Docker Workshop
Lab 3
Working with Dockerfiles

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

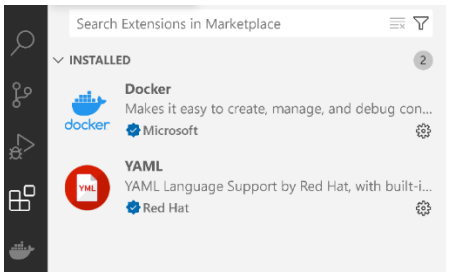
2 Commands used

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.

Ensure that the Docker extension for VS Code from Microsoft is installed as it facilitates the creation of Dockerfiles as well as managing your various Docker containers and images.



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.

Copy these folders to the folder that you initially created on your local machine for this workshop and work from them there.

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. All of the instructions in a Dockerfile will create a new intermediate layer that builds on the layer corresponding to the instruction prior to it. All of this will start from a base image (typically an official DockerHub repo 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 Express web app

The root folder for this project is: `nodejs-dockerfile`
Open this folder in VS Code

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

`Dockerfile`

```
FROM node:14.18.0

WORKDIR /app

COPY . /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false
RUN npm install

EXPOSE 80

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

Let's examine these instructions in detail:

a) The **FROM** instruction specifies a starting base image from which you intend to build your custom image. This is typically an image from an official DockerHub repo (here this is v14 of the Node runtime installed on a specific Linux distro image, typically Debian or Ubuntu). If we wanted a lighter weight image that only incorporates the bare minimum packages for the Node runtime to work, we could have used the Alpine or Slim image for this particular version (for e.g. `14.18.0-slim` or `14.18.0-alpine`)

b) The **WORKDIR** instruction specifies the working directory for any command (COPY, RUN, CMD, ADD, etc) that follows it in the Dockerfile. This means that any commands that are run in the container will be executed relative to this specified directory. This instruction creates the directory if it does not already exist

c) The **COPY** instruction copies files / directories from the file system of the host machine to a destination directory in the container file system that will be generated from this custom image. Typically, this destination directory will also be the working directory, and the files / directories to be copied will be usually in the same folder (the project folder) where the Dockerfile resides in.

d) The **RUN** instruction executes commands within the working directory that is relevant for running a specific application. Here `npm install` installs all the Javascript dependencies (libraries) that are specified in the `dependencies` section of `package.json` file. If you are starting from a base Linux distro image like Ubuntu (for e.g. `ubuntu:jammy` instead of a base image which already has Node installed like `node:14.18.0` in this example), you might also need to run commands before to install the Node runtime first, for e.g.

```
RUN apt update
RUN apt install nodejs
RUN apt install npm
```

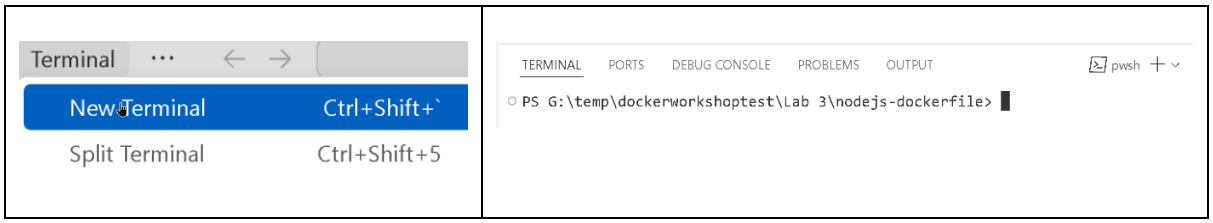
Sequence of installation instructions taken from here:

e) The **EXPOSE** instruction tells the user that the app inside the container is listening on a specific port, in this example, port 80 (which is the default port for HTTP traffic). However, this does not make the port accessible on the networking system of the native host (localhost), for that we will need to perform port publishing as we will see later. This instruction is optional and if it is not included, network access is still possible as long as port publishing is performed. However, as a general rule, we will include this instruction for any applications in a container that listens on a port.

f) The **CMD** instruction is typically the last instruction inside the Dockerfile and it specifies the command that will be executed when a container is generated from the image using the `docker run` command. This instruction is usually in the form of

```
CMD ["executable","param1","param2",...]
```

Open one or more Powershell 7 terminals in the root folder of this project in order to execute Docker CLI commands.
If you are using VS Code, you can also alternatively access the inbuilt Powershell terminal functionality within the IDE itself.



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

```
docker build -t firstnodeapp .
```

The execution of any instructions that causes changes to the file system of the container generated from the custom image created from this Dockerfile will result in an intermediary image layer with new content that is created and cached by Docker to speed up the subsequent builds using the same Dockerfile. Examples of such instructions are FROM, RUN, COPY, ADD, etc. Instructions which do not create any file system changes, but merely provide configuration (such as WORKDIR, ENV, etc) will not create zero size metadata layers with no content. However, every instruction will result in an intermediate image layer, where the final custom image consists of all these intermediate layers stacked on top of each other.

The numbers in the output messages indicate the generation of these intermediate layers that start from the original base image (`FROM node:14.18.0`).

```
=> [1/4] FROM docker.io/library/node:14  0.0s
=> [2/4] WORKDIR /app                    0.0s
=> [3/4] COPY . /app                     0.1s
=> [4/4] RUN npm install
```

Check that the image is generated with the correct tag:

```
docker images
```

NOTE: Once a specific directory has been designated as a work directory via the `WORKDIR` instruction, using the . on the right-hand side of any instruction that requires directories as arguments will always reference this work directory. Therefore, the two instructions below are exactly equivalent and often you will see Dockerfiles that use the shorter form.

```
.....
.....

WORKDIR /app

#Both these 2 instructions below are functionally equivalent
COPY . /app

COPY . .


.....
.....
```

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

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

Check that the container has started successfully with:

```
docker ps
```

The server app (`server.js`) is current listening on port 80 of the 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.

Open an interactive shell terminal into the container to explore the server process running there with:

```
docker exec -it nodeapp /bin/bash
```

Notice the terminal opens into the designated WORKDIR folder of the Dockerfile, which is `/app`

Notice that it contains a `node_modules` folder, which contains the library dependencies that were installed by the `npm install` command based on the content of `package.json`. We can check the number of dependencies in this folder.

```
root@1c3f0f795d7a:/app# ls -l
total 48
-rwxr-xr-x  1 root root   232 Aug 29 05:21 Dockerfile
drwxr-xr-x 66 root root  4096 Aug 29 05:22 node_modules
-rw-r--r--  1 root root 27194 Aug 29 05:22 package-lock.json
-rwxr-xr-x  1 root root   221 Aug 18 02:35 package.json
drwxr-xr-x  2 root root  4096 Aug 29 04:06 public
-rwxr-xr-x  1 root root   939 Aug 29 05:23 server.js
root@1c3f0f795d7a:/app# cd node_modules
root@1c3f0f795d7a:/app# find . -type f | wc -l
….
```

…..
….

Notice you have several hundred files in this folder: all of which are downloaded from the public NPM registry when the `npm install` command executed. This is resource intensive process which takes the longest time to complete among all the Dockerfile instructions. This is an important point that we will keep in mind when we need to optimize sequencing of Dockerfile instructions later.

Next, verify that the running server process `server.js` is actually listening on port 80

```
root@c7ce40bea5e7:/app# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 587444 40004 ?        Ssl  02:56   0:00 node server.js
root          14  0.0  0.0   3864  3252 pts/0    Ss   03:00   0:00 /bin/bash
root          21  0.0  0.0   7636  2744 pts/0    R+   03:01   0:00 ps aux
root@c7ce40bea5e7:/app# ss -tupln
Netid     State      Recv-Q     Send-Q         Local Address:Port          Peer Address:Port
tcp        LISTEN     0          511                  *:80                       *:*
users:(("node",pid=1,fd=18))
root@c7ce40bea5e7:/app# exit
```

You will notice that the process `node server.js` (which corresponds to the last instruction in the Dockerfile (`CMD ["node", "server.js"]`) has a PID of 1, which means it's the main process for this container and if it shuts down, the entire container will stop.

Remove this running container and check that it has being removed:

```
docker rm -f nodeapp
```

```
docker ps -a
```

Now run the container again with publishing of a port mapping from the internal Docker port (80) to a freely available port (3000) on the local host machine:

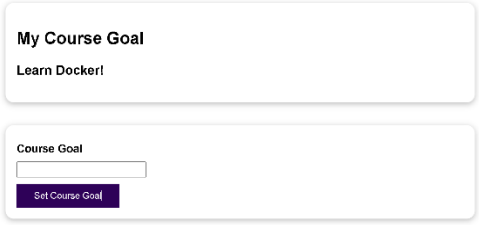
```
docker run -d --name nodeapp -p 3000:80 firstnodeapp
```

Check for the port mapping in the listing of the container with:

```
docker ps
```

The VS Code Docker extension UI also shows this mapping.

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



Try interacting with it by setting some random course goals.

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

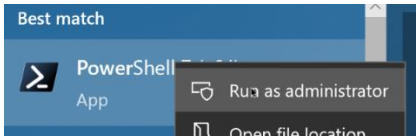
there is another process actively listening on port 3000, in which case you need to identify an existing free port in the standard port range which you can use.

In that case, type this command in a shell terminal to identify all the ports that actively being used on the local machine and select one that is free (from the top of the listing)

```
netstat -an
```

Use that free port number in the previous command in replacement of 3000, and use that number for all future references to 3000 in this lab session.

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
```

Depending on whether you are using Docker Desktop or Rancher Desktop, you should see the name of a process that is bound to this port which is actively listening on that port and redirecting all incoming HTTP requests on that port to the application running with the active container.

For Docker Desktop, the name of the process is:
```
com.docker.backend.exe        Console                    1
```
For Rancher Desktop, the name of the process will be:
```
host-switch.exe              Console                    1
```

Remove this running container again and check that it has actually been removed:

```
docker rm -f nodeapp
```

```
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
```

If you refresh the webpage 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.

5.1 Optimizing sequencing of Dockerfile instructions

To incorporate these new changes in our source code, we repeat the build to generate a new image. We could provide a new tag for this new image (for e.g. `firstnodeapp:v1`), but to keep things simple we will reuse the previous tag.

```
docker build -t firstnodeapp .
```

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. This indicates the intermediate image layer corresponding to those instruction executions from the previous build which has been cached can be reused directly.

Since the source code file is part of the current directory and was changed, then the instruction `COPY . /app` needs to be executed again (we cannot reuse the previous cached intermediate image layer). Subsequently, all other instructions following that will be executed again in the sequence they appear in the Dockerfile EVEN if there was no change from the previous build: the cached intermediate image layers cannot be reused anymore.

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

This is an important observation that we can use to optimize the sequence of instructions in the Dockerfile.

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
```

When you refresh the webpage at `localhost:3000`, you should now be able to see the latest source code 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 .
```

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
```

Now, 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 .
```

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

In particular the `RUN npm install` command will be executed, which results in the downloading of a large number of dependencies (as we saw earlier) which is basically redundant since these dependencies are going to be the same as they are based on the `dependencies` element in `package.json` which has not changed.

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
```

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, we present the best practice principle of optimizing Dockerfiles to speed up build time:

Arrange the sequencing of our instructions so that instructions **that take a long time to complete or 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. The classic example of such an instruction is `RUN npm install` which in turn requires `package.json` to work correctly

With this in mind, we can rewrite your Dockerfile to this more optimized sequence:

```
FROM node:14.18.0

WORKDIR /app

COPY package.json /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false
RUN npm install

COPY . /app

EXPOSE 80

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

We now initially only copy `package.json` over to our working directory because this is the only file that is required for `npm install` (the next instruction) to complete successfully. Finally, only after that do we copy over all the remaining files in the current directory (`COPY . /app`) – this is the main instruction that will be affected by frequent changes that we will be making to `server.js` throughout the lifetime of this project.

Repeat the build to the same image tag:

```
docker build -t firstnodeapp .
```

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 .
```

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
```

You should see the changes you have made take effect.

Repeat the steps above a few more times (change source code, run build, delete running container, create container again) to clearly see this for yourself.

This illustrates an important principle of how we can optimize the build time for our Dockerfiles

a) Arrange 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. For e.g. copying these source code / configuration files into the working directory in the container.

b) 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. For e.g. `npm install`.

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. They are termed as dangling images

```
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
```

The VS Code Docker extension UI can also show or hide this dangling images.

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 dangling images 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.

5.2 Using a .dockerignore file

When you run the `docker build` command, Docker sends all of the files in your current directory (also known as the build context) to the Docker daemon. This includes everything: source code, configuration files, log files, databases, and even files that aren't necessary for building your Docker image.

The `.dockerignore` file allows you to specify which files and directories Docker should ignore when building an image. This speeds up the build process and results in smaller Docker images because the unnecessary files are not included. This is conceptually equivalent to the `.gitignore` file that you may be familiar with if you have worked with Git.

To test this out, create two text files with random names in the root project folder containing the current Dockerfile and populate them with random content:

```
cats.txt
dogs.txt
```

Run the build to generate the image again:

```
docker build -t firstnodeapp .
```

Run a container from this generated image (you may need to remove the previous container with the same name if it is still running):

```
docker rm -f nodeapp
```

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

Now connect to the running container with:

```
docker exec -it nodeapp /bin/bash
```

If you check the content of the `/app` folder, you should see these two text files

```
root@0e7bf3e8d7d1:/app# ls -l
total 56
-rwxr-xr-x  1 root root   226 Feb 26 12:00 Dockerfile
-rwxr-xr-x  1 root root    18 Feb 26 23:53 cats.txt
```

```
-rwxr-xr-x  1 root root    21 Feb 26 23:53 dogs.txt
..
..
root@0e7bf3e8d7d1:/app# exit
```

Now create a `.dockerignore` file in the same project folder with the following single line to exclude both these files from being transferred over to the custom image.

`.dockerignore`

```
# Ignore all .txt files
*.txt
```

Run the build to generate the image again:

```
docker build -t firstnodeapp .
```

Run a container from this generated image (you may need to remove the previous container with the same name if it is still running):

```
docker rm -f nodeapp
```

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

Now connect to the running container with:

```
docker exec -it nodeapp /bin/bash
```

If you check the content of the `/app` folder, you will not see these two text files

```
root@1c3f0f795d7a:/app# ls -l
total 48
-rwxr-xr-x  1 root root   232 Aug 29 05:21 Dockerfile
drwxr-xr-x 66 root root  4096 Aug 29 05:22 node_modules
-rw-r--r--  1 root root 27194 Aug 29 05:22 package-lock.json
-rwxr-xr-x  1 root root   221 Aug 18 02:35 package.json
drwxr-xr-x  2 root root  4096 Aug 29 04:06 public
-rwxr-xr-x  1 root root   939 Aug 29 05:23 server.js
root@1c3f0f795d7a:/app# exit
```

Remove the running container:

```
docker rm -f nodeapp
```

One of the classic use cases for `.dockerignore` file is to exclude library dependencies that may be already present in the root project folder on the native host. This is because the Dockerfile will already typically contain instructions to install the exact same dependencies on the image, and therefore copying these same dependencies to the image is a redundant operation that will significantly slow down the build process.

If you already have Node installed on the host system (if you don't, that's fine), you can download all the dependencies here with the same command that we used in the Dockerfile instruction for the custom image:

```
npm install
```

This will result in the creation of a `node_modules` folder in the root project folder on the host machine that contains all the library dependencies required by `server.js`.

With this in place, you could immediately run server.js on your host system with:

```
node server.js
```

and you should be able to access it on
http://localhost/

NOTE: You have to ensure that port 80 is available on your localhost for this to work, since that is the port that server.js is listening on.

You can end the server with Ctrl+C.

Notice that the Dockerfile runs the instruction `RUN npm install` before it runs `COPY . /app`

This means that there will already be a `node_modules` folder in the image which is then again overwritten with a `node_modules` folder with identical content (since the content is based on the same `package.json`) from the host file system during the COPY instruction. Now if we were constantly changing our source code and running the build to incorporate these changes, the `COPY . /app` instruction will always be rerun (as explained earlier) and therefore this unnecessary copying of `node_modules` is performed repeatedly.

Test this out by changing the source code randomly in `server.js` and running the build after every change:

```
docker build -t firstnodeapp .
```

Notice that the COPY instruction is always re-executed for each build.

```
=> [5/5] COPY . /app
```

In this example, the unnecessary copying of `node_modules` from the host file system to the container file system completes very quickly as the current content of this folder is relatively small. For large scale JavaScript projects (such as React or Angular), the content of the `node_modules` is very large and the copy time will be long.

So in situation like this we can avoid the redundant copying by simply adding this folder to our `.dockerignore` file.

```
# Ignore the node_modules folder
node_modules
```

More examples of other files / folders that are typically excluded using a `.dockerignore` file

5.3 Detached mode and log output

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
```

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

If we type the command below 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.

```
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
```

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
```

5.4 Auto removal with the --rm flag

Often, we only want to run the process in a container once and we are no longer interested to access the container once it has stopped. This could be because any intermediate data produced by the process (such as a webapp in this example) has been persisted to a permanent storage outside of the container (such as a volume, as we will see in an upcoming lab).

This means we will probably delete the container once it has stopped for whatever reason. To simplify this process, we can simply 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 cluster 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
```

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-cli-dockerfile`
Open this folder in VS Code

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

```
Dockerfile
```

```
FROM python:3.10-alpine

WORKDIR /app

COPY . /app

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

To create a custom 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 .
```

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
```

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
```

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
```

To communicate with it when you start it, you need to include options to be attached in interactive mode:

```
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.

Remove the stopped container.

```
docker rm -f pythonapp
```

7 Dockerfile for a Python Flask web app

The root folder for this project is: `flask-dockerfile`
Open this folder in VS Code

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

`Dockerfile`

```
FROM python:3.10-alpine

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

EXPOSE 8080

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

The main part here is the use of `requirements.txt` that states the specific Python packages / libraries and versions required by the app (in this case the Flask package), which is subsequently used by the `pip` package manager to install these packages. As usual, we will perform the instructions for those 2 operations before performing the main COPY . . instruction to optimize the build performance, as already described earlier.

To create a custom 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 firstflaskapp .
```

Check that the image is generated with the correct tag:

```
docker images
```

Next, create a container from this image in detached mode and publish a port mapping from the internal Docker port that the app is listening on (8080) to a any freely available port (8080) on the local host machine:

```
docker run --name flask-server -d -p 8080:8080 firstflaskapp
```

As is the case with the NodeJS web app, if port 8080 is not available on your local machine, then follow the appropriate steps to locate a free port.

Check that the container has started successfully with:

```
docker ps
```

Verify that you can access the Python Flask `app.py` on `localhost:8080`, which simply returns some basic HTML.

When you are done, remove this running container and check that it has being removed:

```
docker rm -f flask-server
```

```
docker ps -a
```

8 Tagging and pushing to Docker Hub

We now have 3 custom images built using 3 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 custom images that we created:

```
docker images
```

As we have already seen before, we can tag these images to a suitable form to enable pushing to our Docker Hub account. NOTE: the tags used below 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
```

```
docker tag firstflaskapp:latest dockerhubaccount/workshop-flaskapps:v1
```

Check that these 3 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

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

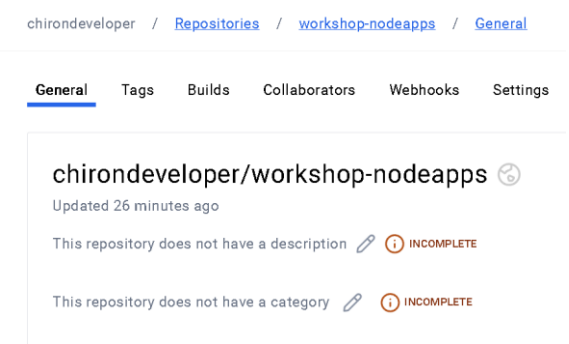
In the output messages that appear with these commands, you will see some messages that look like these below:

```
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:3.10-alpine`)

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.



These 3 new repositories (`workshop-pythonapps` , `workshop-nodeapps` and `workshop-flaskapps`) are created indirectly as a result of the tagging and push operations. However, 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, lets 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 being 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

docker pull dockerhubaccount/workshop-flaskapps: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

docker run --name flask-server -d -p 8080:8080 dockerhubaccount/workshop-flaskapps:v1
```

Finally, remove all these running containers with:

```
docker rm -f nodeapp

docker rm -f pythonapp

docker rm -f flask-server
```