



Kubernetes in 4 Hours

Sander van Vugt

About your instructor

- This course is presented by Sander van Vugt
 - mail@sandervanvugt.com
 - www.sandervanvugt.com
- Course resources are available at
<https://github.com/sandervanvugt/kubernetes>

Agenda

- Understanding Kubernetes
- Kubernetes Installation and Configuration
- Using Kubernetes to Manage Containers

Expectations

- This class is for people new to Kubernetes
- I'll teach you how to set up a test environment based on Minikube
- Don't expect much information about advanced topics

Lab instructions

- This course contains lab instructions which you can follow along or run later at your own convenience
- To get the same experience, use the following recommended minimal settings
 - Linux host (I'm using Fedora 31) with hardware virtualization enabled
 - At least 8GB of RAM (more is always better)
 - At least 40 GB of available disk space (more is always better)
- While running labs in this course, do NOT run another hypervisor on the host at the same time

Question Handling Policy

- If you want me to answer your question, post in Q&A please. I will answer it if it meets the requirements, but you may have to wait a bit
- Questions about personal issues "when I do this, I get error message xyz" only in group chat please
- If you just missed something, please watch the recording. I cannot repeat topics I just explained as that wastes valuable time
- Questions about topics that you should have known in group chat only please
- Off-topic questions are welcome in the Q&A at the end of this course

Poll Question 1

- How would you rate your own Linux knowledge and experience?
 - 0
 - 1
 - 2
 - 3
 - 4
 - 5

Poll question 2

- How would you rate your own knowledge about containers
 - 0
 - 1
 - 2
 - 3
 - 4
 - 5

Poll question 3

- How would you rate your own Kubernetes knowledge and experience?
 - 0
 - 1
 - 2
 - 3
 - 4
 - 5



Kubernetes in 4 Hours

What is Kubernetes?

What is Kubernetes?

- Kubernetes is rapidly evolving open-source software, written in Go, for automating deployment, scaling, and management (orchestration) of containerized applications
 - See kubernetes.io for more details
 - A new major release every 3 months!
- It is about running multiple connected containers across different hosts, where continuity of service is guaranteed
- The solution is based on technology that Google has been using for many years in their datacenters
 - Google Borg inspired the development of Kubernetes
 - Borg has been used for over 15 years for hosting internal Google applications

Kubernetes Orchestration tasks

- Schedule containers to run on specific hosts using Pods
- Join hosts that are running containers in an etcd cluster
- Take care of scalability
- Make storage available
- Expose containers using a service

Other container management solutions

- Docker Swarm
- Rancher
- Red Hat OpenShift
 - Integrated Kubernetes and adds devops workflow services on top of it

What are Containers?

- Containers provide a way to package, ship and run applications
- Docker is a leading solution in containers
 - easily build containers
 - share container images in a simple way by using Docker registries
- Docker Inc offers 2 methods to manage containers in the datacenter
 - Docker swarm: currently part of Mirantis
 - Kubernetes
- Notice that Docker isn't the only container solution around, and even if Kubernetes now focusses on Docker, other container platforms will be integrated also

Container needs in the Datacenter

- A methodology to build, test and verify container images
- A cluster of hosts to run the containers
- Monitoring and self-healing of containers
- A solution for updates and rollbacks
- A flexible network that can self-extend if that is needed

About the Kubernetes Host Platform

Kubernetes can be offered through different host platforms

- As a hosted service in public cloud
- As a set of processes on a Linux host
- As a set of containers running on a Linux host
- Integrated in a minimized container OS as provided by CoreOS or Atomic
- As an all-in-one solution, running on Minikube

Understanding CNCF

- Cloud Native Computing Foundation (CNCF) is a governing body that solves issues faced by any cloud native application (so not just Kubernetes)
- Google donated Kubernetes to the Cloud Native Computing Foundation, which is a foundation in Linux Foundation
- CNCF owns the copyright of Kubernetes
- Kubernetes itself uses an Apache license
- Developers need to sign a Contributor License Agreement with CNCF



Kubernetes in 4 Hours

Installing a Kubernetes Minikube Test Cluster

Minikube Overview

- Minikube offers a complete test environment that runs on Linux, OS-X or Windows
- In this course we'll focus on Minikube as it is easy to setup and has no further dependencies
- In all cases, you'll need to have the **kubectl** client on your management platform

Installing Minikube

- A scripted installation is provided for Fedora Workstation only
- Install Fedora Workstation with at least 4 GB RAM and 20 GB disk space (8 GB and 40GB recommended)
- Use **git clone <https://github.com/sandervanvugt/kube4h>**
- From there, use the **kube-setup** script and follow instructions

Testing Minikube

- **kubectl cluster-info**: shows cluster information
- **kubectl get nodes**: shows nodes currently available
- **minikube ssh**: logs in to the Minikube host
- **docker ps**: shows all Docker processes on the MK host
- **ps aux | grep localkube**: shows the localkube process on MK host

Managing minikube

The **minikube** command has different options

- **dashboard**: opens the K8s dashboard in the local browser
- **delete**: deletes a cluster
- **ip**: show the currently used IP address
- **service *nn***: gets the kubernetes URL for the specific service
- **start**: starts a cluster
- **stop**: stops a cluster
- **status**: gives current status
- **version**: shows current version
- **ssh**: connect to the minikube cluster

Importing Images into Kubernetes

- To start using Kubernetes, you'll need to pull a container image
- You can create your container image and push it to a public or private registry before referring to it in a Kubernetes pod
- Or you can use public container registries to pull images from
 - Use Docker Hub or any public cloud container registry
 - Docker Hub images make sense in a small private deployment
 - Public cloud container registries make sense if you're running Kubernetes from a Public cloud

Running Your First Application

- From **minikube dashboard**, click +CREATE in the upper right corner
- Specify **httpd** as the container image as well as the container name
- This will pull the Docker container and run it in the minikube environment



Kubernetes in 4 Hours

Understanding Kubernetes Resource Types

Understanding Main Kubernetes Resource Types

- Resource types are defined in the Kubernetes APIs
- Pods: the basic unit in Kubernetes, represents a set of containers that share common resources such as an IP address and storage volumes
- Deployments: standard entity that is rolled out with Kubernetes
- Services: make deployments accessible from the outside by providing a single IP/port combination. Services by default provide access to pods in round-robin fashion using a load balancer
- Persistent Volumes: persistent (networked) storage that can be mounted within a container by using a Persistent Volume Claim

Understanding the Pod

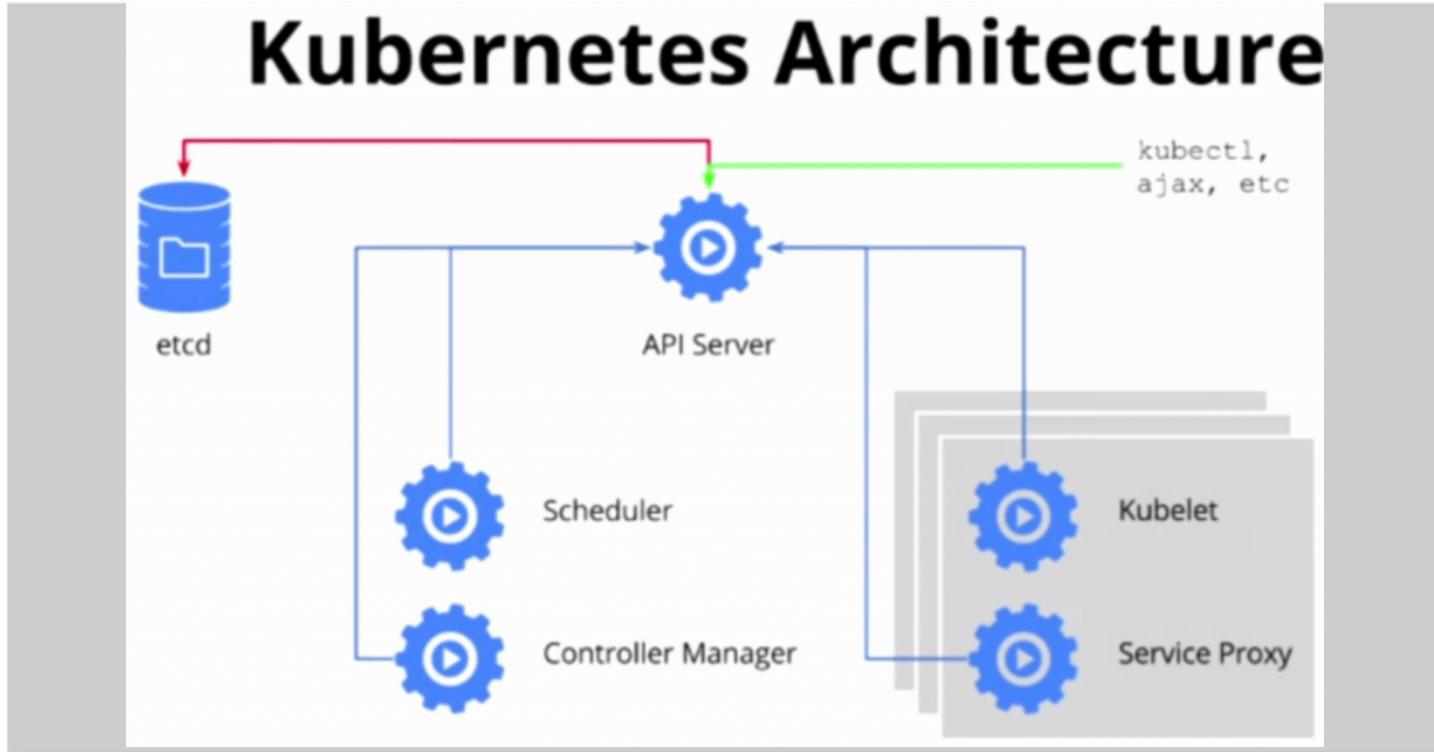
- Kubernetes manages Pods, not containers
- A Pod is using namespaces to ensure that resources in the Pod can communicate easily
 - In a pre-container environment, containers in a Pod would be implemented as applications on the same computer
- Containers can be put together in a Pod, together with Pod-specific storage
- Pods are exposed through an IP address, not the individual containers inside a Pod
- See here for more information about pods:
 - <https://kubernetes.io/docs/concepts/workloads/pods/pod/>



Kubernetes in 4 Hours

Understanding Kubernetes Architecture

Kubernetes Architecture



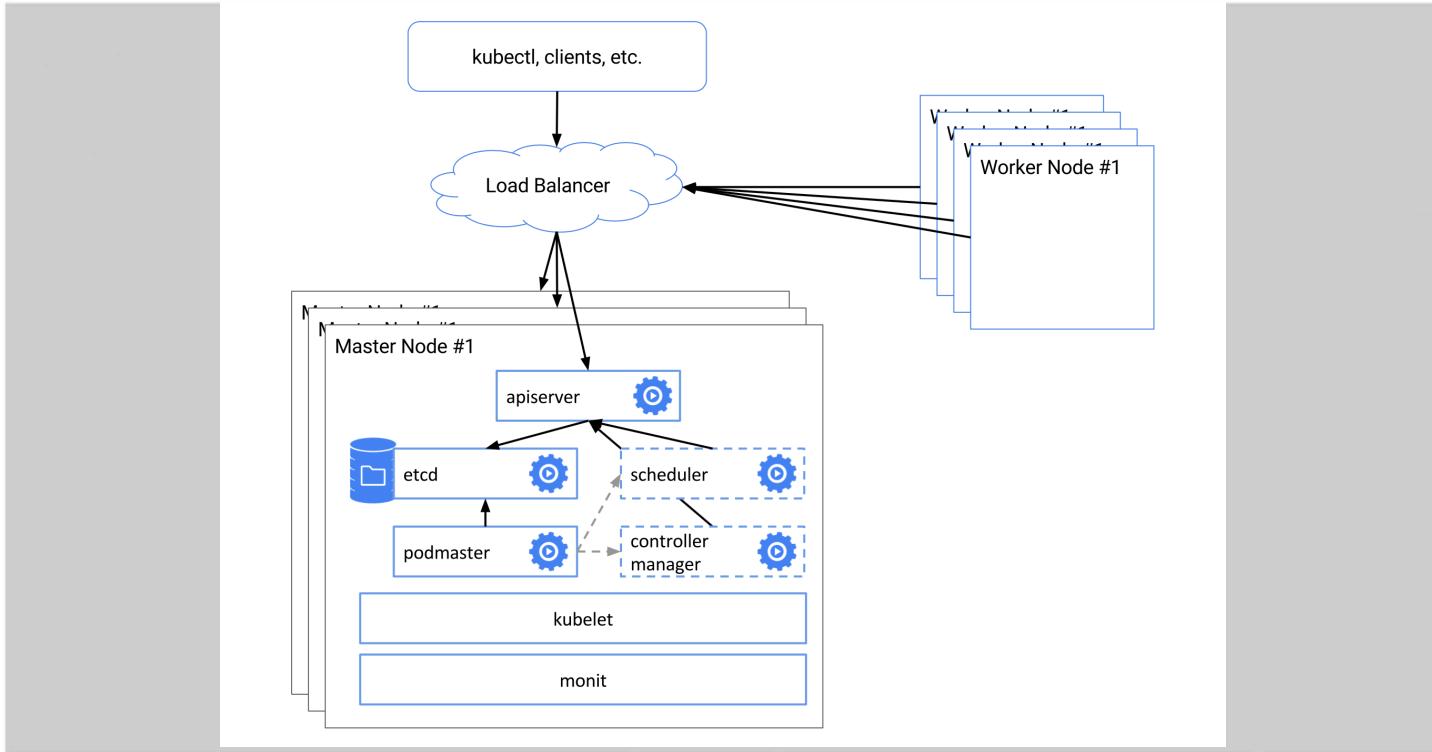
Base Architecture

- Typically, Kubernetes has a master node and multiple worker nodes
- Minikube allows you to run all on one node
 - Great for learning Kubernetes, not for production
- The base Kubernetes components are
 - API server
 - Scheduler
 - Controller Manager
 - Kubelet
 - Kube Proxy
 - etcd
- Each of these can run as a Linux process or as a container

Optional High Availability

- Master node typically is configured in a multi-master HA setup
- Schedulers and Controller managers can elect a leader
- API servers can use a load-balancer as a front end

Kubernetes HA Overview



Worker Nodes

- Worker nodes run the kubelet, kube proxy and container engine
- Kubelet receives instructions from the master to run containers on the underlying container engine
- Kubernetes can run different Container Engines, including Docker (most common) or Rkt
- The Container Runtime Interface (CRI) offers integration options for different container engines, including Windows containers and Rkt
 - <https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/>

Understanding etcd

- Etcd is the backing store for all cluster data and contains the current state of the cluster
- Etcd is a distributed key-value store
- Etcd elects a leader to ensure that data consistency between different nodes is guaranteed
- It can run on a single node, but that takes away the redundancy of the setup

Understanding Networking

- Networking is very similar to any IaaS networking
- The lowest unit in networking is a pod
- A pod is a group of containers that share the same IP address
 - As such, from the networking perspective it looks like a virtual machine or physical host
- The network needs to assign IP addresses to pods and take care of routing
- By using pods, Kubernetes requires 3 levels of networking to be taken care of

Kubernetes levels of networking

- Container-to-container: this is taken care of within the Pod, which allows containers to be coupled
 - Often POSIX IPCs is used for container-to-container traffic
- Pod-to-pod: this is taken care of by regular networking
 - Recommended: use physical networking
 - No NAT allowed between pods
 - Software defined overlay networking solutions may be used (Weave, Flannel and others)
 - Overlay networking solutions can easily be integrated with Kubernetes using the **kubectl** command
- External-to-pod: is taken care of by the services concept (covered later)

Understanding the CNI

- Kubernetes Pod-to-Pod networking is pluggable
- CNI is the Container Network Interface and implements the SDN
- It allows for writing plugins that configure container networking
 - See for an example on the next slide
- The plugins themselves are files that are written to /opt/cni/bin
- CNI network configurations are distributed using the Kubernetes cluster bootstrapping tool **kubeadm**
- See here for more information about Kubernetes networking
 - <https://kubernetes.io/docs/concepts/cluster-administration/networking/>

CNI Networking Example

```
{  
    "cniVersion": "0.2.0",  
    "name": "testnet",  
    "type": "bridge",  
    "bridge": "cni0",  
    "isGateway": true,  
    "ipMasq": true,  
    "ipam": {  
        "type": "host-local",  
        "subnet": "10.10.0.0/16",  
        "routes": [  
            { "dst": "0.0.0.0/0" }  
        ]  
    }  
}
```



Kubernetes in 4 Hours

Kubernetes Beyond Minikube

K8s in Public Cloud

- Kubernetes is commonly used in public cloud
 - AWS
 - Azure
 - Google Cloud
- Installation in private cloud (OpenStack and others) is also common
- Alternatively, Kubernetes can be installed in local datacenter

Kubernetes Deployment

- K8s can be deployed in many different ways
- According to the scale you need, different Kubernetes deployments can be used
 - single node: good for learning, not for production
 - single head node with multiple workers
 - multiple head nodes in HA with multiple workers
 - multiple head nodes in HA with etcd in HA and multiple workers
- Also, you need to decide where to run Kubernetes: public cloud, virtual machines, physical machines?

Installing a Physical Cluster

- **kubeadm** helps you building a physical cluster, running in your local datacenter
- The documentation is here:
<https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/>
- Rough procedure outline
 - Run **kubeadm init** on the head node
 - Create a network
 - `kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"`
 - Run **kubeadm join --token <token> <head-node-IP-address>** to join worker nodes



Kubernetes in 4 Hours

Accessing and Using the Cluster

Methods to access the cluster

- The **kubectl** command line utility provides convenient administrator access, allowing you to run many tasks against the cluster
- Direct API access allows developers to address the cluster using API calls from custom scripts
- The Kubernetes Console by default is available and accessible on port 30000 of the Kubernetes master

Using kubectl

- The **kubectl** command is the generic command that allows you to manage all aspects of pods and containers
 - It provides functionality that normally is provided through the **docker** command, but talks to pods instead of containers
- Use **kubectl create** to create pods
- Or **kubectl get ...** or one of the many other options to get information about pods
- **kubectl** works with subcommands, type just **kubectl** for a list of these, and use **kubectl <subcommand> --help** for more information about any of the subcommands
 - Notice the nice examples section on top of the --help output

Connecting to the API

- The API server exposes its functionality through REST
- Assuming that minikube is running, connect to it by running a local proxy first (on your workstation for instance), after which you can connect using curl
 - `kubectl proxy --port=8001&`
 - `curl http://localhost:8001`
- This shows all of the available API paths and groups, providing access to all exposed functions
- Using a proxy allows you to use `curl` and send API calls directly
- Read the documentation for more info about the API groups
- Every API is built according to strict specifications and always contains specific elements

Connecting to the API: demo

- On the host that runs kubectl: **kubectl proxy --port=8001 &**
- **curl http://localhost:8001/version**
- **curl http://localhost:8001/api/v1/namespaces/default/pods->**
shows the pods
- **curl**
http://localhost:8001/api/v1/namespaces/default/pods/httpd/
shows direct API access to a pod
- **curl -XDELETE**
http://localhost:8001/api/v1/namespaces/default/pods/httpd
will delete the httpd pod



Kubernetes in 4 Hours

Managing Pods

Managing pods with kubectl

- Use **kubectl run** to run a *deployment* based on a default image
 - `kubectl run ghost --image=ghost:0.9`
- Use **kubectl** combined with instructions in a YAML file to do anything you'd like
- **kubectl create -f <name>.yaml**
- **kubectl get pods**
- (see sample code on next slide)
- **kubectl describe pods** shows all details about a pod, including information about containers running within
 - For instance, **kubectl describe pods newhttpd**
- **kubectl edit pod mypod** allows editing of live pods

Using kubectl in a declarative way

- The recommended way to work with kubectl, is by writing your manifest files and using **kubectl apply -f manifest.yaml** to the current objects in your cluster
- This declarative methodology is giving you much more control than the imperative methodology where you create all from the CLI
 - Get current state of an object: **kubectl get deployments nginx -o yaml**
 - Push settings from a new manifest: **kubectl replace -f nginx.yaml**
 - Apply settings from a manifest: **kubectl apply -f nginx.yaml**

Example yaml Input File

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  namespace: default
spec:
  containers:
    - name: busybox
      image: busybox
      command:
        - sleep
        - "3600"
    - name: nginx
      image: nginx
```

Understanding Namespaces

- When creating pods, namespaces are used
- Deployments/services that need to communicate to one another may be created in the same namespace, if no communication is needed, it's recommended to use separate namespaces
- A namespace is a strict isolation that occurs on Linux kernel level
 - Names need to be unique within a namespace, but the same name may exist in multiple namespaces
- Every **kubectl** request uses namespaces to ensure that resources are strictly separated and names don't have to be unique

Using Namespaces

- The following commands allow you to work with namespaces
 - **kubectl get ns**
 - **kubectl create ns demo**
 - **kubectl get ns/demo -o yaml**
 - **kubectl delete ns/demo**
- As seen in the previous slide, you can add namespaces when creating a pod, thus ensuring that a pod is available in a specific namespace only

Getting information about Pods

- The API exposes lots of useful information about pods
- Request this information using **kubectl** or API calls
- **kubectl logs <podname>**
- **kubectl exec -ti <podname> <command>**
- **kubectl attach**
- Tip: use **kubectl nnn -h** for usage information about the -h option



Kubernetes in 4 Hours

Working with Deployments

Understanding Replica Sets

- The pod is the most basic entity in Kubernetes
- To determine how many instances of a pod you want to run, you need replica sets
- Replica sets are using labels to determine how many pods are running
 - While defining the replica set, set the label using labels:
 - In the replicaset, use matchlabels to find the labels
 - From deployments, labels are automatically set

Understanding Deployments

- Deployments are used to create and automate replica sets
- Deployments also instruct the cluster how to create and scale applications
- The deployment controller will monitor instances of an application
- Deployments allow for the creation of multiple replica sets for rolling upgrades or rollbacks
- Request information about these different levels
 - **kubectl run nginx --image=nginx**
 - **kubectl get deployments**
 - **kubectl get rs**
 - **kubectl get pods**

Understanding Deployments versus Replica Sets

- Management should happen at the deployment level, not at the replica sets level
- This demo shows why
 - **kubectl get rs**
 - **kubectl delete rs <name>**
 - **kubectl get rs** it has been recreated automatically!
 - **kubectl get deployments**: because there are still deployments!
 - **kubectl delete deployment <name>** will finally remove it



Kubernetes in 4 Hours

Using Scaling and Rollback

Scaling deployments

- **kubectl scale deployment nginx --replicas=3**
 - Notice that scaling deployments may fail because of limited availability of resources
- **kubectl get deployments nginx -o json**
 - Notice that different output formats are available, use **-o yaml** if you prefer seeing it in yaml, or **-o json** for json
 - In the output look for the labels; a run label is automatically added
- **kubectl get pods -Lrun** filters on pods with the label running
 - The run label is used to determine if sufficient replicas are available
 - Remove one "run" label and you'll notice that a new pod will be created immediately
 - Use **kubectl label pods <name> run-** to remove the label run

Understanding Labels

- Labels are name tags used in Deployments that can be set on objects
- Replicasets use labels to monitor the availability of a sufficient amount of pods
- Services use labels to define network traffic
- Labels can also be set on nodes, and combines with nodeSelector to run specific pods on specific nodes

Using Labels

- **kubectl get pods --show-labels** will show labels
- **kubectl get pods -Lstatus** adds a column based on the label "status" and will show all pods that have a value set for that label
- **kubectl get pods -l status=ok** will show all pods that match a specific label
- **kubectl label pods <somepod> status=failing --overwrite** will reset the label on a pod that already has a label
- Notice that if a label is changed on a pod that is a member of a replicaset, changing the label will trigger the replicaset to start a new pod!
- When using **kubectl run ...** the label automatically is set to run, and the value to the name of the deployment



Kubernetes in 4 Hours

Accessing Services from Outside

Accessing Applications in a pod

- To access an application running in a pod, port-forward can be used
- Run the port-forward argument to **kubectl** as a background process, this will export the pod port locally
 - **kubectl port-forward deployment/nginx 8080:80 & where**
 - **curl localhost:8080**

Understanding Services

- Pods in a deployment don't provide a uniform IP address to access all of them
- To provide such a unique IP address, use services
- Services use the kube-proxy that runs on all cluster nodes and load balances traffic to the replicated pods
- Which pods are added to a service is normally determined by using a label selector
- Using labels makes services very flexible: you don't rely on the existence of a specific pod, but of a pod that carries a specific label, which makes it easy to replace pods

Creating Services

- When running a deployment, Pods have an internal network address in the 172.17.0.0/16 network
- This address is dynamically allocated and cannot be used to address the deployment as it addressed the pod
- That's why a service is used to provide one cluster IP address for all Pods, which allows you to access Pods in a consistent way
- The service connects to the Pods internal IP addresses by making them endpoints
- Different types of services can be used to determine how the service can be accessed

Understanding Service Types

- Different service types determine how services are accessed
 - **ClusterIP** is the default and provides internal access only
 - **NodePort** assigns a random port ID and exposes it on the nodes that run the service
 - **LoadBalancer** is available in public cloud. May be used in private cloud, if Kubernetes provides a plugin for that cloud type
 - **externalName** exposes the service using a name
- To access a cluster IP service locally (without exposing it), you may also use **kubectl proxy**
 - After starting kubectl proxy, the service is accessible on the host where **kubectl proxy** was started

Creating Services: demo 1

- Expose a deployment, using **kubectl expose deployment nginx --port=80**
- Use **kubectl get services** to verify that the pod has been exposed and look for the cluster IP address it uses
- Use **kubectl describe svc nginx** to check the endpoints
- Use **kubectl get endpoints**
- Use **minikube ssh** to get on the minikube node and from there, use **curl <cluster-ip>** to access the service
- Use **kubectl delete service nginx** to remove the service
- Notice that removing a service does NOT remove the pod, it just un-exposes it

Creating services: demo 2

- **kubectl get deployment**
- **kubectl expose deployment httpd --port=80 --type=NodePort**
- **kubectl get svc**
 - Note the random port that has been assigned and is listening at the Kubernetes cluster to provide access to the service
- **kubectl get svc httpd -o yaml**
 - Have a look at the label and the selector that are used in the YAML output
- **curl 192.168.99.100:<NODEPORT>** ## Note that the IP address is the IP of the minikube host!

Understanding Services and DNS

- With services exposing themselves on dynamic ports, resolving service names can be challenging
- As a solution, a DNS service is included by default in Kubernetes and this DNS service is updated every time a new service is added
- So DNS name lookup from within one pod to any exposed service happens automatically

Understanding Ingress

- Ingress offers service access at a specific URL
- This is useful to make sure services can be accessed in a predicted way, instead of using a random high port
- Ingress is using a proxy in the backend to provide access to the deployments
- Ingress controller must be deployed for your platform

Managing Ingress

- Ingress controllers can use a host definition. If defined, traffic addressed to that specific URL only will be redirected to what the Ingress controller defines
- If host is not set (as in nginx-in.yaml) it hits all



Kubernetes in 4 Hours

Working with Volumes

Understanding Volumes

- A pod typically is a couple of containers with one or more volumes attached
- Volumes can be mounted in a specific location in the container
- Different types of volumes are available, according to storage needs
- Volumes typically are created using YAML files while creating the VMs
- Volumes can be internal to a Pod, and as such are a part of the Pod specification
- Persistent Volumes are a different API object and have been added to share storage between Pods

Understanding Volume Types

- Many volume types are supported
 - emptyDir
 - azureDisk
 - cephfs
 - awsElasticBlockStore
 - fc: fibrechannel
 - gcePersistentDisk: Google cloud
 - iscsi
 - nfs
 - rbd
 - gitrepo
 - hostPath: connects to host in minikube environment
- And more: choose what fits your specific needs

Volume YAML File Example

```
apiVersion: v1
kind: Pod
metadata:
  name: morevol
spec:
  containers:
    - name: centos
      image: centos:7
      command:
        - sleep
        - "3600"
      volumeMounts:
        - mountPath: /centos
          name: test
      name: centos
  ...
  - name: centos
    image: centos:7
    command:
      - sleep
      - "3600"
    volumeMounts:
      - mountPath: /centos2
        name: test
    name: centos2
  volumes:
    - name: test
      emptyDir: {}
```

Managing Volumes (demo)

- Creating the Volume
 - **kubectl create -f volumes.yaml**
 - **kubectl get pods**
- After creation, test that you can use it
 - **kubectl exec -ti vol -c centos -- touch /test/myfile**
 - **kubectl exec -ti vol -c centos -- ls -l /test**
 - Note: try this command without --

Understanding Persistent Volumes

- A Persistent Volume is a storage abstraction that is used to store persistent data
 - Use **persistentVolume** to define it
 - Different types (NFS, iSCSI, CephFS and many more) are available
- Using claims with Persistent Volumes creates portable Kubernetes storage that allow you to use volumes, regardless of the specific storage provider
 - Use **persistentVolumeClaims**
 - The persistent volume claim talks to the available backend storage provider and dynamically uses volumes that are available on that storage type
- Kubernetes will use Persistent Volumes according to the availability of the requested volume accessModes and capacity

Understanding ConfigMap

- ConfigMaps can be used to separate dynamic data from static data in a Pod
- They can be used in three different ways:
 - Make variables available within a Pod
 - Provide command line arguments
 - Mount them on the location where the application expects to find a configuration file

Working with ConfigMaps

- Create a simple input file that contains some variables

```
cat variables
VAR1=Hello
VAR2=World
```
- Create the ConfigMap: **kubectl create cm variables --from-env-file=variables**
- Verify creation: **kubectl describe cm variables**
- Create a Pod (see yaml file on next slide): **kubectl create -f cm-test-pod1.yml**
- Check the variables that are available: **kubectl logs po/example**

Working with ConfigMaps - code

```
apiVersion: v1
kind: Pod
metadata:
  name: test1
spec:
  containers:
    restartPolicy: never
    - name: test1
      image: cirros
      command: ["/bin/sh", "-c", "env"]
      envFrom:
        - configMapRef:
            name: variables
```

Using ConfigMaps for Config Files

- ConfigMaps can also be used to provide complete configuration files
- In this, the Pod is using the ConfigMap as a mounted volume which will just contain the configuration file
- And the ConfigMap itself contains the configuration file
- Create the ConfigMap: **kubectl create cm nginx-cm --from-file nginx-custom-config.conf**
- Check the contents of the ConfigMap: **kubectl get configmap/nginx-cm -o yaml**
- Next, create the Pod: **kubectl create -f nginx-cm.yml**
- Check the config file: **kubectl exec -it nginx-cm /bin/bash**
- **cat /etc/nginx/conf.d/default.conf**

Sample file: nginx-custom-config.conf

```
server {  
    listen      8888;  
    server_name localhost;  
    location / {  
        root   /usr/share/nginx/html;  
        index  index.html index.htm;  
    }  
}
```

Sample file: nginx-cm.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-cm
labels:
  role: web
spec:
  containers:
  - name: nginx-cm
    image: nginx
    volumeMounts:
    - name: conf
      mountPath: /etc/nginx/conf.d
  volumes:
  - name: conf
    configMap:
      name: nginx-cm
      items:
      - key: nginx-custom-config.conf
        path: default.conf
```



Kubernetes in 3 Hours

Additional Features

Interesting Kubernetes Features not in this course

- Quota set resource limitations at a namespace level
- Secrets: like configMaps, to work with sensitive data in a way that the data is not readable
- Helm: the Kubernetes Package Manager, applications are packages in a chart and published in a repository as a tarball, allows to group all the different object types that make up an app into one package
- Custom Resource Definitions: the option to create your own objects in the API



Kubernetes in 3 Hours

Summary

Summary

- In this course you've learned about Kubernetes essentials
- Kubernetes is a rapidly evolving environment and there's much more that can be learned
- For additional information, see my videocourse on safari:
<https://www.safaribooksonline.com/library/view/getting-started-with/9780135237823/>