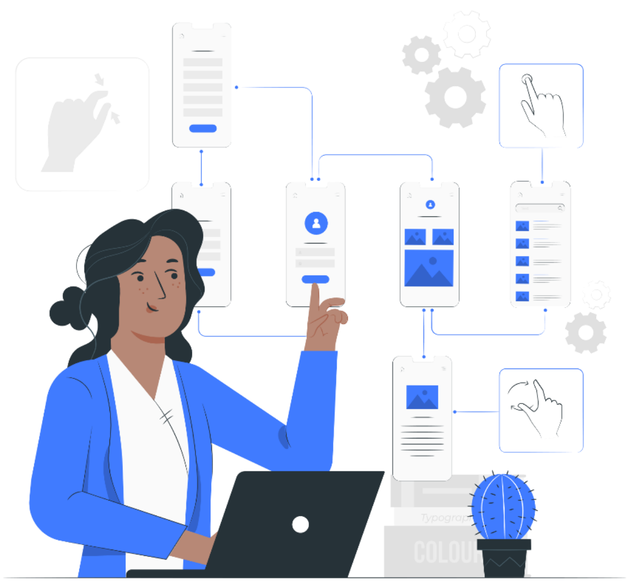
DevOps Project



Contents

1. Instructions for the teaching assistant

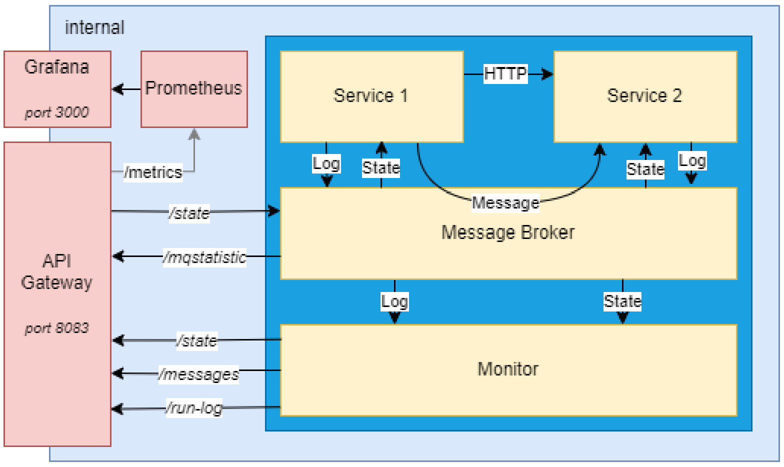


Figure 1 - Overview of the final system

Implemented optional features

Implement a static analysis step in the pipeline by using tools like jlint, pylint or SonarQube.

I chose to use SonarQube by integrating it with GitLab via their official docker image. The explanation to replicate my configuration are included in "./gitlab/gitlab_setup.md" along with the gitlab setup instructions.

The pipeline can be run with or without it being set up, as it checks for the presence of the Sonar related variables in the GitLab configuration.

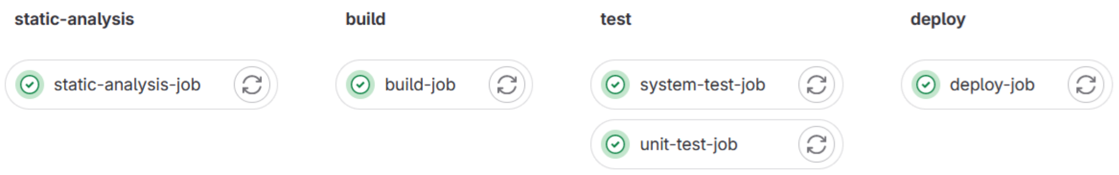
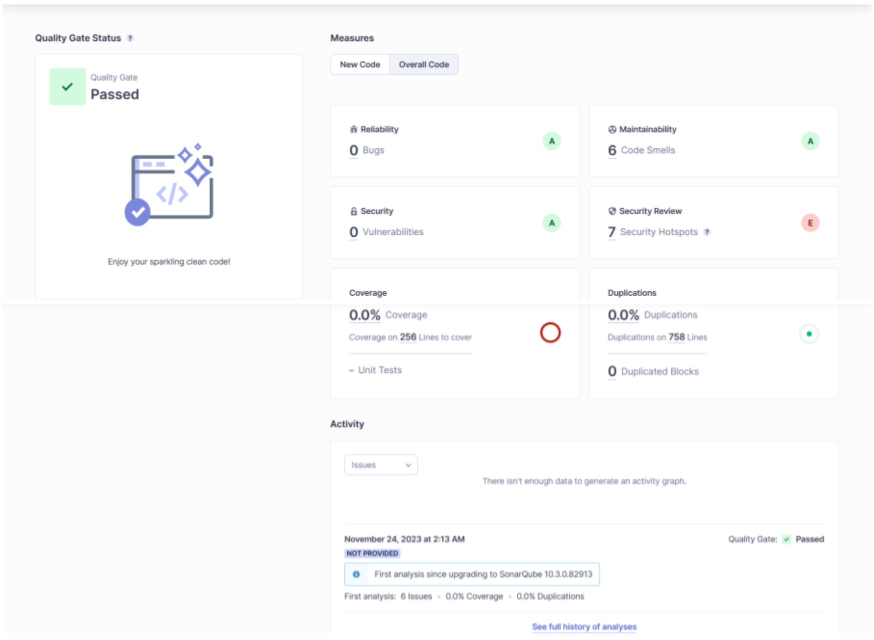


Figure 2 - Pipeline run with static-analysis stage



Quality Gate Status

Quality Gate
Passed

Enjoy your sparkling clean code!

Measures

New Code | Overall Code

Reliability
0 Bugs — A

Maintainability
6 Code Smells — A

Security
0 Vulnerabilities — A

Security Review
7 Security Hotspots — E

Coverage
0.0% Coverage
Coverage on 256 Lines to cover
– Unit Tests

Duplications
0.0% Duplications
Duplications on 758 Lines
0 Duplicated Blocks

Activity

Issues

There isn't enough data to generate an activity graph.

November 24, 2023 at 2:13 AM — Quality Gate: ✓ Passed
NOT PROVIDED
First analysis since upgrading to SonarQube 10.3.0.82913
First analysis: 6 Issues · 0.0% Coverage · 0.0% Duplications

See full history of analyses

Figure 3 - Resulting SonarQube page

I set it up so that the pipeline completes even if the static-analysis fails (some criterias aren't met) as it is only used as recommendation in this project. In bigger project, it would be better to set it up so that it fails if certain criterias aren't right, like too much security issues (or high severity), poor code coverage, …

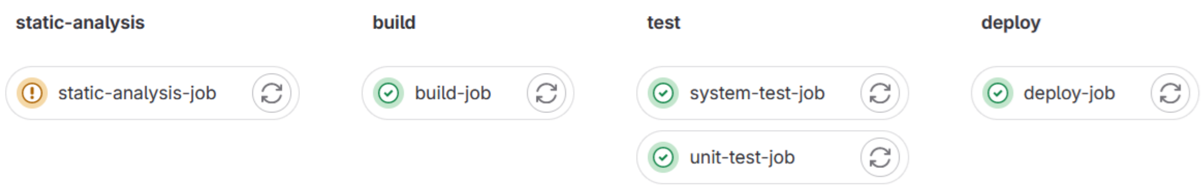


Figure 4 - Pipeline run with failed static-analysis stage

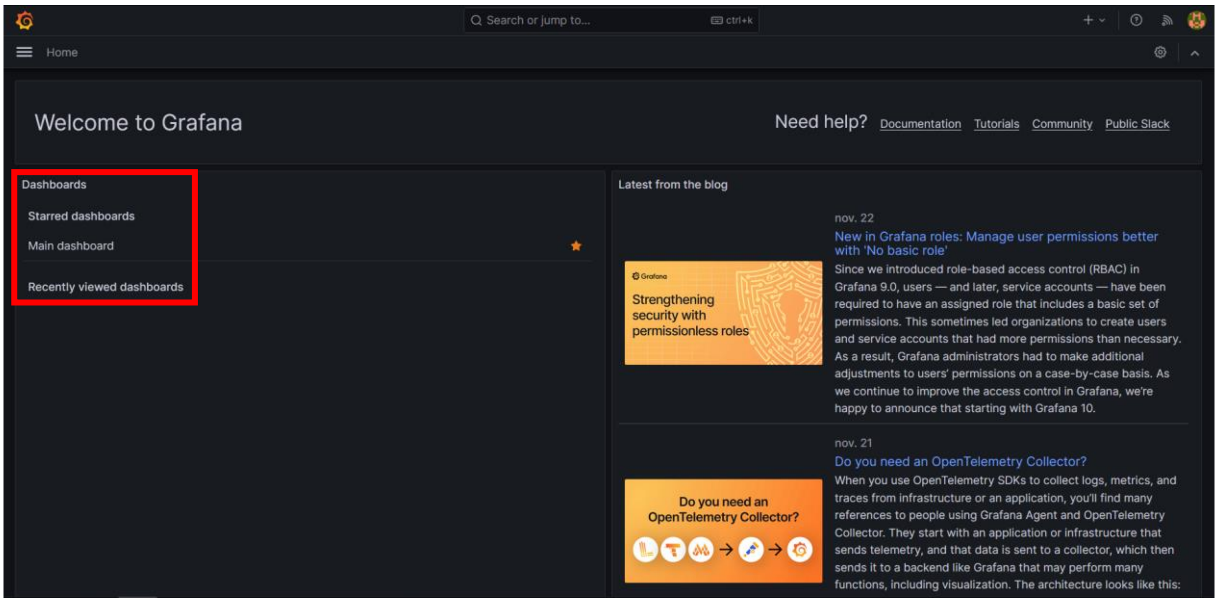
~~Deployment to an external cloud (Ansible exercise, Heroku or similar)~~

I did not implement this step as I don't have any service available to me currently, however, I believe it would be quite simple to implement using the following steps for the deploy job:

- Publish the images to a repository (Docker Hub would be the simplest one) during the build step, using "docker compose publish" for example,
- Connect to the remote machine via SSH, either directly during the deploy job or via ansible,
- Make sure docker is installed,
- Copy the "docker-compose.yaml" which uses the "latest" version of the images,
- Run the "docker compose up -d" command.

Implement monitoring and logging for troubleshooting

For monitoring and logging, I chose to use Prometheus and Grafana. The created Grafana dashboard is available in the browser via **port 3000**. Login is "admin" and password is "password". To access the dashboard you can either use this link or from the main page go to "Dashboard > Main Dashboard" at the bottom.



nov. 22
New in Grafana roles: Manage user permissions better with 'No basic role'
Since we introduced role-based access control (RBAC) in Grafana 9.0, users — and later, service accounts — have been required to have an assigned role that includes a basic set of permissions. This sometimes led organizations to create users and service accounts that had more permissions than necessary. As a result, Grafana administrators had to make additional adjustments to users' permissions on a case-by-case basis. As we continue to improve the access control in Grafana, we're happy to announce that starting with Grafana 10.

nov. 21
Do you need an OpenTelemetry Collector?
When you use OpenTelemetry SDKs to collect logs, metrics, and traces from infrastructure or an application, you'll find many references to people using Grafana Agent and OpenTelemetry Collector. They start with an application or infrastructure that sends telemetry, and that data is sent to a collector, which then sends it to a backend like Grafana that may perform many functions, including visualization. The architecture looks like this:

Figure 5 - Grafana main page with link to dashboard (red box)

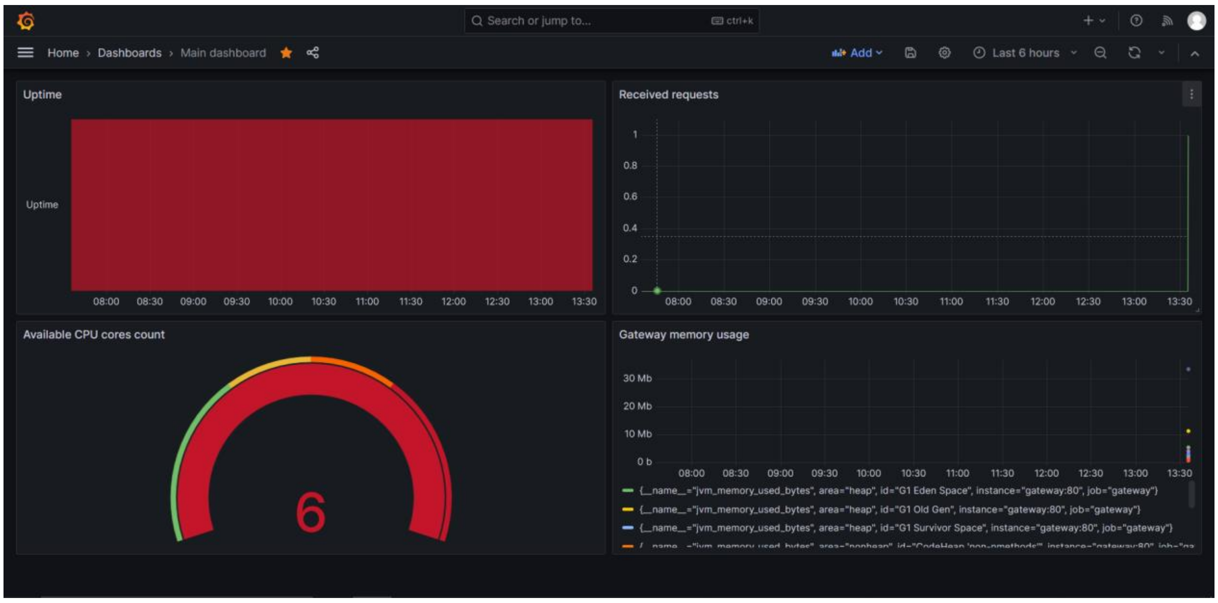

Search or jump to...
ctrl+k
Home › Dashboards › Main dashboard
Add
Last 6 hours
Uptime
Uptime
08:00 08:30 09:00 09:30 10:00 10:30 11:00 11:30 12:00 12:30 13:00 13:30
Received requests
1 0.8 0.6 0.4 0.2 0
08:00 08:30 09:00 09:30 10:00 10:30 11:00 11:30 12:00 12:30 13:00 13:30
Available CPU cores count
6
Gateway memory usage
30 Mb 20 Mb 10 Mb 0 b
08:00 08:30 09:00 09:30 10:00 10:30 11:00 11:30 12:00 12:30 13:00 13:30
{__name__="jvm_memory_used_bytes", area="heap", id="G1 Eden Space", instance="gateway:80", job="gateway"}
{__name__="jvm_memory_used_bytes", area="heap", id="G1 Old Gen", instance="gateway:80", job="gateway"}
{__name__="jvm_memory_used_bytes", area="heap", id="G1 Survivor Space", instance="gateway:80", job="gateway"}

Figure 6 - Grafana dashboard

The information available here is the uptime (red for offline, green for online), the current quantity of received requests, the count of CPU cores available to the system and the memory usage of the gateway service. By default, it shows the data from the last 6 hours, but you can adjust it on the upper-right.

GET /mqstatistic

The core overall statistics I chose are:

- The total number of connections
- The total number of consumers
- The total number of unread messages
- The total number of recently published messages
- The overall message publish rate

One thing to note for the queues statistics is that queues that are bound to a fanout exchange don’t have a “message_stats” attribute. For the attributes of those queues, I chose to print “-1” instead.

Testing of individual components

I have implemented unit tests for the “service2”, “monitor” and “gateway” services. I did not implement any for “service1” as I don’t know the Rust libraries well enough to mock what is needed and the research needed would be quite time-consuming.

Instructions for examiner to test the system

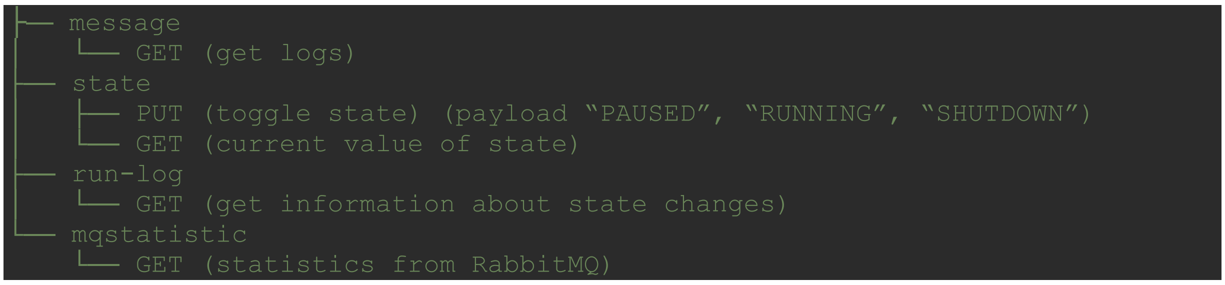
The system can be tested simply via a “docker-compose up -d” command using the “project” branch of the gitlab repository.

```
$ git clone -b project https://course-gitlab.tuni.fi/comp.se.140-fall2023_2023-2024/rnvako.git
$ cd ./rnvako
$ docker-compose build –-no-cache
$ docker-compose up -d
```

All routes can be tested with tools such as curl or Postman with command such as:

```
$ curl localhost:8083/state -X PUT -d ”PAUSED” \
-H ”Content-Type: text/plain” \
-H ”Accept: text/plain”
```

Here is a remainder of the routes that are available:



```
├── message
│   └── GET (get logs)
├── state
│   ├── PUT (toggle state) (payload “PAUSED”, “RUNNING”, “SHUTDOWN”)
│   └── GET (current value of state)
├── run-log
│   └── GET (get information about state changes)
└── mqstatistic
    └── GET (statistics from RabbitMQ)
```

To test the pipeline, please see the “/gitlab” directory in the repository. It contains a containerized GitLab setup along with the instructions (in “./gitlab/gitlab_setup.md”) to set it up on any machine, only requiring docker to be installed.

Testing SonarQube requires the local GitLab setup. As said before instructions to reproduce it are available in “./gitlab/gitlab_setup.md”.

The logging/monitoring service can be accessed via on port 3000 with login “admin” and password “password by default following this link (when the services are up and running). If needed, the port can be changed by setting the “GRAFANA_PORT” environment variable or editing the “docker-compose.yaml” file before running the “docker compose up -d” command.

The unit tests can be run locally by running the “unit-tests.sh” script. Please note that this requires cargo and a JDK17 installation for Rust and Java/Kotlin respectively.

2. Description of the CI/CD pipeline

The pipeline is divided into 4 stages: static-analysis, build, test and deploy. Each stage waits for the previous stage to be completed successfully (excluding the static-analysis stage), and each run of the pipeline is fully independent from the other runs.

The base image used for the pipeline runs is “ubuntu:23.10”, and scripts are run at the start of each job to install the tools necessary to run the job, that is, docker, necessary packages or the build tools of each service.

Version management

In a normal project, the master/main branch would be the release branch from which the deployment is done, and development would be on a separate dev branch, with each new feature being first worked on a feature branch. When a feature is complete, we do a pull request from the feature branch to the dev branch, and when we have enough feature or when the dev branch is stable enough, we merge it into the master/main branch.

In this project since I am working alone on it and we already use branches for a different purpose, I believe using that kind of structure would add unnecessary complexity to the project, this is why I chose to only use the “project” branch and to run the pipeline on each push to this branch.

Building tools

Each service uses its own build tool, Rust uses cargo, Java and Kotlin use Gradle. Those greatly simplify the builds as we only need to call them and pass the target task compared to long compile commands.

Testing: tools and test cases

The tests are divided in 2 distinct parts:

- The unit tests, which use each service respective build tool to run the tests corresponding to the service. This is done during the « unit-test-job ».
- The system tests, which only tests functionalities that are exposed to the users in our case, the gateway API. Since we are testing a simple HTTP API, I chose to use the OpenAPI standard to describe it, then use Schemathesis on a test environment, separated from the production environment, by passing it the specification file and the API entrypoint, making it check if the API correspond to the specification.

Packing

The objective here is to create the docker images from the source code that will then be used during deployment.

To do so, we use docker multi-stage builds:

1. The source code and the build files are first copied to a container containing the appropriate build tools,
2. We run the build task(s),
3. Once this is complete, we extract the generated binary and copy it to the final image, containing only the necessary tools to run the service.

Deployment

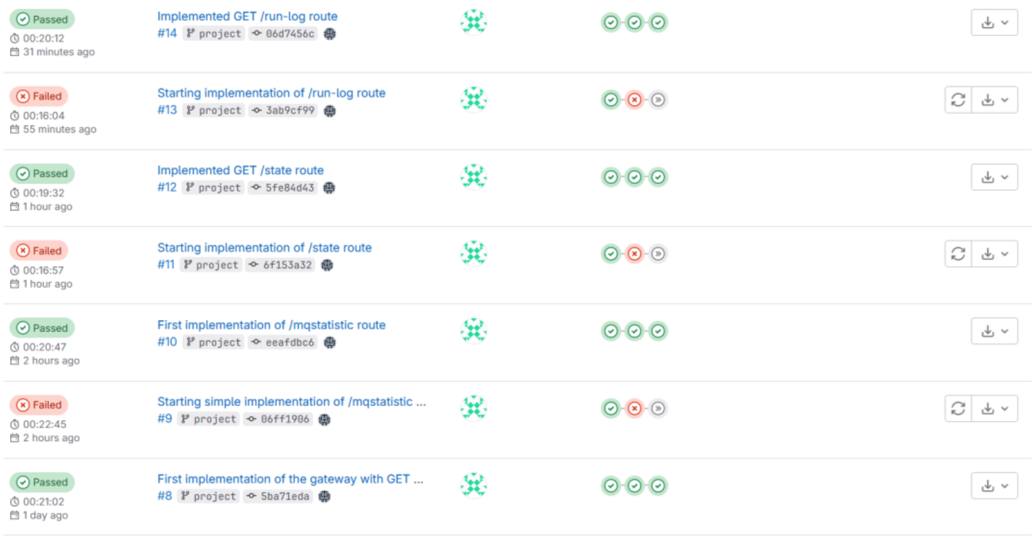
As I don’t have any server at hand for deployment, I chose to do a simple local docker deployment using “docker compose up -d”.

Operating & Monitoring

As the Gateway service expose a simple HTTP API, it is very simple to control using “curl” or “Postman”, for example. In a consumer-oriented solution, a front-end application (software, web, mobile, …) would however be needed to format and display data and actions in a more user-friendly format.

For monitoring, Grafana provides a very user-friendly and comprehensive interface, making it very easy to see what is going on and keep track of the evolution of the system.

3. Example runs of the pipeline



Passed 00:20:12 31 minutes ago	Implemented GET /run-log route #14 project 06d7456c
Failed 00:16:04 55 minutes ago	Starting implementation of /run-log route #13 project 3ab9cf99
Passed 00:19:32 1 hour ago	Implemented GET /state route #12 project 5fe84d43
Failed 00:16:57 1 hour ago	Starting implementation of /state route #11 project 6f153a32
Passed 00:20:47 2 hours ago	First implementation of /mqstatistic route #10 project eeafdbc6
Failed 00:22:45 2 hours ago	Starting simple implementation of /mqstatistic ... #9 project 06ff1906
Passed 00:21:02 1 day ago	First implementation of the gateway with GET ... #8 project 5ba71eda

Figure 7 - Overall view of pipeline runs (before static-analysis job addition)

This is a summary of the runs of the pipeline. Here we see an alternation of failing and succeeding tests, corresponding to the application of the TDD (for the API). Each time I started working on a new route I added the tests for that route, and then implemented the behavior necessary for it.

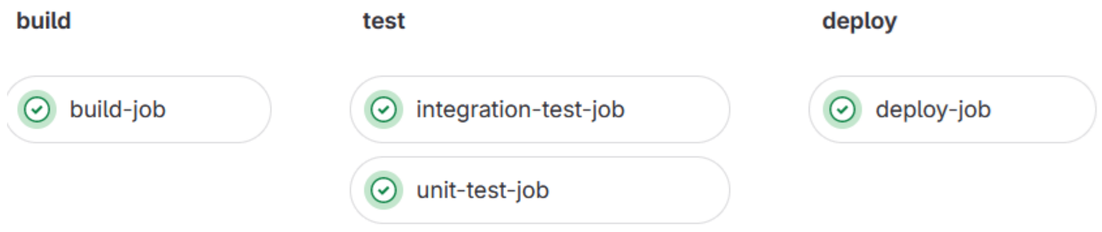


Figure 8 - Pipeline steps on success

Here you can see the successful execution of the pipeline.

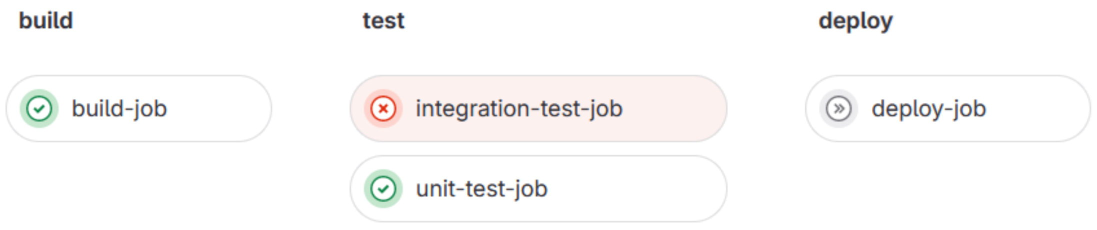


Figure 9 - Pipeline steps with failing test

Here you can see an execution of the pipeline with a failing API test.

For more details about each job run logs, please see the example log files in "./documentation/logs/*.log". There is a log file for each job and 2 for the static-analysis, unit-tests and system-tests jobs (failing and passing).

4. Reflections

Main learning and worst difficulties

During this project, I learned a lot about orchestration and the whole continuous deployment process.

The main difficulties I faced during this project were setting up the communications between all services, setting up GitLab with a runner and SonarQube.

As for what I think should have been done differently, I would say the deployment. In a "real" environment, each service should be versioned, tagged and published to an image repository, which would then be pulled and deployed in a Kubernetes (or similar) environment.

Amount of effort

As I said before, the services themselves were pretty easy to implement, what took me the most time was everything related to GitLab and the communication between services.

I would say that this project took about 40-45 hours overall (excluding work done in previous exercises).

5. Sources

- Cover page icon
- GitLab repository