Git branches

So far, you might have noticed that some of our Git outputs mention the branch “main”. In this section, you will discover what branches are, the purpose they serve as well as how to use them in your projects. We will also introduce the concepts of merging and rebasing and the role they play in the Git workflow.

What is a branch?

A branch can be visualized as a series of commits. As a matter of fact, in Git, every commit is identified by a unique code (SHA-1 code) and, every commit is connected to its previous commit (its parent). So far, we have only used a single branch. In order to know which branch you are working on, use the following command:

```
git branch
```

▼ So which branch have we been working on so far?

```
C:\Users\33611\Documents\Formations\Cap4Lab\git-workflow\git-course>git branch
* main
```

It is called the **main** branch. It is supposed to contain production-ready modifications only.

Note: historically, this branch was called **master**. However, the master/slave denomination is prone to hurting cultural sensitivity, hence the change. More about this here.

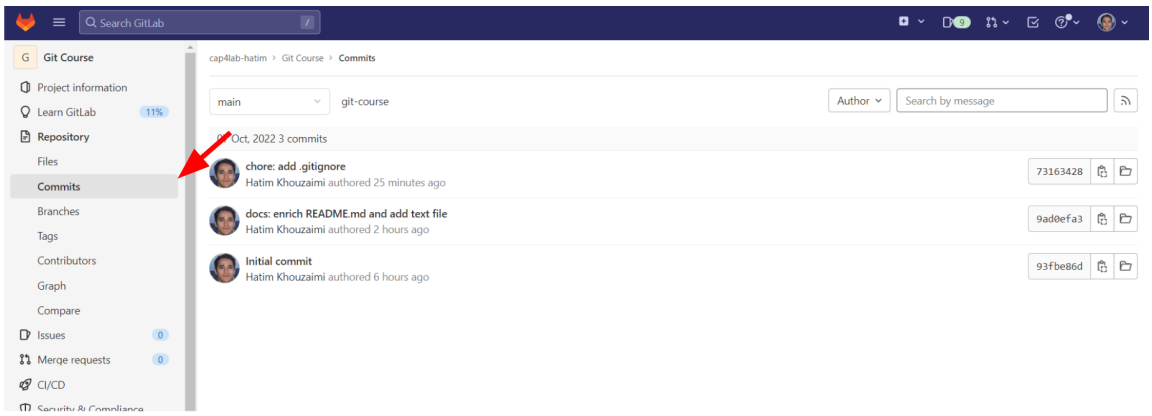
In order to view the different commits corresponding to your branch, use the following command:

```
git log
```

▼ Visualize the different commits of the main branch in your local repository:

```
C:\Users\33611\Documents\Formations\Cap4Lab\git-workflow\git-course>git log
commit 73163428c074f22e00d70bac567e3368e1dffa27 (HEAD -> main, origin/main, origin/HEAD)
Author: Hatim Khouzaimi <hatim.enst@gmail.com>
Date:   Fri Oct 7 18:30:30 2022 +0200

    chore: add .gitignore

commit 9ad0efa3c0563db26883bded95ce7ace458c8345
Author: Hatim Khouzaimi <hatim.enst@gmail.com>
Date:   Fri Oct 7 16:30:33 2022 +0200

    docs: enrich README.md and add text file

commit 93fbe86daffdd25060f8c3f71cd48f94ac297842
Author: Hatim Khouzaimi <hatim.enst@gmail.com>
Date:   Fri Oct 7 10:01:07 2022 +0000

    Initial commit
```

▼ You can also view the different commits for any branch in your remote repository, directly on GitLab. For that, go to Repository/Commits:



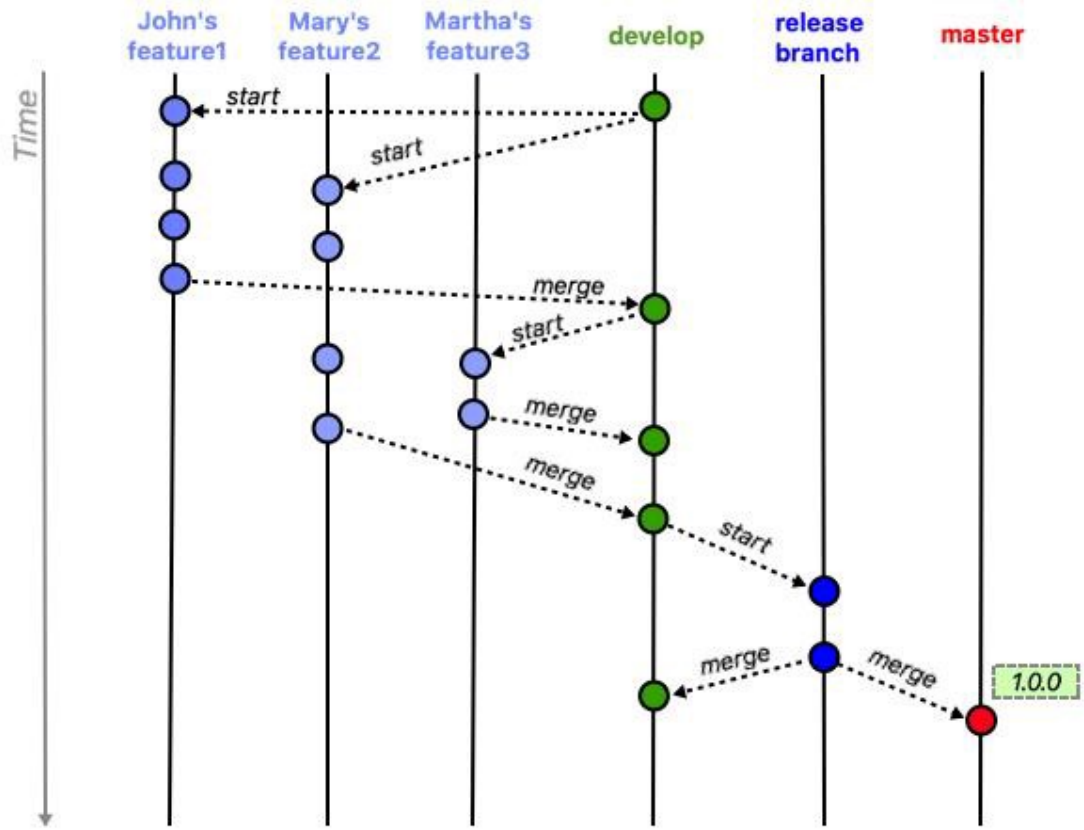
Why use multiple branches?

Imagine you are working with a team of a hundred developers, each one making one to two commits a day. If you are working on a single branch, you will see it evolve blazing fast. And several problems will arise:

- **Instability:** with each contribution, bugs can arise and they are harder to track
- **Difficult project management:** project managers have a hard time knowing which feature has been committed, tested…etc…
- **Difficult contribution:** developers step on each other's toes by committing potentially incompatible code
- **Communication problems:** since the code base is continuously evolving, it becomes harder to isolate the code block developers need to focus on during a meeting
- Among others…

The proper branching system to use

There are several branching strategies to choose from and these are different from project to project. In this course, we will adopt the following:



https://medium.com/empathyco/git-flow-applied-to-a-real-project-c08037e28f88

There are four types of branches to consider here:

- **main (formerly master):** it contains production-ready modifications. A commit in this branch is usually associated with a tag for a release version: 1.0.0, 2.3.1…
- **staging (or release branch):** for stable changes that still need some testing before going to production
- **development (here develop):** this is where developers contribute their work once they have tested it on their own
- **feature branches:** these are usually private branches where every developer works freely to develop a new feature

This way, the production version is only updated once the staging version has been approved and the developers can work in independent sandboxes. Merging the code from different developers is a process in itself, minimizing the risk of bugs and incompatibilities.

Creating new branches

Use the following command to create a new branch in Git:

```
git checkout -b your_new_branch_name
```

▼ Create a new branch called **staging** for our project:

```
git checkout -b staging
```

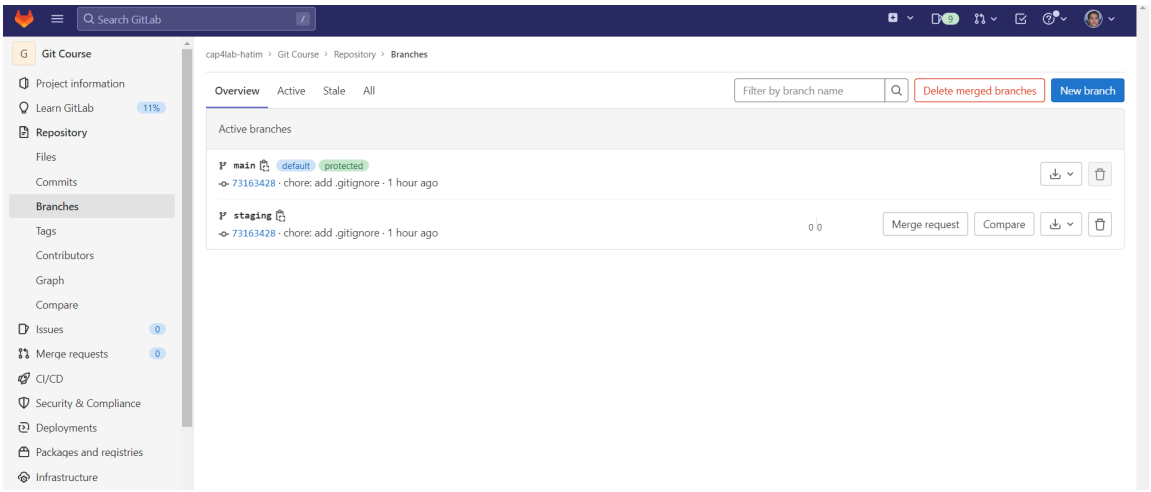
Remember two things at this stage:

- The newly created branch will “start from” the branch you are positioned into, in our case, the **main** branch.

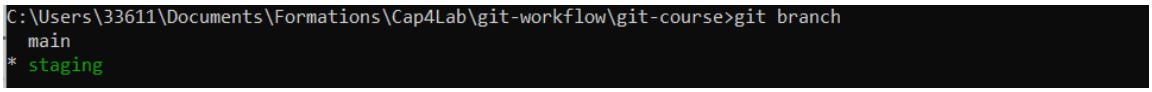
▼ This branch is only created locally, you still need to push it. Here is the command to do so, but you can only type git push and Git will remind you of the command:

```
git push --set-upstream origin staging
```

▼ Check that your newly created branch has been taken into account in your GitLab account:



▼ Check that you are positioned locally in your staging branch:



▼ Create a new **development** branch and push it:

```
git checkout -b development
git push --set-upstream origin development
```

Do not hesitate to check your development branch in your GitLab account. Now we are ready to tackle feature branches in the next section.