IDVOC - Project

version 1.2.0

Information

The project is personal. It is meant to be made alone. If you're stuck, require clarification or need help, please reach a teacher or ask in a public discord channel where said teachers are to provide help for everyone. Getting help from other students may not grant you proper help, and can lead to cheating. A few dozen students ended up with 0 and a cheating flag in their file in 2022 and 2023, I don't want to flag students anymore.

The submission will be done via gitlab.cri.epita.fr. You will simply push your code on the main branch of the project you'll create (as described in the next section) for a submission. There is no need to provide a git tag, a specific commit message or anything of the sort.
Results will also be provided on your project, for both dementors and the final submission. The results will be pushed as an "issue" on your project. You can check the "issue" tab after a published dementor, or subscribe to gitlab's emails and be notified when one issue is opened on your project.

The deadline MUST be respected. You'll have plenty of time for the project, and the deadline is already defined, there will be no exceptions to the deadline. There is no need to send me emails about it, I will refuse any deadline extension/late pushing of your project.

The project will be corrected automatically. This means that you will have to strictly follow the submission information, otherwise your project will not be found, or won't be corrected as you expect it to be. If you make a typo in a filename, the file will not be corrected. If the submission doesn't compile, you will end up with a 0.
Of course I don't want people to get a bad grade or a 0 because of something silly like this, mistakes happen. To prevent this from happening, I will provide 3 dementors.
The last dementor will be close to the deadline, and meant to simulate the final submission. I expect people to finish the project before the 3rd dementor, and to use it as a safety net to make sure your submission is correct.

Because of the dementors, I will not accept any resubmission after the deadline if you made a mistake causing a compilation error or leading to a file not found. Please don't send me emails for this.

Setting up the env

A skeleton is provided on gitlab.cri.epita.fr/cyril/idvoc-2024

You need to use this project as a base for your project in your own namespace.

The project URL will be the following:

gitlab.cri.epita.fr/<your login>/idvoc-2024

The project slug must be "idvoc-2024"

The project name must be "IDVOC-2024"

Then, you need to add me to your repository so I can access it and push the correction.

To do so, go to Manage -> Members and Invite me

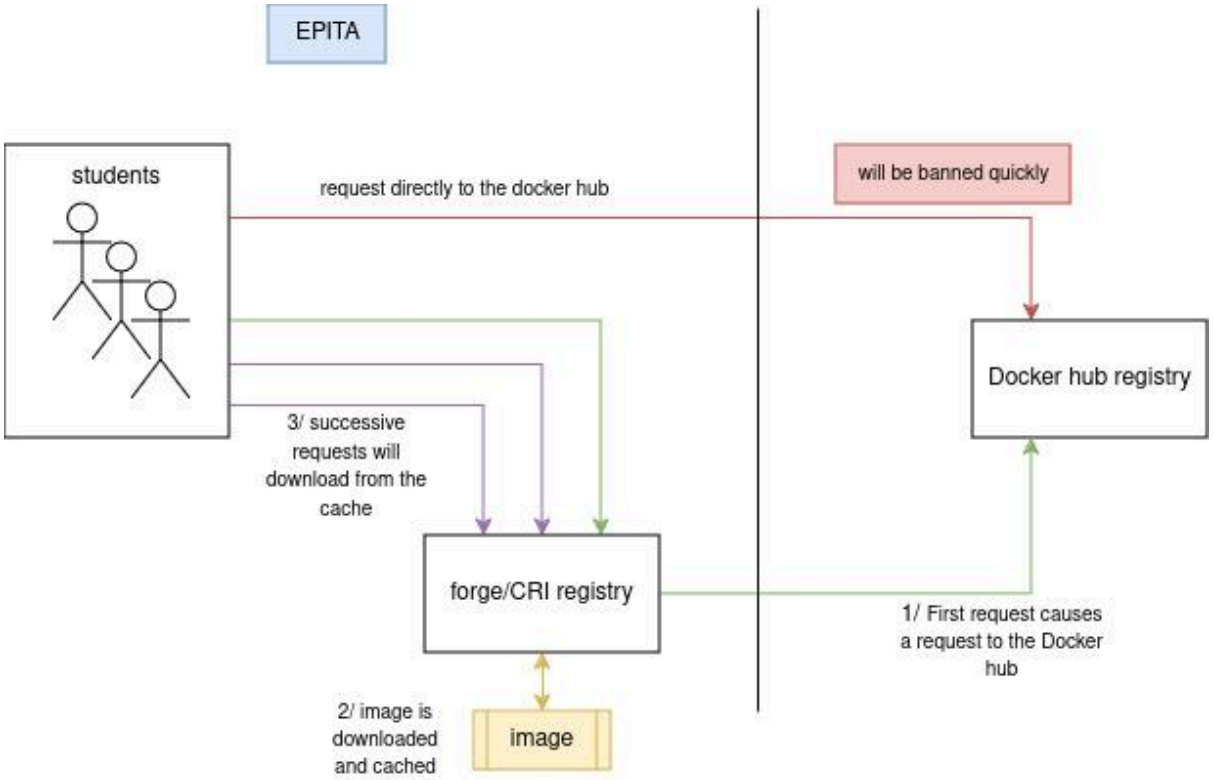
username: cyril
role: Owner
no expiration date

Docker registry proxycache

The docker hub, the default docker registry, imposes a limit on the number of images that can be downloaded. As the limit will quickly be reached if many people work simultaneously on the project from EPITA, one will not be able to download images from the registry. Therefore, it is **very important to avoid being banned from docker** by using the proxycache registry the forge is providing.

This docker registry is a proxycache. Proxy because it transparently proxy the requests you make to the docker hub, and cache because it caches the images, avoiding to download

them a second time if it already exists



It is therefore very important to **not use docker hub directly but use reg.undercloud.cri.epita.fr/docker**.

Example:
~~docker run debian:bookworm-slim~~ ->
docker run reg.undercloud.cri.epita.fr/docker/debian:bookworm-slim
~~FROM debian:bookworm-slim~~ ->
FROM reg.undercloud.cri.epita.fr/docker/debian:bookworm-slim

You can still browse the docker hub to find the images you need, but you need to download them from the proxycache registry

Context

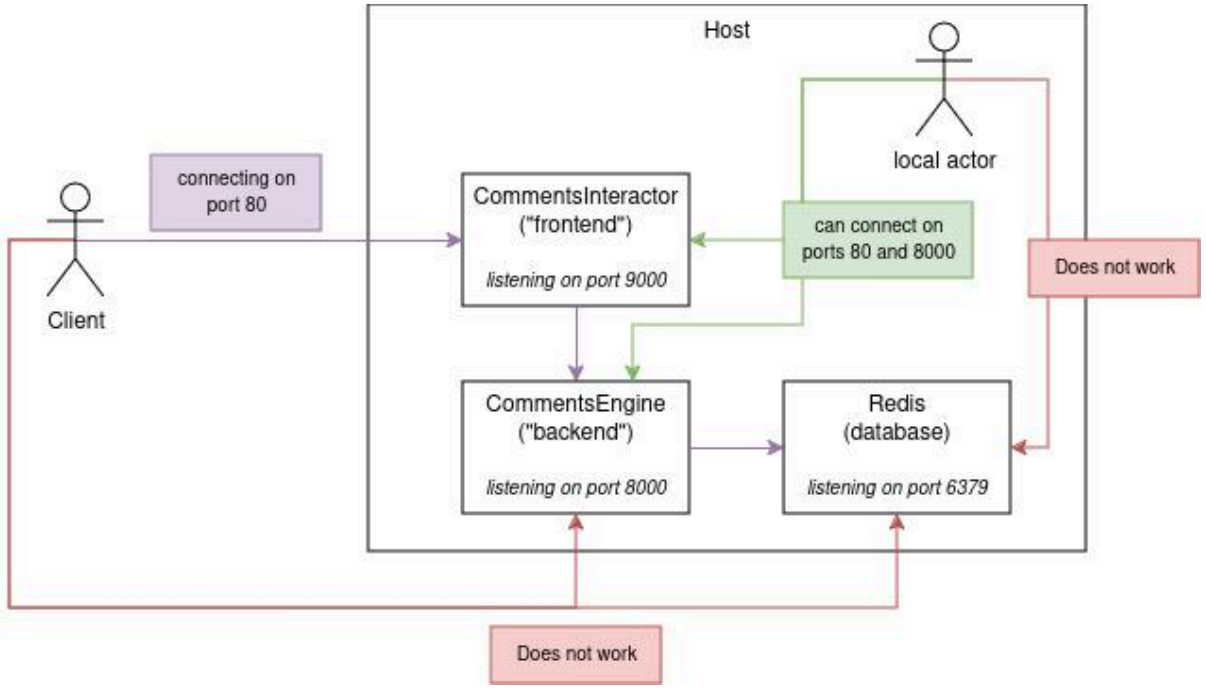
Somebody wrote a very basic application meant to post comments and read comments on some dashboard. Because they have heard about micro services architecture, they wanted to have a try, and designed the application with this approach. Of course, because it's still a PoC and because it was written by someone with limited development skills, the application is far from being perfect.

While the code is definitely terrible, your task here will be to help dockerize this application and deploy it. You will not modify the application's code itself, but only write docker related resources to deploy this application.

The application is divided into 3 components. 2 of them are homemade, and the third one is an external dependency.

The application is composed of a frontend part, CommentsInteractor with a superb and modern UI to put the user at ease. This frontend communicates with the backend, CommentsEngine which is responsible for manipulating the comments published and retrieving them. It exposes API routes used by CommentsInteractor.

CommentsEngine stores its comments in a dedicated database, redis. The way redis is used there is not to be mentioned, as it's … pretty bad, but at least the task will be simpler for you, as redis is really easy to manage.



Here is the diagram of the whole application we're building once step 2 is finished.

First step: Creating Dockerfiles

Submission

For the first step of this project, you have to provide 2 files:

- ./CommentsEngine/Dockerfile
- ./CommentsInteractor/Dockerfile

Making it work

You have to write 2 Dockerfiles to build 2 docker images, one for CommentsEngine and one for CommentsInteractor.
The application provided is in golang. The Dockerfile shall compile the go application and create an image ready to run the built application.
What's expected here:

- The docker image can successfully be built with your Dockerfile
- A container created from this image automatically starts the application
- The application is correctly configured to be reachable if we expose the application's port when running a container (as seen in class)

It is part of the task for you to find how to build the application, how to build a docker image, to manipulate the image you've just built to see if it's working, and to try to reach the component you've just built. You will find plenty of information on the internet about those subjects. The idea behind this task is not only to make you manipulate Docker and write a Dockerfile, but also to encourage you to search yourself and be able to solve problems like this one with limited guidance, as it might – *and most probably will* – be the case once you're out of EPITA.
It is **not** expected for the first step to assemble the components yet. If CommentsInteractor gives you an error "could not HTTP get on CommentsEngine" or if CommentsEngine gives you an error "Problem with redis", it is **normal** at this point. You can see on the schema as the 2 blocks for CommentsInteractor and CommentsEngine started successfully, but not linked together, and no database provided yet.
I recommend you to test your application with **curl as firefox/safari/chrome might interfere with the results (caching, HTTPS, ...)**

Making it cleaner

While your Dockerfiles should be pretty decent at this point and should be enough to make the applications simply work, we need to go further and write proper Dockerfiles following the good practices.
There are plenty of good practices that can be applied, I invite you to look for some online. However, here are some points that we are looking for in this project:

- Use a good docker image and tag from which to start building your image. Remember what has been said in class about them
- Expose the maintainer of the image you've built
- Running the application as root is not the best idea. Create another user and run CommentsEngine and CommentsInteractor as another user than root

- Install curl and bash in the image. It's not always a good idea, but we'd like to have these tools in this case, it's easier to debug. Make sure to not keep cache in the image after installing curl and bash !
- Try to not use too many layers, and create them in the right order. Make sure to share the common layers between the 2 images as they are quite similar, to avoid wasting too much disk space

Second step: Writing docker-compose.yml

Submission

For the first step of this project, you have to provide 1 file:

- ./docker-compose.yml

Making it work

You have to write a docker-compose.yml file to deploy the application (composed of the 3 services described in the Context part of this subject). The idea here is to use the 2 Dockerfiles you have written and to find an image online for redis and assemble them in a compose file that will be able to create the whole application.
Is expected for this step:

- A valid docker-compose.yml file that can be used with docker compose to create 3 containers
- 3 services in the docker-compose.yml named "commentsinteractor", "commentsengine" and "redis"
- The redis service will use one docker image found online that will start redis database, the two other services will build 2 docker images with the 2 Dockerfiles you have written
- The CommentsInteractor component will be exposed to the user on port 80
    - It must still listen on port 9000 inside the container

making it cleaner

Let's try having a docker-compose.yml file a bit fancier !
Here are a few points to address:

- Expose commentsengine on port 8000, but only locally. It must not be accessible from another host than yours
- Make sure redis is not exposed to anyone but commentsengine. In fact, commentsinteractor and redis should not be able to reach each other.
- Define some container_name and hostname, matching the name of the services

- Make sure that a service's dependency will be started when this specific service is started. For example, make redis start when commentsengine starts
- Make redis database persistent across container recreation. Use a docker volume for this
- Add some limits to the containers. Limit each container to 2 cpus and 150 MiB of RAM.
- Write some fancier YAML. Some elements are repeated in this config file, maybe we can use a YAML anchor to avoid re-writing the same things over and over again ?
- Make sure to use the best image and tag for redis

Third step: Seeing metrics

The author of CommentsEngine and CommentsInteractor has included a prometheus library inside the applications, and tried to expose a few metrics. The goal now is to scrape this metrics with prometheus, and observe them with Grafana, and even create a grafana dashboard for this

submission

For this step, you must submit 3 files:

- ./metrics/docker-compose.yml
- ./metrics/prometheus.yml
- ./metrics/dashboard.json

Setting up the metrics stack

In *./metrics*, you will find a skeleton file for a *docker-compose.yml* and *prometheus.yml*, and a *start.sh* script.

About start.sh:

Due to limitations of the AFS, for security reasons you cannot mount a file from the AFS to a container. Therefore, we need to put the files we wish to mount somewhere else, like in /tmp. Because only the AFS is persistent on the school computers, any modification you make to the configuration file will not be saved if you don't reflect the changes on both /tmp and the AFS. As it's annoying, and not something we wish for, I provide a very simple start.sh script to use instead of docker compose up, that will copy the local config file in /tmp and docker compose up. Therefore, you only have to modify the local config file and use start.sh.

The first step is to complete the docker-compose.yml file and to fix the FIXMEs inside. You need to:

- Provide the prometheus.yml configuration file to Prometheus. The source file must be mounted from /tmp/prometheus.yml (and therefore you should use ./start.sh script).
- Expose prometheus locally, on its port 9090
- Use docker networks to make sure prometheus can reach CommentsEngine and CommentsInteractor created in the previous step. You may have to modify the previous docker-compose.yml, this is up to you. The use of network_mode: host is however forbidden.
- You can also add more configuration to grafana service if needed

You also need to fix the configuration file of Prometheus to add a second job, named "commentsApps" that will scrape the metrics of both CommentsEngine and CommentsInteractor.

You can see if things seem to be working by reaching the prometheus interface, and looking at the "Targets" tab.

If the targets are green and UP, you are scraping the metrics successfully ! Now let's observe them in Grafana

Creating a dashboard

Reach Grafana interface, and login using the default credentials admin/admin.

You then need to add a datasource, and to add your Prometheus as a datasource. After that, Grafana will be able to connect to Prometheus and run queries, fetching the metrics you scraped and be able to create charts.

Create a new dashboard for CommentsEngine. The dashboard must be named "CommentsEngine". Add a visualization, named "HTTP hits on /latest" that plots the number of HTTP hits made on the /latest API route. You will be able to see the value go up as you open/refresh the main page of the application.

Save the dashboard, and in the settings of the dashboard, find its JSON model, and save the whole JSON model in your git repository, in "./metrics/dashboard.json". This will be the submission file.

Git workflow

The way you used git will also be part of the evaluation. It is expected to complete the project on top of the already existing commits made available to you in the public idvoc-2024 project. I expect to see all of the commits the original author made. If you have started your project without those commits, don’t worry, you can still recover them and add your commits on top of them. Look about adding another remote, fetching from the remote, and rebasing your branch from the other one. I can only strongly advise you to make a local backup of your repository first (simple cp -r should be enough) if you’re not really at ease with these commands and workflow, in case of a problem, to not lose everything.
In the end, all the original commits from idvoc-2024 must be present in your repository, at the very beginning of the main/master branch, and your work added on top of it.