

Docker Lab 3

Working with Dockerfiles

1	REFERENCES AND CHEAT SHEETS.....	1
2	COMMANDS COVERED	1
3	LAB SETUP	1
4	DOCKERFILE BASICS.....	2
5	DOCKERFILE FOR A NODEJS WEB APP.....	2
6	DOCKERFILE FOR A PYTHON CLI APP	10
7	TAGGING AND PUSHING TO DOCKERHUB	12
8	END	14

1 References and cheat sheets

The official reference for all DockerFile instructions:

<https://docs.docker.com/reference/cli/docker/>

https://kapeli.com/cheat_sheets/Dockerfile.docset/Contents/Resources/Documents/index

<https://devhints.io/dockerfile>

<https://medium.com/@oap.py/dockerfile-cheat-sheet-4ad12569aa0b>

2 Commands covered

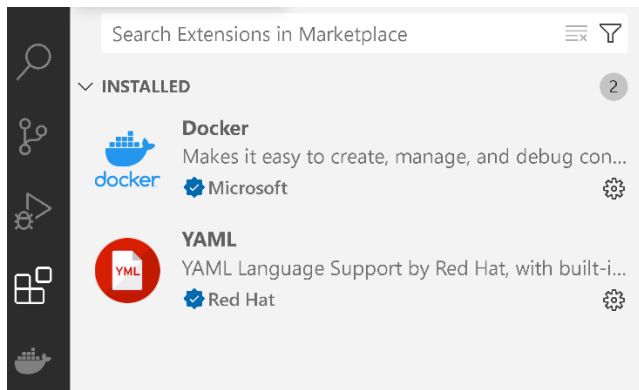
3 Lab setup

This is identical to the previous lab

We will use Visual Studio Code to edit the source code for the various applications.

<https://code.visualstudio.com/download>

Install the Docker extension for VS Code from Microsoft which facilitates the creation of Dockerfiles as well as managing your various Docker containers and images.



There are a variety of other Docker related extensions which can further facilitate your work with Docker-related projects, so feel free to install any of these when working with your real-life projects.

The root folders for all the various Node / Python projects here can be found in the `Lab 3` subfolder in the main `labcode` folder of your downloaded zip for this workshop.

4 Dockerfile basics

In an earlier lab, we saw how we can create a custom image from a running container using the `docker commit` command. The custom image essentially provides a way for us to persist all these changes made in a container that started from a base image (for e.g. `alpine`, `ubuntu`, etc) so that we can start a new container with exactly the same changes in place in the future without needing to repeat all the steps to create those changes. However, the most common approach to create a custom image is to use a Dockerfile to specify the explicit steps involved in the creation of that custom image.

A Dockerfile is a script that contains instructions for building a customized image. Each instruction in a Dockerfile creates a new layer in the image (starting from a base image), and the final image is composed of all the layers stacked on top of each other. The Dockerfile instructions typically will perform operations such as installing dependencies, copying files, setting environment variables, and configuring the container. Once a Dockerfile has been created, it can be used to build an image using the `docker build` command. The resulting image can then be run as a container using the `docker run` command.

Dockerfiles can be used in automation testing to build and run test environments for different applications and services. Using a Dockerfile, you can specify a custom image related to a specific test environment that can be easily and consistently recreated every time an automated test needs to be run, without needing to manually set up and configure the environment for each test run. This allows easy integration of Docker containers into all key phases of the CI/CD pipeline, which enables you to automatically build, test and deploy your application with every code change.

5 Dockerfile for a NodeJS web app

The root folder for this project is: `nodejs-app-first-dockerfile`

In the root folder, create the Dockerfile for building this project

Dockerfile

```
FROM node:14

WORKDIR /app

COPY . /app

RUN npm install

EXPOSE 80

CMD ["node", "server.js"]
```

To create an image based on this Dockerfile, run this command in the folder containing the Dockerfile. We also provide a tag for the image that will be generated.

```
docker build -t firstnodeapp:latest .
```

Check that the image is generated with the correct tag:

```
docker images
```

Next create and run a container from this custom image with a custom name in detached mode:

```
docker run -d --name nodeapp firstnodeapp:latest
```

Check that the container has started successfully with:

```
docker ps
```

Notice that although the app is listening on port 80, this is actually a port on an internal network within the Docker container and is not accessible on the host machine that Docker is running on. Verify this by attempting to open a browser tab at this port (<http://localhost:80>). You will not be able to connect.

Remove this running container

```
docker rm -f nodeapp
```

Check that it is in fact removed:

```
docker ps --all
```

Now run the container again with publishing of a port mapping from the internal Docker port to a freely available port on the local host machine:

```
docker run -d --name nodeapp -p 3000:80 firstnodeapp:latest
```

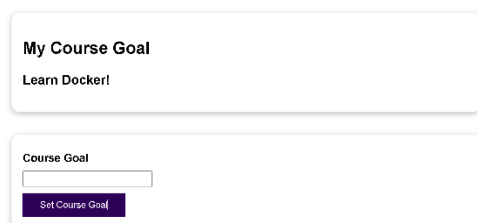
NOTE: We are using port 3000 as this is typically a free port on most Windows machines (i.e. no active process is using it). However, if you encounter an error related to the command above, it means that there is another process actively listening on port 3000, in which case you need to identify an existing free port in the standard port range which you can use.

Type this command in a shell terminal to identify all the ports that actively being used on the local machine:

```
netstat -an
```

<https://www.alphr.com/how-to-check-which-ports-open-windows-10-pc/>

Verify that you can access the Node server.js app on `localhost:3000`



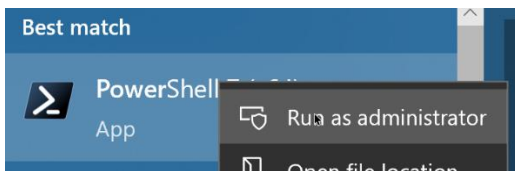
My Course Goal
Learn Docker!

Course Goal

Set Course Goal

Try interacting with it by setting some random course goals.

If you have administrator access to your local host machine, you can open a shell terminal with administrator access:



Type this command to get the PID of the process listening to port 3000 (or whichever port you have published the container to):

```
netstat -abon | findstr 3000
```

Get the PID from the listing that appears. Then check for the process name corresponding to this PID with:

```
tasklist | findstr PID
```

You should see an output similar to the following to confirm that in fact it is Docker that is actively listening on that port and redirecting all incoming HTTP requests on that port to the application running with the active container.

```
com.docker.backend.exe           15388 Console           1
134,560 K
```

Remove this running container again:

```
docker rm -f nodeapp
```

Check that it has actually been removed:

```
docker ps -a
```

Now introduce random change to the code base of the app (`server.js`), for e.g. by adding an extra header somewhere:

```
.....
    <body>
      <section>
        <h2>My Course Goal</h2>
        <h2>This is so cool !</h2>
        <h3>${userGoal}</h3>
    .....

```

Restart a new container again with the same name and port publishing:

```
docker run -d --name nodeapp -p 3000:80 firstnodeapp:latest
```

If you check the webpage rendering from the `server.js` app on `localhost:3000`

you will notice that the changes you incorporated do not show up.

This is because the running container is generated from the image, and the latest code changes have not been incorporated into the image.

To do this, we repeat the build to the same image tag:

```
docker build -t firstnodeapp:latest .
```

Notice that the build messages show that instructions which are not affected by the changes you have made (i.e. changing the source code in `server.js`) are not executed again and are marked by `CACHED`. The other instructions will have to be executed again in the sequence they appear in the Dockerfile starting from the first affected instructions

```
=> CACHED [2/4] WORKDIR /app
=> [3/4] COPY . /app
=> [4/4] RUN npm install
=> exporting to image
```

Delete the currently running container:

```
docker rm -f nodeapp
```

Restart a new container again with the same name and port publishing based on the latest built image:

```
docker run -d --name nodeapp -p 3000:80 firstnodeapp:latest
```

This time you should see the changes you have made take effect.

Now repeat the build process again without making any changes to the source code (or anywhere else):

```
docker build -t firstnodeapp:latest .
```

Notice that it completes almost instantaneously because the build process simply uses the intermediate cached layers from the instructions of the previous build because nothing has changed.

```
=> => transferring context: 184B
=> CACHED [2/4] WORKDIR /app
=> CACHED [3/4] COPY . /app
=> CACHED [4/4] RUN npm install
=> exporting to image
```

Again, make further random changes to the code base of the app (`server.js`), for e.g. by adding an extra header somewhere:

Again, repeat the build to the same image tag:

```
docker build -t firstnodeapp:latest .
```

Again, notice that the build messages show that instructions which are not affected by the changes you have made (i.e. changing the source code in `server.js`) are not executed again and are marked by CACHED. The other instructions will have to be executed again in the sequence they appear in the Dockerfile starting from the first affected instructions

Delete the currently running container:

```
docker rm -f nodeapp
```

Restart a new container again with the same name and port publishing based on the latest built image:

```
docker run -d --name nodeapp -p 3000:80 firstnodeapp:latest
```

You should see the changes you have made take effect.

We can see now that that Docker will execute all instructions in the Dockerfile starting from the first instruction that reflects the changes that we have made in our project and then all following instructions after that one (even if they are not affected by the change).

Based on this understanding, we can arrange the sequencing of our instructions so that instructions that take a long time to complete, consume a lot of resources (i.e. network bandwidth) and are unlikely to be affected by the changes that we would typically make to a project (90% of the time would be source code / configuration file changes), are placed higher up in the Dockerfile so that they will remain cached and not be executed again.

Rewrite your Dockerfile to the following sequence:

```
FROM node:14
```

```
WORKDIR /app

COPY package.json /app

RUN npm install

COPY . /app

EXPOSE 80

CMD ["node", "server.js"]
```

Notice now that the instruction that will be affected by the change to `server.js` (`COPY . /app`) is further down in the Dockerfile.

Repeat the build to the same image tag:

```
docker build -t firstnodeapp:latest .
```

The first time you run this build, all instructions except `WORKDIR /app` will be executed again.

Delete the currently running container:

```
docker rm -f nodeapp
```

Again, make further random changes to the code base of the app (`server.js`), for e.g. by adding an extra header somewhere:

Repeat the build to the same image tag:

```
docker build -t firstnodeapp:latest .
```

Now this time notice that the build finishes much faster with this changed sequence of instructions since only the affected instruction (`COPY . /app`) and instructions after it will need to be re-executed, and Docker can just reuse the cached intermediate layers corresponding to the other unchanged instructions prior to that.

```
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY package.json /app
=> CACHED [4/5] RUN npm install
=> [5/5] COPY . /app
=> exporting to image
```

Restart a new container again with the same name and port publishing based on the latest built image:

```
docker run -d --name nodeapp -p 3000:80 firstnodeapp:latest
```

You should see the changes you have made take effect.

Repeat the steps above a few more times to clearly see this for yourself.

This illustrates an important principle of how we can optimize the build time for our Dockerfiles by arranging our instructions so that the instructions that will be affected by the most common changes we make (usually 90% of the time will be to our project source code / configuration files) are as further down in the Dockerfile as is possible. At the same time, instructions that take a long time to complete, consume a lot of resources (i.e. network bandwidth) and are unlikely to be affected by the changes that we would typically make should be placed higher up in the Dockerfile so that they will remain cached and not be re-executed after the first execution.

If at this point, you check the images in the local repository with:

```
docker images
```

You should see a number of untagged images similar to the output below. These are actually the intermediate image layers that were created during the subsequent rebuilds of our Dockerfile corresponding to the various changes we made.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
firstnodeapp	latest	452a59c815f4	2 hours ago	917MB
<none>	<none>	8664652d8dc0	2 hours ago	917MB
<none>	<none>	2e465563bd29	2 hours ago	917MB
<none>	<none>	a0e18daa3651	3 hours ago	917MB

If you were to inspect the latest image (`firstnodeapp:latest`) with:

```
docker inspect firstnodeapp
```

And study at the set of image layers that comprise this image (typically at the bottom)

```

"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:b2dba74777543b60e1a5be6da44e67659f51b8df1e96922205a5dde6b92dda3c",
    "sha256:f1186e5061f20658954f6bfdfaead0abc5ed2371d70f707da05206e868a226ab",
    "sha256:fe0fb3ab4a0f7be72784fcab5ef9c8fda65ea9b1067e8f7cdf293c12bcd25c13",
    .....
    .....
  ]
}
```

The first layer is the base layer corresponding to the execution of the first instruction in the Dockerfile, and so on.

Now inspect the same set of image layers for any one of those untagged intermediate layers that you see the image listing by using its imageid, for e.g.

```
docker inspect imageid
```

Contrast its set of image layers with the ones for `firstnodeapp:latest`. You should see that they are nearly identical (based on identical SHA1 digest) except for the last 1 or 2 layers, which accords exactly with the logic of how image layers are generated from execution of successive Dockerfile instructions.

There is a significance to whether the container is being run in a detached or attached mode. At this point of time, we are starting the container in detached mode because of the use of the `-d` flag.

Notice that in the app, we are actually outputting messages to the console.


```
app.post('/store-goal', (req, res) => {  
  const enteredGoal = req.body.goal;  
  console.log(enteredGoal);  
  userGoal = enteredGoal;  
  res.redirect('/');  
});
```

However, this does not appear at any point and is not accessible when the container is running in detached mode. Often, we do want to see these console log output because the messages may be relevant for debugging purposes particular when we are at the development stage.

Delete the currently running container:

```
docker rm -f nodeapp
```

This time start it without the `-d` flag, which makes it start in attached mode by default. The PowerShell terminal will block after this command:

```
docker run --name nodeapp -p 3000:80 firstnodeapp:latest
```

This time all the course goals entered at the app will appear in the terminal.

If we type this in a second terminal, we should also be able to see the console output up to the current point in time since we started the container. We covered this command in a previous lab.

```
docker logs nodeapp
```

We can also follow the log output dynamically (`-f`) with timestamps added in (`-t`)

```
docker logs -f -t nodeapp
```

Press Ctrl-C to exit

In the second terminal, delete the currently running container:

```
docker rm -f nodeapp
```

In the first terminal, start it again but this time in detached mode with:

```
docker run -d --name nodeapp -p 3000:80 firstnodeapp:latest
```

As expected, there is no console log output for any goals that we enter into the app. However, at this point we can attach to the running container with:

```
docker attach nodeapp
```

Once we are attached, the console log output is now visible and also accessible again via the `docker logs` command as was the case previously.

In the second terminal, delete the currently running container:

```
docker rm -f nodeapp
```

Often, we want to only run the container once to perform a specific operation and do not expect to start it and run it again. For this we can add the `--rm` flag to automatically remove the container when it exits, which is very useful so that you don't manually have to clean up all these stopped containers periodically in order to not clutter your listing and free up system resources.

In the first terminal, start the container in the detached mode:

```
docker run -d --rm --name nodeapp -p 3000:80 firstnodeapp:latest
```

In the same terminal, stop it with:

```
docker stop nodeapp
```

After stopping, check that the container has been automatically deleted:

```
docker ps -a
```

6 Dockerfile for a Python CLI app

The root folder for this project is: `python-app-starting-setup`

In the root folder, create the Dockerfile for building this project

Dockerfile

```
FROM python

WORKDIR /app

COPY . /app

CMD [ "python", "rng.py" ]
```

To create an image based on this Dockerfile, run this command in the folder containing the Dockerfile. We also provide a tag for the image that will be generated.

```
docker build -t firstpythonapp:latest .
```

Check that the image is generated with the correct tag:

```
docker images
```

Since this app requires interaction with the user via CLI, we will first attempt to start it in attached mode (without the `-d` flag):

```
docker run --name pythonapp firstpythonapp:latest
```

Notice that we get an error regarding the execution the line of code that requires CLI input and the container exits. This is because although we are attached to the container, we cannot provide external input that can be read by the app running in the container.

Remove the stopped container.

```
docker rm -f pythonapp
```

Rerun the container using the `-it` flag that we had covered before in a previous lab.

```
docker run -it --name pythonapp firstpythonapp:latest
```

This time you will be able to interact with the application and provide input on which it acts upon, and subsequently corresponding output is produced according to the program logic (which is to print a random number between a lower and upper limit)

Now if you attempt to restart the stopped container with:

```
docker start pythonapp
```

it starts in detached mode by default, so you cannot communicate with it.

Stop it:

```
docker stop pythonapp
```

You could attempt starting it in attached mode

```
docker start -a pythonapp
```

Here the first user prompt appears, but again the app freezes at the CLI after which you are no longer able to interact with it.

Press Ctrl-C to exit.

Stop it again:

```
docker stop pythonapp
```

This time, start it with the interactive option provided as well:

```
docker start -ai pythonapp
```

Now the container should be able to capture external console input and also provide console response correctly after processing these inputs.

7 Tagging and pushing to DockerHub

We now have 2 custom images built using 2 separate Dockerfiles. To use these images to recreate the exact same application and environment for different purposes (testing, production) in a CI/CD pipeline, we can push them to a public DockerHub account where it can then be pulled to a different server and use to generate containers.

Check again on the 2 custom images that we created:

```
docker images
```

As we have already seen before, we can tag this image to a suitable form to enable pushing to our Docker Hub account. NOTE: the tags provided are not compulsory, you can decide on your own naming scheme for the images you push to your Docker Hub account as long as they adhere to the `repo:version` format.

```
docker tag firstpythonapp:latest dockerhubaccount/workshop-pythonapps:v1
```

```
docker tag firstnodeapp:latest dockerhubaccount/workshop-nodeapps:v1
```

Check that these 2 new additional reference tags have been created in the local registry:

```
docker images
```

Make sure you are logged in (if you have not already previously logged in or you logged out after that):

```
docker login
```

Finally, push the images to Docker Hub in the usual manner:

```
docker push dockerhubaccount/workshop-pythonapps:v1
```

```
docker push dockerhubaccount/workshop-nodeapps:v1
```

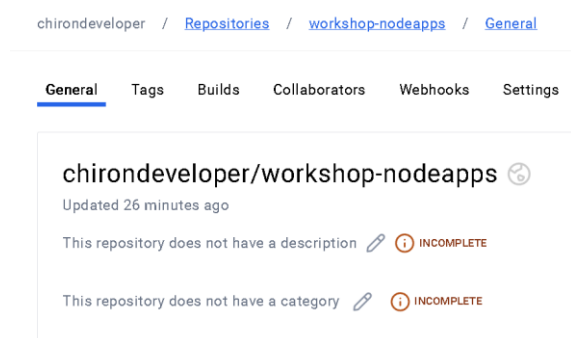
In the output messages that appear with these commands, you will see some messages that look like these below:

```
d6b126aefec1: Mounted from library/python
94e813cd30e6: Mounted from library/python
...
...
0d5f5a015e5d: Mounted from library/node
3c777d951de2: Mounted from library/node
...
...
cc1de0bc8241: Pushed
e586a3c26c70: Pushed
```

The mounted messages are referring to the fact that Docker has detected that some image layers in the custom image being pushed to your Docker Hub account already exist on Docker Hub. These would be the base images for both these custom images (corresponding to the first instructions `FROM node:14` and `FROM python`)

So rather than actual pushing these layers from the local host to Docker Hub to reconstitute the complete image there, Docker Hub simply reuses these existing image layers. Only the unique image layers are actually pushed (the push messages)

Verify that the images are present in the respective repositories in your DockerHub account. You can also provide more information on the repositories (such as description and category) if you wish.



Note that what we have done here is to create new repositories (`workshop-pythonapps` and `workshop-nodeapps`) indirectly as a result of the tagging and push operations. You could also first explicitly create the repositories on Docker Hub if you wish before pushing.

Now that we have pushed both our custom images successfully, let's delete them first before pulling them back from our Docker Hub accounts.

Remove all local images with:

```
docker image prune --all
```

Check that they have been removed with:

```
docker images
```

Finally, pull down these custom images from our Docker Hub account:

```
docker pull dockerhubaccount/workshop-pythonapps:v1
```

```
docker pull dockerhubaccount/workshop-nodeapps:v1
```

Now verify that you can start containers from both these images in the same way that you had done previously:

```
docker    run    -d    --rm    --name    nodeapp    -p    3000:80  
dockerhubaccount/workshop-nodeapps:v1
```

```
docker    run    -it    --name    pythonapp    dockerhubaccount/workshop-  
pythonapps:v1
```

8 END