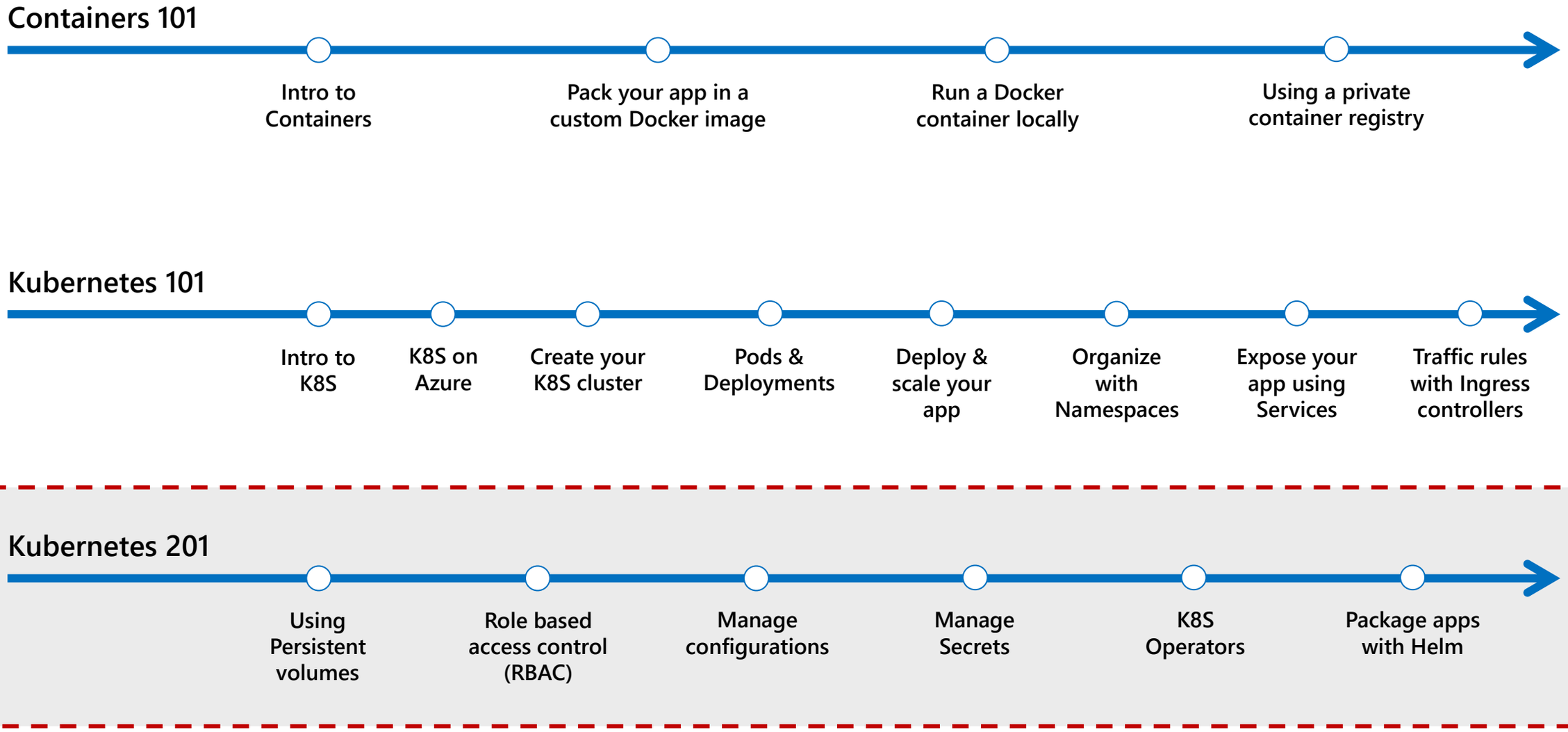
Kubernetes on Azure 201 Workshop

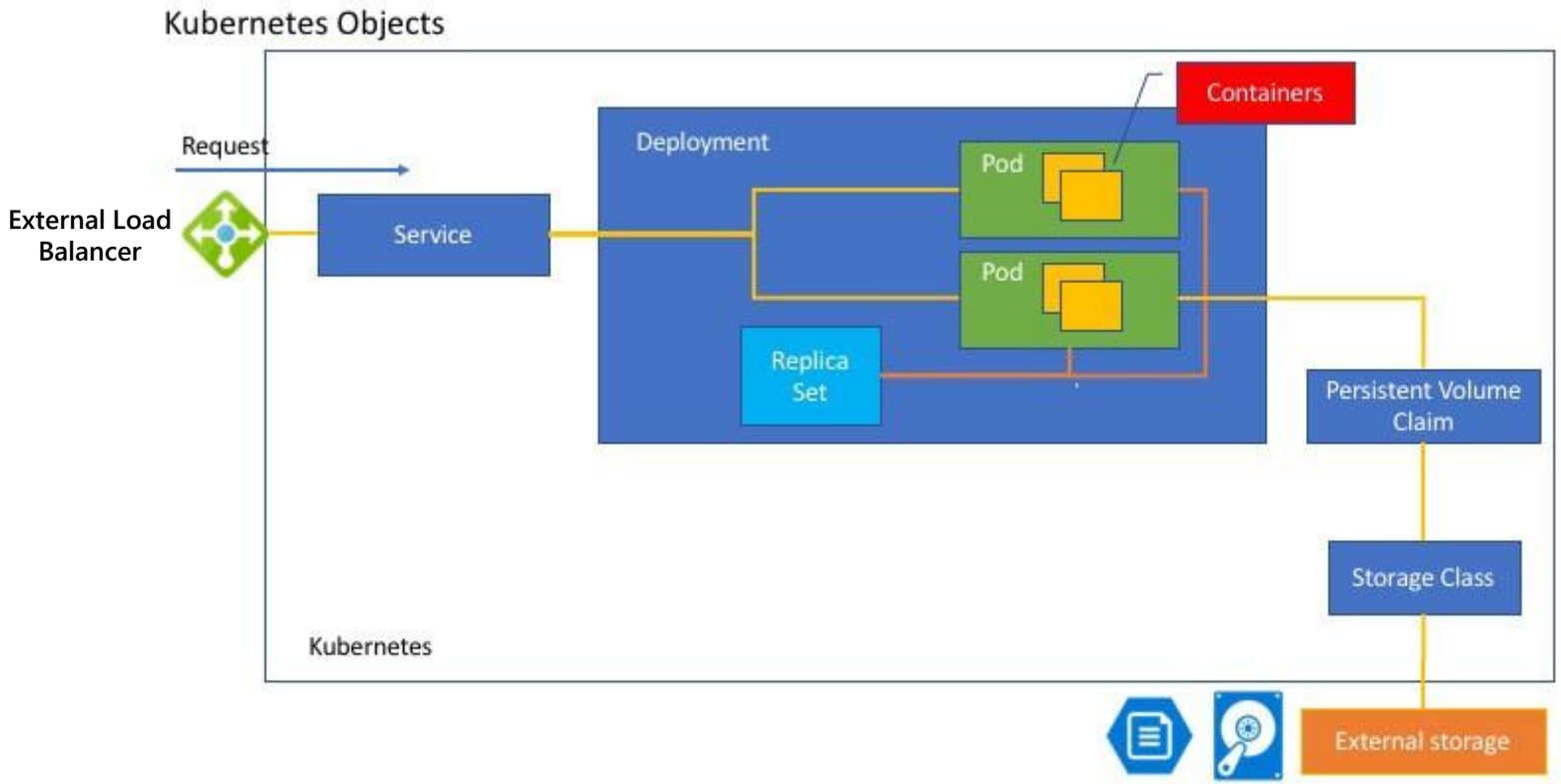
Rui Félix Pereira

2020.05.05

Containers & Kubernetes workshops



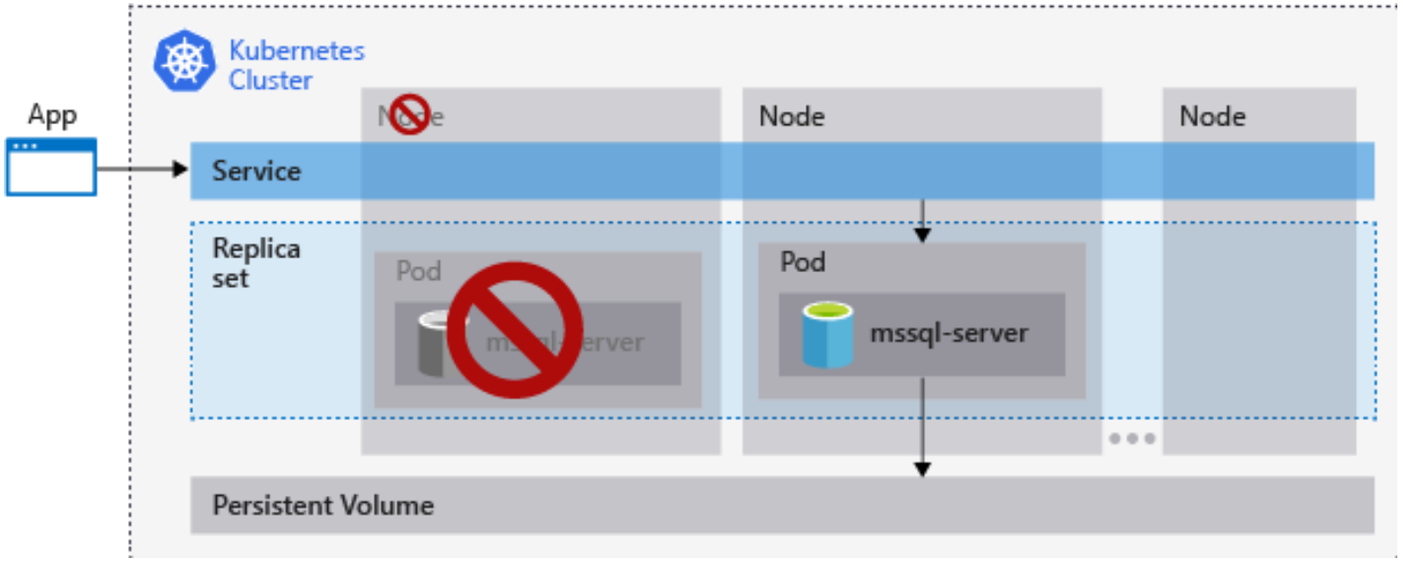
Kubernetes 101 recap



Using Persistent volumes

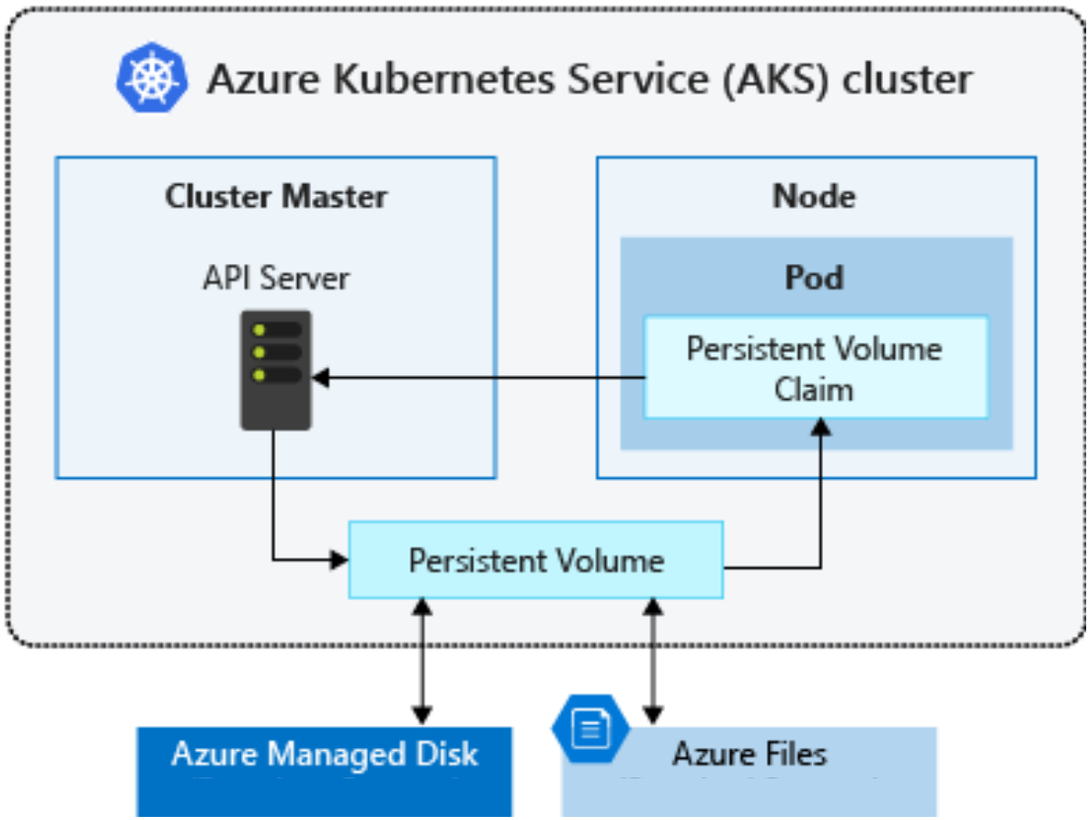
Storage and persistence

- Containers are volatile.
- In some cases, persistent storage is a need for sharing data between containers or to guarantee high availability where data must resist to container failures



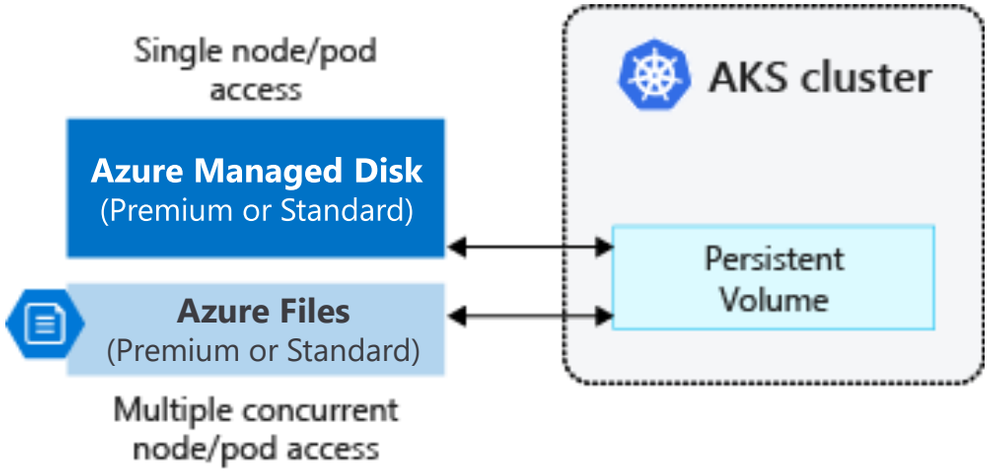
Storage and persistence on AKS

- A PersistentVolume can be **statically** created by a cluster administrator.
- Or **dynamically** created by the Kubernetes API server. If a pod is scheduled and requests storage that is not currently available, Kubernetes can create the underlying Azure Disk or Files storage and attach it to the pod. Dynamic provisioning uses a StorageClass to identify what type of Azure storage needs to be created.



Storage classes

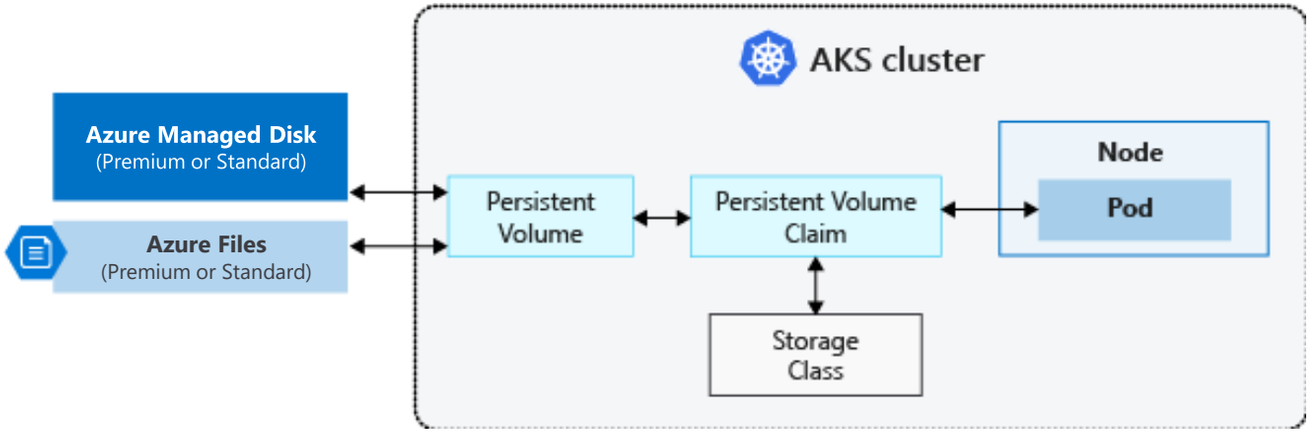
- A storage class is used to define how a unit of storage is dynamically created with a persistent volume
- Each AKS cluster includes 4 pre-created storage classes, both configured to work with Azure disks and files: Standard and Premium




```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
     name: azure-disk-standard
provisioner: kubernetes.io/azure-disk
parameters:
  storageaccounttype: Standard_LRS
  kind: Managed
```

Storage – Persistent volumes

- A persistent volume claim (PVC) is used to automatically provision storage based on a storage class




```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-data-pv-claim
  annotations:
    volume.beta.kubernetes.io/storage-class: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Storage – Using persistent volumes

- Using a persistent volume claim (PVC) in a Pod and mounting it in a mount point

```
kind: Pod
apiVersion: v1
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
       claimName: my-data-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Lab 1: Dynamic Persistent Volumes

Create and use new persistent volume

Task	With Azure

Check storage classes	`kubectl get storageclasses`
	
	`# create new storage class if needed`
	`kubectl create -f sample-storageclass.yaml`

Create persistent volume	`kubectl create -f sample-pvc.yaml`
	
	`# get persistent volumes`
	`kubectl get pv`
	`kubectl describe pv`

Use persistent volume	`kubectl apply -f test-persistent-volumes.yaml`
	
	`# get pods using persistent volume`
	`kubectl get pod task-pv-pod`
	
	`# access persistent volume (write something, destroy the pod and repeat the test again)`
	`kubectl exec -it task-pv-pod -- /bin/bash`

Static storage volumes not managed by AKS

- A PersistentVolume can be **statically** created by a cluster administrator.
- Create the Azure Disk or Azure File Share and collect the resource URI
- Configure it in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod-disk-tst
spec:
  containers:
  - image: nginx:1.15.5
    name: mypod
    volumeMounts:
      - name: azure
        mountPath: /mnt/azure
  volumes:
      - name: azure
        azureDisk:
          kind: Managed
          diskName: mydisk01
          diskURI: /subscriptions/2dd567af-d55f-4c2c-ba14-
4bd7699a59b3/resourceGroups/k8s-
volumes/providers/Microsoft.Compute/disks/mydisk01
```

Lab 2: Static Volumes

Use existing static volumes not managed by AKS

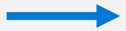
Task	With Azure
Create Azure Disk or Azure File Share and collect the Id	`# Sample Azure Disk Id` `/subscriptions/2dd567af-d55f-4c2c-ba14-4bd7699a59b3/resourceGroups/k8s-volumes/providers/Microsoft.Compute/disks/mydisk01`
Deploy Pod using the created disk	`kubectl apply -f azure-disk-pod.yaml`
Test and use the persistent volume inside the Pod	`# access persistent volume (write something, destroy the pod and repeat the test again)` `kubectl exec -it mypod-disk-tst -- /bin/bash`

Dynamic Provisioning

- Use Dynamic Provisioning whenever possible

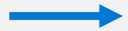
StorageClass

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
...
parameters:
  kind: Managed
  storageaccounttype: Premium_LRS
provisioner: kubernetes.io/azure-disk
reclaimPolicy: Delete
volumeBindingMode: Immediate.
```



PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: managed-premium
  resources:
    requests:
      storage: 5Gi
```



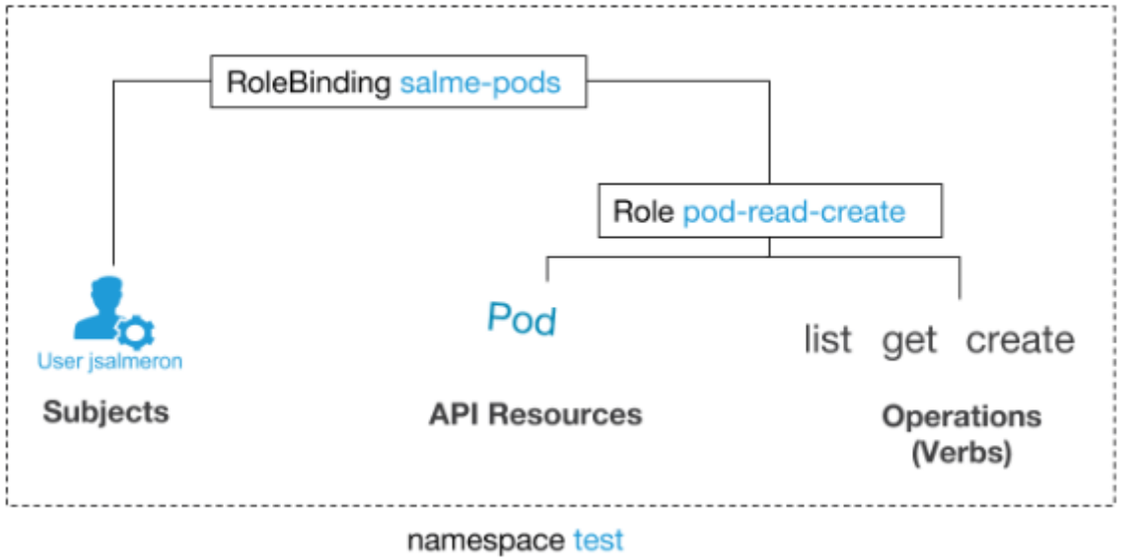
Pod

```
kind: Pod
..
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/mnt/azure"
        name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azure-managed-disk
```

Role based access control (RBAC)

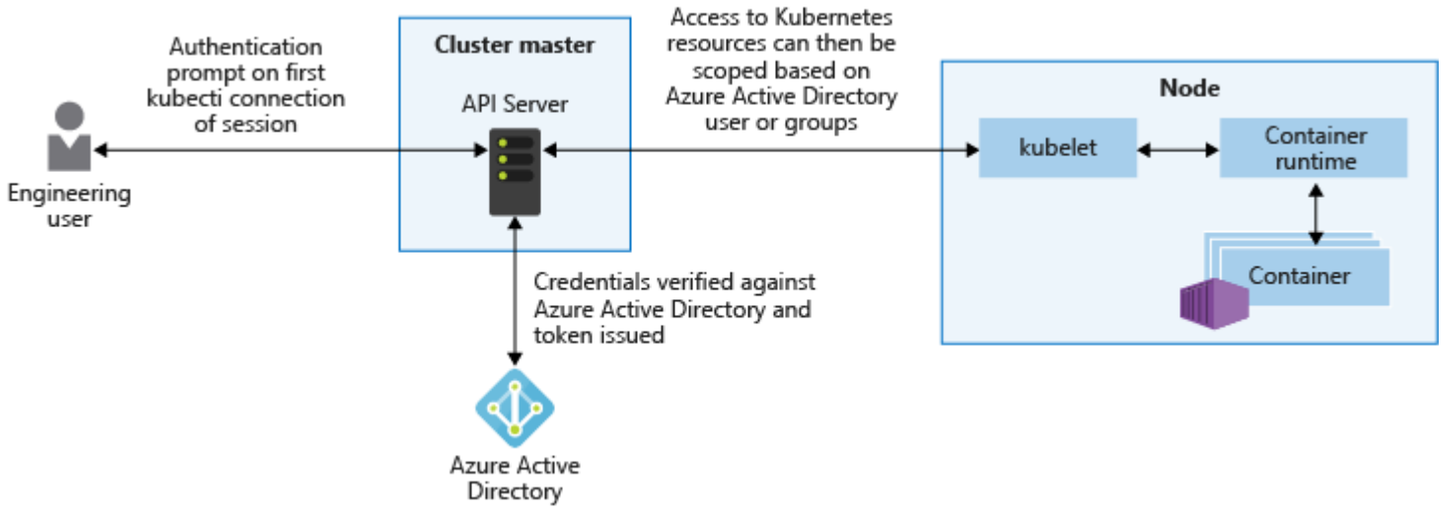
Role based access control (RBAC)

- Role grants permissions to Kubernetes objects, typically in a namespaces
- RoleBinding can be assigned to users or groups



AKS RBAC with Azure Active Directory

- AKS uses Azure Active Directory as an Identity Services for users and groups



RBAC Role

- Roles grants permissions to Kubernetes objects, typically namespaces

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev-user-full-access
  namespace: dev
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["*"]
  verbs: ["*"]
- apiGroups: ["batch"]
  resources:
  - jobs
  - cronjobs
  verbs: ["*"]
```

RBAC RoleBinding

- RoleBinding can be assigned to users or groups and optionally to namespaces

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev-user-access
  namespace: dev
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: dev-user-full-access
subjects:
- kind: Group
  namespace: dev
  name: groupObjectId
```

RBAC ClusterRoleBinding

- The are cluster roles that can be assigned to users and grouos

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: contoso-cluster-admins
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: "user@contoso.com"
```

Lab 3a: RBAC

Protect namespaces with RBAC

Task	With Azure

Create AKS with RBAC support	`# This assumes that you have an AKS created with RBAC support`
	`https://docs.microsoft.com/en-us/azure/aks/azure-ad-integration-cli`
	
	`# Get resource ID of the AKS in the Azure Active Direcotry`
	`AKS_ID=$(az aks show --resource-group myResourceGroup --name myAKSCluster --query id -o tsv)`

Create users and groups in Azure Active Directory	`# Create group for Application developers: appdev group.`
	`APPDEV_ID=$(az ad group create --display-name appdev --mail-nickname appdev --query objectId -o tsv)`
	
	`# Assign it as a user of the AKS`
	`az role assignment create --assignee $APPDEV_ID --role "Azure Kubernetes Service Cluster User Role" \`
	`  --scope $AKS_ID`
	
	`# Create and Application developer user named aksdev that is part of the appdev group.`
	`AKSDEV_ID=$(az ad user create --display-name "AKS Dev" --user-principal-name aksdev@contoso.com \`
	`  --password P@ssw0rd1 --query objectId -o tsv)`
	
	`# Assign it as a member of the appdev group`
	`az ad group member add --group appdev --member-id $AKSDEV_ID`

Lab 3b: RBAC

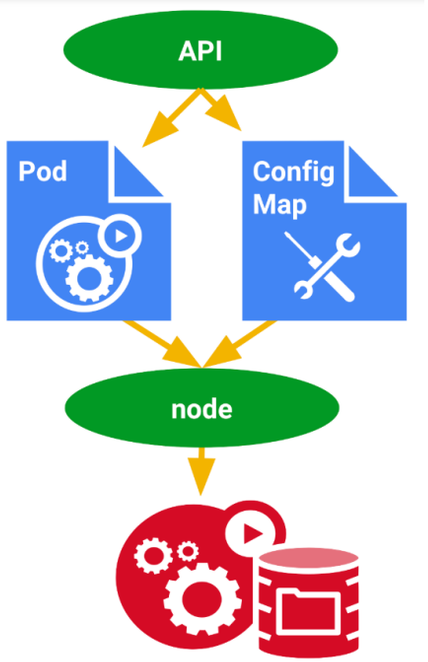
Protect namespaces with RBAC

Task	With Azure
Get admin credentials for AKs	`az aks get-credentials --resource-group myResourceGroup --name myAKSCluster -admin`
Create a new namespace	`kubectl create namespace dev`
Create a role for namespace dev	`kubectl apply -f role-dev-namespace.yaml`
Get the resource ID for the appdev group in the Azure Active Directory	`az ad group show --group appdev --query objectId -o tsv`
Create a RoleBinding for the appdev group and the previously created Role	`# On the last line of file rolebinding-dev-namespace.yaml, replace groupObjectId with the group object ID output from the previous command` `kubectl apply -f rolebinding-dev-namespace.yaml`
Test it with a user	`az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --overwrite-existing`

Manage configurations

ConfigMaps

- ConfigMaps are useful for storing and sharing non-sensitive, unencrypted configuration information.
- ConfigMaps bind configuration files, command-line arguments, environment variables, port numbers, and other configuration artifacts to your Pods' containers and system components at runtime
- Allow you to separate your configurations from your Pods and components, preventing hardcoding configuration data to Pod specifications




```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-special-config
  namespace: default
data:
  log_level: INFO
  special.type: xpto
```

ConfigMaps data usage

- ConfigMaps data can be consumed in pods in a variety of ways
- Populate the values of environment variables
- Set command-line arguments in a container
- Populate config files in a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: my-special-config
              key: special.type
restartPolicy: Never
```

Lab 4: ConfigMap

Create and use ConfigMap from Pod

Task	With Azure

Create ConfigMap	`# Create config map from poreperties files`
	`kubectl create configmap myconfigmap --from-file xpto.properties`
	
	`# Or create it form yaml`
	`kubectl create –f myconfigmap.yaml`

Get and test ConfigMap	`# Get config map value` `kubectl get configmap my-special-config -o yaml`
Use the ConfigMap from Pod	`kubectl create –f pod-using-configmap.yaml`

Manage secrets

Secrets

- Secrets are useful for storing and sharing sensitive, encrypted configuration information.
- Allow you to separate your sensitive secrets (e.g., passwords, connection strings, etc.) from your Pods, preventing hardcoding

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Secrets usage

- Secrets, like ConfigMaps data can be consumed in pods in a variety of ways
- Populate the values of environment variables
- Set command-line arguments in a container
- Populate secret files in a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

Lab 5: Secret

Create and use Secret from Pod

Task	With Azure

Create sample secrets	`echo -n 'admin' | base64`
	`YWRtaW4=`
	
	`echo -n '1f2d1e2e67df' | base64`
	`MWYyZDFlMmU2N2Rm`

Create Secret	`kubectl create –f secret.yaml`

Get and test Secret	`# Get secret value`
	`kubectl get secret mysecret -o yaml`
	
	`# Decode secret value`
	`echo 'MWYyZDFlMmU2N2Rm' | base64 –decode`

Use the Secret from Pod	`kubectl create –f pod-using-secret.yaml`

Kubernetes Operators

Custom Resources - Definition

- Kubernetes is highly extensible
- We can define Custom Resources on top of the out-of-the-box objects/resource types/APIs that come with k8s
- Defines a new API in K8S
- Seamless integration with existing APIs
- We can use kubectl

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: tomcats.tomcat.apache.org
spec:
  group: tomcat.apache.org
  names:
    kind: Tomcat
    listKind: TomcatList
    plural: tomcats
    singular: tomcat
  scope: Namespaced
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      ...
```

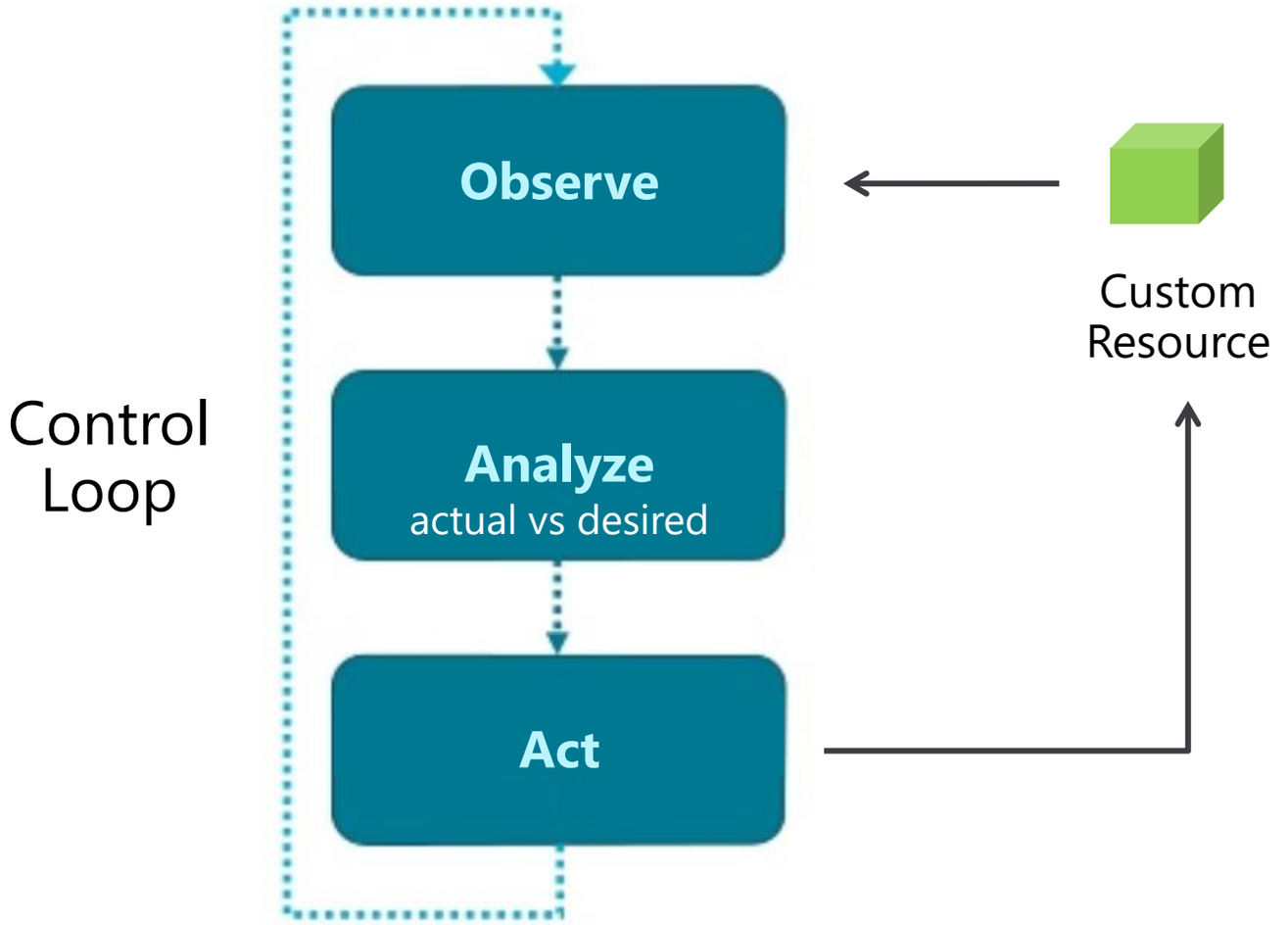
Custom Resources – Object/Instance

- We can define and configure K8S objects out of the custom resource definitions, just like any other Kubernetes object
- Using YAML and kubectl …

```
apiVersion: tomcat.apache.org/v1alpha1
kind: Tomcat
metadata:
  name: tomcat
spec:
  replicas: 2
  image: tomcat:latest
  imagePullPolicy: IfNotPresent
  webArchiveImage: ananwaresystems/webarchive:1.0
  deployDirectory: /usr/local/tomcat/webapps
```

Custom Resources & Operators

- Remember that K8S uses the concept of desired state configuration
- We can automate the control of these custom resources using Operators
- Operator Watch CR objects
- Analyze differences between Actual and Desired State
- Act on changes
- An operator is itself a Deployment/Pod
- You can create your own operators



Operators, operators, operators

- An operator is itself a Deployment/Pod
- You can create your own operators
- A lot of operators exist today to automate the management of stateful or more complex resources like a database or big application
- Extend and automate the native K8S automation capabilities, targeting specific scenarios and workloads

Good examples

```
https://github.com/coreos/etcd-operator
https://github.com/operator-framework/awesome-operators
```

https://operatorhub.io/

Lab 6: Operators

Create a custom resource and deploy an operator to manage it

Task	With Azure
Create a custom resource	`kubectl apply -f lab06-operator/tomcat-crd-definition.yaml`

Create an operator for the custom resource	`# operator runs with a service account and a specific role`
	`kubectl apply -f lab06-operator/service_account.yaml`
	`kubectl apply -f lab06-operator/role.yaml`
	`kubectl apply -f lab06-operator/role_binding.yaml`
	
	`# deploy the operator`
	`kubectl apply -f lab06-operator/operator.yaml`

Create an instance of the custom resource type	`# Create a Tomcat cluster` `kubectl apply -f lab06-operator/tomcat-cr-instance.yaml`

Check what is deployed and running	`kubectl get tomcat`
	`kubectl get pod`
	`kubectl get svc`

Package apps with Helm

Helm

- Package manager for Kubernetes
- Helm helps you manage Kubernetes applications
- Avoiding K8S ymal templates copy & paste all the time
- Helm Charts help you define, install, and upgrade even the most complex Kubernetes application
- Charts are easy to create, version, share, and publish — so start using Helm and stop the copy-and-paste.

https://github.com/helm/charts

Lab 7: Helm

Create and use a new Helm chart

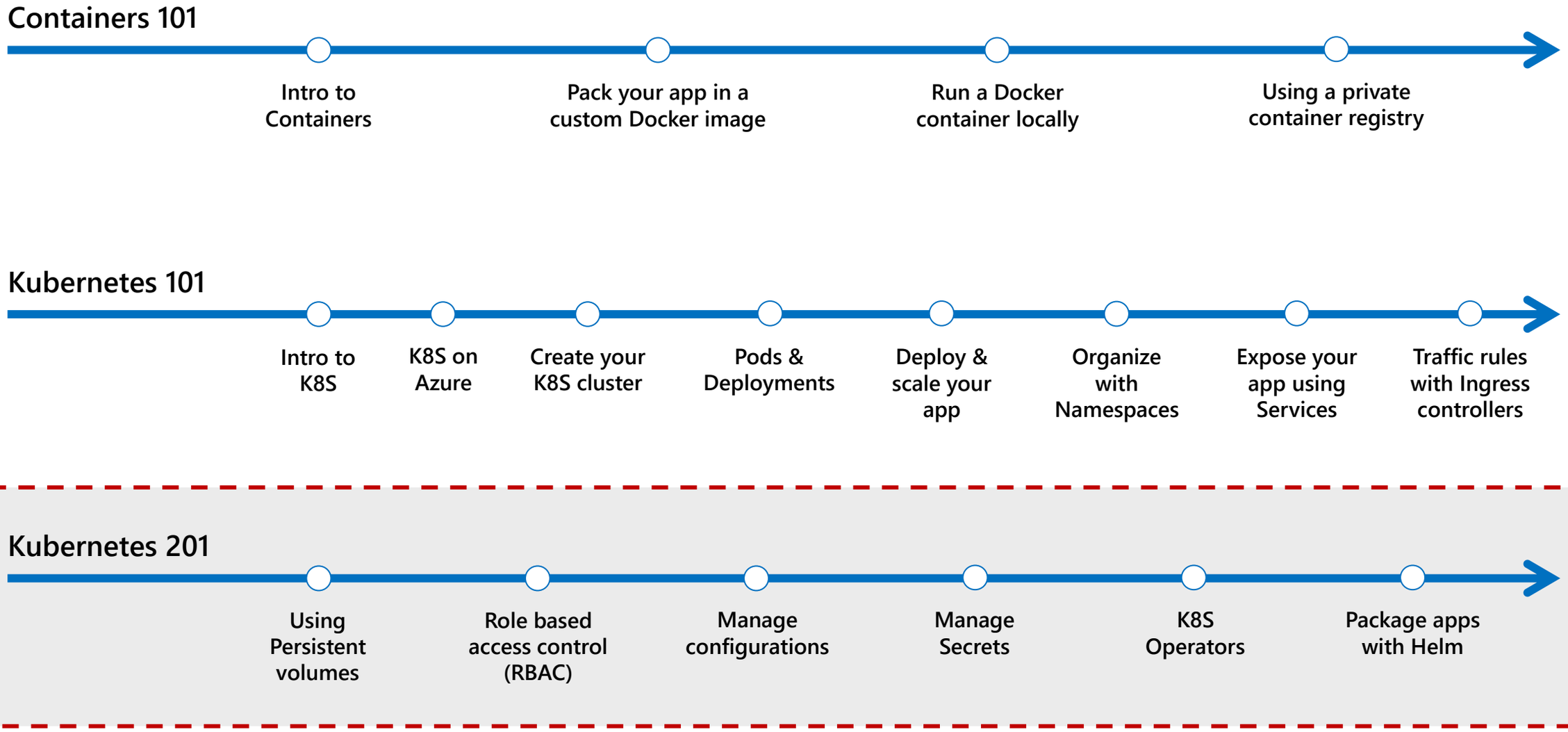
Task	With Azure
Create a new Helm chart/package	`helm create webfrontend`

Edit your chart values	`# Make the following updates to webfrontend/values.yaml:`
	`- Change the image of your container from image.repository to microsoft/azure-vote-front:v1`
	`- Change the service type to expose your application from service.type to LoadBalancer`
	
	`## Edit your chart version`
	`Update appVersion to v1 in webfrontend/Chart.yaml`

Install your chart	`helm install webfrontend webfrontend/`

Validate it!!!	`# Validate that your chart exists`
	`helm list`
	
	`# Validate deployed objects`
	`kubectl get all`

Containers & Kubernetes workshops



Thank you