COMP.SE.140 Project – Fall 2022

Introduction

Version history

1.0	14.11 Kari	Released to students
1.1	15.11 Kari	Small fixes
1.2	17.11 Kari	Clarification on GitLab version
1.3	19.11 Kari	Fixed some bugs
1.4	20.11 Kari	Still some errors (on the CURL-command) fixed
1.5	22.11 Kari	One put was still

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.

Note: the students expected to read this document carefully.

The schedule

- The instructions disclosed: 14.11.2022
  - Students can start by installing the gitlab-ci
  - New versions to resolve ambiguous parts may be published later.
- Discussions in the lecture: 15.11.2022
  - Students are asked to give clarification questions
- Latest submission if you want course to graded in 2022: 07.12.2022
- Latest submission to pass the course: 31.01.2023

The exercise

You will further develop the message queue software 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 it to creating “under the hood” understanding.

The project should be developed in git branch called “project” so that branch “main” should not be touched. (The teacher can test the message queue example separately.).

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

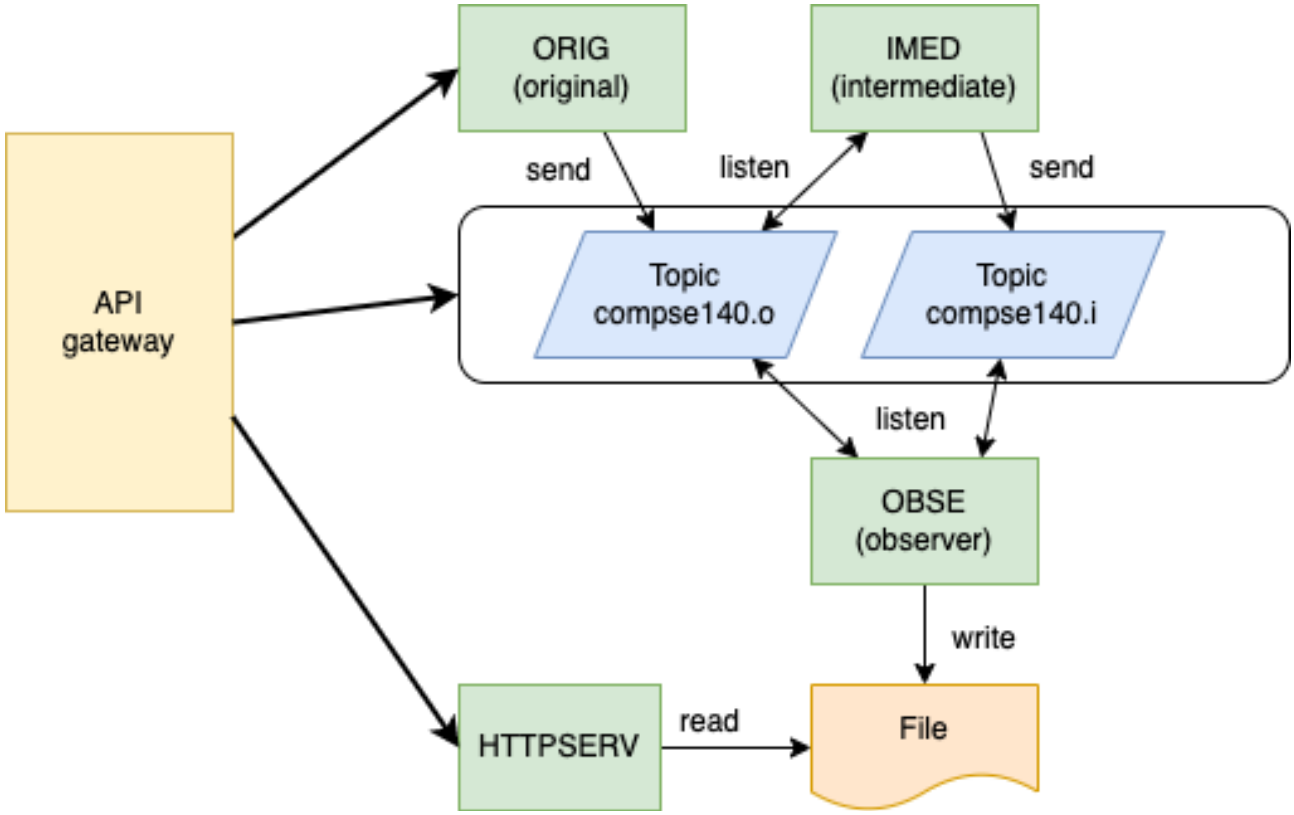

Figure 1.The target system.

Thus, the main phases of the project are:

1. Install the pipeline infrastructure using gitlab-ci. This means that you should:
    - install gitlab and runner. Register your runner to your own gitlab. If you have access to such gitlab where you can register your own runner, that can be used, too. Putting everything to local machine is just a work-a-round in cases where cloud capacity is not available.
    - Define the pipeline using gitlab-ci.yaml for the application you implemented for the message-queue exercise. The result of the pipeline should be a running system, so the containers should be started automatically. (In other words: "git push (to your own gitlab instance) => the system is up and running)
    - 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 cover the functionality shown to external world, i.e., the responses of the API gateway. *Optionally, the tests can cover some individual services, too.*
3. Implements changes to the system by using the pipeline. 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.
4. *(Optional) implement a static analysis step in the pipeline by using tools like jlint, pylint or SonarQube.*
5. Deploy the application at least to your own machine. The minimum requirement is to automatically run “docker-compose up”. *Optionally, deployment with Ansible (extension of the Ansible exercise) or deployment to an external cloud (Heroku or similar).*
6. *(Optional) implement monitoring and logging for troubleshooting. This should be a separate service that the user can use through browser. It should show at least start time of the service, number of requests it has received after start.*
7. Provide an end report (file "EndReport.pdf" in the root of the repo). Table of content in Appendix A.

Installing gitlab-ci

Do not underestimate the difficulty of this step. Reserve time for reading the documentation.

There are the following substeps.

1. Install your own gitlab. (https://docs.gitlab.com/ce/install/) The courses-gitlab cannot be used since it does not allow students to register their own gitlab runners. In case you do have spare virtual machine or server, a docker-based installation (https://docs.gitlab.com/ce/install/docker.html) is recommended. Note: there are two versions of gitlab ce (community edition) and ee (enterprise edition). I changed the above links to from “ee” to “ce”, but the ideas are the same. CE is the free version and thus recommended. One student also informed that it has better support on running docker-images. Both can be used.
2. Install your gitlab runner. (https://docs.gitlab.com/runner/ , https://docs.gitlab.com/runner/install/index.html) A docker-based approach (https://docs.gitlab.com/runner/install/docker.html) is recommended here, too.
3. Register your runner with your gitlab (https://docs.gitlab.com/runner/register/)

Notes:

- **Important**: the above instructions assume that you use 15.X version of Gitlab and runner. Also, you need to use mutually compatible versions.
- You may want a system that nicely restarts with docker-compose. For that you may try following these instructions: https://www.czerniga.it/2021/11/14/how-to-install-gitlab-using-docker-compose/ Course staff tried it. It can more or less followed but be careful, read and also understand carefully. Note that first start of Gitlab can be very slow.

The application and its new features

The starting point of the application is the docker-compose exercise (the four services + RabbitMQ) you already have.

Implement an API gateway service that provides the external interface to the system. This service should be exposed from port 8083. The API gateway should provide the following REST-like API

GET /messages (as text/plain)

Returns all message registered with OBSE-service. Assumed implementation forwards the request to HTTPSERV and returns the result.

Example response (part of):

```
2021-12-01T06:35:01.373Z Topic my.o: MSG_1
2021-12-01T06:35:01.473Z Topic my.i: Got MSG_1
2022-10-01T06:35:01.200Z 1 MSG_1 to compse140.i
2022-10-01T06:35:01.373Z 2 MSG_1 to compse140.o
```

PUT /state (payload “INIT”, “PAUSED”, “RUNNING”, “SHUTDOWN”)

PAUSED = ORIG service is not sending messages

RUNNING = ORIG service sends messages

If the new state is equal to previous nothing happens.

There are two special cases:

INIT = everything (except log information for /run-log and /messages) is in the initial state and ORIG starts sending again,

state is set to RUNNING

SHUTDOWN = all containers are stopped

GET /state (as text/plain)

get the value of state

GET /run-log (as text/plain)

Get information about state changes

Example response:

```
2020-11-01T06:35:01.373Z: INIT
2020-11-01T06.35:01.380Z: RUNNING
2020-11-01T06:40:01.373Z: PAUSED
2020-11-01T06:40:01.373Z: RUNNING
```

Note: your code should assume text/plan MIME-type for the above and application/json for the two following.

GET /node-statistic (optional) (in JSON)

Return core statistics (the five (5) most important in your mind) of the RabbitMQ. (For getting the information see https://www.rabbitmq.com/monitoring.html)

Output should syntactically correct and intuitive JSON. E.g:

```
{ “fd_used”: 5, …}
```

GET /queue-statistic (optional) (in JSON)

Return a JSON array per your queue. For each queue return “message delivery rate”, “messages publishing rate”, “messages delivered recently”, “message published lately”. (For getting the information see https://www.rabbitmq.com/monitoring.html)

Modify the ORIG service to send messages forever until pause paused or stopped.

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 options.

The intention is that you can test the pipe line by "git push". The easiest way to do that is a second "remote" for your repository. If the concept is new to you, see https://docs.github.com/en/get-started/getting-started-with-git/managing-remote-repositories

It is possible to use docker inside docker (so call dind) but note that there are some limitations. Some reading:

- https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/
- https://blog.nestybox.com/2019/09/14/dind.html
- https://blog.loof.fr/2018/01/to-dind-or-not-do-dind.html
- www.google.com

Returning the project

As previously, use course-gitlab and Plus. In Plus-service you only return the URL to the repository. Use git branch "project" for returning this!

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

- Compulsory parts work according to requirements	0..20 %
- Implementation of optional features (each optional feature is worth of 5%)	0..30 %
- 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.)

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.

Reminder: the optional features are:

- *implement a static analysis step in the pipeline by using tools like jlint, pylint or SonarQube.*
- *deployment to an external cloud (Ansible exercise, Heroku or similar)*
- *implement monitoring and logging for troubleshooting*
- *GET /node-statistic*
- *GET /queue-statistic*
- *Testing of individual components*

About assessment

To keep the required effort of assessment bearable, we assume

- The system (the extended application) can 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 running 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:8083/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 you use a browser in your own testing, you may fail. Use curl 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.)

Appendix A - template for the document

1. 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.

2. 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

3. Example runs of the pipeline

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

4. 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