Task

Your task is to develop a CI/CD pipeline on GitLab for the provided FastAPI application. This would entail writing python unit tests for the FastAPI application, writing a Dockerfile to containerize the application, and writing the configuration file for the GitLab CI/CD tool.

The provided FastAPI application provides three simple course management APIs. It provides APIs to get all the courses in the database, get a single course with a course id from the databse, and create a new course in the database and return it. For simplicity, the database is represented using a python dictionary. The first task is to write unit tests that verifies the status code and json response for the APIs, GET /courses/{course_id} and POST /courses/, for existing and non existing courses in the database.

The second task is to write a Dockerfile that will be used to containerize the

The third task is to write the config file for GitLab CI/CD tool to build the pipeline. In the build pipeline, you have to perform linting, testing, compiling, and dockerize the provided application.

The GitLab repository for the assignment should be created by the student using the form here and that one needs to be utilized for the task.

Requirements

The tests should be written in the file `app/tests/test_main.py`. You can refer to this example regarding testing FastAPI application. The python unit tests will be run with the `pytest` command. The tests should be as follows:

- Check if GET `/courses/{course_id}` returns the status code `200` and that the json response is correct (as per db) when the course exists in the database
- Check if GET `/courses/{course_id}` returns status code `404` and that the json response is `{"detail": "Course does not exist"}` when the course does not exist in the database
- Check if POST `/courses/` returns status code `200` and that the json response contains the correct json data if the course does not exist in the db
- Check if POST `/courses/` returns status code `400` and that the json response is `{"detail": "Course already exists"}` if the course already exists in the db.

Attention

Make sure you only have these four tests when submitting your files for grading.

Note

You can refer to the official example for writing the FastAPI tests and executing it with `pytest`.

The Dockerfile should do the following:

- Use image `python:3.10.8-alpine3.16`
- Work directory should be `/application`
- Copy the requirements.txt to the image
- Install the requirements
- Copy the application to the image
- Expose port 8000
- Run the command `uvicorn main:app --reload`

The name of the image **should be courses-api:v1**

The CI/CD config file should do the following:

- Install the requirements from the provided `requirements.txt` automatically before execution of each job in the pipeline **if required**.
- The name of the file (`main.py`) if required to be provided to any stage should be parameterized with the help of a variable called `app_file`
- **It should have the folowing jobs:**
    - A job named `apilint`, that should perform the linting on `main.py`. Install and use the `pylint` with `pip3` for linting. Other dependencies are used from the `requirements.txt`
    - A job named `apitest`, that should run the `pytest` unit tests. Install `pytest` with `pip3` for testing. All other required dependencies are already installed with `requirements.txt`
    - A job named `apicompile`, that should compile all files within the folder `app` using `compileall`.
    - A job named `apideploy`, that dockerizes the application by creating a Docker image using the Dockerfile. The name of the image should be `courses-api:v1`. Use the command `docker build --network host -t <tag> .` to build the image. Without the –network flag, the grader can run into issues in downloading the requirements.
- Use the image `python:3.10.8-alpine3.16` for jobs `apilint`, `apitest`, and `apicompile`
- Use the image `docker:20.10` for the job `apideploy`

You can refer the GitLab CI/CD pipeline configuration reference.

Attention

Do not run any of the jobs in quiet mode. `pytest`, `pylint`, python compilation, and docker build should all display output.

Note

You can refer to gitlab-ci-template file given in the repository. You can also experiment with various `stages` in the configuration file.

Attention

Make sure to name the jobs as mentioned above, otherwise our grading system will not be able to detect your jobs.

GitLab Runner and Local testing

GitLab Runner is an application that works with GitLab CI/CD to run jobs in a pipeline. It is the service responsible for executing one or many jobs in the CI/CD pipeline and reporting their progress back to the GitLab instance. It can execute the job on the same machine as its host gitlab-runner process, or preferably on a different machine. That machine might exist before the job is submitted, or it might be launched on the fly in response to the job's submission. As it is written in Go, there are no language requirements and hence supports multiple operating systems.

The GitLab Runner can be installed locally and used on GNU/Linux, macOS, FreeBSD, and Windows. There are three ways to install it. Use a repository for rpm/deb packages, binaries/executable in Windows, MacOS (also in homebrew; see the link), Linux/Unix, or within a docker container.

Note

The steps in the above link will discuss about the registeration of the GitLab runner. For running locally, this step is not required. Registering enables the GitLab runner to associate your GitLab repository with the runner. This is not required to test your code with the runner and hence need not necessarily be pursued.

Once installed, you can test your CI/CD config file as follows (all non-docker installation methods):

```
gitlab-runner exec shell <jobname>
```

For docker based installation you can check the documentation on how to start and execute the runner.

You can read about various options and limitations of the local GitLab runner here.

Grading

You must submit the git url in the form **git@version.aalto.fi:cs-e4190/{repo-name}.git**.

Attention

Make sure to test your pipeline on GitLab before submitting it. This is useful for testing any possible errors with the build process. The grader relies on a local GitLab runner which is subject to the limitations described in the documentation.

Note

The assignment runs multiple unit tests which give fractional points based on how the requirements in task are fulfilled, according to the table below.

Test	Points
ci-lint of `.gitlab-ci.yml` pass successfully	20
`apilint` pass successfully	20
`apitest` pass successfully	60
`apicompile` pass successfully	20
`apideploy` pass successfully	60