

Git & GitHub

Introdução Engenharia Informática

Mário Antunes

October 27, 2025

Exercises

Practical Lab: Git & GitHub

From Local Repository to Open-Source Collaboration

Objective: This lab will guide you through the complete lifecycle of a Git repository. You will learn to create a local repository, manage versions, work with branches, and finally, collaborate on a remote project using GitHub.

Prerequisites:

- **Git Installed:** You must have Git installed on your machine.
 - **A GitHub Account:** You will need a free GitHub account for the collaboration exercises.
 - **A Text Editor:** Any text editor (like VS Code, Sublime Text, or Nano) will work.
-

Part 0: Setup & Authentication

Before we can work with remote repositories, we need to install Git and tell GitHub who we are.

Step 1: Install Git

On Debian-based systems (like debian trixieOS or Ubuntu), you can install Git using apt.

1. First, update your package list:

```
$ sudo apt update
```

2. Then, install Git:

```
$ sudo apt install git
```

Step 2: Configure Your Identity

You must tell Git your name and email. This information will be baked into every commit you make.

```
$ git config --global user.name "Your Name"  
$ git config --global user.email "your.email@ua.pt"
```

(Use your UA email.)

Step 3: Authenticate to GitHub

To push your code to GitHub, you must prove who you are. You have two main options: SSH (recommended) or a Personal Access Token (PAT).

Method 1: Using an SSH Key (Recommended)

This method is more secure and convenient. You add a “key” to your GitHub account, and your computer uses it to authenticate automatically.

1. Generate a new ed25519 SSH key. This command creates a key pair without asking for a password (-N "").

```
$ ssh-keygen -t ed25519 -C "your.email@example.com" -f ~/.ssh/id_ed25519 -N ""
```
2. Display your new **public** key in the terminal so you can copy it.

```
$ cat ~/.ssh/id_ed25519.pub
```
3. Copy the entire output (starting with ssh-ed25519 ... and ending with your email).
4. Add the key to GitHub:
 - Go to **GitHub.com** and click your profile icon in the top-right.
 - Go to **Settings -> SSH and GPG keys** (in the "Access" sidebar).
 - Click **New SSH key**.
 - Give it a **Title** (e.g., "My TrixieOS Laptop").
 - Paste your copied key into the **Key** box.
 - Click **Add SSH key**.

Method 2: Using a Fine-Grained Personal Access Token (PAT)

A PAT is like a password that you can use for Git operations.

1. Go to **GitHub.com -> Settings -> Developer settings** (at the bottom of the sidebar).
 2. Go to **Personal access tokens -> Fine-grained tokens**.
 3. Click **Generate new token**.
 4. Set the following options:
 - **Token name:** Give it a descriptive name (e.g., "IEI Class Token").
 - **Expiration:** Select **No expiration**.
 - **Repository access:** Select **All repositories**.
 - **Permissions:** Scroll down to "Repository permissions" and find **Contents**. Change its access to **Read and write**.
 5. Click **Generate token**.
 6. **IMPORTANT:** Copy the token (it starts with `github_pat_...`) *immediately*. You will **never** see it again after you leave this page.
 7. When you need to connect to GitHub (in Part 2 and 3), you will use this token in the URL:
`https://<YOUR_USERNAME>:<YOUR_TOKEN>@github.com/<YOUR_USERNAME>/<REPOSITORY>.git`
-

Part 1: Your Local Repository

Exercise 1: git init (Creating a Repository) Our first step is to tell Git to start tracking a new project.

1. Create a new folder for your project and navigate into it.

```
$ mkdir my-git-project
$ cd my-git-project
```
2. Now, initialize it as a Git repository.

```
$ git init
```
3. This creates a hidden `.git` folder. You've officially created a repository!

Exercise 2: The Core Loop (add, commit, status, log) Let's create a file, "stage" it, and "commit" it to our history.

1. Create a file named `index.html` inside your `my-git-project` folder and add the following content:

```
<h1>Welcome to My Project</h1>
```
2. Check the "status" of your repository.

```
$ git status
```

Git will show you `index.html` as an "untracked file."
3. Tell Git you want to track this file by adding it to the **Staging Area**.

```
$ git add index.html
```

4. Check the status again. The file is now “staged” and ready to be committed.

```
$ git status
```

5. Now, save this “snapshot” to your history with a **commit**.

```
$ git commit -m "Initial commit: Add homepage"
```

6. Finally, look at the history log.

```
$ git log
```

Exercise 3: Fixing a Bad Commit (--amend)

Good commit messages are vital. Let’s fix a bad one.

1. Make a small change to `index.html`. For example, add a paragraph:

```
<h1>Welcome to My Project</h1>
<p>This is a project for my IEI class.</p>
```

2. Commit this change with a **bad** message. The `-a` flag is a shortcut for `git add` (for tracked files) and `git commit`.

```
$ git commit -a -m "fix stuff"
```

3. Check your log: `git log --oneline`. You’ll see your “fix stuff” message. Let’s fix it.

4. Run the **amend** command. This will replace your *previous* commit with a new one.

```
$ git commit --amend -m "Doc: Update homepage text"
```

5. Check your log again: `git log --oneline`. The “fix stuff” commit is gone, replaced by your better message.

Exercise 4: Ignoring Files (.gitignore)

We never want to commit secret keys or temporary files.

1. Create a file named `.env` and add a “secret” to it.

```
$ echo "DATABASE_PASSWORD=12345" > .env
```

2. Run `git status`. You’ll see Git wants to add `.env`. We don’t want this.

3. Create a file named `.gitignore` (yes, it starts with a dot).

4. Add the following line inside `.gitignore`:

```
.env
```

5. Run `git status` again. The `.env` file has vanished from the list, but Git now wants to track the `.gitignore` file, which is exactly what we want.

6. Add and commit the `.gitignore` file.

```
$ git add .gitignore
$ git commit -m "Feat: Add .gitignore to ignore environment files"
```

Exercise 5: Branching (branch, checkout)

Let’s work on a new feature in isolation without breaking our main code.

1. Create a new branch for a new “about” page.

```
$ git branch feature/about-page
```

2. Switch to your new branch.

```
$ git checkout feature/about-page
```

(*Shortcut: git checkout -b <branch-name> creates and switches in one command.*)

3. Create an `about.html` file with this content:

```
<h1>About Us</h1>
<p>This is the about page.</p>
```

4. Add and commit this new file *on your feature branch*.

```
$ git add about.html  
$ git commit -m "Feat: Add new about page"
```

5. Now, switch back to your main branch and look at your files.

```
$ git checkout main  
$ ls
```

The `about.html` file is gone! This is because it only exists on the feature branch.

Exercise 6: Merging (merge)

Your “about page” feature is complete. Let’s merge it into the `main` branch.

1. Make sure you are on the branch you want to receive the changes (i.e., `main`).

```
$ git checkout main
```

2. Run the `merge` command to pull in the changes from your feature branch.

```
$ git merge feature/about-page
```

3. Check your files with `ls`. The `about.html` file is now present on `main`.

4. Look at your history to see the merge commit.

```
$ git log --oneline --graph
```

Exercise 7: Resolving Merge Conflicts

What happens when two branches edit the same line?

1. From your `main` branch, create a new branch.

```
$ git checkout -b change-title-A
```

2. On this `change-title-A` branch, edit `index.html` to say:

```
<h1>Welcome to the IEI Project</h1>
```

3. Commit this change.

```
$ git commit -a -m "Update title on branch A"
```

4. Now, go back to `main` and create a *conflicting* change.

```
$ git checkout main  
$ git checkout -b change-title-B
```

5. On this `change-title-B` branch, edit the *same line* in `index.html` to say:

```
<h1>Welcome to the TIA Project</h1>
```

6. Commit this change: `git commit -a -m "Update title on branch B"`

7. Now, let’s try to merge `change-title-B` into `change-title-A`.

```
$ git checkout change-title-A  
$ git merge change-title-B
```

CONFLICT! Git will stop and tell you there is a conflict in `index.html`.

8. **Fix it:** Open `index.html`. You will see the conflict markers (`<<<<`, `====`, `>>>>`). Edit the file to be correct (e.g., delete the markers and choose one title, or write a new one).

9. **Finalize:** Once fixed, add the file and `commit`.

```
$ git add index.html  
$ git commit -m "Merge: Resolve title conflict"
```

Part 2: GitHub - Collaboration

Exercise 8: clone, remote, & origin Let's connect our local repository to a remote one on GitHub.

1. Go to [GitHub.com](#). Create a **new, empty, public repository**. Name it `git-practice-repo`.
2. **Do NOT** initialize it with a README. We want it to be empty.
3. GitHub will show you URLs. Find the **Code** button.
 - **If you set up an SSH Key:** Select the **SSH** tab and copy the URL (e.g., `git@github.com:<YOUR_USERNAME>/practice-repo.git`).
 - **If you created a PAT:** Select the **HTTPS** tab and copy the URL (e.g., `https://github.com/<YOUR_USERNAME>/practice-repo.git`).
4. In your local terminal, go back to your `my-git-project` folder.
5. Add this new GitHub repository as your "remote" named "origin", using the URL that matches your authentication method.
 - **If using SSH:**
`$ git remote add origin <PASTE_YOUR_SSH_URL_HERE>`
 - **If using PAT:** Use the special URL format from Part 0, replacing the placeholders.
`$ git remote add origin https://<YOUR_USERNAME>:<YOUR_TOKEN>@github.com/<YOUR_USERNAME>/practice-repo.git`
6. Verify that the remote was added. "`bash $ git remote -v`"

Exercise 9: push (Pushing Your Work) Your local repository has history, but the remote one is empty. Let's push your work.

1. First, let's rename our local `master` branch to `main` to match GitHub's standard.
`$ git branch -M main`
2. Now, **push** your local `main` branch to the remote `origin`. The `-u` flag sets it as the default, so you can just use `git push` in the future.
`$ git push -u origin main`
3. Refresh your GitHub repository page. All your files (`index.html`, `about.html`, `.gitignore`) and your commit history are now online!

Exercise 10: tag & release (Marking a Version) Your project is at a stable point. Let's tag it as version 1.0.

1. Create a "tag" that points to your latest commit.
`$ git tag -a v1.0.0 -m "First stable release"`
 2. Push your new tag to GitHub (tags don't push automatically).
`$ git push origin v1.0.0`
 3. **On GitHub:** Go to your repository's main page. Find "Releases" on the right side. Click "Create a new release" (or "Draft a new release").
 4. Select your `v1.0.0` tag, give it a title like "Version 1.0.0", and write a short description. Click "Publish release". You now have an official release!
-

Part 3: The Full Open-Source Workflow

Exercise 11: fork (Contributing to a Project) You will now contribute to a project that you do not own.

1. Go to this repository on GitHub (the tictactoe repository from last class): <https://github.com/mariolpantunes/tictactoe>
2. In the top-right corner, click the "**Fork**" button. This will create a copy of the repository under your own GitHub account.
3. Now, on **your** fork's GitHub page, click the green "<> Code" button.
 - **If you set up an SSH Key:** Select the **SSH** tab and copy the URL (e.g., `git@github.com:<YOUR_USERNAME>/tictactoe.git`).
 - **If you created a PAT:** Select the **HTTPS** tab and copy the URL (e.g., `https://github.com/<YOUR_USERNAME>/tictactoe.git`).
4. In your terminal (outside your old project folder), **clone** **your fork** using the correct command for your auth method.

- **If using SSH:**
`$ git clone <PASTE_YOUR_SSH_URL_HERE>`
`$ cd tictactoe`
- **If using PAT:**
`$ git clone https://<YOUR_USERNAME>:<YOUR_TOKEN>@github.com/<YOUR_USERNAME>/tictactoe`

Exercise 12: The Pull Request (pull request) Let's make a change and propose it to the original project.

1. Create a new branch for your change.

```
$ git checkout -b add-my-name
```

2. Edit the CONTRIBUTORS.md file and add your name to the list.

3. Add and commit your change.

```
$ git add CONTRIBUTORS.md
$ git commit -m "Add [Your Name] to contributors list"
```

4. Push this new branch to your fork (origin).

```
$ git push origin add-my-name
```

5. **Go to GitHub:** Go to your fork's repository page. You should see a green banner that says "This branch is 1 commit ahead..." Click the "**Contribute**" button and then "**Open a pull request**".

6. Review the changes, add a nice message, and click "**Create pull request**".

7. **Congratulations!** You have just made a pull request, the heart of open-source collaboration.
-

Bonus Challenge: rebase (Cleaning History)

Let's re-do Exercise 6, but with rebase for a cleaner history.

1. Get back to a state before the merge. A good way is to reset main.

```
$ cd ~/my-git-project # Go back to your first project
$ git checkout main
$ git reset --hard HEAD~1 # This rewinds 'main' one commit (deletes the merge)
```

2. You still have your feature/about-page branch. Let's make another commit on main to create a divergence.

```
$ echo "" >> index.html
$ git commit -a -m "Add a comment to homepage"
```

3. Now, main has a commit that feature/about-page does not.

4. Switch to your feature branch and use rebase to replay your branch's commits on top of the new main.

```
$ git checkout feature/about-page
$ git rebase main
```

5. Now, switch back to main and merge.

```
$ git checkout main
$ git merge feature/about-page
```

6. It will say "Fast-forward". Look at your log (`git log --oneline --graph`). The history is perfectly linear and clean!