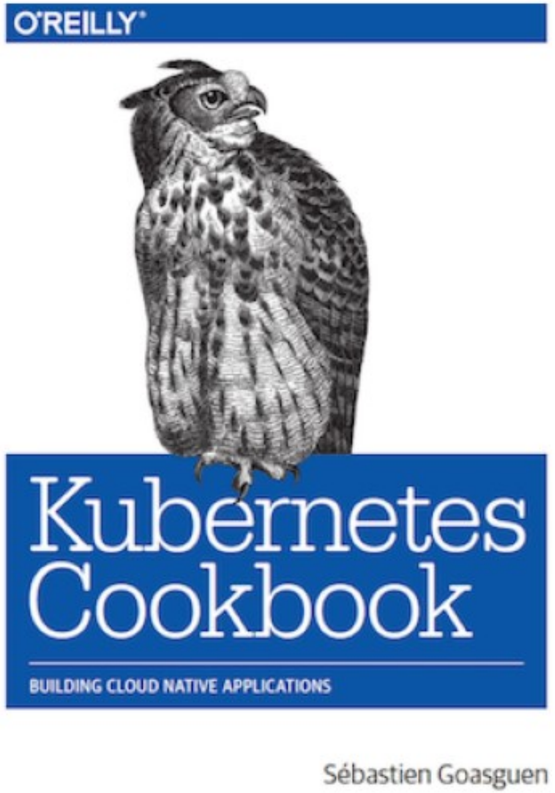
Building and Managing Kubernetes Applications - O'REILLY

By Sebastien Goasguen, author of the Docker Cookbook and co-author of Kubernetes cookbook. **@sebgoa** and **@triggermesh**



Pre-requisities

- minikube , https://github.com/kubernetes/minikube
- or Docker for Desktop (Mac/Windows)
- kubectl , https://kubernetes.io/docs/user-guide/prereqs/
- git

Manifests here:

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

Other Options for Kubernetes Testing

- `kind` with:

```
GO111MODULE="on" go get sigs.k8s.io/kind@v0.4.0
kind create cluster
```

- `k3d` with:

```
go install github.com/rancher/k3d
k3d create
```

Kubernetes Training

Goal: Building and Managing Kubernetes Applications

- Three parts in three hours
- We break every 50 minutes.
- Hands-on, follow along terminal stream
- 10 minutes break.

Agenda

- Review of most common API objects
- Focus on the Pod Specification
- Helm and **Operators**
- Other tools: kustomize ...

First a brief recap of Kubernetes

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

Part I: API Review and Declarative Mindset

- Pods, ReplicaSets, Deployments
- Secrets, ConfigMaps
- `kubectl create` and `kubectl apply`

Check API Resources/Version with `kubectl`

Check it with `kubectl`:

```
$ kubectl api-resources
$ kubectl api-versions
$ kubectl get pods
```

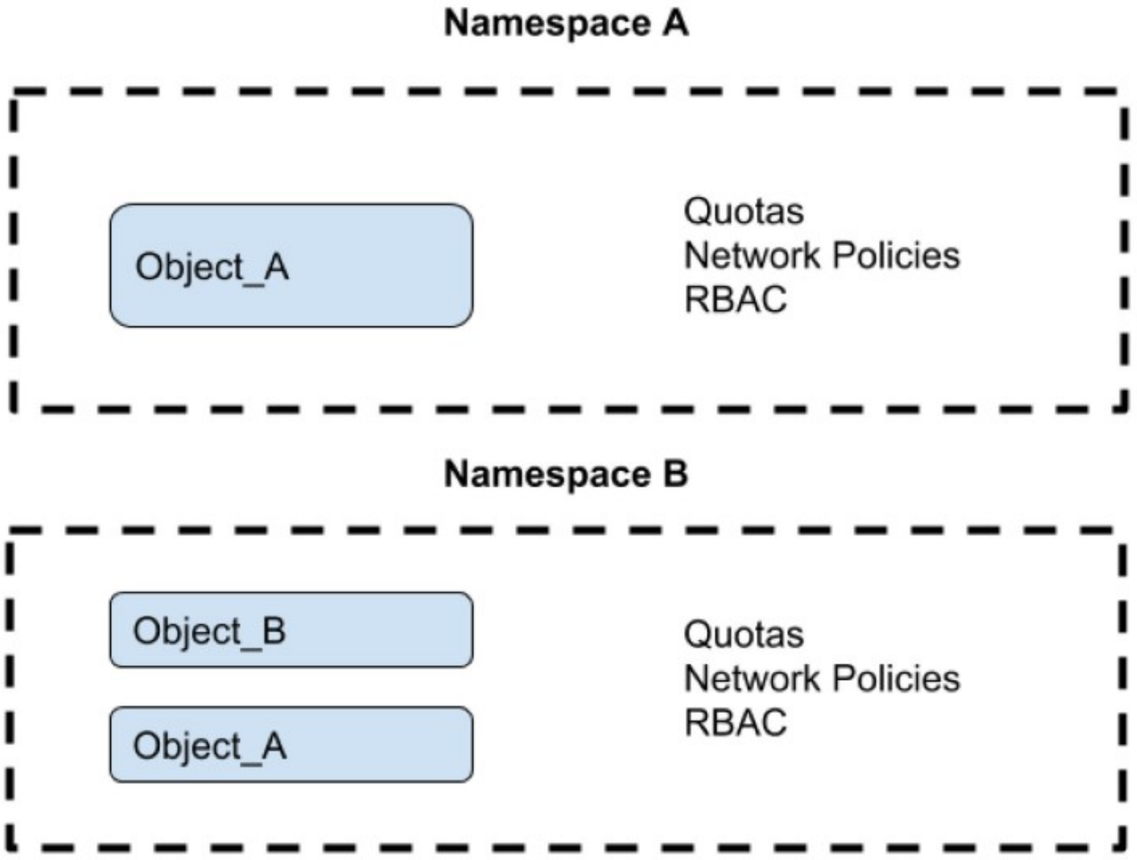
But there is much more

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

Namespaces

Every request is namespaced:

```
GET https://localhost:8001/api/v1/namespaces/default/pods
```



Labels

You will notice that every resource can contain labels in its metadata.

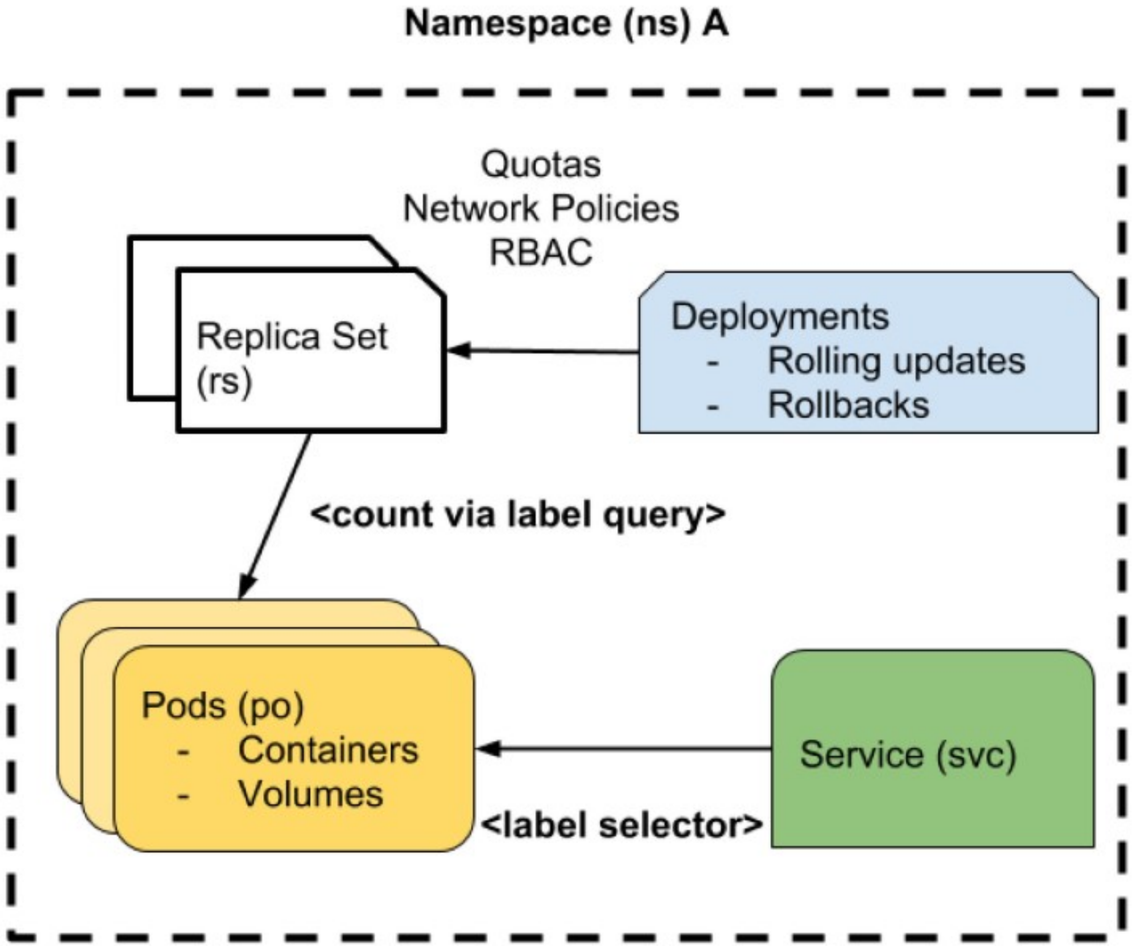
<pre> apiVersion: v1 kind: Pod metadata: ... labels: run: ghost </pre>

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

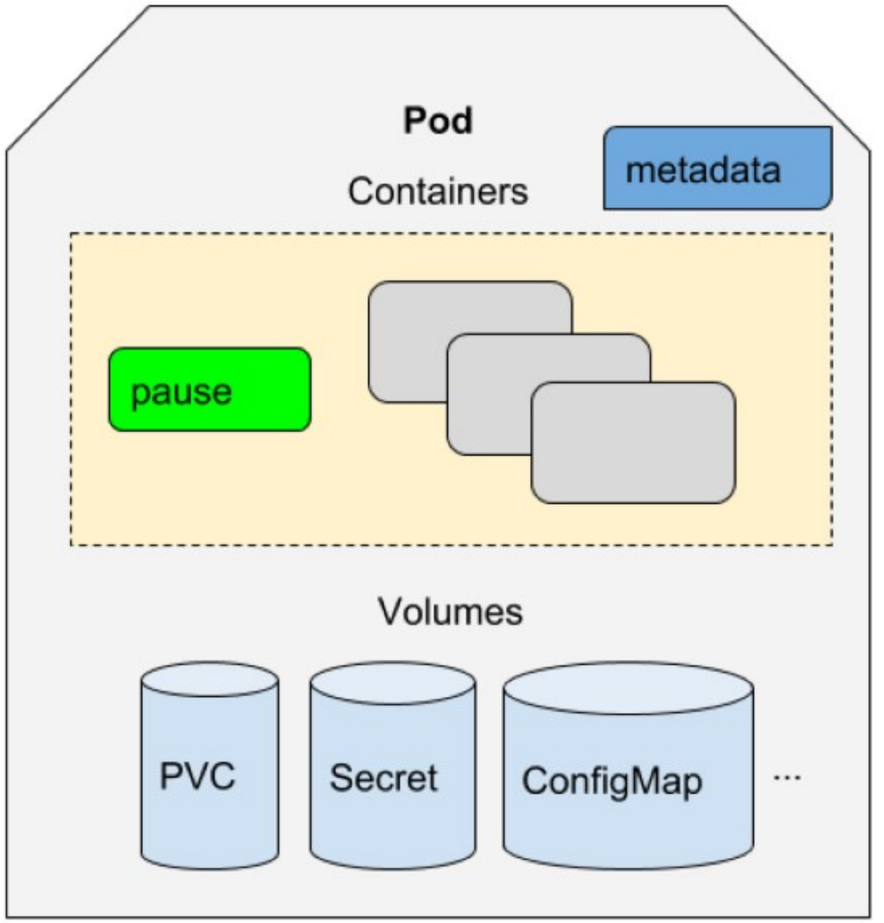
```
$ kubectl get pods -l run=ghost
NAME                     READY     STATUS    RESTARTS   AGE
ghost-3378155678-eq5i6   1/1       Running   0          10m
$ kubectl get pods -Lrun
NAME                     READY     STATUS    RESTARTS   AGE       RUN
ghost-3378155678-eq5i6   1/1       Running   0   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

Core Objects

See "Introduction to Kubernetes course"



Become Friends with Pods



Pod

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

Simplest form, single container

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

And launch it with:

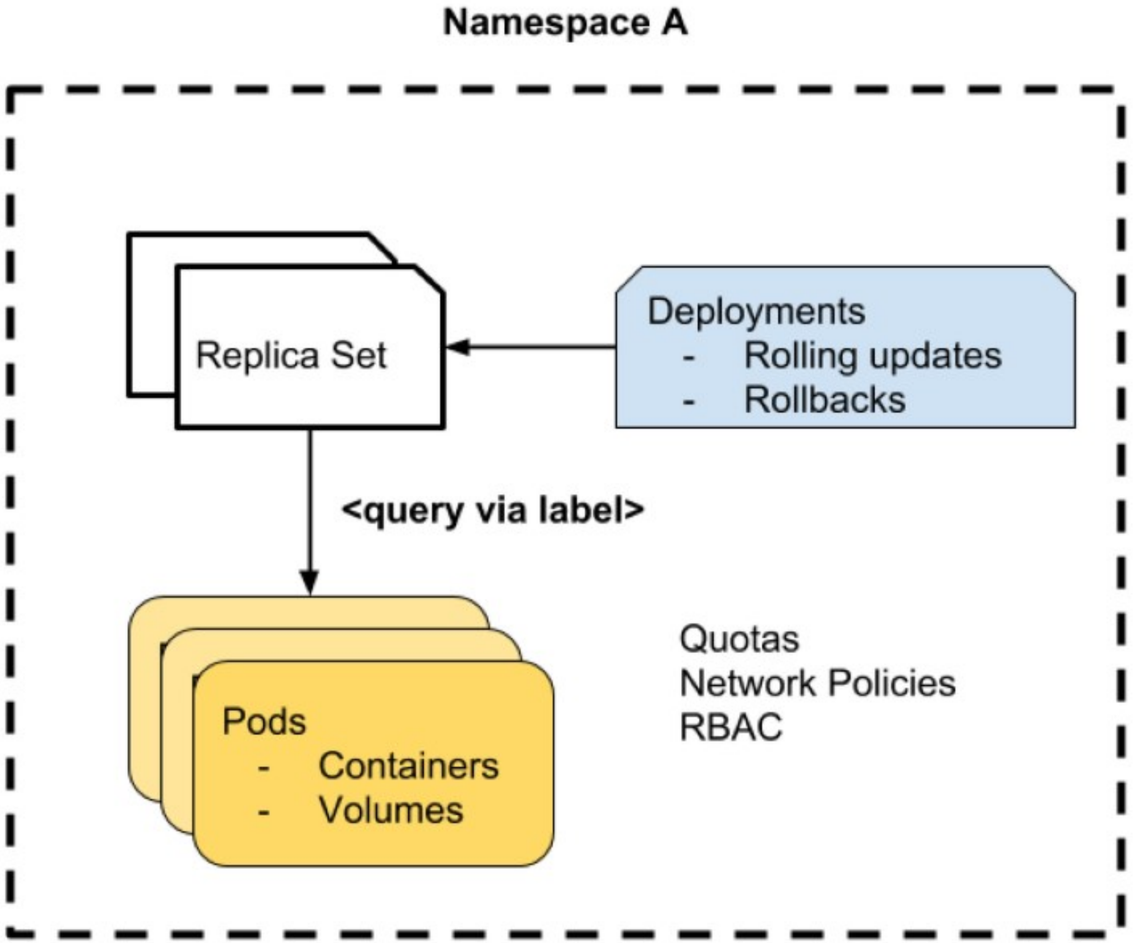
```
kubectl create -f pod.yml
```

RS Objects

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

```
apiVersion: extensions/v1beta
kind: ReplicaSet
metadata:
  name: redis
  namespace: default
spec:
  replicas: 2
  selector:
    app: redis
  template:
    metadata:
      name: redis
      labels:
        app: redis
    spec:
      containers:
      - image: redis:3.2
```

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
```

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

Watch 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`

Accessing *Services* part I

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
```

Accessing *Services* part II

```
$ 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>`

Exercise: WordPress

Create a deployment to run a MySQL Pod.

```
$ kubectl run mysql --image=mysql:5.5 --env=MYSQL_ROOT_PASSWORD=root
$ kubectl expose deployments mysql --port 3306
```

And now wordpress:

```
$ kubectl run wordpress --image=wordpress --env=WORDPRESS_DB_HOST=mysql \
                                          --env=WORDPRESS_DB_PASSWORD=root
$ kubectl expose deployments wordpress --port 80 \
                                       --type LoadBalancer
```

Other Objects

- DaemonSets
- StatefulSets
- CronJobs
- Jobs
- Ingress
- Persistent Volume Claims
- ...

e.g CronJob

A Pod that is run on a schedule

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

But a bit more focus on Pods

CKAD certification is 90% Pods

Volumes

Define array of volumes in the Pod spec. Define your volume types.

```
...
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

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
```

ConfigMap

To store a configuration 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
```

For persistency use PV and PVC

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

In Minikube dynamic provisioning is setup, you only need to write a volume claim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

Init-containers

Maybe you want to do some prep work before starting a container. Prep a file system, run some provisioning script...They run to completion and then the app starts.

You can run an *initializing* container:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
```

Requests and Limits

Great example to follow in the docs

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Relax isolation

Share the process namespace betwen all containers in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
  - name: nginx
    image: nginx
  - name: shell
    image: busybox
...
```

Probes

- Liveness probe to know when to restart a container
- Readiness probe to know when to send traffice to it

Both can be an `exec` and `http` call or a `tcp` socket connection.

```
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

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.

Declarative vs. Imperative

Different ways of managing objects, but use only one:

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Imperative commands	Live objects	Development projects	1+	Lowest
Imperative object configuration	Individual files	Production projects	1	Moderate
Declarative object configuration	Directories of files	Production projects	1+	Highest

- Imperative management using the CLI
- Imperative management using config files
- Declarative management using config files

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: Helm Package Manager

Helm in a Handbasket

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 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.

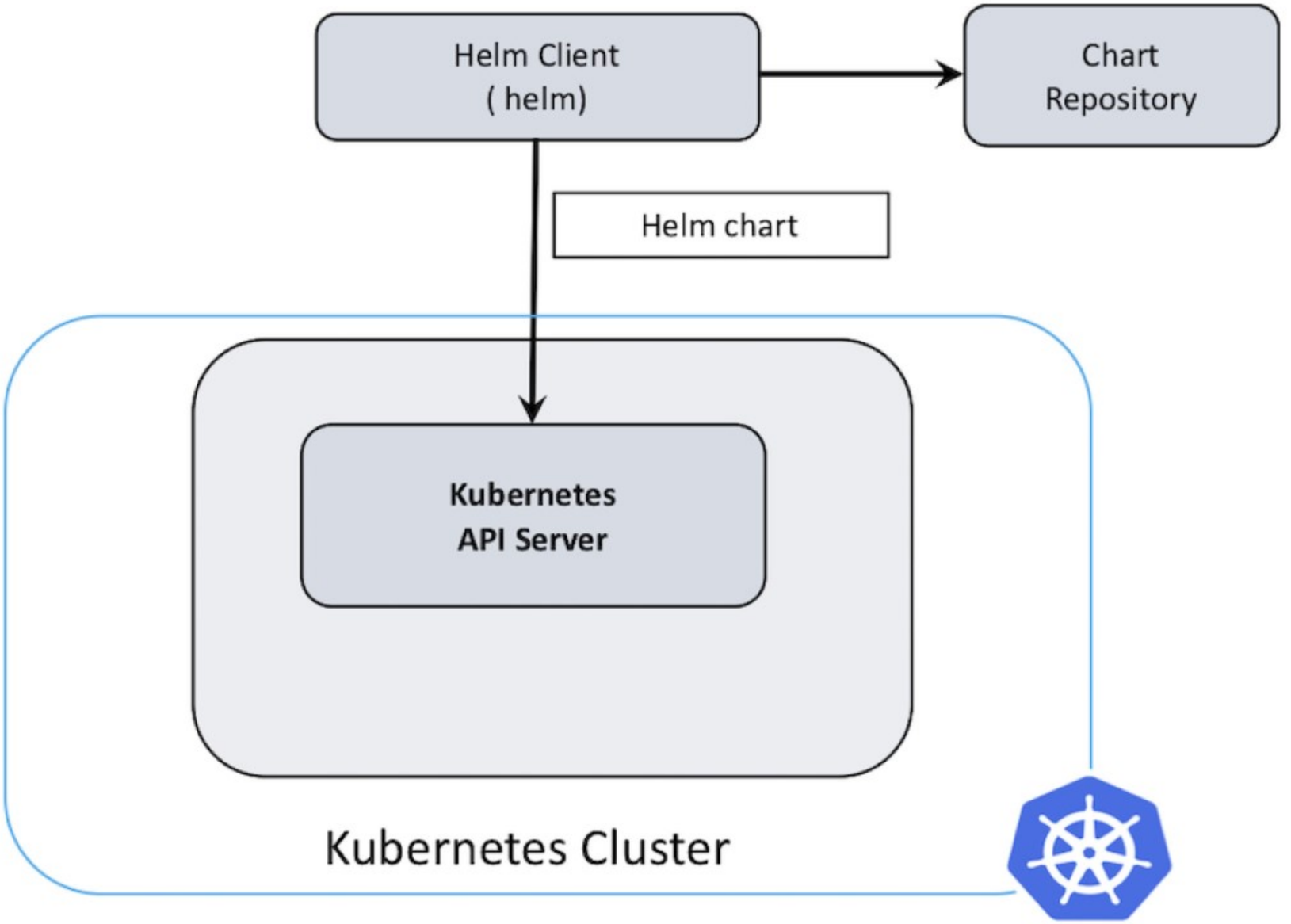
Helm Installation

https://helm.sh/docs/intro/install/

For example on OSx:

```
brew install helm
```

Architecture



Helm Example

Helm is a client that runs on your machine. You search Chart repositories, pick a Chart to install and create a *release*.

- Install Helm
- Install application

```
$ helm repo list
$ helm search repo redis
$ helm show all stable/redis
$ helm install oreilly stable/redis
```

Skeleton of a Chart

```
$ helm create oreilly
Creating oreilly
$ cd oreilly/
$ tree
.
├── Chart.yaml
├── charts
├── templates
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ...
└── values.yaml
```

Additional `helm` commands

```
$ helm show ...
$ helm show values stable/mariadb
$ helm install -f config.yaml stable/mariadb
$ helm install --set ...
$ helm install oreilly https://example.com/charts/foo-1.2.3.tgz
$ helm upgrade -f panda.yaml happy-panda stable/mariadb
$ helm get values happy-panda
$ helm rollback happy-panda 1
$ helm history ...
```

Creating a Chart repository

You can create your own index file and serve your own charts

```
$ helm repo index ...
```

And sign your charts

```
$ helm package --sign ...
$ helm verify ...
```

Helm Hub and Stable Repo

...

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com
helm repo update
```

Part III: `kustomize` and CRD operators

Best Practice is still evolving, stay tuned, joined SIG-APPS or App Def working group

Over 67 application configuration tools:

See Declarative Management in Kubernetes

kompose

A bridge between `docker-compose` and k8s. Keep your app in compose format and automatically convert it into Kubernetes objects.

Meant as on-boarding tool to learn the Kubernetes API.

```
$ kompose convert -f docker-compose.yaml
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
```

kustomize

Install

```
go get sigs.k8s.io/kustomize
```

Then write a `kustomization.yaml` file:

```
$ tree
.
├── kustomization.yaml
└── pod.yaml
```

And create the new manifest:

```
kustomize build .
```

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 objet: 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

```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
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.

Operators / Controllers

Kubernetes is based on Controllers. Control loops that make the observed and the desired state converge.

With CRDs we can create new APIs, but we need to write those control loops and write the reconciliation logic.

Operators are controllers with added logic for operational tasks (i.e Day 2 of application life-cycle):

- Database backup
- Database migration
- Replica/Sharding operations.
- etc ...

Operator-sdk

Operator framework initiated by CoreOS and now under Red Hat.

Compile it from source or download a release from GitHub

Create a new operator:

```
operator-sdk new oreilly
```

Then use the CLI to build your new operator:

```
$ operator-sdk add api --api-version=app.example.com/v1alpha1 --kind=AppService
$ operator-sdk add controller --api-version=app.example.com/v1alpha1 --kind=AppService

# Build and push the app-operator image to a public registry such as quay.io
$ operator-sdk build quay.io/example/app-operator
$ docker push quay.io/example/app-operator
```

then launch the operator:

```
kubectl apply -f config/
```

Kubebuilder

Check the *kubebuilder book* at https://book.kubebuilder.io/

Kubebuilder does "the same thing" as the operator-sdk.

Create a new API object:

```
kubebuilder create api --group fruits --version v1alpha1 --kind Banana
```

Now build the controller:

```
make install & make run
```

To deploy it in-cluster, you need to create a docker image and push it to a registry.

```
make docker-push
make docker-build
```

Python Lovers

Check out `kopf`

`kopf`

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