This guide assumes some basic familiarity with Kubernetes, Azure CLI/PS and kubectl but does not assume any pre-existing deployment.

It also assumes that you are familiar with the usual Terraform plan/apply workflow. For this tutorial, you will need.

* an [Azure account](https://portal.azure.com/#home)
* a configured Azure CLI or powershell
* Terraform module
* [kubectl](https://developer.hashicorp.com/terraform/tutorials/kubernetes/aks#kubectl) module
* Azure Storage explorer.
* Minecraft client
* Notepad++

1. **SET UP AND INITIALIZE YOUR TERRAFORM WORKSPACE**

In your terminal, clone the following repo. It contains the example configuration used in this workshop:

git clone <https://github.com/OzDoll/AKS-Workshop.git>

In here, you will find three files used to provision the AKS cluster.

1. [aks-cluster.tf](https://github.com/hashicorp/learn-terraform-provision-aks-cluster/blob/main/aks-cluster.tf) provisions a resource group and an AKS cluster. The default\_node\_pool defines the number of VMs and the VM type the cluster uses.
2. resource "azurerm\_kubernetes\_cluster" "default" {  
    name = "${random\_pet.prefix.id}-aks"  
    location = azurerm\_resource\_group.default.location  
    resource\_group\_name = azurerm\_resource\_group.default.name  
    dns\_prefix = "${random\_pet.prefix.id}-k8s"  
     
    default\_node\_pool {  
    name = "default"  
    node\_count = 2  
    vm\_size = "Standard\_B2s"  
    os\_disk\_size\_gb = 30  
    }  
     
    service\_principal {  
    client\_id = var.appId  
    client\_secret = var.password  
    }  
     
    role\_based\_access\_control\_enabled = true  
     
    tags = {  
    environment = " AKS Workshop"  
    }  
   }
3. [variables.tf](https://github.com/hashicorp/learn-terraform-provision-aks-cluster/blob/main/variables.tf) declares the appID and password so Terraform can use reference its configuration
4. [terraform.tfvars](https://github.com/hashicorp/learn-terraform-provision-aks-cluster/blob/main/terraform.tfvars) defines the appId and password variables to authenticate to Azure
5. [outputs.tf](https://github.com/hashicorp/learn-terraform-provision-aks-cluster/blob/main/outputs.tf) declares values that can be useful to interact with your AKS cluster
6. [versions.tf](https://github.com/hashicorp/learn-terraform-provision-aks-cluster/blob/main/versions.tf) sets the Terraform version to at least 0.14 and defines the [required\_provider](https://developer.hashicorp.com/terraform/language/providers/requirements" \l "requiring-providers) block

1. **CREATE AN ACTIVE DIRECTORY SERVICE PRINCIPAL ACCOUNT**

There are many ways to authenticate to the Azure provider. In this tutorial, you will use an Active Directory service principal account.

First, you need to create an Active Directory service principal account using the Azure CLI.

Run the following command:

az ad sp create-for-rbac --skip-assignment

Your output should be something like the following.

az ad sp create-for-rbac --skip-assignment  
{  
 "appId": "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa",  
 "displayName": "azure-cli-2019-04-11-00-46-05",  
 "name": "<http://azure-cli-2019-04-11-00-46-05>",  
 "password": "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa",  
 "tenant": "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"  
}

1. **UPDATE YOUR TERRAFORM.TFVARS FILE**

Replace the values in your terraform.tfvars file with your appId and password that you received from the previous step.

Terraform will use these values to authenticate to Azure before provisioning your resources. Your terraform.tfvars file should look like the following.

# terraform.tfvars  
appId = "xxxxxxxxxxxxxxxxxxxxxxxxx"  
password = "xxxxxxxxxxxxxxxxxxxxxxxxx"

### Initialize Terraform

After you have saved your customized variables file, initialize your Terraform workspace, which will download the provider and initialize it with the values provided in your terraform.tfvars file

To initialize terraform run:

terraform init

Your output should look something like:

terraform init  
Initializing the backend...  
  
Initializing provider plugins...  
- Reusing previous version of hashicorp/random from the dependency lock file  
- Reusing previous version of hashicorp/azurerm from the dependency lock file  
- Installing hashicorp/random v3.0.0...  
- Installed hashicorp/random v3.0.0 (signed by HashiCorp)  
- Installing hashicorp/azurerm v3.0.2...  
- Installed hashicorp/azurerm v3.0.2 (signed by HashiCorp)  
  
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

Number of changes may vary depending what your infrastructure looks like, if there’s nothing new to add, you’ll be notified as well. You can run this command as many times as needed to make sure everything is correct.

1. **PROVISION THE AKS CLUSTER**

After you have initialized your directory, execute the command "terraform apply" and carefully review the actions that will be taken. Your command prompt will display the progress of the execution and the resources that will be created.

Run the following command:

terraform apply

Your output should look something like:

terraform apply

An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:  
 + create  
  
Terraform will perform the following actions:  
  
 # ...  
  
Plan: 1 to add, 0 to change, 0 to destroy.  
  
 # ...

The command "terraform apply" will create an Azure resource group and an AKS cluster. You will be able to view the list of resources that will be created during this process before confirming it with a "yes" command. It typically takes around 5 minutes to complete. Once it is successful, you will see the outputs defined in the "aks-cluster.tf" file on your terminal.

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.  
  
Output should be similar to this, but with your own personal generated name:

kubernetes\_cluster\_name = light-eagle-aks  
resource\_group\_name = light-eagle-rg

1. **CONFIGURE KUBECTL**

Now that you've provisioned your AKS cluster, you need to configure kubectl.

Run the following command to retrieve the access credentials for your cluster and automatically configure kubectl.

Run the following:

az aks get-credentials --resource-group $(terraform output -raw resource\_group\_name) --name $(terraform output -raw kubernetes\_cluster\_name)

To generate the necesary Kubeconfig file, run in your command line the following:

az aks get-credentials --resource-group RGNAME --name AKSNAME --file kubeconfig

If you need to re-export them, simply run in your command line:

export KUBECONFIG='LOCATION OF YOUR FILE’

The resource group name and Kubernetes Cluster name correspond to the output variables showed after the successful Terraform run.

1. **ACCESS KUBERNETES DASHBOARD**

To verify that your cluster's configuration, visit the Azure Portal's Kubernetes resource view. [Azure recommends](https://docs.microsoft.com/en-us/azure/aks/kubernetes-dashboard" \l "start-the-kubernetes-dashboard) using this view over the default Kubernetes dashboard, since the AKS dashboard add-on is deprecated for Kubernetes versions 1.19+.

Run the following command to generate the Azure portal link.

az aks browse --resource-group $(terraform output -raw resource\_group\_name) --name $(terraform output -raw kubernetes\_cluster\_name)

Go to the URL in your preferred browser to view the Kubernetes resource view.

**Now to section 2 of our workshop: Let's setup a Minecraft server in our already created AKS cluster.**

1. **REGISTER AKS RESOURCE**

First, we need to register the AKS Resource Provider under your subscription:

az provider register --namespace 'Microsoft.ContainerService'

1. **CREATE FILESHARE**

We intend to create a File Share on this storage account so that our Minecraft server files can be stored and accessed easily by both the container and us through any Windows or Linux PC. This approach enables us to restart or delete a container without losing our Minecraft files or worlds.

Additionally, we can mount the file shares on Windows or Linux systems and transfer files easily, making configuration changes and uploading worlds much simpler. Although other options exist for storing persistent data on virtual disks, we prefer this method because we will be accessing the data frequently to manually move files around, given the nature of the application.

When selecting a name for the Storage Account, keep in mind that it must be unique and can only contain lowercase letters and numbers, with a length between 3-24 characters. If the account name is already in use, a message will appear in red letters. To replace the existing name, we have chosen a creative name, ' minecraft123'. We can use the same resource group where we plan to keep the AKS cluster.

az storage account create --name minecraft123 --resource-group RESOURCEGROUPNAME

We must specify the Resource Group and Name of the Storage Account that we have just created, followed by the name of the file share. In this case, I will use the name 'sharename' in lowercase letters.

Additionally, I have chosen to use the 'TransactionOptimized' tier since it is the most cost-effective for our Minecraft server, given the low storage space used and high transactions generated by Minecraft.

The quota limit for this tier will be 2GB, which is more than sufficient. It's worth noting that quotas can be adjusted after creation, so there's no need to be overly concerned about this value.

Now, we will require to create a file share for our AKS cluster to access:

az storage share-rm create --resource-group MyResourceGroup --storage-account minecraft123 --name sharename --access-tier "TransactionOptimized" --quota 2

Next, we need to gather the access keys of the Storage account, those are important in configuring how we're going to mount them.

az storage account keys list --account-name minecraft123

It should look like:

[

{

"creationTime": "2023-03-08T18:38:51.760700+00:00",

"keyName": "key1",

"permissions": "FULL",

"value": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

},

{

"creationTime": "2023-03-08T18:38:51.760700+00:00",

"keyName": "key2",

"permissions": "FULL",

"value": " xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx "

}

]

Now, let's upload some files to our file share. There are 2 options we can do regarding this. We can either use the File share portal, or [Azure Storage Explorer](https://azure.microsoft.com/en-us/products/storage/storage-explorer)

You will upload the provided script in the repo: <https://github.com/OzDoll/AKS-Workshop/blob/main/minecraft-container-scripts/main-script.sh>

Ein Bild, das Text, drinnen, Monitor, Bildschirm enthält.

Automatisch generierte BeschreibungFrom the portal

Ein Bild, das Text, Monitor, Screenshot, schwarz enthält.

Automatisch generierte BeschreibungFrom the Azure Storage Explorer, we have access to all storage drives under the subscription we're working with. This comes useful to upload files in bulk, and in case the script to unpack our Minecraft server fails.

Alternatively, you can mount the drive to your Laptop using the connect command available under the Fileshare options.

1. **YAML DEPLOYMENT CONFIGURATION**

A YAML file is a text-based format that is specifically structured to specify configuration-related data. While it requires precise formatting, it can be utilized to deploy any resource under an AKS cluster. Although alternative tools exist, this guide will solely focus on YAML since it is the most straightforward method and doesn't require third-party utilities.

I will provide guidance on creating a single YAML file that encompasses all the necessary configurations for deploying our container and running our Minecraft server. Alternatively, you could create multiple files for each section, but by adding " - - - " (three dashes) between the configurations, we can add them all in one file, and they will be perceived as separate deployments. For this deployment, I will use the 'default' namespace. However, you can modify it if necessary, but ensure that all resources are under the same namespace.

The complete Yalm is available at the repo in case you need to review and validate.

1. Encrypt storage secret keys in Base 64.

In Kubernetes, secrets are a type of object that is designed to store sensitive information, such as passwords, tokens, or other confidential data, used by applications or services deployed on a cluster. The 'Secret' is like a 'key' that grants access to the storage account and allows our container to mount the file share inside of it. Without this 'key,' the container may run, but it won't be able to access the file share or any of its files.

Now that we have comprehended the purpose of a 'Secret,' the YAML configuration and deployment cannot permit us to merely copy and paste the storage account name and key as plain text. If anyone has access to the code, they can easily read it and exploit it for their own purposes. Therefore, we must 'conceal' the information and encode it as base64, as this is the default expectation.

echo -n 'minecraft123' | base64

echo -n 'HERE-GOES-YOUR-' | base64

If you’re on a windows computer, use: https://www.base64encode.org/

This is the YAML configuration for the 'Secret.' Replace the Storage Account name/key with your own Base64-encoded outputs for each value. Pay attention to the 'name' that we assign to our 'Secret' since we'll need it for the upcoming configurations.

apiVersion: v1

kind: Secret

metadata:

  name: storage-secret

  namespace: default

type: Opaque

data:

  azurestorageaccountname: xxxxxxxxxxxxxxxxxxxxxxxx

  azurestorageaccountkey: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx==

1. Persistent Volume configuration

Next up is the configuration for PersistentVolume. This will include details about the File Share we'll be using and any special permissions we may want for it.

The items we need to take note of and modify are highlighted in bold:

* **'name'** is the name we want to give to the PersistentVolume we're creating.
* **'storageClassName'** is simply the name of the storage class we'll be using for this PV. Both sections must match.
* **'storage'** is the capacity we're allocating for the share. You can set it to the same as your 2GB quota.
* **'secretName'** must match the name of the Secret we created in the previous section.
* **'shareName'** is the name of the File Share you created within your Storage Account.

We also have the 'mountOptions' section, but it will remain as I have set them. These are the permissions that the Mount Point will use, and I've configured them to use the Minecraft User ID (UID/GID) that will be created later, as well as 750 folder and file permissions.

apiVersion: v1

kind: PersistentVolume

metadata:

  name: pvname

spec:

  storageClassName: "scname"

  capacity:

    storage: 2Gi

  accessModes:

  - ReadWriteMany

  storageClassName: scname

  mountOptions:

  - dir\_mode=0750

  - file\_mode=0750

  - uid=1001

  - gid=1001

  azureFile:

    secretName: storage-secret

    secretNamespace: default

    shareName: sharename

    readOnly: false

1. Persistent Volume Clain configuration

We now move on to configuring the Persistent Volume Claim (PVC) which will be utilized by our pod/application to "request" the PV. The PVC has only a few sections that we need to focus on.

The "name" section is where we can assign a name for this PVC. The "storageClassName" must match the same name that we created for the PV above. Finally, "storage" specifies the 2GB limit once again.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: pvclaim

spec:

  accessModes:

    - ReadWriteMany

  storageClassName: "scname"

  resources:

    requests:

      storage: 2Gi

1. Minecraft server deployment

We now have the deployment configuration for the Minecraft server. The configuration has several 'name' sections for Metadata/Labels, and 'app' for labels/containers that must match. These are only for the deployment/pod's name and labels. You may change the name if needed, but ensure that they are identical.

Next, we must modify the sections under 'VolumeMounts' and 'Volumes' to match the names of the File Share, Persistent Volume, and Persistent Volume Claim we previously created. The POD will use all the previous items created to complete the mount from the Secret to the PV/PVC, etc.

There are a few other sections that must remain unchanged. We will use the Ubuntu image provided by Docker Hub, however, feel free to choose the image you feel more comfortable with, as long as it’s Linux based. With the configurations that we have in place in this YAML and the main-script.sh file, we do not need to create a custom container image and pull it from a Container Registry.

The configuration contains two commands (args) that will be executed during container creation. The first one will create the user Minecraft and execute the main-script.sh command, which will take care of everything else. The configuration also includes the ports required by Minecraft for connection and the mount path for the File Share, which will be mounted under /home/Minecraft.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: minecraft-server

  labels:

    app: minecraft-server

spec:

  replicas: 1

  selector:

    matchLabels:

      app: minecraft-server

  template:

    metadata:

      labels:

        app: minecraft-server

    spec:

      restartPolicy: Always

      containers:

      - name: minecraft-server

        image: docker.io/library/ubuntu:20.04

        command: ["/bin/sh"]

        args: ["-c", "useradd -u 1001 Minecraft ; sh /home/Minecraft/main-script.sh"]

        ports:

        - containerPort: 19132

        - containerPort: 19133

        volumeMounts:

        - name: sharename

          mountPath: /home/Minecraft

      volumes:

      - name: sharename

        persistentVolumeClaim:

          claimName: pvclaim

1. **EXPOSE THE SERVICE AND LOAD BALANCING.**

The final section is about the Load Balancer that will manage the Public IP used for connecting to the Minecraft server and routing the traffic to/from the Minecraft container. The "name" specifies the name of the LB service, while "app" refers to the name of the Pod/Deployment, as mentioned earlier. Lastly, "loadBalancerIP" specifies the Public IP used for connecting to Minecraft.

apiVersion: v1

kind: Service

metadata:

  name: mine-loadbalancer

spec:

  type: LoadBalancer

  ports:

  - port: 19132

    protocol: UDP

    name: udp-19132

  - port: 19133

    protocol: UDP

    name: dup-19133

  selector:

    app: minecraft-server

  loadBalancerIP: xx.xxx.xxx.xx

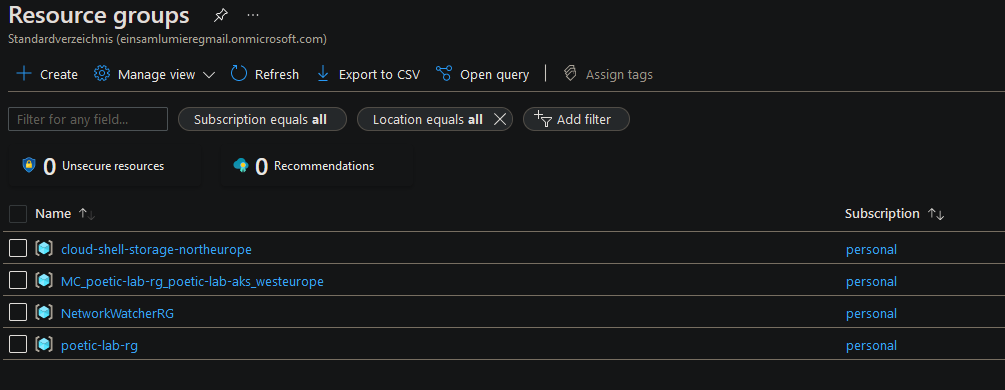
As we don't have a Public IP yet, let me guide you on creating a static public IP with a basic SKU that matches our LB. The IP must be created in the same resource group where we have our cluster resources.

In AKS, the resource group of AKS is different from the resource group created to hold all AKS resources, such as VMSS, vnet, IPs, etc. You can find the MC\_ resource group name by accessing your AKS cluster and clicking on 'Properties'

Ein Bild, das Text, Screenshot, Monitor, Bildschirm enthält.

Automatisch generierte Beschreibung

Or, you can directly go to Resource Groups and look for the associated MC RG.



To create our standard load balancer run:

az network public-ip create --name minecraft-basic-ip --resource-group YOURRESOUCEGROUP --allocation-method Static  --sku standard

Once the operation is finished, you will be presented with a lengthy output. Among this output, you can find the Public IP address that you need to use for your load balancer. The output would look similar to this:

                {

                "publicIp": {

                    "etag": "W/\"afe0e825-cddb-470f-b0b9-e8f76e83a3b4\"",

                    "id": "/subscriptions/7af4bacd-05c8-4c53-b0da-d4544ac35408/resourceGroups/MC\_poetic-lab-rg\_poetic-lab-aks\_westeurope/providers/Microsoft.Network/publicIPAddresses/minecraft-basic-ip",

                    "idleTimeoutInMinutes": 4,

                    "ipAddress": "xx.xx.xx.xx",

                    "ipTags": [],

                    "location": "westeurope",

                    "name": "minecraft-basic-ip",

                    "provisioningState": "Succeeded",

                    "publicIPAddressVersion": "IPv4",

                    "publicIPAllocationMethod": "Static",

                    "resourceGroup": "MC\_poetic-lab-rg\_poetic-lab-aks\_westeurope",

                    "resourceGuid": "xxxxxxxxxxxxxxxxxxxxx7",

                    "sku": {

                    "name": "Basic",

                    "tier": "Regional"

                    },

                    "type": "Microsoft.Network/publicIPAddresses"

                }

                }

Replace the loadBalancerIP: xx.xxx.xxx.xx with the value from the IP address you received from the command.

1. **YAML DEPLOYMENT**

After completing all the necessary configurations, go to Cloud Shell and create a file where you can save the configurations. You can use either nano or vi to create the file, let's say the file is named "minecraft.YAML". Copy and paste all the configurations into the file and save it. Alternatively, use notepad++ and save it in the same directory you will be running the deployments from.

We can deploy our server either in the default namespace, or, we can further isolate it for security purposes. For that, we’ll create a new namespace, as we want to make sure there’s isolation between our deployment, and the rest of our K8s infrastructure:

Kubectl create namespace minecraft

As we created a new namespace, we need to chance the proper values in our deployment yaml file to match. In your yaml file look for:

 namespace: default  
and  
 secretNamespace: default

and replace it with

  namespace: minecraft

and  
 secretNamespace: minecraft

Go to the folder where your deployment is stored and run:

If using the default namespace:

Kubectl apply -f minecraft.YAML

If using the Minecraft namespace:

Kubectl apply -f minecraft.YAML -n Minecraft

If your deployment is correctly configured, you should see the following:

damarquez@DESKTOP-J7LQMAH:/mnt/c/Users/damarquez/Documents/GitHub/minecraft-container-scripts$ kubectl apply -f minecraft.YAML

secret/storage-secret created  
persistentvolume/pvname created  
persistentvolumeclaim/pvclaim created  
deployment.apps/minecraft-server created  
service/mine-loadbalancer created

To remove the deployment and associated services that we have created, you can execute the following command:

Kubectl delete -f minecraft.YAML -n minecraft

1. **LET’S MAKE SURE IT’S ALL RUNNING!**

Let’s run a couple useful commands and check if everything’s running correctly. Execute the following command to display a list of all the currently running pods.

If you used a namespace other than the default one, you need to include the ‘-n namespace’ option in the command, otherwise it will display everything in the default namespace automatically.

kubectl get pods -n minecraft

Your output would look like:

damarquez@DESKTOP-J7LQMAH:/mnt/c/Users/damarquez/Documents/GitHub/minecraft-container-scripts$ kubectl get pods -n default

NAME READY STATUS RESTARTS AGE

minecraft-server-c6c6c666f-qsmqf 1/1 Running 0 60m

To check the logs of our server and make sure it’s all running, run:

kubectl logs <pod-name> -n <namespace>

You will be able to observe the progress of each step and ultimately the Minecraft server will initiate.

The initial time you run this, or when an upgrade is necessary, a zip file will be obtained from <https://www.minecraft.net/en-us/download/server/bedrock> and extracted within your container and consequently your File Share. The process may require up to 20 minutes, so it's important to remain patient. If there are no upgrades required, the server will start in approximately 2 minutes.

If everything’s running correctly, your output should look like this:

damarquez@DESKTOP-J7LQMAH:/mnt/c/Users/damarquez/Documents/GitHub/minecraft-container-scripts$ kubectl logs minecraft-server-c6c6c666f-qsmqf -n default

Fri Mar 10 09:28:55 UTC 2023 - Writing down latest updated version for future checks.

Fri Mar 10 09:28:55 UTC 2023 - Starting the Minecraft server! Almost there...NO LOG FILE! - setting up server logging...  
[2023-03-10 09:29:16:709 INFO] Starting Server  
[2023-03-10 09:29:16:709 INFO] Version 1.19.63.01  
[2023-03-10 09:29:16:709 INFO] Session ID e565fc6c-1530-4882-aeff-1a5a64f58062  
[2023-03-10 09:29:16:858 INFO] Level Name: Bedrock level  
[2023-03-10 09:29:17:465 INFO] Game mode: 0 Survival  
[2023-03-10 09:29:17:465 INFO] Difficulty: 1 EASY  
[2023-03-10 09:32:21:281 INFO] opening worlds/Bedrock level/db  
[2023-03-10 09:34:18:395 INFO] IPv4 supported, port: 19132: Used for gameplay and LAN discovery  
[2023-03-10 09:34:18:395 INFO] IPv6 supported, port: 19133: Used for gameplay  
[2023-03-10 09:34:22:306 INFO] Server started.

To connect to your Minecraft server, use the IP address assigned to the load balancer and start playing. A default world will be created and anyone can connect and play with the IP address.

If you want to upload your custom worlds, edit the server properties, permissions, and whitelistings. For that porpuse, go back to the Fileshare section, and use either option as you see fit.

Open your Minecraft client, go to multiplayer and add the IP of your LB to it, and voila!

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Baum, Screenshot enthält.

Automatisch generierte Beschreibung

If you have any issues, let me know.

Important:

To Destroy the whole AKS cluster as costs will occur. To do so run the following command on your bash:

**terraform destroy**

When done, do check your Azure Portal for any left resources. The process takes around 5 minutes.

**Extra:**

## **7 CrobJobs – Minecraft Restart and Updates**

There are multiple ways, and way easier, to go about frequent checks to see if there is a new Minecraft version. One quick option is to simply modify the sleep at the end of the main-script.sh file if you don’t want to set up Cronjobs, when the sleep ends the container will complete and restart; and the version check will be performed at start, but since we are learning about AKS why not learn about Crobjobs?

The main-script.sh file has been created in a way that every time the container starts, it checks if a new Minecraft version has been released, if it does not find a new version it will simply start the server as is, but if it does detect that a new version has been released under <https://www.minecraft.net/en-us/download/server/bedrock> it will download it, backs up your whitelist - -properties – permissions - and world files, replace the Minecraft installation, and then restore your configurations. So, the idea here is to simply re-execute the script and restarting the container one way.

A cronjob is a ‘job’ that gets executed and performs a desired task at specific times or intervals. You really could go a dozen different ways with this, but I’ll just go a simple route which is to just delete the container that is running our Minecraft Server which will automatically re-create it and re-run the script, thus updating it if an update is present or just starting the Minecraft process back up if no updates are found. You can specify the exact time and day of when to execute the job. Thinking of a time that doesn’t disrupt gameplay, maybe 5am every Thursday sounds about right for this guide.

Cronjobs are very useful and a great addition to an AKS cluster, you can use it for so many different tasks, if you run a web server for example you can set it up to rotate your ssl certificates without going a longer route; Your imagination is the limit.

Create a new file and let’s create our new yaml file.

### **7.1 Service Account**

RBAC is enabled by default on the AKS cluster, so the first part of the Yaml file will be for us to create a Service Account. This will be the account that is given permissions to execute the container deletion. Without this the cronjob will simply not have access to delete the container. You can change the service account name below and the namespace if you did not use the default for this guide.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | kind: ServiceAccount  apiVersion: v1  metadata:  name: myserviceaccount  namespace: default  --- |

### **7.2 Roles**

The next section is to create the roles that we will need to delete the pod. The access given below is limited to this action. You can look into the Kubernetes.io’s RBAC Authorization for more examples and features here <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | apiVersion: rbac.authorization.k8s.io/v1  kind: Role  metadata:  name: myroles  namespace: default  rules:  - apiGroups: [""]  resources:  - namespaces  - pods  - cronjobs  verbs: ["get", “list”, "delete"]  --- |

### **7.3 Role Bindings**

Okay, so we have created a service account and a list of roles and to link them we create a RoleBinding, which I’ve named myrolebinding. This will assign the roles to the service account. Here we will need to add the names of the role binding and the roles that you created above, so just add them in as follows.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | apiVersion: rbac.authorization.k8s.io/v1  kind: RoleBinding  metadata:  name: myrolebinding  namespace: default  roleRef:  apiGroup: rbac.authorization.k8s.io  kind: Role  name: myroles  subjects:  - kind: ServiceAccount  name: myserviceaccount  --- |

### **7.4 Cron Job**

What we have created above is just the access for your cronjob, below is the cronjob itself. Here are the details for each section:

“name” specifies the name of your cronjob. “schedule” is when and how often your cronjob will execute. Azure uses UTC so make sure to take this into account. Head over to <https://crontab.guru/> for more cronjob examples. “serviceAccountName” would be the name you used for the Service Account. “args” here is where we add the commands we want to execute. Since a pod’s complete name differs, we need to first get the current container name and we just need the deployment then to get this done, after this we can delete the pod. If you specified another name for your Minecraft container, replace “minecraft-server” with your deployment name.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | apiVersion: batch/v1  kind: CronJob  metadata:  name: myrestartcronjob  namespace: default  spec:  concurrencyPolicy: Forbid  schedule: "0 11 \* \* 4" #Every Thursday at 5AM UTC-6.  successfulJobsHistoryLimit: 1  failedJobsHistoryLimit: 1  jobTemplate:  spec:  backoffLimit: 1  activeDeadlineSeconds: 600  template:  spec:  serviceAccountName: myserviceaccount  restartPolicy: Never  containers:  - name: kubectl  image: bitnami/kubectl  command: [ "/bin/sh", "-c" ]  args:  - 'kubectl delete pod $(kubectl get pod -l app=minecraft-server -o jsonpath="{.items[0].metadata.name}")' |

### **7.5 Deploying Cronjob**

Now save your changes and deploy the cronjob, the four resources configured will be created.

|  |  |
| --- | --- |
| 1 | kubectl apply -f cronjob.yaml |

It should look as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | damian@Azure:~$ kubectl apply -f cronjob.yaml  serviceaccount/myserviceaccount created  role.rbac.authorization.k8s.io/myroles created  rolebinding.rbac.authorization.k8s.io/myrolebinding created  cronjob.batch/myrestartcronjob created  damian@Azure:~$ |

You can check the cronjob with the command below.

|  |  |
| --- | --- |
| 1 | kubectl get cronjobs |

Example:

|  |  |
| --- | --- |
| 1  2  3  4 | damian@Azure:~$ kubectl get cronjobs  NAME SCHEDULE SUSPEND ACTIVE LAST SCHEDULE AGE  myrestartcronjob 0 11 \* \* 4 False 0 <none> 30s  damian@Azure:~$ |

When the cronjob runs, you will see the cronjob in a running state, the Minecraft server terminating while the new one gets executed a second later. When it completes it will remain in the pod lists as completed and you can check the logs at any time. This is due to the configuration for the Cronjob under the ‘HistoryLimits’ values.

|  |  |
| --- | --- |
| 1  2  3  4  5 | damian@Azure:~$ kubectl get pods  NAME READY STATUS RESTARTS AGE  minecraft-server-777f9c8b85-kslr2 1/1 Running 0 11s  minecraft-server-777f9c8b85-sfhd2 1/1 Terminating 0 10h  myrestartcronjob-27383275-f8c9f 1/1 Running 0 12s |

If you check the logs for the cronjob you will see the confirmation that the Minecraft-server pod was deleted. If the job failed you will find the error logs here.

Example

|  |  |
| --- | --- |
| 1  2  3 | damian@Azure:~$ kubectl logs myrestartcronjob-27383275-f8c9f  pod "minecraft-server-777f9c8b85-sfhd2" deleted  damian@Azure:~$ |