# Kubernetes Primer

## Kubernetes Background

Kubernetes is an application orchestrator.

It orchestrates containerized cloud-native microservices apps.

### Orchestrator

An orchestrator is a system that deploys and manages applications. It can deploy your applications and dynamically respond to changes.

### Containerized App

A containerized application is an app that runs in a container.

Before containers, apps ran on phtsical servers or virtual machines. Containers are the next iteration of package and ru napps.

- Faster, lightweight, suited to business reqs than servers and VMs
- Noew apps run in containers in could-native ecosystems

### Cloud Native

A cloud-native app is designed to meet cloud-like demands of

- auto-scaling
- self-healing
- rolling updates
- rollbacks

Cloud-native apps are not just fot public cloud. The y can also run anywhere that have Kubernetes. Cloud native is about the way applications behave and react to events

### Microservices Apps

A microservices app is built from lots of small, specialised, independient parts that work together to form a meanigful app.

Each individual services is called a microservice. Web frontend, catalog, shopping cart, authentication, logging, persistent store, etc. Each microservice runs as one or more containers.

### Kubernetes

Kubernetes deploys and manages (orchestrates) applications that are packaged and run as a containers (containerized) and that are built in ways (cloud-native microservices) that allow them to scale, self-heal and be updated in-line with modern cloud-like requeriments.

### History

Google created a new platform called Kubernets that it donated it to the newly formed Cloud Native Computing Fundation CNCF in 2014 as an open-source project.

Kubernetes enables abstracts underlying infrastructure and it makes it easy to move applications on and off clouds. Kubernetes in 2014 has become the most important cloud-native technology on the planet.

- Written in Go
- Built in GitHub
- Discussed on Twitter @kubernetsio and slack.k8s.io

### Kubernetes and Docker

Docker builds applications into containeri mages and can run them as containers.

Kubernets can't do either of those. Instead, it sits a higher level and orchestrates things.

Kubernetes make high-level orchestration decisions such as which nodes should ru nthe containers. The Docker runtime is bloated and overkill for what Kubernetes needs. The Kubernetes project began work to make the container runtime layer pluggable so that users could choose the best container runtime for their needs.

In 2016 Kubernetes introduced the container runtime interface (CRI) that made this cntaoiner runtime layer pluggable. In 2020 Kubernetes deprecated the Docker runtime.

Containerd has replaced Docker as default container runtime in most. Containerd is a stripped-down version of Docker optimized for Kubernetes. All container images created by Docker will work on Kubernetes.

Develop teams uses Docker to build images and operations teams uses Kubernetes to run them.

## Kubernetes as OS of the Cloud

In many ways, Kubernetes is like an OS for the cloud.

- You install traditional Linux on a server, and it abstracts server resources and schedules application processes.
- You install Kubernetes on a cloud, and it abstracts cloud resources and schedules application microservices.

In the same way Linux abstracts the hardware differences between server plataforms, Kubernetes abstracts the differences between different private and public clouds.

Kubernetes is a major step towards a true hybrid cloud. allowing to seamlessly move and balance workloads across multiple different public and private cloud infrastructures.

You can also migrate to and from differente clouds, meaning you can choose a cloud today and not have to stick with that decision for the rest of your life.

### Application Scheduling

As a OS abstracts a specific computer resources (CPU, memory, storage, networking) Kubernetes does a similar thing with cloud and datacenter resources.

# Kubernetes Principles of Operation

Overview of the major concepts.

## Kubernetes from 40K feet

### Kubernetes as a Cluster

K8s is like any other cluster, a bunch of machines to host apps. Kubernetes cluster is made of a control plane and worker nodes.

- Control plane
  - Exposes the API
  - Scheduler for assigning work
  - records the state of the cluster and apps in a persistent store
- Worker Node: Where the app runs

### Kubernetes as a Orchestrator

1. Start with an app

2. Package it as a container

3. Give it to the cluster (Kubernetes)

   Already control plane nodes implement the cluster intelligence and worker nodes are where user applications run

   1. Design and wirte the application as small independent microservices in your favorites languages
   2. Package each microservice as its own container
   3. Wrap each container in a Kubernetes Pod
   4. Deploy pods to the cluster via higher level controllers such a devployments, demonSets, StatefulSets, Cronjobs, etc

At highter level, kubernetes has several controllers that augment Pods.

## How it works

- The cluster
  - One or more **control plane node**
  - a bunch of worker nodes

1. Design and write the app as small independent microservices
2. Package each microservice as its own container
3. Wrap each container in a **Kubernetes Pod**
4. Deploy Pods to the cluster via higher level controllers )Deployments, DaemonSets, StatefulSets, CronJobs)

- Kubernetes manage applications declaratively (a set of config file)

# Control Plane and worker nodes

CP and WN are Linux Hosts: bare metal servers or instances in a private or public cloud, ARM or IoT devices.

## Control Plane

**Kubernetes Control Plane Node** runs a collection of system services that make up the control plane of the cluster.

- Called Heads or Head Nodes.

Simple setups run a single control plane node. For prod envs, multiple CP nodes are configured. 3 or 5 is recommended.

**Good practice:** not run user apps on CP nodes. This frees to concentrate entirely on managing the cluster

#### API Server

API Server is the Grand Central Station of Kubernetes.

- ALL COMMUNICATION between all components must go through API server

It exposes RESTful API to POST YAML configuration files to over HTTPS, sometimes called **manifests**.

The Manifests describes desired app state: container image, ports to expose, Pod replicas to run, etc.

Requests to the API server are subject to authentication and authorization checks.

#### Cluster Store

The only **stateful** part of the control plane.

- Persistently stores the entire configuration and state of the cluster
- Based on **etcd** distributed database. 3-5 etcd replicas for high availability
- Default config stores a replica of the cluster store on every control plane node and automatically configures HA.

#### Controller Manager and Controllers

Controller Manager implements all background controllers that monitor cluster components and **respond to events**

- The CM is a controller of controllers
  - Deployment Controller
  - StatefulSet Controller
  - ReplicaSet Controller
- GOAL OF EACH CONTROLLER: ensure the **observed state** matches **desired state**

1. OBtain desired state
2. Observe current state

3. Determine differences
4. Reconcile differences

**Scheduler**

Scheduler watches the API server for new work tasks and assigns them to appropiate healthy worker nodes.

When identifying nodes capable of running a task, performs various predicate checks.

- affinity rules
- required network ports availables on the node
- sufficient available resources
- etc

If does not find a suitable node, the task is not scheduled and gets marked as **pending**

**Cloud Controller Manager**

Control Plane will be running a **cloud controller manager** to facilitate integrations with cloud services.

- Instances
- Load balancers
- storage

### Worker Nodes

Worker nodes are where user app runs.

1. Watch the API server for new work assignments
2. Execute work assignments
3. Report back to the control plane via API Server

**Kubelet**

- Main Kubernetes Agent
- Runs on every worker node
- When you join a node to a cluster, the process installs the kubelet
- Responsible for registering it with the cluster
- Registers CPU, memory, storage into the cluster pool
- Watch the API server for new work tasks
- **can not execute a tasks/ Simply reports back to the control plane**

**Container Runtime**

- Performs container-related tasks like pulling images, start or stop containers
- CRI (Container Runtime Interface)
- containerd (container-dee) is the container supervisor and runtime logic stripped out from Docker.
- Donated by Docker

**Kube-proxy**

- Runs on every node
- Responsible for local cluster networking
- Ensures each node gets its own unique IP Address
- Implements local iptables
- handle rounting
- handles load balancing and traffic on the Pod network

### Kubernetes DNS

- Kubernetes has an internal DNS service
- Cluster's DNS service has a static IP address that is hard-coded into every Pod on the cluster.
- Service registration is automatic
- Based on CoreDNS project

## Packagking Apps for Kubernetes

An application needs:

1. Packaged as a container
2. Wrapped in a Pod
3. Deployed via a declarative manifest file

- You write an app microservice
- You built it into a container image and store in a registry -> containerized
- You define a Kubernetes Pod to run the containerized app
  - a **Pod** is a wrapper that allows a container to run on a Kubernetes Cluster.
  - \ The preferred model is to deploy all Pods via highter-level controllers
    - Most common controller is Deployment.
    - Deployment Offers scalability, self-healing, rolling updates for stateless apps
    - Deployments are definedd in YAML manifest files.
- Once defined in Deployment YAML file, you can use Kubernetes command-line tool to post it to the API server as the desired state of the app.

## Declarative model and desired state

**The declarative model and the concept of desired state are at the very heart of Kubernetes**.
Kubernetes declarative model works like:

1. Declare the **desired state of an app microservice in a manifest file**
2. Post it to the API server
3. Kubernetes stores it in the cluster store as the application's desired state
4. Kubernetes implements the desired state on the cluster
5. A controller makes sure the observed state of the application does not vary from the desired state

- Manifest files are written in a YAML and tell kubernetes what an app sohuld look like -> **desired state**.
  - Which image to use, replicas, network ports, updates, etc

- Post to the API via `kubectl` utility via HTTP POST (usually port 443)
- Kubernetes authenticates and authorizates the request. Inspects the manifest, identifies which controller to send, record the config in the cluster store.
- Schedules the work to worker nodes where kublet co-coordinates the work
- Controllers run as background reconciliation loops that constantly monitor the state of the cluster

# Pods

Atomic Units:

- VMware: Virtual Machine
- Docker: Container
- Kubernetes: Pod

1. Kubernetes demands that every container runs inside a Pod.
2. Pods are objects in the Kubernetes API.

## Pods & Containers

*Pod therm comes from a pod of whales*. The simplest: run a single container in every Pod. There are advanced user-cases that run multiple containers in a single Pod. The point **Kubernetes Pod is a construct for running one or more containers.**

## Anatomy

- Multiple containers in a Pod: there all share Pod environments
  - network stack
  - volumes
  - IPC namespace
  - shared memory
  - same IP address
  - etc
- If two containers in the same opd neet to talk to each other they can use the Pod's localhost interface.

__Multi container pods are ideal for tightly coupled containers that may need to share memory and storage.__Put each container in a single pots keeps things clean but increment the traffic between pods.

To scale an app do not scale by adding more containers to existing pods. Multi-Container pods are ONLY for situations where complementary containers need to **share resources**.

- atomic ops: Deployment of a Pod is an atomic operation. A single Pod ONLY can be scheduled to a single node. Pod is only ready for service when all its containers are up and running.

**Lifecycle**

Pods are mortal. If a Pod dies unexpectedly, Kubernetes starts a new one in its place. New pods feels like the old one, but with a new ID and IP address.

You have to desgin the apps kowning this implications.

**Immutability**

Can't be changed once the are running. To change a config, you replace it with a new one running the new configuration.

# Deployments

The Pod deployment is via higher level controllers as **Deployments, DaemonSets, StatefulSets.**

Deployment is a Kubernetes object that wraps around a Pod and adds features such self-healing, scaling, zero-downtime rollouts, and versioned rollbacks. Controllers run as watch loops observing the cluster, matching desired and observed states.

# Service Objects and Networking

Rollouts and scaling ops replace old Pods with new ones with new IPs. *Services* provide relaiable networking for a set of Pods. Service is a Kubernetes API object, privdes reliable name and IP and load balancing to the microservices Pods behind it.

like Pods and Deployments, Services are a fully-fledged object in the Kubernetes API. They have a front-end consisting of stable DNS name, IP address, and port. On the backend, they load-balance traffic across a dynamic set of Pods.

Service is a stable network abstraction point that provides TCP and UPD load-balancing across a dynamic set of Pods. So they don;t poseess application intelligence. The cannot provide application-layer host, and path routing.

Ingress understands HTTP and provides host and path-based routing.

# Getting Kubernetes

## 1. Playgrounds

- easiest way
- not for production
- Play with Kubernetes, Docker Desktop, Minikube, k3d, etc.

## 2. Hosted Kubernetes

- Cloud platforms offer hosted Kubernetes service
- Not all services are equial
- Great on-ramp to lets you to focus on your applications
- Close to a zero-effort production-grade Kubernetes cluster.
- Example: Google Kubernetes Engine:
  - deploy
  - high performance
  - highly available
  - security best-practices out-of-the-box
- AWS: Elastic Kub Service EKS
- Azure: Azure Kub Service AKS
- Civo CLoud Kubernetes
- DigitalOcean: DOKS
- Google CLoud Platform: Google Kub Engine GKE
- Linode: Linode Kubernetes Engine LKE

## 3. DIY Kubernetes Clusters

- Hardest way to get Kubernetes Cluster
- build by yourself
- Most flexibility and control

## Playground Example: Play With Kubernetes

- Quick and simple.
- Requeriments: computer, internet connection, DockerHub and GitHub account.
- suffer from capacity and performance issues
- It last four hours
- FREE

1. labs.play-with-k8s.com
2. ADD NEW INSTANCE
3. This is a Kubernetes node
4. Run commands to see components pre-installed on the node

```
docker --version
kubectl version --output=yaml
```

5. Run prpovided `kubeadm init` to initialize a new cluster.

```
kubeadm init --apiserver-advertise-address $(hostname -i) --pod-network-cidr...
```

The node is initialized as the control plane node. The output of kubeadm init give a list of commands, buy PWK has already configured.

6. Verify de nodes `kubectl get nodes`. The status NotReady is because the missing configured Pod Network.
7. Initialice Pod network (cluster networking). copy from kubeadm init the `kubetctl apply -f https....`.
8. verify the cluster to see status Ready
9. copy the `kubeadm join` from command kubeadm init
10. join command includes the cluster join-token, IP socket, API server.
11. Add new instance
12. Paste the `kubeadm join` command into the terminal of node2
13. `kubeadm join --token ...`
14. Switch back to the node1 and run `get nodes`

node1 was initialized as control plane node and additional nodes will join as worker nodes.

## 2. Google Kubernetes Engine GKE

GKE is a hosted Kubernetes service that runs on the Google Cloud Platform

- Fast and east way to get a production-grade Kubernetes cluster
- Managed control plane
- Itemized billing

## kubectl

- `kubectl` is the main Kubernetes command-line tool.
- converts commands to HTTP REST requests with JSON content required by the Kubernetes API server.
- uses a config file to know wich cluster and API endpoint to send commands to
- config file is on `$HOME/.kube/config`.
  - defines Clusters, Users credentials and Contexts.

Clusters is the list of Kub clusters. It makes it possible for a single kubectl workstation to manage multiple clusters: name, certificate info, API server endpoint

Users define users access on each cluster, as dev or ops users with different permissions: name, username, credentials.

Contexts group clusters and users undera friendly name

```
kubectl config view
```

```
apiVersion: v1
kind: Config
clusters:
  - cluster:
  name: shield
    certificate-authority: C:\Users\nigel'.minikube/ca.crt
    server: https://191.168.1.77:8443
users:
- name: coulson
  user:
    client-certificate: ...
    client-key: ...
context:
- context:
    cluster: shield
    user: coulson
  name: director
current-context: director
```

```
kubectl config current-context
kubectl use-context holamundo
>> switched to context "holamundo"
```

# 04. Working With Pods

## Theory

Controllers infuse Pods with self-healing, scaling, rollouts and rollbacks. Every controller has a PodTemplate defining the Pods.

### Pod Theory

- Atomic unit of shceduling in Kuerbetes: Apps are deploying inside Pods

1. Write the app/code
2. Package it as a container image
3. Wrap the container image in a Pod
4. Run it on Kubernetes

- Pods augment containers
  - Labels
  - Restart policies
  - affinity
  - security policies
  - termination control
  - resource limites

```
# List all possible Pod attributes
kubectl explain pods  --recursive
kubectl explain pod.spec.restartPlicy
```

- **Labels** Let group Pods and associate them with other objects in powerful ways.
- **Annotations** let add experimental features and integrations with 3rd partyh tools
- **Probes** let test the health and status of Pods and apps
- **Affinity** give control over where in a cluster Pods are allowd to run
- **Termination** lets you gracefully terminate Pods and apps
- **Security policies** let enforce security features
- **Resource Requests** let specify a minimum and maximum values for CPU, mem, disk IO, etc. usain bolt

**EVERY CONTAINER IN A POD IS GUARANTEED TO BE SCHEDULED TO THE SAME WORKER NODE**

**PODS PROVIDED A SHARED EXECUTION ENVIRONMENT FOR ONE OR MORE CONTAINERS**:

- Shared filesystem, network stack, memory, volumes.

### Deploy Pods

1. Pod manifest: Static Pods
2. Controller

static Pods have no super-powers as self-healing, scaling or rolling updates. They are monitored and managed by the worker node kubelet process. There's no control-plane process watching and capable of starting a new one on a different node.

Pods deployed via controllers have benefits of being monitored and managed by hihgly available controlle rrunning. If kubelet restart fails, the controller can start a replacement pod.

- Pods are mortal: There's no fixing them and bringing back from the dead. Pods are cattle (pets vs cattle paradigm). When die they get replaced by another. Same config, different IP address and UID. This is why apps store **state** and **data** outside the Pod.
- You almost always deploy and manage Pods via controllers
- The single-container model is the simplest, but multi-container pods are importal in production environments and vital for service meshes.

### Deploying Pods

1. define it in YAML *manifest file*
2. Post the YAML to the API Server
3. The API server authenticates and authorizes
4. The config is validated
5. The scheduler deploys the Pod to a healthy worker node with available resources
6. The local kubelet monitors it

**If the Pod is deployed via a controller, the config will be added to the cluster store as desired state and controller will monitor it.**

## anatomy of a Pod

- Pod: **execution environment shared by one or more containers**. It can be useful to think Pods as shared environments and containers as application processes.
- In Docker or containerd a Pod is a special container with containers running inside.
- The collection of resources that containers it runs inherit and share are:
    - net namespace:
        - IP Address
        - Port Range
        - Routing Table
        - etc
    - pid namespace
    - mnt namespace: filesystems and volumes
    - UTS namespace: Hostname
    - IPC namespace> unix domain sockets and shared memory

### Shared Networking

Each Pod creates its own network namespace. Own P, single range of TCP and UDP ports, single rounting table. If it's a single container Pod the container has full access to the network namespace. If it's a multi container Pod, all containers share the IP, port range and rounting table.

### Pod Network

The unique IP address that Pods gets are fully routable on an interneal network called pod network.

- flat: every pod can talk directly to every other Pod without ocmplex routing
- Flat security perspective: Kubernetes Network Policies to lock down access

### Pods Features

- Pod deployment is an atomic operation
- Defined in a declarative YAML object yo post to the API server
- Once all containers on a Pod are pulled and running, the Pods enters the running phase
- If it's a short-lived Pod, when all containers erminate succesfully the Pod itself terminates and enters the succeeded state. -> Never or OnFailure restart policy
- If it's a long-lived Pod, the local kubelet may attempt to restart them. -> Always restart policy
- Immutability: uou cant modify them after they are deployed

#### Immutability Behaviors

- When updates are needed -> replace al old Pods with new ones
- When failures occur, replace failed Pods with new ones

### Scaling

Multi containers pod are only for coscheduling and colocating containers, you never scale an app adding more of the same app containers to a Pod.

## Multi-Container Pods

At very high-level, every container should have a single clearly defined responsibility. This is called separation of concerns. Colocating multiple containers in the same Pod allows containers to be designed with a single respondiblity but cooperate closely with others. For example, a container that puts conttent updates in a volume shared with web container. The Multiple Container Pod patterns are:

- Sidercar pattern
- Adapter pattern
- Abassador pattern
- Init pattern

### Sidecar

- Most popular
- Main app container
- Sidecar container: augment or perform a secondary task for the main app container.

### Adapter

Adapter pattern is a specific variation of the sidecar pattern where the helper container takes non-standarized output from the main container and rejis it into a format required by an external system. A example is NGINX logs being sent to Prometheus, who does not understand NGINX logs. A common a pproach is put an adapter container into NGINX Pod that converts NGINX logs into a format accepted by prometheus.

### Ambassadorn

Variation of the sidecar. Helper container brokers connectivity to an external system. Example the main app container can dump its output to a port the ambassador container is listening. Looks like a interface.

### Init

Init is NOT a form of sidecar. Runs a init container that's guaranteed to start and complete before the main app container. It's guaranteed to only run once. For example, the main app need permissions settings, an external API to be up and accepting connections.

---

## Hands On with Pods

```
kubectl get pods
```

Pod manifest file

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: hello-pod
  labels:
    zone: prod
    version: v1
spec:
  containers:
  - name: hello-ctr
    image: nigelpoulton/k8sbook:1.0
    ports:
    - containerPort: 8080
```

- kind: type of object
- apiVersion: schema version
  - `<api-group>/<version>`
  - core group: omits the api-group part
  - Pods in the core API omits the group name so just v1 its ok
- metadata:
  - names: identify the object in the cluster
  - labels: loose couplings with other objects
  - annotations: 3rd party tools and services
  - Namespace: no specify Namespace -> default Namespace. Not good practiceuse default in RW apps
- spec: define the containers the Pod will run

### Manifest files

Forces to describe applications in Kubernetes. Can be used by operations team to understand how the application works and what it requires from the environment.

### Deploying Pods from a manifest file

```
kubectl apply -f pod.yml
pod/hello-pod created
kbectl get pods
```

**kubectl get**

- `-o wide` single line
- `-o yaml` full copy
  - desired state .spec
  - observed state .status
  - the extra information come from default values

**kubectl describe**

nicely formatted multi-line overview of an object

```
kubectl describe pods hello-pod
```

**kubectl logs**

```
kubectl logs

multipod container:
kubectl logs multipod --container syncer
```

**kubectl exec: running commands in Pods**

```
kubectl exec hello-pod -- ps aux
```

get shell access in a running Pod

```
kubectl exec -it hello-pod -- sh
curl localhost:8080
```

**hostnames**

Every container inherits its hostname from the name of the Pod

#### kubectl edit

```
kubectl edit pod hello-pod
```

#### multi container pod

```
apiVersion: v1
kind: Pod
metadata:
  name: initpod
  labels:
    app: initializer
spec:
  initContainers:
  - name: init-ctr
    image: busybox
      command: ['sh', '-c', 'until nslookup k8sbook; do echo waiting for
k8sbook service;\
      sleep 1; done; echo Service found!']
    containers:
    - name: web-ctr
      image: nigelpoulton/web-app:1.0
      ports:
        - containerPort: 8080
```

deploy

```
kubectl apply -f initpod.yml
pod/initpod created

kubectl get pods --watch
```

the init:0/1 status tells that zero out of one init containers has completed succesfully. The pod will ramain in this phase until k8sbook service is created.

```
kubectl apply -f initsvc.yml
kubectl get pods --watch
```

#### sidecar container

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: git-synx
  labels:
    app: sidecar
spec:
  containers:
  - name: ctr-web
    image: nginx
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/
  - name: ctr-sync
    image: k8s.gcr.io/git-sync:v3.1.6
    volumeMounts:
    - name: html
      mountPath: /tmp/git
    env:
    - name: GIT_SYNC_REPO
      value: https://github.com/nigelpoulton/ps-sidecar.git
    - name: GIT_SYNC_BRANCH
      value: master
    - name: GIT_SYNC_DEPTH
      value: "1"
    - name: GIT_SYNC_DEST
      value: "html"
      epmtyDir: {}
```

the second container it watches a GitHub repo and syncs changes into the same shared volume. If the contents of the GitHub repo change, sidecar will notice and copy the new content into the shared volume where the web container will notice and update the web page.

### Clean-up

```
kubectl delete pod git-sync
pod "git-sync" deleted
```

# 05: Virtual Clusters with Namespaces

**Namespaces are a native way to divide a single Kubenetes cluster into multiple virtual clusters.**

## Use cases for Namespaces

- Easy way to apply quotas and policies to groups of objects
- If no target Namespace is spacified the default Namespace will be used to deployment.
- `kubectl api-resources`
- a good way of sharing a single cluster among different departments and environments. For example Dev, Test and QA, each one with they own sets of users and permisions, and unique resource quotes

Namespaces ar not good for isolating hostile workloads. To **strong isolation** the correct method is to use multiple clusters.

Every Kubernetes cluster has a set of a pre-created Namespaces.

```
kbuectl get namespaces
NAME STATUS AGE
default ACTIVE ...
kube-system ...
kube-public ...
kube-node-lease ...
```

- default: newly crated objects with no specified Namespace
- Kube-system: for DNS, metrics server and control plane components
- Kube-public: for objects that need to be readable by anyone
- kube-node-lease: node heartbeat and managing leases

```
kubectl describe ns default
kubectl describe ns -n my-namespace
```

## Creating and Mananging Namespaces

```
git clone https://github.com/nigelpoulton/TheK8sBook.git
cd namespaces
```

```
kubectl create ns hydra
namespace/hydra created
```

```yaml
# shield-ns.yml
kind: Namespace
apiVersion: v1
metadata:
    name: shield
    labels:
        env: marvel
```

```
kubectl apply -f shield-ns.yml
namespace/shield created
```

```
kubectl get ns

kubectl delete ns hydra
```

**Configuring kubectl to use a specific NS**

```
kubectl config set-context --current --namespace shield
```

### Deploying to Namespaces

- imperative method: required to add -n or --namespace flag to commands.
- declarative method: specifies the Namespace in YAML manifest file.

```yaml
apiVersion: v1
kind: ServiceACcount
metadata:
    namespace: shield
    name: default
---
apiVersion: v1
kind: Service
metadata:
    namespace: shield
    name: the-bus
spec:
    ports:
    - nodePort: 31112
      port: 8080
      targetPort: 8080
    selector:
      env: marvel
---
```

```
apiVersion: v1
kind: Pod
metadata:
    namespace: shield
    name: trisklion
```

```
kubectl apply -f shield-app.yml
serviceaccount/default ocnfigured
service/the-bus configured
pod/triskelion created
```

```
kubectl get pods -n shield
kubectl get svc -n shield
```

```
curl localhost:31112
```

## Clean-up

```
kubectl delete -f shield-app.yml
kubectl delete ns shield

kubectl config set-context --current --namespace default
```

# 06: Kubernetes Deployments

- Components to Deployments

1. the spec
2. the controller

Spec is declarative YMAL object. Describe desired state of a stateless app

Deployment controller implements and mages it. Controller operates as a background loop

1. start with a stateless app
2. package it as a container
3. define it in a Pod template

This stateless app doesn't scale, self-heal, updates or rollbacks autmatically.

4. Wrap them in a Deployment object

The container holds the application, the Pod augments the container with labels, annotations and other metadata, and **Deployment further augments things with scaling and updates**.

**POST the deployment object to the API Server where Kubernetes implemetns it and the Deployment controller watches it.**

- A deployment object only manages a single Pod template.
- A deployment can manage multiple replicas of the same Pod.

#### ReplicaSets

- Not recommended manage ReplicaSets directly -> Manage via Deployment controller
- CONTAINERS: Great way to package apps and dependencies
- PODS: allow containers to run on Kubernetes and enable schaduling and other stuff.
- ReplicaSets: Manage Pods and bring self-healing and scaling.
- Deployments: Manage ReplicaSets and add rollouts and rollbacks.

## Self-healing and Scalability

- CONTAINERS: Co-locate containers, share volumes, memory, simple networking, etc.

If NODE fails, the POD IS LOST. Deployments:

- self-healing: replace Pods at fail
- scaled: Increase or decrease load

## All About the STATE

- Desired state
- Observed State
- Reconciliation

Imperative model has no concept of desired state, it's just a list of steps and structions. Kubernetes prefers declarative model.

### Reconciliation

- ReplicaSets are implemented as a controller running as a background reocncilation loop checking the right number of Pod replicas are present on the cluster.

### Rolling updates with deployments

- Zero-downtime rolling-updates (rollouts) of stateless apps
- Requerimients from microservices apps:
  - Loose coupling via APIs
  - Backwards and forwards compatibility

Each Deploymenmt describes:

- How many Pod replicas
- What images to use for Pod's containers
- What network ports to expose
- Details about how to perform rolling updates

**The repolica seets enters on a watch loop suring observed state and desired state are in agreement. A deployment objects sits above the RSet, governing its configuration and providing mechanisms for rollouts and rollbacks.**

New configuration YAML files creates new replicaSets than controlls new Pods with the updated configuration. When old config Pods are zero, all new pods has the new configuration. Old replicaSets config still remains for rollbacks to reverting to previous versions. This process is the oposite to the described rollout.

## Create a Deployment

YAML snippet from deploy.yml file. Defines a single-container Pod wrapped in a Deployment object.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
  replicas: 10 # Number of Pods to deploy/mng
  selector:
    matchLabels:
      app: hello-world
  revisionHistoryLimit: 5
  progressDeadlineSeconds: 300
  minReadySeconds: 10
  strategy: # Controls how updates happen
    type: RollingUpdate
      maxUnavailable: 1
      maxSurge: 1
  template: # Pod template
```

```yaml
    metadata:
    labels:
    app: hello-world
  spec:
    containers:
    - name: hello-pod
    image: nigelpoulton/k8sbook:1.0
    ports:
    - containerPort: 8080
```

- spec.selector is a list of labels that pods must have in order for the deployment to manage them. The Deployment selector matches the labels assigned to the Pod lower in the Pod template
- spec.revisionHistoryLimit How many older versions / replicaSets t keep.
- spec.progressDeadlineSeconds How long to wait during a rollout for each new replica to come online in minutes. The clk is rst for each new repolica.
- spec.strategy how to update the Pods when a rollout occurs

```
# explore
kubectl get deploy hello-deploy

kubectl describe deploy hello-deploy
```

Deployments automatically create associated ReplicaSets

```
kubectl get rs
```

Toe get detailed information about the replica set use `kubectl describe rs hello-deploy-XXXXXXXXXXXX`

### Accessing the App

To access to all the replicas of the application you need a Kubernetes Service object.

The YAML defines a Service that works with Pod repolicas depoloyed.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
  labels:
    app: hello-world
  spec:
    type: NodePort
    ports:
    - port: 8080
      nodePort: 30001
```

```
    protocol: TCP
  selector:
    app: hello-world
```

```
$ kubectl apply -f svc.yml
servic e/hello-svc created
```

With a Service deployed, it can access tha app by hitting any of the cluster nodes on port 30001.

### Scalling Operations

Imperatively

```
kubectl scale
```

declaratively updating a YAML file and re-posting it to the API server.

```
kubectl scale deploy hello-deploy --replicas 5
```

The current state no longer matches the manifest. This can cause issues. __You should always keep the YAML manifests in sync with live enviroments. -> the correct wayy, edity YAML file, set a different number of replicas and run `kubectl apply -f deploy.yml`

## Rolling Update

Rollout: a new version of the app has already been created and containerized as an image with the 2.0 tag... You have to perform a rollout. We will ignore real-world CI/CD workflows and version control tools.

1. Update the image version in the Deployments resource manifest.

```
nigelpoulton/k8sbook:1.0
nigelpoulton/k8sbook:2.0
```

The .spec section contains settings governing the updates

```
revisionHistoryLimit: 5
progressDeadlineSeconds: 300
minReadySeconds: 10
strategy:
  type: RollingUpdate
  rollingUpdate:
```

```
    maxUnavailable: 1
    maxSruge: 1
```

- revisionHistoryLimit: keep 5 previous releases. This means the previous 5 ReplicaSet objects will be kept to easily rollback.

- progressDeadlineSeconds: 5 minute windows

- minReadySeconds: rate at which replicas are replaced. The example tells that any new replica must be up and running for 10 serconds without issues before replace the next one in sequence. **In real world the value must be large enought to trap common failures**.

- strategy:

    - Rolling update strategy
    - Never have more than one Pod below desired state -> maxUnavailable -> NEVER HAVE MORE THAN 11 (if 10 desired)
    - Never have more than one Pod above desired state. -> maxSurge -> NEVER HAVE LESS THAN 9 REPLCAS (if 10 desired)

The Deployment file has a selector block. This is a list of labels the Deployment controller looks for when finding Pods to update during rollout operation.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  selector: # THE DEPLOY MANAGE
    matchLabels: # ALL REPLICAS CON THE CLUSTER
      app: hello-world # WITH THIS LABEL
...
```

```
kubectl apply -f deploy.yml
deployment.apps/hello-deploy configure
```

```
kubectl rollout status deployment hello-deploy
>> Waiting for depoyment "hello-deploy"
>> rollout... 4 out of 10 new replicas...
>> ...
```

```
kubectl get depoloy hello-deploy
```

```
kubectl rollout pause deploy hello-deploy

kubectl describe deploy hello-deploy

kubectl rollout resume depoloy hello-deploy

kubectl get deploy hello-deploy
```

## Rollback

```
kubectl rollout history deployment hello-deploy
REVISION CHANGE-CAUSE
1        <none>
2        kubectl apply --filmename-deploy.yml

kubectl get rs
NAME  READY AGE DESIRED CURRENT

kubectl describe rs hello-deploy-XXXXXX
```

A rollback follows the same logic and rules as an update/rollout.

```
kubectl rollout undo deployment hello-deploy --to-revision=1
```

The rollback was imperative. The current state of the cluster no longer matches with sou8rce YAML files. This is a fumental flaw with the imperative appoach and a major reason to only uupdate the cluster declaratively by updating YAML files and reposting them.

## Clean-up

```
kubectl delete -f deploy.yml
kbuectl delete -f svc.yml
```

# 07: Services

- Services provides stable and reliable networking for a set of unreliable Pods

Failures, scaling-ups, rollbacks, rollouts introduces new Pods with new IP address. **This creates massiva IP churn and demostrates why never connect directly to any Pod.**

- FIRST: K8s **Services** is an object in Kubernetes that provid3es stable **networking for Pods**. Like Deployment, Pod and ReplicaSet is a REST object in the API that is **defined in a manifest file and post to the API server.**

- SECOND: Every Service gets its own
  - **stable IP address**
  - **stable DNS name**
  - **stable port**
  - also load balanced traffic to Pods

- THIRD: Servces use **labels** and **selectos** to select the Pods to send traffic.

With Service, clients can access to the service without interruption due to Pods.

### Labels

Services match with Pods via labels and selectors. The logic is an logic AND between the Service labels and Pod labels.

```yaml
apiVersion: v1
kind: service
metadata:
  name: hello-svc
  spec:
    ports:
    - port: 8080
    selector:
      app: hello-world # Label 1
      env: tkb
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 10
  <snip>
  template:
    metadata:
      labels:
```

```
        app: hello-world
        env: tkb
    spec:
      containers:
<Snip>
```

The Service is selecting on Pods with both labels. It makes no difference if the Pods have additional labels.

If the Deployment has a Pods template with the same labels, any Pods it deploys will match the Service's selector and receive traffic from it.

- When a Service is created -> Kubernetes automaticaly creates associated **Endpoint** object than stores dunamic list of healthy Pods matching the Service's label selector.

**LEER EndopointSlices que estan reemplazando a los Endpoints**.

## Dual Stack Networking

Pods and Services can have IPv4 and IPv6 in dual stack where IoT apps are supported.

### Accesing Services from inside te cluster

- Default cluster: **ClusterIP**

*ClusterIp* Service has a stable virtual IP address that **only accessible from inside the cluster**. It's programmed into the internal network fabric and guaranteed to be stable for the life of the service.

Eveyr Service created gets a ClusterIP that's registered in the cluster's internal DNS service. ALl Pods in the cluster are pre-programmed to use the cluster's DNS service.

So if a Pod knows the name of a Service it can resolve it to a ClusterIP address and connect to the Pods behind it. It does not work outside of the cluster.

### Accessing from outside the cluster

Two types of Service for requests from outside the cluster

#### NodePort

NodePort Service build on top of the clusterIP -> Allow external clients to hit a dedicated port on every cluster node and reach the Service. This type of service add to ClusterIP an additional NodePort that can be used to reach the Service from outside the cluster.

```
apiVErsion: v1
kind: Service
metadata:
  name: skippy
spec:
  type: NodePort
  ports:
  - port: 8080
```

```
    nodePort: 30050
  selector:
    app: hello-world
```

- Pods on the cluster can access this Service by hit skippy:8080.
- Clients can send traffic on port 30050.

**LoadBalancer**

LoadBalancer Srvices make external acccesswith a high-performance highly-available public IP or DNS name. Only work on clouds that support them.

### Service Discovery

- DNS (Preferred)
- Env Variables (not preferred)

Kub clusters run on an internal DNS service that is the centre of service discovery. Every Pod/container can reslve every Service name to a ClusterIP and connect to the Pods behind it.

## Hands-on

### Dual Stack Networking Primer

- All cluster nodes need a routable IPv4 and IPv6 addresses
- CNI network plugin must support dual stacks

```
# dual-stack-svc.yml
kind: Service
metadata:
  name: dual-stack-svc
spec:
  ipFamilyPolicy: PreferDualStack
  type: ClusterIP
  ports:
  - port: 8080
  protocol: TCP
selector:
  app: hello-world
```

- ipFamilyPolicy
  - SingleStack
  - PreferDualStack
  - RequireDualStack

Prefer and Require dual stack tell Kub to allocate the Service an IPv4 and IPV6. The last one will fail if the cluster does not support dual stack networking.

```
kubectl apply -f dual-stack-svc.yml
kubectl get svc
kubectl describe svc dual-stack-svc
```

- output values:
  - Selector: list of labels
  - ipFamilyPoliy: single or dual stack
  - IP: permanent internal ClusterIP
  - Port: port the Service listens inside the cluster
  - TargetPort: port the app is listening
  - NodePort: cluster-wide port for external access
  - Endpoints: dynamic list of healthy Pod IPs

**To access to the app from the web browser hit an IP address of at least one of the cluster nodes on the NodePort port.**

- The web app deployed listen on 8080
- Kubernetes Service was config to listen on port 30013 on every cluster node
- **NodePorts are between 30.000 and 32.767**.

### declarative way

```
apiVersion: v1
kind: Service
metadata:
  name: svc-test
spec:
  type: NodePort
  ports:
  - port: 8080
  nodePort: 30001
  targetPort: 8080
  protocol: TCP
selector:
  chapter: servic es
```

- Services are objects defined in v1 core API group
- Service Object
- in metadata block you can define name, labels and annotations -> labels are for identify Service, and are no related to selecting Pods
- selector tells Service to send traffic to all healthy Pods on the cluster with the chapter=services label.

```
kubectl apply -f svc.yml

kubectl get svc svc-test
```

ensure any firewall and security rules allow the traffic to flow. on local cluster, use localhost:30001

### Endpoints Services

**Endpoint** object (same name as the Service) hods a list of all the Pods in the Service.

```
kubectl get endpointslices
kubectl describe endpointslice
```

---

## LoadBalancer Services

- Easiest and the best type of Service
- `type=LoadBalancer` internet-facing load-balancer provider to the Service

```yaml
# lb.yml
## listening on 9000
## forwarding to 8080 Pods
### label chapter=services

apiVersion: v1
kind: Service
metadata:
  name: cloud-lb
spec:
  type: LoadBalancer
  ports:
  - port: 9000
    targetPort: 8080
  selector:
    chapter: services
```

```
kubectl apply -f lb.yml
kubectl get svc --watch
```

- EXTERNAL-IP shows public address assigned to the Service by the cloud
- It may be a DNS name instead

```
kubectl delete -f deploy.yml -f svc.yml -f lb.yml
```

# 08: Ingress

- Access to multiple web apps through a single LoadBalancer Service.
- Ingress it's a resource in the Kubernetes API
- LoadBalancer is a Kubernetes Service object of type=LoadBalancer

### Benefits

- **NodePorts**:
  - only works on high port numbers
  - require knowledge of node name or IPs
- **LoadBalancer**:
  - require 1-to-1 mapping between an internal Service and a cloud load-balancer
  - a cluster with 25 internet-facing apps will need 25 cloud load-balancers
- **Ingress** Fixes:
  - uses a single cloud load-balancer
  - ports 80 to 443
  - host-based and path-based routing to send traffic to the backend service

## Architecture

- Stable resource in the Kubernetes API
- v1 object
- spec: defines rules to govern traffic rounting and the controller implements them
- Once created **Ingress Controller** you deploy **Ingress Objects** with rules to hwo route requests.
- Ingress operates at layer 7 of the OSI model (App):
  - It has awareness of HTTP headers
  - can inspect them and forward traffic based on hostnames and paths

| host-based | path-based | K8s Backend Svc |
| --- | --- | --- |
| shield.mcu.com | mcu.com/shield | svc-shield |
| hydra.mcu.com | mcu.com/hydra | svc-hydra |

### INGINX Ingress Controller

- Installed from a YAML file hosted in Kubernetes GitHub repo.
- Namespace, ServiceAccounts, ConfigMap, Roles, etc.
- See `https://github.com/kubernetes/ingress-nginx/releases`

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/

controller-v1.1.0/deploy/static/provider/cloud/deploy.yaml
```

```
kubectl get pods -n ingress-nginx \ -l app.kubernetes.io/name=ingress-nginx
```

### Ingress Classes for Clustrers with multiple Ingress Controllers

1. Assign each Ingress controller to an Ingress class
2. When create Ingress objects assign them to an Ingress class

```
kubectl get ingressclass
```

### Config host-based and path-based rounting

This section deploys two apps and a single Ingress object. Ingress routes both apps via a single load balancer

1. Deploy an app called `shield` and from it with a ClusterIP Service called svc-shield

2. Deploy an app called `hydra` and front it with a ClusterIP Service called svc-hydra

3. Deployu an Ingress object to route the following hostnames and paths

    - Host-based: shield.mcu.com >> svc-shield
    - Host-based: hydra.mcu.com >> svc-hydra
    - Path-based: mcu.com/shield >> svc-shield
    - Path-based: mcu.com/hydra >> svc-hydra

4. A cloud load balancer will be created and the ingress controller will monitor it for traffic

5. Configure DNS name resolution to point shield.mcu.com, hydra.mcu.com and mcu.com to the cloud load-balancer

6. A client will send traffic to shield.mcu.com DNS name resolution will send the traffic to the load-balancer's public endpoint

7. Ingress will read HTTP headers for the hostname resolution

8. Ingress rule will trigger and the traffic will be routed to the svc shield clusterIP backend

9. the clusterip service will ensure the traffic reaches the shield pod

```
kubectl delete ingress mcu-all
kubectl delete namespace ingress-nginx
kubectl delete clusterrole ingress-nginx
kubectl delete clusterroblebinding ingress-nginx

sudo vim /etc/hosts
```

# 9 Service Discovery deep dive

A way to find other apps in the Kubernetes cluster

- Registration
- Discovery

## Service Registration

- The process of an app listing its connection details in a *service registry* so other apps can find it and consume it

1. K8s uses internal DNS as service registryu
2. All K8s Services are auto-registered with DNS

- K8s providess well-known internal DNS service called cluster DNS. Every Pod in the cluster is auto-config to known where to find it
- Implemented in kube-system Namespace as set of Pods managed by a Deployment called `coredns`and frontend by a Service called `kube-dns`.

```
kubectl get pods -n kube-system -l k8s-app=kube-dns
kubectl get deploy -n kube-system -l k8s-app=kube-dns
```

This lists the service fronting them. ClusterIP is the well known IP configured on every Pod/container

```
kubectl get svc -n kube-system -l k8s-app=kube-dns
```

1. You post a **new Service manifest** to the **API server**.
2. Request is authenticated, authorized and subjected to policies
3. Service is allocated a stable virtual IP address, called **ClusterIP**.
4. An Endpoint or EndpointSlice is created to hold a **list of healthy Pods** matching the Service's label selector
5. The *Pod Network* is configured to send traffic to the clusterIP
6. Service's name and IP are registered with the cluster DNS

Anytime Kubernetes Controller detects a Service object, creates a DNS record, mapping the Service name to its ClusterIP. Apps or server *dont need to perform their own service registration*. **the name registered in DNS is the value stored in metadata.name property**.

```
apiVersion: v1
kind: Service
metadata:
    name: ent
spec:
```

```
    selector:
        app: web
    ports:
    - port: 8080
    ...
# At this point the Service front-end config is registered with DNS
# The Service can be discovered
```

## Service backend

- At this point svc fronted is registered and can be discovered
- backend needs building to send traffi cto.

```
kubectl get endpoint end
# shows Endpoints object for a service ent
# every Service has a Endpoint/EndpointSlice
```

## Service Registration Summary

1. Post a new service resource manifest to the API server
2. Its authentiacted and authorized
3. The Service is allocated a ClusterIP
4. Associated Endpoint/EndpointSlice object is created to hold the list of healthy Pod IPs matching the label selector
5. The cluster DNS is running as a Kubernetes-native application and watching the API server for new Service objects
6. It observes it and registers the appropiate DNS A and SRV records
7. Every node is running a kube-proxy that observes the new objects and creates local iptables rules to route the traffic to the Service's ClusterIP , to the pods matching the Service's label selector

## Service Discovery

Assume

| App | Service name | ClusterIP |
| --- | --- | --- |
| Enterprise | ent | 192.168.201.240 |
| Cerritos | cer | 192.168.200.217 |

Apps need to know

- **name** of the Service fronting the apps: App developers are responsible -> write apps with the name of apps to consume
  - Theye need to code the names of Services fronting the remote apps
  - if cerritos app wants to connecto to enterprise, it neds to send requests to the end Service
- How to convert the **name** to an IP address: Kubernetes take car of this point -> Converts names to an IP

Kubernetes converting names to IP addresses using the clusdter DNS

- K8s populates every container /etc/resolv.conf file with the IP address of the cluster NDS
- Service names will be sent to the cluster DNS (kube-dns Service)

```
kubectl get svc -n kube-system -l k8s-app=kube-dns
```

Pods in enterprise sending requests to cerritos:

1. Know the Service namae of the remote app
2. Name resolution (service discovery)
3. Network rounting

ClusterIP are on "special" network called the "service network". This means containers send all ClusterIP traffic to their default gateway.

**The container's default gateway sends the traffic to the node it's running on**. The node sends to its own default gateway.

Every K8s node runs a system service called kube-proxy that implements a controlelr watching the API server for new Services and Endpoint objects. When it sees them ,it creates local IPVS rules telling the node to intercept traffic for the Service's ClusterIP and forward it to individual Pod IPs.

## Sumarising Service Discovery

enterprise app is sending traffic to cerritos

1. Developers needs the name of the cerritos Service (cer)
2. An instance of enterprise tries to send traffic to the cer Service. The client sends the service name to the clusdter DNS asking it to resolve. This is configured in /etc/resolve.conf.
3. DNS cluster replies with the ClusterIP
4. Client sends the direction to his default gateway -> the node where it's running
5. The node sends it to its own default gateway
6. A trap ocurrs and the request is redirected to the IP address of a Pod that matches the Service's label selector

## Service discovery and Namespaces

**EVERY CLUSTER HAS AN ADDRESS SPACE** based on a DNS domain (cluster domain). usually `cluster.local` and objects have unique names within it. por example `ent.default.svc.cluster.local`. Always are FQDN (fully qualified domain name)

```
<object>.<namespace>.svc.clusdter.local
```

creating a couple of namespace called dev and prod will partition the clusdter address pace into two address space

- dev: `<service-name>.dev.svc.clusder.local`
- prod: `<service-name>.prod.svc.cluster.local`

### Service Discovery Example

```
#sd-example.yml
# defines
## 2 Namespaces -> diff name
## 2 Deployments -> same name
## 2 Services -> same name
## standalone jump Pod

apiVersion: v1
kind: Namespace
metadata:
    name: dev
---
apiVersion: v1
kind: Namespace
metadata:
    name: prod
---
apiVersion: aps/v1
kind: Deployment
metadata:
    name: enterprise
    labels:
        app: enterprise
    namespace: dev
spec:
    selector:
        matchLabels:
            app: enterprise
        replicas: 2
    template:
        metadata:
            labels:
                app: enterprise
        spec: containers:
        - image: nigelpoulton/k8sbook:text-dev
        name: enterprise-ctr
        ports:
        - containerPort: 8080
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: enterprise
    labels:
        app: enterprise
    namespace: prod
spec:
```

```
    selector:
        matchLabels:
            app: enterprise
    replicas: 2
    template:
        metadata:
            labels:
                app: enterprise
        spec:
            containers:
            - image: nigelpoulton/k8sbook:text-prod
            name: enterprise-ctr
            ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
    name: ent
    namespace: dev
spec:
    selector:
        app: enterprise
    ports:
    - port: 8080
    type: ClusterIP
---
apiVersion: v1
kind: Service
metadata:
    name: ent
    namespace: prod
spec:
    selecotr:
        app: enterprise
    ports:
        - port: 8080
    type: ClusterIP
---
apiVersion: v1
kind: Pod
metadata:
    name: jump
    namespace: dev
spec:
    terminationGracePeriodSeconds: 5
    containers:
    - name: jump
      image: ubuntu
      tty: true
      stdin: true
```

```
kubectl apply -f sd-example.yml
```

```

```

1. log-in to the jump Pod in the dev
2. check its /etc/resolv.conf
3. connect to ent in the local dev Namespace
4. Connect ent in the remote prod namespace

```
kubectl exec -t jump --namespace dev --bash
root@jump:/#
cat /etc/resolv.conf
```

The search domains lists the dev namespace and the nameserver is set to the IP of the clusdter DNS.

```
apt-get update && apt-get instal curl -y
curl ent:8080
Hello from the Dev Namespace!

curl ent.prod.svc.cluster.local:8080
Hello from the PROD Namespace!

# Short names are automatically resolved to the local Namespace
# FQDNs are required to connect across Namespaces
```

## Troubleshooting service discovery

- Pods: Managed by the `coredns` Deployment
- Service: A ClusdterIP Service called `kube-dns` -> listen on 53 TCP/UDP
- Endpoint: `kube-dns`

**Objects relating to the clusdter DNS are in `kube-system` namespace and taggeth with `k8s-app=kube-dns` label.**

Check de Deployment and Pods

```
kubectl get deploy -n kube-system -l k8s-app=kube-dns
kubectl get pods -n kube-system -l k8s-app=kube-dns
```

- Check logs form each of `coredns` Pods.
- Substitute the names of the Posd in your environment.

```
kubectl logs coredns-XXXXXXXX-XXXX -n kube-system
```

Check the Endpoints object: should show service up, IP address in ClusdterIP field and listening on port 53 TCP/UDP

```
kubectl get svc kube-dns -n kube-system
kubectl get endpointslice -n kube-system -l k8s-app=kube-dns
```

- **Troubleshoot Pods** with networking tools as ping traceroute, curl, dig, nslookup, etc.
- **Troubleshoot DNS Resolution** with nslookup to resolve kub Service. `dns lookup kubernetes`
  - IP address of the clusdter DNS
  - FQDN of the kubernetes Service
  - **if output don't show this values:**
    - delete DNS Pods `kubectl delete pod -n kube-system -l k8s-app=kube-dns`
    - run `get pod -n kube-system -l k8s-app=kube-dns` to check the've restarted
    - test again

# 10: Storage

- Kubernetes support lots of types of storage
  - block, file, object, external, cloud datasystem, etc.
- ALL TYPES of storage are called **VOLUME** in kubernetes.
- Storage providers uses a plugin to allow the storage resources to be surfaced as volumes in Kubernetes
- Moden plugins are based on the *Container Storage Interface (CSI)* open standard to providing clean storage interface for container orchestrators.
- The Kubernetes **persistent volume sybsystem** is a set of API objects to make it easy for apps to consume storage.
  - Persistent Volumes PV: map to external storage assets
  - Persistent Volume Claims PVC: like tickets that authorize Pods to use PVs
  - Storage Classes SC: Make PV and PVC runs automatic
- Example
  1. K8s cluster is running on AWS, AWS adm has created 25GB EBS volume called `ebs-vol`
  2. Kub adm creates a PV called `k8s-vol` that maps to `ebs-vol` using ```ebs.csi.aws.com CSI plugin
  - PV is a simply way of representing external storage asset on the K8s cluster
  3. Pod uses a PVC con claim access to the PV an using it.

### Storage Providers

- Azure File
- AWS Elastic Block Store (EBS) Each provider needs a CSI plugin to expose their storage assets to Kubernetes.

### Container Storage Interface CSI

Open source project that defines a standards-based interface so the storage can be uniform across multiple cont orchestrators.The day-to-day interaction with CSI will be referencing appropiate CSI plugin in the YAML manifest file.

## Dynamic Provisioning wigh Storage Classes

- Storage Classes allow to define differenttiers of storage
- are resources in the storage.k8s.io/v1 API group
- StorageClass type (sc shortname in kubectl)

```
# defines a class of storage called fast-local
# based on AWS SSDs (io1) in Ireland (eu-west-1a)
```

```yaml
kind: StorageClasee
apiVersion: storage.k8s.io/v1
metadata:
    name: fast-local
provisioner: ebs.csi.aws.com
parameters:
    type: io1
    iopsPerGB: "10"
    encrypted: true
allowedTopologies:
-   matchLabelExpressions:
    -   key: topology.ebs.csi.aws.com/zone
        values:
        -   eu-west-1a
```

- SC objects are immutable
- metadata.name should be meanigful
- parameters block is for plugin-specific values
- each class can onlu relato to a single *type* of storage on a single backend

## SC Workflow

1. Have a storage backend (cloud or on premises)
2. Have a K8s cluster connected to the backend storage
3. Install and config the CSI storage plugin
4. Create one or more SC on K8s
5. Deploy Pods with PVCs that reference those SCs

- Pay cloase attention to how
    - PodSpec refs PVC (by name)
    - PVC refs SC (by name)

```yaml
# PVC
kinds: StorageClass                # StorageClass SC ################
apiVersion: storage.k8s.io/v1      ##
metadata:                          ##
    name: fast                     ## Referenced by the PVC
provisioner: pd.csi.storage.gke.io
parameters:                        ##
    type: pd-ssd                   ##################################
---
apiVersion: v1
kind: PersistentVolumeClaim        # Persistent Volume Claim ########
metadata:                          ##
    name: mypvc                    ## Referenced by the PodSpec
spec:                              ##
    accessModes:                   ##
    -   ReadWriteOnce              ##
    resources:                     ##
```

```
        requests:                 ##
            storage: 50Gi         ##
    storageClassName: fast        ## Matches name of the StorageClass
---                               ###################################
apiVersion: v1
kind: Pod                         # Pod #############################
metadata:                         ##
    name: mypod                   ##
spec:                             ##
    volumes:                      ##
    -   name: data                ##
        persistentVolumeClaim:    ##
            claimName: mypvc      # Matches PVC name ################
    containers: ...
    <SNIP>
```

### Access Mode

- ReadWriteOnce RWO: defines a PV that can only be bound as RW by a **SINGLE PVC**
- ReadWriteMany RWM: defines a PV that can only be bound as RW by multiple PVCs. Only supported by file and object storage
- ReadOnlyMany ROM: defines a PV that can only be bound as R/O by multiple PVCs

### Reclaim

ReclaimPoliciy: Tells K8s how to deal with a PV when its PVC is released

- Delete
    - Most Dangerous
    - Default for PVs created dynamically via SC
    - Deletes the PV and associated storage resource when PVC is released
- Retain
    - will keep the PV object on the cluster and the data associated
    - Other PVC are prevented from using it in the future

### Summarizing Storage

- Lets dynamically create backend storage resources auto-mapped to PVs on K8s.
- You define SCs in YAML files and references a plugin storage provider
- Once deployed, SC watches the API for new PVC objects referencing its name
- When a PVCs matching appear, SC creates the asset on the backend to maps to the PV

## Hands-on

### Using an Existing StorageClass

- Using a regional Google Kubernetes Engine cluster wit CSI plugin installed

```
kubectl get sc premium-rwo
kubectl describe sc premium
```

- K8s platforms in general creates three SCs
  - default: for PVCs that don't explicitly specify an SC
  - In prod environments you should always specify an SC that meets app requirements
- PROVISIONER column shows two of the SCs using the CSI plugin
- RECLAIM POLICY is set to Delete: The volumes will be deleted when PVC is deleted
- VOLUMEBINDINGMODE: *immediate* will create the volume on the external storage system as soon as the PVC is created. Immediate is dangerous if have multiple datacenters or cloud regions. Setting WaitForFirstConsumer will delay creation until a Pod using the PVC is created.

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

### Creating a new StorageClass

Defines a class called sc-fast-repl

- Fast SSD storage (type pd-ssd)
- Replicated (replication-type: regional-pd)
- Create on-demand (volumeBindingMode: WaitForFirstConsumer)
- Keep data when the PVC is deleted (reclaimPolicy: Retain)

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: sc-fast-repl
provisioner: pd.csi.storage.gke.io
parametres:
  type: pd-ssd
  replicatoin-type: regional-pd
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Retain
```

```
kubectl apply -f sc-gke-fast-repl.yml
kuibectl get sc
```

1. Created sc-fast-repl SCSC controler started watching the API sv for new PVC
2. ap deployed created the pvc2 PVC that requestses a 20GB volume from the sc-fast-repl SC
3. SC controller notices the PVC dynamically created the backend volume and PV

Even though Pod and PVC deleted, kubeclt get pv whill show PV still exists because the class it was created using Retain reclaim policy. Manually delete the PV with a kbuectl delete pv command.