Intermediate Exercises for Kubernetes

This document provides a set of exercises to implement some practical knowledge on what we’ve developed using the theory part of the session. You should be comfortable with kubectl commands before attempting this. Please review the cheat sheet from the basic section if needed.

# Exercise 1: Creating the namespaces

To create three namespaces called Dev, Test, and Prod using AKS, you can follow these steps:

1. Open the Azure Cloud Shell by clicking on the Cloud Shell icon in the Azure portal.
2. Select the PowerShell from the dropdown menu.
3. Run the following command to create the Dev namespace:

**kubectl create namespace dev**

1. Run the following command to create the Test namespace:

**kubectl create namespace test**

1. Run the following command to create the Prod namespace:

**kubectl create namespace prod**

After running these commands, you will have three namespaces called Dev, Test, and Prod created in your AKS cluster. You can verify the creation of these namespaces by running the following command:

**kubectl get namespaces**

This command will show you a list of all the namespaces in your cluster, including the ones you just created.

# Exercise 2: Use a ConfigMap to specify resources

1. When Kubernetes is configured, it already deploys several configmaps. You can see this by using the command:  
   **kubectl get cm**
2. To get more information on configmaps, use the command:  
   **kubectl explain cm**
3. From the Github, upload the two yaml files nginx-configmap.yaml and nginx-pod.yaml.
4. Apply the configmap using the command:  
   **kubectl apply -f nginx-configmap.yaml -n dev**
5. Apply the pod using the command:  
   **kubectl apply -f nginx-pod.yaml -n dev**
6. View the environment variables of the container using the following command:  
   **kubectl exec -n dev nginx-pod –- printenv**
7. Amend the configmap file, save it, and then apply it again using the command:  
   **kubectl apply -f nginx-configmap.yaml - test**
8. Apply the pod to a different namespace, this time to test using:  
   **kubectl apply -f nginx-pod.yaml -n test**
9. Verify that the file has been updated using the command:  
   **kubectl exec -n test nginx-pod – printenv**
10. Repeat steps 7-9 for the namespace prod (optional).
11. Delete all 3 namespaces with the command:  
    **kubectl delete namespace dev  
    kubectl delete namespace test  
    kubectl delete namespace prod**

# Exercise 3: Apply a Deployment using Kustomize

1. Download the files from the Kustomize folder in Github. Create a ate a directory structure to hold the Kubernetes deployment files:  
   First change directory to clouddrive using **cd clouddrive**

**mkdir myapp/base  
mkdir myapp/envrionments/staging  
mkdir myapp/environments/production**

1. Use the github and put the relevant files in their corresponding folders.
2. Use the command:  
   **kubectl create namespace staging**to create the staging namespace and then use:

**kubectl apply -k environments/staging**to apply deployment to the staging namespace

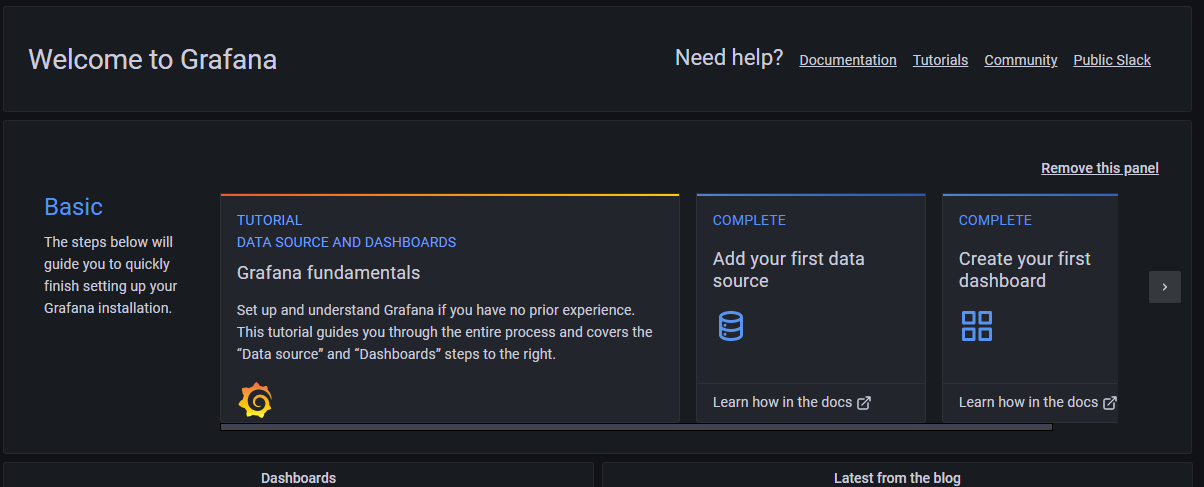
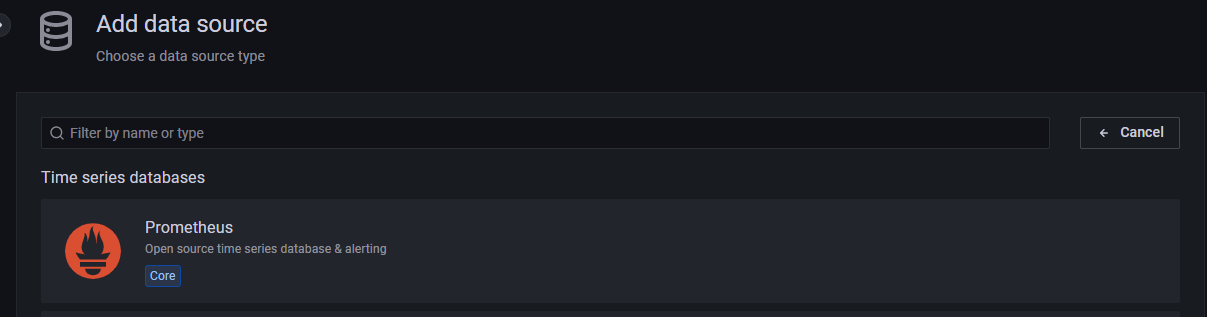
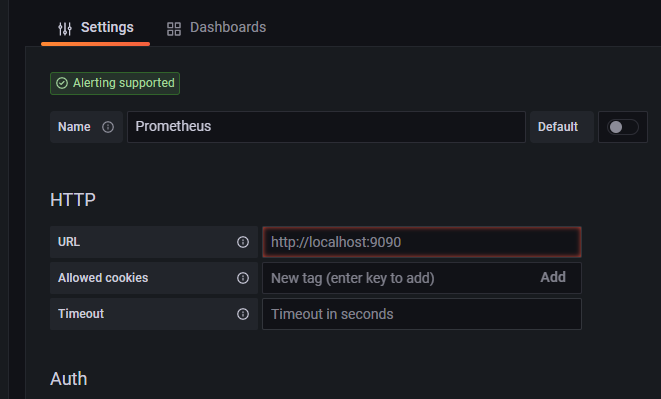
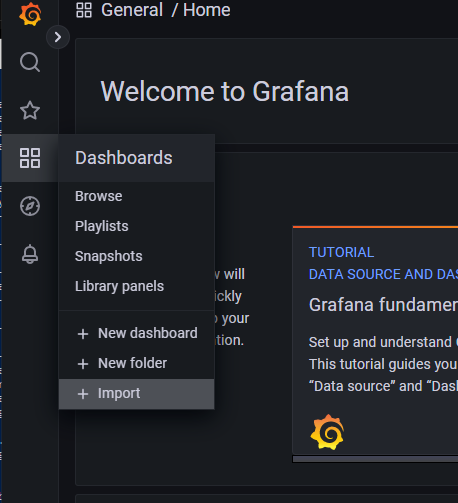
1. Verify the deployment was successful with the command:  
   **kubectl get pods -n staging**
2. Now view the deployment file under production, notice how we are using 5 replicas in production. Head to the kustomization file in production and notice how we are generating a namespace for production and that our deployment will merge.
3. Use the command:  
   **kubectl apply -k environments/production**to allow the kustomize file to produce the production namespace and apply the deployment.

# Using Kustomize with Secrets

1. Download the files from the Kustomize folder in Github. Create a ate a directory structure to hold the Kubernetes deployment files as shown in the video:  
   First change directory to clouddrive using **cd clouddrive**
2. Notice that in the base folder, the deployment has an env section denoting a secret.
3. The base kustomization file is very similar to the first example we looked at.
4. Head to the production folder and look at the kustomize file to see it reference the secret.
5. Apply the production kustomize file with the command:  
   **kubectl apply -k production**
6. Use the command:  
   **kubectl get pods**to verify it was successful and note the pod name.
7. To get the username env variable and region, use the command:   
   **kubectl exec nginx-deployment-<id> -- env**

# Using Helm & Prometheus Monitoring

To implement a monitoring solution in a cluster, we will see how that is possible using Helm. The Helm Chart is managed and maintained by the Helm community. We saw in the theory the benefits of using Helm and we will use this to initialise the setup.

1. Open two terminals. In the first, use the command:  
   **minikube start**  
   Make sure Docker is open and running (this was installed as a pre-req for the basic training).  
   Once your terminal is running, we will first check the version of helm with the command:  
   **helm version**If you need to install Helm, you can do so from this link:  
   <https://helm.sh/docs/intro/install/>
2. Use the command below to add the repo:  
   **helm repo add prometheus-community** [**https://prometheus-community.github.io/helm-charts**](https://prometheus-community.github.io/helm-charts)
3. Use the command:  
   **helm repo update**to update any repositories.
4. To complete the installation, use the command:  
   **helm install prometheus prometheus-community/prometheus**
5. Once complete, we can see the advantages to using helm by using the command:  
   **kubectl get all**
6. Let’s take a look at our services using the command  
   **kubectl get service**You can see that they are all using Cluster-IP and that there is no way of accessing these externally.
7. To expose the Prometheus-server, use the command:  
   **kubectl expose service prometheus-server --type=NodePort --target-port=9090 --name=prometheus-server-ext**  
   Verify that it is there by using:  
   **kubectl get service**
8. Access the Prometheus server in browser using the command:  
   **minikube service prometheus-server-ext**
9. Access the dashboard and explore the various tabs. Head to the graph tab and set the search bar to node\_memory\_Active\_bytes
10. Now that we have configured Prometheus and have set it up, we can enhance our monitoring capabilities by linking it with Grafana. In the second terminal use the following command:  
    **helm repo add grafana** [**https://grafana.github.io/helm-charts**](https://grafana.github.io/helm-charts)
11. Ensure the repo is up to date by using:  
    **helm repo updates**
12. Install Grafana with the command:  
    **helm install grafana grafana/grafana**
13. Notice how a secret was configured in the terminal. We know secrets are encrypted and to access them, we will need to decode them. Use the command:  
    **kubectl get secret --namespace default grafana -o jsonpath="{.data.admin-password}" | ForEach-Object { [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($\_)) }**
14. Expose the Grafana service in the same way we did the Prometheus service using the command:  
    **kubectl expose service grafana --type=NodePort --target-port=3000 --name=grafana-ext**
15. Launch the Grafana in browser using the command:  
    **minikube service grafana-ext**
16. Input the user name as ‘admin’ and copy the decoded password from step 13.
17. Head to the “Add your first data source”  
      
    and select Prometheus  
      
    Input the Prometheus url including localhost.  
      
    Scroll to the bottom and save.
18. Return to the home screen and locate the Dashboards icon, select import:  
      
    Enter ‘6417’ in the import via grafa.com field and click load:  
    