Running Your Microservices Securely on AKS

Agraj Mangal
Mar 31 · 9 min read ★

Azure offers different ways to build & run your microservices ranging from Service Fabric to Azure Kubernetes Service to App Services to Azure Spring Cloud to even running Docker Enterprise and Apache Mesos in IAAS mode on Azure — the choice of compute depends on a lot of factors not excluding the affinity and knowledge of these services, the market hype (especially in case of Kubernetes), compatibility of the existing technology stack and the effort to migrate to the new service. Without trying to compare these compute options in Azure, in this article we focus on running your microservices "securely" in Azure Kubernetes Service. To make it more readable I plan to break this into a series of steps and articles.

1. Creating a Spring Boot App locally and working with Azure SQL Database
2. Deploying the Initial Infrastructure on Azure using Terraform and Azure DevOps
3. Using AKS Managed Azure AD Integration instead of Service Principals for Cluster Identity and using Azure AD Pod Identity to associate Managed Identities with Pods.
4. Dockerizing the Spring Boot App and deploying on AKS using Azure Devops pipelines.
5. Using KeyVault for managing secrets for application instead of using Base 64 encoded kubernetes secrets.
6. Securing the communication b/w PAAS services and AKS clusters using Private Link for Key Vault, Azure SQL and Container Registry.
7. Use Azure Policy Add-on for AKS to enforce basic governance (no privileged pods, only allow images from specific ACR etc.)

8. Use Kubernetes Network Policies to restrict ingress & egress traffic b/w namespaces or b/w pods & services.
9. Use a Web Application Firewall (WAF) with Application Gateway or Front Door or API Management to protect against common exploits like XSS, SQL injection, CSRF attacks etc.

There will be many things that you can further do to secure your deployment that are NOT covered in this series, at least initially (but you must consider for your use case)

- Using a Private AKS Cluster and deploy Bastion and Jumpbox to access. You would then have to use self-hosted agents instead of using Microsoft Hosted Azure DevOps agents to deploy to the Private AKS cluster.
- Use a Firewall (like Azure Firewall or any other NVA) to control inbound & output filtering, detect & block malicious traffic — Typically we recommend to deploy the firewall in the Hub VNet in a typical Hub-Spoke topology that can centrally exert control on multiple Spoke VNets
- Azure Confidential Computing: AKS now supports running confidential computing nodes (using Intel SGX processors) to protect your data while in use. Details here
- Node security using Kured — managing and maintaining the data plane nodes (VMs) is kind of a joint responsibility. While Azure platform automatically applies OS security patches it cannot and should not restart your VM (which may be required in some cases) — so its the customer responsibility to monitor and restart these nodes as & when required. Using an open source solution like Kured really helps here.
- Scanning container images for vulnerabilities before using them in your DevOps pipeline. Or enabling Azure Defender for container registries that can notify you if it finds some issues with your images

Let’s get Started

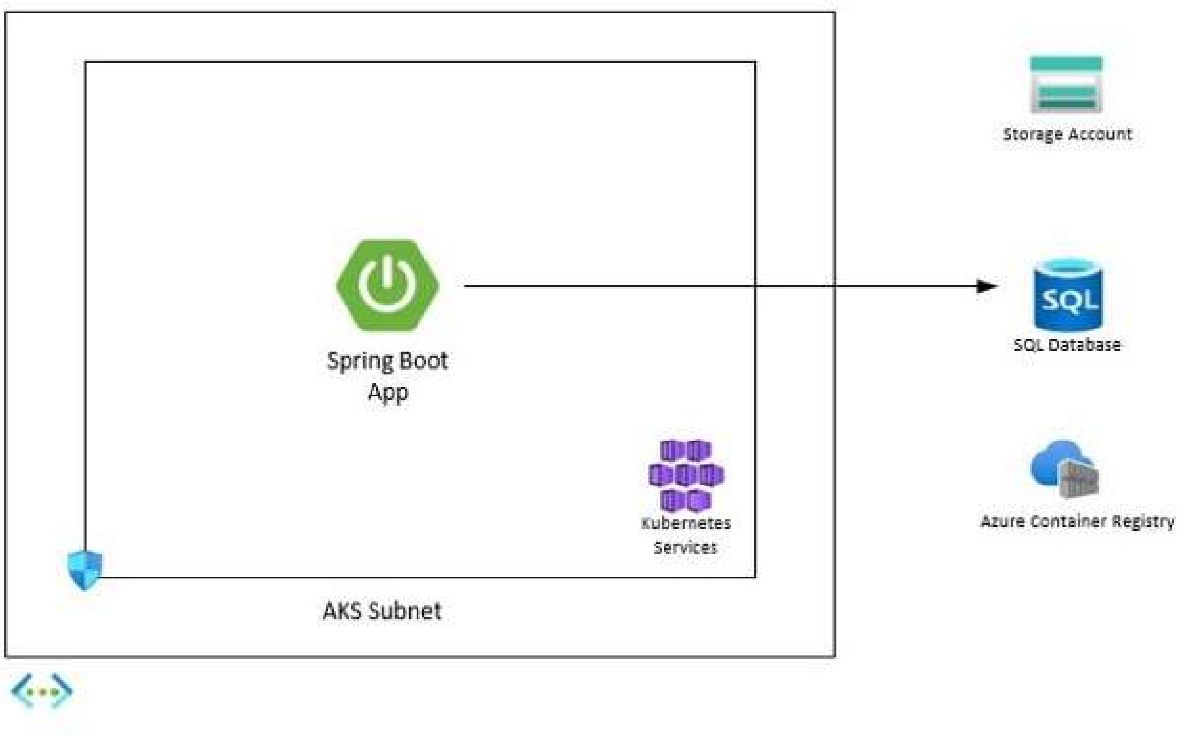
> *Step 1: Local Spring Boot App + Azure SQL*

To begin quickly, we will use the Spring Boot SQL Quickstart as a starting point which gives you a Spring Boot App running locally and accessing SQL Server in Azure. Once you have that running, lets go to the next step

> *Step 2: Deploy Initial Infrastructure in Azure*

Our initial infrastructure in Azure consists of

1. A Virtual Network with just one subnet to host our AKS cluster (yes, we will add more subnets subsequently in this VNet)
2. A very basic AKS Cluster with Azure CNI Network Plugin enabled instead of the default Kubenet plugin and using AKS-managed Azure AD instead of legacy Azure AD integration and an instance of Azure Container registry to hold your docker images for the microservices.
3. SQL server, a server-less SQL database and a storage account to hold the audit logs



Initial Deployment Architecture

All the terraform code for creating the above is located here. As a best practice we will be using Azure Storage as the remote backend for terraform for storing the Terraform state. I have a build pipeline that creates a resource group, a storage account and a key vault to hold the access keys of the storage account. This storage account will hold our terraform state files.

```
export COMMON_RESOURCE_GROUP_NAME=<your_resource_group>
export TF_STATE_STORAGE_ACCOUNT_NAME=<storage_account_name>
export TF_STATE_CONTAINER_NAME=<container_name>
export KEYVAULT_NAME=<key_vault_to_hold_access_key>
export LOCATION=<location_azure_resources>
# Creating resources
az group create -n $COMMON_RESOURCE_GROUP_NAME -l $LOCATION
az storage account create -g $COMMON_RESOURCE_GROUP_NAME -l $LOCATION \
    --name $TF_STATE_STORAGE_ACCOUNT_NAME \
    --sku Standard_LRS \
    --encryption-services blob
ACCOUNT_KEY=$(az storage account keys list --resource-group $COMMON_RESOURCE_GROUP_NAME --accou
az storage container create --name $TF_STATE_CONTAINER_NAME \
      --account-name $TF_STATE_STORAGE_ACCOUNT_NAME \
      --account-key $ACCOUNT_KEY
az keyvault create -g $COMMON_RESOURCE_GROUP_NAME -l $LOCATION --name $KEYVAULT_NAME
echo "Store storage access key into key vault secret..."
az keyvault secret set --name tfstate-storage-key --value $ACCOUNT_KEY --vault-name $KEYVAULT_N
```

build-infrastructure.sh hosted with ♡ by GitHub view raw

Setup Azure Resources for Terraform Remote Backend

Note that this storage account `TF_STATE_STORAGE_ACCOUNT_NAME` and key vault `KEYVAULT_NAME` are different from the ones that will be used by the application. If you need more details, refer to this excellent blog by Julien for details.

I will be using Azure DevOps pipeline for build & release pipelines for this project and everything (including the initial infrastructure) will be deployed via that pipeline. There were several learning for me while using Azure Devops for running this simple Terraform template that I want to highlight:

1. **Using AZ CLI Task to run Terraform**
   I used the following code snippet to deploy the Terraform code via Azure CLI task

```

- task: AzureCLI@2
  displayName: "Using Terraform to build Infrastructure"
  inputs:
    azureSubscription: azureServiceConnection
    scriptType: bash
    scriptLocation: inlineScript
    failOnStandardError: true
    workingDirectory: $(System.DefaultWorkingDirectory)/src/terraform
    addSpnToEnvironment: true
    inlineScript: |
      export ARM_CLIENT_ID=$servicePrincipalId
      export ARM_CLIENT_SECRET=$servicePrincipalKey
      export ARM_TENANT_ID=$tenantId
      export ARM_SUBSCRIPTION_ID=$(az account show --query id | xargs)
      export ACCESS_KEY=$(az keyvault secret show --name tfstate-storage-key --vault-name $KEYV
      terraform init \
        -backend-config="storage_account_name=$TF_STATE_STORAGE_ACCOUNT_NAME" \
        -backend-config="container_name=$TF_STATE_CONTAINER_NAME" \
        -backend-config="key=terraform-ref-architecture-tfstate" \
        -backend-config="access_key=$ACCESS_KEY
```

deploy-terraform.yaml hosted with ♡ by GitHub view raw

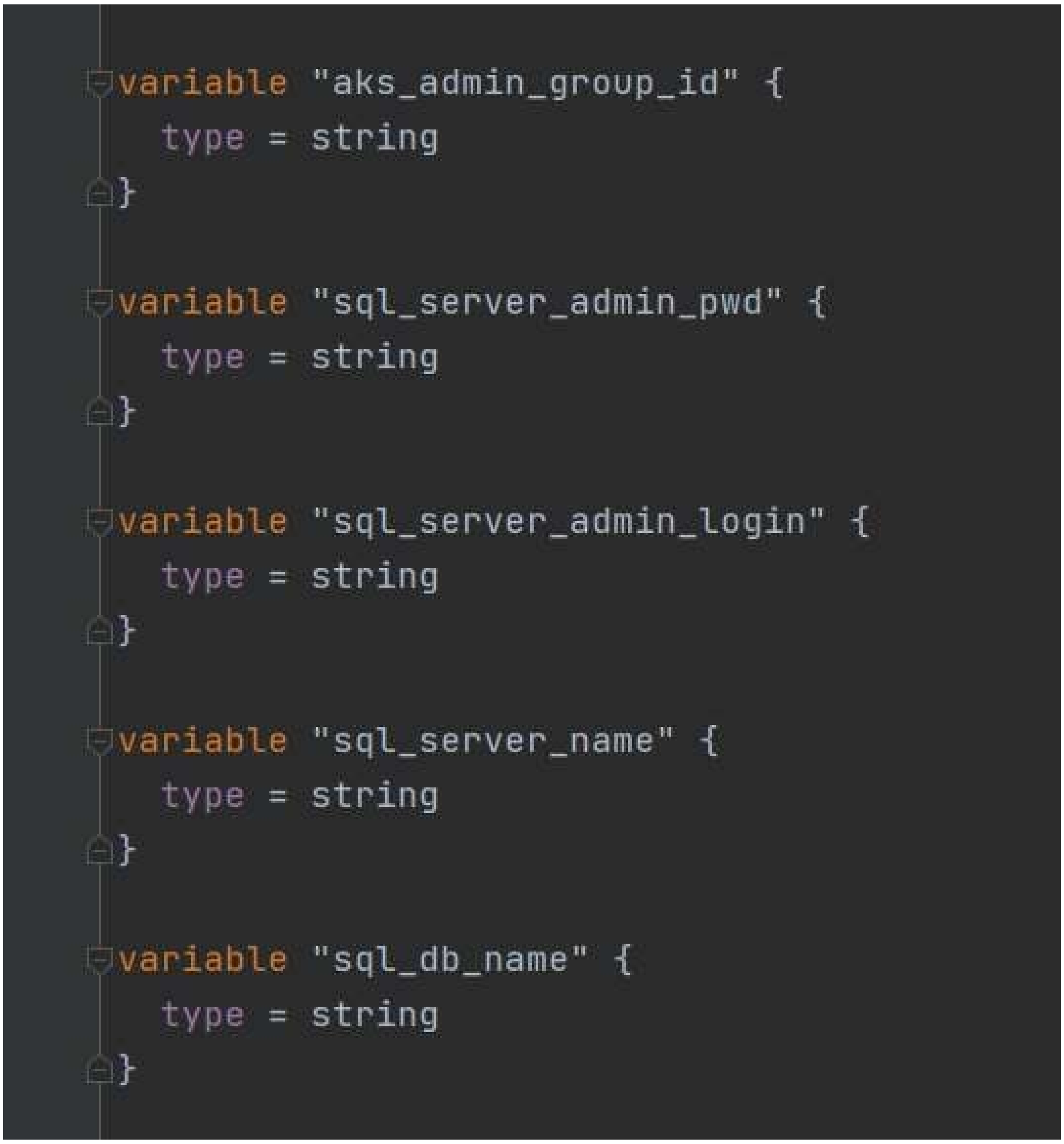
Sample AZ CLI Task to Deploy Terraform Code

Apart from the regular stuff with attaching the service connection via `azureSubscription` and changing the `workingDirectory` to point to the directory containing the terraform code, pay special attention to

- `addSpnToEnvironment: true` — This adds the service principal and key of the service connection to the script execution environment which you can access via `$env:servicePrincipalID` , `$env:servicePrincipalKey` and `$env:tenantId` which we have used in the above snippet to configure the environment variables required to execute Terraform `init/plan/apply/destroy` .
- The terraform `init` command uses the Remote storage and specifies the `storage account` name, the `container` and the `key` to store the state. For terraform to be able to access the storage account, either use the access key associated with the account or a SAS token. I have stored the access key of the storage account in a keyvault which I'm using above but you can also use a variable group in Azure DevOps and associate that with the keyvault to fetch secrets and make them available as environment variables.

2. Overriding the terraform variables via Azure DevOps Variables

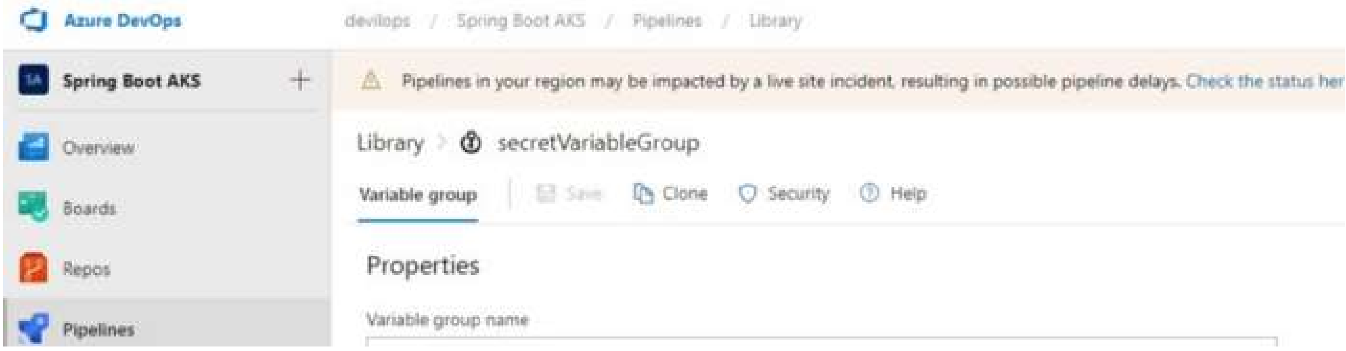
Now we have used some variables in our terraform code for holding SQL server and SQL Database names and login details. Also before we deploy this terraform code, we must create an Azure AD group for AKS cluster administrators and assign its objectId to the terraform variable — `aks_admin_group_id` — Members of this AAD group will have administrator access to our cluster.



```
variable "aks_admin_group_id" {
  type = string
}

variable "sql_server_admin_pwd" {
  type = string
}

variable "sql_server_admin_login" {
  type = string
}

variable "sql_server_name" {
  type = string
}

variable "sql_db_name" {
  type = string
}
```

Variables from Terraform Template

Some of these values are secrets and should reside in a key vault and not as a part of configuration so I have created a variable group in Azure DevOps linked it to Key Vault containing the secret values

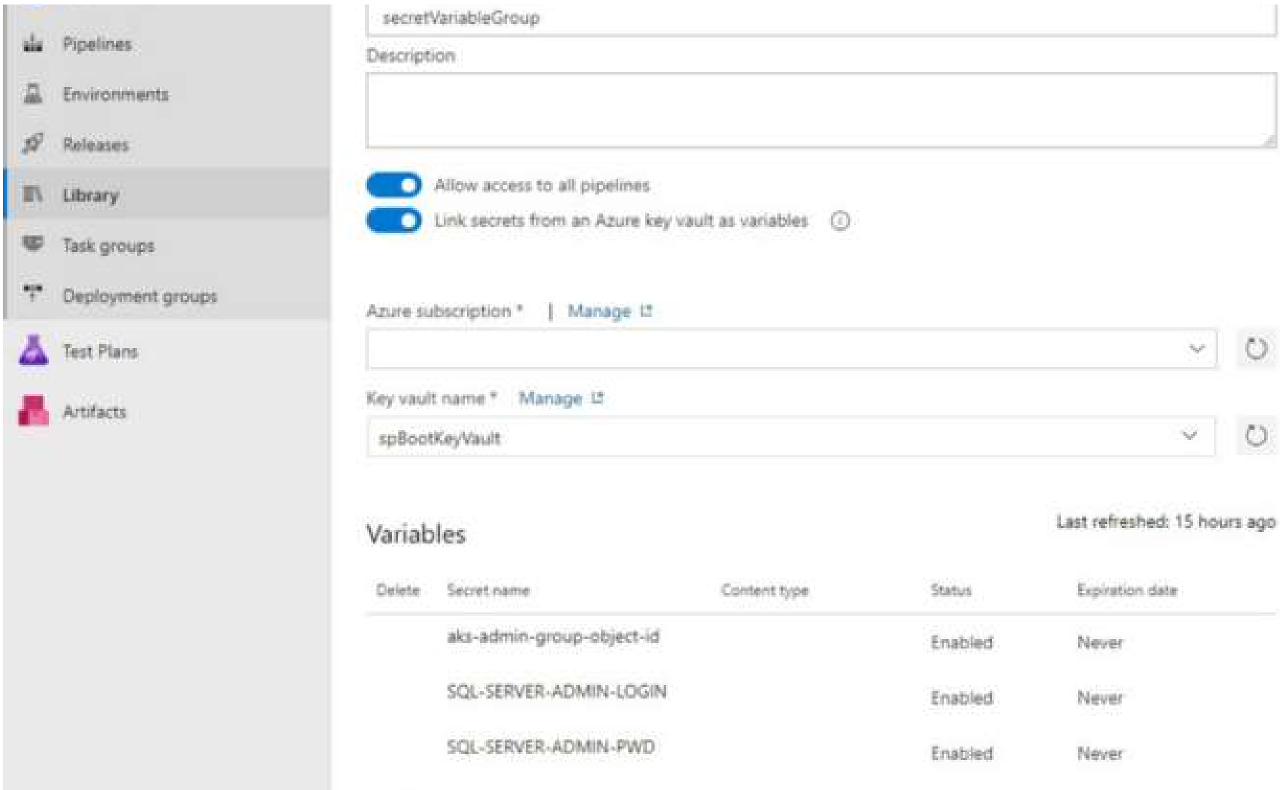


Pipelines in your region may be impacted by a live site incident, resulting in possible pipeline delays. Check the status her

Library > secretVariableGroup

Properties

Variable group name



secretVariableGroup

Description

Allow access to all pipelines

Link secrets from an Azure key vault as variables

Azure subscription *

Key vault name *

spBootKeyVault

Variables

Last refreshed: 15 hours ago

Delete	Secret name	Content type	Status	Expiration date
	aks-admin-group-object-id		Enabled	Never
	SQL-SERVER-ADMIN-LOGIN		Enabled	Never
	SQL-SERVER-ADMIN-PWD		Enabled	Never

secretVariableGroup holding secrets from Key Vault

By default all secrets are available as environment variables in your tasks, so just substitute the values of variables by appending `TF_VAR_` to the variable name, so if your variable in terraform was named as `sql_server_admin_login` you can override that by specifying `TF_VAR_sql_server_admin_login` as we have done before doing `terraform plan`

```yaml
variables:
  - group: buildVariableGroup      # variables for build pipeline (terraform state)
  - group: terraformVariableGroup  # non-secret values
  - group: secretVariables         # secret values from key vault

pool:
  vmImage: ubuntu-latest

steps:

- task: AzureCLI@2
  displayName: "Using Terraform to build Infrastructure"
  inputs:
    azureSubscription: azureServiceConnection
    scriptType: bash
    scriptLocation: inlineScript
    failOnStandardError: true
    workingDirectory: $(System.DefaultWorkingDirectory)/src/terraform
    addSpnToEnvironment: true
```

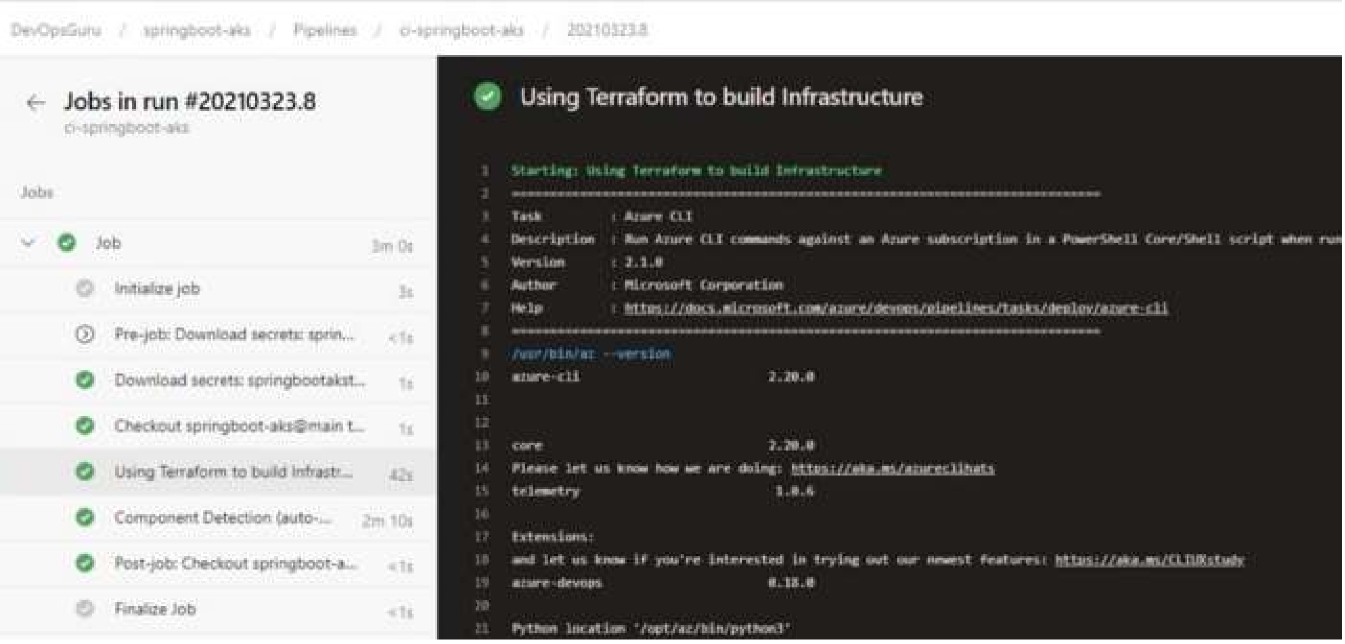
```
20      inlineScript: |
21        export ARM_CLIENT_ID=$servicePrincipalId
22        export ARM_CLIENT_SECRET=$servicePrincipalKey
23        export ARM_TENANT_ID=$tenantId
24        export ARM_SUBSCRIPTION_ID=$(az account show --query id | xargs)
25        export ACCESS_KEY=$(az keyvault secret show --name tfstate-storage-key --vault-name $KEYV
26        terraform init \
27          -backend-config="storage_account_name=$TF_STATE_STORAGE_ACCOUNT_NAME" \
28          -backend-config="container_name=$TF_STATE_CONTAINER_NAME" \
29          -backend-config="key=terraform-ref-architecture-tfstate" \
30          -backend-config="access_key=$ACCESS_KEY"
31        export TF_VAR_aks_admin_group_id=$(aks-admin-group-object-id)
32        export TF_VAR_sql_db_name=$(SQL-DB-NAME)
33        export TF_VAR_sql_server_name=$(SQL-SERVER-NAME)
34        export TF_VAR_sql_server_admin_login=$(SQL-SERVER-ADMIN-LOGIN)
35        export TF_VAR_sql_server_admin_pwd=$(SQL-SERVER-ADMIN-PWD)
36        terraform plan -out=infra.out
37        terraform apply -auto-approve infra.out
```

override-tf-vars.yaml hosted with ♡ by GitHub view raw

Using Azure DevOps Variables for Terraform Vars

Lines 2–5 are used to specify multiple variable groups (secrets and non-secrets) Always good to split your variables over multiple reusable groups which can be shared in related pipelines.

Lines 32–37 exposes the secrets and variables in those groups as Terraform variables following our convention `TF_VAR_<variable_name>` — now we can execute the `plan` and `apply` phase of our pipeline and deploy the infrastructure in Azure.

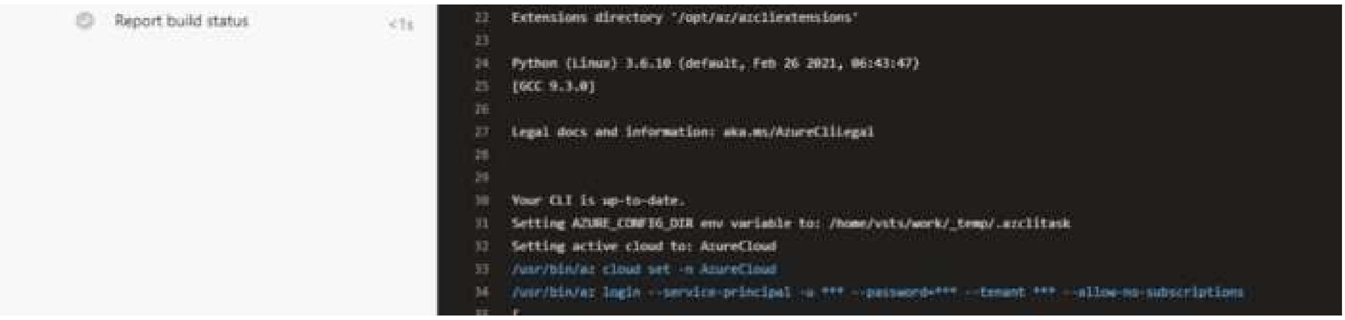

DevOpsGuru / springboot-aks / Pipelines / ci-springboot-aks / 20210323.8
←
Jobs

Jobs in run #20210323.8
ci-springboot-aks

Job 3m 0s
Initialize job 3s
Pre-job: Download secrets: sprin... <1s
Download secrets: springbootakst... 1s
Checkout springboot-aks@main t... 1s
Using Terraform to build Infrastr... 42s
Component Detection (auto-... 2m 10s
Post-job: Checkout springboot-a... <1s
Finalize Job <1s

Using Terraform to build Infrastructure

```
Starting: Using Terraform to build Infrastructure
==============================================================================
Task         : Azure CLI
Description  : Run Azure CLI commands against an Azure subscription in a PowerShell Core/Shell script when run
Version      : 2.1.0
Author       : Microsoft Corporation
Help         : https://docs.microsoft.com/azure/devops/pipelines/tasks/deploy/azure-cli
==============================================================================
/usr/bin/az --version
azure-cli                         2.20.0


core                              2.20.0
Please let us know how we are doing: https://aka.ms/azureclihats
telemetry                          1.0.6

Extensions:
and let us know if you're interested in trying out our newest features: https://aka.ms/CLIUXstudy
azure-devops                      0.18.0

Python location '/opt/az/bin/python3'
```



```
Extensions directory '/opt/az/azcliextensions'

Python (Linux) 3.6.10 (default, Feb 26 2021, 06:43:47)
[GCC 9.3.0]

Legal docs and information: aka.ms/AzureCliLegal


Your CLI is up-to-date.
Setting AZURE_CONFIG_DIR env variable to: /home/vsts/work/_temp/.azclitask
Setting active cloud to: AzureCloud
/usr/bin/az cloud set -n AzureCloud
/usr/bin/az login --service-principal -u *** --password=*** --tenant *** --allow-no-subscriptions
{
```

Pipeline Sucessfully Deploys to Azure

Go to the portal and confirm that your infrastructure has been deployed correctly.

Step 3: Dockerize the Spring Boot App

Now that we have the required infra in Azure, let's start by adding a Dockerfile to our project — we will be using the Multi-stage build approach and Layered Jar mode for Spring Boot and extracting different layers in our image instead of using a fat jar

```
FROM adoptopenjdk:11-jre-hotspot as builder
WORKDIR application
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM adoptopenjdk:11-jre-hotspot
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
COPY --from=builder application/application/ ./
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
```

Dockerfile hosted with ♡ by GitHub view raw

Dockerfile for our Spring

You can test it locally using

```
docker build . -t agrajm/spring-boot-aks:v1.0
```

This will build it locally and create a local image for you that you can run as follows:

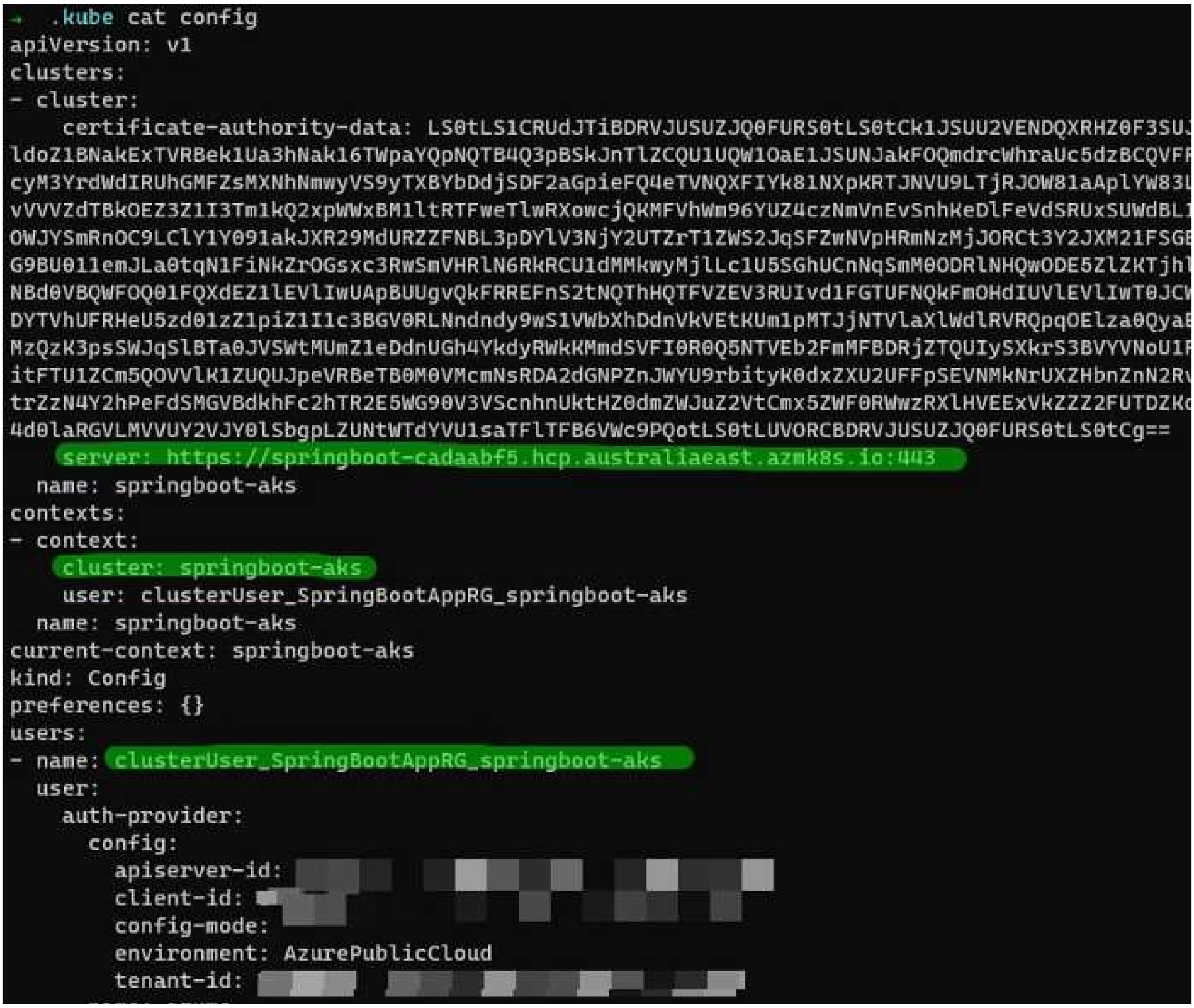
```
docker run -it -p8080:8080 \
-e SPRING_DATASOURCE_URL="<Your_DB_Conn_String>" \
-e SPRING_DATASOURCE_USERNAME="<Your_DB_UserName>" \
-e SPRING_DATASOURCE_PASSWORD="<Your_DB_Password>" \
agrajm/spring-boot-aks:v1.0
```

> *Step 4: Deploy to AKS — Enters kubelogin for DevOps !*

Let's first try & deploy locally to our AKS cluster — we need to authenticate against it first. To do so type

```
az aks get-credentials --resource-group <RG_NAME> --name
<AKS_CLUSTER_NAME>
```

This will create or update your `KUBECONFIG` with details on how to authenticate against the AKS cluster. My config currently looks like

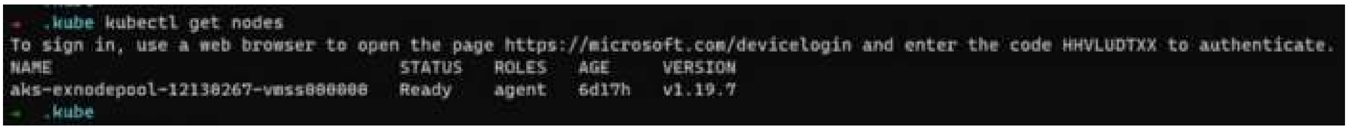


```
→  .kube cat config
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUU2VENDQXRHZ0F3SUJ
ldoZ1BNakExTVRBek1Ua3hNak16TWpaYQpNQTB4Q3pBSkJnTlZCQU1UQW1OaE1JSUNJakFOQmdrcWhraUc5dzBCQVFF
cyM3YrdWdIRUhGMFZsMXNhNmwyVS9yTXBYbDdjSDF2aGpieFQ4eTVNQXFIYk81NXpKRTJNVU9LTjRJOW81aAplYW83L
vVVVZdTBkOEZ3Z1I3Tm1kQ2xpWWxBM1ltRTFweTlwRXowcjQKMFVhWm96YUZ4czNmVnEvSnhKeDlFeVdSRUxSUWdBLI
OWJYSmRnOC9LClY1Y091akJXR29MdURZZFNBL3pDYlV3NjY2UTZrT1ZWS2JqSFZwNVpHRmNzMjJORCt3Y2JXM21FSGE
G9BU01lemJLa0tqN1FiNkZrOGsxc3RwSmVHRlN6RkRCU1dMMkwyMjlLc1U5SGhUCnNqSmM0ODRlNHQwODE5ZlZKTjhl
NBd0VBQWFOQ01FQXdEZ1lEVlIwUApBUUgvQkFRREFnS2tNQThHQTFVZEV3RUIvd1FGTUFNQkFmOHdIUVlEVlIwT0JCW
DYTVhUFRHeU5zd01zZ1piZ1I1c3BGV0RLNndndy9wS1VWbXhDdnVkVEtKUm1pMTJjNTVlaXlWdlRVRQpqOElza0QyaE
MzQzK3psSWJqSlBTa0JVSWtMUmZ1eDdnUGh4YkdyRWkKMmdSVFI0R0Q5NTVEb2FmMFBDRjZTQUIySXkrS3BVYVNoU1F
itFTU1ZCm5QOVVlK1ZUQUJpeVRBeTB0M0VMcmNsRDA2dGNPZnJWYU9rbityK0dxZXU2UFFpSEVNMkNrUXZHbnZnN2Rv
trZzN4Y2hPeFdSMGVBdkhFc2hTR2E5WG90V3VScnhnUktHZ0dmZWJuZ2VtCmx5ZWF0RWwzRXlHVEExVkZZZ2FUTDZKc
4d0laRGVLMVVUY2VJY0lSbgpLZUNtWTdYVU1saTFlTFB6VWc9PQotLS0tLUVORCBDRVJUSUZJQ0FURS0tLS0tCg==
    server: https://springboot-cadaabf5.hcp.australiaeast.azmk8s.io:443
  name: springboot-aks
contexts:
- context:
    cluster: springboot-aks
    user: clusterUser_SpringBootAppRG_springboot-aks
  name: springboot-aks
current-context: springboot-aks
kind: Config
preferences: {}
users:
- name: clusterUser_SpringBootAppRG_springboot-aks
  user:
    auth-provider:
      config:
        apiserver-id: [redacted]
        client-id: [redacted]
        config-mode: [redacted]
        environment: AzurePublicCloud
        tenant-id: [redacted]
```

```
    name: azure
→  .kube |
```

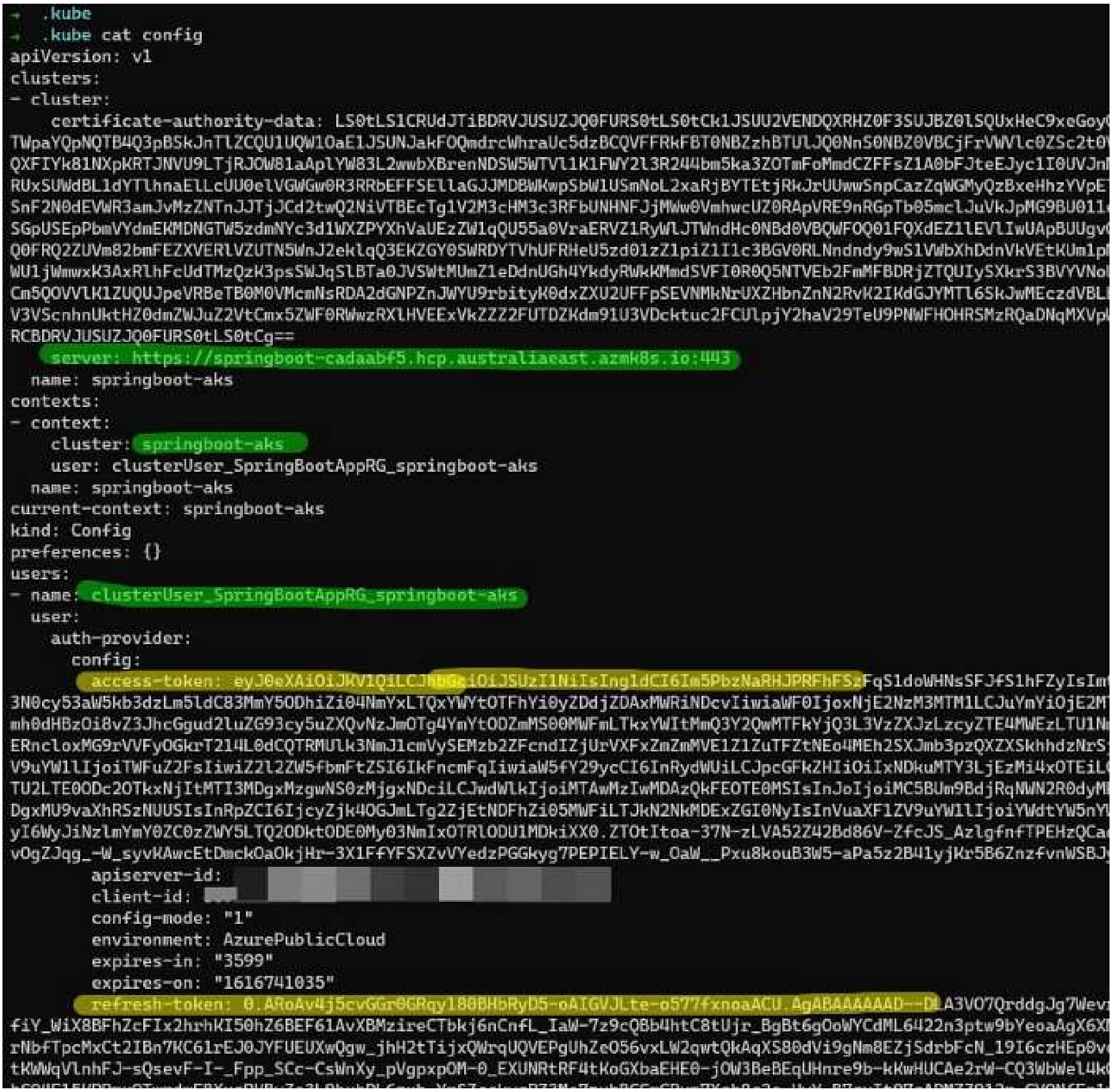
Redacted ~/.kube/config

When I try to get the nodes of this cluster using kubectl



```
→  .kube kubectl get nodes
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code HHVLUDTXX to authenticate.
NAME                                STATUS   ROLES   AGE    VERSION
aks-exnodepool-12138267-vmss000000  Ready    agent   6d17h  v1.19.7
→  .kube
```

It asks me to authenticate with my Azure AD instance as our cluster is using AKS managed Azure AD integration — once I open a new tab and do `devicelogin` and put that code & my credentials in, it issues me a temporary access & refresh token which is updated in the same `./kube/config` as can be seen below:



```
→  .kube
→  .kube cat config
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUU2VENDQXRHZ0F3SUJBZ0lSQUxHeC9xeGoy
TWpaYQpNQTB4Q3pBSkJnTlZCQU1UQW1OaE1JSUNJakFOQmdrcWhraUc5dzBCQVFFRkFBT0NBZzhBTUlJQ0NnS0NBZ0VBCjFrVWVlc0ZSc2t0
QXFIYk81NXpKRTJNVU9LTjRJOW81aAplYW83L2wwbXBrenNDSW5WTVl1K1FWY2l3R244bm5ka3ZOTmFoMmdCZFFsZ1A0bFJteEJyc1I0UVJn
RUxSUWdBL1dYTlhnaElLcUU0elVGWGw0R3RRbEFFSElaGJJMDBWKwpSbW1USmNoL2xaRjBYTEtjRkJrUUwwSnpCazZqWGMyQzBxeHhzYVpe
SnF2N0dEVWR3amJvMzZNTnJJTjJCd2twQ2NiVTBEcTg1V2M3cHM3c3RFbUNHNFJjMWw0VmhwcUZ0RApVRE9nRGpTb05mclJuVkJpMG9BU011
SGpUSEpPbmVYdmEKMDNGTW5zdmNYc3d1WXZPYXhVaUEzZW1qQU55a0VraERVZ1RyWlJTWndHc0NBd0VBQWFOQ01FQXdEZ1lEVlIwUApBUUgv
Q0FRQ2ZUVm82bmFEZXVERlVZUTN5WnJ2eklqQ3EKZGY0SWRDYTVhUFRHeU5zd01zZ1piZ1I1c3BGV0RLNndndy9wS1VWbXhDdnVkVEtKUm1p
WU1jWmwxK3AxRlhFcUdTMzQzK3psSWJqSlBTa0JVSWtMUmZ1eDdnUGh4YkdyRWkKMmdSVFI0R0Q5NTVEb2FmMFBDRjZTQUIySXkrS3BVYVNo
Cm5QOVVLK1ZUQUJpeVRBeTB0M0VMcmNsRDA2dGNPZnJWYU9rbityK0dxZXU2UFFpSEVNMkNrUXZHbnZnN2RvK2IKdGJYMTl6SkJwMEczdVBL
V3VScnhnUktHZ0dmZWJuZ2VtCmx5ZWF0RWwzRXlHVEExVkZZZ2FUTDZKdm91U3VDcktuc2FCUlpjY2haV29TeU9PNWFHOHRSMzRQaDNqMXVp
RCBDRVJUSUZJQ0FURS0tLS0tCg==
    server: https://springboot-cadaabf5.hcp.australiaeast.azmk8s.io:443
  name: springboot-aks
contexts:
- context:
    cluster: springboot-aks
    user: clusterUser_SpringBootAppRG_springboot-aks
  name: springboot-aks
current-context: springboot-aks
kind: Config
preferences: {}
users:
- name: clusterUser_SpringBootAppRG_springboot-aks
  user:
    auth-provider:
      config:
        access-token: eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Im5PbzNaRHJPRFhFSzFqS1doWHNsSFJfS1hFZyIsIm
3N0cy53aW5kb3dzLm5ldC83MmY5ODhiZi04NmYxLTQxYWYtOTFhYi0yZDdjZDAxMWRiNDcvIiwiaWF0IjoxNjE2NzM3MTM1LCJuYmYiOjE2M
mh0dHBzOi8vZ3JhcGgud2luZG93cy5uZXQvNzJmOTg4YmYtODZmMS00MWFmLTkxYWItMmQ3Y2QwMTFkYjQ3L3VzZXJzLzcyZTE4MWEzLTU1N
ERncloxMG9rVVFyOGkrT214L0dCQTRMUlk3NmJ1cmVySEMzb2ZFcndIZjUrVXFxZmZmMVE1Z1ZuTFZtNEo4MEh2SXJmb3pzQXZXSkhhdzNrS
V9uYW1lIjoiTWFuZ2FsIiwiZ2l2ZW5fbmFtZSI6IkFncmFqIiwiaW5fY29ycCI6InRydWUiLCJpcGFkZHIiOiIxNDkuMTY3LjEzMi4xOTEiL
TU2LTE0ODc2OTkxNjItMTI3MDgxMzgwNS0zMjgxNDciLCJwdWlkIjoiMTAwMzIwMDAzQkFEOTE0MSIsInJoIjoiMC5BUm9BdjRqNWN2R0dyM
DgxMU9vaXhRSzNUUSIsInRpZCI6IjcyZjk4OGJmLTg2ZjEtNDFhZi05MWFiLTJkN2NkMDExZGI0NyIsInVuaXF1ZV9uYW1lIjoiYWdtYW5nY
yI6WyJiNzlmYmY0ZC0zZWY5LTQ2ODktODE0My03NmIxOTRlODU1MDkiXX0.ZTOtItoa-37N-zLVA52Z42Bd86V-ZfcJS_AzlgfnfTPEHzQCa
vOgZJqg_-W_syvKAwcEtDmckOaOkjHr-3X1FfYFSXZvVYedzPGGkyg7PEPIELY-w_OaW__Pxu8kouB3W5-aPa5z2B41yjKr5B6ZnzfvnWSBJ
          apiserver-id: [redacted]
          client-id: [redacted]
          config-mode: "1"
          environment: AzurePublicCloud
          expires-in: "3599"
          expires-on: "1616741035"
          refresh-token: 0.ARoAv4j5cvGGr0GRqyl808HbRyD5-oAIGVJLte-o577fxnoaACU.AgABAAAAAAD--DLA3VO7QrddgJg7Wev
fiY_WiX8BFhZcFIx2hrhKI50hZ6BEF61AvXBMzireCTbkj6nCnfL_IaW-7z9cQBb4htC8tUjr_BgBt6gOoWYCdML6422n3ptw9bYeoaAgX6X
rNbfTpcMxCt2IBn7KC61rEJ0JYFUEUXwQgw_jhH2tTijxQWrqUQVEPgUhZeO56vxLW2qwtQkAqXS80dVi9gNm8EZjSdrbFcN_19I6czHEp0v
tKWWqVlnhFJ-sQsevF-I-_Fpp_SCc-CsWnXy_pVgpxpOM-0_EXUNRtRF4tKoGXbaEHE0-jOW3BeBEqUHnre9b-kKwHUCAe2rW-CQ3WbWel4k
```

```
DPIAyxkUCWNfJQo-g7djCQObClNnZzpQXact2TgLlh7mHyOgSAknBzQIi2HyuYBYJflwTl8rnfojVTYk6wtfkGZkevrmxLaH08dLuaJjVeu4
      tenant-id: '[redacted]
    name: azure
→  .kube |
```

Access & Refresh tokens after successful login

Now we can deploy our Application to AKS using `kubectl apply` but we must create the deployment YAML for our Spring Boot App. Let's use the below deployment config

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-aks
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-boot-aks
  template:
    metadata:
      labels:
        app: spring-boot-aks
    spec:
      containers:
        - name: app
          image: $(ACR_LOGIN_SERVER)/$(ACR_REPO_NAME):$(Build.BuildNumber)
          ports:
          - containerPort: 8080
          env:
          - name: SPRING_DATASOURCE_URL
            valueFrom:
              secretKeyRef:
                name: springsecret
                key: url
          - name: SPRING_DATASOURCE_USERNAME
            valueFrom:
              secretKeyRef:
                name: springsecret
                key: username
          - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: springsecret
                  key: password
          imagePullPolicy: Always
```

spring-boot-deployment.yaml hosted with ♡ by GitHub view raw

Deployment for Kubernetes

Now that you are authenticated to the cluster, you can simply apply the changes using `kubectl`

```
kubectl apply -f depoloy/spring-boot-deployment.yaml
```

This will work as my currently logged in user is already a part of the AAD group which is the admin group for AKS but how will this work with your pipelines — you cannot respond to the `devicelogin` challenge from your pipeline — enters Kubelogin !

Even with an AAD managed AKS cluster, kubelogin allows us to do non-interactive login using a Service Principal or in the latest release — even using the Azure CLI token making it really ideal to use in CI/CD scenarios. We will be using the latter option. So the service principal associated with our Azure CLI must also be added into the same AAD group to be able to authenticate with the cluster and deploy the app to it.

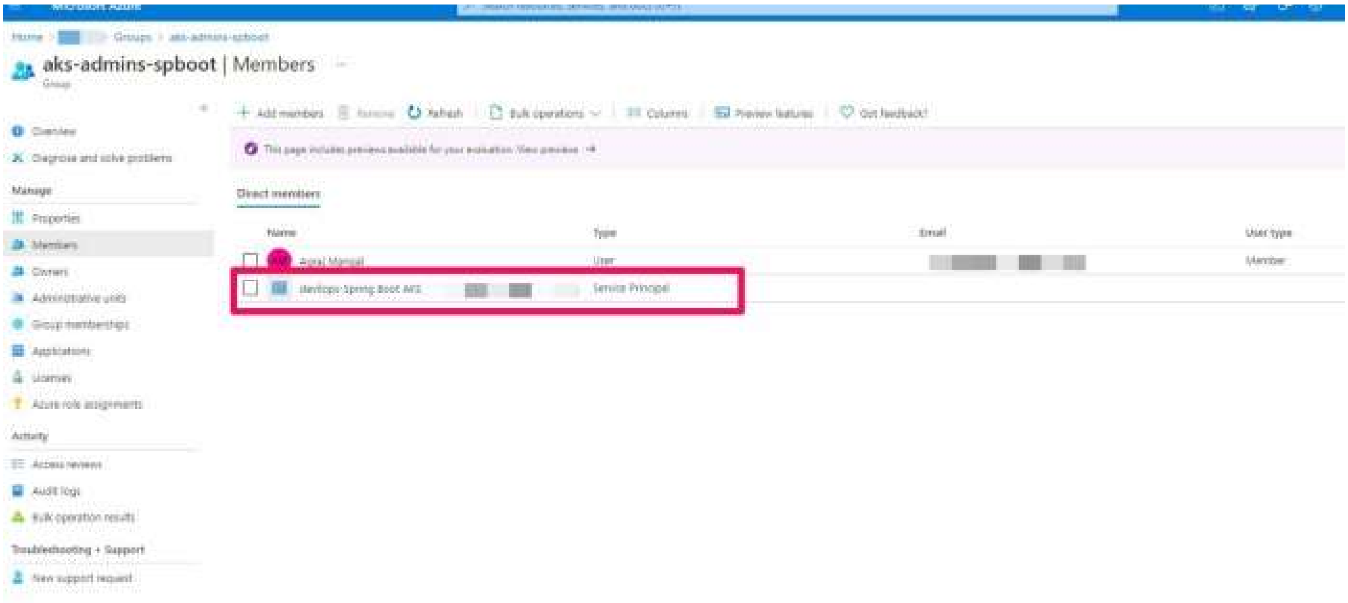
Also, since `kubelogin` is not installed by default on the latest ubuntu images my Microsoft hosted build agents are using, I‘ve to install it first before I can use it. Snippet from the pipeline code that makes use of it

```
export KUBECONFIG=$(pwd)/.kubeconfig-$(AKS_CLUSTER_NAME)
az aks get-credentials --resource-group $(RESOURCE_GROUP_NAME) --name $(AKS_CLUSTER_NAME) --over
wget https://github.com/Azure/kubelogin/releases/download/v0.0.9/kubelogin-linux-amd64.zip
unzip kubelogin-linux-amd64.zip
sudo mv bin/linux_amd64/kubelogin /usr/local/bin
echo "kubelogin installed"
kubelogin convert-kubeconfig -l azurecli
kubectl apply -f deploy/*
```

kubelogin-pipeline.yaml hosted with ♡ by GitHub view raw

kubelogin convert-kubeconfig -l azurecli

The main command is `kubelogin convert-kubeconfig -l azurecli` which lets you use the underlying token of the service principal being used by the Azure CLI task for your pipeline. You must add that SP to your AAD group as I’ve done for my AKS Admin AAD group



Home > [redacted] > Groups > aks-admins-spboot

aks-admins-spboot | Members

Group

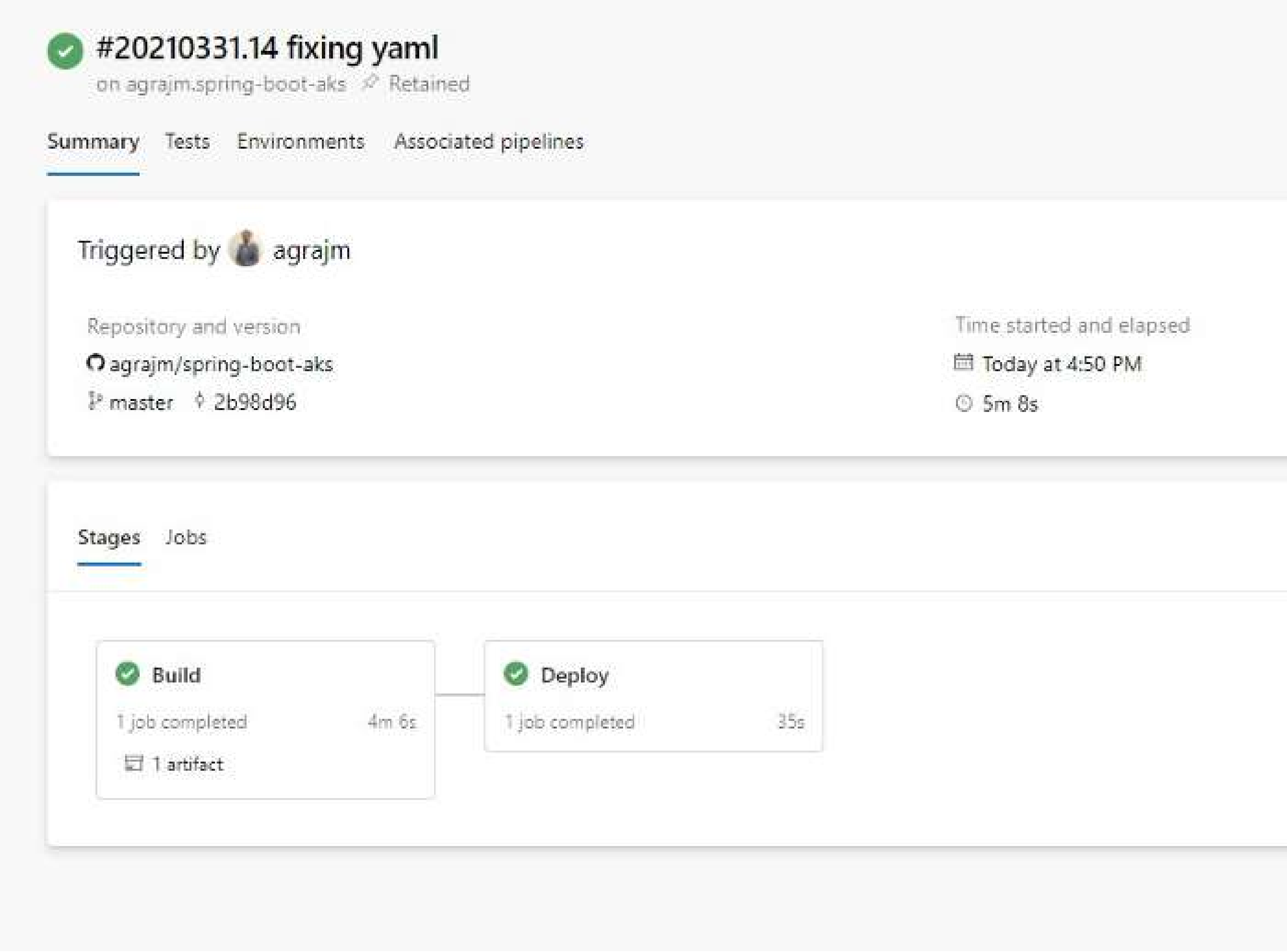
This page includes previews available for your evaluation. View previews →

Direct members

	Name	Type	Email	User type
□	Agraj Mangal	User	[redacted]	Member
□	devops-Spring Boot AKS [redacted]	Service Principal		

Adding your deployment SP to AKS Admin AAD group

Now the AZDO pipeline should be able to authenticate & deploy the Spring Boot Application to your AKS cluster.



#20210331.14 fixing yaml

on agrajm.spring-boot-aks Retained

Triggered by agrajm

Repository and version

agrajm/spring-boot-aks

master 2b98d96

Time started and elapsed

Today at 4:50 PM

5m 8s

Build

1 job completed 4m 6s

1 artifact

Deploy

1 job completed 35s

Next Steps

So far we have a working Spring boot app deployed in AKS talking to Azure SQL database. But this is just the tip of the iceberg and we need to put in all the security controls I promised at the start — we'll continue this journey to learning together how to secure your microservices (Spring Boot & otherwise) on AKS in the next set of articles. Till then, happy hacking !

Spring Kubernetes Azure DevOps Terraform