GIT and Github

Git is a Version Control System (VCS). VCS is software that tracks and manage changes to files over time.

Installing GIT in MacOS

command to install git

```
brew install git
```

if already install, to update

```
brew update git
```

Configuring GIT

to check GIT version

```
git --version
```

Configuring the user name that git will associate with your work

```
git config --global user.name "your name"
```

to check git that you had git user name already use this command

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

Configuring Git user email. Git email address to be match with Github account

```
git config --global user.email sakibdalal73@gmail.com
```

to check git has email address already

```
git config --global user.email
```

Installing a GUI for GIT

Installing GitKraken GUI

Thank You for Downloading GitKraken

Basic UNIX Commands - Terminal

list the contents of directory , show hidden files

```
ls
ls -a
```

open finder

```
open .
```

for windows

```
start .
```

print working directory - printout current path

```
pwd
```

change directory, come back to recant dir, main dir

```
cd dir_name
cd ..
cd ~
```

make new file’s

```
touch filename.ext
touch file1.txt file2.txt
```

make new folder or directory

```
mkdir dir-name
```

deleting files

```
rm file-name
```

deleting directory

```
rm -rf dir-name
rmdir dir-name
rmdir -r dir-name
```

clear terminal

```
command + K
clear
```

Git Repository

Repository- a git repo is a workspace which tracks and manages files with a folder

git status command gives information on the current status of a git repository and its contents

```
git status
```

create a git repository

```
git init
```

```
git status
ls -a
```

Committing

git add command is use ti stage changes to be committed, is a way of telling git “please insert this change in our next commit.

```
git status
touch mytext.txt
git status
nano mytext.txt
touch mytext2.txt
git status
```

```
# Output
(base) sakibdalal@Sakibs-MacBook-Pro git % git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    mytext.txt
    mytext2.txt

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

git add command is use to add specific files to the staging area. Separate files with spaces to add multiple at once

Git add Command

```
git add file1 file2
```

```
git add mytext.txt
git status
```

```
# Output:
(base) sakibdalal@Sakibs-MacBook-Pro git % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   mytext.txt
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    mytext2.txt
```

```
git add mytext2.txt
```

```
# Output
(base) sakibdalal@Sakibs-MacBook-Pro git % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   mytext.txt
    new file:   mytext2.txt
```

Git Commit Command

we use the git commit command to actually commit changes from the staging area. running git commit will commit all staged changes.

```
git commit -m "my message"
```

```
git commit -m "creating text files - stage 1"

# Output
[master (root-commit) 58a34ad] creating text files - stage 1
 2 files changed, 2 insertions(+)
 create mode 100644 mytext.txt
 create mode 100644 mytext2.txt

git status

# Output
On branch master
nothing to commit, working tree clean
```

```
touch chapter01.txt
git status
```

```
# Output
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    chapter01.txt

nothing added to commit but untracked files present (use "git add" t

nano mytext.txt
# make some changes

git status

# Output
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directo
    modified:   mytext.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    chapter01.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
git add mytext.txt chapter01.txt

git status

# Output
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   chapter01.txt
    modified:   mytext.txt
```

```
git commit -m "begin work on chapter 01"

# Output
```

```
[master 9c54509] begin work on chapter 01
 2 files changed, 1 insertion(+)
 create mode 100644 chapter01.txt

git status
```

Git Log Command

git log command the stages of commit done.

```
git log

# Output
commit 9c54509ec43322452f0f4a72b34ad8e48063c2ee (HEAD -> master)
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Fri Dec 8 13:02:45 2023 +0530

    begin work on chapter 01

commit 58a34ad13bf190c18c440f6fad737be8b7e474bf # COMMIT HASH
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Fri Dec 8 12:52:02 2023 +0530

    creating text files - stage 1
```

Git add all Command

it is use to add all stages changed

```
git add .
git status
```

Task :

Committing Basics Exercise

```
mkdir Shopping
cd Shopping
git init
touch yard.txt groceries.txt
git status
git add .
git status
```

```
git commit -m "create yard and groceries list"
git status
git log
```

Atomic Commits

When possible, a commit should encompass a single feature, change, or fix. In other words, try to keep each commit focused on a single thing.

This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review.

```
git add individual_file_or_directory_to_add
git commit -m "commit name"
git status
git log
```

Breaking down the project into individual commits can help in undoing the changes individually done before.

Write commits in Present tense

Default editor to commit in MAC

MacOS uses Vim as a default editor to commit changes

Configure VS Code to commit changes

> Git - Setup and Config
>
> This book is available in English.
>
> https://git-scm.com/book/en/v2/Appendix-C:-Git-Commands-Setup-and-Config

```
git config --global core.editor "code --wait"
```

```
git add .
git commit  # will open visual studio code to add commit descriptior
```

```
this is my new commit in vscode

added new stuf in chapter01.txt
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
#
# On branch master
# Changes to be committed:
#   modified:   chapter01.txt
#
```

close the vs commit file and commit will be executed

Git log documents

Git - git-log Documentation (git-scm.com)

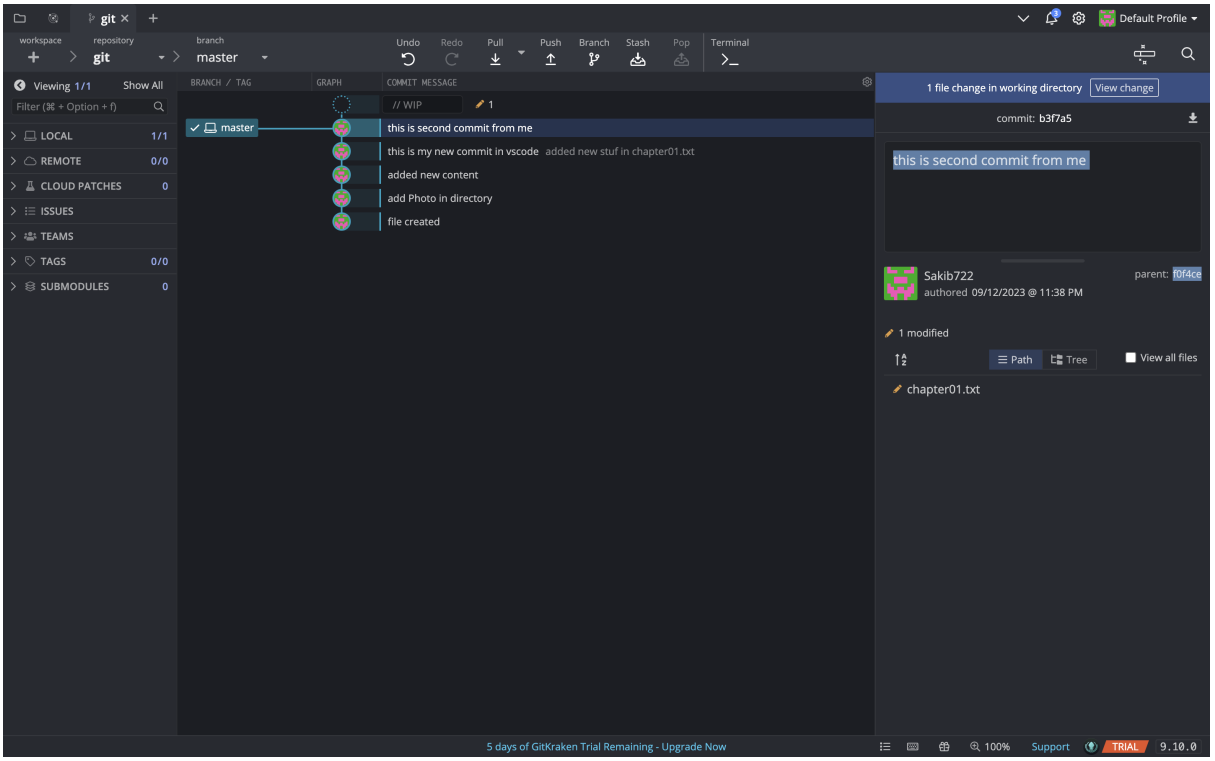
we need commit formatting from the document

```
git log --pretty=oneline
git log --abbrev-commit

git log --pretty=oneline --abbrec-commit
```

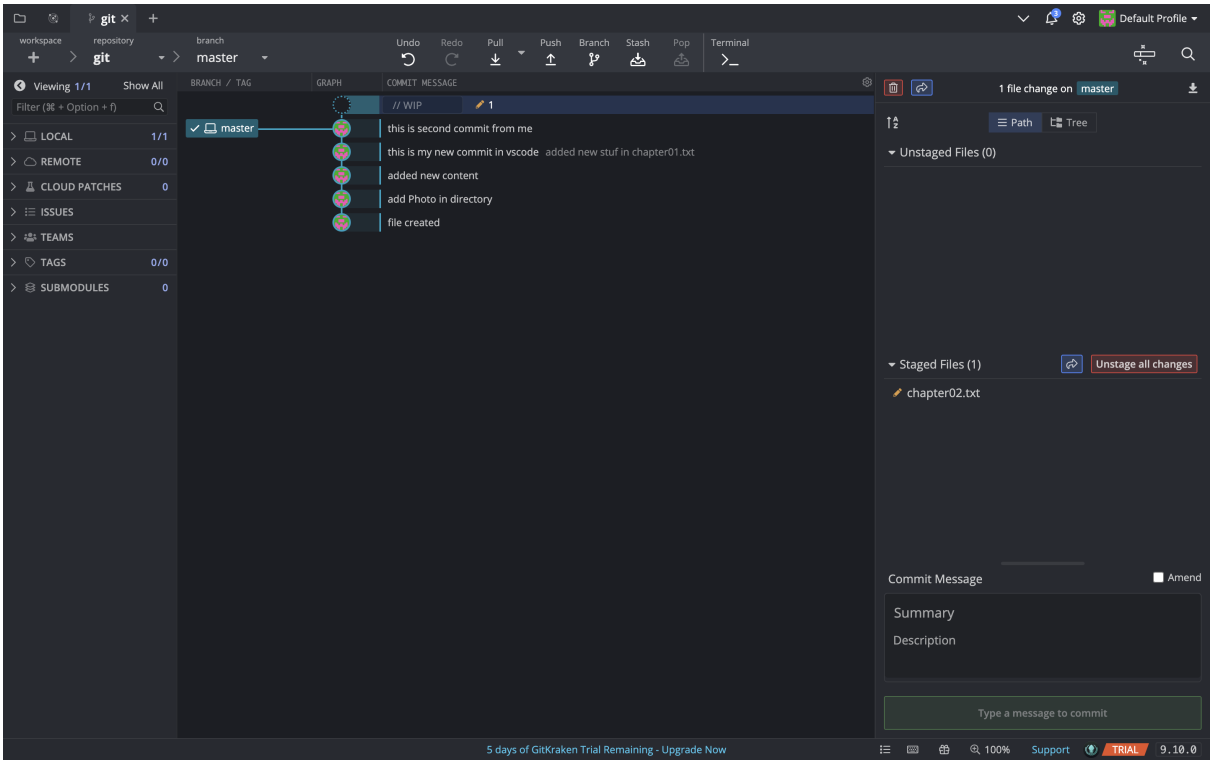
Remove extra lines in commit

```
git log --oneline

# Output
b3f7a59 (HEAD -> master) this is second commit from me
f0f4ce0 this is my new commit in vscode
62c8feb added new content
4711594 add Photo in directory
9b1f239 file created
```

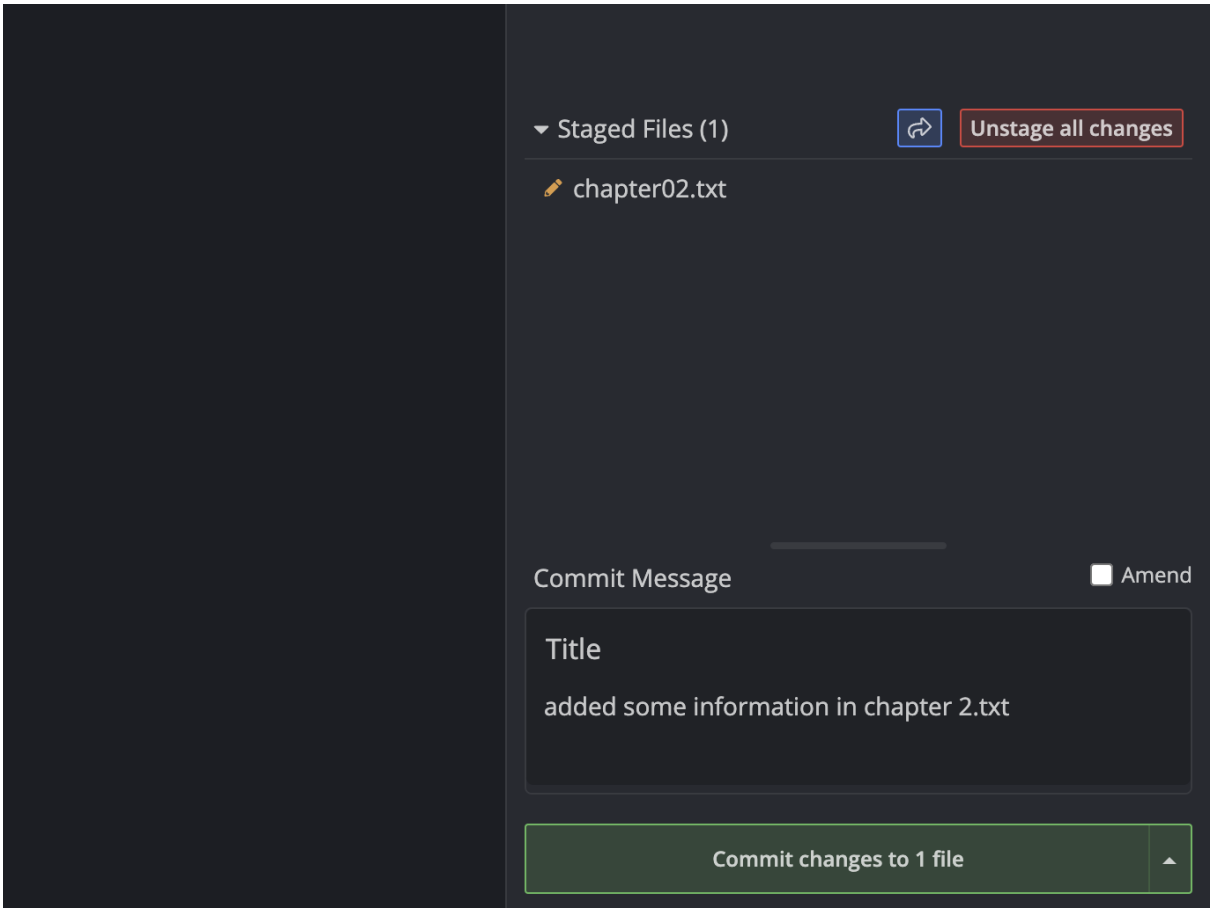
Commit using GUI - Git Kraken



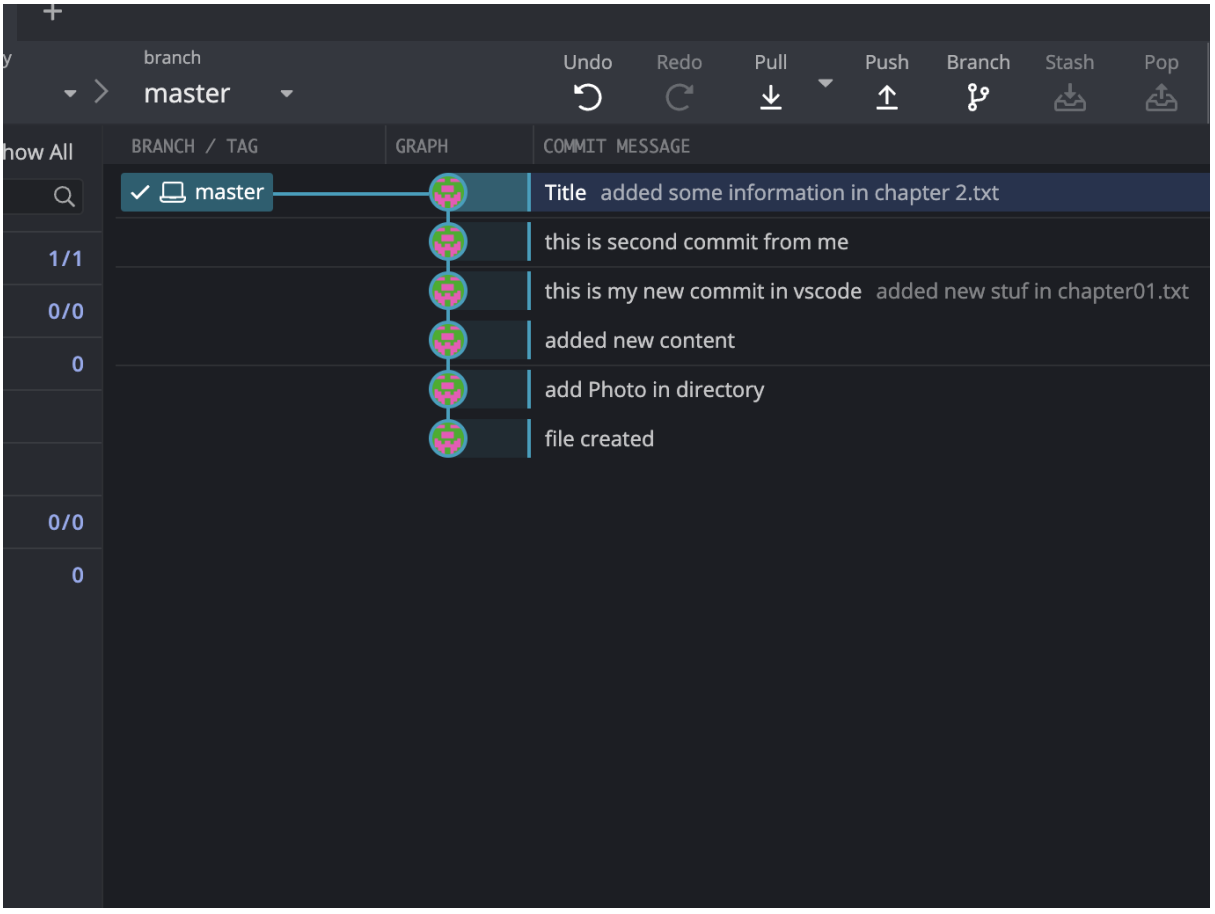
View changes → stage file



Add commit Description and information and click on commit



new commit executed



Amending Commits

Suppose you just made a commit and then realised you forgot to include a file! Or, maybe you made a typo in the commit message that you want to correct.

Rather than making a brand new separate commit, you can “redo” the previous commit using the —amend option. (only for the currently done commit)

Syntax:

```
git commit -m "some commit"
git add forgotten_file
git commit --amend
```

example:

```
nano chapter01.txt
touch chapter04.txt

git add chapter01.txt
git commit -m "chapter01 edited"
```

```
git add chapter04.txt
git commit --amend # Opens VS code to edit commits
```

Ignoring Files

We can tell Git which file and directories to ignore in given repository, using a .gitignore file. This is useful for files you know you NEVER want to commit, including:

- Secrets, API keys, credentials, etc
- Operating System files (.DS_Store on MAC)
- log files
- Dependencies & packages

.gitignore

Create a file called .gitignore in the root of a repository. Inside the file, we can write patterns to tell Git which file and folders to ignore:

- .DS_Store will ignore file name .DS_Store
- folderName/ will ignore an entire directory
- *.log will ignore any files with the .log extension

we can find a .gitignore file in githubs repositories

```
mkdir gitIgnore
cd gitIgnore
git init
git status

touch main.txt
touch secrets_file.txt

git status

git add main.txt
git commit -m "start the project"
git status
```

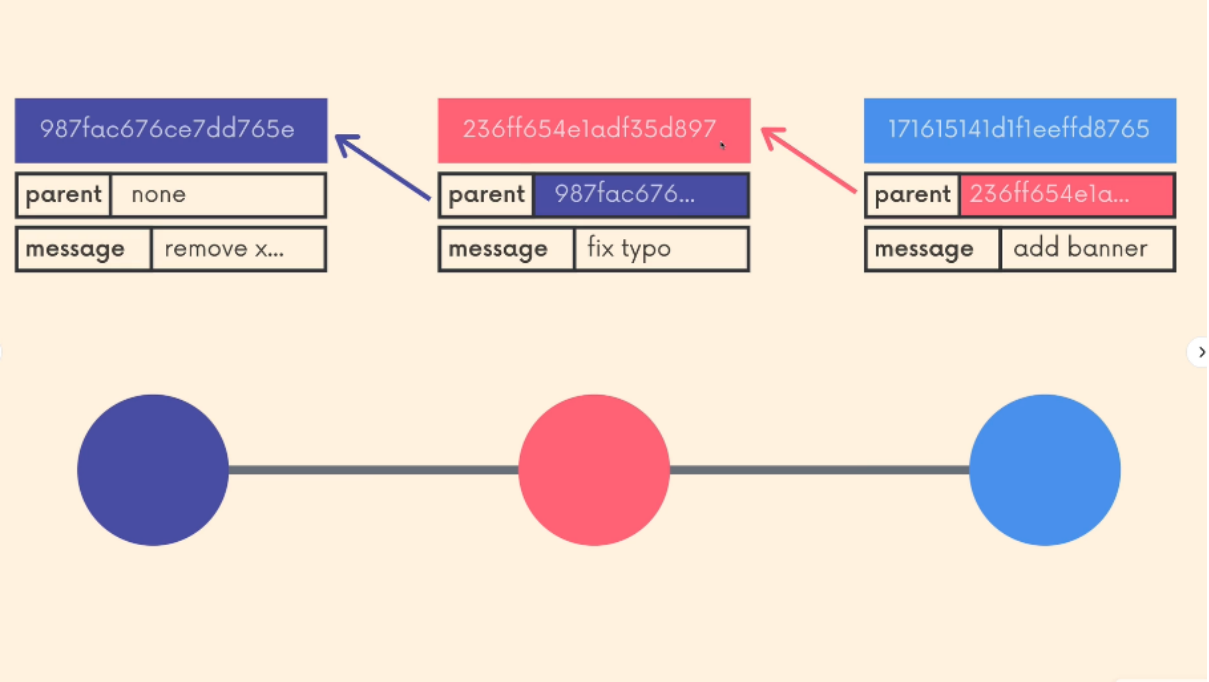
```
touch .gitignore
nano .gitignore  # add the file name to be ignored by git

git status
```

```
git add.
git commit -m "added .gitignore file"
```

git has a website which has the .gitignore file to be included in the git

> https://www.toptal.com/developers/gitignore

Branching



Contexts

On large projects, we often work in multiple contexts:

1. You're working on 2 different color scheme variations for your website at the same time, unsure of which you like best
2. You're also trying to fix a horrible bug, but it's proving tough to solve. You need to really hunt around and toggle some code on and off to figure it out.
3. A teammate is also working on adding a new chat widget to present at the next meeting. It's unclear if your company will end up using it.
4. Another coworker is updating the search bar autocomplete.
5. Another developer is doing an experimental radical design overhaul of the entire layout to present next month.

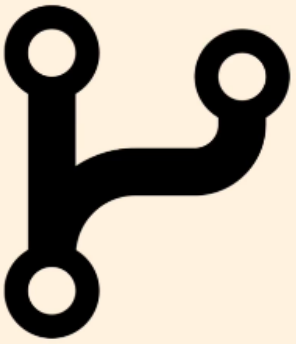
Branches

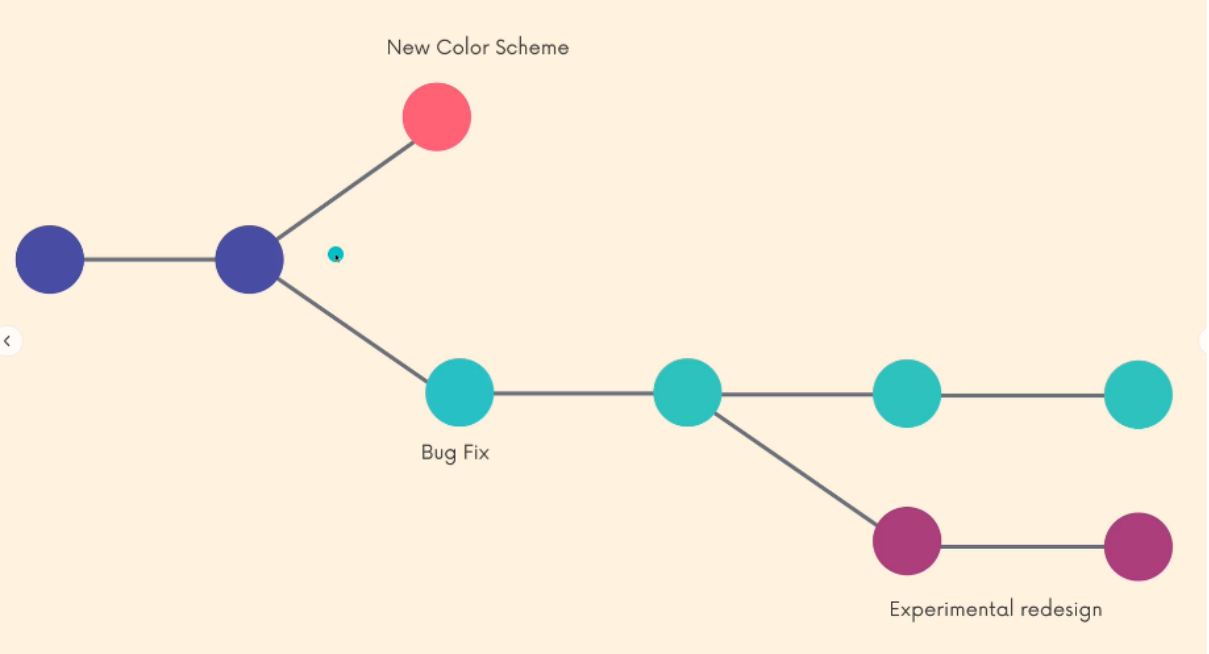
Branches are an essential part of Git!

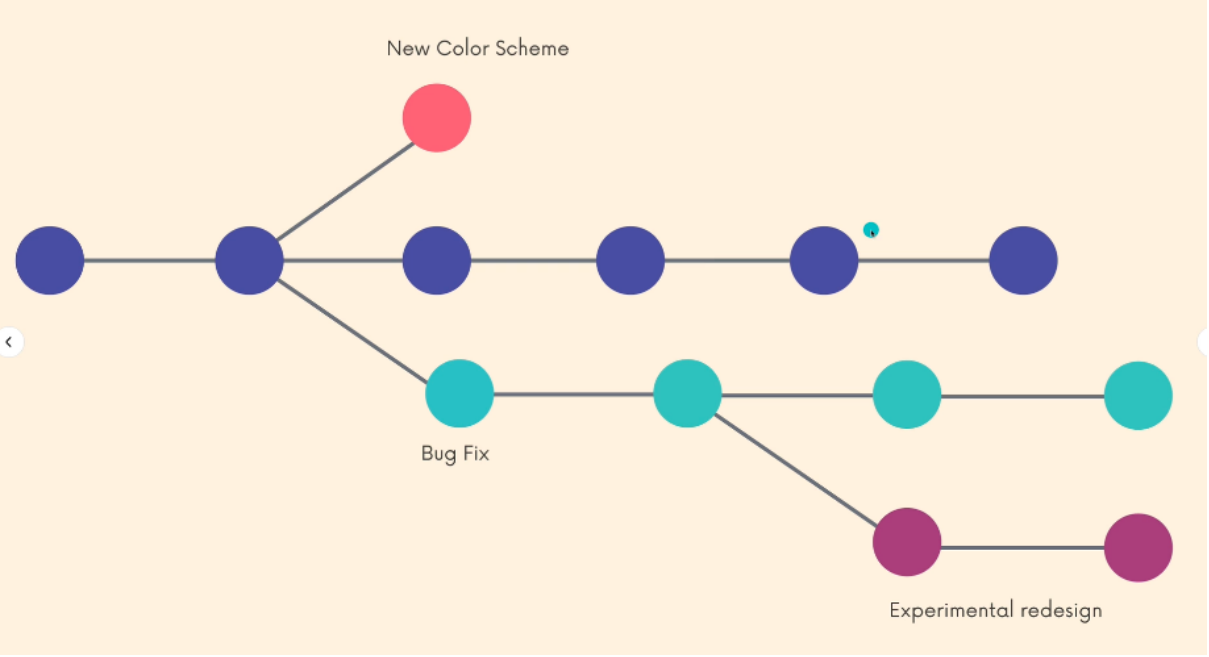
Think of branches as alternative timelines for a project.

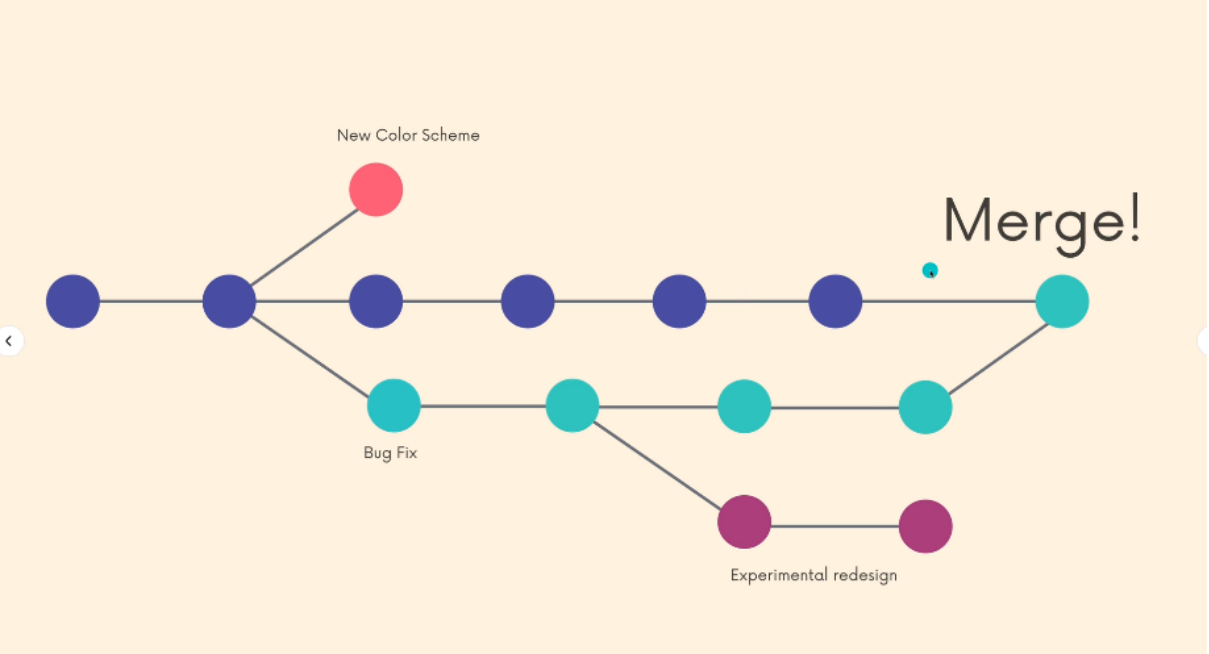
They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel.

If we make changes on one branch, they do not impact the other branches (unless we merge the changes)




New Color Scheme
Bug Fix
Experimental redesign


New Color Scheme
Bug Fix
Experimental redesign



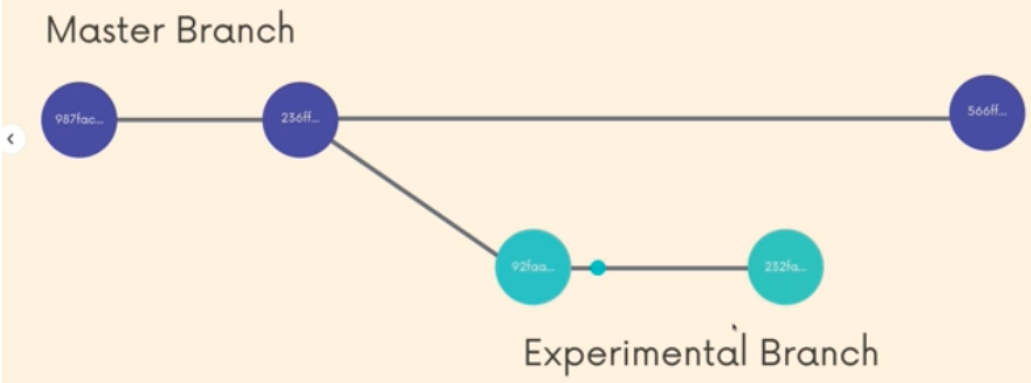
Master Brach in Git

git status
On branch master

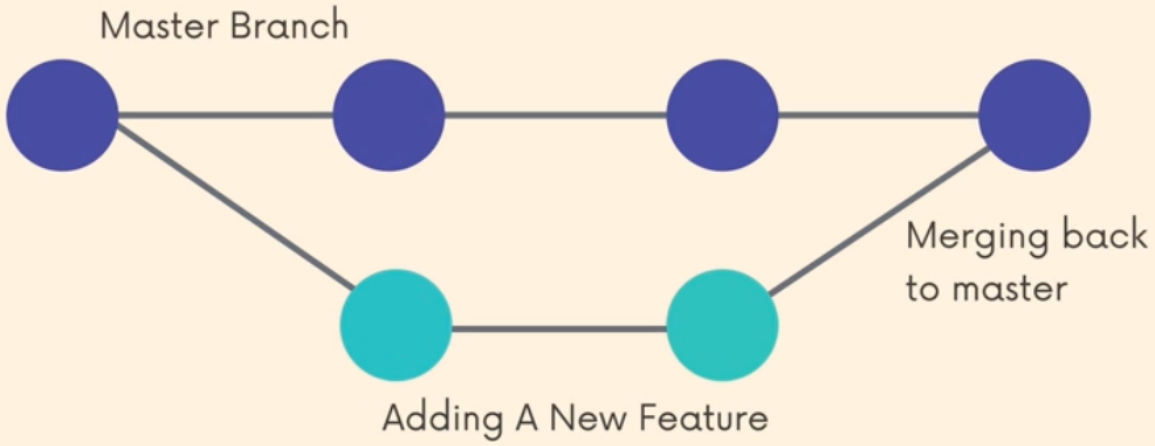
In git, we are always working on a branch “The default branch name is master

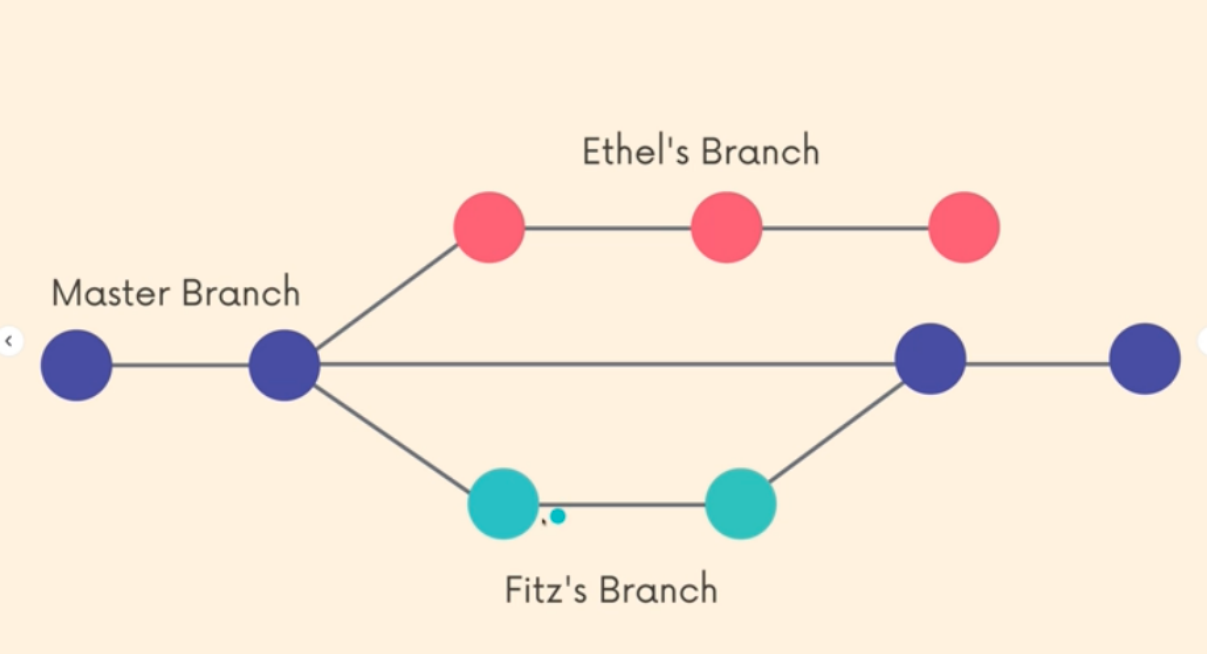
It doesn’t do anything special or have fancy powers. It’s just like any other branch.

Branching



A Common Workflow





HEAD → master

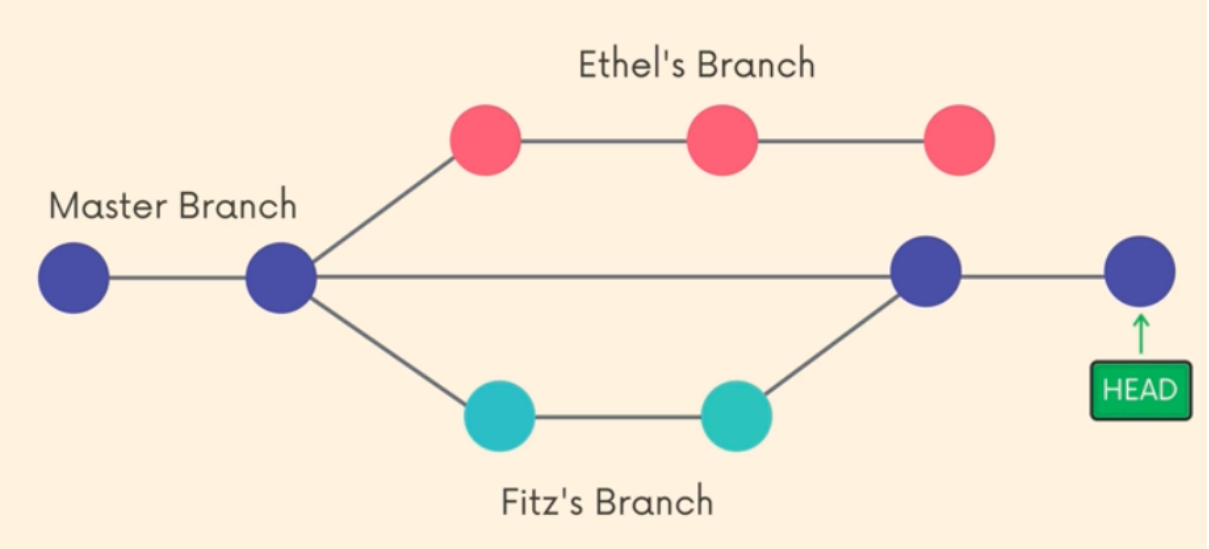
```
commit 40ff1edb998e1733978e6c16a93a2061d09cc646 (HEAD -> master)
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Sun Dec 10 15:42:29 2023 +0530

    edited chapter01.txt and added chapter04.txt

commit e5c382df4b332e4f37c9a463f76261157d11da79
```

HEAD

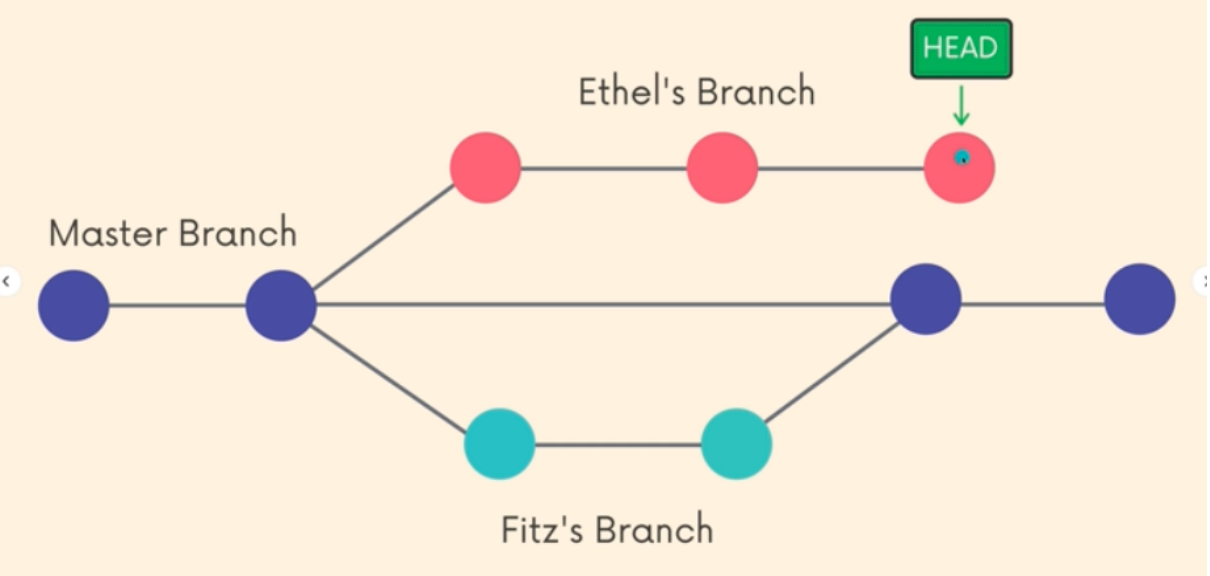
We'll often come across the term **HEAD** in Git.

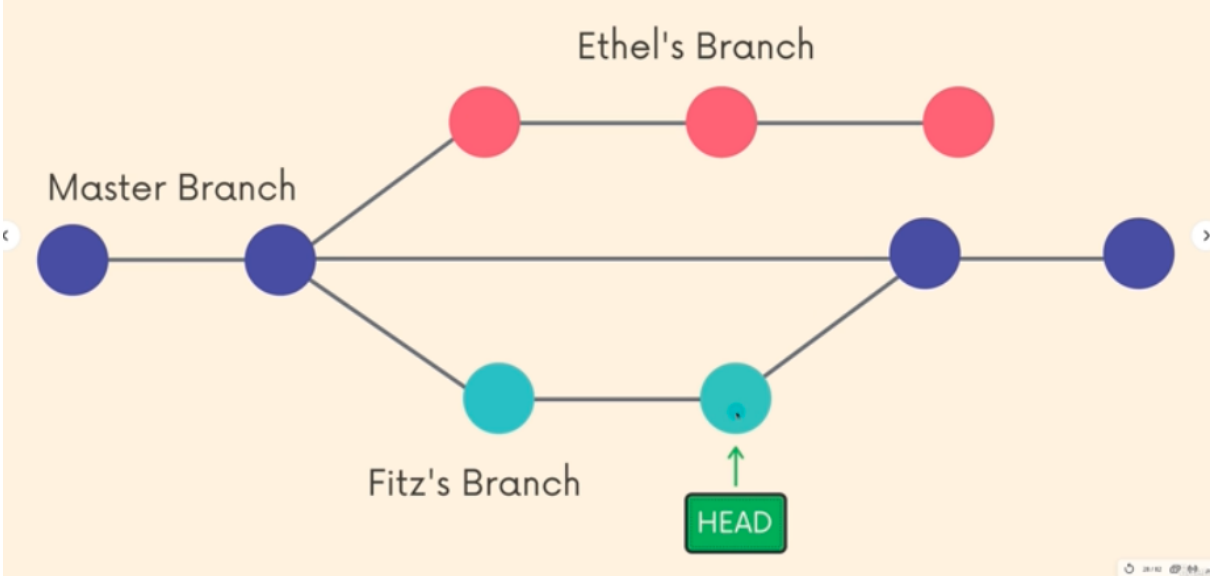
HEAD is simply a pointer that refers to the current "location" in your repository. It points to a particular branch reference.

So far, HEAD always points to the latest commit you made on the master branch, but soon we'll see that we can move around and HEAD will change!



head refers to last commit on master branch





Viewing Branches

Use git branch to view your existing branches. this default brach in every git repo is master, through you can configure this

look for the * which indicates the branch you are currently on.

```
git branch

# Output
* master
```

```
git branch -v
```

Creating Branches

Use git branch <branch-name> to make a new branch based upon the current HEAD. This just creates the branch. It does not switch you to that branch (the HEAD stays the same)

Syntax:

```
git branch <branch-name>
```


```
mkdir Branching-git
cd Branching-git
git init
git status

touch file_A.txt
nano file_A.txt

git add .
git commit -m "add file A"

git log
nano file_A.txt
git add .
git commit -m "edited file A"
git log --oneline

git branch
```

```
git branch A_branch. # new branch created as A_branch

git branch
```

```
git log
```

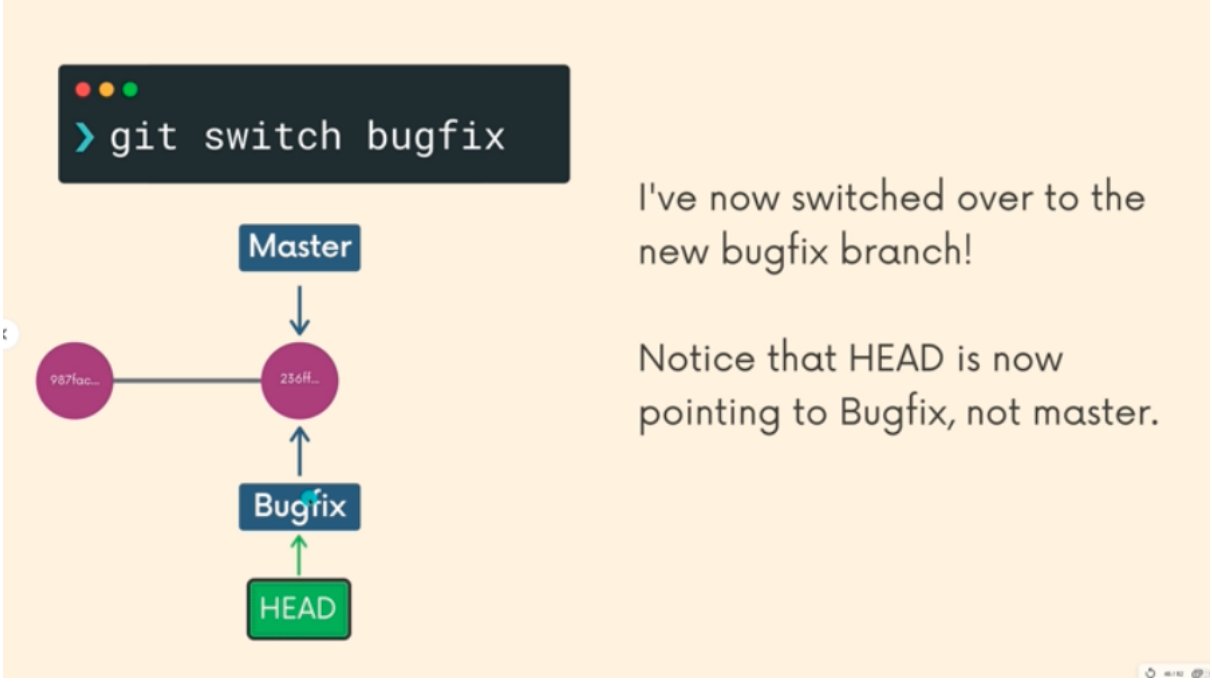
```
commit a98fc8b920c23f7ffef91ff80840a43aa31ee772 (HEAD -> master, A_branch)
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Sun Dec 10 17:51:22 2023 +0530

    edited file A

commit e30b7c85eabc56ed13c9094868124c8223ad8124
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Sun Dec 10 17:48:52 2023 +0530

    add and create file A
(END)
```

Switching Branch

Once you have created a new branch, use git switch <branch-name> to switch to it.

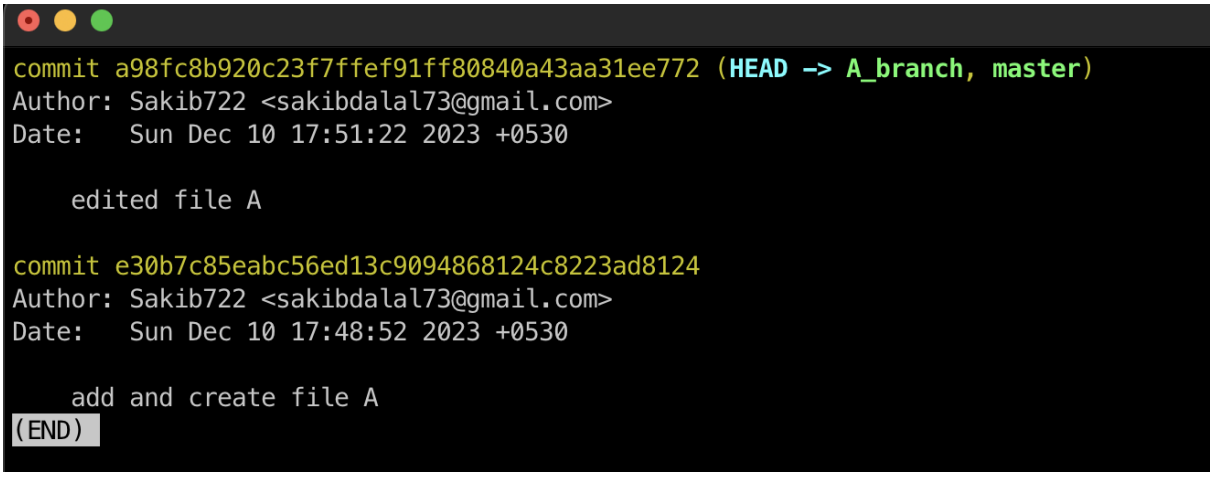
Syntax:

```
git switch <branch-name>
```

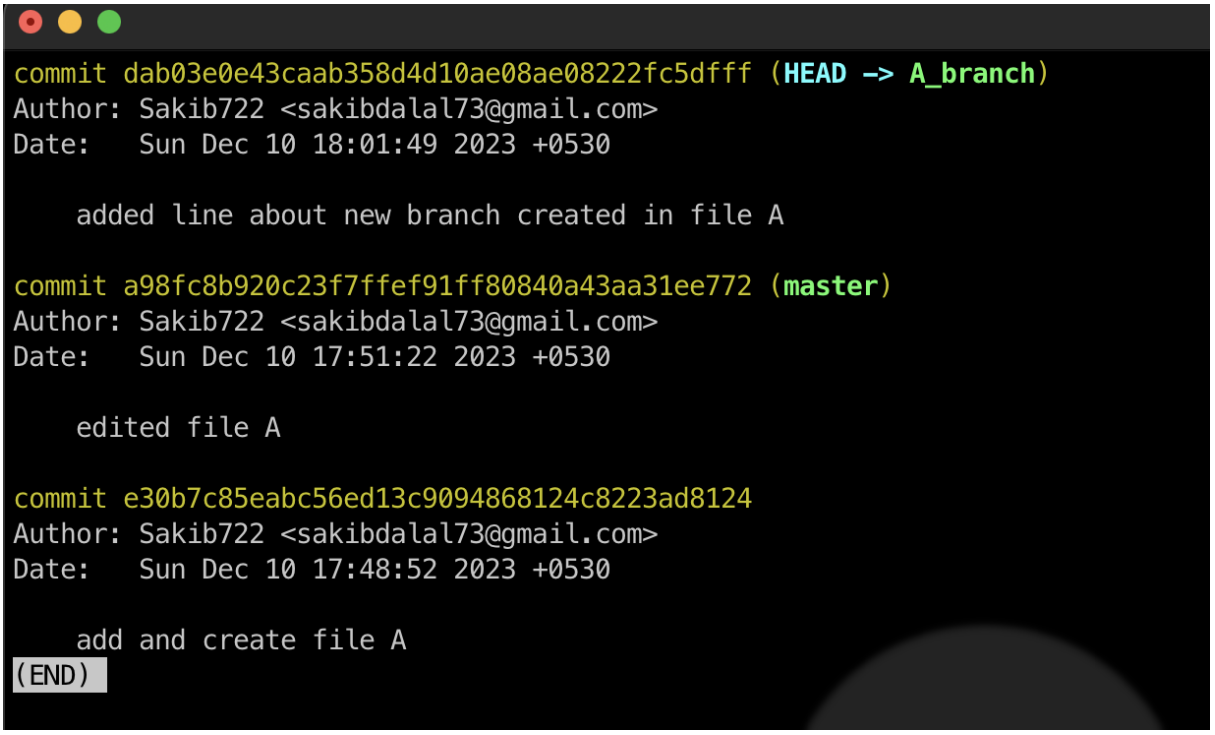



```
git switch A_branch

# Output
Switched to branch 'A_branch'
```

```
git status
git log
```



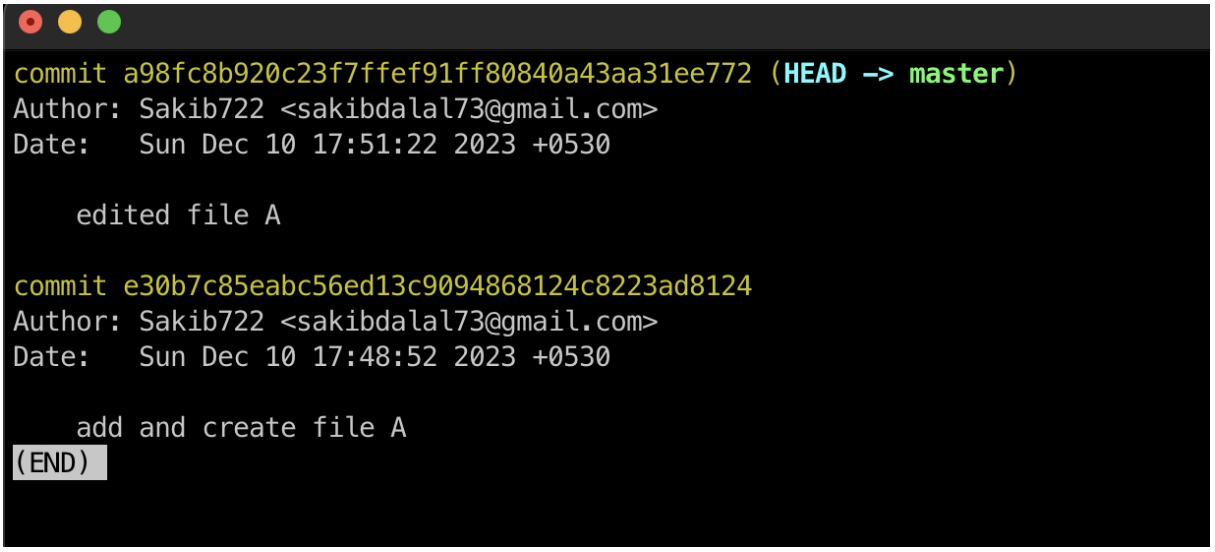

```
nano A_file.txt
git add .
git commit -m "added line that we created new branch in file A"
git log
```



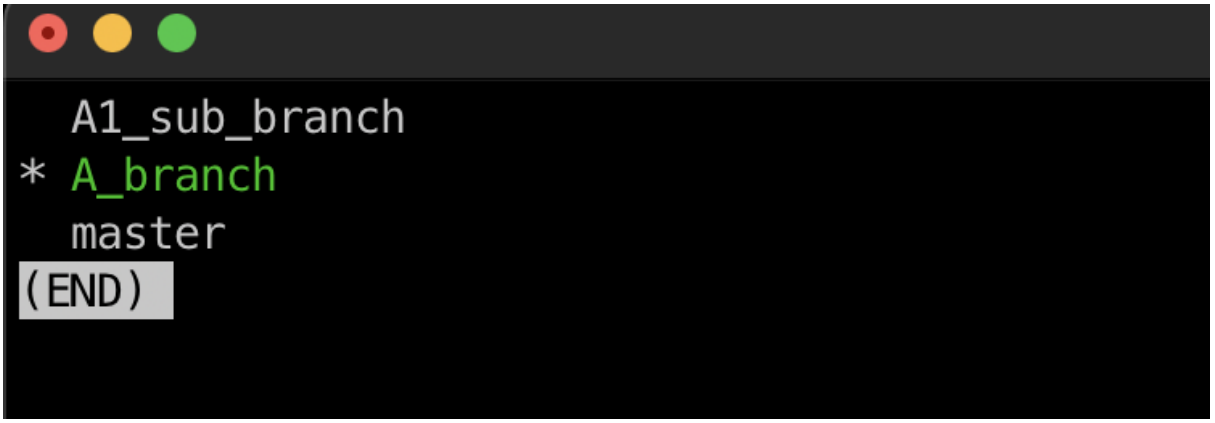

```
git switch master
```

```
nano file_A.txt
# no changes on this main brach will be done which was done on the A
```

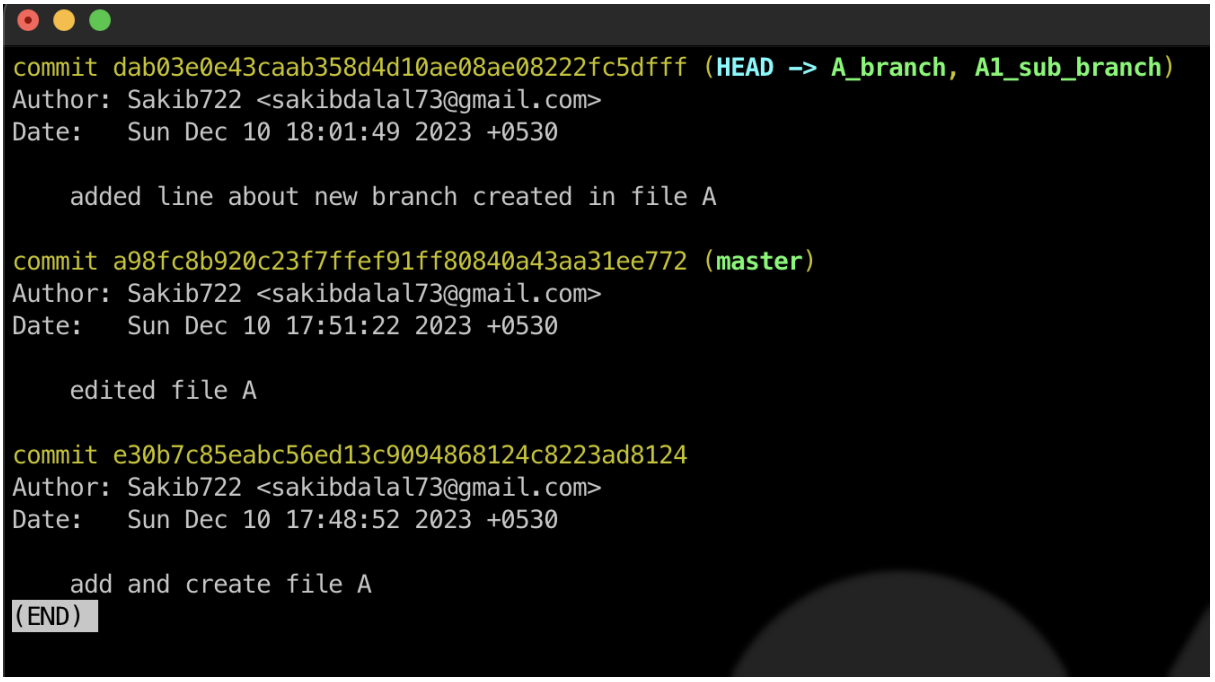
```
git log
```



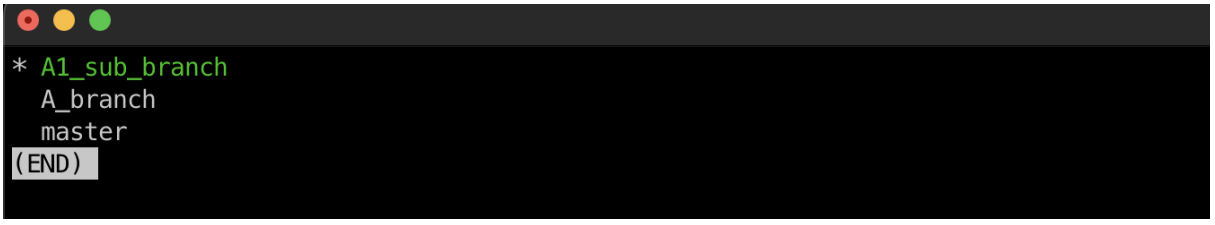

```
git switch A_branch
git log
git branch A1_sub_branch
git branch
```




```
git log
```




```
git switch
git branch
git log
```



```
commit dab03e0e43caab358d4d10ae08ae08222fc5dfff (HEAD -> A1_sub_branch, A_branch)
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Sun Dec 10 18:01:49 2023 +0530

    added line about new branch created in file A

commit a98fc8b920c23f7ffef91ff80840a43aa31ee772 (master)
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Sun Dec 10 17:51:22 2023 +0530

    edited file A

commit e30b7c85eabc56ed13c9094868124c8223ad8124
Author: Sakib722 <sakibdalal73@gmail.com>
Date:   Sun Dec 10 17:48:52 2023 +0530

    add and create file A
(END)
```

Another way of switching


```
git checkout <branch-name>
```

Creating and Switching the branch in Git

Use git switch with -c flag to create a new branch AND switch to it all in one go

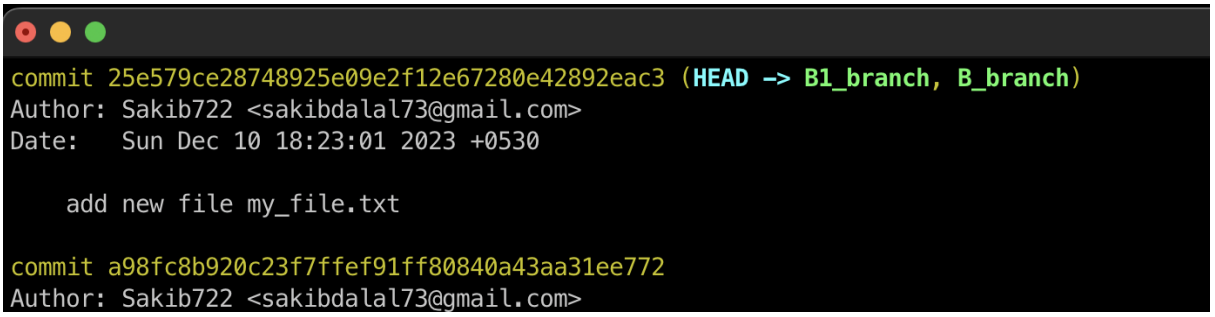
Remember -c as short for “create”

Syntax:

```
git switch -c <branch-name>
```

example:

```
git branch main
git switch -c B_branch
git switch -c B1_branch

git log
git branch
```



Delete and Rename Branches

Delete a existing branch

git switch -c delete_me

Syntax:

```
git branch -d <branch-name>
```

Note: don't be at branch which you want's to delete, change the branch to delete.

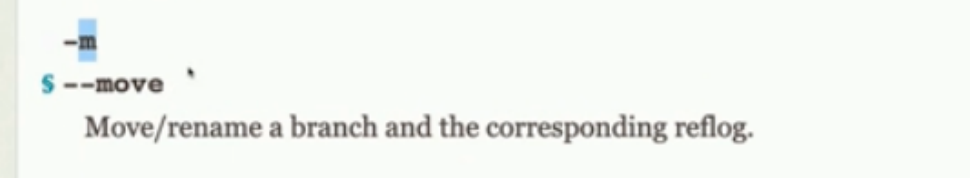
Example:

```
git switch master
git switch -c delete_me # Created delete_me branch
git switch master

git branch -d delete_me # Deleted delete_me branch # -d stand for de
```

Note: For forcefully delete the branch use

```
git branch -D <branch-name>
```

Rename the existing branch



Note: to rename the branch we need to be on the branch which we wanted to rename.

```
git branch -m <new_name_of_branch>
```

example:

```
git switch master
git switch -c new_branch
# rename the new_branch to rename branch

git branch -m rename
```

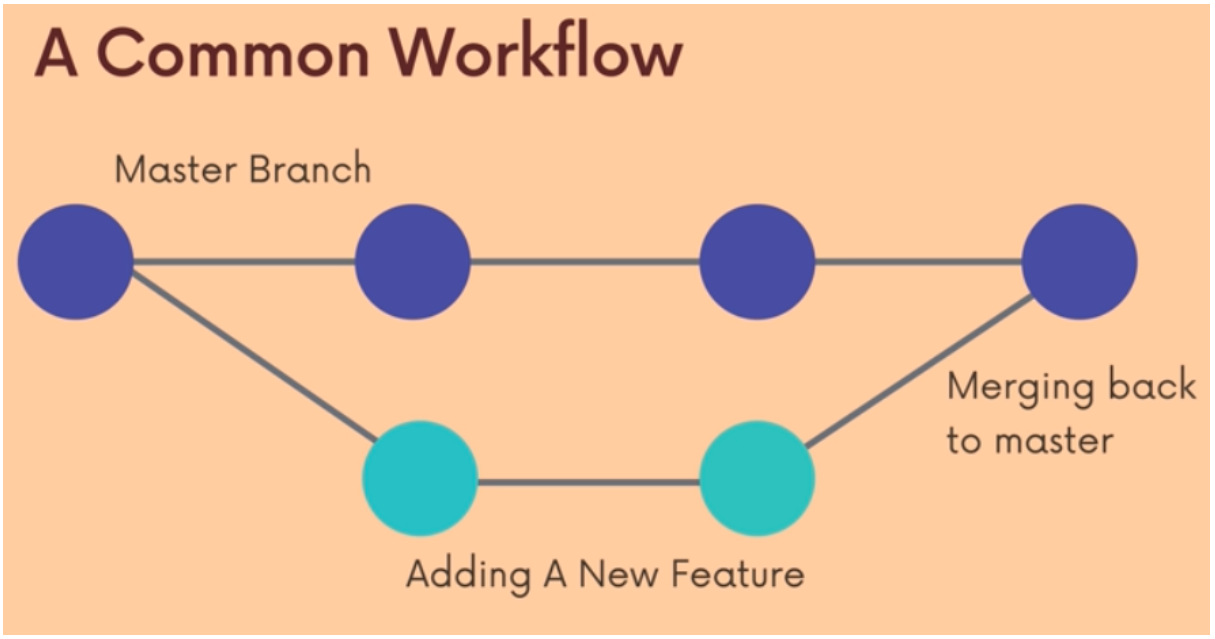
Branching Exercise

Merging Branches

Branching makes it super easy to work within self-contained contexts, but often we want to incorporate changes from one branch into another!

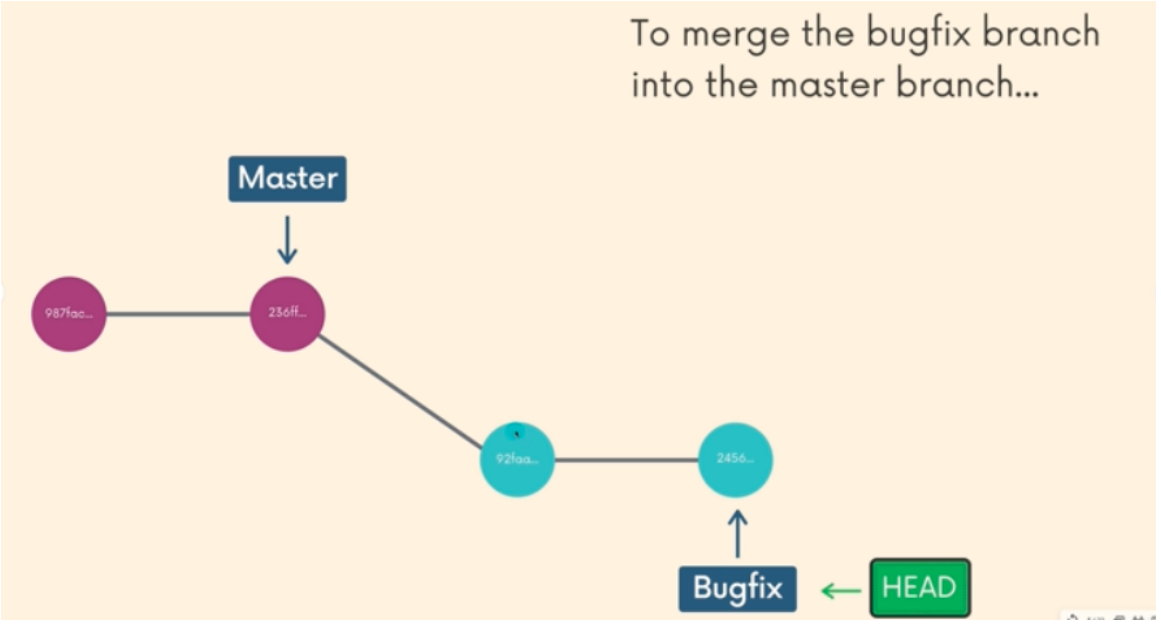
we can do this using the git merge command

```
git merge <branch name>
```



Remember these two merging concepts:

- We merge branches, not specific commits
- We always merge to the current HEAD branch



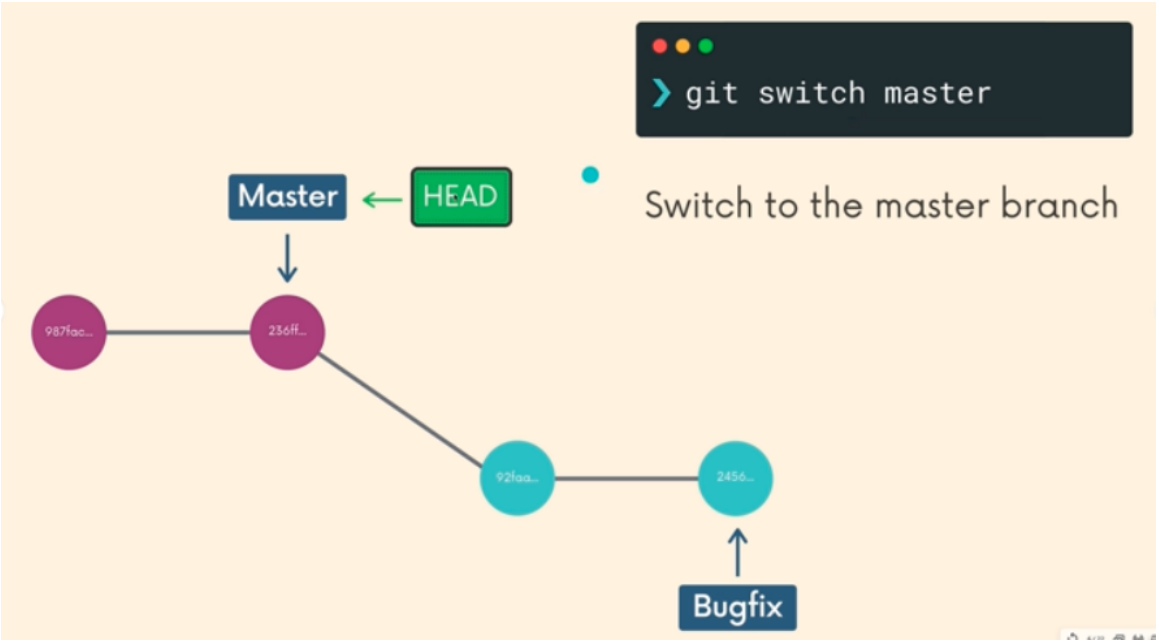
Merging Made Easy

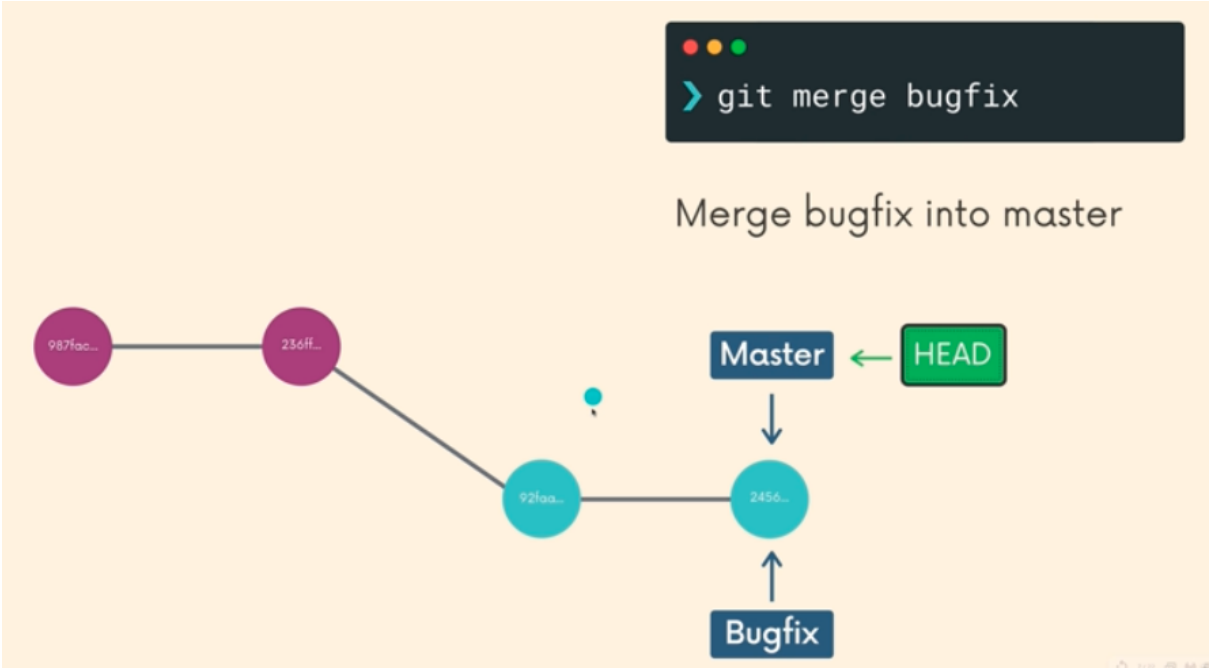
To merge, follow these basic steps:

1. Switch to or checkout the branch you want to merge the changes into (the receiving branch)
2. Use the **git merge** command to merge changes from a specific branch into the current branch.

To merge the bugfix branch into master...

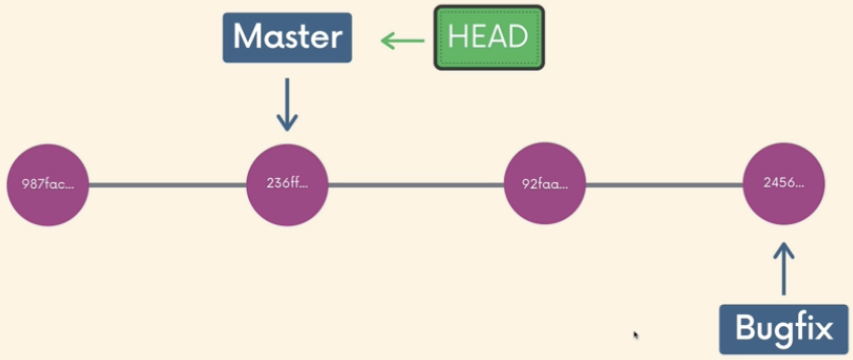
```
❯ git switch master
❯ git merge bugfix
```





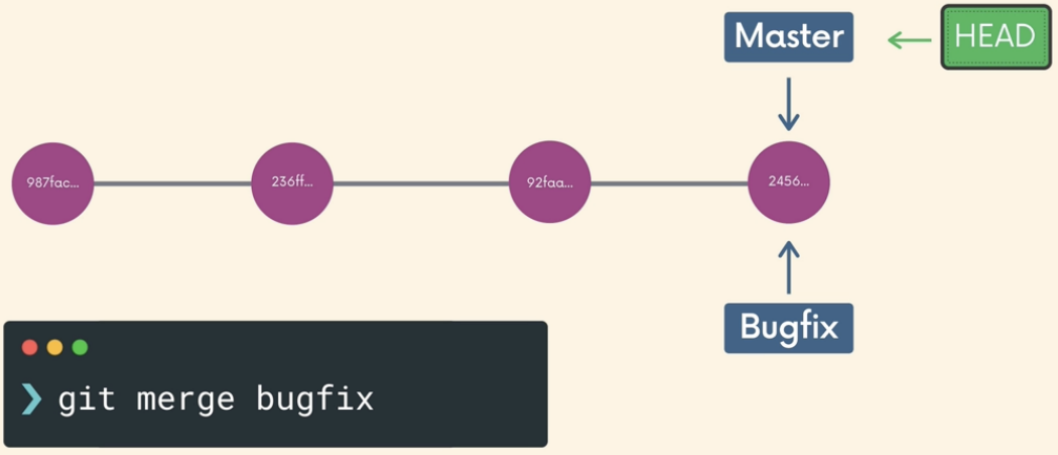
This is what it really looks like

Remember, branches are just defined by a branch pointer

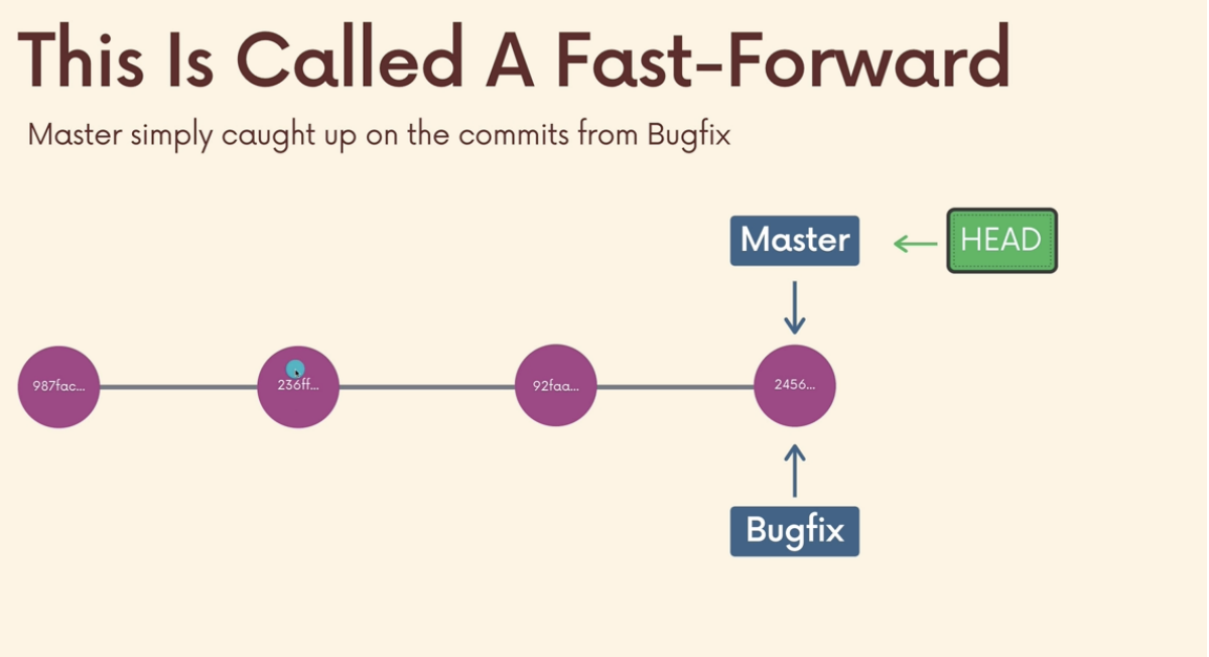


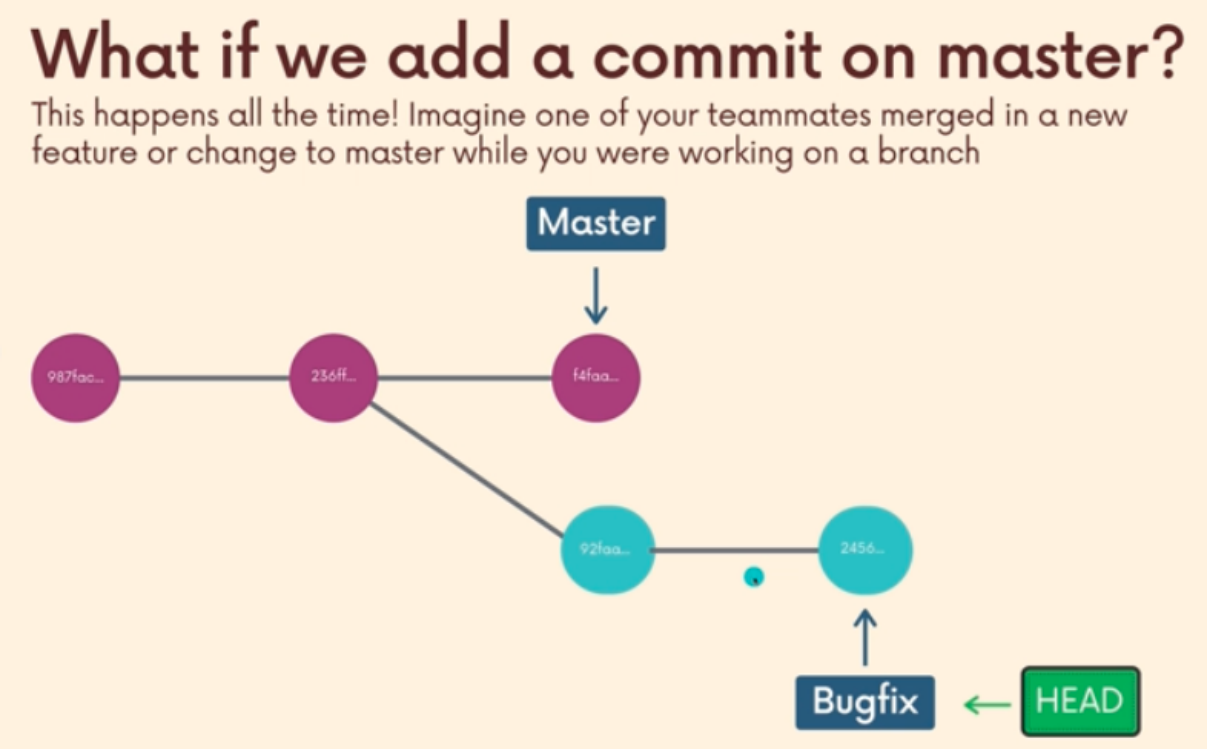
This is what it really looks like

Master & Bugfix now point to the same commit, until they diverge again

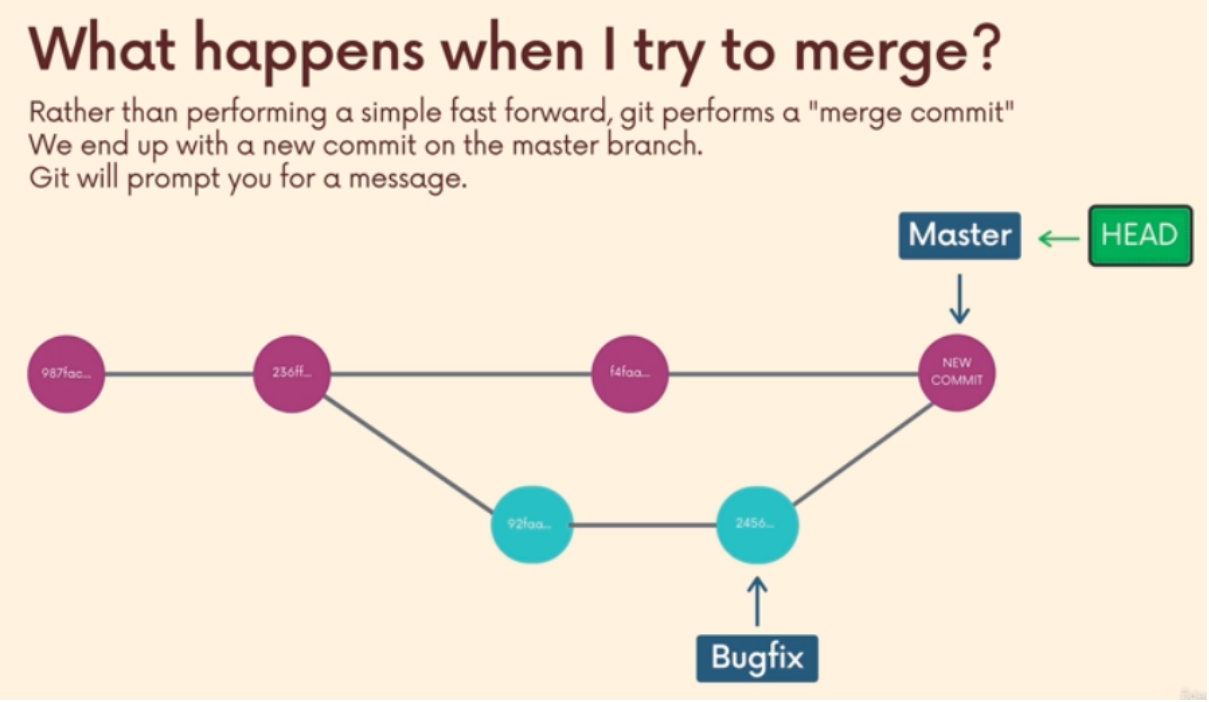



```
❯ git merge bugfix
```

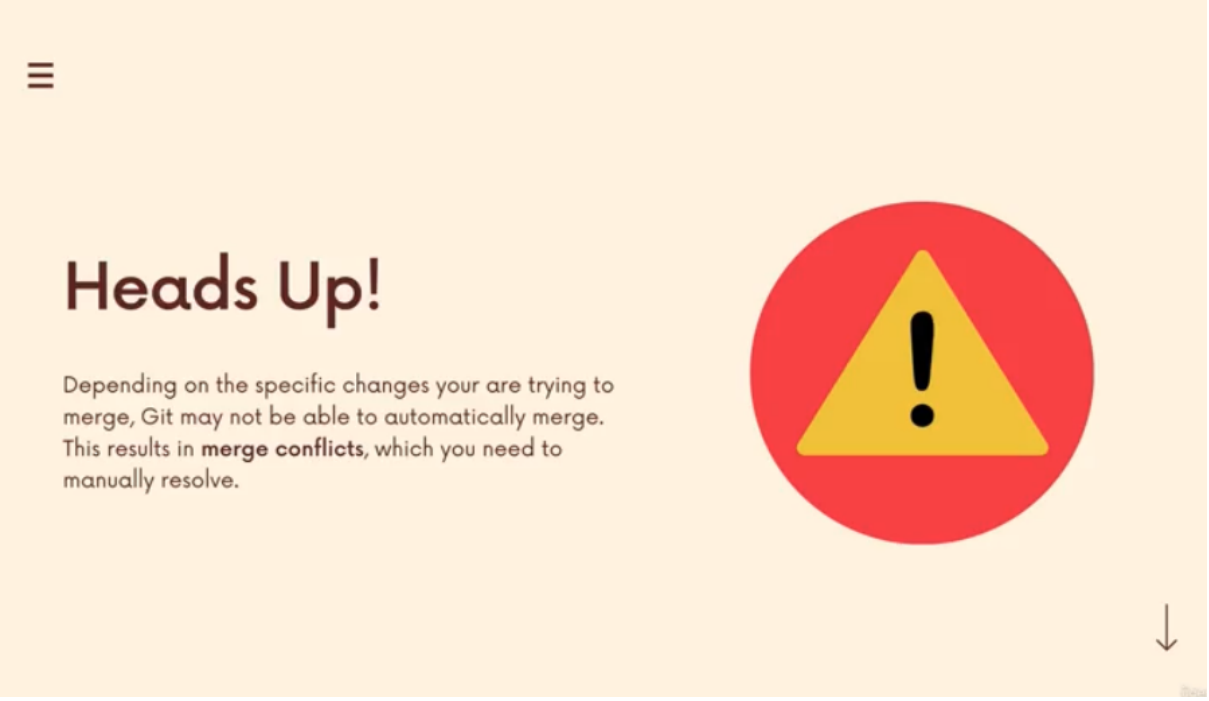




this create a conflict in files.



Merge conflict



```
› CONFLICT (content): Merge conflict in blah.txt
  Automatic merge failed; fix conflicts and then commit the result.
```

```
<<<<<<< HEAD
I have 2 cats
I also have chickens
=======
I used to have a dog :(
>>>>>>> bug-fix
```

WHAT THE...

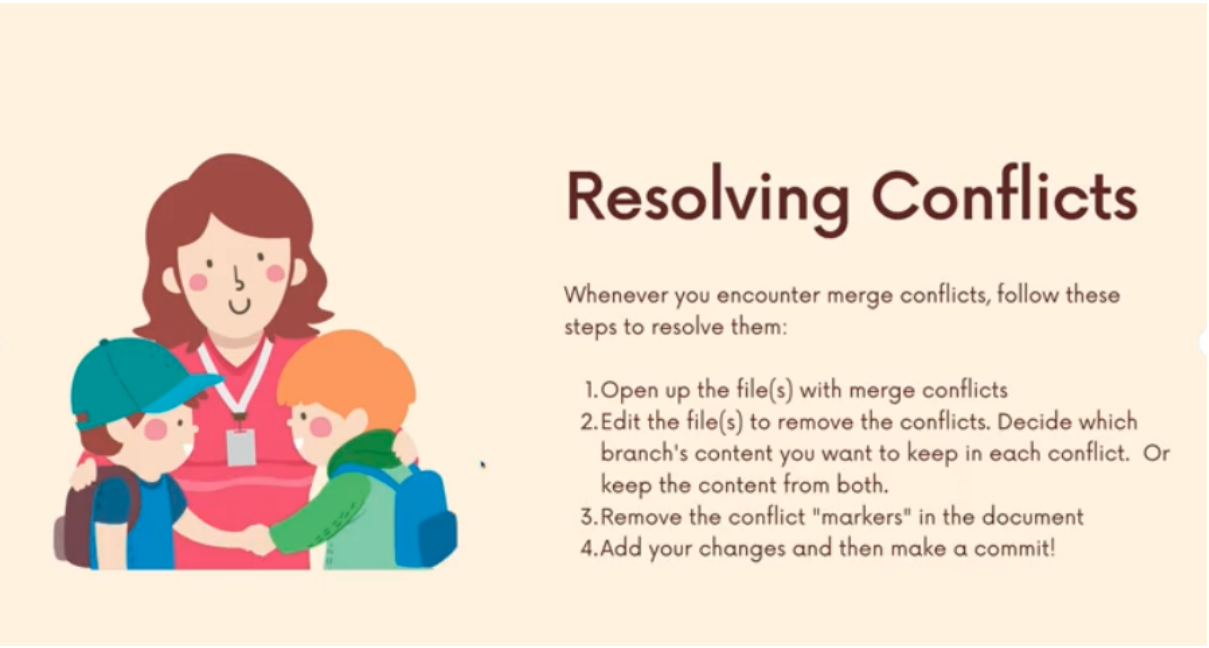
When you encounter a merge conflict, Git warns you in the console that it could not automatically merge.

It also changes the contents of your files to indicate the conflicts that it wants you to resolve.

```
<<<<<<< HEAD
I have 2 cats
I also have chickens
=======
I used to have a dog :(
>>>>>>> bug-fix
```

Conflict Markers

The content from your current **HEAD** (the branch you are trying to merge content into) is displayed **between the** `<<<<<<< HEAD` **and** `=======`



Git Merging Exercise

Git Diff

We can use the git diff command to view changes between commits, branches, files, our working directory, and more.

We often use git diff alongside commands like git status and git log, to get a better picture of a repository and hoe it has changed over time.

Without additional options, git diff lists all the changes in working directory that are NOT staged for the next commit.

```
git diff
```

Compares staging area and working directory

Example

```
mkdir gitfile
cd gitfile
git init
touch file.txt
nano file.txt
# edited and writen this is my first file
git add .
git commit -m "first commit"

# edit file.txt again
nano file.txt
git status
```

```
git diff
```

```
# Output
diff --git a/file.txt b/file.txt
index 751b266..848291b 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 this is me. My first commit -------- this line already existed in t
+this it edit to check git diff work or not ------ this line shows t
```

git diff command displays the changes done in the files before committing it.

Example 2:


```
git diff # before committing changes done
```

Compared Files

For each comparison, Git explains which files it is comparing. Usually this is two versions of the same file.

Git also declares one file as "A" and the other as "B".

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644
--- a/rainbow.txt
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```

File Metadata

You really do not need to care about the file metadata.

The first two numbers are the hashes of the two files being compared. The last number is an internal file mode identifier.

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644
--- a/rainbow.txt
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```

Markers

File A and File B are each assigned a symbol.

- File A gets a minus sign (-)
- File B gets a plus sign (+)

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644
--- a/rainbow.txt
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```

Chunks

A diff won't show the entire contents of a file, but instead only shows portions or "chunks" that were modified.

A chunk also includes some unchanged lines before and after a change to provide some context

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644
--- a/rainbow.txt
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```

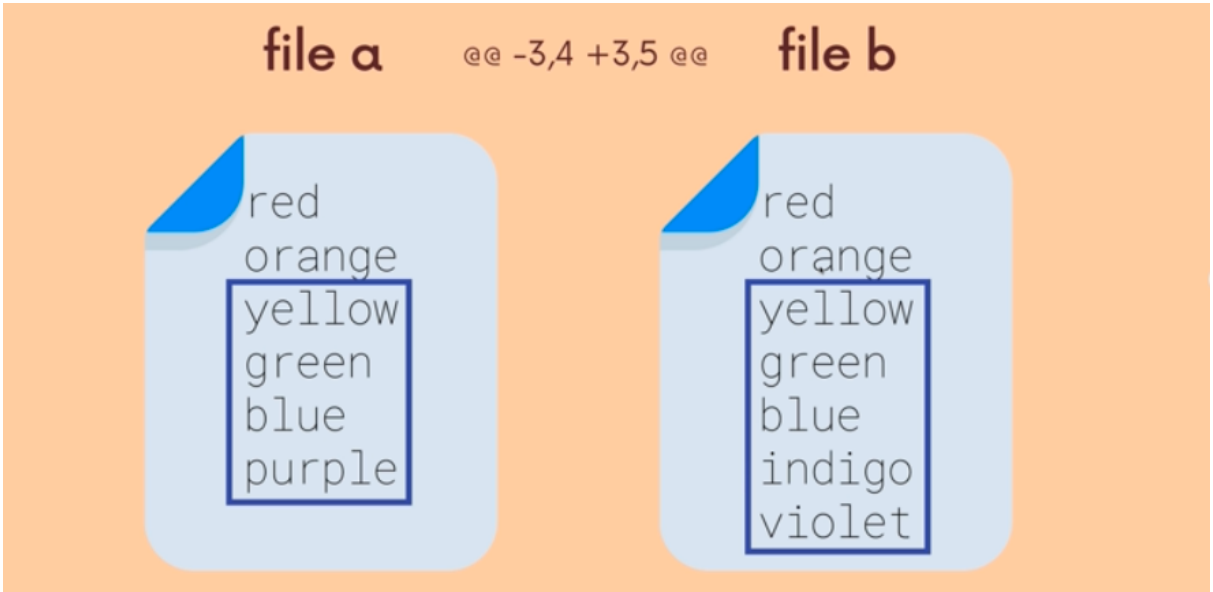
Chunk Header

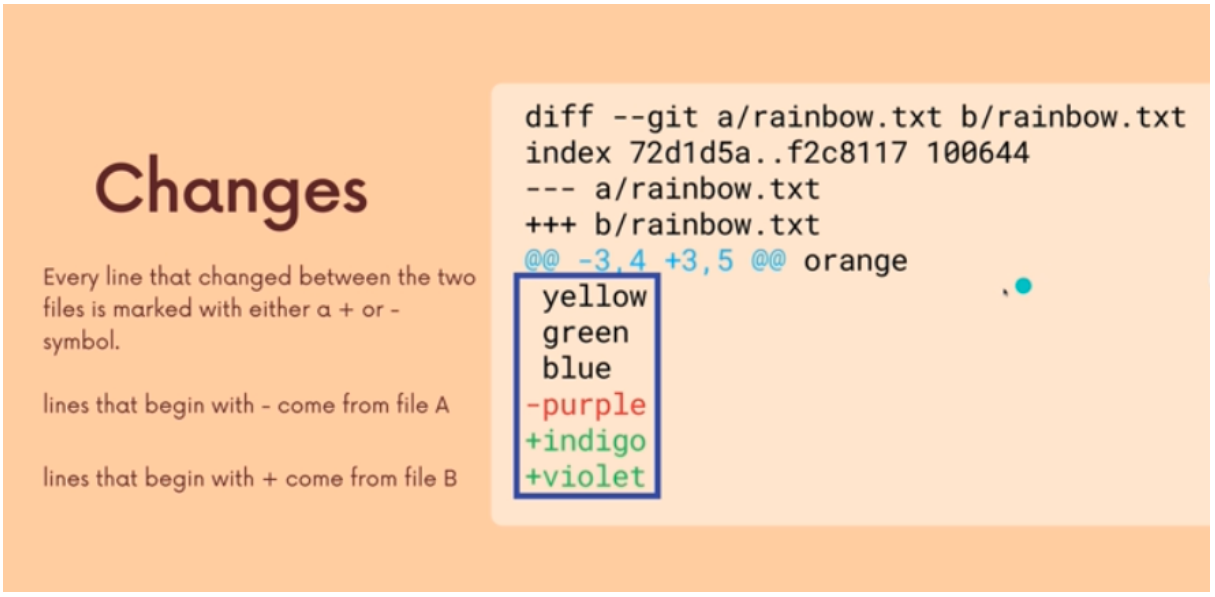
`@@ -3,4 +3,5 @@`
Each chunk starts with a chunk header, found between `@@` and `@@`.

From file a, 4 lines are extracted starting from line 3.

From file b, 5 lines are extracted starting from line 3

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644
--- a/rainbow.txt
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```





Git Diff HEAD

git diff HEAD lists all changes in the working tree since last commit.

```
git diff HEAD
```

Git Diff --staged or --cached

git diff --staged or --cached will list the changes between the staging area and our last commit.

“Show me what will be included in my commit if i run git commit right now”

```
git diff --staged
git diff --cached
```

Diff-ing Specific Files

We can view the changes within a specific file by providing git diff with a filename.

```
git diff HEAD [filename]
```

```
git diff --staged [filename]
```

Example:

```
git diff HEAD file.txt
```

```
git diff --staged file.txt
```

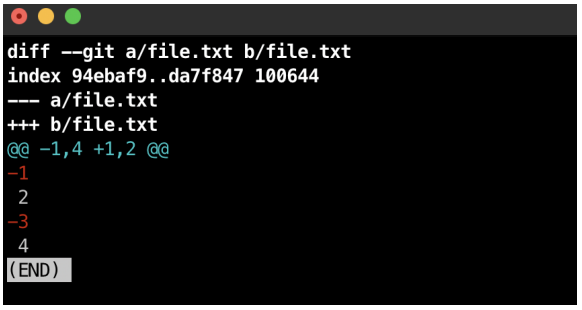
Comparing Branches by using git diff command

git diff branch1 branch2 will list the changes between the tips of branch1 and branch2

```
git diff branch1 branch2
```

example:

```
git diff master A_branch
```




```
git diff A_branch master
```

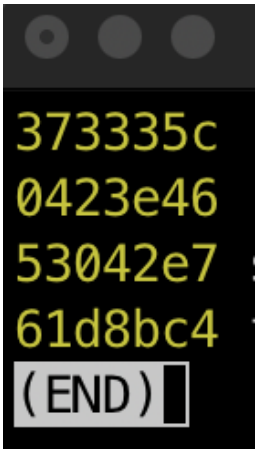
Comparing Commits by using git diff command

To compare two commits, provide git diff with the commit hashes of the commits in question.

```
git diff commit1 commit2
```

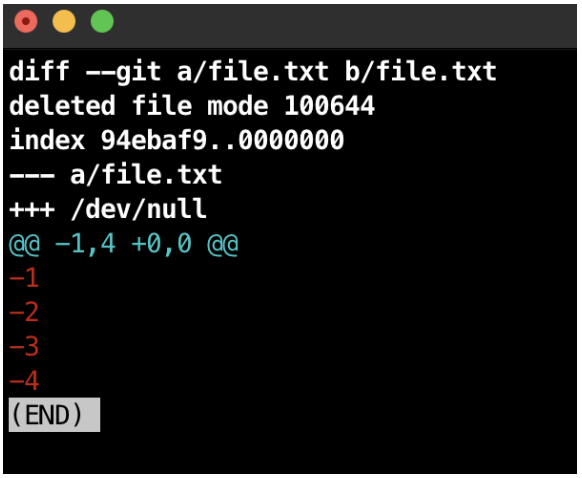
while commit1 and commit2 is the pointers of the commit which can be get using:

```
git log --oneline
```



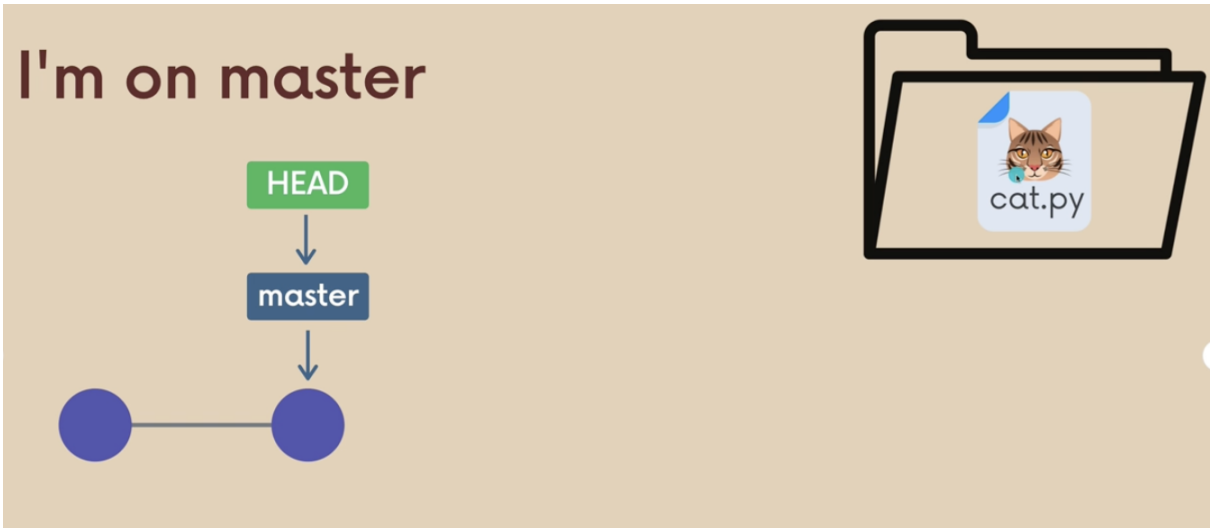
example:

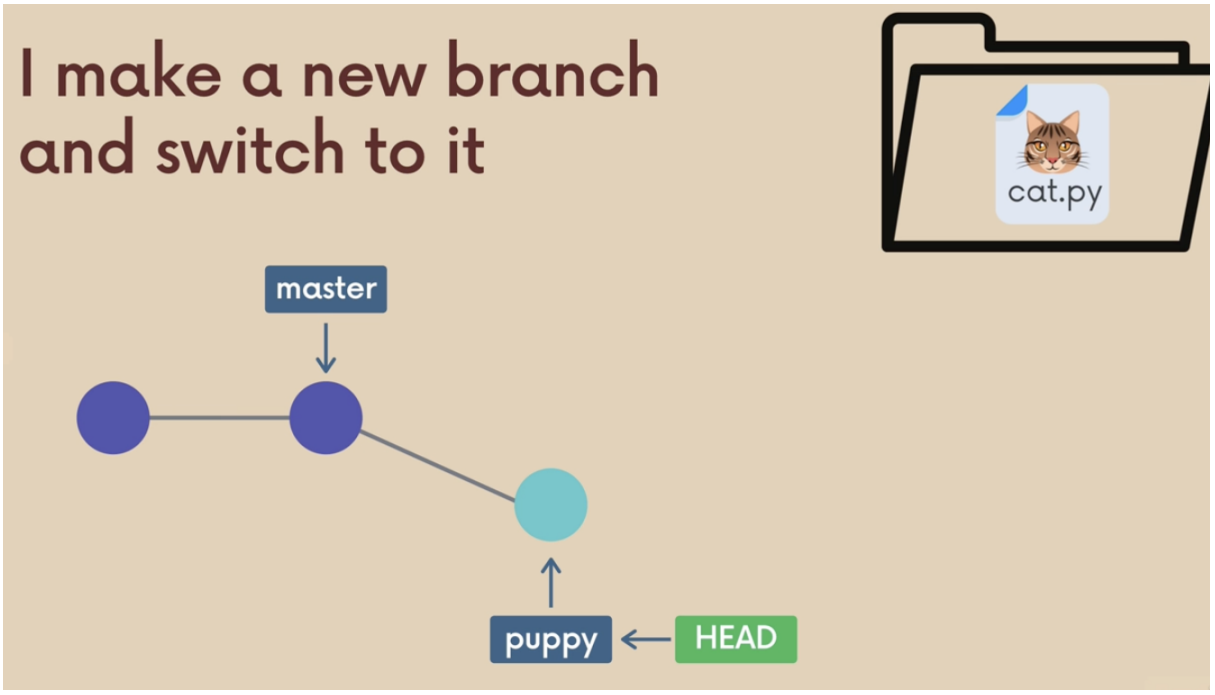
```
git diff 373335c 0423e46
```

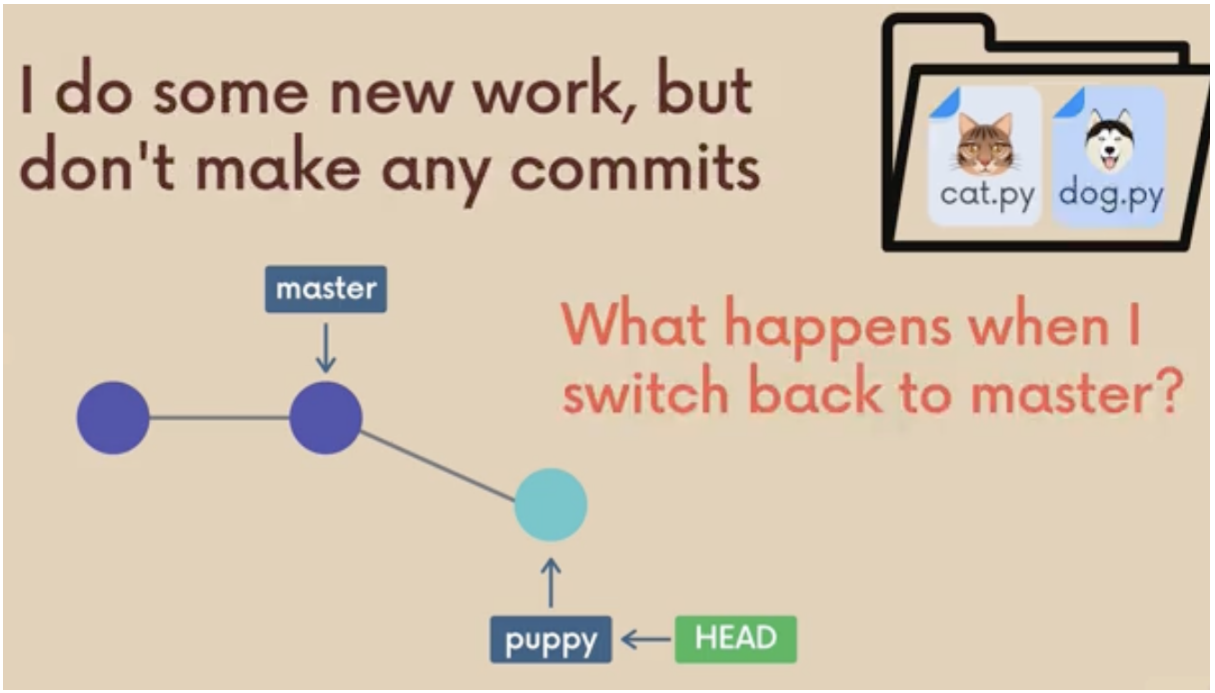


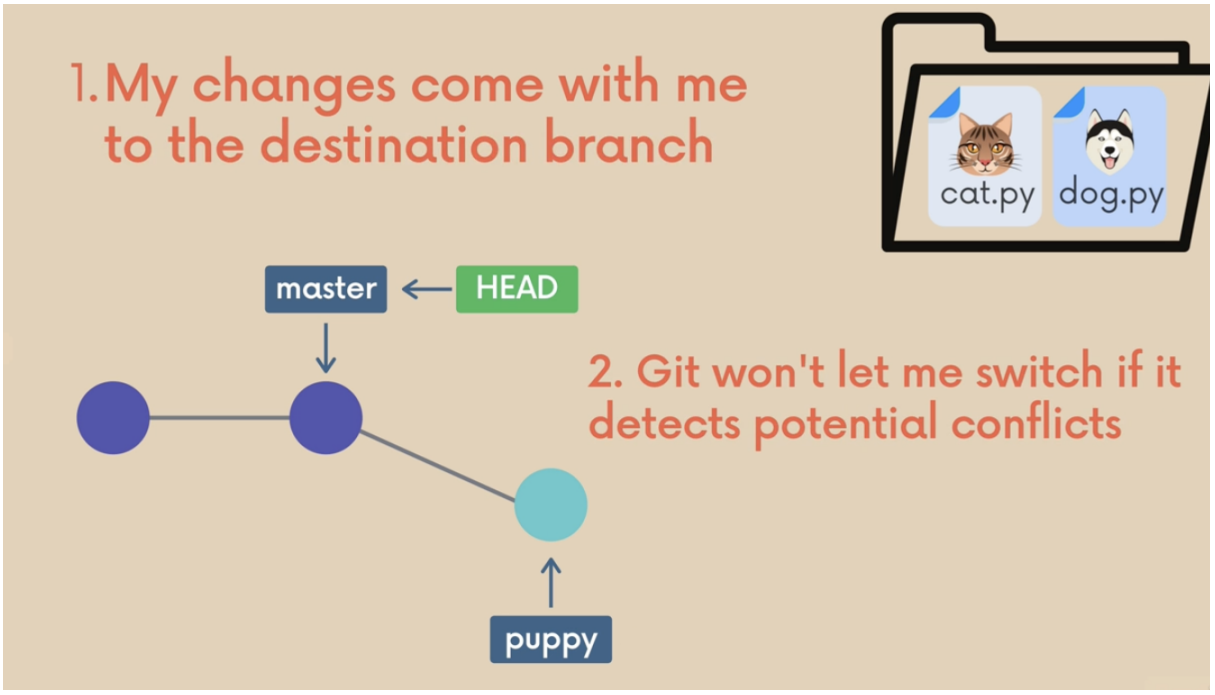
🎸 Git Diff Exercise

Git Stashing


I'm on master
HEAD
master
cat.py


I make a new branch
and switch to it
cat.py
master
puppy
HEAD





Example:

```
# master
touch file.txt
nano file.txt
git add .
git commit -m "start"
# A_branch
git switch -c A_branch
```

```
nano file.txt
git status

git switch master
git add .
git commit -m "01"
```

2nd Case use of stack

```
git switch -c B_branch
nano file.txt
git status

git switch master
# ERROR
error: Your local changes to the following files would be overwritte
    file.txt
Please commit your changes or stash them before you switch branches
Aborting
```

to solve this error we use **stash** in git.

Stashing

Git provides an easy way of stashing there uncommitted changes so that we can return to them later, without having to make unnecessary commits.

Git Stash

git stash is super useful command that helps you save changes that you are not yet ready to commit. You can stash changes ans then come back to them later.

Running git stash will take all uncommitted changes (staged and untagged) and stash them, reverting the changes in your working copy.

```
git stash

# OR

git stash apply
```

Git stash pop

Use git stash pop to remove the most recently stashed changes in your stash re-apply them to your working copy.

```
git stash pop
```

Example of git stash and git stash pop:

```
git init
git branch A
git switch A
touch file.txt
git status
git add .
git commit -m "First Commit"

git switch -c B
ls
nano file.txt
#"Hello World"
git status
git stash

git switch A
nano file.txt
# "A branch"
git add .
git commit -m "A branch"

git switch B
git stash pop
git status
git add .
git commit -m "Hello World"
```

Stashing Multiple Times

You can add multiple stashes onto the stack of stashes. They will all be stashed in the order you added them.

```
> git stash
# do some other stuff
> git stash
```

```
# do some other stuff
> git stash
```

To display the stash list # multiple stash in one branch

```
git stash list
```

Applying Specific Stashes

git assumes you want to apply the recent stash when you run git stash apply, but you can also specify a particular stash like **git stash apply stash@{2}**

```
git stash list

git stash apply stath@{id}
```

Dropping Stashes

To delete a particular stash, you can use git stash drop <stash-id>

```
git stash drop stash@{2}
```

Clear stash list

```
git stash clear
```

✍️ Stashing Exercise

Time Traveling with commits

Checkout

We can use checkout to create branches, switch to new branches, restore files, and undo history.

```
git checkout <branch-name>

or
```

```
git switch <branch-name>
```

We can use git checkout commit <commit-hash> to view a previous commit.

Remember, you can use the git log command to view commit hashes. We just need the first 7 digit of a commit hash.

example: # git log --oneline

```
git checkout d8194d6
```

Don’t panic when you see the following message…

```
You are in 'detached HEAD' state. You can look around, make experime
changes and commit them, and you can discard any commits you make in
state without impacting any branches by switching back to a branch.
```

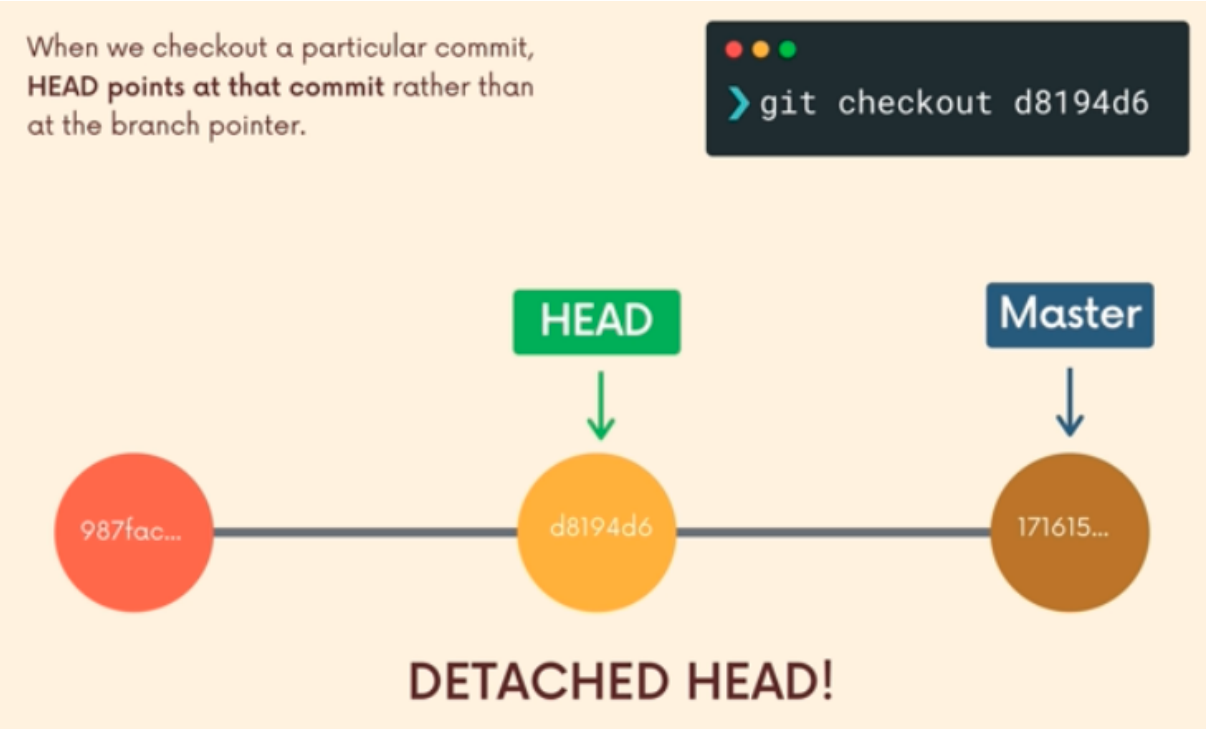
Example:

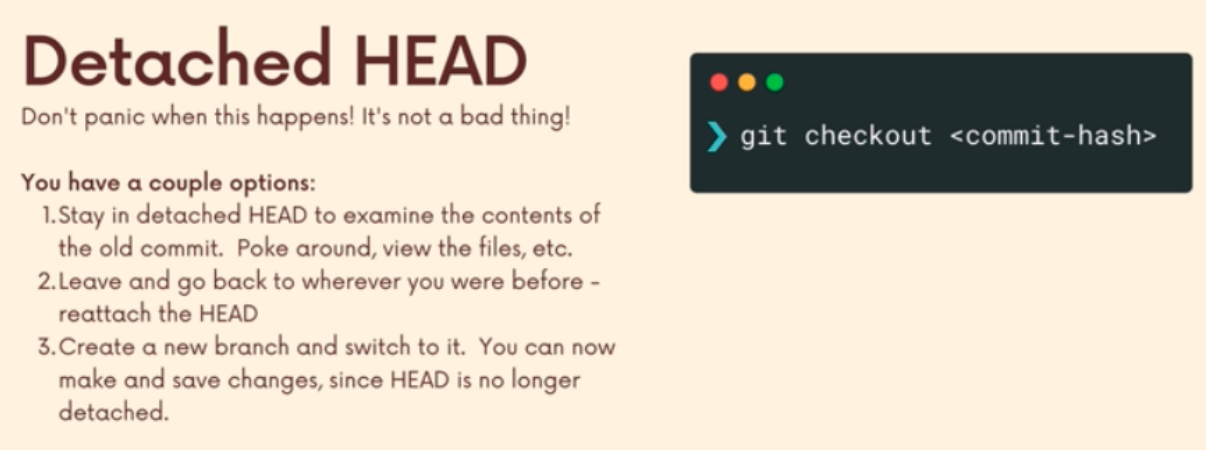
```
cd git
git init
git status

touch file.txt
nano file.txt
# hello

git status
git add .
git commit -m "1"

git log --oneline

nano file.txt
# hello world
git add .
git commit -m "2"
```

```
git log --oneline
# coppy the first commit hash ex.d8194d6 # "1"

git checkout d8194d6
```

```
cat file.txt
# will display only "hello"

git log --oneline
# coppy the second commit hash ex.23gn432 # "2"

git checkout 23gn432
cat file.txt
# display "hello world"
```

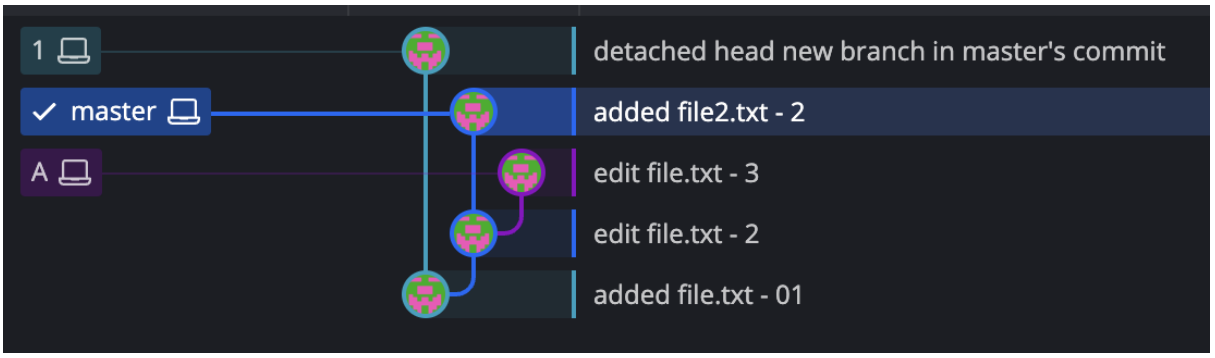
we can also use git checkout <hash> in branches





Creating a new branches in different commits is possible.

Example




```
cat .git/HEAD # to check head status
```

Re-attached HEAD

```
git checkout master

or

git switch master
```

Checkout HEAD

git checkout supports a slightly odd syntax for referencing previous commits relative to a particular commit.

HEAD~1 refers to the commit before HEAD(parent)

HEAD~2 refers to 2 commit before HEAD(grandparent)

```
git checkout HEAD~1

git checkout HEAD~2
```

git switch - will take you to previous HEAD position commit branch.

```
git switch -
```

Discarding Changes

Suppose you've made some changes to a file but don't want to keep them. To revert the file back to whatever it looked like when you last committed, you can use:

git checkout HEAD <filename> to discard any changes in the file, reverting back to the HEAD.

```
git checkout HEAD <file-name>
or
git checkout HEAD~n <file-name>
```

Example:

```
cd git
git init
git status

touch cat.txt
git add .
git commit -m "created cat.txt"
git status

nano cat.txt
# First Commit

git add .
git commit -m "First Commit"
git status
git log

nano cat.txt
# Second Commit

git add .
git commit -m "Second Commit"
git status
git log --oneline
```

```
# Undo changes / no commit should be done before undoing

nano cat.txt
# abc -- to be undo
git status
```

```
git checkout HEAD cat.txt
# all chenges done will be undo -- abc
```

Another Option for discard changes

Here's another option to revert a file

```
git checkout -- <file-name>
```

Rather than typing HEAD, you can substitute --followed by the file(s) you want to restore.

Restore # work same as checkout for discard changes

git restore is a brand new Git command that helps with undoing operations.

Because it is so new, most of the existing Git tutorials and books do not mention it, but it is worth knowing!

Recall that **git checkout** does a million different things, which many git users find very confusing. **git restore** was introduced alongside **git switch** as alternative to some of the users for **checkout**.

Unmodifying Files with Restore

Suppose you've made some changes to a file since your lsat commit. You've saved the file but then realise you definitely do NOT want those changes anymore!

To restore the file to the contents in the HEAD, use **git restore <file-name>.**

```
git restore <file-name>
```

Note: The above command is not "undoable" if you have uncommitted changes in the file, they will be lost!

Example:

```
cd git
git init
git status

touch cat.txt
git add .
git commit -m "created cat.txt"
```

```
git status

nano cat.txt
# First Commit

git add .
git commit -m "First Commit"
git status
git log

nano cat.txt
# Second Commit

git add .
git commit -m "Second Commit"
git status
git log --oneline
```

```
# Undo changes / no commit should be done before undoing

nano cat.txt
# abc -- to be undo
git status

git restore cat.txt
# all chenges done will be undo -- abc
```

git restore <file-name> restores using HEAD as the default source, but we can change that using the **--source** option.

For example, **git restore --source HEAD~1 home.html** will restore the contents of home.html to its state from the commit prior to HEAD. You can also use a particular commit hash as the source.

```
git restore --source HEAD~n <file-name>
```

Example:

```
# restoring the existing cat.txt file

nano cat.txt # type any thing and save
```

```
git log --oneline # displays 3 commits

git restore --source HEAD~1 cat.txt
cat cat.txt # first commit

git restore cat.txt
cat cat.txt # first commit # second commit
```

Restore staged file # due to accidental git add .

If you have accidentally added a file to your staging area with **git add** and you don't wish to include it in the next commit, you can use **git restore** to remove it from staging.

Use the **--staged** option like this:

```
git restore --staged <file-name>
```

Example:

Example:

```
cd git
git init
git status

touch cat.txt
git add .
git commit -m "created cat.txt"
git status

nano cat.txt
# First Commit

git add .
git commit -m "First Commit"
git status
git log

nano cat.txt
# Second Commit

git add .
git commit -m "Second Commit"
```

```
git status
git log --oneline
```

```
touch secret.txt
nano secret.txt
    # API = omfklesnfkdlf23kwnds3q

nano cat.txt
    # Third Commit
git status
git add .

# now i want to resore the secret file from the staged files
git restore --staged secret.txt

# now we can make a commit which does not contain secret.txt file
git commit -m "Third Commit"

git status
git log --oneline
```

Git Reset # reset a commit

Suppose you've just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead. To undo those commits, you can use **git reset**.

git reset <commit-hash> will reset the repo back to a specific commit. The commits are gone. this is also call as regular reset or plane reset.

```
git reset <commit-hash>
```

Example:

```
cd git
git init
```

```
git status
```

```
touch cat.txt
nano cat.txt
    # First Commit
git add .
git commit -m "First Commit"

git log --oneline

nano cat.txt
    # Second Commit
git add .
git commit -m "Second commit"

git log --oneline
```

```
nano cat.txt
    # to be reset
git add .
git commit -m "reset this"
```

```
git log --oneline
# copy the hash of "reset this" commit

git reset 11d8e4b # past the hash of commit
# "reset this" commit will be deleted
```

we don’t loose the changes in the files, while we loose the commits done before.

to restore the previously done changes we can use

```
git restore <file-name>
```

we can also create a new branch and commit the changes within new branch, while no changes will be happened on master branch.

Reset --hard

If you want to undo both the commits AND the actual changes in your files, you can use the --hard option.

for example, **git reset --hard HEAD~1** will delete the last commit and associated changes.

```
git reset --hard <commit-hash>

# for HEAD
git reset --hard HEAD~1
```

Example:

```
cd git
git init
git status
```

```
touch cat.txt
nano cat.txt
    # First Commit
git add .
git commit -m "First Commit"

git log --oneline

nano cat.txt
    # Second Commit
git add .
git commit -m "Second commit"

git log --oneline
```

```
nano cat.txt
    # to be reset
git add .
git commit -m "reset this"
```

```
git log --oneline
# copy the hash of "reset this" commit

git reset --hard 11d8e4b # past the hash of commit
# "reset this" commit will be deleted

or
```

```
git reset --hard HEAD~1
```

Git Revert

Yet another similar sounding and confusing command that has to do with undoing changes.

git revert

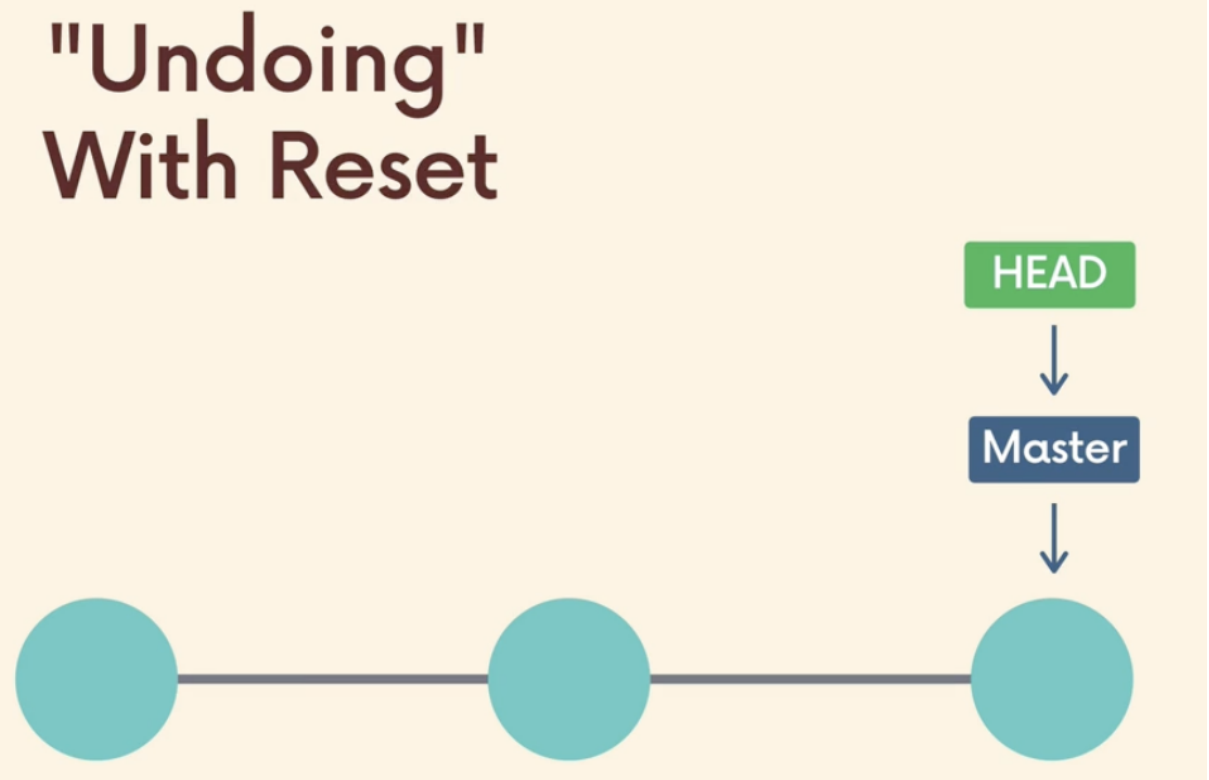
git revert is similar to **git reset** in that both “undo” changes, but they accomplish it in different ways.

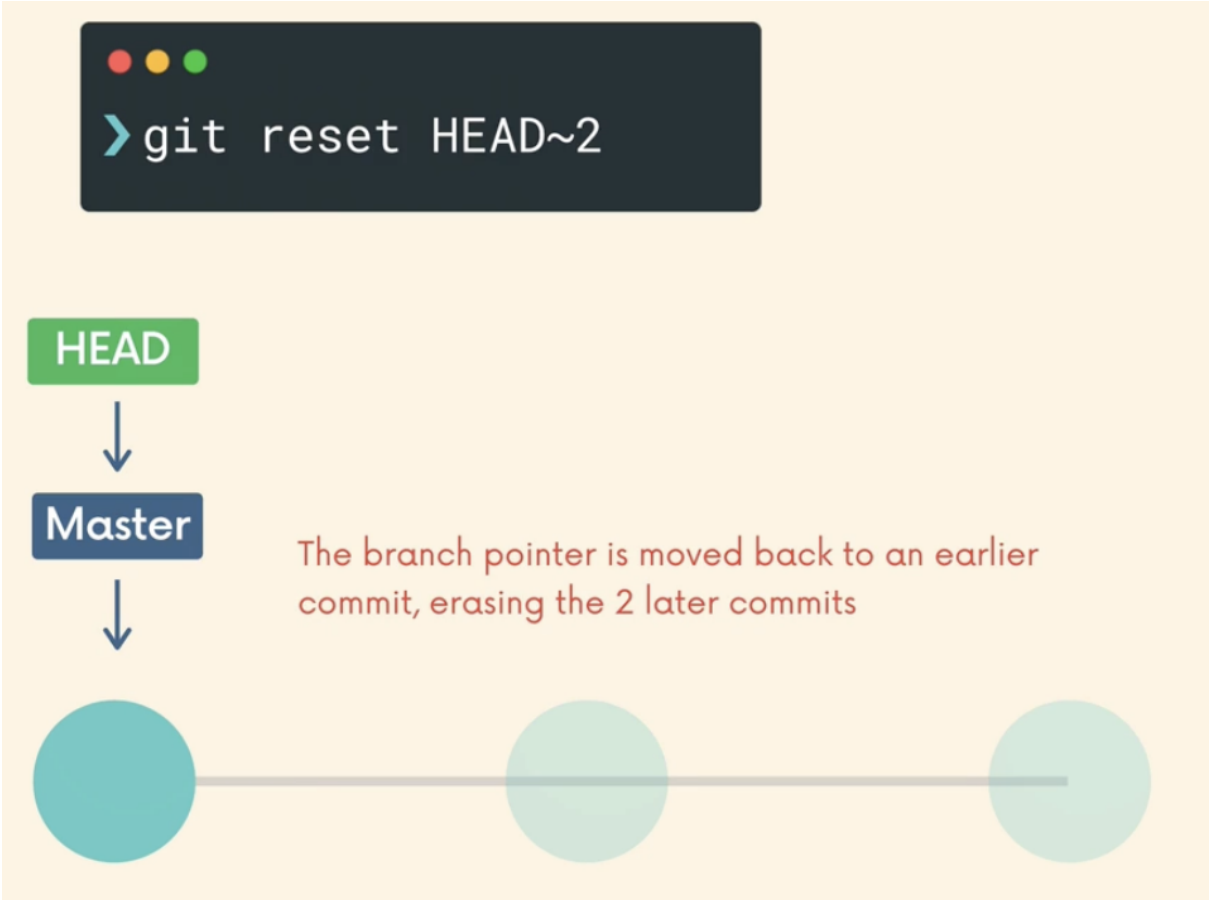
git reset actually moves the branch pointer backwards, eliminating commits.

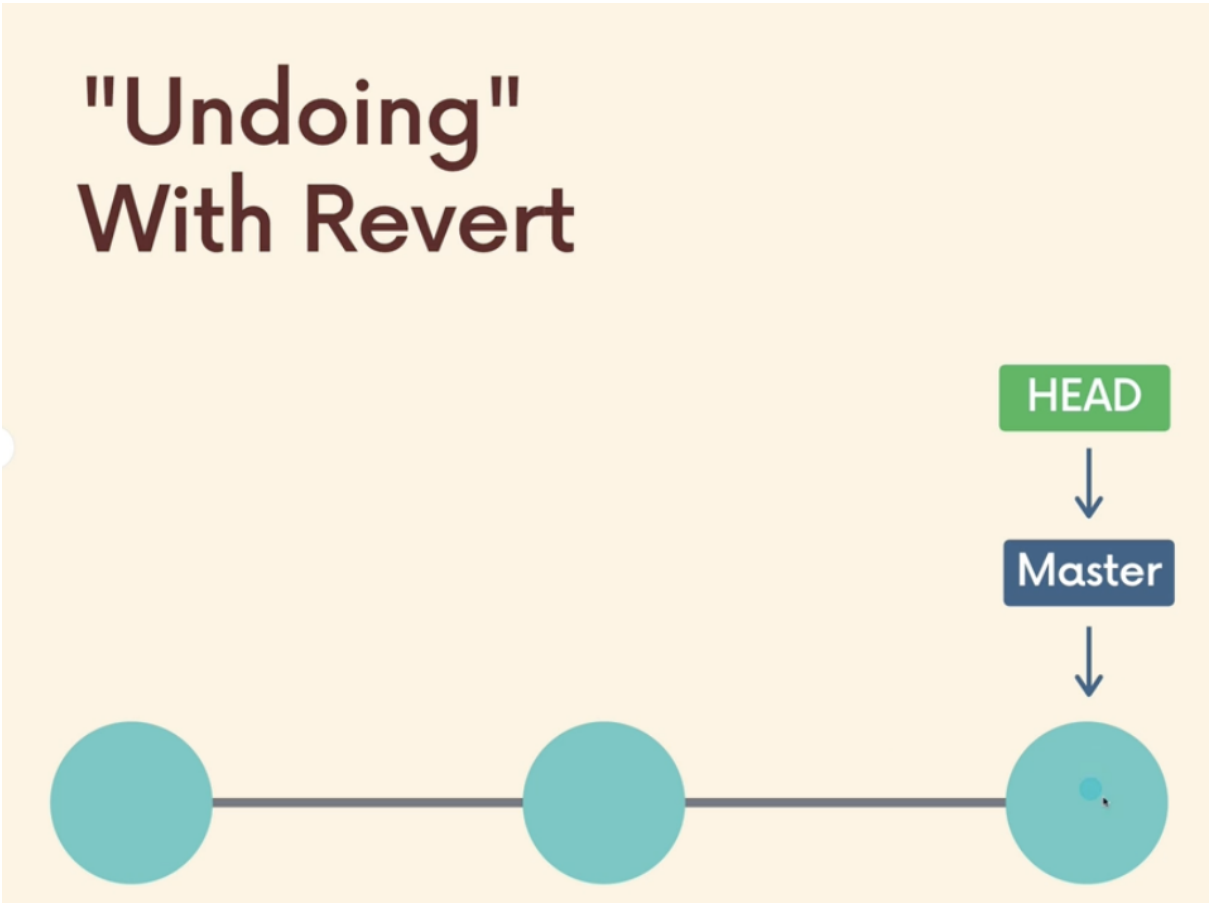
git revert instead creates a branch new commit which reverses / undos the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.

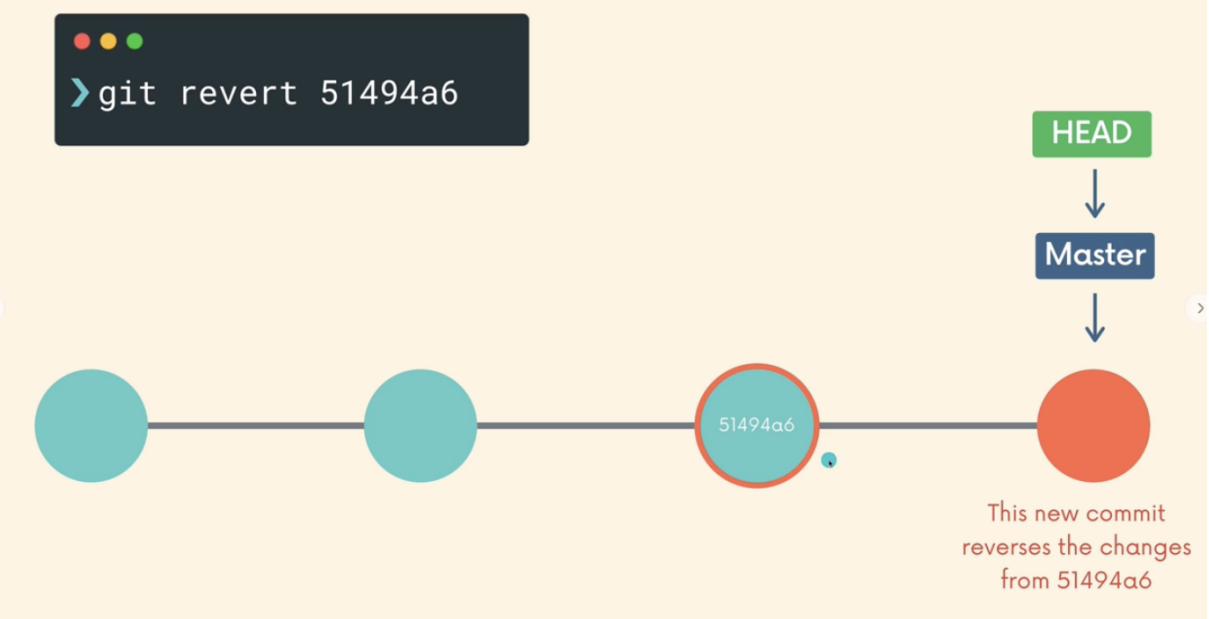
```
git revert <commit-hash>
```

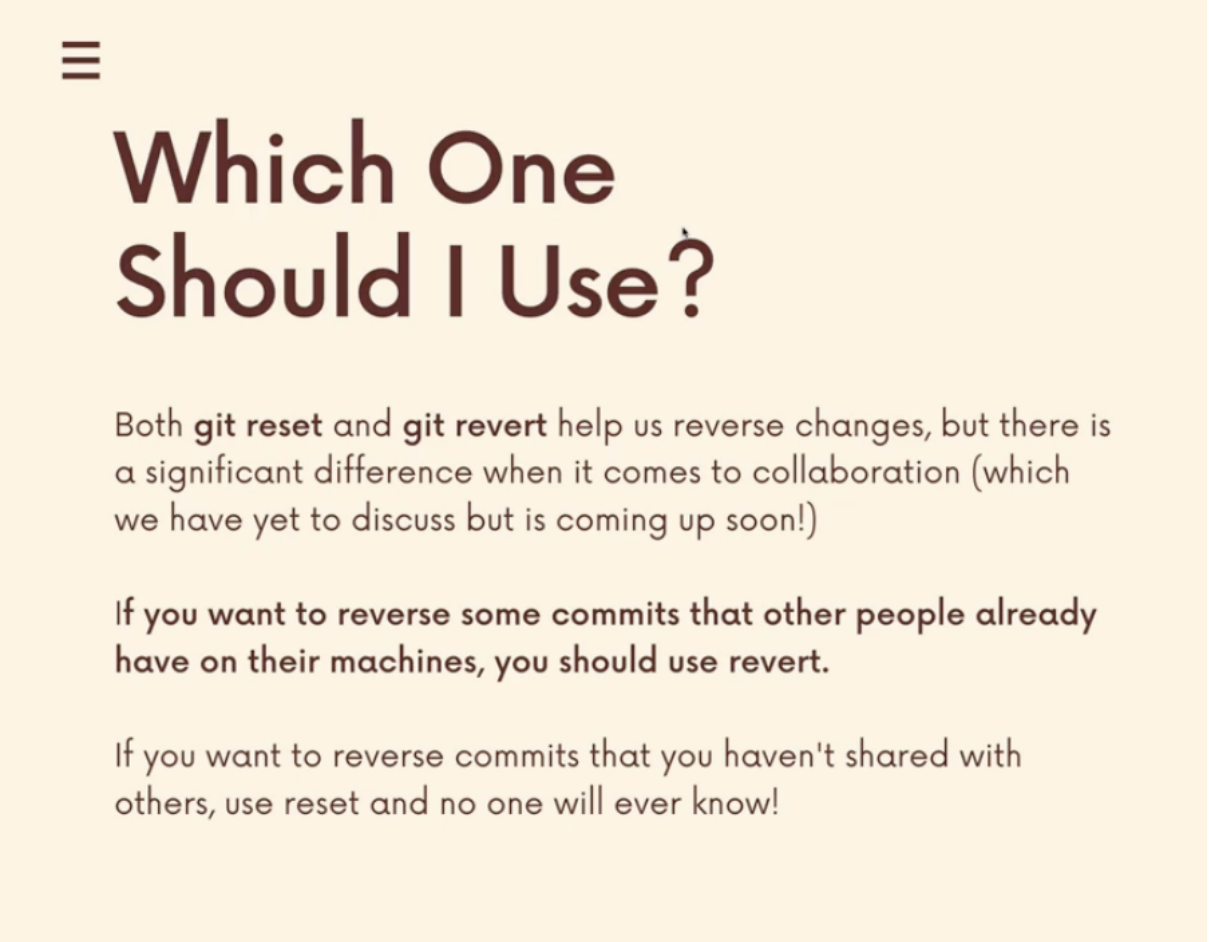
Example:




git reset HEAD~2
HEAD
Master
The branch pointer is moved back to an earlier commit, erasing the 2 later commits


"Undoing"
With Revert
HEAD
Master





Example2:

```
cd git
git init
```

```
git status
```

```
touch cat.txt
nano cat.txt
    # First Commit
git add .
git commit -m "First Commit"

git log --oneline

nano cat.txt
    # Second Commit
git add .
git commit -m "Second commit"

git log --oneline
```

```
nano cat.txt
    # to be revert
git add .
git commit -m "revert this"
```

```
git log --oneline
# copy the hash of "revert this" commit

git revert 11e2fe
# editer will be open asking for entering new commit name
# you will be switching to the new commit and the bad commit will be
# you can access the bad commit by using hash or HEAD~n with checkou
```

🎵 Undoing Things Exercise

Github

What is Github

Github is a hostile platform for git repositories. You can put your own Git repos on Github and access them from anywhere and share them with people around the world.

Beyond hosting repos, Github also provides additional collaboration features that are not native to Git (but are super useful). Basically, Github helps people share and collaborate on repos.

similar to Github:- GitLab, BitBucket, and Gerrit.

Open Source Projects

Today Github is the home of open source projects on the internet. Projects ranging from React to Swift are hosted on Github.

If you plan on contributing to open source projects, you'll need to get comfortable working with Github.

Exposure

Your Github profile showcases your own projects and contributions to other projects. It can act as a sort of resume that many employers will consult in the hiring process. Additionally, you can gain some clout on the platform for creating or contributing to popular projects.

Cloning

So far we've created our own Git repositories from scratch, but often we want to get a local copy of an existing repository instead.

To do this, we can clone a remote repository hosted on Github or similar websites. All we need is a URL that we can tell Git to clone for use.

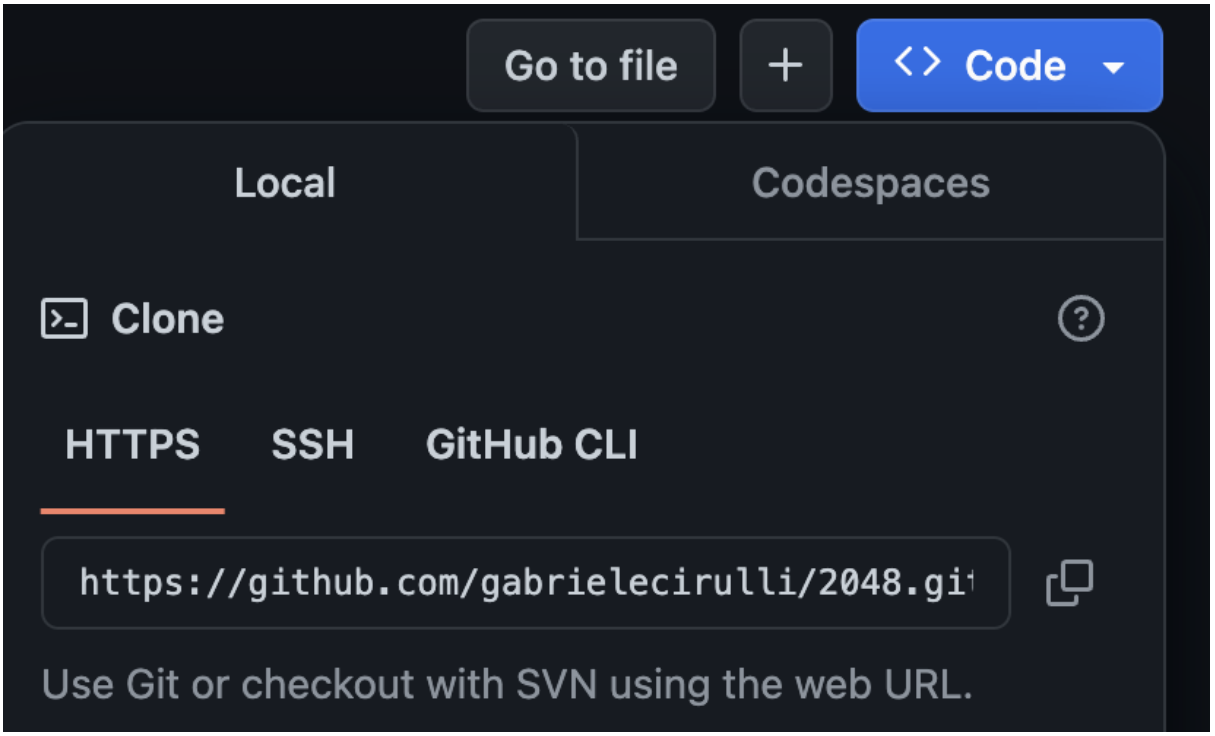
git clone

To clone a repo, simply run **git clone <url>.**

Git will retrieve all the files associated with the repository and will copy them to your local machine.

In addition, Git initialises a new repositories on your machine, giving you access to the full Git history of the cloned project.

```
git clone <url>
```



example:

```
git clone https://github.com/poteto/hiring-without-whiteboards.git
```

SSH Keys setup Github

You need to be authenticated on Github to do certain operations, like pushing up code from your local machine. Your terminal will prompt you every single time for Github email and password, unless..

You generate and configure an SSH key! Once configured, you can connect to Github without having to supply your username / password.

Checking SSH exists

1. Open Terminal.
2. Enter `ls -al ~/.ssh` to see if existing SSH keys are present.

```
$ ls -al ~/.ssh
# Lists the files in your .ssh directory, if they exist
```

3. Check the directory listing to see if you already have a public SSH key. By default, the filenames of supported public keys for GitHub are one of the following.
    - *id_rsa.pub*
    - *id_ecdsa.pub*
    - *id_ed25519.pub*

Checking for existing SSH keys - GitHub Docs

Before you generate an SSH key, you can check to see if you have any existing SSH keys.

https://docs.github.com/en/authentication/connecting-to-github-with-ssh/checking-for-existing-ssh-keys

Generate SSH Key

Generating a new SSH key and adding it to the ssh-agent - GitHub Docs

After you've checked for existing SSH keys, you can generate a new SSH key to use for authentication, then add it to the ssh-agent.

https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

1. Open Terminal.
2. Paste the text below, replacing the email used in the example with your GitHub email address.

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

> **Note:** If you are using a legacy system that doesn't support the Ed25519 algorithm, use:
>
> ```
> ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
> ```

This creates a new SSH key, using the provided email as a label.

```
> Generating public/private ALGORITHM key pair.
```

When you're prompted to "Enter a file in which to save the key", you can press **Enter** to accept the default file location. Please note that if you created SSH keys previously, ssh-keygen may ask you to rewrite another key, in which case we recommend creating a custom-named SSH key. To do so, type the default file location and replace id_ALGORITHM with your custom key name.

```
> Enter a file in which to save the key (/Users/YOU/.ssh/id_ALGORITHM):
[Press enter]
```

3. At the prompt, type a secure passphrase. For more information, see "Working with SSH key passphrases."

```
> Enter passphrase (empty for no passphrase): [Type a passphrase]
> Enter same passphrase again: [Type passphrase again]
```

Adding your SSH key to the ssh-agent

Before adding a new SSH key to the ssh-agent to manage your keys, you should have checked for existing SSH keys and generated a new SSH key. When adding your SSH key to the agent, use the default macOS `ssh-add` command, and not an application installed by macports, homebrew, or some other external source.

1. Start the ssh-agent in the background.

```
$ eval "$(ssh-agent -s)"
> Agent pid 59566
```

   Depending on your environment, you may need to use a different command. For example, you may need to use root access by running `sudo -s -H` before starting the ssh-agent, or you may need to use `exec ssh-agent bash` or `exec ssh-agent zsh` to run the ssh-agent.

2. If you're using macOS Sierra 10.12.2 or later, you will need to modify your `~/.ssh/config` file to automatically load keys into the ssh-agent and store passphrases in your keychain.
   - First, check to see if your `~/.ssh/config` file exists in the default location.

```
$ open ~/.ssh/config
> The file /Users/YOU/.ssh/config does not exist.
```

   - If the file doesn't exist, create the file.

```
touch ~/.ssh/config
```

   - Open your `~/.ssh/config` file, then modify the file to contain the following lines. If your SSH key file has a different name or path than the example code, modify the filename or path to match your current setup.

- Open your `~/.ssh/config` file, then modify the file to contain the following lines. If your SSH key file has a different name or path than the example code, modify the filename or path to match your current setup.

```
Host github.com
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/id_ed25519
```

> **Notes:**
>
> - If you chose not to add a passphrase to your key, you should omit the `UseKeychain` line.
> - If you see a `Bad configuration option: usekeychain` error, add an additional line to the configuration's' `Host *.github.com` section.
>
> ```
> Host github.com
>   IgnoreUnknown UseKeychain
> ```

3. Add your SSH private key to the ssh-agent and store your passphrase in the keychain. If you created your key with a different name, or if you are adding an existing key that has a different name, replace *id_ed25519* in the command with the name of your private key file.

```
ssh-add --apple-use-keychain ~/.ssh/id_ed25519
```

> **Note:** The `--apple-use-keychain` option stores the passphrase in your keychain for you when you add an SSH key to the ssh-agent. If you chose not to add a passphrase to your key, run the command without the `--apple-use-keychain` option.
>
> The `--apple-use-keychain` option is in Apple's standard version of `ssh-add`. In MacO versions prior to Monterey (12.0), the `--apple-use-keychain` and `--apple-load-`

created your key with a different name, or if you are adding an existing key that has a different name, replace *id_ed25519* in the command with the name of your private key file.

```
ssh-add --apple-use-keychain ~/.ssh/id_ed25519
```

> **Note:** The `--apple-use-keychain` option stores the passphrase in your keychain for you when you add an SSH key to the ssh-agent. If you chose not to add a passphrase to your key, run the command without the `--apple-use-keychain` option.
>
> The `--apple-use-keychain` option is in Apple's standard version of `ssh-add`. In MacOS versions prior to Monterey (12.0), the `--apple-use-keychain` and `--apple-load-keychain` flags used the syntax `-K` and `-A`, respectively.
>
> If you don't have Apple's standard version of `ssh-add` installed, you may receive an error. For more information, see "Error: ssh-add: illegal option -- apple-use-keychain."
>
> If you continue to be prompted for your passphrase, you may need to add the command to your `~/.zshrc` file (or your `~/.bashrc` file for bash).

4. Add the SSH public key to your account on GitHub. For more information, see "Adding a new SSH key to your GitHub account."

Steps:

```
~
ssh-keygen -t ed25519 -C "sakibdalal73@gmail.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/Users/sakibdalal/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/sakibdalal/.ssh/id_ed25519
Your public key has been saved in /Users/sakibdalal/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:WJPEQVgQbZ5FwOcB+QkgwfMKaPyc2kZULGMBX6Y6htI sakibdalal73@gmail.com
The key's randomart image is:
+--[ED25519 256]--+
| ..o=++@B*.      |
|  .+*+..B.+      |
|...o+o o+B o     |
|o+o.  .oo.+      |
|++E.... S        |
|o .=.            |
|  +              |
| . o             |
|  .              |
+----[SHA256]-----+

~
```

```
~
eval "$(ssh-agent -s)"
Agent pid 4361

~
```

```
~
open ~/.ssh/config
The file /Users/sakibdalal/.ssh/config does not exist.

~
touch ~/.ssh/config

~
nano ~/.ssh/config
```

```
Host github.com
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/id_ed25519
```

```
ssh-add --apple-use-keychain ~/.ssh/id_ed25519
Enter passphrase for /Users/sakibdalal/.ssh/id_ed25519:
Identity added: /Users/sakibdalal/.ssh/id_ed25519 (sakibdalal73@gmail.com)
```

Adding a new SSH key to your account

You can add an SSH key and use it for authentication, or commit signing, or both. If you want to use the same SSH key for both authentication and signing, you need to upload it twice.

After adding a new SSH authentication key to your account on GitHub.com, you can reconfigure any local repositories to use SSH. For more information, see "Managing remote repositories."

> **Note:** GitHub improved security by dropping older, insecure key types on March 15, 2022.
>
> As of that date, DSA keys (`ssh-dss`) are no longer supported. You cannot add new DSA keys to your personal account on GitHub.com.
>
> RSA keys (`ssh-rsa`) with a `valid_after` before November 2, 2021 may continue to use any signature algorithm. RSA keys generated after that date must use a SHA-2 signature algorithm. Some older clients may need to be upgraded in order to use SHA-2 signatures.

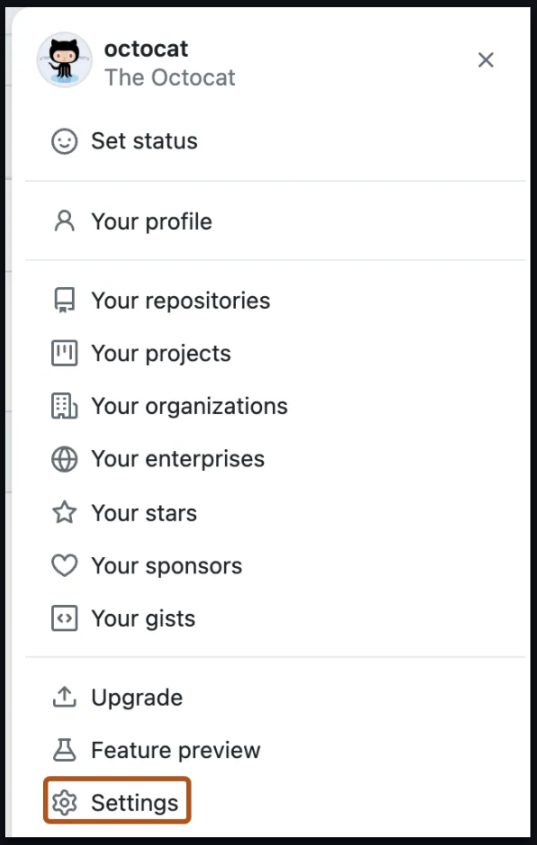
1. Copy the SSH public key to your clipboard.

   If your SSH public key file has a different name than the example code, modify the filename to match your current setup. When copying your key, don't add any newlines or whitespace.

   ```
   $ pbcopy < ~/.ssh/id_ed25519.pub
   # Copies the contents of the id_ed25519.pub file to your clipboard
   ```

   > **Tip:** If `pbcopy` isn't working, you can locate the hidden `.ssh` folder, open the file in your favorite text editor, and copy it to your clipboard.

2. In the upper-right corner of any page, click your profile photo, then click **Settings**.



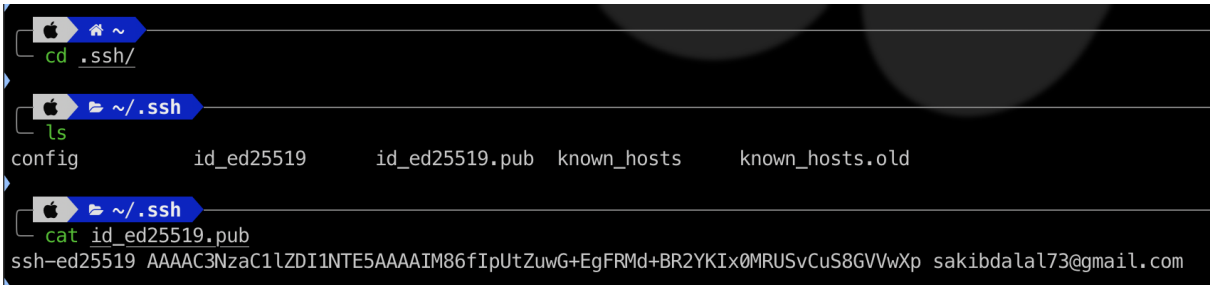
3. In the "Access" section of the sidebar, click **SSH and GPG keys**.
4. Click **New SSH key** or **Add SSH key**.
5. In the "Title" field, add a descriptive label for the new key. For example, if you're using a personal laptop, you might call this key "Personal laptop".
6. Select the type of key, either authentication or signing. For more information about commit signing, see "About commit signature verification."
7. In the "Key" field, paste your public key.
8. Click **Add SSH key**.
9. If prompted, confirm access to your account on GitHub. For more information, see "Sudo mode."

Adding a new SSH key to your GitHub account - GitHub Docs

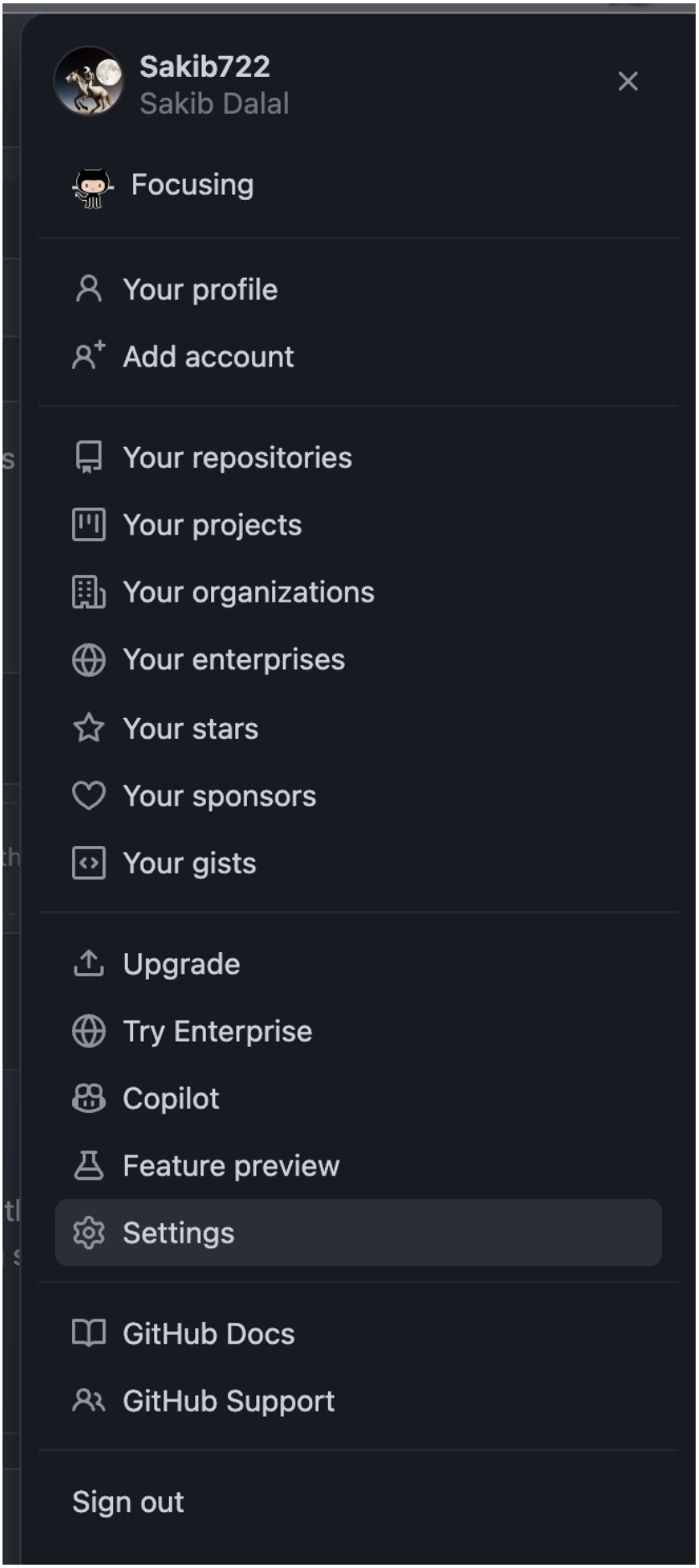
To configure your account on GitHub.com to use your new (or existing) SSH key, you'll also need to add the key to your account.

https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

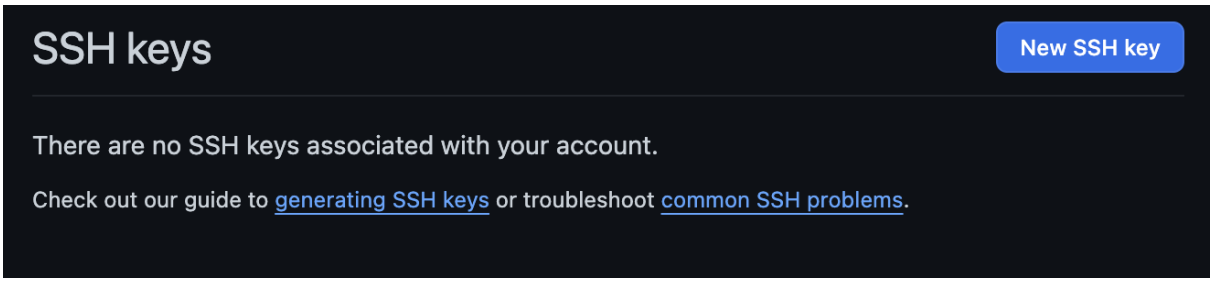
Steps:

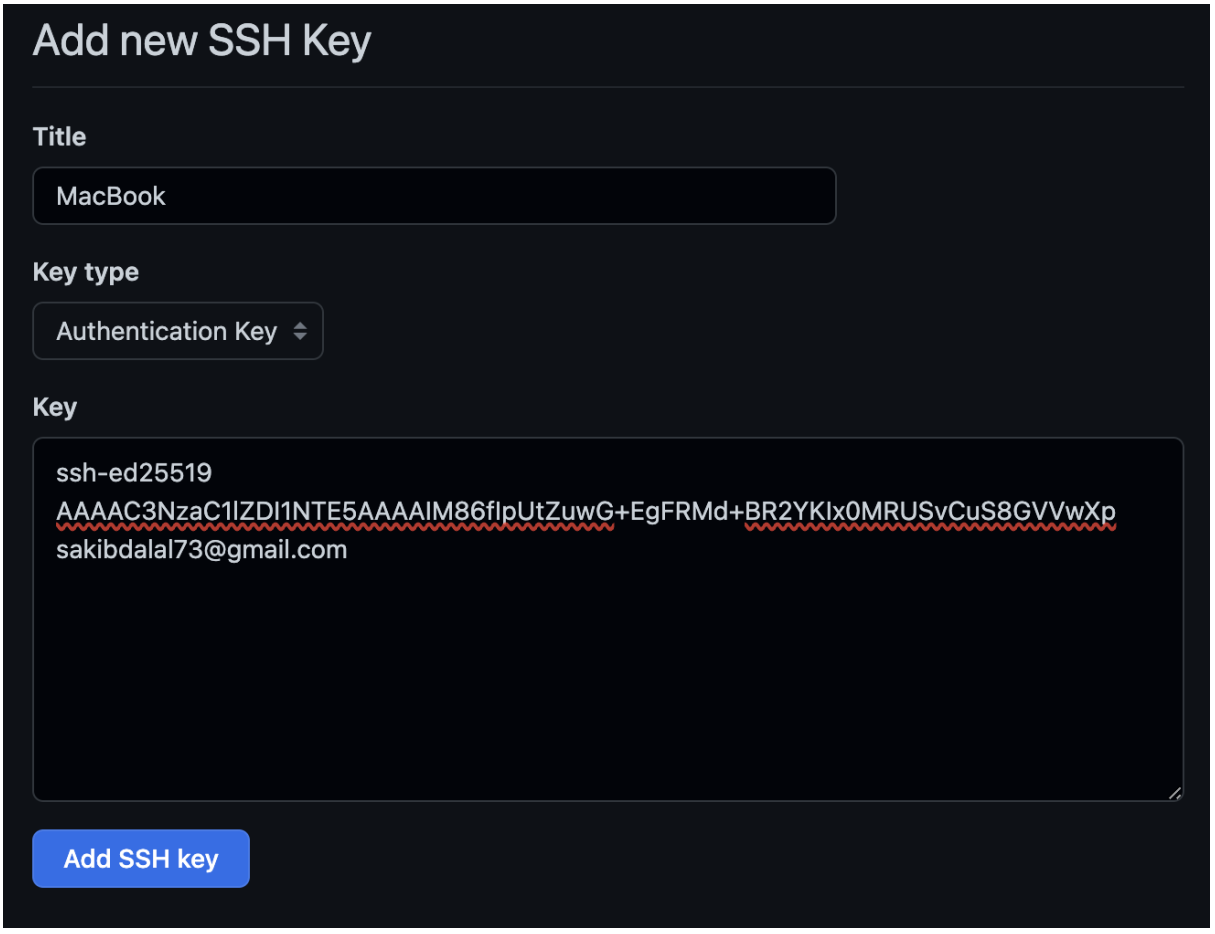


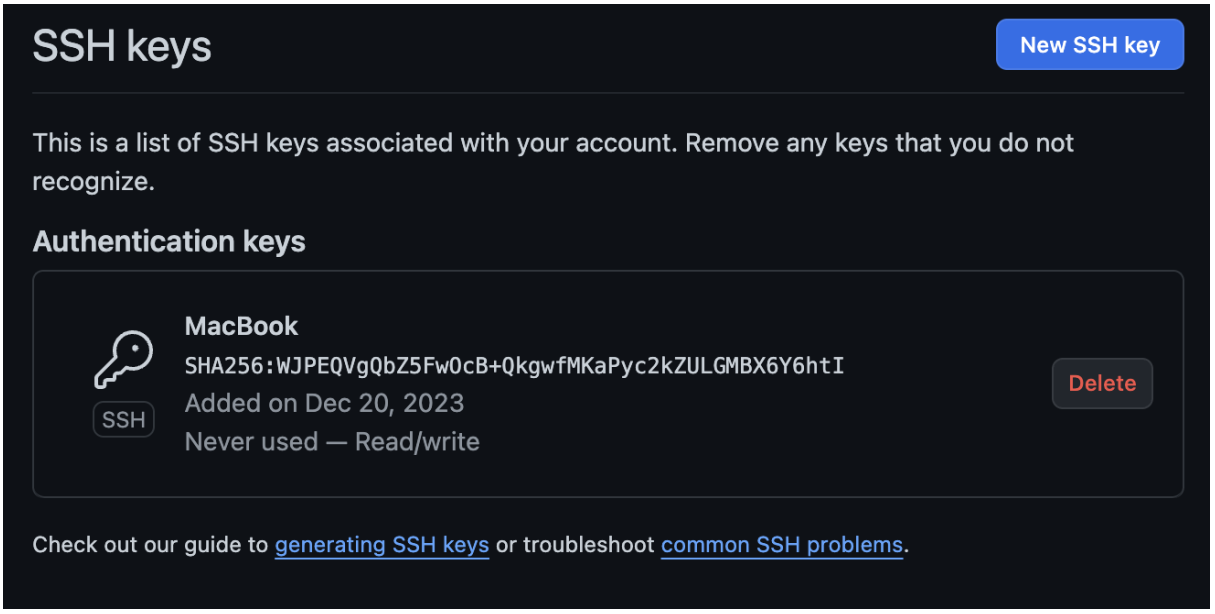
copy the ssh


Sakib722
Sakib Dalal
Focusing
Your profile
Add account
Your repositories
Your projects
Your organizations
Your enterprises
Your stars
Your sponsors
Your gists
Upgrade
Try Enterprise
Copilot
Feature preview
Settings
GitHub Docs
GitHub Support
Sign out


Sakib Dalal (Sakib722)
Your personal account
Go to your personal profile
Public profile
Account
Appearance
Accessibility
Notifications
Access
Billing and plans
Emails
Password and authentication
Sessions
SSH and GPG keys
Organizations
Enterprises


SSH keys
New SSH key
There are no SSH keys associated with your account.
Check out our guide to generating SSH keys or troubleshoot common SSH problems.





How to get code on Github

Option 1: Existing Repo

If you already have an existing repo locally that you want to get on Github.

- Create a new repo on Github
- Connect your local repo (add a remote)
- Push up your changes to Github

Option 2: Start From Scratch

If you haven’t begun work on your local repo, you can

- Create a brand new repo on Github
- Clone it down to your machine
- Do some work locally
- Push up your changes to Github

Remote

Before we can push anything up to Github, we need to tell Git about our remote repository on Github. We need to setup a “destination” to push up to.

In Git, we refer to these “destinations” as remotes. Each remote is simply a URL where a hosted repository lives.

Viewing Remotes

To view any existing remotes for you repository, we can run git remote or git remote -v (verbose, for more info)

This just displays a list of remotes. If you haven’t added any remotes yet, you won’t see anything!

```
git remote -v
```

Create a New Github Repo

By using Option - 1

Adding a new Remote

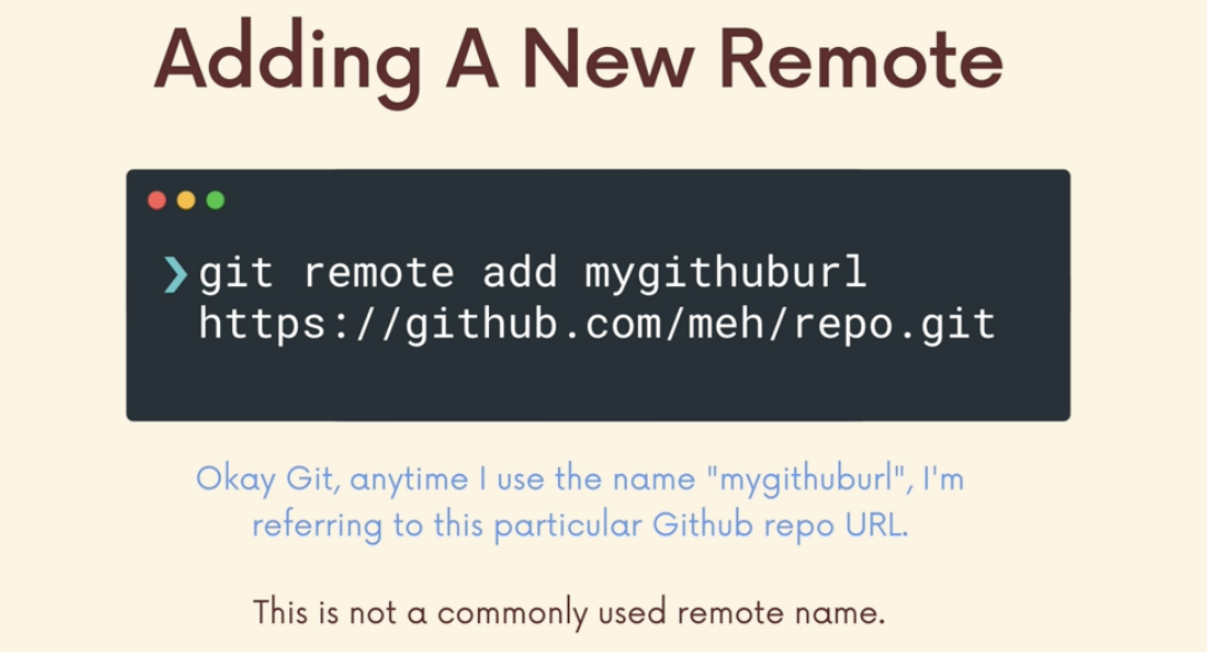
A remote is really two things: a URL and a label. To add a new remote, we need to provide both to Git.

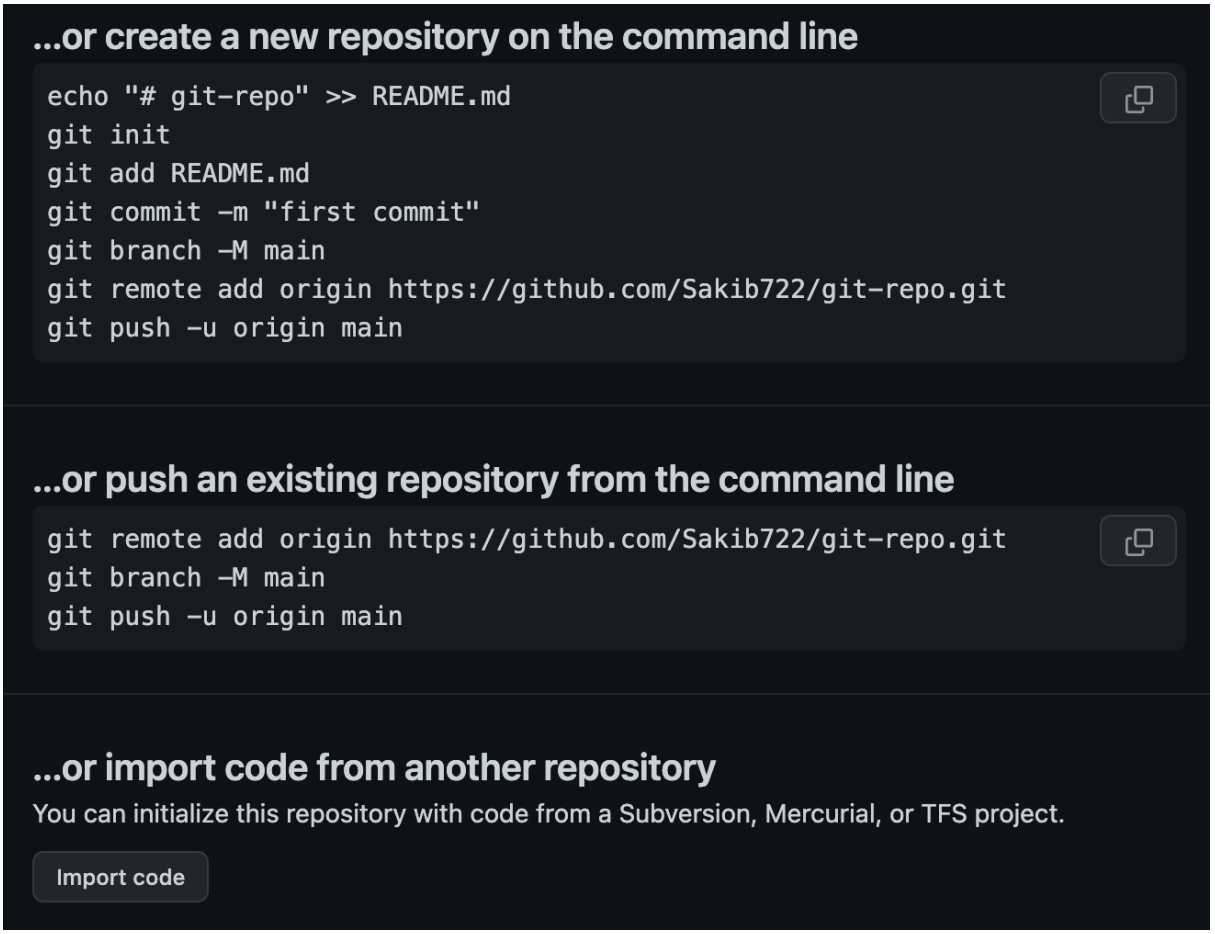
```
git remote add <name> <url>
```

Origin?

Origin is a conventional Git remote name, but it is not at all special. It’s just a name for a URL.

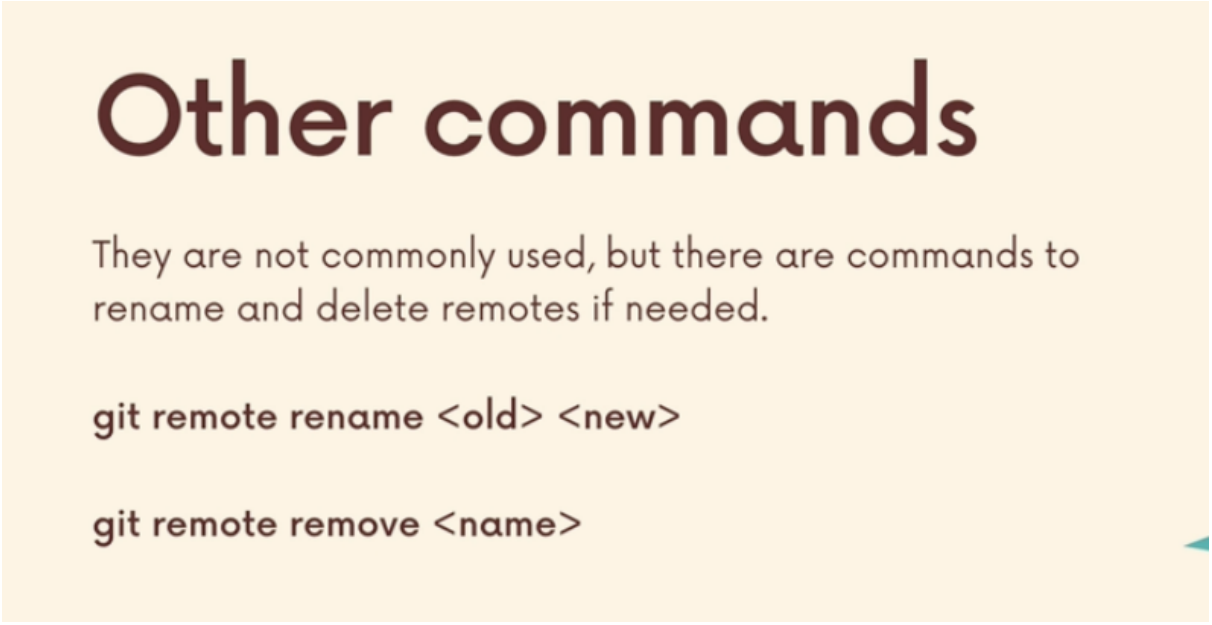
Where we clone a Github repo, the default remote name setup for us is called origin. You can change it. Most people leave it.





Example:

```
mkdir git
cd git
git init
git status
git remote -v

git remote add origin https://github.com/Sakib722/git-repo.git
git remote -v
```



Pushing

Now that we have a remote set up, let's push some work up to Github! To do this, we need to use the **git push** command.

We need to specify the remote we want to push up to AND the specific location branch we want to push up to that remote.

```
git push <remote> <branch>
```

example:

```
git push origin master
# tells git to push up the master branch to our original remote
```

Note: Before doing this use this code

git remote add for ssh not for https <url> from Github

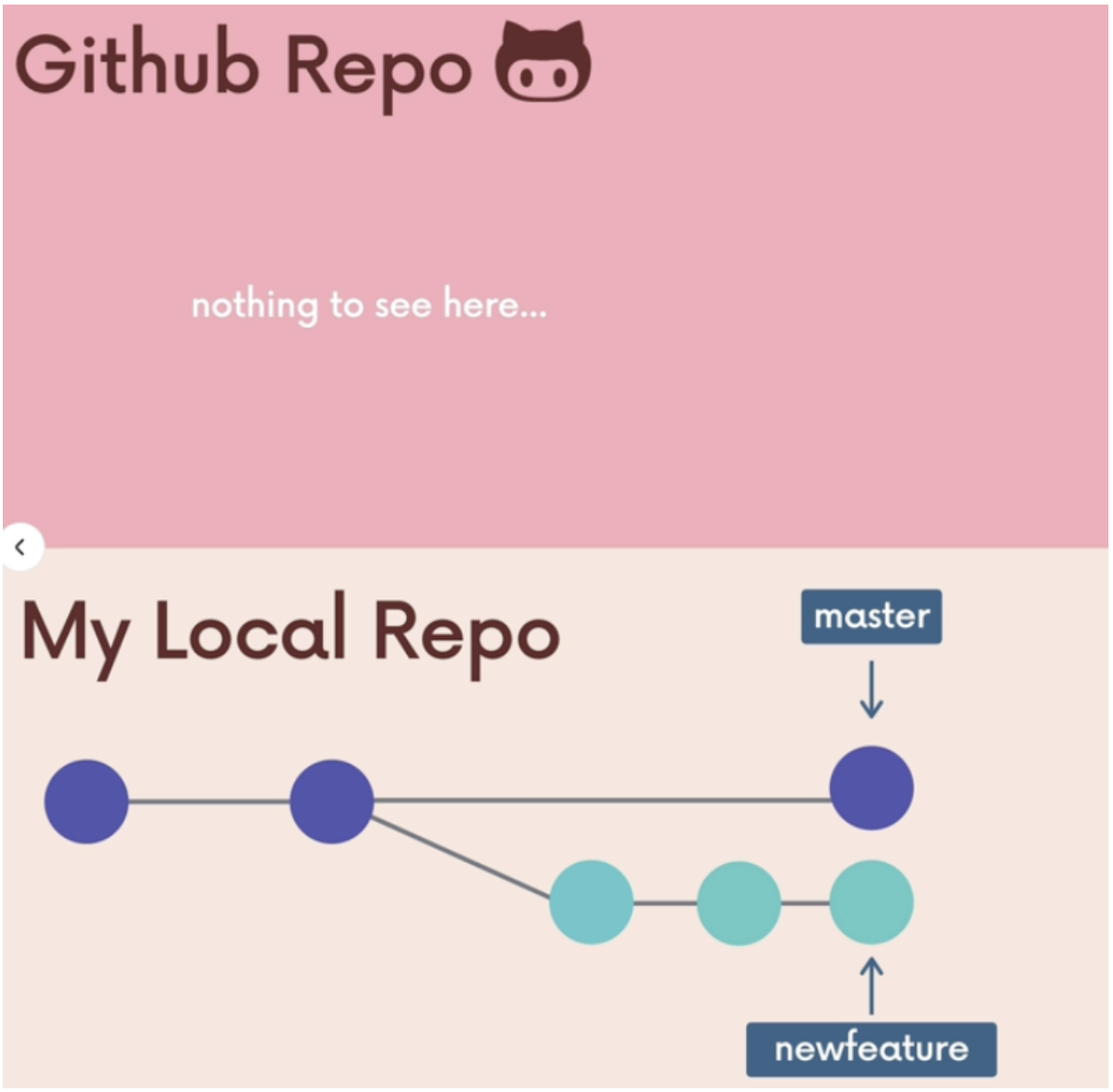
```
cd /Users/sakibdalal/.ssh/
```

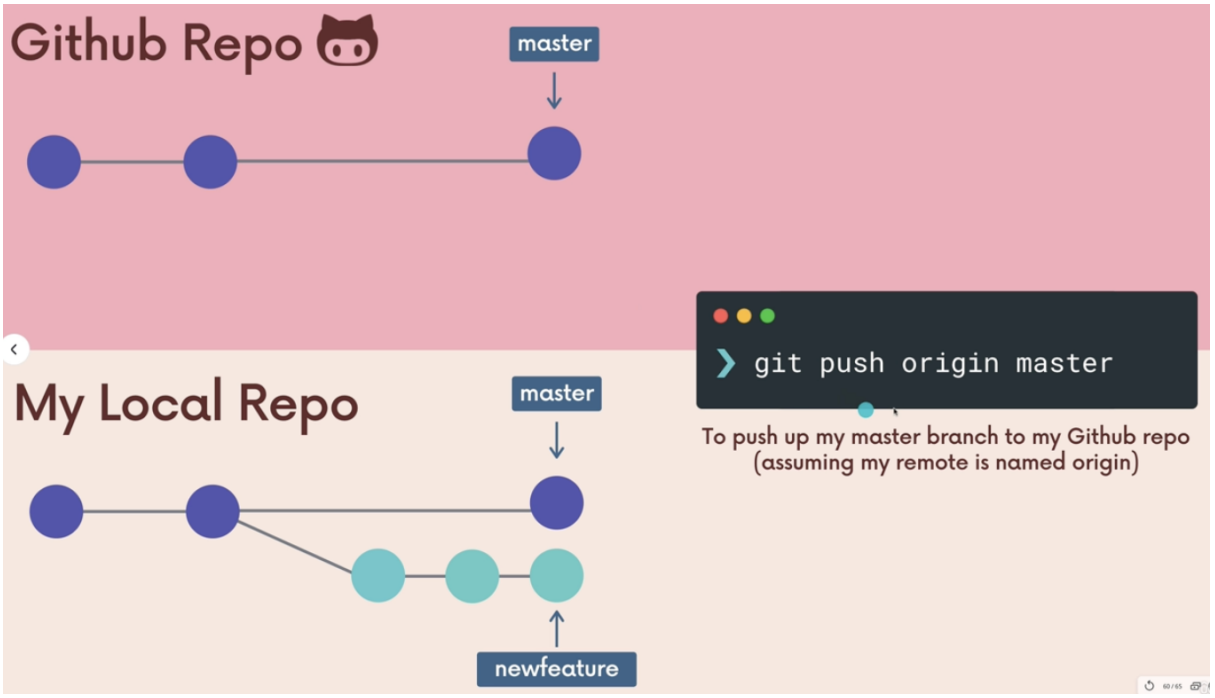
```
ssh-add -l
```

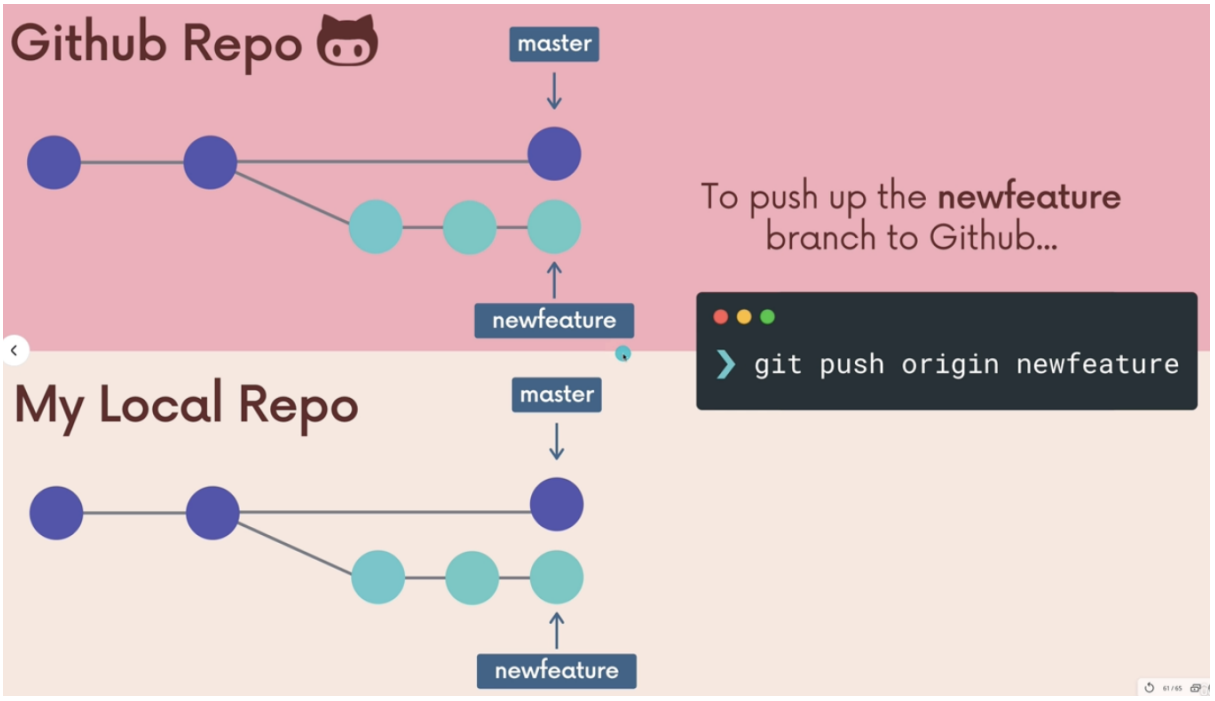
```
git config core.sshCommand "ssh -v"
```

```
ssh-add -K ~/.ssh/id_ed25519
```

In Background Example







Note:

We always need to **git push** after a change, commit in a repo.

Push In Detail

While we often want to push a local branch up to a remote branch of the same name, we don't have to!

To push our local pancake branch up to a remote branch called waffle we could do:

git push origin pancake:waffle

```
git push <remote-name> <local-branch-name>:<remote-branch-name>
```

example:

```
git push origin pancake:waffle
```

The -u option # git push -u

The -u option allows us to set the upstream (connection pointing to remote machine) of the branch we're pushing You can think of this as a link connecting our local branch to a branch on Github.

Running **git push -u origin master** sets the upstream of the local master branch so that it tracks the master branch on the origin repo.

```
git push -u origin master

git push -u <remote-name> <branch-name>
```

or

```
git push -u origin B:C

git push -u <remote-name> <local-branch-name>:<remote-branch-name>
```

Once we've set the upstream for a branch, we can use the git push shorthand while push our current branch to the upstream.

we can use only **git push** to push to the Github repo

```
git push
```

By using Option - 2 # Start from Scratch

Create a NEW Github Repository.

We will clone the repository from Github to local machine.

Note: set HTTPS instead of SSH and copy the URL link, if you are using SSH then refer to SSH url link.

on terminal:

```
git clone <url>
```

example:

```
# with https show error as we do not set https
git clone https://github.com/Sakib722/Option-2.git

# use ssh
git clone git@github.com:Sakib722/Option-2.git
```

as we set the remote already in Github repo so we can check the remote name by using

```
git remote -v
```

Note: where in Github instead of using master branch we uses main.

Working with Remote repo’s

Remote Tracking Branches

“At the time you last communicated with this remote repository, here is where x branch was pointing”

They follow this pattern <remote> / <branch>. (eg origin / master)

- origin/master references the state of the master branch on the remote repo named origin
- upstream/logoRedesign references the state of the logoRedesign branch on the remote named upstream (a common remote name)

Remote Branches

Run **git branch -r** to view the remote branches our local repository knows about.

```
git branch -r

# Output: origin/master, origin/<other-branches>
```

Fetching and Pulling

Question: cloning a repository in to a local machine will bring only the main or master branch while if a repository consist of multiple branches what should we do?

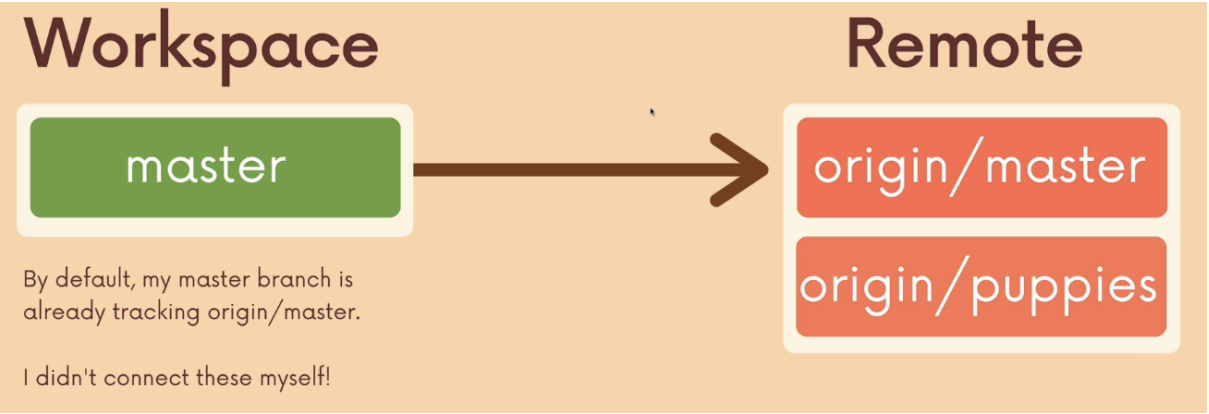
Remote Branches

Once you've cloned a repository, we have all the data and Git history for the project at that moment in time. However, that does not mean it's all in my workspace!

The Github repo has a branch called puppies, but when I run **git branch** I don't see it on my machine! All I see is the master branch. What's going on?

If we do **git branch -r** we can see all the branches on the remote side.

```
git branch -r
```



I want to work on the puppies branch locally! I could **git checkout origin/puppies**, but that puts me in detached HEAD.

```
git checkout origin/master
```

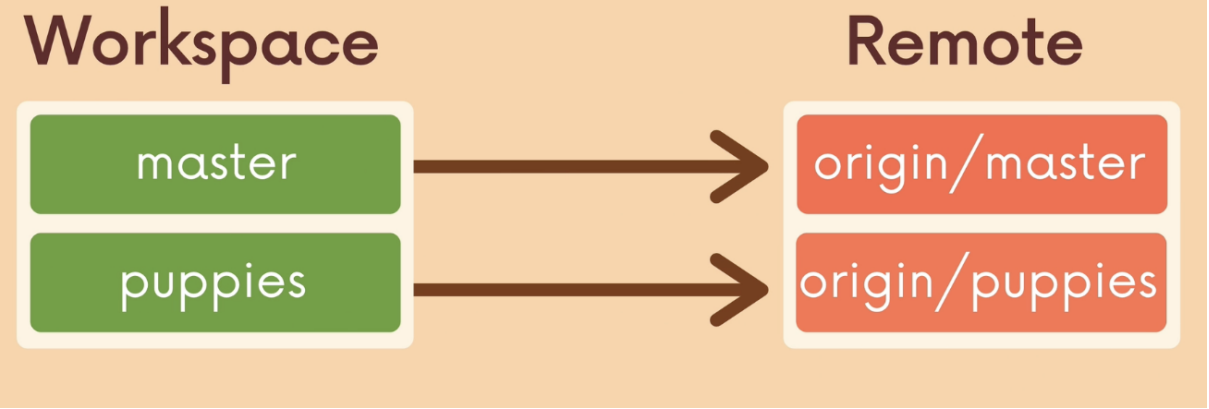
I want my own local branch called **puppies**, and I want it to be connected to **origin/puppies**, just like my local **master** branch is connected to **origin/master.**

Solution:

Run **git switch <remote-branch-name>** to create a new local branch from the remote branch of the same name.

```
git switch <remote-branch-name>

#
git switch puppies
```

git switch puppies makes me a local puppies branch AND sets it up to track the remote branch origin/puppies.

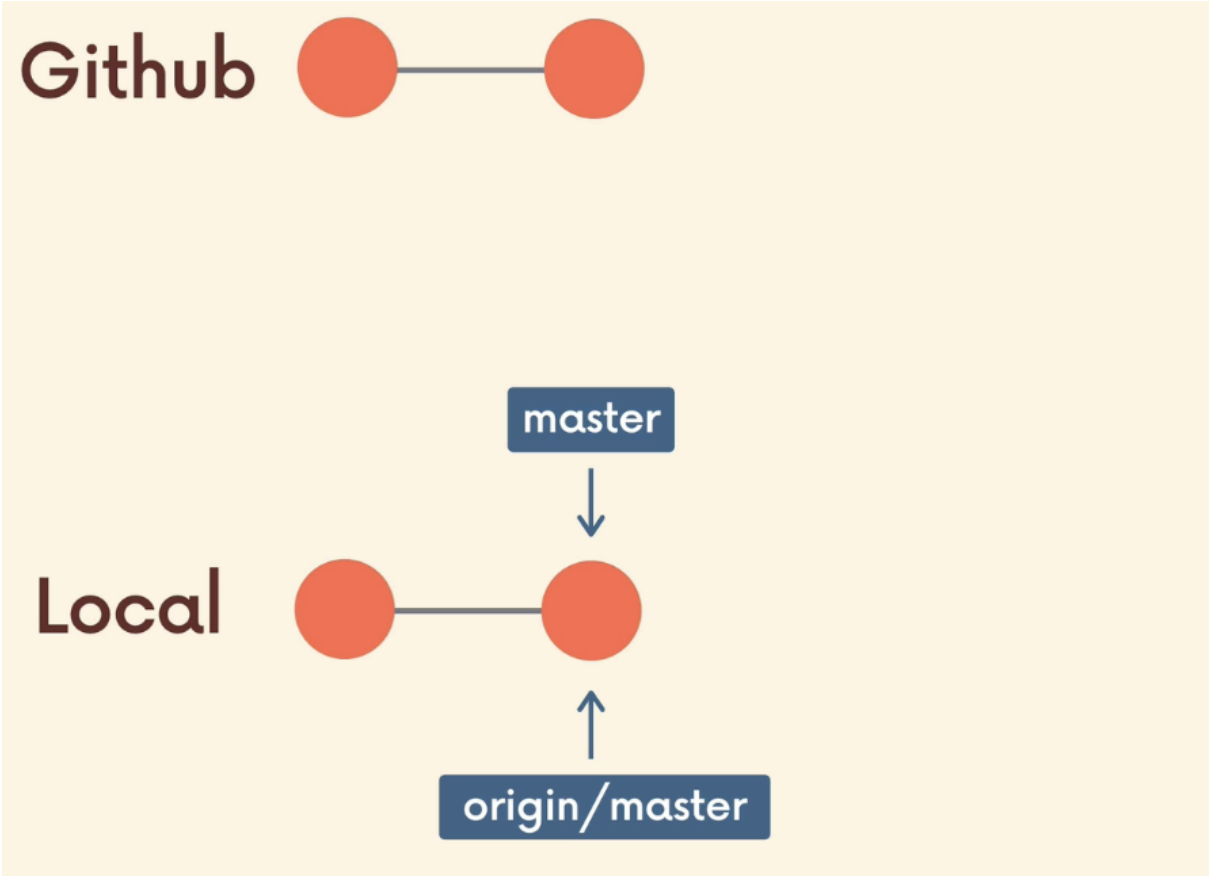


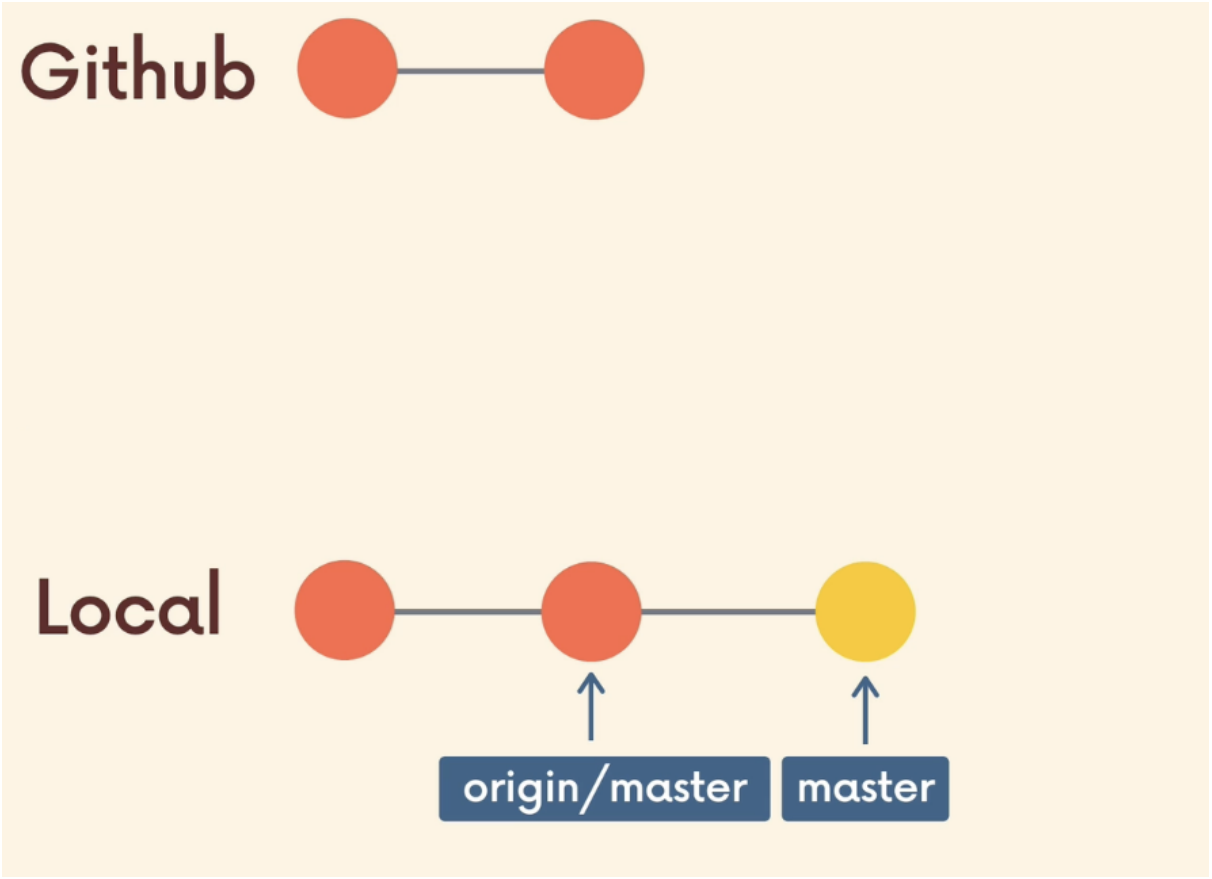

```
git branch

# main and puppies
```

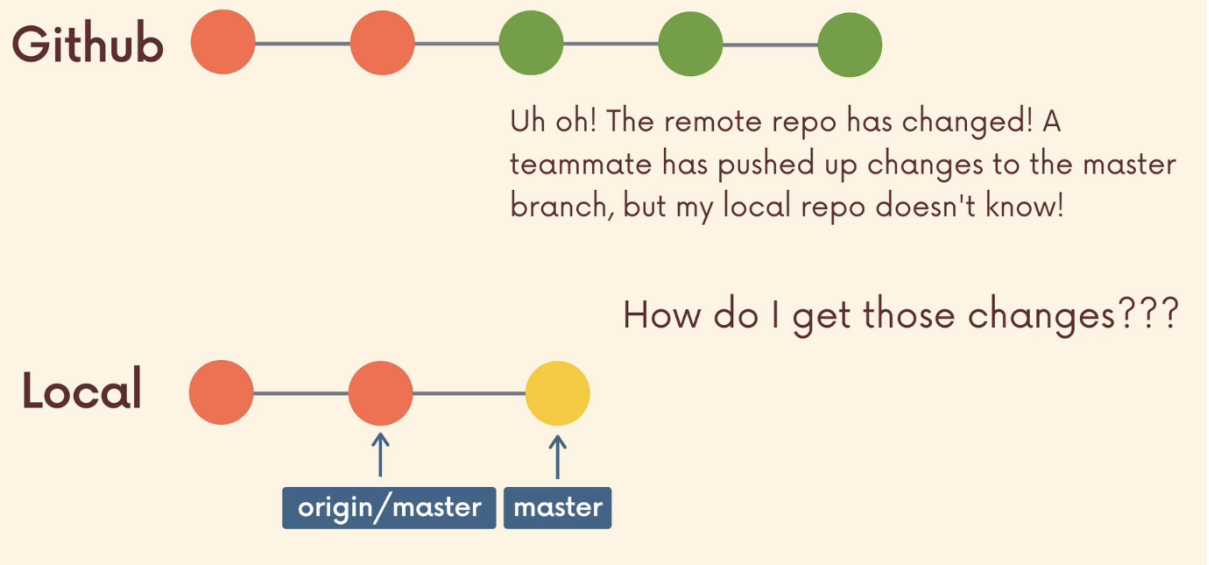

```
git checkout --track <remote-name>/<remote-branch-name>
```

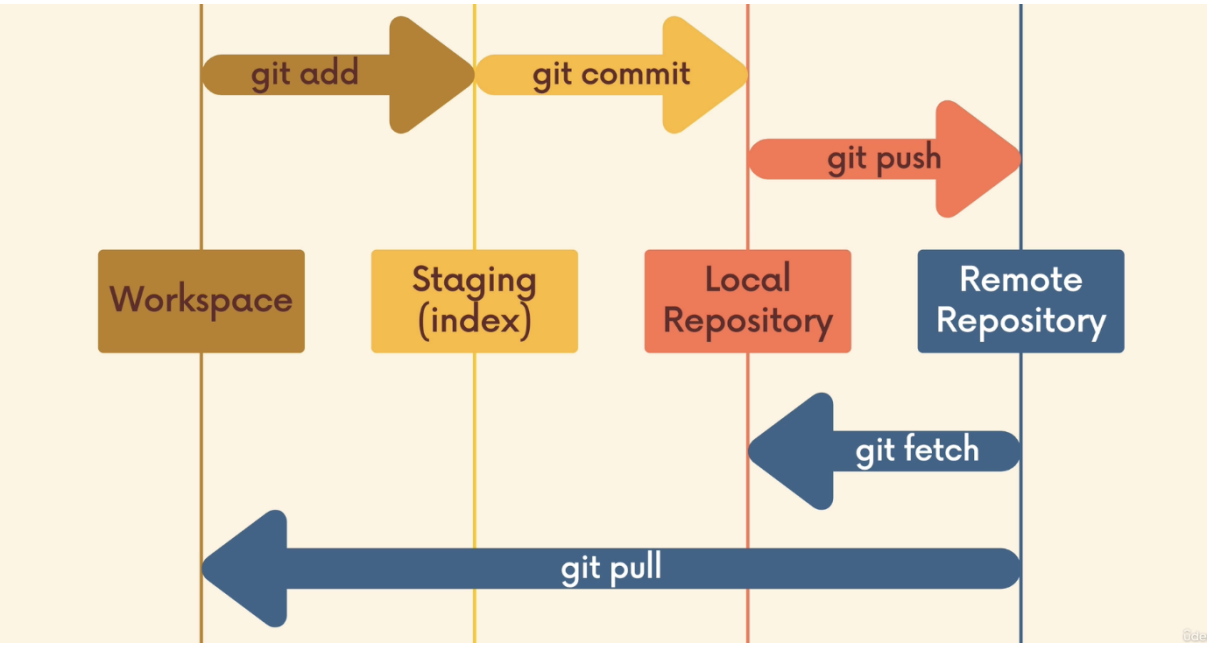
Git Fetch

Example when to use


Github
master
Local
origin/master


Github
Local
origin/master
master





Fetching

Fetching allows us to download changes from a remote repository, But those changes will not be automatically integrated into our working files.

It lets you see what others have been working on, without having to merge those changes into your local repo.

Think of as “please go and get the latest information from Github, but don’t screw up my working directory”.

Git Fetch

The **git fetch <remote-name>** command fetches branches and history from a specific remote repository. It only updates remote tracking branches.

```
git fetch <remote>
```

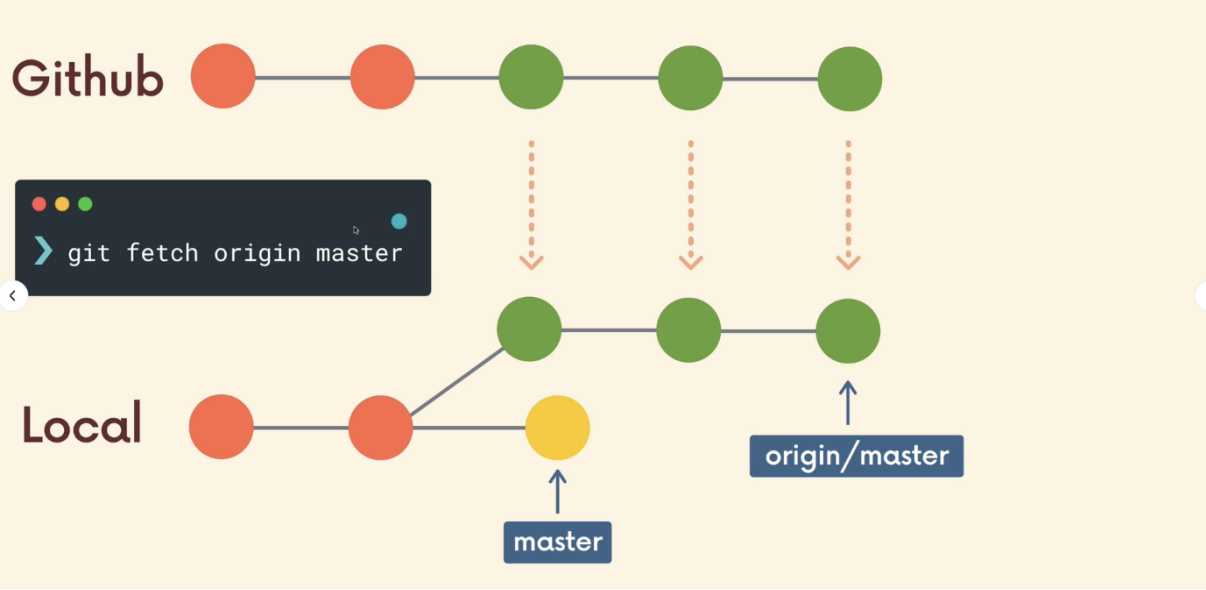
git fetch origin would fetch all changes from the origin remote repository.

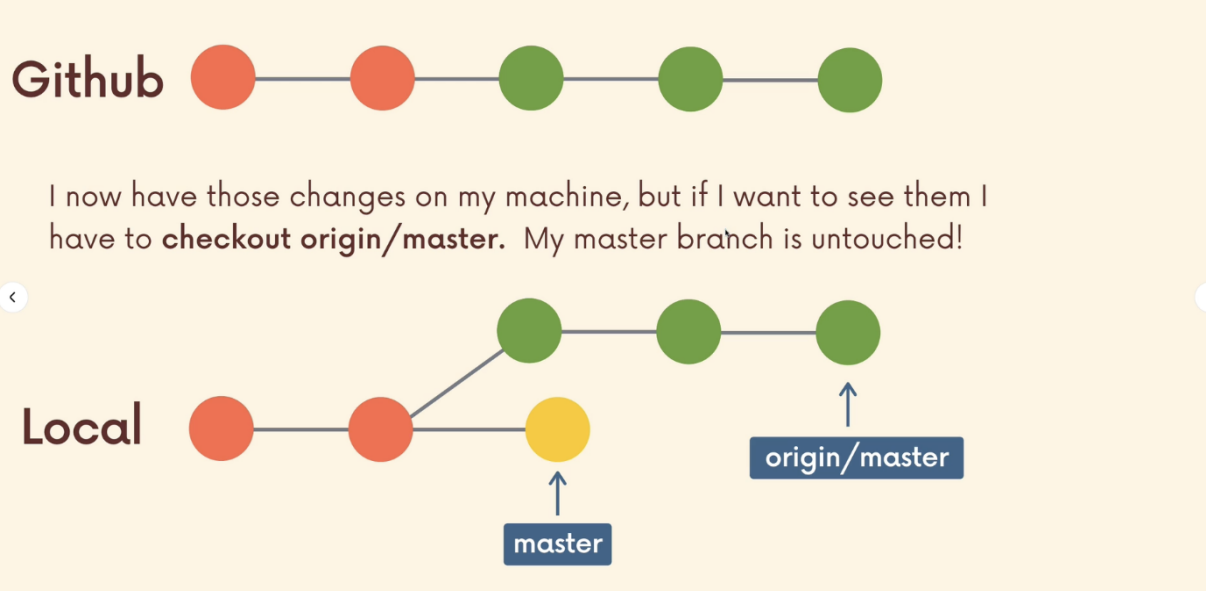
We can also fetch a specific branch from a remote using **git fetch <remote-name> <branch-name>**

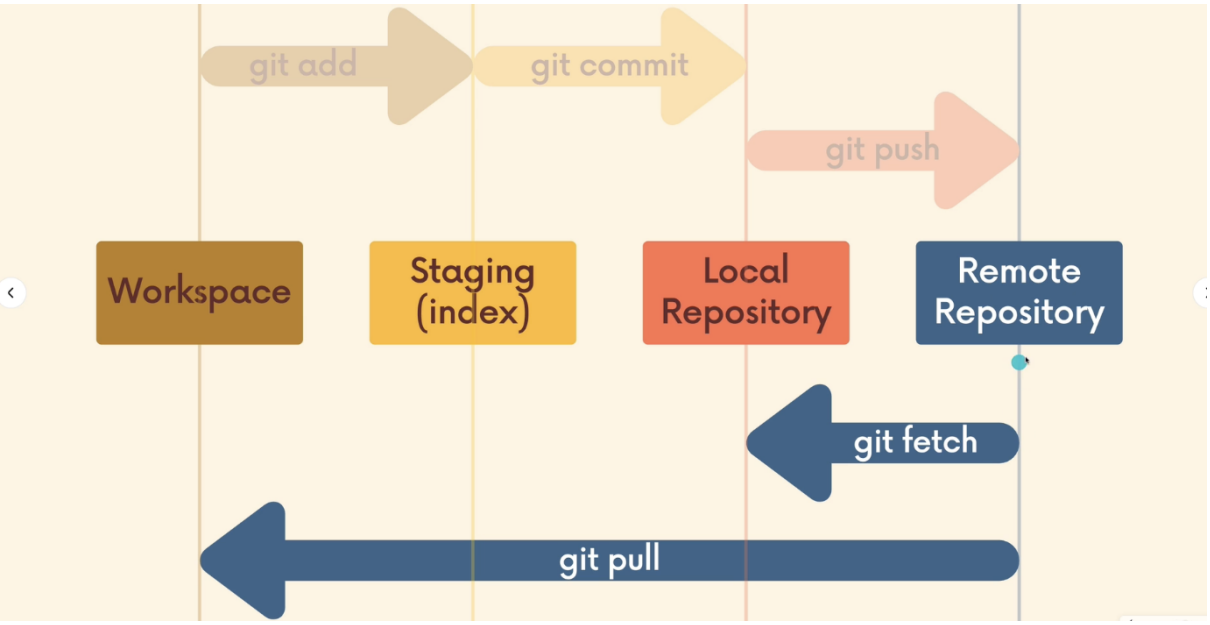
```
git fetch <remote-name> <branch-name>
```

For example, **git fetch origin master** would retrieve the latest information from the master branch on the origin remote repository.

Hence:







Pulling

git pull is another command we can use to retrieve changes from a remote repository. Unlike fetch, pull actually updating a remote repository. Unlike fetch, pull actually updates our HEAD branch with whatever changes are retrieved from the remote.

“go and download data from Github AND immediately update my local repo with those changes.”

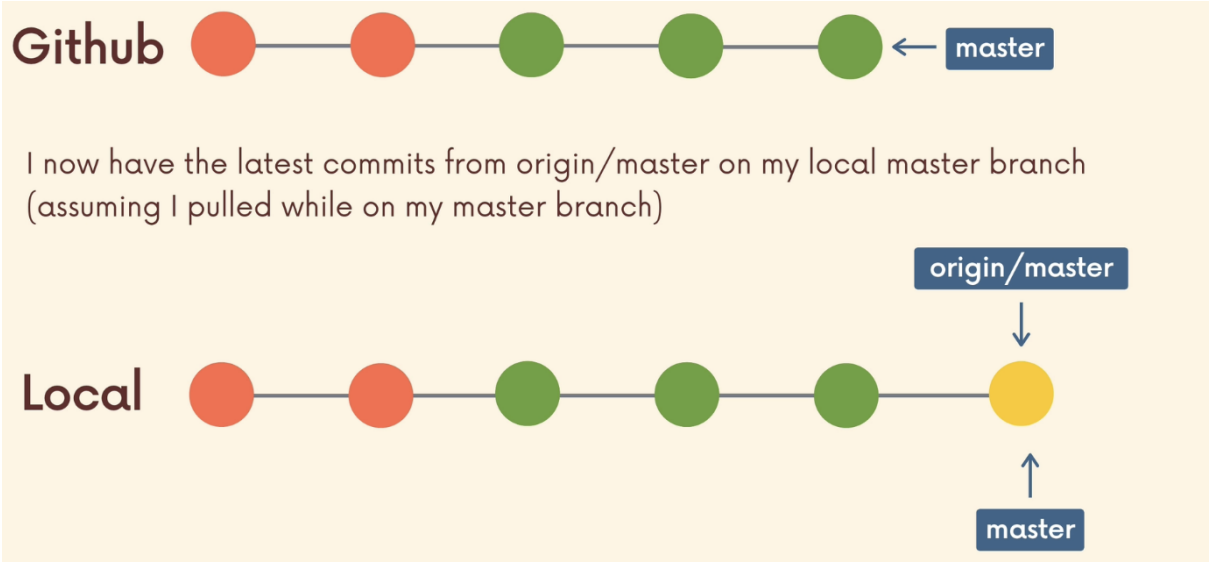
Git Pull

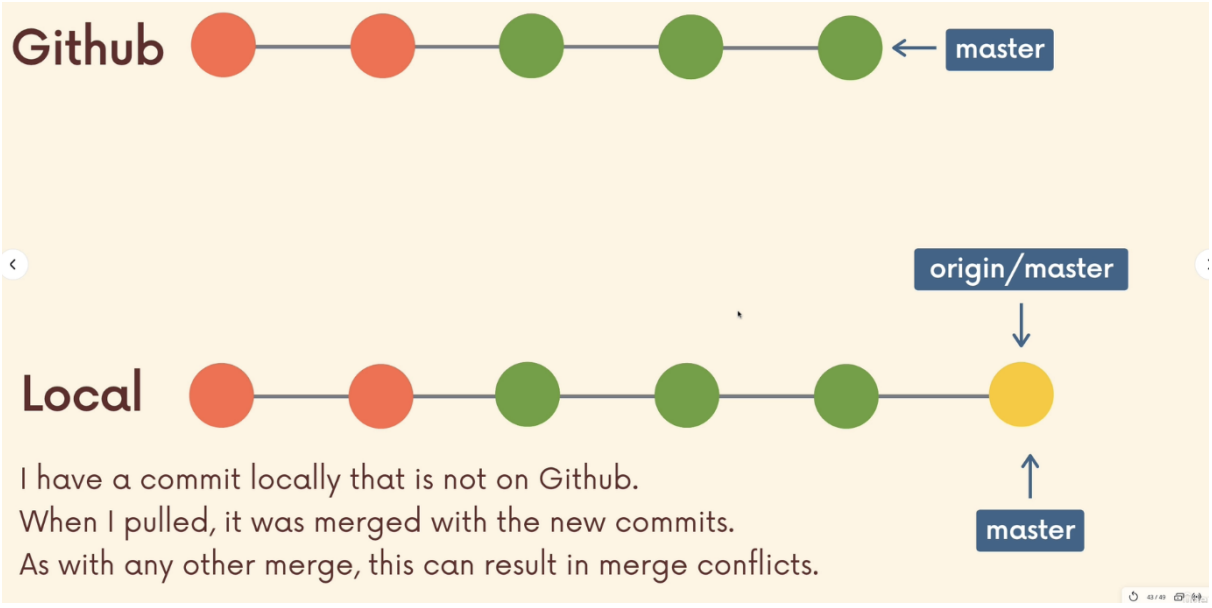
To pull, we specify the particular remote and branch we want to pull using **git pull <remote> <branch>**. Just like with git merge, it matters WHERE we run this command from. Whatever branch we run it from is where the changes will be merge into.

```
git pull <remote-name> <branch-name>
```

git pull origin master would fetch the latest information from the origin’s master branch and merge those changes into our current branch.

Note: pulls can result in merge conflicts.





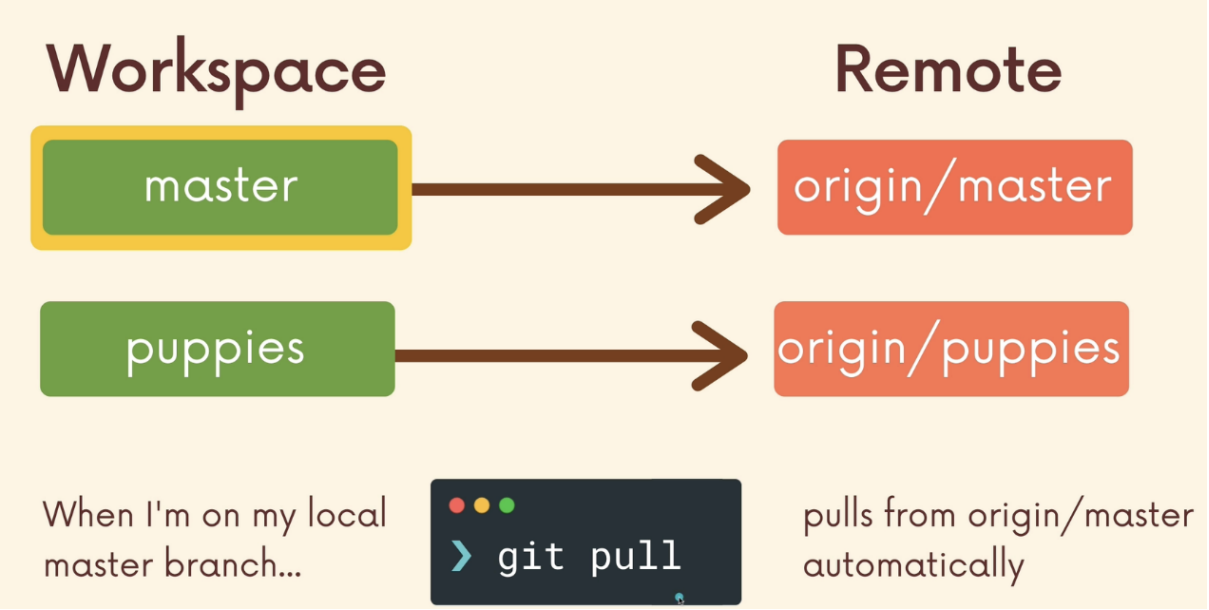
An even easier syntax!

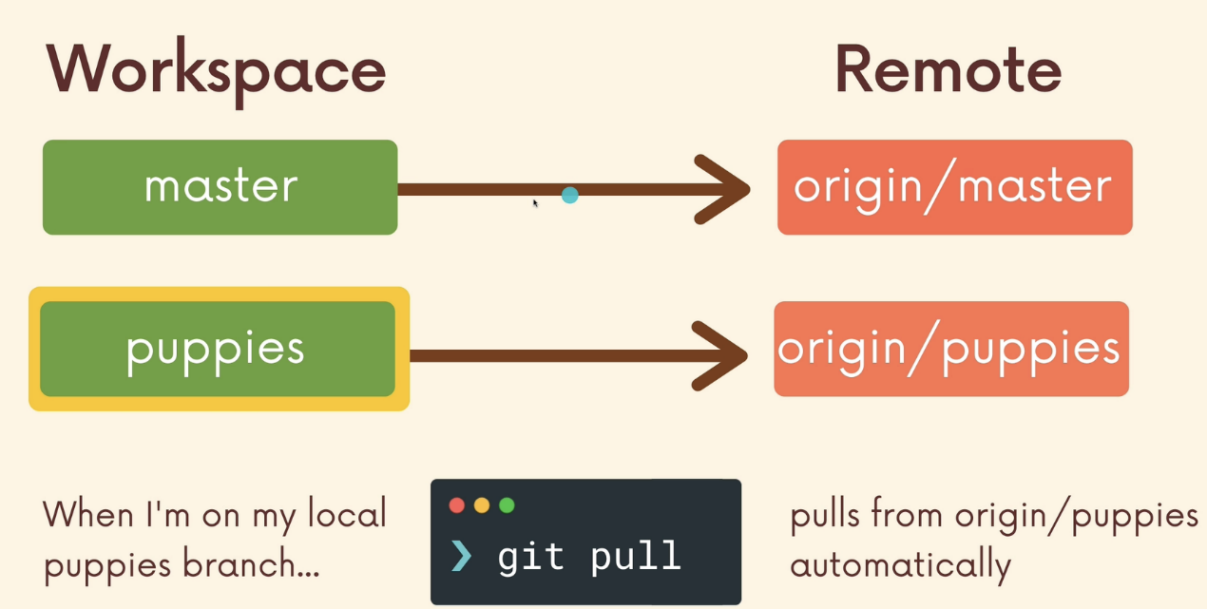
If we run **git pull** without specifying a particular remote or branch to pull from, git assumes the following:

- remote will default to origin
- branch will default to whatever tracking connection is configured for your current branch.

Note: this behavior can be configured, and tracking connections can be changed manually. Most people dont mess with that stuff

```
> git pull
```





Public Vs. Private Repos

Public Repos

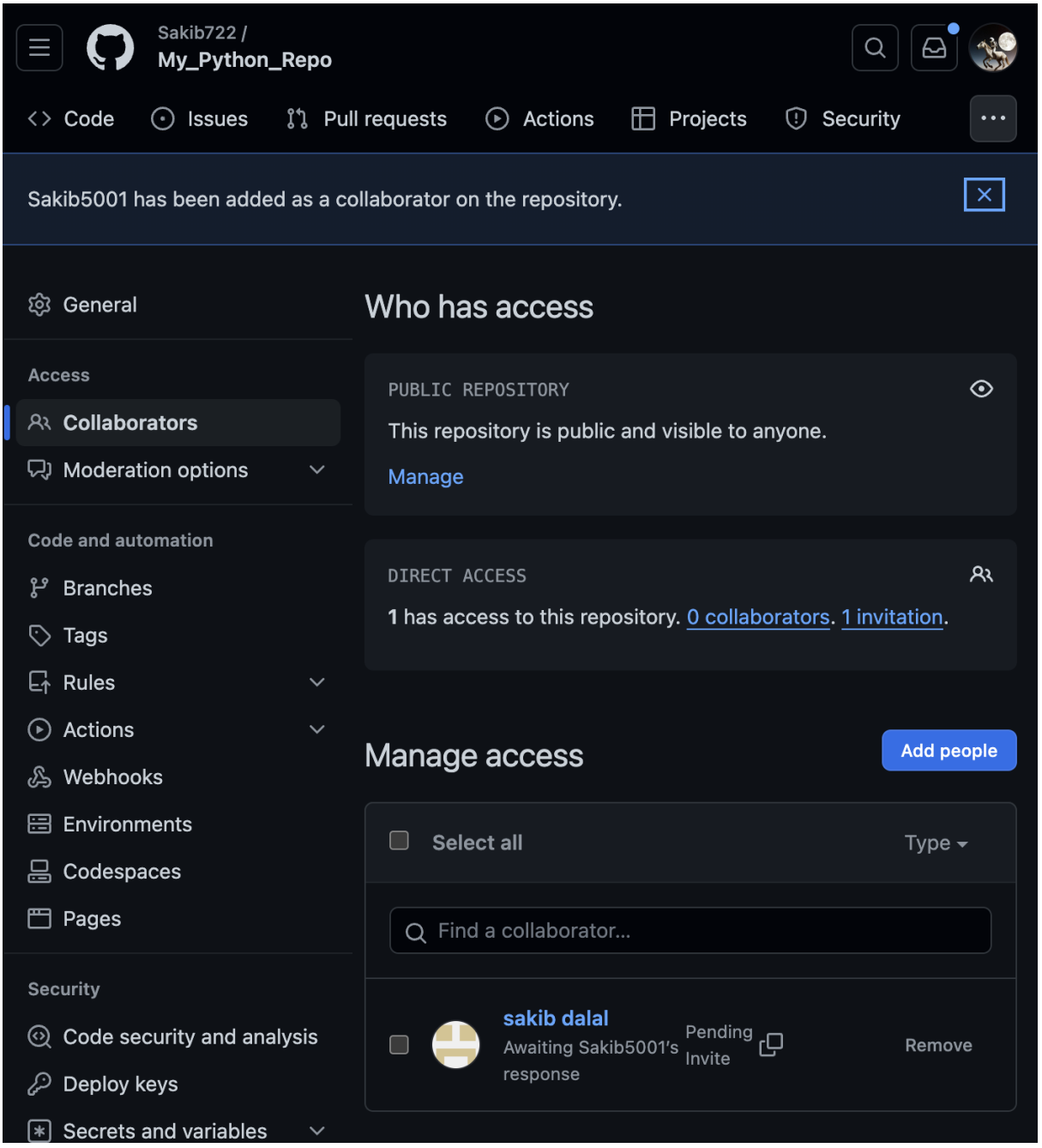
Public repos are accessible to everyone on the internet. Anyone can see the repo on Github.

Private Repos

Private repos are only accessible to the owner and people who have been granted access.

404 error will show up if the repository is been private by owner Github account.

Adding Collaborators To Repos



Go to the specific repo setting and select collaborator's. In collaborators select add people, type the username or email you want to allow access to the repo.

READMEs

A README file is used to communicate important information about a repository including:

- What the project does
- How to run the project
- Why it's noteworthy
- Who maintains the project

If you put a README in the root of your project, Github will recognise it and automatically display it on the repos home page.

README.md

READMEs are Markdown files, ending with the .md extension. Markdown is a convenient syntax to generate formatted text. It’s easy to pick up.

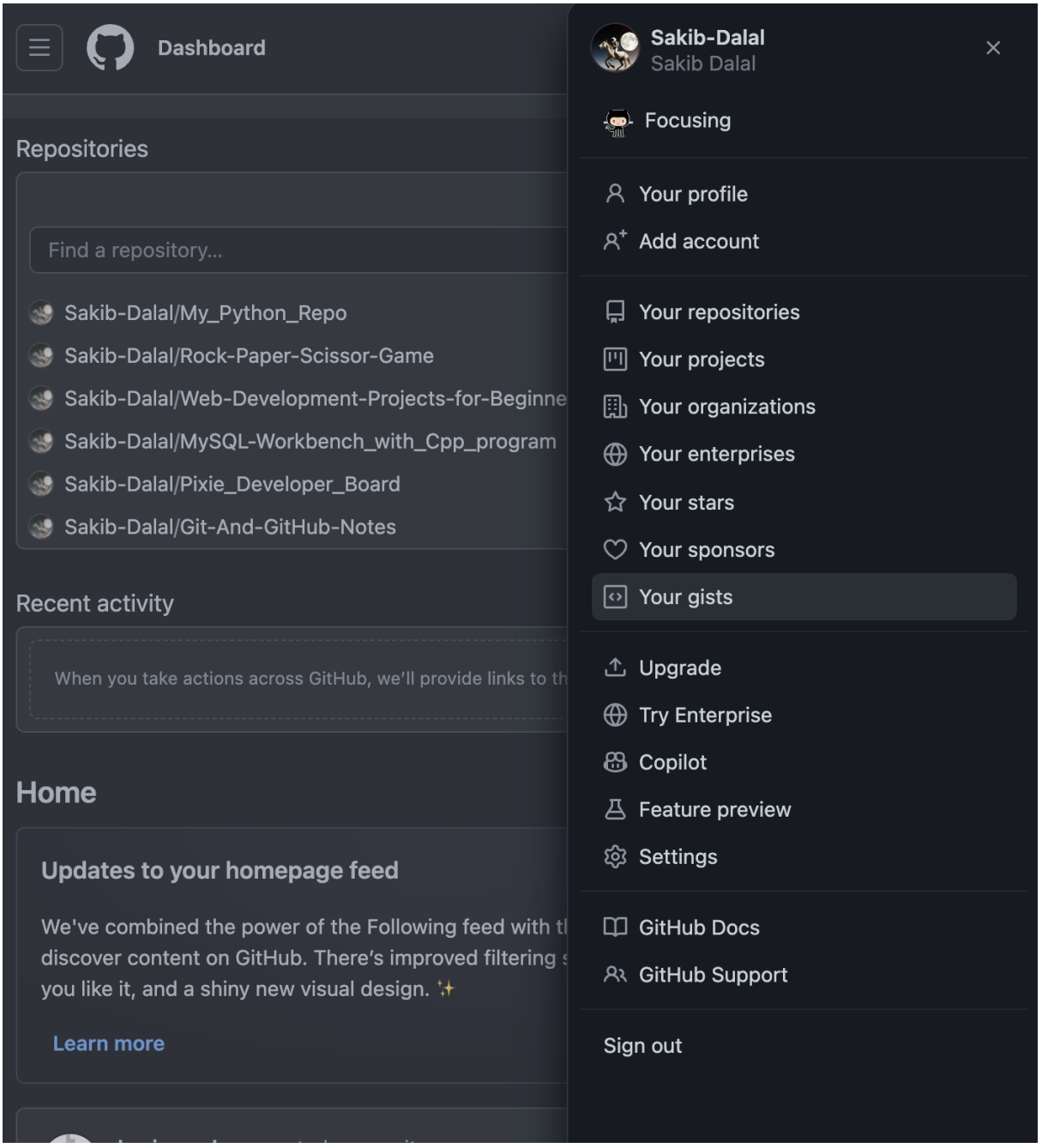
Markdown:

> markdown-it demo
>
> https://markdown-it.github.io/

```
touch README.md
# with .md extinction as Markdown
```

Github Gists

Github Gists are a simple way to share code snippets and useful fragments with others. Gists are much easier to create, but offer far fewer features than a typical Github repository.



Select “Your gists”

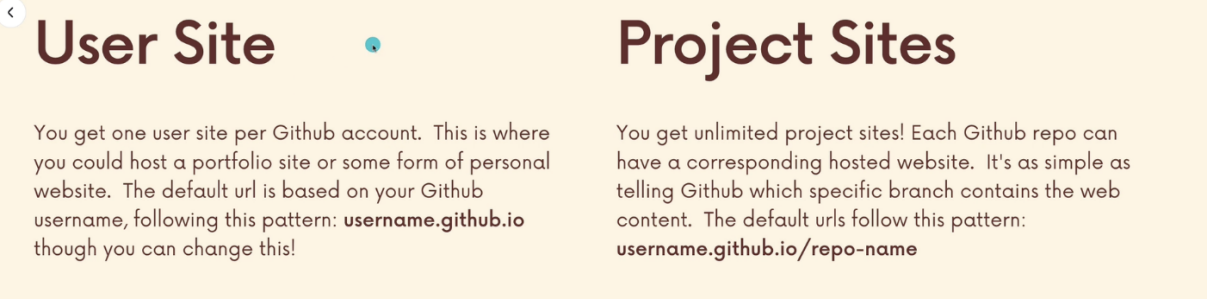
Create your own gists and share or keep it private.

GitHub Pages

Github Pages are public webpages that are hosted and published via Github. They allow you to create a website simply by pushing your code to Github.

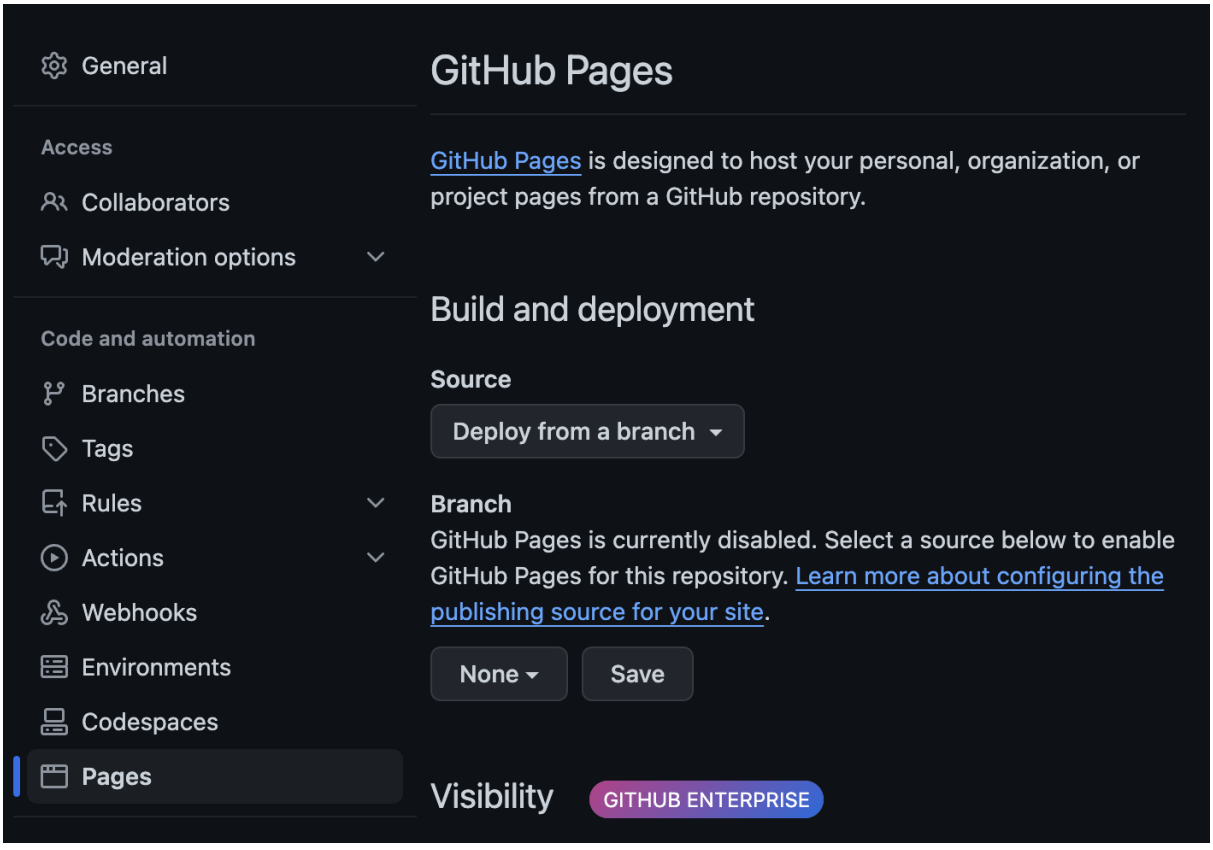
Github pages is a hosting service for static webpages, so it does not support server-side code like Python, Ruby Just HTML/CSS/JS.

web pages will be having domain as .**github.io** we can customise.

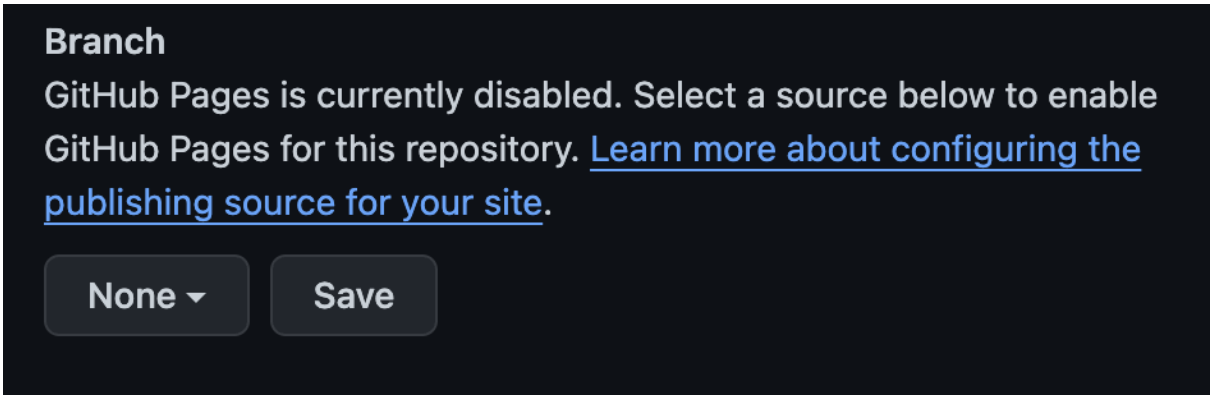


Steps:

- Make a new repository.
- Make local machine repo.
- Include a **Index.html** file on the repo to be displayed on Github.
- got to Github and select Github Pages from **Repository Settings.**
- Select Pages



- In pages select the branch



- Select main branch.
- open the url provided by Github.

Git Collaborative Workflows

Centralised Workflow

AKA Everyone Works On Master/Main

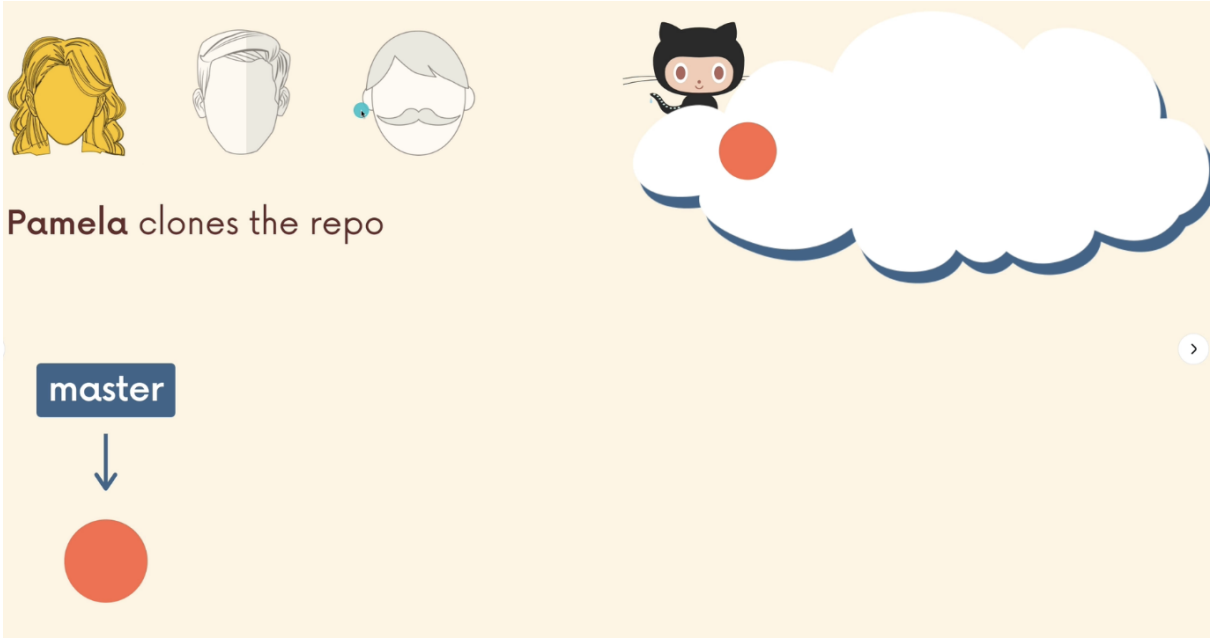
AKA The Most Basic Workflow Possible

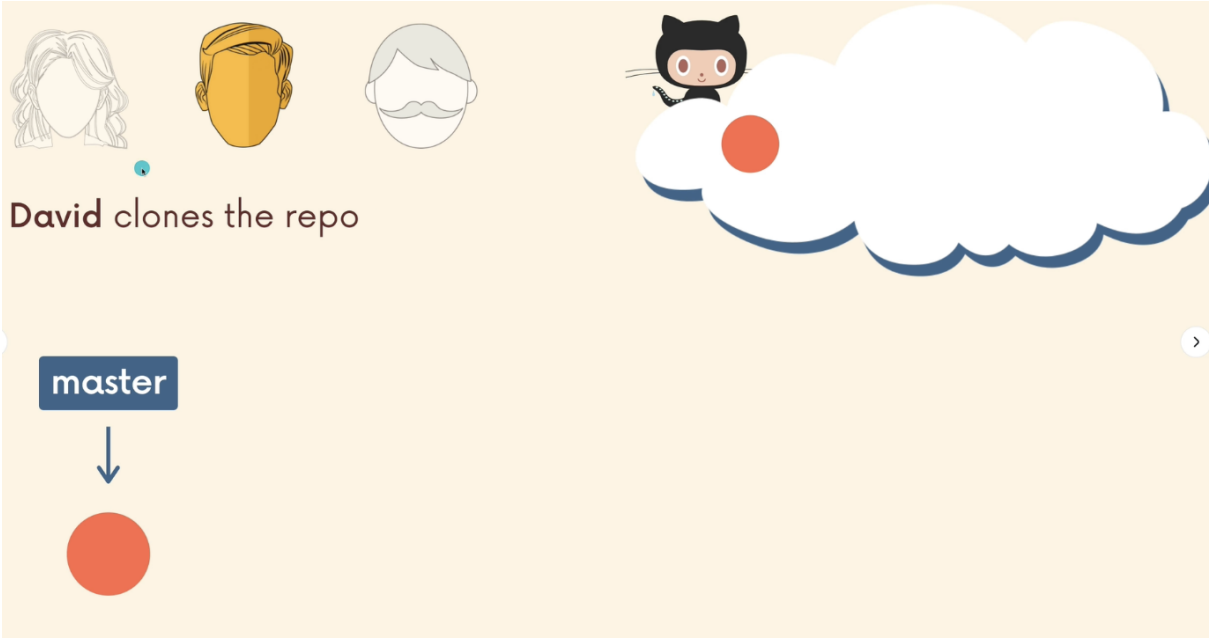
The simplest collaborative workflow is to have everyone work on the master branch (or main, or any other SINGLE branch).

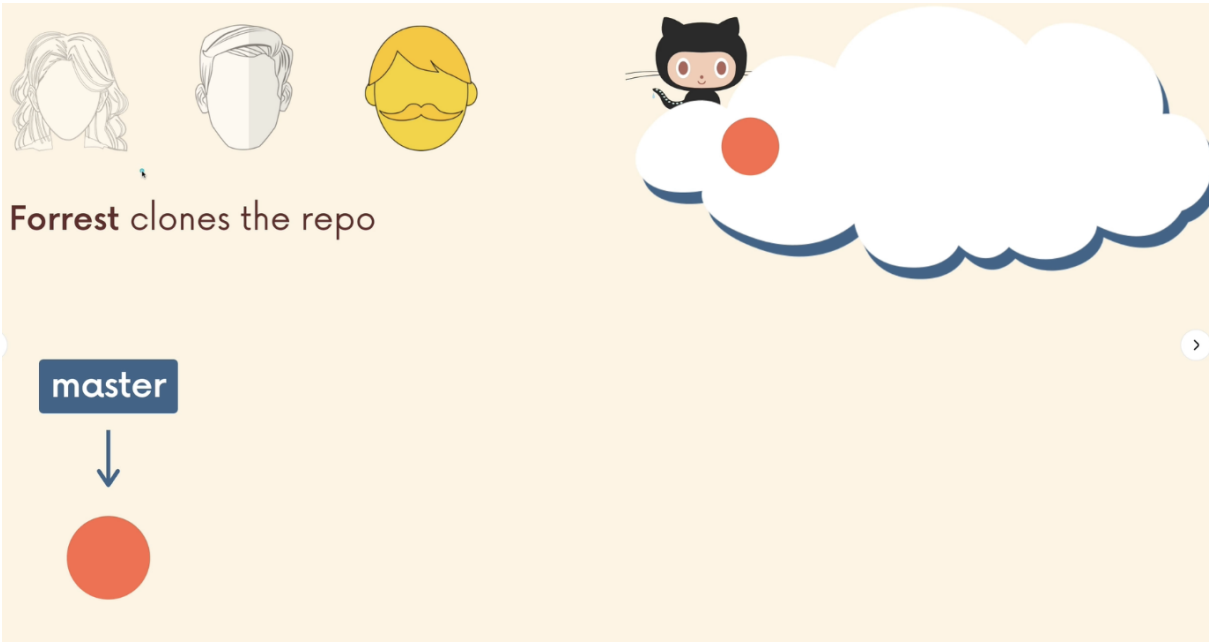
It’s straightforward and can work for tiny teams, but it has quite a few shortcomings!

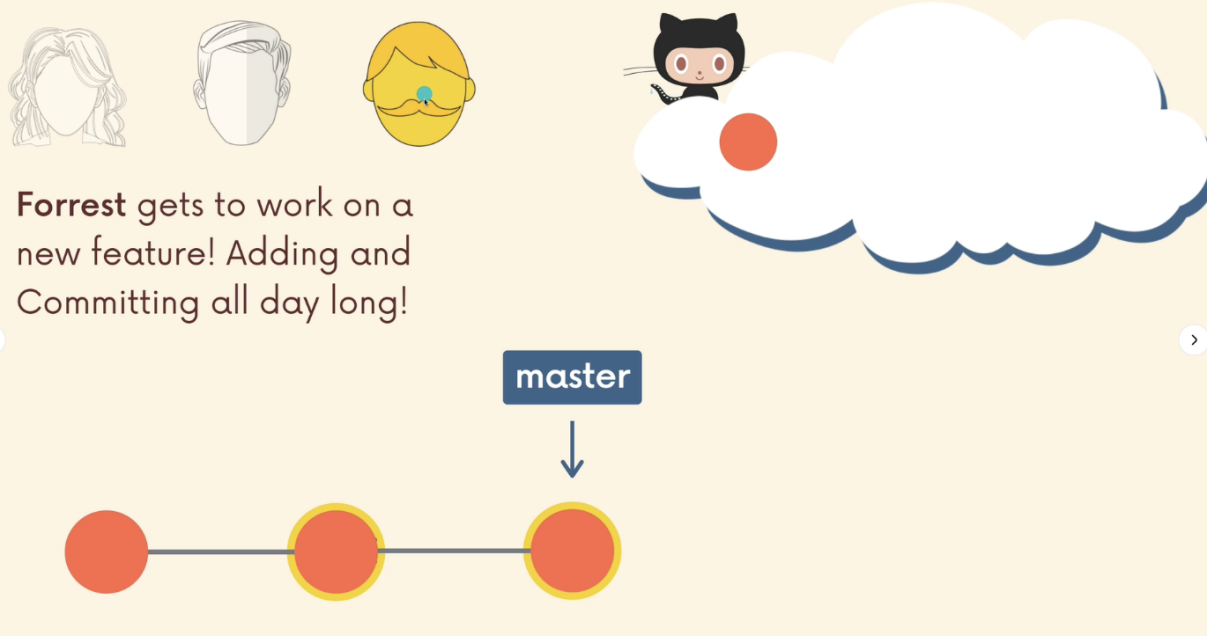
Example:

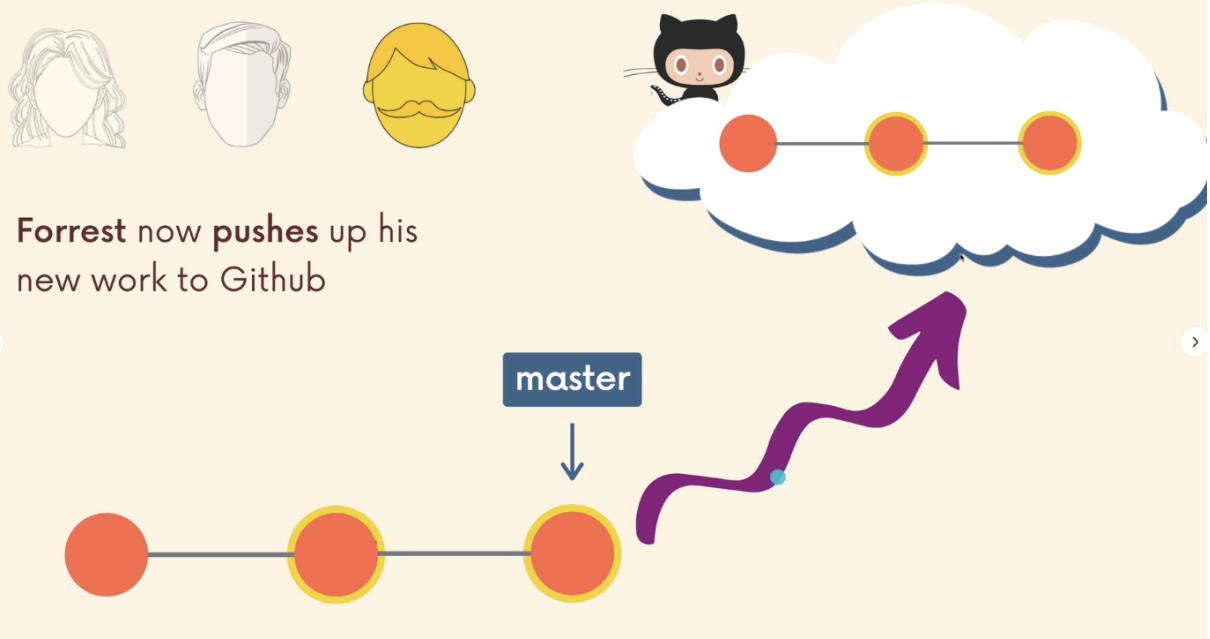
We have three different collaborations.

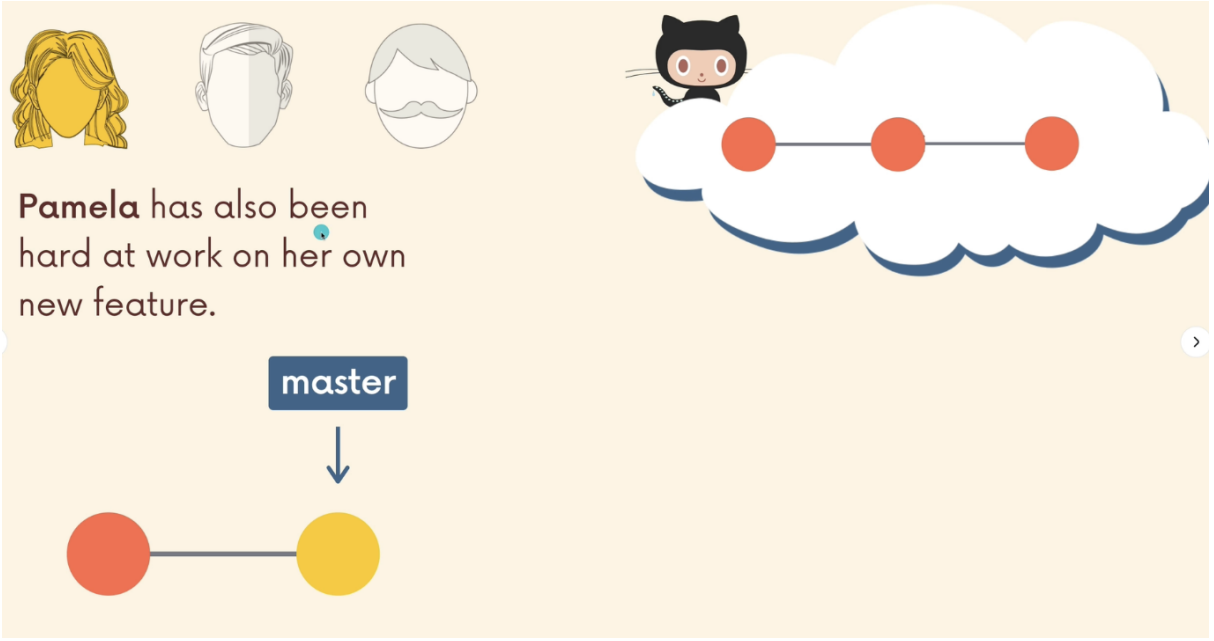


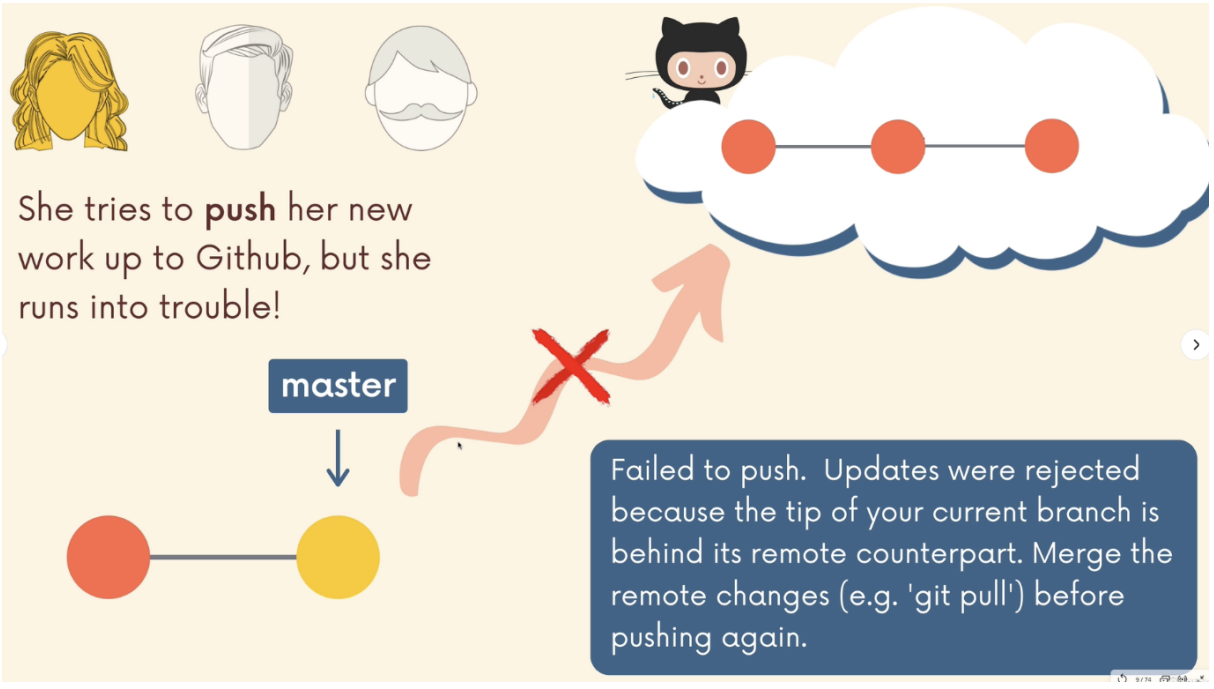

David clones the repo
master

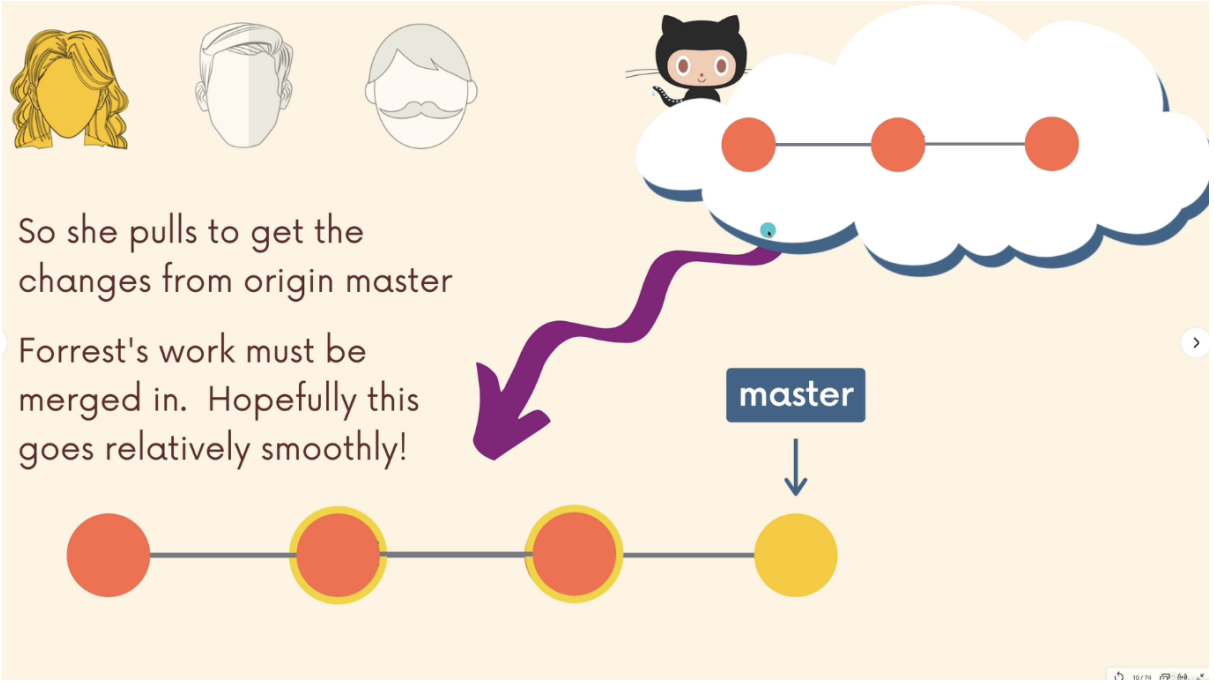

Forrest clones the repo
master

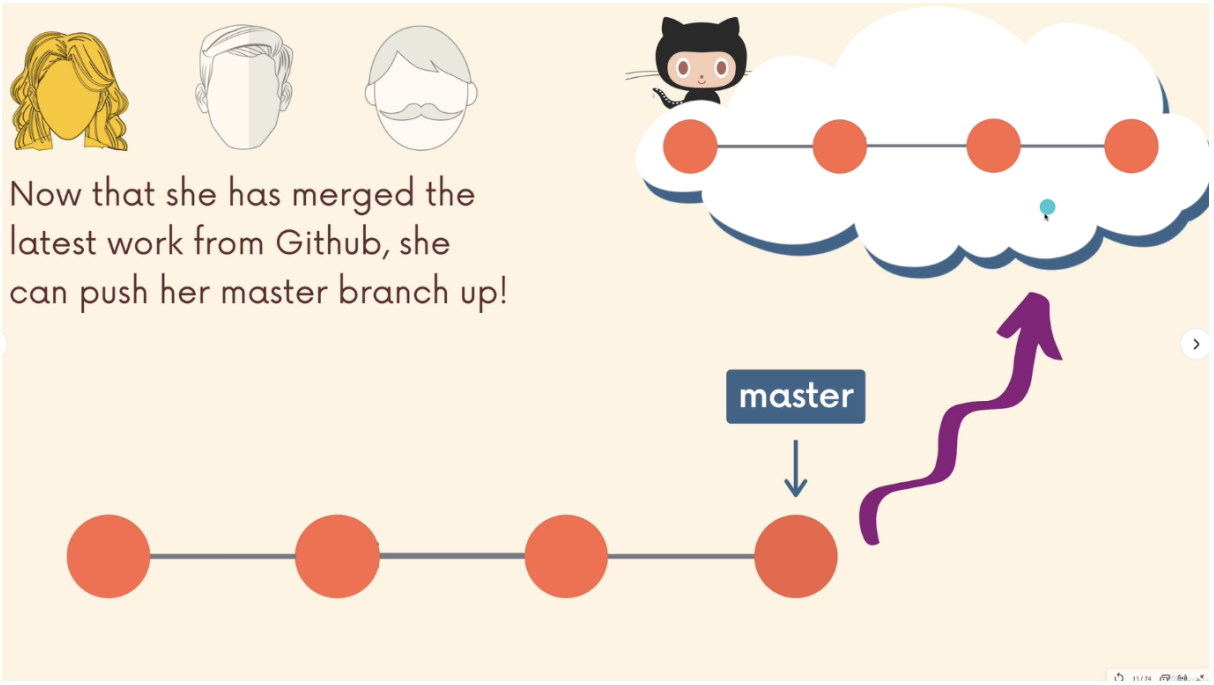

Forrest gets to work on a new feature! Adding and Committing all day long!
master

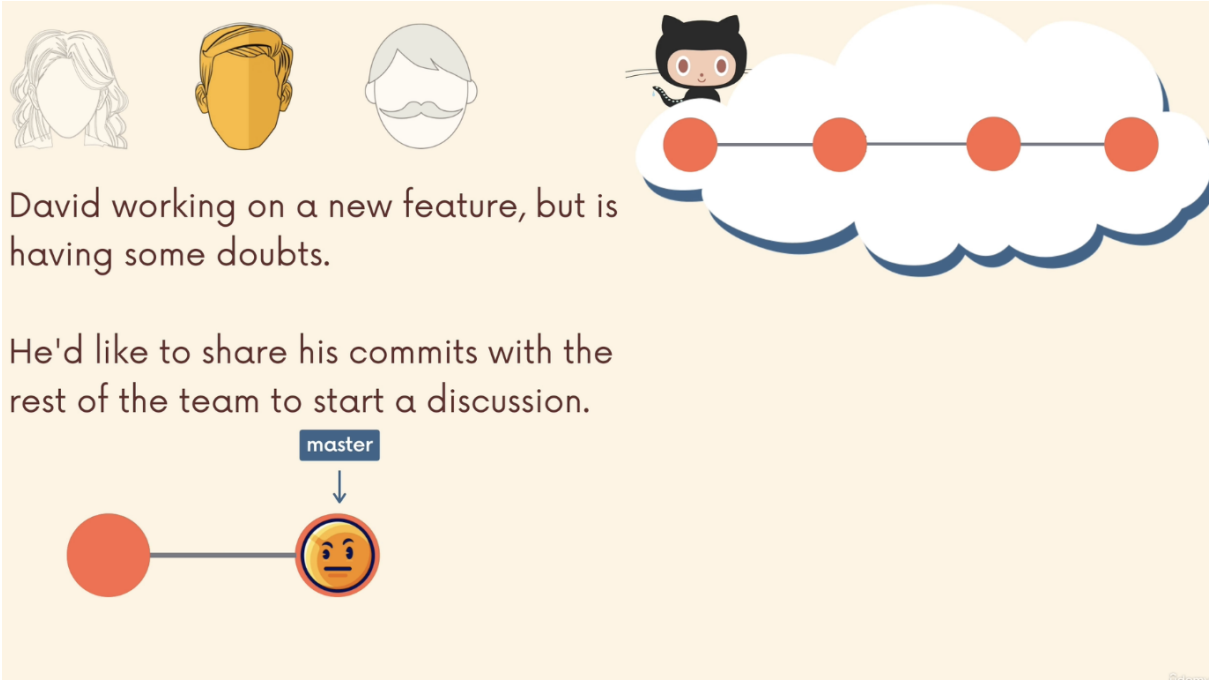

Forrest now pushes up his new work to Github
master

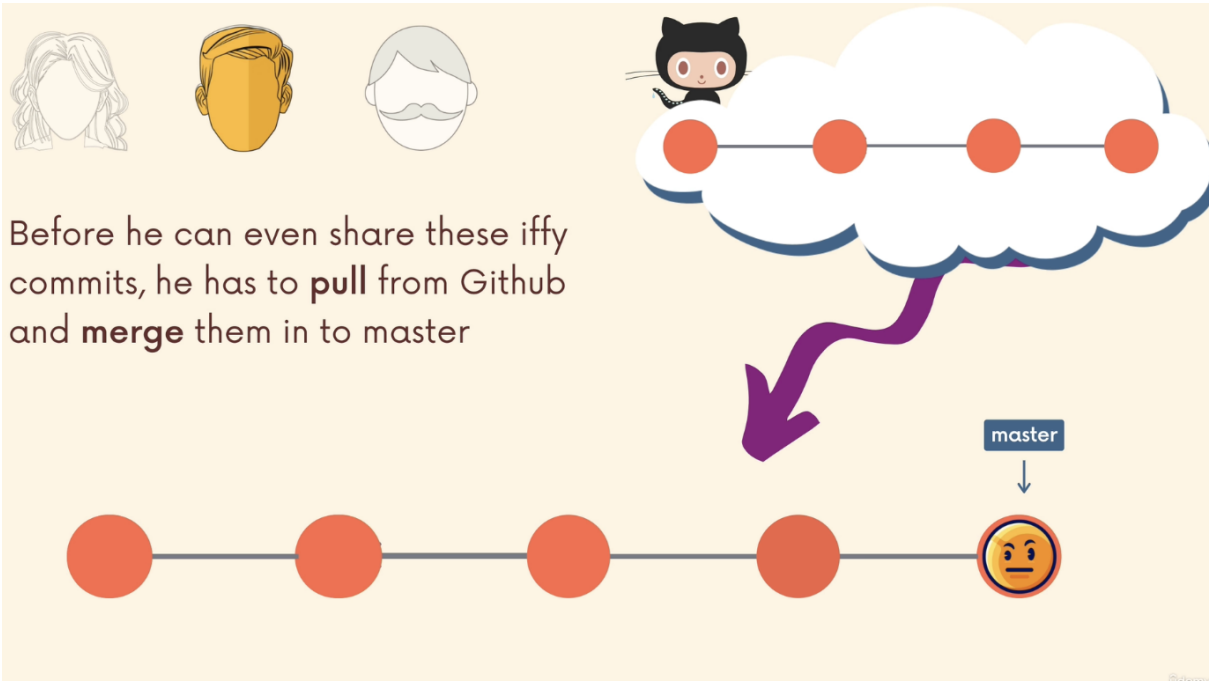

Pamela has also been hard at work on her own new feature.
master

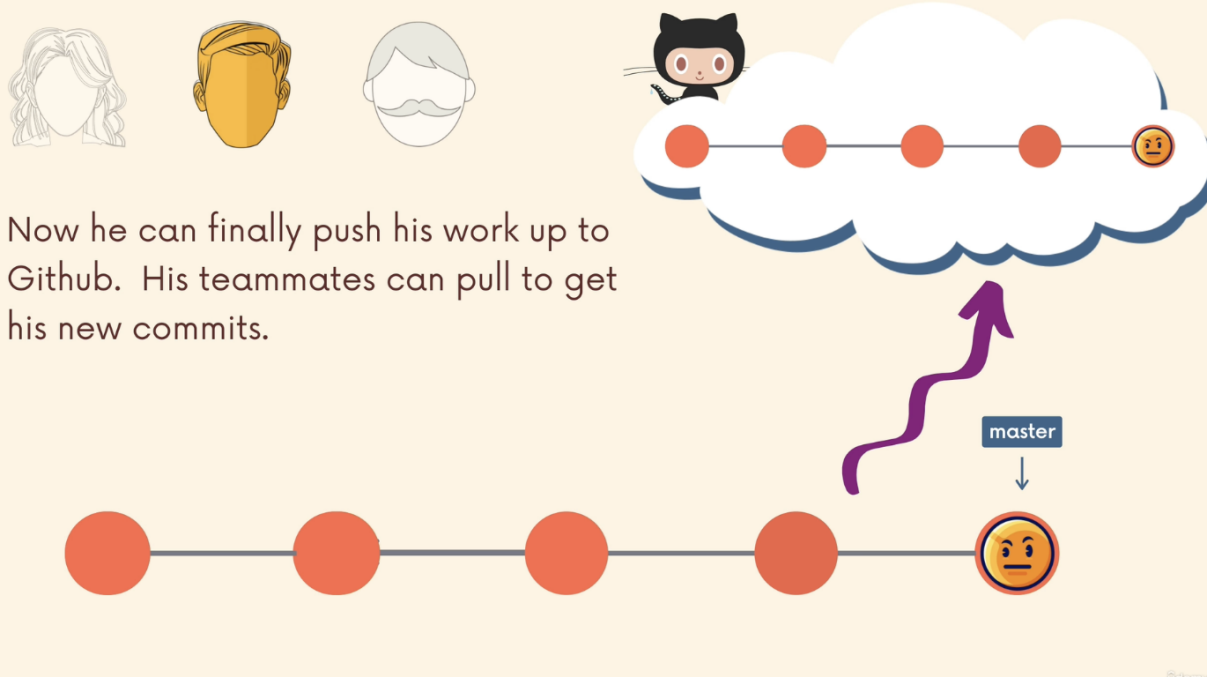

She tries to push her new work up to Github, but she runs into trouble!
master
Failed to push. Updates were rejected because the tip of your current branch is behind its remote counterpart. Merge the remote changes (e.g. 'git pull') before pushing again.


So she pulls to get the changes from origin master
Forrest's work must be merged in. Hopefully this goes relatively smoothly!
master


Now that she has merged the latest work from Github, she can push her master branch up!
master


David working on a new feature, but is having some doubts.
He'd like to share his commits with the rest of the team to start a discussion.
master


Before he can even share these iffy commits, he has to **pull** from Github and **merge** them in to master
master



The Problem:

While it's nice and easy to only work on the master branch, this leads to some serious issues on teams!

- Lots of time spent resolving conflicts and merging code, especially as team size scales up.
- No one can work on anything without disturbing the main codebase. How do you try adding something radically different in? How do you experiment?
- The only way to collaborate on feature with another teammate is to push incomplete code to master. Other teammates now have broken code.

Note: Use **git stash** before git pull a repository into local machine.

Enter Feature Branches

Don't work on master branches work on feature branches.

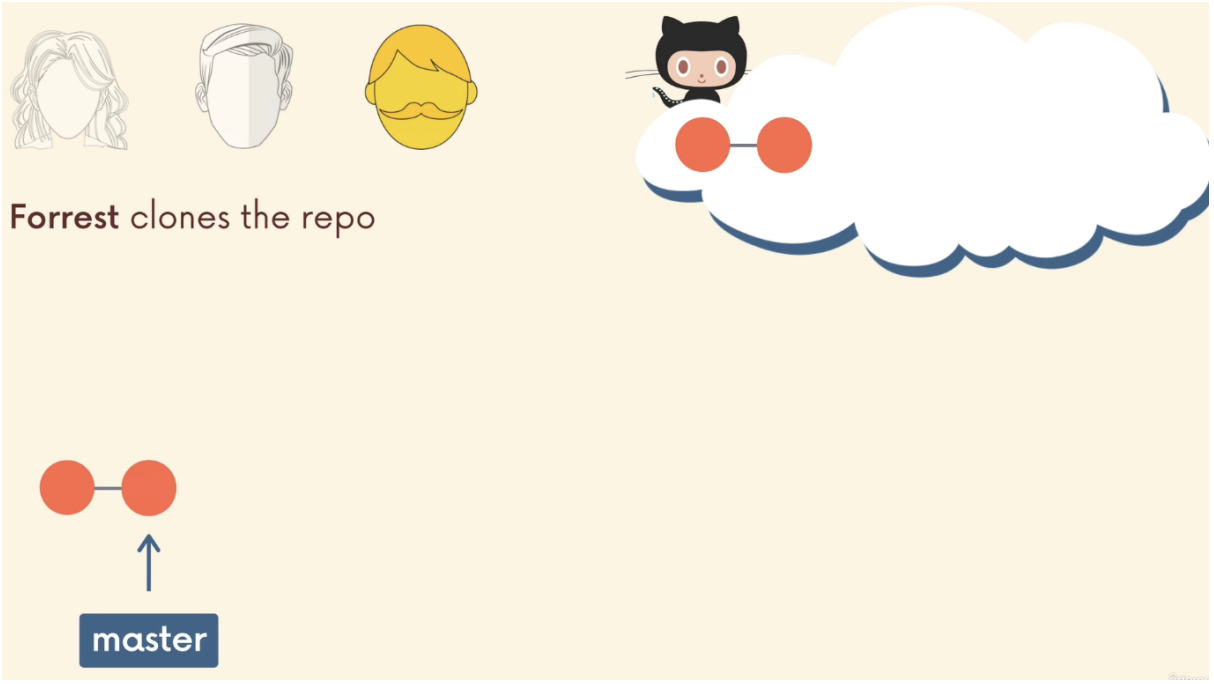
Feature Branch

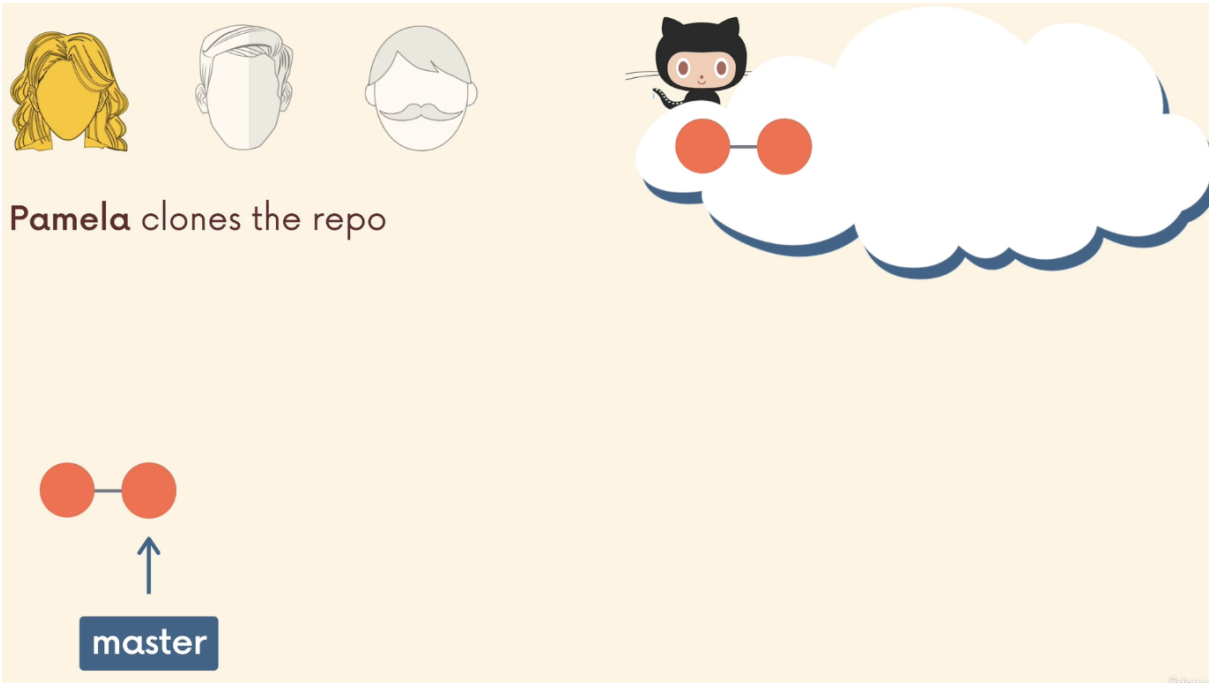
Rather than working directly on master/main, all new development should be done on separate branches!

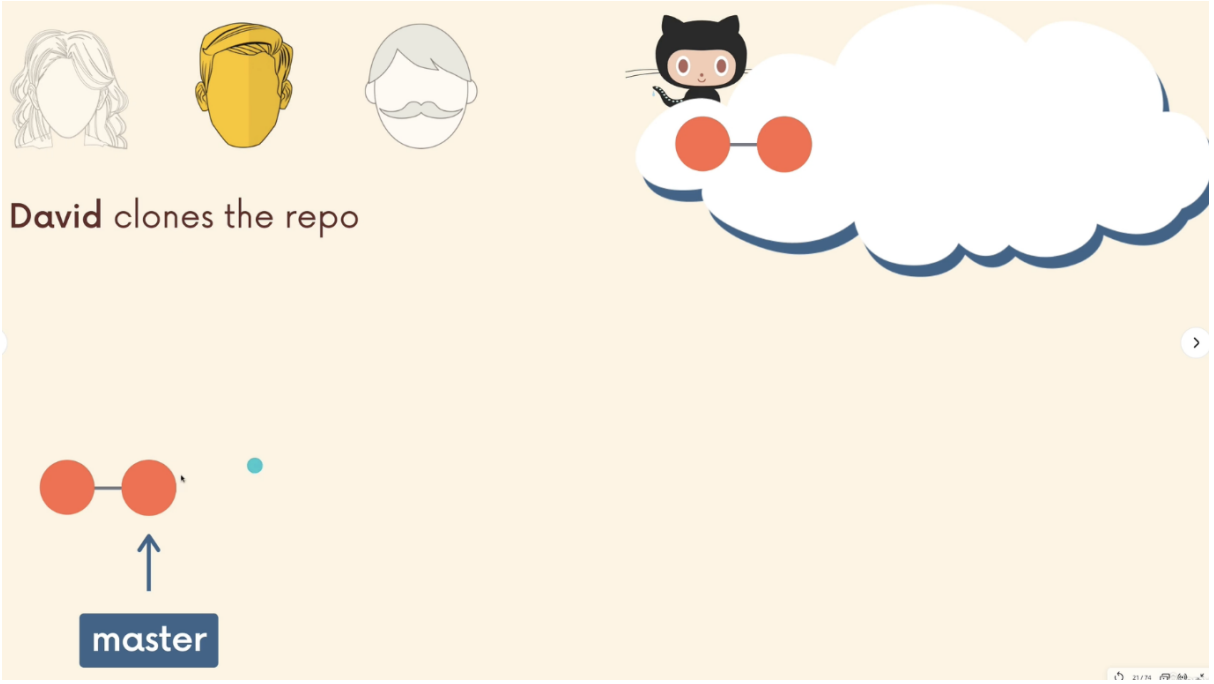
- Treat master/main branch as the official project history
- Multiple teammates can collaborate on single feature and share code back and forth without polluting the master/main branch.

- Master/main branch won’t contain broken code (or at least, it won’t unless someone messes up.

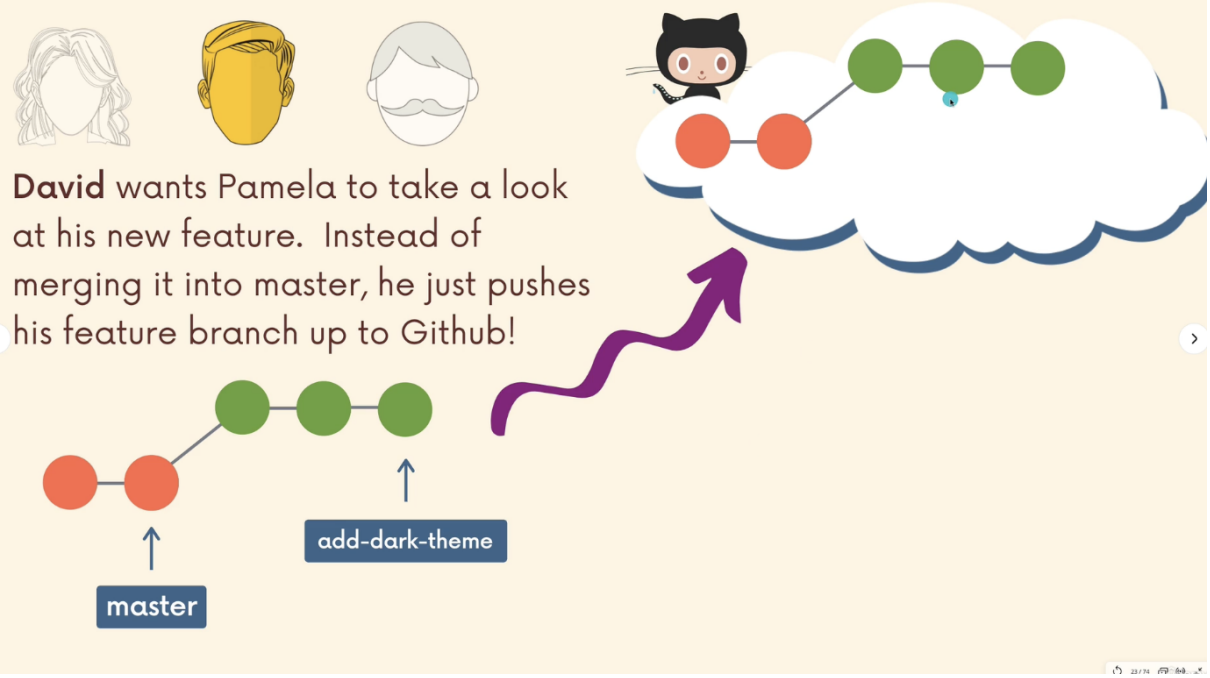
Example:

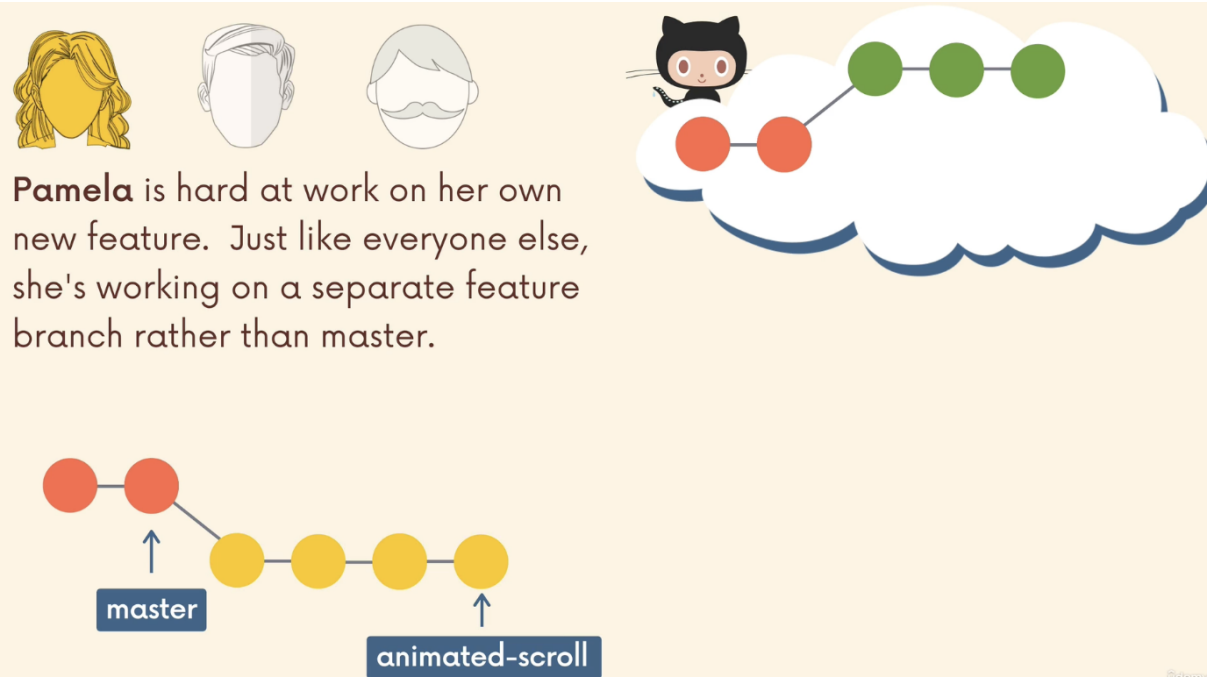


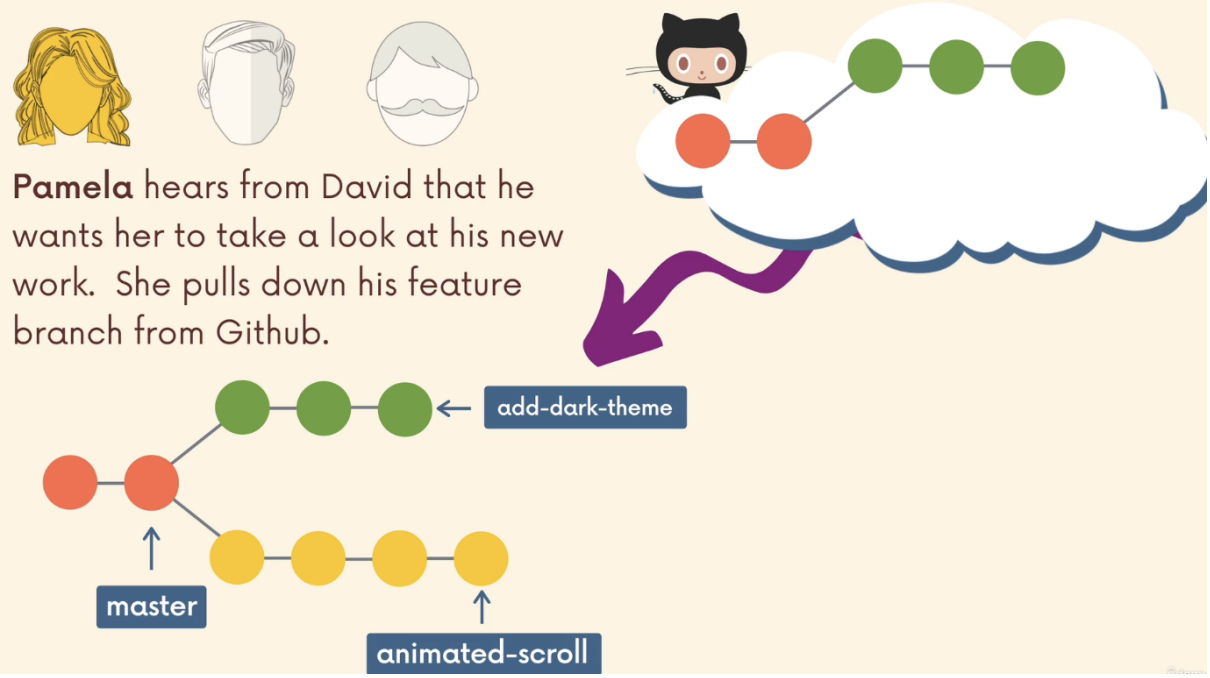


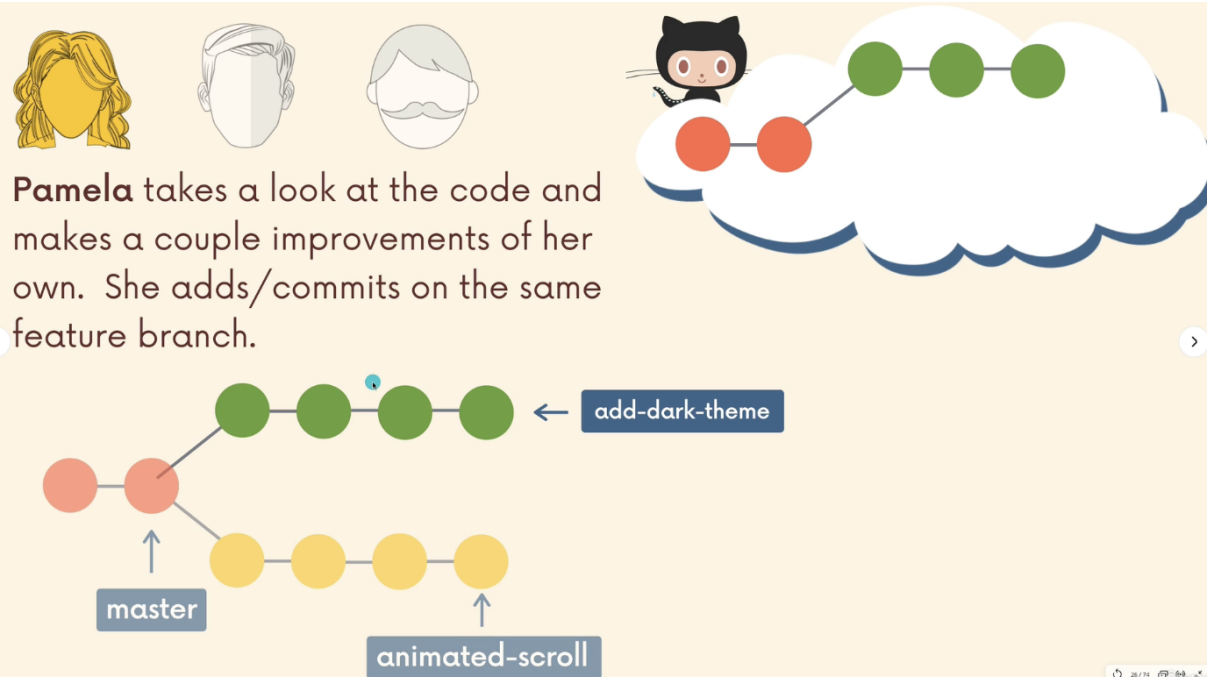

David clones the repo
master

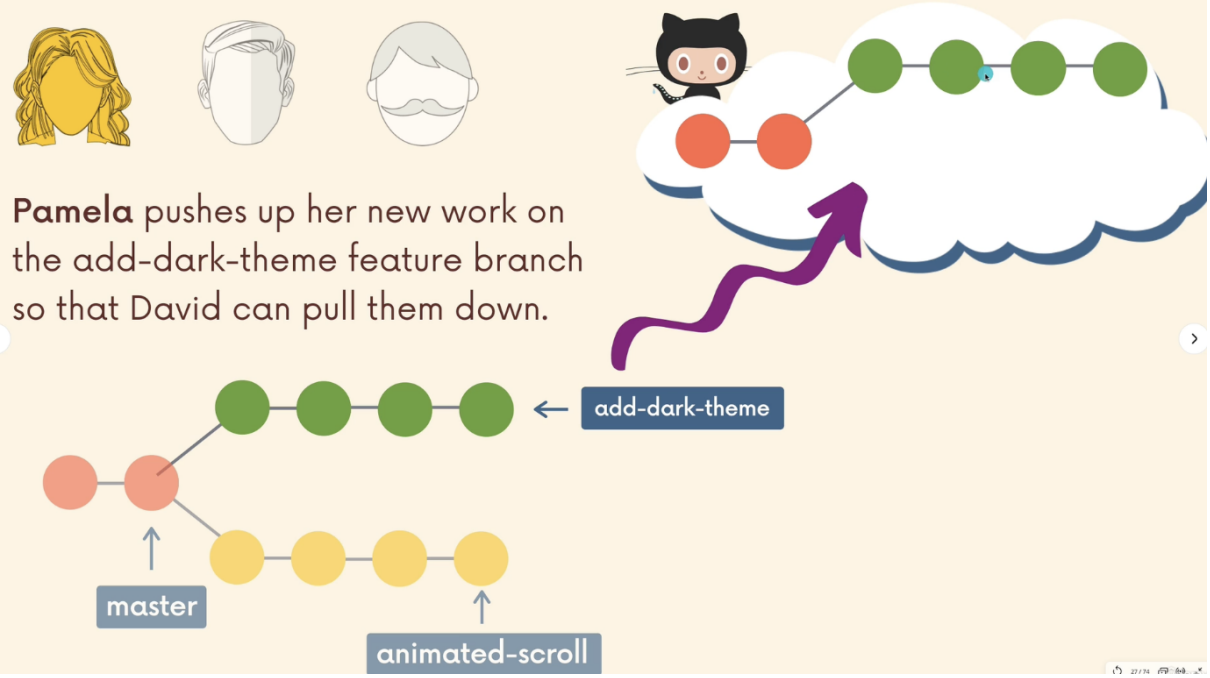

David starts work on a new feature. He does all this work on a separate branch!
add-dark-theme
master


David wants Pamela to take a look at his new feature. Instead of merging it into master, he just pushes his feature branch up to Github!
add-dark-theme
master

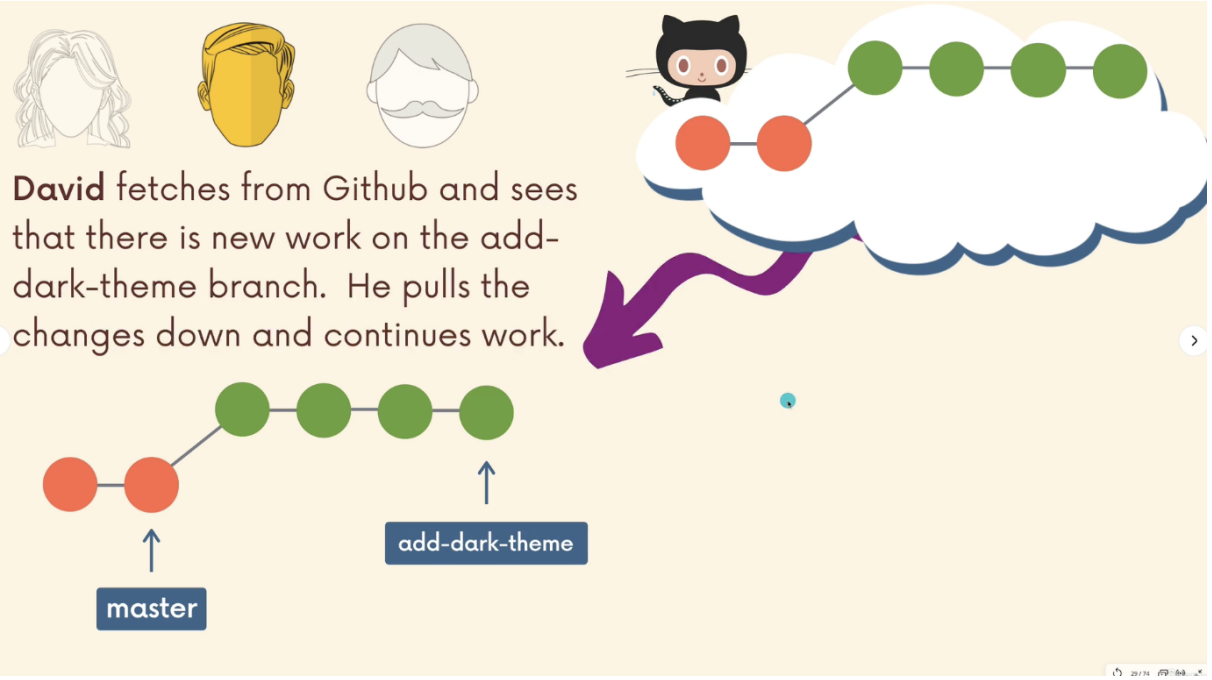

Pamela is hard at work on her own new feature. Just like everyone else, she's working on a separate feature branch rather than master.
master
animated-scroll

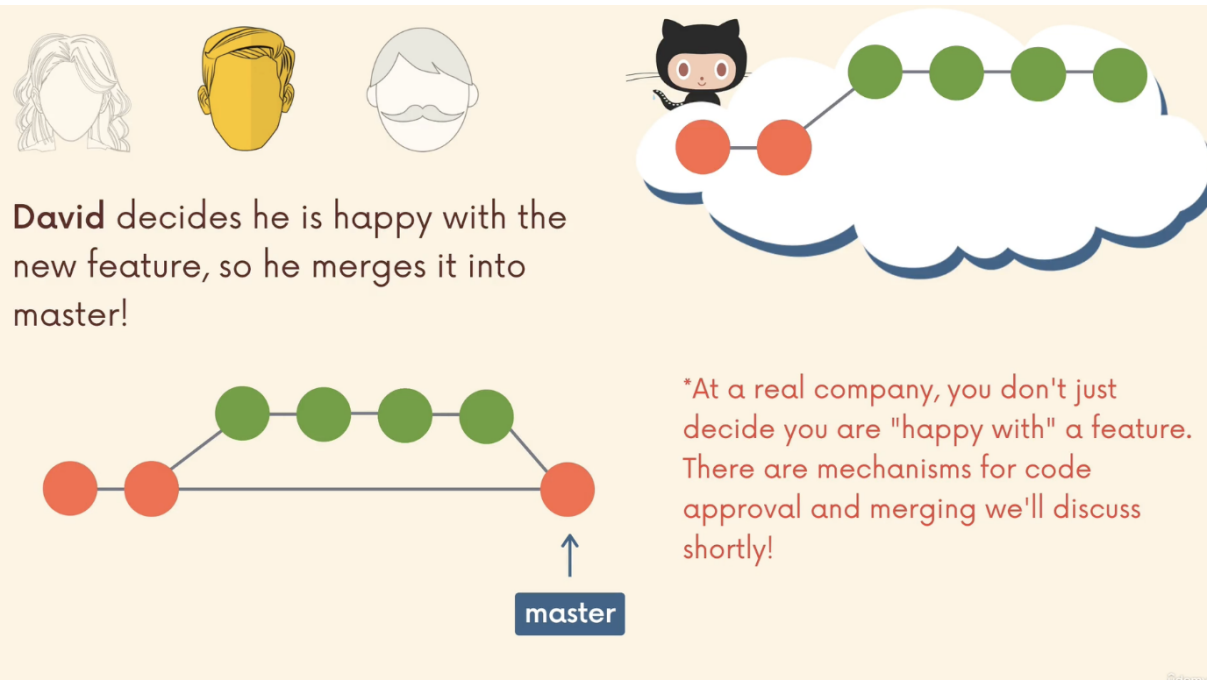

Pamela hears from David that he wants her to take a look at his new work. She pulls down his feature branch from Github.
add-dark-theme
master
animated-scroll

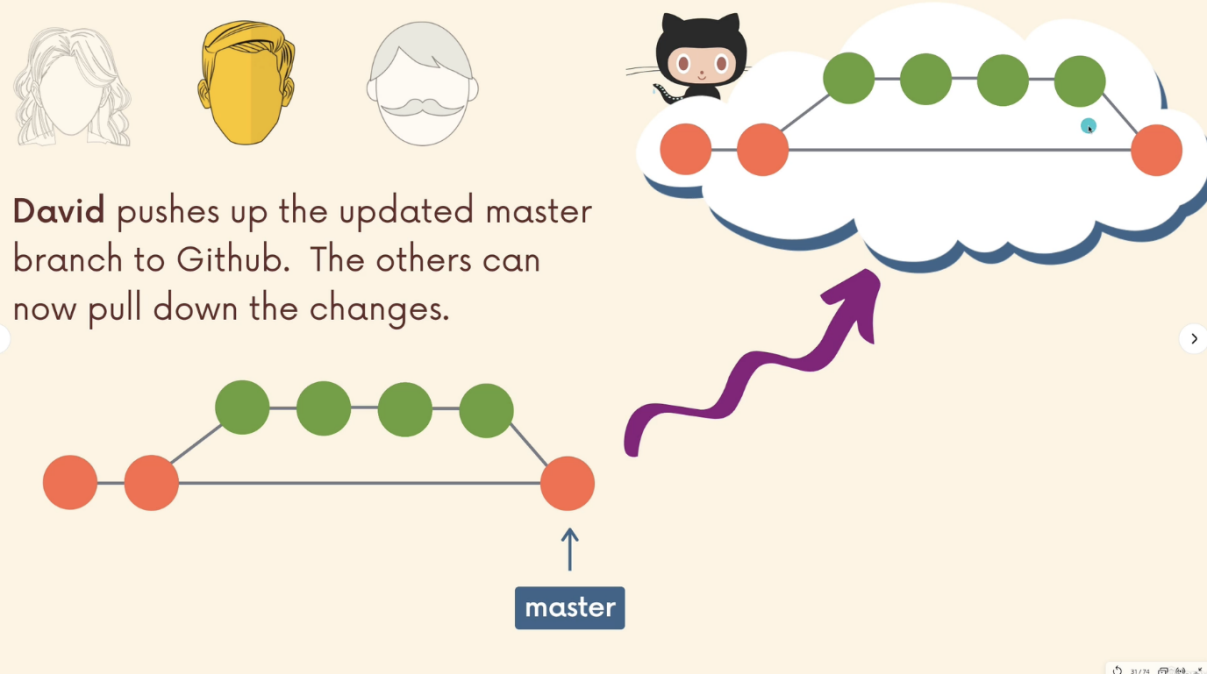

Pamela takes a look at the code and makes a couple improvements of her own. She adds/commits on the same feature branch.
add-dark-theme
master
animated-scroll


Pamela pushes up her new work on the add-dark-theme feature branch so that David can pull them down.
add-dark-theme
master
animated-scroll


David returns to work the next morning. After an hour on reddit he decides he should actually do some work.
add-dark-theme
master


David fetches from Github and sees that there is new work on the add-dark-theme branch. He pulls the changes down and continues work.
add-dark-theme
master


David decides he is happy with the new feature, so he merges it into master!
master
*At a real company, you don't just decide you are "happy with" a feature. There are mechanisms for code approval and merging we'll discuss shortly!



Merging in Feature Branches

At some point new work on feature branches will need to be merged in to the master branch! There are a couple of options for how to do this..

1. Merge at will, without any sort of discussion with teammates. JUST DO IT WHENEVER YOU WANT.
2. Send an email or chat message or something to your team to discuss if the changes should be merged in.
3. Pull Requests!

Pull Requests

Pull Request are a feature built in to products like Github & Bitbucket. **They are not native to Git itself.**

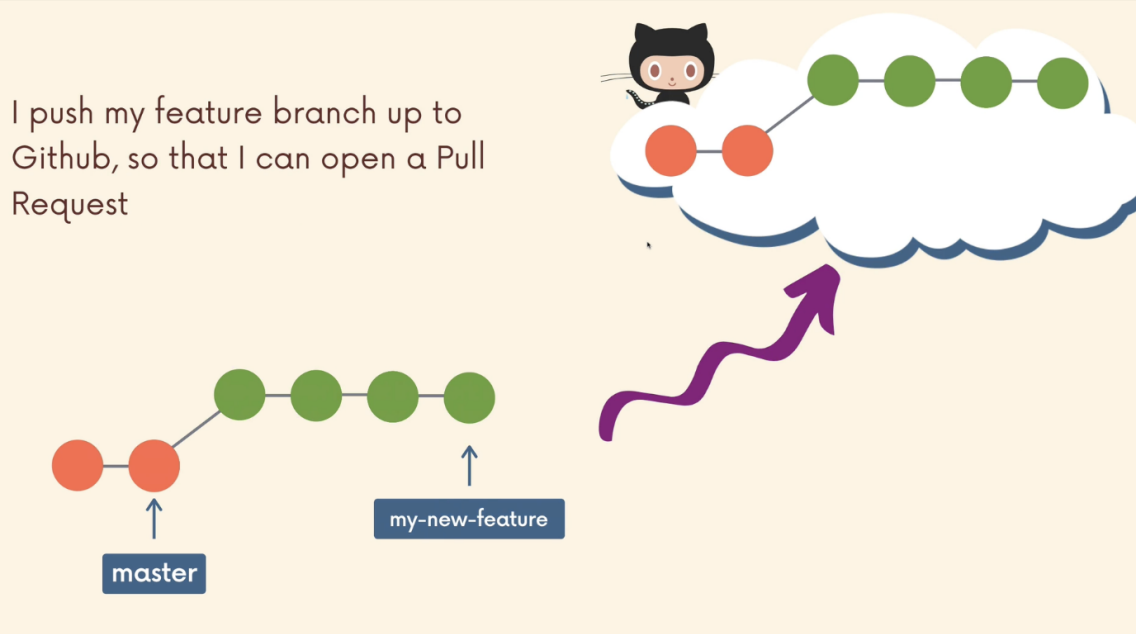
They allow developers to alert team-members to new work that needs to be reviewed. They provide a mechanism to approve or reject the work on a given branch. They also help facilitate discussion and feedback on the specified commits.

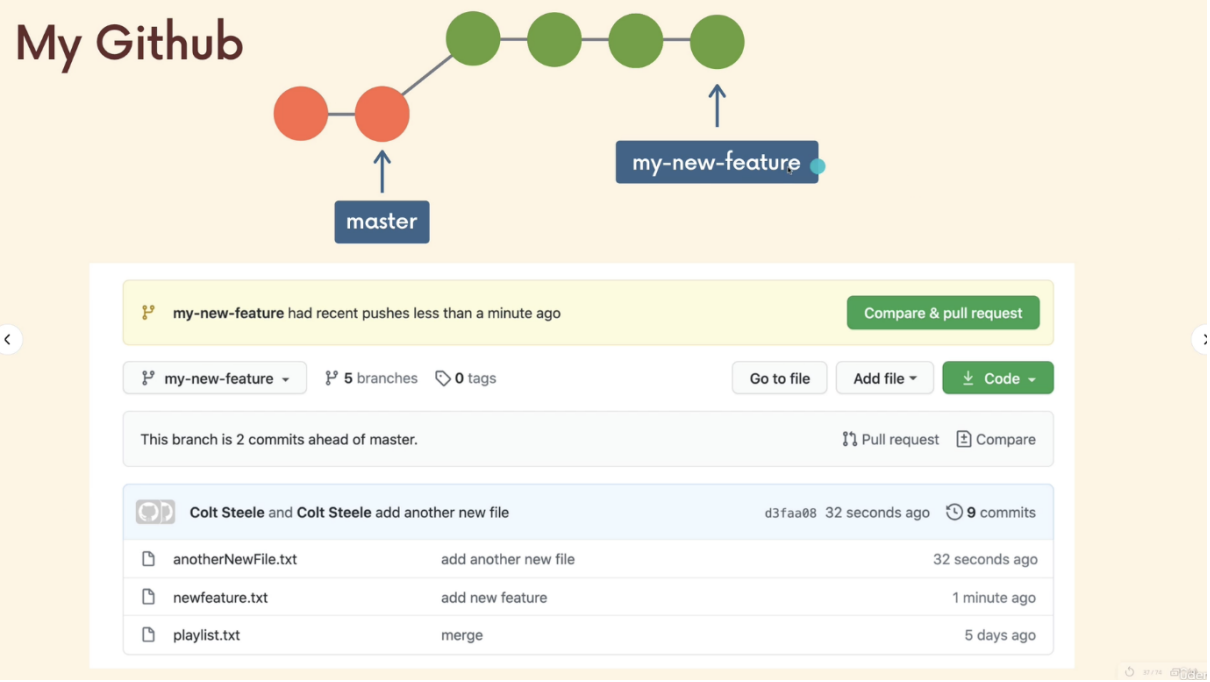
The Workflow

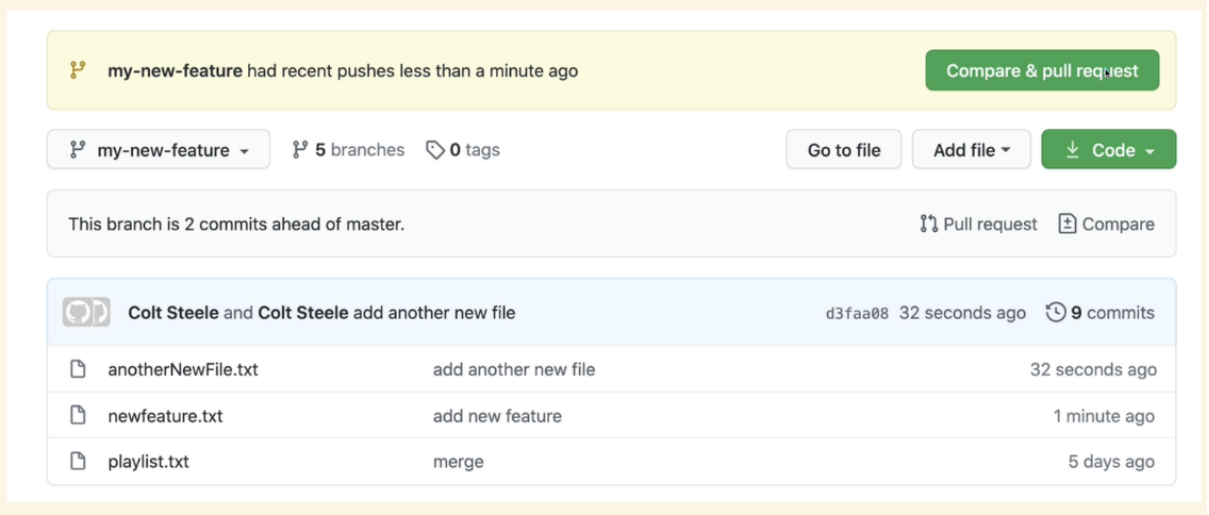
1. Do some work locally on feature branch
2. Push up the feature branch to Github
3. Open a pull request using the feature branch just pushed up to Github.

4. Wait for the PR to be approving and merged. Start a discussion on the PR. This part depends on the team structure.

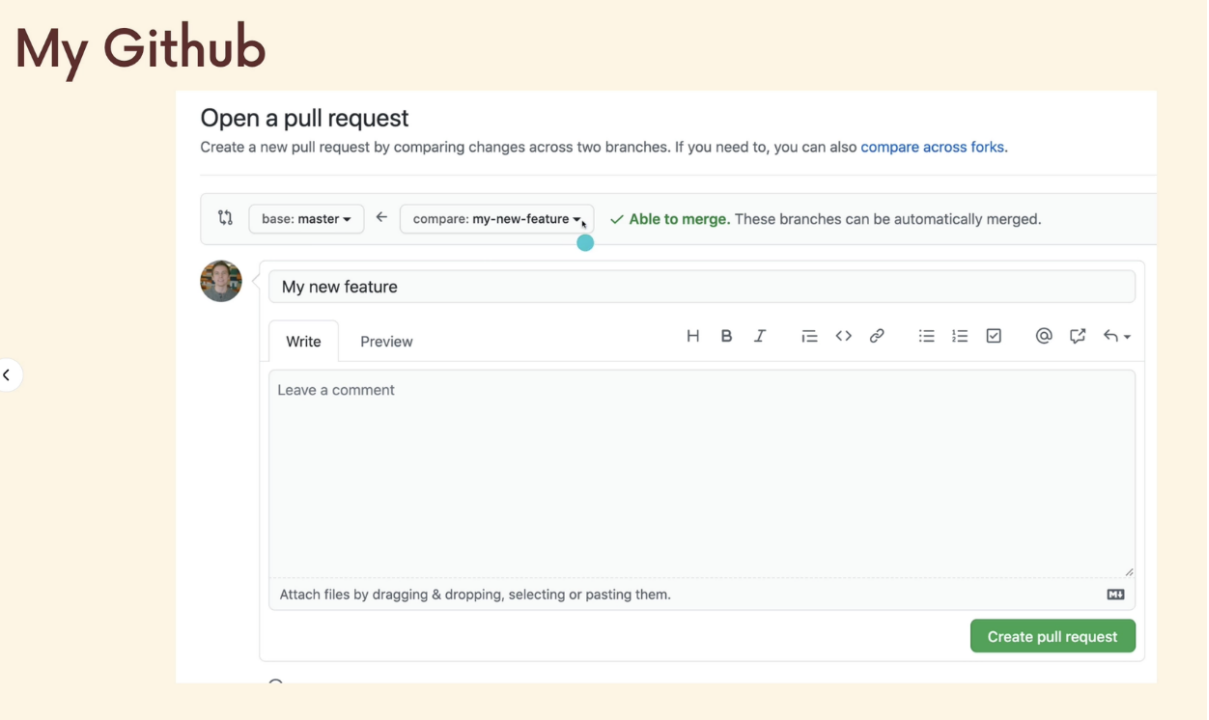
Process:



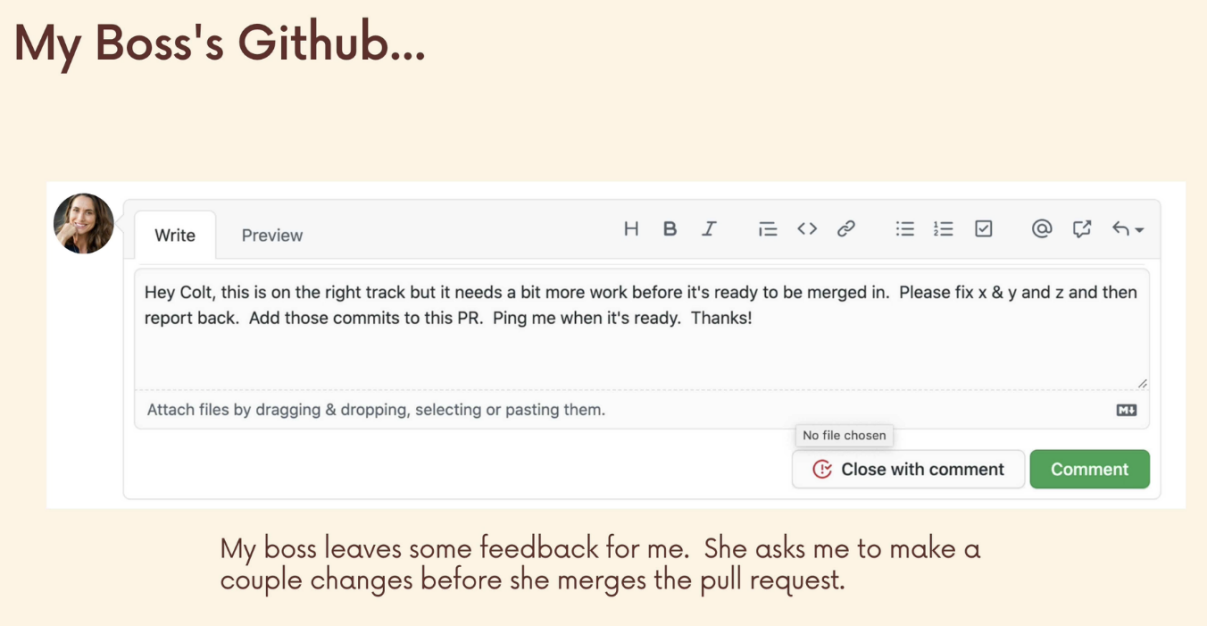


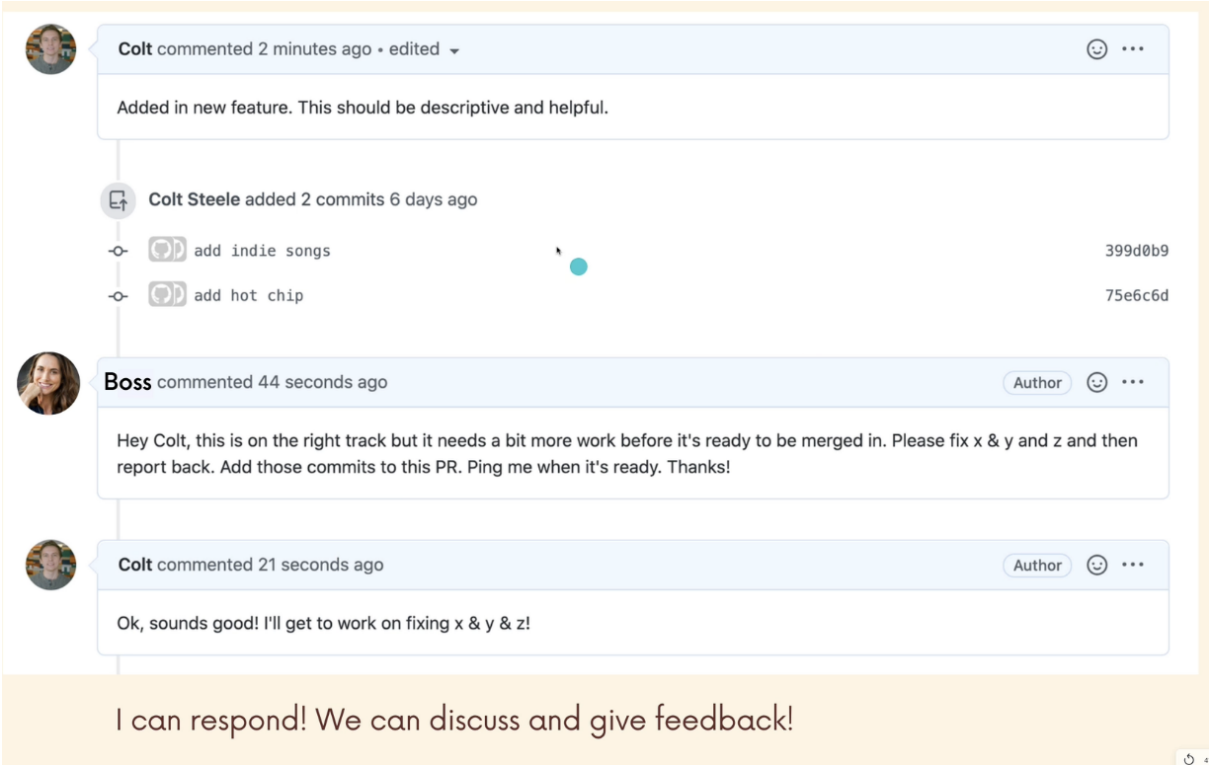

my-new-feature had recent pushes less than a minute ago
Compare & pull request
my-new-feature
5 branches
0 tags
Go to file
Add file
Code
This branch is 2 commits ahead of master.
Pull request
Compare
Colt Steele and Colt Steele add another new file
d3faa08 32 seconds ago
9 commits
anotherNewFile.txt
add another new file
32 seconds ago
newfeature.txt
add new feature
1 minute ago
playlist.txt
merge
5 days ago


My Github
master
my-new-feature
my-new-feature had recent pushes less than a minute ago
Compare & pull request
my-new-feature
5 branches
0 tags
Go to file
Add file
Code
This branch is 2 commits ahead of master.
Pull request
Compare
Colt Steele and Colt Steele add another new file
I click the PR button
anotherNewFile.txt
add another new file
32 seconds ago
newfeature.txt
add new feature
1 minute ago
playlist.txt
merge
5 days ago



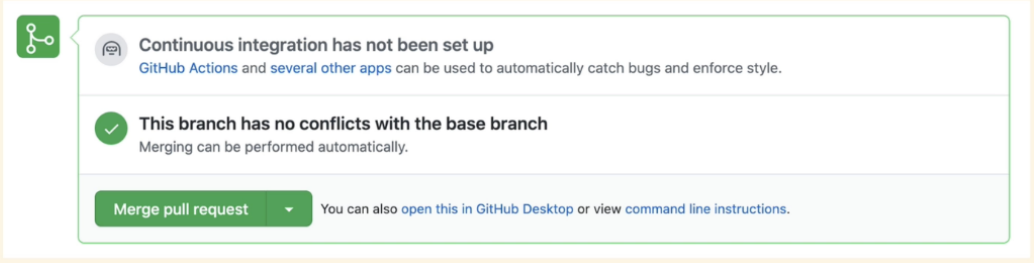
Select the base branch to merge with eg. main and compare branch to be merge.





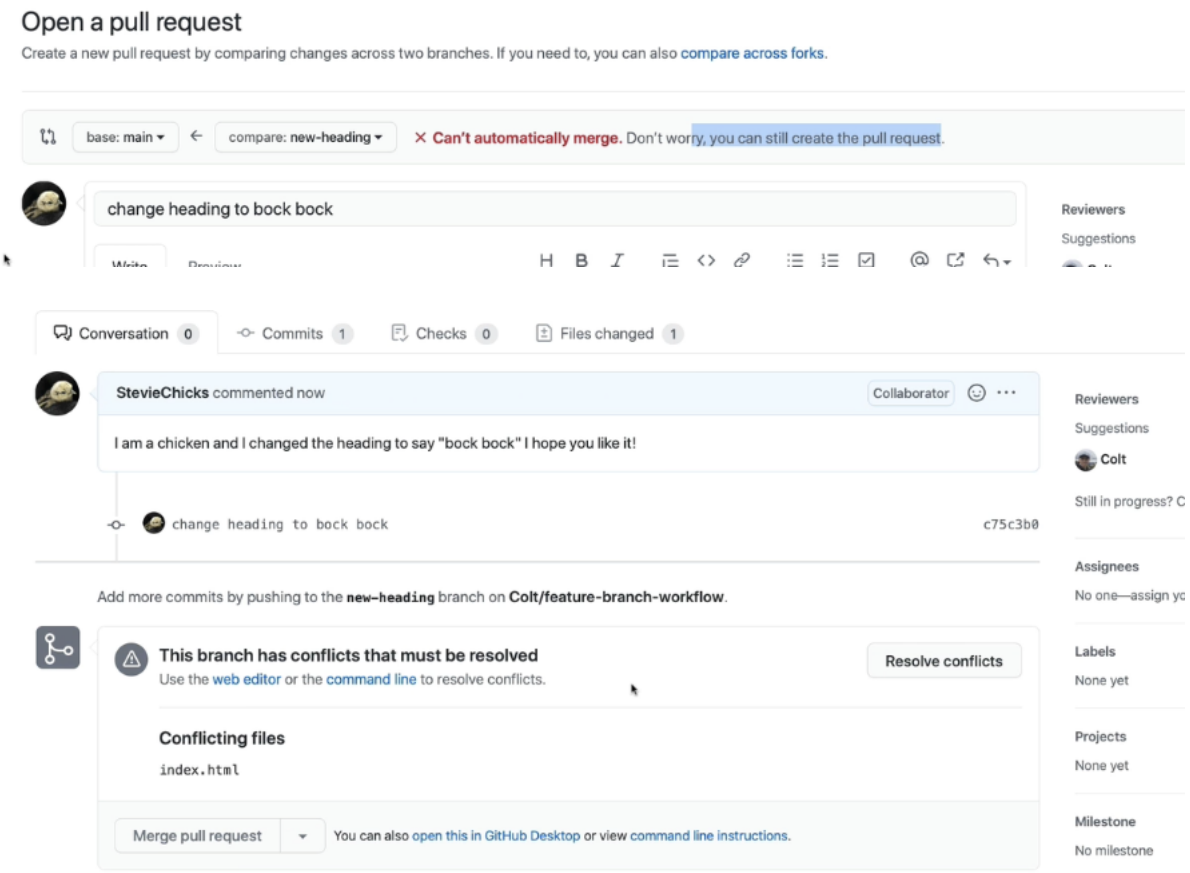
I can respond! We can discuss and give feedback!

My Boss's Github...



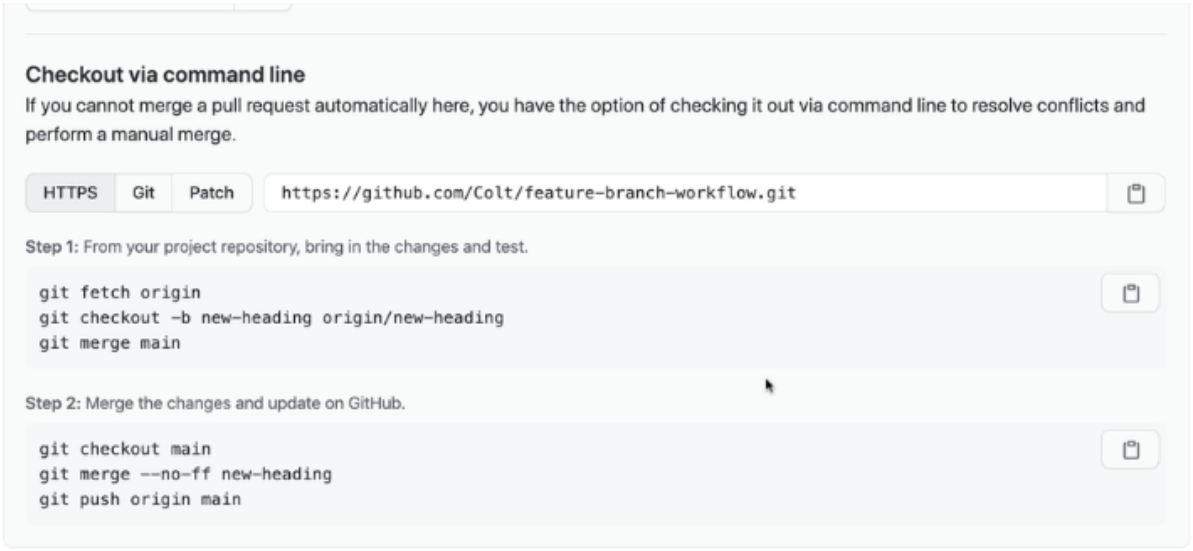
Once I make the requested changes, my boss (or whoever is in charge of merging) can merge in my pull request!

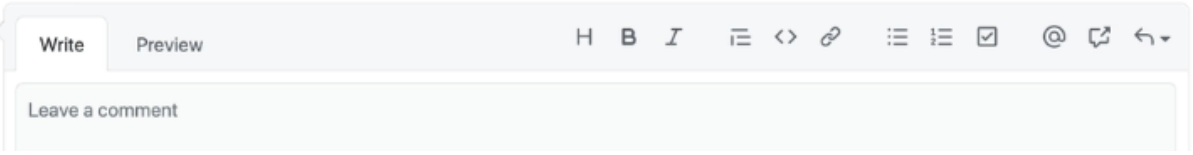
Merging Pull Requests With Conflicts



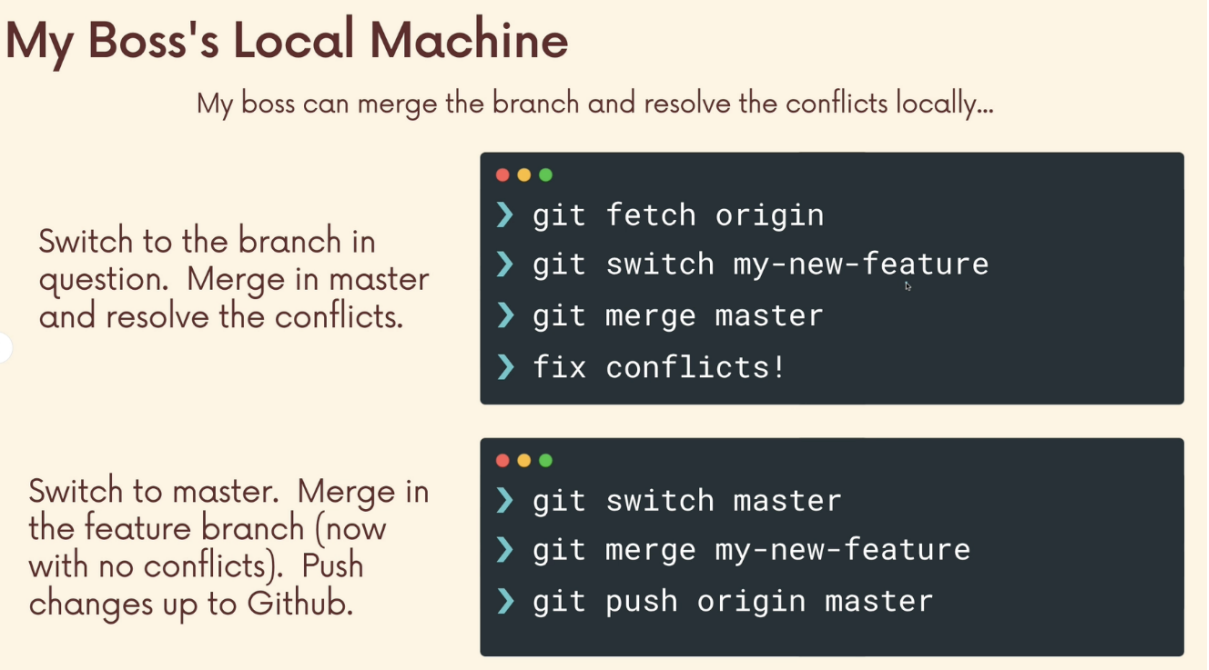
We can edit the conflict by clicking on the edit button on the Github page or we can also do it manually using git. Click on the **command line instructions** option, it will provide steps to

resolve conflict.



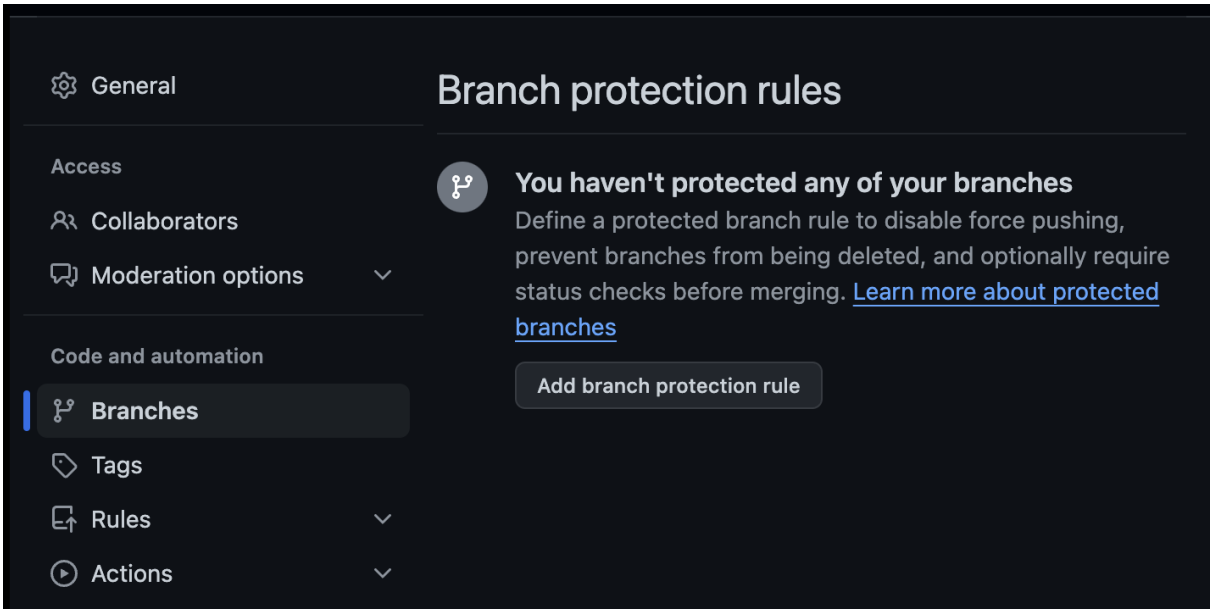



```
git fetch origin
git switch new-heading
git merge main

git checkout main
git pull origin main
git merge main
# resolve the conflict

git add .
git commit -m "Resolve Conflict"
git switch main
git merge new-heading # or --no-ff new-heading
git branch -D new-heading
git push origin main
```

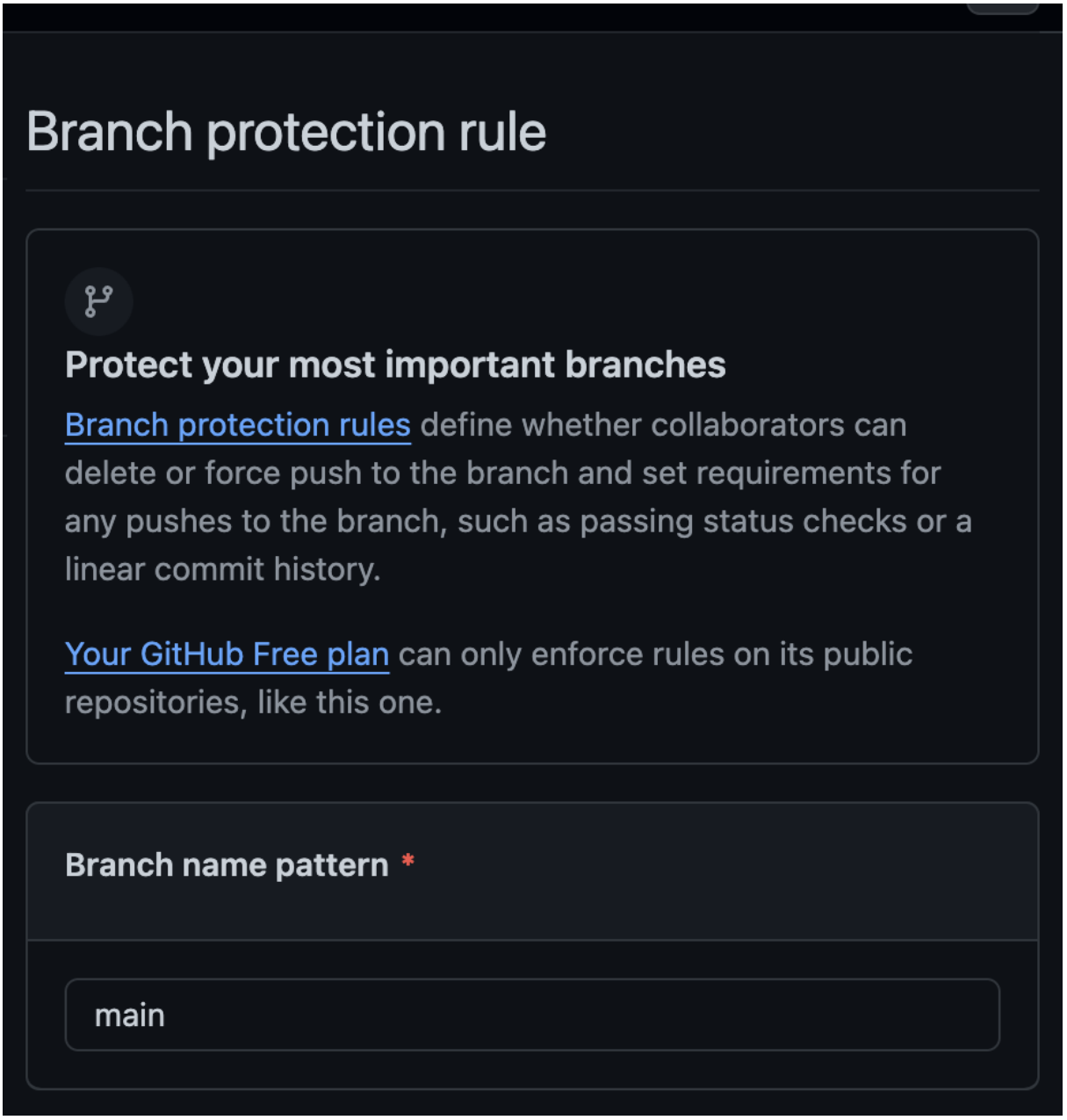


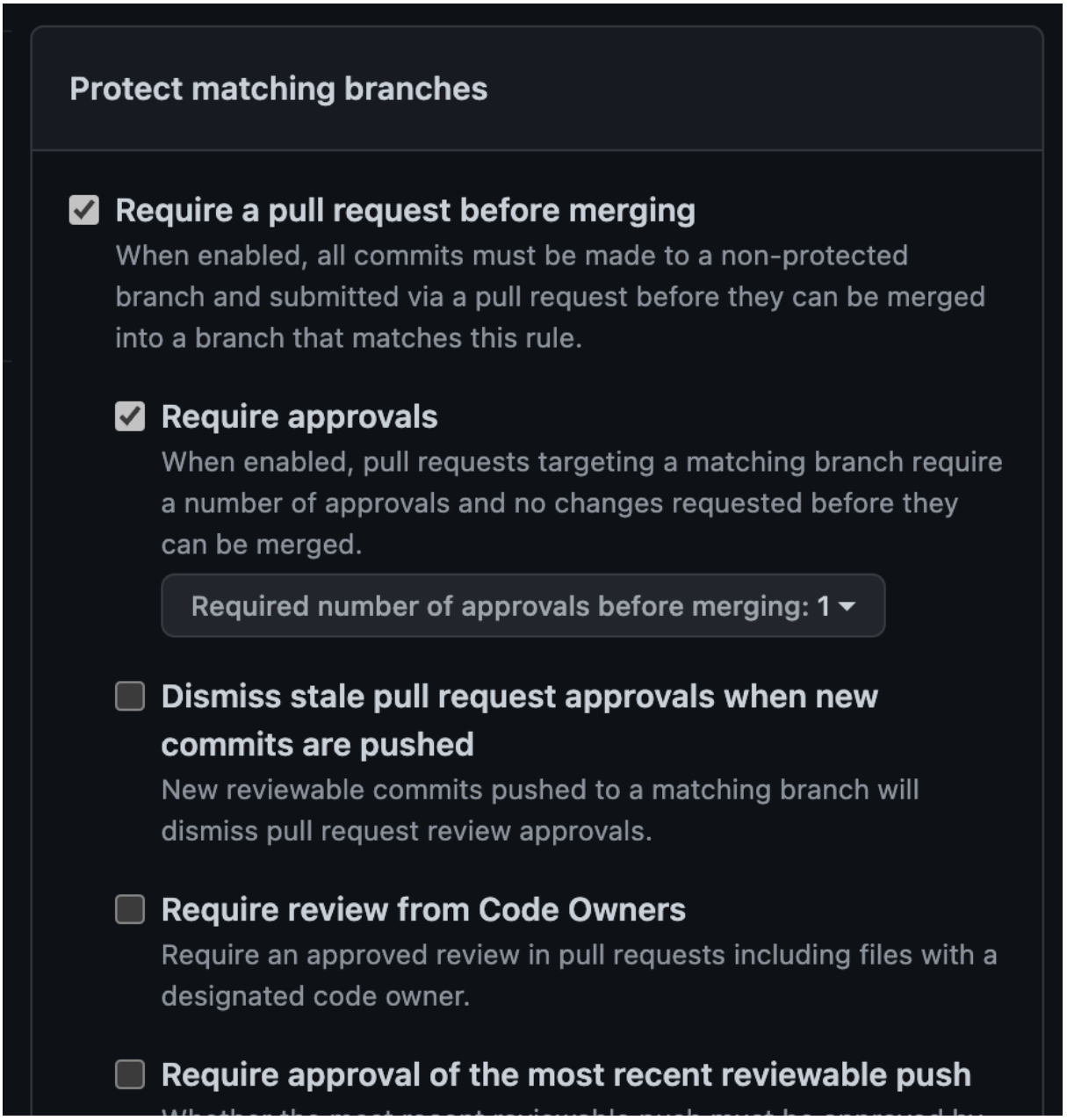
Configuring Branch Protection Rules

In Repository Setting's select Branches option.

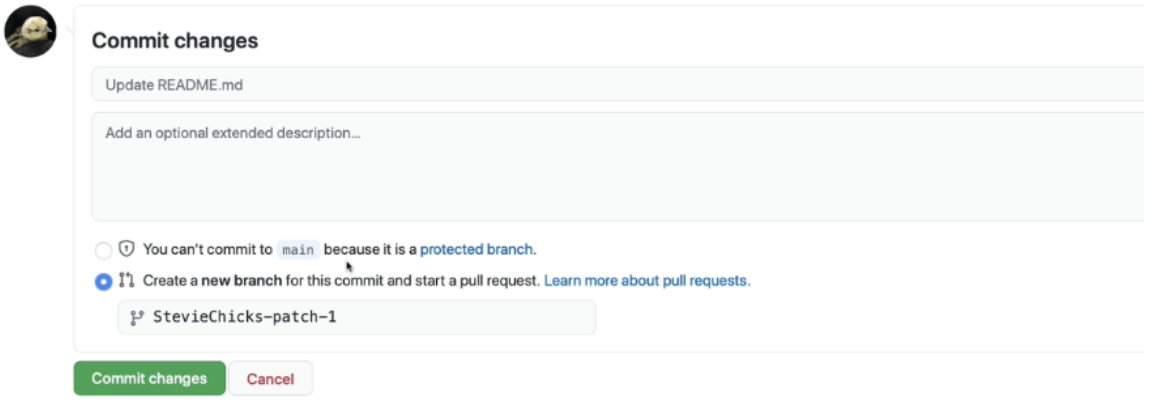


Click on Add branch protection rule and enter "main" as Branch name pattern option.

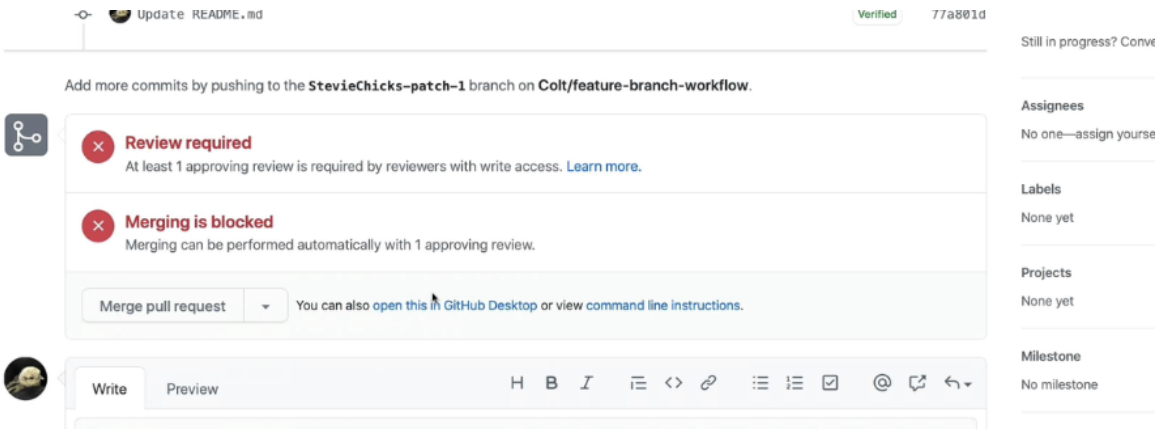

Branch protection rule
Protect your most important branches
Branch protection rules define whether collaborators can delete or force push to the branch and set requirements for any pushes to the branch, such as passing status checks or a linear commit history.
Your GitHub Free plan can only enforce rules on its public repositories, like this one.
Branch name pattern *
main



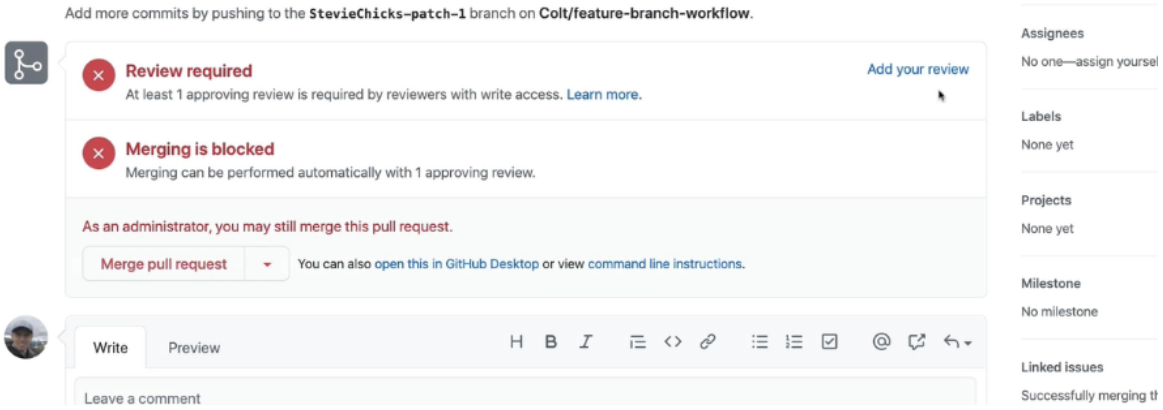
If someone try's to commit a change in main branch it will display a protection error.



The user should perform a Pull Request to the repository owner for the approval.



repository owner:



Fork & Clone: Another Workflow

The “fork & clone” workflow is different from anything we’ve seen so far. Instead of just one centralised Github repository, every developer has their own Github repository in addition to the “main” repo. Developers make changes and push to their own forks before making pull requests.

It’s very commonly used on large open-source projects where there may be thousands of contributors with only a couple maintainers.

Forking

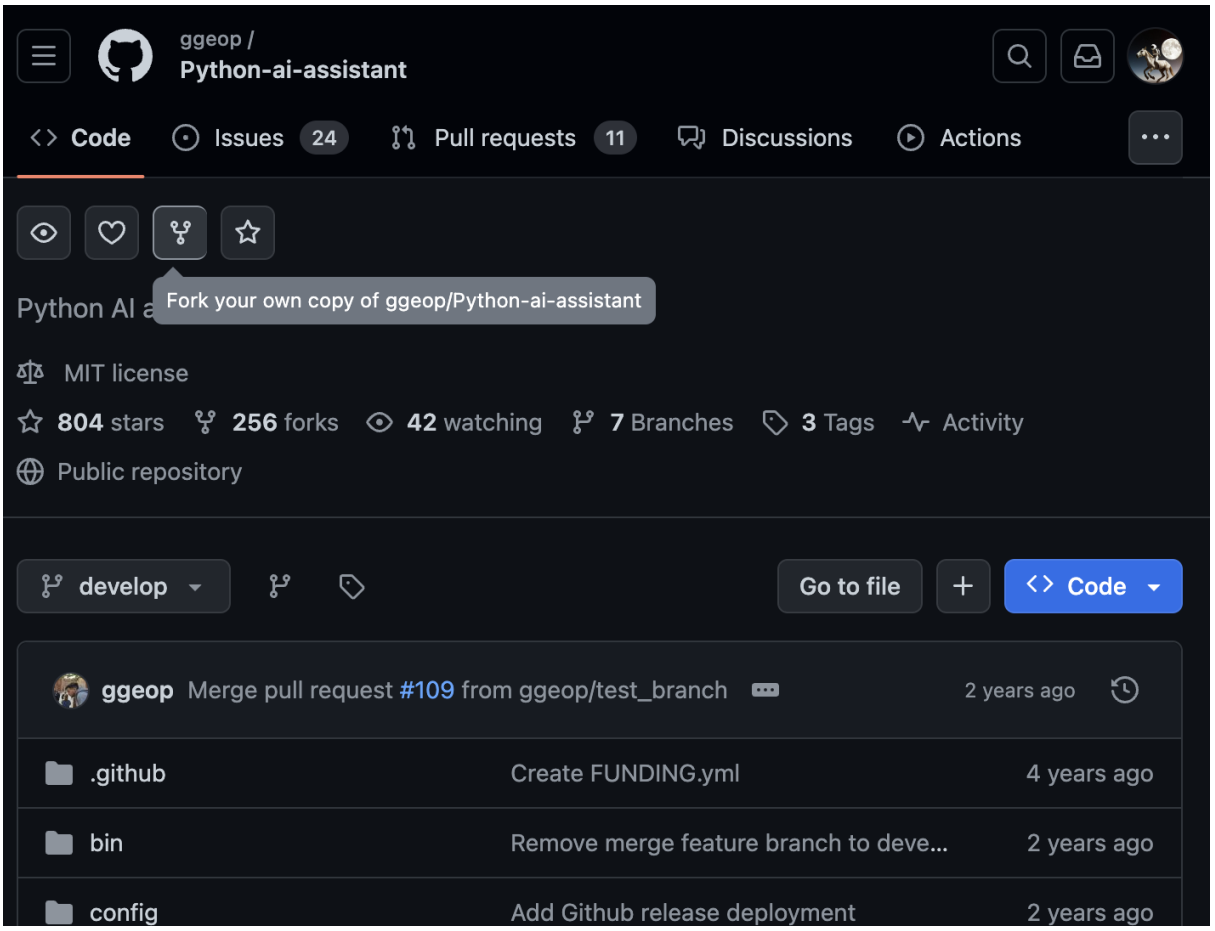
Github allow us to create personal copies of other peoples repositories. We call those copies a “fork” of the original.

When we fork a repo, we’re basically asking Github “Make me my own copy of this repo please”

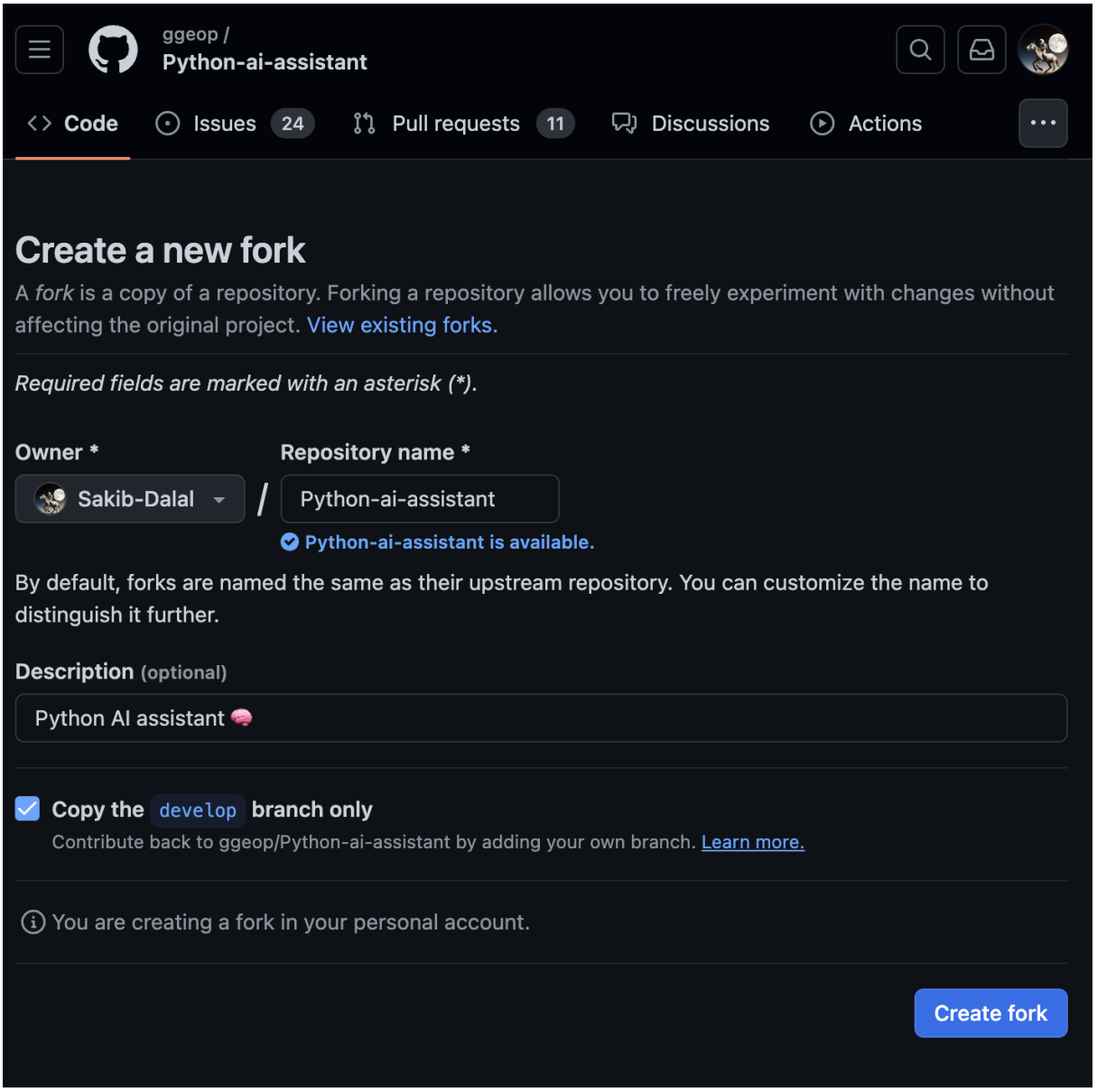
As with pull requests, forking is not a Git feature. The ability to fork is implemented by Github.

Note: It is not part of git, it’s part of Github.

Example:

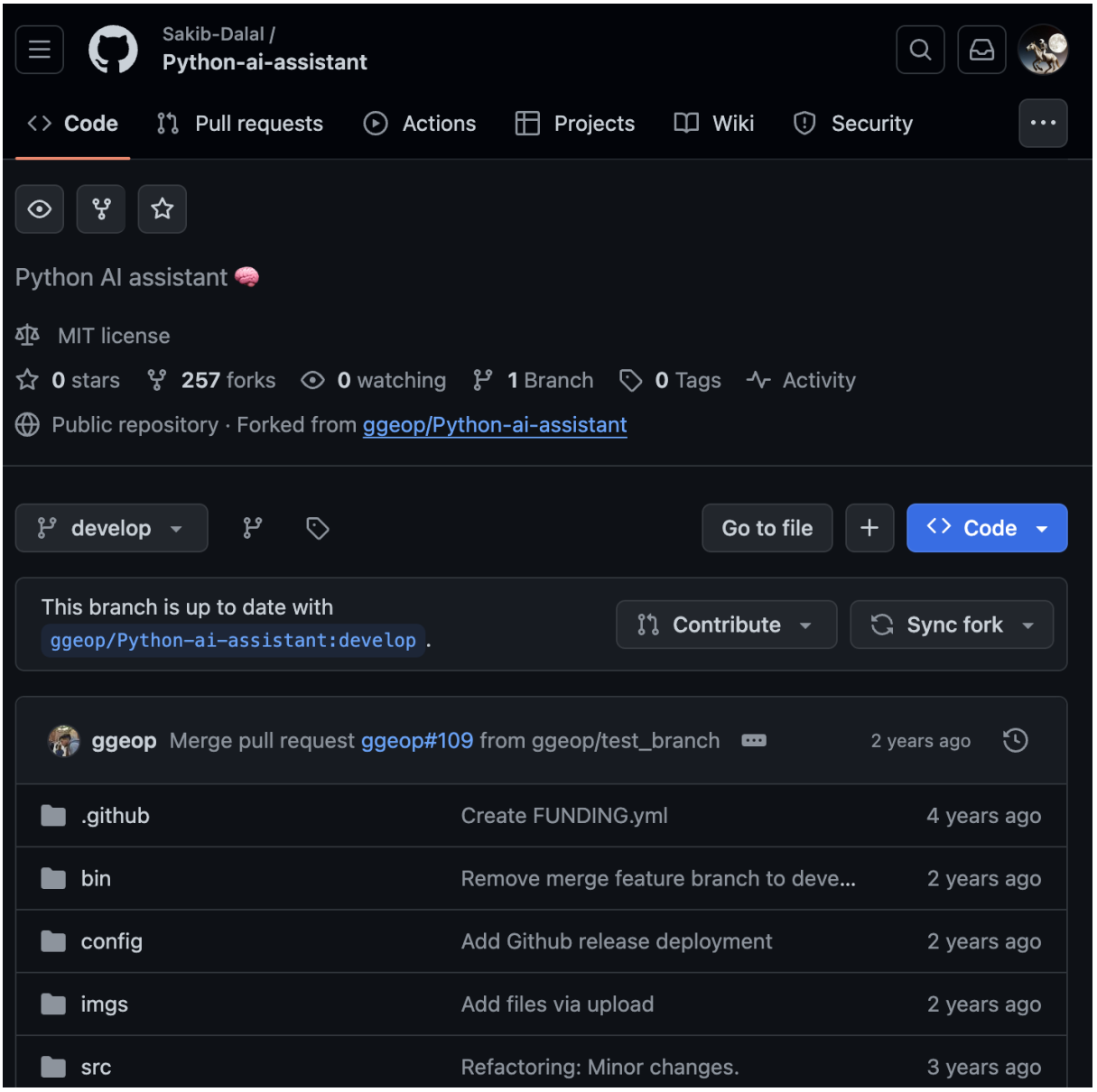


Click on the fork icon.



Click on Create Fork.

It will create a copy of a repo on your Github account.



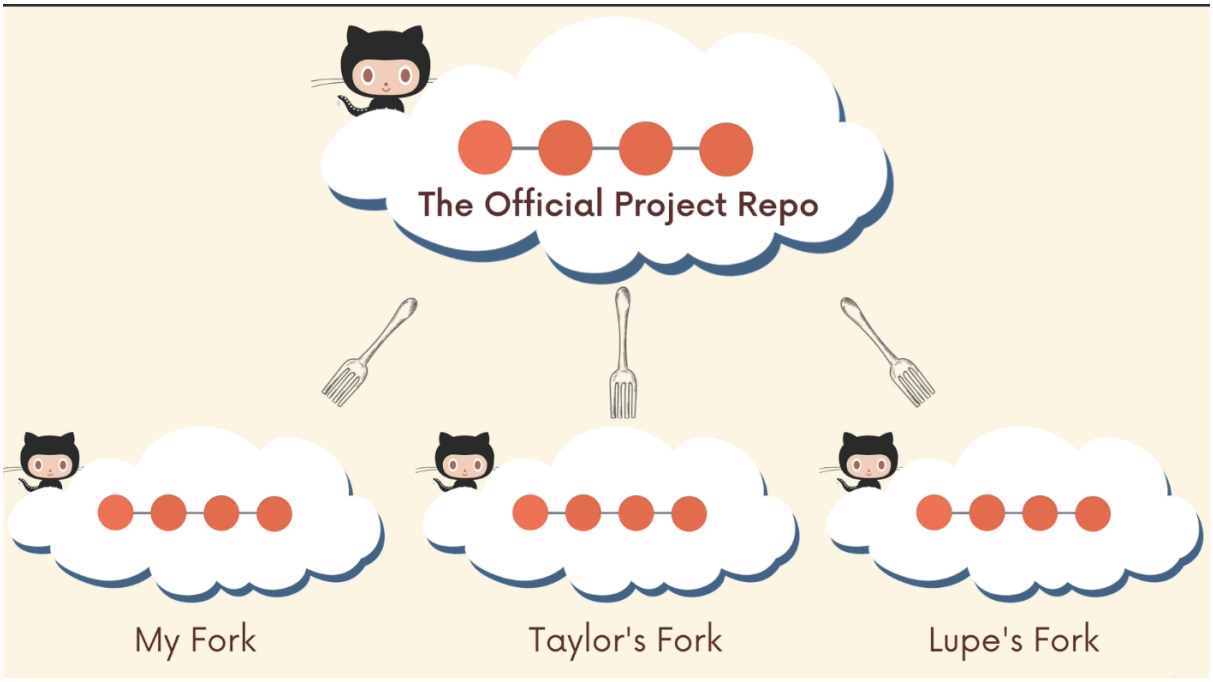
Now What?

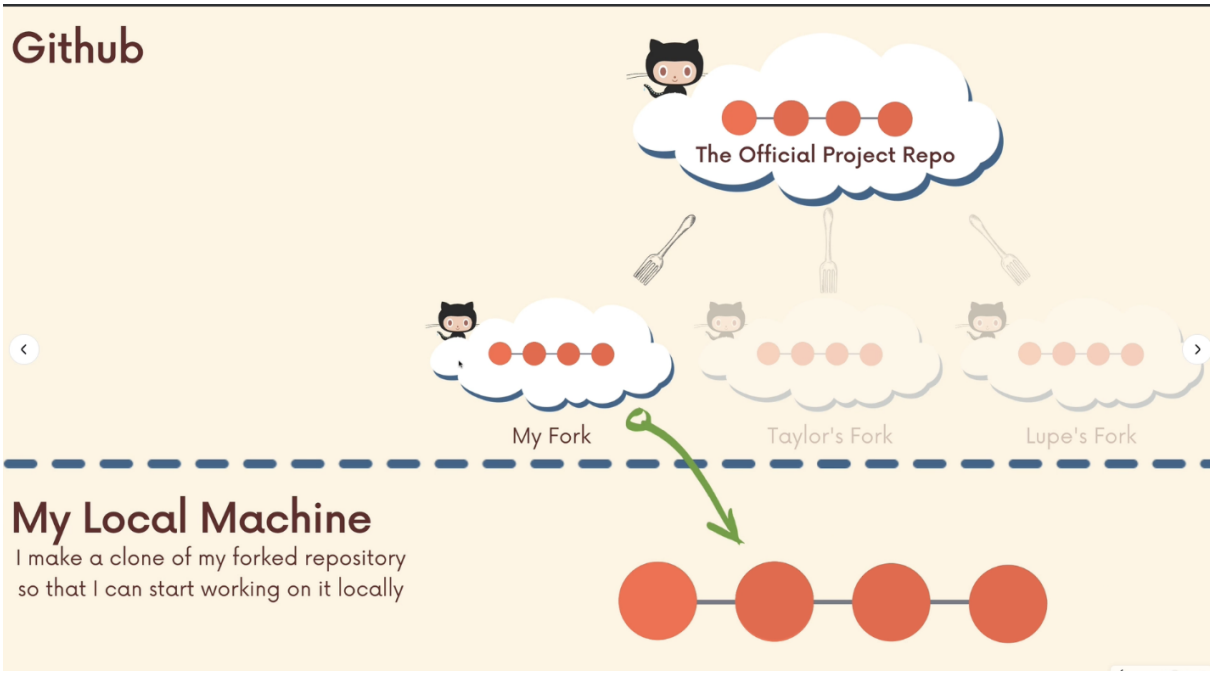
Now that I've forked, I have my own copy of the repo where I can do whatever I want!

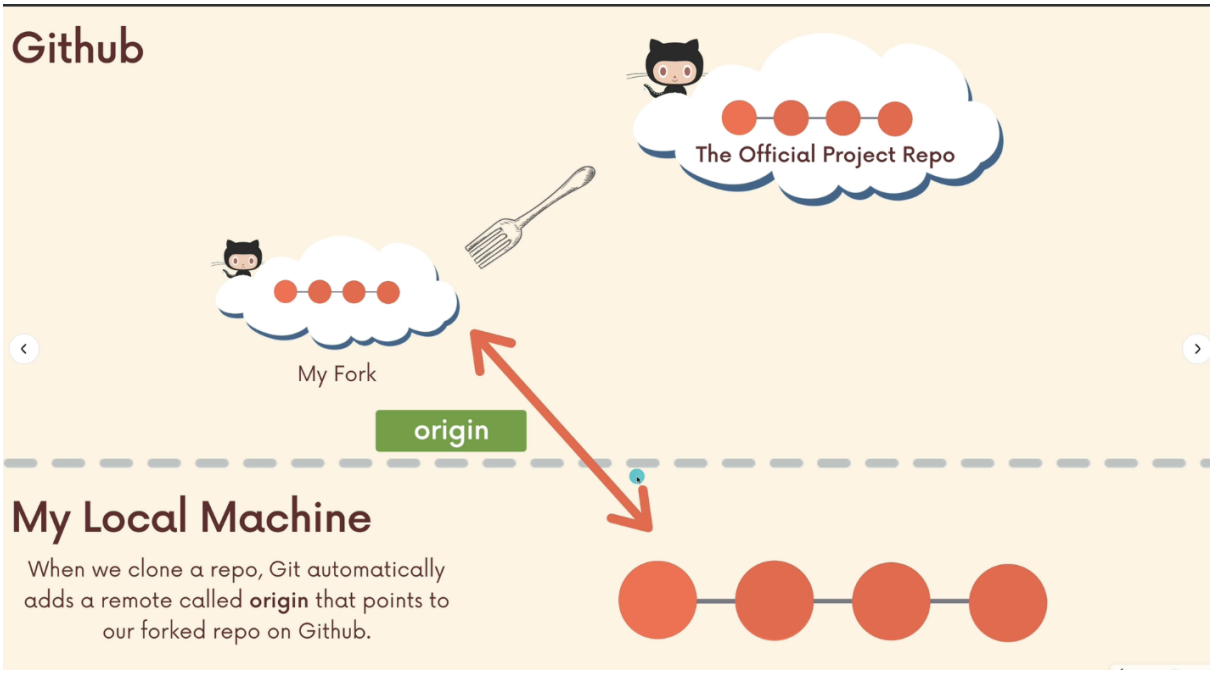
I can clone my fork and make changes, add features, and break things without fear of disturbing the original repository.

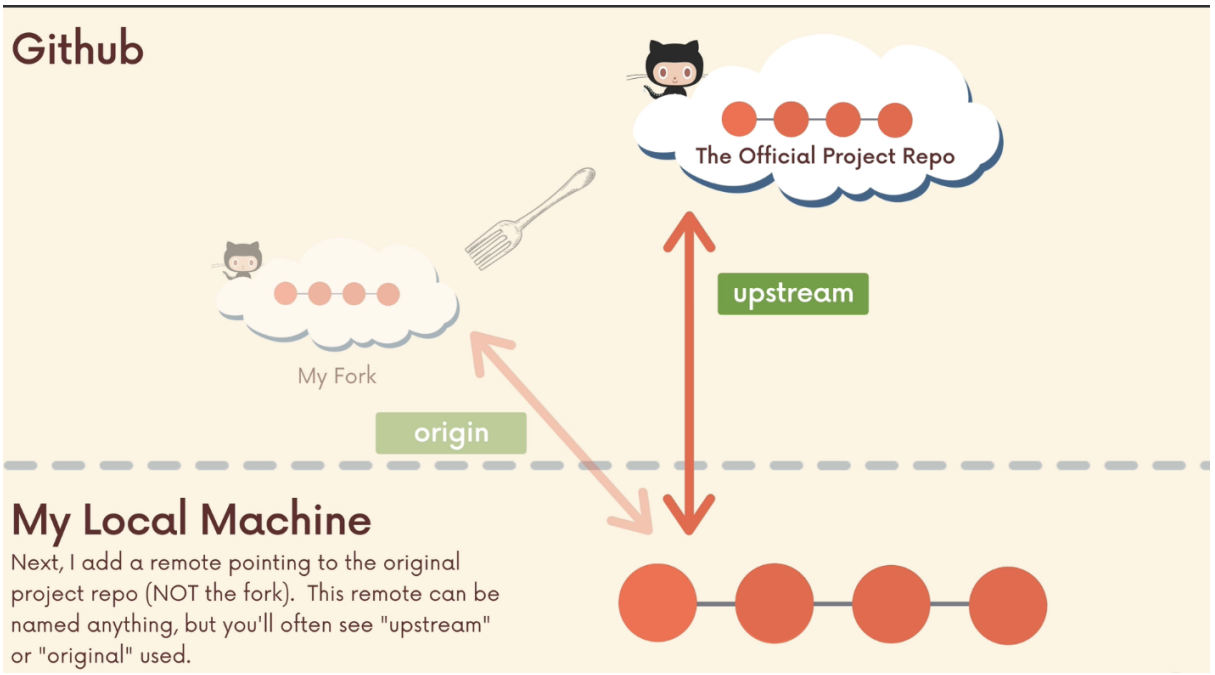
If I do want to share my work, I can make a pull request from my fork to the original repo.

Example:

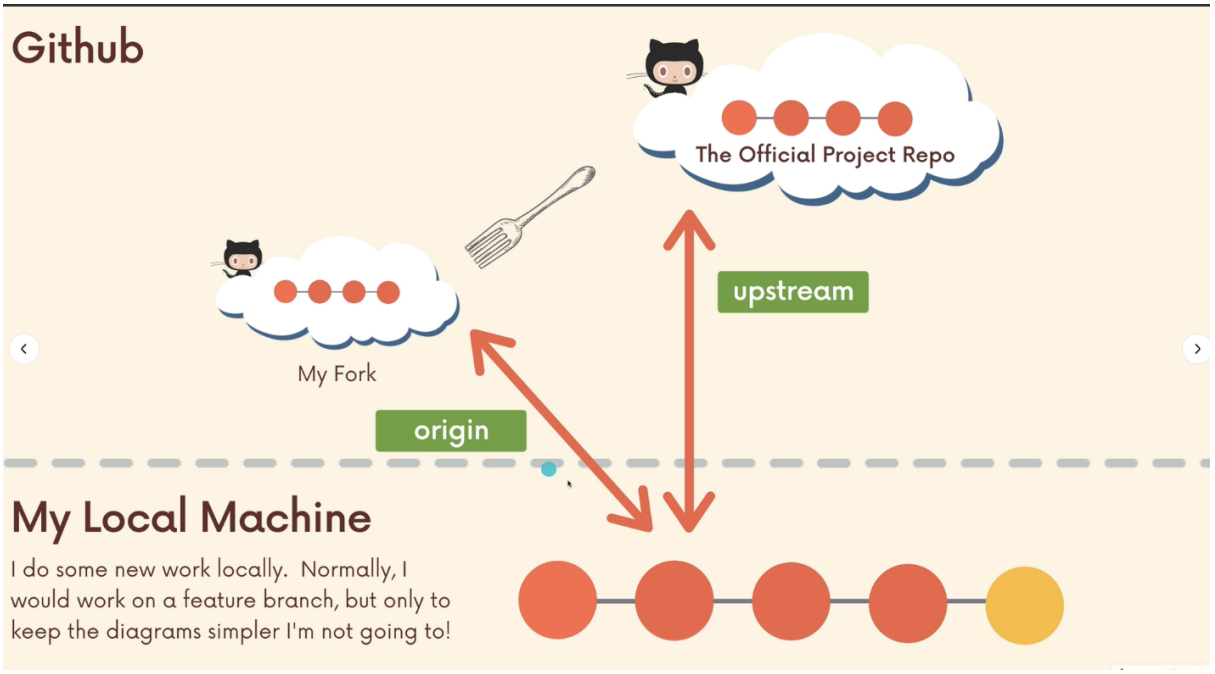


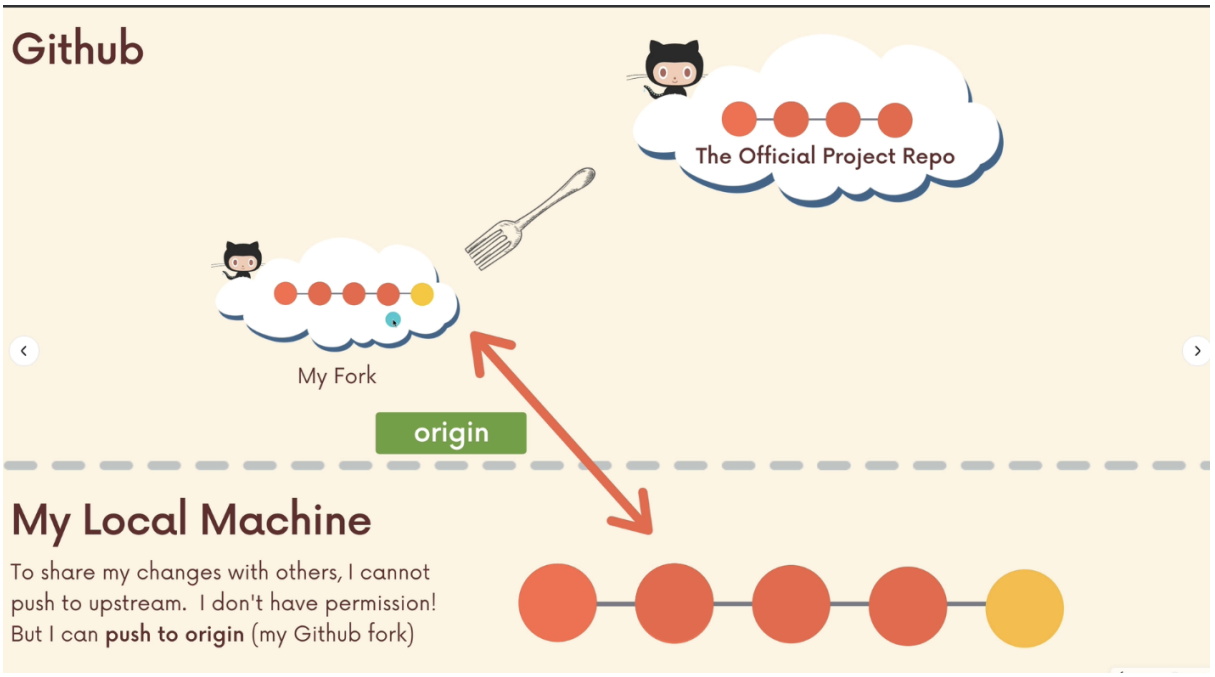


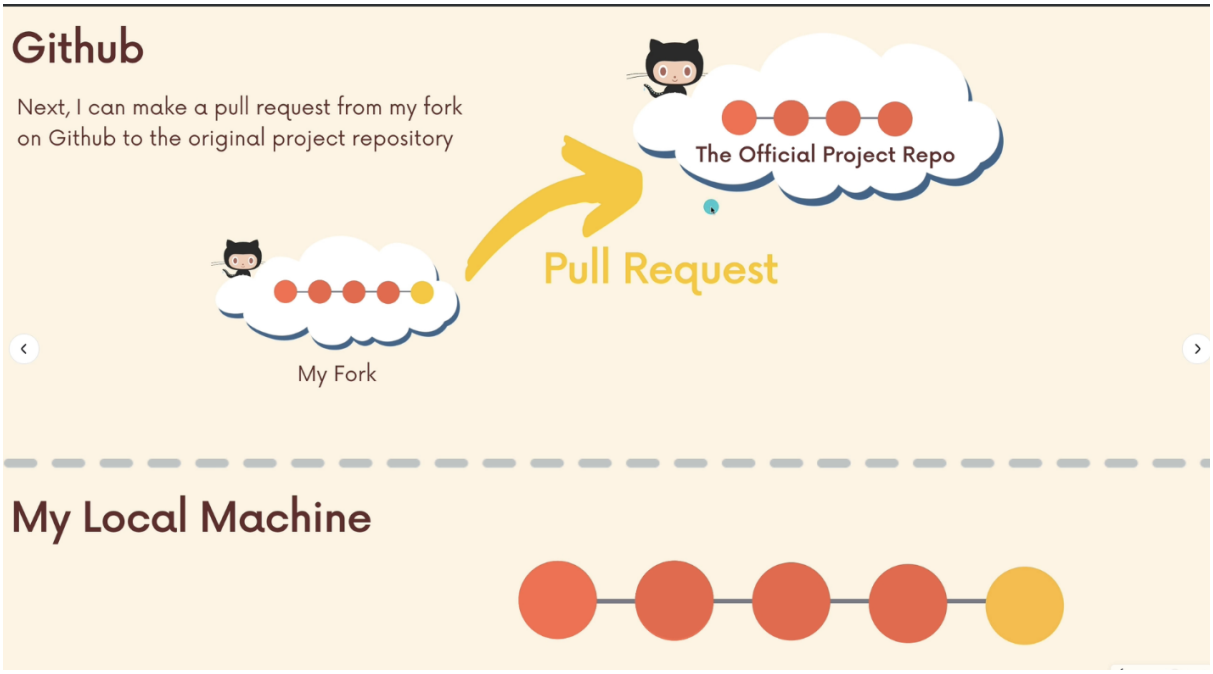


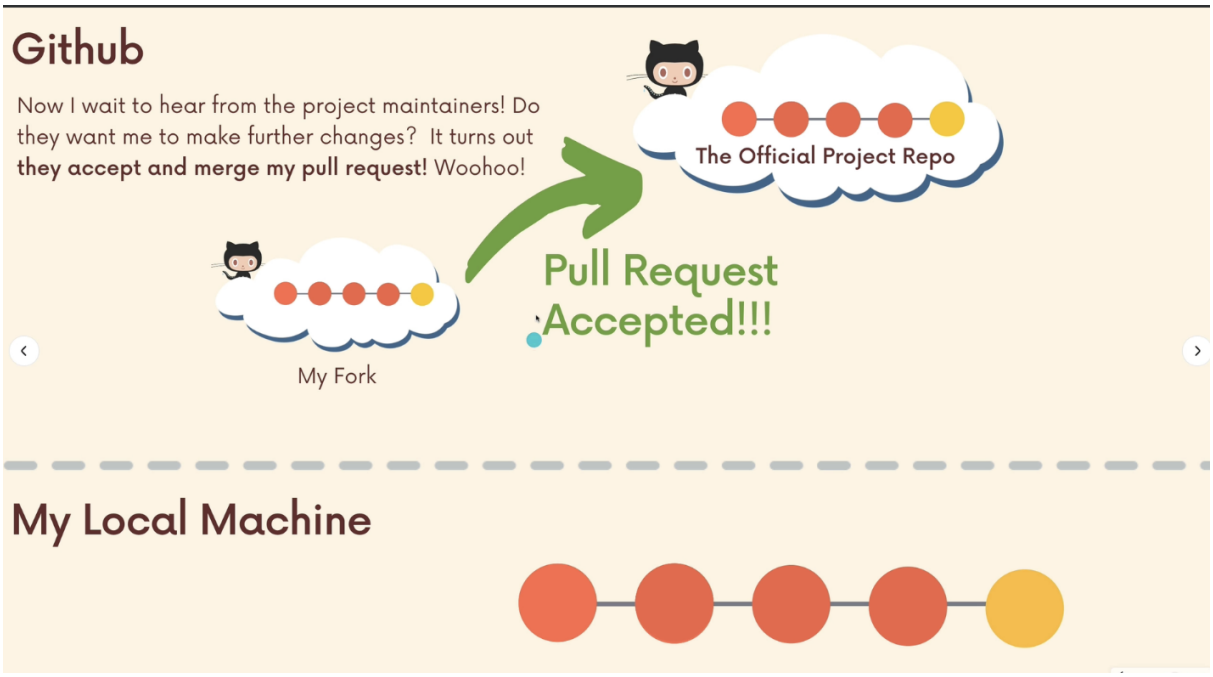


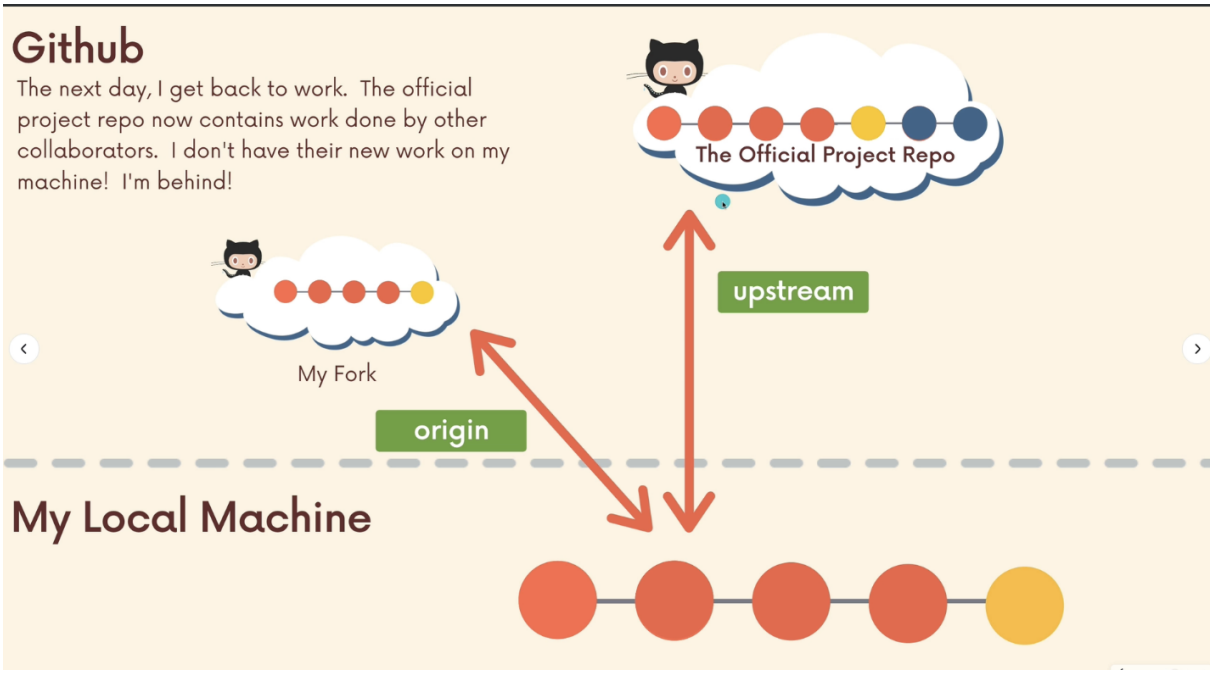
Now we have two remotes “origin” and “upstream”

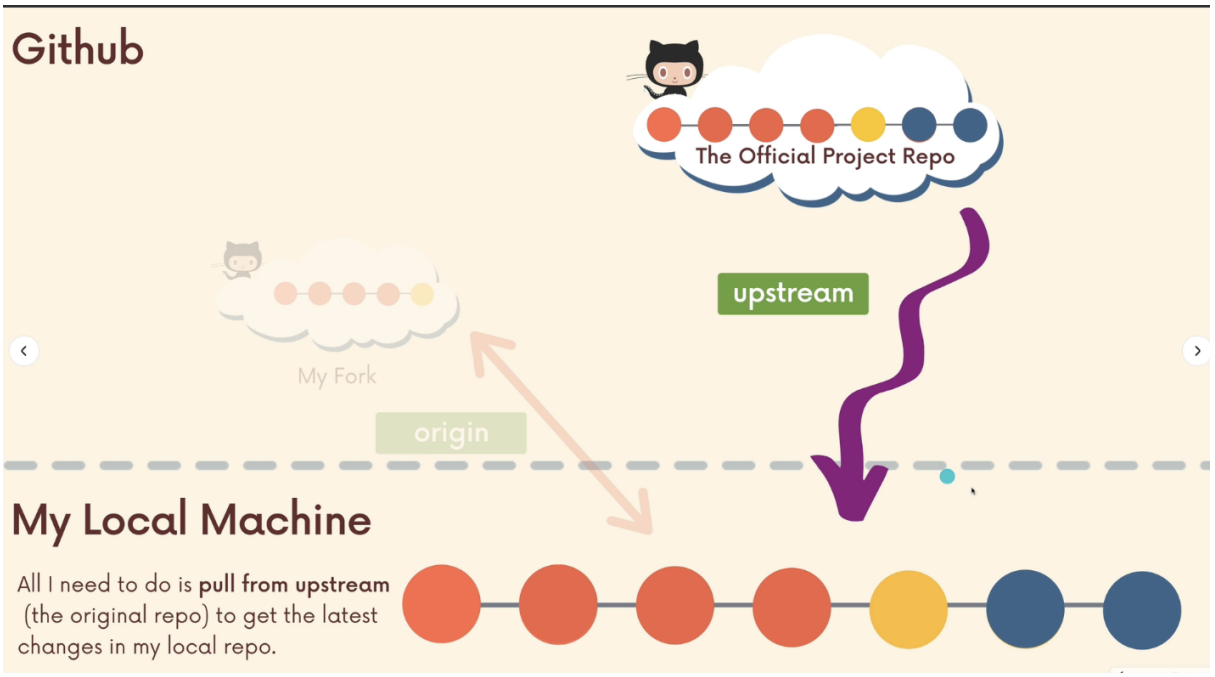

Github
The Official Project Repo
upstream
My Fork
origin
My Local Machine
I do some new work locally. Normally, I would work on a feature branch, but only to keep the diagrams simpler I'm not going to!

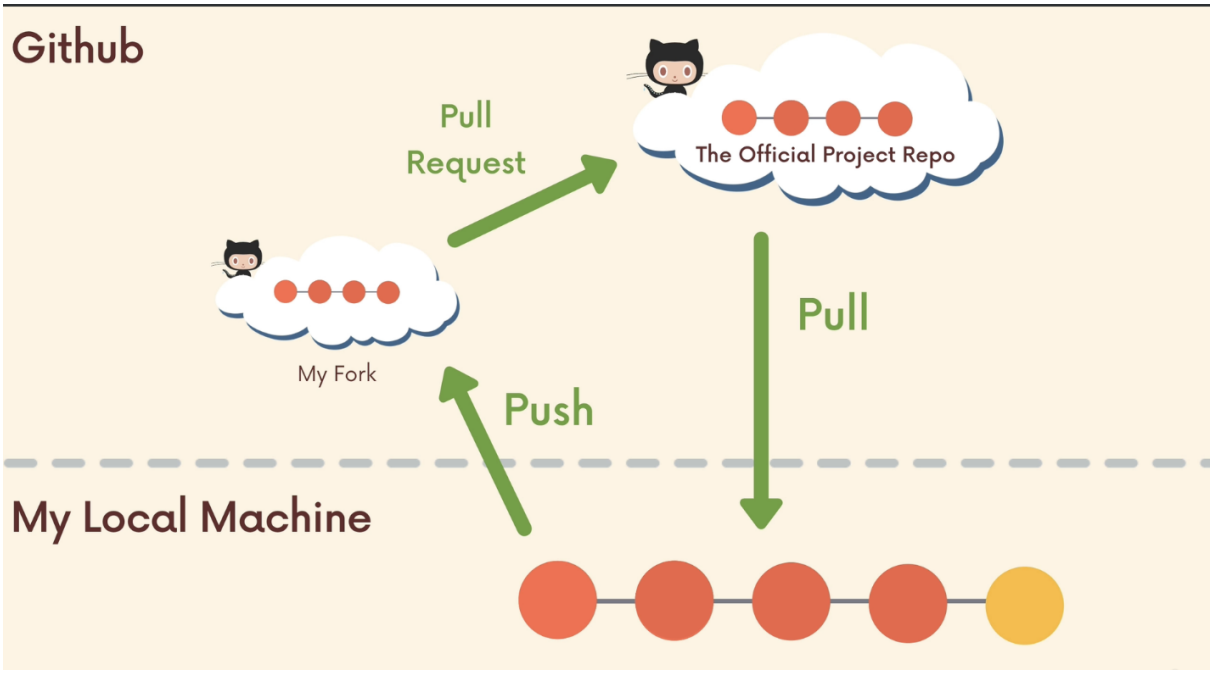

Github
The Official Project Repo
My Fork
origin
My Local Machine
To share my changes with others, I cannot push to upstream. I don't have permission! But I can **push to origin** (my Github fork)


Github
Next, I can make a pull request from my fork on Github to the original project repository
The Official Project Repo
Pull Request
My Fork
My Local Machine


Github
Now I wait to hear from the project maintainers! Do they want me to make further changes? It turns out they accept and merge my pull request! Woohoo!
The Official Project Repo
Pull Request Accepted!!!
My Fork
My Local Machine


Github
The next day, I get back to work. The official project repo now contains work done by other collaborators. I don't have their new work on my machine! I'm behind!
The Official Project Repo
upstream
My Fork
origin
My Local Machine


Github
The Official Project Repo
upstream
My Fork
origin
My Local Machine
All I need to do is pull from upstream (the original repo) to get the latest changes in my local repo.



Summary

1. Fork the Project
2. Clone the fork
3. Add upstream remote
4. Do some work
5. Push to origin
6. Open Pull Request

Fork & Clone

The “Fork & Clone” Workflow might seem complicated, but it’s extremely common for good reason!

It allows a project maintainer to accept contributions from developers all around the world without having to add them as actual owners of the main project repository or worry about giving them all permission to push to the repo.

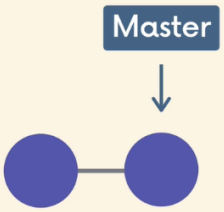
Rebasing

There are two main ways to use the git rebase command:

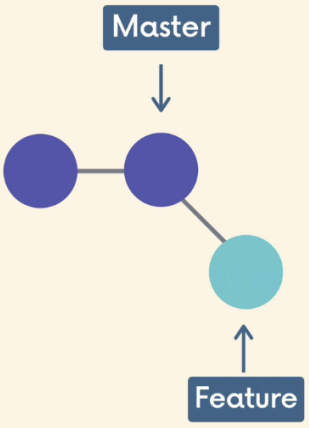
- as a alternative to merging
- as a cleanup tool

Example:

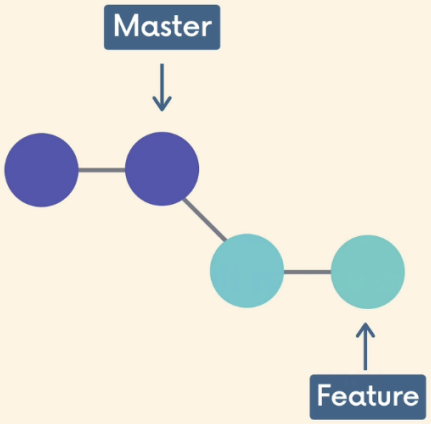
I'm working on a collaborative project



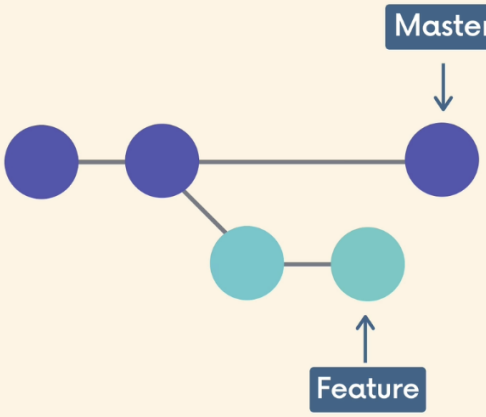
I do some work on the branch



I do some more work

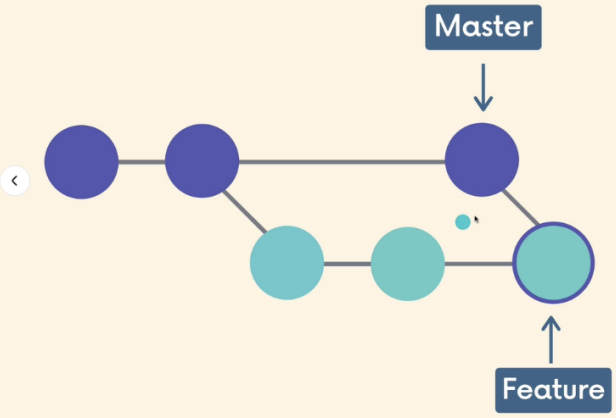


Master has new work on it!



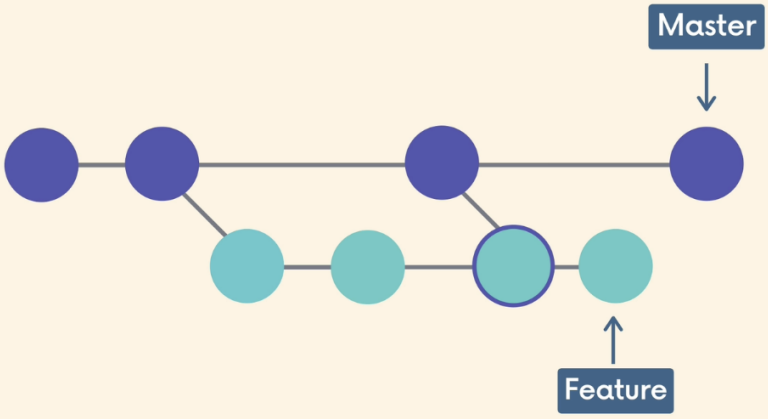
My feature branch doesn't have this work!

I merge master into feature

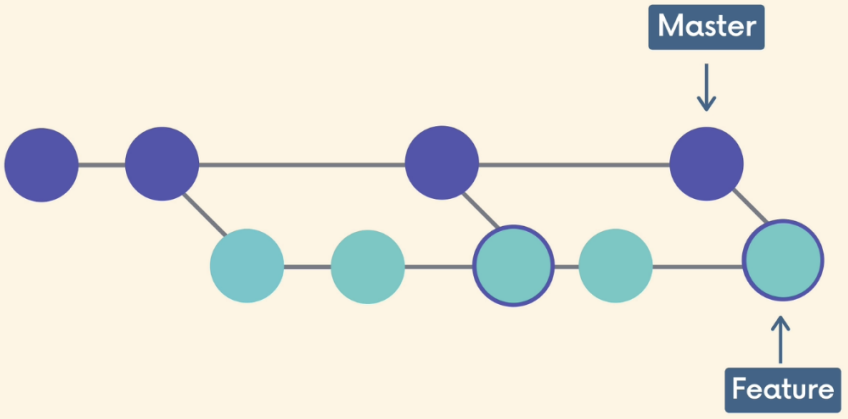


This results in a new merge commit

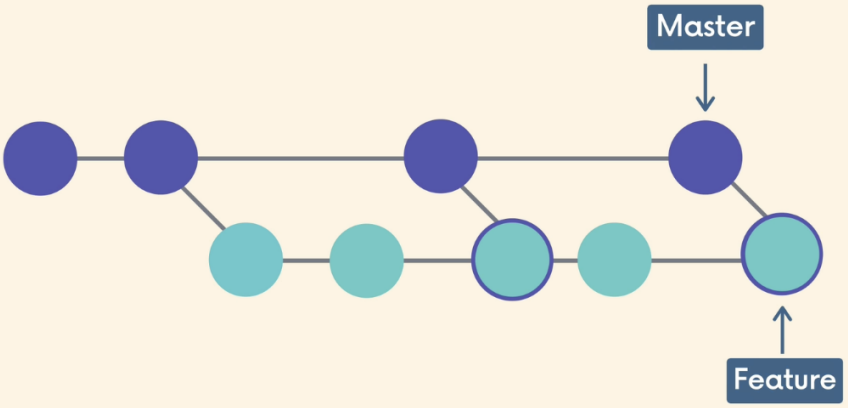
A coworker adds new work to master



I merge master in to my feature branch



This results in yet another merge commit!

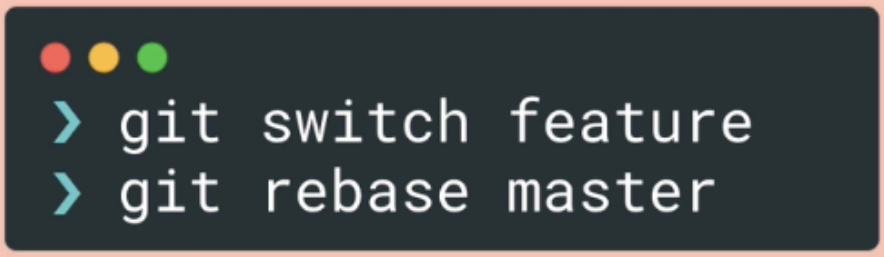


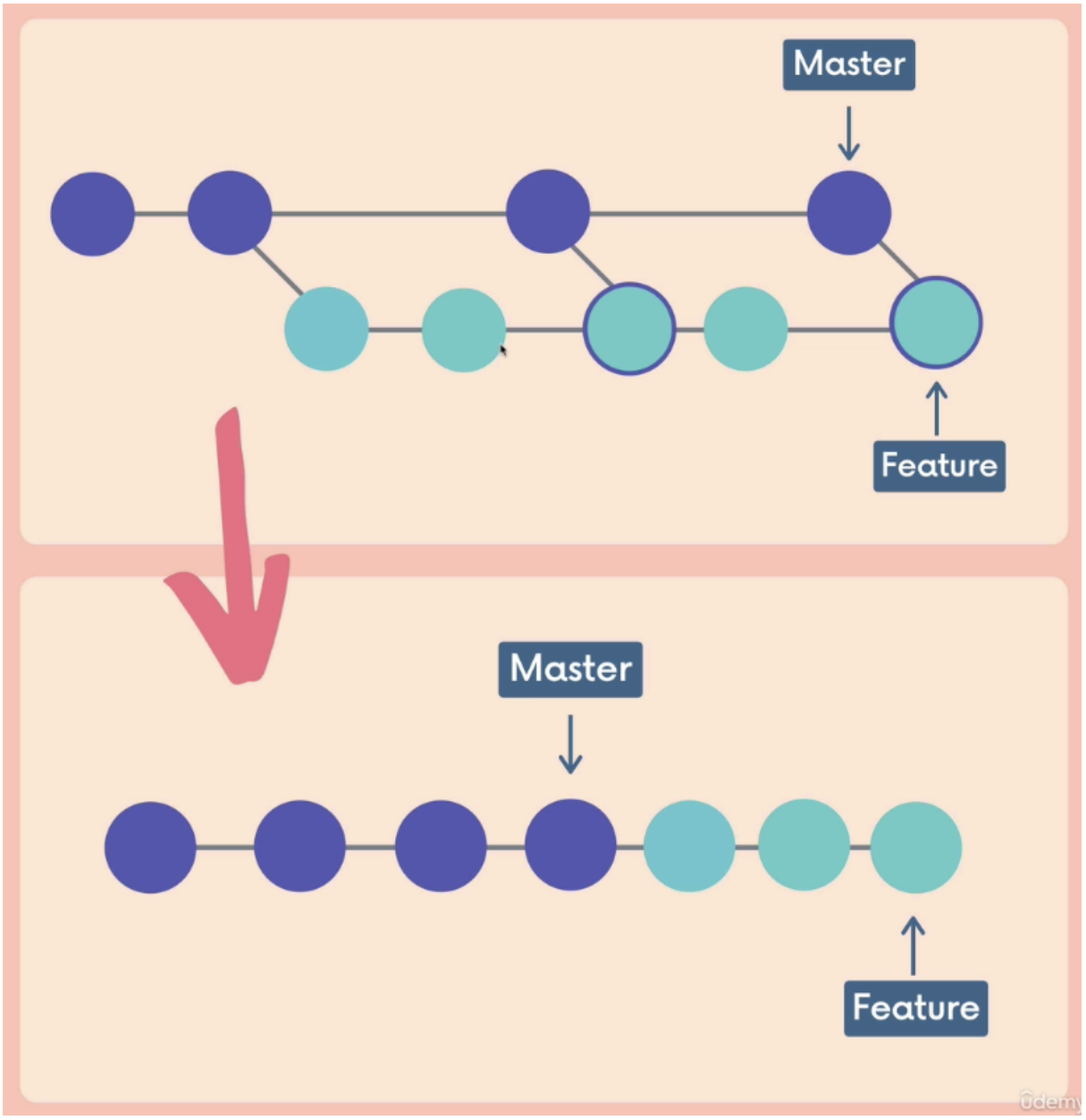
The feature branch has a bunch of merge commits. If the master branch is very active, my feature branch's history is muddied

Rebasing!

We can instead rebase the feature branch onto the master branch. This moves the entire feature branch so that it BEGINS at the tip of the master branch. All of the work is still there, but **we have re-written history.**

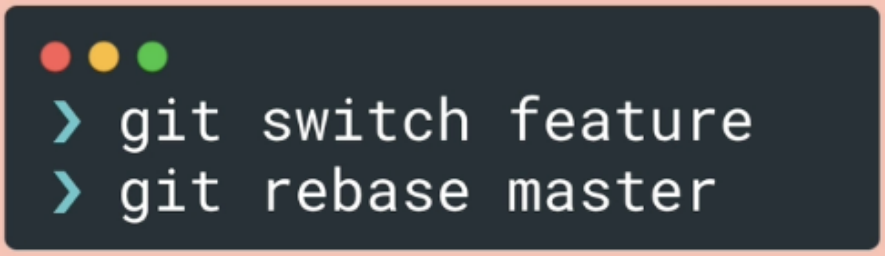
Instead of using a merge commit, rebasing rewrites history by **creating new commits** for each of the original feature branch commits.

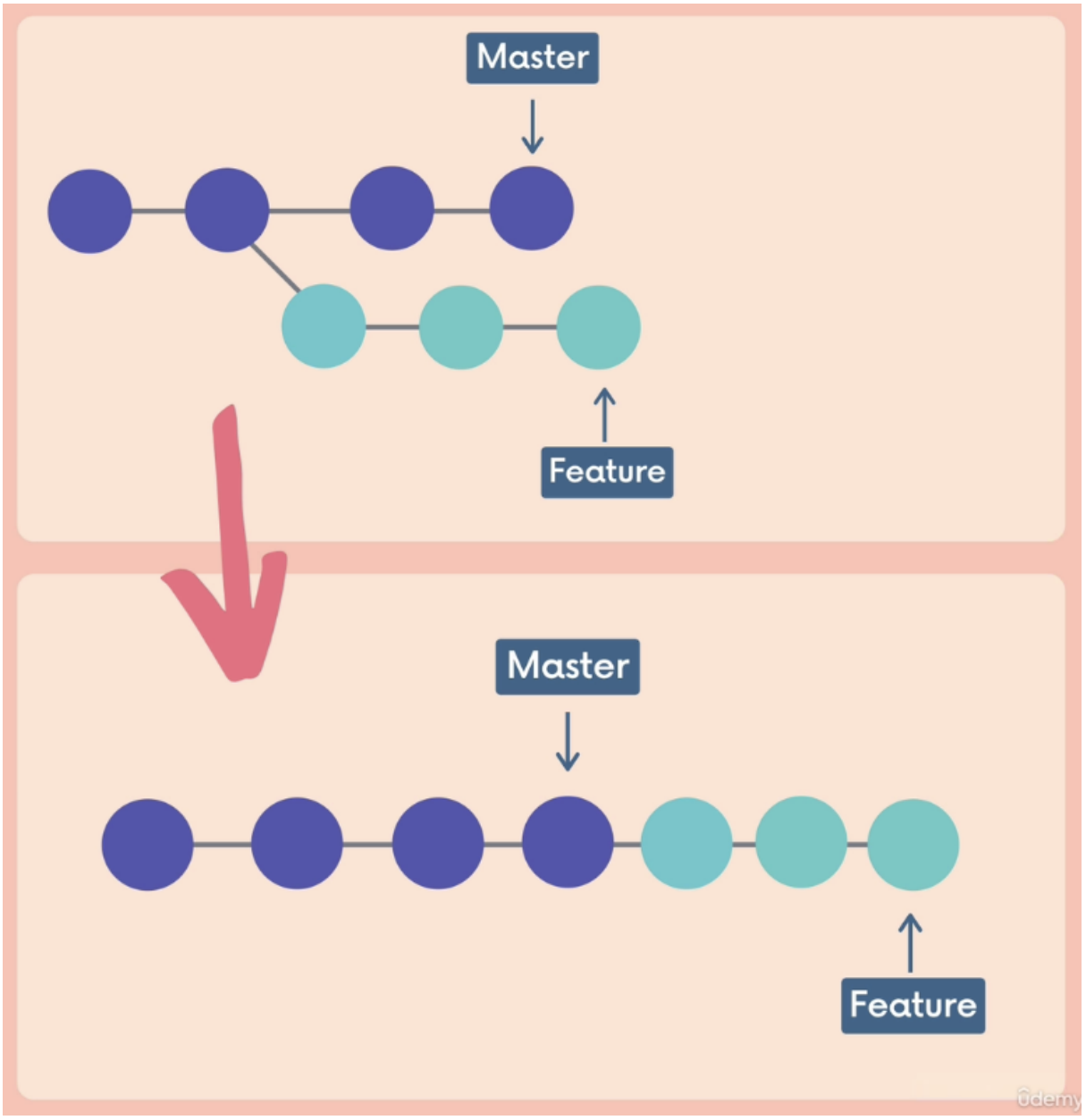



Master
Feature
Master
Feature

Rebasing!

We can also wait until we are done with a feature and then rebase the feature branch onto the master branch.





Why Rebase?

We get much cleaner project history. No unnecessary merge commits! We end up with a linear project history.

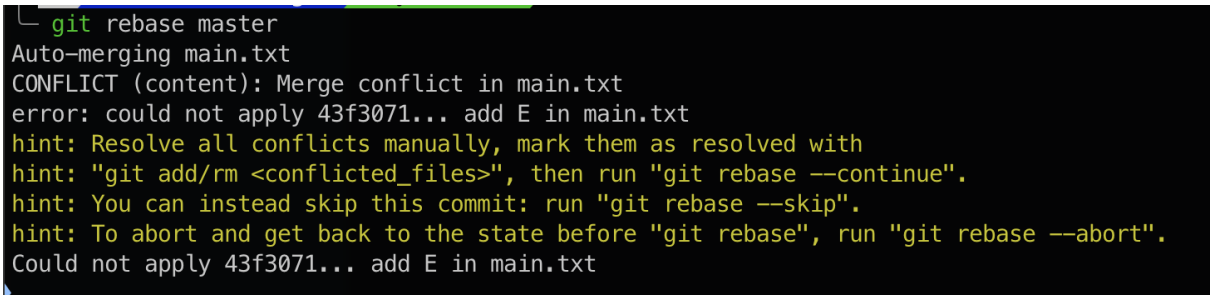
WARNING!

Never rebase commits that have been shared with others. If you have already pushed commits up to Github..DO NOT rebase them unless you are positive no one on the team is using those commits.

SERIOUSLY!

You do not want to rewrite any git history that other people already have. It's pain to reconcile the alternate history!

Rebase Conflict



Abort Rebasing

```
git rebase --abort
```

Continue Rebasing

```
git rebase --continue
```

Skip Rebasing

```
git rebase --skip
```

Cleaning Up History With Interactive Rebase

Rewriting History

Something we want to rewrite, delete, rename, or even reorder commits (before sharing them) We can do this using **git rebase!**

There are two main ways to use the git rebase command:

- as a alternative to merging
- as a cleanup tool

We are now going to look for the second way “as a cleanup tool”

Interactive Rebase

Running git rebase with the -i option will enter the interactive mode, while **allows us to edit commits, add files, drop commits, etc.** Note that we need to specify how far back we want to rewrite commits.

Also, notice that we are not rebasing onto another branch. Instead, we are rebasing a series of commits onto the HEAD they currently are based on.

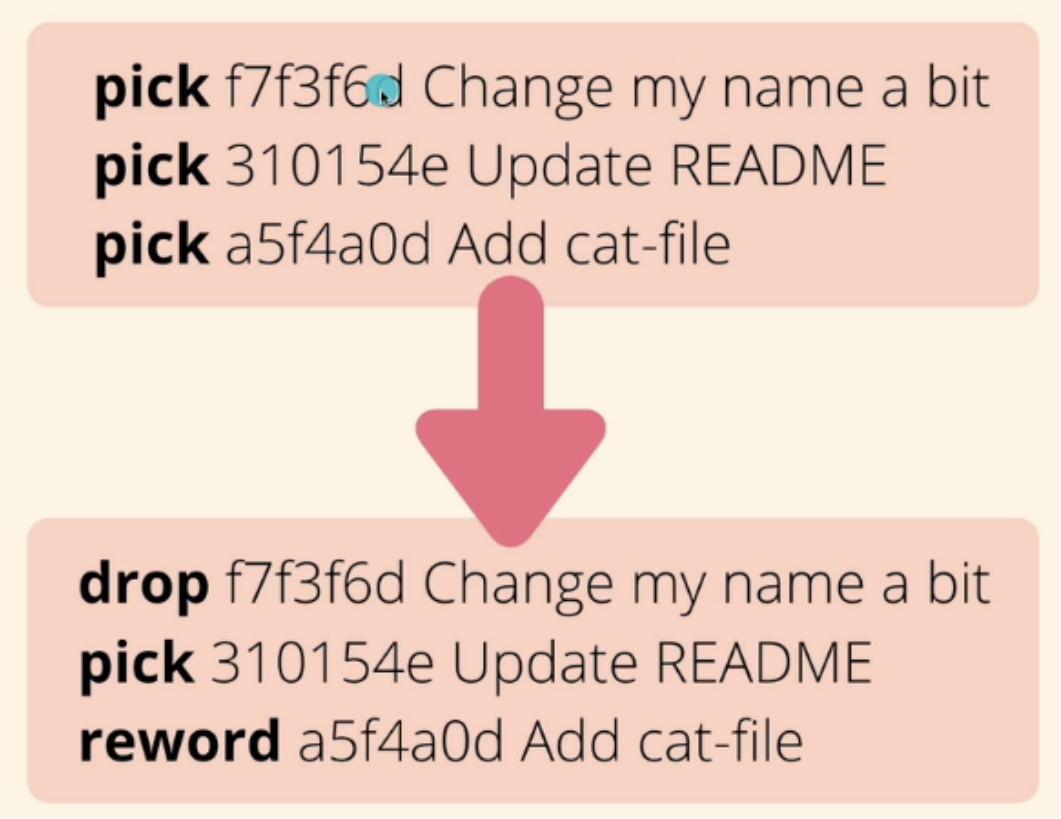
```
git rebase -i HEAD~4
```

After running this code an text editor window will popup

What Now?

In our text editor, we'll see a list of commits alongside a list of commands that we can choose from. Here are a couple of the more commonly used commands:

- pick - use the commit
- reword - use the commit, but edit the commit message
- edit - use commit, but stop for amending
- fixup - use commit contents but meld it into precious commit and discard the commit message
- drop - remove commit



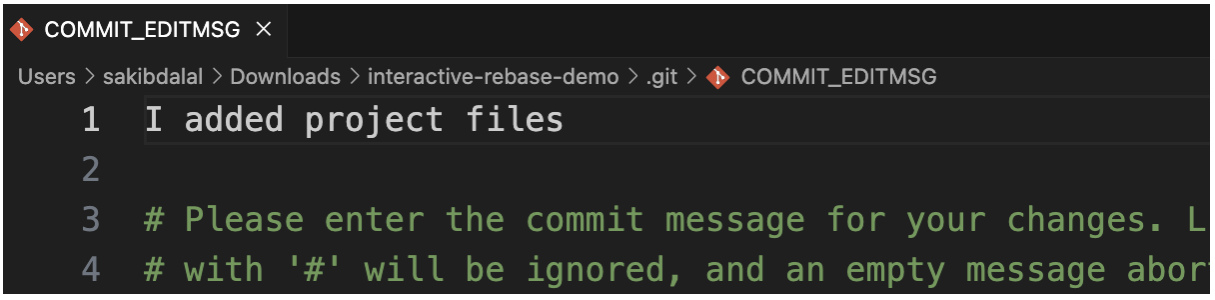
Reword

```
git rebase -i HEAD~9
```

Example: We want to rename the “I added project file” commit

```
pick cbee26b I added project files
pick 519aab6 add basic HTML boilerplate
pick 240827f add bootstrap
pick 6e39a76 whoops forgot to add bootstrap js script
pick 4ff2290 add top navbar
pick 2a45e71 fix navbar typos
pick 655204d fix another navbar typo
pick 2029e20 my cat made this commit
pick 3b23c17 Create README.md
```

we will **use reword instead of pick,** and close the editor. New text editor window will popup asking to enter new name for existing commit.



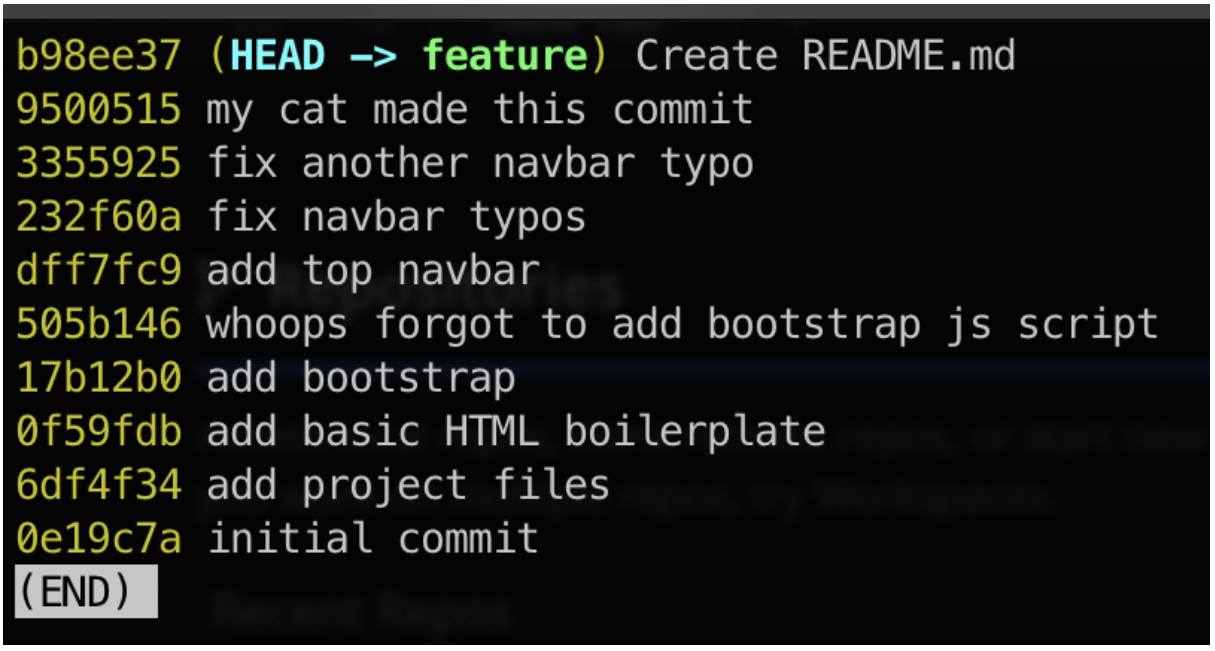
TO

Save and close the editor. we will find commit name changed.

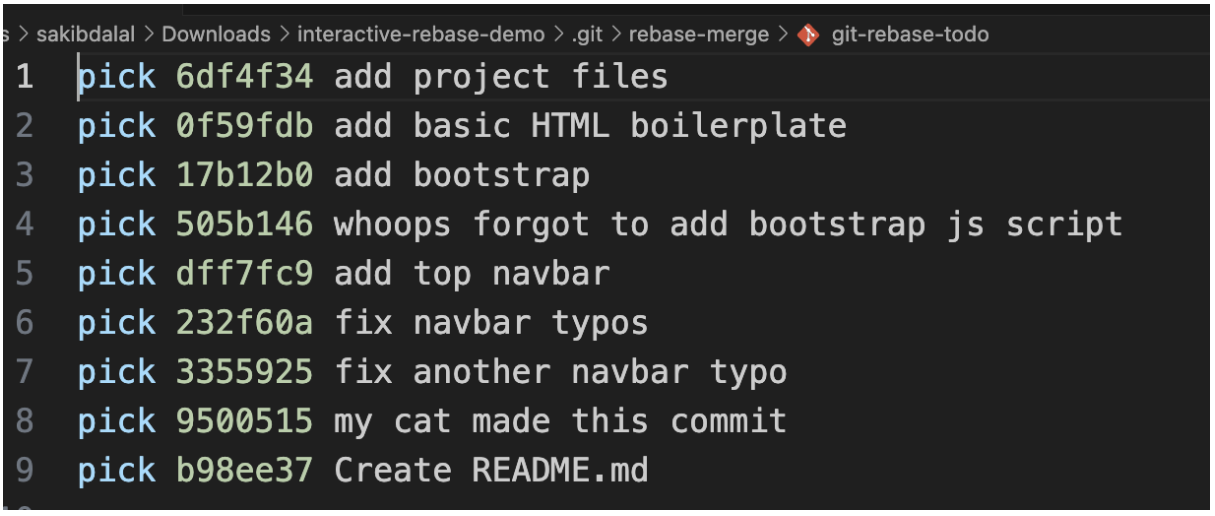
```
git log --oneline
```

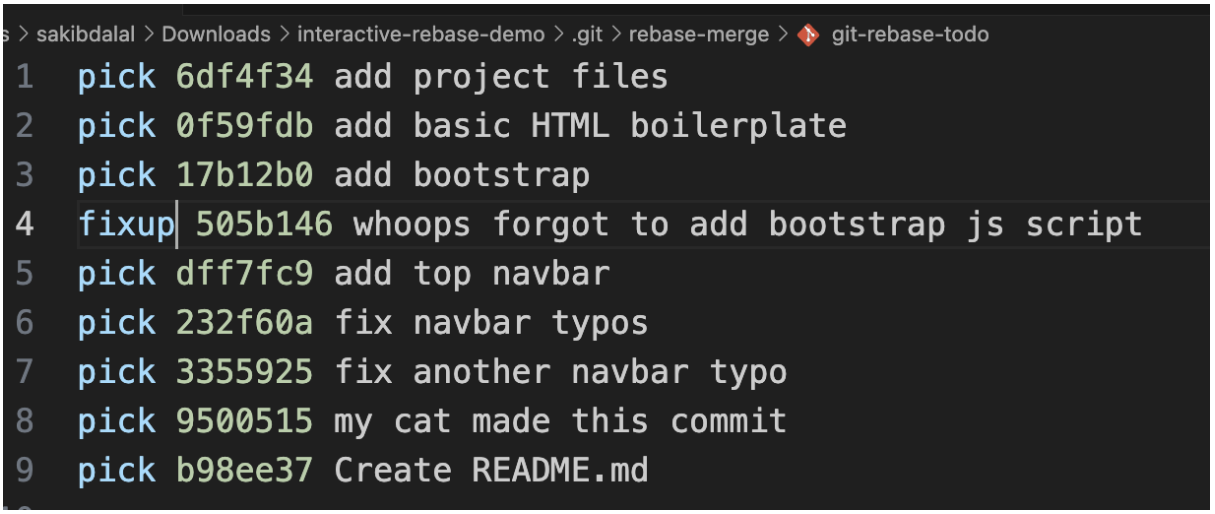
Note: we can reword multiple commits

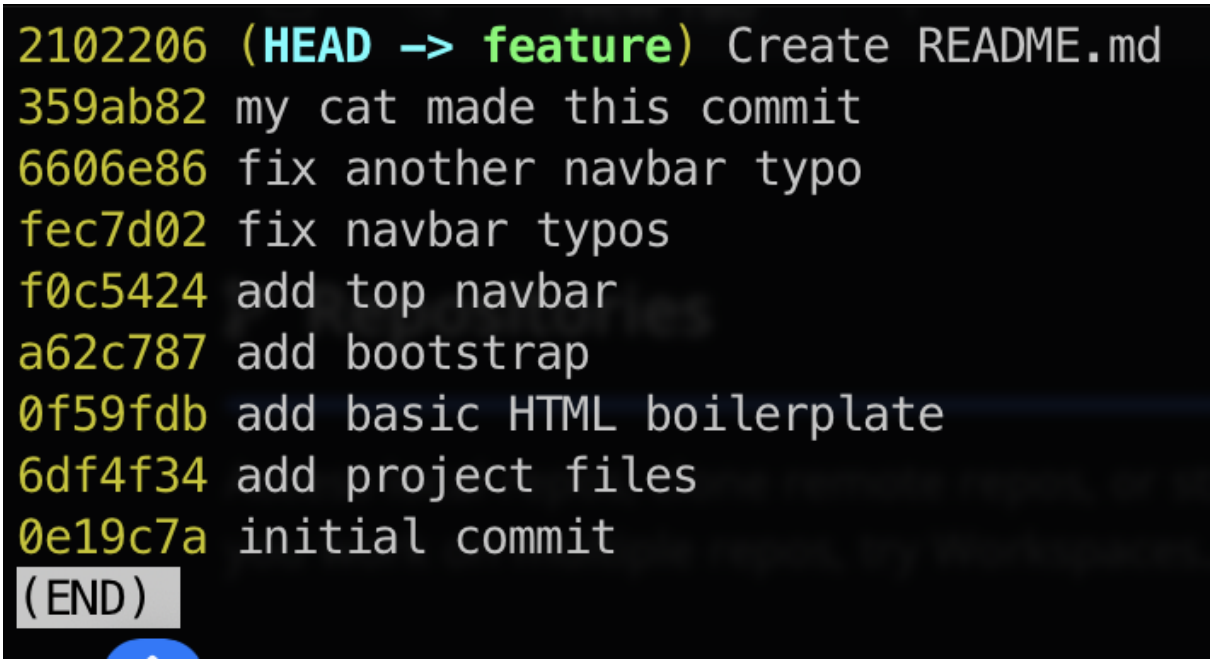
Fixup



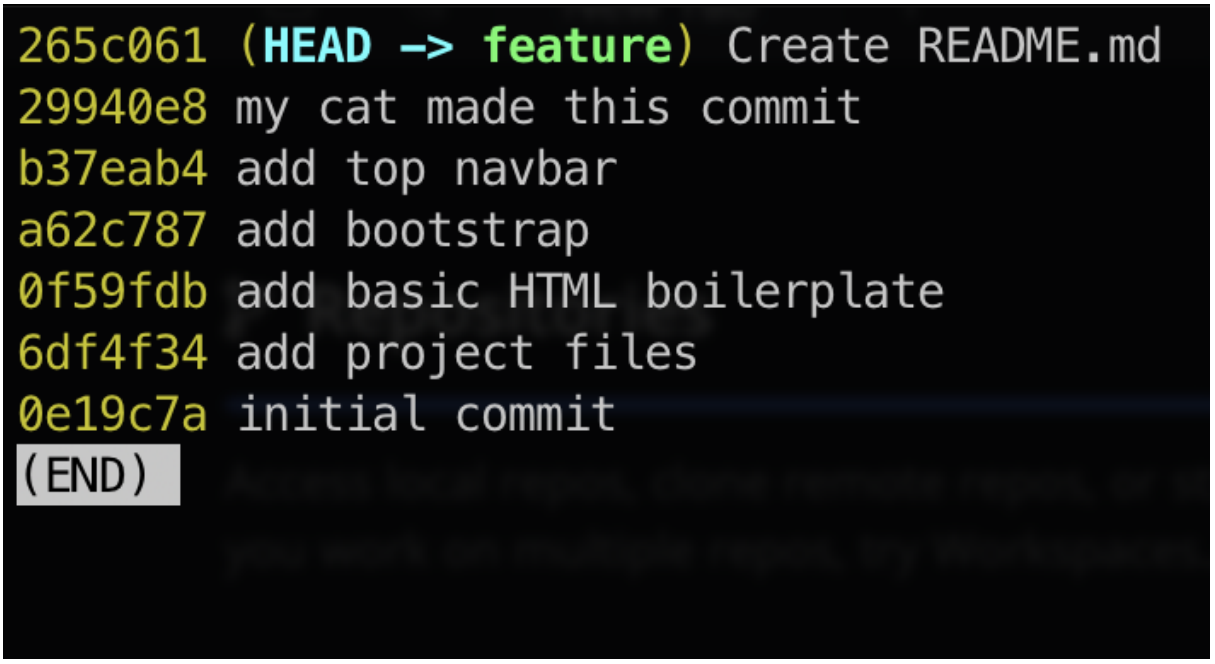

```
git rebase -i HEAD~9
```



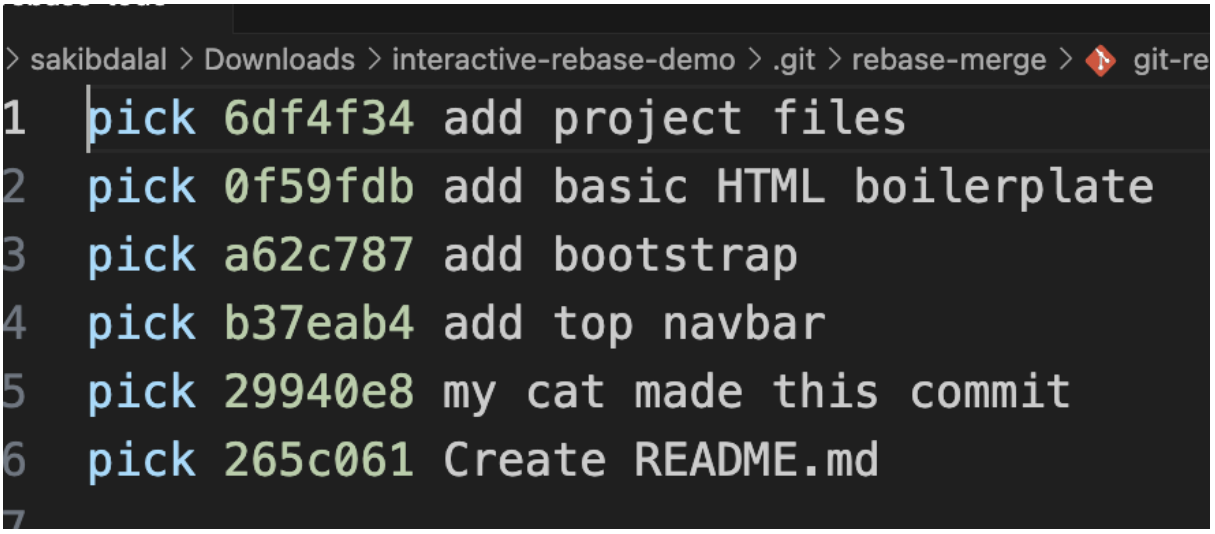

s > sakibdalal > Downloads > interactive-rebase-demo > .git > rebase-merge > git-rebase-todo
1 pick 6df4f34 add project files
2 pick 0f59fdb add basic HTML boilerplate
3 pick 17b12b0 add bootstrap
4 fixup 505b146 whoops forgot to add bootstrap js script
5 pick dff7fc9 add top navbar
6 pick 232f60a fix navbar typos
7 pick 3355925 fix another navbar typo
8 pick 9500515 my cat made this commit
9 pick b98ee37 Create README.md

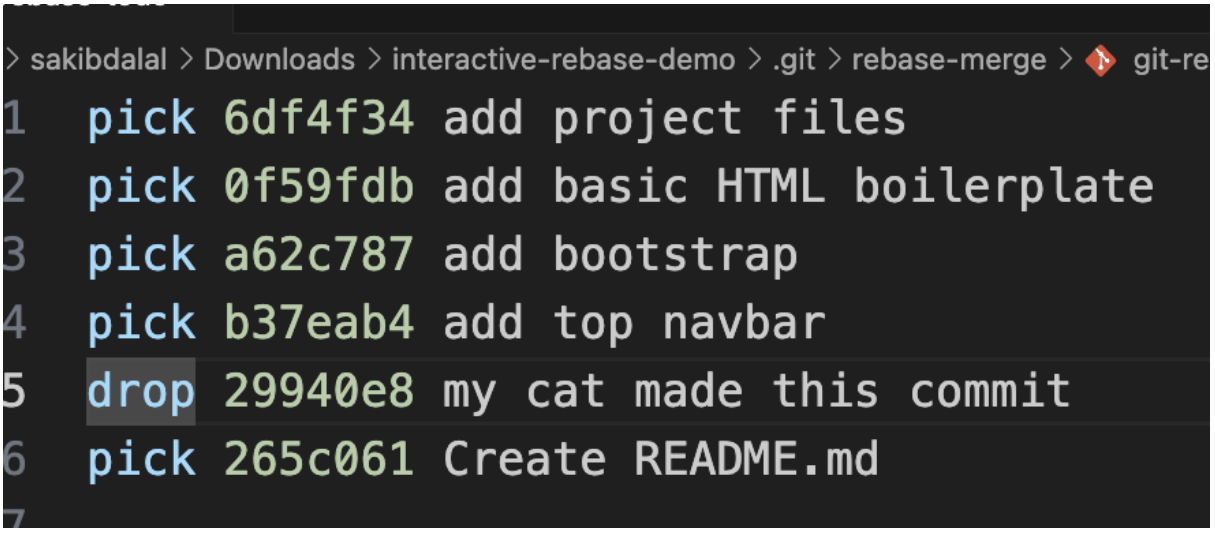

2102206 (HEAD -> feature) Create README.md
359ab82 my cat made this commit
6606e86 fix another navbar typo
fec7d02 fix navbar typos
f0c5424 add top navbar
a62c787 add bootstrap
0f59fdb add basic HTML boilerplate
6df4f34 add project files
0e19c7a initial commit
(END)

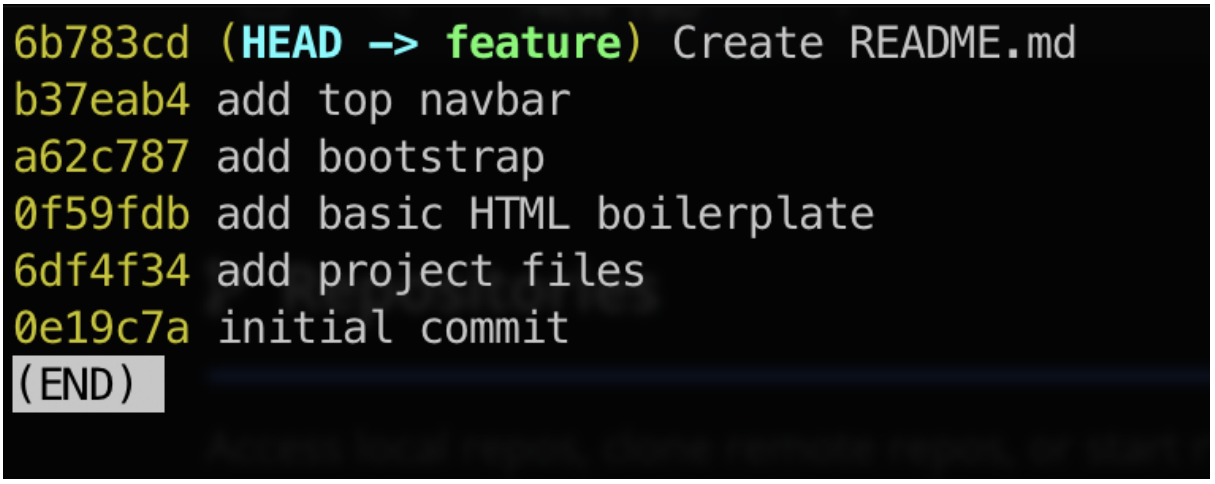
Drop




```
git rebase HEAD~6
```



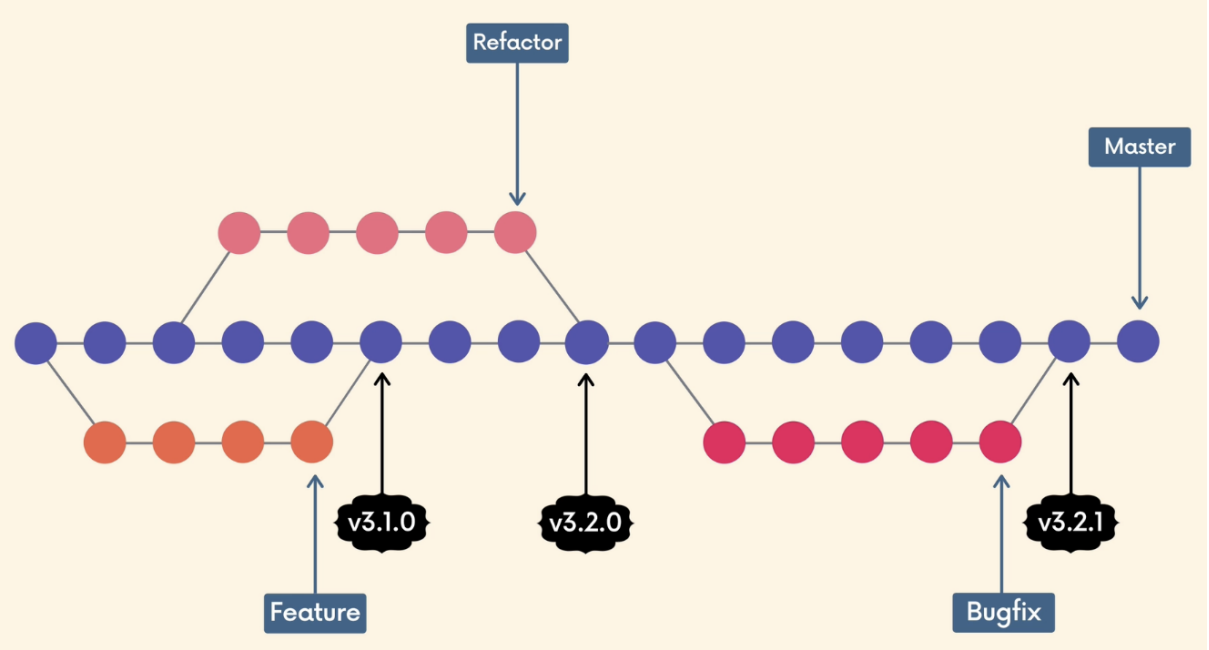




Git Tags

Tags are pointers that refer to particular points in Git history. We can make a particular moment in time with a tag. Tags are most often used to make version releases in projects(v1.0, v1.1, etc)

Think of tags as branch references that do NOT CHANGE. Once a tag is created, it always refers to the same commit. It's just a label for a commit.



The Two Types of Tags

There are two types of Git tags we can use: Lightweight and annotated tags.

Lightweight tags are…lightweight. They are just a name/label that points to a particular commit.

annotated tags store extra meta data including the author's name and email, the date, and a tagging message (like a commit message)

Versioning

Semantic Versioning

The semantic versioning spec outline a standardised versioning system for software releases. It provide a consistent way for developers to give meaning to their software releases (how big of a change is this release??)

Versions consists of three numbers separated by periods.

Ex: 2.4.1

Initial Release

Typically, the first release is 1.0.0

1.0.0

Patch Release

Patch releases normally do not contain new features or significant changes. They typically signify bug fixes and other changes that do not impact how the code is used

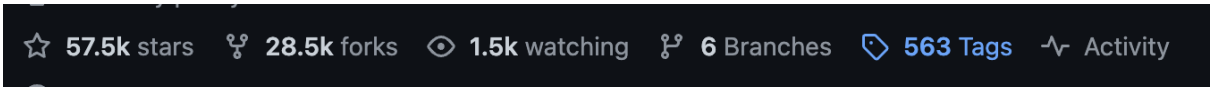
1.0.1

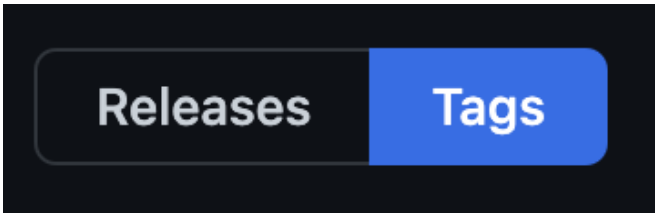
Minor Release

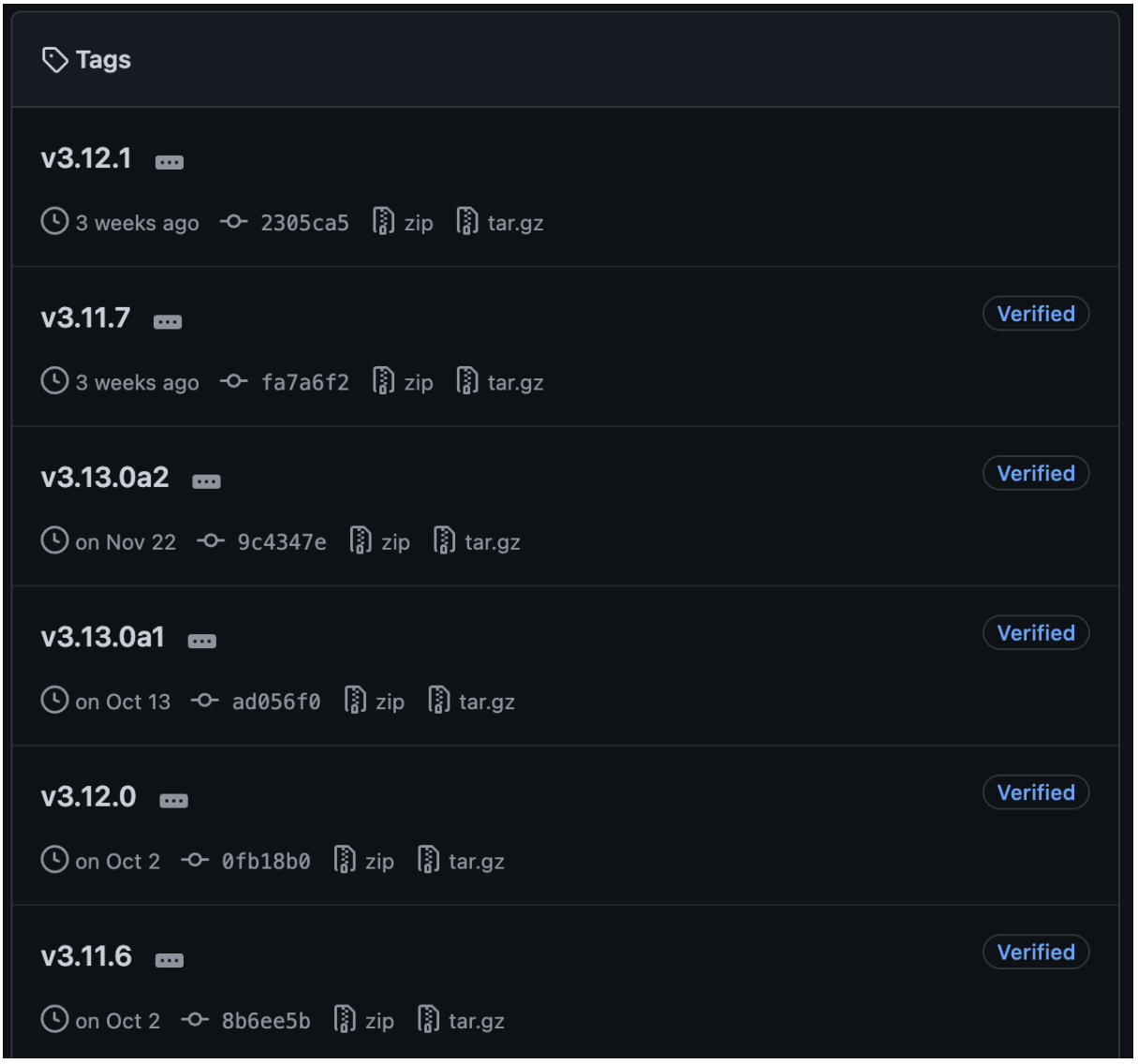
Minor releases signify that new features or functionality have been added, but the project is still backwards compatible. No breaking changes. The new functionality is optional and should not force users to rewrite their own code.

1.1.0

We can find the tags for a particular repo.







Viewing Tags

Git tag will print a list of all the tags in the current repository.

```
git tag
```

We can search for tags that match a particular pattern by using git tag -l and then passing in wildcard pattern. For example, git tag -l “*beta” will print a list of tags that include “beta” in their name.

```
git tag -l "*beta*"
OR
git tag -l "v17*"

AND
```

```
git tag -l
```

Checking Out Tags

To view the state of a repo at a particular tag, we can use git checkout <tag>. This puts us in detached HEAD!

```
git checkout <tag>
```

Comparing two tags with git diff

```
git diff <tag-01> <tag-02>

ex.
git diff v17.0.0 v17.0.1
```

Creating Lightweight Tags

To create a lightweight tag, use **git tag <tag-name>** By default, Git will create the tag referring to the commit that HEAD is referencing.

```
git tag <tag-name>
```

ex.

```
git tag v1.0.0
```

Annotated Tags

Use git tag -a to create a new annotated tag. Git will then open your default text editor and prompt you for additional information.

Similar to git commit, we can also use the **-m** option to pass a message directly and forge the opening of the text editor.

```
git tag -a <tag-name>
```

ex:

```
git tag -a v1.0.0
```

an editor will prompt, asking to include some text info related to this tag.

to see the meta data of a tag we use:

```
git show <tag-name>
```

Tagging Previous Commits

So far we've seen how to tag the commit that HEAD references. We can also tag an older commit by providing the commit hash: git tag -a <tag-name> <commit-hash>

```
git tag <tag-name> <commit-hash>
```

ex.

```
git tag v1.0.0 0e19c7a

git log --oneline
git tag
```

Forcing Tags

Git will yell at us if we try to reuse a tag that is already referring to a commit. If we use the -f option, we can FORCE our tag through.

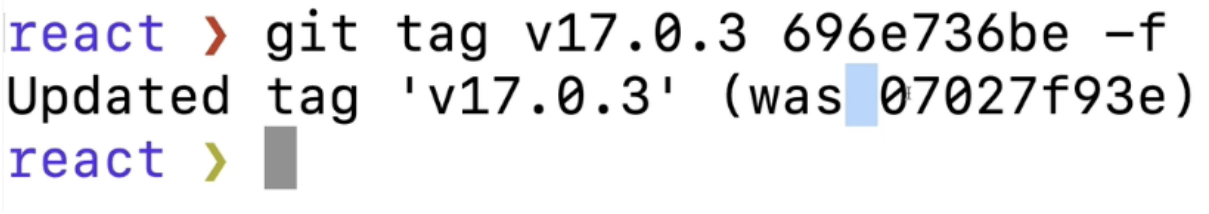
```
git tag -f <tag-name>
```

ex.

```
react ❯ git tag v17.0.3 696e736be
fatal: tag 'v17.0.3' already exists
react ❯
```

To resolve this error we uses force tag

```
git tag v17.0.3 696e736be -f
```




```
git log --oneline
```

Deleting Tags

To delete a tag, use git tag -d <tag-name>

```
git tag -d <tag-name>
```

ex.

```
git tag -d v1.0.0

git log --oneline
git tag
```

Pushing Tags

By default, the git push command doesn't transfer tags to remote servers. If you have a lot of tags that you want to push up at once, you can use the --tags option to the git push command. This will transfer all of your tags to the remote server that are not already there.

```
git push --tags

git push origin --tags
AND
# for single tag
git push origin <tag-name>
```

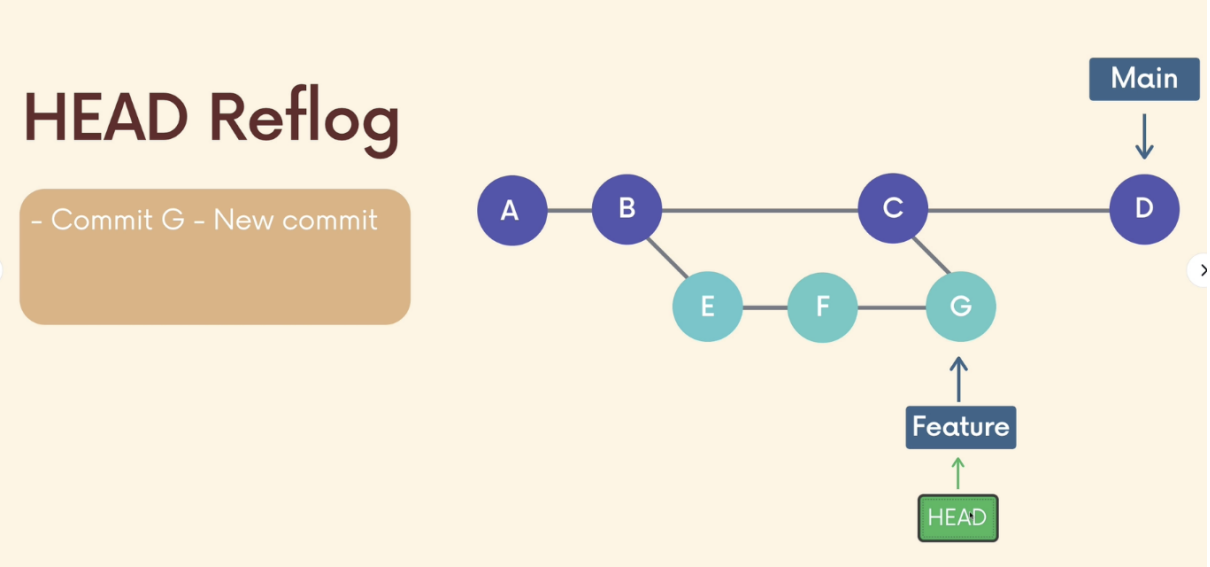
Reflogs

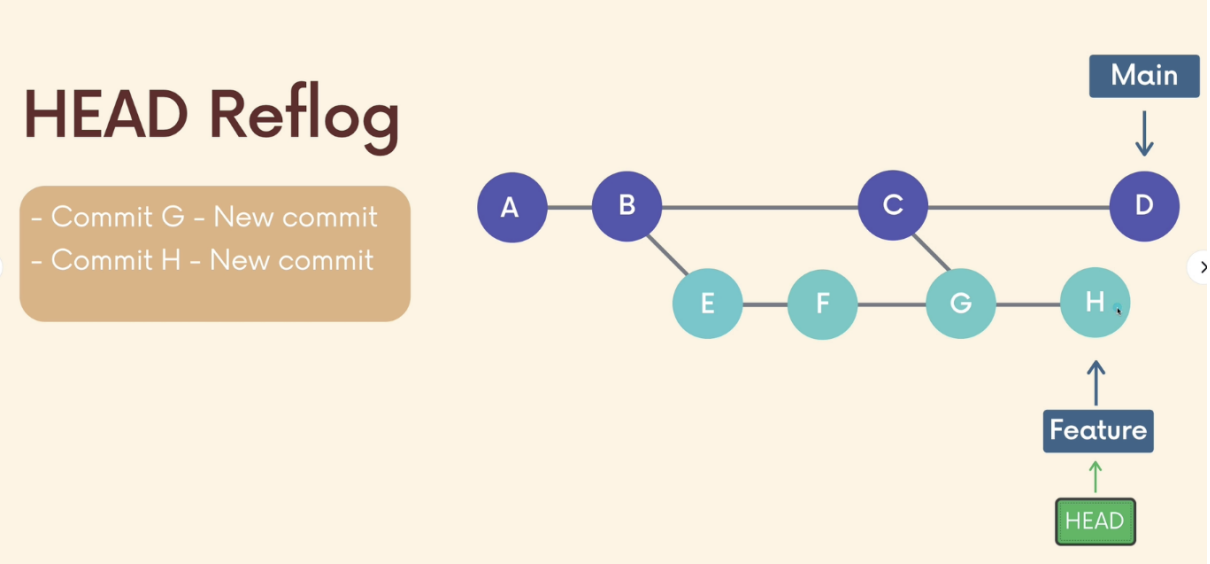
Git keeps a record of when the tips of branches and other references were updated in the repo.

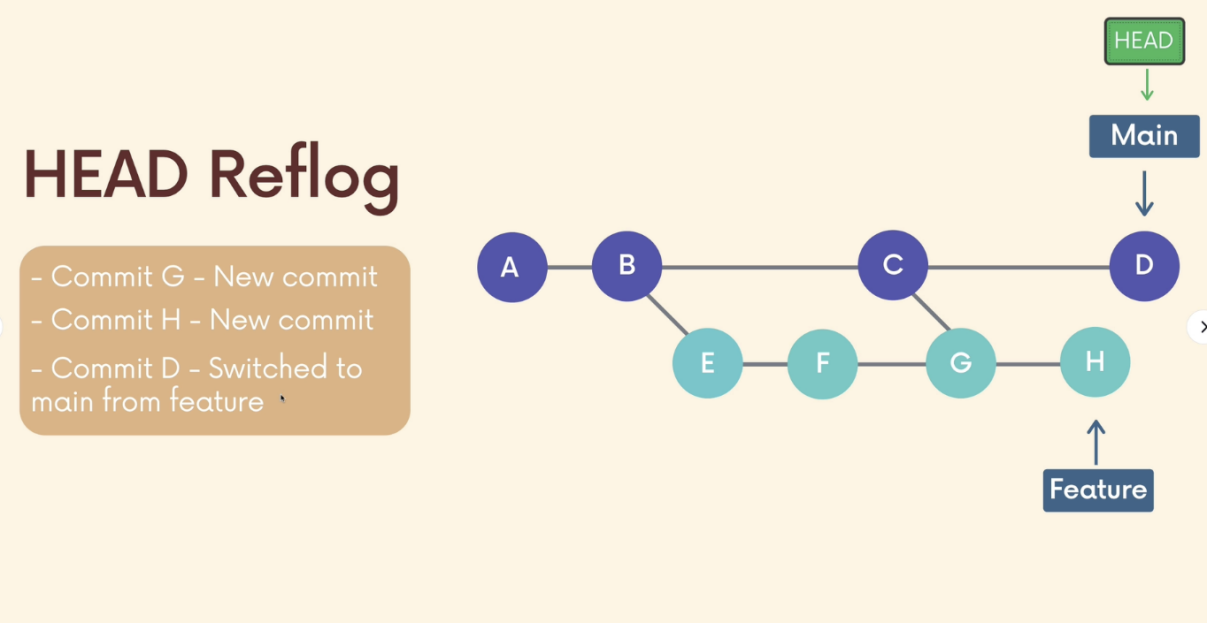
We can view and update these references logs using the git reflog command.

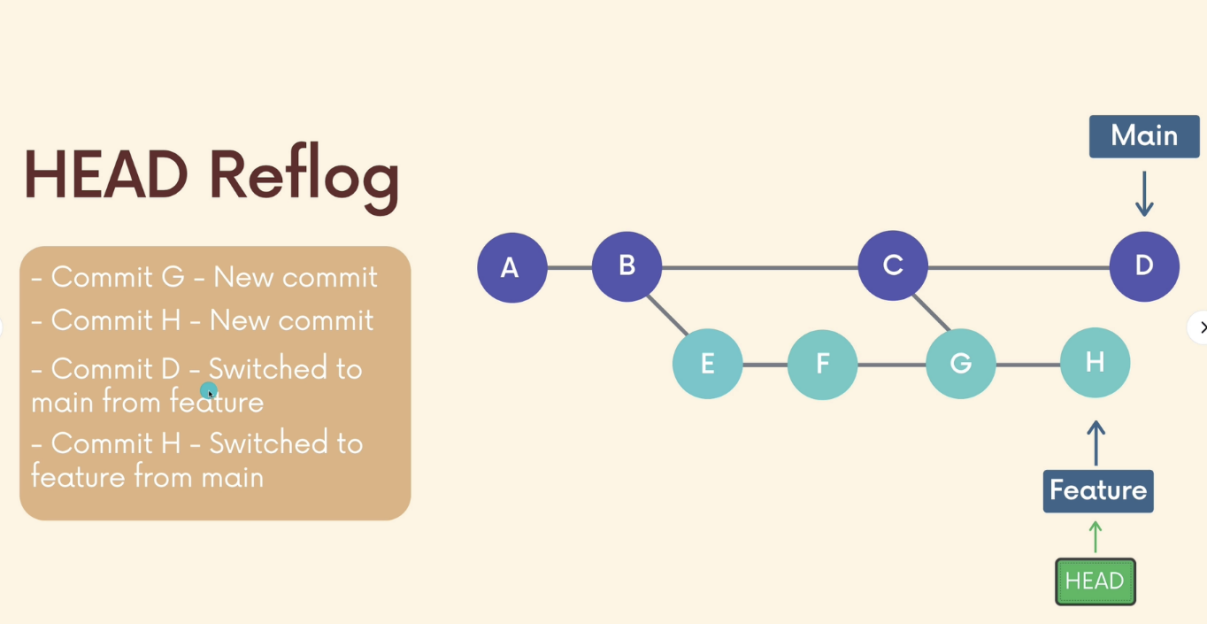
```
git reflog
```

Ex.









Limitations!

Git only keeps reflogs on your local activity. They are not shared with collaborators.

Reflogs also expire. Git cleans out old entries after around 90 days, through this can be configured.

Git Reflog

The **git reflog** command accepts subcommands **show, expire, delete, and exists**. Show is the only commonly used variant, and it is the default subcommand.

git reflog show will show the log of a specific reference (it default to HEAD).

For example, to view the logs for the tip of the main branch we could run **git reflog show main.**

```
git reflog show HEAD
```

```
git reflog
```

```
git reflog show main
```

Reflog References

We can access specific git refs is **name@{qualifier}**. We can use this syntax to access specific ref pointer and can pass them to other commands including checkout, reset, and merge.

```
name@{qualifier}
```

Ex.

```
git reflog show HEAD@{10}
```

```
git checkout HEAD@{2}
```

```
git diff HEAD@{0} HEAD@{5}
```

Time References

Every entry in the reference logs has a timestamp associated with it. We can filter reflogs entries by time/date by using time qualifiers like:

- 1.day.ago
- 3.minutes.ago
- yesterday
- Fri, 12 Feb 2021 14:06:21 -0800

```
git reflog master@{one.week.ago}
```

```
git checkout bugfix@{2.days.ago}
```

```
git diff main@{0} main@{yesterday}
```

Reflogs Rescue

We can sometimes use reflog entries to access commits that seem lost and are not appearing in git log.

#️⃣ accidentally reset a commit, we can use git reflog to bring back the commit.

```
git reset --hard <commit-hash>
```

```
git reflog show
```

```
#
git reset --hard master@{1}

OR

git reset --hard <reflog-commit-hash>
```

The deleted commit will come back as HEAD.

Note: We can use reflog show if we accidentally use rebase or rebase -i. Use git reset —hard <commit-hash-from-reflog>

Writing Custom Git Aliases

Global Git Config

Git looks for the global config file at either **~/.gitconfig** or **~/.config/git/config**. Any configuration variables that we change in the file will be applied across all Git repos.

We can also alter configuration variables from the command line if preferred.

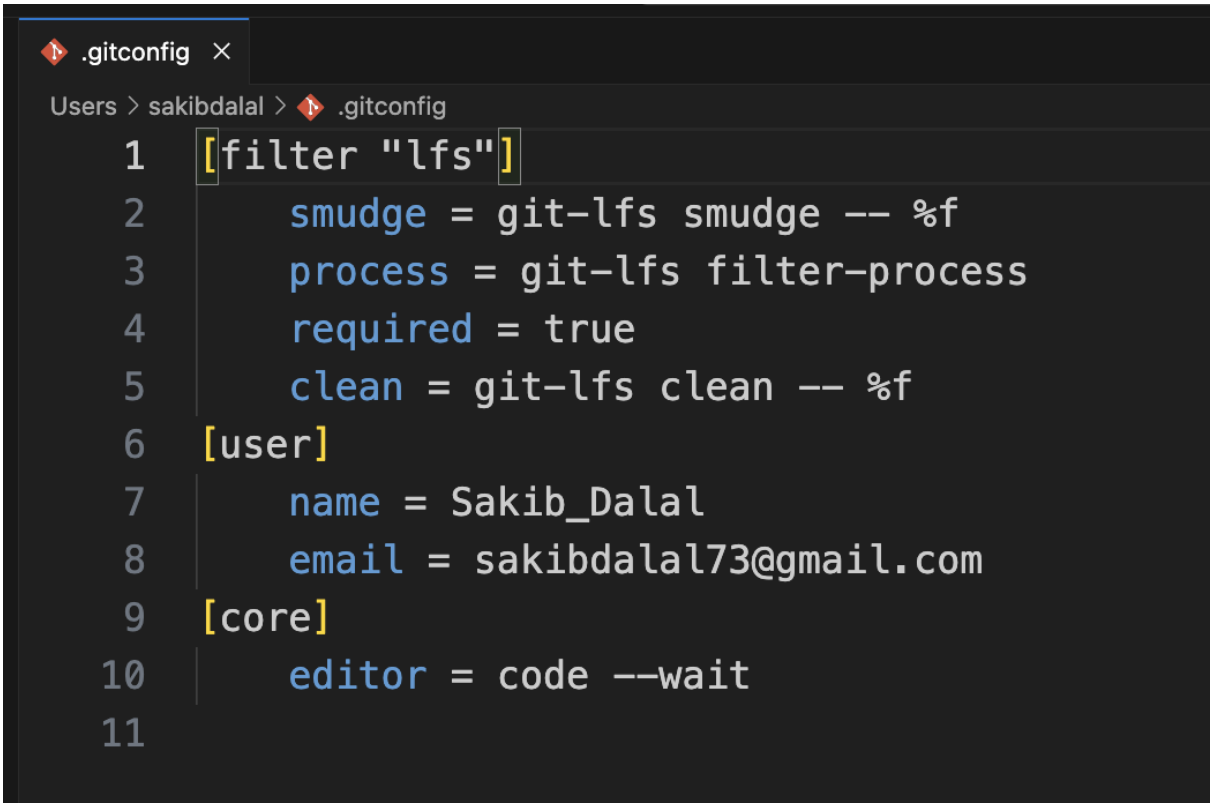
Basic command line config.

```
git config --global user.name
git config --global user.name <name>

git config --global user.email
```

We can open the global config file into our editor, just use

```
code ~/.gitconfig

OR

open ~/.gitconfig
```

It will show



we can change name, email, etc..

If we want to change user name or email we refer to the users entity.

Adding Aliases

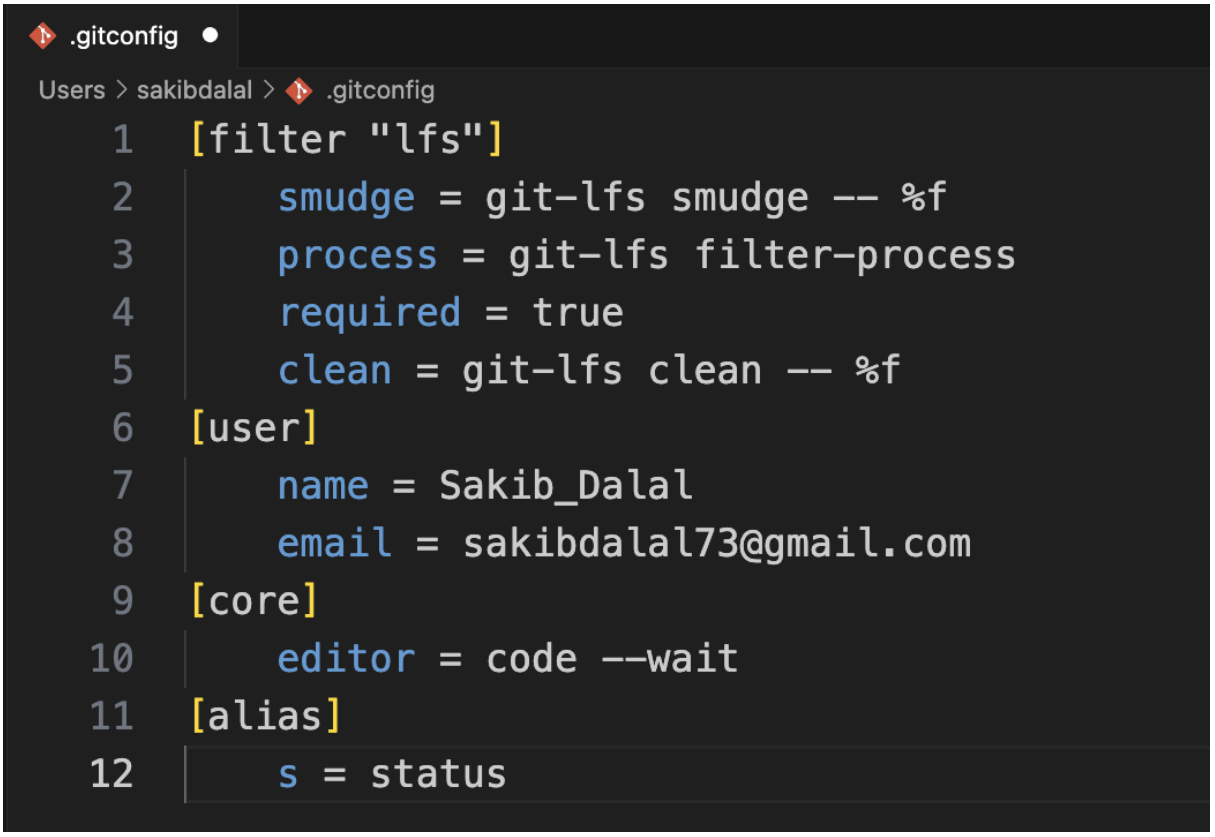
We can easily set up Git aliases to make our Git experiences a bit simple and faster.

For example, we could define an alias “**git ci**” instead, of having to type “**git commit**”

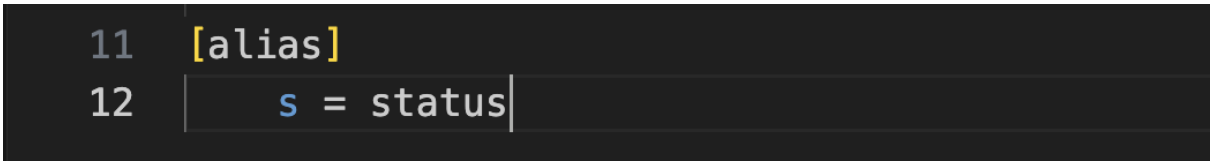
Or, we could define a custom **git lg** command that prints out a custom formatted commit log

Example: Instead of using “git status” we can define our custom alias.

OPEN THE .**gitconfig** FILE AN MAKE AN ENTITY NAMED **[alias]**

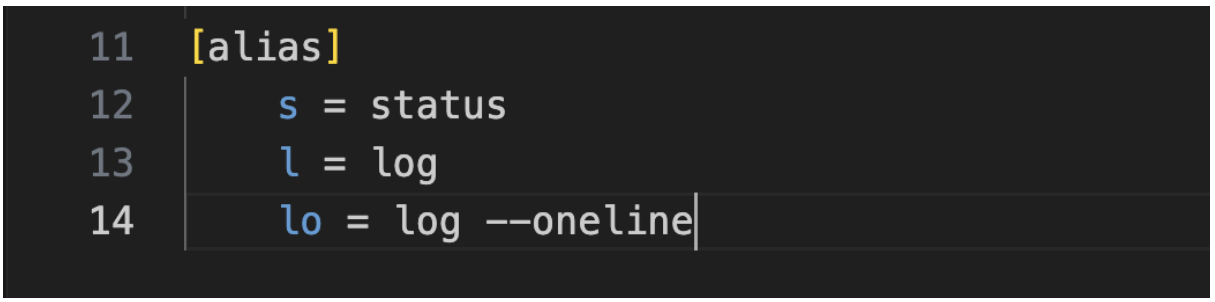


Custom alias




```
git s
# will work as git status
```

also:



Configure Aliases from Command Line

```
git config --global alias.s status

git s
# work as git status
```

Aliases With Arguments # ex. git commit -m “msg”

```
11  [alias]
12      s = status
13      l = log
14      lo = log --oneline
15      cm = commit -m
```

Where:

```
15      cm = commit -m
```

Requires a argument

```
git cm "Update file"
```

```
git cm <"message">

# work as git commit -m "message"
```

Similar :

```
[alias]
    s = status
    l = log
    lo = log --oneline
    cm = commit -m
    a = add
```