# Docker Lab part #1

### Introduction to Docker images, containers and registry

Open a Putty session to your Docker-Kubernetes environment:

127.0.0.1:<port>

You can see the port in the NAT / Port Forwarding properties of the VM

Settings > Network > Advanced > Port Forwarding

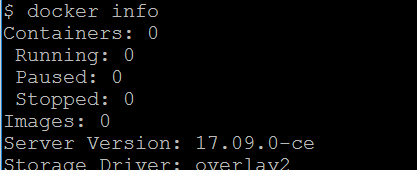
Log on as:

username = docker

password = tcuser

Verify that Docker machine is up and running

docker info



If it isn’t you can run “minikube start” from a DOS prompt, not inside the Putty session

Now let’s download and run a simple Docker image

docker run hello-world

It cannot find it locally so it goes to find it to the registry, ie Docker Hub

Let’s run it again and see how long it takes to run

docker run hello-world

How does that compare with booting up a virtual machine? That’s one of the reasons why containers have become so popular.

Let’s see now what has been created

docker image ls

docker container ls

Uhmm! The container is not around, but the image still is. Let’s remove the image before we do more complex scenarios.

docker image rm hello-world

But it is giving an error that the container is not there. Check again.

docker container ls

We cannot see it because it is no longer active. But it is still there. You can see also inactive containers like this:

docker container ls -a

We did run it twice or more, right? So you see all times you did run it. You could remove them individually or you can remove “all” inactive containers at once like this

docker container prune

Try deleting the image now

docker image rm hello-world

docker image ls

Gone!!

### Building “our own” containers

Building a container requires a Dockerfile that provides instructions on how to build it. Let’s retrieve the Dockerfile and its related files from the “Docker-lesson” Github repo. We can use the “git” tool which is included with the Minikube installation:

git clone https://github.com/cermegno/Docker-lesson.git

Let’s verify the files have been downloaded

ls

cd Docker-lesson

ls

cd Sample-container

Examine all 3 files with cat, ex:

cat requirements.txt

cat app.py

cat Dockerfile

You have seen a “requirements.txt” before . It contains Python dependencies. Comment with the instructor the syntax of the Dockerfile. Does it make sense?

Now let’s build the container.

IMPORTANT: Don't forget the “.” at the end of the command to specify the directory that contains the build files

docker build -t myfirstcontainer **.**

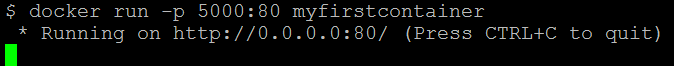
docker image ls

There are two new images:

* It has retrieved the official Python image python:2.7-slim we specified in the “Dockerfile”
* and it has also created the image we requested

Since the image is created we can spin up a container

docker run -p 5000:80 myfirstcontainer



I can’t wait for this!! Open a web browser to see the web page:

<http://127.0.0.1:5000>

Can’t you see it? No, really?

Let’s troubleshoot it

Open another putty session to Minikube and use “curl” command

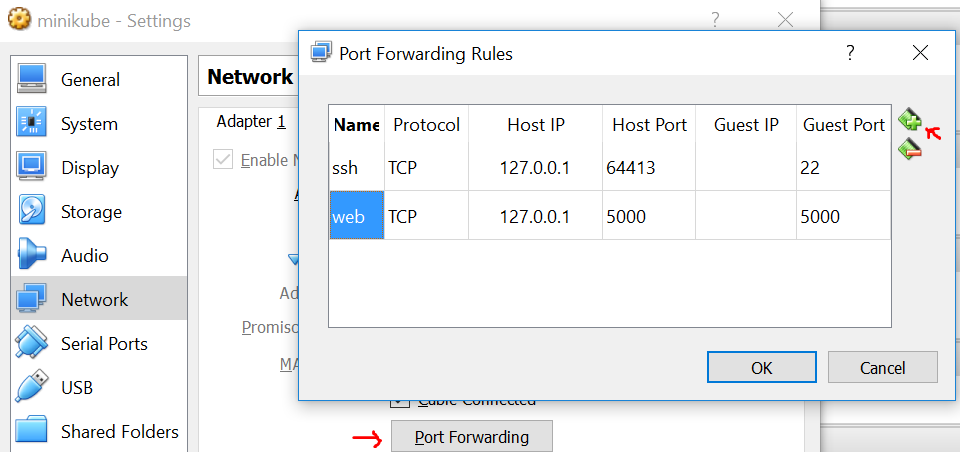
curl 127.0.0.1:5000

Alright, the container is serving the web page but we cannot see it from our “host” laptop. What do you think we need to do?

You’re right! We need to set port forwarding in Virtualbox. Go to:

Settings > Network > Advanced > Port Forwarding

And set it up like this:



Do you remember the saying about “*pets and chickens*” in Platform 3 apps? We look after pets but not after chickens. So generally we don’t log in to our containers, but sometimes it is handy to know how in order to do troubleshooting.

The “docker exec” command allow us to run a command inside a container. In particular we can run a shell to gain access to it.

Firstly you are going to need to get the container ID for the container. Run "docker ps", make note of the first column and then try this:

docker ps

docker exec -it <container ID> /bin/bash

Once you are in, you can see where you are, see the file we added to the container and verify the version of Python it is running:

pwd

ls

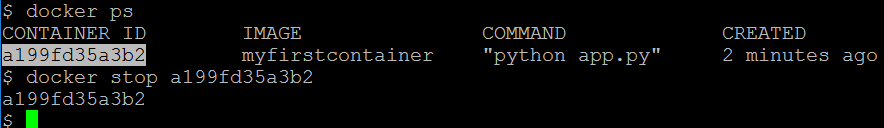
python -V

exit

Let’s stop the container

docker stop <CONTAINER\_ID>

Like this



### Exploring Docker registry

Now let’s play with the Docker Hub registry. Go ahead and create your own account. It is quick and free!

<https://hub.docker.com/>

Now we can log in to the registry from Minikube

docker login

This is the syntax to push your image into a new repository

docker push <your\_account\_name>/<image\_name>[:tag]

If you make use of tags it will display the different versions. For example check this out in my own repository:

<https://hub.docker.com/r/cermegno/webapp1234/tags/>

To tag your image before you push it use this

docker tag image username/repository:tag

Push your image and verify in the web browser that the image has been uploaded

Now we will pull it back, but before doing that, let’s remove it:

docker image rm myfirstcontainer

Arghh! Again! There is a container running

docker container stop <Container ID>

docker container prune

OK, now we can pull it back!! You have two options

* Pull it and run it later

docker pull <your\_account\_name>/<image\_name>[:tag]

docker image ls

* or if you want to run it now, you can pull it and run it at once as we did with the hello world example

docker run -p 5000:80 <your\_account\_name>/<image\_name>[:tag]

You could share your containers with your team if you were working on a project together

# Docker Lab part #2

### Multi-tier app

Running a single container has been fun, but how do we give our web app a backend database? Let’s create a new container by pulling down the official Redis image, called “redis”

docker run --name myredis -d redis

docker ps

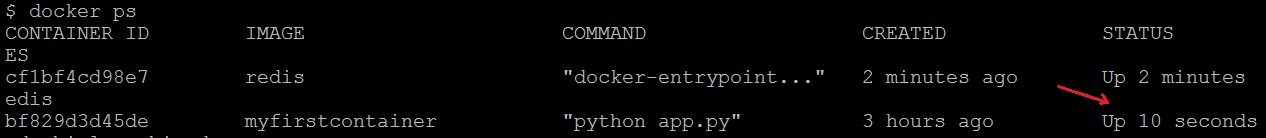
docker image ls

With the “--name” flag we create a well-defined name for the container instead of the funny auto-generated ones

Let’s restart our Flask container and see if this time we have a connection to Redis

docker container restart bf829d3d45de

docker ps



Open a web browser:

<http://127.0.0.1:5000>

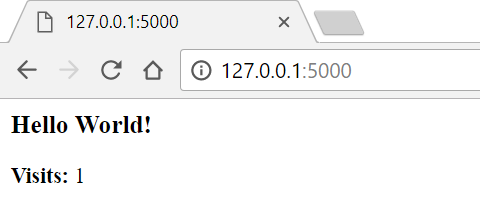
Still no luck, right?

By default they don’t see each other. You have to “*link*” them. Stop it again, but now run it with the link flag:

docker stop bf829d3d45de

docker run -p 5000:80 --link myredis:redis -d myfirstcontainer

Try the browser again:



WOW!!! Awesome!

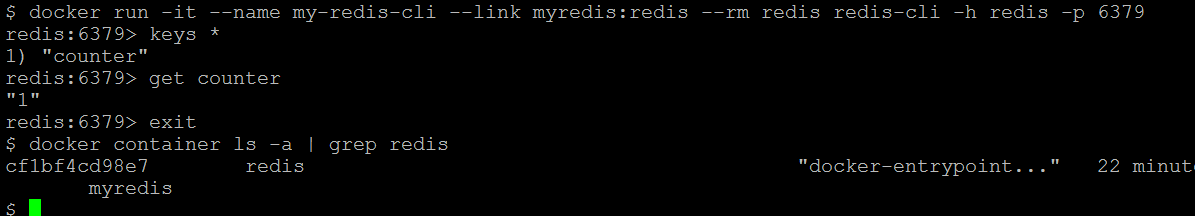
What do you need to do now to access this Redis database externally?

Try to access it from your “host” OS using a tool like Redis Desktop Manager. Can you see the counter?

This command is a handy way of getting a redis-cli connection into your Redis database to help you troubleshoot

docker run -it --name my-redis-cli --link myredis:redis --rm redis redis-cli -h redis -p 6379

The “--rm” flag removes the container automatically when you exit



Before moving onto the next lab stop and remove the Redis and Flask containers

# Kubernetes Lab

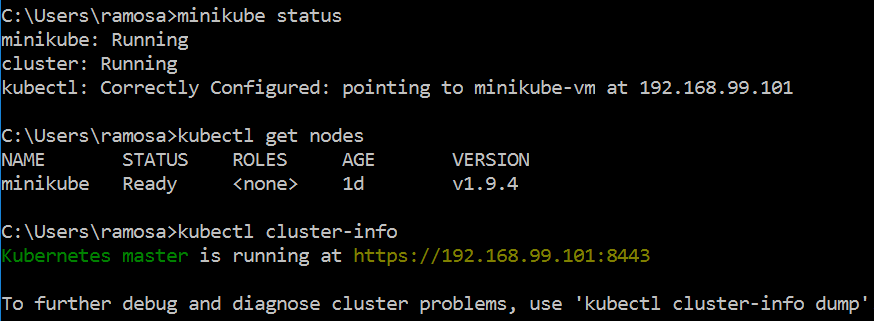
In a CMD prompt you can check the status of your cluster:

minikube status

kubectl get nodes

kubectl cluster-info

The first command is Minikube specific but the last two commands apply to any Kubernetes cluster



Two more interesting commands

kubectl get events

kubectl config view

The second command displays a YAML config file with the details of the cluster.

Now let’s launch the Kubernetes dashboard. It has very similar look-and-feel to the real Kubernetes dashboard

minikube dashboard

Take your time to explore the dashboard. You can identify many of the concepts we have explain during the workshop. As you can see there are no deployments or pods yet.

Let’s do our first deployment. We can use the Docker image you pushed to the Docker Hub public repository as this is the default location Minikube will look at. The syntax is as follows

kubectl run webapp1234 --image=<repository>/<image\_name>:tag --port=80

Where “*repository*” is your Docker Hub account name. If you want to use an image with a tag other than the default “*latest*” specify it after the colon. The port must match the port you specified in the Dockerfile. In my case it was “*80*”.

Let’s go!

kubectl run webapp1234 --image=cermegno/webapp1234 --port=80

Now quickly switch over to the dashboard in your web browser and see the deployment appearing and the pods being created. The deployment eventually say “Pods 1/1". You can get more information with the “kubectl” utility as well

kubectl get deployment

kubectl get pod

In the Putty session you can see the containers by typing

docker container ls

or

docker ps

Notice the "PORTS" column is empty. Kuberbenetes uses its own port forwarding mechanisms instead of Docker’s.

OK, so now that our deployment is complete, can we access our web app? You can try to access it within the Putty session

curl 127.0.0.1:80

but it fails. However the app is running

docker exec -t <Container\_ID> ps -ef

Notice "python app.py"

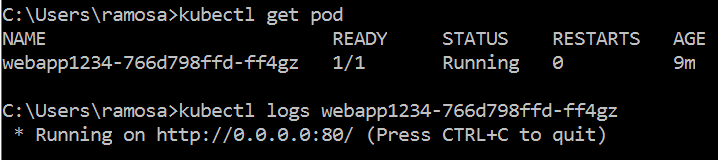
As we have covered in the theory slides, in order to access it we will need to create a “*service*”. We will do that a bit later. Now let's check a few more things. For example, we can take a look at the logs. First get the POD name from here:

kubectl get pod

Now you can retrieve the logs

kubectl logs <pod\_name>

The output is a clear sign the app is running!



And you can get details about a resource like a pod:

kubectl describe pod <pod\_name>

### Scaling our deployment

Kubernetes uses the concept of pods to wrap containers. If you want to scale your deployment you can increase the amount of pods

Back in the dashboard go to deployments, expand the 3-dots drop-down and select “Scale”, and set it to 2 pods. Check the result

kubectl get pod

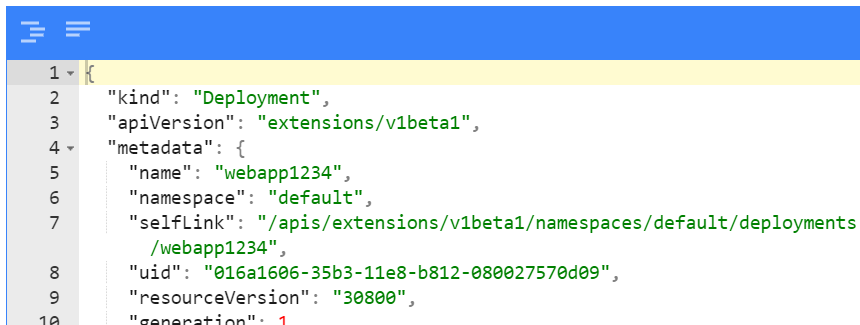
It can also be scaled from the command line

kubectl scale --replicas=3 deployment/webapp1234

In the Putty session you can see the containers. Pay attention to the “CREATED” time.

docker ps

Did you notice there that for every container we requested Kubernetes has added another container at the same time? That container is often referred to as a sidecar, it is part of the same pod and it provides certain functions to the main container

In the 3-dots drop-down you noticed another option “View/Edit YAML”. 

The configuration of most objects can be displayed as YAML format. You could save the YAML, edit it and use it for a fully custom deployment, or you could “*update*” it in-place inside the dashboard

### Kubernetes Services

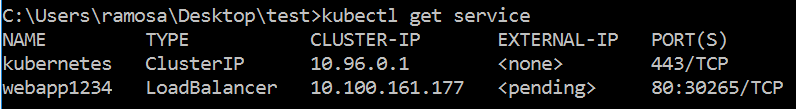
The pods are running but in order to access them externally we need to “expose a service”. You must speficy what type of service you want. There are currently 3 types available. The most common one is “LoadBalancer” but there are 2 more basic ones: NodePort and ClusterIP

kubectl expose deployment webapp1234 --type=LoadBalancer

kubectl get service

kubectl describe service webapp1234

In my case it has created a mapping from port 80 to port 30265. We don’t have a Load Balancer in our Minikube environment, that’s why it shows as “*pending*”



If we want to find out the URL to access it, you can type

minikube service webapp1234 --url

open the app with curl within the Putty session



Alternatively you can ask Kubernetes to open it for you in your default web browser

minikube service webapp1234

# Bonus Points – Optional content

By default Minikube downloads the image from an external repository because of this policy

"imagePullPolicy": "Always"

You can see this if you display the YAML format of a deployment

For example we still had the “myfirstcontainer” image in the private registry inside Minikube.

kubectl delete deployment webapp1234

kubectl run webapp1234 --image=myfirstcontainer --port=80

kubectl get deployment

But unless you change the policy the deployment won’t succeed. You could try deploying from YAML instead.

kubectl get deployment webapp1234 -o yaml > newdeploy.yaml

Now open “newdeploy.yaml” file with Notepad++ and set the policy to “IfNotPresent”

"imagePullPolicy": "IfNotPresent"

Now delete the failed deployment and deploy with the “yaml” file

kubectl delete deployment webapp1234

kubectl create -f newdeploy.yaml

# Optional – tiering the environment down

Not sure why you would like to do such a thing. But if you want to get rid of Minikube you can easily do that

minikube stop

minikube delete