Lab No. 3 Container Orchestration

The goal of this lab is to provide a *gentle* introduction to Kubernetes. In this lab you will learn to:

1. provision a kubernetes cluster in google cloud container engine,
2. use kubectl to interact with basic kubernetes resources (pods, deployments, replicasets and services)

Part 1: Set Up a Kubernetes Cluster¶

1. Log into the Google Cloud Platform, go to **APIs & Services** and make sure that **Google Kubernetes Engine API** is enabled.

2. Create a virtual machine named **lab03** based on the `ubuntu-1804-bionic-v20180823` image. We are going to use this machine to provision a Kubernetes cluster and interact with it. Make sure you enable an external IP address in order to be able to SSH into this machine from anywhere. A disk of 10GB will suffice. You can assign as little memory as allowed since we are just going to execute commands from this machine, we are not going to run any services on it.

3. SSH into **lab03**

4. Install docker and add the current user to the `docker` user group (using the same procedure that was used in Part 1 of Lab No. 2 (../lab02/lab02.html#lab02-part1). Do not forget to logout and log back in, in order for group membership changes to be reflected.

5. Install the **kubectl** utility

   ```
   > sudo snap install kubectl --classic
   ```

6. Run the following command to test the `kubectl` installation.

   ```
   > kubectl version
   ```

   The error message that reports that the command was not able to connect to `localhost:8080` is due to the fact that we have not configured a Kubernetes cluster.

7. We are going to create a Kubernetes cluster using the Google Container Engine. The following command will take care of that (be patient, it takes a few minutes to complete).

   ```
   > gcloud container clusters create lab03cluster --zone us-west1-c
   ```

8. Use the command `gcloud container clusters list` to verify the cluster configuration.

9. Use the `gcloud compute instances list` and notice how the previous command created three new virtual machines.
10. Run the `kubectl version` command again. Notice how the `kubectl` tool was automatically configured to connect to the Kubernetes cluster that was just created.

Part 2: Using `kubectl`

1. Inspect the options available for the `kubectl` command.

```
> kubectl | less
```

2. One of most basic tasks is to check the nodes of a cluster:

```
> kubectl get nodes
NAME                                          STATUS    ROLES     AGE       VEF
gke-lab03cluster-default-pool-973a4e96-1k2t   Ready     <none>    2m        v1.
gke-lab03cluster-default-pool-973a4e96-qvvl   Ready     <none>    2m        v1.
gke-lab03cluster-default-pool-973a4e96-rlv0   Ready     <none>    2m        v1.
```

3. **Pods** are "the smallest deployable units of computing that can be created and managed in Kubernetes" (take a few minutes to review Pods documentation located at https://kubernetes.io/docs/concepts/workloads/pods/pod/ (https://kubernetes.io/docs/concepts/workloads/pods/pod/)). **Pods** are managed with **Deployments** (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)) Create a Deployment of the *nginx* web server:

```
> kubectl run foobar --image=nginx
```

4. In a Kubernetes cluster, containers run as part of a Pod. In the previous command, Kubernetes created a deployment that executes a single container of the nginx webserver. Verify that a deployment was created:

```
> kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
foobar    1         1         1            1           2m
```

5. Verify that a corresponding pod was created:

```
> kubectl get pods
NAME                    READY     STATUS              RESTARTS   AGE
foobar-f94c78b8b-bjt74  1/0       Running             0          2m
```

6. A very useful feature of `kubectl` is its option to produce output in json format. Explore running the previous `kubectl` with the `-o json` argument. This is particularly useful when creating shell scripts that need to parse the output of `kubectl`.
7. Suppose you need to know on which host the pod that we created is running. Inspect the output of `kubectl -h` to find out which command will let you know in which host a pod is running.
8. Delete the deployment:

```
> kubectl delete deployment foobar
```

9. Verify that the foobar pods have been removed.

Part 3: Deploy a custom image to Kubernetes

In Part 1 we deployed an nginx webserver in a Pod. However, that was not particularly useful because the service was never *exposed* to the outside world. In this section we are going to create an application that will return the name of the container where it is running. In Part 4 we will scale the application and we are going to expose it so it can be called from the web.

1. **On your own:** Create a Docker image that executes a golang application that runs an http server that listens on port 80. When that service is called, it returns the name of the host where is it running (this value is returned by the `os.Hostname()` method). For example, suppose the application is running in a container whose name is `5f17eddbe42d`, and that container has an IP Address of `172.16.1.1`. When the endpoint is called using a `curl` command, it should return the following:

```
> curl http://172.16.1.1
5f17eddbe42d
```

2. Make sure you test your image locally.
3. Tag the image as `gcr.io/<YOUR_GCP_PROJECT_ID>/echohost:latest` (replace your project id accordingly). In case you need help with the `docker tag` command, the documentation is available at https://docs.docker.com/engine/reference/commandline/tag/ (https://docs.docker.com/engine/reference/commandline/tag/).
4. Run the `gcloud auth configure-docker` to setup your credentials for the GCP Container Registry.
5. Push the image to the Google Cloud Container Registry. Note that in the next command we will use a gcloud utility to push to the registry instead of the normal `docker push` command. The reason is because the `gcr.io` requires some additional steps to authenticate.

```
> gcloud docker -- push gcr.io/<YOUR_GCP_PROJECT_ID>/echohost:latest
```

6. Verify that the image was pushed to the container registry in the GCP web console (Left navigation menu, under **Tools** > **Container Registry**)
7. Deploy the application to your kubernetes cluster:

```
> kubectl run echohost --image=gcr.io/<YOUR_GCP_PROJECT_ID>/echohost:latest
```

8. Run `kubectl get pods` to verify that the pod is running

Part 4: Managing Pods, Deployments, Services and ReplicaSets

At this point we have a custom application deployed to a kubernetes cluster. In this Part of the lab you will use `kubectl` commands to enable the app to do useful work.

1. First, we are going to make our service accessible to the outside world. The following command will create a **Service** (https://kubernetes.io/docs/concepts/services-networking/ (https://kubernetes.io/docs/concepts/services-networking/)):

```
> kubectl expose deployment echohost --port 80 --type LoadBalancer
```

This command will create a service with a load balancer with an external IP Address associated to it. The service takes care of forwarding incoming requests to the pods in the deployment.

2. Confirm that the service is running. Wait until an External IP address is assigned.

```
> kubectl get  services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
echohost     LoadBalancer   10.59.251.135   35.185.250.86   80:32441/TCP   2m
kubernetes   ClusterIP      10.59.240.1     <none>          443/TCP        18h
```

3. Once an External IP Address has been assigned, open another terminal (don't log into **lab03**) and test your service with `curl` (Use your own IP Address, and of course, the return name in your case will be different).

```
> curl http://35.185.250.86
echohost-b88c69458-pxzcx
```

Run the previous command several times. The output should not change because the deployment that we created has only one Pod. Compare the output of the service with the output of the `kubectl get pods` command.

4. Scale the service by creating a **ReplicaSet** (https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/))

```
> kubectl scale deployment echohost --replicas 3
```

5. Verify that a replicaset was created:

```
> kubectl get replicaset
```

6. Verify the effect of creating a replicaset in the pods that are running. Also, what is the effect of scaling the service in the output of your service (e.i. when you `curl` your service's External IP Address)? Explain why you observe this behavior.

What to turn in:

1. A link to a github repository where you provide all sources required for building the image for Part 3. You shoud include at a minimum one golang file with the application source code, the Dockerfile required to build the image, and either a Makefile or a script that performs the compilation of the golang source. **Do not include compiled binaries!** As in Lab 2, assume the Docker image needs to be built on a machine that does not have the Golang distribution, the only software that is guaranteed to be installed on that system are docker, make, git and curl/wget. If there is another tool that you require to build your submission, make sure that you ask your instructor first and document it in the repository.
2. Answer the questions of the last item of Part 4.

Once you are done with the lab, remove the cluster with this command:

```
> gcloud container clusters delete lab03cluster --zone us-west1-c
```