COMP.SE.140 – Docker-compose an microservices hands-on

Version history

V0.1 03.09.2025	First draft for teaching staff
V0.2 04.09.2025	Review+changes by teaching staff
V1.0 07.09.2025	First version to students
V1.1 08.09.2025	First fixes discovered during lecture preparation
V1.2 09.09.2025	Fixed version to students.
V1.3 16.09.2025	Small clarifications (see red color)

Synopsis

The purpose of this exercise is to learn (or recap) how to create a system of two interworking services that are started up and stopped together. This requires creation of your own Dockerfiles and docker-compose.yaml, and also development of simple applications code. The application can be implemented in any programming language (shell script and HTML not allowed), but different programming language must be used for the two applications. The teaching staff is interested in seeing as may programming languages as possible. *[Last time we gave such challenge in 2020 we got 14 different programming languages – I hope you can beat this!]*

In addition to the above practical implementation task, you should make observations about some principles and technologies that cloud-native microservices rely on. In a way this exercise is lowest in the abstraction level (closest to bits) in this course.

Learning goals

- Recap your hands on with Docker and Docker Compose. This is assumed to be known from earlier courses and will be needed in the next steps of the course.
- Recap the principles of TCP/IP and HTTP-protocols.
- Understand the relation of containers to the operating system – what kind of virtualization it actually is. Thus, we play with some operating system concepts.
- Understand the container networking – we play with docket networks.
- Observe the role of persistent storage – two alternative ways are implemented and compared.

Task definition

In this exercise we will build a simple system composed of three small services. Two of them (`Service1` and `Service2`) should be implemented in different programming languages. A third service (`Storage`) contain one implementation of shared persistent storage.

The detailed specifications of the services are

`Service1`

- The only service that can be called from outside. If other services are accessible, the points are reduced.
- Analyses its state and creates the following record:

```
Timestamp1[1]: uptime <X> hours, free disk in root: <X>
MBytes
```

- Works as proxy: forwards the received requests to `Service2` and `Storage`
  - /status => `Service2`
  - /log => `Storage`

[1] ISO 8601 format and UTC as "2025-09-03T12:06:18Z" should be used.

- Stores a log of the incoming requests to two alternative persistent storages:
    - The one maintained by **`Storage`**-service.
    - The other implemented with a volume named **`vStorage`**

`Service2`

- Analyses its state and creates the following record:

  ```
  Timestamp2: uptime <X> hours, free disk in root: <X> Mbytes
  ```

  X can be either an integer or a decimal number.

- Stores a log of the incoming requests to two alternative persistent storages:
    - The one maintained by **`Storage`**-service. You can select how this is implemented.
    - The other implemented with a volume named **`vStorage`**

`Storage`

Implements a simple REST-interface

- HTTP POST /log => append the incoming record persistently (the log does not disappear when the container is stopped, but your document should tell how the teacher can clean the log (see below); text/plain is used as a MIME-type
- HTTP GET /log => gets the content of whole stored log; text/plain is used as a MIME-types

The REST interface and expected behaviour of whole system is the following

GET localhost:8199/status

1. **`Service1`** analyses its status and creates the above-described record
2. **`Service1`** sends the created record to **`Storage`** (HTTP POST Storage)
3. **`Service1`** writes the record to at the end of **`vStorage`**
4. **`Service1`** forward the request to **`Service2`** (HTTP GET Service2)
5. **`Service2`** analyses its status and creates the above-described record
6. **`Service2`** sends the created record to **`Storage`** (HTTP POST Storage)
7. **`Service2`** writes the record to at the end of **`vStorage`**
8. **`Service2`** sends the record as a response to **`Service1`**. (text/plain)
9. **`Service1`** combines the records (record1\nrecord2) end returns as a response (text/plain)

GET localhost:8199/log

1. **`Service1`** forwards the request to **`Storage`**
2. Returns the content of the log in `text/plain`.

The persistent storages have two purposes: 1) to store data between sessions and 2) share data between services. In this exercise we have two implementations of the storage:

1. Simple that mounts the **`vStorage`** in container to local file **`./vstorage`** of the host. Note, that this is considered as a bad design, but we use it for learning purposes
2. One that is based on a separate container. The internal implementation of this container is not specified by the teaching staff. However, you should instruct how the teacher should erase the data.

Note that the outputs of these commands

```
cat ./vstorage
curl localhost:8199/log
```

should be equal, and contain two (2) lines per received /status-request.

Attached report

The document should contain

- Basic information about the platform you used:
  - HW and/or virtual machine
  - Operating System
  - Version numbers of Docker and Docker-Compose
- A diagram showing the services, network and storage. Add service name and IP-address of each service in the diagram.
- Analysis of the content of the status records. Which disc space and uptime you actually measured? How relevant are those measurements? What should be done better?
- Analysis and comparison of the persistent storage solutions – what is good and what is bad in each solution?
- Teacher's instruction for cleaning up the persistent storage
- What was difficult?
- What were the main problems?

Submitting for grading

After the system is ready the student should return (in the git repository – in branch "exercise1").

- Content of three Docker and one docker-compose.yaml files
- Source codes of the applications.
- Output of "**`docker container ls`**" and "**`docker network ls`**" (executed when the services are up and running.) in a text file "docker-status.txt"
- Report as specified above in file Report.pdf. If you use .md files for documenting, please transform to pdf.
- Optional "llm.txt" (see below)

Please do not include extra files in the repository.

These files are returned with some git service. A course-gitlab repository can be created to course-gitlab if you request one. Any git-repo that the staff can access without extra effort is ok.

You should prepare your system in a way that the course staff can test the system with the following procedure (on Linux):

```
$ git clone -b exercise1 <the git url you gave>
$ docker-compose up --build
… wait for 10s
$ curl localhost:8199/status
$ docker-compose down
<students instructions for cleaning up>
```

Grading

The points from this exercise depend on timing and content:

- Maximum 10 points are given. 1 point is 1% in the final course grading.
- how well the requirements (including technical instructions to the submit your project) are met: 6p
- following the good programming and docker practices: 2p
- quality of the document: 2p
- missing the first deadline (29.09.2025): points reduced by 0.5 points / starting day.
  The absolute deadline is 07.10.2025 2359.

Note: the assessment is done on Linux operating system. If you develop on some other operating system, try to test on some Linux.

On using ChatGPT or similar AI (large language models - LLM) tools

The university-level guidelines say:

> *"If a student uses a language model in an assignment or a thesis, for example, as part of language editing, this must always be mentioned. When individual students describe their use of language models, we can share good practices. The use of a language model for language revision is justified, for example, to produce a grammatically or structurally fluent text (cf. proofreading and translation tools and similar tools)."*

In this exercise we interpret this as follows

- If language models are used, a separate report (llm.txt) must be written (included in the submission). This report includes:
  - The used LLM tool
  - Motivation/reason to use LLM
  - How and why LLM helped
  - What kind of mistakes LLM did
  - What were things that LLM was not able to provide
- You allow course staff to use this report in grading and use of it (after anonymization) for teaching development and research purposes.
- The course staff will investigate different ways to discover use of LLM – students using LLM without reporting it, will be discontinued from the course.

Hints

It might be a good idea to create and test the applications first.

Do not provide the link from the browser (the one you see when you access your repo with a GUI) – that does not work with git clone, and your points will be reduced. Check that

```
    git clone -b exercise1 <the git url you gave>
```

really works. Failing this will reduce 1 point.

Useful material:

- Material section of the Moodle page
- docs.docker.com

Docker images are easy to access, if they are tagged when built

```
    $ docker build --tag=service1 .
```

If Docker image is rebuilt, docker-compose should also be given a hint that rebuilt should override the existing one

```
    $ docker–compose up –build
```

Volumes in docker have many features (see https://docs.docker.com/engine/storage/volumes/ if you want to learn fine details. However, in this exercise simple directives in compose.yaml should be enough. Typical errors are those discussed in here: https://stackoverflow.com/questions/42248198/how-to-mount-a-single-file-in-a-volume