



KAZAKH-BRITISH
TECHNICAL
UNIVERSITY

Assignment #1
Web Application
Development

Prepared by:
Seitbekov S.
Checked by:
Serek A.

Almaty, 2024

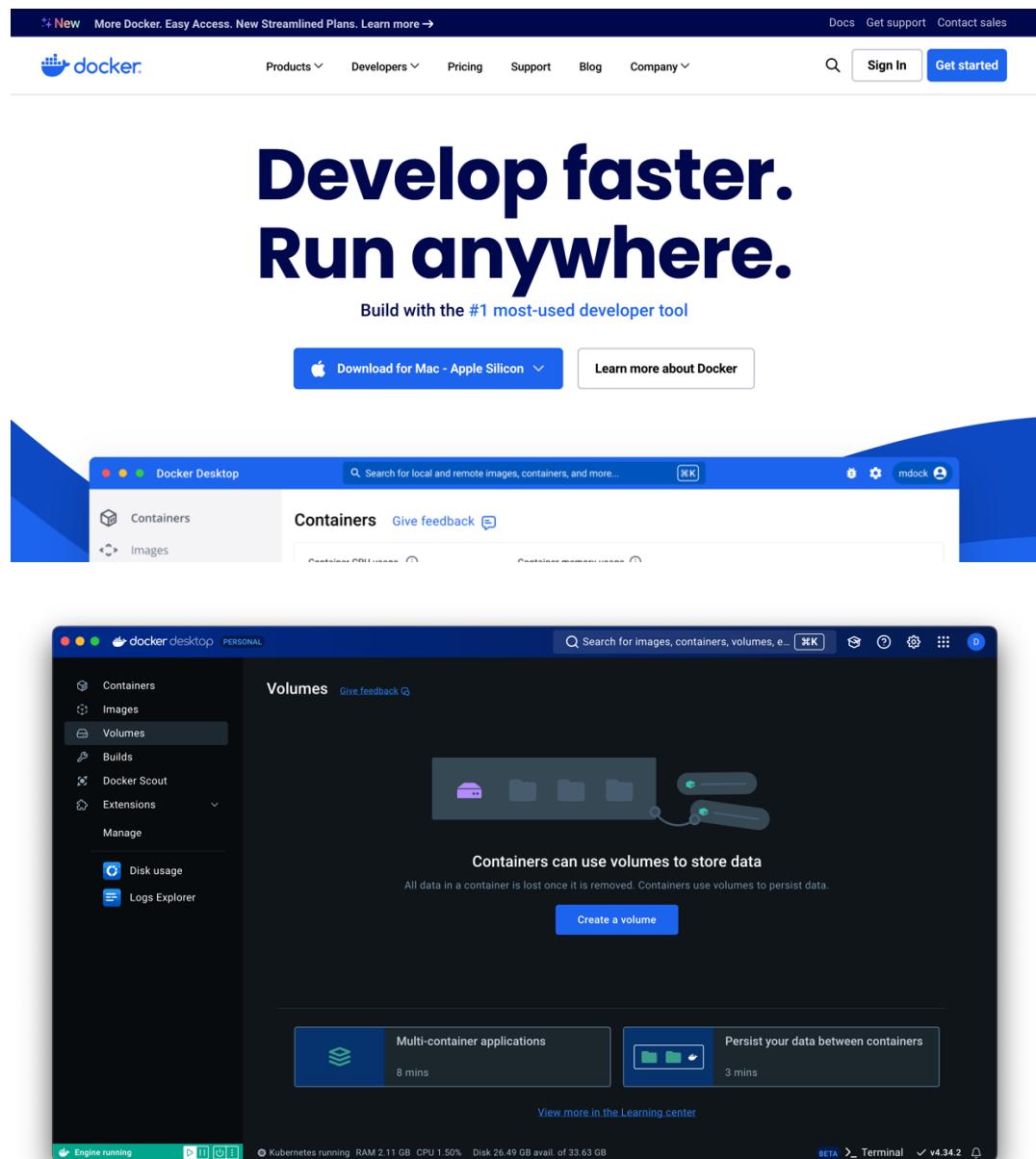
Intro to Containerization: Docker

Exercise 1: Installing Docker

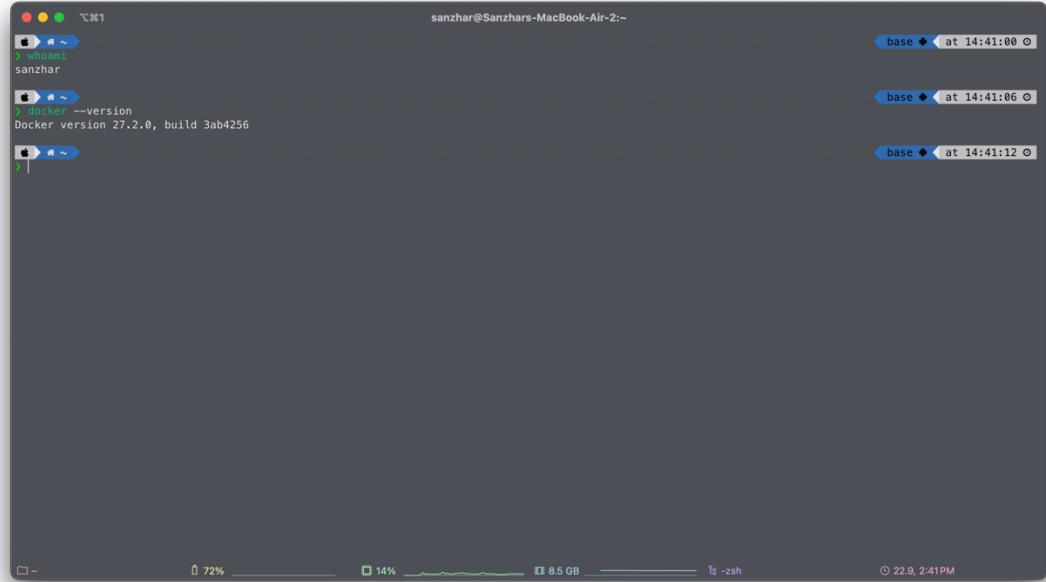
1. Objective: Install Docker on your local machine.

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.



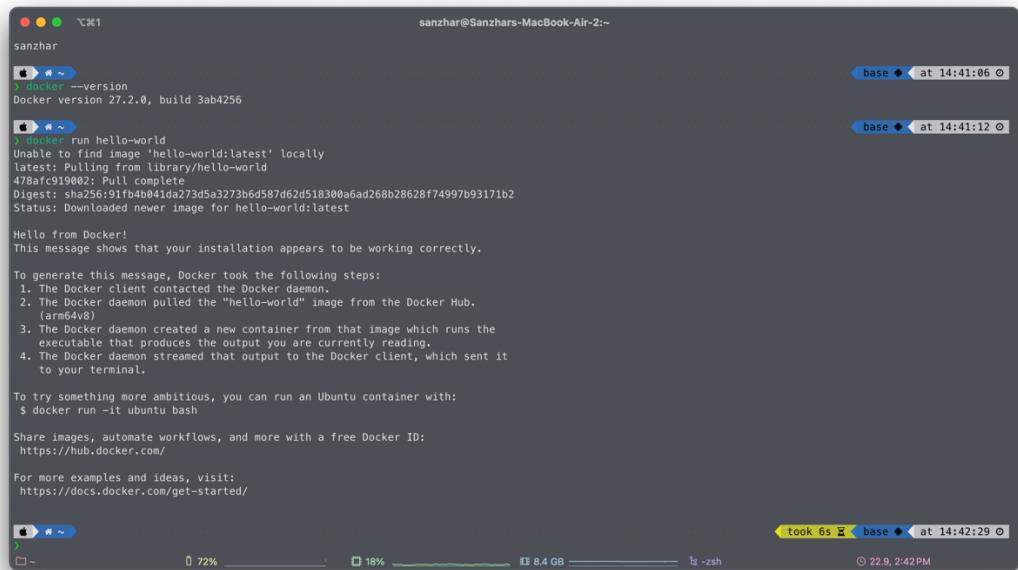
```

sanzhar@Sanzhars-MacBook-Air-2:~ base ◆ at 14:41:00 ○
$ docker --version
Docker version 27.2.0, build 3ab4256
base ◆ at 14:41:06 ○
base ◆ at 14:41:12 ○

```

The screenshot shows a macOS terminal window titled 'Terminal'. It displays the command 'docker --version' being run and its output. The terminal is part of a larger desktop environment with a dock at the bottom showing icons for Finder, Mail, Safari, and others. The status bar at the top right indicates the user is 'sanzhar' and the time is '14:41:00'. The status bar at the bottom right shows battery level (72%), signal strength (14%), disk usage (8.5 GB), and a terminal icon (-zsh). The system tray shows the date and time as '22.9, 2:41PM'.

- Run the command docker run hello-world to verify that Docker is set up correctly.



```

sanzhar@Sanzhars-MacBook-Air-2:~ base ◆ at 14:41:06 ○
$ docker --version
Docker version 27.2.0, build 3ab4256
base ◆ at 14:41:12 ○
base ◆ at 14:41:12 ○
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
478afc919002: Pull complete
Digest: sha256:91fb4d041da273d5a3273b6d587d62d518300a6ad268b28628f74997b93171b2
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.
   (arm64v8)
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.

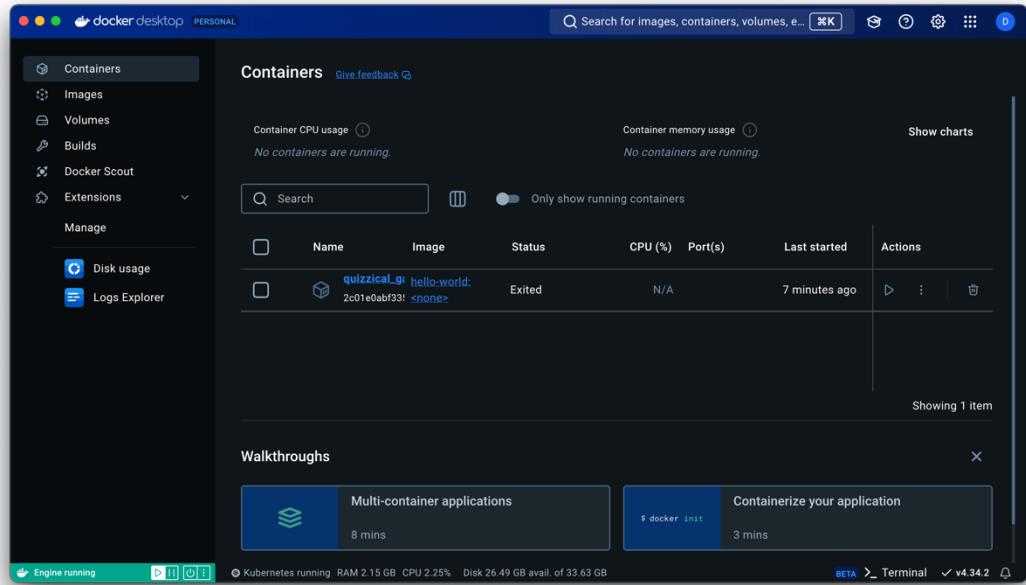
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
base ◆ at 14:42:29 ○

```

The screenshot shows a macOS terminal window titled 'Terminal'. It displays the command 'docker run hello-world' being run and its output. The terminal is part of a larger desktop environment with a dock at the bottom showing icons for Finder, Mail, Safari, and others. The status bar at the top right indicates the user is 'sanzhar' and the time is '14:41:06'. The status bar at the bottom right shows battery level (72%), signal strength (18%), disk usage (8.4 GB), and a terminal icon (-zsh). The system tray shows the date and time as '22.9, 2:42PM'.



3. Questions:

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

Answer: Key components of Docker are Docker Engine (core software that manages containers), Docker CLI (command line for interacting), Docker Daemon (background service managing containers and images), Docker Hub (official repository for images)

- How does Docker compare to traditional virtual machines?

Answer: Docker uses lightweight containers that share the host OS kernel, resulting in faster startup and lower resource usage, unlike VMs, which require a full OS installation

- What was the output of the docker run hello-world command, and what does it signify?

Answer: A message confirming Docker is installed correctly, signifying the container ran successfully and Docker's client-server setup is working properly. Also, it pulls the image if there is no such image locally yet.

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>.

```
sanzhar@Sanzhars-MacBook-Air-2:~
```

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(arm64v8)
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

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

```
docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
92c3b3500be6: Pull complete
ee57511b3c68: Pull complete
33791ce134bf: Pull complete
cc4f244ec205: Pull complete
3cad04a21c99: Pull complete
486c5264d3ad: Pull complete
b3fd15a02525: Pull complete
Digest: sha256:04ba374843cd2fc5c593885c0eacddebabd5ca375f9323666f28dfd5a9710e3
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest

What's next:
View a summary of image vulnerabilities and recommendations - docker scout quickview nginx
```

took 6s ✘ base ◆ at 14:42:29 ○

```
-zsh
```

took 8s ✘ base ◆ at 14:55:48 ○

22.9, 2:55 PM

- List all Docker images on your system using docker images.

```
sanzhar@Sanzhars-MacBook-Air-2:~
```

REPOSITORY	SIZE	TAG	IMAGE ID	CREATED
nginx	ago 193MB	latest	195245f0c792	5 weeks
docker/desktop-kubernetes	ago 419MB	kubernetes=v1.30.2-cni-v1.4.0-critools-v1.29.0-cri-dockerd-v0.3.11-1-debian	5ef0802e902d	2 month
registry.k8s.io/kube-apiserver	ago 112MB	v1.30.2	84c601f3f72c	3 month
registry.k8s.io/kube-scheduler	ago 60.5MB	v1.30.2	c7dd04b1bafe	3 month
registry.k8s.io/kube-controller-manager	ago 107MB	v1.30.2	e1dc3c400d3e	3 month
registry.k8s.io/kube-proxy	ago 87.9MB	v1.30.2	66dbb96a9149	3 month
docker/disk-usage-extension	ago 202MB	0.2.9	83bc5b217a10	7 month
docker/logs-explorer-extension	ago 12.5MB	0.2.6	d42880f71e98	7 month
registry.k8s.io/etcd	ago 139MB	3.5.12-0	014faa467e29	7 month
docker/desktop-kubernetes	ago 422MB	kubernetes=v1.29.1-cni-v1.4.0-critools-v1.29.0-cri-dockerd-v0.3.8-1-debian	e953cb03dd7e	8 month
registry.k8s.io/kube-apiserver	ago 123MB	v1.29.1	f3b81ff188c6	8 month
registry.k8s.io/kube-scheduler	ago 58MB	v1.29.1	140ecfd0789f	8 month
registry.k8s.io/kube-controller-manager	ago 118MB	v1.29.1	8715bb0e3bc2	8 month
registry.k8s.io/kube-proxy	ago 85.4MB	v1.29.1	b125ba1d1878	8 month
registry.k8s.io/etcd	ago 3.5.10-0	3.5.10-0	79f8d13ae8b8	10 mont
hs ago 130MB	0.2.5	0.2.5	0dd0ba6d3eb9	10 mont
docker/logs-explorer-extension	ago 12.4MB	<none>	6c7be49d2a11	11 mont
nginx	ago 69%			

took 8s ✘ base ◆ at 14:55:48 ○

```
-zsh
```

22.9, 2:56 PM

- Run a container from the pulled image using docker run -d <image-name>.

```

sanzhars@Sanzhars-MacBook-Air-2:~ docker ps
CONTAINER ID        IMAGE               COMMAND      CREATED             STATUS              PORTS
5b605db4db27        nginx              "/docker-entrypoint..."   27 seconds ago   Up 27 seconds      80/tcp
sanzhars@Sanzhars-MacBook-Air-2:~ docker run -d nginx
5b605db4db2794bab8b197d6afde4e304ab553e181e9678928a5bebabd82b49

```

- List all running containers using docker ps and stop a container using docker stop <container-id>.

```

sanzhars@Sanzhars-MacBook-Air-2:~ docker ps
CONTAINER ID        IMAGE               COMMAND      CREATED             STATUS              PORTS
5b605db4db27        nginx              "/docker-entrypoint..."   27 seconds ago   Up 27 seconds      80/tcp
sanzhars@Sanzhars-MacBook-Air-2:~ docker stop 5b605db4db27
5b605db4db27
sanzhars@Sanzhars-MacBook-Air-2:~ docker ps
CONTAINER ID        IMAGE               COMMAND      CREATED             STATUS              PORTS

```

3. Questions:

- What is the difference between docker pull and docker run?

Answer: docker pull downloads a Docker image from a registry to local system. **docker run** creates and starts a new container from a Docker image, also pulling the image if it's not already available locally.

- How do you find the details of a running container, such as its ID and status?

Answer: We can use “**docker ps**” to see the container’s ID, status, and other details.

- What happens to a container after it is stopped? Can it be restarted?

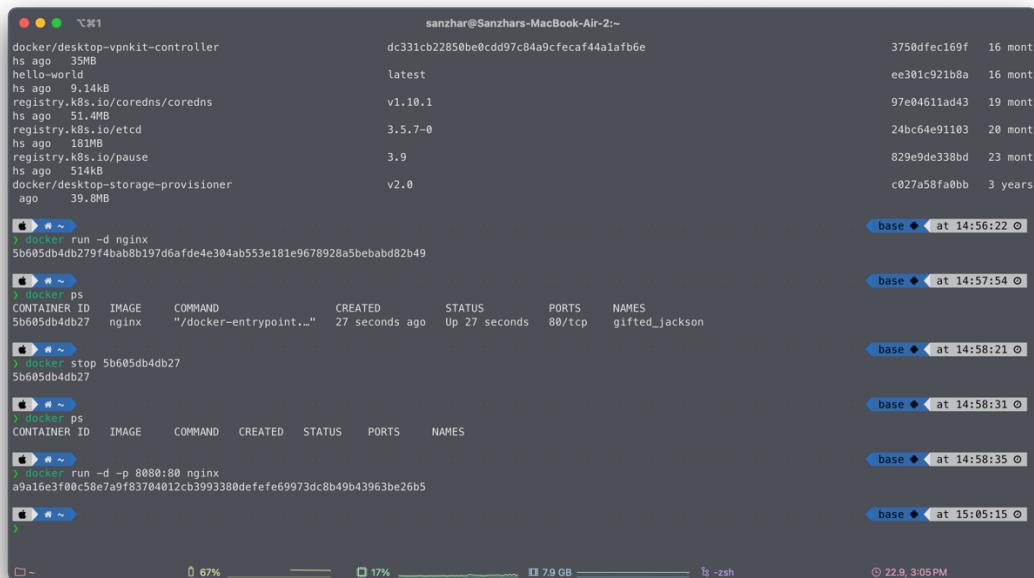
Answer: The container still exists but is in a stopped state, it can be restarted using command **docker start container_id**

Exercise 3: Working with Docker Containers

1. Objective: Learn how to manage Docker containers.

2. Steps:

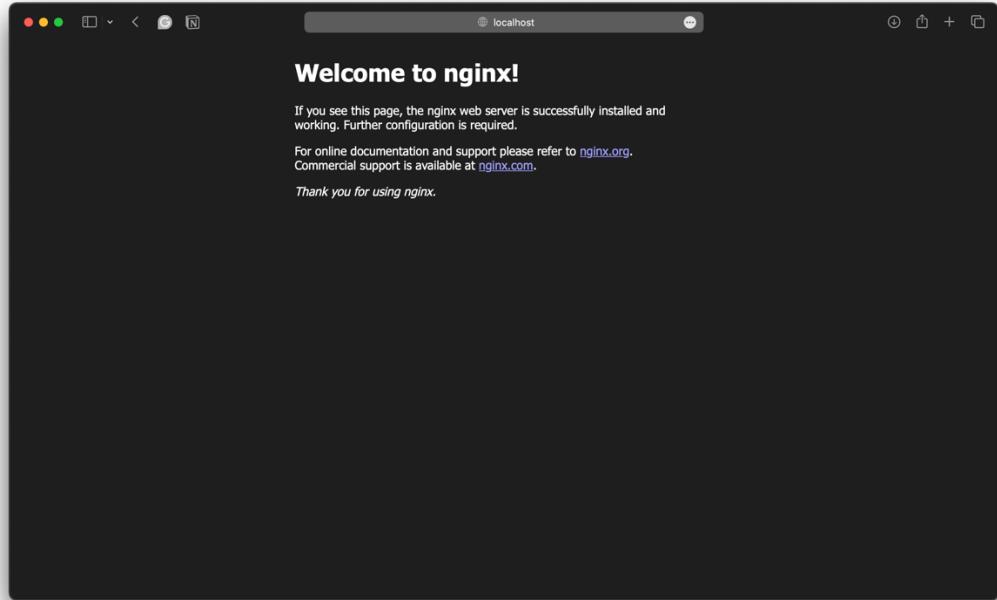
- Start a new container from the nginx image and map port 8080 on your host to port 80 in the container using **docker run -d -p 8080:80 nginx**.



The screenshot shows a terminal window on a Mac OS X desktop. The user is navigating through Docker commands to manage a container named 'gifted_jackson'. The session starts with listing local images, then creating a new container from the 'nginx' image. The user then stops the container and restarts it with port mapping. Finally, they check the logs of the running container.

```
sanzhar@sanzhars-MacBook-Air-2:~$ docker images
sanzhar@Sanzhars-MacBook-Air-2:~$ docker run -d nginx
5b605db4db279f4bab8b197d6afde4e304ab553e181e9678928a5bebabd82b49
sanzhar@sanzhars-MacBook-Air-2:~$ docker ps
CONTAINER ID        IMAGE       COMMAND   CREATED          STATUS    PORTS     NAMES
5b605db4db27        nginx      "/docker-entrypoint..."   27 seconds ago   Up 27 seconds   80/tcp    gifted_jackson
sanzhar@sanzhars-MacBook-Air-2:~$ docker stop 5b605db4db27
5b605db4db27
sanzhar@sanzhars-MacBook-Air-2:~$ docker ps
CONTAINER ID        IMAGE       COMMAND   CREATED          STATUS    PORTS     NAMES
sanzhar@sanzhars-MacBook-Air-2:~$ docker run -d -p 8080:80 nginx
a916cef00c58e7a9f83704012cb3993380defefef69973dc8b49b43963be26b5
sanzhar@sanzhars-MacBook-Air-2:~$ docker logs a916cef00c58e7a9f83704012cb3993380defefef69973dc8b49b43963be26b5
[...]
```

- 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`.

```
docker exec -it a9a16e3f00c5 /bin/bash
root@a9a16e3f00c5:/# ls
bin  boot  dev  docker-entrypoint.d  docker-entrypoint.sh  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@a9a16e3f00c5:/#
```

- Stop and remove the container using docker stop <container-id> and docker rm <container-id>.

```

sanzhar@sanzhars-MacBook-Air-2:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 14:58:31
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 14:58:35
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 15:05:15
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 15:06:28

sanzhar@sanzhars-MacBook-Air-2:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 15:07:08
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 15:07:20
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 15:07:26
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 15:07:28

sanzhar@sanzhars-MacBook-Air-2:~$ docker stop a9a16e3f00c5
a9a16e3f00c5
sanzhar@sanzhars-MacBook-Air-2:~$ docker rm a9a16e3f00c5
a9a16e3f00c5
sanzhar@sanzhars-MacBook-Air-2:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
a9a16e3f00c58e7a9f83704012cb3993380defefe69973dc8b49b43963be26b5 base at 15:07:28

```

The screenshot shows a macOS terminal window with several Docker commands run sequentially. The first command, `docker ps`, lists a single container named 'base' with a unique ID. The second command, `docker stop a9a16e3f00c5`, stops the container. The third command, `docker rm a9a16e3f00c5`, removes the stopped container. Finally, another `docker ps` command is run, which shows no active containers, indicating the removal was successful. The terminal also displays system status bars at the bottom, including battery level (67%), signal strength (13%), RAM usage (7.6 GB), and a terminal tab indicator.

3. Questions:

- How does port mapping work in Docker, and why is it important?

Answer: port mapping connects a port on the host machine to a port inside the Docker container (-p 8080:80 maps host port 8080 to container port 80). It's important for enabling external access to services running inside containers.

- What is the purpose of the docker exec command?

Answer: docker exec executes a command inside a running container, allowing to interact with the container without stopping it (accessing a shell with docker exec -it container_id /bin/bash).

- How do you ensure that a stopped container does not consume system resources?

Answer: we can use docker rm container_id to remove the stopped container, ensuring it no longer occupies any system resources like disk space, CPU, RAM.

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.

A terminal session on a Mac OS X system (sanzhar@Sanzhars-MacBook-Air-2) demonstrating the creation and removal of a Docker container named 'hello-docker'. The session starts with listing containers, then creating a new one with the command 'docker run -d --name hello-docker alpine /bin/sh'. It then enters the container via 'docker exec -it' and runs 'ls' to show the default Alpine Linux file structure. After exiting the container, it is stopped with 'docker stop' and removed with 'docker rm'. Finally, the user navigates to the directory and runs 'python app.py' to verify the application is working.

```
sanzhar@Sanzhars-MacBook-Air-2:~/hello-docker
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
a9a16e3f00c5 alpine "/bin/sh" About a minute ago Up About a minute 0.0.0.0:8080->80/tcp lucid_leavitt

$ docker exec -it a9a16e3f00c5 /bin/bash
root@a9a16e3f00c5:/# ls
bin boot dev docker-entrypoint.d docker-entrypoint.sh etc home lib media mnt opt proc root run sbin srv sys tmp usr var
root@a9a16e3f00c5:/# exit
exit

What's next:
Try Docker Debug for seamless, persistent debugging tools in any container or image - docker debug a9a16e3f00c5
Learn more at https://docs.docker.com/go/debug-cli/
$ docker stop a9a16e3f00c5
a9a16e3f00c5
$ docker rm a9a16e3f00c5
a9a16e3f00c5
$ docker ps
CONTAINER ID IMAGE CREATED STATUS PORTS NAMES
$ cd hello-docker
$ cd hello-docker
$ ./hello-docker
$ python app.py
Hello, Docker!
```

- Create a simple Python script (e.g., app.py) that prints "Hello, Docker!" to the console.

A terminal session on a Mac OS X system (sanzhar@Sanzhars-MacBook-Air-2) showing a Vim editor window titled 'vim app.py'. The code in the file is a single line: 'print("Hello, Docker!")'. The terminal also shows the output of the Python script when run, which is 'Hello, Docker!'. The status bar at the bottom indicates the file is in 'NORMAL' mode, has 1 line and 21 columns, and is using utf-8 encoding.

```
NORMAL > app.py[+]
1 print("Hello, Docker!")

$ python app.py
Hello, Docker!
```

- 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.

vim Dockerfile

```
1 FROM python:3.11
2 WORKDIR /app
3
4 COPY . /app
5
6 ENTRYPOINT ["python", "app.py"]
7
```

NORMAL > Dockerfile
"Dockerfile" 8L, 78B

□ -hello-docker 62% □ 65% 7.2 GB □ Vim +zsh 22.9, 4:24 PM

- Build the Docker image using `docker build -t hello-docker ..`

```
sanzhar@Sanzhars-MacBook-Air-2:~/hello-docker
```

```
❯ docker build -t hello-docker .
```

```
[+] Building 25.6s (9/9) FINISHED
  => [internal] load build definition from Dockerfile
  => [internal] load metadata for docker.io/library/python:3.11
  => [internal] library /python:3 pull token for registry-1.docker.io
  => [internal] load .dockerignore
  => [internal] transfering context: 2B
  => [1/1] FROM docker.io/library/python:3.11sha256:1573371e080919fe47d2f71c889eae5234f59114c23b08b340d0d02c74d39
    => resolve docker.io/library/python:3.11sha256:1573371e080919fe47d2f71c889eae5234f59114c23b08b340d0d02c74d39
  => sha256:843d8321825bc8302752ae0030613bd15+6eeef2fe32f3c1520-5b0c7 64,00B / 64,00MB
  => sha256:2347246992329c204abb7f6f7d33e2822370d76501e20a46562ab3217 5,99kB / 5,99kB
  => sha256:843d8321825bc8302752ae0030613bd15+6eeef2fe32f3c1520-5b0c7 64,00B / 64,00MB
  => sha256:c9925319e4472d1f71c889eae5234f59114c23b08b340d0d02c74d39 9,00kB / 9,00kB
  => sha256:f27911a480115973d713022beef97319e84f13b38499b0566e081c73747f2358efb18739 2,33kB / 2,33kB
  => sha256:364d1975f694748a093371c8691f455531e5be8a562801c1ebf235922119e 23,59kB / 23,59kB
  => sha256:a348c2a80463134ce70dac4744651808081a8a80b9347ff5c8436043bd5 202,65MB / 202,65MB
  => extracting sha256:56c9925319e835168319b236d0712402bde6137462057159 6,24MB / 6,24MB
  => sha256:295616ca8edee4279fd16caaa8d00719d978970397820857c3475s95708ce 22,63MB / 22,63MB
  => sha256:e01a88086061ad5621e606039116598b7150874781090797c3531672318 5,55MB / 5,55MB
  => extracting sha256:364d1975f694748a093371c8691f455531e5be8a562801c1ebf235922119e
  => extracting sha256:325153de1615112363db771d420cd6b137a556156e743371c1520-5b0c7 2,77kB / 2,77kB
  => sha256:325153de1615112363db771d420cd6b137a556156e743371c1520-5b0c7 0,35MB / 0,35MB
  => extracting sha256:353d1615112363db771d420cd6b137a556156e743371c1520-5b0c7 0,35MB / 0,35MB
  => extracting sha256:203c16ca89edec4270d16caaa85db0a9d9819639f822857c73475s95708ce 0,05MB / 0,05MB
  => extracting sha256:e18648ed6f45d6e249e9eb3df1b59bbf7801750f4781009107977cb3c316f 0,05MB / 0,05MB
  => [internal] load build context
  => transforming context: 593,25B
  => [2/3] WORKDIR /app
  => [3/3] COPY . /app
  => exporting to image
  => exporting layers
  => writing image sha256:b960d4f73e3323fb5ac8660b2aef807dad48167177d338189145396643a6438
  => naming to docker.io/library/hello-docker
```

- Run the container using docker run hello-docker.

The screenshot shows a terminal window on a Mac OS X desktop. The title bar says "sanzhar@Sanzhars-MacBook-Air-2:~/hello-docker". The main pane contains the following text:

```
sanzhar@Sanzhars-MacBook-Air-2:~/hello-docker
$ docker run hello-docker
Hello, Docker!
$
```

At the top right of the terminal window, there is a status bar with the text "took 19s" and "hello-docker at 16:24:37". Below the terminal window, the desktop background is visible, showing a dark grey gradient. At the bottom of the screen, there is a dock with several icons, including Finder, Mail, Safari, and others. The battery level is shown as 62%.

3. Questions:

- What is the purpose of the FROM instruction in a Dockerfile?

Answer: The FROM instruction specifies the base image to use for the container. It is the starting point of the build process, providing the necessary environment, dependencies, etc.

- How does the COPY instruction work in Dockerfile?

Answer: The COPY instruction copies files or directories from the host filesystem into the container's filesystem during the build process.

- What is the difference between CMD and ENTRYPOINT in Dockerfile?

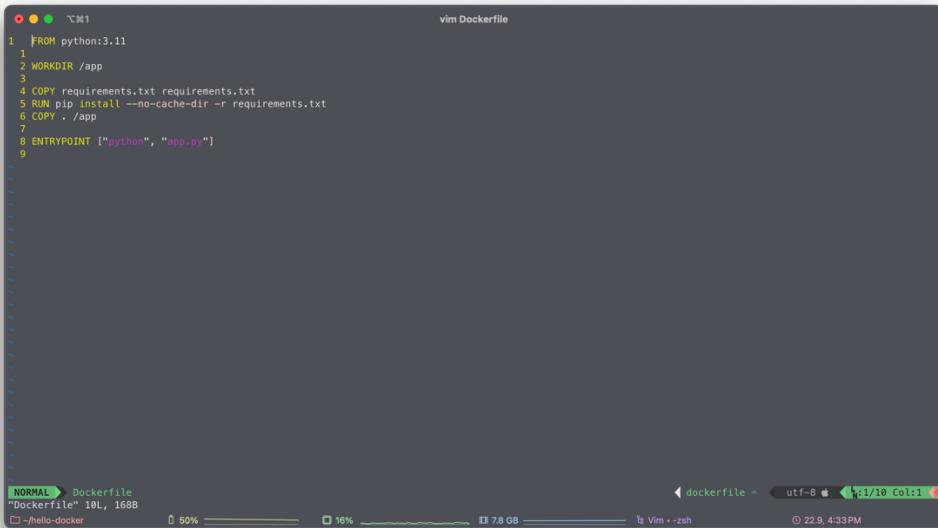
Answer: CMD specifies the default command (with parameters) to run in the container, but it can be overridden by user input at runtime. ENTRYPOINT defines the command (process) that will always run when the container starts and is not easily overridden.

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.**



```
vim Dockerfile
FROM python:3.11
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
COPY . /app
ENTRYPOINT ["python", "app.py"]
```

The screenshot shows a terminal window with Vim editing a Dockerfile. The Dockerfile contains the following code:

```
FROM python:3.11
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
COPY . /app
ENTRYPOINT ["python", "app.py"]
```

The status bar at the bottom indicates the file is 10L long and 168B in size. The Vim interface shows standard navigation keys like h, j, k, l, and movement keys like gg, G, and f.

- **Use a .dockerignore file to exclude unnecessary files from the image.**



```
vim .dockerignore
*.pyc
.venv/
.vscode/
*.log
.dockerignore
assets/
```

The screenshot shows a terminal window with Vim editing a .dockerignore file. The file contains the following content:

```
*.pyc
.venv/
.vscode/
*.log
.dockerignore
assets/
```

The status bar at the bottom indicates the file is 7L long and 63B in size. The Vim interface shows standard navigation keys like h, j, k, l, and movement keys like gg, G, and f.

- **Rebuild the Docker image and observe the build process to understand how caching works.**

```

sanzhar@Sanzhars-MacBook-Air-2:~/hello-docker
❯ docker build -t hello-docker-optimized .

[+] Building 6.9s (11/11) FINISHED
--> Internal load build definition from Dockerfile
--> transferring dockerfile: 207B
--> Internal load metadata for docker.io/library/python:3.11
--> [auth] library/python@sha256:157a371e60389919fe4a72dff71ce86eaa5234f59114c23b0b346d0d82c74d39
--> Internal load .dockerignore
--> Internal load build context: 103B
--> Internal load build context: 358B
--> transferring context: 358B
--> CACHED [2/5] WORKDIR /app
--> [3/5] COPY requirements.txt requirements.txt
--> [4/5] RUN pip install --no-cache-dir -r requirements.txt
--> [5/5] COPY ./app .
--> exporting layers
--> writing image sha256:adc7179eed67c933f6829811bcb882fdfa9dc9f6373cc143dd92db53c4a3b89eb
--> naming to docker.io/library/hello-docker-optimized

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/pfwild7b1iwhvey2br9baq2ln

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview

```

took 8s ✘ hello-docker ◆ at 16:28:56 ○

```

sanzhar@Sanzhars-MacBook-Air-2:~/hello-docker
❯ docker images
REPOSITORY          TAG      IMAGE ID   CREATED        SIZE
hello-docker-optimized    latest   adc7179eed67  6 seconds ago  1.03GB
hello-docker          latest   9b60a4f73e33  5 minutes ago  1.61GB
registry.k8s.io/kube-apiserver  v1.28.2  30bb499447fe  12 months ago  120MB
registry.k8s.io/kube-scheduler  v1.28.2  64fc40cee371  12 months ago  57.8MB
registry.k8s.io/kube-proxy     v1.28.2  7da62c127fc0  12 months ago  68.3MB
registry.k8s.io/kube-controller-manager  v1.28.2  89d57b83c178  12 months ago  116MB
registry.k8s.io/etcd         3.5.9-0   9cdd6470f48c  16 months ago  181MB

```

-zsh ○ 22.9, 4:29PM

- **Compare the size of the optimized image with the original.**

```

sanzhar@Sanzhars-MacBook-Air-2:~/hello-docker
❯ docker build -t hello-docker .
REPOSITORY          TAG      IMAGE ID   CREATED        SIZE
hello-docker-optimized    latest   adc7179eed67  50 seconds ago  1.03GB
hello-docker          latest   9b60a4f73e33  6 minutes ago  1.61GB
registry.k8s.io/kube-apiserver  v1.28.2  30bb499447fe  12 months ago  120MB
registry.k8s.io/kube-proxy     v1.28.2  7da62c127fc0  12 months ago  68.3MB
registry.k8s.io/kube-controller-manager  v1.28.2  89d57b83c178  12 months ago  116MB
registry.k8s.io/kube-scheduler  v1.28.2  64fc40cee371  12 months ago  57.8MB
registry.k8s.io/etcd         3.5.9-0   9cdd6470f48c  16 months ago  181MB
dockerdocker/vpnkit-controller  dc331ch22850be0cd97c84a9cfecaf44a1afb6e  3750afe169f  16 months ago  35MB
registry.k8s.io/coredns/coredns  v1.10.1   97e04611ad43  19 months ago  51.4MB
registry.k8s.io/pause          3.9       829e9de338bd  23 months ago  514kB
dockerdocker/desktop-storage-provisioner  v2.0     c027a58fa0bb  3 years ago  39.8MB

```

-zsh ○ 22.9, 4:30PM

3. Questions:

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

Answer: Each command in a Dockerfile (FROM, RUN, COPY) creates a layer in the Docker image. These layers are stacked on top of each other to form the final image. Changes in one layer only affect that layer and the layers built after it. Each layer increases the overall size of the image, so more layers or large layers can result in a bigger image. Docker caches layers to speed up builds. If a layer has already been built and hasn't changed, Docker reuses the cached layer, avoiding the need to rebuild it from scratch

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

Answer: Docker caches each layer during the build process. During rebuild an image, Docker checks if the layer has changed since the last build. If the layer hasn't changed, Docker reuses the cached version of that layer instead of rebuilding it. This can drastically reduce build times, especially for complex images with many steps

- What is the role of the .dockerignore file?

Answer: The .dockerignore file specifies which files and directories should be excluded from the build context, preventing them from being copied into the Docker image. It improves build speed and reduces image size.

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).

```

vim main.go
NORMAL > main.go[+]
□ -/go-multi-stage 0 48% ━━━━━━━━ □ 16% ━━━━━━ □ 9.3 GB ━━━━━━ ⑤ Vim + ~zsh
□ 22.9, 4:46 PM

```

```

5 package main
4
3 import "fmt"
2
1 func main() {
6     fmt.Println("Hello, World!")
1 }
2

```

- 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.

```

vim (Vim)
NORMAL > Dockerfile
Dockerfile
"Dockerfile" 16L, 172B written
□ -/go-multi-stage 0 46% ━━━━━━━━ □ 17% ━━━━━━ □ 9.4 GB ━━━━━━ ⑤ Vim + ~zsh
□ 22.9, 4:55 PM

```

```

12 FROM golang:1.22-alpine AS builder
11
10 WORKDIR /app
9
8 COPY . .
7
6 RUN go build main.go
5
4 FROM alpine:latest
3
2 WORKDIR /app
1
13 COPY --from=builder /app/main .
1
2 ENTRYPOINT ["./main"]
3

```

- Build and run the Docker image, and compare the size of the final image with a single-stage build.

```

[+] Building 4.2s (13/13) FINISHED
--> [internal] load build definition from Dockerfile
--> [internal] load metadata for docker.io/library/alpine:latest
--> [internal] load metadata for docker.io/library/golang:1.22-alpine
--> [internal] load .dockerignore
--> transferring context: 2B
--> [builder 1/4] FROM docker.io/library/golang:1.22-alpine@sha256:48eab5e3505d8c8b42a06fe5f1cf4c346c167cc6a89e772f31cb9e5c301dcf60
--> [stage-1 1/3] FROM docker.io/library/alpine@sha256:beefdb8a1da6d2915560fde36db0b524eb737fc57cd1367effd16dc0d060
--> [internal] load build context
--> transferring context: 237B
--> CACHED [stage-1 2/3] WORKDIR /app
--> CACHED [builder 2/4] WORKDIR /app
--> [builder 3/4] COPY .
--> [builder 4/4] RUN go build main.go
--> [stage-1 3/3] COPY --from=builder /app/main .
--> exporting to image
--> exporting layers
--> writing image sha256:702fdaecbb8c06ffe3401l3614853ed74c30c2d375d211c63076560a07e83d21
--> naming to docker.io/library/go-multi-stage

View build details: docker-desktop://dashboard/build/desktop-linux/rx22cgq8o0ffiwyc1fgnuoit

What's next:
  View a summary of image vulnerabilities and recommendations - docker scout quickview

[+] Building 5s (2/2) FINISHED
--> [internal] load build definition from Dockerfile
--> [internal] load metadata for docker.io/library/golang:1.22-alpine
--> [internal] load .dockerignore
--> transferring context: 2B
--> CACHED [2/2] WORKDIR /app
--> [2/2] COPY .
--> [2/2] RUN go build main.go
--> exporting to image
--> exporting layers
--> writing image sha256:702fdaecbb8c06ffe3401l3614853ed74c30c2d375d211c63076560a07e83d21
--> naming to docker.io/library/go-multi-stage

Hello, World!

```

The screenshot shows a macOS terminal window with two tabs. Tab 1 is titled ".o-multi-stage (-zsh)" and shows the output of a multi-stage Docker build. Tab 2 is titled "~ (-zsh)" and shows the output of running the Docker image. The terminal shows the build process, including copying files, building Go code, and creating a multi-layered Docker image. The final command "Hello, World!" is executed and printed to the screen.

```

[+] Building 4.8s (10/10) FINISHED
--> [internal] load build definition from Dockerfile
--> [internal] load metadata for docker.io/library/golang:1.22-alpine
--> [auth] library/golang:pull token for registry-1.docker.io
--> [internal] load .dockerignore
--> transferring context: 2B
--> [1/4] FROM docker.io/library/golang:1.22-alpine@sha256:48eab5e3505d8c8b42a06fe5f1cf4c346c167cc6a89e772f31cb9e5c301dcf60
--> [internal] load build context
--> transferring context: 157B
--> CACHED [2/4] WORKDIR /app
--> [3/4] COPY .
--> [4/4] RUN go build main.go
--> exporting to image
--> exporting layers
--> writing image sha256:5f8d2ef3e6e77e791ef1f5d1472d9966826f5bfe0124f8631f3e2fc456057d8d
--> naming to docker.io/library/go-single-stage

View build details: docker-desktop://dashboard/build/desktop-linux/o3e165c9r14maasl134m6qpro

What's next:
  View a summary of image vulnerabilities and recommendations - docker scout quickview

[+] Building 6s (2/2) FINISHED
--> [internal] load build definition from Dockerfile
--> [internal] load metadata for docker.io/library/golang:1.22-alpine
--> [internal] load .dockerignore
--> transferring context: 2B
--> CACHED [2/2] WORKDIR /app
--> [2/2] COPY .
--> [2/2] RUN go build main.go
--> exporting to image
--> exporting layers
--> writing image sha256:702fdaecbb8c06ffe3401l3614853ed74c30c2d375d211c63076560a07e83d21
--> naming to docker.io/library/go-single-stage

Hello, World!

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
go-single-stage	latest	5f8d2ef3e6e7	9 seconds ago	259MB
go-multi-stage	latest	702fdaecbb8e	2 minutes ago	10.8MB
hello-docker-optimized	latest	adc7179eed67	29 minutes ago	1.03GB
hello-docker	latest	9b60d4f73e33	35 minutes ago	1.61GB

The screenshot shows a macOS terminal window with two tabs. Tab 1 is titled ".o-multi-stage (-zsh)" and shows the output of a single-stage Docker build. Tab 2 is titled "~ (-zsh)" and shows the output of running the Docker image. The terminal shows the build process, including copying files and building Go code. The final command "Hello, World!" is executed and printed to the screen. Below the terminal, a table lists the images available in the repository, showing their names, tags, IDs, creation times, and sizes.

3. Questions:

- **What are the benefits of using multi-stage builds in Docker?**

Answer: multi-stage builds allow us to discard unnecessary files and dependencies used during the build process, create a clean and more secure image with easier maintenance.

- **How can multi-stage builds help reduce the size of Docker images?**

Answer: In a multi-stage build, we can use one image to build the application and then switch to a smaller, minimal base image to run the final application. This ensures the final image contains only the essential files and the executable.

- **What are some scenarios where multi-stage builds are particularly useful?**

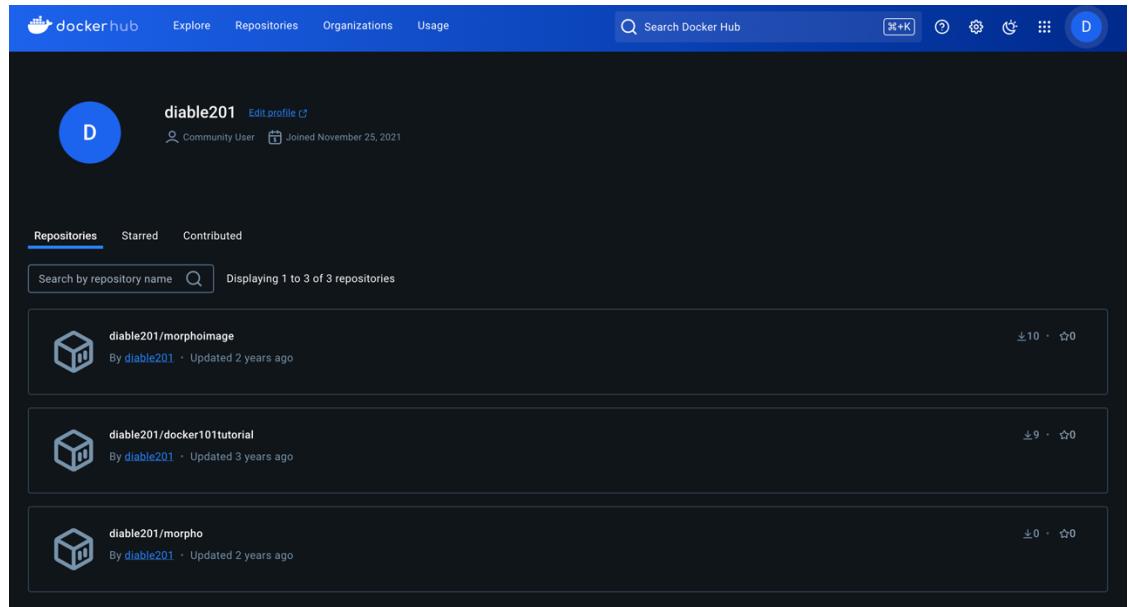
Answer: Multi-stage builds are highly useful for languages like Go, Java, or C++ where we can compile code in the first stage and copy only the binary to the final runtime environment. Also multi-stage builds are useful for CI/CD pipelines and production containers.

Exercise 4: Pushing Docker Images to Docker Hub

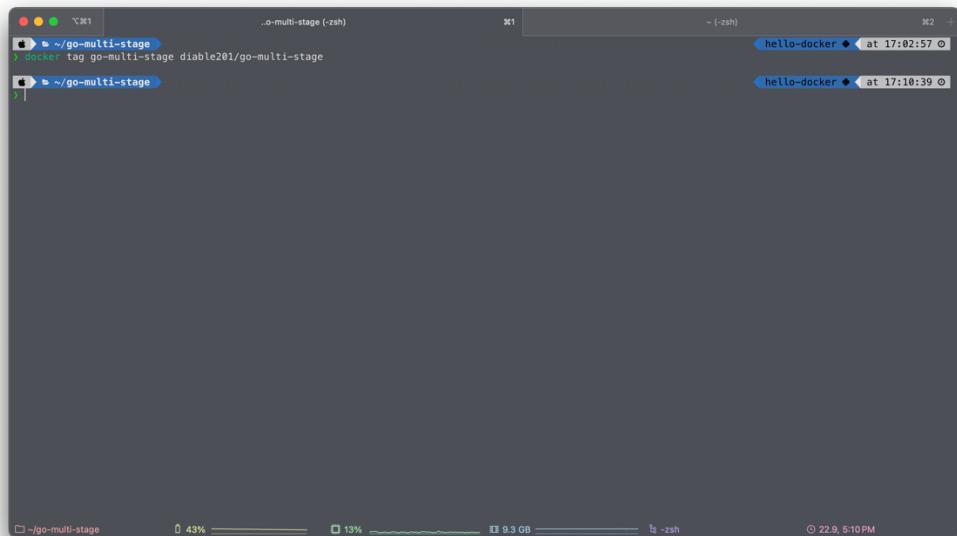
1. Objective: Learn how to share Docker images by pushing them to Docker Hub.

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).



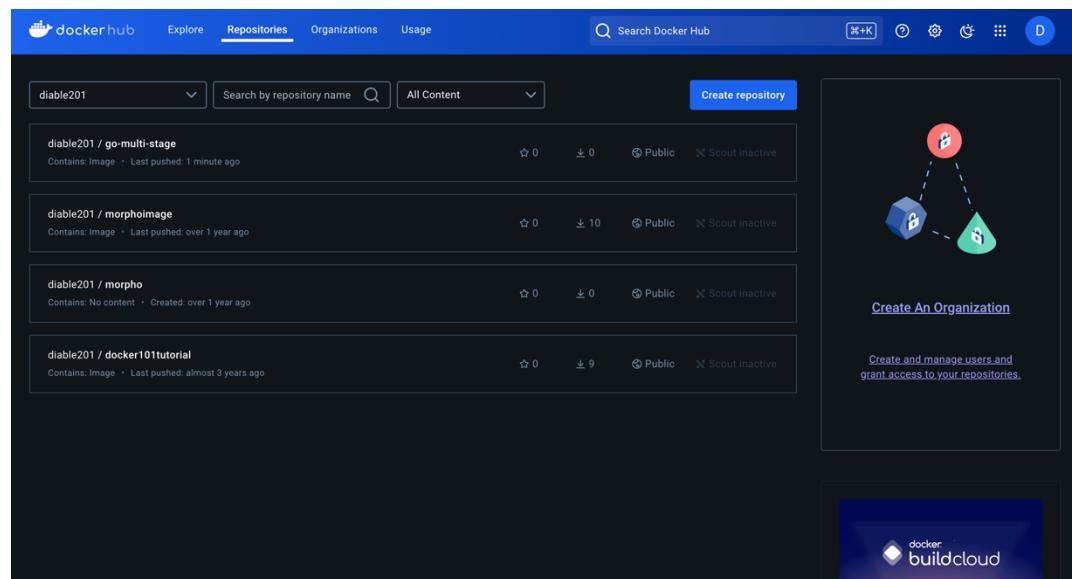
- Log in to Docker Hub using docker login.

```
...o-multi-stage (-zsh)          ...o-multi-stage (-zsh)
❯ docker login -u diable201      took 13s ✘ hello-docker ◆ at 17:12:04 ○
Password:                         took 23s ✘ hello-docker ◆ at 17:12:32 ○
Login Succeeded
```

- **Push the image to Docker Hub using `docker push <your-username>/hello-docker`.**

```
...o-multi-stage (-zsh)          ...o-multi-stage (-zsh)
❯ docker push diable201/go-multi-stage      took 13s ✘ hello-docker ◆ at 17:12:04 ○
Using default tag: latest
The push refers to repository [docker.io/diable201/go-multi-stage]
e06acf18fb029: Pushed
7f4fc8f5d33aa: Pushed
16113d51b718: Mounted from library/golang
latest: digest: sha256:b0beb1f01d4f90cd93f9b88dc2a37ee0d1836f1004c63315f876137a4644e0328 size: 945
took 23s ✘ hello-docker ◆ at 17:12:32 ○
took 12s ✘ hello-docker ◆ at 17:13:22 ○
```

- **Verify that the image is available on Docker Hub and share it with others.**



```
Last login: Sun Sep 22 16:48:03 on ttys001
base ✘ at 17:15:23 ⚡
$ docker pull diable201/go-multi-stage
Using default tag: latest
latest: Pulling from diable201/go-multi-stage
Digest: sha256:b0e6bf01d4f90cd93fb88dc2a37ee0d1836f1004c63315f876137a4644e0328
Status: Image is up to date for diable201/go-multi-stage:latest
docker.io/diable201/go-multi-stage:latest

What's next:
View a summary of image vulnerabilities and recommendations → docker scout quickview diable201/go-multi-stage
base ✘ at 17:15:32 ⚡
$ took 4s ✘ base ✘ at 17:15:32 ⚡
$ 42% 15% 7.9 GB -zsh
$ 22.9, 5:15 PM
```

3. Questions:

- **What is the purpose of Docker Hub in containerization?**

Answer: Docker Hub is a cloud-based registry where users can store, share, and distribute container images. Users can push their own images, either as public repositories or private repositories. It's easy to pull images onto any system, enabling consistent environments.

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

Answer: We use command docker tag local_image username/repository:tag, like docker tag go-multi-stage diable201/go-multi-stage:latest

- What steps are involved in pushing an image to Docker Hub?

Answer: Firstly, we need login using docker login command and entering username with password for Docker Hub, next we need tag our local image, for example docker tag go-multi-stage diable201/go-multi-stage:latest. And then we can push our tagged image to Docker Hub, for example docker push diable201/go-multi-stage:latest. We can verify our image in Docker Hub by visiting site.