Microsoft

Kubernetes Technical Briefing

Module 6:
Advanced Kubernetes Topics
Part 1

Agenda

- Integration with Azure Key Vault
- Kubernetes RBAC with Azure AD
- Node Selectors/Affinity
- Taints and Tolerations
- Pod Affinity/Anti-Affinity
- Pod Topology Spread Constraints
- Stateful Sets
- Network Policy

Integration with Azure Key Vault

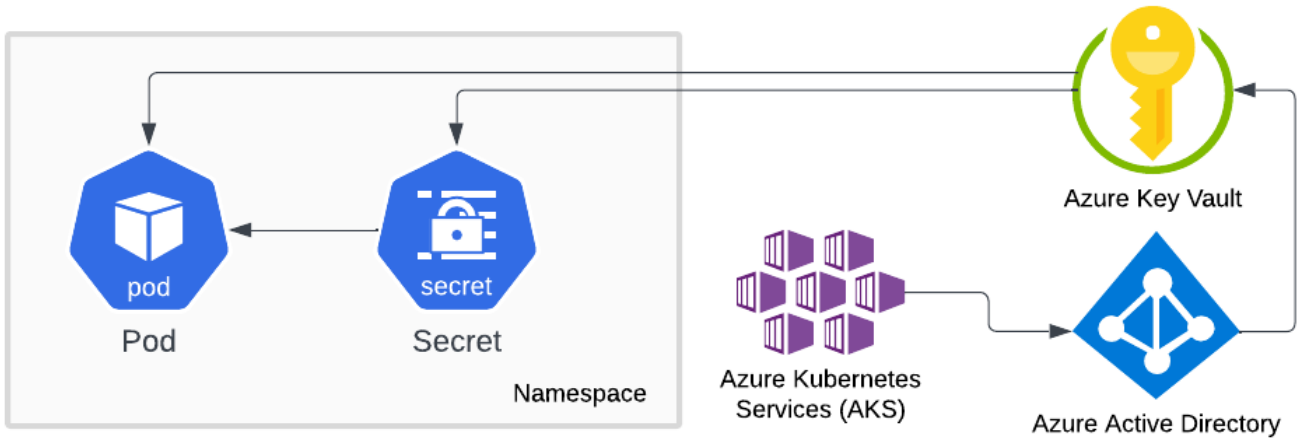
1

Don’t Use Secrets to Store Sensitive Information

- Kubernetes Secrets are *encoded*, NOT *encrypted*.
- It’s not good practice to save sensitive content in YAML manifests or check Secrets resources into source control.
- In a production environment, it's best to keep sensitive information in an external *vault* and inject that content only when and where it's needed.
- Azure Kubernetes Service supports integration with ***Azure Key Vault*** to facilitate best practices when accessing sensitive information.

Azure Key Vault

- ***Azure Key Vault*** is a cloud service for securely storing and accessing secrets.
- A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, or cryptographic keys.
- When you enable Azure Key Vault support in your AKS cluster, you'll be able to automatically create Kubernetes Secrets in your cluster when you need them.
- You also have the option of directly injecting sensitive content directly into containers, without creating Kubernetes Secrets first.



Azure Key Vault Integration

- When Azure Key Vault is enabled in AKS, a custom resource called ***SecretProviderClass*** is available.
- This manifest provides the connection information to Key Vault.
  - The ***secretObjects*** section is optional and only used if you want to create a Kubernetes Secret object.
- When a Pod mounts the ***Secrets CSI*** volume, the Key Vault secret(s) is injected into the container (and a Secret object is created if specified).
- If a Secret object is created, it's automatically deleted when the last Pod that accesses that Secret is deleted.

```yaml
  volumeMounts:
  - name: secrets-kube-kv
    mountPath: /mnt/kv-secrets
    readOnly: true
volumes:
  - name: secrets-kube-kv
    csi:
      driver: secrets-store.csi.k8s.io
      readOnly: true
      volumeAttributes:
        secretProviderClass: azure-kv-secret
```

```yaml
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: azure-kv-secret
spec:
  provider: azure
  secretObjects:
  - data:
    - key: secret-db-connection
      objectName: dbConnection
    secretName: local-secret-1
    type: Opaque
  parameters:
    usePodIdentity: "false"
    keyvaultName: kiz-kube-kv
    useVMManagedIdentity: "true"
    userAssignedIdentityID: 4a0d4211-310e-
    tenantId: 72f988bf-86f1-41af-91ab-2d7c
    objects: |
      array:
        - |
          objectName: dbConnection
          objectType: secret
          objectVersion: ""
```

Azure Key Vault Demo

Injecting Secrets from Azure Key Vault into Containers – Review Lab

Kubernetes Role Based Access Control (RBAC) with Azure AD

2

Kubernetes Role-based Access Control (RBAC)

RBAC is a method of regulating access to resources based on the roles of individual users or service accounts within your organization.

RBAC Components:

- **Service Account** - Provides an identity for processes that run in a Pod.
- **Role** – A list of rights (permissions) to specific resource types within a namespace.
- **RoleBinding** – Defines the *binding* of a user/service account to a Role within a namespace
- **ClusterRole** – A cluster-wide resource listing permissions for specific namespaces, all namespaces or cluster-scoped resources.
- **ClusterRoleBinding** – Defines the binding of user/service account to a **ClusterRole** throughout the cluster

Service Accounts

- ***Service Accounts*** are resources maintained by the Kubernetes API Server.
- Each namespace has a ***default*** Service Account, that's created when the namespace is created.
- By default, all containers run in the context of the default Service Account.
- Service Accounts are scoped within a namespace.

NOTE: If a container is designed to interact with the API Server, it's best practice **NOT** to use the *default* service account. Each application should create its own, Service Account.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: default
secrets:
- name: default-token-qd87j
```

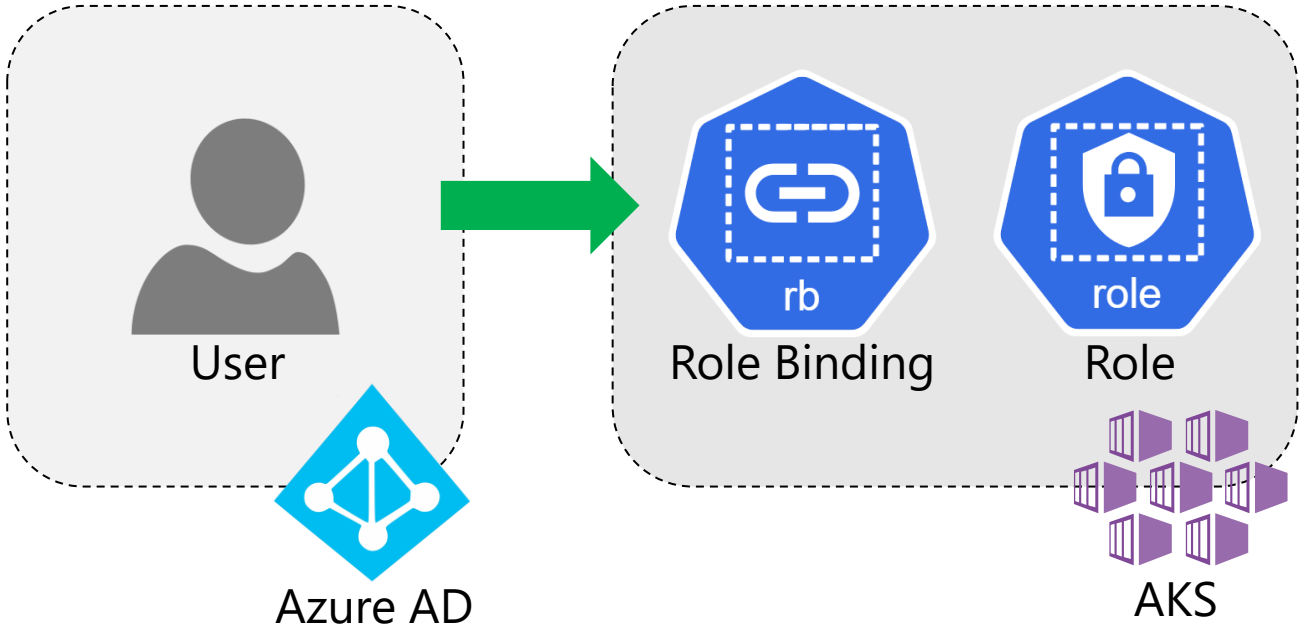
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: default
  containers:
```

User Accounts

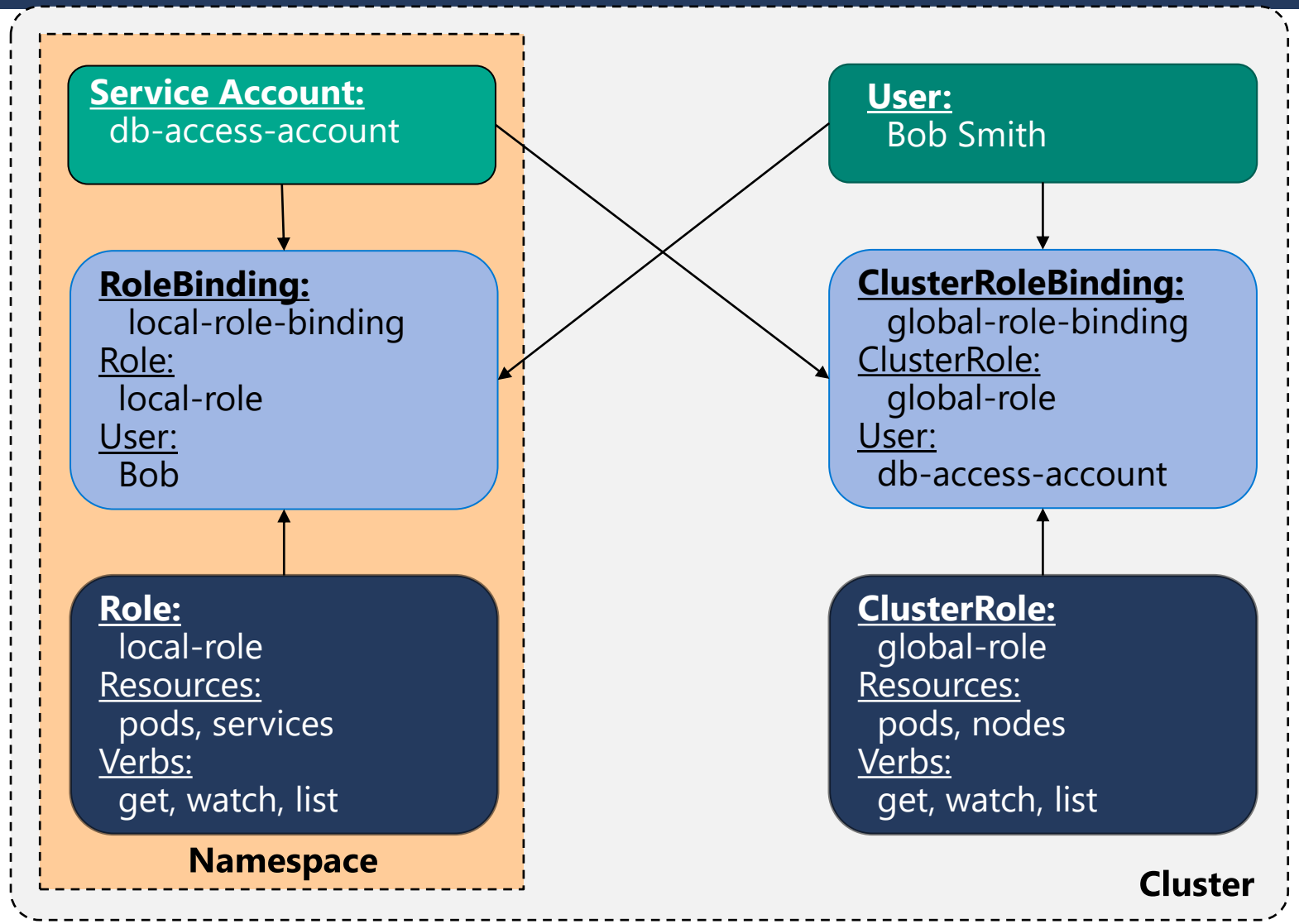
- Unlike *Service Accounts*, Kubernetes **does not** have objects which represent normal user accounts (no **Kind: User**).
- Kubernetes supports several interfaces for external authentication.
  - Static Passwords or Tokens
  - X.509 Certificates
  - **Single Sign-On using OpenID**
  - Authentication Proxy
  - Webhook Token Authentication
- Azure Active Directory can facilitate external user authentication into AKS clusters with OpenID Connect.
- Externally authenticated users can be *bound* to *Roles/ClusterRoles (RBAC)*, which facilitate resource authorization within the cluster.

Kubernetes RBAC with Azure AD Integration

- Kubernetes RBAC is enabled by default in AKS. Disabling RBAC is **not recommended**.
- To enable Azure AD integration, use the `--enable-aad` option when creating or updating an AKS cluster.
- Kubernetes RBAC works with Azure AD as follows:
  1. A user is authenticated in Azure
  2. The user's token and email is passed to the cluster.
  3. A **RoleBinding** resource binds the user to a **Role**, which define which actions are allowed on which Kubernetes resources.



Kubernetes RBAC Illustrated



RBAC Demo

Review Roles and Bindings

Node Selectors / Node Affinity

3

Node Selectors

- Pods can use Node labels to specify which Nodes to be scheduled on.
- Labels *must be* on the Nodes prior to deploying Pods selecting the Nodes.
- If selected Node label is not found on any node, Pods *will not get scheduled*.

```yaml
apiVersion: v1
kind: Node
metadata:
  annotations:
    node.alpha.kubernetes.io/ttl: "0"
    volumes.kubernetes.io/controller-managed-
  labels:
    kubernetes.io/os: linux
    size: large
    topology.kubernetes.io/region: us-east-1
    topology.kubernetes.io/zone: us-east-1a
```

```yaml
spec:
  nodeSelector:
    kubernetes.io/os: linux
  containers:
  - name: nginx
    image: k8slab/nginx:1.0
    ports:
    - containerPort: 80
      protocol: TCP
```

Node Affinity

- The affinity/anti-affinity language is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation
- Can indicate that the rule is soft "preference" rather than a hard requirement, so if the scheduler can't satisfy it, the Pod will still be scheduled.
- Use ***weight*** to set preference order.

```yaml
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

Taints and Tolerations

Taints and Tolerations

- Since a cluster can be a collection of heterogenous nodes, sometimes it makes sense to set a policy restricting which Pods are scheduled on which Nodes.
- Nodes can be marked as "*tainted*" with particular attributes
- Pods can be designated as being able to "*tolerate*" certain taints
- Depending on the taint-effect, the scheduler will decide if a Pod can be scheduled on a tainted Node.
- Node taint-effects can be:
  - **NoSchedule** – New Pods will NOT be scheduled on the tainted node unless they can tolerate the taint
  - **PreferNoSchedule** – Pods CAN be scheduled ONLY IF they won't fit on any other node.
  - **NoExecute** – New Pods will NOT be scheduled on the tainted node *and* existing Pods without toleration will be EVICTED from a tainted node.

Taints and Tolerations with Node Selectors/Affinity

- Adding tolerations to Pod for tainted Nodes does *NOT* guarantee those Pods will be scheduled those tainted Node.
- Taints, Tolerations and Node Selectors/Affinity can be used together to achieve complex results.
- **Example:** You want certain Pod to be scheduled on specific types of Nodes and you don't want other types of Pods on those same Nodes:
  - Add labels to the Nodes specifying a special attribute
  - Add taints to the Nodes to keep out intolerant Pods
  - Add tolerations to your Pods to tolerate the taints
  - Add a Node Selector or Affinity to your Pods that select matching node labels.
- **Results:** Only desired Pods will be scheduled on the tainted Nodes.

Non-Technical Example

1. You and your friends (**containers in a pod**) want to go out to dinner.
2. You have 3 restaurants (**nodes**) to choose from.
3. One restaurant puts up a sign (**label**) saying they're a *Fancy* restaurant.
4. The Fancy restaurant adds a *dress code* (**taint**) and does not allow customers inside (**NoSchedule**) who are not properly attired. Most customer decide to go elsewhere.
5. You and your friends put on nice clothes (**tolerations**) so you can enter fancy restaurants.
6. However, there's nothing to say that you must go to a fancy restaurant. You can go anywhere.
7. You and your friends want to *only* go to fancy restaurants (**node selector/affinity**).

Node Selectors, Taints and Tolerations Demo

Coordinating Workloads using Node Selectors, Taints and Tolerations

Pod Affinity/Anti-Affinity

5

Pod Affinity and Anti-Affinity

- Uses Labels Selectors that match labels on other Pods
- Not recommend on large clusters (over several hundred Nodes)
- Requires nodes to be consistently labeled using *Well-Known Labels* of a *topologyKey*.

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: target
          operator:  In
          values:
            - pvc-pod
      topologyKey: "kubernetes.io/hostname"
```

Actions for Affinity

- **Required During Scheduling Ignored During Execution** – Will not schedule new non-matching Pods but will leave existing Pods running.
- **Preferred During Scheduling Ignored During Execution** – Will schedule new non-matching Pods if there's no room anywhere else. Use ***weight*** to set preference order.

```
affinity:
  podAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: target
            operator:  In
            values:
            - pvc-pod
        topologyKey: "kubernetes.io/hostname"
```

- FUTURE: **Required During Scheduling Required During Execution** – Will evict non-matching Pods.

Pod Affinity/Anti-Affinity Demo

Working with Pod Affinity when Scaling Deployments and StatefulSets

Pod Topology Spread Constraints

Pod Topology Spread Constraints

- Pod topology spread constraints are similar to pod anti-affinity rules but provide more control of Pod distribution across failure topology (nodes, zones, regions).
- In addition to the *labelSelector*, Pod topology spread constraints have the following options:
- **MaxSkew** – Specifies the max difference in Pod counts between topology keys.
- **TopologyKey** – The key of node labels. The scheduler tries to place a balanced number of Pods into each topology domain.
- **WhenUnsatisfiable** – Indicates how to deal with a Pod if it doesn't satisfy the spread constraint (if exceeds **MaxSkew**):
  - **DoNotSchedule** (default) tells the scheduler not to schedule it.
  - **ScheduleAnyway** tells the scheduler to still schedule it while prioritizing nodes that minimize the skew.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  #...
spec:
  replicas: 3
  #...
  template:
  #...
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: topology.kubernetes.io/zone
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: webapp
      containers:
```

Pod Topology Spread Constraints

Observe Even Pod Distribution Across Nodes

Stateful Sets

7

StatefulSets

- Pod controller designed to manage stateful applications.
- Unlike Deployments, a ***StatefulSet*** maintain a sticky identity for each of its Pods
- Pods are created from the same spec but are not interchangeable
- Each Pod has a persistent identifier that it maintains across rescheduling.
- Pods are created in order and are assigned sequential numbers, starting with 0…N-1.

StatefulSets - continued

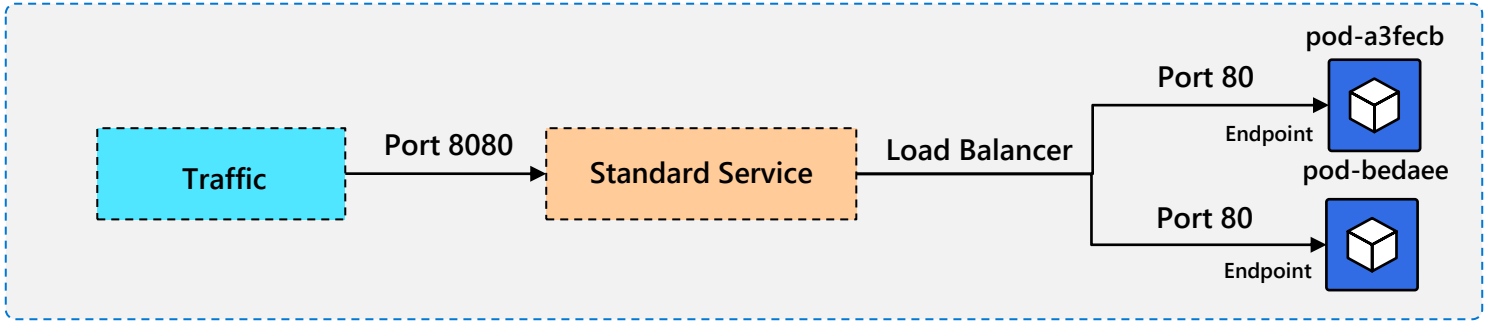
- ***StatefulSets*** define a Volume Claim Templates, which define parameters to use when dynamically creating ***PersistentVolumeClaims*** and ***PersistentVolumes*** for each Pod.
- Pod order and persistent volume claims are maintained when Pods are deleted.
- Replacement Pods are linked to the same PVCs.
- Deleting Pods in a ***StatefulSet*** (or the StatefulSet itself) will not delete the associated volumes. This is by design.

Headless Service

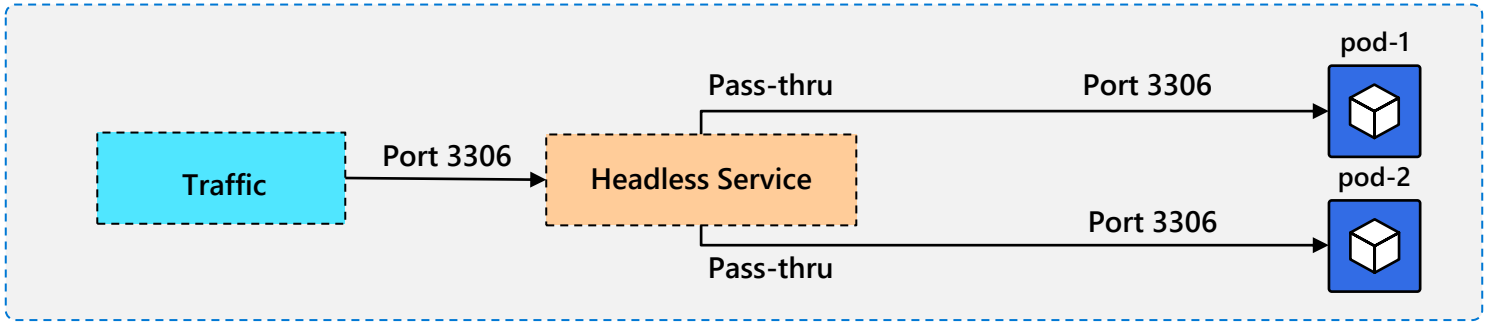
- A “headless” service (***clusterIP=None***) is required for ***StatefulSets***.
- The service is referenced by the Pod definition and allows direct access to each without load balancing.
- Creates a DNS entry for each Pod instead of just for the Service.
- Pods are accessed directly through the headless service using DNS and the FQDN address.
  - Fully Qualified domain name: *{pod}.{service}.{namespace}.svc.cluster.local*
  - Example: **mysql.connect(“mysql-02.db-service.micro.svc.cluster.local”)**

How Headless Services Work

A standard Service load balances traffic across all Pods listed as ***Endpoints***



A Headless Service *doesn't have Endpoints*. It routes traffic directly to specific Pods.



StatefulSet Demo

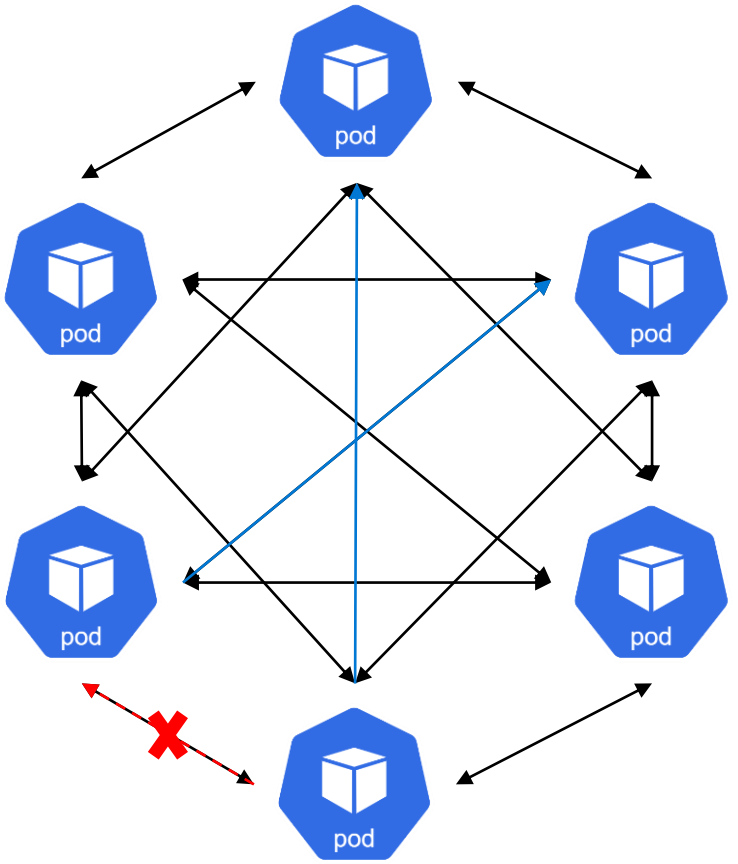
Working with StatefulSets

Network Policy

8

Network Policy

- All Pods are accessible by all other pods by defaults. They accept traffic from anyone.
- A **Network Policy** is an application-centric construct which allows you to specify how a pod is allowed to communicate with various network "entities" over the network.
- When Network Policies are in place, Pods will reject connections not allowed by those Network Policies.
- Network Policies use labels to isolate Pods by specifying how traffic flows ***in*** (Ingress) and ***out*** (egress) of the Pods.



Network Policy Configuration

- **podSelector**: Each NetworkPolicy includes a podSelector which selects the grouping of pods to which the policy applies.
- **policyTypes**: Each NetworkPolicy includes a policyTypes, which indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both.
- **ingress**: Each NetworkPolicy may include a list of allowed ingress rules. Each rule allows traffic which matches both the from and ports sections.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ingress-network-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```

Network Policy Configuration

- **egress**: Each NetworkPolicy may include a list of allowed ingress rules. Each rule allows traffic which matches both the from and ports sections.
- **ipBlock**: This selects particular IP CIDR ranges to allow as ingress sources or egress destinations. These should be cluster-external IPs, since Pod IPs are ephemeral and unpredictable.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

AKS Network Policy Options

- Network policy is a Kubernetes feature available in AKS that lets you control the traffic flow between pods.
- Azure provides two ways to implement network policy. You choose a network policy option when you create an AKS cluster. The policy option can't be changed after the cluster is created:
  - Azure's own implementation, called **Azure Network Policies**.
  - **Calico Network Policies**, an open-source network and network security solution.
- Both implementations use Linux IPTables to enforce the specified policies.
- Policies are translated into sets of allowed and disallowed IP pairs, which are then programmed as IPTable filter rules.

AKS Network Policy Comparison

This table lists differences between Azure and Calico policies and their capabilities:

Capability	Azure	Calico
Supported platforms	Linux	Linux, Windows Server 2019 (preview)
Supported networking options	Azure CNI	Azure CNI and kubenet
Additional features	None	Extended policy model consisting of Global Network Policy, Global Network Set, and Host Endpoint. For more information on using the calicoctl CLI to manage these extended features, see calicoctl user reference.
Support	**Supported by Azure support and Engineering team**	**Calico community support. For more information on additional paid support, see Project Calico support options.**
Logging	Rules added / deleted in IPTables are logged on every host under */var/log/azure-npm.log*	For more information, see Calico component logs

Microsoft

Thank you