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.

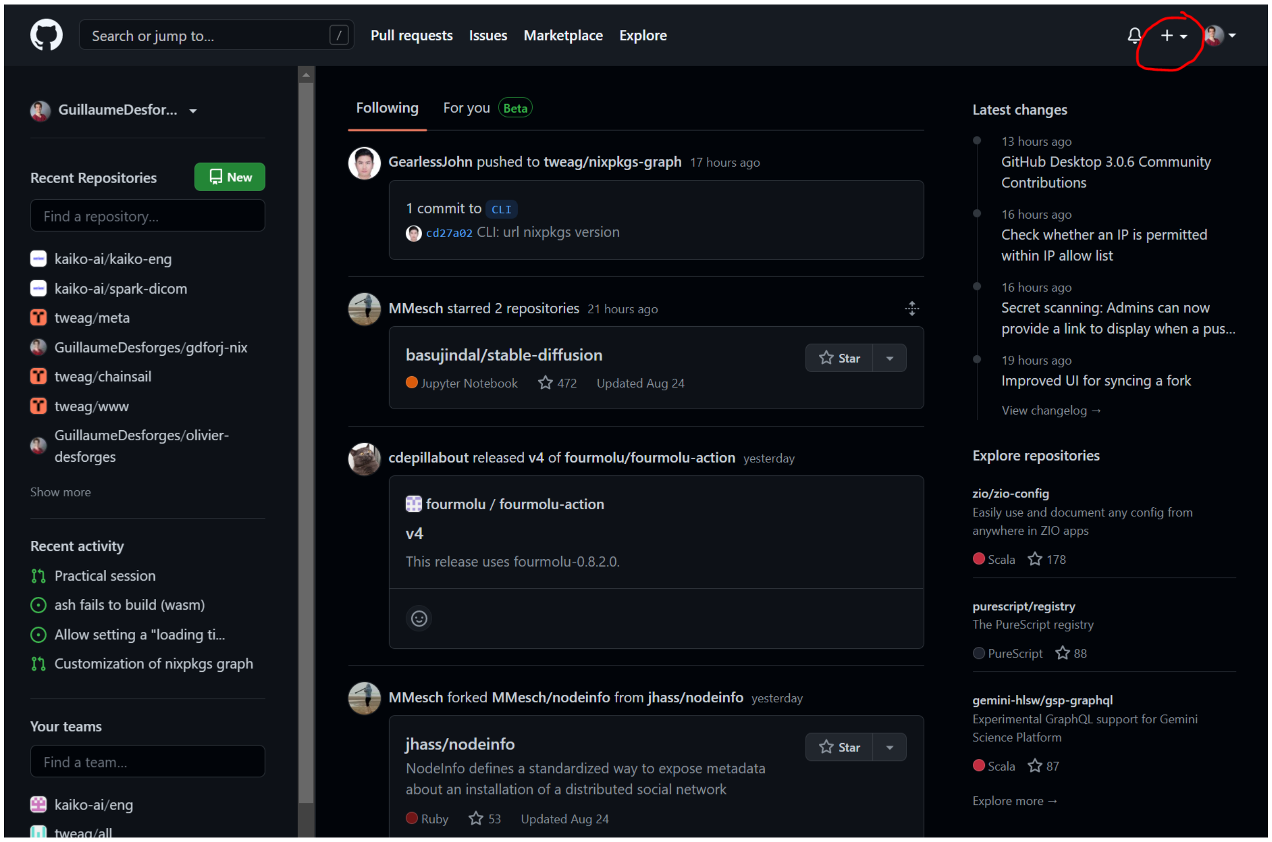
https://github.com/

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



kaiko-ai/kaiko-eng
kaiko-ai/spark-dicom
tweag/meta
GuillaumeDesforges/gdforj-nix
tweag/chainsail
tweag/www
GuillaumeDesforges/olivier-desforges

Practical session
ash fails to build (wasm)
Allow setting a "loading ti...
Customization of nixpkgs graph

kaiko-ai/eng
tweag/all

GearlessJohn pushed to tweag/nixpkgs-graph 17 hours ago
1 commit to CLI
cd27a02 CLI: url nixpkgs version

MMesch starred 2 repositories 21 hours ago
basujindal/stable-diffusion
Jupyter Notebook 472 Updated Aug 24

cdepillabout released v4 of fourmolu/fourmolu-action yesterday
fourmolu / fourmolu-action
v4
This release uses fourmolu-0.8.2.0.

MMesch forked MMesch/nodeinfo from jhass/nodeinfo yesterday
jhass/nodeinfo
NodeInfo defines a standardized way to expose metadata about an installation of a distributed social network
Ruby 53 Updated Aug 24

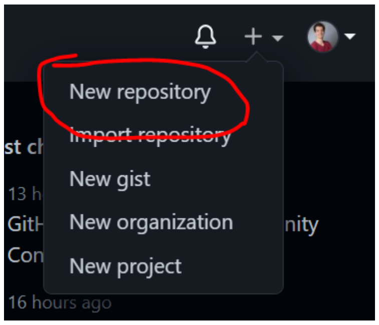
13 hours ago
GitHub Desktop 3.0.6 Community Contributions
16 hours ago
Check whether an IP is permitted within IP allow list
16 hours ago
Secret scanning: Admins can now provide a link to display when a pus...
19 hours ago
Improved UI for syncing a fork

zio/zio-config
Easily use and document any config from anywhere in ZIO apps
Scala 178

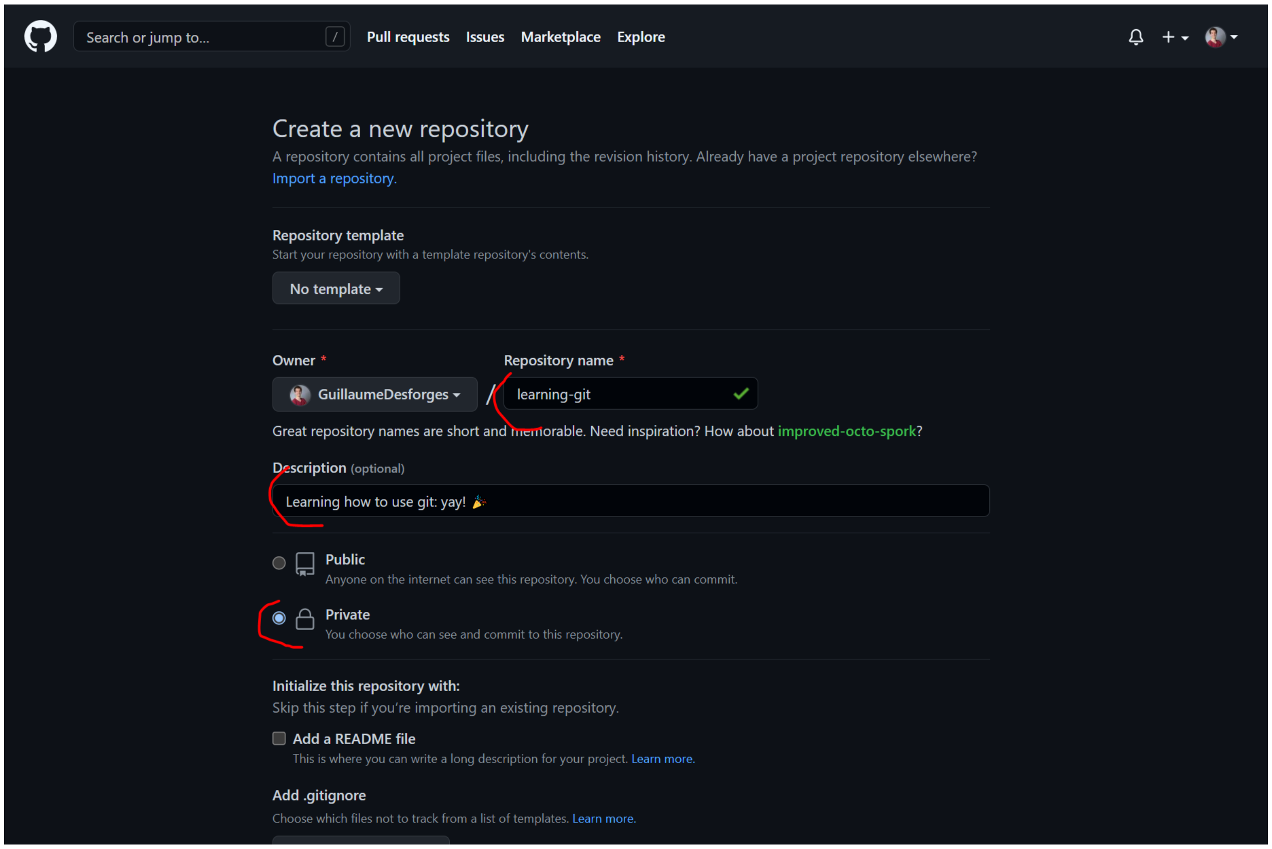
purescript/registry
The PureScript registry
PureScript 88

gemini-hlsw/gsp-graphql
Experimental GraphQL support for Gemini Science Platform
Scala 87

- Click on “New repository”



- You can name your repository however you want > Usually, we use lowercase characters (`a`, `b`, `c`, ...) spearated 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”



Search or jump to... Pull requests Issues Marketplace Explore

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Repository template

Start your repository with a template repository's contents.

No template

Owner * GuillaumeDesforges / Repository name * learning-git ✓

Great repository names are short and memorable. Need inspiration? How about improved-octo-spork?

Description (optional)

Learning how to use git: yay! 🎉

○ Public — Anyone on the internet can see this repository. You choose who can commit.

● Private — You choose who can see and commit to this repository.

Initialize this repository with:

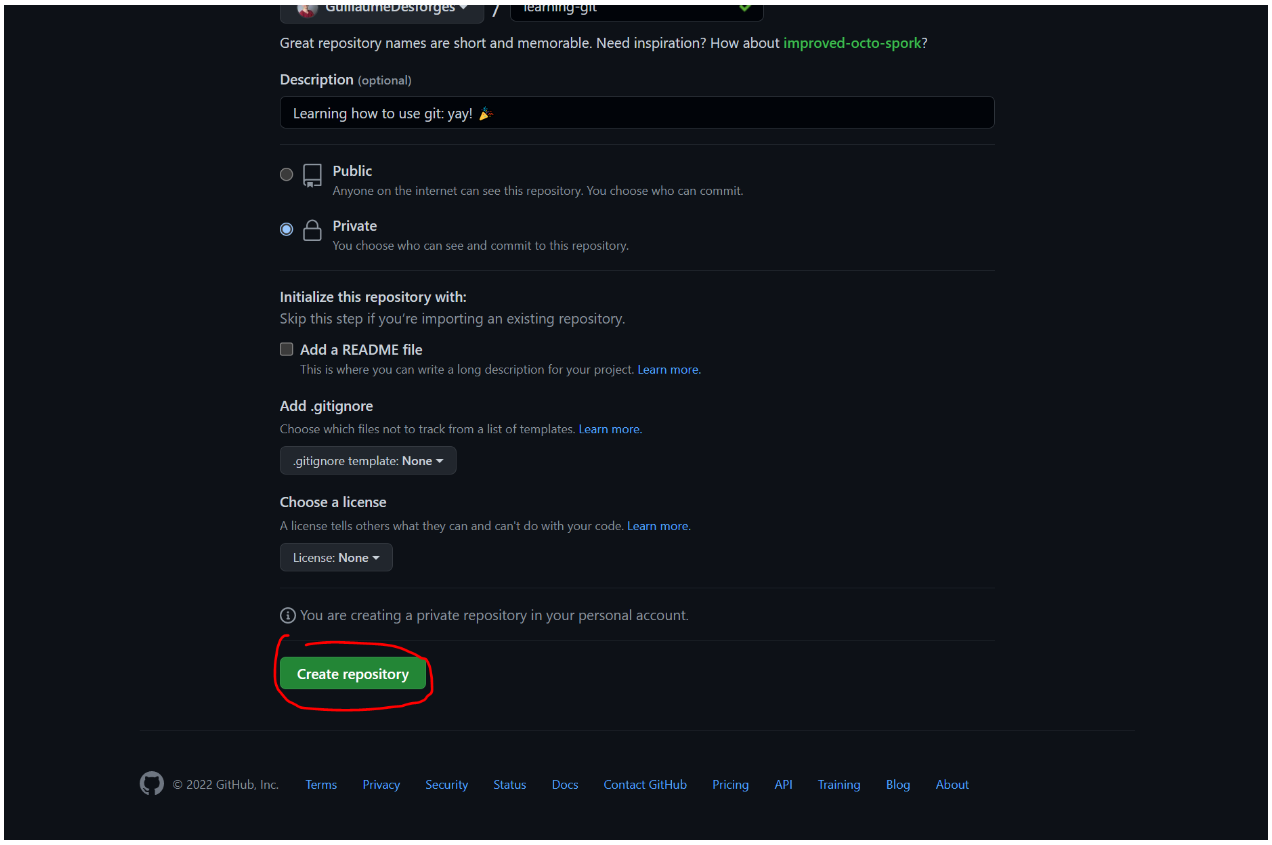
Skip this step if you're importing an existing repository.

☐ Add a README file — This is where you can write a long description for your project. Learn more.

Add .gitignore

Choose which files not to track from a list of templates. Learn more.

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


Great repository names are short and memorable. Need inspiration? How about improved-octo-spork?

Description (optional): Learning how to use git: yay! 🎉

(○) Public — Anyone on the internet can see this repository. You choose who can commit.

(●) Private — You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

☐ Add a README file — This is where you can write a long description for your project. Learn more.

Add .gitignore — Choose which files not to track from a list of templates. Learn more.
.gitignore template: None

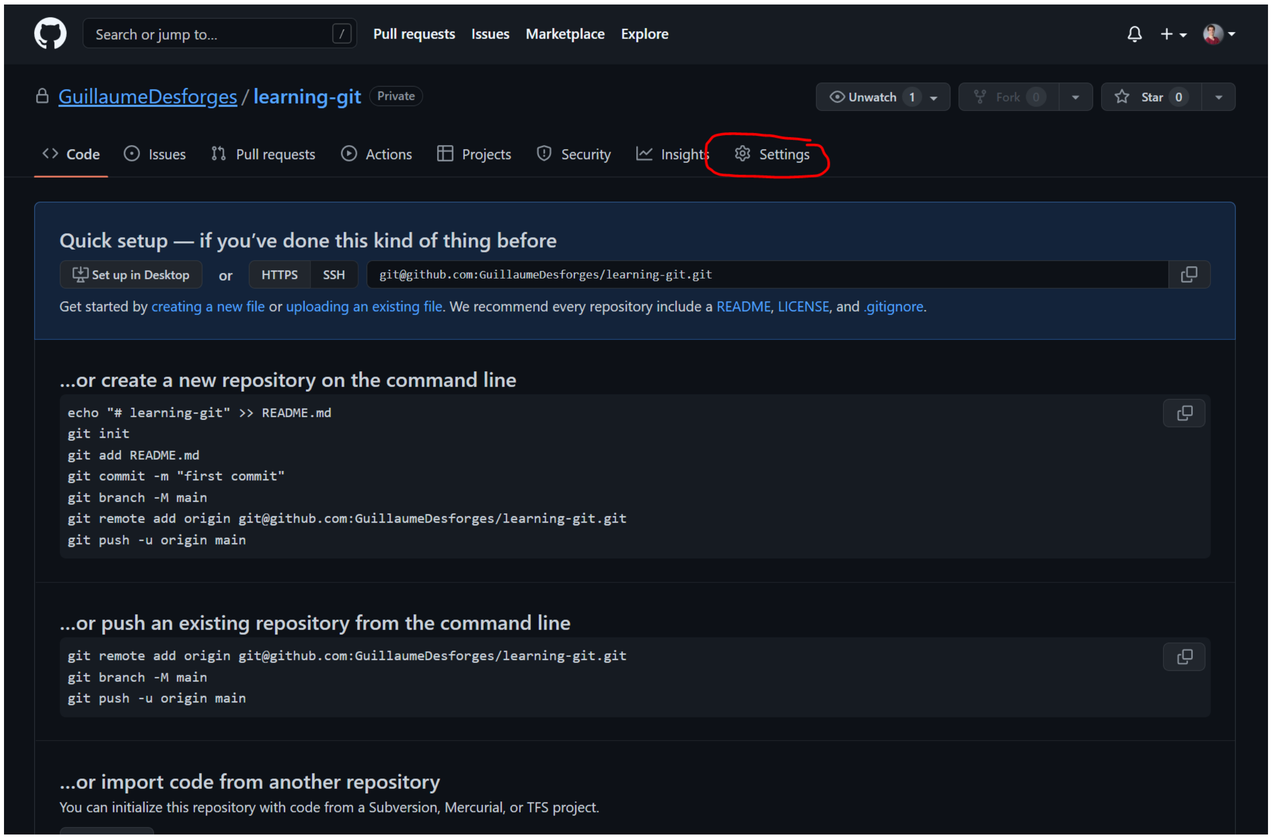
Choose a license — A license tells others what they can and can't do with your code. Learn more.
License: None

ⓘ You are creating a private repository in your personal account.

Create repository
© 2022 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

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)



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH `git@github.com:GuillaumeDesforges/learning-git.git`

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# learning-git" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:GuillaumeDesforges/learning-git.git
git push -u origin main
```

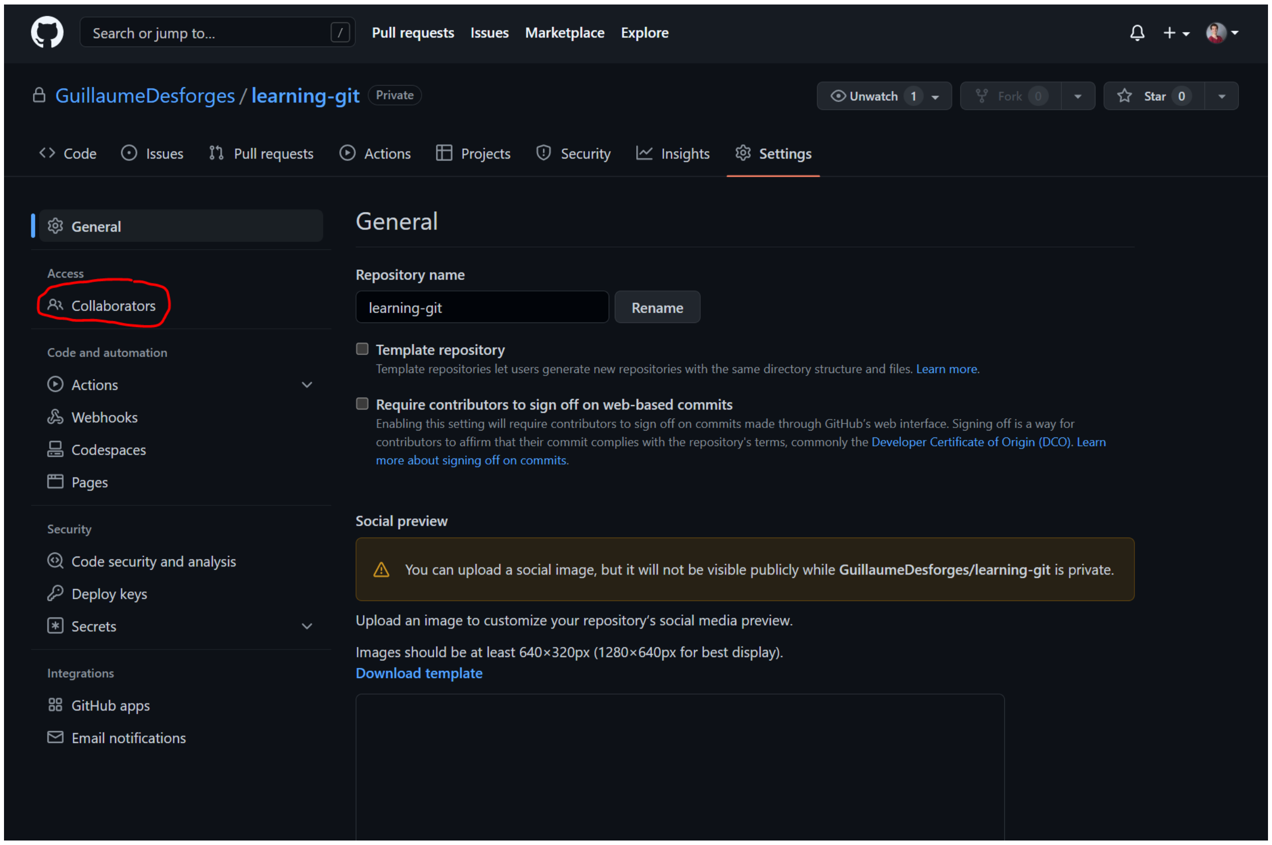
...or push an existing repository from the command line

```
git remote add origin git@github.com:GuillaumeDesforges/learning-git.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

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



General

Repository name

learning-git Rename

Template repository

Template repositories let users generate new repositories with the same directory structure and files. Learn more.

Require contributors to sign off on web-based commits

Enabling this setting will require contributors to sign off on commits made through GitHub's web interface. Signing off is a way for contributors to affirm that their commit complies with the repository's terms, commonly the Developer Certificate of Origin (DCO). Learn more about signing off on commits.

Social preview

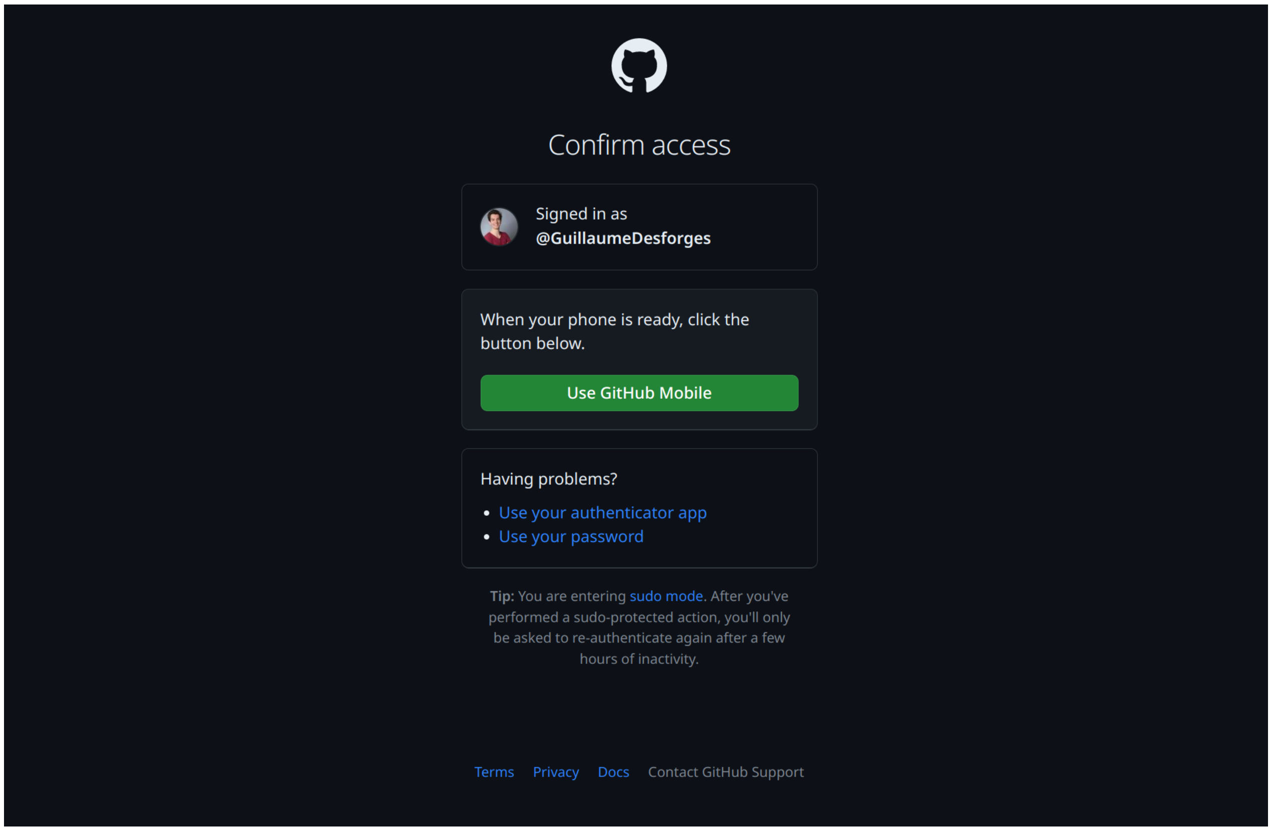
You can upload a social image, but it will not be visible publicly while **GuillaumeDesforges/learning-git** is private.

Upload an image to customize your repository's social media preview.

Images should be at least 640×320px (1280×640px for best display).

Download template

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



Confirm access

Signed in as
@GuillaumeDesforges

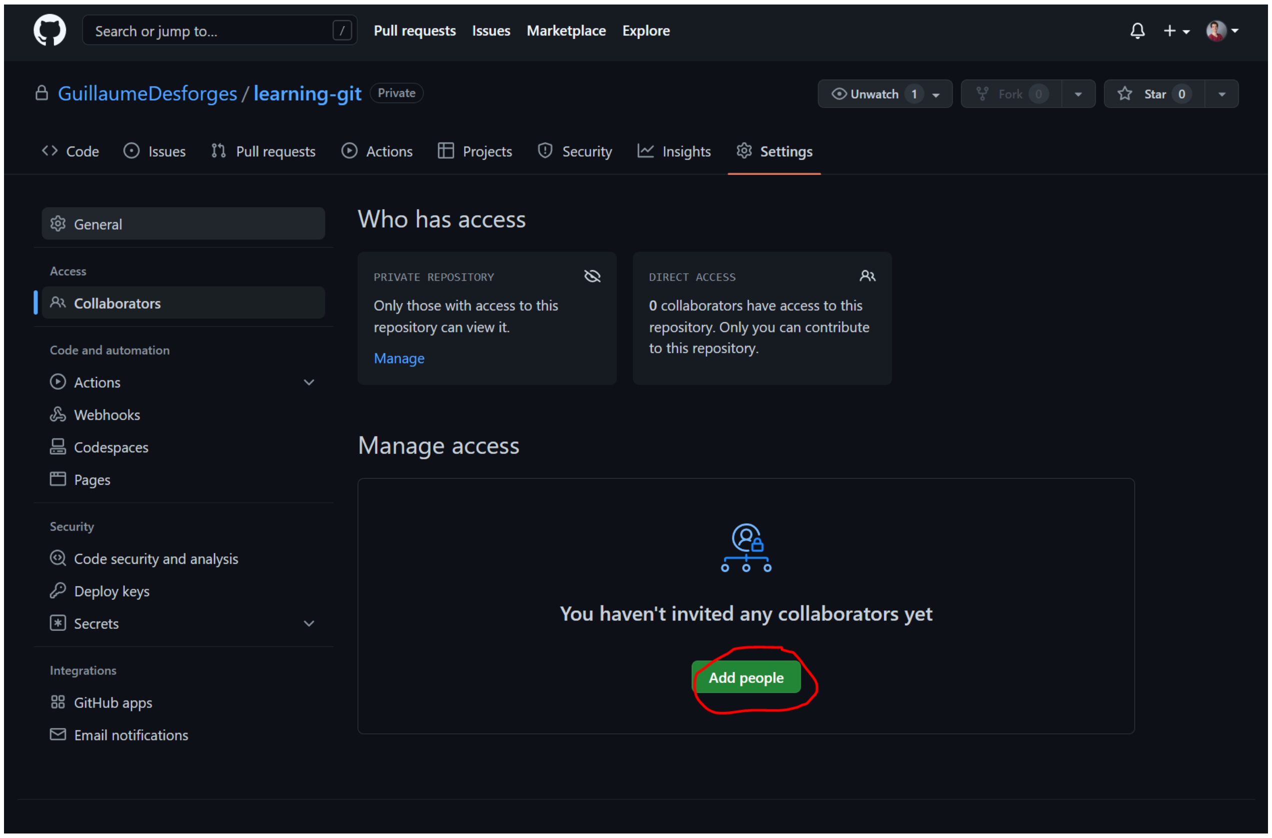
When your phone is ready, click the button below.

Having problems?

- Use your authenticator app
- Use your password

Tip: You are entering sudo mode. After you've performed a sudo-protected action, you'll only be asked to re-authenticate again after a few hours of inactivity.

- Click on “Add people”



Who has access

PRIVATE REPOSITORY
Only those with access to this repository can view it.
Manage

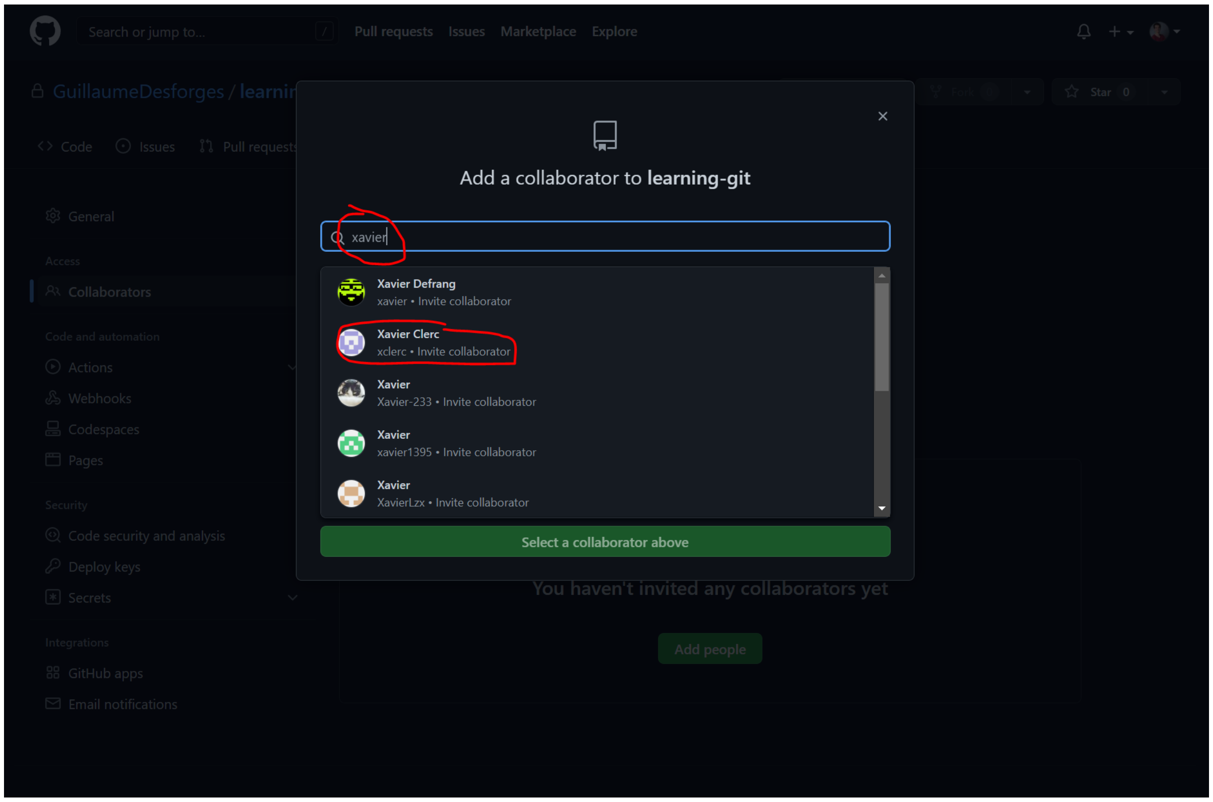
DIRECT ACCESS
0 collaborators have access to this repository. Only you can contribute to this repository.

Manage access

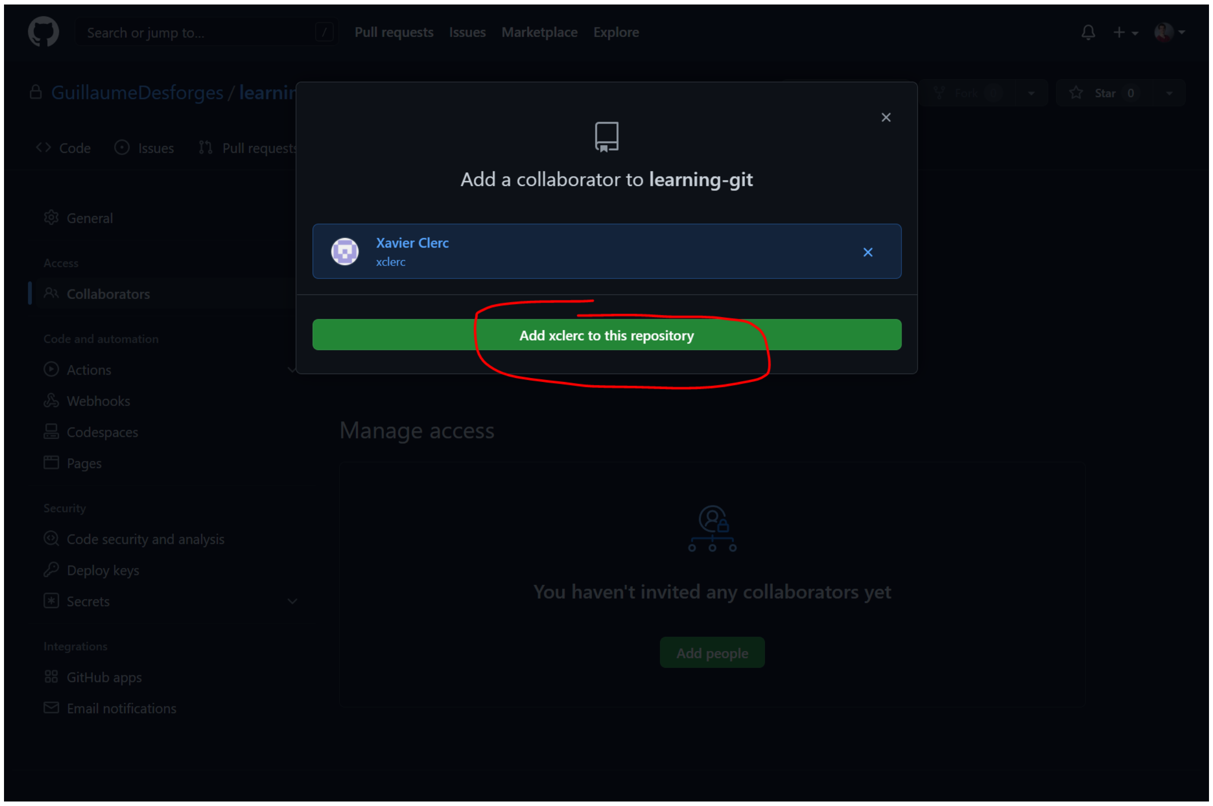
You haven't invited any collaborators yet

- Find and add your partner

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


Search or jump to...
Pull requests
Issues
Marketplace
Explore
GuillaumeDesforges / learnin
Code
Issues
Pull requests
General
Access
Collaborators
Code and automation
Actions
Webhooks
Codespaces
Pages
Security
Code security and analysis
Deploy keys
Secrets
Integrations
GitHub apps
Email notifications
Add a collaborator to learning-git
xavier
Xavier Defrang
xavier • Invite collaborator
Xavier Clerc
xclerc • Invite collaborator
Xavier
Xavier-233 • Invite collaborator
Xavier
xavier1395 • Invite collaborator
Xavier
XavierLzx • Invite collaborator
Select a collaborator above
You haven't invited any collaborators yet
Add people

– 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"Elasped 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:

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

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

      ```python
      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:

          ```python
          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()
          ```

```python
      print(f"Elasped 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
- If you have time, you can rerun this scenario by merging `sum` into `faster-sum` instead of rebasing.

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: 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-edit`
  - Create a file `multiply.py` with the following content:
    ```python
    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-edit` branch to start working on it:
    ```
    git checkout feature-edit
    ```
  - Review the commit history using:
    ```
    git log --oneline
    ```
  - 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:
    ```
    git rebase -i HEAD~1
    ```
  - 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:
    ```
    git push --force-with-lease
    ```
    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:
    ```
    git pull
    ```

- Review the commit history to confirm that the commit message has been successfully updated.

Questions

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

Sixth 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-addition`.
  - 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-addition` branch to the remote repository.
- On computer 2
  - Fetch and check out the `feature-addition` branch:
    ```
    git checkout feature-addition
    ```
  - You decide to reset the branch to the previous state by mistake:
    ```
    git reset --hard HEAD~1
    ```
  - Realize that this reset has removed the last commit, causing the `addition.py` file to disappear. Now, you need to recover this lost commit.
- On computer 2
  - Use `git reflog` to view the history of all actions in the repository:
    ```
    git reflog
    ```
  - 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 by resetting back to it:
    ```
    git reset --hard <commit-hash>
    ```
    Replace `<commit-hash>` with the hash of the lost commit you want to recover.
- On computer 2
  - Verify that the `addition.py` file is restored and that the commit is back in the branch history:
    ```
    git log --oneline
    ```
  - Push the recovered commit back to the remote repository to ensure all work is preserved:
    ```
    git push --force-with-lease
    ```
- 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:
    ```
    git pull
    ```

Questions

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

Seventh 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:
  1. Start by creating a new branch `feature-A` from the `main` branch.
     ```
     git checkout -b feature-A
     ```
  2. Modify `script.py` by adding a new function:
     ```
     def new_feature():
         print("This is a new feature")
     ```
  3. You now need to switch back to the `main` branch. Use `git stash` to temporarily save your changes.
     ```
     git stash
     ```
  4. Switch to the `main` branch.
     ```
     git checkout main
     ```
  5. Make some changes in `main`, commit the changes, and push them to the remote repository.
     ```
     # Make changes to fix the bug
     git commit -am "Fix critical bug in main"
     git push origin main
     ```
  6. Switch back to the `feature-A` branch and reapply your stashed changes.
     ```
     git checkout feature-A
     git stash apply
     ```

Questions

15. Why might you use `git stash` instead of committing incomplete changes?
16. How does stashing help prevent conflicts when switching branches?
17. What is the difference between `git stash apply` and `git stash pop`?

Eighth 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:
  1. Create a new branch `feature-B` from `main`.
     ```
     git checkout -b feature-B
     ```
  2. Add a new feature by modifying `script.py` in several steps:
     - First commit:
       ```
       def feature_b_part1():
           print("Feature B - Part 1")
       git commit -am "Add part 1 of feature B"
       ```
     - Second commit:
       ```
       def feature_b_part2():
           print("Feature B - Part 2")
       git commit -am "Add part 2 of feature B"
       ```
     - Third commit:
       ```
       def feature_b_part3():
           print("Feature B - Part 3")
       git commit -am "Add part 3 of feature B"
       ```
  3. Before merging `feature-B` into `main`, use interactive rebase to squash the three commits into one:
     ```
     git rebase -i HEAD~3
     ```
  4. During the rebase process, mark the second and third commits as `squash` or `s` to combine them with the first.
  5. Edit the commit message as needed, save, and close the editor.
  6. Merge the squashed commit into `main`.
     ```
     git checkout main
     git merge feature-B
     git push origin main
     ```

Questions

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

Ninth 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:
  1. Create a new branch `feature-C` from `main`.
     ```
     git checkout -b feature-C
     ```
  2. Add a new feature to `script.py`:
     ```
     def feature_c():
         print("Feature C is now available!")
     git commit -am "Add feature C"
     ```
  3. Also, add a bug fix in the same branch:
     ```
     def bug_fix():
         print("Critical bug fixed!")
     git commit -am "Fix critical bug"
     ```
  4. Push the changes to the remote repository:
     ```
     git push origin feature-C
     ```
  5. Realize that the bug fix is critical and needs to be applied to the `main` branch immediately without merging `feature-C`.
  6. On `main`, use `git cherry-pick` to apply the bug fix commit to `main`:
     ```
     git checkout main
     git cherry-pick <commit-hash-of-bug-fix>
     ```
  7. Push the changes to `main`:
     ```
     git push origin main
     ```

Questions

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