Introduction to Kubernetes

ITGilde

"Samen Sterker, Beter, Slimmer en Leuker!"

December 3, 2019

Contents

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 --tty ubuntu
```

The standard output of the container is now connected to the a tty, displayed in your terminal. Press Ctrl + c 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

$ 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 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 maintened 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/local/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 perstistant.
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:

- Un-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 it's 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 it's 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.

```
$ sudo 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 /verb|/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"

sudo 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 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 hour home-directory

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

sudo 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/manifests
```

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: .....
Kubernetes Network Subnet: .....

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:

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

$ docker system prune
```

In Debian, 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:

```
$ 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 -
```

Add the repository:

```
$ 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
```

Finally, install Kubernetes:

```
$ sudo apt install kubeadm kubectl kubelet
```

Configure the master and define the subnet:

```
$ 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 user data. Execute these commands:

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

And it displays the command to execute on the nodes, to join the cluster.

```
$ 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: Node

Login into the node virtual machine, and install Docker:

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

$ sudo apt upgrade

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

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

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

#  Resync index files for the repositories
$ sudo apt update

# Install Docker
$ sudo apt install docker-ce docker-ce-cli containerd.io

# Add the user to the group docker
$ sudo usermod -aG docker student
```

Execute the same procedure as on the master, but instead of executing `kubeadm init`, execute the saved command.

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

Leave the node and login into the master. Get 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   v1.14.0
node1.test    NotReady   <none>   10m   v1.14.0
```

This caused by the fact that we didn't install a Container Network Interface (CNI). In our setup we're going to use Flannel
Install the plugin:

```
kubectl apply -f \
https://raw.githubusercontent.com/coreos/flannel/master/\
Documentation/kube-flannel.yml
```

After that, your cluster becomes ready. The output should be similar too:

```
NAME          STATUS    ROLES    AGE   VERSION
master        Ready     master   68m   v1.14.0
node1.test    Ready     <none>   16m   v1.14.0
```

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 \
  --generator=run-pod/v1 --replicas=1
```

Note: the parameter `--generator=run-pod/v1` makes it somewhat less deprecated and you won't see a warning stating that you should use `kubectl create` instead.
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 nginx-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 nginx-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

$ kubecrtl 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

In the output of the last command, this was a part of the output:

```
Containers:
   nginx:
    Image:        nginx:latest
    Port:         80/TCP
    Host Port:    0/TCP
```

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

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

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/nginx-deployment nginx=1.15.10-alpine-perl
```

And

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

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 one node dies, the deployment will automaticaly create a new one with a different ip. This is the problem a service solves.
Create a service, using the `expose` command:

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

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
nginx-deployment   ClusterIP   10.102.243.80   <none>        80/TCP    59s
```

More information:

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

Output should look similar to:

```
Name:              nginx-deployment
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 nginx-deployment
```

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/nginx-deployment -o yaml
```

With an output similar to:

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2019-04-02T14:11:58Z"
  labels:
    app: nginx1
  name: nginx-deployment
  namespace: default
  resourceVersion: "32790"
  selfLink: /api/v1/namespaces/default/services/nginx-deployment
  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

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

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
      - /home/student/dockerfiles
    volumeMounts:
    - name: git-repo
      mountPath: /home/student/dockerfiles
  volumes:
  - name: git-repo
    hostPath:
      path: /home/student/dockerfiles
```

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 '/home/student/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!