

Git collaboration - a practical session

In this practical session, you will:

- Understand basic Git concepts and terminology.
- Configure Git on your machine.
- Set up a GitHub repository.
- Perform basic Git operations: commit, fetch, branch, merge, and push.
- Resolve merge conflicts collaboratively.

Before we start

Requirements

These exercises are designed to be done in pairs.

Each will need

- A laptop with git installed
 - [“Installing Git”](#)
- A GitHub account with SSH keys installed
 - [“Getting started with your GitHub account”](#)
 - [“Adding a new SSH key to your GitHub account”](#)

You will be asked to open a terminal (also called a “shell”) to use git commands.

- On Windows, you can use the “Windows Console” (`cmd.exe`), or on more recent version versions of Windows either “PowerShell” or the built-in “Terminal” application.
- On MacOS, you can use the built-in “Terminal” application.
- On Linux, use the terminal emulator of your choice.

Configure git Before we start, we need to configure git. Use the following commands (replace with your own information).

```
git config --global "user.name" "First name Last name"
git config --global "user.email" "firstname.lastname@example.com"
```

You should also make sure that the default name for the main branch is “main”.

```
git config --global "init.defaultBranch" "main"
```

By default, the editor used in the terminal could be “vim”, which can be a bit rough for beginners.

On Windows, change it to “notepad.exe”

```
git config --global "core.editor" "notepad"
```

On Linux and MacOS, change it to “nano” (read about [nano](#)).

```
git config --global "core.editor" "nano"
```

A few tips

- Use the command `git status` before and after every `git` command to see what is happening.
- Before asking any question, make sure you’ve looked at what `git status` says.
- Have I told you about the command `git status`? Really, it is useful.
- You can print the entire graph with the command `git log --all --graph --oneline`.
- Don’t hesitate to look for answers on Google or ask questions to ChatGPT.

Warnings

- Be very careful whenever you use `-f` or `--force` as an option of any command! Any forced operation might delete data permanently.
- Although you will create and edit Python files, you do NOT need Python for this session.

Links and references

- GitHub's git cheat sheet: <https://education.github.com/git-cheat-sheet-education.pdf>
- Official git documentation: <https://git-scm.com/docs>
- Atlassian's git tutorial: <https://www.atlassian.com/git/tutorials/setting-up-a-repository>

Agenda

The goal of this practical session is to demonstrate how git enables collaborative work by allowing you to share your contributions and to manage conflicts.

You will learn

- How to make commits;
- How to fetch commits made by others;
- How to create a branch and switch between branches;
- How to merge your changes with others;
- How to share your commits.

Collaboration using git

Setting things up

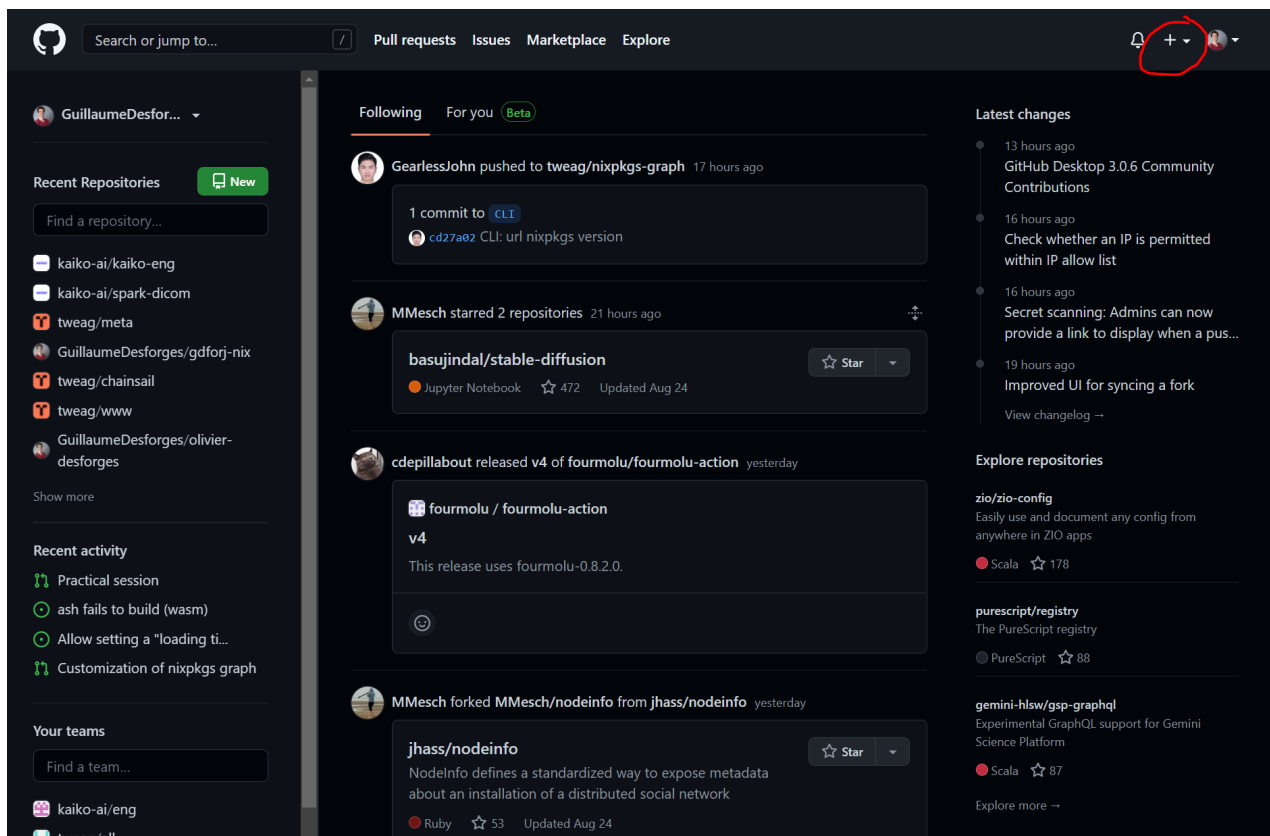
Collaboration requires a remote repository. We'll be using GitHub to get one for free.

<https://github.com/>

Create a GitHub project This has to be done by only one of the two partners, as it will be shared.

Check that you are logged in to GitHub!

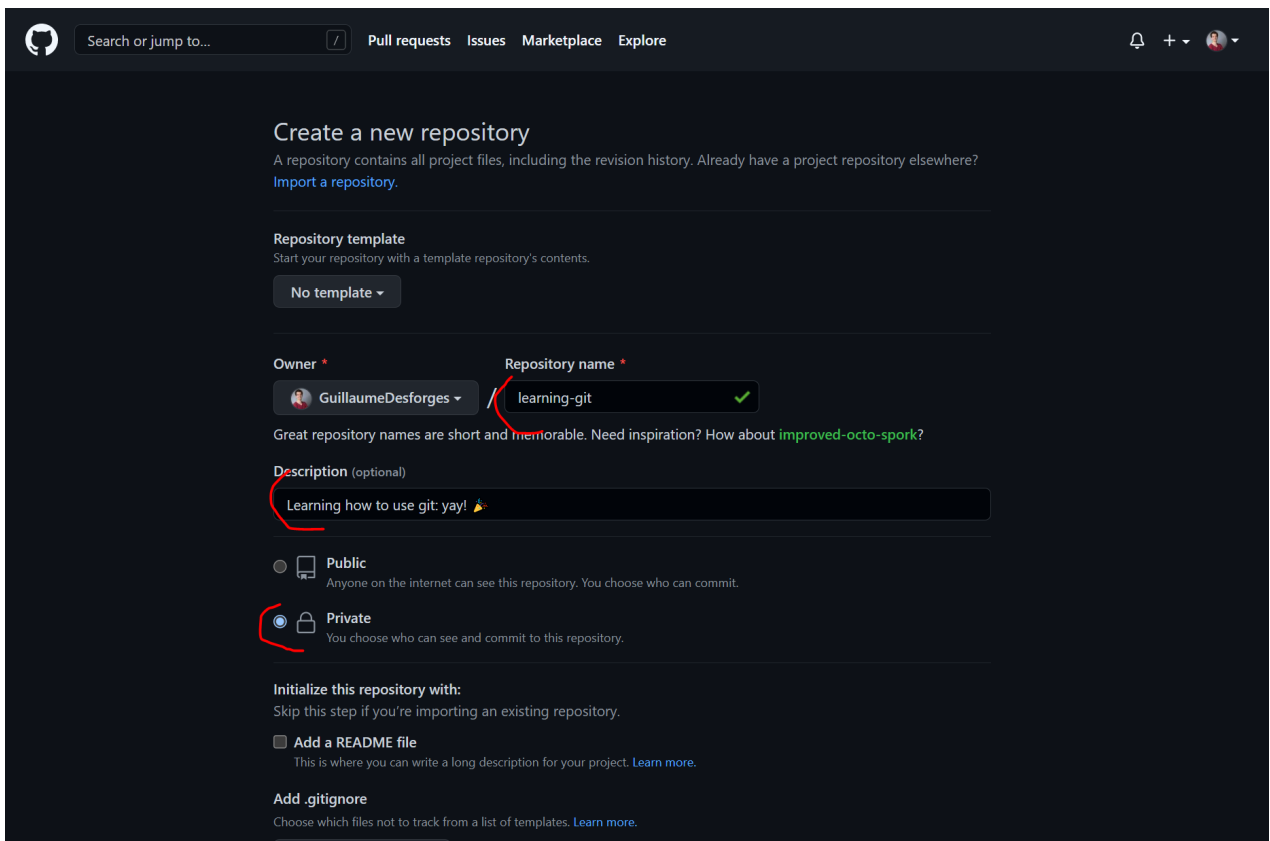
1. Create a repository
- Click on “+” in the top right corner



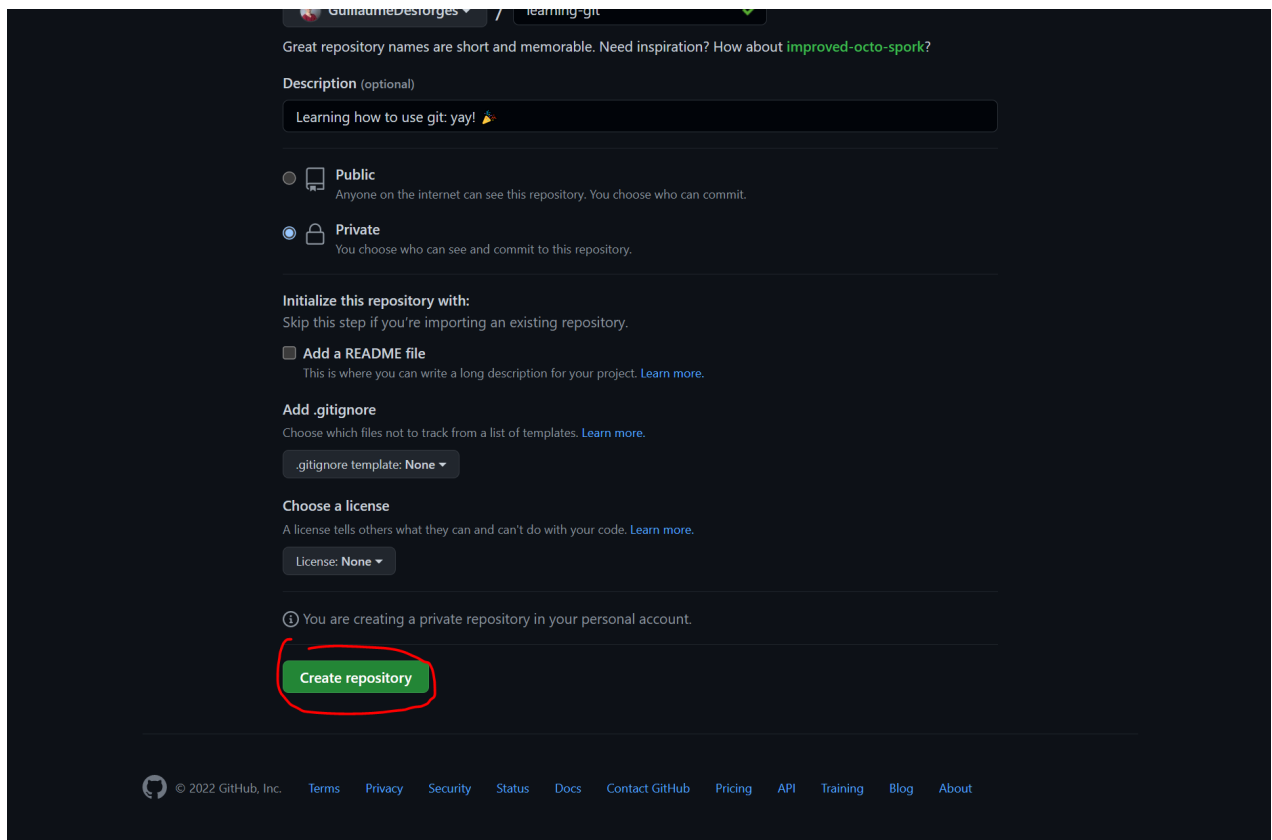
- Click on “New repository”



- You can name your repository however you want > Usually, we use lowercase characters (a, b, c, ...) separated by hyphens - instead of spaces. > Example: if your project is name “My awesome Git project”, name your repository `my-awesome-git-project`.
- Write a description > Usually a single sentence.
- Select “Private”

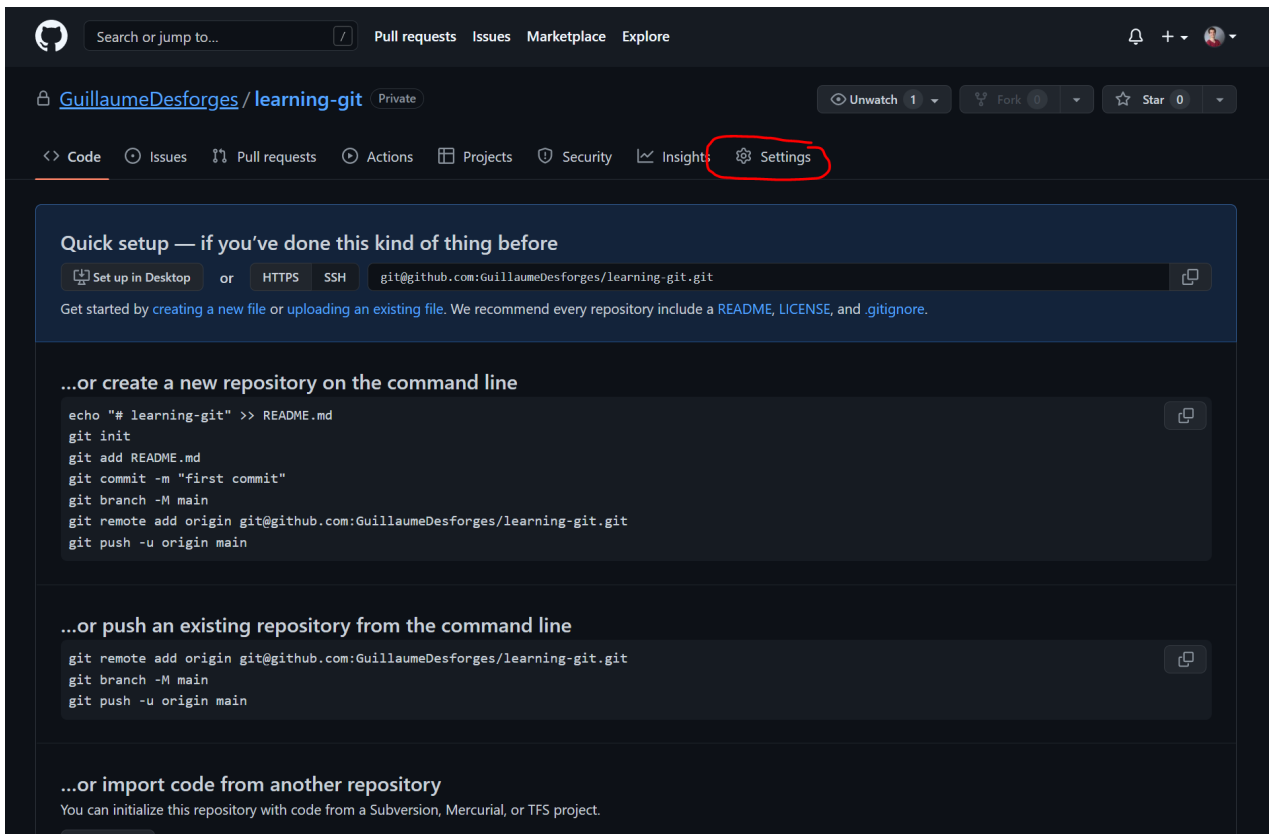


- Click on the “New repository” button at the end

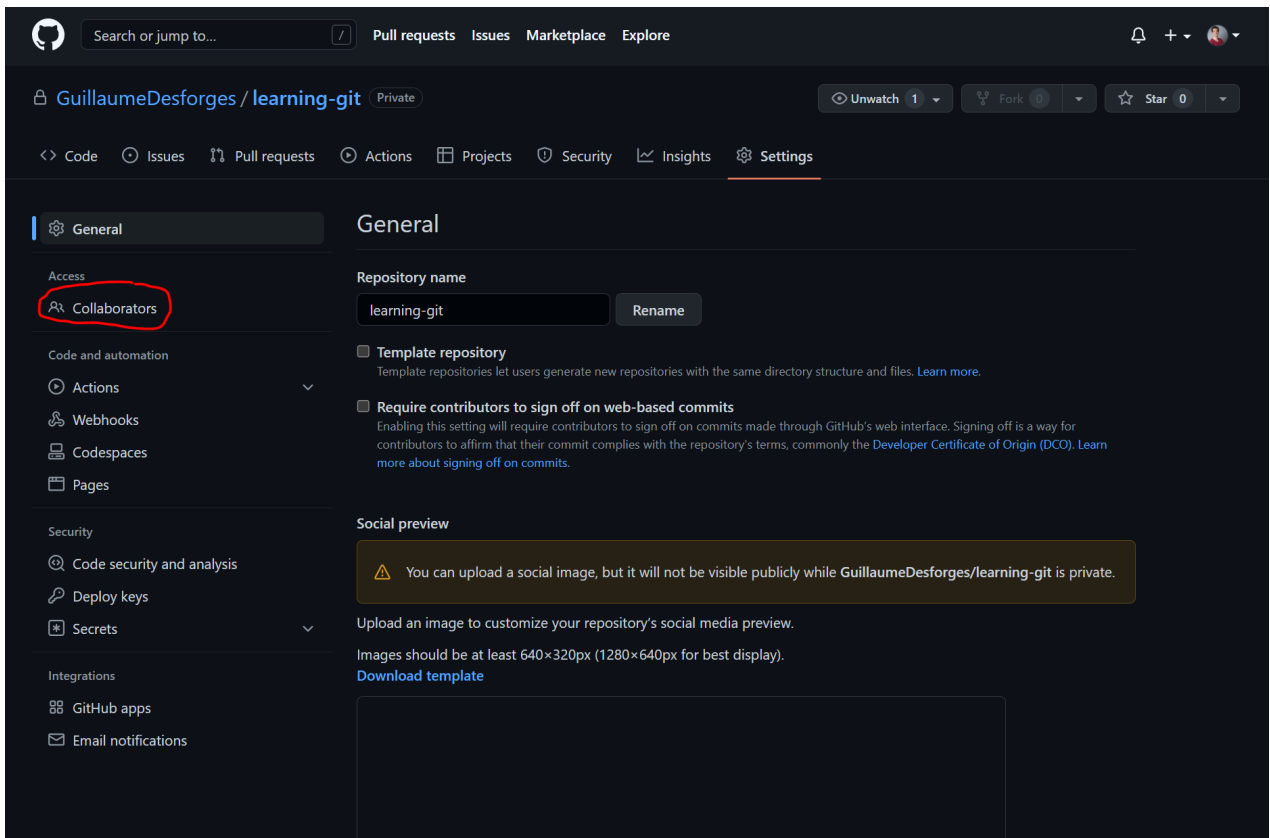


2. Give access to you partner

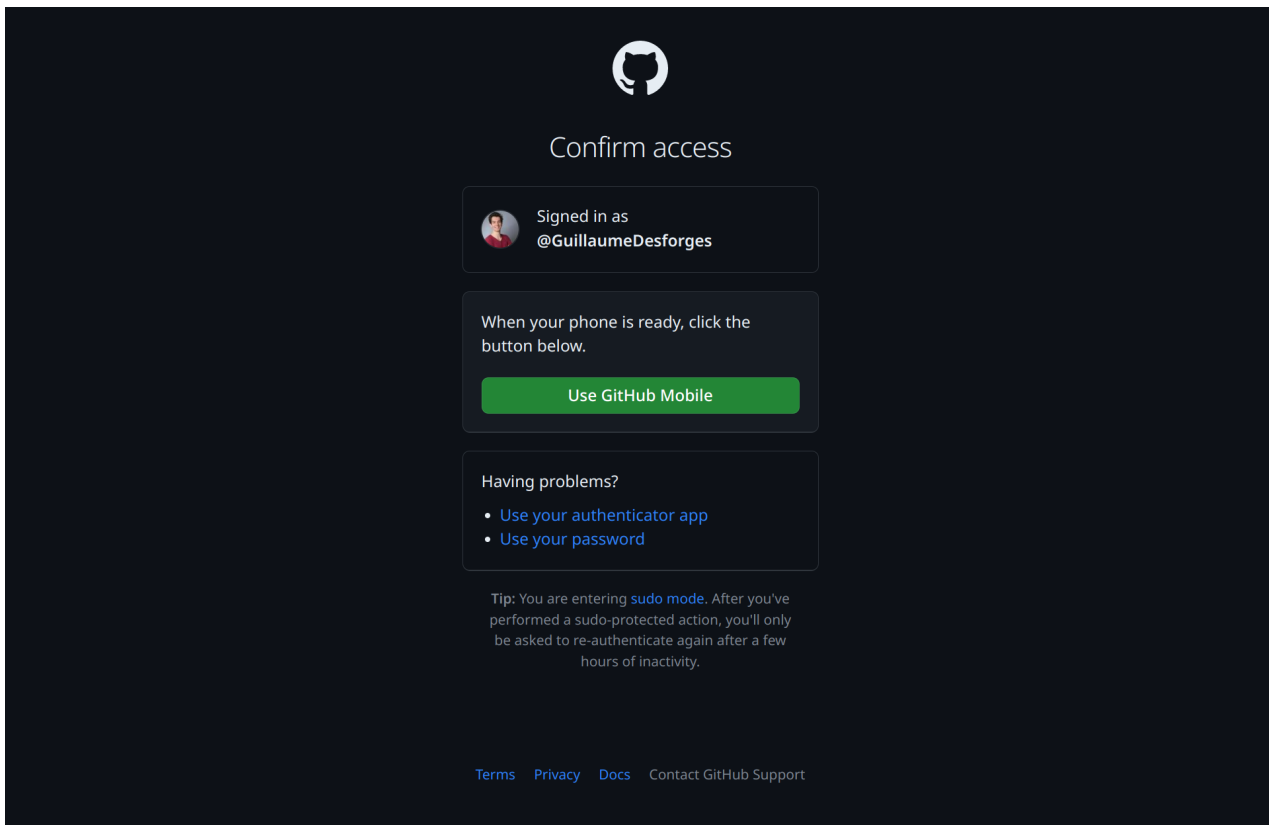
- Go to the web page of your new repository (if you just created it, you should be on it already)
- Go to the repository settings ("Settings" tab)



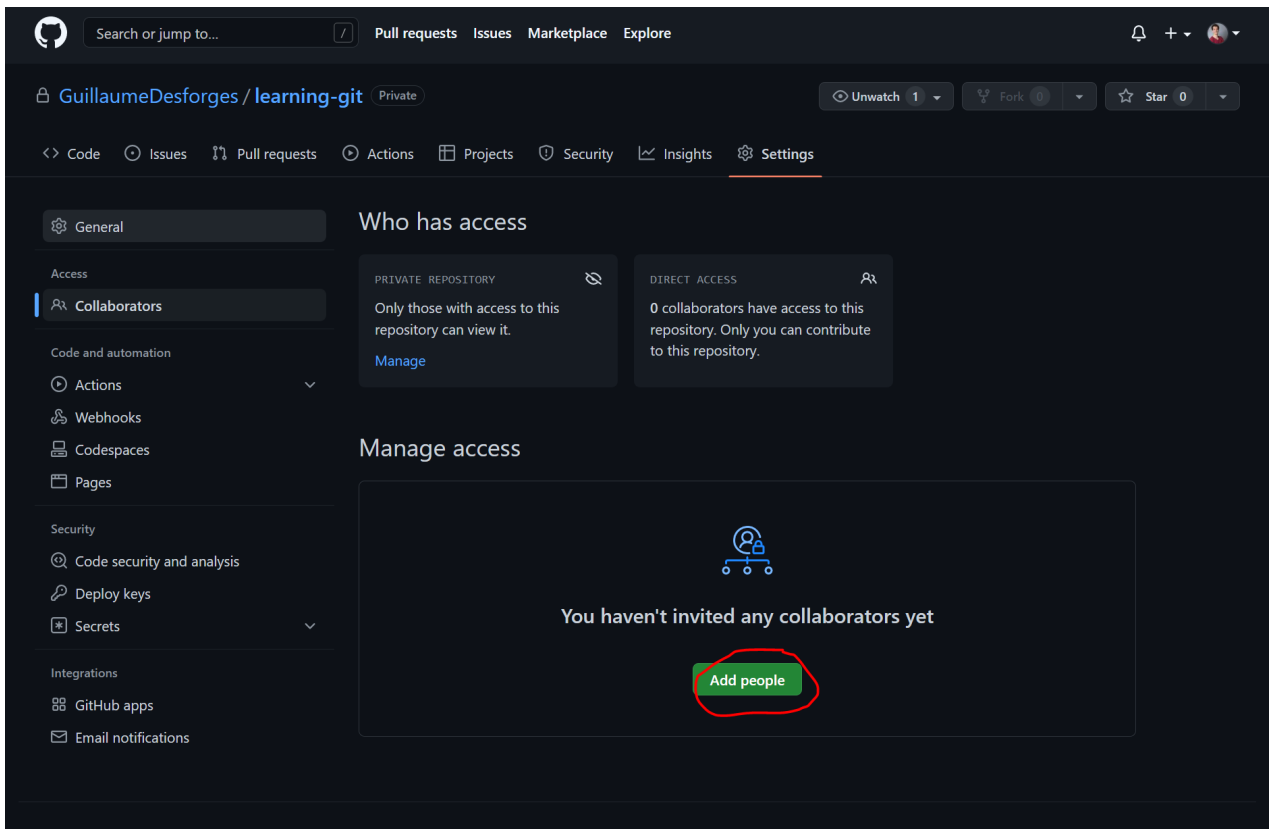
- Go to the “Collaborators” tab (under “Access” group in the left panel)



You might be required to authenticate via MFA (if not, skip this)

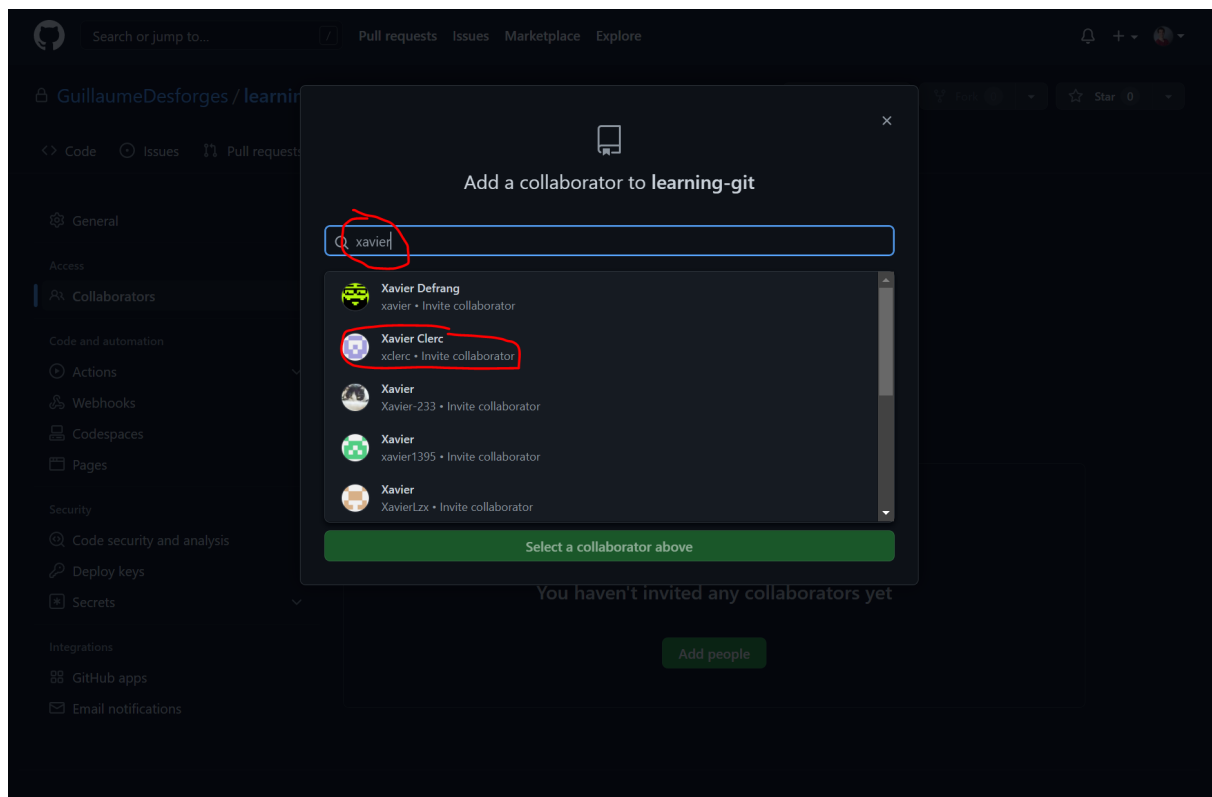


- Click on “Add people”

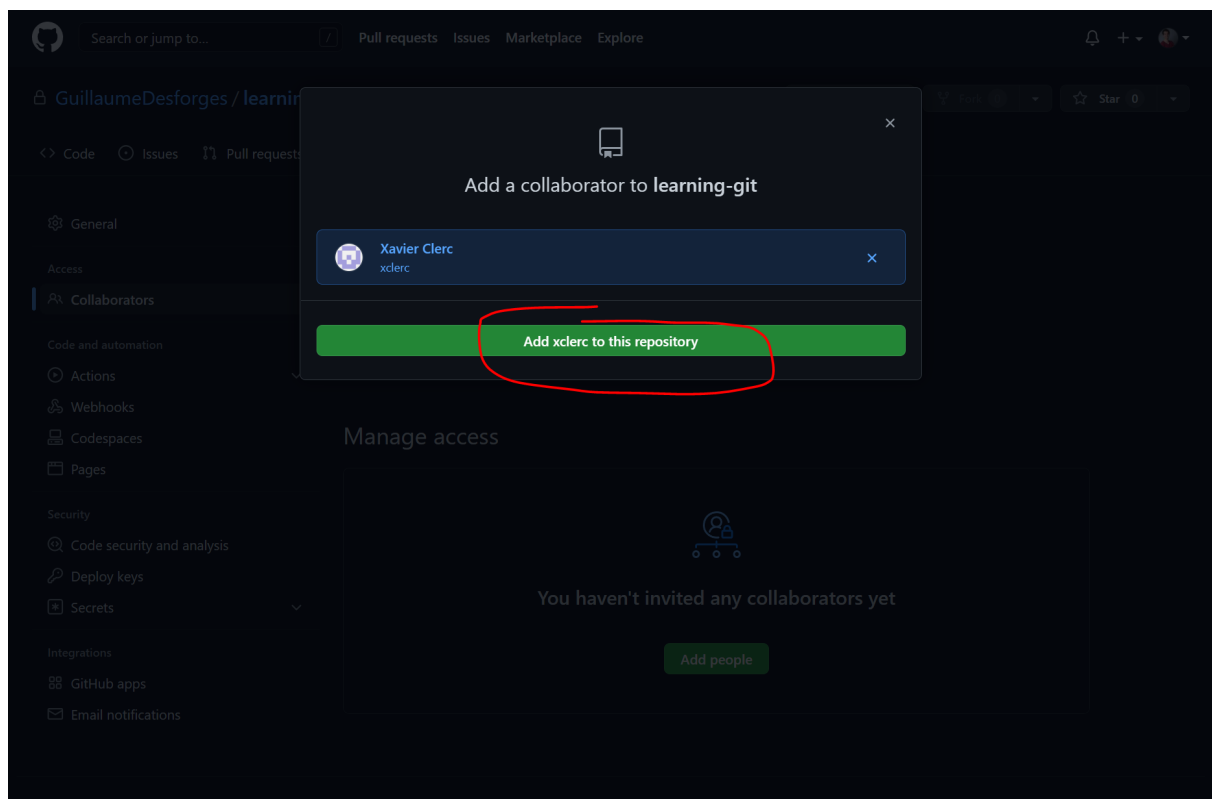


- Find and add your partner

- Type their GitHub “handle” in the text input
- Suggestions should be shown, click on the right one



- When found, click on the button below “Add XXX to this repository”



Create a local copy of the repository on your laptop On the repository web page in GitHub, as the project is empty for now, you'll find a "Quick setup" banner. Click on the "SSH" button in there, you should see something in the form of:

```
git@github.com:yourHandle/your-repo-name.git
```

Copy this URL, and clone it to your laptop, for example:

```
git clone git@github.com:yourHandle/your-repo-name.git
```

Where `your-repo-name` is replaced by the name of your own repository.

A folder must have been created with this name (e.g. `your-repo-name`).

Change the shell [working directory](#) to it using the command `cd`.

```
cd your-repo-name
```

Where `your-repo-name` is replaced by the name of your own repository.

Congratulations! You just "cloned" the repository from the remote (GitHub). Right now this folder is empty, but soon enough we'll fill it with some files.

If you have enabled "See hidden files" option, notice the `.git` folder.

git stores all the data it needs to manage the repository in this `.git` folder. If you delete it, everything you haven't pushed will be lost forever!

Hands-on

Now, it's time to start for real!

We'll go through several scenarios which you will often encounter.

One person will play the role of "computer 1" and the other the role of "computer 2".

Though the examples might seem trivial, pay special attention to understand what is happening.

First scenario: collaboration without conflict We'll start with the simplest.

In this scenario, you will learn how to create a new file, commit changes locally, and synchronize them with a remote repository without encountering conflicts. You will also understand how to fetch changes made by collaborators and integrate them into your local repository.

- On computer 1
 - Create a new file in the folder of the repository, with the following Python script

```
print("Hello world")
```
 - Commit this new file to your local history
 - Push this commit to the remote repository

You can check that the remote repository was properly updated by refreshing the web page of your repository on GitHub.

- On computer 2
 - Fetch these changes, check you have the proper `script.py` file locally
 - Edit `script.py`:

```
print("Hello world")
print("This is me")
```
 - Commit these changes
 - Push this commit to the remote repository
- On computer 1
 - Fetch these changes, check that you have the updated `script.py` file locally

Questions

1. Why do you need to commit your changes locally before pushing them to a remote repository?
2. How can you make sure that both collaborators have the latest version of the project?

Second scenario: collaboration with conflict In this scenario, you will learn how to handle situations where two collaborators make conflicting changes to the same file. You will practice resolving merge conflicts, ensuring that both sets of changes are preserved and properly integrated into the project.

On the same repository as the first scenario:

- On computer 1
 - Edit `script.py`:

```
print("Hello world")
print("This is me")
print("Life should be")
```
 - Commit this change
 - Push this commit to the remote repository
- On computer 2
 - Edit `script.py`

```
print("Hello world")
print("This is me")
print("Fun for everyone")
```
 - Commit these changes
 - Try to push this commit to the remote repository... it should fail!
 - Fetch the changes from the remote repository and resolve conflicts in order to have both lines added
 - * See complete instructions on GitHub: [“Resolving a merge conflict using the command line”](#)
 - Push this commit to the remote repository
- On computer 1
 - Fetch these changes, check that you have the updated `script.py` file locally

Questions

3. What caused the conflict in this scenario? How did Git help identify it?
4. What does it mean to “resolve” this conflict?
5. What could go wrong if conflicts are resolved incorrectly?
6. After resolving the conflict, what steps did you take to ensure that all changes were correctly integrated?

Third scenario: use a Pull Request In this scenario, you will learn how to use Git branches to manage separate lines of development. You will also learn how to create a pull request, review code changes collaboratively, and merge approved changes into the main branch. This process emphasizes code review and collaboration in a team setting.

On the same repository

- On computer 1
 - Create a new branch
 - Edit `script.py`

```
print("Hello world")
print("This is me")
print("Life should be")
print("Fun for everyone")
print("Hello world")
print("Come and see")
```
 - Commit these changes
 - Push this commit to the remote repository
 - Go to GitHub web page, create a “Pull Request” from this new branch to the main branch

- * See complete instructions on GitHub: “Creating a pull request”
 - Assign your partner as a “Reviewer”
- On computer 2
 - Check out your notifications on GitHub (<https://github.com/notifications>)
 - Go to the Pull Request
 - Add a comment to the requested changes
 - * Go to the “File changed” tab
 - * Select multiple lines from the newly added lines
 - Click on liner number “5”
 - Hold “Shift” (or “Maj”) and click on line number “6”
 - * Move your mouse cursor to code of the last selected line, a blue “+” button should appear on its left
 - * Click on the blue button, a new text box should appear
 - * Write your comment in the text box about the changes you would like the other to make
 - * Once done, click on “Start review”
 - * For now, the comment has not been posted. You can make multiple comments before posting them.
 - * Once you have made all your comments, click on “Review changes” on the top right, then “Submit review”
- On computer 1
 - Check out your notifications on GitHub
 - Check out the review, reply or apply requested changes
- You can repeat this process as much as you’d like, until both parties are satisfied
- On computer 1
 - Click on “Merge Pull Request”
 - Pull the changes of the main branch
 - Delete the branch of the Pull Request
 - * on the remote repository
 - * locally

questions

7. What are the advantages of using a pull request for managing changes in a collaborative project?

Fourth scenario: rebase vs merge In this scenario, you will learn the difference between merging and rebasing in Git. You will practice rebasing a branch onto another, resolving conflicts that arise.

On the same repository

- First read the [rebase documentation](#), especially the first and last sections.
- On computer 1
 - Create a new branch named `sum`
 - * create a file `sum.py` with the following content


```
import numpy as np
import time

def sum(y):
    res = 0
    for yi in y:
        res += yi
    return res

def main():
    y = np.random.rand(100000000)
    start = time.time()
```

```

sum(y)
end = time.time()
print(f"Elapsed time: {end - start}")

```

```

if __name__ == "__main__":
    main()

```

- Check that the code runs
- Commit these changes
- Push to the remote repository

- On computer 2

- Fetch previous changes
- Create another branch **faster-sum** which starts from the **sum** branch (i.e. checkout **sum** then create the branch)
- Modify **sum.py** by adding the following function:

```

def faster_sum(y):
    return np.sum(y)

```

- Check that the code still runs
- Commit these changes
- Replace the **main** function by

```

def main():
    y = np.random.rand(100000000)

    start = time.time()
    sum(y)
    end = time.time()
    python_time = end - start
    print(f"Python sum: {python_time} s")

    start = time.time()
    faster_sum(y)
    end = time.time()
    numpy_time = end - start
    print(f"Numpy sum: {numpy_time} s")

    print(f"Numpy runs {python_time / numpy_time} times faster!")

```

- Check that the code still runs
- Commit these changes
- Push to the remote repository

- On computer 1

- **On branch sum**
 - * Modify the whole file as follows:

```

import numpy as np
import time

def sum(y):
    """Return the sum of all elements in array `y`."""
    result = 0
    for yi in y:
        result += yi

```

```

    return result

def main():
    y = np.random.rand(100000000)
    start = time.time()
    sum(y)
    end = time.time()
    print(f"Elapsed time: {end - start} seconds")

if __name__ == "__main__":
    main()

```

- * Check that the code runs
- * Commit and push changes

- On computer 2
 - Fetch all recent changes
 - Rebase **faster-sum** on **sum**
 - * there should be some conflicts due to modification of the same line by both branches
 - * follow git-printed instructions to solve the conflict (**git status** usually helps)
 - Check that the script still runs as expected
- On computer 1:
 - Review and merge the branch into **main**

Questions

8. How did rebasing affect the commit history compared to merging? Do you see any benefit?
9. In what situations might one approach be preferred over the other?

Fifth scenario: stashing changes In this scenario, you will learn how to temporarily save uncommitted changes using Git stash, and how to apply or discard these changes later. This is useful when you need to switch branches or work on something else without losing your current work.

- On computer 1:
 - Start by creating a new branch **feature-A** from the **main** branch
 - Modify **script.py** by adding a new function: `python def new_feature(): print("This is a new feature")`
 - You now need to switch back to the **main** branch Use **git stash** to temporarily save your changes
 - Switch to the **main** branch
 - Make some changes in **main**, commit the changes, and push them to the remote repository
 - Switch back to the **feature-A** branch and reapply your stashed changes
- On computer 2:
 - Review and merge **feature-A** into **main**

Questions

10. Why might you use **git stash** instead of committing incomplete changes?
11. How does stashing help prevent conflicts when switching branches?
12. What is the difference between **git stash apply** and **git stash pop**?

Sixth scenario: cherry-picking commits In this scenario, you will learn how to cherry-pick specific commits from one branch and apply them to another branch. This is useful when you need to apply a bug fix or feature to multiple branches without merging.

- On computer 1:
 - Create a new branch **feature-B** from **main**
 - Add a bug fix to **script.py**: `python def bug_fix(): print("Critical bug fixed!")`
 - Commit and push these changes

- Add a new feature to `script.py`: `python def feature_c(): print("Feature C is now available!")`
- Commit and push these changes
- Oh no! We realized that the bug fix is critical and needs to be applied to the `main` branch immediately without merging `feature-B`!
- On `main`, use `git cherry-pick` to apply the bug fix commit to `main`, but not the feature
- Push the changes to `main`
- On computer 2:
 - Review and merge `feature-B` into `main`

Questions

13. What are some scenarios where cherry-picking is more appropriate than merging?
14. How does cherry-picking affect the commit history, and what should you be careful about?
15. What are the potential risks of cherry-picking, especially in larger projects?

Seventh scenario: rewriting history with rebase In this scenario, you will learn how to use Git rebase to rewrite commit history by renaming a commit message. This is useful for correcting mistakes or clarifying commit messages after they have been made.

- On computer 1
 - Create a new branch named `feature-C`
 - Create a file `multiply.py` with the following content:


```
def multiply(a, b):
    return a * b

if __name__ == "__main__":
    result = multiply(3, 4)
    print(f"3 multiplied by 4 is {result}")
```
 - Commit these changes with the message "Initial commit of multiplication function".
- On computer 2
 - Fetch the new branch created by Computer 1
 - Check out the `feature-C` branch to start working on it
 - Review the commit history using `git log`
 - Notice that the commit message "Initial commit of multiplication function" is not very descriptive. You decide to rename it to "Add multiply.py with basic multiplication function" to be more specific.
 - Start an interactive rebase to rename the commit
 - In the editor that opens, change the word `pick` to `reword` for the commit you want to rename
 - Save and close the editor. Git will prompt you to edit the commit message
 - Change the commit message to "Add multiply.py with basic multiplication function"
 - Save and close the editor to complete the rebase
- On computer 2
 - Push the updated commit to the remote repository Note: Use `--force-with-lease` instead of `--force` to prevent accidentally overwriting other collaborators' work.
- On computer 1
 - Fetch the latest changes from the remote repository
 - Review the commit history to confirm that the commit message has been successfully updated.
 - Review and merge `feature-C` into `main`

Questions

16. What are the risks associated with rewriting commit history using `git rebase -i`? How can these risks be mitigated?
17. How did using `--force-with-lease` protect the integrity of the remote repository when you pushed the rewritten commit?

Eighth scenario: recovering lost commits with git reflog In this scenario, you'll explore how to recover from a mistake where a commit appears to be lost. You will use `git reflog`, a powerful tool that keeps track of all the changes made in the repository, even those that are not reflected in the branch history.

- On computer 1
 - Create a new branch named `feature-D`.
 - Create a file `addition.py` with the following content:

```
def add(a, b):  
    return a + b  
  
if __name__ == "__main__":  
    result = add(5, 7)  
    print(f"5 plus 7 is {result}")
```
 - Commit these changes with the message "Add addition.py with basic addition function".
 - Push the `feature-D` branch to the remote repository.
- On computer 2
 - Fetch and check out the `feature-D` branch
 - Run the command:

```
git reset --hard HEAD~1
```
 - Realize that this reset has removed the last commit from your local history, causing the `addition.py` file to disappear. Now, you need to recover this lost commit.
 - Use `git reflog` to view the history of all actions in the repository
 - Identify the commit that was reset (you should see the original commit message "Add addition.py with basic addition function" in the reflog output)
 - Recover the lost commit (hint: either reset or copy the commit)
 - Verify that the `addition.py` file is restored and that the commit is back in the branch history
 - Push the recovered commit back to the remote repository to ensure all work is preserved
- On computer 1
 - Fetch the latest changes from the remote repository to confirm that the `addition.py` file and the commit have been successfully restored
 - Review and merge `feature-C` into `main`

Questions

18. What is the purpose of `git reflog`, and how does it differ from `git log`?
19. Why is it important to be cautious when using commands like `git reset --hard`? What could have happened if `git reflog` was not available?
20. How did `git reflog` help you recover from the mistake? What are some scenarios where this tool might be essential in a real project?

Ninth scenario: squashing commits before merging In this scenario, you will learn how to squash multiple commits into a single commit before merging a branch. This is useful to simplify the history.

- On computer 1:
 - Create a new branch `feature-E` from `main`
 - Add a new feature by modifying `script.py` in several steps:
 - * First commit:

```
def feature_e_part1():  
    print("Feature E - Part 1")
```
 - * Second commit:

```
def feature_e_part2():  
    print("Feature E - Part 2")
```
 - * Third commit:

```
def feature_e_part3():  
    print("Feature E - Part 3")
```
 - Before merging `feature-E` into `main`, use interactive rebase to squash the three commits into one
 - Push the branch
- On computer 2:

- Review and merge **feature-E** into **main**

Questions

21. How does squashing affect the ability to track the history of changes?