

Understanding Git and Github

This first article will be focused into Git and Github. What are they used for, why should we use them and are they the same? Furthermore we are going to learn the basic commands with examples and graph pictures. In this way we are going to provide knowledge for both sides, theoretically and practically.

1. git

Many people work on the same document for different reasons: school projects, work related documents, checklists for events. Everyone learned for sure the value of saving the document every time you work. This is the best example of explaining Git because Git is simply a more powerful tool to help you manage your work, any type of work actually — not just code.

More precisely Git is a *Distributed Version Control System* (VCS). It is called version because you can move between multiple versions of files on your local machine and distributed because it allows multiple people to work on a project.

Not only does it keep track of the latest version of your file, but *every* version, which can be extremely useful when you need to change back to a previous state.

2. Installing

To be able to use Git, you first need to have it on your machine. You can check if you already installed it before with the command:

```
git remote --version
```

If this command doesn't return anything you need to install it based on your operating system shown as in this [Documentation](#).

3. Basic setup

If you want to check the configuration settings, you can use the command:

git config --list

If you would like, you can go ahead and save your git username and email so that you won't have to enter them again for future git commands.

git config --global user.name "name"
git config --global user.email "email"

4. Initialising git

Now we can start setting up the version control in our project. Go in the directory of the project you want to set up with the standard command “cd” and initialise a git repository like this:

git init

This creates a new subdirectory named .git that contains all of your necessary repository files and it works like a repository skeleton. At this point, nothing in your project is tracked yet.

```
belasinoimeri@U-CM-L112 ~ % mkdir gitworkshop
belasinoimeri@U-CM-L112 ~ % cd gitworkshop
belasinoimeri@U-CM-L112 gitworkshop % git init
Initialized empty Git repository in /Users/belasinoimeri/gitworkshop/.git/
```

git clone

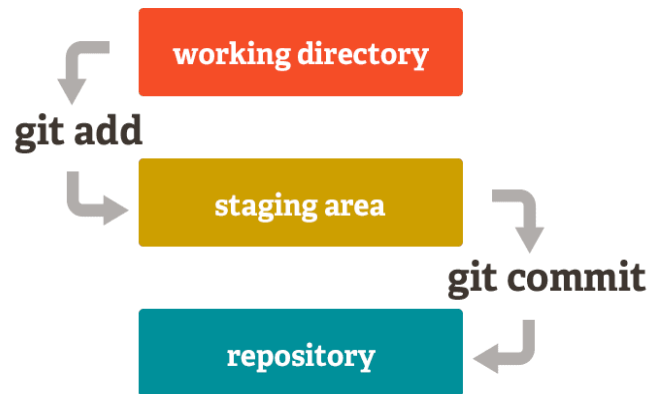
This is used to create a copy of an **existing** repository already backed up in Github.

Git init and Git clone can be easily confused, because at a high level they can both be used to initialize a git repository. Git clone is dependent on git init. Git clone first calls git init to create a new repository and then copies the data from the existing repository.

5. Adding and committing

To start version-controlling your existing files inside the folder you should track them and do an initial commit. You should start by adding the files to git so they can be attached to your git project and then you commit a message, so you can understand what changes you did at this step.

```
git add <file>  
git commit -m 'first commit'
```



1. **Working dir** - Modify files in your working tree.
2. **Staging area** - Selectively stage just those changes you want to be part of your next commit
3. **Repository** - You do a commit, which takes the files as they are in the staging area and stores that **snapshot** permanently to your Git directory.

6. Remote backup

Until now we have our project locally, but we would like to save it remotely and back it up. The project needs to be hosted on a server or computer somewhere. Not every developer and business, wants or can host their own server. For those using Git to manage their project, this is where GitHub repositories come in.

GitHub is a very popular, but not the only, **Git repository hosting service**. For DPS teams these repositories are already created, but in case you want to create your own repository you should head over to [Github](#) like below.

Owner bsinoimeri / Repository name Git/Github ✓

Great repository names are Your new repository will be created as Git-Github about psychic-octo-garbanzo?

Description (optional)
A tutorial about the most used commands in Git and the difference between Git and Github

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

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



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

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: None Add a license: None ⓘ

Create repository

Later on use the link of the repository to add it as the origin of your local git project:

 Set up in Desktop or HTTPS SSH 

git remote add origin https://github.com/bsinoimeri/Git-Github.git

The git remote command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Then you can go ahead and push your code to GitHub and viola you have backed up your code!

git push origin master

```
belasinoimeri@U-CM-L112 gitworkshop % git remote add origin https://github.com/bsinoimeri/Git-Github.git

belasinoimeri@U-CM-L112 gitworkshop % git push origin master

Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 243 bytes | 243.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/bsinoimeri/Git-Github.git
* [new branch]      master -> master
```

7. gitignore

Git sees every file as one of three things:

- tracked - a file which has been previously staged or committed
- untracked - a file which has not been staged or committed
- ignored - a file which Git has been explicitly told to ignore

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source and should not be committed like /node_modules or /packages.

8. Status checking

The main tool you use to determine which files are in which state is the git status command. It allows you to see which of your files have already been committed and which haven't. If all the files have already been committed and pushed, you should see something like this:

git status

```
belasinoimeri@U-CM-L112 gitworkshop % git status
On branch master
nothing to commit, working tree clean
```

If you add a new file to your project, and the file didn't exist before, when you run a git status you should see your untracked file like this:

```
belasinoimeri@U-CM-L112 gitworkshop % git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        BelaGitAnother.txt

nothing added to commit but untracked files present (use "git add" to track)
```

This makes git status really useful for a quick check of what you have backed up already and what you only have locally.

9. Stashing your work

Git stash temporarily stashes changes you've made so you can work on something else, and then come back and re-apply them later on. Stashing

is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

git stash

! The idea is that i changed the file BelaGit.txt but i didn't add it or commit it, so i just stashed it for later use.

```
[belasinoimeri@U-CM-L112 gitworkshop % git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   BelaGit.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        BelaGitAnother.txt

no changes added to commit (use "git add" and/or "git commit -a")
[belasinoimeri@U-CM-L112 gitworkshop % git stash
Saved working directory and index state WIP on master: fbf5e95 First Commit
```

10. Log the commits

Once you've built up a project you have a history of commits, so you can review and revisit any commit in the history. One of the best utilities for reviewing the history of a Git repository is:

git log --oneline

Each commit has a unique **SHA-1 identifying hash**. These IDs are used to travel through the committed timeline and revisit commits. Below you can find some of the most useful commands in logging commits:

git log	<i>Show a list of all commits in a repository. This command shows everything about a commit, such as commit ID, author, date and commit message.</i>
git log -p	<i>List of commits showing only commit messages and changes</i>

<code>git log -S 'something'</code>	<i>List of commits with the particular string you are looking for</i>
<code>git log --author 'Author Name'</code>	<i>List of commits by author</i>
<code>git log --oneline</code>	<i>Show a summary of the list of commits in a repository. This shows a shorter version of the commit ID and the commit message.</i>
<code>git log --since=yesterday</code>	<i>Show a list of commits in a repository since yesterday</i>
<code>git log --grep "term" --author "name"</code>	<i>Shows log by author and searching for specific term inside the commit message</i>

11. Searching

With just about any size codebase, you'll often need to find where a function is called or defined, or display the history of a method. Git provides a couple of useful tools for looking through the code and commits stored in its database quickly and easily. We'll go through a few of them.

<code>git grep 'something'</code>	<i>Searches for parts of strings in a directory</i>
<code>git grep -n 'something'</code>	<i>Searches for parts of strings in a directory and the -n prints out the line numbers where git has found matches</i>
<code>git grep -C<number of lines> 'something'</code>	<i>Searches for parts of string with some context (some line before and some after the 'something' we are looking for)</i>
<code>git grep -B<number of lines> 'something'</code>	<i>Searches for parts of string and also shows lines BEFORE it</i>

<code>git grep -A<number of lines> 'something'</code>	<i>Searches for parts of string and also shows lines AFTER it</i>
---	---

12. Seeing who wrote what

The blame command is a Git feature, designed to help you determine who made changes to a file. Despite its negative-sounding name, git blame is actually pretty innocuous; its primary function is to point out who changed which lines in a file, and why. It can be a useful tool to identify changes in your code.

<code>git blame 'filename'</code>	<i>Show alteration history of a file with the name of the author</i>
<code>git blame 'filename' -l</code>	<i>Show alteration history of a file with the name of the author and the git commit ID</i>

13. Advanced file adding

There are a few more advanced ways of adding files to Git that will make your workflow more efficient. Instead of trying to look for all the files that have changes and adding them one-by-one, we can do the following:

<code>git add filename</code>	<i>Adding files one by one</i>
<code>git add -A</code>	<i>Adding all files in the current directory</i>
<code>git add .</code>	<i>Adding all files changes in the current directory</i>
<code>git add -p</code>	<i>Choosing with 'Y' or 'N' what changes to add</i>

11. Checkout

This command makes your working directory match the exact state of a specific commit or a branch. You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project. Nothing you do in here will be saved in your repository until you checkout at master again.

*! Do not forget, you can checkout **commits** and **branches***

```
belasinoimeri@U-CM-L112 gitworkshop % git log --oneline
fbf5e95 (HEAD -> master, origin/master) First Commit
belasinoimeri@U-CM-L112 gitworkshop % git checkout fbf5e95
Note: checking out 'fbf5e95'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at fbf5e95 First Commit
```

git checkout fbf5e95

For example, above I am working in the fbf5e95 commit.

12. Undo-ing stuff in Git

Mistakes happen and they happen quite frequently with coding! The important thing is that we're able to fix them. Have no fear here! Git has everything you need in case you make a mistake with the code you push, overwrote something, or just want to make a correction to something you pushed.

The git rm command can be used to remove individual files or a collection of files. It can be thought of as the inverse of the git add command. The *-f* or *--force* option is used to override the safety check that Git makes to ensure the files.

```
belasinoimeri@U-CM-L112 gitworkshop % git rm -r BelaGitAnother.txt --force
rm 'BelaGitAnother.txt'
```

The git reset command helps in backtracking specific commits, so you can go back in time to the commit you want to change. Below you can find some useful commands about reset:

<code>git reset HEAD</code>	<i>Switch to the version of the code of the most recent commit</i>
<code>git reset HEAD -- filename</code>	<i>Switch only for a specific file</i>
<code>git reset HEAD^</code>	<i>Switch to the version of the code before the most recent commit</i>
<code>git reset HEAD^ -- filename</code>	<i>Switch only for a specific file</i>
<code>git reset HEAD~3 -- filename</code>	<i>Switch back 3 commits</i>
<code>git reset 0766c053 -- filename</code>	<i>Switch back to a specific commit, where '0766c053' is the commit ID</i>
<code>git reset --hard 0766c053</code>	<i>The previous commands were what's known as "soft" resets. Your code is reset, but git will still keep a copy of the other code in case you need it. On the other hand, the --hard flag tells Git to overwrite all changes in the working directory.</i>

Other commands that can be helpful in undoing commits are: [git clean](#), and [git revert](#).

13. Rewriting history

The `git commit --amend` command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot.



Initial History

Amended History

Amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit, which is visualized with an asterisk (*) in the diagram above.

13. Using Branches

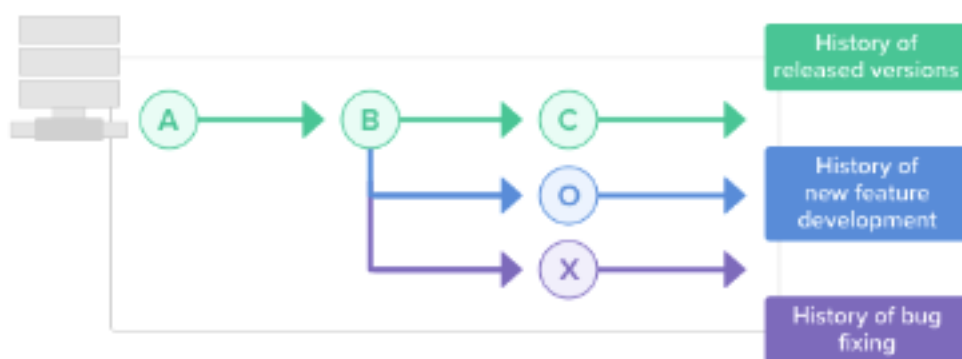
In a collaborative environment, it is common for several developers to share and work on the same source code. For example, some developers will be fixing bugs and others will be implementing new features.

Even when you are working alone you may want to back up some code that you are currently working on, but isn't entirely stable. Maybe you're adding a new feature, you're experimenting and breaking the code a lot, but you still want to keep a back up to save your progress!

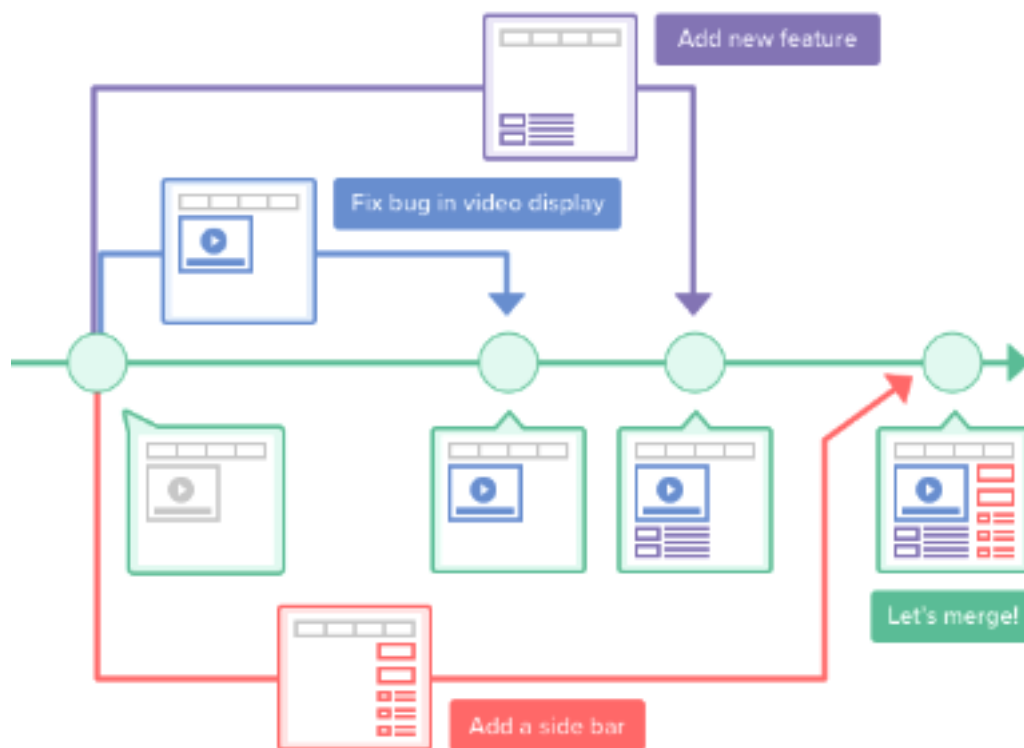
What is a Git branch?

A Git branch is essentially an independent line of development. Branching allows you to work on a separate copy of your code without affecting the master branch and isolate your work from others. The **master branch** of your GitHub repository should always contain working and stable code.

When you first create a branch, a complete clone of your master branch is created under a new name. You can then modify the code in this new branch independently. Once your new feature has been fully integrated and the code is stable, you merge it into the master branch!



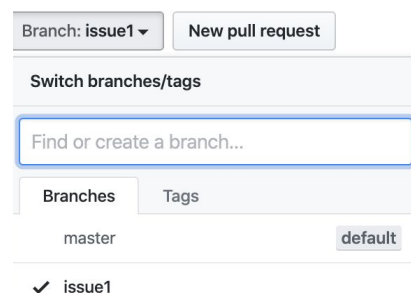
Different branches can be merged into any one branch as long as they belong to the same repository. The diagram below illustrates how development can take place in parallel using branches.



It is a common practice to create a new branch for each task (a branch for bug fixing, a branch for new features, etc.). This method allows others to easily identify what changes to expect and also makes backtracking simple.

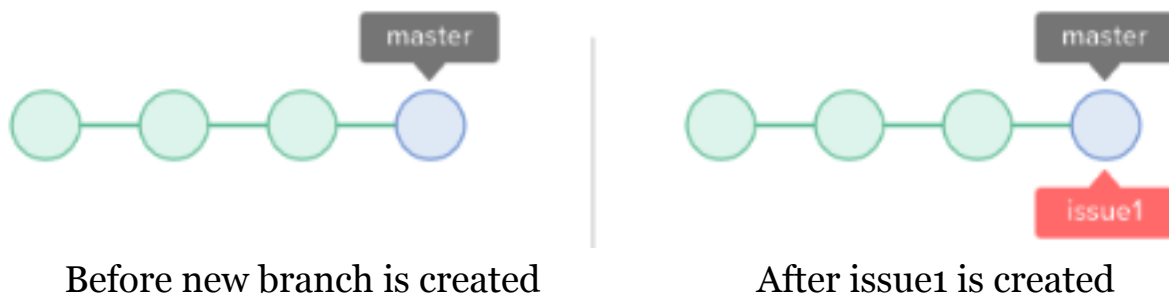
Let's create a branch called “testingbranch” using the command below. It will create a new branch except master like in the picture.

git branch issue1



```
belasinoimeri@U-CM-L112 gitworkshop % git branch issue1
belasinoimeri@U-CM-L112 gitworkshop %
```

The illustration below provides a theoretical visualisation on what happens when the branch is created. The repository is the same, but a new pointer is added to the current commit.



Useful commands for branches

Here's all of the things you need to create and work on a branch:

git checkout -b branchname	<i>Create a local branch to work on</i>
git checkout branch_1 git checkout branch_2	<i>Switching between 2 branches</i>
git push -u origin branch_2	<i>Pushing local branch to remote as backup</i>
git branch -d branch_2	<i>Deleting a local branch - this won't let you delete a branch that hasn't been merged yet</i>
git branch -D branch_2	<i>Deleting a local branch - this WILL delete a branch even if it hasn't been merged yet!</i>
git branch -a	<i>Viewing all current branches for the repository, including both local and remote branches. Great to see if you already have a branch for a particular feature.</i>

git branch -a --merged	<i>Viewing all branches that have been merged into your current branch, including local and remote. Great for seeing where all your code has come from!</i>
git branch -a --no-merged	<i>Viewing all branches that haven't been merged into your current branch, including local and remote</i>
git branch	<i>Viewing all local branches</i>
git branch -r	<i>Viewing all remote branches</i>
git rebase origin/master	<i>Rebase master branch into local branch</i>
git push origin +branchname	<i>Pushing local branch after rebasing master into local branch</i>

In the terminal picture I am showing you how to delete a branch named testingbranch, just to give you an example how the commands above work:

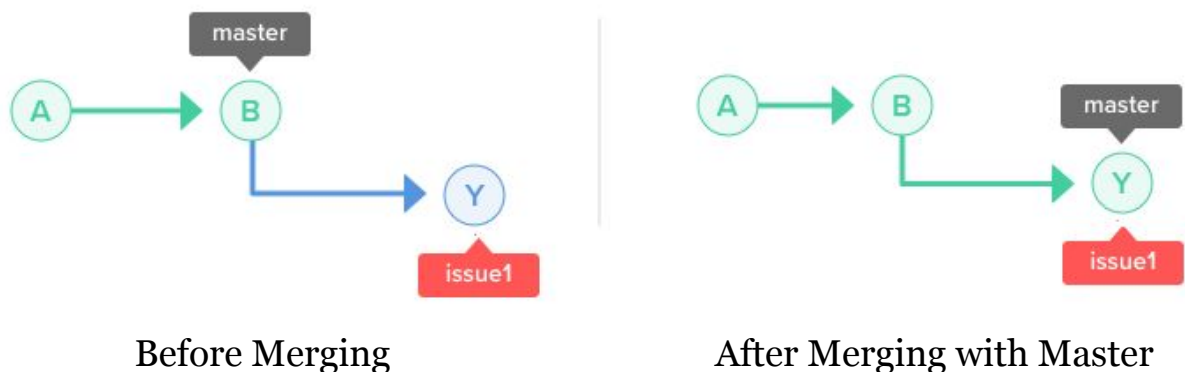
```
belasinoimeri@U-CM-L112 gitworkshop % git branch
issue1
* master
testingbranch
belasinoimeri@U-CM-L112 gitworkshop % git branch -D testingbranch
Deleted branch testingbranch (was fbf5e95).
belasinoimeri@U-CM-L112 gitworkshop % git branch
issue1
* master
```

Merging

Now you are capable of creating a branch and working on it. After you are done with creating the new feature into your branch, you can merge it in order to integrate it to another branch. In the main scenario you will want to merge it back into master branch, so that master has all the latest code features.

Case 1:

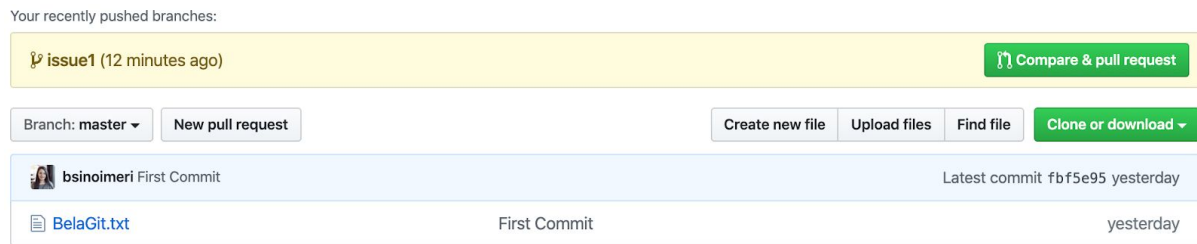
In this case, merging "issue1" back into "master" is not much of an issue. That's because the state of "master" has not changed since "issue1" was created. Git will merge this by moving the "master" position to the latest position of "issue1". This merge is called a "fast-forward".



Below I just changed some code into the branch issue1 and pushed it again because everything was up to date with master till now:

```
belasinoimeri@U-CM-L112 gitworkshop % git push origin issue1
Everything up-to-date
belasinoimeri@U-CM-L112 gitworkshop % git checkout issue1
Switched to branch 'issue1'
belasinoimeri@U-CM-L112 gitworkshop % code .
belasinoimeri@U-CM-L112 gitworkshop % git add .
belasinoimeri@U-CM-L112 gitworkshop % git commit -m "Changed for Branch"
[issue1 bbb37ff] Changed for Branch
1 file changed, 1 insertion(+), 1 deletion(-)
belasinoimeri@U-CM-L112 gitworkshop % git push origin issue1
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 288 bytes | 288.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/bsinoimeri/Git-Github.git
fbf5e95..bbb37ff issue1 -> issue1
```

This is how the Github repository looks like after these commands. You can compare and pull requests directly from Github for the next merging step, but I would **highly** recommend using the terminal.



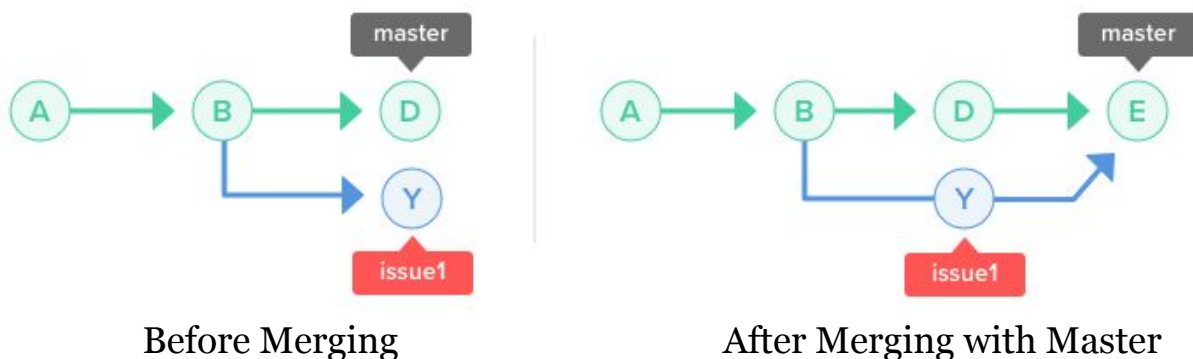
Now in order to merge it from the terminal you have first to make sure you are looking at the master branch and then use the merge command:

git checkout master
git merge issue1
git push

```
belasinoimeri@U-CM-L112 gitworkshop % git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
belasinoimeri@U-CM-L112 gitworkshop % git merge issue1
Updating fbf5e95..bbb37ff
Fast-forward
 BelaGit.txt | 2 + -
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Case 2:

In this case, "master" has been updated several times since "issue1" was branched out. The changes from "issue1" and "master" need to be combined when a merge is executed on these two branches. For this sort of merge, a **"merge commit"** will be created and the "master" position will be updated to the newly created merge commit.



A non fast-forward merge leaves the "issue1" branch as it is. This leaves you with a clearer picture of the branch, so you can easily find where the branch starts or ends and also track the changes that are made to the feature inside the branch. The only bad side is that you have to be very careful if you have **conflicts**.

14. Best practices

These were the most basic Git commands you can use in your daily work when you are collaborating in a project. Do not be afraid of using the terminal, try new commands and google if something goes wrong.

- Commit often locally
- Publish (git push) once
- Write meaningful commit messages
- Never **force push** in production
- **Rollback forward**

15. Homework

- Understand what's github fork, what's the difference between fork and clone and how to sync with a local repository.
- Understand git rebase
- Complete the [interactive tutorial](#) to learn about git's branching model.
- Check these useful links: [Git Simulation](#), [Visual Git Guide](#)

! Last but not least, never confuse **Git** with **Github**.

