COMP.SE.140 Project – Fall 2024 Joonatan J.

Instructions for the teaching assistant

No extra features were implemented, so only the basic features should be evaluated. There are / there were some parts of the implementation, that came close to some extra features. On development phase there was some logging on historical commits to identify the problem. Also, the testing covers probably more than was required, it basically ensures that the application works as expected.

System testing

System testing can be done easily. Git, docker and docker-compose are required. This guide is for Linux operating system, but with some tweaks it might also be runnable in Windows host.

1. `git clone https://github.com/ataajn/CICD-2024.git -b project`
2. `cd CICD-2024`
3. `sudo docker-compose up –build`

After steps above, the container should be working. The package contains also UI that is usable at http://localhost:8198/, but as requested, the main way to use the application is the command line, where using command curl, all the features are working.

Get requests (state, run-log and request) are working without any authentication. Put requests for changing the state do require basic authentication header. On default, those credentials are

`Username: ataajn`

`Password: skumnisse`

Example request to change the state to running:

```
curl -X PUT -u ataajn:skumnisse http://localhost:8198/state/ -d "RUNNING" -H
"Content-Type: text/plain" -H "Accept: text/plain"
```

Other way to test the system is to setup a Gitlab-runner and push code to the repository (or manually re-run the pipeline). You should put Gitlab-runner to be the executing worker for that project.

The system is designed in a way that whenever new code is pushed to the repository, gitlab-runner automatically builds, tests, and deploys it into production. In this basic case, the production means that Gitlab-runner runs on the machine it's setup on, but that machine can of course host the software into public internet.

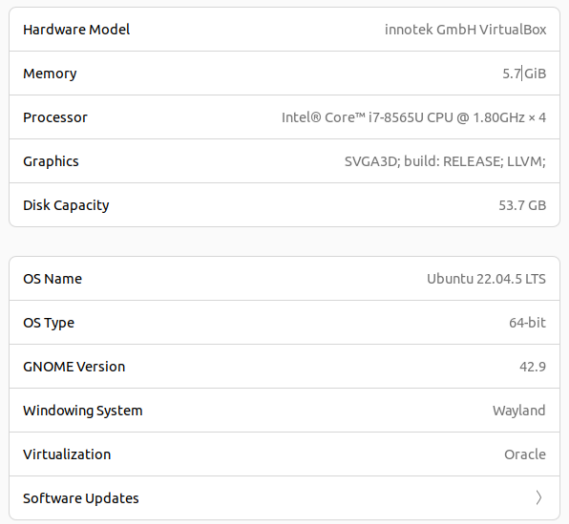
If used on Gitlab-runner, it is worth noting that you should run it in privileged mode and put socket into the volumes part. Gitlab-runner's config is found on default at path `/etc/gitlab-runner/config.toml`

```
volumes = ["/cache","/var/run/docker.sock:/var/run/docker.sock"]
```

```
privileged = true
```

Development system information

All the development was done inside Oracle VirtualBox virtual machine, using linux ubuntu. Specified information below:



Version of Docker was 27.4.1 and version of Docker-compose was 1.29.2 during development.

Description of the CI/CD pipeline

Application was developed to be run in a Gitlab CI, to automate every aspect of the development all the way to production.

Version management

Application was coded in a test-driven manner, which means that every feature of the system had a test implemented before it. That is why the version management history consists mostly of first the test implementation commits, and the code implementation commits after.

The code was always tested in the local machine before pushing it into git. For that reason, the commit history was clean even though mostly only the main branch was used. When establishing the pipeline using gitlab-ci.yml, another branch project_ci_testing was used, because it required a lot of commits, since it could not be tested on a local machine. After the pipeline was established, the branch was merged into main using –squash (as one commit). The branch project_ci_testing was naturally only pushed into Gitlab-remote, not GitHub, because it had nothing to do with GitHub.

Used tools and tests

The project uses pre-built images for nginx, python and node. Images are stored into Gitlab image registry, and that registry is used during Gitlab CI -runs.

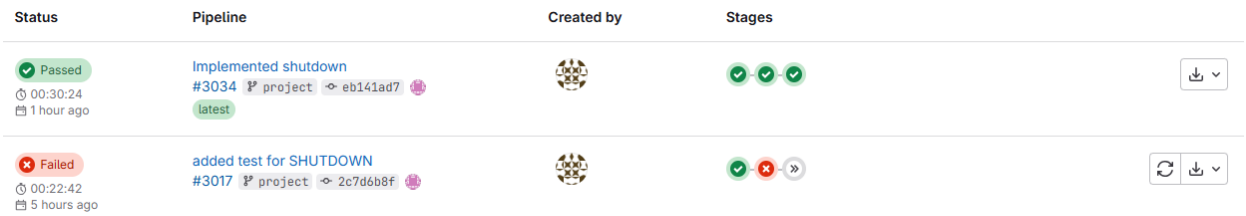
For tests, Node is used with Jest and Supertest. The tests test all the functionalities in the application, to ensure that the application behaves in a certain way after a certain input. The software contains 10 test cases, some of which test the same function but in a different state.

Deployment

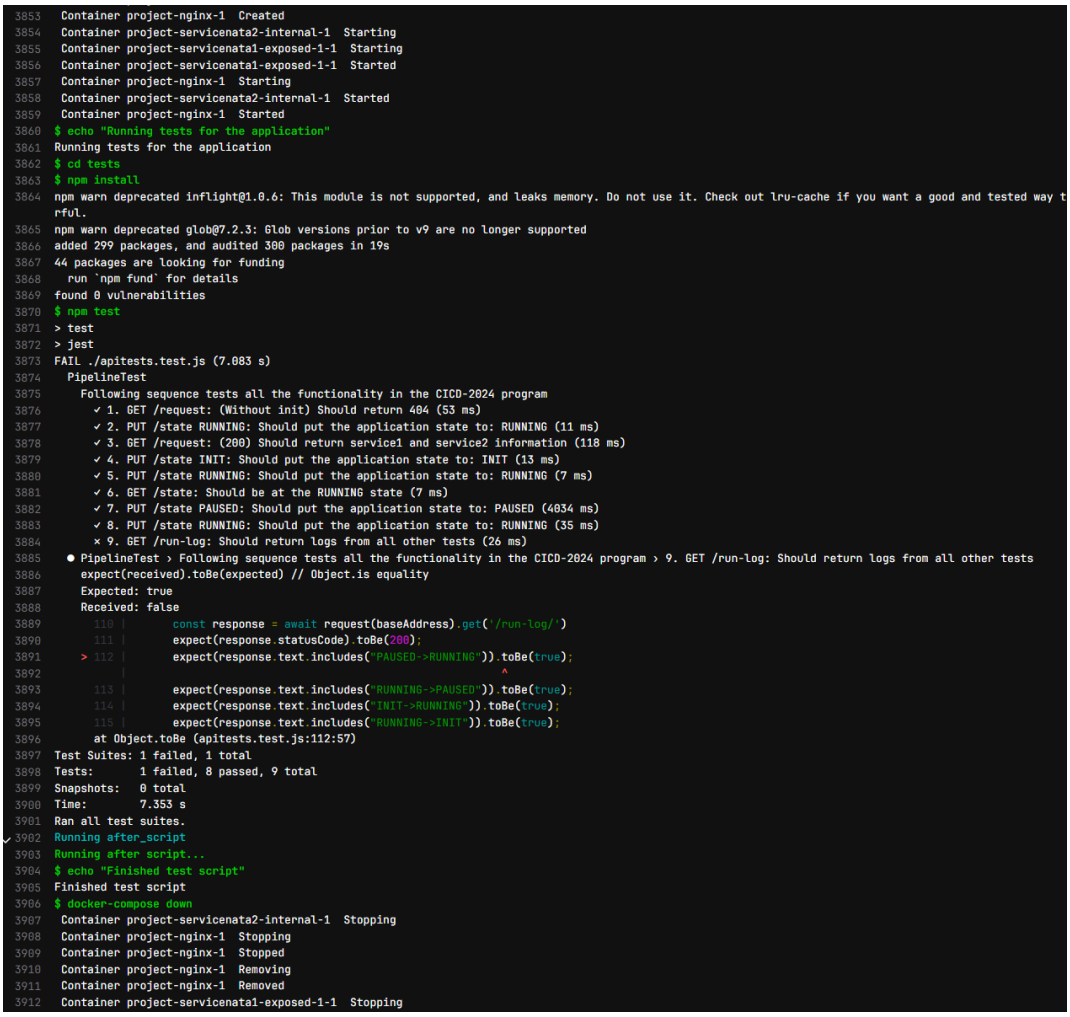
During most of the development the deploy phase was commented out, but in the end, every time a commit was made, and it passed all the tests, the application was automatically deployed to the test machine which I used to run Gitlab-runner on. The machine was the same virtual machine that was used for development.

Example runs of the pipeline

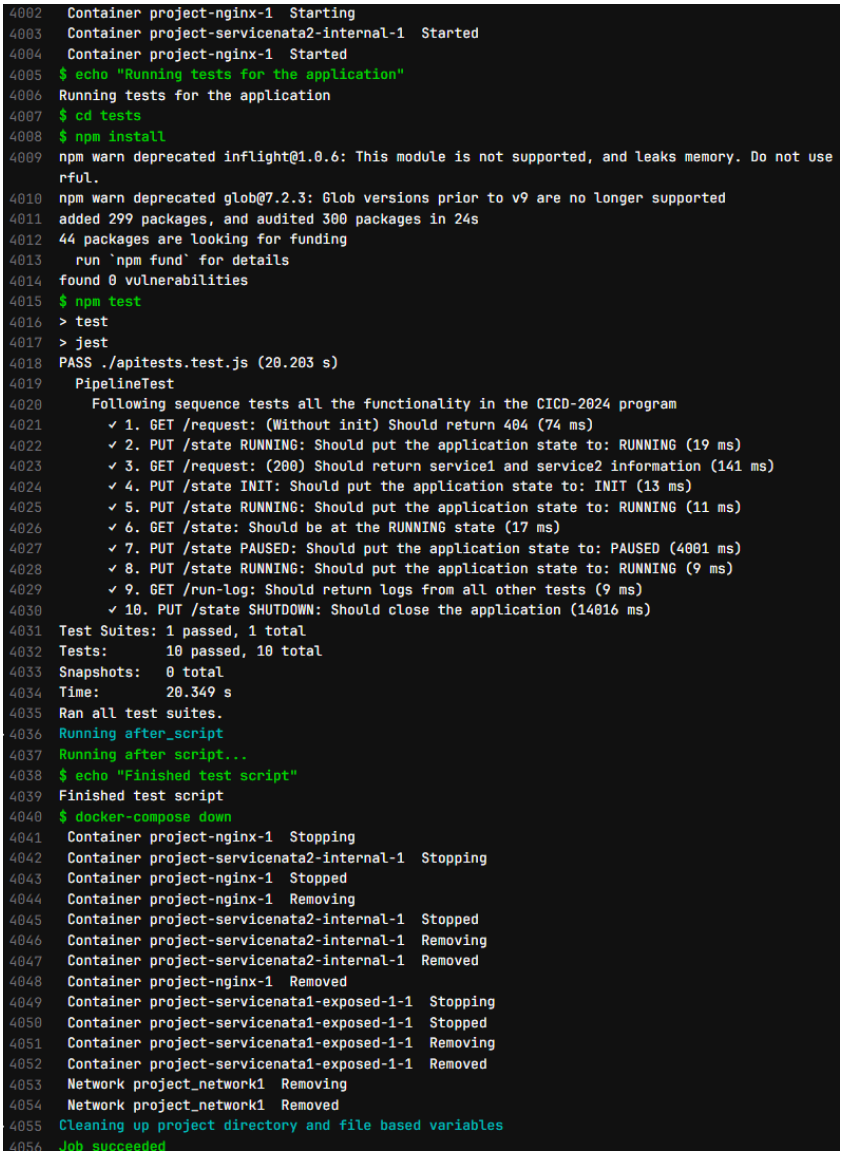
Below are included a few screenshots from the Gitlab, to showcase the pipeline and test architecture.



1 : Overview of a passing pipeline, and a not passing pipeline (test case fails)



2 : Screenshot from a failing run, since the test is not passed.



3 : Passing build and tests

Reflections

I had no prior experience on pipelines before this project. I learned a lot about automated deployment specifically. In the future, if I do personal projects that require deployment to an external server, I will most likely use automated deployment. It was hard to get one running, but when I got hold of things, it was convenient to use, so easy it was. Just push to the correct branch and the new version is in deployment and the new features are ready to be used.

If I had an actual case that I would really be using that software, I would have implemented the automatic release to an external server. I used to run a telegram bot on a server, and having these skills back then would have made things a lot easier.

The project was time-consuming for me. Most of the time went into debugging the Gitlab-runner, container registry and getting all the tools work. The virtual machine itself was one of the tools with the most problems, and when the machine ran out of disk space, I had problems with increasing it. Also Jest + Supertest and nginx took some time, since I was not familiar with those tools before this course. In the beginning I used a lot of time planning everything in the beginning, how everything should work technically, so that the number of surprises would be as low as possible. It is hard to say exactly how many hours were used during the project, but 50-80 would be a good, estimated range.

This project was clearly an obscure exercise, that had some requirements I would highly doubt ever using in an actual project. For example, the requirement that the application should not respond to any requests on “paused”. But I consider the requirements to be well suited to achieve the main objective of this course – learning.

There were two parts of the application I was not certain about – the first one being the “paused” functionality. I believe my implementation does fulfill the requirement for that, but it is far from perfect. I had another route in the mind as well – to find out if nginx supports some kind of request blocking when application is on a state – and to know which state the application would be on – I could use common volumes. Another possible solution would have been to shut some applications down and to run another instead, but that would cause more downtime I was willing to accept.

The other part not clearly stated in the project notes, was the authentication. In my mind there were 3 different alternatives, what the instructions could mean. I implemented authentication, that requires basic authentication for every request made that changes the state of the app. Alternatively, the authentication could work not from nginx but from python service. One option would have been to implement full token authentication on login, but in my opinion that was unlikely, since in the graph and example curl there were no tokens anywhere. The other option would have been to only authenticate a PUT requests that change the state from INIT to anything else. That is possible, but without any tokenlike information in the request, the server could not know who was sending the requests. But that would have been easy to implement – just put all the authentication to python server and check it if and only if the state change was to be INIT->something else.