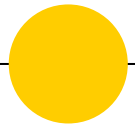
Command line & Git basics



Command line concepts

- **Command line** = **Terminal** = **Shell** = the place to communicate with your computer without using a Graphical User Interface (GUI).
- **Bash / ZSH** = the language used in Unix-based shells.
- ~ means your home directory
- **Directory** = Folder
- **Absolute path** = location of a file from the root directory.
- **Relative path** = location of a file from the present working directory (pwd)

Command line commands

- **ls** | ls -a | ls -l
- **pwd**
- **cd** | cd ..
- **history** | arrow keys
- **touch**
- **cat | head**
- **echo**
- **clear**
- **rm**
- **mkdir**
- **ctrl + A | ctrl + E**
- **ctrl + U | ctrl + K**
- **--help**
- **TAB key**

Git - Checking prework

Run the following command:

```
$ git config --global user.name
```

It should print the empty results. That is because right after the installation the user name is not configured. Set the user.name configuration option using the following command:

```
$ git config --global user.name "John Doe"
```

Instead of John Doe type your name and surname.
Next, run the command:

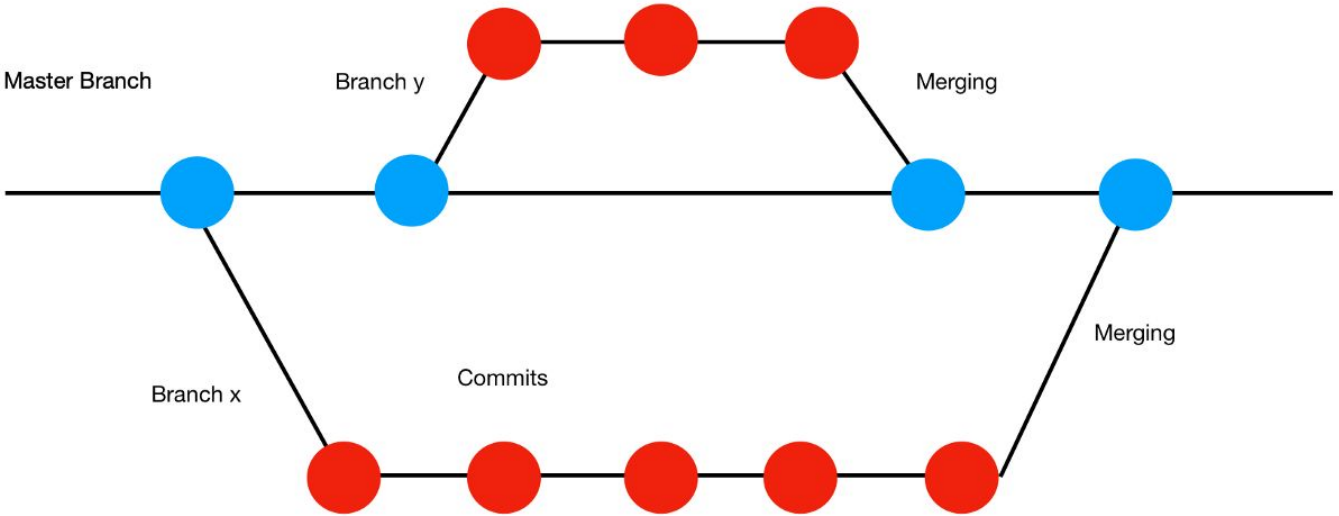
```
$ git config --global user.email john.doe@example.net
```

This command will set your email.

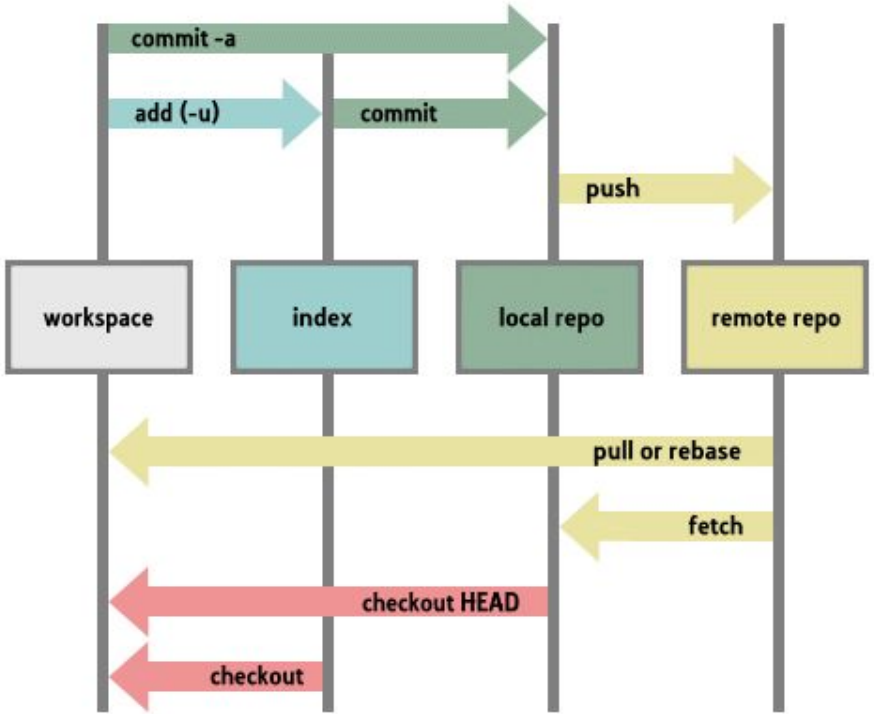
Why do we use git?

- Keep track of all files in a project
- Record any changes to project files
- Restore previous versions of files
- Compare and analyze code
- Merge code from different different team members

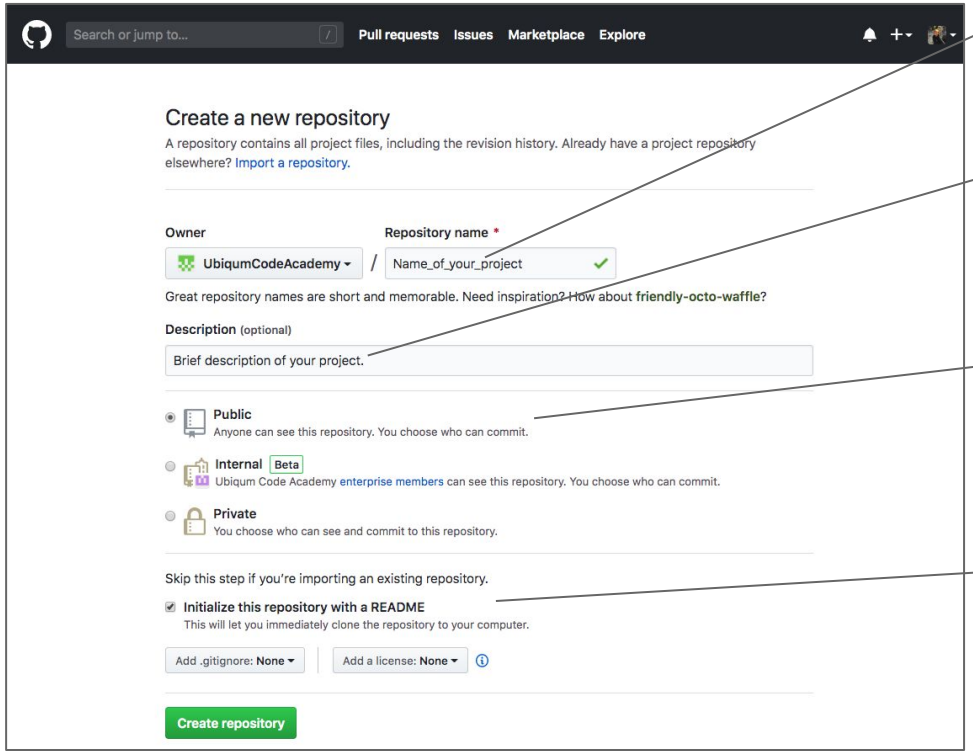
Version control



Git workflow



Creating a repo (Github)



Meaningful name. Better "predict-survey-data" than "module-2".

Concise description. Your project in one sentence. E.g. "Using Machine learning to predict brand preference on survey data."

Consider the privacy. Is your repo part of your public portfolio? Go public. Do you have confidential data? Private!

Add a README. It's the place where you will describe your project extensively (using Markdown syntax).

Cloning a repo

- git clone <git's repo url>
- Git's 'default' branch is called master.
- Git has support for multiple remotes. By default, your local copy will be configured to use one remote, named origin
- The command above creates a cloned directory, initializes a .git directory inside it, pulls down all the data in the server for that repository, and checks out a working copy of the latest version (it sets up your local master branch to track the remote master branch on the server you cloned from)

Adding content manually

- touch file1
- git status (will show file1 as untracked)

- git add file1 or git stage file1 (will add the change to the staging area; stage is an alias for add)
- git status (will now show file1 as added and ready to be committed)

- git commit –m “added file1” (will commit everything in the staging area into our local repo)
- git status (will now say there’s nothing to commit… our local repo is ahead of the remote by 1 commit)

Modifying some content locally (I)

- echo changed >> file1
- git status (will show file1 as modified)
- git add file1(will add the change to the staging area. Yes, we have to “add” modifications the same way we added new files)
- git status (will now show file1 as added and ready to be committed)

Modifying some content locally (II)

- git commit –m “modified file1” (will commit everything in the staging area into our local repo)
- Or we could just use –a (add automatically changes for already versioned files) or specify the file to commit when committing, so that there’s no need to stage the change first:
- git commit –a “modified file1” or git commit –m “modified file1” file1

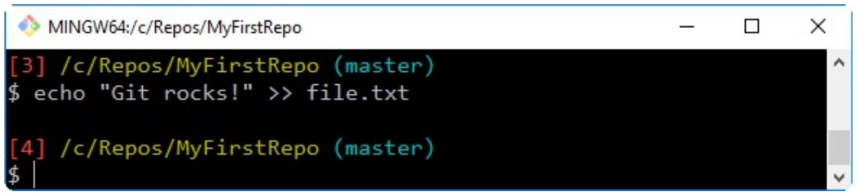
Pushing content to remote server

- git push origin master

Git - code along

1) Adding a file

Let's create a text file, just to give it a try:

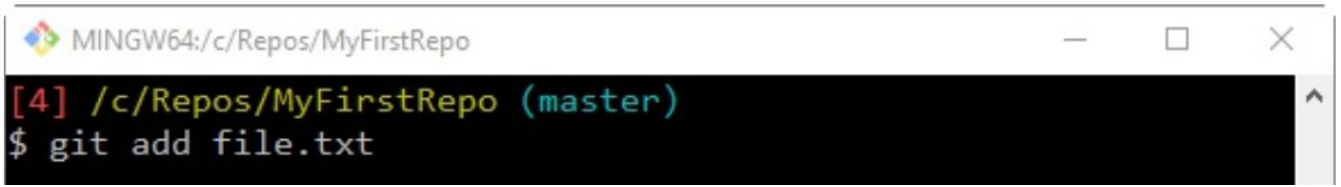


And now what? Is that all? No! We have to tell Git to put this file in your repository, *explicitly*. **Git doesn't do anything that you don't want it to do**. If you have some spare or temp files in your repository, Git will not take care of them, but will only remind you that there are some files in your repository that are not under version control (in the next chapter, we will see how to instruct Git to ignore them when necessary).

Git - code along

1) Adding a file

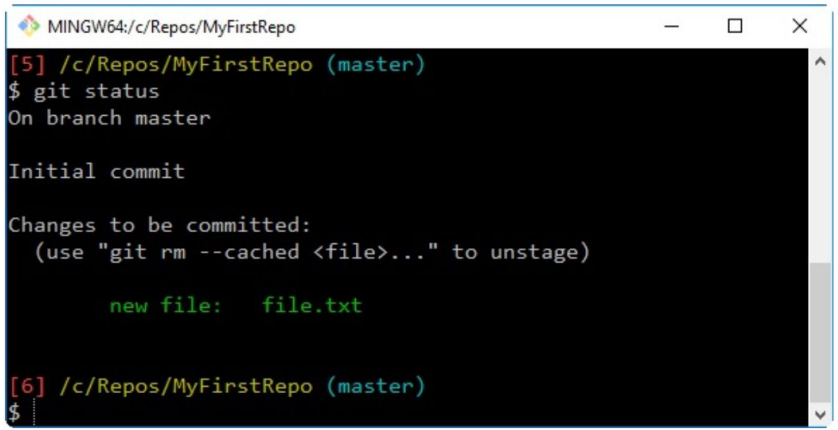
Okay, back to the topic. I want `file.txt` under the control of Git, so let's add it, as shown here:



Git - code along

1) Adding a file

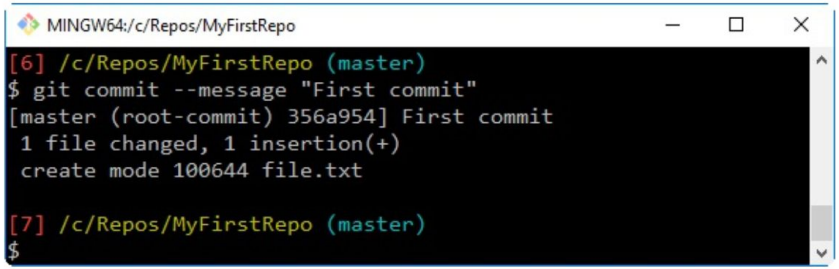
Using the `git status` command, we can check the status of the repository, as sho
in this screenshot:



Git - code along

1) Committing the added file

At this point, Git knows about `file.txt`, but we have to perform another step to fix the snapshot of its content. We have to commit it using the appropriate `git commit` command. This time, we will add some flavor to our command, using the `--message` (or `-m`) subcommand, as shown here:



With the commit of `file.txt`, we have finally fired up our repository. Having done the first commit (also known as the root-commit, as you can see in the screenshot), the repository now has a `master` branch with a commit inside it.

Git Basic Commands - fixing changes

Undo uncommitted changes

- echo changed >> file1.txt
- git status (will show file1.txt as modified)
- **git checkout -- file1.txt** (will revert the changes and leave the file as it is in the index.. i.e. as it was when last committed)
- Note that while the git checkout [branch] command will not override local changes, the same isn't true when specifying specific files in the command.

Reset

- git reset --soft will move the branch pointer (i.e. discarding the last commit). It won't touch the index or the working copy, so that the discarded changes will show up as staged for commit
- **git reset** [--mixed] does the same as previous but, in addition, it will clear the index as well, so that the discarded changes will show up as modified and unstaged
- git reset --hard does the same as previous but, in addition, it will clear the working copy as well, so that the discarded changes will just disappear forever. Use this one carefully!

Unstage

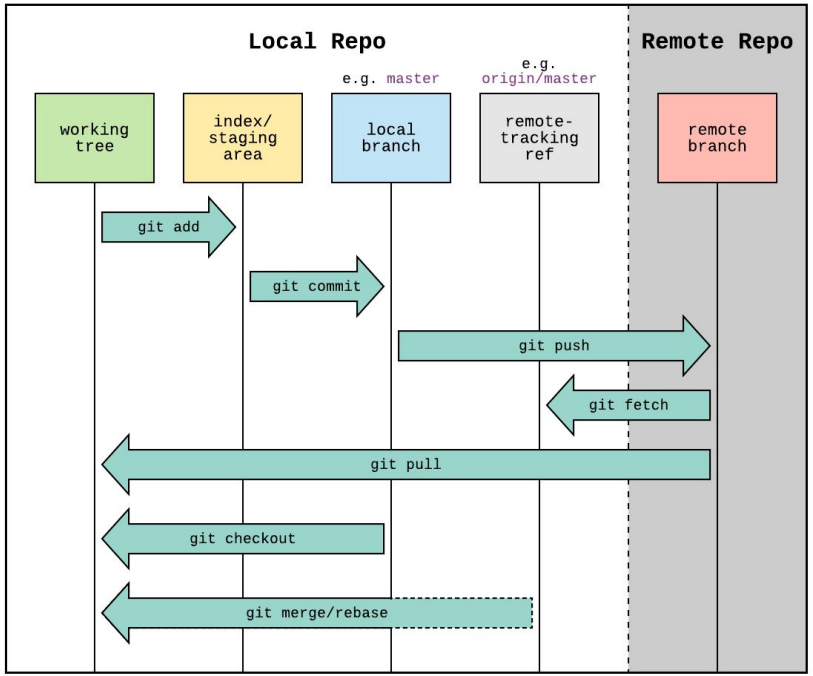
- echo changed >> file1.txt
- git status (will show file1.txt as modified)
- git add file1.txt (change is now staged)
- **git reset HEAD file1.txt** (file is now shown as modified but unstaged)

Amend last local commit

- echo changed >> file1.txt
- git status (will show file1.txt as modified)
- echo changed >> file1.txt
- echo changed >> file2.txt
- git add file1.txt
- git commit –m “modified two files”
- git add file2.txt (we forgot to add file2.txt)
- **git commit –-amend –m “modified two files… for real”**
- This last commit will replace the previous one. It’s like the first commit never took place.
- Don’t use this on commits that have already been pushed !!!

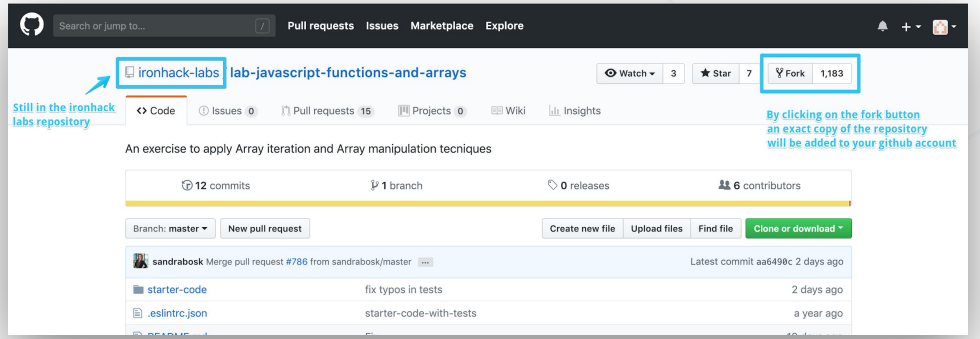
Git flow

Flow

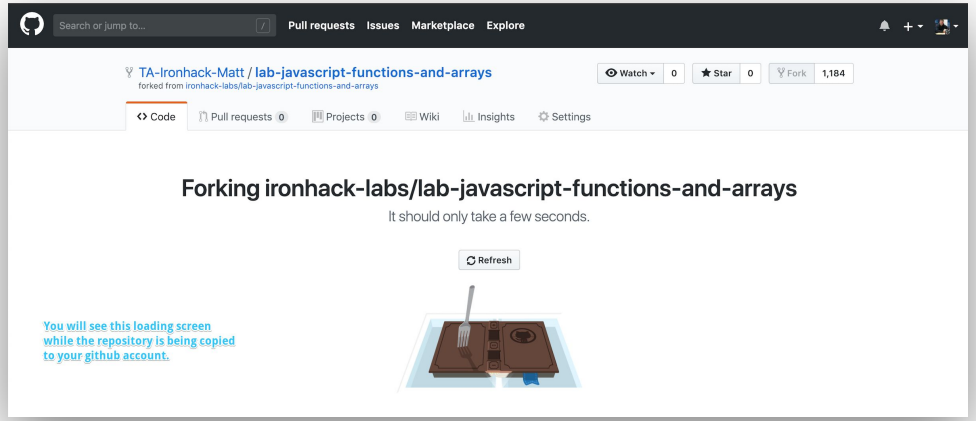

Local Repo
Remote Repo
e.g. master
e.g. origin/master
working tree
index/ staging area
local branch
remote-tracking ref
remote branch
git add
git commit
git push
git fetch
git pull
git checkout
git merge/rebase

Forking the Repository

STEP 1
Fork the assignment repository to your GitHub account.



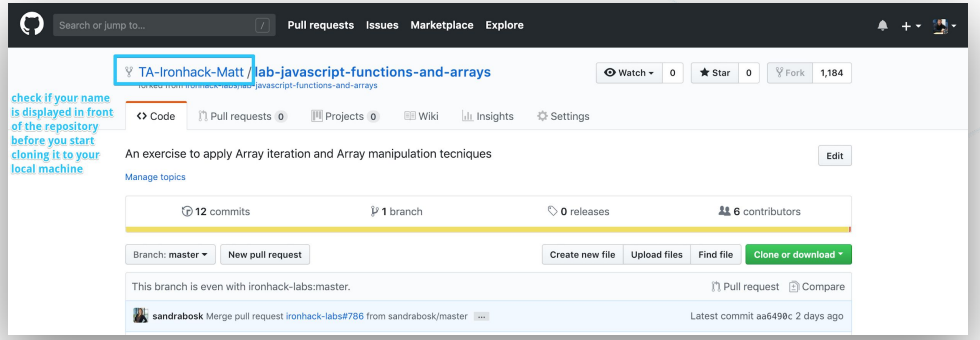
... during the forking process you will see a loading screen. After the forking is done successfully you will be redirected to the **copied** repository **on your GitHub account**.



Cloning the repository

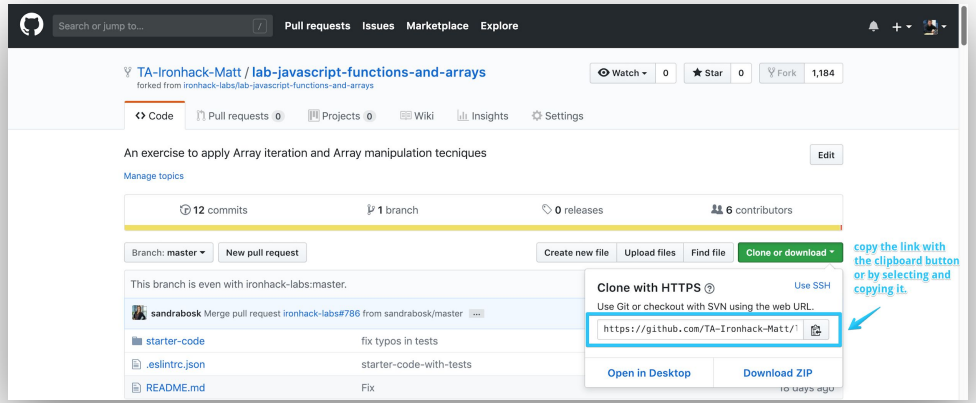
STEP 2

Check if the repository is correctly cloned to your github account before you start with the cloning process.



STEP 3

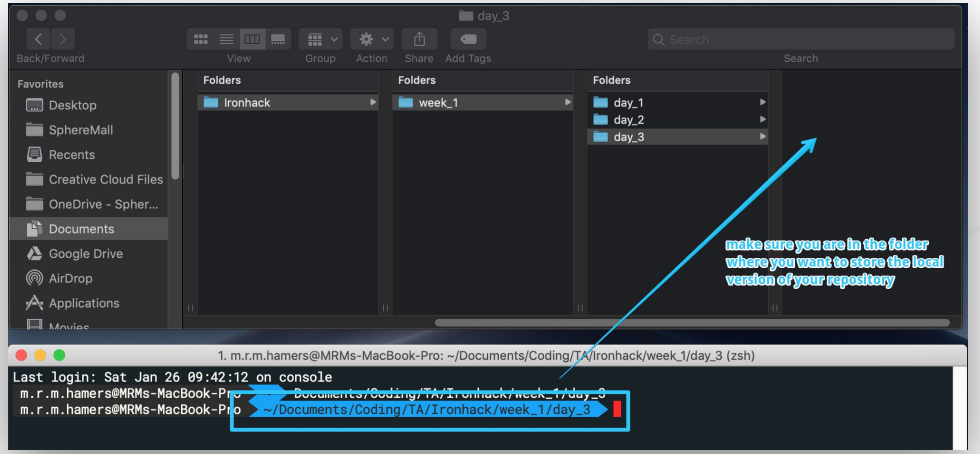
Click on the green 'Clone or download' button and copy the link by selecting and copying or by using the clipboard button.



Clone remote to local

STEP 4

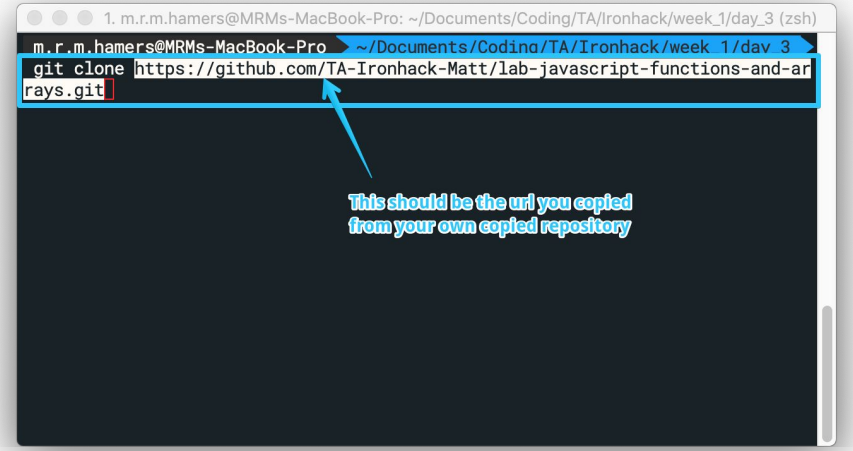
Open your terminal and navigate to the location where you want to store the local version of your repository.



STEP 5

When you navigated to the correct folder in your terminal run the following command:

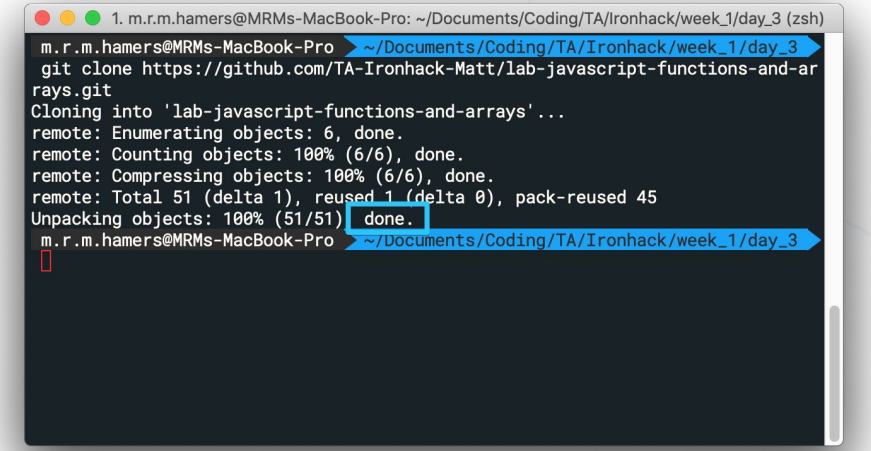
```
$ git clone <paste url from clipboard>
```



Clone remote to local

STEP 6

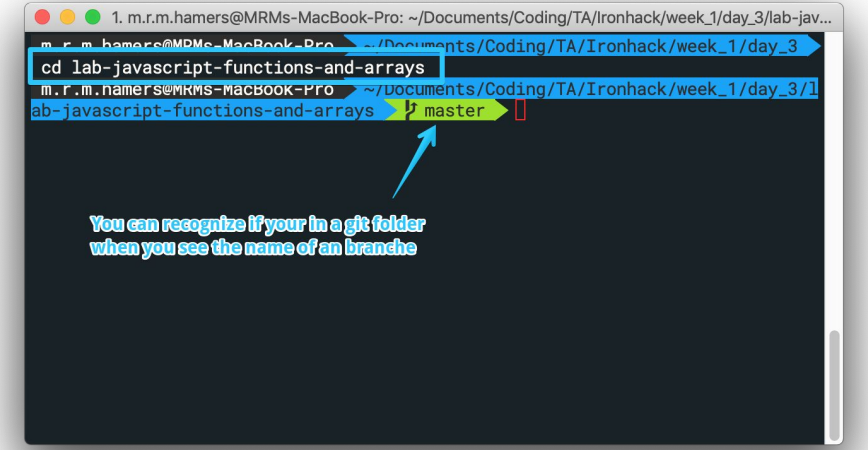
Once the cloning process is done you see the following information in your terminal.



STEP 7

Before you open the project, you have to navigate into the cloned folder because that's where git is initialized:

```
$ cd <name of cloned folder >
```

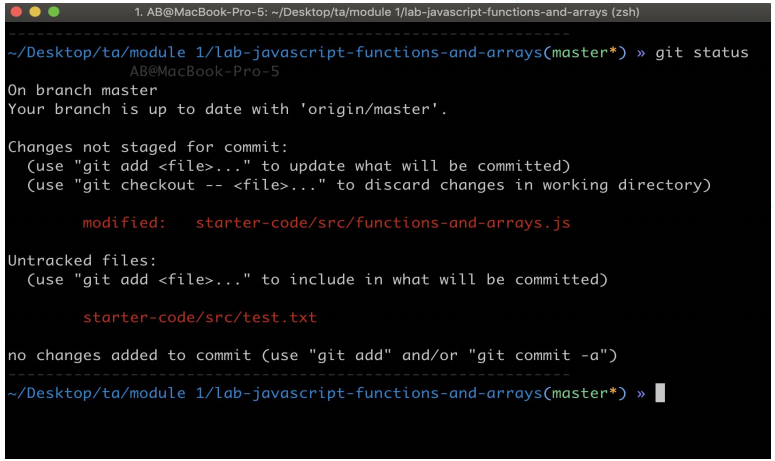


Add & commit changes

STEP 8

After you made changes to your project, you can check which files you changed by running:

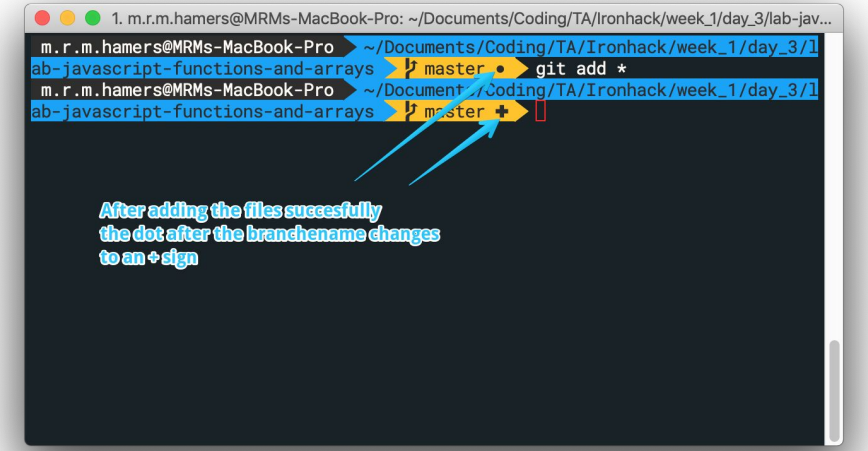
```
$ git status
```



STEP 9

If you would like to add to git all the files you made changes in, you should run the following command inside your git folder:

```
$ git add *
```

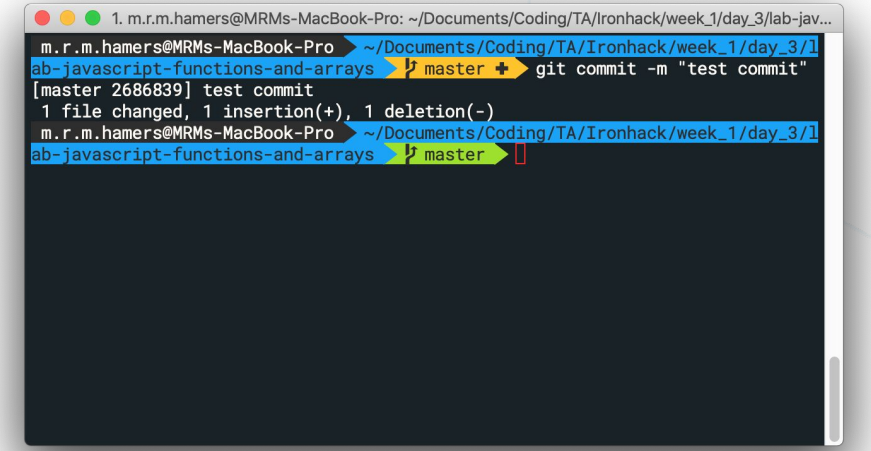


Add & commit changes

STEP 10

After all changed files are added you should commit your files and add a commit message describing the things you've changed, with the following command:

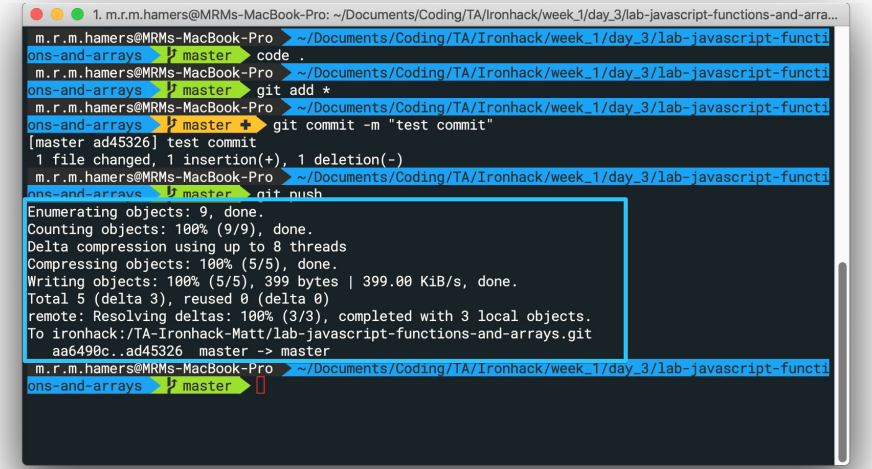
```
$ git commit -m "<your message>"
```



STEP 11

After committing your changes, you can push the commits from your local machine to your remote repository on GitHub with the following command:

```
$ git push origin master
```



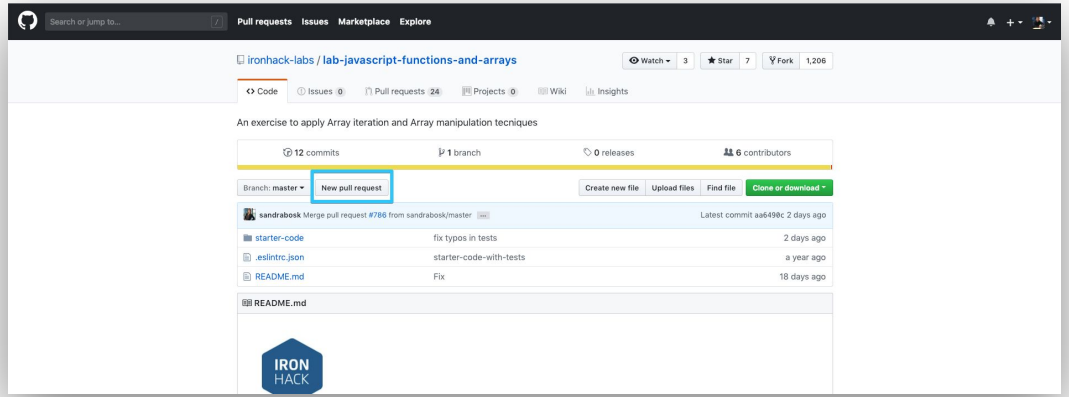
Create pull request

STEP 12

After pushing your changes from your local master to the remote master, your commit should be visible on GitHub (the changes you made locally are now available in GitHub repository as well).

STEP 13

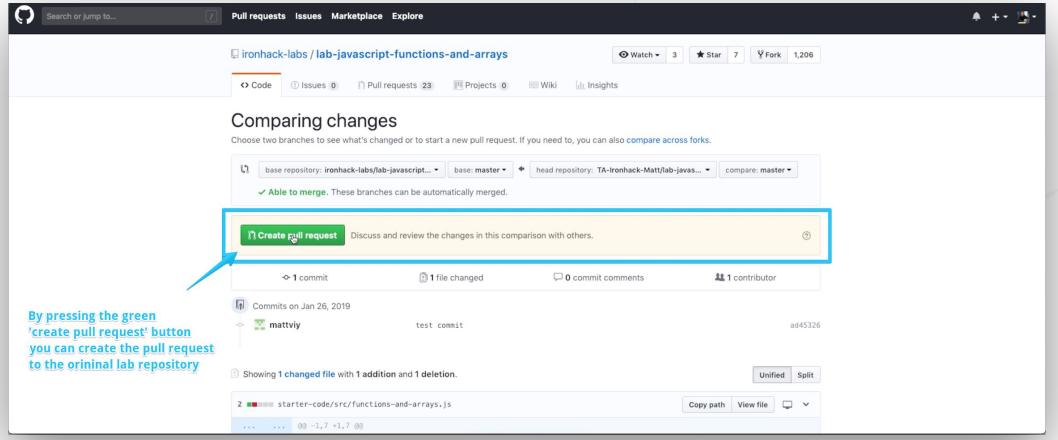
When your commit is successfully pushed to GitHub, you can create a pull request by clicking on the 'New pull request' button.



Create pull request

STEP 14

By pressing the green “create pull request” button you will create the pull request to the original lab repository.



Keep in mind that being handy with Git and GitHub is mandatory when being developer for multiple reasons:

- you’ll always be part of the team and collaboration is a must => how to collaborate on GitHub (you already know this!)
- you will make mistakes, but that’s okay, because you can pull up the previous version that was working => how to get previous version of my project
- always refer to official docs

Resources

Command line cheat sheet: https://www.git-tower.com/blog/command-line-cheat-sheet/

Github repo with bash resources: https://github.com/awesome-lists/awesome-bash

Git cheat sheet: https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

More git practice (after the lab!):
https://codeforphilly.github.io/decentralized-data/tutorials/actually-using-git/lessons/conflicting-branches/