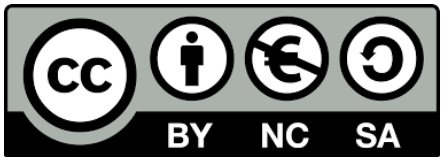
2. Documentation and Version Control Systems

Miquel Àngel París i Peñaranda

Web Application Deployment

2nd C-VET Web Application Development



Index

Remote Repositories.

In previous practical examples, we've seen how to create a repository from a working directory on the local machine. Now, we will see how to retrieve a repository located on another machine. This process is known as cloning a repository.

When we talk about cloning a repository, we refer to copying a complete remote workspace onto our machine, including all commits and branches made on the other machine. Even though Git works in a distributed way, we will need a reference repository acting as a server.

Step 1: Creating the Server

When we use ` git init`, we establish the current directory as a working directory. This creates a hidden `.git` subdirectory with all the information about version control. This allows us to work locally, creating our commits, branches, etc., but not to work in a distributed manner.

To create a repository in the current directory that can be shared, we need a "bare" repository, which will have a different structure from working directories, as it won't contain the `.git` folder and will use the base directory to store its content.

First, we create the directory where we want to create the repository:

```
$ mkdir sharedProject
```

Once created, we enter the directory and initialize the bare repository:

```
$ cd sharedProject
```

And initialize the repository here:

```
$ git init --bare
```

Output:

```
root@prserver:/home/prserver# mkdir SharProject
root@prserver:/home/prserver# cd SharProject/
root@prserver:/home/prserver/SharProject# git init --bare
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/prserver/SharProject/
```

If we check the directory's content, we will see it has the same structure as the `.git` directory in a local repository:

$ ls

Output:

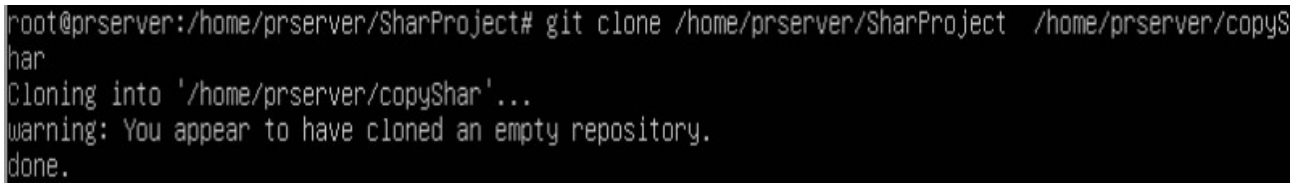
Step 2: Cloning the Repository (on the local computer)

In general, to clone a project, we write git clone source [destination]:

For example, if we want to clone a repository located in a folder on our machine, we would write:

$ git clone ~/sharedProject ~/project3

Output:

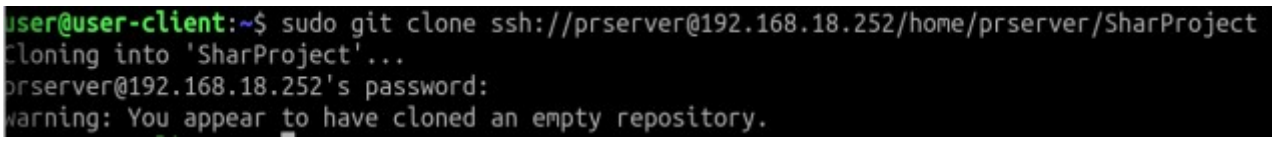


This command clones the `sharedProject` project into a folder called `project3`. If we omit the second folder, the repository will be cloned into the current directory with the same name as the original folder.

On the other hand, if we want to clone a remote repository, we connect to it through HTTPS or SSH protocols. Suppose the computer where we created the shared repository has the IP address `192.168.18.252`. We would connect to it with:

$ git clone ssh://user@192.168.18.252/home/user/sharedProject

Output:



Note that we haven't specified a destination path, so the shared repository will be cloned into a folder with the same name in the current directory. The computer we want to clone from must have an SSH server activated:

$ systemctl status ssh (check)

$ sudo apt install openssh-server

Next, we will be prompted for the password of the user we connected to via SSH. If we don’t want to use the same local user, we can create a new one for these tasks.

Finally, the system informs us that we have cloned an empty repository. This is because the repository has been initialized but does not yet contain any content.

Step 3: Adding Content to the Repository

Now, from the local computer, we navigate to the folder where we cloned the repository:

```
$ cd sharedProject/
```

We add some content, for example, a file named `file1.md` with the content "# Shared Project":

```
$ echo "# Shared Project" >> file1.md
```

We add the file to be confirmed in the next commit:

```
$ git add file1.md
```

And we make the commit:

```
$ git commit -m "Added file1.md"
```

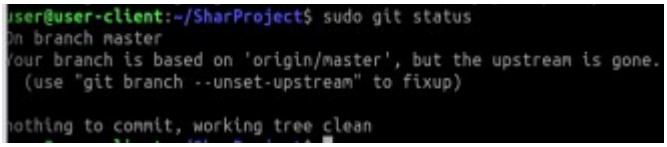
Output:

Notice that the commit message shows `master (root-commit) c76e80e`, indicating that it is the `master` branch and the root commit. This is related to the empty repository warning shown when we cloned the repository, meaning it is the first commit made in it.

If we check the repository status now:

```
$ git status
```

Output:



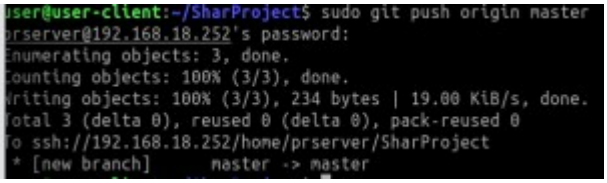
We see that we are on the `master` branch, but this branch doesn’t exist in the upstream (the origin of the clone). In this situation, it is normal since we haven’t yet synchronized the first commit with the server.

Step 4: Pushing the Commit to the Server

The changes have been made in our local copy of the project. To share them with the rest of the project collaborators, we need to synchronize them with the server. To send the changes to the server, we use the `git push` command:

```
$ git push origin master
```

Output:



This tells Git to push our `master` branch to the origin. Notice that the new `master` branch is created since it didn't yet exist in the repository.

The options `origin` and `master` can be omitted, leaving the command as just `git push`. If we don't specify the remote server, Git will choose the one we have configured, which in this case is `origin`. We can check which remote servers we have with:

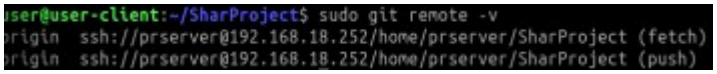
```
$ git remote
```

Output:

And get more information about them with:

```
$ git remote -v
```

Output:

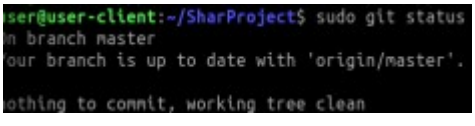


This also gives us the user, host, and directory of the repositories.

If we check the status again:

```
$ git status
```

Output:



We confirm that we are now up to date with the `master` branch from the origin.

Step 5: Fetching Content from the Serve

When we want to get changes from other developers on the project, we use the `git fetch` and `git pull` commands.

- `git fetch [remote-name]`: This retrieves the changes made in the remote project since we cloned it or since we last fetched updates. This command only brings the data but doesn't combine it with our working directory. It allows us to review the changes and apply them manually when needed.
- `git pull [remote-name]`: This retrieves the changes from the remote project and combines them with our working directory. This workflow is quicker and easier than using `fetch`, although it provides less control over the repository.

To get the latest changes in the remote repository, we write:

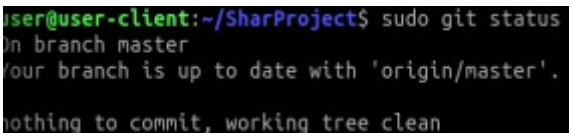
```
$ git fetch
```

Output:

We can omit `origin` since it's the default remote. If we check the status now:

```
$ git status
```

Output:



We see that we are on the `master` branch, which is behind the `master` branch in the origin by one commit.

If we check the content of the current working directory, we still see the content from the initial commit:

```
$ ls
```

Output:

Now, using `pull` to retrieve and combine the latest changes:

```
$ git pull
```

Output:

We have now synchronized our local copy with the server's content:

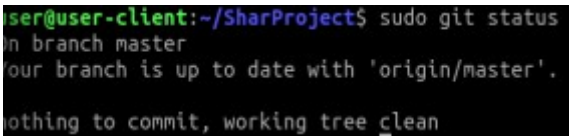
```
$ ls
```

Output:

And if we check the status, we are informed that we are up to date with the `master` branch from the origin:

```
$ git status
```

Output:



You can find more information about working with remote Git repositories at the following link: [Git Documentation: Working with Remotes].

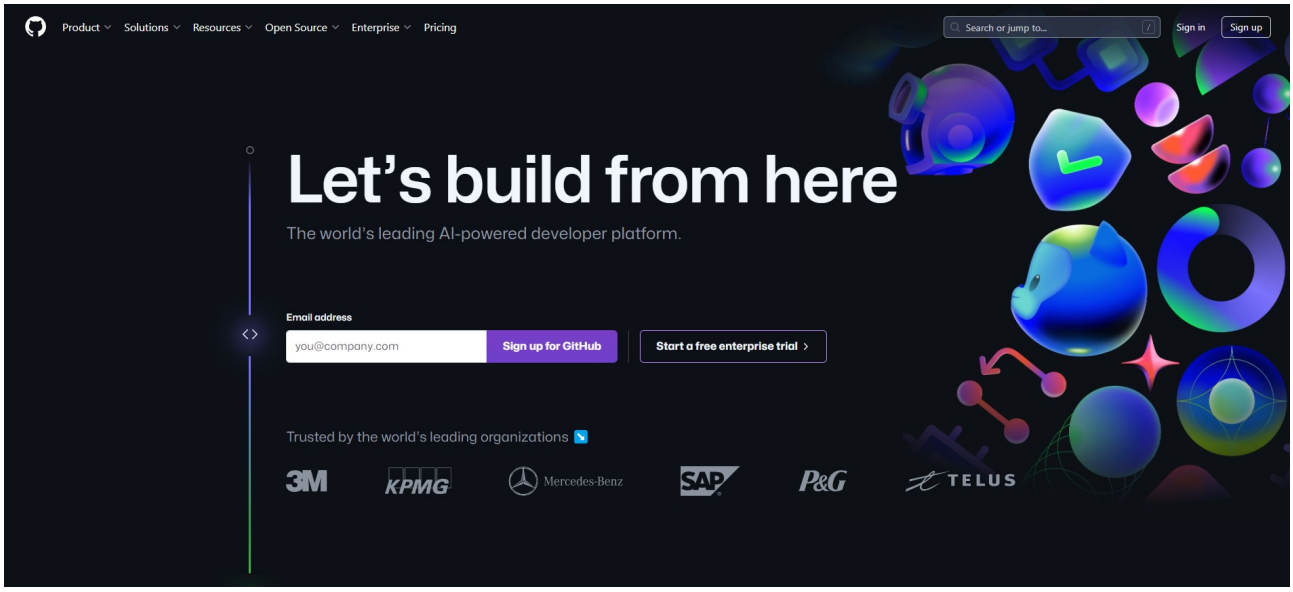
GitHub

The collaborative development platform GitHub allows us to create centralized and shared bare Git repositories.

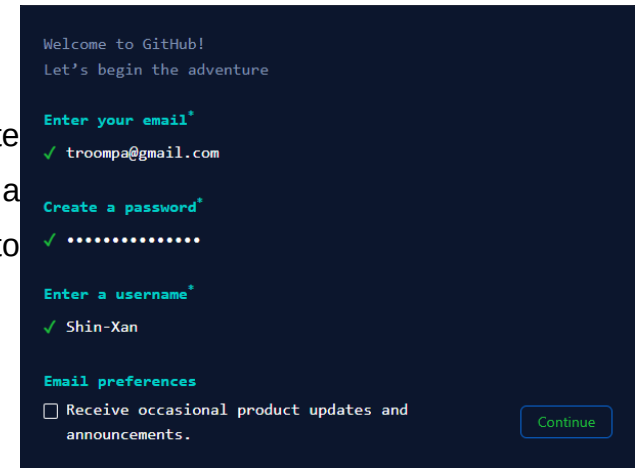
In this section, we will create an account on this platform and our first public repository.

A. Creating an Account

First, we go to the GitHub website (www.github.com) and create a free account. All we need to do is enter our email address where prompted and click the "Sign up for GitHub" button.



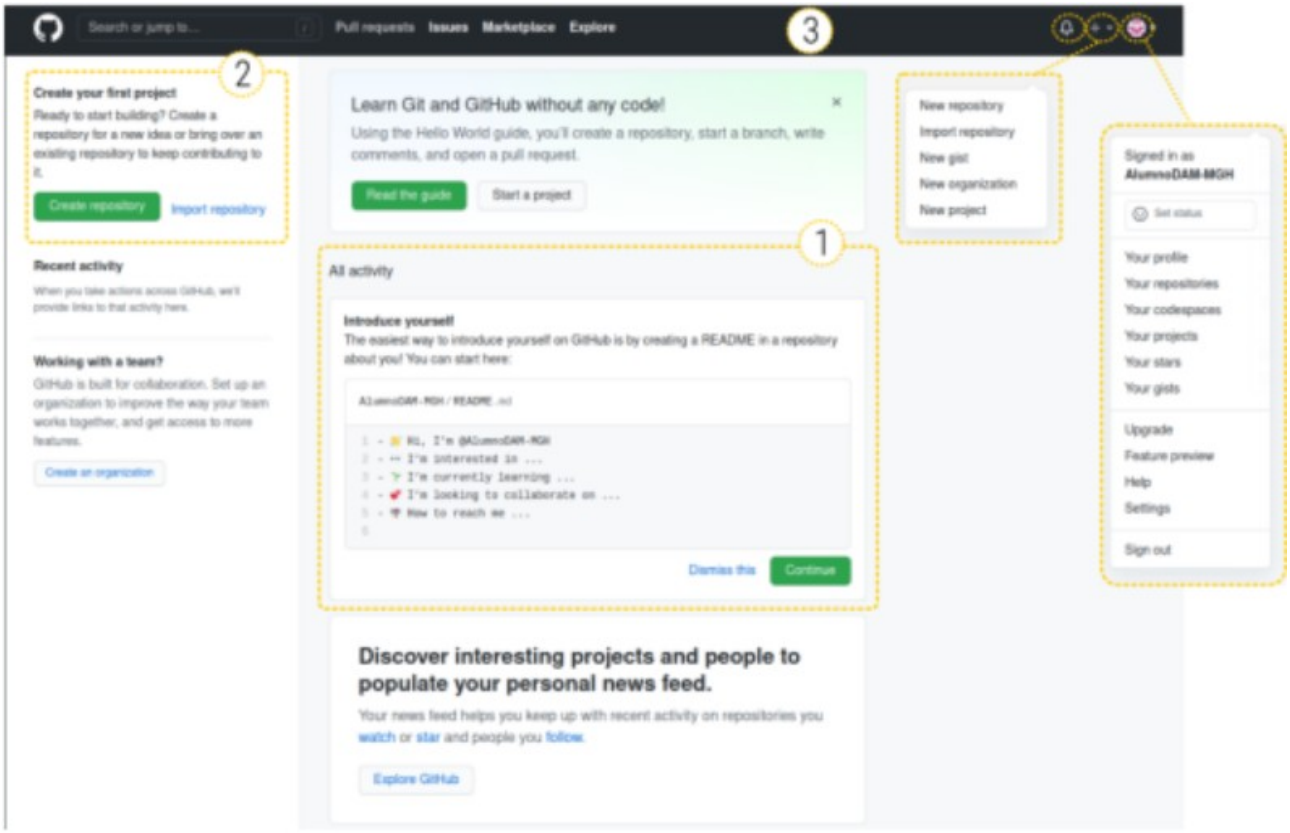
A short setup wizard will begin, where we confirm or enter our email if we haven't already, choose a password and username, decide if we want to receive updates, and finally verify our account by solving a small puzzle.



Once the setup is complete, we click "Create Account." At this point, we'll receive a verification code in the email we provided to complete the account creation.

After doing so, the initial GitHub screen will appear, presenting several options.

From here, we could create a repository by selecting "Create a Repository." For now, we'll click "Skip this for now" to go to the main screen, where we can access all the options GitHub offers.



On GitHub's main screen, we can highlight three areas:

1. The central column displays all recent activity, both our own and that of our contacts.
2. The left column shows information about our repositories, teams, etc. From here, we can start creating repositories.
3. At the top, there is a menu bar with links to the homepage, a search box, and a menu with four items:
    - **Pull Requests**: To access and validate merge requests from other users on our repositories.
    - **Issues**: To manage notes and suggestions related to our repository, whether for fixing bugs, requesting new features, or asking questions, among others.

- **Marketplace**: To add extensions and other functionalities to GitHub.
- **Explore**: To explore repositories GitHub suggests that might interest us.

Additionally, in this top bar, there are three icons:

- The **bell ico**n, showing if we have any notifications.
- The **Add ico**n (a plus sign), which helps manage new repositories, organizations, or projects. In a project, beyond repository management, we also have other options related to project management, such as task management.
- The **avatar** icon, which allows us to control various aspects of our account (profile, repositories, teams, etc.).

B. Creating a Personal Access Token (PAT)

Once we have our GitHub account, and before creating a repository, we'll create a personal access token (PAT).

When working from the command line with Git on our repositories hosted on GitHub, we'll need a way to authenticate with the platform. Traditionally, it was enough to provide our credentials (GitHub username and password) when pushing changes to the server. However, to improve security, since August 13, 2021, GitHub no longer allows this type of authentication from the command line and requires using HTTPS with a token generated by the platform.

Here's how to generate this token, which we'll need to store on our computer and use when prompted for a password:

Step 1: Locate the icon with your profile picture at the top right, and click on it to access "**Settings**."

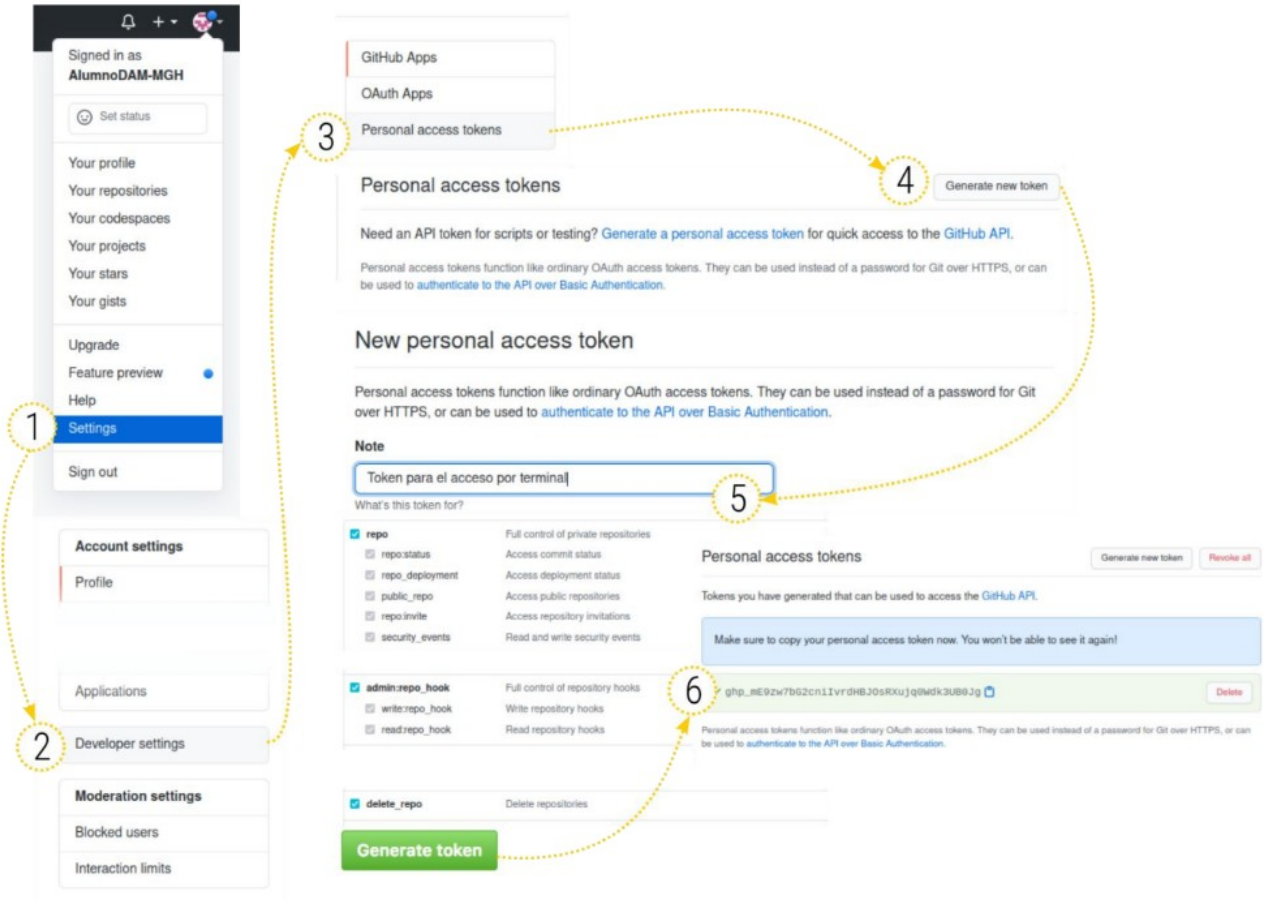
Step 2: Click on "**Developer Settings**" in the left-hand panel.

Step 3: On the developer settings page, click "Personal Access Tokens."

Step 4: In the "Personal Access Tokens" section, select "Generate New Token."

Step 5: Provide a **description** for your token, such as "Token for terminal access," as well as its **scope** or permissions. In our case, select all repository interaction options (repo, admin.repo_hook, and delete_repo), then click "Generate Token."

Step 6: A token will be generated, which you should copy and store for use as a password in the terminal. To copy it, click on the copy icon to the right of the token. Be careful, as this token will not be shown again for security reasons once you leave the page.



You can find more information about this process at the following links:

- [Token authentication requirements for Git operations]
- [Creating a personal access token]

C. Creating a Repository

Now let's create our first repository on GitHub. From the main window, follow the steps below.

Step 1. Create a Repository:From the "Create your First Project" section on the left, or from the first item in the Add (+) submenu, click "New Repository." This will take us to the page for creating a new repository.

Step 2. Fill in Repository Information: To create the repository, we need to provide the following information:

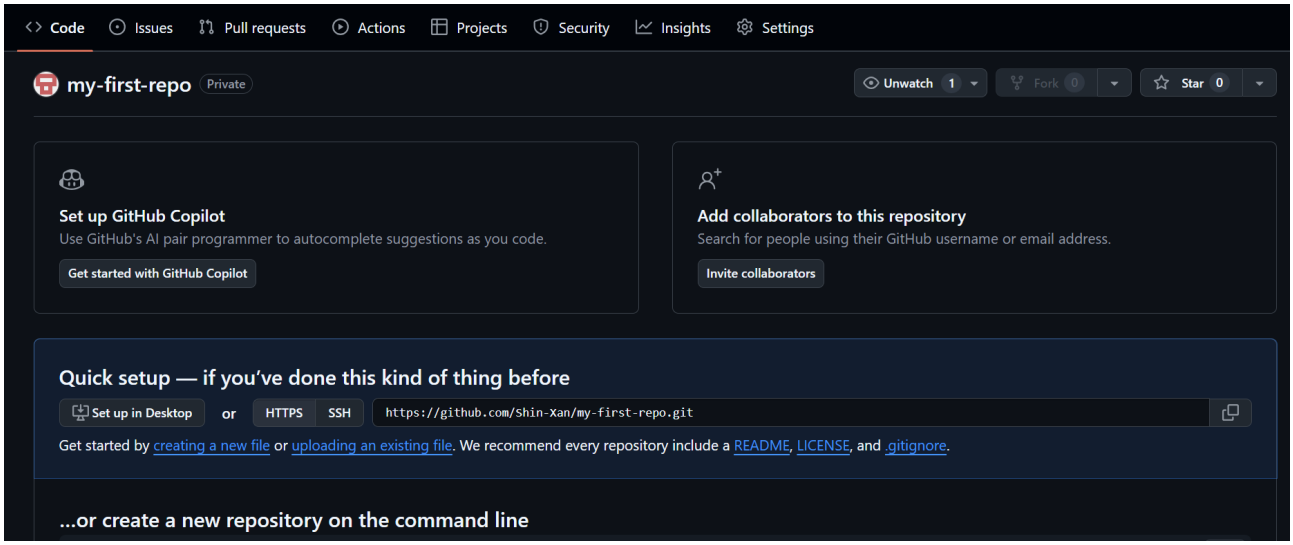
- Repository name.
- Description.
- Whether it will be public or private.
- Initialization options like creating a README.md file, a .gitignore, or adding a license.

Pay attention to these details:

- The repository name is linked to the user, so different users can have repositories with the same name. This also facilitates importing repositories from other users. The format will be `username/repository`.
- We've chosen to initialize the repository with a README file but without a .gitignore or license, though we could skip this step and initialize the repository manually. If we leave it uninitialized, GitHub will show us the steps to complete the initialization via the terminal.
- Note that the default branch created is `main`. In the past, the default branch in Git was called `master`, but since October 2020, GitHub has switched to `main` due to the negative connotations of the master/slave terminology.

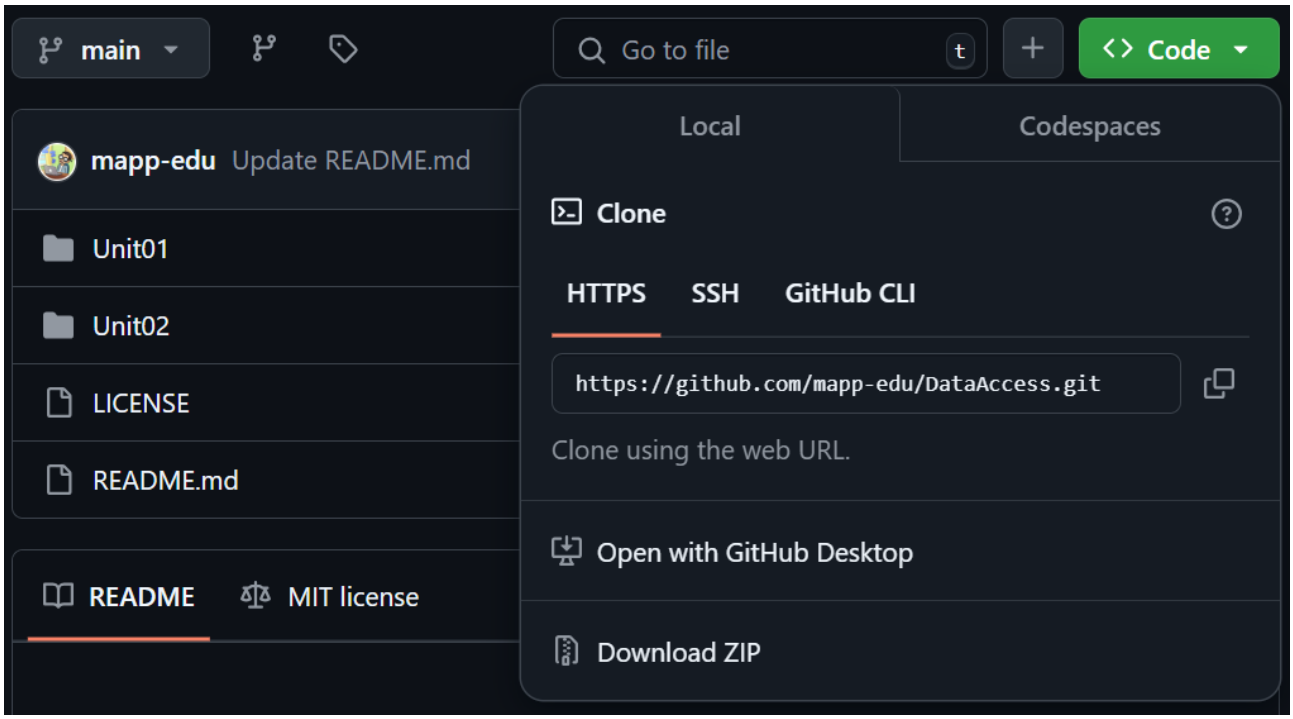
Step 3. Create the Repository: Once we've filled in the details, click "Create Repository," and our repository will be ready.

Repositories we create will have a webpage where we can view, manage, and modify the repository's code. Key elements on this page include:

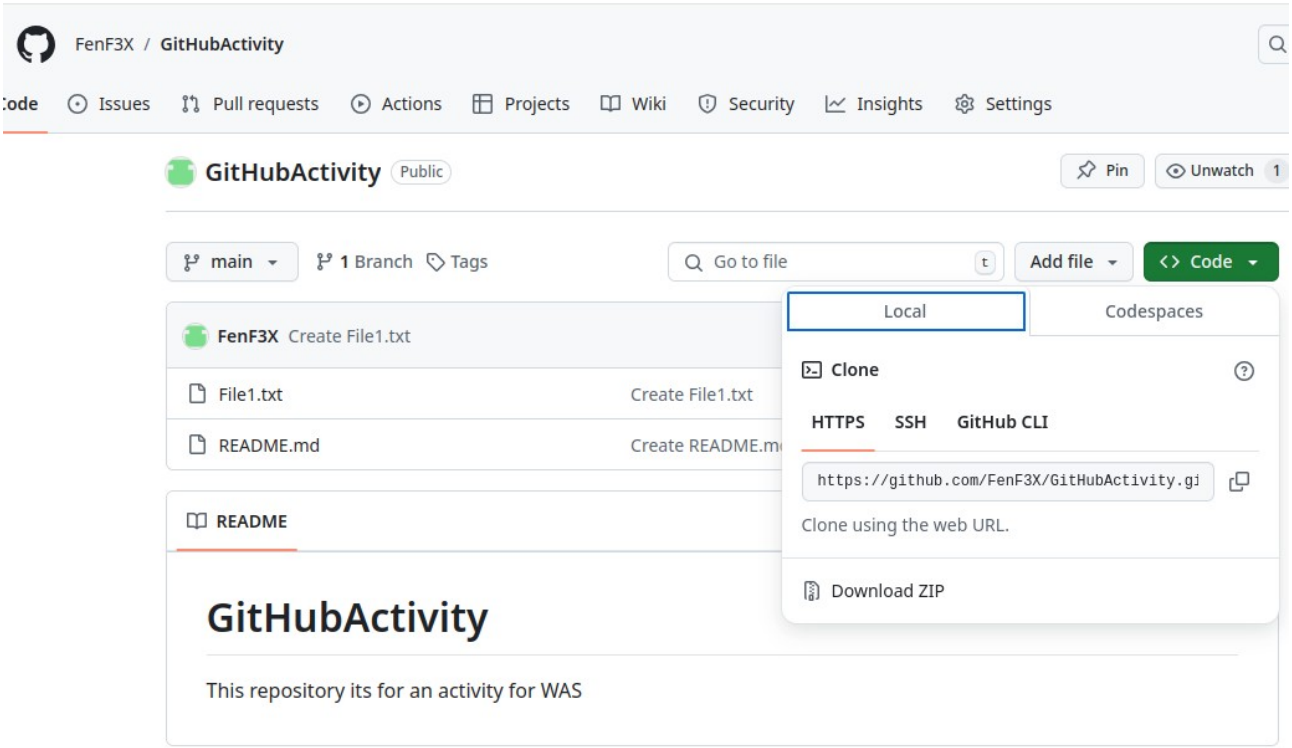


- The repository's path at the top left, which shows the folder structure and allows us to navigate back through directories.
- Information about how many users are watching, starring, or forking the project at the top right.
- A menu with options for the project: **Code**, **Issues**, Pull requests, **Wiki**, or **Settings**, among others.
- A central area displaying the repository's content, including commits or HEAD information. This area also shows the content of the README.md file, which can be used to describe the project or the folder's contents.

Step 4. Cloning the Repository: Now, we'll download the repository to our local machine. Look for the green "Code" button, then expand the submenu with the cloning options:



Output:



We can clone via HTTPS, SSH, or GitHub's new command-line interface (CLI). We can also download the project in a zip file (without version control information).

We'll use HTTPS for this example, so we copy the URL `https://github.com/user/my-first-repo.git`.

Once copied, open a terminal on your computer and use this URL as the origin to clone the repository:

```
$ git clone https://github.com/user/my-first-repo.git
```

Output:

```
prserver@prserver:~$ git clone https://github.com/FenF3X/GitHubActivity.git
Cloning into 'GitHubActivity'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
prserver@prserver:~$
```

To verify the remotes, navigate to the folder:

```
$ cd my-first-repo/
$ git remote -v
```

Output:

```
prserver@prserver:~/GitHubActivity$ git remote -v
origin  https://github.com/FenF3X/GitHubActivity.git (fetch)
origin  https://github.com/FenF3X/GitHubActivity.git (push)
```

Step 5. Adding Content: Let's add some content to our local Git repository and sync it with the remote one. We'll create a new folder within the project called `images` and place an image in it. Here's how to do it:

```
$ mkdir images
$ cp ~/Downloads/WAD.png images/
$ git add images/
```

Next, we'll edit the `README.md` file to embed the image we added:

```
# Our First GitHub Repository


This is our first GitHub repository, a collaborative development platform that allows us to create centralized and shared Git repositories.


![Web Aplication Deployment Logo](images/WAD.png)
```

Then, we add the changes to be included in the next commit:

```
$ git add README.md/
```

Step 6. Committing and Pushing Changes: Let's check the repository's status:

```
$ git status
```

Output:

```
prserver@prserver:~/GitHubActivity$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
        new file:   images/WAD.png
```

We see that there are several new files ready for commit. We make the commit:

```
$ git commit -m "Added images folder"
```

Output:

```
prserver@prserver:~/GitHubActivity$ git commit -m "Added images folder"
[main 1786955] Added images folder
 2 files changed, 5 insertions(+)
 create mode 100644 images/WAD.png
```

After committing, we verify that our main branch is one commit ahead of the remote:

```
$ git status
```

Output:

```
prserver@prserver:~/GitHubActivity$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Finally, we push the changes to the remote repository using:

```
$ git push
```

Output:

```
prserver@prserver:~/GitHubActivity$ git push
Username for 'https://github.com': FenF3X
Password for 'https://FenF3X@github.com':
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 581 bytes | 193.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/FenF3X/GitHubActivity.git
   f48703d..1786955  main -> main
```

At this point, GitHub will ask for your credentials to push to the repository. Remember to use your personal access token instead of your GitHub password. When you refresh the GitHub repository page, you'll see that the content has been updated.

Output (main screen of your github repo):

IMPORTANT: Branches and Forks

Let's remember that GitHub is one of the largest hosting platforms for open-source software projects. One of the main principles of such projects is the ability to share code, modify it, and adapt it to our needs. GitHub allows us to create full copies of a project in our user account. This is known as a **fork**, and it enables us to adapt projects to our needs or create new projects based on an existing one.

The concept of a fork is different from that of creating a branch. A fork generates a complete physical copy of the repository with the intention of starting independent development from the original. In contrast, a **branch** represents a split within the original repository, aimed at making changes that will later be merged back into the main branch.

However, with a fork, you can still collaborate on the original project even if you are not part of the development team. When you believe that the changes you've made can be beneficial or fix an issue in the original code, you can submit a **pull request** to the original project for the developers to consider merging your code with theirs.