Contents

1 Linux File Systems **3**
1.1 Other Things 4

2 Terminology **5**

3 Git **7**
3.1 How Git Works 7
3.1.1 Commits 8
3.1.2 Commit Phases 9
3.2 Creating a Repo 9
3.3 Knowing the Tracked and Untracked 10
3.4 Staging Area Commands 10
3.4.1 Benifits of Staging Area 10
3.5 Commits 11
3.5.1 Seeing Commits 12
3.6 Branches 13
3.7 Github 16
3.7.1 Pushing 16
3.7.2 Collaborating on Github 19
3.7.3 Forking 21
3.8 Summary Table 23

4 Tikz **31**
4.1 Basic Plot 31
4.2 Drawing a basic flow chart 32

Chapter 1

Linux File Systems

Source of the chapter: https://www.youtube.com/watch?v=HIXzJ3Rz9po

To do later: basic bash command (https://www.youtube.com/watch?v=oxuRxtrO2Ag).

In this chapter, we will look on how Linux file systems is composed. A diagram shown in Figure 1.1 is illustrating the hierarchy in Linux.

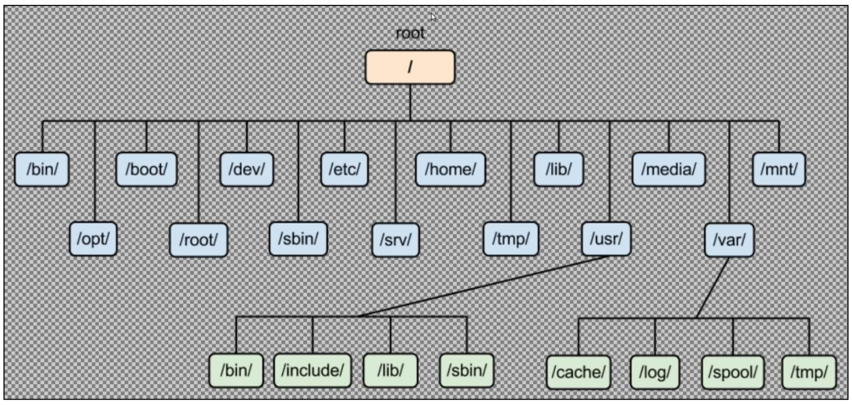


Figure 1.1: Linux File Systems

- In Linux, there is no partition of our physical drives into letters as in Windows (E,C,$\cdots$).

  All the drives and the hardware attached to our computer (USB, mouse,$\cdots$) will be represented in some folder or a file.

- In Linux, all the things are represented by files or folder

- Usually Linux begin with a root directory, represented by \.

  – We should not confuse between the root in the file systems hierarchy and the root user, which has the control over all the system. They are not the same thing.

- Some Functions of the folder:

  – bin: for binary, this where most of the programs live in

- boot: all the necessary file needed by the kernel in order to boot the machine
- cdrom: this is a legacy folder for the old days where do we find our inserted CD. Nowadays, all our dvd and so are in the media folder.
- dev: for device, mouse, keyboard, $\cdots$.
- media: this is an important folder, because if we insert a USB or put an external drive, it will show on the media file
- etc: hold configuration files. Sometimes we need to go to this folder, to change something
- home: all the user folder are in this directory. When we open an terminal, we are automatically in the home.
- opt: for optional,where optional software would be, like Google chrome browser for example
- sbin: most of the command lines are here.

1.1 Other Things

Linux file naming is case sensitive, unlike windows.

If we want to see the hidden folder, we press Ctrl + h

Hidden files, a dot is placed in front of their name, so if we want to hide a file, we type .file name.

Chapter 2

Terminology

- Terminal
  - Short for terminal emulator
  - The window which float on our screen, and use it to type the commands
- Bash: it is a ***language*** which run in a terminal
  - We could have other languages too
  - To know what language we are running: `echo $0`
- The original language was named shell, and bash and zsh are descender that add some new things to shell.
- Command Line: the line we are typing in
- When we open a terminal, we get a ***prompt*** at the beginning of the ***command line***

Chapter 3

Git

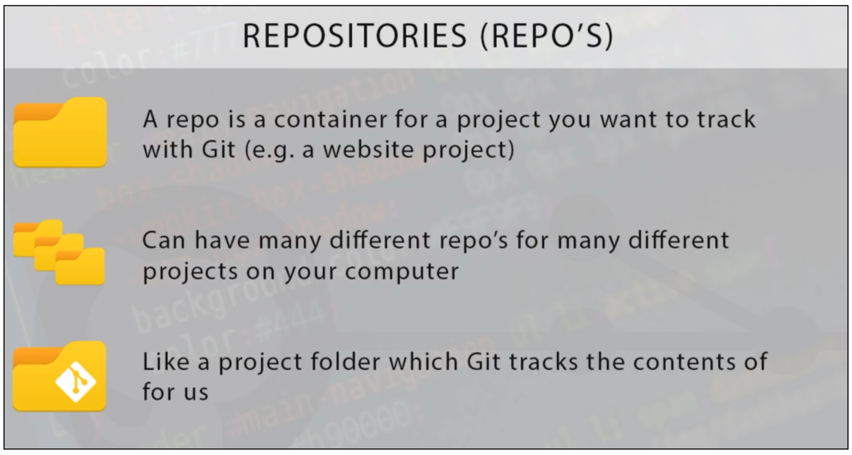
3.1 My Github Account

- User Name: Ranim-94
- Website: https://github.com/Ranim-94

Source of this chapter: https://www.youtube.com/watch?v=MFtsLRphqDM&list=PL4cUxeGkcC9goXbg
4TBzOOOocPR&index=2

3.2 How Git Works

The main idea about how Git work are presented in Figure 3.1.


REPOSITORIES (REPO'S)

A repo is a container for a project you want to track with Git (e.g. a website project)

Can have many different repo's for many different projects on your computer

Like a project folder which Git tracks the contents of for us

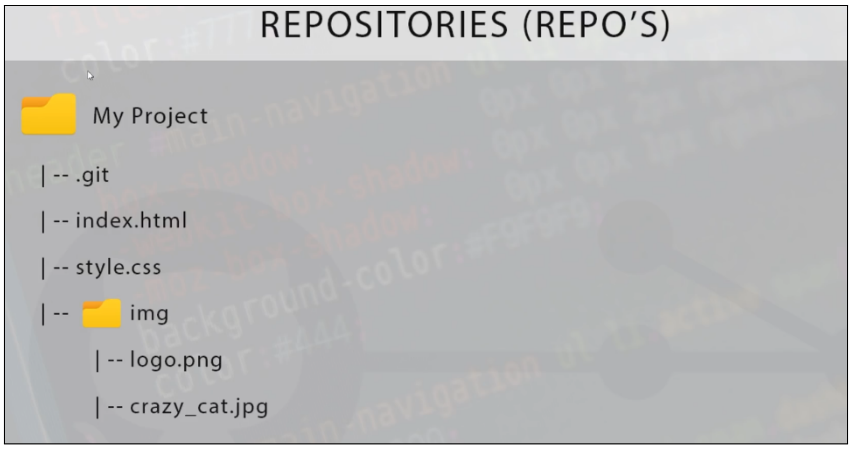
Figure 3.1: Main Components

- Repo can be:
  - Local: so for example on our computer
  - Remote: like some online service that has some servers, and our project will be stored on these severs.

    An example about remote repo is GitHub.

Remote repo will be discussed later.

Let's look an example about some local repo, shown in Figure 3.2.


REPOSITORIES (REPO'S)

```
My Project
|-- .git
|-- index.html
|-- style.css
|-- img
    |-- logo.png
    |-- crazy_cat.jpg
```

Figure 3.2: Example about some local repo

For example we have decided to named our project `y Project`. So the folder `y Project` will be our local repository.

- When we start using git, a `.git` file will be launched.
- Since this `.git` file is the root in our project, it will track all the changes of the files and even file in subfolders.
- If we have put `.git` file in the `img` folder, we can only track the changes in `img` folder.

3.2.1 Commits

Commits are like safe points we can create. It like safe point in games, so if we loose we can back to this safe point and don't repeat from zero.

An example about commit is showed in, where it is showed 5 commits points created for some web page design project.

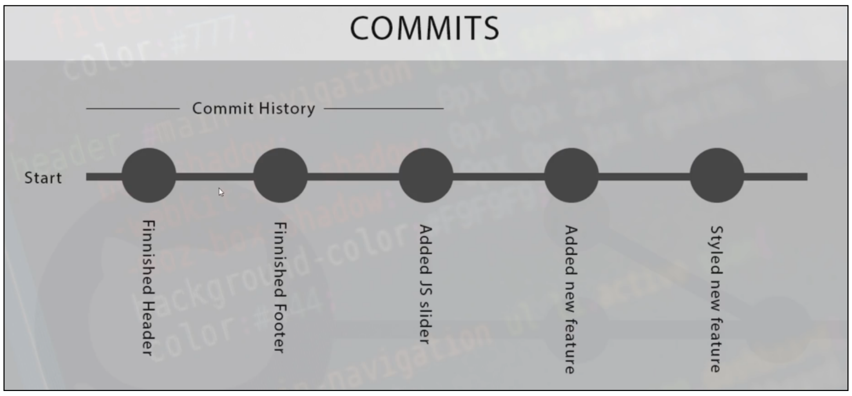


Figure 3.3: Commits Example for Some Web Application

For example, we are working in the project and we have finished the header, so we decide to commit because it some logical point we can get back to it, and we don't loose it.

3.2.2 Commit Phases

Now before we create some repository, there are usually 3 stages before we commit our steps. The phases are showed in Figure 3.4.

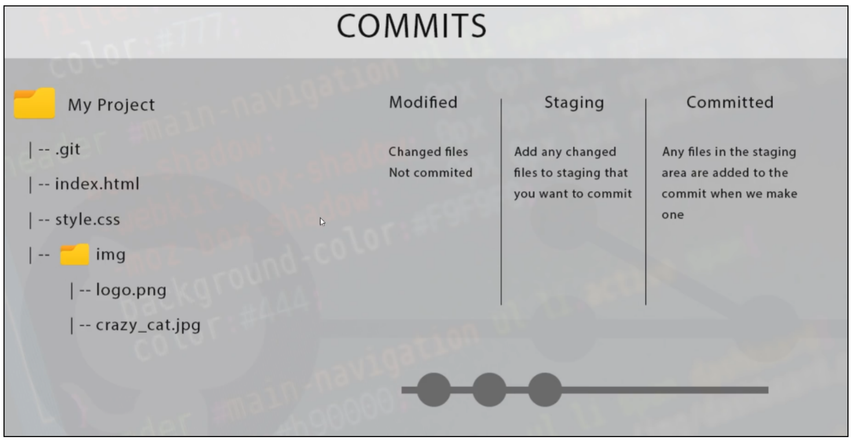

COMMITS
My Project
| -- .git
| -- index.html
| -- style.css
| -- img
| -- logo.png
| -- crazy_cat.jpg
Modified
Changed files Not commited
Staging
Add any changed files to staging that you want to commit
Committed
Any files in the staging area are added to the commit when we make one

Figure 3.4: Commits Phases

Suppose we are working in the project, and we keep changing the `index.html` and `style.css` files. We keep changing till we are fixed at a certain choice. What will happen is:

- As long we are changing, we are in phase called ***modified***, the files are changed, but they are not committed (saved in some check point in time) because we are not sure about our choice yet.
- Once we are sure, we stage these files: staging is like saying that these particular changing will be used for committing
- Finally, we commit the files in the staging phase, and a ***snap shot*** will be created for the files in the staging phase, which are in our case `index.html` and `style.css`.

The stick showed in Figure 3.4 is called ***branch***. And here in this case it is called ***master branch***, because it represents the main code we are working on.

In future, we can for example extend some new branch from the master branch, work in our code, then merge back to the master branch. We do this for example if we want to work on some new features of the code without altering the essential code.

3.3 Creating a Repo

To create a repo inside a project folder:

1. navigate to it using `cd` command

2. Type the Command `git init`
    - Once we type this, a .git folder will be created, indicating that the folder now is repo.
    - In Atom editor, it will make the directory and any files turn to green, indicating that it will be tracked by git.

Note:

It is not necessary that the directory should be empty, it could contain some existing files, and we can use the same command `git init` to make the directory a repository.

Creating a File:

Now we will create an `html` file named index, by typing the command: `touch index.html`

3.4 Knowing the Tracked and Untracked

If we want to know the status of our files, we use the command `git status`, and it will give us the untracked files in red.

3.5 Staging Area Commands

Add File to Staging Area: `git add Name_file.extension`

Removing File from staging Area: `git restore --staged Third_text.txt`

Maybe we need to do this because we have staged a file by mistake.

Adding all files inside the project: `git add .`

Remark: between `add` and `.` there is a space.

It help us from adding file by file manually.

Adding all specific types of files inside the project: (lets say `txt`) `Git add *.txt`

3.5.1 Benifits of Staging Area

Staging Area add more security before we commit directly, so it give us a chance to review our work before save it as check point.

Also, it help us to split our work.

Let's say that we are working on a website having 2 books. The logic will say that we need to work on each book separately.

So what we do is we do stage commit for the 1st book, and then make the commit, and then stage the 2nd book, and make a commit for the 2nd book.

We stage the 2 books differently because the logic is that we we want to fix each book (each task) separately, and then make commits for these different tasks.

3.6 Commits

Command: `git commit -m "First change"`.

The message should be a descriptive message.

Each commit will have a certain ***reference id***.

Example: `git commit -m"Adding the commit v1"`

Output:

```
[master 0a8ce60] Adding the commit v1
3files changed, 3 insertions(+), 5 deletions(-)
delete mode 100644 Number_4.txt
create mode 100644 Style.txt
create mode 100644 Third_text.txt
```

It is telling us that the this commit has an ID `0a8ce60`, which has done on the master branch.

To see the ***history of the commits***, we can:

- Use the command `git log`
- Use the other command `git log --oneline`.

  This command will give us the history only with the messages we have wrote, with no id.

3.6.1 Seeing Commits

Now in this section we will learn how to see the commits, that is see our project through different points in the commit history.

An example is shown in Figure 3.5.

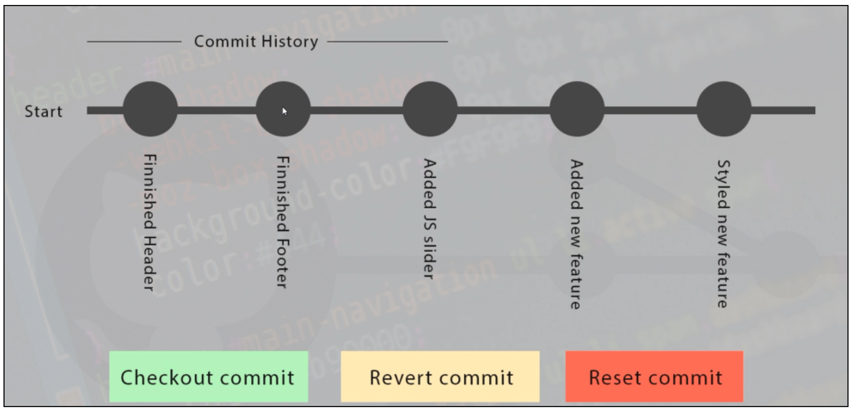


Figure 3.5: Commits Seeing

There are 3 ways to see the commit as indicated by the boxes in Figure 3.5.

- Checkout commit: suppose we want to see the 2nd commit. This command will let do it in a read only mode, so we can't do anything to the 2nd commit or any.

  Command: `git checkout commit_id`

  Note: after seeing the commit, we need to type `git checkout Master`, so we can go again to the current time, because when have used `git checkout commit_id`, it is like that we have go back in time in order to see the code.

- Revert commit: Suppose we have decided that we don't need the 3rd commit, we can use the revert command and it will delete this (not exactly delete it, we will see how later), and it will make like it was not their at all.

  Command: `git revert commit_id`

  The `commit_id` must belong to the commit we want to delete it.

  When we use the command `git revert commit_id` what will happen is:

  – It will create a new commit
  – This new commit will eliminate the effect of the commit we want to delete
  – If we type `git log --oneline`, the commit we want to delete it will not removed, instead we will see it along with a new commit named `revert something`, which indicates that this commit will eliminate the commit we want to delete it.

- Reset Commit: suppose we have decided that the we don't want the last 3 commits, and we want to get back to the 2nd commit and start from their. We can use the reset command, and it will delete permanently the last 3 commits.

  There is 2 ways we can use the reset mode:

  - `git reset commit_id`
  - `git reset commit_id --hard`

  Now if we use `git reset commit_id`, the commits will be gone, but the new changes in our files will still their, only the commits will be gone.

  We can do this for example if want to reduce the number of commits from 5 to 2, but keeping with files because we are sure that this will be our final decision.

  If we want to litteraly go to the status of the code of the 2nd commit, we use the command: `git reset commit_id --hard`

3.7 Branches

Now so far, we were working on the master branch.

Usually the master branch contains the stable version of the code, that will be published later.

Suppose we want to add new features to the code, and we don't want to mess up the stable code, what can we do is to add a new branch as shown in Figure 3.6, work in these new features, and once we have made sure that these features are stable, we can merge the commits, and get back to the master branch.

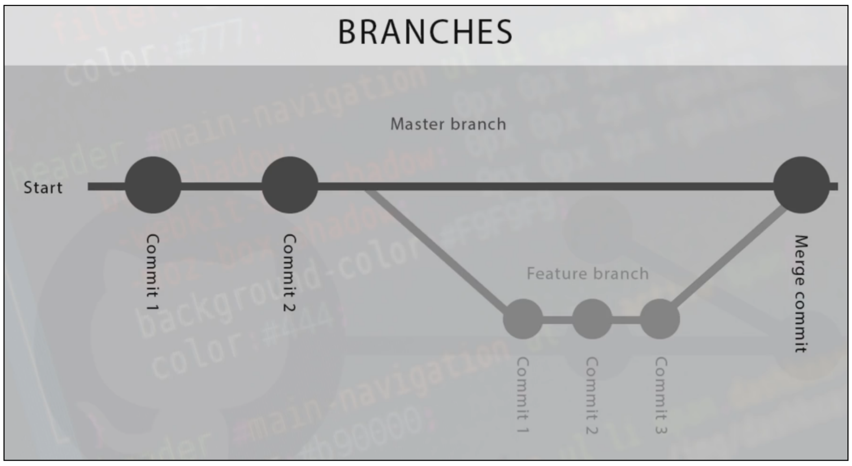


Figure 3.6: Commits Seeing

Also we can do 2 branches for example, like to make 2 teams work in parallel on 2 branches.

Now when we open the `Git Bash` command, it will indicate to us that we are in the `master` branch.

Example: `USER@LENOVO-PC MINGW64 ~/Desktop/Git_Test_2 (master)`

Suppose that we want to work on some new feature on some different branch.

1st we ***create a branch*** named feature-1.

Command: `git branch feature-1`.

To ***see all the branches we have***, we use the command `git branch -a`. Also it will give us at which branch we are in, by putting * next to the branch we are in.

```
USER@LENOVO-PC MINGW64 ~/Desktop/Git_Test_2 (master)
$ git branch featur-1

USER@LENOVO-PC MINGW64 ~/Desktop/Git_Test_2 (master)
$ git branch -a
  featur-1
* master
```

Here it is saying that we have a branch called `feature -1`, and our current branch we are in is the `master` branch, since there is * next to it.

The 2nd step is that we ***need to move to this created new branch*** named `feature-1`, because till now we are still on the `master` branch.

To move from `master` → `feature-1` branch: `git checkout feature-1`.

We can also checkout branches the same way we do for commits.

Example of branch moving:

```
USER@LENOVO-PC MINGW64 ~/Desktop/Git_Test_2 (master)
$ git checkout featur-1
Switched to branch 'featur-1'

USER@LENOVO-PC MINGW64 ~/Desktop/Git_Test_2 (featur-1)
```

If we want to create a branch and directly move to this branch all in one command: `git checkout -b feature-1`, where `b` stands for branch.

Note:
Now we are on this new branch called `feature-1`,2 things we have to consider:

1. In this new branch, all the status of the code will get copied.

   So in other words, if we have like 3 text files at the starting point of the branch, we will have them

2. However, even if we have commit some new changes (new text files) in the new branch, and don't do the merge step, the new files won't be added to the `master` branch.

   So if we get back to the `master` branch, we will find the original files.

If we want to ***delete a branch***:

1. we have 1st to move again to the `master` branch
2. use the command : `git branch -d feature-1` where `d` stands for delete.

Now to ***merge some branch***, we need:

1. Get back to the branch we want making the merge, usually the `master` branch.
2. Use the command: `git merge featur-1`

Merge Conflict:

Suppose we have 3 text files, named `File_1`,`File_2`,`File_3`.

We have decided to test some features, so we will do a new branch, and our work is on `File_3`.

Now some new person come in and directly deicde to work on this new feature of `File_3`, but from the `master` branch.

So when I finish and he finish, and the both us merge the files (well the person will only commit the change, and I will be merging), we will have ***merge conflict***.

The solution is simply to tell the guy to work on some branch or to work with me.

So the lesson is pay attention before merging if we have some other people working with us and do merging.

3.8 Github

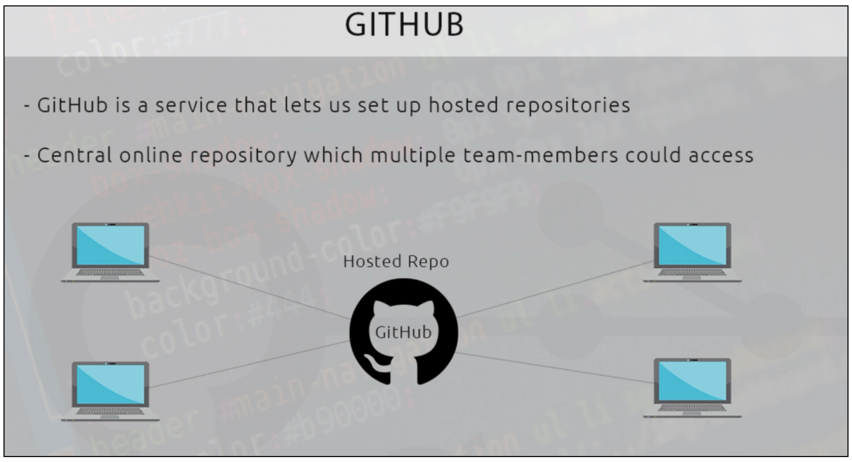

GITHUB
- GitHub is a service that lets us set up hosted repositories
- Central online repository which multiple team-members could access
Hosted Repo
GitHub

Figure 3.7: Github

Github is ***remote repository*** in which we can store our project in it. Anyone with computer and internet access can view then the code or the files.

Imagine that there are some project in Github. First operation we can do is to ***clone the repository***, so we can have a local version of the project on our computer.

2nd operation we can do is known as ***pushing***. This happen when a person works on some feature for the code, then when he finishes, he can ***push*** the code to the remote repository and the code will be updated.

The 3rd operation which come naturaaly after pushing is ***pulling***. After the code has been updated by adding feature of the 2nd step, we can ***pull*** the code so we can update the local version in our computers by adding this new feature.

3.8.1 Pushing

When working with Gihub, we have 2 cases:

1. We started some project on our computers and we decide to put it on Github so other developers can work with it
2. We didn't start any project yet, we create a Github repository first and clone the project to our computers so we can work on it

We will handle the 2 cases. We start first by case 1.

Pushing a local repository → Github:

1. Make sure that everything is commit by typing `git status`
2. We click the button `New` and we give this repository a name as shown in Figure 3.8.

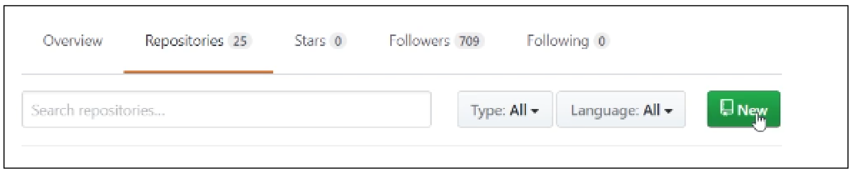
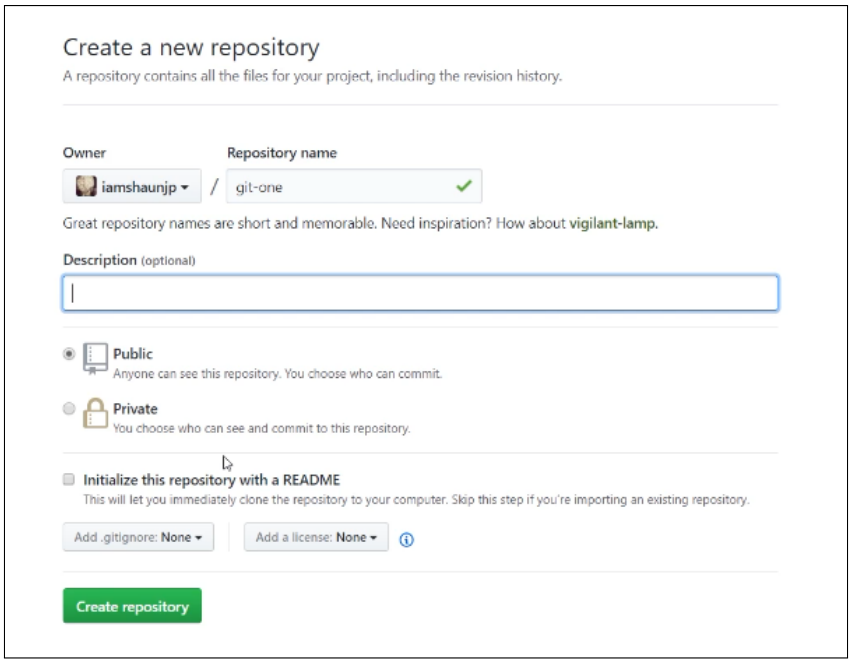


Figure 3.8: Clicking the New button

- There are many options when creating a repository, like make it private or public, initialize it with a read me file, ⋯ as shown in Figure 3.9 .



Create a new repository

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

Owner: iamshaunjp / Repository name: git-one

Great repository names are short and memorable. Need inspiration? How about vigilant-lamp.

Description (optional)

● Public — Anyone can see this repository. You choose who can commit.

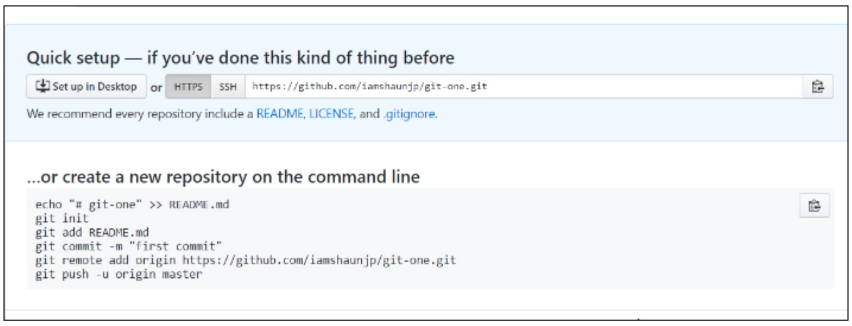
○ Private — You choose who can see and commit to this repository.

☐ 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

Figure 3.9: Filling Information about the repository

3. After doing this, Github will give us some `url`, we copy this `url` because we will use it, by clicking the button next to it as shown in Figure 3.10.



Quick setup — if you've done this kind of thing before

```
https://github.com/iamshaunjp/git-one.git
```

We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# git-one" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/iamshaunjp/git-one.git
git push -u origin master
```

Figure 3.10: Each Repository will have some url

4. We open our project on our local computer, and use this command: `git push url master`. For now we will push the master branch, but we can push other breaches too, we will do this later.

After we done this, we should see our code, and we can see the branches, the commit history.

<u>Set up an Alias to the url Repository:</u>

Instead of copying and pasting each time the `url`, which tend to be a little bit long, we can git it an alias name and use this alias instead.

To set us an alias name to our `url` repository: `git remote add alias_name url_repo`, we can choose any name for the alias. Majority of the people choose the name origin.

One advantage about this alias name, is if we do some change in the local repository and we want to push this new version, we can us the alias name in the `push` command instead of the `url` name: `git push alias_name`.

Now we move to the 2$^{\text{nd}}$ type of creating a project.

<u>From Github → our local computers:</u>

1. We click the `New` button again and give the repository a name
    - It is preferable to initialize the repository with a read me text file because we have created the repository directly in Github in this case
2. We copy the `url` of the repository
3. We navigate to the directory we want to clone this repository
4. Open `Git bash` and type the command: `git clone url_repository`

When creating a project from Github to local computers, Github will directly give the repository an alias name, which is set to be named `origin`, so if we make some changes on our local computers and we want to push the code again, we can directly use `origin` in the `push` command, and we don't have to create an alias name as in case 1.

3.8.2 Collaborating on Github

Note: review the concept of branch first in section 3.6.

On Github, we can do collaboration by pushing branches so other people can review the code.

When we push the branch we have worked on our local repository, we ***don't merge*** it with the `master` branch which exists locally on the computer, and then push the `master` branch to Github. If we do this, it will overwrite the remote `master`, but other people may not agree with this merging operation because our feature can contain some bugs, need to be modified, · · · . So ***what we do is that we push only the branch***, then we merge it with the master branch after making sure with the reviewers that everything is correct.

Tip:

Before working on some new branch, it is a good practice that we ***pull the master branch*** in case someone did some update or some changes. To pull we type the command: `git pull origin master` where `origin` is the alias name of the `url` repository.

When finishing working with the branch, we can push by typing: `git push origin branch_name`.

After pushing the branch, if we go to the Github, we should see an update where we have some new branch logo, along with the name of the branch, as shown in Figure 3.11. The name of the branch in is `index-html`.

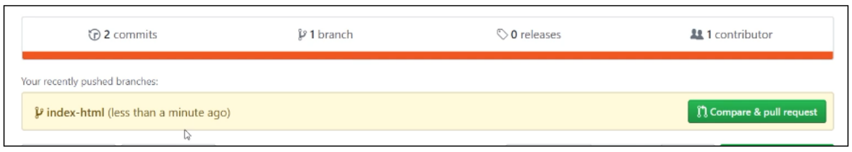
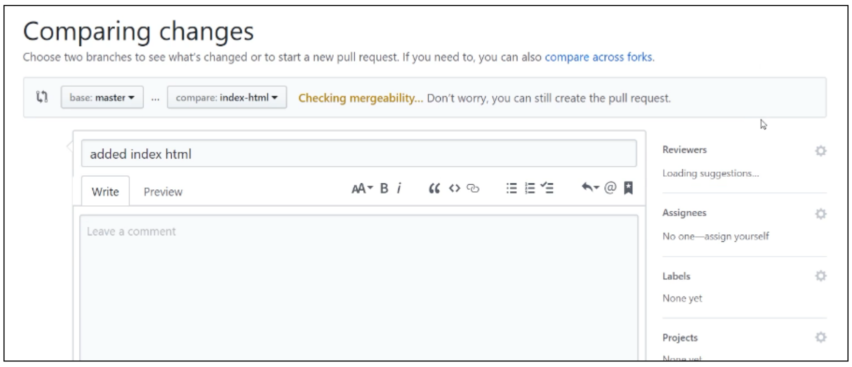

2 commits
1 branch
0 releases
1 contributor
Your recently pushed branches:
index-html (less than a minute ago)
Compare & pull request

Figure 3.11: Pushing the branch into Github

In Figure 3.11, we have the green button said: Compare and pull request. If we click this button, we can compare the branch with the current state of the `master` branch, and we can make a pull request which demand to merge this branch with the `master` branch.

Now we click on the green button of Figure 3.11, and we obtain options in Figure 3.12


Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base: master ... compare: index-html Checking mergeability... Don't worry, you can still create the pull request.

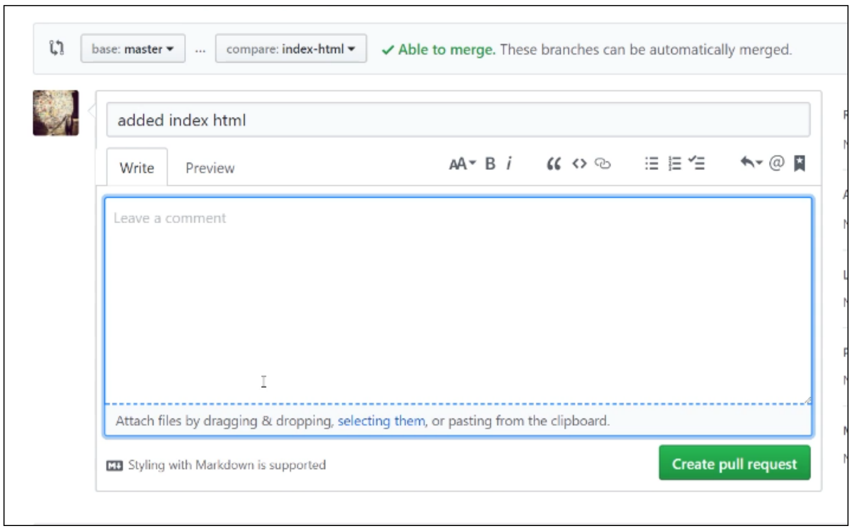
added index html

Leave a comment

Figure 3.12: Compare and Pull Request

In Figure 3.12, we see the message we typed when we committed the branch `index-html`, which is `added index html`. We can also leave some comments which explain to other developer the feature we worked on.

The next step is we can press the green button named `Create Pull Request` (which is on the same space of the options of Figure 3.12), shown in Figure 3.13.


base: master ... compare: index-html ✓ Able to merge. These branches can be automatically merged.

added index html

Leave a comment

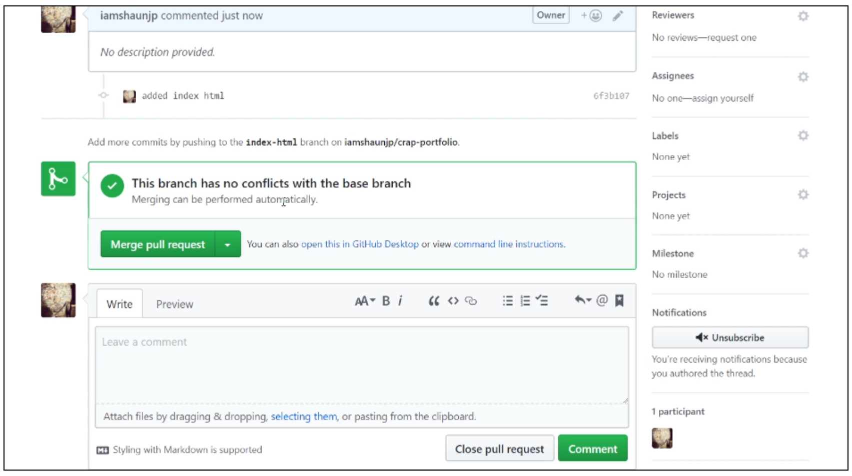
Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported

Figure 3.13: Create Pull Request

The button `Create Pull Request` means that we send some demand to merge our branch `index-html` with the `master` branch. The other developers will get some ping or some notification about our request of merging, and they can review the code and see if we can do the merging or not.

When we click `Create Pull Request`, we get Figure 3.14.


iamshaunjp commented just now

No description provided.

added index html 6f3b107

Add more commits by pushing to the **index-html** branch on **iamshaunjp/crap-portfolio**.

This branch has no conflicts with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported

Reviewers: No reviews—request one

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

You're receiving notifications because you authored the thread.

1 participant

Figure 3.14: After we do pull request

We can see in Figure 3.14 that we have:

- A green box which indicate that no conflict in case of merging
- Some box for comments: this is for other developers so they can leave comment for us
- Also in the same page of Figure 3.14 there are the options (they are not shown due to cropping)
  - `Commits` so the developers see the commits I have made
  - `File Changes`, this options will contains the files. It has 2 colors:
    - the red indicates the things we delete
    - The green indicate the thing we added
- Once the review has finished, we can click on the button `Merge Pull Request` shown in Figure 3.14, then click again `Confirm Merge`, and the branch will be merged `Master` branch.

3.8.3 Forking

Usually, when developers finish working with a certain project, they will create some kind public repository, and make it available to other developers to do some enhancement or contribution.

So how we can do some contribution?

If we have seen for example some project on some account, and we like to add some features to the code, we need to do ***forking***.

Forking will copy the project belonging to the account of the other developer (the one we want to do some contribution to it) and put this project to our account.

Next step is to clone this repository, do the changing we want, then push back again the project. When we push back the project, the project will be pushed into our account, and not the other developer account.

What we can do is go to the repository belonging to the developer we want to make contribution to, and click on `New pull request` → `Create Pull Request`, and it will pull it to the developer master branch. One thing to be said, that we don't have the permission to merge with the master branch of the other developer without his permission, that's why only him has the right to do the merging after reviewing the code, and the button `Create Pull Request` will notify the developer that someone want to make some contribution.

He will get the files, along `+` sign next to the lines being added in the file.

3.9 Summary Table

Begin of Table 3.1	
Task	Command Syntax and Examples
Configuration and Create an Account	• Configure an email to make a git account: <u>Command:</u> `git config --global user.name "Ranim Tom"` • Configure the email now: <u>Command:</u> `git config --global user.email ranim1tom@gmail.com` • Check if everything is ok – `git config -list` (this command check all the configurations) – Output: ∗ `user.name = Ranim Tom` ∗ `user.email = ranim1tom@gmail.com`
Start a project inside a folder	1. Navigate to the folder first 2. Type `git init`
Creating a file	Use the command `touch`, `touch index.txt`
Knowing the tracked and untracked file	• `git status` • It will give us the untracked files in red

Table 3.1: Table of Summary Commands

Continue Table 3.1	
Task	Command Syntax and Examples
Staging Area Commands (section 3.4)	
Add a file to staging area	`git add Name_file.extension`
Removing a file from staging Area	`git restore --staged Third_text.txt`
Adding all files inside the project	• `git add .` • Between `add` and `.` there is a space • Adding specific type of files (let's say `txt`) – `git add *.txt`
Commit Commands (section 3.5)	
Doing a commit and inserting a descriptive message	`git commit -m "First change"`
See the history of commits	• `git log` – `git log` will display the commits even with their reference id • 2nd choice: `git log --oneline` – This command will display only the messages we have wrote,so no reference id • history of all commits from one author only (if we are working with a team for example) – `git log --author="Ranim"`
Seeing the status of the files by using commits (3.5.1)	• `git checkout commit_id` – This is read only mode – We need to type `git checkout Master` to return to the master branch

Table 3.1: Table of Summary Commands

Continue Table 3.1	
Task	Command Syntax and Examples
Continue Commit Commands (section 3.5)	
Eliminating Commits	1. Deleting one of the commits only, let's say commit number 3 • `git revert commit_id` • The `commit_id` must belong to the commit we want to remove, commit number 3 in this case 2. Let's say we have 5 commits, and we want to get back to the commit number 2 • `git reset commit_number_2_id` – This command will let us go back to the commit number 2, and the last 3 commits (commit 3,4 and 5) will be deleted, but the the new chnages will be preserved • `git reset commit_number_2_id --hard` – We will get back literally to commit number 2, and we will lose all the changes

Table 3.1: Table of Summary Commands

Continue Table 3.1	
Task	Command Syntax and Examples
Branching Commands (section 3.6)	
Create a branch	`git branch feature-1`
See all the branches	• `git branch -a` • It will give us at which branch we are in, by putting * next to the branch we are in
2nd step: move to this created new branch named `feature-1`, because till now we are still on the `master` branch	`git checkout feature-1`
Create a branch and directly move to this branch all in one command	• `git checkout -b feature-1` – `b` stands for branch
Delete a branch	1. we have 1st to move again to the `master` branch 2. use the command : `git branch -d feature-1` • `d` stands for delete.
Merge some branch	1. Get back to the branch we want making the merge, usually the `master` branch. 2. Use the command: `git merge featur-1`

Table 3.1: Table of Summary Commands

Continue Table 3.1	
Task	Command Syntax and Examples
Working with Github (section 3.7)	
Pushing a local repository → Github	1. Make sure that everything is commit by typing `git status` 2. We click the button `New` and we give this repository a name 3. After doing this, Github will give us some `url`, we copy this `url` because we will use it, by clicking the button next to it. 4. We open our project on our local computer, and use this command: • `git push url master`
Set up an Alias to the url Repository	• `git remote add alias_name url_repo` – We can choose any name for the alias – Majority of the people choose the name `origin` • Advantage about using alias name: If we do some change in the local repository and we want to push this new version, we can use the alias name in the `push` command instead of the `url` name as follow: – `git push alias_name`

Table 3.1: Table of Summary Commands

Continue Table 3.1	
Task	Command Syntax and Examples
Continue Working with Github (section 3.7)	
From Github → our local computers	1. We click the `New` button again and give the repository a name • It is preferable to initialize the repository with a read me text file because we have created the repository directly in Github in this case 2. We copy the `url` of the repository 3. We navigate to the directory we want to clone this repository 4. Open `Git bash` and type the command: • `git clone url_repository` Note: When creating a project from Github to local computers, Github will directly give the repository an alias name, which is set to be named `origin`, so if we make some changes on our local computers and we want to push the code again, we can directly use `origin` in the `push` command, and we don't have to create an alias name as in case 1.

Table 3.1: Table of Summary Commands

Continue Table 3.1	
Task	**Command Syntax and Examples**
Collaborating on Github (3.7.2)	
General Steps: • We take the project from Github • Work on some new feature • When finish working, we push this new feature (and not the `master` branch), so other reviewers can see it. • After finishing, we can do the merging with the `master` branch	1. Pulling: Before start working on some new feature, it is a good practice to pull the code into our computers in case someone made some changes • `git pull origin master` 2. Pushing a branch (upload our work to Github) • `git push origin branch_name` • We should see an update about the branches after pushing, that a new branch is added • Don't forget that even if we are working with branches, we should make sure that everything is committed in the branch before doing the pushing 3. Next step: demand to merge our branch with the `master branch` • Click the button `Create Pull Request` • Other developers will get a notification about the request of merging, and they can start to review the code 4. Once the review is finished, we can click the button `Merge Pull Request`, then click again `Confirm Merge` and the branch will be merged with `Master` branch.
End of Table 3.1	

Table 3.1: Table of Summary Commands

Chapter 4

Tikz

Website contain helpful topics: http://quicklatex.blogspot.com/

4.1 Basic Plot

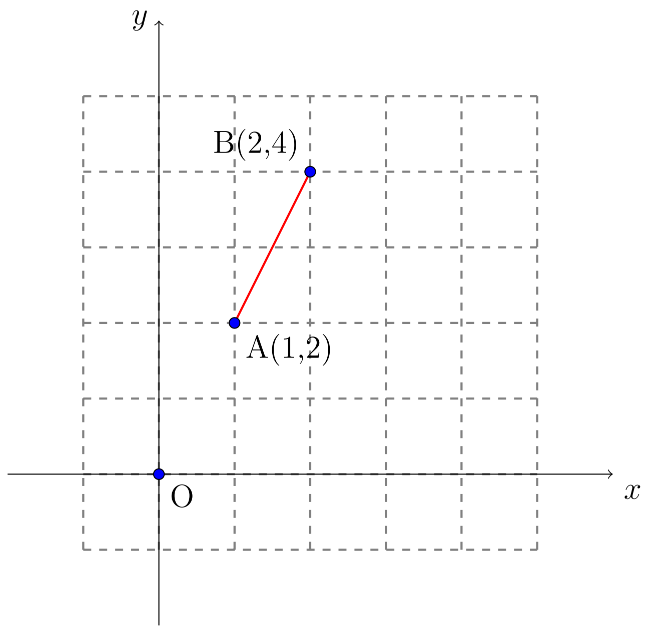


Figure 4.1: A 2D Cartesian System

4.2 Drawing a basic flow chart

- Include the `tikz package` and its extension:
  - `\usetikzlibrary{shapes,shadows,arrows}`.
- In working with complex diagrams like flowcharts, define always the main component you need.

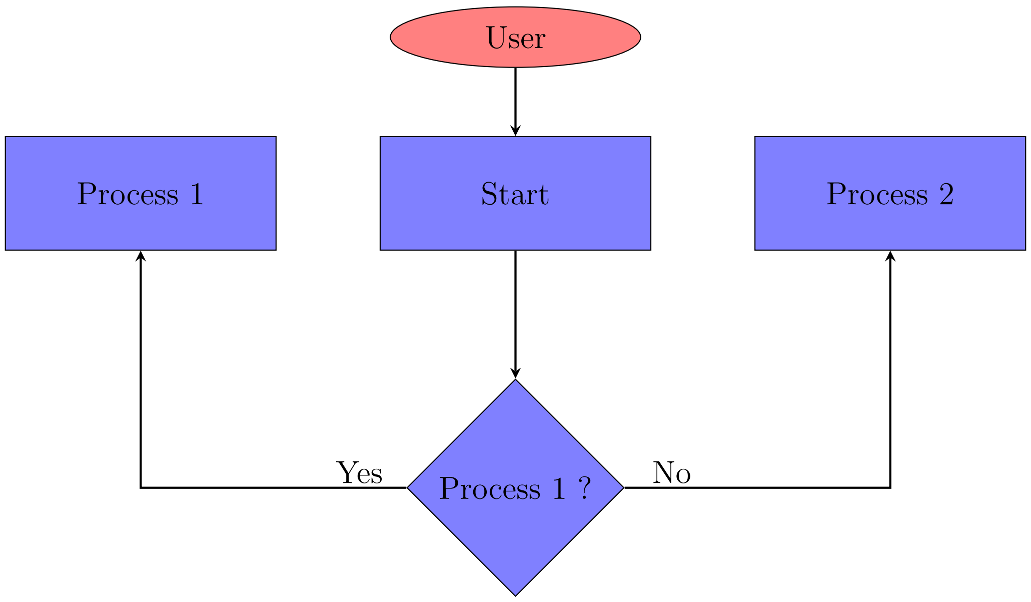


Figure 4.2: A simple flow chart

Some other flow charts of a hierarchical organization:

- http://www.texample.net/tikz/examples/class-diagram/
- http://www.texample.net/tikz/examples/hierarchical-diagram/