Mandatory Assignment 1

Welcome to the World of Git (5 Points)

University of Oslo - IN3110/IN4110

Fall 2021

1.1: Your First Git Repository (1 point)

Task: If you haven't already done it, install git (`https://git-scm.com/`). We already created a private assignment repository for you. Clone your repository. Now, create an `assignment1` folder in your git repository. Add a textfile to the `assignment1` directory. The textfile should be named `myfirstcommit.txt` and contain the string "This is my first commit.". Push your first commit to github.

1.2: Recovering Old Versions of Files (1 point)

With this task you will learn how to recover old versions of a file. Why do we need to know this? Well, sometimes when introducing a new feature you might realize that the original framework actually worked better for your purposes, or you made a mistake and need to debug.

Task: Add, commit and push a new file called `friendly_greeting.txt` containing a friendly greeting to your repository. Then change the file so the greeting is less friendly, and add, commit and push the modified file. Since the greeting is no longer friendly change the name of the text file `friendly_greeting.txt` to `less_friendly_greeting.txt`. Use the command

```
git mv friendly_greeting.txt less_friendly_greeting.txt
```

To fetch the old version, first use

```
git log
```

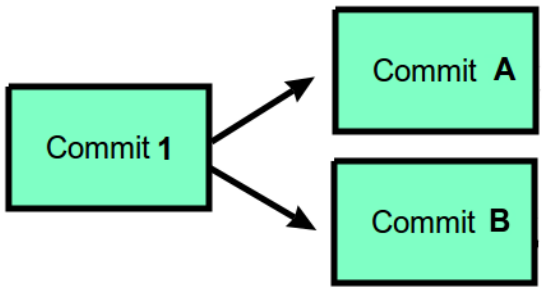
to get a list of the commits, and identify the commit containing the old version of the greeting, and note its commit hash (the string of letters/numbers after "commit " - basically the "name" of the commit). Then use

```
git checkout COMMITHASH friendly_greeting.txt
```

where `COMMITHASH` is the commit hash you found using `git log`. This will recover the friendly version `friendly_greeting.txt` from the commit you chose, and `git add` it for you. Finally, use `git commit` to make a commit which revcovers the friendly version, and push this commit to github. Now you will have a friendly and a not friendly greeting version in your repository

1.3: Dealing with Conflicts in Git (1 point)

During your career you will quite likely work with humans. Humans are different, which is fantastic in that sense that everyone brings something to the table. But it also means, that we might encounter conflicts. This exercise will use a lot of words to explain a common "error" which frequently happens when using git to work together. So it is actually a good idea to read through the whole exercise because that type of error will happen and git will complain, even if nobody does anything wrong. But on the sunny side understanding the reasons why git complains makes fixing its complaints a lot less frustrating[1].



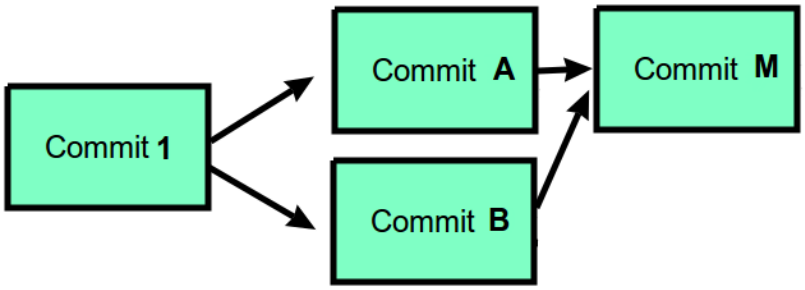
In git, each commit goes sequentially "on top of" another commit. This means that changes made in the commit are *intentional,*. However, if two people are working on a repository separately, and both make commits A and B locally and want to push them, git can get concerned[2]. Git will be perfectly happy to accept whichever of A and B is pushed first (let's say A is first, B is second), but when the other one is pushed, it will not be quite sure what to do with it, and throw an error which can look intimidating.

This is *by design.* While it would sometimes have been possible to just pretend commit B was really supposed to go on top of A - maybe A and B are modifying separate files - git won't do that by itself, because it is possible that this will have unintended side-effects. Instead, git will inform you that commit A was made "in between" commit 1 and the commit you are trying to make

[1] This is also good life advice.

[2] In a perfect world, each person having their own *branch* would be tidier and minimize the time spent resolving conflicts, but who's perfect?!

(commit B), and so won't allow you to push[3] until you have done a `git pull` to get commit A.



When you do `git pull`, git will see that you are trying to `pull` commit A which goes on top of commit 1. However, your local copy of the repository already has a commit B on top of commit 1, so completing the `git pull` means you have to `merge` first. This means that you have to make a new `git commit` M which does what commit B did, but on top of commit A instead of commit 1.

Sometimes this is very easy - maybe there is no conflict between B and A at all, and git will intelligently, as it was designed, do it for you. Sometimes, maybe A introduced some new stuff you need to adapt B to work with - maybe a function you use in B was renamed, so you need to change B a bit. In any case, once you have done what needs to be done and made commit M locally, it will basically fit on top of both B and A, and it can be pushed just fine.

Let's try to simulate such a scenario and fix the resulting merge conflict:

1. Add, commit, and push a new, file `gitconflict.txt` with the contents

   ```
   Here's a line

   Hello world!

   Here's another line
   ```

   to your git repository.

2. Copy the directory on your local machine containing your git repository to a different directory.

3. In the two different local copies of your repository, make different changes (For example, change the line "Hello world!" to "Hello world from A!" in one, and "Hello world from B!" in the other.) to `gitconflict.txt` and commit them separately.

[3] It's possible to tell git to do the push anyway. In case you didn't guess it yet: This is generally a bad idea.

4. Then, attempt to push the changes of both repositories to github. The first push will be fine, but the second one will fail and tell you the remote has stuff you don't. It will suggest you pull, so do that. This will cause a merge conflict which you will need to resolve manually. Your error will look something like this:

```
Auto-merging gitconflict.txt
CONFLICT (content): Merge conflict in gitconflict.txt
Automatic merge failed; fix conflicts and then commit the result.
```

If you type `git status`, you will see that git started merging stuff automatically, but didn't quite know what to do with `gitconflict.txt`, so git left it in a half-merged state. Opening the copy of `gitconflict.txt` in the second repository shows you something like this:

```
Here's a line

<<<<<<< HEAD
Hello world from B!
=======
Hello world from A!
>>>>>>> daa83f4b022b0b5b61a40fef6bb8eedacfe9fd5a

Here's another line
```

Notice that the line on which conflicting changes was made has been replaced by some autogenerated text from git. The stuff above `=======` is what commit B wants this line to contain, while the stuff below is what commit A wants this line to contain[4].

To finish the merge, simply modify the whole chunk to be whichever you want the 'final version' of the file to contain. (For example, maybe "Hello world from everyone!".) Then save the file.

You can then `git commit` your changes (You probably won't have to `git add` the file, because git did it for you.) and push them. Having done so, try typing `git log --graph` to see a visualization of what happened. Good job, you just resolved a conflict!

1.4: Creating A Pull Request (2 points)

In software development you will often work with a team on a specific project. To propose changes and collaborate on a project an important tool is the so-called "Pull Request". As you will in later assignments have to perform pull requests on the repositories of your peers, you will learn in this section already, how to do these.

[4]The text after `>>>>>>>` is the commit hash of commit A

Consider the following example: Sarah owns a project repository and Bill would like to suggest an improvement or new feature to that project. The procedure for Bill's code contribution consists of five steps:

1. Bill creates a *copy* of Sarah's repository. This copy is called a 'fork'.
2. Bill implements his improvements or new feature and pushes these changes to his 'fork'.
3. Bill sends a request to Sarah to include his changes. He does this by creating a 'pull request'.
4. Sarah reviews Bill's changes and can either reject or accept the pull request. If rejected, Bill can commit further improvement to his fork until Sarah is happy.
5. Once the pull request is accepted, Sarah merges the pull request. This means that Bill's code changes will be merged into Sarah's code repository.

The figure below visualizes these steps. Note that this procedure even works if Bill does not have write access to Sarah's repository.

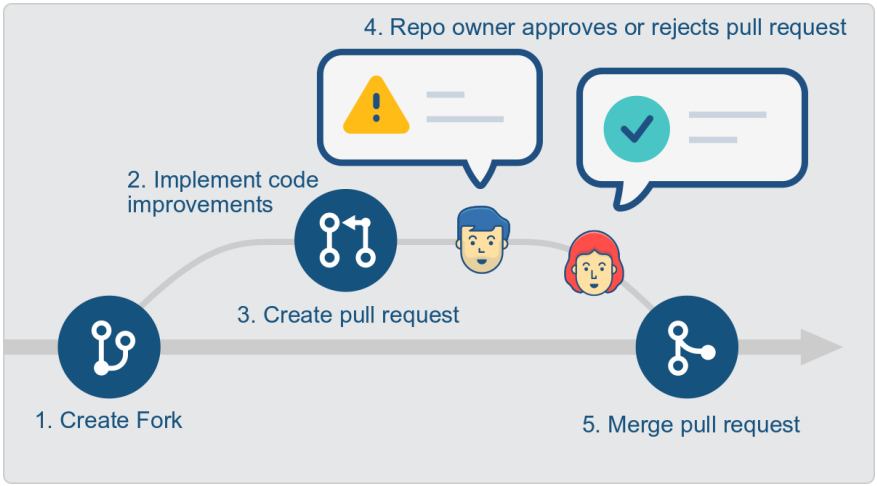
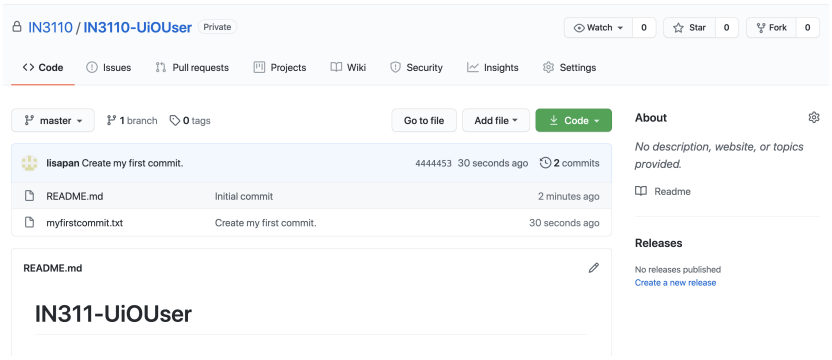


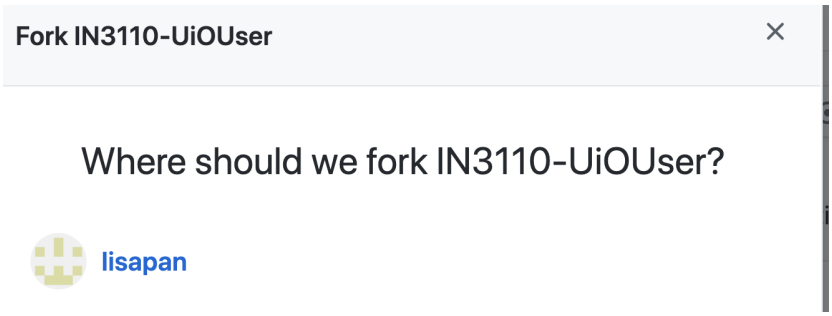
Figure 1: Steps in collaborative software development

In this assignment we will perform steps 1-5 of the steps described in Fig. 1. The steps you need to take are described in detail below.

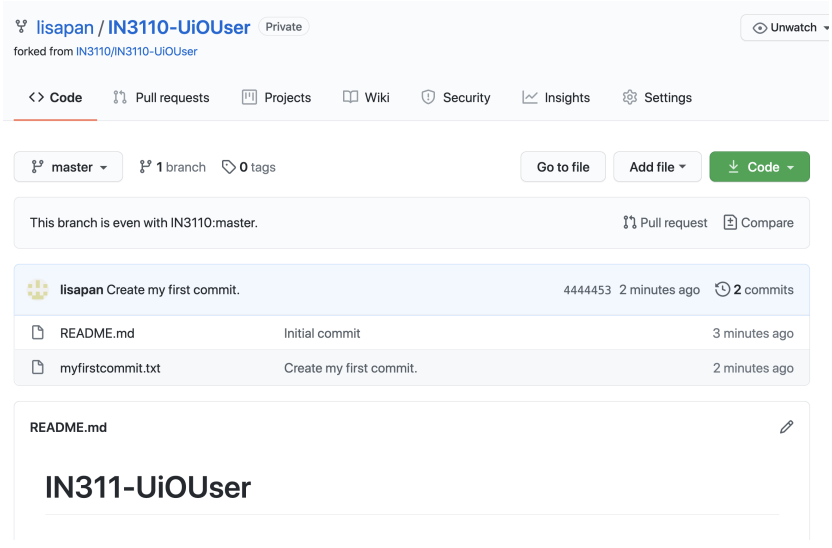
1. Go to your IN3110/IN4110 repository. Click on the 'Fork' button on the top right to create your personal copy of the repository.



This could open up a dialogue window asking where to fork your repository to. Make sure you choose your personal "UiOUser name".

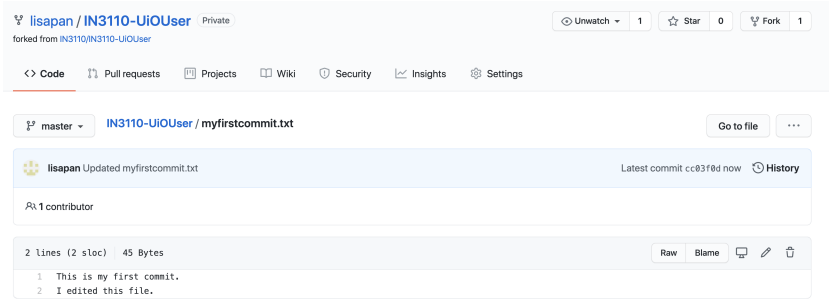


2. Once forked, you will be forwarded to the repository page of your personal copy.

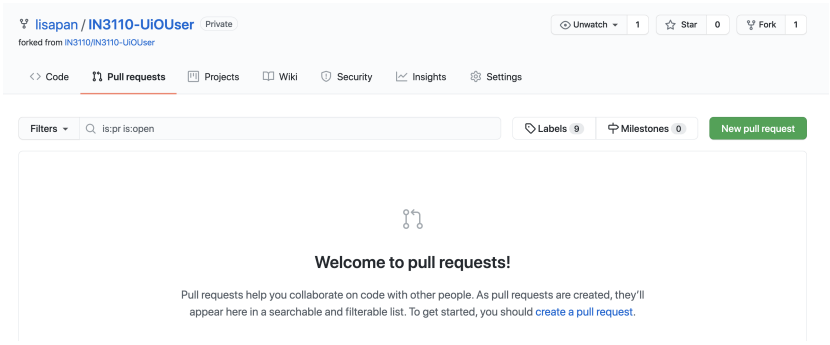


You can clone this repository as usual to your computer. [5]

3. Modify the textfile `myfirstcommit.txt` in your forked repository by adding a line to it saying "I edited this file.". Commit and push these changes to your forked repository. **Note:** Don't forget to write a commit message mentioning what you changed. Your modified file in your forked repository will look like this.

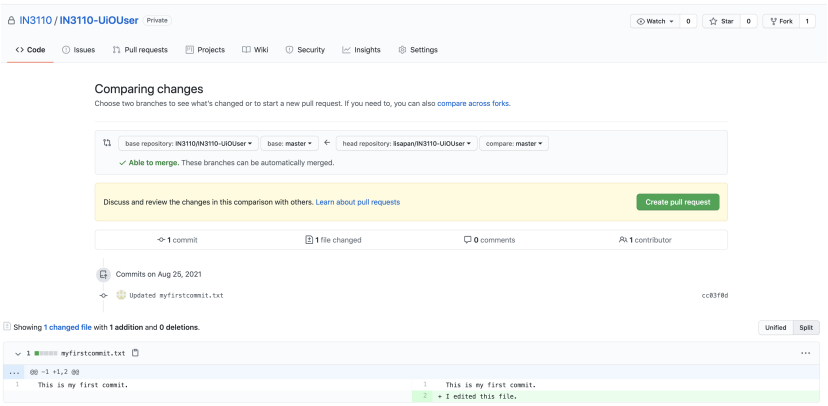


4. It is time to create a 'pull request'. Click on the flag "Pull requests" in your forked repository and click on the green button "New pull request".

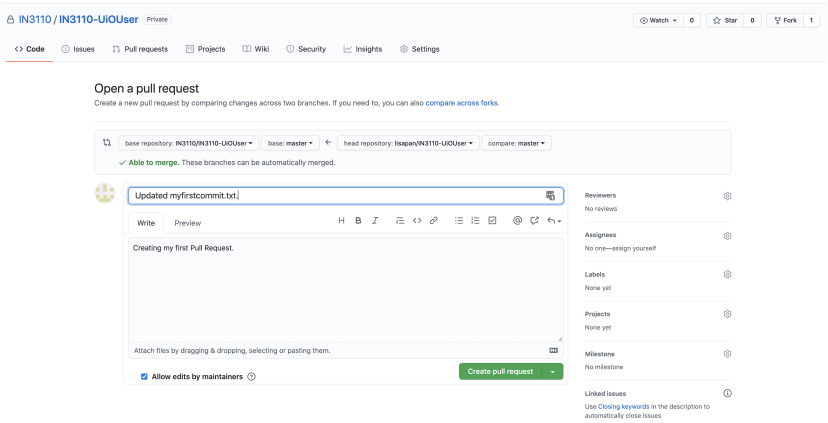


Make sure that the base repository is the "IN3110/IN3110-UioUser" repository. Merge your modifications from your fork into that repository (See image below).

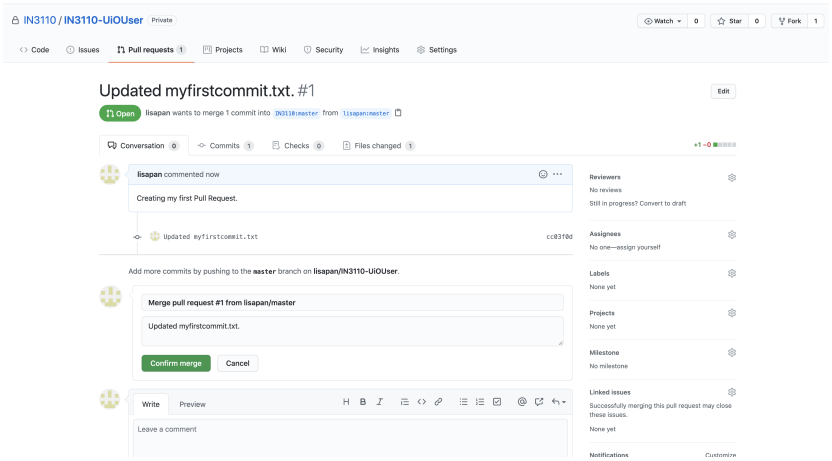
[5] `https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository-from-github/cloning-a-repository`



Leave a short descriptive message with the pull request as well as a short comment describing what you did (see image below). Think of it like leaving a message to your co-worker :)). Push the button "Create pull request".



5. You will now see an open "Pull request" at the "IN3110/IN3110-UioUser" repository. To close the pull request, you can merge it. Finish the pull request by confirming the merge.



Congrats! You finished your first pull request! You should be able to see a "closed" pull request under the flag "Pull Requests" as below.