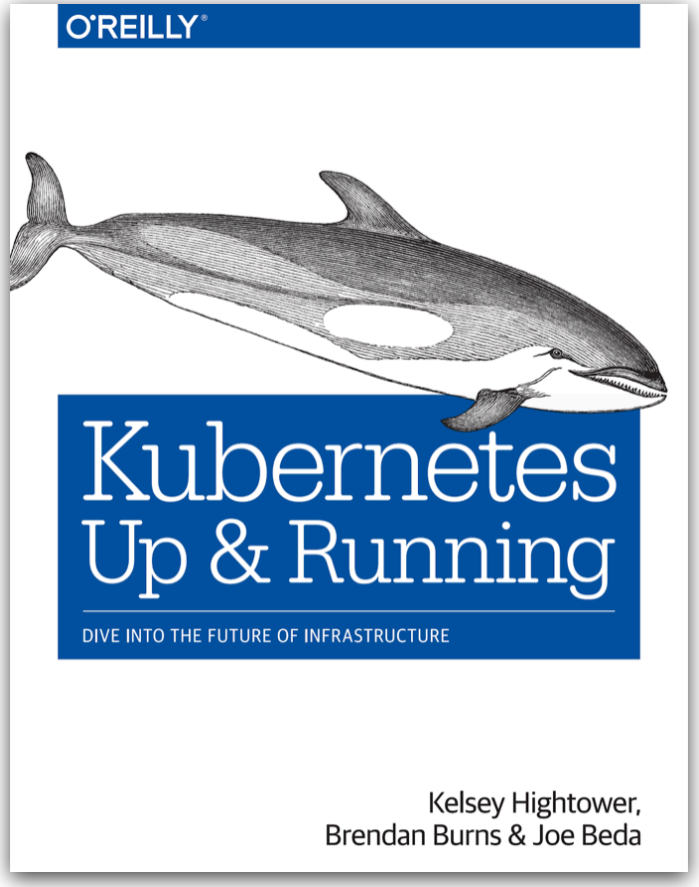
Cloud Native
#5 - Hello K8s!

This Session

Hands-on with Kubernetes



git clone http://github.com/kubernetes-up-and-running/examples

Containers at scale

Containers is great technology for encapsulation applications

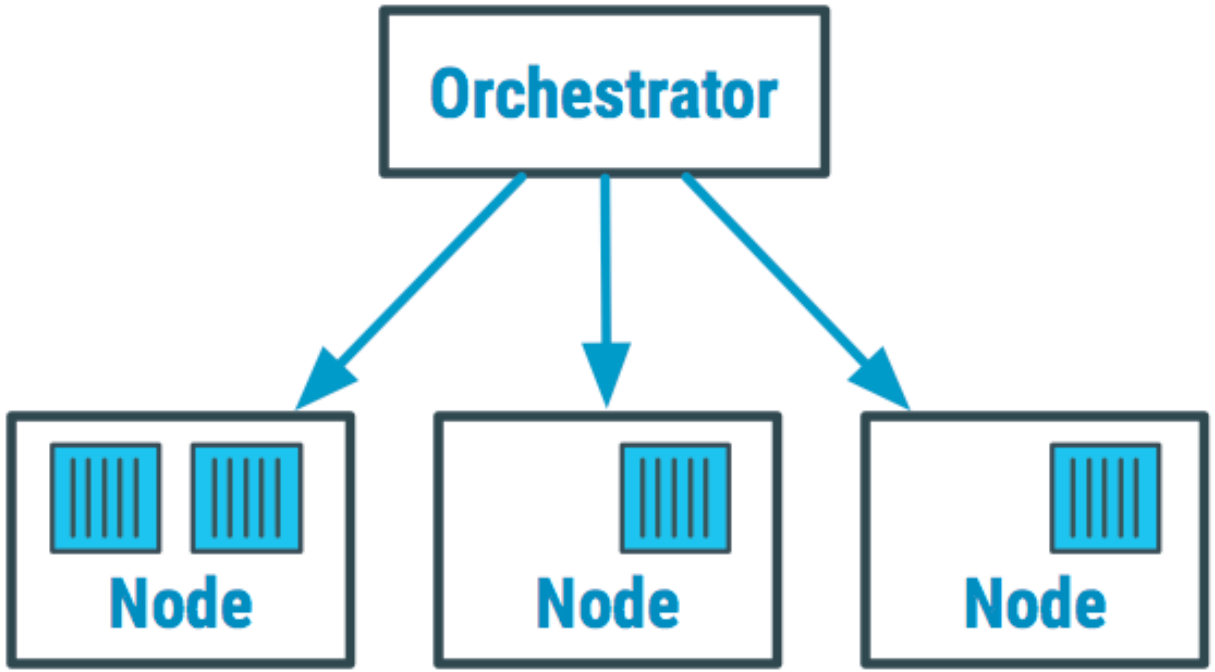
Previously, we looked at Docker to build, create, start, & stopped containers

Docker does management at a **node level**

How can we do the same for a **cluster of nodes**?

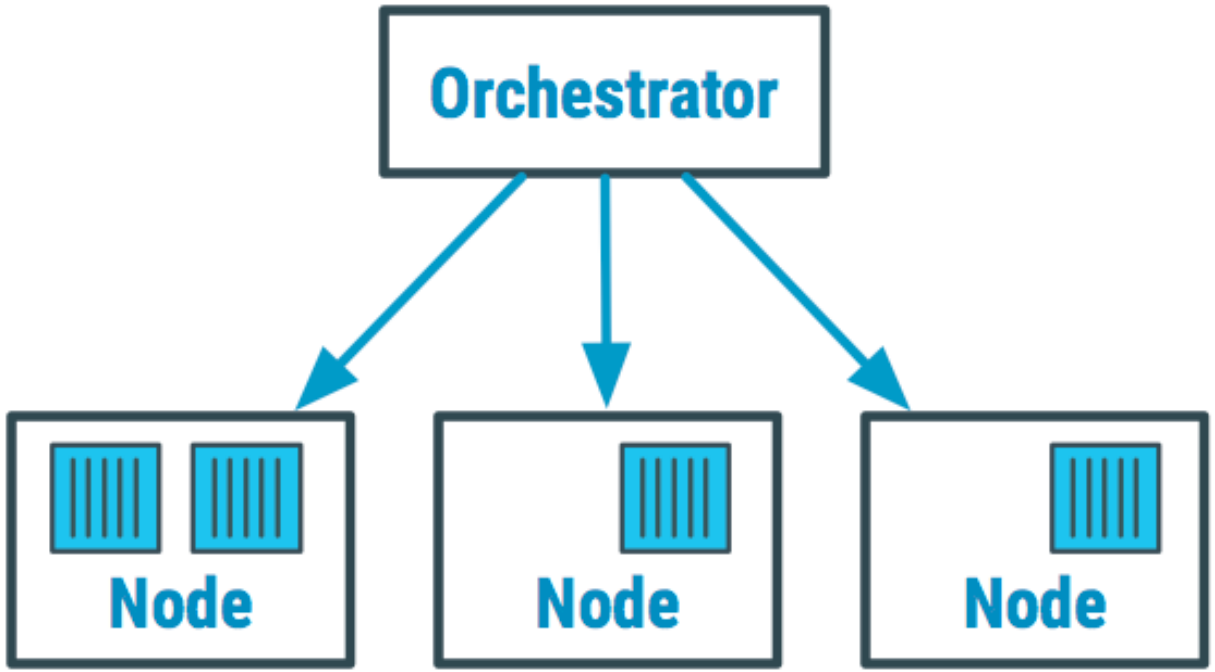
Container Orchestrator

- Manage and organize both nodes and containers running on a cluster
- Resource allocation and container scheduling according to requirements (CPU/ RAM/disk)
- Manage networking
- Access control



Container Orchestrator

- Track state of nodes and containers
  - Relocate containers in case of unresponsive node
- Service discovery
- Scale services & do load balancing
- Attach persistent storage to containers

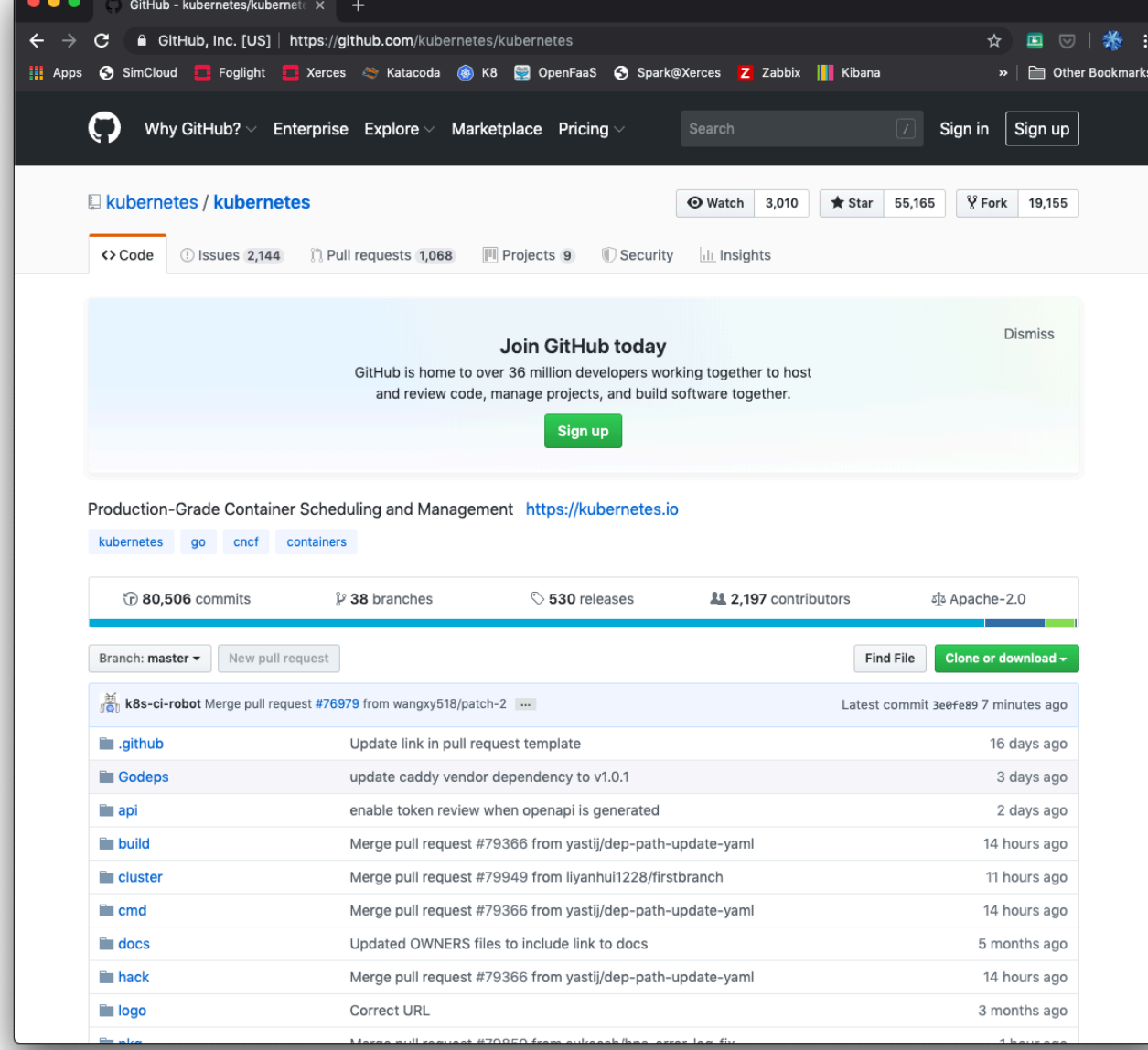


What is Kubernetes?

“Kubernetes” = Greek for governor, helmsman, captain

Originally designed by Google, maintained by CNCF

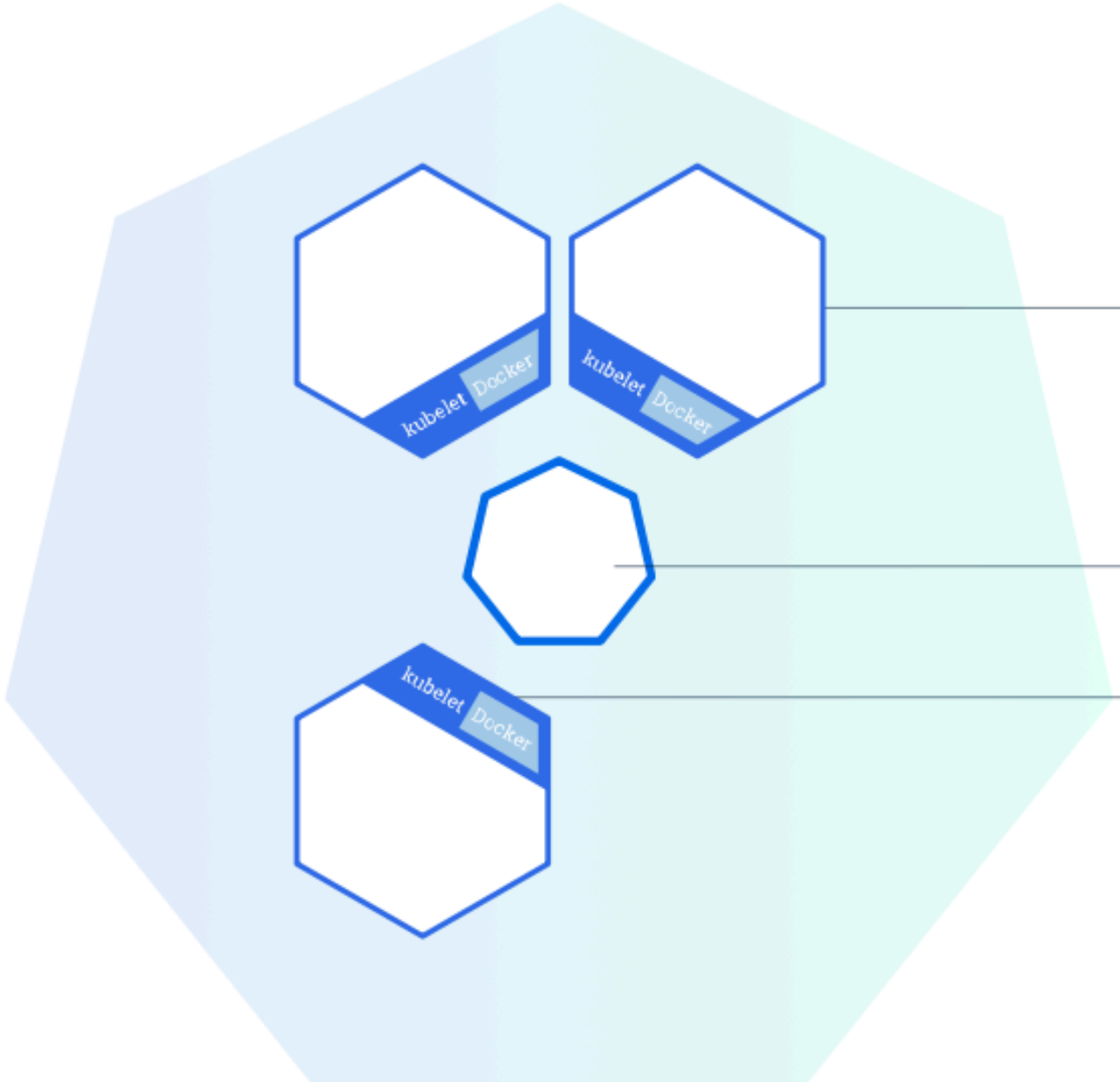
Aim to provide "platform for automating deployment, scaling and operations of application containers across clusters of hosts"

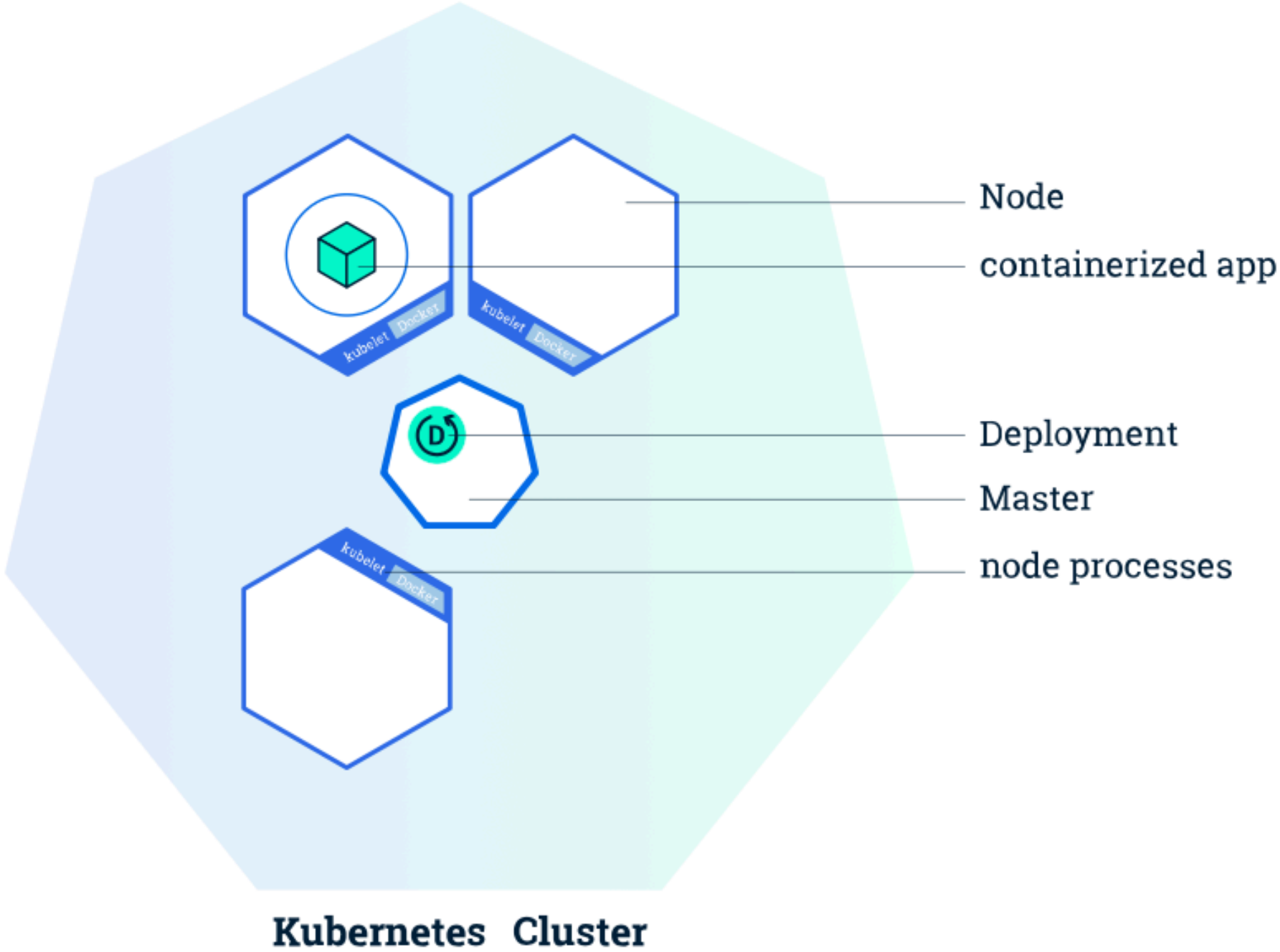


kubernetes

Key concepts

- Immutability - created artifact cannot be changed (containers)
- Declarative configuration - declare desired state (like Terraform)
- Self-healing systems - maintain desired states despite changes
- Portability - Applications can be used on another cluster without being changed

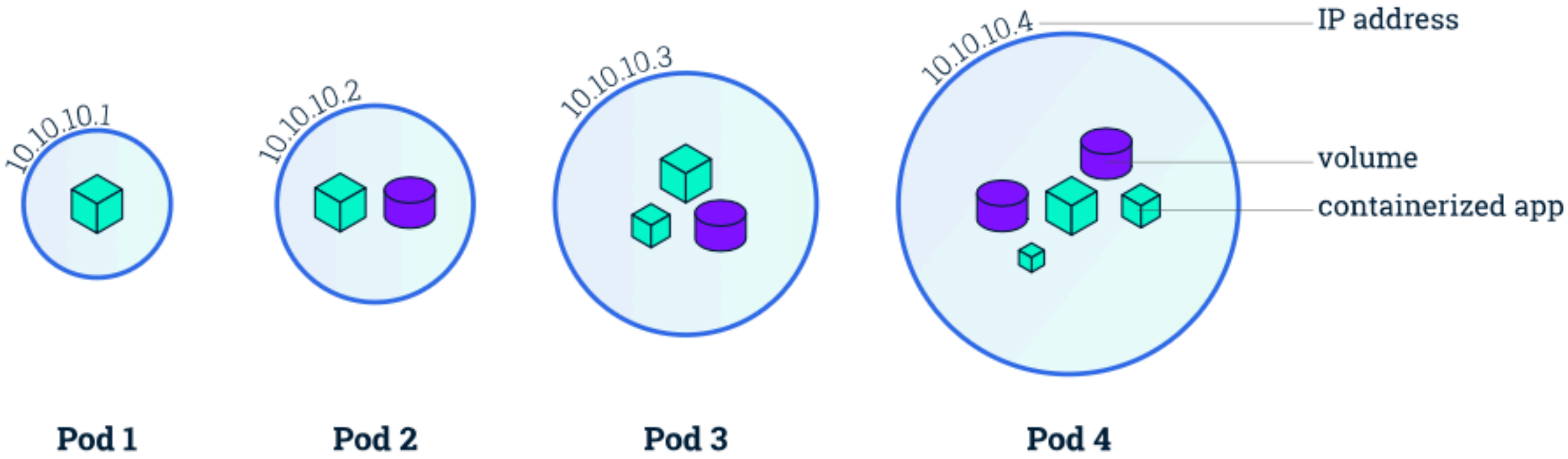

kubelet
Docker
kubelet
Docker
kubelet
Docker
Node (= a VM or a physical machine)
Master
node processes
Kubernetes cluster

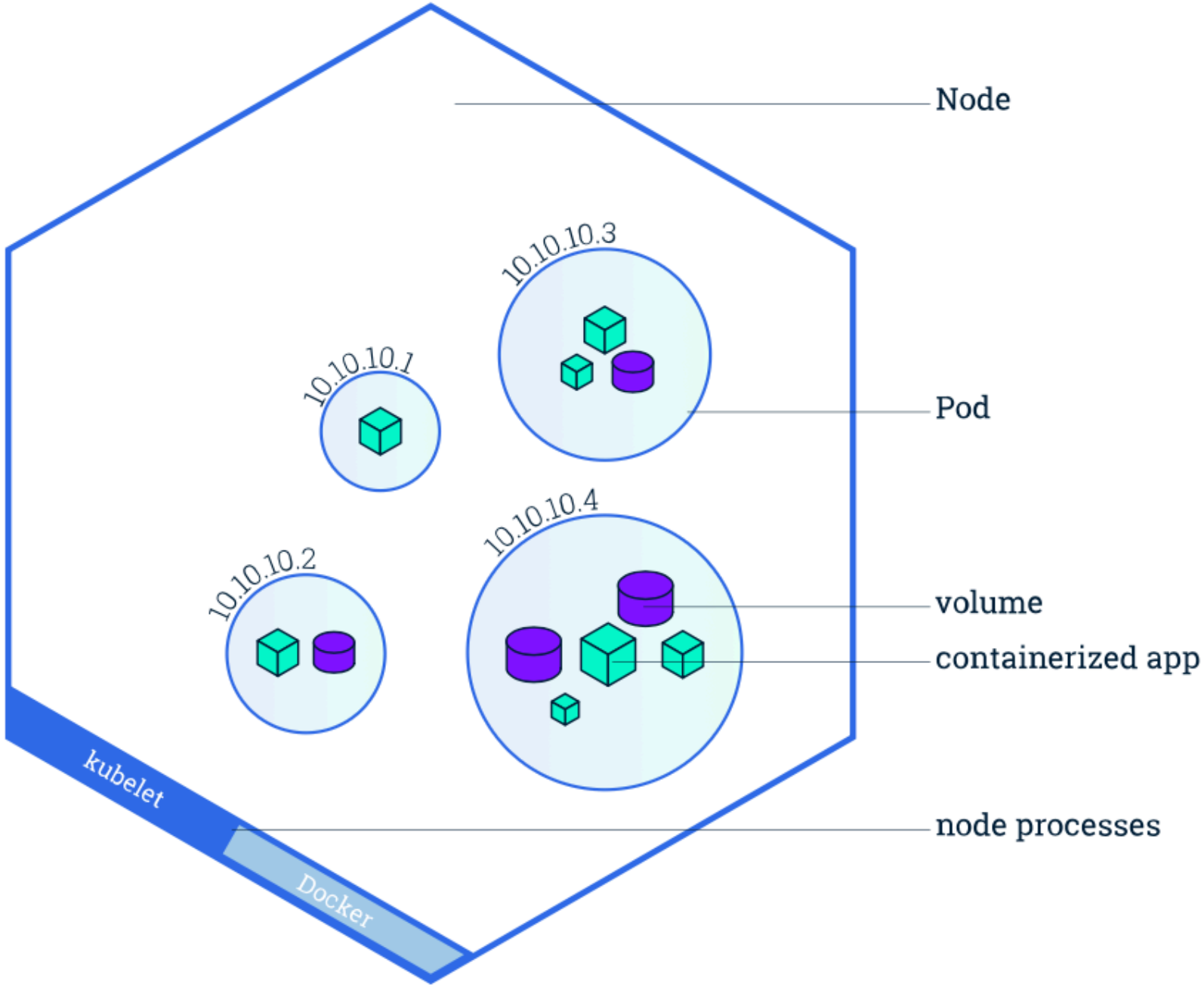

kubelet
Docker
kubelet
Docker
D
kubelet
Docker
Node
containerized app
Deployment
Master
node processes
Kubernetes Cluster

A Pod is smallest deployable unit of computing in Kubernetes

Co-located multiple apps(containers) into a single atomic unit, scheduled onto a single machine

Upon creation, statically allocated to a certain node




Node
10.10.10.3
10.10.10.1
Pod
10.10.10.4
10.10.10.2
volume
containerized app
kubelet
Docker
node processes

Pod

Each container runs in its own cgroup (CPU + RAM allocation), but they share some namespaces and filesystems, such as:

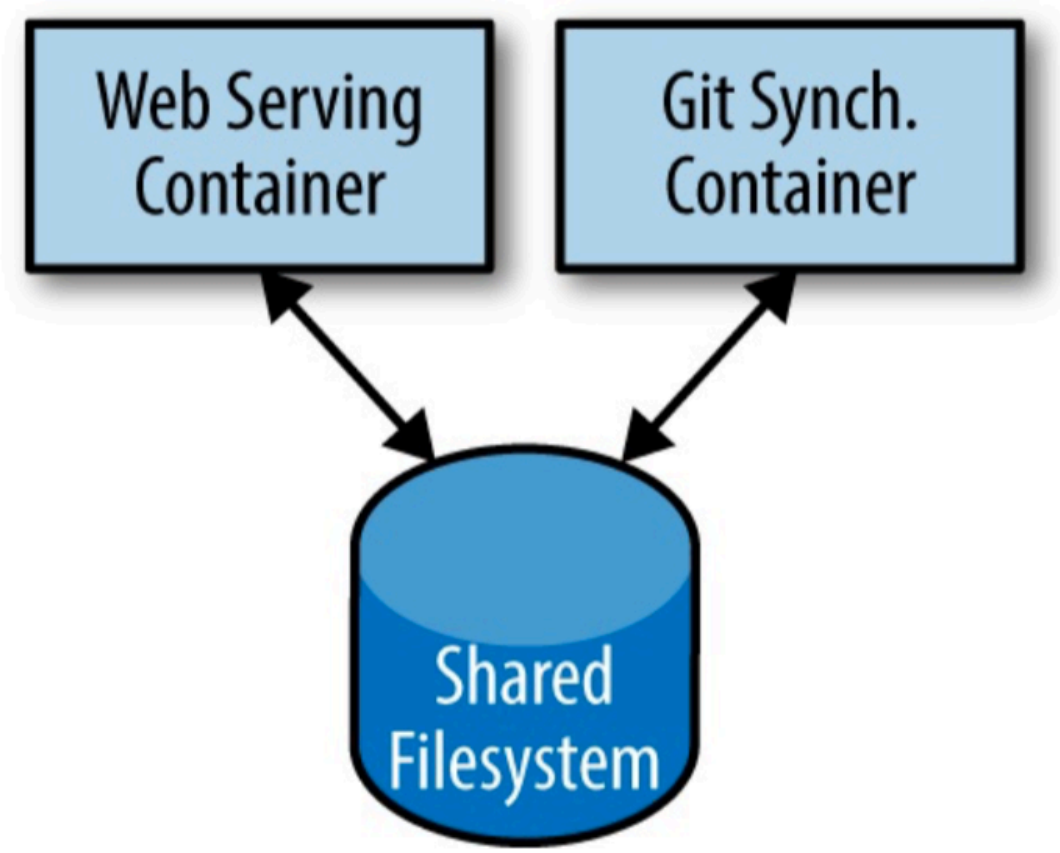
- IP address and port space
- Same hostname
- IPC channels for communication

So, why a Pod and not container directly?

For a group of symbiotic containers, that need to be kept together at all times

Pod considered running if all containers are scheduled and running

Can you deploy a container in Kubernetes?
Yes, inside a Pod!



When to have multiple containers inside a Pod?

When it's impossible for them to work on different machines (sharing local filesystem or using IPC)

When one of them facilitates communication with the other without altering it (adapter)

When one of them offers support for the other (logging/monitoring)

When one of them configures the other

Pod scheduling

Scheduler tries to scatter replicas for reliability

Pods are never moved or migrated (immutability)

What happens when a node terminates?

- Any Pod scheduled to the node needs to be deleted in order to be rescheduled

Pod life-cycle

Pods are **ephemeral**

- Kubernetes scheduler can terminate them whenever
- Like containers, any stored data will be lost once the main process stops

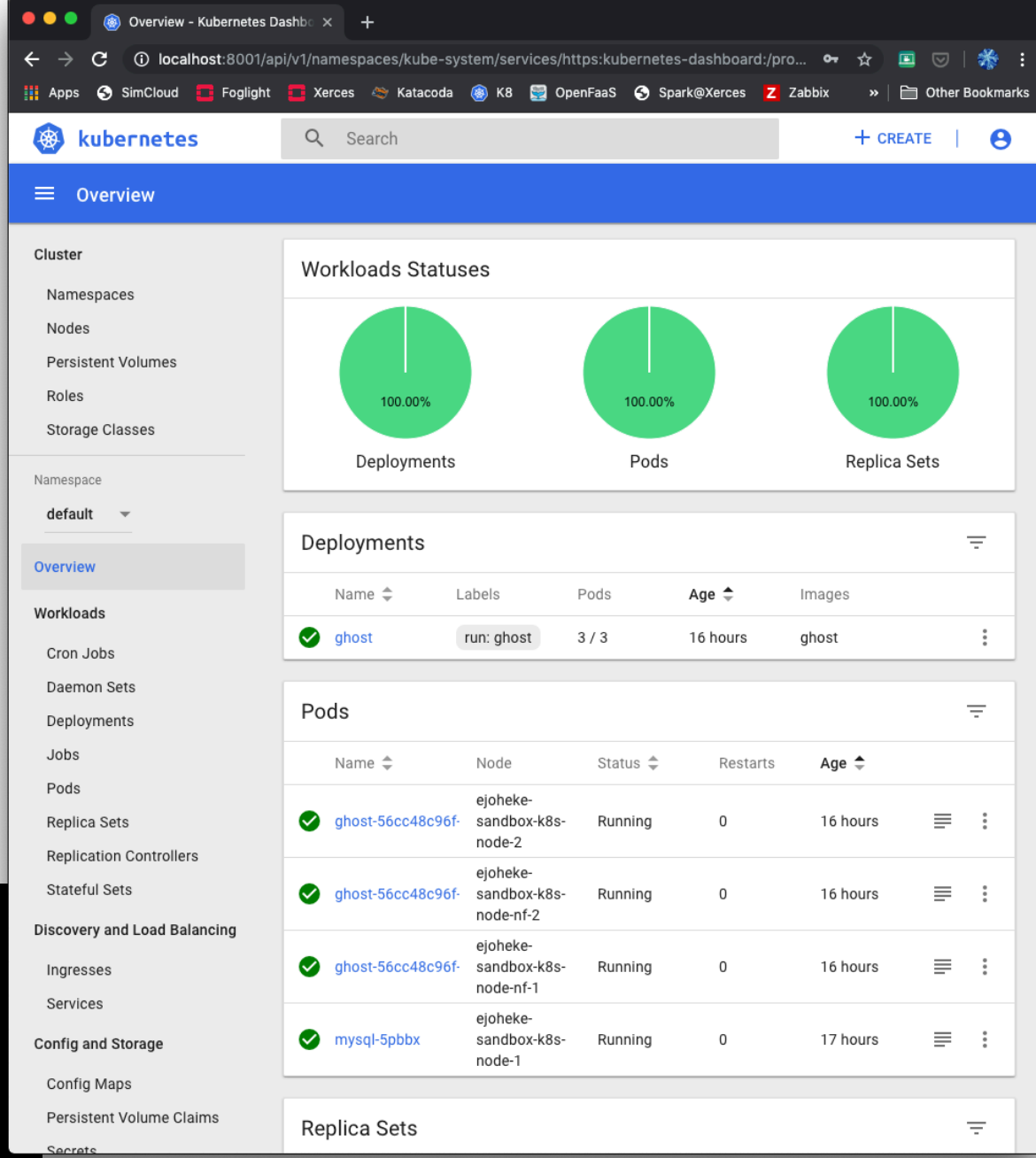
- If a Pod needs persistence, declare it! (PersistentVolumeClaim)

Interacting with Kubernetes

- kubectl command line tool
- Kubernetes dashboard

- Configurations submitted as json or yaml files

```
$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
ghost-56cc48c96f-6vzl6   1/1       Running   0          16h
ghost-56cc48c96f-9gwrx   1/1       Running   0          16h
ghost-56cc48c96f-d2mmt   1/1       Running   0          16h
mysql-5pbbx              1/1       Running   0          17h
```



A word on kubectl

— Know your tool. There is no other way. Not particularity fancy. Think of kubectl as the dev tool for cloud.

```
$ kubectl get componentstatuses
NAME                 STATUS    MESSAGE             ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}

$ kubectl get nodes
NAME                          STATUS    ROLES     AGE       VERSION
ejoheke-sandbox-k8s-master-1    Ready     master    3d21h     v1.12.5
ejoheke-sandbox-k8s-node-1      Ready     node      3d21h     v1.12.5
ejoheke-sandbox-k8s-node-2      Ready     node      3d21h     v1.12.5
ejoheke-sandbox-k8s-node-nf-1   Ready     node      3d21h     v1.12.5
ejoheke-sandbox-k8s-node-nf-2   Ready     node      3d21h     v1.12.5

$ kubectl apply -f my-config-file.yaml
```

Declarative Approach

Define desired state and apply it.
Typically as a yaml-file.

Creating pods

https://github.com/kubernetes-up-and-running/examples

```
$ cat kuard-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
  - image: gcr.io/kuar-demo/kuard-amd64:1
    name: kuard
    ports:
    - containerPort: 8080
      name: http
      protocol: TCP

$ kubectl apply -f kuard-pod.yaml
pod/kuard created

$ kubectl delete  -f kuard-pod.yaml
pod "kuard" deleted
```

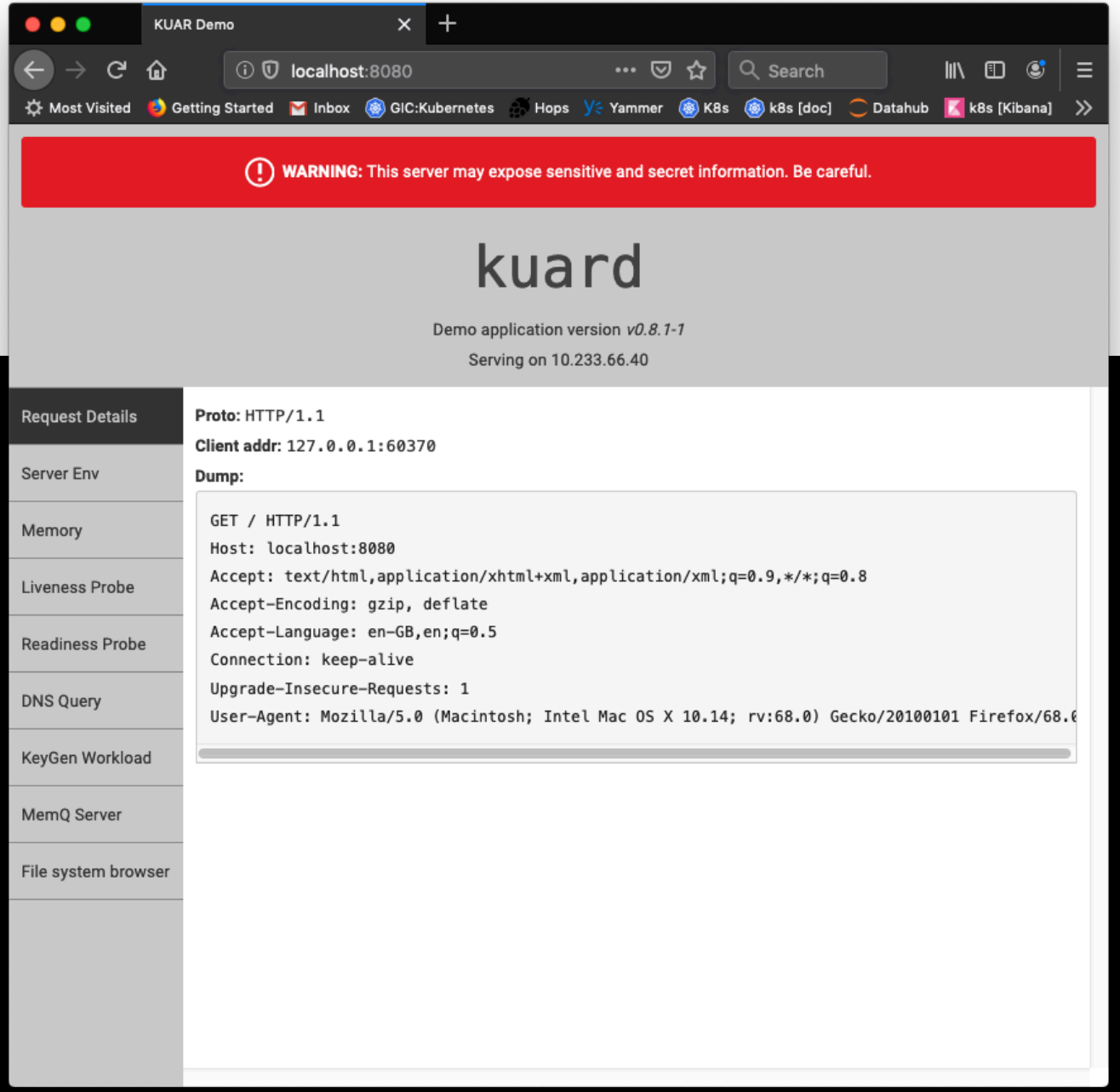
a note on "5-6-kuard-pod-full.yml"

Edit storage or it will not boot.

```
$ cat 5-6-kuard-pod-full.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      emptyDir: {}
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
```

Accessing pods

```
$  kubectl port-forward kuard 8080:8080
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from é::1ê:8080 -> 8080

$ curl http://localhost:8080
```



Learning how to debug: describe & logs

The `describe` command is your best friend. Take the time to learn to read the output. Will come in handy. Promise.

```
$ kubectl describe nodes ejoheke-sandbox-k8s-master-1

$ kubectl describe pods kuard
```

Learning how to debug . Describe & Logs


```
$ kubectl logs kuard
2019/07/12 08:46:21 Starting kuard version: v0.8.1-1
2019/07/12 08:46:21 Config:
{
  "address": ":8080",
  "debug": false,
  "debug-sitedata-dir": "./sitedata",
  "keygen": {
    "enable": false,
    "exit-code": 0,
    "exit-on-complete": false,
    "memq-queue": "",
    "memq-server": "",
    "num-to-gen": 0,
    "time-to-run": 0
  },
  "liveness": {
    "fail-next": 0
  },
  "readiness": {
    "fail-next": 0
  },
  "tls-address": ":8443",
  "tls-dir": "/tls"
}
2019/07/12 08:46:21 Could not find certificates to serve TLS
2019/07/12 08:46:21 Serving on HTTP on :8080
```

Logging into a Pod for deeper debugging

Might not be available (depends on what binaries your container includes)

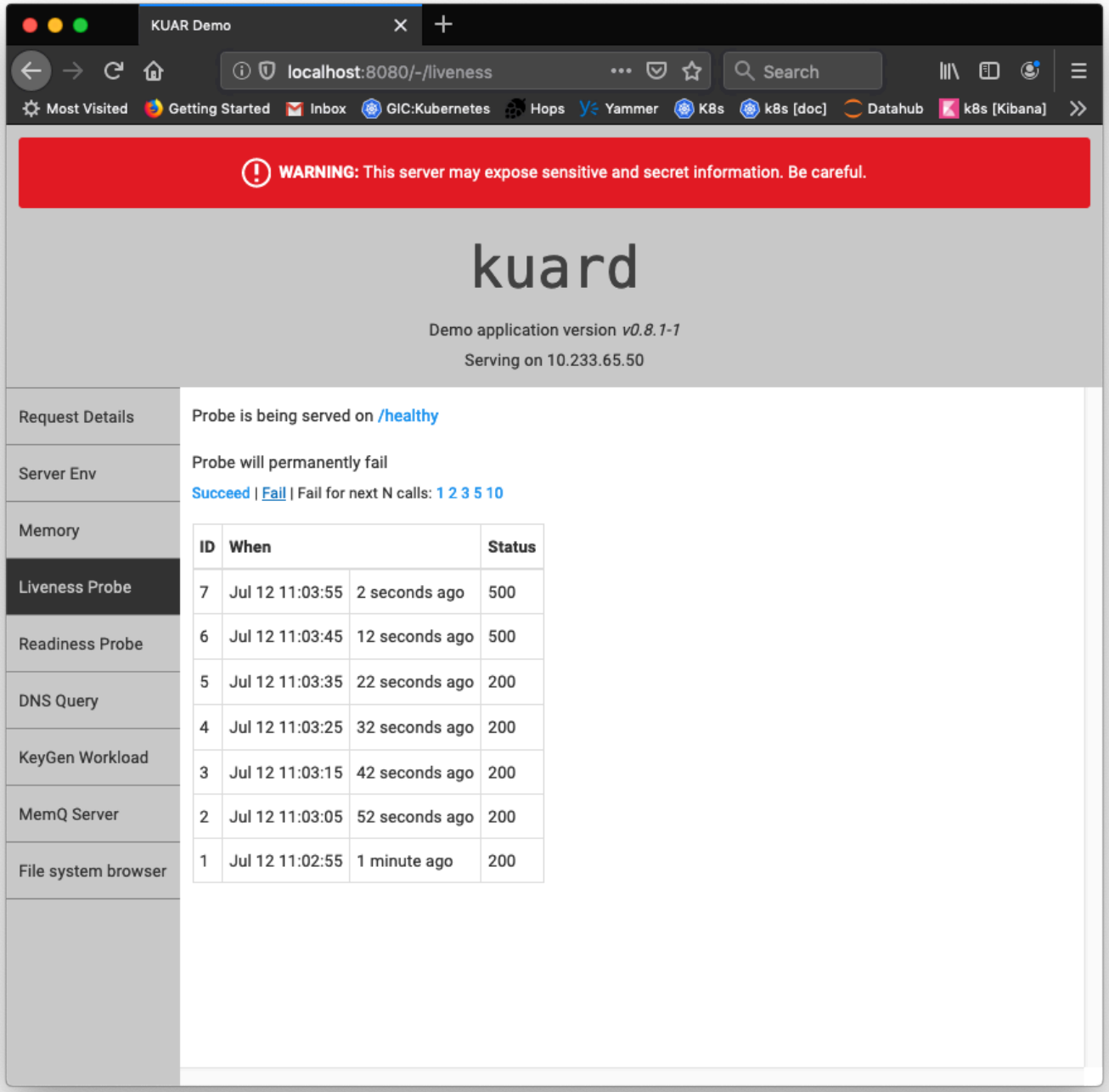
```
kubectl exec -it kuard  sh
~ $ ls
bin  dev  etc  home  kuard lib  media  mnt  proc  root  run  sbin  srv  sys  tmp  usr var
~ $ exit

$ kubectl exec  kuard nslookup kuard
Name:      kuard
   Address 1: 10.233.66.40 kuard
(k8-env) cloud_native_course$ kubectl exec  kuard hostname -- -i
10.233.66.40
```

Alive & Ready

- Kubernetes is responsible for managing and coordinating your application.
- Liveness is if an application is running properly
  - Containers that fail liveness checks are restarted
- Developer-provided liveness probe
  - httpGet/TCP/exec/etc.

```
livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

```
$ cat examples/5-6-kuard-pod-full.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
   containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
     livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      readinessProbe:
        httpGet:
          path: /ready
          port: 8080
        initialDelaySeconds: 30
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
```

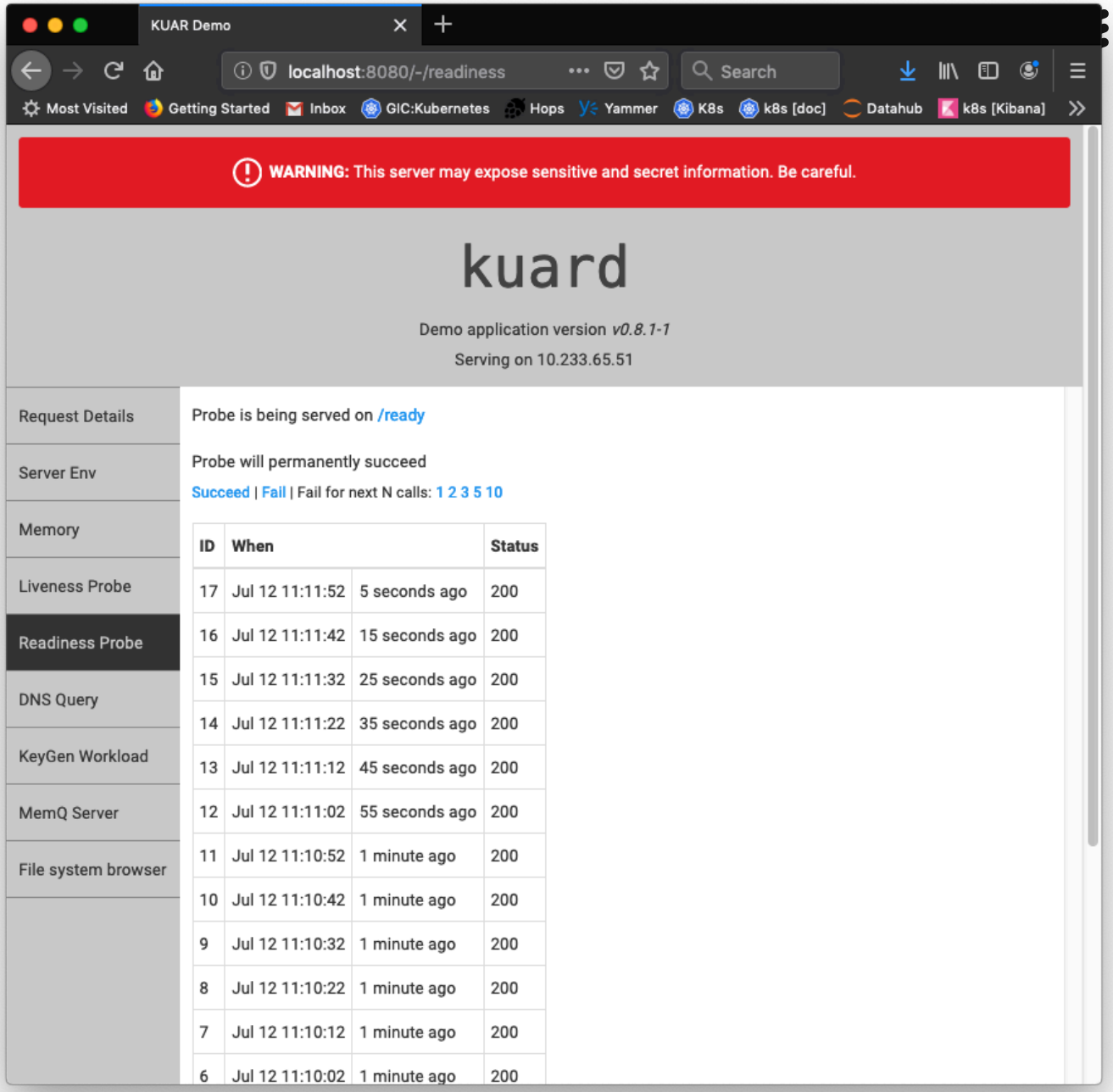


http://localhost:8080/-/liveness

Alive & Ready

- Kubernetes is responsible for managing and coordinating your application
- Readiness tells when a container is ready to serve requests
- Important when load balancing and scaling
- Distinguish boot and config latency from faults

```
$ cat examples/5-6-kuard-pod-full.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
   containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
     livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      readinessProbe:
        httpGet:
          path: /ready
          port: 8080
        initialDelaySeconds: 30
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
```



http://localhost:8080/-/readiness

ConfigMaps

- Holds configuration data
- Inject configuration into Pods upon start

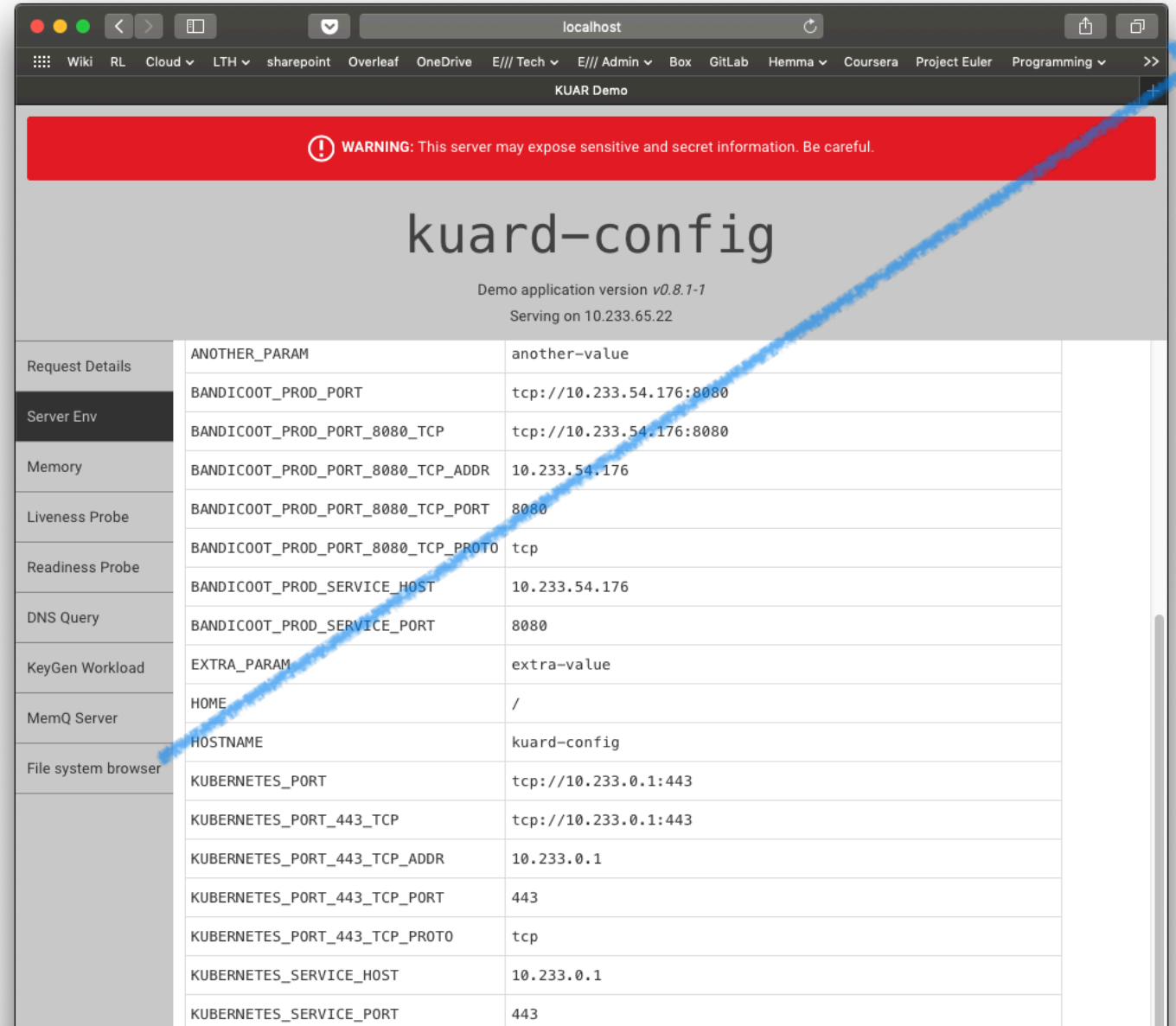
```
$ cat simple-config.txt
# This is a sample config file
parameter1 = value1
parameter2 = value2

$ kubectl create configmap my-config \
  --from-file=simple-config.txt \
  --from-literal=extra-param=extra-value \
  --from-literal=another-param=another-value

$ kubectl get configmap my-config -o yaml
apiVersion: v1
data:
  simple-config.txt: |
    # This is a sample config file
    parameter1 = value1
    parameter2 = value2
  another-param: another-value
  extra-param: extra-value
kind: ConfigMap
metadata:
  creationTimestamp: 2019-07-10T08:41:34Z
  name: my-config
  namespace: default
  resourceVersion: "266897"
  selfLink: /api/v1/namespaces/default/configmaps/my-config
  uid: 85d57f0d-a2ee-11e9-89b9-fa163eaeeaa4
```

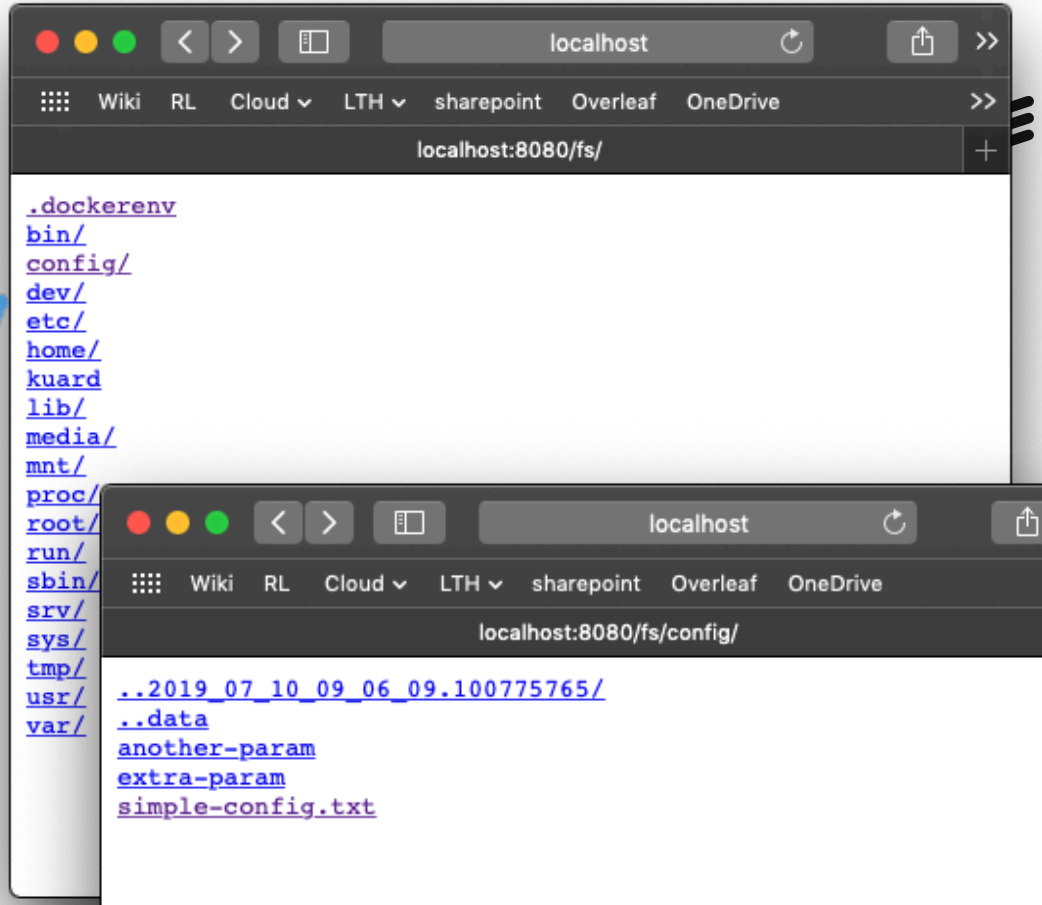
ConfigMaps

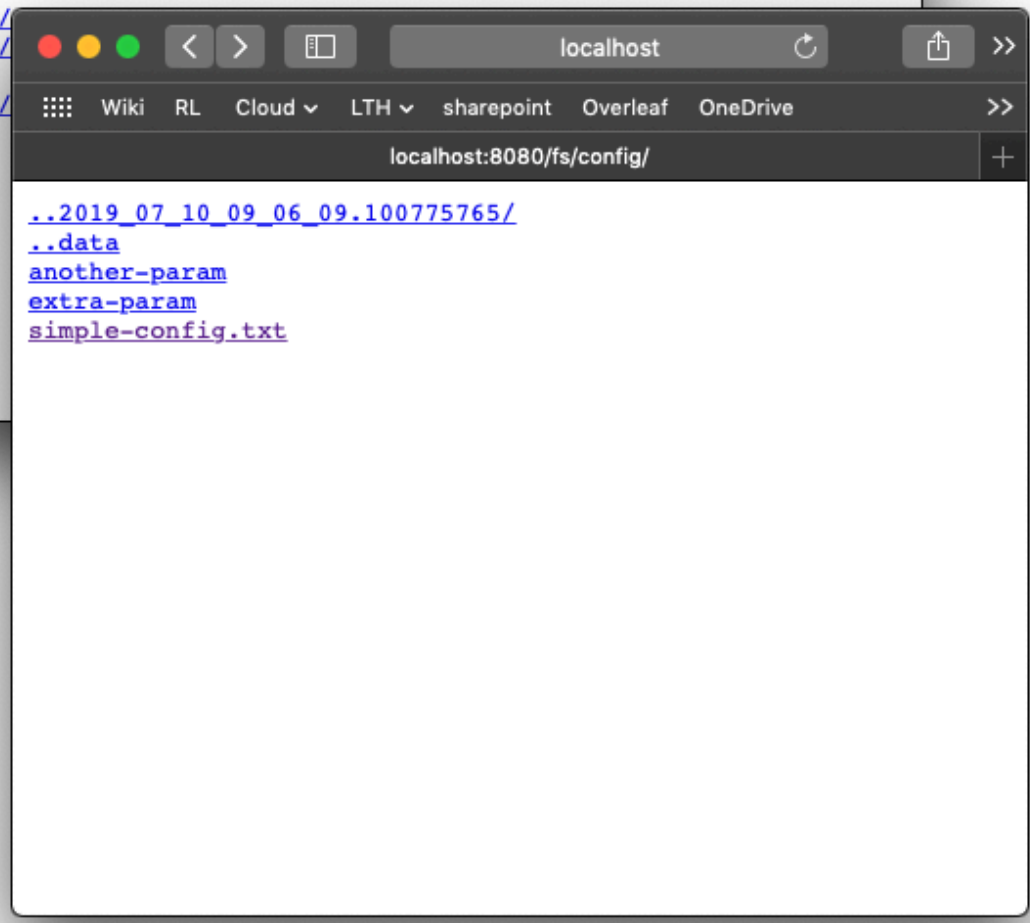
- ConfigMap contents can be given as a
  - filesystem
  - environment variable
  - command-line argument

```
cat kuard-config.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard-config
spec:
  containers:
    - name: test-container
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      command:
        - "/kuard"
        - "$(EXTRA_PARAM)"
      env:
       - name: EXTRA_PARAM
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: extra-param
      volumeMounts:
        - name: config-volume
          mountPath: /config
  volumes:
    - name: config-volume
      configMap:
        name: my-config
  restartPolicy: Never
```

```
kubectl apply -f kuard-config.yaml
kubectl port-forward kuard-config 8080
```







Secrets

- ConfigMap is for public data
- Secrets are for confidential data such as passwords, secret tokens, cerificates, etc.
  - No real encryption, though!
  - Avoids simple "reading over your shoulder" attacks
- Secrets is collection of key/value pairs

```
$ kubectl create secret generic kuard-tls \
 --from-file=kuard.crt \
 --from-file=kuard.key

$ kubectl describe secrets kuard-tls
Name:         kuard-tls
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
kuard.crt:  1050 bytes
kuard.key:  1679 bytes
```

Secrets

- Secrets are exposed like ConfigMaps
- File system exposure as RAM disks (thus not written to disk on nodes)

```
$ cat kuard-secret.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard-tls
spec:
  containers:
    - name: kuard-tls
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      volumeMounts:
      - name: tls-certs
        mountPath: "/tls"
        readOnly: true
  volumes:
    - name: tls-certs
      secret:
        secretName: kuard-tls
```

— Secrets are useful for handling credentials, for example when accessing a private Docker registry

```
$ kubectl create secret docker-registry my-image-pull-secret \
 --docker-username=<username> \
 --docker-password=<password> --docker-email=<email>$


$ cat kuard-secret-ips.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard-tls
spec:
  containers:
    - name: kuard-tls
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      volumeMounts:
      - name: tls-certs
        mountPath: "/tls"
        readOnly: true
  imagePullSecrets:
  - name:  my-image-pull-secret
  volumes:
    - name: tls-certs
      secret:
        secretName: kuard-tls
```

Resource management

- Kubernetes allocates resources for your application
- "requests" are the required minimum resources
- "limits" are the maximum resources
- Enforced using Linux cgroups

```
$ cat kaurd-pod-reslim.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Volumes

- Containers in a pod can share data via a local volume
  - Data survives container restart but lost on termination
  - Remember that Pods are ephemeral!
  - Need a "scratch space" (working directory)?
    - emptyDir: {}
    - ...but know that it, too, will be lost (docs)
- A Pod mount en external volume for persistent data
  - NFS share
  - OpenStack Cinder volume
  - And many others

```
$ cat examples/5-6-kuard-pod-full.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      nfs:
        server: my.nfs.server.local
        path: "/exports"
      volumes:
    - name: "host-data"
      hostPath:
        path: "/var/lib/kuard"
  volumes:
    - name: cache-volume
      emptyDir: {}
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
     volumeMounts:
        - mountPath: "/data"
          name: "kuard-data"
        - mountPath: /cache
          name: cache-volume
```

Labels and annotations

- A key idea of Kubernetes is to keep parts loosely coupled, no hard-wired knowledge
- Use labels, annotations & selectors selectors to identify sets and services
- Both are key/value pairs
- Labels provide the foundation for grouping objects.
- Annotations are designed to hold non-identifying info to be used by tools & libraries

Labels and annotation

```
$ kubectl run alpaca-prod --image=gcr.io/kuar-demo/kuard-amd64:1 --replicas=2 \
  --labels="ver=1,app=alpaca,env=prod"
deployment.apps/alpaca-prod created
$

$ kubectl get deployments --show-labels
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE   LABELS
alpaca-prod        2         2         2            0           1s    app=alpaca,env=prod,ver=1
alpaca-test        1         1         1            0           0s    app=alpaca,env=test,ver=2
bandicoot-prod     2         2         2            0           0s    app=bandicoot,env=prod,ver=2
bandicoot-staging  1         1         1            0           0s    app=bandicoot,env=staging,ver=2
```

Service

Abstraction defining a logical set of Pods and provides network access to them

- Accessible via a cluster-internal DNS name
- Dynamic set of Pods identified using e.g. label selector
- Unlike Pods, Service stay until explicitly deleted (not ephemeral)

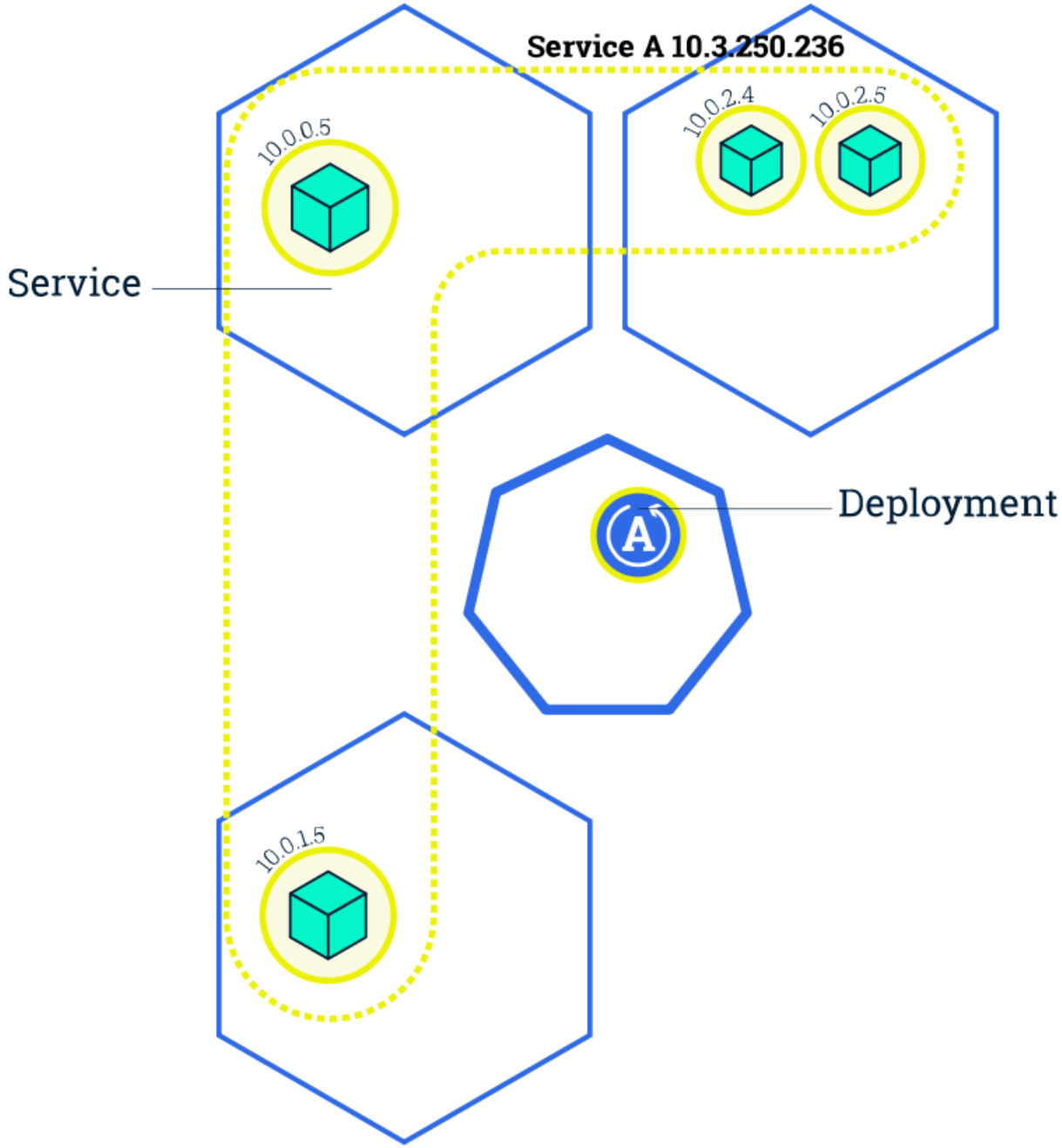
Service Types

Different Service types, depending on desired service exposure:

- ClusterIP (cluster-internal services) (default!)
- NodePort (exposed externally via a common port on each node)
- LoadBalancer (provisions cloud-supported load balancer for external exposure)

- ExternalName ("imports" a service running outside the cluster into the cluster's context)

Services


Service A 10.3.250.236
10.0.0.5
10.0.2.4
10.0.2.5
Service
Deployment
10.0.1.5

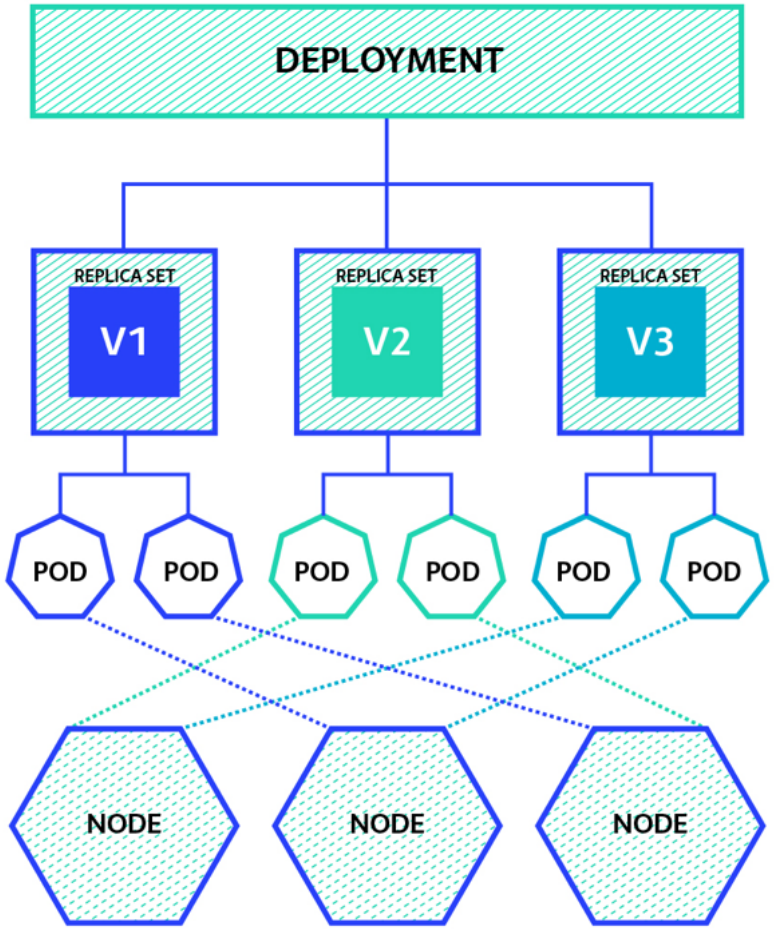
Creating Services

— From command line or yaml/json-files

```
$ kubectl expose deployment alpaca-prod --port=8080
service/alpaca-prod exposed
$
$ kubectl get service -o wide
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP PORT(S)    AGE       SELECTOR
alpaca-prod    ClusterIP 10.233.49.30 <none>      8080/TCP   3m5s      app=alpaca,env=prod,ver=1
bandicoot-prod ClusterIP 10.233.25.71 <none>      8080/TCP   2m49s     app=bandicoot,env=prod,ver=2
kubernetes     ClusterIP 10.233.0.1   <none>      443/TCP    21h       <none>

$ kubectl exec kuard nslookup alpaca-prod
Name:      alpaca-prod
Address 1: 10.233.49.30 alpaca-prod.default.svc.cluster.local
```

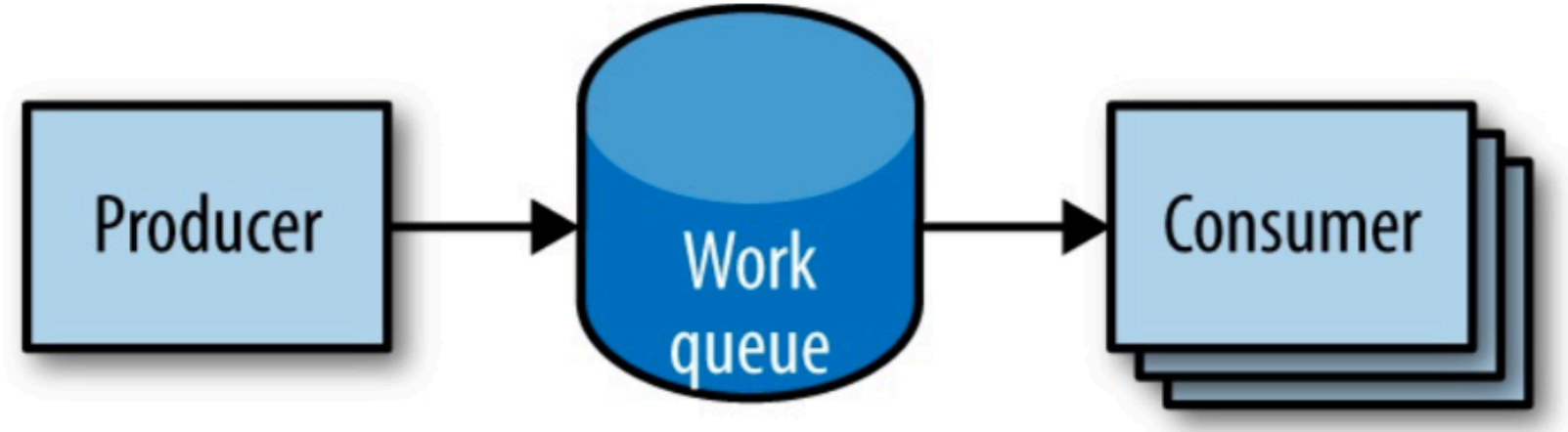
Managing multiple Pods with Controllers

- Deployment
  - Specify a desired number of replicas
  - Rolling updates
- Daemon Set
  - Deploy Pod onto each node (useful for e.g. log file handling networking)
- Stateful Set
  - Like a Deployment but
    - Stable, unique network identifiers
    - Stable, persistent storage
    - Ordered, graceful deployment and scaling
    - Ordered, automated rolling updates



Jobs

Type	Use case	Behavior	completions	parallelism
One shot	Database migrations	A single pod running once until successful termination	1	1
Parallel fixed completions	Multiple pods processing a set of work in parallel	One or more Pods running one or more times until reaching a fixed completion count	1+	1+
Work queue: parallel Jobs	Multiple pods processing from a centralized work queue	One or more Pods running once until successful termination	1	2+



Application versioning

There comes a time when a new version needs to be released

Usually with no service downtime

Want to test with new version works before going through with the full release

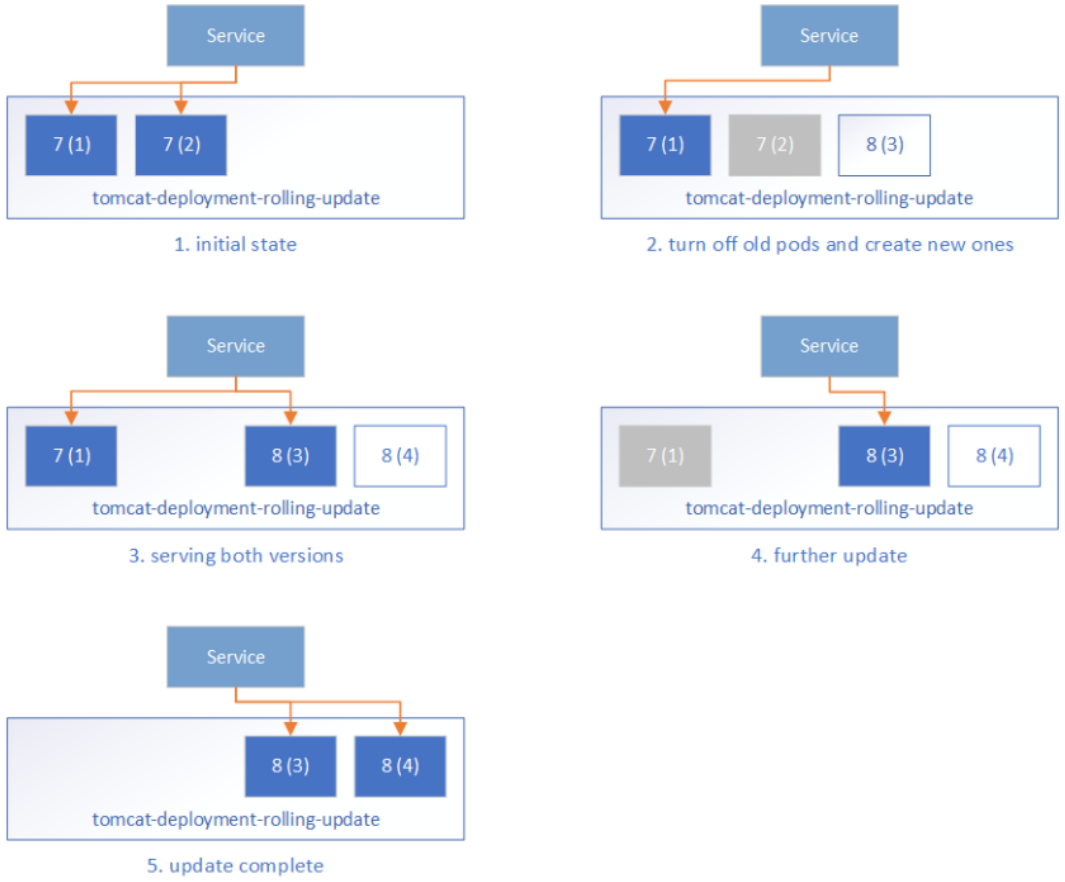
Kubernetes has an abstraction for this - Deployments

Deployment strategies – rolling update

Configured with max unavailable and max surge

max unavailable = number of pods that can be doing updates/rollbacks at a time, from the number of replicas

max surge = number of additional pods to be used for update/rollback



Manually scaling a deployment

- Command line

```
$ kubectl scale deployment nginx --replicas=10
```

- Or edit the deployment YAML file:

```
spec:
   replicas: 5
...
```

```
$ kubectl apply -f nginx-deployment.yaml
```

```
$ kubectl describe deployment nginx
Name:                   nginx
Namespace:              default
CreationTimestamp:      Wed, 10 Jul 2019 14:12:57 +0200
Labels:                 run=nginx
Selector:               run=nginx
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
OldReplicaSets:  <none>
NewReplicaSet:   nginx-6c847f87d9 (5/5 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  17m   deployment-controller  Scaled up replica set nginx-6c847f87d9 to 2
  Normal  ScalingReplicaSet  4m    deployment-controller  Scaled up replica set nginx-6c847f87d9 to 10
```

Horizontal Pod Autoscaling

Automatically shrink/increase Deployment on certain parameters (e.g. CPU load)

```
$ kubectl autoscale rs kuard --min=2 --max=5 --cpu-percent=80
```

Update Deployment

Edit nginx deployment file:

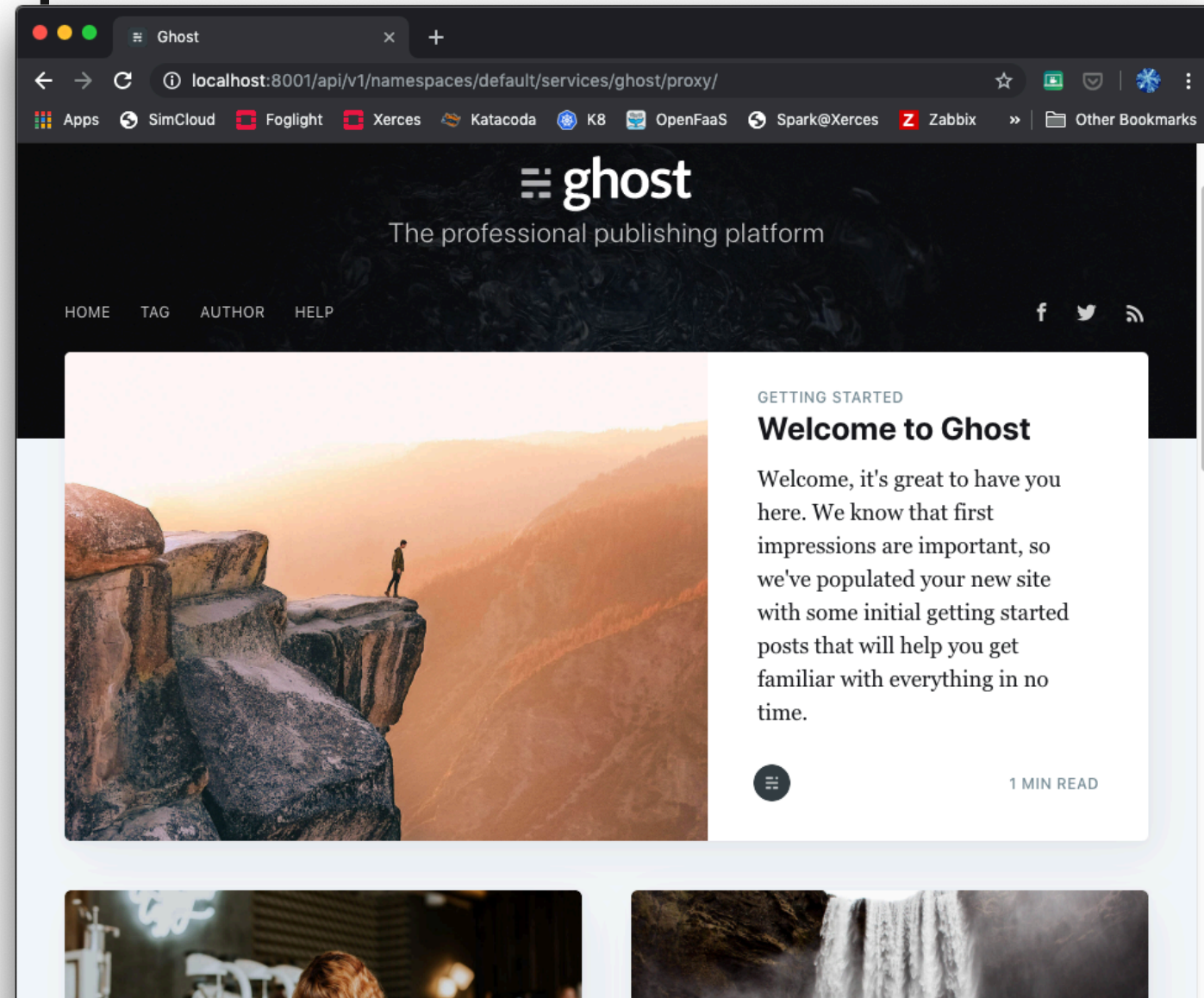
```
metadata
  annotations:
  kubernetes.io/change-cause: "Update nginx to 1.9.10"
...
     containers:
      - image: nginx:1.9.10
...
```

```
$ kubectl apply -f nginx-deployment-v2.yaml
deployment.extensions/nginx configured
$
$ kubectl rollout status deployment nginx
deployment "nginx" successfully rolled out

$ e$ kubectl get replicaset -o wide
NAME               DESIRED   CURRENT   READY     AGE       CONTAINERS   IMAGES
nginx-6548c7d877   5         5         5         6m14s     nginx        nginx:1.9.10
nginx-6c847f87d9   0         0         0         52m       nginx        nginx:1.7.12
```

Rollout magic

```
$ kubectl rollout history deployment nginx
deployments "nginx"
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         Update nginx to 1.9.10


$ kubectl rollout undo deployment nginx
deployment.extensions/nginx

$ $ kubectl rollout history deployment nginx
deployments "nginx"
REVISION  CHANGE-CAUSE
1         <none>
3         Update nginx to 1.9.10
4         <none>
```

Time for a real application



An open source blog engine

Available in a containerized format at docker.com

Now let's turn that into a kubernetes application

Configure Ghost

— Normally Ghost is configured with with a JavaScript file (ghost-config.js)

```
$ cat ghost-config.js
var path = require('path'),
    config;

config = {
    development: {
        url: 'http://localhost:2368',
        database: {
            client: 'sqlite3',
            connection: {
                filename: path.join(process.env.GHOST_CONTENT,
                                    '/data/ghost-dev.db')
            },
            debug: false
        },
        server: {
            host: '0.0.0.0',
            port: '2368'
        },
        paths: {
            contentPath: path.join(process.env.GHOST_CONTENT, '/')
        }
    }
};

module.exports = config;
```

— In Kunetes, we create a ConfigMap:

```
kubectl create configmap --from-file ghost-config.js ghost-config
```

Start Ghost

```
$ kubectl apply -f ghost.yaml

$ kubectl expose deployments ghost --port=2368

$ kubectl proxy
```

Now point your browser here:

http://localhost:8001/api/v1/namespaces/default/services/ghost/proxy/

```
$ cat ghost.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ghost
spec:
  replicas: 1
  selector:
    matchLabels:
      run: ghost
  template:
    metadata:
      labels:
        run: ghost
    spec:
      containers:
      - image: ghost
        name: ghost
        command:
        - sh
        - -c
        - cp /ghost-config/ghost-config.js
          /var/lib/ghost/config.js
          && docker-entrypoint.sh node current/index.js
        volumeMounts:
        - mountPath: /ghost-config
          name: config
      volumes:
      - name: config
        configMap:
          defaultMode: 420
          name: ghost-config
```

But, what if the node dies?

— Replace the internal sqlite database with a MySQL service instead.

```
$ cat ghost-config-mysql.js
var path = require('path'),
    config;

config = {
    development: {
        url: 'http://localhost:2368',
        database: {
            client: 'mysql',
            connection: {
                host: 'mysql'
                user: 'root'
                password: 'some-password-here'
                database: 'ghost_db'
                charset: 'utf8'
            },
            debug: false
        },
        server: {
            host: '0.0.0.0',
            port: '2368'
        },
        paths: {
            contentPath: path.join(process.env.GHOST_CONTENT,
        }
    }
};

module.exports = config;
```

Summary

- Kubernetes is a platform and orchestrator for deploying containers across multiple nodes
- Declarative model
- Pods are ephemeral
  - Data can be persistently stored if required
  - Controllers (Deployments, StatefulSet, DaemonSet) help manage life-cycle
- Services expose (multiple) Pods in a stable way