2-hour Training Mastering Kubernetes

1.1 Environment Preparation

Your Environment

Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a VM on your laptop for users looking to try out Kubernetes or develop with it day-to-day.

Run Minikube remotely

We'll run minikube remotely on a playground:

- Open: https://kubernetes.io/docs/tutorials/hello-minikube/
- Click `` `Launch Terminal` ``
- To verify it works ok:

```
$ kubectl get nodes
NAME       STATUS   ROLES                  AGE     VERSION
minikube   Ready    control-plane,master   2m26s   v1.20.2
```

- Click `` `Preview Port 30000` `` to see Kubernetes Dashboard.

Install Kubectx

Kubectx is a helper tool to switch contexts and namespaces in kubectl.

- To install on the playground:

```
snap install kubectx --classic
```

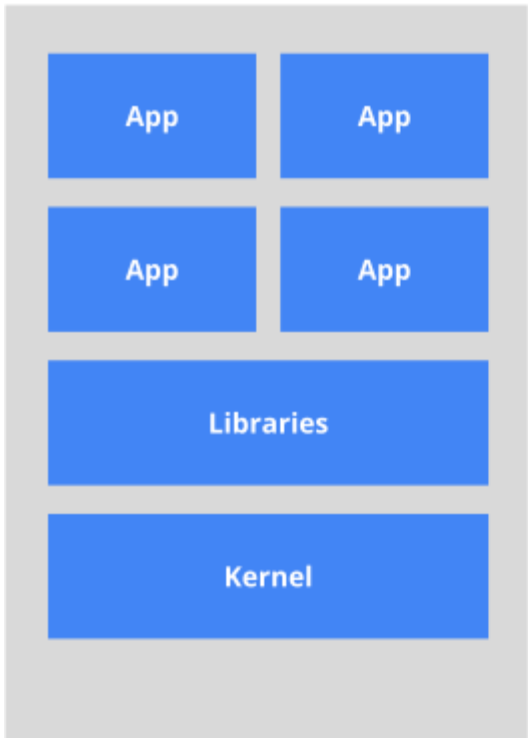
- This installs the following tools: `kubectx` and `kubens`.

```
# kubectx is for switching contexts
$ kubectx
minikube
# kubens is for switching namespaces
$ kubens
default
kube-node-lease
kube-public
kube-system
kubernetes-dashboard
```

Section 1.2: Kubernetes Architecture

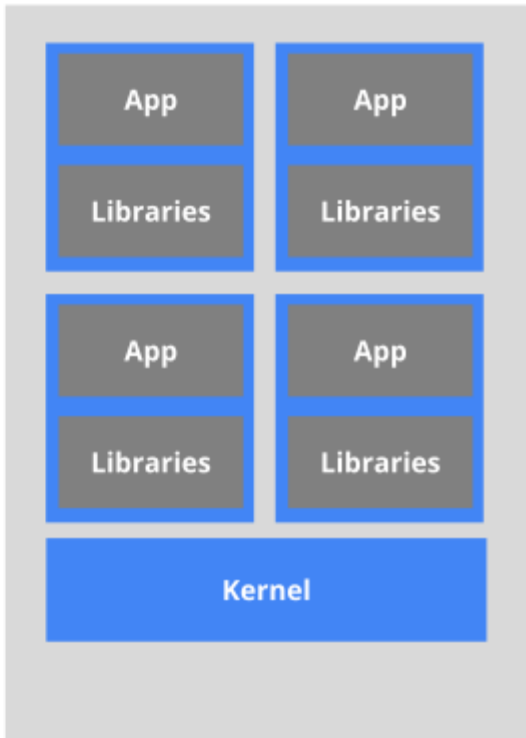
Why Containers

The old way: Applications on host



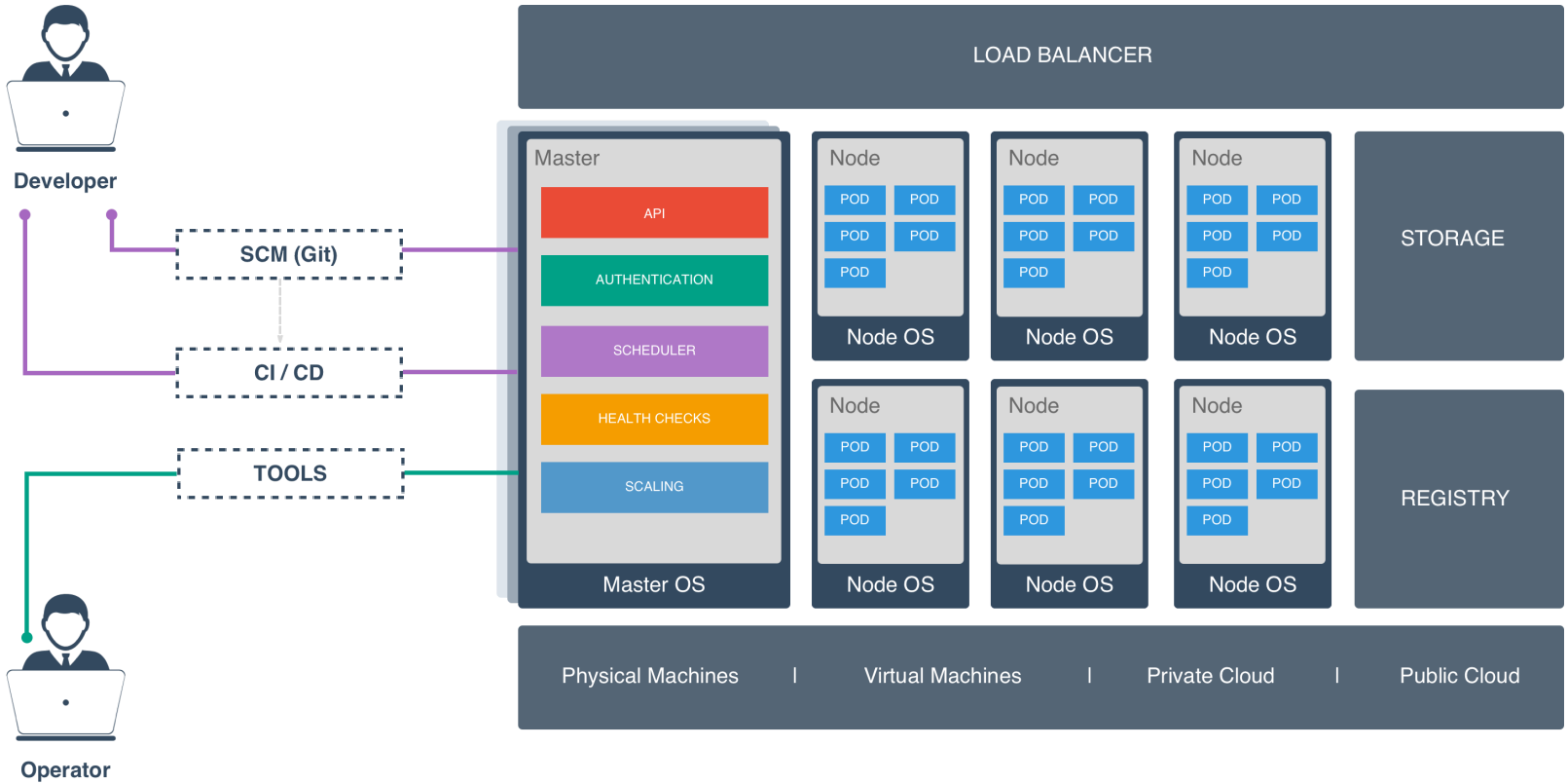
Heavyweight, non-portable
Relies on OS package manager

The new way: Deploy containers

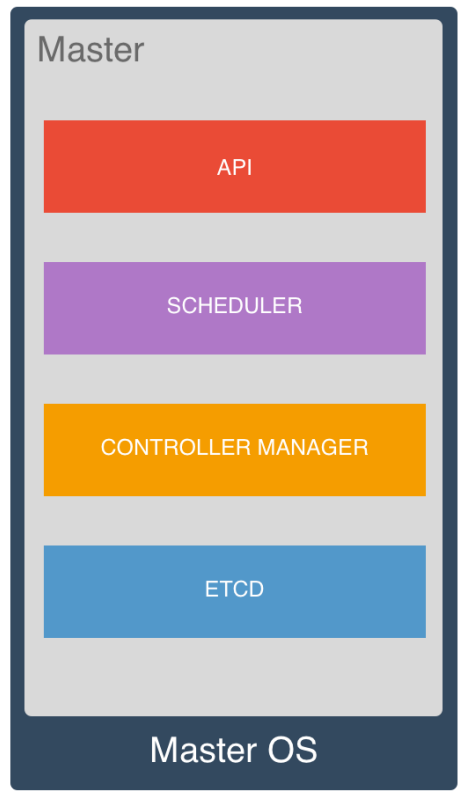


Small and fast, portable
Uses OS-level virtualization

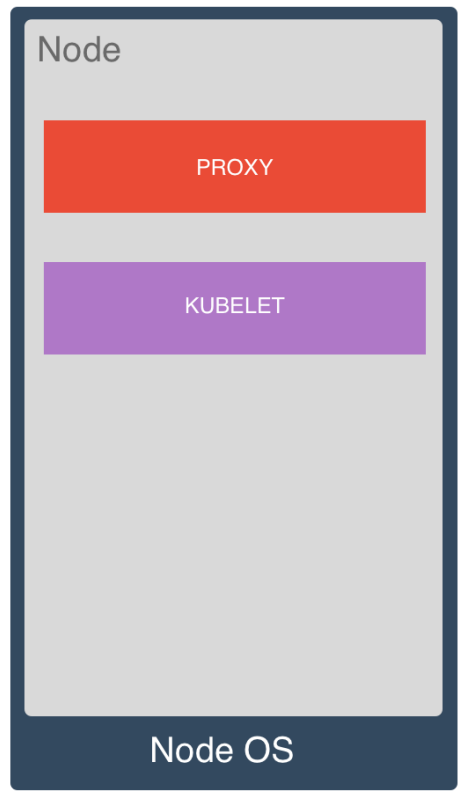
Kubernetes Architecture



Master Architecture



Node Architecture



1.4. Running Containers

Clone repository

- Clone the repository with the examples

```
git clone https://github.com/dkapanidis/training
cd training/kubernetes/101/
```

Pods

A pod is **a group of one or more containers** (such as Docker containers), the shared storage for those containers, and options about how to run the containers.

Pods are always co-located and co-scheduled, and run in a shared context.

A pod models an application-specific “logical host” - it contains one or more application containers which are relatively tightly coupled — in a pre-container world, they would have executed on the same physical or virtual machine.

Pods

Why not just run multiple programs in a single (Docker) container?

- **Transparency**. Making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, (e.g. process management and resource monitoring)
- **Decoupling software dependencies**. The individual containers may be versioned, rebuilt and redeployed independently. Kubernetes may even support live updates of individual containers someday. Ease of use. Users don't need to run their own process managers, worry about signal and exit-code propagation, etc.
- **Efficiency**. Because the infrastructure takes on more responsibility, containers can be lighter weight.

Why not support affinity-based co-scheduling of containers?

- That approach would provide co-location, but would not provide most of the benefits of pods, such as resource sharing, IPC, guaranteed fate sharing, and simplified management.

Hello World Pod

- Let's see a simple hello-world Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: "alpine:3.5"
    command: ["echo", "Hello", "World"]
```

- We use Image `alpine:3.5`
- We run the command `echo Hello World`
- We tell Pod not to restart if finished

Hello World Pod

- Let's create the Pod with `kubectl` CLI. In order to create it:

```
$ kubectl create -f 01.hello-world/
```

- Let's see the Pods:

```
$ kubectl get pods
```

- To see the logs:

```
$ kubectl logs hello-world
```

- Let's delete the Pod using the file reference:

```
$ kubectl delete -f 01.hello-world/
```

- You can also delete a Pod using the name reference:

Long Running Single-Container Pod

- A more useful example is a long running container. We'll now create a pod with one container that will run `nginx`.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.13-alpine
    ports:
    - containerPort: 80
```

Long Running Single-Container Pod

- Let's deploy it (You can reference a directory to the `create` command):

```
$ kubectl create -f 02.single-container-pod/
```

You'll see that the Pod once it finishes to download the image it is in `Running` State and is staying running.

- If you click on the Pod you'll not see any logs, but that's ok because Nginx doesn't give any logs. We'll see later how we can use the Nginx, but for now let's delete the Pod.

```
$ kubectl delete pod nginx
```

Multi-Container Pod

We'll now run a Pod with two containers.

Multi-container Pods should be avoided when possible:

- Containers inside a Pod are co-located (cannot spread to different machines)
- They scale up or down together (cannot scale separately)
- They share the same network IP (may exist port conflicts)

This makes the containers that are under the same port to be **tightly-coupled**, instead of **loosely-coupled**.

But there are some scenarios where this tightly-coupled containers are necessary, such as side-kick, health-check or log-collecting processes.

In our example we'll use a `nginx` together with an `alpine` container that will do requests of the nginx every `2` secs.

Multi-Container Pod

- Multi-container pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container
spec:
  terminationGracePeriodSeconds: 0
  containers:
  - name: nginx
    image: nginx:1.13-alpine
    ports:
    - containerPort: 80
  - name: alpine
    image: alpine:3.5
    command: ["watch", "wget", "-qO-", "localhost"]
```

- Let's deploy it:

```
$ kubectl create -f 03.multi-container-pod/
```

Multi-Container Pod

- What would happen if one of the containers crashes?

```
$ minikube ssh -- 'docker kill $(docker ps -ql)'
```

- The Pod will enter `Error` State, will recreate the failing container (a new container)

```
$ kubectl get pods
```

- What would happen if one of the containers keep crashing?
  - The Pod will enter `CrashLoopBackOff` State, it will wait some time (increased by each failure) and retry.
- What would happen if I loose the node where the Pod is assigned?
  - The Pod is lost, that's why we shouldn't create Pods directly

Multi-Container Pod

- Let’s cleanup the Pod:

```
$ kubectl delete pod multi-container
```

Replica Sets

With Pods we manage to declare the definition of one unit of deployment, e.g. one instance of Nginx.

In a cluster environment we probably want to have more than one instances of our applications for example for the sake of redundancy and throughput.

In order to manage this need we'll use Replica Sets. A Replica Set defines the Pods that needs to run and the number of their replicas.

We'll use the same example of Nginx before but now we'll run multiple instances.

Replica Sets

- Replica Sets:

```yaml
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
      template: replicaset
  template:
    metadata:
      labels:
        app: nginx
        template: replicaset
    spec:
      containers:
      - name: nginx
        image: nginx:1.13-alpine
        ports:
        - containerPort: 80
```

Replica Sets

- Let's deploy it:

```
$ kubectl create -f 04.nginx-replicaset/
```

- Let's see the Pods:

```
$ kubectl get pods
```

- Let's see the ReplicaSet (you can use both singular or plural on the CLI):

```
$ kubectl get replicaset
```

- You can also use aliases for faster typing:

```
$ kubectl get rs
```

- To see the available resources (and their aliases):

Replica Sets

We can change the number of the desired replicas and the controller will make sure to update the cluster:

- We can scale up:

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

- or scale down:

```
$ kubectl scale replicaset nginx --replicas=2
```

If a Pod is deleted, Replica Set Controller will create a new Pod to meet the desired number of replicas.

So if I loose a node, the Pods there will be rescheduled to other Nodes automatically.

- Let's delete the ReplicaSet:

```
$ kubectl delete replicaset nginx
```

Deleting the ReplicaSet it propagates the deletion of the Pods it manages

2.1. Deployment Process

2.1. Deployment Process

In this section we'll learn how to manage deployments on Kubernetes without downtimes, manage upgrades, scale and do rollbacks.

We'll learn about the following resources:

- Deployments

2.1. Deployment Process

With Replica Sets we now have the definition multiple Pods under a specific version.

In order to be able to manage an upgrade or downgrade of a service without having downtime during the upgrade we have to handle multiple instances of both versions during the upgrade.

The objective of Deployment is to handle this process.

2.1. Deployment Process

- Let's use a Deployment definition to deploy 2 Nginx instances in our Cluster.

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.12-alpine
        ports:
        - containerPort: 80
```

2.1. Deployment Process

- Let's deploy it:

```
$ kubectl create -f 05.nginx-deployment/
```

You'll see that the Deployment has generated a Replica Set and that in turn generated 2 Pods.

- Let's see the Pods:

```
$ kubectl get pods
```

- Let's see the ReplicaSets:

```
$ kubectl get replicasets
```

- Let's see the Deployment:

```
$ kubectl get deployment
```

2.1. Deployment Process

The Deployment is configured with 2 replicas, but we can update the definition of the Deployment to increase or decrease the number of Pods and that information is delegated to the Replica Set which in turn is responsible of the number of Pods running.

- As an example we'll increase the number of Pods to 8.

```
$ kubectl scale deployment nginx --replicas=8
```

2.1. Deployment Process

We currently use Nginx `v1.12`, but there is a new version of Nginx `v1.13` we want to deploy to the cluster.

- Let's say we have 8 instances of Nginx running, the objective is to change all of them from v1.12 to v1.13 without loosing service availability.

```
$ kubectl set image deployment nginx *=nginx:1.13-alpine
```

- We can see the status of the rollout:

```
$ kubectl rollout status deployment nginx
```

- If something goes wrong with the new deployment, we can simply rollback the deployment to the previous version:

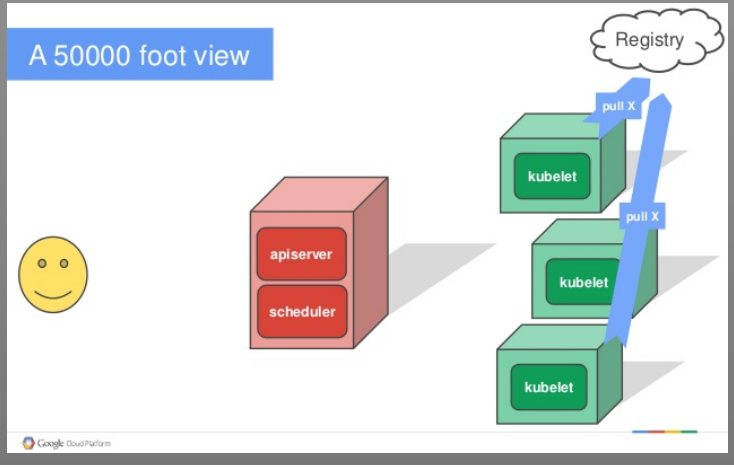
```
$ kubectl rollout undo deployment nginx
```

- Let's cleanup the Deployment:

2.1. Deployment Process

In this section we learned how to manage deployments on Kubernetes without downtimes, manage upgrades, scale and do rollbacks.

Section 2.2: Pulling Images



Pulling Images

For each Pod there is a setting `imagePullPolicy` (by default is `IfNotPresent`)

- **IfNotPresent**: Image will be reused if already exists locally, otherwise will pull from registry.
- **Always**: Will always pull from registry.
- **Never**: Will never pull from registry and only try to run if the image already exists locally.

If you specify image tag `:latest` then it will always pull the image

If you don't specify image tag, `:latest` will be assumed

Nevertheless, you should avoid use `:latest` tag

Private Registry

In order for Kubernetes to access Images on Private Registry it needs credentials

- Native support for Google Container Registry (GCR) when running on GCE
- Native support for AWS EC2 Container Registry, when running on AWS EC2
- Otherwise use a Secret to Store Docker Config:

```
$ kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKEI
secret "myregistrykey" created.
```

And set it to `imagePullSecrets` on Pods or ServiceAccount.

2.3. Networking Basics

Section 2.3: Networking Basics

As we mentioned before Pods are ephemeral and they can be born and die during the day.

In order to be able to group together a set of Pods and make them discoverable without having to keep in mind that they may die and be regenerated (e.g. because a node was faulty) we need something more abstract. This is where Services come in.

A Service is basically a group of Pods that consistitute a Service (e.g. a group of Nginx instances).

When someone wants to access this group of Pods it does it through the service, which will redirect the request to one of those Pods.

Section 2.3: Networking Basics

In this section we'll deploy a Deployment with 3 nginx replicas (Pods) and a Service that connects them.

This is the most common way to deploy an stateless application.

- The Deployment is the same as before, below is the Service definition:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
    - name: web
      port: 80
  selector:
    app: nginx
```

Section 2.3: Networking Basics

There are three types of Services:

- **ClusterIP**: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster (default).
- **NodePort**: Exposes the service on each Node's IP at a static port (the NodePort).
- **LoadBalancer**: Exposes the service externally using a cloud provider's load balancer.
- **ExternalName**: Maps the service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value.

We used `LoadBalancer`, but since we're in minikube and don't have a cloud provider it is equivalent to `NodePort`.

Section 2.3: Networking Basics

- Let's deploy it:

```
$ kubectl create -f 06.nginx-service/
```

- Get the Service:

```
$ kubectl get service nginx
```

- Get the NodePort of the Service

```
$ kubectl get service nginx -o jsonpath={.spec.ports[*].nodePort}
```

- Get the IP of minikube

```
$ minikube ip
```

- Get the output of NODE_IP:NODE_PORT

Section 2.3: Networking Basics

In Docker we discussed about launching multiple containers on one node

- Each container was assigned one IP
- Only services exposed could be seen to outside world

In Kubernetes we have more than one node

- we want two Pods that are in different machines to be able to talk together
- But we don't want to expose everything to the outside world

This is solved by the network model

Section 2.3: Networking Basics

On Kubernetes there are two IP ranges:

- The Pods range
- The Services range

Each Pod is assigned one IP (IP-per-pod model) from the Pods range.

Each Pod can talk to other Pods directly with their IP (in same or different Nodes)

Each Service is assigned an IP on the Services range. That IP is doesn't change during the lifetime of the Pods attached to the Service

There are various implementations of the network model:

- Contiv
- Flannel
- GCE
- L2 networks and linux bridging
- Nuage Networks VCS

Section 2.3: Networking Basics

Let's create two Pods and talk to each other

- First create an `nginx` pod:

```
$ kubectl create -f 02.single-container-pod/
```

- Get the IP of the Pod

```
$ kubectl get pod nginx --template={{.status.podIP}}
```

- Then create a temporal interactive Pod to ping the Pod above

```
$ kubectl run --rm ping -it --image alpine:3.5 sh
wget -qO- 172.17.0.5
```

- Now cleanup

```
$ kubectl delete -f 02.single-container-pod/
```

Section 2.3: Networking Basics

But this way we need to **know** the other Pod's IP. What if the Pod dies and IP changes?

We need a way to discover other Services. There are two ways:

- **Environment Variables**: Each Pod when run it has environment variables for each Service that point to the virtual IP of the Service.
- **DNS**: An optional (though strongly recommended) cluster add-on is a DNS server. For each Service it creates a set of DNS records.

Using DNS we can simply do:

```shell
$ kubectl run --rm ping -it --image alpine:3.5 sh
wget -qO- nginx
```

- DNS will respond for `nginx` with the IP of Service `nginx`
- The virtual IP will basically load balance between the different instances of `nginx`
- If a Pod dies, another one will take its place, the service IP will stay the same and the service will continue to work as expected.

Section 2.3: Networking Basics

Let's cleanup:

```shell
$ kubectl delete -f 06.nginx-service/
```

Section 2.3: Networking Basics

During this section we learned:

- How the networking works in Kubernetes
- How we can expose a Service to the outside world
- How to do Service discovery inside the Cluster

Section 2.4: Configuration Management

Section 2.4: Configuration Management

In this section we'll discuss about the following resources:

- Environment Variables
- ConfigMaps
- Secrets

Section 2.4: Configuration Management

In order to configure our environment there are two ways to do it:

- Environment Variables
- Mounted Files

Why configurable containers, why not store it inside the image?

- Configuration is different on each environment
- The same Image should be able to be reused in different environments (Build once, run anywhere)
- Configuration should not be burned inside the Image, but deployed on runtime

Section 2.4: Configuration Management

- Let's create a Pod with an environment variable

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-pod
spec:
  restartPolicy: Never
  containers:
  - name: app
    image: "alpine:3.5"
    command: ["env"]
    env:
    - name: HELLO
      value: "Hello from the environment"
```

Section 2.4: Configuration Management

- Let's create the Pod:

```
$ kubectl create -f 07.env-pod/
```

- Let's see the logs:

```
$ kubectl logs env-pod
```

- Let's cleanup:

```
$ kubectl delete pod env-pod
```

Section 2.4: Configuration Management

Let's create a Pod with a mounted file.

- The content of the file can be stored at a ConfigMap:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: volumemount-configmap
data:
  key-a: bob
  key-b: alice
  key-c: |
    This is a multiline
    data value
```

Section 2.4: Configuration Management

- And in a Pod we can reference that ConfigMap to bind mount it in a directory:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volumemount
spec:
  restartPolicy: Never
  containers:
  - name: app
    image: "alpine:3.5"
    command: ["cat", "/config/app.conf"]
    volumeMounts:
    - mountPath: /config
      name: config
  volumes:
  - name: config
    configMap:
      name: volumemount-configmap
      items:
      - key: key-c
        path: app.conf
```

Section 2.4: Configuration Management

- Let's create the Pod:

```
$ kubectl create -f 08.volumemount-configmap/
```

Let's see the logs:

```shell
$ kubectl logs volumemount
```

Let's cleanup:

```shell
$ kubectl delete -f 08.volumemount-configmap/
```

3.1. Persistent Volumes

3.1. Persistent Volumes

The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed

To do this we introduce two new API resources

PersistentVolumeClaim (PVC): is a request for storage by a user.

PersistentVolume (PV): is a piece of networked storage in the cluster that has been provisioned by an administrator.

StorageClass provides a way for administrators to describe the “classes” of storage they offer.

Persistent Volume Claim

- This is a PersistentVolumeClaim:

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim-1
  labels:
    temporal: "true"
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

- Let's create it:

```
$ kubectl create -f 09.myclaim-pvc/
```

Persistent Volume

If you check your Persistent Volumes now, you'll see there is already a PV generated and bound to the PVC.

- This is how it looks (`-o yaml` means output in yaml format):

```
$ kubectl get persistentvolume -o yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pvc-ded278e1-08f3-11e7-8842-0800276b3654
  annotations:
    volume.beta.kubernetes.io/storage-class: standard
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
    hostPath:
      path: /tmp/hostpath-provisioner/pvc-ded278e1-08f3-11e7-8842-0800276b3654
  persistentVolumeReclaimPolicy: "Delete"
```

Storage Class

- The PV was generated by an existing Storage Class in minikube.

```
$ kubectl get storageclass
NAME                 PROVISIONER                AGE
standard (default)   k8s.io/minikube-hostpath   4d
```

In this case minikube uses Host paths inside the `/tmp/` directory and provides them as storage to the claims.

There are other implementations:

- AWS uses EBS volumes as storage
- GCE uses PD volumes as storage
- etc

But the PVC is the same as it is abstracted from the storage provider.

- Let's delete the PersistentVolumeClaim:

Ghost

- As an example we'll deploy Ghost

```
...
spec:
  replicas: 1
  template:
    metadata:
      ...
    spec:
      containers:
      - image: ghost:0.10
        name: ghost
        ...
        volumeMounts:
            - name: ghost
              mountPath: /var/lib/ghost
              subPath: "ghost"
      volumes:
        - name: ghost
          persistentVolumeClaim:
            claimName: ghost-claim
```

Ghost

- Let's deploy it:

```
$ kubectl create -f 10.ghost/
```

- Ghost's Items

```
$ kubectl get pvc,pv,pod,svc
```

- Let's delete it:

```
$ kubectl delete -f 10.ghost/
```

Section 3.2: Memory and CPU Quotas

Section 3.2: Memory and CPU Quotas

Kubernetes schedules a Pod to run on a Node only if the Node has enough CPU and RAM available to satisfy the total CPU and RAM requested by all of the containers in the Pod.

- **resources:requests** field: When you create a Pod, you can request CPU and RAM resources for the containers that run in the Pod.

Also, as a container runs on a Node, Kubernetes doesn't allow the CPU and RAM consumed by the container to exceed the limits you specify for the container.

If a container exceeds its RAM limit, it is terminated. If a container exceeds its CPU limit, it becomes a candidate for having its CPU use throttled.

- **resources:limits** field: You can also set limits for CPU and RAM resources.

Section 3.2: Memory and CPU Quotas

- Here is an example of assigning CPU and RAM resources:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpu-ram-limited
spec:
  containers:
  - name: nginx
    image: nginx:1.13-alpine
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "1"
```

The configuration file for the Pod requests `` `250 milicpu` `` and `` `64 mebibytes` `` of RAM.

It also sets upper limits of `` `1 cpu` `` and `` `128 mebibytes` `` of RAM.

(As a reference: megabyte = 1000 bytes, mebibyte = 1024 bytes)

Section 3.2: Memory and CPU Quotas

- Let's deploy it:

```
$ kubectl create -f 11.cpu-ram-limited-pod/
```

- You can check the usage of CPU and Memory per node by describing nodes:

```
$ kubectl describe node
...
  Namespace     Name                       CPU Requests  CPU Limits  Memory Requests Memory Limits
  ─────────     ────                       ────────────  ──────────  ─────────────── ─────────────
  ...
  default       cpu-ram-limited               250m (12%)  1 (50%)   64Mi (3%) 128Mi (6%)
  ...
```

- Let's delete it:

```
$ kubectl delete pod cpu-ram-limited
```