Report

Gatzweiler Florian

1. Introduction

This is the report for the first exercise of the "Continuous Development and Deployment – DevOps" course at Tampere University in Finland 🇫🇮.

Information on running this Docker container via the command line is available in the ReadMe.md file!

2. Table of Contents

3. Platform used

The development platform is a `Macbook Pro 16-inch 2021`. It uses an `Apple M1 Pro` CPU and provides `16 GB` of memory storage. The software runs on `MacOS Tahoe`, version `26.0`.

The version of Docker used is `28.0.4`, build `b8034c0`.

```
docker --version
```

The version of Docker Compose is `v2.34.0-desktop.1`

```
docker-compose --version
```

3.1. Frameworks used

For the development of the services javascript (node.js) and python is used.

`service1` and `storage` using the Node.js framework in version `node:14.17.3-alpine3.14`

`service2` uses python in verison `python:3.10-alpine`

4. Graphical Representation

This section provides a graphical representation and explains how it can be read. The components, network, communication and volumes are then described.

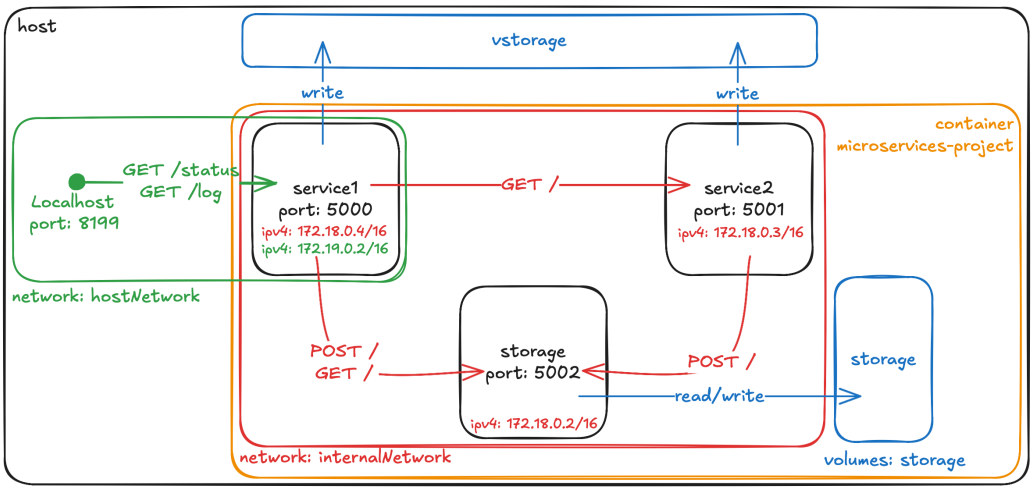

Diagram Showing the Services, Networks and Storage.[1]

Color / Style	Description
Host (Black)	Host boundary
Orange	Project/Container (the `microservices-project` Docker Compose project)
Services (Black)	Service name & internal port (inside the container)
Red	Internal, isolated Docker network `internalNetwork`; Internal connections between services
Green	Host-to-service communication via `hostNetwork` (e.g., `localhost:8199`).
Blue	Shared/Internal volumes and file operations (read/write access between services/host)

Description of the Graphical Representation.

4.1. Components

This image shows the architecture of the Docker-based microservices project and illustrates the relationships between the host, the three services (`service1`, `service2` and `storage`), their networks and the shared storage volumes.
The outermost black rectangle represents the `host` machine. It interacts with the `microservices` container via the `hostNetwork`. The docker software, which hosts our `microservices` container, runs on it. The `hostNetwork` is accessible to the `host` and `service1`, which is one of the three services hosted by the Docker container. However, there is also an `internalNetwork` that connects all the services and is completely isolated from the host.

Component	Ports	IPv4 Addresses	Connected To	Actions & Endpoints
`service1`	5000 8199 *(on localhost)*	`172.18.0.4`, `172.19.0.2`	`host`, `service2`, `storage`, *vstorage*	`GET /`, `POST /`, `/status`, `/log`
`service2`	5001	`172.18.0.3`	`storage`	`POST /`
`storage`	5002	`172.18.0.2`	*storage*	`POST /`
storage	-	-	`storage`	read/write
vstorage	-	-	`service1`, `service2`	write

Description of the Components.

4.2. Networking

The networks were created within the Docker Compose file. The *bridge* `driver` was chosen for both networks. This would also be the default `driver`. However, to avoid any communication with the host, it was decided to set the `internal` property to true for the `internalNetwork`. Additionally, the `attachable` property was set to prevent other containers from joining this network. This guarantees a completely isolated network. By omitting ports from the Compose file, Docker isolates the services from the outside world. However, this is not as straightforward as defining the network as internal! Moreover, if no network is created, Docker will define a `defaultNetwork` with the bridge `driver` and allow only those services with defined ports to be accessible by the host. While this is possible, it is not secure, and if the default settings change in the future, there could be a problem.

As mentioned above, only two networks were created in my case, providing all the necessary functionality! The following command produces the following output:

```
docker network ls
```

```
NETWORK ID     NAME                                      DRIVER
SCOPE
bc9fb40fd2a3   bridge                                    bridge
local
986c122cd4e9   host                                      host
local
abb67f73709c   microservices-project_hostNetwork         bridge
local
0aa0f05c95e7   microservices-project_internalNetwork     bridge
local
a32c00b91ead   none                                      null
local
```

The Docker networks starting with „microservices-project“ are related to my container.

`service1`:
Runs on port `5000`, with two internal IPv4 addresses (`172.18.0.4/16` and `172.19.0.2/16`). It is accessible from the host via port mapping (`localhost:8199`).

`service2`:
Runs on port `5001`, with internal IPv4 address `172.18.0.3/16`.

`storage`:
Runs on port `5002`, with internal IPv4 address `172.18.0.2/16`.

Internal communication is not based on IP addresses; service names are used instead for the server name!

The IPv4 addresses were extracted by running the following commands:

```
docker network inspect microservices-project_internalNetwork
```

```
docker network inspect microservices-project_hostNetwork
```

4.3. Communication Flows

Host to service1:
The `host` sends HTTP GET requests (`GET /status`, `GET /log`) to `service1` via the `hostNetwork` (green arrow).

service1 to service2:
`service1` sends HTTP GET requests to `service2` (red arrow labeled `GET /`).

service1 to storage:
`service1` sends HTTP POST requests to `storage` (red arrow labeled `POST /`).
It also sends an HTTP GET request to `storage` (red arrow labelled `GET /`).

service2 to storage:
`service2` sends HTTP POST requests to `storage` (red arrow labeled `POST /`).

4.4. Volumes

vstorage (blue lines/arrows): A shared volume mounted on both `service1` and `service2`, allowing both services to write to the same persistent storage on the host.

storage volume (blue line): The storage service has its own dedicated volume for persistent data, accessible only by `storage`.

5. Analysis of Records

This section analyses the records captured by ‚services1' and ‚services2'. It provides an overview of how the data was loaded and how relevant the measurements are. Some possible improvements are then suggested.

5.1. How Disk Space and Uptime were actually Measured

The disk space and uptime measurements were different for `service1` and `service2`. The result is the same for both. The following two records demonstrate the output generated on my computer. The first line is generated by `service1` and the second by `service2`.

```
2025-09-23T12:40:50Z: uptime: 0.00 hours, free disk in root:
972886 MBytes
2025-09-23T12:40:50Z: uptime: 0.00 hours, free disk in root:
972886 MBytes
```

service1

It uses the ‚process' property from the Node.js framework to calculate the uptime in seconds. The uptime in hours is calculated by dividing it by 3600 (seconds per hour) and limiting the number of decimal places.

```
const uptimeHours   = (process.uptime() / 3600).toFixed(2);
```

There is the possibility to import the `OS` library for the free disk space. It can calculate the free space, but as this didn't work for me, I chose another option. In UNIX environments, the command `df` can be executed to display information about the total and available space on a file system. By pointing to the root directory (`/`), we can obtain the desired result. With the optional `-m` option, the command returns space in 1M-block sizes. Then the information has to be extracted from the result. In order to run the command, a new library must be imported `const { execSync } = require('child_process');`.

```
df -m /
```

```
Filesystem   1M-blocks   Used    Available   Use %   Mounted on
overlay      1031017     2090    976482      0%      /
```

The information is extracted by splitting the result into two lines and separating the information blocks by splitting and removing the whitespaces. We then use the information from block 4 (block 3 in array-type writing).

```
const output = execSync('df -m /').toString();
const lines  = output.trim().split('\n');
const parts  = lines[1].split(/\s+/);
return parts[3];
```

service2
There is no approach like the one used at `service1`. At `service2`, the approach was to save the timestamp when entering the executable for the first time. The up time is then calculated by subtracting the current timestamp from the one saved at the beginning. Then it has to be converted into hours — done!

```
startTime = datetime.now(timezone.utc)
...
timestamp =
datetime.now(timezone.utc).replace(microsecond=0).isoformat().repl
ace('+00:00', 'Z')
uptimeHours = ((datetime.now(timezone.utc) -
startTime).total_seconds() / 3600).__format__('.2f')
```

For free disk space, an external library (`shutil) is used. This provides a function that returns the available space in bytes. Only the conversion from bytes to megabytes is then required.

```
total, used, free = shutil.disk_usage(path)
free_mb = free // (1024 * 1024)
```

5.2. Relevance of those Measurements
Both measurements provide basic information about the system on which they are running. Uptime is a simple indicator of service stability. In this environment, where the main task of two services is logging, we can discuss the relevance. Overall,

however, it is always important to log disk space and uptime. However, there are other important metrics to log.

5.3. Possible Improvements

Other important metrics, such as CPU usage, memory usage and network activity, should also be logged. Another improvement would be to log which service created which log entry. Regarding the way I get the free disk space in `service1`, improvements should definitely be made. This is more of a hack.

6. Analyses of Storage Solutions

This section outlines the differences between the two storage solutions used. There is one main differences between `vstorage` and `storage`. The host can access `vstorage`, but only the `storage` **service** can access `storage`.

6.1. vstorage

There is definitely a need to improve the way `vstorage` is implemented. As the requirement was that the log file should be accessible by `cat ./vstorage`, a way had to be found to store the log file inside the root directory of the container. This is normally not possible when mounting a directory in Compose, as this would create a new folder on the host machine or bind an existing one. Therefore, the GitHub repository provides a file called „vstorage“ which can be used instead of a folder by binding it. It is also not thread-safe. If two requests are made at the same time, thread safety must be implemented!

Good: Data survives container restarts and removals because it is stored on the host.
Good: It is also easily accessible by opening the log file on the host server. No Docker is needed for this.

Bad: This log file is written by two services.
Bad: This log file can be accessed by anyone with file access rights on the host system. It is not managed by the container.
Bad: Disadvantages compared to database file systems. (e.g. no version history).

6.2. storage

It is also not thread-safe. If two requests are made at the same time, thread safety must be implemented!

Good: This volume can only be accessed by the storage service, which ensures data integrity and reduces the risk of accidental overwrites by other services.
Good: It is managed by Docker, so everything is managed in one place. For example, moving the Docker container.

Bad: Limitation if shared access is required.
Bad: Disadvantages compared to database file systems. (e.g. no version history).

7. Instructions for Cleaning Up

This section contains all the information needed to clean the container.

To clean the `storage` volume created by the Docker Compose file, it just needs to be added to the `compose down` command in Compose, which removes the container and all its services, as well as any networks created beforehand.

```
docker compose down --volumes
```

To clear the `vstorage` file, which is stored in the root of the container inside the host as discussed above, access the relevant directory in the terminal. This file can be cleared by executing the following UNIX command and redirecting nothing into the file. This overwrites the file with empty content.

```
> vstorage
```

You can exit with `CONTROL` + `C`

8. Difficulties

The programming task itself was straightforward and easy to understand. The Docker documentation was excellent and easy to understand. So, in summary, I wouldn't say there were many difficulties at all. There were just a few minor issues, such as when I built my Docker Compose for the first time without a network. However, because Docker creates a default network for you, I had this network the whole time, which allowed my components to communicate even though this was not my intention! It wasn't difficult, it just took me some time to identify the error and understand why it occurred.

Also, I didn’t read the requirements carefully enough. Initially, I implemented a POST method to write to the log file via the storage service. I have since commented it out, so it is currently inaccessible, but this has cost me some time that I could have saved.

Moreover, I spent too much time thinking about how the logging should be done. My initial thoughts were correct when I entered the request handler. However, I was then confused by the requirements because they mention the following: „Note that the outputs of these commands ... should be equal, and contain two (2) lines per received /status-request.“. This led me to believe that I should probably only log requests for the `/status` endpoint. After changing it back and forth a few times, I talked to the professor, who said it should be fine to keep the original logic. So now I'm logging every request, which leads to two lines per received status request, as well as log entries for every request, like the `/log`.

9. Main Problems

My experience with Docker is more likely to install and use finished Docker containers. So my main problem was learning Docker first. I researched how to develop a Docker container. I discovered how useful the Docker documentation is. I also discovered the huge benefits of a Docker Compose file.

---

1. used tool: https://excalidraw.com