**Assignment 1, Web Application Development**

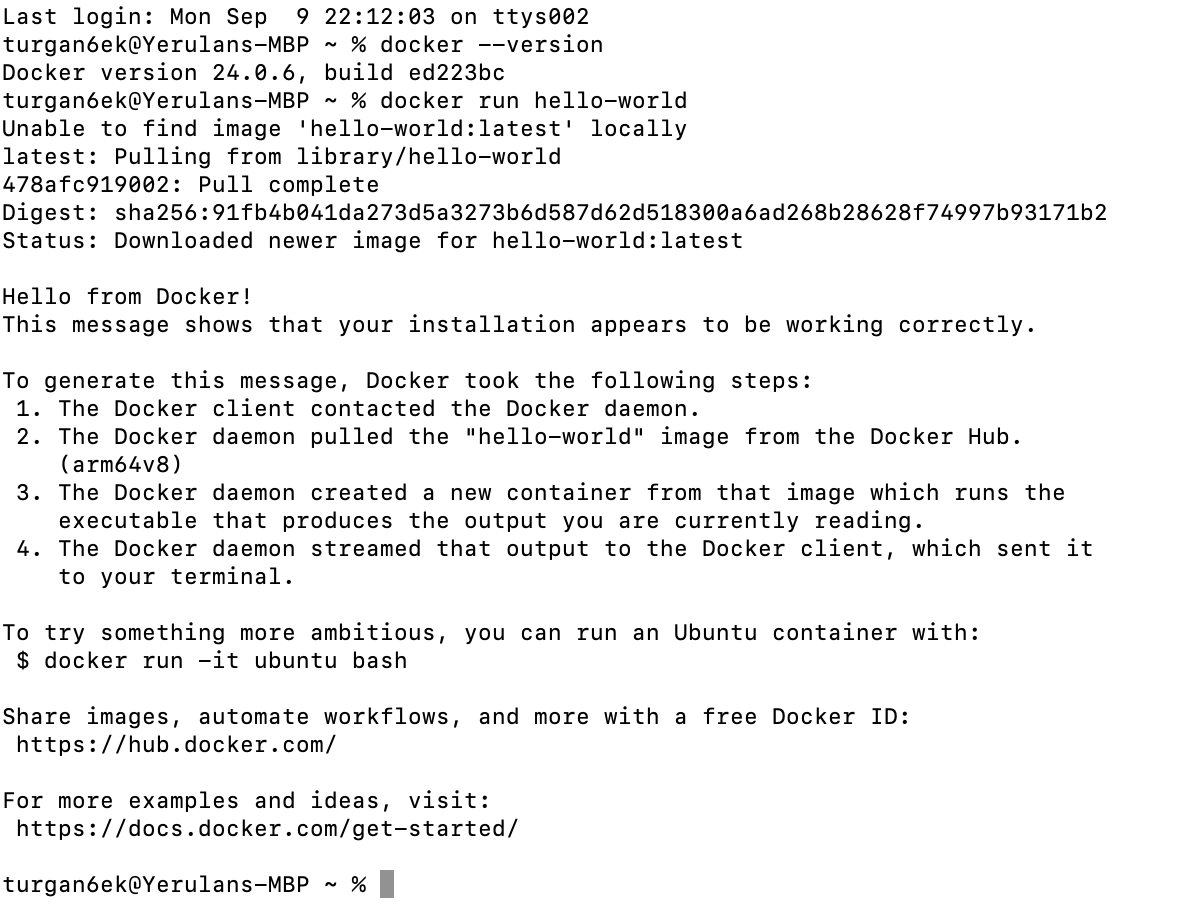
**Turganbek Yerulan 23MD0446**

### **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)?
4. Docker Engine
5. Docker CLI
6. Docker Images
7. Docker Containers
8. Dockerfile
9. Docker Hub
10. Docker Compose
11. Docker Volumes
12. Docker Networking
    * How does Docker compare to traditional virtual machines?

Docker is more lightweight and faster than traditional virtual machines (VMs) because it shares the host OS, while VMs run full guest OS instances, making them resource-heavy and slower. Docker containers are smaller, more portable, and start quickly, whereas VMs offer stronger isolation but are larger and harder to move. Docker is ideal for efficient resource use, while VMs are better for scenarios requiring full OS isolation.

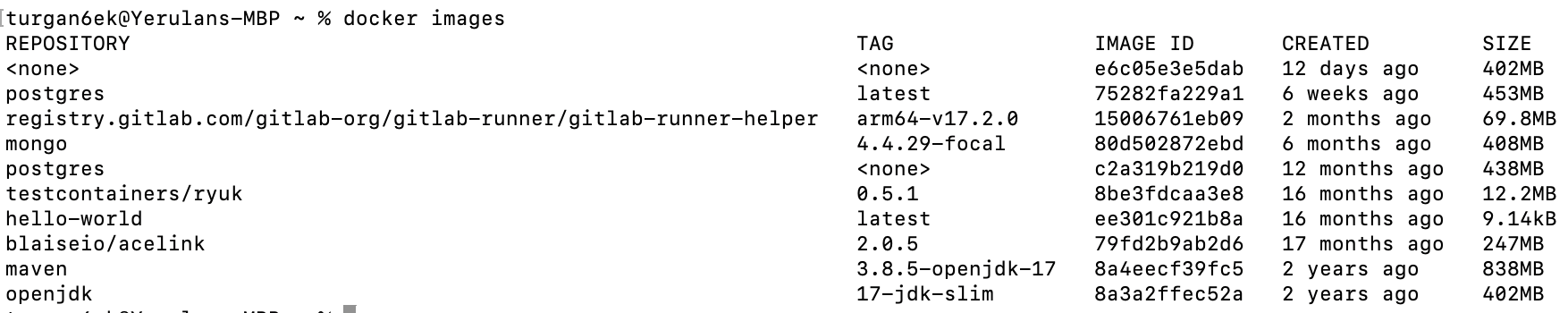
* + What was the output of the docker run hello-world command, and what does it signify?  
    

#### **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?
4. Docker pull: only downloads the image
5. If the image is not downloaded yet, downloads it and creates the container from downloaded image.
   * How do you find the details of a running container, such as its ID and status?

“docker ps” command lists all the info about running containers, using tag -a can be used to list including stopped containers.

* + What happens to a container after it is stopped? Can it be restarted?  
    After a Docker container is stopped, it remains in a stopped state and is not removed unless explicitly deleted. The container’s data, settings, and state are still available, meaning you can inspect or restart it later.





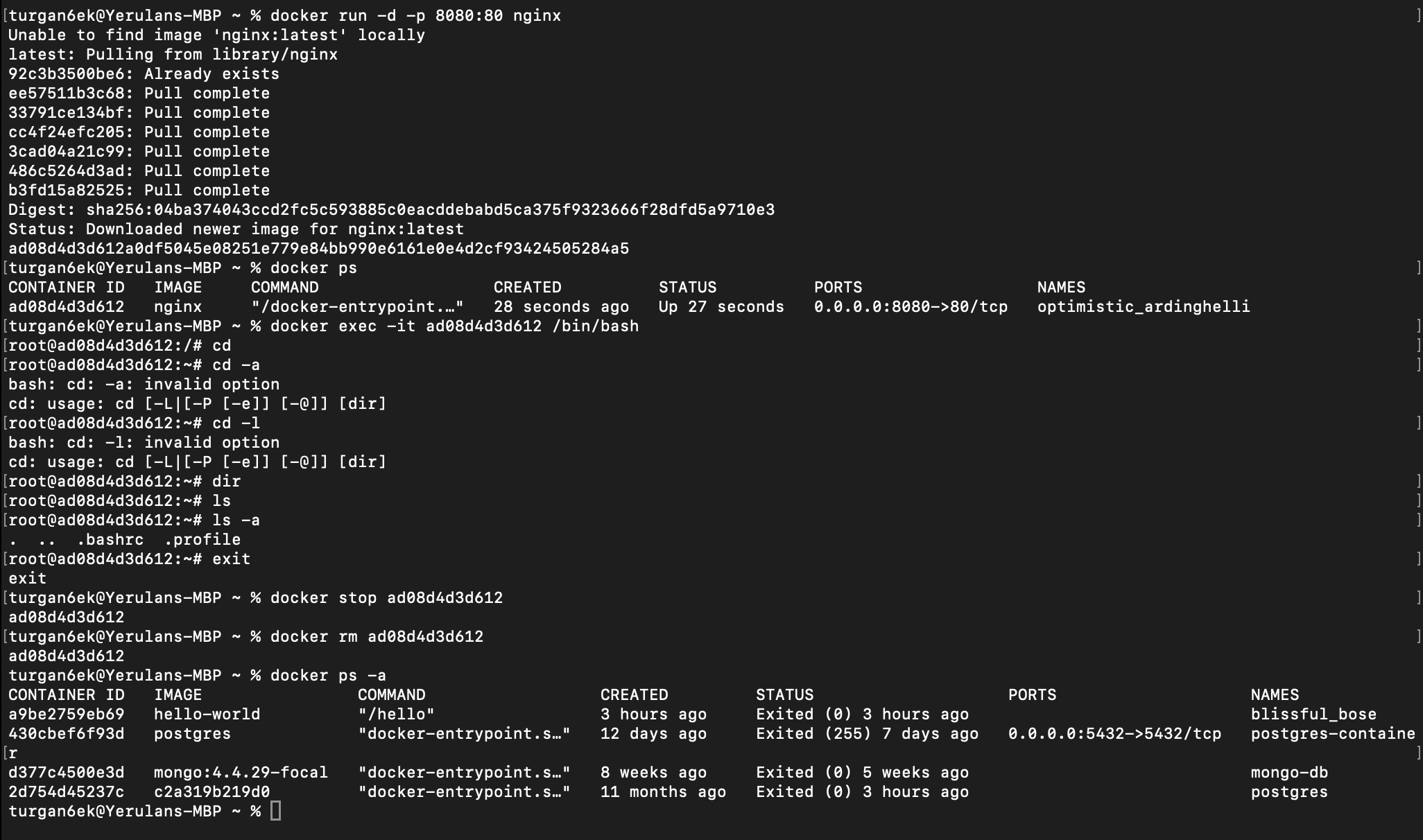
#### **A close-up of a white background Description automatically generated**

#### **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?  
     using -p option, M:N where M is the port of the host and 80 is port in container.
   * What is the purpose of the docker exec command?

The docker exec command is used to run a command in a running Docker container.

* + How do you ensure that a stopped container does not consume system resources?  
    After stopping and removing container, check using docker ps -a command



### **Dockerfile**

#### **Exercise 1: Creating a Simple Dockerfile**

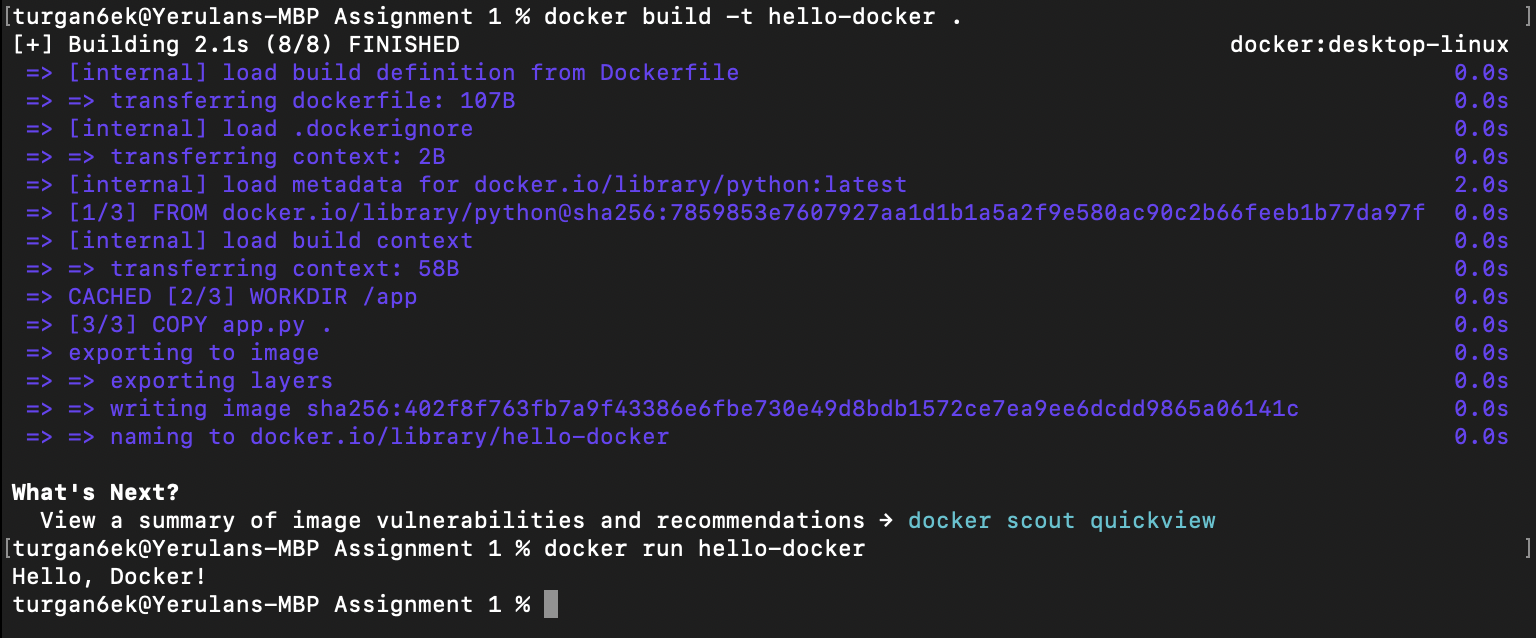
1. **Objective**: Write a Dockerfile to containerize a basic application.
2. **Steps**:
   * Create a new directory for your project and navigate into it.
   * Create a simple Python script (e.g., app.py) that prints "Hello, Docker!" to the console.
   * Write a Dockerfile that:
     + Uses the official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

The FROM instruction in a Dockerfile specifies the base image for the subsequent instructions. It defines the starting point for the build process. Each Dockerfile must start with a FROM instruction, which can be an official image (like python, ubuntu, etc.) or a custom image. The specified image provides the operating system and any necessary dependencies required for your application.

* + How does the COPY instruction work in Dockerfile?

The COPY instruction in a Dockerfile is used to copy files and directories from your host machine into the Docker image during the build process.

* + What is the difference between CMD and ENTRYPOINT in Dockerfile?  
    `CMD` specifies default commands or arguments that can be overridden when running a container, while `ENTRYPOINT` defines the main command that always executes when the container starts and is not easily overridden. In essence, use `ENTRYPOINT` for the core functionality of the container and `CMD` for providing default options or commands that can be customized.



#### **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?

Docker layers are the building blocks of Docker images. Each instruction in a Dockerfile (like FROM, COPY, or RUN) creates a new layer in the image. Layers are stacked on top of each other to form the final image. Since layers are immutable, if the content of a layer changes, Docker only needs to rebuild that layer and the layers above it, rather than the entire image. This helps optimize image size and build times by avoiding redundant work.

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

Docker's build cache stores the results of each layer created during the build process. When you rebuild an image, Docker checks if any of the previous layers can be reused based on the instruction and the context. If nothing has changed in a layer (e.g., the command and its inputs are the same), Docker uses the cached version instead of executing the instruction again. This can significantly speed up the build process, especially for large images with many dependencies.

* + What is the role of the .dockerignore file?  
    The .dockerignore file specifies files and directories to exclude from the build context when creating a Docker image. This helps reduce the size of the context sent to the Docker daemon and minimizes the final image size by excluding unnecessary files (like logs, temporary files, or source code not needed for the image). By preventing irrelevant files from being included, it can also speed up the build process and improve security by not exposing sensitive files in the image.

Before: 1.02GB  


In our case we don’t have any dependencies so using .dockerignore won’t help. I used smaller image of python to optimize space.

After: 151MB  


#### **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 in Docker allow you to separate the build environment from the runtime environment, enabling cleaner and more efficient images. They help reduce complexity by allowing you to use different base images for building and running your application. This leads to improved organization of Dockerfiles and a clearer separation of concerns.
   * How can multi-stage builds help reduce the size of Docker images?  
     Multi-stage builds can significantly reduce the size of Docker images by allowing you to include only the necessary binaries and dependencies in the final image. In the first stage, you can compile your application and install all the necessary tools, while the final stage can be based on a minimal image (like Alpine) that only contains the compiled application. This approach avoids bundling unnecessary files, libraries, and tools that were needed during the build process.
   * What are some scenarios where multi-stage builds are particularly useful?  
     Multi-stage builds are particularly useful in scenarios involving compiled languages like Go or C++, where the build process requires a full development environment to compile the application, allowing only the final executable to be included in a minimal runtime image. They're beneficial for microservices architectures, enabling each service to be packaged with its specific dependencies, and for Node.js applications, where you can separate development and production dependencies. Additionally, multi-stage builds facilitate asset compilation for front-end frameworks like React or Angular, where assets are built in one stage and served from a lightweight server in the final image. Overall, they help streamline complex build processes while producing smaller, more secure Docker images tailored for production.

For the task above, single-stage takes 876MB while multi-stage takes only 11MB.



#### **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 storing and sharing Docker images. It serves as a central repository where developers can publish their images, making it easy to share them with others or deploy them across different environments. Docker Hub also provides features like automated builds, webhooks, and access control, facilitating collaboration and streamlining the workflow for containerized applications.

* + How do you tag a Docker image for pushing to a remote repository?  
    docker tag hello-docker <your-username>/hello-docker
  + What steps are involved in pushing an image to Docker Hub?

docker tag  
docker login  
docker push

