COMP.SE.140 DevOps final project report – Fall 2024

Student: Ari Kukkaro

Email: [ari.kukkaro@tuni.fi](mailto:ari.kukkaro@tuni.fi)

Student number: 239944

# Instructions for running the system

System has two backends service1 and service2 they are located in srv1 and srv2 folders accordingly. Service1 is written in Python and service2 is written in C#. Then there is nginx folder that contains nginx for routing requests and balancing loads. Also nginx folder includes frontend that is written in React. **HTML for web page** is located on nginx/public and it **renders ‘root’** that uses files in nginx/src folder. F**ile index.js injects root for HTML** and renders **App.js that has frontend buttons and logic**. Then nginc.conf file uses load\_module/etc/nginx/modules/ngx\_http\_js\_module.so; imports java script module for nginx.conf file so that js\_import /states.js; file functions can be executed when nginx received certain requests. Then dictionary js\_shared\_dict\_zone zone=my\_zone:50m; saves persistent data of the system in dictionary. There are three key-value pairs that my\_zone dict stores. These are **currentState, PreviousState** and cumulating **run-log** of the system. Then in */states.js* file there are function *handleStates(r)* that handles **GET /state** and **PUT /state**. I that function if INIT is not set it is set. Then Get /state reponses with current state and PUT /state handles also StateTransition() which logs state transition for **GET /run-log.** Then *GetRunLog(r) function* returns run-log response. **Note** that putting states is possible with **port 8197**, however **system itself does not** unfortunately **correspond the states.** That was one feature that caused hardships to me. Next chapter summarized different endpoints of the system and gives instruction to test te system.

# Instructions for testing the system

Gitlab repository was used for mostly creating CI/CD pipeline, but it also contains necessary files of the system. Changes to Github repository with branch “project” should be mirrored to Gitlab repository branch “project”. Initially clone repository with command ***git clone -b project*** [***https://github.com/GiantSteps44/DevOps-course.git***](https://github.com/GiantSteps44/DevOps-course.git)*then* ***cd DevOps-course*** *and* ***docker compose up –build.***

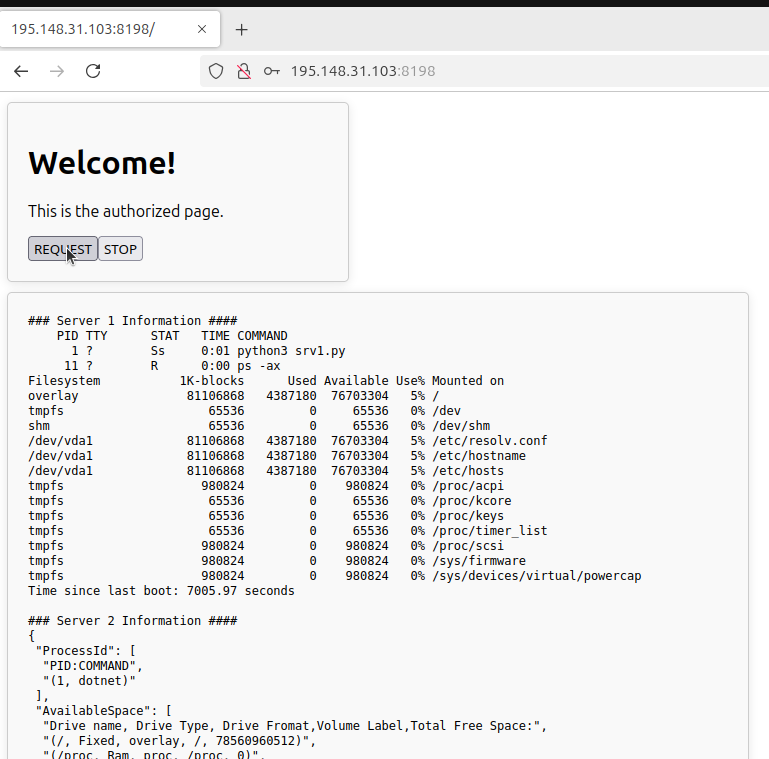
*(How ever if this would not work for some reason back up is to execute* ***git clone -b main*** [***https://compse140.devops-gitlab.rd.tuni.fi/AriKu/devops.git***](https://compse140.devops-gitlab.rd.tuni.fi/AriKu/devops.git) *this repo branch has production code, whereas project branches of both repos were designed to have code under development).*

**System features on port 8198:**

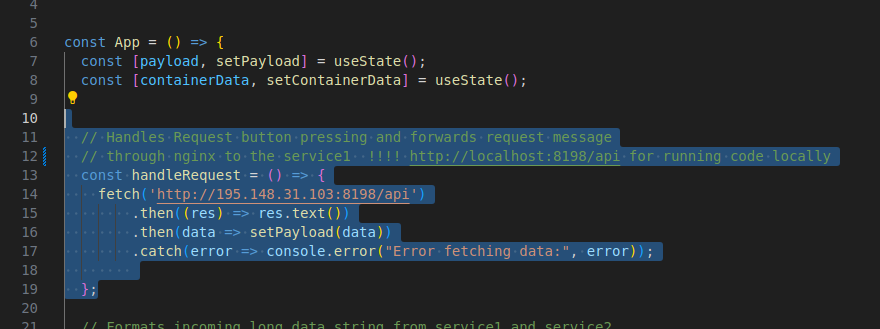
* **GET /** endpoint: Usage via web browser**.** First login page emerges then, data page is shown.

Software is deployed to CSC cPouta cloud. System can be used with [http://195.148.31.103:8198](http://195.148.31.103:8198/)/

from cPouta cloud and login.txt file contains credentials. Data page showed in Figure 1 below.

**Figure 1: cPouta data page after login.**

**If this endpoint is tested locally make sure that App.js file** has fetch(‘<http://localhost:8198/api>’) this function forwards REQUEST button signal to nginx and routes that to backend upstream. App.js fetch shown in the Figure 2 below.

**Figure 2: App.js REQUEST button signal forwarding.**

**API Gateway features on port 8197 (either on cPouta or locally):**

* **GET /state** endpoint: Provides current state of the system
* **PUT /state** endpoint: Puts new state for the system and logs state transition
* **GET /request** endpoint: provides similar data than GUI as plain text from srv1 and srv2.

Notification system in port 8198 does not correspond PUT states, due to implementation difficulties with **js\_content** that can be executed once per request.

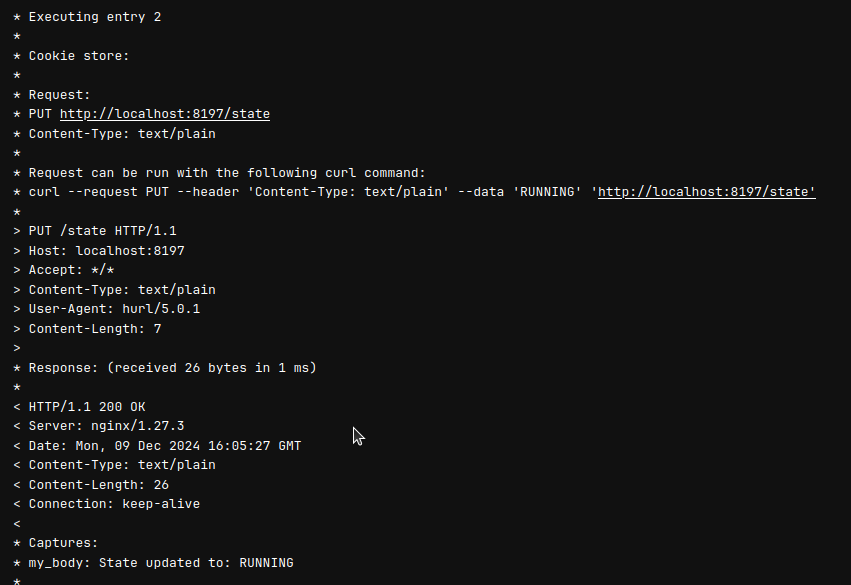
* **GET /run-log** endpoint: reponses with accumulated state transitions and time when state transitions have happened as log.

**Additional Testing endpoint features on port 8197 (either on cPouta or locally):**

* **GET /login** endpoint: testing that unauthorized text if not logged in
* **GET /back2** endpoint: testing that only srv2 responses with data to request
* **GET /** endpoint: for testing that HTML page as text is reponsed.

Automated API test files are located in /tests folder these include basic API endpoint testing for each endpoint above. Automated test are runned as part of CI pipeline in gitlab when code is pushed to remote’s named <https://compse140.devops-gitlab.rd.tuni.fi/AriKu/devops.git> branch project. In there **Hurl** API testing container is pulled and it executes test.sh file that runs each different API test .hurl file. If you want to try hurl out locally maybe the easiest way is to install hurl according to  
<https://hurl.dev/docs/installation.html> . After installing hurl locally in the project root folder **execute**  docker compose -up (-d) and then with another terminal *cd ~/tests/*

and run hurl --test \*.hurl or hurl -v --test \*.hurl for verbose test results. Testing results should look similar to the CI pipeline Figure 3 below.

**Figure 3: Hurl PUT /state “RUNNING” API test. (Image from gitlab)**

In next subchapter there presented development platform and its docker and docker compose versions. Then there is also local testing platform presented and which docker and docker compose versions were used with that one.

## Platform used in the development:

**Development plaform:**

*VM Hypervisor: VMWare workstation 17.6.1 Pro*

*Distributor OS ID: Ubuntu*

*Description: Ubuntu 24.04.1 LTS*

*Release: 24.04*

*Codename: noble*

*CPU architecture: amd64*

*Cores: 2 cores (Intel i7-10850H)*

*RAM: Allocated 8GB*

**Docker & Docker compose:**

*Container Platform: Docker*

*Release: version 27.3.1, build ce12230*

*Container orchestrator: Docker compose*

*Release: version v2.29.7*

**Testing plaform:**

*VM Hypervisor: VMWare workstation 17.6.1 Pro*

*Distributor OS ID: Kali Linux*

*Description: Kali 23.2*

*Release: 23.2*

*CPU architecture: amd64*

*Cores: 2 cores (Intel i7-10850H)*

*RAM: Allocated 4GB*

**Docker & Docker compose:**

*Container Platform: Docker*

*Release: version 26.1.5+dsfg1*

*Container orchestrator: Docker compose*

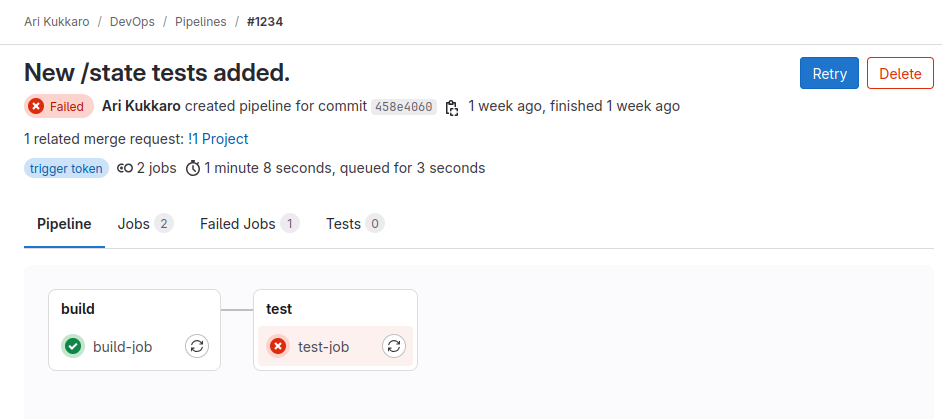
*Release: version v2.20.2*

# The description of the CI/CD pipeline

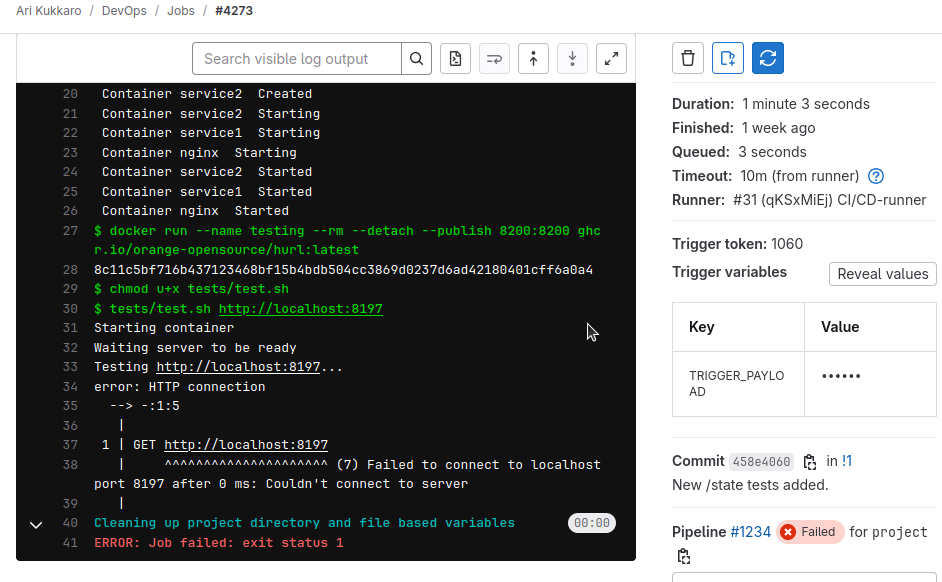
Assignment instruction was to use two different repos one in GitHub <https://github.com/GiantSteps44/DevOps-course/tree/project> and another in Gitlab: <https://compse140.devops-gitlab.rd.tuni.fi/AriKu/devops/-/tree/main?ref_type=heads> in assignment. Github repo branch “project” and Gitlab repository branch “project” contains development side files for this assignment. Idea was to try mirror Github project files to Gitlab project branch when new files are pushed to Github. And this worked occationally. Therefore direct pushes to Gitlab project was done manually also. Github project runner was installed and registered locally as illustrated in Figure 4.

**Figure 4: local gitlab-runner .toml file (token omitted)**

Then idea was to push deployment ready code to Gitlab main branch separately. When code is pushed to Gitlab project repo initially gitlab passwords are used to login docker and enable gitlab to download most recent docker images from dockerhub and after that docker compose pulls and builds nginx (containing frontend), service1 (three instances), service2 images. After that in testing phase docker compose runs software containers and also pull Hurl image from dockerhub and runs automated tests from test.sh file against the software and then docker compose down –remove-orphans is executed making sure that testing pipeline can be run again without raising locally error that there is already, for example container named nginx. Figure 5 illustrates Build and Test pipelines.

**Figure 5: Simple CI pipeline**

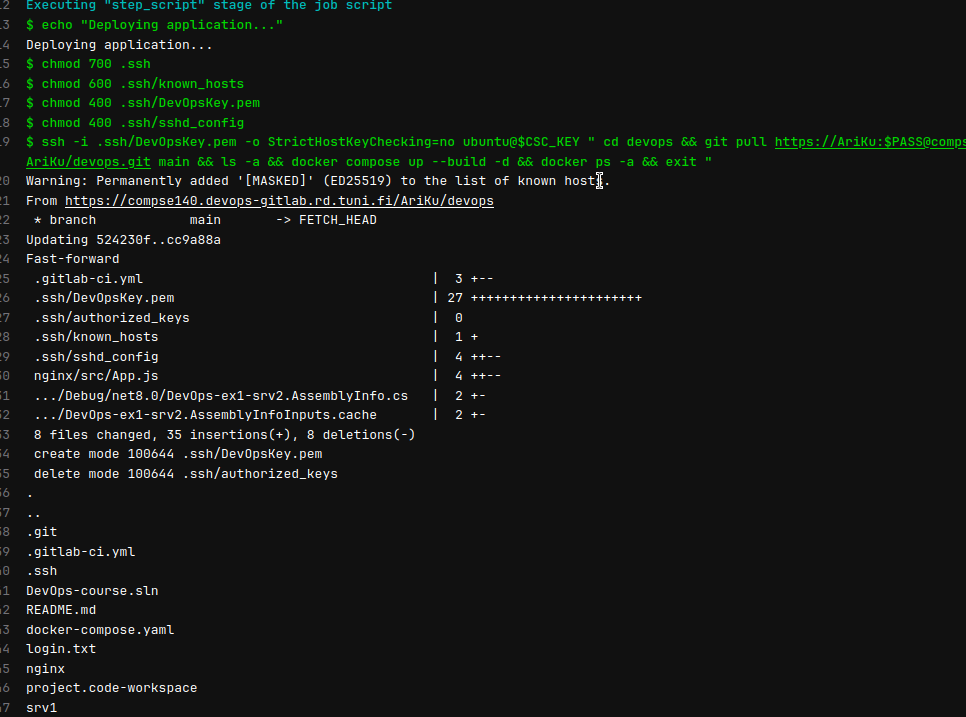
When code is pushed to Gitlab project repo automated Hurl test are performed automatically such very simple CI pipeline is illustrated in Figure 4. Then hurl test was performed similarly than in Figure 3 and 6, in latter of which /state tests file where added before the /state endpoint. This can be found in the Gitlab and from drop down menu “pipeline” then navigate page 8.

**Figure 6: CI test for /state endpoint**

Intention was to follow **Test driven development** principle as much as possible in implementing simple tests first then coding simple code that would first fail and then to be able to pass the automated test and this was done incrementally feature by feature.

Packing was retrospectively thinking already done in build phase. There srv1, srv2, nginx and frontend was already packed into the distinct docker images which where runned as containers after that for tests. What I could have done there could be proper build phase where for example C# code is built into the executable. Or in this packing phase dockerfiles could be packed as one docker image and pushed to docker registry where it can be pulled in deployment phase. For the simplicity just docker compose was used for the deployment.

For the deployment CSC cPouta Ubuntu VM instance was allocated and the configurations, ssh connection was established, but also HTTP for Software GET requests was established. After many and many failed deployments. Deployment to cPouta was done succesfully by first cloning Github main branch to cPouta VM and then using ssh connection pulling updated files to the cPouta devops/ folder as depict in the Figure 7.

**Figure 7: CD pipeline in action**

Continuous monitoring pipeline was not implemented to monitor the software.

# Reflection on the assignment and lessons learned

Main learnings of the assignment were many. First of all I learned to use Gitlab CI/CD and configure gitlab project runners locally. Then also I learned how to use nginx and extend its functionalities through nginx java scripts module. Third I learned how to do test driven development and implement simple API tests. However, automated API tests were rather simple and I could learn more about how to do proper software testing. If it is possible, I was planning to participate software testing course later. Then I learned how to assign and configure CSC virtual machine in the Cloud. Then of course how to perform Continuous deployment on that Cloud platform. Quite, frankly I learned a lot, but still there is much to learn within this context of DevOps.

Most difficult part of the assignment for me was creating system states with the nginx and nginx java script module. I did not manage to link the states with the actual system running in the port 8198. Most difficult part was that when **js\_content** which forwards for example GET request to states.js function as function name(r), such function could be executed once per API call. There was trouble how to route such requests between different nginx location endpoints. For example, if PUT /state “PAUSED” was asked how to prevent the main system to response requests during PAUSED state.

Then another hard part was how to implement proper tests of the system. When using docker containers it seemed quite natural to make sure with the automated tests that published ports work as well as communication between different containers work as such. However, testing could have tested different services first without putting them to containers and just then packing these services to container and after that another testing that containers can communicate in the networks. That could have been more accordingly the DevOps phases that lecture slides suggested from development→version management→build→test→pack→deploy. All in all the course has been difficult as mentioned at the beginning of the course, but at the same time it has been very useful and super interesting to learn such things.

Estimated time effort for this Assignment was about ~70-80 hours.