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)?
        1. Docker Engine
        2. Docker CLI
        3. Docker Images
        4. Docker Containers
        5. Dockerfile
        6. Docker Hub
        7. Docker Compose
        8. Docker Volumes
        9. 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?

```
Last login: Mon Sep  9 22:12:03 on ttys002
turgan6ek@Yerulans-MBP ~ % docker --version
Docker version 24.0.6, build ed223bc
turgan6ek@Yerulans-MBP ~ % docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
478afc919002: Pull complete
Digest: sha256:91fb4b041da273d5a3273b6d587d62d518300a6ad268b28628f74997b93171b2
Status: Downloaded newer image for hello-world:latest

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.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

turgan6ek@Yerulans-MBP ~ %
```

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`?
        1. Docker pull: only downloads the image
        2. 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.

```
turgan6ek@Yerulans-MBP ~ % docker images
REPOSITORY                                                    TAG               IMAGE ID       CREATED         SIZE
<none>                                                        <none>            e6c05e3e5dab   12 days ago     402MB
postgres                                                      latest            75282fa229a1   6 weeks ago     453MB
registry.gitlab.com/gitlab-org/gitlab-runner/gitlab-runner-helper   arm64-v17.2.0     15006761eb09   2 months ago    69.8MB
mongo                                                         4.4.29-focal      80d502872ebd   6 months ago    408MB
postgres                                                      <none>            c2a319b219d0   12 months ago   438MB
testcontainers/ryuk                                           0.5.1             8be3fdcaa3e8   16 months ago   12.2MB
hello-world                                                   latest            ee301c921b8a   16 months ago   9.14kB
blaiseio/acelink                                              2.0.5             79fd2b9ab2d6   17 months ago   247MB
maven                                                         3.8.5-openjdk-17  8a4eecf39fc5   2 years ago     838MB
openjdk                                                       17-jdk-slim       8a3a2ffec52a   2 years ago     402MB
```

```
turgan6ek@Yerulans-MBP ~ % docker run -d hello-world
ff743ac3195b92532d7f4f0cf02c70ccf7753dcbf3db9bfd76b973c3bf8af129
```

```
turgan6ek@Yerulans-MBP ~ % docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS                    NAMES
2d754d45237c   c2a319b219d0   "docker-entrypoint.s…"   11 months ago   Up 5 seconds   0.0.0.0:5432->5432/tcp   postgres
turgan6ek@Yerulans-MBP ~ % docker stop 2d754d45237c
2d754d45237c
turgan6ek@Yerulans-MBP ~ % docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
turgan6ek@Yerulans-MBP ~ %
```

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

```
turgan6ek@Yerulans-MBP ~ % docker run -d -p 8080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
92c3b3500be6: Already exists
ee57511b3c68: Pull complete
33791ce134bf: Pull complete
cc4f24efc205: Pull complete
3cad04a21c99: Pull complete
486c5264d3ad: Pull complete
b3fd15a82525: Pull complete
Digest: sha256:04ba374043ccd2fc5c593885c0eacddebabd5ca375f9323666f28dfd5a9710e3
Status: Downloaded newer image for nginx:latest
ad08d4d3d612a0df5045e08251e779e84bb990e6161e0e4d2cf93424505284a5
turgan6ek@Yerulans-MBP ~ % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS                   NAMES
ad08d4d3d612   nginx     "/docker-entrypoint.…"   28 seconds ago   Up 27 seconds   0.0.0.0:8080->80/tcp   optimistic_ardinghelli
turgan6ek@Yerulans-MBP ~ % docker exec -it ad08d4d3d612 /bin/bash
root@ad08d4d3d612:/# cd
root@ad08d4d3d612:~# cd -a
bash: cd: -a: invalid option
cd: usage: cd [-L|[-P [-e]] [-@]] [dir]
root@ad08d4d3d612:~# cd -l
bash: cd: -l: invalid option
cd: usage: cd [-L|[-P [-e]] [-@]] [dir]
root@ad08d4d3d612:~# dir
root@ad08d4d3d612:~# ls
root@ad08d4d3d612:~# ls -a
.  ..  .bashrc  .profile
root@ad08d4d3d612:~# exit
exit
turgan6ek@Yerulans-MBP ~ % docker stop ad08d4d3d612
ad08d4d3d612
turgan6ek@Yerulans-MBP ~ % docker rm ad08d4d3d612
ad08d4d3d612
turgan6ek@Yerulans-MBP ~ % docker ps -a
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS                     PORTS                    NAMES
a9be2759eb69   hello-world          "/hello"                 3 hours ago     Exited (0) 3 hours ago                              blissful_bose
430cbef6f93d   postgres             "docker-entrypoint.s…"   12 days ago     Exited (255) 7 days ago   0.0.0.0:5432->5432/tcp   postgres-containe
r
d377c4500e3d   mongo:4.4.29-focal   "docker-entrypoint.s…"   8 weeks ago     Exited (0) 5 weeks ago                              mongo-db
2d754d45237c   c2a319b219d0         "docker-entrypoint.s…"   11 months ago   Exited (0) 3 hours ago                              postgres
turgan6ek@Yerulans-MBP ~ %
```

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.

```
turgan6ek@Yerulans-MBP Assignment 1 % docker build -t hello-docker .
[+] Building 2.1s (8/8) FINISHED                                  docker:desktop-linux
 => [internal] load build definition from Dockerfile                              0.0s
 => => transferring dockerfile: 107B                                              0.0s
 => [internal] load .dockerignore                                                 0.0s
 => => transferring context: 2B                                                   0.0s
 => [internal] load metadata for docker.io/library/python:latest                  2.0s
 => [1/3] FROM docker.io/library/python@sha256:7859853e7607927aa1d1b1a5a2f9e580ac90c2b66feeb1b77da97f  0.0s
 => [internal] load build context                                                 0.0s
 => => transferring context: 58B                                                  0.0s
 => CACHED [2/3] WORKDIR /app                                                     0.0s
 => [3/3] COPY app.py .                                                           0.0s
 => exporting to image                                                            0.0s
 => => exporting layers                                                           0.0s
 => => writing image sha256:402f8f763fb7a9f43386e6fbe730e49d8bdb1572ce7ea9ee6dcdd9865a06141c      0.0s
 => => naming to docker.io/library/hello-docker                                   0.0s

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
turgan6ek@Yerulans-MBP Assignment 1 % docker run hello-docker
Hello, Docker!
turgan6ek@Yerulans-MBP Assignment 1 %
```

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

```
turgan6ek@Yerulans-MBP Assignment 1 % docker images
REPOSITORY                                        TAG               IMAGE ID       CREATED          SIZE
hello-docker                                      latest            402f8f763fb7   6 minutes ago    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

```
turgan6ek@Yerulans-MBP Assignment 1 % docker images
REPOSITORY                                        TAG               IMAGE ID       CREATED          SIZE
optimized-hello-docker                            latest            c717a16ffad4   15 seconds ago   151MB
hello-docker                                      latest            402f8f763fb7   16 minutes ago   1.02GB
```

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
```

```
turgan6ek@Yerulans-MBP single-stage-go % docker push turgan6ek/hello-docker
Using default tag: latest
The push refers to repository [docker.io/turgan6ek/hello-docker]
55e9aefa8296: Pushed
47abe315f79f: Pushed
075c5d4a5750: Mounted from library/python
b2a9d25bd8bf: Mounted from library/python
90eec9d96301: Mounted from library/python
cc322244ab6f: Mounted from library/python
61f03db12f5b: Mounted from library/golang
391e69d2f4b6: Mounted from library/golang
61f1e6ae67cb: Mounted from library/golang
latest: digest: sha256:fea65990b3ba10f02fa0ecfeff72ac8272cfd0f7d656cd4eb0da3444553484f9 size: 2209
```