Ingress versus Load Balancer

A LoadBalancer service is the standard way to expose a service to the internet. On AKS, this will spin up a Network Load Balancer that will give you a single IP address that will forward all traffic to your service.

The big downside is that each service you expose with a LoadBalancer will get its own IP address - that means all your microservices running in your AKS cluster will get its own IP address which translates to its own DNS (because no one calls a service using ip correct?). This can get costly and soon become hard to manage.

Enter ingress...

Ingress is actually NOT a type of service. Instead, it sits in front of multiple services and act as a “smart router” or entrypoint into your cluster. A typical will let you do both path based and subdomain based routing to backend services. For example, you can send everything on foo.yourdomain.com to the foo service, and everything under the yourdomain.com/bar/ path to the bar service.

Ingress is the most useful if you want to expose multiple services under the same IP address, and these services all use the same L7 protocol (typically HTTP). You only pay for one load balancer and one IP, and you get other features such as SSL termination for example.

Demo Load Balancer (NGINX)

```
  wget
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/raw/33813fdeeddad70cff7e0039b32d729e3d2aa0b6/k8sws-aks-lb.yaml -O k8sws-aks-lb.yaml

    # Update the templated value
    sed -i "s|<index>|1|g" k8sws-aks-lb.yaml
    cat k8sws-aks-lb.yaml
    kubectl apply -f k8sws-aks-lb.yaml

    # repeat for 2
    kubectl delete -f k8sws-aks-lb.yaml
```

Demo Ingress (NGINX)

```
  INGRESS_IP=$(kubectl get service/nginx-ingress-ingress-nginx-controller  -n
ingress -o json | jq -r .status.loadBalancer.ingress[].ip)

  wget
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/d9df361cebcc9e43b585db2396d469ab69add025/k8sws-aks-ingress.yaml -O k8sws-aks-
ingress.yaml
    # Update the templated value
    sed -i "s|<index>|1|g" k8sws-aks-ingress.yaml
    sed -i "s|<ingress_ip>|${INGRESS_IP}|g" k8sws-aks-ingress.yaml
    kubectl apply -f k8sws-aks-ingress.yaml


    wget
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/d9df361cebcc9e43b585db2396d469ab69add025/k8sws-aks-ingress.yaml -O k8sws-aks-
ingress.yaml
    # Update the templated value
    sed -i "s|<index>|2|g" k8sws-aks-ingress.yaml
    sed -i "s|<ingress_ip>|${INGRESS_IP}|g" k8sws-aks-ingress.yaml
    kubectl apply -f k8sws-aks-ingress.yaml

    # http://k8sws-aks-ingress.20.75.3.198.nip.io/v1/chaincallc


    kubectl delete -f k8sws-aks-ingress.yaml
```

State Persistence

Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. We have already seen examples of using secrets and configmaps as volumes.

Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod. Consequently, a volume outlives any containers that run within the pod, and data is preserved across container restarts. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes.

An example of ephemeral volume is `emptyDir`

An emptyDir volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. As the name says, the emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

> A container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes.

Persistent Volume (PV) and Persistent Volume Claims (PVC)

Ref : https://kubernetes.io/docs/concepts/storage/persistent-volumes/

In kubernetes there is a distinction between managing compute instances versus managing states - these are different problems. PV and PVC provide an abstraction from details of how a storage is provided and how it is consumed (i dont care if its azure file SSD or HDD or even azure disk. I request and i get). Think of these as you would be callign an api and not caring what the api does in the backend.

PV is a resource in the cluster similar to a node and has a lifecycle independent of any individual Pod that uses the PV.

PVC is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany, see AccessModes).

Finally there is Storage Classes - the motive behind storage classes is that users need PV with varying properties (SSD or HDD for performance). As an admin you need to be able to offer a variety of PVs that differ in more ways than size and access modes without exposing the details to how these are implemented. This is where storage classes comes in.

Provisioning

Static A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses: the PVC must request a storage class and the administrator must have created and configured that class for dynamic provisioning to occur

For the most part - you want dynamic and that is what we will focus on.

In AKS, four initial StorageClasses are created for cluster using the in-tree storage plugins:

https://docs.microsoft.com/en-us/azure/aks/concepts-storage

STORAGE CLASSES

- **default** Uses Azure StandardSSD storage to create a Managed Disk. The reclaim policy ensures that the underlying Azure Disk is deleted when the persistent volume that used it is deleted.
- **managed-premium** Uses Azure Premium storage to create a Managed Disk. The reclaim policy again ensures that the underlying Azure Disk is deleted when the persistent volume that used it is deleted.
- **azurefile** Uses Azure Standard storage to create an Azure File Share. The reclaim policy ensures that the underlying Azure File Share is deleted when the persistent volume that used it is deleted.

- **azurefile-premium** Uses Azure Premium storage to create an Azure File Share. The reclaim policy ensures that the underlying Azure File Share is deleted when the persistent volume that used it is deleted.

```
  # Default AKS storage class
  kubectl get sc

  # detail - NOTICE provisioner, skuname and volume expansion
  kubectl get sc/azurefile -o yaml
  kubectl get sc/azurefile-premium -o yaml
```

Now even though the storage classes are created, its not yet functioning. Remember the steps

1. Create storage classes - AKS takes care of it
2. Create PVC that references the Storage Class - **this is pending**
3. K8s uses SC provisioner to provision a PV - this is automatic

So we are missing step 2. Lets create a PVC

```
# Notice the status of the PVC
kubectl get pv,pvc,sc

echo '
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-pvc-1
spec:
  accessModes:
    - ReadWriteMany
  # NOTICE THE CLASS NAME
  storageClassName: azurefile
  resources:
    requests:
      storage: 1Gi
' | kubectl apply -f -

#create a POD using the PVC
echo '
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
    resources:
      requests:
        cpu: 100m
```

```
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      volumeMounts:
      - mountPath: "/mnt/azure"
        name: volume
    volumes:
      - name: volume
        persistentVolumeClaim:
          claimName: my-pvc-1
  ' | kubectl apply -f -
```

> Runnign the above script will create a storage account in the MC_* resource group with a fileshare. The first time its created, it will be 1gi.

Its possible to use the same PVC for different pods (thinking sharing a directory). Lets try that

```
#create a POD using the SAME PVC
echo '
kind: Pod
apiVersion: v1
metadata:
  name: mypod2
spec:
  containers:
  - name: mypod2
    image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
      limits:
        cpu: 250m
        memory: 256Mi
    volumeMounts:
    - mountPath: "/mnt/azure"
      name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: my-pvc-1
' | kubectl apply -f -

# exec into the first pod and create a file
kubectl exec -it mypod -- sh
  cd mnt/azure
  touch hello.txt
  exit
```

```
kubectl exec -it mypod2 -- sh
  ls /mnt/azure/
```

Lets delete the two pods and PVC

```
kubectl delete pod/mypod
kubectl delete pod/mypod2

# THE file created previously still exists
kubectl get pv,pvc

# let delete the PVC - NOTICE IT REMOVES THE CREATED PV
# AND ALSO DELETES the share from Azure - This is the RECLAIM policy
kubectl delete persistentvolumeclaim/my-pvc-1
```

But often times, we need to have volumes that we dont want files deleted from even after the PVC has been deleted. Lets create a custom SC,PVC and PV for this

```
#  kubectl get sc azurefile -o yaml
echo '
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
kind: StorageClass
metadata:
  labels:
    addonmanager.kubernetes.io/mode: EnsureExists
    kubernetes.io/cluster-service: "true"
  name: azurefile-retain
  resourceVersion: "83"
mountOptions:
- mfsymlinks
- actimeo=30
parameters:
  skuName: Standard_LRS
provisioner: kubernetes.io/azure-file
reclaimPolicy: Retain # we want to retain
volumeBindingMode: Immediate
' | kubectl apply -f -

echo '
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-pvc-1
spec:
  accessModes:
    - ReadWriteMany
  # NOTICE THE CLASS NAME
  storageClassName: azurefile-retain
  resources:
    requests:
      storage: 1Gi
' | kubectl apply -f -

# VERIFY
kubectl get sc,pvc,pv

echo '
kind: Pod
apiVersion: v1
metadata:
  name: mypod2
spec:
  containers:
  - name: mypod2
    image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
      limits:
        cpu: 250m
        memory: 256Mi
    volumeMounts:
    - mountPath: "/mnt/azure"
      name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: my-pvc-1
' | kubectl apply -f -

# upload a file or create a file
kubectl delete pod/mypod2
kubectl delete persistentvolumeclaim/my-pvc-1

# AZURE FILE SHARE STILL EXISTS
```

HPA

The Horizontal Pod Autoscaler automatically scales the number of Pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization or memory.

Reference (with figure): https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Demo

In this demo we will deploy a deployment along with an ingress service and trigger kubernetes for hpa.

```
    # Download the file to replace the ingress name
     wget
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/773f8b60769525f28b801ef2da2f903e9b197d43/k8sws-aks-hpa.yaml -O k8sws-aks-
hpa.yaml

    #get the ingress ip
    INGRESS_IP=$(kubectl -n ingress get svc/nginx-ingress-ingress-nginx-controller
-o json | jq -r .status.loadBalancer.ingress[].ip)

    # Update the ingress name
    sed -i "s|{INGRESS_IP}|${INGRESS_IP}|g" k8sws-aks-hpa.yaml
    kubectl apply -f k8sws-aks-hpa.yaml

    # MONITOR - NEW WINDOW
    while($true)
    {
    kubectl get pods,hpa
    Start-Sleep -Seconds 3

    }

    # start sending request - NEW WINDOW
    while($true)
    {
    Invoke-WebRequest -URI http://k8sws-svc-
c.20.69.195.149.nip.io/admin/memorySpike
    Start-Sleep -Seconds 3
    }



     # Delete
     kubectl delete -f k8sws-aks-hpa.yaml
```

Taints and Tolerations

Ref https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

> Taints and Tolerations are from a node's perspective.

- Taints allow a node to repel a set of pods.
- Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.
- Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints

`kubectl taint nodes node-name key=value:taint-effect`

`taint-effect`: What happens to Pods that do not tolerate this taint. There are three types of taint-effects.

- NoSchedule : Pods will not be scheduled on the nodes
- PreferNoSchedule: System will try to avoid scheduling pods on the nodes but this is not guaranteed.
- NoExecute: New Pods will not be scheduled on the pods if they dont have tolerations and existing pods (with no tolerations) will also be evicted. An example here can be when you may want to bring down a node, so you taint it and allow the scheduler to move the pods to another server before bringing the server down. Example - `kubectl taint nodes aks-agentpool-35064155-vmss000002 patching=value:NoExecute`

Demo

```
#create a deployment
kubectl apply -f
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/8039987b3c8a07142f989f06f73b750a81091526/k8sws-core-deployments.yaml
#scale the deployment to be the number of nodes
 kubectl scale deploy k8sws-core-deployments --replicas 3
# notice the pods are evenly distributed
 kubectl get pods -o wide
# taint a node

#kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes aks-agentpool-35064155-vmss000000 app=critical:NoSchedule
# delete the pods
kubectl delete pods -l app=k8sws-core-deployments
#get pods again - NOTICE that the pods are not scheduled on the tainted node
kubectl get pods -o wide
#create another deployment - this time with tolerations
kubectl apply -f
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/830f2ae926397079c7ccf0a60497496588f441be/k8sws-core-taint-toleration.yaml
#scale
kubectl scale deploy k8sws-core-taint-toleration --replicas 3

# remove a taint (the "-" at the end)
#kubectl taint nodes aks-agentpool-35064155-vmss000000 app=critical:NoSchedule-
```

With the last deployment, we saw that the pods were distributed across the nodes. But what if - we wanted our pods to only run on specific nodes. Taints and Tolerations are fine from a node's perspective i.e i can only take in pods with certain tolerations but what about pods having the same rights - i.e. i want to be scheduled only on certain nodes. I mean, hey you need two to tango right? This is where Node Selectors and Affinity/anti-affinity come into play.

Node Selectors

`nodeSelector` provides a very simple way to constrain pods to nodes with particular labels -- **this is important. NodeSelectors are based off labels not taints**

```
  #create a label for the node
  kubectl get nodes
  kubectl label nodes aks-agentpool-35064155-vmss000000 app=critical

  #create a deployment
  kubectl apply -f
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/18884092e7206e99514936795c4956579de8c15e/k8sws-core-nodeselector.yaml
  #scale the deployment to be the number of nodes
  kubectl scale deploy k8sws-core-nodeselector --replicas 5
  # notice how the pods are in ONE node only
  kubectl get pods -o wide
```

Affinity and anti-affinity

Ref https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity

Key enhancements versus nodeselector

- The affinity/anti-affinity language is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation;
- you 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;
- you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located

Node affinity

similar to nodeSelector -- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node

> From K8s Documentation: There are currently two types of node affinity, called requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that must be met for a pod to be scheduled onto a node (similar to nodeSelector but using a more expressive syntax), while the latter specifies preferences that the scheduler will try to enforce but will not guarantee. The "IgnoredDuringExecution" part of the names means that, similar to how nodeSelector works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod continues to run on the node. In the future we plan to offer requiredDuringSchedulingRequiredDuringExecution which will be identical to requiredDuringSchedulingIgnoredDuringExecution except that it will evict pods from nodes that cease to satisfy the pods' node affinity requirements. Thus an example of requiredDuringSchedulingIgnoredDuringExecution would be "only run the pod on nodes with Intel CPUs" and an example preferredDuringSchedulingIgnoredDuringExecution would be "try to run this set of pods in failure zone XYZ, but if it's not possible, then allow some to run elsewhere".

```
  #create a label for the node
  kubectl get nodes
  kubectl label nodes aks-agentpool-35064155-vmss000000 app=critical

  #create a deployment - requiredDuringSchedulingIgnoredDuringExecution
  kubectl apply -f
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/0919e8974c19ec3044c3e4ad5fd0b0cc801a491d/k8sws-core-node-affinity-1.yaml
  # get pods
  kubectl get pods -o wide

  # delete that deployment
   kubectl delete -f
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/0919e8974c19ec3044c3e4ad5fd0b0cc801a491d/k8sws-core-node-affinity-1.yaml

  #create another deployment - preferredDuringSchedulingIgnoredDuringExecution
  # the difference is that it may not match a node and will still be scheduled-
check the values
  kubectl apply -f
https://gist.githubusercontent.com/nishantnepal/fcae1e0c599f75bdb977f555e54cd918/r
aw/8a4c43373650b62c37d46de34faa499d47a777fb/k8sws-core-node-affinity-2.yaml
```

Pod anti-affinity

Typical example is an ingress controller where you want pods of an ingress controller spread across all nodes/zones instead of being in a single zone.

References https://blog.nillsf.com/index.php/2020/12/03/how-to-run-your-own-admission-controller-on-kubernetes/