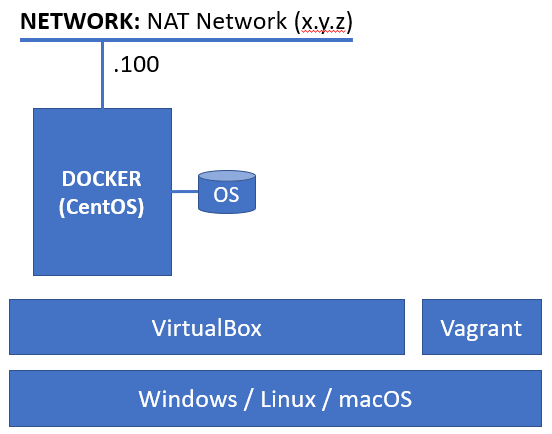
# Practice M2: Introduction to Docker

For this practice we will use an infrastructure like this:



All exercises that follow assume that we are working in an environment with both **VirtualBox** and **Vagrant** installed

For the VM we will use a **CentOS 8.x** minimal based **Vagrant** box

Please note, that some commands end with this **\** symbol. This is because the command is long and does not fit on one line. Written this way makes it more readable. When you type in such commands, omit the **\** symbol and continue with the rest of the command

## Part 1: Docker installation

**References**

<https://docs.docker.com/engine/install/centos/>

<https://docs.docker.com/engine/install/linux-postinstall/>

**Commands**

**version, system info, container run**

### Preparation

Spin up a **CentOS** based virtual machine. You can do it either by importing a template or using **Vagrant**

Let us go with the second approach

Open a terminal session and navigate to an empty folder of your choice

Invoke the initialization command to create an empty environment

**vagrant init --minimal shekeriev/centos-8-minimal**

Now open the **Vagrant** file with a text editor of your choice

If you are working in a graphical environment of **Windows**, **Linux**, or **macOS**, you can install the **VSCode** free editor by **Microsoft** and add one of available **Vagrant** plugins

Now, that we are in, let us add one block, between the **config.vm.box** and **end**, to set some port forwarding and change the memory size

**config.vm.network "forwarded\_port", guest: 8080, host: 8080, auto\_correct: true**

**config.vm.provider :virtualbox do |vb|**

**vb.customize ["modifyvm", :id, "--memory", "2048"]**

**end**

Save and close the file

Return to the terminal session and start the environment

**vagrant up**

And then, once the machine is up and running, establish **ssh** session to it

**vagrant ssh**

### Remove old installations

This is not directly applicable to our case, as we start fresh

Should you have old installation of Docker, it is recommended to remove it first

**sudo dnf remove docker docker-client docker-client-latest \**

**docker-common docker-latest docker-latest-logrotate \**

**docker-logrotate docker-selinux docker-engine-selinux docker-engine**

### Installation preparation

First, we must make sure that we have the appropriate repository registered

**sudo dnf config-manager \**

**--add-repo https://download.docker.com/linux/centos/docker-ce.repo**

If using an earlier version of **CentOS 8.x** install the **containerd.io** package manually and before the **docker-ce** package. If on **Fedora** **Server** or on latest **CentOS 8.x** skip this step

**sudo dnf install https://download.docker.com/linux/centos/7/x86\_64/stable/Packages\**

**/containerd.io-1.2.13-3.1.el7.x86\_64.rpm**

### Docker installation

Now, we are ready to install Docker itself. Execute the following:

**sudo dnf install docker-ce docker-ce-cli**

If on **Fedora** install also the **containerd.io** package

Let us start the **docker** daemon and check its status

**sudo systemctl start docker**

**systemctl status docker**

If the service is working normally, we can receive more information about the installed versions and the current configuration with

**sudo docker version**

**sudo docker system info**

And let us start our very first container

**sudo docker container run hello-world**

Or this one:

**sudo docker container run shekeriev/welcome-dob:2021**

### Additional settings

To be able to work with **docker** without the need to use always **sudo**, we must add our user to the **docker** group

**sudo usermod -aG docker $USER**

To apply the change, we must log off and then log on back again

We can check by executing any of the previous **docker** commands, for example

**docker container run hello-world**

As a last step, we can mark the **docker** service for automatic start on boot

**sudo systemctl enable docker**

One last, but important step. If running on earliest versions of **CentOS 8.x**, we must take care for **docker** connectivity to Internet. If we are using another Linux distribution, or earlier version of CentOS, we may continue and safely ignore the next few commands

If we have a firewall running, we must check if the **docker0** interface is zoned properly with this command:

**firewall-cmd --get-active-zones**

If it is in a separate zone named **docker** then we are good and there is nothing extra required on our side

If the **docker0** interface is not in the zone specified above, we must add the it to the trusted zone of the firewall

**sudo firewall-cmd --add-interface docker0 --zone trusted --permanent**

And then reload the rules

**sudo firewall-cmd --reload**

Now, we should not have troubles accessing Internet resources from within containers

For more post-installation settings like remote connectivity, storage options, etc. visit the following address

<https://docs.docker.com/engine/install/linux-postinstall/>

### Create a complete Vagrantfile

Let us try to write an all-in-one **Vagrantfile** that can be used to spin up a VM with **docker** installed and configured

Create a new folder and switch to it

Open a new **Vagrantfile** for editing

You can copy and paste what we have in our first **Vagrantfile** and extend it or start clean

Any way, you should type the following text

Vagrant.configure("2") do |config|

  config.vm.define "dobdocker" do |dobdocker|

    dobdocker.vm.box="shekeriev/centos-8-minimal"

    dobdocker.vm.hostname = "docker.dob.lab"

    dobdocker.vm.network "private\_network", ip: "192.168.99.100"

    dobdocker.vm.network "forwarded\_port", guest: 8080, host: 8080, auto\_correct: true

    dobdocker.vm.provision "shell", path: "dobdocker.sh"

    dobdocker.vm.provider :virtualbox do |vb|

        vb.customize ["modifyvm", :id, "--memory", "2048"]

    end

  end

end

Save and close the file

Now create a new **dobdocker.sh** file

Type the following (these are the steps we did manually)

#!/bin/bash

echo "\* Add hosts ..."

echo "192.168.99.100 docker.dob.lab docker" >> /etc/hosts

echo "\* Add Docker repository ..."

dnf config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

echo "\* Add the missing dependency ..."

dnf install -y https://download.docker.com/linux/centos/7/x86\_64/stable/Packages/containerd.io-1.2.13-3.1.el7.x86\_64.rpm

echo "\* Install Docker ..."

dnf install -y docker-ce docker-ce-cli

echo "\* Enable and start Docker ..."

systemctl enable docker

systemctl start docker

echo "\* Firewall - open port 8080 ..."

firewall-cmd --add-port=8080/tcp --permanent

echo "\* Add vagrant user to docker group ..."

usermod -aG docker vagrant

Save and close the file

Give it a try with

**vagrant up**

If everything goes according to the plan, then as a result there should be a working **docker** machine

## Part 2: Working with images and containers

**References**

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

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

**Commands**

**search**, **image** **pull**, **image** **ls**, **image** **rm**, **container** **run**, **container** **ls**, **container** **rm**, **container** **create**, **container** **start, container attach, container stop, container prune, container export**, **image import**, **image save**, **image load**

### Working with containers

We could search for all images that contain **ubuntu** in their name

**docker search ubuntu**

The first result in the list is with the biggest rating and it is marked as an official image

We can download it locally with

**docker image pull ubuntu**

In fact, the above command is equivalent to this one

**docker image pull ubuntu:latest**

As a result of the execution of the above two commands, we will have the latest version downloaded locally

Should we want a particular version, we can execute

**docker image pull ubuntu:18.04**

List with the available images can be obtained with the execution of the following command

**docker image ls**

Let us start a container out of the **ubuntu:18.04** image

**docker container run -it ubuntu:18.04**

Once, we are in the container, we can execute the following sequence of commands

**root@35ac9218a880:/# ls**

**bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var**

**root@35ac9218a880:/# ps ax**

**PID TTY STAT TIME COMMAND**

**1 pts/0 Ss 0:00 /bin/bash**

**15 pts/0 R+ 0:00 ps ax**

**root@35ac9218a880:/#**

Should we want to temporary exit the container without stopping it, we can do it by pressing and holding the **Ctrl** key, then pressing **P** and then **Q** and finally releasing the **Ctrl** key

Alternatively, we can press **Ctrl+P** and then **Ctrl+Q**

Now, we can ask for the list of all working containers

**docker container ls**

The result should look similar to

**CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**

**35ac9218a880 ubuntu:18.04 "/bin/bash" 8 minutes ago Up 8 minutes cocky\_fermat**

Should we want to go back to the container, we can execute

**docker container attach 35ac9218a880**

Where the **35ac9218a880** sequence is the container ID (the first column of the **docker container ls** command)

Instead it, we can use the value from the last column - the name of the container (**cocky\_fermat**)

We could terminate the container by executing the **exit** command

**root@35ac9218a880:/# exit**

In order to see the list of all containers including the stopped ones, we can execute

**docker container ls -a**

We can create container without starting it with

**docker container create -it --name ubuntu-20 ubuntu /bin/bash**

Then, we can start it by executing the following command

**docker container start -ai ubuntu-20**

Let us exit the container without stopping it. We can use the **Ctrl+P** and **Ctrl+Q** key combination

Then, we can stop it with

**docker container stop ubuntu-20**

Should we want to stop all running containers (currently we do not have any), we can execute the following

**docker container stop $(docker container ls -q)**

Particular stopped container or list of containers, we can remove with

**docker container rm ubuntu-20**

All stopped containers can be deleted with

**docker container prune**

We can remove one or more images with

**docker image rm ubuntu**

### Archive and transfer containers

Let us start a container out of the **Alpine** (**alpine**) image

**docker container run -it --name my-alpine alpine**

Then, we can execute the following commands **ls**, **echo**, **cat**, and **exit**, as shown bellow

**/ # ls**

**bin dev etc home lib media mnt proc root run sbin srv sys tmp usr var**

**/ # echo 'Hello from Alpine container!' > readme.txt**

**/ # cat readme.txt**

**Hello from Alpine container!**

**/ # ls**

**bin etc lib mnt readme.txt run srv tmp var**

**dev home media proc root sbin sys usr**

**/ # exit**

Now the container is stopped

We can save it as **tar** archive and then transfer it to another **docker** server and import it there

As we do not have another server, we will use our current

The export command is the following

**docker container export -o my-alpine.tar my-alpine**

The import process requires additional options because during the export the settings for **CMD** and **ENV** are lost

The actual import command may look like

**docker image import my-alpine.tar --change "CMD /bin/sh" my-new-alpine**

Now, we can start a container out of our new image and check if our custom file is there

**docker container run -it my-new-alpine**

**/ # ls**

**bin etc lib mnt readme.txt run srv tmp var**

**dev home media proc root sbin sys usr**

**/ # cat readme.txt**

**Hello from Alpine container!**

**/ # exit**

### Archive and transfer images

We can archive images directly and move them between **docker** servers

Let us pull a fresh image

**docker image pull busybox**

And archive it

**docker image save -o busybox.tar busybox**

In order to try the import from archive, first we must remove the existing image

**docker image rm busybox**

And the import the image from the archive

**docker image load -i busybox.tar**

We can create and run a container out of it

**docker container run -it busybox**

Explore it and then execute the **exit** command to stop it

## Part 3: Creating images

**Commands**

**image** **ls**, **container** **run**, **container** **ls**, **container** **commit, image** **build**, **image history**, **container prune**, **image rm**

### Create image from container

Let us create and run a container out of the **Ubuntu (ubuntu)** image first

**docker container run -it --name my-ubuntu ubuntu**

Apply the **Ctrl+P** and **Ctrl+Q** combination to close the connection without terminating the container

We can check if the container is still running, but let us apply a filter this time

**docker container ls -f name=my-ubuntu**

The process of image creation out of a container (even a running one) is very simple

The actual command is this

**docker container commit --author "SoftUni Student" my-ubuntu new-ubuntu**

And check if our new image is there

**docker image ls new-ubuntu**

Let us test if we can create and run a container based on it

This time we will launch the container with one additional option that will cause the container to be deleted automatically once stopped

**docker container run -it --rm new-ubuntu**

**root@fc0bc2b9b8ab:/# exit**

If we check the list of stopped containers, we will not find it there

**docker container ls -a**

### Create image with heredoc

Have you heard about heredoc?

It is technique for building simple multi-line text documents on the fly

It can be used to create simple **docker** images as well

Let us execute the following command

**docker image build -t alp-htop - << EOF**

**FROM alpine**

**RUN apk --no-cache add htop**

**EOF**

Now, we can launch a container based on our new image

**docker container run -it alp-htop**

**/ # htop**

**/ # exit**

### Create image from Dockerfile

We will create two identical from user point of view images using the **Dockerfile** technique

Let us create a folder **nginx-1** and change to it

**mkdir nginx-1 && cd nginx-1**

Now, open an empty **Dockerfile** for editing

**vi Dockerfile**

Enter the following text

**FROM ubuntu**

**LABEL maintainer="SoftUni Student"**

**RUN apt-get update**

**RUN apt-get install -y nginx**

**ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]**

**EXPOSE 80**

Save and close the file

Next step is to generate or build the image

Image building is done with

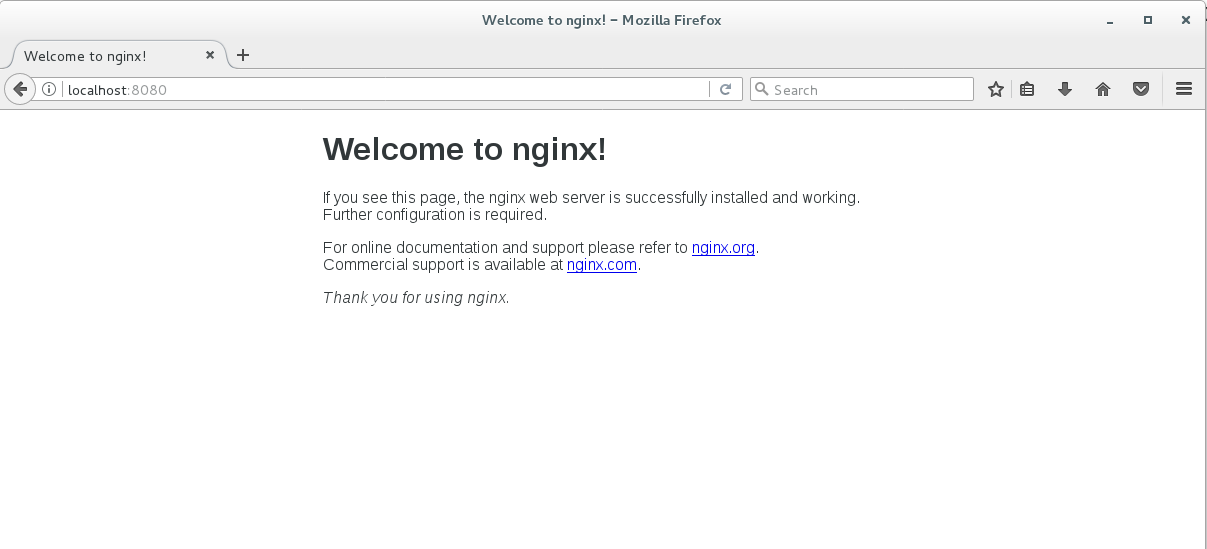
**docker image build -t nginx-1 .**

Once the image is built, spin up a container based on it with

**docker container run -d -p 8080:80 --name web-1 nginx-1**

Now, we can open a browser on the host and navigate to <http://localhost:8080>

If all went according to plan, we must see output similar to this



We can stop it with the following command

**docker container stop web-1**

Now, change to the upper folder

**cd ..**

Create another new folder and switch to it

**mkdir nginx-2 && cd nginx-2**

Open a new empty file for editing

**vi Dockerfile**

Enter the following text

**FROM ubuntu**

**LABEL maintainer="SoftUni Student"**

**RUN apt-get update && apt-get install -y nginx**

**ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]**

**EXPOSE 80**

Save and close the file

Build the new image with

**docker image build -t nginx-2 .**

Spin up a container out of the new image

**docker container run -d -p 8080:80 --name web-2 nginx-2**

If we open again a browser tab and visit <http://localhost:8080> we will see the same result, as expected

So, for us, the result is one and the same but for **docker** this is not the case

Let us inspect the first image with

**docker image history nginx-1**

**IMAGE CREATED CREATED BY SIZE**

**e232a6ba87f2 14 minutes ago /bin/sh -c #(nop) EXPOSE 80/tcp 0 B 53cee9ce5577 14 minutes ago /bin/sh -c #(nop) ENTRYPOINT ["/usr/sbin/... 0 B 9c4a4265ca72 14 minutes ago /bin/sh -c apt-get install -y nginx 56.5 MB e7bb8c797027 14 minutes ago /bin/sh -c apt-get update 38.5 MB d26f0a8d43ae 15 minutes ago /bin/sh -c #(nop) LABEL maintainer=SoftUn... 0 B 7b9b13f7b9c0 2 weeks ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0 B <missing> 2 weeks ago /bin/sh -c mkdir -p /run/systemd && echo '... 7 B <missing> 2 weeks ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\... 2.76 kB <missing> 2 weeks ago /bin/sh -c rm -rf /var/lib/apt/lists/\* 0 B <missing> 2 weeks ago /bin/sh -c set -xe && echo '#!/bin/sh' >... 745 B <missing> 2 weeks ago /bin/sh -c #(nop) ADD file:5aff8c59a707833... 118 MB**

And then the second one

**docker image history nginx-2**

**IMAGE CREATED CREATED BY SIZE**

**9540f6cd2f5f 2 minutes ago /bin/sh -c #(nop) EXPOSE 80/tcp 0 B b0e51c3cacc0 2 minutes ago /bin/sh -c #(nop) ENTRYPOINT ["/usr/sbin/... 0 B 372d0ba9f9a4 2 minutes ago /bin/sh -c apt-get update && apt-get insta... 95 MB 1fec01b2e84d 2 minutes ago /bin/sh -c #(nop) LABEL maintainer=SoftUn... 0 B 7b9b13f7b9c0 2 weeks ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0 B <missing> 2 weeks ago /bin/sh -c mkdir -p /run/systemd && echo '... 7 B <missing> 2 weeks ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\... 2.76 kB <missing> 2 weeks ago /bin/sh -c rm -rf /var/lib/apt/lists/\* 0 B <missing> 2 weeks ago /bin/sh -c set -xe && echo '#!/bin/sh' >... 745 B <missing> 2 weeks ago /bin/sh -c #(nop) ADD file:5aff8c59a707833... 118 MB**

As it can be seen, both images have different number of layers

The difference is because of the different number of **RUN** commands in the two **Dockerfile**s

### ENTRYPOINT and CMD

Let us see how **ENTRYPOINT** and **CMD** work together

Return to the upper folder

**cd ..**

Create a new one

**mkdir entry-cmd && cd entry-cmd**

Open an empty **Dockerfile** for editing

**vi Dockerfile**

Enter the following text

**FROM busybox**

**LABEL description="ENTRYPOINT vs CMD demo" maintainer="SoftUni Student"**

**ENTRYPOINT ["ping", "-c", "4"]**

**CMD ["www.softuni.bg"]**

Save and close the file

Build the image with

**docker image build -t pinger .**

Create and run a new container based on the image with

**docker container run --name p1 pinger**

And now run a second one but this time with modified command

**docker container run --name p2 pinger tuionui.com**

This way, by using both **ETRYPOINT** and **CMD** in their **exec** form, we got an image that when used for containers can be controlled by appending a parameter

### Clean up

We can return our system to a clean state by removing all containers and images

First, we can stop all running containers

**docker container stop $(docker container ls -q)**

Then, we can remove all stopped containers

**docker container prune**

And finally, we can remove all images

**docker image rm $(docker image ls -q)**