AKS-Kubernetes-Lab

Securing your container workloads in Kubernetes

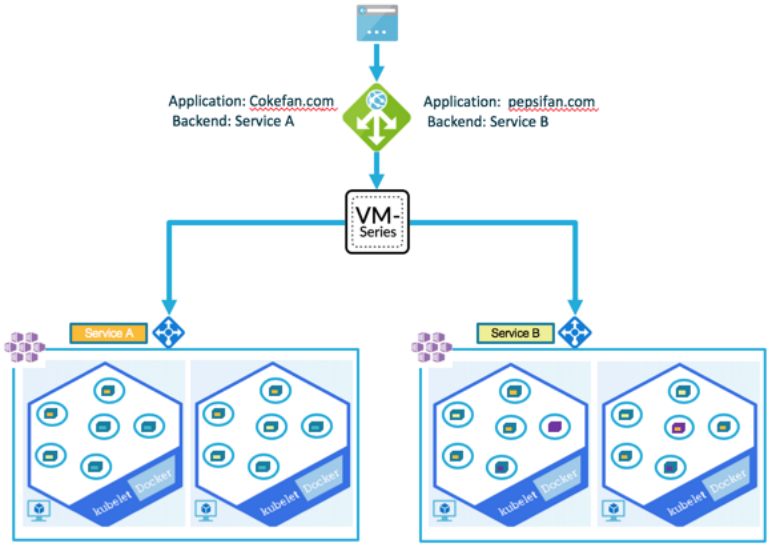
Table of Contents

About the Azure Kubernetes Service Terraform Template

Azure Kubernetes Service (AKS) Terraform Templates are files that can deploy, configure, and launch AZURE resources such as Resource Groups, VNETS, subnets, security groups, application gateways, route tables, Kubernetes clusters, and more. These templates are used for ease of deployment and are key to any cloud deployment model.

For more information on Templates refer to Google's documentation
https://docs.microsoft.com/en-us/azure/terraform/

This document will walk through the setup and deployment of a Terraform template that deploys the AKS infrastructure and a Palo Alto Networks VM-Series firewall that provides advanced protection for the Kubernetes cluster North/South traffic. During the deployment the template will create two Azure resource groups. One that has the infrastructure including the bootstrapped VM-Series Firewall and another with the k8s cluster resources. The guide also walks through the deployment of two separate applications. Each 2-tier application consists of database and web pods. After completing this guide, the following infrastructure will be instantiated:



Support Policy

This template is released under an as-is, best effort, support policy. These scripts should be seen as community supported and Palo Alto Networks will contribute our expertise as and when possible. We do not provide technical support or help in using or troubleshooting the components of the project through our normal support options such as Palo Alto Networks support teams, or ASC (Authorized Support Centers) partners and backline support options. The underlying product used (the VM-Series firewall) by the scripts or templates are still supported, but the support is only for the product functionality and not for help in deploying or using the template or script itself.

Instances Used

When deploying this Terraform template the following machine types are used:

Instance	Machine Type	QTY
PayGo Bundle 1 – VM-Series Firewall	Standard_D3_v2	1
Kubernetes Ubuntu Cluster Nodes	Standard_D3_v2	2
Internal Load Balancer		1
Application Gateway		1

Note: There are Azure costs associated with each machine type launched, please refer to the Microsoft instance pricing page **https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/**

Prerequisites

Here are the prerequisites required to successfully launch this template:

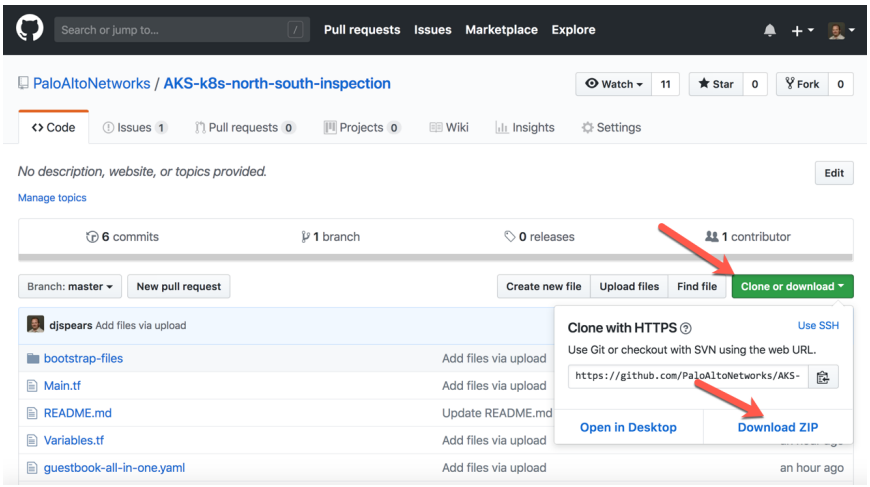
- Terraform application - Instructions on the installation can be found here: https://www.terraform.io/intro/getting-started/install.html
- Azure account- Account creation instructions can be found here: https://azure.microsoft.com/en-us/resources/videos/sign-up-for-microsoft-azure/
- Azure command-line tool – Instructions for doing this can be found here: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest
- Kubernetes command-line tool – Instructions for doing this can be found here: https://kubernetes.io/docs/tasks/tools/install-kubectl/

Download GitHub files

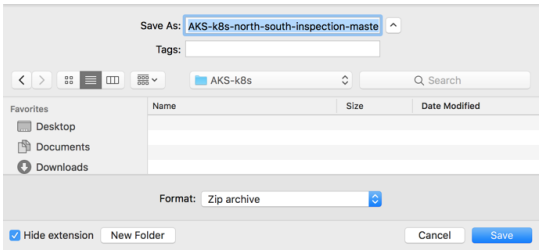
In this activity, you will:

Download a zip copy of the GitHub files used for this lab

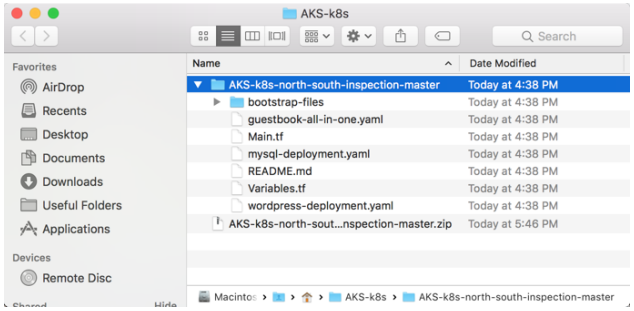
During this lab, the Terraform templates and Kubernetes (k8s) command will be executed from a local computer. This lab requires some customization of the terraform files. To download the files from GitHub, click on the Clone or download drop down and select Download ZIP.



Save the zip file to a new directory. This directory will be used to deploy the Terraform template and will automatically keep the Terraform state files so the deployment can be managed in the future:



Unzip the files:



Azure Service Principal Creation

In this activity, you will:

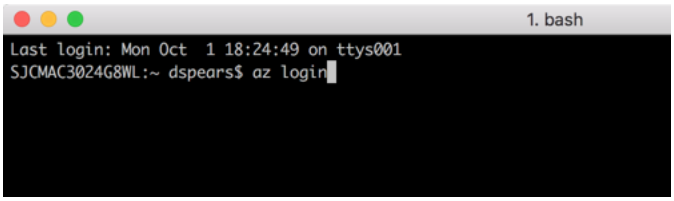
Authenticate to an Azure subscription via the Azure command line tool

Create a Service Principal with the appropriate RBAC to deploy a kubernetes (k8s) cluster

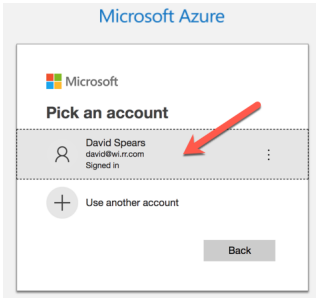
Update the Terraform Variables.tf file with the Service Principal information needed to execute

Microsoft has documented the steps to create a service principal that can be used to deploy a k8s cluster. That document can be found here: https://docs.microsoft.com/en-us/azure/container-service/kubernetes/container-service-kubernetes-service-principal

This guide assumes that the perquisites have been completed and the Azure command line tool has been installed. Open a terminal window and type the command **az login** to authenticate the command line tool to the appropriate subscription:



Next a browser window should open that will give the option to select the Azure account associate with the subscription that will get the deployment:

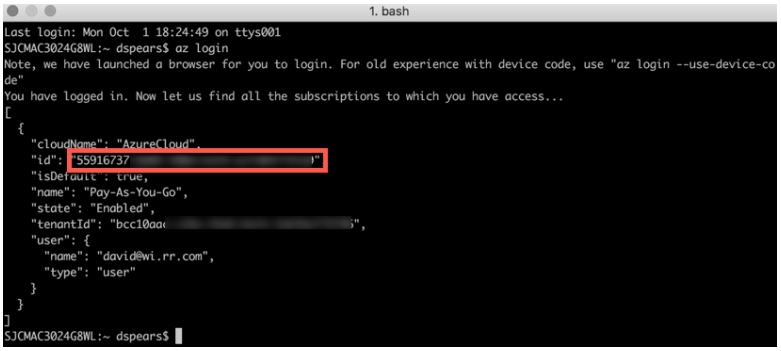


Once the account has been selected, the following message will appear:

You have logged into Microsoft Azure!

You can close this window, or we will redirect you to the Azure CLI documents in 10 seconds.

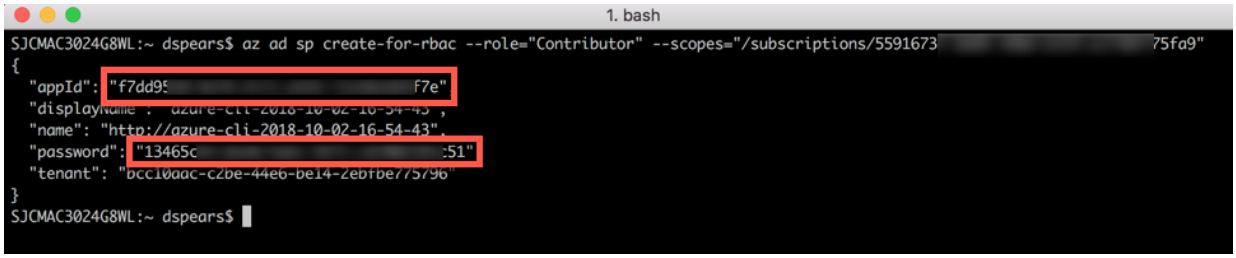
Check the terminal window. There should be confirmation that the login process was a success:



Copy the "id" from the output. This is the subscription id for the service principal. To be able to deploy a k8s cluster in Azure the service principal must have the "contributor" role. Use the following command to create the service principal:

$ az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/<id>"

where "id" is the subscription id copied from the last step:



The Terraform deployment files consist of a main, variables, and output files. The Variables.tf file contains information that is easily modified and commonly changed for various situations. The variables in the Variables.tf file are used by the Main.tf file during deployment. Deploying this Terraform template in Azure does require modification of the Variable.tf file to include deployment-specific information.

Copy the “appId” and “password” fields from the service principal creation output. These are needed for the terraform script and need to be added to the Variables.tf file. Open an editor of your choice and update these fields and save the file:

```
// PROJECT Variables
variable "client_id" {
    default = "<appId>"
}
variable "client_secret" {
    default = "<password>"
}

variable "agent_count" {
    default = 2
}

variable "ssh_public_key" {
    default = "<path to public ssh key>"
}

variable "dns_prefix" {
    default = "k8s-AZURE-HOW"
}

variable cluster_name {
    default = "k8s-Cluster-MGMT"
}

variable resource_group_name {
    default = "k8s-RG"
}

variable location {
```

```
// PROJECT Variables
variable "client_id" {
    default = "f7dd95                    f7e"
}
variable "client_secret" {
    default = "13465c                    :51"
}

variable "agent_count" {
    default = 2
}

variable "ssh_public_key" {
    default = "/AKS/AKS-PANWFW-k8s/djs-gcp-keyb.pub"
}

variable "dns_prefix" {
    default = "k8s-AZURE-HOW"
}

variable cluster_name {
    default = "k8s-Cluster-MGMT"
}

variable resource_group_name {
    default = "k8s-RG"
}

variable location {
    default = "Central US"
}
```

Bootstrap storage account creation

In this activity, you will:

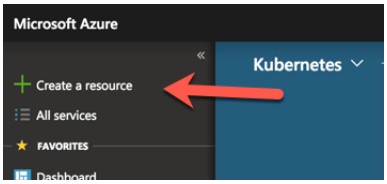
Create an Azure Resource Group and deploy a storage account

Create a file share with the folder structure needed to bootstrap the VM-Series Firewall

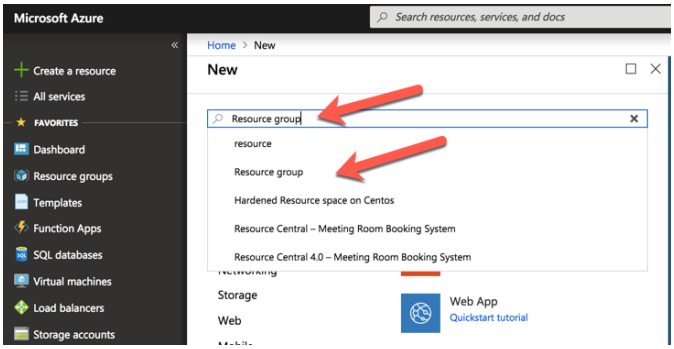
Copy the files to the Azure file share needed for bootstrapping

Update the Terraform Variables.tf file with the Azure storage access key that will allow the VM-Series Firewall to bootstrap

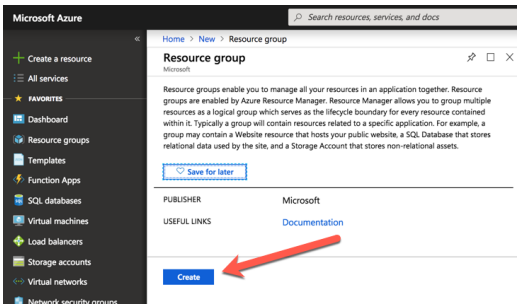
The terraform template is going to bootstrap the initial VM-Series firewall configuration. To accomplish this an Azure storage account will be created with the appropriate files. To start, open the Azure Portal and create a new resource group. Click on the “+ Create a resource” link:



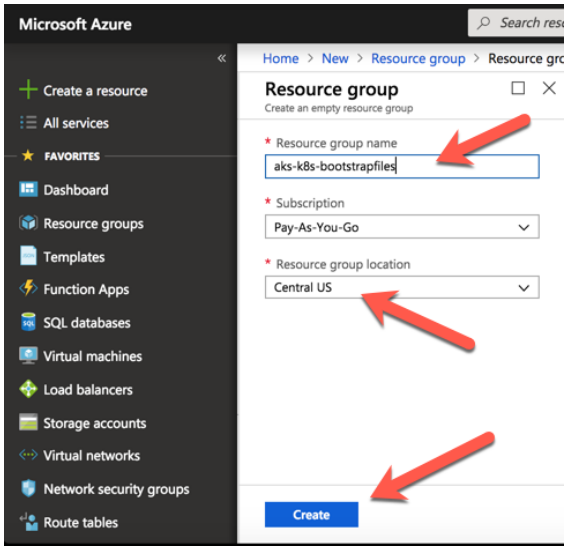
Next enter “Resource group” in the search and select Resource group:



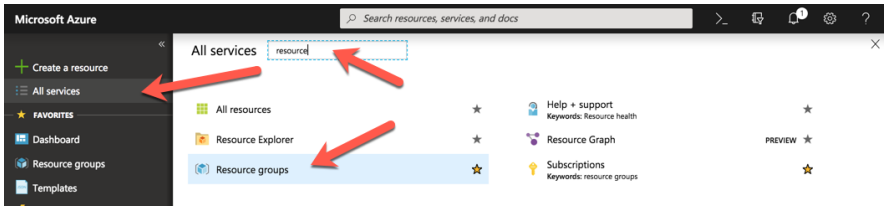
Next select “Create” to create:



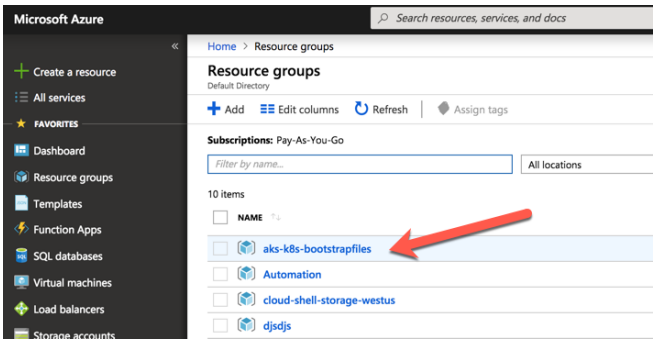
In the next window, create a resource group name and select the Resource group location. It is recommended for this lab to use the same location that the terraform script deploys in. The default setting is Central US. Click “Create” to create the Resource group.



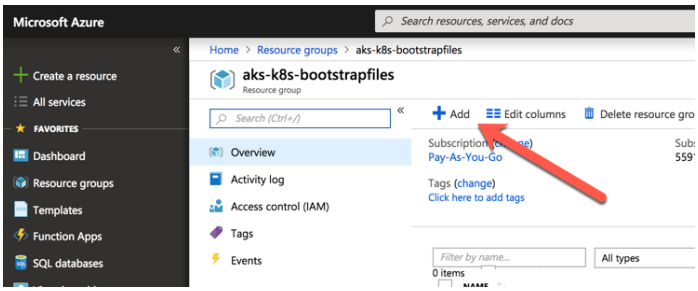
Navigate to the new Resource group. If a favorite is not available, click the “All Services” option on the left Nav and type “resource” in the All services search window. Click on Resource groups to open all the resources.



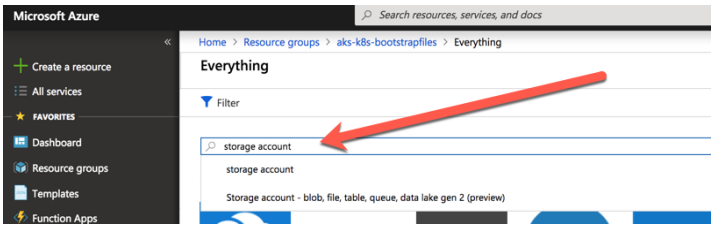
Now click the newly created Resource group:



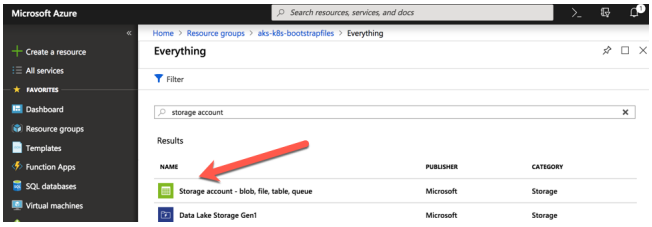
Once in the resource group the next step is to create a storage account. Click on the plus sign to add a resource in the resource group:



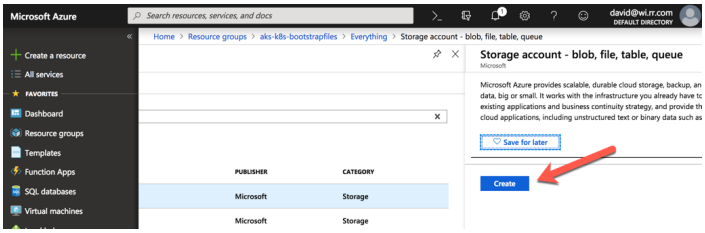
Type storage account in the search field:



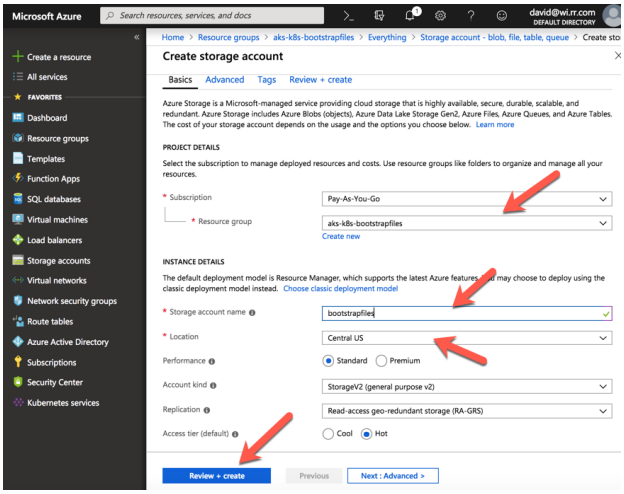
Select the Storage account published by Microsoft:



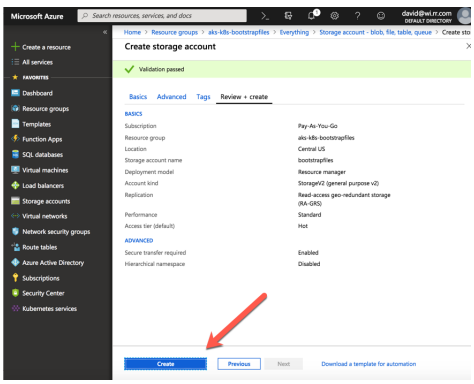
Next click “Create”:



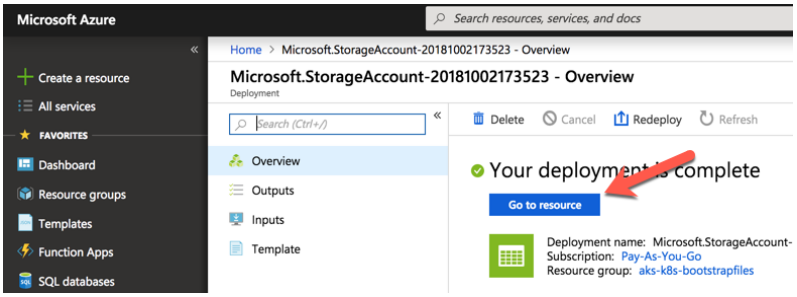
Make sure the Resource group is correct. Enter a Storage account name and select the same location as the rest of the deployment. Finally click “Review and create”



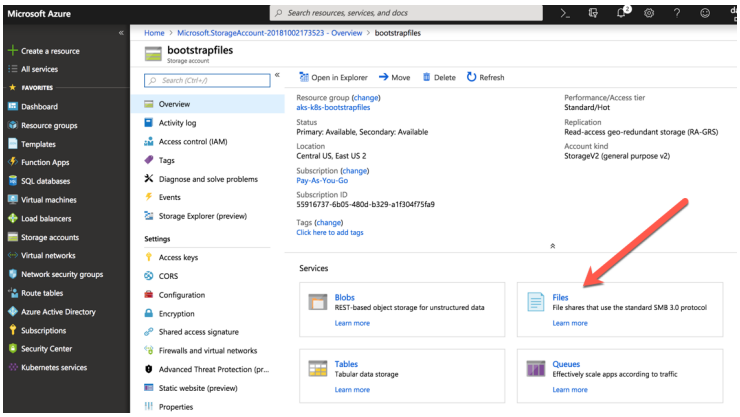
Once the validation is complete, select Create:



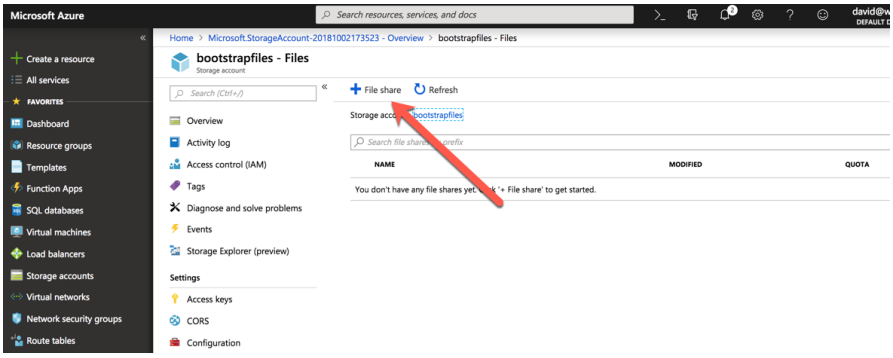
After the deployment is complete, click on the go to resource button:



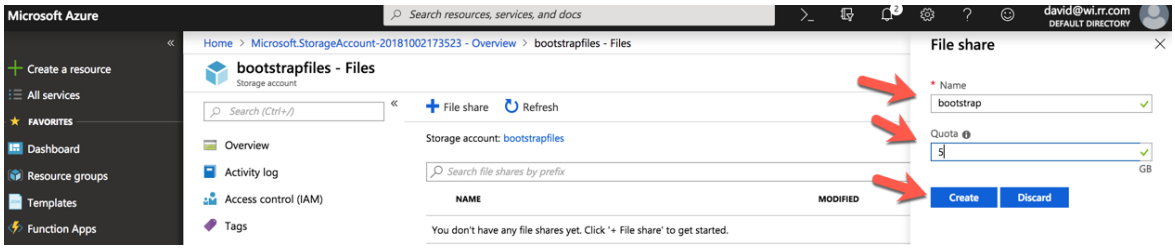
Once the storage account is open. Click on the Files section. This is where the folders and files to bootstrap the firewall will be placed.



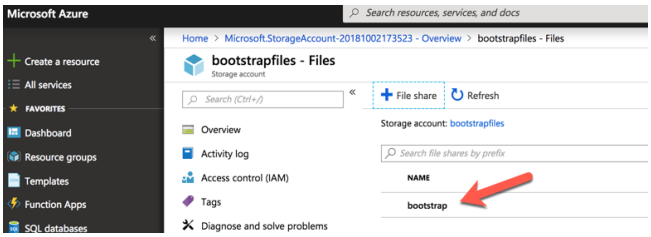
Next click the plus sign to create a new File Share:



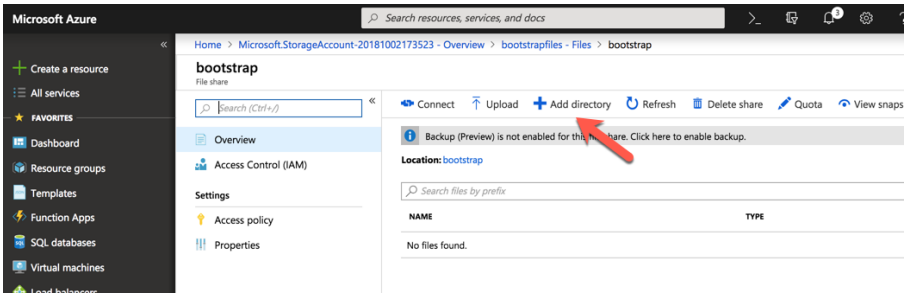
When the dialogue window opens, enter the file share information and click create. Note: The Name will be used to update the Variables.tf file in a few steps:



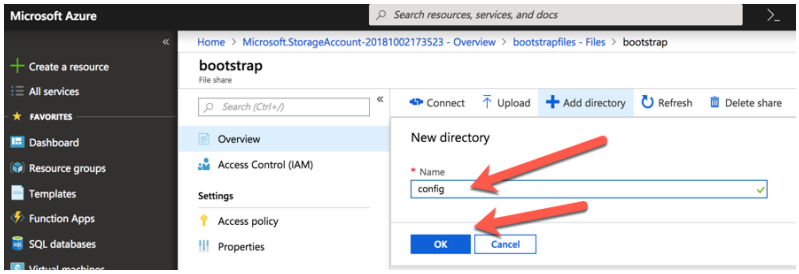
Click on the newly created file share:



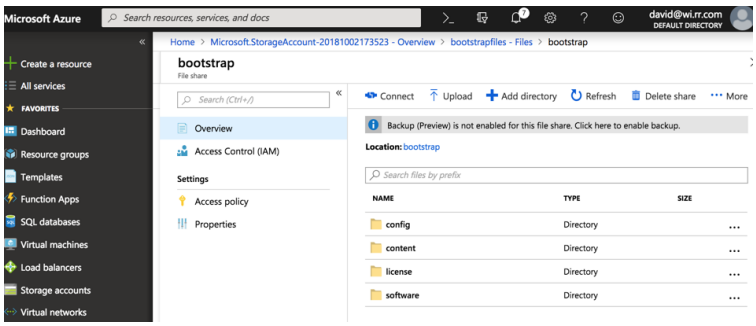
Click on the “Add directory” to create a directory:



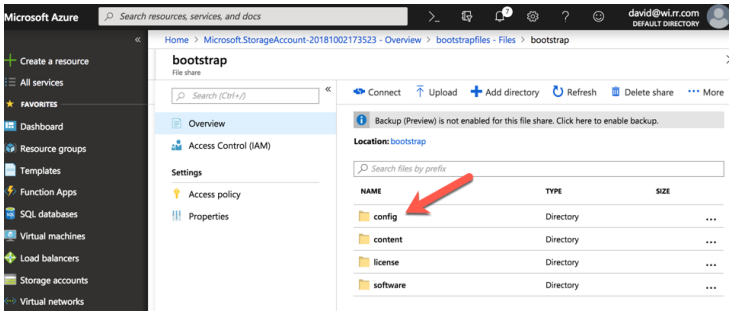
Enter config and click ok:



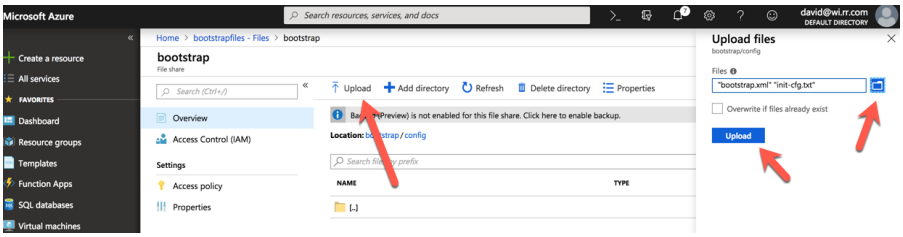
Repeat this step to create a content, license, and software directory. It is important that all 4 directories are present:



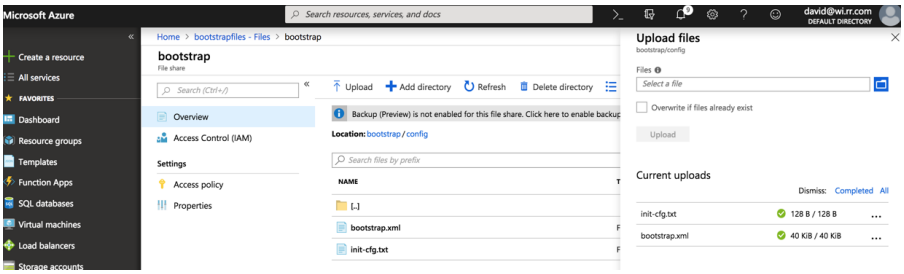
Click on the config folder:



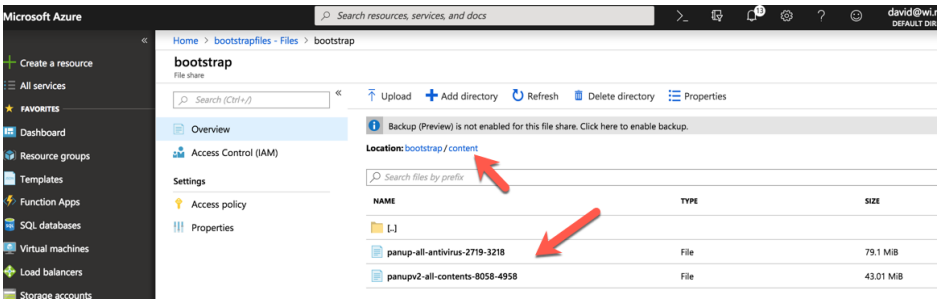
Click “Upload”. When the upload blade opens, select the folder browse and navigate to the files previously downloaded from GitHub. Select the bootstrap.xml and init-cfg.txt. Then click “Upload”:



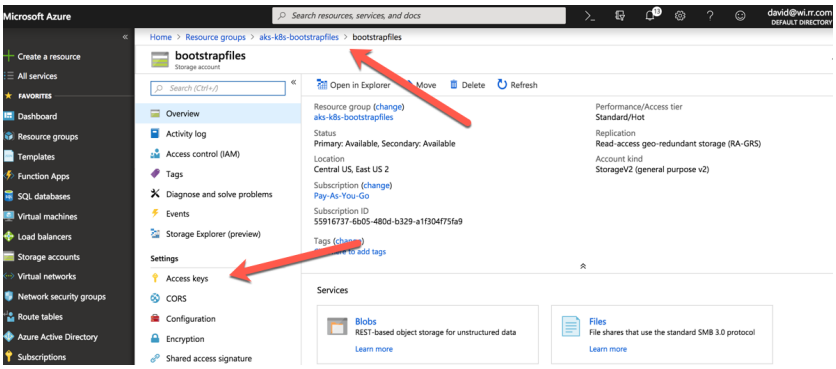
Once the files have been uploaded, they should be visible in the directory:



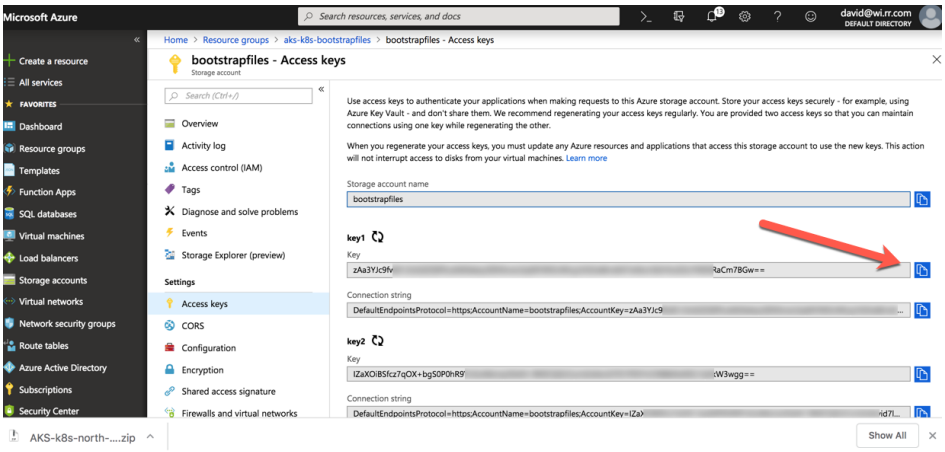
It is also possible to add content updates to the content directory that will get loaded into the firewall during the bootstrapping process. The follow figure shows some content files uploaded to the content directory:



The next step is to identify the Access Key and update the Terraform Variables.tf file. Navigate to the Storage account and click Access keys:



Next click the copy button to copy the access key for the storage account:



Open the Variables.tf file in an editor and update the custom data variable. The access key, storage account name, and share name need to be added:

```
variable "fwOffer" {
  default = "vmseries1"
}

variable "fwPublisher" {
  default = "paloaltonetworks"
}

variable "adminUsername" {
  default = "paloalto"
}
variable "adminPassword" {
  default = "Pal0Alt0@123"
}
variable "gvmSize" {
  default = "Standard_A1"
}
variable "customdata" {
  default = "storage-account=<insert storage account name>,access-key=<insert file key>,file-share=<insert share name>,share-directory=None"
}
```

This is a screen shot of the file with the updated information:

```
variable "fwOffer" {
  default = "vmseries1"
}

variable "fwPublisher" {
  default = "paloaltonetworks"
}

variable "adminUsername" {
  default = "paloalto"
}
variable "adminPassword" {
  default = "Pal0Alt0@123"
}
variable "gvmSize" {
  default = "Standard_A1"
}
variable "customdata" {
  default = "storage-account=bootstrapfiles,access-key=zAa3YJc5[redacted]n78Gw==,file-share=bootstrap,share-directory=None"
}
```

SSH keys

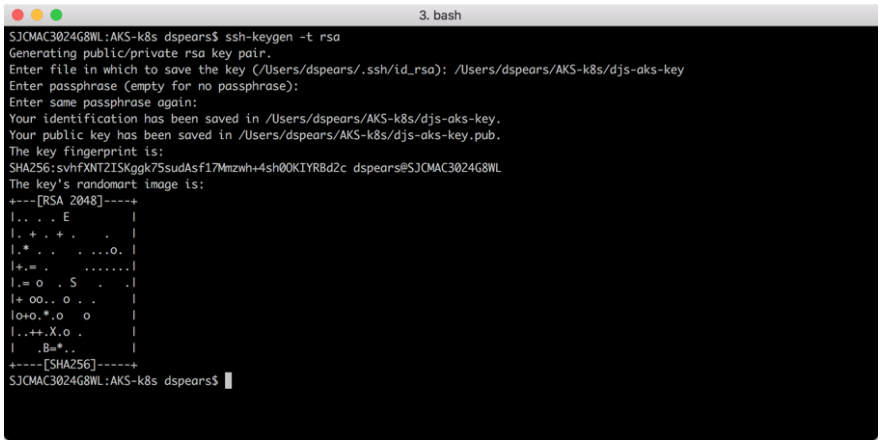
In this activity, you will:

Generate SSH Keys – if needed

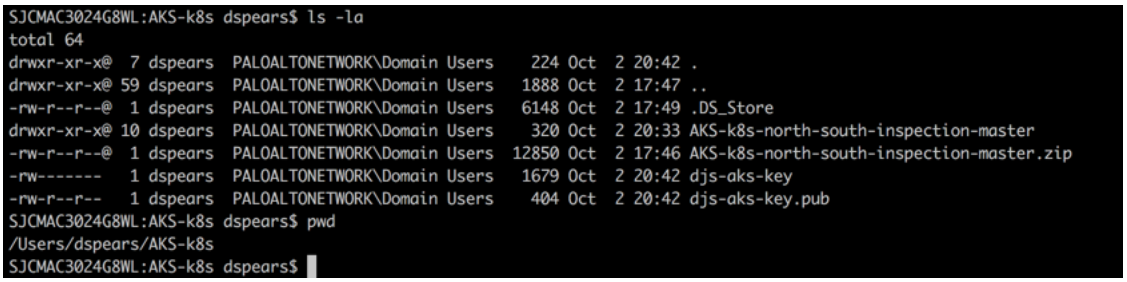
Update the Terraform Variables.tf with the path to the SSH keys

The Terraform Variables.tf file has an option for supplying ssh keys that can be used to log into the Kubernetes nodes after deployment.

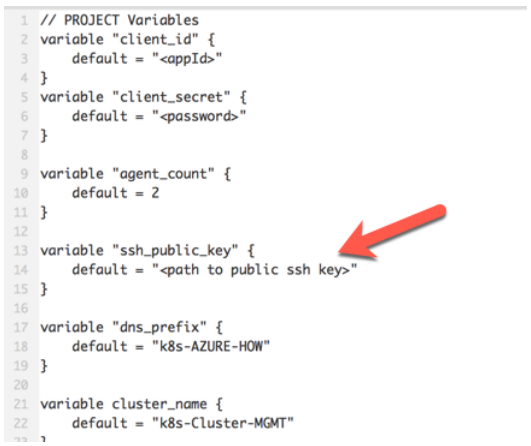
If you do not already have an SSH key, the follow example shows how to create an SSH key on a Mac using the **ssh-keygen -t rsa** command:



In the previous example the keys were generated and stored in the same directory as the other lab files. The public and private keys can be seen using the ls -la command.



Next edit the Terraform Variables.tf file to include the path to the public SSH key. The following diagram shows the field that needs to be updated and the field after it has been updated:



Deploy the Terraform Template

In this activity, you will:

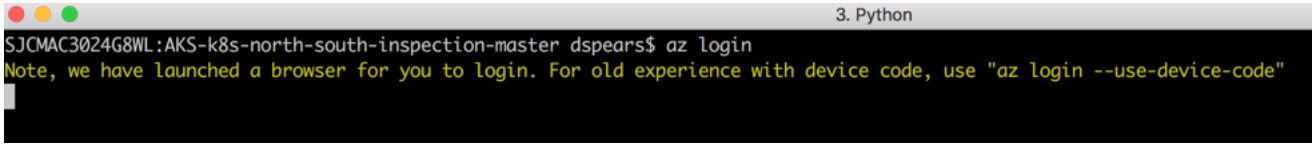
Authenticate to Azure via the Azure command line tool

Initialize Terraform and download the appropriate plugins

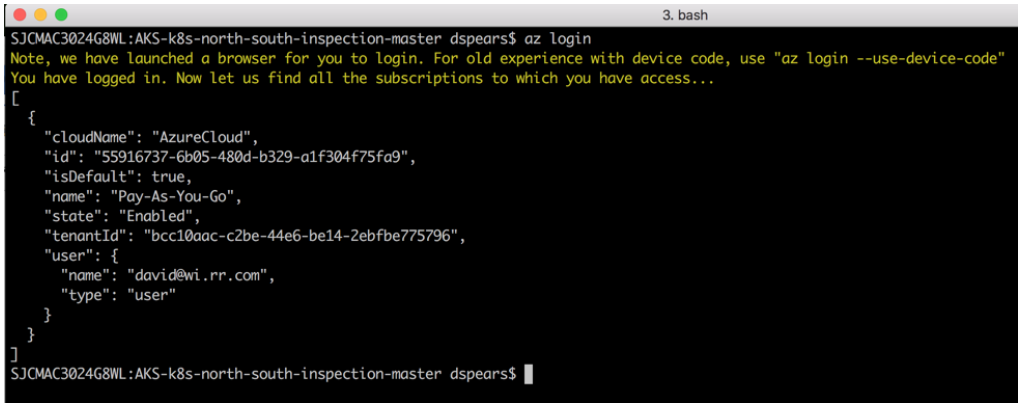
Apply the Terraform template

Open a terminal shell and navigate to the directory containing the Terraform template files.

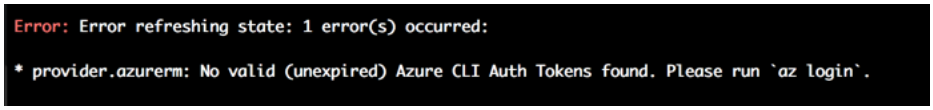
The Azure cli tool token obtained earlier has most likely expired. Use the "**az login**" login command to get a new token:



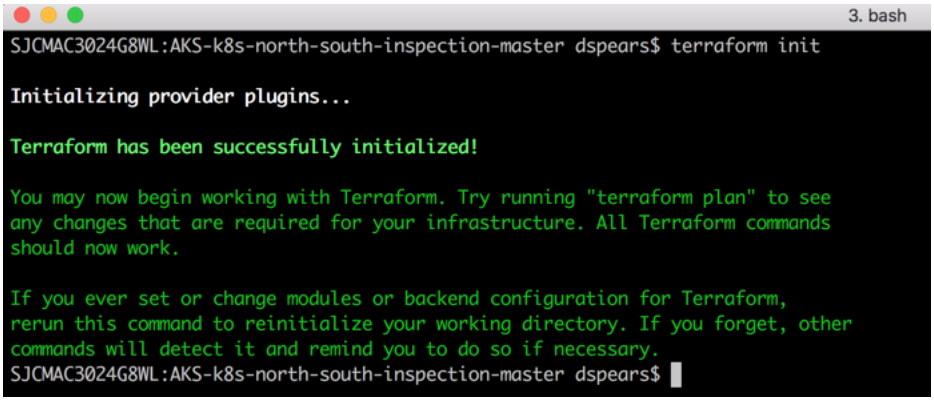
After getting redirected to the Microsoft Azure Login and completing the login process successfully, the following prompt will be displayed:



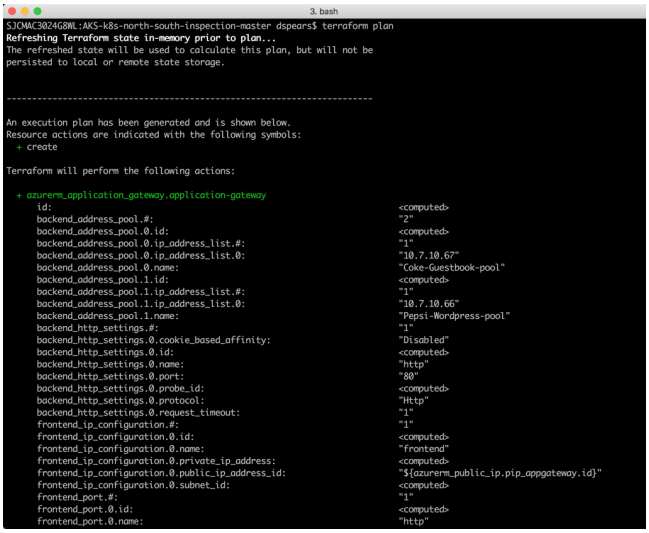
As a note, the following error message is displayed when the azure cli tool token has expired:



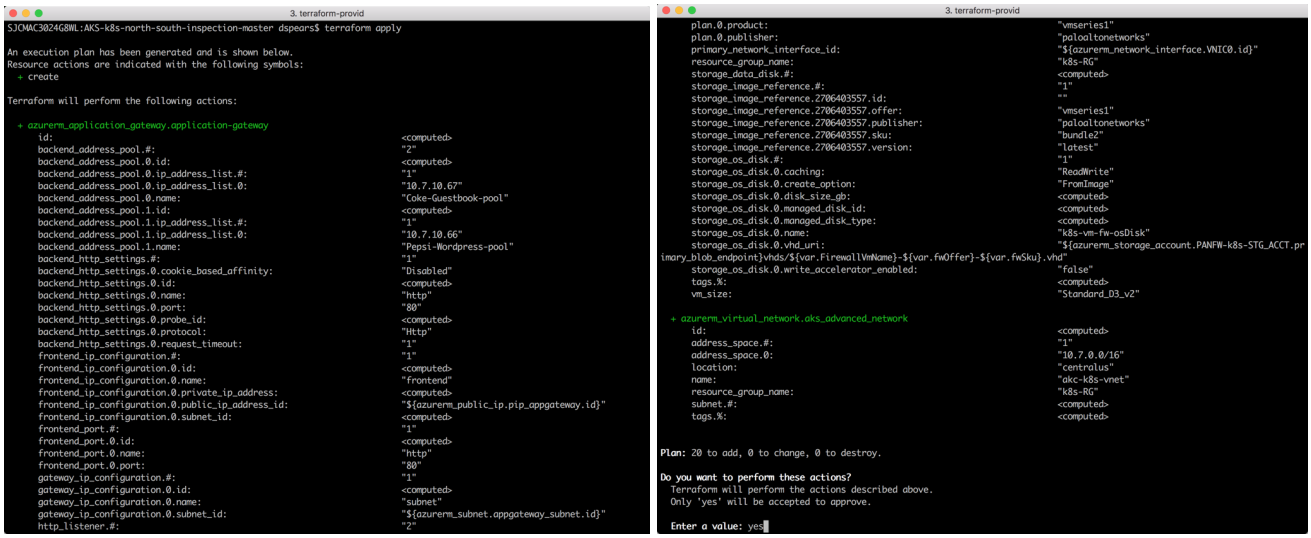
Ensure you are in the directory with the Main.tf and Variables.tf files and execute the "**terraform init**" command which will initialize terraform and ensure all the provider plugins are download and up to date:



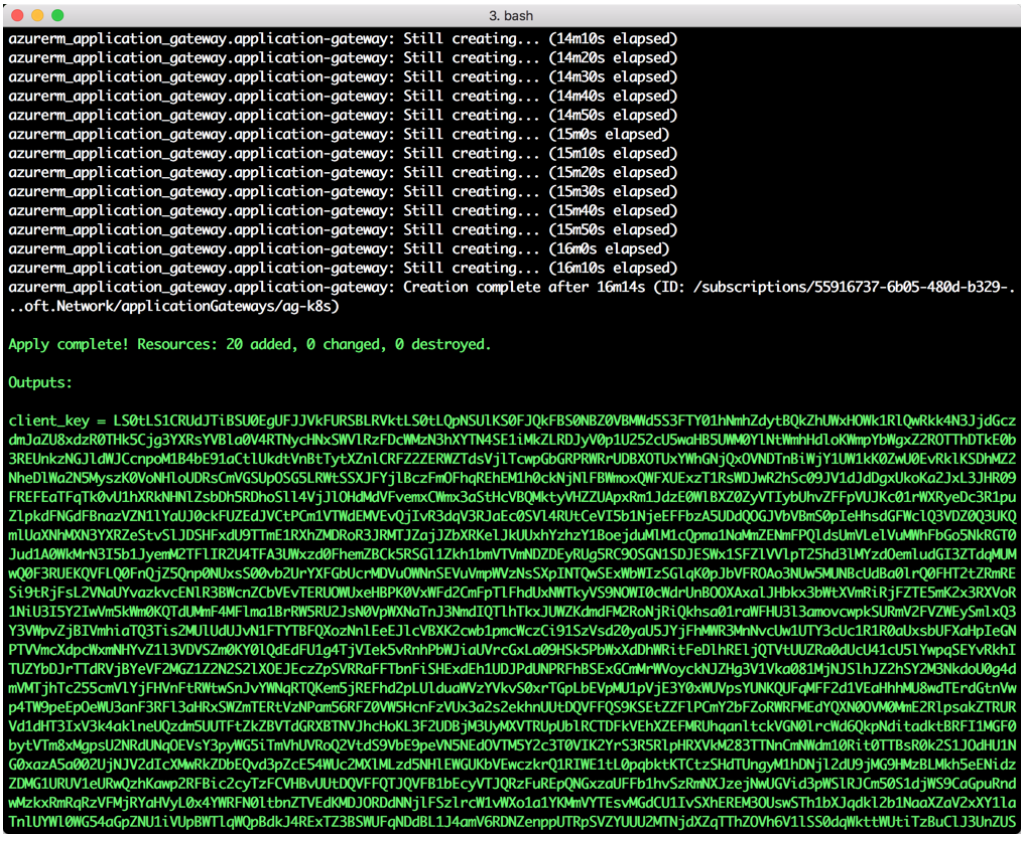
Once the terraform init has completed run the **terraform plan** command. This will show what changes will be implemented with the terraform script. This will also identify if there are any errors detected with the terraform files:



Now run the **terraform apply** command to deploy the template. At the action prompt enter **yes**.



It will take a few minutes to complete. If all goes well, Terraform will output; "Apply Complete!" and provide some additional output information about the resources deployed:



Review what was deployed

In this activity, you will:

Review the resources that have been launched

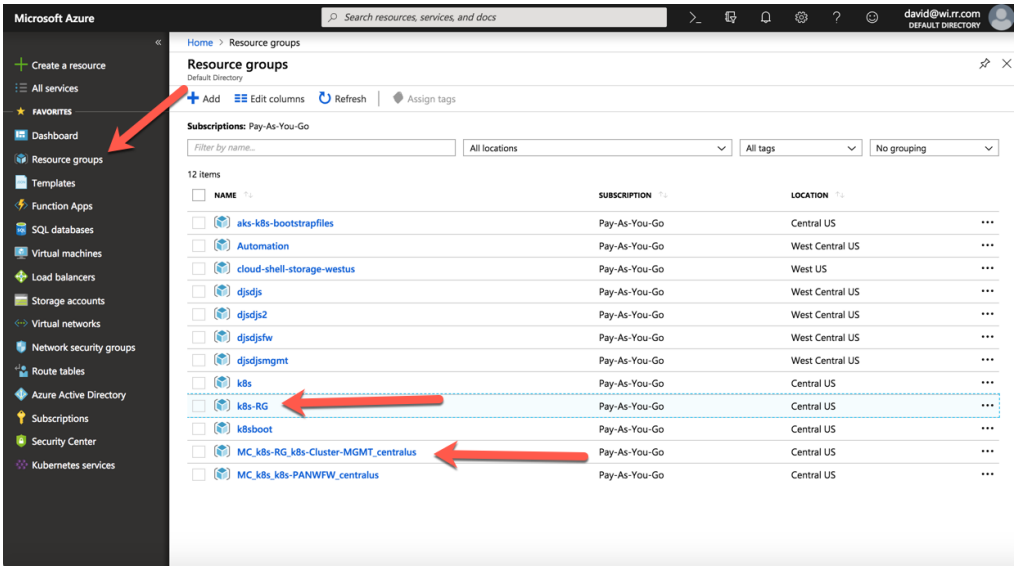
Inspect k8s cluster

Log into the VM-Series firewall

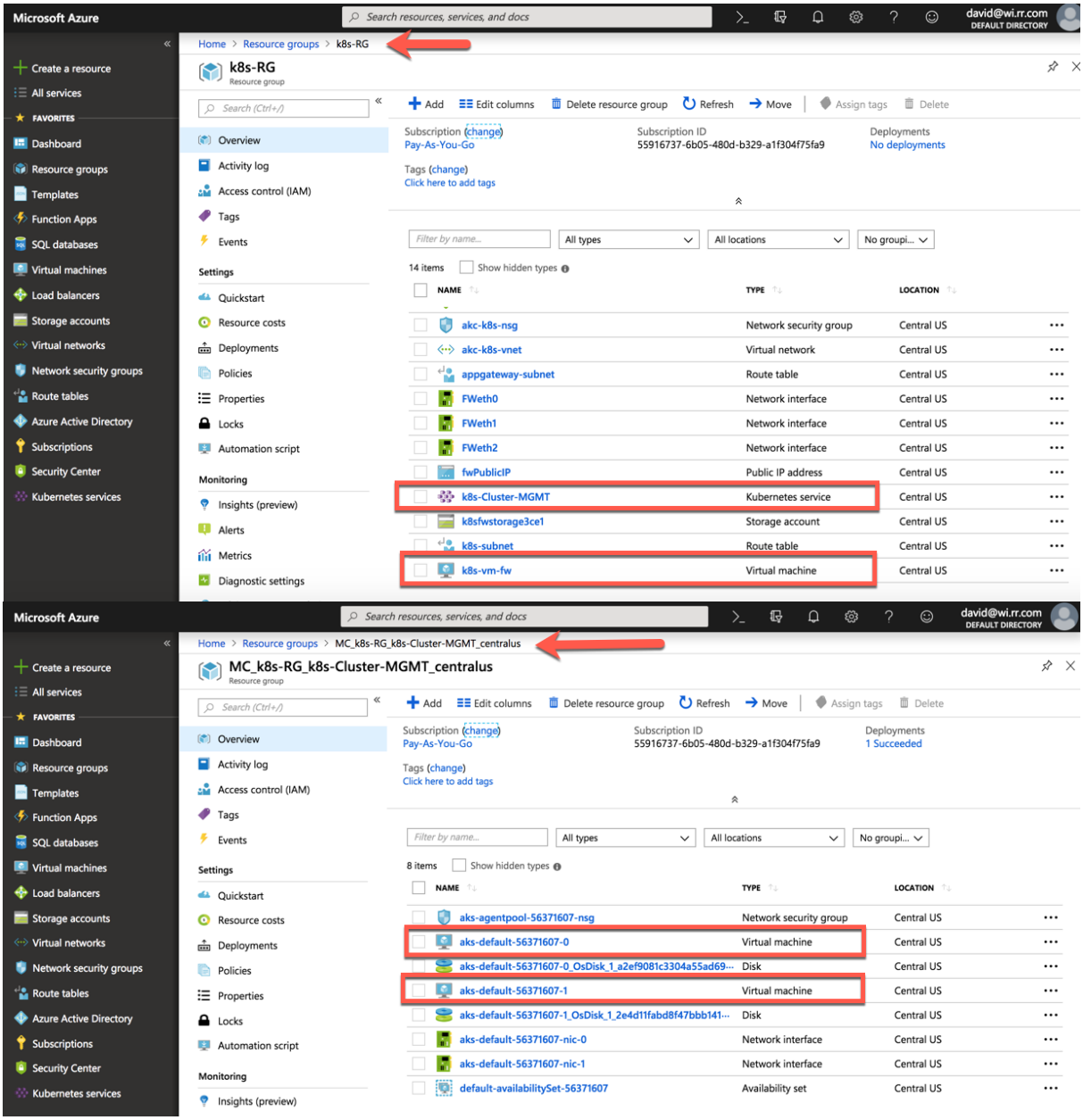
Confirm bootstrap success

Task 1 – Look around Azure console

Navigate to Resource Groups. Notice that there are two resource groups that were deployed. The first one, k8s-RG, has the infrastructure that was defined in the Terraform template. The second has the k8s nodes and associated resources.

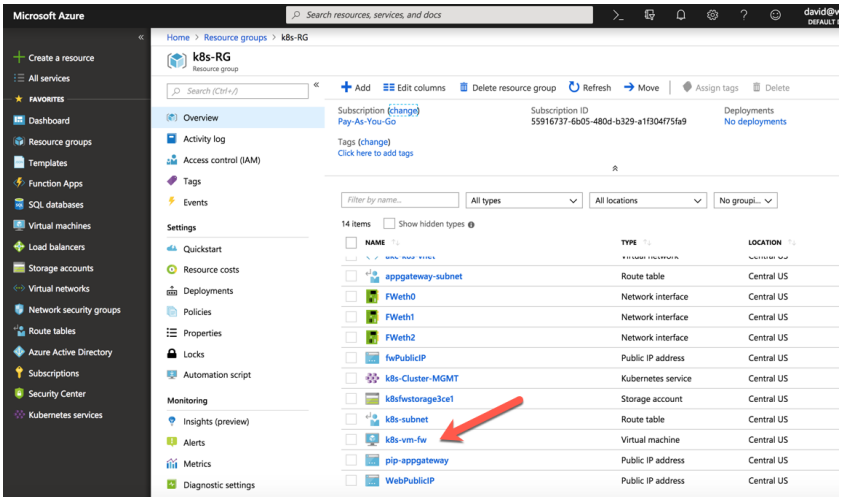


Open the two resource groups to view what has been deployed:

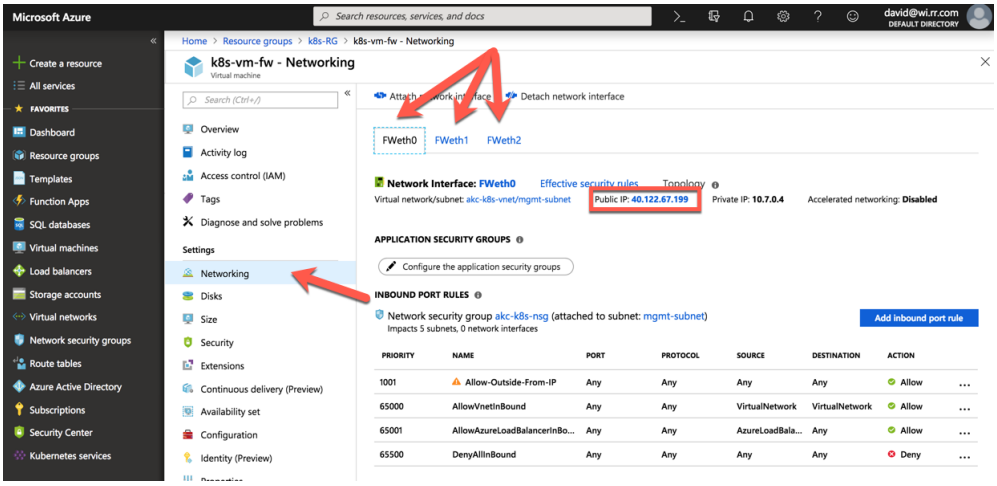


There should be 1 firewall, 1 k8s service master, and two k8s nodes displayed.

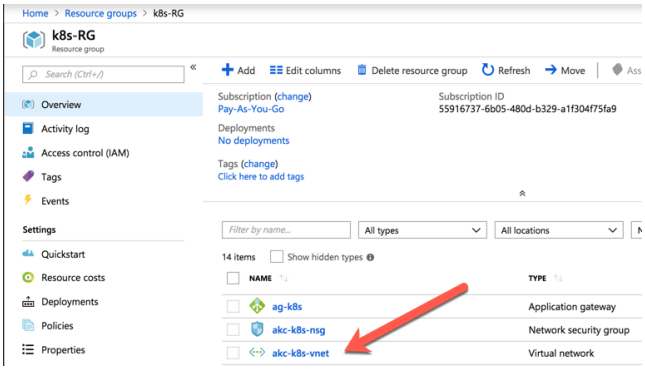
Click on the firewall to open a detailed view of the deployed firewall:



Explore the options on the firewall. One interesting area to review is the Networking section. The IP address and security information for each interface can be identified:

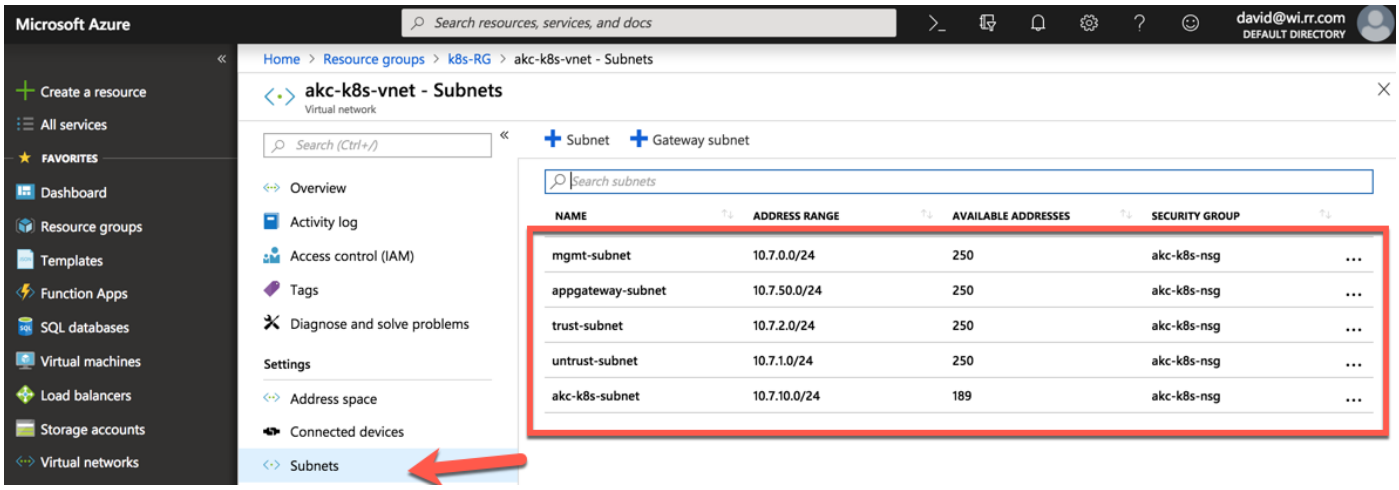


Navigate to akc-k8s-vnet virtual network in the k8s-RG resource group to see the different networks that have been created as part of the lab.

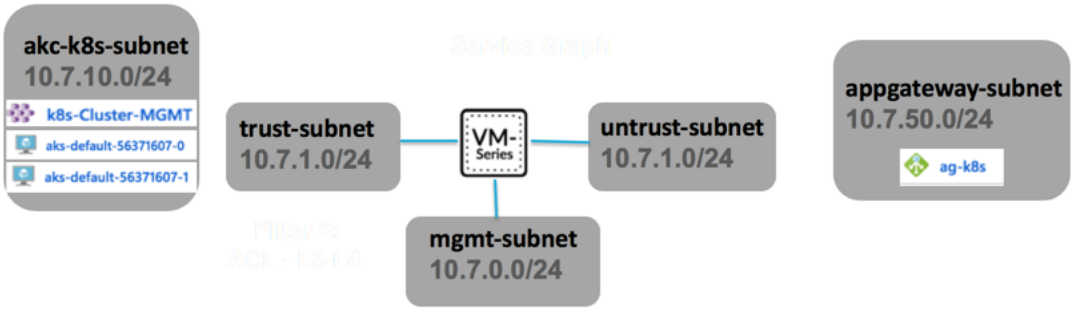


Click Subnets on the left Nav. You should see 5 subnets:

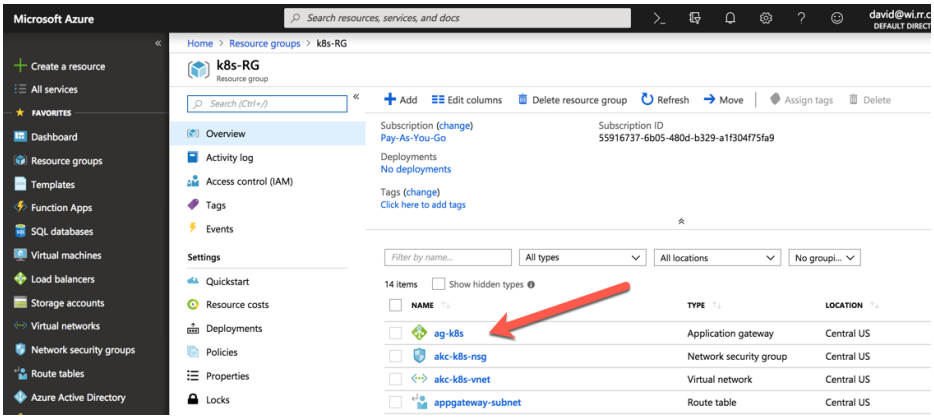
- mgmt-subnet, trust and untrust are used by the firewall
- appgateway-subnet is used by the application gateway
- akc-k8s-subnet is where the k8s nodes and load balancing services are deployed



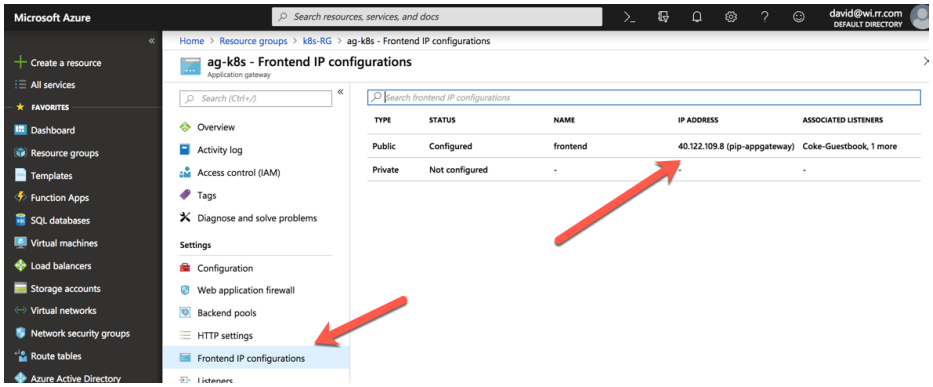
The following diagram describes the network topology of what has been deployed:



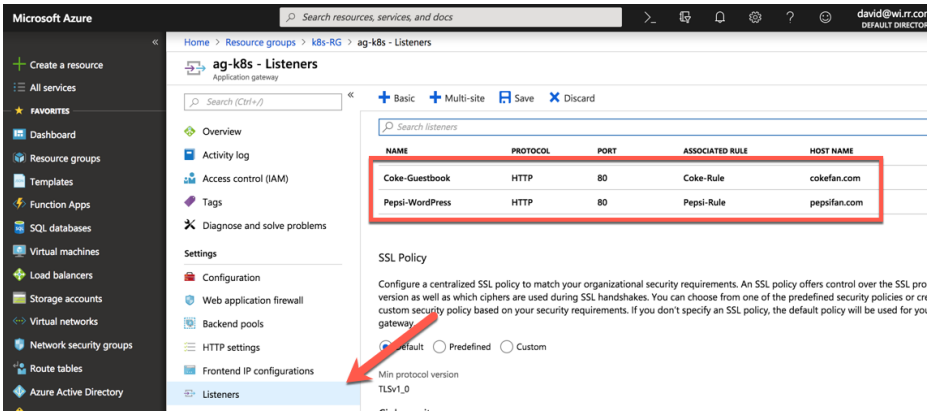
Next Navigate to the k8s-RG resource group and open the application gateway:



Click on the Frontend IP configurations options on the left Nav and notice that there is a single front-end IP address. The application gateways only support a single frontend address. This address will be needed later in the lab.



Next, go to Listeners on the left Nav. Notice that there are two listeners. This lab will leverage the Applications Gateway's ability to do host header redirection to send traffic to the correct internal load balancer address based on the http request.

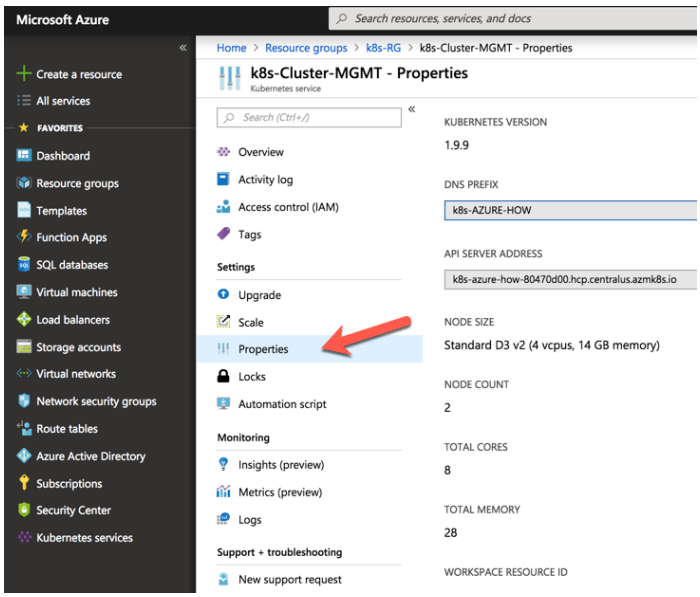


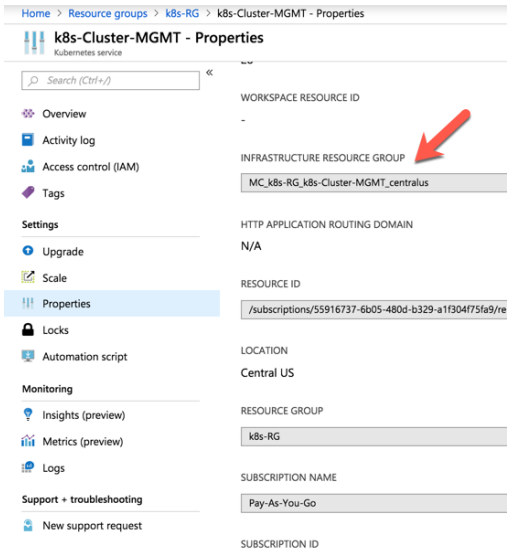
Feel free to navigate through other parts of the Azure Console. This will come in handy in activities later on.

Task 2 – Review the Kubernetes Cluster

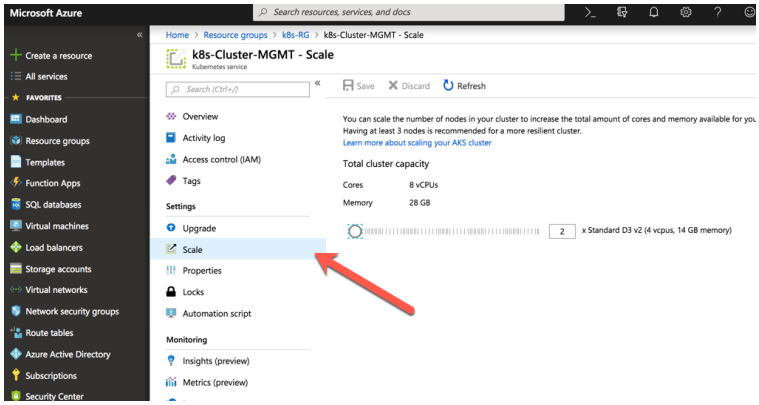
Kubernetes is a portable, extensible, open-source orchestrator that is used to manage containerized workloads. Kubernetes has a large and rapidly growing ecosystem. The portability of Kubernetes allows for workloads to be migrated between various clouds (public or private). Further documentation is available at: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Navigate to the k8s-RG resource group and click on the k8s-Cluster-MGMT resource. Click on Properties in the k8s-Cluster-MGMT blade. This will show the k8s version, number of nodes deployed, and the infrastructure resource group that was created to deploy k8s resources. This is where the k8s nodes get deployed.





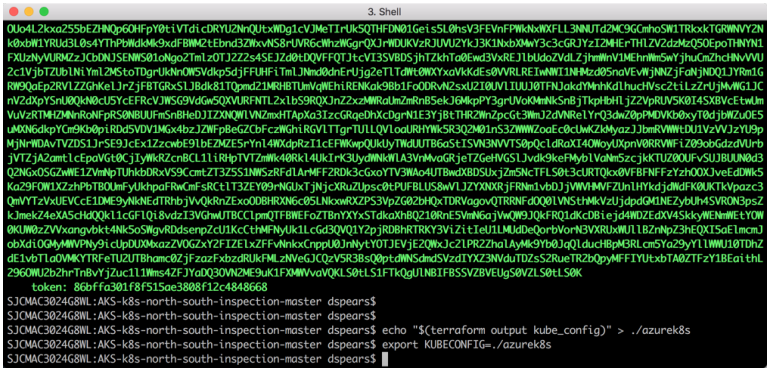
Clicking on the Scale link in the left Navigation displays the current number of nodes. From here the number of nodes deployed in the cluster can be increased or decreased.



Task 3 – Connect to the Kubernetes Cluster

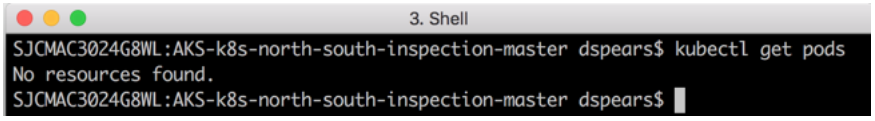
Navigate back to the terminal window used to deploy the Terraform script. In order to run Kubctl commands, the Kubernetes config from the Terraform state need to be captured and stored in a file that kubectl can read. Execute the following commands in the same directory that the terraform files are in:

$ echo "$(terraform output kube_config)" > ./azurek8s
$ export KUBECONFIG=./azurek8s



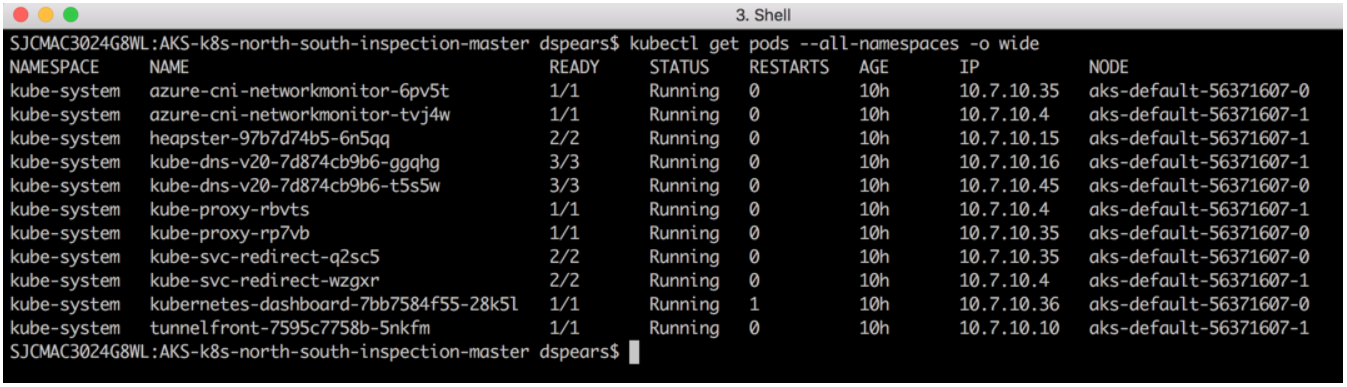
Let's explore some pods and services that have deployed. Run this command in the cloud shell:

$ kubectl get pods



Since we have not deployed any resources this is normal. Now let us see what system pods have been deployed. Run this command in the shell:

$ kubectl get pods --all-namespaces -o wide



Note: If the output does not show all the pods in a running state, wait and rerun the **kubectl get pods --all-namespaces -o wide** command until they do. An example of this is state is in the following screen-print:

```
davejspears@cloudshell:~ (djs-gcp-2018)$ kubectl get pods -o wide
NAMESPACE     NAME                                                  READY   STATUS              RESTARTS   AGE     IP          NODE
kube-system   heapster-v1.4.3-6644cc4b46-dwn8s                      0/2     Pending             0          11m     <none>      gke-cluster-1-default-pool-2df01519-n3w8
kube-system   heapster-v1.4.3-7875d9f9ff-mlv7g                      2/2     Terminating         0          12m     10.8.0.4    gke-cluster-1-default-pool-2df01519-6sgf
kube-system   kube-dns-778977457c-2xwfq                             3/3     Running             0          13m     10.8.0.3    gke-cluster-1-default-pool-2df01519-6sgf
kube-system   kube-dns-778977457c-144rf                             0/3     ContainerCreating   0          12m     <none>      gke-cluster-1-default-pool-2df01519-n3w8
kube-system   kube-dns-autoscaler-7db47cb9b7-j8n7n                  1/1     Running             0          13m     10.8.1.3    gke-cluster-1-default-pool-2df01519-n3w8
kube-system   kube-proxy-gke-cluster-1-default-pool-2df01519-6sgf   1/1     Running             0          12m     10.5.2.2    gke-cluster-1-default-pool-2df01519-6sgf
kube-system   kube-proxy-gke-cluster-1-default-pool-2df01519-n3w8   1/1     Running             0          12m     10.5.2.3    gke-cluster-1-default-pool-2df01519-n3w8
kube-system   kubernetes-dashboard-6bb875b5bc-n7wj8                 1/1     Running             0          13m     10.8.0.2    gke-cluster-1-default-pool-2df01519-6sgf
davejspears@cloudshell:~ (djs-gcp-2018)$
```

Now let us see what services have been deployed as part of the system:
Run the following in the shell:

$ kubectl get svc

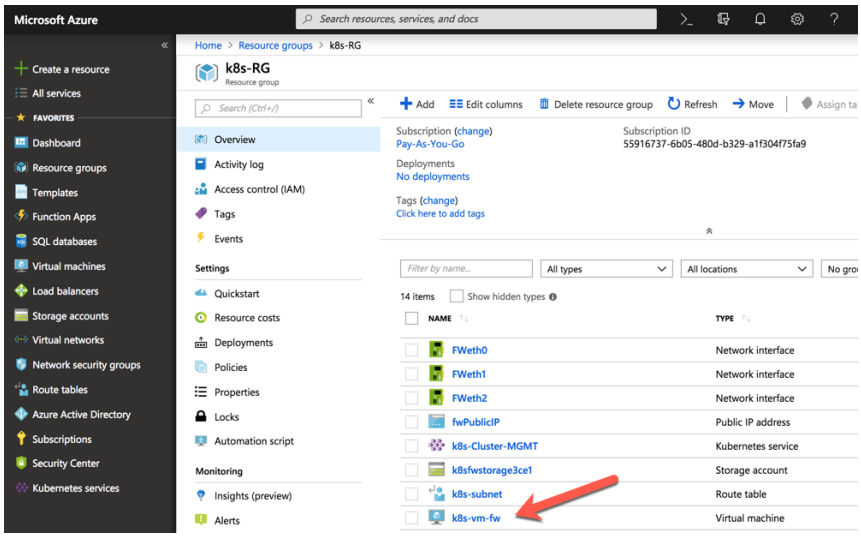
```
3. Shell
SJCMAC3024G8WL:AKS-k8s-north-south-inspection-master dspears$ kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.21.0.1    <none>        443/TCP   10h
SJCMAC3024G8WL:AKS-k8s-north-south-inspection-master dspears$
```

As you can see no services besides the system cluster have been deployed.

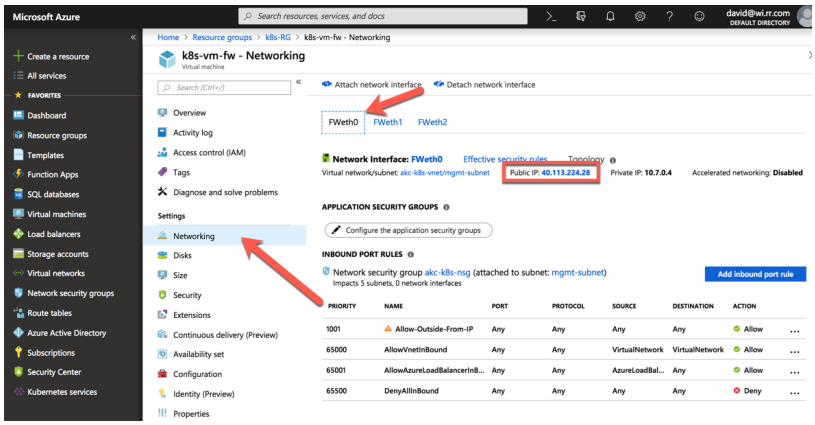
Task 3 – Log into the firewall

The VM-Series firewall deployed as part of the lab has been bootstrapped. Bootstrapping is a feature of the VM-Series firewall that allows you to load a pre-defined configuration into the firewall during boot-up. This ensures that the firewall is configured and ready at initial boot-up, thereby removing the need for manual configuration. The bootstrapping feature also enables automated deployment of the VM-Series.

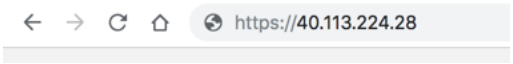
Navigate to the k8s-RG resource group and click on the VM-Series firewall Virtual machine:



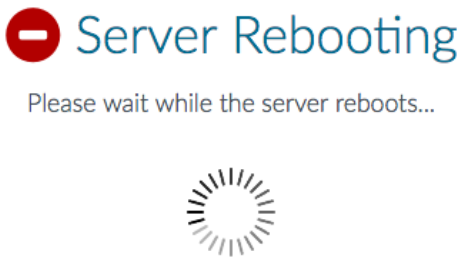
Click on Networking in the left Nav. Copy the Public IP of FWeth0 which is the mgmt interface of the VM-Series firewall:



Open another browser tab and navigate to the firewall management interface:



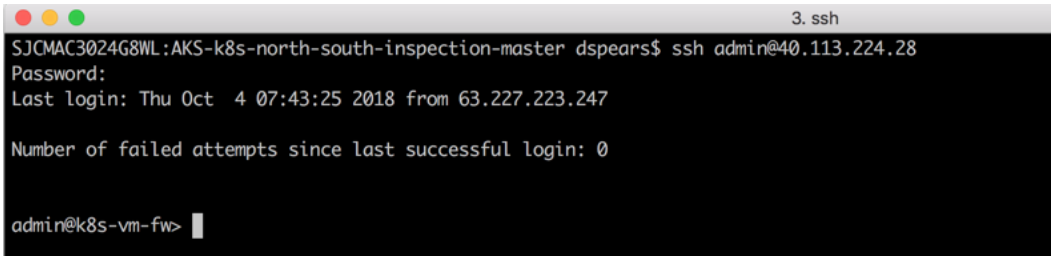
If you get a security exception, please ignore for this lab and proceed to the firewall login page. The VM-Series firewall by default uses a self-signed certificate which causes the exception. Depending on how quickly you do this, you might see the following message. It is normal and part of the bootup process:



If you wish to SSH into the FW, the following syntax can be used:

$ ssh admin@<ip address of firewall>

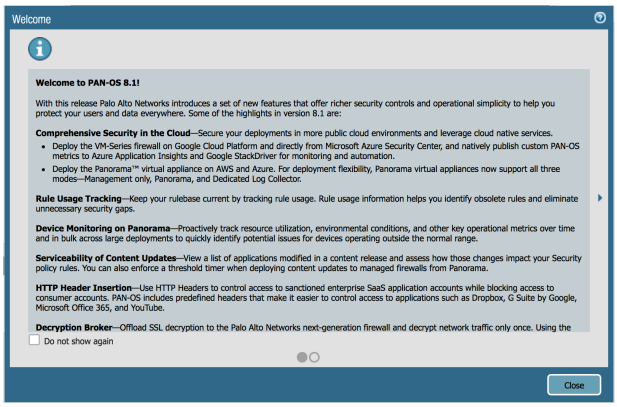
The password is: Pal0Alt0@123 -yes, those are zeros.



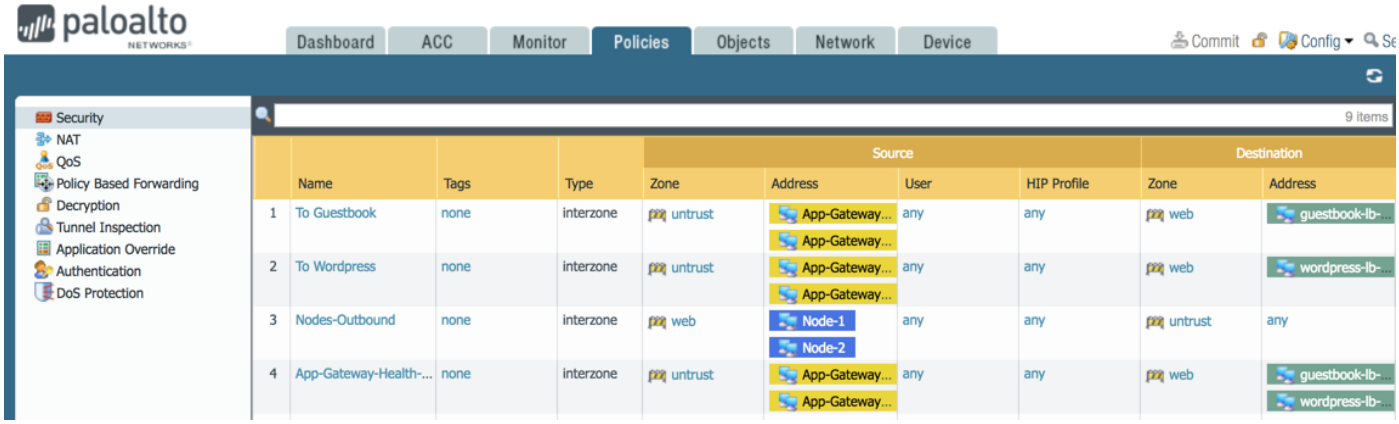
When presented with the login screen you should be able to login to the firewall using (Hint: It's a good idea to jot this password down or save it to a notepad as you will regularly need it):

username: **admin**
password: **Pal0Alt0@123** **-yes, those are zeros.**

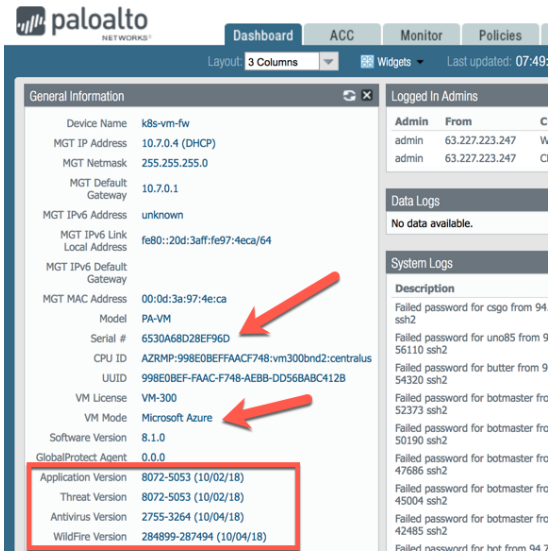
Once logged in you will see a welcome screen, dismiss the welcome dialog box by clicking Close.



Click the Policies tab and you will notice a predefined security policy which was imported using the bootstrapping feature. There are also some predefined NAT policies:



Click on the Dashboard tab, check to verify that the firewall has a serial number. The image defined in the terraform template is a Pay as you Go bundle2. This was used because a license will be required to view the logs later in the lab. If you added content files in the bootstrap folder, you should also see that these have been uploaded.



Launch a two tiered WordPress application

In this activity, you will:

Optionally: Explore the application's manifest file

Launch a two-tier WordPress application within your cluster

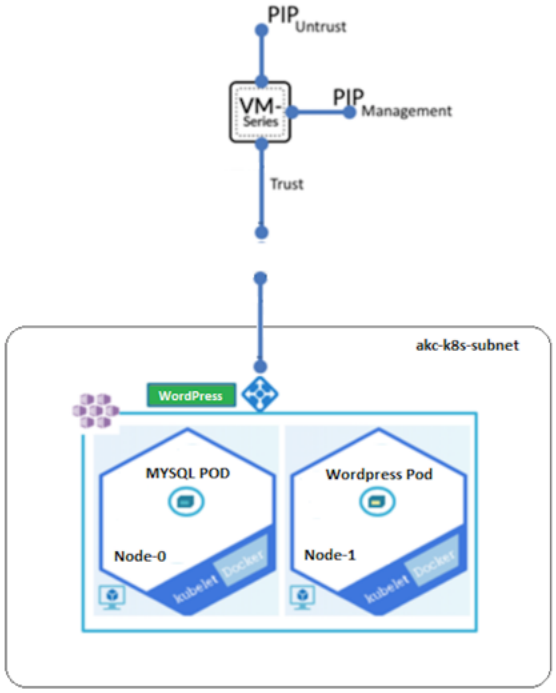
In this activity we will start using Kubernetes specific terms such as Pods, Services, etc.
Here is a good primer: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

Task 1 – WordPress Application Deployment YAML file

WordPress is a piece of software which has become one of the most widely used content management systems. It is open source, licensed under the GPL, and written in PHP.

WordPress allows users to create and edit websites through a central administrative dashboard, which includes a text editor for modifying content, menus and various design elements. WordPress provides plugins which provide additional functionality through WordPress Plugin Directory. Plugins can be installed through either upload or by one-click installation through the WordPress Plugin Library.

This lab will deploy the following simple WordPress application on the cluster nodes created during the Terraform template deployment:



As you can see this is a two-tiered application with Pods that are dedicated to front-end WordPress services and backend MYSQL DB services.

If interested, the following section dives a bit deeper into the templates being used to create this application. There are two application manifests for this deployment. The first is for the MYSQL DB and the second is for the WordPress frontend. Optionally, open the links below it in a browser of your choice to view the files.

https://github.com/PaloAltoNetworks/AKS-k8s-north-south-inspection/blob/master/mysql-deployment.yaml
and
https://github.com/PaloAltoNetworks/AKS-k8s-north-south-inspection/blob/master/wordpress-deployment.yaml

The manifest file declares various aspects of the application. For instance, it tells the orchestrator what type of resources you intend to deploy. In this case we will first deploy a MYSQL DB server and then a WordPress Frontend.

MYSQL Service:

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
    tier: mysql
  clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
      - image: djspears/panw:wordpress-mysql
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

Some things to notice are the listening port, 3306, the container image, and the credentials that will be used during the deployment.

Wordpress-Frontend :

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
      - image: djspears/panw:wordpress
        name: wordpress

        - name: WORDPRESS_DB_HOST
          value: wordpress-mysql
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 80
          name: wordpress
        volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wp-pv-claim
```

Highlighted in this file are the area that specifies the load balancer service and also the container image. Even though we have two tiers in our application, only one (the frontend service) is exposed to the outside world via a load balancer. The annotation listed above tells AKS and Kubernetes that the load balancer would be of type: Internal.

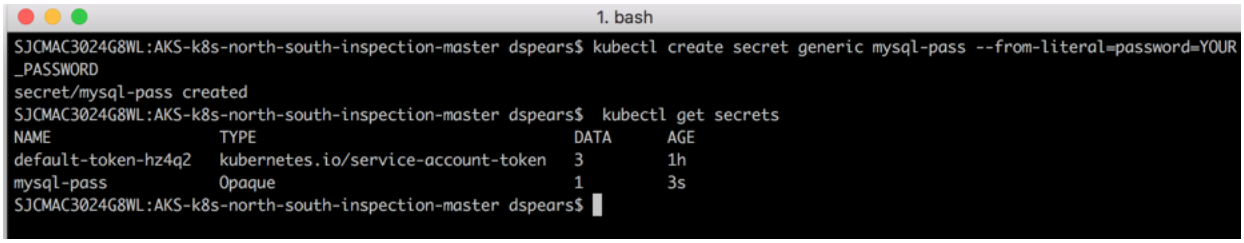
Task 2 – Launch the Application

As mentioned previously, the application deployment will be done in two steps. The first step will be to deploy the MYSQL DB server. One of the parameters that needs to be passed to the DB server is a root password. To do this securely, the kubectl secrets command will be used. Kubectl secrets are objects intended to hold sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a secret is safer and more flexible than putting it verbatim in a pod definition or in a docker image. To create a secret, execute the following commands in the terminal window:

$ kubectl create secret generic mysql-pass --from-literal=password=YOUR_PASSWORD

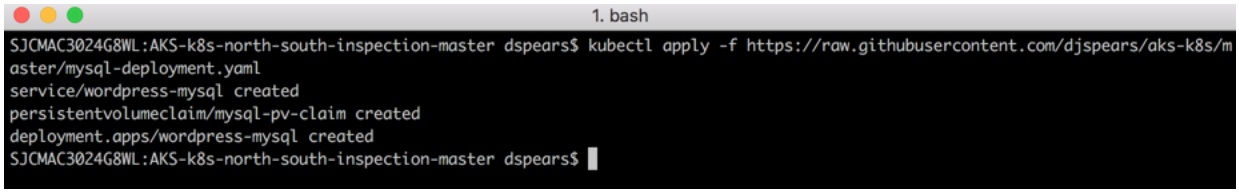
And the following command will verify that the secrets have been stored

$ kubectl get secrets



Now the MYSQL pod can be deployed. To do this, execute the following command:

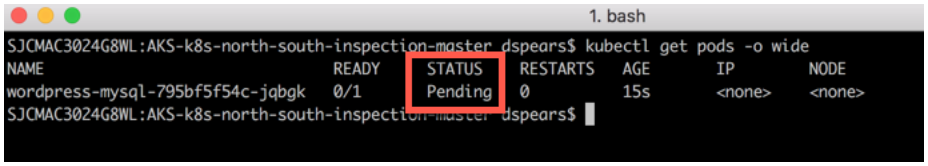
$ kubectl apply -f https://raw.githubusercontent.com/PaloAltoNetworks/AKS-k8s-north-south-inspection/master/mysql-deployment.yaml

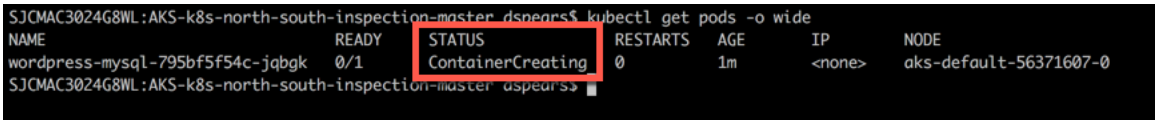


You should see the services and deployments being created. Next, validate the new pods in your cluster have been created. In your terminal execute:

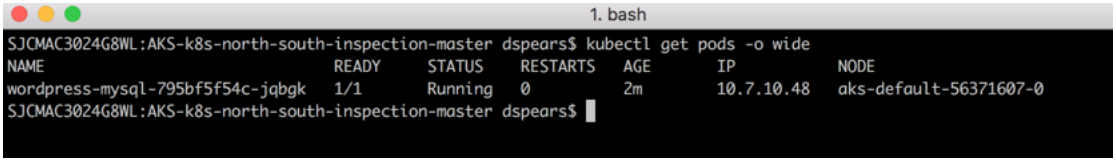
$ kubectl get pods -o wide

You may see the status as Pending or ContainerCreating. This is usually a normal situation:



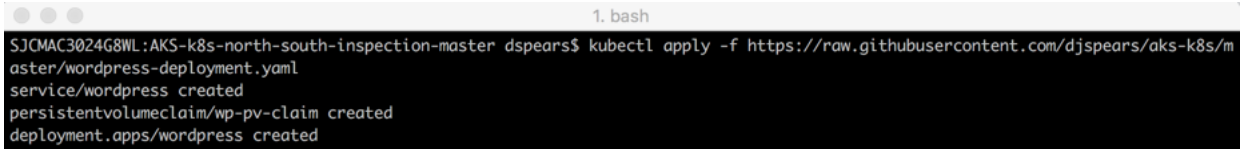


By executing the **kubectl get pods -o wide** again, you start seeing that the Ready and Status of pods change as they start up. Verify that the pod gets to a running status.



With the MYSQL DB Running, create the WordPress frontend by executing the following command:

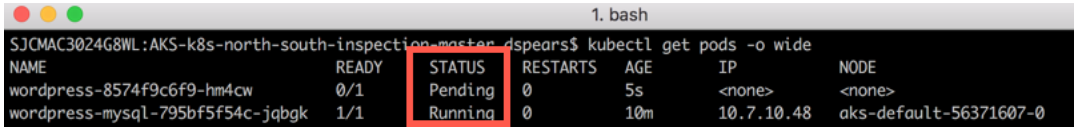
$ kubectl apply -f https://raw.githubusercontent.com/PaloAltoNetworks/AKS-k8s-north-south-inspection/master/wordpress-deployment.yaml

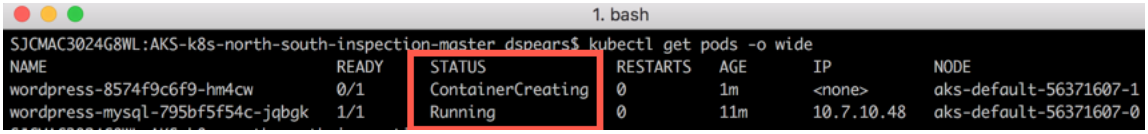


Next, validate the new pods in your cluster have been created. In your terminal execute:

$ kubectl get pods -o wide

Again, you may see the status as Pending or ContainerCreating. This is usually a normal situation:





Again, verify that the pod gets to a running status.

```
1. bash
SJCMAC3024G8WL:AKS-k8s-north-south-inspection master dspears$ kubectl get pods -o wide
NAME                              READY     STATUS    RESTARTS   AGE       IP           NODE
wordpress-8574f9c6f9-hm4cw        1/1       Running   0          5m        10.7.10.31   aks-default-56371607-1
wordpress-mysql-795bf5f54c-jqbgk  1/1       Running   0          15m       10.7.10.48   aks-default-56371607-0
SJCMAC3024G8WL:AKS-k8s-north-south-inspection master dspears$
```

Launch a two tiered Guestbook application

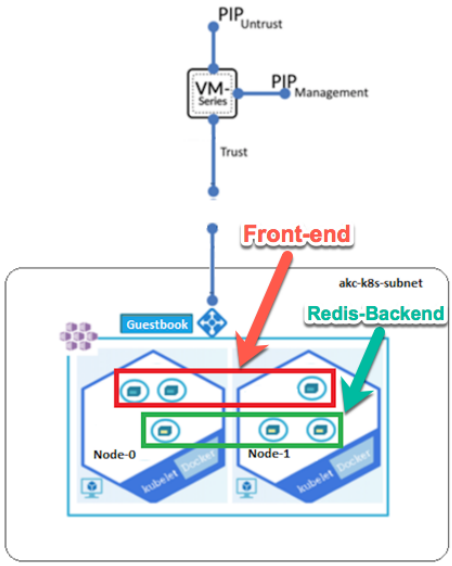
In this activity, you will:

Optionally: Explore the application's manifest file

Launch a two-tier WordPress application within your cluster

Task 1 – Guestbook Application Deployment YAML file

Guestbooks have been used by businesses for many years as a way to connect with customers and obtain contact information for future events and promotions. Today, businesses such as popular retail stores, 5-star hotels and even small family-owned B & B's are turning to iPad guestbook apps to help them gather information and enhance the customer's "in-biz" experience. Acquiring email addresses and a **social media** following is a crucial part of any marketing plan. With much of the population using computers on a daily basis, an email marketing plan is of the utmost importance. Using a guest book app in your store makes collecting email addresses a snap and offers enticing features with which the traditional paper and pen guestbook just can't compete. The guestbook application we will build and secure today could be used for Hotel website visits, shopping sites or any other business that wants to keep track of their customer and provide them with promotions or advertisements.
This lab will deploy the following simple Guestbook application on the cluster nodes created during the Terraform template deployment:



As you can see this is a two-tiered application with Pods that are dedicated to front-end web services and backend DB services.
If interested, the following section dives a bit deeper into the templates being used to create this application. This is a link to the application manifest. Optionally, click the link below and open it in a browser of your choice.

https://github.com/PaloAltoNetworks/AKS-k8s-north-south-inspection/blob/master/guestbook-all-in-one.yaml

The manifest file in this case we will deploy a 2-tier simple redis application with a fronted and backend tier. The backend tier will consist of a redis-master and redis-slave for db redundancy. Front-end Service:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
  labels:
    app: guestbook
    tier: frontend
spec:
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: guestbook
    tier: frontend
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below:
          # value: env
        ports:
        - containerPort: 80
```

This tells the orchestrator that the service will be exposed via an internal load balancer

Redis-backend-master :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    app: redis
    tier: backend
    role: master
spec:
  #type: LoadBalancer
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    tier: backend
    role: master
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis-master
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: gcr.io/google_containers/redis:e2e  # or just image: redis
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
```

Redis-backend-slave:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    tier: backend
    role: slave
spec:
  #type: LoadBalancer
  ports:
  - port: 6379
  selector:
    app: redis
    tier: backend
    role: slave
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis-slave
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v1
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access an environment variable to find the master
          # service's host, comment out the 'value: dns' line above, and
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 6379
```

Even though there are two tiers in the application, only one (the frontend service) is exposed to the outside world via a load balancer. The annotation listed above tells GCP and Kubernetes that the load balancer would be of type: Internal.

Task 2 – Launch the Application

Back in terminal shell type the following command to deploy the application pods:

$ kubectl apply -f https://raw.githubusercontent.com/PaloAltoNetworks/AKS-k8s-north-south-inspection/master/guestbook-all-in-one.yaml

```
1. bash
SJCMAC3024G8WL:AKS-k8s-north-south-inspection-master dspears$ kubectl apply -f https://raw.githubusercontent.com/PaloAltoNetworks/A
KS-k8s-north-south-inspection/master/guestbook-all-in-one.yaml
service/redis-master created
deployment.extensions/redis-master created
service/redis-slave created
deployment.extensions/redis-slave created
service/frontend created
deployment.extensions/frontend created
```

You should see the services and deployments being created. Next, validate the new pods in your cluster have been created. In your terminal execute:

$ kubectl get pods -o wide

You may see the status as Pending or ContainerCreating. This is usually a normal situation:

```
1. bash
SJCMAC3024G8WL:AKS-k8s-north-south-inspection-master dspears$ kubectl get pods -o wide
NAME                                READY     STATUS              RESTARTS   AGE       IP           NODE
frontend-67f65745c-km7d2            0/1       ContainerCreating   0          4s        <none>       aks-default-56371607-1
frontend-67f65745c-vt8dp            0/1       ContainerCreating   0          4s        <none>       aks-default-56371607-1
frontend-67f65745c-vttjj            0/1       ContainerCreating   0          4s        <none>       aks-default-56371607-0
redis-master-7747787588-rl86s       0/1       ContainerCreating   0          5s        <none>       aks-default-56371607-1
redis-slave-865486c9df-f252b        0/1       ContainerCreating   0          4s        <none>       aks-default-56371607-0
redis-slave-865486c9df-l2j5h        0/1       ContainerCreating   0          4s        <none>       aks-default-56371607-1
wordpress-8574f9c6f9-hm4cw          1/1       Running             0          47m       10.7.10.31   aks-default-56371607-1
wordpress-mysql-795bf5f54c-jqbgk    1/1       Running             0          57m       10.7.10.48   aks-default-56371607-0
```

By executing the **kubectl get pods -o wide** again, you start seeing that the Ready and Status of pods change as they start up. Verify that the pods gets to a running status.

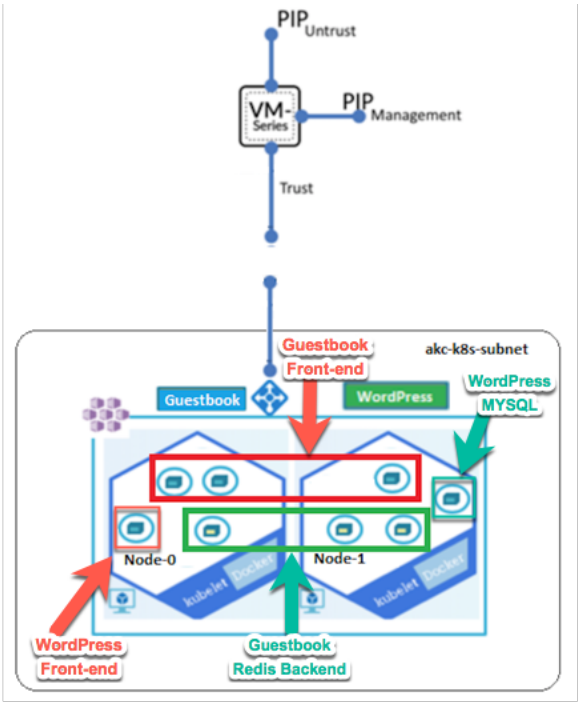
```
1. bash
SJCMAC3024G8WL:AKS-k8s-north-south-inspection-master dspears$ kubectl get pods -o wide
NAME                                READY     STATUS    RESTARTS   AGE       IP           NODE
frontend-67f65745c-km7d2            1/1       Running   0          1m        10.7.10.14   aks-default-56371607-1
frontend-67f65745c-vt8dp            1/1       Running   0          1m        10.7.10.11   aks-default-56371607-1
frontend-67f65745c-vttjj            1/1       Running   0          1m        10.7.10.46   aks-default-56371607-0
redis-master-7747787588-rl86s       1/1       Running   0          1m        10.7.10.6    aks-default-56371607-1
redis-slave-865486c9df-f252b        1/1       Running   0          1m        10.7.10.41   aks-default-56371607-0
redis-slave-865486c9df-l2j5h        1/1       Running   0          1m        10.7.10.34   aks-default-56371607-1
wordpress-8574f9c6f9-hm4cw          1/1       Running   0          49m       10.7.10.31   aks-default-56371607-1
wordpress-mysql-795bf5f54c-jqbgk    1/1       Running   0          59m       10.7.10.48   aks-default-56371607-0
```

Explore the newly deployed applications

In this activity, you will:

Explore aspects of the application deployments

The following diagram shows what has been instantiated:



Let's validate this by listing the new pods in your cluster. In your terminal window execute:

$ kubectl get pods -o wide

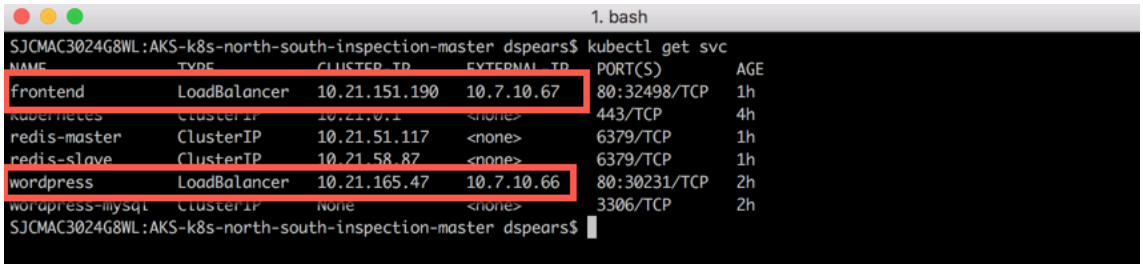
You should see the pods for both the WordPress and Guestbook Application:

```
SJCMAC3024G8WL:AKS-k8s-north-south-inspection-master dspears$ kubectl get pods -o wide
NAME                              READY     STATUS    RESTARTS   AGE       IP           NODE
frontend-67f65745c-km7d2          1/1       Running   0          1m        10.7.10.14   aks-default-56371607-1
frontend-67f65745c-vt8dp          1/1       Running   0          1m        10.7.10.11   aks-default-56371607-1
frontend-67f65745c-vttjj          1/1       Running   0          1m        10.7.10.46   aks-default-56371607-0
redis-master-7747787588-rl86s     1/1       Running   0          1m        10.7.10.6    aks-default-56371607-1
redis-slave-865486c9df-f252b      1/1       Running   0          1m        10.7.10.41   aks-default-56371607-0
redis-slave-865486c9df-l2i5h      1/1       Running   0          1m        10.7.10.34   aks-default-56371607-1
wordpress-8574f9c6f9-hm4cw        1/1       Running   0          49m       10.7.10.31   aks-default-56371607-1
wordpress-mysql-795bf5f54c-jqbgk  1/1       Running   0          59m       10.7.10.48   aks-default-56371607-0
```

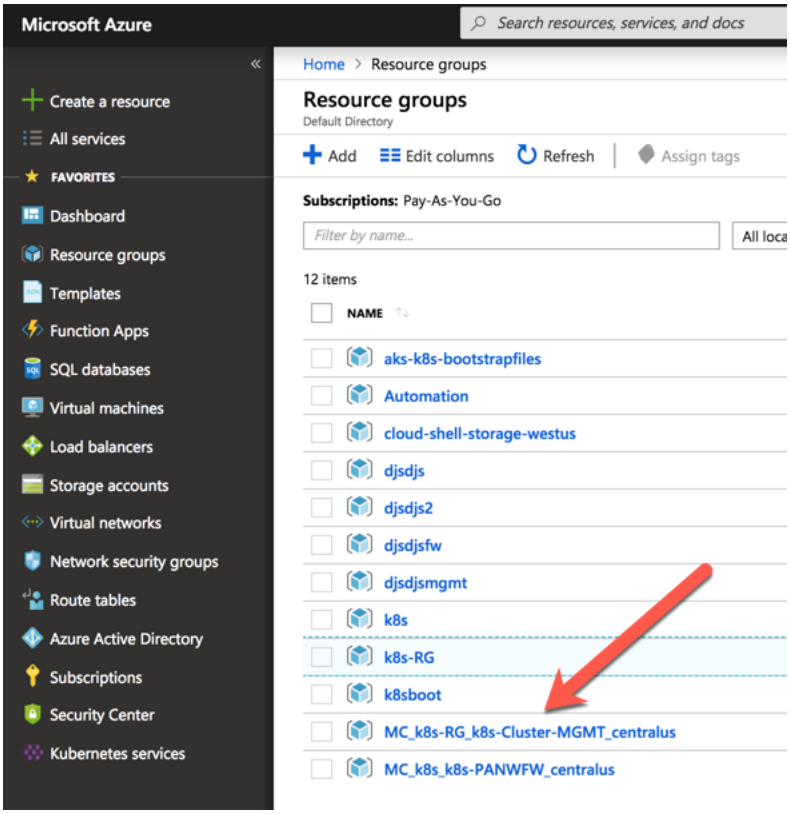
Next let's look at the load balancing service for the front-end pod. Execute the following command in the shell:

$ kubectl get svc

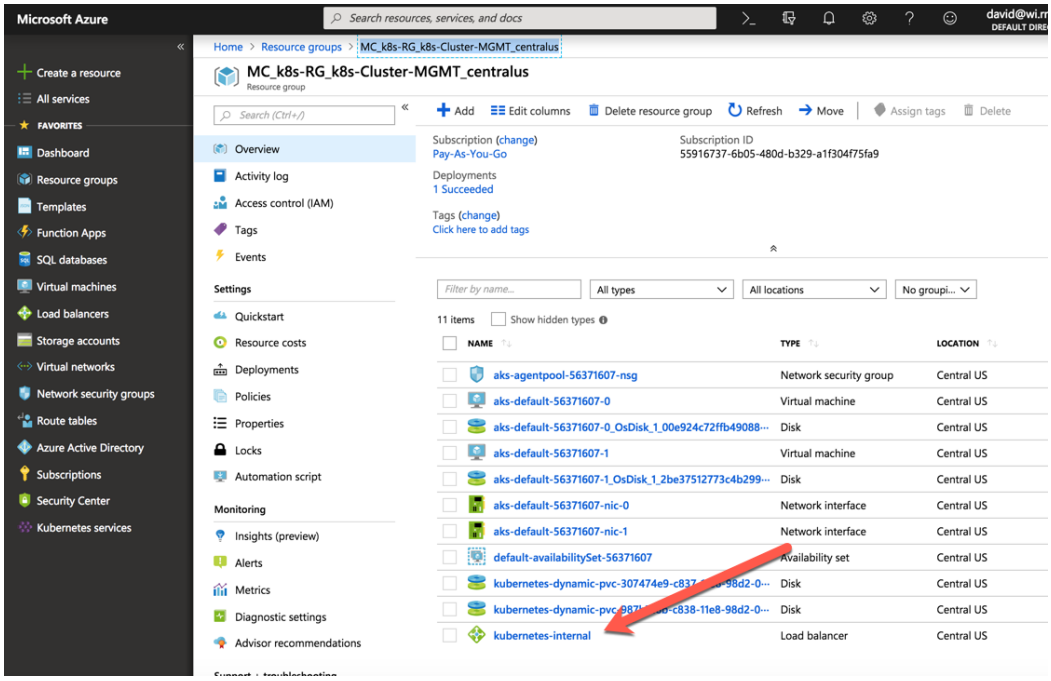
You can see there is a load balancer External IP for both the frontend Guestbook application and an External IP address for the WordPress server. Note that the IP address is in the 10.7.10.0/24 subnet. This is one of the subnets that was deployed in the Azure VNET during the Terraform execution.



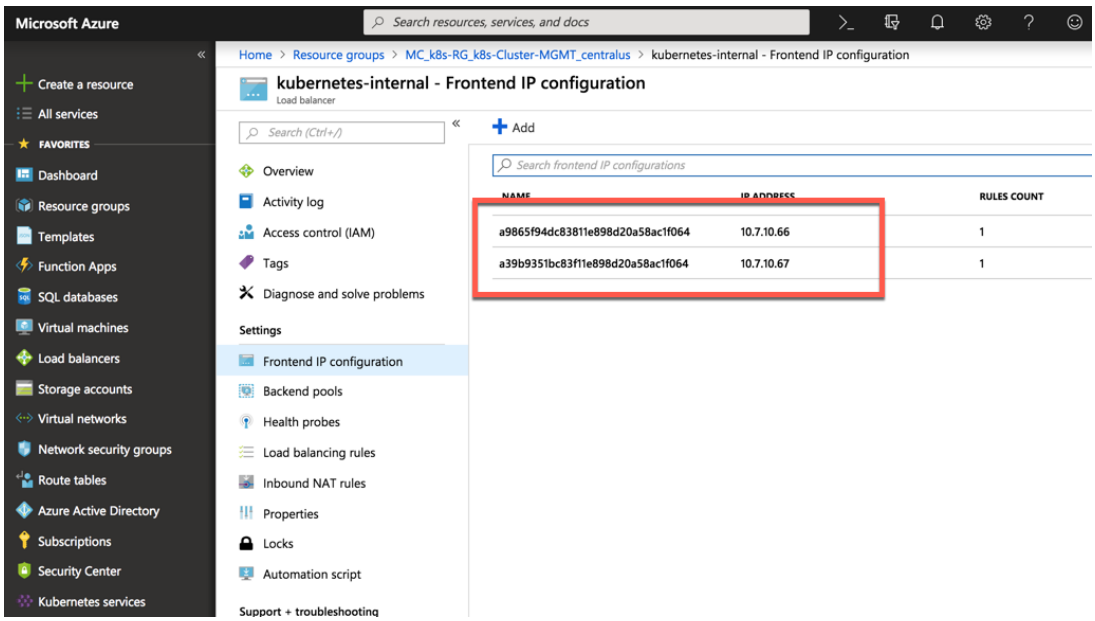
These load balancer IP addresses can be seen via the Azure Dashboard as well. Navigate to the Resource Groups and click on the "MC_k8s-RG_k8s-Cluster-MGMT_centralus" Resource group. This group was created automatically for the k8s node resources.



Click on the Kubernetes-internal Load balancer:



Click on the Frontend IP configuration on the left Nav. The application load balancer IP ADDRESS are displayed:



Securing Inbound Traffic

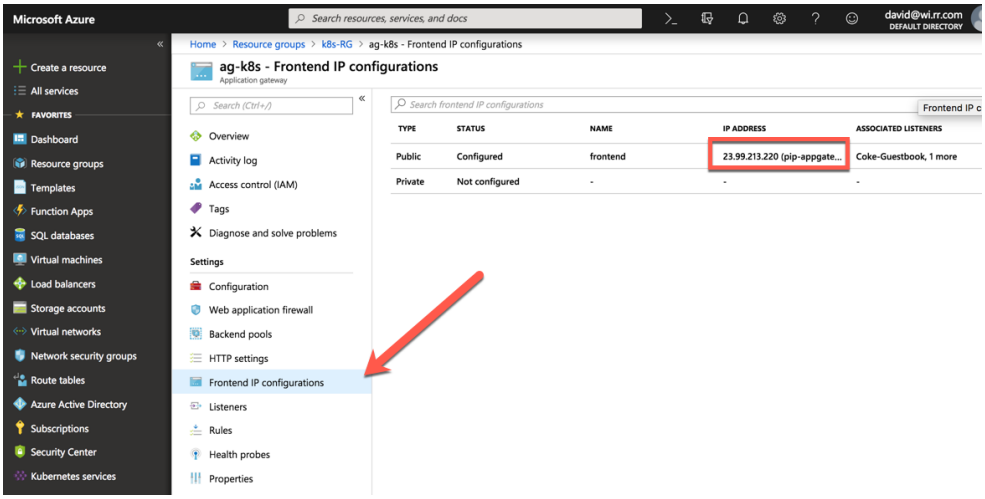
In this activity, you will:

Secure traffic that is inbound to your frontend services

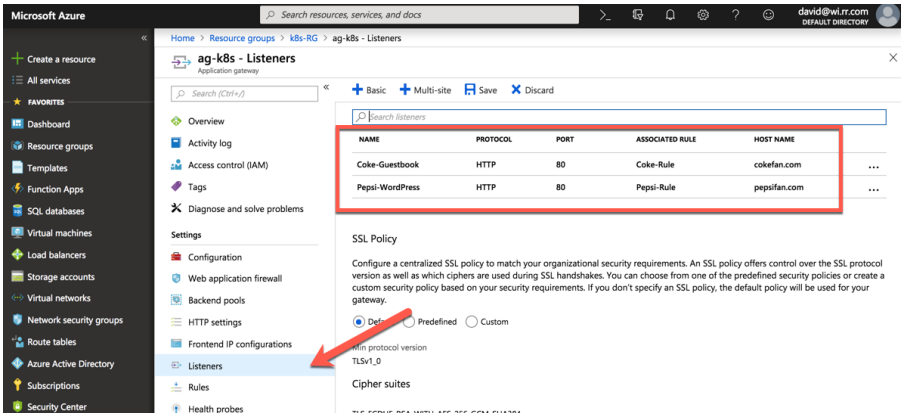
Validate that traffic is visible in the Firewall logs

Task 1 – Azure Application Gateway IP Address

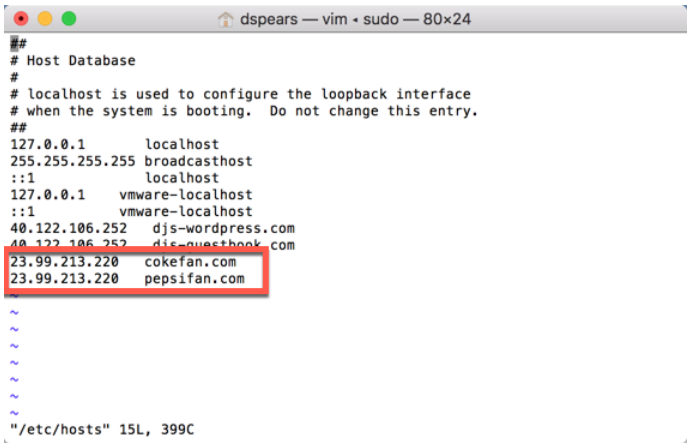
This Terraform deployment created an Azure Application gateway in front of the VM-Series firewall. As previous discussed, the Application Gateway is configured to do host header redirection. In order for this to function the frontend IP addresses must be identified and a few hosts entries need to be made on the testing machine. Open the Application Gateway Frontend IP configurations in the Resource groups > k8s-RG > ag-k8s blade:



Copy this address as it will be needed to create a DNS entry in the local host file. Go to Application Gateway Listeners on the left Nav to see the DNS entries that the Application Gateway is configured to serve.



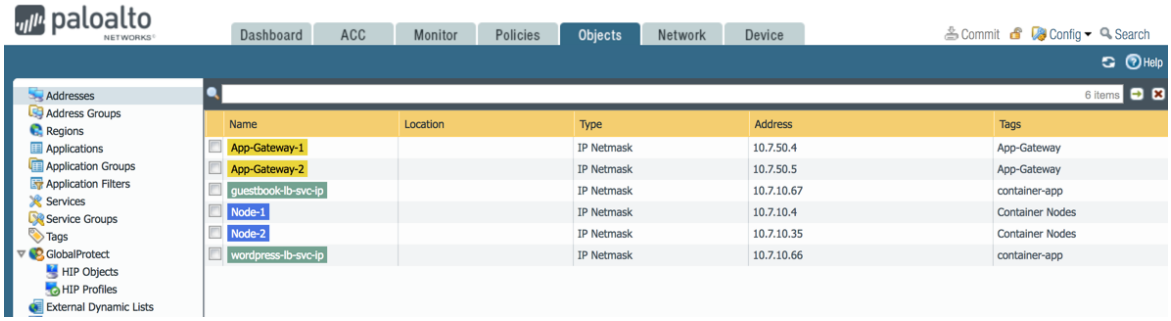
Open the local hosts file and create a pepsifan.com and cokefan.com entry. Each entry will have the IP address of the Application Gateway Frontend IP address:



Task 2 – Update the Firewall's Address Objects

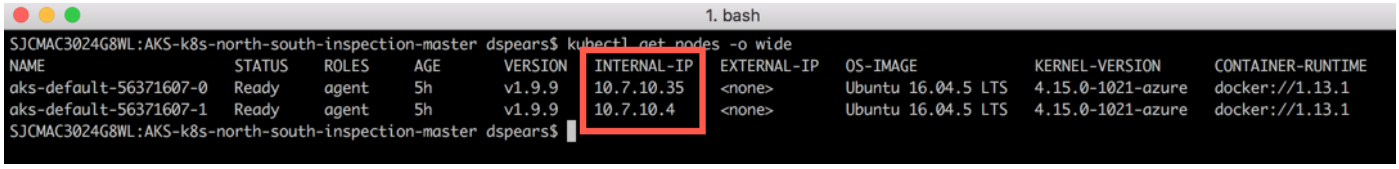
Open the VM-Series firewall. This design is not using any NATs for the inbound traffic flow. The bootstrapped configuration should have the correct addressing but this task will validate that.

Click the Objects Tab and navigate to "Addresses" on the left. The Addresses used in the policy are defined here:

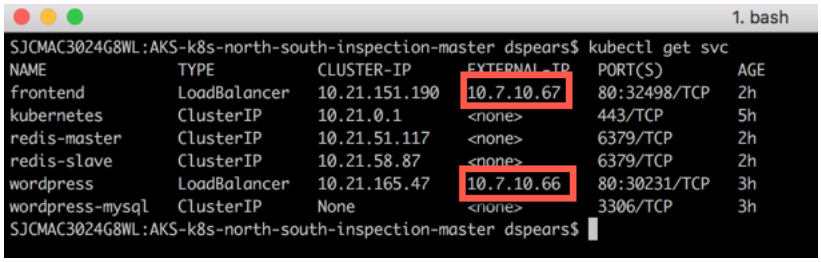


Open the terminal window and check that the Address objects are correct. Execute the following command to verify the nodes:

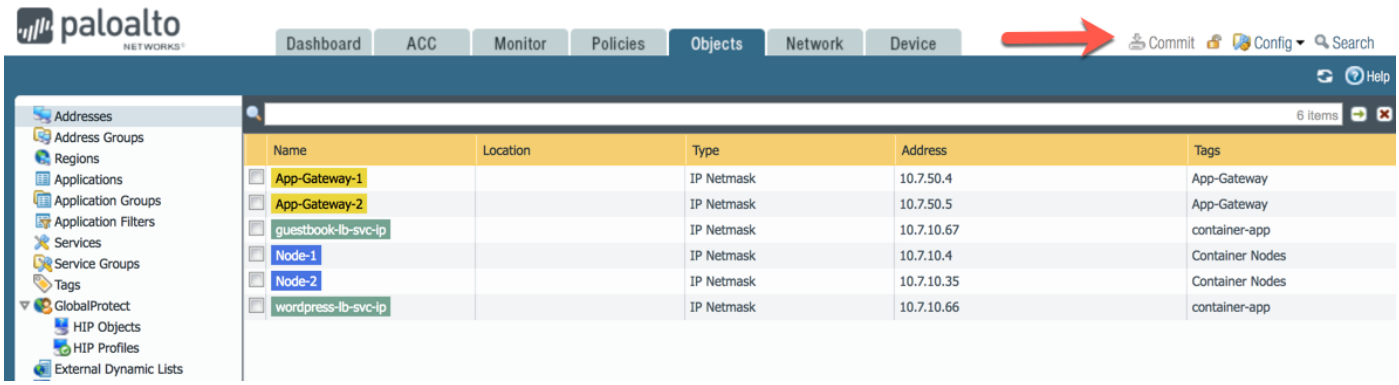
$ kubectl get nodes -o wide



Next enter the **"kubectl get svc"** command to verify the lb-svc-ip's:

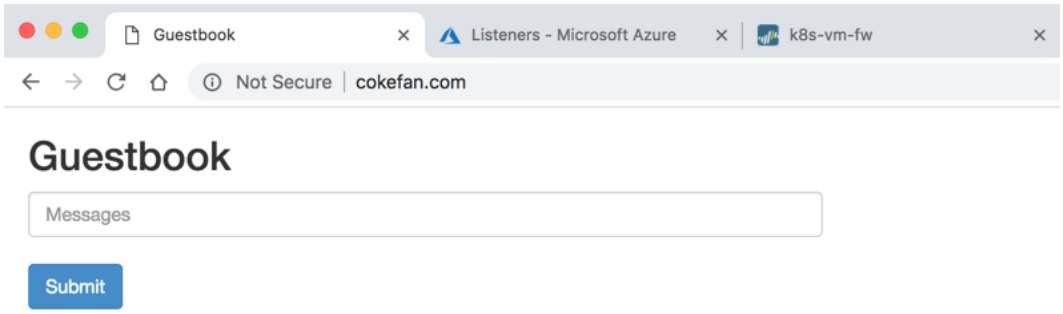


If a change is needed, make the changes and click the commit link on the top right

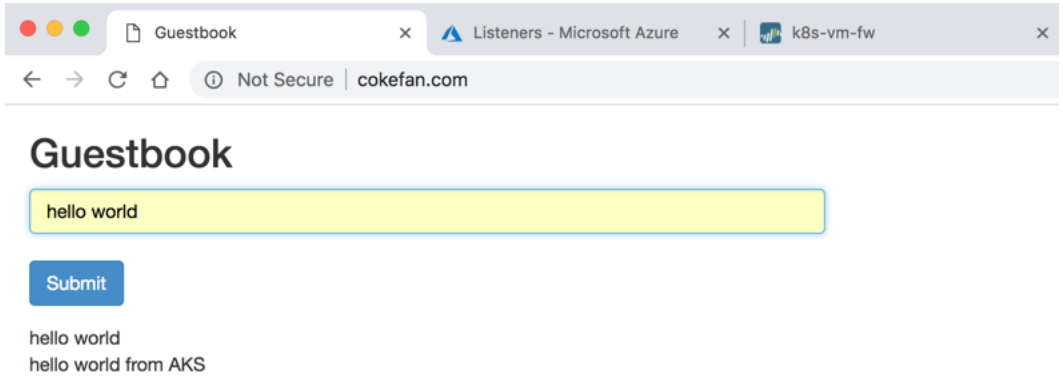


Task 3 – Connect to the Guestbook Frontend

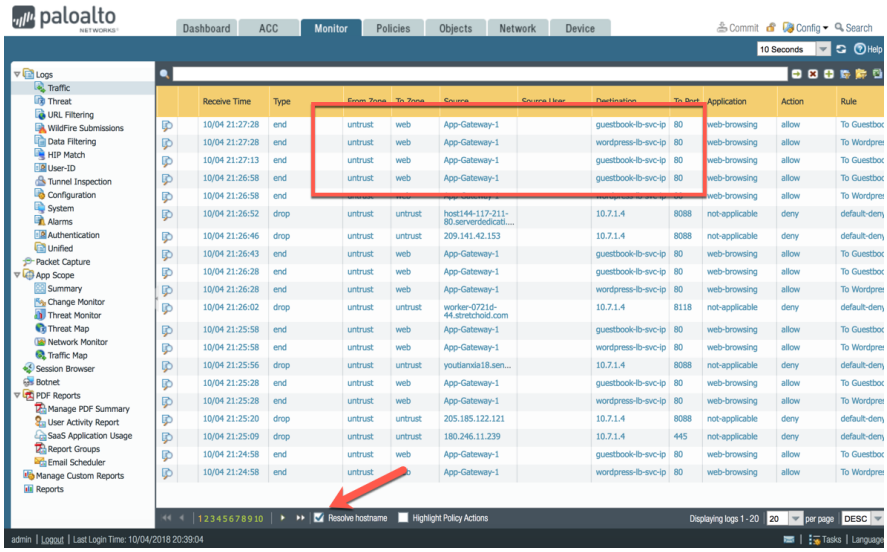
The VM-Series is now protecting your Kubernetes workload. In order to connect to the guestbook's frontend service, you will open a browser and navigate to the http://cokefan.com website:



Enter something in the Messages box and click submit. The messages should be echoed below:

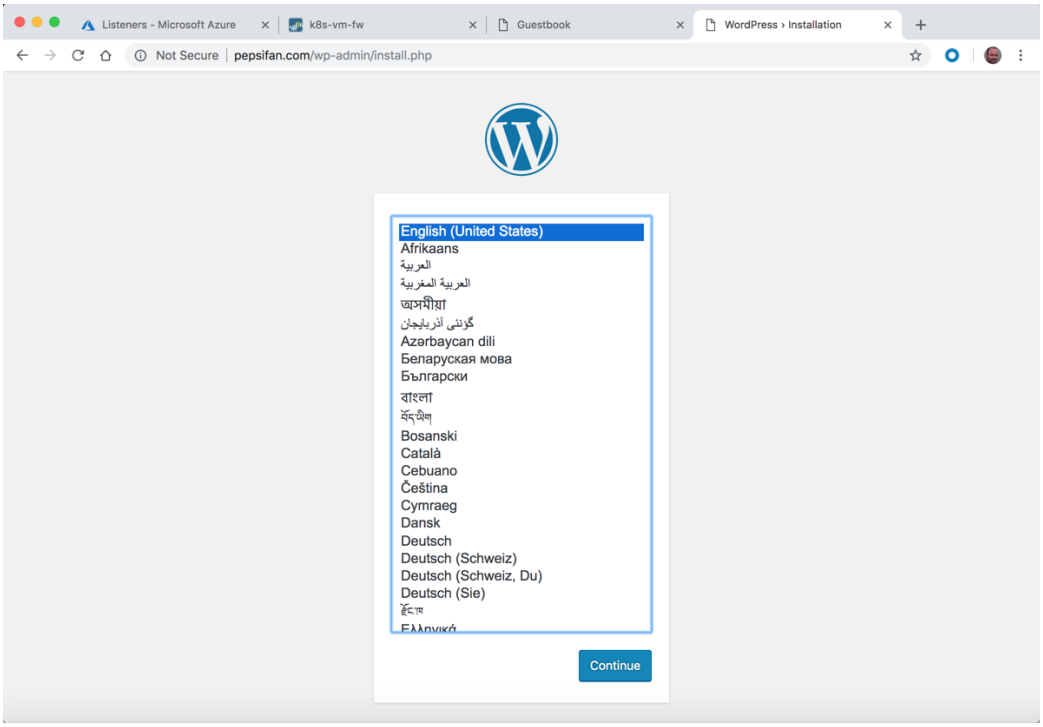


Open the VM-Series firewall monitor tab and validate that traffic is flowing through the firewall:



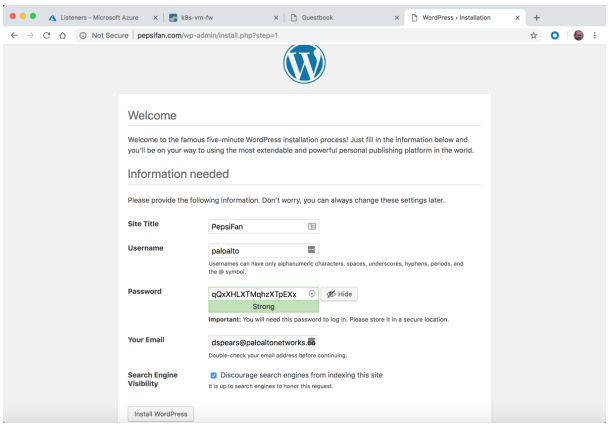
ProTip: Tick the Resolve hostname to make the logs more readable.

Now check that the http://pepsifan.com site works. Open a new tab and open the pepsifan.com site:

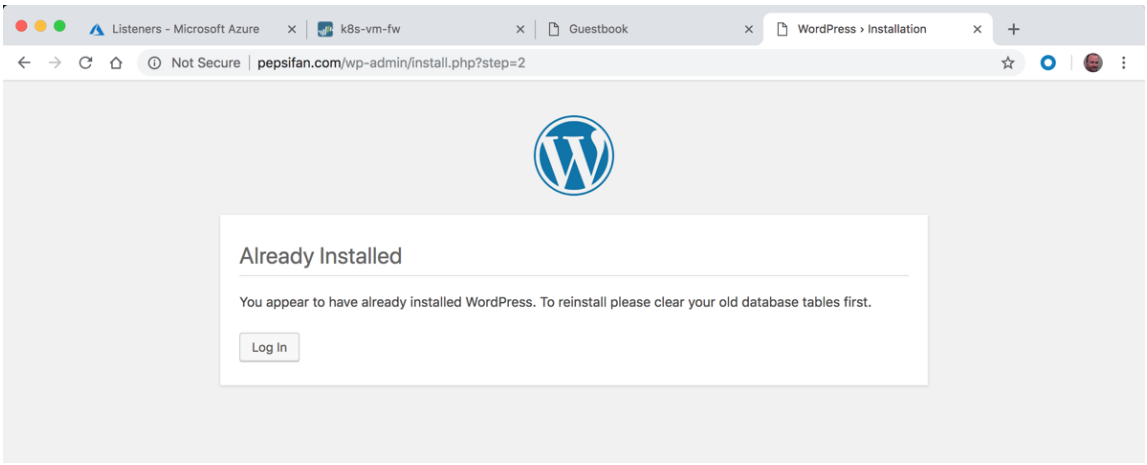


You may see the following error message. This is usually because the WordPress website takes a little time to get up and running. Click refresh a few times to get to the next step:

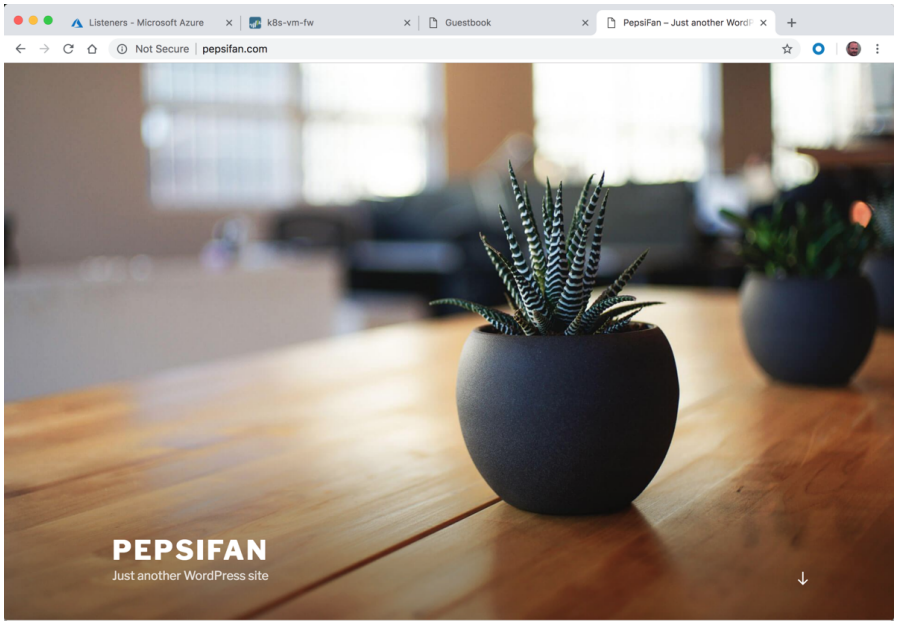
You should see the WordPress install page. Click Continue if you wish to go through the installation process:



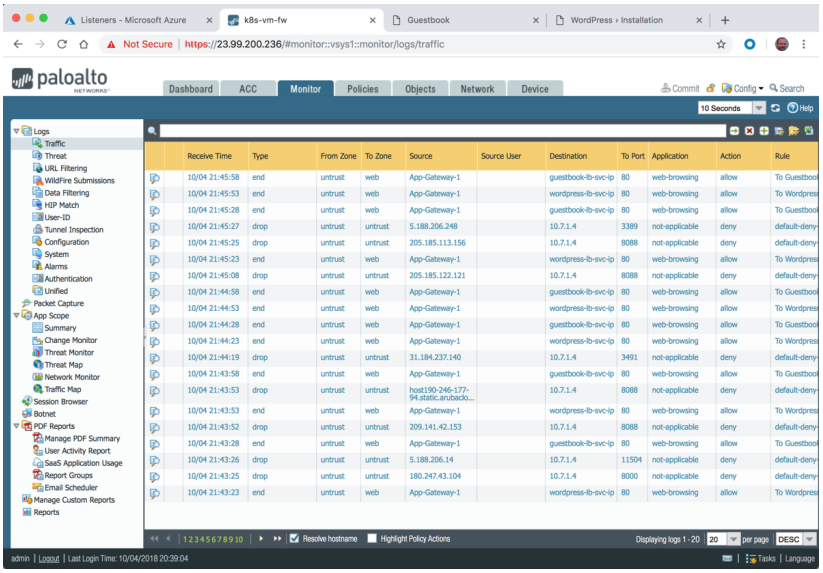
After pressing Install WordPress, you might see a 502 error from the Application Gateway. If you press refresh a few times you should see the following:



Go to the root of the http://pepsifan.com site and you should now see the default theme:



Verify that the pepsifan.com traffic is running through the firewall:



Securing Outbound Traffic

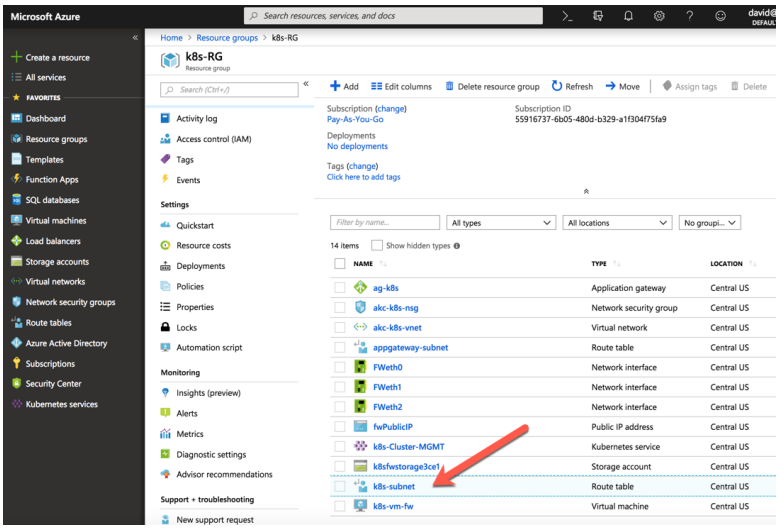
In this activity, you will:

Secure outbound traffic from the cluster nodes

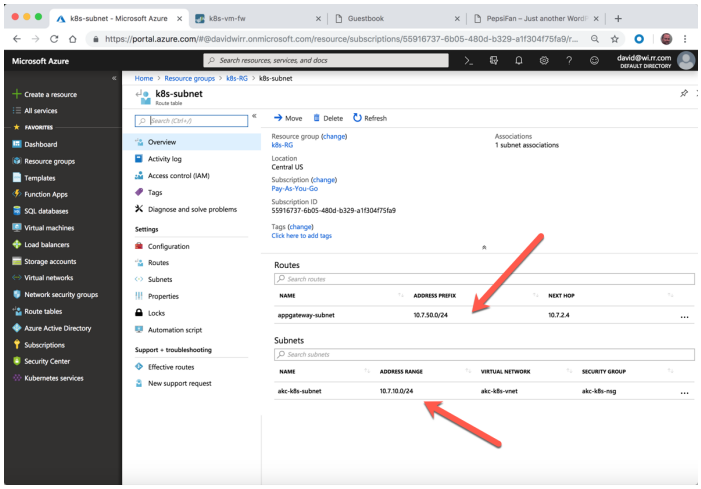
Validate traffic is in the Firewall logs

Task 1 – Add Outbound Route

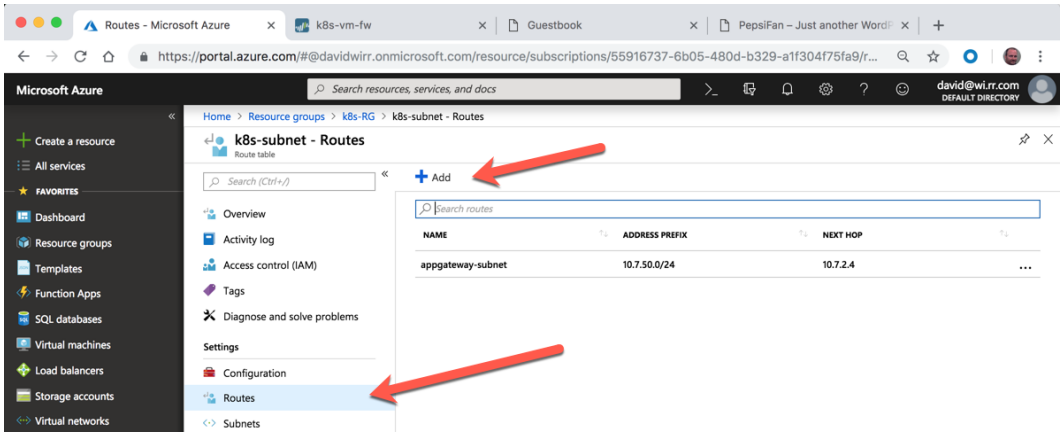
To secure any traffic that is originating from within the cluster we need to add a user defined route (UDR) to the routing table on the VNET subnet that the nodes are on. In this deployment that is the 10.7.10.0/24 subnet which is labeled akc-k8s-subnet. Navigate to the k8s-RG resource group and click on the k8s-subnet route tab:



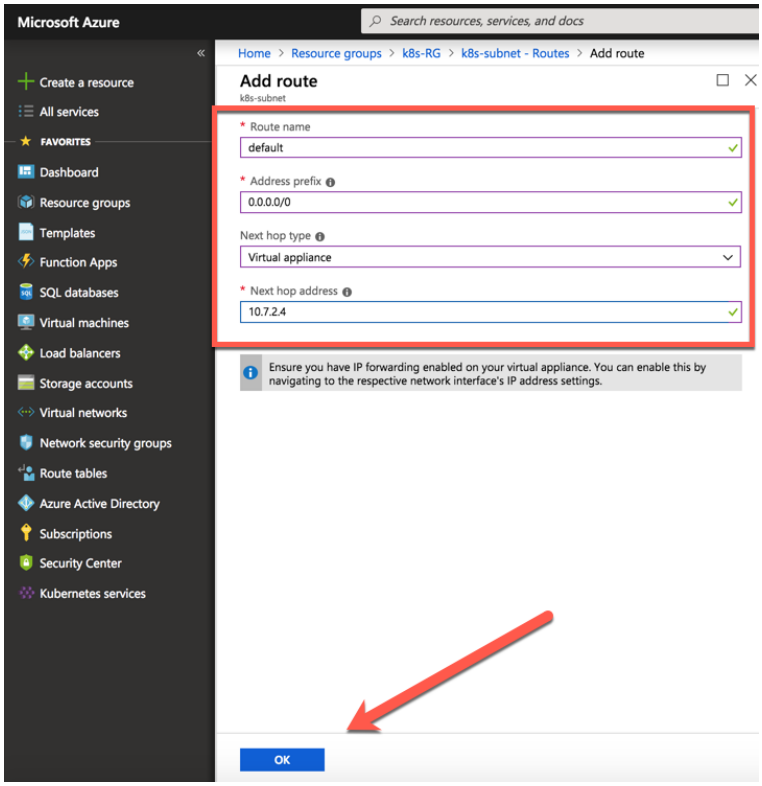
You can see a route to the app gateway subnet and that this is assigned to the 10.7.10.0/24 subnet:



Click on Routes on the left NAV and the click the “+Add” to add a new route:



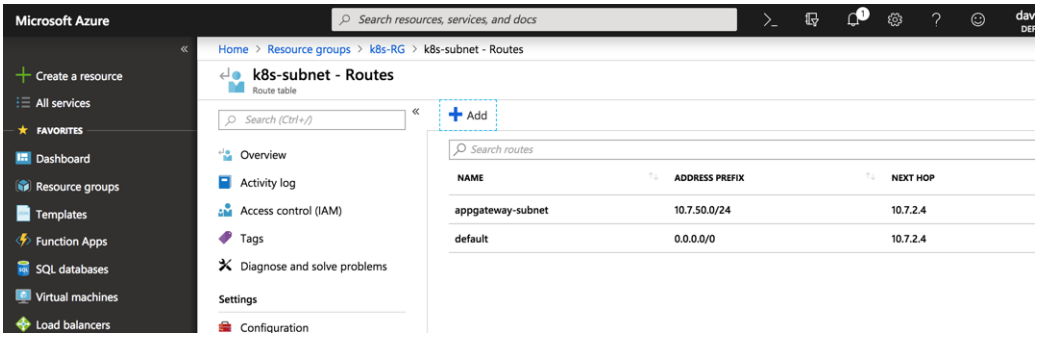
Create a route with the following Parameters:



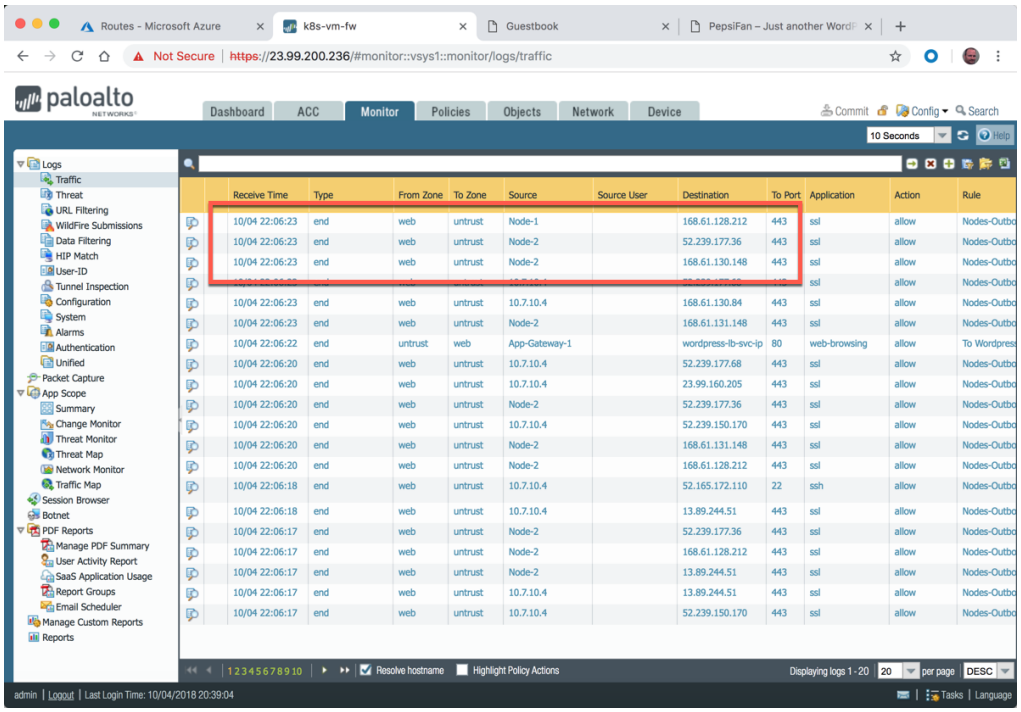
Route name: default
Address prefix: 0.0.0.0/0
Next hop type: Virtual Appliance
Next hop address: 10.7.2.4

And then click Create

The new route should appear in the list:



Navigate back to the firewall monitor tab and you can now see outbound traffic as well from the cluster nodes.

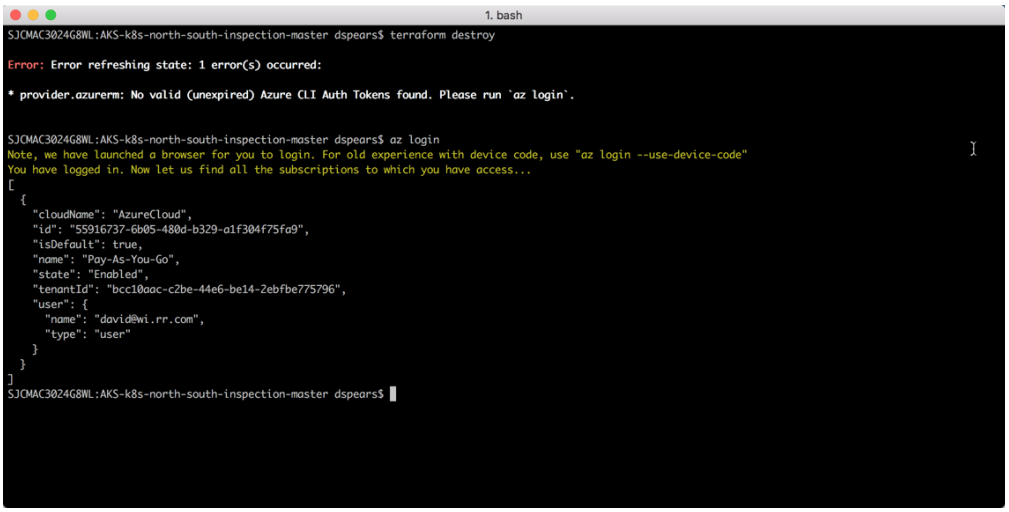


These source addresses are the instance addresses of the Kubernetes cluster node servers:

Lab Termination

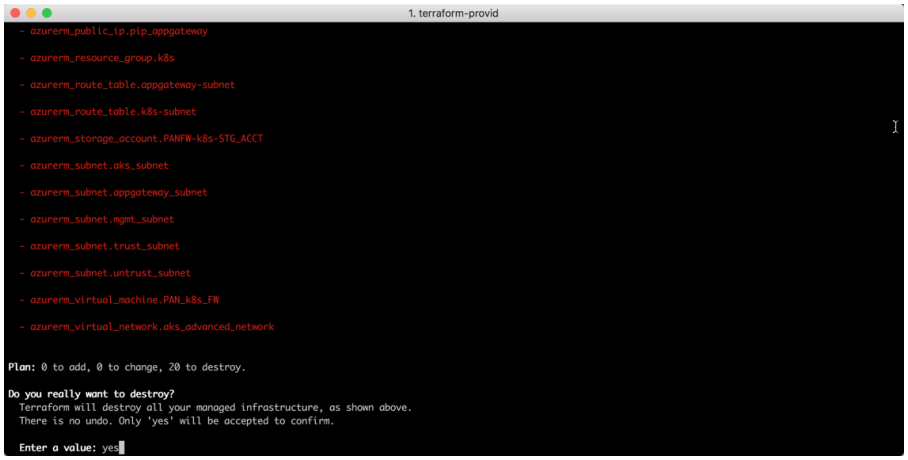
One advantage of Terraform is that it provides the ability to remove the deployment so it is not incurring ongoing cost but could be easily instantiated at a later time for testing and demonstrations. To destroy the lab, go to the terminal prompt and navigate to the directory that was used to deploy the environment and execute:

$ **terraform destroy**

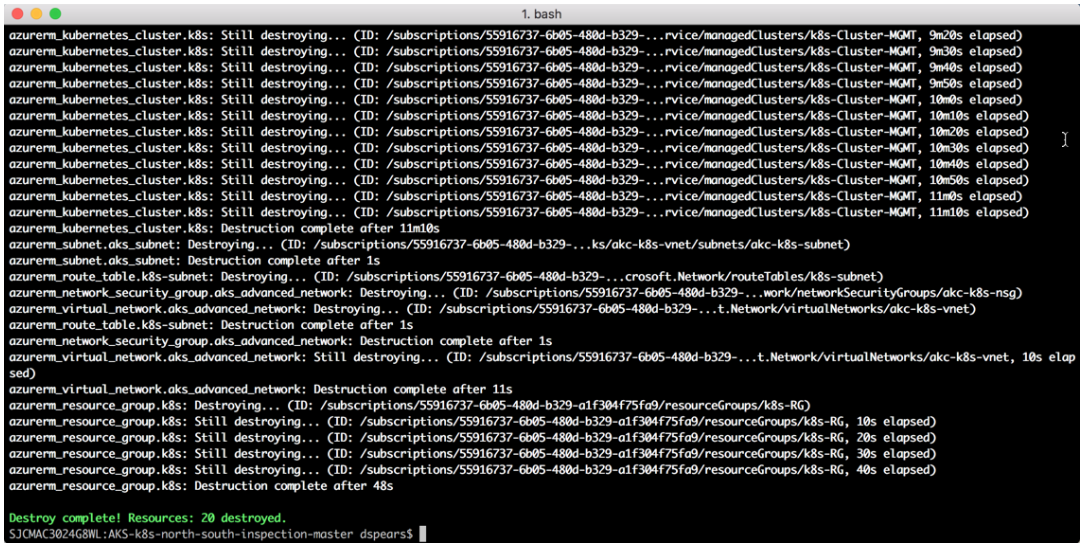


If an error message appears regarding the CLI Auth Tokens, run the **az login** command to get a new token.

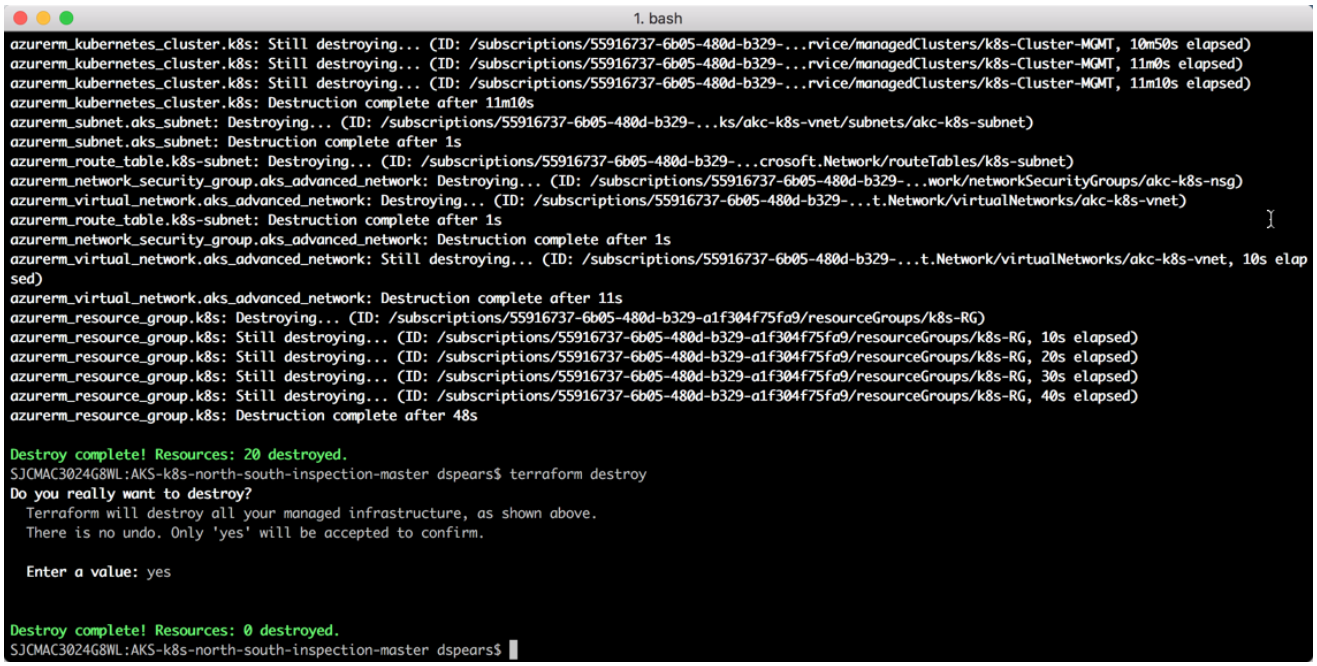
Terraform will show the list of items that will be removed. Type **yes** at the prompt to start the process:



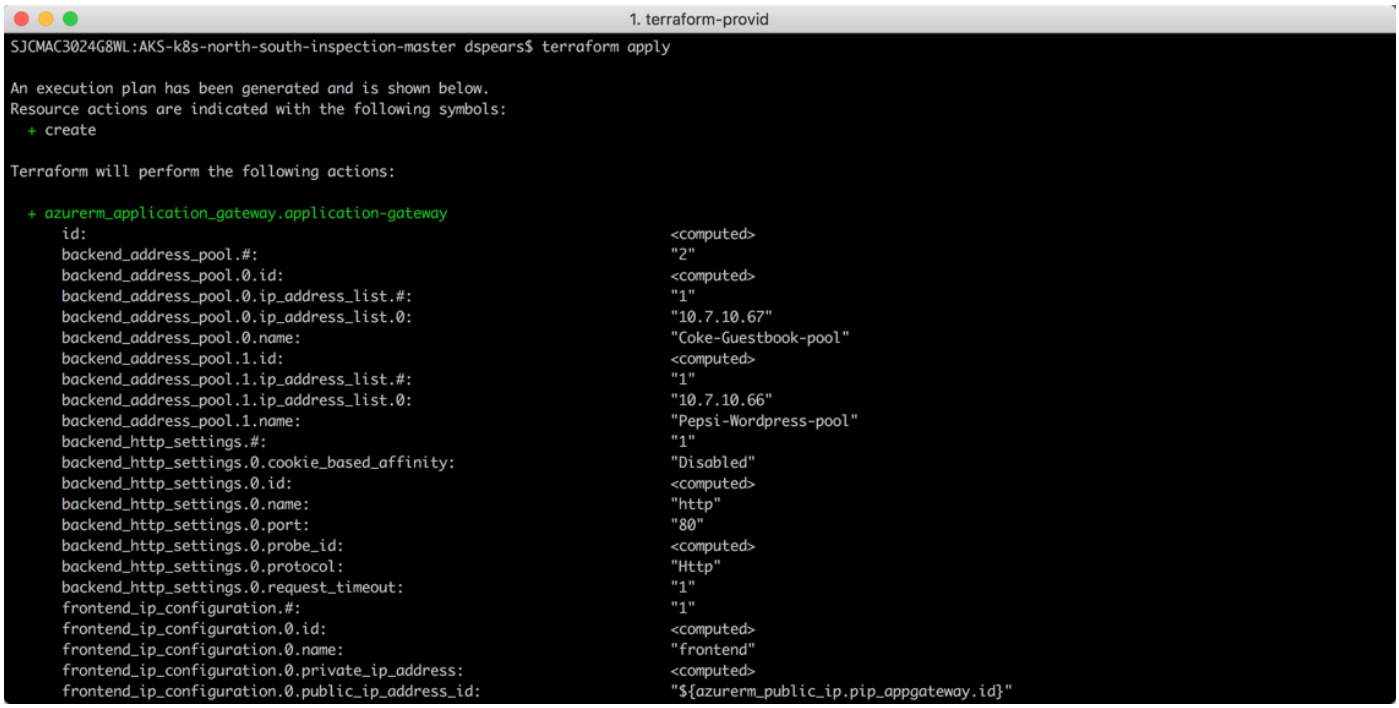
This should result in the complete removal of all the resources:



This can be validated by executing the **terraform destroy** command one more time:



At any point in the future, it is possible to come back to this directory and simply run the **terraform apply** command and quickly install the environment again:



End of Activity

Conclusion

Congratulations! You have now successfully integrated the VM-Series firewall to gain visibility into North/South traffic for two container application hosted in a Kubernetes cluster.