

Instructions for the teaching assistant

Implemented optional features

I didn't implement optional features other than the browser for testing the endpoints, but it wasn't a bonus point just helped me to test the app.

Instructions for examiner to test the system.

To access the pipeline and testing branch of the code head to the course GitLab repo and check the 'project' branch:

<https://compse140.devops-gitlab.rd.tuni.fi/gremka/final-project-gitlab.git>

To access the code from my own GitHub used as the "Development" branch and access the 'project' branch:

<https://github.com/TheKaski/DevOpsCourse.git>

To test the docker application natively you will need to have python installed on your machine as well as docker compose of some sort. If you are running Windows docker desktop is required to run the application with docker-compose. Basically, to start the application you can run the command:

'docker-compose up --build'

in the repository root folder where the docker-compose.yaml is located. To run the tests open a second terminal where python commands can be executed and from there run the api tests by simply running

'python ./tests/api_tests.py'

If you are using Linux the command might be python3 depending on your installed python version. There are in total four test cases testing briefly each endpoint / feature. Feel free to check the code but it basically uses curl to interact with the apiGate nginx service directly. All tests should pass according to my experimenting with pipeline, Windows and Linux machines.

Additionally, if you want to view how the graphical system / UI works head on localhost:8198/ site to view the web page. The page will ask you to sign in go ahead and give the credentials 'admin' and 'secret' as a username and password. This should log you in and you should be able to do requests with simple buttons and see how the program handles them. There is a small side note though. Note that the PAUSED state only blocks the /request endpoint which there is a separate endpoint for receiving the text/plain form. The old endpoint of receiving system data still works as it was out of the scope of this exercise and was left as previously implemented.

The system has been implemented with nginx authentication cache and when you change the state from other states to INIT state you are still able to change the states on your window. This is because the browser stores the credentials until you close the browser so technically it logs you basically in immediately back and therefore you can change the state after changing it to INIT. However, if you want to test how the change to INIT state affects the program go ahead and toggle state to INIT then click updateState to verify that the state is toggled to init. Now close your browser and re-open the page. Now the browser will either ask credentials again or give you the page where you cannot change the state because the authentication cache has been cleared. This is a bit hacky, but the thing is that this problem is only with the browser as browsers cache the credentials and I was not able to clear the cache in runtime. So, with the curl requests this is not a problem because you will have to provide the credentials manually with the requests when they are needed and therefore, they are not cached similarly.

Platforms used for development:

- Windows 11 x86 PC, AMD Ryzen 2700X 8-core, 16Gb Ram, RTX 2060
 - Docker Compose version v2.24.6-desktop.1
 - Docker version 25.0.3, build 4debf41
 - Python 3.11.9
- Debian 12 VM run inside Oracle VirtualBox, used for testing and running gitlab-runner
 - docker-compose version 1.29.2, build unknown
 - Docker version 20.10.24+dfsg1, build 297e128
 - Python 3.11.2
- gitlab-runner:
 - Version: 17.6.0
 - Git revision: 374d34fd
 - Git branch: 17-6-stable
 - GO version: go1.22.7
 - Built: 2024-11-20T09:30:36+0000
 - OS/Arch: linux/amd64
 - Both docker-executor and shell-executors were tested and finally ended with the shell option

Description of the CI/CD pipeline

The final version of the pipeline uses shell-executor of GitLab-runner for running the jobs inside the machine the runner is located in. The pipeline has three stages build, test and deploy all executed sequentially.

The build stage uses docker compose down --volumes to clear the old logs and information left from previous runs. This works for this project but with continuous deployment other












strategies might want to be considered as this clears all volumes. However, for this project works well enough. Project build is executed with 'docker compose build' command.

The test job startswith before_script where the project is started in detached mode with 'docker compose up -d'. Once executed the state of services is verified with 'docker ps' command which gives insight print to the pipeline run in gitlab. After this the DOCKER_HOST_IP environment variable is stored and exported to make sure the tests have access to the services inside the pipeline (Check struggles and lessons learned for more info). Once done a curl request to the DOCKER_HOST_IP:8198 is done to make sure services truly are accessible. The script then uses python3 in this case to run api_tests.py script to test different features of the system. After tests the data is cleared and deploy build is executed.










After successful test run the application gets deployed with docker compose up -d again. This means that the system is now available as a local service on the machine where located and can be accessed also with the machines IP address in local network.

Example runs of the pipeline










Here is the most recent commit at the moment which passes all the stages

Status	Pipeline	Created by	Stages	
 Passed ⌚ 00:00:45 📅 3 minutes ago	Clean after testing #1036  project  f623ae62  		  	 

Here is a log of pausing docker runner and changing to other

 Failed 📅 1 day ago	Paused the runaaja and moved to shel... #906  project  1ecd2be   		 
---	--	---	---

Here is a log of failing test job:

 Failed ⌚ 00:00:11 📅 1 day ago	Added PUT authentication and reset o... #928  project  6287a6a3 		   
---	--	---	---

Reflections

The WORST difficulties

The project did not get the best start from the beginning with the process of altering the pipeline. Setting up GitLab-runner was familiar for me, and I started work early as I had previous experiences working with self-hosted GitLab server and own GitLab pipeline runner. So, I knew from the start that problems are to be expected, how ever not that they would take so much time debugging.

Biggest problem at the start was to get the correct permissions for the runner to run basic Linux commands in the shell when setting up the pipeline for build step. There were occasions when the pipeline ran and faced errors, but magically would go through when rerunning again from web UI which gave frustrations to me. I quickly moved to shell executor after first trying to use docker in docker approach with the docker executor. This allowed me to move to the next step which was the testing phase.

In this part I had the worst struggles as I could not get the tests to access my containers running on the pipeline. All tests on localhost and VM natively showed good progress however the connection was not working inside the pipeline. At this point I even contacted Kari about the problem but with no luck. First, I thought it must have been the problem with docker inside the shell, so I moved back to docker-executor but that didn't solve the problem so again back to shell. At one point I got responses from localhost route / but note for example route /state which was weird for me and didn't know it by then but later faced the same problem again but with PUT requests not being supported inside the pipeline

As there were problems with the connection I reverted back again to using docker-executor and this time got the idea to modify the config.toml to include the docker host volume in order to make the services accessible within the pipeline, this would have to work. This approach magically worked, and tests were passing kind of. I started development work but soon realized, that PUT requests were not supported in the pipeline but worked again on both VM and Windows when running natively with docker. I showed my problem to a course friend, and we could not understand what was wrong with the pipeline again. At this point my GitLab version history was Full of fixes of fixes and new fixes, all broken. And somewhere down all

those commits were my code halfway implemented halfway functional I just needed the pipeline to understand that.

This point, there was nothing else to try but AGAIN try the shell as everything worked natively on the VM why the shell work wouldn't. This time by accessing the docker-host within the shell executor I was finally able to access the endpoints of services within the pipeline. Big win for me. However, PUT requests were still unsupported, why? I tried everything from modifying the nginx and API gateway configurations, but nothing worked for me. The message wasn't even printed from nginx it was the service 1 printing the errors. I changed python versioning of the containers to match my local system but that didn't work. Then magically, my course friend told me that he uses the command 'docker compose' instead of 'docker-compose' which I had been using on pipeline, and my local development platforms. This changed everything. Suddenly all requests were working and after nearly 200 commits I finally had the opportunity to use the pipeline properly. Finally green flags from build to deployment were happening and the project took a new turn. All because of one '-' character.

Main lessons learned

I learned a lot about docker commands and networking with docker when trying to reach services. I also extended my knowledge from GitLab-runner with trying to use the docker-executor and faced some problems with accessing the dockers host. I learned hands on working with the pipeline and how functional pipeline can deliver continuously new features to deployment environment. I also had to solve problems such as syncing the three copies of service 1 to all share a common state. It was the first time I used custom volumes between the containers for such purposes.

Amount effort (hours) used

50 hours multiple weeks of effort was put into debugging, and I think that is clear from the commit history. The unfortunate thing of the commit history also is that the TDD approach does not come through other than from the end of the project where the pipeline is finally working. This is because I had to implement the features despite problems with pipeline to be able to have at least something to submit for this exercise. Although some of the commit

messages do indicate that tests were otherwise working. Also TDD approach was used when pushing only to “Development” repo when the tests were passed finally.