

2-day Training Mastering Kubernetes



Introduction



Google Cloud Platform
Partner

Dimitris Kapanidis

- Founder and Senior Consultant at **Harbur**
- **Docker BCN** Meetup organizer (1.816 members)
- **Kubernetes BCN** Meetup organizer (401 members)
- Member of **Docker Captains** and **Google Developer Experts**



Training Preparation

Online Chat

- Online chat channel during the training: <https://gitter.im/harbur/training>
- copy/paste material will be sent through this channel
- **NOTE:** Do not share credentials here, this is an open channel

Slides

- The slides of the training are shared through the chat in PDF format
- Download the slides before the training
- Use the PDF if you want to review previous slides

Stickers

- There is a stack of stickers in the room. Pick one
- We'll use stickers so that I can visualize when you complete a task
- I'll periodically ask you to "Place your sticker when you finish" and pause
- Stick it on backside of your Laptop's screen when you finish your task
- Once I resume remember to remove the sticker

Training Evaluation

- At the end of the Training I'll post you an Evaluation Survey link on the Chat
- It takes approx 5mins to complete the survey
- Your feedback is important!

Training Structure

- Each day is split in 4 classes
- Each class will last 1h 45m
 - 15m coffee breaks
 - 1h lunch break
- 1st day is focused on:
 - Describing the foundation concepts of Kubernetes architecture
 - Interactive laboratories
- 2nd day is focused on:
 - Describing more advanced features
 - Last class is dedicated exclusively in Q & A and specific issues you want to deep dive into

Schedule – Day 1

- 09:00 – 10:45 Section 1
- Coffee Break
- 11:00 – 13:00 Section 1
- Lunch Break
- 14:00 – 15:45 Section 2
- Coffee Break
- 16:00 – 18:00 Section 2

Table of Contents – Day 1

Section 1

- 1.1. Environment Preparation
- 1.2. Kubernetes Architecture
- 1.3. Running Kubernetes
- 1.4. Running Containers
- Laboratory 1: Run 10 Redis instances

Section 2

- 2.1. Deployment Process
- 2.2. Pulling Images
- 2.3. Networking Basics
- 2.4. Configuration Management
- Laboratory 2: Run Guestbook

Schedule – Day 2

- 09:00 – 10:45 Section 3
- Coffee Break
- 11:00 – 13:00 Section 4
- Lunch Break
- 14:00 – 15:45 Section 5
- Coffee Break
- 16:00 – 18:00 Q & A

Table of Contents – Day 2

Section 3

- 3.1. Persistent Volumes
- 3.2. Memory and CPU Quotas
- 3.3. Health Checks
- 3.4. Kubernetes Addons

Section 4

- 4.1. Autoscaling
- 4.2. Operating Kubernetes with Kops

Table of Contents – Day 2 (cont.)

Section 5

- 4.3: Package Management with Helm
- 5.1: Authorization with RBAC
- 5.4: Istio.sh

1.1 Environment Preparation



1.1 Environment Preparation

In this section we'll see how to prepare the environment with the necessary tools

- **Install Minikube** to Run Kubernetes locally
- **Install Kubectl** to manage Kubernetes through CLI
- **Install Kubernetic** to manage Kubernetes through GUI
- **Install AWS CLI** to access Amazon Web Services through CLI
- **Install Kops** to Run Kubernetes on AWS
- **Install Helm** to manage Kubernetes Packages

Install Minikube



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.

Install Minikube

Minikube is easy to install

on **OSX**

```
$ curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/v0.22.2/minikube-darwin-amd64 \
&& chmod +x minikube \
&& sudo mv minikube /usr/local/bin/
```

on **Linux**

```
$ curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/v0.22.2/minikube-linux-amd64 \
&& chmod +x minikube \
&& sudo mv minikube /usr/local/bin/
```

On **Windows** there is an installer [here](#)

Requirements for Minikube

Minikube uses Virtualization to run a Virtual Machine that hosts all the components for running Kubernetes. Once the VM is ready you also need `kubectl` (The Kubernetes Client) to interact with it.

Requirements:

- OS X
 - xhyve driver, VirtualBox or VMware Fusion installation
- Linux
 - VirtualBox or KVM installation,
- Windows
 - Hyper-V
- VT-x/AMD-v virtualization must be enabled in BIOS
- `kubectl` must be on your path.

Install Kubectl

Kubectl is even easier to [install](#)

on **OSX** with Brew (*Recommended*)

```
brew install kubernetes-cli
```

on **OSX** without Brew

```
$ curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/$(uname -m)/latest)kubernetes-linux.tar.gz"
&& tar -xzf kubernetes-linux.tar.gz
&& mv kubernetes-linux/bin/kubectl /usr/local/bin/
```

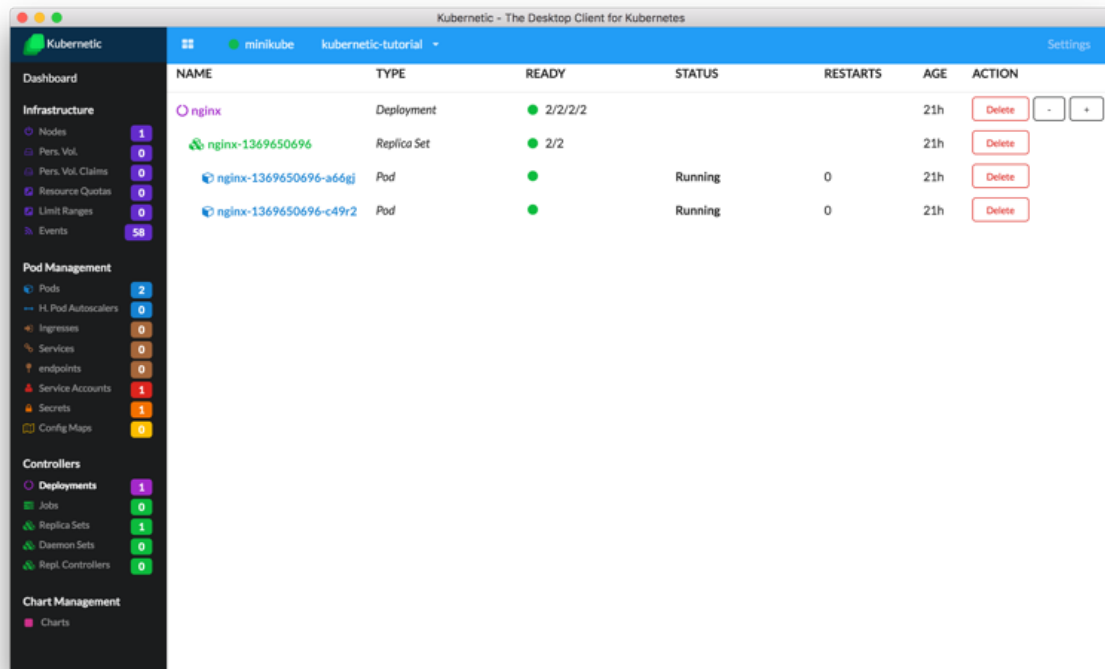
on **Linux**

```
$ curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/$(uname -m)/latest)kubernetes-linux.tar.gz"
&& tar -xzf kubernetes-linux.tar.gz
&& mv kubernetes-linux/bin/kubectl /usr/local/bin/
```

on **Windows**

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/$(uname -m)/latest)kubernetes-windows.tar.gz
```

Install Kubernetes



Kubernetic is a Desktop Client for Kubernetes Cluster management

Installing Kubernetes

There are Installers for OSX, Linux or Windows at <http://kubernetes.com/>

It reuses `kubectl` cluster configuration, so once the CLI is configured, Kubernetes can be used to visualize and manage the cluster

Kubernetes is developed by *Harbur Cloud Solutions S.L.*

During *Beta* phase it can be downloaded for free, for more info check the website

During the Training we'll use it for visualizing better the concepts of Kubernetes

Install AWS CLI

For Operating a Cluster in AWS we'll need [AWS CLI](#) and some credentials
on **OSX** with Brew (*Recommended*)

```
brew install awscli
```

on other OSes:

```
pip install awscli
```

Configure AWS CLI

To check if AWS CLI is installed

```
$ aws --version  
aws-cli/1.14.30 Python/3.6.4 Darwin/16.7.0 botocore/1.8.34
```



To configure credentials you'll need an [AWS Access Key](#)

```
$ aws configure  
AWS Access Key ID: foo  
AWS Secret Access Key: bar  
Default region name [us-west-2]: us-west-2  
Default output format [None]: json
```

Check credentials work by listing the users

```
$ aws iam list-users
```



Install Kops

[Kops](#) stands for Kubernetes Operations (kops) – Production Grade K8s Installation, Upgrades, and Management

Platforms:

- **AWS**
- **GCE**

on **OSX** with Brew (*Recommended*)

```
$ brew install kops
```

on other OSes visit [releases](#)

Configure Kops

To check if Kops is installed

```
$ kops version  
Version 1.8.0
```



To [configure](#) Kops you need a AWS S3 to store the State (save it at `/.bashrc`)

```
export KOPS_STATE_STORE=s3://prefix-example-com-state-store
```

Then you can check to see your clusters

```
$ kops get clusters  
NAME          CLOUD  ZONES  
mycluster.mydomain.io  aws    eu-west-1a,eu-west-1b,eu-west-1c
```



Install Helm



[Helm](#) is The Kubernetes Package Manager

Install Helm

on **OSX** with Brew (*Recommended*)

```
$ brew install kubernetes-helm
```

On **Linux** (or OSX without Brew)

```
$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get > get_helm.sh
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

On **Windows**

```
$ curl https://storage.googleapis.com/kubernetes-helm/helm-v2.2.2-windows-amd64.zip
```

Install Helm

To review helm is installed

```
$ helm version -c  
Client: &version.Version{SemVer:"v2.8.0", GitCommit:"14af25f1de6832228539259b82194!"}
```

NOTE: `-c` stands for client version only, since we don't have kubernetes installed yet.

1.1 Environment Preparation

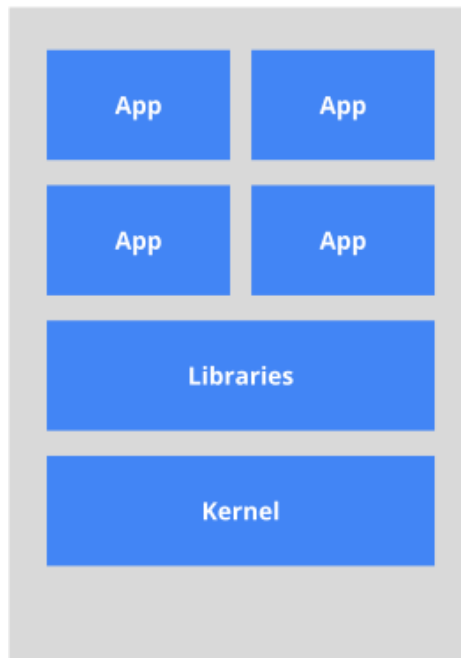
After this section we now have achieved the following:

- **Install Minikube** to Run Kubernetes locally
- **Install Kubectl** to manage Kubernetes through CLI
- **Install Kubernetic** to manage Kubernetes through GUI
- **Install AWS CLI** to access Amazon Web Services through CLI
- **Install Kops** to do Kubernetes Operations (e.g. Upgrades)
- **Install Helm** to manage Kubernetes Packages

Section 1.2: Kubernetes Architecture

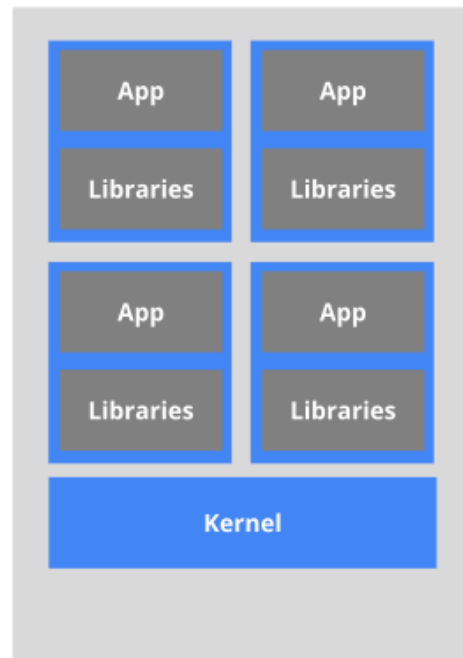
Why Containers

The old way: Applications on host



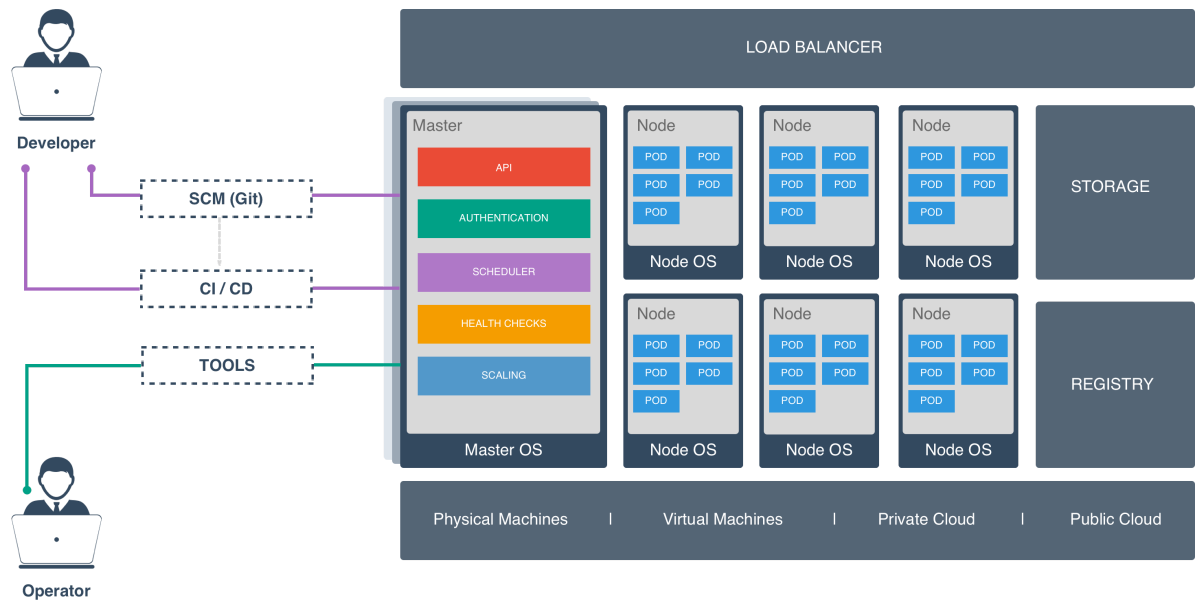
*Heavyweight, non-portable
Relies on OS package manager*

The new way: Deploy containers

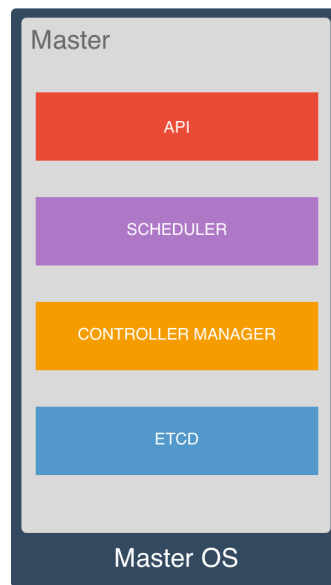


*Small and fast, portable
Uses OS-level virtualization*

Kubernetes Architecture



Master Architecture



ETCD

etcd is a distributed, reliable key-value store which Kubernetes uses for persistent storage of all of its REST API objects.

- **Access Control:** give only kube-apiserver read/write access to etcd. Access to etcd is equivalent to root in your cluster.
- **Data Reliability:** Either etcd needs to be run as a cluster (multiple machines each running etcd) or etcd's data directory should be located on durable storage (e.g., GCE's persistent disk). In either case, if high availability is required—as it might be in a production cluster—the data directory ought to be backed up periodically, to reduce downtime in case of corruption.

etcd is the only place that Kubernetes keeps state.

Kubernetes API Server

The Kubernetes API server validates and configures data for the api objects which include pods, services, replicationcontrollers, and others.

The API Server services REST operations and provides the frontend to the cluster's shared state through which all other components interact.

In a typical Kubernetes cluster, the API served on port 443.

- All the Cluster State
- No Logic is added In the API Server.
- It only exposes the API and stores the state in etcd.
- The Kubernetes CLI (`kubectl`) only talks with the API, not directly to other components (This makes the system declarative, as you declare your desired state and other components drive the cluster to the desired state)

Kubernetes Controller Manager

The Kubernetes controller manager is a daemon that embeds the core control loops shipped with Kubernetes.

In applications of robotics and automation, a control loop is a non-terminating loop that regulates the state of the system.

In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state.

Examples of controllers that ship with Kubernetes today are:

- Deployment Controller
- Replication Controller
- Endpoints Controller
- Namespace Controller
- ServiceAccounts Controller

Kubernetes Scheduler

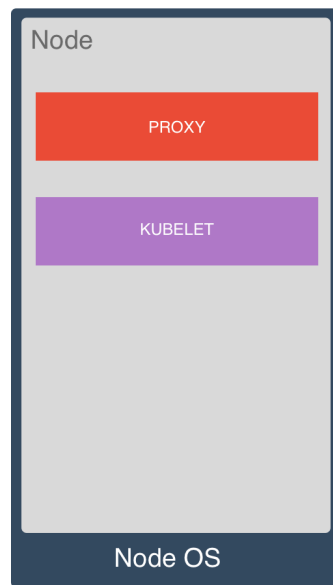
The Kubernetes scheduler is a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity.

The scheduler needs to take into account individual and collective resource requirements, quality of service requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, deadlines, and so on.

Workload-specific requirements will be exposed through the API as necessary.

Basically, it is responsible for Scheduling which Pods run where.

Node Architecture



Kubernetes Kubelet

The kubelet is the primary “node agent” that runs on each node.

The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy.

The kubelet doesn't manage containers which were not created by Kubernetes.

Basically, it is responsible for running the Pods assigned to the specific Node.

Kubernetes Proxy

The Kubernetes network proxy runs on each node.

This reflects services as defined in the Kubernetes API on each node and can do simple TCP,UDP stream forwarding or round robin TCP,UDP forwarding across a set of backends.

Service cluster ips and ports are currently found through Docker-links-compatible environment variables specifying ports opened by the service proxy.

There is an optional addon that provides cluster DNS for these cluster IPs.

The user must create a service with the apiserver API to configure the proxy.

Basically, it is responsible from the Networking of the Node.

1.2 Kubernetes Architecture

During this section we learned about the different components of Kubernetes architecture:

- Etcd
- Kubernetes API Server
- Kubernetes Controller Manager
- Kubernetes Scheduler
- Kubernetes Kubelet
- Kubernetes Proxy

Section 1.3: Running Kubernetes

Section 1.3: Running Kubernetes

In this section we'll talk about options how to run a Kubernetes Cluster

- Minikube (For Single-node Clusters in Laptops)
- Kops (For Multi-node Clusters in AWS)

Choosing the proper Cluster

If you want to experiment with a Kubernetes Cluster yourself use:

- Minikube

If you want a collaborative environment (For Dev, Test, Production) and have access to AWS use:

- Kops

Today we'll use Minikube for most of the course

Starting Minikube

Check if minikube is installed properly:

```
$ minikube version
minikube version: v0.25.0
```



To check the state of your Kubernetes

```
$ minikube status
minikubeVM: Running
localkube: Running
```



- minikubeVM is the Virtual Machine where kubernetes runs
- localkube is the Kubernetes (all components in one binary) inside the minikube

Starting Minikube

To start the Kubernetes

```
$ minikube start
Starting local Kubernetes cluster...
Starting VM...
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

- If the minikubeVM image is not downloaded before, it will download it for the first time.
- It will start the VM with Kubernetes.
- Configure `kubectl` locally to use the cluster.

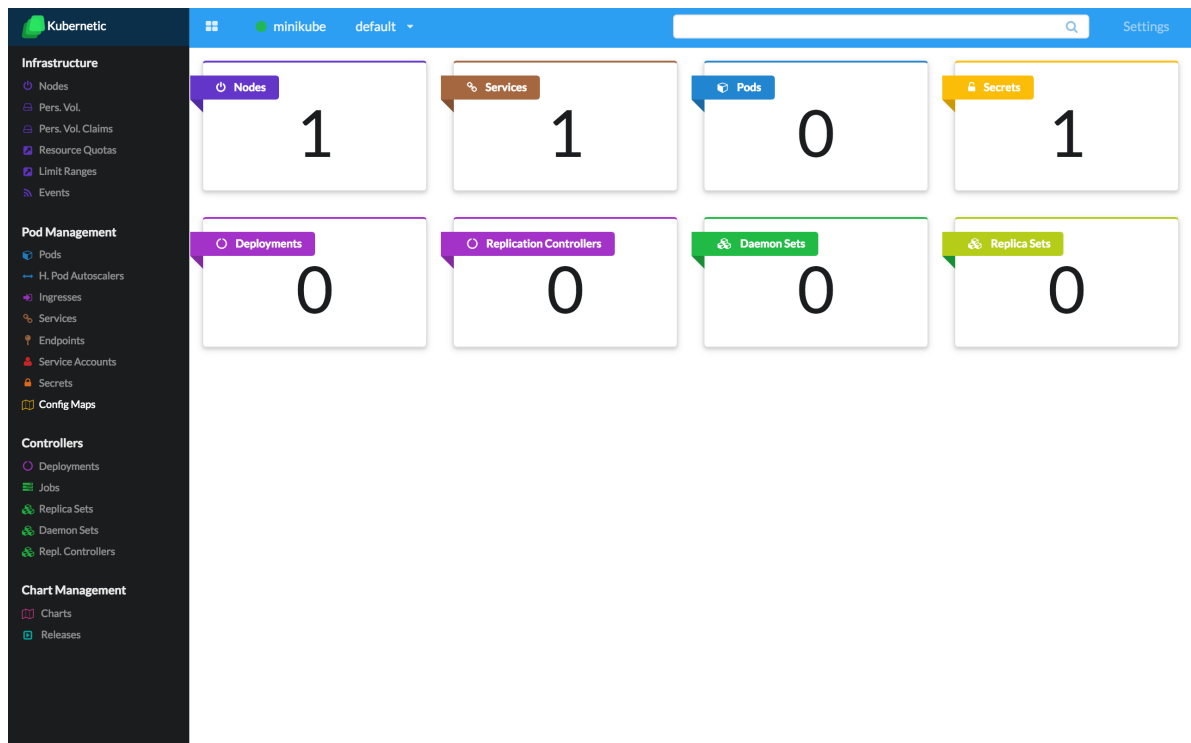
Now you can check connectivity with your cluster using `kubectl`:

```
$ kubectl get nodes
NAME          STATUS    AGE
minikube      Ready     2d
```



Starting Kubernetes

Now launch Kubernetes, you should be able to connect to the Cluster



Section 1.3: Running Kubernetes

In this section we managed to create a Kubernetes cluster and connect to it.

Section 1.4: Running Containers



Section 1.4: Running Containers

In this section we'll learn how to run our first containers in Kubernetes and we'll learn about:

- Pods
- ReplicaSets

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:

```
$ cat kubernetes/01.hello-world/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 kubernetes/01.hello-world/hello-world-pod.yaml
```



Let's see the Pods (`-a` is to also see the Completed ones):

```
$ kubectl get pods -a
```



To see the logs:

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



Let's delete the Pod using the file reference:

```
$ kubectl delete -f kubernetes/01.hello-world/hello-world-pod.yaml
```



You can also delete a Pod using the name reference:

```
$ kubectl delete pod hello-world
```



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`.

```
$ cat kubernetes/02.single-container-pod/single-container-pod.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 kubernetes/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

```
$ cat kubernetes/03.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 kubernetes/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

```
$ cat kubernetes/04.nginx-replicaset/nginx-replicaset.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 kubernetes/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):

```
$ kubectl get
```



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

Section 1.4: Running Containers

In this section learned how to run our first containers in Kubernetes and the following resources in the Kubernetes API:

- Pods
- ReplicaSets

Lab 1: Run 10 Redis instances

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.11-alpine
          ports:
            - containerPort: 80
```

- Change the name of the ReplicaSet and number of replicas
- Configure the label selector and the labels of the Pods
- Name the Container and the image of the Container (`redis:3.2.8-alpine`)
- Configure the port of the Container (`6379`)

Section 2.1: Deployment Process



Section 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

Section 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.

Section 2.1: Deployment Process

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

```
$ cat kubernetes/05.nginx-deployment/nginx-deployment.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
```



Section 2.1: Deployment Process

Let's deploy it:

```
$ kubectl create -f kubernetes/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 replicaset
```



Let's see the Deployment:

```
$ kubectl get deployment
```



Section 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
```



Section 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 losing 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:

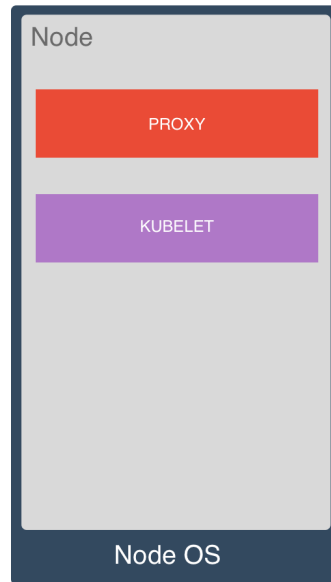
```
$ kubectl delete deployment nginx
```



Section 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

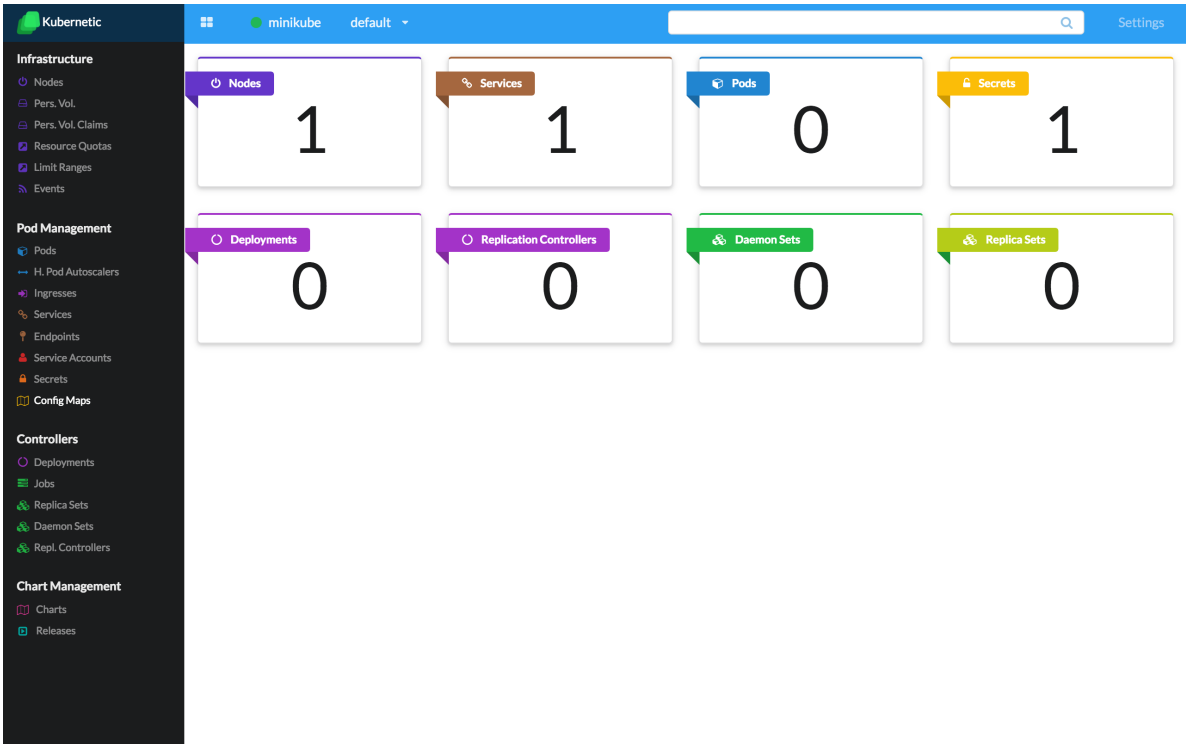
- Native support for Google Container Registry (GCR) when running on GCE
- Native support for AWS EC2 Container Registry, when running on AWS EC2

Otherwise use a Secret to Store Docker Config:

```
$ kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
secret "myregistrykey" created.
```

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

Section 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 constitute 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:

```
$ cat kubernetes/06.nginx-service/nginx-service.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 kubernetes/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

```
$ curl -sL $(minikube ip):$(kubectl get service nginx -o jsonpath={.spec.ports[*].nodePort})
```



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
- Weave Net
- ...

Section 2.3: Networking Basics

Let's create two Pods and talk to each other

First create an `nginx` pod:

```
$ kubectl create -f kubernetes/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 kubernetes/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:

```
$ 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:

```
$ kubectl delete -f kubernetes/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

```
$ cat kubernetes/07.env-pod/env-pod.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 kubernetes/07.env-pod/env-pod.yaml
```



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:

```
$ cat kubernetes/08.volumemount-configmap/volumemount-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:

```
$ cat kubernetes/08.volumemount-configmap/volumemount-pod.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 kubernetes/08.volumemount-configmap/
```



Let's see the logs:

```
$ kubectl logs volumemount
```



Let's cleanup:

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



Laboratory 2: Run Guestbook

In this Lab we'll run the Guestbook application in Kubernetes. Download code:

```
git clone git@github.com:spidny/training.git
cd guestbook-node
```

Now compile containers (connect Docker client to docker inside minikube)

```
$ eval $(minikube docker-env)
$ docker build -t guestbook:v1 .
```

And deploy the Guestbook + Redis:

```
$ cd ../guestbook-node-k8s/
$ kubectl create -f redis-deployment.yaml
$ kubectl create -f redis-svc.yaml
$ kubectl create -f guestbook-deployment.yaml
$ kubectl create -f guestbook-svc.yaml
```

Find the exposed port of Guestbook and connect to write a message

Scale the Guestbook to 10 instances and check it works ok

Laboratory 2: Run Guestbook

To cleanup do inside `guestbook-node-k8s` directory

```
$ kubectl delete -f .  
deployment "guestbook" deleted  
service "guestbook" deleted  
deployment "redis" deleted  
service "redis" deleted
```

Section 3.1: Persistent Volumes

Section 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:

```
$ cat kubernetes/09.myclaim-pvc/myclaim-pvc.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 kubernetes/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:

```
$ kubectl delete persistentvolumeclaim myclaim-1
```



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 kubernetes/10.ghost/
```



Ghost's Items

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



Let's delete it:

```
$ kubectl delete -f kubernetes/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:

```
$ cat kubernetes/11.cpu-ram-limited-pod/cpu-ram-limited-pod.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 millicpu 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 kubernetes/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
-----	----	-----	-----	-----
...				
default	cpu-ram-limited	250m (12%)	1 (50%)	64Mi (3%)
...				



Let's delete it:

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



Section 3.3: Health Checks

Section 3.3: Health Checks

- **Liveness Probe:** The kubelet uses liveness probes to know when to restart a Container.

For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

- **Readiness Probe:** The kubelet uses readiness probes to know when a Container is ready to start accepting traffic.

A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

Liveness Probe

```
$ cat kubernetes/12.liveness-probe-pod/liveness-probe-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-probe
spec:
  containers:
  - name: liveness-probe
    args:
    - /server
    image: gcr.io/google_containers/liveness
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      failureThreshold: 2
      initialDelaySeconds: 3
      periodSeconds: 2
```



liveness image returns OK (200) at /healthz for 10 seconds and then returns error (500).

Liveness Probe is checking if /healthz at port 8080 is ok, starting after 3 seconds delay, repeating every 2 seconds. It needs to fail at least 2 consecutive times.

Liveness Probe

Let's deploy it:

```
$ kubectl create -f kubernetes/12.liveness-probe-pod/
```



Let's see the Pod:

```
$ kubectl get pod
```



Let's delete it:

```
$ kubectl delete pod liveness-probe
```



Readiness Probe

```
$ cat kubernetes/13.readiness-probe-pod/readiness-probe-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: readiness-probe
    name: readiness-probe
spec:
  containers:
  - name: readiness-probe
    args:
    - /server
    image: gcr.io/google_containers/liveness
    readinessProbe:
      httpGet:
        path: /healthz
        port: 8080
      failureThreshold: 2
      initialDelaySeconds: 3
      periodSeconds: 2
```



Same as liveness, but now instead of restarting the Pod, it marks it as `unhealthy` and disconnects it from Services.

Readiness Probe

Let's deploy it:

```
$ kubectl create -f kubernetes/13.readiness-probe-pod/
```



Let's see the Pod:

```
$ kubectl get pod
```



Let's see the Service:

```
$ kubectl describe svc readiness-probe
```



Let's delete it:

```
$ kubectl delete -f kubernetes/13.readiness-probe-pod/
```



Section 3.4: Kubernetes Addons

Section 3.4: Kubernetes Addons

In this section we'll talk about the Kubernetes Addons:

- Dashboard
- StorageClass
- DNS
- Heapster
- Ingress

Minikube Addons

Minikube comes with some bundled addons that can be enabled/disabled:

```
$ minikube addons list
```



Dashboard

Dashboard is a general-purpose web UI for Kubernetes clusters

To open it:

```
$ minikube dashboard
```



StorageClass

Minikube comes with a default storageclass that creates the PersistentVolumes.

We've seen it in action in the Persistent Volumes section, when we created a PersistentVolumeClaim and it generated the Persistent Volume.

DNS

kube-dns is the DNS service inside the Cluster. This responds when we request Services by names.

We've seen it in action in the Networking Basics section when we did `wget -qO- nginx`.

Heapster

Heapster is responsible for monitoring of resource consumption in the Cluster.

```
$ minikube addons enable heapster
```



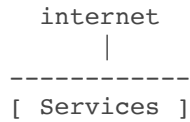
Heapster addon also includes Grafana frontend which can be accessed with:

```
$ minikube addons open heapster
```

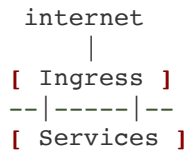


Ingress Controller

Typically, services and pods have IPs only routable by the cluster network. All traffic that ends up at an edge router is either dropped or forwarded elsewhere. Conceptually, this might look like:



An Ingress is a collection of rules that allow inbound connections to reach the cluster services.



It can be configured to give services externally-reachable urls, load balance traffic, terminate SSL, offer name based virtual hosting etc.

Ingress Controller

First enable Ingress:

```
$ minikube addons enable ingress
```



Ingress Controller

Here is an example of an Ingress:

```
$ cat kubernetes/14.coffee-ingress/coffee-ingress.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: coffee-ingress
spec:
  rules:
  - host: coffee.example.com
    http:
      paths:
      - backend:
          serviceName: tea
          servicePort: 80
        path: /tea
      - backend:
          serviceName: coffee
          servicePort: 80
        path: /coffee
```



Ingress Controller

Let's deploy it:

```
$ kubectl create -f kubernetes/14.coffee-ingress/
```



Ingress Controller

- It will create two deployments, `coffee` with 2 replicas, `tea` with 3 replicas.
- For each one there is one Service `coffee` and `tea` respectively (Services are type `ClusterIP`, only visible inside the cluster)
- It will create an Ingress rule that does the following:
 - <http://coffee.example.com/coffee> > `coffee` Service
 - <http://coffee.example.com/tea> > `tea` Service

Ingress Controller listens on port `80` and redirects all traffic depending the rules.

Configure `/etc/hosts` with the following line (replace `192.168.99.100` real with minikube ip)

```
192.168.99.100 coffee.example.com
```

Ingress Controller

Let's delete it:

```
$ kubectl delete -f kubernetes/14.coffee-ingress/
```



Section 4.1: Autoscaling

Section 4.1: Autoscaling

There are two types of autoscaling:

- Pod Autoscaling
- Cluster Autoscaling

And there are two ways to scale something:

- Vertically (Grow bigger)
- Horizontally (Grow in number)

Horizontal Pod Autoscaler



Horizontal Pod Autoscaler

Let's deploy `nginx` Chart again.

Now let's create HPA:

```
kubectl autoscale deployment nginx --min 1 --max 10 --cpu-percent=50
```

And exec inside the `nginx` pod and do some CPU consumption

```
dd if=/dev/zero of=/dev/null
```

This will trigger the HPA to add more instances.

AWS Cluster Autoscaler

The cluster autoscaler on AWS scales worker nodes within an autoscaling group.

```
$ helm install stable/aws-cluster-autoscaler
```

Configuration can be seen [here](#)

It needs some IAM Permissions to update the AutoScalingGroup in AWS:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "autoscaling:DescribeAutoScalingGroups",
        "autoscaling:DescribeAutoScalingInstances",
        "autoscaling:SetDesiredCapacity",
        "autoscaling:TerminateInstanceInAutoScalingGroup"
      ],
      "Resource": "*"
    }
  ]
}
```

Section 4.2: Operating Kubernetes

Section 4.2: Operating Kubernetes

In this section we'll learn how to operate a cluster using Kops.

Get Clusters

First of all make sure kops is configured properly (with S3 bucket for state)

To get list of clusters:

```
$ kops get clusters
NAME          CLOUD    ZONES
k8s.mydomain.com  aws  eu-west-1a,eu-west-1b,eu-west-1c
```

Create a Cluster

We can create a cluster like this:

```
$ kops create cluster \  
  --zones us-west-2a \  
  ${NAME}
```

This will create entries of the cluster in S3 (But will not do anything in AWS yet)

More options can be configured when creating a cluster:

```
$ kops create cluster \  
  --zones eu-west-1a,eu-west-1b,eu-west-1c \  
  --master-zones eu-west-1a \  
  --vpc=vpc-l24f3a0a \  
  --master-size=t2.small \  
  --node-size=t2.medium \  
  --node-count=1 \  
  ${NAME}
```

Review the Cluster

Once the create cluster finished we get the following text:

```
Must specify --yes to apply changes
```

```
Cluster configuration has been created.
```

```
Suggestions:
```

- * list clusters with: `kops get cluster`
- * edit this cluster with: `kops edit cluster k8s.mydomain.com`
- * edit your node instance group: `kops edit ig --name=k8s.mydomain.com nodes`
- * edit your master instance group: `kops edit ig --name=k8s.mydomain.com master-u:`

```
Finally configure your cluster with: kops update cluster k8s.mydomain.com --yes
```

Review the Cluster

Once the objects are created you can edit them before submitting them to AWS:

```
$ kops edit cluster k8s.mydomain.com
```

Review the Cluster

```
apiVersion: kops/v1alpha2
kind: Cluster
metadata:
  creationTimestamp: "2017-02-19T14:19:51Z"
  name: k8s.mydomain.com
spec:
  api:
    dns: {}
  channel: stable
  cloudProvider: aws
  configBase: s3://kopsbucket/k8s.mydomain.com
  etcdClusters:
    - etcdMembers:
        - instanceGroup: master-eu-west-1a
          name: a
        name: main
    - etcdMembers:
        - instanceGroup: master-eu-west-1a
          name: a
        name: events
  kubernetesApiAccess:
    - 0.0.0.0/0
  kubernetesVersion: 1.5.2
  masterInternalName: api.internal.k8s.mydomain.com
  masterPublicName: api.k8s.mydomain.com
  networkCIDR: 172.14.0.0/16
```

Get Instance Groups

```
$ kops get instancegroups
```

```
Using cluster from kubectl context: k8s.mydomain.com
```

NAME	ROLE	MACHINETYPE	MIN	MAX	SUBNETS
master-eu-west-1a	Master	t2.small	1	1	eu-west-1a
nodes	Node	t2.medium	1	4	eu-west-1a

Edit Instance Group Master

To edit the Instance Group of Master:

```
$ kops edit ig master-eu-west-1a
```

The output is

```
apiVersion: kops/v1alpha2
kind: InstanceGroup
metadata:
  creationTimestamp: "2017-02-19T14:19:51Z"
  labels:
    kops.k8s.io/cluster: k8s.mydomain.com
  name: master-eu-west-1a
spec:
  image: kope.io/k8s-1.5-debian-jessie-amd64-hvm-ebs-2017-01-09
  machineType: t2.small
  maxSize: 1
  minSize: 1
  role: Master
  subnets:
    - eu-west-1a
```

Edit Instance Group Nodes

To edit the Instance Group of Nodes:

```
$ kops edit ig nodes
```

The output is

```
apiVersion: kops/v1alpha2
kind: InstanceGroup
metadata:
  creationTimestamp: "2017-02-19T14:19:52Z"
  labels:
    kops.k8s.io/cluster: k8s.mydomain.com
  name: nodes
spec:
  image: kope.io/k8s-1.5-debian-jessie-amd64-hvm-ebs-2017-01-09
  machineType: t2.medium
  maxSize: 4
  minSize: 1
  role: Node
  subnets:
    - eu-west-1a
```

Configure Cluster

Once finished, create the actual cluster using:

```
kops update cluster k8s.mydomain.com --yes
```

To upgrade the cluster

```
kops upgrade cluster k8s.mydomain.com
```

To do rolling-update of the instances:

```
kops rolling-update cluster k8s.mydomain.com
```

Section 4.3: Package Management

Section 4.3: Package Management

In this section we'll explain the Package Management with Helm

Why Helm

Use Helm to...

- Find and use popular software packaged as Kubernetes charts
- Share your own applications as Kubernetes charts
- Create reproducible builds of your Kubernetes applications
- Intelligently manage your Kubernetes manifest files
- Manage releases of Helm packages

Why Helm

Helm is a tool that streamlines installing and managing Kubernetes applications. Think of it like apt/yum/homebrew for Kubernetes.

- Helm has two parts: a client (helm) and a server (tiller)
- Tiller runs inside of your Kubernetes cluster, and manages releases (installations) of your charts.
- Helm runs on your laptop, CI/CD, or wherever you want it to run.
- Charts are Helm packages that contain at least two things:
 - A description of the package (Chart.yaml)
 - One or more templates, which contain Kubernetes manifest files
- Charts can be stored on disk, or fetched from remote chart repositories (like Debian or RedHat packages)

Helm Repositories

Helm has repositories where packages can be found:

```
$ helm repo list
NAME      URL
stable    https://kubernetes-charts.storage.googleapis.com/
```

You can add new repositories (e.g. for private charts) in any static site (e.g. S3)

Helm Install

To search for a helm package:

```
@ helm search wordpress
NAME          VERSION DESCRIPTION
stable/wordpress 0.4.2   Web publishing platform for building blogs and ...
```

To install package:

```
@ helm install stable/wordpress
```

Helm Releases

Installed Charts are called Releases. To see actual releases in your cluster:

```
$ helm ls
NAME                REVISION    UPDATED                         STATUS          CHART
dunking-serval      1           Wed Mar 15 02:09:16 2017    DEPLOYED        wordpress-0.4.1
rude-starfish        1           Wed Mar 15 02:01:06 2017    DEPLOYED        redis-0.4.5
```

Helm Upgrades and rollbacks

To install package:

```
$ helm install --name wordpress stable/wordpress --version=0.4.1
```

To see releases:

```
$ helm ls
```

To upgrade package:

```
$ helm upgrade wordpress stable/wordpress
```

To rollback:

```
$ helm rollback wordpress
```

```
$ helm history wordpress
```

REVISION	UPDATED	STATUS	CHART	DESCRIPTION
1	Wed Mar 15 02:15:35 2017	SUPERSEDED	wordpress-0.4.1	Install complet
2	Wed Mar 15 02:16:56 2017	SUPERSEDED	wordpress-0.4.2	Upgrade complet
3	Wed Mar 15 02:17:35 2017	DEPLOYED	wordpress-0.4.1	Rollback to 1

Section 5.1: Authorization

Section 5.1: Authorization

In this section we'll explain the possible Authorization models in Kubernetes

Authentication Options

- Node authorization
- ABAC mode
- RBAC
- Webhook

Objective

We want to have two teams working on the same cluster but isolated in separate namespaces:

- **Alice** belongs to **Blue** team in `blue` namespace
 - **Bob** belongs to **Green** team in `green` namespace
-
- Alice should be:
 - able to see and edit all resources in `blue` namespace
 - unable to see or edit any resources in `green` namespace
 - Bob should be:
 - able to see and edit all resources in `green` namespace
 - unable to see or edit any resources in `blue` namespace

Using RBAC

To enable RBAC, start the apiserver with `--authorization-mode=RBAC`.

To use it in minikube:

```
$ minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

Create Namespaces

Let's create a namespace for team `blue` and a namespace for team `green` :

```
$ kubectl create namespace green  
$ kubectl create namespace blue
```



Create The Alice user certificate

Create a **private key** for alice :

```
$ openssl genrsa -out alice.key 2048
```



Create **certificate sign request** for alice :

```
$ openssl req -new -key alice.key -out alice.csr -subj "/CN=alice/O=harbur"
```



Sign the **certificate** for alice :

```
$ openssl x509 -req -in alice.csr -CA ~/.minikube/ca.crt -CAkey ~/.minikube/ca.key
```



Create The Bob user certificate

Create a **private key** for bob :

```
$ openssl genrsa -out bob.key 2048
```



Create **certificate sign request** for bob :

```
$ openssl req -new -key bob.key -out bob.csr -subj "/CN=bob/O=harbur"
```



Sign the **certificate** for bob :

```
$ openssl x509 -req -in bob.csr -CA ~/.minikube/ca.crt -CAkey ~/.minikube/ca.key -out bob.crt
```



Create Alice Context in Kubectl

Create alice User in kubectl:

```
$ kubectl config set-credentials alice --client-certificate="$(pwd)/alice.crt" --client-key="$(pwd)/alice.key"
```

Create alice Context in kubectl:

```
$ kubectl config set-context alice --cluster=minikube --namespace=blue --user=alice
```

Retrieve Contexts:

```
$ kubectl config get-contexts
CURRENT NAME      CLUSTER   AUTHINFO  NAMESPACE
*          minikube  minikube  minikube
          alice   minikube  alice     blue
```

Create Bob Context in Kubectl

Create bob User in kubectl:

```
$ kubectl config set-credentials bob --client-certificate="$(pwd)/bob.crt" --client-key=bob.key
```

Create bob Context in kubectl:

```
$ kubectl config set-context bob --cluster=minikube --namespace=green --user=bob
```

Retrieve Contexts:

```
$ kubectl config get-contexts
CURRENT NAME      CLUSTER  AUTHINFO  NAMESPACE
*          minikube  minikube  minikube
           alice    minikube  alice     blue
           bob     minikube  bob       green
```

Alice cannot list Pods

```
$ kubectl --context=alice get pods
Error from server (Forbidden): User "alice" cannot list pods in the
namespace "blue". (get pods)
```



NOTE: By default context's namespace is set to `blue`. The above command is identical to the one below:

```
$ kubectl --context=alice get pods --namespace blue
Error from server (Forbidden): User "alice" cannot list pods in the
namespace "blue". (get pods)
```



Create Blue Team Role

Deploy Role `blue-team`:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: blue
  name: blue-team
rules:
- apiGroups: [ "", "extensions", "apps" ]
  resources: [ "deployments", "replicasets", "pods" ]
  verbs: [ "get", "list", "watch", "create", "update", "patch", "delete" ]
```

NOTE: You can also use ["*"] to cover all options

This role gives edit privileges to `deployments`, `replicasets` and `pods` on `blue` namespace:

```
$ kubectl create -f kubernetes/rbac/role-blue-team.yaml
```



Create Green Team Role

Deploy Role `green-team`:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: green
  name: green-team
rules:
- apiGroups: [ "", "extensions", "apps" ]
  resources: [ "deployments", "replicasets", "pods" ]
  verbs: [ "get", "list", "watch", "create", "update", "patch", "delete" ]
```

NOTE: You can also use ["*"] to cover all options

This role gives edit privileges to `deployments`, `replicasets` and `pods` on `green` namespace:

```
$ kubectl create -f kubernetes/rbac/role-green-team.yaml
```



Create Alice to Blue Team Role Binding

Deploy Role Binding `alice-to-blue-team`:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: alice-to-blue-team
  namespace: blue
subjects:
- kind: User
  name: alice
  apiGroup: ""
roleRef:
  kind: Role
  name: blue-team
  apiGroup: ""
```

This role binding grants the `blue-team` role to `alice` user:

```
$ kubectl create -f kubernetes/rbac/rolebinding-alice-to-blue-team.yaml
```



Create Bob to Green Team Role Binding

Deploy Role Binding `bob-to-green-team`:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: bob-to-green-team
  namespace: green
subjects:
- kind: User
  name: alice
  apiGroup: ""
roleRef:
  kind: Role
  name: green-team
  apiGroup: ""
```

This role binding grants the `green-team` role to `bob` user:

```
$ kubectl create -f kubernetes/rbac/rolebinding-bob-to-green-team.yaml
```



Alice

Alice can now see Pods in `blue` namespace:

```
$ kubectl --context=alice get pods
```



And can perform a deploy:

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



And delete the deployment:

```
$ kubectl --context=alice delete deployment nginx
```



But cannot see deployments in `green` namespace:

```
$ kubectl --context=alice get pods --namespace green
```



Neither deploy in `green` namespace:

```
$ kubectl --context=alice run nginx --image=nginx --namespace green
```



Bob

Bob can now see Pods in `green` namespace:

```
$ kubectl --context=bob get pods
```



And can perform a deploy:

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



And delete the deployment:

```
$ kubectl --context=bob delete deployment nginx
```



But cannot see deployments in `blue` namespace:

```
$ kubectl --context=bob get pods --namespace blue
```



Neither deploy in `blue` namespace:

```
$ kubectl --context=bob run nginx --image=nginx --namespace blue
```



Cleanup

To cleanup the environment

```
$ kubectl delete role blue-team --namespace blue
$ kubectl delete role green-team --namespace green
$ kubectl delete rolebinding alice-to-blue-team --namespace blue
$ kubectl delete rolebinding bob-to-green-team --namespace green
$ kubectl config delete-context alice
$ kubectl config delete-context bob
$ kubectl config unset users.alice
$ kubectl config unset users.bob
$ kubectl delete namespace blue green
```



Helm init

Helm needs a ServiceAccount with proper privileges in order to be deployed in a Cluster with RBAC activated:

Create a Service Account `tiller`:

```
$ kubectl -n kube-system create sa tiller
```



Create a ClusterRoleBinding from `kube-system:tiller` service account to `cluster-admin` cluster role

```
$ kubectl create clusterrolebinding tiller --serviceaccount=kube-system:tiller --cluster=cluster-admin
```



Initialize Helm with service account `tiller`:

```
$ helm init --service-account tiller
```



NOTE: add `--upgrade` if you already have tiller running.

Section 5.4: Istio.sh

An open platform to connect, manage, and secure microservices

Istio.sh

In this section we'll see how Istio works

Istio.sh

- Intelligent Routing and Load Balancing
- Resilience Across Languages and Platforms
- Fleet-Wide Policy Enforcement
- In-Depth Telemetry and Reporting

Install Istio

First we need to install the `istioctl` locally:

```
curl -L https://git.io/getLatestIstio | sh -  
export PATH="$PATH:../istio-0.2.10/bin"
```

Then install Istio components in Kubernetes cluster:

```
$ kubectl apply -f istio-0.2.10/install/kubernetes/istio.yaml
```



Install Istio

Ensure the following Kubernetes services are deployed: `istio-pilot`, `istio-mixer`, `istio-ingress`, `istio-egress`.

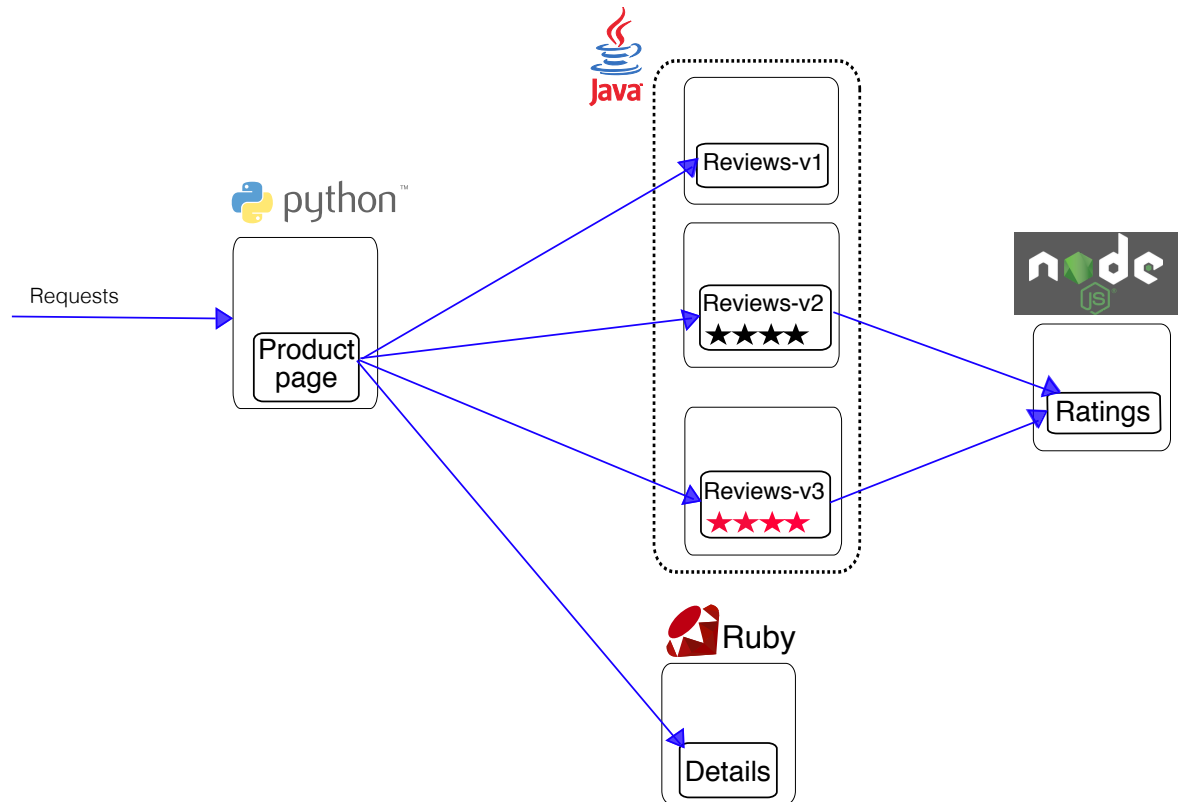
```
$ kubectl get services,pods -n istio-system
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
istio-egress	10.83.247.89	<none>	80/TCP	5h
istio-ingress	10.83.245.171	35.184.245.62	80:32730/TCP,443:30574/TCP	5h
istio-pilot	10.83.251.173	<none>	8080/TCP,8081/TCP	5h
istio-mixer	10.83.244.253	<none>	9091/TCP,9094/TCP,42422/TCP	5h

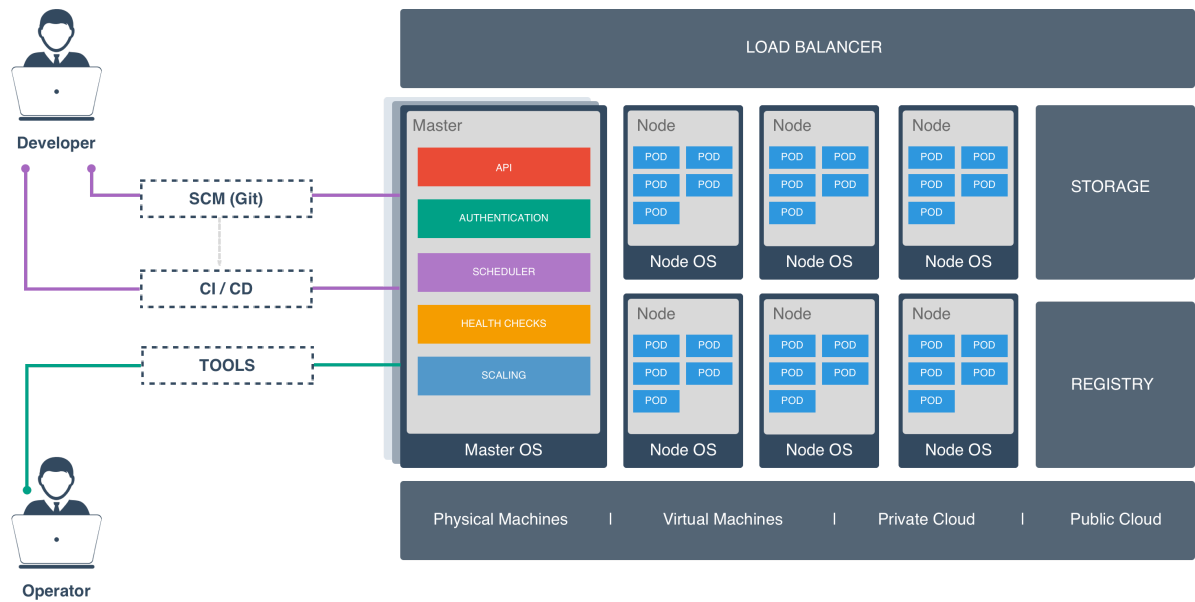
NAME	READY	STATUS	RESTARTS	AGE
istio-ca-3657790228-j21b9	1/1	Running	0	5h
istio-egress-1684034556-fhw89	1/1	Running	0	5h
istio-ingress-1842462111-j3vcs	1/1	Running	0	5h
istio-initializer-184129454-zdgm5	1/1	Running	0	5h
istio-pilot-2275554717-93c43	1/1	Running	0	5h
istio-mixer-2104784889-20rm8	2/2	Running	0	5h



BookInfo Without Istio



BookInfo With Istio



Install BookInfo

Deploy BookInfo demo application

```
$ kubectl apply -f <(istioctl kube-inject -f istio-0.2.10/samples/bookinfo/kube/bookinfo.yaml)
```

Get Services:

```
$ kubectl get services,pods
```

Get Ingress URL:

```
$ kubectl get ingress -o wide
```

Point browser to `http://$GATEWAY_URL/productpage`

If you refresh the page several times, you should see different versions of reviews shown in productpage, presented in a round robin style (red stars, black stars, no stars), since we haven't yet used Istio to control the version routing.

Intelligent Routing

This guide demonstrates how to use various traffic management capabilities of an Istio service mesh.

- **Request routing** This task will first direct all incoming traffic for the Bookinfo application to the v1 version of the reviews service. It will then send traffic only from a specific test user to version v2, leaving all other users unaffected.
- **Fault injection** We will now use Istio to test the resiliency of the Bookinfo application by injecting an artificial delay in requests between the reviews:v2 and ratings services. Observing the resulting behavior as the test user, we will notice that the v2 version of the reviews service has a bug. Note that all other users are unaware of this testing against the live system.
- **Traffic Shifting** Finally, we will use Istio to gradually migrate traffic for all users from to a v3 version of the reviews service, one which includes the fix for the bug discovered in v2.

Request routing

Set the default version for all microservices to v1:

```
$ istioctl create -f istio-0.2.10/samples/bookinfo/kube/route-rule-all-v1.yaml
```

Get Route Rules:

```
$ istioctl get routerules -o yaml
```

Open the BookInfo URL (`http://$GATEWAY_URL/productpage`) in your browser:

You should see the BookInfo application `productpage` displayed. Notice that the `productpage` is displayed with no rating stars since `reviews:v1` does not access the ratings service.

Route a specific user to reviews:v2

```
$ istioctl create -f istio-0.2.10/samples/bookinfo/kube/route-rule-reviews-test-v2
```



View Rule

```
$ istioctl get routerule reviews-test-v2 -o yaml
```



Log in as user “jason” at the productpage web page.

You should now see ratings (1–5 stars) next to each review. Notice that if you log in as any other user, you will continue to see reviews:v1.

Understanding what happened

In this task, you used Istio to send 100% of the traffic to the v1 version of each of the BookInfo services. You then set a rule to selectively send traffic to version v2 of the reviews service based on a header (i.e., a user cookie) in a request.

Once the v2 version has been tested to our satisfaction, we could use Istio to send traffic from all users to v2, optionally in a gradual fashion. We'll explore this in a separate task.

Fault Injection

We will now use Istio to test the resiliency of the Bookinfo application by injecting an artificial delay in requests between the reviews:v2 and ratings services. Observing the resulting behavior as the test user, we will notice that the v2 version of the reviews service has a bug. Note that all other users are unaware of this testing against the live system.

Create latency:

```
$ istioctl create -f istio-0.2.10/samples/bookinfo/kube/route-rule-ratings-test-de: 
```

View Rule:

```
$ istioctl get routerule ratings-test-delay -o yaml 
```

Understanding what happened

The reason that the entire reviews service has failed is because our BookInfo application has a bug. The timeout between the productpage and reviews service is less ($3s + 1 \text{ retry} = 6s$ total) than the timeout between the reviews and ratings service (10s). These kinds of bugs can occur in typical enterprise applications where different teams develop different microservices independently. Istio's fault injection rules help you identify such anomalies without impacting end users.

Notice that we are restricting the failure impact to user "jason" only. If you login as any other user, you would not experience any delays.

Fixing the bug

At this point we would normally fix the problem by either increasing the productpage timeout or decreasing the reviews to ratings service timeout, terminate and restart the fixed microservice, and then confirm that the productpage returns its response without any errors.

However, we already have this fix running in v3 of the reviews service, so we can simply fix the problem by migrating all traffic to reviews:v3 as described in the traffic shifting task.

Traffic Shifting

<https://istio.io/docs/tasks/traffic-management/traffic-shifting.html>

```
$ istioctl delete routerule reviews-test-v2 -n default
```

Add v3 to 50% of traffic:

```
$ istioctl replace -f istio-0.2.10/samples/bookinfo/kube/route-rule-reviews-50-v3.y
```

Add v3 to 100% of traffic:

```
$ istioctl replace -f istio-0.2.10/samples/bookinfo/kube/route-rule-reviews-v3.yam
```

Understanding what happened

In this task we migrated traffic from an old to new version of the reviews service using Istio's weighted routing feature. Note that this is very different than version migration using deployment features of container orchestration platforms, which use instance scaling to manage the traffic. With Istio, we can allow the two versions of the reviews service to scale up and down independently, without affecting the traffic distribution between them.

Course Summary

Course Summary

During the course we learned the following:

- Your Environment
- Kubernetes Architecture
- Running Kubernetes
- Running Containers
- Pulling Images
- Network Basics
- Deployments with no Downtimes
- Configuration Management

Course Summary

During the course we learned the following:

- Persistent Volumes
- Memory and CPU Quotas
- Health Checks
- Kubernetes Addons
- Autoscaling
- Operating Kubernetes
- Package Management with Helm
- Authorization with RBAC
- Istio.sh

Thank you
