pwd to see the current directory

Touch filename to add a new file

Mkdir nameofdirectory to make new directory

cd nameofdirectory to change directory

git config –global user.name(or email) “your name( or email)”

git config –list OR git config -l to view configuration list, eg: name email etc

git init to create our own repository

git clone URLofRepoToClone to copy an existing repository

git status to check status of repository

git log displays info about existing commits

git show displays info about given commit

Press j to go down in pager

Press k to go up in pager

q to quit pager

press q to get out of it

Up/down, spacebar, J, F

U, B

git rm somefilename will delete the file from repository

we’ll then commit this change using git commit -m “somefilename deleted”

git mv ABC XYZ to change file names (ABC will change to XYZ)

git log –oneline flag of git log i.e used to display less info about commits

Git log --stat stats like all the modifications in files

Git log -p detailed stats patch that is what all was changed

Git log -p --stat both combined

That's right! git log -p -w will show the patch information, but will not highlight lines where *only* whitespace changes have occurred.

By supplying a SHA, the git log -p command will *start at that commit*! No need to scroll through everything! Keep in mind that it will *also* show all of the commits that were made *prior* to the supplied SHA.

Git log -p fififi5 to open specific commit and all others will also open

Git show for specific commit

Git show fififi5 for only this commit

git log -p -2 will display patch info for last 2 commits

Fantastic job! I first used git log --oneline to find the SHA of the commit, then I used git log --stat with the SHA to find the right info.

git add add files from the working directory to the staging index

git commit take files from staging index and save them in repository

git diff displays the difference between two versions of a file

(output similar to git log -p command)

git add <file1> <file2> … <fileN>

*TIP: Did you also notice the helpful text that's located just beneath "Changes to be committed"? It says (use "git rm --cached <file>..." to unstage) This is a hint of what you should do if you accidentally ran git add and gave it the wrong file.*

*As a side note, git rm --cached is not like the shell's rm command. git rm --cached will not destroy any of your work; it just removes it from the Staging Index.*

or git add . for staging all files from the working directory.

The period refers to the current directory and can be used as a shortcut to refer to all files and directories (including all nested files and directories!).

The only thing to be careful of is that you might accidentally include more files than you meant to. Right now we *want* both css/app.css and js/app.js to be staged, so running this command is fine right now. But let's say you added some images to an img directory but didn't want to stage them just yet. Running git add . *will* stage them. If you do stage files that you didn't mean to, remember that git status will tell you the command to use to "unstage" files.

Git commit will open text editor too

*git commit -m "Initial commit" for specifying message sath k sath*

git commit -a -m “commit msg” for skipping the staging area only for tracked files

git diff and git log -p do the same thing(!log -p will show commited changes)

The git diff command can be used to see changes that have been made but haven't been committed, yet.

step 1 - touch .gitignore

.gitignore (read more)

to ignore some files to be accidentally staged

git checkout somefilename it reverts changes to modified files before they are staged.

To unstage staged changes we use some reset command, its opposite of add.

git tag add tags to specific commits

git branch allow multiple lines of development

git checkout switch bw diff brances and tags

git merge combine changes in diff branches

**TAGGING**

git tag -a to add tag

git tag to view tags

git tag -d to delete tag

git tag -a v1.0 a87984 tagging past commit

**BRANCHING**

**Git branch to see all the branches**

**Git branch sidebar to add a new branch named sidebar (at the most recent commit)**

**Git checkout sidebar shifts the head pointer to sidebar branch now onwards the commits are added to this sidebar branch.**

It's important to understand how this command works. Running this command will:

* remove all files and directories from the Working Directory that Git is tracking
  + (files that Git tracks are stored in the repository, so nothing is lost)
* go into the repository and pull out all of the files and directories of the commit that the branch points to

So this will remove all of the files that are referenced by commits in the master branch. It will replace them with the files that are referenced by the commits in the sidebar branch. This is very important to understand, so go back and read these last two sentences.

**git checkout master to move back to master branch**

**git log --oneline will clearly display info about branches too**

**Git branch -d sidebar to delete the sidebar branch if all its commits are merged and the data is secured that I available on someother branch too. Can only delete if we are not on sidebar branch currently.**

**Git branch -D sidebar to surely(forefully) delete the branch named sidebar**

**Git branch newbranchname SHA to add a new branch to the SHA mentioned**

**Git branch -b footer master to add a new branch footer based of master branch and move the head pointer to the footer, no matter which branch you are on**

**Git log --oneline --graph --all to show how branches will work**

The --graph flag adds the bullets and lines to the leftmost part of the output. This shows the actual *branching* that's happening. The --all flag is what displays *all* of the branches in the repository.

Running this command will show all branches and commits in the repository:

**MERGING**

**Git merge branchname**

**Git merge --abort to abort the merging**

**UNDOING(READ MORE)**

**git commit –amend alter the most recent commit**

**git revert SHA reverses given commit**

**git reset (use carefully) erases commits (in order)**

**Git commit –amend to edit the most recent commit**

git revert <SHA-of-commit-to-revert> creates a new commit with reverse changes

**git reflog for history**

git reset <reference-to-commit>

It can be used to:

* move the HEAD and current branch pointer to the referenced commit
* erase commits with the --hard flag
* moves committed changes to the staging index with the --soft flag
* unstages committed changes --mixed flag

Typically, ancestry references are used to indicate previous commits. The ancestry references are:

* ^ – indicates the parent commit
* ~ – indicates the first parent commit

*$ git checkout -- index.html*

*remove the uncommitted changes from the working directory*

**git reset HEAD <file> to remove a staged file**

**GITHUB**

open -a /Applications/Sublime\ Text.app to open in sublime

<https://jeffreyeverhart.com/2017/09/14/open-files-folder-sublime-text-terminal/> link to use sublime with git

|  |  |
| --- | --- |
| git clone URL | [Git clone is used to clone a remote repository into a local workspace](https://git-scm.com/docs/git-clone) |
| git push | [Git push is used to push commits from your local repo to a remote repo](https://git-scm.com/docs/git-push) |
| git pull | [Git pull is used to fetch the newest updates from a remote repository](https://git-scm.com/docs/git-pull) |

Git config --global credentail.helper cache will basically cache our github credentials so that we don’t have to enter them on git bash for every request we make to remote repository(in my case, I just needed to enter them once )

**Git remote –v to see the links for fetch and push**

**Git fetch** git fetch fetches remote updates but doesn't merge; git pull fetches remote updates and merges.

Git log origin/master to view commit history of fetched changes on remote repo branches

Git merge origin/master to merge/sync master branch of remote repo into our local branch

Git pull combines git fetch and merging

Git remote show origin for more info about origin

| **Command** | **Explanation & Links** |
| --- | --- |
| git remote | [Lists remote repos](https://git-scm.com/docs/git-remote) |
| git remote -v | [List remote repos verbosely](https://git-scm.com/docs/git-remote#Documentation/git-remote.txt--v) |
| git remote show <name> | [Describes a single remote repo](https://git-scm.com/docs/git-remote#Documentation/git-remote.txt-emshowem) |
| git remote update | [Fetches the most up-to-date objects](https://git-scm.com/docs/git-remote#Documentation/git-remote.txt-emupdateem) |
| git fetch | [Downloads specific objects](https://git-scm.com/docs/git-fetch) |
| git branch -r | [Lists remote branches](https://git-scm.com/docs/git-branch#Documentation/git-branch.txt--r); can be combined with other branch arguments to manage remote branches |

git remote update will fetch the contents of all remote branches and allow us to merge the contents ourselves.

**Git push -u origin refactor pushing branch refactor to remote repo (so that fellow collaborators can fetch and check if its okay)**

**Now we can merge this branch too with remote repos master branch**

**Example: refactors base was commit 3 of master, new commits added in refactor say 3a and 3b, also new commits added in master say 4 and 5**

**Above commands will help us keep the splitting track linear by changing base and doing fast forward merge instead of 3 way merge.**

**That is basically now 1,2,3,4,5(master)(refactor starts here now),3a,3b.**

**Now checkout to master and merge refactor (fast forward)**

**Now can delete refactor both remotely and locally**

**Git push –delete origin refactor**

**Git branch -d refactor**

**Git checkout refactor**

**Git rebase master**

**These two will help to keep the branch history linear by changing base.**

**Making changes in readme – ctrl+o + enter + ctrl+x**

git push origin master

**EXTRA**

**Git rebase -i**

**Git push -f**

**EXTRA NOTES FROM COURSERA TRANSCRIPT**

WEEK3

Although your friends might work on

their parts by themselves,

from time to time,

everyone needs to send out progress

updates to let each

other know what they've been working on.

You will then need to combine their work into

your own portion of the project

to make sure it's all compatible.

Using Git to manage a project

helps us collaborate successfully.

Everyone will develop their piece of

the project independently in

their own local repositories

maybe even using separate branches.

Occasionally they'll push finished code into

a central remote repository where others

can pull it and incorporate

it into their new developments.

So how does this work?

Alongside the local development branches like master,

Git keeps copies of the commits that have been

submitted to the remote repository and separate branches.

If someone has updated a repository since

the last time you synchronize your local copy,

Git will tell you that it's time to do an update.

If you have your own local changes

when you pull down the code from the remote repo,

you might need to fix merge

conflicts before you can push your own changes.

In this way Git let's

multiple people work on

the same project at the same time.

When pulling new code it will

merge the changes automatically if

possible or will tell us to manually

perform the integrating if there are conflicts.

Here we see the URLs associated with the origin remote.

There are two URLs.

One will be used to fetch data from the remote repository, and

the other one to push data to that remote repo.

Play video starting at 38 seconds and follow transcript0:38

They'll usually point at the same place.

But in some cases, you can have the fetch URL use HTTP for read only access,

and the push URL use HTTPS or SSH for access control.

This is fine as long as the contents of the repo that you read when fetching

are the same that you write to in pushing.

Remote repositories have a name assigned to them, by default,

the assigned name is origin.

This lets us track more than one remote in the same Git directory.

Play video starting at 1 minute 9 seconds and follow transcript1:09

While this is not the typical usage, it can be useful when collaborating with

different teams on projects that are related to each other.

We won't look at how to do that here, but we'll include a link for

more information in the next reading.

If we want to get even more information about our remote,

we can call git remote show origin.

When we called git pull,

we saw that there was also a new remote branch called experimental.

Our friend Blue Kale told us that they've started working on a new feature in

that branch.

Let's check out the output of git remote show origin and

see what it says about that new branch.

Play video starting at 1 minute 38 seconds and follow transcript1:38

We see that there's a new remote branch called experimental,

which we don't have a local branch for yet.

To create a local branch for it, we can run git checkout experimental.

Play video starting at 1 minute 50 seconds and follow transcript1:50

When we checked out the experimental branch,

Git automatically copied the contents of the remote branch into the local branch.

The working tree has been updated to the contents of the experimental branch.

Now we're all set to work on the experimental feature together with

our colleague.

As we called up before,

when using Git to work on

a new feature or a big refactor of some kind,

it's recommended best practice

to create separate branches.

There are many advantages to doing this.

For example, it might take you a while to

finish a new feature and in the meantime,

there could be a critical bug that needs

fixing in the main branch of the code.

By having separate branches,

you can fix the bug in the main branch,

release a new version and then go back to working on

your feature without having to

integrate your code before it's ready.

Another advantage of working in

separate branches is that you could

even release two or more versions out of the same tree.

One being the stable version

and the other being the beta version.

That way, any disruptive changes can be tested on

a few users or computers before they're fully released.

So let's start a new branch to work on

a small refactor of

our code. Do you remember how to do that?

You could create the branch first,

and then check it out or

we can just create it and check it out

with git checkout-b and the new branch name.

**WEEK4**

GitHub tells us it created a fork of this project for us,

which we can commit our changes to.

And if we submit changes to this file, it will create a new branch so

that we can send a pull request.

But what exactly is a fork?

Forking is a way of creating a copy of the given repository so

that it belongs to our user.

In other words, our user will be able to push changes to the forked copy,

even when we can't push changes to the other repo.

When collaborating on projects hosted on GitHub, the typical workflows,

first, create a fork of the repo, and then work on that local fork.

A forked repo is just like a normal repo,

except Github knows which repo it forked from.

So we can eventually merge our changes back into the main repo by creating

a pull request.

A pull request is a commit or series of commits that you send to the owner

of the repository so that they incorporate it into their tree.

This is a typical way of working on GitHub, because in most projects,

only a few people have commit rights to the repo.

But anybody can suggest patches, bug fixes or even new features by sending pull

requests that people with commit access can then apply.

Typically, the owners of the repo will review the changes before merging them.

Checking that they match the development guidelines for the project and

that the license is valid and so on.

Now, to push the change to our forked repo,

we need to create the corresponding remote branch.

Do you remember the command for that?

Yes. It's git push - u origin add -readme.

As we called out, before creating a pull request,

it's always important to check

that the code will merge successfully.

GitHub tells us that our change can be

automatically merged, which is great news.

If this wasn't the case,

we'd need to rebase our change against

the current branch of the original repo

so that it could be merged.

Remember that forcing pushes is fine for

pull request branches because

nobody else should have cloned it.

But this isn't something that we want to do with

public repos