9.1 Project

1. Instructions for the teaching assistant

Implemented optional features

- Static code analysis step

Instructions for examiner to test the system

- Just a simple `docker-compose up` should work, however, the `--abort-on-container-exit` option should be used to have all containers stop upon sending a “SHUTDOWN” command to `/state`.

Host version information

- **OS**: Rocky Linux 8.7 (Green Obsidian)
- **Docker**: Docker version 20.10.22, build 3a2c30b
- **Docker Compose**: Docker Compose version v2.14.1

2. Description of the CI/CD pipeline

Platform

The GitLab, GitLab CI runner and the application itself run on the same Rocky Linux virtual machine. The virtual machine was provisioned with Terraform and bootstrapped with Ansible. The installation of GitLab was done by hand following the steps from the official documentation.

Although specified as not recommended in the documentation, the GitLab CI runner was installed and registered on the same host as the GitLab service itself. The runner runs in shell mode, which requires that compatible versions of Python and Docker Compose are installed on the host. Additionally, the deploy stage requires a specific path and permissions to execute properly.

Application

The base application, originally developed for the message queue exercise, was further developed to meet the project requirements. In addition to the exact features specified in the project assignment, some additional features were implemented to meet the requirements;

for example Redis was added to store the global state of the application as the existing message broker RabbitMQ was not found to be a suitable key-value store.

Version management

Commits were mainly pushed directly into the `project` branch. Only a few times were new branches created, primarily just to be something that can easily be discarded if a new idea didn't work out.

Testing

The new features were developed with the test-driven development paradigm. Pytest was used as the testing framework, as the microservices themselves were also developed in Python. A separate Docker Compose file was created exclusively for testing, meaning that in some cases, the tests go into integration testing territory.

The tests cover the API endpoints for the gateway service. Basics like return codes, content types and HTTP methods are covered. Some test cases also leverage the fact that the tests are run in Docker Compose, making it possible to for example send PUT requests to an endpoint and then check the Redis key-value store to verify whether the services are integrated correctly.

In addition to testing, flake8 was added as a static code analysis tool in the pipeline to enforce the PEP8 style guide.

Packaging

The applications were developed in Python – an interpreted language that doesn't require an explicit build stage. Therefore any packaging in this project is limited to builds of the Docker images.

Deployment

The deployment is essentially what was described as the minimum required to pass – a run of `docker compose up`, albeit with some caveats: the solution for the 'shutdown' command relies on a feature of Docker Compose used with the option `--abort-on-container-exit`.

As a way to cleanly exit all containers (like RabbitMQ or Nginx) was never found in the development of this project, this feature is used to gracefully shutdown the entire application whenever one of the microservices exits, instead of leaving some services idling while some exit as expected. This feature is, however, incompatible with another feature that was ideally going to be used in the deployment stage, namely the `--detach` option. This awkwardly leaves the `deploy-job` stage as the application's "parent process" and its `timeout`

argument as the lifetime. Ultimately, the choice between the options depends on what is seen as more important – the detached mode or correctly shutting down all containers.

Operation, monitoring

Operation is limited to Docker CLI commands. There are no monitoring features implemented.

3. Example runs of the pipeline

Snippets of failing and passing stages are included in the `EXAMPLES` directory in the project root. These are parts of verbose logs that have been filtered with `grep` etc. to show the general idea of what each case looks like.

4. Reflections

Main learnings and worst difficulties

In hindsight, there are some parts in the project that I think are well implemented, and some parts that were implemented poorly. In general, I am satisfied with the code base and especially the amount of reusable code it contains. The test stage is also something that I am satisfied with, as running tests entirely in Docker Compose wasn't something that I hadn't thought of before and was surprised with how well it worked. Having all services highly configurable with environment variables (and avoiding hard-coding anything) is a good part. Using Terraform and Ansible to provision the infrastructure as the very first step is also something I look back on positively.

Initially, integrating Redis as a solution to hold the service's state (which is a global element in the application, to be fair) was something I thought was a good addition. I had the general idea that a key-value storage was essential and researched whether RabbitMQ itself had any suitable features before choosing Redis. Only when implementing the very last features did I realize that I should have gone with a separate 'control' topic within RabbitMQ itself. The general problem was that many of the microservices are message driven – most of the application logic is run only upon receiving a message – meaning that polling the state from Redis would have required for example refactoring the application to run on multiple threads. Because of this I had to resort to some hacks in the last stages of development and ultimately view adding Redis as the wrong choice.

The deployment is another part where I am not satisfied with my solution. In short, I went with a Docker Compose deploy on localhost, which, coupled with the problem of using `--abort-on-container-exit` instead of `--detach`, resulted in a very hacky

deployment that barely fills the minimum requirements. This type of deployment also requires the host to have a specific directory structure in place, which I will mention in the appendix documenting the system requirements. The biggest contributing factors to this were the fact that I didn't want to pay for a second virtual machine for the deployment, and had previous bad experiences with Docker-in-Docker (dind) and didn't want to go that route either.

Amount effort (hours) used

By my estimates, the project took roughly 32 hours of work.

Appendix: system requirements

This documents what dependencies my system uses beyond the installation of GitLab + GitLab CI. I didn't see it as necessary to compile these into a separate Ansible playbook, but decided to include anyways for any possible debugging.

- Docker was installed following these steps: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-rocky-linux-8
- RPM packages installed with `dnf`:
  - `docker-compose-plugin`
  - `git`
  - `python39`
- The deployment directory at `/opt/deploy/project/` must exist and the `gitlab-runner` user must have necessary permissions (read-write-execute) there in order to successfully deploy.