

## Portfolio 5

### Version control and Git

A **version control system** (sometimes called revision control) is a tool that lets users track the history and attribution of their project files over time (stored in a repository), and which helps the developers in the team to work together. Modern version control systems help them work simultaneously, in a non-blocking way, by giving each developer his or her own sandbox, preventing their work in progress from conflicting, and all the while providing a mechanism to merge changes and synchronize work.

**Distributed version control** systems such as **Git** give each developer his or her own copy of the project's history, a clone of a repository. This is what makes Git fast:

- 1) Nearly all operations are performed locally, and are flexible.
- 2) Users can set up repositories in many ways.

Repositories meant for developing also provide a separate working area (or a worktree) with project files for each developer. The branching model used by Git enables cheap local branching and flexible branch publishing, allowing to use branches for context switching and for sandboxing different works in progress (making possible, among other things, a very flexible topic branch workflow).

The fact that the whole history is accessible allows for long-term undo, rewinding back to last working version, and so on. Tracking ownership of changes automatically makes it possible to find out who was responsible for any given area of code, and when each change was done. Users can compare different revisions, go back to the revision a user is sending a bug report against, and even automatically find out which revision introduced a regression bug. The fact that Git is tracking changes to the tips of branches with reflog allows for easy undo and recovery.

A unique feature of Git is that it enables explicit access to the staging area for creating commits (new revisions of a project). This brings additional flexibility to managing users working area and deciding on the shape of a future commit.

## Git Branch

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed in Git Basics, the first module of this series. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

The git branch command lets users create, list, rename, and delete branches. It doesn't let users switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

### Usage

`git branch`

List all of the branches in user repository.

`git branch <branch>`

Create a new branch called <branch>. This does not check out the new branch.

`git branch -d <branch>`

Delete the specified branch. This is a "safe" operation in that Git prevents user from deleting the branch if it has unmerged changes.

`git branch -D <branch>`

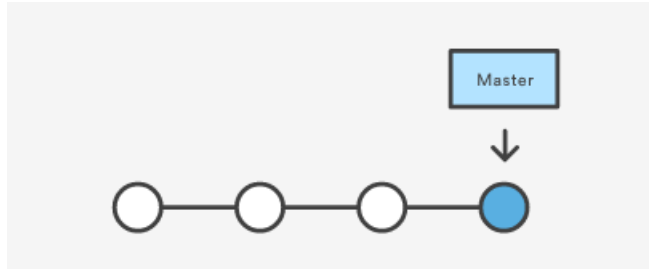
Force delete the specified branch, even if it has unmerged changes. This is the command to use if user want to permanently throw away all of the commits associated with a particular line of development.

`git branch -m <branch>`

Rename the current branch to <branch>.

## Creating Branches

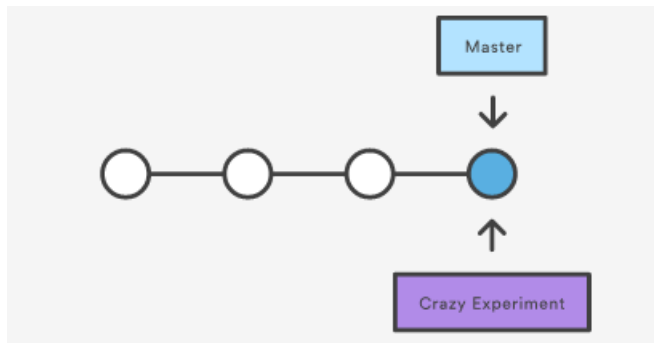
It's important to understand that branches are just pointers to commits. When user create a branch, all Git needs to do is create a new pointer and it doesn't change the repository in any other way. So, if user start with a repository that looks like this:



Then, user create a branch using the following command:

```
git branch crazy-experiment
```

The repository history remains unchanged. All user get is a new pointer to the current commit:



Note that this only creates the new branch. To start adding commits to it, user need to select it with git checkout, and then use the standard git add and git commit commands.

## Git Checkout

The git checkout command lets user navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. Think of it as a way to select which line of development users are working on.

### Usage

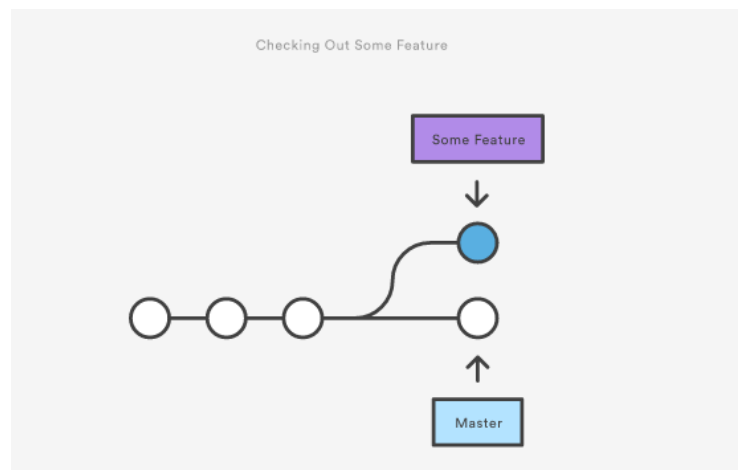
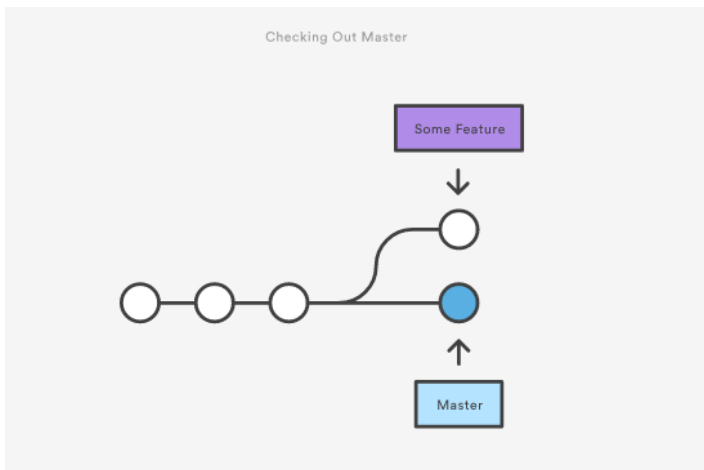
`git checkout <existing-branch>`

Check out the specified branch, which should have already been created with git branch. This makes <existing-branch> the current branch, and updates the working directory to match.

`git checkout -b <new-branch>`

Create and check out <new-branch>. The -b option is a convenience flag that tells Git to run git branch <new-branch> before running git checkout <new-branch>. git checkout -b <new-branch> <existing-branch>

Git checkout works hand-in-hand with git branch. When users want to start a new feature, users need to create a branch with git branch, then check it out with git checkout. They can work on multiple features in a single repository by switching between them with git checkout.



## Git Merge

Merging is Git's way of putting a forked history back together again. The `git merge` command lets user take the independent lines of development created by `git branch` and integrate them into a single branch. Most often, it is necessary when a file is modified by two people on two different computers at the same time. When two branches are merged, the result is a single collection of files that contains both sets of changes.

In some cases, the merge can be performed automatically, because there is sufficient history information to reconstruct the changes, and the changes do not conflict. In other cases, a person must decide exactly what the resulting files should contain. Many revision control software tools include merge capabilities.

Note that all of the commands presented below merge into the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that `git merge` is often used in conjunction with `git checkout` for selecting the current branch and `git branch -d` for deleting the obsolete target branch.

### Usage

`git merge <branch>`

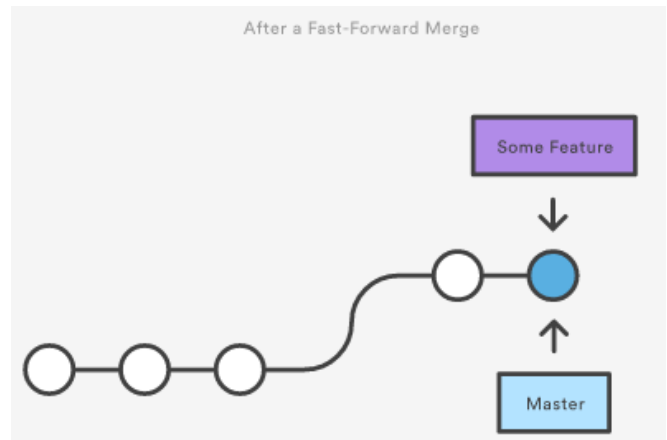
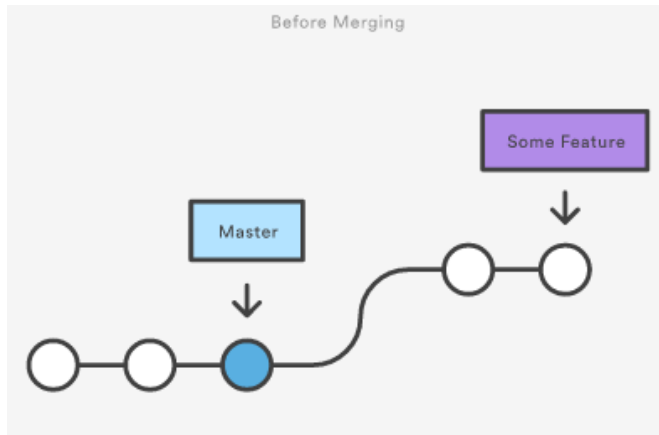
Merge the specified branch into the current branch. Git will determine the merge algorithm automatically.

`git merge --no-ff <branch>`

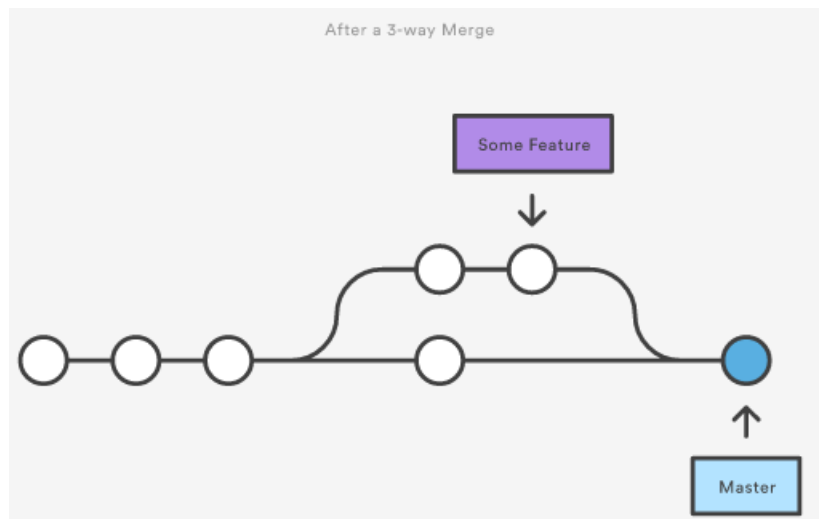
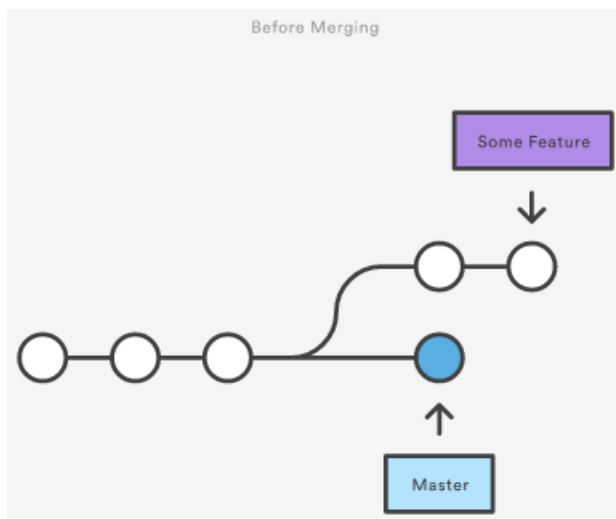
Merge the specified branch into the current branch, but always generate a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in user's repository.

Once users have finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of user's repository, Git has several distinct algorithms to accomplish this: a fast-forward merge or a 3-way merge.

A **fast-forward merge** can occur when there is a linear path from the current branch tip to the target branch. Instead of "actually" merging the branches, all Git has to do to integrate the histories is move (i.e., "fast forward") the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the following:



However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a **3-way merge**. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.



While user can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through rebasing) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

## Resolving Conflicts

If the two branches users are trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that user can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When user encounter a merge conflict, running the git status command shows user which files need to be resolved. For example, if both branches modified the same section of hello.py, user would see something like the following:

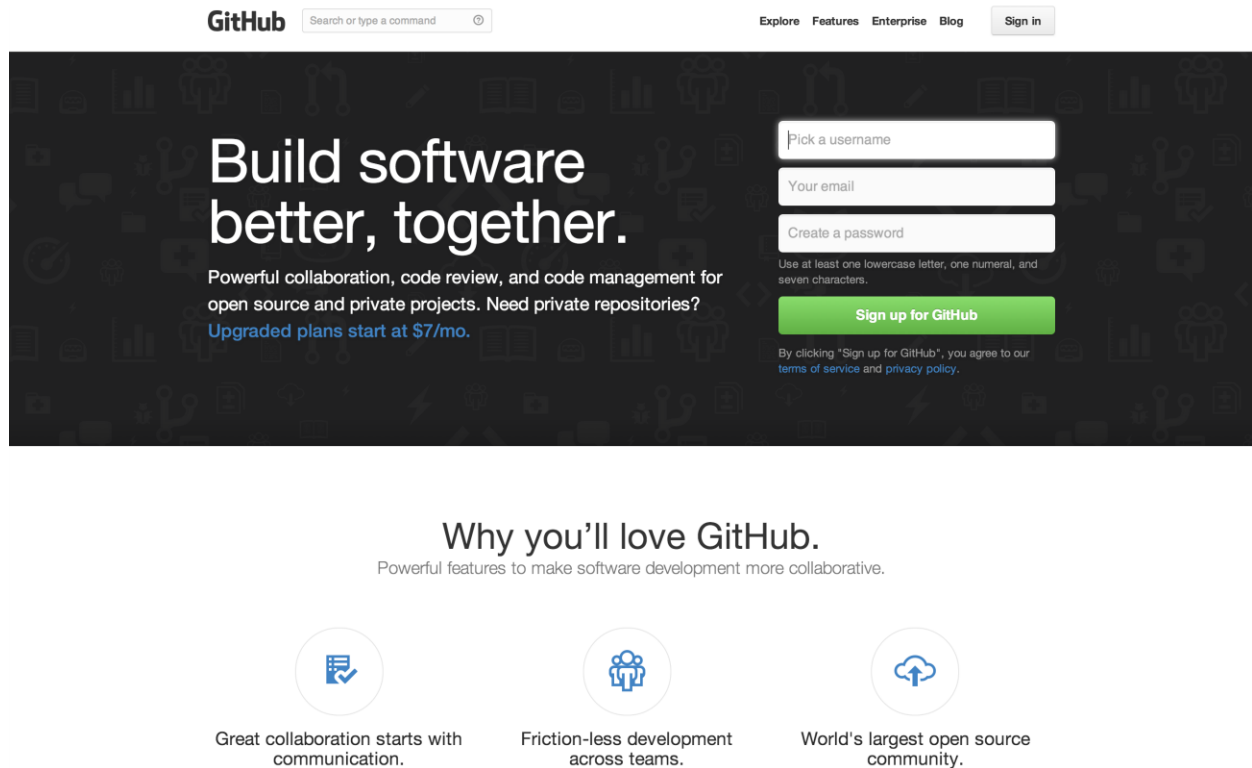
```
# On branch master
# Unmerged paths:
# (use "git add/rm ..." as appropriate to mark resolution)
#
# both modified: hello.py
#
```

Then, users can go in and fix up the merge to they liking. When users are ready to finish the merge, all they have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, users run a normal git commit to generate the merge commit. It's the exact same process as committing an ordinary snapshot, which means it's easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It's not possible to have conflicting changes in a fast-forward merge.

## Demo

### 1. Setting Up GitHub And Git For The First Time



The screenshot shows the GitHub homepage. At the top, there is a navigation bar with the GitHub logo, a search bar, and links for Explore, Features, Enterprise, Blog, and Sign in. The main content area has a dark background with the text "Build software better, together." and a sub-headline "Powerful collaboration, code review, and code management for open source and private projects. Need private repositories? Upgraded plans start at \$7/mo." To the right, there is a sign-up form with three input fields: "Pick a username", "Your email", and "Create a password". Below the password field, there is a note: "Use at least one lowercase letter, one numeral, and seven characters." and a green "Sign up for GitHub" button. At the bottom of the sign-up form, there is a small disclaimer: "By clicking 'Sign up for GitHub', you agree to our terms of service and privacy policy." Below the main content area, there is a section titled "Why you'll love GitHub." with the sub-headline "Powerful features to make software development more collaborative." and three icons representing collaboration, friction-less development, and the open source community.

GitHub

Search or type a command

Explore Features Enterprise Blog Sign in

# Build software better, together.

Powerful collaboration, code review, and code management for open source and private projects. Need private repositories? Upgraded plans start at \$7/mo.

Pick a username

Your email

Create a password

Use at least one lowercase letter, one numeral, and seven characters.

Sign up for GitHub

By clicking "Sign up for GitHub", you agree to our terms of service and privacy policy.

## Why you'll love GitHub.

Powerful features to make software development more collaborative.

- Great collaboration starts with communication.
- Friction-less development across teams.
- World's largest open source community.

Figure 1: GitHub's signup page.

First, user need to sign up for an account on GitHub.com. It's as simple as signing up for any other social network. Then, if users want to work on their project on their local computer, users need to have Git installed. In fact, GitHub won't work on their local computer if they don't install Git. Install Git for Windows, Mac or Linux as needed.



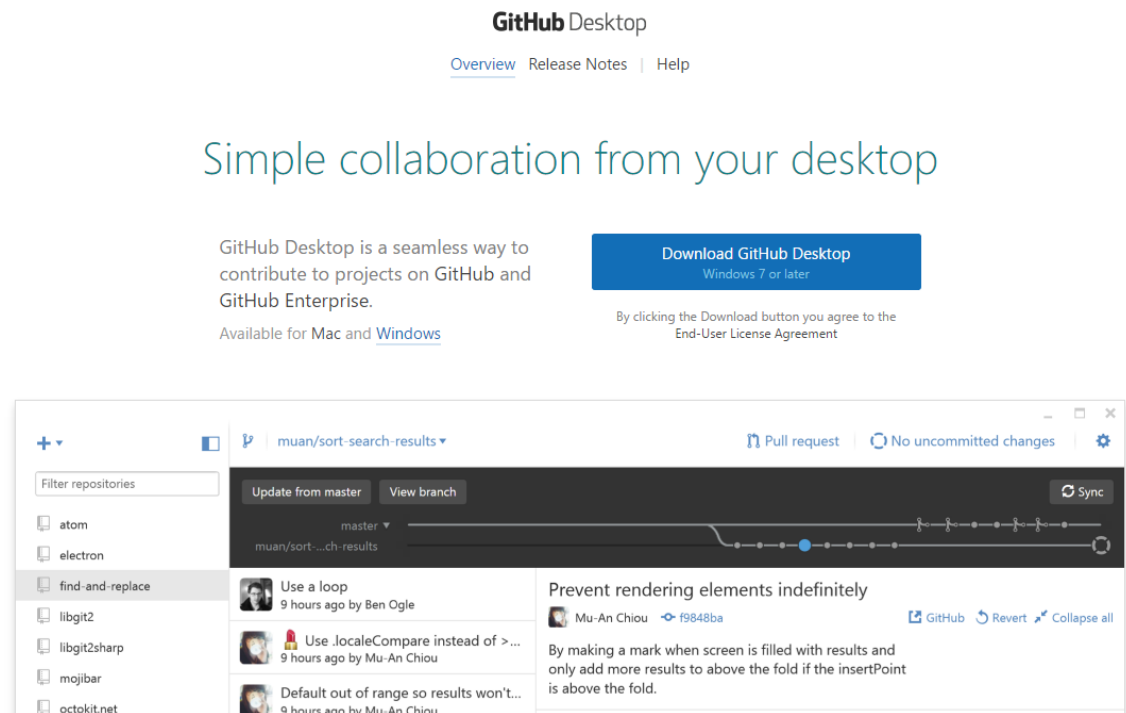


Figure 1.1: GitHub's desktop download page. <https://desktop.github.com/>

## 2. Creating Online Repository

Both Git and GitHub refer to this as a repository, or “repo” for short, a digital directory or storage space where user can access their project, its files, and all the versions of its files that Git saves.

Go back to GitHub.com and click the tiny book icon next to username. Or, go to the new repository page if all the icons look the same. Give the repository a short, memorable name.

Create a new repository

A repository contains all the files for your project, including the revision history.

---

Owner: AnonTester / Repository name: Portfolio\_5 ✓

Great repository names are short and memorable. Need inspiration? How about [probable-sniffle](#).

Description (optional)

---

☒ **Public**  
Anyone can see this repository. You choose who can commit.

☐ **Private**  
You choose who can see and commit to this repository.

---

☐ **Initialize this repository with a README**  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

---

**Create repository**

Figure 1.2: Create new repository page.

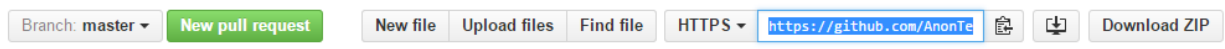
Don't worry about clicking the checkbox next to "Initialize this repository with a README." A Readme file is usually a text file that explains a bit about the project.

Click the green "Create Repository" button and finished. User now have an online space for Their project to live in.

### 3. Cloning a repository

When user create a repository on GitHub, it exists as a remote repository. User can clone their repository to create a local copy on user computer and sync between the two locations.

1. On GitHub, navigate to the main page of the repository
2. Under user repository name, click to copy the clone URL for the repository.



3. Open Git Shell.
4. Change the current working directory to the location where user want the cloned directory to be made.
5. Type **git clone**, and then paste the URL copied in Step 2.
6. Press **Enter**. User local clone will be created.

```
C:\Windows\System32\WindowsPowerShell\v1.0\Powershell.exe
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Muhd Fauzan\Documents\GitHub> git clone https://github.com/AnonTester/Portfolio_5.git
Cloning into 'Portfolio_5'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
C:\Users\Muhd Fauzan\Documents\GitHub>
```

Figure 1.3: Local clone created.

## 4. Add Files to Git

Before adding new files to git repo, user must enter to directory folder by using cd command.

```
C:\Users\Muhd Fauzan\Documents\GitHub> ls

Directory: C:\Users\Muhd Fauzan\Documents\GitHub

Mode                LastWriteTime         Length Name
----                -
d-----          5/10/2016  10:03 PM             Portfolio_5

C:\Users\Muhd Fauzan\Documents\GitHub> cd Portfolio_5
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 ~0 -0 !]> ls

Directory: C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5
```

Figure 1.4: Go to directory Folder.

Now user can add newly created or copied files to their git repository. To add a new files, use git command `git add [filename]`.

Adding Single File:

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 ~0 -0 !]> git add "Hello World.txt"
```

Adding All Files:

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master]> git add -A
```

## 5. Commit File Changes

After successfully adding new files to user local repository. Now commit user changes. This commit user files in local repository only.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 ~0 -0 !]> git add "Hello World.txt"
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 ~0 -0 !]> git commit -m "1st commit"
[master (root-commit) 7cf1eab] 1st commit
1 file changed, 1 insertion(+)
create mode 100644 Hello world.txt
```

## 6. Push User Changes

Now push user changes to remote git repository using command `git push`. This will upload user files on remote repository.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master]> git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 233 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/AnonTester/Portfolio_5.git
 * [new branch]      master -> master
```

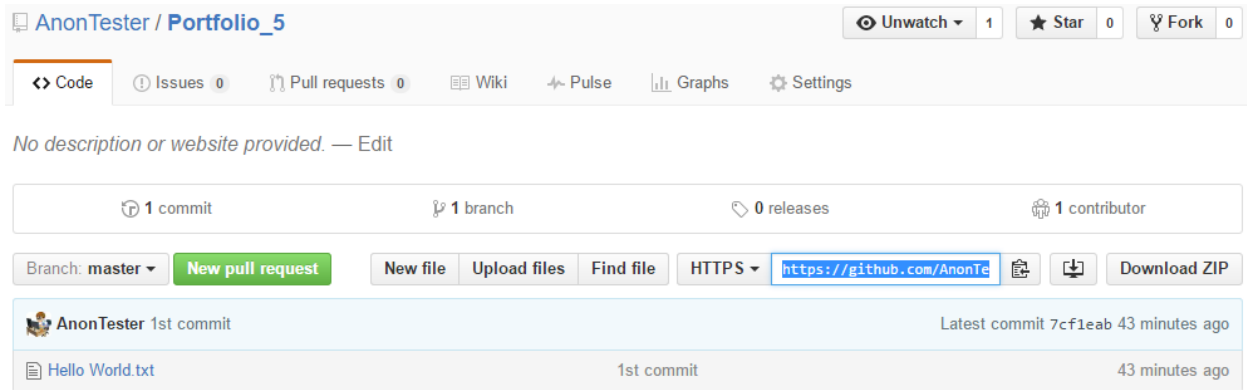


Figure 1.5: New file added in user repository.

## 7. Creating a New Branch

When user first create a git repository (with a command such as `git init`), one branch is created. It's user main (and only) branch. The branch, by default, is named **master**.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 -0 -0 | +0 -0 -1]> git branch
* master
```

Some user wants to keep their master branch clean, by clean I mean without any changes, like that user can create at any time a branch from their master. Each time, that user want to commit a bug or a feature, they need to create a branch for it, which will be a copy of their master branch.

When user do a pull request on a branch, they can continue to work on another branch and make another pull request on this other branch. Before creating a new branch, pull the changes from upstream. Their master needs to be up to date. User can create a branch via **git branch branch-name**.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 -0 -0 | +0 -0 -1]> git branch mybranch
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 -0 -0 | +0 -0 -1]> git branch
* master
  mybranch
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 -0 -0 | +0 -0 -1]>
```

Figure 1.6: Creating new branch named mybranch.

User can now make changes or add new files in the new branch. But before that, user must change their current branch (master) to new branch by using command **git-checkout**.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master +1 -0 -0 | +0 -0 -1]> git checkout mybranch
A      Hello world 2.txt
Switched to branch 'mybranch'
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [mybranch +1 -0 -0 | +1 -0 -0 ]>
```

Figure 1.7: Switching branch from master to mybranch.

Then, user can add files to new branch without affecting the files inside master branch.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [mybranch +1 -0 -0 | +1 -0 -0 ]> git add "Hello world 2.txt"
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [mybranch +1 -0 -0 | +1 -0 -0 ]> git commit -m "Add new Hello world"
[mybranch fabd9fc] Add new Hello world
1 file changed, 1 insertion(+)
create mode 100644 Hello world 2.txt
```

Figure 1.8: Adding file to mybranch.

To apply changes to the online repository simply use **git push --set-upstream origin branch-name**.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [mybranch +1 -0 -0 ]> git push --set-upstream origin mybranch
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 262 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/AnonTester/Portfolio_5.git
 * [new branch]      mybranch -> mybranch
Branch mybranch set up to track remote branch mybranch from origin.
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [mybranch +1 -0 -0 ]>
```

Figure 1.9: Push file changes to mybranch.

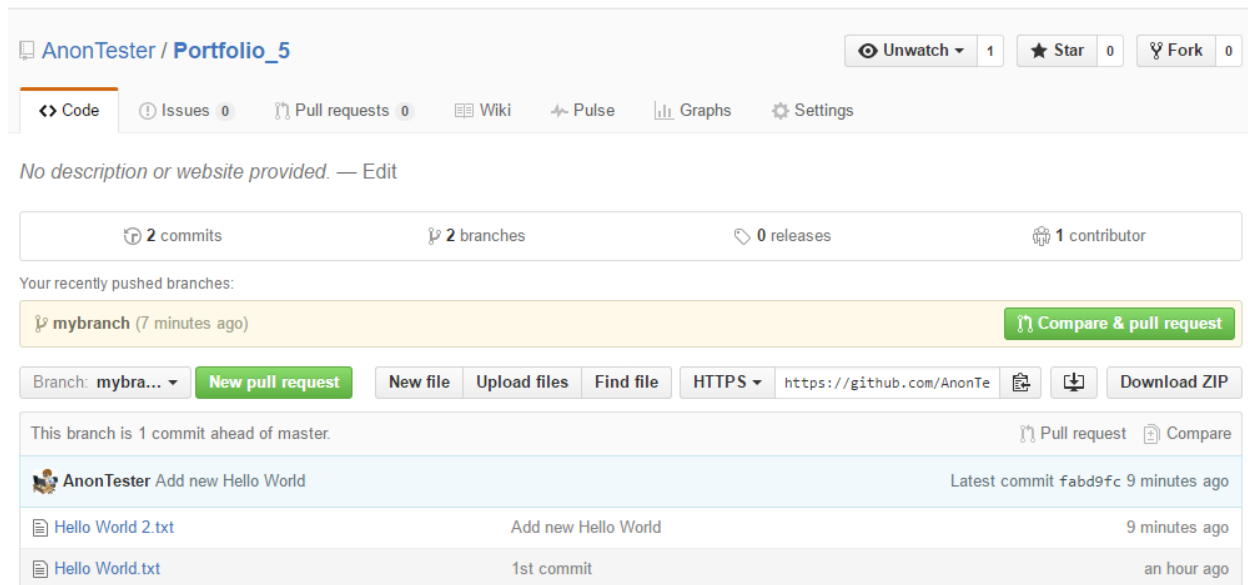


Figure 2.0: List of updated files in user new branch.

## 8. Merge with original branch

To apply the changes made from the new branch to the original branch(master), first git checkout back to the master branch. Then use git merge command **git merge branch-name**. Next, push the changes to master using a command **git push origin master**.

```
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [mybranch]> git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master]> git merge mybranch
Updating 7cf1eab..fabd9fc
Fast-forward
 Hello World 2.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 Hello world 2.txt
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master]> git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/AnonTester/Portfolio_5.git
 7cf1eab..fabd9fc master -> master
C:\Users\Muhd Fauzan\Documents\GitHub\Portfolio_5 [master]>
```

Figure 2.1: Merge new branch with master branch

## Portfolio 5 A3499 Muhd Fauzan

The screenshot shows the GitHub interface for the repository 'AnonTester / Portfolio\_5'. At the top, there are navigation links: 'Code', 'Issues 0', 'Pull requests 0', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. Below these, a message states 'No description or website provided. — Edit'. A summary bar indicates '2 commits', '2 branches', '0 releases', and '1 contributor'. Below this, a toolbar includes a branch selector set to 'master', a 'New pull request' button, and buttons for 'New file', 'Upload files', 'Find file', 'HTTPS', a repository URL, and a 'Download ZIP' button. The commit history table shows three entries: the latest commit 'Add new Hello World' by 'AnonTester' 29 minutes ago, a commit 'Hello World 2.txt' 29 minutes ago, and the '1st commit' 'Hello World.txt' 2 hours ago.

AnonTester / Portfolio\_5

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

No description or website provided. — Edit

2 commits 2 branches 0 releases 1 contributor

Branch: master New pull request New file Upload files Find file HTTPS https://github.com/AnonTe Download ZIP

AnonTester	Add new Hello World	Latest commit fabd9fc 29 minutes ago
Hello World 2.txt	Add new Hello World	29 minutes ago
Hello World.txt	1st commit	2 hours ago

Figure 2.2: Hello World 2 file now inside master branch.