Microsoft

Kubernetes on Azure

Scott Coulton
Developer Advocate

@scottcoulton

About me.

Scott Coulton

Developer Advocate

Spent the last 4 years on container related development

I love golang

I am also a Docker Captain

@scottcoulton

scotty-c

Agenda

- Kubernetes 101
  - Introduction into Kubernetes
  - Kubernetes components
  - Deploying Kubernetes on Azure
  - Pods, services and deployments
  - Rabc, roles and service accounts
  - Stateful sets
  - Kubernetes networking and service discovery
  - Load balancing and ingress control

@scottcoulton

Course assumptions

Prior knowledge

- A basic understanding of Linux
- Be able to read bash scripts
- Understand what a container is

Equipment needed

- A bash shell (WSL is fine)
- An Azure account with access to create resources service principals
- Azure cli 2.0

@scottcoulton

Tools we will need

Please install

- kubectl https://kubernetes.io/docs/tasks/tools/install-kubectl/
- kubectx https://github.com/ahmetb/kubectx
- jq (from your package manager)

@scottcoulton

Code examples

Code for this course can be downloaded from

https://github.com/scotty-c/kubernetes-on-azure-workshop

Introduction into Kubernetes

@scottcoulton

So what is Kubernetes

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available. Google open-sourced the Kubernetes project in 2014.

Why do I need Kubernetes and what can it do?

Kubernetes has a number of features. It can be thought of as:

- a container platform
- a microservices platform
- a portable cloud platform and a lot more

Why do I need Kubernetes and what can it do?

Kubernetes provides a **container-centric** management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.

Kubernetes components

@scottcoulton

Kubernetes components

Kubernetes is broken down into two node types.

- Master node
- Worker node

Master node

A master node is responsible for

- Running the control plane
- Scheduling workloads
- Security controls

Worker node

A worker node is responsible for

- Running workloads

Master node

A master nodes components (control plane)

- kube-apiserver
- etcd
- kube-scheduler
- kube-controller-manager
- cloud-controller-manager

Worker node

A worker nodes components

- kubelet
- Kube-proxy

Kube-apiserver

The kube-apiserver is responsible for

- The entry point into the cluster
- It exposes the Kubernetes API
- It's a REST service
- Validates and configures data for the api objects

etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data

Exciting news Cosmos DB has an etcd api

kube-scheduler

Kube-scheduler is responsible for

- watches newly created pods that have no node assigned, and selects a node for them to run on

Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines

kube-controller-manager

Kube-controller-manager is responsible for

- Node Controller: Responsible for noticing and responding when nodes go down.
- Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods)
- Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.

cloud-controller-manager

Cloud-controller-manager is responsible for

- For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- For setting up routes in the underlying cloud infrastructure
- For creating, updating and deleting cloud provider load balancers
- For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

kubelet

Kubelet is responsible for

- All containers in a pod are running

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

Kube-proxy

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

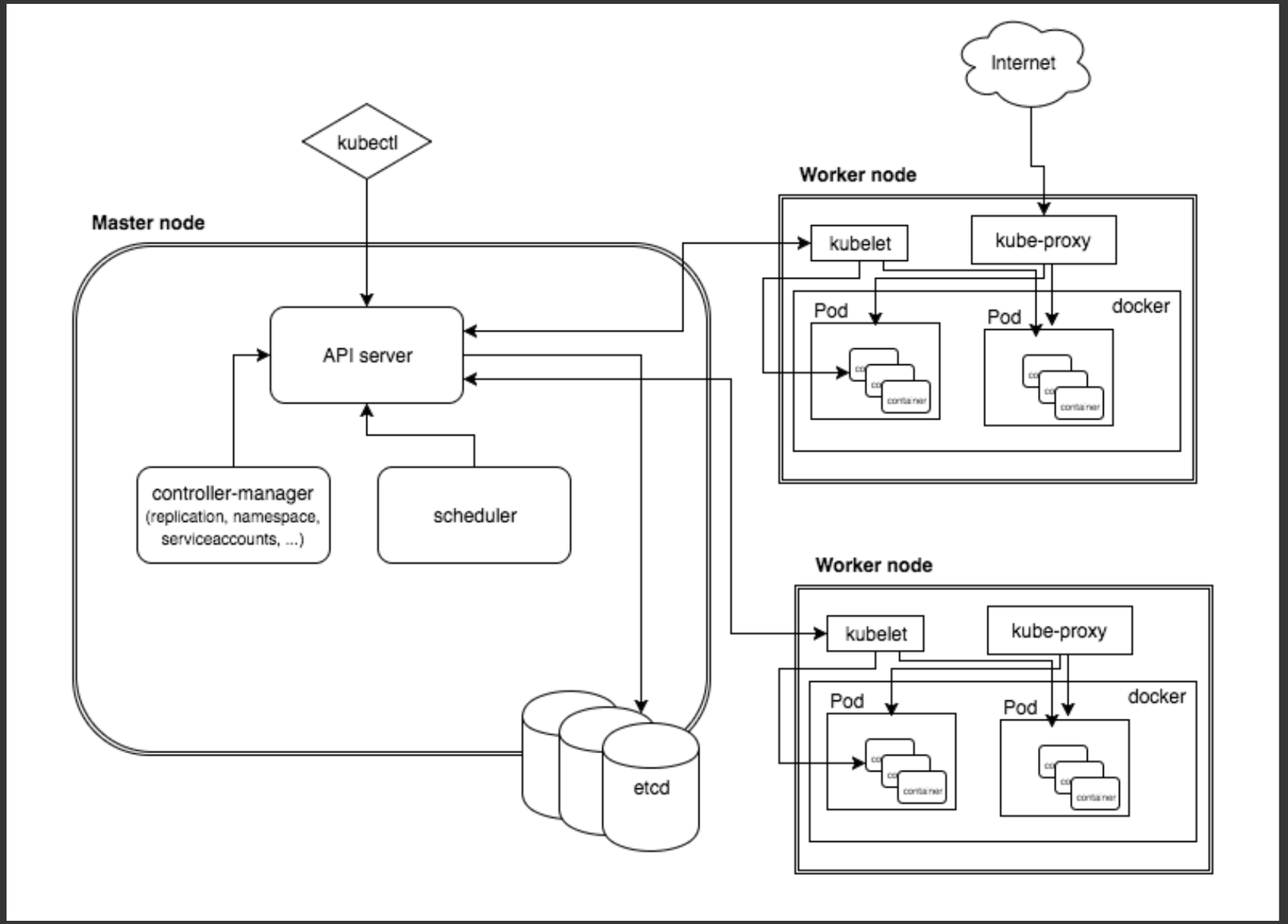
Container runtimes

Kubernetes can use differnet container runtimes

- Docker
- Moby
- Containerd
- Cri-o

At Azure we use Moby

Kubernetes architecture



Deploying Kubernetes on Azure

@scottcoulton

The offical docs are here

https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough

@scottcoulton

Create a resource group

```
az group create --name k8s --location eastus
```

Create your cluster

```
az aks create --resource-group k8s \
    --name k8s \
    --generate-ssh-keys \
    --kubernetes-version 1.12.5 \
    --enable-rbac \
    --node-vm-size Standard_DS2_v2
```

If you don’t have kubectl already

```
az aks install-cli
```

Get cluster credentials

```
az aks get-credentials --resource-group k8s --name k8s
```

Test out your cluster

```
kubectl get nodes

kubectl get pods --all-namespaces
```

Pods, services and deployments

@scottcoulton

"A pod is not equal to a container"

@scottcoulton

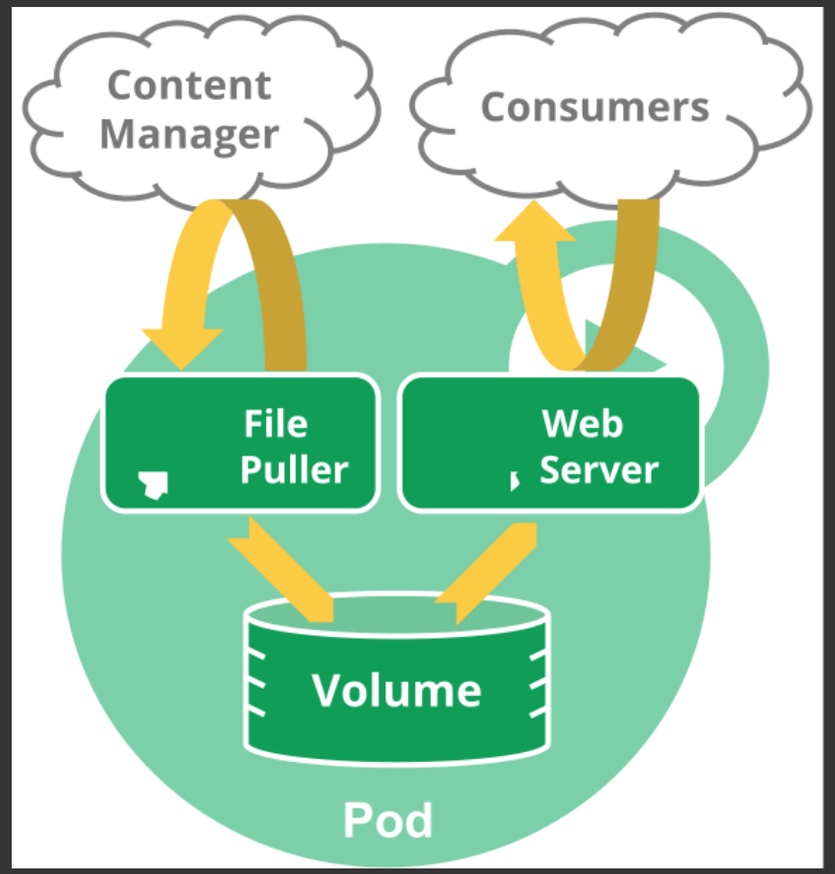
Pods

Pods are a single or group of containers that share

- localhost
- storage
- ip address
- port range

The shared context of a pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation

Pods


Content Manager
Consumers
File Puller
Web Server
Volume
Pod

Defining a pod

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! &&
sleep 3600']
```

Services

A service in Kubernetes exposes a set of pods

- Creates a vip
- Sets up basic routing to the pods
- Talks to the cloud-manager to assign a public ip or load balancer

Defining a service

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

Replicaset

A replicaset defines that state of a running application

- The amount of pods that should are running
- It self-heals the pods to their disered state

Deployments

A deployments defines the lifecycle of an application

- Is made up of pods
- It controls replicasets
- Includes the functionality to update the desired state
- Rolling updates are included

Defining a deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Let's deploy our own deployment

@scottcoulton

Our deployment

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 3
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: scottyc/webapp:latest
        ports:
        - containerPort: 3000
          hostPort: 3000
EOF
```

Check our deployment

```
kubectl get deployments
```

```
kubectl get pods
```

```
kubectl get service
```

What issue did you find ?

@scottcoulton

Answer: We have not exposed a service

@scottcoulton

Exposing our deployment

```
kubectl expose deployment webapp-deployment --
type=LoadBalancer
```

```
kubectl get service
```

To access our app http://<your-pub-ip>:3000

@scottcoulton

Rbac, roles and service accounts

@scottcoulton

rbac

Role based access control (rbac)

- Seperation of applications
- Access control for users
- Access control for applications (service accounts)

namespaces

Namespaces are the logical serperation in kubernetes
Things that are namespaced

- dns <service-name>.<namespace-name>.svc.cluster.local
- Deployments, services and pods
- Access control for applications (service accounts)
- Resource quotas
- Secrets

namespaces

Things that are NOT namespaced

- Nodes
- Networking
- Storage

Service accounts vs user accounts

The differences are

- User accounts are for humans. Service accounts are for processes, which run in pods.
- User accounts are intended to be global. Names must be unique across all namespaces of a cluster, future user resource will not be namespaced. Service accounts are namespaced.

Create a namespace

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Namespace
metadata:
  name: webapp-namespace
EOF
```

Create a service account

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: webapp-service-account
  namespace: webapp-namespace
EOF
```

Create a role

```
cat <<EOF | kubectl apply -f -
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: webapp-role
  namespace: webapp-namespace
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list", "watch"]
EOF
```

Create a role binding

```
cat <<EOF | kubectl apply -f -
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: webapp-role-binding
  namespace: webapp-namespace
subjects:
  - kind: ServiceAccount
    name: webapp-service-account
    namespace: webapp-namespace
roleRef:
  kind: Role
  name: webapp-role
  apiGroup: rbac.authorization.k8s.io
EOF
```

Deploying an application to our namepace

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment
  namespace: webapp-namespace
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: scottyc/webapp:latest
        ports:
        - containerPort: 3000
          hostPort: 3000
EOF
```

Now let's set up kubectl to use the service account

@scottcoulton

We need the secret for the service account

```
SECRET_NAME=$(kubectl get sa webapp-service-
account --namespace webapp-namespace -o json |
jq -r .secrets[].name)
```

We will get the ca

```
kubectl get secret --namespace webapp-namespace
"${SECRET_NAME}" -o json | jq -r
'.data["ca.crt"]' | base64 --decode > ca.crt
```

We will get the user token

```
kubectl get secret --namespace webapp-namespace
"${SECRET_NAME}" -o json | jq -r
'.data["ca.crt"]' | base64 --decode > ca.crt
```

Then we will create our kubeconfig file

```
context=$(kubectl config current-context)

CLUSTER_NAME=$(kubectl config get-contexts "$context" | awk '{print $3}' | tail -n 1)

ENDPOINT=$(kubectl config view -o jsonpath="{.clusters[?(@.name == \"${CLUSTER_NAME}\")].cluster.server}")

kubectl config set-cluster "${CLUSTER_NAME}" --kubeconfig=admin.conf --server="${ENDPOINT}" --certificate-
authority=ca.crt --embed-certs=true

kubectl config set-credentials "webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --
kubeconfig=admin.conf --token="${USER_TOKEN}"

kubectl config set-context "webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --kubeconfig=admin.conf
--cluster="${CLUSTER_NAME}" --user="webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --namespace
webapp-namespace

kubectl config use-context "webapp-service-account-webapp-namespace-${CLUSTER_NAME}" --
kubeconfig="${KUBECFG_FILE_NAME}"
```

Export the file to use in the current terminal

```
export KUBECONFIG=admin.conf
```

Use your kubectl commands to see what you have access too

@scottcoulton

Statefull sets

@scottcoulton

Stateful sets

Most applications will need some form of state. This is where stateful sets come in handy. This will allow us to persist our data

Before we can access the volumes we need to set up

- storage class
- pvc (persistent volume claim)

Storage classes

Storage classes are the mechanism to provision storage on various backends.

Some of the common backends are

- Local disk, NFS, ISCSI
- Cloud disks (Azure disk, AWS EBS)
- Advanced replicated storage (Rook, Portworx)

Storage classes Azure disk

In this course we will use the dynamic storage class that ships with AKS

@scottcoulton

Persistent volume claims

Persistent volume claims are the units of storage that can be attached to pods.

PVC are configured into two classes

- static
- dynamic

Creating a static claim

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: aks-volume-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
EOF
```

Using the claim

```
cat <<EOF | kubectl apply -f -
kind: Pod
apiVersion: v1
metadata:
  name: nginx-pvc
spec:
  volumes:
    - name: nginx-storage
      persistentVolumeClaim:
       claimName: aks-volume-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: nginx-storage
EOF
```

@scottcoulton

Ingress controller

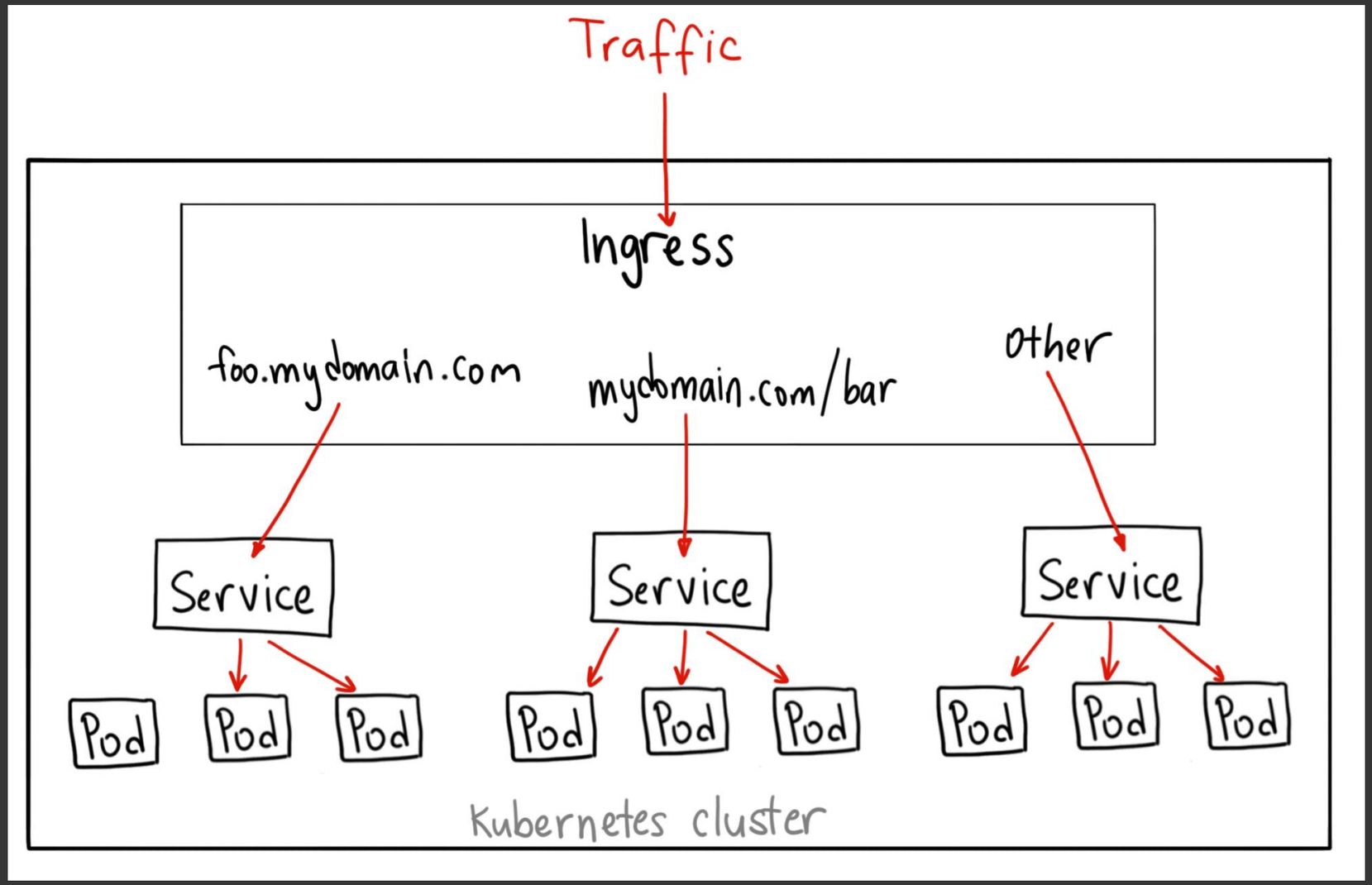
@scottcoulton

Ingress controllers

We have services what do ingress controllers give us.

- Layer 7 routing
- ssl termination
- Single IP address for multiple services

Ingress controllers



Azure HTTP routing addon

Disclaimer !!! This addon is not for production use.

https://docs.microsoft.com/azure/aks/http-application-routing

@scottcoulton

Azure HTTP routing addon

The addon gives us two things straight out of the box.

- A single ingress controller

The Ingress controller is exposed to the internet by using a Kubernetes service of type LoadBalancer. The Ingress controller watches and implements Kubernetes Ingress resources, which creates routes to application endpoints.

- External-DNS controller

Watches for Kubernetes Ingress resources and creates DNS A records in the cluster-specific DNS zone.

@scottcoulton

Enable the addon

```
az aks enable-addons --resource-group k8s --
name k8s --addons http_application_routing
```

Deploy our application

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 1
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: scottyc/webapp:latest
        ports:
        - containerPort: 3000
          hostPort: 3000
EOF
```

@scottcoulton

Deploy a service

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 3000
  selector:
    app: webapp
  type: ClusterIP
EOF
```

Add an ingress rule

```
DNS=$(az aks show --resource-group k8s --name k8s --query
addonProfiles.httpApplicationRouting.config.HTTPApplicationRoutingZoneName -o tsv)

cat <<EOF | kubectl apply -f -
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: webapp
  annotations:
    kubernetes.io/ingress.class: addon-http-application-routing
spec:
  rules:
  - host: webapp.$DNS
    http:
      paths:
      - backend:
          serviceName: webapp
          servicePort: 80
        path: /
EOF
```

Microsoft

Azure ❤