Report – COMP.SE.140 Exercise 1

Docker Compose & Microservices

1. Platform information

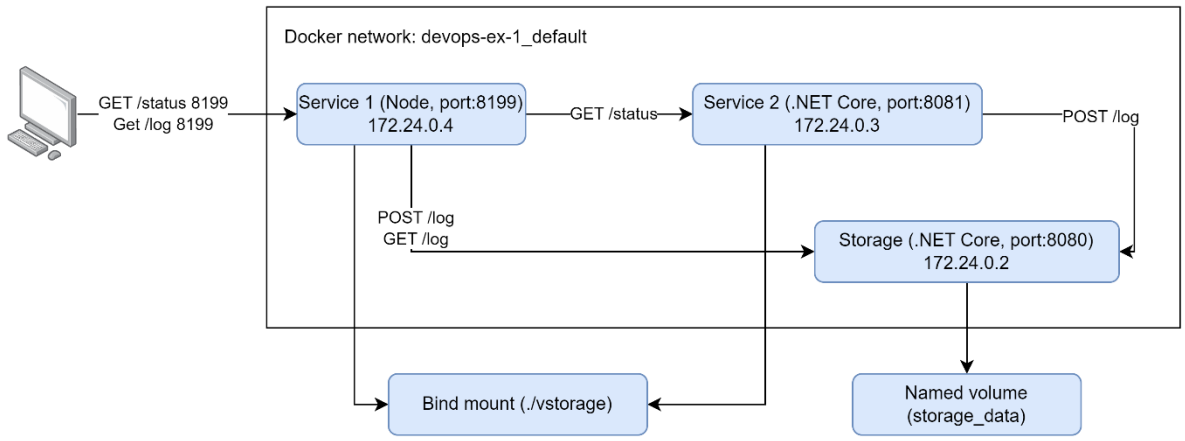
Hardware: Laptop PC, AMD Ryzen 7 3700U, 16GB

Operating System: Windows 11 Home (10.0.26100 Build 26100)

Docker Engine: v24.0.7, Docker Compose: v2.23.3-desktop.2

Docker Desktop for Windows (WSL2)

2. System architecture



The technology choices were based primarily on my own prior experience with these platforms.

- Service 1 was implemented with Node.js (JavaScript).
  - A minimal web server was built without any external libraries. All logic is contained in index.js.
- Service 2 was implemented with .NET Core (C#).
  - Built using top-level statements and the Minimal API pattern, which is suitable for an exercise of this size. All functionality is defined in Program.cs.
  - In real-world projects I prefer the traditional Program class structure, as it is in my view more explicit and maintainable than top-level statements.
- Storage was also implemented with .NET Core (C#).
  - Implemented similarly to Service 2, with top-level statements and a Minimal API.

3. Analysis of status records

Each /status request generates two rows in both persistent storages (the bind mount and the named volume):

Service1: Timestamp1 <ISO8601>: uptime Z.ZZ hours, free disk in root: Y MBytes

Service2: Timestamp2 <ISO8601>: uptime Z.ZZ hours, free disk in root: Y Mbytes

The measurements:

- Uptime: the time in hours since the container start. Decimals are used in order to get a meaningful value already within the first hour.
- Free disk: the available disk space in the container's root filesystem. This value is usually the same in both services, as they share the same Docker storage driver backend.

These measurements are not very useful for monitoring the actual health of the service. They only expose basic container-level system information. A more relevant approach would be to measure processor load, memory consumption, or to implement custom health checks inside the service.

4. Comparison of persistent storage solutions

In this exercise two alternative persistent storage solutions were implemented in parallel:

Bind mount (./vstorage)

- Good:
    - Very simple to implement, as the log file is directly visible on the host file system.
    - Easy to verify the contents with just looking into the file in the host system.
    - No need for separate service to hold the information.
- Bad:
    - Considered a poor design for production systems, as it tightly couples the container to the host file system.
    - Behavior may differ between environments (linux vs. windows) because of differences in file systems.
    - File locking and concurrent writes are not well defined.

Dedicated Storage service with a named volume (storage_data)

- Good:
  - Implements a clear REST API (POST /log, GET /log) that abstracts the storage from the clients.
  - Decouples the application services from the underlying storage implementation.
  - Scales better, as additional services could use the same API without direct file system access.
  - Data survives container restarts because of the named volume.
- Bad:
  - Slightly more complex to implement, since a dedicated service and API endpoints are required.
  - The stored log is not directly visible on the host and requires execution of commands inside the container.
  - Adds one more moving part to the system.

Summary:
The bind mount is simple and transparent, but not suitable for real-world systems due to portability and maintainability concerns. The dedicated Storage service with a named volume represents a much more realistic and extensible solution. It separates concerns properly and is the approach that would be taken in a production environment. The bind mount is therefore best seen as a pedagogical tool to demonstrate Docker's volume mechanism.

5. Teacher's instruction for cleaning up the persistent storage

Cleaning the bind mount (./vstorage)

rm -f ./vstorage/log.txt

Cleaning the named volume

To remove the log file from container

docker compose exec storage rm -f /app/data/storage.log

To remove the whole volume

docker compose down -v

6. Difficulties and problems faced

It was at first a bit difficult to understand the task and it's point. Not really a real-world example.

There was no linux machine available for me to use with docker, so I had to install ubuntu on my personal computer in a virtual box.

Overall, the exercise was quite straightforward, since I have used Docker in previous projects and at work. I had previously built services with Node.js and with C#, but I had not combined them into the same Docker Compose project before. This was therefore a useful new experience, but not a major difficulty.