KAZAKH-BRITISH TECHNICAL UNIVERSITY

Assignment #1
Web Application Development

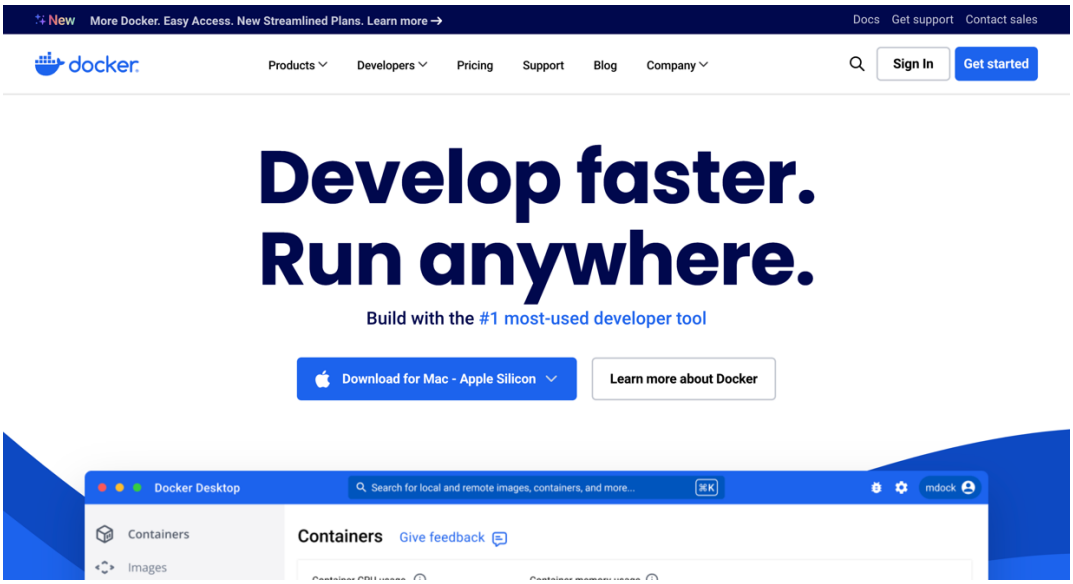
Prepared by:
Seitbekov S.
Checked by:
Serek A.

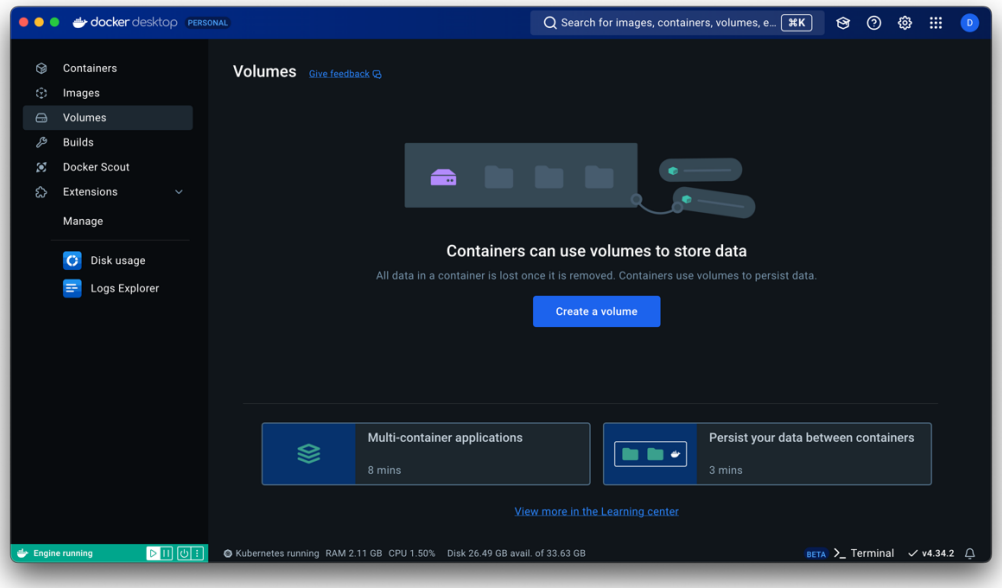
Almaty, 2024

Intro to Containerization: Docker

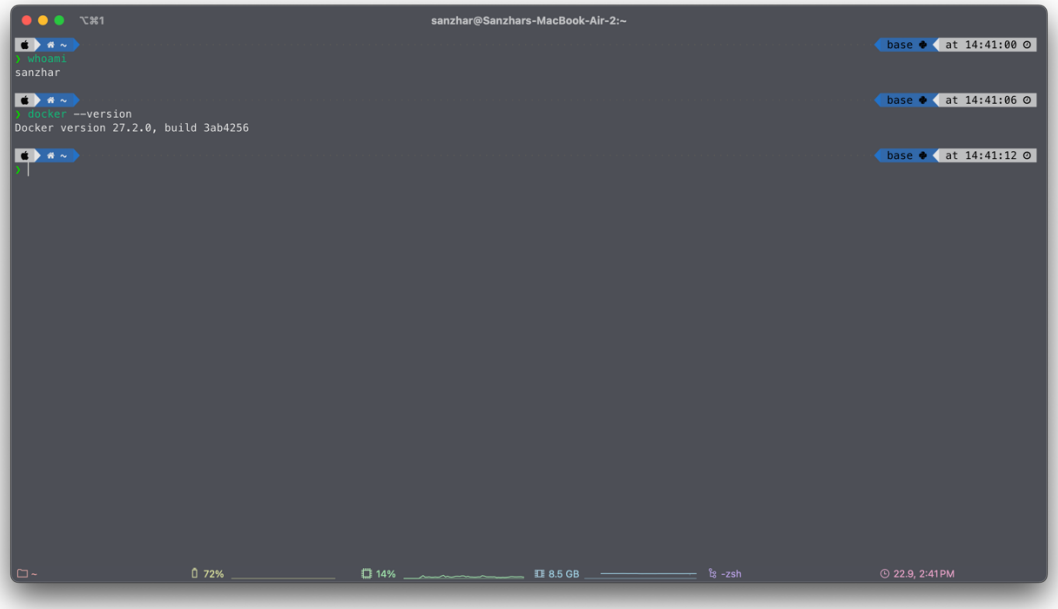
Exercise 1: Installing Docker

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).**

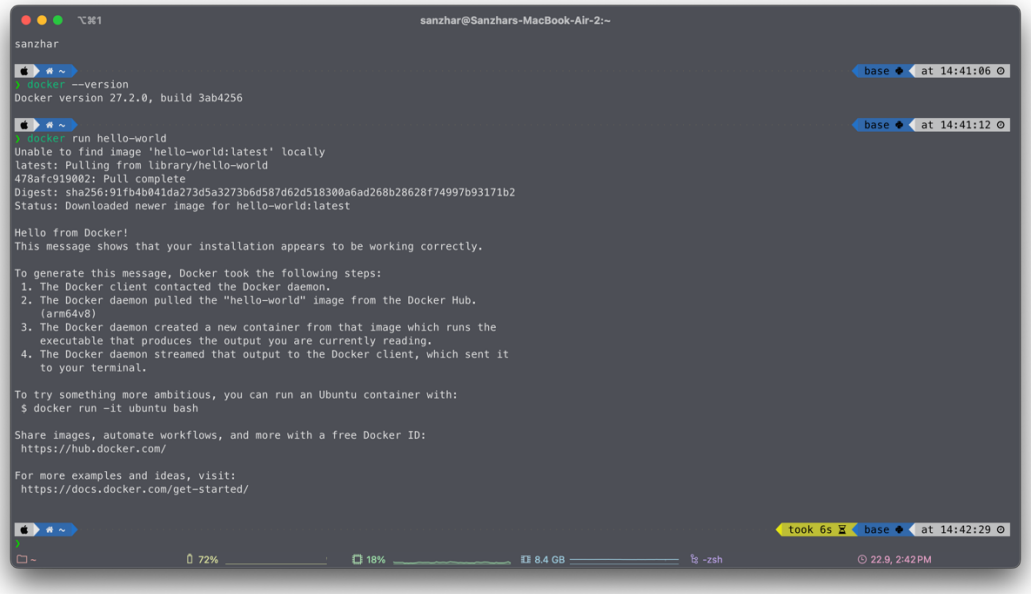


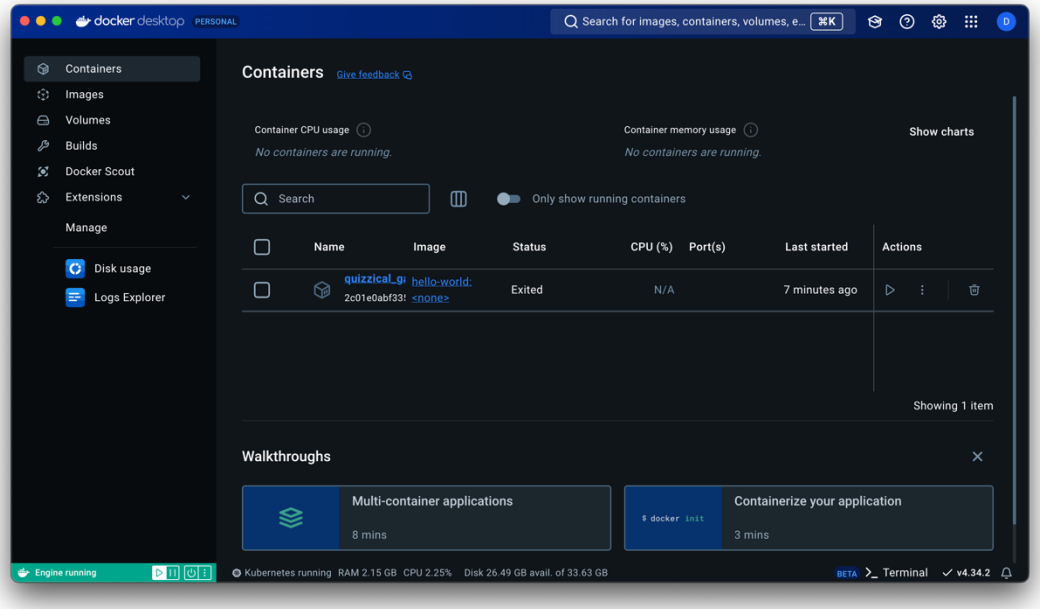


- **After installation, verify that Docker is running by executing the command docker --version in your terminal or command prompt.**



- **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)?**

     Answer: Key components of Docker are Docker Engine (core sofware 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?**

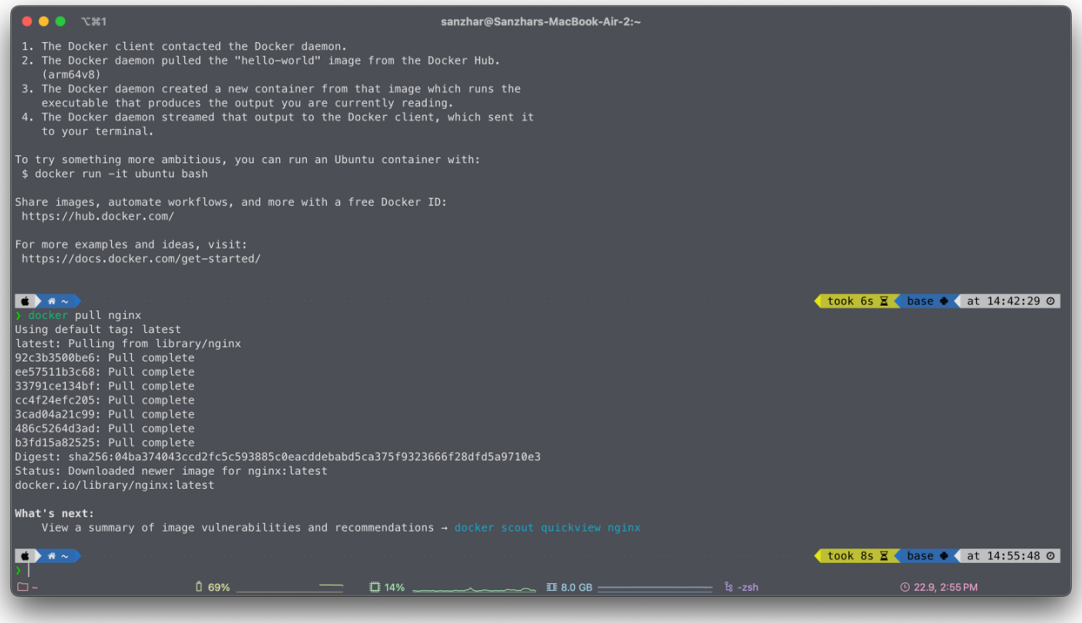
     Answer: A message confirming Docker is installed correctly, signifying the container ran successfully and Docker's client-server setup is working properly. Also, it pulls the image if there is no such image locally yet.

Exercise 2: Basic Docker Commands

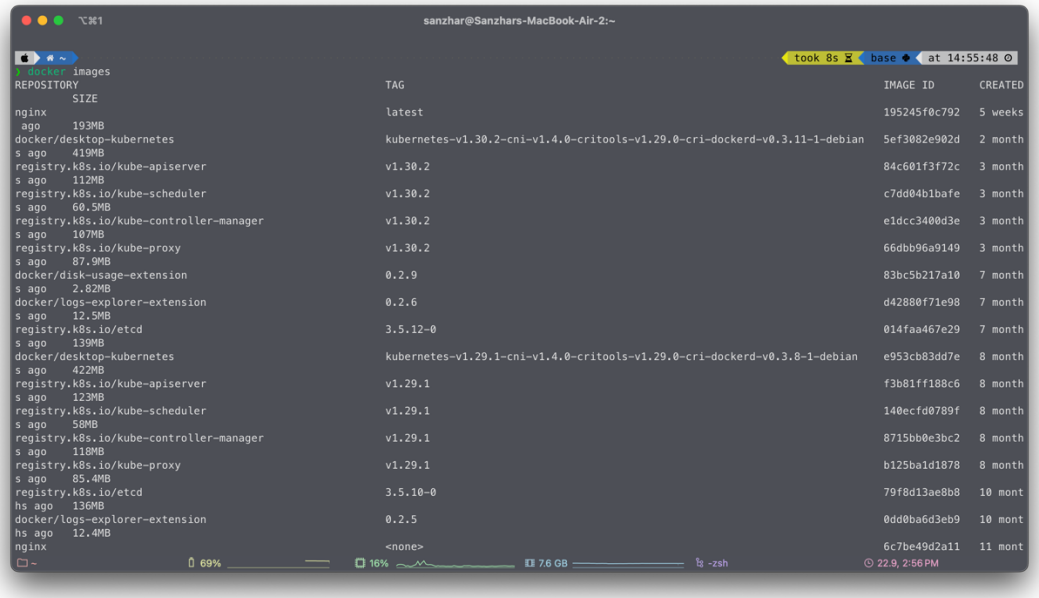
1. **Objective: Familiarize yourself with basic Docker commands.**

2. **Steps:**

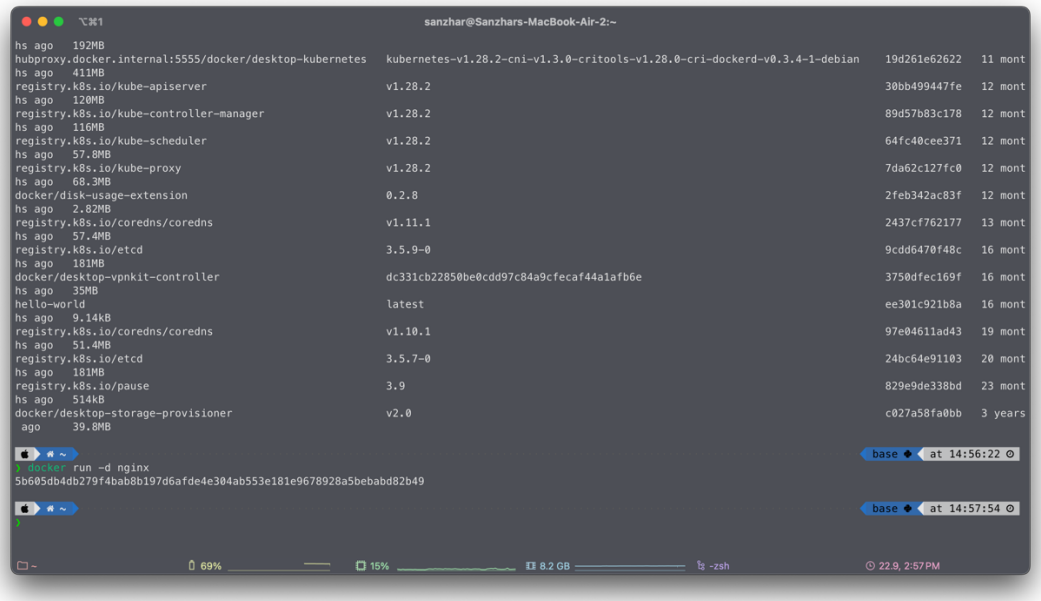
   - **Pull an official Docker image from Docker Hub (e.g., nginx or ubuntu) using the command docker pull <image-name>.**



   - **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?**

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?**

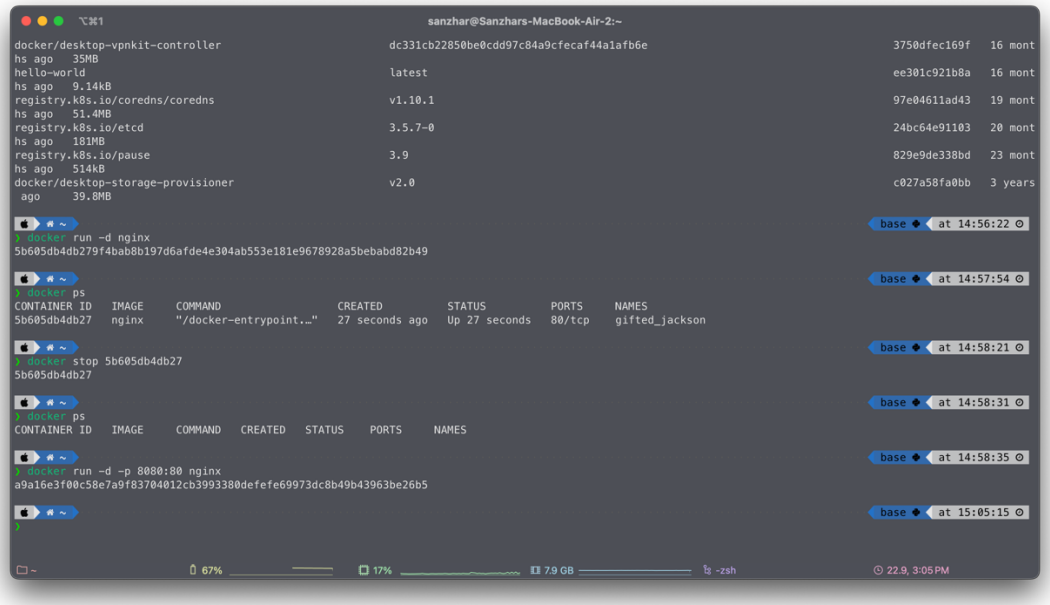
  Answer: The container still exists but is in a stopped state, it can be restarted using command docker start container_id

Exercise 3: Working with Docker Containers

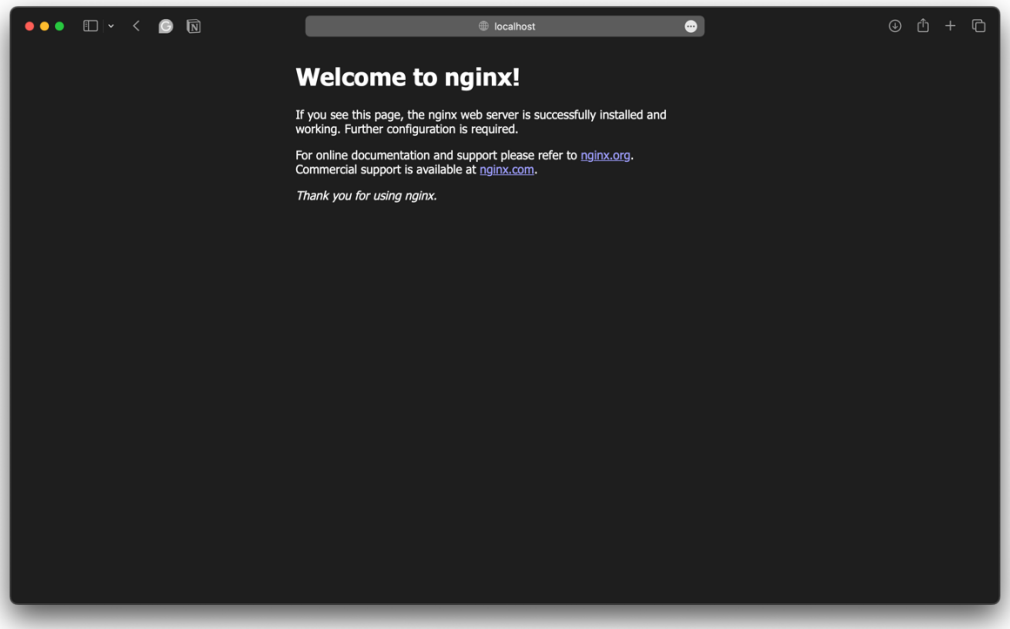
1. **Objective: Learn how to manage Docker containers.**

2. **Steps:**

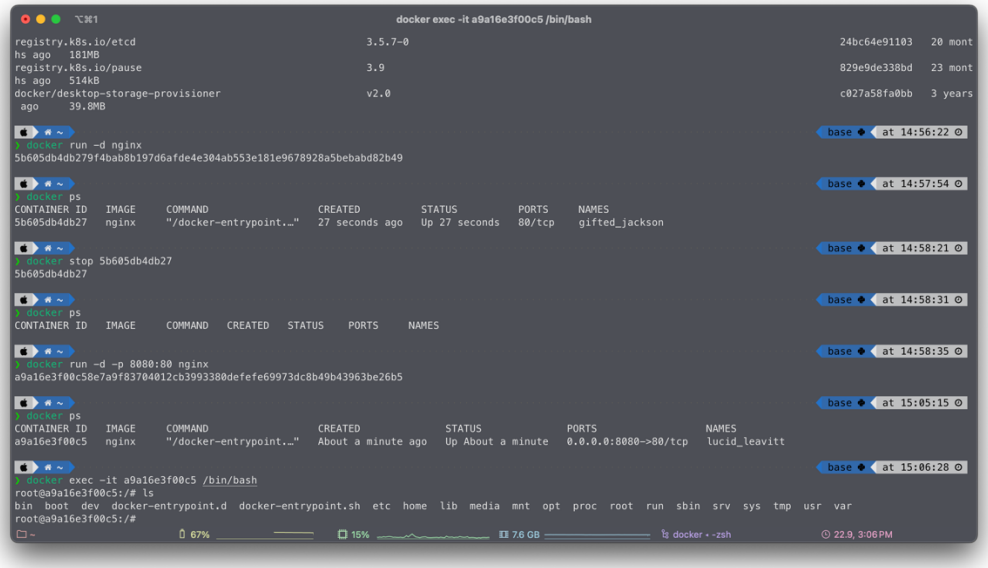
   - **Start a new container from the nginx image and map port 8080 on your host to port 80 in the container using docker run -d -p 8080:80 nginx.**



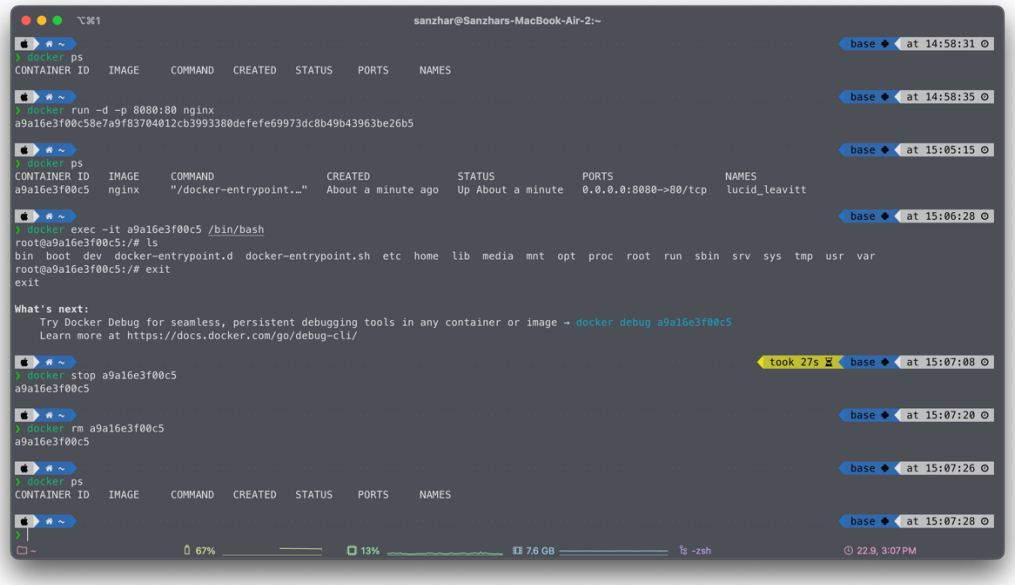
- **Access the Nginx web server running in the container by navigating to http://localhost:8080 in your web browser.**



- **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?**

  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).

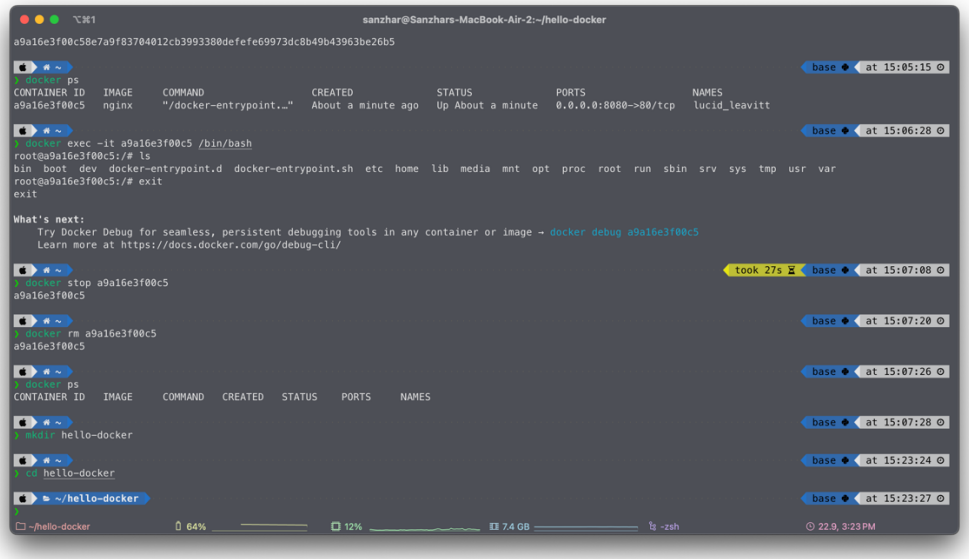
- **How do you ensure that a stopped container does not consume system resources?**

  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.

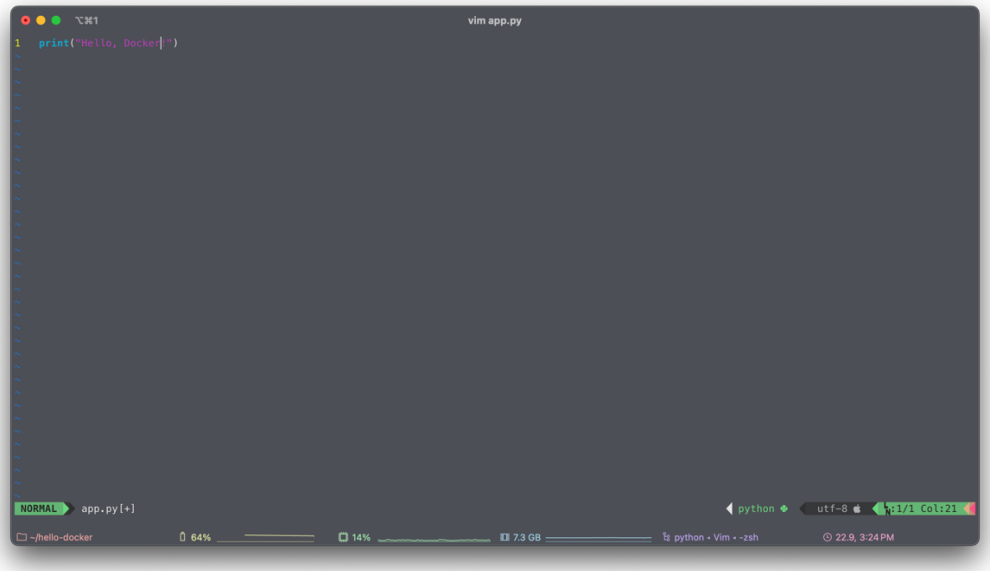
Dockerfile

Exercise 1: Creating a Simple Dockerfile

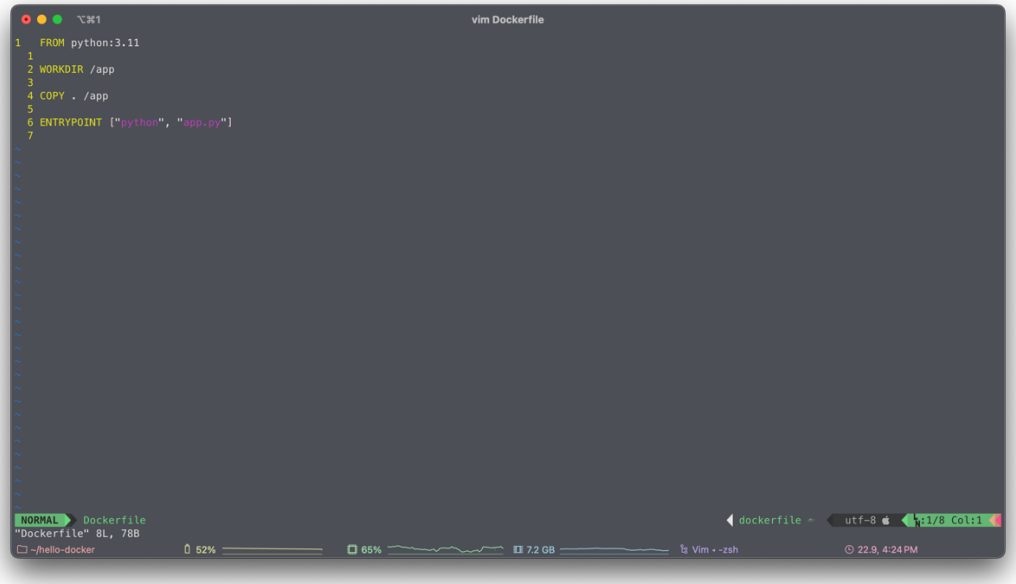
1. **Objective: Write a Dockerfile to containerize a basic application.**
2. **Steps:**
    - **Create a new directory for your project and navigate into it.**



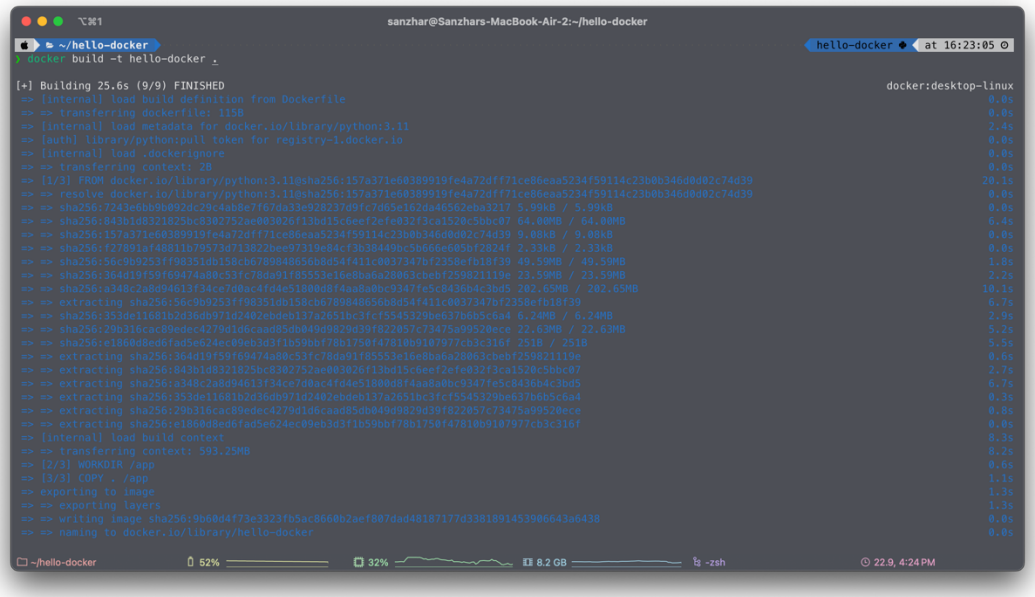
    - **Create a simple Python script (e.g., app.py) that prints "Hello, Docker!" to the console.**



- **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?**

     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.

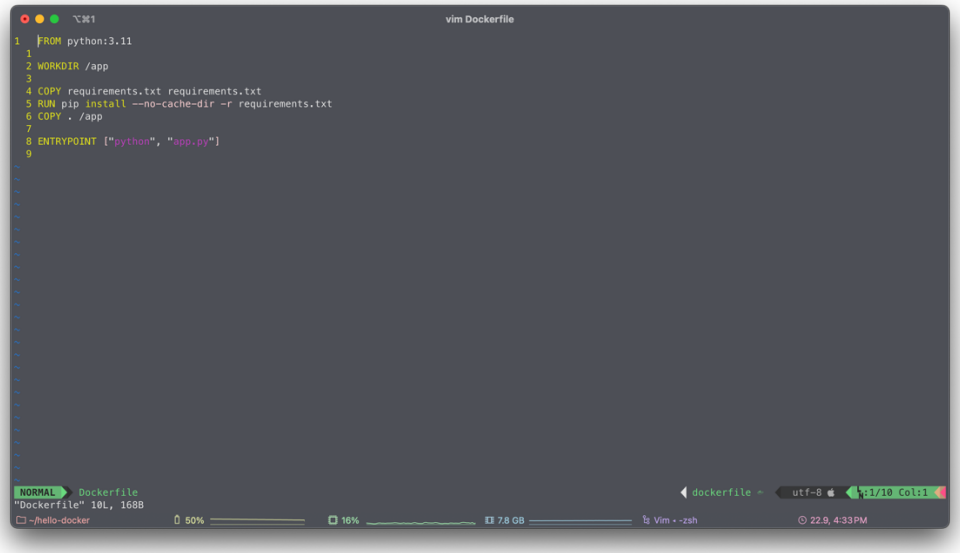
Exercise 2: Optimizing Dockerfile with Layers and Caching

1. **Objective: Learn how to optimize a Dockerfile for smaller image sizes and faster builds.**

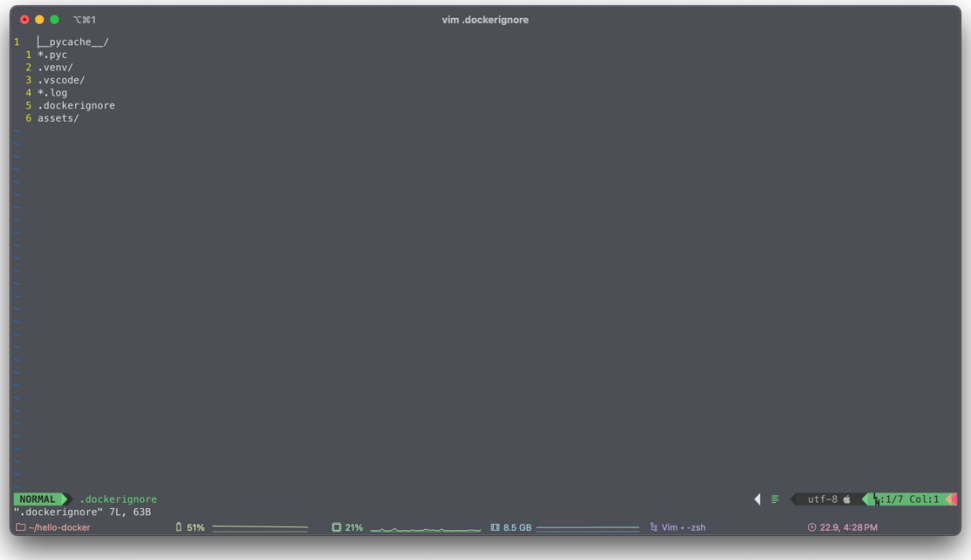
2. **Steps:**

    - **Modify the Dockerfile created in the previous exercise to:**

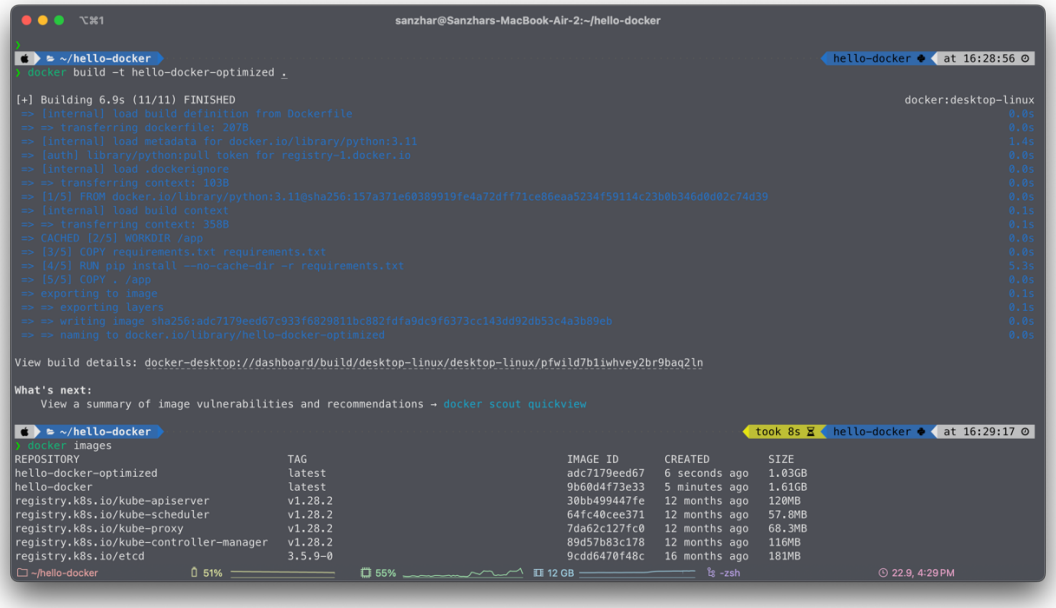
        - **Separate the installation of Python dependencies (if any) from the copying of application code.**



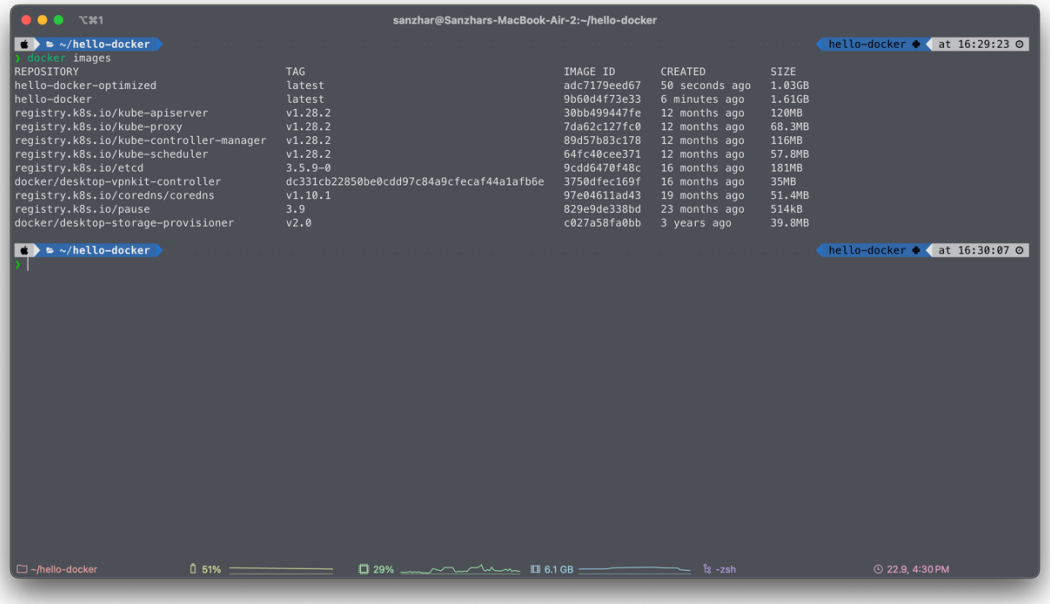
        - **Use a .dockerignore file to exclude unnecessary files from the image.**



- **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?**

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

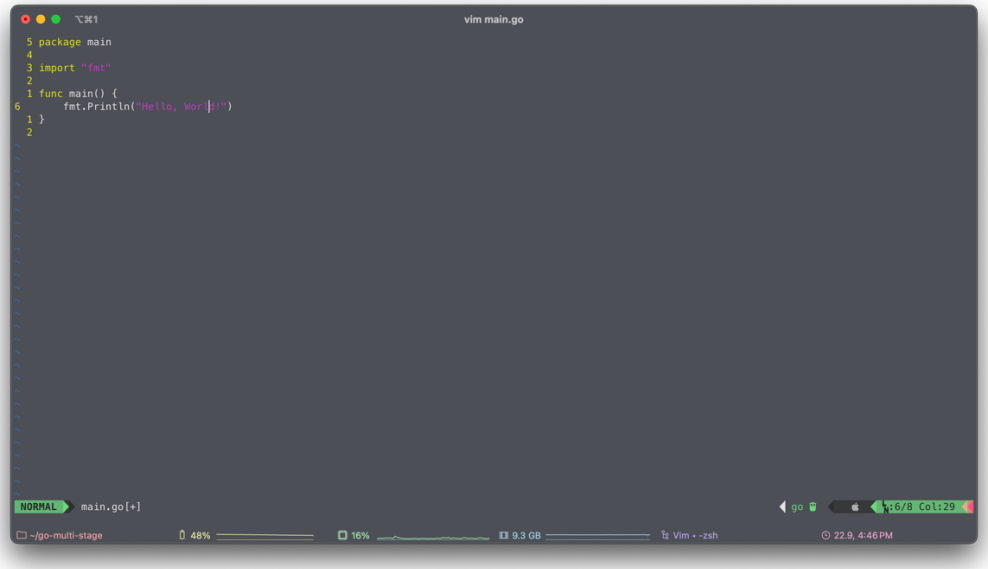
- **What is the role of the .dockerignore file?**

  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.

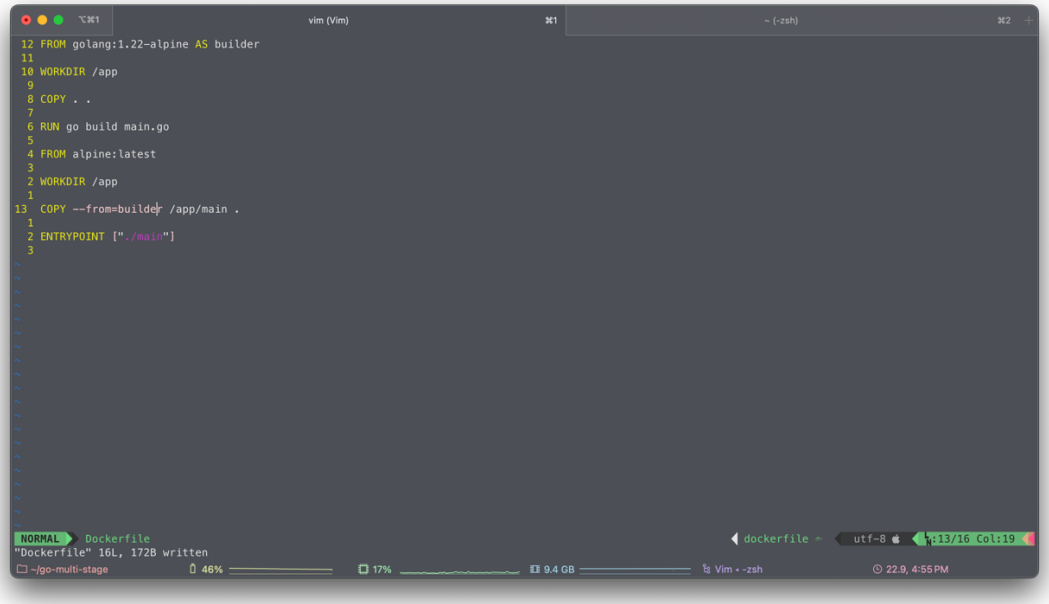
Exercise 3: Multi-Stage Builds

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

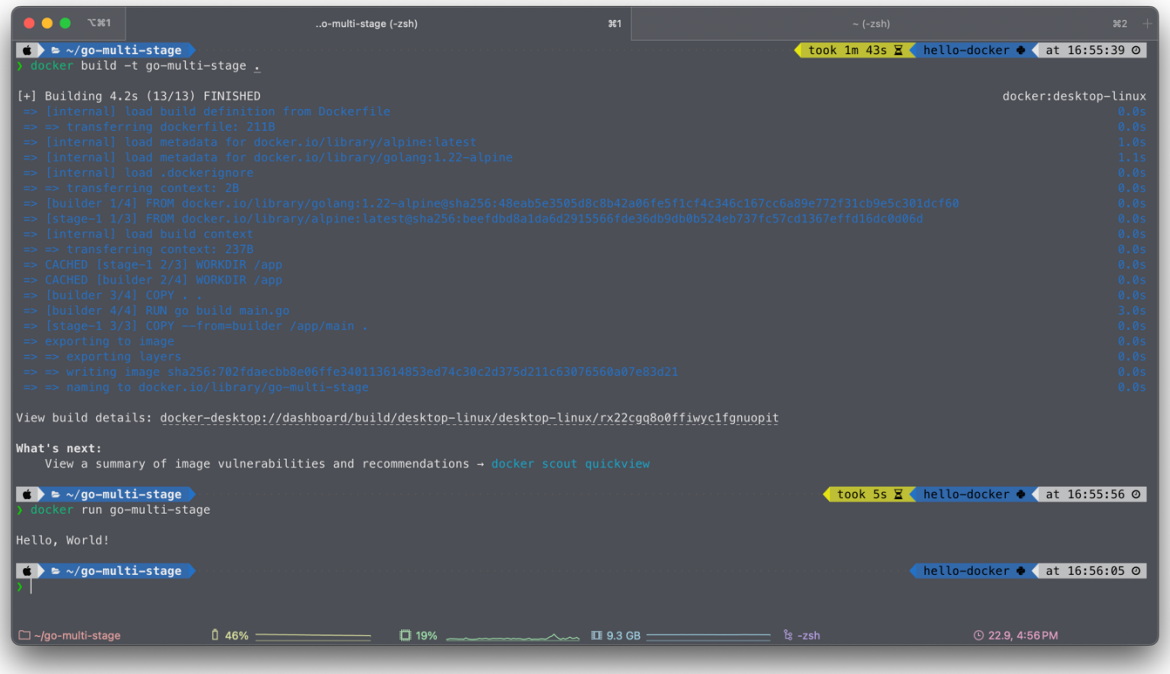
- **Create a new project that involves compiling a simple Go application (e.g., a "Hello, World!" program).**

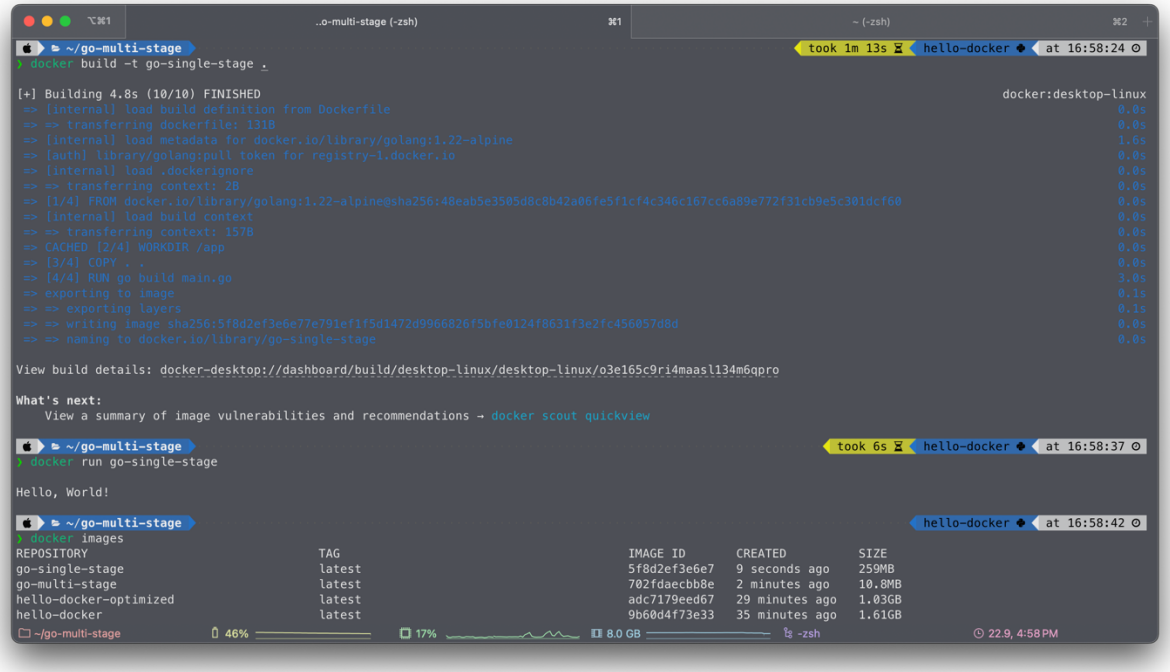


- **Write a Dockerfile that uses multi-stage builds:**
  - **The first stage should use a Golang image to compile the application.**
  - **The second stage should use a minimal base image (e.g., alpine) to run the compiled application.**



- **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?**

  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?**

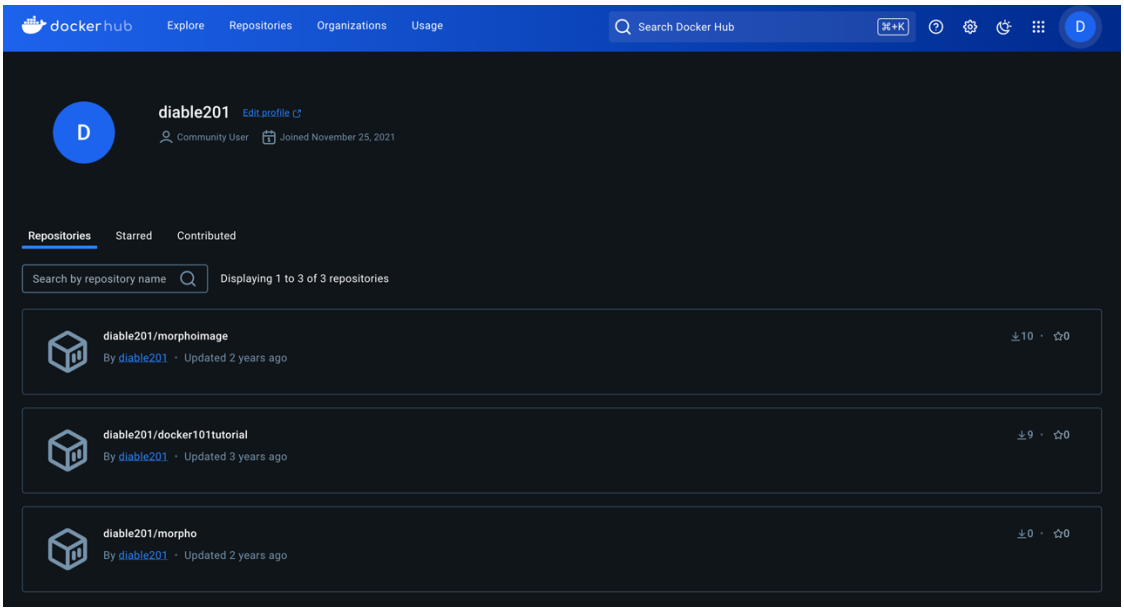
  Answer: Multi-stage builds are highly useful for languages like Go, Java, or C++ where we can compile code in the first stage and copy only the binary to the final runtime environment. Also multi-stage builds are useful for CI/CD pipelines and production containers.

Exercise 4: Pushing Docker Images to Docker Hub

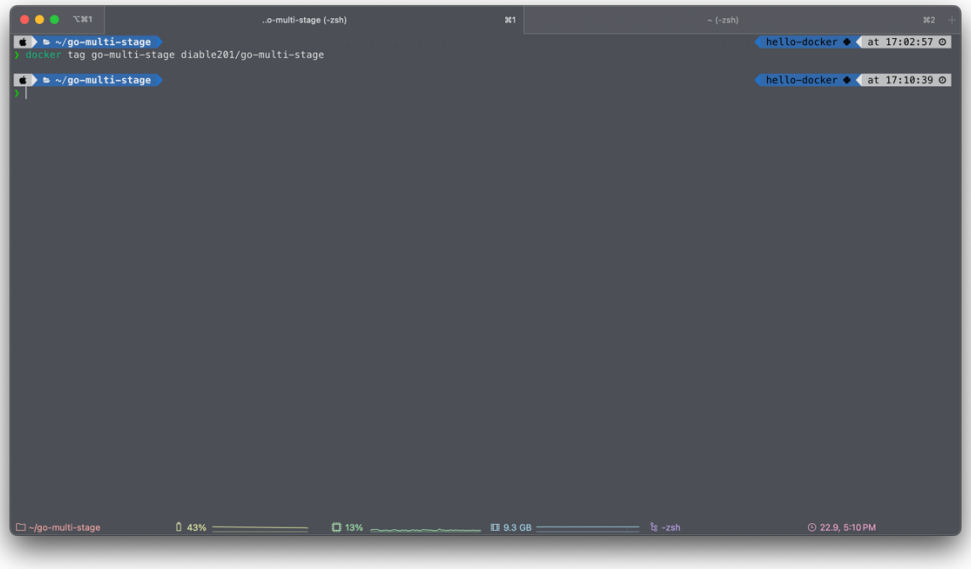
1. **Objective: Learn how to share Docker images by pushing them to Docker Hub.**

2. **Steps:**

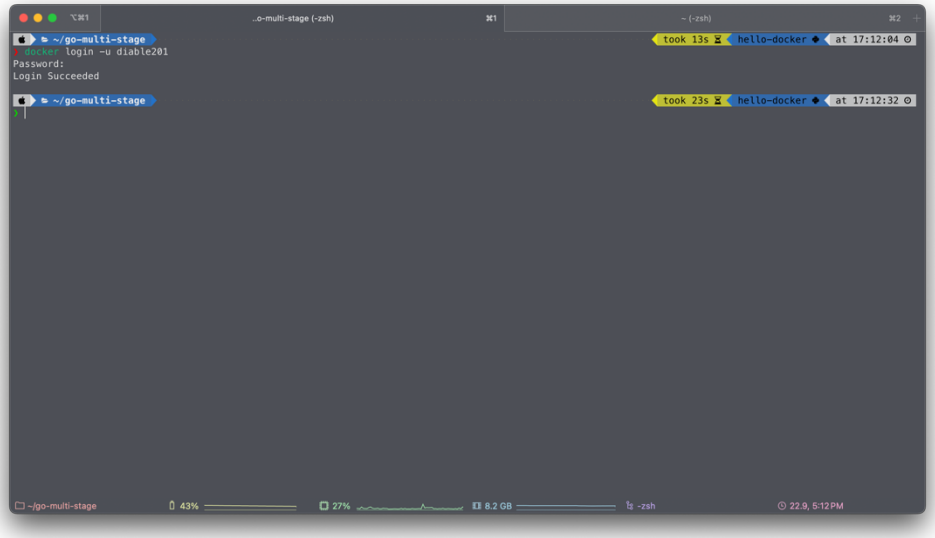
   - **Create an account on Docker Hub.**



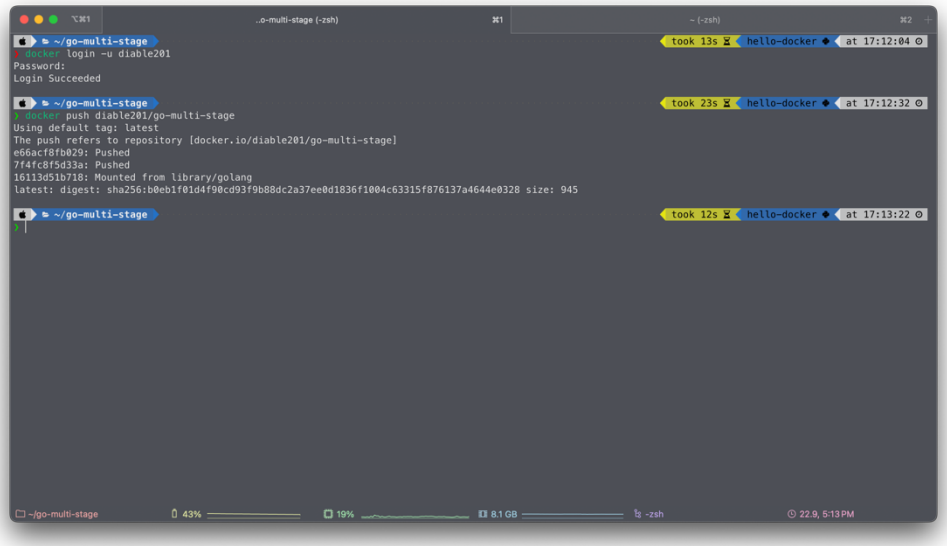
- **Tag the Docker image you built earlier with your Docker Hub username (e.g., docker tag hello-docker <your-username>/hello-docker).**



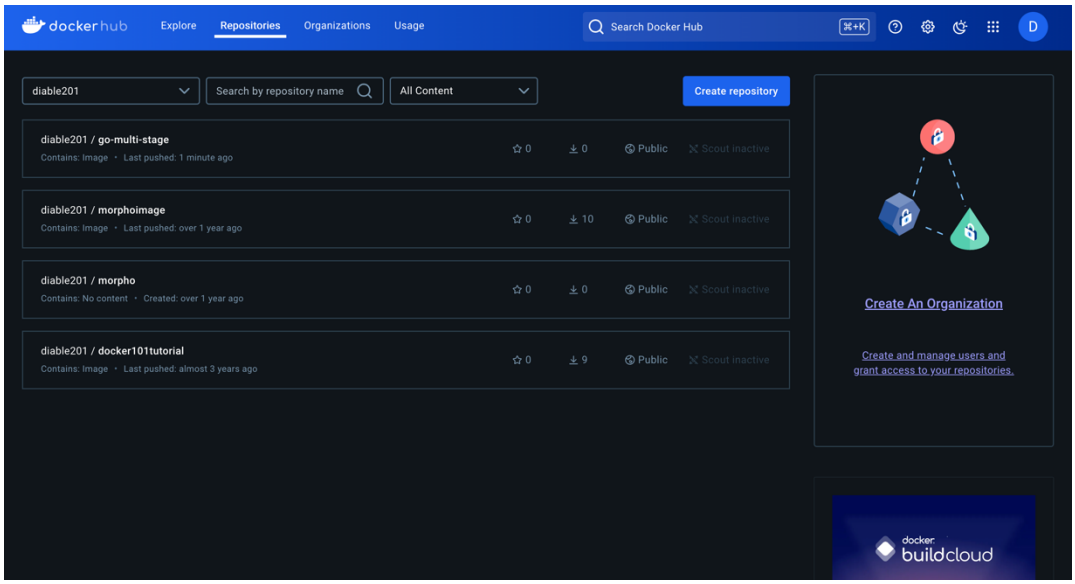
- **Log in to Docker Hub using docker login.**

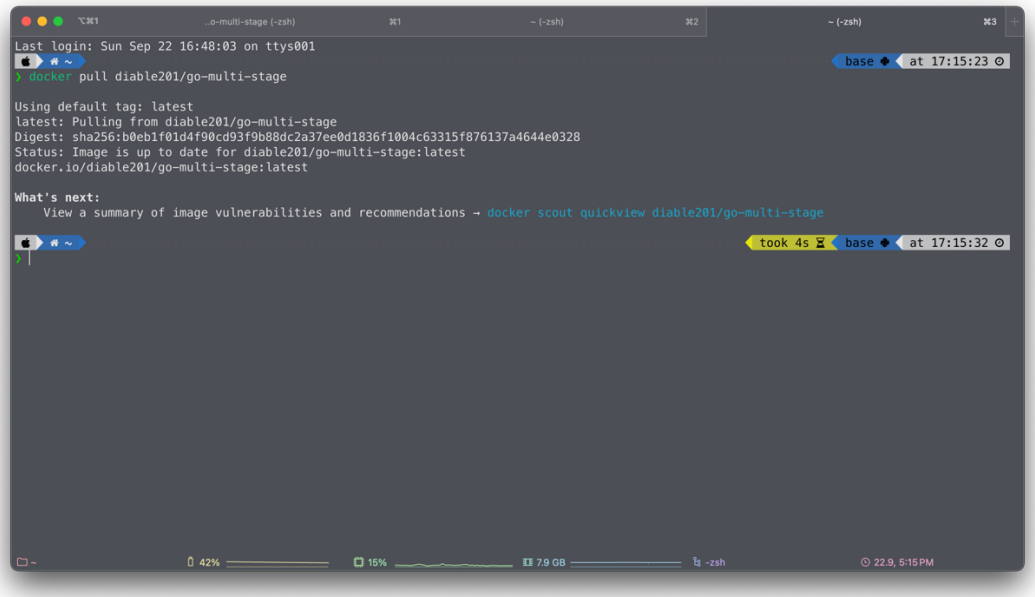


- **Push the image to Docker Hub using docker push <your-username>/hello-docker.**



- **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?**

  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.