1 COMP.SE.140 Project – Fall 2024

1.1 Introduction

Version history

0.1	03.11 Kari	Initial draft
0.2	10.11 Kari	Second draft for the course staff
1.0	11.11 Kari	First version published to students
1.1	12.11 Kari	First fixes
1.2	21.11 Kari	Added requirements for deployment
1.3	26.11 Kari	To respond student queries
1.4	08.12 Kari	To respond further student queries

Text new or changed in 1.4 in red.

Main learning of this exercise is to have a practical experience with a CD pipeline and teach you how create such pipeline to automatically build, test and deploy the code to the hosting environment. In addition, the students will get some basic understanding of the OPS-side. An average student is assumed to spend about 50h hours with this project.

Notes:

- **the students are expected to read this document carefully**
- **these instructions will be completed with instructions on how to deploy by 22.11**

1.2 The schedule

• The instructions disclosed:	11.11.2024
• Students can start studies and preparations.	
• Discussions in the lecture:	12.11.2024
• Students are asked to give clarification questions	
• Deployment instructions	22.11.1024
• Latest submission if you want course graded in 2024:	09.12.2024
• Latest submission to pass the course:	31.01.2025

2 The exercise

You will further develop the previous exercise (with nginx) and build a CI/CD pipeline for it. Instead of using the most advanced and popular technologies, students are asked to implement automated pipeline from rather primitive open source components. The aim is to create “under the hood” understanding.

The project should be developed in git branch called “project” so that the previous branches are not be touched. Then, the teacher can test the nginx exercise while you are working on the project.

The target system is shown in Figure 1. As you can see it is an evolution of the code you have developed in previous weekly exercises. Now your task is to create a CD pipeline, and then by using this pipeline to further develop the target system.

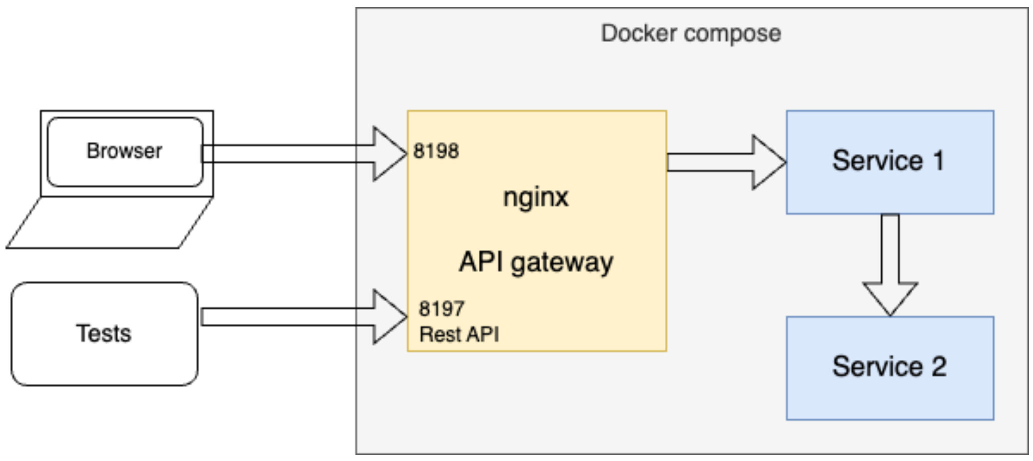

Figure 1.The target system.

Thus, the main phases of the project are:

1. Create the pipeline infrastructure using gitlab-ci. This means that you should:
   - Add a second remote to your repository (see Section 4)
   - Install your gitlab runner. Register your runner to the newly created remote. (see Section 5).
   - Define the pipeline using gitlab-ci.yaml for the application you implemented for the nginx exercise. We aim at continuous deployment, so a running system should be created automatically. The containers should be started automatically. (In other words: "git push (to the second remote) => the system is up and running). The pipeline should have at least the following phases: build, test and deploy.
   - Test the pipeline with the current version of the application.
2. Create, setup and test an automatic testing framework
   - First, you need to select the testing tools. We do not require any specific tool, even your own test scripts can be used.
   - The tests should at least include the functionality shown to external world, i.e., the responses of the API gateway. One test per API call is enough. *Optionally, the tests can cover some individual services, too.*
3. Implement changes to the system by using the pipeline during the development (see Section 6). The development should be done in test-driven manner (test before implementation – see https://en.wikipedia.org/wiki/Test-driven_development)
   - For each new feature, you should first implement tests, then implement the feature and after passing the tests move to next feature. This behavior should be verifiable from in the version history.
   - Tests must be in a separate folder "tests" at the root of your folder tree.
   - No need to aim at full coverage – a single test per new feature is enough.
4. *(Optional) implement a static analysis step in the pipeline by using tools like jlint, pylint or SonarQube.*

5. *(Optional) implement monitoring and logging for troubleshooting. This should be shown in the browser view to API gateway (the one accessed through port 8098). It should show at least start time of the service, number of requests it has received after start.*
6. Implement deployment. There are two alternatives:
   - **Simple** that does not give any extra points: deployment system starts the docker containers so that they can be tested by the teacher on a local machine. This is very simple "deploy to local machine" that everybody would call deployment.
   - **Challenging** that uses a separate virtual machine accessible (run the pipeline, test the system with browser. No SSH-login assumed) by the teacher. Recommendation is to use CSC virtual machine (see instructions from Moodle or directly from https://moodle.tuni.fi/pluginfile.php/4841520/mod_resource/content/1/CSC%20Instructions.pdf).
7. Provide an end report (file "EndReport.pdf" in the root of the repo). Rough table of content given in Appendix A.

3 Group-work option

This project has been designed to be an individual work, but group work may be possible on student's request. The request should include

- Group members (email + student id)
- Proposed additional work (features or additional practices)
- Planned worksplit and how to make it visible

4 New GitLab server and registration of your pipeline

Since students are not allowed install their own CI-pipelines to course-gitlab we have installed our own at https://compse140.devops-gitlab.rd.tuni.fi . You should

- Register yourself in that server.
- After that, make https://compse140.devops-gitlab.rd.tuni.fi as second remote to your repository. (There are multiple instructions, for instance https://docs.github.com/en/get-started/getting-started-with-git/managing-remote-repositories)
- Check that you can push your code to the new repo.

5 Create and install your github runner

A nice tutorial: https://docs.gitlab.com/runner/install/index.html

Information about Gitlab-CI

- Course content from 2022. Slides: https://plus.tuni.fi/graderA/static/compse140-f2023/_downloads/2022_22GitlabCI.pdf and video: https://tuni.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=9b4b0c25-1ae9-4b03-858e-af4e013c70e1
- Gitlab's own documentation: https://docs.gitlab.com/ce/ci/

6 The application and its new features

The starting point of the application is the nginx exercise (two + Nginx) you already have, but extended with some new features:

1. The system should have the following states:

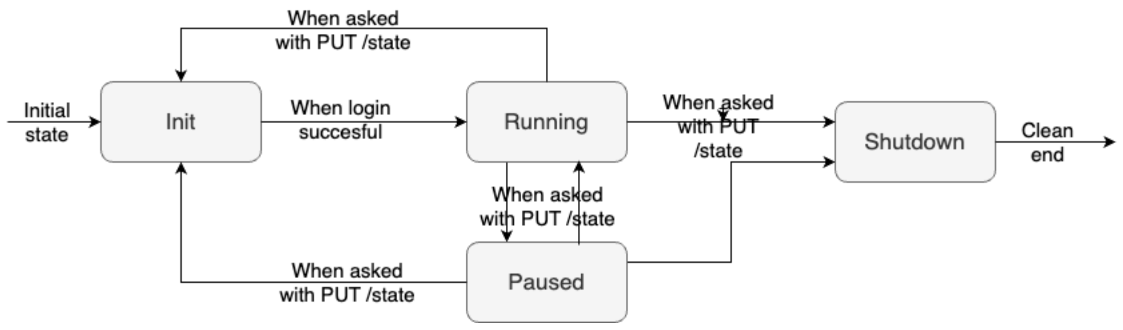
Initial state
Init
When login succesful
Running
When asked with PUT /state
When asked with PUT /state
Shutdown
Clean end
When asked with PUT /state
Paused
When asked with PUT /state

Figure 2. States of the system.

The system should remember all the state transitions. Note that the behavior of the system should correspond to the state.

2. The nginx service should be extended to an API gateway. You can decide if the nginx and API gateway are in same process (i.e., yellow box in Figure 1 may be split to two), different services and separate containers. The API gateway should listen port 8197 and provide the following REST-like API. The API works even without login, but the state cannot be changed from INIT. (Note: this would be considered as a security problem if this were a real system.)

PUT /state (payload "INIT", "PAUSED", "RUNNING", "SHUTDOWN")

PAUSED = the system does not response to requests
RUNNING = the system responses to requests normally
If the new state is equal to previous nothing happens.

There are two special cases:
INIT = everything (except log information for /run-log) is set to the initial state and new login is needed get the system running again. However, the containers do not need to restart, INIT is only about application state.
SHUTDOWN = all containers are stopped

GET /state (as text/plain)

get the value of state.

GET /request

Similar function as REQUEST-button of the GUI (see instructions for nginx exercise), but as a text/plain response to the requester.

GET /run-log (as text/plain)

Get information about state changes
Example response:

```
2023-11-01T06.35:01.380Z: INIT->RUNNING
2023-11-01T06:40:01.373Z: RUNNING->PAUSED
```

```
2023-11-01T06:40:01.373Z: PAUSET->RUNNING
```

7 Implementation constraints and hints

Many implementation issues have been left open on purpose – the students should find answers by themselves.

These instructions assume that students have their own Linux virtual machine, but use of Windows, Mac or any other option is not forbidden. In case of “exotic” options are used, the student is responsible of making the evaluation of the outcome possible. The assistant will use Linux, so make sure that the system work in Linux.

Implementation of the feature “monitoring and logging for troubleshooting” can be done in many ways, but a simple web-page is one natural option.

8 Submitting the project

Use git branch “project” for returning this!

9 Grading

As already been communicated this project affects 40% of in the evaluation of the overall course. For that 40% we use the following table

- The system works according to requirements	0..15 %
- The commit history shows that test-driven approach has been used	0..5 %
- The CI/CD pipeline is clean and complete	0..10 %
- Overall quality (clean code, good comments, ….)	0..5%
- Quality of the end report	0..5% (+ up to 5% compensation of a good analysis of your solution and description of a better way to implement.)
- Implementation of optional features (each optional feature is worth of 5%) o Testing of individual services o Static analysis step in the pipeline o Monitoring and logging for troubleshooting. o Deployment to external server.	0..15 %

Note: optional points can compensate problems elsewhere, but the total sum is capped at 50%. That means that max 10% can be used to compensate lost points in exercises and exam.

10 About assessment

To keep the required effort of assessment bearable, we assume

- The system (the extended application) can also be started up with following command sequence. Gitlab and gitlab-runner might be started with separate action.

  ```
  $ git clone -b project <the git url you gave>

  $ cd <created folder>

  $ docker-compose build –-no-cache

  $ docker-compose up -d
  ```

- Testing can be done with simple curl-commands and the API spec must be followed. In case you are unsure about the spec, please ask. For example,

  ```
        curl localhost:8197/state -X PUT -d ”PAUSED” \
            -H ”Content-Type: text/plain”
            -H ”Accept: text/plain” (except node-statistics)
  ```

  needs to work. **I.e. the content-type is assumed to be text/plain. If you use a browser in your own testing, you may fail. Use curl (or postman) during your development!**
  (It would be nice to test the gitlab CI. The assistant might try something, but we understand that the setup does not work for a cloned git repo out of the box.)

11 Appendix A - template for the document

Instructions for the teaching assistant

Implemented optional features

List of optional features implemented.

Instructions for examiner to test the system.

Pay attention to optional features.

Data about the platform you used in development (hardware, CPU architecture, operating system, version of docker and docker-compose)

Description of the CI/CD pipeline

Briefly document all steps:

- Version management; use of branches etc
- Building tools
- Testing; tools and test cases
- Packing
- Deployment
- Operating; monitoring

Example runs of the pipeline

Include some kind of log of both failing test and passing.

Reflections

Main learnings and worst difficulties

Especially, if you think that something should have been done differently, describe it here.

Amount effort (hours) used

Give your estimate