WSE: Docker

1) Introduction

Part 1: Docker Basics

- Slides

Part 2: Docker Hands-on

- Working with the Docker CLI
- Running a container

2) Working with the Docker CLI

```
# Let’s make sure docker is available
# Check if docker is installed correctly and display version information
docker version

# Detailed system-wide information about your Docker installation
docker info

# Let ’ s see if there is a container running on the system
docker ps

# Pull a docker image from the registry
docker pull ubuntu:24.04
docker pull ubuntu:latest

# To display all locally available docker images using docker images type:
docker images
docker image ls
```

3) Running a Container (Basics)

Let's experiment with how installing something into a container at runtime behaves Note: Modifying the contents of a container at runtime is not something you would normally do. We are doing it here for instructional purposes only!

```
# Create a container from the ubuntu image
docker run --interactive --tty --rm ubuntu:latest
docker run -it --rm ubuntu:latest

# Try to ping google.com
# This results in `bash: ping: command not found`
ping google.com
```

```
# Install ping
apt update
apt install iputils-ping --yes

# This time it succeeds
ping google.com
exit
```

The container is now stopped and removed. Let's try again

```
docker run -it --rm ubuntu:latest
ping google.com
```

It fails because we installed it into that read/write layer specific to the first container, and when we tried again it was a separate container with a separate read/write layer!

We can give the container a name so that we can tell docker to reuse it:

```
# Create a container from the ubuntu image (with a name and WITHOUT the --rm flag)
docker run -it --name wse-ubuntu ubuntu:latest

# Install and use ping
apt update
apt install iputils-ping
ping google.com
exit
```

3.1) Inspect the container and interact with the container

```
# List all containers
docker container ps -a
docker container inspect wse-ubuntu

# Restart the container and attach to running shell
docker start wse-ubuntu
docker attach wse-ubuntu

# Test ping: should now succed
ping google.com
exit
```

Command `docker attach`

- Connects the terminal directly to the main process (PID 1) of the container (process that was started when container was created, e.g., `/bin/bash`, `python ...`, etc.)
- The terminal is attached to the process's stdin/stdout/stderr
- If you exit the process (e.g., `exit` or `Ctrl + c`), the contaier may stop (unless it was designed to stay alive)

Note that the container is stopped again. When we restart the container, we can also use the following command to get an interactive access to the running container (recommended way)

```
# Restart the container get interactive access
(docker start wse-ubuntu)
docker exec -it wse-ubuntu bash

# Test ping and exit
ping google.com
exit
```

Command `docker exec -it ... bash`

- Starts a new process (in this case `bash`) inside the running container - separate from the container’s main process.
- This launches a new shell process inside the container, even if the container’s main process is something else (like `python ...`)
- When yout `exit`, this only ends the session. It won’t accidentally kill or disrupt the main process (container keeps running)

Comparision

- Use `docker exec -it <container> bash` when you want to inspect or interact with a running container safely. (recommended way)
- Use `docker attach <container>`, when you need to directly connect to the container’s primary process (e.g., to see its live output as if you started it).

3.2) Persisting data of a container

We generally never want to rely on a container to persist the data. Dependencies should be included in the image -> **use a Dockerfile**

But: often, applications produce data that we need to persist (e.g. database data, user data, logs, etc.) even if the containers are removed or recreated. To persist data, you need to store it outside the container’s writable image layer (container layer):

- Docker Volume
  - Plugin system (https://docs.docker.com/engine/extend/) with many implementations to integrate external storage systems such as NFS, Amazon EBS, Rclone (https://rclone.org/docker/), etc.
- Bind Mount

- (tmpfs Mount as a temporary store)

Volume Mount

Create a volume to safely persist the data:

```
# create a named volume
docker volume create wse-volume

# Create a container and mount the volume into the container filesystem
docker run  -it --rm --mount source=wse-volume,destination=/wse-data/ ubuntu
docker run  -it --rm -v wse-volume:/wse-data ubuntu

# ... with read only:
docker run  -it --rm --mount source=wse-volume,destination=/wse-data/,readonly ubuntu
docker run  -it --rm -v wse-volume:/wse-data:ro ubuntu

# Show mounted volume
ls -la

# Create and store a file into the mounted volume
echo "Hello world" > /wse-data/hello.txt
cat /wse-data/hello.txt

exit
```

Inspect volume through Docker Desktop and the CLI

```
docker volume inspect wse-volume
```

Create a new container and mount the existing volume:

```
# Create a new container and mount the existing volume into the container filesystem
docker run  -it --rm -v wse-volume:/wse-data ubuntu

# Print file content
cat /wse-data/hello.txt

exit
```

Bind Mount

Alternatively, we can mount a directory from the host system using a `bind mount`:

```
# Create a container that mounts a directory from the host filesystem into the container
docker run  -it --rm --mount type=bind,source="${PWD}"/data,destination=/wse-data ubuntu
docker run  -it --rm -v ${PWD}/data:/wse-data ubuntu

# Create and store a file into the mounted volume
echo "Hello world" > /wse-data/hello.txt
cat /wse-data/hello.txt

exit
```

3.3) Creating a custom Docker image

A Dockerfile specifies instructions on how a new Docker image should be built. You can start `FROM scratch` or use an existing Docker image as base for your custom image. Images are built using layers and metadata.

The following shows how Docker can be used to create a custom image that executes a Pascal program.

First, create the Pascal program and save it as `HelloWorld.pas`:

```
nano HelloWorld.pas
```

Enter the following text:

```
PROGRAM HelloWorld;
BEGIN (* HelloWorld *)
  WriteLn('hello, world');
END. (* HelloWorld *)
```

Save.

Next, create the Dockerfile and save it as `Dockerfile`:

```
nano Dockerfile
```

Enter the following text:

```
FROM freepascal/fpc:3.2.2-alpine-3.19-full
WORKDIR /app
COPY ./HelloWorld.pas .
RUN fpc -Mtp -Criot -gl -gh HelloWorld.pas
```

The steps listed here are all interpreted by the Docker builder during the build process.

1. Using the `FROM` instruction, this Dockerfile specifies that the base image should be `freepascal/fpc:3.2.2-alpine-3.19-full`, so we can use all installed programs, libraries, etc. that are present in this image, also while building the image.
2. The second line contains a `WORKDIR` instruction that tells the Docker builder that it should switch into the `/app` directory at this point in time while processing the Dockerfile and thus building the image.
3. The next step is to copy our file `HelloWorld.pas` into the image. We copy the file from our build context (`.`) to the current working directory (`.`) of the builder, which is `/app` at this point in time. Remember that we switched to it earlier. The build context is passed to the builder when we call `docker build` later on.
4. The last line instructs the builder to execute the command `fpc -Mtp -Criot -gl -gh HelloWorld.pas` during the build process, which compiles our Pascal program.

To initiate the build process, execute `docker build -t hello-pascal:latest .` in the shell

- Use `-t` to tag an image, i.e. give it a meaningful name (in our case `hello-pascal:latest`)
- Use `-f` to explicitly specify the Dockerfile that should be built (default: `Dockerfile`)

We can then watch the Docker builder build the image. Once the image is built, we can start a container based on our custom image as follows: `docker run --rm -ti hello-pascal:latest`:

```
docker run --rm -ti hello-pascal:latest
/app #
```

We have now opened a shell inside the container and can execute our compiled `HelloWorld` program and exit the container afterwards:

```
/app # ./HelloWorld
hello, world
Heap dump by heaptrc unit of /app/HelloWorld
0 memory blocks allocated : 0/0
0 memory blocks freed     : 0/0
0 unfreed memory blocks : 0
True heap size : 0
True free heap : 0
/app # exit
```

When we list our images, we can see that our image `hello-pascal:latest` is quite large. It has around 400 MB in size. This is because our custom image is based on the Free Pascal image, which is also almost 400 MB in size. We just increased its size a bit when we copied and compiled our Pascal program.

```
docker images
IMAGE                                   ID             DISK USAGE    CONTENT SIZE
...
freepascal/fpc:3.2.2-alpine-3.19-full   3e0378e6b308   398MB         0B
...
```

```
hello-pascal:latest                         c5383cbd51ba     399MB          0B
...
```

To execute our program, we do not need a whole Free Pascal installation anymore. This is where multi-stage builds come into play.

Create another Dockerfile and save it as `multistage.Dockerfile`:

```
nano multistage.Dockerfile
```

Enter the following text:

```
# --- this is stage 1
FROM freepascal/fpc:3.2.2-alpine-3.19-full AS build-stage
WORKDIR /app
COPY ./HelloWorld.pas .
RUN fpc -Mtp -Criot -gl -gh HelloWorld.pas

# --- this is stage 2
FROM alpine:3.19
WORKDIR /app
COPY --from=build-stage /app/HelloWorld .
CMD [ "./HelloWorld" ]
```

In a multi-stage build, each `FROM` instruction creates a new stage. The resulting image that is built is based on the final stage that the builder executes. All previous stages serve as intermediate builds, and we can reference these stages, e.g. to copy data from one stage to another.

In our case, the builder executes the same steps as before in our first stage, called `build-stage`. We then instruct the builder to start a second stage. This time, we tell the builder to base the final image on a much smaller image `alpine:3.19`. We again create the `/app` directory and copy our compiled program `HelloWorld` from the intermediate image, which was the result of our `build-stage`, into the second stage, and thus into the final image. Finally, we use the `CMD` instruction to specify which program should be started when starting a container based on our new image.

Build the image and start a container:

```
docker build -f multistage.Dockerfile -t hello-pascal:latest .
docker run --rm hello-pascal:latest
hello, world
```

The resulting image is much smaller now:

```
docker images
IMAGE                                     ID                DISK USAGE     CONTENT SIZE
...
alpine:3.19                               83b2b6703a62      7.4MB          0B
...
hello-pascal:latest                       7bc32c71c3ef      7.84MB         0B
...
```