# Practice M8: Exam Prep

During this practice we will assume that we are working in Linux environment. It could be a physical machine or a virtual one. The distribution of choice is not that important, but it will be better to stick to some of the well supported distributions.

Most of the steps can be executed in Windows and/or macOS environment as well, either directly or in a VM.

The infrastructure can be built on-premise or in the cloud (AWS, GCP, Azure). Any combination of tools is acceptable.

## Assignment

Prepare a clustered environment to host a containerized web application. Application’s images must be built in an automated way – check every 3 minutes and if there are changes, build the images, publish them, and then re-deploy them. The cluster should be monitored – a dashboard, showing the utilization must be created.

## Possible Solution

We will implement one possible solution. It will include the following set of technologies:

* it will be hosted in AWS
* the infrastructure and cluster will be built with KOPS
* the cluster will be based on Kubernetes and will have one master and two worker nodes
* for monitoring we will use Elastic Stack
* logs and metrics will be collected with Filebeat and Metricbeat
* Jenkins and Kaniko will be used for image building

#### Infrastructure

As we are going to use AWS, we must take care for few preparation steps:

* Go to the AWS console (<http://console.aws.amazon.com/>) and create a user that will have only programmatic access with the following permissions:
  + **AmazonEC2FullAccess**
  + **AmazonRoute53FullAccess**
  + **AmazonS3FullAccess**
  + **IAMFullAccess**
  + **AmazonVPCFullAccess**
* Store the **Access Key** and **Secret Access Key** somewhere safe
* Check and change the AWS configuration if needed with **aws**

#### Cluster

Before continuing further, you must prepare a domain, Route 53 hosted zone, and S3 bucket. Please refer to Part 3 of Practice M3.

Once, the prerequisites are met, we can continue:

* Define the cluster:

**kops create cluster --name=kops.devopslab.tk --state=s3://dof-kops-20190130 --zones=eu-central-1a --node-count=2 --node-size=t2.medium --master-size=t2.micro --dns-zone=kops.devopslab.tk**

* Or we can create Terraform file(s):

**kops create cluster --name=kops.devopslab.tk --state=s3://dof-kops-20190130 --zones=eu-central-1a --node-count=2 --node-size=t2.medium --master-size=t2.micro --dns-zone=kops.devopslab.tk --out=. --target=terraform**

* We will continue without Terraform. Apply the state:

**kops update cluster kops.devopslab.tk --state=s3://dof-kops-20190130 --yes**

* Monitor the status of the cluster being created with either:

**kops validate cluster --state=s3://dof-kops-20190130**

**kubectl get pods --all-namespaces**

* Wait until everything is up and running, then you can continue with next tasks

#### Monitoring

Now that we have a working infrastructure, it is time to take care of the monitoring part. For this task we will deploy an **Elastic Stack**.

First, we will prepare our custom images for each component:

* Go to folder **M8/M8-1/docker-images-elk**
* Enter in every sub-folder and examine the files
* Build and publish the images with:

**docker image build -t {IMAGE} .**

**docker push {IMAGE}**

* Where **{IMAGE}** is the name under which you would like to publish the corresponding component

Next, we must adjust the **Elastic** deployment files:

* Go to folder **M8/M8-1/elastic-stack**
* Examine and adjust if needed both **\*deployment.yml** and **\*service.yml** files for every component
* Adjust the **logstash-config.yml** file if needed
* Go one folder up and execute:

**kubectl create -f elastic-stack/ -R --namespace=default**

* Check the name of the **Kibana** pod and follow its logs to know when it is ready:

**kubectl logs -f kibana-xxxxxxxx-yyyyy**

It is time to deploy the **Beat** pods:

* Go to folder **M8/M8-1/filebeat**
* Examine and adjust the files if needed
* Being in the **M8/M8-1** folder, execute:

**kubectl create -f filebeat/ -R --namespace=default**

And then we can deploy the **Metricbeat** pods:

* Go to folder **M8/M8-1/metricbeat**
* Examine and adjust the files if needed
* Being in the **M8/M8-1** folder, execute:

**kubectl create -f metricbeat/ -R --namespace=default**

Now, we can go to **Kibana**, create the **Index Pattern**, then explore the data, and finally, create one or more visualizations, and put them in a dashboard.

In this part there is plenty room for improvement. For example, we can add/enable additional modules of **Metricbeat**, and/or change the **grok filter** in the **Logstash** configuration.

Some **grok patterns** and resources can be found here:

* <https://qbox.io/blog/logstash-grok-filter-tutorial-patterns>
* <https://logz.io/blog/logstash-grok/>
* <https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns>

You can test your patterns on applications like those:

* <https://logz.io/blog/logstash-grok/>
* <http://grokconstructor.appspot.com/do/match>

And a Logstash plugin for Kubernetes: <https://github.com/checkr/logstash-filter-k8s>

#### Preparation and Exploration (Kompose)

Let’s assume that our starting point is a simple docker compose file found in folder **M8/M8-1/docker-compose**. It is using a local image, but we can easily change it to one available in **Docker Hub**. There is a tool that can help us create **YAML** files for use with **Kubernetes** or for direct deployment on **Kubernetes** – it is called **Kompose**. Let’s install it (there are many was to accomplish this):

* Download the file:

**curl -L https://github.com/kubernetes/kompose/releases/download/v1.17.0/kompose-linux-amd64 -o kompose**

* Set its execution bit:

**chmod +x kompose**

* Move it to a folder of your choice, for example:

**sudo mv ./kompose /usr/local/bin/kompose**

Now, let’s continue with the experiment:

* Go to folder **M8/M8-1/docker-compose** if not there
* Examine the **docker-compose.yml** file
* Now try to convert it with:

**kompose convert**

* Examine the resulting files. As you can see, they won’t work against our **Kubernetes** cluster, as they do not refer available image
* Now, examine the **docker-compose-alt.yml** file. It refers to an image
* We can convert it with:

**kompose convert -f docker-compose-alt.yml**

* Or deploy it to our cluster:

**kompose up -f docker-compose-alt.yml**

* Examine the status with the proposed command:

**kubectl get deployment,svc,pods,pvc**

* Clean up a bit:

**kubectl delete service web-page**

**kubectl delete deployment web-page**

#### Preparation and Exploration (Kaniko)

**Kaniko** is a tool that offers us a way to build **Docker** images in **Kubernetes**. Let’s play with it:

* Go to folder **M8/M8-1/kaniko**
* Examine and adjust the **pod-kaniko.yml** file if needed
* In order to be able to publish to a registry, we must prepare the credentials part, first. Login to Docker Hub:

**docker login -u $USERNAME -p $PASSWORD**

* Check if the **config.json** file is created:

**ls -al ~/.docker/config.json**

* Create a config map:

**kubectl create configmap docker-config --from-file=$HOME/.docker/config.json**

* Create the **Kaniko** pod:

**kubectl apply -f pod-kaniko.yml**

* Ask for the pods few times:

**kubectl get pods**

* Check the logs of the **Kaniko** pod:

**kubectl logs kaniko**

* Now, create a deployment using the newly created and published image:

**kubectl create -f deployment-with-service.yml**

* Check that everything is working
* Clean up:

**kubectl delete -f deployment-with-service.yml**

#### CD/CI Preparation

There are many CD/CI tools and multiple ways to deploy and use them. For this particular exercise, we will deploy **Jenkins** in our **Kubernetes** cluster. And instead of doing it manually, we will use **Helm** package manager:

* Create a separate namespace:

**kubectl create namespace jenkins**

* Create a service account and binding:

**kubectl create serviceaccount --namespace kube-system tiller**

**kubectl create clusterrolebinding tiller-cluster-admin --clusterrole=cluster-admin --serviceaccount=kube-system:tiller**

* Initialize **Helm**:

**helm init --service-account tiller**

* Install **Jenkins**:

**helm install --name jenkins --namespace jenkins stable/jenkins**

* Get information for all pods:

**kubectl get pods --all-namespaces**

* Now, we should add one more binding:

**kubectl create clusterrolebinding jenkins --clusterrole=cluster-admin --serviceaccount=jenkins:default**

* Ask for the status of **Jenkins**:

**helm status jenkins**

* Get the password for the admin user:

**printf $(kubectl get secret --namespace jenkins jenkins -o jsonpath="{.data.jenkins-admin-password}" | base64 --decode);echo**

* Get information about the service:

**kubectl get svc --namespace jenkins jenkins**

* Get the public DNS of the master node
* Open a new tab and navigate to the provided URL plus the port that you saw on the previous step
* If nothing opens, then go to the AWS console and adjust the security group’s rules
* Once in, resolve the issue with the plugins by updating them and restarting the Jenkins instance
* Next, we must configure **Jenkins**. Go to **Manage Jenkins** and then **Configure System**
* Enter the following:
  + For **Jenkins URL** set the **Cluster IP** from above - <http://100.71.161.176:8080> (change it to yours)
  + In **Cloud > Kubernetes > Kubernetes URL** set **ec2-3-120-183-142.eu-central-1.compute.amazonaws.com** (substitute it with your master DNS name or IP address)
  + Select **Cloud > Kubernetes > Disable https certificate check**
  + Change the **Cloud > Kubernetes > Kubernetes Namespace** to the **Jenkins** namespace
  + In **Cloud > Images > Kubernetes Pod Template > Namespace** set to the value in the previous point
* Click on **Save**

#### CD/CI with Jenkins and Docker

Let’s try to automate the image building process:

* Create additional configuration with **Docker Hub** (or another registry) credentials in the **jenkins** namespace

**kubectl create configmap docker-config --from-file=$HOME/.docker/config.json -n jenkins**

* Return to **Jenkins** and create new project:
  + Go to **New Item**
  + Select **Pipeline**
  + Enter a name, for example **Pipeline-Docker**
  + Click **OK**
  + In **Pipeline > Script** section enter the following (or copy it from the **M8/M8-1/jenkins/job-1-pipeline-docker.txt** file):

**def label = "docker-${UUID.randomUUID().toString()}"**

**podTemplate(label: label, yaml: """**

**apiVersion: v1**

**kind: Pod**

**spec:**

**containers:**

**- name: docker**

**image: docker:1.11**

**command: ['cat']**

**tty: true**

**volumeMounts:**

**- name: dockersock**

**mountPath: /var/run/docker.sock**

**- name: docker-config**

**mountPath: /root/.docker**

**volumes:**

**- name: dockersock**

**hostPath:**

**path: /var/run/docker.sock**

**- name: docker-config**

**configMap:**

**name: docker-config**

**"""**

**) {**

**def image = "shekeriev/k8s-jenkins:latest"**

**node(label)**

**{**

**stage('Build Docker image')**

**{**

**git 'https://github.com/shekeriev/simple-docker-image.git'**

**container('docker')**

**{**

**sh "docker build -t ${image} ."**

**}**

**}**

**stage ("Push")**

**{**

**container('docker')**

**{**

**sh "docker push ${image}"**

**}**

**}**

**}**

**}**

* + Click **Save**
* Click on **Build Now**
* Meanwhile you can monitor the log of the pod:

**kubectl logs -n jenkins jenkins-xxxxxxxx-yyyyyy**

Okay, so far, we used a **Docker** container to build our image from a **Git** project and publish the image to **Docker Hub**. There are also other options. Let’s examine how we can use **Kaniko** with **Jenkins**:

#### CD/CI with Jenkins and Kaniko

We met **Kaniko** earlier, let’s see if and how we can utilize it:

* Create additional configuration with **Docker Hub** (or another registry) credentials in the **jenkins** namespace or reuse the one from the previous task (then you should adjust the job’s definition):

**kubectl create configmap kaniko-docker-config --from-file=$HOME/.docker/config.json -n jenkins**

* Return to **Jenkins** and create new project:
  + Go to **New Item**
  + Select **Pipeline**
  + Enter a name, for example **Pipeline-Kaniko**
  + Click **OK**
  + In **Pipeline > Script** section enter the following (or copy it from the **M8/M8-1/jenkins/job-2-pipeline-kaniko.txt** file):

**def label = "kaniko-${UUID.randomUUID().toString()}"**

**podTemplate(name: 'kaniko', label: label, yaml: """**

**kind: Pod**

**metadata:**

**name: kaniko**

**spec:**

**containers:**

**- name: kaniko**

**image: gcr.io/kaniko-project/executor:debug**

**imagePullPolicy: Always**

**command:**

**- /busybox/cat**

**tty: true**

**volumeMounts:**

**- name: kaniko-docker-config**

**mountPath: /root/.docker**

**restartPolicy: Never**

**volumes:**

**- name: kaniko-docker-config**

**configMap:**

**name: kaniko-docker-config**

**"""**

**) {**

**node(label) {**

**stage('Build with Kaniko') {**

**git 'https://github.com/shekeriev/simple-docker-image.git'**

**container(name: 'kaniko', shell: '/busybox/sh') {**

**withEnv(['PATH+EXTRA=/busybox:/kaniko']) {**

**sh '''#!/busybox/sh**

**/kaniko/executor -f `pwd`/Dockerfile -c `pwd` --destination=shekeriev/k8s-jenkins:latest --force**

**'''**

**}**

**}**

**}**

**}**

**}**

* + Click **Save**
* Click on **Build Now**
* Meanwhile you can monitor the log of the pod:

**kubectl logs -n jenkins jenkins-xxxxxxxx-yyyyyy**

#### CD/CI with Jenkins and Kaniko for multi-image solution

Most probably we will face a situation where we must build more than two images at the same time. Here is one way to do it:

* Create additional configuration with **Docker Hub** (or another registry) credentials in the **jenkins** namespace or reuse the one from the previous task (then you should adjust the job’s definition):

**kubectl create configmap kaniko-docker-config --from-file=$HOME/.docker/config.json -n jenkins**

* Return to **Jenkins** and create new project:
  + Go to **New Item**
  + Select **Pipeline**
  + Enter a name, for example **Pipeline-Kaniko-Multi**
  + Click **OK**
  + In **Pipeline > Script** section enter the following (or copy it from the **M8/M8-1/jenkins/job-3-pipeline-kaniko-multi.txt** file):

**def label = "kaniko-${UUID.randomUUID().toString()}"**

**podTemplate(name: 'kaniko', label: label, yaml: """**

**kind: Pod**

**metadata:**

**name: kaniko**

**spec:**

**containers:**

**- name: kaniko**

**image: gcr.io/kaniko-project/executor:debug**

**imagePullPolicy: Always**

**command:**

**- /busybox/cat**

**tty: true**

**volumeMounts:**

**- name: kaniko-docker-config**

**mountPath: /root/.docker**

**restartPolicy: Never**

**volumes:**

**- name: kaniko-docker-config**

**configMap:**

**name: kaniko-docker-config**

**"""**

**) {**

**node(label) {**

**stage('Build with Kaniko') {**

**git 'https://github.com/shekeriev/simple-docker-image.git'**

**container(name: 'kaniko', shell: '/busybox/sh') {**

**withEnv(['PATH+EXTRA=/busybox:/kaniko']) {**

**sh '''#!/busybox/sh**

**/kaniko/executor -f `pwd`/Dockerfile -c `pwd` --destination=shekeriev/k8s-jenkins-multi:1 --force --cleanup**

**'''**

**}**

**}**

**container(name: 'kaniko', shell: '/busybox/sh') {**

**withEnv(['PATH+EXTRA=/busybox:/kaniko']) {**

**sh '''#!/busybox/sh**

**/kaniko/executor -f `pwd`/Dockerfile -c `pwd` --destination=shekeriev/k8s-jenkins-multi:2 --force --cleanup**

**'''**

**}**

**}**

**}**

**}**

**}**

* + Click **Save**
* Click on **Build Now**
* Meanwhile you can monitor the log of the pod:

**kubectl logs -n jenkins jenkins-xxxxxxxx-yyyyyy**

#### CD/CI – build and apply

We can go even further, by applying the image changes to the deployed solution. For this, we will extend the example in the **CD/CI with Jenkins and Docker** section:

* Go to **New Item**
* In the **Copy from** field enter the name of the job from the **Docker** section
* In the **Enter an item name** set some name, for example **Pipeline-Docker-Apply**
* Change (new or changed sections are in **Bold**) the pipeline script to (or copy it from the **M8/M8-1/jenkins/job-4-pipeline-docker-apply.txt** file):

def label = "docker-${UUID.randomUUID().toString()}"

podTemplate(label: label, yaml: """

apiVersion: v1

kind: Pod

spec:

containers:

- name: docker

image: docker:1.11

command: ['cat']

tty: true

volumeMounts:

- name: dockersock

mountPath: /var/run/docker.sock

- name: docker-config

mountPath: /root/.docker

**- name: kubectl**

**image: lachlanevenson/k8s-kubectl:v1.8.0**

**command: [cat]**

**tty: true**

**volumeMounts:**

**- name: dockersock**

**mountPath: /var/run/docker.sock**

**- name: docker-config**

**mountPath: /root/.docker**

volumes:

- name: dockersock

hostPath:

path: /var/run/docker.sock

- name: docker-config

configMap:

name: docker-config

"""

) {

**def imagetag = new Date().format('yyyyMMdd.HHmmss')**

**def image = "shekeriev/k8s-jenkins:${imagetag}"**

node(label)

{

stage('Build Docker image')

{

git 'https://github.com/shekeriev/simple-docker-image.git'

container('docker')

{

sh "docker build -t ${image} ."

}

}

stage ("Push")

{

container('docker')

{

sh "docker push ${image}"

}

}

**stage ("Apply the changes with kubectl")**

**{**

**container('kubectl')**

**{**

**sh "sed 's/%IMAGE-PLACEHOLDER%/${imagetag}/g' -i yaml/application.yml"**

**sh "kubectl apply -f yaml/application.yml"**

**}**

**}**

}

}

* Click **Save**
* Click on **Build Now**
* Check the result either on the command line and/or in the browser

#### Clean up

Don’t forget to clean up, because your bill at the end of the month could cause you a headache 😉

There is just one step required:

**kops delete cluster kops.devopslab.tk --yes --state=s3://dof-kops-20190130**