**GitHub**

**What is GitHub?**



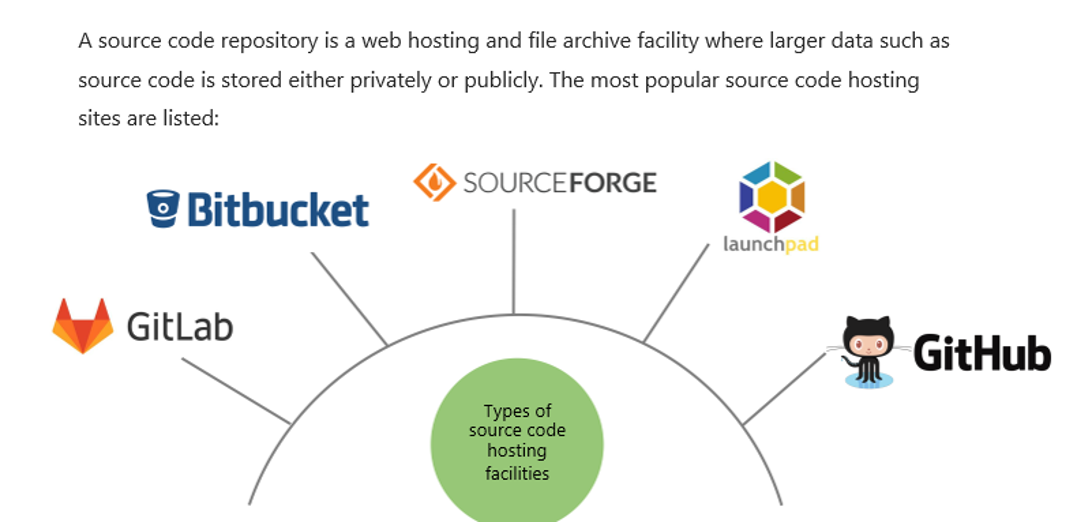
GitHub is a Git repository hosting (Source Code Hosting) service , but it adds many of its own features. It is a web-based platform used for version control and it provides a Web-based graphical interface. It also provides access control and several collaboration features, such as a wikis and basic task management tools for every project.

Like GitHub, there are other source code hosting platforms but GitHub is the most popular one.

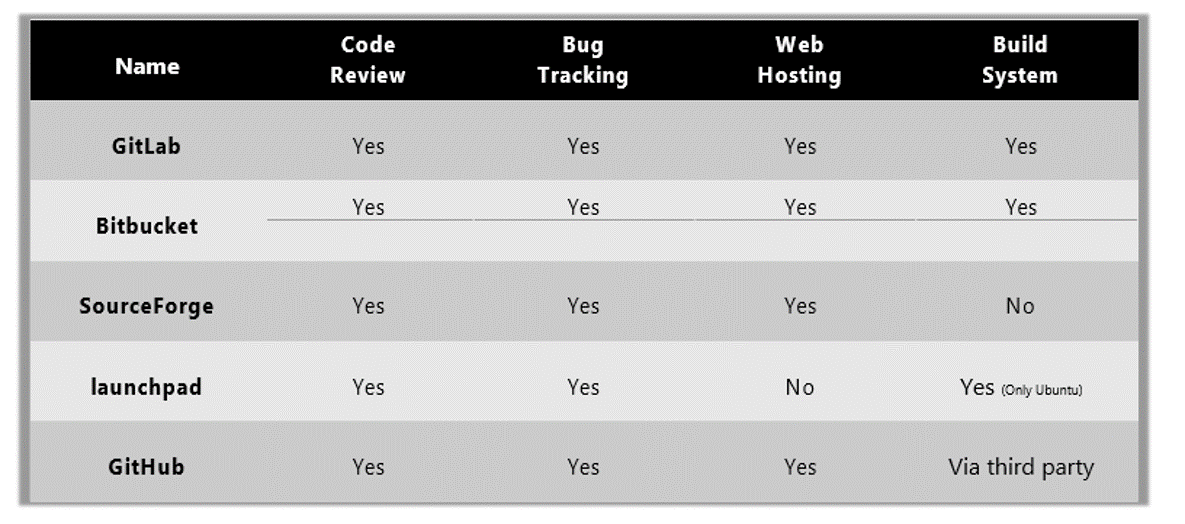
**What is the difference between Git and GitHub?**

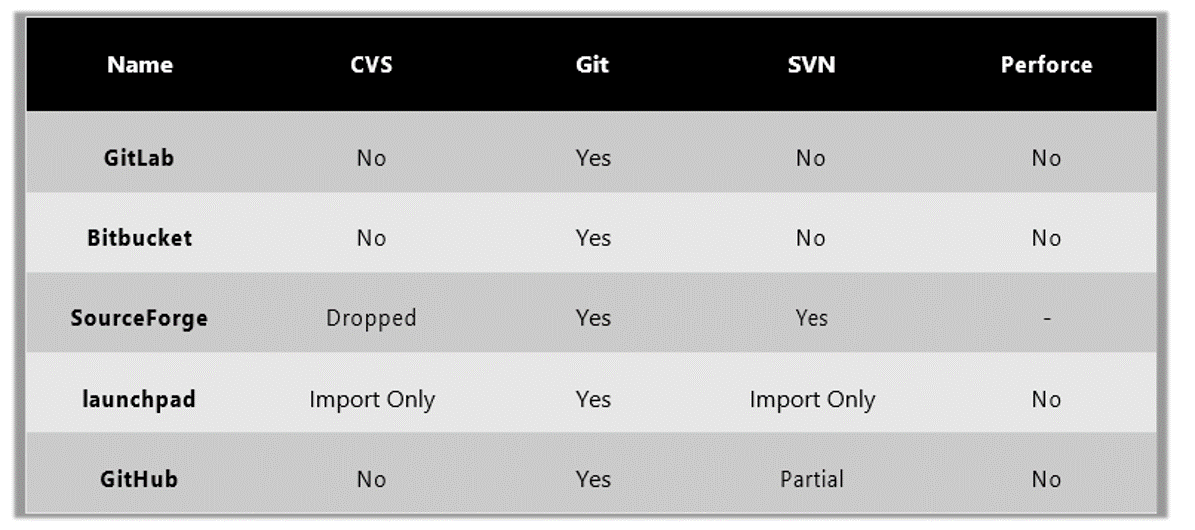
Git is a version control system that lets you manage and keep track of your source code history locally. GitHub is a cloud-based hosting service that lets you manage Git repositories.

**Source Code Hosting Facilities**

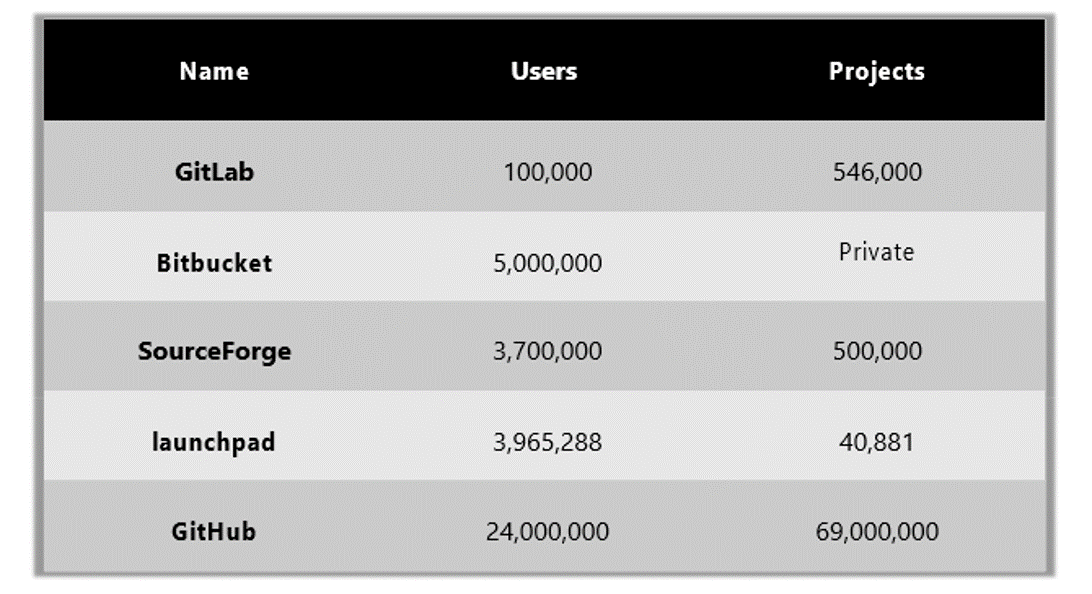


This is a nice graph which shows the most important features of the hosting sites



And this graph shows the hosting sites supporting the VCS

GitHub is the most popular one as you see .

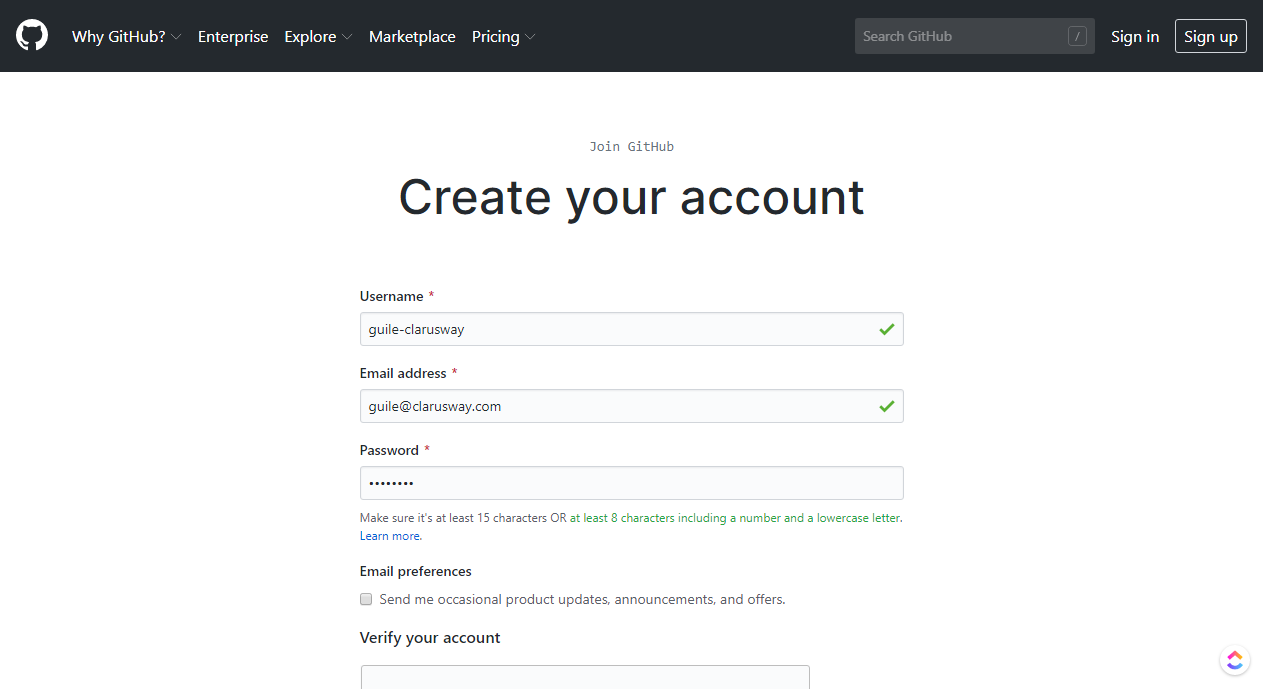


**Creating GitHub Account**

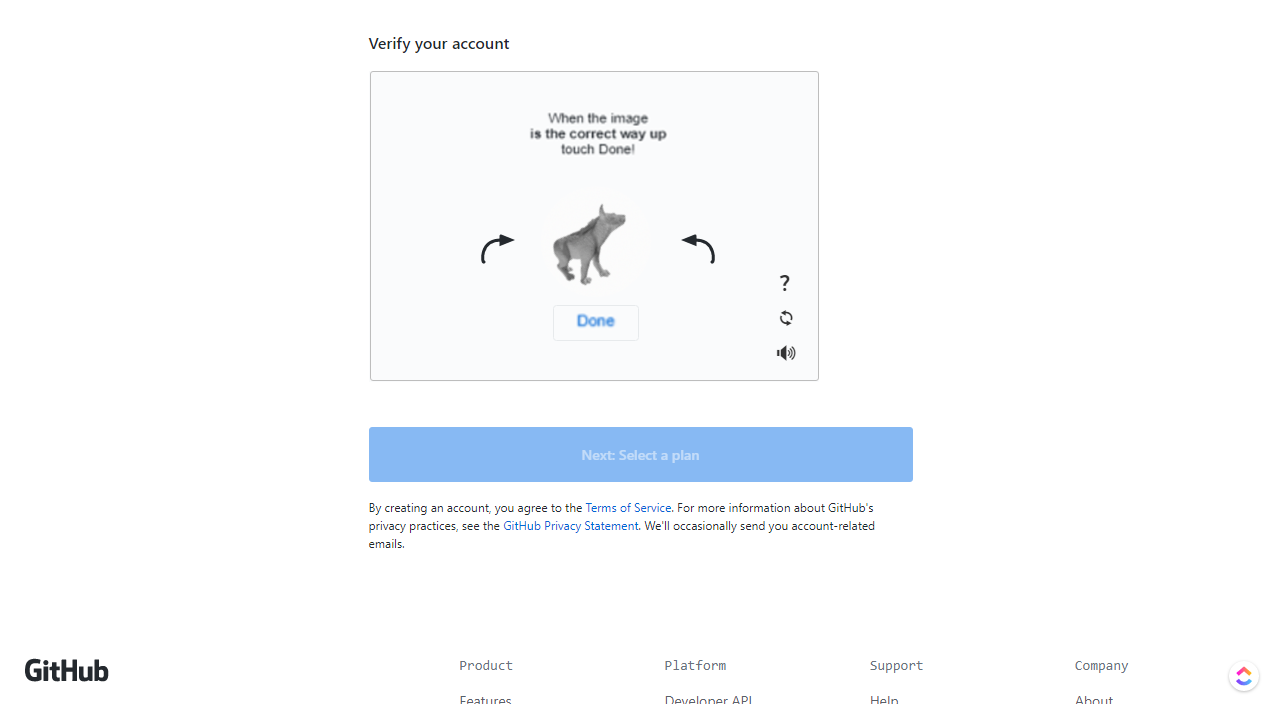
Let's go to web page and open an account on GitHub.

[https://github.com](https://github.com/)

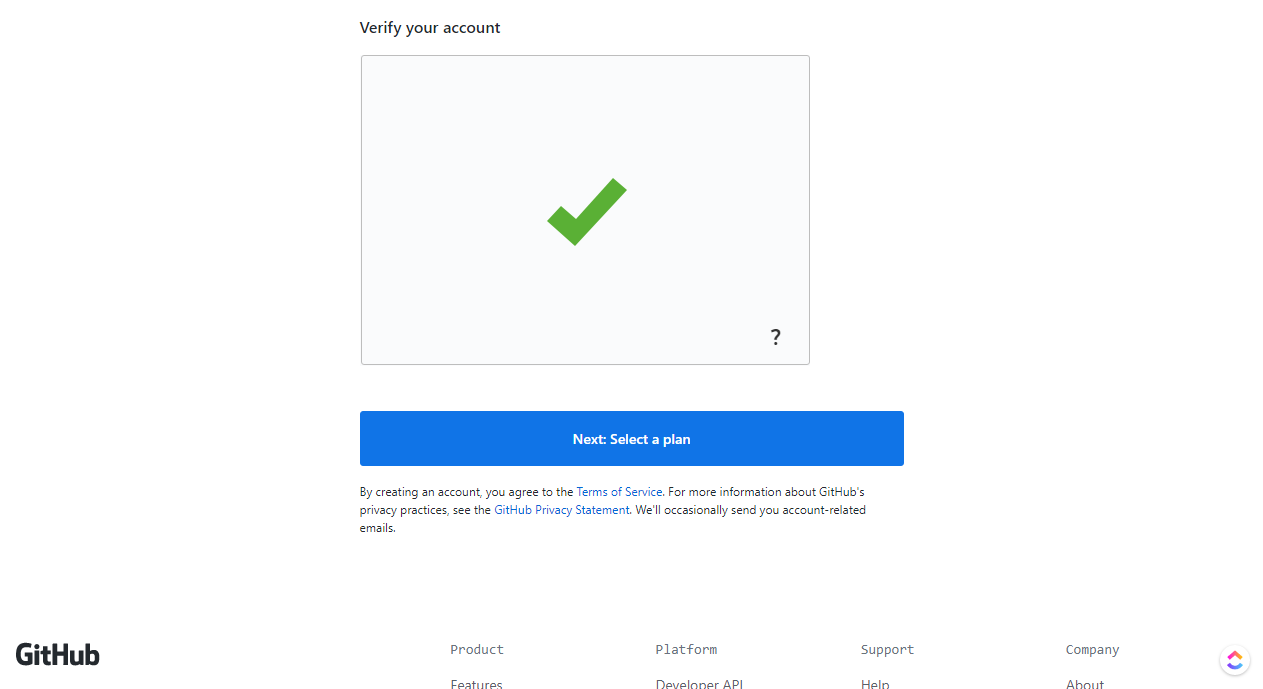
Enter a username, a valid e-mail address and a password.



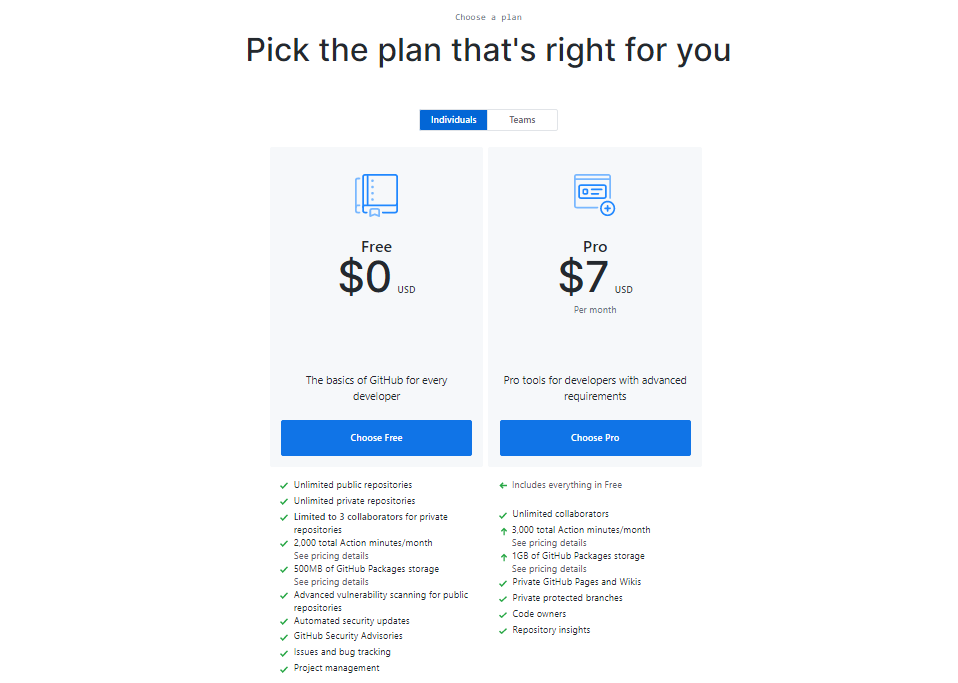
Then solve the puzzle to prove you are not a robot.



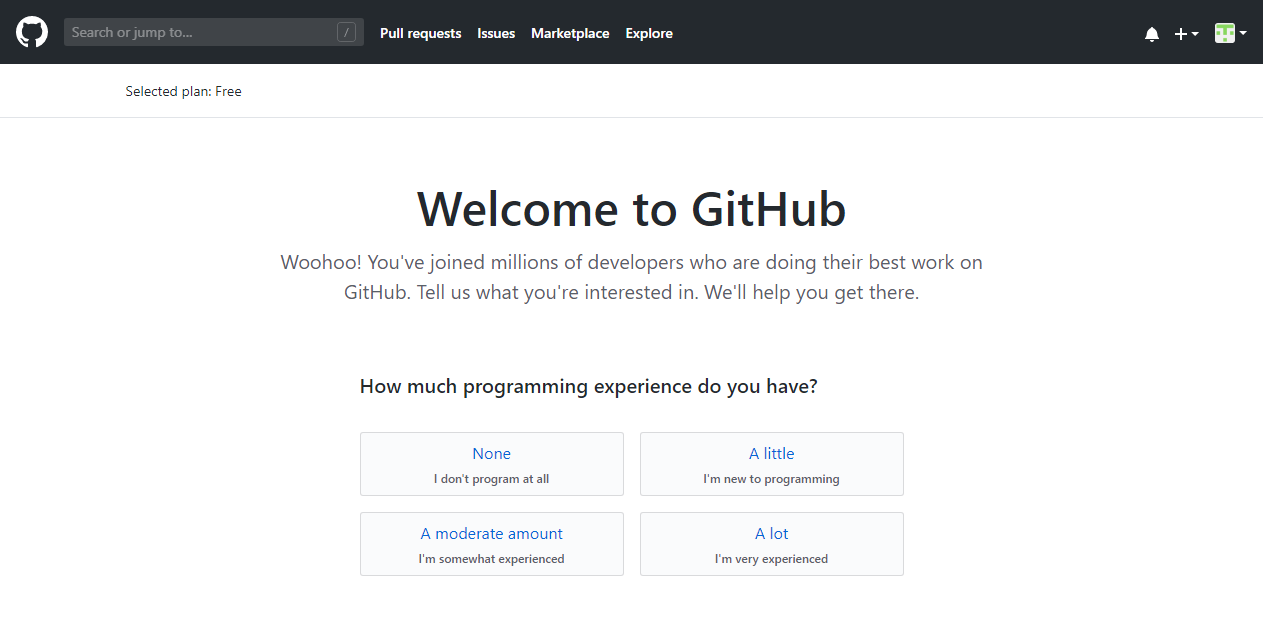
After verification click "Next" to choose a plan.

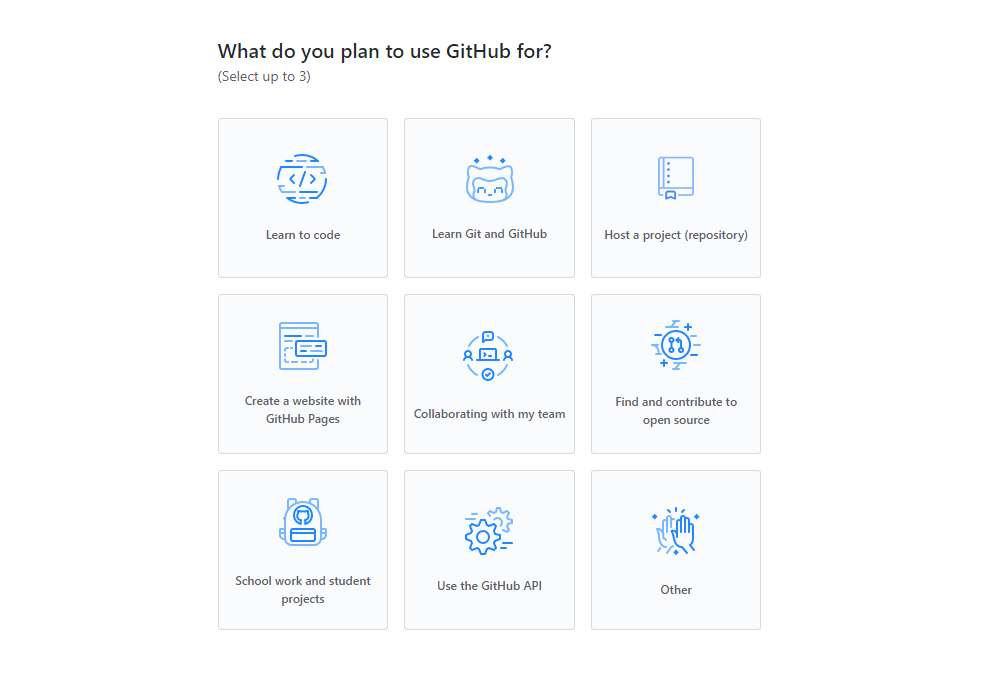


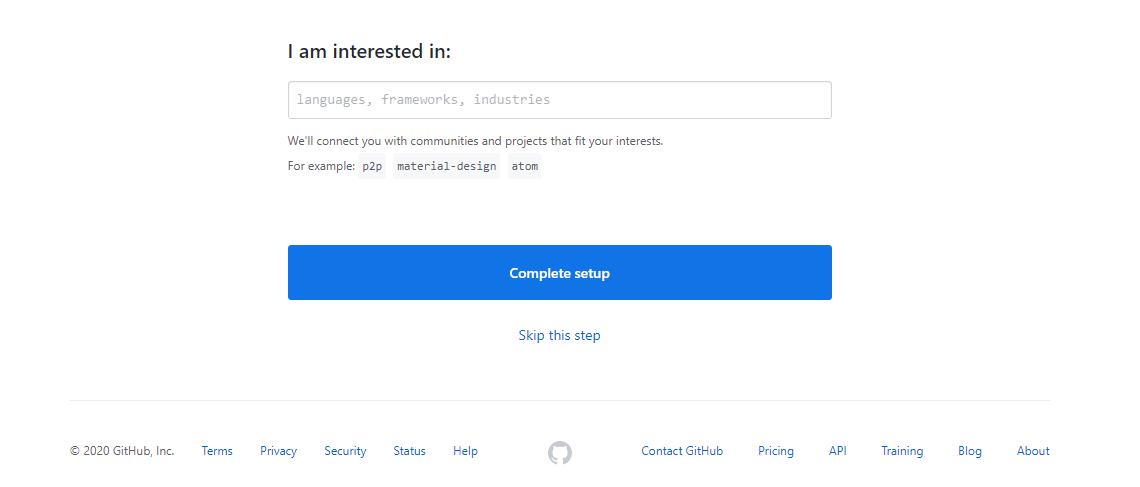
Then choose a plan (Free $0 USD), you may upgrade it later if you like.



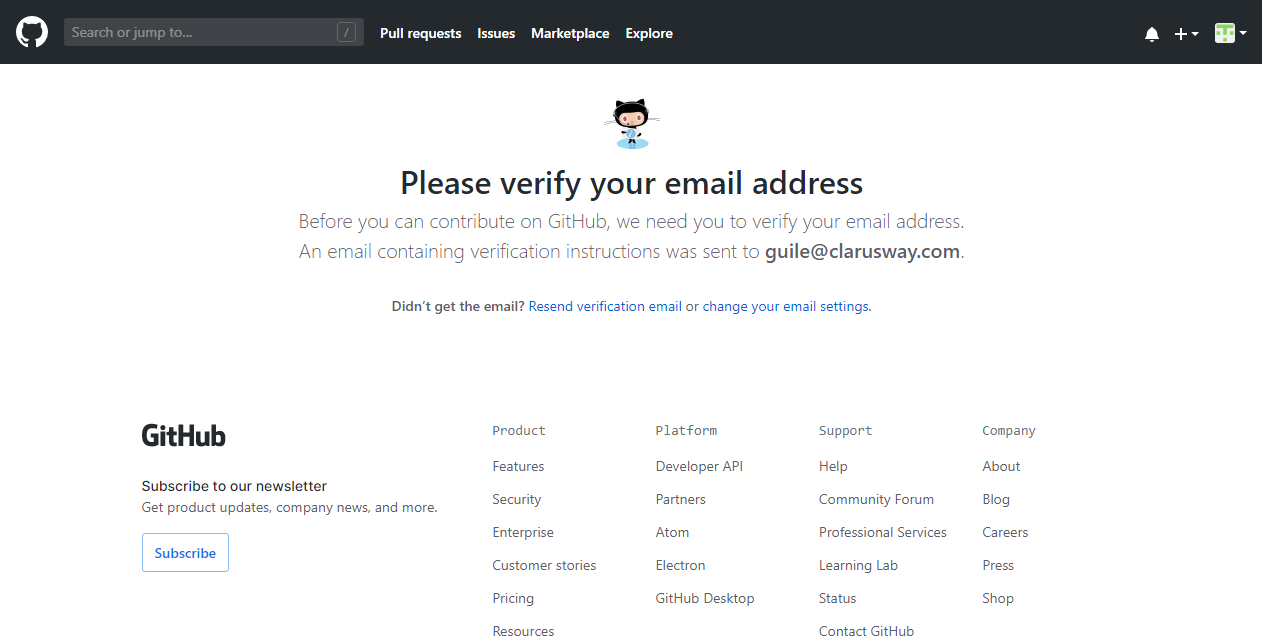
After GitHub's welcome, you can help them by answering some questions. Just pick the ones that fit you or you can proceed to the end of the page to skip.



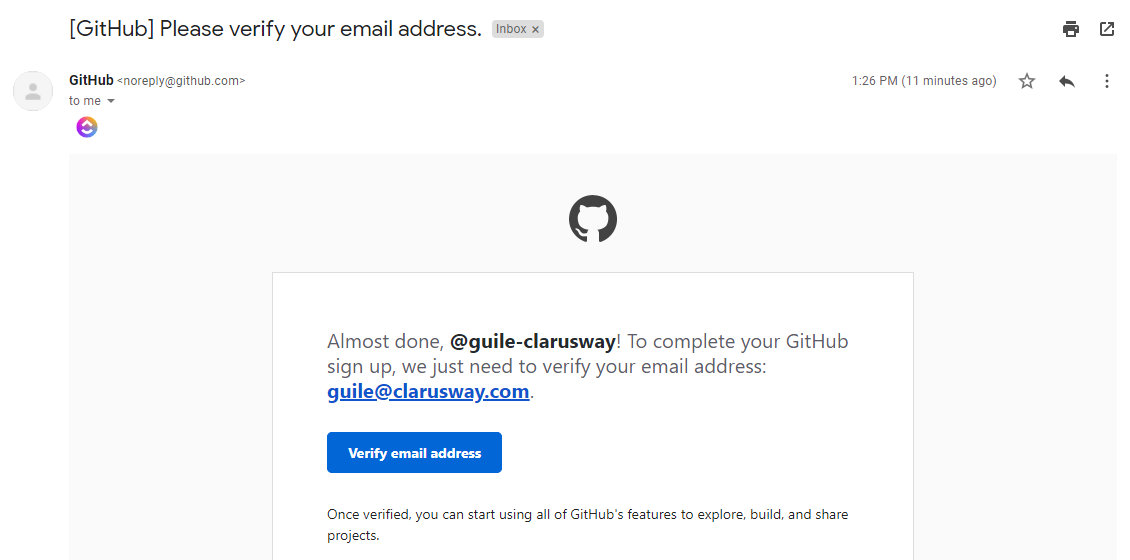




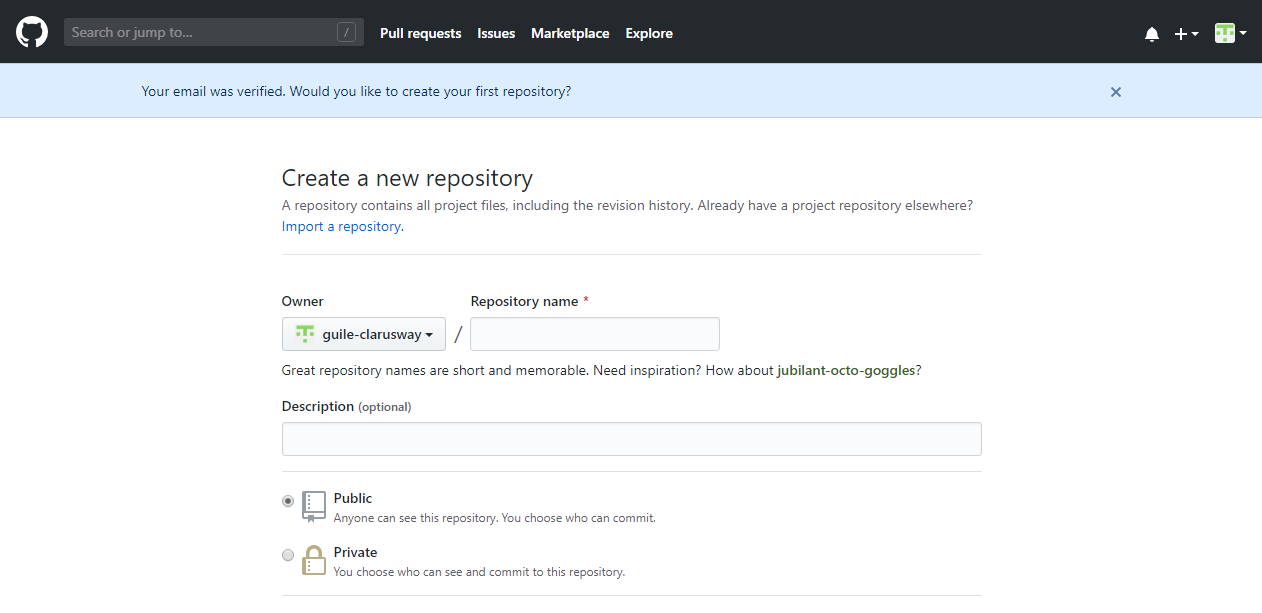
After completing the set up, a verification email will be sent by GitHub.



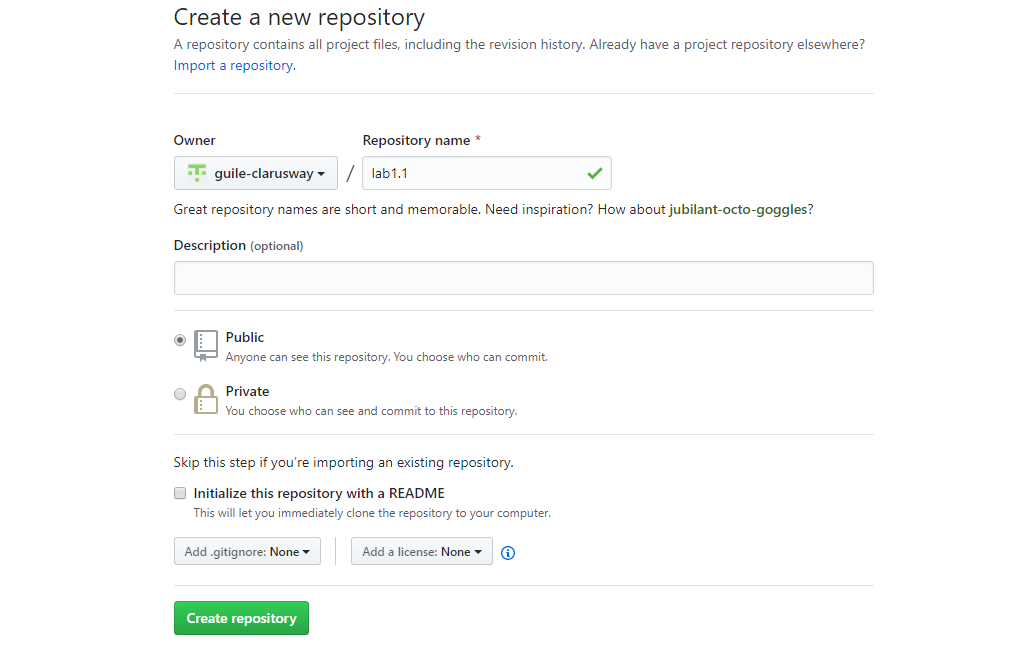
Go, check the email, and verify your email adress.



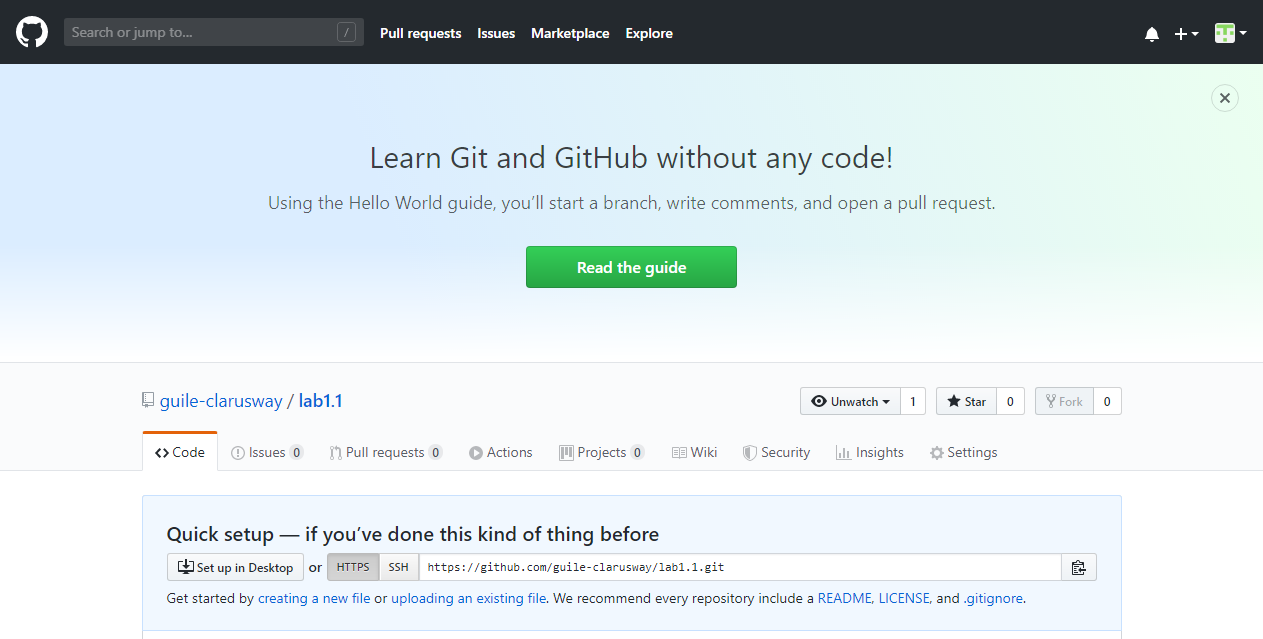
**Create Your First Repo**



Type the name of your first repository, "lab1.1" in this case, and click "Create repository".



Your first repo named **lab1.1** is created. Note down your repo URL, since you will need the repo URL when configuring your local repo on your computer.



My repo URL :

[**https://github.com/fikretgit/lab1.1.git**](https://github.com/fikretgit/lab1.1.git)

**----------------------------------------------------------------------------------**

**sayfadan kopya;**

**…or create a new repository on the command line**

**echo "# lab1.1" >> README.md**

**git init**

**git add README.md**

**git commit -m "first commit"**

**git branch -M main**

**git remote add origin https://github.com/fikretgit/lab1.1.git**

**git push -u origin main**

**----------------------------------------------------------------------------------**

**…or push an existing repository from the command line**

**git remote add origin https://github.com/fikretgit/lab1.1.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.**

[**Import code**](https://github.com/fikretgit/lab1.1/import)

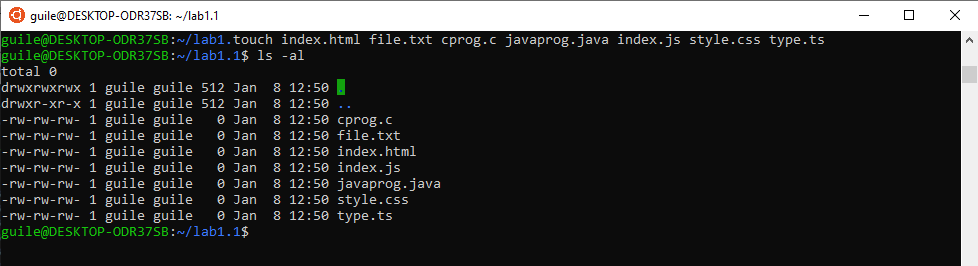
**Pre-Class’dan devam;**

Now open Git Bash or a terminal in your computer and let's begin to create our local repo.

First, create some files by entering "touch" command. These files are just empty demo files and we will use them only to interact with GitHub. For the sake of this demo and for a better understanding create these files under the "lab1.1" folder which has the same name with our GitHub repo. **This is not mandatory** **but makes sense.**

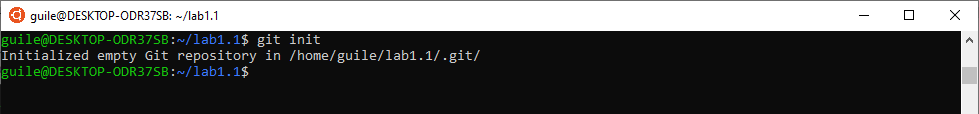
**$ touch index.html file.txt cprog.c javaprog.java index.js style.css type.ts**

**$ ls –al**



You then execute "git init" command to create an empty local repo.

**$ git init**



Initialized empty Git repository in "home/guile/lab1.1/.git".

"Git add" to add the files to staging area from the working area.

**$ git add .**

**TİPS**

* If you have a warning like this, you can ignore it for now.
* warning: LF will be replaced by CRLF in lemonade.txt.

The file will have its original line endings in your working directory

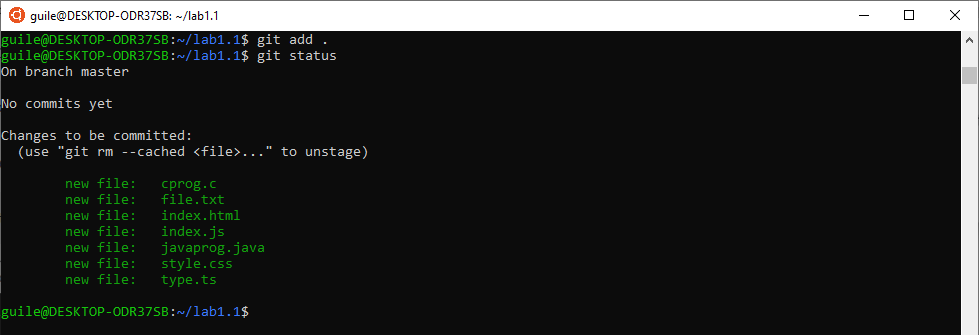
In Unix systems the end of a line is represented with a line feed (LF). In windows a line is represented with a carriage return (CR) and a line feed (LF) thus (CRLF). when you get code from git that was uploaded from a unix system they will only have an LF.

If you are a single developer working on a windows machine, and you don't care that git automatically replaces LFs to CRLFs, you can turn this warning off by typing the following in the git command line

**git config core.autocrlf true**

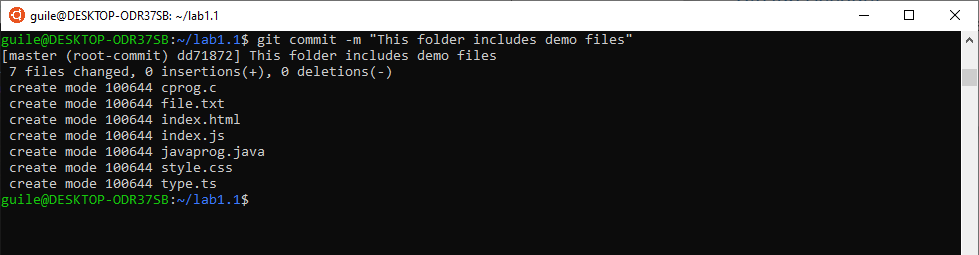
And now we can see the status of our files with the "git status" command.

**$ git status**



Let's commit our files to local repo with the command "git commit –m".

**$ git commit –m “**This folder includes demo files**”**



Now it is time to add our file to GitHub with command "git remote add". Use your own repo URL.

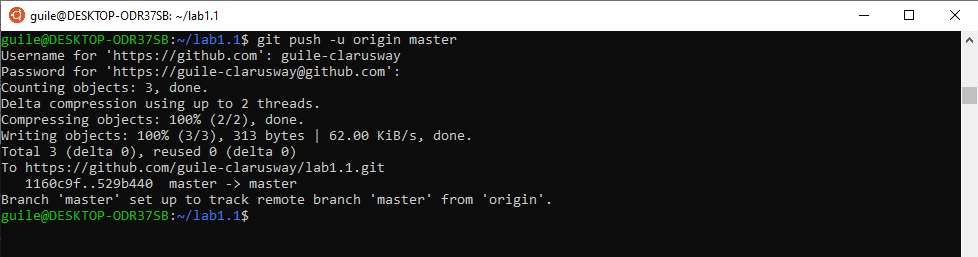
**$ git remote add origin YOUR\_REPO\_URL**

# For example: git remote add origin https://github.com/guile-clarusway/lab1.1.git

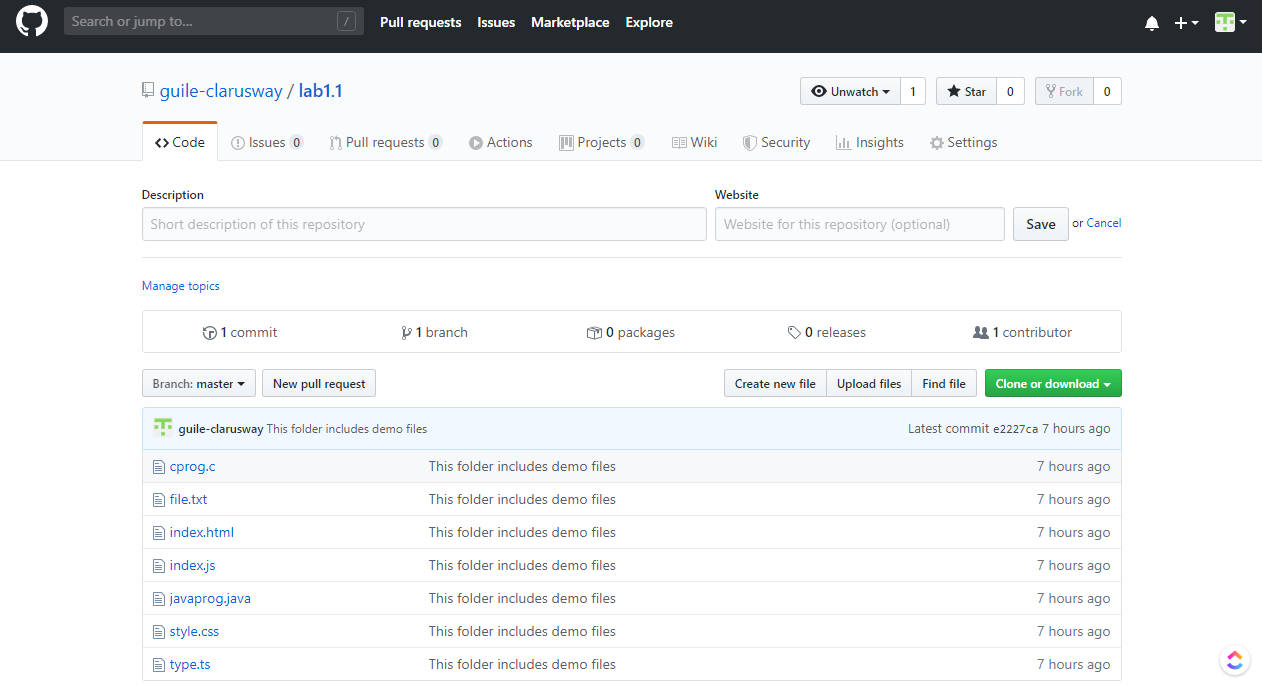
Local git instance added to remote repository. And then we can go ahead and push our files to remote repository we have just created on GitHub. Enter your GitHub username and password if asked.

**$ git push -u origin main**

This is what you may see after the command "git push –u origin main" executed.



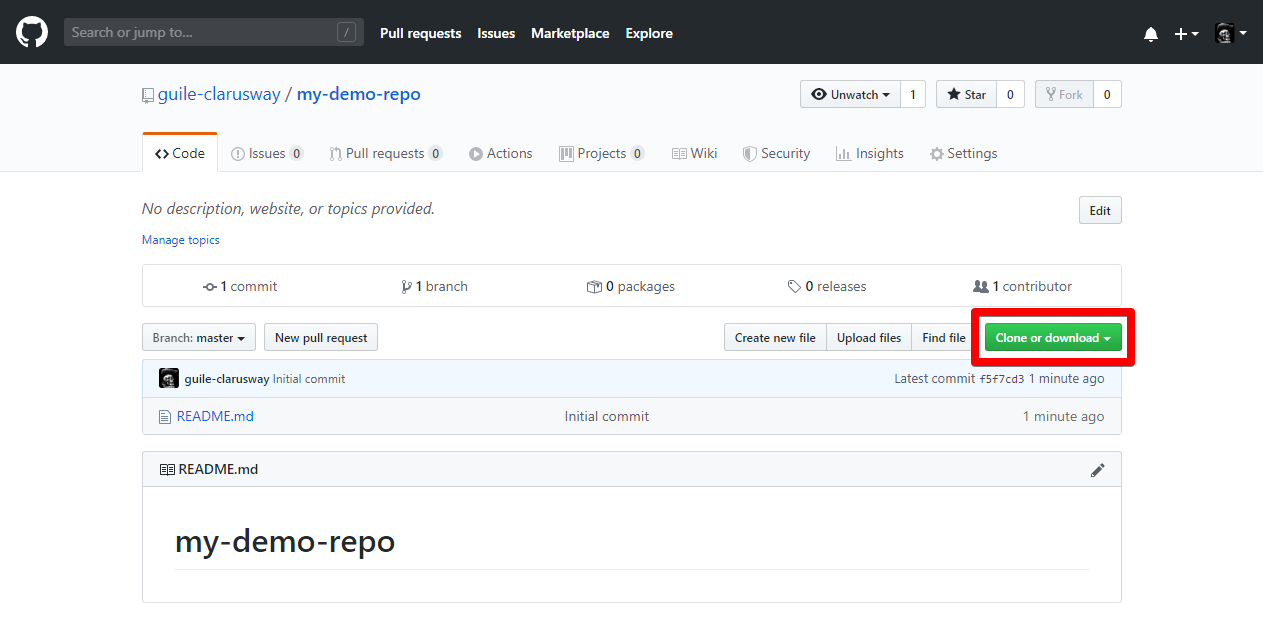
And this is what it looks like in GitHub after "git push command" executed.



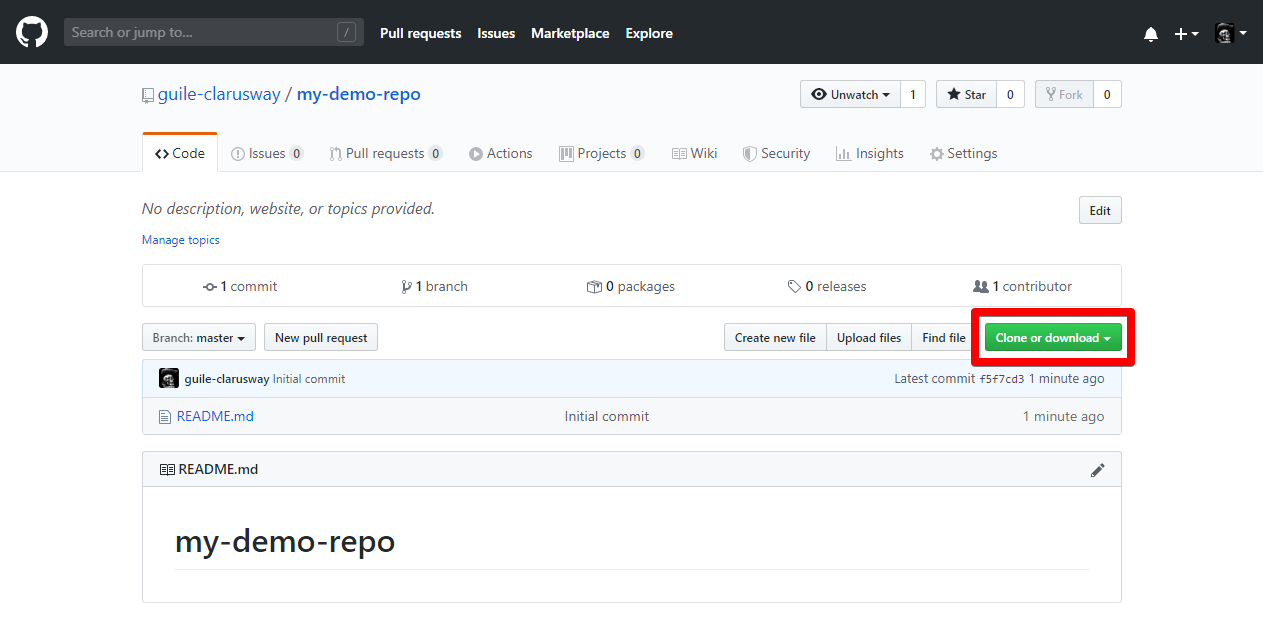
**Creating a Remote Repo on GitHub**

In this lesson, we will work both on local Git repo and remote [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229) repo. We will see the interactions between them.

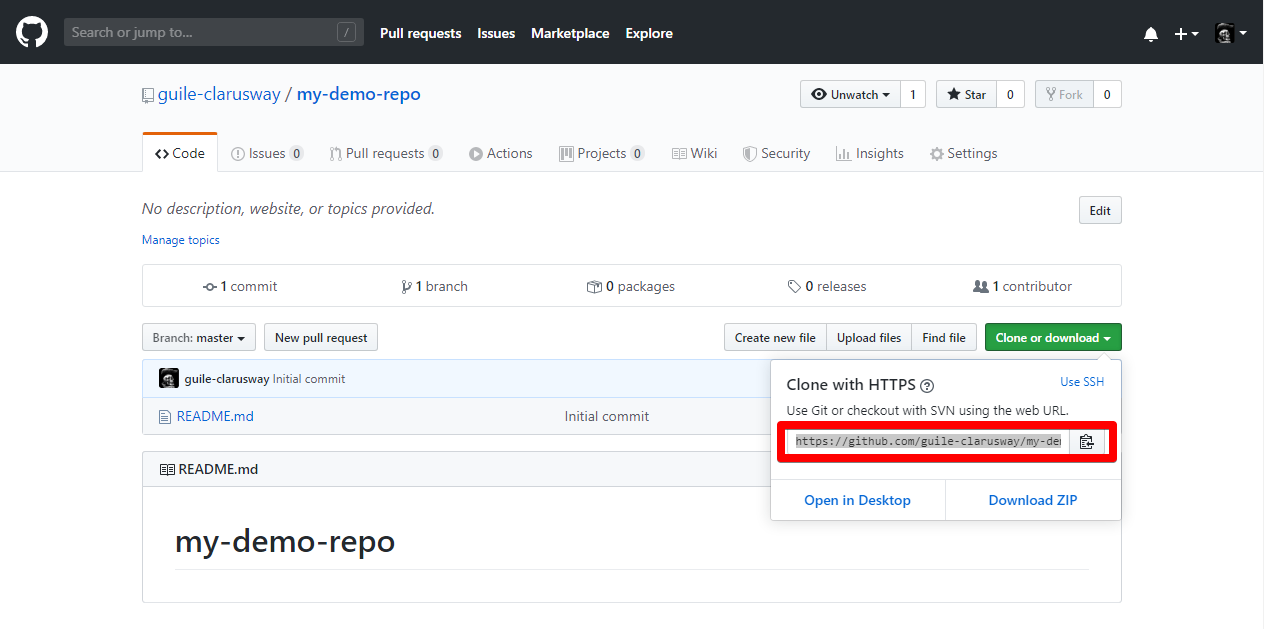
Let's go and sign into [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229), then create a new repository but don't forget to select initiating a **Read Me** file. This will automatically create our first commit, so later we will be able to clone it to our local repo and work on it. For this demo we will name our repo *my-demo-repo*. Click **Create repository** when you are done.



New repo has been created. Now click on the **Clone or download** button.



Then copy the URL of the repo. We will use this URL later on cloning.



**Cloning Remote Repo to Local**

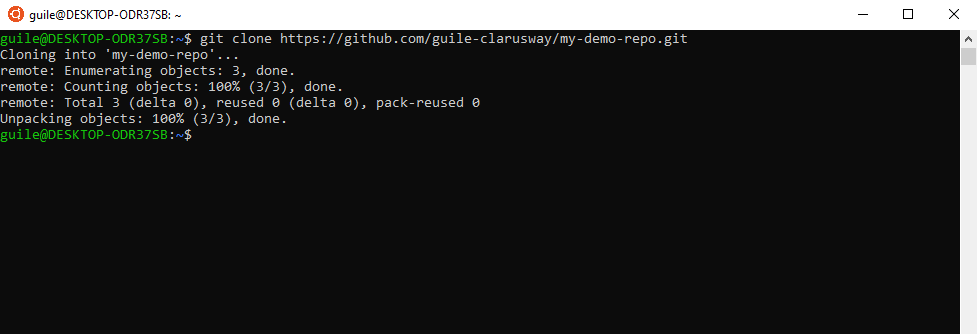
Now that, our repo is created and ready, we can start working on it. But remember, Git is a Distributed VCS and some other collaegues in the same project may need to work with the same repo. Thus, we will clone it to our local repo and work on it locally.

Open GitBash/Terminal and type the command below. Use your own repo URL which you copied after creating it.

**$ git clone YOUR\_REPO\_URL**

That is the example for this case. You will use **your own repo URL**.

**$ git clone https://**[**github**](https://lms.clarusway.com/mod/lesson/view.php?id=2229)**.com/guile-clarusway/my-demo-repo.git**



If you see something like that you have successfully cloned it to your local repo. Let's check,

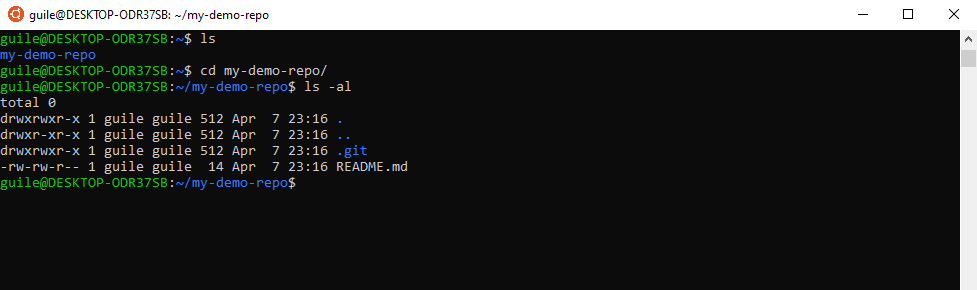
**$ ls**

and open the folder.

**$ cd my-demo-repo**

Now let's see what is in the repo.

**$ ls -al**



You should notice that **.git** already exists in addition to the **README.md** which is the only file in the repo. You don't need to use git init to initiate a repo when you clone it from remote.

Now we can make changes on our repo locally.

**Creating a New Branch**

Let’s assume we have a new idea but we are not exactly sure about it. We want to have an experiment. Branches are the keys to this kind of situation. You can create a new branch, make some progress on it and you can go back to the master branch if you are not satisfied with the new branch. You can think branches as a reference or bookmark to a specific commit. For this demo, we will create a new branch and work on it.

First, check your current branch

**$ git branch**

Then create a new branch. We will use **demobranch** for this lesson.

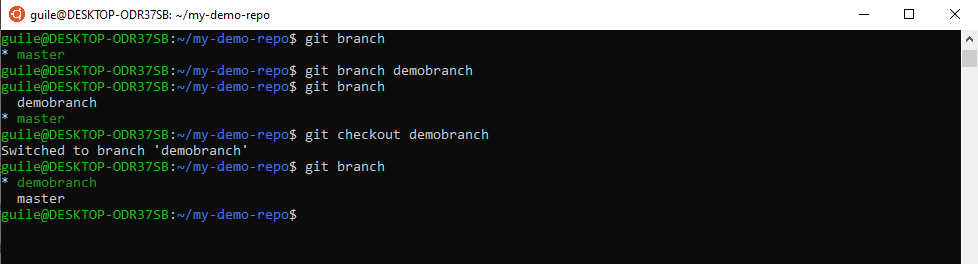
**$ git branch demobranch**

Check again and as you see the new branch is created but we are still on the master branch since it's green colored. Now switch to the demobranch

**$ git checkout demobranch**

**$ git branch**

We are now on the new branch and we can start working.

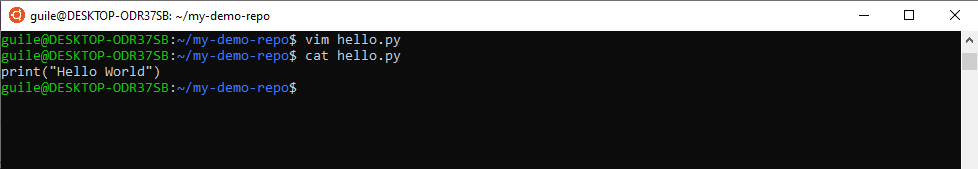


Let's add a simple Python file to our project. Use your favorite editor, add print("Hello World") to the file.

**$ vim hello.py**

You can check it if you want.

**$ cat hello.py**



We can stage,

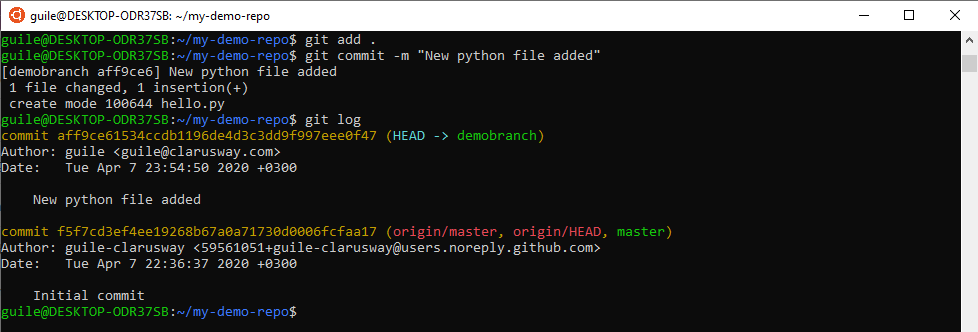
**$ git add .**

And commit it now.

**$ git commit -m "New python file added"**

Let's check our log

**$ git log**



**Pushing Branch to Remote**

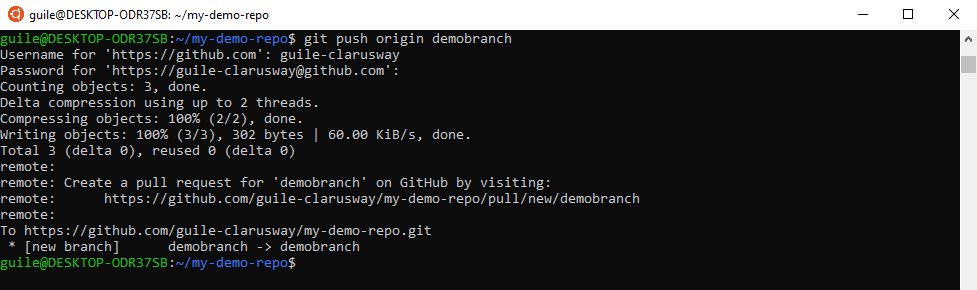
Now that we are done with the local repo, let's push it to remote. Command below will upload our local repo content to the remote repo.

**$ git push origin demobranch**

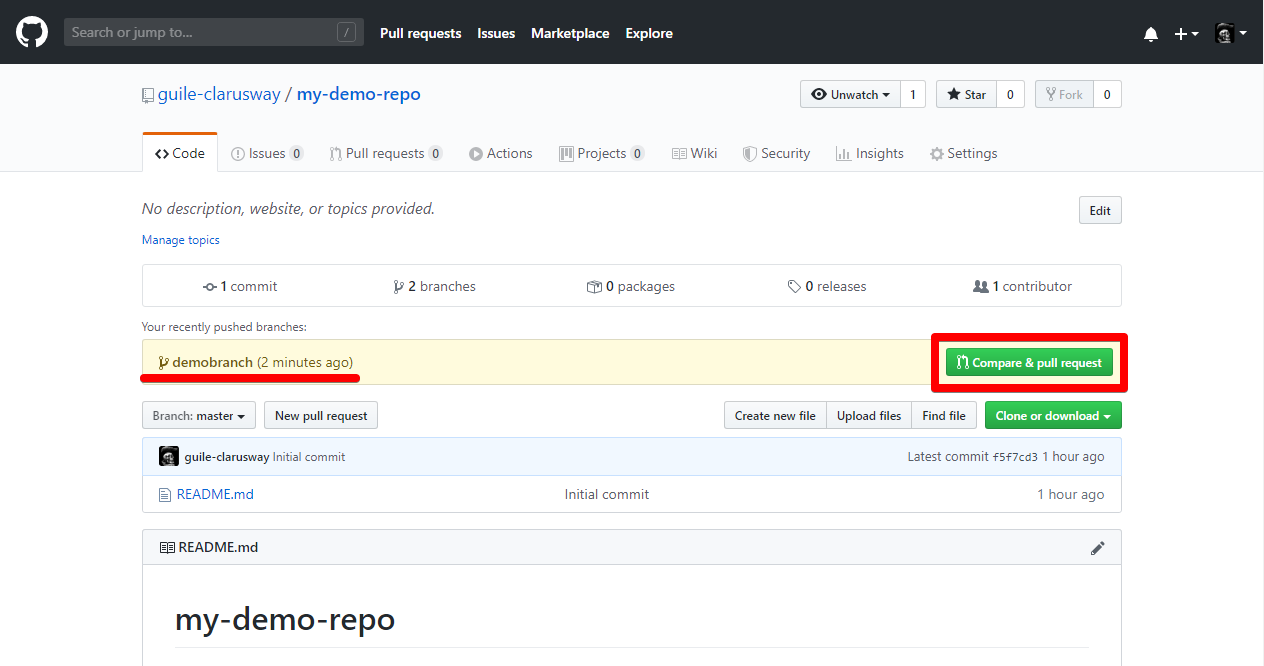
Enter your [Github](https://lms.clarusway.com/mod/lesson/view.php?id=2229) username and password if asked.

If you remember the first [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229) Lesson, we used git remote add origin command previous to git push command. Here we don't need this command because we cloned our repo from remote and both the remote and the local repo are already associated with each other.

That is what you are supposed to see after pushing.



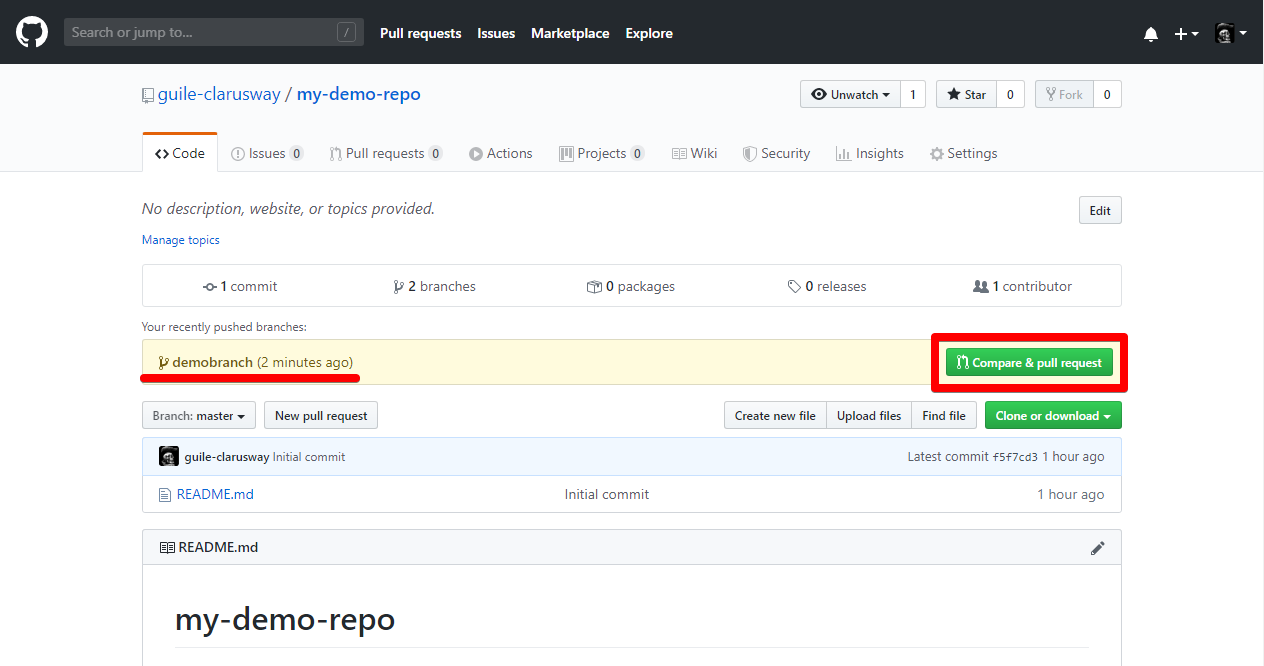
Now let's go and check our remote repo. If you have successfully pushed your branch, you should see the **demobranch**. Remember we had only one branch when we first created our remote repo. Now we have two.



**Creating Pull Request (PR) on GitHub**

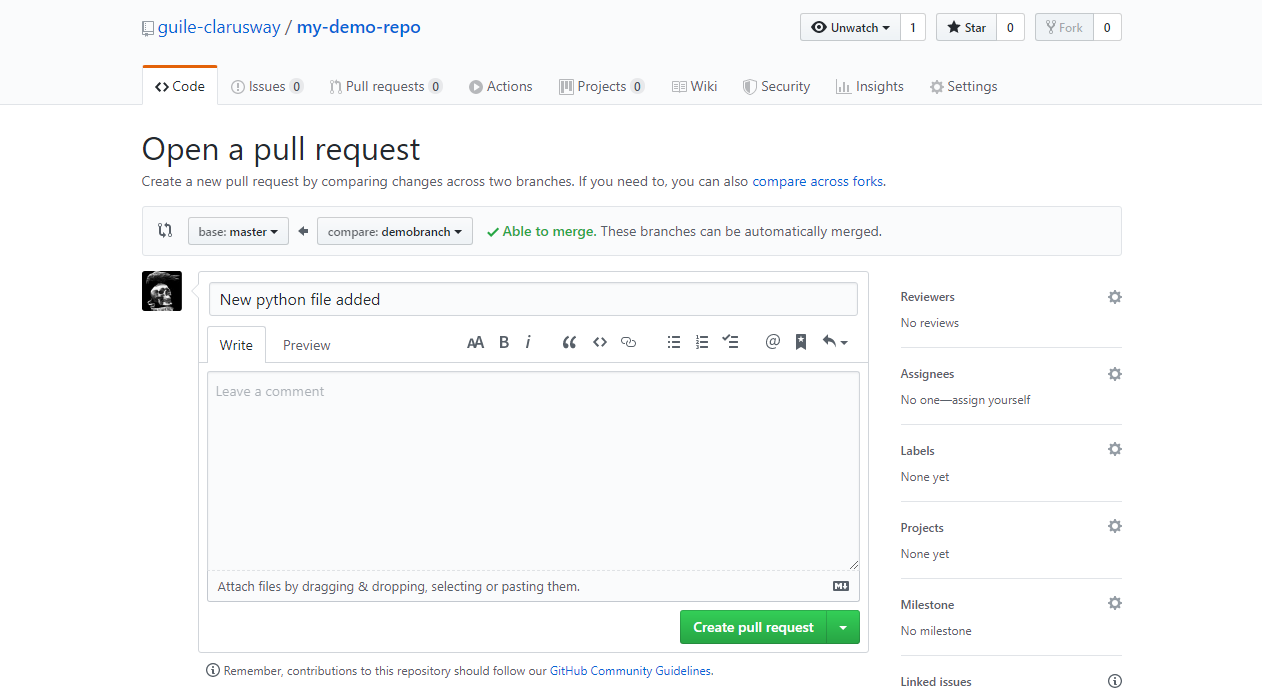
Pull Request (PR) is a method that you tell other contributors or team members that you had some changes (fixing bugs, adding new features, etc.) on the repo and you want to contribute these to the repo. After those changes have been checked and confirmed by the authorized people it is merged to the repo.

Here we will create a pull request for our own repo. Click **Compare & pull request** to open a pull request.



Notice the **Able to merge** remark. Our branch can be merged. You can write a description if you want, your last commit message will be inserted automatically ("New python file added").

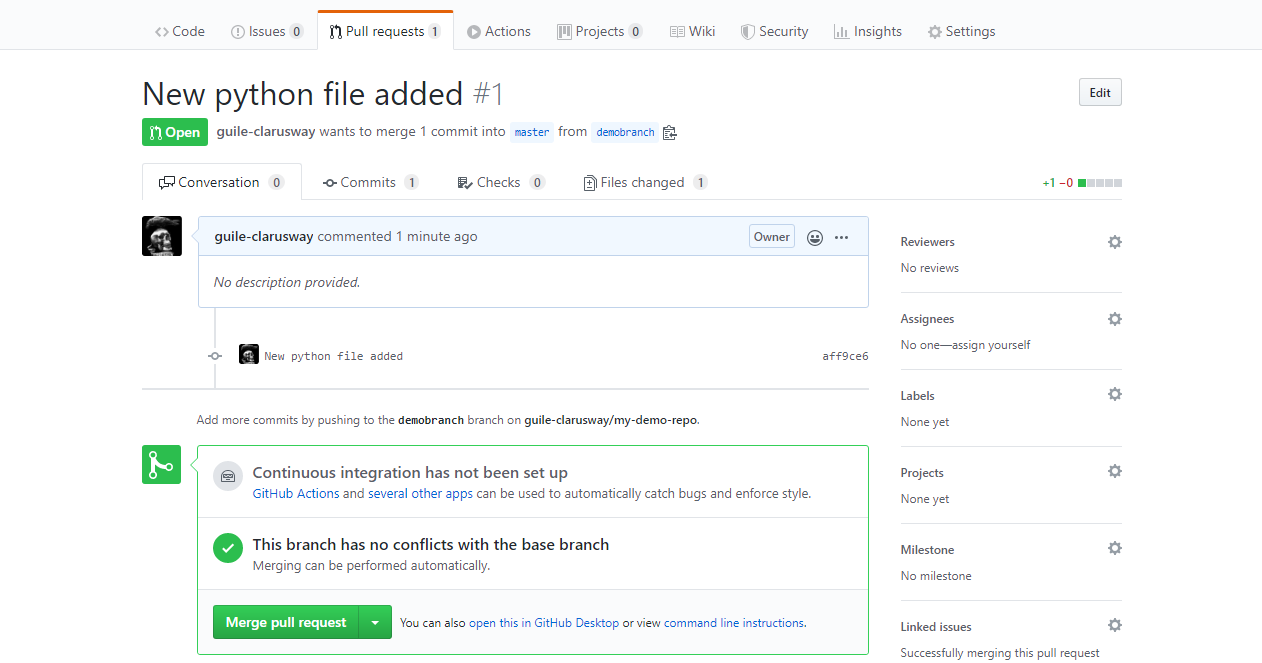
Click **Create pull request**.



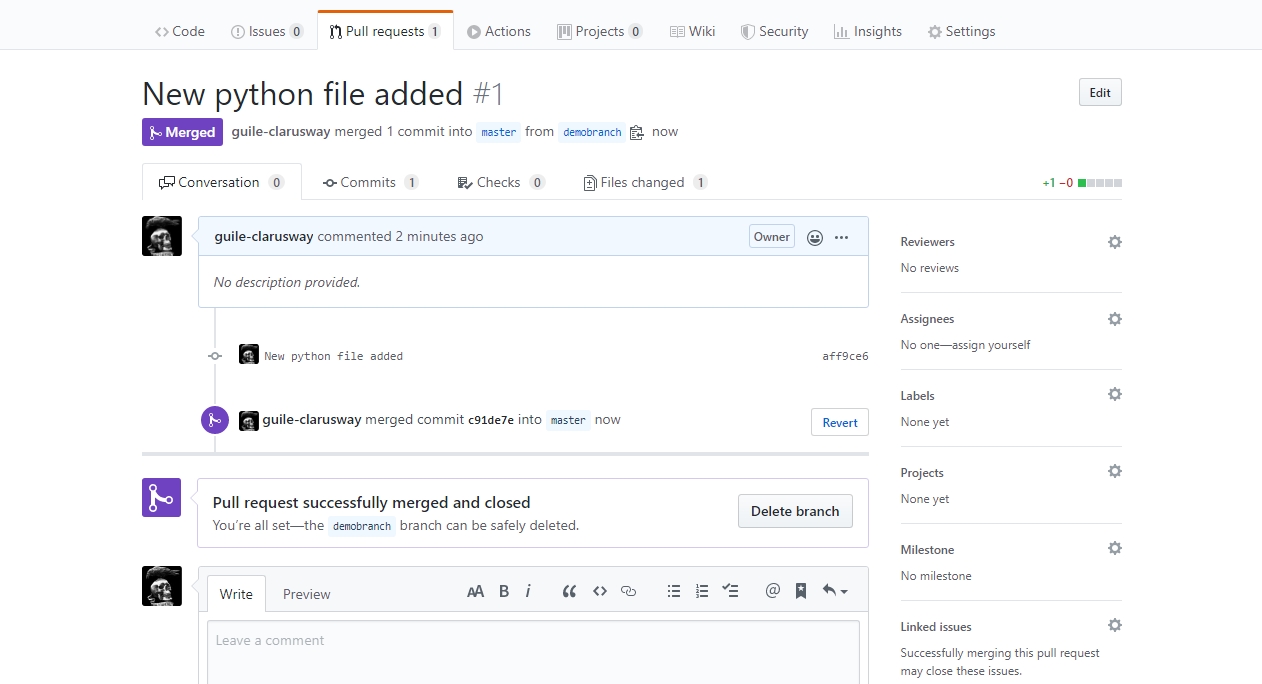
**Merging Pull Request**

Now it's time to merge the pull request. Since we are the only owner and the contributor of this repo, we will merge our own pull request.

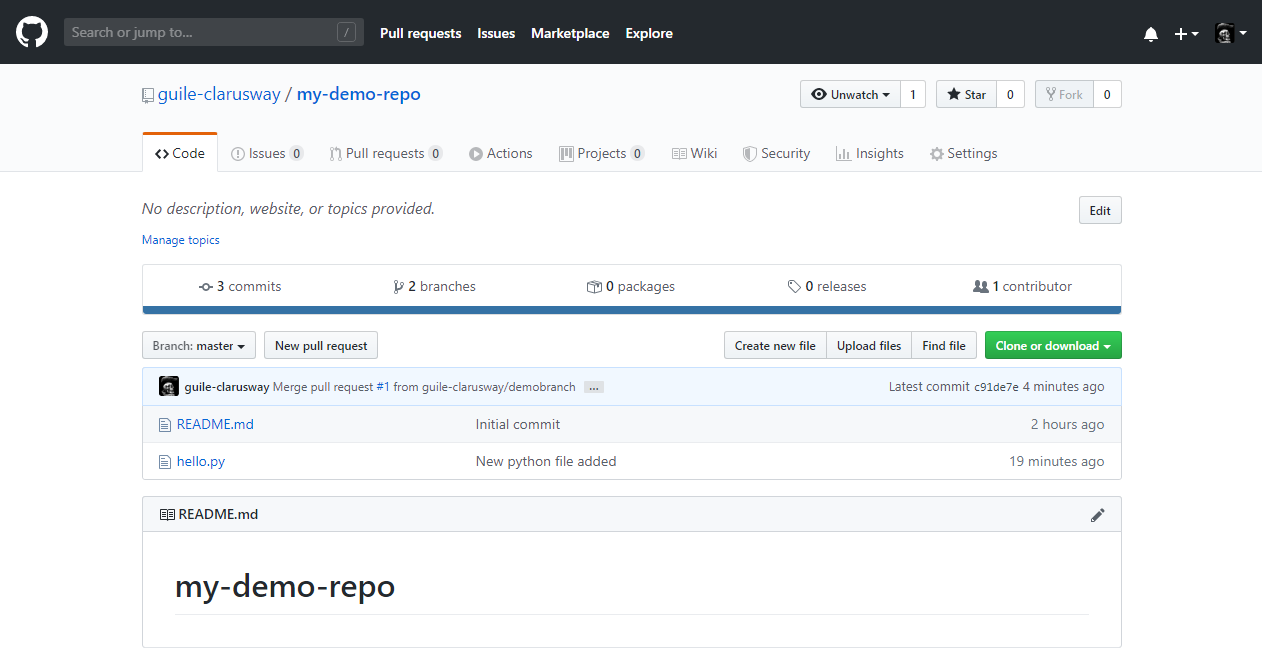
As you see there is no conflict to merge. Click **Merge pull request** and confirm it.



Pull request successfully merged and closed.



Click **Code** tab to see how the repo looks like after merge. As you see **demobranch** is merged to the **master** branch.



**Updating Local Repo**

As you know we have created a new branch on our local repo and pushed it to remote repo. Now, our remote repo ([GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229) repo) is on master branch and up to date, but local repo is not. So, let's go back to GitBash/Terminal and update our local master branch.

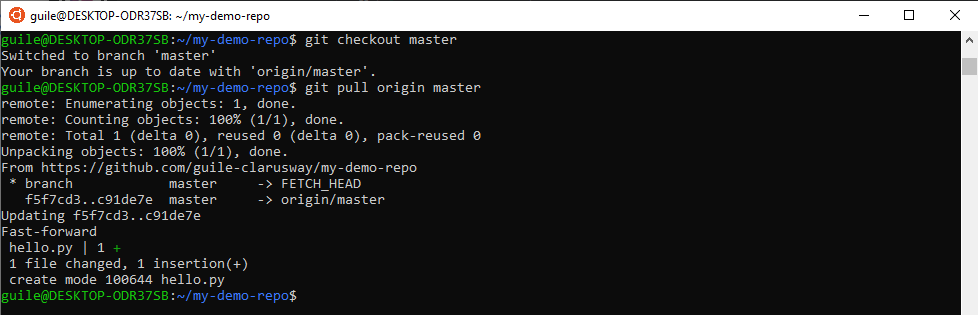
First, switch to master branch.

**$ git checkout master**

Then type

**$ git pull origin master**

to update our local repo.



**Forking Projects**

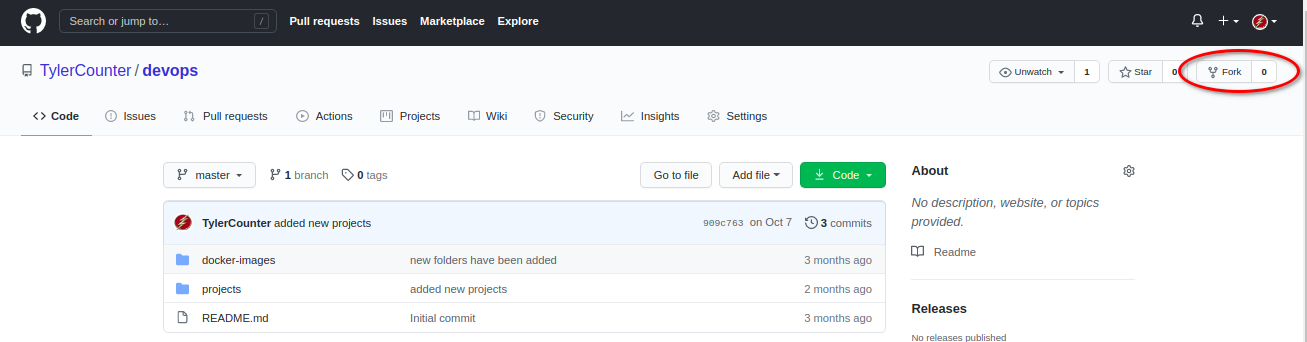
After using [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229) by yourself for a while, you may find yourself wanting to contribute to someone else’s project. Or maybe you’d like to use someone’s project as the starting point for your own. This process is known as **forking**.

Creating a “**fork**” is producing a personal copy of someone else’s project. Forks act as a sort of bridge between the original repository and your personal copy. You can submit Pull Requests to help make other people’s projects better by offering your changes up to the original project. Forking is at the core of social coding at [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229).

**Fork the repository**

First, go to https://[github](https://lms.clarusway.com/mod/lesson/view.php?id=2229).com/TylerCounter/devops

To fork the **devops** repository, click the **Fork** button in the header of the repository.



When it’s finished, you’ll be taken to your copy of the **devops** repository.

**Clone your fork**

You’ve successfully forked the **devops** repository, but so far, it only exists on [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229). To be able to work on the project, you will need to clone it to your computer.

How you clone is up to you.

Some options are cloning with the command line, or by using [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229) Desktop.

**Making and pushing changes**

Go ahead and make a few changes to the project using your favorite text editor. You could, for example, change the text in README.MD to add your [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229) username.

When you’re ready to submit your changes, stage, and commit your changes.

Right now, you’ve essentially told Git, “Okay, I’ve taken a snapshot of my changes!” You can continue to make more changes, and take more commit snapshots. When you’re ready to push your changes up to [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229).com, push your changes to the remote.

**Making a Pull Request**

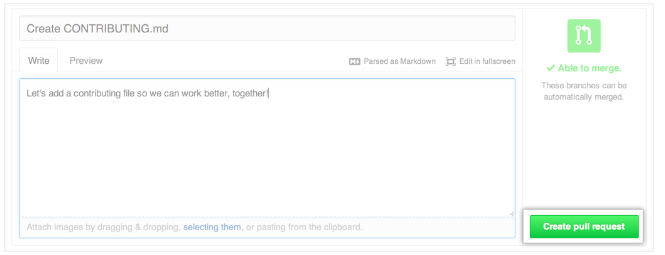
At last, you’re ready to propose changes into the main project! This is the final step in producing a fork of someone else’s project, and arguably the most important. If you’ve made a change that you feel would benefit the community as a whole, you should definitely consider contributing back.

To do so, head on over to the repository on [GitHub](https://lms.clarusway.com/mod/lesson/view.php?id=2229).com where your project lives. For this example, it would be at https://www.[github](https://lms.clarusway.com/mod/lesson/view.php?id=2229).com//devops. You’ll see a banner indicating that you’ve recently pushed a new branch and that you can submit this branch “upstream,” to the original repository:

C:\Users\acer\Downloads\pull-request.png

Clicking on Compare and Pull Request sends you to a discussion page, where you can enter a title and optional description. It’s important to provide as much useful information and a rationale for why you’re making this Pull Request in the first place. The project owner needs to be able to determine whether your change is as useful to everyone as you think it is.

When you’re ready to type out your heartfelt argument, click on Send a pull request.

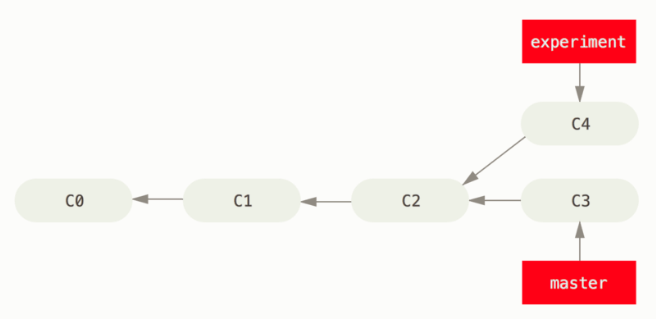


**Pull Requests** are an area for discussion. In this case, Tyler is very busy and probably won’t merge your changes. For other projects, don’t be offended if the project owner rejects your Pull Request, or asks for more information on why it’s been made. It may even be that the project owner chooses not to merge your pull request, and that’s totally okay. Your copy will exist in infamy on the Internet. And who knows–maybe someone you’ve never met will find your changes much more valuable than the original project. Share and share alike!

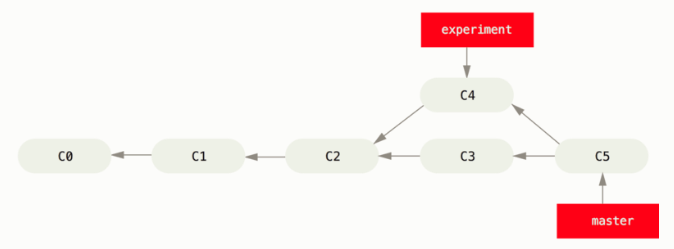
**Rebasing**

In Git, there are two main ways to integrate changes from one branch into another: the **merge** and the **rebase**.

If you go back to an earlier example from Basic Merging, you can see that you diverged your work and made commits on two different branches.



The easiest way to integrate the branches, as we’ve already covered, is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).



However, there is another way: you can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called **rebasing**. With the rebase command, you can take all the changes that were committed on one branch and replay them on a different branch.

For this example, you would check out the experiment branch, and then rebase it onto the master branch as follows:

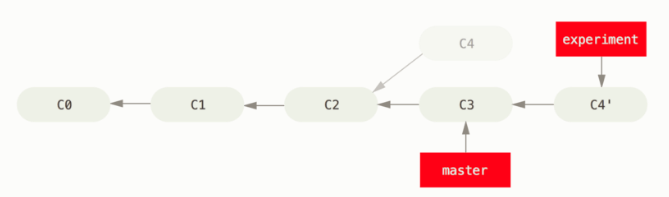
**$ git checkout experiment**

**$ git rebase master**

First, rewinding head to replay your work on top of it...

Applying: added staged command

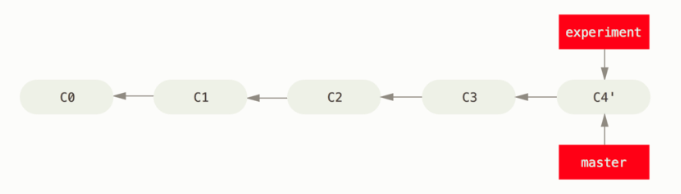
This operation works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.



At this point, you can go back to the master branch and do a fast-forward merge.

**$ git checkout master**

**$ git merge experiment**



Now, the snapshot pointed to by C4' is exactly the same as the one that was pointed to by C5 in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

Often, you’ll do this to make sure your commits apply cleanly on a remote branch — perhaps in a project to which you’re trying to contribute but that you don’t maintain. In this case, you’d do your work in a branch and then rebase your work onto origin/master when you were ready to submit your patches to the main project. That way, the maintainer doesn’t have to do any integration work — just a fast-forward or a clean apply.