# Practice M2: Advanced Containerization Concepts

For this practice we will use an infrastructure like this:

A screenshot of a computer

AI-generated content may be incorrect.

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

For the VM we can use either a **AlmaLinux OS 9.x**-based or **Debian 12.x-**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, you can omit the* ***\*** *symbol and continue with the rest of the command*

## Part 1: Networking and Volumes

**Commands**

**network ls, network create, network inspect, network rm, volume ls, volume inspect, volume create, volume rm, container ls, container run, container create, container start, container stop, container exec, container attach, container inspect, container rm**

### Networks

Enter the **Docker** instance

**vagrant ssh**

#### Default Network

Let us check what networks we have defined:

**docker network ls**

We can ask for additional information about one of the existing networks. For example, for **bridge** one

**docker network inspect bridge**

We may see the IP address range, the reference to an existing network adapter on the system, etc.

Let’s start two containers based on the **alpine** image

**docker container run -dt --name alp1 alpine sleep 1d**

**docker container run -dt --name alp2 alpine sleep 1d**

Now, inspect the network

**docker network inspect bridge**

We can see the two containers with their IP addresses

Alternatively, we can execute the following command to limit the information to just the information we need

**docker network inspect bridge -f '{{range .Containers}}{{.Name}}, {{println .IPv4Address}}{{end}}'**

Now, open a session to the first container

**docker container exec -it alp1 sh**

And check its IP address

**ip a**

It is indeed what we saw earlier

Now, try to ping a public site

**ping -c 4 abv.bg**

It works. And what about pinging the other container (**alp2**) by its name?

**ping -c 4 alp2**

Hm, it doesn’t work. And by IP address?

**ping -c 4 172.17.0.3**

Ha, this one worked

So, our containers can communicate by address and not by name

We can try the other way around *(from* ***alp2*** *to* ***alp1****)*, but the result should be the same. So, we will skip it

Close the session to the container

**exit**

Stop and remove the containers

**docker container rm --force alp1 alp2**

#### Custom Network

Let’s create our own bridge network named **mybridge**

**docker network create -d bridge --subnet 10.0.0.0/24 mybridge**

And check if it appears in the list

**docker network ls**

We can ask for additional information about the network we just created with:

**docker network inspect mybridge**

Let’s create two containers and put them on our new network

We will base the containers again on the **alpine** image

**docker container run -dt --name co1 --network mybridge alpine sleep 1d**

**docker container run -dt --name co2 --network mybridge alpine sleep 1d**

Now, check our network

**docker network inspect mybridge**

We can see the two containers with their addresses

Now, we can enter the first container and test its network connectivity

**docker container exec -it co1 sh**

And check its IP address

**ip a**

It is indeed what we saw earlier

Now, try to ping a public site

**ping -c 4 abv.bg**

It works. And what about pinging the other container (**co2**) by its name?

**ping -c 4 co2**

Ha, it works. And by IP address?

**ping -c 4 10.0.0.3**

Ha, this one worked too

So, our containers can communicate by address and by name

This is quite a difference between the default bridge network and a custom made one

Close the session to the container

**exit**

Stop and remove the containers

**docker container rm --force co1 co2**

And remove the network

**docker network rm mybridge**

### Volumes

We can share data with containers in several ways. Let us explore them in the next exercises

#### On the Fly

First, we will create a container to which we will attach a volume (a local folder available on the **Docker** host)

Let’s first check if there are any volumes

**docker volume ls**

No, there aren’t any

Now, create the container

**docker container run -it -v /test-vol --name c1 ubuntu /bin/bash**

Now, we should have a **/test-vol** folder inside the container

**ls -l /**

Let us exit the container with **Ctrl+p** и **Ctrl+q**

Check to see if there are any changes in the volume area

**docker volume ls**

Hm, a volume appeared. It is a direct result of the container creation command above

Create a second container that will inherit the volume(s) from the first one

**docker container run -it --volumes-from c1 --name c2 ubuntu /bin/bash**

Again, exit the container with **Ctrl+p** и **Ctrl+q**

Attach back to the first one (**c1**)

**docker container attach c1**

And create a text file in the attached folder

**echo 'Hi from C1!' >> /test-vol/file.txt**

Exit again with **Ctrl+p** и **Ctrl+q** and attach back to the second one (**c2**)

**docker container attach c2**

Check the contents of the file

**cat /test-vol/file.txt**

And add some text to it

**echo 'C2 is here!' >> /test-vol/file.txt**

Check that the text is written to the file

**cat /test-vol/file.txt**

And exit (stop) the container (**c2**)

**exit**

Now, attach back to the first container (**c1**)

**docker container attach c1**

Check the contents of the file

**cat /test-vol/file.txt**

And exit the container (**c1**)

**exit**

Let us ensure that both containers are stopped

**docker container ls**

Now, we can start a third container (**c3**), that will inherit the volume from **c1**

**docker container run -it --volumes-from c1 --name c3 ubuntu /bin/bash**

Check the contents of the file

**cat /test-vol/file.txt**

And add a row to it

**echo 'C3 joined the party!' >> /test-vol/file.txt**

Exit the container with **Ctrl+p** и **Ctrl+q**

Then start again the **c1** container

**docker container start -i c1**

And check the contents of the file

**cat /test-vol/file.txt**

Exit again with **Ctrl+p** и **Ctrl+q**

Let’s list the running containers

**docker container ls**

We should have **c1** and **c3** listed as running

To check where the data is being stored in a persistent manner, execute

**docker container inspect c1 | grep -i source**

It should be the same for the **c3** container as well *(check if you like)*

We can now see the folder on our **Docker** host where the data is being stored *(substitute the* ***<volume-id>*** *identifier with the one you see on your screen, the whole identifier not a part of it)*

**sudo ls -l /var/lib/docker/volumes/<volume-id>/\_data**

Alternatively, we can ask for the volumes list with

**docker volume ls**

Detailed information about the volume, we can get by executing *(substitute the* ***<volume-id>*** *identifier with the one you see on your screen, the whole identifier not a part of it)*

**docker volume inspect <volume-id>**

Now, we can inspect the file content from the host

Change to the **root** user with

**sudo -i**

Navigate to the folder

**cd /var/lib/docker/volumes/<volume-id>/\_data**

Check the file contents

**cat file.txt**

Add a text to it

**echo 'Changed on the host' >> file.txt**

Then exit the **root** session

**exit**

And return to one of the containers (for example, **c1**)

**docker container attach c1**

Let’s check the file again

**cat /test-vol/file.txt**

Close the session to the container with the **Ctrl+p** and **Ctrl+q** sequence

As a last step we can stop both running containers

**docker container stop c1 c3**

And try to delete the volume

**docker volume rm <volume-id>**

Aha, we must delete the stopped containers first. So, let’s do it

**docker container rm c1 c2 c3**

Now, we can remove the volume as well *(repeat the command from the previous volume removal attempt)*

#### Attach a Prepopulated (Existing) Folder

Let us first create the folder, for example **web**

**mkdir web**

And add a simple **index.html** file in it

**echo '<h2>Hello from a Docker Volume</h2>' > web/index.html**

Now, we can start the container with the folder attached to it

**docker container run -d -p 8080:80 --name co-apache -v $(pwd)/web:/var/www/html php:8.0-apache**

Open a new browser tab on your host and enter the following URL: <http://192.168.99.100:8080>

If your **Docker** instance has another IP address, then use it instead

We can create a session to the container

**docker container exec -it co-apache bash**

And browse and explore the folder

**ls -al**

And the file

**cat index.html**

Let us change the file being inside the container

**echo '<br />Changed inside the container' >> index.html**

If we refresh the browser tab, we will see the difference

Exit the container session

**exit**

Let us change the file on the **Docker** host

**echo '<br /><br />Updated on the host' >> web/index.html**

If we refresh the browser tab, we will see the difference

Let’s check if there is a volume for the mounted folder

**docker volume ls**

It appears that there is not. This is an important difference, compared to the previous example

Let us stop the container

**docker container stop co-apache**

#### Dedicated Volume

In this exercise we will create the volume upfront with

**docker volume create vol-1 --label mode=prod**

And another one

**docker volume create vol-2**

Then we will list the volumes with

**docker volume ls**

Explore the volume’s details

**docker volume inspect vol-1**

The name of the volume is much simpler compared to what we saw earlier

We can filter the volumes list by label

**docker volume ls -f label=mode=prod**

And format the volumes list

**docker volume ls --format "{{.Name}}: {{.Driver}}: {{.Mountpoint}}"**

We can create a file directly into the volume

**echo '<h2>Volume created with <u>docker volume create</u></h2>' | \**

**sudo tee /var/lib/docker/volumes/vol-1/\_data/index.html**

Now, we are ready to start a new container with the volume attached

**docker container run -d -p 8080:80 --name co-apache1 -v vol-1:/var/www/html php:8.0-apache**

We can open a browser tab on the host and navigate to <http://192.168.99.100:8080> (adjust the address to match yours)

Stop the container

**docker container stop co-apache1**

#### Volume Containers

With the technique that we will explore now, we aim not only for data sharing, but for less space occupation as well

Let us create the container with

**docker container create -v /con-data --name con-store alpine /bin/true**

And check where the data is stored

**docker container inspect con-store | grep -i source**

Now, we can add a simple **readme.txt** file which will be available for other containers

**echo 'Read Me File in a Container Volume' | \**

**sudo tee /var/lib/docker/volumes/<volume-id>/\_data/readme.txt**

Let us start a new container that is connected to the volume

**docker container run -d --volumes-from con-store --name alp1 alpine sleep 1d**

If we open a session to the container, we will see that everything is according to the plan

**docker container exec -it alp1 /bin/sh**

And check the file

**cat /con-data/readme.txt**

Then close the session

**exit**

Finally, we can stop the container

**docker container stop alp1**

## Part 2: Custom Container Images

**Commands**

**image** **pull**, **image** **ls**, **image** **build**, **image push**, **image import**, **image save**, **image load**, **image history**, **image rm**, **container prune**, **login, container** **run**, **container** **ls**, **container** **commit, container export**

### 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 below

**/ # 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. Check with

**docker container ls -a**

We can start it again with

**docker container start -ai my-alpine**

Check if the file is there

**cat readme.txt**

Stop the container again

**exit**

Now, 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**

Check the size of the resulting file

**ls -lh**

Now, check the current list of locally available images

**docker image ls**

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**

In fact, we are importing it as an image which later will be used to spin new container

Check again the list of locally available images

**docker image ls**

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

Remember the **Busybox** image that we downloaded earlier?

Let’s 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**

Confirm that the image is gone

**docker image ls**

And the import the image from the archive

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

Now, confirm that we have the image, and we can use it

**docker image ls**

We can create and run a container out of it (we can even make the container disappear once stopped)

**docker container run -it --rm busybox**

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

Check that the container is absent as promised

**docker container ls -a**

Yes, it is gone 😊

### 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**

Let’s first check if the container is still running

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

And then 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**

Yes, we can, and it is working. Now, close the session

**root@fc0bc2b9b8ab:/# exit**

If we check the list of stopped containers, we will not find it there as we used the **--rm** option again

**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 by creating the **Dockerfile** on the fly

Let us execute the following command

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

**FROM alpine**

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

**EOF**

Do we have the new image listed?

**docker image ls**

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

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

Check that we can use the added package

**/ # htop**

And then close the session

**/ # 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 (use **nano** if you like)

**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

A screenshot of a computer

AI-generated content may be incorrect.

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 (use **nano** if you like)

**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

Now stop it with the following command

**docker container stop web-2**

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**

The output should be similar to

**IMAGE CREATED CREATED BY SIZE COMMENT**

**b16421e0b876 3 minutes ago /bin/sh -c #(nop) EXPOSE 80 0B**

**3d5c566eb9a8 3 minutes ago /bin/sh -c #(nop) ENTRYPOINT ["/usr/sbin/ng… 0B**

**96a3afaaa16e 3 minutes ago /bin/sh -c apt-get install -y nginx 59.2MB**

**a96d134e620d 4 minutes ago /bin/sh -c apt-get update 33.7MB**

**92309843847a 4 minutes ago /bin/sh -c #(nop) LABEL maintainer=SoftUni … 0B**

**54c9d81cbb44 7 days ago /bin/sh -c #(nop) CMD ["bash"] 0B**

**<missing> 7 days ago /bin/sh -c #(nop) ADD file:3ccf747d646089ed7… 72.8MB**

And then the second one

**docker image history nginx-2**

The output should be similar to

**IMAGE CREATED CREATED BY SIZE COMMENT**

**335de904496b About a minute ago /bin/sh -c #(nop) EXPOSE 80 0B**

**ff29f1a69e9c About a minute ago /bin/sh -c #(nop) ENTRYPOINT ["/usr/sbin/ng… 0B**

**ec1107f8aad4 About a minute ago /bin/sh -c apt-get update && apt-get install… 92.8MB**

**92309843847a 4 minutes ago /bin/sh -c #(nop) LABEL maintainer=SoftUni … 0B**

**54c9d81cbb44 7 days ago /bin/sh -c #(nop) CMD ["bash"] 0B**

**<missing> 7 days ago /bin/sh -c #(nop) ADD file:3ccf747d646089ed7… 72.8MB**

As it can be seen, both images have different number of layers (seven vs. six)

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

In any case, when we check their size

**docker image ls**

It is the same. So, can we go one step further? Sure, we can

Let’s go for a third run

Now, change to the upper folder

**cd ..**

Create another new folder and switch to it

**mkdir nginx-3 && cd nginx-3**

Open a new empty file for editing (use **nano** if you like)

**vi Dockerfile**

Enter the following text

**FROM ubuntu**

**LABEL maintainer="SoftUni Student"**

**RUN apt-get update && apt-get install -y nginx && rm -rf /var/lib/apt/lists/\***

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

**EXPOSE 80**

Save and close the file

Build the new image with

**docker image build -t nginx-3 .**

Spin up a container out of the new image

**docker container run -d -p 8080:80 --name web-3 nginx-3**

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

Now stop it with the following command

**docker container stop web-3**

Again, for us, the result is one and the same but for **Docker** this is not the case

Let us inspect this one as well

**docker image history nginx-3**

The output should be similar to

**IMAGE CREATED CREATED BY SIZE COMMENT**

**f9e661a365e7 2 minutes ago /bin/sh -c #(nop) EXPOSE 80 0B**

**b85ce1089c69 2 minutes ago /bin/sh -c #(nop) ENTRYPOINT ["/usr/sbin/ng… 0B**

**0b5861ca838f 2 minutes ago /bin/sh -c apt-get update && apt-get install… 92.8MB**

**92309843847a 19 minutes ago /bin/sh -c #(nop) LABEL maintainer=SoftUni … 0B**

**54c9d81cbb44 7 days ago /bin/sh -c #(nop) CMD ["bash"] 0B**

**<missing> 7 days ago /bin/sh -c #(nop) ADD file:3ccf747d646089ed7… 72.8MB**

Again, six layers as with the attempt no.2

Let’s check the size of the three images again

**docker image ls nginx-\***

Okay, we managed to save 30+ MB 😊

### Adding content during build

Now, change to the upper folder

**cd ..**

Create another new folder and switch to it

**mkdir -p nginx-4/web && cd nginx-4**

Create a custom index.html file

**echo 'It works! :)' > web/index.html**

Wait, where shall we copy this file (or the whole folder) in the image? Hm, let’s start again the last container (**web-3**)

**docker container start web-3**

Now, how can we enter? Perhaps, the attach command? Hm, no, it won’t work this time

We can execute a whole new process in the container with the following command

**docker container exec -it web-3 bash**

Check the configuration of the **NGINX** server with

**cat /etc/nginx/nginx.conf**

And then the file for the default web site

**cat /etc/nginx/sites-enabled/default**

Aha, the files are read from **/var/www/html**

So, we should place our content there

Close the session to the container

**exit**

Long way to check something simple. Is there another way?

Yes, there is. In fact, it is just a modification of what we did

Try this command

**docker container exec -it web-3 cat /etc/nginx/sites-enabled/default**

Okay, finally we can stop the container

**docker container stop web-3**

And continue with the building process

Now, open a new empty file for editing (use **nano** if you like)

**vi Dockerfile**

Enter the following text

**FROM ubuntu**

**LABEL maintainer="SoftUni Student"**

**RUN apt-get update && apt-get install -y nginx && rm -rf /var/lib/apt/lists/\***

**COPY web/ /var/www/html**

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

**EXPOSE 80**

Save and close the file

Build the new image with

**docker image build -t nginx-4 .**

Spin up a container out of the new image

**docker container run -d -p 8080:80 --name web-4 nginx-4**

If we open again a browser tab and visit <http://localhost:8080> we will see a different result 😊

Now stop it with the following command

**docker container stop web-4**

### Publishing an image

Let’s assume that we are glad by what we accomplished so far, and we want to publish our latest image (**nginx-4**)

First, we must make sure that we have access to a registry

The easiest option is to register at **Docker Hub** here: <https://hub.docker.com/>

Then, we must login and authenticate our client to the registry

**docker login**

Enter the username and the password you used for **Docker Hub**

Now, tag the image appropriately

**docker image tag nginx-4 <repository>/super-nginx:demo**

Check that it appears in the list

**docker image ls**

Pay attention to their size and ID. They are in fact two names or records for the same image

Now, push it to the registry

**docker image push <repository>/super-nginx:demo**

Return to the browser and check that you see your newly published image

We can safely delete the local copy both for the original image (**nginx-4**) and the published one

**docker image rm nginx-4**

**docker image rm <repository>/super-nginx:demo**

Check that they are gone

**docker image ls**

Now, start a container out of the published image

**docker container run -d -p 8080:80 <repository>/super-nginx:demo**

If we open again a browser tab and visit <http://localhost:8080> we will see it 😊

### 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 (or use **nano** if you like)

**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

### ENTRYPOINT and CMD extended

We can go even further

Let’s first remove all stopped containers

**docker container prune**

Now, let’s create a **pinger.sh** bash script with the following content

**#!/bin/bash**

**docker container run --rm pinger $1**

Save and close the file

Then set its executable permissions

**chmod +x pinger.sh**

And test it without arguments

**./pinger.sh**

Now, test it with an argument

**./pinger.sh abv.bg**

It acts the same, just the command is shorter

Let’s check the stopped containers

**docker container ls -a**

No traces of the last two executions

This way, we can create and use container-based utilities

## Part 3: Best Practices and Troubleshooting

### Best Practices

We will explore some of the best or recommended practices in the following lines

#### Provide Details via Labels

It is always a good idea to provide as much information as possible about an image

One way to do this is to utilize the **LABEL** instruction in the **Dockerfile**

Let’s create a folder structure for this experiment

**mkdir -p labels/html**

Enter the **labels** folder and create a **Dockerfile** with the following content

**FROM nginx**

**LABEL vendor="DevOps International"**

**LABEL version="1.0"**

**LABEL description="A sample web application that displays It Works"**

**COPY html/ /usr/share/nginx/html/**

In addition, create the new index page

**echo "It Works (but in container)" > html/index.html**

Now let’s build an image out of this

**docker build -t labels .**

Then, we can inspect the newly built image but filter just for the labels

**docker image inspect --format='{{json .Config.Labels}}' labels**

The labels are there. This is much better 😉

Should we want, we can restructure the **Dockerfile** and use another way of specifying multiple labels

One version of the file could look like this

**FROM nginx**

**LABEL vendor="DevOps International" version="1.0" description="A sample web application that displays It Works"**

**COPY html/ /usr/share/nginx/html/**

And another, like this

**FROM nginx**

**LABEL vendor="DevOps International" \**

**version="1.0" \**

**description="A sample web application that displays It Works"**

**COPY html/ /usr/share/nginx/html/**

No matter which one we choose, the result will be the same – all the information from the labels will be part of the metadata of the image

#### Push Readme with the Image

There is a nice tool called **pushrm** which can be used to publish descriptive information in an image repository

More here: <https://github.com/christian-korneck/docker-pushrm>

Let’s download the binary

Go to the latest release by visiting <https://github.com/christian-korneck/docker-pushrm/releases/latest>

Currently, the latest release is **v1.9.0**

Download it with

**wget** [**https://github.com/christian-korneck/docker-pushrm/releases/download/v1.9.0/docker-pushrm\_linux\_amd64**](https://github.com/christian-korneck/docker-pushrm/releases/download/v1.9.0/docker-pushrm_linux_amd64)

*For* ***ARM64*** *(****AARCH64****) you must download the* ***docker-pushrm\_linux\_arm64*** *package*

Make sure that the target directory is there

**mkdir -pv ~/.docker/cli-plugins**

And install it

**mv -v docker-pushrm\_linux\_amd64 ~/.docker/cli-plugins/docker-pushrm**

Finally, change its permissions

**chmod +x ~/.docker/cli-plugins/docker-pushrm**

Please note that this is a CLI plugin. That is why it is installed in the home folder of the current user

You can test if the installation went fine by executing the Docker CLI

**docker**

**…**

**plugin Manage plugins**

**pushrm\* Push Readme to container registry**

**system Manage Docker**

**…**

If the result is similar to the above, then the installation was successful

Now, let’s reuse our little playground from the previous exercise

Add a **README.md** file in the **labels** folder with the following content

**# A simple NGINX-based web application**

**It could be run with a command like this**

**```bash**

**docker container run -d -p 8080:80 it-works**

**```**

**If you open the address of the running container in a browser, you will see**

**\*\*It Works (but in container)\*\***

Now login to Docker Hub

**docker login**

Then make sure you are in the right place (the **labels** folder) and first, build the image

**docker build -t <account>/it-works:latest .**

*Where* ***<account>*** *is your name in* ***Docker Hub***

And then push the image

**docker push <account>/it-works:latest**

Finally, push the readme file

**docker pushrm <account>/it-works:latest**

That is it, now go to Docker Hub and see the result 😉

#### Use the Right Base Image

Let’s explore how the base image selection affects the size of the resulting image

Clone the sample repository

**git clone https://github.com/shekeriev/docker-app-pack**

And enter the folder **docker-app-pack/python/world-of-kubernetes**

Explore the **Dockerfile** there

Build an image with

**docker image build -t sample-app-1 .**

Now, check the size of the image with

**docker image ls**

Now change the first line in the **Dockerfile** to match this

**FROM python:3.13-slim-bookworm**

And build a new image with

**docker image build -t sample-app-2 .**

Check the size of the image again

**docker image ls**

There is a difference, isn’t it?

See if there is another image (perhaps an **alpine**-based one) that could lead to an even smaller size and test it

#### Multi-stage Build

Let’s explore how the image build process could be optimized

*This is applicable for applications written in languages that require a sort of building from source code*

*For interpreted languages check the previous section – there, important decision is the selection of a base image*

Clone the sample repository, if you do not have it

**git clone https://github.com/shekeriev/docker-app-pack**

And enter the folder **docker-app-pack/go/world-of-docker**

Explore the **Dockerfile** there

Build an image out of it

**docker image build -t world-of-docker:go .**

Now, check the size of the image with

**docker image ls**

Now, explore the **Dockerfile.multistage** file

It utilizes the so-called multi-stage build

Build another image following the multi-stage approach

**docker image build -t world-of-docker:goms -f Dockerfile.multistage .**

Check the size of the image again

**docker image ls**

There is a difference, isn’t it? 😉

Of course, this technique could be combined with others. For example, utilizing the **.dockerignore** file to filter some of the artifacts, selecting an appropriate base image for the last stage, etc.

#### Scan Images

For this, we can use **Trivy**, which is a popular image scanning tool

More information here: <https://trivy.dev/latest/>

There are multiple ways to install it, but for this exercise, we will use the install script

**curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sudo sh -s -- -b /usr/local/bin latest**

Once done, check that it is working by executing

**trivy --version**

Now, we can scan an image that is either locally stored or one in **Docker Hub**

In any case, the generic execution is done like this

**trivy image nginx:latest**

By default, **trivy** looks for an image in the following sources (in order) - **docker**, **containerd**, **podman**, and **remote**

Exploring the detailed output, we can make an informed decision about which image to use and/or if our image is fine

#### Use a Linter

Let’s explore one of the options here – **hadolint**. More information here: <https://github.com/hadolint/hadolint>

First install the lates version for your OS from: <https://github.com/hadolint/hadolint/releases/latest>

As we are going to use it in a **Vagrant** box, we will download the Linux version

**wget https://github.com/hadolint/hadolint/releases/download/v2.12.0/hadolint-Linux-x86\_64**

*If running on ARM64 (AARCH64) platform, then download the* ***hadolint-Linux-arm64*** *package*

Rename the file

**mv hadolint-Linux-x86\_64 hadolint**

Change its permissions

**chmod +x hadolint**

Move it to a folder that is part of the **PATH** environment variable

**sudo mv -v hadolint /usr/local/bin/**

Test that the CLI is available and working

**hadolint --version**

Create a folder for this experiment

**mkdir hadolint**

Enter there and create a **Dockerfile** with the following content

**FROM ubuntu**

**RUN apt-get update**

**RUN apt-get upgrade -y**

**RUN apt-get install -y git**

Even though we know that there are issues with it, let’s lint it

**hadolint Dockerfile**

We may see something like this

**Dockerfile:1 DL3006 warning: Always tag the version of an image explicitly**

**Dockerfile:3 DL3009 info: Delete the apt-get lists after installing something**

**Dockerfile:5 DL3059 info: Multiple consecutive `RUN` instructions. Consider consolidation.**

**Dockerfile:7 DL3008 warning: Pin versions in apt get install. Instead of `apt-get install <package>` use `apt-get install <package>=<version>`**

**Dockerfile:7 DL3059 info: Multiple consecutive `RUN` instructions. Consider consolidation.**

**Dockerfile:7 DL3015 info: Avoid additional packages by specifying `--no-install-recommends`**

If we ask for the exit code with

**echo $?**

We will notice that because of the findings it is not equal to zero

Let’s account for most of the suggestions and change the file to

**FROM ubuntu:24.04**

**RUN apt-get update \**

**&& apt-get upgrade -y \**

**&& apt-get install -y --no-install-recommends git \**

**&& apt-get clean \**

**&& rm -rf /var/lib/apt/lists/\***

And run the linting once again

**hadolint Dockerfile**

The result should be similar to this

**Dockerfile:3 DL3008 warning: Pin versions in apt get install. Instead of `apt-get install <package>` use `apt-get install <package>=<version>`**

To tackle warnings or other issues with the lint, we can add ignore hints in the **Dockerfile**

Let’s change it once again to match this

**FROM ubuntu:24.04**

**# hadolint ignore=DL3008**

**RUN apt-get update \**

**&& apt-get upgrade -y \**

**&& apt-get install -y --no-install-recommends git \**

**&& apt-get clean \**

**&& rm -rf /var/lib/apt/lists/\***

Now, if we execute the lint check, it should not return any findings

And if we ask again for the exit code, we will see that it is equal to zero

Note that the ignore hint, instead of attaching it to particular instruction, could be set globally, in the beginning of the file. Considering this, our file will look like

**# hadolint global ignore=DL3008**

**FROM ubuntu:24.04**

**RUN apt-get update \**

**&& apt-get upgrade -y \**

**&& apt-get install -y --no-install-recommends git \**

**&& apt-get clean \**

**&& rm -rf /var/lib/apt/lists/\***

Of course, if we build images out of the different versions, we should notice many differences 😉

### Troubleshooting

There is not a silver bullet and an ultimate approach to troubleshooting

Each case is different and requires specific steps

However, there are actions that are always helpful and will give us some insight into what the cause may be

Those include checking the logs of the container, inspecting its configuration, checking the **Dockerfile** used to build the image (if available), etc.

Let’s do a simple experiment

Clone the sample repository, if you do not have it

**git clone https://github.com/shekeriev/docker-app-pack**

And enter the folder **docker-app-pack/python/world-of-kubernetes-broken**

Explore the **Dockerfile** there

Build an image out of it

**docker image build -t python-app .**

Now, create and start a container out of the new image

**docker container run -d --name python-web -p 8080:5000 python-app**

Now, open a browser and try to reach the web application

You should see an error message like this

**Internal Server Error**

Okay, let’s try to understand what the problem may be

One of the first actions to take in cases like this should be to check the logs of the container in question

**docker container logs python-web**

Amongst the messages, we may notice the following

**jinja2.exceptions.TemplateNotFound: index.html**

Okay, it appears that a file is missing

If we check in the source code and the folder structure, we will find the file

So, let’s open a session inside the container and search for the file

**docker container exec -it python-web bash**

Explore the files there

**root@10764dd2fb32:/app# ls**

**\_\_pycache\_\_ app.py images index.html requirements.txt**

**root@10764dd2fb32:/app# cat index.html**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta content="text/html; charset=UTF-8" http-equiv="content-type">**

**<title>World of Kubernetes</title>**

**</head>**

**<body>**

**<div align="center">**

**<img width="400px" height="400px" src="{{ url\_for('static', filename='images/kubernetes-logo.png') }}">**

**<h1>Welcome to the world of <span style="color: blue;">Kubernetes</span>!</h1>**

**<h2>There is plenty to explore here ...</h2>**

**<h2>You can start with the name: <span style="color: blue;">κυβερνήτης</span> &rArr; <span style="color: blue;">kubernetes</span> &rArr; <span style="color: blue;">k8s</span> :)</h2>**

**<br /><br /><br />**

**Running on <b>{{ running\_on }}</b>**

**</div>**

**</body>**

**</html>**

**root@10764dd2fb32:/app#**

It appears that the file is there and has content

Before you close the session, compare once more the file and folder structure/hierarchy in the source folder and in the container

You should notice that there is a difference

Close the session

**exit**

What could cause this?

You will be surprised to understand that the issue lies within the **Dockerfile**

Go and check it once more

Did you notice it?

It is a single character that should not be there

The bad instruction is this one

**COPY app/\* /app/**

Instead, it should be

**COPY app/ /app/**

Go and change it, rebuild the image and rerun the container and see if it will behave differently 😉