ITG-408 Introduction to Docker & K8S

“Samen Sterker, Beter, Slimmer en Leuker!”

April 28, 2021

Contents

Contents **ii**

1 Getting started with Containers **1**
1.1 Docker Installation 1
1.2 Running Containers 3
1.3 Create your own Images 5
1.4 Experiments with persistance 11
1.5 Container Registries 14
1.6 Using Volumes 22

2 Introduction to Kubernetes **25**
2.1 Kubernetes Installation 25

3 Managing K8S **33**
3.1 First Steps 33
3.2 Introduction to kubectl 41

4 Kubernetes PODs **45**
4.1 Hostnames & sub-domains 45
4.2 Probes 51

5 Kubernetes Deployments **63**
5.1 Creating Deployments 63
5.2 Communicating with PODs managed by Deployments 66
5.3 Upgrading PODs in Deployments 68

6 Kubernetes DaemonSets **71**
6.1 Creating DaemonSets 71
6.2 Communicating with PODs managed by DaemonSets 72
6.3 Upgrading PODs in DaemonSets 74

7 Kubernetes storage **75**

8 Kubernetes ConfigMaps **79**

9 Kubernetes Secrets **89**

1 Getting started with Containers

Virtual Machine Name:	....
Virtual Machine IP Address:	....
User Name:	....
User Password:	....
Root Password:	....

1.1 Docker Installation

Prerequisites

In order to be able to install the Docker Community Edition, you have to make sure that your installation includes the latest updates, and you need to install some dependencies.

Log in to your virtual machine as a normal user.

By default, it is not possible to use https repositories, and the utility `add-apt-repository` which provides an easy way to add them, is also not installed.

First, update your current installation:

```
$ sudo apt update

$ sudo apt upgrade
```

Install the necessary tools:

```
$ sudo apt-get install apt-transport-https \
    ca-certificates curl gnupg2 \
    software-properties-common
```

Install Docker CE

With all the dependencies installed, you are ready to install Docker.

Import the public GPG key of the Docker repository. This allows you to verify that the repository metadata has not been tampered with since it was generated by the repository host.

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
| sudo apt-key add -
```

Add the repository:

```
$ sudo add-apt-repository \
   "deb https://download.docker.com/linux/ubuntu/ bionic stable"
```

Resynchronize the package index files on your system:

```
$ sudo apt update
```

Finally, install Docker:

```
$ sudo apt install docker-ce docker-ce-cli containerd.io
```

It is not a good idea to administer Docker as root user. To be able to execute administrative commands to the Docker Engine, add your login to the docker group

```
$ sudo usermod -aG docker $USER
```

Logout, and login again, to acquire the group membership.

Verify your installation

The Docker engine is already started. To verify this, execute:

```
$ sudo systemctl status docker
```

The first lines of the output will look similar to this:

```
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled;
           vendor preset: enabled;
   Active: active (running) since Mon 2019-04-01 11:49:37 CEST; 10min ago
     Docs: https://docs.docker.com
 Main PID: 7900 (dockerd)
```

loaded: The unit file is loaded into memory

enabled: The unit will be started at boot

vendor preset: Enabled while installing the software

active: The unit is up and running

The `docker` command is available to interact with the docker daemon, and can be used to acquire more information about the installation:

```
$ docker --version

$ docker info
```

Just have a quick look into the output. For now it's not important to understand all of it, we just want to make sure that you are, as a non-privileged user, able to communicate with the Docker Engine.

1.2 Running Containers

Hello World

To further test our installation, let's run our first container as a test:

```
$ docker run hello-world
```

This command does several things, as stated in the output:

1. The Docker client contacted the Docker daemon
2. The Docker daemon pulled the "hello-world" image from the Docker Hub (an online repository of images)
3. The Docker daemon created a new container from that image
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal

Execute the command:

```
$ docker image ls
```

The output should be similar to:

```
REPOSITORY    TAG     IMAGE ID      CREATED       SIZE
hello-world   latest  fce289e99eb9  3 months ago  1.84kB
```

This image is now stored locally, if you want to run it again, just execute:

```
$ docker run hello-world
```

It is possible to download an image, without running it:

```
$ docker pull ubuntu

$ docker images
```

Allocate TTY

The hello-world container, started, echoed a message, and because the job it's done, it quited. The same thing will happen if we just run the ubuntu container. Instead of that, let's allocate a TTY, a virtual terminal, to the ubuntu container.

```
$ docker run -it ubuntu
```

The standard output of the container is now connected to the a tty, displayed in your terminal. Press Ctrl + d to get back your command prompt.

Attach / Deattach Containers

Let's download another image and run it: the NGINX webserver

```
$ docker run nginx
```

Remember the hello-world container: it echoed a message and quited. But now, the container seems to 'hang'. The container is running, the last command executed actually started the webserver, and that is still attached to your standard output.

This is, normally, not the behaviour you want. Press Ctrl + c , and repeat the command, adding a parameter to detach the output of the terminal from your standard output:

```
$ docker run --detach nginx
```

Start / Stop Containers

The ps command is available to view the running containers:

```
$ docker ps
```

You should see at least 3 containers: 2 Ubuntu instances and NGINX. Let's stop the containers, using id's:

```
$ for c in $(docker ps -q);
  do docker stop $c;
done
```

Start, using the id again, the NGINX container:

```
$ docker start <id>
```

Publish Ports

Let's have a better look into the process list:

```
$ docker ps --latest --no-trunc
```

In the column PORTS, 80/tcp is listed. But you can't access this port, it's only available in the container.

Stop and remove the container:

```
$ docker stop <id>

$ docker rm 757adf79ff91
```

Run the container again, adding the –ports parameter:

```
$ docker run --detach --publish 81:80/tcp nginx

$ docker ps
```

This command will map port 80 in the container to port 81 on your local machine.

You can use `curl` to verify:

```
curl http://localhost:81
```

Clear everything up:

```
$ docker stop <id>

$ docker system prune
```

1.3 Create your own Images

First steps

Instead of pulling images from an online repository, you can created them yourself using the `docker build` command. This command build images, using a file with the name Dockerfile as an input file and built the images given the commands in this file.

Create a working directory:

```
$ mkdir ~/dockerbuild

$ cd ~/dockerbuild
```

And create the file 'Dockerfile' with the following content:

```
FROM alpine:latest
```

```
RUN apk add --update nginx

CMD ["nginx", "-g", "daemon off;"]
```

First it will pull the latest version of the alpine container, using this image as the base layer.

Alpine is a very small Linux distribution, especially build for usage in containers. To install packages you have to use the `apk` command. The second layer will contain the installed NGINX package.

If run the container, the command `nginx` will be executed with some parameters.

Build the image, let's give the name mynginx and version it as the latest one:

```
$ docker build --tag mynginx:latest .

$ docker images
```

And run it:

```
$ docker run mynginx:latest
```

The output of the `docker run` command states:

```
nginx: [emerg] open() "/run/nginx/nginx.pid" failed
(2: No such file or directory)
```

A directory is missing in the container, we need to add this directory in the container. Modify the Dockerfile:

```
FROM alpine:latest

MAINTAINER ITGilde  <info@itgilde.nl>

RUN apk add --update nginx
RUN mkdir /run/nginx/

CMD ["nginx", "-g", "daemon off;"]
```

Build and run it again:

```
$ docker build --tag mynginx:latest .

$ docker run --detach mynginx

$ docker ps
```

In the output of the `docker ps` command, we're missing the PORTS that we can "publish". Modify the file again:

```
FROM alpine:latest

MAINTAINER ITGilde  <info@itgilde.nl>

RUN apk add --update nginx
RUN mkdir /run/nginx/

EXPOSE 80  443

CMD ["nginx", "-g", "daemon off;"]
```

And try again.

Inspect your work

With the help of the `inspect` command you can find out more about the image you just build:

```
$ docker image inspect mynginx
```

Parts of the output:

```
[
    {
        "Id": "sha256:...
        "RepoTags": [
            "mynginx:latest"
        ],
        "Parent": "sha256:...
        "ContainerConfig": {
          "ExposedPorts": {
                "443/tcp": {},
                "80/tcp": {}
            },
        ]"Cmd": [
            "/bin/sh",
            "-c",
            "#(nop) ",
            "CMD [\"nginx\" \"-g\" \"daemon off;\"]"
        ],
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:...
                "sha256:...
                "sha256:...
            ]
          },
```

Apparently, the container consists of 3 layers: 2 RUN commands and a read/write layer on top of it. Having too layers much can be bad for the image size and performance. Having fewer layers reduces the complexity and maintainability of an image in the long term. You can reduce the number of builds using:

```
$ docker build --tag <tag>
```

on the latest Docker versions, or you can combine commands using a double ampersand:

```
FROM alpine:latest

MAINTAINER ITGilde  <info@itgilde.nl>

RUN apk add --update nginx && mkdir /run/nginx/

EXPOSE 80  443

CMD ["nginx", "-g", "daemon off;"]
```

Using interactive run mode, you can also debug your container:

```
docker run -it mynginx:latest /bin/sh
```

Run the `ps` command in the container, and notice that the command `/bin/sh` actually replaced the `nginx` command.

Leave the interactive mode using Ctrl + q , Ctrl + p

TIP: If this key combination doesn't work for you, create a file `~/.docker/config.json`, and change it like this:

```
{
    "detachKeys": "ctrl-e,e"
}
```

Docker Hub

We already pulled images from an online repository. In Docker terminology a repository is called registry. By default, Docker is configured to use Docker Hub:

```
$ docker info | grep Registry
```

```
Registry: https://index.docker.io/v1/
```

You can use `docker search` to look for available images:

```
$ docker search alpine
```

It's a good idea to limit the output to official images:

```
$ docker search --filter "is-official=true" alpine
```

This way, you are sure that you receive an image maintained that is delivered by the vendor or developer behind this image. On top of that, these images are scanned for vulnerabilities.

Publish your own Image

If you are happy with the image, you can publish it to Docker Hub.

Go to `https://hub.docker.com` and get yourself a free Docker Account. After clicking on the verification mail, you are able to login.

After you logged in, click on "Create Repository" and create a public repository.

Back in a terminal, login to your account:

```
$ docker login
```

Find the image id from the image build in the previous lab:

```
$ docker image ls mynginx:latest
```

Tag your image for storing on Docker Hub:

```
$ docker tag <image id> <account name>/<image name>:latest
```

Push the image:

```
$ docker push <account name>/<image name>:latest
```

Use `docker search` to verify, ask another student to run the image.

1.4 Experiments with persistance

In this part of the labs we will research the property of persistance of data in a container

If needed please clean your docker environment from any running containers. E.g. check with `docker ps` and remove any containers with `docker rm -f <container id`container name>|

Please spin up a new NGINX container, have port 80 inside of the container project to port 80 on the host system.

```
$ docker run --name nginx -d  -p 80:80 nginx:1.17.1
```

Check with `docker ps` if the container is running:

```
$ docker ps
```

Good. Now let's find out where NGINX does have it's DocumentRoot located. Please log on to the container with `docker exec`

```
$ docker exec -it <container id|container name> sh
```

In the container navigate to /usr/share/nginx/html

```
# cd /usr/share/nginx/html
# ls
```

You will find the `DocumentRoot` files in this directory.

Question: try to alter the `index.html` file. How are you going to do that?

As there's no VI or any other editor in the container image, we have to use for example `echo` to write in the `index.html` file.

```
# echo "Welcome to K8S4ALL" > index.html
# cat index.html
```

Exit the container using the `exit` command.

On the docker host execute a `curl` to see if you can reach the new home page:

```
$ curl http://localhost
```

Ok. The new homepage should be visible. Now stop the container using docker stop.

```
$ docker stop <container-id|container-name>
```

Now start it again:

```
$ docker start <container-id|container-name>
```

And go to the webpage again

```
$ curl http://localhost
```

Please explain what you see.

Now remove the container using `docker rm -f`

```
$ docker rm -f  <container-id|container-name>
```

Verify if the container is gone

```
$ docker ps
```

Please Re-run the container:

```
$ docker run --name nginx -d  -p 80:80 nginx:1.17.1
```

Navigate to the home page again

```
$ curl http://localhost
```

What do you see? Please explain what you see.

When you stop the container the last RW layer will still be intact. When you start or restart the container this RW layer will be picked up again and you will see you altered `index.html` file. Once you remove the container, the top RW layer will be removed and when you run the container image again you will see that the changes were not perststistant.

So in order to provide a container with persistant storage, storage that will have a longer lifetime than the container, we will need some additional functionality. For this, see the lab section about `volumes` in which we will explain persistant storage using Docker Volumes.

1.5 Container Registries

In this section you will learn how to create:

- In-secure docker registries
- Secure docker registries
- Authenticated secure docker registries

Preparations

To have an image that we can put into our registries we are going to build one using techniques we already have discussed in prior sections.

Please clone the following git repository in your home directory

```
$ git clone https://thegitcave.org/pascal/sample-container.git
```

A directory sample-container will be created in your home-directory. Please navigate inside this directory and start building the container; after that verify with `docker image ls` that the image has been built.

```
$ cd sample-container
$ docker build . --tag sample-container:v1.0
$ docker image ls
```

Feel free to experiment with the name of the image and its tag. However, keep it consistent as you will need it for other excercises too.

Creating an insecure Docker registry

An insecure registry is a registry that has its certificates generated and signed by Docker itself. As such there's no authority guaranteeing the authenticity of the certificates and it's called a insecure registry.

Create a directory to store your container images that will be managed by the registry:

```
$ sudo mkdir /srv/registry
```

Create an insecure registry by launcing a new container with the image registry:v2. The image will be pulled from Dockerhub. We will have the registry listening on port 5000. As it's important to have the images also available after the registry container has been removed we will instruct it to use the directory /srv/registry as persistant storage. The `--restart=always` option makes sure that the container is always restarted. Also after a reboot of the host or restart of the docker service.

```
$ docker run --detach \
--restart=always \
--name registry \
--publish 5000:5000 \
--volume /srv/registry:/var/lib/registry \
registry:2
```

Let's do the first test-drive on this registry. Like the pushing of the image to docker hub we have to `tag` the (built) image to have it directed to our private registry. Please tag the image using the hostname where your registry is hosted on. E.g: st03node01. As the registry is listening on port 5000, we need to include the port-nr too, e.g: `st03node01:5000`

```
$ docker tag sample-container:v1.0 st03node01:5000/sample-container:v1.0
```

And push it to the private registry

```
$ docker push st03node01:5000/sample-container:v1.0
```

Ok. That didn't work so well. You most probably got an error like this:

```
The push refers to repository [st00node01:5000/sample-container]
Get https://st00node01:5000/v2/: http: server gave HTTP response to HTTPS client
```

A cryptic and confusing message; but what Docker is trying to tell us is that it is not trusting this registry. We have to explicitly tell docker to 'trust' these insecure registries. We do this by adding a special 'insecure-registry' entry (or more than one) in `/etc/docker/daemon.json`

You can check the insecure registries that docker trusts with the `docker info` command:

```
$ docker info
```

```
...
Experimental: false
Insecure Registries:
 st00node01.itgildelab.net:5000
 st00node01:5000
 127.0.0.0/8
Live Restore Enabled: false
...
```

Please create a file called `/etc/docker/daemon.json` with root like this:

```
$ vi /etc/docker/daemon.json
```

It's content should be:

```
{
  "insecure-registries" : [ "st00node01:5000", "st00node01.itgildelab.net:5000"]
}
```

Of course you will have to use the hostname of your own system instead of that of the st00node01.

Any time we alter the `/etc/docker/daemon.json` we need to restart the docker-daemon in order for it to re-read and process this config file. Please do this with the following commands:

```
$ sudo systemctl restart docker
```

This command should not return an error. If it does, please inspect the syntax of your `/etc/docker/daemon.json` content very carefully.

Now retry the push to the registry of your crafted image:

```
$ docker push st03node01:5000/sample-container:v1.0
```

Now remove the old images from the system:

```
$ docker image rm st03node01:5000/sample-container:v1.0
$ docker image rm sample-container:v1.0
```

And try to pull the image in again from your private insecure registry:

```
$ docker pull st03node01:5000/sample-container:v1.0
```

This concludes our lab regarding the creation of an insecure registry.

Creating a secure Docker registry

A secure Docker registry is a registry where we will supply the certificates ourselves. This can be certificates that are officially distributed by an authorized agency or certificates that are self signed by us.

Prepare the installation of the secure registry by creating another directory to hold the images that we will have managed by the secure registry.

```
$ mkdir /srv/secregistry
```

We also need a directory to store the certificate and our private key in. Ofcourse this directory must be secured from prying eyes by proper permissions and ownership. For our labs we will store the certificate and key in our home directory.

Create the following script: `mkcerts` in your homedirectory.

```
REGSERVERNAME="st99node01.itgildelab.net"

mkdir -p certs
cd certs

openssl req -new -sha256 -newkey rsa:4096 -x509 -sha256 \
 -nodes -days 365 -out ${REGSERVERNAME}.crt -keyout ${REGSERVERNAME}.key \
 -subj "/C=NL/ST=LB/O=Acme, Inc./CN=${REGSERVERNAME}"
```

Change the name of REGSERVERNAME to the name of the docker host that will host your secure registry container. (E.g. st03node01.itgildelab.net)

Put executable rights on the script and execute it to create the certificate and key. Feel free to change details of the cert but notice that `CN` should have the hostname of your docker host assigned to it.

```
$ chmod +x ./mkcerts
$ ./mkcerts
```

To create and launch the secure registry create the following script: `mksecreg` in your home directory

```
REGSERVERNAME="st99node01.itgildelab.net"

docker run -d \
  --restart=always \
  --name secregistry \
  -v ${PWD}/certs:/certs \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/${REGSERVERNAME}.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/${REGSERVERNAME}.key \
  -p 443:443 \
  -v /srv/secregistry:/var/lib/registry \
  registry:2
```

Put executable rights on the script and execute it to create the registry. Please take note of the following:

- THe registry will be called secregistry
- It will run on port 443 and be exposed on 443 on the docker host
- It will have ENV VARS that will tell it where to find the CERT
- It will have an ENV VAR that tells it on which port to listen inside the container
- It has one volume for the registry's contents
- It has another volume for the CERTS store in our home-directory

```
$ chmod +x ./mksecreg
$ ./mksecreg
```

The registry will be launched. Please verify with `docker ps`

```
docker ps
```

Let's do the first test-drive on this secure registry. Like the pushing of the image to the insecure registry have to `tag` the (built) image to have it directed to our secure registry. Please tag the image using the hostname where your registry is hosted on. E.g: st03node01. As the registry is listening on port 443, we DO NOT need to include the port-nr e.g: `st03node01`

Extra note: as we removed the sample-container:v1.0 image we need to use the image tag of the insecure registry to re-tag the image.

```
$ docker tag st03node01:5000/sample-container:v1.0 st03node01.itgildelab.net/sample-container:v1.0
```

And push it to the private registry

```
$ docker push st03node01.itgildelab.net/sample-container:v1.0
```

Ok. That didn't work so well either. You most probably got an error like this:

```
denied: requested access to the resource is denied
```

What needs to be done is we need to tell Docker that we do have a cert for this registry that it can use.

```
$ sudo mkdir -p /etc/docker/certs.d/st03node01.itgildelab.net
$ sudo cp certs/st03node01.itgildelab.net.crt /etc/docker/certs.d/st03node01.itgildelab.net
```

No restart of docker-daemon is needed here. We can just retry

```
$ docker push st03node01.itgildelab.net/sample-container:v1.0
```

This should now succeed. Please try also to pull the image.

Creating an authenticated secure Docker registry

The goal of this excercise is to create a secure registry that has authentication functionality. E.g. we need to have to log on to it in order to be able to pull or push images from/to it.

We will use the secure registry as built in the previous lab excercise and extend this one.

Please remove the current running secure registry

```
$ docker ps | grep registry | grep 443
$ docker rm -f <container-id|container-name>
```

Authenticated secure registries make use of so called HTTP Basic Authentication, hence we are going to use an apache utility for this.

Create a `htpassword` file for our authenticated secure registry. Create a script called `mkauth` with the following content in your home directory.

```
mkdir -p auth
rm -f auth/htpasswd
for N in 1 2 3 4 5 6 7 8
do
  docker run \
    --entrypoint htpasswd \
      registry:2 -Bbn student0${N} mysecret >> auth/htpasswd
done
```

Execute the script to create the htpasswd file that will authorize users student01..student08 with password `mysecret` for access to the registry

```
$ sh ./mkauth
```

Next create a script `mksecregauth` that will create a new secure registry that will use the existing certs and get it's authentication info from the directory in your home-directory:

```
REGSERVERNAME="st00node01.itgildelab.net"

docker run -d \
  --restart=always \
  --name secregistry \
  -v ${PWD}/certs:/certs \
  -v ${PWD}/auth:/auth \
  -e REGISTRY_AUTH=htpasswd \
  -e REGISTRY_AUTH_HTPASSWD_REALM="ITGILDELAB SECURE REGISTRY" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/${REGSERVERNAME}.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/${REGSERVERNAME}.key \
  -p 443:443 \
  -v /srv/secregistry:/var/lib/registry \
  registry:2
```

Do not forget to alter the REGSERVERNAME variable to it has the value of your docker host

Next: execute the script to launch the registry

```
$ ./mksecregauth
```

Try to pull an image from the newly created authenticated registry:

```
$ docker pull st00node01.itgildelab.net/pamvdam/samplecontainer:v1.0
```

This will result in an error like:

```
Error response from daemon: Get https://st00node01.itgildelab.net/v2/pamvdam/samplecontainer/manifest
```

This is correct as we have nog logged on yet. So let's logon using student01 user and password `mysecret`

```
$ docker login st00node01.itgildelab.net -u student01 -p mysecret
```

This will result in a `login succeeded`

Now let's try to pull the image again:

```
$ docker pull st00node01.itgildelab.net/pamvdam/samplecontainer:v1.0
```

This will work now. Please try to tag a new image and push it to the registry and pull it again.

This concludes our lab excercise on authenticated secure registries.

1.6 Using Volumes

If you want provide data from your host into the container, volumes are the preferred way to supply this persisting storage.

As example, we're going to use another webserver: Apache httpd.

```
$ docker search --filter "is-official=true" apache
```

Run this container in interactive mode:

```
$ docker run -it httpd:latest /bin/sh
```

Find the location where Apache expect the html files:

```
$ grep DocumentRoot /usr/local/apache2/conf/httpd.conf
```

It’s the directory: /usr/local/apache2/htdocs. Create on the host a new directory:

```
mkdir ~/public_html

cd ~/public_html
```

And create a simple document:

```
$ cat << EOF >> index.html
<html>
<head>
<title>test</title>
</head>
<body>
<h1>test</h1>
</body>
</html>
EOF
```

Mount this directory as a volume into the container:

```
$ docker run --detach --name www \
  --mount type=bind,source=/home/student/public_html \
  target=/usr/local/apache2/htdocs \
  --publish 80:80/tcp httpd:latest
```

Test it:

```
$ curl http://localhost
```

Another way to verify what is mounted is to use `inspect`

```
$ docker inspect www
```

In the output you’ll find:

```
"Mounts": [
  {
    "Type": "bind",
    "Source": "/home/student/public_html",
    "Destination": "/usr/local/apache2/htdocs",
    "Mode": "",
    "RW": true,
    "Propagation": "rprivate"
  }
]
```

2 Introduction to Kubernetes

2.1 Kubernetes Installation

Information needed for the master node:

Virtual Machine Name:	. . . .
Virtual Machine IP Address:	. . . .
User Name:	. . . .
User Password:	. . . .
Root Password:	. . . .
POD Network CIDR: 192.168.0.0/16	. . . .

Information needed for the worker node:

Virtual Machine Name:	. . . .
Virtual Machine IP Address:	. . . .
User Name:	. . . .
User Password:	. . . .
Root Password:	. . . .

Prerequisites

As a normal user, login on the master node, that we used in the previous labs.

Before we start, clean everything:

```
# Decomission the SWARM cluster

$ ssh student01@st01node02.itgildlab.net docker swarm leave
$ ssh student01@st01node03.itgildlab.net docker swarm leave

# On the SWARM master node

$ docker swarm leave --force
```

On every node, clean up any running containers

```
$ for c in $(docker ps -q);
   do docker stop $c;
done

$ docker system prune
```

In Debian and Ubuntu, Docker is configured to access cgroups directly. This is not recommended if you are going to use Kubernetes. Actually, in general, it's better to use the systemd driver to access the cgroups.

Reconfigure Docker:

Please note all these commands need to be executed as the root user!

```
$ sudo su  -

$ cat << EOF > /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": { "max-size": "100m" },
  "storage-driver": "overlay2"
}
EOF

$ exit
```

Create a directory for the control files:

```
sudo mkdir -p /etc/systemd/system/docker.service.d
```

Restart Docker:

```
$ sudo systemctl daemon-reload

$ sudo systemctl restart docker
```

And verify:

```
$ docker info | grep Cgroup
```

A requirement for Kubernetes is to have swap disabled, otherwise the kubelet service will not start on the masters and nodes. The idea of kubernetes is to tightly pack instances to as close to 100% utilized as possible. Disable swap:

```
$ sudo swapoff -a

$ sudo systemctl stop swap.target

$ sudo systemctl mask swap.target

$ sudo sed -i '/swap/d' /etc/fstab
```

Verify the number of cpu's:

```
$ lscpu
```

You'll need at least 2 cores. Check the amount of memory in your system:

```
$ vmstat -sSm | grep memory
```

On the master node, you'll need at least 2,5GB memory, on a worker node 1GB is enough.

Kubernetes Installation: Master

If you met all the requirements, you're ready to install Kubernetes.

First, import the GPG key, like we did for the Docker repository:

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg \
  | sudo apt-key add -
```

```
$ sudo apt-cache policy | grep apt.kubernetes.io || \
    sudo apt-add-repository "deb http://apt.kubernetes.io/ \
    kubernetes-xenial main"
```

- Note 1: If you see a message with a `500` in it, ignore it, it's ok.
- Note 2: This repository must be used for all recent Debian and Ubuntu distributions.

Resynchronize the package index files on your system:

```
$ sudo apt update
```

Finally, install Kubernetes:

```
$ sudo apt install kubeadm=1.19.3-00 kubelet=1.19.3-00 kubectl=1.19.3-00 --allow-downgrades -y
```

Configure the master and define the subnet:

On Azure we will use `192.168.0.0/16` for the POD Network

```
$ sudo kubeadm init --pod-network-cidr <private subnet>
```

This will take a few minutes. The output at the end of this command is important. First it explains how to configure the directories for the kubectl cluster configuration.

Please revert back to your normal user (e.g. student01) and execute these commands:

Note these commands should NOT be executed with the root user!

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

In the kubeadm output you can also find the command that we need to use to have the (Worker) Nodes join the cluster. This command will be of the form:

```
$ kubeadm join <ip address:6443 --token <token> \
    --discovery-token-ca-cert-hash sha256:<hash>
```

Copy this command and save it to a text file.

TIP: If you forgot to take notice of this command, generate a new token:

```
$ sudo kubeadm token create --print-join-command
```

Kubernetes Installation: (Worker) Nodes

For each of your worker nodes login to them with your student01 user

Become root:

```
sudo su -
```

```
# Bring your system up-to-date
$ sudo apt update

$ sudo apt upgrade
```

In Debian and Ubuntu, Docker is configured to access cgroups directly. This is not recommended if you are going to use Kubernetes. Actually, in general, it’s better to use the systemd driver to access the cgroups.

Reconfigure Docker:

Please note all these commands need to be executed as the root user!

```
$ sudo su  -

$ cat << EOF > /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": { "max-size": "100m" },
  "storage-driver": "overlay2"
}
EOF

$ exit
```

Create a directory for the control files:

```
sudo mkdir -p /etc/systemd/system/docker.service.d
```

Restart Docker:

```
$ sudo systemctl daemon-reload

$ sudo systemctl restart docker
```

And verify:

```
$ docker info | grep Cgroup
```

A requirement for Kubernetes is to have swap disabled, otherwise the kubelet service will not start on the masters and nodes. The idea of kubernetes is to tightly pack instances to as close to 100% utilized as possible. Disable swap:

```
$ sudo swapoff -a

$ sudo systemctl stop swap.target

$ sudo systemctl mask swap.target

$ sudo sed -i '/swap/d' /etc/fstab
```

Verify the number of cpu's:

```
$ lscpu
```

You'll need at least 2 cores. Check the amount of memory in your system:

```
$ vmstat -sSm | grep memory
```

On a worker node we will need at least 1GB of memory.

If you met all the requirements, you're ready to have the (Worker) nodes to join the cluster.

First, import the GPG key, like we did for the Docker repository:

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg \
  | sudo apt-key add -
```

```
$ sudo apt-cache policy | grep apt.kubernetes.io || \
    sudo apt-add-repository "deb http://apt.kubernetes.io/ \
    kubernetes-xenial main"
```

Note: This repository must be used for all recent Debian and Ubuntu distributions.

Resynchronize the package index files on your system:

```
$ sudo apt update
```

Instell the needed Kubernetes packages:

```
$ sudo apt install kubeadm=1.19.3-00 kubelet=1.19.3-00 kubectl=1.19.3-00 --allow-downgrades -y
```

Have the (Worker) Node join the cluster by executing the saved kubeadm command on the (Worker) Node, this must also be done as the root user.

```
$ kubeadm join ...
```

Leave the node and repeat the procedure for your remaining (Worker) Nodes.

When you have finished preparing the last Worker Node, login into the master again with your student01 user and observe the status of your cluster:

```
$ kubectl cluster-info
```

The first line from output of this command should be:

```
Kubernetes master is running at https://<ip address>:6443
```

Container Network Interface

However if you view the status of the nodes:

```
$ kubectl get nodes
```

The cluster is not ready:

```
NAME         STATUS     ROLES    AGE   VERSION
master       NotReady   master   62m   v1.19.3
node1.test   NotReady   <none>   10m   v1.19.3
```

Try to find out why by running the following command:

```
$ kubectl describe nodes
```

Try to explain what's preventing the nodes to come into the Ready status

For Azure we need to use the so called Canal plugin, which is actually a hybrid form of the Flannel and Calico Network Plugins

Install the plugin:

```
$ kubectl apply -f http://st99node01/DL/canal.yaml
```

After that, observe the status of your cluster with the following commands:

```
$ kubectl get nodes
$ kubectl get pods -n kube-system
$ kubectl describe nodes
```

Kubectl Commandline Completion

The `kubectl` tool also support commandline completion which you most probably already are familiar with if you worked with `bash` before. In order to enable the commandline completion facility for `kubectl` execute the following commands as your own user (e.g. student01):

```
$ echo 'source <(kubectl completion bash)' >> ~/.bashrc
$ source ~/.bashrc
```

Try to complete the following commands using the `TAB` key;

```
$ kubectl get po<TAB>
$ kubectl get depl
```

This concludes the lab for our introduction to K8S

3 Managing K8S

3.1 First Steps

Create your first pod

Information of the master node:

Virtual Machine Name:	....
Virtual Machine IP Address:	....
User Name:	....
User Password:	....
Root Password:	....
Kubernetes Network Subnet:	....

It's time to run the first pod on our new Kubernetes cluster, and we're going to use a deprecated command for it: `docker run`. The reason that it is deprecated, is that the number of parameters went out of control and became a little bit overwhelming for new users. Login into the master node, and execute:

```
$ kubectl run nginx1 --image nginx:latest
```

In K8S versions prior tot 1.18 this would actually lead to the creation of a `deployment` instead of a single POD.

in K8S version 1.19 and later, this will lead to the creation of a single `POD`.

Let's examine the result:

```
$ kubectl get pods
```

The output shows you the newly created pod:

```
NAME      READY    STATUS     RESTARTS    AGE
nginx1    1/1      Running    0           50s
```

Want to know on what node the POD has been scheduled? Or which POD IP it got assigned? Use `kubectl get pods -o wide`

And with even more details over the specific pod:

```
$ kubectl describe pods/nginx1
```

A part of the output:

```
Name:               nginx1
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               node1.test/192.168.122.101
Start Time:         Tue, 02 Apr 2019 12:17:46 +0200
Labels:             run=nginx1
Annotations:        cni.projectcalico.org/podIP: 192.168.174.132/32
Status:             Running
IP:                 192.168.174.132
```

Note: you can also export it into YAML format:

```
$ kubectl get pods/nginx1 -o yaml > nginx1.yaml
```

The echoserver has gotten an IP address from Calico. Ping this address:

```
$ ping -c1 <IP address>
```

Test the webserver:

```
$ curl http://<IP Address>
```

Delete the nginx1 pod:

```
$ kubectl delete pod/nginx1
```

Open the nginx1.yaml file and remove all lines in the `metadata` section except:

```
  labels:
    run: nginx1
  name: nginx1
  namespace: default
```

And remove the complete status section.

Create the pod again, using the YAML file:

```
$ kubectl create -f nginx1.yaml
```

Scaling

One of the most powerfull features of Kubernetes is that you can scale up the number of pods to serve your application. This is done by a replication controller.

Create a YAML file, nginx1-rc.yaml with the following content:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

```
$ kubectl create -f nginx1-rc.yaml
```

Check the results:

```
$ kubectl get pods
```

```
NAME          READY   STATUS    RESTARTS   AGE
nginx-2d84h   1/1     Running   0          59s
nginx-5bz4s   1/1     Running   0          59s
nginx-qrnpv   1/1     Running   0          59s
```

And view the status of the replication controller:

```
$ kubectl get rc/nginx
```

```
NAME    DESIRED   CURRENT   READY   AGE
nginx   3         3         3       102s
```

Scale up manually:

```
$ kubectl scale --replicas=4 rc nginx
```

Review the status. More information can be requested with:

```
$ kubectl describe rc nginx
```

Now let's do some experiments.

Find out which PODs are running with `kubectl get pods -o wide` and start delete some PODs. Check again with `kubectl get pods -o wide`. Repeat this a few times, what do you see? Please explain. Look at the POD IP addresses, what happens to them?

Clean up everything:

```
$ kubectl delete -f nginx1.yaml

$ kubectl delete -f nginx1-rc.yaml
```

Instead of using the replication controller, you can use replicaset, which you can consider as the next generation replica controller.

A replicaset is a part of a deployment. Deployments represent a set of multiple, identical pods with no unique identities. Deployments manage replicasets. Normally only a single one, but during upgrades there could be multiple, one for each version.

Create a new YAML file, nginx-dpl.yaml, with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx1
  template:
    metadata:
```

```
    labels:
      app: nginx1
  spec:
    containers:
    - name: nginx
      image: nginx:latest
      ports:
      - containerPort: 80
```

Deploy it, and view the status with the commands:

```
$ kubectl get pods


$ kubectl get deployment/nginx-deployment


$ kubectl describe deployments/nginx-deployment
```

The `deployments/nginx-deployment` is a notation that says nginx-deployment is a resource of the type deployment. Another valid notation is using spaces like:

```
$ kubectl get deployment nginx-deployment


$ kubectl describe deployment nginx-deployment
```

These commands list the PODs that are currently running using `kubectl get pods`. Delete the PODs that are running for this deployment. How can your recognize them?

```
$ kubectl get pods
$ kubectl delete pod <pod-id>
```

Check again with `kubectl get pods`. What happened? Please explain.

Now let's go one step up. Show the replicaset(s) belonging to this deployment and delete them

```
$ kubectl get pods
$ kubectl get rs
$ kubectl delete rs <rs-id>
$ sleep 5
$ kubectl get pods
$ kubectl get rs
```

Please explain what you see.

Now let's go one more step further up

```
$ kubectl get pods
$ kubectl get rs
$ kubectl get deployment
$ kubectl delete deployment <deployment-id>
$ kubectl get deployment
$ kubectl get rs
$ kubectl get pods
```

What happened? Please explain what you see.

Create a deployment yaml file called `nginxc-deployment.yaml` with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: nginxc-blue
  name: nginxc-blue
spec:
  replicas: 1
  selector:
    matchLabels:
      run: nginxc-blue
  template:
    metadata:
      labels:
        run: nginxc-blue
    spec:
      containers:
      - image: pamvdam/nginxc:v1.1
        name: nginxc-blue
        env:
          - name: COLOR
            value: "blue"
        ports:
        - containerPort: 80
```

Create this deployment using `kubectl create -f`

```
$ kubectl create -f nginxc-deployment.yaml
```

Verify the deployment

```
$ kubectl get deployment
$ kubectl get rs
$ kubectl get pods -o wide
```

Verify if we can reach the webservice inside the container using the POD IP

```
$ kubectl get pods -o wide
$ curl http://<POD-IP>
```

Now try to change the color of the container by changing the ENV var inside the POD

You can dump the yaml using `kubectl get deployment` or use `kubectl edit deployment` to change the value of the ENV VAR inside the POD's spec.

```
...
    spec:
      containers:
      - image: pamvdam/nginxc:v1.1
        name: nginxc-blue
        env:
          - name: COLOR
            value: "purple"
...
```

Supported colors are:

- black
- orange
- yellow
- blue
- purple
- green
- red

After applying the changed manifest check to see if the color of the deployed PODs has changed.

3.2 Introduction to kubectl

Edit Objects

```
$ kubectl get deployments/nginxc-blue
```

In the output of the last command, part of the output looked like this:

```
Containers:
   nginxc-blue:
    Image:        nginxc-blue:latest
    Port:         80/TCP
    Host Port:    0/TCP
```

Let’s change the port from port 80 to 81:

```
$ kubectl edit deployments/nginxc-blue
```

Change the line `- containerPort` and save it.

Execute again:

```
$ kubectl describe deployments/nginx-deployment
```

After this exercise, change the port back to 80 again.

Another nice feature is the possibility to update the image:

```
$ kubectl set image deployments/nginxc-blue nginxc-blue=nginx:1.15.10-alpine-perl
```

And

```
$ kubectl describe deployments/nginxc-blue
```

Confirms that the image is changed.

Service Object

So we have pods running nginx. Every pod with a different IP. You can imagine that this is not necessarely what you want, because if an POD dies, the deployment will automaticaly create a new one with a different ip. This is the problem a service will solve.

Create a service, using the `expose` command:

```
$ kubectl expose deployments/nginxc-blue
```

And verify:

```
$ kubectl get services
```

A new cluster IP is created:

```
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes          ClusterIP   10.96.0.1       <none>        443/TCP   21h
nginxc-blue   ClusterIP   10.102.243.80   <none>        80/TCP    59s
```

More information:

```
$ kubectl describe services/nginxc-blue
```

Output should look similar to:

```
Name:              nginxc-blue
Namespace:         default
Labels:            app=nginx1
Annotations:       <none>
Selector:          app=nginx1
Type:              ClusterIP
IP:                10.102.243.80
Port:              <unset>  80/TCP
TargetPort:        81/TCP
Endpoints:         192.168.174.154:80,192.168.174.155:80,192.168.174.156:80
Session Affinity:  None
Events:            <none>
```

The endpoints can also be received via:

```
$ kubectl get ep nginxc-blue
```

Test the webserver, using `curl` on the cluster IP address

Explain

As we did before with another object, the service configuration can be displayed in YAML format:

```
$ kubectl get service/nginxc-blue -o yaml
```

With an output similar to:

```
apiVersion: v1
kind: Service
```

```
metadata:
  creationTimestamp: "2019-04-02T14:11:58Z"
  labels:
    app: nginx1
  name: nginxc-blue
  namespace: default
  resourceVersion: "32790"
  selfLink: /api/v1/namespaces/default/services/nginxc-blue
  uid: 4712f0bc-5551-11e9-aff2-000c2988328a
spec:
  clusterIP: 10.102.10.163
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx1
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

The `explain` command can be very helpfull to understand more about this configuration.

Execute the following commands:

```
$ kubectl explain service

$ kubectl explain service.spec

$ kubectl explain service.spec.clusterIP
```

4 Kubernetes PODs

4.1 Hostnames & sub-domains

In this section we will research the use of hostnames and sub-domains in PODs

Please create a file called `bb1-pod.yaml` with the following content

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostname: busybox-1
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
```

```
$ kubectl create -f bb1-pod.yaml
```

Mind that we use `hostname` and `subdomain` properties in the `POD spec`.

Enter the `busybox` container using the following command:

```
$ kubectl exec -it busybox1 -- sh
```

Once inside the container type the following commands:

```
/ # hostname
```

The command will display `hostname` as specified in the `POD spec`.

Execute the following command inside the `busybox-1` container

```
/ # ping busybox-1
/ # ping busybox-1.default-subdomain
/ # ping busybox-1.default-subdomain.default.svc.cluster.local
/ # nslookup busybox-1
/ # nslookup busybox-1.default-subdomain
```

Try to explain the results you see.

There is definately no DNS record. But where is the hostname and the FQDN registered? You can check the `/etc/hosts` file inside the `busybox-1` container.

```
/ # cat /etc/hosts
```

The hostname and FQDN are registered in the `/etc/hosts` file but NOT in K8S DNS.

Exit the container and return to the Linux prompt:

```
/ # exit
```

Create a new YAML file, called bb2-pod.yaml with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    name: busybox
```

Mind that we use a different hostname but the samen subdomain. Also note that this POD has the same label `name: busybox`

Create the POD using the following command:

```
$ kubectl create -f bb2-pod.yaml
```

Now create a new YAML file for the service called `pod-svc.yaml` with the following contents:

```
apiVersion: v1
kind: Service
metadata:
  name: bb-svc
spec:
  selector:
    name: busybox
  ports:
  - name: foo # Dummy port, busybox does not do ports
    port: 1234
    targetPort: 1234
```

Create the service using the following command:

```
$ kubectl create -f pod-svc.yaml
```

Mind that the service manages PODs that will have the label `name: busybox`. So in our case both PODs will be addressed by the service.

Now logon to the second busybox container:

```
$ kubectl exec -it busybox2 -- sh
```

Inside the container execute the following commands:

```
/ # ping busybox-1
/ # ping busybox-1.default-subdomain
/ # ping busybox-1.default-subdomain.default.svc.cluster.local
/ # ping busybox-2
/ # ping busybox-2.default-subdomain
/ # ping busybox-2.default-subdomain.default.svc.cluster.local
/ # nslookup busybox-1
/ # nslookup busybox-1.default-subdomain
/ # nslookup busybox-2
/ # nslookup busybox-2.default-subdomain
/ # nslookup default-subdomain
/ # nslookup bb-svc
/ # ping bb-svc
```

Try to explain the results.

Do the IP addresses that the hostnames resolve to familiar? From what IP pool have they been drawn? From what IP pool does the servicename come from?

There now certainly is a DNS record, but take a look at the service info

```
$ kubectl get svc bb-svc
```

The takeaway here is that the service does generate a hostname / FQDN that has an IP drawn from the cluster IP pool. The hostname and/or hostname + subdomain as specified in the POD will get an IP address from the POD IP pool. Only the service name is persistent.

4.2 Probes

Probes can be use inside a project to test the `liveness` or `readiness` of a POD. Starting from Kubernetes 16.x there is also a so called `startupProbe`. In this lab we will discuss the `livenessProbe` and `readinessProbe`

LivenessProbe

With a `livenessProbe` one can test if a POD is alive. The cluster will periodically call this probe to see if's responding with the expected information.

Please create a deployment file called `nginxc-purple.yaml` with the following contents:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: nginxc-purple
  name: nginxc-purple
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginxc-purple
  template:
    metadata:
      labels:
        run: nginxc-purple
    spec:
      containers:
      - image: pamvdam/nginxc:v1.1
        name: nginxc-purple
        env:
          - name: COLOR
            value: "purple"
        ports:
        - containerPort: 80
```

Create the deployment

```
$ kubectl create -f nginxc-purple.yaml
```

Create a yaml manifest called `purple-svc.yml` to expose the deployment inside the cluster with the following content

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginxc-purple
  name: nginxc-purple
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginxc-purple
  type: NodePort
```

Create the service

```
$ kubectl create -f purple-svc.yml
```

Verify if the deployment succeeded

```
$ kubectl get deployment nginxc-purple
$ kubectl get pods -o wide
```

Verify if you can access the service using the clusterIP

```
$ kubectl get svc
$ curl http://<clusterIP>
```

Repeat the last command multiple times, you should see that you will hit different PODs (check reported hostname) on subsequent invokes of the `curl` command.

Now let's put a Liveness Probe in the POD:

Alter the `nginxc-purple.yaml` using your favorite editor such that it looks like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: nginxc-purple
  name: nginxc-purple
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginxc-purple
  template:
    metadata:
      labels:
        run: nginxc-purple
    spec:
      containers:
      - image: pamvdam/nginxc:v1.1
        name: nginxc-purple
        env:
          - name: COLOR
            value: "purple"
        ports:
        - containerPort: 80
        livenessProbe:
          exec:
            command:
            - cat
            - /tmp/healthy
          initialDelaySeconds: 5
          periodSeconds: 5
```

Apply the altered manifest to the deployment

```
$ kubectl apply -f nginxc-purple.yaml
$ kubectl get deployment nginxc-purple
```

Verify what happens with the PODs. (The `-l run=nginc-purple` option is a fine example of how to use labels to filter the output of a kubectl query command.)

```
$ kubectl get pods -l run=nginxc-purple
```

After a while you will see restarts. Let's troubleshoot a little bit further into the matter;

```
$ kubectl describe pods -l run=nginxc-purple
```

What do you see? Please explain.

Try to fix a POD one at a time by opening a shell in it and 'fix' the problem.

```
$ kubectl exec -it <pod-id> -- sh
```

Hint: a oneline to 'fix' the issue in one of the PODs is:

```
$ kubectl exec -it <pod-id> -- touch /tmp/healthy
```

If you do this one POD at a time, you can see the restart stop if you observe the pods with `kubectl get pods -l run=nginxc-purple`

For extra credit; construct a deployment or POD with more than one container add a failing `livenessProbe` to one container and see determine what actually gets restarted; the container or the POD?

ReadinessProbes

The `readinessProbe` can be used inside a POD to regulate wether or not the POD is ready to accept network traffic directed to it. A POD with a failed `readinessProbe` will not get network traffic (re-)directed at it.

In order to avoid any side effects of our initial purple deployment, we will delete it:

```
$ kubectl delete deployment nginxc-purple
$ kubectl get deployment
```

Let’s alter our `nginxc-purple.yaml` file again. Now to insert a `readinessProbe` into it. Please remove the `livenessProbe` section and alter `nginxc-purple.yaml` so it looks like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: nginxc-purple
  name: nginxc-purple
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginxc-purple
  template:
    metadata:
      labels:
        run: nginxc-purple
    spec:
      containers:
      - image: pamvdam/nginxc:v1.1
        name: nginxc-purple
        env:
          - name: COLOR
            value: "purple"
        ports:
        - containerPort: 80
        readinessProbe:
          exec:
            command:
            - cat
            - /tmp/ready
          initialDelaySeconds: 5
          periodSeconds: 5
```

Apply the altered manifest to the deployment

```
$ kubectl apply -f nginxc-purple.yaml
$ kubectl get deployment nginxc-purple
```

As the service for the deployment is still there (check with `kubectl get svc`) try to see if you can response from the service on the clusterIP address using `curl`

```
$ kubectl get svc
$ curl http://<cluster ip>
```

None of the PODs will give a response. As the readynessProbe fails. Check with `kubectl describe pod`

```
$ kubectl describe pod <pod-id>
```

Now let’s activate one of these PODs to get it in Ready state.

```
$ kubectl exec -it nginxc-purple-<pod-id> -- touch /tmp/ready
```

Check if we can access the service from the POD(s)

```
$ kubectl get svc
$ curl http://<cluster ip>
```

’Fix’ the other PODs also. You will see which each POD you fix the `curl` could route you to a different POD

This concludes our lab about `probes`

Sidecar POD

The Sidecar pattern allows to extend or augment the functionality of a pre-existing container without changing it. A good use-case for a sidecar container is logging.

Create a new YAML file: nginx-slog.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-sidecar
  labels:
    run: pod-with-sidecar
spec:
  volumes:
  - name: shared-logs
    emptyDir: {}

  containers:

  - name: app-container
    image: alpine:latest
    command: ["/bin/sh"]
    args: ["-c", "while true; do date >> /var/log/index.html ; sleep 5;done"]

    volumeMounts:
    - name: shared-logs
      mountPath: /var/log

  - name: sidecar-container
    image: nginx:1.7.9
    ports:
      - containerPort: 80

    volumeMounts:
    - name: shared-logs
      mountPath: /usr/share/nginx/html
```

In this configuration, two containers are created, within one pod: a container which is just a simple virtual machine, generating some logs, and a container which is able to access the logs. Imagine a log parser running on the webserver...

Create the pod:

```
$ kubectl apply -f nginx-slog.yaml
```

Verify that they are up-and-running:

```
$ kubectl get pods/pod-with-sidecar
```

```
NAME               READY   STATUS    RESTARTS   AGE
pod-with-sidecar   2/2     Running   0          20s
```

More details:

```
$ kubectl describe pods/pod-with-sidecar
```

Please mention the single IP address, the containers are sharing this address, and because port 80 is exposed on the NGINX container, you'll be able to access this webserver on that specific address.

Init POD

The name init is coming from init systems, that brings order in processes started at boot.

Init containers are special containers that will be executed before the other containers. The other containers will only start if the init container is started succesfully.

Create a YAML file: init-git.yaml, with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: init-git
  labels:
    app: init
spec:
  containers:
  - name: app-container
    image: alpine
    command: ['sh', '-c', 'echo The app is running!','sleep 30']
  initContainers:
  - name: init-clone-repo
    image: alpine/git:latest
    args:
      - clone
      - --single-branch
      - --
      - https://linvirt@bitbuket.org/linvirt/dockerfiles.git
      - /tmp/repo
    volumeMounts:
    - name: git-repo
      mountPath: /tmp/repo
  volumes:
  - name: git-repo
    hostPath:
      path: /tmp/repo
```

Create the POD

```
$ kubectl create -f init-git.yaml
```

Verify:

```
$ kubectl get pods
```

As you can see there is a problem, after 30 seconds:

```
 NAME       READY   STATUS                  RESTARTS   AGE
init-git    0/1     Init:CrashLoopBackOff   3          31s
```

The status CrashloopBackOff means: An Init Container has failed repeatedly.

More information:

```
$ kubectl logs init-git
```

```
Error from server (BadRequest):
container "app-container" in pod "init-git" is waiting to start: PodInitializing
```

And even more information about the status can be found with:

```
$ kubectl describe pods/init-git
```

The init-clone-repo container won't start. Let's have a look why:

```
$ kubectl logs pod/init-git -c init-clone-repo
```

```
Cloning into '/tmp/repo/dockerfiles'...
Could not resolve hostname bitbuket.org: Try again
```

```
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Change the host from bitbuket.org to bitbucket.org and try again!

5 Kubernetes Deployments

In this section we will learn:

- How to create Deployments
- How to communicate with PODs managed by Deployments
- How to upgrade PODs in Deployments

5.1 Creating Deployments

Please create a file called `pyco_deployment.yaml` with the following content

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: pyco
  name: pyco
spec:
  replicas: 3
  selector:
    matchLabels:
      run: pyco
  template:
    metadata:
      labels:
        run: pyco
    spec:
      containers:
      - env:
        - name: CCOLOR
          value: purple
        image: pamvdam/pyco2:v1.2
        imagePullPolicy: Always
        name: pyco
        ports:
        - containerPort: 8080
          protocol: TCP
```

Create the `Deployment` using the following command:

```
$ kubectl create -f pyco_deployment.yaml
```

Check if the PODs are coming up fine with:

```
$ kubectl get pods -o wide
```

Try to connect to the PODs using their POD IPs as displayed by the previously executed command:

```
$ curl <POD IP ADDR>:8080
```

You should see output that looks similar to the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Python Color Container App</title>
</head>
<body>
        <div style="text-align:center;">
        <img width="300" src="/static/cc.svg">

        <H2>Welcome to the purple container</H1>
        <P>
        This container is running on <I>pyco-85b4488547-4n7bl</I>
        <P>
        You are visitor nr <I>1</I> connecting from: <I>192.168.0.0</I>
        <P>
        <H3>Pyco2 version: 1.0</H1>
</body>
```

You can repeat this for the other 2 PODs too.

If you look at the POD names that were displayed by the `kubectl get pods` command you can see they consist of 3 parts

- Part 1: deployment name
- Part 2: identifier of the replicaset generated by the deployment
- Part 3: identifier of the PODs generated by the replicaset

Use

- `kubectl get` ...
- `kubectl describe` ...

To get more information about the deployment and the generated replicasets. Verify that the PODs are indeed generated by the replicaset and that the replicaset is generated by the deployment.

Now let’s try to delete a POD. Observe what happens and try to explain.

Try to delete the replicaset. Observe what happends with the PODs and the Replicaset and try to explain.

Lastly delete the deployment itself and observe what happens with the PODs, the Replicaset and the Deployment itself.

5.2 Communicating with PODs managed by Deployments

Communicating with PODs inside a deployment is done by creating a service for it. We can do this using the `kubectl expose` command or writing the proper yaml specification for it.

First let's create our deployment again:

```
$ kubectl create -f deployment1.yaml
```

Observe with `kubectl get pods` that the PODs are coming up nicely.

Now we are going to create service for our deployment by crafting some yaml. Please create a file called `pyco-svc.yml` with the following content:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: pyco
  name: pyco
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    run: pico
```

Create the service:

```
$ kubectl apply -f pyco-svc.yml
```

Observe the service for the cluster IP

```
$ kubectl get svc pyco
```

Connect to the service using the Cluster IP e.g:

```
$ curl <svc_cluster_ip_addr>:8080
```

Does it work? If not diagnose with `kubectl get ep pyco` what could be the issue.

Correct the issue in the yaml of the service if needed and re-apply the service yaml.

```
apiVersion: v1
```

```
kind: Service
metadata:
  labels:
    run: pyco
  name: pyco
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    run: pyco
```

```
$ kubectl apply -f pyco-svc.yml
```

Connect to the service using the Cluster IP e.g:

```
$ curl <svc_cluster_ip_addr>:8080
```

5.3 Upgrading PODs in Deployments

Upgrading of container images in PODs shows the true power of deployments

There are multiple ways to upgrade container images in PODs. The disruptive one is being done with `kubectl replace`

Please alter your `pyco_deployment.yaml` and alter the image of the container in the POD to: `pamvdam/pyco2:v1.0`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: pyco
  name: pyco
spec:
  replicas: 3
  selector:
    matchLabels:
      run: pyco
  template:
    metadata:
      labels:
        run: pyco
    spec:
      containers:
      - env:
        - name: CCOLOR
          value: purple
        image: pamvdam/pyco2:v1.0
        imagePullPolicy: Always
        name: pyco
        ports:
        - containerPort: 8080
          protocol: TCP
```

Check the status of the PODs.

```
$ kubectl get pods
```

As this is an disruptive upgrade/downgrade all PODs will be deleted before beging replaced. Normally we will not use `kubectl replace` for an upgrade/update.

We are now going to apply an rolling update. Please execute the following command:

```
$ kubectl set image deployment pyco pyco=pamvdam/pyco2:v1.3 --record
```

```
$ kubectl get pods -l run=pyco
```

Verify with curl and `kubectl describe pod` if the correct image has been used.

With the `--record` option we can see what cause the change to the deployment.

```
$ kubectl rollout history deployment pyco
```

This also works with `kubectl apply` by the way, in contrast to popular belief:

```
$ kubectl apply -f pyco_deployment.yaml --record
```

```
$ kubectl rollout history deployment pyco
```

Now let's do something less useful, update the image to pamvdam/pyco2:v0.1

```
$ kubectl set image deployment pyco pyco=pamvdam/pyco2:v0.1 --record
```

Verify with `kubectl get pods`

The PODs will not be updated, due to what reason? Please check with `kubectl describe deployment pyco`

Roll back the upgrade:

```
$ kubectl rollout undo deployment pyco
```

You can also roll back to a specific version of the rollout according to the `kubectl rollout history` output. E.g:

```
$ kubectl rollout undo deployment pyco --to-revision=2
```

```
$ kubectl get pod <pod-id>
$ kubectl describe pod <pod-id>
```

This concludes the labs on `Deployments`

6 Kubernetes DaemonSets

In this section we will learn:

- How to create DaemonSets
- How to communicate with PODs managed by DaemonSets
- How to upgrade PODs in DaemonSets

6.1 Creating DaemonSets

Please create a file called `daemonset1.yaml` with the following content

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    run: nginxc-green
  name: nginxc-green
spec:
  selector:
    matchLabels:
      run: nginxc-green
  template:
    metadata:
      labels:
        run: nginxc-green
    spec:
      containers:
      - image: pamvdam/nginxc:v1.1
        name: nginxc-green
        env:
          - name: COLOR
            value: "green"
        ports:
        - containerPort: 80
```

Create the `DaemonSet` using the following command:

```
$ kubectl create -f daemonset1.yaml
```

Verify that it exists

```
$ kubectl get daemonset nginxc-green
```

Verify if the PODs come up well:

```
$ kubectl get pods -l run=nginxc-green
```

Try to connect to the `green-container` PODs using the POD IPs you can obtain by executing:

```
$ kubectl get pods -l run=nginxc-green -o wide
$ curl <pod-ip>
```

6.2 Communicating with PODs managed by DaemonSets

Let's try to get more persistent access to our `DaemonSet`. In the previous subsection we already were able to connect to the `DaemonSet's` PODs by connecting to their `POD IPs`.

The obvious way to expose a `DaemonSet` is to use a `service`. Let's try to do that.

```
$ kubectl expose daemonset nginxc-green
```

To your surprise you will see that Kubernetes does not support the `expose` on `DaemonSets`. However we can still make a service for the PODs managed by the `DaemonSet`. But first we will use another technique to connect to the PODs in our `DaemonSet`.

Please copy your `daemonset1.yaml` to a new file called `daemonset2.yaml`. Edit `daemonset2.yaml` so that the `DaemonSet` will look like this:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    run: nginxc-green
  name: nginxc-green
spec:
  selector:
    matchLabels:
      run: nginxc-green
  template:
    metadata:
      labels:
        run: nginxc-green
    spec:
      containers:
```

```
      - image: pamvdam/nginxc:v1.1
        name: nginxc-green
        env:
          - name: COLOR
            value: "green"
        ports:
        - containerPort: 80
          hostPort: 80
```

Alter the `DaemonSet` by appling the new YAML:

```
$ kubectl apply -f daemonset2.yaml
```

You will now find PODs of your `DaemonSet` listening on port `80` of the nodes the PODs got scheduled on. (Check with `kubectl get pods -o wide`)

```
$ kubectl get pods -o wide
$ curl http://<nodeip>
```

You can access the PODs in the daemonsets already by targeting the node IP address of your node and using port 80! What you actually do here is bypassing the service mechanism.

```
$ curl http://st03node02
```

Repeat it a few times and you will see you will only target 1 single POD. The POD that is actually running on the node you target.

```
$ curl http://st03node03
```

You will see this command will have you target the POD and only the POD that is running on the addressed node. Ergo, no loadbalancing here. And that's correct as we are not using a service.

We can use this pattern to address a POD running on a specific NODE.

When do need to have loadbalancing among the PODs, we need to craft a service manually like hereunder:

Now let's craft a service for our `DaemonSet`

Create a file called `ds-svc.yaml` with the following content:

```
kind: Service
apiVersion: v1
metadata:
  name: ds-svc
spec:
  selector:
    run: nginxc-green
  ports:
    - port: 80
```

This piece of `YAML` creates a service that will manage PODs with the label `run=nginxc-green` and the traffic for the PODs on port 80 will be directed to port 80 in the container of the PODs.

Create the service for managing the PODs in the `DaemonSet` (here the PODs generated with label `run=nginxc-green`. Verify if you can connect to the service by listing the ClusterIP and using `curl`

```
$ kubectl get svc ds-svc
$ curl http://<cluster-ip>
```

6.3 Upgrading PODs in DaemonSets

We can do an rolling upgrade of the PODs in a `DaemonSet`. Execute the following commands to upgrade the image in the PODs to from nginxc:v1.1 to nginxc:v3.1

```
$ kubectl set image ds/nginxc-green nginxc-green=pamvdam/nginxc:v3.1
```

You can watch the rollout being performaned by using:

```
$ kubectl rollout status ds nginxc-green
```

To be sure, check the output of a curl to the service deployed for you `DaemonSet`. Check if the displayed version is indeed the updated one.

```
$ kubectl get svc
$ curl http://<cluster-ip>
```

This concludes the labs on `DaemonSets`

7 Kubernetes storage

MTDIR

Creating storage shared between containers in PODS - MTDIR volumes

- Create the following POD with one container named my-container running alpine

```
kind: Pod
apiVersion: v1
metadata:
  name: simple-volume-pod
spec:
  volumes:
  - name: simple-vol
    emptyDir: {}
  containers:
  - name: my-container
    volumeMounts:
      - name: simple-vol
        mountPath: /var/simple
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "while true; do date >> /var/simple/index.html; sleep 2; done"]
```

- Check with kubectl exec inside the PODs container to see what happens on the /var/simple filesystem.
- Extend the POD to include one more container. This container should be called my-web and running the nginx:1.7.1 image. Ensure that this PODs mounts the volume provided by emptyDir on /usr/local/nginx/html.
- Check using http (e.g. curl, links) from outside if you can see the index.html file.

Persistent Volumes

The NFS server for this LAB is your own masternode e.g.: `st01node01`

Each student has a directory on it e.g: /mnt/pvs/st01pvol01, /mnt/pvs/st02vol01 etc.

Make sure that on all your worker nodes you install the `nfs-client` package, otherwise your PODs will not be able to NFS mount the PVs

```
$ apt install nfs-client -y
```

- Create a Persistent Volume of a NFS mount using a manifest like this:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv01
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: st01node01
    path: "/mnt/pvs/st01pvol01"
```

Change the path to your student location yourself

- Create a Persistent Volume Claim for use in the POD

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 10Mi
```

- Create the following deployment that will excercise the PVC

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: nfs-busybox
  name: nfs-busybox
spec:
  replicas: 2
  selector:
    matchLabels:
      run: nfs-busybox
  template:
    metadata:
      labels:
        run: nfs-busybox
    spec:
      containers:
      - image: busybox
        name: nfs-busybox
        command:
          - sh
          - -c
          - 'while true; do date > /mnt/index.html; hostname >> /mnt/index.html; \
            sleep $(($RANDOM % 5 + 5)); done'
        imagePullPolicy: Always
        volumeMounts:
          # name must match the volume name below
          - name: nfs
            mountPath: "/mnt"
      volumes:
      - name: nfs
        persistentVolumeClaim:
          claimName: nfs
```

- Check the state of the deployment. Why is it failing?
- Create a PV that will make the deployment succeeed

8

Kubernetes ConfigMaps

Creating ConfigMaps

ConfigMaps are used to separate config from the Container Image. In the following lab excercises you will learn how to create ConfigMaps and how to use them in your PODs.

As taught in the course-ware there are various ways to construct a `configMap`

- Using `kubectl create configmap` from literal values
- Using `kubectl create configmap` from files or even complete directories
- Using a `configMap` YAML manifest

Goal of this excercise is to create a POD with the `nginxc` image where we can choose the color of the container deployment by using a `configMap`

First; let's construct a `configMap` from the commandline:

```
$ kubectl create configmap myconfig --from-literal=color=red
```

Verify the `configMap`

```
$ kubectl get cm myconfig -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: color-container
  labels:
    run: color-container
spec:
  containers:
  - name: color-container
    image: pamvdam/nginxc:v1.1
    env:
      - name: COLOR
        valueFrom:
          configMapKeyRef:
            name: myconfig
            key: color
```

Expose the POD with a service type NodePort so you can verify the color of the container from outside

```
$ kubectl expose pod color-container --type NodePort --port 80
$ kubectl get svc color-container
```

Mark the NodePort and run an browser on your local machine to verify if you can see the red colored container on one of the IP addresses or hostnames of your nodes. (Do not forget to include the port nr)

Now let's try to change the `configMap` so we can get a different color container.

- Retrieve the yaml from the configMap `myconfig`
- Alter the value of the key color from red to `yellow`
- and re-`apply` the configMap.

Finally verify if the container did change color.

Most probably the color did not change it’s color as there’s no way for the container to know that the ENV var has changed. Restart the POD to see the POD change it’s color

Using configMaps as volumes

This lab excercise will use a program that is used in real-life and that we will adapt for using configMaps (and later secrets). Actually we will change nothing in the application code, all changes are done on the K8S level.

The application we will use for this LAB is a so called questionnaire program. It has been in use by us to collect presence lists but also to do course evaluations and even for assessments.

The application is written in `Python` using the `Flask` webapp framework. The app has two sections, one admin section for reviewing the submitted results which is password protected from prying eyes and the other is handling the questionnaires and storing the results. Both questionnaires and filled out questionnaires are stored in JSON format.

To get a view of the application let's create a simple deployment for it using the following YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: q10rv2
  name: q10rv2
spec:
  replicas: 3
  selector:
    matchLabels:
      run: q10rv2
  template:
    metadata:
      labels:
        run: q10rv2
    spec:
      containers:
      - image: pamvdam/q10rv2:sf
        name: q10rv2
        ports:
        - containerPort: 5000
```

Create the deployment using `kubectl create -f ..`

Verify the deployment

```
$ kubectl get deployment q10rv2
$ kubectl get pods -o wide
```

Expose the application using a NodePort service

```
$ kubectl expose deployment q10rv2 --type NodePort
$ kubectl get svc
```

Try to access the webapp from the browser on your workstation, you can find the application running on one of the urls: <hostname>:<nodeport> e.g. st00node03.itgildelab.net:31020.

- On st00node03.itgildelab.net:31020 you will find the admin page, you will need to logon with a username and password. (default username: admin, password: secret)

- on st00node03.itgildelab.net:31020/texteditor you will find a questionnaire about favorite texteditors. You can fill it in

Goal of this excercise is to put a new questionnaire in a configMap and deploy it together with the Q10RV2 app image. This way we can extend the questionnaire app without having to build and submit a new image.

The JSON for a new questionnaire looks like this:

```
{
    "title": "Your Favorite Salesforce Mascot",
    "questions": [
        {
            "type": "string",
            "label": "Your Name",
            "required": true
        },
        {
            "type": "checkbox",
            "label": "Which of the following Salesforce mascots do you know?",
            "required": true,
            "options": [
                "Astro(logical)",
                "Einstein",
                "Appy",
                "Cloudy",
                "Codey",
                "Hootie",
                "None",
                "+Other"
            ]
        },
        {
            "type": "radio",
            "label": "What mascot do you identify yourself with?",
            "required": true,
            "options": [
                "Astro(logical)",
                "Einstein",
                "Appy",
                "Cloudy",
                "Codey",
                "Hootie",
                "None",
                "+Other"
            ]
        },
        {
            "type": "text",
            "label": "Why",
            "help": "Kindly restrict your argumentation to 100 words or less"
        }
    ]
}
```

Please create a file with the above JSON called `mascot.json`. Then create a `configMap` from this fine piece of JSON using the following command:

```
$ kubectl create configmap qset --from-file=mascot.json
```

This will create a configMap called `qset` with the contents of file `mascot.json`. Verify with `kubectl get cm qset` the contents.

```
$ kubectl get cm qset
```

Now that we have the configMap we need to have this config emerge inside the container. The best method for this is using the configMap as a file so by mounting it as a volume in the container.

The questionnaire's JSON files are located in /app/questionnaires. So there's where the configMap needs to be mounted.

Please extend the deployment yaml of the Q10RV2 app with the following YAML for mounting of the configMap

```
       volumeMounts:
          - name: questionnaires-vol
            mountPath: /app/questionnaires
      volumes:
        - name: questionnaires-vol
          configMap:
            name: qset
```

The complete deployment file look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: q10rv2
  name: q10rv2
spec:
  replicas: 3
  selector:
    matchLabels:
      run: q10rv2
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: q10rv2
    spec:
      containers:
      - image: pamvdam/q10rv2:sf
        name: q10rv2
        env:
        volumeMounts:
          - name: questionnaires-vol
            mountPath: /app/questionnaires
      volumes:
```

```
    - name: questionnaires-vol
      configMap:
        name: qset
```

- (Re-)deploy the Q10RV2 app using this extended YAML
- Verify if you can get the new questionnaire by veriyfing the URL: (e.g.) `st03node01.itgildelab.net/mascot`

So it appears the UX/UI group is not really pleased with the horizontal nature of the radioboxes. They would like us to change the mascot dictionaire so they will be alligned in a vertical fashion.

- Update the configMap like below and have the new configMap used by the deployment

```
{
    "title": "Your Favorite Salesforce Mascot",
    "questions": [
        {
            "type": "string",
            "label": "Your Name",
            "required": true
        },
        {
            "type": "checkbox",
            "label": "Which of the following Salesforce mascots do you know?",
            "required": true,
            "orientation:"vertical",
            "options": [
                "Astro(logical)",
                "Einstein",
                "Appy",
                "Cloudy",
                "Codey",
                "Hootie",
                "None",
                "+Other"
            ]
        },
        {
            "type": "radio",
            "label": "What mascot do you identify yourself with?",
            "required": true,
            "orientation":"vertical",
            "options": [
                "Astro(logical)",
                "Einstein",
                "Appy",
                "Cloudy",
                "Codey",
                "Hootie",
                "None",
                "+Other"
            ]
        },
        {
            "type": "text",
            "label": "Why",
            "help": "Kindly restrict your argumentation to 100 words or less"
        }
    ]
}
```

- Play with with another questionnaire or even try to add multiple questionnaires to the configMap
- Use `kubectl rollout restart deployment q10rv2` to have the deployment act upon the updated `configMap`

9 Kubernetes Secrets

Creating Secrets

In this section your will learn how to create secrets and how to use them in your PODs

We will use the Q10RV2 webapp again. In this case we will configure the password using a K8S secret for the admin part of the webapp.

Using `kubectl create secret` set the value for the key `qpassword` to `mysecret`. This is at least different fromt he default password for `Q10RV2`

```
$ kubectl create secret generic qsecret --from-literal=qpassword='mysecret'
```

Verify the secret

```
$ kubectl get secret qsecret -o yaml
```

- The Q10RV2 app reads the QPWD ENV variable to set the password for the admin part.
- It's therefore best to have the secret manifest itself as an ENV variabele inside the container/POD.

Let's integrate the following secret using YAML to the deployment YAML of Q10RV2

```
       env:
          - name: QPWD
            valueFrom:
              secretKeyRef:
                name: qsecret
                key: qpassword
```

Ergo, the ENV VAR QPWD will be filled with the value derived from the key `qpassword` which can be found in the secret called `qpassword`

The complete YAML of the Q10RV2 deployement augmented with the configMap for the questionnaire and the secret for the admin/password part will look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: q10rv2
  name: q10rv2
spec:
  replicas: 3
  selector:
    matchLabels:
      run: q10rv2
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: q10rv2
    spec:
      containers:
      - image: pamvdam/q10rv2:sf
        name: q10rv2
        env:
          - name: QPWD
            valueFrom:
              secretKeyRef:
                name: qsecret
                key: qpassword
        volumeMounts:
          - name: questionnaires-vol
            mountPath: /app/questionnaires
      volumes:
        - name: questionnaires-vol
          configMap:
            name: qset
```

- (Re-)deploy the new deployment for the Q10RV2 webapp
- Verify it the application still works

- Verify if you can access the admin part with the new password `mysecret`

Docker Registry in K8S with configMaps and Secrets

This part will be a project you will jointly execute with your trainer.