Makhatbek Iliyas
Assignment 1, Web Application Development

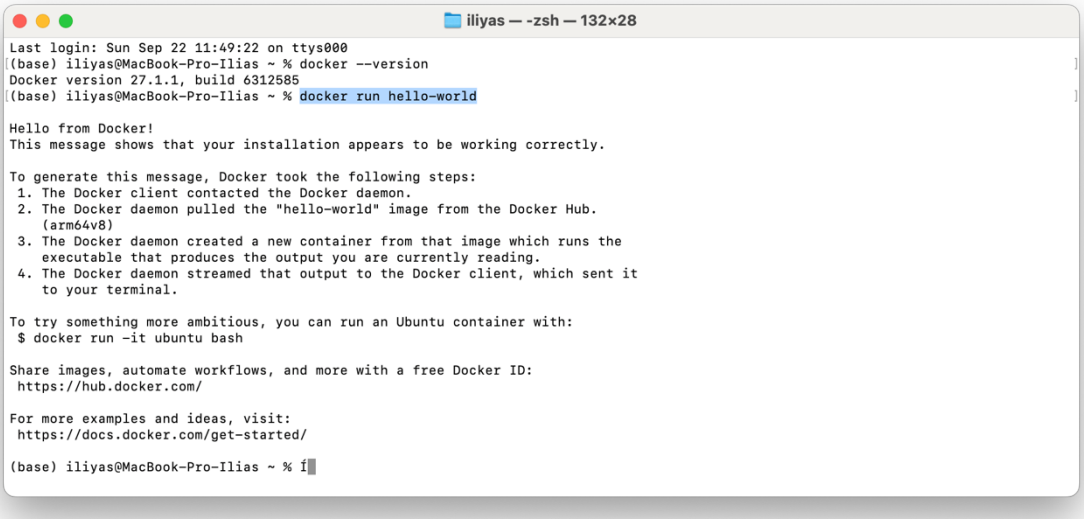
Part I - Intro to Containerization: Docker
Exercise 1: Installing Docker

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

Process:

`docker --version`

`docker run hello-world`



3. **Questions and answers**:
    - What are the key components of Docker (e.g., Docker Engine, Docker CLI)?

Docker Engine: This is the core part of Docker. It helps us run and manage containers. It includes a server that runs the containers, an API, and a command line interface.

Docker CLI: This is the command-line interface we use to interact with Docker. We type commands here to manage our containers and images.

Docker Images: These are the files that contain everything needed to run a container, like code, libraries, and settings.

Docker Containers: These are the running instances of Docker images. We can think of them as lightweight, isolated environments where our applications run.

Docker Hub: This is a cloud service where we can store and share our Docker images.

- How does Docker compare to traditional virtual machines(VMs)?

Lightweight: Docker containers share the host OS kernel, making them faster and lighter compared to VMs, which have their own full OS.
Boot Time: Containers start much quicker (seconds) compared to VMs (minutes).
Resource Usage: Docker uses fewer resources (CPU, memory) than VMs.
Isolation: VMs offer stronger isolation since each VM runs a full OS, while containers share the host OS, so isolation is at the process level.
Size: Docker images are smaller compared to full VM images because they don't include an entire OS.

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

Output:

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

The output means Docker pulled the image, ran the container, and confirmed that setup is working correctly.

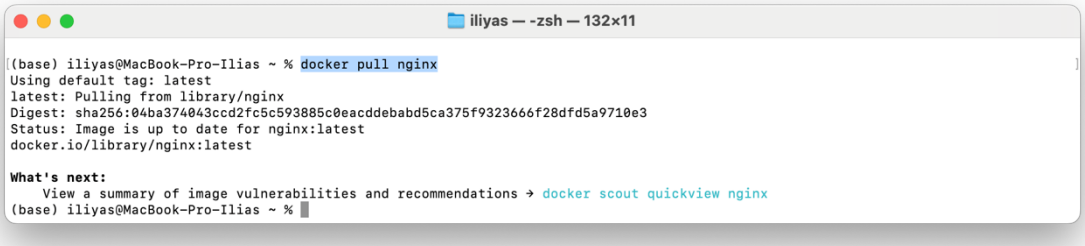
Exercise 2: Basic Docker Commands

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>`.

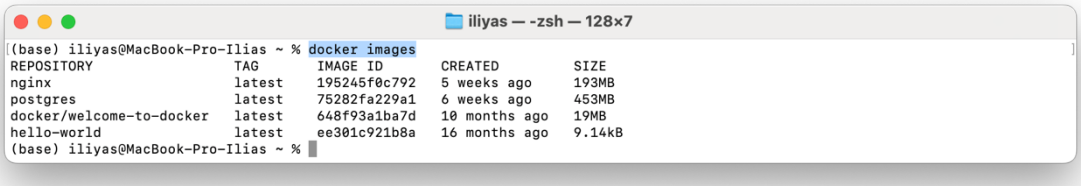
- List all Docker images on your system using `docker images`.
- Run a container from the pulled image using `docker run -d <image-name>`.
- List all running containers using `docker ps` and stop a container using `docker stop <container-id>`.

Process:

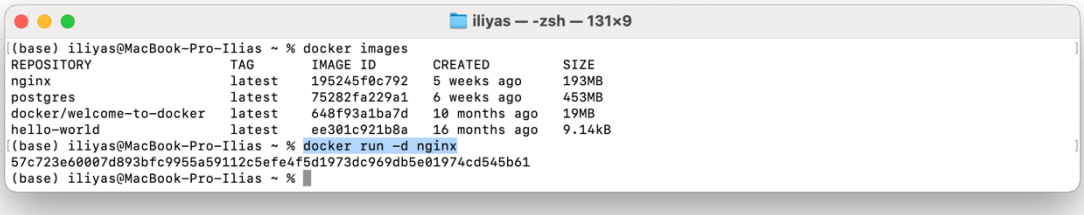
`docker pull nginx`



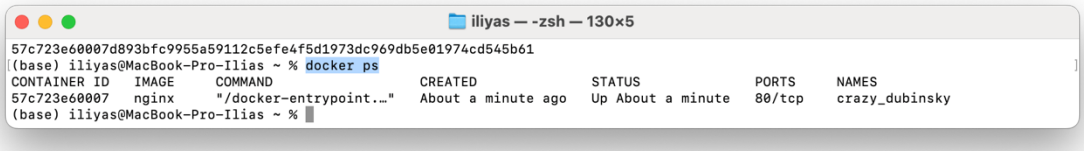
`docker images`



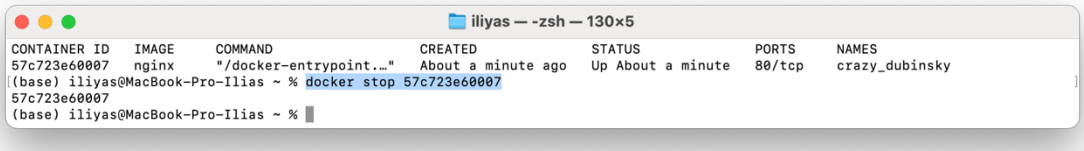
`docker run -d nginx`



`docker ps`



`docker stop 57c723e60007`



3. Questions:

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

`docker pull` is used to download a Docker image from a registry to local machine. It doesn't run anything; it just gets the image.

`docker run` takes that image and starts a container from it. This means it actually runs the application inside the container.

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

To find details about a running container, we can use the command `docker ps`. This command shows us a list of running containers, including their Ids, status and other info.

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

After we stop a container, it doesn't get deleted; it just stops running. We can restart it later using the command `docker start <container_id>`. So, the data in the container is still there, and we can pick up right where we left off.

Exercise 3: Working with Docker Containers

1. Objective: Learn how to manage Docker containers.

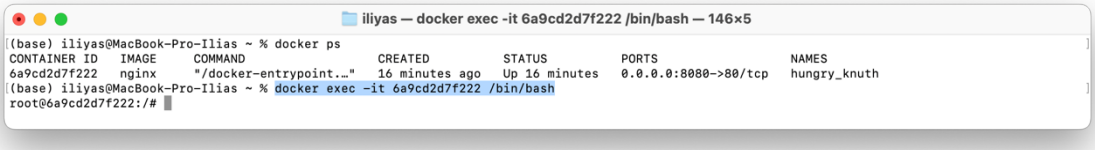
2. Steps:

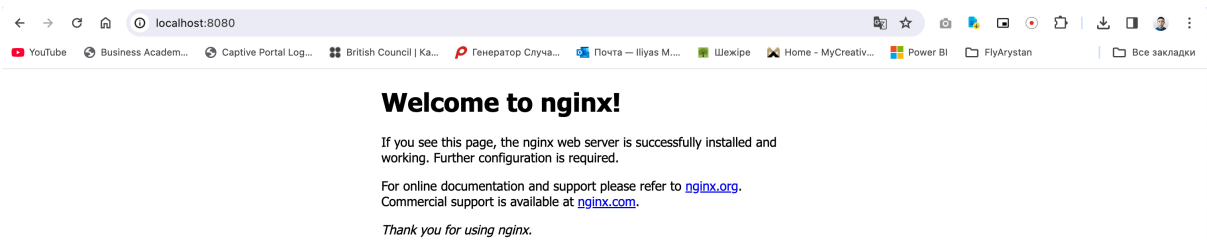
- Start a new container from the `nginx` image and map port 8080 on your host to port 80 in the container using `docker run -d -p 8080:80 nginx`.
- Access the Nginx web server running in the container by navigating to `http://localhost:8080` in your web browser.
- Explore the container's file system 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>`.

Process:

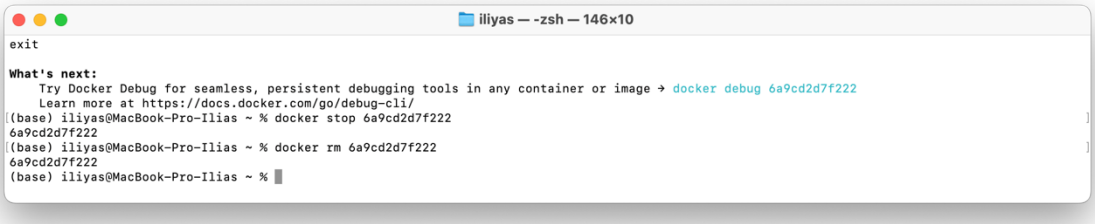
```
docker run -d -p 8080:80 nginx
```


```
docker exec -it 6a9cd2d7f222 /bin/bash
```





```
exit
docker stop 6a9cd2d7f222
docker rm 6a9cd2d7f222
```



3. Questions:

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

Port mapping in Docker lets us connect the ports on our host machine to the ports in our container. When we run a container, we can specify which port on the host should link to which port in the container. For example, if our app in the container runs on port 80, we can map it to port 8080 on the host. This is important because it allows us to access the services running inside the container from outside, like using a web browser to reach our app.

- What is the purpose of the `docker exec` command?

The `docker exec` command lets us run a command inside a running container. We can use it to open a shell or run scripts without stopping the container. This is useful for troubleshooting or making changes without needing to restart the whole container.

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

To make sure a stopped container doesn't use system resources, we can remove it using the `docker rm <container-id>` command. This frees up the resources that the container was using.

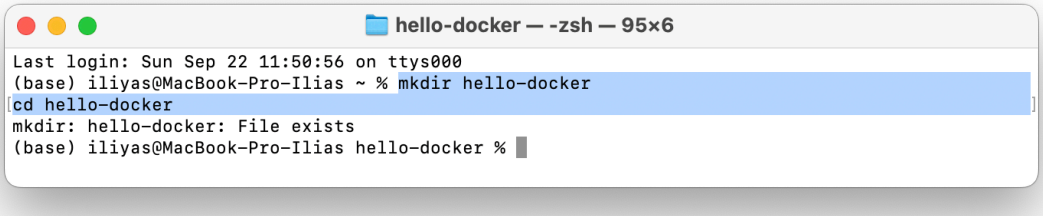
Part II – Dockerfile

Exercise 1: Creating a Simple Dockerfile
1. Objective: Write a Dockerfile to containerize a basic application.
2. Steps:

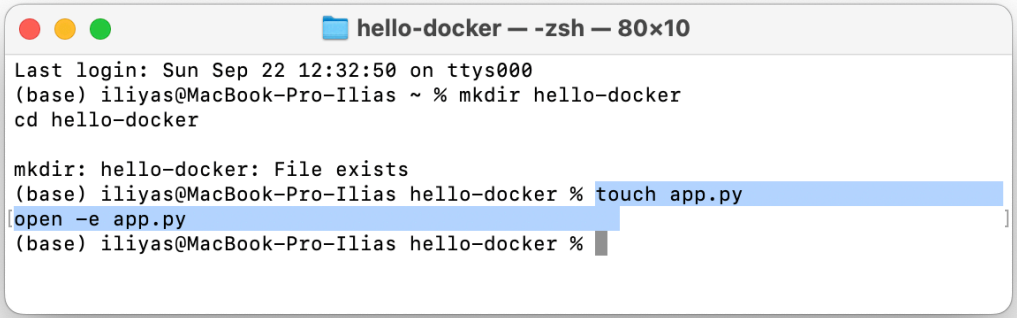
- Create a new directory for your project and navigate into it.
- Create a simple Python script (e.g., `app.py`) that prints "Hello, Docker!" to the console.
- Write a Dockerfile that:
  - Uses the official Python image as the base image.
  - Copies `app.py` into the container.
  - Sets `app.py` as the entry point for the container.
- Build the Docker image using `docker build -t hello-docker .`.
- Run the container using `docker run hello-docker`.

Process:

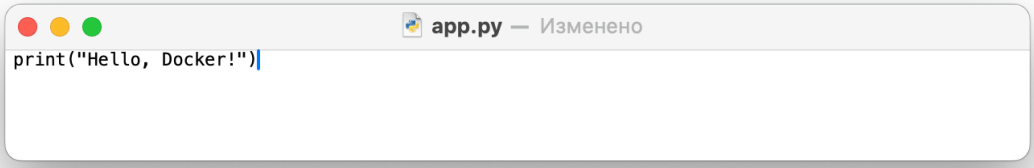
```
mkdir hello-docker
cd hello-docker
```



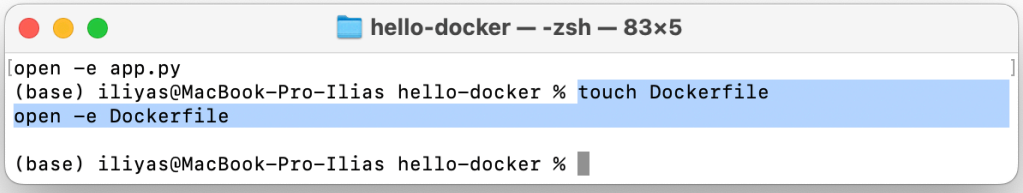

```
touch app.py
open -e app.py
```



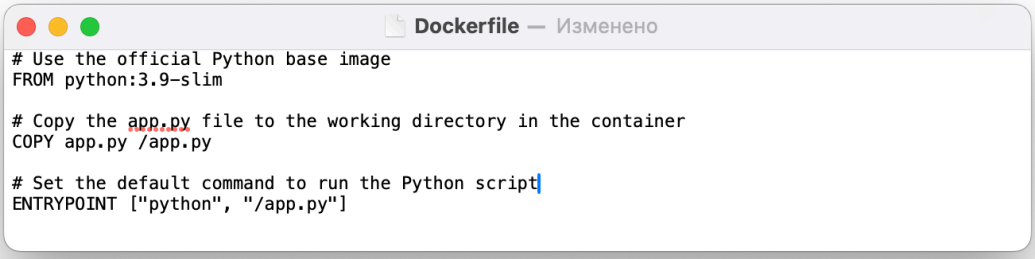
```
print("Hello, Docker!")
```



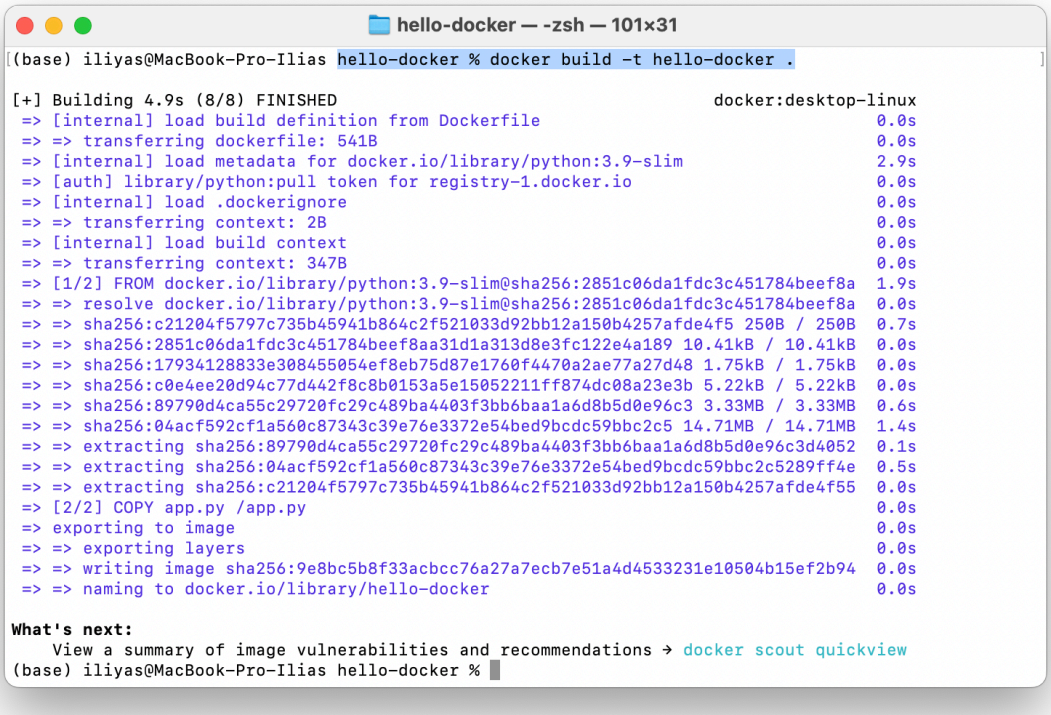

```
touch Dockerfile
open -e Dockerfile
```



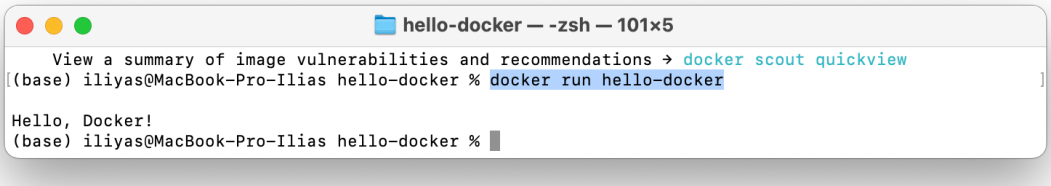

```
FROM python:3.9-slim
COPY app.py /app.py
ENTRYPOINT ["python", "/app.py"]
```



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




```
docker run hello-docker
```



3. Questions:

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

We use the `FROM` instruction to specify the base image we want to use for our Docker container. It's like starting with a template.

- How does the `COPY` instruction work in Dockerfile?

The `COPY` instruction is used to copy files or folders from our computer into the Docker image. We tell Docker where the files are on our machine and where we want them to go inside the image.

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

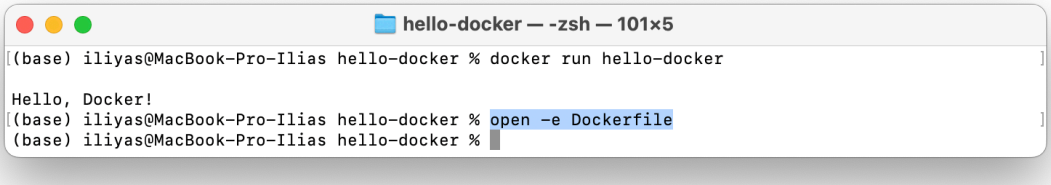
`CMD` provides default arguments to the `ENTRYPOINT` command, while `ENTRYPOINT` sets the main command that will always run. If we need to change what the container runs, we can do that easily with `CMD`, but `ENTRYPOINT` is more fixed.

Exercise 2: Optimizing Dockerfile with Layers and Caching

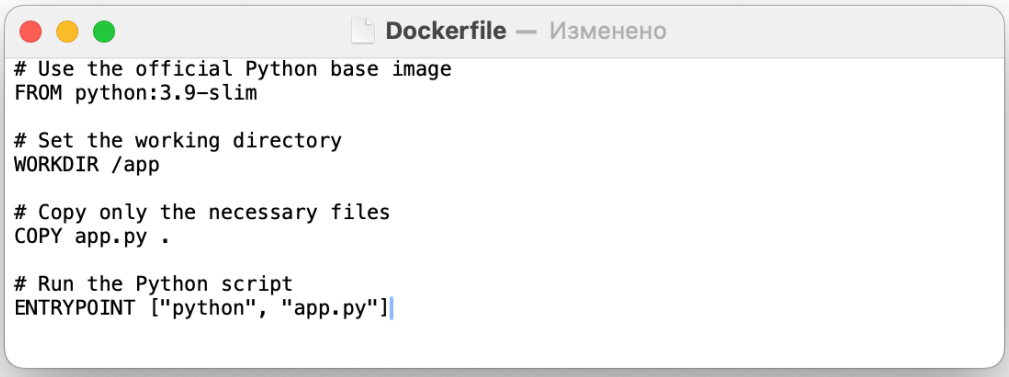
1. **Objective**: Learn how to optimize a Dockerfile for smaller image sizes and faster builds.
2. **Steps**:
   - Modify the Dockerfile created in the previous exercise to:
     - Separate the installation of Python dependencies (if any) from the copying of application code.
     - Use a `.dockerignore` 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.

Process:

```
open -e Dockerfile
```

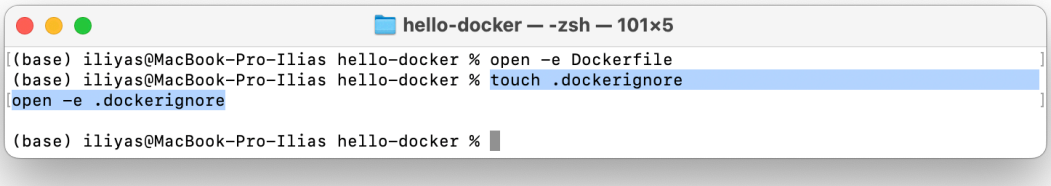



```
FROM python:3.9-slim
WORKDIR /app
COPY app.py .
ENTRYPOINT ["python", "app.py"]
```

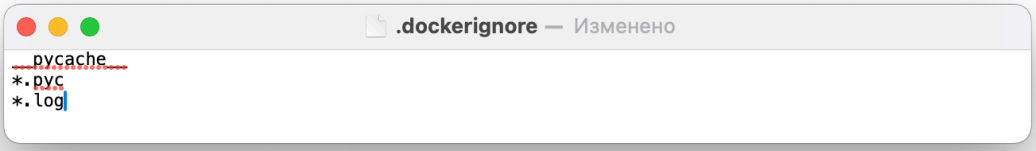



```
touch .dockerignore
```

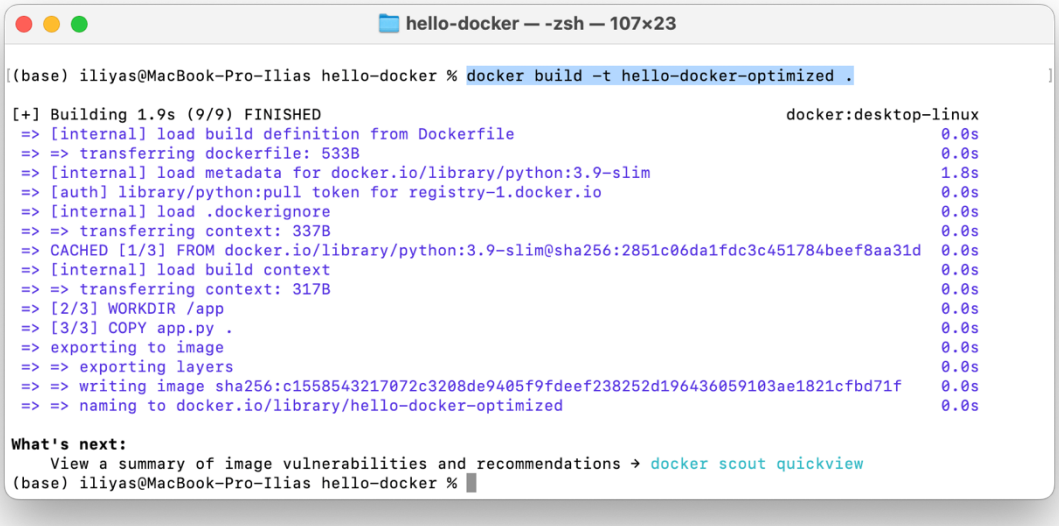
```
open -e .dockerignore
```



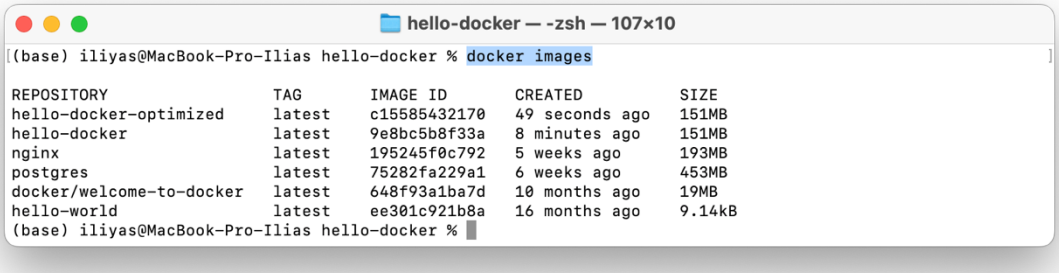

```
__pycache__
*.pyc
*.log
```




```
docker build -t hello-docker-optimized .
```




```
docker images
```



3. Questions:

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

Docker images are made of layers. Each layer is a change, like adding a file or installing something. Layers help keep the image size smaller because if we use the same layer in different images, Docker doesn't save it again. This also makes building faster since Docker only rebuilds layers that changed.

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

Docker's build cache stores the results of our previous builds. When we run a build command, Docker checks if it has already created the same layer before. If it has, it reuses that layer instead of rebuilding it. This speeds up the build process because we don't have to start from scratch every time.

- What is the role of the `.dockerignore` file?

The `.dockerignore` file tells Docker which files and folders to ignore when building an image. We can use it to avoid sending unnecessary files, like temporary files or local configuration files, to the Docker daemon. This helps keep our images smaller and the build process faster by only including what we really need.

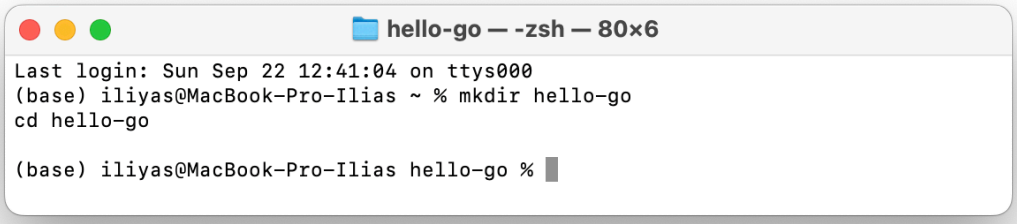
Exercise 3: Multi-Stage Builds

1. Objective: Use multi-stage builds to create leaner Docker images.
2. Steps:

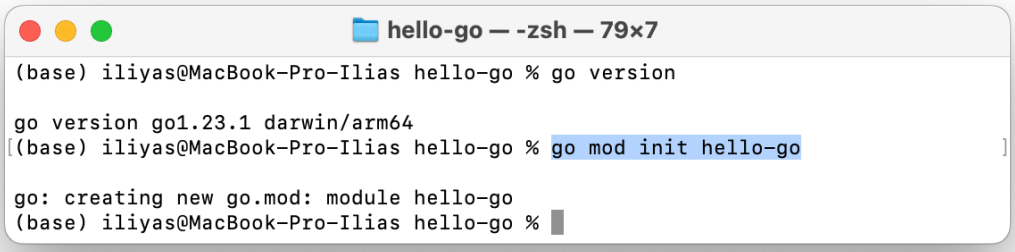
- Create a new project that involves compiling a simple Go application (e.g., a "Hello, World!" program).
- Write a Dockerfile that uses multi-stage builds:
  - The first stage should use a Golang image to compile the application.
  - The second stage should use a minimal base image (e.g., `alpine`) to run the compiled application.
- Build and run the Docker image, and compare the size of the final image with a single-stage build.

Process:

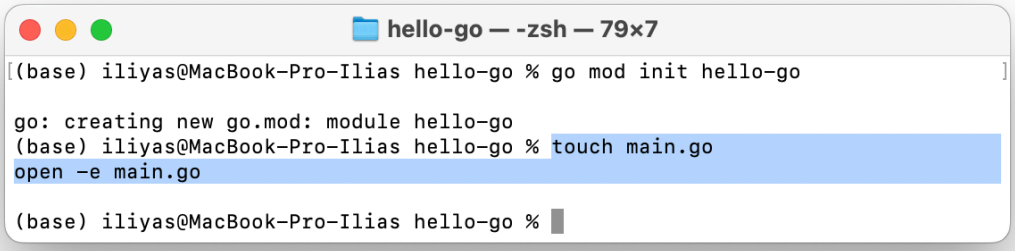
```
mkdir hello-go
cd hello-go
```



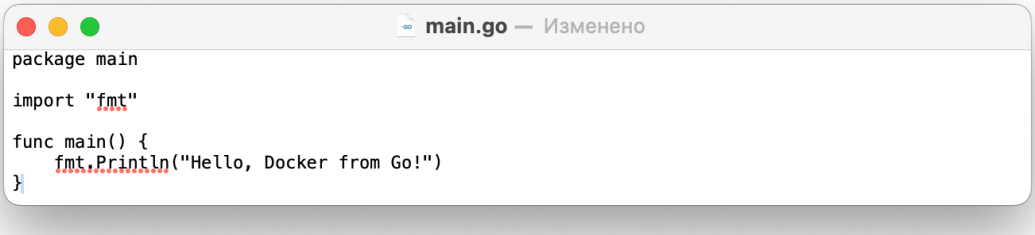

```
go mod init hello-go
```



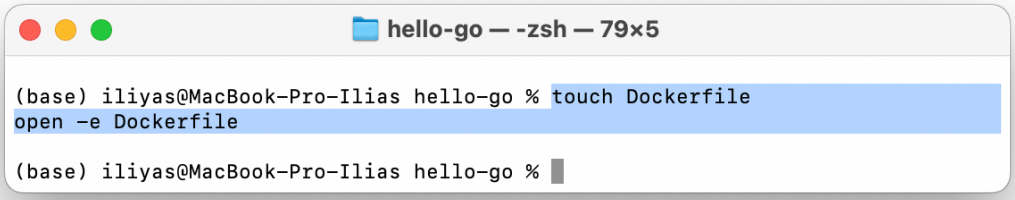

```
touch main.go
open -e main.go
```



```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, Docker from Go!")
}
```




```
touch Dockerfile
open -e Dockerfile
```

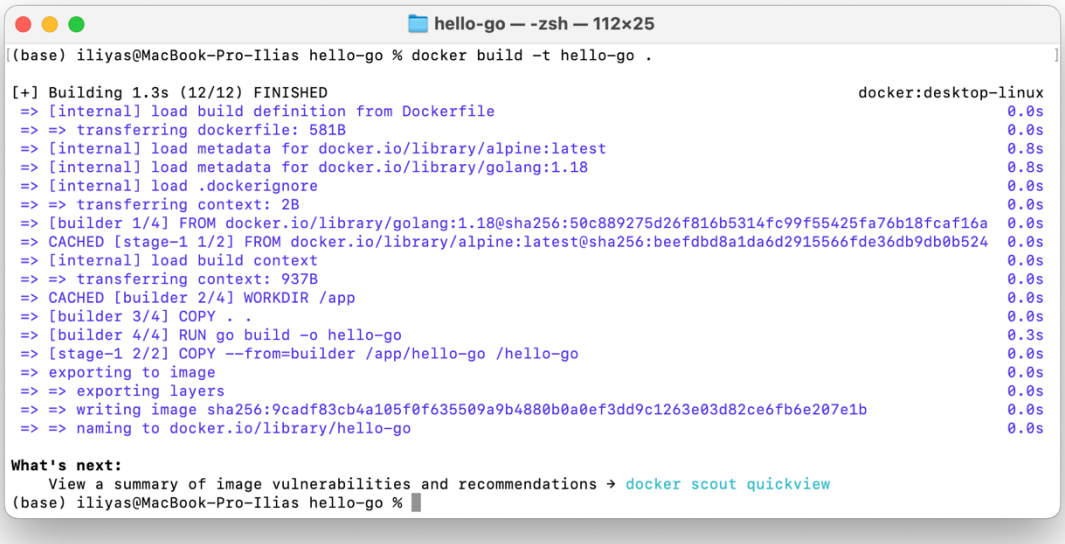



```
FROM golang:1.18 AS builder
WORKDIR /app
COPY . .
RUN go build -o hello-go

FROM alpine:latest
COPY --from=builder /app/hello-go /hello-go
ENTRYPOINT ["/hello-go"]
```

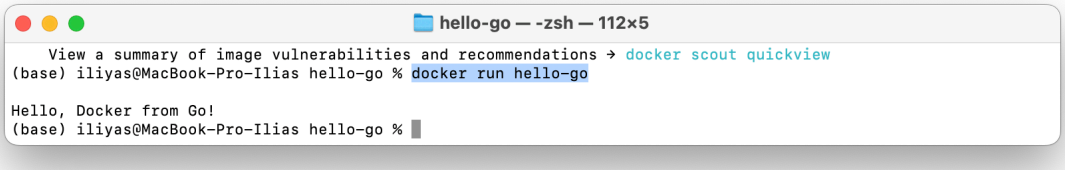
Dockerfile — Изменено

```
# Stage 1: Build the Go application
FROM golang:1.18 AS builder
WORKDIR /app
COPY . .
RUN go build -o hello-go

# Stage 2: Create a minimal image to run the compiled Go application
FROM alpine:latest
COPY --from=builder /app/hello-go /hello-go
ENTRYPOINT ["/hello-go"]
```

```
docker build -t hello-go .
```




```
docker run hello-go
```



3. Questions:

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

Multi-stage builds let us create smaller Docker images by separating the build process, which means we only include what's necessary in the final image. This helps reduce image size and speeds up downloads. It also makes our Dockerfile easier to manage, as each stage focuses on a specific task, improving organization and security by leaving out unnecessary tools.

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

In multi-stage builds, we can compile code in one stage and then copy only the necessary files to the final image. This means we don't include things like compilers or extra tools that are only needed for building, so the final image is smaller.

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

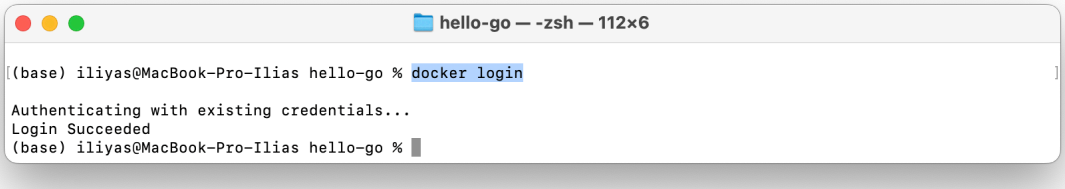
Compiling Code: When we write code that needs to be compiled (like C++ or Java), we can use one stage to compile it and another stage to run the compiled application.
Using Different Base Images: If we need different tools or environments for building and running, we can use one image for building (like a heavy image with compilers) and a lighter one for running.
Multi-Language Projects: If our project involves multiple programming languages, we can build each part in its own stage and then combine them in the final image.

Exercise 4: Pushing Docker Images to Docker Hub

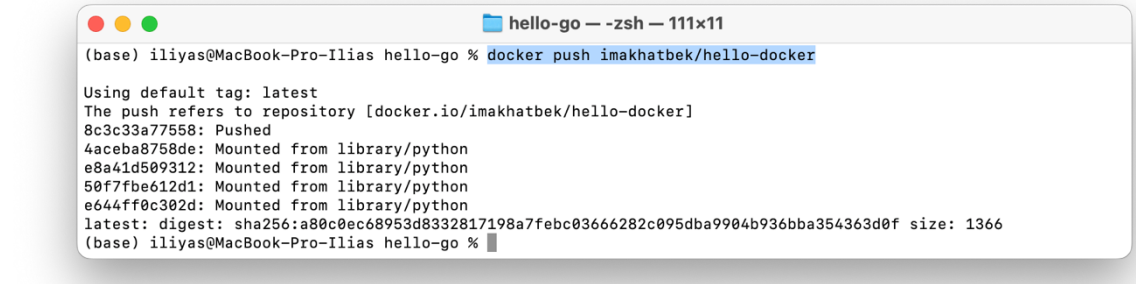
1. **Objective**: Learn how to share Docker images by pushing them to Docker Hub.
2. **Steps**:
    - Create an account on Docker Hub.
    - Tag the Docker image you built earlier with your Docker Hub username (e.g., `docker tag hello-docker <your-username>/hello-docker`).
    - Log in to Docker Hub 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.

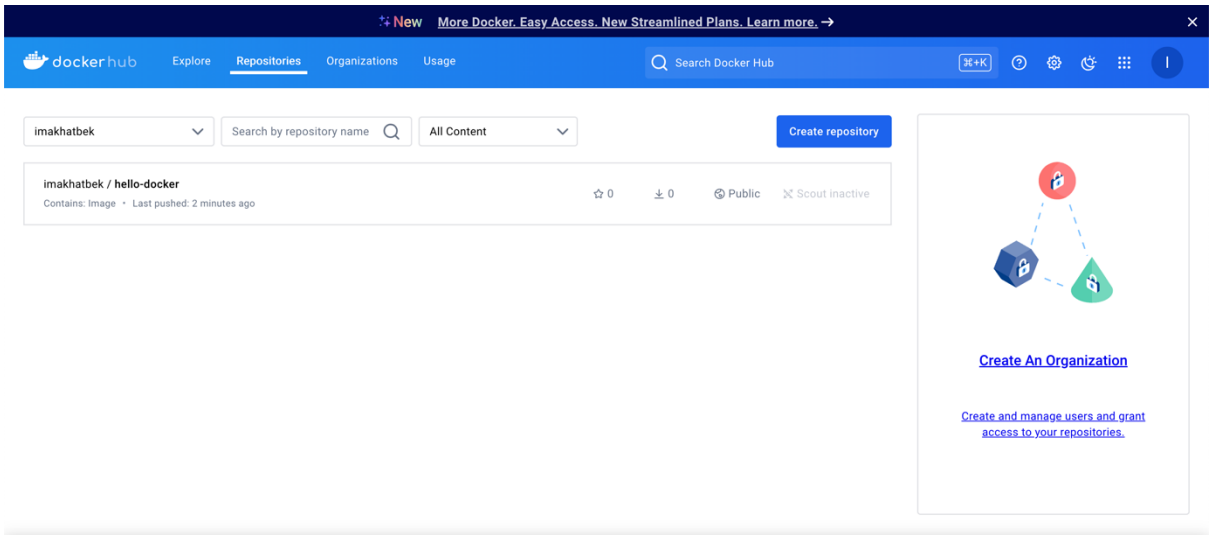
```
docker tag hello-docker imakhatbek/hello-docker
```


```
docker login
```



```
docker push imakhatbek/hello-docker
```





3. Questions:

- What is the purpose of Docker Hub in containerization?

Docker Hub is a place where we can store and share our Docker images. It allows us to easily find, download, and upload images so that we can use them on different computers or share them with others.

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

We tag a Docker image by using the command docker tag.
`docker tag <image-name> <username>/<repository-name>`. This helps us give our image a name that includes our Docker Hub username and the name of the repo where we want to store it.

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

1. Log in to Docker hub
   `docker login`

2. Tag the Image
   `docker tag <image-name> <username>/<repository-name>`

3. Push the image:
   `docker push <username>/<repository-name>`