Tampere University

COMP.SE.140 Continuous Development and Deployment - DevOps

Autumn 2025

Exercise 1: Docker-compose and microservices hands-on

Teemu Salonen

12.09.2025

Link for cloning the repository: https://github.com/SalonenTeemu/continuous-development-and-deployment.git

Contents

1 Basic information about the platform and system

I ran the system both on my local Windows PC and on a Linux Ubuntu virtual machine hosted on it. The Docker and Docker Compose versions are different between the two environments, but this did not seem to cause issues, even with the older versions.

Local Windows PC

- Operating system: Windows 11
- Docker version: 28.0.4
- Docker compose version: v2.34.0

Ubuntu virtual machine

- Operating system: Ubuntu 24.04.2 LTS
- Docker version: 27.5.1
- Docker compose version: 1.29.2

2 Services, network, and storage

Services

- Service1: Python (Flask), exposed externally through `localhost:8199`
- Service2: Node.js (Express), internal microservice called by Service1
- Storage service: Go language, provides a REST API to store/retrieve records to/from a PostgreSQL database

Storage and volumes

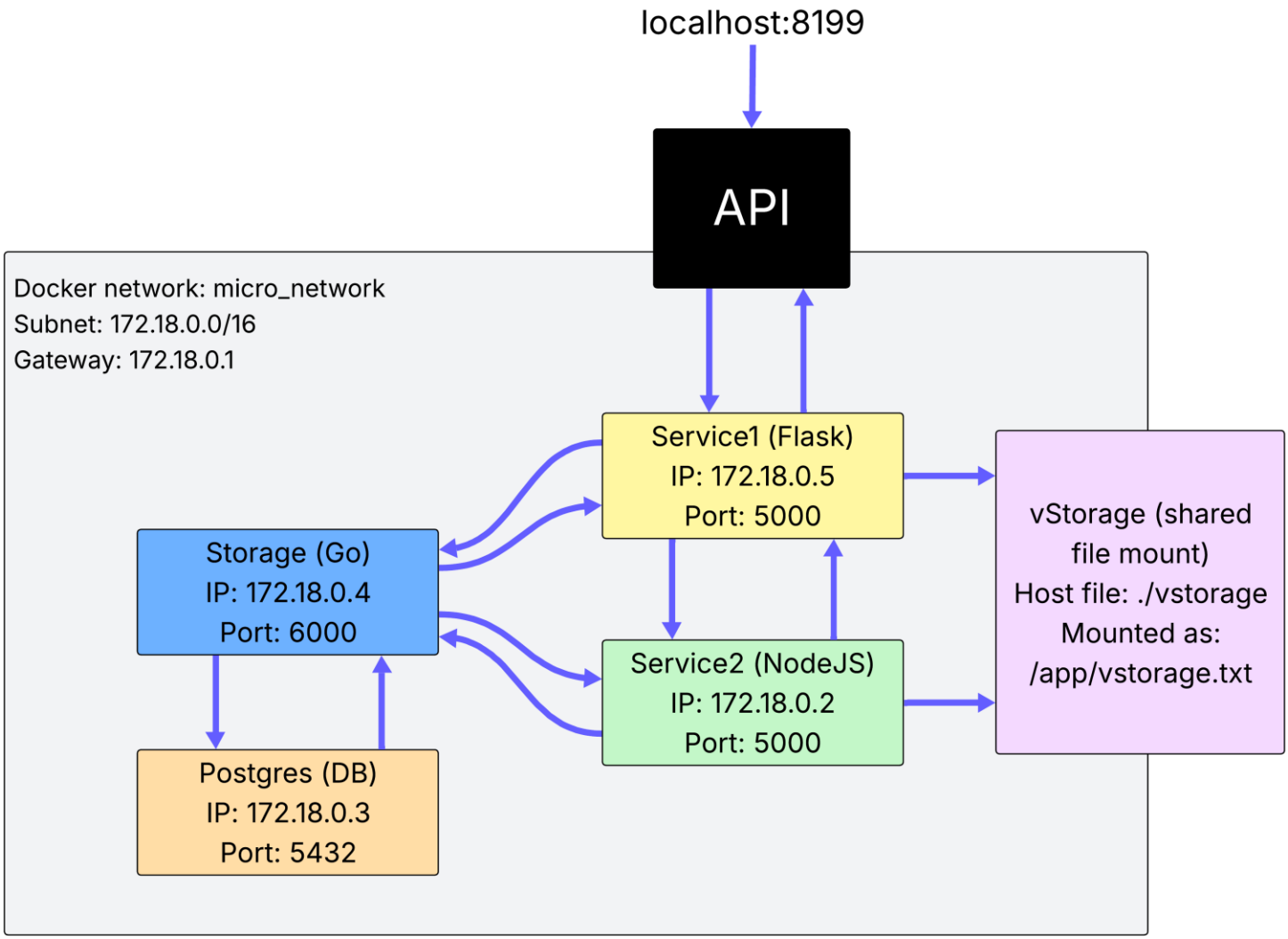
- vStorage: A single file in the project root, mounted into Service1 and Service2 for session persistence
- Postgres: PostgreSQL database used by the Storage service for persistent storage

As required by the task, vStorage is a single mounted file in the root of the project directory, shared between Service1 and Service2.

Network

- Micro network: The internal docker network between the containers.

Diagram



3 Analysis of the status records

Examples of the created status records

Local Windows 11 PC:

```
1. Service1@2025-09-11T08:33:21Z: uptime 2.29 hours, free disk in root: 956918 Mbytes
2. Service2@2025-09-11T08:33:21Z: uptime 2.29 hours, free disk in root: 1009363 Mbytes
```

Ubuntu VM:

```
1. Service1@2025-09-11T08:30:02Z: uptime 1.55 hours, free disk in root: 7900 MBytes
2. Service2@2025-09-11T08:30:02Z: uptime 1.55 hours, free disk in root: 9400 Mbytes
```

Observations

- The uptime is calculated from system boot time, which seemed to reflect the local PC and VM uptimes correctly. The uptime also seemed to be consistent across the services because it is based on the shared kernel, not the individual container start time.
- The calculated free disk space values roughly match the actual free space on the root disk. On the Ubuntu VM, the root was a logical volume (`/dev/mapper/ubuntu--vg-ubuntu—lv`) with around 8 GB free space. The values reported by the services differed slightly from the actual free space, especially in the case of Service2. This could be due to differences in how each library calculates disk usage. Python `psutil` library was used in Service1 and NPM library `systeminformation` in Service2. Overlay storage in Docker containers adds another layer that could affect the values.
- For more meaningful service-level metrics, free space should be measured on each service’s mounted volume (like `/app/vstorage`) rather than on the system root disk `/`. Measuring the root can still be useful if you want a rough view of the overall system’s underlying disk state.

4 Analysis of the persistent storage solutions

Option 1: Mount to local host file (./vstorage)

Pros:

- Simple and quick to implement
- Data persists between container restarts
- Easily shared between services via bind mount

Cons:

- Depends on the host filesystem → may not behave consistently across environments
- Can become a bottleneck for multiple writers → host processes like antivirus may also have an effect
- Risk of data corruption if multiple containers write concurrently
- For single-file mounts, the file must exist on the host or be created with a script or entrypoint before mounting

Option 2: Separate storage container

Pros:

- Storage is isolated and independent of other services
- Supports proper persistence mechanisms (databases, volumes)
- Isolation from host filesystem
- Can be managed, scaled, or updated independently

Cons:

- More complex setup and configuration
- Requires running and maintaining an additional container

In conclusion, option 2 aligns better with microservice principles by keeping storage separate, isolated, and scalable. I can still see option 1 being useful for simple demos or testing purposes where quick setup and simplicity are priorities.

5 Instructions for cleaning up the persistent storage

To clear the logs stored in the PostgreSQL database used by the Storage service:

1) Stop the running containers with: `docker-compose down`
2) Remove the named Docker volume holding the database data with: `docker volume rm continuous-development-and-deployment_pg_data`

Alternatively, to remove all containers and volumes at once, including the database volume, run: `docker-compose down -v `.

These commands will remove the named volume `pg_data` clearing all logs.

6 Difficulties and main problems

Difficulties and problems encountered:

- Docker bind mounts create a directory if the host path does not exist, which caused `IsADirectoryError` when attempting to mount a file. The exercise required a local file, so I created an empty file in the project root. Also, this could be possibly handled automatically, for example by using an entrypoint script.
- I had some minor initial problems implementing the Storage service because I decided to try out Go language for the first time.
- Some issues with the Ubuntu VM setup, especially related to networking (network adapter) which weren't directly related to the exercise but required some troubleshooting to solve.
- Overall, I had relatively few issues with Docker and Docker Compose side of the exercise but had more challenges when working with a new programming language and dealing with VM-related issues.