What The Hack – Proctor’s Guide

Kubernetes as Infrastructure

# Challenge Set 0: Pre-requisites - Ready, Set, GO!

## Lecture:

* Be a smart Cloud Architect.
* Always be sure to have right tools in your toolbox!
* Join the MS Team for this hack.

## Challenges:

* Make sure that you have joined the Teams group for this track. The first person on your team at your table should create a new channel in this Team with your team name.
* Install the recommended tool-set:
  + Windows Subsystem for Linux
  + Azure CLI
    - Update to the latest
    - Must be at least version 2.0.42
    - **NOTE:** If you’re running into issues running Azure CLI command on Windows, Disable Global Protect (VPN)
  + Visual Studio Code
* **Note:** You can start the next challenge even if this one is still running by using the the Azure Cloud Shell.
* **Tip:** You can complete almost all of the challenges with the Azure Cloud Shell! But be a good cloud architect and make sure you have experience installing the tools locally.

## Proctor Notes & Guidelines

* Install the WSL: <https://docs.microsoft.com/en-us/windows/wsl/install-win10>
* Install the Azure CLI in the WSL: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>
* Install VS Code: <https://code.visualstudio.com/>
* Optionally install Azure Storage Explorer: <http://storageexplorer.com>

# Challenge Set 1: Got Containers?

## Lecture:

* Introduce the FabMedical app and its components
  + It’s an app that manages medical conferences and speakers at the events
* It is written in node.js
* Introduction to Docker
* Introduction to containerizing your application

## Challenges:

* Deploy build agent VM with Linux + Docker using provided ARM Template & parameters file in the “Files” tab of the Team’s General channel.
* Run the Fab Medical application locally on the VM & verify access
  + Each part of the app (api & web) runs independently.
  + Build the API app by navigating to the **content-api** folder and run “**npm install**”.
  + To start the app, run “**nodejs ./server.js &**”
  + Verify the API app runs by hitting its URL with one of the three function names. Eg: “**http://localhost:3000/speakers**”
  + Repeat for the steps above for the content-web app, but verify it’s available via a browser on the Internet!
  + **NOTE:** The content-web app expects an environment variable named “**CONTENT\_API\_URL**” that points to the API app’s URL.
* Create a Dockerfile for the content-api app that will:
  + Create a container based on the node:8 container image
  + Build the Node application like you did above (**Hint:** npm install)
  + Exposes the needed port
  + Starts the node application
* Create a Dockerfile for the content-web app that will:
  + Do the same as the Dockerfile for the content-api
  + Also sets the environment variable value as above
* Build Docker images for both content-api & content-web
* Run both containers you just built and verify that it is working.
  + **Hint:** Run the containers in ‘detached’ mode so that they run in the background.
  + **NOTE:** The containers need to run in the same network to talk to each other.
    - Create a Docker network named “fabmedical”
    - Run each container using the "fabmedical” network
    - **Hint:** Each container you run needs to have a “name” on the fabmedical network and this is how you access it from other containers on that network.
    - **Hint:** You can run your containers in “detached” mode so that the running container does NOT block your command prompt.

## Proctor Notes & Guidelines

* Deploy build agent VM with Linux + Docker using provided ARM Template
  + **az group create –g <resourcegroupname> -l <region>**
  + **az group deployment create –g <rgname> -n <deploymentName> --template-file docker-buildagent-vm.json --parameters @docker-buildagent-vm.parameters.json**
* Get the Fab Medical code from the tarball stored in our git repo under:
  + /001-IntroToKubernetes/Student/Resources/Code/FabMedical.v1.tar.gz
* Run the Fab Medical application locally on the VM & verify access
  + Each part of the app (api & web) runs independently.
  + Build the API app by navigating to the content-api folder and run “**npm install**”.
  + To start a node app, run “**nodejs ./server.js &**”
  + Verify the API app runs by checking it at its URL with one of the three function names. I.e. “**http://localhost:3000/speakers**”
  + Repeat for the steps above for the Web app.
  + **Note:** The content-web app expects an environment variable named “**CONTENT\_API\_URL**” that points to the API app’s URL.
  + Environment variable value should be “**http://localhost:3001**”
* Some reference articles on how to Dockerize a Node.js app:
  + <https://nodejs.org/en/docs/guides/nodejs-docker-webapp/>
  + <https://buddy.works/guides/how-dockerize-node-application>
  + <https://www.cuelogic.com/blog/why-and-how-to-containerize-modern-nodejs-applications>
* Dockerfiles for both content-api and content-web are in the proctor’s resources folder!
* Build Docker images for both content-api & content-web. **NOTE:** there is a “.” at the end of each of the commands below!
  + **docker build –t content-api .**
  + **docker build –t content-web .**
* Run the application in the Docker containers in a network and verify access
  + Create a Docker network named “fabmedical”:
    - **docker network create fabmedical**
  + Run each container using a name and using the "fabmedical” network. The containers should be run in “detached” mode so they don’t block the command prompt.
    - **docker run -d -p 3001:3001 --name api --net fabmedical content-api**
    - **docker run -d -P --name web --net fabmedical content-web**

# Challenge Set 2: Azure Container Registry

## Lecture:

* Introduction to Container Registries
* Introduction to the Azure Container Registry

## Challenges:

* Deploy an Azure Container Registry (ACR)
* Ensure your ACR has proper permissions and credentials set up
* Login to your ACR
* Push your Docker container(s) to the ACR
* List all images in your ACR

## Proctor Notes & Guidelines

* Proctor guide material needed for ACR in general

# Challenge Set 3: Introduction to Kubernetes

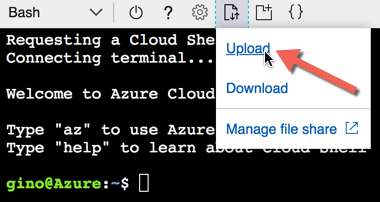
## Lecture:

* Introduction to Kubernetes
* Introduction to the Azure Kubernetes Service

## Challenges:

* Install the Kubernetes command line tool (kubectl).
  + **Hint**: This can be done easily with the Azure CLI
* Create a new, multi-node AKS cluster with **RBAC disabled.**
  + Use a single core DS1v2 machine for your worker nodes.
  + Use the latest version of Kubernetes supported by AKS.
* Use kubectl to prove that the cluster is a multi-node cluster and is working
* Bring up the Kubernetes dashboard in your browser
  + **Hint**: Again, the Azure CLI makes this very easy.

## Proctor Notes & Guidelines

* This first hour is the most critical to make sure all teams make it through and aren’t left behind.
  + Make sure that ALL team members have joined the Teams group for this track and created a channel named after their team name.
  + Make sure that ALL team members are set up with the tools.
  + Encourage teams to use the CLI and WSL to make things easier.
  + **NOTE**: If someone has to install the tools, make sure they first create the cluster in the portal and then install the tools in parallel.
    - To upload files to the Cloud Shell, use the upload button:  
      
    - Show them BOTH of these if they are unfamiliar with how to navigate to them.
* Remind teams that kubectl can be installed through the CLI, but don’t give away the answer:
  + **az aks install-cli**
* All teams should have an AKS cluster stood up fairly quickly
  + Version shouldn’t matter, but good to make sure it is the latest
  + Keep it simple: Basic networking, RBAC disabled, let it create a new service principal.
  + **NOTE**: Sometimes during the validation step when creating a new cluster, it will fail because it cannot find the new Service Principal. This is a timing issue. Click the Previous button to go back one page and then Forward to redo the validation.
  + They can use the CLI for this with a simple command to make a 3-node cluster:
    - **az aks create --resource-group myAKSCluster --name myAKSCluster --node-count 3 --generate-ssh-keys --disable-rbac**
  + Docs to install AKS:
    - Portal: <https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough-portal>
    - CLI: <https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough>
* Have the teams show you the running cluster with:
  + **kubectl get nodes**
  + It should have more than 1 node.
  + **NOTE**: They will need to learn how to connect kubectl to their cluster using “az aks get-credentials”.
* The Kubernetes dashboard can be brought up with the CLI easily:
  + **az aks browse --name myAKSCluster --resource-group myAKSCluster**

# Challenge Set 4: Your First Deployment

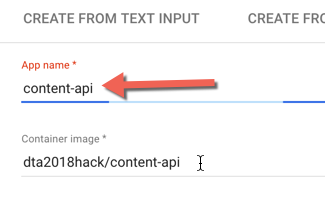
## Lecture:

* Review the basic components of Kubernetes
  + The physical architecture
  + Pods
  + Deployments
  + Services
    - Talk about externally facing services using a load balancer
    - In AKS this is an Azure LB
* Review the structure of YAML files for Deployments and Services
* Introduce the web app for FabMedical and its dependencies on the API app.

## Challenges:

* **NOTE:** If you have not or cannot deploy your containers to the Azure Container Registry, we have staged the FabMedical apps on Docker Hub at these locations:
  + **API app**: whatthehackmsft/content-api
  + **Web app**: whatthehackmsft/content-web
* Deploy the **API app** through the Kubernetes dashboard using these settings:
  + Number of pods: 1
  + Service: Internal
  + Port and Target Port: 3001
  + CPU: 0.5
  + Memory: 128MB
* We have not exposed the API app to the external world. Therefore, to test it you need to:
  + Figure out how to get a bash shell on the API app pod just deployed.
  + Curl the url of the “/speakers” end point.
  + You should get a huge json document in response.
* Deploy the **Web app** from the command line using kubectl and YAML files
  + **NOTE**: Sample YAML files to get you started can be found in the Files section of the General channel in Teams.
  + **NOTE**: The Web app expects to have an environment variable pointing to the URL of the API app named:
    - **CONTENT\_API\_URL**
  + Create a deployment yaml file for the Web app using the specs from the API app, except for:
    - Port and Target Port: 3000
  + Create a service yaml file to go with the deployment
    - **Hint**: Not all “types” of Services are exposed to the outside world
  + **NOTE:** Applying your YAML files with kubectl can be done over and over as you update the YAML file. Only the delta will be changed.
  + **NOTE**: The Kubernetes documentation site is your friend. The full YAML specs can be found there: <https://kubernetes.io/docs>
* Find out the External IP that was assigned to your service. You can use kubectl or the dashboard for this.
* Test the application by browsing to the Web app’s external IP and port and seeing the front page come up.
  + Ensure that you see a list of both speakers and sessions on their respective pages.
  + If you don’t see the lists, then the web app is not able to communicate with the API app.

## Proctor Notes & Guidelines

* Students need to figure out on their own where in the dashboard you create an app.
  + Click “+ CREATE” button on the top right
  + Use the “CREATE AN APP” tab
  + Advanced Settings will be needed for CPU and Memory
* Students will need to find extra settings to add to their template YAML files. They should make use of the Kubernetes docs in addition to whatever else they can find on the web.
  + **NOTE**: Fully fleshed out YAML files are available to proctors in the Files area of the Proctor’s Infra channel on Teams.
  + **NOTE**: If they go to “Edit Deployment” on the API app deployed from the dashboard, they will see the full YAML for that deployment. This hint should be saved until they get desperate.
* In the Deployment YAML, they’ll need these settings in the spec section:
  + containers.resources.requests.cpu: 0.5 (or 500m)
  + containers.resources.requests.memory: 128Mi
  + containers.ports.containerPort: 3001
  + containers.env.name: CONTENT\_API\_URL
  + containers.env.value: <http://content-api:3001>
    - The value “content-api” in the URL must be whatever was used as the name of the service during deployment. Also seen on this screen when creating in the portal:  
      
* In the Service YAML, they need to figure out that the type should be changed to “LoadBalancer”.
* When the service is deployed it will take some time for an External IP to be assigned.
  + In the dashboard, go to the Services page and look at the “External Endpoints” column
  + Issue the following kubectl command and look in the “EXTERNAL-IP” column
    - **kubectl get services**
  + You will see “<pending>” if the IP hasn’t yet been assigned.
* To verify that the API app is correct deployed the students need to:
  + Figure out the name of the pod the API app was deployed to, eg: content-api-23aceed
  + Then use a kubectl command like this to get a bash shell:
    - **kubectl exec -it content-api-23aceed -- /bin/bash**
  + To verify the API app is working curl the /speakers endpoint:
    - **curl** [**http://localhost:3001/speakers**](http://localhost:3001/speakers)
* They should see a huge JSON document printed to the screen.

# Challenge Set 5: Scale and High Availability

## Lecture:

* Review scaling in Kubernetes, scaling pods vs. adding nodes to the cluster
* Show an example of scaling and how Kubernetes treats the replicas number as a desired state.
* Discuss scaling out the cluster and how this is a manual process (for now!)
* Review HA in Kubernetes, how it is pod-centric.

## Challenges:

* Scale the Web app to 2 instances
  + This should be done by modifying the YAML file for the Web app and re-deploying it.
* Scale the API app to 4 instances
  + This should be done through the Kubernetes dashboard.
* Watch the ReplicaSets and Pods pages in the dashboard to see how they change.
  + You will find an error occurs because the cluster does not have enough resources to support that many instances.
  + There are two ways to fix this: increase the size of your cluster or decrease the resources needed by the deployments.
* To fully deploy the application, you will need 4 instances of the API app running and 2 instances of the Web app.
  + **Hint:** If you fixed the issue above correctly, you should be able to do this with the resources of your original cluster.
* When your cluster is fully deployed, browse to the “/stats.html” page of the web application.
  + Keep refreshing to see the API app’s host name keep changing between the deployed instances.
* Scale the API app back down to 1, and immediately keep refreshing the “/stats.html” page.
  + You will notice that without any downtime it now directs traffic only to the single instance left.

## Proctor Notes & Guidelines

* In the YAML file, they will have to update the “spec.replicas” value.
* The error they will encounter is that there aren’t enough CPUs in the cluster to support the number of replicas they want to scale to.
* The two fixes are:
  + Use the Azure portal to add more nodes to the AKS cluster.
  + Change the deployment and reduce the needed CPU number from “0.5” to “0.125” (500m to 125m).
    - This is the preferred solution.
* **NOTE**: In case they do NOT get an error and are able to scale up, check how many nodes they have in their cluster and the size of the node VMs. Over provisioned clusters **will not fail**.
  + If a team doesn’t get a failure, just have them double the number of Web and API app instances.

# Challenge Set 6: Deploy MongoDB to AKS

## Lecture:

* Explain that MongoDB is needed for V2 of the app
* We’re going to deploy it quickly to set us up for the app update and storage challenges

## Challenges:

* Deploy a MongoDB container in a pod for v2 of the FabMedical app
* **Hint:** Check out the Docker Hub container registry and see what you can find.
* Confirm it is running with:
  + **kubectl exec -it <mongo pod name> -- mongo "--version"**

## Proctor Notes & Guidelines

* The original mongo container maintained by the mongo project is at:
  + <https://hub.docker.com/_/mongo/>
* Bitnami also provides a mongo container that can be found here:
  + [https://hub.docker.com/r/bitnami/mongodb/](o%09https:/hub.docker.com/r/bitnami/mongodb)

# Challenge Set 7: Updates and Rollbacks

## Lecture:

* Review deployments
* Explain Kubernetes rolling updates mechanism as well as deployment rollbacks.
* Mention Blue/Green deployments (similar to deployment slots) for apps that can’t handle rolling updates.

## Challenges:

* We have staged an updated version of the app on Docker Hub with id and version:
  + **whatthehackmsft/content-web:v2**
  + **whatthehackmsft/content-api:v2**
* For version two, you will also need an initializer container available on Docker Hub at:
  + **whatthehackmsft/content-init**
  + Use the content-init “Job” yaml provided to run the initialization of MongoDB for our new version of the app.
* Perform a rolling update of the Web app on your cluster to the new version two of content-web
  + You’ll be doing this from the command-line with a kubectl command (remember, Kubernetes docs are your friend!)
  + In the Kubernetes dashboard on the Pods page, you should be able to see new pods with the new version come online and the old pods terminate
    - You can also do this by listing the pods with kubectl.
  + At the same time, hit the front page to see when you’re on the new version by refreshing constantly until you see the conference dates updated to 2019.
* Now roll back this update.
  + Again, this is done from the command-line using a (different) kubectl command.
  + Confirm that we are back to the original version of the app by checking that the conference dates are back to 2017.
* Perform the update again, this time using the blue/green deployment methodology.
  + You will need a separate deployment file using different tags.
  + Cut over is done by modifying the app’s service to point to this new deployment.

## Proctor Notes & Guidelines

* They will need to use the “kubectl set image” command to perform the rolling update:
  + **kubectl set image deployment/content-web content-web=whatthehackmsft/content-web:v2**
    - Where “deployment/content-web” is the name of the deployment used
  + “kubectl set image” takes the name of the deployment and the new image to update to.
  + **NOTE:** Use a similar command to perform a rolling update on content-api
* **kubectl get pods –** Running this will show all the pods getting updated and terminated.
* Rollbacks are performed with:
  + **kubectl rollout undo deployment/content-web**
  + This will roll-back the last update to the “content-web” deployment.
* Blue/Green deployments are described here:
  + <https://www.ianlewis.org/en/bluegreen-deployments-kubernetes>
  + Basically, they will create a separate deployment YAML with different tags and deploy it.
    - **NOTE:** The “content-web-deploy-solution.bluegreen.yaml” solution file is an example of an updated deployment using the v2 flag.
  + When the new pods are ready to go, they will update the service YAML to point to the new tags.

# Challenge Set 8: Storage

## Lecture:

* Explain Kubernetes storage concepts.
* Highlight the cloud storage and AKS capabilities
* Mention that attendees will be leveraging the static persistent storage (azure disk) for storing the mongo DB configuration and data

## Challenges:

* Make sure that you are using the latest version of the Fabmedical container images:
  + **whatthehackmsft/content-api:v2**
  + **whatthehackmsft/content-web:v2**
* Destroy the previous MongoDB pod created in the Challenge Set 6.
* In this challenge you will provision the MongoDB pod with a persisted disk volume.
* Create two Azure data disks (one for the MongoDB configuration and another one for data)
* Create a deployment yaml for MongoDB to be deployed with the necessary configuration for using the volume as an Azure Data Disk.
  + Find the reference template in the Teams Files section: **tempate-mongodb-deploy.yml**
  + **NOTE**: You can use the same MongoDB container image from Docker Hub that you used in a previous challenge.
* Verify that MongoDB is working fine by connecting to the corresponding MongoDB Pod in the interactive mode. Make sure that the disks are associated correctly (Highlighted below)
* **kubectl exec -it <mongo-db pod name> bash**

root@mongo-db678745655b-f82vj:/# **df -Th**  
Filesystem     Type     Size  Used Avail Use% Mounted on  
overlay        overlay   30G  4.2G   25G  15% /  
tmpfs          tmpfs    1.7G     0  1.7G   0% /dev  
tmpfs          tmpfs    1.7G     0  1.7G   0% /sys/fs/cgroup  
/dev/sdc       ext4     2.0G  304M  1.5G  17% /data/db  
/dev/sdd       ext4     2.0G  3.0M  1.8G   1% /data/configdb  
/dev/sda1      ext4      30G  4.2G   25G  15% /etc/hosts  
shm            tmpfs     64M     0   64M   0% /dev/shm  
tmpfs          tmpfs    1.7G   12K  1.7G   1% /run/secrets/kubernetes.io/serviceaccount  
tmpfs          tmpfs    1.7G     0  1.7G   0% /sys/firmware

* root@mongo-db678745655b-f82vj:/# **mongo --version**  
  MongoDB shell version v3.6.1  
  connecting to: mongodb://127.0.0.1:27017  
  MongoDB server version: 3.6.1
* Initialize sample content (Speakers & Sessions data) in the mongo DB by running the content\_init nodeJS application as a Kubernetes Job. Reference template is can be found in the Files area in Teams, called: **template-content-init-deploy.yml**
  + Logs for content-init will provide the detailed logs showing whether it was able to successfully connect and add the contents to the MongoDB. You can use the Kubernetes dashboard or kubectl to check the logs.
  + **NOTE**: If the AKS cluster was created using the default Service Principle then we must grant it permission to pull images from the ACR.
    - **Hint**: Have a look here: <https://docs.microsoft.com/en-us/azure/container-registry/container-registry-auth-aks>)
* Make sure that the “contentdb” database is populated by connecting to the MongoDB pod with an interactive terminal and verify the database collections.
* root@mongo-db678745655b-f82vj:/# **mongo**  
  MongoDB shell version v3.6.1  
  connecting to: mongodb://127.0.0.1:27017  
  MongoDB server version: 3.6.1  
  >  
  > **show dbs**  
  admin       0.000GB  
  config      0.000GB  
  contentdb   0.000GB  
  local       0.000GB
* Destroy the MongoDB pod to prove that the data persisting to the disk
  + **kubectl delete deployment <mongo-db-deployment>**
* Recreate the Mongo Db Pod
  + **kubectl apply -f <mongo-db-deployment>**
* Once the Pod is created, verify that data is persisted to the Azure disks by following the previous MongoDB verification step.
* Update the MongoDB connection string in the content-api deployment YAML and deploy it, eg:
  + env:  
     - name: MONGODB\_CONNECTION  
     value: mongodb://mongodb:27017/contentdb
* Verify the API can retrieve the data by calling the speaker / session end points with curl:
  + curl <http://localhost:3001/speakers>
  + curl http://localhost:3001/sessions

## Proctor Notes & Guidelines

* Need to make sure that the attendees are using the latest container images including the content-init nodejs container
* Make sure that the attendees are verifying the MongoDB connection, data and disks by connecting to the MongoDB with an interactive terminal.
* Make sure that the attendees understand the concept of storage volumes and how AKS provides value by providing the azure disk / file storage in both dynamic and static mode.

# Challenge Set 9: Helm

## Lecture

* There are multiple ways to make meatballs. You can prepare the meatball sauce and meatballs from scratch using raw ingredients separately or you can simply open a pre-packaged meatball ready-made meal and warm it up in a pan or microwave.
* Helm provides a simplified interface for making a “meatball dinner” metaphorically without having to worry about or go through all the individual steps of assembling the ingredients necessary to make the meal. It also saves time. With Helm, the different YAML components are packaged into a representation known as a Chart and the chart contains all the metadata necessary to deploy the application on the k8s cluster.
* The Helm client needs to be installed locally and the Tiller server needs to be installed on the Kubernetes cluster.
* The Helm charts needs to be made available for deployment. We could use a git repo as the repository or we could use Azure Container Registry to host the pre-packaged charts. We can also create a package and make it available in a local repo and use that for deployment as well.

## Challenges:

* Fetch the script for installing Helm to the local machine where you will be using Helm
  + **curl https://raw.githubusercontent.com/helm/helm/master/scripts/get -o get\_helm.sh**
* Set permissions that will make the script executable on the machine
  + **chmod 700 get\_helm.sh**
* Install Helm client locally
  + **./get\_helm.sh**
* Initial Helm and install in on the Kubernetes cluster
  + **helm init**
* Helm charts from a local package
  + Deploy the specified app for this challenge using the steps and yaml files provided. You will have to install the namespace, deployment and service yaml in that sequence.
  + Verify that the app has been deployed successfully by browsing the web app via the LoadBalancer IP address at the defined port number.
  + Redeploy the app to use v2 of the image and verify that the update is visible in the web app. Repeat these steps with v3 and v4 of the container images.
  + Convert these yaml files that were just used to deploy the app into a Helm chart using v1 of the container image.
  + Create a Helm package on the local machine for each version of the web app.
  + Remove the previously deployed app by deleting the namespace that was created via the yaml file
  + Deploy the helm chart with v1 of the image you just created.
  + Verify that the app has been deployed successfully
  + Make a note of the difference in number of steps involved in the deployment using individual yaml files vs the Helm chart
* Helm charts from a remote repo in Azure Container Registry
  + Push the Helm chart you just packaged to the remote ACR repo
  + Remove the package locally
  + Uninstall the app and redeploy it using the Helm chart from the ACR repo
  + Verify that the app has been deployed successfully

## Proctor Notes & Guidelines

* Proctor material needed for Helm charts from local packages.
* Proctor material needed for Helm charts from remote repositories.

# Challenge Set 10: Networking

## Lecture:

* Talk about Kubernetes Networking concepts. Highlight AKS networking capabilities like Basic vs Advanced Networking, Http Application Routing etc.
* At the end of the lecture mention that attendees will be leveraging the AKS Http Application Routing profile feature to access the web application externally using DNS name.

## Challenges:

* Make sure that HTTP Application Routing on the AKS cluster is enabled.
* Delete the existing content-web deployment and service.
* Copy the AKS cluster DNS host name from Azure Portal
* Deploy the content-web service and Ingress Controller using the HTTP Application Routing Add on feature.
  + The reference template can be found in the Files section in Teams: template-web-ingress-deploy.
  + Change the ACR & AKS DNS Name to match yours.
* Verify the DNS records are created, and if so, access the application using the DNS name, e.g http://fabmed.[YOUR\_AKS\_DNS\_ID].[REGION].aksapp.io

## Proctor Notes & Guidelines

* Make sure that attendees have a clear picture of services and different types of networking.
* The Ingress Controller has many capabilities, attendees are going to experiment only with its DNS routing capability in this challenge
* Make sure that each attendee’s AKS cluster is enabled with HTTP Application Routing.
* Refer to the AKS Doc for the verification of logs
* Validate DNS entries in the portal by navigating to the “special” resource group created for each AKS cluster and find the “DNS Zone” object in there.

# Challenge Set 11: Operations and Monitoring

## Lecture:

* Monitoring provides us with the ability to view what is happening within the cluster nodes, pods and containers.
* Benefits:
  + View application logs.
  + Identify resource bottlenecks.
  + View the controller's or pod's overall performance.
  + Review the resource utilization of workloads running on the host that are unrelated to the standard processes that support the pod.
  + Understand the behavior of the cluster under average and heaviest loads.
  + Identify capacity needs and determine the maximum load that the cluster can sustain.
* Talk about Kubernetes logging capabilities
  + kubectl logs – full logs for the container
* Talk about troubleshooting
  + kubectl describe pod – for finding failure reasons
  + kubectl exec – for getting a shell in a running container

## Challenges:

* Find the logs for your application’s containers
  + Using the Kubernetes Dashboard
  + Using kubectl
  + Notice how you can check the logs of any of your pods individually.
* Start a bash shell into one of the containers running on a pod and check the list of running processes
* Find out if your pods had any errors.
* Azure Monitor:
  + Enable "Azure Monitor for Containers" on the AKS cluster
  + Show a screenshot of CPU and memory utilization of all nodes
  + Show a screenshot displaying logs from the frontend and backend containers
* Kibana:
  + Install Fluentd and Kibana resources on the Kubernetes cluster to use an external ElasticSearch cluster
  + Create a Kibana Dashboard that shows a summary of logs from the front-end app only
  + Create a Kibana Dashboard that shows a summary of logs from the back-end app only
  + Create a Kibana Dashboard that gives a count of all log events from the kubernetes cluster for the last 30 minutes only.

## Proctor Notes & Guidelines

* You can check for the logs of a pod using:
  + **kubectl logs**
* Pod failures can be investigated during troubleshooting with:
  + **kubectl describe pod**
* A bash shell can be opened on any pod so you can poke around on the filesystem to debug issues. You can open the shell with:
  + **kubectl exec**
* Proctor material needed for Azure Monitor
* Proctor material needed for Kibana