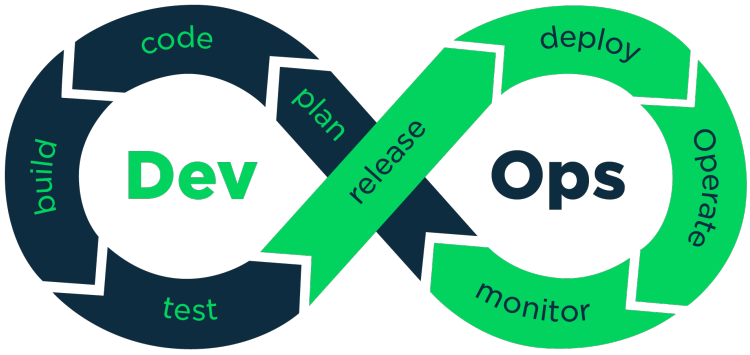
B2 - Introduction to DevOps

B-DOP-200

Chocolatine bootstrap

Eat a chocolatine and master GitHub Actions



1.0.1

Chocolatine bootstrap

During the course of the programming of your different projects, you often find yourself doing routine tasks, such as compiling and testing your program (if you are a conscientious programmer).

But have you found yourself in the situation where the one time you did not do these routine checks, you broke your delivery and threw weeks of work out the window? If you have, you are in the right place! And if you have not, well good for you, you are still in the right place. ;)

A good way to proceed, when it comes to important tasks that need to be done, is to **automate** them. And thus comes the notion of *automation*, where you put your trust into a computer's memory more than in your own.

Do you want to stop being surprised of a broken delivery? Do you want to increase the quality of your work? Do you want to make sure your GitHub repository is clean and attractive?
Then, it is time to take actions, to take **GitHub Actions**!

GitHub Actions

During this bootstrap, you are going to become familiar with this very handy automation tool, hard-wired into your GitHub repositories.

Step 0 – And… action!

Another little quiz to start with

As with every DevOps project you undertake, or will have to undertake, it is important to first understand the different notions and terminology you will encounter during your journey.
And thus comes the usual quiz by which you have to start your bootstrap. Answer the following questions with your friends:

- What is GitHub Actions?
- In the context of GitHub Actions, what is a *workflow*?
- In the same context, what is a *job*?
- On which occasions or *events* can a workflow be run?
- Why are chocolatines so tasty?

Installing GitHub Actions

No need to! GitHub Actions is automatically available for each GitHub repository, be they public or private.

You just need a simple file editor, preferably one that supports YAML syntax highlighting (your usual IDE is probably already supporting it).
Even better would be an IDE that supports GitHub Actions workflows syntax (Visual Studio Code for example).

Step 1 – Mirror mirror, who is the real repo?

Before coming to this page, you might have been tempted to see GitHub Actions for yourself on your Epitech repositories.
But wait, where are they?!

As of now, **GitHub Actions cannot be used on your Epitech repositories**.
So, one may ask, what is the point of GitHub Actions if we cannot use it on our repositories?
Well, the point is that it is still very useful, and that you can use the *official workaround*™ to harness the power of GitHub Actions for your Epitech programming life.

Thus comes the example we will use during our bootstrap, as a way to make you discover both how to use GitHub Actions, and how to set up the *official workaround*™: **repository mirroring!**

How the workaround works

The workaround works on the basis that GitHub Actions are still fully enabled for all your personnal, non-Epitech repositories.
As such, here is how it works:

1. You create a private **personnal** repository.
2. When a **push** is made to the **personnal** repository, the **same push** is then consequently made to your **Epitech** repository.

With this behaviour, the Epitech repository is always the exact copy of your private repository, which we also call a *mirror*.
You can now freely use GitHub Actions for your projects! Amazing!

And the best part is that you are going to set-up this system… with GitHub Actions. ;)

Task 1 – Repository creation

OK, time to get to work! Being registered to this bootstrap has provided you a repository dedicated to this bootstrap.
Since the bootstrap is not evaluated, we will use this repository to test our mirroring system (*official workaround*™, remember).

Create your own private repository in which you will set the system up. You are free to choose its name.

Task 2 – Go with the (work)flow

Time for the main course!
You are now going to define the behaviour that your (personnal) repository should have when you make a push to it. Such "behaviour" is a set of commands and actions that are to be executed when needed. With GitHub Actions, they take the form of what we call a *workflow*.

A workflow is a YAML file (yes, YAML again; and you will have even more YAML in the next projects of the track) that is placed directly into your repository, in a `.github/workflows` directory.

Create the `.github/workflows` directory in which you will place your workflow files.

Task 3 – What time is it?

Let us start with a very very simple task: write a simple workflow file named `what_time_is_it.yml` that, each time there is a push, starts a single job, with a single step, that runs the `date` command.

When you are done, commit your workflow, and push.
You should see, in the *Actions* tab on the repository's page, your first ever GitHub Action successfully run! Have you already felt that much happy to see a date? ;)

Task 4 – Get this workflow some files!

An important fact to remember is that **whenever a workflow is run, there are no files in the container it is run in**.

This means that, in order to use the files that actually are on our repository, we need to fetch them; or in our case, we need to *checkout*.

Fortunately, a pre-made module, called an *action*, allows us to do that effortlessly. And guess its name… *Checkout*!

Create a workflow file named `repository_size.yml` with a single job inside. Said job must, in order:

1. *Use* the `actions/checkout` action to fetch the repository's files.
2. *Run* an `echo` command to display the branch name on which the workflow is run.
3. *Run* the `du -skh --exclude=".git"` command.

GitHub Actions sets various environment variables by default, some of them might be handy. Go look at the documentation to find them all.

Task 5 – Mirror mirror, time to come to life!

You now know how to write a workflow, use an action, run commands, and interact with the files of your repository. Great!
You are now ready for the real challenge of this bootstrap, setting up the proper repository mirroring we talked about earlier.

Fortunately, there also exists an action, called… wait for it… *Mirroring Repository*!

Create a final workflow file named `repository_mirroring.yml`, in which you will use the aforementioned `pixta-dev/repository-mirroring-action` to mirror each of your pushes to this bootstrap's Epitech repository.

Since the container in which the workflow will run will need to perform Git operations beyond the repository associated to it, you will need to **create a new SSH key**, and make it available via a special, and **secure**, kind of environment variable: a *secret*.

Do not use the same SSH key as the one you use on your computer, as if it is compromised, attackers will get access to everything your SSH key is authorized to.
By keeping said repository SSH key limited in its scope and permissions, the consequences of it being compromised will be mitigated.

In conclusion

Congratulations for making it to the end! You now have an already handy workflow to help you mirror all your repositories in order to use GitHub Actions for your Epitech projects. Very cool indeed.

Now, it is time for you to start the project itself. You will have to go deeper into the features of the system, and everything you have done during this bootstrap will be of great benefit throughout the project.
You will need to do more steps by hands, as the actions you will be allowed to use are… limited.

Do not worry! You will still be able to use `actions/checkout`. ;)
But that's just an action, a GitHub Action! Aaaaand… cut.