Ashkan Khademian
153124668
Software, Web & Cloud
Tampere University

Docker-compose and microservices hands-on

COMP.SE.140 - DevOps

Solution Overview

My Platform

- HW: Private Laptop, MSI Modern 14
- OS: Ubuntu 24.04.2 LTS
- Docker & Compose Versions:
    - Docker version 28.3.3, build 980b856
    - Docker Compose version v2.39.1

Architectural Description

Patterns

The main architectural pattern used in this system is Microservices. There are three high-level components (subsystems) that are brought up using a single docker-compose file, found at *docker-compose.yml*, and three different Dockerfiles located in each subsystem directory, i.e.:

1. *service1/Dockerfile*
2. *service2/Dockerfile*
3. *storage/Dockerfile*

The subsystems are communicating with each other via the HTTP protocol through the Docker default networks defined by the docker-compose.

Subsystems

There are three high-level components (subsystems) in this architecture. The description of each subsystem technology can be found in Table 1.

	Main Language	Framework	Environments	Other Assets
service1	Python	Flask	Development, Production	uWSGI
service2	Javascript	Node	Development, Production	-
storage	Python	Flask	Development, Production	uWSGI

Table 1. Services Technologies

As stated in Table 1, there are two environments introduced for each microservice: Development and Production. Development is totally local-based and allowed me to run and use the IDE debugger for verifying the implemented stuff. Production Environment is fully dockerized with docker-compose and is the final result ready for being submitted for grading.

Basically, switching between environments is done by using different set of environment variables. This is an example *.env* file used in my *service1* (which is not pushed to the remote):

```
FLASK_ENV=development
SERVICE2_URL=http://localhost:3000
STORAGE_URL=http://localhost:5001
SHARED_STORAGE_PATH=../data/vstorage
```

However, if you look into *.env.production* in the same service, you'll find production-ready env vars:

```
FLASK_ENV=production
SERVICE2_URL=http://service2:3000
STORAGE_URL=http://storage:5000
SHARED_STORAGE_PATH=./vstorage
```

The same structure goes with the other services.

On the other hand, the uWSGI used along with the Python-Flask servers is the common good practice for setting up Pythonic backend servers. Usually, a Python backend server (Django, Flask, etc.) is wrapped by a Web Server Gateway Interface (WSGI). the options are Gunicorn[1], uWSGI[2], etc. These WSGI tools wrap the Python application, handle the server worker processes, and redirect the external calls to the Python application to be handled.

Rationale: Frameworks Used

As most of my past experience in the industry was involved with Python-Django backend development, I chose Python as the main programming language in two out of three services. Since the backend servers needed for the requirements weren't complex that much, I chose the Flask framework, which is more lightweight than the Django framework. For setting up the baseline for the projects, in my past experiences, I've had some spiking project with Flask, and I've used the same baseline in *service1* and *storage*. You can see my prior work in microservices with Flask and Python in this repository: https://github.com/ashkan-khd/punchline-api
Basically, the same baseline for setting up the REST endpoints and configuring the Dev and Production Environments is reused.

[1] https://gunicorn.org/
[2] https://uwsgi-docs.readthedocs.io/en/latest/

As for the second programming language, in our first semester here at TUNI, I had a course COMP.CS.500 Introduction to Full Stack Development. There, I was taught to use Node.js as a backend service, and I've mostly reused the same structures for setting up my *service2* component. You can see my prior work in Node and Javascript in this repository, although I'm not sure if it's public to see by others: https://course-gitlab.tuni.fi/intro-2-full-stack-fall2024/nvs688

Deployment Diagram

The services that were explained so far are illustrated in Fig 1.

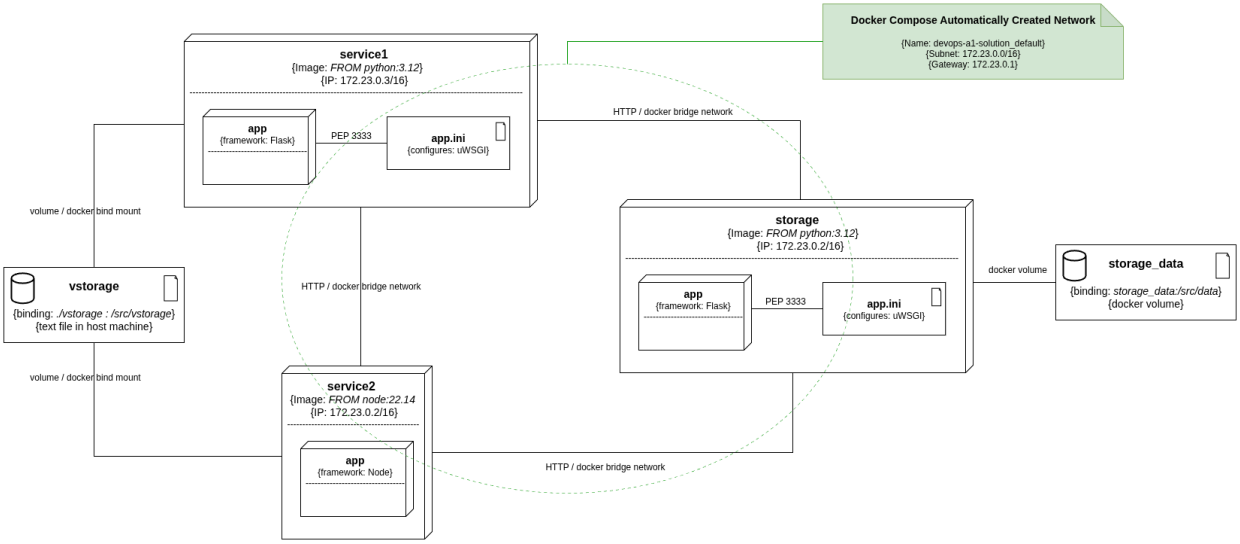

Fig 1. UML Deployment Diagram

All three main deployment nodes in Fig 1 are Dockerized containers that are set up using docker-compose. Both *service1* and *storage,* which include a Flask application (app), containing all the implemented logic, are set up for production use with the help of uWSGI. The configuration for uWSGI can be found in *app.ini* in the same service. *service2* is a simpler container running one node backend server.

The communication between the deployment nodes, each of which a dockerized container, is realized over a Docker network. below, you can see what assets are created after initializing the project with *docker compose up -d*.

```
❯ docker compose up -d
[+] Running 5/5
 ✔ Network devops-a1-solution_default        Created
0.0s
 ✔ Volume "devops-a1-solution_storage_data"  Created
0.0s
 ✔ Container devops.service2                 Started
```

```
0.4s
 ✔ Container devops.storage                  Started
0.4s
 ✔ Container devops.service1                 Started
0.4s
```

The *devops-a1-solution_default* network, which is created by the docker-compose file by default, is a bridge network over which nodes can refer to each other with both container name and docker-compose service name. Meaning that *service1* wants to call `service2, it can simply direct the request to: http://service2:3000`. The routing is automatically done by the network defined by Docker without the need to specify the exact IP address of the other service. As the real final IPs of the services are never referenced in the application code, it's important to note that the IPs mentioned in the deployment diagram are subject to change with every deployment of the architecture.

As for storage, there are two storage units in use:

1. *vstorage*: Bind Mount
2. *storage_data*: Named Volume

vstorage is simply a text file in the host machine, available in the repository next to docker-compose.yml. These types of Docker volumes are known as bind mounts. This mount storage is bound to both *service1* and *service2*, where both of them can read from and write into. On the other hand, *storage_data* is a Docker Named Volume that persists the database inside the *storage* container, so that with every system restart, the database won't be erased.

Internal Design Description

Patterns and Practices

The same internal architecture is used for the design of both Python and JS systems. The current design is highly influenced by practices in Component-based Design and Object-oriented Design. This way, I was able to harness the changes by creating a loosely coupled system.

Component Diagram

In Fig 2, you can find the Component Diagram of the system. Because of the high similarity between the internal components of each subsystem (i.e., *service1*, *service2*, and *storage*), the internal specifications of *service2* and *storage* are abstracted away and we only explain how service1 works.

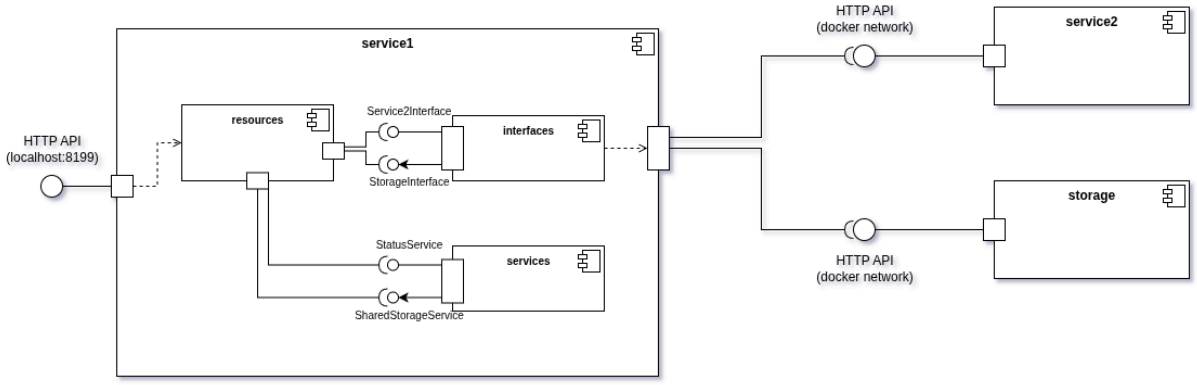

Fig 2. UML Component Diagram

Inside *service1*, there exist three components. The *resources* component publishes each endpoint as one Python class. These resources depend on *interfaces* and/or *services* components. The *interfaces* component contains classes that make HTTP requests to *service2* and *storage* components (which are needed in the requirements). The *services* components contains functionality classes for saving data in *vstorage* (i.e., *SharedStorageService*) and getting the status of system (i.e. *StatusService*).

In a similar manner the other components are implemented. The only difference is that in *service2*, the *resources* component is called *controller*.

Output Analysis

Analysis of the status records

Questions:

1. Which disc space and uptime you actually measured?
2. How relevant are those measurements?
3. What should be done better?

A sample *curl* *http://127.0.0.1:8199/status* to the project would output like below:

```
2025-09-26T16:45:06Z: uptime 8.907 hours, free disk in root: 36207.74 MBytes
2025-09-26T16:45:06Z: uptime 8.907 hours, free disk in root: 36207.74 MBytes
```

Since the timestamps in both *service1* and *service2* are accurate to the second, not more, and as the request is handled in a fraction of a second in all the microservices setup, both services output the same timestamp. Similarly, both components calculate the uptime of the system by reading through “/proc/uptime”, and the free disks by going through the usage of the root

directory “/”, the values for uptime and free disk are also similar in both systems. In short, both containers report the uptime of the host machine for the Docker engine (the uptime of my local computer, not the containers), and the free disk space on my local machine as well. So I could say these metrics are host-relevant rather than container-relevant.

I believe if I had calculated the actual uptime of each container, maybe the status being reported would’ve been more meaningful. Additionally, it would’ve been better if I had set the timestamp to be millisecond-specific so that I could see the exact time of the request being received in each container.

Comparison of the persistent storage solutions

Questions:

1. What is good in each solution?
2. What is bad in each solution?

It’s already been established what the different types of storage are used in this project in the Deployment Diagram section. Here I discuss the pros and cons of each.

vstorage (Bind Mount)

Pros. This storage is easy to debug and maintain. To see into its contents, you can simply use *cat vstorage*. In terms of implementation, *service1* and *service2* only have to write into a local file since they don’t even know that this storage is being shared between them. So the operations are as simple as reading from and writing into the files.

Cons. There is a high possibility of a race condition. At the moment, the implementation in *service1* and *service2* is simple in my setup, as these containers don’t use that many asynchronous capabilities available in their programming languages. But assume that *service1* would asynchronously write into *vstorage* and concurrently call *service2*’s */status* endpoint. This would make both services write their status into *vstorage* at the same time. This is an operation that would most likely result in unexpected results in the *vstorage*. In general, the lack of isolation of your local storage is a bad practice in microservices architecture that backfires with race conditions like this.

Storage (Container)

Pros. Storage is a standalone container for the database operations (reading the logs and appending to the existing logs). This results in more control over accessing the shared data from outer services (*service1* and *service2* in our case). At the moment in my implementation, a trace starting from *localhost:8199/status*, goes through services in a complete sequential manner. but in complex cases where *service1* and *service2* might call *storage* in parallel, some simple locking

mechanism for writing into the database of the *storage* container can be implemented, so it would be possible in this type of storage to avoid race conditions. In our implementation, the database of Storage container is bound to some Docker Named Volume, *storage_data*, which is created with this description in docker-compose.yml:

```
volumes:
  storage_data:
```

This ensures that with every restart of the containers or the host machine, the database for the *storage* container remains intact (not erased).

Cons. This approach is rather more complicated than using only a simple bound mount like before. We have to create and maintain an entire backend service for managing access from other services to the data. Other services would also need to require the interface provided by the *storage* container, which here is under REST endpoints, but can be other protocols like gRPC and so on.

Instructions for cleaning up the persistent storage

vstorage. Since vstorage must be provided for *service1* and *service2*, removing it entirely will cause issues for bringing up *service1* and *service2* in the next deployment rounds. One must ensure that at least an empty text file called *vstorage* is provided before using *docker-compose up –-build*.

Storage. As explained before, *storage* depends on a Docker Named Volume called *storage_data*. Since usually removing it will cause issues for the *storage* container to operate (as it won't have a data store), it's recommended that for removing the database, this command is used:

```
docker-compose down -v
```

This will stop and remove all the containers and then this Docker volume. Alternatively, one can remove this Docker volume using this Docker command, if the volume is not in use (the containers are already down):

```
docker volume rm devops-a1-solution_storage_data
```

NB! The Docker volume name mentioned above, *devops-a1-solution_storage_data*, is the result of the automatic addition of a prefix by docker-compose. It might be different in different versions of docker-compose, so using *docker-compose down -v* is a more generic option.

Conclusion

What was difficult? What were the main problems?

It was a little hard to understand the main goals and requirements in the docs of the assignment. I'm still not 100% sure that I got it right, and it's the complete solution for that. I guess I had to go through the requirements more than five times, and still, I was figuring out newer things as I was halfway through implementing the project.

Secondly, as I think about the actual functional requirement being implemented, I see that the project was too small. I just implemented some lightweight servers with no more than two endpoints. However, it took me way more than what I initially expected to implement this. because there were too many non-functional (quality) requirements to consider. Also, the changes were dispersed, and for a new change I might've made in *storage*, I had to change different places in *service2* and then *service1,* and test the entire program once more to ensure everything is in place.