1. Instructions for the teaching assistant

Implemented optional features

1. The MQ Statistics
2. The Logging/Troubleshooting website thingy
    a. The API serves this from index, displaying the same information in a different service seemed like just waste of time and resources.
3. The tests include integration tests from the other services through the API (12 tests or something) if it counts.

Instructions for examiner to test the system.

From the assignment:

```
$ git clone -b project <the git url you gave> $ cd <created folder>
$ docker-compose build --no-cache
$ docker-compose up -d
```

There is nothing special required to get the system running if you use the above.
Tested with a fresh WSL linux without caches.
If you do however use the pipelines to test, it assumes that the new version for docker compose is installed (i.e., **docker compose** up -d, not **docker-compose** up -d). If you want to test the pipeline, update docker to support the new syntax, or modify the gitlab-ci to use docker-compose. It is a drop-in replacement.

Curls can be made to the API; it is tested with them.

The logging/troubleshoot page is available at localhost:8083 with a browser. It is a static endpoint without WebSockets or fancy frameworks, so updating is done via F5 or ctrl-r ☺

// NOTE: the "SHUTDOWN" state PUT doesn't exit the MQ, more explanations in the "difficulties" section.

2. Description of the CI/CD pipeline

Briefly document all steps:

- Version management; use of branches etc
    - Version management was very basic in this project, since working alone. The CI pipeline was in the remote ci, and that's where I pushed most of the time for automatic testing. The main github origin was left more pristine, with rebases and squashed commits.

- Building tools
    - The primary tool for the project was Bun, which is a node-like JavaScript runtime, with all kinds of built-in things and built-in TypeScript support, so there was no need for a TS build step. Very neat for this kind of small stuff. Service 1 was implemented with C# and ASP.NET framework.

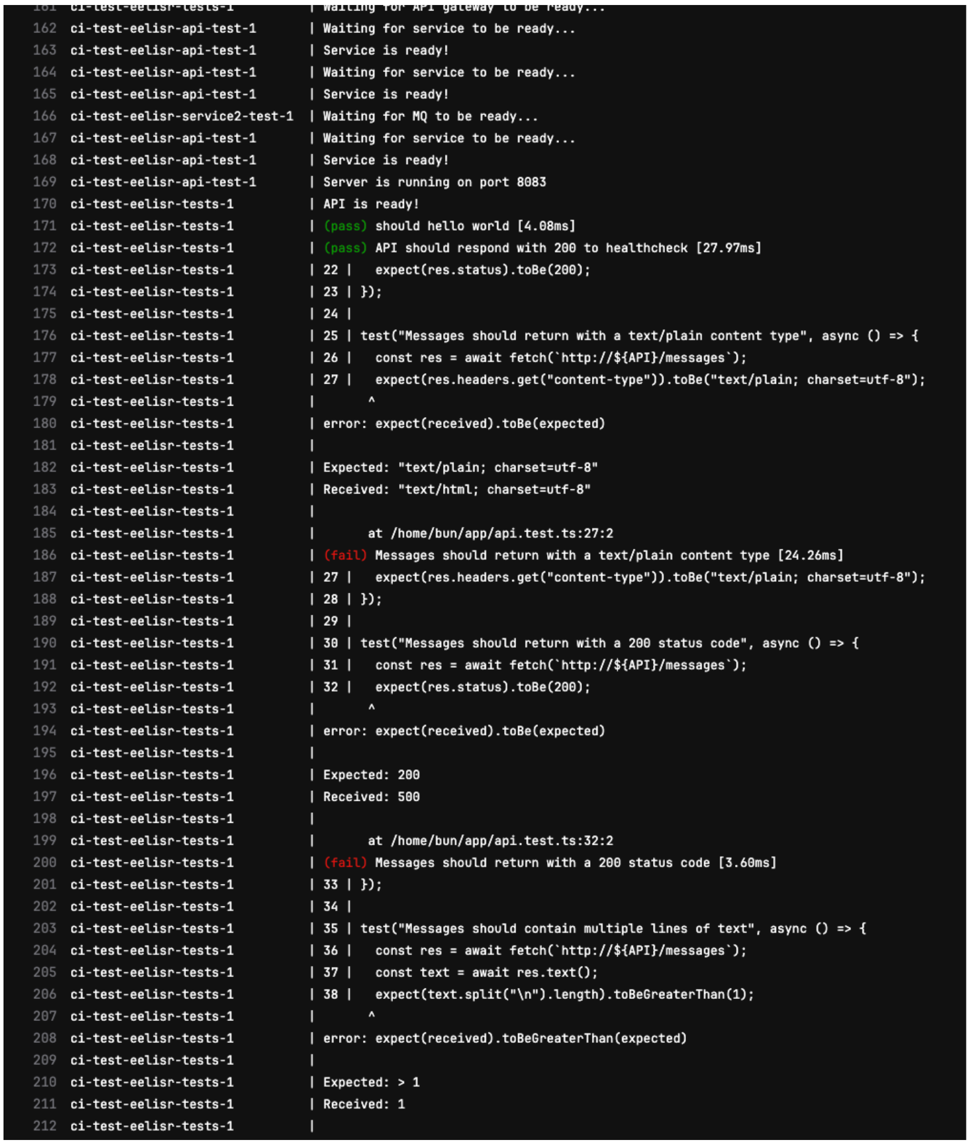
- Testing; tools and test cases
    - The tests are also implemented with Bun, which was very easy since the language is the same, and testing is as easy as ‘bun test’ and it would sniff out the tests from the test files. Tests are in **tests/api.test.ts**. There are tests for all API endpoints, as well as some integration tests to test the actual functionality when changing states etc.
    - The /mqstatistic test fails, because it can’t parse the response JSON, don’t know why but it works anyway, so didn’t bother to fix the test.
    - The /shutdown test is flaky, since well everything is shutting down and maybe even not a great test :D.

- Packing & Deployment
    - No packing or publishing to other services, the pipeline just run docker compose on the runner machine.

- Operating; monitoring
    - Monitoring can be done via the provided API, or straight from Docker.

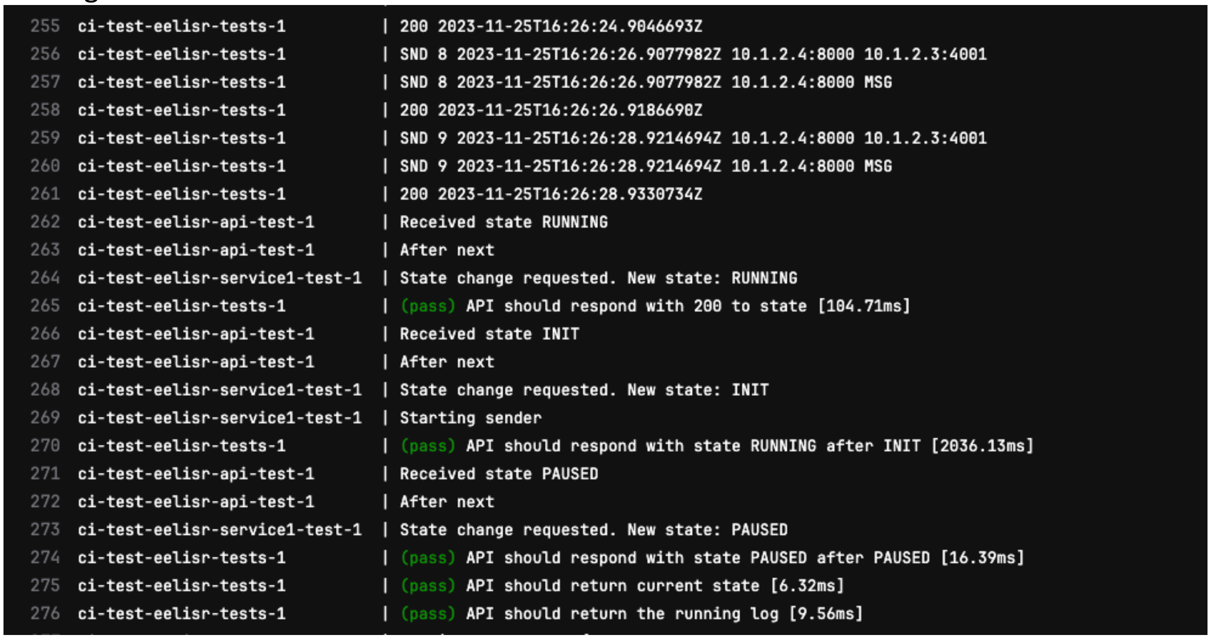
3. Example runs of the pipeline

Failing tests in the pipeline because endpoint was not implemented:



```
ci-test-eelisr-tests-1         | Waiting for API gateway to be ready...
ci-test-eelisr-api-test-1      | Waiting for service to be ready...
ci-test-eelisr-api-test-1      | Service is ready!
ci-test-eelisr-api-test-1      | Waiting for service to be ready...
ci-test-eelisr-api-test-1      | Service is ready!
ci-test-eelisr-service2-test-1 | Waiting for MQ to be ready...
ci-test-eelisr-api-test-1      | Waiting for service to be ready...
ci-test-eelisr-api-test-1      | Service is ready!
ci-test-eelisr-api-test-1      | Server is running on port 8083
ci-test-eelisr-tests-1         | API is ready!
ci-test-eelisr-tests-1         | (pass) should hello world [4.08ms]
ci-test-eelisr-tests-1         | (pass) API should respond with 200 to healthcheck [27.97ms]
ci-test-eelisr-tests-1         | 22 |   expect(res.status).toBe(200);
ci-test-eelisr-tests-1         | 23 | });
ci-test-eelisr-tests-1         | 24 |
ci-test-eelisr-tests-1         | 25 | test("Messages should return with a text/plain content type", async () => {
ci-test-eelisr-tests-1         | 26 |   const res = await fetch(`http://${API}/messages`);
ci-test-eelisr-tests-1         | 27 |   expect(res.headers.get("content-type")).toBe("text/plain; charset=utf-8");
ci-test-eelisr-tests-1         |        ^
ci-test-eelisr-tests-1         | error: expect(received).toBe(expected)
ci-test-eelisr-tests-1         |
ci-test-eelisr-tests-1         | Expected: "text/plain; charset=utf-8"
ci-test-eelisr-tests-1         | Received: "text/html; charset=utf-8"
ci-test-eelisr-tests-1         |
ci-test-eelisr-tests-1         |       at /home/bun/app/api.test.ts:27:2
ci-test-eelisr-tests-1         | (fail) Messages should return with a text/plain content type [24.26ms]
ci-test-eelisr-tests-1         | 27 |   expect(res.headers.get("content-type")).toBe("text/plain; charset=utf-8");
ci-test-eelisr-tests-1         | 28 | });
ci-test-eelisr-tests-1         | 29 |
ci-test-eelisr-tests-1         | 30 | test("Messages should return with a 200 status code", async () => {
ci-test-eelisr-tests-1         | 31 |   const res = await fetch(`http://${API}/messages`);
ci-test-eelisr-tests-1         | 32 |   expect(res.status).toBe(200);
ci-test-eelisr-tests-1         |        ^
ci-test-eelisr-tests-1         | error: expect(received).toBe(expected)
ci-test-eelisr-tests-1         |
ci-test-eelisr-tests-1         | Expected: 200
ci-test-eelisr-tests-1         | Received: 500
ci-test-eelisr-tests-1         |
ci-test-eelisr-tests-1         |       at /home/bun/app/api.test.ts:32:2
ci-test-eelisr-tests-1         | (fail) Messages should return with a 200 status code [3.60ms]
ci-test-eelisr-tests-1         | 33 | });
ci-test-eelisr-tests-1         | 34 |
ci-test-eelisr-tests-1         | 35 | test("Messages should contain multiple lines of text", async () => {
ci-test-eelisr-tests-1         | 36 |   const res = await fetch(`http://${API}/messages`);
ci-test-eelisr-tests-1         | 37 |   const text = await res.text();
ci-test-eelisr-tests-1         | 38 |   expect(text.split("\n").length).toBeGreaterThan(1);
ci-test-eelisr-tests-1         |        ^
ci-test-eelisr-tests-1         | error: expect(received).toBeGreaterThan(expected)
ci-test-eelisr-tests-1         |
ci-test-eelisr-tests-1         | Expected: > 1
ci-test-eelisr-tests-1         | Received: 1
ci-test-eelisr-tests-1         |
```

Passing tests in the final form:



```
ci-test-eelisr-tests-1         | 200 2023-11-25T16:26:24.9046693Z
ci-test-eelisr-tests-1         | SND 8 2023-11-25T16:26:26.9077982Z 10.1.2.4:8000 10.1.2.3:4001
ci-test-eelisr-tests-1         | SND 8 2023-11-25T16:26:26.9077982Z 10.1.2.4:8000 MSG
ci-test-eelisr-tests-1         | 200 2023-11-25T16:26:26.9186690Z
ci-test-eelisr-tests-1         | SND 9 2023-11-25T16:26:28.9214694Z 10.1.2.4:8000 10.1.2.3:4001
ci-test-eelisr-tests-1         | SND 9 2023-11-25T16:26:28.9214694Z 10.1.2.4:8000 MSG
ci-test-eelisr-tests-1         | 200 2023-11-25T16:26:28.9330734Z
ci-test-eelisr-api-test-1      | Received state RUNNING
ci-test-eelisr-api-test-1      | After next
ci-test-eelisr-service1-test-1 | State change requested. New state: RUNNING
ci-test-eelisr-tests-1         | (pass) API should respond with 200 to state [104.71ms]
ci-test-eelisr-api-test-1      | Received state INIT
ci-test-eelisr-api-test-1      | After next
ci-test-eelisr-service1-test-1 | State change requested. New state: INIT
ci-test-eelisr-service1-test-1 | Starting sender
ci-test-eelisr-tests-1         | (pass) API should respond with state RUNNING after INIT [2036.13ms]
ci-test-eelisr-api-test-1      | Received state PAUSED
ci-test-eelisr-api-test-1      | After next
ci-test-eelisr-service1-test-1 | State change requested. New state: PAUSED
ci-test-eelisr-tests-1         | (pass) API should respond with state PAUSED after PAUSED [16.39ms]
ci-test-eelisr-tests-1         | (pass) API should return current state [6.32ms]
ci-test-eelisr-tests-1         | (pass) API should return the running log [9.56ms]
```

4. Reflections

Main learnings and worst difficulties

The project had much stuff that I had done previously, and the basic premise of the project was easy enough. Setting up my own CI runner was new and was nice to see the reason for why sometimes we need to wait for runners to free up, they are just processes on some computer somewhere, and they are not limitless.

This was the first time I had to convert a console application (the .NET one) into a server, which was a curious thing. The .NET servers I've done in the past have been much larger and boilerplatey, but here I took shortcuts to keep the programs tidy. But, by turning a console application with in-memory state into a server I created a monstrosity that shouldn't exist. Keeping the system state in memory and commanding it via "REST" API is nightmare fuel and making that kind of a program in the real world will cause (and has caused) lots of damage due to unpredictable state. Would not recommend.

I didn't need to touch service 2 and Monitor, which was a nice surprise. Goes to show that this kind of microservice design has its perks, which leads to my main (reinforcement of knowledge) learning:

Microservices are nothing but a burden and complexity monster if the system is not large enough.

This system doesn't do much and could be replaced by just one little "monolith" application. Microservices design shows its perks when there are multitudes of teams that don't need to know anything else about each other than the interfaces they provide and consume. By making a one-man-project into a microservice based thing, the amount of time and brain work needed to create something grows exponentially. This project did teach it the hard way.

Worst difficulty for me during this project was the shutdown command to my surprise and it still doesn't work as correctly as it should. In the end I implemented it by just making bunch of "destroy yourself" calls to the services from the API, which is maybe the most kosher way of doing it. Otherwise, I would've needed to mount the docker daemon to the API container and run the "docker-compose down" from a child process there, but that is a highly hacky way of doing things and by any means not recommended. It would've been maybe possible to make it work by making a dind, but that is also not recommended, and I have a strong will to not learn the wrong thing. The "shutdown" requesting seems to be a quite good alternative, but I couldn't get the RabbitMQ to exit, since it doesn't have any "quit" API and it doesn't exit the container even if connections and queues are closed. Maybe some kind of rabbitmqctl connection to the container could've worked over HTTP.

All in all, most of the technical difficulties spanned from trying to make docker do things it isn't made for (commanding the host or other containers). Coding the services was enjoyable though.

Amount effort (hours) used

The actual implementing took me almost a whole Friday and Saturday. With brakes/gym etc. subtracted, I think the implementation on this (some would say crude) level took maybe **20** hours, give, or take.