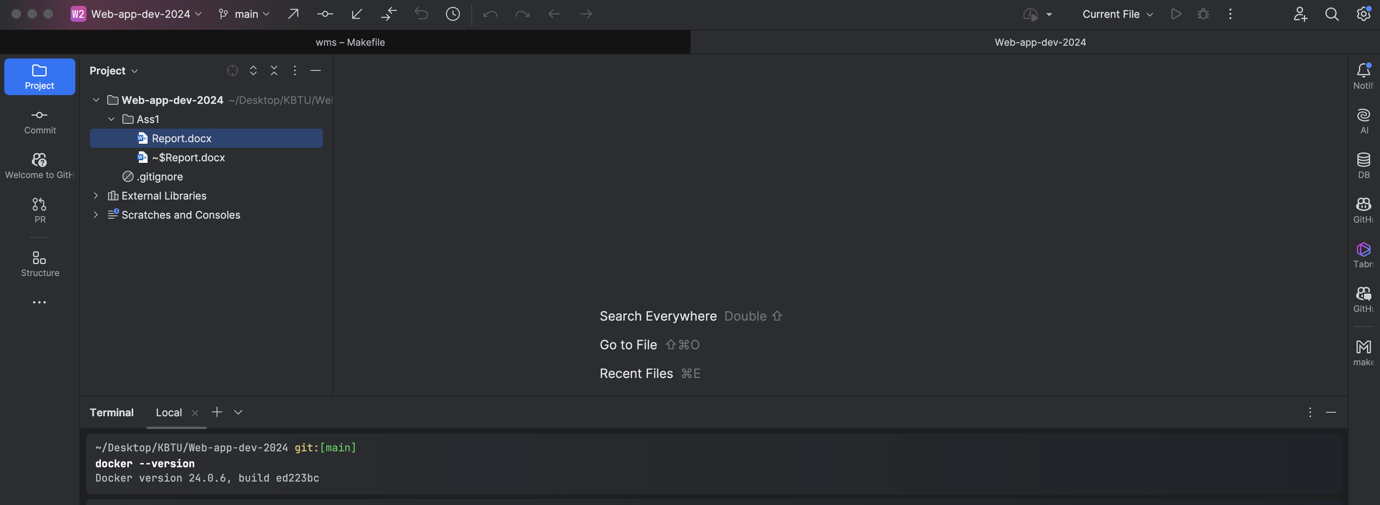
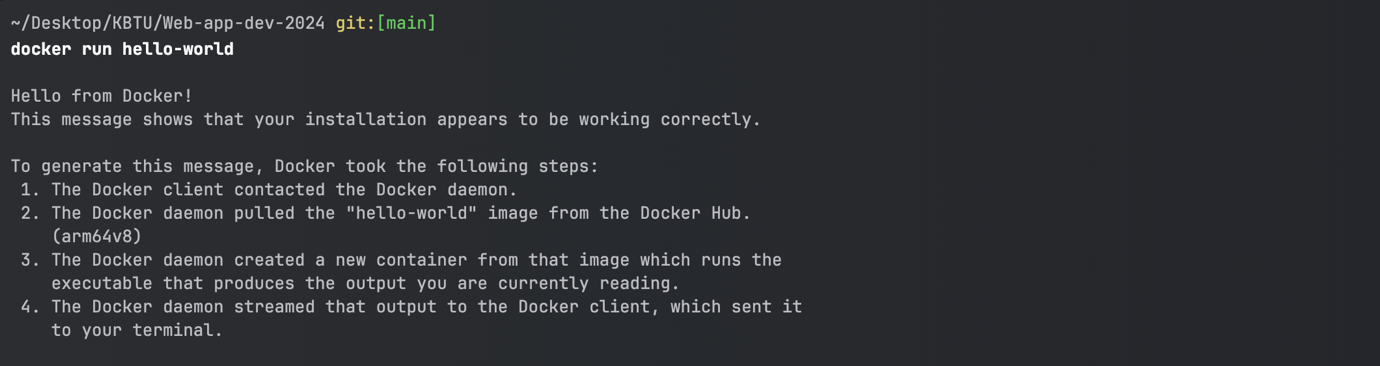
**Assignment 1, Web Application Development**

**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)?

• **Docker Engine:** The core part of Docker that manages containers. It includes:

• **Docker Daemon**: Runs in the background, managing Docker objects like images, containers, networks, and volumes.

• **REST API**: Allows communication between Docker Daemon and Docker CLI or other services.

• **Docker CLI**: The command-line interface used to interact with Docker. It communicates with the Docker Daemon to execute tasks like running containers, pulling images, etc.

• **Docker Hub**: A public registry where Docker images are stored and can be pulled by users to run containers.

• **Docker Compose:** A tool to define and run multi-container Docker applications using a docker-compose.yml file.

* + How does Docker compare to traditional virtual machines?

**Docker**:

• Uses containers that share the host OS kernel, making them lightweight and faster to start.

• Containers are isolated but run on the same OS, allowing efficient resource utilization.

• Ideal for deploying applications consistently across different environments.

• Lower overhead because there’s no need for a full OS per container.

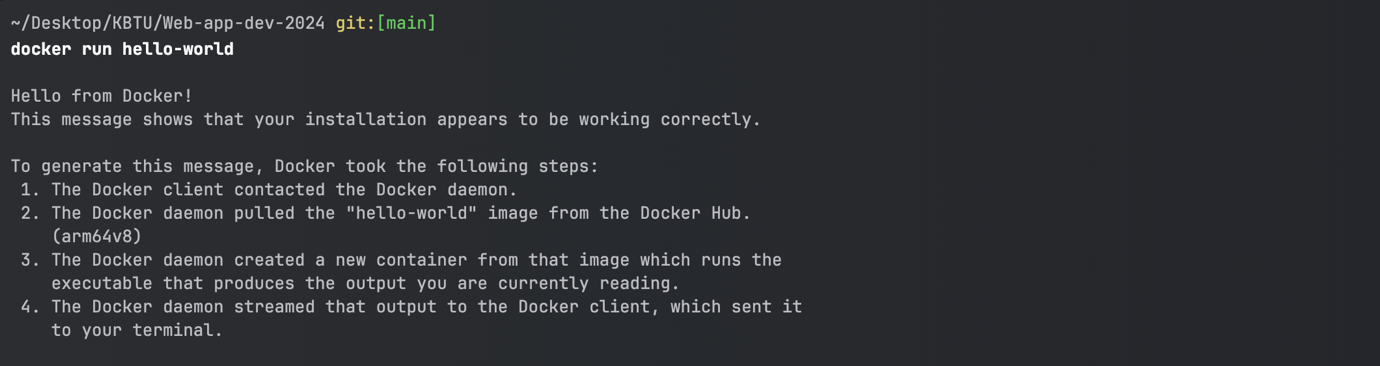
**Traditional Virtual Machines:**

• Each VM runs its own full operating system on top of the host system via a hypervisor.

• VMs are more isolated from each other than containers but require more resources.

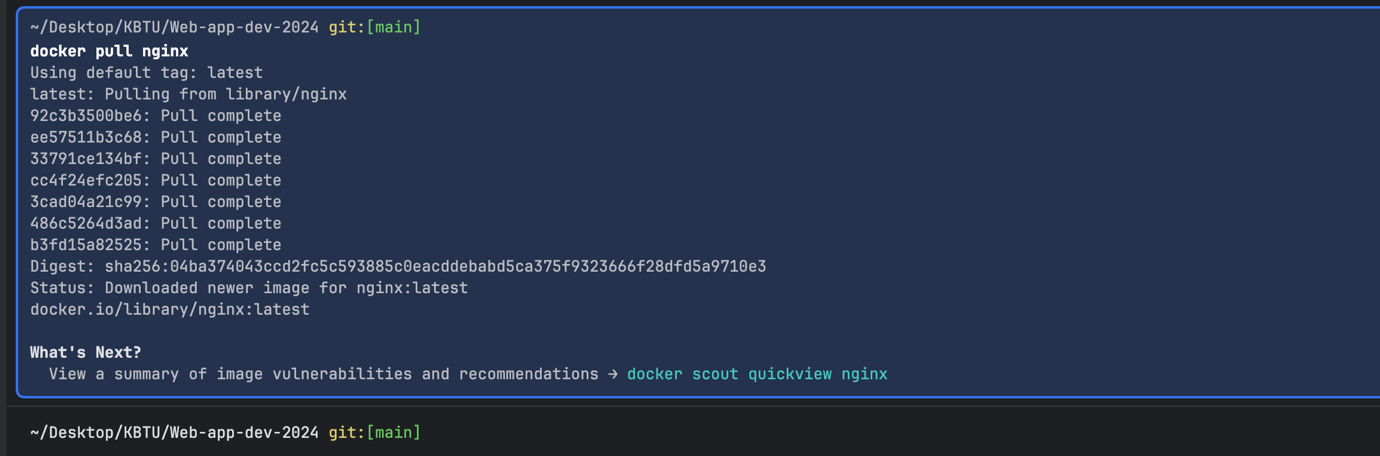
• VMs are slower to start due to the need to boot up an entire OS for each machine.

• Better suited for running multiple different OS environments on a single physical machine.

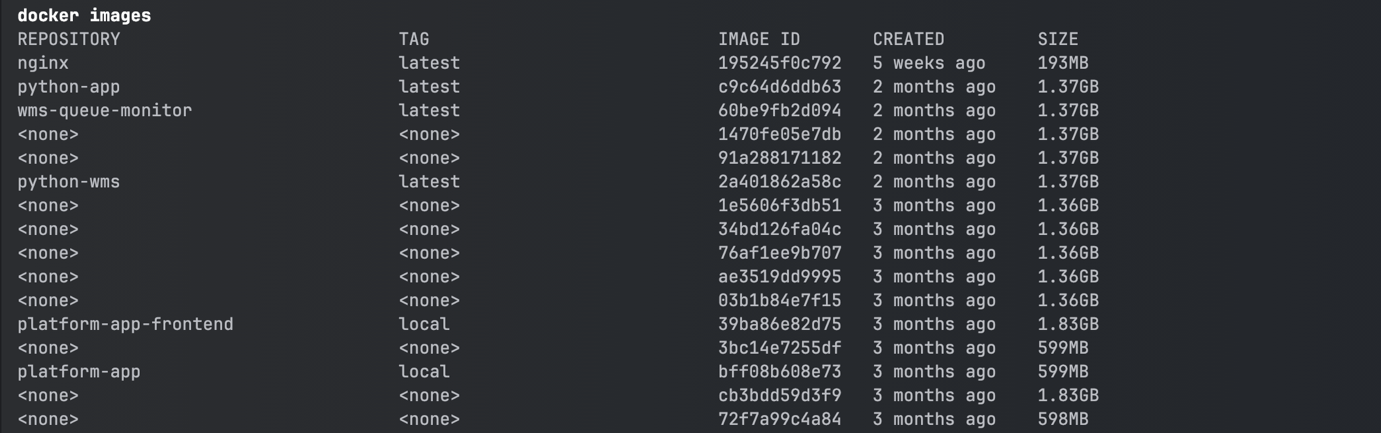
* + What was the output of the docker run hello-world command, and what does it signify?
  + It shows that intsallation was OK!
  + 

**Exercise 2: Basic Docker Commands**

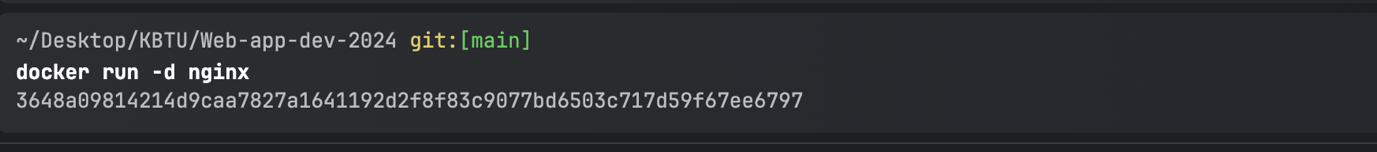
1. **Objective**: Familiarize yourself with basic Docker commands.
2. **Steps**:
   * Pull an official Docker image from Docker Hub (e.g., nginx or ubuntu) using the command docker pull <image-name>.

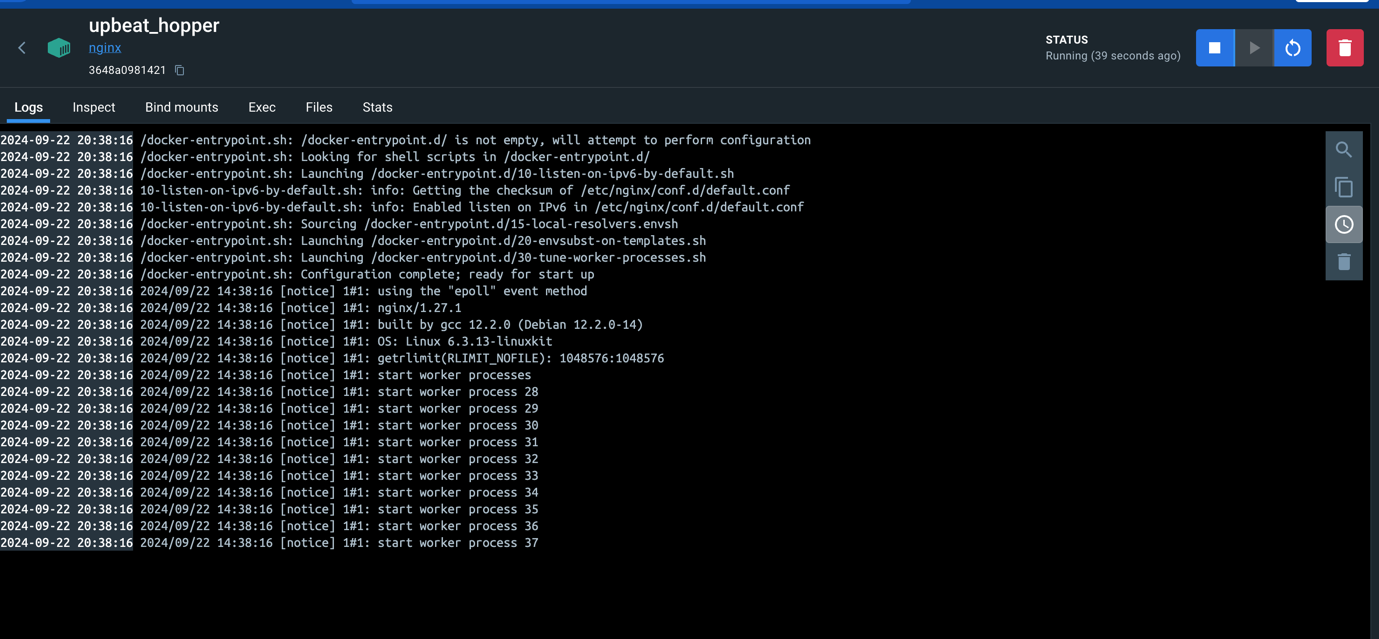
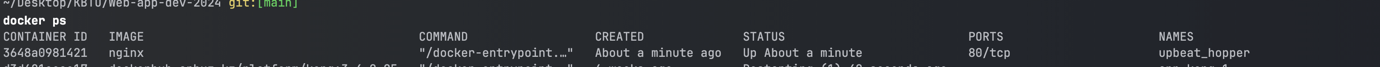
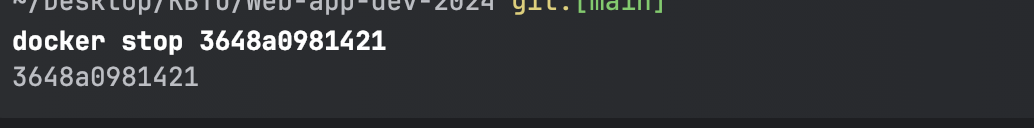


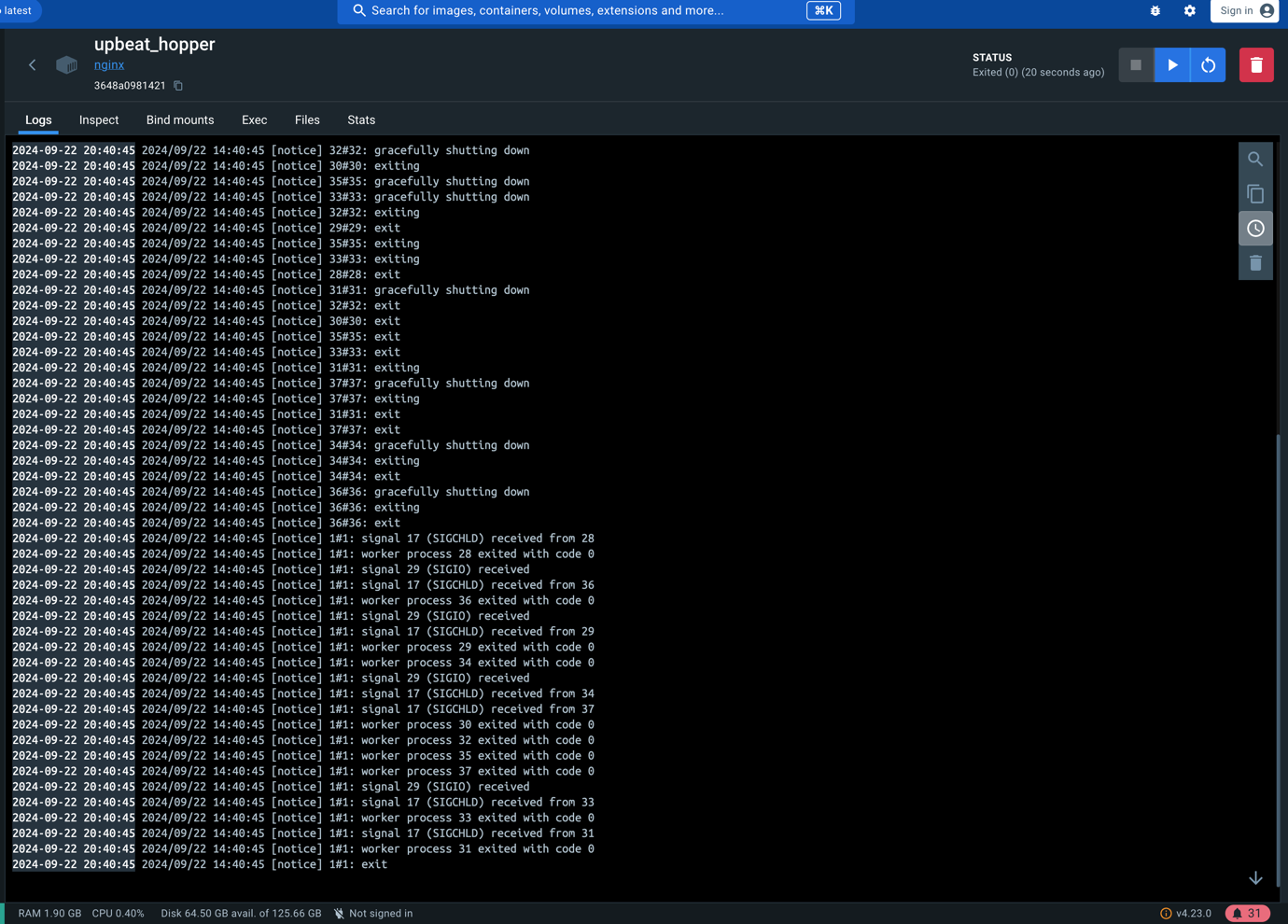
* + List all Docker images on your system using docker images.



* + 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>.
  + 



**Questions**:

* + What is the difference between docker pull and docker run?

**docker pull <image>:**

• This command downloads a Docker image from a registry (like Docker Hub) to your local system.

• It does not create or start a container. It only ensures the image is available locally for use.

**docker run <image>:**

• This command pulls the image (if it’s not already available locally) and then creates and starts a container from the image.

• It combines both downloading the image (if necessary) and running the container in one step.

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

**docker ps**: Lists all running containers, displaying details like Container ID, image name, creation time, status, and port mappings.

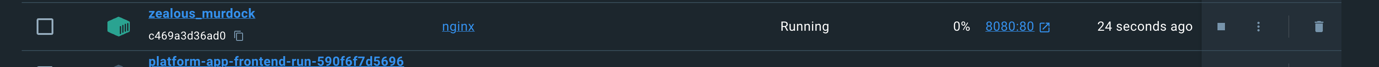
* + What happens to a container after it is stopped? Can it be restarted?

• When a container is stopped (via docker stop <container\_name>), the container’s process halts, but the container still exists in a stopped state.

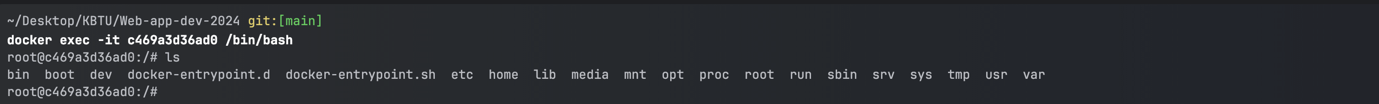
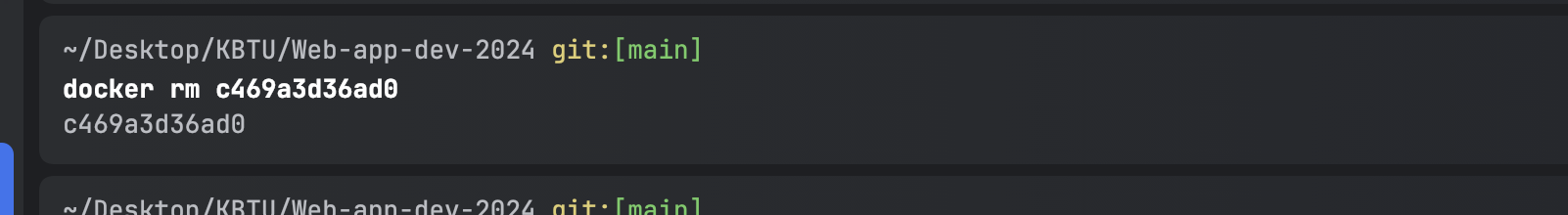
• Yes, it can be restarted using the docker start <container\_name> command. The stopped container retains all of its configurations, and when restarted, it will resume its previous state.

• You can also use docker restart <container\_name> to stop and immediately start the container in one command.

**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>.
  + 

1. **Questions**:
   * How does port mapping work in Docker, and why is it important?

• Port mapping in Docker allows you to access a service running inside a Docker container from the host machine. When a container runs, it is isolated and only accessible from within Docker’s internal network. Port mapping makes it accessible from the outside.

• The syntax is: docker run -p <host\_port>:<container\_port> <image>

• Host port: The port on your machine (or host OS) that will be exposed.

• Container port: The port inside the container that the service listens on.

• For example, running docker run -p 8080:80 nginx will map port 80 (inside the container) to 8080 (on the host machine). This means you can access the web server running inside the container by visiting http://localhost:8080.

• Importance:

• Port mapping is crucial because it allows containers to communicate with the outside world (e.g., other services, applications, or users) and makes it possible for users to access services running in a container from a browser or API client.

• Without port mapping, services running in containers would be isolated and inaccessible from the host system.

* + What is the purpose of the docker exec command?

• docker exec allows you to run additional commands inside a running container. This is useful for debugging or interacting with a container while it’s already running.

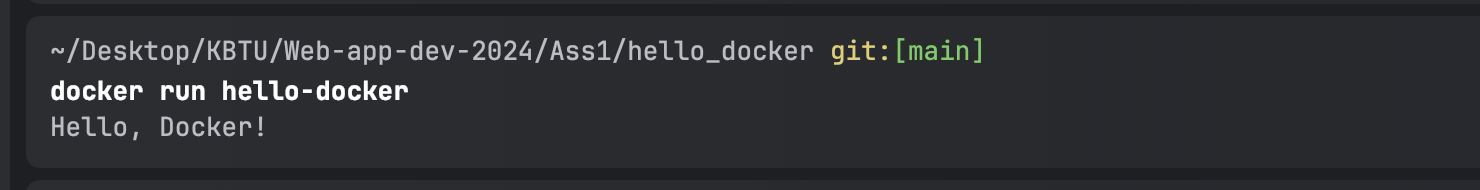
* + How do you ensure that a stopped container does not consume system resources?

Once a container is stopped (using docker stop <container\_name>), it does not consume CPU or memory resources anymore. However, it may still occupy disk space for its configuration, logs, or state.

• To completely remove a stopped container and ensure it doesn’t consume any system resources (including disk space), you should use docker rm <container\_name> to delete the container.

**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 specifies the base image to be used for your Docker imag

* + 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 (the directory where the Dockerfile is located) into the Docker container.

• Syntax: COPY <source> <destination>

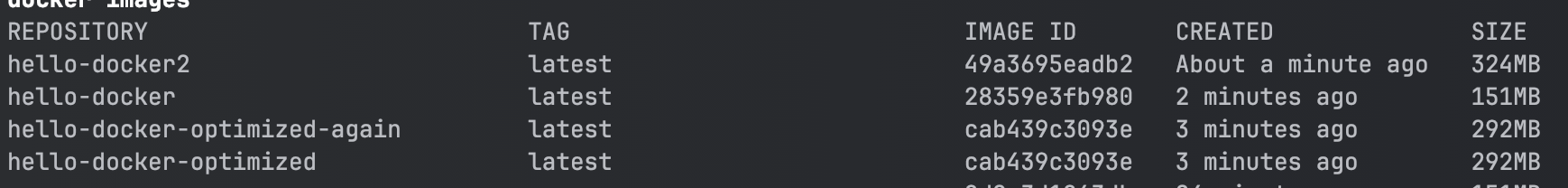
• <source>: The file or directory on your host machine that you want to copy.

• <destination>: The path inside the container where the file/directory should be copied.

* + What is the difference between CMD and ENTRYPOINT in Dockerfile?

CMD provides default commands that can be overridden, while ENTRYPOINT defines a fixed command that will always run. If flexibility is needed, CMD is generally preferred; if the container is designed to run one specific command, ENTRYPOINT is more appropriate.

**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 a Docker image. Each command in the Dockerfile creates a new layer in the image.

• Layers are stacked on top of each other, where each layer contains the changes made by that specific command (e.g., copying files, installing software).

• Docker reuses layers whenever possible to reduce both build times and image size. For example, if one layer doesn’t change, Docker can skip rebuilding it and use the cached version.

• Effect on image size: The more layers there are and the more redundant data stored in them (e.g., unnecessary files), the larger the image size.

• Effect on build times: By caching and reusing layers, Docker can speed up subsequent builds, especially when only parts of the Dockerfile change (e.g., code but not dependencies).

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

• Docker’s build cache stores previously built layers, allowing Docker to reuse them when possible. If a layer’s content hasn’t changed between builds, Docker will reuse the cached layer instead of rebuilding it.

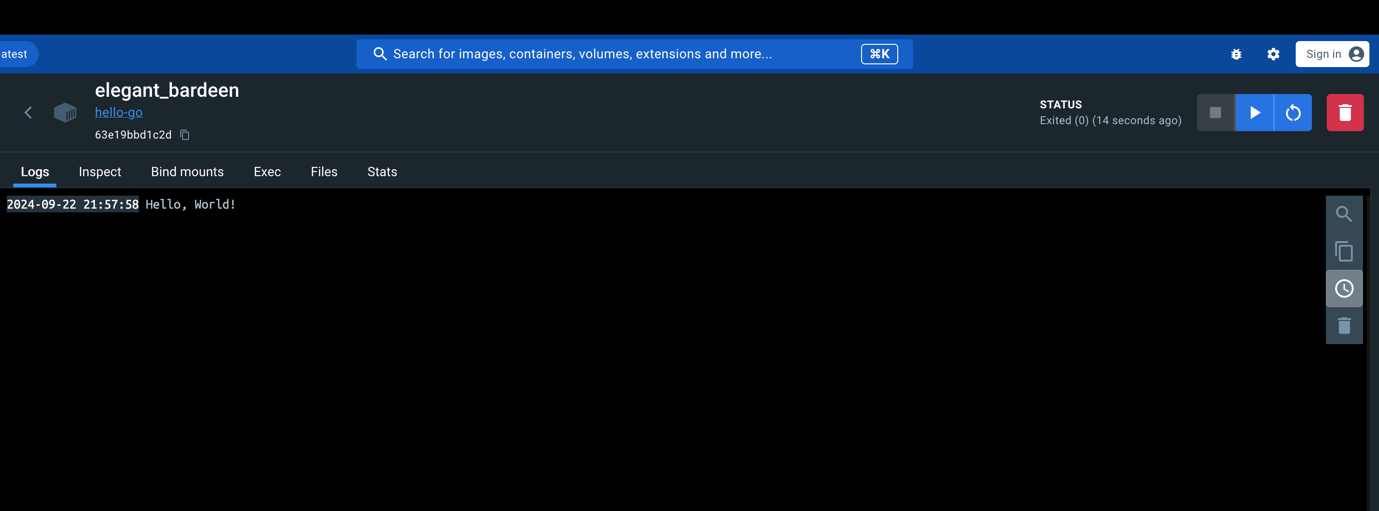
• Speeding up the build process:

• Docker starts building from the top of the Dockerfile, using the cache for each layer if the command hasn’t changed and the source files are the same. This way, only the new or changed parts of the image need to be built, significantly reducing build times.

• For example, if your Python dependencies (requirements.txt) haven’t changed, Docker will reuse the cached layer where those dependencies were installed, avoiding the need to reinstall them during every build.

* + What is the role of the .dockerignore file?
  + • The .dockerignore file tells Docker which files and directories should be excluded from the image-building process.
  + • This is useful because you can prevent unnecessary files (e.g., configuration files, logs, .git directories, .pyc files) from being copied into the Docker image. This reduces the size of the image and avoids potential security issues from including sensitive files.
  + • By excluding unnecessary files, the .dockerignore file also helps ensure that builds are faster because Docker processes fewer files when building 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?

• Smaller Final Images: Multi-stage builds allow you to separate the build environment from the runtime environment, resulting in smaller final images that only contain the necessary binaries and dependencies.

• Improved Build Efficiency: By organizing the build process, multi-stage builds can make it easier to manage dependencies and only include what’s essential for the final application.

• Cleaner Dockerfiles: They help in maintaining a cleaner and more readable Dockerfile by logically separating build and run stages.

• Faster Deployments: Smaller images can lead to faster deployments and quicker transfers when pushing or pulling images.

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

• In a multi-stage build, you can use a heavier base image (like golang for compiling) for the build process, but only the compiled binaries are transferred to a minimal base image (like alpine) in the final stage. This reduces the amount of unused files, libraries, and build tools in the final image, leading to a much smaller image size.

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

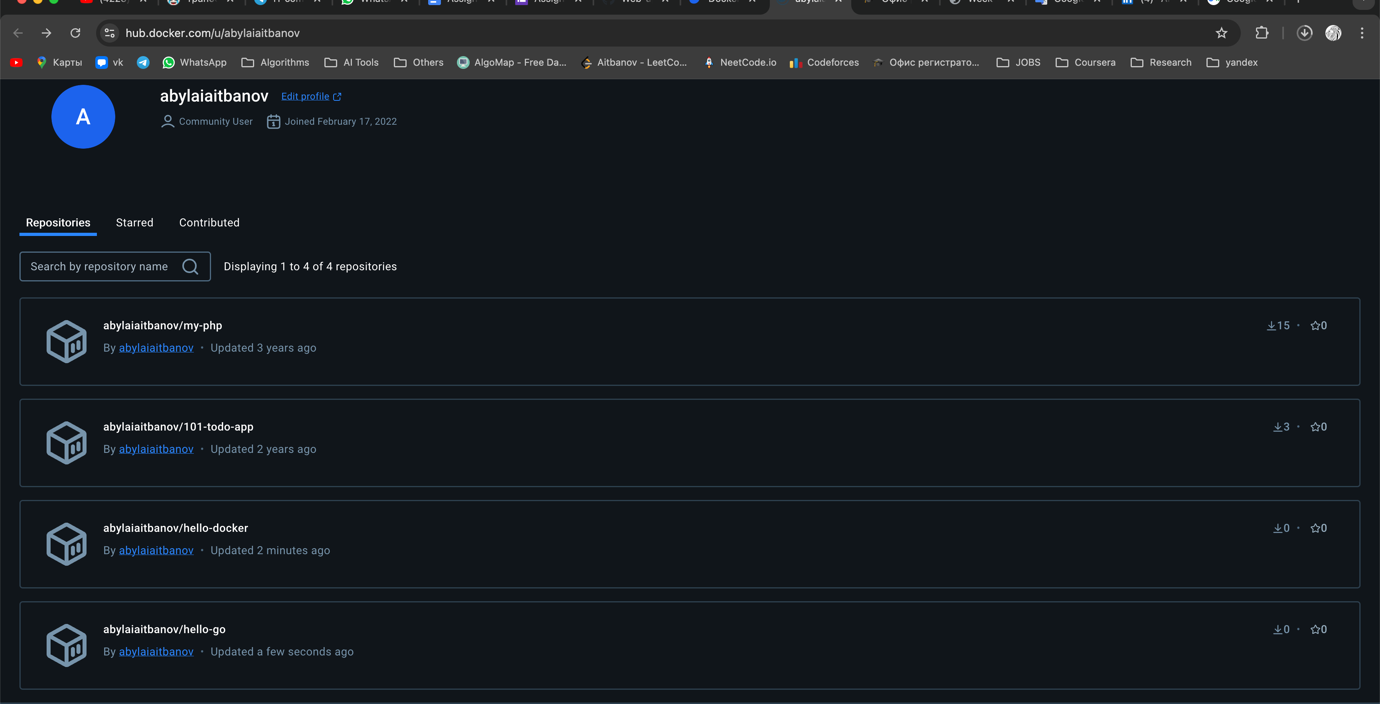
• Compiling Applications: When using languages that require compilation (e.g., Go, C, C++), you can compile in one stage and run in a minimal image in another.

• Building Frontend Assets: If you have a frontend application that needs to be built (e.g., with Webpack or similar tools), you can build the assets in a Node.js image and then copy the built files to a minimal server image (like Nginx).

• Testing and Deployment: You can run tests in a more complete environment and only package the necessary artifacts in the final image for deployment.

• Reducing Dependencies: If your application has many dependencies, multi-stage builds allow you to keep only what’s necessary for running the application, removing unnecessary libraries or files used only during the build phase.

**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 that allows users to share, store, and distribute Docker images. It acts as a central repository where developers can find pre-built images and publish their own, facilitating collaboration and deployment across different environments.

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

docker tag <existing-image> <repository>/<image-name>:<tag>

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

• Create an Account: Sign up for a Docker Hub account.

• Tag the Image: Use the docker tag command to tag your image with your Docker Hub username.

• Log In: Use docker login to authenticate to Docker Hub.

• Push the Image: Use the docker push command to upload the tagged image to your Docker Hub repository.