Merging branches

Merging branches

As a developer, the feature branch is your sandbox to explore, develop, and perform tests to make sure that your work is functional and compatible with the rest of the project (not breaking any previously available feature). When you think it is ready to be published in the project for others to access and use, you should merge it into the **development** branch. In some projects, it will be directly merged in the **staging** or even the **main** branch.

Note: merging in those branches is generally not performed by the developer him/herself. It goes through a validation process that we will discuss later.

In order to merge branches, you need to **git checkout** to the branch you want to merge into then use **git merge** while specifying the branch you want to merge from.

```
git checkout destination_branch
git merge source_branch
```

Fast-forward merging

Suppose you consider that you finished working on your feature branch and you consider it ready to be merged into **development**.

▼ Merge your **file1-update** branch into **development** (notice how the output mentions fast-forward):

```
git checkout development
git merge file1-update
```

▼ Check what **git status** shows you:

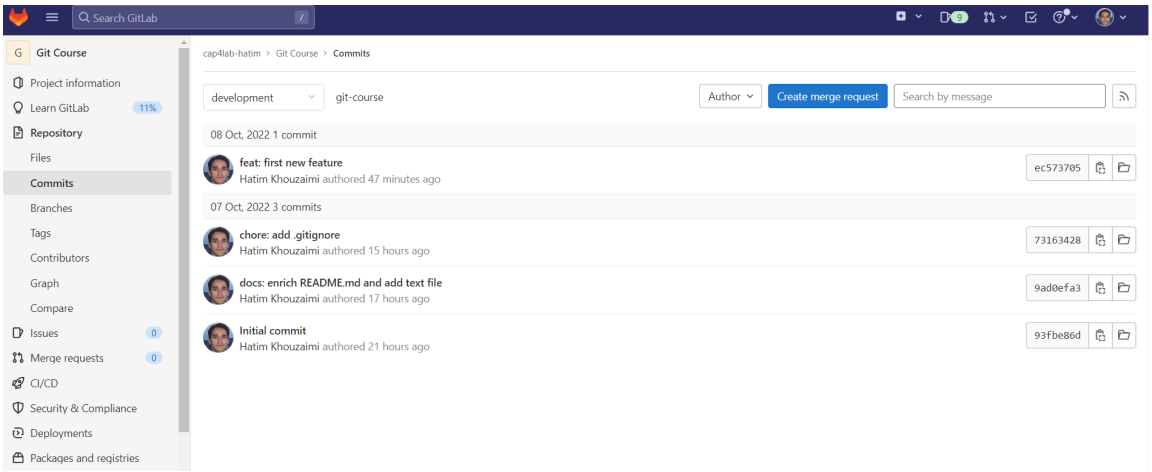
```
C:\Users\33611\Documents\Formations\Cap4Lab\git-workflow\git-course>git status
On branch development
Your branch is ahead of 'origin/development' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Indeed, you have merged your feature branch into development, but only locally. It is time to push the merge to your remote repository.

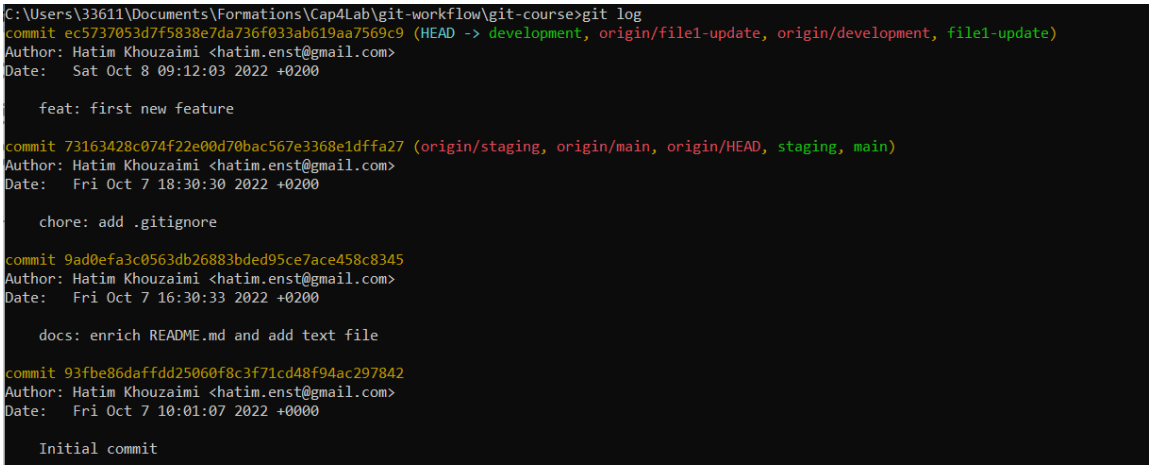
▼ Push the merge:

```
git push
```

▼ Check that your merge has been taken into account in the remote repository:



▼ Do not forget that you can also check your commit tree using the command line using **git log**:



Congratulations! You have just merged your first feature branch! However, while you were working on your feature branch, nothing happened in the development branch. Therefore, you have performed what is called a fast-forward merge. This might not be the case in projects with several contributors. Let us see what happens in the next section.

Non-fast-forward merging

Let us simulate the case where new commits are pushed to the development branch while we are working on our feature branch and see how this affects the merging process.

▼ Start by creating a new feature branch called non-ff:

```
git checkout -b non-ff
```

▼ Create a new file called **non-ff.txt** and add the sentence **“This modification has been added before the branch development was subject to an external commit.”** to it. Commit and push your work using the message “feat: in progress: non-ff”:

```
git add non-ff.txt
git commit -m "feat: in progress: non-ff"
git push --set-upstream origin non-ff
```

▼ Go back to your **development** branch:

```
git checkout development
```

▼ Add the sentence **“Yet another feature.”** to **file1.txt**. Then commit and push your work into development under the message “feat: external developer’s work”:

```
git add file1.txt
git commit -m "feat: external developer's work"
git push
```

▼ Go back to your **non-ff** branch:

```
git checkout non-ff
```

▼ Add the sentence **“This modification has been added after the branch development was subject to an external commit.”** to **non-ff.txt**. Commit and push your work using the message “feat: non-ff”:

```
git add non-ff.txt
git commit -m "feat: non-ff"
git push
```

▼ Go back to your **development** branch and merge it with **non-ff**:

```
git checkout development
git merge non-ff
```

▼ Visualize the development branch using **git log --oneline --graph**:

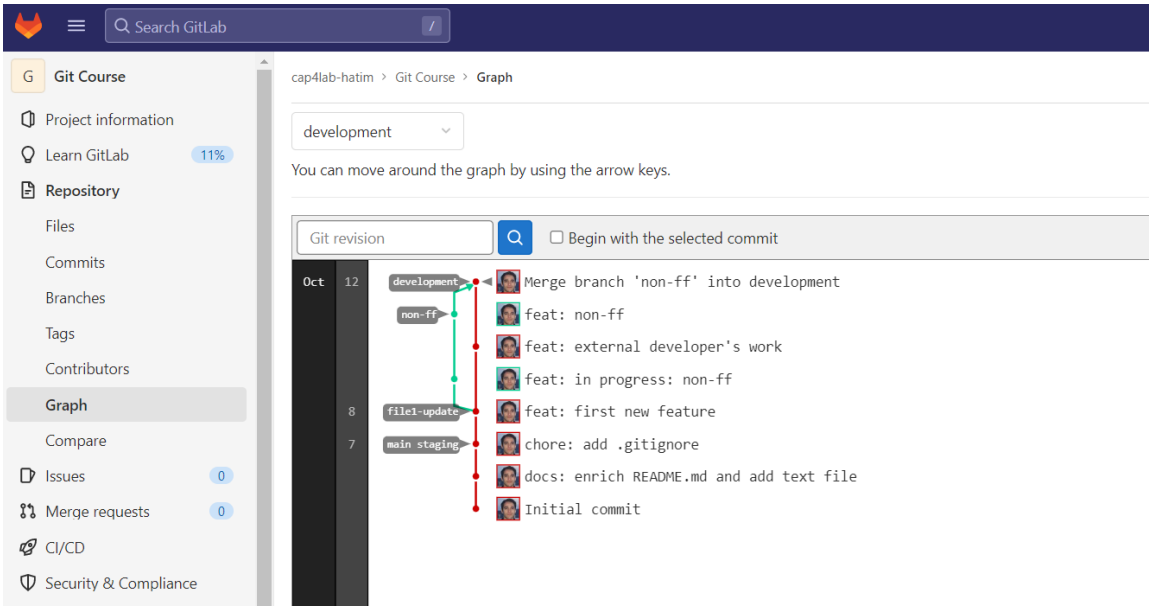
```
C:\Users\33611\Documents\Formations\Cap4Lab\git-workflow\git-course>git log --oneline --graph
*   cb87626 (HEAD -> development) Merge branch 'non-ff' into development
|\
| * d5a9716 (origin/non-ff, non-ff) feat: non-ff
| * 22e7844 feat: in progress: non-ff
* | 78be490 (origin/development) feat: external developer's work
|/
* ec57370 (origin/file1-update, file1-update) feat: first new feature
* 7316342 (origin/staging, origin/main, origin/HEAD, staging, main) chore: add .gitignore
* 9ad0efa docs: enrich README.md and add text file
* 93fbe86 Initial commit
```

Notice how the history of development is no longer linear. A new commit to represent the merge has been added and it has two parents, the previous head of the **development** branch as well as the head of the **non-ff** branch. Such a commit is not needed when nothing happens to the development branch while you are working in your feature branch. In this case, a **fast-forward** merge is performed meaning that the commits of the **feature branch** enter the history of **development**.

▼ So far, we have only merged those branches locally. Let us push the **development** branch to the remote repository:

```
git push
```

▼ Now go to the “Graph” section in your GitLab repository. There you can observe that the branches have been merged in a non-fast-forward fashion:



Keeping an accurate history of everything that happens, with all the feature branches and the merge process information has its advantages. It gives you the possibility to track changes in the most fine-grained fashion possible. Some developers even force a merge commit even in cases where a simple fast-forward merge strategy can be applied, just to keep metadata about the merge itself. You can achieve that by adding the **--no-ff** flag to your merge command:

```
git merge source_branch --no-ff
```

Nevertheless, keeping the full history with all commits comes with the price of a polluted log. The history is difficult to read and the real modifications one might look for are buried among a lot of details. Also, changes corresponding to a single feature become spread out among many commits. In the next section, we will see how to make it possible to maintain a cleaner commit history.