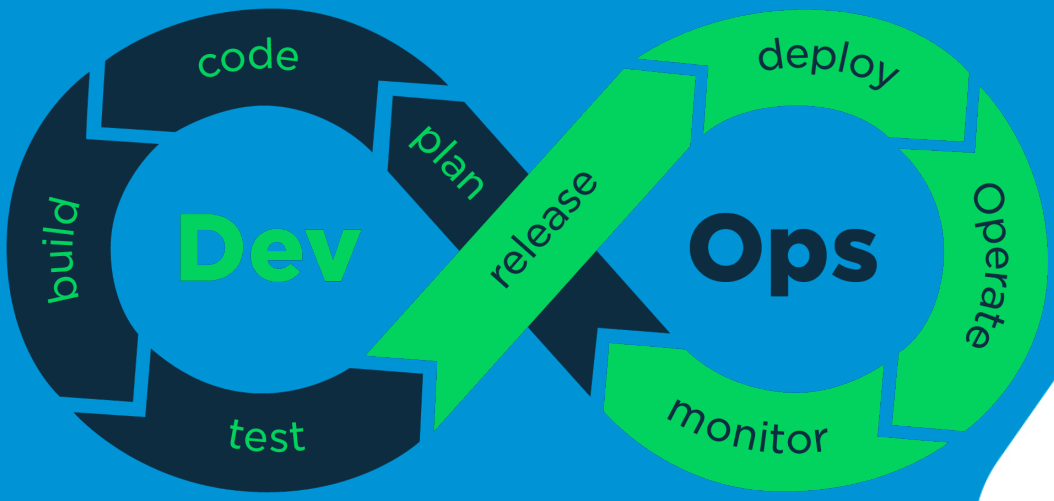
{EPITECH}

CHOCOLATINE

IMPROVE INTEGRATION AND TESTING WITH GIT-HUB ACTIONS



CHOCOLATINE

Whatever your opinions might be, this is a *chocolatine*.

But what purpose does it serve? It allows you to go through the day and through your work better. In other words, it facilitate your work and allows you to focus on things that matter (discussing its name does not pertain to this category).

Your repositories are hosted on GitHub, and GitHub has an intuitive way of automating the launch of predefined actions, triggered by events of your choice. It is, in the same way as chocolatines, a great way to facilitate and enhance your development and your workflow.

In this project, you will setup a GitHub Actions workflow to enforce good practices and diverse rules in a repository.

GitHub Actions are free and unlimited for public repositories, and free up to 2,000 minutes per month for personnal private repositories, which is quite generous!

Technical details

You will have to turn in a single YAML workflow file named `chocolatine.yml`, placed either:

- ✓ at the root of your repository;
- ✓ or in the `.github/workflows` folder.

Your workflow must be usable with your different Epitech projects and their respective technologies and associated tools.

For this project, the **only** external actions allowed are `actions/checkout` and `pixta-dev/repository-mirroring-action`.
All other external actions (which you can find on the GitHub marketplace for example) are strictly **forbidden**.

If particular settings or elements are not specified or addressed in the subject, you are free to do as you please with them.

Evaluation

Your workflow will be tested by copying your `chocolatine.yml` file, **and only this file**, into a test repository's `.github/workflows` directory.

Make then sure that the workflow is **self-contained** and does **not need any external files**.

Secrets

You might need to use secured data or values to make your workflow successfully run.

In this case, you **must** use secrets.

If any hardcoded sensitive value is found in your workflow file, your entire project will fail. You have been warned.

Features

Using a GitHub Actions workflow, you will need to implement a set of features.

The workflow must be run on every **push** and on every **pull request** creation, **unless**:

- ✓ the branch name starts with `ga-ignore-`;
- ✓ or the current repository is the same as the mirror repository.

In any of these two cases, no job must not be run at all.

The "not being in the mirror repository" requirement is important, as it will prevent the mirroring from being triggered in the target repository itself.

Triggering an error by exiting with a positive value is **not considered proper handling** of this case.

Have a look at the default environment variables available in the GitHub Actions environment, you might find something useful. ;)

Furthermore, **each job must**:

- ✓ start by **checking out the repository** to the relevant branch;
- ✓ only be run if the **previous job has succeeded**.

The order in which the jobs are expected to be run is the order in which they are defined below in the subject.

Environment variables

You need to define several environment variables, available at the **workflow level**.

- ✓ `MIRROR_URL`: the URL of the Epitech repository which will act as a mirror;
- ✓ `EXECUTABLES`: a comma-separated list of the paths of the executables expected to be produced (it can have a single executable, like `"mysh"` or `"directory1/subdirectory2/my_hunter"`, or several, like `"directory1/executableA,directory2/executableB"`).

All environment variables' values will be strings.

"available at the workflow level" means that the environment variables must be only defined **once**, and be available for **all jobs and steps** of the workflow.

Jobs

Checking the coding style

Create a job with a `check_coding_style` ID that runs in a `ghcr.io/epitech/coding-style-checker:latest` Docker container, and that in order:

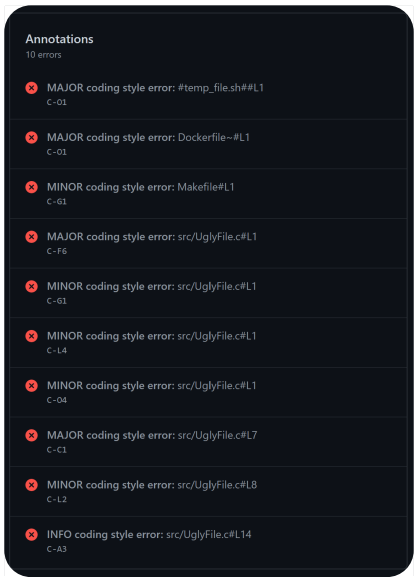
- ✓ runs the coding style checker script with the following command:

```
Terminal
check.sh $(pwd) $(pwd)
```

This command will create the well-known `coding-style-reports.log` file.

- ✓ if any, displays each coding style error as an *error annotation*;
- ✓ makes the job fail if there are any coding style errors.

The output of the job must be similar to this:



By using the appropriate error annotation parameters, you **must** be able to click on it and be taken to the precise line of the offending file.

This will only work if the offending line has been introduced or modified in the current commit.

Checking that the program compiles correctly

Create a job with a `check_program_compilation` ID that, in order:

- ✓ launches `make` at the root of the repository (this **step** must have a 2 minutes timeout);
- ✓ launches (on a separate step) `make clean` at the root of the repository;
- ✓ verifies that each file specified in the `EXECUTABLES` environment variable exists and is executable (the job must fail if any of the executables is not there, or not executable).

This job must must be run in an `epitechcontent/epitest-docker` Docker container.

Running tests (because that's what heroes do)

Create a job with a `run_tests` ID that launches `make tests_run` at the root of the repository (this **step** must have a 2 minutes timeout).

This job must must be run in an `epitechcontent/epitest-docker` Docker container.

Pushing to the mirror repository

Create a job with a `push_to_mirror` ID that runs a mirroring to the repository specified in the `MIRROR_URL` environment variable.

It must use a secret named `GIT_SSH_PRIVATE_KEY` to specify the SSH private key to use.

This job must only be executed when a push is made to the repository.

Good practices

You can do things in lots of different ways with GitHub Actions, but some are better than others.

Here are some examples of good practices to follow:

- ✓ give **meaningful names** to your jobs and steps;
- ✓ use as much **GitHub Actions features** as possible before falling back to custom shell commands;
- ✓ use the **latest** actions' versions;
- ✓ when checking out, only fetch what is **necessary**;
- ✓ display missing or non-executable executables as **error annotations**.

This will be evaluated during the defense.

Bonus

Going further is amazing! Here are some ideas to improve your workflow:

- ✓ check for **banned functions**;
- ✓ add a `LIBRARIES` environment variable to check for the presence of libraries after compilation (useful for projects where you have to make libraries and no executables).

v 3.1