

# Kubernetes Technical Briefing

Module 3:  
Intermediate Kubernetes  
Topics



# Agenda

- Volumes and Persistence
- Multi-Container Pods
- Init Containers
- Jobs and Cron Jobs
- Daemon Sets
- Health Probes
- Ingress
- Helm Package Manager

# Volumes and Persistence



# Kubernetes Volumes

- Deleting or restarting a POD destroys any persistent data stored in the containers file system.
- Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod.
- At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a Pod.
  - Azure Disk -> /mnt/external
- A Pod can use any number of volume types simultaneously.

# Ephemeral Volumes



- Ephemeral volume types have a lifetime of a pod. Ephemeral volumes include:
- **Empty Dir** – A temporary folder for all containers within a Pod to read/write to.
- **Host Path** – Mounts a file or directory from the host node's filesystem into your Pod. Not practical in a multi-node cluster.
- **Config Map/Secret** – A Read-only folder that provides a way to inject configuration data into pods.

```
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    volumeMounts:
    - name: tempVol
      mountPath: /usr/share/nginx/html
      readOnly: true
    - name: hostVal
      mountPath: /usr/home/somefile
  volumes:
  - name: tempVol
    emptyDir: {}
  - name: hostVal
    hostPath:
      path: /var/local/aaa/1.txt
      type: FileOrCreate
```

# Persistent Volumes



- Persistent volumes exist beyond the lifetime of a pod. Persistent volumes include:
- **Azure Disk** – Mounts an Azure Data Disk into a pod.
- **Azure File** – Mounts an Azure File Share (SMB 2.1/3.0) to a Pod.
- **Persistent Volume Claim** - A way for users to "claim" durable storage without knowing the details of the cloud environment.

```
volumeMounts:
- name: shareVol
  mountPath: /usr/share/nginx/html
- name: diskVol
  mountPath: /usr/data
volumes:
- name: shareVol
  azureFile:
    secretName: azure-storage
    shareName: data
    readOnly: false
- name: diskVol
  azureDisk:
    kind: Managed
    diskName: tempdisk2
    diskURI: "/subscriptions/aa82959...
```



# Persistent Volumes



- A **PersistentVolume** (PV) resource is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using **Storage Classes**.
- PVs are volume plugins like Volumes but have a lifecycle independent of any individual Pod that uses the PV.
- It is a cluster wide resource (not namespaced), available to for any user to claim.
- There are two ways PVs may be provisioned:
  - **Statically** - A cluster administrator creates multiple PVs that carry the details of the real storage, which is available for use by cluster users.
  - **Dynamically** - When none of the available static PVs match a user's Persistent Volume Claim, the cluster will try to dynamically provision a volume specially for the PVC, based on a specified Storage Classes.

# Static Persistent Volume Claims

- Once a **PersistentVolume** has been created, it can be claimed by a **PersistentVolumeClaim**. The PVC can then be mounted as a volume in the Pod.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    usage: file-storage
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  azureFile:
    secretName: azure-storage
    shareName: data
    readOnly: false
```



```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: file-storage-claim
  annotations:
    volume.beta.kubernetes.io/s
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      usage: file-storage
```



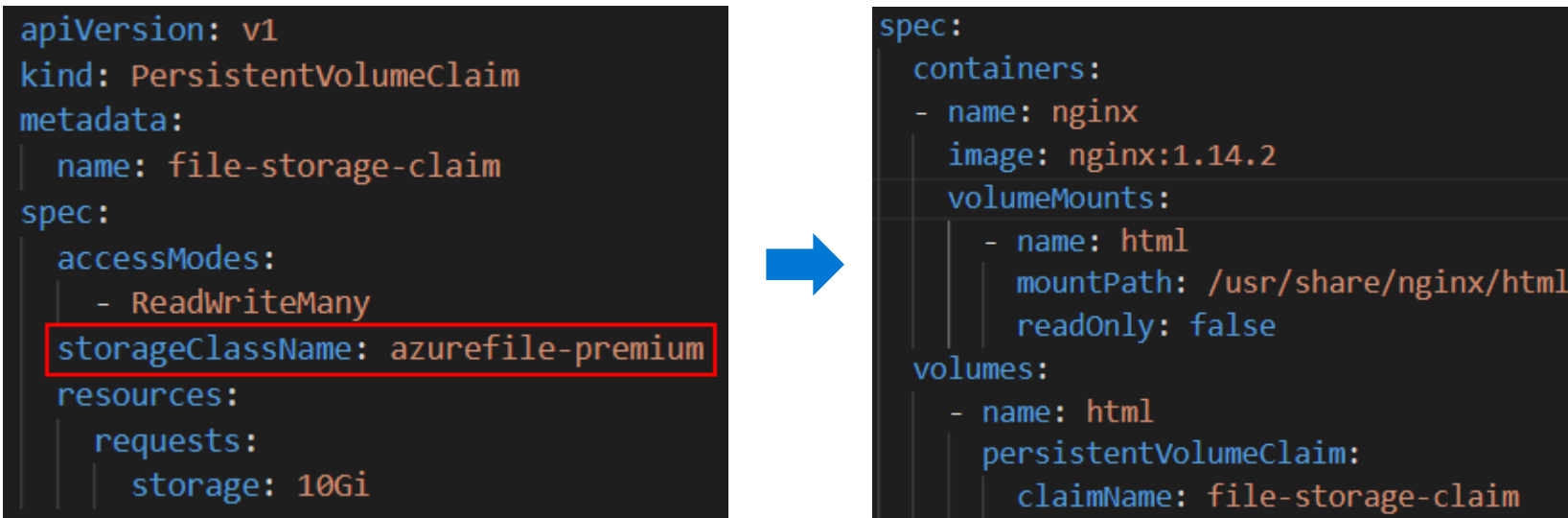
```
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
  volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: false
  volumes:
    - name: html
      persistentVolumeClaim:
        claimName: file-storage-claim
```

- This is an example of a *Statically* provisioned **PersistentVolumeClaim**.



# Dynamic Persistent Volume Claims

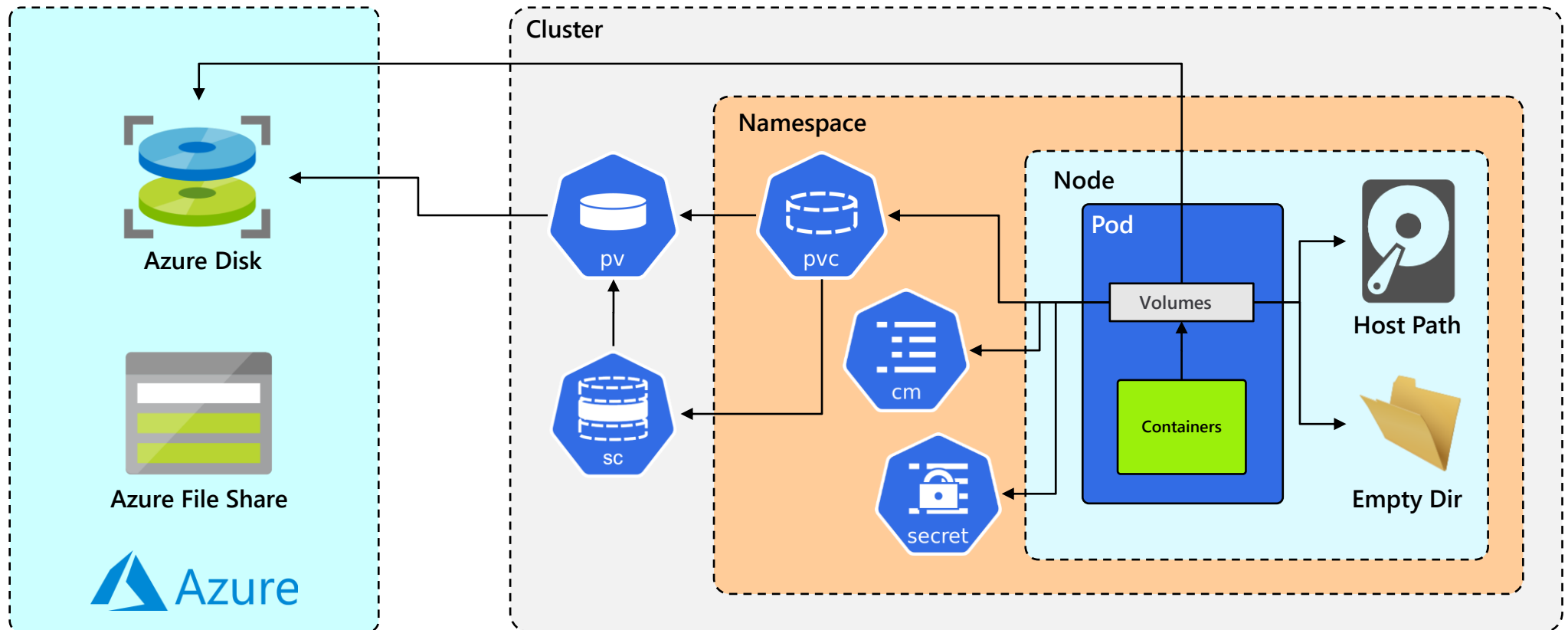
- A **PersistentVolumeClaim** can also be provisioned dynamically by specifying the **StorageClass** (wrapper around driver) to use when creating the PVC.



- A storage account is automatically created in the *infrastructure resource group* for use with the storage class to hold the Azure file shares.
- NOTE:** If the cluster is deleted (like cattle), all the resources in the *infrastructure resource group*, including dynamically provisioned file shares and disks, will also be deleted.

# Overview of Kubernetes Volumes

Kubernetes offers many flexible options for mounting volumes into Pods.



# Volumes Demo

Working with Volumes

# Persistent Volume with Multiple Pod Replicas

- A **PersistentVolume** can be mounted on a host in any way supported by the resource provider.
- Scaling a Deployment introduces the problem persistent volumes across multiple Pod replicas.
- What happens when multiple Pods, on different Nodes, attempt to access the same Persistent Volume?

# Advanced Persistent Volumes Demo

Working with Persistent Volumes when Scaling Pods

# Persistent Volume Access Modes

- Persistent Volumes support the following **Access Modes**, which determine how multiple Pods mount the same volume:
- **ReadWriteOnce (RWO)** – Mount a volume as read-write by a single node.
- **ReadOnlyMany (ROX)** – Mount the volume as read-only by many nodes.
- **ReadWriteMany (RWX)** – Mount the volume as read-write by many nodes.
- While the ***ReadWriteMany*** access mode would allow multiple Pods to share a volume across multiple Nodes, most external volume providers do not support this functionality.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: file-storage-claim
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: azurefile
  resources:
    requests:
      storage: 10Gi
```

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-



# Multi-Container Pods

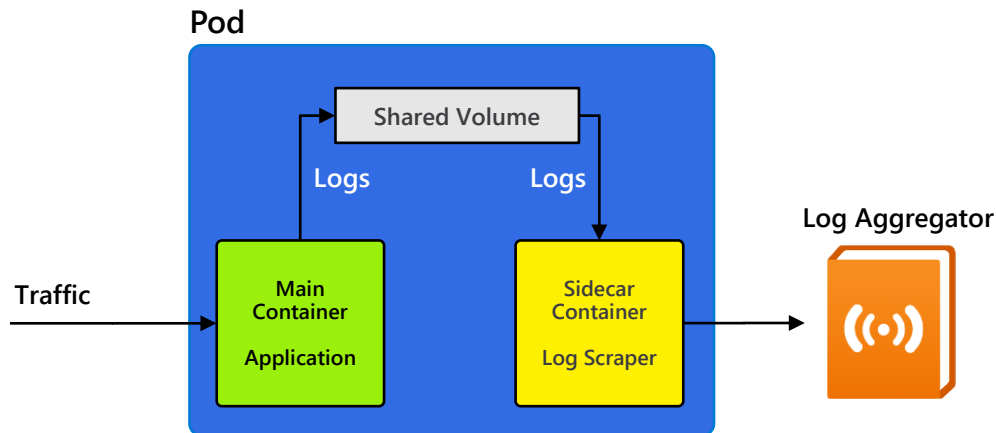


# Keep Highly Coupled Containers Together

- As a rule, most Pods only contain a single container. However, there are times when it makes sense to put multiple, related containers inside a single Pod.
- Containers running in a Pod share the same IP, port and communicate using native inter-process communication channels (localhost).
- Containers in a Pod also share the same lifecycle and are scaled together.
- When one container fails, the status of the Pod changes to a non-running state, which prevents other containers from receiving traffic.
- Pods Restarts count is the total number of restarts of all its containers.
- Only highly-coupled related containers (those that interact with each other frequently) should be placed in the same Pod.

# Sidecar Pattern

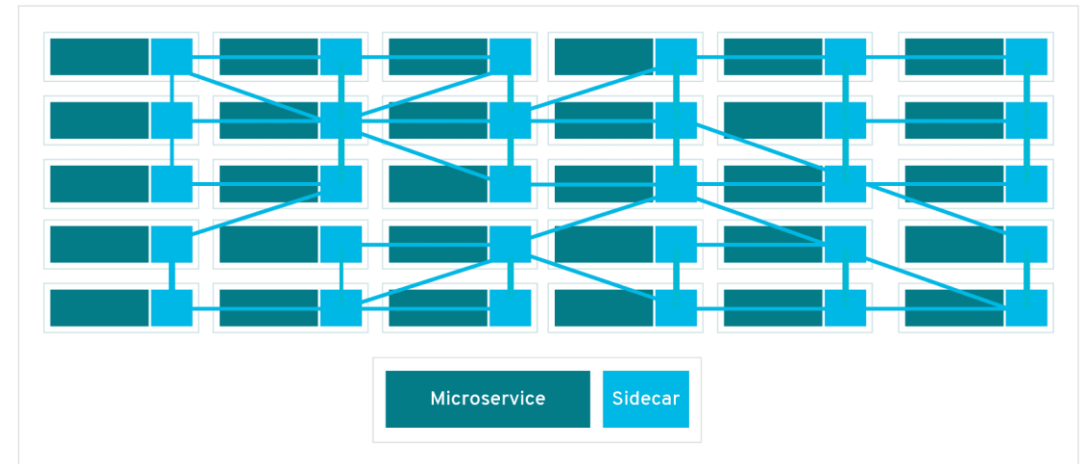
- A common scenario for using multiple container in a Pod is to implement the Sidecar Pattern.
- A Sidecar container is a second container added to the Pod definition because it needs to access the same resources being used by the main container.



```
spec:
  volumes:
  - name: shared-logs
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log/nginx
  - name: sidecar-container
    image: busybox
    command: ["sh", "-c", "while true;"]
    volumeMounts:
    - name: shared-logs
      mountPath: /var/log/nginx
```

# Sidecar Envoy Proxy

- The [Envoy Proxy](#) project implements the Sidecar pattern by automatically injecting an Envoy container into every Pod in a namespace.
- Such a system is called a Service Mesh.
- Service meshes provide microservices support services like:
  - Dynamic service discovery
  - Load balancing
  - Health checks
  - Encryption
  - Circuit breakers
  - Authentication/Authorization
  - Observability and Rich metrics
- Used by Istio, Linkerd and other popular service meshes.



# Multi-Container Pods Demo

Working with Multi-Container Pods

# Init Containers

3



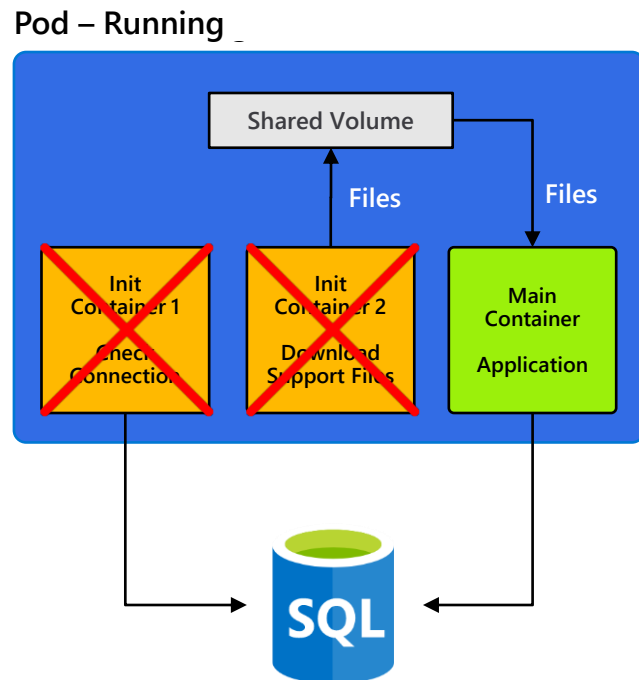


# Init Containers

- Sometimes containerized applications require dependencies (like external databases connection) to be available before they can start.
- Some applications can't start until costly initialization steps are performed.
- Rather than adding initialization logic to every application, the process can be delegated to a separate *initialization* container.
  - Such an *init container* runs once before the main container is created
  - If initialization is successfully, the *init container* is deallocated and the main container is created.
  - If initialization fails, the Pod will keep restarting the *init container*. None of the main containers will be created until this step completes.
  - If multiple *init containers* are specified, each one is *run one at a time, in the order they're defined in the spec*, until they all complete successfully.
  - After Pod startup, *init containers* are not listed as part of the Pod container inventory.

# Running Init Containers

- Creating and running ***init*** containers is part of the Pod start up process.
- Pods do not reach a Running state until this process is completed successfully.



```
spec:
  initContainers:
  - name: homepage
    image: busybox
    args:
    - "/bin/sh"
    - "-c"
    - "wget -O /work-dir/index.html http://neverssl.com/online;"
    volumeMounts:
    - name: tempvol
      mountPath: /work-dir # internal mount path
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: tempvol
      mountPath: /usr/share/nginx/html
```

# Init Containers Demo

Working with Init Containers

# Jobs and Cron Jobs

4



# Container Restart Policy

- When Kubernetes controller detects a container has failed, it will attempt to restart it based on the container's ***restartPolicy***:
  - Always
  - Never
  - OnFailure
- A *Deployment* controller expects Containers to run continuously (think of a web server or database server always listening for requests/queries).
- The only *restart policy* a Deployment supports is ***Always***.
- Sometimes, you may want to run a Container that performs a finite task and then just stop.
- Just like an *Init Container*, but without creating any other Containers afterwards.

# Jobs

- A **Job** is a Kubernetes controller that creates and runs task-based workload.
- A **Job** creates a Pod, whose Container(s) is expected to execute once, run for a short period of time and then stop.
- When the Container ends its processing, the Pod goes into a Completed state.
- A **Job** can specify that multiple Pods be created (**completions**) and how many Pods should be created at the same time (**parallelism**).
- Containers defined in a Job only support **Never** and **OnFailure** restart policies.

```
apiVersion: batch/v1
kind: Job
...
spec:
  completions: 10
  parallelism: 4
  backoffLimit: 3
  activeDeadlineSeconds: 100
  template:
    metadata:
      name: countdown-multi
      labels:
        app: countdown-multi
    spec:
      containers:
        - name: counter
          image: centos:7
          command:
            - "bin/bash"
            - "-c"
            - "for i in 3 2 1 ; do
restartPolicy: OnFailure
```



# CronJobs

- A **CronJob** is a Kubernetes controller that creates and manages **Jobs** according to a cron schedule.
- **CronJobs** are useful for creating periodic and recurring tasks, like running backups or sending emails.
- **CronJobs** maintain a certain number of successful and failed jobs, so their logs can be examined.
- When a **CronJob** deletes a Job, that Job's Pods are deleted.

```
apiVersion: batch/v1beta1
kind: CronJob
...
spec:
  schedule: "*/1 * * * *"
  successfulJobsHistoryLimit: 2
  failedJobsHistoryLimit: 2
  jobTemplate:
    spec:
      completions: 10
      parallelism: 4
      template:
        ...
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from
                - sleep 2
              restartPolicy: OnFailure
```

# Jobs and CronJobs Demo

Working With and Scheduling Jobs

# Daemon Sets

5



# Daemon Sets

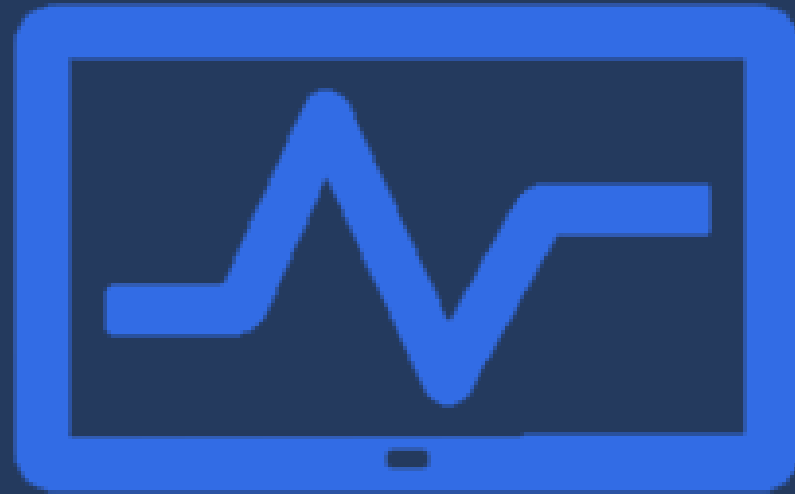
- **DaemonSets** are controllers that ensure new Nodes run an instance of a Pod.
- As nodes are added to the cluster, Pods are scheduled on them automatically.
- As nodes are removed from the cluster, those Pods are terminated.
- Most system **DaemonSets** create Pods that can tolerate any taint.
- Some typical uses for a **DaemonSet** are:
  - Running a cluster storage daemon on every node
  - Running a logs collection aggregator
  - Running a node monitoring agent on every node (Ex: omsagent for *Azure Monitor*).
  - Running a CSI Secrets provider (Ex: azure-secrets-provider for *Azure Key Vault*)

# DaemonSets Demo

Exploring System DaemonSets

# Health Probes

6





# Health Checks

- How do you know if a container has completed its start up sequence and is ready to accept requests?
- What if a container is “running” but not functioning correctly?
  - For example, a webserver not receiving/processing requests
- Kubelet uses **probes** to monitor running containers and determine if they’re running and/or can receive traffic from services.
- Based on information it receives from probes, a Kubelet can restart containers or notify Kube-proxy to update its Endpoints and take Pods out of a service’s load balancer rotation.

# Startup, Liveness and Readiness Probes

- There are three different types of probes available to attach to containers:
  - **Startup Probe** – Detects when an application has started. Some applications, like messaging servers, have extended startup sequences. Startup probes delay other probes from inadvertently telling the *Kubelet* restart the container for being unresponsive.
  - **Liveness Probe** – Once the application has started, a liveness probe monitors it periodically to ensure it's still running and tell the *Kubelet* to restart the container if it doesn't get a response.
  - **Readiness Probe** – Applications which service external requests, through a Service, can be monitored to ensure they are still receiving those requests. When they stop receiving requests, Readiness Probes can tell the *Kubelet* to exclude those pods from the service's load balancer rotation.

```
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    startupProbe:
      tcpSocket:
        port: 80
      failureThreshold: 30
      periodSeconds: 10
    livenessProbe:
      httpGet:
        path: /healthy
        port: 80
      failureThreshold: 1
      periodSeconds: 10
    readinessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 5
      timeoutSeconds: 2
      periodSeconds: 10
```

# Health Probes Demo

Using Health Probes to Monitor Containers

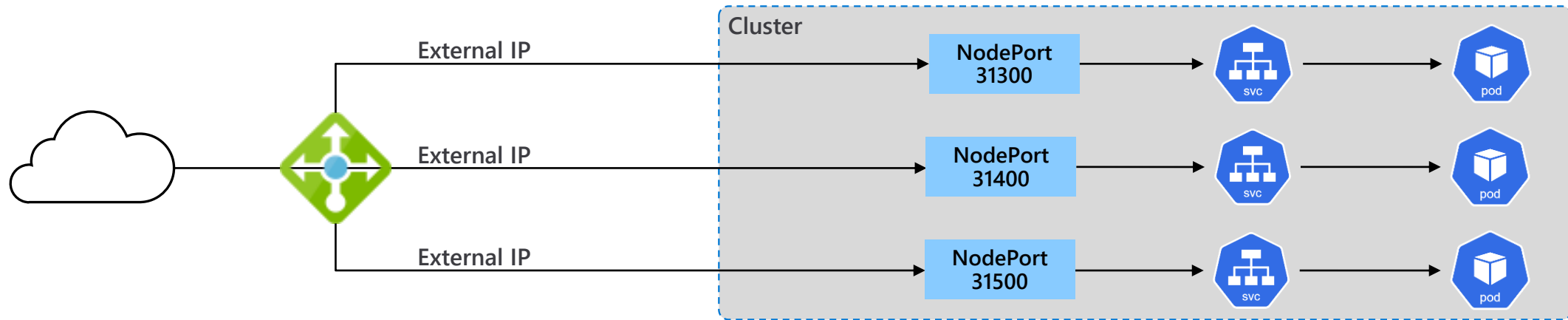
# Ingress

7



# Kubernetes Load Balancer vs Ingress

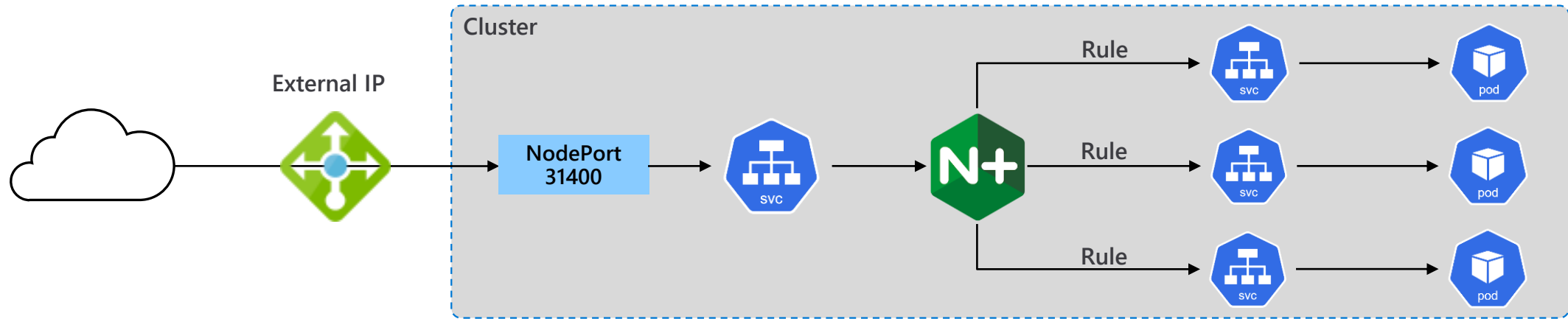
A Kubernetes Load Balancer maps an external IP to a single service endpoint inside the cluster.



- Each Kubernetes Load Balancer service requests an external IP address from Azure.
- For multiple service endpoints, you need to create multiple Load Balancers and IPs.
- If you expose dozens or hundreds of endpoints, this approach doesn't scale well.

# Kubernetes Load Balancer vs Ingress

An **Ingress** Resource enables routing to internal traffic through a single external endpoint.



- Routing rules are defined in **Ingresses** in each namespace.
- A single **Ingress Controller** combines Ingress rules across all namespaces and routes external HTTP and HTTPS (layer 7 only) traffic to internal **Cluster IP** services.
- Kubernetes does not provide a concrete implementation of an **Ingress Controller** to perform this routing.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "*.foo.com"
    http:
      paths:
      - pathType: Prefix
        path: "/foo"
        backend:
          serviceName: service2
          servicePort: 80
      - pathType: Prefix
        path: "/xyz"
        backend:
          serviceName: service4
          servicePort: 4053
```

# 3<sup>rd</sup> Party Ingress Controller

- To support **Ingress** resources, you must install a 3<sup>rd</sup> party **Ingress Controller**.
- Kubernetes supports the following popular Ingress Controllers:
  - The [NGINX Ingress Controller for Kubernetes](#) works with the [NGINX](#) webserver (as a proxy).
  - The [Traefik Kubernetes Ingress provider](#) is an ingress controller for the [Traefik](#) proxy.
  - The [AKS Application Gateway Ingress Controller](#) is an ingress controller that configures the [Azure Application Gateway](#).

# Ingress Controller Demo

Working with Ingress Resource and Ingress Controllers



# Helm Package Manager

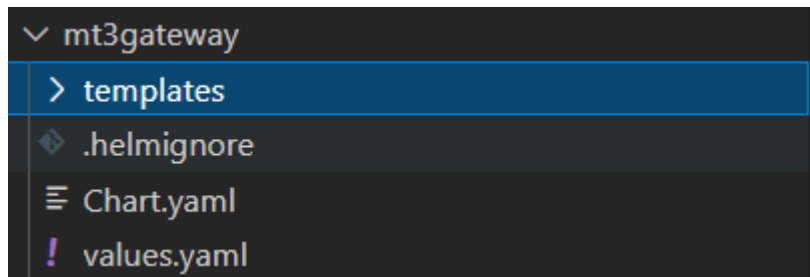


# Helm Package Manager

- Helm is a package manager for Kubernetes-based applications.
- Helm **Charts** help you define, install, and upgrade even the most complex Kubernetes application.
- Charts are easy to create, version, share, and publish.
- Helms can package and preconfigure Kubernetes applications into charts, which allow you to:
  - Manage and share your own application chart packages
  - Leverage other popular software packaged as charts
  - Create reproducible builds of Kubernetes applications
  - Manage you Kubernetes manifest files
- You can Push and Pull Helm charts to Azure Container Registries.

# A Helm Chart Manages an Entire Application

- Helm streamlines installing/upgrading applications
  - Perform consistent releases without dealing with dozens of files
  - Update/rollback releases - as needed.
- A Helm **Chart** consists of:
  - A **Chart.yaml** file containing metadata about the chart
  - An optional **values.yaml** file containing parameters you can use in Kubernetes *manifest* files
  - Parameters can be used for substitutions or in macros.
  - A **templates** folder containing your manifest files.



```
apiVersion: v2
appVersion: "1.0"
description: A Helm chart for Math Trick
name: mathtrick3gateway
version: 0.1
```

**Chart.yaml**

```
repo: kizacr.azurecr.io
namespace: gateway
platform: dotnet
tags:
  mt3gatewayweb: latest
  mt3gatewaygateway: latest
```

**values.yaml**

```
spec:
  containers:
    - name: mt3chained-step1
      image: {{ .Values.repo }}/mt3chained
      ports:
```

**mt3chained-step1-dep.yaml**

# A Helm Chart Contains the Manifests

- The **templates** folder contains all application related manifest files.
- Sub-folder structure under templates doesn't matter to Helm. You can organize your manifest files as desired.
- The **helm** command is used to manage charts:

```
helm [command] [release name] [chart name] [flags]
```

- **Command** - Specifies the operation that you want to perform. Some valid commands include ***install, upgrade, ls, uninstall***.
- **Release Name** - Specifies the name you want to give to the release. This is just for management purposes.
- **Chart Name** – Specifies the name of the chart you want to install/upgrade. Can be a folder name for a local chart or package name for an external chart.
- **Flags** – Optional flags.

```
▼ mt3chained
  ▼ templates
    ▼ steps
      ≡ mt3chained-step1-dep.yaml
      ≡ mt3chained-step1-svc.yaml
      ≡ mt3chained-step2-dep.yaml
      ≡ mt3chained-step2-svc.yaml
      ≡ mt3chained-step3-dep.yaml
      ≡ mt3chained-step3-svc.yaml
      ≡ mt3chained-step4-dep.yaml
      ≡ mt3chained-step4-svc.yaml
      ≡ mt3chained-step5-dep.yaml
      ≡ mt3chained-step5-svc.yaml
    ≡ mt3chained-cm.yaml
    ≡ mt3chained-web-dep.yaml
    ≡ mt3chained-web-ing.yaml
    ≡ mt3chained-web-svc.yaml
  ◆ .helmignore
  ≡ Chart.yaml
  ! values.yaml
```

# Helm Examples

- **Example**: Install a local chart in the *mychart* folder, giving it a release name of *demo*. Use the **-set** flag to override a parameter specified in the **values.yaml** file

```
helm install demo mychart --set repo=myrepo.azurecr.io
```

- **Example**: Apply any changes to manifests files by upgrading a previously installed chart

```
helm upgrade demo mychart
```

- **Example**: Adds a reference to the *scubakiz* external helm repo and installs the *clusterinfo* chart from that repo.

```
helm repo add scubakiz https://scubakiz.github.io/clusterinfo/
```

```
helm install clusterinfo scubakiz/clusterinfo
```

- **Example**: Upgrades an existing chart (or installs it if it doesn't exist) with the parameters specified in an alternate **values.staging.yaml** file

```
helm upgrade --install demo mychart -f values.staging.yaml
```

- **Example**: Uninstalls a previously installed chart. All resources in *templates* are deleted.

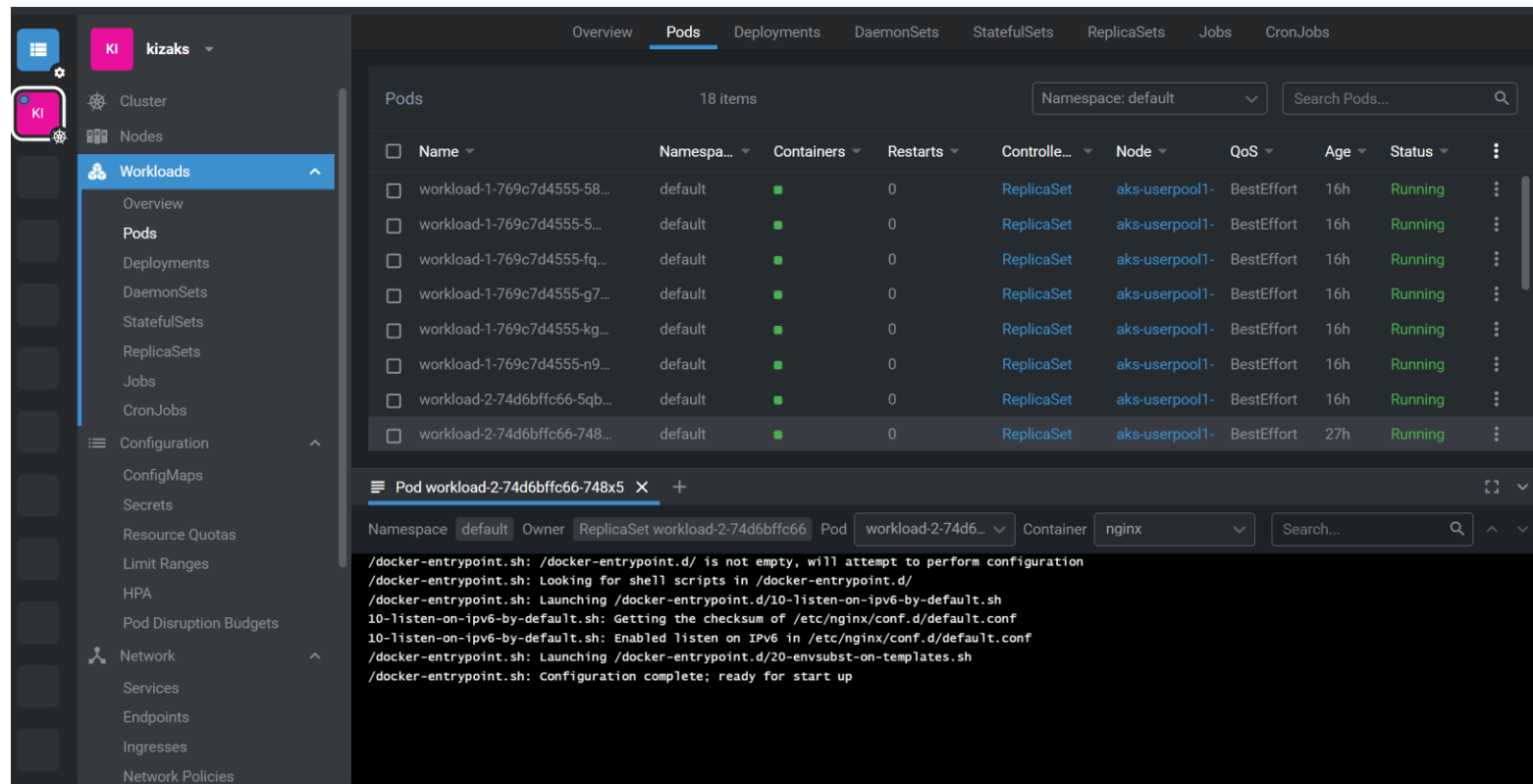
```
helm uninstall demo
```

# Helm Demo

Use Helm to manage application releases to an AKS Cluster

# PRO TIP: Use Lens to Manage your Cluster

- It's important to be very comfortable using **kubect**l to manage your cluster.
- Once you understand the fundamentals of Kubernetes, it's often more efficient to use a free desktop GUI like [Lens](#) to observe and manage your cluster.





Lab – Module 3

# Intermediate Kubernetes Topics







Thank you