

The Git & GitHub Bootcamp

Contents

| | |
|---|-----|
| 1 Introduction..... | 3 |
| 2 Installation..... | 34 |
| 3 git basics..... | 43 |
| 4 Committing in detail..... | 76 |
| 5 Branching..... | 84 |
| 6 Merging..... | 113 |
| 7 Diff..... | 125 |
| 8 Stashing..... | 137 |
| 9 Undoing stuff & Time traveling..... | 152 |
| 10 GitHub basics..... | 179 |
| 11 Fetching and Pulling..... | 213 |
| 12 More on GitHub..... | 238 |
| 13 git workflows (for collaboration)..... | 248 |
| 14 Rebasing & Cleaning up history..... | 285 |
| 15 git Tags marking important moments in history..... | 298 |
| 16 git behind the scenes..... | 311 |
| 17 Reflogs retrieving lost work..... | 335 |
| 18 Aliases..... | 343 |
| 19 Squash..... | 345 |
| 20 Cherry Pick..... | 348 |
| 21 Graph..... | 350 |

1

Introduction

Welcome!



What Is Git?



≡

Git

The world's most popular
version control system



Version Control System?





Version Control

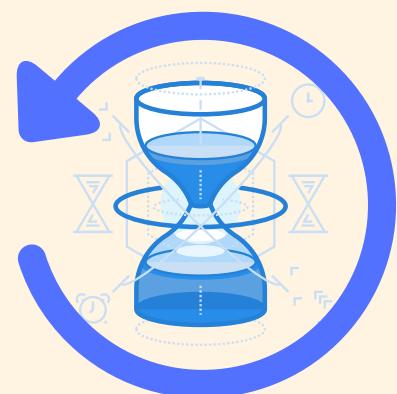
Version control is software that tracks and manages changes to files over time.



Version Control

Version control systems generally allow users to revisit earlier versions of the files, compare changes between versions, undo changes, and a whole lot more.

* We'll go into more detail shortly!

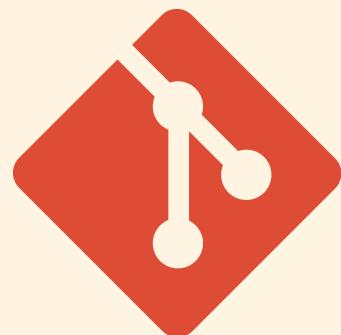




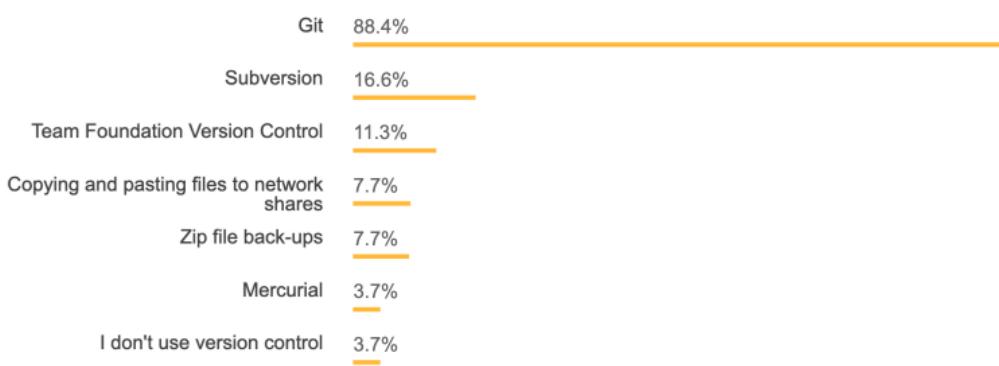
Git is just one VCS

Git is just one of the many version control systems available today. Other well-known ones include Subversion, CVS, and Mercurial.

They all have similar goals, but vary significantly in how they achieve those goals. Fortunately, we only need to care about Git because...



Git is the clear "winner"



In Stack Overflow's 2018 Developer Survey, nearly 90% of respondents reported Git as their version control system of choice. Over the last few years, the survey hasn't even bothered to ask about version control because Git is so widely used.





Git-ing To The Point

What exactly does Git do for us?



Git helps us...

Track changes across multiple files

Compare versions of a project

"Time travel" back to old versions

Revert to a previous version

Collaborate and
share changes

Combine changes



The Big Picture



Oh boy, I sure do
love playing my
video games!

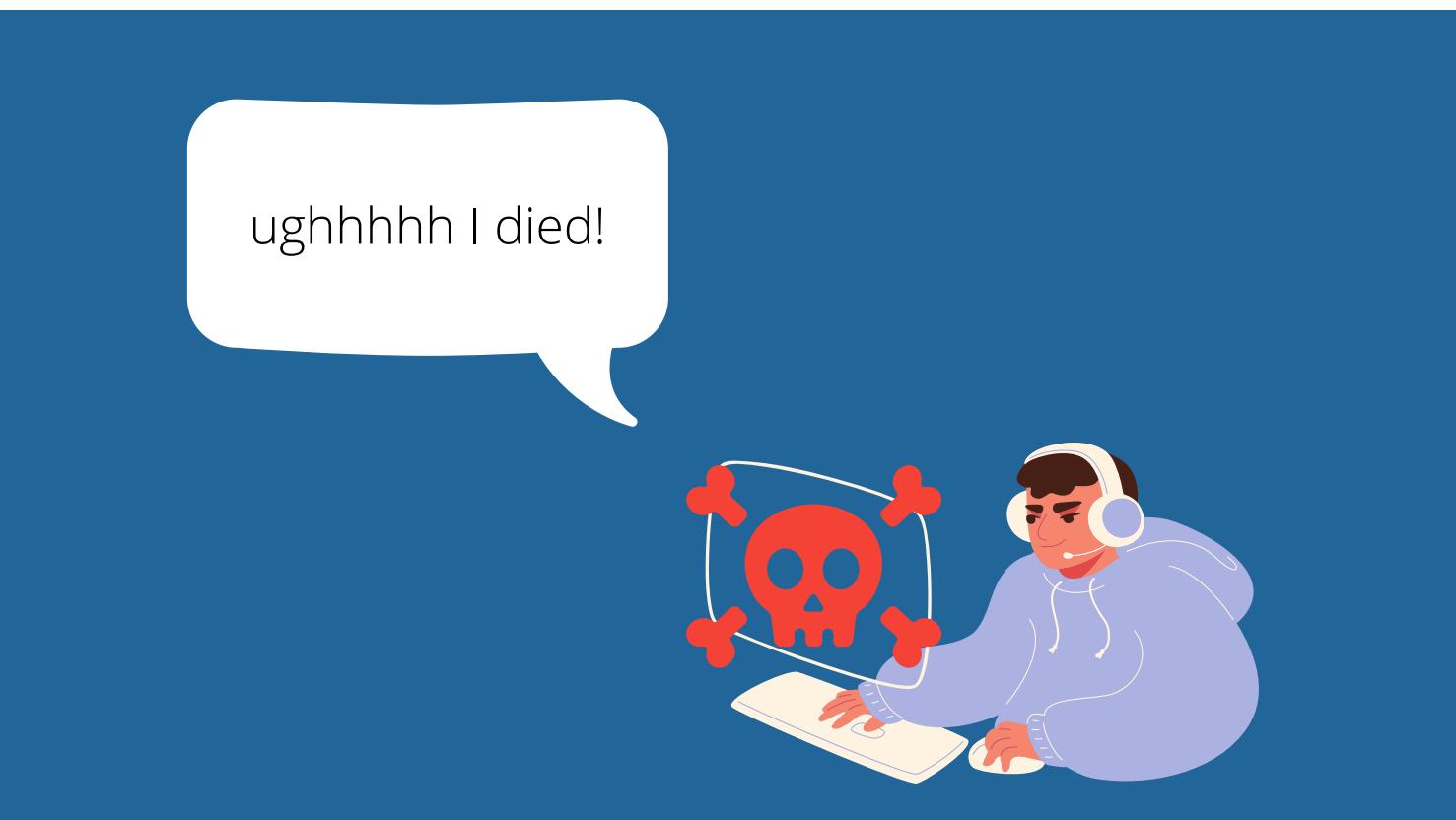


I'm going to save
my game now in
case I die soon!



Oh jeez, this is
going to be a
difficult fight!





ughhhh I died!



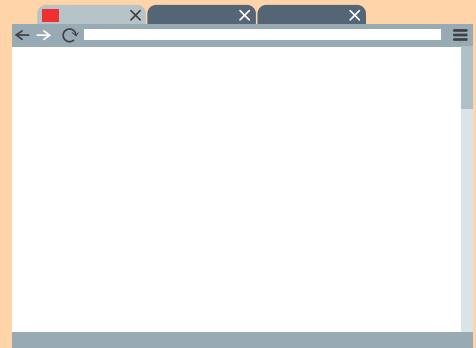
Thank heavens I
saved my game!
I can just revert!



Let's Look At An Example

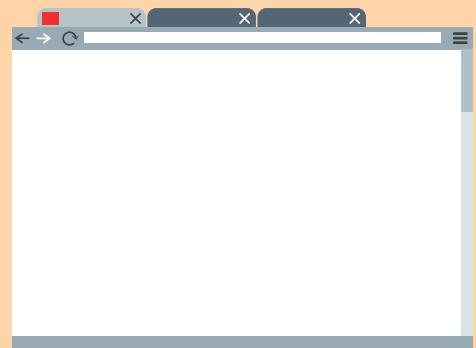


I Start A New Project!



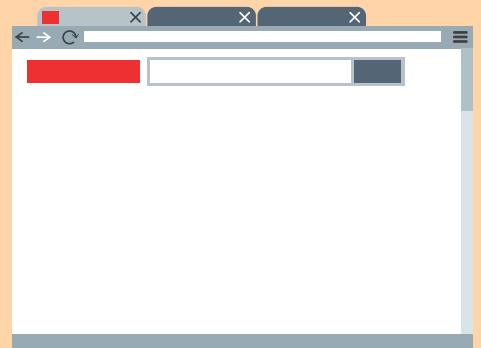
Add A Checkpoint

Initialize Project



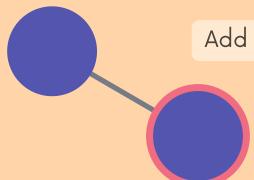
I work on the navbar

Initialize Project

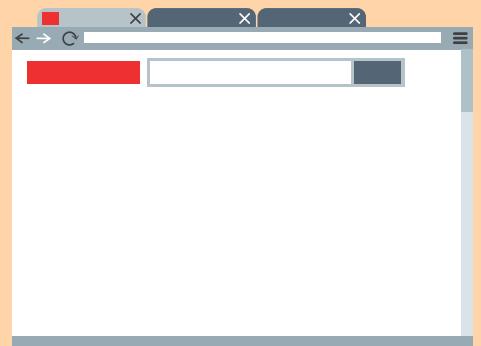


Add A Checkpoint

Initialize Project

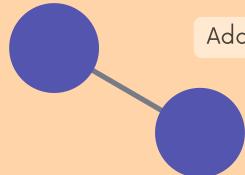


Add Top Navbar

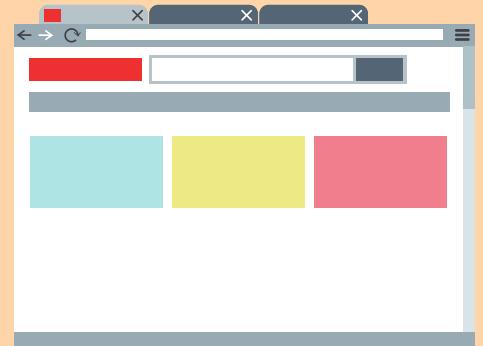


I add some content

Initialize Project

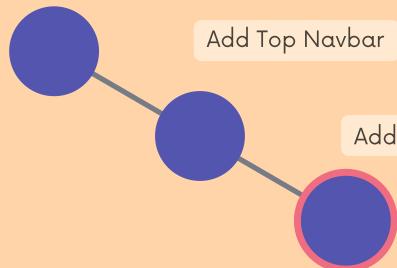


Add Top Navbar



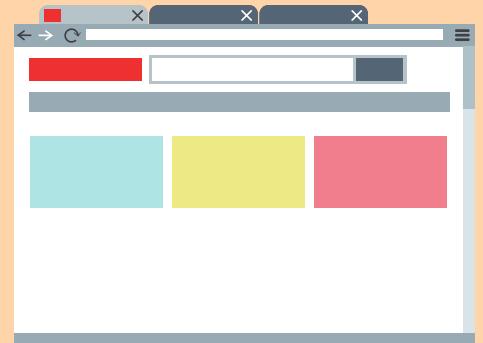
Add A Checkpoint

Initialize Project



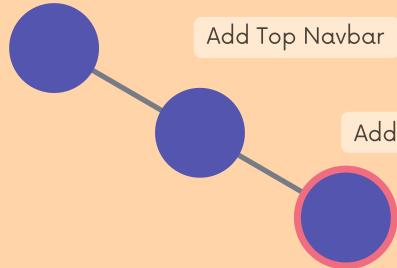
Add Top Navbar

Add First Row



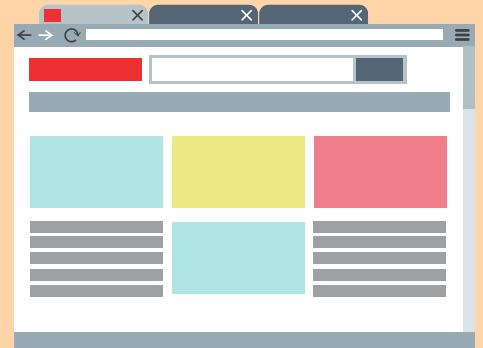
I add more content

Initialize Project



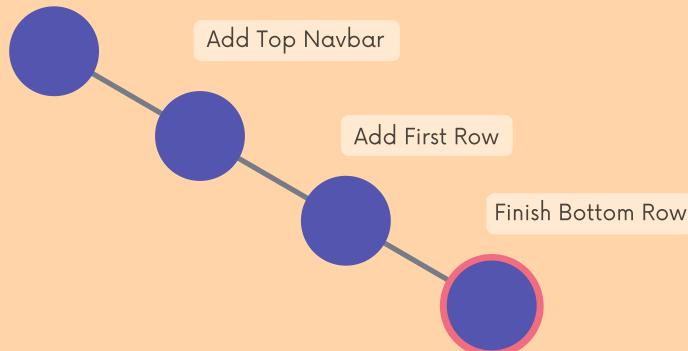
Add Top Navbar

Add First Row



Add A Checkpoint

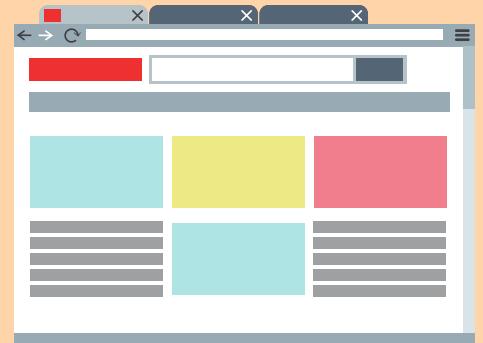
Initialize Project



Add Top Navbar

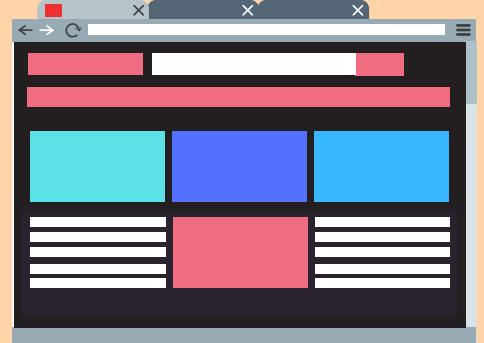
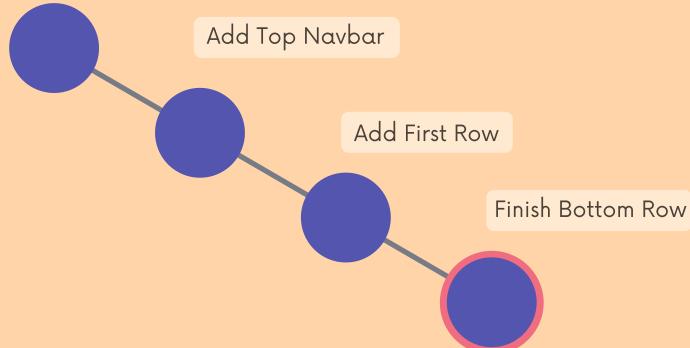
Add First Row

Finish Bottom Row



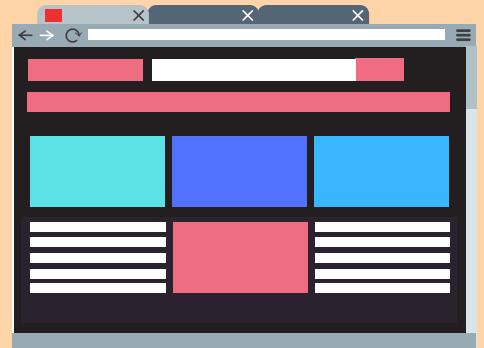
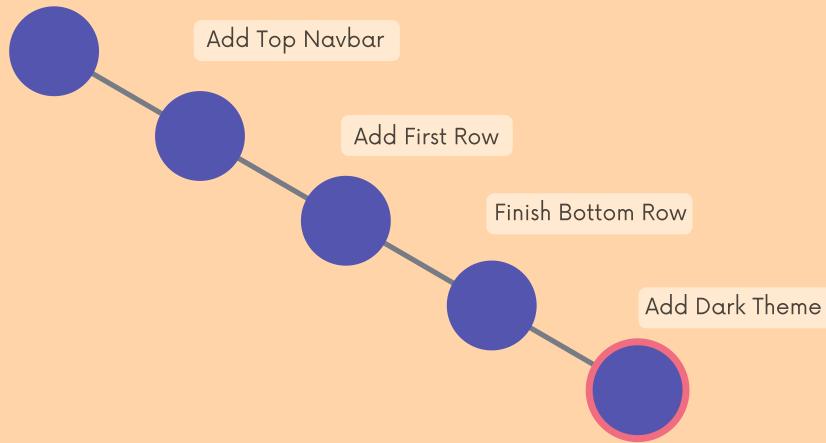
I change the theming

Initialize Project



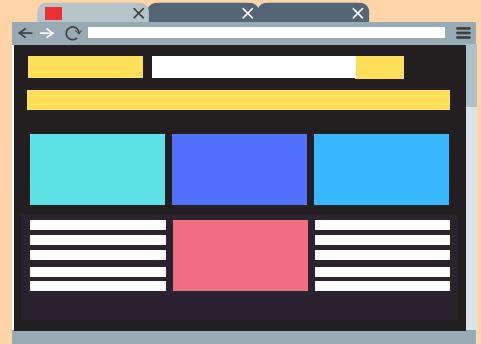
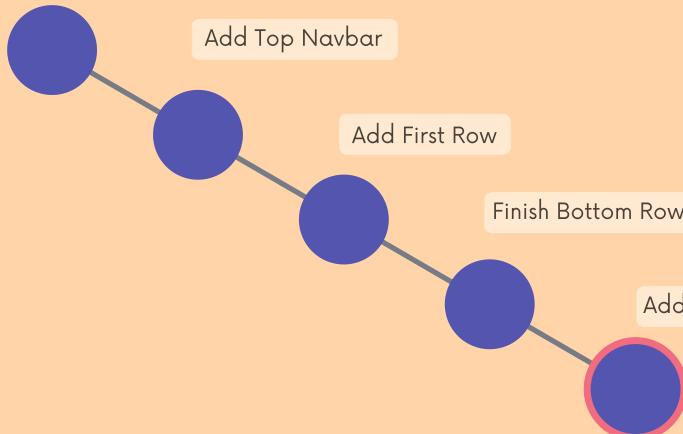
Add A Checkpoint

Initialize Project



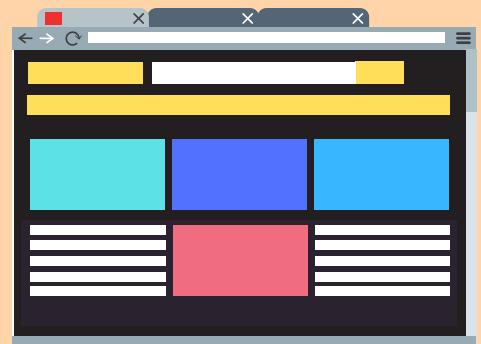
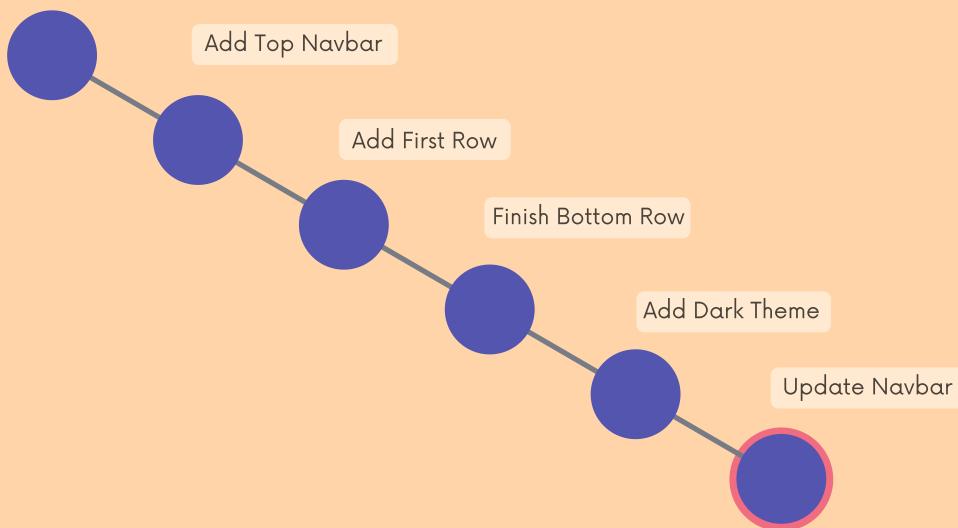
I alter the navbar

Initialize Project



Add A Checkpoint

Initialize Project





ANGRY BOSS SAYS...
THE COLORS ARE BAD!

No problem!

Initialize Project

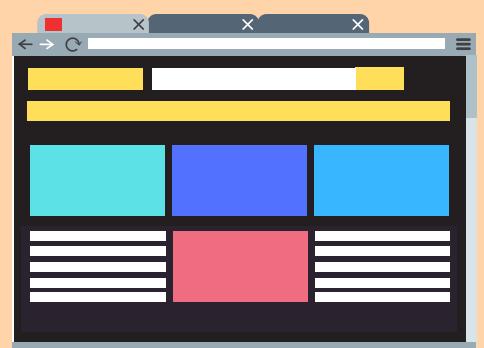
Add Top Navbar

Add First Row

Finish Bottom Row

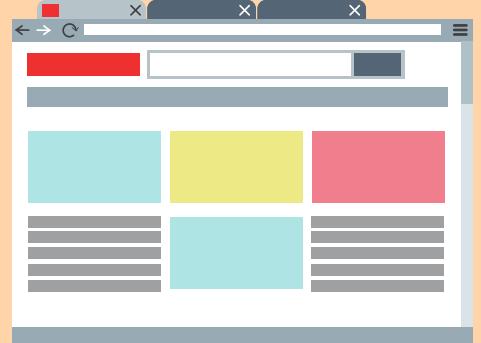
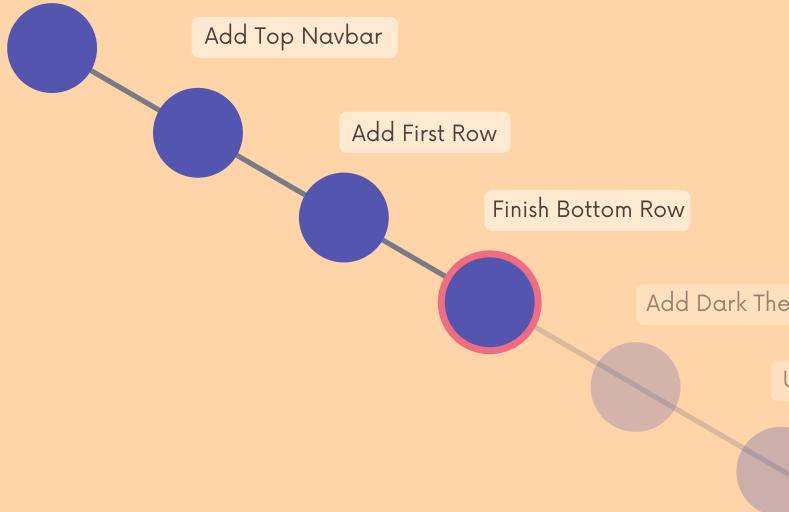
Add Dark Theme

Update Navbar



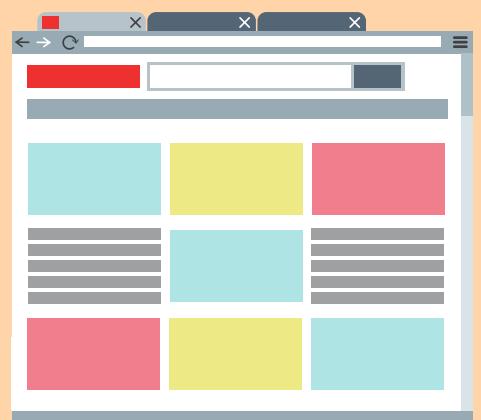
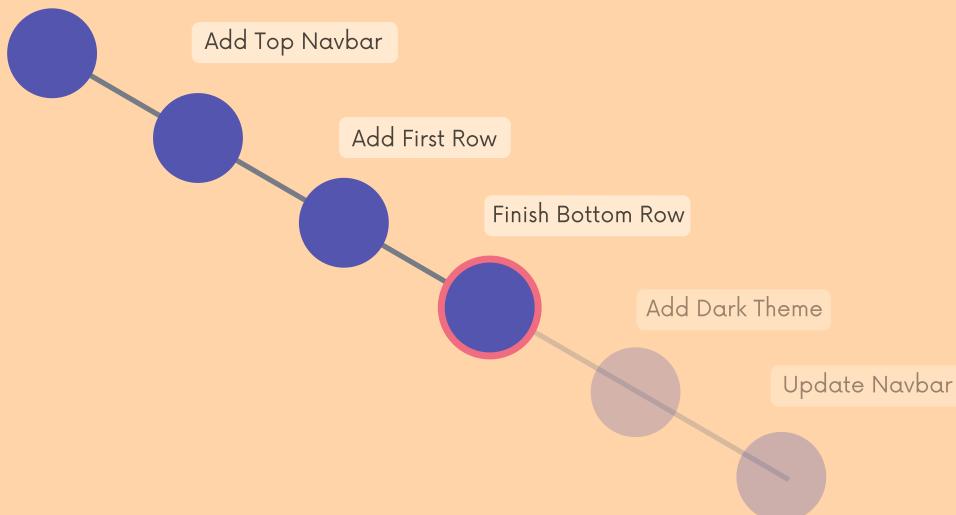
I can go back to prior checkpoints I made!

Initialize Project



I can even start more work off of an old checkpoint

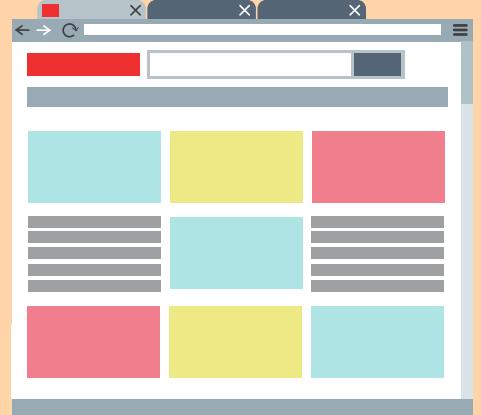
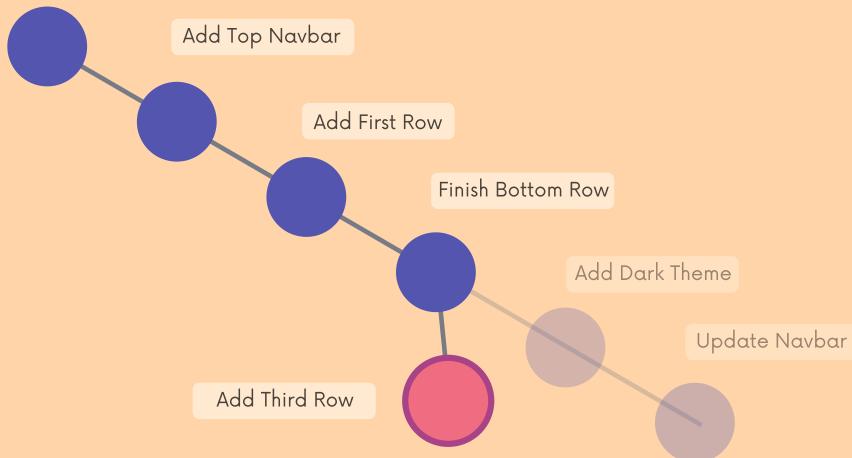
Initialize Project



I add more content!

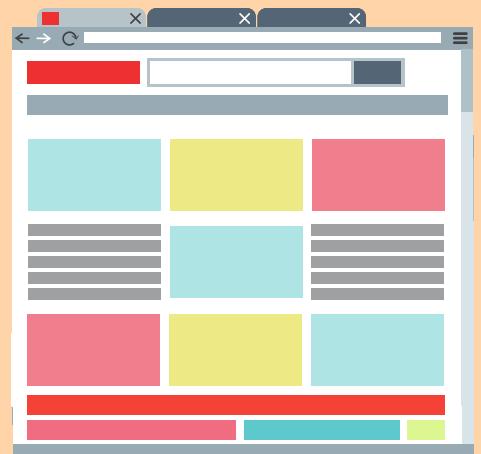
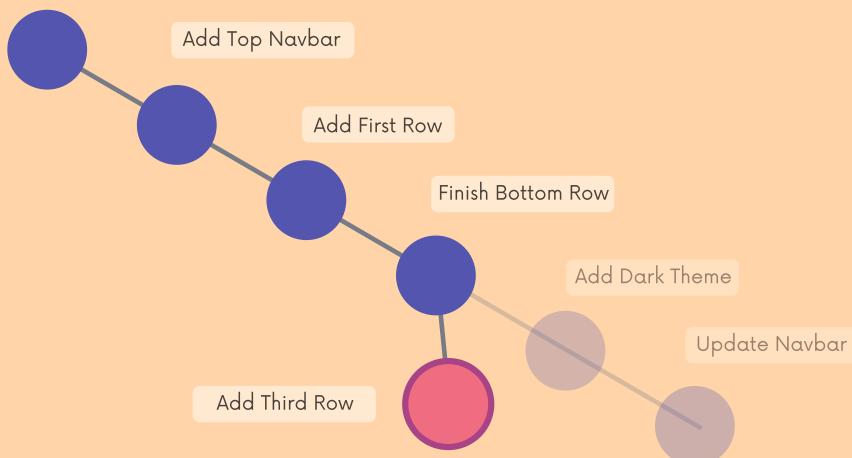
I add a new checkpoint!

Initialize Project

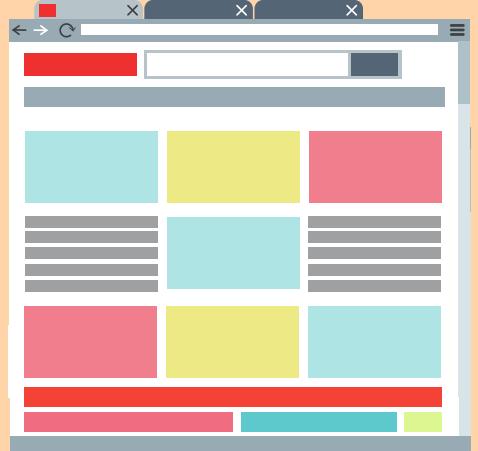
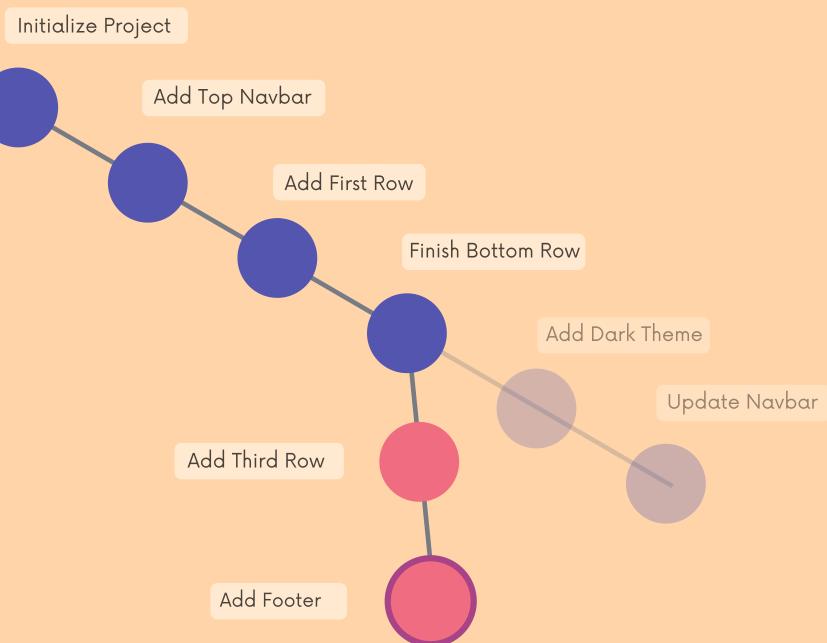


I add a new footer

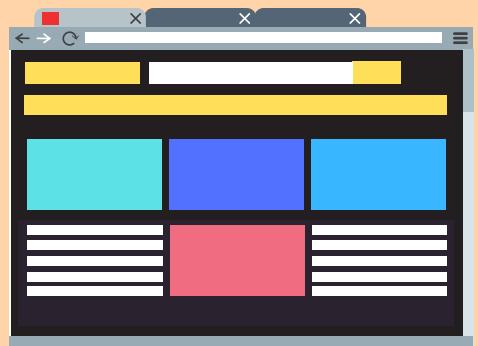
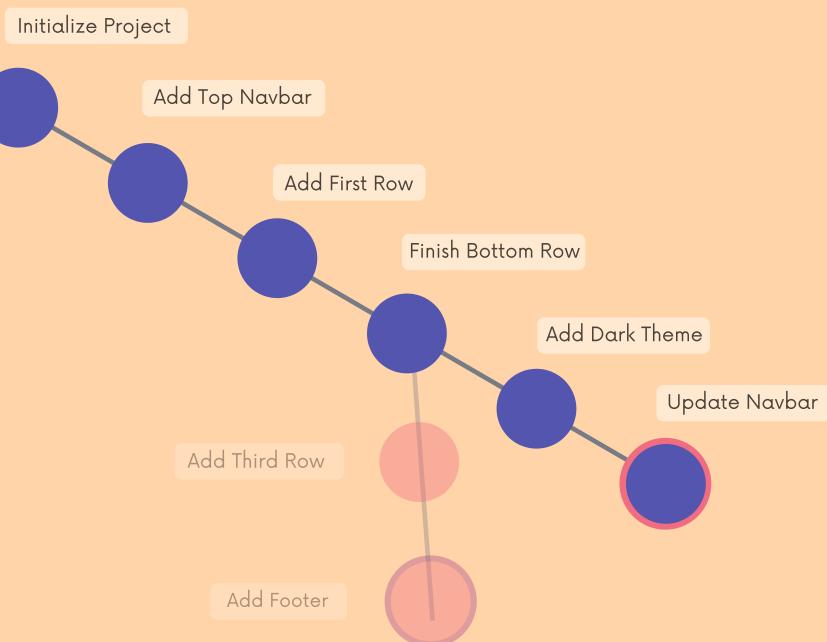
Initialize Project



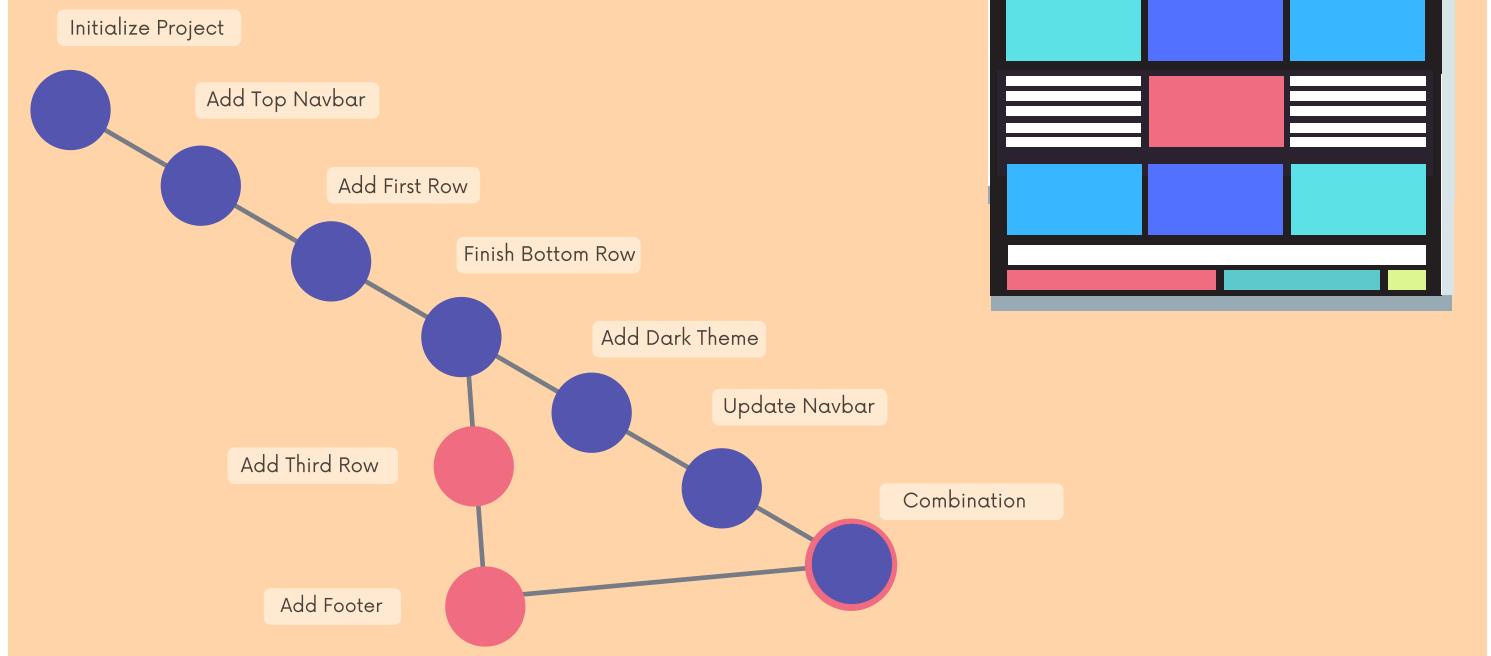
Another checkpoint!



I can switch back to a dark mode checkpoint



And I can even combine checkpoints!



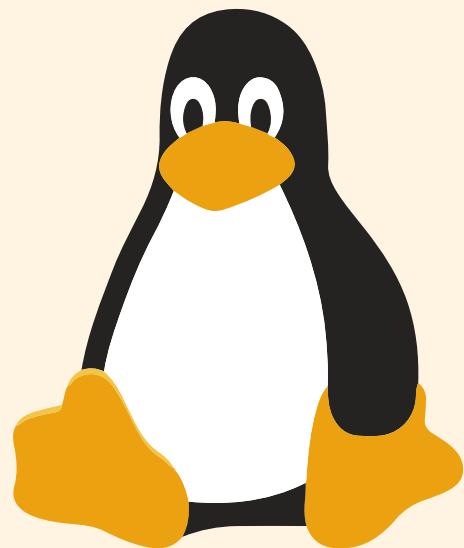
A Tiny Bit of
Git History



Linus Torvalds

Linus Torvalds is a legendary software engineer. He is the creator and main developer behind Linux and Git!

In 2005, while working on Linux, he became frustrated with the available version control systems. The existing tools were slow, closed-source, and usually paid.



The Birth Of Git

Torvalds wanted a version control system that was super fast AND free, unlike the existing tools.

On April 3rd 2005 he got to work on his own VCS, which would become Git. In a matter of days he had most basic functionality done.

The first official Git release came a couple months later. 15 years later in 2020, over 90% of developers worldwide use Git on a daily basis!





Behind The Name

Torvalds referred to Git as "the stupid content tracker" while he was working on it. Eventually he settled on the name Git.

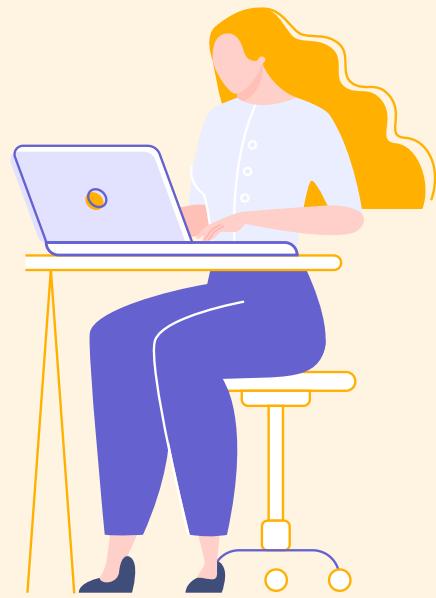
The official Git source code explains a couple different meanings for the name, depending on your mood:

- a random three-letter combination that is pronounceable, and not actually used by any common UNIX command.
- stupid. contemptible and despicable. simple.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "g#ddamn idiotic truckload of sh*t": when it breaks



Engineers & Coders

From massive tech giants like Facebook and Google to the tiniest of startups, developers across the globe use Git. If you plan on becoming a developer, Git is essentially a must-have.





Tech-Adjacent Roles

Many people in non-developer roles end up learning the basics of Git to collaborate with their coworkers. Designers in particular often need to work with Git.



Governments

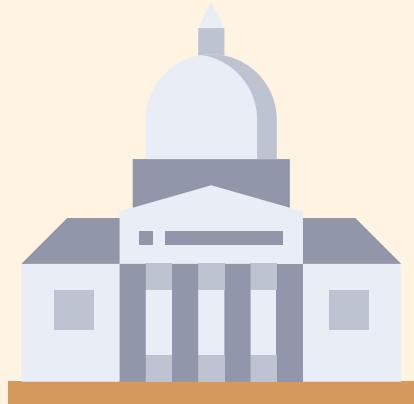
In recent years, governments have started using Git to manage the drafting of laws. Washington DC City council uses Git & Github to publish its laws. Citizens have even found and fixed typos collaboratively using Git!





Scientists

Git is commonly used by research teams at universities and agencies around the world to manage code, data sets, and more. Git + Github is especially powerful in the peer-reviewed world of science and research.



Writers

Some writers love using Git to manage drafts of complex novels, screenplays, or other works with lots of moving parts and constant changes across multiple files.

In particular, Git is gaining popularity for use in collaborative textbook writing with multiple authors.



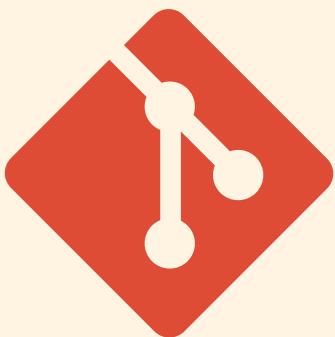
Anyone Really

People find very creative uses for Git ranging from keeping a daily diary to drafting PhD theses to tracking changes to photoshop files. At least one composer writes his symphonies using Git!



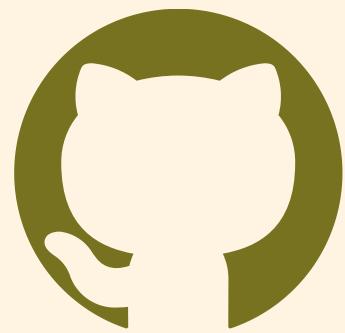
Git ≠ Github





Git

Git is the version control software that runs locally on your machine. You don't need to register for an account. You don't need the internet to use it. You can use Git without ever touching Github.



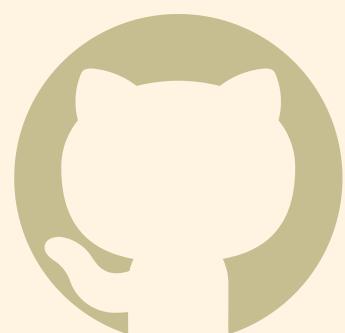
Github

Github is a service that hosts Git repositories in the cloud and makes it easier to collaborate with other people. You do need to sign up for an account to use Github. It's an online place to share work that is done using Git.



Git

Git is the version control software that runs locally on your machine. You don't need to register for an account. You don't need the internet to use it. You can use Git without ever touching Github.



Github

Github is a service that hosts Git repositories in the cloud and makes it easier to collaborate with other people. You do need to sign up for an account to use Github. It's an online place to share work that is done using Git.

Git

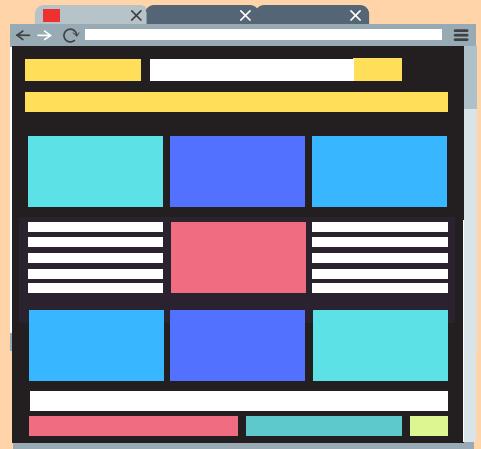
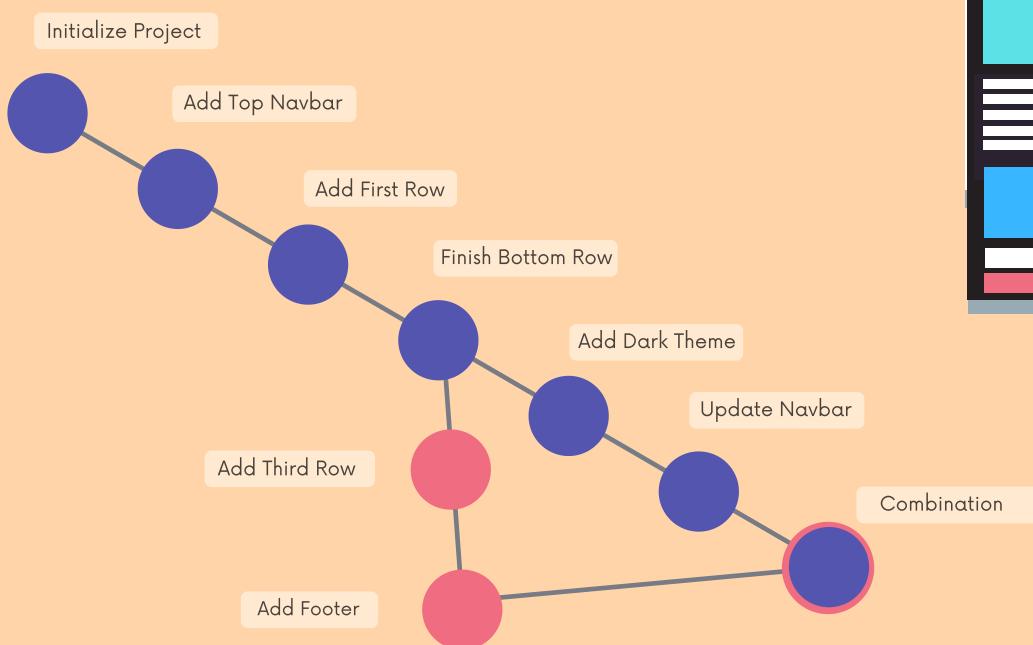
Git is the version control software that runs locally on your machine. You don't need to register for an account. You don't need the internet to use it. You can use Git without ever touching Github.



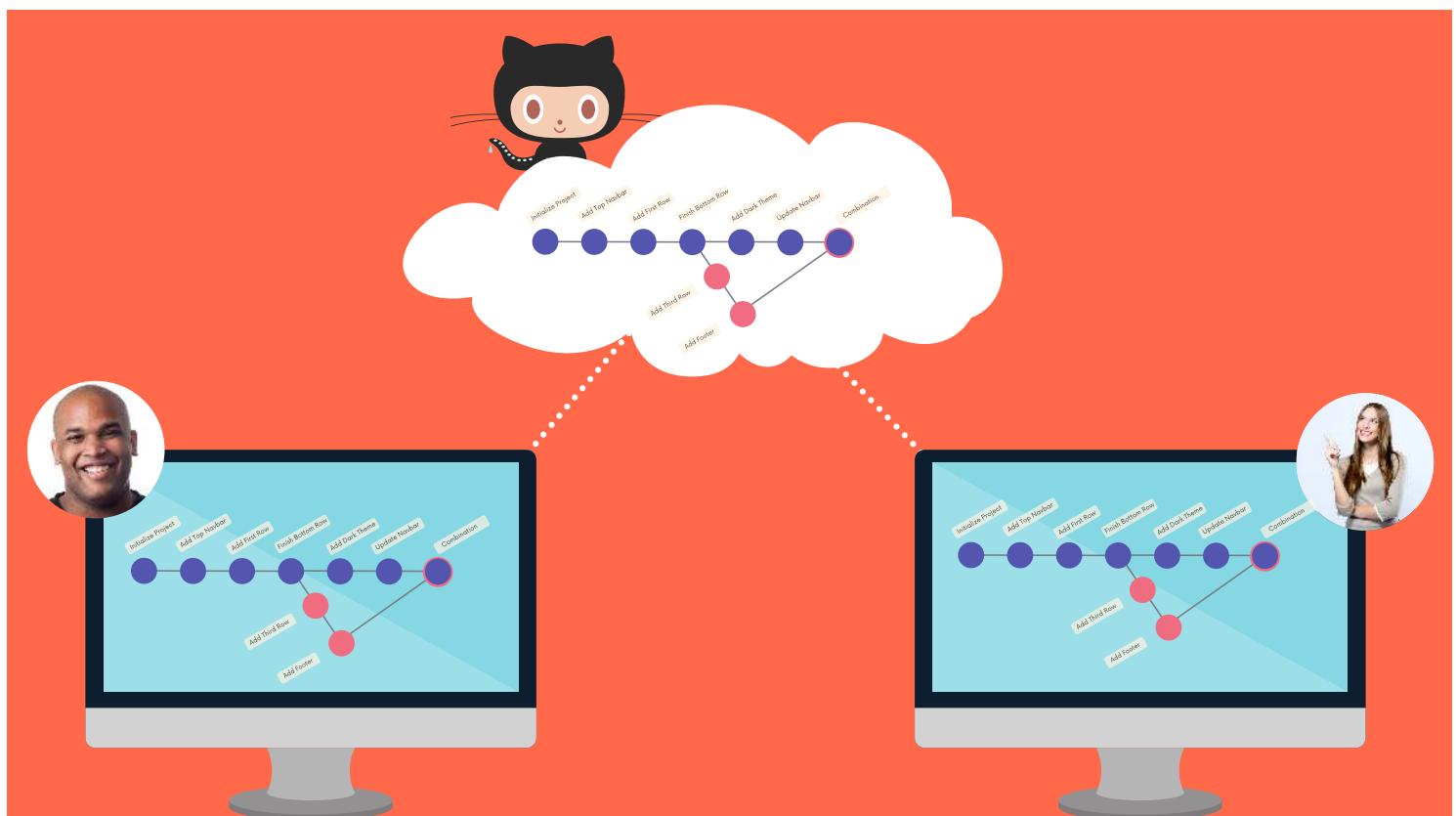
Github

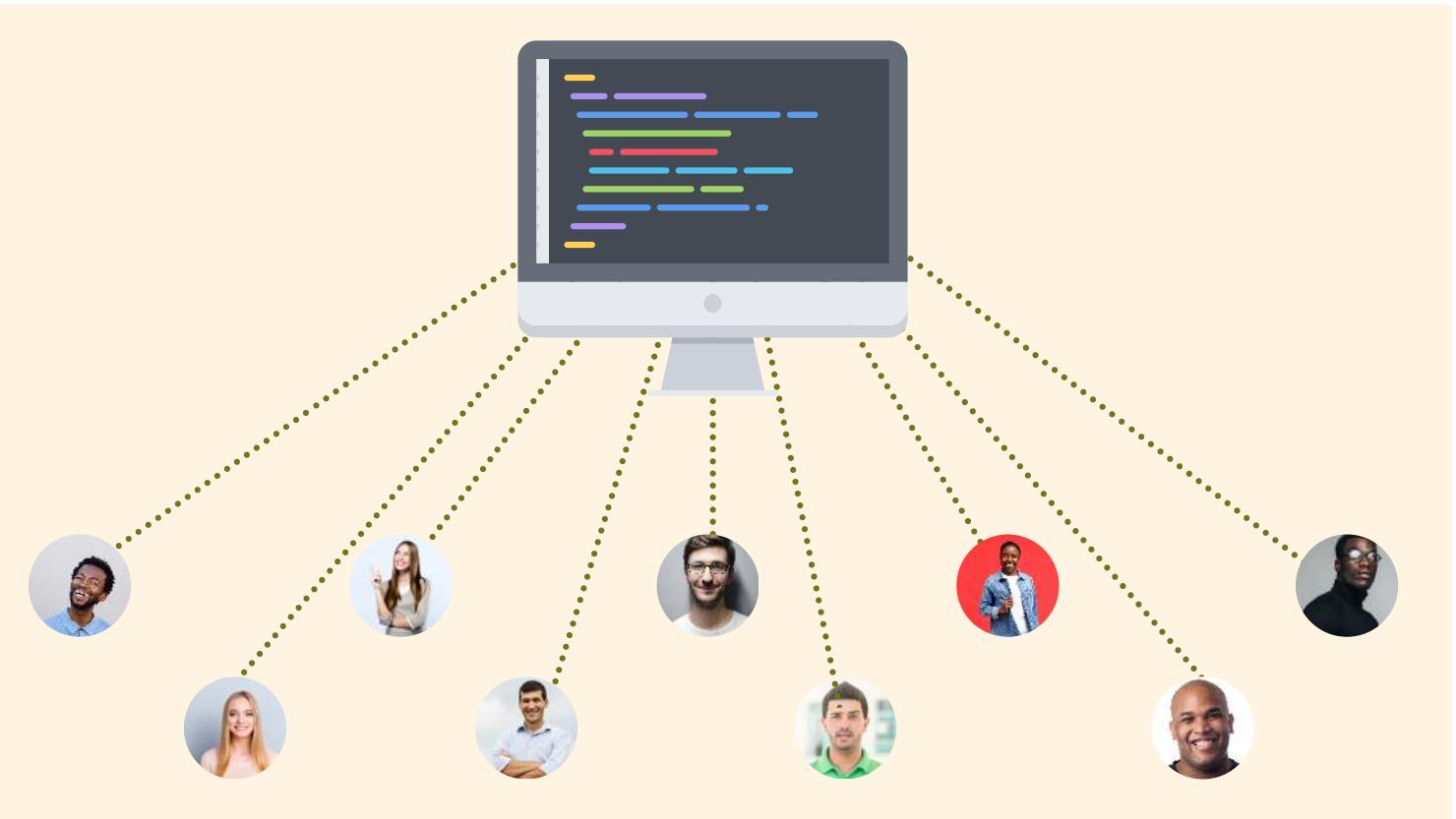
Github is a service that hosts Git repositories in the cloud and makes it easier to collaborate with other people. You do need to sign up for an account to use Github. It's an online place to share work that is done using Git.

This is all on my local machine



I can upload my history to Github to share with others





2

Installation



Where Do We Start?



Git Is (Primarily) A Terminal Tool

Git was created as command-line tool. To use it, we run various git commands in a Unix shell. This is not the most user friendly experience, but it's at the very core of Git!





The Rise of GUI's

Over the last few years, companies have created graphical user interfaces for Git that allow people to use Git without having to be a command-line expert.

Popular Git GUI's include:

- Github Desktop
- SourceTree
- Tower
- GitKraken
- Ungit



GUI Clients



Pros

- Way lower barrier-of-entry for beginners compared to the command-line.
- Friendlier to use. Can be a much better experience (when it works)
- Some people prefer the visual experience, even those who can use the command-line

Cons

- At times, there is lots of "magic" involved. The inner-workings of Git are obfuscated and hidden away with GUI's.
- Often leads to dependence on a particular piece of software.
- When things go seriously wrong, it can be very challenging to fix without the command-line
- The interfaces, buttons, and menus vary between different GUI's.



≡

The Command Line



Pros

- Git is a command-line tool. All the documentation and resources online will refer to the command-line
- The command-line can be way faster once you get comfortable with it!
- Some of the more advanced Git features are only available on the command-line
- The commands are always the same, no matter what machine you are on!

Cons

- Not beginner-friendly. At All. Can be difficult to learn and remember the commands at first.
- Even for some practiced users, the command-line interface is just not a good experience. It's really a matter of preference.



There is a lot of stupid
GUI gate-keeping



Are you a developer? Do you plan on becoming a developer?

Learn the command-line!
you'll need to use the command-line any way

Learning Git for other purposes?

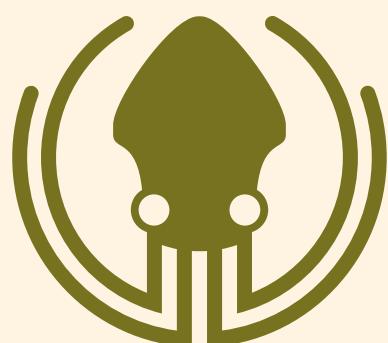
Try a GUI!
(or don't!)

≡

Installing a GUI

There are many options to choose from depending on your operating system, and most of them are incredibly simple to install!

I will be using **GitKraken** throughout the course, and I recommend you install it too. It's free, though there is a paid tier that we don't need.



Installing Git

Installing Git locally is slightly trickier, depending on your specific operating system.

Git is intended to run on Unix-style systems like Linux and MacOS, so if you're on a Windows machine you may need to take a couple extra steps.



Mac Install

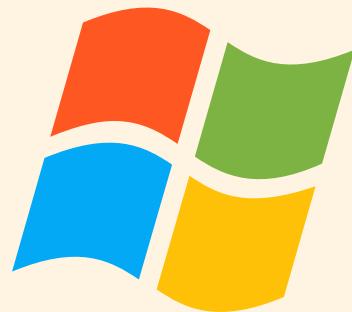
- First, check to see if you have Git installed already using the command `git --version`
- If not, or if you have an old version, download the latest Git installer package using the link in the description
- Verify your install worked by running `git --version` again afterwards

```
git --version
```

Windows Install

Git Bash emulates the unix-based Git command-line experience for Windows machines, and it's super easy to install!

- Download [Git For Windows](#)
- Find the downloaded .exe file and open it to execute Git Bash.

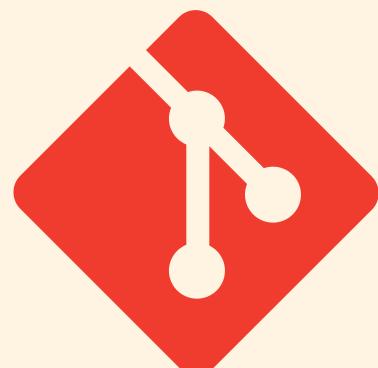


Configuring Git

Now that Git is hopefully installed, it's time to configure some basic information. You do not need to register for an account or anything, but you will need to provide:

- Your name
- Your email

If you are using a GUI, it should prompt you for your name and email the first time you open the app.



Configuring Git

To configure the name that Git will associate with your work, run this command:

```
git config --global user.name "Tom Hulce"
```

Configuring Git

Do the same thing for your email using the following command. When we get to Github, you'll want your Git email address to match your Github account.

```
git config --global user.email blah@blah.com
```

Let's Get Started!



3 git basics



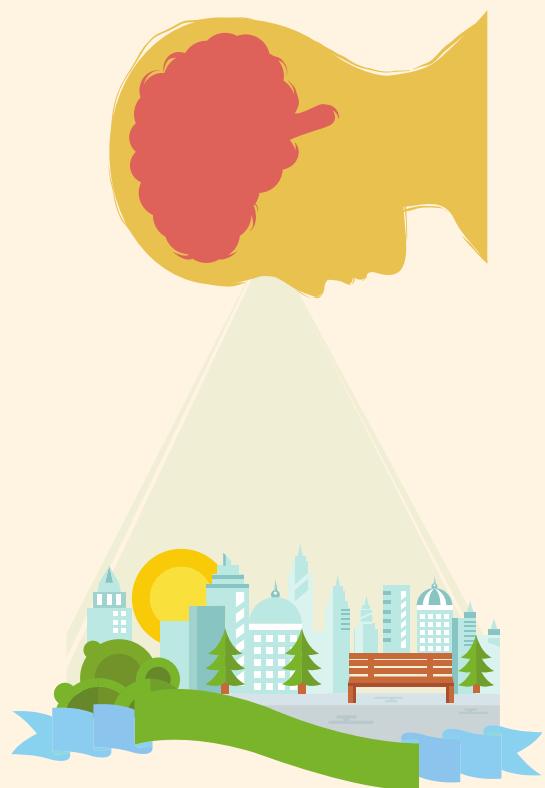
It's Time To Learn

Git Basics



A Quick High-Level

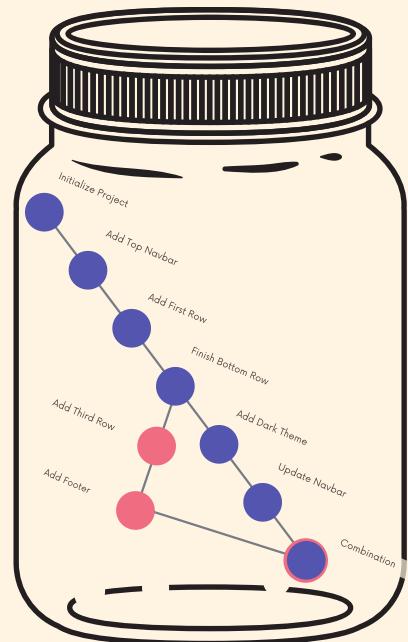
Conceptual Overview



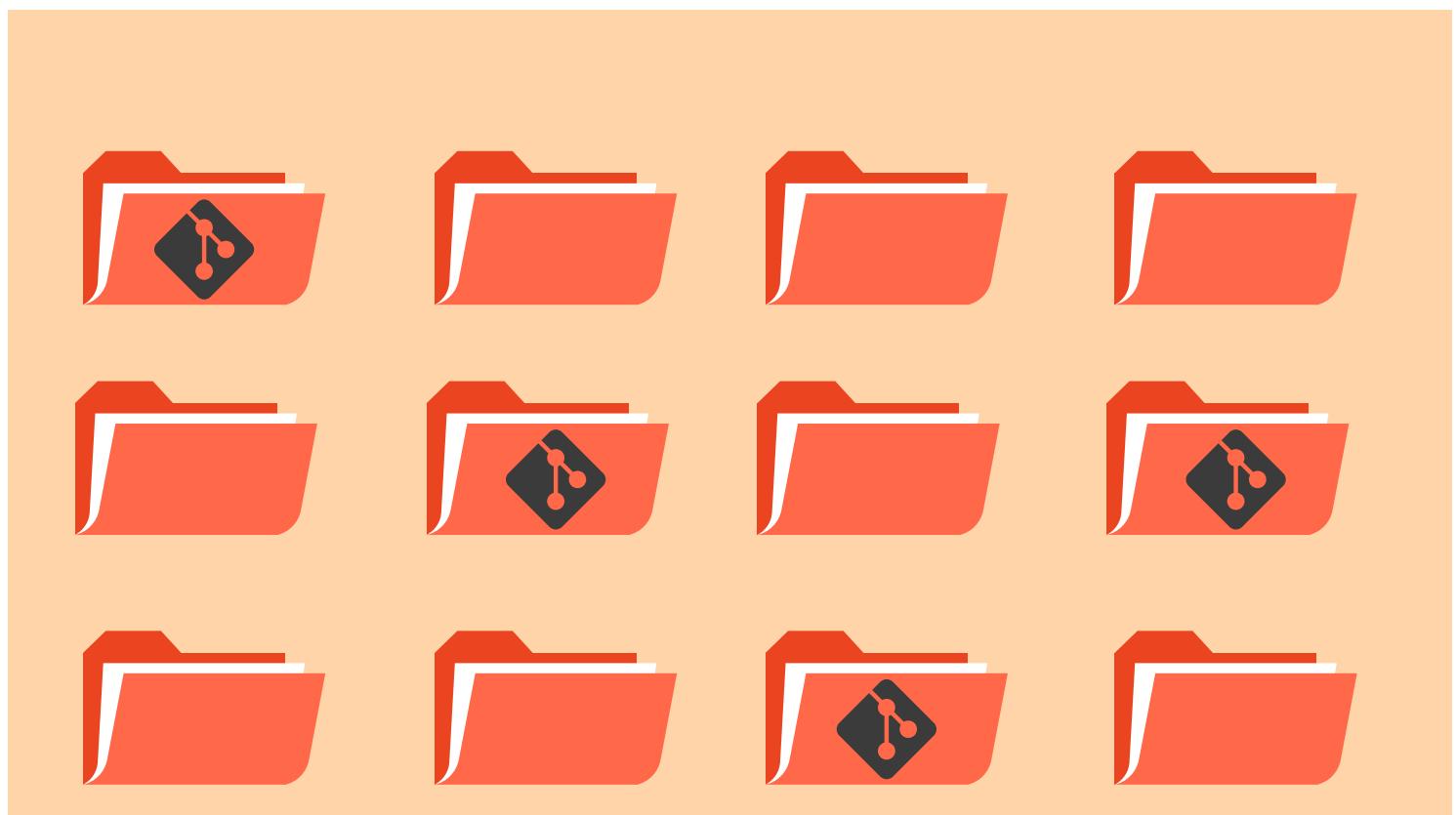
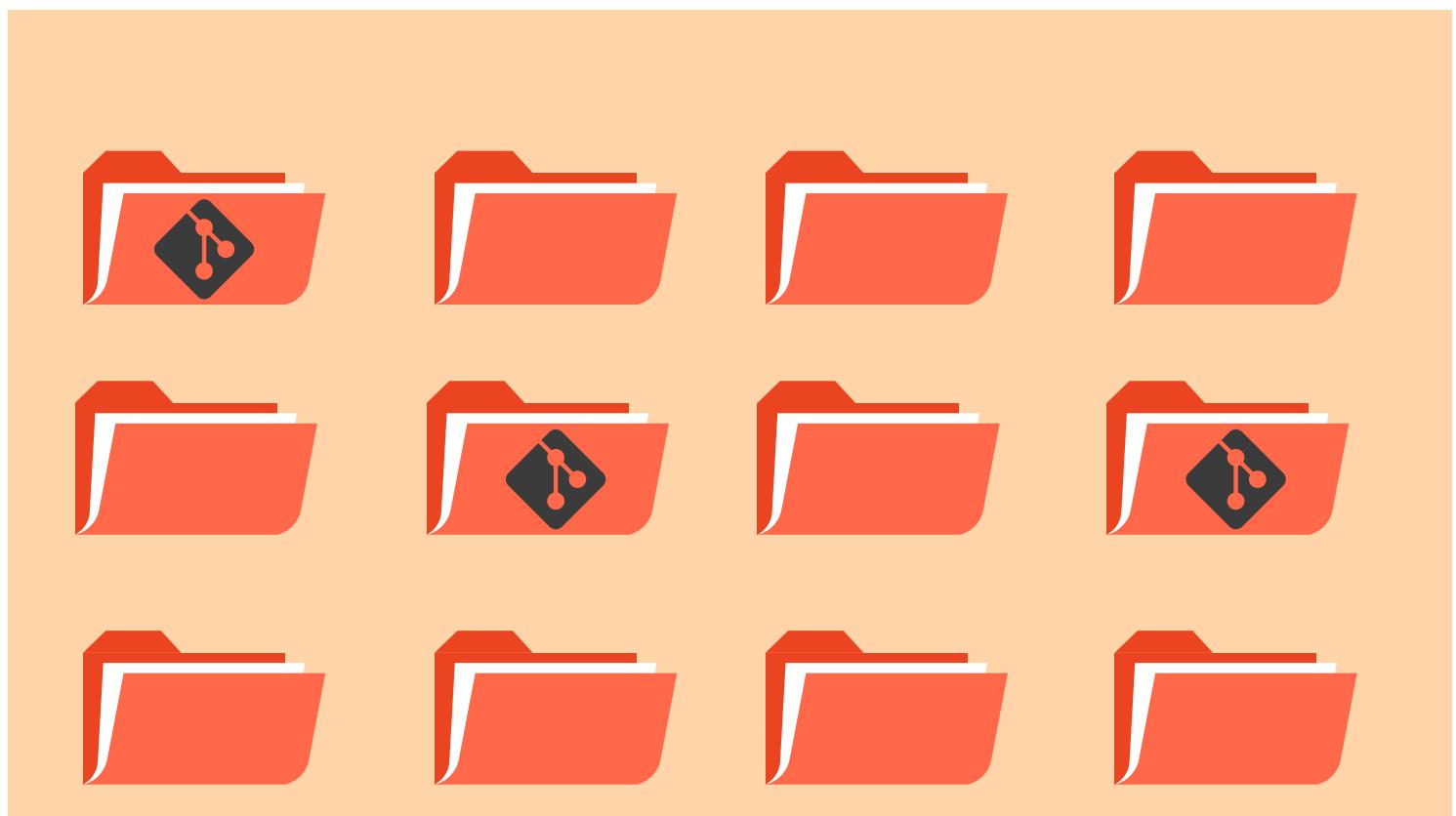
Repository

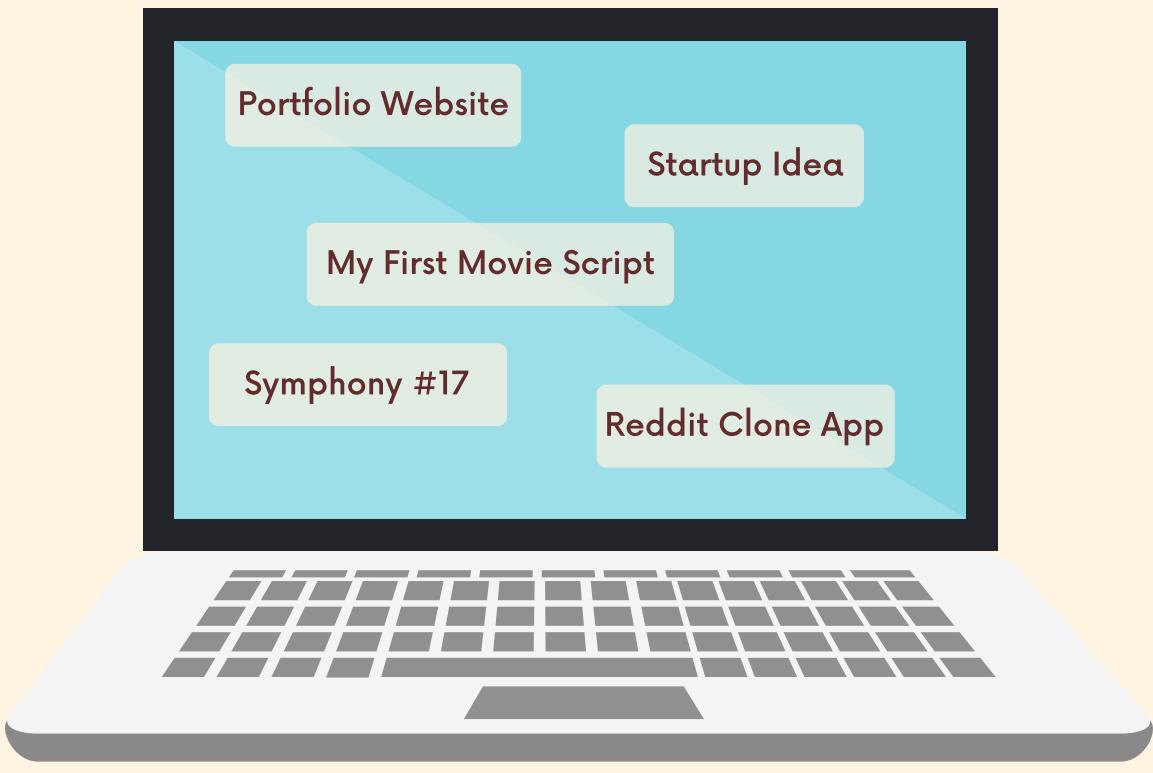
A Git "Repo" is a workspace which tracks and manages files within a folder.

Anytime we want to use Git with a project, app, etc we need to create a new git repository. We can have as many repos on our machine as needed, all with separate histories and contents









≡

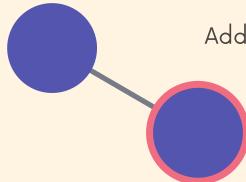
Committing

The most important Git feature!

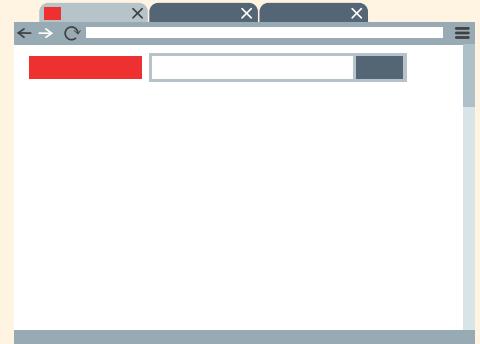


This is my repo!

Initialize Project

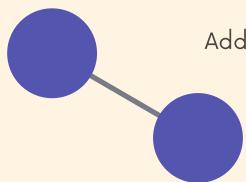


Add Top Navbar

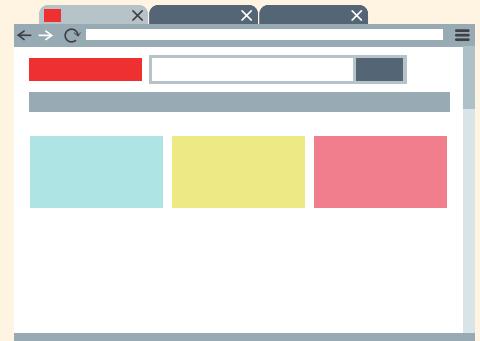


I add some content

Initialize Project



Add Top Navbar

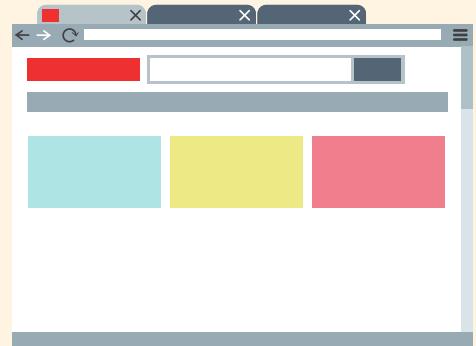


Add A Commit

Initialize Project

Add Top Navbar

Add First Row



Committing

Making a commit is similar to making a save in a video game. We're taking a snapshot of a git repository in time.

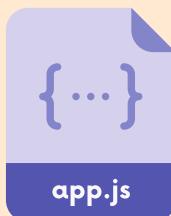
When saving a file, we are saving the state of a single file. With Git, we can save the state of multiple files and folders together.



Save



Commit



☰

The Basic Git Workflow

Work On Stuff

Make new files, edit files, delete files, etc

Add Changes

Group specific changes together, in preparation of committing

Commit

Commit everything that was previously added





Our First Git Command!

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

It's very useful, but at the moment we don't actually have any repos to check the status of!



```
git status
```



If We Try It...



```
git status
```

fatal: not a git repository
(or any of the parent directories)

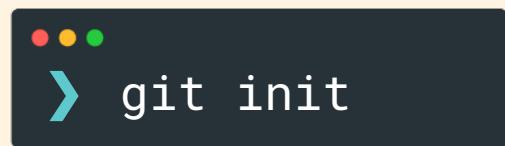




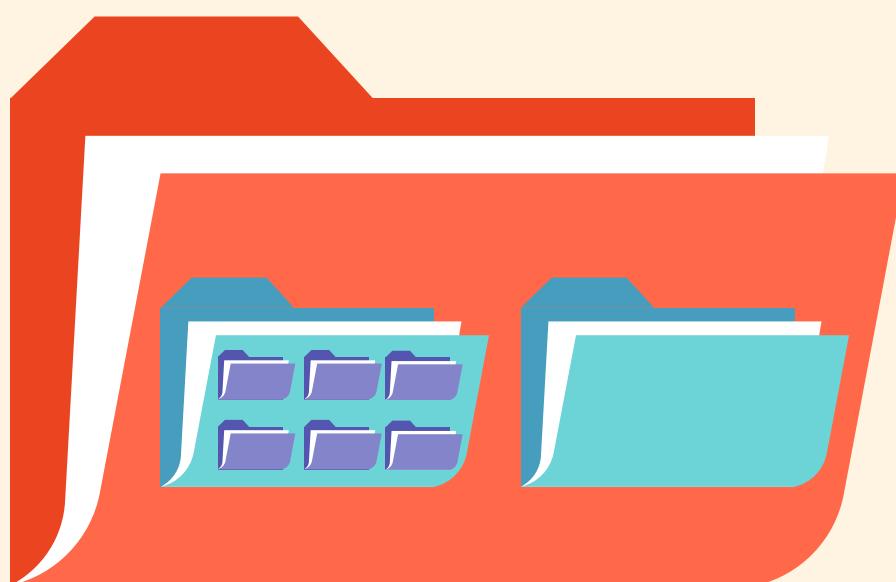
Our Actual First Git Command!

Use `git init` to create a new git repository. Before we can do anything git-related, we must initialize a repo first!

This is something you do once per project. Initialize the repo in the top-level folder containing your project



Git Tracks A Directory and All Nested Subdirectories



WARNING

DO NOT INIT A REPO INSIDE OF A REPO!

Before running `git init`, use `git status` to verify that you are not currently inside of a repo.



If you do end up making a repo inside of a repo, you can delete it and try again!

To delete a repo, locate the associated `.git` directory and delete it.



NOW WHAT?

≡

Git Is Now Watching Us!

Try making a new file in your repo, and then run `git status` again.

You'll see that git noticed the file, but it is "untracked"





≡

Committing

The most important Git feature!





Committing

Making a commit is similar to making a save in a video game. We're taking a snapshot of a git repository in time.

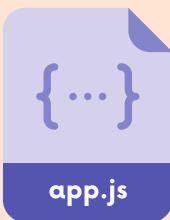
When saving a file, we are saving the state of a single file. With Git, we can save the state of multiple files and folders together.



Save



Commit



deleted team.html

modified about.html

modified about.css

created navbar.html

created navbar.css

created navbar.js

created logo.jpg

I made changes in 7
different files this morning

deleted team.html

modified about.html

modified about.css

Add new team members
to about page

created navbar.html

created navbar.css

created navbar.js

created logo.jpg

deleted team.html

modified about.html

modified about.css

created navbar.html

created navbar.css

created navbar.js

created logo.jpg

Add branded navbar



The Basic Git Workflow

Work On Stuff

Make new files, edit files, delete files, etc

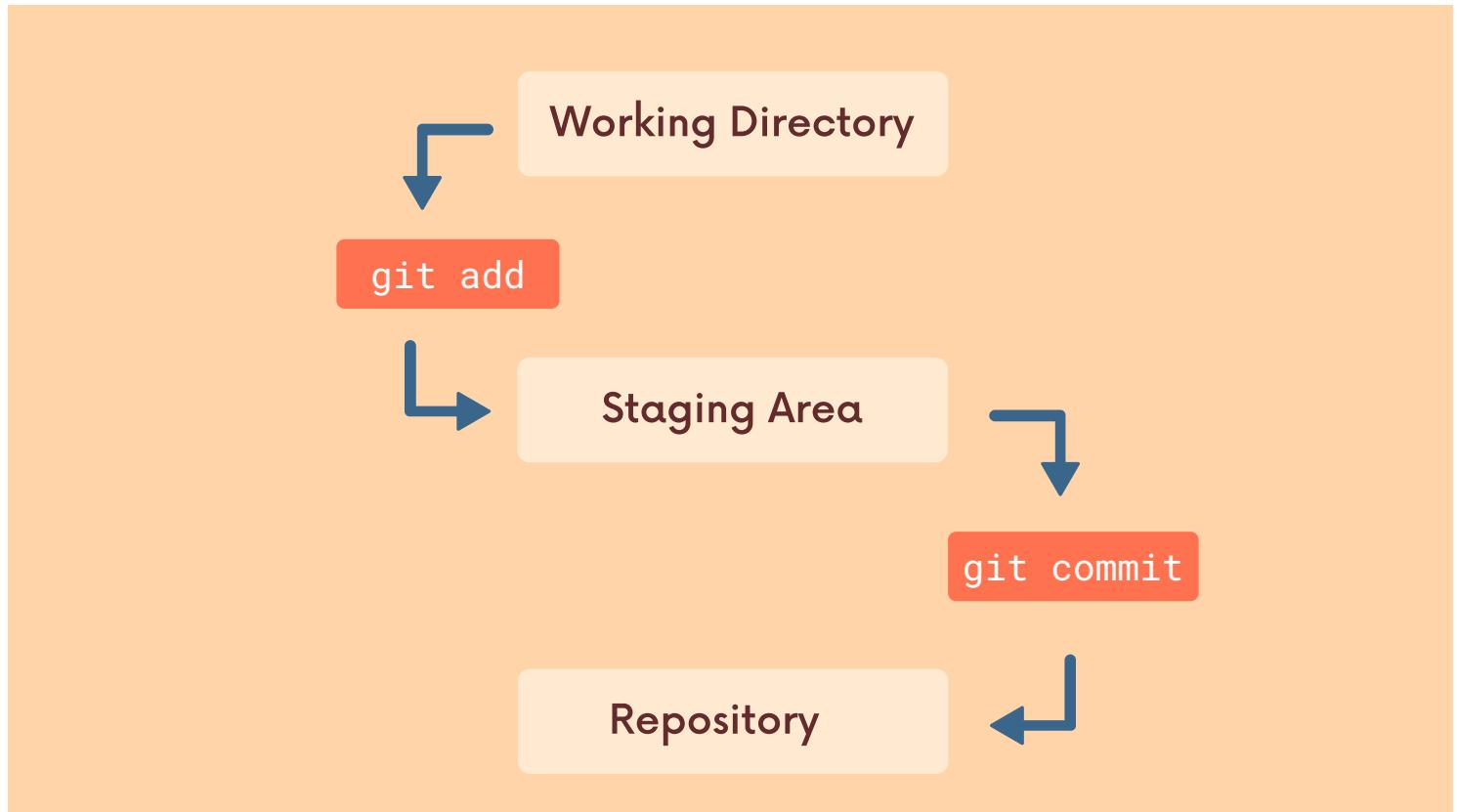
Add Changes

Group specific changes together, in preparation of committing

Commit

Commit everything that was previously added





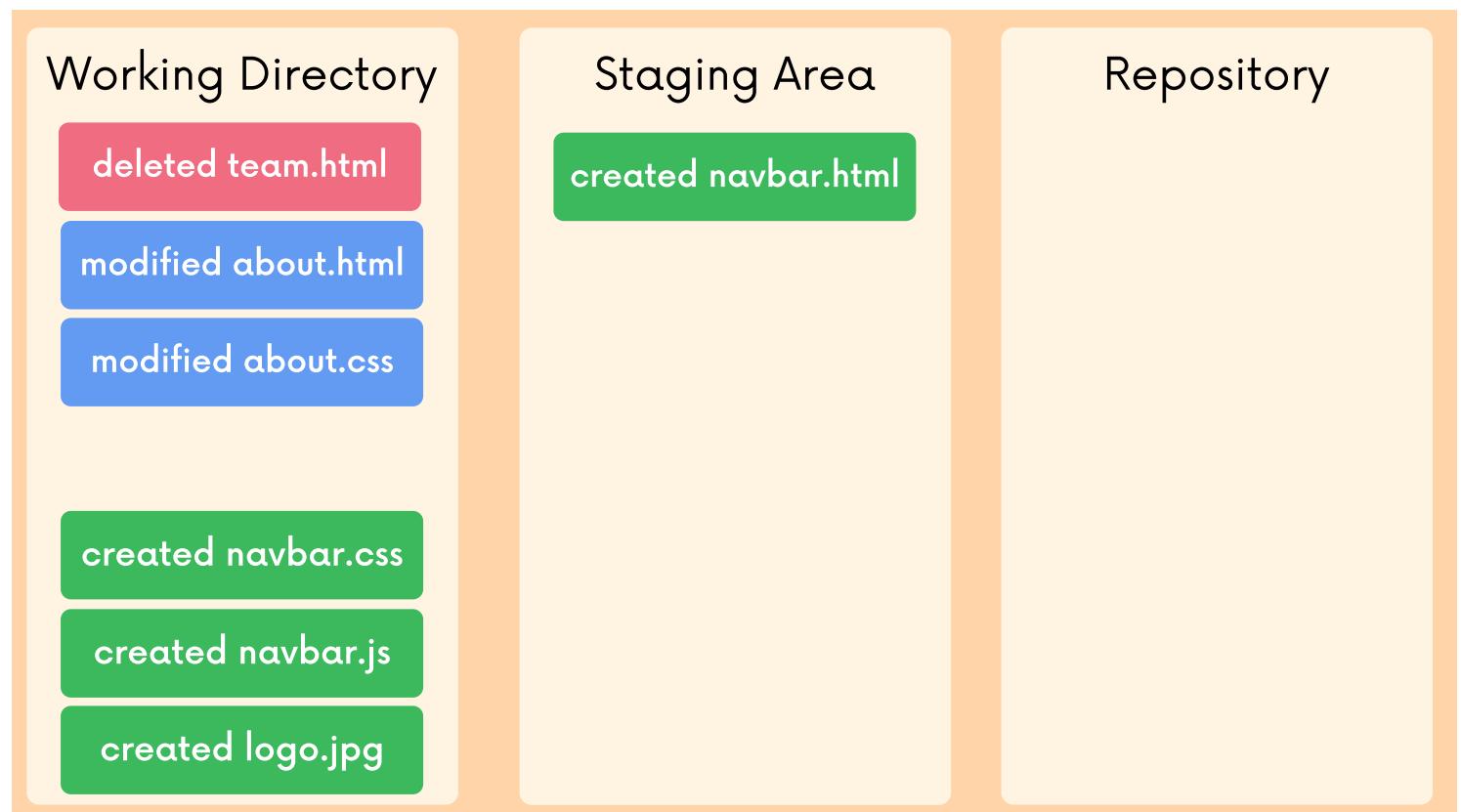
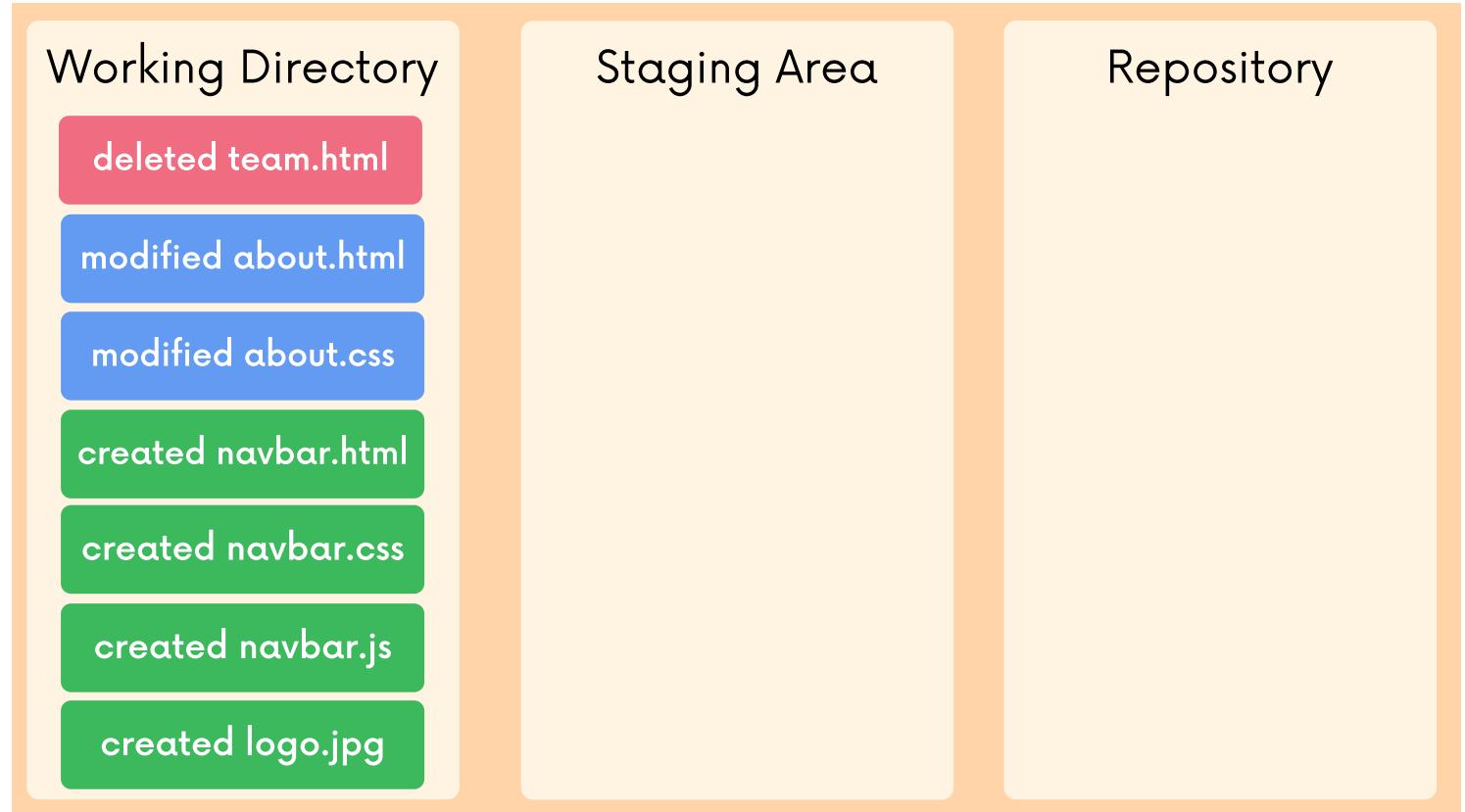
≡

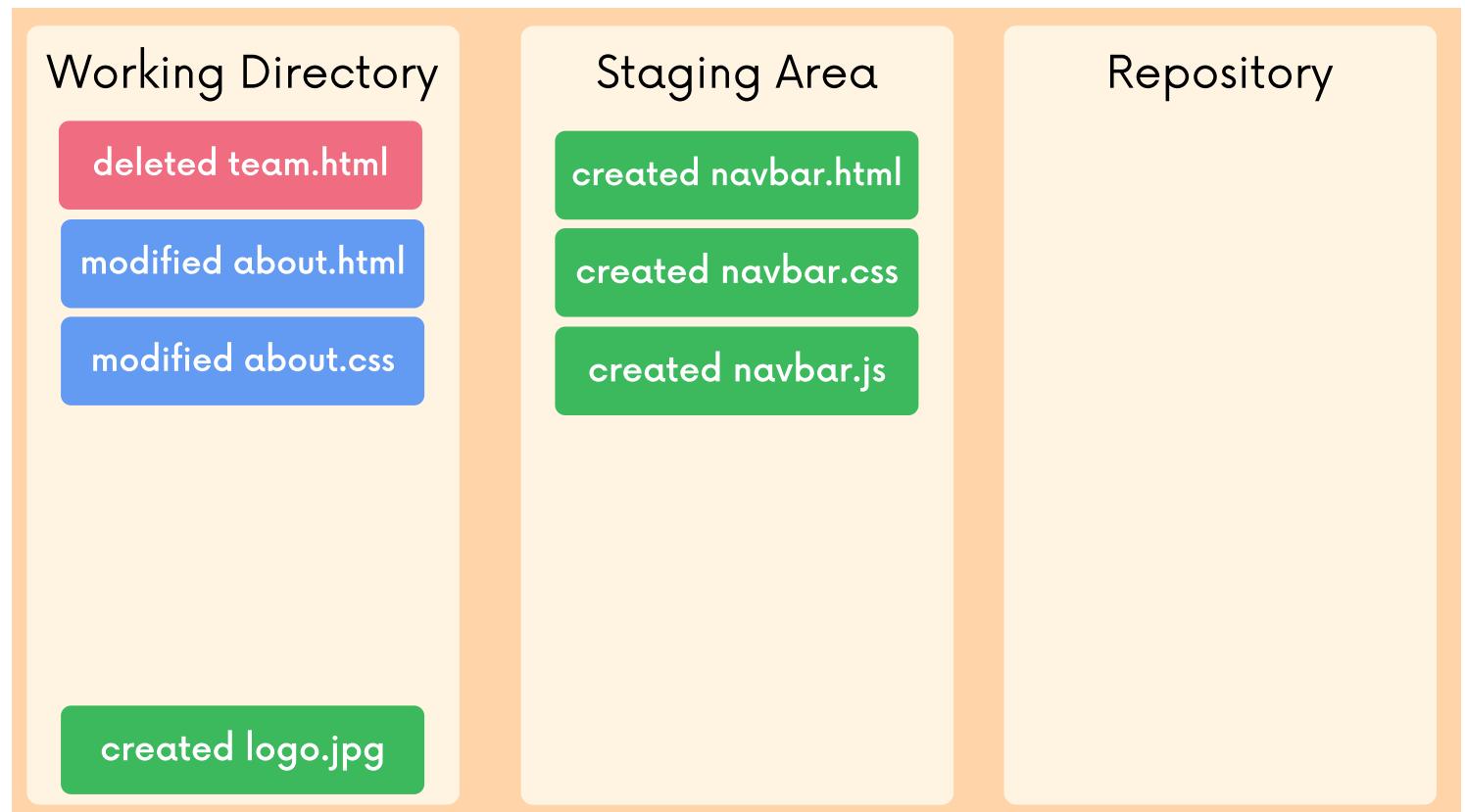
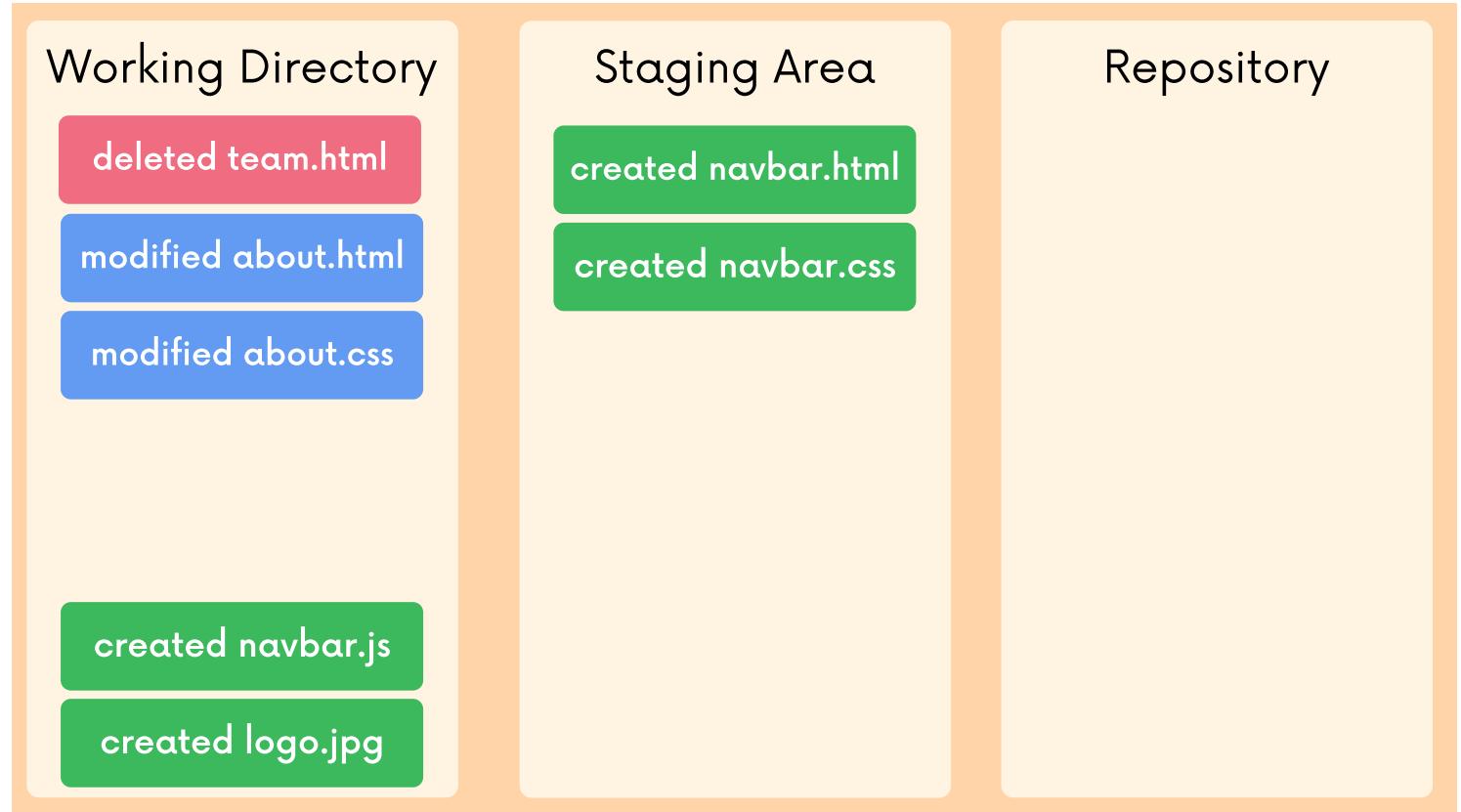
Adding

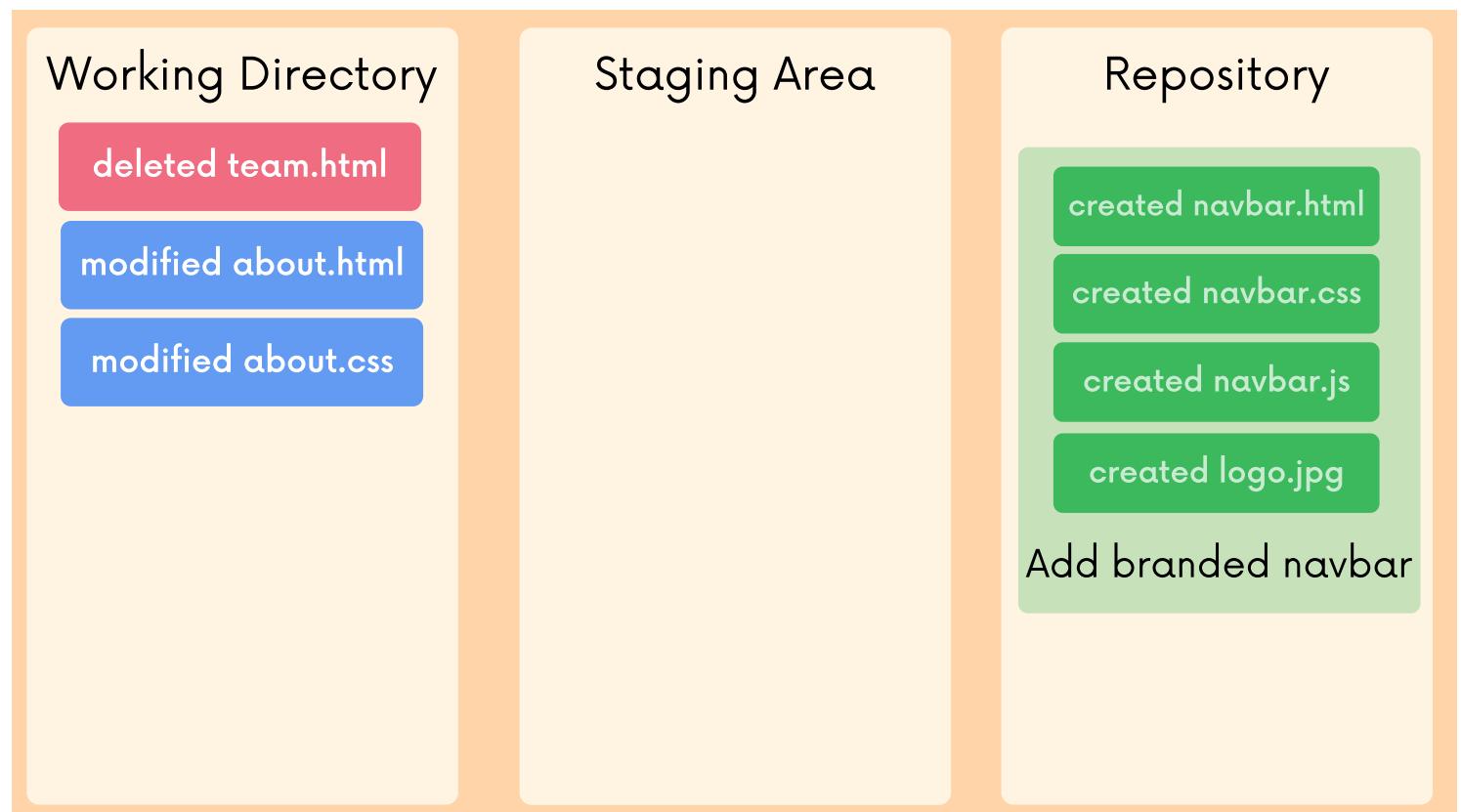
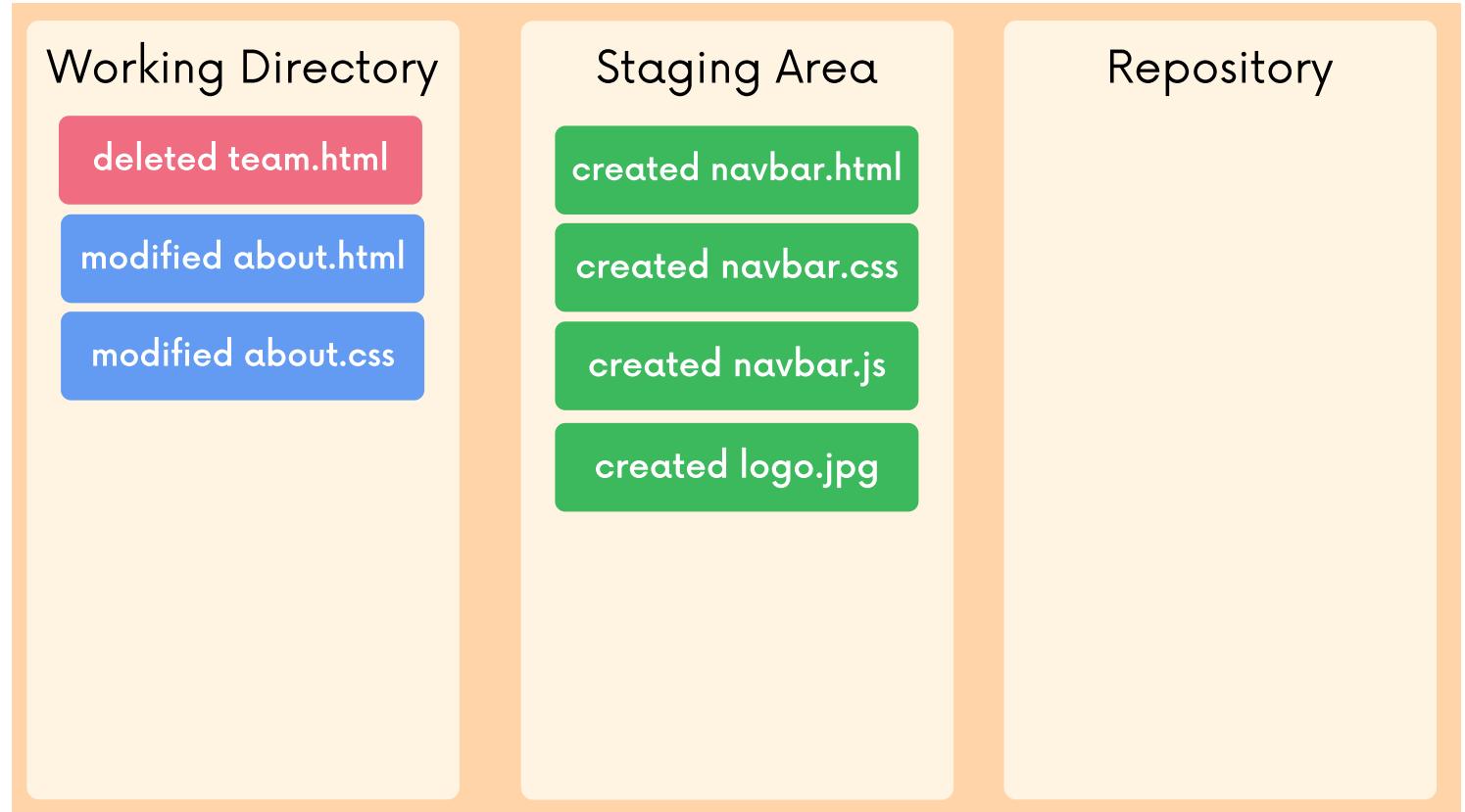
We use the `git add` command to stage changes to be committed.

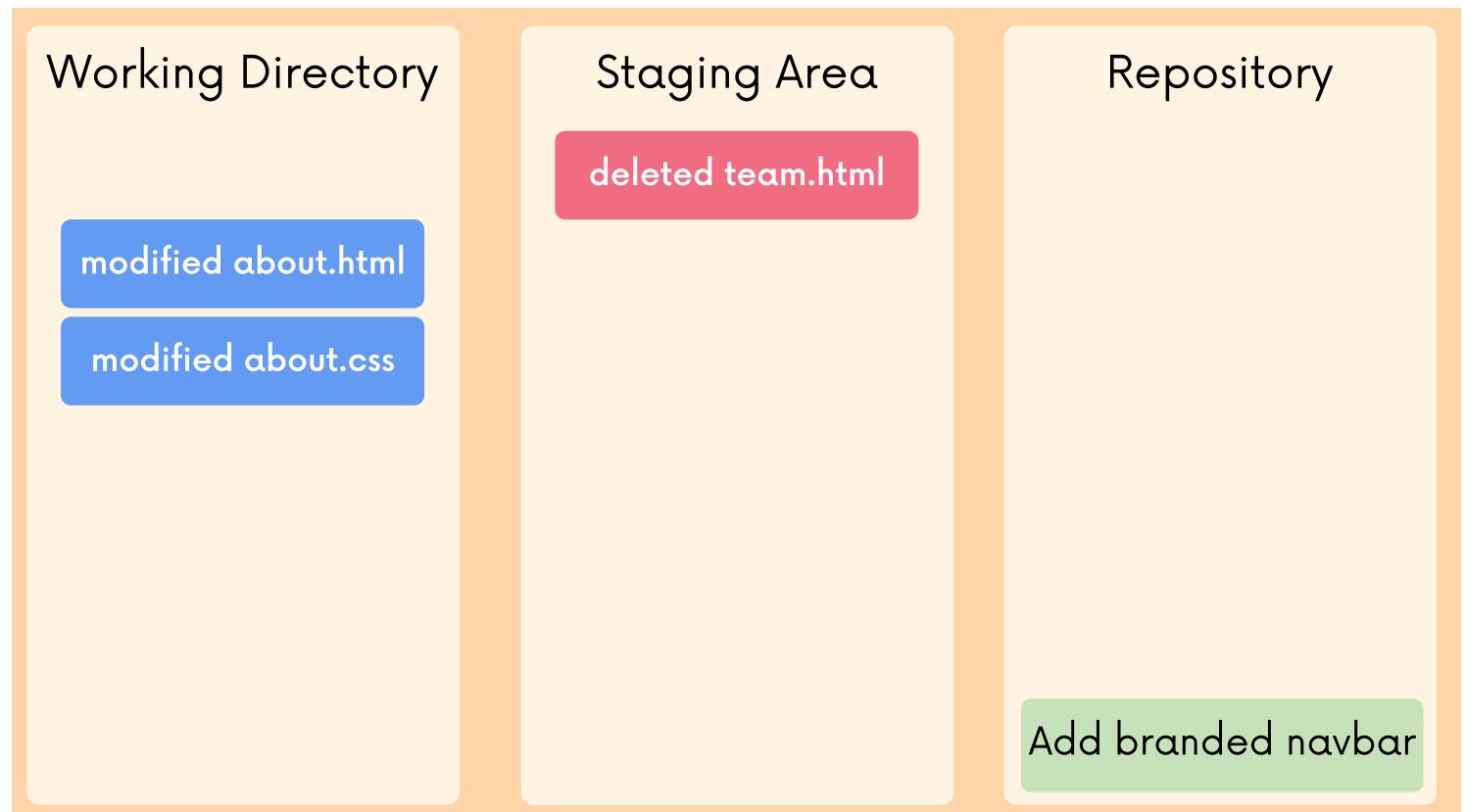
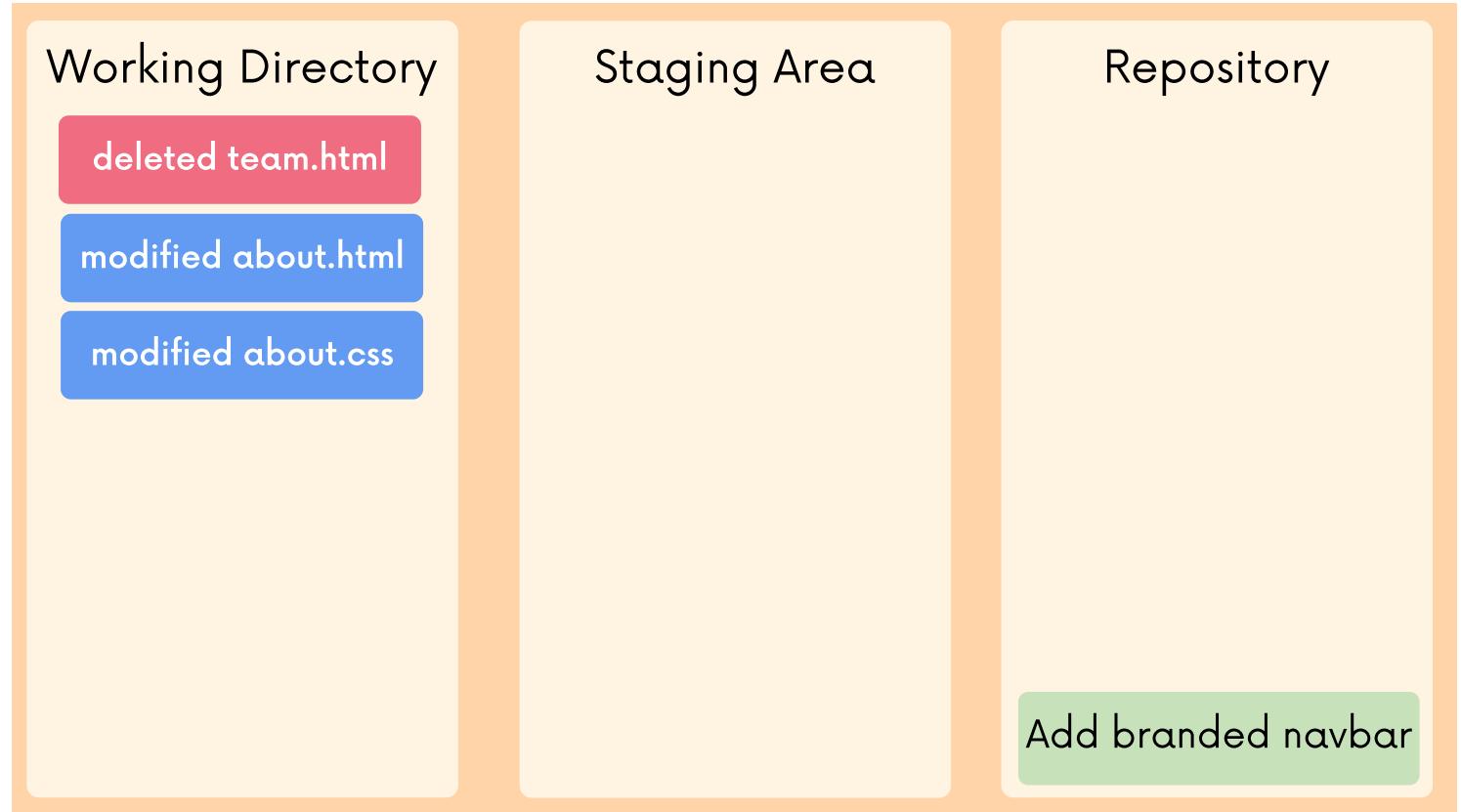
It's a way of telling Git, "please include this change in our next commit"

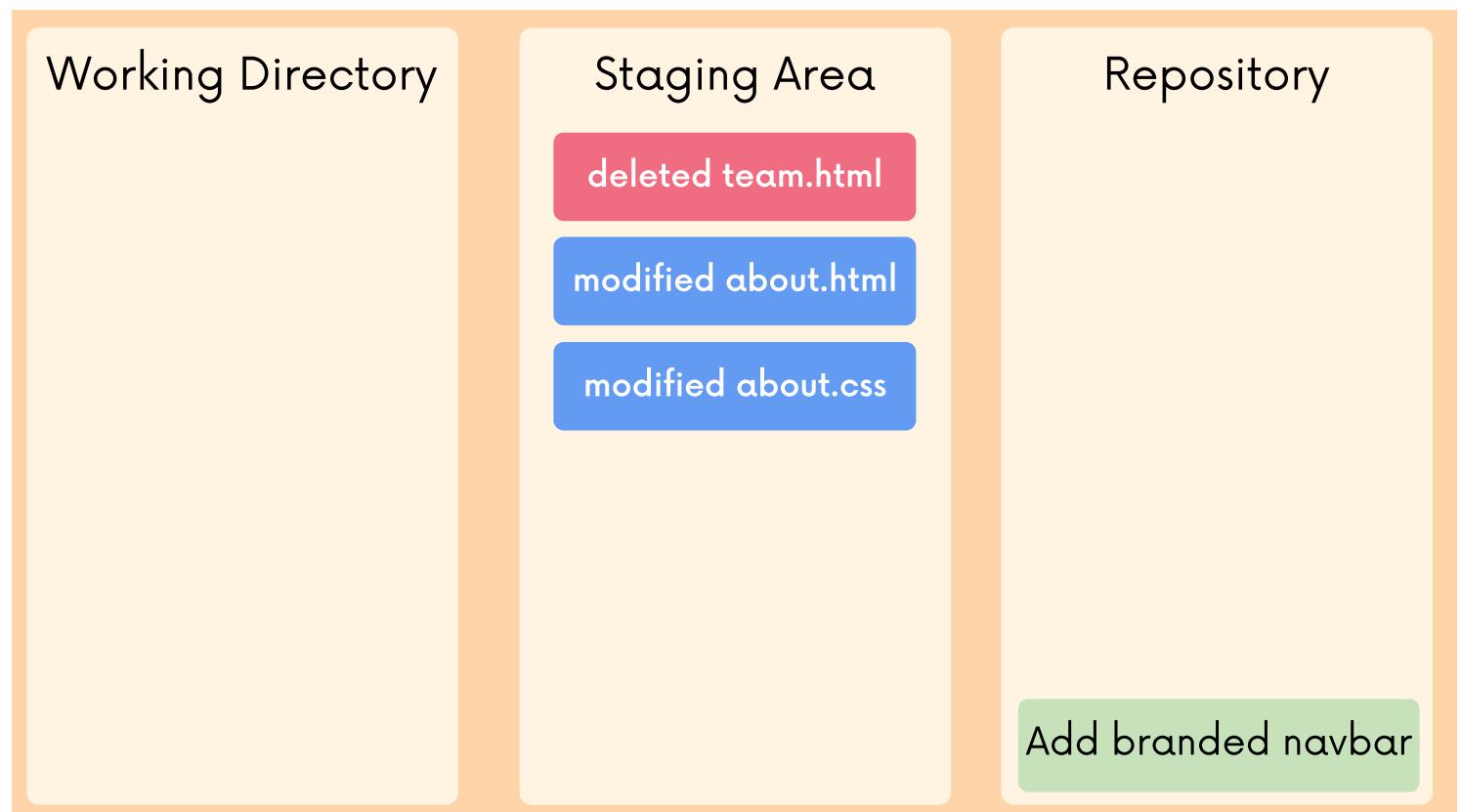
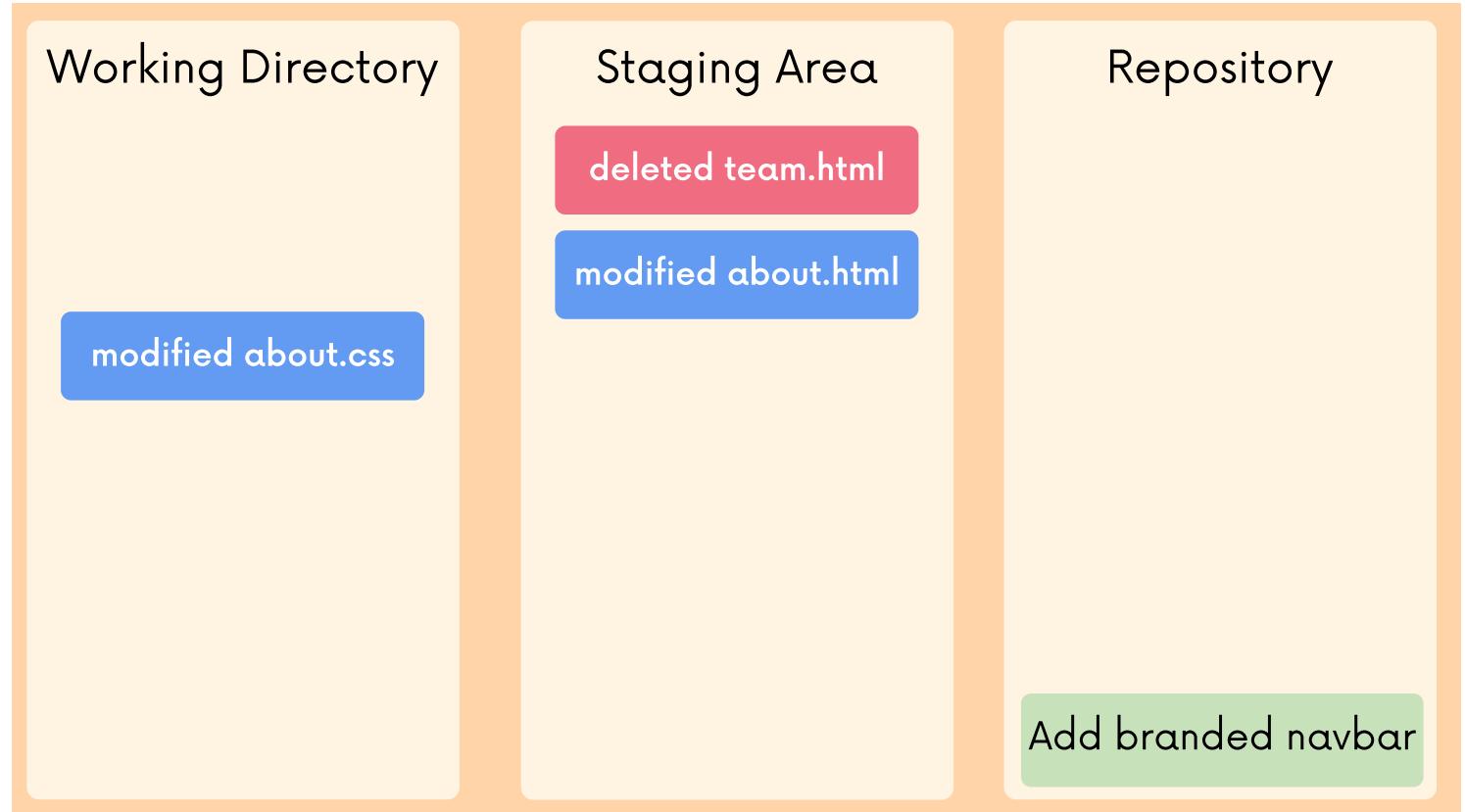


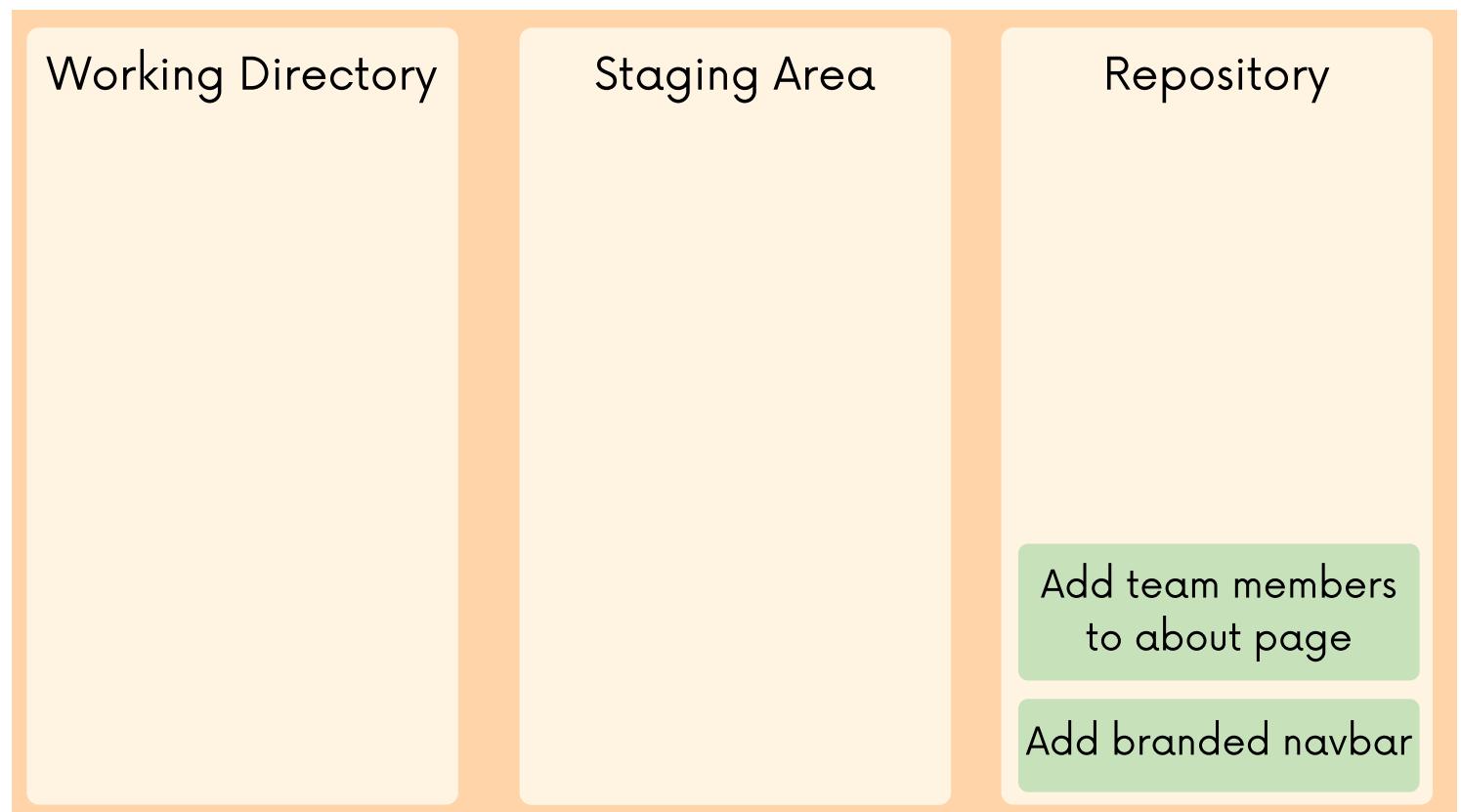
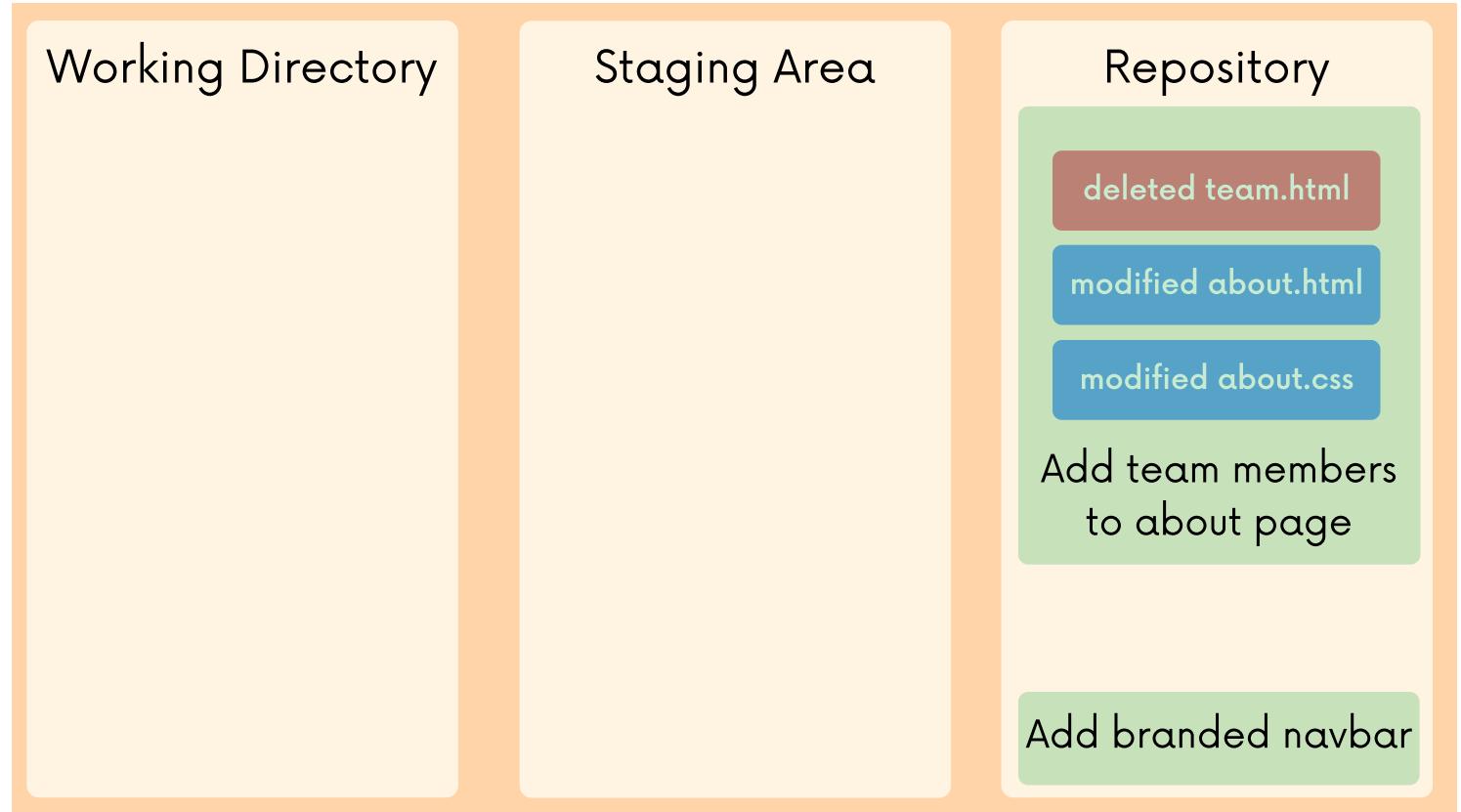


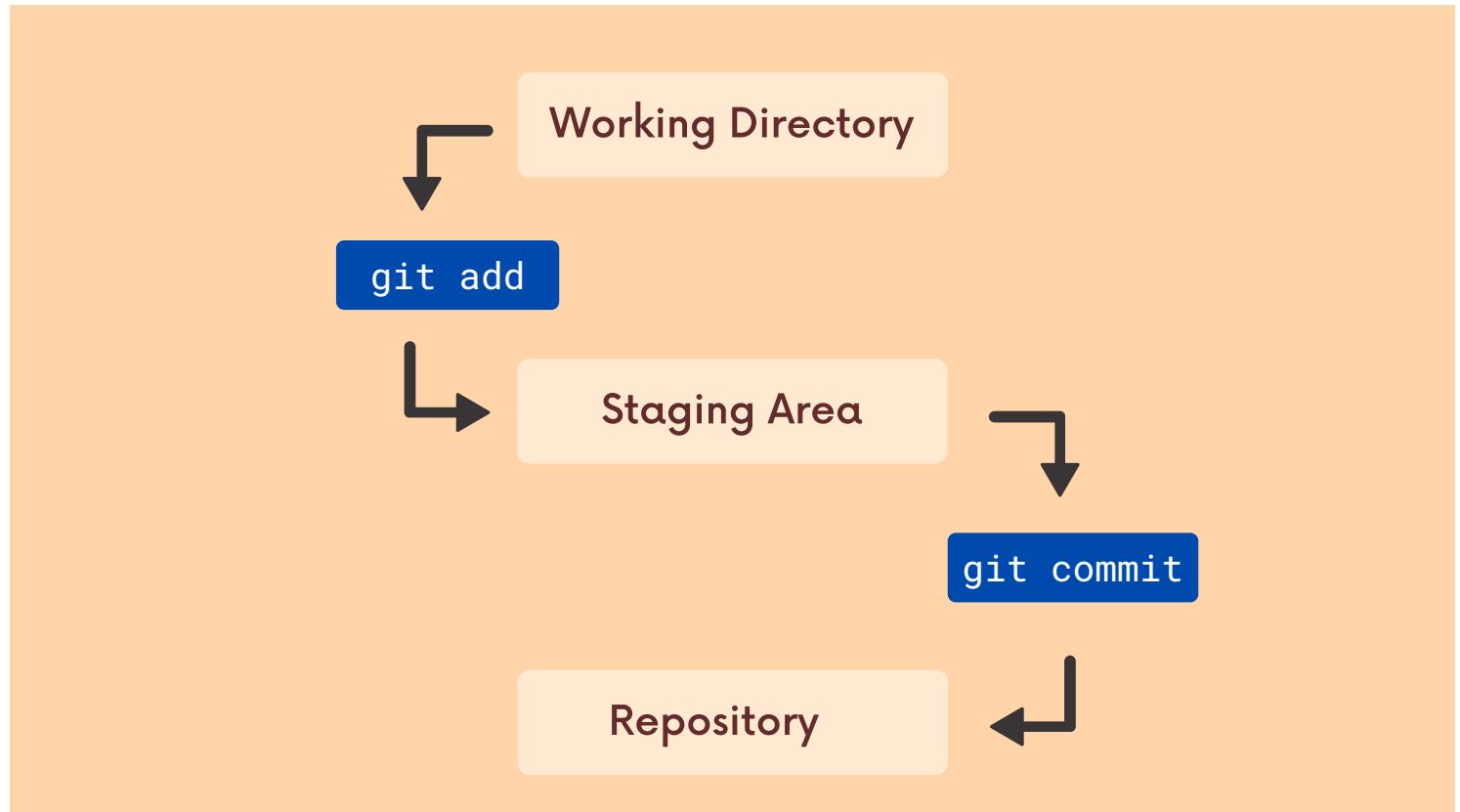












≡

Adding

Use `git add` to add specific files to the staging area. Separate files with spaces to add multiple at once.

```
git add file1 file2
```





Adding

Use `git add .` to stage all changes at once



```
git add .
```



Working Directory

modified navbar.html

modified about.html

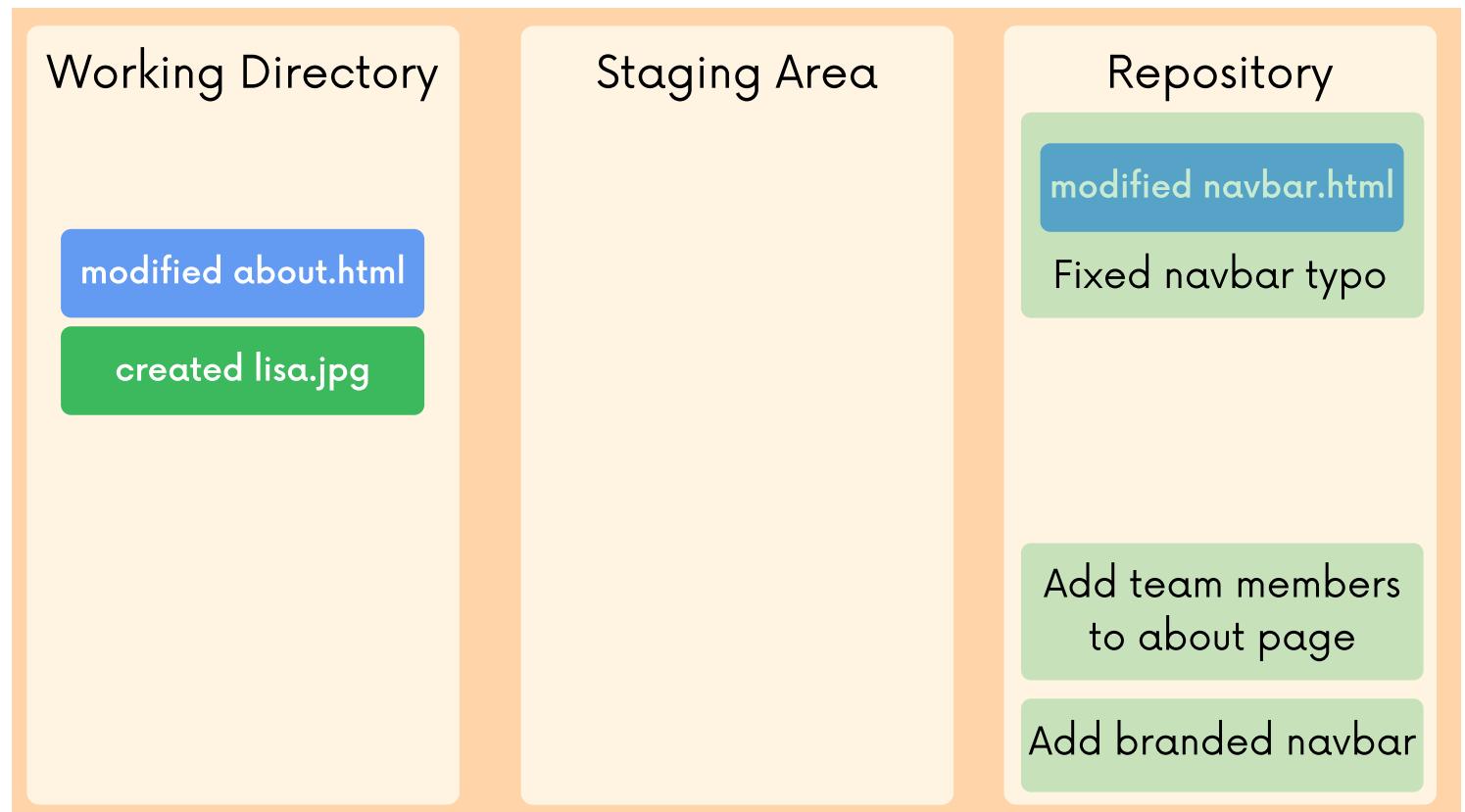
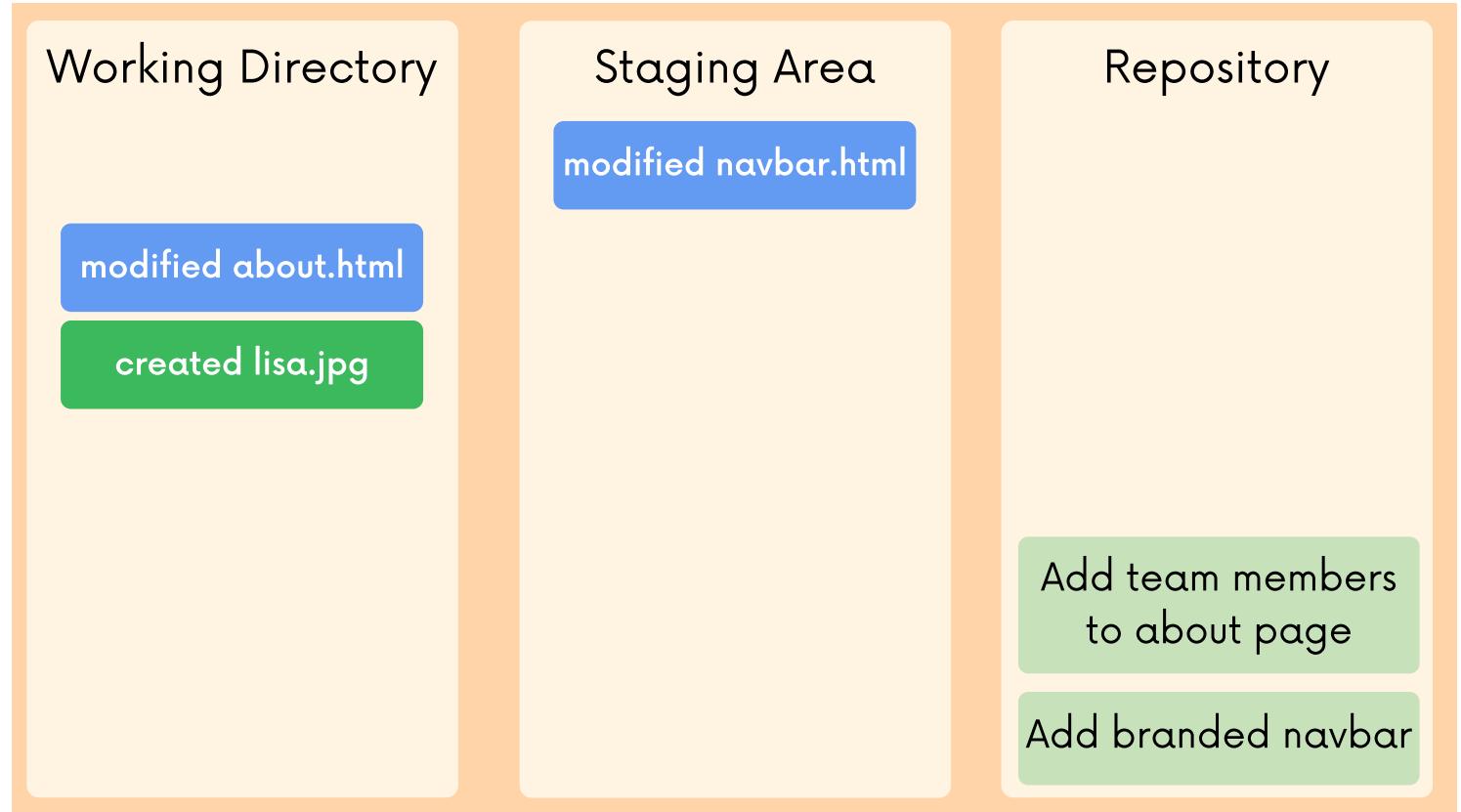
created lisa.jpg

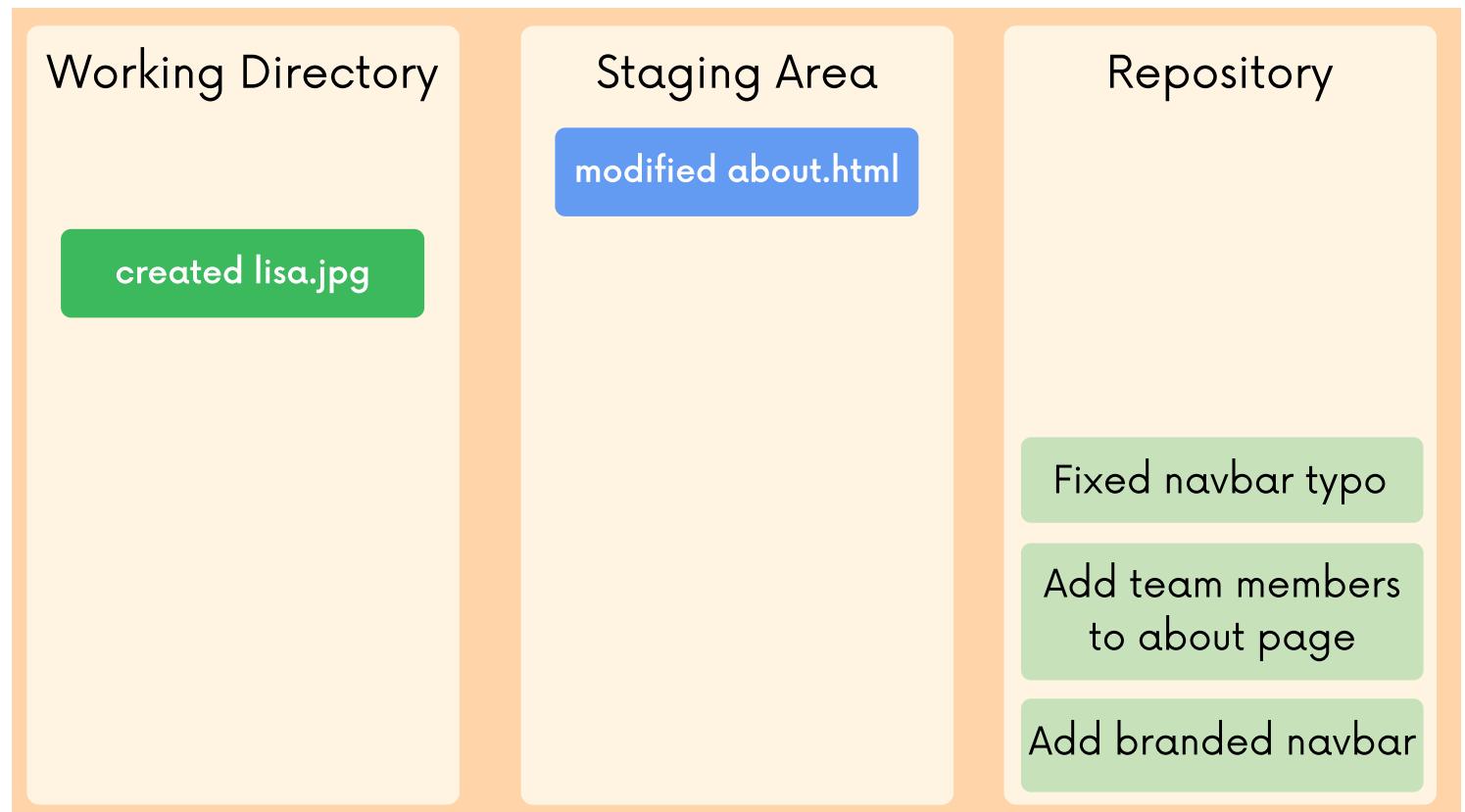
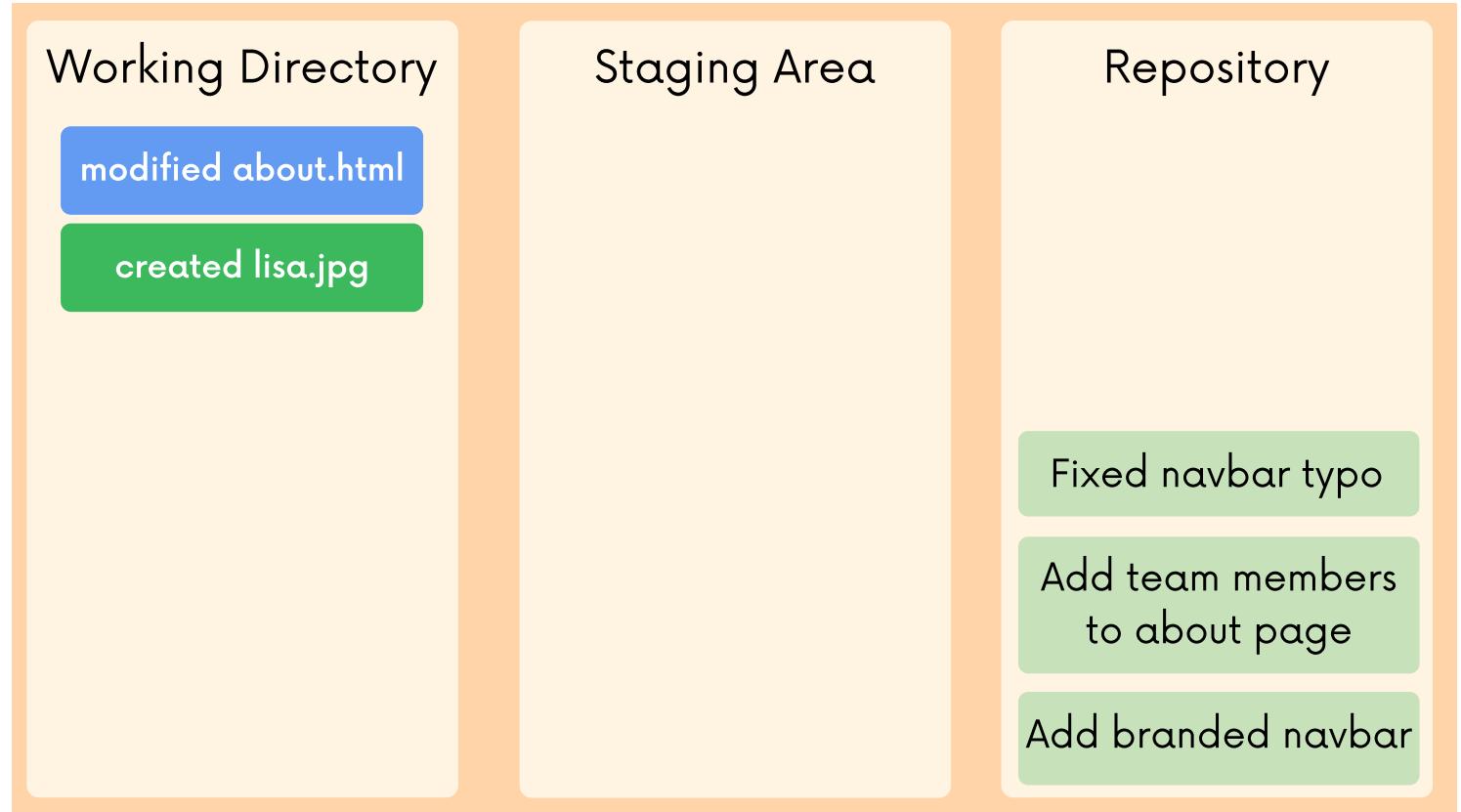
Staging Area

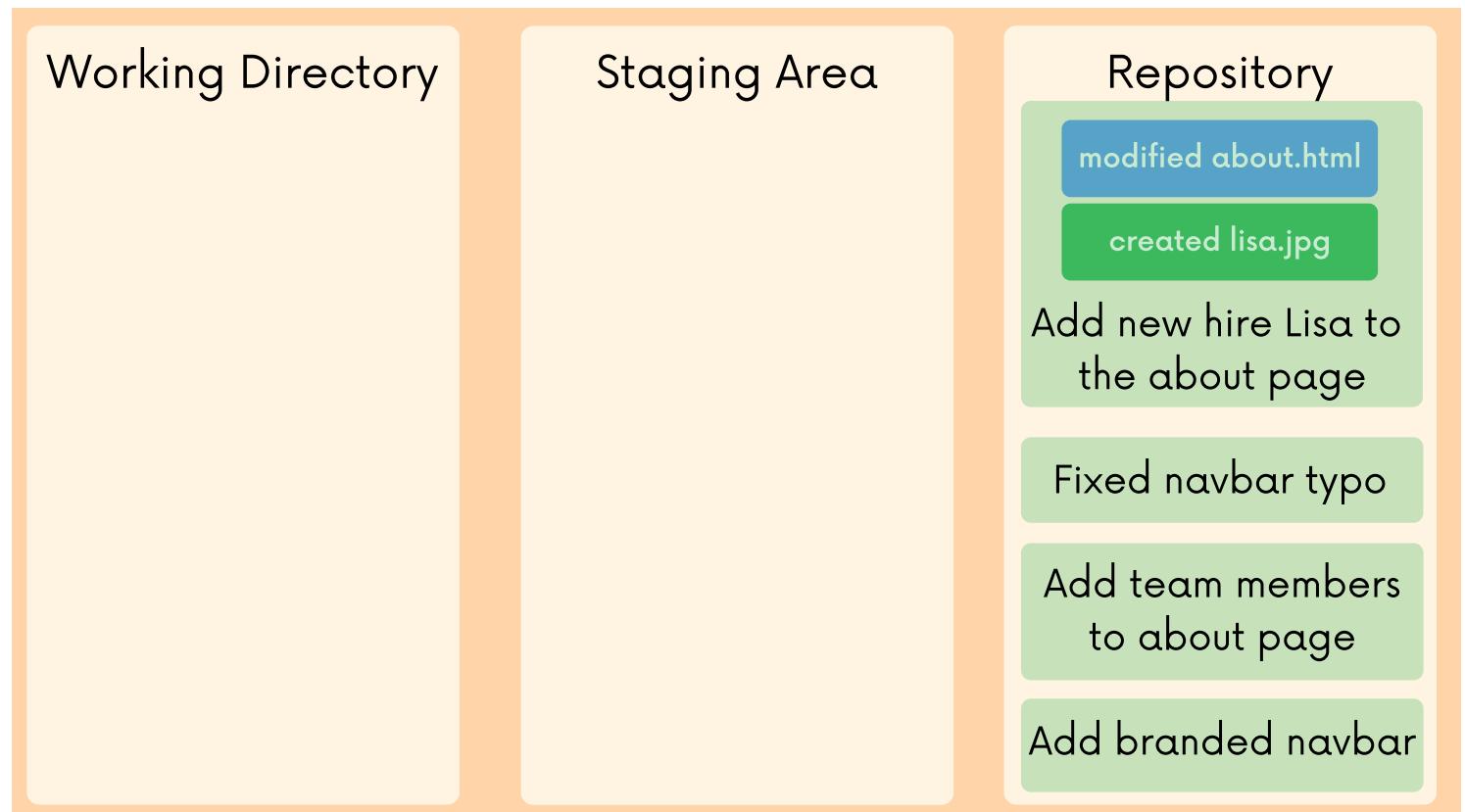
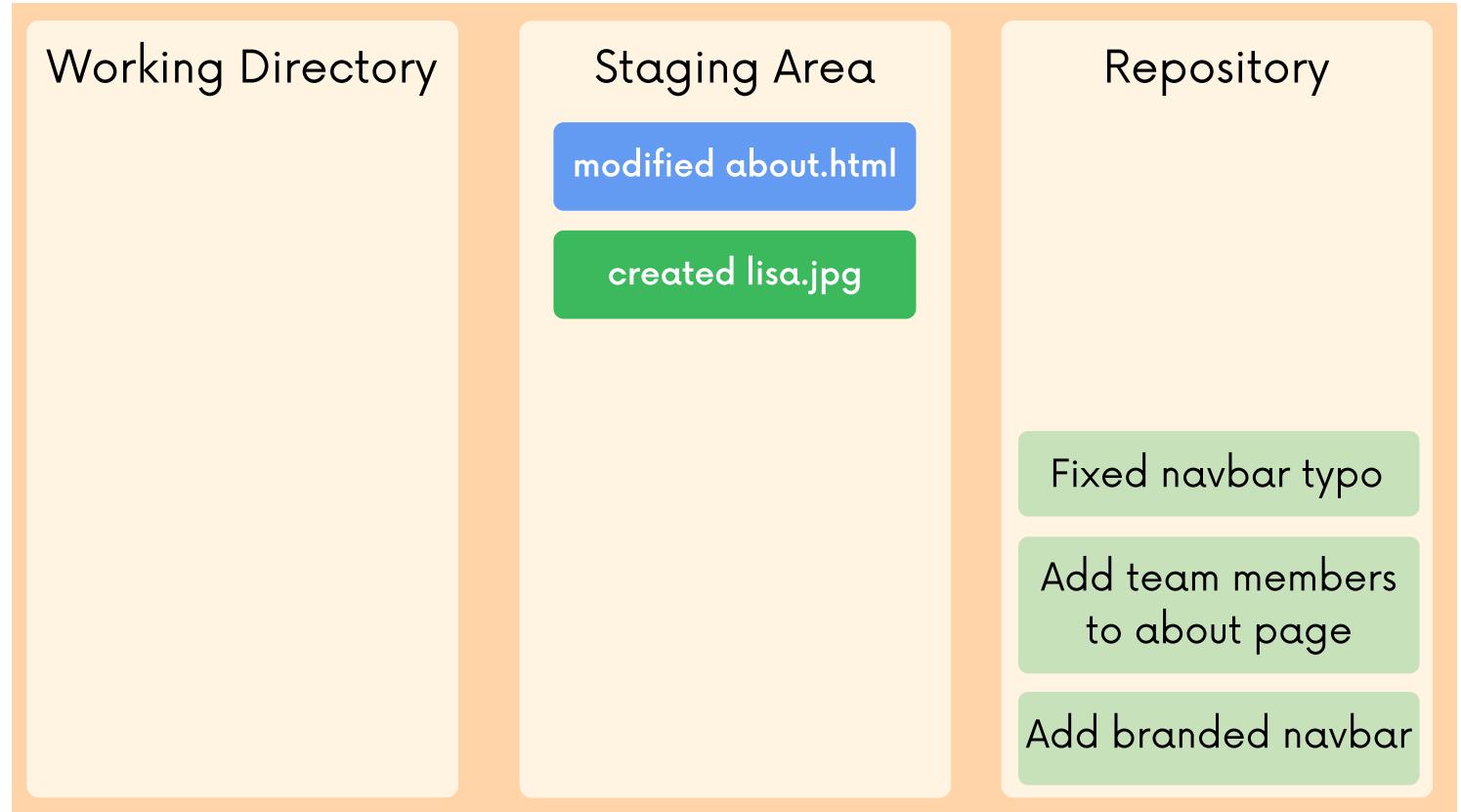
Repository

Add team members
to about page

Add branded navbar







Working Directory

Staging Area

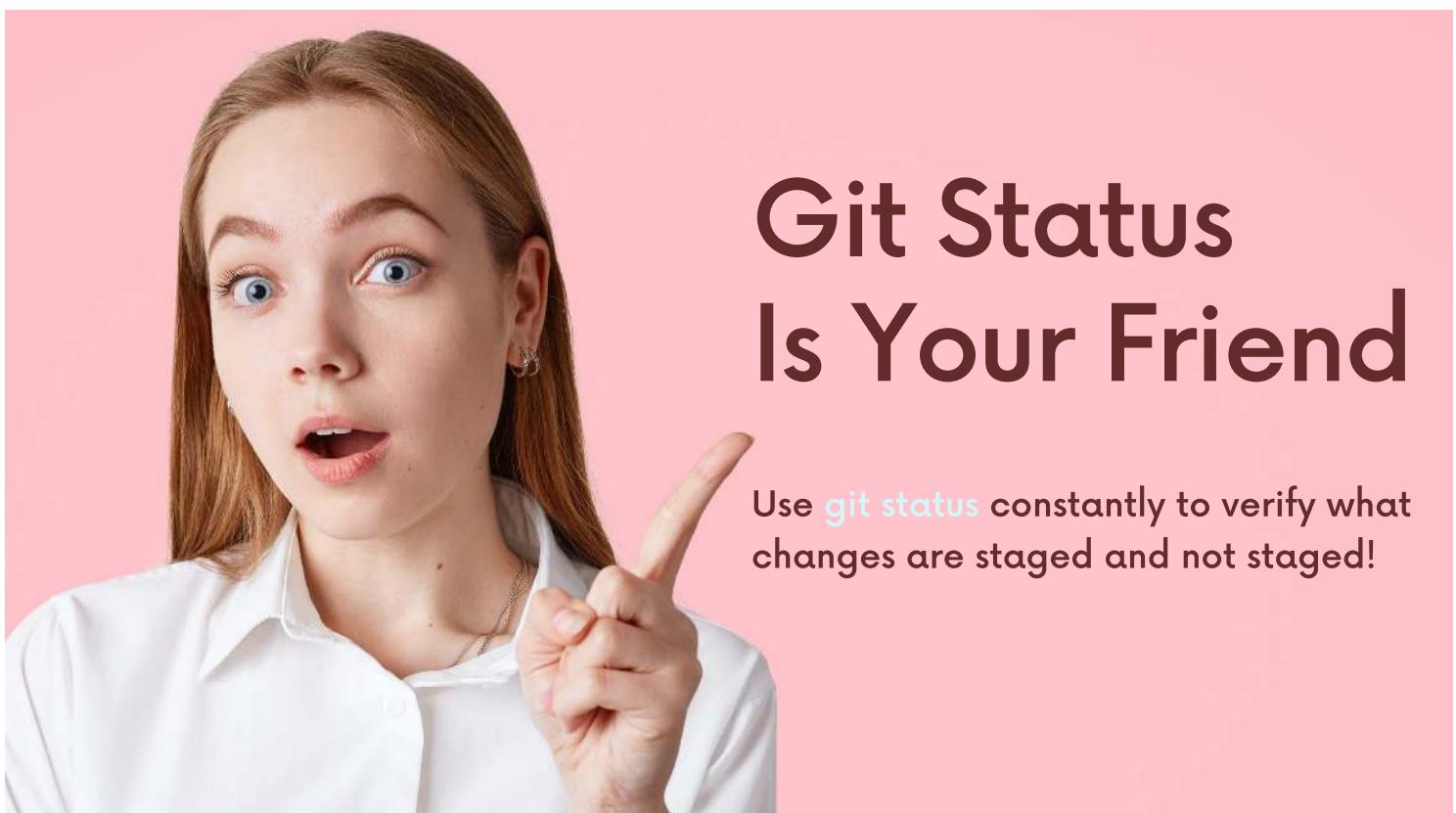
Repository

Add new hire Lisa to
the about page

Fixed navbar typo

Add team members
to about page

Add branded navbar



Git Status Is Your Friend

Use `git status` constantly to verify what changes are staged and not staged!



Git Commit

We use the `git commit` command to actually commit changes from the staging area.

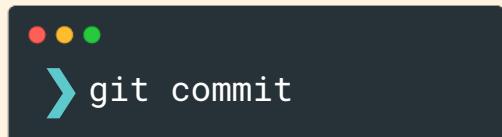
When making a commit, we need to provide a commit message that summarizes the changes and work snapshotted in the commit



Git Commit

Running `git commit` will commit all staged changes. It also opens up a text editor and prompts you for a commit message.

This can be overwhelming when you're starting out, so instead you can use...



Git Commit

The `-m` flag allows us to pass in an inline commit message, rather than launching a text editor.

We'll learn more about writing good commit messages later on.



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



4

Committing in detail



More About Committing & Other Stuff



Git Commit

The `-m` flag allows us to pass in an inline commit message, rather than launching a text editor.

We'll learn more about writing good commit messages later on.

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





Amending Commits

Suppose you just made a commit and then realized 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

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



Some Basic Guidelines

- Commit early and often
- Make commits atomic (group similar changes together, don't commit a million things at once)
- Write meaningful but concise commit messages



The Git Docs

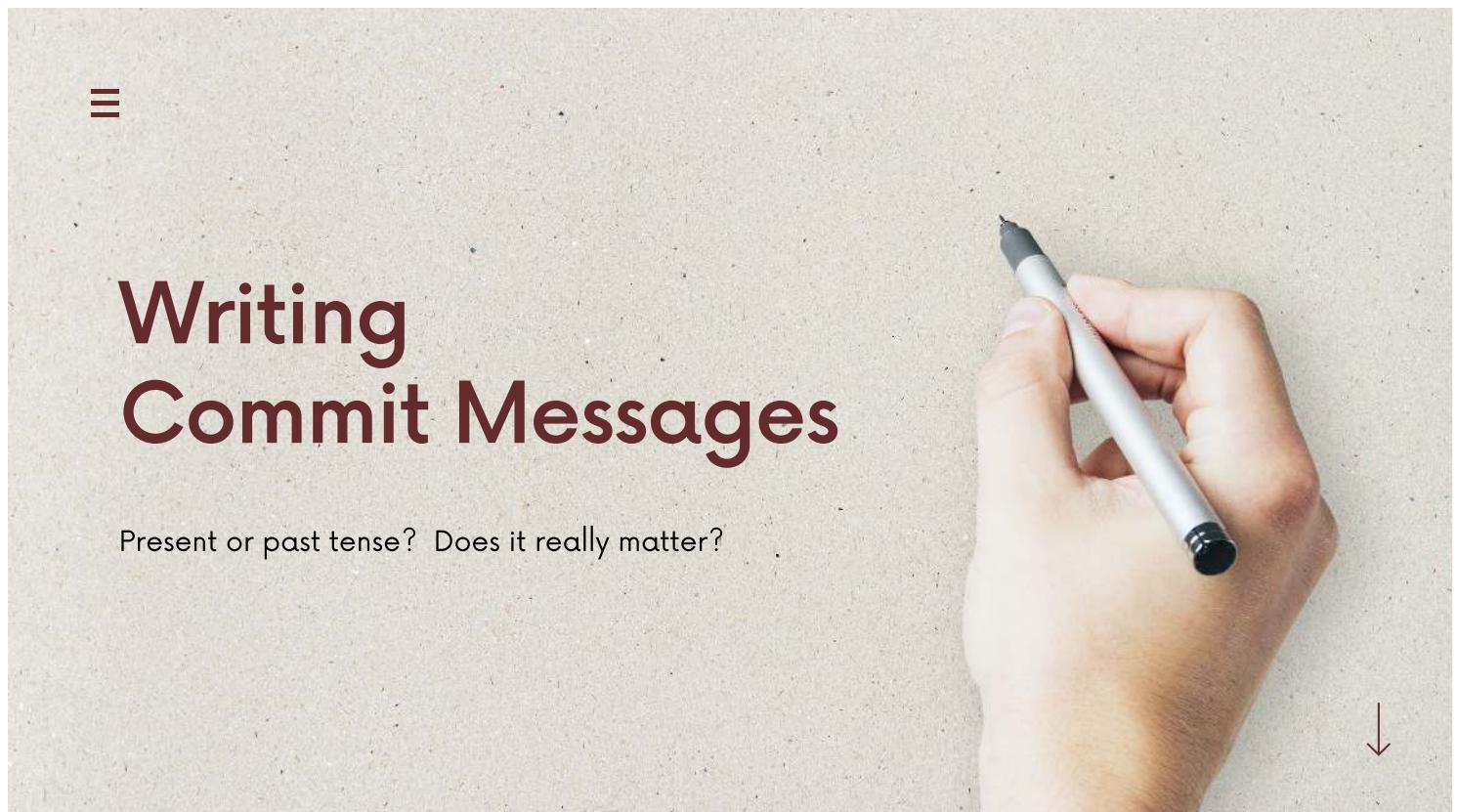


The Git Docs





The Git Docs



Writing Commit Messages

Present or past tense? Does it really matter?



≡

Present-Tense Imperative Style??

From the Git docs:

Describe your changes in imperative mood, e.g. "make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "I changed xyzzy to do frotz", as if you are giving orders to the codebase to change its behavior.



≡

You do NOT have to follow this pattern

Though the Git docs suggest using present-tense imperative messages, many developers prefer to use past-tense messages. All that matters is consistency, especially when working on a team with many people making commits





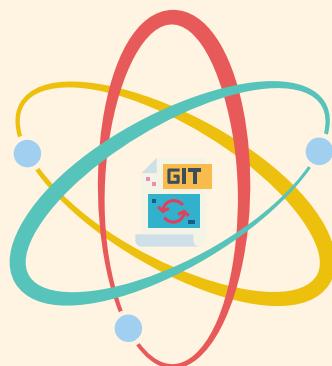
Configuring Your Default Editor



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.

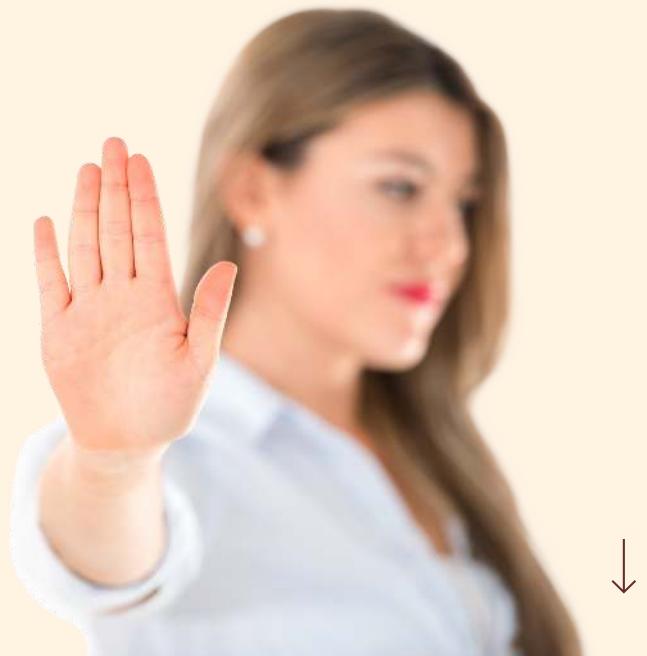




Ignoring Files

We can tell Git which files and directories to ignore in a 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`

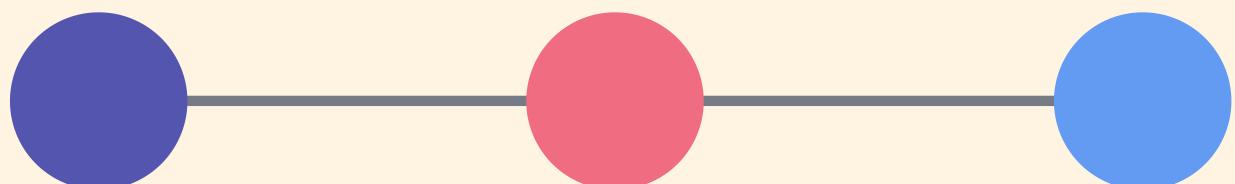
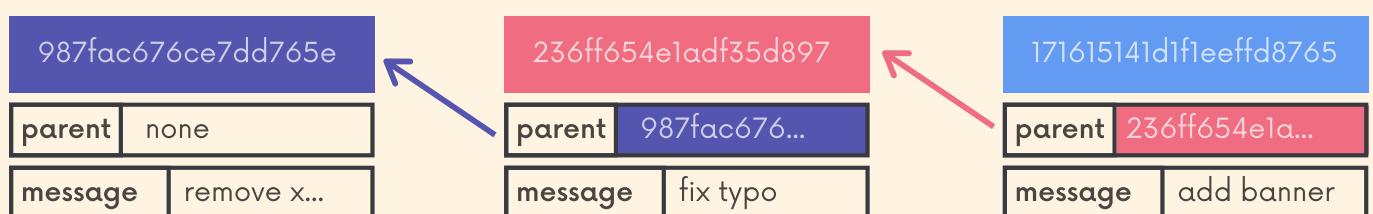
Create a file called `.gitignore` in the root of a repository. Inside the file, we can write patterns to tell Git which files & folders to ignore:

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



5 Branching

Git 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.



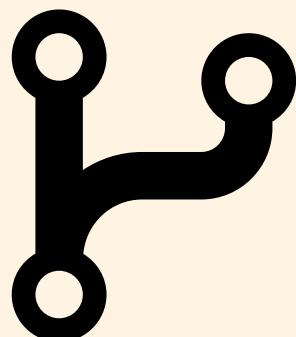
Branches

Branches are an essential part of Git!

Think of branches as alternative timelines for a project.

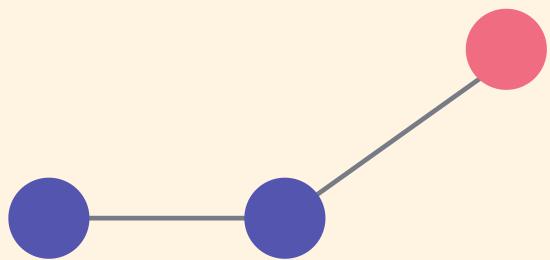
They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel.

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

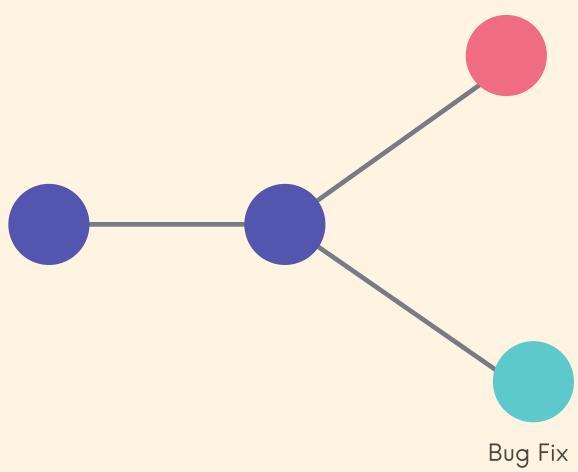




New Color Scheme

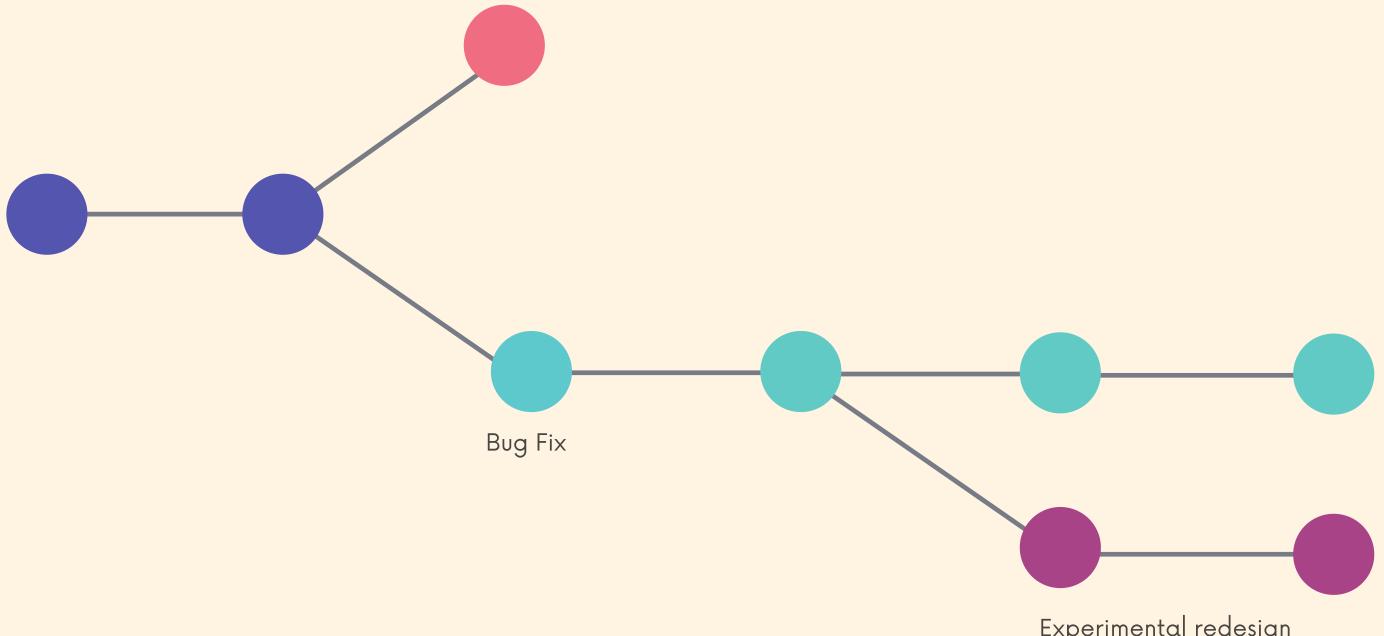


New Color Scheme



Bug Fix

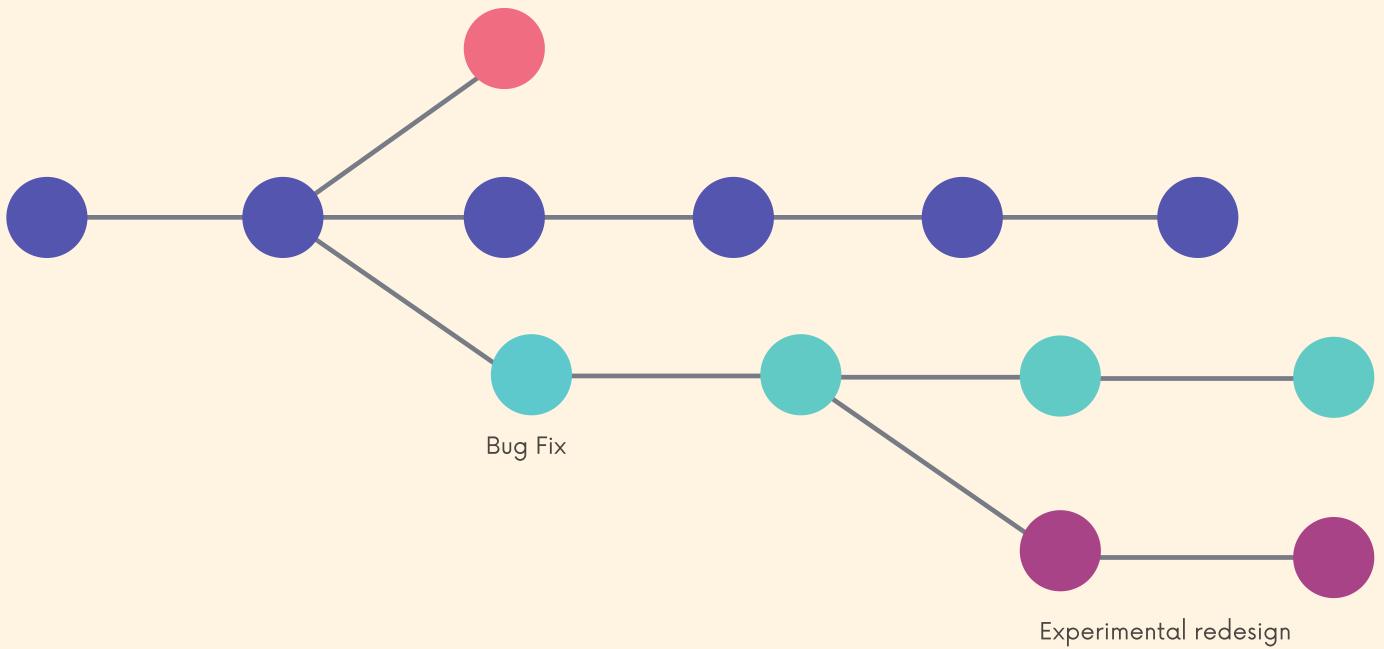
New Color Scheme



Bug Fix

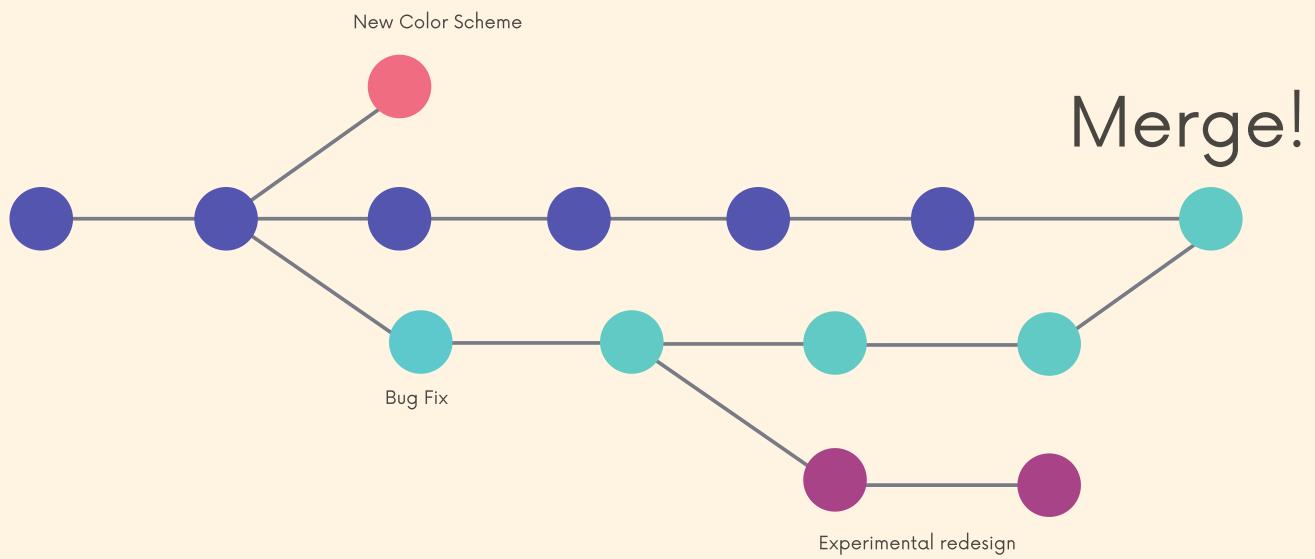
Experimental redesign

New Color Scheme



Bug Fix

Experimental redesign



You Probably Have Seen This Before

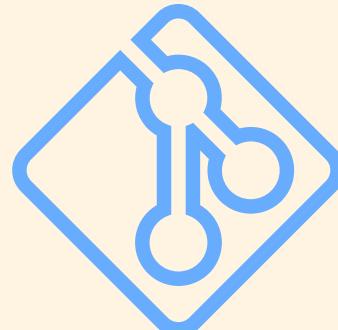
On branch master
nothing to commit



The Master Branch

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.



*technically that's not 100% true as we'll see later



Master

Many people designate the master branch as their "source of truth" or the "official branch" for their codebase, but that is left to you to decide.

From Git's perspective, the master branch is just like any other branch. It does not have to hold the "master copy" of your project.



Master? Main?

In 2020, Github renamed the default branch from **master** to **main**. The default Git branch name is still **master**, though the Git team is exploring a potential change.
We will circle back to this shortly.

Branching

Master Branch



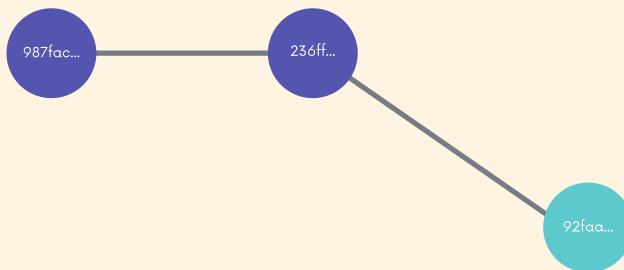
Branching

Master Branch



Branching

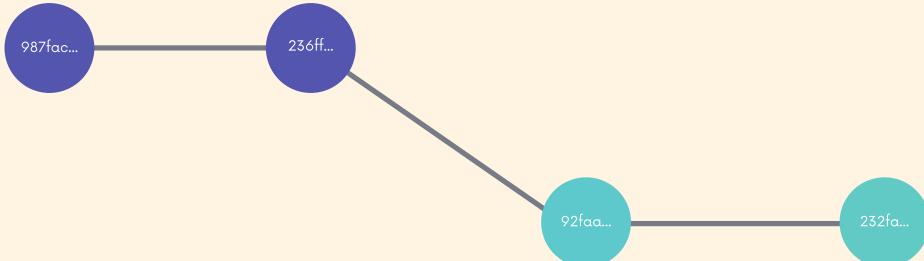
Master Branch



Experimental Branch

Branching

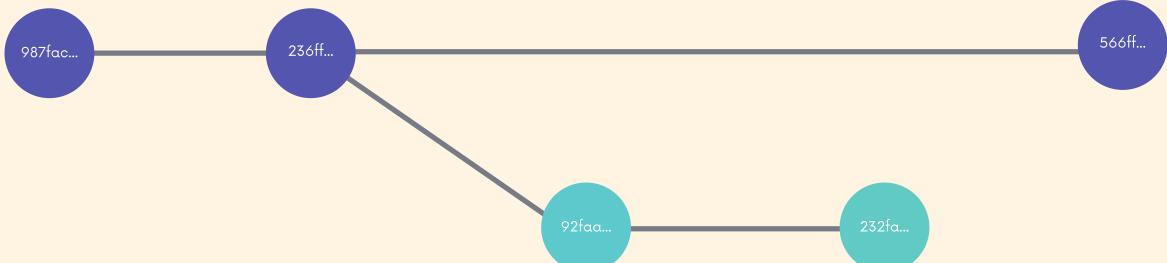
Master Branch



Experimental Branch

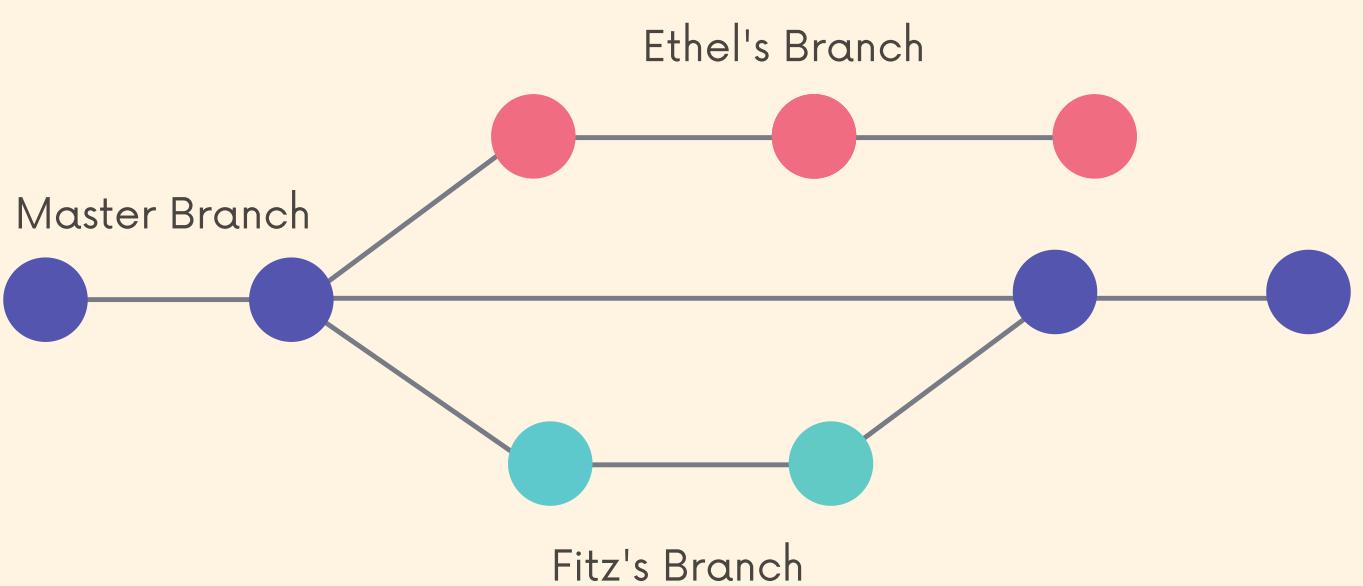
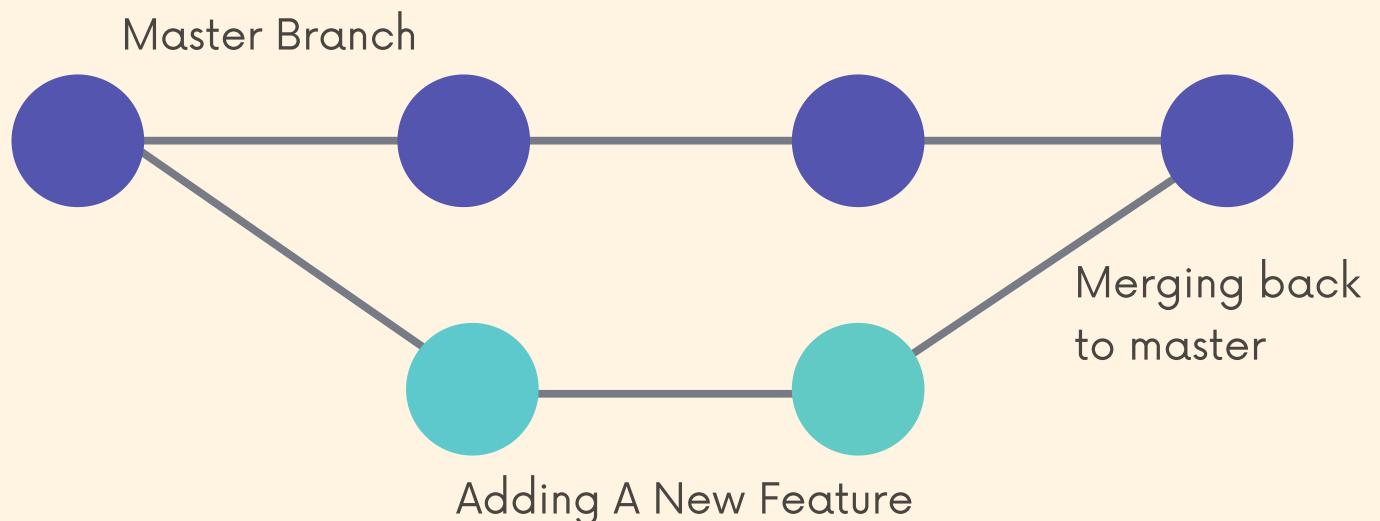
Branching

Master Branch



Experimental Branch

A Common Workflow





commit bca5fc8b05deda4a13e7588c7ca352e47560c1dd
(HEAD -> master)



WTF IS THAT CRAZINESS

HEAD? What Is That?

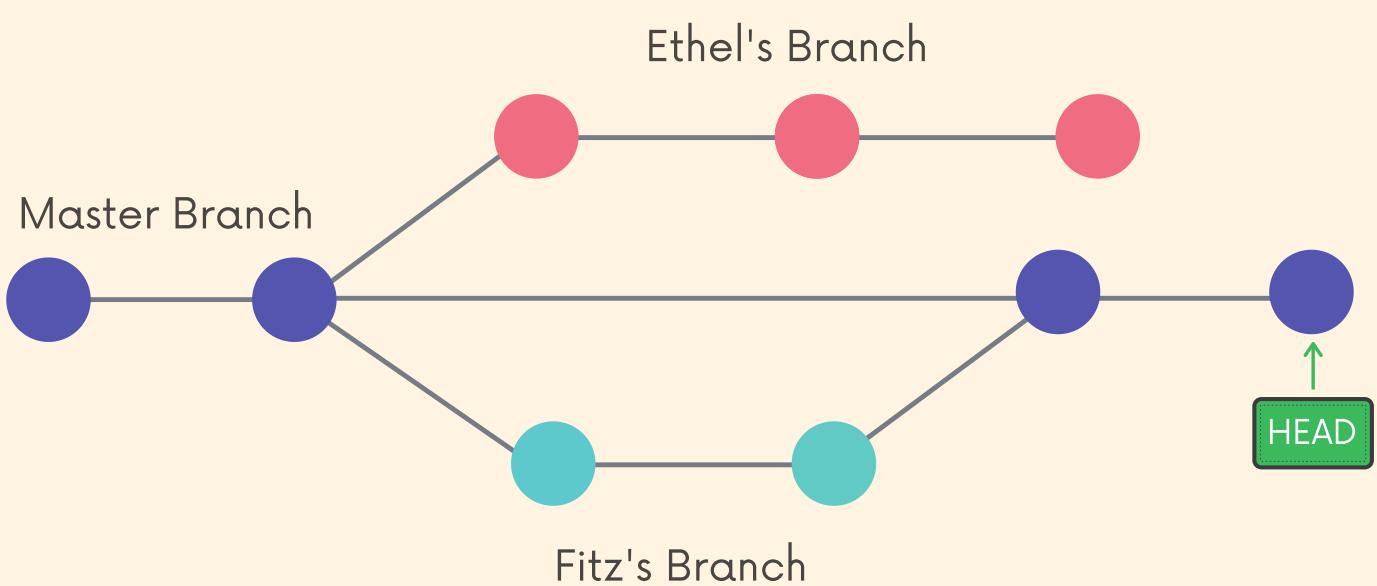
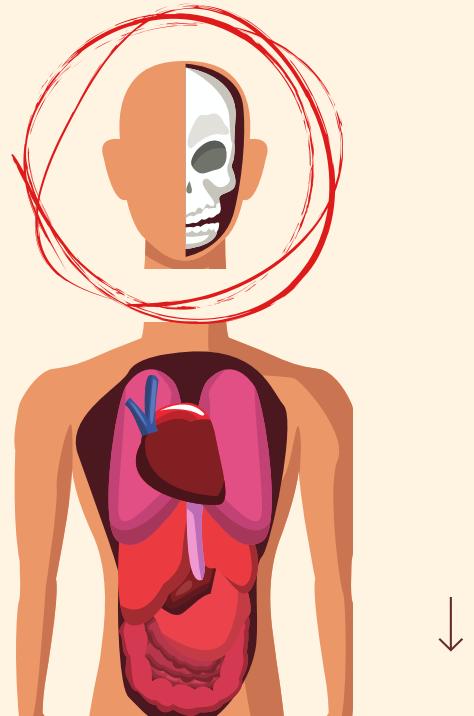


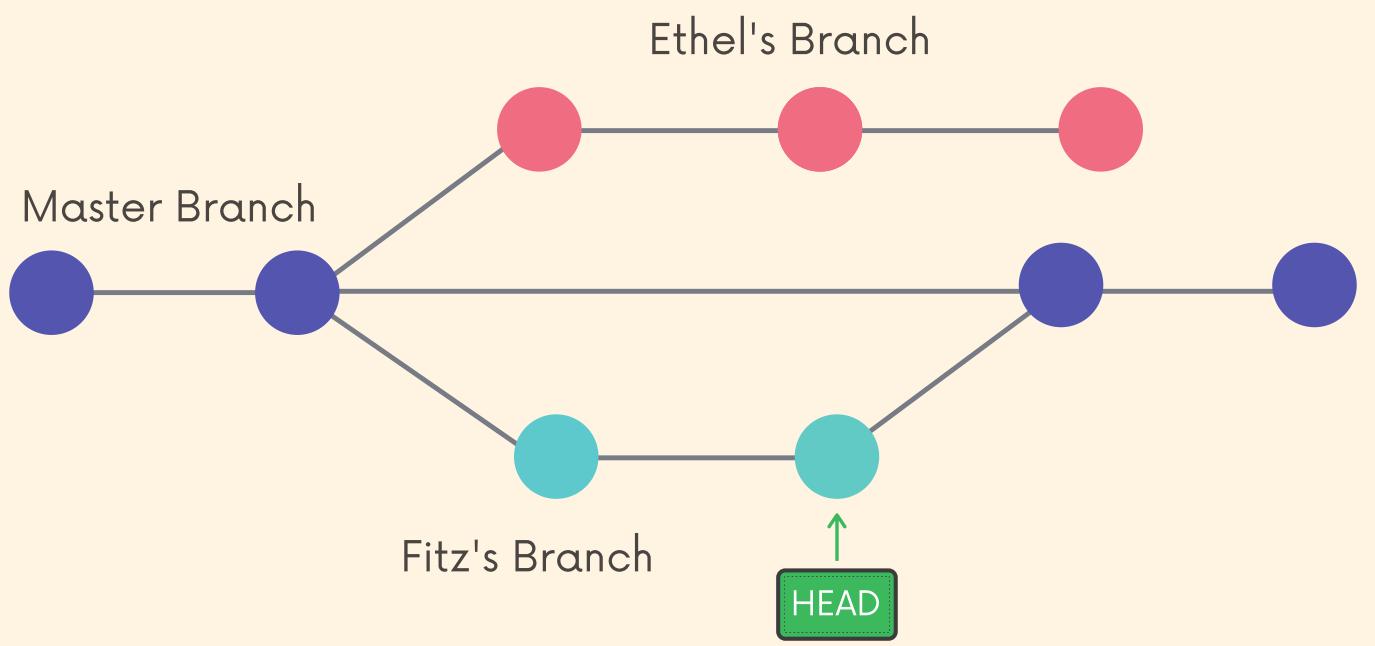
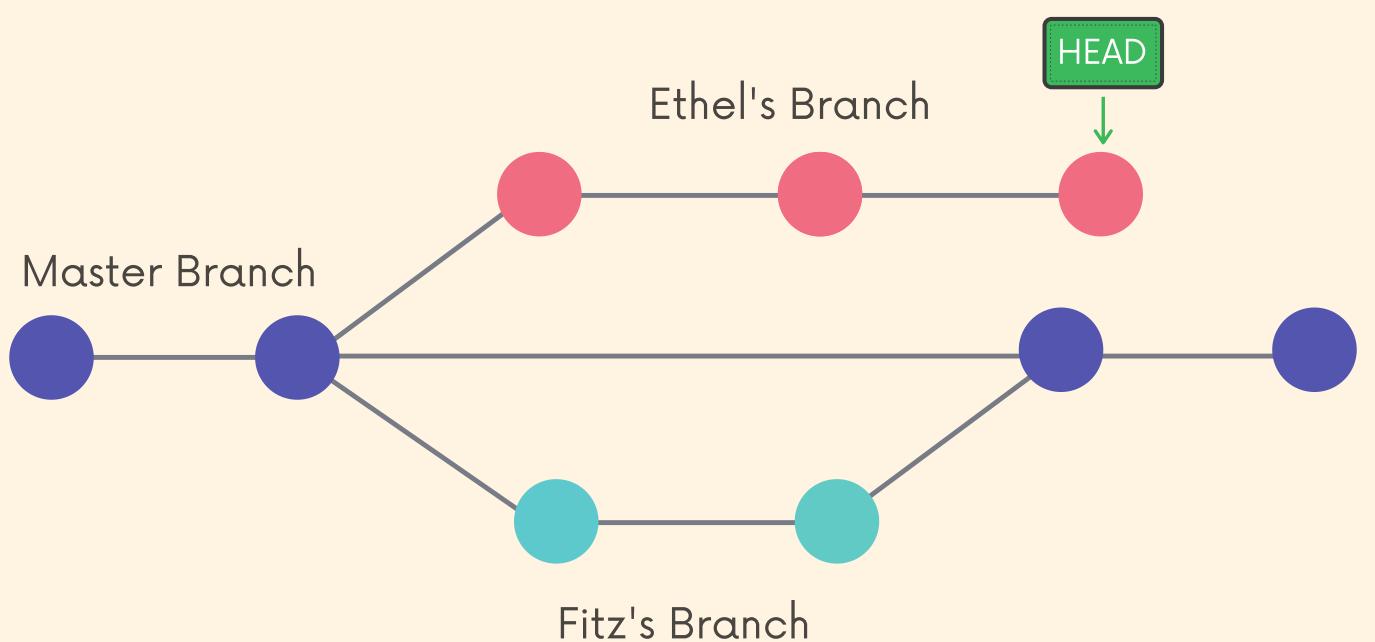
HEAD

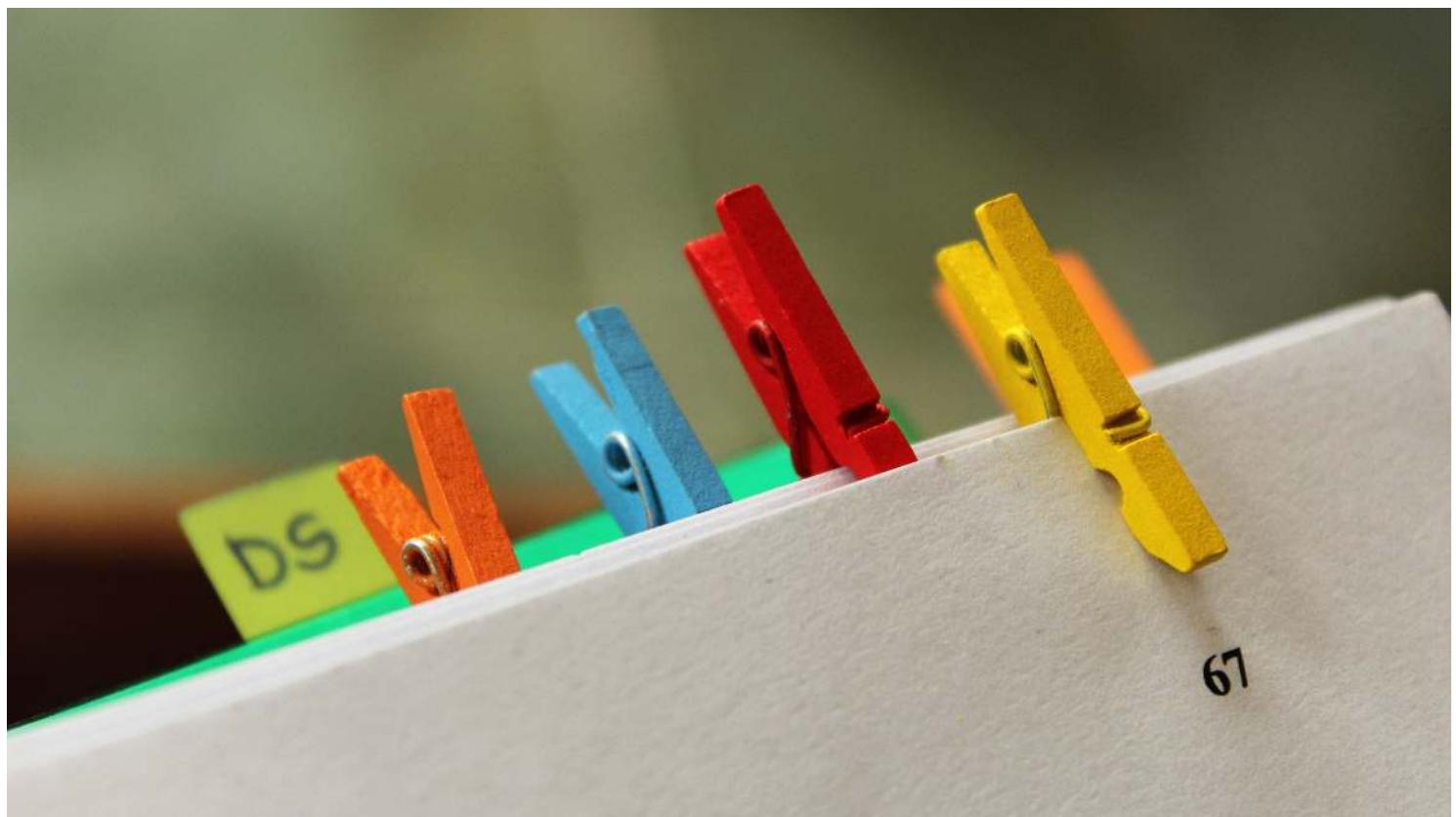
We'll often come across the term **HEAD** in Git.

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!









574

king and spitting.
Sovereign LORD helps me,
be disgraced.
ave I set my face like flint,
w I will not be put to
ne.
dictates me is near.
n will bring charges against
e each other!
accuser?
confront me!
ereign LORD who helps me?
e that will condemn me?
ll wear out like a garment;
is will eat them up.
g you fears the LORD
s the word of his servant?
o walks in the dark,
no light,
name of the Lord
on his God.
ll you who light fires
ide yourselves with flaming
ches,
the light of your fires
se torches you have set
are.

The islands will look to me
and wait in hope for my arm.
6Lift up your eyes to the heavens,
look at the earth beneath;
the heavens will vanish like smoke,
the earth will wear out like a
garment
and its inhabitants die like flies;
But my salvation will last forever,
my righteousness will never
fail.

7“Hear me, you who know what
you people who have my law
hearts:
Do not fear the reproach of men,
or be terrified by their insults.
8For the moth will eat them up
garment;
the worm will devour them
wool.
But my righteousness will last
my salvation through all generations.

9Awake, awake! Clothe yourself
strength,
O arm of the Lord;
awake, as in days gone by,
as in generations of old.
Was it not you who cut Rahab to pieces,
who pierced that mighty serpent?

575

the LORD Almighty is his name.
10I have put my words in your mouth
and covered you with the shadow of
my hand—
11I who set the heavens in place,
who laid the foundations of the earth,
and who say to Zion, ‘You are my
people.’”

The Cup of the LORD's Wrath

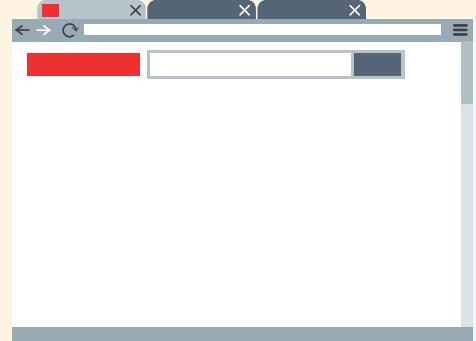
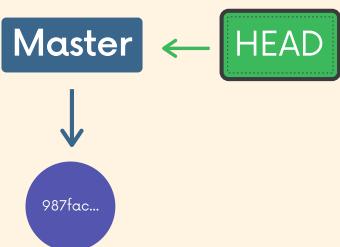
12Awake, awake!
Rise up, O Jerusalem,
you who have drunk from the hand of
the LORD
the cup of his wrath,
you who have drained to its dregs
the goblet that makes men stagger.
13Of all the sons she bore
there was none to guide her;
of all the sons she reared
there was none to take her by the
hand.
14These double calamities have come
upon you—
who can comfort you?—
ruin and destruction, famine and
sword—
who can console you?
15Your sons have fainted;
they lie at the head of every street,
like prisoners in a net.

Free yourself from the
neck,
O captive Daughte
3For this is what the
“You were sold for r
and without mon
redeemed.”

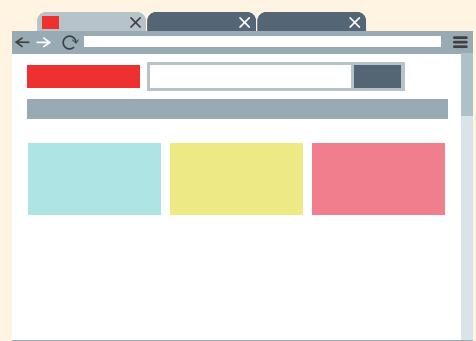
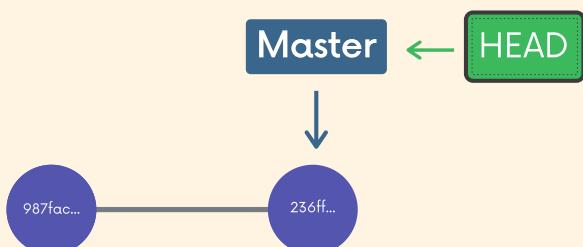
4For this is what th
“At first my people
to live;
lately, Assyria h
5“And now what
clares the LORD.
“For my people h
for nothi
and those wh
“And all day lon
my name is c
6Therefore my p
name;
therefore in
that it is I who
Yes, it is I.”

7How beautiful
are the feet

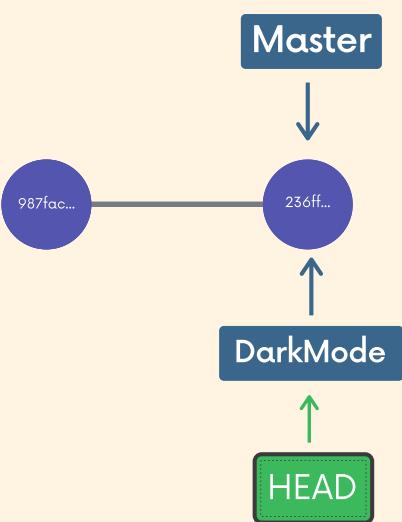
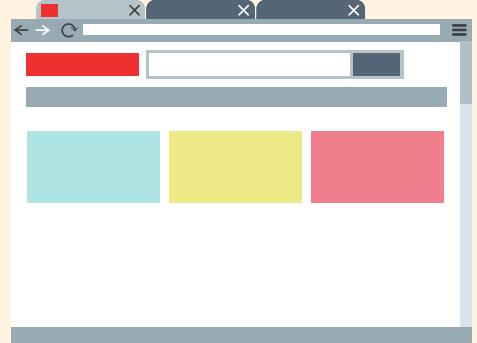
My First Commit On Master



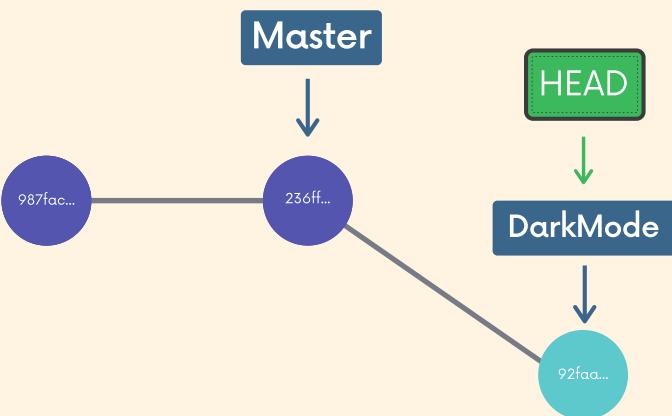
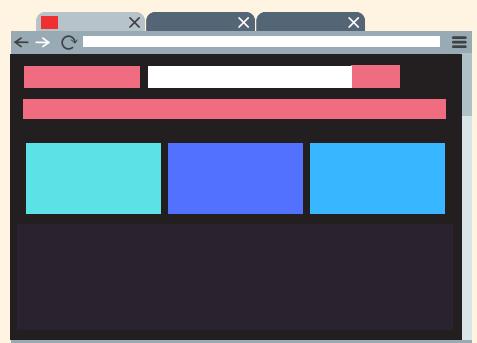
Another Commit



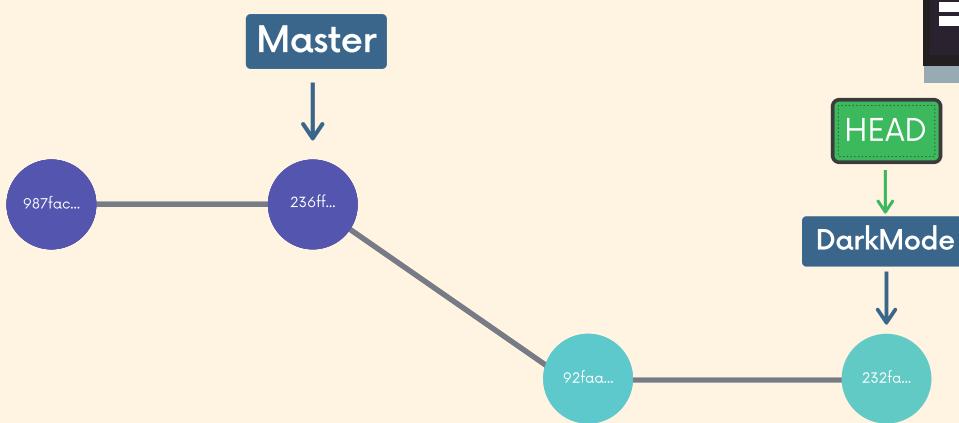
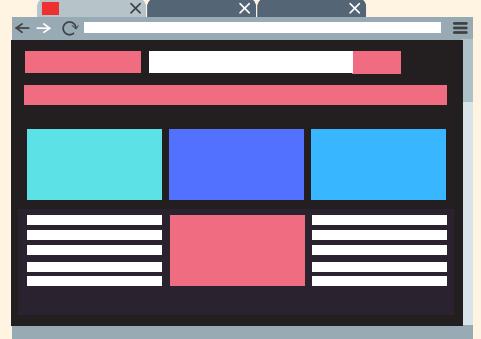
I make a new branch!



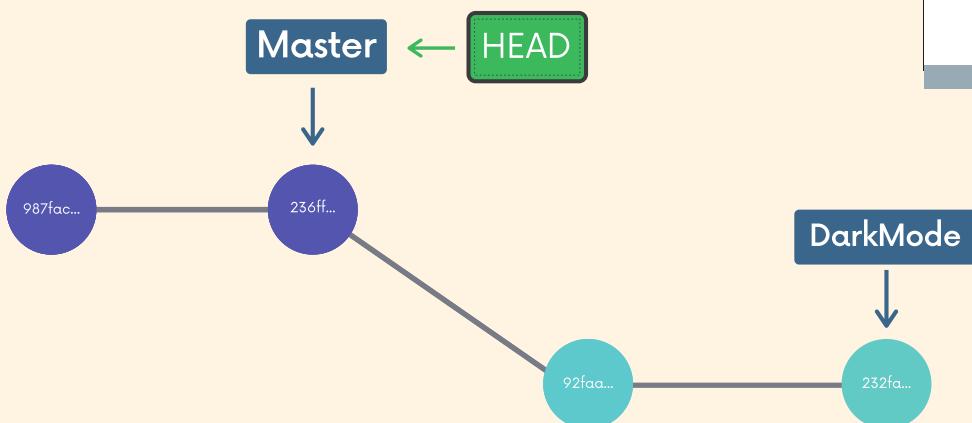
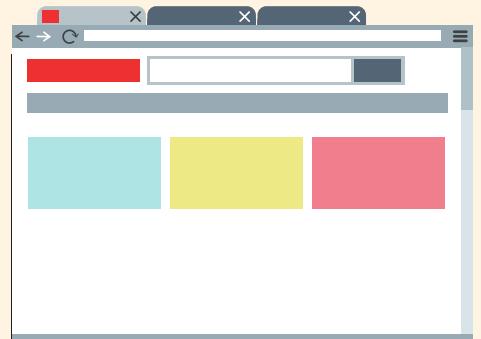
I make a commit on the new branch



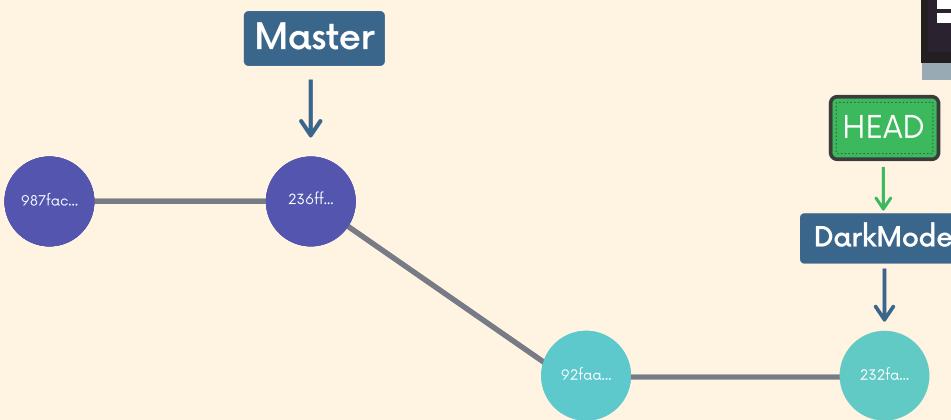
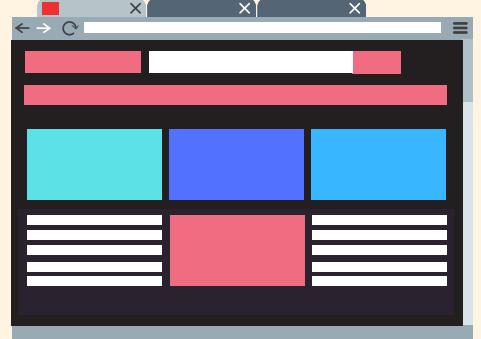
2nd Commit On New Branch



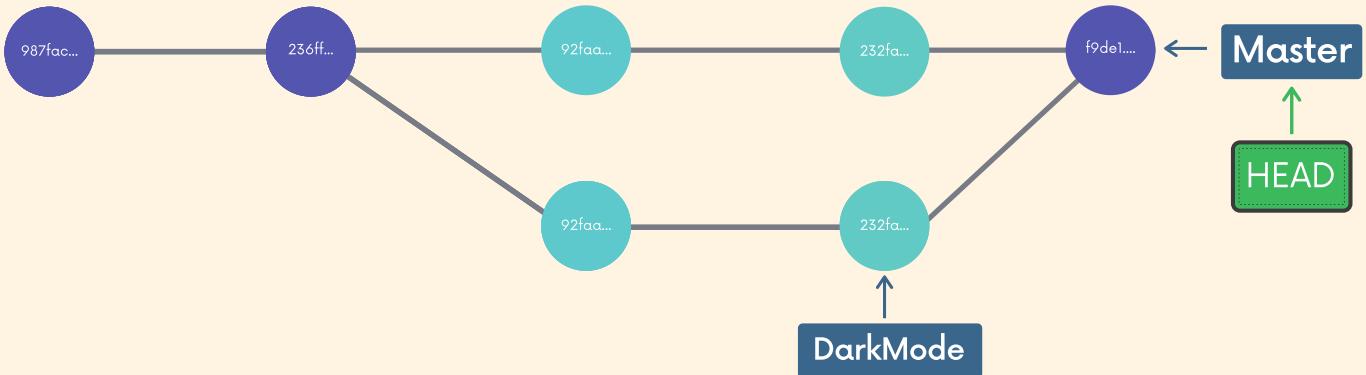
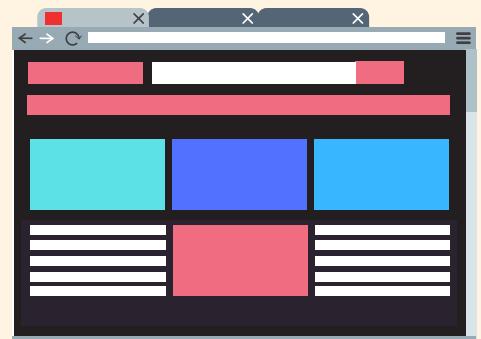
Return to Master



Going Back To My Dark Mode Branch



Merge Changes Into Master Branch?





Viewing Branches

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

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



```
git branch
```



Creating Branches

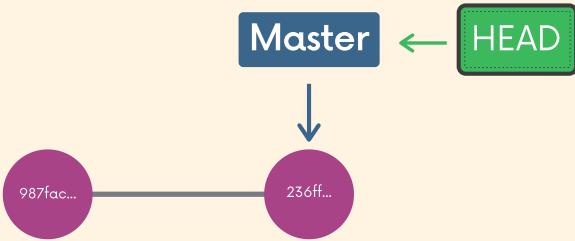
Use `git branch <branch-name>` to make a new branch based upon the current HEAD

This just creates the branch. It does not switch you to that branch (the HEAD stays the same)

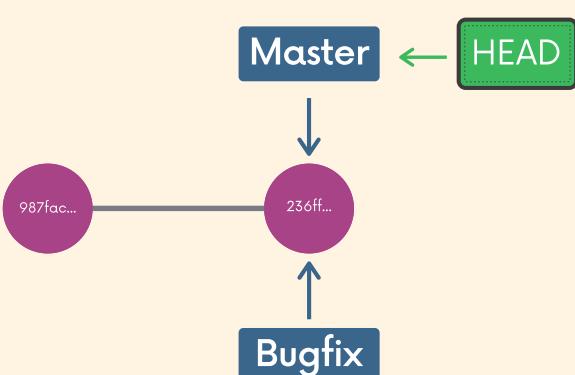


```
git branch <branch-name>
```





```
git branch bugfix
```



I've made a new branch,
but I'm still working on master.

Switching Branches

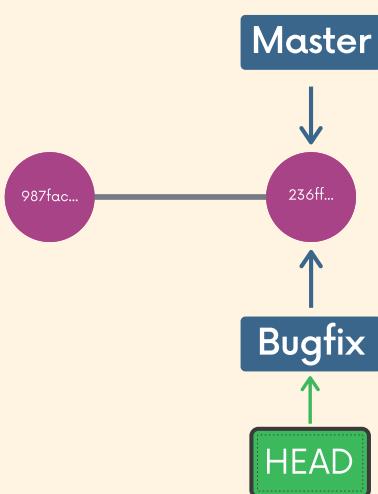
Once you have created a new branch,
use `git switch <branch-name>` to switch to it.



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



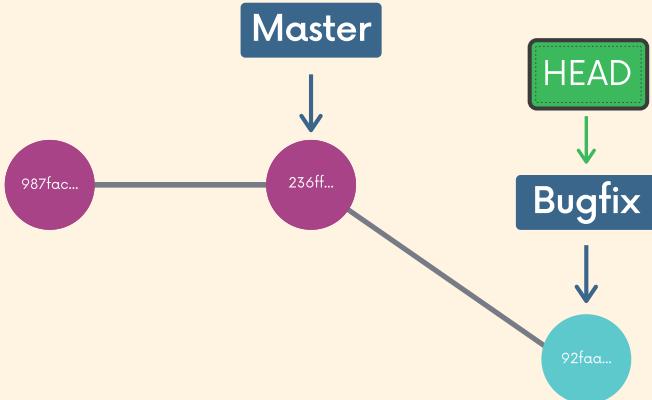
```
git switch bugfix
```



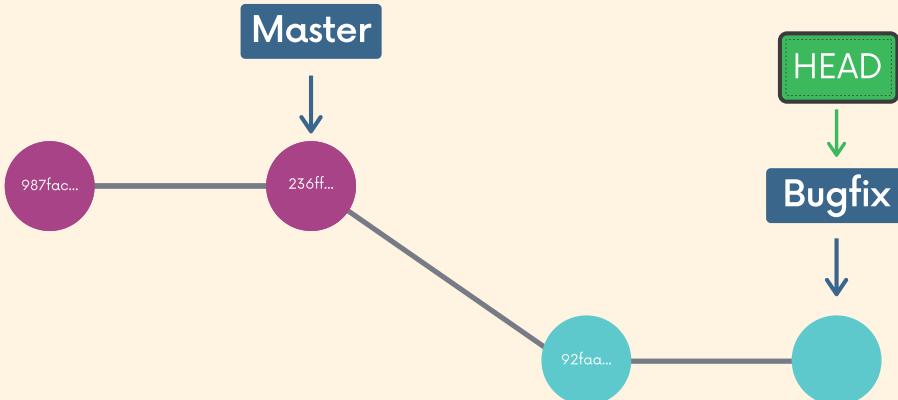
I've now switched over to the
new bugfix branch!

Notice that HEAD is now
pointing to Bugfix, not master.

When I make a new commit, it exists only on my new branch, not on master!



And another commit...



```
git switch bugfix
```

master branch



236ff...

bugfix branch



HEAD

The commit exists only on my new branch, not on master!



Another way of switching??

Historically, we used `git checkout <branch-name>` to switch branches. This still works.

The `checkout` command does a million additional things, so the decision was made to add a standalone switch command which is much simpler.

You will see older tutorials and docs using `checkout` rather than `switch`. Both now work.



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



Creating & Switching

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

Remember **-c** as short for "create"

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



master branch



```
git switch -c refactor
```

master branch



236ff...

refactor branch



the **-c** flag makes the new refactor branch and switches to it all at once!

master branch



236ff...

bugfix branch



2456...



Remember, branches are made **based on the HEAD**

master branch



236ff...



bugfix branch

2456...

HEAD



new branch!

If I branch from the bugfix branch,
my new branch includes all the
commits from bugfix.



Viewing More Info

Use the `-v` flag with `git branch` to view more information about each branch.



git branch -v

| | | |
|----------|---------|-------------------|
| * master | 88df51f | add documentation |
| bugfix | 7c2a586 | fix hover bug |
| chatdemo | afadcf9 | add chat widget |



6 Merging

Merging Branches

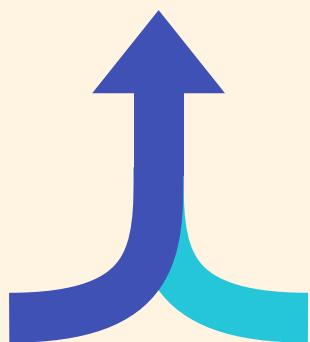


≡

Merging

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





Merging

The merge command can sometimes confuse students early on. 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.

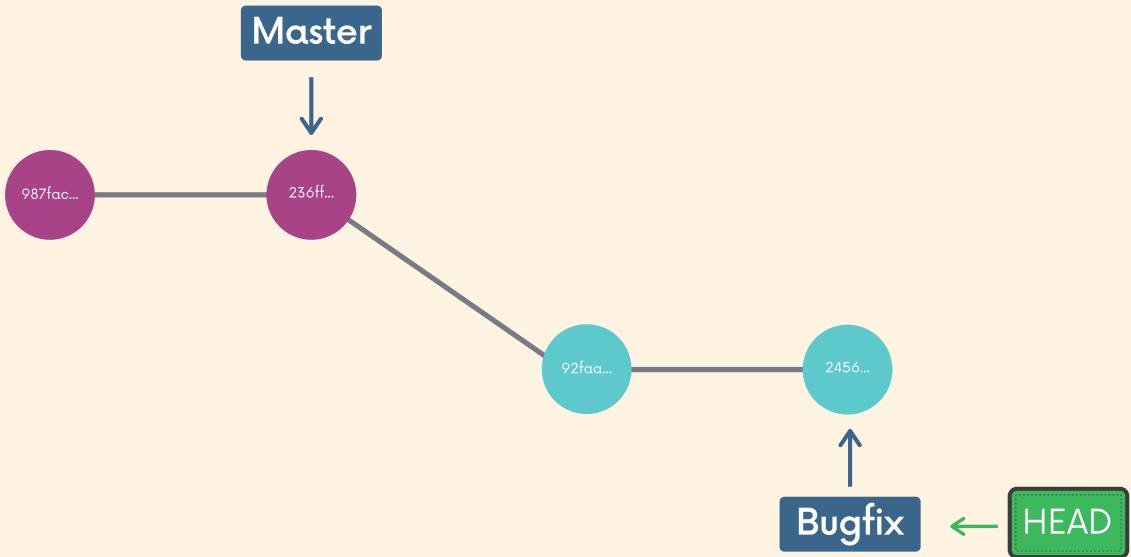
To merge the bugfix branch into master...



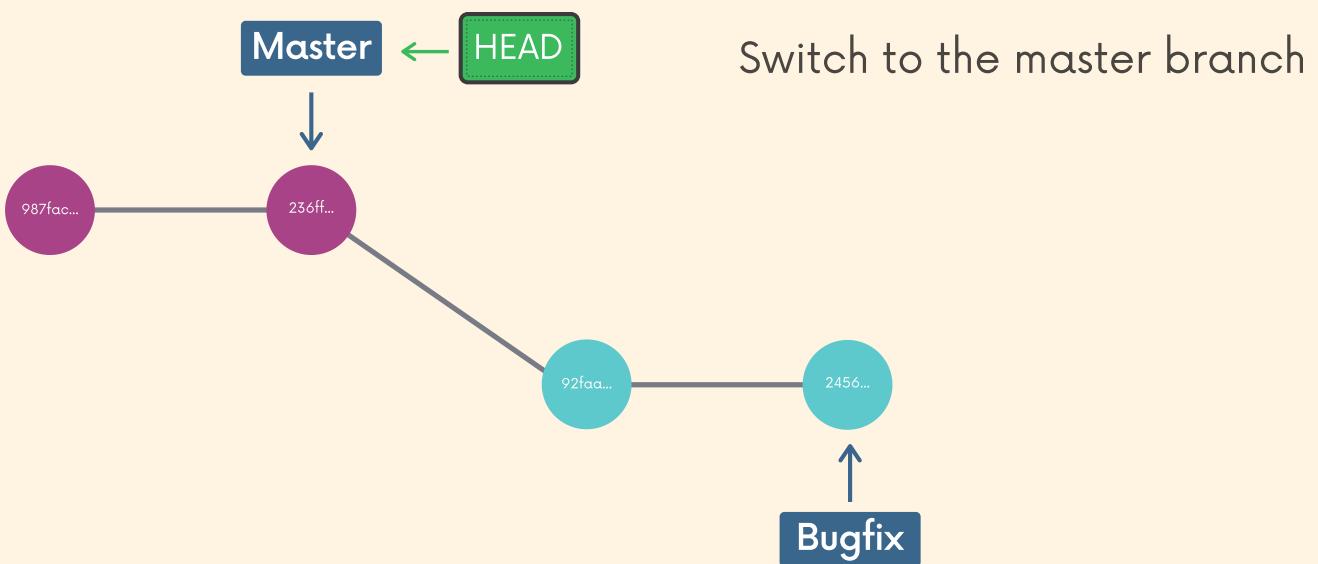
```
git switch master  
git merge bugfix
```



To merge the bugfix branch
into the master branch...



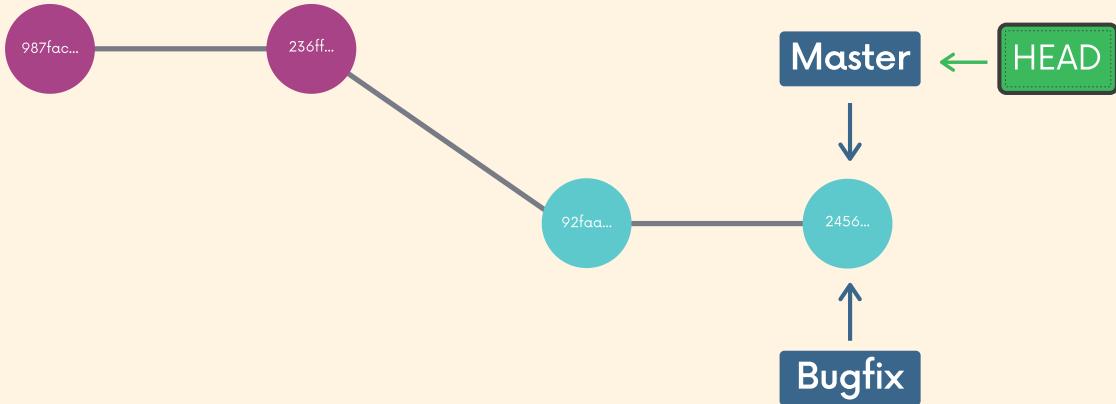
```
git switch master
```



Switch to the master branch

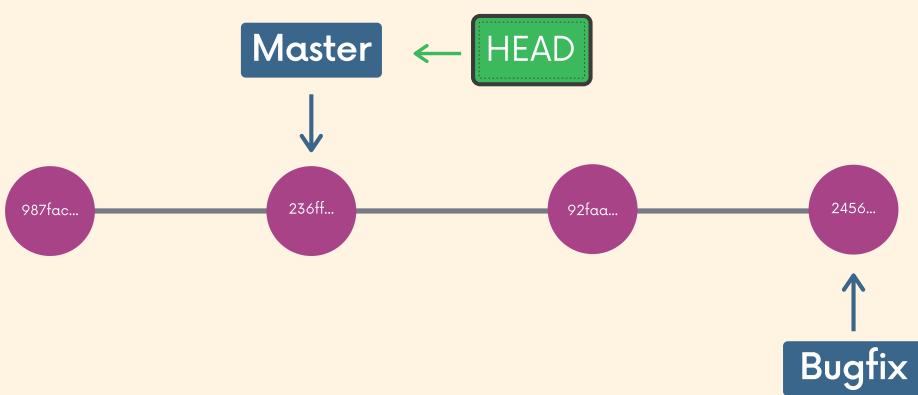
```
git merge bugfix
```

Merge bugfix into master



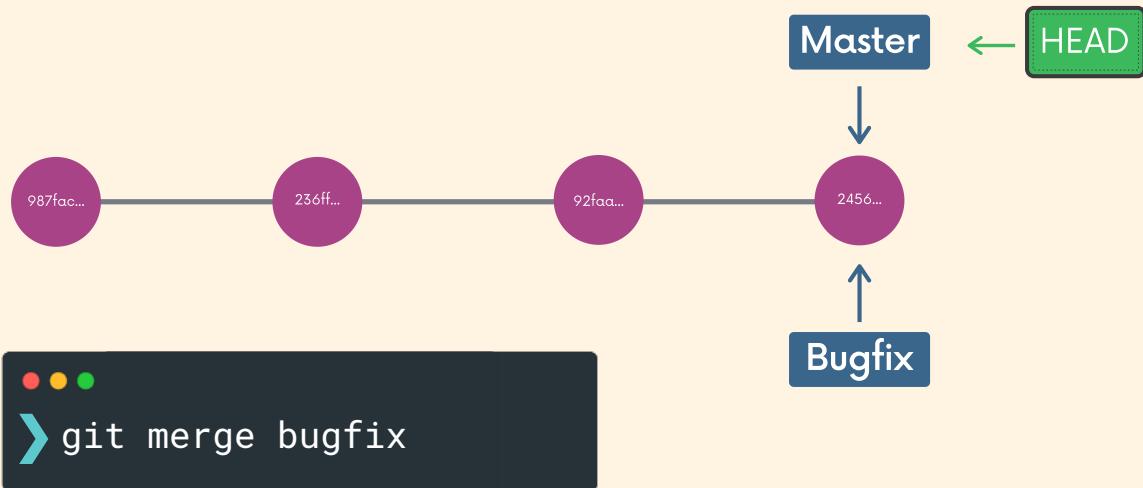
This is what it really looks like

Remember, branches are just defined by a branch pointer



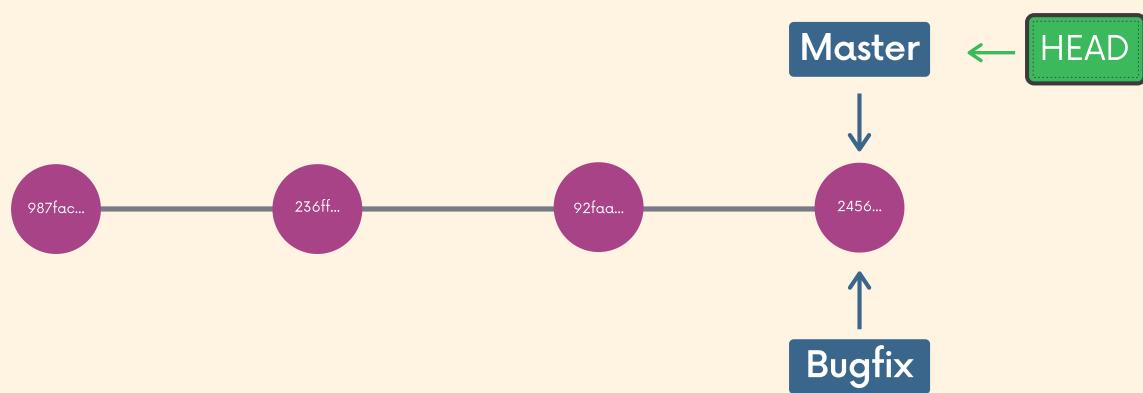
This is what it really looks like

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



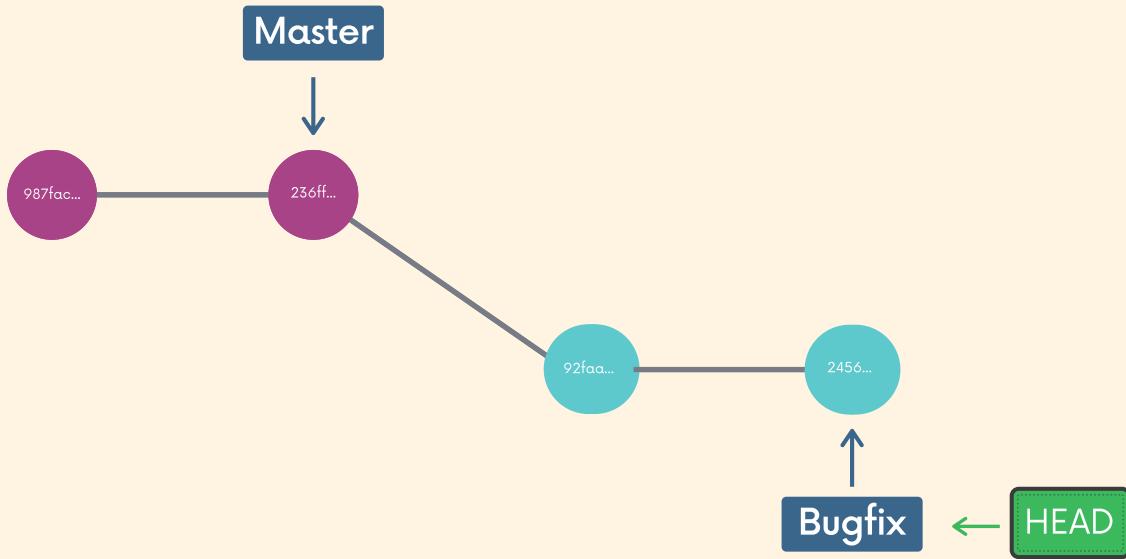
This Is Called A Fast-Forward

Master simply caught up on the commits from Bugfix



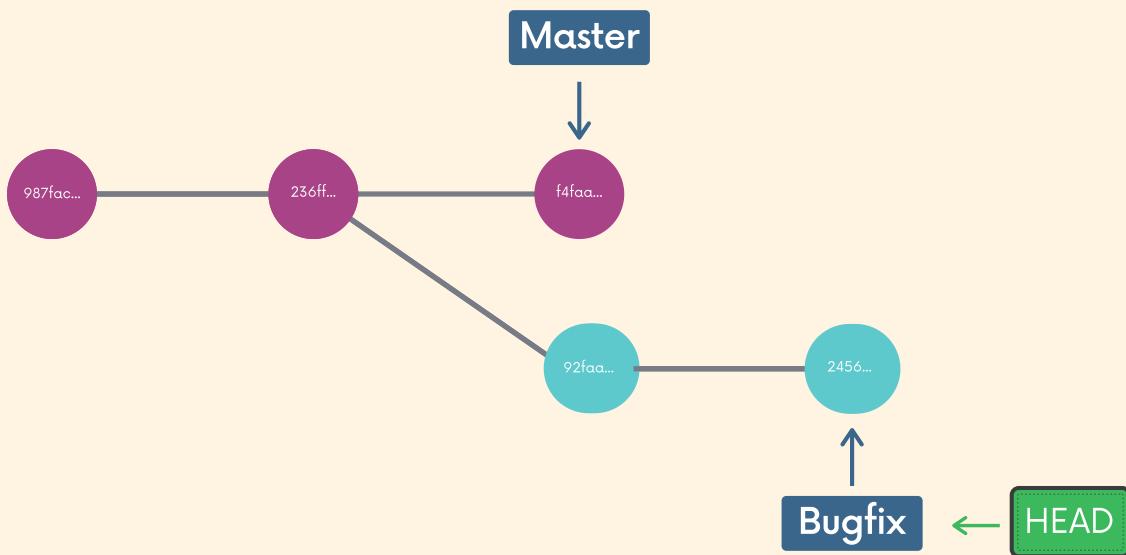
Not All Merges Are Fast Forwards!

We've branched off of master, just like before...



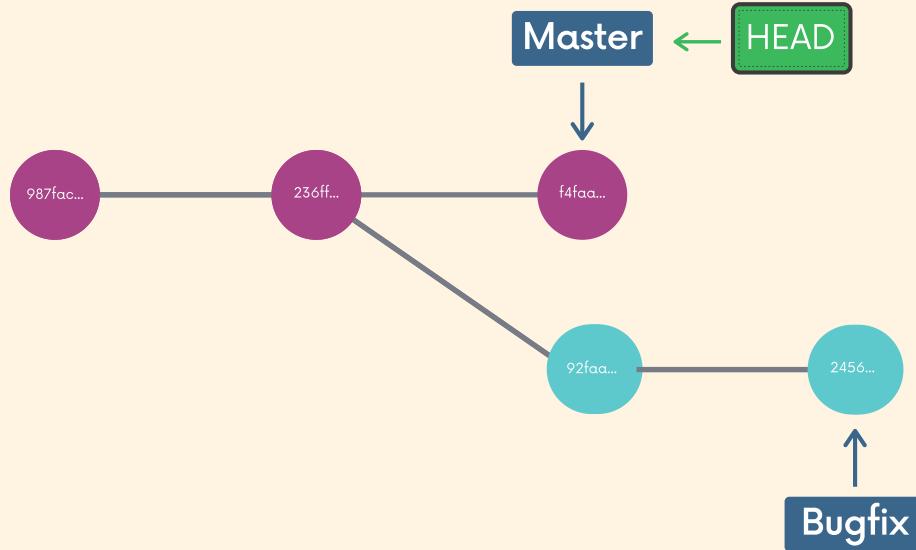
What if we add a commit on master?

This happens all the time! Imagine one of your teammates merged in a new feature or change to master while you were working on a branch



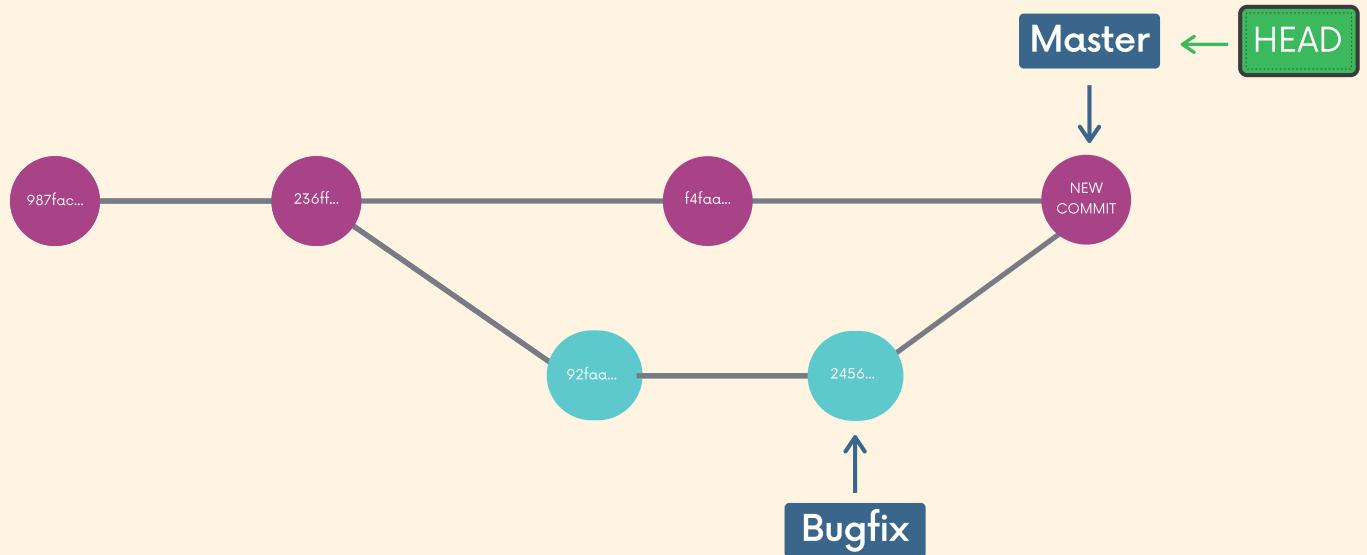
What happens when I try to merge?

Rather than performing a simple fast forward, git performs a "merge commit"
We end up with a new commit on the master branch.
Git will prompt you for a message.



What happens when I try to merge?

Rather than performing a simple fast forward, git performs a "merge commit"
We end up with a new commit on the master branch.
Git will prompt you for a message.



Heads Up!

Depending on the specific changes you are trying to merge, Git may not be able to automatically merge. This results in **merge conflicts**, which you need to manually resolve.



- 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 =====

<<<<< HEAD
I have 2 cats
I also have chickens
=====

I used to have a dog :(

>>>>> bug-fix

Conflict Markers

The content from the branch you are trying to merge from is displayed **between the ===== and >>>>> symbols.**



Resolving Conflicts

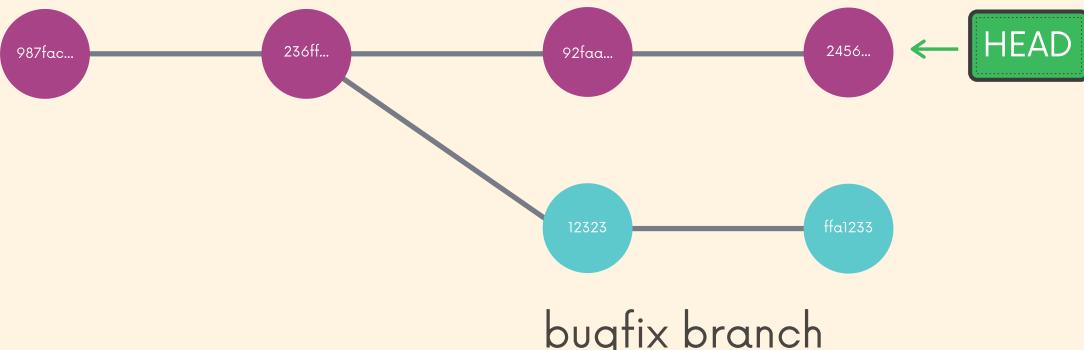
Whenever you encounter merge conflicts, follow these steps to resolve them:

1. Open up the file(s) with merge conflicts
2. Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.
3. Remove the conflict "markers" in the document
4. Add your changes and then make a commit!

Deleting A Branch

```
git branch -d bugfix
```

master branch



Deleting A Branch

master branch



7 Diff

Comparisons With Git Diff



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 how it has changed over time.





git diff

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



Compares Staging Area and Working Directory



git diff

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



What Does It Mean?



```
diff --git a/style/helpers.scss b/style/helpers.scss
index d530522..1a31a11 100644
--- a/style/helpers.scss
+++ b/style/helpers.scss
@@ -19,3 +19,14 @@
 @function pow($base, $exponent) {
   @return exponent($base, $exponent);
 }
+// Transition mixins
+@mixin transition($args...) {
+  -webkit-transition: $args;
+  -moz-transition: $args;
+}
+
+@mixin transition-property($args...) {
+  -webkit-transition-property: $args;
+  -moz-transition-property: $args;
+}
diff --git a/style/main.css b/style/main.css
index e95c4fe..8d875b1 100644
--- a/style/main.css
+++ b/style/main.css
@@ -6 +22,10 @@
 h1 {
   width: 50px;
   position: relative;
   padding: 15px;
   cursor: default;
+  -webkit-touch-callout: none;
+  -webkit-user-select: none;
+  -moz-user-select: none;
  background: #bbada0;
  border-radius: 6px;
  width: 50px;
```

Last Commit

red
orange
yellow
green
blue
purple

rainbow.txt

New Changes

red
orange
yellow
green
blue
indigo
violet

rainbow.txt

Git Diff Output

```
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

+indigo
+violet
```

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

+indigo
+violet
```

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/style/helpers.scss b/style/helpers.scss
index d530522..1a31a11 100644
--- a/style/helpers.scss
+++ b/style/helpers.scss
@@ -19,3 +19,14 @@
 @function pow($base, $exponent) {
   @return exponent($base, $exponent);
 }
+
+// Transition mixins
+@mixin transition($args...) {
+  -webkit-transition: $args;
+  -moz-transition: $args;
+}
+
+@mixin transition-property($args...) {
+  -webkit-transition-property: $args;
+  -moz-transition-property: $args;
+}

diff --git a/style/main.css b/style/main.css
index e05c4fe..8d875b1 100644
--- a/style/main.css
+++ b/style/main.css
@@ -22,6 +22,10 @@ h1 {
  margin-top: 50px;
  position: relative;
  padding: 15px;
+
+ cursor: default;
+ -webkit-touch-callout: none;
+ -webkit-user-select: none;
+ -moz-user-select: none;
  background: #bbada0;
  border-radius: 6px;
  width: 500px;
```

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

```

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

```

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/style/main.css b/style/main.css
index e05c4fe..8d875b1 100644
--- a/style/main.css
+++ b/style/main.css
@@ -22,6 +22,10 @@ h1 {
    margin-top: 50px;
    position: relative;
    padding: 15px;
+   cursor: default;
+   -webkit-touch-callout: none;
+   -webkit-user-select: none;
+   -moz-user-select: none;
    background: #bbada0;
    border-radius: 6px;
    width: 500px;
@@ -63,8 +67,12 @@ h1 {
    background: #eee4da;
    text-align: center;
    line-height: 116.25px;
-   font-size: 60px;
-   font-weight: bold; }
+   font-size: 55px;
+   font-weight: bold;
+   -webkit-transition: 200ms ease;
+   -moz-transition: 200ms ease;
+   -webkit-transition-property: top, left;
+   -moz-transition-property: top, left; }
.tile.tile-position-1-1 {
    position: absolute;
    left: 0px;
```

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
```

file a

@@ -3,4 +3,5 @@

file b

red
orange
yellow
green
blue
purple

red
orange
yellow
green
blue
indigo
violet

Changes

Every line that changed between the two files is marked with either a + or - symbol.

lines that begin with - come from file A

lines that begin with + come from file 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

Changes

Every line that changed between the two files is marked with either a + or - symbol.

lines that begin with - come from file A

lines that begin with + come from file B

```
diff --git a/style/main.css b/style/main.css
index e05c4fe..8d875b1 100644
--- a/style/main.css
+++ b/style/main.css
@@ -22,6 +22,10 @@ h1 {
    margin-top: 50px;
    position: relative;
    padding: 15px;
+   cursor: default;
+   -webkit-touch-callout: none;
+   -webkit-user-select: none;
+   -moz-user-select: none;
    background: #bbada0;
    border-radius: 6px;
    width: 500px;
@@ -63,8 +67,12 @@ h1 {
    background: #eee4da;
    text-align: center;
    line-height: 116.25px;
-   font-size: 60px;
-   font-weight: bold; }
+   font-size: 55px;
+   font-weight: bold;
+   -webkit-transition: 200ms ease;
+   -moz-transition: 200ms ease;
+   -webkit-transition-property: top, left;
+   -moz-transition-property: top, left; }
.tile.tile-position-1-1 {
    position: absolute;
    left: 0px;
```



git diff

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]
```



Comparing Branches

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

```
git diff branch1..branch2
```





Comparing Commits

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



```
git diff commit1..commit2
```



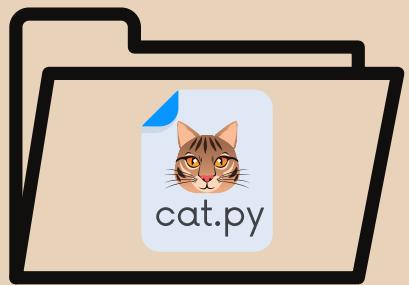
8 Stashing



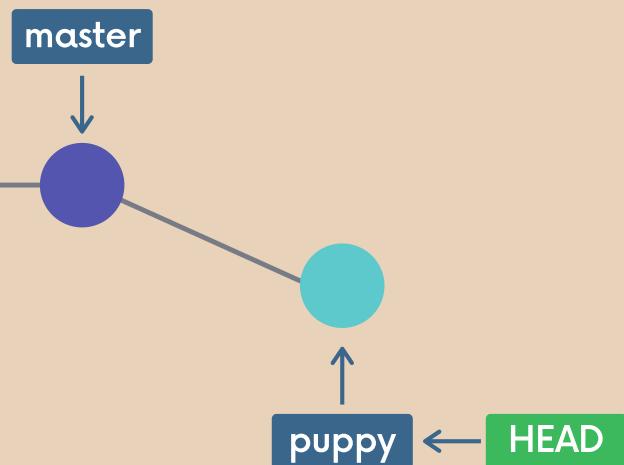
Git Stashing



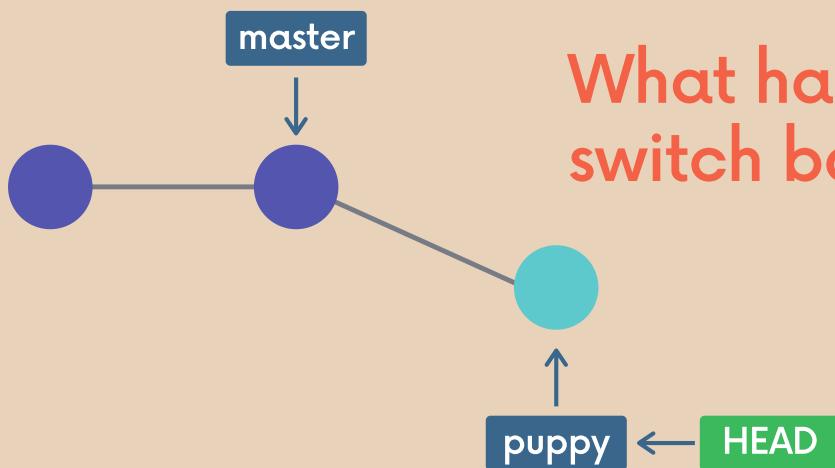
I'm on master



I make a new branch and switch to it

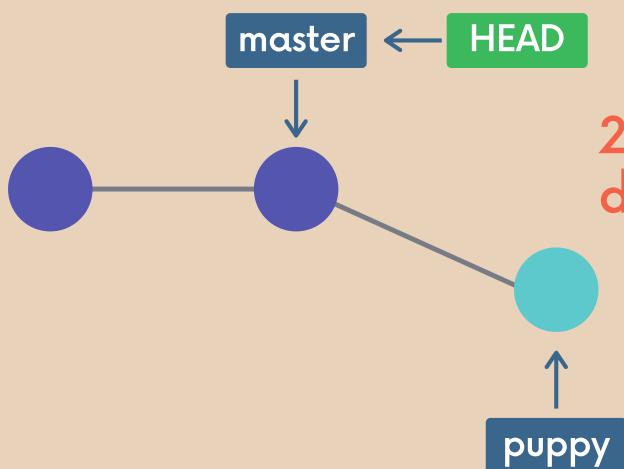
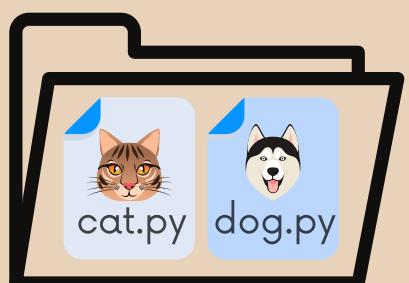


I do some new work, but don't make any commits



What happens when I switch back to master?

1. My changes come with me to the destination branch



2. Git won't let me switch if it detects potential conflicts



Stashing

Git provides an easy way of stashing these 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 and then come back to them later.

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

```
❯ git stash
```

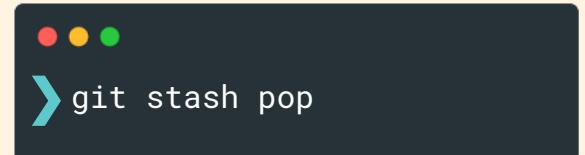
You can also use `git stash save` instead





Stashing

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



Branch: bugfix

Working Directory

modified nav.css

modified nav.js

Staging Area

modified index.js

created footer.js

Repository

0026739

bb43f1f

I'm working away on fixing a bug,
but then...



Branch: bugfix

Working Directory

modified nav.css

modified nav.js

Staging Area

modified index.js

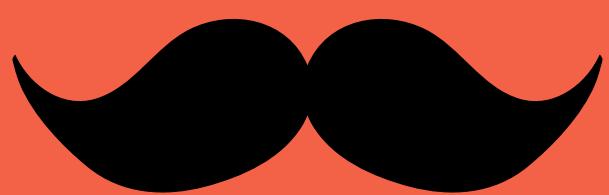
created footer.js

Repository

0026739

bb43f1f

I'm not at all ready to commit these changes, but I also don't want to take them with me to **master**...

I Can 
Stash Them!

Branch: bugfix

git stash

Working Directory

Staging Area

Repository

0026739

bb43f1f

All of my uncommitted
changes have been
stashed away!

The Stash



Branch: bugfix

git stash

Working Directory

Staging Area

Repository

0026739

bb43f1f

All of my uncommitted
changes have been
stashed away!

The Stash



Stashing

Now that I have stashed my changes, I can switch branches, create new commits, etc.

I head over to master and take a look at my coworker's changes.

When I'm done, I can re-apply the changes I stashed away at any point

Branch: bugfix

git stash pop

Working Directory

modified nav.css

modified nav.js

Staging Area

modified index.js

created footer.js

Repository

0026739

bb43f1f

The Stash

My stashed changes are restored!



Stash Apply

You can use `git stash apply` to apply whatever is stashed away, without removing it from the stash. This can be useful if you want to apply stashed changes to multiple branches.

☰
git stash apply



Branch: bugfix

☰
git stash apply

Working Directory

modified nav.css

modified nav.js

Staging Area

modified index.js

created footer.js

Repository

0026739

bb43f1f

My stashed changes are restored!

The Stash





Heads Up!

If you have untracked files (files that you have never checked in to Git), they will not be included in the stash.

Fortunately, you can use the `-u` option to tell git stash to include those untracked files.



```
git stash -u
```



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
```



Viewing Stashes

run `git stash list` to view all stashes



`>git stash list`

```
stash@{0}: WIP on master: 049d078 Create index file  
stash@{1}: WIP on master: c264051 Revert "Add file_size"  
stash@{2}: WIP on master: 21d80a5 Add number to log
```



Applying Specific Stashes

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



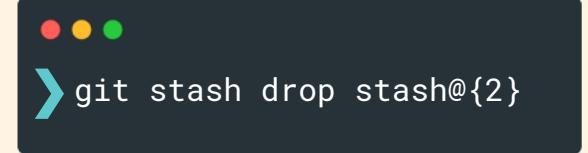
`>git stash apply stash@{2}`





Dropping Stashes

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

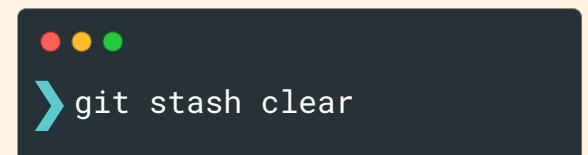


```
git stash drop stash@{2}
```



Clearing The Stash

To clear out all stashes, run `git stash clear`



```
git stash clear
```



Do I Really Need This?

99% of the time, I just use `git stash` and `git stash pop`.

Just my personal experience, YMMV!



```
● ● ●  
❯ git stash  
      sometime later on...  
❯ git stash pop
```



9 Undoing

stuff &

Time

traveling

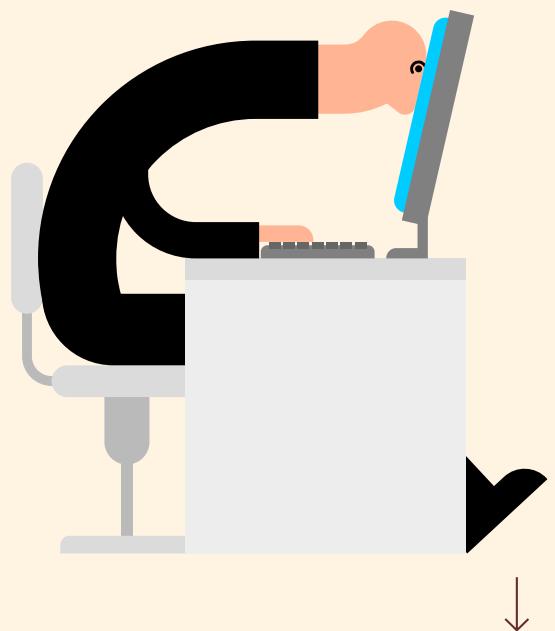
Undoing Stuff & Time Traveling



Checkout

The **git checkout** command is like a Git Swiss Army knife. Many developers think it is overloaded, which is what lead to the addition of the **git switch** and **git restore** commands

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





Checkout

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 digits of a commit hash.

Don't panic when you see the following message...

```
git checkout d8194d6
```



Detached HEAD??

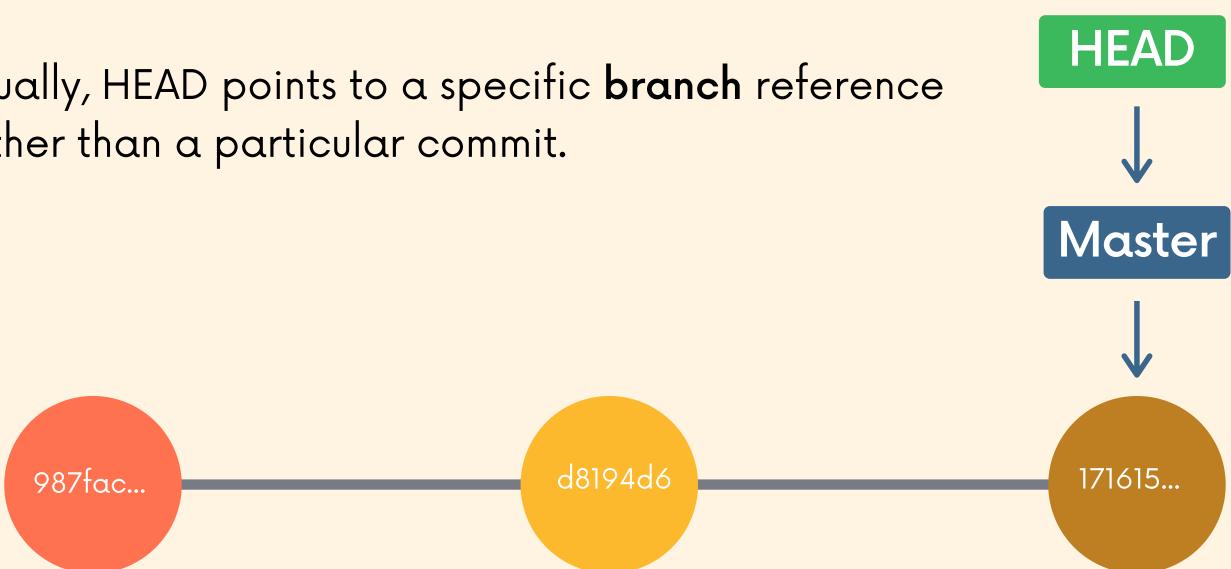
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.



What On Earth Is Going On??



Usually, HEAD points to a specific **branch** reference rather than a particular commit.



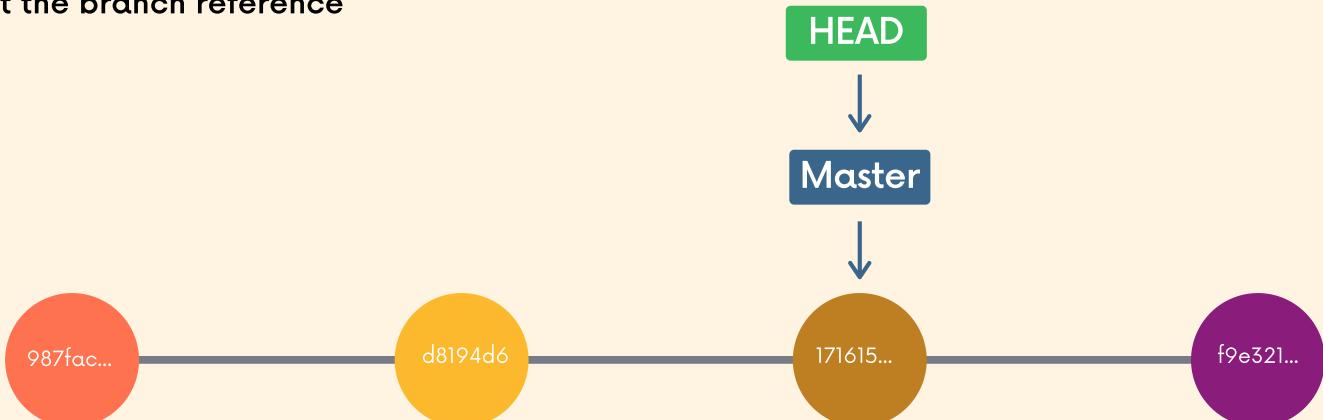
How It Works

- HEAD is a pointer to the current branch reference
- The branch reference is a pointer to the last commit made on a particular branch



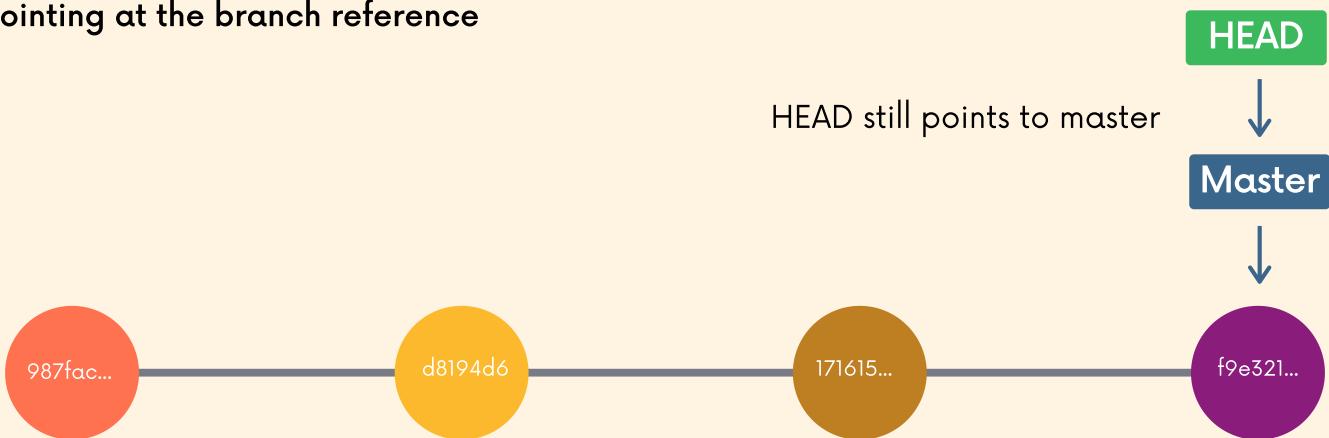
When we make a new commit, the branch reference is updated to reflect the new commit.

The HEAD remains the same, because it's pointing at the branch reference



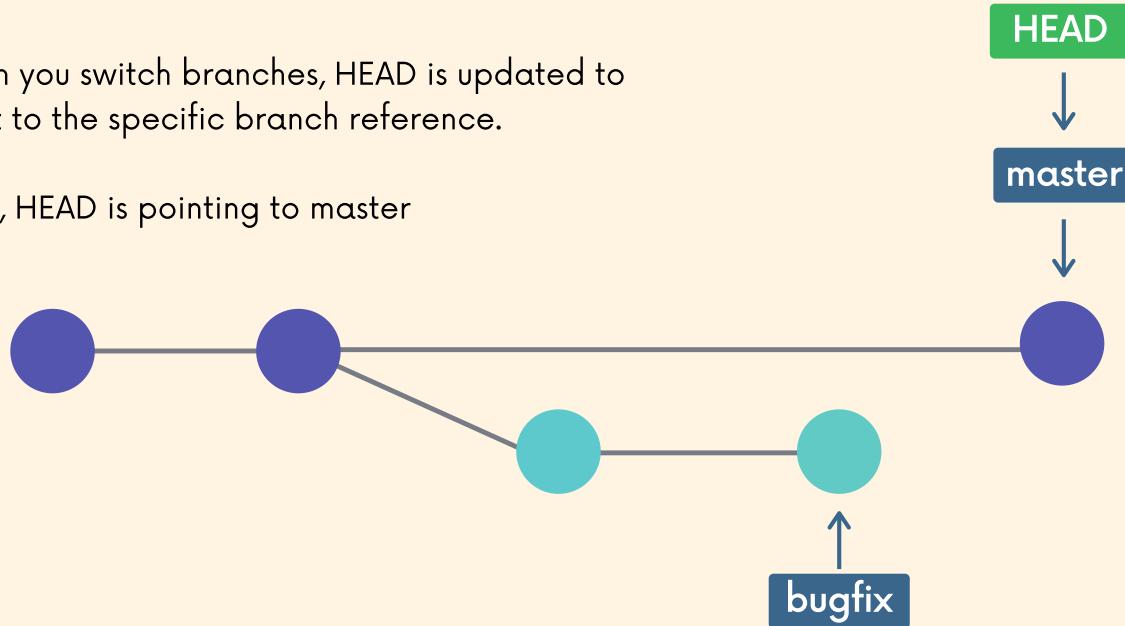
When we make a new commit, the branch pointer is updated to reflect the new commit.

The HEAD remains the same, because it's pointing at the branch reference

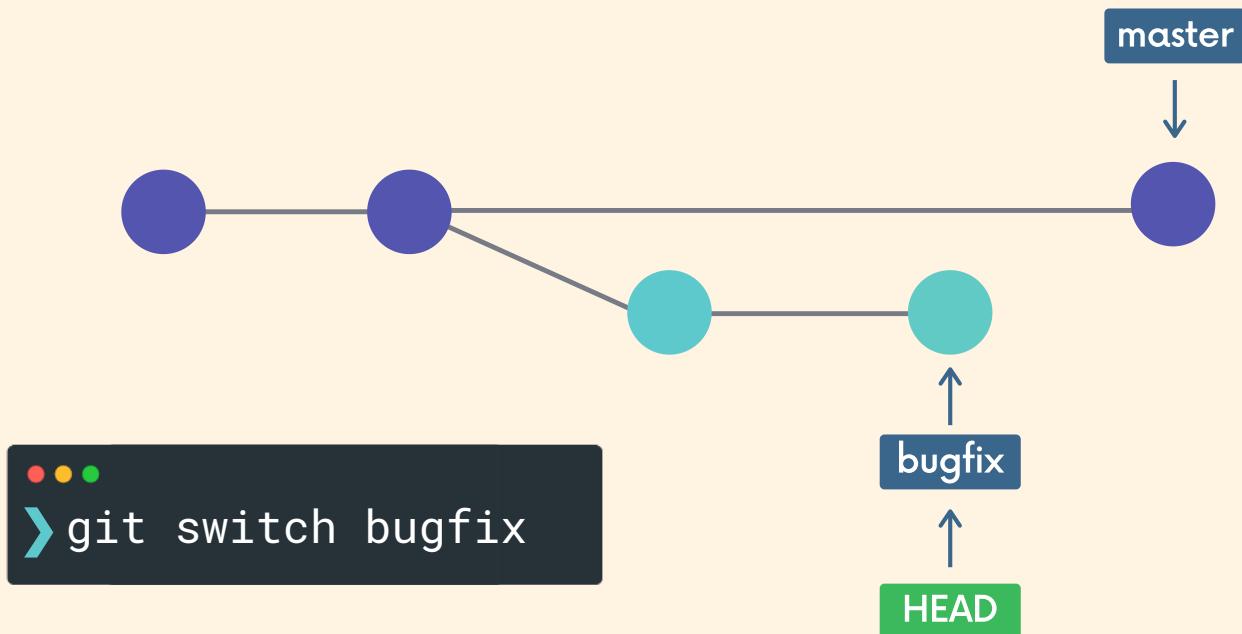


When you switch branches, HEAD is updated to point to the specific branch reference.

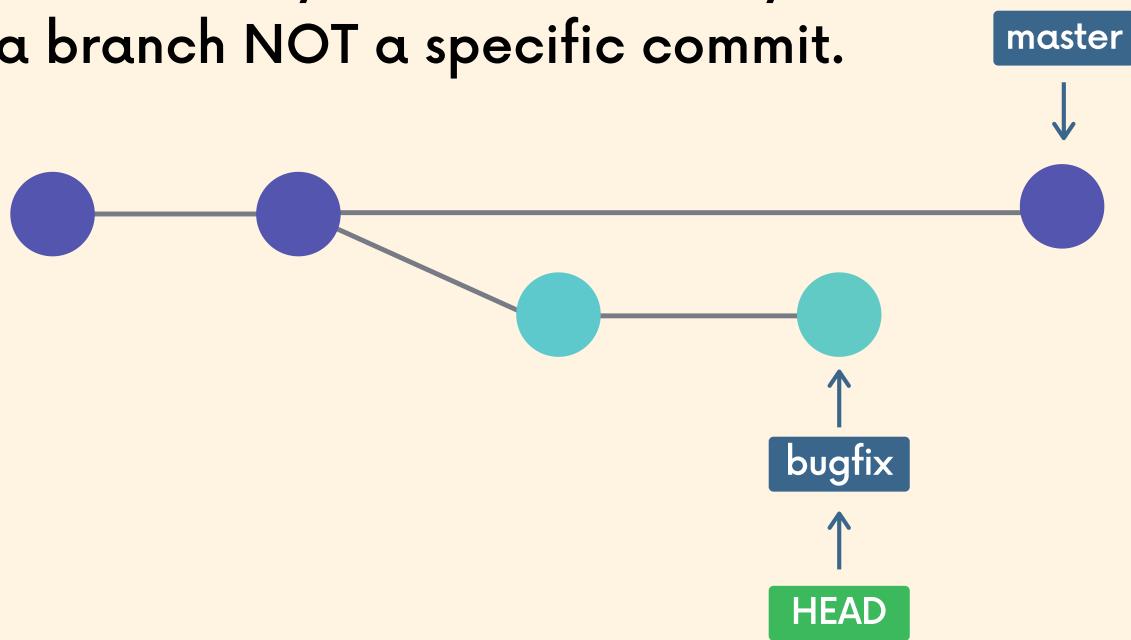
Here, HEAD is pointing to master



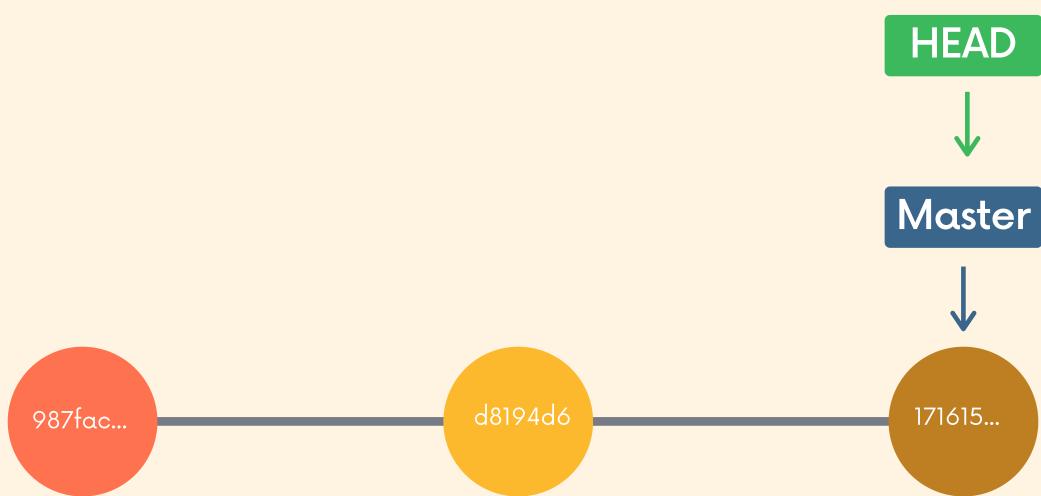
If we switch to the bugfix branch, HEAD is now pointing at the bugfix reference.



This is all to say that HEAD usually refers to a branch NOT a specific commit.

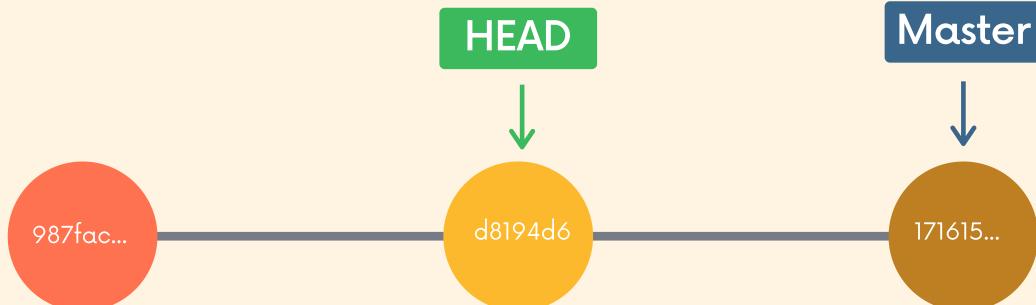


Back to this Detached HEAD thing



When we checkout a particular commit,
HEAD points at that commit rather than
at the branch pointer.

```
git checkout d8194d6
```



DETACHED HEAD!



Detached HEAD

Don't panic when this happens! It's not a bad thing!

You have a couple options:

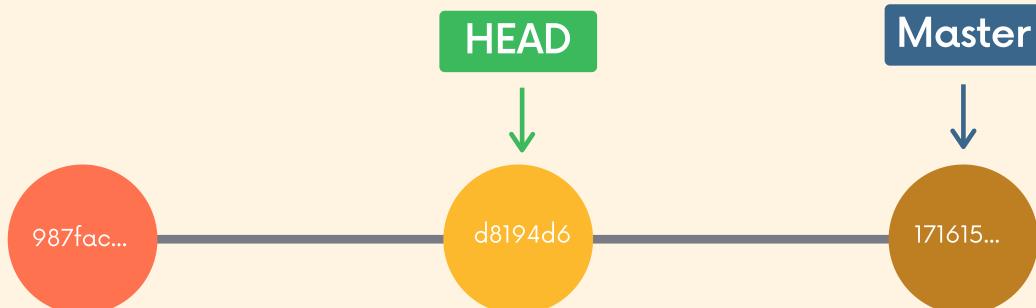
1. Stay in detached HEAD to examine the contents of the old commit. Poke around, view the files, etc.
2. Leave and go back to wherever you were before - reattach the HEAD
3. Create a new branch and switch to it. You can now make and save changes, since HEAD is no longer detached.

```
git checkout <commit-hash>
```



If you checkout an old commit and decide you want to return to where you were before....

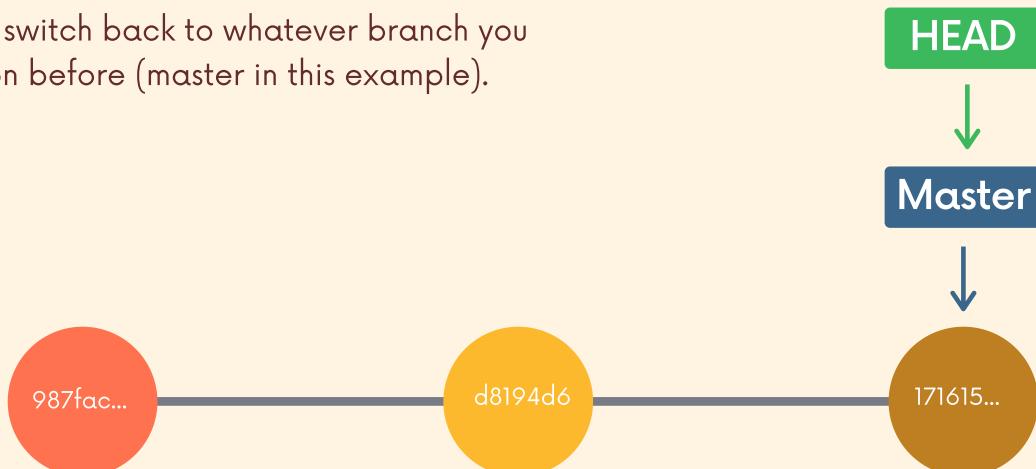
```
git checkout d8194d6
```



DETACHED HEAD!

```
git switch master
```

Simply switch back to whatever branch you were on before (master in this example).

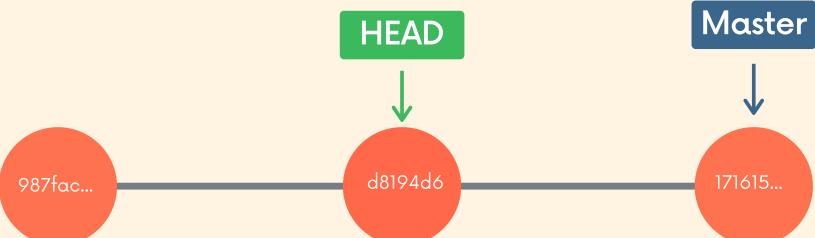


RE-ATTACHED HEAD!

Suppose you want to go back to an old commit and make some new changes



```
git checkout d8194d6
```

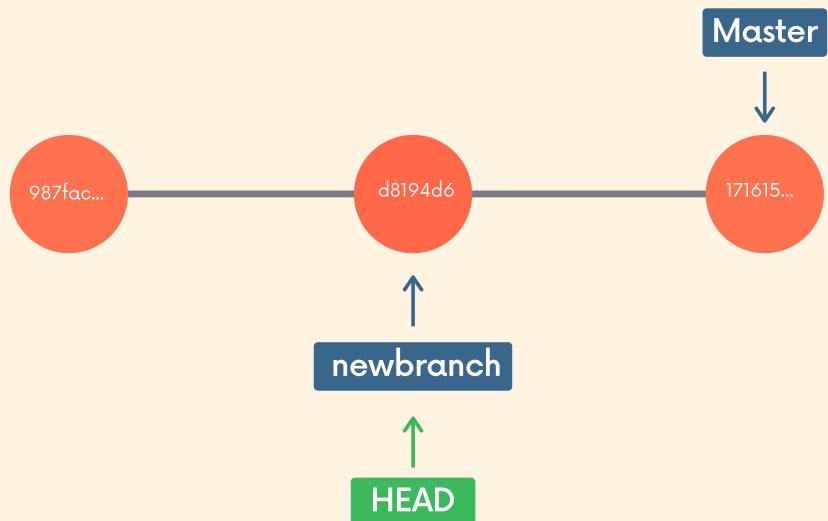


Checkout the old commit.
Now in detached HEAD state.

```
git switch -c newbranch
```

While in detached HEAD,
Make a new branch and switch to it.

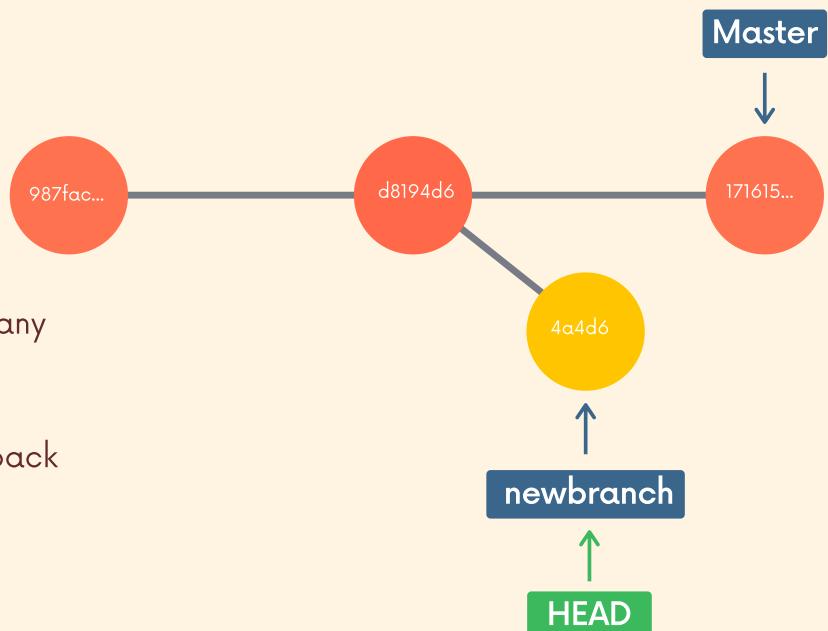
Head is now back to pointing at a
branch reference!



```
git add .  
git commit -m "new commit"
```

Now on the new branch, make as many
new commits as you want!

It's like you time traveled! We went back
to an old commit and made a new
branch based on it.

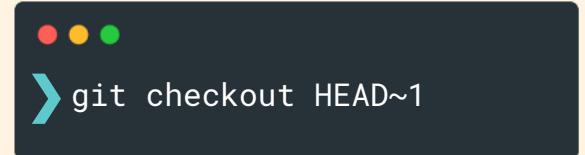


Checkout

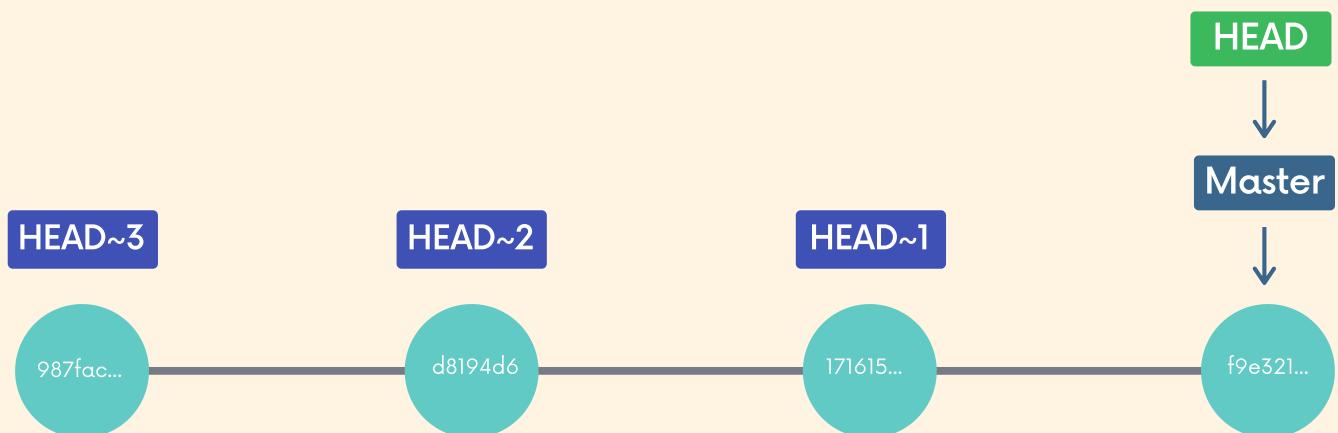
`git checkout` supports a slightly odd syntax for referencing previous commits relative to a particular commit.

HEAD~1 refers to the commit before HEAD (parent)
HEAD~2 refers to 2 commits before HEAD (grandparent)

This is not essential, but I wanted to mention it because it's quite weird looking if you've never seen it.



```
git checkout HEAD~1
```

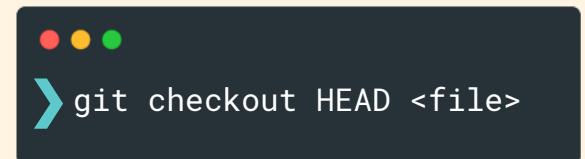




Discarding Changes

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 that file, reverting back to the HEAD.



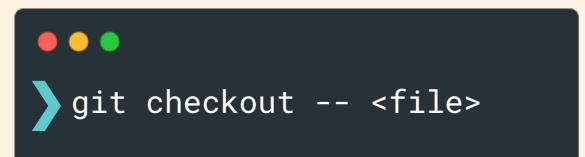
```
git checkout HEAD <file>
```



Another Option

Here's another shorter option to revert a file...

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



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



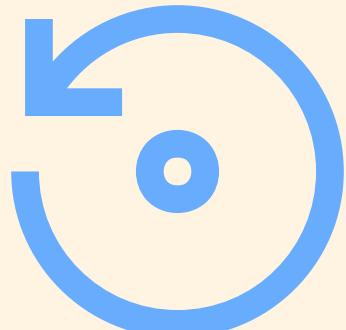


Restore

`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 alternatives to some of the uses for `checkout`.



Unmodifying Files with Restore

Suppose you've made some changes to a file since your last commit. You've saved the file but then realize 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!





Unmodifying Files with Restore

`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~1 app.js
```



Unstaging Files with Restore

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 app.js`



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





Feeling Confused?

git status reminds you what to use!

```
GitDemo > git status                               timetravel
On branch timetravel
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   cat2.txt

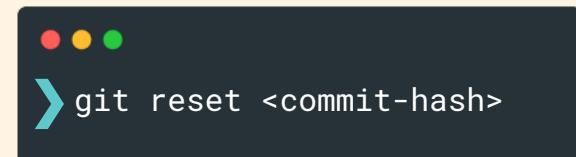
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   cat2.txt
```



Git Reset

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



OH NO! I didn't mean to make that commit here!

Working Directory

Staging Area

Repository

modified about.html

created lisa.jpg

39c3fd

0026739

bb43f1f

Working Directory

modified about.html

created lisa.jpg

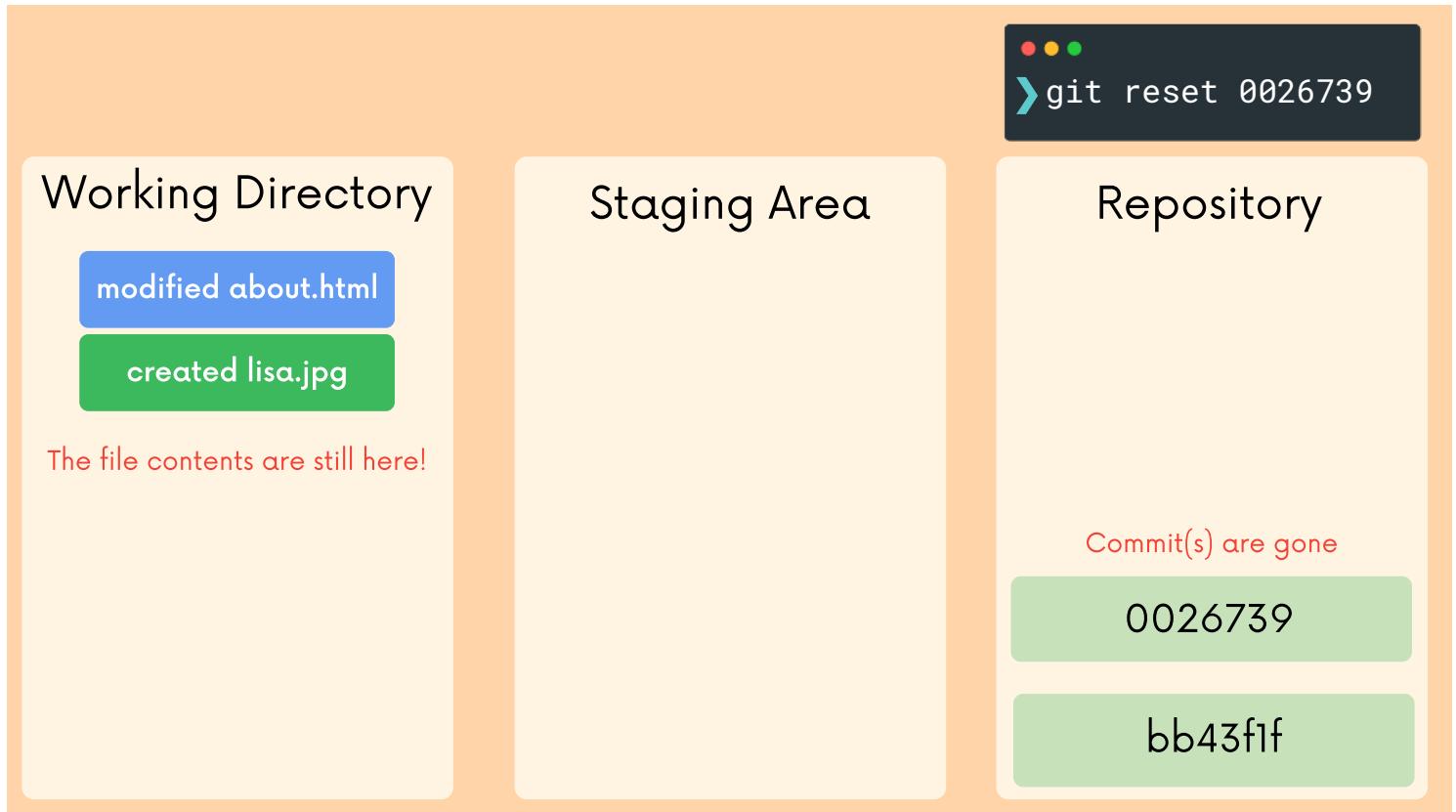
Staging Area

```
git reset 0026739
```

Repository

0026739

bb43f1f



≡

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>



OH NO! I don't want that commit OR the changes

Working Directory

Staging Area

Repository

modified about.html

created lisa.jpg

39c3fd

0026739

bb43f1f

```
git reset --hard 0026739
```

Working Directory

The changes in the file(s)
are gone too!

Staging Area

Repository

Commit(s) are gone

0026739

bb43f1f

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 they both "undo" changes, but they accomplish it in different ways.

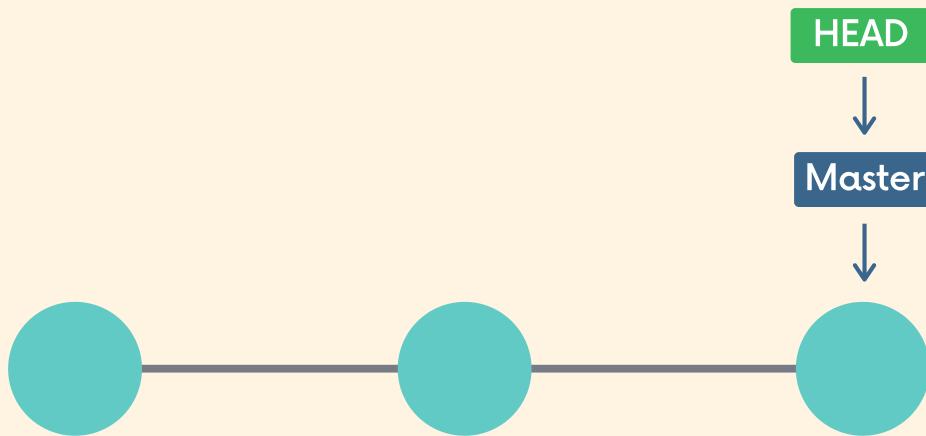
git reset actually moves the branch pointer backwards, eliminating commits.

git revert instead creates a brand 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.

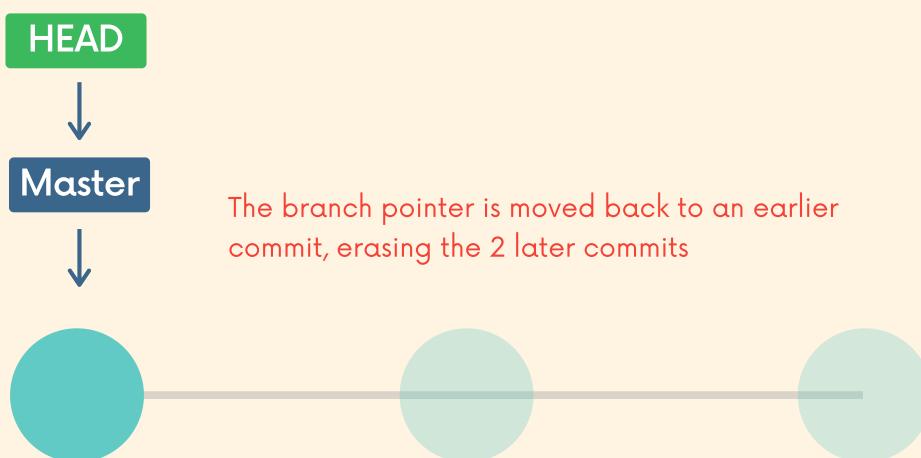
```
❯ git revert <commit-hash>
```



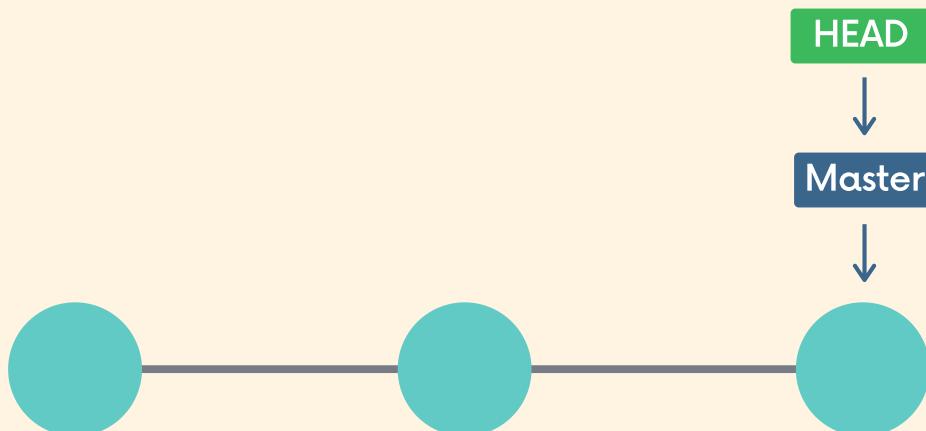
"Undoing" With Reset



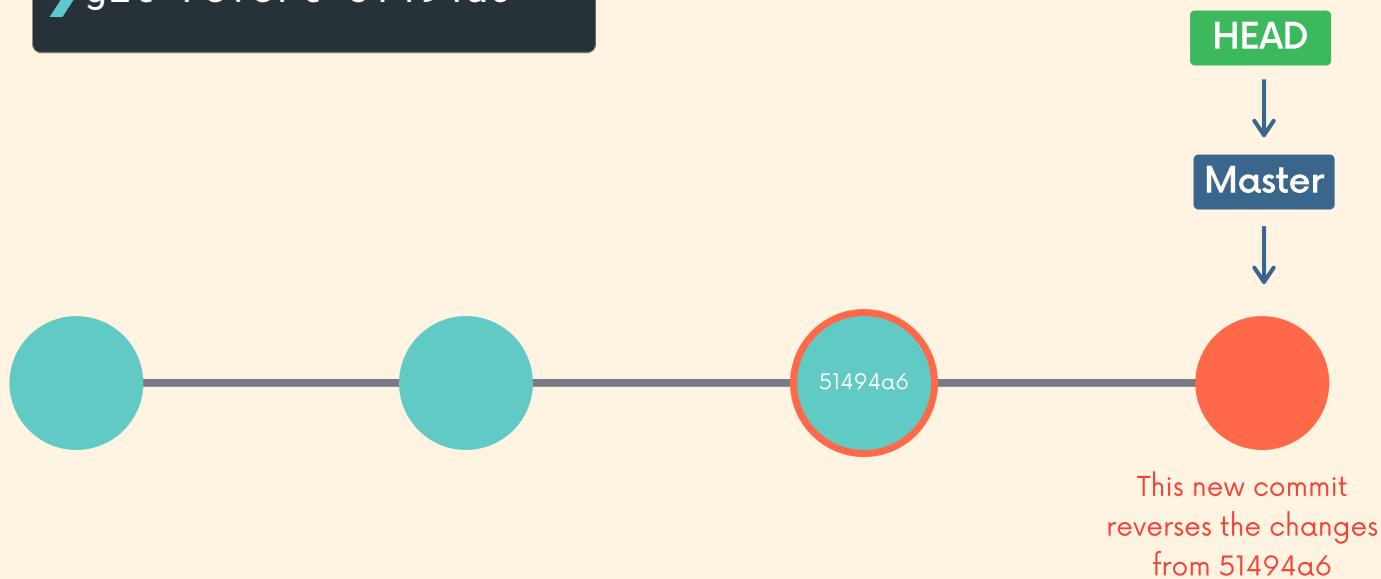
```
❯ git reset HEAD~2
```



"Undoing" With Revert



```
❯ git revert 51494a6
```



≡

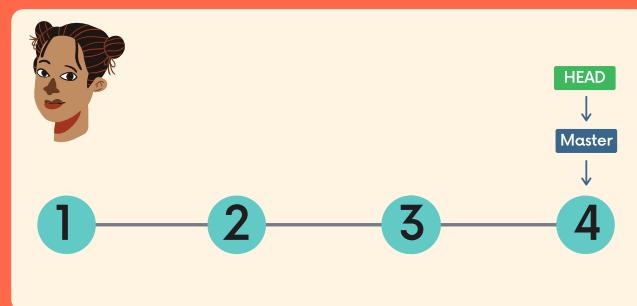
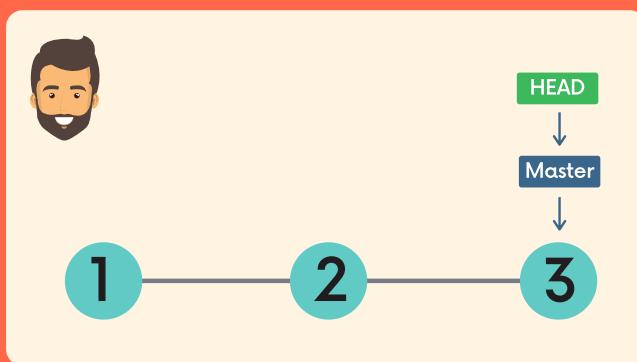
Which One Should I Use?



Both `git reset` and `git revert` help us reverse changes, but there is a significant difference when it comes to collaboration (which we have yet to discuss but is coming up soon!)

If you want to reverse some commits that other people already have on their machines, you should use revert.

If you want to reverse commits that you haven't shared with others, use reset and no one will ever know!





1

2

3



1

2

3

4

My Changes

I use `git reset` to remove commits that I already shared with my team!



???



1

2

3



???



1

2

3

4

My Changes

This makes their lives harder. I altered history that they already have. BAD!



HEAD
↓
Master
↓

1

2

3



HEAD
↓
Master
↓

1

2

3

4

My Local Repo

HEAD
↓
Master
↓

Let's try again...



HEAD
↓
Master
↓

1

2

3



HEAD
↓
Master
↓

1

2

3

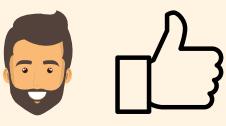
4

My Changes

HEAD
↓
Master
↓

1 2 3 4

I use **git revert** to reverse the same commits as before, by ADDING a new commit to the chain



HEAD
↓
Master
↓



HEAD
↓
Master
↓



My Changes

HEAD
↓
Master
↓



My team can merge in the new
"undo" commit without issue.
I didn't alter history.

10 GitHub basics

Github Basics

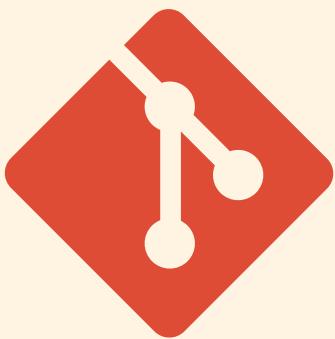


What Is Github?

Github is a hosting 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.

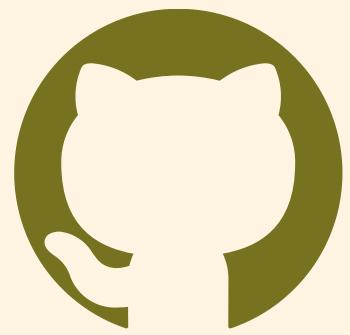
Beyond hosting repos, Github also provides additional collaboration features that are not native to Git (but are super useful). Basically, Github helps people share and collaborate on repos.





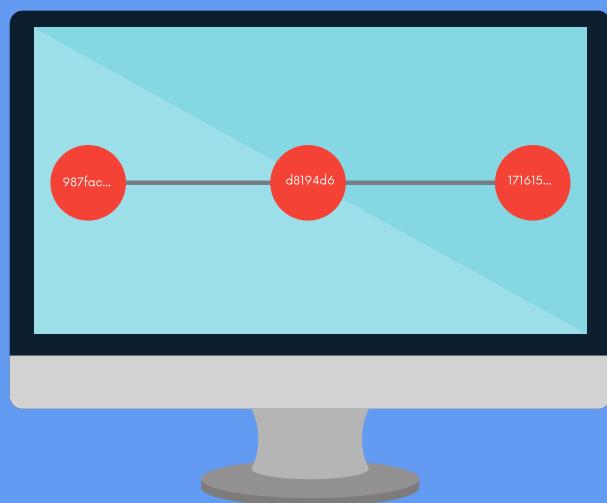
Git

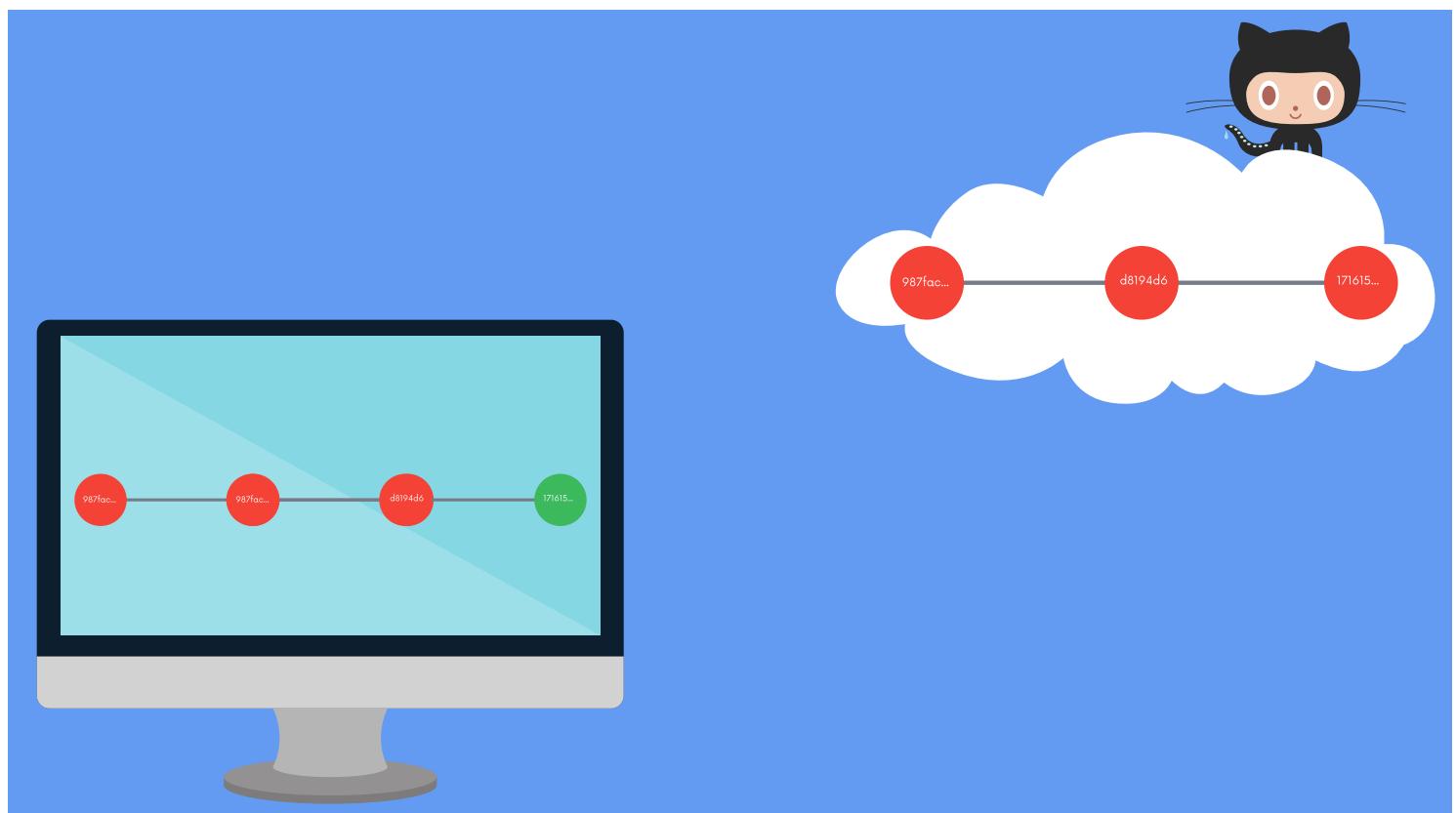
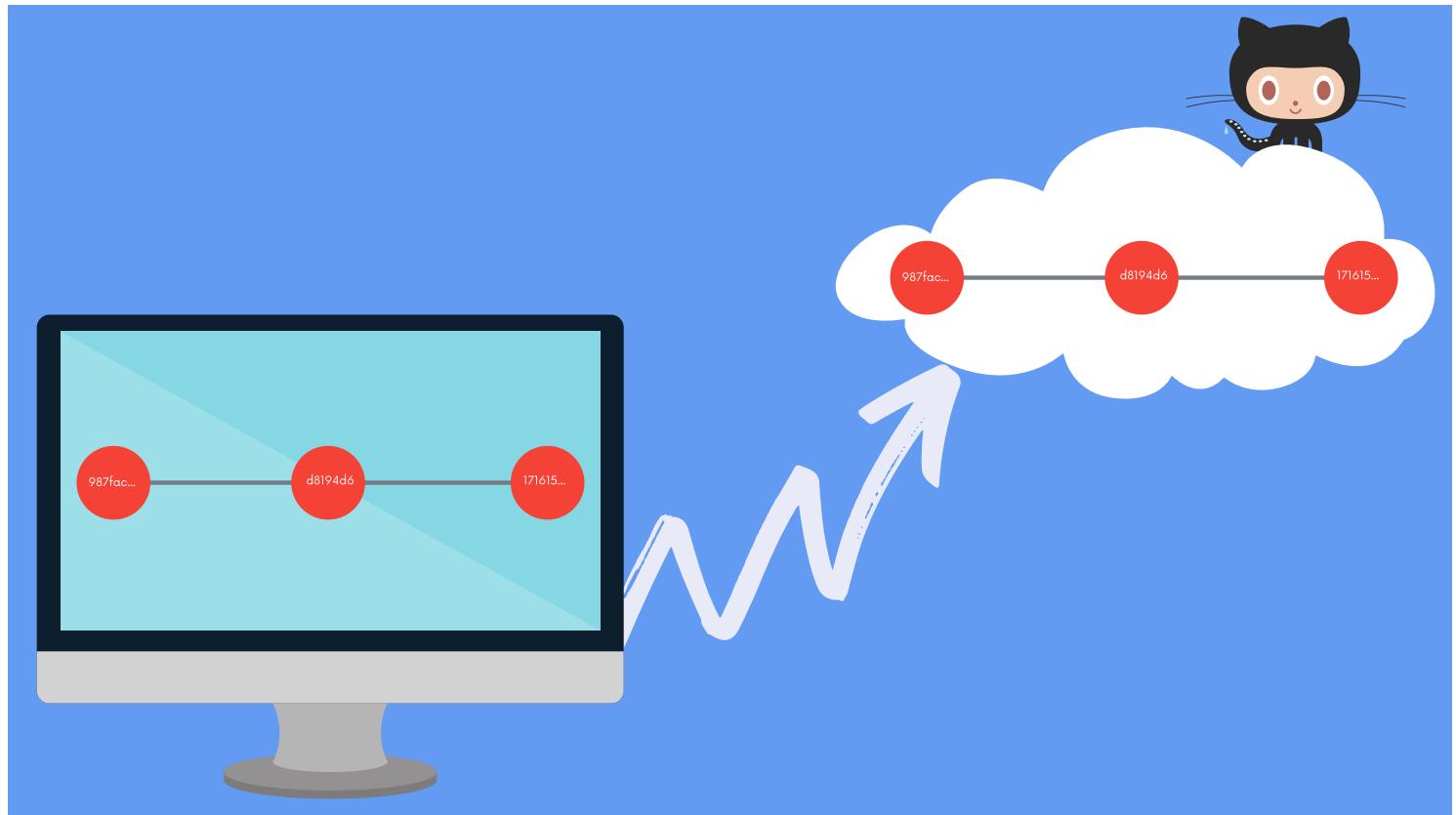
Git is the version control software that runs locally on your machine. You don't need to register for an account. You don't need the internet to use it. You can use Git without ever touching Github.

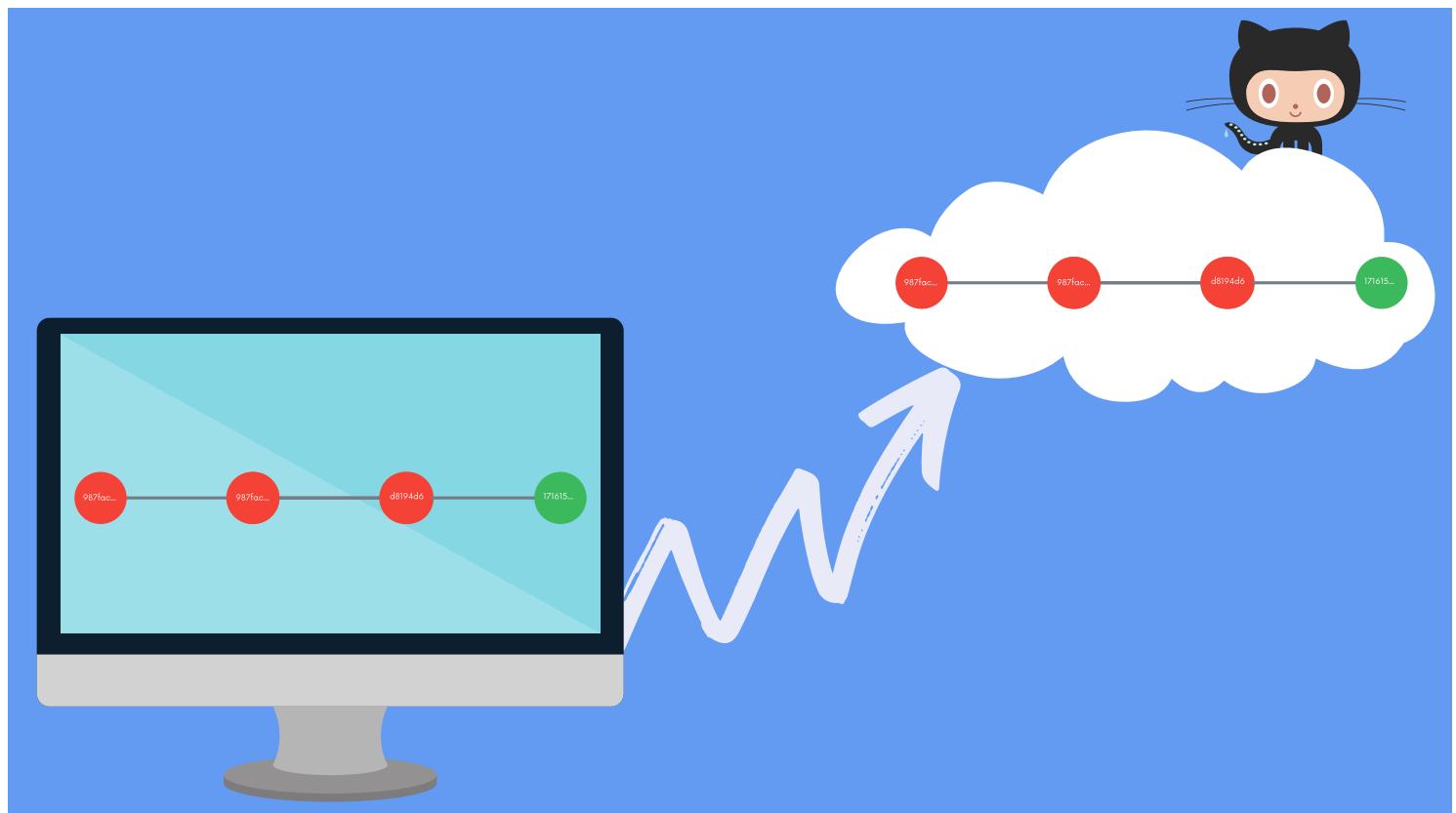
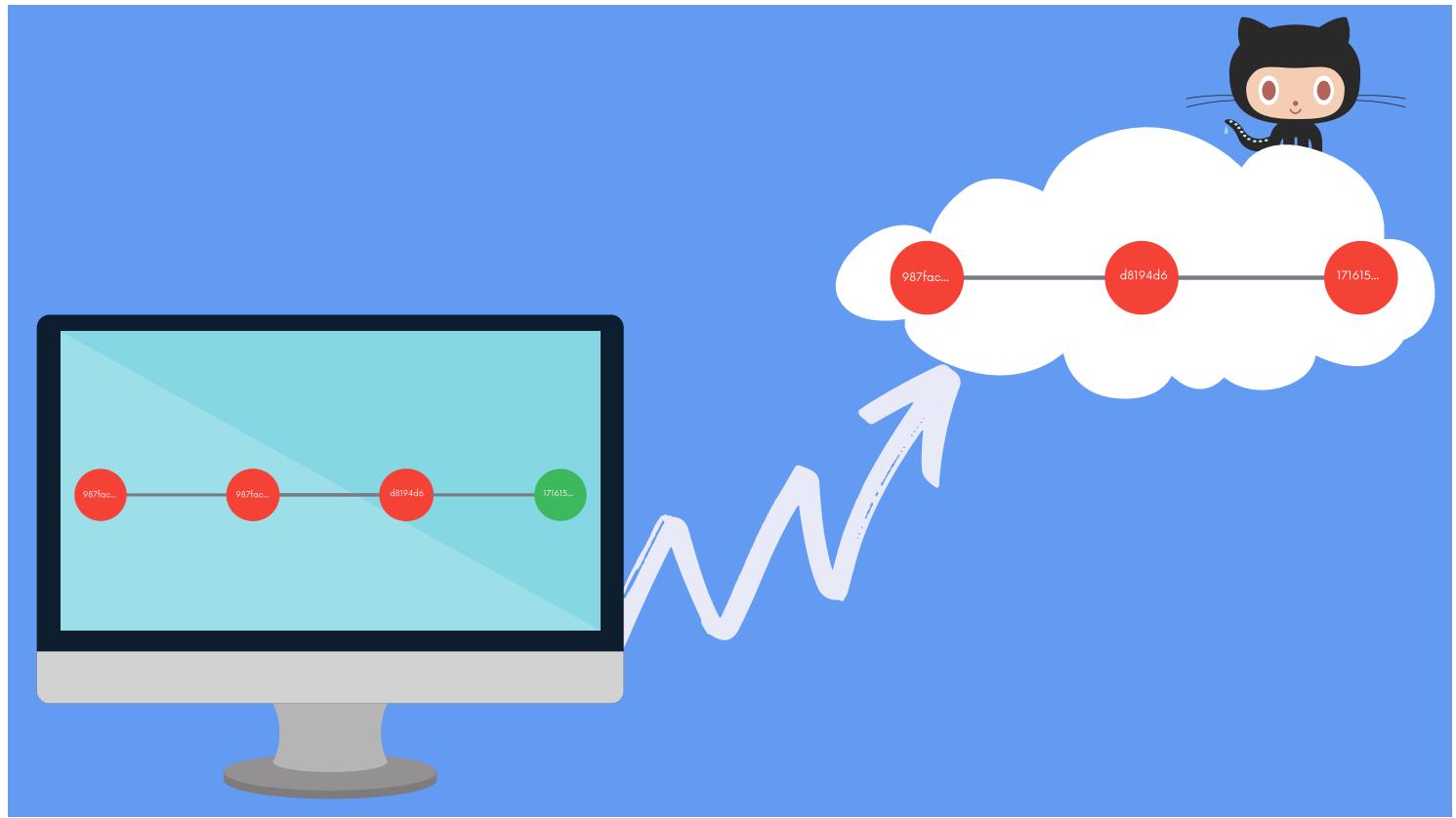


Github

Github is a service that hosts Git repositories in the cloud and makes it easier to collaborate with other people. You do need to sign up for an account to use Github. It's an online place to share work that is done using Git.

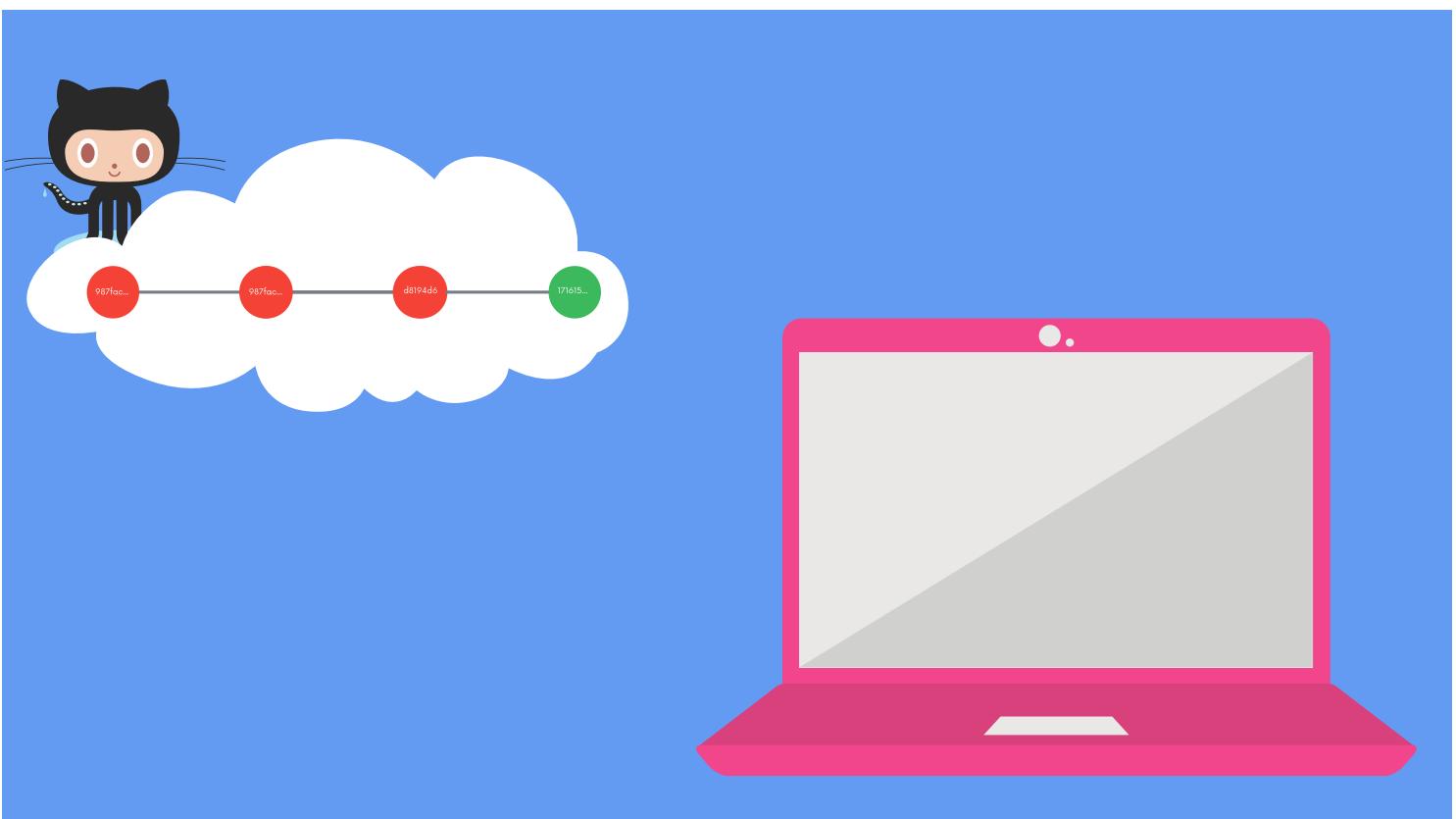
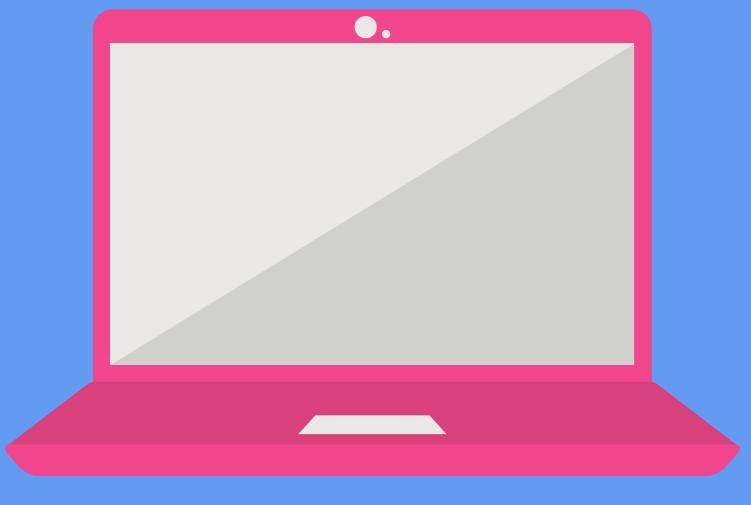


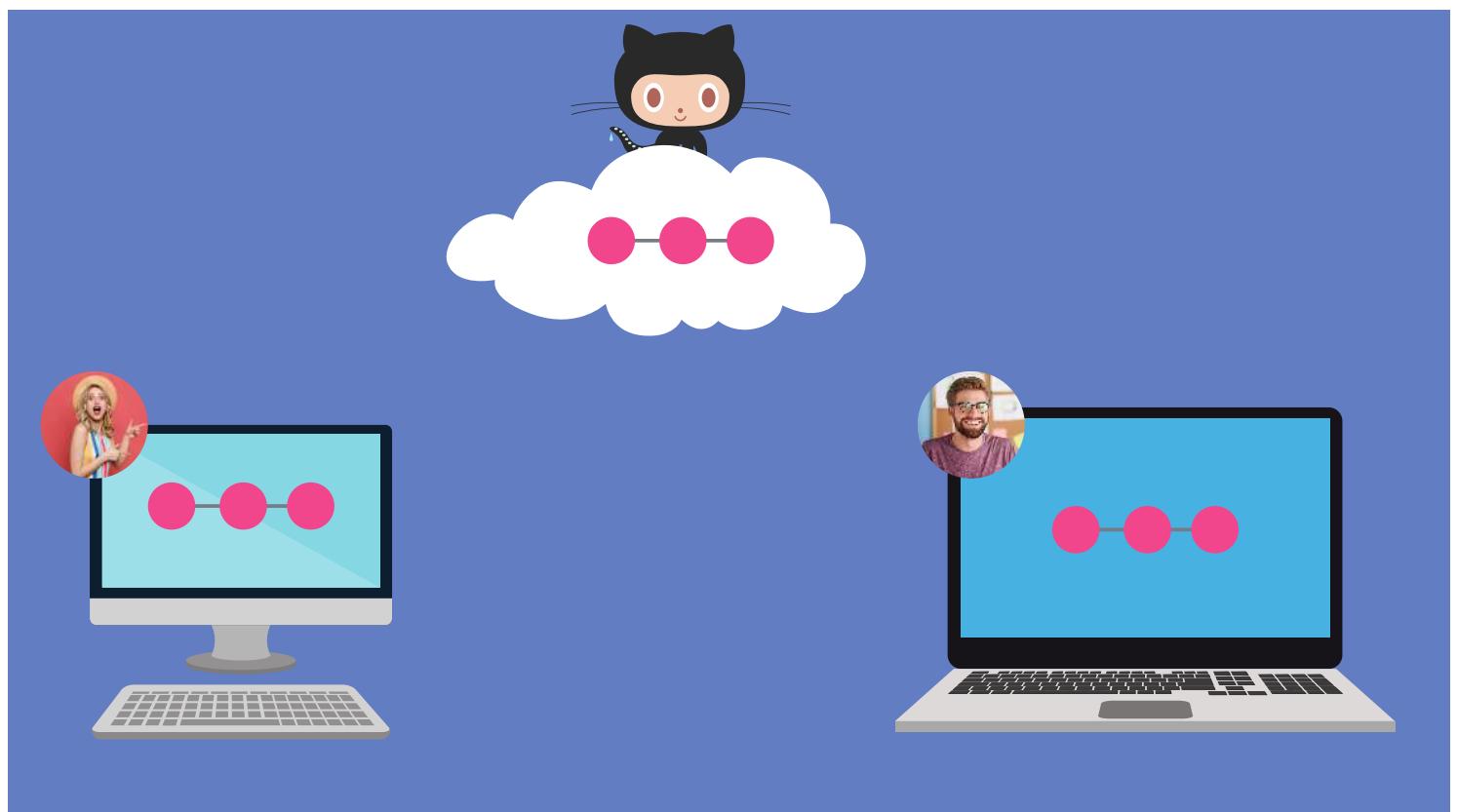
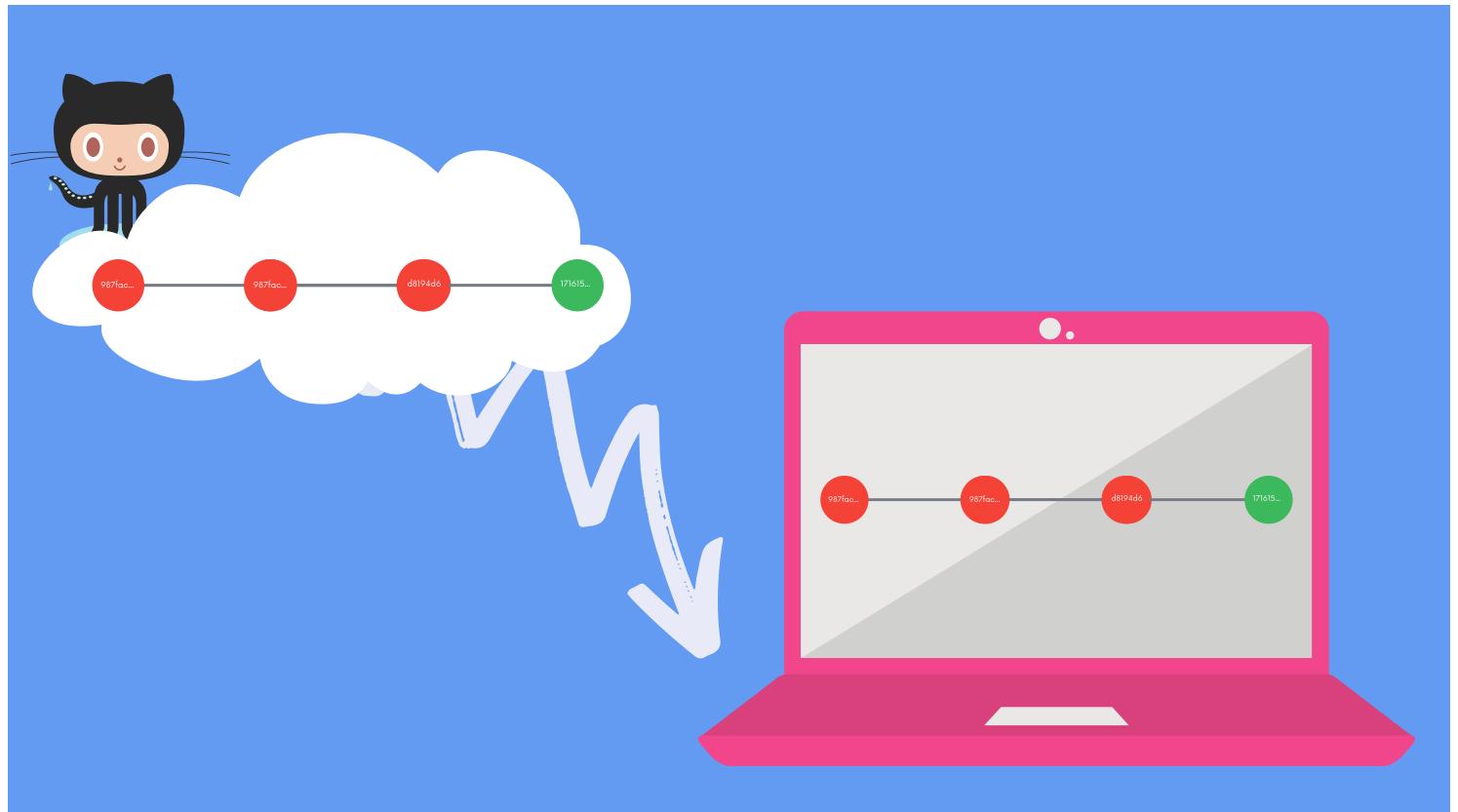


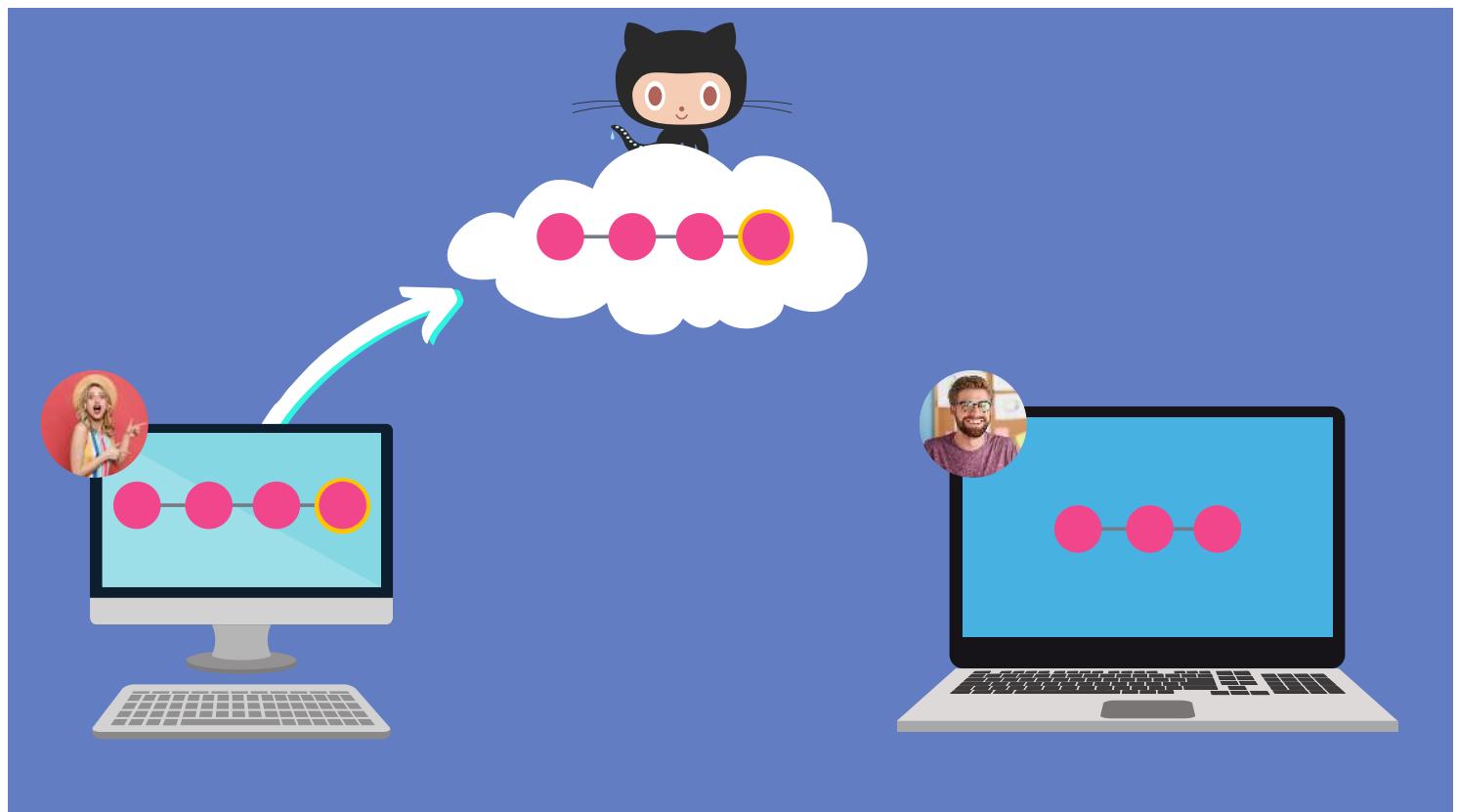
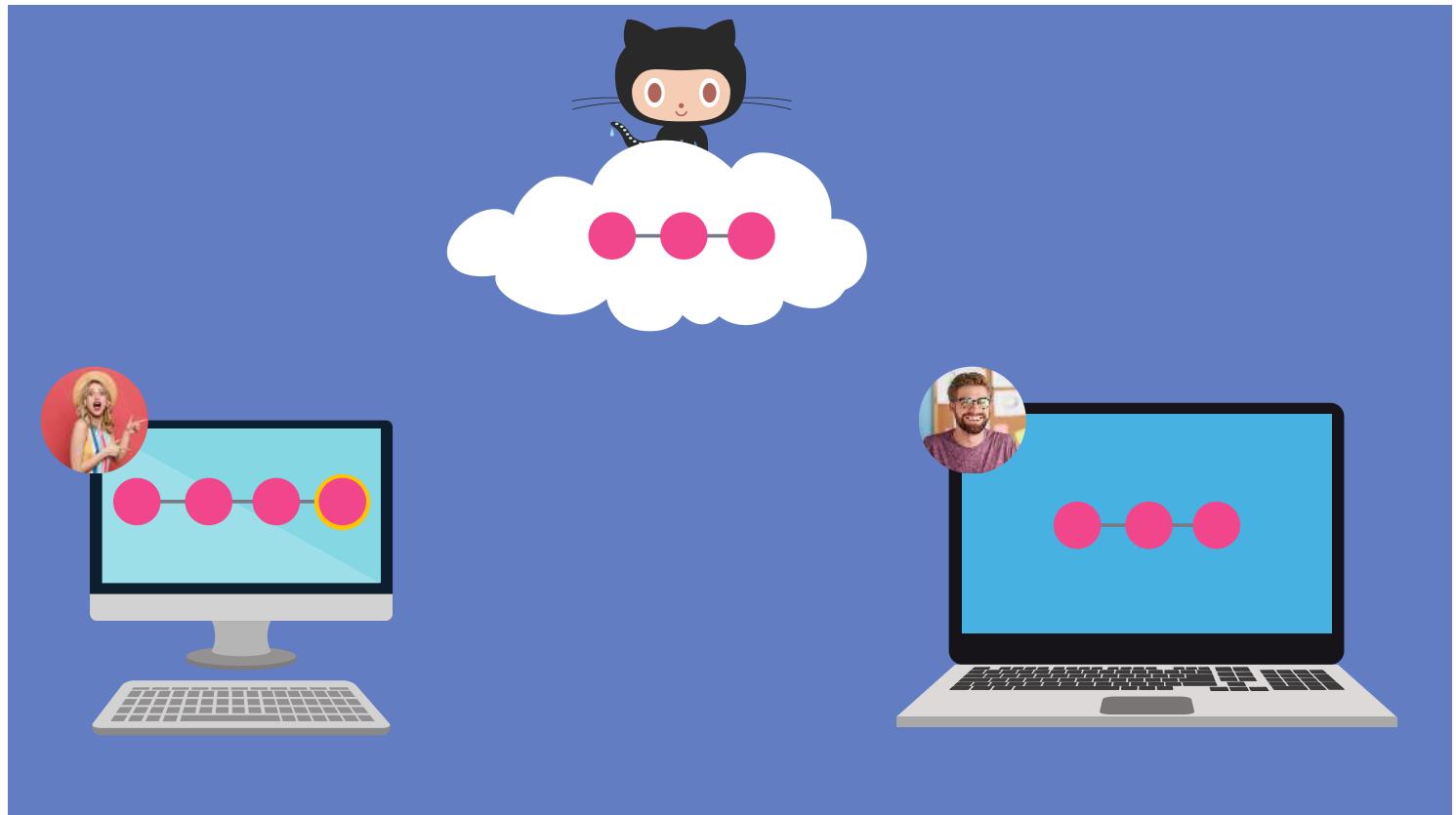


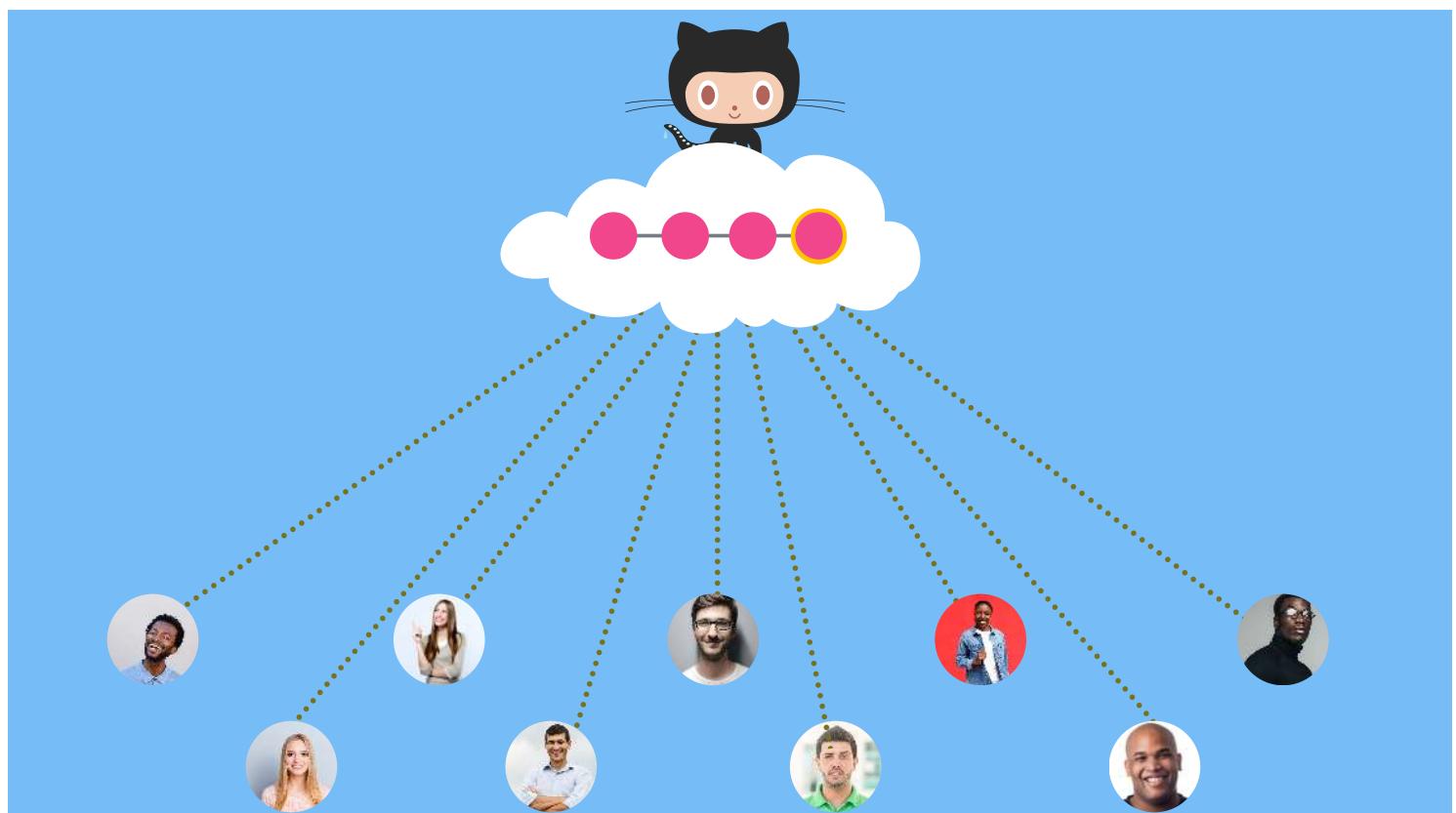
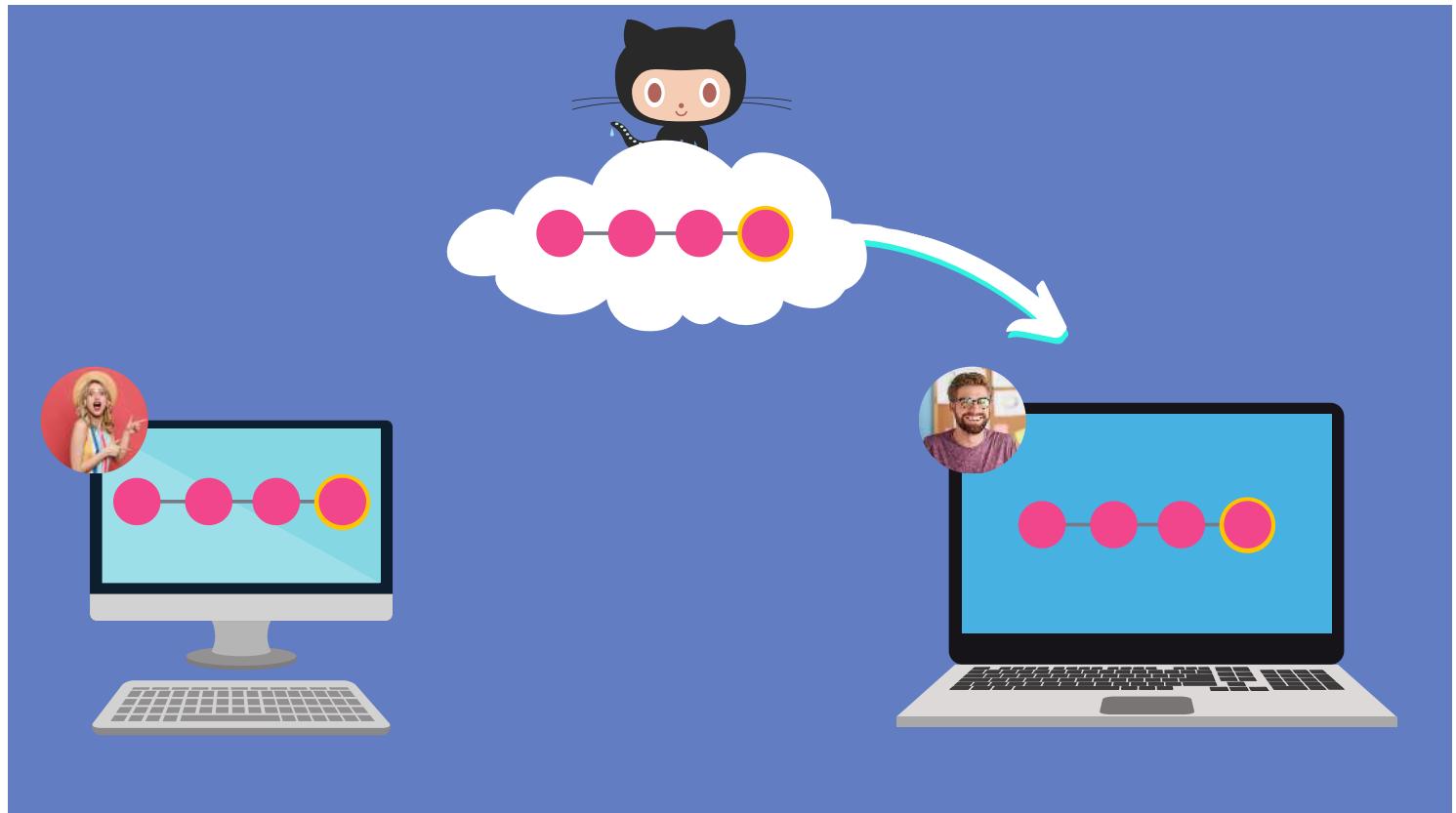
I GOT A NEW LAPTOP!

Now I need my code









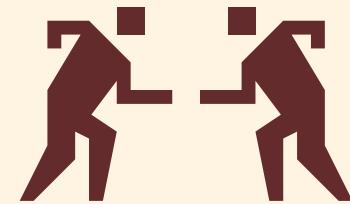


Github is not your only option...

There are tons of competing tools that provide similar hosting and collaboration features, including GitLab, BitBucket, and Gerrit.

With that said....

VS



It's very popular!

Founded in 2008, Github is now **the world's largest host of source code**. In early 2020, Github reported having over 40 million users and over 190 million repositories on the platform.





It's Free!

Github offers its basic services for free! While Github does offer paid Team and Enterprise tiers, the basic Free tier allows for unlimited public and private repos, unlimited collaborators, and more!



Why You Should Use Github (or at least know how to use it)





Collaboration

If you ever plan on working on a project with at least one other person, Github will make your life easier! Whether you're building a hobby project with your friend or you're collaborating with the entire world, Github is essential!



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 others' projects.

It can act as a sort of resumé 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.



Stay Up To Date

Being active on Github is the best way to stay up to date with the projects and tools you rely on. Learn about upcoming changes and the decisions/debate behind them.

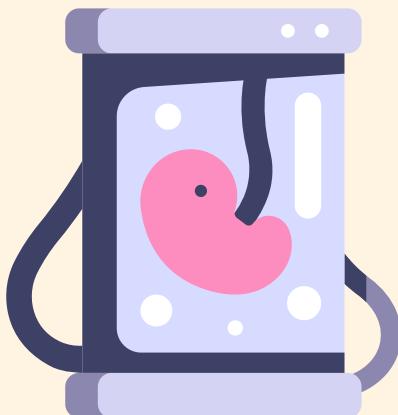




Cloning

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>`.

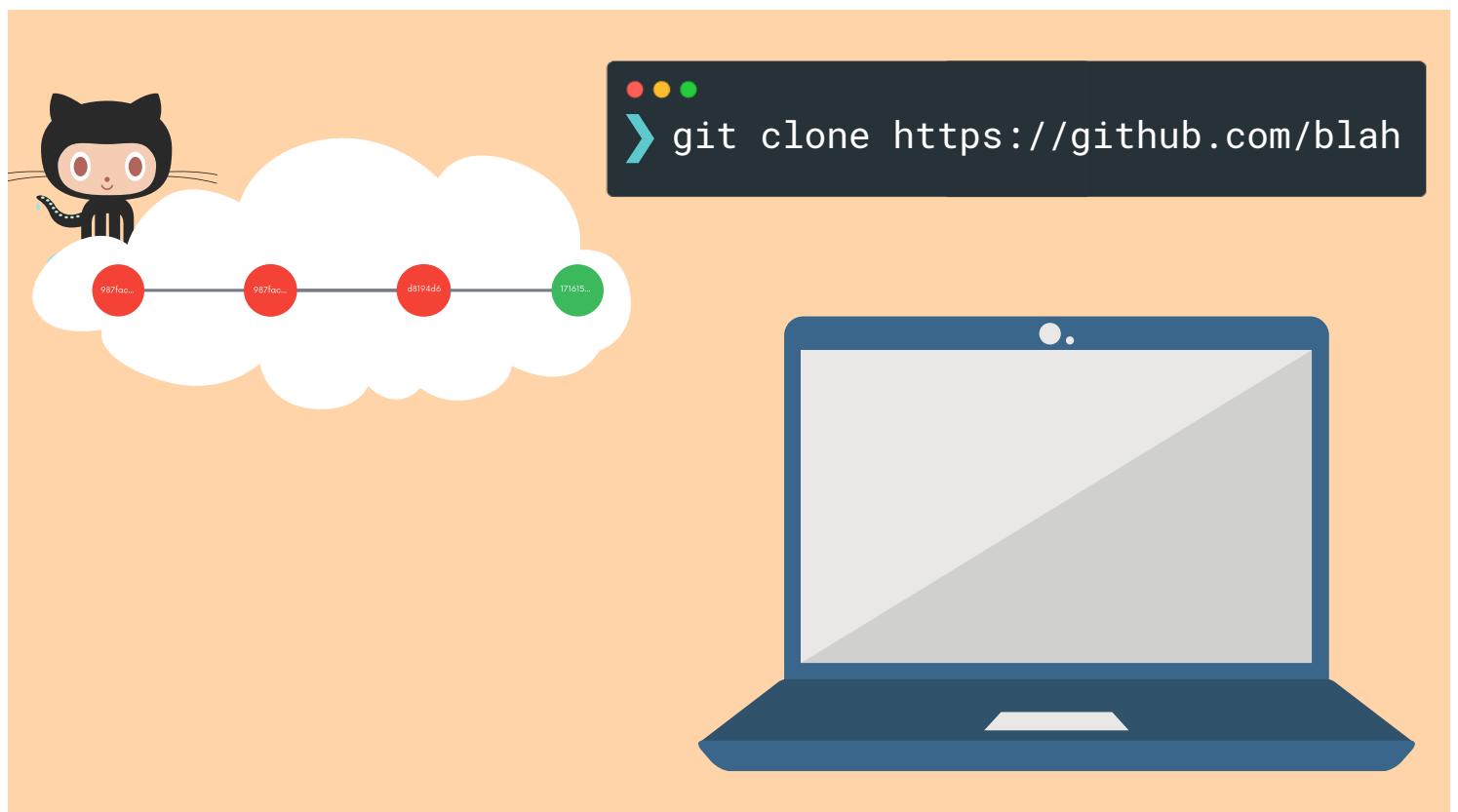
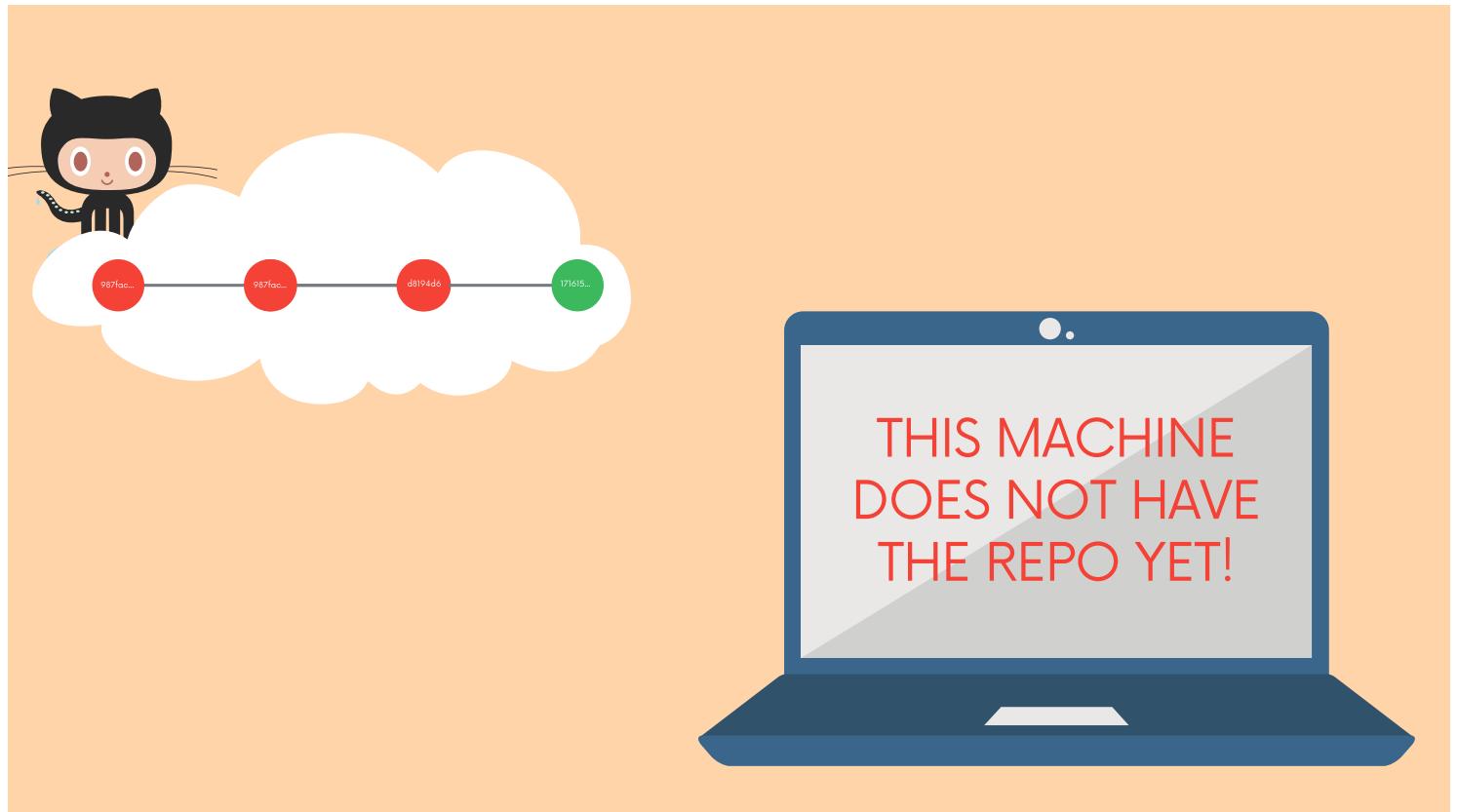
Git will retrieve all the files associated with the repository and will copy them to your local machine.

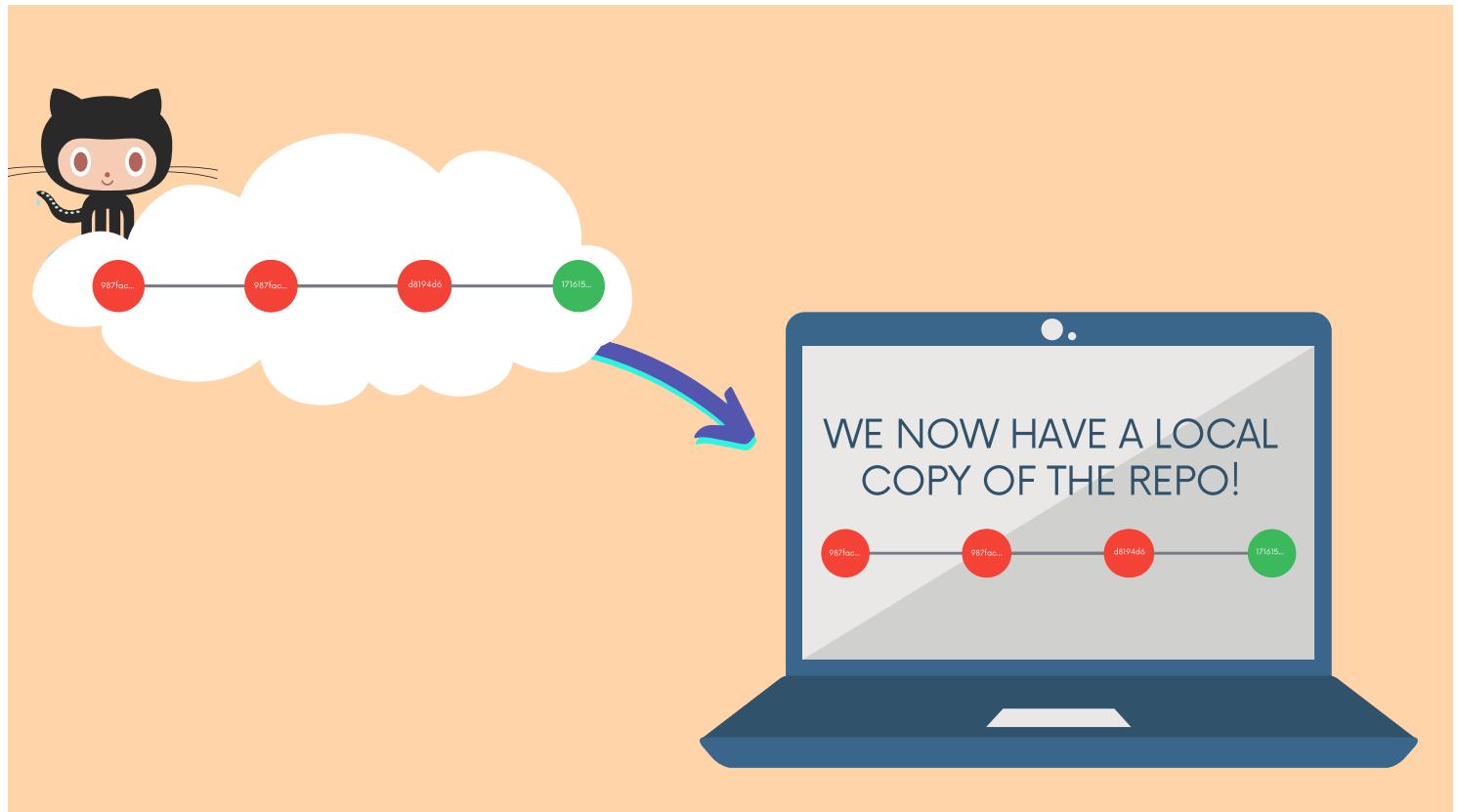
In addition, Git initializes a new repository on your machine, giving you access to the full Git history of the cloned project.

```
❯ git clone <url>
```

Make sure you are not inside of a repo when you clone!







≡

Permissions?

Anyone can clone a repository from Github, provided the repo is public. You do not need to be an owner or collaborator to clone the repo locally to your machine. You just need the URL from Github.

Pushing up your own changes to the Github repo...that's another story entirely! You need permission to do that!





We are not limited to Github Repos!

`git clone` is a standard git command.

It is NOT tied specifically to Github. We can use it to clone repositories that are hosted anywhere! It just happens that most of the hosted repos are on Github these days.



Let's Register!



Configuring SSH Keys



≡

SSH Keys

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 your 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.





How Do I Get My Code On Github?



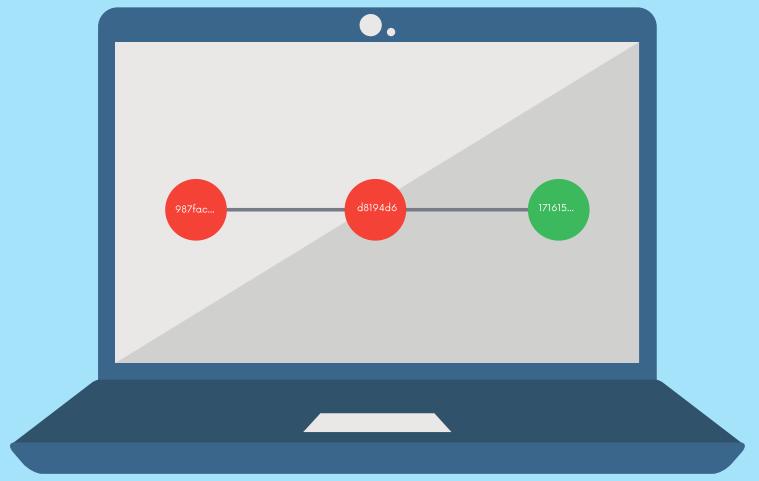
Option 1: Existing Repo

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

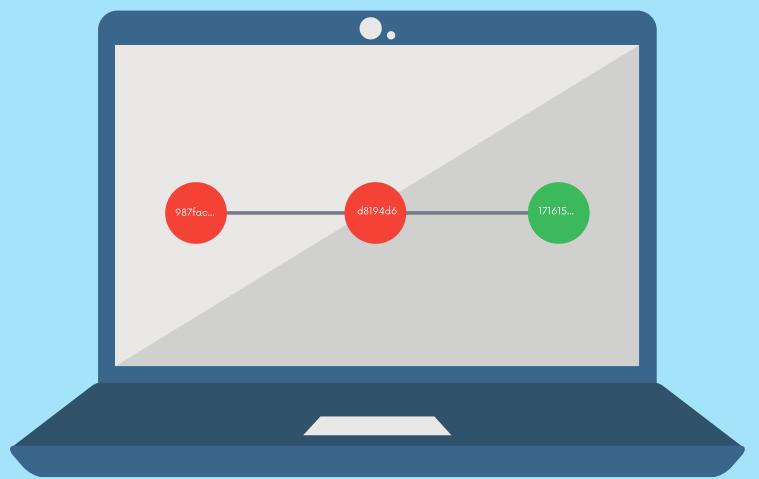
- Create a new repo on Github
- Connect your local repo (add a remote)
- Push up your changes to Github

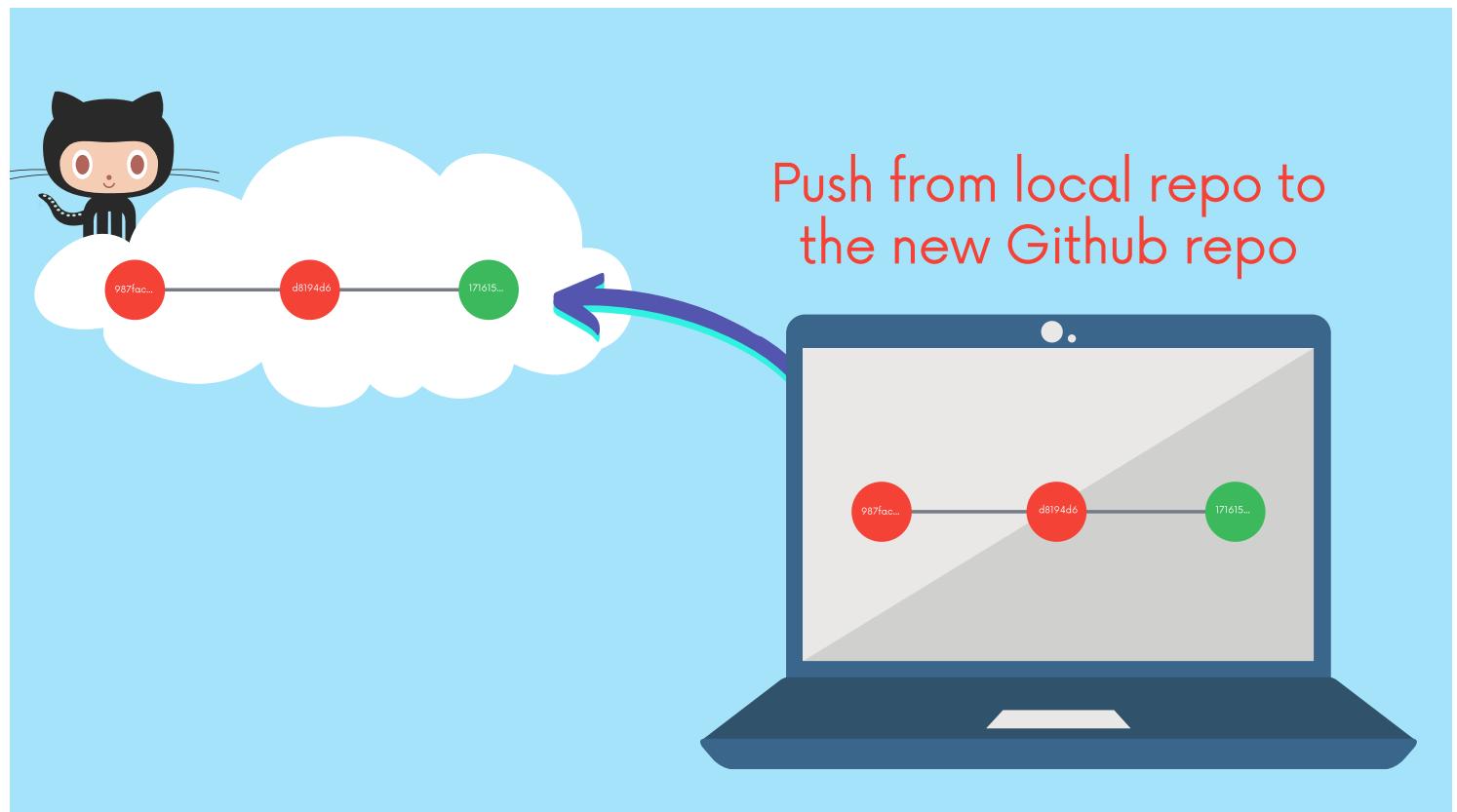


This lovely repo only exists on the laptop:



Create Empty
Github Repo!





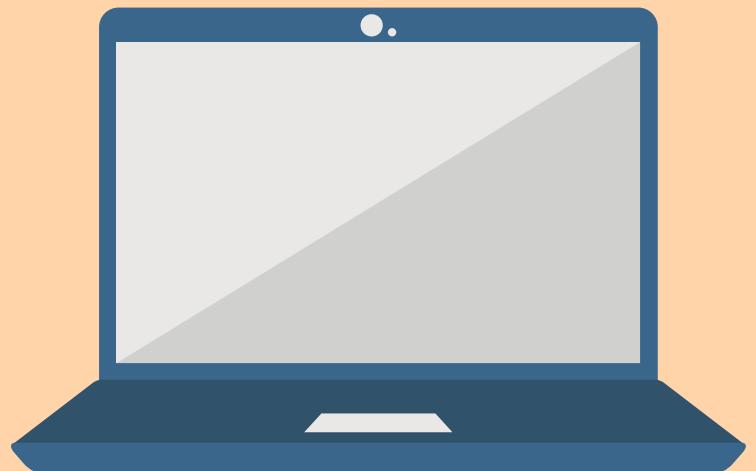
Option 2: Start From Scratch

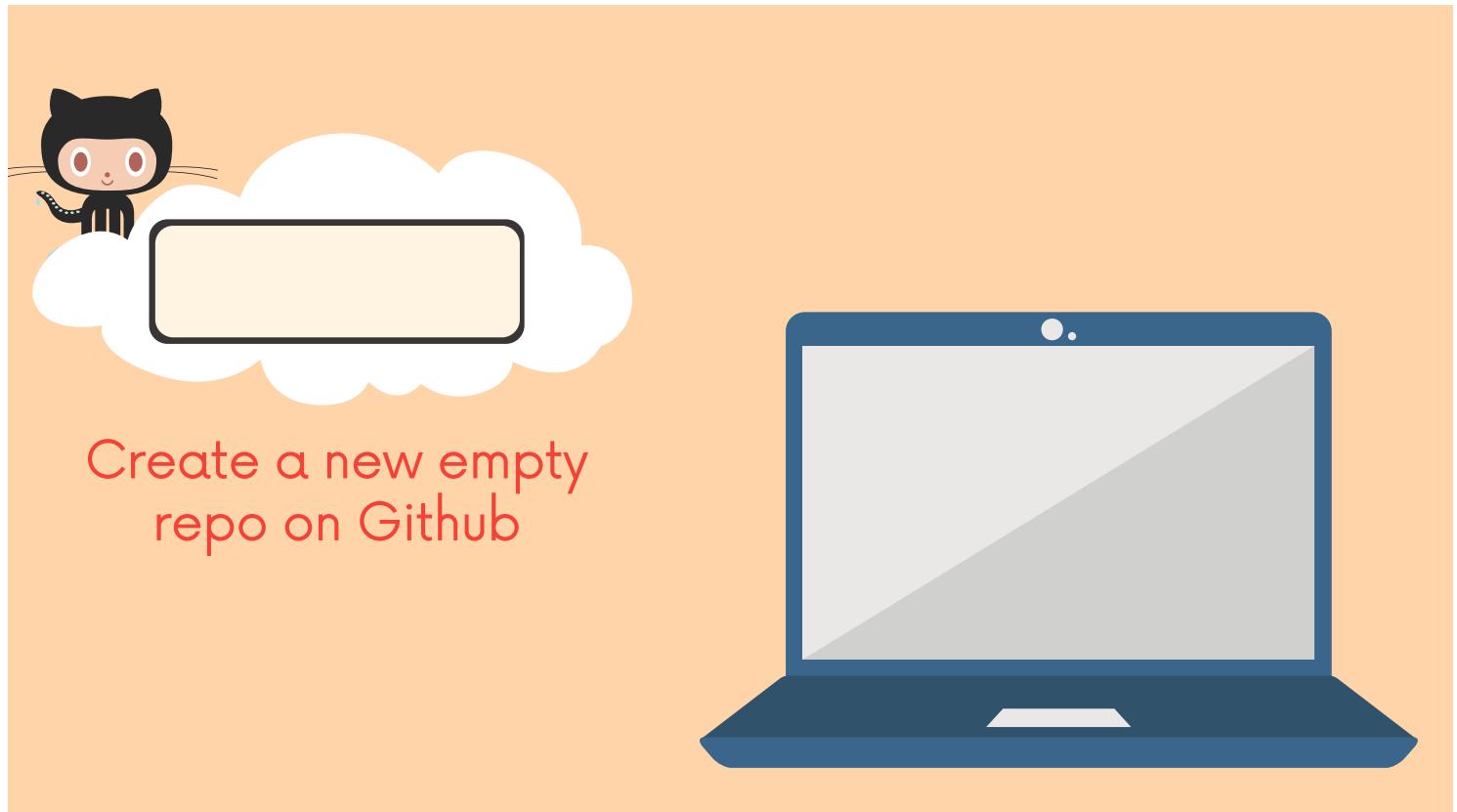
If you haven't begun work on your local repo, you can...

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

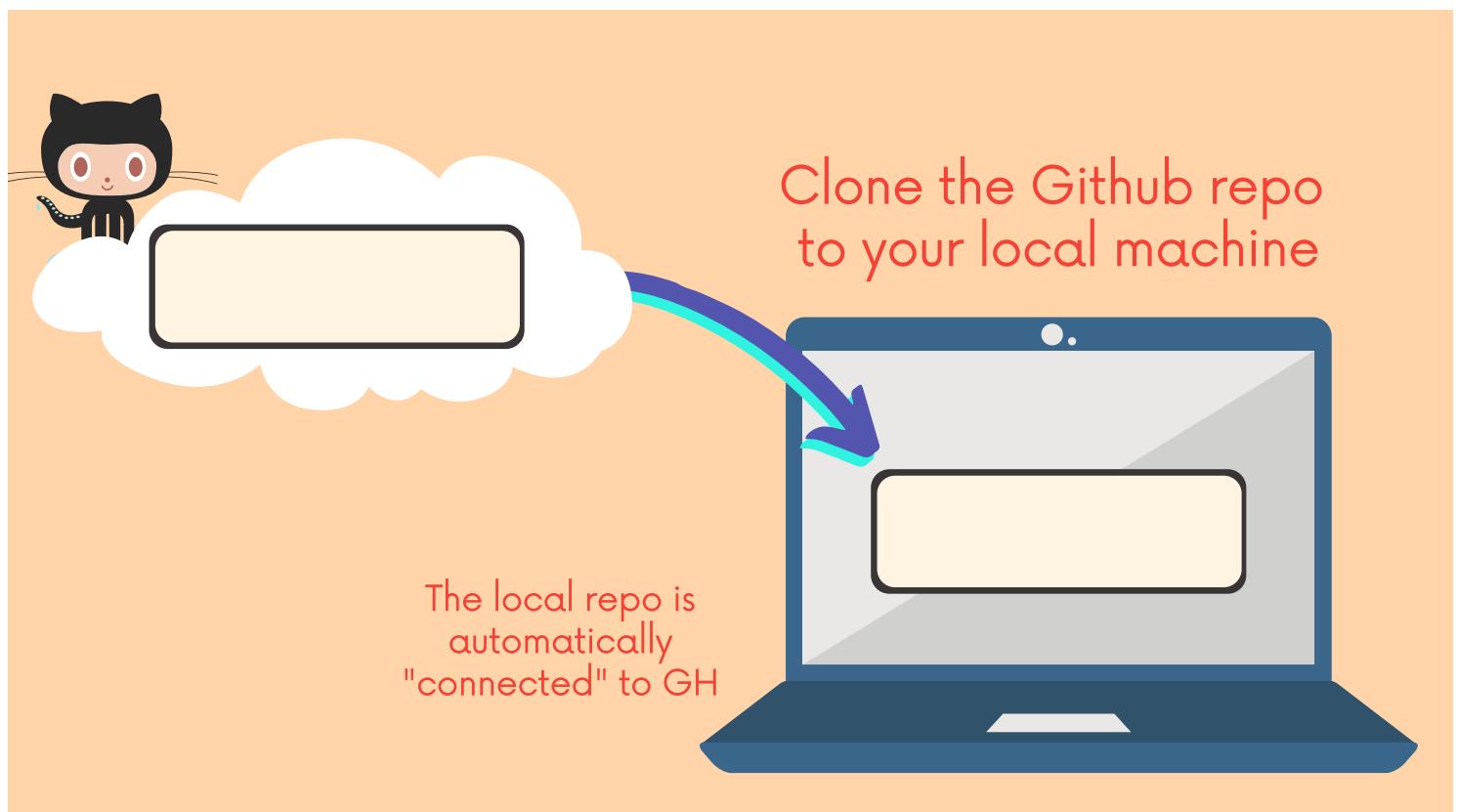


No repo exists yet!



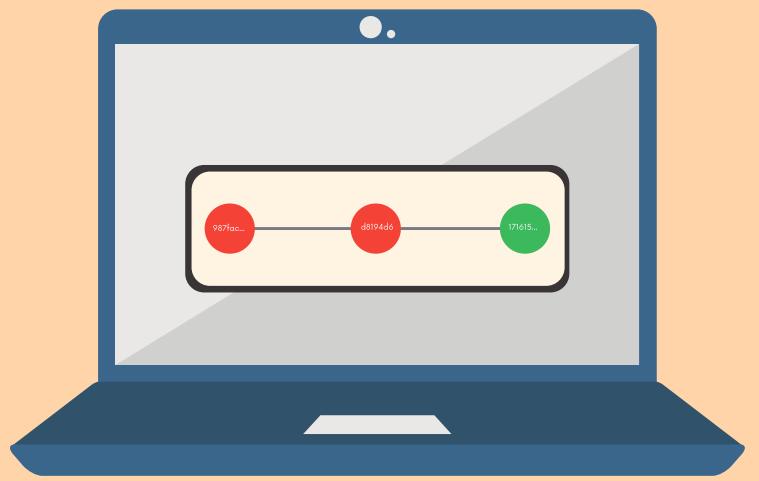


Create a new empty
repo on Github

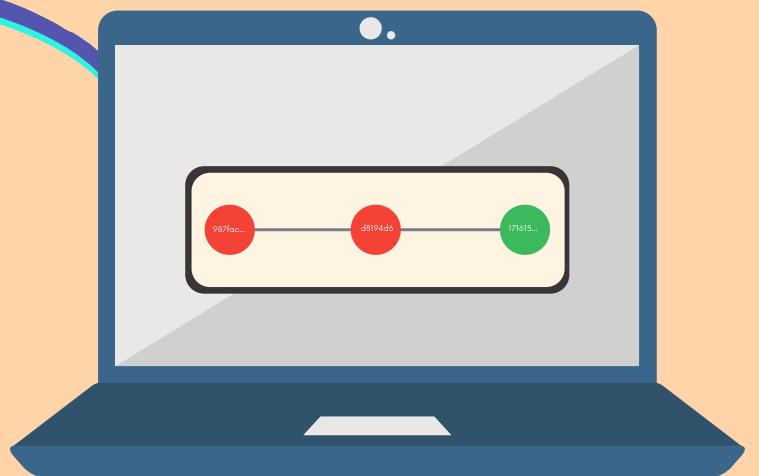




Do some work and make
some commits locally



Push that new work
up to Github!



Pushing

To get your own changes and Git history up on Github, we need to PUSH them up. The typical workflow looks something like this:

- Make some changes locally
- Add and commit those changes
- Repeat...
- Push new commits up to Github



First, We Need To Make a Repo On 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 your 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
```





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>
```



Adding A New Remote

```
❯ git remote add origin  
https://github.com/blah/repo.git
```

Okay Git, anytime I use the name "origin", I'm referring to this particular Github repo URL.





Origin?

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

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



Adding A New Remote

```
git remote add mygithuburl  
https://github.com/meh/repo.git
```

Okay Git, anytime I use the name "mygithuburl", I'm referring to this particular Github repo URL.

This is not a commonly used remote name.





Checking Our Work

Try viewing your remotes with `git remote -v`, and you should now see a remote showing up!

Remember, by setting up a remote we are just telling Git about a remote repository URL. We have not "communicated" with the Github repo at all yet.

```
● ● ●  
❯ git remote -v
```

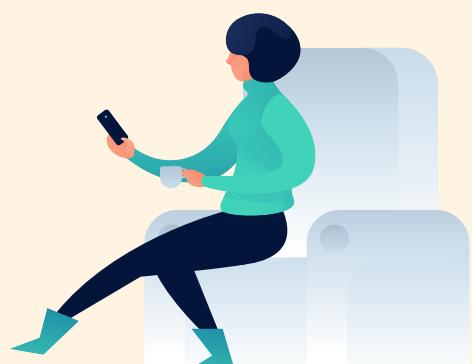


Other commands

They are not commonly used, but there are commands to rename and delete remotes if needed.

`git remote rename <old> <new>`

`git remote remove <name>`





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 local branch we want to push up to that remote.



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



An Example

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



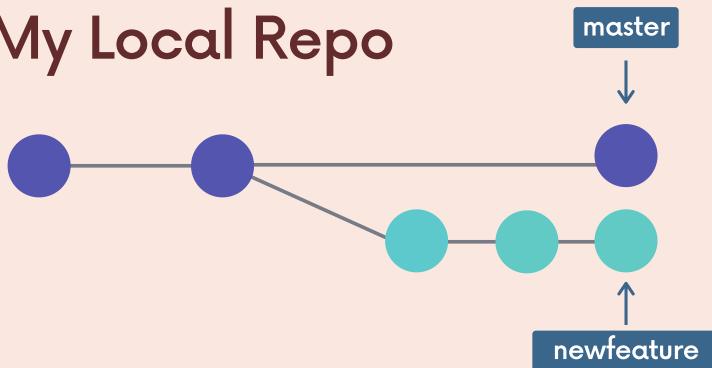
```
git push origin master
```



Github Repo

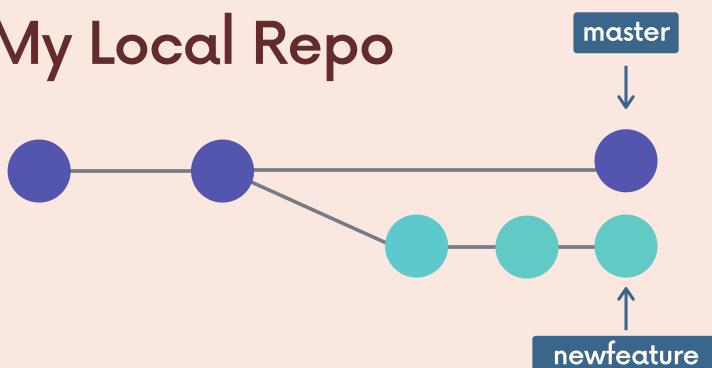
nothing to see here...

My Local Repo



Github Repo

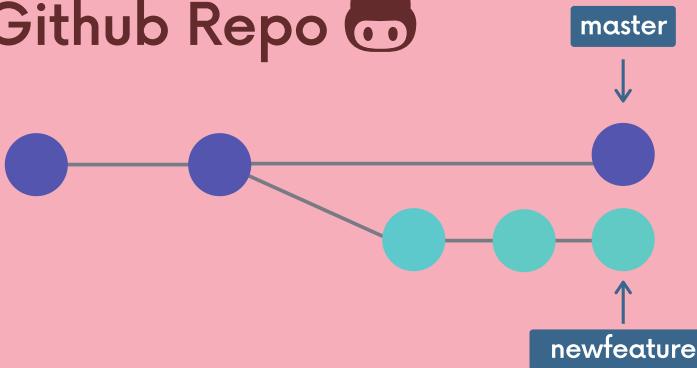
My Local Repo



```
git push origin master
```

To push up my master branch to my Github repo
(assuming my remote is named origin)

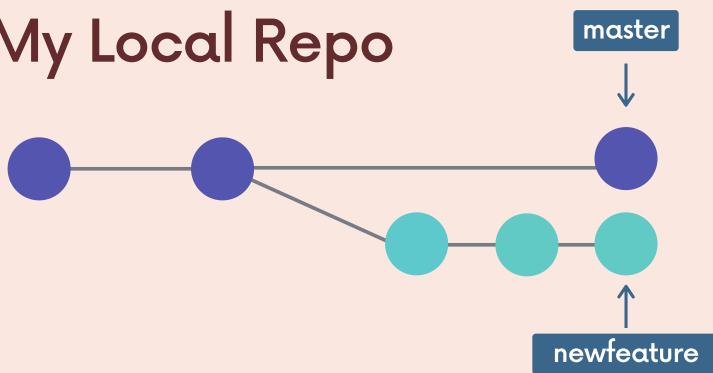
Github Repo 🐱



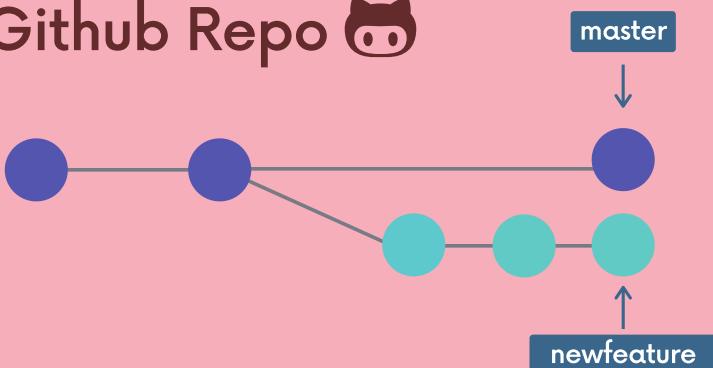
To push up the **newfeature** branch to Github...

```
git push origin newfeature
```

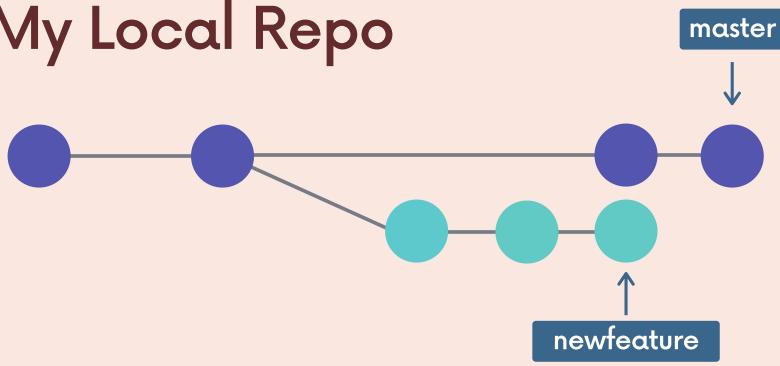
My Local Repo



Github Repo 🐱

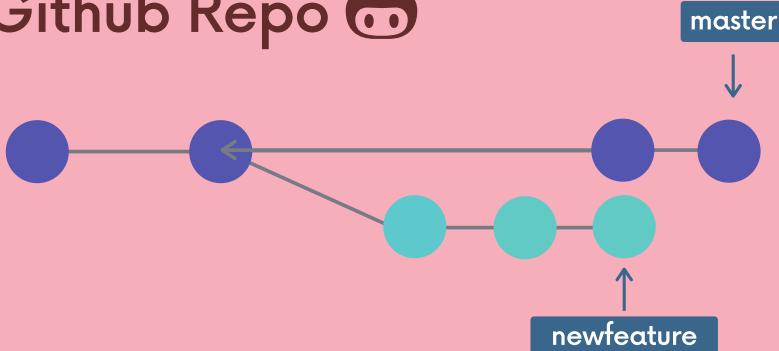


My Local Repo



I make some new commits locally.
My Github repo has no idea!

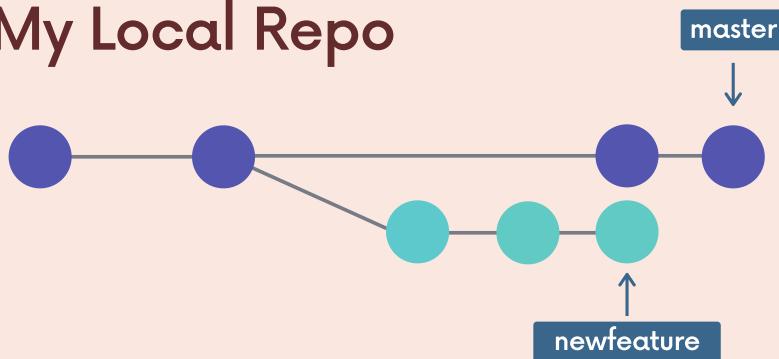
Github Repo 🐱



Push up the master branch again, to make sure the Github repo has the new commits

```
git push origin master
```

My Local Repo



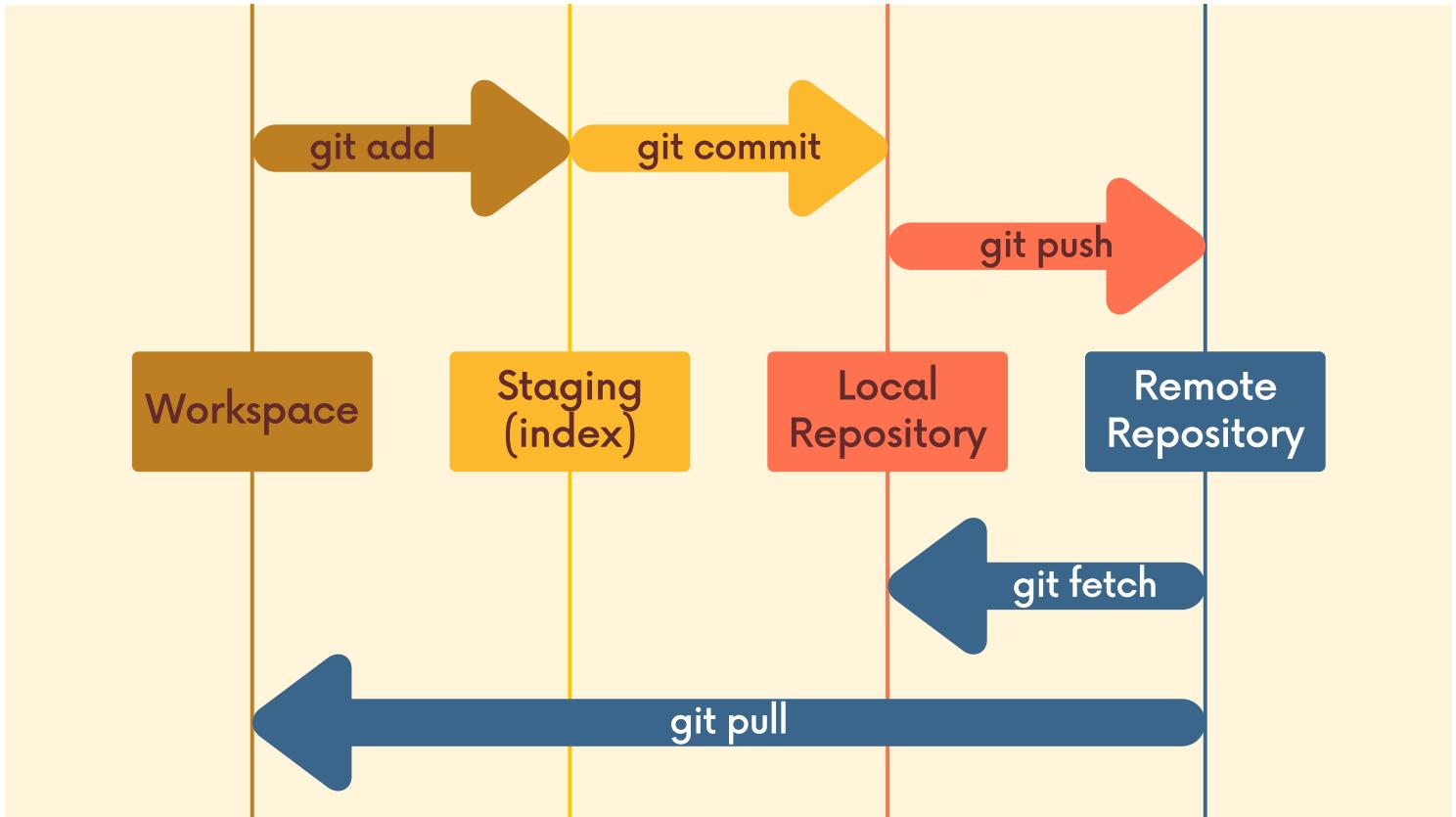
Workflow Recap

Remember to follow these basic steps:

- Create a new empty repo on Github
- Copy the repo URL
- Add a remote to your local repo, using the URL
- Push your changes up to the remote

```
git remote -v
```





11 Fetching and Pulling

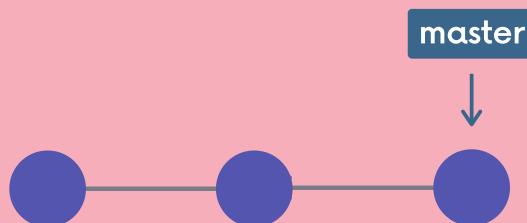
Fetching & Pulling



A Closer Look At Cloning

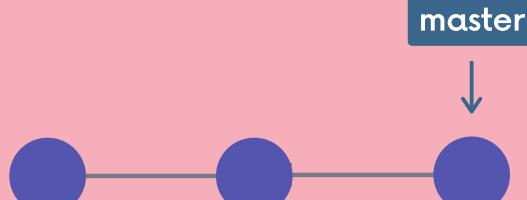


Github Repo



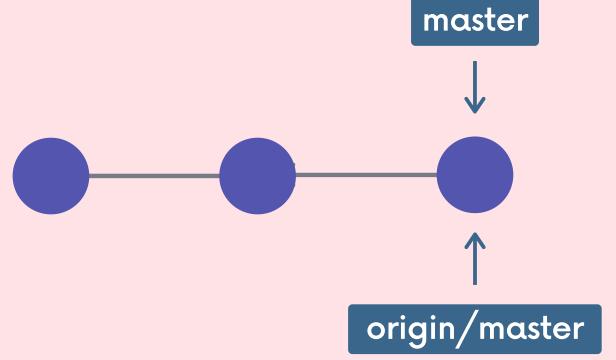
My Computer

Github Repo

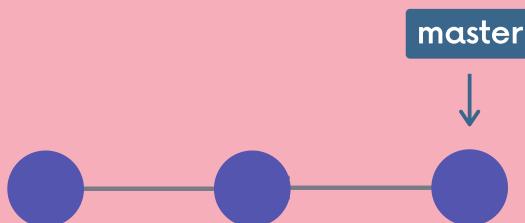


My Computer

git clone

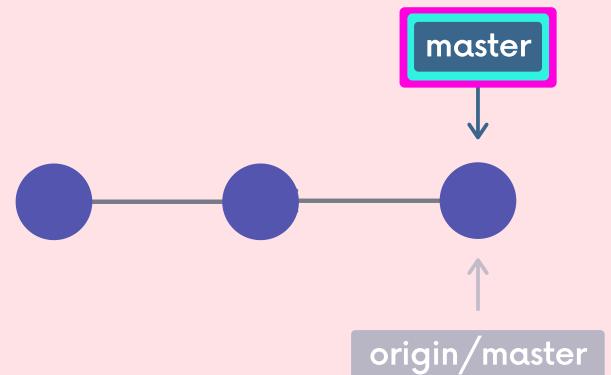


Github Repo

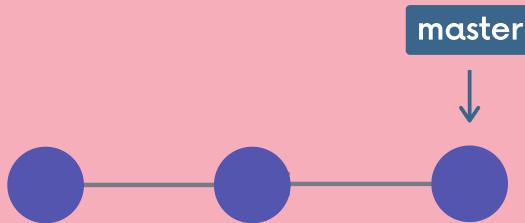


My Computer

A regular branch reference.
I can move this around myself.

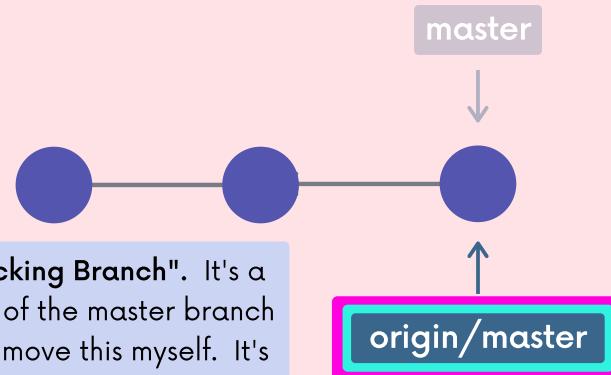


Github Repo



My Computer

This is a "Remote Tracking Branch". It's a reference to the state of the master branch on the remote. I can't move this myself. It's like a bookmark pointing to the last known commit on the master branch on origin





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>.

- `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)



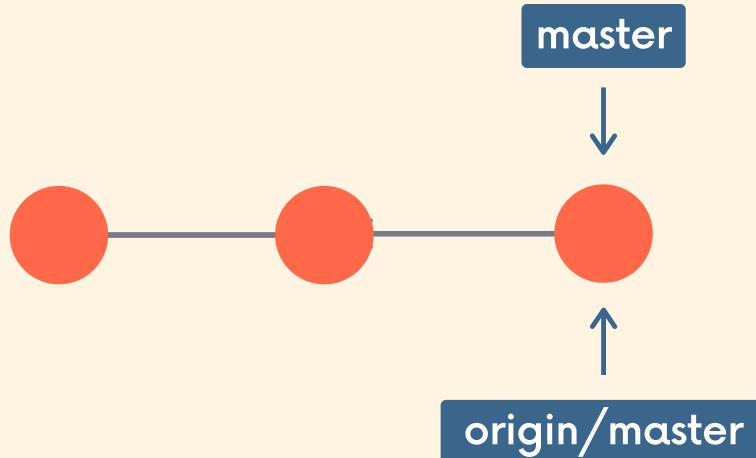
Remote Branches

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

```
●●●  
❯ git branch -r  
origin/master
```

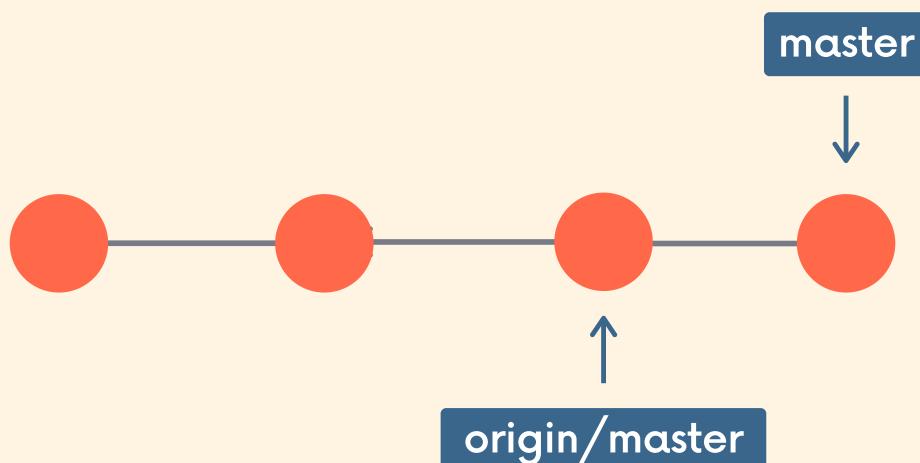


My Computer



My Computer

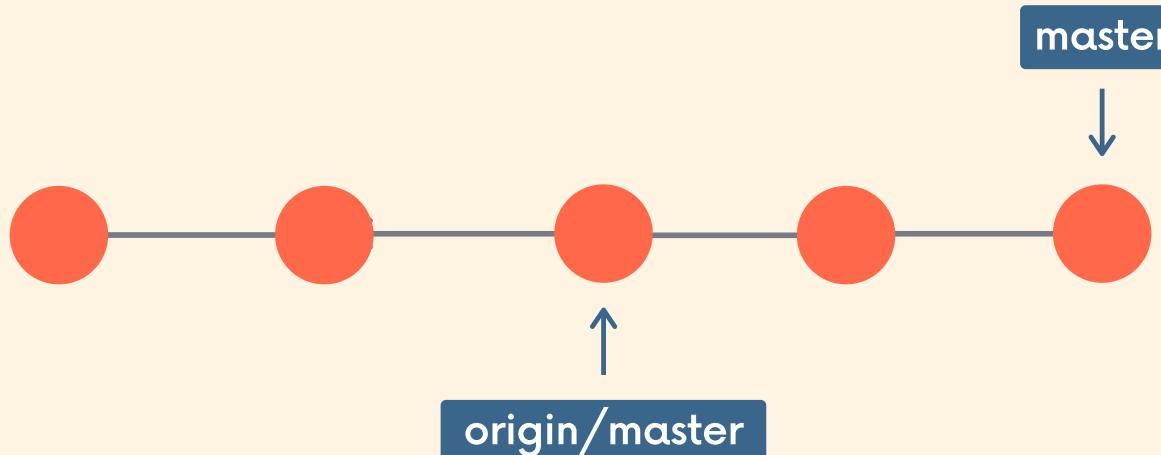
I make a new commit locally. My master branch reference updates, like always.



The remote reference stays the same

My Computer

I make another commit, and the local branch reference moves again.



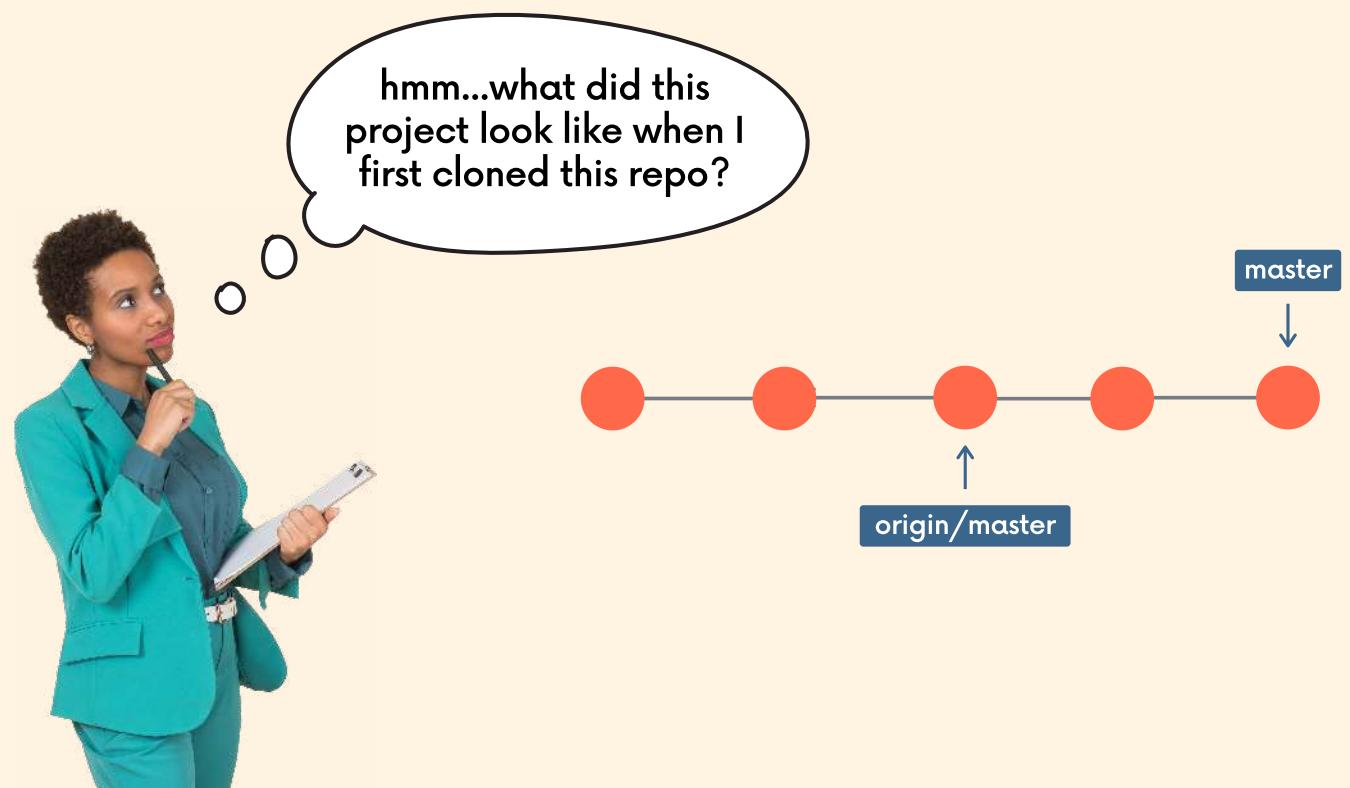
Remote reference doesn't move!



When I run `git status`

```
❯ git status
On branch master
Your branch is ahead of 'origin/master' by 2
commits.
  (use "git push" to publish your local commits)
```





≡

You can checkout these remote branch pointers



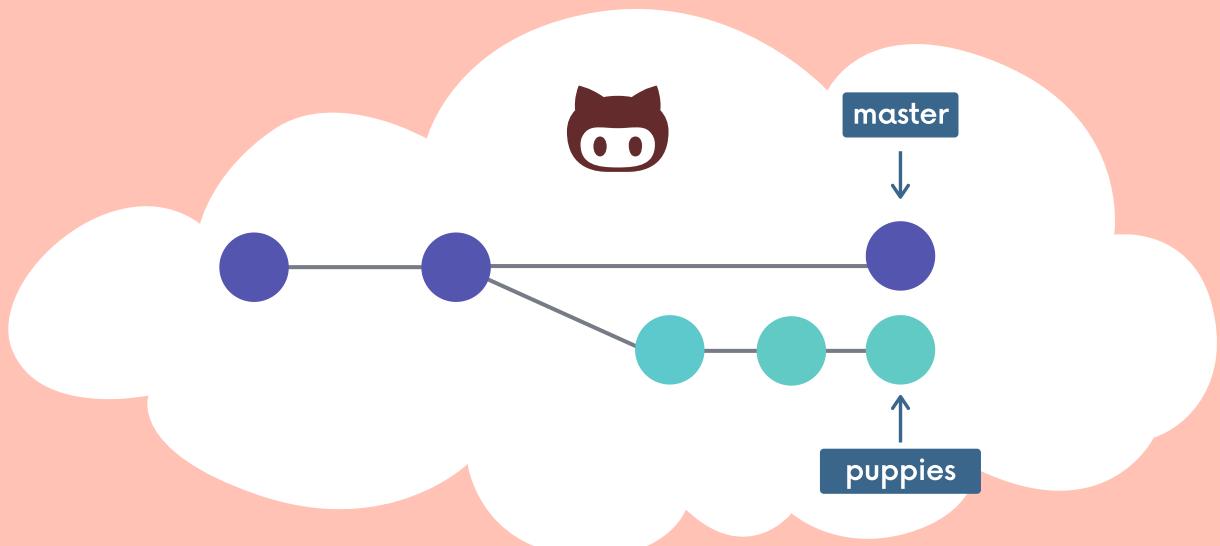
```
git checkout origin/master
```

Note: switching to 'origin/master'.
You are in 'detached HEAD' state. You can look
around, make experimental changes and commit
them, and you can discard any commits you make
in this blah blah blah

Detached HEAD! Don't panic. It's fine.



Suppose I Just Cloned This Github Repo



≡

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?

```
● ● ●
> git branch
*master
```



Remote Branches

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

```
● ● ●
> git branch -r
origin/master
origin/puppies
```



Workspace

master



By default, my master branch is already tracking origin/master.

I didn't connect these myself!

Remote

origin/master

origin/puppies



I want to work on the puppies branch locally!

I could `checkout origin/puppies`, but that puts me in detached HEAD.

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`.



It's super easy!

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

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

```
▶ git switch puppies
```



```
git switch puppies  
Branch 'puppies' set up to track remote  
branch 'puppies' from 'origin'.  
Switched to a new branch 'puppies'
```

Workspace



Remote



≡

NOTE!

the new command `git switch` makes this super easy to do!
It used to be slightly more complicated using `git checkout`

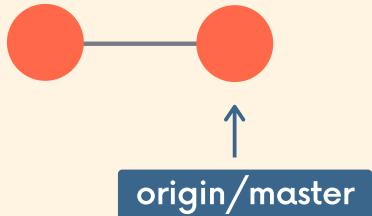
```
git checkout --track origin/puppies
```



Github



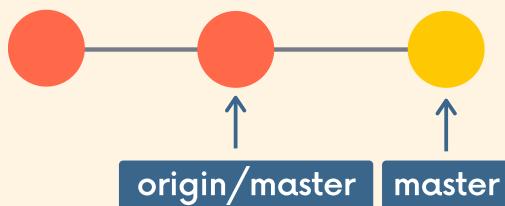
Local



Github



Local



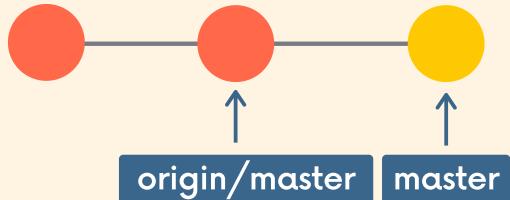
Github

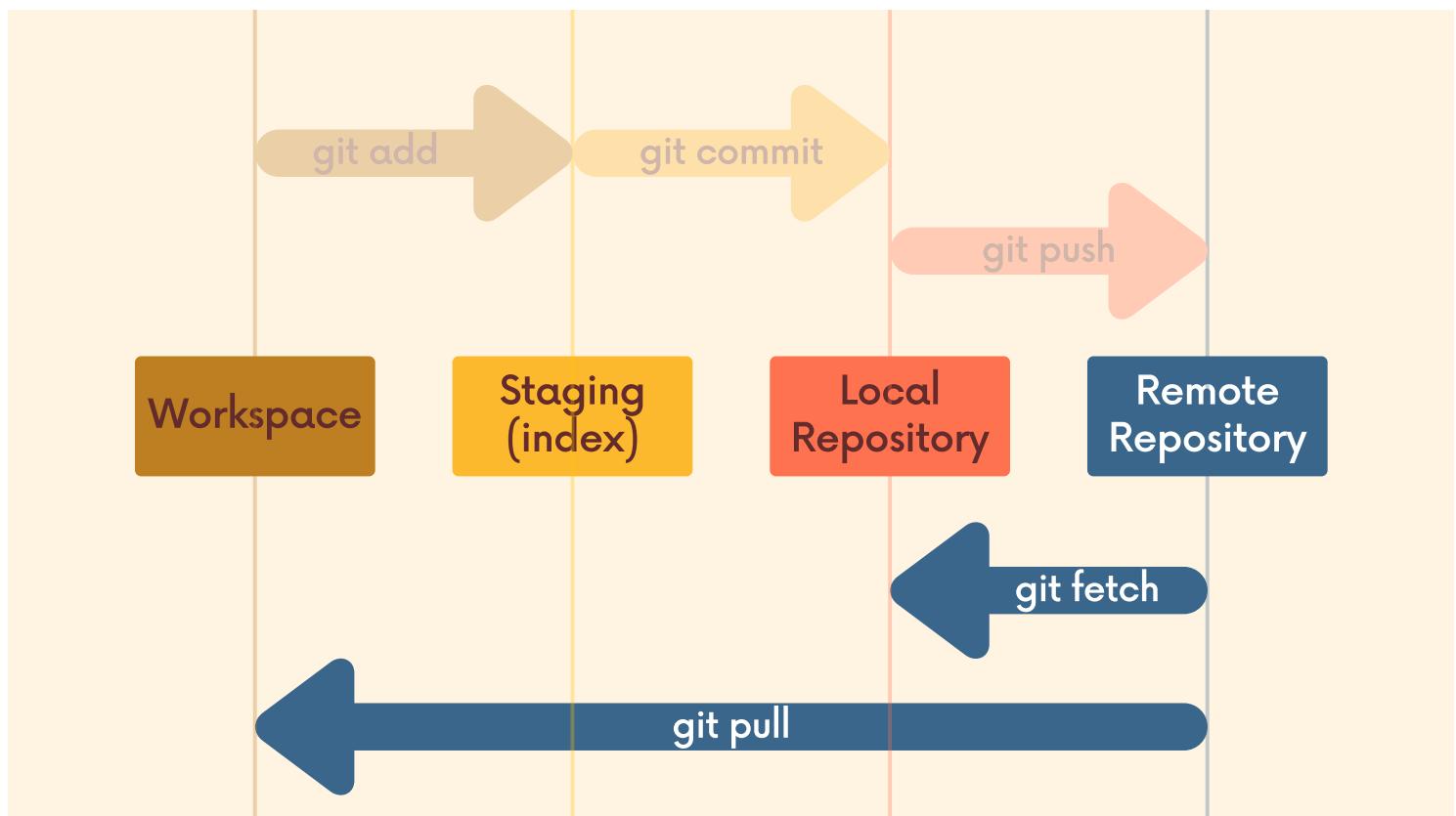
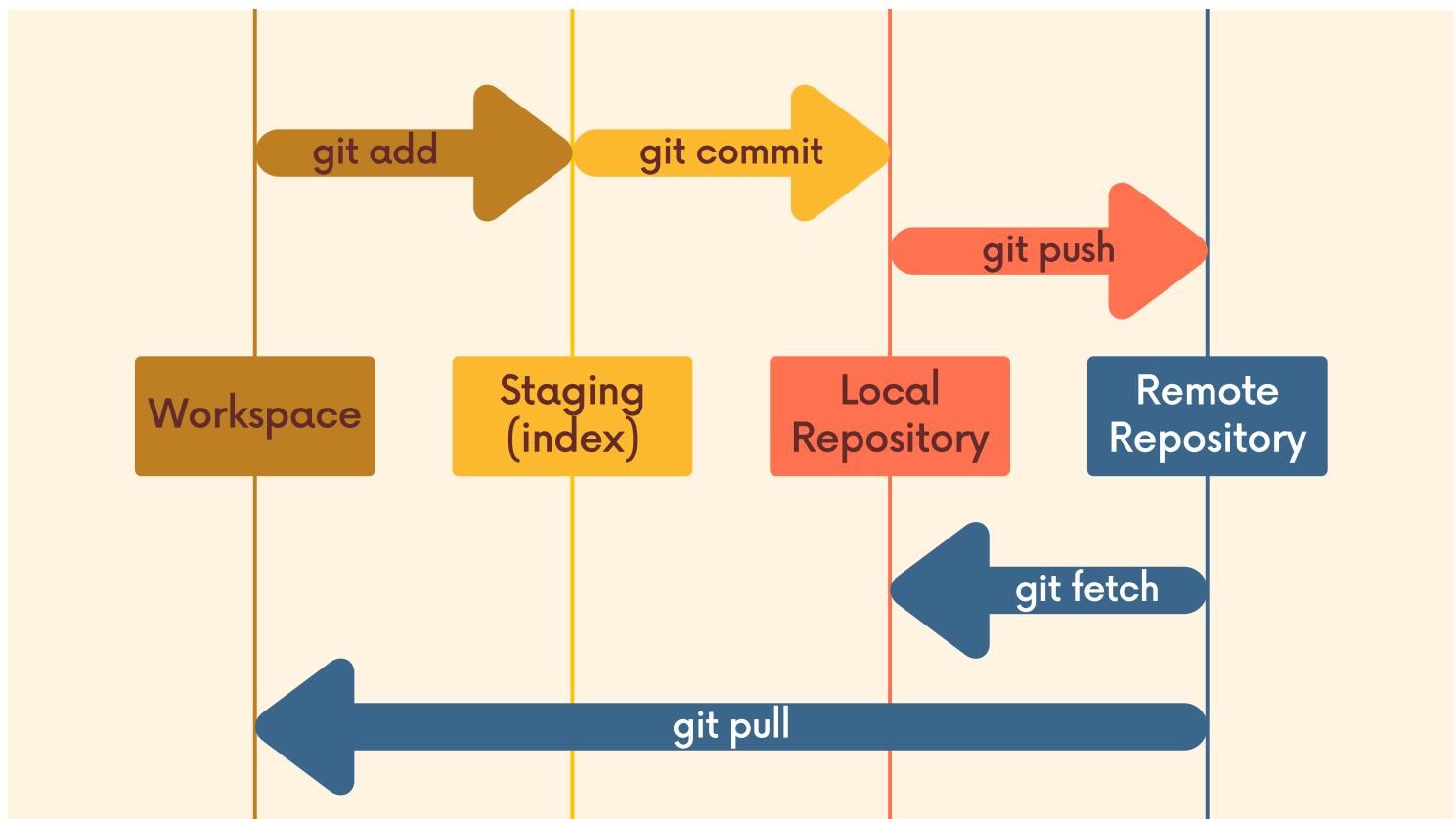


Uh oh! The remote repo has changed! A teammate has pushed up changes to the master branch, but my local repo doesn't know!

How do I get those changes???

Local







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 it as "please go and get the latest information from Github, but don't screw up my working directory."



Git Fetch

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

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

```
git fetch <remote>
```

If not specified, `<remote>` defaults to `origin`



Git Fetch

We can also fetch a specific branch from a remote using
`git fetch <remote> <branch>`

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

```
git fetch <remote> <branch>
```



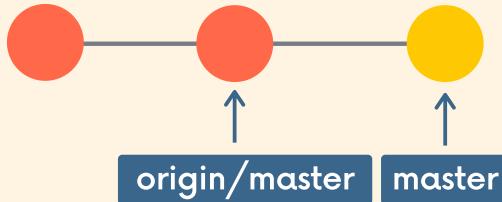
Github

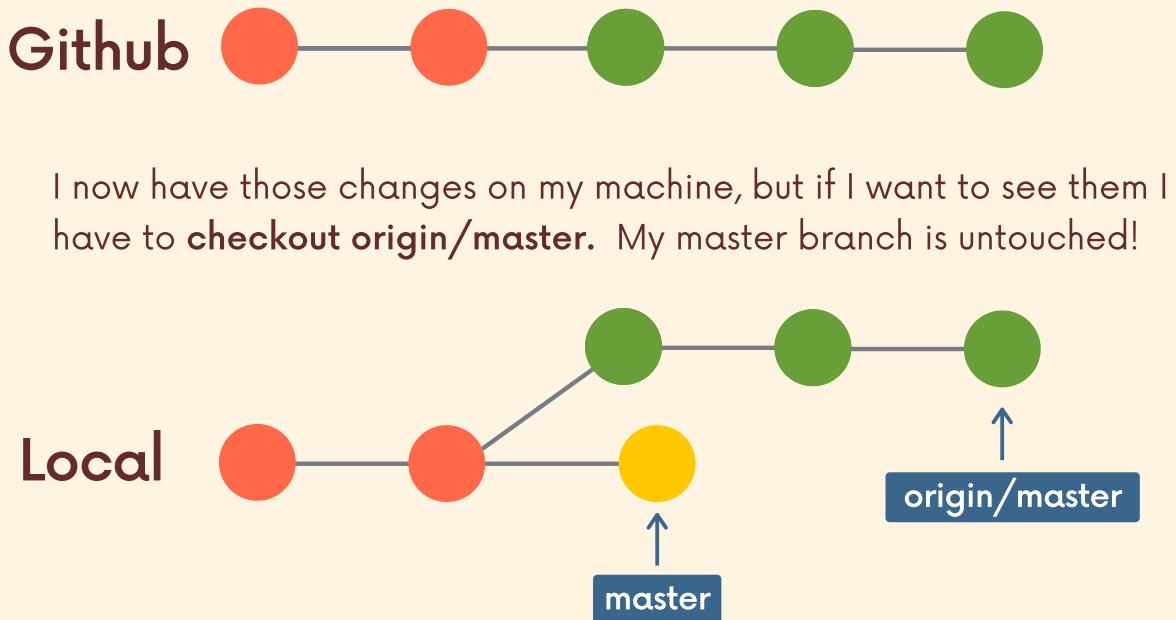
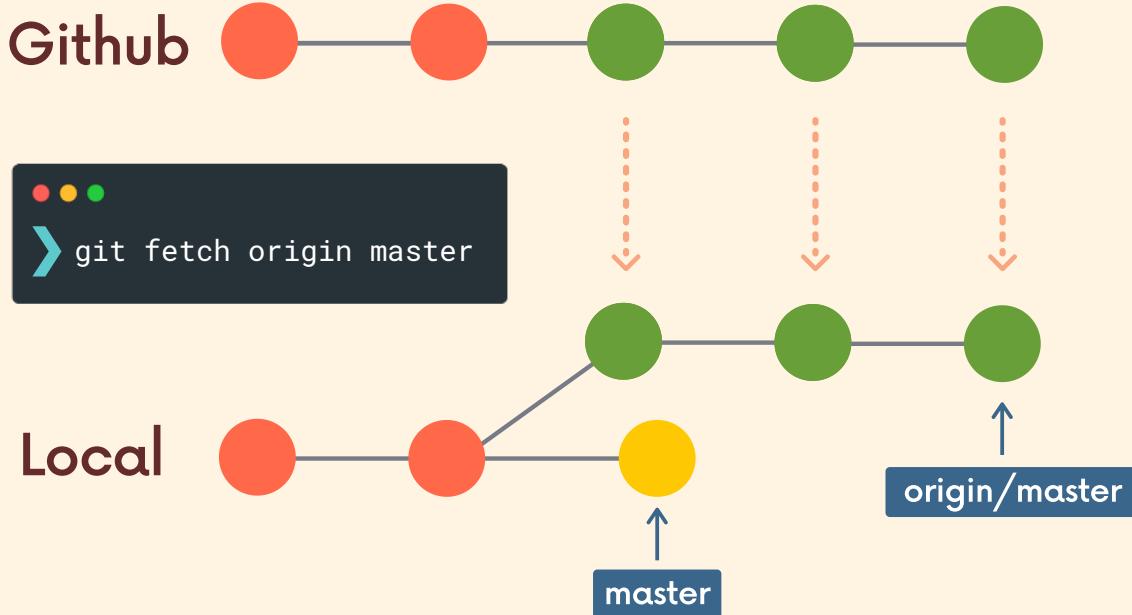


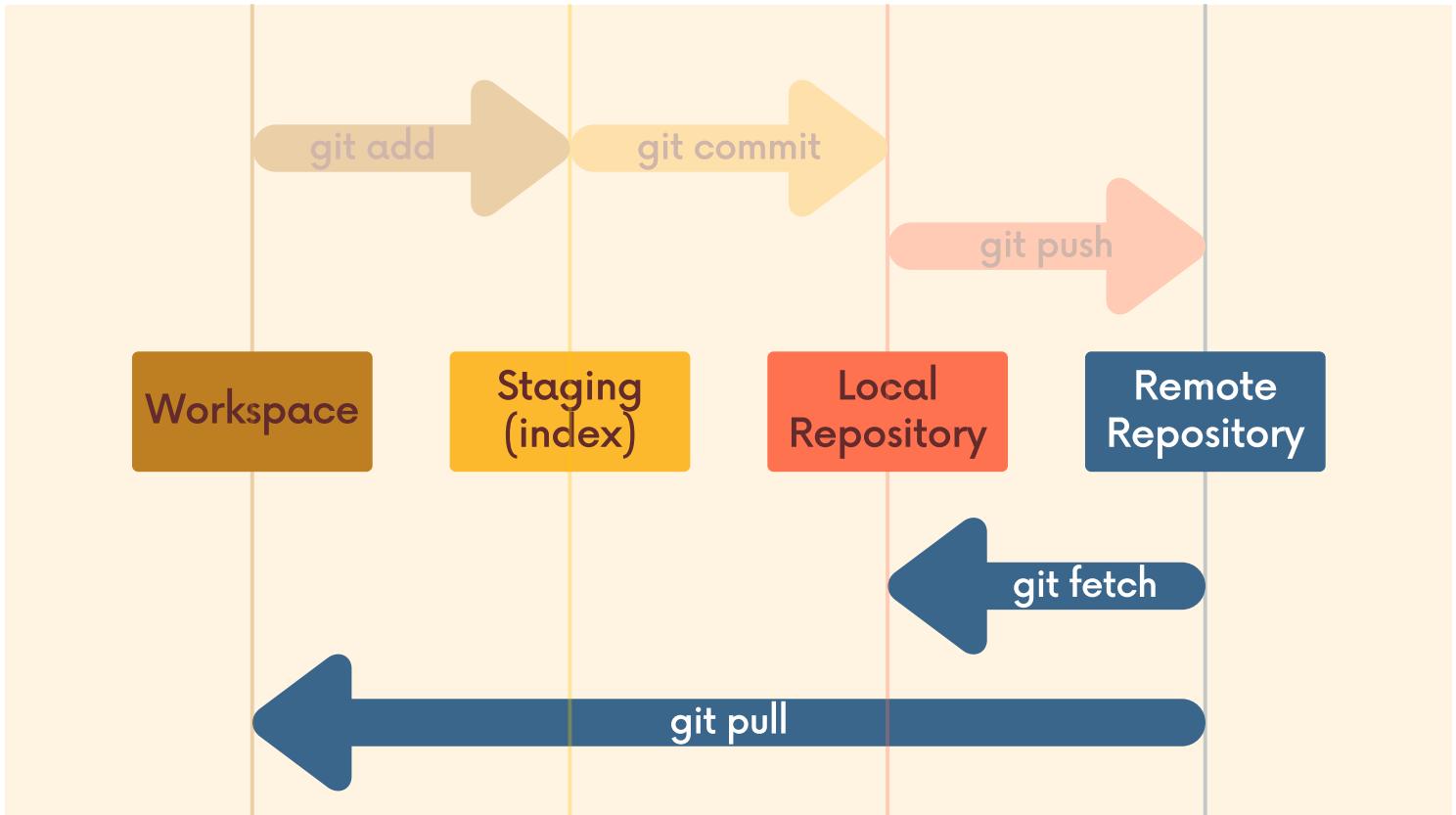
Uh oh! The remote repo has changed! A teammate has pushed up changes to the master branch, but my local repo doesn't know!

How do I get those changes???

Local







≡

Pulling

`git pull` is another command we can use to retrieve changes from 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 = git fetch + git merge

update the remote tracking branch
with the latest changes from the
remote repository

update my current branch with
whatever changes are on the remote
tracking branch



git pull

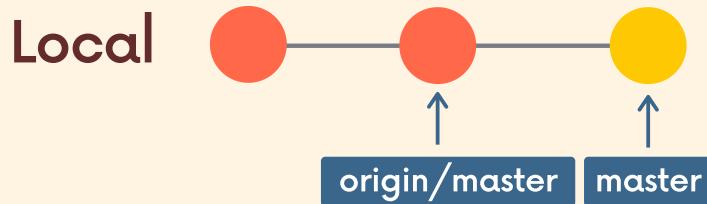
To pull, we specify the particular remote and branch we want to pull using `git pull <remote> <branch>`. Just like with git merge, it matters WHERE we run this command from. Whatever branch we run it from is where the changes will be merged into.

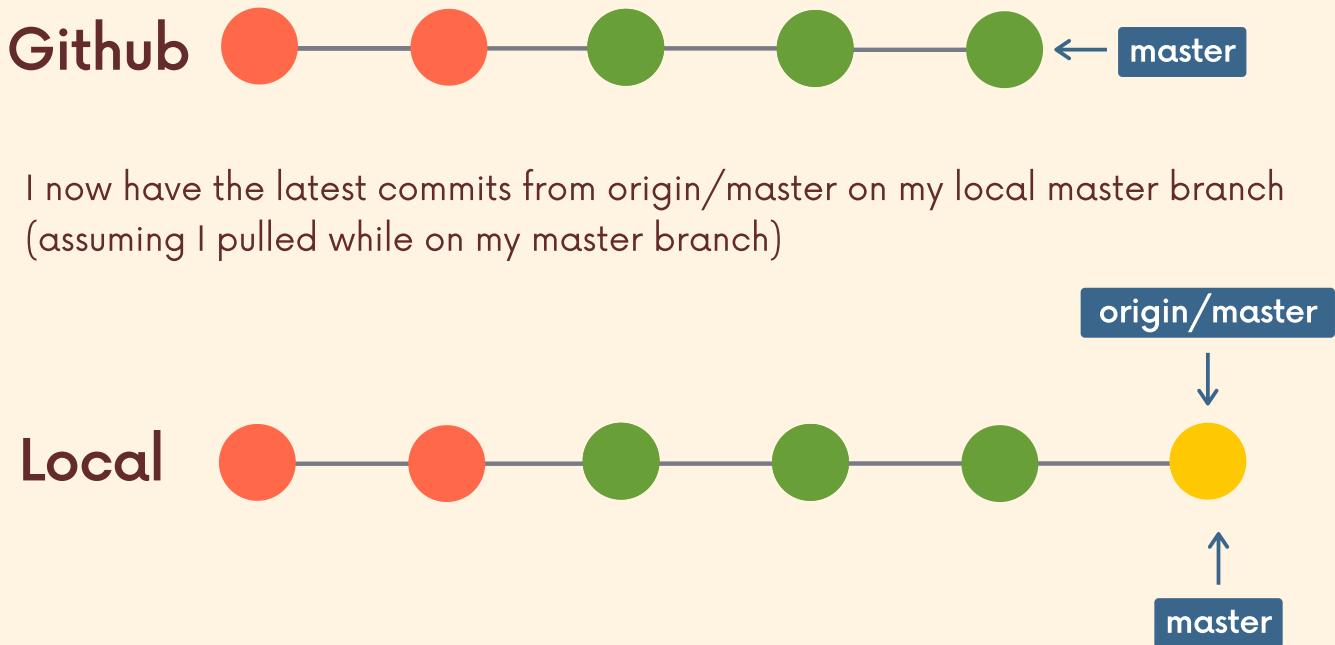
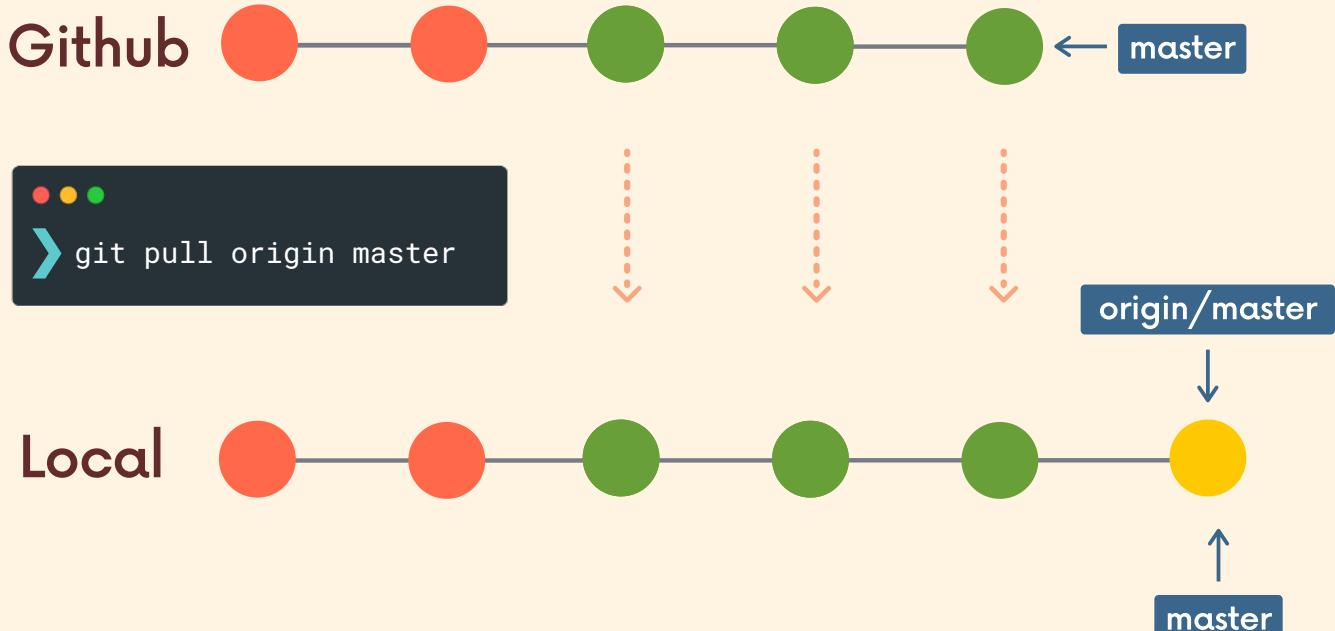
`git pull origin master` would fetch the latest information from the origin's master branch and merge those changes into our current branch.

```
git pull <remote> <branch>
```



pulls can result in merge conflicts!!







I have a commit locally that is not on Github.
When I pulled, it was merged with the new commits.
As with any other merge, this 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 don't mess with that stuff

```
git pull
```



Workspace

master

puppies

Remote

origin/master

origin/puppies

When I'm on my local
master branch...

› git pull

pulls from origin/master
automatically

Workspace

master

puppies

Remote

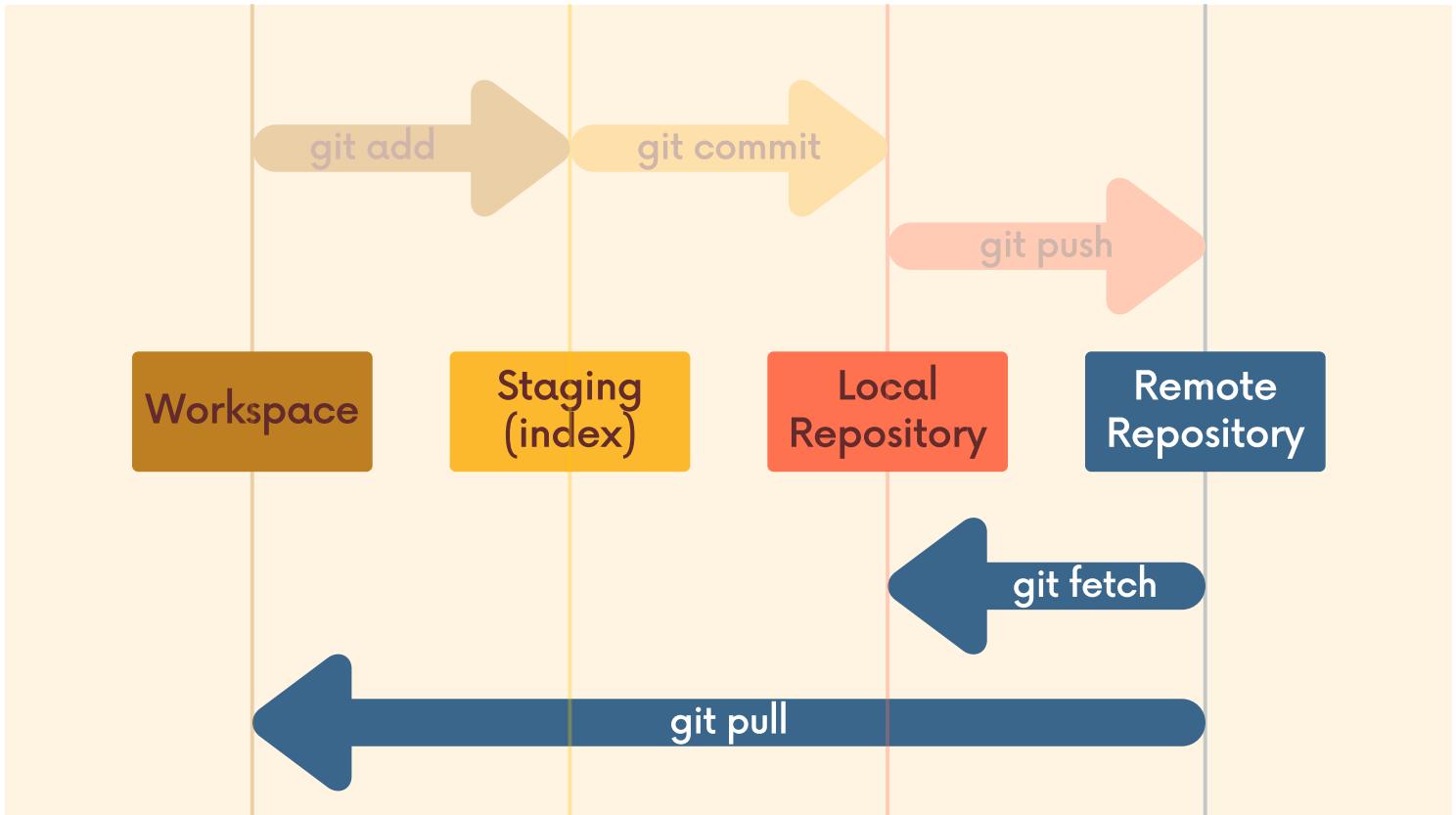
origin/master

origin/puppies

When I'm on my local
puppies branch...

› git pull

pulls from origin/puppies
automatically



☰

git fetch

- Gets changes from remote branch(es)
- Updates the remote-tracking branches with the new changes
- Does not merge changes onto your current HEAD branch
- Safe to do at anytime

git pull

- Gets changes from remote branch(es)
- Updates the current branch with the new changes, merging them in
- Can result in merge conflicts
- Not recommended if you have uncommitted changes!



12 More on GitHub

Github Odds & Ends

Public Vs.
Private Repos





Public Repos

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



Private

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





Deleting Github Repos

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



Adding Collaborators



can I collaborate
with myself?



I'll pretend to be
two different users!





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



READMEs

If you put a README in the root of your project,
Github will recognize it and automatically display
it on the repo's home page.





README.md

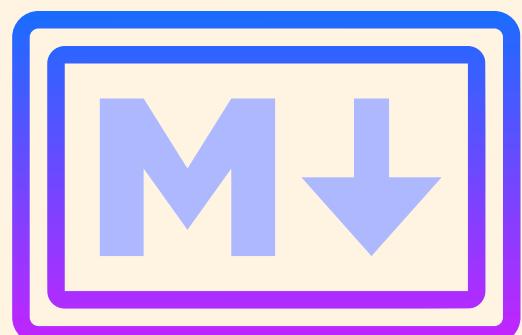
READMEs are Markdown files, ending with the .md extension. Markdown is a convenient syntax to generate formatted text. It's easy to pick up!



Markdown Syntax

Markdown syntax supports the following:

- Headings
- Text Styles (bold, italics, etc.)
- Code Blocks
- Lists (ordered, and unordered)
- Links
- Images
- And more!

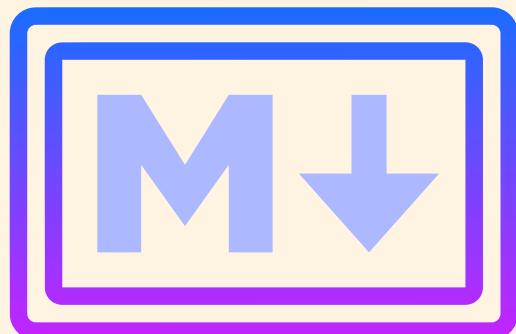


≡

Markdown Headings

In Markdown, we can write six different levels of headings, all using the # character.

```
# The Largest Heading  
## The Second Largest Heading  
### The Third Largest Heading  
...  
##### The Smallest Heading
```



Github Gists

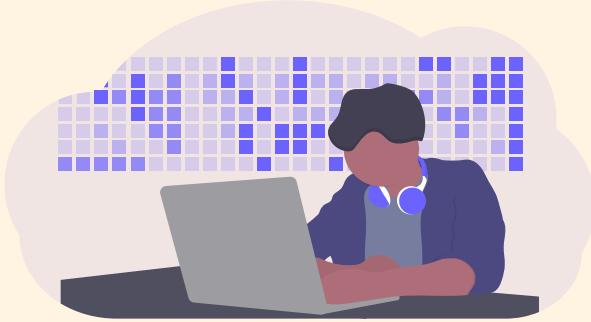
Github Gists are a simple way to share code snippets and useful fragments with others. Gists are much easier to create, but offer far fewer features than a typical Github repository.

gh 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.

static sites

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



User Site

You get one user site per Github account. This is where you could host a portfolio site or some form of personal website. The default url is based on your Github username, following this pattern: **username.github.io** though you can change this!

Project Sites

You get unlimited project sites! Each Github repo can have a corresponding hosted website. It's as simple as telling Github which specific branch contains the web content. The default urls follow this pattern: **username.github.io/repo-name**

13 git workflows (for collaboration)

git workflows (for collaboration)



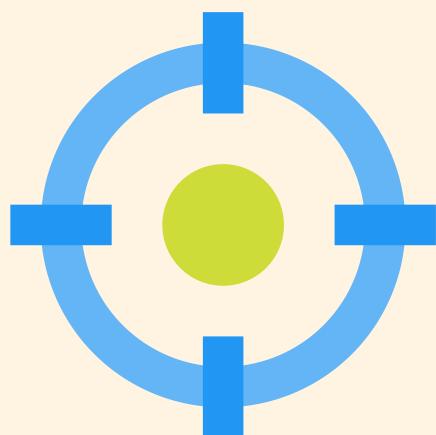
≡

Centralized 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!





Pamela clones the repo

master



David clones the repo

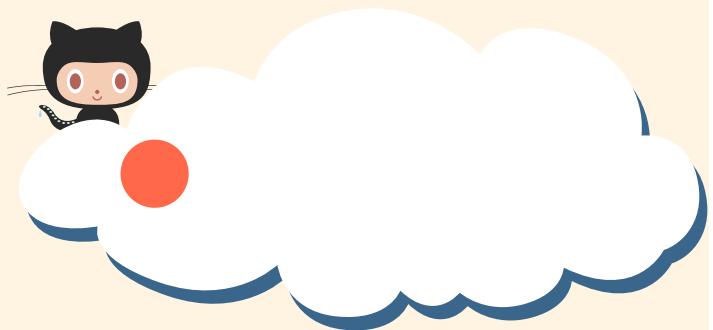
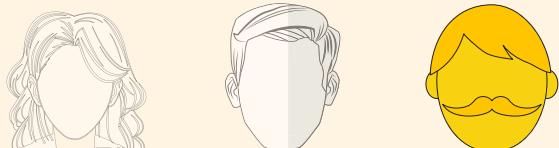
master





Forrest clones the repo

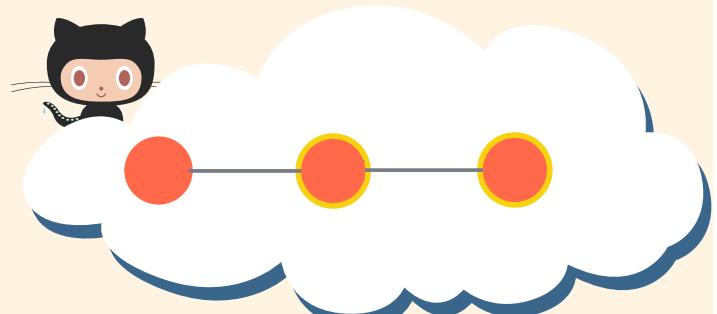
master



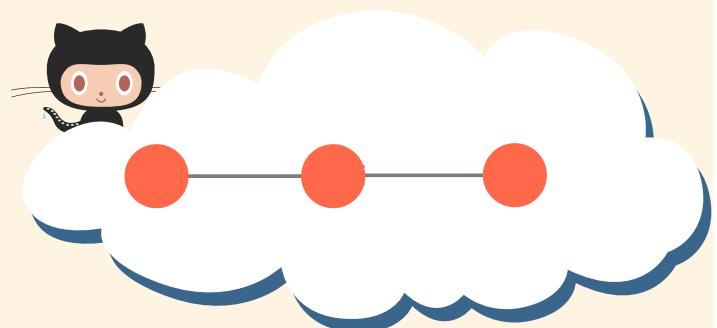
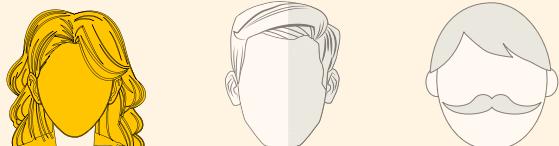
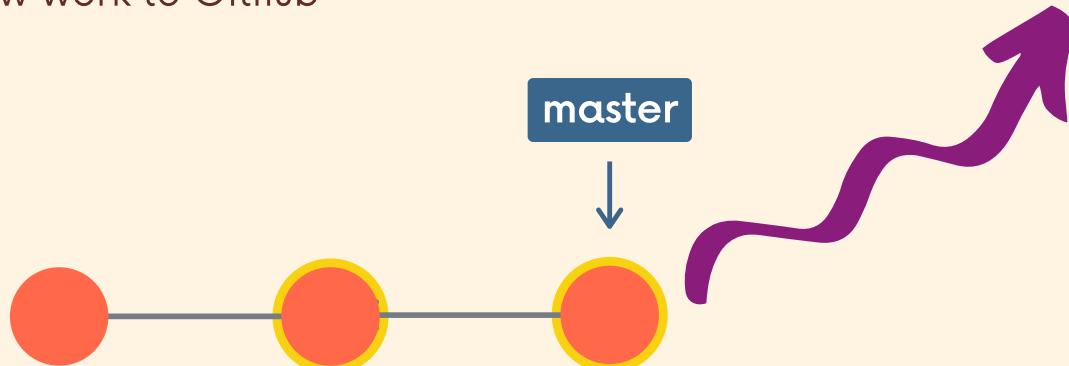
Forrest gets to work on a new feature! Adding and Committing all day long!

master

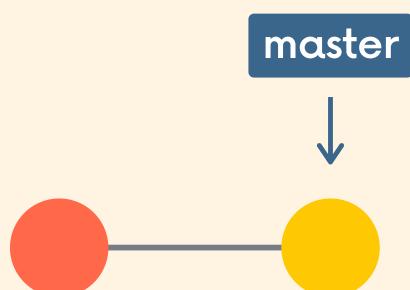




Forrest now pushes up his new work to Github



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





She tries to **push** her new work up to Github, but she runs into trouble!



master

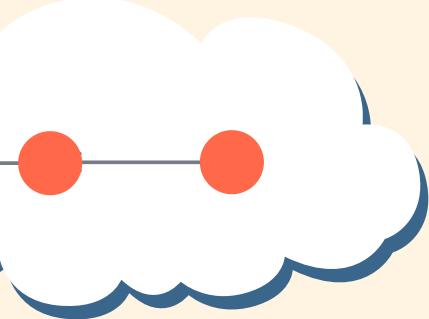


Failed to push. Updates were rejected because the tip of your current branch is behind its remote counterpart. Merge the remote changes (e.g. 'git pull') before pushing again.



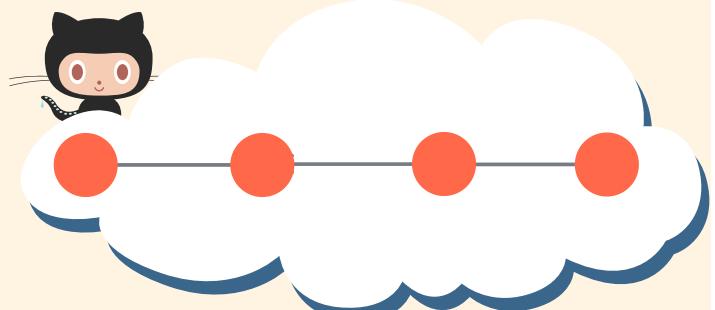
So she pulls to get the changes from origin master

Forrest's work must be merged in. Hopefully this goes relatively smoothly!

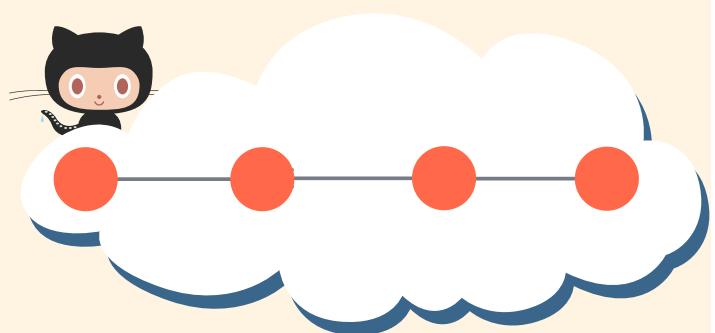
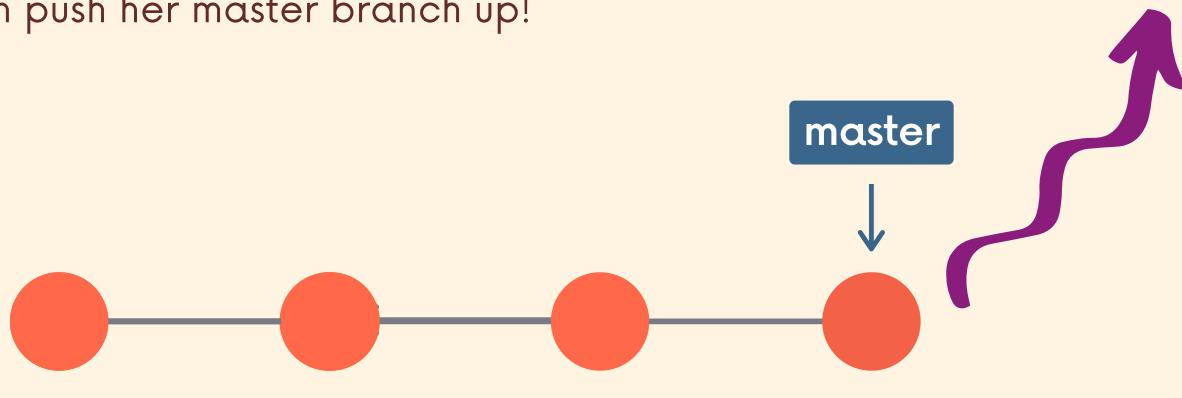


master



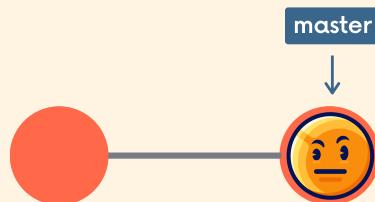


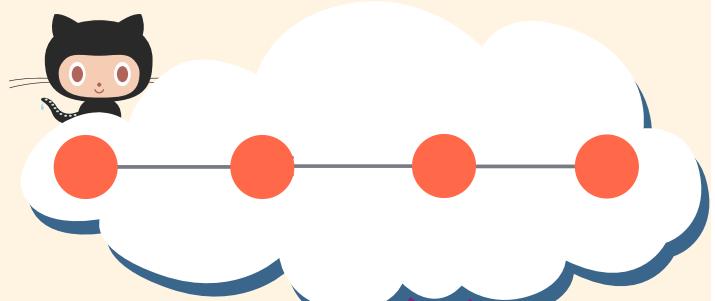
Now that she has merged the latest work from Github, she can push her master branch up!



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.

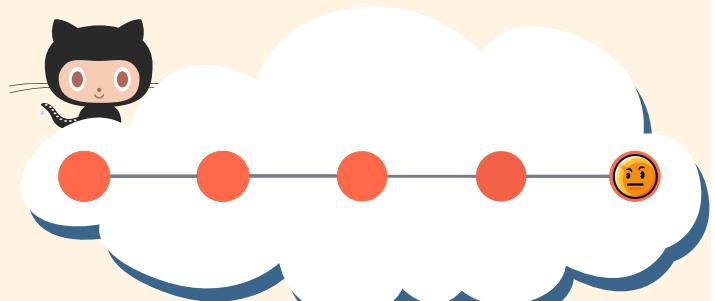




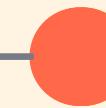
Before he can even share these iffy commits, he has to **pull** from Github and **merge** them in to master



master



Now he can finally push his work up to Github. His teammates can pull to get his new commits.



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 a feature together with another teammate is to push incomplete code to master. Other teammates now have broken code...



Enter Feature Branches

DON'T WORK ON MASTER SILLY GOOSE!

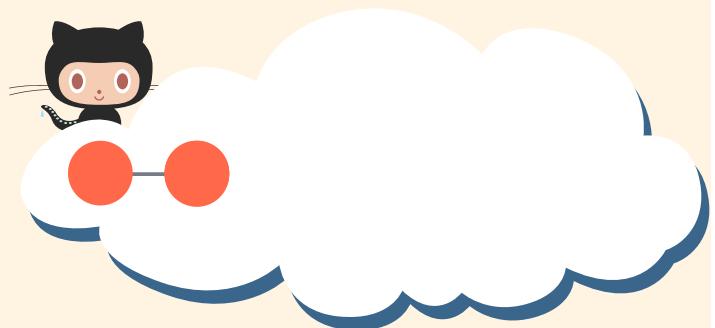




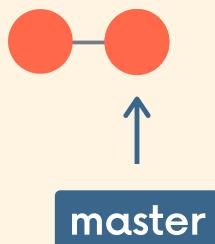
Feature Branches

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 a 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)

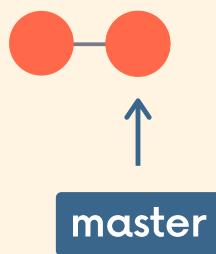


Forrest clones the repo

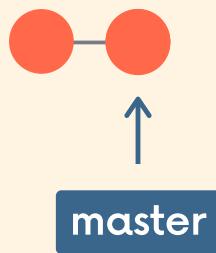




Pamela clones the repo

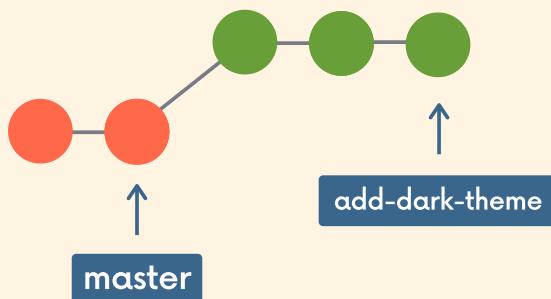


David clones the repo

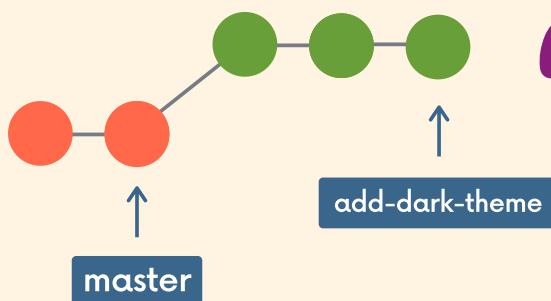
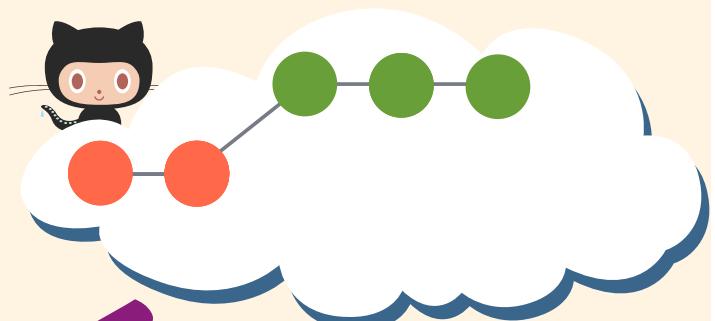


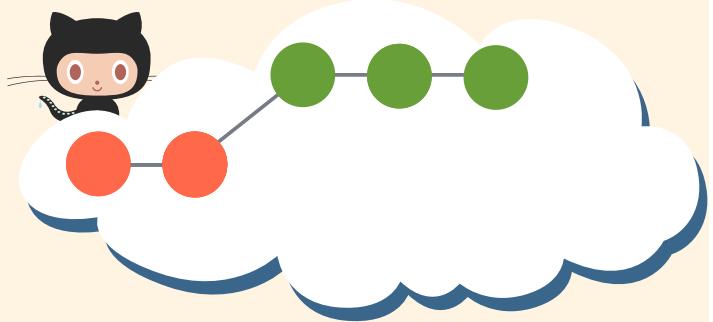


David starts work on a new feature. He does all this work on a separate branch!

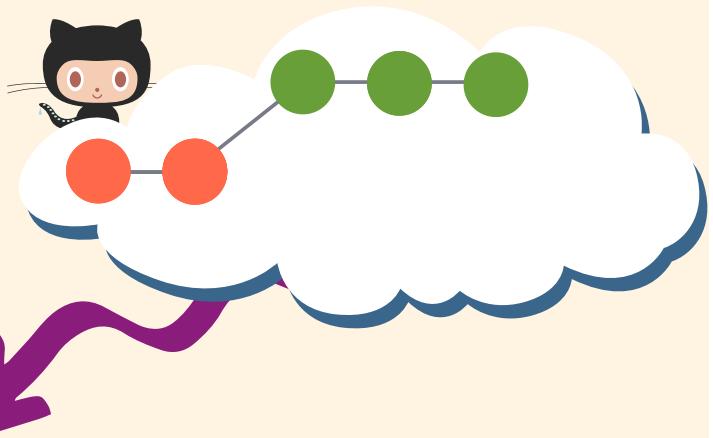
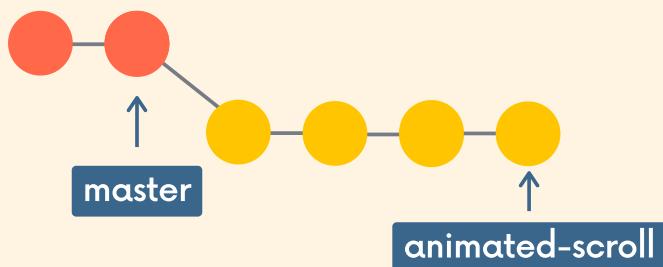


David wants Pamela to take a look at his new feature. Instead of merging it into master, he just pushes his feature branch up to Github!

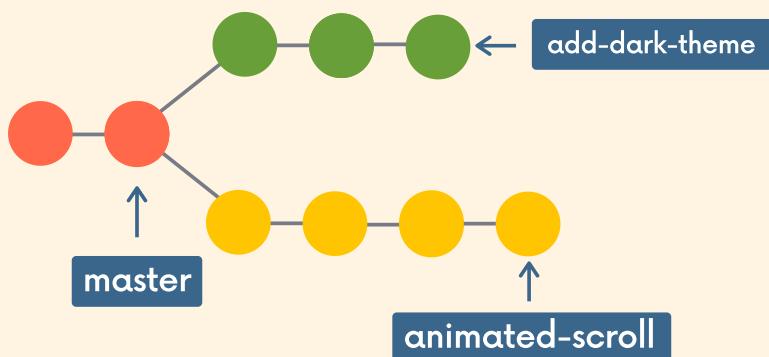


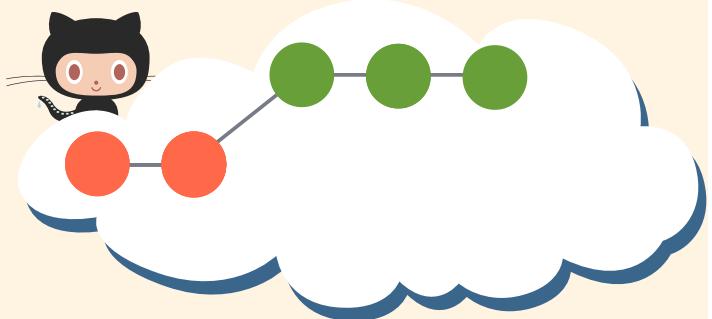


Pamela is hard at work on her own new feature. Just like everyone else, she's working on a separate feature branch rather than master.

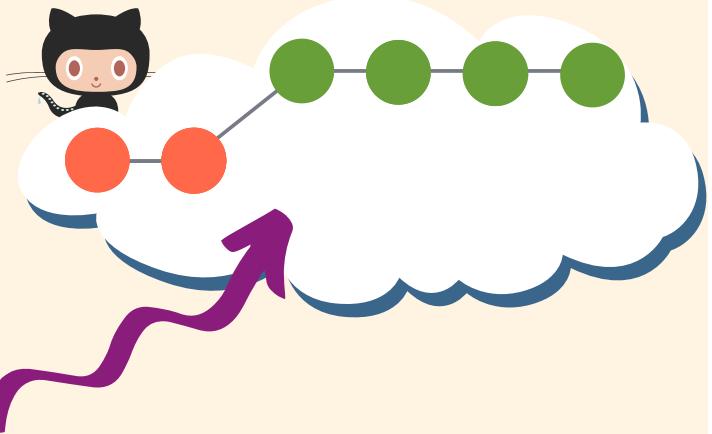
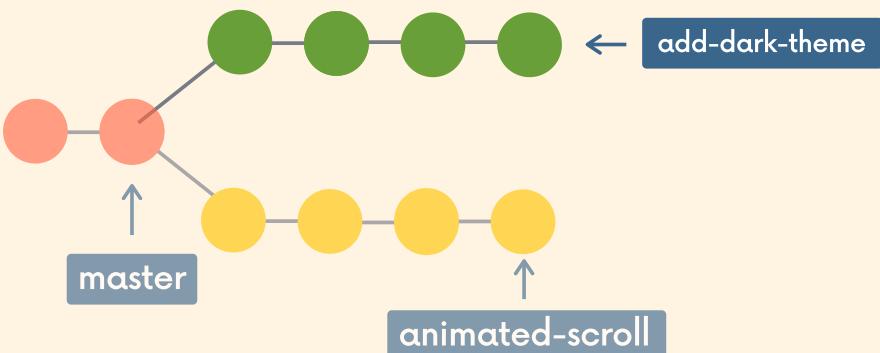


Pamela hears from David that he wants her to take a look at his new work. She pulls down his feature branch from Github.

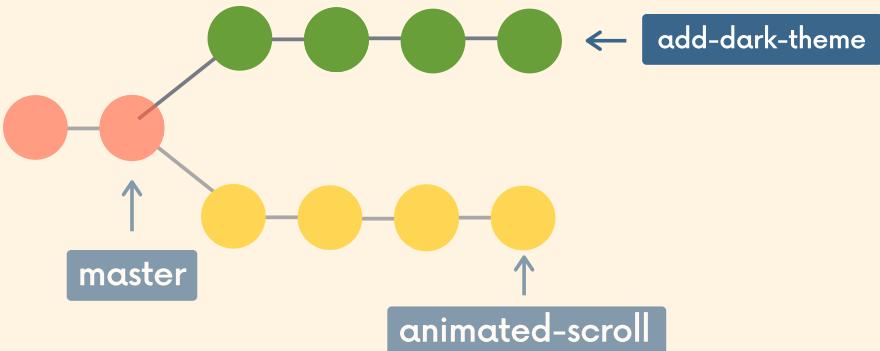




Pamela takes a look at the code and makes a couple improvements of her own. She adds/commits on the same feature branch.

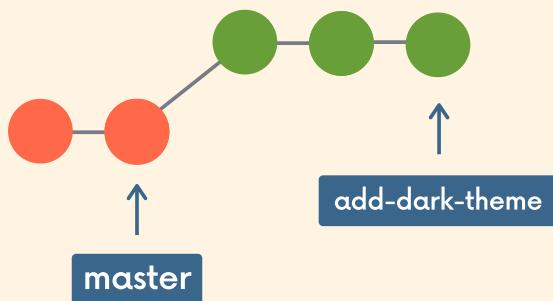
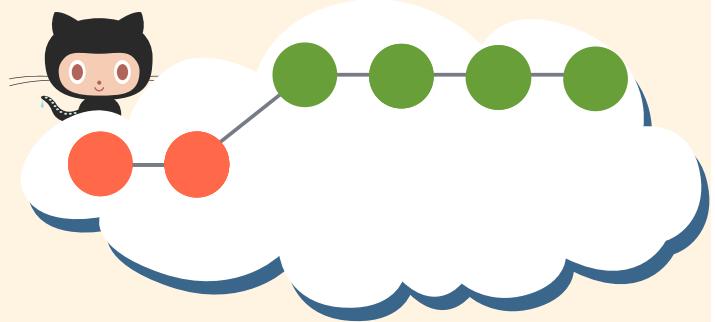


Pamela pushes up her new work on the add-dark-theme feature branch so that David can pull them down.

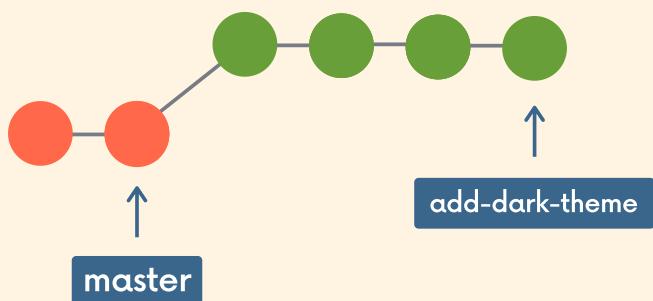
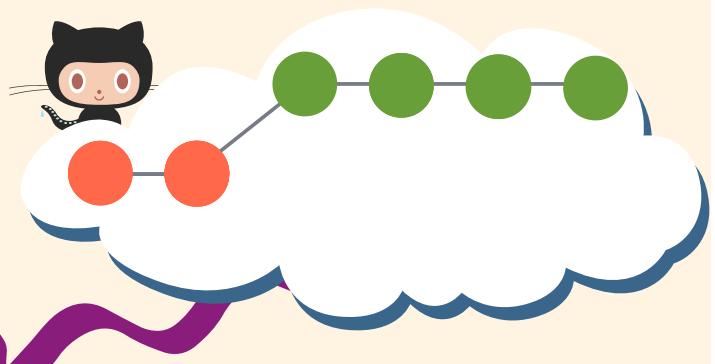


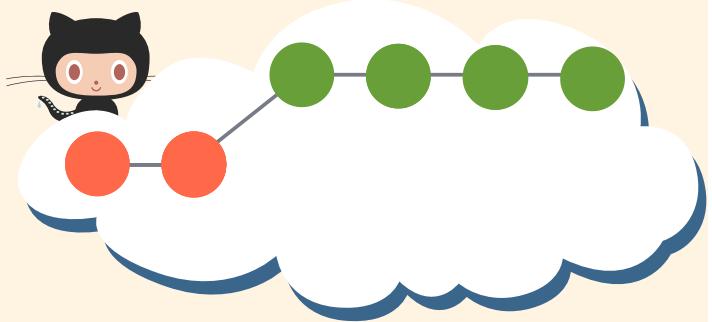


David returns to work the next morning. After an hour on reddit he decides he should actually do some work.

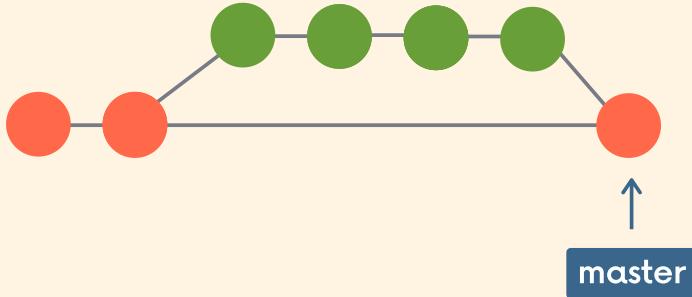


David fetches from Github and sees that there is new work on the add-dark-theme branch. He pulls the changes down and continues work.

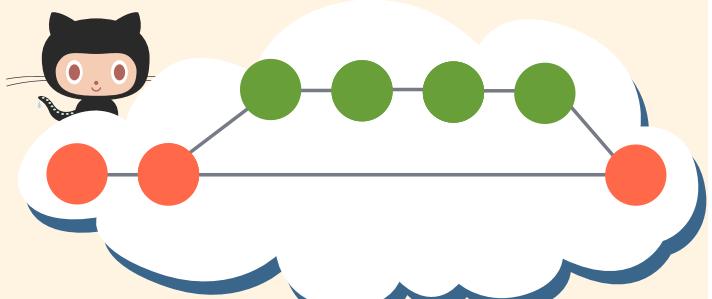




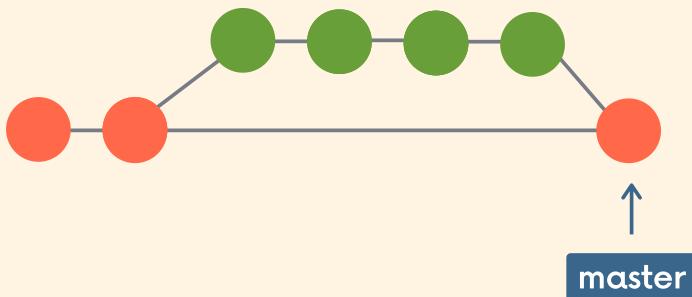
David decides he is happy with the new feature, so he merges it into master!



*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!



David pushes up the updated master branch to Github. The others can now pull down the changes.





Feature Branch Naming

There are many different approaches for naming feature branches. Often you'll see branch names that include slashes like `bug/fix-scroll` or `feature/login-form` or `feat/button/enable-pointer-events`

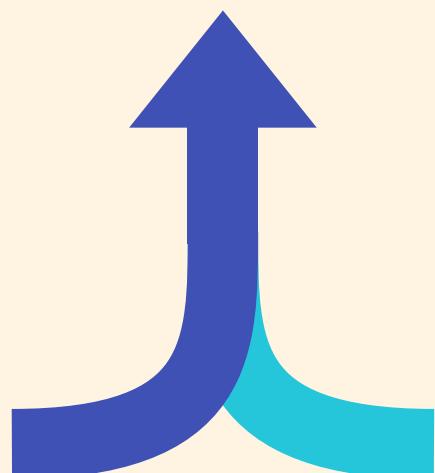
Specific teams and projects usually have their own branch naming conventions. To keep these slides simple and concise, I'm just going to ignore those best-practices for now.



Merging In Feature Branches

At some point new the 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 Requests are a feature built in to products like Github & Bitbucket. **They are not native to Git itself.**

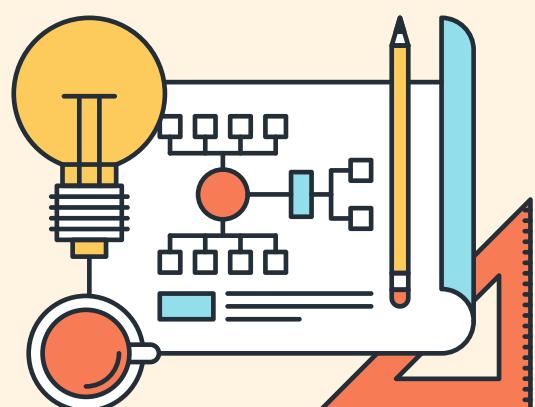
They allow developers to alert team-members to new work that needs to be reviewed. They provide a mechanism to approve or reject the work on a given branch. They also help facilitate discussion and feedback on the specified commits.

"I have this new stuff I want to merge in to the master branch...what do you all think about it?"

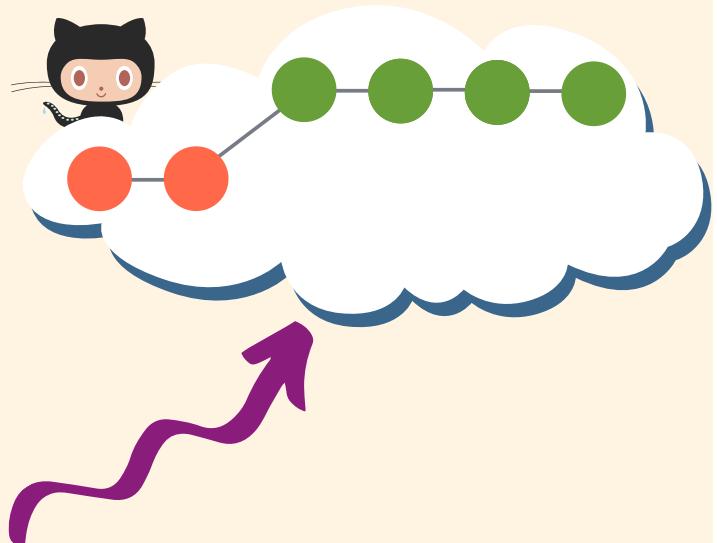
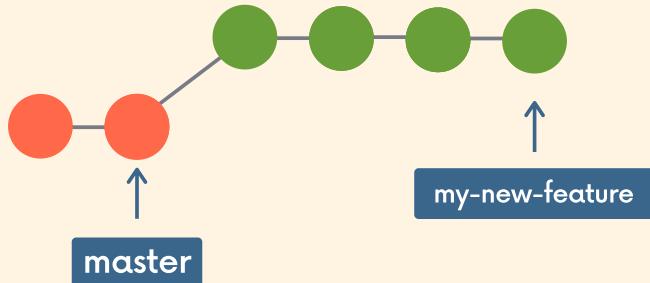


The Workflow

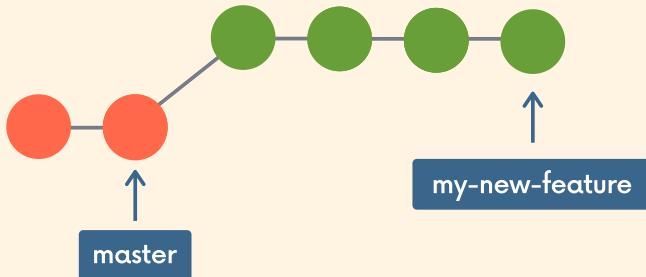
1. Do some work locally on a feature branch
2. Push up the feature branch to Github
3. Open a pull request using the feature branch just pushed up to Github
4. Wait for the PR to be approved and merged. Start a discussion on the PR. This part depends on the team structure.



I push my feature branch up to Github, so that I can open a Pull Request



My Github



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.

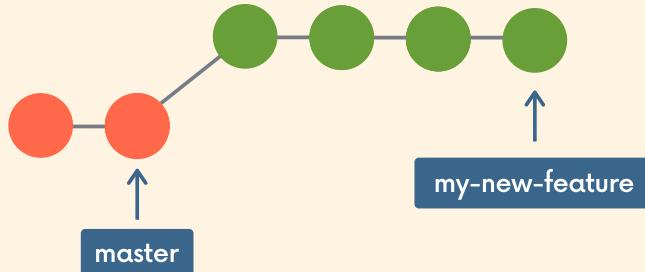
Pull request Compare

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



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 d3f

| | | |
|------------------------------------|----------------------|----------------|
| anotherNewFile.txt | add another new file | 32 seconds ago |
| newfeature.txt | add new feature | 1 minute ago |
| playlist.txt | merge | 5 days ago |

I click the PR button

My Github

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: [master](#) ▾ ← compare: [my-new-feature](#) ▾ ✓ Able to merge. These branches can be automatically merged.



My new feature

[Write](#) [Preview](#)

H B I \equiv \leftrightarrow \oplus \equiv \oplus \otimes \oplus \otimes \oplus \otimes

Leave a comment

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

[Create pull request](#)

My Boss's Github...

A screenshot of a GitHub pull request comment interface. At the top, there are 'Write' and 'Preview' tabs, along with various rich text editing icons (bold, italic, etc.). Below the text area, there is a placeholder for attachments with the message 'Attach files by dragging & dropping, selecting or pasting them.' and a note 'No file chosen'. At the bottom right are two buttons: 'Close with comment' (disabled) and 'Comment'.

Hey Colt, this is on the right track but it needs a bit more work before it's ready to be merged in. Please fix x & y and z and then report back. Add those commits to this PR. Ping me when it's ready. Thanks!

Attach files by dragging & dropping, selecting or pasting them.
No file chosen

Close with comment Comment

My boss leaves some feedback for me. She asks me to make a couple changes before she merges the pull request.

A screenshot of a GitHub pull request thread. It shows a comment from 'Colt' and a reply from 'Boss'. Below the comments, two commits are listed: 'add indie songs' and 'add hot chip'. A final comment from 'Colt' at the bottom indicates he will fix the issues.

Colt commented 2 minutes ago • edited

Added in new feature. This should be descriptive and helpful.

Colt Steele added 2 commits 6 days ago

- o add indie songs 399d0b9
- o add hot chip 75e6c6d

Boss commented 44 seconds ago

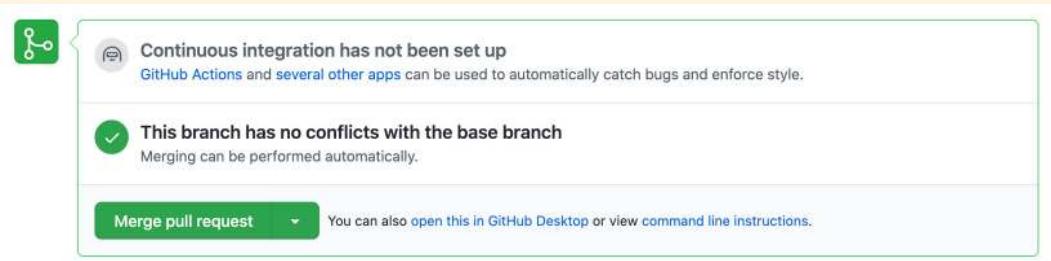
Hey Colt, this is on the right track but it needs a bit more work before it's ready to be merged in. Please fix x & y and z and then report back. Add those commits to this PR. Ping me when it's ready. Thanks!

Colt commented 21 seconds ago

Ok, sounds good! I'll get to work on fixing x & y & z!

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!

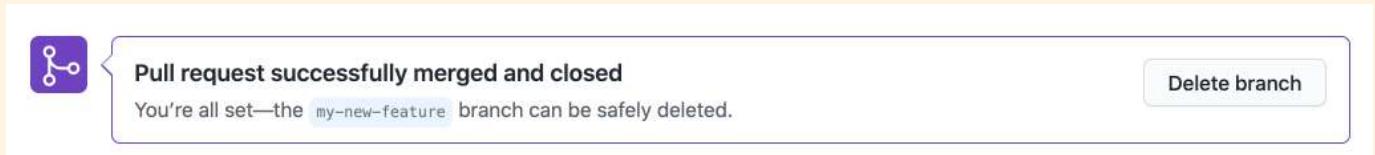
My Boss's Github...



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

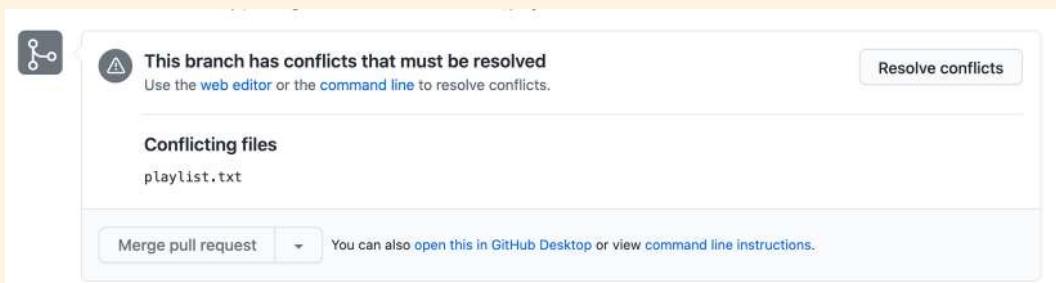
The above text will be used in the resulting merge commit.

My Boss's Github...



The changes from the my-new-feature branch have now been merged into the master branch!!!

My Boss's Github...



Just like any other merge, **sometimes there are conflicts** that need to be resolved when merging a pull request. This is fine. Don't panic.

You can perform the merge and fix the conflicts on the command line like normal, or you can use GitHub's interactive editor.

My Boss's Local Machine

My boss can merge the branch and resolve the conflicts locally...

Switch to the branch in question. Merge in master and resolve the conflicts.

```
...  
git fetch origin  
git switch my-new-feature  
git merge master  
fix conflicts!
```

Switch to master. Merge in the feature branch (now with no conflicts). Push changes up to Github.

```
...  
git switch master  
git merge my-new-feature  
git push origin master
```

Don't Worry

Github Gives You Instructions If You Forget What To Do!



 Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

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

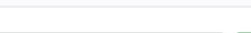
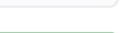
[Merge pull request](#) You can also [open this in GitHub Desktop](#) or view command line instructions.

 ✓ Create a merge commit
All commits from this branch will be added to the base branch via a merge commit.

Squash and merge
The 3 commits from this branch will be combined into one commit in the base branch.

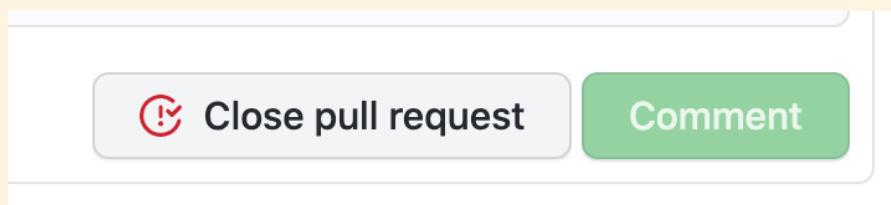
Rebase and merge
The 3 commits from this branch will be rebased and added to the base branch.

When merging, we can perform a Squash & Merge which will reduce all the commits from the feature branch down into a single commit on master.

My Boss's Github...

My boss can also decide to close my PR instead :(





Recap-ing Pull Requests

Pull Requests are a fancy way of requesting changes from one branch be merged into another branch.

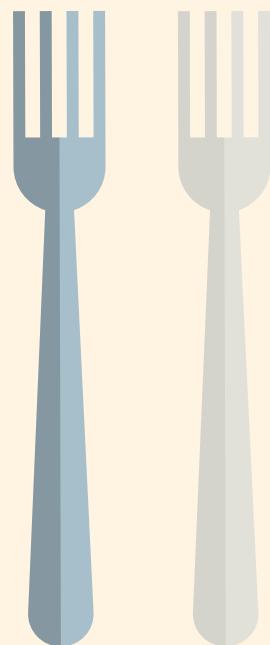
Tools like Github & Bitbucket allow us to generate pull requests via an online interface. Team members can then view the changes and decide to merge them in or reject them. PR's also provide a place to discuss the changes and provide feedback.



Fork & Clone: Another Workflow

The "fork & clone" workflow is different from anything we've seen so far. Instead of just one centralized 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 (and similar tools) 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.



This repo is not mine. I want a copy!

[aapatre / Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE](#)

Code Issues 3 Pull requests 1 Discussions Actions Projects 1 Wiki Security Insights

master 2 branches 6 tags Go to file Add file Code

Commits

| File | Description | Time Ago |
|--------------------------|---|--------------|
| Update README.md | Merge pull request #146 from aapatre/develop | 6 days ago |
| .github | Wait for enroll button to be clickable after selecting state/province | 10 days ago |
| core | Adding unittests | 13 days ago |
| tests | Adding more unittests | 23 days ago |
| .coveragerc | Create .deepsource.toml | 16 days ago |
| .deepsource.toml | fixed deepsource ignore | 16 days ago |
| .gitignore | Adding restyled configuration | last month |
| .restyled.yaml | fixed typo | 6 days ago |
| CHANGELOG.md | Create CODE_OF_CONDUCT.md | 15 days ago |
| CODE_OF_CONDUCT.md | Create CONTRIBUTING.md | 15 days ago |
| CONTRIBUTING.md | Initial commit | 3 months ago |
| LICENSE | Update README.md | 4 days ago |
| README.md | Update SECURITY.md | 15 days ago |
| SECURITY.md | Added logging | 13 days ago |
| logconfig.ini | Create pull_request_template.md | 15 days ago |
| pull_request_template.md | Bump version in pyproject | 18 days ago |
| pyproject.toml | Adding more OS tests | 15 days ago |
| pytest.ini | | |

About

Do you want to LEARN NEW STUFF for FREE? Don't worry, with the power of web-scraping and automation, this script will find the necessary Udemy coupons & enroll you for PAID UDEMY COURSES, ABSOLUTELY FREE!

Readme

GPL-3.0 License

Releases 6

v1.0.0 (Latest) 6 days ago + 5 releases

Packages

No packages published

Contributors 11

I click the "fork" button



S aapatre / Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE

Code Issues 3 Pull requests 1 Discussions Actions Projects 1 Wiki Security Insights

master 2 branches 6 tags Go to file Add file Code

aapatre Update README.md 2 ✓ 35e5cb1 4 days ago 345 commits

- .github Merge pull request #146 from aapatre/develop 6 days ago
- core Wait for enroll button to be clickable after selecting state/province 10 days ago
- tests Adding unittests 13 days ago
- .coveragerc Adding more unittests 23 days ago
- .deepsource.toml Create .deepsource.toml 16 days ago
- .ignore fixed deepsource ignore 16 days ago
- .restyled.yaml Adding restyled configuration last month
- CHANGELOG.ind Fixed type 6 days ago
- CODE_OF_CONDUCT.md Create CODE_OF_CONDUCT.md 15 days ago
- CONTRIBUTING.md Create CONTRIBUTING.md 15 days ago
- LICENSE Initial commit 3 months ago
- README.md Update README.md 4 days ago
- SECURITY.md Update SECURITY.md 15 days ago
- logconfig.ini Added logging 13 days ago
- pull_request_template.md Create pull_request_template.md 15 days ago
- pyproject.toml Bump version in pyproject 18 days ago
- pytest.ini Adding more OS tests 15 days ago

About

Do you want to LEARN NEW STUFF for FREE? Don't worry, with the power of web-scraping and automation, this script will find the necessary Udemy coupons & enroll you for PAID UDEMY COURSES, ABSOLUTELY FREE!

Readme GPL-3.0 License

Releases 6

v1.0.0 (Latest) 6 days ago + 5 releases

Packages

No packages published

Contributors 11

Watch 52 Star 1k Fork 155



Watch 52 Star 1k Fork 155

Now I have my very own copy!

This branch is even with aapatre:master.

About

Do you want to LEARN NEW STUFF for FREE? Don't worry, with the power of web-scraping and automation, this script will find the necessary Udemy coupons & enroll you for PAID UDEMY COURSES, ABSOLUTELY FREE!

Releases

Packages

No packages published
Publish your first package

Languages

Python 100.0%

The original repo...

aapatre / Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE

My newly-created fork...

Colt / Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE

forked from aapatre/Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE

Now What?

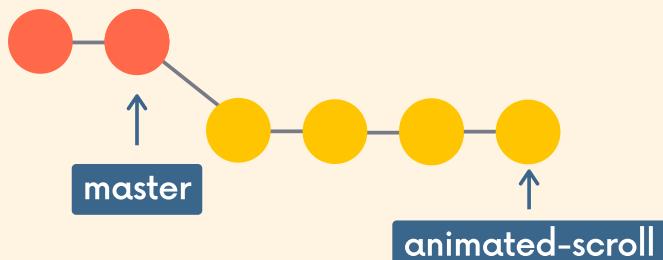
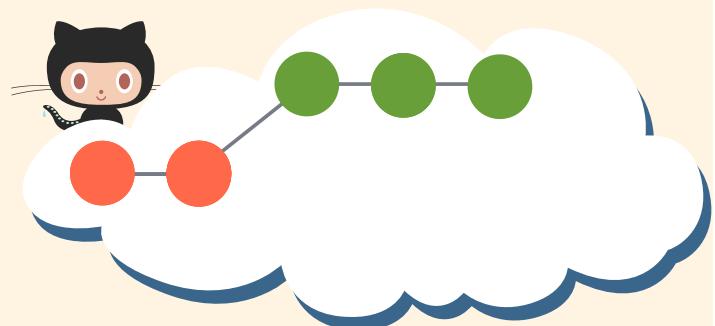
Now that I've forked, I have my very own copy of the repo where I can do whatever I want!

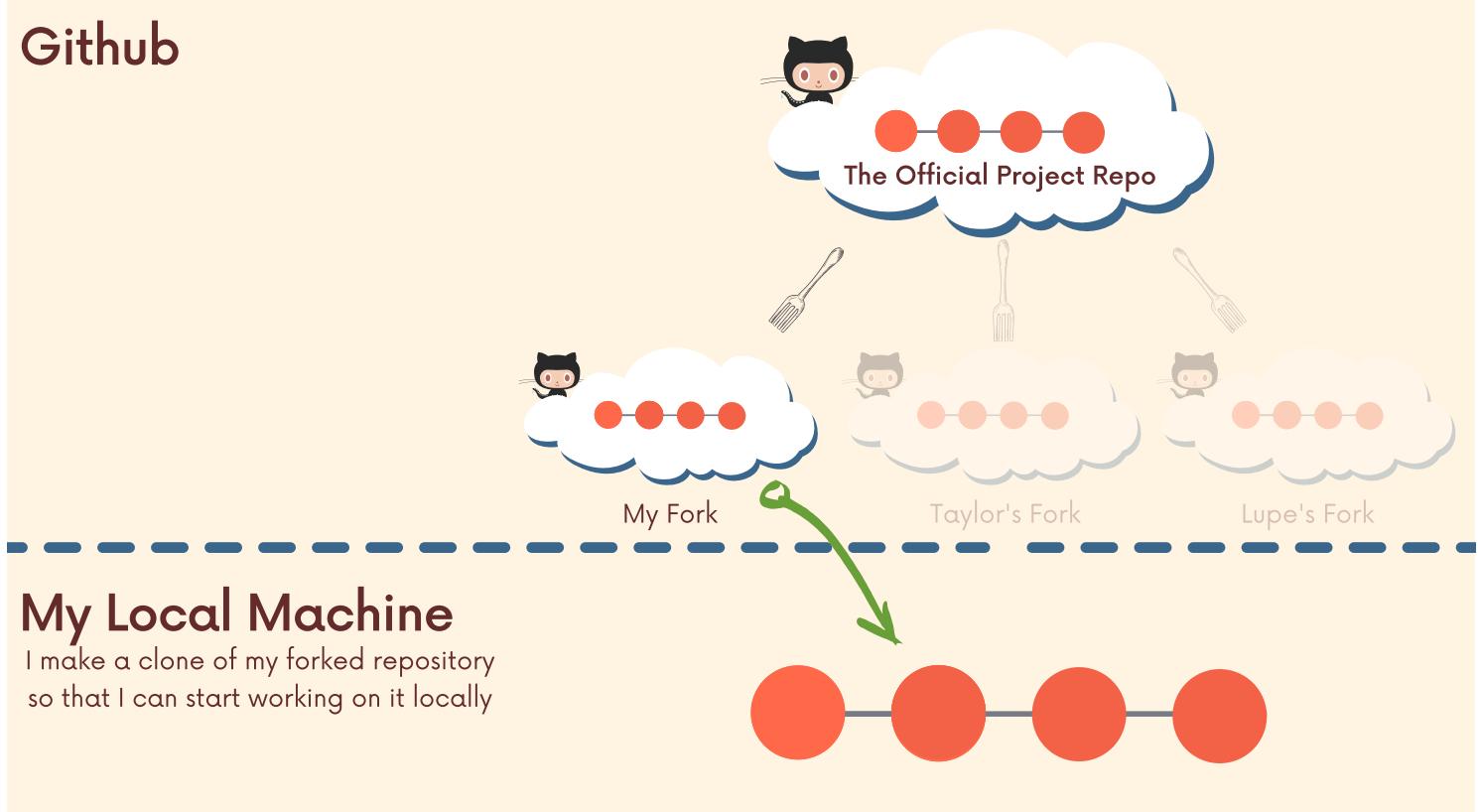
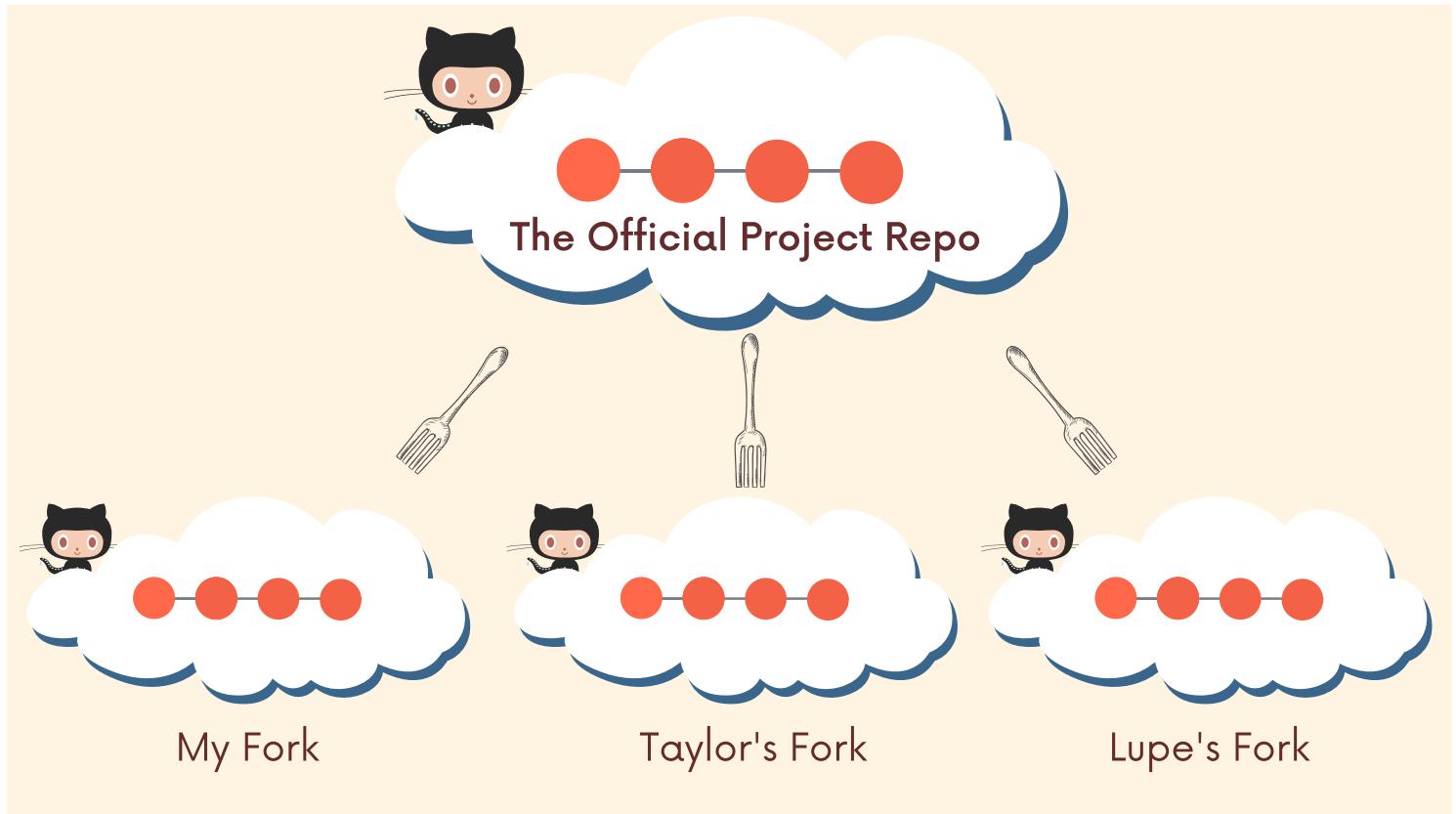
I can clone my fork and make changes, add features, and break things without fear of disturbing the original repository.

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

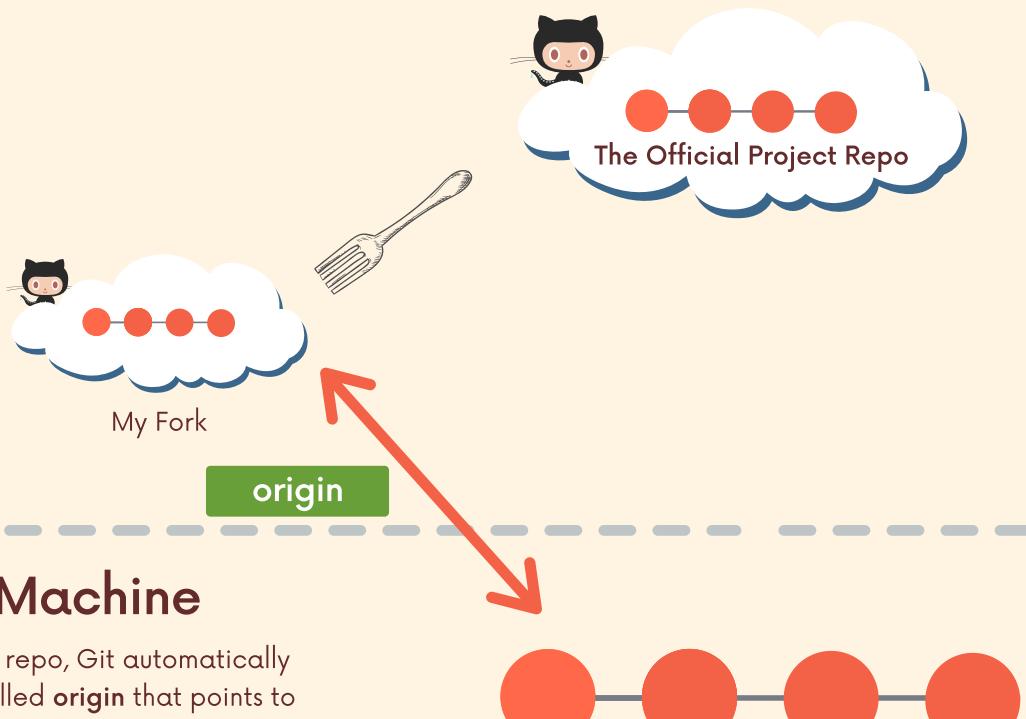


Pamela is hard at work on her own new feature. Just like everyone else, she's working on a separate feature branch rather than master.

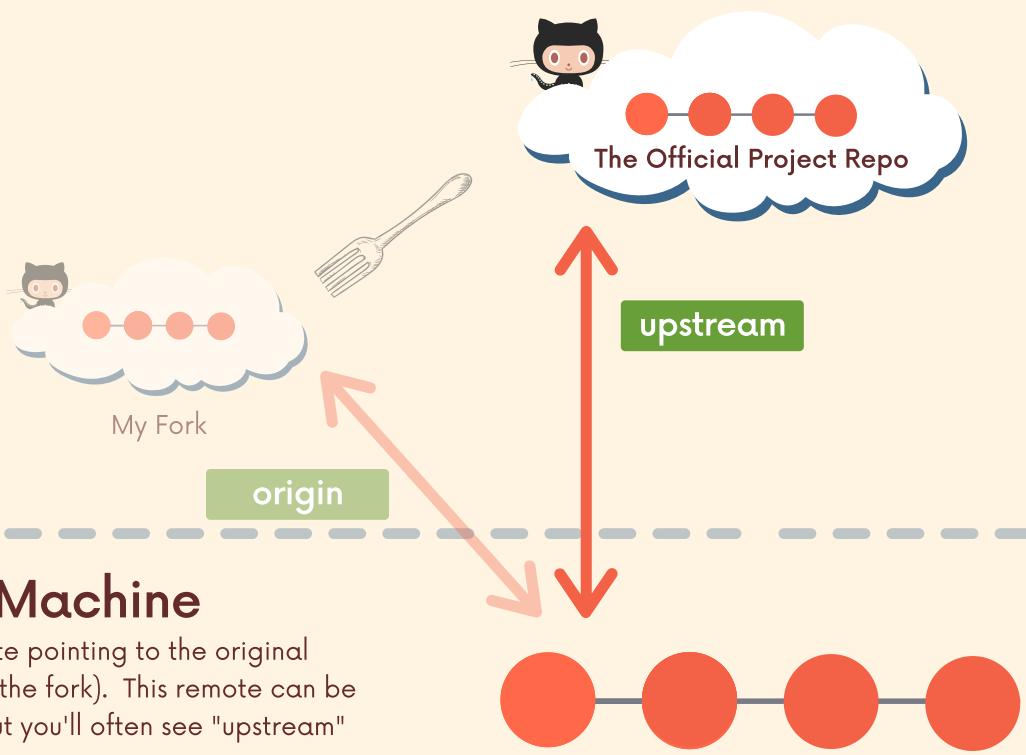




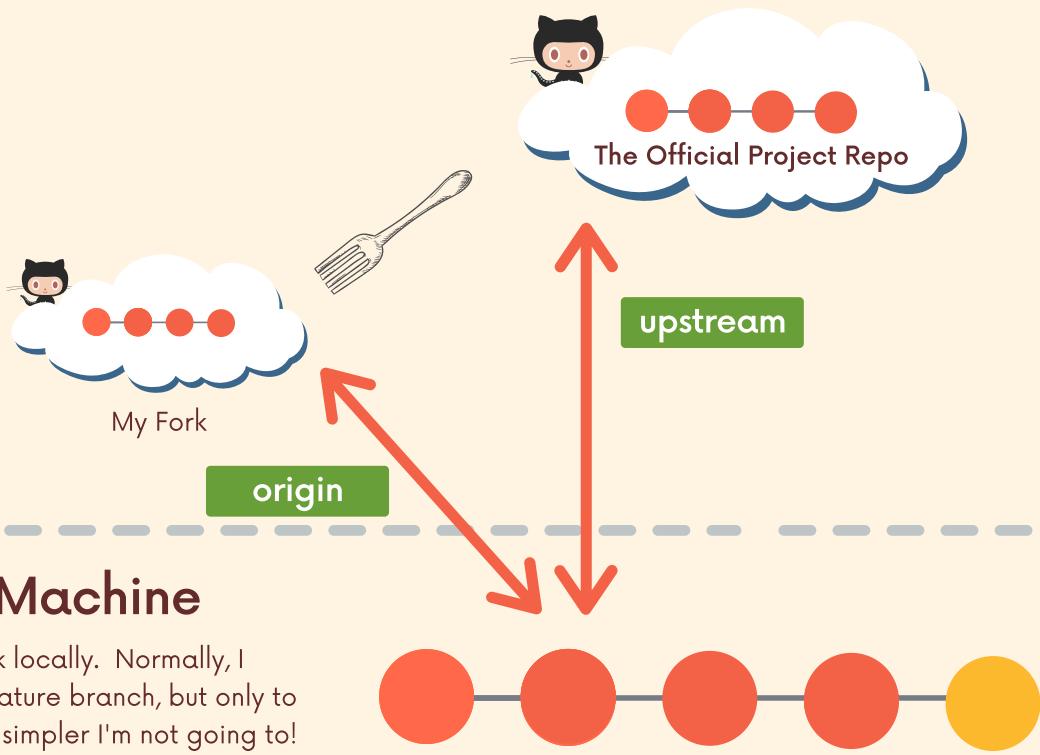
Github



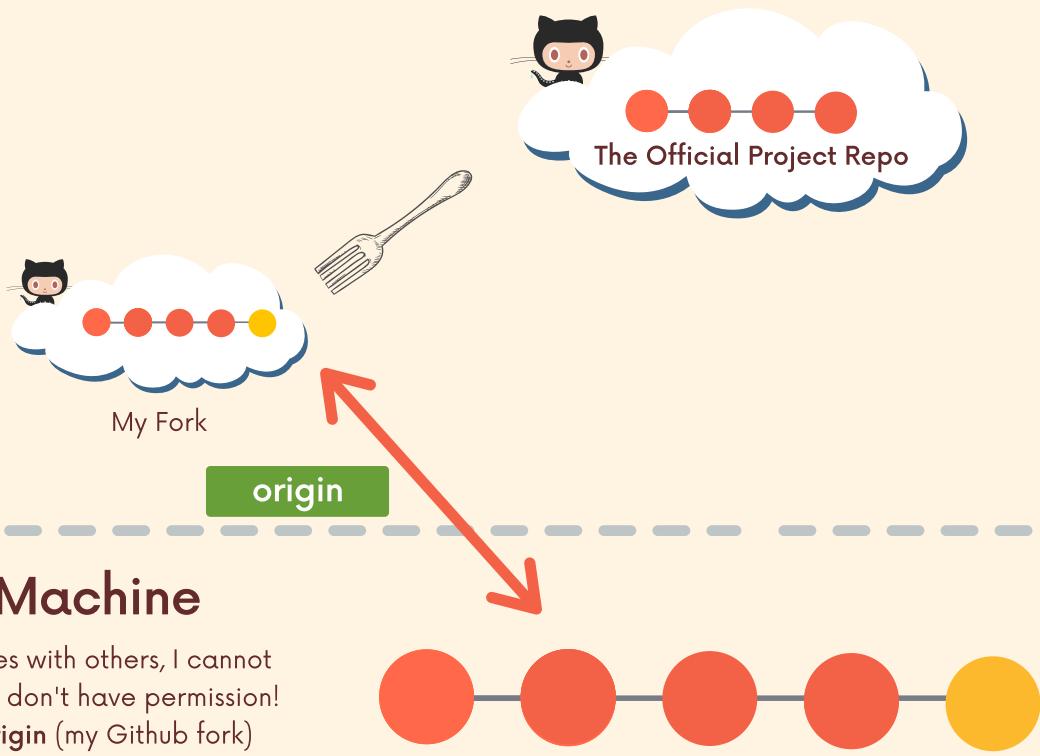
Github



Github

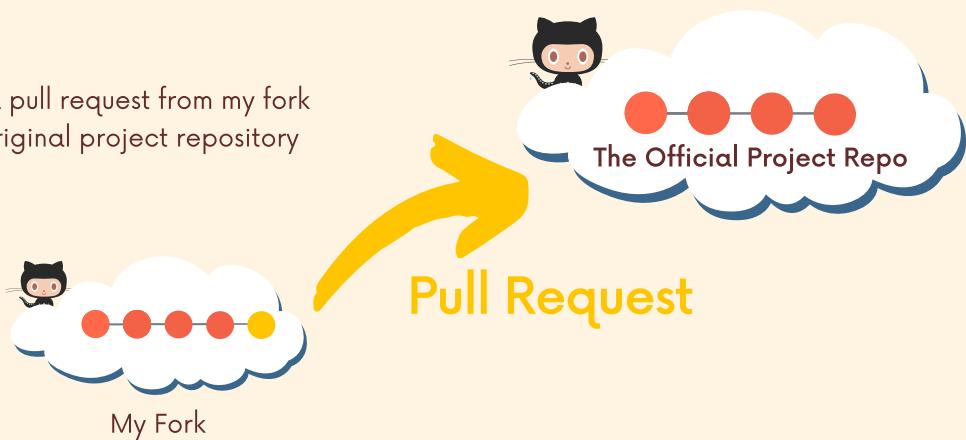


Github



Github

Next, I can make a pull request from my fork on Github to the original project repository



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!

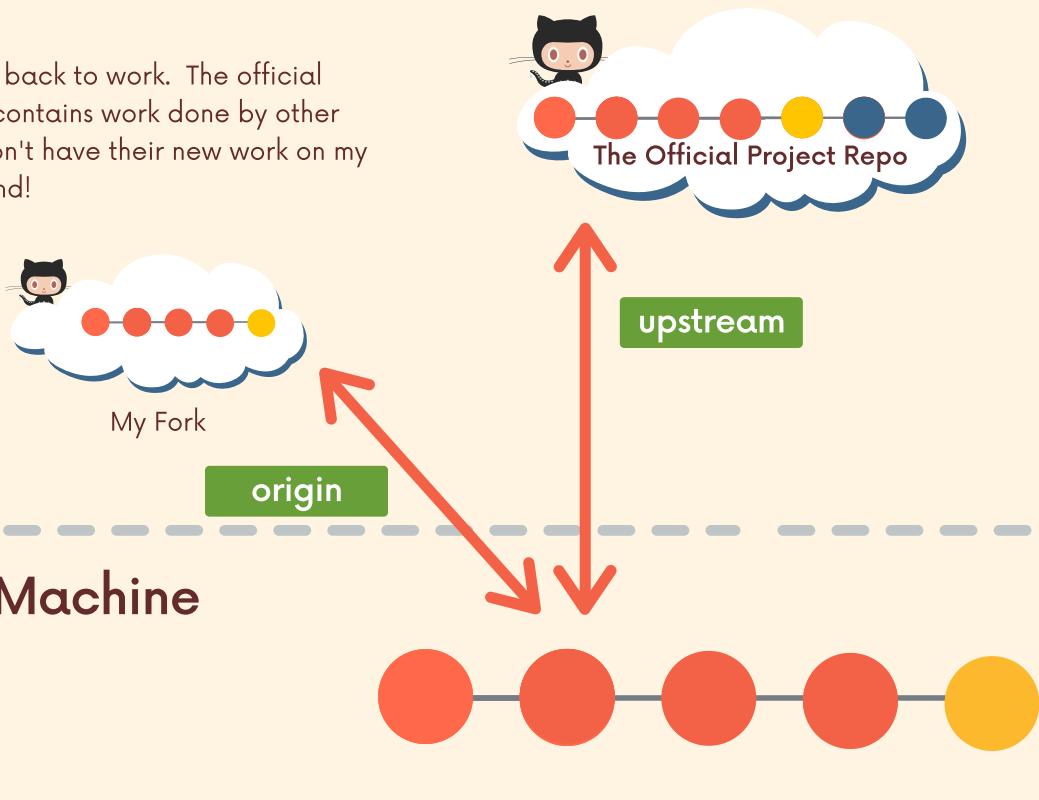


My Local Machine



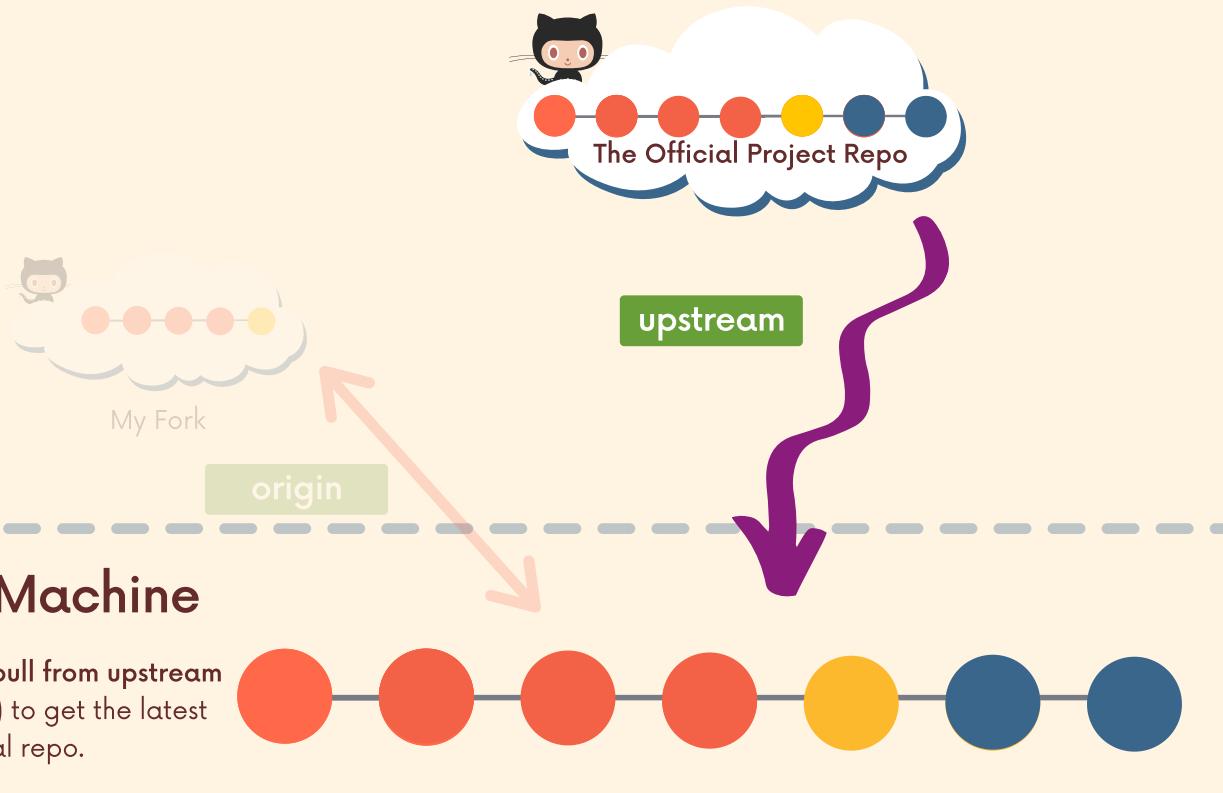
Github

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!

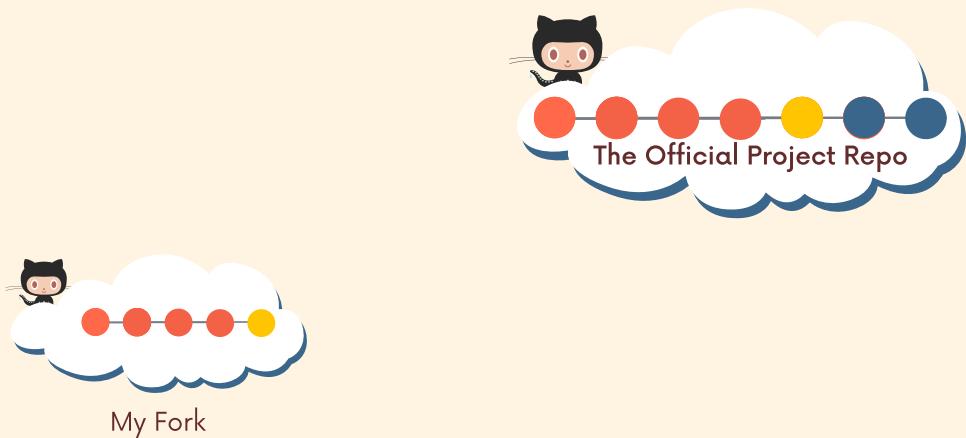


Github

All I need to do is pull from upstream (the original repo) to get the latest changes in my local repo.



Github

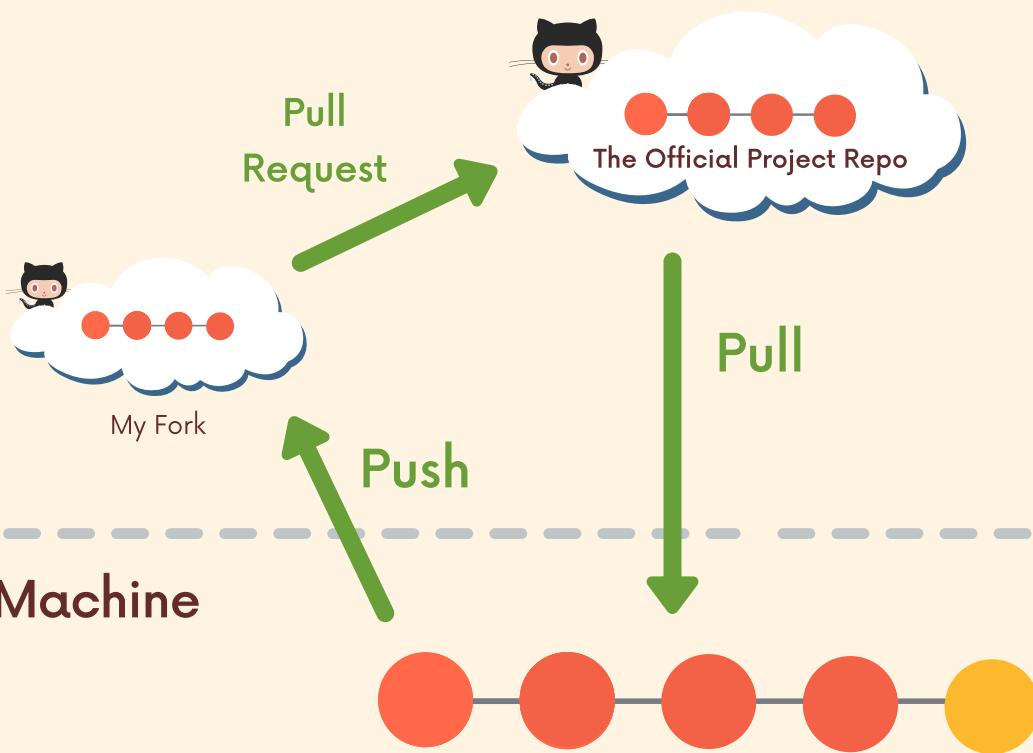


My Local Machine

Now I have the latest changes from the upstream repo! I can work on some new features locally without working about being out of date.



Github





To Summarize!

- 1.I fork the original project repo on Github
- 2.I clone my fork to my local machine
- 3.I add a remote pointing to the original project repo.
This remote is often named upstream.
- 4.I make changes and add/commit on a feature
branch on my local machine
- 5.I push up my new feature branch to my forked repo
(usually called origin)
- 6.I open a pull request to the original project repo
containing the new work on my forked repo
- 7.Hopefully the pull request is accepted and my
changes are merged in!



An Even Briefer Summary

- 1.FORK THE PROJECT
- 2.CLONE THE FORK
- 3.ADD UPSTREAM REMOTE
- 4.DO SOME WORK
- 5.PUSH TO ORIGIN
- 6.OPEN PR



14 Rebasing & Cleaning up history

Rebasing



≡

Rebasing

When I first learned Git, I was told to avoid rebasing at all costs.

"It can really #%@* things up"
"It's not for beginners!"





Rebasing

So I avoided the `git rebase` command for YEARS!



Rebasing

It's actually very useful, as long as you know when NOT to use it!





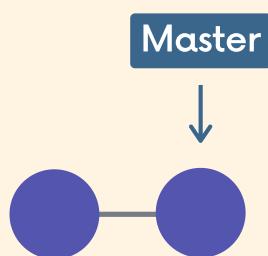
Rebasing

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

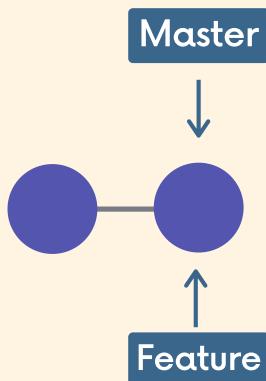
- as an alternative to merging
- as a cleanup tool



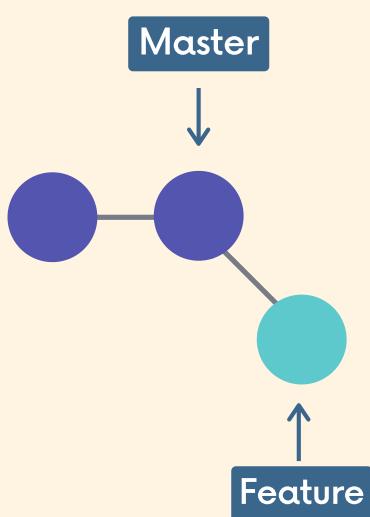
I'm working on a collaborative project



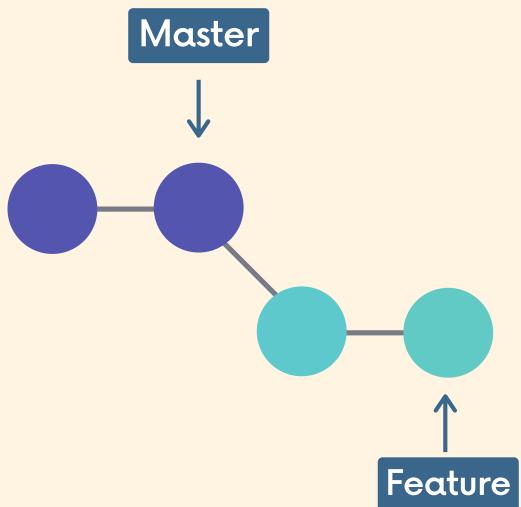
I make a feature branch!



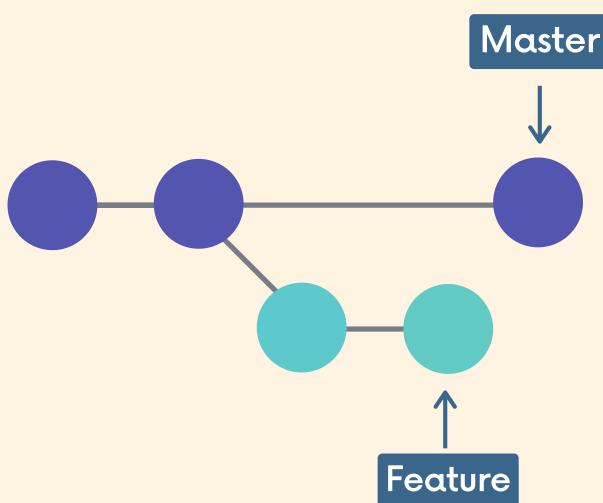
I do some work on the branch



I do some more work

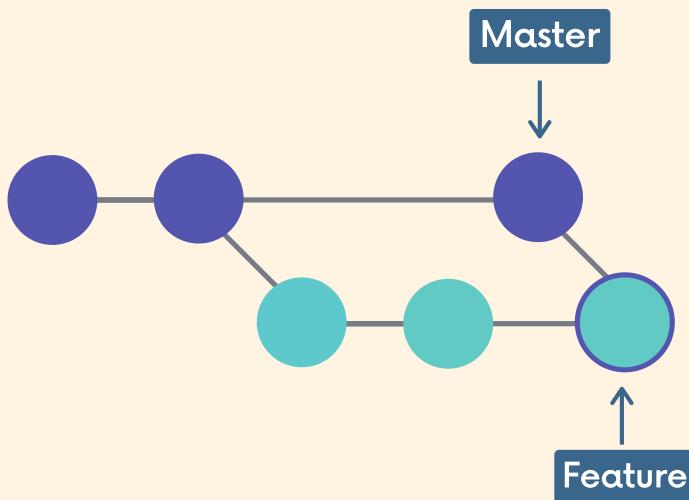


Master has new work on it!



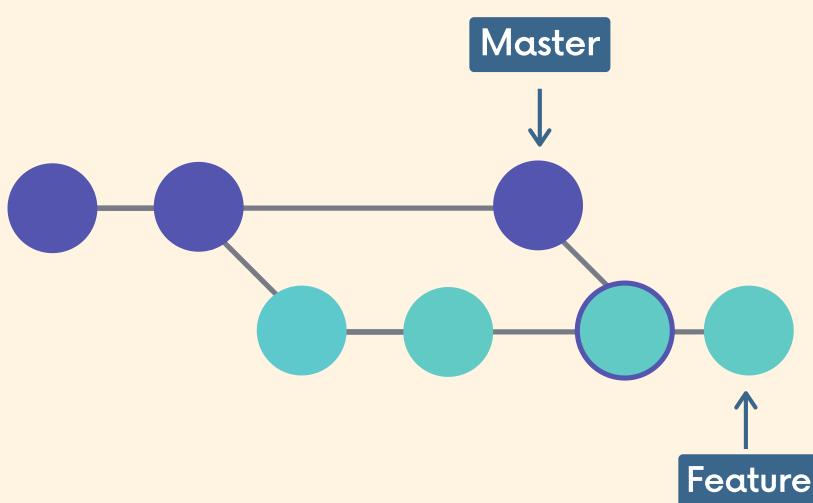
My feature branch doesn't have this work!

I merge master into feature

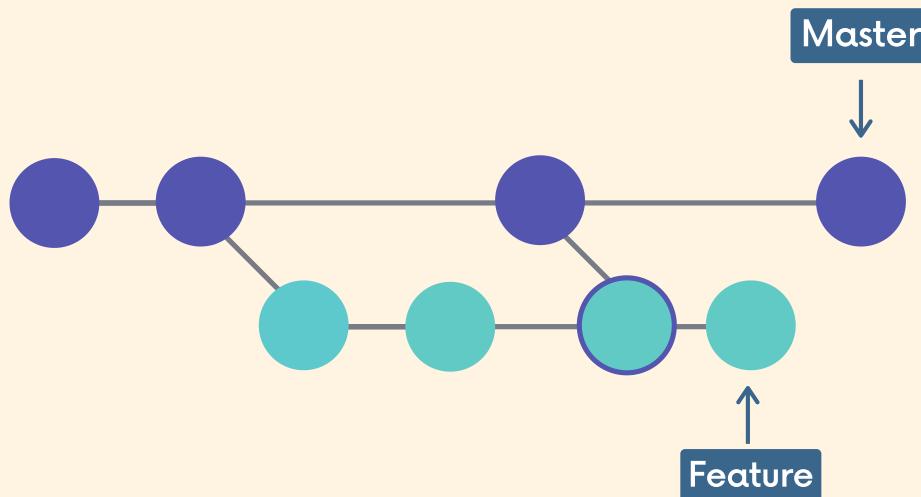


This results in a new merge commit

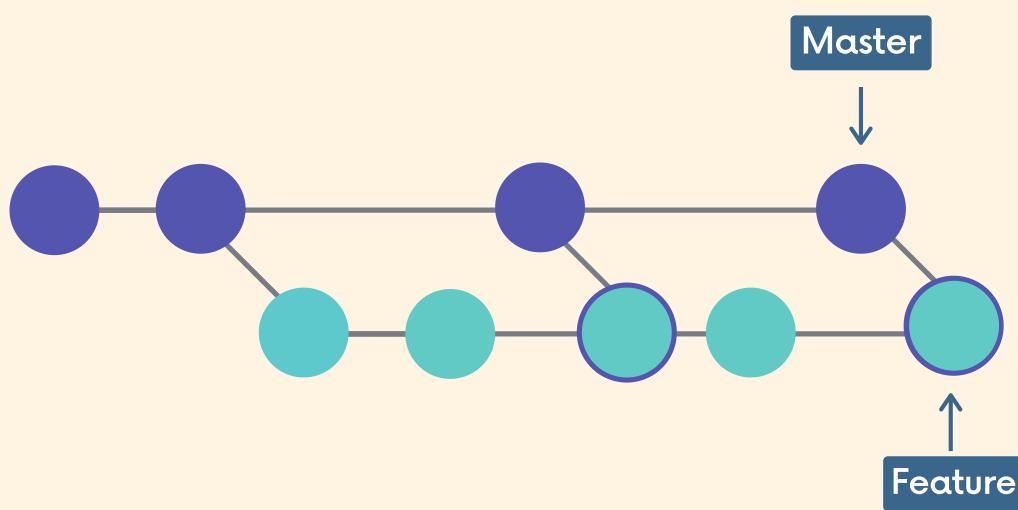
I keep working on my feature branch



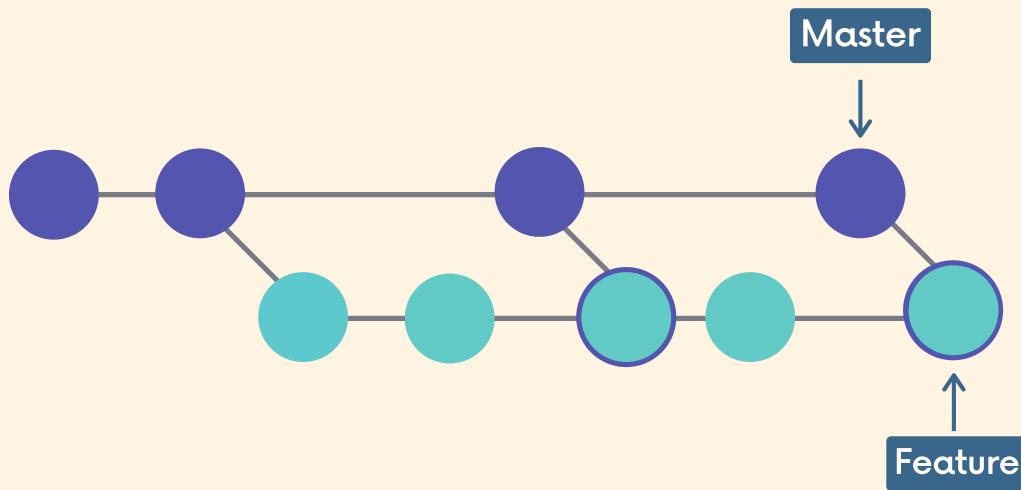
A coworker adds new work to master



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!

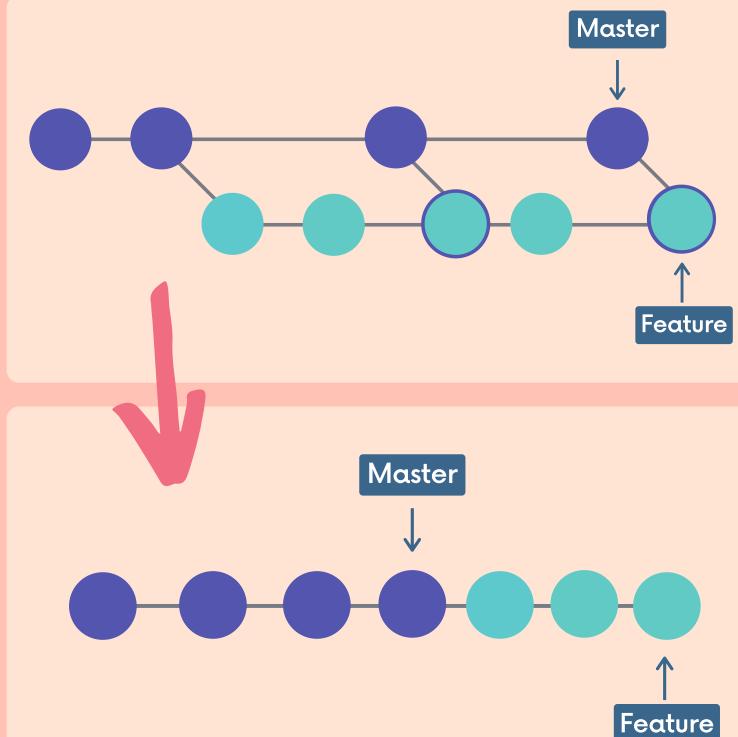
We can instead rebase the feature branch onto the master branch. This moves the entire feature branch so that it BEGINS at the tip of the master branch. All of the work is still there, but we have re-written history.

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

```

> git switch feature
> git rebase master

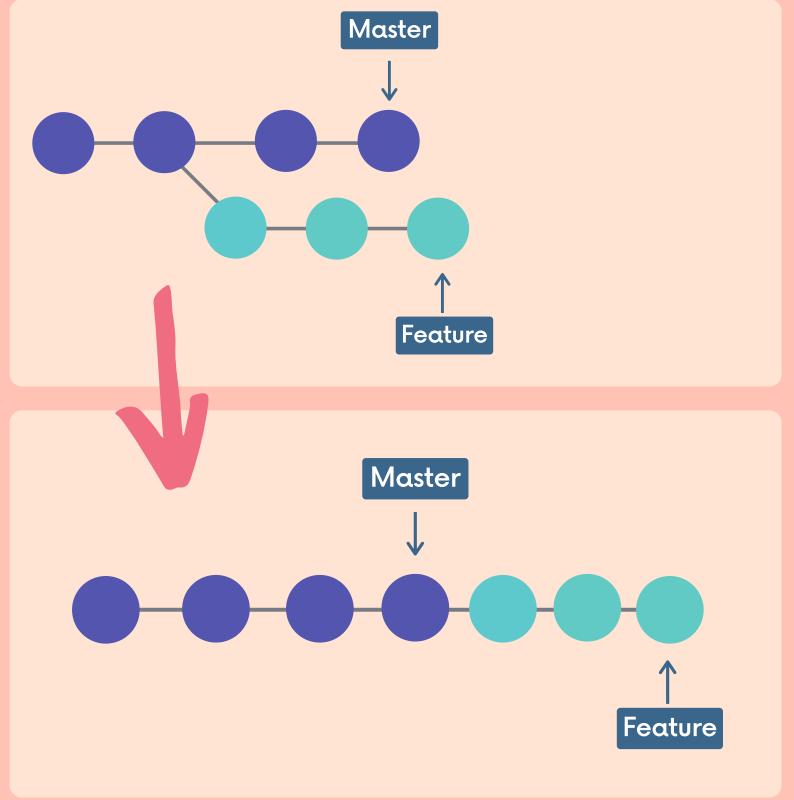
```



Rebasing!

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

```
❯ git switch feature  
❯ git rebase master
```



Why Rebase?

We get a 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 a pain to reconcile the alternate histories!





Rebasing

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

- as an alternative to merging
- as a cleanup tool



Rewriting History

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



Interactive Rebase

Running `git rebase` with the `-i` option will enter the interactive mode, which 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
```

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 previous commit and discard the commit message
- **drop** - remove commit

pick f7f3f6d Change my name a bit
pick 310154e Update README
pick a5f4a0d Add cat-file



drop f7f3f6d Change my name a bit
pick 310154e Update README
reword a5f4a0d Add cat-file

15 git Tags
marking
important
moments in
history

Git Tags

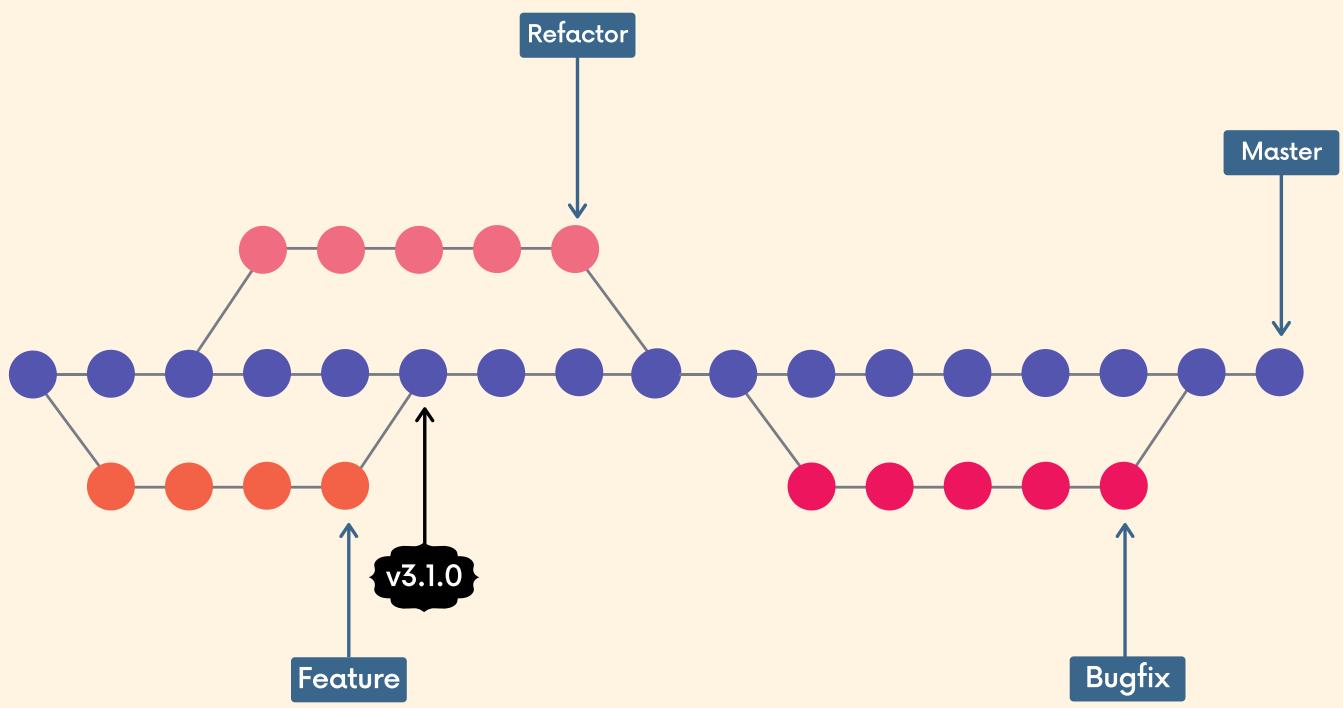
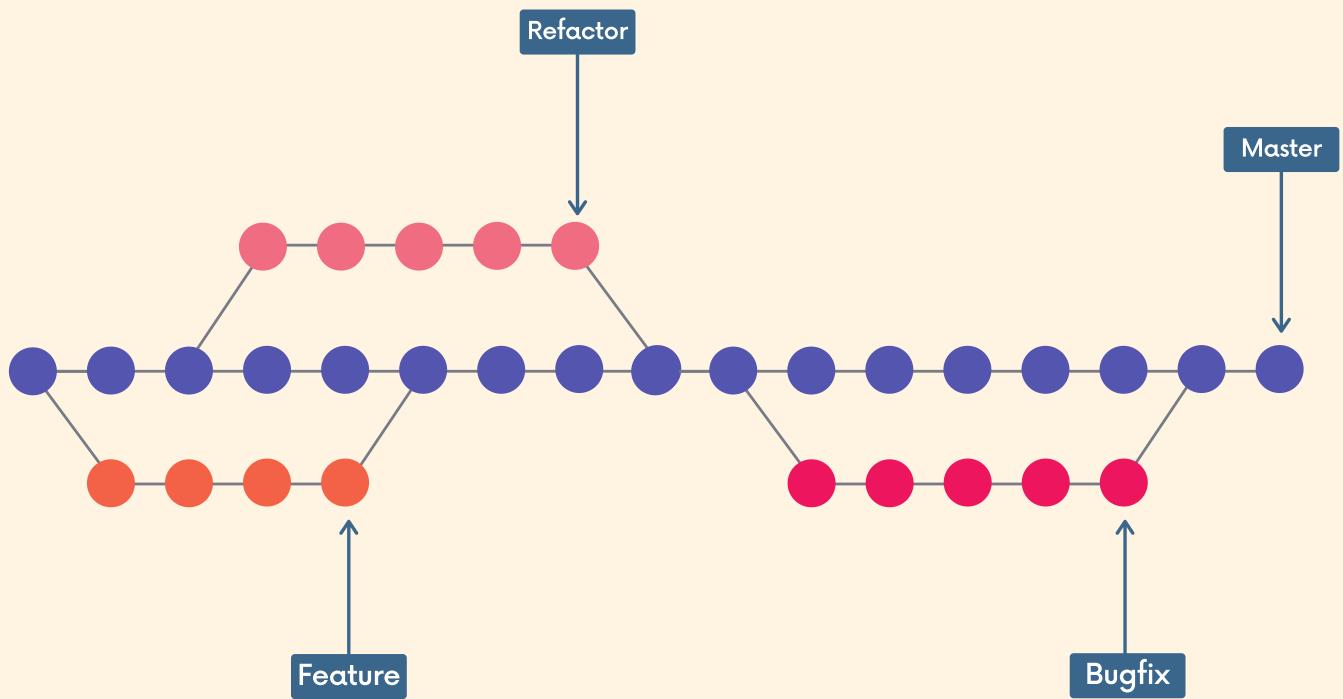


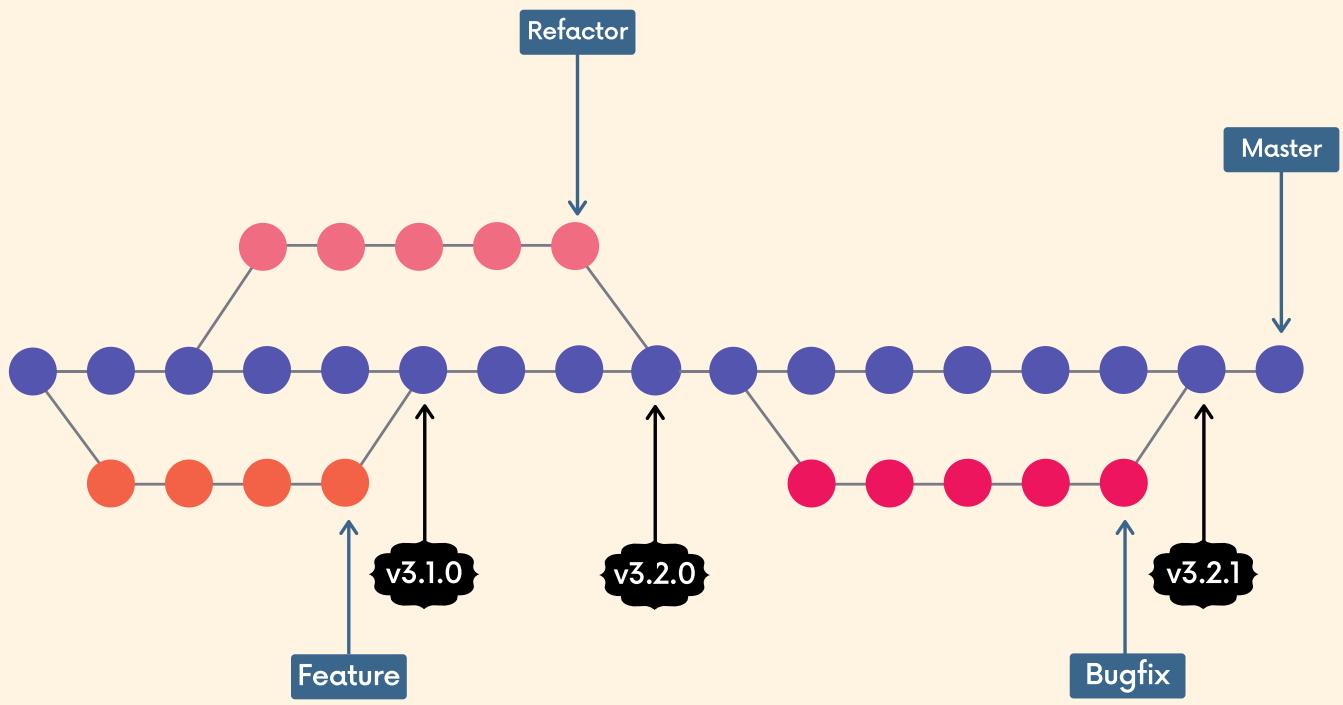
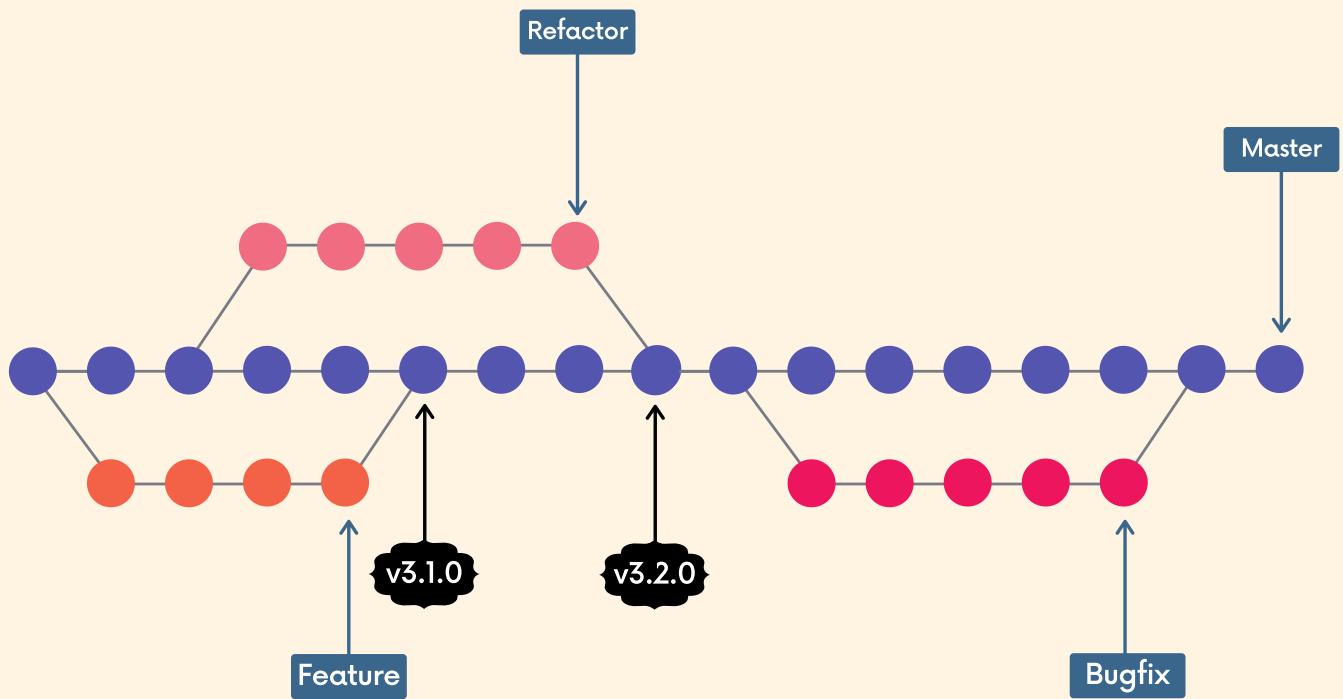
Git Tags

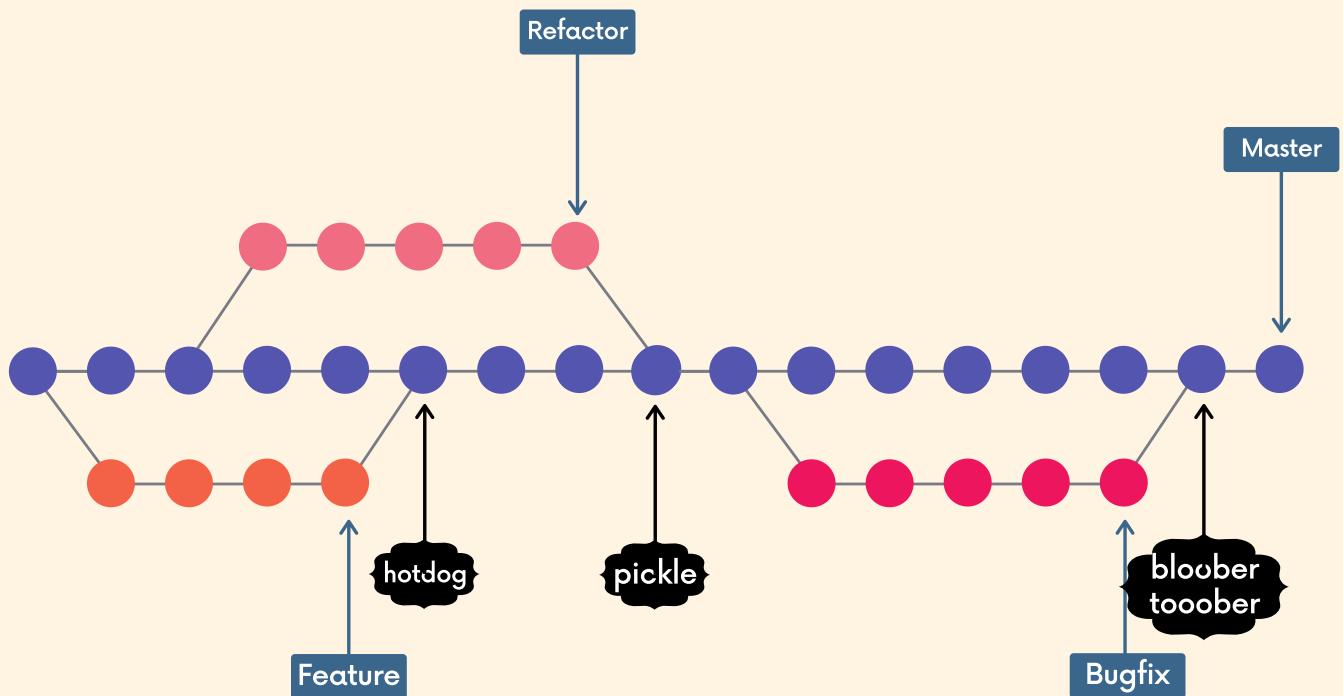
Tags are pointers that refer to particular points in Git history. We can mark a particular moment in time with a tag. Tags are most often used to mark version releases in projects (v4.1.0, v4.1.1, etc.)

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

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)



Semantic Versioning

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

Versions consist of three numbers separated by periods.

2.4.1

2.4.1

MAJOR RELEASE

MINOR RELEASE

PATCH RELEASE

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

Major Release

Major releases signify significant changes that is no longer backwards compatible. Features may be removed or changed substantially.

2.0.0

Viewing Tags

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

```
❯ git tag
```

Viewing Tags

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

```
❯ git tag -l "*beta*"
```

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>
```

The Two Types

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)



Creating Lightweight Tags

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



`>git tag <tagname>`

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 forgo the opening of the text editor



`>git tag -a <tagname>`

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 <tagname> <commit-hash>`

```
❯ git tag <tagname> <commit>
```

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 <tagname>
```

Deleting Tags

To delete a tag, use `git tag -d <tagname>`



```
›git tag -d <tagname>
```

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
```

16 git behind the scenes



Git Behind The Scenes



What is in .git??



There's more, but this is the juicy stuff

Config

The config file is for...configuration. We've seen how to configure global settings like our name and email across all Git repos, but we can also configure things on a per-repo basis.



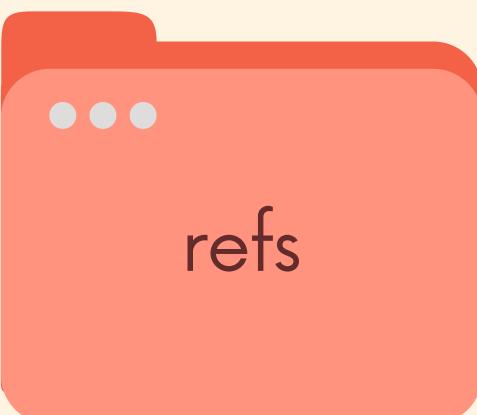
config

Refs Folder

Inside of refs, you'll find a heads directory. **refs/heads** contains one file per branch in a repository. Each file is named after a branch and contains the hash of the commit at the tip of the branch.

For example **refs/heads/master** contains the commit hash of the last commit on the master branch.

Refs also contains a **refs/tags** folder which contains one file for each tag in the repo.



HEAD

HEAD is just a text file that keeps track of where HEAD points.

If it contains refs/heads/master, this means that HEAD is pointing to the master branch.

In detached HEAD, the HEAD file contains a commit hash instead of a branch reference



HEAD

Index

The index file is a binary file that contains a list of the files the repository is tracking. It stores the file names as well as some metadata for each file.

Note that the index does NOT store the actual contents of files. It only contains references to files.



Index

Objects Folder

The objects directory contains all the repo files. This is where Git stores the backups of files, the commits in a repo, and more.

The files are all compressed and encrypted, so they won't look like much!



4 Types of Git Objects



commit

tree

blob

annotated
tag



TIME OUT!

We need to talk about hashing

HASHING FUNCTIONS

Hashing functions are functions that map input data of some arbitrary size to fixed-size output values.



CRYPTOGRAPHIC HASH FUNCTIONS

1. One-way function which is infeasible to invert
2. Small change in input yields large change in the output
3. Deterministic - same input yields same output
4. Unlikely to find 2 outputs with same value

SHA-1

Git uses a hashing function called SHA-1 (though this is set to change eventually).

- SHA-1 always generates 40-digit hexadecimal numbers
- The commit hashes we've seen a million times are the output of SHA-1

Git Database

Git is a **key-value data store**. We can insert any kind of content into a Git repository, and Git will hand us back a unique key we can later use to retrieve that content.

These keys that we get back are SHA-1 checksums.



Keys

Values

| | |
|--|--|
| 1f7a7a472abf3d d9643fd615f6da 379c4acb3e3a | 83baae61804e6 5cc73a7201a725 2750c76066a30 |
| V1 of app.js | V2 of app.js |

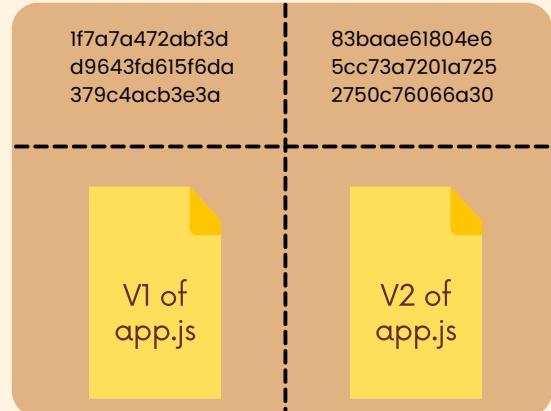


Please give me the content for this key:

83BAAE61804E65CC73A720
1A7252750C76066A30

Keys

Values

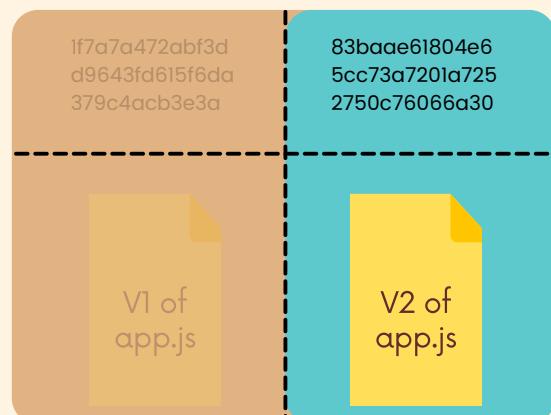


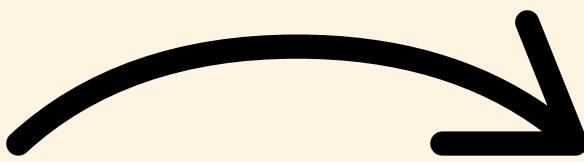
Please give me the content for this key:

83BAAE61804E65CC73A720
1A7252750C76066A30

Keys

Values



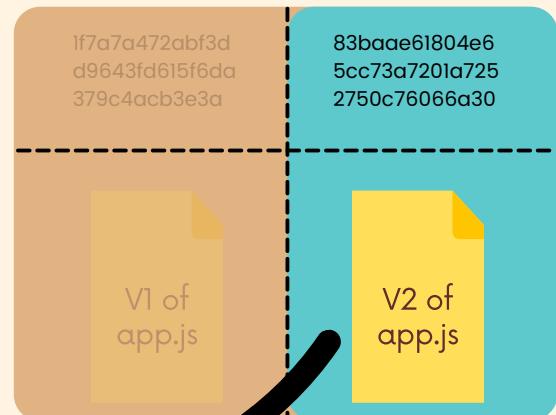


Please give me the content for this key:

83BAAE61804E65CC73A720
1A7252750C76066A30

Keys

Values



HASHING FUNCTIONS

Git uses SHA-1 to hash our files, directories, and commits.



SHA-1

1F7A7A472ABF3DD9643FD61
5F6DA379C4ACB3E3A

83BAAE61804E65CC73A720
1A7252750C76066A30

Let's Try Hash^{ing}

The **git hash-object** command takes some data, stores it in our .git/objects directory and gives us back the unique SHA-1 hash that refers to that data object.

In the simplest form (shown on the right), Git simply takes some content and returns the unique key that WOULD be used to store our object. But it does not actually store anything



```
git hash-object <file>
```

Let's Try Hash^{ing}



```
echo 'hello' | git hash-object --stdin
```

The **--stdin** option tells **git hash-object** to use the content from stdin rather than a file. In our example, it will hash the word 'hello'

The **echo** command simply repeats whatever we tell it to repeat to the terminal. We pipe the output of **echo** to **git hash-object**.

Let's Try Hashing



```
❯ echo 'hello' | git hash-object --stdin -w
```

Rather than simply outputting the key that git would store our object under, we can use the `-w` option to tell git to actually write the object to the database.

After running this command, check out the contents of `.git/objects`

Let's Try Hashing



```
❯ git cat-file -p <object-hash>
```

Now that we have data stored in our Git object database, we can try retrieving it using the `git cat-file` command.

The `-p` option tells Git to pretty print the contents of the object based on its type.

1.Tell Git to store "hello". Note the hash that we get back.

```
❯ echo 'hello' | git hash-object --stdin -w  
ce013625030ba8dba906f756967f9e9ca394464a
```

2. Pass the hash from above to git cat-file. Git retrieves the corresponding data "hello" that it had stored under that key

```
❯ git cat-file -p  
ce013625030ba8dba906f756967f9e9ca394464a  
hello
```

'hello' → 1f7a7a472abf3dd9643fd615
f6da379c4acb3e3a

1f7a7a472abf3d
d9643fd615f6da
379c4acb3e3a

'hello' → 1f7a7a472abf3dd9643fd615
f6da379c4acb3e3a

'goodbye' → dd7e1c6f0fefef118f0b63d9f1
0908c460aa317a6

1f7a7a472abf3d
d9643fd615f6da
379c4acb3e3a

dd7e1c6f0fefef118
f0b63d9f10908c
460aa317a6

'hello' → 1f7a7a472abf3dd9643fd615
f6da379c4acb3e3a

'goodbye' → dd7e1c6f0fefef118f0b63d9f1
0908c460aa317a6

'chicken' → 4768c088926158b50144c71
0989e13fc697550fc

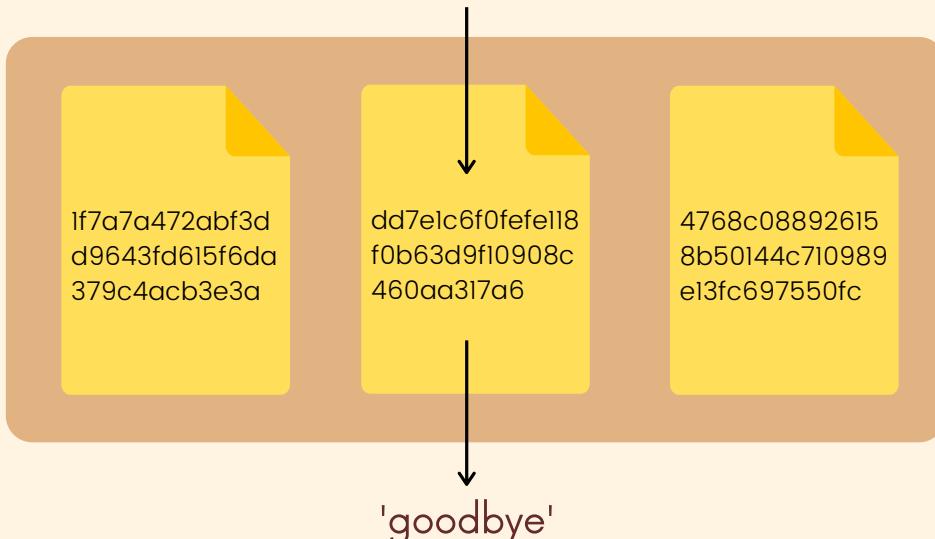
1f7a7a472abf3d
d9643fd615f6da
379c4acb3e3a

dd7e1c6f0fefef118
f0b63d9f10908c
460aa317a6

4768c08892615
8b50144c710989
e13fc697550fc

Hello Git, I would like to know what you have stored under the key:

```
dd7e1c6f0fefef118f0b63  
d9f10908c460aa317a6
```



Blobs

Git blobs (binary large object) are the object type Git uses to store the **contents of files** in a given repository. Blobs don't even include the filenames of each file or any other data. They just store the contents of a file!



Trees

Trees are Git objects used to store the contents of a directory. Each tree contains pointers that can refer to blobs and to other trees.

Each entry in a tree contains the SHA-1 hash of a blob or tree, as well as the mode, type, and filename

c38719da...

tree

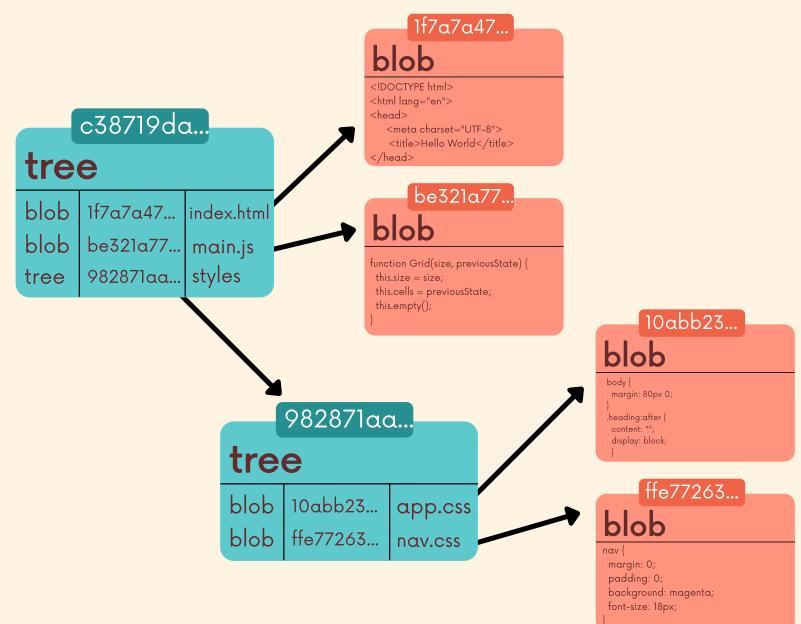
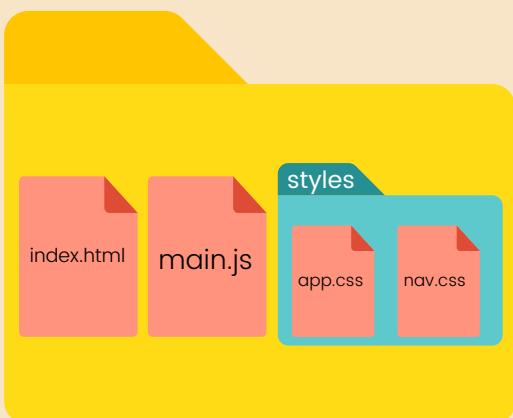
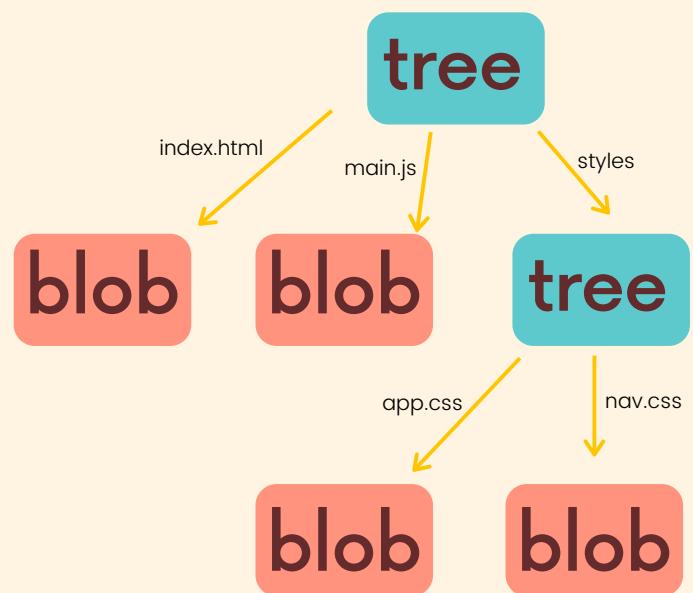
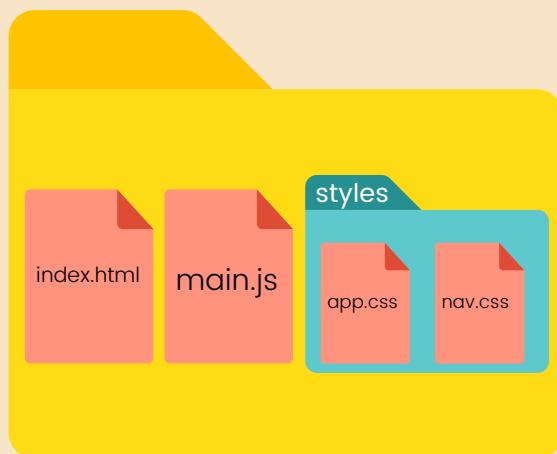
| | | |
|------|--------------|--------|
| blob | 1f7a7a47... | app.js |
| tree | 982871aa... | images |
| blob | be321a77... | README |
| tree | 80ff1ae33... | styles |

Viewing Trees



```
› git cat-file -p master^{tree}
```

Remember that `git cat-file` prints out Git objects. In this example, the `master^{tree}` syntax specifies the tree object that is pointed to by the tip of our master branch.



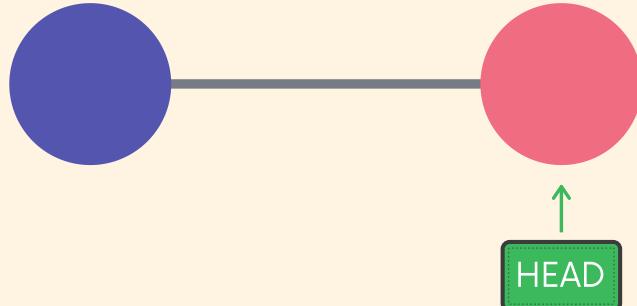
Commits

Commit objects combine a tree object along with information about the context that led to the current tree. Commits store a reference to parent commit(s), the author, the committer, and of course the commit message!

| | |
|---------------------------|-------------|
| fa49b07... | |
| commit | |
| tree | c38719da... |
| parent | ae234ffa... |
| author | Sirius |
| committer | Sirius |
| this is my commit message | |

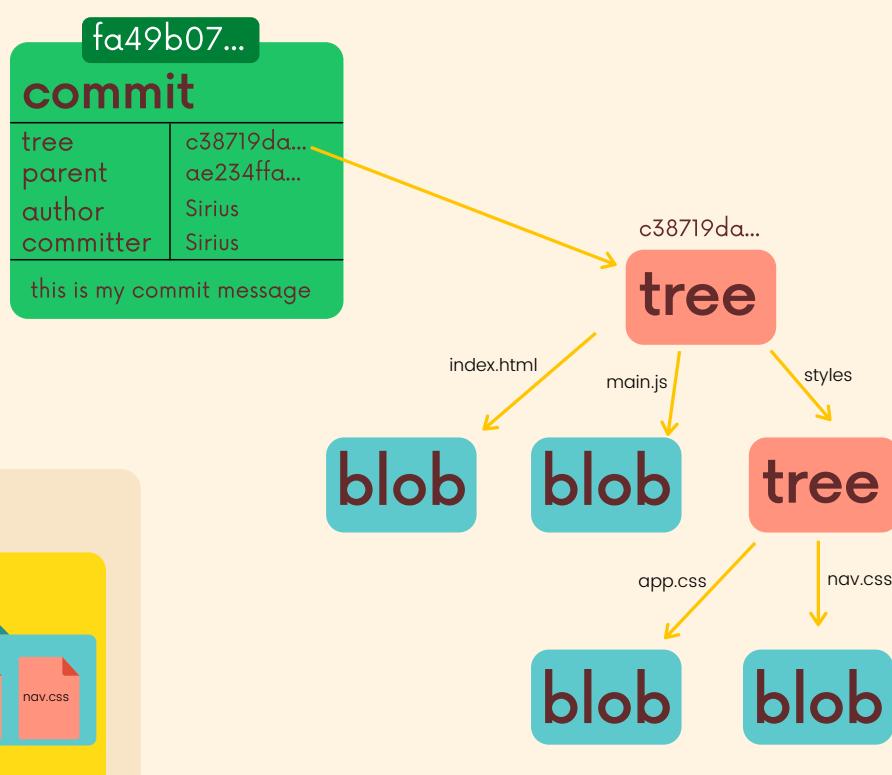
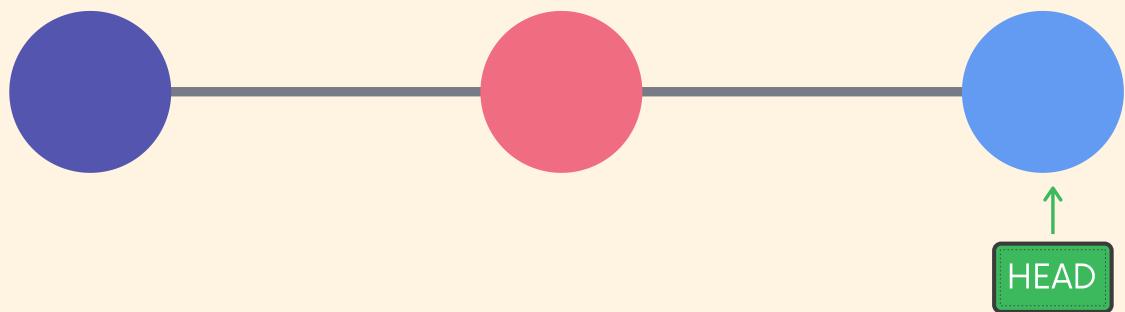
| | |
|--------------------|----------------|
| 987fac676ce7dd765e | |
| tree | 647ffea2... |
| parent | none |
| author | Wolfgang |
| committer | Wolfgang |
| message | initial commit |

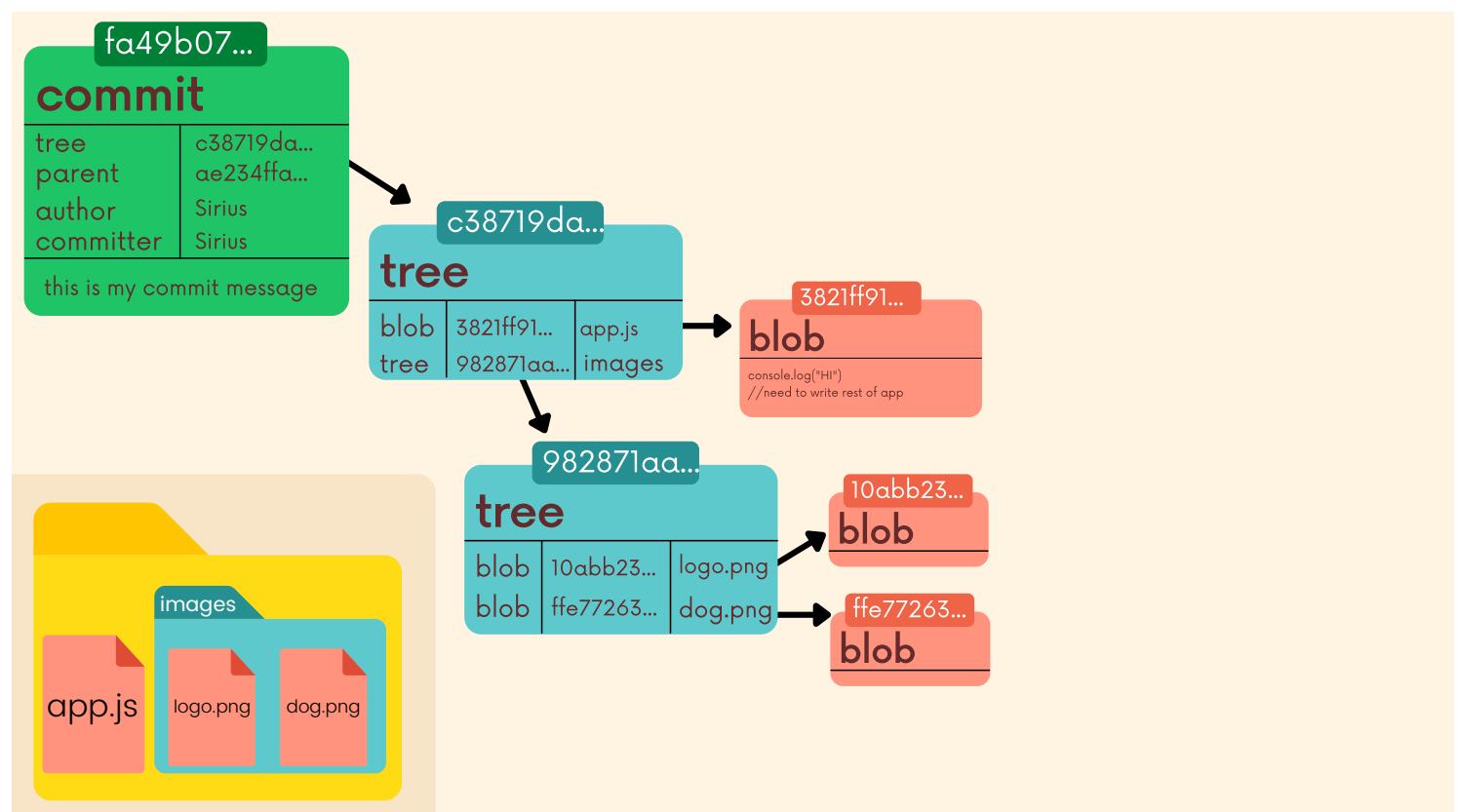
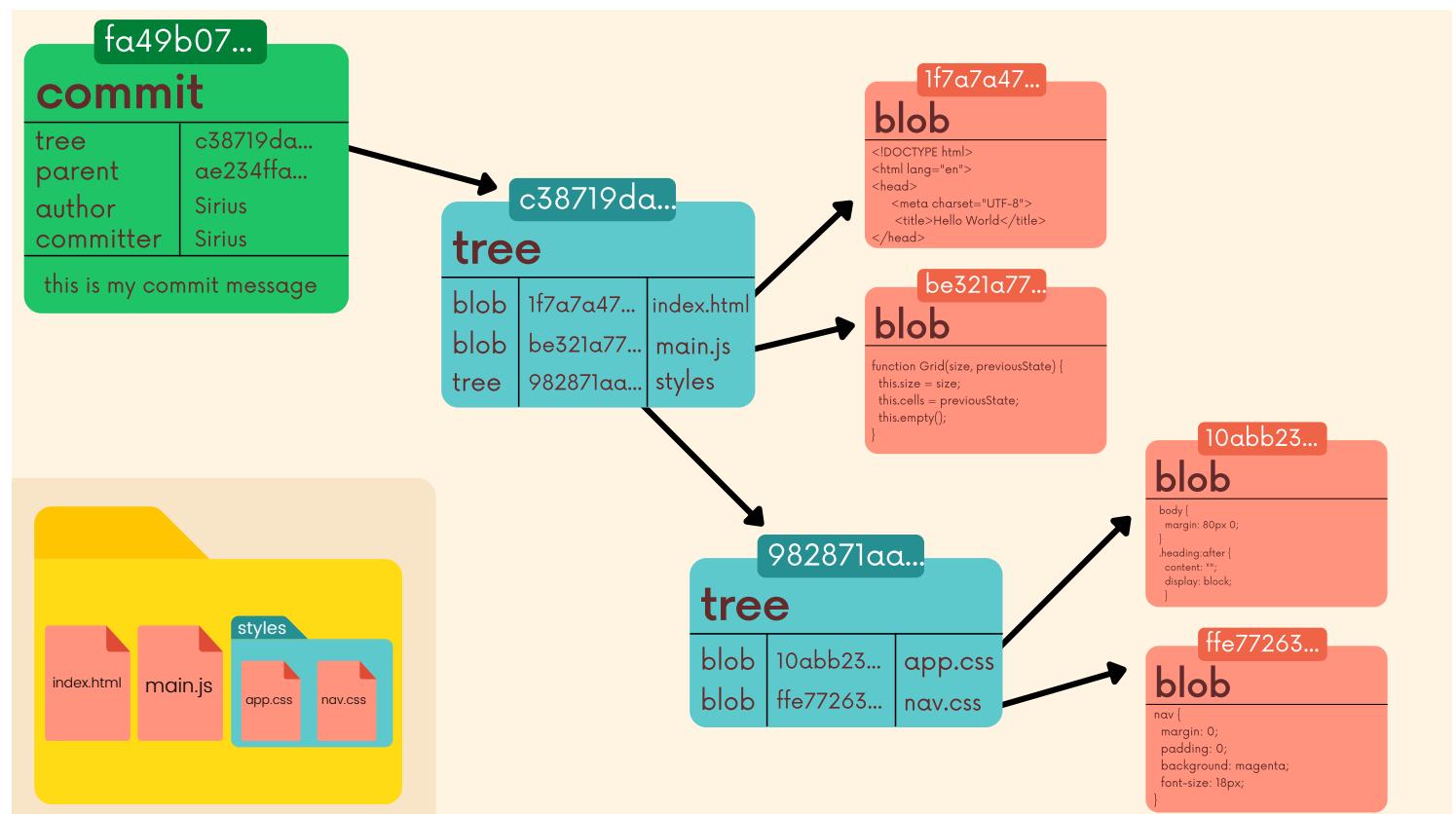
| | |
|---------------------|---------------|
| 236ff654e1adf35d897 | |
| tree | 7df7df123f... |
| parent | 987fac676... |
| author | Wolfgang |
| committer | Wolfgang |
| message | fix typo |

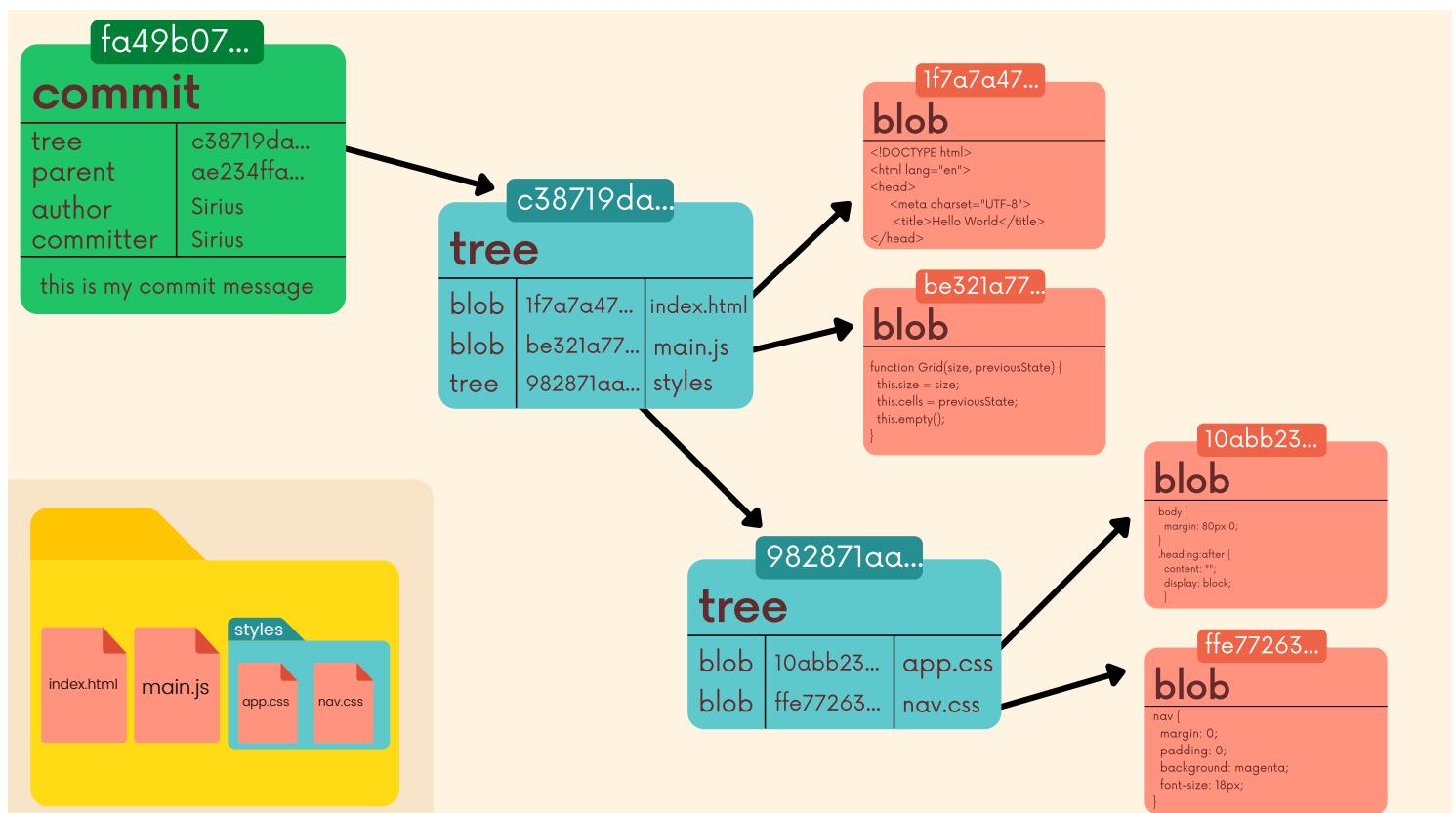
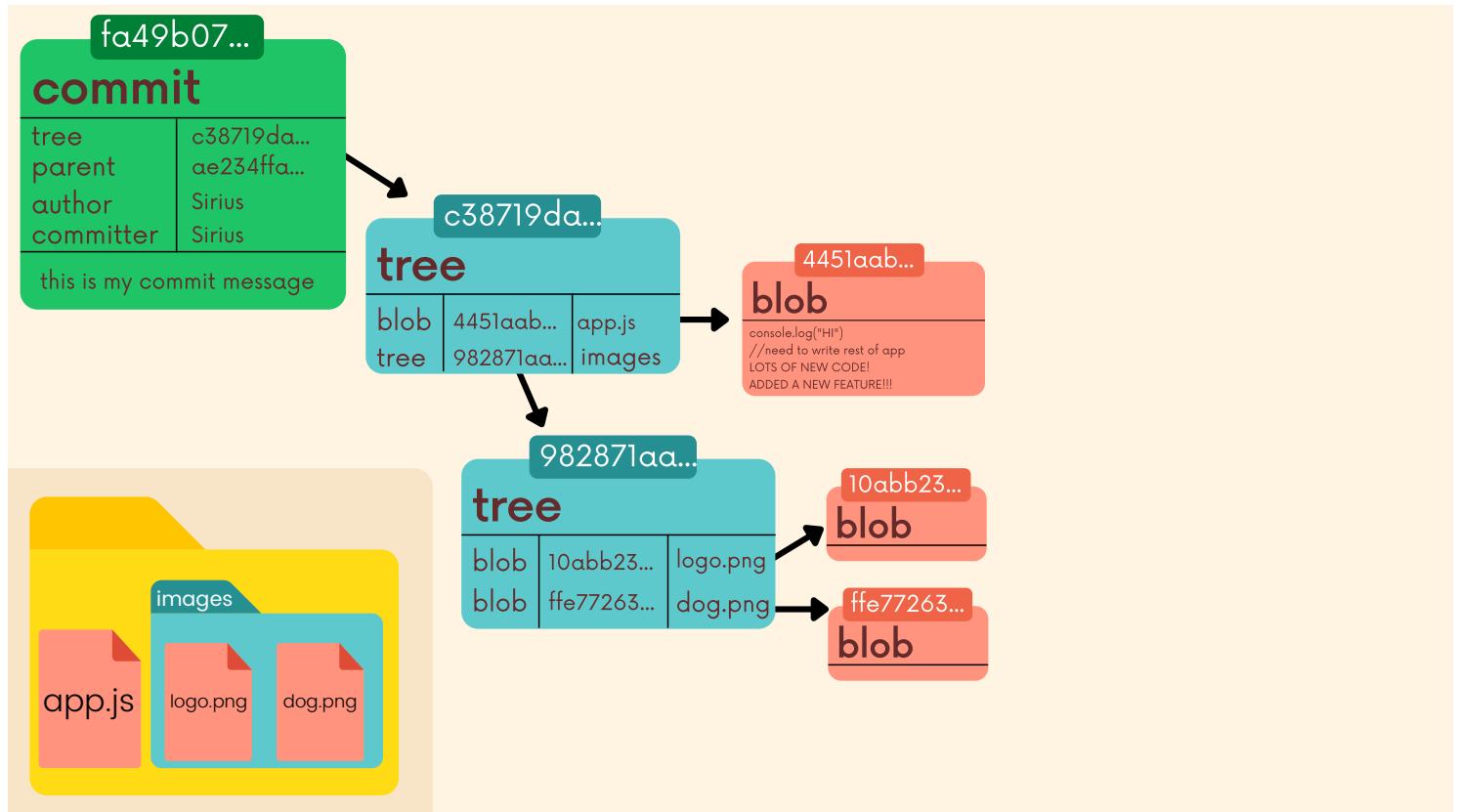


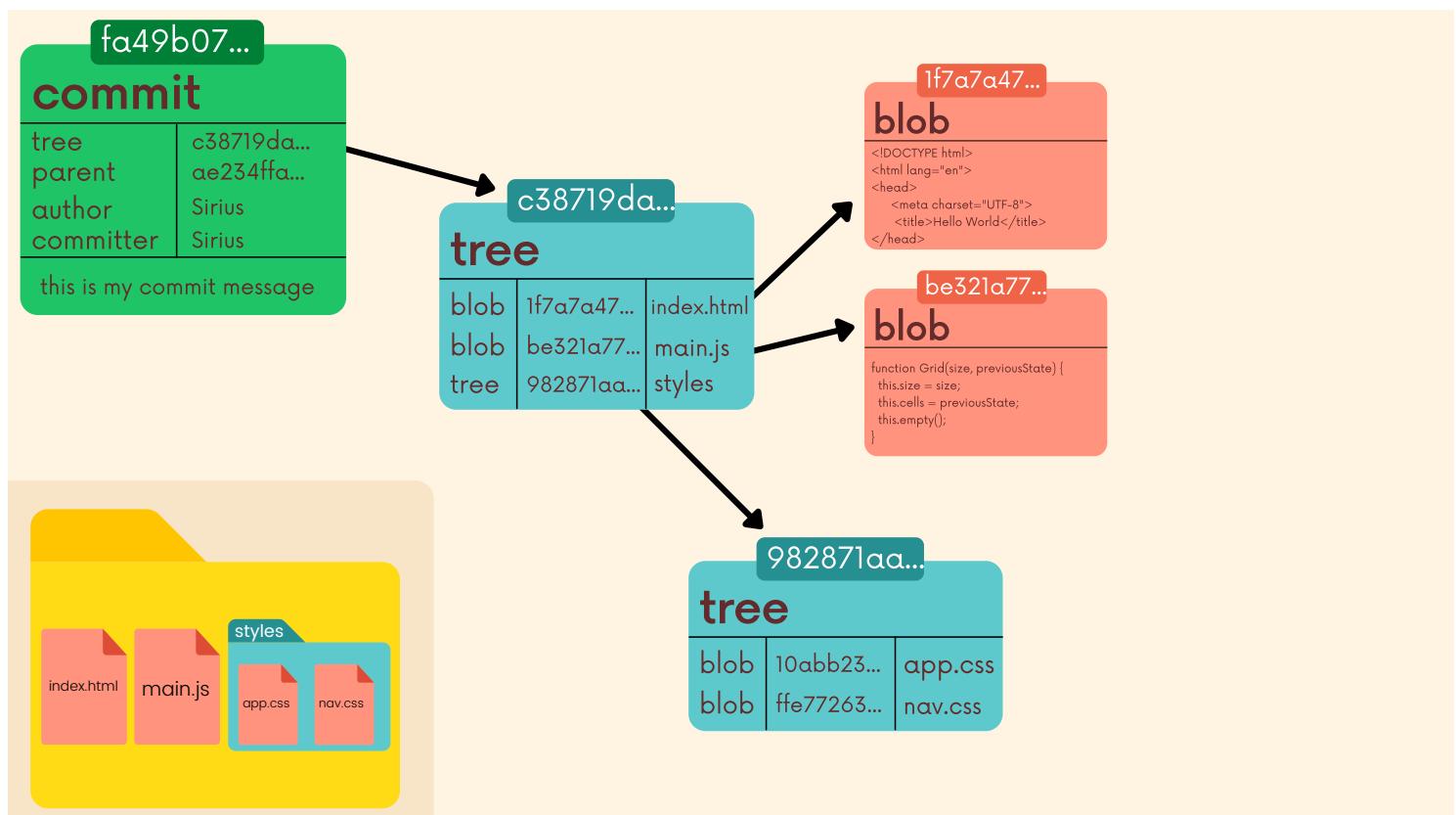
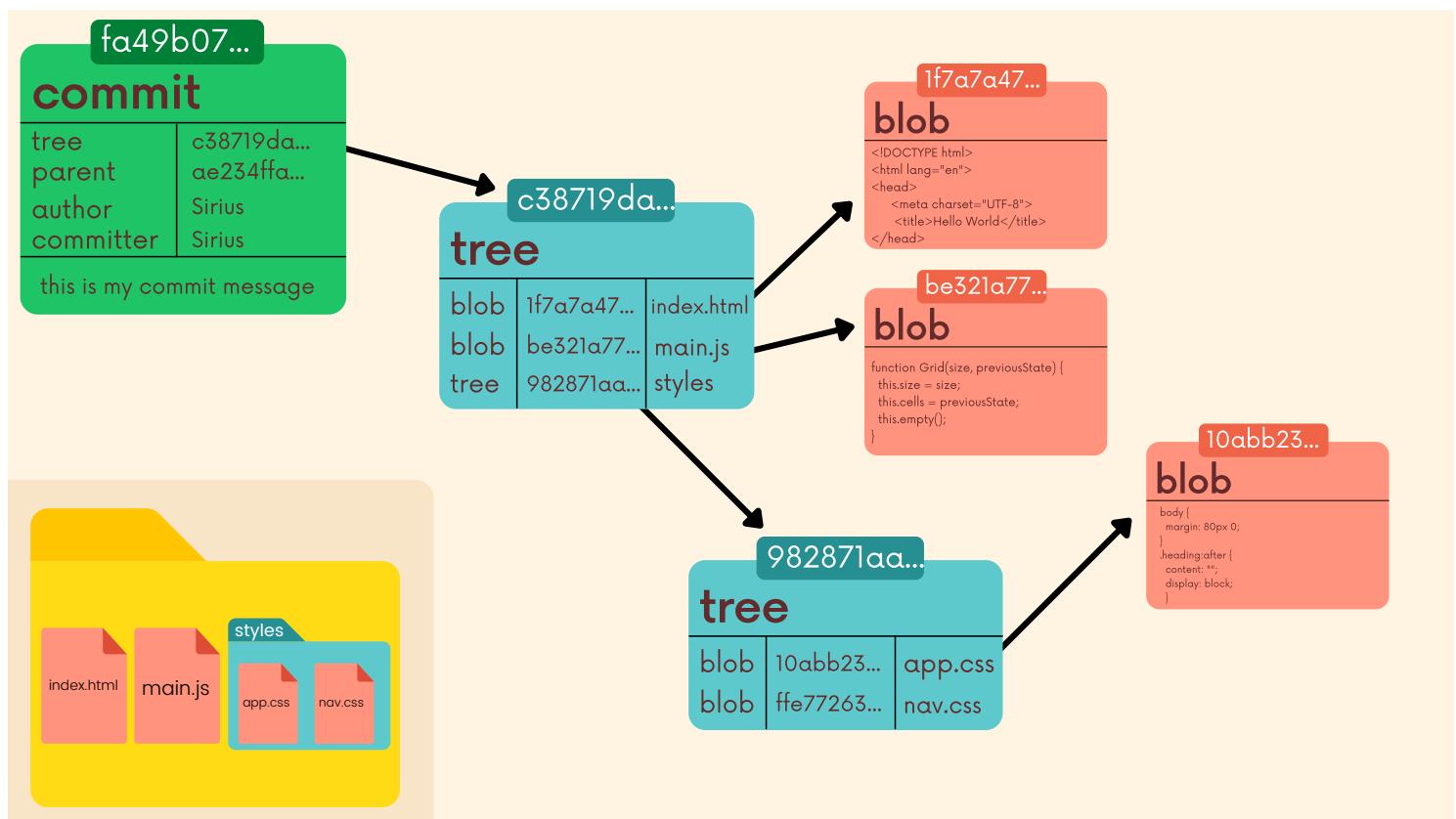
When we run `git commit`, Git creates a new commit object whose parent is the **current HEAD commit** and whose tree is the current content of the index.

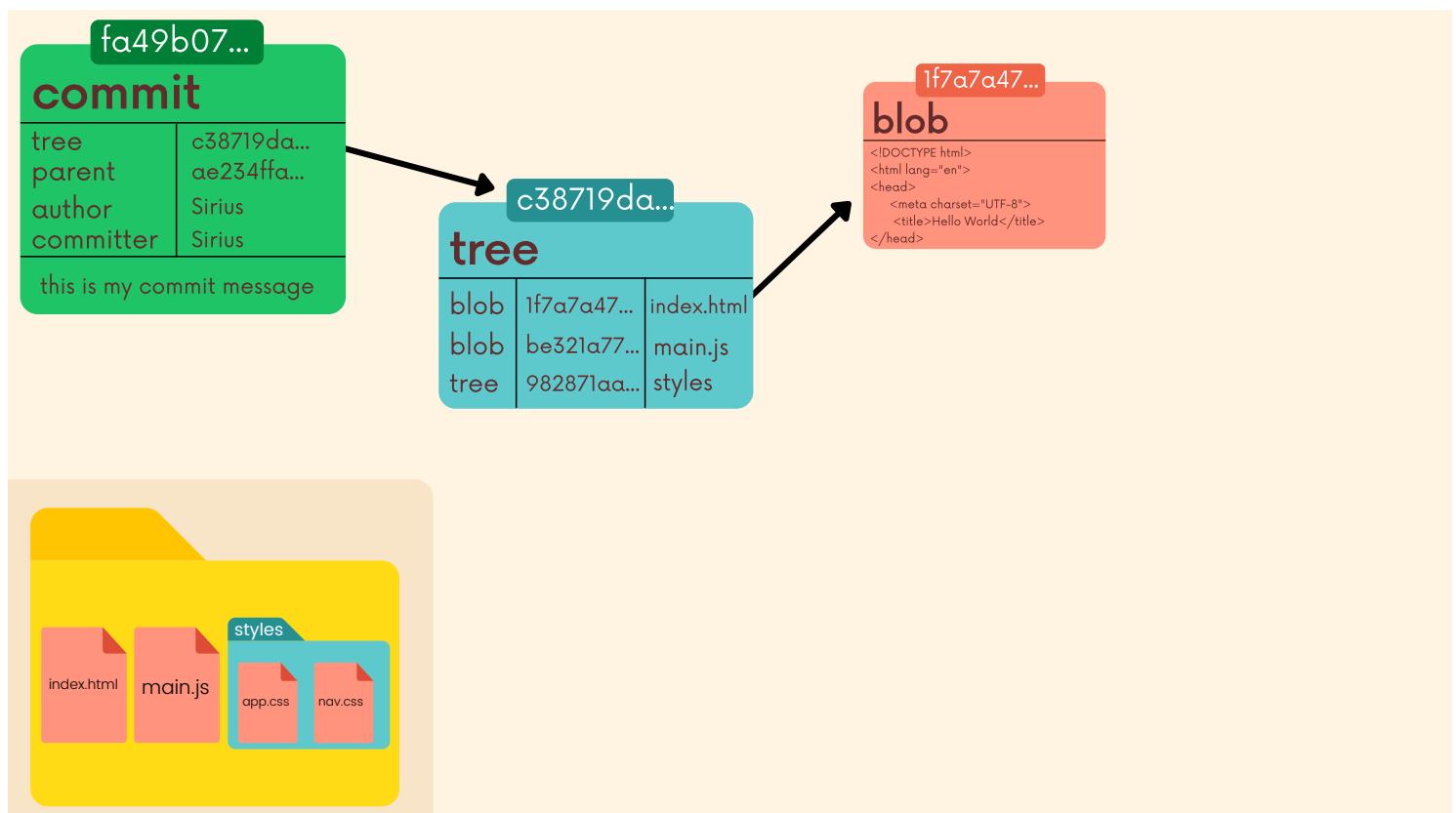
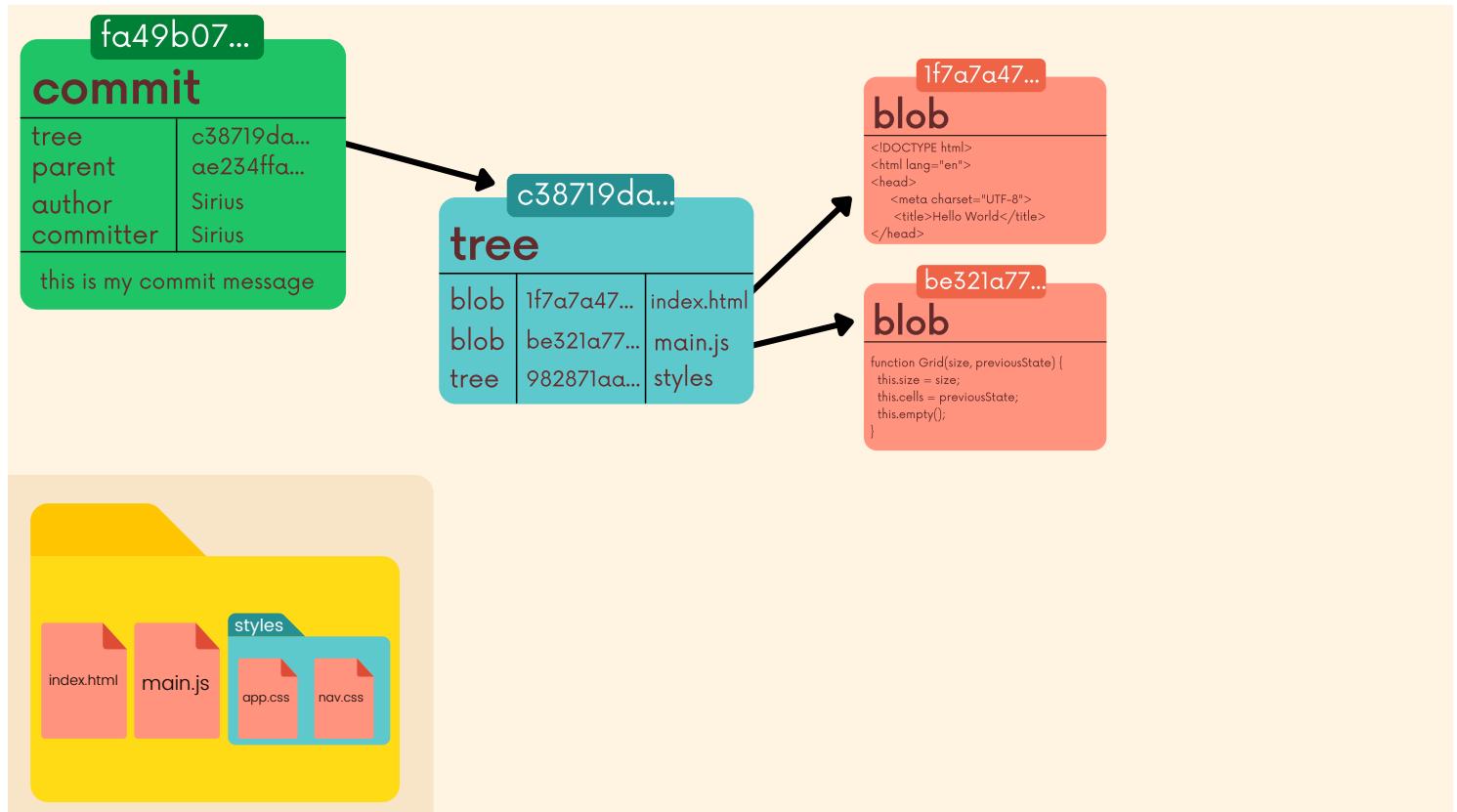
| | | |
|------------------------|---------------------|-----------------------|
| 987fac676ce7dd765e | | |
| tree 647ffea2... | tree 7df7df123f... | tree a912dda2a... |
| parent none | parent 987fac676... | parent 236ff654e1a... |
| author Wolfgang | author Wolfgang | author Wolfgang |
| committer Wolfgang | committer Wolfgang | committer Wolfgang |
| message initial commit | message fix typo | message add banner |











fa49b07...

commit

| | |
|-----------|-------------|
| tree | c38719da... |
| parent | ae234ffa... |
| author | Sirius |
| committer | Sirius |

this is my commit message

c38719da...

tree

| | | |
|------|-------------|------------|
| blob | 1f7a7a47... | index.html |
| blob | be321a77... | main.js |
| tree | 982871aa... | styles |



fa49b07...

commit

| | |
|-----------|-------------|
| tree | c38719da... |
| parent | ae234ffa... |
| author | Sirius |
| committer | Sirius |

this is my commit message



17 Reflogs retrieving lost work

Reflogs



≡

reflogs

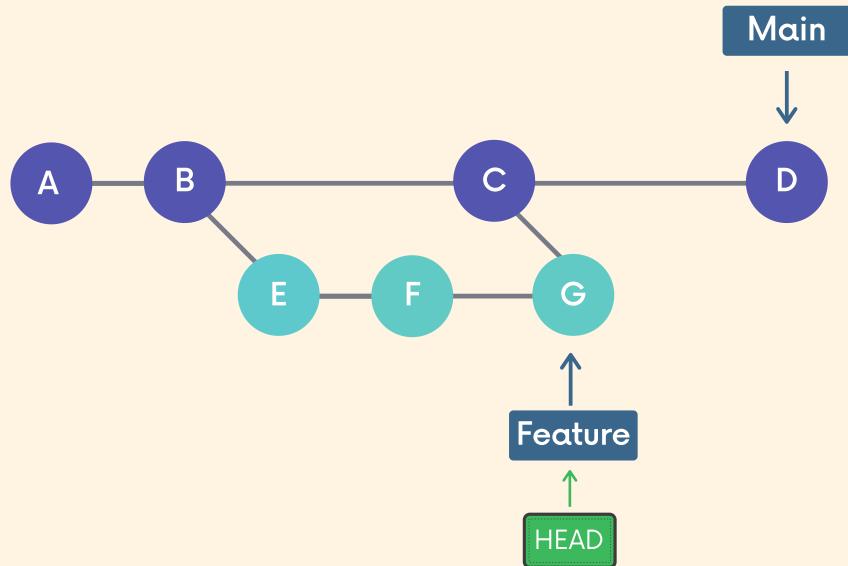
Git keeps a record of when the tips of branches and other references were updated in the repo.

We can view and update these reference logs using the `git reflog` command



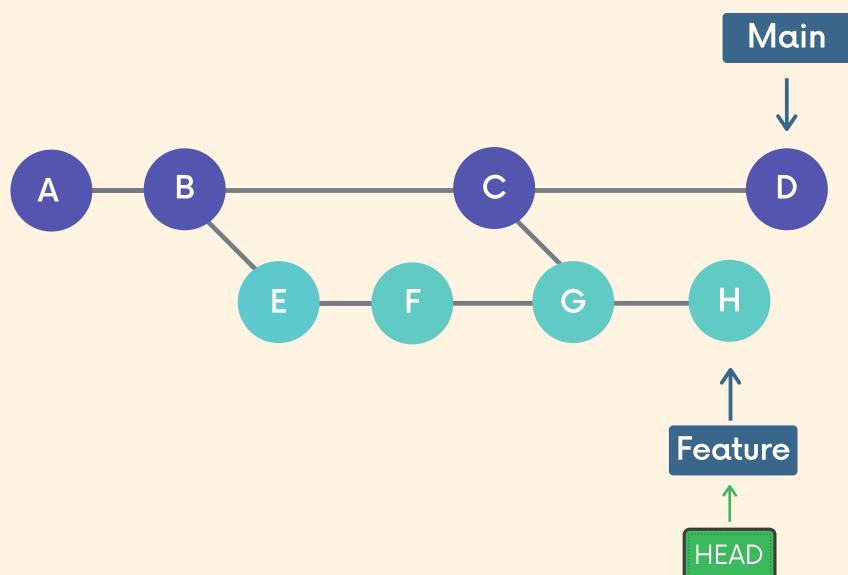
HEAD Reflog

- Commit G - New commit



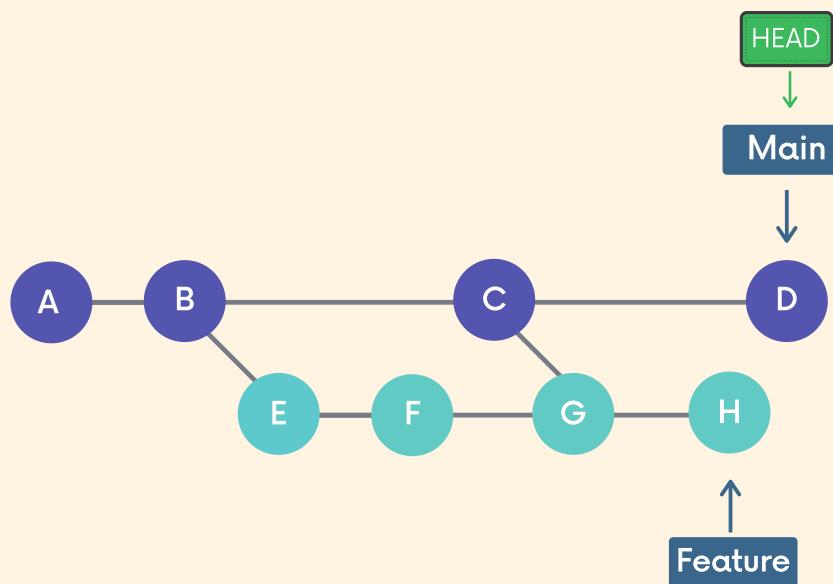
HEAD Reflog

- Commit G - New commit
- Commit H - New commit



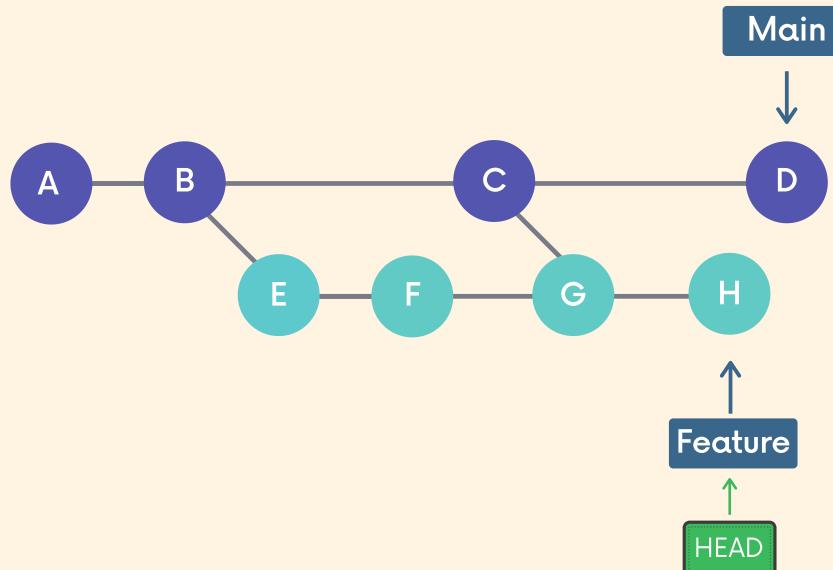
HEAD Reflog

- Commit G - New commit
- Commit H - New commit
- Commit D - Switched to main from feature



HEAD Reflog

- Commit G - New commit
- Commit H - New commit
- Commit D - Switched to main from feature
- Commit H - Switched to feature from main





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, though 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 defaults 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
```

HEAD Reflog

HEAD@{0}: commit: fix some bug

HEAD@{1}: checkout: moving from main to bugfix

HEAD@{2}: commit: another commit yay

HEAD@{3}: commit: add some feature

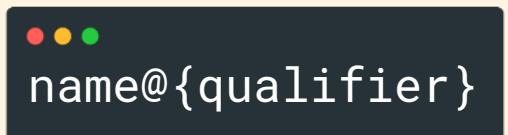
...

HEAD@{21}: clone: from

<https://github.com/facebook/react.git>

Reflog References

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



Timed 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.



18 Aliases

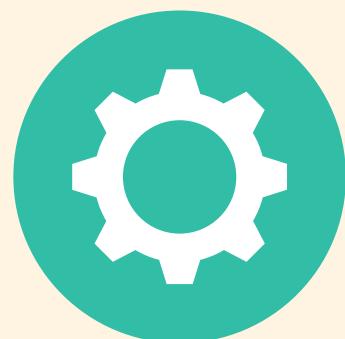
Git Aliases



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.



Adding Aliases

We can easily set up Git aliases to make our Git experience a bit simpler 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.

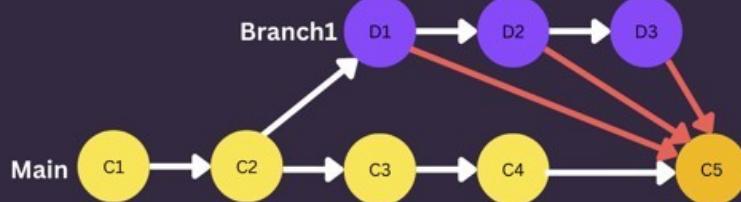
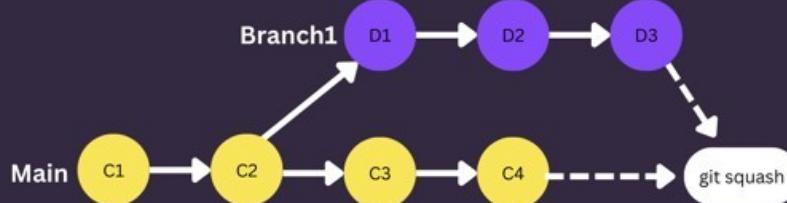
[alias]

s = status

l = log

19 Squash

Git Squash



The final command is Git Squash. As its name suggests it **squeezes** the commits into a single commit and then merged into target branch. It's kind of has both the features of Merge and Rebase. It creates a merge commit but it also keeps the projects history linear and clean. But the catch of Git Squash is that we lose the details of individual commits. So a lot of context about individual commits are lost in order to have a cleaner project history.

Positives

- Produces linear and clean commit history
- Creates a merge commit

Negatives

- Less detailed commits when squash and merged
- Tracking individual commits might be hard and can cause issues in debugging

Conclusion

In the end all three commands do the same thing but in their own different style. Even though you might not be using all of these commands right now it might be useful to learn how they work for future opportunities or for your personal project management.

So long story short

- Merge:** Creates a merge commit, gives all the info about the branch
- Rebase:** Moves the head of the current branch to the last node of the target branch and produces a more linear git history
- Squash:** Making all the commits into a one single commit and creates a clean linear history but does not provide as much information about commits.

Commands

To merge entire feature branch to main branch as a single commit

```
$ git checkout main  
$ git merge --squash feature
```

To squash N number of commits in the same branches

Copy the hash of last good commit using

```
$ git log  
$ git rebase -i <copied hash>
```

This will open an editor. Leave the first hash as **pick** and edit the reset hashes to **squash** and save.

Resolve conflicts if any.

Now, a **Commit message editor** will be opened. Give a commit message and save.

Now, git will have squashed all commits after the last good commit to a single commit.

20 Cherry

Pick

Cherry pick means, choosing one or more commits from one branch and applying it to another.



Single Cherry Pick

```
$ git log <branch to cherry pick from>
```

Copy the hash of the desired commit.

```
$ git checkout <branch to which the cherry pick has to be applied>
```

```
$ git cherry-pick <copied hash>
```

Resolve conflicts if any.

Multiple Cherry Pick (let's assume 3)

```
$ git log <branch to cherry pick from>
```

Copy the hashes of the desired commits.

```
$ git checkout <branch to which the cherry pick has to be applied>
```

```
$ git cherry-pick <copied hash 1> <copied hash 2> <copied hash 3>
```

This will create commits in the same order of the hashes specified in the branch to which the cherry pick has to be applied.

Resolve conflicts if any.

21 Graph

```
$ git log --online --graph --all
```