Assignment №1

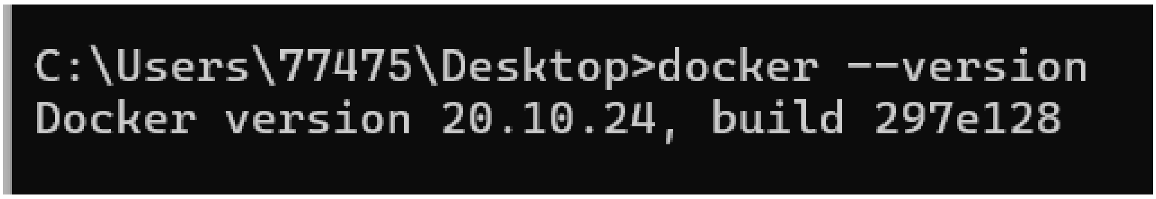
Almabekuly Aibek

Almaty, 2024

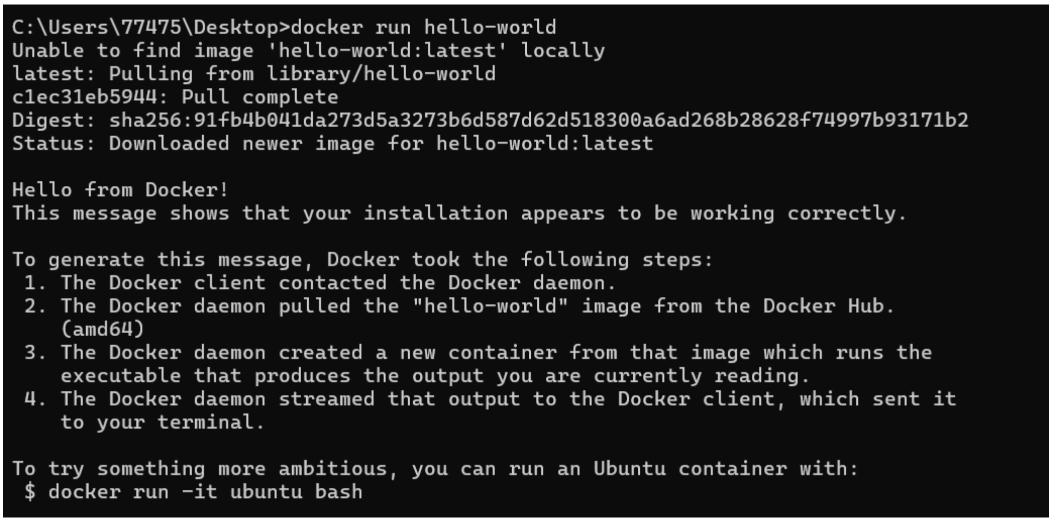
Exercise 1: Installing Docker

1. Objective: Install Docker on your local machine.
2. Steps:
    - Follow the installation guide for Docker from the official website, choosing the appropriate version for your operating system (Windows, macOS, or Linux).
    - After installation, verify that Docker is running by executing the command docker --version in your terminal or command prompt.
    - Run the command docker run hello-world to verify that Docker is set up correctly.
3. Questions:
    - What are the key components of Docker (e.g., Docker Engine, Docker CLI)?
    - How does Docker compare to traditional virtual machines?
    - What was the output of the docker run hello-world command, and what does it signify?

Result:



```
C:\Users\77475\Desktop>docker --version
Docker version 20.10.24, build 297e128
```



```
C:\Users\77475\Desktop>docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:91fb4b041da273d5a3273b6d587d62d518300a6ad268b28628f74997b93171b2
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
```

Key components of Docker:

- Docker Engine, Docker CLI, Docker Daemon, Docker HUB

Docker vs. traditional virtual machines:

- Docker uses containerization, sharing the host OS kernel, which makes it more lightweight and faster than traditional VMs that require full OS installations.

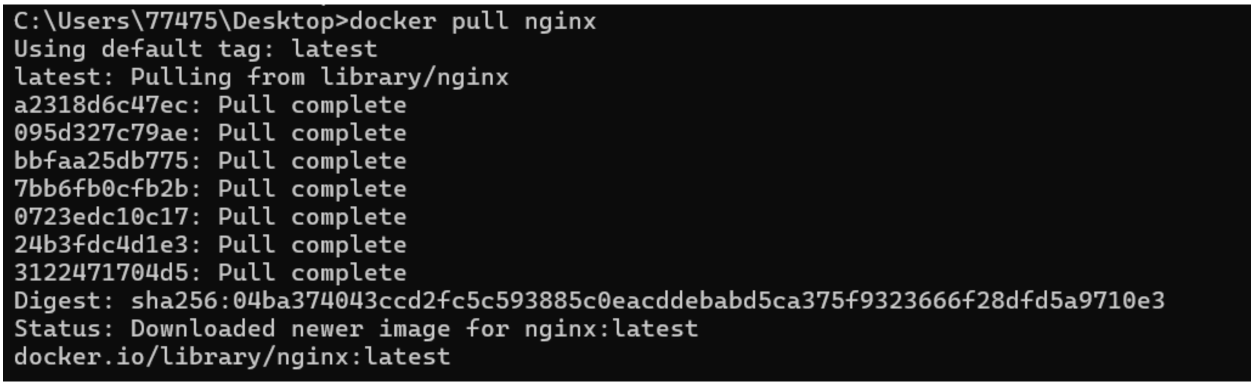
Output of docker run hello-world:

- The output confirms Docker is installed and working correctly by printing a message. It indicates the container successfully ran and connected to Docker Engine.

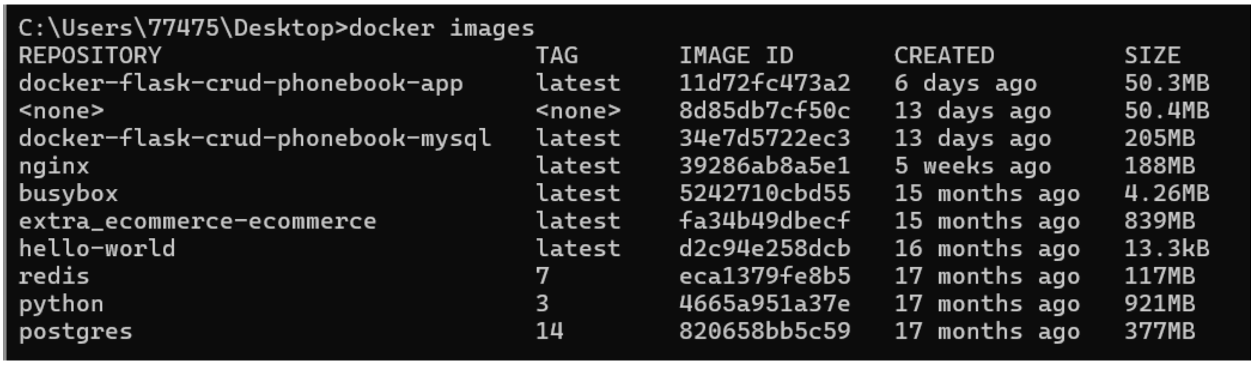
Exercise 2: Basic Docker Commands

1. Objective: Familiarize yourself with basic Docker commands.
2. Steps:
   - Pull an official Docker image from Docker Hub (e.g., nginx or ubuntu) using the command docker pull <image-name>.
   - List all Docker images on your system using docker images.
   - Run a container from the pulled image using docker run -d <image-name>.
   - List all running containers using docker ps and stop a container using docker stop <container-id>.
3. Questions:
   - What is the difference between docker pull and docker run?
   - How do you find the details of a running container, such as its ID and status?
   - What happens to a container after it is stopped? Can it be restarted?

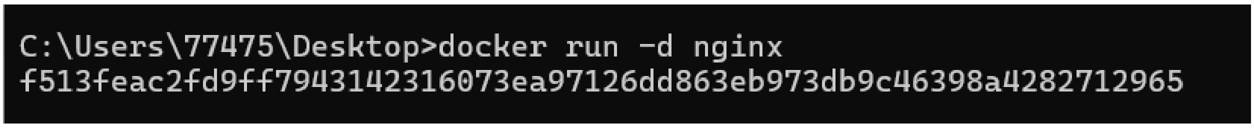
Result:



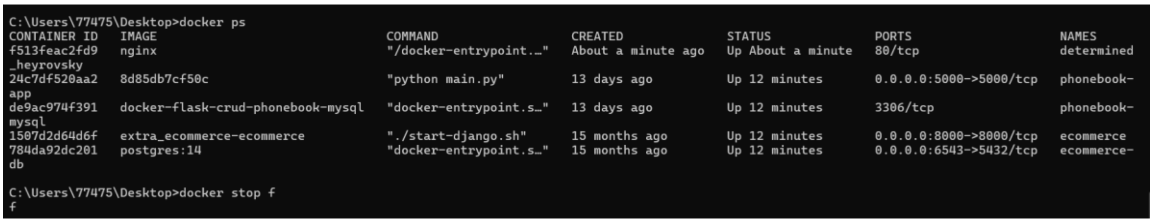
```
C:\Users\77475\Desktop>docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
a2318d6c47ec: Pull complete
095d327c79ae: Pull complete
bbfaa25db775: Pull complete
7bb6fb0cfb2b: Pull complete
0723edc10c17: Pull complete
24b3fdc4d1e3: Pull complete
3122471704d5: Pull complete
Digest: sha256:04ba374043ccd2fc5c593885c0eacddebabd5ca375f9323666f28dfd5a9710e3
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```



```
C:\Users\77475\Desktop>docker images
REPOSITORY                         TAG       IMAGE ID       CREATED         SIZE
docker-flask-crud-phonebook-app    latest    11d72fc473a2   6 days ago      50.3MB
<none>                             <none>    8d85db7cf50c   13 days ago     50.4MB
docker-flask-crud-phonebook-mysql  latest    34e7d5722ec3   13 days ago     205MB
nginx                              latest    39286ab8a5e1   5 weeks ago     188MB
busybox                            latest    5242710cbd55   15 months ago   4.26MB
extra_ecommerce-ecommerce          latest    fa34b49dbecf   15 months ago   839MB
hello-world                        latest    d2c94e258dcb   16 months ago   13.3kB
redis                              7         eca1379fe8b5   17 months ago   117MB
python                             3         4665a951a37e   17 months ago   921MB
postgres                           14        820658bb5c59   17 months ago   377MB
```



```
C:\Users\77475\Desktop>docker run -d nginx
f513feac2fd9ff7943142316073ea97126dd863eb973db9c46398a4282712965
```



```
C:\Users\77475\Desktop>docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED              STATUS              PORTS                    NAMES
f513feac2fd9   nginx                              "/docker-entrypoint.…"   About a minute ago   Up About a minute   80/tcp                   determined
_heyrovsky
24c7df520aa2   8d85db7cf50c                       "python main.py"         13 days ago          Up 12 minutes       0.0.0.0:5000->5000/tcp   phonebook-
app
de9ac974f391   docker-flask-crud-phonebook-mysql  "docker-entrypoint.s…"   13 days ago          Up 12 minutes       3306/tcp                 phonebook-
mysql
1507d2d64d6f   extra_ecommerce-ecommerce          "./start-django.sh"      15 months ago        Up 12 minutes       0.0.0.0:8000->8000/tcp   ecommerce
784da92dc201   postgres:14                        "docker-entrypoint.s…"   15 months ago        Up 12 minutes       0.0.0.0:6543->5432/tcp   ecommerce-
db

C:\Users\77475\Desktop>docker stop f
f
```

Difference between docker pull and docker run:

docker pull downloads an image from a registry.

docker run creates and starts a container from an image (it pulls the image if not already present).

How to find details of a running container (ID and status):

Use docker ps to see the container ID, status, and other details of running containers.

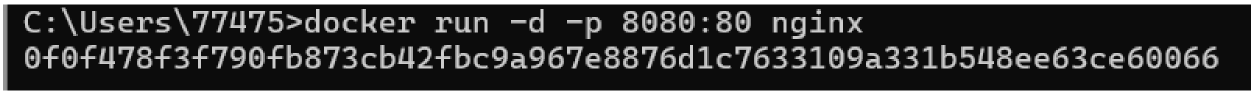
What happens after a container is stopped:

The container remains in a stopped state and can be restarted with docker start <container_id>.

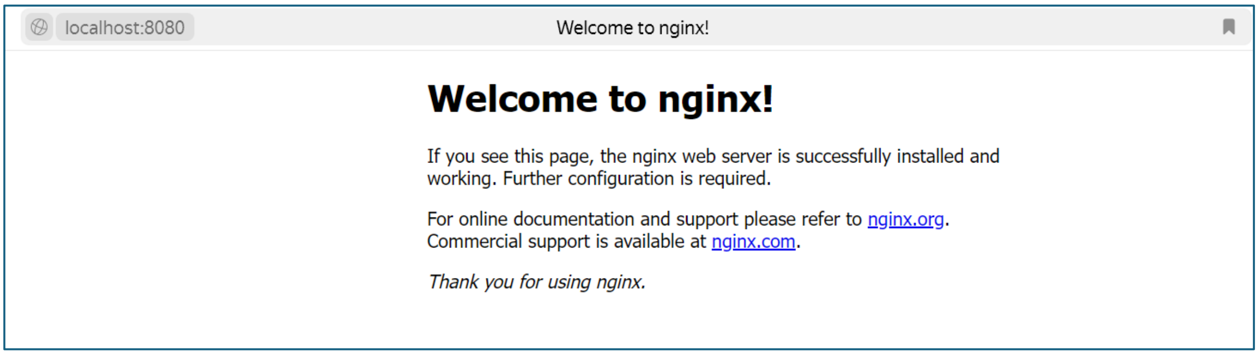
Exercise 3: Working with Docker Containers

1. Objective: Learn how to manage Docker containers.
2. Steps:
   - Start a new container from the nginx image and map port 8080 on your host to port 80 in the container using docker run -d -p 8080:80 nginx.
   - Access the Nginx web server running in the container by navigating to http://localhost:8080 in your web browser.
   - Explore the container's file system by accessing its shell using docker exec -it <container-id> /bin/bash.
   - Stop and remove the container using docker stop <container-id> and docker rm <container-id>.
3. Questions:
   - How does port mapping work in Docker, and why is it important?
   - What is the purpose of the docker exec command?
   - How do you ensure that a stopped container does not consume system resources?

Result:



```
C:\Users\77475>docker run -d -p 8080:80 nginx
0f0f478f3f790fb873cb42fbc9a967e8876d1c7633109a331b548ee63ce60066
```

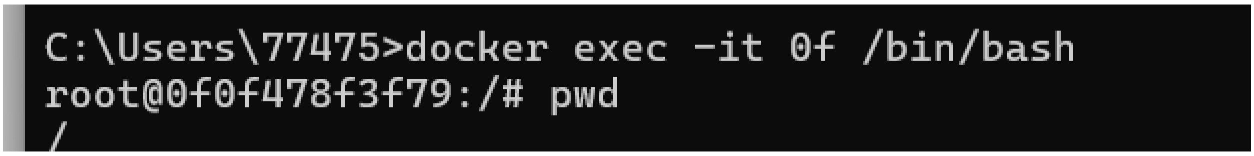


Welcome to nginx!

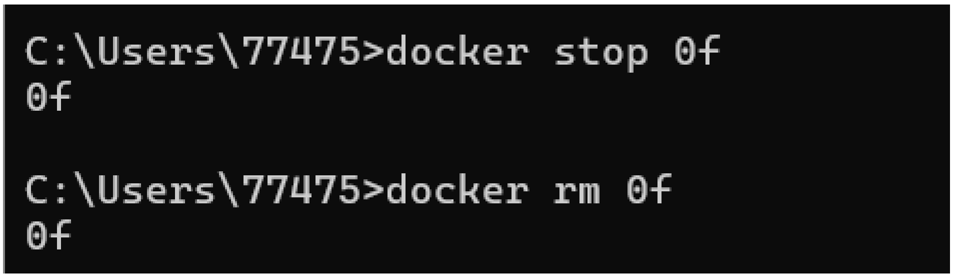
If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.



```
C:\Users\77475>docker exec -it 0f /bin/bash
root@0f0f478f3f79:/# pwd
/
```



```
C:\Users\77475>docker stop 0f
0f

C:\Users\77475>docker rm 0f
0f
```

Port mapping in Docker:

Port mapping links a port on your computer to a port inside the container, allowing outside access to the services running inside the container. For example, with -p 8080:80, port 8080 on the host is mapped to port 80 inside the container.

Purpose of docker exec:

This command allows you to run another command inside a running container, like opening a terminal or running a script inside the container.

Stopping a container from using resources:

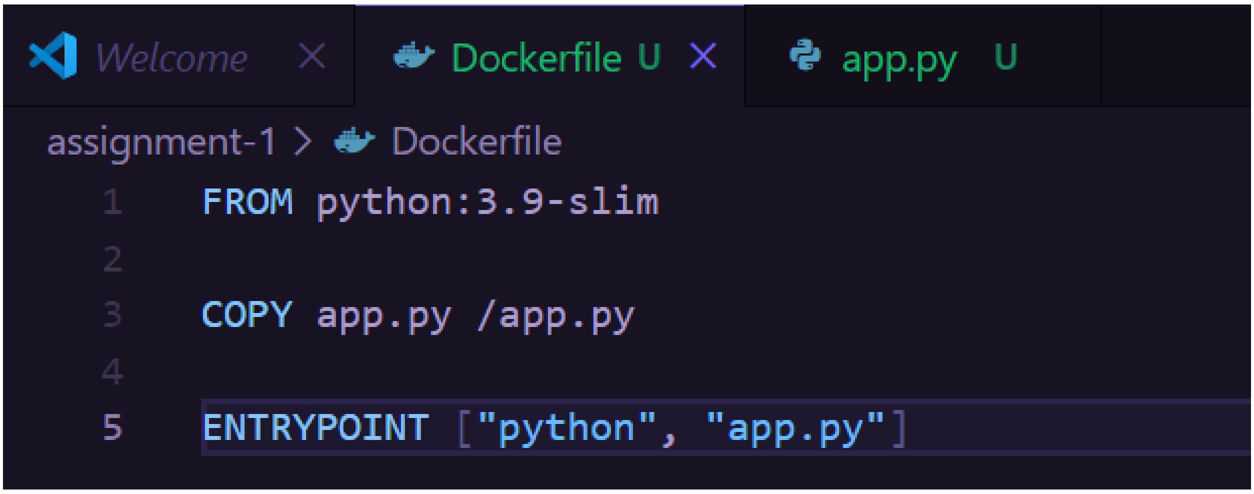
After stopping a container, you can delete it with docker rm to ensure it doesn’t take up any resources on the system.

Dockerfile

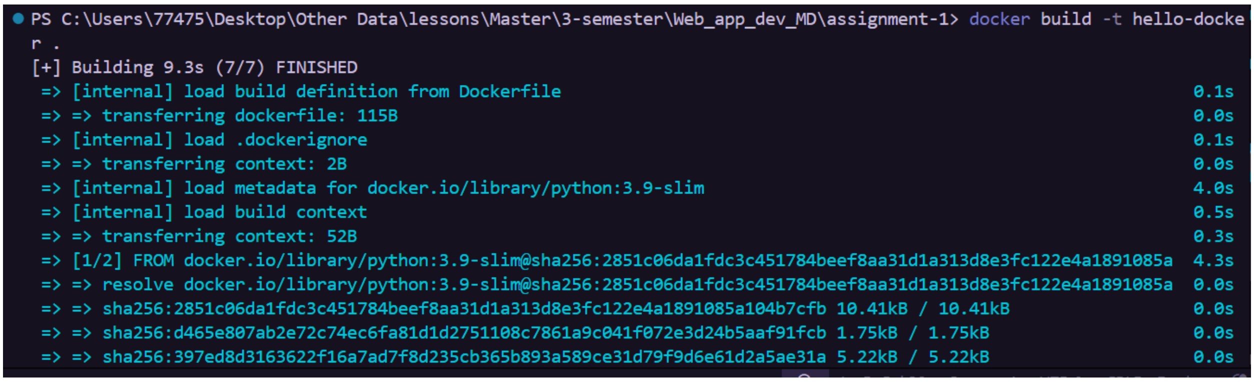
Exercise 1: Creating a Simple Dockerfile

1. Objective: Write a Dockerfile to containerize a basic application.
2. Steps:
    - Create a new directory for your project and navigate into it.
    - Create a simple Python script (e.g., app.py) that prints "Hello, Docker!" to the console.
    - Write a Dockerfile that:
        - Uses the official Python image as the base image.
        - Copies app.py into the container.
        - Sets app.py as the entry point for the container.
    - Build the Docker image using docker build -t hello-docker ..
    - Run the container using docker run hello-docker.
3. Questions:
    - What is the purpose of the FROM instruction in a Dockerfile?
    - How does the COPY instruction work in Dockerfile?
    - What is the difference between CMD and ENTRYPOINT in Dockerfile?

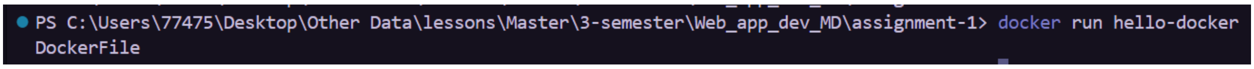
Result:



```
FROM python:3.9-slim

COPY app.py /app.py

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



```
PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1> docker build -t hello-docke
r .
[+] Building 9.3s (7/7) FINISHED
 => [internal] load build definition from Dockerfile                                                              0.1s
 => => transferring dockerfile: 115B                                                                              0.0s
 => [internal] load .dockerignore                                                                                 0.1s
 => => transferring context: 2B                                                                                   0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim                                                4.0s
 => [internal] load build context                                                                                 0.5s
 => => transferring context: 52B                                                                                  0.3s
 => [1/2] FROM docker.io/library/python:3.9-slim@sha256:2851c06da1fdc3c451784beef8aa31d1a313d8e3fc122e4a1891085a  4.3s
 => => resolve docker.io/library/python:3.9-slim@sha256:2851c06da1fdc3c451784beef8aa31d1a313d8e3fc122e4a1891085a  0.0s
 => => sha256:2851c06da1fdc3c451784beef8aa31d1a313d8e3fc122e4a1891085a104b7cfb 10.41kB / 10.41kB                  0.0s
 => => sha256:d465e807ab2e72c74ec6fa81d1d2751108c7861a9c041f072e3d24b5aaf91fcb 1.75kB / 1.75kB                    0.0s
 => => sha256:397ed8d3163622f16a7ad7f8d235cb365b893a589ce31d79f9d6e61d2a5ae31a 5.22kB / 5.22kB                    0.0s
```



```
PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1> docker run hello-docker
DockerFile
```

Purpose of the FROM instruction:

The FROM instruction specifies the base image to use for building the container. It sets up the foundation upon which the rest of the image is built.

How the COPY instruction works:

The COPY instruction copies files or directories from your local machine into the container's filesystem. For example, COPY app.py /app.py copies app.py from your project directory into the container.

Difference between CMD and ENTRYPOINT:

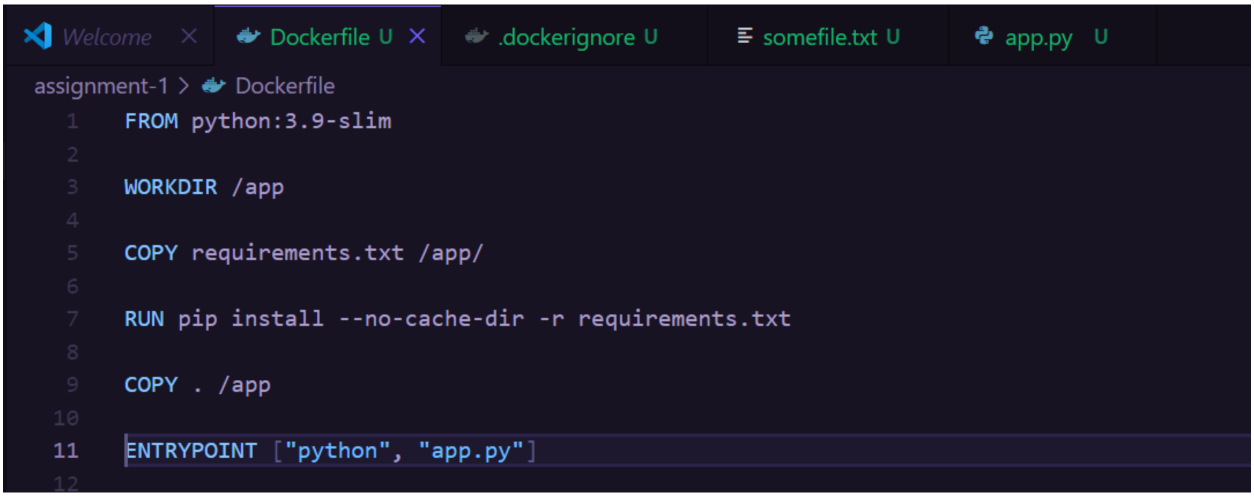
CMD defines the default command that runs when the container starts, but it can be overridden by passing a different command at runtime.

ENTRY POINT defines a fixed command that always runs when the container starts. It can't be overridden but can accept additional arguments.

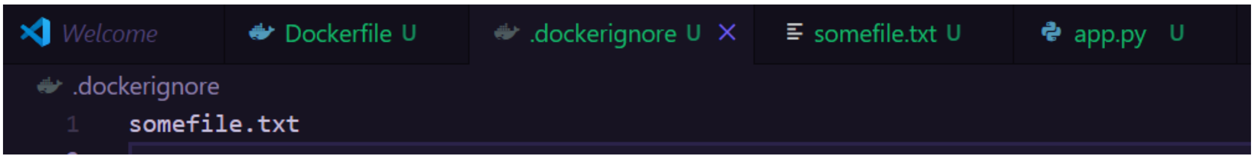
Exercise 2: Optimizing Dockerfile with Layers and Caching

1. Objective: Learn how to optimize a Dockerfile for smaller image sizes and faster builds.
2. Steps:
   - Modify the Dockerfile created in the previous exercise to:
     - Separate the installation of Python dependencies (if any) from the copying of application code.
     - Use a .dockerignore file to exclude unnecessary files from the image.
   - Rebuild the Docker image and observe the build process to understand how caching works.
   - Compare the size of the optimized image with the original.
3. Questions:
   - What are Docker layers, and how do they affect image size and build times?
   - How does Docker's build cache work, and how can it speed up the build process?
   - What is the role of the .dockerignore file?

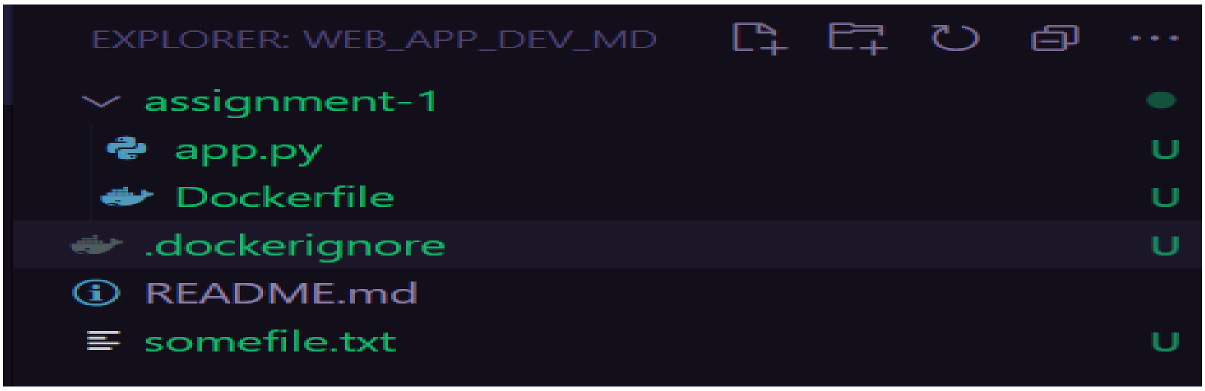
Result:

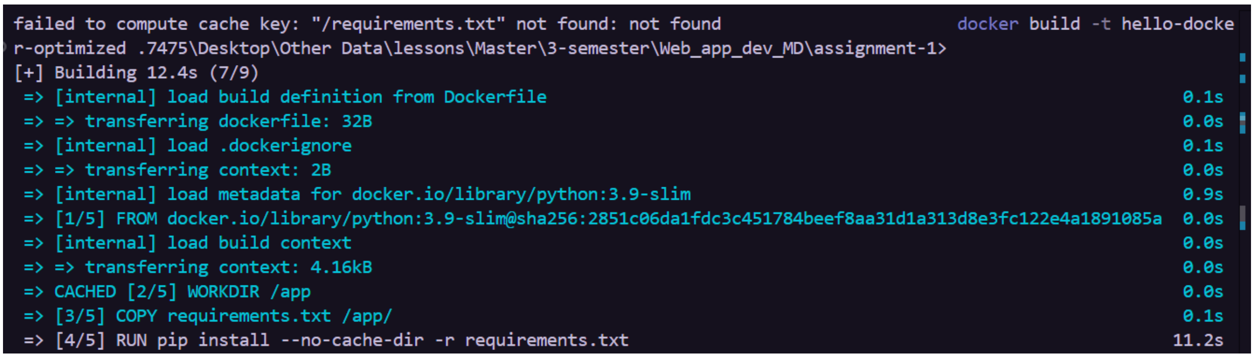


```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt /app/

RUN pip install --no-cache-dir -r requirements.txt

COPY . /app

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

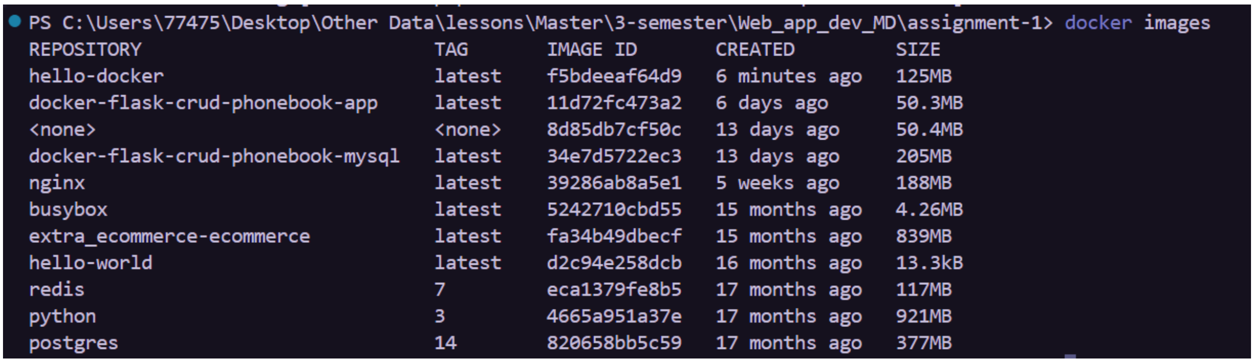


```
somefile.txt
```





```
failed to compute cache key: "/requirements.txt" not found: not found                 docker build -t hello-docke
r-optimized .7475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1>
[+] Building 12.4s (7/9)
 => [internal] load build definition from Dockerfile                                                         0.1s
 => => transferring dockerfile: 32B                                                                          0.0s
 => [internal] load .dockerignore                                                                            0.1s
 => => transferring context: 2B                                                                              0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim                                           0.9s
 => [1/5] FROM docker.io/library/python:3.9-slim@sha256:2851c06da1fdc3c451784beef8aa31d1a313d8e3fc122e4a1891085a  0.0s
 => [internal] load build context                                                                            0.0s
 => => transferring context: 4.16kB                                                                          0.0s
 => CACHED [2/5] WORKDIR /app                                                                                0.0s
 => [3/5] COPY requirements.txt /app/                                                                        0.1s
 => [4/5] RUN pip install --no-cache-dir -r requirements.txt                                                11.2s
```



```
PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1> docker images
REPOSITORY                      TAG       IMAGE ID       CREATED          SIZE
hello-docker                    latest    f5bdeeaf64d9   6 minutes ago    125MB
docker-flask-crud-phonebook-app latest    11d72fc473a2   6 days ago       50.3MB
<none>                          <none>    8d85db7cf50c   13 days ago      50.4MB
docker-flask-crud-phonebook-mysql latest  34e7d5722ec3   13 days ago      205MB
nginx                           latest    39286ab8a5e1   5 weeks ago      188MB
busybox                         latest    5242710cbd55   15 months ago    4.26MB
extra_ecommerce-ecommerce       latest    fa34b49dbecf   15 months ago    839MB
hello-world                     latest    d2c94e258dcb   16 months ago    13.3kB
redis                           7         eca1379fe8b5   17 months ago    117MB
python                          3         4665a951a37e   17 months ago    921MB
postgres                        14        820658bb5c59   17 months ago    377MB
```

What are Docker layers, and how do they affect image size and build times?

Docker layers are individual steps in the Dockerfile (like FROM, COPY, RUN). Each layer adds to the final image size. If a layer changes, all subsequent layers are rebuilt, impacting both image size and build time.

How does Docker's build cache work, and how can it speed up the build process?

Docker caches each layer during the build process. If a layer hasn’t changed, Docker uses the cached version instead of rebuilding it. This speeds up builds by only processing the layers that have been modified.

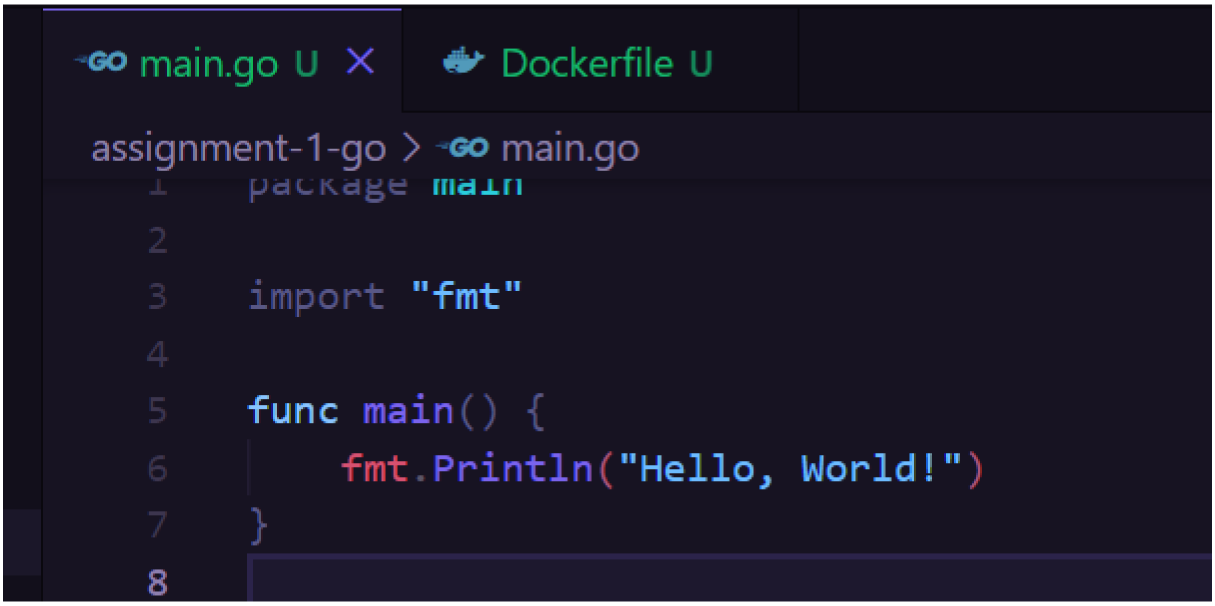
What is the role of the .dockerignore file?

The .dockerignore file excludes specified files and directories from being copied into the image during the build. This helps reduce image size, prevent unnecessary files from being included, and speeds up the build process.

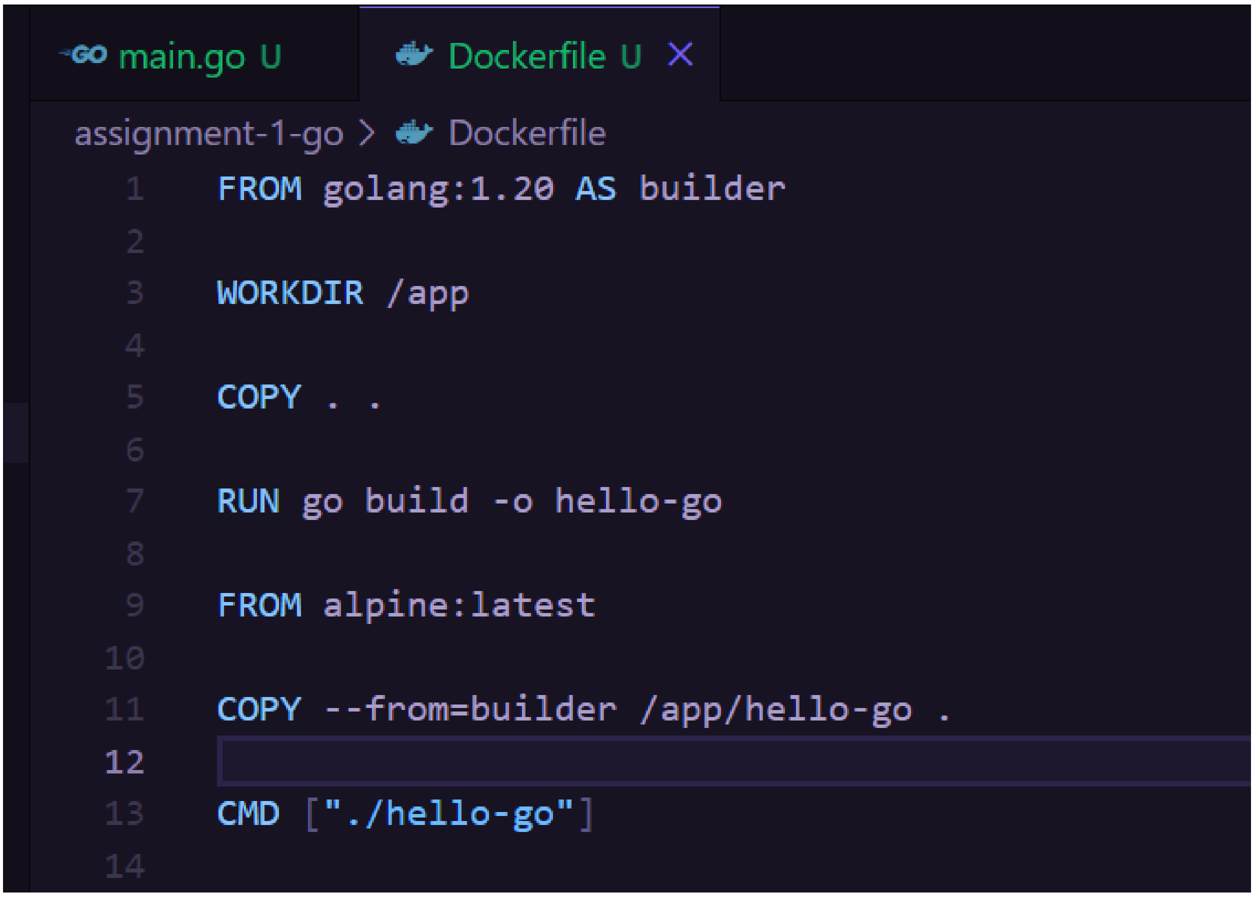
Exercise 3: Multi-Stage Builds

1. Objective: Use multi-stage builds to create leaner Docker images.
2. Steps:
   - Create a new project that involves compiling a simple Go application (e.g., a "Hello, World!" program).
   - Write a Dockerfile that uses multi-stage builds:
     - The first stage should use a Golang image to compile the application.
     - The second stage should use a minimal base image (e.g., alpine) to run the compiled application.
   - Build and run the Docker image, and compare the size of the final image with a single-stage build.
3. Questions:
   - What are the benefits of using multi-stage builds in Docker?
   - How can multi-stage builds help reduce the size of Docker images?
   - What are some scenarios where multi-stage builds are particularly useful?

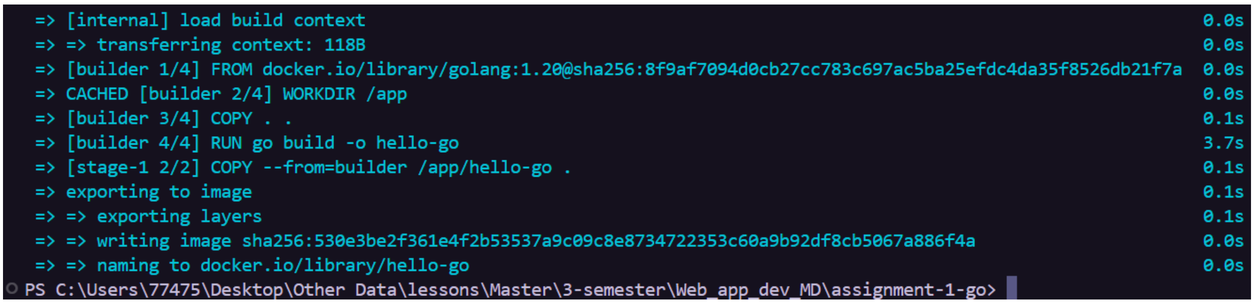
Result:



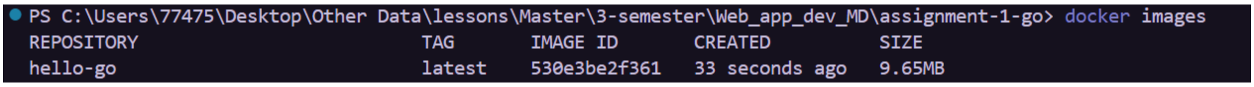
```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}

```



```dockerfile
FROM golang:1.20 AS builder

WORKDIR /app

COPY . .

RUN go build -o hello-go

FROM alpine:latest

COPY --from=builder /app/hello-go .

CMD ["./hello-go"]

```



```
 => [internal] load build context                                                                              0.0s
 => => transferring context: 118B                                                                              0.0s
 => [builder 1/4] FROM docker.io/library/golang:1.20@sha256:8f9af7094d0cb27cc783c697ac5ba25efdc4da35f8526db21f7a  0.0s
 => CACHED [builder 2/4] WORKDIR /app                                                                          0.0s
 => [builder 3/4] COPY . .                                                                                     0.1s
 => [builder 4/4] RUN go build -o hello-go                                                                     3.7s
 => [stage-1 2/2] COPY --from=builder /app/hello-go .                                                          0.1s
 => exporting to image                                                                                         0.1s
 => => exporting layers                                                                                        0.1s
 => => writing image sha256:530e3be2f361e4f2b53537a9c09c8e8734722353c60a9b92df8cb5067a886f4a                   0.0s
 => => naming to docker.io/library/hello-go                                                                    0.0s
○ PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1-go>
```



```
● PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1-go> docker images
  REPOSITORY                               TAG       IMAGE ID       CREATED          SIZE
  hello-go                                 latest    530e3be2f361   33 seconds ago   9.65MB
```

Benefits of multi-stage builds:

They keep the Dockerfile organized and separate the build environment from the runtime environment.

Reducing image size:

They allow you to compile in a larger image and copy only the necessary files to a smaller image, removing unnecessary dependencies.

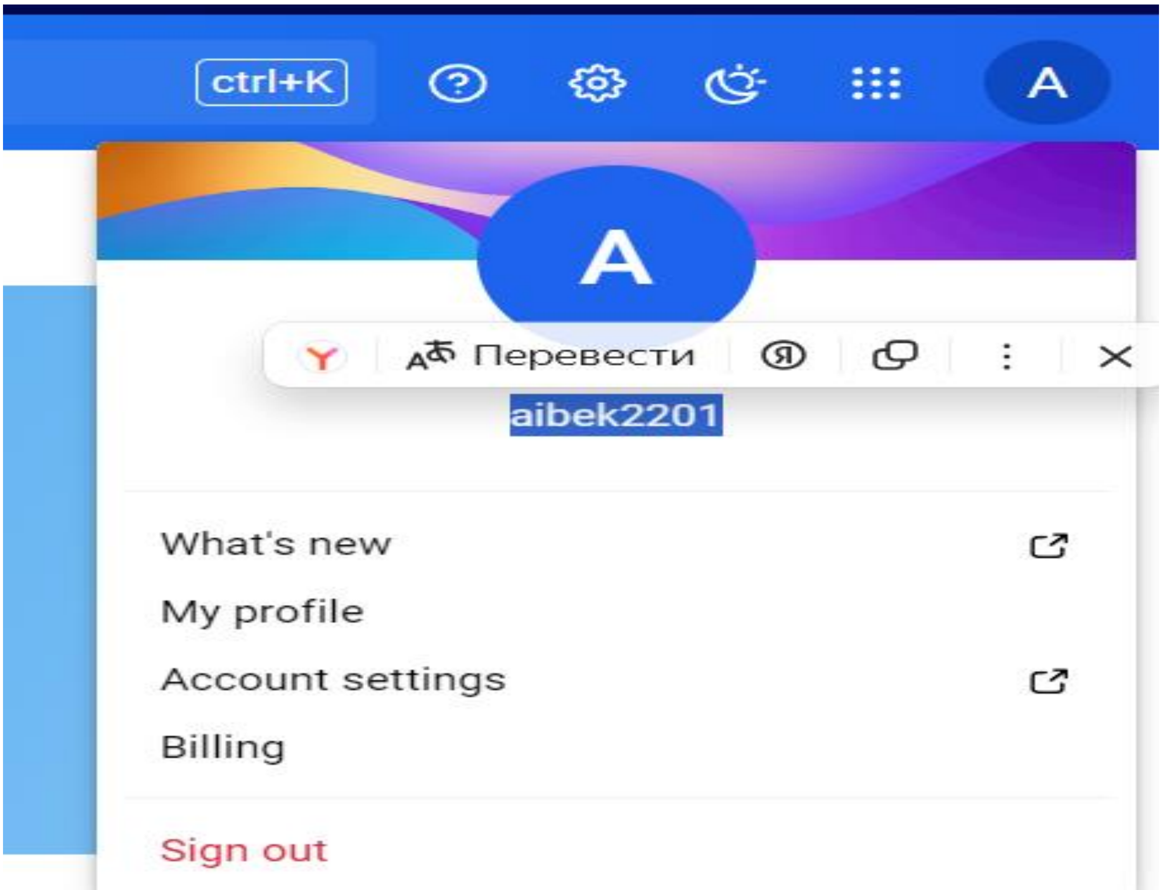
Useful scenarios:

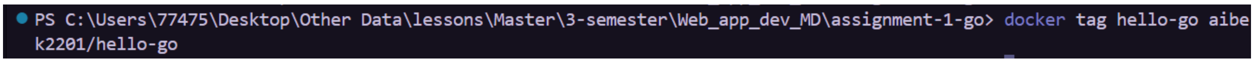
When building apps with heavy dependencies, compiling binaries, or creating lightweight images for deployment.

Exercise 4: Pushing Docker Images to Docker Hub

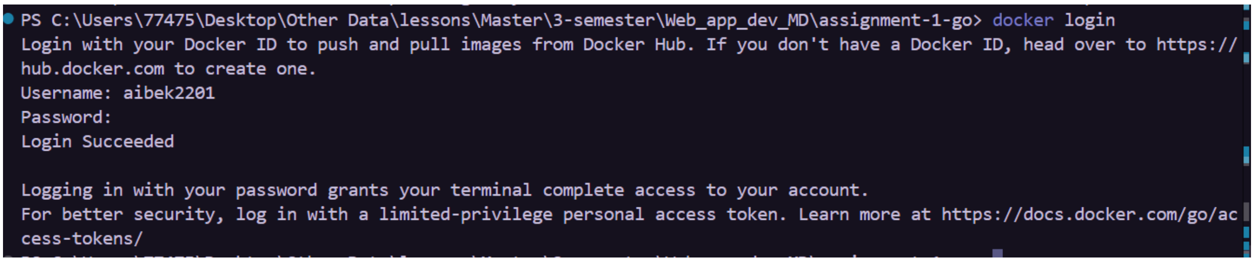
1. Objective: Learn how to share Docker images by pushing them to Docker Hub.
2. Steps:
   - Create an account on Docker Hub.
   - Tag the Docker image you built earlier with your Docker Hub username (e.g., docker tag hello-docker <your-username>/hello-docker).
   - Log in to Docker Hub using docker login.
   - Push the image to Docker Hub using docker push <your-username>/hello-docker.
   - Verify that the image is available on Docker Hub and share it with others.
3. Questions:
   - What is the purpose of Docker Hub in containerization?
   - How do you tag a Docker image for pushing to a remote repository?
   - What steps are involved in pushing an image to Docker Hub?

Result:

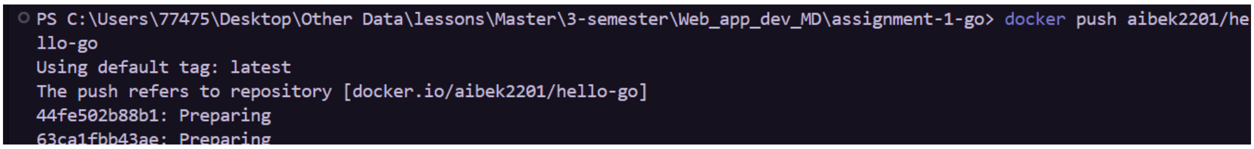




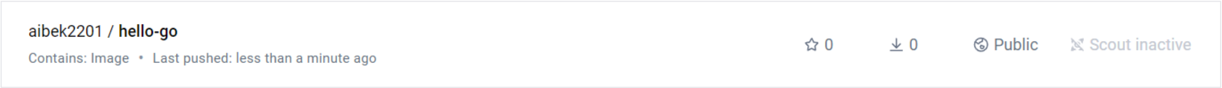
```
● PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1-go> docker tag hello-go aibe
  k2201/hello-go
```



```
PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1-go> docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://
hub.docker.com to create one.
Username: aibek2201
Password:
Login Succeeded

Logging in with your password grants your terminal complete access to your account.
For better security, log in with a limited-privilege personal access token. Learn more at https://docs.docker.com/go/ac
cess-tokens/
```



```
PS C:\Users\77475\Desktop\Other Data\lessons\Master\3-semester\Web_app_dev_MD\assignment-1-go> docker push aibek2201/he
llo-go
Using default tag: latest
The push refers to repository [docker.io/aibek2201/hello-go]
44fe502b88b1: Preparing
63ca1fbb43ae: Preparing
```



aibek2201 / hello-go

Contains: Image • Last pushed: less than a minute ago

Purpose of Docker Hub:

To store and share Docker images.

docker tag <local-image> <your-username>/<repository-name>:<tag>

Steps to push an image to Docker Hub:

Log in with docker login.

Tag the image.

Push the image using docker push <your-username>/<repository-name>:<tag>.