# Practice M3: Kubernetes

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.

All steps can be executed in Windows and/or macOS environment as well.

## Part 1: Additional Objects

Let’s do few experiments with a local Kubernetes cluster, for example a Minikube installation.

#### Namespaces (folder M3/M3-1/1-ns)

We can divide the cluster on logically separated clusters by defining namespaces:

* We can do it declaratively:

**kubectl create -f demo-namespace.yml**

* Now list the available namespaces:

**kubectl get namespaces**

* Delete the namespace. And create it in an imperative manner:

**kubectl delete namespaces demo-namespace**

* Or with:

**kubectl delete -f demo-namespace.yml**

* Now let’s create it again:

**kubectl crate namespace demo-namespace**

* Let’s list the namespaces again:

**kubectl get ns**

* And create a pod in it:

**kubectl create -f demo-pod.yml -n demo-namespace**

* Now if we just ask for the pods with:

**kubectl get po**

* We can be surprised. Instead we can execute:

**kubectl get po -n demo-namespace**

* Clean up the pod and the namespace. We can do it one object at time, or at once with:

**kubectl delete ns demo-namespace**

#### Daemon Sets (folder M3/M3-1/2-daemonsets)

Daemon Sets are like the Replication Controllers and Replica Sets. There is one important difference though – their workload goes to every node or only to specific nodes and with only one copy, so no multiple replicas spread across the cluster

* Create the daemon set:

**kubectl create -f daemon-sets.yml**

* Check what has been created:

**kubectl get ds**

* Get list of the running pods:

**kubectl get pods**

* Set a label of the node:

**kubectl label node minikube disk=samsung**

* Now get the list of running pods:

**kubectl get pods**

* And information about the daemon set:

**kubectl get ds**

* Let’s change the label and see what happens:

**kubectl label node minikube disk=wdc --overwrite**

**kubectl get pods**

**kubectl get ds**

* Clean up:

**kubectl delete ds daemon-set**

#### Jobs (folder M3/M3-1/3-jobs)

There are situations in which we need to run tasks that start, do something, and end. This is covered by a special object type – Jobs.

* Start the job:

**kubectl create -f batch-job.yml**

* Get information about the job:

**kubectl get jobs**

* Check the pods:

**kubectl get pods**

* Get detailed information about the job:

**kubectl describe jobs.batch batch-job**

* Check the history of the pods:

**kubectl get pods -a**

* Get again the info about the jobs
* Delete the job. This will delete the pod as well:

**kubectl delete job batch-job**

We can run a job multiple time:

* Examine the file **batch-job-serial.yml**
* Execute it

**kubectl create -f batch-job-serial.yml**

* Check the results

**kubectl get jobs**

**kubectl describe jobs batch-job-serial**

**kubectl get po**

* Delete the job

**kubectl delete job batch-job-serial**

And of course, run it in parallel:

* Examine the file **batch-job-parallel.yml**
* Execute it

**kubectl create -f batch-job-parallel.yml**

* Check the results

**kubectl get jobs**

**kubectl describe jobs batch-job-parallel**

**kubectl get pods**

* Once the job is complete, we can delete it and all related objects with the usual command:

**kubectl delete job batch-job-parallel**

#### Cron Jobs (folder M3/M3-1/4-cron)

There may a need to execute job on a schedule:

* Examine the file **batch-job-cron.yml**
* Start the job

**kubectl create -f batch-job-cron.yml**

* Examine what happens

**kubectl get cronjobs**

* Check if there are any new pods created:

**kubectl get pods**

* Repeat the check in let’s say one minute or so
* Delete the cron job. It will delete the pods as well:

**kubectl delete cronjobs batch-job-cron**

#### Volumes (folder M3/M3-1/5-volumes)

Let’s examine few use cases of volumes. In general we use them to share data between pods or to provide persistent storage to pods.

One option is to mount an empty directory and have for example one pod producing content, and another one consuming it:

* Examine the **volume-pod-emptydir.yml** file
* Create the resources in the file

**kubectl create -f volume-pod-emptydir.yml**

* Examine the resources created

**kubectl get po**

**kubectl describe pod volume-pod-emptydir**

**kubectl get svc volume-pod-emptydir-svc**

**kubectl get ep**

* Check the result in a browser
* Cleanup with

**kubectl delete -f volume-pod-emptydir.yml**

We can also mount git repository as a volume:

* Examine the **volume-pod-gitrepo.yml**
* Create the resources in the file

**kubectl create -f volume-pod-gitrepo.yml**

* Examine the resources created

**kubectl get po**

**kubectl describe pod volume-pod-gitrepo**

**kubectl get svc volume-pod-gitrepo-svc**

**kubectl get ep**

* Check the result in a browser
* Remove the resources

**kubectl delete -f volume-pod-gitrepo.yml**

There are also other types of volumes, but we will work with them in later modules.

#### Environment Variables and ConfigMaps (folder M3/M3-1/6-params)

We can give parameters to the pods, the same way as we were able to do it when working directly with Docker

* Examine the file **environment-variables.yml**
* Apply the file
* Examine the result
* Check that the environment variable is reflected
* Clean up

Even though environment variables do their job, there is a better way. It is through ConfigMaps. Many forms are supported, but here we will examine the simplest one – we will create a secret from a literal on the command line

* Execute the following command:

**kubectl create configmap demo-config --from-literal=var1=ConfigMap**

* List available config maps

**kubectl get configmap**

* Examine a stored config map

**kubectl get configmap demo-config -o yaml**

* Now examine the **configmap-demo.yml** file
* And create the deployment
* Check the result
* Clean up

#### Secrets (folder M3/M3-1/7-secrets)

Sensitive information can and should be stored in an appropriate way and separate from the rest of the code or configuration files. Thus, there are objects of type Secret. Here we will demonstrate very simple way of declaring and using them:

* Let’s create a secret on the command line. It will store a password:

**kubectl create secret generic demo-secret --from-literal=secret=MyPassword**

* We can examine it with:

**kubectl get secret demo-secret -o yaml**

* The important information is stored encoded in **Base64** encoding
* Now, let’s examine the file **secrets-demo.yml**
* And let’s create the deployment
* We are ready to check the result
* And finally, clean up the resources

## Part 2: Health and Resources Monitoring

#### Health Checks (folder M3/M3-2/1-health)

First let’s implement a liveness command check.

* Examine the file **liveness-command.yml**
* Now create the pod

**kubectl create -f liveness-command.yml**

* And get information about it within the first 30 seconds

**kubectl describe pod liveness-cmd**

* You can see that the liveness check has been assigned, and currently everything seems to be okay
* If you ask again for the information at least 35 seconds after the pod creation:

**kubectl describe pod liveness-cmd**

* Now we see that the check has failed and after another 30 seconds, if we ask for the pods:

**kubectl get pod liveness-cmd**

* We can see that the pod has been restarted
* Let’s clean up

**kubectl delete -f liveness-command.yml**

Now we can extend the pod’s definition to include a readiness probe as well, this way we can guarantee that if a pod is not ready, there won’t be any traffic directed to it.

* Examine the file **liveness-command-ext.yml**
* Now create the pod

**kubectl create -f liveness-command-ext.yml**

* And get information about it within the first 30 seconds

**kubectl describe pod liveness-cmd**

* You can see that the liveness and readiness checks have been assigned, and currently everything seems to be okay
* If you ask again for the information at least 35 seconds after the pod creation:

**kubectl describe pod liveness-cmd**

* Now we see that the readiness check has failed and after another 30-40 seconds, if we ask for the pods:

**kubectl get pod liveness-cmd**

* We can see that the pod has been restarted because of the liveness check failed
* Let’s clean up

**kubectl delete -f liveness-command-ext.yml**

#### Heapster (folder M3/M3-2/2-heapster)

There is a whole installation procedure, which can be found here: <https://github.com/kubernetes-retired/heapster>

It is marked for deprecation, but we will take a look anyways.

* When running on Minikube, we can just enable the Heapster plugin. First let’s see what plugins we have:

**minikube addons list**

* Now, we can enable the plugin with:

**minikube addons enable heapster**

* Wait for few minutes (it can take up to 5 min) for the services to start
* If want to check resource usage of nodes on the command line, we can use:

**kubectl top nodes**

* And for the pods:

**kubectl top pods**

* And the same, but for all namespaces:

**kubectl top pods --all-namespaces**

* There is also a web interface, based on Grafana. In order to check how to access it, we must execute:

**minikube service list**

* And pick up the URL for the row that has **monitoring-grafana** for a name or execute the following:

**minikube service monitoring-grafana -n kube-system**

* There you can ignore and close the welcome wizard and from the top left menu pick up a dashboard to look at
* If we want, we can disable the plugin with:

**minikube addons disable heapster**

As alternative to Heapster it is recommended to use Prometheus and/or Kubernetes Metrics Server. We will deploy Prometheus in next part of our module.

#### Autoscaling (folder M3/M3-2/3-auto)

We will test horizontal autoscaling. Before continuing further ensure that metrics server is up and running. If you are using Minikube, then you must enable the corresponding addon. For this purpose:

* Enable the addon:

**minikube addons enable metrics-server**

* Examine the file **auto-scale.yml**
* And create the resources:

**kubectl create -f auto-scale.yml**

* Check that all replicas are there and running:

**kubectl get pods**

* Now create auto scale rule:

**kubectl autoscale deployment auto-scale-deploy --cpu-percent=10 --min=1 --max=5**

* Ask for more information:

**kubectl get horizontalpodautoscalers.autoscaling auto-scale-deploy**

* Wait few minutes metrics to be collected and ask again:

**kubectl get horizontalpodautoscalers.autoscaling auto-scale-deploy -o yaml**

* After few minutes (at least 5) the system should scale down our deployment to one replica:

**kubectl get pods -o wide**

**kubectl get deployments**

* Now we can open second terminal in order to monitor the scaling process:

**watch -n 1 kubectl get hpa,deployment**

* In the first terminal, we are going to simulate workload to trigger scale up:

**kubectl run -it --rm --restart=Never load-generator --image=busybox -- sh -c "while true; do wget -O - -q http://auto-scale-svc.default; done"**

* If we switch to the other terminal, we can see the scale up process in action
* Return to the first terminal and press **Ctrl+C** to stop the load generator pod
* Now, we can delete all resources:

**kubectl delete hpa auto-scale-deploy**

**kubectl delete -f auto-scale.yml**

## Part 3: Helm, KOps, Terraform + AWS + Kubernetes

#### Helm (folder M3/M3-3/1-helm)

Install the Helm solution (the client part):

* Download the latest file from (<https://github.com/helm/helm/releases>):

**curl -LO https://storage.googleapis.com/kubernetes-helm/helm-v2.12.1-linux-amd64.tar.gz**

* Unpack it

**tar -zxvf helm-v2.12.1-linux-amd64.tar.gz**

* Now move it to a location of your choice, for example:

**sudo mv linux-amd64/helm /usr/local/bin/**

* Check that everything is working:

**helm**

* As usual, we can set up a command completion for the session:

**source <(helm completion bash)**

* Or for the user:

**echo "source <(helm** **completion bash)" >> ~/.bashrc**

Install the server part:

* We install the server part by executing:

**helm init**

* Now let’s check if everything went according to plan:

**kubectl get pods --namespace kube-system**

Now let’s create a simple chart:

* Execute:

**helm create demo**

* We can examine the files and folders that are part of this chart:

**tree demo**

**vi demo/Chart.yaml**

**vi demo/templates/deployment.yaml**

**vi demo/templates/service.yaml**

**vi demo/values.yaml**

* Okay, let’s execute the following to install the chart:

**helm install demo**

* Or in order to make it distinguishable:

**helm install demo --name demo-chart**

* Then follow the instructions in order to access the application
* Once ready, we can delete or uninstall the application by typing:

**helm delete demo-chart --purge**

* If we omitted the name, then we can list all installations made by helm:

**helm list**

We can also search for a premade charts and install them:

* To search for chars, execute:

**helm search**

**helm search prom**

* We can do the same, but on the Internet. We must open <https://hub.helm.sh/> in our browser and search for a chart
* Let’s install Prometheus, but we will aim for the complete solution, so we will choose Prometheus Operator, but before the actual installation we will create a separate namespace:

**kubectl create namespace monitoring**

**helm install --name=prom --namespace=monitoring stable/prometheus-operator**

* We can get information about the pods by executing:

**kubectl --namespace monitoring get pods**

* Now we can set port forwarding for all three services included in the package. For Grafana:

**kubectl port-forward --namespace=monitoring prom-grafana-8656585969-rjbqj 3000 &**

* For the Alert Manager:

**kubectl port-forward --namespace=monitoring alertmanager-prom-prometheus-operator-alertmanager-0 9093 &**

* And for the Prometheus Operator:

**kubectl port-forward --namespace=monitoring prometheus-prom-prometheus-operator-prometheus-0 9090 &**

* Of course the above names have to be adjusted to reflect the actual names in your case
* If we have a port already in use, or we did a forward by a mistake, in order to undone it, we must kill the corresponding **kubectl** process
* Now, we can open the Grafana web UI by directing our browser to: <http://localhost:3000>
* The user name is **admin** and the password is **prom-operator**
* We can experiment with different views and dashboards
* Once, ready, we can delete the application

**helm delete prom --purge**

* Additionally, we must remove few CRDs:

**kubectl delete crd prometheuses.monitoring.coreos.com**

**kubectl delete crd prometheusrules.monitoring.coreos.com**

**kubectl delete crd servicemonitors.monitoring.coreos.com**

**kubectl delete crd alertmanagers.monitoring.coreos.com**

* If we created a separate namespace, we can remove it as well:

**kubectl delete namespaces monitoring**

* And kill the **kubectl** process to release the ports

#### KOps (folder M3/M3-3/2-kops)

First, we must install kops:

* Download the binary

**curl -LO https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag\_name | cut -d '"' -f 4)/kops-linux-amd64**

* Make it executable:

**chmod +x kops-linux-amd64**

* Move it in an accessible location:

**sudo mv kops-linux-amd64 /usr/local/bin/kops**

* Now we can check if everything is okay by executing:

**kops**

* If desired we can set a bash completion for the current session:

**source <(kops completion bash)**

* Or permanently:

**echo "source <(kops completion bash)" >> ~/.bashrc**

Then we must install **aws** cli tool if not installed already.

Next, we must create a dedicated user with the appropriate permissions. We will create a kops user with administrative privileges which is not a good practice, but will do the job for the purpose of this lab

Then we must configure our local **aws** cli with **aws configure**

We will need a shared storage. For this purpose, we will create a **S3 Bucket**. It will be used to store the KOps’ state

Let’s go to the S3 management console and create one with unique name, for example **dof-kops-20181221**

Alternative options is to do it on the command line with:

**aws s3 mb s3://dof-kops-20181221**

Next step is to create a Route53 DNS hosted zone. Again, we can go with the web console or on the command line.

Before we continue further, we must ensure that we have a domain under our control. If we do not own one, for the purpose of the labs, we can use a free service like Dot TK - [www.dot.tk](http://www.dot.tk)

Now that we have a domain, we can continue. Let’s create the following sub-domain and DNS hosted zone:

**aws route53 create-hosted-zone --name kops.devopslab.tk --caller-reference 1**

Next, we must then set up our NS records in the parent domain

And then create the cluster configuration with:

**kops create cluster --name=kops.devopslab.tk --state=s3://dof-kops-20181221 --zones=eu-central-1a --node-count=2 --node-size=t2.micro --master-size=t2.micro --dns-zone=kops.devopslab.tk**

Now, if there are no errors, we can either edit the configuration with:

**kops edit cluster kops.devopslab.tk**

Or initiate the cluster creation with:

**kops update cluster kops.devopslab.tk --yes**

No matter which option we will choose, we must specify the state as well, because it is stored in a S3 bucket:

**kops edit cluster kops.devopslab.tk --state=s3://dof-kops-20181221**

Instead of specifying each time the state resource, we can export it as an environment variable with:

**export KOPS\_STATE\_STORE=s3://dof-kops-20181221**

We can check if all the nodes are up:

**kubectl get nodes**

Now we can create the deployment and the service we used earlier:

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

Examine what has been created:

**kubectl get deployments -o wide**

**kubectl get service -o wide**

**kubectl get pods -o wide**

Now we can delete everything with:

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

Once we are ready, we can delete the cluster with:

**kops delete cluster kops.devopslab.tk --yes**

#### Terraform and Kubernetes (in Minikube) (folder M3/M3-3/3-minikube)

We can prepare one simple **main.tf** file in order to do a very simple experiment combining Terraform and Kubernetes in our case Minikube implementation. We can use the provided file:

**cp main.tf.demo1 main.tf**

As usual we must initialize the provider first with:

**terraform init**

Then we can check the plan:

**terraform plan**

And finally, we can apply the plan

**terraform apply**

We can examine what has been created:

**kubectl get service -o wide**

**kubectl get pods -o wide**

In order to be able to access our web application we can execute:

**minikube service list**

And then follow the URL of our service

We can modify the file and instead of pod declaration, we can put replication controller block. It would be faster if we just rename the files:

**cp main.tf.demo2 main.tf**

Then we can run:

**terraform plan**

**terraform apply**

We can check the result and examine what resources have been created with regular commands

Finally, we can clean with:

**terraform destroy**

#### Terraform and AWS (folder M3/M3-3/4-aws)

We will use AWS EKS module for Terraform. More information for it can be found here: <https://registry.terraform.io/modules/terraform-aws-modules/eks/aws/2.0.0>

We should keep in mind that this setup can (and most likely will) lead to some expences.

* Examine the set of files main.tf, outputs.tf, and variables.tf
* Then initialize the environment

**terraform init**

* And ask for the plan, and then apply

**terraform plan**

**terraform apply**

* During the creation process, which can take up to 10 minutes, we can go to the web console and watch the progress. We may visit VPC, EC2, and EKS sections.
* Once ready, write down the **cluster\_endpoint** value
* Create file **auth-eks.yml** by copying everything after **config\_map\_aws\_auth** until **kubectl\_config**
* Create file **config-eks.yml** by copying everything after **kubectl\_config** until **region**
* Or you can rename the two files created in your working folder. You can refer to the two sample files included
* Now install **aws-iam-authenticator**:

**curl -o aws-iam-authenticator** [**https://amazon-eks.s3-us-west-2.amazonaws.com/1.11.5/2018-12-06/bin/linux/amd64/aws-iam-authenticator**](https://amazon-eks.s3-us-west-2.amazonaws.com/1.11.5/2018-12-06/bin/linux/amd64/aws-iam-authenticator)

**chmod +x aws-iam-authenticator**

**sudo mv aws-iam-authenticator /usr/local/bin/**

* Now check that the tool is working

**aws-iam-authenticator help**

* Next, export a variable to configure **kubectl**

**export KUBECONFIG=$KUBECONFIG:~/.kube/config-eks.yml**

* Correct the path to match yours or move the **config-eks.yml** file to your **~/.kube** folder
* Now check that **kubectl** is able to communicate with your cluster:

**kubectl cluster-info**

* Now that we can communicate with our master, let’s enable worker nodes:

**kubectl apply -f auth-eks.yml**

* Watch the progress:

**kubectl get nodes --watch**

* Now we can give them a work to do:

**kubectl create -f terraform-aws.yml**

* Check the progress:

**kubectl get pods -o wide**

* Now we can ask for the service:

**kubectl get service**

* Copy the value under External IP and paste it in a browser to check if you can access the application
* Once done with exploration, we can destroy the whole infrastructure:

**kubectl delete -f terraform-aws.yml**

**terraform destroy**

* After several minutes (more than 5) all resources will be dismissed