Learn DevOps: Kubernetes

Kubernetes

Why Kubernetes

- The salary of DevOps related jobs are sky high, up to $146,207 on average in San Fransisco
- **Containerization** is happening, and **Kubernetes** won the container war!
- Employers are looking for people with **Docker** & **Kubernetes** skills to start using Kubernetes
- **Containerization** and **Kubernetes** are the most important technologies to learn today!

Kubernetes

- If you are in a Ops or DevOps role, you'll want to **start using containers** to deliver software **more efficient**, **easier**, and **faster**
- Kubernetes is a great tool to **run and manage** your containers:
  - You can use Kubernetes on your **desktop**, **on-premise**, or in the **cloud**
  - It gives you all the flexibility and cost savings that you always wanted within one **framework**
  - It can make you **more independent** from cloud vendors
  - **Organizations are adopting Kubernetes** and it is something that will come up in your next interview

Who am I

- My name is Edward Viaene
- I am a **consultant** and **trainer** in Cloud and Big Data technologies
- I'm a big advocate of **Agile** and **DevOps techniques** in all the projects I work on
- I held various roles from **banking** to **startups**
- I have a **background** in Unix/Linux, Networks, Security, Risk, and distributed computing
- Nowadays I specialize in everything that has to do with **Cloud** and **DevOps**

Online Training

- **Online training** on Udemy
  - **DevOps**, **Distributed Computing**, **Cloud, Terraform, Big Data**
  - Using online video lectures
  - 52,000+ enrolled students in 100+ countries
  - Read more and see stats on https://www.udemy.com/user/ward-viaene/

Kubernetes

- What is included in this course?
  - Cluster Setup lectures using **minikube** or the **Docker client** (for desktop usage) and production cluster setup on AWS using **kops**
    - Lectures and demos on **Kubeadm** are also available for **on-prem setups**
  - Kubernetes **Basics** + **Advanced** topics with lots of **demos**
    - Including demos showing you how to run **Wordpress + MySQL** on Kubernetes
  - Kubernetes **administration** topics
  - Package and Deploy applications using **Helm**

Kubernetes

- What version am I using? (don't worry too much about it)
  - I **update this course regularly** to include newer features
  - I always use the latest version available, so depending on the lecture it can be an older version (<1.9) or a newer version
    - You should **always use the latest version available**
    - I update all the scripts and yaml definitions on Github to make sure **they work with the latest version available**
    - If not, you can send me a message and I'll update the script on Github

Course Overview

Introduction	Kubernetes Basics	Advanced topics	Administration	Packaging
What is Kubernetes	Node Architecture	Service auto-discovery	Master Services	Introduction to Helm
Cloud / On-premise setup	Scaling pods	ConfigMap	Quotas and Limits	Creating Helm Charts
Cluster Setup	Deployments	Ingress	Namespaces	Helm Repository
Building Containers	Services	External DNS	User Management	Building & Deploying
Running your first app	Labels	Volumes	RBAC	
Building Container Images	Healthchecks	Pod Presets	Networking	
	ReadinessProbe	StatefulSets	Node Maintenance	
	Pod State & LifeCycle	Daemon Sets	High Availability	
	Secrets	Monitoring	TLS on ELB	
	WebUI	Autoscaling		
		Node Affinity		
		InterPod (Anti-)Affinity		
		Taints and Tolerations		
		Operators		

Extras
kubeadm
TLS Certificates with cert-manager

Kubernetes

Course Introduction

Who am I

- Hi, I'm Edward, and I'll be your trainer for this course
- Getting started with Kubernetes is **no easy task**, so reach out to me using the Q&A if you're stuck somewhere in this course
- **Setting up your Kubernetes cluster for the first time can be hard**, but once you're passed the initial lectures, it will get easier, and you'll deepen your knowledge by learning all the details of Kubernetes
- When you're finished with this course, you can **continue with my 2 other (Advanced) Kubernetes courses**, you'll get a coupon code in the last lecture (bonus lecture)

Course Overview

Introduction	Kubernetes Basics	Advanced topics	Administration	Packaging
What is Kubernetes	Node Architecture	Service auto-discovery	Master Services	Introduction to Helm
Cloud / On-premise setup	Scaling pods	ConfigMap	Quotas and Limits	Creating Helm Charts
Cluster Setup	Deployments	Ingress	Namespaces	Helm Repository
Building Containers	Services	External DNS	User Management	Building & Deploying
Running your first app	Labels	Volumes	RBAC	
Building Container Images	Healthchecks	Pod Presets	Networking	
	ReadinessProbe	StatefulSets	Node Maintenance	
	Pod State & LifeCycle	Daemon Sets	High Availability	
	Secrets	Monitoring	TLS on ELB	
	WebUI	Autoscaling		
		Node Affinity		
		InterPod (Anti-)Affinity		
		Taints and Tolerations		
		Operators		

Extras
kubeadm
TLS Certificates with cert-manager

Course objectives

- To be able to **understand**, **deploy** and **use** Kubernetes
- To get started with **Containerization** and run those containers on Kubernetes
- To deploy Kubernetes on your **desktop**, **on-prem** and on **AWS**
- To be able to run **stateless** and **stateful** applications on Kubernetes
- To be able to **administer** Kubernetes
- To be able to **package and deploy** applications using Helm

Kubernetes

Support and Downloads

Feedback and support

- To provide feedback or get support, use the discussion groups
- We also have a Facebook group called Learn DevOps: Continuously Deliver Better Software
- You can scan the following barcode or use the link in the next document after this introduction movie

Procedure Document

- Use the next procedure document in the next lecture to download all the resources for the course
- All resources are in a github repository
  - You can clone that git repository
  - You can download a zip file on the github website
- Repository URL: https://github.com/wardviaene/kubernetes-course

Introduction

Course Overview

Introduction	Kubernetes Basics	Advanced topics	Administration	3rd party
What is Kubernetes	Node Architecture	Service auto-discovery	Master Services	Flannel
Cloud / On-premise setup	Scaling pods	ConfigMap	Quotas and Limits	Calico
Cluster Setup	Deployments	Ingress	Namespaces	Linkerd
Building Containers	Services	Volumes	User Management	
Running your first app	Labels	Pet Sets	Networking	
Building Container Images	Healthchecks	Daemon Sets	**Node Maintenance**	
	Secrets	Monitoring	High Availability	
	WebUI	Autoscaling	Federation / Security	

What is Kubernetes

What is kubernetes

- Kubernetes is an open source **orchestration** system for Docker containers
  - It lets you schedule **containers** on a cluster of machines
  - You can run **multiple containers** on one machine
  - You can run long running **services** (like web applications)
  - Kubernetes will **manage** the state of these containers
    - Can start the container on specific nodes
    - Will restart a container when it gets killed
    - Can move containers from one node to another node

What is kubernetes

- Instead of just running a few docker containers on one host manually, Kubernetes is a platform that will manage the containers for you
- Kubernetes clusters can start with one node until thousands of nodes
- Some other popular docker orchestrators are:
  - Docker Swarm
  - Mesos

Kubernetes Advantages

- You can run **Kubernetes** anywhere:
  - On-premise (own datacenter)
  - Public (Google cloud, AWS)
  - Hybrid: public & private
- Highly modular
- Open source
- Great community
- Backed by Google

Containers

Virtual Machines vs Containers

Containers on Cloud Providers

Docker

- **Docker** is the most popular container software
  - An alternative to Docker is rkt - which also works with Kubernetes
- Docker **Engine**
  - The Docker runtime
  - Software to make run docker images
- Docker **Hub**
  - Online service to **store** and **fetch** docker images
  - Also allows you to **build docker** images online

Docker Benefits

- **Isolation**: you ship a binary with all the dependencies
  - no more it works on my machine, but not in production
- Closer **parity** between dev, QA, and production environments
- Docker makes development teams able to ship **faster**
- You can run the same docker image, unchanged, on laptops, data center VMs, and Cloud providers.
- Docker uses Linux Containers (a kernel feature) for operating system-level isolation

Containerization

The Kubernetes Journey

A trailmap to guide you

Kubernetes Setup

The difference ways of setting up Kubernetes

Kubernetes setups

- Kubernetes should really be able to run **anywhere**
- But, there are more **integrations** for certain Cloud Providers, like AWS & GCE
  - Things like **Volumes** and **External Load Balancers** work only with **supported** Cloud Providers
- I will first use **minikube** to quickly spin up a local single machine with a Kubernetes cluster
- I'll then show you how to spin up a cluster on AWS using **kops**
  - This tool can be used to spin up a highly available **production cluster**

Kubernetes Setups

- Doing the labs yourself is possible (and highly recommended):
  - Using the AWS Free tier (gives you 750 hours of t2.micro's / month)
    - http://aws.amazon.com
  - Using your local machine
    - Use minikube from https://github.com/kubernetes/minikube
  - Using DigitalOcean
    - Use https://m.do.co/c/007f99ffb902 to get a $10 coupon

Kubernetes Setup

Set up Kubernetes locally

Minikube Setup

- **Minikube** is a tool that makes it easy to run Kubernetes locally
- Minikube runs a single-node Kubernetes cluster inside a Linux VM
- It's aimed on users who want to just test it out or use if for development
- It cannot spin up a production cluster, it's a one node machine with no high availability

Minikube Setup

- It works on **Windows**, **Linux**, and **MacOS**
- You will need **Virtualization Software** installed to run minikube:
  - VirtualBox is free and can be downloaded from www.virtualbox.org
- You can download minikube from https://github.com/kubernetes/minikube
- To launch your cluster you just need to enter (in a shell / terminal / powershell):

```
$ minikube start
```

Demo

Local kubernetes setup using minikube

Demo

Local kubernetes setup using the docker client

Kubernetes Setup

minikube vs docker client vs kops vs kubeadm

minikube / docker client / kops / kubeadm

- There are **multiple tools** to install a kubernetes cluster
- I showed you how to use minikube / docker client, to do local installs
- If you want a **production cluster**, you'll need **different tooling**
  - Minikube and docker client are great for local setups, but not for real clusters
  - Kops and kubeadm are tools to spin up a production cluster
  - You don't need both tools, just one of them

minikube / docker client / kops / kubeadm

- On AWS, the best tool is **kops**
  - At some point AWS EKS (hosted Kubernetes) will be available, at that point this will probably be the preferred option
- For other installs, or if you can't get kops to work, you can use kubeadm
  - Kubeadm is an **alternative approach**, kops is still recommended (on AWS) - you also have AWS integrations with kops automatically
  - The kubeadm lectures can be found at the end of this course, and let you spin up a cluster on DigitalOcean

Kubernetes Setup

Setting up Kubernetes on AWS

Cloud Setup

- To setup Kubernetes on AWS, you can use a tools called **kops**
  - kops stands for **Kubernetes Operations**
- The tool allows you to do production grade Kubernetes **installations**, **upgrades** and **management**
  - I will use this tool to start a Kubernetes cluster on AWS
- There is also a **legacy** tool called kube-up.sh
  - This was a simple tool to bring up a cluster, but is now deprecated, it doesn't create a production ready environment

Cloud Setup

- Kops only works on Mac / Linux
- If you're on windows, you'll need to boot a virtual machine first
- You can use Vagrant to quickly boot up a linux box
- You can download **Virtualbox** from virtualbox.org **Vagrant** from vagrantup.com (you need both)
- Once downloaded, to boot up a new linux VM, just type in cmd/powershell:

```
C:\ mkdir ubuntu
C:\ cd ubuntu
C:\ubuntu> vagrant init ubuntu/xenial64
C:\ubuntu> vagrant up
[…]
```

Demo

Kubernetes setup on AWS

Demo

DNS troubleshooting

Building Containers

Building your own app in Docker

Building containers

- To build containers, you can use **Docker Engine**
- You can download Docker Engine for:
  - Windows: https://docs.docker.com/engine/installation/windows/
  - MacOS: https://docs.docker.com/engine/installation/mac/
  - Linux: https://docs.docker.com/engine/installation/linux/
- Or you can use my vagrant **devops-box** which comes with docker installed
  - In the demos I will always use an ubuntu-xenial box, setup with vagrant

Dockerfile

- Dockerizing a simple nodeJS app only needs a few files:

Dockerfile

```
FROM node:4.6
WORKDIR /app
ADD . /app
RUN npm install
EXPOSE 3000
CMD npm start
```

index.js

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://
%s:%s', host, port);
});
```

package.json

```
{
  "name": "myapp",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node index.js"
  },
  "engines": {
    "node": “^4.6.1”
  },
  "dependencies": {
    "express": "^4.14.0",
  }
}
```

Dockerfile

- To build this project, **docker build** can be used
- Docker build can be executed **manually** or by CI/CD software like **jenkins**
- To build the docker image from the previous slide:

```
$ cd docker-demo
$ ls
Dockerfile   index.js   package.json
$ docker build .
[…]
$
```

- After the docker build process you have built an image that can run the nodejs app

Demo

Your own docker image

Docker Registry

Push containers to Docker Hub

Dockerfile

- You can run the docker app by executing "**docker run**" locally
  - Docker can be run locally for **development** purposes
- To make an image available to Kubernetes, you need to **push the image to a Docker Registry, like Docker Hub**
  - The first step will be to make an account on **Docker Hub**
  - Then you can **push** any locally built images to **the Docker Registry** (where docker images can be stored in)

Dockerfile

- To push an image to Docker Hub:

```
$ docker login
$ docker tag imageid your-login/docker-demo
$ docker push your-login/docker-demo
```

- Or, to immediately tag an image during the build process:

```
$ cd docker-demo
$ ls
Dockerfile   index.js   package.json
$ docker build -t your-login/docker-demo .
[…]
$ docker push your-login/docker-demo
[…]
$
```

Docker remarks

- You can build and deploy any application you want using docker and kubernetes, if you just take into account a few **limitations**:
  - You should only run **one process in one container**
    - Don't try to create one giant docker image for you app, but split it up if necessary
  - All the data in the **container** is **not preserved**, when a container stops, all the changes within a container are lost
    - You can preserve data, using volumes, which is covered later in this course
  - For more tips, check out the 12-factor app methodology at 12factor.net

Docker remarks

- Here are a few official images you might use for your app:
  - https://hub.docker.com/_/nginx/ - webserver
  - https://hub.docker.com/_/php/ - PHP
  - https://hub.docker.com/_/node - NodeJS
  - https://hub.docker.com/_/ruby/ - Ruby
  - https://hub.docker.com/_/python/ - Python
  - https://hub.docker.com/_/openjdk/ - Java

Demo

Pushing docker image to Docker Hub

Running first app

First app

- Let's run our **newly built** application on the new Kubernetes cluster
- Before we can launch a container based on the image, we need to create a **pod definition**
  - **A pod** describes an application running on Kubernetes
  - A pod can contain **one or more tightly coupled containers**, that make up the app
    - Those apps can easily communicate which each other using their local **port numbers**
  - Our app only has **one** container

Create a pod

- Create a file pod-helloworld.yml with the pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
    - containerPort: 3000
```

- Use kubectl to create the pod on the kubernetes cluster:

```
$ kubectl create -f k8s-demo/pod-helloworld.yml
$
```

Useful Commands

Command	Description
kubectl get pod	Get information about all running pods
kubectl describe pod <pod>	Describe one pod
kubectl expose pod <pod> --port=444 --name=frontend	Expose the port of a pod (creates a new service)
kubectl port-forward <pod> 8080	Port forward the exposed pod port to your local machine
kubectl attach <podname> -i	Attach to the pod
kubectl exec <pod> -- command	Execute a command on the pod
kubectl label pods <pod> mylabel=awesome	Add a new label to a pod
kubectl run -i --tty busybox --image=busybox --restart=Never -- sh	Run a shell in a pod - very useful for debugging

Demo

Running the first app

Demo

Useful kubectl commands

Running first app

Setting up a LoadBalancer for our first app

First app

- In a real world scenario, you need to be able to access the app from **outside** the cluster
- On AWS, you can easily add an **external Load Balancer**
- This AWS Load Balancer will route the traffic to the correct pod in Kubernetes
- There are other solutions for other cloud providers that don't have a Load Balancer
  - Your own **haproxy** / **nginx** load balancer in front of your cluster
  - Expose ports directly

First app with Load Balancer

pod: helloworld.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
    - name: nodejs-port
      containerPort: 3000
```

service: helloworld-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: helloworld-service
spec:
  ports:
  - port: 80
    targetPort: nodejs-port
    protocol: TCP
  selector:
    app: helloworld
  type: LoadBalancer
```

- You could now point a hostname like www.example.com to the ELB to reach your pod from the internet

Demo

Running the first app behind a load balancer

Introduction to Kubernetes

Recap

First app

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
    - name: nodejs-port
      containerPort: 3000
```

First app

Workstation
Kubernetes Cluster
Docker Hub
Container image
k8s-example
Pod 1
k8s-example
Pod 1
k8s-example
Container image
k8s-example
$ kubectl port-forward nodehelloworld.example.com 8081:3000

First app

Workstation
er
Container image
k8s-example
Pod 1
k8s-example
Service
apiVersion: v1
kind: Service
metadata:
name: helloworld-service
spec:
ports:
- port: 80
targetPort: nodejs-port
protocol: TCP
selector:
app: helloworld
type: **LoadBalancer**
apiVersion: v1
kind: Service
metadata:
name: helloworld-service
spec:
ports:
- port: 31001
nodePort: 31001
targetPort: nodejs-port
protocol: TCP
selector:
app: helloworld
type: **NodePort**

First app

```
$ kubectl attach nodehelloworld.example.com -i
$ kubectl exec -it nodehelloworld.example.com -- bash
$ kubectl logs nodehelloworld.example.com
```

First app

Workstation
Kubernetes Cluster
Docker Hub
Container image
k8s-example
Pod 1
k8s-example
Pod 1
k8s-example
Pod
busybox
Container image
k8s-example
Container image
busybox
$ kubectl run -i --tty busybox —image=busybox --restart=Never -- sh

Kubernetes Basics

Node Architecture

Architecture overview

Scaling pods

Scaling

- If your application is **stateless** you can horizontally scale it
  - Stateless = your application does't have a **state**, it doesn't **write** any **local files** / keeps local sessions
  - All traditional databases (MySQL, Postgres) are **stateful**, they have database files that can't be split over multiple instances
- Most **web applications** can be made stateless:
  - **Session management** needs to be done outside the container
  - Any files that need to be saved **can't be saved locally** on the container

Scaling

- Our example app is **stateless**, if the same app would run multiple times, it doesn't change state
- For more information about best practices, have a look at 12factor.net
  - or see my course: **Learn DevOps**: Continuously delivering better software / scaling apps on-premise and in the cloud
- Later in this course I'll explain how to use **volumes** to still run stateful apps
  - Those stateful apps can't horizontally scale, but you can run them in a single container and **vertically scale** (allocate more CPU / Memory / Disk)

Scaling

- Scaling in Kubernetes can be done using the **Replication Controller**
- The replication controller will **ensure** a specified number of **pod replicas** will run at all time
- A pods created with the replica controller will **automatically** be **replaced** if they fail, get deleted, or are terminated
- Using the replication controller is also **recommended** if you just want to make sure **1 pod** is always running, even after reboots
    - You can then run a replication controller with just **1 replica**
    - This makes sure that the pod is always running

Scaling

- To replicate our example app 2 times

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: helloworld-controller
spec:
  replicas: 2
  selector:
    app: helloworld
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: wardviaene/k8s-demo
        ports:
        - containerPort: 3000
```

Demo

Horizontally scale a pod with the replication controller

Deployments

Replication Set

- **Replica Set** is the next-generation Replication Controller
- It supports a new selector that can do selection based on **filtering** according a **set of values**
  - e.g. “environment” either “dev” or “qa”
  - not only based on equality, like the Replication Controller
    - e.g. “environment” == “dev”
- This **Replica Set**, rather than the Replication Controller, is used by the Deployment object

Deployments

- A deployment declaration in Kubernetes allows you to do app **deployments** and **updates**
- When using the deployment object, you define the **state** of your application
  - Kubernetes will then make sure the clusters matches your **desired** state
- Just using the **replication controller** or **replication set** might be **cumbersome** to deploy apps
  - The **Deployment Object** is easier to use and gives you more possibilities

Deployments

- With a deployment object you can:
  - **Create** a deployment (e.g. deploying an app)
  - **Update** a deployment (e.g. deploying a new version)
  - Do **rolling updates** (zero downtime deployments)
  - **Roll back** to a previous version
  - **Pause / Resume** a deployment (e.g. to roll-out to only a certain percentage)

Deployments

- This is an example of a deployment:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: wardviaene/k8s-demo
        ports:
        - containerPort: 3000
```

Useful Commands

Command	Description
kubectl get deployments	Get information on current deployments
kubectl get rs	Get information about the replica sets
kubectl get pods --show-labels	get pods, and also show labels attached to those pods
kubectl rollout status deployment/helloworld-deployment	Get deployment status
kubectl set image deployment/helloworld-deployment k8s-demo=k8s-demo:2	Run k8s-demo with the image label version 2
kubectl edit deployment/helloworld-deployment	Edit the deployment object
kubectl rollout status deployment/helloworld-deployment	Get the status of the rollout
kubectl rollout history deployment/helloworld-deployment	Get the rollout history
kubectl rollout undo deployment/helloworld-deployment	Rollback to previous version
kubectl rollout undo deployment/helloworld-deployment --to-revision=n	Rollback to any version version

Demo

A deployment

Services

Services

- **Pods** are very **dynamic**, they come and go on the Kubernetes cluster
  - When using a **Replication Controller**, pods are **terminated** and created during scaling operations
  - When using **Deployments**, when **updating** the image version, pods are **terminated** and new pods take the place of older pods
- That's why Pods should never be accessed directly, but always through a **Service**
- A service is the **logical bridge** between the "mortal" pods and other **services** or **end-users**

Services

- When using the “kubectl expose” command earlier, you created a new Service for your pod, so it could be accessed externally

- Creating a service will create an endpoint for your pod(s):

  - a **ClusterIP**: a virtual IP address only reachable from within the cluster *(this is the default)*

  - a **NodePort**: a port that is the same on each node that is also reachable externally

  - a **LoadBalancer**: a LoadBalancer created by the **cloud provider** that will route external traffic to every node on the NodePort (ELB on AWS)

Services

- The options just shown only allow you to create **virtual IPs** or **ports**
- There is also a possibility to use **DNS names**
  - **ExternalName** can provide a DNS name for the service
  - e.g. for service discovery using DNS
  - This only works when the **DNS add-on** is enabled
- I will discuss this later in a **separate** lecture

Services

- This is an example of a Service definition (also created using kubectl expose):

```
apiVersion: v1
kind: Service
metadata:
  name: helloworld-service
spec:
  ports:
  - port: 31001
    nodePort: 31001
    targetPort: nodejs-port
    protocol: TCP
  selector:
    app: helloworld
  type: NodePort
```

- Note: by default service can only run between ports 30000-32767, but you could change this behavior by adding the --service-node-port-range= argument to the kube-apiserver (in the init scripts)

Demo

A new service

Labels

Labels

- Labels are key/value pairs that can be attached to objects
  - Labels are like **tags** in AWS or other cloud providers, used to tag resources
- You can **label** your **objects**, for instance your pod, following an organizational structure
  - **Key**: environment - **Value**: dev / staging / qa / prod
  - **Key**: department - **Value**: engineering / finance / marketing
- In our previous examples I already have been using labels to tag pods:

```
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
```

Labels

- Labels are **not unique** and **multiple labels** can be added to one object
- Once labels are attached to an object, you can use filters to narrow down results
  - This is called **Label Selectors**
- Using Label Selectors, you can use **matching expressions** to match labels
  - For instance, a particular pod can only run on a node labeled with “environment” equals “development”
  - More complex matching: “environment” in “development” or “qa”

Node Labels

- You can also use labels to tag **nodes**
- Once nodes are tagged, you can use **label selectors** to let pods only run on **specific nodes**
- There are **2 steps** required to run a pod on a specific set of nodes:
  - First you **tag** the node
  - Then you add a **nodeSelector** to your pod configuration

Node Labels

- First step, add a label or multiple labels to your nodes:

```
$ kubectl label nodes node1 hardware=high-spec
$ kubectl label nodes node2 hardware=low-spec
```

- Secondly, add a pod that uses those labels:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
     - containerPort: 3000
  nodeSelector:
    hardware: high-spec
```

Demo

Node Selector using labels

Health Checks

Health checks

- If your application **malfunctions**, the pod and container can still be running, but the application might not work anymore
- To **detect** and **resolve** problems with your application, you can run **health checks**
- You can run 2 different type of health checks
  - Running a **command** in the container **periodically**
  - Periodic checks on a **URL** (HTTP)
- The typical production application behind a load balancer should always have **health checks** implemented in some way to ensure **availability** and **resiliency** of the app

Health checks

- This is how a health check looks like on our example container:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
     - containerPort: 3000
    livenessProbe:
      httpGet:
        path: /
        port: 3000
      initialDelaySeconds: 15
      timeoutSeconds: 30
```

Demo

Performing health checks

Readiness Probe

Readiness Probe

- Besides livenessProbes, you can also use **readinessProbes** on a container within a Pod
- **livenessProbes**: indicates whether a container is **running**
  - If the check fails, the container will be restarted
- **readinessProbes**: indicates whether the container is **ready to serve** requests
  - If the check fails, the container **will not be restarted**, but **the Pod's IP address will be removed from the Service**, so it'll not serve any requests anymore

Readiness Probe

- The **readiness** test will make sure that **at startup**, the pod will only receive traffic when the test succeeds
- You can use these probes **in conjunction**, and you can configure different tests for them
- If your container always exits when something goes wrong, you don't need a livenessProbe
- In general, you configure **both** the livenessProbe and the readinessProbe

Demo

Performing health checks (readinessProbe)

Pod State

Pod State

- In this lecture I'll walk you through the different statuses and states of a Pod and Container:
  - **Pod Status** field: high level status
  - **Pod Condition**: the condition of the pod
  - **Container State**: state of the container(s) itself
- I'll then show you the lifecycle of a pod in the next lecture

Pod State

- Pods have a status field, which you see when you do *kubectl get pods*:

```
$ kubectl get pods -n kube-system
NAME                                                          READY     STATUS    RESTARTS   AGE
dns-controller-7cc97fb976-4b9nt                               1/1       Running   0          4h
etcd-server-events-ip-172-20-38-169.eu-west-1.compute.internal   1/1       Running   0          4h
etcd-server-ip-172-20-38-169.eu-west-1.compute.internal          1/1       Running   0          4h
kube-apiserver-ip-172-20-38-169.eu-west-1.compute.internal       1/1       Running   0          4h
```

- In this scenario all pods are in the **running status**
  - This means that the **pod has been bound to a node**
  - All **containers have been created**
  - **At least one container** is still **running**, or is starting/restarting

Pod State

- Other valid statuses are:
  - **Pending**: Pod has been **accepted** but is **not running**
    - Happens when the container image is still **downloading**
    - If the pod cannot be scheduled because of **resource constraints**, it'll also be in this status
  - **Succeeded**: All containers within this pod have been **terminated successfully** and will not be restarted

Pod State

- Other valid statuses are:
  - **Failed:** All containers within this pod have been **Terminated**, and at least one container returned a failure code
    - The failure code is the **exit code** of the process when a container terminates
  - **Unknown:** The **state of the pod couldn't be determined**
    - A **network error** might have been occurred (for example the node where the pod is running on is down)

Pod State

- You can get the pod conditions using kubectl describe pod PODNAME

```
$ kubectl describe pod kube-apiserver-ip-172-20-38-169.eu-west-1.compute.internal -n kube-system
[…]
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
```

- These are conditions which the pod has passed
  - In this example, Initialized, Ready, and PodScheduled

Pod State

- There are 5 different types of PodConditions:
  - **PodScheduled**: the pod has been scheduled to a node
  - **Ready**: Pod can serve requests and is going to be added to matching Services
  - **Initialized**: the initialization containers have been started successfully
  - **Unschedulable**: the Pod can't be scheduled (for example due to resource constraints)
  - **ContainersReady**: all containers in the pod are ready

Pod State

- There's also a **container** state:

```
$ kubectl get pod kube-apiserver-ip-172-20-38-169.eu-west-1.compute.internal -n kube-system -o yaml
[…]
  containerStatuses:
  - containerID: docker://7399a5ffb84ac91bf64f54c2395ed632736ef284d11b784ec827fd9d0a56083f
    image: gcr.io/google_containers/kube-apiserver:v1.9.8
    imageID: docker-pullable://gcr.io/google_containers/kube-
apiserver@sha256:79d444e6cb940079285109aaa5f6a97e5c0a5568f6606e003ed279cd90bcf1ca
    lastState: {}
    name: kube-apiserver
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: 2018-08-06T08:12:14Z
[…]
```

- Container state can be Running, Terminated, or Waiting

Pod Lifecycle

Pod Lifecycle

Demo

Pod lifecycle

Secrets

Secrets

- Secrets provides a way in Kubernetes to distribute **credentials**, **keys**, **passwords** or "**secret**" **data** to the pods
- Kubernetes itself uses this Secrets mechanism to provide the credentials to access the internal API
- You can also use the **same mechanism** to provide secrets to your application
- Secrets is one way to provide secrets, native to Kubernetes
  - There are still **other ways** your container can get its secrets if you don't want to use Secrets (e.g. using an **external vault services** in your app)

Secrets

- Secrets can be used in the following ways:
  - Use secrets as **environment variables**
  - Use secrets **as a file** in a pod
    - This setup uses **volumes** to be mounted in a container
    - In this volume you have **files**
    - Can be used for instance for **dotenv** files or your app can just read this file
  - Use an **external image** to pull secrets (from a **private image registry**)

Secrets

- To generate secrets using files:

```
$ echo -n "root" > ./username.txt
$ echo -n "password" > ./password.txt
$ kubectl create secret generic db-user-pass --from-file=./username.txt —from-file=./password.txt
secret "db-user-pass" created
$
```

- A secret can also be an SSH key or an SSL certificate

```
$ kubectl create secret generic ssl-certificate --from-file=ssh-privatekey=~/.ssh/id_rsa --ssl-cert-=ssl-cert=mysslcert.crt
```

Secrets

- To generate secrets using yaml definitions:

secrets-db-secret.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  password: cm9vdA==
  username: cGFzc3dvcmQ=
```

```
$ echo -n "root" | base64
cm9vdA==
$ echo -n "password" | base64
cGFzc3dvcmQ=
```

- After creating the yml file, you can use kubectl create:

```
$ kubectl create -f secrets-db-secret.yml
secret “db-secret" created
$
```

Using secrets

- You can create a pod that exposes the secrets as environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
    - containerPort: 3000
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-secret
            key: username
      - name: SECRET_PASSWORD
        [...]
```

Using secrets

- Alternatively, you can provide the secrets in a file:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
    - containerPort: 3000
    volumeMounts:
    - name: credvolume
      mountPath: /etc/creds
      readOnly: true
  volumes:
  - name: credvolume
    secret:
      secretName: db-secrets
```

The secrets will be stored in:
/etc/creds/db-secrets/username
/etc/creds/db-secrets/password

Demo

Secrets

Demo

Wordpress

Web UI

Web UI

- Kubernetes comes with a **Web UI** you can use instead of the kubectl commands
- You can use it to:
  - Get an **overview** of running applications on your cluster
  - **Creating** and **modifying** individual Kubernetes **resources** and **workloads** (like kubectl create and delete)
  - Retrieve information on the **state** of **resources** (like kubectl describe pod)

Web UI

- In general, you can access the kubernetes Web UI at https://<kubernetes-master>/ui

- If you cannot access it (for instance if it is not enabled on your deploy type), you can install it manually using:

```
$ kubectl create -f https://rawgit.com/kubernetes/dashboard/master/src/deploy/kubernetes-dashboard.yaml
```

- If a password is asked, you can retrieve the password by entering:

```
$ kubectl config view
```

Web UI

- If you are using minikube, you can use the following command to launch the dashboard:

```
$ minikube dashboard
```

- Or if you just want to know the url:

```
$ minikube dashboard --url
```

Demo

Web UI - Kops

Demo

Web UI

Advanced topics

Service Discovery

Using DNS

DNS

- As of Kubernetes 1.3, DNS is a **built-in** service launched automatically using the addon manager
  - The addons are in the /etc/kubernetes/addons **directory** on **master node**
- The DNS service can be used within pods to **find other services** running on the same cluster
- Multiple containers **within 1 pod** don't need this service, as they can **contact** each other **directly**
  - A container in the same pod can connect the the port of the other container directly using **localhost:port**
- To make DNS work, a pod will need a **Service definition**

DNS

- An example of how app 1 could reach app 2 using DNS:

```
$ host app1-service
app1-service has address 10.0.0.1
$ host app2-service
app2-service has address 10.0.0.2
$ host app2-service.default
app2-service.default has address 10.0.0.2
$ host app2-service.default.svc.cluster.local
app2-service.default.svc.cluster.local has address 10.0.0.2
```

Default stands for the default namespace
Pods and services can be launched in different namespaces (to logically separate your cluster)

DNS - How does it work?

```
$ kubectl run -i --tty busybox --image=busybox --restart=Never -- sh
Waiting for pod default/busybox to be running, status is Pending, pod ready: false
If you don't see a command prompt, try pressing enter.
/ # cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local lan
nameserver 10.0.0.10
options ndots:5
```

Demo

Service Discovery

ConfigMap

ConfigMap

- Configuration parameters that are not secret, can be put in a **ConfigMap**
- The input is **again** key-value pairs
- The ConfigMap **key-value pairs** can then be read by the app using:
  - **Environment** variables
  - **Container commandline arguments** in the Pod configuration
  - Using **volumes**

ConfigMap

- A ConfigMap can also contain **full configuration** files
  - e.g. an webserver config file
- This file can then be **mounted** using volumes where the application expects its config file
- This way you can "**inject**" configuration settings into containers without changing the container itself

ConfigMap

- To generate configmap using files:

```
$ cat <<EOF > app.properties
driver=jdbc
database=postgres
lookandfeel=1
otherparams=xyz
param.with.hierarchy=xyz
EOF
$ kubectl create configmap app-config —from-file=app.properties
$
```

Using ConfigMap

- You can create a pod that exposes the ConfigMap using a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
    - containerPort: 3000
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: app-config
```

The config values will be stored in files:
/etc/config/driver
/etc/config/param/with/hierarchy

Using ConfigMap

- You can create a pod that exposes the ConfigMap as environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
    - containerPort: 3000
    env:
      - name: DRIVER
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: driver
      - name: DATABASE
      [...]
```

Demo

ConfigMap

Ingress

Ingress

- Ingress is a solution available since Kubernetes 1.1 that allows **inbound connections** to the cluster
- It's an alternative to the external **Loadbalancer** and **nodePorts**
  - Ingress allows you to **easily expose services** that need to be accessible from **outside** to the **cluster**
- With ingress you can run your own **ingress controller** (basically a loadbalancer) within the Kubernetes cluster
- There are a default ingress controllers available, or you can **write your own** ingress controller

Ingress

Ingress rules:

host-x.example.com => pod 1

host-y.example.com => pod 2

host-x.example.com/api/v2 => pod n

Ingress rules

- You can create ingress rules using the ingress object

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: helloworld-rules
spec:
  rules:
  - host: helloworld-v1.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: helloworld-v1
          servicePort: 80
  - host: helloworld-v2.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: helloworld-v2
          servicePort: 80
```

Demo

Ingress Controller

External DNS

External DNS

- In the previous lecture I explained you how to **setup an ingress controller**
- On public cloud providers, you can use the ingress controller **to reduce the cost of your LoadBalancers**
  - You can use 1 LoadBalancer that **captures** all the **external traffic** and send it to the ingress controller
  - The ingress controller can be configured to **route the different traffic** to all your apps based on HTTP rules (host and prefixes)
  - This only works for **HTTP(s)-based** applications

External DNS

- One great tool to enable such approach is **External DNS**
- This tool will **automatically create the necessary DNS records** in your external DNS server (like route53)
- For **every hostname** that you use in **ingress**, it'll create a new record to send traffic to your loadbalancer
- The **major DNS providers** are supported: Google CloudDNS, Route53, AzureDNS, CloudFlare, DigitalOcean, etc
- **Other setups** are also possible without ingress controllers (for example directly on hostPort - nodePort is still WIP, but will be out soon)

Ingress with LB and External-DNS

Demo

External DNS

Volumes

Running apps with state

Volumes

- Volumes in kubernetes allow you to **store data outside the container**
- When a container **stops**, all data on the container itself is **lost**
  - That's why up until now I've been using **stateless** apps: apps that don't keep a **local** state, but store their state in an **external service**
    - External Service like a database, caching server (e.g. MySQL, AWS S3)
- Persistent Volumes in Kubernetes allow you **attach a volume** to a container that will **exists** even when the **container** stops

Kubernetes Volumes

- Volumes can be attached using different **volume plugins**:

Volumes

- Using volumes, you could deploy **applications with state** on your cluster
  - Those applications need to read/write to files on the **local filesystem** that need to be persistent in time
- You could run a **MySQL** database using persistent volumes
  - Although this might not be ready for production (yet)
  - Volumes are new since the June 2016 release in Kubernetes, so depending when you’re taking this course - you still might want to be **careful** about this

Volumes

- If your **node stops** working, the pod can be rescheduled on another node, and the volume can be attached to the new node

Volumes

- To use volumes, you need to **create the volume** first

```
$ aws ec2 create-volume --size 10 --region us-east-1 --availability-zone us-east-1a --volume-type gp2
{
    "VolumeId": "vol-055681138509322ee",
    "VolumeType": "gp2",
    "Encrypted": false,
    "CreateTime": "2016-11-08T13:51:33.317Z",
    "AvailabilityZone": "eu-west-1a",
    "Size": 10,
    "SnapshotId": "",
    "Iops": 100,
    "State": "creating"
}
```

Note down this VolumeID

- This will create a 10 GB volume in us-east-1a

- Tip: the nodes where your pod is going to run on also need to be in the **same availability zone**

Volumes

- To use volumes, you need to **create a pod** with a volume definition

```
[…]
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    volumeMounts:
    - mountPath: /myvol
      name: myvolume
    ports:
    - containerPort: 3000
  volumes:
  - name: myvolume
    awsElasticBlockStore:
      volumeID: vol-055681138509322ee
```

Demo

Using volumes

Volumes

Provisioning

Volumes

- The kubernetes plugins have the capability to **provision storage** for you
- The **AWS Plugin** can for instance **provision storage** for you by creating the volumes in AWS before attaching them to a node
- This is done using the **StorageClass** object
  - This is still in beta when writing this course, but will be stable soon
  - It's best to double check the correct definitions in the documentation (http://kubernetes.io/docs/user-guide/persistent-volumes/)
  - I'll also keep my github repository up to date with the latest definitions

Volumes

- To use **auto provisioned volumes**, you can create the following yaml file:

storage.yml

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  zone: us-east-1
```

- This will allow you to create volume claims using the **aws-ebs provisioner**

- Kubernetes will provision volumes of the type **gp2** for you (General Purpose - SSD)

Volumes

- Next, you can create a volume claim and specify the size:

my-volume-claim.yml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
  annotations:
    volume.beta.kubernetes.io/storage-class: "standard"
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

Volumes

- Finally, you can launch a pod using a volume:

my-pod.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

Demo

Wordpress with Volumes

Pod Presets

Pod Presets

- Pod presets can **inject information into pods at runtime**
  - Pod Presets are used to **inject Kubernetes Resources** like Secrets, ConfigMaps, Volumes and Environment variables
- Imagine you have 20 applications you want to deploy, and they all need to get a specific credential
  - You can edit the 20 specifications and add the credential, or
  - You can use presets to create 1 Preset object, which will inject an environment variable or config file **to all matching pods**
- When **injecting** Environment variables and VolumeMounts, the Pod Preset will **apply the changes** to **all containers** within the pod

Pod Presets

- This is an example of a Pod Preset

```
apiVersion: settings.k8s.io/v1alpha1   # you might have to change this after PodPresets become stable
kind: PodPreset
metadata:
  name: share-credential
spec:
  selector:
    matchLabels:
      app: myapp
  env:
    - name: MY_SECRET
      value: "123456"
  volumeMounts:
    - mountPath: /share
      name: share-volume
  volumes:
    - name: share-volume
      emptyDir: {}
```

Pod Presets

- You can use **more than one PodPreset**, they'll all be applied to matching Pods
- If there's a **conflict**, the PodPreset will **not be applied** to the pod
- PodPresets can match **zero or more Pods**
  - It's possible that no pods are currently matching, but that matching pods will be launched at a later time

Demo

Pod Presets

StatefulSets

Stateful distributed apps on a Kubernetes cluster

StatefulSets

- Pet Sets was a **new feature** starting from Kubernetes 1.3, and got renamed to StatefulSets which is stable since Kubernetes 1.9
- It is introduced to be able to run **stateful applications**:
  - That need a **stable pod hostname** (instead of podname-randomstring)
    - Your podname will have a sticky identity, using an index, e.g. podname-0 podname-1 and podname-2 (and when a pod gets rescheduled, it'll keep that identity)
  - Statefulsets allow **stateful apps stable storage** with volumes based on their ordinal number (podname-**x**)
    - **Deleting** and/or **scaling** a **StatefulSet down** will not delete the volumes associated with the StatefulSet (preserving data)

StatefulSets

- A StatefulSet will allow your stateful app to use **DNS** to find other **peers**
    - Cassandra clusters, ElasticSearch clusters, use **DNS** to find other members of the cluster
        - for example: **cassandra-0.cassandra** for all pods to reach the first node in the cassandra cluster
    - Using StatefulSet you can run for instance 3 cassandra nodes on Kubernetes named cassandra-0 until cassandra-2
    - If you wouldn't use StatefulSet, you would get a dynamic hostname, which you wouldn't be able to use in your configuration files, as the name can always change

StatefulSets

- A StatefulSet will also allow your stateful app to **order the startup and teardown**:
  - Instead of randomly terminating one pod (one instance of your app), you'll know which one that will go
    - When **scaling up** it goes from 0 to n-1 (n = replication factor)
    - When **scaling down** it starts with the highest number (n-1) to 0
  - This is useful if you first need to **drain** the data from a node before it can be shut down

Demo

StatefulSets - Cassandra

Daemon Sets

Daemon Sets

- Daemon Sets ensure that **every single node** in the Kubernetes cluster runs the same pod resource
  - This is useful if you want to **ensure** that a certain pod is running on every single kubernetes node
- When a node is **added** to the cluster, a new pod will be **started** automatically
- Same when a node is **removed**, the pod will not be **rescheduled** on another node

Daemon Sets

- Typical **use cases**:
  - Logging aggregators
  - Monitoring
  - Load Balancers / Reverse Proxies / API Gateways
  - Running a daemon that only needs one instance per physical instance

Daemon Sets

- This is an example Daemon Set specification:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: monitoring-agent
  labels:
    app: monitoring-agent
spec:
  template:
    metadata:
      labels:
        name: monitor-agent
    spec:
      containers:
      - name: k8s-demo
        image: wardviaene/k8s-demo
        ports:
        - name: nodejs-port
          containerPort: 3000
```

Resource Usage Monitoring

Resource Usage Monitoring

- Heapster enables **Container Cluster Monitoring** and **Performance Analysis**
- It's providing a monitoring platform for Kubernetes
- It's a prerequisite if you want to do **pod auto-scaling** in Kubernetes
- Heapster exports clusters metrics **via REST endpoints**
- You can use **different backends** with Heapster
  - I'll use **InfluxDB**, but others like Google Cloud Monitoring/Logging and Kafka are also possible

Resource Usage Monitoring

- **Visualizations** (graphs) can be shown using Grafana
  - The Kubernetes dashboard will also show graphs once monitoring is enabled
- All these technologies (Heapster, InfluxDB, and Grafana) can be started in pods
- The **yaml files** can be found on the github repository of Heapster
  - https://github.com/kubernetes/heapster/tree/master/deploy/kube-config/influxdb
  - After downloading the repository the whole platform can be deployed using the addon system or by using kubectl create -f directory-with-yaml-files/

Resource Usage Monitoring

Demo

Using Metrics Server (Kubernetes 1.8+)

Demo

Setting up heapster with influxdb and Grafana

Autoscaling

Horizontal Pod Autoscaling

Autoscaling

- Kubernetes has the possibility to **automatically scale pods** based on metrics
- Kubernetes can automatically scale a Deployment, Replication Controller or ReplicaSet
- In Kubernetes 1.3 **scaling based on CPU** usage is possible out of the box
  - With alpha support, application based metrics are also available (like queries per second or average request latency)
    - To enable this, the cluster has to be started with the env var ENABLE_CUSTOM_METRICS to true

Autoscaling

- Autoscaling will **periodically query** the utilization for the targeted pods
  - **By default 30 sec**, can be changed using the "—horizontal-pod-autoscaler-sync-period" when launching the controller-manager
- Autoscaling will use **heapster**, the monitoring tool, to gather its metrics and make scaling decisions
  - Heapster must be installed and running before autoscaling will work

Autoscaling

- An **example**:
  - You run a **deployment** with a **pod** with a **CPU resource** request of **200m**
  - 200m = 200 millicpu (or also 200 millicores)
  - 200m = 0.2, which is 20% of a CPU core of the running node
    - If the node has 2 cores, it's still 20% of a single core
  - You introduce auto-scaling at 50% of the CPU usage (which is 100m)
  - Horizontal Pod Autoscaling will increase/descrease pods to maintain a target CPU utilization of 50% (or 100m / 10% of a core within this pod)

Autoscaling

- This is a pod that you can use to test autoscaling:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hpa-example
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: hpa-example
    spec:
      containers:
      - name: hpa-example
        image: gcr.io/google_containers/hpa-example
        ports:
        - name: http-port
          containerPort: 80
        resources:
          requests:
            cpu: 200m
```

Autoscaling

- This is an example autoscaling specification:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-example-autoscaler
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: hpa-example
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Demo

Autoscaling

Affinity and anti-affinity

Affinity and anti-affinity

- In a previous demo I showed you how to use nodeSelector to make sure pods get scheduled on specific nodes:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: wardviaene/k8s-demo
        […]
     nodeSelector:
        hardware: high-spec
```

Affinity and anti-affinity

- The affinity/anti-affinity feature allows you to do **more complex scheduling** than the nodeSelector and also **works on Pods**
  - The language is **more expressive**
  - You can create **rules that are not hard requirements**, but rather a **preferred rule**, meaning that the scheduler will still be able to schedule your pod, even if the rules cannot be met
  - You can create rules that take other pod labels into account
    - For example, a rule that makes sure 2 different pods will never be on the same node

Affinity and anti-affinity

- Kubernetes can do **node affinity** and **pod affinity/anti-affinity**
  - Node affinity is similar to the nodeSelector
  - Pod affinity/anti-affinity allows you to create rules **how pods should be scheduled taking into account other running pods**
  - Affinity/anti-affinity mechanism is **only relevant during scheduling**, once a pod is running, it'll need to be recreated to apply the rules again
- I'll first cover **node affinity** and will then cover pod affinity/anti-affinity

Affinity and anti-affinity

- There are currently 2 types you can use for node affinity:
  - 1) requiredDuringSchedulingIgnoredDuringExecution
  - 2) preferredDuringSchedulingIgnoredDuringExecution
- The **first one** sets a **hard requirement** (like the nodeSelector)
  - The rules must be met before the pod can be scheduled
- The **second type** will try to enforce the rule, but it will not guarantee it
  - Even if the rule is not met, the pod can still be scheduled, it's a soft requirement, a preference

Affinity and anti-affinity

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: env
            operator: In
            values:
            - dev
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: team
            operator: In
            values:
            - engineering-project1
  containers:
  […]
```

Affinity and anti-affinity

- I also supplied **a weighting** to the preferredDuringSchedulingIgnoredDuringExecution statement
- The **higher this weighting**, the **more weight is given to that rule**
- When scheduling, Kubernetes will score every node by summarizing the weightings per node
  - For example if you have **2 different rules with weights 1 and 5**
  - If both rules match, the node will have a score of **6**
  - If only the rule with weight **1 matches**, then the score will **only be 1**
- The node that has the **highest total score**, that's where the pod will be scheduled on

Built-in node labels

- In addition to the labels that you can add yourself to nodes, there are **pre-populated labels** that you can use:
  - kubernetes.io/hostname
  - failure-domain.beta.kubernetes.io/zone
  - failure-domain.beta.kubernetes.io/region
  - beta.kubernetes.io/instance-type
  - beta.kubernetes.io/os
  - beta.kubernetes.io/arch

Affinity and anti-affinity

Demo

Interpod Affinity and anti-affinity

Interpod Affinity and anti-affinity

- This mechanism allows you to **influence scheduling based on the labels of other pods** that are **already running** on the cluster

- **Pods belong to a namespace**, so your affinity rules will **apply to a specific namespace** (if no namespace is given in the specification, it defaults to the namespace of the pod)

- Similar to node affinity, you have **2 types** of pod affinity / anti-affinity:

  - requiredDuringSchedulingIgnoredDuringExecution

  - preferredDuringSchedulingIgnoredDuringExecution

- The **required type** creates **rules that *must* be met** for the pod to be scheduled, the **preferred type** is a "**soft**" type, and the **rules *may* be met**

Interpod Affinity and anti-affinity

- A good use case for **pod affinity** is **co-located pods**:
  - You might want that 1 pod is always co-located on the same node with another pod
  - For example you have an app that uses redis as cache, and you want to have the redis pod on the same node as the app itself
- Another use-case is to co-locate pods within the **same availability zone**

Interpod Affinity and anti-affinity

- When writing your pod affinity and anti-affinity rules, you need to specify a **topology domain**, called **topologyKey** in the rules
- The topologyKey refers to a node label
- If the affinity rule matches, the **new pod** will only be **scheduled** on **nodes** that have the **same topologyKey** value as the **current running pod**

Interpod Affinity and anti-affinity

new pod
app=redis

```
podAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
          - key: "app"
            operator: In
            values:
            - myapp
      topologyKey: "kubernetes.io/hostname"
```

Interpod Affinity and anti-affinity

```
podAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
          - key: "app"
            operator: In
            values:
            - myapp
      topologyKey: "failure-domain.beta.kubernetes.io/zone"
```

Pod Affinity and anti-affinity

- Contrary to affinity, you might want to use **pod anti-affinity**
- You can use anti-affinity to make sure a **pod is only scheduled once on a node**
  - For example you have 3 nodes and you want to schedule 2 pods, but they shouldn't be scheduled on the same node
  - Pod anti-affinity allows you to create a rule that says to **not schedule on the same host if a pod label matches**

Interpod Affinity and anti-affinity

new pod
app=db

```
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
          - key: "app"
            operator: In
            values:
            - myapp
      topologyKey: "kubernetes.io/hostname"
```

Node1

new pod
app=db

Node2

pod
app=myapp

Interpod Affinity and anti-affinity

- When writing pod affinity rules, you can use the following operators:
  - **In, NotIn** (does a label have one of the values)
  - **Exists, DoesNotExist** (does a label exist or not)
- Interpod affinity and anti-affinity currently requires a substantial amount of processing
  - You might have to take this into account if you have a lot of rules and a larger cluster (e.g. 100+ nodes)

Interpod affinity

Demo

Pod Anti-Affinity

Demo

Taints and tolerations

Taints and tolerations

- In the previous lectures I explained you the following concepts:
  - **Node** affinity (similar to the nodeSelector)
  - **Interpod** affinity / anti-affinity
- The next concept, **tolerations**, is the opposite of node affinity
  - Tolerations allow a node to **repel a set of pods**
  - **Taints mark** a node, tolerations are applied to pods to influence the scheduling of the pods

Taints and tolerations

- One use case for taints is to make sure that when you create a new pod, they're not scheduled on the master
  - **The master has a taint**: (node-role.kubernetes.io/master:NoSchedule)
- To add a new taint to a node, you can use kubectl taint:

```
kubectl taint nodes node1 key=value:NoSchedule
```

- This will make sure that **no pods will be scheduled** on node1, as long as they **don't have a matching toleration**

Taints and tolerations

- The following toleration would allow a new pod to be scheduled on the tainted node1:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

- You can use the following operators:
    - Equal: providing a key & value
    - Exists: only providing a key, checking only whether a key exists

Taints and tolerations

- Just like affinity, **taints** can also be a **preference** (or “soft”) rather than a requirement:
  - **NoSchedule**: a hard requirement that a pod will not be scheduled unless there is a matching toleration
  - **PreferNoSchedule**: Kubernetes will try and avoid placing a pod that doesn’t have a matching toleration, but it’s not a hard requirement
- If the taint is applied while there are **already running pods**, these will **not be evicted**, unless the following taint type is used:
  - **NoExecute**: **evict** pods with non-matching tolerations

Taints and tolerations

- When using **NoExecute**, you can specify within your toleration **how long the pod can run** on a **tainted node** before being evicted:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: “NoExecute"
  tolerationSeconds: 3600
```

- If you don’t specify the tolerationSeconds, the toleration will match and the pod will keep running on the node

- In this example, the toleration will **only match for 1 hour** (3600 seconds), after that the **pod will be evicted** from the node

Taints and tolerations

- Example **use cases** are:
  - The existing node taints for **master nodes**
  - Taint nodes that are **dedicated** for a **team or a user**
  - If you have a few nodes with **specific hardware** (for example GPUs), you can taint them to avoid running non-specific applications on those nodes
  - An alpha (but soon to be beta) feature is to **taint nodes by condition**
    - This will automatically taint nodes that have node problems, allowing you to add tolerations to time the eviction of pods from nodes

Taints and tolerations

- You can enable alpha features by passing the --feature-gates to the Kubernetes controller manager, or in kops, you can use **kops edit** to add:

```
spec:
  kubelet:
    featureGates:
      TaintNodesByCondition: "true"
```

- In the next slide I'll show you a few taints that can be possibly added.

- This is an example of a toleration that could be used:

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Taints and tolerations

- **node.kubernetes.io/not-ready**: Node is not ready
- **node.kubernetes.io/unreachable**: Node is unreachable from the node controller
- **node.kubernetes.io/out-of-disk**: Node becomes out of disk.
- **node.kubernetes.io/memory-pressure**: Node has memory pressure.
- **node.kubernetes.io/disk-pressure**: Node has disk pressure.
- **node.kubernetes.io/network-unavailable**: Node's network is unavailable.
- **node.kubernetes.io/unschedulable**: Node is unschedulable.

Taints and tolerations

Demo

Custom Resource Definitions

Custom Resource Definitions

- **Custom Resource Definitions** lets you extend the Kubernetes API
- Resources are the **endpoints in the Kubernetes API** that store collections of API Objects
  - For example, there is the built-in Deployment resource, that you can use to deploy applications
  - In the **yaml files** you **describe the object**, using the **Deployment resource type**
  - You **create** the **object on the cluster** by using **kubectl**

Custom Resource Definitions

- A Custom Resource is a resource that you might add to your cluster, it's not available on every cluster
- It's an **extension of the Kubernetes API**
- Custom Resources are **also described in yaml files**
- As an administrator you can dynamically add CRDs (Custom Resource Definitions) to add extra functionality to your cluster
- Operators, explained in the next lecture, use these CRDs to extend the Kubernetes API with their own functionality

Operators

Operators

- An Operator is a method of **packaging**, **deploying**, and **managing** a Kubernetes Application (definition: https://coreos.com/operators/)

- It puts **operational knowledge** in an application

  - It brings the user **closer to the experience of managed cloud services**, rather than having to know all the specifics of an application deployed to Kubernetes

  - Once an Operator is deployed, it can be **managed using Custom Resource Definitions** (arbitrary types that extend the Kubernetes API)

- It also provides a great way to deploy Stateful services on Kubernetes (because a lot of complexities can be hidden from the end-user)

Operators

- **Any third party can create operators** that you can start using
- There are **operators** for Prometheus, Vault, Rook (storage), MySQL, PostgreSQL, and so on
- In the demo, I'll show you how to start using an **Operator for PostgreSQL**
- If you'd just deploy a PostgreSQL container, it'd only start the database
- If you're going to use this **PostgreSQL operator**, it'll allow you to also **create replicas, initiate a failover, create backups, scale**
    - An operator contains a lot of the **management logic** that you as an administrator or user might want, rather than having to implement it yourself

Operators

- **Example yaml extract:**

```yaml
apiVersion: cr.client-go.k8s.io/v1
kind: Pgcluster
metadata:
  labels:
    archive: "false"
    archive-timeout: "60"
    crunchy_collect: "false"
    name: mycluster
    pg-cluster: mycluster
    primary: "true"
  name: mycluster
  namespace: default
```

Postgres-operator

Demo

Cron Job

Scheduling recurring jobs

Cron Jobs

- Still in alpha
- Check again when 1.5 comes out

Administration

Kubernetes Administration

Master Services

Architecture overview

Resource Quotas

Resource Quotas

- When a Kubernetes cluster is used by multiple **people** or **teams**, **resource management** becomes more important
  - You want to be able to **manage the resources** you give to a person or a team
  - You don't want one person or team **taking up all the resources** (e.g. CPU/Memory) of the cluster
- You can divide your cluster in **namespaces** (explained in next lecture) and enable resource quotas on it
  - You can do this using the **ResourceQuota** and **ObjectQuota** objects

Resource Quotas

- Each container can specify **request capacity** and **capacity limits**
  - **Request capacity** is an **explicit request** for resources
    - The scheduler can use the **request capacity** to make decisions on where to put the pod on
    - You can see it as a **minimum amount of resources the pod needs**
  - **Resource limit** is a limit imposed to the container
    - The container will not be able to utilize more resources than specified

Resource Quotas

- Example of resource quotas:
  - You run a **deployment** with a **pod** with a **CPU resource** request of **200m**
  - 200m = 200 millicpu (or also 200 millicores)
  - 200m = 0.2, which is 20% of a CPU core of the running node
    - If the node has 2 cores, it's still 20% of a single core
  - You can also put a limit, e.g. on 400m
  - Memory quotas are defined by MiB or GiB

Resource Quotas

- If a capacity quota (e.g. mem / cpu) has been specified by the administrator, then each pod needs to specify capacity quota during creation
  - The administrator can specify default request values for pods that don't specify any values for capacity
  - The same is valid for limit quotas
- If a resource is requested more than the allowed capacity, the server API will give an error 403 FORBIDDEN - and kubectl will show an error

Resource Quotas

- The administrator can set the following resource limits within a namespace:

Resource	Description
requests.cpu	The sum of **CPU requests** of all pods cannot exceed this value
requests.mem	The sum of **MEM requests** of all pods cannot exceed this value
requests.storage	The sum of **storage requests** of all persistent volume claims cannot exceed this value
limits.cpu	The sum of **CPU limits** of all pods cannot exceed this value
limits.memory	The sum of **MEM limits** of all pods cannot exceed this value

Resource Quotas

- The administrator can set the following object limits:

Resource	Description
configmaps	total number of **configmaps** that can exist in a namespace
persistentvolumeclaims	total number of **persistent volume claims** that can exist in a namespace
pods	total number of **pods** that can exist in a namespace
replicationcontrollers	total number of **replicationcontrollers** that can exist in a namespace
resourcequotas	total number of **resource quotas** that can exist in a namespace
services	total number of **services** that can exist in a namespace
services.loadbalancer	total number of **load balancers** that can exist in a namespace
services.nodeports	total number of **nodeports** that can exist in a namespace
secrets	total number of secrets that can exist in a namespace

Namespaces

Namespaces

- Namespaces allow you to create **virtual clusters** within the same physical cluster
- Namespaces **logically separates** your cluster
- The standard namespace is called "**default**" and that's where all resources are launched in by default
  - There is also namespace for kubernetes specific resources, called **kube-system**
- Namespaces are intended when you have **multiple teams** / **projects** using the Kubernetes cluster

Namespaces

- The name of resources need to be unique within a namespace, but not across namespaces
  - e.g. you can have the deployment “helloworld” multiple times in different namespaces, but not twice in one namespace
- You can divide resources of a Kubernetes cluster using namespaces
  - You can limit resources on a per namespace basis
  - e.g. the marketing team can only use a maximum of 10 GiB of memory, 2 loadbalancers, 2 CPU cores

Namespaces

- First you need to create a new namespace

```
$ kubectl create namespace myspace
```

- You can list namespaces:

```
$ kubectl get namespaces
NAME          LABELS    STATUS
default          <none>    Active
kube-system   <none>    Active
myspace        <none>    Active
```

- You can set a default namespace to launch resources in:

```
$ export CONTEXT=$(kubectl config view | awk '/current-context/ {print $2}')
$ kubectl config set-context $CONTEXT —namespace=myspace
```

Namespaces

- You can then create resource limits within that namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
  namespace: myspace
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Namespaces

- You can also create object limits:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
  namespace: myspace
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
```

- Note: All those quota limits are in absolute numbers

Demo

Namespace quotas

User Management

User Management

- There are **2 types** of users you can create
  - A **normal user**, which is used to access the user externally
    - e.g. through kubectl
    - This user is **not managed using objects**
  - A **Service user**, which is **managed by an object in Kubernetes**
    - This type of user is used to **authenticate within** the cluster
    - e.g. from inside a pod, or from a kubelet
    - These credentials are managed like **Secrets**

User Management

- There are multiple **authentication strategies** for normal users:
  - Client Certificates
  - Bearer Tokens
  - Authentication Proxy
  - HTTP Basic Authentication
  - OpenID
  - Webhooks

User Management

- Service Users are using **Service Account Tokens**
- They are stored as **credentials using Secrets**
  - Those Secrets are also mounted in pods to allow communication between the services
- Service Users are **specific to a namespace**
- They are created automatically by the API or manually using **objects**
- Any API call **not authenticated** is considered as an **anonymous** user

User Management

- Independently from the authentication mechanism, normal users have the following **attributes**:
  - a Username (e.g. user123 or user@email.com)
  - a UID
  - Groups
  - Extra fields to store extra information

User Management

- After a normal users authenticates, it will have access to everything
- To **limit** access, you need to configure **authorization**
- There are again multiple offerings to choose from:
  - AlwaysAllow / AlwaysDeny
  - ABAC (Attribute-Based Access Control)
  - RBAC (Role Based Access Control)
  - Webhook (authorization by remote service)

User Management

- **Authorization** is still **work in progress**
- The ABAC needs to be configured **manually**
- RBAC uses the rbac.authorization.k8s.io **API** group
  - This allows admins to **dynamically** configure permissions **through the API**
- In Kubernetes 1.3 RBAC is still **alpha** and even considered **experimental**
  - RBAC is promising and will become **stable**
  - For the current state about ABAC/RBAC, see http://kubernetes.io/docs/admin/authorization/

Demo

Adding users

RBAC

Authorization

- After authentication, **authorization** controls what the user can do, where does the user have access to
- The access controls are implemented on an **API level** (kube-apiserver)
- When an API request comes in (e.g. when you enter **kubectl get nodes**), it will be **checked** to see whether you have access to execute this command

Authorization

- There are multiple **authorization** module available:
  - **Node**: a special purpose authorization mode that authorizes API requests made by **kubelets**
  - **ABAC**: attribute-based access control
    - Access rights are controlled by policies that combine attributes
    - e.g. user “alice" can do anything in namespace “marketing”
    - ABAC does not allow very granular permission control

Authorization

- **RBAC**: role based access control
  - Regulates access using **roles**
  - Allows admins to dynamically configure permission policies
  - This is what I'll use in the demo
- **Webhook**: sends authorization request to an external REST interface
  - Interesting option if you want to write your own **authorization server**
  - You can parse the incoming **payload** (which is JSON) and reply with access granted or access denied

RBAC

- To enable an **authorization mode**, you need to pass --authorization-mode= to the API server at startup
  - For example, to enable RBAC, you pass —authorization-mode=RBAC
- Most tools now provision a cluster with **RBAC enabled by default** (like kops and kubeadm)
  - For minikube, it'll become default at some point (see https://github.com/kubernetes/minikube/issues/1722)

RBAC

- If you’re using **minikube**, you can add a parameter when starting minikube:

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

RBAC

- You can **add RBAC resources** with *kubectl* to grant permissions
  - You first describe them in **yaml** format, then apply them to the cluster
- First you define **a role**, then you can **assign users/groups** to that role
- You can create roles **limited to a namespace** or you can create roles where the **access applies to all namespaces**
  - **Role** (single namespace) and **ClusterRole** (cluster-wide)
  - **RoleBinding** (single namespace) and **ClusterRoleBinding** (cluster-wide)

RBAC Role

- RBAC Role granting read access to pods and secrets within default namespace

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: [“pods”, “secrets”]
  verbs: ["get", "watch", "list"]
```

RBAC Role

- Next step is to assign users to the newly created role

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: bob
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

RBAC Role

- If you rather want to create a role that spans all namespaces, you can use ClusterRole

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: pod-reader-clusterwide
rules:
- apiGroups: [""]
  resources: [“pods”, “secrets”]
  verbs: ["get", "watch", "list"]
```

RBAC Role

- If you need to assign a user to a cluster-wide role, you need to use ClusterRoleBinding

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader-clusterwide
  apiGroup: rbac.authorization.k8s.io
```

Authorization demo

RBAC demo

Networking

Networking

- The approach to networking is quite different than in a default Docker setup
- In this course I already covered:
  - **Container to container** communication within a pod
    - Through **localhost** and the **port number**
  - **Pod-To-Service** communication
    - Using **NodePort**, using **DNS**
  - **External-To-Service**
    - Using **LoadBalancer**, **NodePort**

Networking

- In Kubernetes, the pod itself should always be routable
- This is **Pod-to-Pod** communications
- Kubernetes assumes that pods should be able to communicate to other pods, regardless of which node they are running
  - Every pod has its **own IP address**
  - Pods on different nodes need to be able to communicate to each other using those IP addresses
    - This is implemented differently depending on your networking setup

Networking

- On AWS: **kubenet networking** (kops default)
  - Every pod can get an IP that is **routable** using the AWS Virtual Private Network (VPC)
  - The kubernetes master allocates a /24 subnet to each node (254 IP addresses)
  - This subnet is added to the VPCs route table
  - There is a limit of **50 entries**, which means you can't have more than 50 nodes in a single AWS cluster
    - Although, AWS can raise this limit to 100, but it might have a **performance impact**

Networking

- Not every cloud provider has VPC-technology (although GCE, Azure does as well)
- There are **alternatives** available
  - **Container Network Interface (CNI)**
    - Software that provides libraries / plugins for network interfaces within containers
    - Popular solutions are **Calico**, **Weave** (standalone or with CNI)
  - **An Overlay Network**
    - **Flannel** is an easy and popular way

Flannel

node 1
Pod
container 1
container 2
veth0
10.3.1.2
docker0
10.3.1.1
flannel0
10.3.x.x
host IP
192.168.0.2
MAC addr
src: 192.168.0.2
dst: 192.168.0.3
UDP
Encapsulated packet with
src: 10.3.1.2
dst: 10.3.2.2
Packet
Flannel acts as network gateway between nodes
node 2
Pod
container 1
container 2
veth0
10.3.2.2
docker0
10.3.2.1
flannel0
10.3.x.x
host IP
192.168.0.3

Node Maintenance

Node Maintenance

- It is the **Node Controller** that is responsible for managing the Node objects
  - It assigns **IP space** to the node when a new node is launched
  - It keeps the **node list** up to date with the available machines
  - The node controller is also monitoring the **health of the node**
    - If a node is **unhealthy it gets deleted**
    - Pods running on the unhealthy node will then get **rescheduled**

Node Maintenance

- When adding a new node, the **kubelet** will attempt to register itself
- This is called **self-registration** and is the default behavior
- It allows you to **easily add more nodes** to the cluster without making API changes yourself
- A new node object is **automatically** created with:
  - The metadata (with a name: IP or hostname)
  - Labels (e.g. cloud region / availability zone / instance size)
- A node also has a **node condition** (e.g. Ready, OutOfDisk)

Node Maintenance

- When you want to **decommission** a node, you want to do it gracefully
  - You drain a node before you shut it down or take it out of the cluster
- To drain a node, you can use the following command:

```
$ kubectl drain nodename --grace-period=600
```

- If the node runs pods not managed by a controller, but is just a single pod:

```
$ kubectl drain nodename --force
```

Demo

Drain a node

High Availability

High Availability

- If you're going to run your cluster in production, you're going to want to have all your master services in a **high availability (HA)** setup
- The setup looks like this:
  - **Clustering etcd**: at least run 3 etcd nodes
  - **Replicated API servers** with a LoadBalancer
  - Running multiple instances of the **scheduler** and the **controllers**
    - Only one of them will be the leader, the other ones are on stand-by

Architecture overview - HA

No High Availability

3 nodes

5 nodes

Architecture overview - HA

High Availability

- A cluster like minikube doesn't need HA - it's only a one node cluster
- If you're going to use a production cluster on AWS, **kops** can do the heavy lifting for you
- If you're running on an other cloud platform, have a look at the **kube deployment tools** for that platform
    - **kubeadm** is a tool that is in alpha that can set up a cluster for you
- If you're on a platform without any tooling, have a look at http://kubernetes.io/docs/admin/high-availability/ to implement it yourself
- In the next demo I'll show you how to modify the **kops setup** to run multiple master nodes

Demo

HA setup

Federation

Federation

- Federation allows you to manage multiple Kubernetes clusters
  - They can be in different Regions at the same Cloud Provider
  - It can be an on-site cluster + a cluster in the cloud (hybrid)
  - It can be a cluster than spans multiple Cloud Providers

Federation

- The Setup
  - It requires running the Federation plane:
    - etcd cluster
    - federation-apiserver
    - federation-controller-manager
  - You can run these binaries as pods on an existing cluster

TLS on AWS ELB

TLS on AWS ELB

- You can setup **cloud specific features** (like TLS termination) on AWS LoadBalancers that you create in Kubernetes using services of type LoadBalancer

- You can do this using **annotations**:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:xx-xxxx-x:xxxxxxxxx:xxxxxxx/xxxxx-xxxx-xxxx-xxxx-xxxxxxxxx
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
```

- In this lecture I'll go over the **possible annotations** for the AWS Elastic Load Balancer (ELB)

TLS on AWS ELB

Annotation	Description
service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval	Used to enable access logs on the load balancer
service.beta.kubernetes.io/aws-load-balancer-access-log-enabled	
service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name	
service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix	
service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags	Add tags
service.beta.kubernetes.io/aws-load-balancer-backend-protocol	Backend protocol to use

TLS on AWS ELB

Annotation	Description
service.beta.kubernetes.io/aws-load-balancer-ssl-cert	Certificate ARN
service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled	Connection draining
service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout	Timeout when backend node stops during scaling
service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout	Connection idle timeout
service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled	Cross-AZ loadbalancing
service.beta.kubernetes.io/aws-load-balancer-extra-security-groups	Extra security groups

TLS on AWS ELB

Annotation	Description
service.beta.kubernetes.io/aws-load-balancer-internal	Set ELB to internal loadbalancer
service.beta.kubernetes.io/aws-load-balancer-proxy-protocol	Enable proxy protocol
service.beta.kubernetes.io/aws-load-balancer-ssl-ports	what listeners to enable HTTPS on (default to all)

TLS on AWS ELB

demo

Packaging and Deploying

Helm

Helm

- Helm the best way to find, share and use software built for Kubernetes
  (definition from https://helm.sh/)

- Helm is a **package manager** for Kubernetes

- It helps you to manage Kubernetes **applications**

- Helm is maintained by the **CNCF** - **The Cloud Native Computing Foundation** (together with Kubernetes, fluentd, linkerd, and others)

  - It is now maintained in collaboration with **Microsoft**, **Google**, **Bitnami** and the **helm contributor community**

Helm

- To start using helm, you first need to download the **helm client**
- You need to run “helm init” to **initialize helm** on the Kubernetes cluster
  - This will install **Tiller**
  - If you have **RBAC installed** (recent clusters have it enabled now by default), you’ll also need add a **ServiceAccount and RBAC rules**
- After this, helm is ready for use, and you can **start installing charts**

Helm - charts

- Helm uses a packaging format called **charts**
  - A **chart** is a **collection** of **files** that **describe** a set of Kubernetes **resources**
  - A single chart can **deploy an app**, a piece of software, or a database
  - It can have dependencies, e.g. to install wordpress chart, you need a mysql chart
  - You can write **your own chart** to deploy your application on Kubernetes using helm

Helm - charts

- Charts use **templates** that are typically developed by a package maintainer
- They will generate **yaml** files that Kubernetes understands
- You can think of templates as dynamic yaml files, which can contain logic and variables

Helm - charts

- This is an example of a **template within a chart**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favoriteDrink }}
```

- The favoriteDrink value can then be overridden by the user when running helm install

Helm - Common commands

Command	Description
helm init helm reset	Install tiller on the cluster Remove tiller from the cluster
helm install	Install a helm chart
helm search	search for a chart
helm list	list releases (installed charts)
helm upgrade	upgrade a release
helm rollback	rollback a release to the previous version

Helm charts

demo

Helm

Create your own helm charts

Helm Charts

- You can **create helm charts** to **deploy your own apps**
- It's the **recommended** way to deploy your applications on Kubernetes
  - Packaging the app, allows you **deploy the app in 1 command** (instead of using kubectl create / apply)
  - Helm allows for **upgrades** and **rollbacks**
  - Your helm chart is **version controlled**

Helm Charts

- To create the files necessary for a new chart, you can enter the command:

helm create *mychart*

Demo

helm create NAME

Demo

node-app-demo helm chart

Demo

Create a Chart repository using AWS S3

Demo

Build and Deploy a Chart using Jenkins

Serverless

Functions in Kubernetes

What is Serverless

- Public Cloud providers often provide Serverless capabilities in which you can **deploy functions**, rather than instances or containers
  - Azure Functions
  - AWS Lambda
  - Google Cloud Functions
- With these products, you **don't need to manage the underlying infrastructure**
- The functions are also **not "always-on"** unlike containers and instances, which can greatly the reduce cost of serverless if the function doesn't need to be executed a lot

What is Serverless

- Serverless in public cloud can **reduce the complexity, operational costs, and engineering time to get code running**
  - You don't need to manage a Windows/Linux distribution
  - You don't need to build containers
  - You only pay for the time your function is running
  - A developer can "just push" the code and does not to worry about many operational aspects
    - Although "cold-starts", the time for a function to start after it has not been invoked for some time, can be an operational issue that needs to be taken care of

What is Serverless

- This is an example of a AWS Lambda Function:

```
exports.handler = function(event, context) {
 context.succeed(“Hello, World!”);
};
```

- You’d still need to setup when the code is being executed
- For example in AWS you would use the API Gateway, to setup a URL that will invoke this function when visited

Serverless in Kubernetes

- Rather than using containers to start applications on Kubernetes, you can also use **Functions**
- Currently, the **most popular projects** enabling functions are:
  - OpenFaas
  - Kubeless
  - Fission
  - OpenWhisk
- You can install and use any of the projects to let developers launch functions on your Kubernetes cluster
- As an **administrator**, you'll **still need to manage the underlying infrastructure**, but from a **developer** standpoint, he/she will be able to **quickly and easily deploy functions on Kubernetes**

Serverless in Kubernetes

- All these projects are **pretty new** (as of September 2018), so their feature set will still drastically change
  - If you're looking in adopting a serverless technology for your Kubernetes cluster, it's best to **compare the features and the maturity of multiple software products, and make your own decision**
- In this course, I'll demo Kubeless, which is easy to setup and use

Kubeless

Kubeless

- Kubeless is a Kubernetes-native framework (source: https://github.com/kubeless/kubeless/)
    - It leverages the Kubernetes resources to provide auto-scaling, API routing, monitoring, etc
- It uses **Custom Resource Definitions** to be able to create functions
- It's **open source** and **non-affiliated** to any commercial organization
- It has a UI available for developers to deploy functions

Kubeless

- With kubeless you deploy a function in your preferred language
- Currently, the **following runtimes are supported**:
    - Python
    - NodeJS
    - Ruby
    - PHP
    - .NET
    - Golang
    - Others

Kubeless

- Once you deployed your function, you'll need to determine how it'll be **triggered**
- Currently, the **following function are supported**:
  - **HTTP functions**
    - HTTP functions gets executed when an HTTP endpoint is triggered
    - You write a function and return the text/HTML that needs to be displayed in the browser
  - Scheduled function

Kubeless

- Once you deployed your function, you need to determine how it'll be **triggered**
- Currently, the **following function are supported**:
  - PubSub (Kafka or NATS)
    - Triggers a function when data is available in Kafka / NATS
  - AWS Kinesis
    - Triggers based on data in AWS Kinesis (similar to Kafka)

Kubeless

demo

Kubeless - PubSub

demo

Kubeless UI

demo

Microservices

Microservices

- Kubernetes makes it **easy to deploy a lot of diverse applications**
- Those applications can be **monoliths** that don't have anything to do with each other, or **microservices**, small services that make up one application
- The **microservices architecture** is **increasingly popular**
- This approach allows developers to split up the application in **multiple independent parts**
- Having to manage microservices can put an **operational strain** on the engineering team

Microservices - Monoliths

Microservices

Microservices

Kubernetes Cluster

Microservices

Kubernetes Cluster

Microservices

Kubernetes Cluster

Microservices

Kubernetes Cluster

Istio

Kubernetes Cluster
Pod
App 1
Envoy proxy
Pod
App 2
Envoy proxy
Config data
Policy checks, Telemetry
TLS Certs
Pilot
Mixer
Citadel
Control Plane API

Istio Install

demo

Istio enabled apps

demo

hello world app

Traffic Routing

demo

hello world app - v2

Kubernetes Cluster

Canary deployments

demo

hello world app - v2

Kubernetes Cluster

Retries

demo

hello world app - v3

Kubernetes Cluster

Security

Mutual TLS

Security

- The **goals of Istio security** are (source: https://istio.io/docs/concepts/security/#authentication)
  - **Security by default**: no changes needed for application code and infrastructure
  - **Defense in depth**: integrate with existing security systems to provide multiple layers of defense
  - **Zero-trust network**: build security solutions on untrusted networks

Security

- Istio provides **two types** of authentication:
  - 1) **Transport authentication** (service to service authentication) using Mutual TLS
  - 2) **Origin authentication** (end-user authentication)
    - Verifying the end-user using a JSON Web Token (JWT)

Security

- Mutual TLS in Istio:
  - Can be turned on **without having to change the code of applications** (because of the sidecar deployment)
  - It provides each service with a **strong identity**
    - Attacks like impersonation by rerouting DNS records will fail, because a fake application can't prove its identity using the certificate mechanism
  - **Secures (encrypts)** service-to-service and end-user-to-service communication
  - Provides key and certificate management to **manage generation, distribution, and rotation**

hello world app - mutual TLS

Mutual TLS

demo

Security

RBAC with mutual TLS

RBAC

- Now that we're using **Mutual TLS**, we have **strong identities**
- Based on those **identities**, we can start to doing **Role Based Access Control** (RBAC)
- RBAC allows us to limit access **between our services,** and from **end-user to services**
- Istio is able to verify the identity of a service by **checking the identity of the x.509 certificate (which comes with enabling mutual TLS)**
  - For example: service A can be contacted by B, but not by C
  - Good to know: The identities capability in istio is built using the **SPIFFE standard** (Secure Production Identity Framework For Everyone, another CNCF project)

RBAC

- **RBAC in istio** (source: https://istio.io/docs/concepts/security/)
  - Can provide **service-to-service** and **end-user-to-service** authorization
  - Supports **conditions** and **role-binding**
    - You can bind to **ServiceAccounts** (which can be linked to pods)
    - End-user-to-service can for example let you create **condition on being authenticated using JWT**
  - It has high performance, as its natively enforced on Envoy (the sidecar proxy)

RBAC

- RBAC is **not enabled by default**, so we have to enable it
- We can enable it globally, or on a namespace basis
  - For example, in the demo, we'll only enable it for the "default" namespace
- We can then c s the rules and a **ServiceRoleB** subject (for example a Kubernetes **S**

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: RbacConfig
metadata:
  name: default
spec:
  mode: 'ON_WITH_INCLUSION'
  inclusion:
    namespaces: ["default"]
```

hello world app

RBAC in istio

demo

Security

End-user authentication

End-user authentication

- Up until now we've talked about **service-to-service** communication
  - I showed you how to enable **mutual TLS** for service-to-service authentication
  - After having **strong identities** using the x.509 certificates that mutual TLS provides, I showed you how to use **role based access control** (RBAC)
- In this lecture I will explain **end-user to service authentication**
  - Istio currently supports **JWT tokens** to authenticate end-users

End-user authentication

- JWT stands for **JSON web tokens**
- it's an **open standard** for representing claims securely between two parties (see https://jwt.io/ for more information)
- In our implementation, we'll receive **a JWT token** from an authentication server after logging in (still our hello world app)
  - The app will provide us with a token that is **signed with a key**
  - The data is not encrypted, but the **token contains a signature**, which can be **verified** to see **whether it was really created by the server**
  - Only the server has the (private) **key**, so we can't recreate or tamper with the token

End-user authentication

- This is an example of a token:
- **eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c**
- It consists of **3 parts**, divided by a dot (.)
  - The first part contains the **headers**
  - The second part contains the **payload**
  - The third part is the **signature based** on the headers+payload

End-user authentication

- You can use jwt.io to decode the token:
- The headers are in this case:
  - { "alg": “HS256", "typ": “JWT" }
- And the body:
  - { "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }

End-user authentication

- In webapps using authentication, the server can **issue a JWT token when the user is authenticated**
- In the JWT **payload**, data can be stored, like the **username**, **groups**, etc
- This can then used later by the app, when the users sends new requests
  - If the **signature in the token** is valid, then the JWT is valid, and the data within the token can be used
  - This can also be used as an **alternative approach to server sessions** (in this case (some) session data is stored local at the client)

End-user authentication

- Using **microservices**, every app would have to be **separately configured**
  - **Every service** would need to **validate the token**
    - Once validated the service would need to check whether the user has **access** to this service (authorization)
- With istio, this can be **taken away from the app code** and **managed centrally**
- You can **configure the jwt token signature/properties you expect in istio**, and create policies to allow/disallow access to a service
  - For example: the “hello” app can only be accessed if the user is authenticated
    - The **sidecar** will verify the **validity** of the signature, to make sure the token is valid

hello world app

Kubernetes Cluster

end-user authentication

demo

Egress traffic

demo

Egress traffic

Distributed tracing

demo

Distributed tracing

Kubernetes Cluster

Metrics with Grafana

demo

Congratulations

Bonus Lecture

My Other Courses:
http://www.udemy.com/user/ward-viaene

COUPON CODE: BONUS_LECTURE

Advanced Kubernetes Usage

Advanced Kubernetes Usage	Technologies
Logging	ElasticSearch (as StatefulSet), fluentd, Kibana, LogTrail
Deploying on Kubernetes	Spinnaker
Microservices on Kubernetes	Linkerd
Federation	kubefed
Monitoring	Prometheus

https://www.udemy.com/learn-devops-advanced-kubernetes-usage/

COUPON CODE: **BONUS_DEVOPS**

Bonus Lecture

Category	Course Title	COUPON
Learn DevOps	Continuously Deliver Better Software	BONUS_DEVOPS
Learn DevOps	Scale applications on-premise and in the cloud	BONUS_DEVOPS
Learn DevOps	Infrastructure Automation with terraform	BONUS_DEVOPS
Learn DevOps	**Advanced Kubernetes Usage**	**BONUS_DEVOPS**
Learn Big Data	The Hadoop Ecosystem Masterclass	BONUS_DEVOPS