KAZAKH-BRITISH TECHNICAL UNIVERSITY

Kazakh-British Technical University
School of Information Technologies and Engineering

Assignment #1
Web Application Development
Introduction to a Docker

Prepared by: Maratuly T.
Checked by: Serek A.

Almaty, 2024

CONTENT

Introduction 3
1. Intro to containerization: Docker 4
1.1 Exercise 1: Installing Docker 4
1.2 Exercise 2: Basic Docker Commands 7
1.3 Exercise 3: Working with Docker Containers 8
2. Dockerfile 10
2.1 Exercise 1: Creating a Simple Dockerfile 11
2.2 Exercise 2: Optimizing Dockerfile with Layers and Caching 12
2.3 Exercise 3: Multi-Stage Builds 14
2.4 Exercise 4: Pushing Docker Images to Docker Hub 16
Conclusion 19

INTRODUCTION

This work is responsible for getting familiar with Docker along with its components, structure, application, etc. Docker is a great tool for developing as only having it we can install and launch all the required containers (with necessary images, run needed command) and easily control them. Ubuntu was an operational system (OS) which we used to cover this lab. Pay attention to a user's username (here 'nekofetishist' is a username of Maratuly Temirbolat with ID 23MD0409).

1. Intro to Containerization: Docker

Before the beginning of the lab's process pay attention that username 'nekofetishist' belongs to Maratuly Temirbolat (ID 23MD0409). Figure 1.1 illustrates who is the owner of the account which will be used to cover the work.

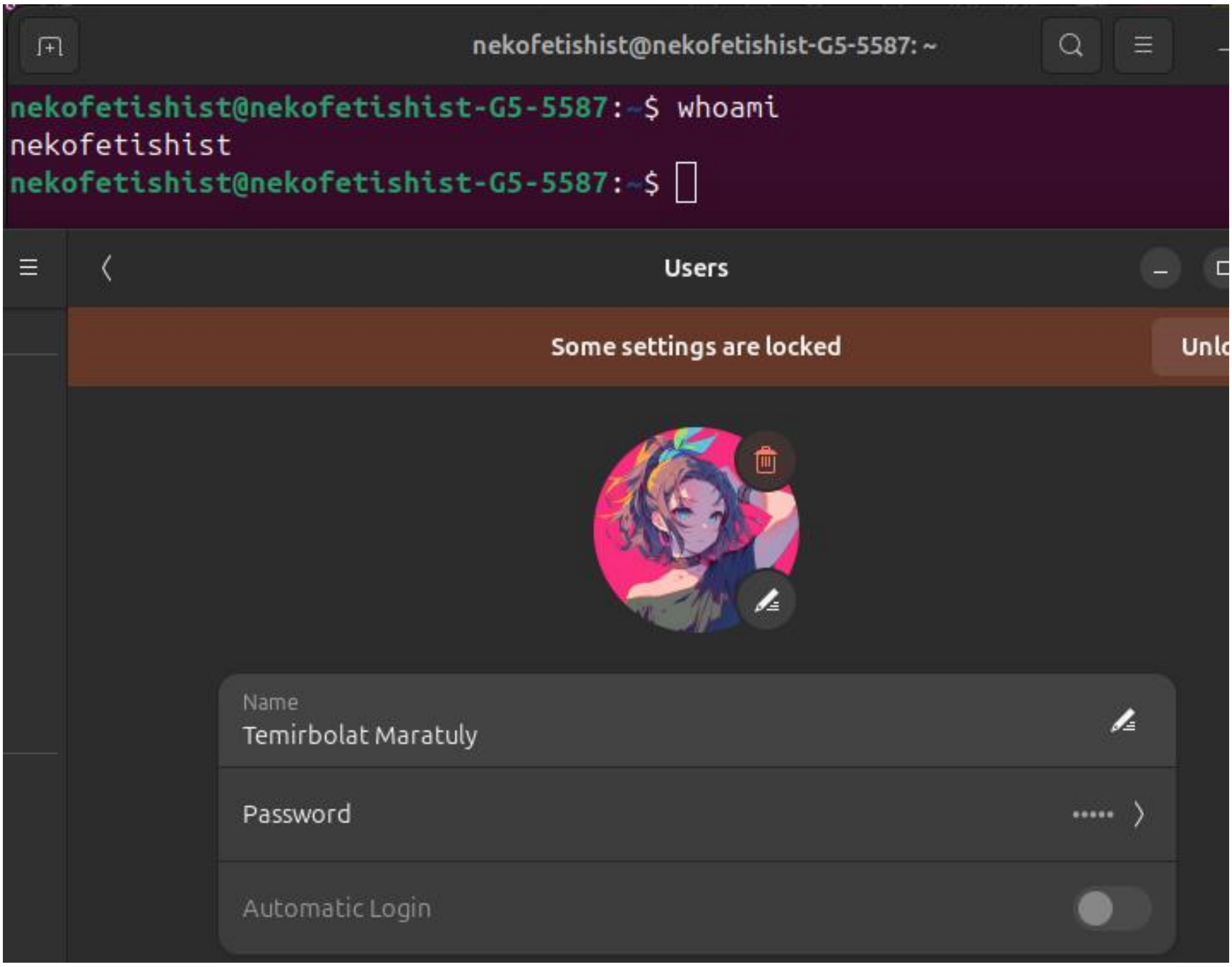


Figure 1.1 - Username and account's owner credentials information

Thus, according to a figure, the command 'whoami' from the Ubuntu command line shows the current user, moreover, this username is used in the whole path (marked with green letters).

1.1 Exercise: 1 Installing Docker

As soon as we verified the account's credentials, we needed to install docker for further work. All the work was done in command interface only. So, there are not any descriptions and explanations about docker desktop interface to work with run containers, related screenshots, etc. As we installed docker via official documentation, we need to make sure that it works. The best way to use 'docker --version' which must show the installed version of a docker. The information about docker's version is illustrated in Figure 1.2.

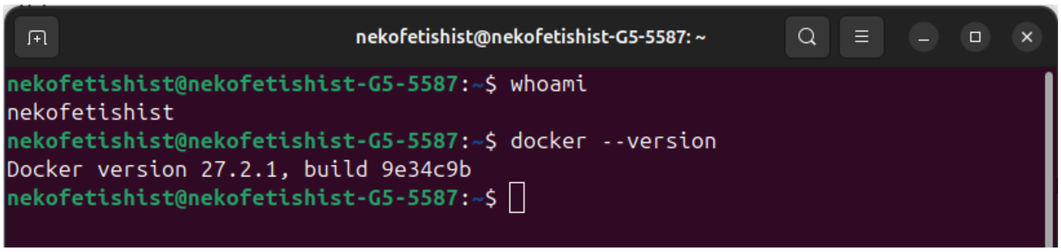


Figure 1.2 - Docker's installed version information

As we installed the docker, we can run 'hello-world' container to see that docker works properly with no issues and errors. Figure 1.3 demonstrates the results of this command. Docker will download 'hello-world' image if you do not have it.

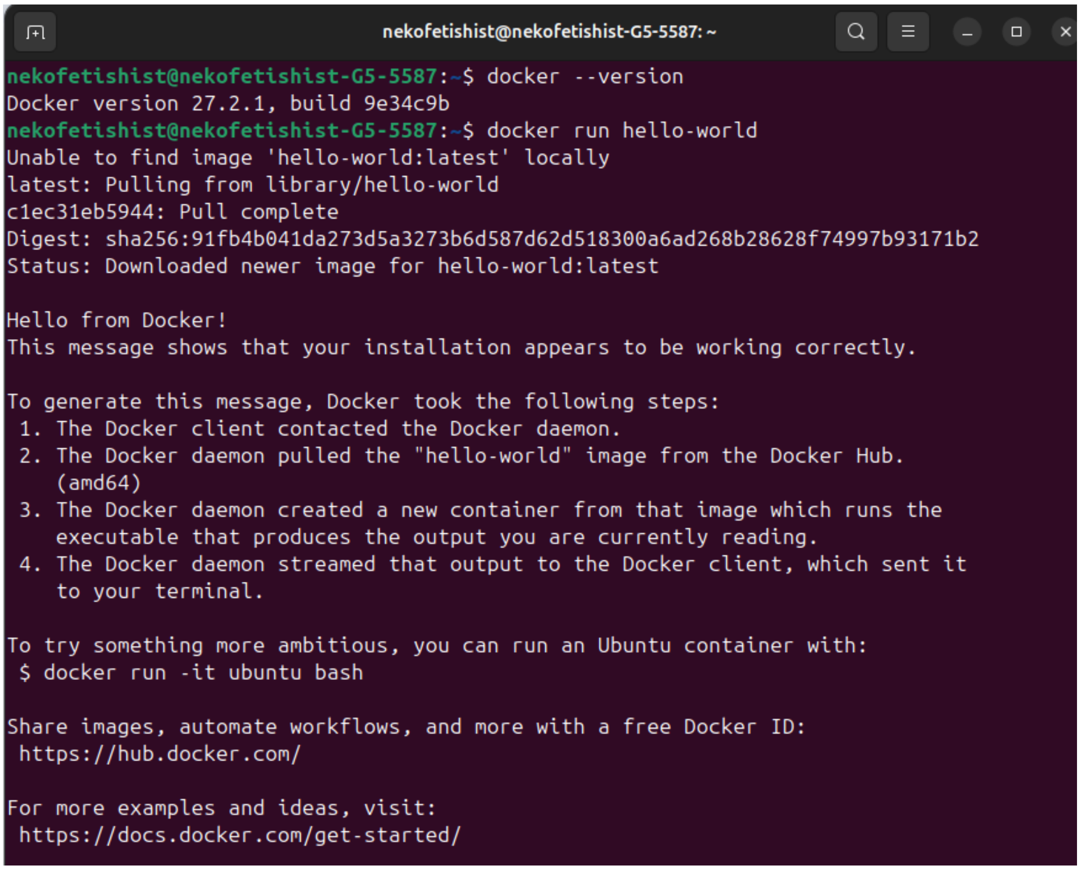


Figure 1.3 - Logs after running 'hello-world' container

Now, we need to provide answers for the following questions:

1. What are the key components of Docker (e.g., Docker Engine, Docker CLI)?

The answer is that docker has several components (Docker engine, Docker CLI, Docker image, Docker container, Docker registry, Docker hub, Docker network, etc.). Docker Engine is a core part that is responsible for running and managing containers.

It consists of 2 parts: Docker daemon (docker's server side) and Docker API (interface that communicates between Docker Daemon and other Docker components). Docker CLI or Docker Client or Docker Command-Line-Interface is used to work with Docker, for example, run containers, stop them, etc. Docker Image is a read-only template (e.g., template of ubuntu os, or python interpretator, etc.) with instructions to create a docker container. Dockerfile is your own image. Docker Container is a runnable instance of an image. By default, these containers are isolated from each other. Docker Registry is a place where you host/store different images or group os images. Docker Hub is a registry to store and share images. Docker Network is a way to provide isolation for docker containers. It is possible to set containers to see each other or be in the same network.

2. How does Docker compare to traditional virtual machines? **The answer is** that a traditional virtual machine's (VM's) operating system is commonly used with fully provided interface, while docker has only it is command line or could run in background. Moreover, VM operating system is much bigger in size, while docker uses containers that are lightweight. Therefore, VMs are much slower than docker containers. As VMs are completely isolated machines with their own os, they have their own kernel, but docker containers share the host's kernel.

3. What was the output of the docker run hello-world command, and what does it signify? **The answer is** that the first output of the command was that there was no hello-world image, and it started downloading it. After that it showed the message that docker is installed correctly and ready to work. Figure 1.4 shows the approach.

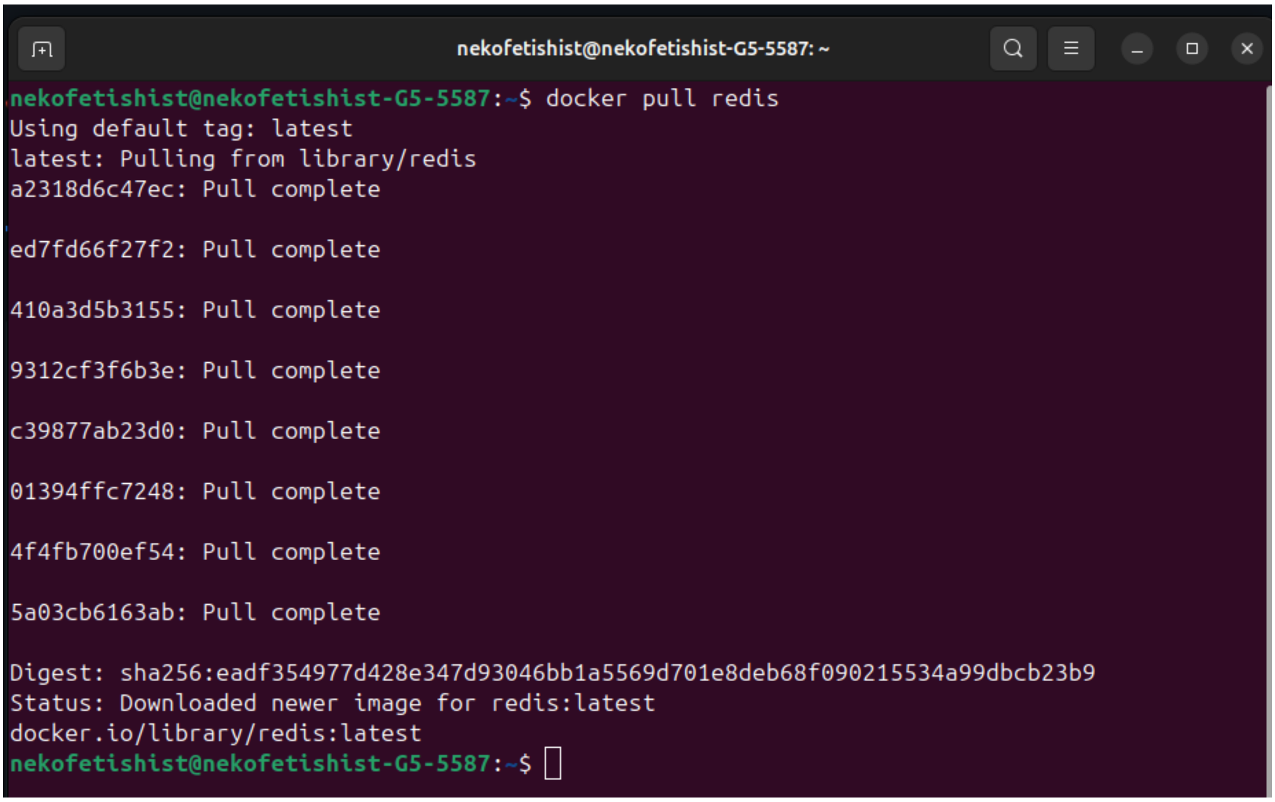


Figure 1.4 - The view of pulling an image which is absent on a local host

So, if there are no required image to run a container, it can download it by itself with run command.

1.2 Exercise 2: Basic Docker Commands

This section is responsible for learning and getting familiar with basic docker commands to run an application in isolated environment. For this exercise ‘redis’ image was used to be wrapped into container and then launched. If you want to download a ‘redis’ or whatever image you want to use in further work, you need to make sure that it is not installed. To view all images run ‘docker images’ command. The list of all installed images is shown in Figure 1.5.

```
nekofetishist@nekofetishist-G5-5587:~$ docker images
REPOSITORY                  TAG       IMAGE ID       CREATED          SIZE
dev-backend                 latest    48fa4cee05d4   14 hours ago     1.67GB
buil_with_cache_alpine      latest    f9e9c31c9b99   32 hours ago     716MB
buil_with_cache             latest    b8280fd4ce2e   32 hours ago     2.25GB
first_build                 latest    9625c0e6b21a   32 hours ago     2.14GB
redis                       latest    590b81f2fea1   7 weeks ago      117MB
docker/welcome-to-docker    latest    c1f619b6477e   10 months ago    18.6MB
hello-world                 latest    d2c94e258dcb   16 months ago    13.3kB
nekofetishist@nekofetishist-G5-5587:~$
```

Figure 1.5 - Example of view how list of installed docker’s images

As can be clearly seen, ‘redis’ container is now present in a list of installed images. To run this container, use the command ‘docker run –d redis.’ The flag ‘-d’ means to run a container in background. The result is a hash string of a launched container. Figure 1.6 illustrates the results.

```
nekofetishist@nekofetishist-G5-5587:~$ docker run -d redis
c70b6a311914c9981f9b5871119b97abdc8dfd18755de6608074e576bc3e4a0f
nekofetishist@nekofetishist-G5-5587:~$
```

Figure 1.6 - Hash string of a run container in detached mode

We can now see that ‘redis’ container is present in a list of launched containers. To see the results, run ‘docker ps’ command. Figure 1.7 contains this data.

```
nekofetishist@nekofetishist-G5-5587:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS      NAMES
c70b6a311914   redis     "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   6379/tcp   distracted_haibt
```

Figure 1.7 - The list of all containers which are running

To stop a container which is active, use ‘docker stop <container_id>’ (here container id is c70b6a311914) command. Figure 1.8 presents this step.

```
Error response from daemon: No such container: redis
nekofetishist@nekofetishist-G5-5587:~$ docker stop c70b6a311914
c70b6a311914
```

Figure 1.8 - Result after stopping docker container

Now, we need to answer the following questions:

1. What is the difference between docker pull and docker run? **The answer is** that Docker pull command is responsible for downloading images from remote repository (e.g., Docker hub) to a local machine. Docker run command is used to launch a container using image (image cannot be installed locally, then this command triggers immediate download if it is absent)

2. How do you find the details of a running container, such as its ID and status? **Ther answer is** that using command ‘docker ps’ or docker container ls we can see the details of runnable containers or if we add -a flag in the end we can see all the containers (their id, used image, command, created when, status, ports, and names).

3. What happens to a container after it is stopped? Can it be restarted? **The answer is** that when container is stopped using stop command, it means that the process which was working also interrupted, but its configuration, volumes are maintained the same. It is just not running in the system. The container can be restarted using the command ‘docker start <container_name_or_id>’.

1.3 Exercise 3: Working with Docker Containers

The aim of this subsection is to know how to manage Docker containers, work in an interactive mode, etc.

```
nekofetishist@nekofetishist-G5-5587:~$ docker run -d -p 8080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
a2318d6c47ec: Already exists
095d327c79ae: Pull complete
bbfaa25db775: Pull complete
7bb6fb0cfb2b: Pull complete
0723edc10c17: Pull complete
24b3fdc4d1e3: Pull complete
3122471704d5: Pull complete
Digest: sha256:04ba374043ccd2fc5c593885c0eacddebabd5ca375f9323666f28dfd5a9710e3
Status: Downloaded newer image for nginx:latest
5ec85597434f9d5f8ca0259f6b01399d2ab75afb37cd8720260a13d8ad263361
nekofetishist@nekofetishist-G5-5587:~$
```

Figure 1.9 - View of a process to launch ‘nginx’ container

We needed to start a new container using ‘nginx’ image and map port 8080 on our local host to a port 80 in the container. To do it we can apply ‘docker run –d –p 8080:80 nginx’ command, where ‘-d’ flag has already been introduced, while ‘-p’ is to map 8080 port of a localhost to 80 of an isolated container. Figure 1.10 shows that ‘nginx’ is available on port 8080.

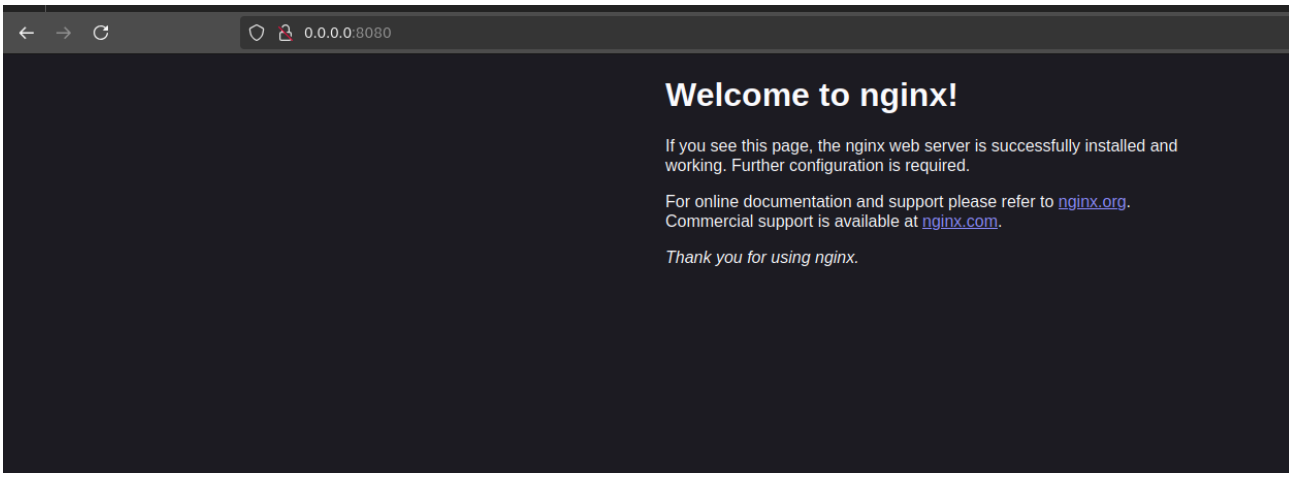


Figure 1.10 - Result of a reached URL

If we want to work with this container in an interactive mode, we can use ‘docker exec –it <container_id> <path>’ command and work from that path manually. Figure 1.11 contains this approach results.

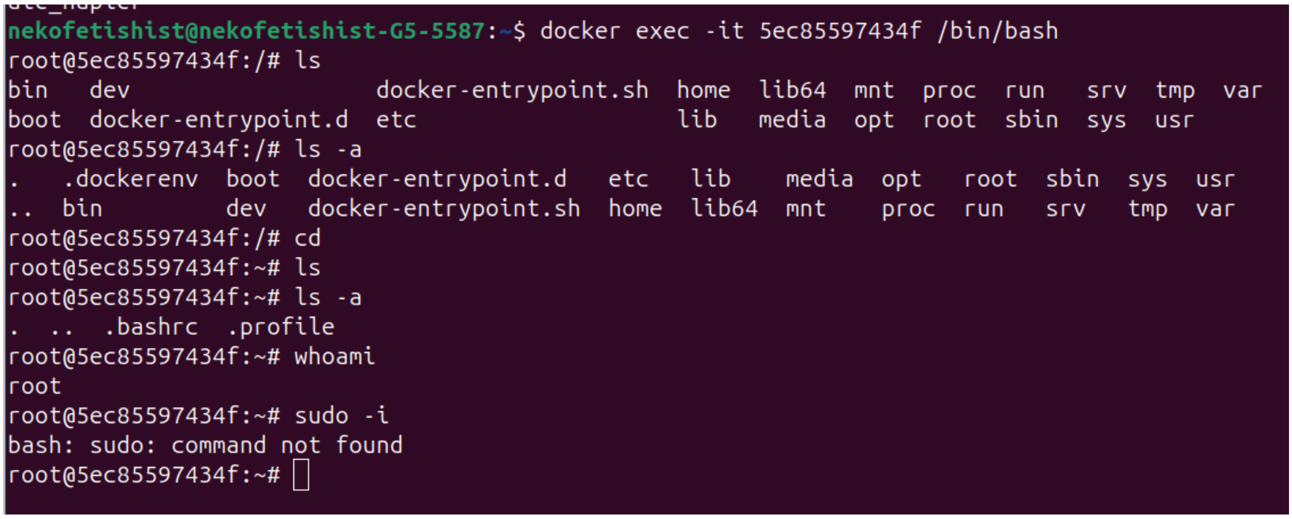


Figure 1.11 - View of working from a container in an interactive mode

Then, we can remove the executing container. However, before the removal, we must ensure it is not active. Run ‘docker ps’ to see all run docker containers. As it is present there, we need to interrupt it with the special command ‘docker stop <container_id>’ from a command line. As we stopped the container, we can simply get rid of it by applying ‘docker rm <container_id>’ command. Figure 1.12 illustrates this series of actions.

```
nekofetishist@nekofetishist-G5-5587:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS
5ec85597434f   nginx     "/docker-entrypoint.…"   5 minutes ago   Up 5 minutes   0.0.0.0:808
ate_napier
nekofetishist@nekofetishist-G5-5587:~$ docker stop 5ec85597434f
5ec85597434f
nekofetishist@nekofetishist-G5-5587:~$ docker rm 5ec85597434f
5ec85597434f
nekofetishist@nekofetishist-G5-5587:~$ 
```

Figure 1.12 - Series of steps to delete an executable container

To finish this section, we need to cover the following questions:

1. How does port mapping work in Docker, and why is it important? **The answer is** that port mapping is a Docker feature that allows us to connect a real host’s port to an isolated virtual port of a run container using which we reach it from outside. It is especially important, because for example, if we have a back-end application which is containerized using Docker, we want to reach our endpoints in a Docker using 8000 port. We can do it by exposing this port. Or if we do not have a database, we can set e.g., postgres on 5432 in container.

2. What is the purpose of the docker exec command? **The answer is** that Docker exec command is responsible for interacting with run container and does not stop it. Using it we can reach container environment and not restart it. We can do the following: run cmd commands inside, debug, etc.

3. How do you ensure that a stopped container does not consume system resources? **The answer is** that if a docker container is running it consumes our RAM and HHS (SSD) disc space as it has storage, but as it is stopped RAM consuming is stopped while disc storage is still occupied. To be sure that it is free you can delete a container using ‘docker rm <container_id_or_name>’ to get rid of it or using ‘docker run –rm' to be sure that container is deleted as it is stopped.

2. Dockerfile

This section is aimed to write a dockerfile to containerize a basic application, optimize this image with layers, caching it and get familiar with multi-stage builds. Additionally, we will publish this image to Docker hub.

2.1 Exercise 1: Creating a Simple Dockerfile

At a first step we need to create a simple program which prints ‘Hello, Docker!’ in a cmd. The script was written in Python language. Figure 2.1 represents this step.

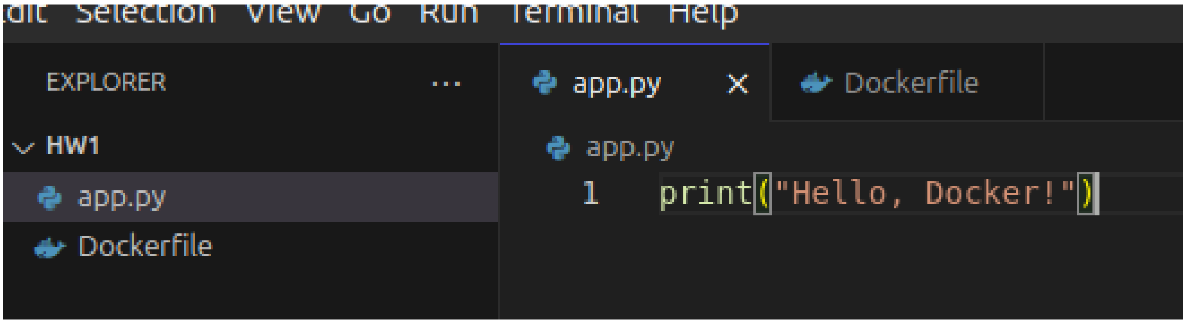


Figure 2.1 - App which we want to containerize

Next, we need to prepare a dockerfile to containerize it. The dockerfile with its series of steps is shown in Figure 2.2.

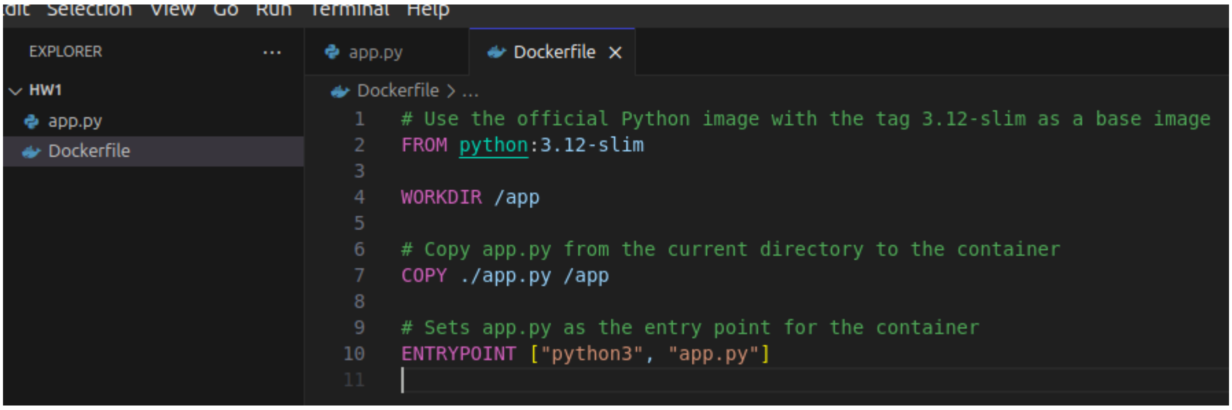


Figure 2.2 - Dockerfile scenario

To build a container, run ‘docker build –t hello-docker .’. Figure 2.3 shows it.

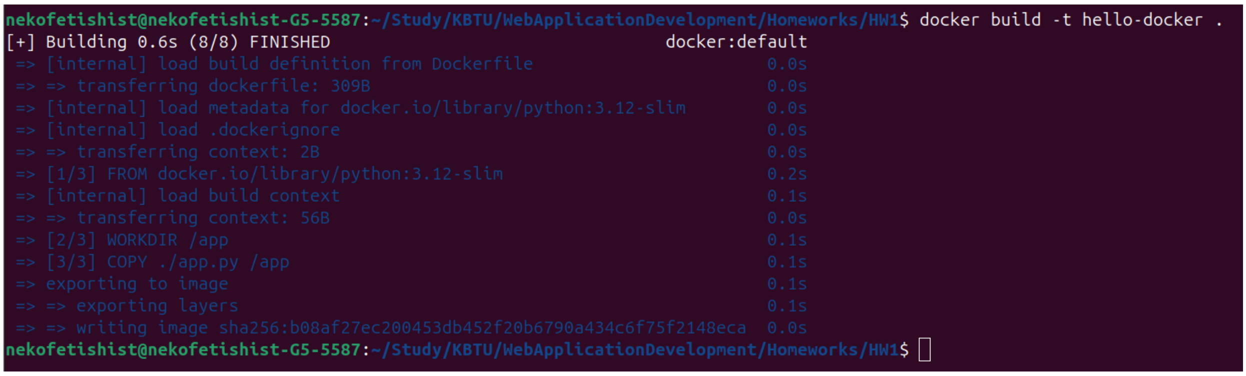


Figure 2.3 - Command to build a ‘hello-docker’ image locally

The program is written in Python programming language. Now, we will run a container based on a ‘hello-docker’ container using ‘docker run hell0-docker’ command. Figure 2.4 demonstrates this step.

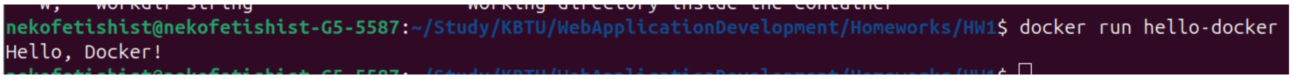


Figure 2.4 - Launch of a ‘hello-docker’ container

Now, we need to provide answers for the following issues:

1. What is the purpose of the FROM instruction in a Dockerfile? **The answer is** that FROM instruction is responsible to specify what image would be used as a base for a container. It is a starting point which says what image to use to build my own image with my own instructions. Images could be built as layers upon other images step by step e.g.: ubuntu → python3, etc.

2. How does the COPY instruction work in Dockerfile? **The answer is** that COPY is used to make a copy and move specified files, or directories from host machine into the container’s image from specified source folder to a destination one.

3. What is the difference between CMD and ENTRYPOINT in Dockerfile? **The answer is** that CMD command could be overridden using docker run command while ENTRYPOINT could not be so simple. CMD is used to set default parameters.

2.2 Exercise 2: Optimizing Dockerfile with Layers and Caching

This subsection will show how to optimize a Dockerfile for smaller image sizes and faster builds. Let us use the Dockerfile instructions from Figure 2.5.

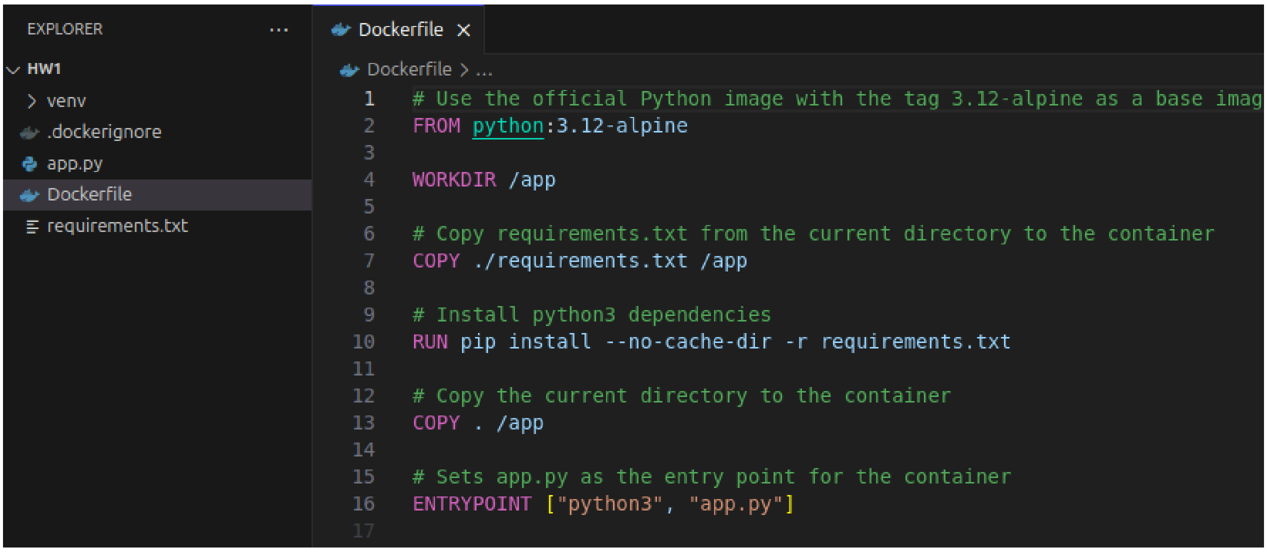


Figure 2.5 - Dockerfile with instructions to make it more optimized

First of all, to optimize the entire process we use here the lightest version of a python image (with alpine tag) making it less in size. Set /app as a working directory of a future container, copy the project lib requirements to this working directory, then, run the installation of these packages, however, add flag '--no-cache-dir' in order not to create cached files. Following that, copy everything from a local directory to a working directory. Finally, execute this app.py file. Pay attention that we have '.dockerignore' file which contains the list of files' names that we want to exclude. Figure 2.6 represents '.dockerignore' file's content.

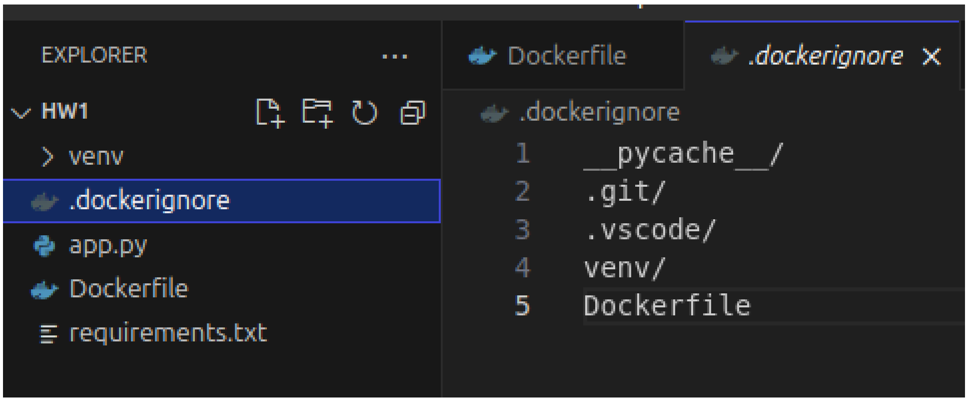


Figure 2.6 - Content of a '.dockerignore' file

According to a figure, we exclude all '__pycache__' dirs. along with their entire files, '.git', '.vscode', '.venv' folders as well as Dockerfile by itself to save space. Next, let us compare images' sizes. Run 'docker images' command. The comparison of images is shown in Figure 2.7.

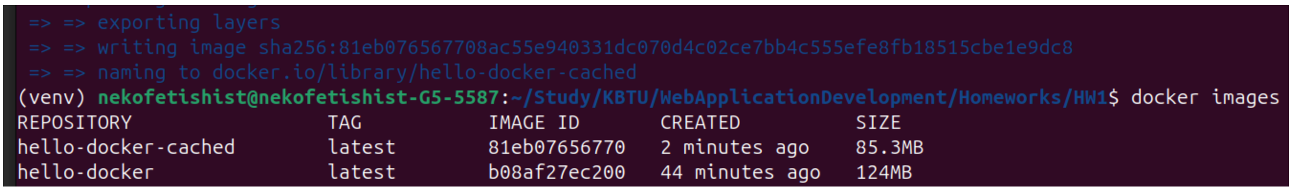


Figure 2.7 - Comparison of images with optimized approach and not

It can be clearly seen that the difference between images 'hello-docker-cached' (optimized version) and 'hello-docker' (initial version) is about 60 mb which is about 85-90 %.

We need to answer the following issues:

1. What are Docker layers, and how do they affect image size and build times? **The answer is** that each raw with command (RUN, FROM, etc.) in Dockerfile could be considered as a layer. Each time we build a docker file it goes through each line and

gets the hash of each executed command. As the hash maintains the same it uses cached command execution making the build process much faster by skipping these layers. It affects the image size, as each non cached executed layer increases the overall image size.

2. How does Docker's build cache work, and how can it speed up the build process? **The answer is** that as I mentioned in the above answer, as the hash of an executed command maintains the same, it uses cached variant and skips layer step making the build process much faster. For example, if you first copy dependencies (requirements.txt) and install them (RUN pip install), and your dependencies do not change, Docker will reuse the cached layer for that step in future builds

3. What is the role of the .dockerignore file? **The answer is** that the role if .dockerignore file is the same as for '.gitignore'. It has a finite number of patterns name (could be names of folders, files, or regular expression pattern for specific files). Having these names, docker will ignore these files or folders and will not copy them during the built state. It can reduce overall image size, increase build process.

2.3 Exercise 3: Multi-Stage builds

This subsection is responsible for understanding how to use multi-stage builds to create leaner Docker images. The reason for using multi-stage builds is to create a docker file which finally uses the smallest image to just only run binary files with app's logic. This task was achieved by firstly creating simple application to print 'Hello, World!' in Golang language. Figure 2.8 represents the app's code.

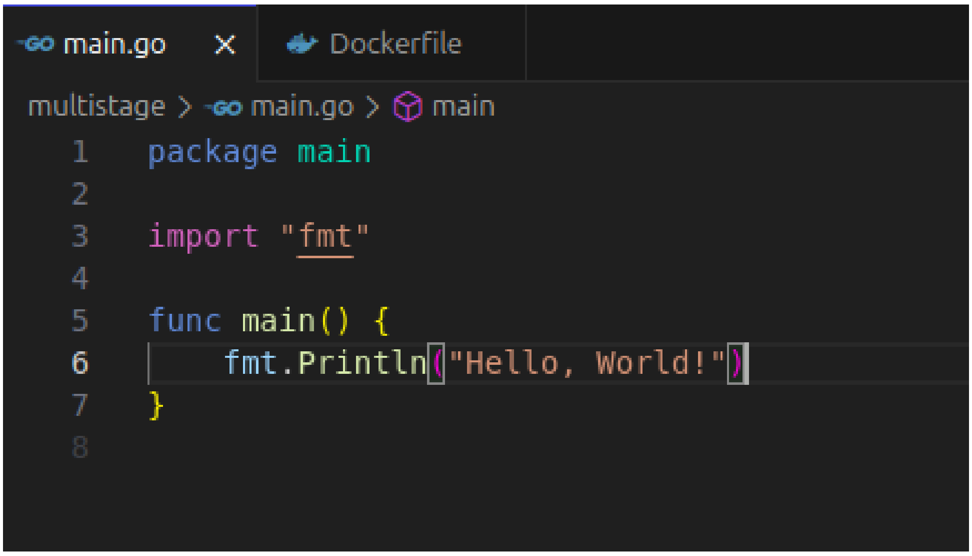


Figure 2.8 - Program's code in Go language

As we completed 'main.go' file we can start filling 'Dockerfile' with specific instructions. The 'Dockerfile' with written series of instructions if provided in Figure 2.9. We initially created a simple 'Dockerfile' with complete image and all actions.

```dockerfile
# Use golang as the base image
FROM golang AS builder

WORKDIR /app

# Copy main.go from the current directory to the container
COPY ./main.go /app

# Build the Go application
RUN go build main.go

# Sets main as the entry point for the container to run the Go application
ENTRYPOINT ["/app/main"]
```

Figure 2.9 - Dockerfile with all instructions and complete image

Now, let us modify this file with the usage of multi-stages. Figure 2.10 shows the details of these modifications.

```dockerfile
# Use golang as the base image
FROM golang AS builder

WORKDIR /app

# Copy main.go from the current directory to the container
COPY ./main.go /app

# Build the Go application
RUN go build main.go

# Use alpine as the base image
FROM golang:alpine AS runner

# Set the working directory
WORKDIR /app

# Copy the binary from the builder stage to the runner stage
COPY --from=builder /app/main /app

# Run the binary
ENTRYPOINT [ "/app/main" ]
```

Figure 2.10 - Edited Dockerfile with multi-stage building approach

The whole ‘Dockerfile’ actions are divided into 2 group of steps: build a binary file using complete image and then copy this file into a new limited image which has only opportunity to launch bin file (‘main’). Finally, we get a noticeably light and quickly built ‘dockerfile’ as we used exceedingly small Go image (alpine) to run only binary file. The comparison of these two built images is shown in Figure 2.11.

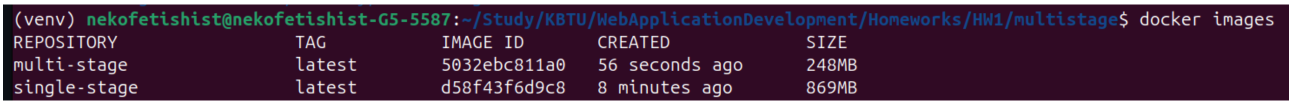


Figure 2.11 - Comparison of images with multi-stage building and not

The difference between ‘single-stage’ (initial image build) and ‘multi-stage’ (final image build) is almost 600 mb.

Let us answer the following questions:

1. What is the benefit of using multi-stage builds in Docker? **The answer is** that using multi-stage builds in Docker we can reduce the overall docker image because in the last stage it usually has only small images (like alpine) and needed packages to exactly run only the project. As it has only necessary modules it is the smallest size. We can have multiple FROM images to finally use the smallest one.

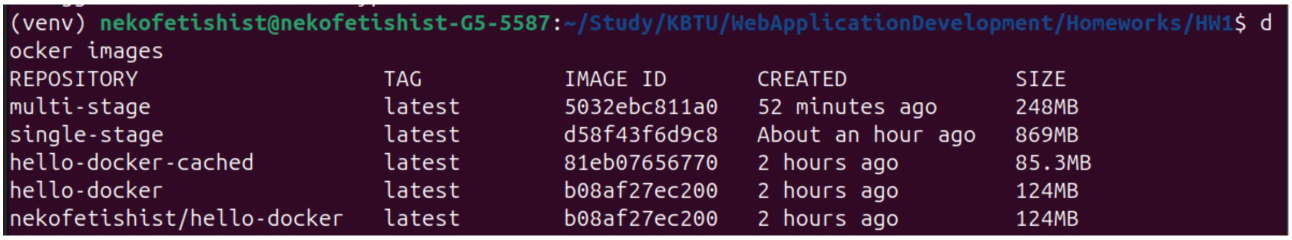
2. How can multi-stage build help reduce the size of Docker images? **The answer is** that all the stages (excepting the last one) are used for building or compiling applications. While the last one is applied for copying all the required files from the above stages and only running the app. That is why the final image has the lowest size. Also, the last stage usually uses the lightest image like alpine.

3. What are some scenarios where multi-stage builds are particularly useful? **The answer is** that multi-stage builds are extremely useful for those programming languages which generate binary files that are used in further work. These languages are Golang, C++, etc. So, the first scenario is to build binary files which are then just copied to the last stage. Another scenario is about static files. In web (Django), multi-stage is used to compile and collect static assets (JS, HTML, CSS).

2.4 Exercise 4: Docker hub

The final subsection gives an opportunity to learn how to push a built docker image to a remote repository (Docker hub) where it is publicly available. We will use ‘hello-docker’ image which we built earlier for this purpose because the process is the same, the difference is only in names and tags that you want to use. To make sure that we have a ‘hello-docker’ image and do not lose it use the command ‘docker images’. The command must return a list of rows where ‘hello-docker’ repository must be present. Alternatively, we can run a command with the same behavior. The command is ‘docker image ls’. Additionally, do not forget to create an account in Docker Hub.

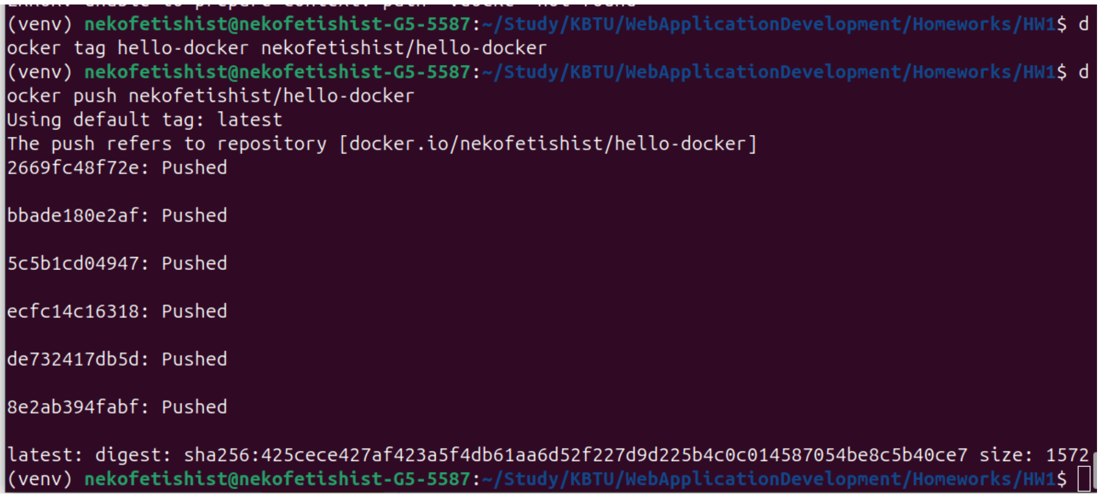
As you do it, login into the system with the command ‘docker login’. Figure 2.12 shows the list of existing images.



```
(venv) nekofetishist@nekofetishist-G5-5587:~/Study/KBTU/WebApplicationDevelopment/Homeworks/HW1$ d
ocker images
REPOSITORY                 TAG       IMAGE ID       CREATED            SIZE
multi-stage                latest    5032ebc811a0   52 minutes ago     248MB
single-stage               latest    d58f43f6d9c8   About an hour ago  869MB
hello-docker-cached        latest    81eb07656770   2 hours ago        85.3MB
hello-docker               latest    b08af27ec200   2 hours ago        124MB
nekofetishist/hello-docker latest    b08af27ec200   2 hours ago        124MB
```

Figure 2.12 - List of existing images on a local machine

The next step is to tag the docker image. To reach this target use the following command ‘docker tag hello-docker nekofetishist/hello-docker’. Instead of a ‘hello-docker’ and ‘nekofetishist’ you could use your image name and username in Docker Hub, respectively. As you did it, please, run the command ‘docker push nekofetishist/hello-docker’. See Figure 2.13.



```
(venv) nekofetishist@nekofetishist-G5-5587:~/Study/KBTU/WebApplicationDevelopment/Homeworks/HW1$ d
ocker tag hello-docker nekofetishist/hello-docker
(venv) nekofetishist@nekofetishist-G5-5587:~/Study/KBTU/WebApplicationDevelopment/Homeworks/HW1$ d
ocker push nekofetishist/hello-docker
Using default tag: latest
The push refers to repository [docker.io/nekofetishist/hello-docker]
2669fc48f72e: Pushed

bbade180e2af: Pushed

5c5b1cd04947: Pushed

ecfc14c16318: Pushed

de732417db5d: Pushed

8e2ab394fabf: Pushed

latest: digest: sha256:425cece427af423a5f4db61aa6d52f227d9d225b4c0c014587054be8c5b40ce7 size: 1572
(venv) nekofetishist@nekofetishist-G5-5587:~/Study/KBTU/WebApplicationDevelopment/Homeworks/HW1$
```

Figure 2.13 - Results of tagging and pushing docker image

We need to verify that the image is present on a Docker Hub under our user. The results are shown in Figure 2.14.

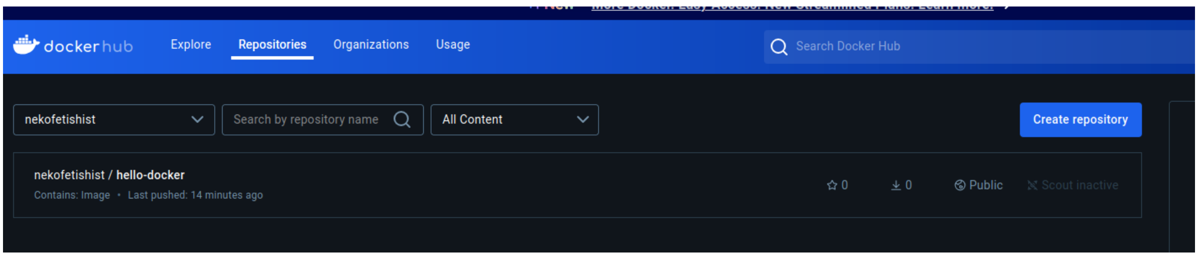


Figure 2.14 - Publicly available docker image to a Docker Hub

Let us answer the following questions:

1. What is the purpose of Docker Hub in containerization? **The answer is** that Docker Hub is a cloud-based repository where images are stored and can be shared. That is why the main purpose is to store docker images (after setting tag and pushing them).

2. How do you tag a Docker image for pushing to a remote repository?

The answer is that to tag a docker image we need to use the following command: ‘docker tag <image_name>:<local_tag> <username_in_docker_hub>/<repository_name>:<tag>’

3. What steps are involved in pushing an image to Docker Hub? **The answer is** to push an image into a Docker hub we need to follow the next steps:

3.1 Register an account in Docker hub

3.2 Login from cli using command: docker login

3.3 Create a docker file with your own instructions

3.4 Build this docker image using command: ‘docker build <image_name> <path>’

3.5 Tag this docker image using command: ‘docker tag <image_name>:<local_tag> <username_in_docker_hub>/<repository_name>:<tag>’

3.6 Push the image using command: ‘docker push <username_in_docker_hub>/<repository_name>:<tag>’

CONCLUSION

In conclusion, we managed to build simple dockerfiles for different apps, run them and push to a Docker hub making it public. Moreover, the usage of the smallest images and multi-stage building let us construct very lightweight and quick containers with only all needed configuration to launch an app.