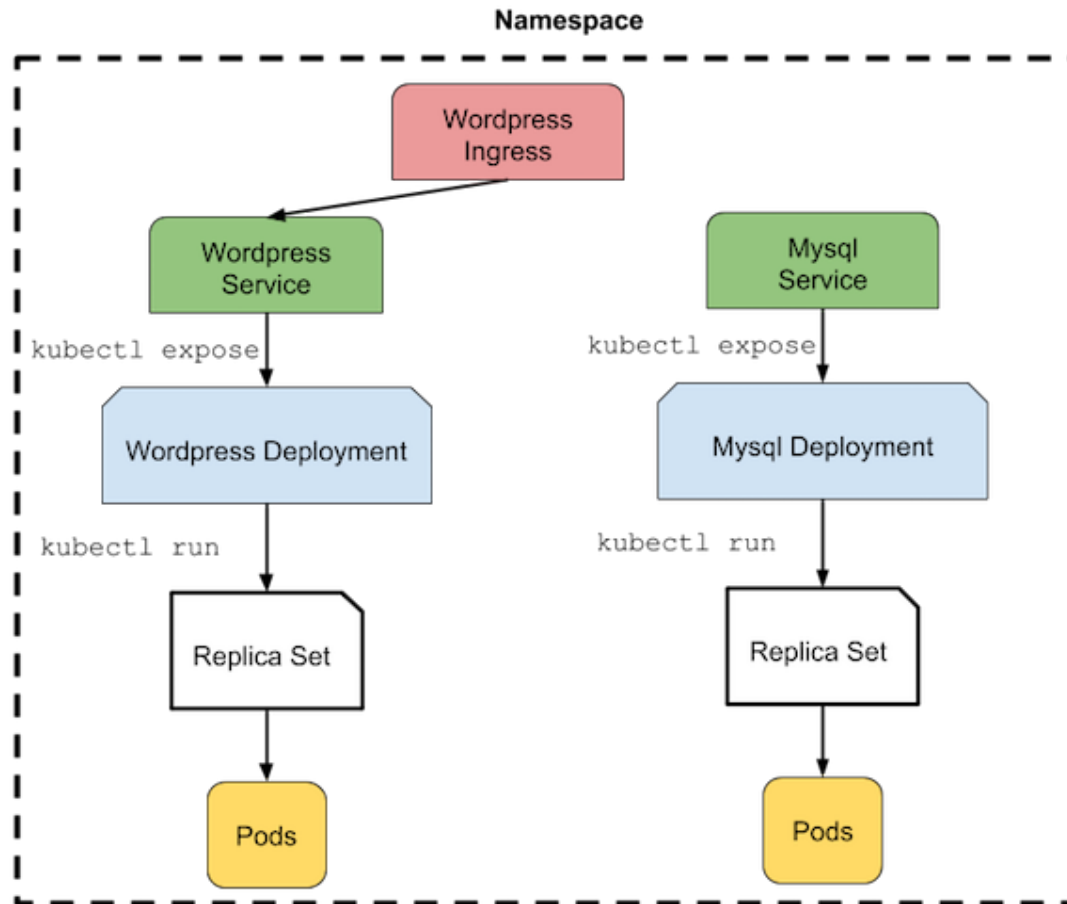
Let's run the Guestbook app !

- Find it on GitHub

All-in-one

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

Exercise: Deploy Wordpress Using Two Deployments



Deploy the Database

First, 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
```

And expose this deployment

```
$ kubectl expose deployments mysql --port 3306
```

Note that we use the default service type (e.g *ClusterIP*) since we do not want to expose our database outside our Kubernetes cluster.

Deploy the Wordpress frontend

And now create a *wordpress* deployment and service

```
$ kubectl run wordpress --image=wordpress --env WORDPRESS_DB_HOST=mysql --  
env WORDPRESS_DB_PASSWORD=root  
$ kubectl expose deployment wordpress --port 80 --type NodePort
```

And access the Wordpress frontend:

```
$ minikube service wordpress
```

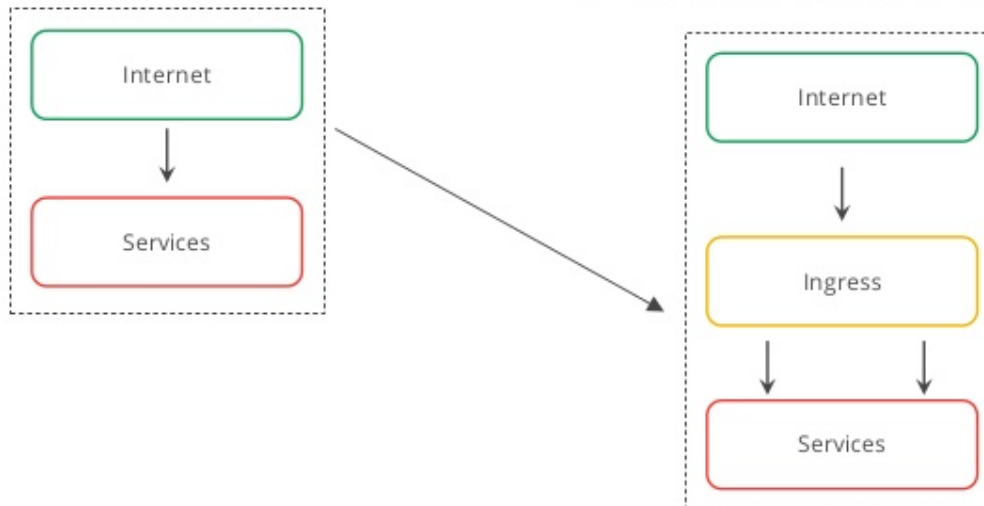
Ingress Controller

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.

The Ingress

is collection of rules that allow inbound connections to reach the cluster services



Many Ingress Controllers

- [nginx](#)
- [haproxy](#)
- [kong](#)
- [contour](#)
- [traefik](#)

Ingress API Resource

Ingress objects still an extension API like deployments, replicaset 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, binding its port 80/443 to the same host ports.

Note that you could implement your own Ingress Controller.

- Deploy the Ingress controller on minikube

```
$ minikube addons list  
$ minikube addons enable ingress
```

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
```

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.

```
kubectl create -f https://raw.githubusercontent.com/sebgoa/oreilly-kubernetes/master/manifests/wordpress/wp.yaml
```

Imperative/ Declarative

See a [blog about it](#)

```
kubectl create ns ghost
kubectl create quota blog --hard=pods=1 -n ghost
kubectl run ghost --image=ghost -n ghost
kubectl expose deployments ghost --port 2368 --type LoadBalancer -n ghost
kubectl run --generator=run-pod/v1 foobar --image=nginx
```

Get the manifests and become more declarative

```
kubectl get deployments ghost --export -n ghost -o yaml
kubectl create service clusterip foobar --tcp=80:80 -o json --dry-run
kubectl replace -f ghost.yaml -n ghost
kubectl apply -f <object>.<yaml,json>
```

Part II

- Configuration with *ConfigMap* and *Secrets*
- Data with *Volumes*

Configuration

Secrets and ConfigMap objects are used to configure apps running in containers.

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
  volumes:
  - name: test
    configMap:
      name: velocity
```

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 PersistentVolume 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)

Hint:

```
apiVersion: v1
kind: Pod
metadata:
  name: data
spec:
  containers:
    - image: mysql:5.5
      name: db
      volumeMounts:
        - mountPath: /var/lib/mysql
          name: barfoo
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: root
  volumes:
    - name: barfoo
      persistentVolumeClaim:
        claimName: foobar
```

Part III: Helm, Custom Resource Definitions and Python

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).

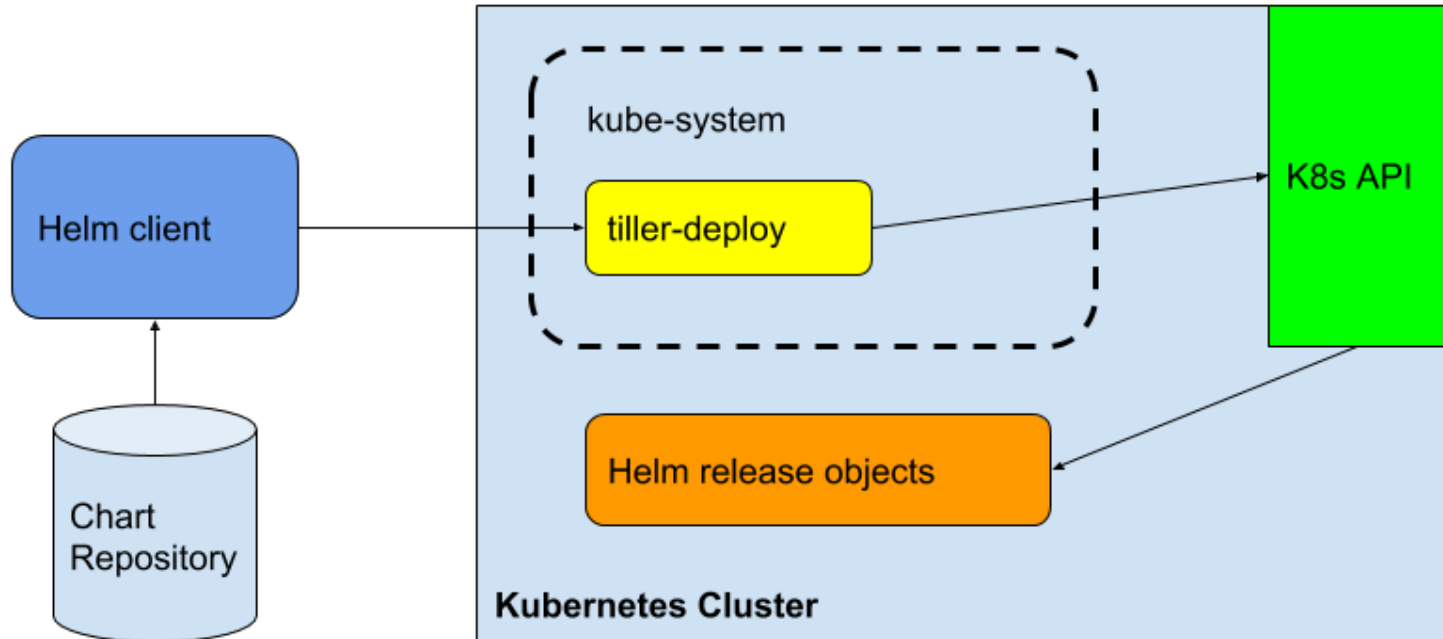
```
$ helm create oreilly
Creating oreilly
$ cd oreilly/
$ tree
.
├── Chart.yaml
├── charts
├── templates
└── values.yaml
```

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 Architecture



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
```

Custom Resource Definitions.

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 the original object: 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

```
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
NAME KIND
databases.foo.bar CustomResourceDefinition.v1beta1.apiextensions.k8s.io
```

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 foobar.yml
```

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

```
$ kubectl get databases
NAME KIND
my-new-db DataBase.v1.foo.bar
```

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

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"
```

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