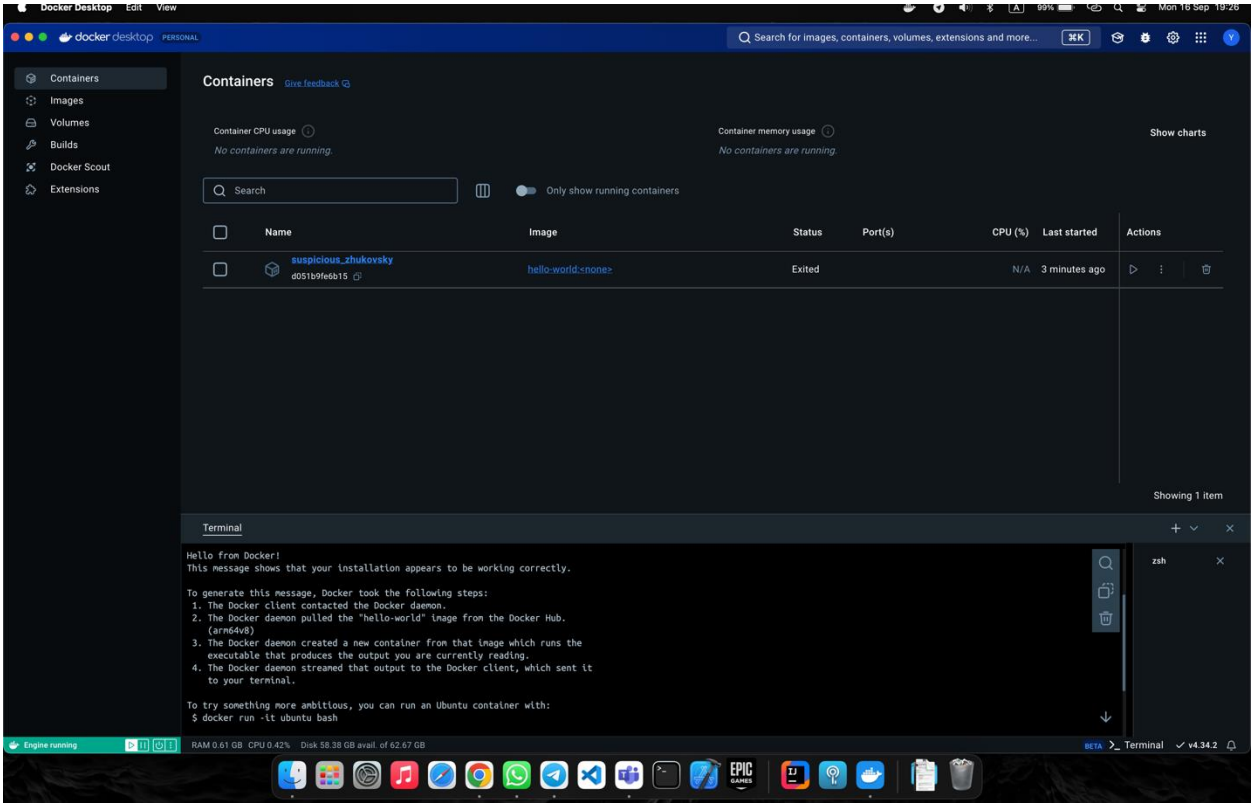
Assignment 1, Web Application Development

Put all deliverables into github repository in your profile. Share link to google form 24 hours before defense. Defend by explaining deliverables and answering questions.
Deliverables: report in pdf
Google form: https://docs.google.com/forms/d/e/1FAIpQLSe0GyNdOYlvM1tX_I_CtlPod5jBf-ACLGdHYZq1gVZbUeBzIg/viewform?usp=sf_link

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)?

- How does Docker compare to traditional virtual machines?
- What was the output of the `docker run hello-world` command, and what does it signify?

Answers:

1. A server with a long-running daemon process dockerd. APIs which specify interfaces that programs can use to talk to and instruct the Docker daemon. A command line interface (CLI) client docker.

2. Docker uses containerization, sharing the host OS kernel, making it lightweight, fast, and resource-efficient. It's ideal for microservices and quick deployment. Virtual machines use full virtualization, running a complete OS per instance, offering stronger isolation and security but with more resource overhead. VMs are better for running different OS instances or legacy applications requiring full isolation.

3.

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.

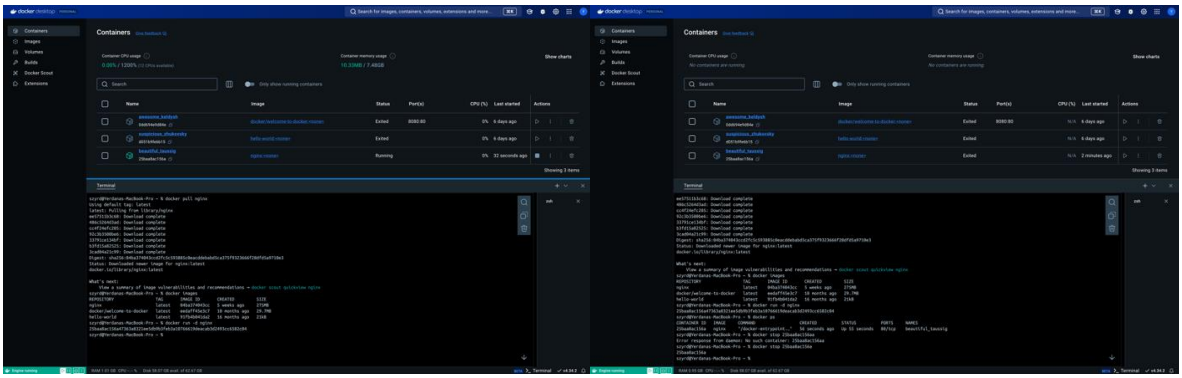
What It Signifies:

- *Successful Installation: The message confirms that Docker is installed and running correctly on your system.*
- *Image and Container Test: Docker successfully pulled the hello-world image from Docker Hub, created a container, and ran the test executable inside it.*
- *Client-Daemon Communication: It demonstrates that the Docker client can communicate with the Docker daemon without issues.*

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`?

docker pull downloads an image from Docker Hub to your local system.

docker run does everything docker pull does but also creates and starts a new container from the downloaded image.

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

Use the command docker ps. It lists container IDs, statuses, names, and more.

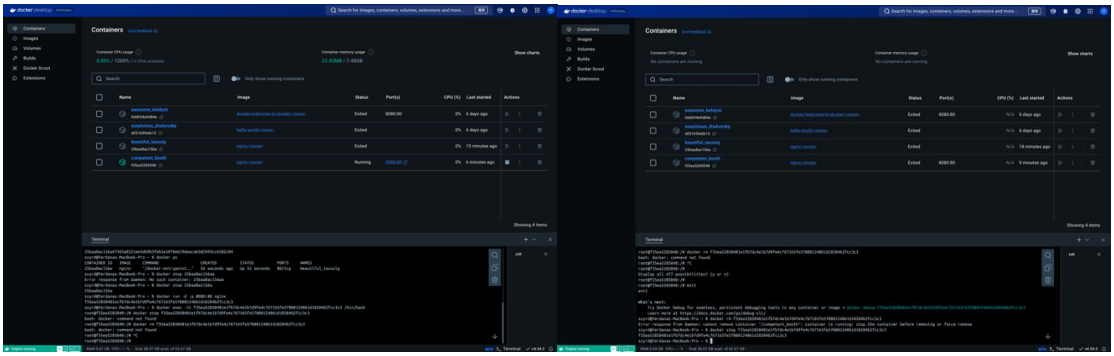
    - What happens to a container after it is stopped? Can it be restarted?

After a container is stopped, it retains its state and data but is no longer running. You can restart it using 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>`.

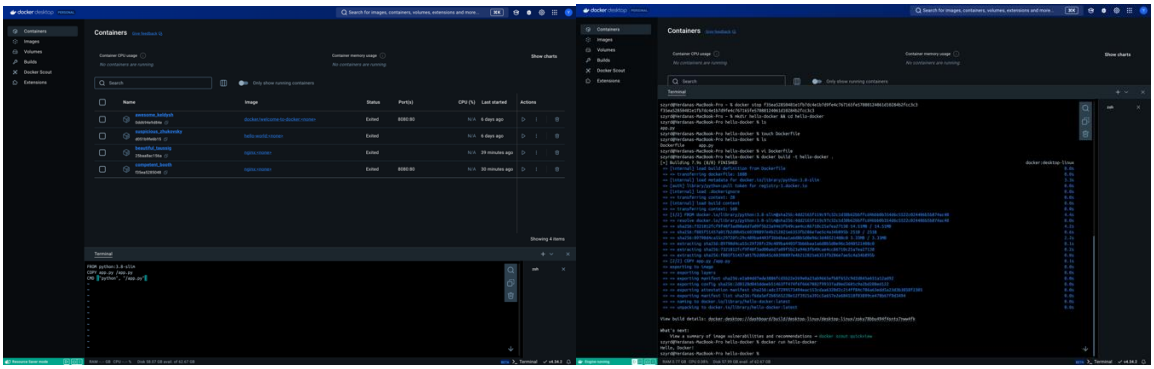


3. **Questions**:
   - How does port mapping work in Docker, and why is it important?
     Port mapping exposes a container's port to the host machine, allowing external access. It's essential for running web services accessible outside the container.
   - What is the purpose of the `docker exec` command?
     It allows you to execute commands inside a running container, which is useful for debugging or interacting with the container's environment.
   - How do you ensure that a stopped container does not consume system resources? **Remove it using docker stop <container-id>. Stopped containers don't consume CPU or memory, but they do occupy disk space.**

Dockerfile

Exercise 1: Creating a Simple Dockerfile

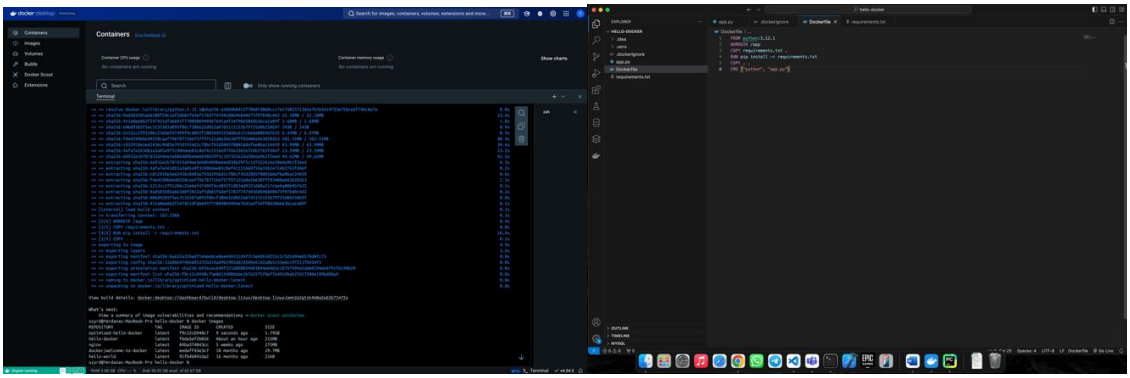
1. **Objective**: Write a Dockerfile to containerize a basic application.
2. **Steps**:
   - Create a new directory for your project and navigate into it.
   - Create a simple Python script (e.g., `app.py`) that prints "Hello, Docker!" to the console.
   - Write a Dockerfile that:
     - Uses the official 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`.

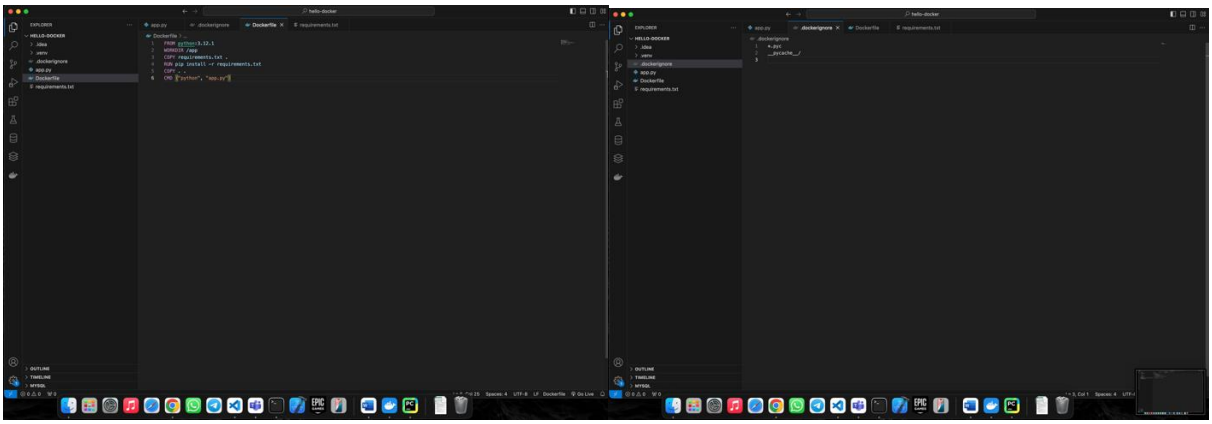


3. **Questions**:
    - What is the purpose of the `FROM` instruction in a Dockerfile?
      It specifies the base image to use for creating the Docker image.
    - How does the `COPY` instruction work in Dockerfile?
      It copies files from the host machine into the Docker image at a specified path.
    - What is the difference between `CMD` and `ENTRYPOINT` in Dockerfile?
      `CMD` specifies default commands or arguments for an image, but they can be overridden at runtime. `ENTRYPOINT` sets the command that will always run and usually works with `CMD` to provide 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.

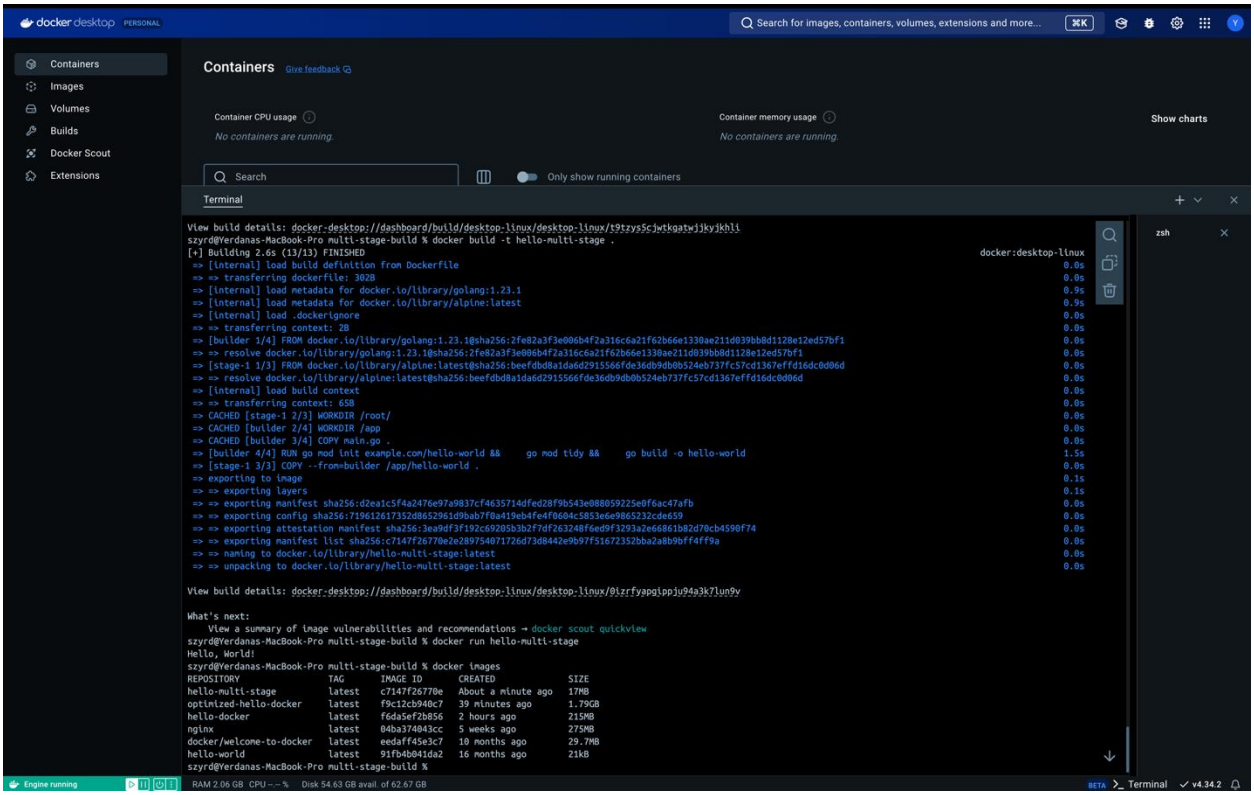




3. **Questions**:
    - What are Docker layers, and how do they affect image size and build times? **Each line in a Dockerfile creates a layer in the Docker image. Layers are cached, so rebuilding the image will skip unchanged layers, reducing build time.**
    - How does Docker's build cache work, and how can it speed up the build process?
      Docker caches layers during the build process. If a layer doesn't change, Docker will use the cached version, speeding up the build.
    - What is the role of the `.dockerignore` file? **It excludes specified files and directories from the build context, preventing unnecessary files from being included in the image.**

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.

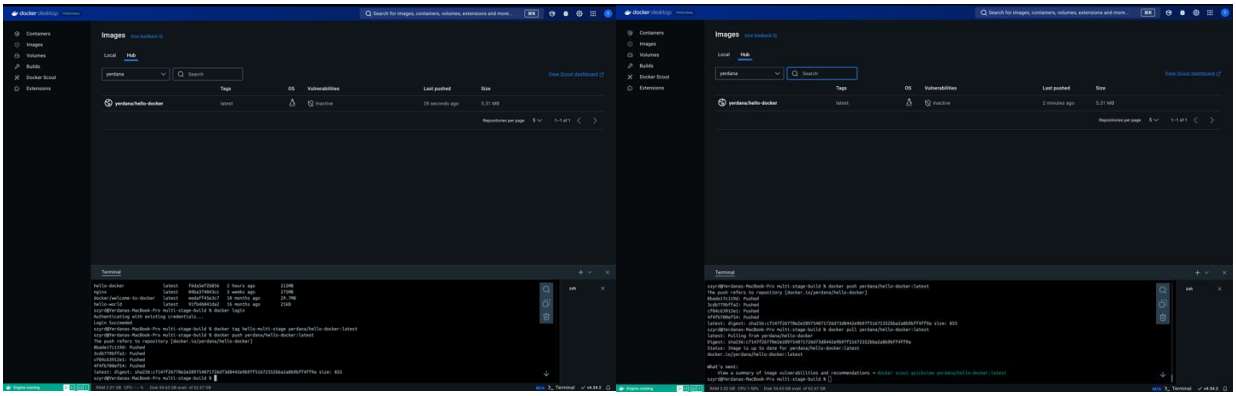


3. **Questions**:
    - What are the benefits of using multi-stage builds in Docker?
      Multi-stage builds allow you to separate the build and runtime environments, resulting in smaller, more secure images. The final image only contains the necessary artifacts, without the build dependencies.
    - How can multi-stage builds help reduce the size of Docker images?
      By using a minimal base image in the final stage and copying only the required artifacts (like binaries), multi-stage builds eliminate unnecessary files and dependencies from the image.
    - What are some scenarios where multi-stage builds are particularly useful?
      Multi-stage builds are beneficial when building applications that require a lot of build-time dependencies, such as compiled languages (Go, Java, etc.), complex front-end applications, or when needing to include only specific build artifacts 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.



3. **Questions**:
   - What is the purpose of Docker Hub in containerization?
     Docker Hub is a cloud-based registry service for building, storing, and distributing Docker images. It allows developers to share their images publicly or privately with others.
   - How do you tag a Docker image for pushing to a remote repository?
     I'm tag a Docker image using the docker tag command. I need to include the repository name and optionally the tag (e.g., docker tag local-image yerdana/repository:tag).
   - What steps are involved in pushing an image to Docker Hub?

     1. **Log in to Docker Hub using docker login.**
     2. **Tag the image with your Docker Hub username and repository name using docker tag.**
     3. **Push the tagged image using docker push.**