Description of the CI/CD pipeline.

It is recommended to have a call with the student to be able to set up local gitlab container to run and test pipeline in the correct way. Local machine is not strong enough to run every things and is documented step by step

The pipeline consists of three stages:

- Build
  This stage simply runs the following command

  ```
  docker-compose build
  ```

  It will try to create containers from HTTPENV, OBSE, ORIG and IMED images and install node module dependencies to verify.
- Test
  This is the second stage of the pipeline. The stage will run the command:

  ```
  docker-compose -f docker-compose.test.yml up --build
  ```

  It will install all node modules dependency and run `eslint` and `jest` in HTTPENV container. `eslint` uses airbnb and recommended config.
- Deploy
  The stage is not yet implemented due to lack of environment for testing.

Instructions for examiner to test the system.

First running the docker command at root folder:

```
docker-compose up --build
```

After about 30 seconds, HTTPENV container is available at localhost:8081. The main page is the Swagger documentation of available API gateways.

Optional features:

1. Implement a static analysis step in the pipeline by using tools like jlint, pylint or SonarQube.
   At httpenv folder, run this nodejs command

   ```
   npm run eslint
   ```

   The command will trigger `eslint` node module based on `airbnb` config and verify if all js files under api folder are written in correct format.
2. Implement monitoring and logging for troubleshooting
   The httpenv is an expressjs application and by default expressjs is supported by a HTTP request logger middleware, which is morgan. If the docker compose command was executed before and several requests were sent to the httpenv container, there is an access.log can be accessed via cli at this path : ```httpenv/api/log/access.log` inside the container.

Example runs (some kind of log) of both failing test and passing.

There are two branches available in the repository. master is the default branch and it only has passing test cases. However, failed-pipeline is the branch that has a failed test to fail the CI pipeline. It can be run in three ways at root folder on both `master` and `failed-pipeline` branch.

1. Run

   ```
   docker-compose -f docker-compose.test.yml up --build
   ```

2. Goes to httpenv folder and run:

   ```
   npm run test:lint
   ```

3. Trigger CI pipeline at gitlab repository

Main learnings and worst difficulties

To be able to control the sending message flow of the OBSE container, it is required to store the current state at some sort of data. Adding another database image in the docker compose is a bit overkill, so it is decided to simply store the current state undere json files and share between containters. It is a learning topic to let other containers having privileges to a certain folder of the container.

In java, mocking in test is well supported and easily to use. Tests in this cases can be fast and easiy run without waiting for external connection setup. However, many tests instructions for nodejs developer are mostly simple and not very well focus on writing test in the correct way. For example, when running tests that require connecting to the database, instructions actually want to establish database connection and assert callback. However, this kind of approach will slow down tests and force tests closely connecting to the up and running to the database, and completely ignore the idea of mocking and stubbing. In short term, it is hard to mock when writinng tests in javascript due to lack of instructions.

Amount effort (hours) used

6 hours in days -> 30 hours