Full Stack Azure Development

Project #2

#### Preamble

As noted in project #1, this semester I decided to take the approach of not just doing singular project, but rather to use the 3 projects to create one large Epic. During project #2, the infrastucture is moved off of Terraform deployed VMs and onto Azure’s Azure Kubernetes Service (AKS). This offloads the responsibility of maintaining the under lying hardware to Azure and allow us to focus more on application development.

Taking the project further, the static, non-persistent Event Card from the Project #2 will be replaced with a persistent Kanban Board list. While not fully functional, this small POC will demonstrate several key concepts:

* Data persistence
* Multi-user access across the web
* Real-time updating to all connected clients
* Microservice layering – Public Web App, Private Backend DB

The current app features a listing of Kanban boards, a user can add a board and the data will flow to the Postgres DB, as well as updating any attached clients. Future enhancements could be extended from this framework to add fully functional Kanban actions – selecting a Board could display a Kanban Board in TODO, IN-PROGRESS & DONE columns; dragging a note from TODO to IN-PROGRESS would update the state in the DB, as well as attached clients.

#### Objectives

There is a lot of work to do in getting all this together, here is a look at the over-all goals of Project #2:

* Stand up a single node AKS area in the Cloud
* Create a Container Repository to hold our application images (ACR)
* Extend our application from a static, non-persistent entity to a fully persistent, multiple user system
* Use persistent storage volumes (PVC)
* Utilize a Postgres backend for data management
* Utilize a Node.js/Express/Socket.io web architecture to serve our application across the web
* Use a Kubernetes External Loadbalancer to expose our application over the public web, while keeping our Postgres DB out of the public eye

#### Infrastructure Layout

A screenshot of a cell phone

Description automatically generated

#### Create Cluster

First steps include createing the AKS cluster and the ACR repository. These are pretty straight forward tasks and are as follows:

**Creating the AKS Cluster**

In the Azure search bar, enter Kubernetes services

From the Kubernetes services dashboard, click ‘+ Add’

Select ‘+ Add Kubernetes cluster’

Use the following details to create a cluster

Subscription: Select your subscription

Resource group: click ‘Create new’

Enter: FSKubeRG

Kubernetes cluster name: FSKubeCluster

Region: (US) East US

Kubernetes version: Leave default (1.15.11)

VM size: B2s

Node count: 1

Click ‘Node pools’

Leave defaults, except for VM scale sets, set this to Disabled (can be changed later)

Click ‘Authentication’

Leave the defaults, this will also create a new Service Principal (SP)

Click ‘Networking’

Leave all defaults, except HTTP application routing, set this to Yes.

This will also setup the Monitoring service and initiate a Log Analytics workspace

Click ‘Integrations’

Leave all the defaults. We will add the container registry later in the DevOps setup

Click ‘Tags’

Create a new tag:

FSKube : K8s

Click ‘Review + create’

Click ‘Create’

You should see a “Your deployment is underway’ screen

**Setup Container Registry (ACR)**

The ACR is where the Docker images are stored for creating our StatefulSets and Deployments. The images are built locally and pushed to the ACR. This is an added benefit of not keeping the images in Github SVC.

Use Azure search for Container registries and click '+ Add'

Subscription: select proper subscription

Resource group: select FSKubeRG

Registry name: FSKubeCR

Location: East US

SKU: Basic

Click 'Tags' from the top tabs on the configuration screen

FSKube : FS Container Registry

Click 'Review + create'

Click 'Create'

You should see a "Your deployment is underway' screen

**Link the ACR registry to your GitHub**

**Create an SP for the ACR to use**

Azure will need a service principal (SP) in order to provide services to the ACR. Azure also provides a handy script to do this, which is in my GitHub repo.

I will update the create\_ACR\_sp.sh for the 2 required variables:

Set ACR\_NAME=FSKubeCR

Set SERVICE\_PRINCIPAL\_NAME=fskube-acr-sp

Make the file executable: chmod 755 create\_ACR\_sp.sh

Run the file to create the SP:

./create\_ACR\_sp.sh

Copy the output:

Service principal ID: 0e2fafd3-1e4a-4b6b-8f77-89225649f116

Service principal password: 1tsvvPV`S9TFXzf-%hx24G=3}aj2U'gi

**Create a Pull Secret using the SP**

Azure gives a really silly (and non-conformant) auto generated password. For most of the steps requiring one of these passwords, I set an environment variable for it. Using quotes (or double-quote if the password includes a quote!) seems to help this process – beware of anything that may need to be escaped (backtick, eg)

export MYPASS='<thepassword>' # Use single quotes

Then use the env var in the call:

$ kubectl create secret docker-registry fskubeacrsecret \

--docker-server=FSKubeCR.azurecr.io \

--docker-username=0e2fafd3-1e4a-4b6b-8f77-89225649f116 \

--docker-password=$MYPASS

Go to the resource "Container registry" and select FSKubeCR

Select Access Keys from the left pane

Enable the Admin to use & copy the auto-generated passwords:

password La1196i93T1FGHvmj+uz1IVXC9VAkkwq

password2 haZ1Mij4VwMBgCATf7B4=FSJdQ0IlTWT

**Link the ACR to AKS**

The final step in setting up this relationship is to use a little Azure CLI on our local box to simplify this task:

Login to azure on command line

$ az login # Do not use SP

$ az aks update -n FSKubeCluster -g FSKubeRG --attach-acr FSKubeCR

This process will take a minute and then output a JSON response.

Login to cluster:

$ az aks get-credentials -g FSKubeRG --name FSKubeCluster

Note: The first time you do this, Kubernetes will merge the .kube config with the information currently on your local machine. It should also properly set the context for working in the cloud.

$ kubectl get namespaces

NAME STATUS AGE

default Active 28h

kube-node-lease Active 28h

kube-public Active 28h

kube-system Active 28h

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

aks-agentpool-44135841-0 Ready agent 28h v1.15.11

**Push Docker builds to ACR**

**Build the new Dockerfiles, if necessary**

If you are using the Azure\_full\_stack repo, then execute each of these commands in the directories indicated.

from compose\_postrgres:

docker build -f Dockerfile.pgdb -t fskubecr.azurecr.io/kanban-pg:latest .

docker build -f Dockerfile.pgadmin -t fskubecr.azurecr.io/kanban-db-admin:latest .

from server/app:

docker build -f Dockerfile fskubecr.azurecr.io/kanban-web:latest .

All the Docker images are now built, so they need to be uploaded to the ACR. This process makes the images available to AKS to build out our Application and Database, as well as the DB admin tool.

**Log in to the ACR**

$ az acr login --name fskubecr

**Get the ACR login server connection information**

$ az acr list -g FSKubeRG --query "[].{acrLoginServer:loginServer}" --output table

AcrLoginServer

-------------------

fskubecr.azurecr.io

**Push the images with the ACR domain in Docker**

Using the images just built with Docker on the local machine, push each image to the repository location:

$ docker push fskubecr.azurecr.io/kanban-web:latest

$ docker push fskubecr.azurecr.io/kanban-db-admin

$ docker push fskubecr.azurecr.io/kanban-pg

Wait for the push to complete, then list the images in the Azure CR:

$ az acr repository list --name FSKubeCR --output table

Result

---------------

kanban-db-admin

kanban-pg

kanban-web

Showing the tags for a specific image can be done as follows:

Example:

$ az acr repository show-tags --name FSKubeCR --repository kanban-pg --output table

**Create persistent volume storage**

In order for Postgres DB to survive updates, some crashes & pod restarts, the storage needs to be provisioned from outside the node and mounted on a directory in the cluster. This is done via Persistent Storage Volumes and Storage Claims.

Navigate to the k8s directory under the kanban-life repo.

As we are deploying and application with a database backend, we want to make sure our data persists, Azure can provide a Disk resource to use with our pods. This disk can be associated with an AKS resource using Kubernetes.

pg*\_storage\_setup.yaml*

apiVersion: v1

kind: PersistentVolume

metadata:

name: kanban-postgres01

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Recycle

---

apiVersion: v1

kind: PersistentVolume

metadata:

name: kanban-postgres2

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Recycle

This yaml defines a persistent “volume” named Kanban-postgresNN -disk and allocates 1gb of space to the volume

To apply this yaml and set up the storage, use kubectl to apply the code:

$ kubectl apply -f pg\_storage\_setup.yaml

Examine the volume created

<FIXME>

**Deploy the Postgres Backend to FSKubeCluster**

The next step is deploying the architecture is to deploy the Postgres backend. This is different than a deployment as it is a StatefulSet. StatefulSets maintain their “identies” during deployment and recreation to make updating and other maintenance tasks cause less disruption.

*Deploy\_backend.yaml <FIXME>*

apiVersion: v1

kind: Service

metadata:

name: postgres

labels:

db: postgres

spec:

selector:

db: postgres

ports:

- port: 5432

name: postgres

clusterIP: None

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: postgres

spec:

serviceName: "postgres"

selector:

matchLabels:

db: postgres

tier: backend

replicas: 2

template:

metadata:

labels:

db: postgres

tier: backend

spec:

containers:

- name: kanban-pg

image: fskubecr.azurecr.io/kanban-pg

imagePullPolicy: Always

ports:

- name: postgres

containerPort: 5432

volumeMounts:

- name: data

mountPath: /data

subPath: pgdata

env:

- name: POSTGRES\_USER

value: postgres

- name: POSTGRES\_PASSWORD

value: changeme

- name: POSTGRES\_DB

value: pgkanban

- name: PGDATA

value: /data/pgdata

initContainers:

- name: volume-mount-hack

image: busybox

command: ["sh", "-c", "chown -R 999:999 /data"]

volumeMounts:

- name: data

mountPath: /data

volumeClaimTemplates:

- metadata:

name: data

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 1Gi

This code includes a service so that other members of the cluster can find the DB. It also establishes that the numbers of replicas be 2 – that is, 2 pods will contain the exact same image and Kubernetes will use the service to find what it needs – essentially using internal loadbalancing. Other “in cluster” resources can refer to the service by using “postgres” as a de facto internal DNS name.

This code also mounts the persistent storage volume for safeguarding the DB Data directory.

$ kubectl apply -f deploy\_backend.yaml

After the deployment completes, watch for the Public IP to populate for the containers:

$ kubectl get service postgres

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

postgres ClusterIP None <none> 5432/TCP 83m

As shown in the output above, the DB has NO access through an external IP – that is, it cannot be reached from outside the cluster. This is a good first step in securing the DB access.

**Deploy PGAdmin frontend**

Now that the Database is deployed, there needs to be a means to administer it. PgAdmin is the long-standing tool to do this and suits the need well here.

*Deploy\_frontend.yaml*

apiVersion: v1

kind: Service

metadata:

name: pgadmin

spec:

type: LoadBalancer

selector:

app: pgadmin

tier: frontend

ports:

- port: 5050

targetPort: 80

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: pgadmin

spec:

replicas: 1

selector:

matchLabels:

app: pgadmin

tier: frontend

template:

metadata:

labels:

app: pgadmin

tier: frontend

spec:

containers:

- name: kanban-db-admin

image: fskubecr.azurecr.io/kanban-db-admin:latest

imagePullPolicy: Always

ports:

- containerPort: 5050

env:

- name: PGADMIN\_DEFAULT\_EMAIL

value: pgadmin4@pgadmin.org

- name: PGADMIN\_DEFAULT\_PASSWORD

value: admin

This code also sets up a service, unlike the one for the Database though, it needs to be accessible to the Web so that admins can log in and use the tool. PgAdmin itself will use the DNS name “postgres” to find the DB in the cluster, as will be shown during the PgAdmin setup. For now, the service will spin up a loadbalancer in Azure and connect it to the AKS and the cluster resource for the PgAdmin. In fact, if you look at the Loadbalancer resources after deploying the yaml file, the new balancer is shown as a resource.

A screenshot of a cell phone

Description automatically generated

The next bit of the yaml code defines this as a Deployment, this time without persistent storage. If a admin desires, another storage volume could be created and attached to this deployment, just like the Database deployment.

The deployment is applied to the AKS:

$ kubectl apply -f deploy\_frontend.ymal

Once the deployment completes, the following sanity checks show the state of the service & deployment

$ kubectl get service pgadmin

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

pgadmin LoadBalancer 10.0.160.124 52.191.101.157 5050:32507/TCP 18h

$ kubectl get deployment pgadmin

NAME READY UP-TO-DATE AVAILABLE AGE

pgadmin 1/1 1 1 22h

**Deploying the Web Application**

The last piece of deployment is the web app itself. The app is built using Node.js, Express and Websocket.io. While the code is fairly simply, the basic workings are:

* Node creates an HTTP server backed by express
* The Node routings server a webpage, marked up with JavaScript to provide updates to the page without refreshing using web sockets
* A user accesses the public IP for the app and opens a socket the webpage
* Node Listens for traffic across the sockets and updates any connected clients when things change

The screen shots following the deployment instructions will illustrate the interaction of the WWW, AKS load balancing and resulting actions from Node server.

For now, to deploy the app, the following yaml is applied to the AKS

*Deploy\_web.yaml*

apiVersion: v1

kind: Service

metadata:

name: kanban-web

spec:

type: LoadBalancer

selector:

app: kanban-web

tier: frontend

ports:

- port: 4001

targetPort: 4001

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: kanban-web

spec:

replicas: 2

selector:

matchLabels:

app: kanban-web

tier: frontend

template:

metadata:

labels:

app: kanban-web

tier: frontend

spec:

containers:

- name: kanban-web

image: fskubecr.azurecr.io/kanban-web:latest

imagePullPolicy: Always

ports:

- containerPort: 4001

env:

- name: DB\_HOST

value: postgres

- name: DB\_USER

value: postgres

- name: DB\_PASS

value: changeme

- name: DB\_NAME

value: pgkanban

This code also sets up another load balancer and routes traffic from the IP to the Node servers. Reminder that all the cluster assets can see the other members and inter-cluster traffic is unimpeded by the presence of the load balancers.

Deploy the yaml

$ kubectl -f deploy\_web.yaml

**Checking cluster health**

Now that all resources are deployed, it is time to take a look at everything that is created. The following checks on the Pods, Services, Deployments and StatefulSets reveals how everything is setup, connected and running:

Show pods

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

jenkins-746cd8fcf7-4wbv7 1/1 Running 0 46h

kanban-web-86bf444fcd-gtc69 1/1 Running 1 18h

kanban-web-86bf444fcd-m97xp 1/1 Running 1 18h

pgadmin-7485b667bd-vq52j 1/1 Running 0 18h

postgres-0 1/1 Running 0 4m39s

postgres-1 1/1 Running 0 3m57s

Show deployments

$ kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

kanban-web 2/2 2 2 22h

pgadmin 1/1 1 1 23

Show StatefulSets

$ kubectl get statefulsets

NAME READY AGE

postgres 2/2 22h

Show services

$ kubectl get services

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

jenkins LoadBalancer 10.0.144.158 52.147.214.172 8080:30499/TCP 46h

kanban-web LoadBalancer 10.0.135.73 52.191.101.213 4001:30063/TCP 18h

pgadmin LoadBalancer 10.0.160.124 52.191.101.157 5050:32507/TCP 18h

postgres ClusterIP None <none> 5432/TCP 85m 24h

Notice only PgAdmin & Kanban-web are exposed to the WWW.

**Testing the Application**

With everything up and running, functionality can be demostrated.

**Setup PgAdmin**

PgAdmin needs to be connected to the DB using the Server create. Log in the IP at port 5050 to setup PgAdmin

#### Setup Continous Intergration/Continous deployments

Jenkins will be the tool of choice here. Additional Pod deployments will place Jenkins in the Kanban cluster and allow for automated and on-demand building of the Kanban application

**Creating the Jenkins Docker Image**

The first step will be to create a Dockerfile with almost everything needed to setup Jenkins in AKS. After creation, the image will be pushed to the ACR, just like the other application images built for Kanban. After that is completed, the image will be deployed on the FSKubeCluster and share space with the other associated applications.

*Dockerfile*

FROM jenkins/jenkins:lts-alpine

LABEL maintainer="Harvash <harvash2004@yahoo.com>"

ENV JENKINS\_USER admin

ENV JENKINS\_PASS admin

# Skip initial setup

ENV JAVA\_OPTS -Djenkins.install.runSetupWizard=false

COPY jenkins\_plugin\_list.txt /usr/share/jenkins/plugins.txt

RUN /usr/local/bin/install-plugins.sh < /usr/share/jenkins/plugins.txt

# Scaling

RUN /usr/local/bin/install-plugins.sh kubernetes

USER root

RUN apk update && apk add gcc libc-dev make git libffi-dev openssl-dev python3-dev libxml2-dev libxslt-dev

ENV PYTHONUNBUFFERED=1

RUN echo "\*\*\*\* install Python \*\*\*\*" && \

apk add --no-cache python3 && \

if [ ! -e /usr/bin/python ]; then ln -sf python3 /usr/bin/python ; fi && \

\

echo "\*\*\*\* install pip \*\*\*\*" && \

python3 -m ensurepip && \

rm -r /usr/lib/python\*/ensurepip && \

pip3 install --no-cache --upgrade pip setuptools wheel && \

if [ ! -e /usr/bin/pip ]; then ln -s pip3 /usr/bin/pip ; fi

RUN pip install docker-compose && \

pip install openshift && \

pip install ansible && \

pip install azure-cli && \

pip install packaging && \

pip install cx-Oracle && \

pip install jmespath && \

pip install kubernetes && \

pip install msrestazure

USER jenkins

As seen in the Dockerfile, several OS level dependencies are installed with PIP (Python package manager), along with a manifest of needed Jenkins plugins so that they need not be configured after deployment. The plugins are in a file in the Jenkins directory with the Dockerfile and can be updated and the build re-made (although, the image still needs to be re-pushed to ACR)

Be aware, this creates a fairly large image (1+GB) and will take a while to push up to the Cloud.

$ docker push [fskubecr.azurecr.io/jenkins:v1](http://fskubecr.azurecr.io/jenkins:v1)

**Setting up the Jenkins server**

There is still some work to do in order to fully utilize the CI/CD capabilities.

# First, create the RSA keys to talk to Github (do NOT commit these – add them to .gitignore

# cd to the Jenkins directory in the Kanban Repo

$ ssh-keygen -t rsa -b 4096 -C "k8s.jenkins"

Create a pod & storage volume for Jenkins to use for persistent storage:

*Jenkins\_storage\_setup.yaml*

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: kanban-jenkins-pv

spec:

accessModes:

- ReadWriteOnce

storageClassName: default

resources:

requests:

storage: 5Gi

*deploy\_jenkins.yaml*

apiVersion: v1

kind: Service

metadata:

name: jenkins

spec:

type: LoadBalancer

selector:

app: jenkins

tier: frontend

ports:

- port: 8080

targetPort: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: jenkins

spec:

selector:

matchLabels:

app: jenkins

tier: frontend

replicas: 1

template:

metadata:

labels:

app: jenkins

tier: frontend

spec:

securityContext:

fsGroup: 1000

containers:

- name: jenkinsci

image: fskubecr.azurecr.io/jenkins:v1

imagePullPolicy: Always

ports:

- name: jenkins

containerPort: 8080

volumeMounts:

- name: jenkins-pvc

mountPath: "/var/jenkins\_home"

- name: ssh-keys

readOnly: true

mountPath: /etc/ssh-keys

volumes:

- name: jenkins-pvc

persistentVolumeClaim:

claimName: kanban-jenkins-pv

Deploy the items:

$ kubectl apply -f jenkins\_storage\_setup.yaml

$ kubectl apply -f deploy\_jenkins.yaml

Copy the keys to the Jenkins SSH directory, along with SSH Config, which is in the Jenkins repo directory

$ kubectl cp ./jenkins\_rsa jenkins-6947d46b7-rqd2n:/var/jenkins\_home/.ssh

$ kubectl cp ./jenkins\_rsa.pub jenkins-6947d46b7-rqd2n:/var/jenkins\_home/.ssh

$ kubectl cp ./ssh\_config jenkins-6947d46b7-rqd2n:/var/jenkins\_home/.ssh/config

The proper keys need to be uploaded to the Kanban repo being used, the following should get them in the right place.

* Login to Github
* Select the Kanban repo
* Click user icon in upper right > settings > SSH and GPG keys
* New SSH key >> paste jenkins\_rsa

**Jenkins Configuration**

Navigate to the External Pod IP at port 8080 to access Jenkins and set configuration, as well as creating first user

A screenshot of a cell phone

Description automatically generated

From the menu:

Manage > Configuration click Save without changing anything (this sets Jenkins URL's)

Manage > Configure Global Security > Authorization > tick "Jenkins’ own user database" and check "Logged in users can do anything"

Add your username and password to create first admin user

Next, add the Azure Service Principal information to Credentials:

Manage > Manage credentials > Under “Stores scoped to Jenkins”, click Jenkins > Global Credentails > Add Credential

A screenshot of a cell phone

Description automatically generated

After creating the SP entry, using the same method above, create a new credential for the SSH information:

A screenshot of a cell phone

Description automatically generated

**Create the Jenkins Job**

Login to Jenkins using the External IP from Pods to

New Item > Freestyle Job = deploy\_kanban

Under source code management, select ‘Git’ and add the repo & credential

A screenshot of a cell phone

Description automatically generated

Further down in the configuration, tick the Generic Webhook Trigger checkbox and add the following to Token & Cause boxes:

A screenshot of a cell phone

Description automatically generated

Then tick: Use secret text(s) or file(s) > Add > Microsoft Azure Service Principal

A screenshot of a cell phone

Description automatically generated

Lastly, click “Build” > Invoke Ansible Playbook

A screenshot of a cell phone

Description automatically generated

Click “Save” to complete the job configuration.

**Setting up GitHub Webhook**

Login to Github and select the Kanban repo

Click settings gear and select “Webhooks” from the side menu, as well as “Add webhook” in the main pane. In the Payload URL, enter: http://52.147.214.172:8080/generic-webhook-trigger/invoke?token=automated\_git\_update

A screenshot of a cell phone

Description automatically generated

#### Testing CI/CD Pipeline

Once Jenkins and GitHub are provisioned as above, builds maybe deployed either manually or by commit to the Master repo branch.

To test a manual deployment, delete the pgadmin deployment

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

jenkins-746cd8fcf7-4wbv7 1/1 Running 0 2d16h

kanban-web-86bf444fcd-gtc69 1/1 Running 1 37h

kanban-web-86bf444fcd-m97xp 1/1 Running 1 37h

pgadmin-7485b667bd-vq52j 1/1 Running 0 37h

postgres-0 1/1 Running 0 18h

postgres-1 1/1 Running 0 18h

$ kubectl delete deployment pgadmin

deployment.extensions "pgadmin" deleted

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

jenkins-746cd8fcf7-4wbv7 1/1 Running 0 2d17h

kanban-web-86bf444fcd-gtc69 1/1 Running 1 38h

kanban-web-86bf444fcd-m97xp 1/1 Running 1 38h

postgres-0 1/1 Running 0 19h

postgres-1 1/1 Running 0 19h

Now, login to Jenkins > Kanban-Deploy > Build Now

To test the CI/CD interaction, update the Port field of the deploy\_frontend.yaml from 5050 to 5051, commit & push the updated code:

*Deploy\_fronend.yaml (change port)*

containers:

- name: kanban-db-admin

image: fskubecr.azurecr.io/kanban-db-admin:latest

imagePullPolicy: Always

ports:

- containerPort: 5051 # changed from 5050 to 5051

Pod Services BEFORE deploying manually: