COMP.SE.140 Project

Introduction

Version history

0.1	21.10 Kari	Initial version
1.0	25.10 Kari	Released to students
1.1	27.10 Kari	First corrections after lecture

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.

The project can be implemented either as pair work or as individual assignment. However, the first part needs to be done individually, and the pair need to report the work split within the pair. Some rules are specific to pair work and are they marked with "PAIR" in the following text.

Note: the students expected to read this document carefully.

The schedule

- The instructions disclosed: 25.10.2020
  - Students can start by installing the gitlab-ci
  - New versions to resolve ambiguous parts may be published later.
- Discussions in the lecture: 27.10.2020
  - Students are asked to give clarification questions
- Students that want implement as a PAIR inform Kari: 03.11.2020
- PAIR: simple pipelines returned in courses-gitlab: 13.11.2020
- Latest submission without reductions in grading: 03.12.2020
- Latest submission to pass the course: 10.12.2020

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 main phases of the project are:

1. Install the pipeline infrastructure using gitlab-ci. This means that you should:
   - install gitlab and runners on their own machine. A fresh virtual machine is recommended. Instructions to help in this process are below in section gitlab-ci.

    - 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 => the system is up and running)
    - Test the pipeline with the current version of the application.
    - PAIR: if you are a member of a pair you should return the pipe-lines before entering the next phase. So, this first phase is still individual work (do not return equal pipelines!) even for pairs. Returning to be done with git URL in plussa.
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.
    - Create test to the existing functionality of the application (see “Application and its new features” below)
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 mush 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. *Optionally, 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) with
    - Description of the CI/CD pipeline.
    - Instructions for examiner to test the system. Pay attention to optional features. This need to be in the README.md-file
    - Example runs (some kind of log) of both failing test and passing. The students need to show how the pipeline works both in case of success and failure.
    - Main learnings and worst difficulties (especially, if you think that something should have been done differently, describe it here)
    - Amount effort (hours) used
    - PAIR: description of the individual roles of both students.

Installing gitlab-ci

There are multiple ways and instructions on how to install gitlab-ci. Our recommendation is to use dockerized approach as described in https://gitlab.com/gitlab-org/gitlab-foss/issues/50851 . There you can find two files:

1. Docker-compose.yml to start gitlab and runner
2. Shell-script to register the runner with gitlab

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 8081. The API gateway should provide the following REST-like API

GET /messages

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

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 is in the initial state and ORIG starts sending again,
state is set to RUNNING
SHUTDOWN = all containers are stopped

GET /state

get the value of state

GET /run-log

Get information about state changes
Example output:

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

GET /node-statistic (optional)

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)

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.
~~Implement a copy of the OBSE process~~

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.

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

Returning the project

As previously, you can use public Github, Gitlab, or similar but you can also use course-gitlab. In Plus-service you only return the URL to the repository. Students who do the work individually, can use the existing repo. For pairs, who want to use course-gitlab a new repo will be created. Remember to ask it.

If you plan to continue with the project mark the version with tag and inform that tag to in returning.

PAIR: Since, the students need to use their own gitlab for building and testing but use a "public" git (github, courses-gitlab or similar) for returning and to proof that both have contributed, you should use two remotes: your own gitlab is one and the "public" is the other. Use of more than one remote should be familiar from "Programming 3", but for instance this link https://jigarius.com/blog/multiple-git-remote-repositories provides additional info.

PAIR: if you work as a pair: you should by 03.11.2020 23:59 return

- Information about your pair:
  student#; Last; First; email
  student#; Last; First; email
- If a new repo is needed

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 PAIR: ○ at least one optional feature needs to be implemented to reach 20% ○ pipelines are evaluated separately	0..20 %
- Implementation of optional features (each optional feature is worth of 5%)	0..25 %
- 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 with other sections, but the total sum is capped at 40%.