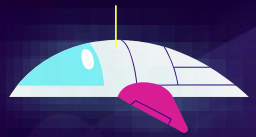
(GIT WIZARDS)

GIT AND GITHUB



Exploring the magic of Git & GitHub!

(GIT WIZARDS)

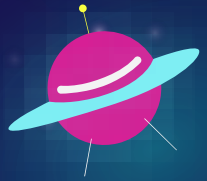


Table of contents

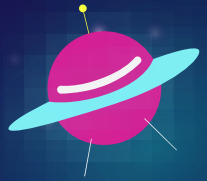


Table of contents

(GIT WIZARDS)

CH1

Intro

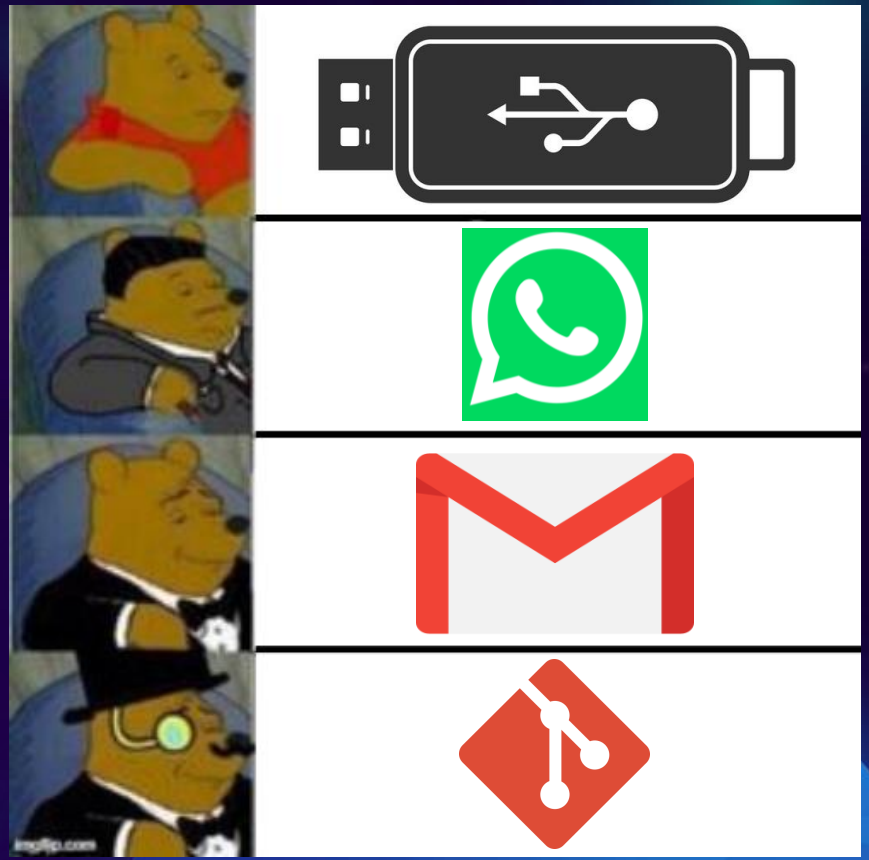
What Is Git?

Problem 1

How to keep track of your code's changes?

Problem 2

How to manage sharing the code among your team?



VERSION CONTROL SYSTEM

A system that is like a database of references that stores all the change that occurred to the code.

Features

- Compare changes over time.
- See who last modified file to fix bugs.
- It allows you to revert selected files back to a previous version.
- Revert the entire project back to a previous state

VERSION CONTROL SYSTEMS

Local

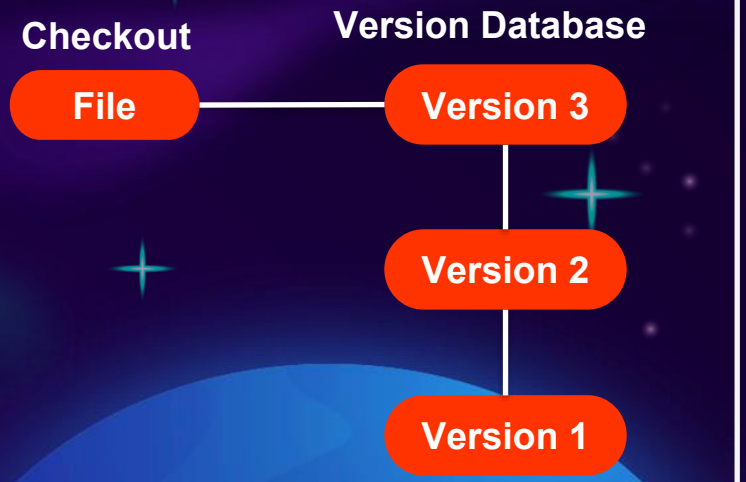
Stores all changes to files only on your computer.

Local Version Control

- A local database located on your local computer, in which every file change is stored as a patch.
- Every patch set contains only the changes made to the file since its last version.

Disadvantages

- No Collaboration
- Single point of failure



VERSION CONTROL SYSTEMS

Local

Stores all changes to files only on your computer.

Centralized

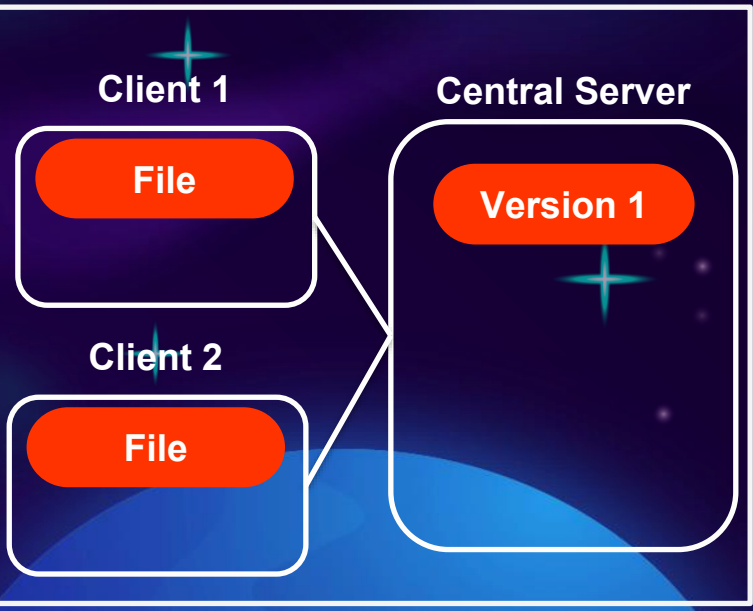
Only a single central server contains all the versioned files.

Centralized Version Control

- Enables multiple clients to simultaneously access files on the server, pull them to their local computer or push them onto the server from their local computer.
- This allows for easy collaboration with other developers or a team.
- Administrators have control over who can do what.

Disadvantages

- Single point of failure



VERSION CONTROL SYSTEMS

Local

Stores all changes to files only on your computer.

Centralized

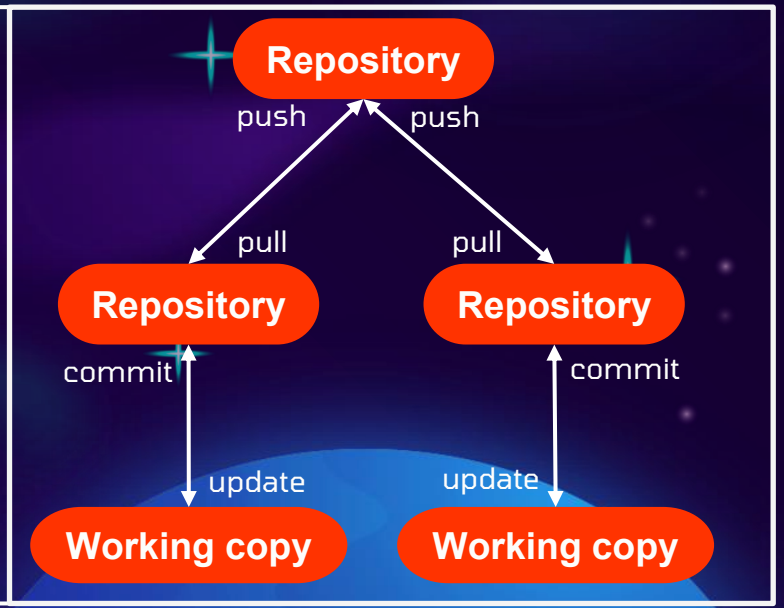
Only a single central server contains all the versioned files.

Distributed

A hybrid system that allows you to have copy on your computer.

Distributed Version Control

- Clients fully mirror the repository, including its entire history, rather than merely viewing the most recent snapshot of the files from the server.
- Each person working on a project has a local copy of the entire project, meaning they each have a local database containing their whole history.
- With this model, any client repository can send a copy of the project's version to any other client or back onto the server when it becomes available in the event that the server goes down or becomes unavailable.



Git

- An open-source distributed version control system tool that supports distributed non-linear workflows and tracks changes in any set of computer files.
- Developed in 2005 by *Linus Torvalds*, the creator of the *Linux* .

Repository

A central storage location for managing and tracking changes in files and directories

Working Tree

A directory (and its files and subdirectories) on your file system that is associated with a repository

File's States

Untracked 01

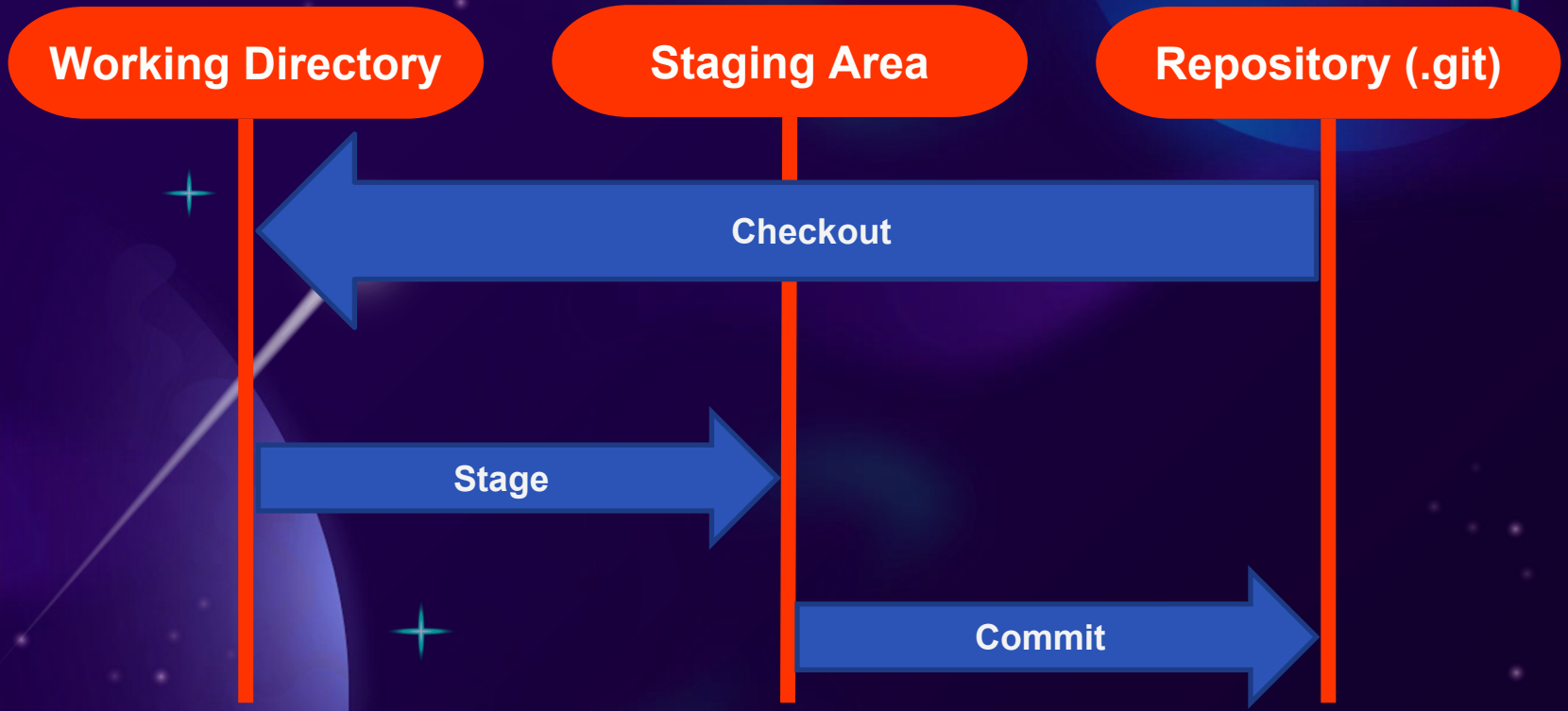
You have changed the file but have not committed it to your database (local repo) yet.

Staged 02

You have chosen a modified file in its current version to go into the next commit.

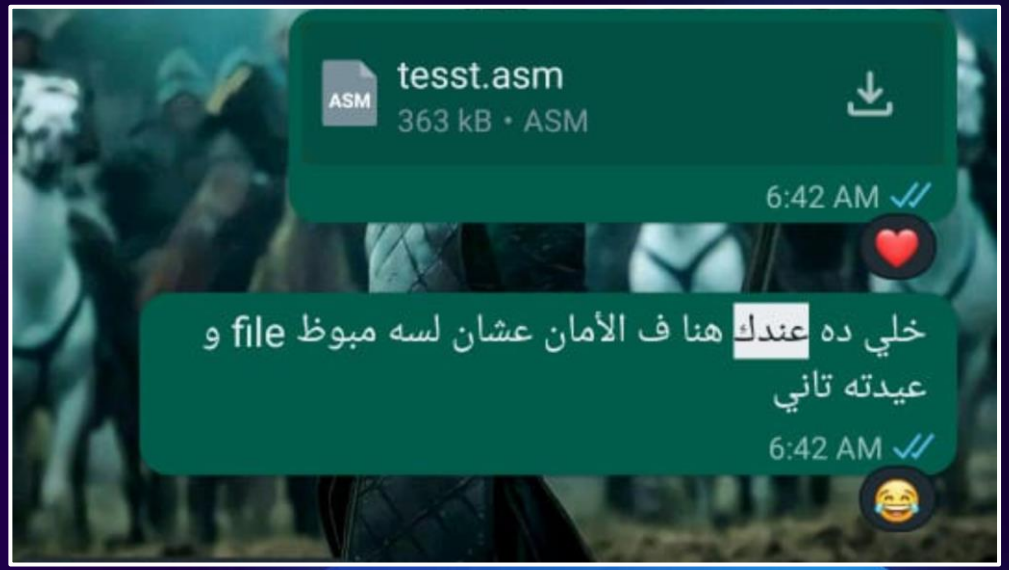
Tracked 03

The file has been committed into your repository.


Working Directory
Staging Area
Repository (.git)
Checkout
Stage
Commit

Git Is Awesome

BUT!!



Let’s make it more *Interactive*

GIT WIZARDS

Getting Started with Git

Git Config

The git config command is a convenience function that is used to set Git configuration values on a global or local project level. These configuration levels correspond to .gitconfig text files. Executing git config will modify a configuration text file.

<section>.<key>

1

local

By default ,git config will write to a local level if no configuration option is passed

2

global

Meaning is applied to operating system user

3

system

This covers all users on an operating system and all repos.

Git config --list

Git reads the configuration files and displays the values of all the configuration settings currently in effect.
includes settings such as user.name and user.email, which specify your name and email address to be associated with your Git commits. It can also show other settings like core.editor .

Git config --global user.name

To retrieve the currently configured global user name

To set your name :

git config –global user.name “your name”

Git config --global --unset user.name

By using the --unset option, you can remove the globally configured username in Git, effectively unsetting it. Once unset, Git will not associate any commits with that username, unless you provide a different username explicitly in a local repository or set a new global username.

NOTE

When you initialize Git, it is important to configure your user name and email address

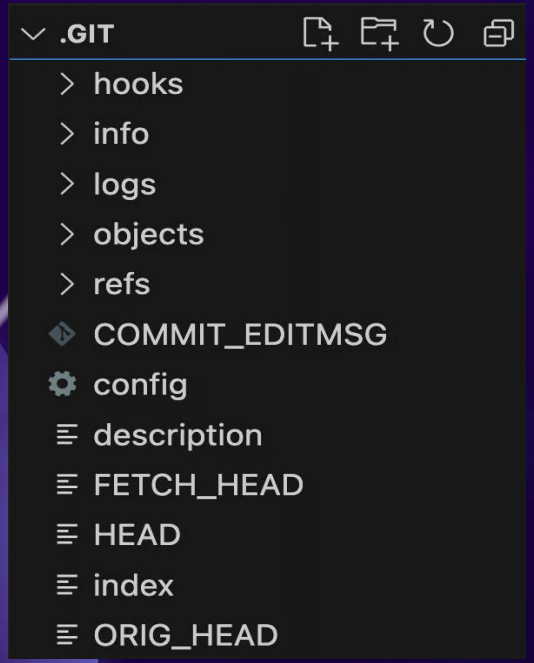
Create a new repo

Git init

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes subdirectories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit

steps



01

Create a new folder to be the repsitry location .

02

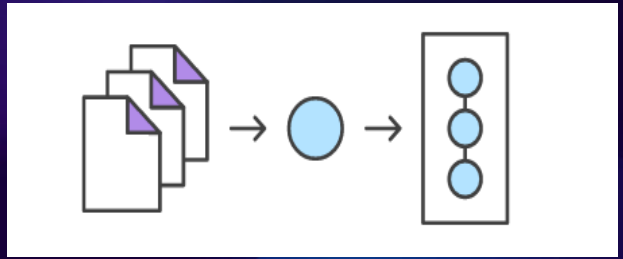
```
$ mkdir myfirstrepo
$ cd myfirstrepo
$ git init
```

Git add

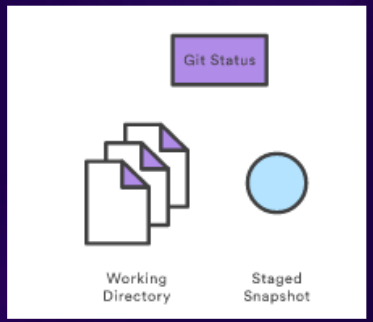
The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run
git commit.

Git commit

Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory. When you're ready to save a copy of the current state of the project, you stage changes with git add. After you're happy with the staged snapshot, you commit it to the project history with git commit. The git reset command is used to undo a commit or staged snapshot.



Git status

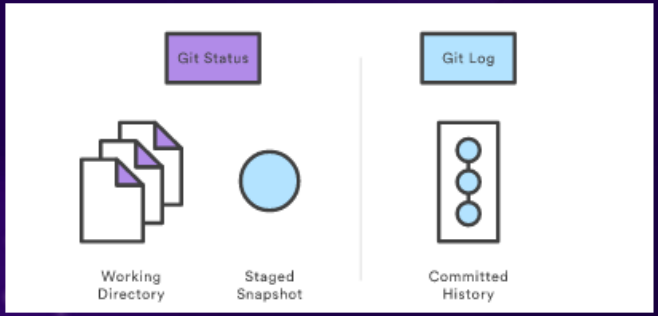


The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. Status output does *not* show you any information regarding the committed project history.

```
echo “BATMAN” >> README.md
```

```
git status
```

```
git commit -m “BATMAN”
```



Git log

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.

Git log

--oneline

shows the commit information on a single line making it easier to browse through commits at-a-glance.

--graph

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages.

(GIT WIZARDS)

CH3

Decoding Git

Let’s pop the hood and understand what git really is?

To follow along in our jorney

Rule1: We will pose a riddle

Rule2: Speculate an answer

Rule3: Investigate

Does git care who you are?

Git care only about one thing the .git folder

.git

Investigation

- Make a new dir called xgit and cd it.
- Command: git init
- Command: git status
- Cut the .git folder
- Paste it in root dir
- Command: git status

Outcomes Of Investigation

Does git care who you are?

.git

Git's storage model is neutral towards different users. It stores all its state in the .git folder and before executing nearly any command. It goes and find the .git folder to know what to do.It searches in the current working directory and its parents

How git stores its state in .git folder

Option1: storing diffs between one commit and another.
Option2: storing the entire repo at each commit.

.git

Investigation

- Command: git log

(CH3: Decoding Git)

- Git refers to each commit with something like this

`0c96c651ea67b5d191c5e54460c49a7fd8cf4e07`

- 40 symbol. Each symbol is between a-f and 0-9
- Same size for different commits

- Hashing: refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions.

One-Way: It's impossible to reverse a hashing process.

Non-Locality: In the case of the non-local hash function, similar inputs generate dissimilar outputs.

- Let's look for our commit in the .git `0c96c651ea67b5d191c5e54460c49a7fd8cf4e07`

```
.git/objects/0c
.git/objects/0c/96c651ea67b5d191c5e54460c49a7fd8cf4e07
.git/objects/1d
.git/objects/1d/9e56c6a66329129c80893a3e496901b6ad0a39
.git/objects/1f
.git/objects/1f/cbcccd777291c3f7effec7e8763236c1b81ef0
.git/objects/30
```

If you try to open these files you will find compressed data.

> So a SHA in git is just a pointer to something

- If you try to open these files you will find compressed data.
  Let's uncompress it

.git

Investigation

- Command: git cat-file –p `0c96c651ea67b5d191c5e54460c49a7fd8cf4e07`

- What is in a commit ?

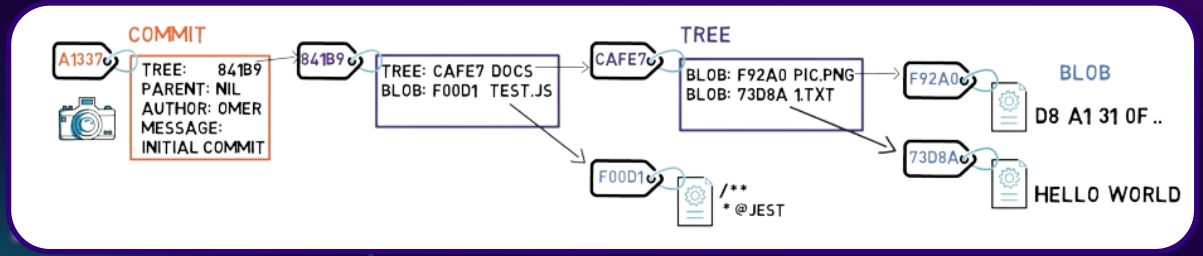
- Meta-data
- Other SHAs

```
$ git cat-file -p  0c96c651ea67b5d191c5e54460c49a7fd8cf4e07
tree 4d5ba58529fbbcba5d538c3315db54227505e0a0
author AhmedHamdiy <ah2882003@gmail.com> 1713329033 +0200
committer AhmedHamdiy <ah2882003@gmail.com> 1713329033 +0200
```

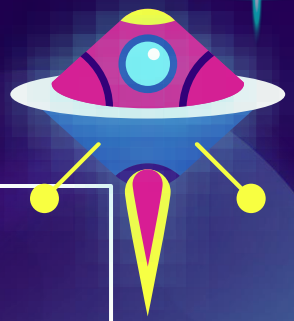
What these other SHAs refer to

- Parent --> The commit that precedes current commit
- Tree --> The main directory that the git tracks

> So a commit stores a snapshot of the entire repository.



CH4



Branches

merge & rebase

Oh, do you really develop everything on main?
That's a problem.

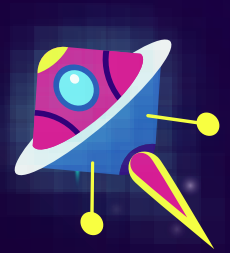
Branches

We all know what is a branch but nobody asks why is a branch.

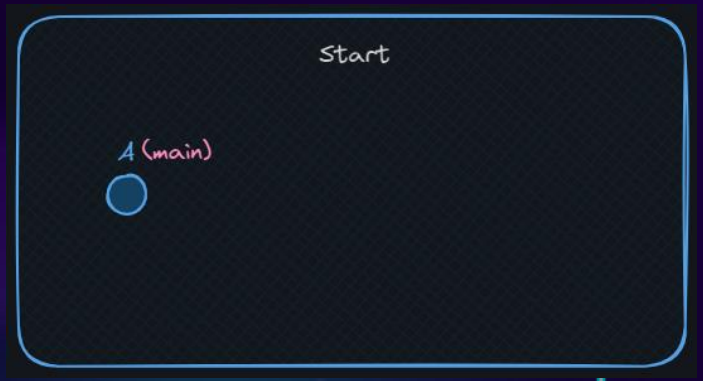
“Branching means you diverge from the main line of development and continue to do work without messing with that main line, to be safe and keep the main safe from you.”

(CH4: BRANCHES)

Mission 0



Desired State



git-branches

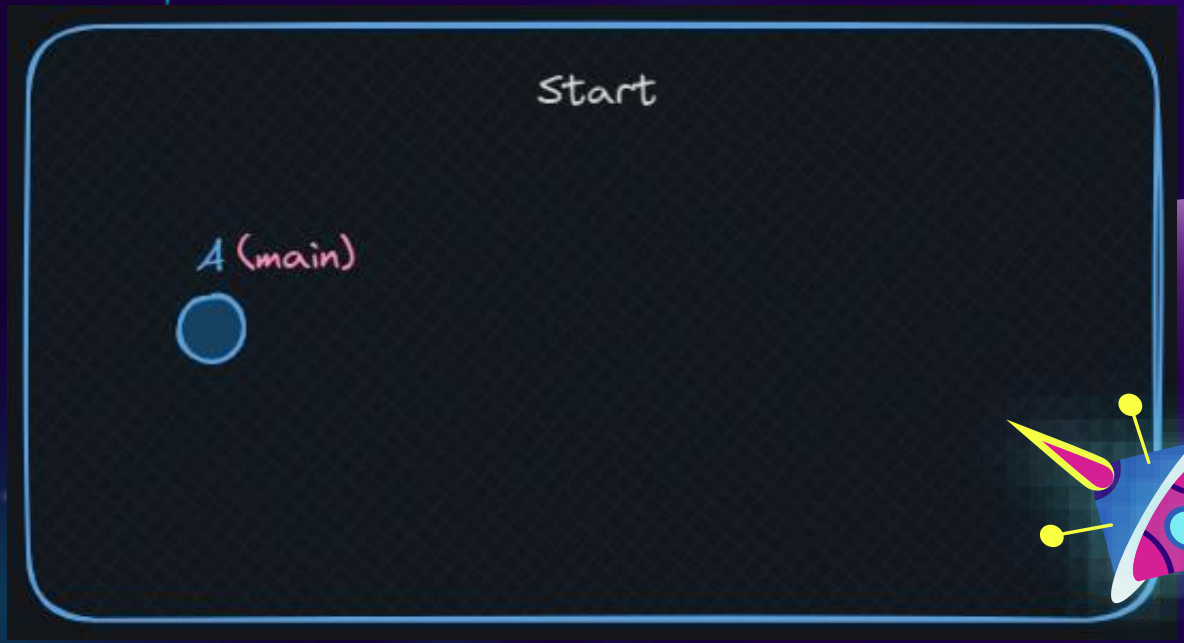
Code your way to succeed

Prepare your repo:

- Make a new dir called `git-branches`
- Make a repo in this dir
- Commit A

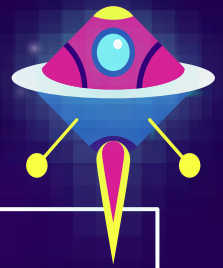
Mission 0

Desired State



(CH4: BRANCHES)

Mission 0



Great Job!

Mission 0 status: done

git-branches

```
mkdir git-branches
cd git-branches
git init
touch A
git add .
git commit -m A
```

Creating branches

- To create a new branch:

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

- To switch to a branch:

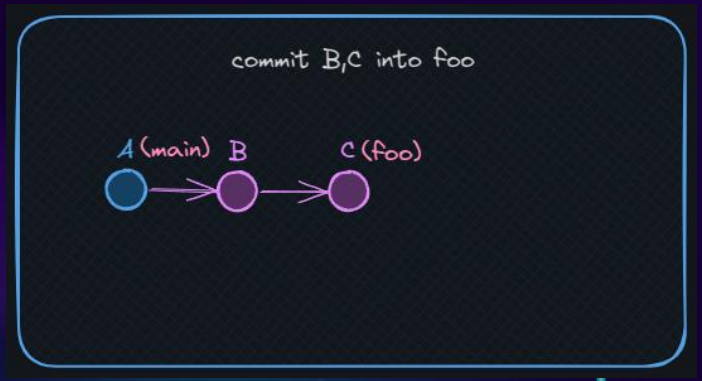
```
git checkout <branch-name>
```

⚠️ **Attention**: after you create a branch git doesn't automatically switch to it

(CH4: BRANCHES)

Mission 1

Desired State



git-branches

Code your way to succeed

Your first branch:

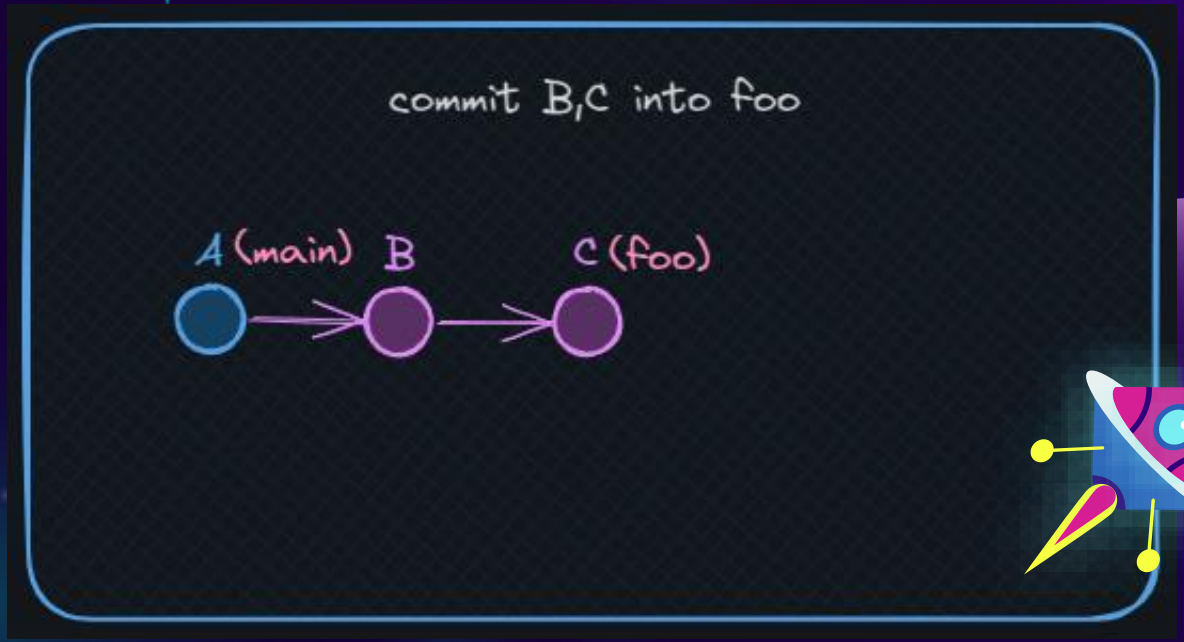
- Create a branch “foo” and switch to it.
- find foo in .git
- Commit “B” and “C” into foo

Note: make the change the same as the commit message

(CH4: BRANCHES)

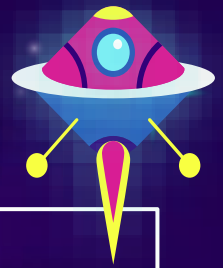
Mission 1

Desired State



(CH4: BRANCHES)

Mission 1



Great Job!

Mission 1 status: done

git-branches

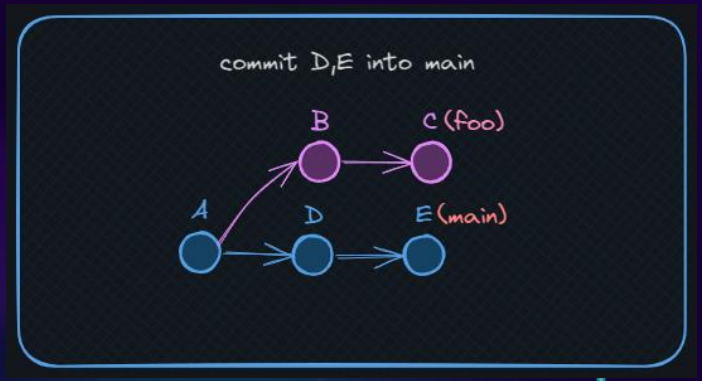
```
git branch foo
git checkout foo
find .git

# refs/heads/foo
```

```
Touch B
git add .
git commit -m B
Touch C
git add .
git commit -m C
```

Mission 2

Desired State



git-branches

Code your way to succeed

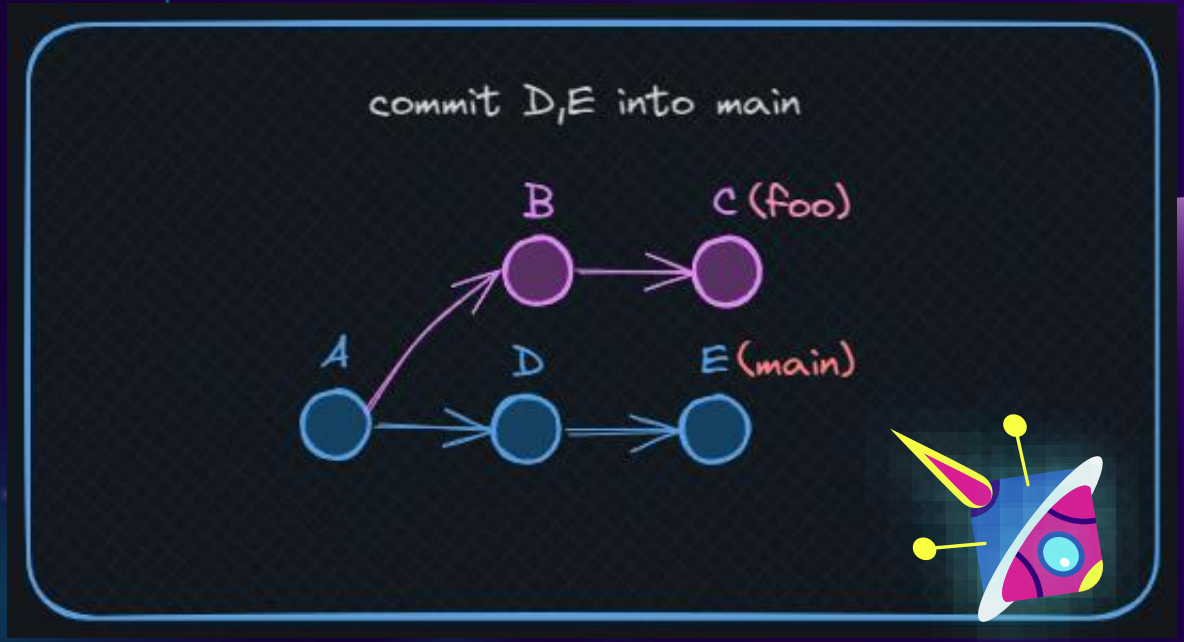
Things happen on main, you know:

- Switch to main
- commit “D” and “E”

NOTE: make sure to commit in a new file
We don’t want conflicts yet 😉

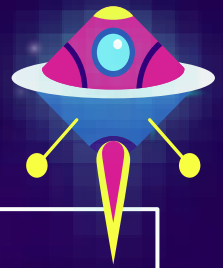
Mission 2

Desired State



(CH4: BRANCHES)

Mission 2



Great Job!

Mission 2 status: done

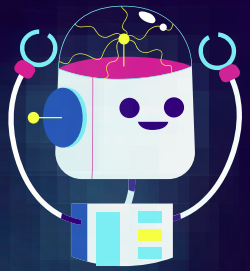
git-branches

```
git checkout main
Touch D
git add .
git commit -m D
```

```
Touch E
git add .
git commit -m E
git log --graph
--oneline --parents
```

Combining Your Work:

In git, there are 2 main ways to join branches:



merge

rebase

merge

"A merge is attempting to **combine** two histories together that have diverged at some point in the past. There is a common commit point between the two, this is referred to as the **best common ancestor**"

- The docs

merge

To merge a branch into the currently checked branch:

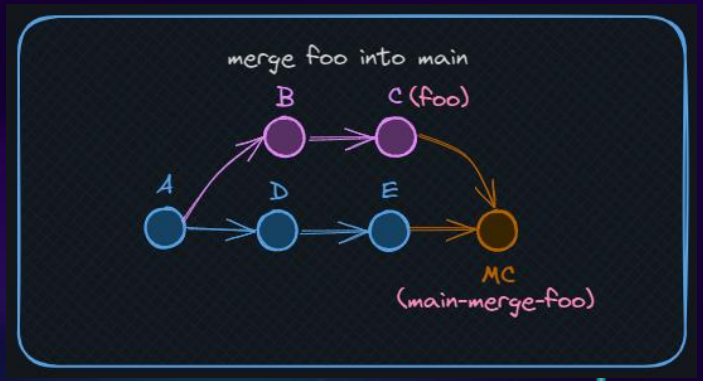
```
git merge <source-name>
```

Merge has 2 outcomes depending on the state, more on that later.

⚠ **NOTE**: the target-branch is the currently checked branch

Mission 3

Desired State



git-branches

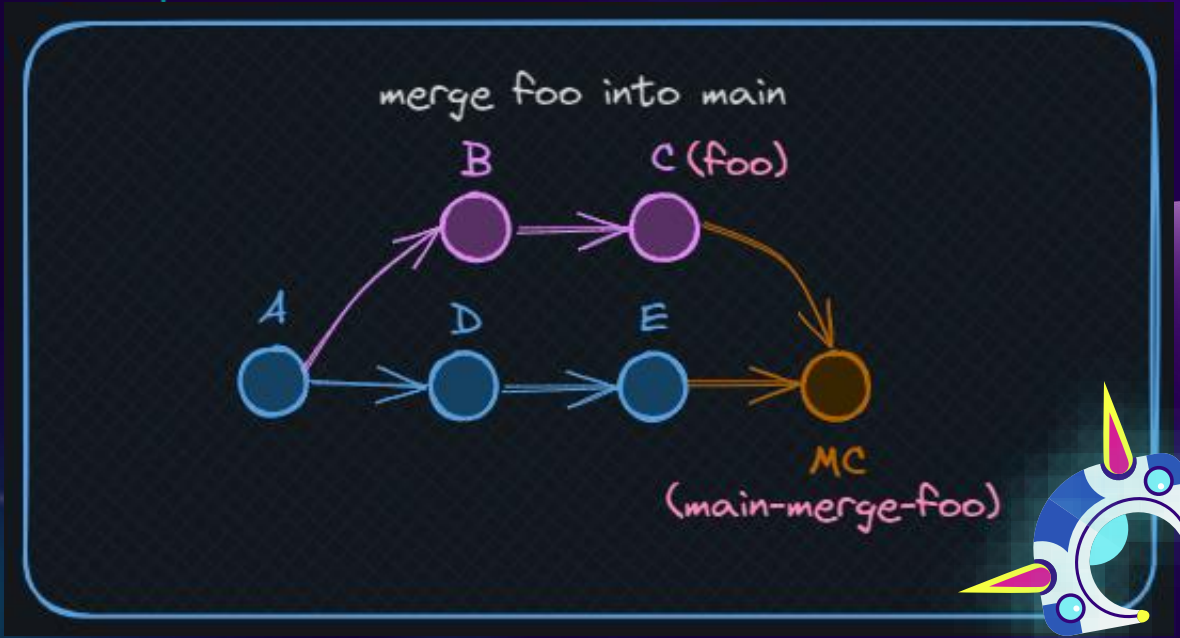
Code your way to succeed

Let's merge:

- Create a new branch "main-merge-foo" of main.
- Merge foo
- Look at the graph

Mission 3

Desired State



(CH4: BRANCHES)

Mission 3

Great Job!

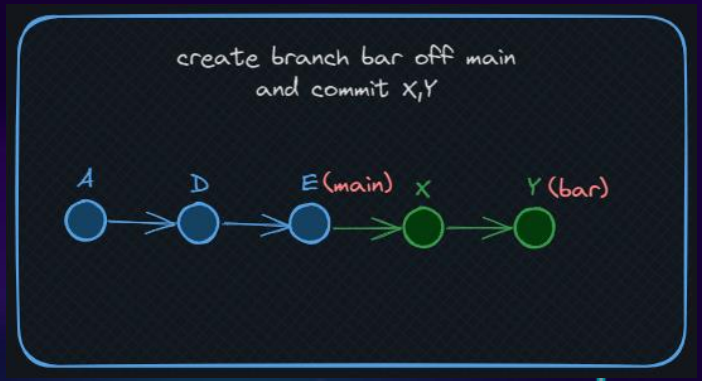
Mission 3 status: done

git-branches

```
git checkout main
git checkout -b main-merge-foo
git merge foo
git log --graph --oneline --parents
```

Mission 4

Desired State



git-branches

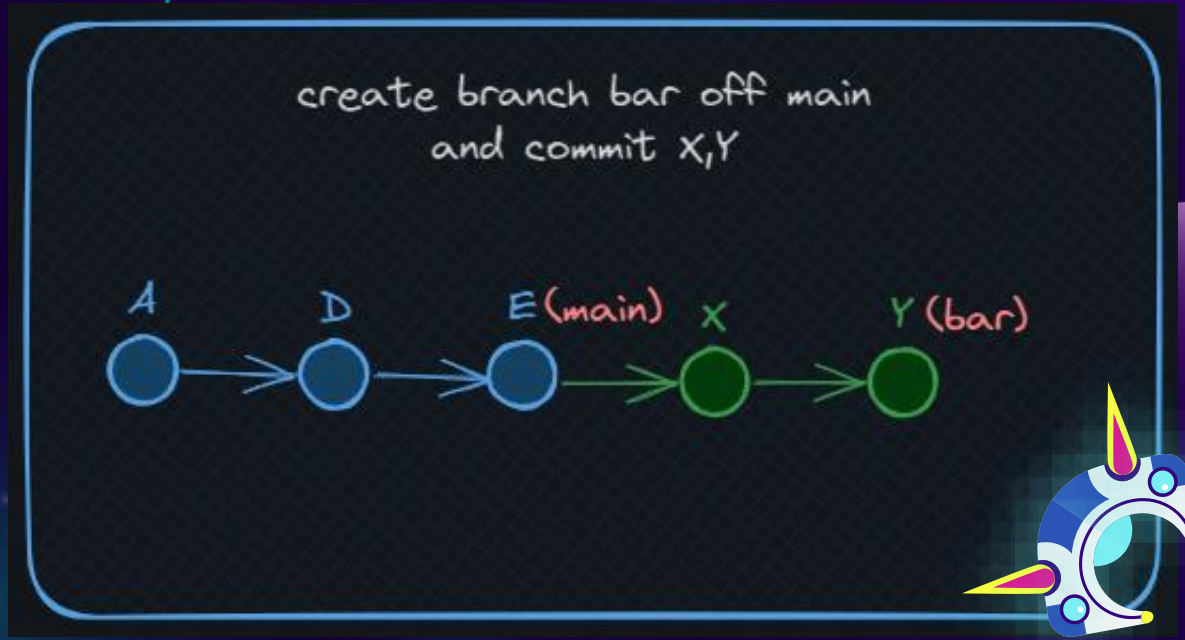
Code your way to succeed

Let's Fast-Forward merge:

- Make a new branch "bar" off main.
- Commit "X" and "Y"
- Merge bar into main

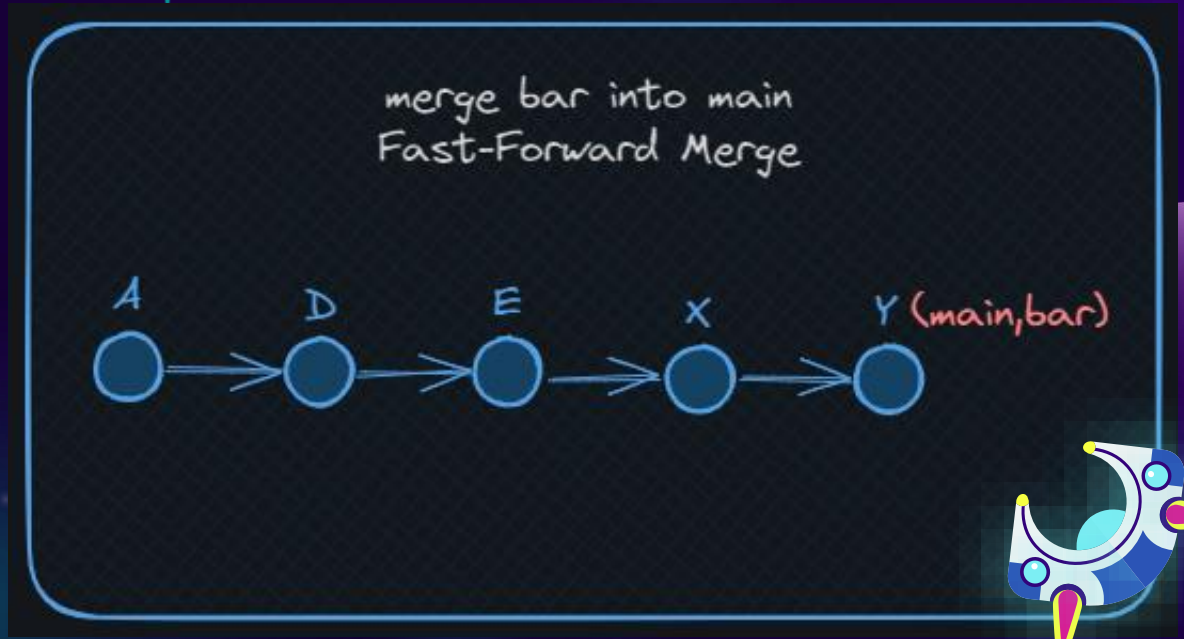
Mission 4

Desired State



Mission 4

Desired State



Mission 4

Great Job!

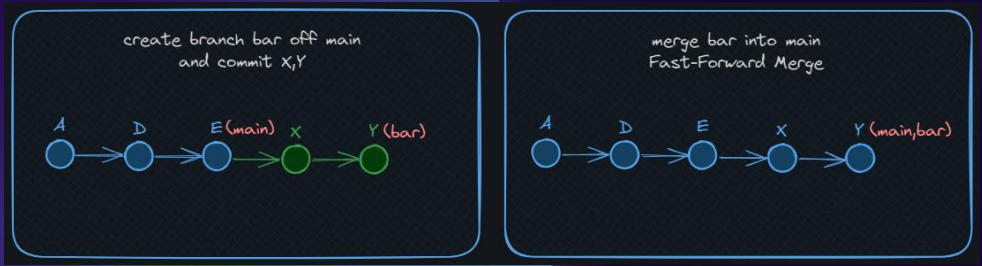
Mission 4 status: done

git-branches

```
git checkout main
git checkout -b bar
touch X
git add .
git commit -m X
touch Y
git add .
git commit -m Y
git checkout main
git merge bar
```

Merge Outcomes:

- **Fast Forward Merge:**
  just update the pointer/reference (no merge commits)



- **Divergence Merge:**
  create a merge commit to combine 2 commits/histories have 2 parents

rebase

“git-rebase - `Reapply` commits on top of another base tip”
- "the docs"

- **To rebase a branch:**

```
git rebase <target-branch>
```

⚠ **NOTE:** rebase is often used at your private feature branch, so target-branch is often **“main”**

rebase

- **How rebase works:**

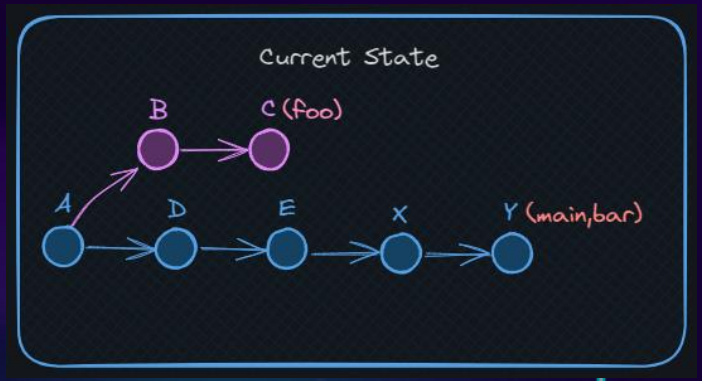
1. checkout the latest commit at `<target-branch>`
2. replay one commit at a time of the `<source-branch>`
3. update source branch ref to the latest commit made.

Rebase doesn't create merge-commits.

⚠ **NOTE:** rebase **alters** history and creates new commits as commits in git are immutable

Mission 5: last mission in this chapter

Current State



git-branches

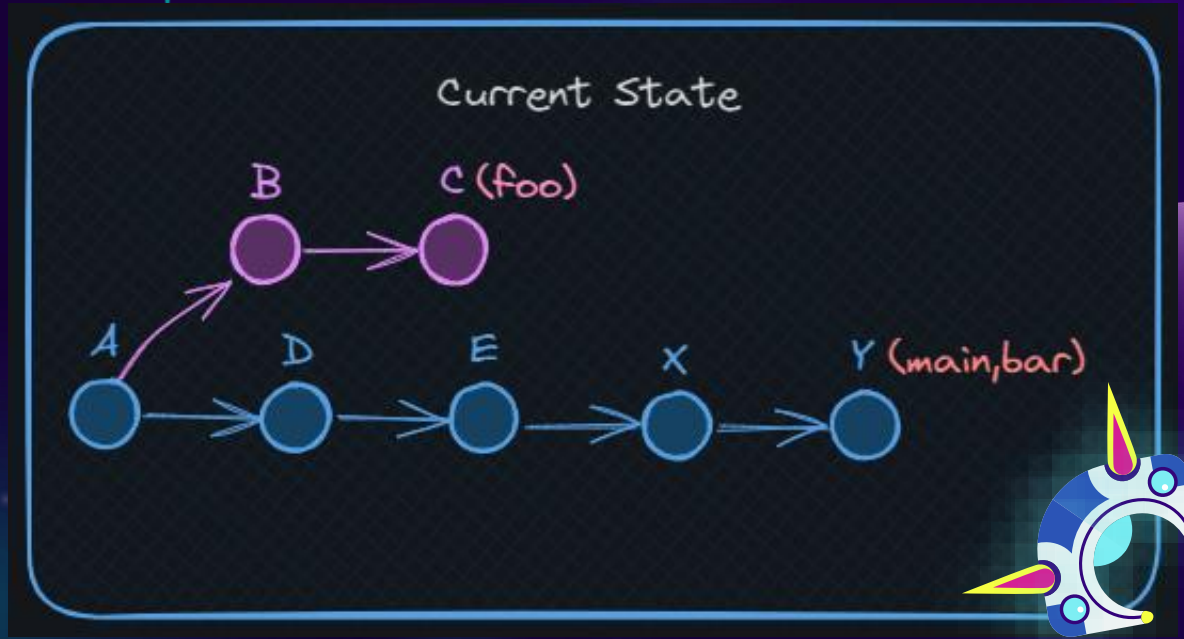
Code your way to succeed

Rebase Your feature branch:

- Checkout foo
- Rebase off main

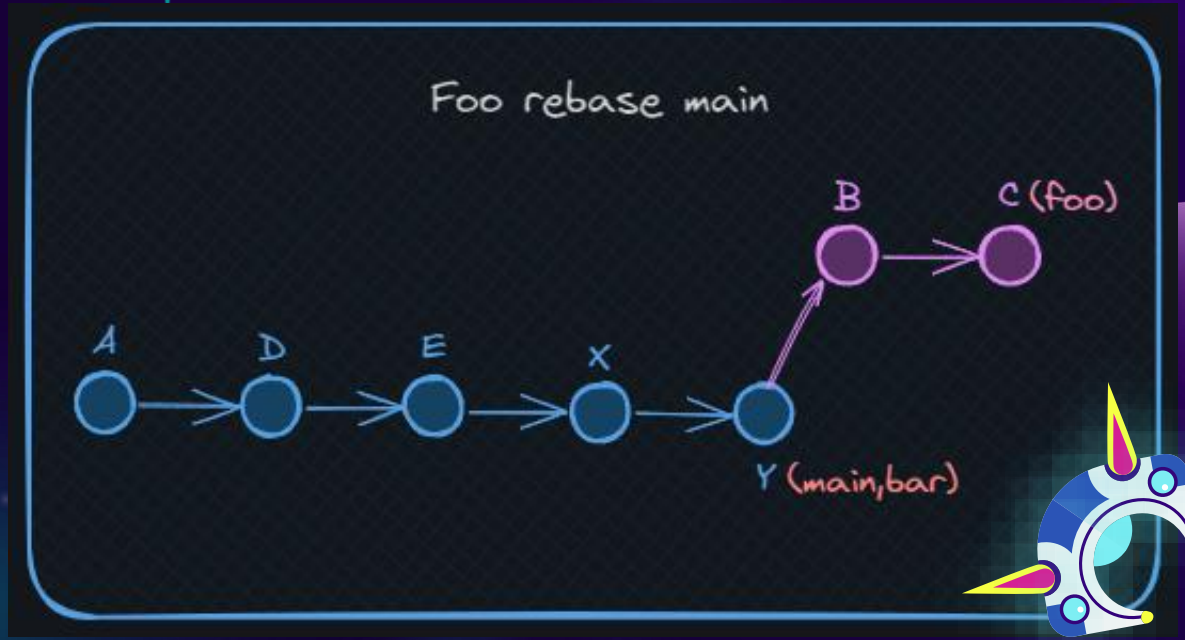
Mission 5

Current State



Mission 5

Desired State



Mission 5: last mission in this chapter

Great Job!

Mission 4 status: done

git-branches

```
git checkout foo
git rebase main
git log --graph --oneline
```

Merge VS. Rebase

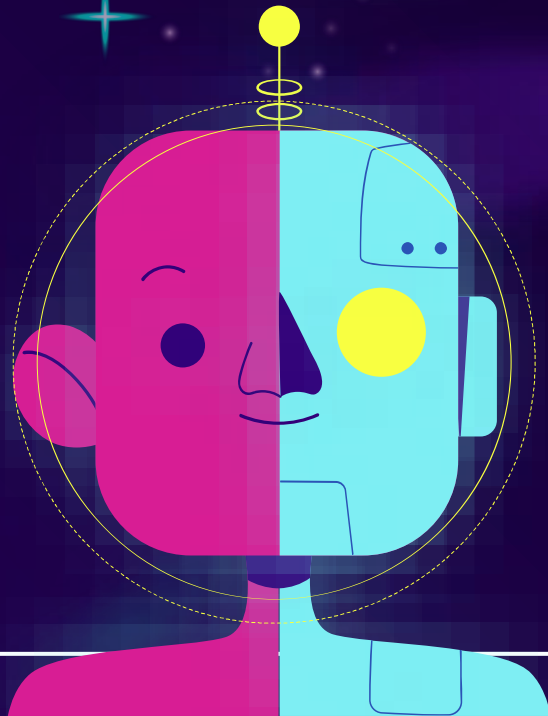
Merge

doesn't **alter** history 👍

doesn't require **push force** 👍

works with **private** and 👍
'**public** branches

makes annoying merge 👎
commits



Rebase

👍 no annoying merge commits

👍 linear history which is easier to search

👎 alters history

👎 requires `push force`

(CH4: BRANCHES)

Workflows Wars

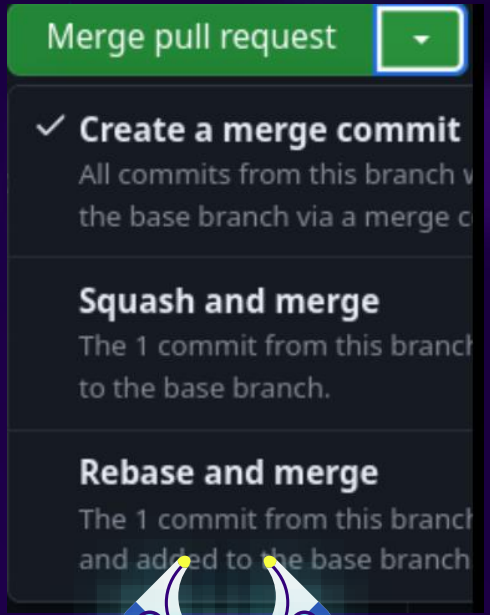
Workflow wars

Merge Workflows (just merge)

ℹ️ merge back into main

👎 annoying merge commits

⚠️ **NOTE1:** people are always so **opinionated** about which workflow is better, but **rebase** is diffidently the **best** 😉



Rebase flow (rebase in private - FF merge in public)

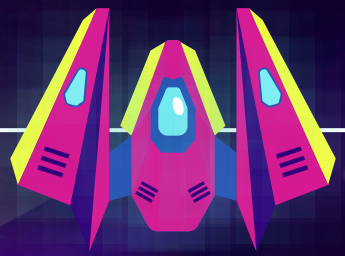
ℹ️ rebase main into your branch first, then fast forward merge into main

⚠️ **NOTE2:** the difference between the 2 flows in simple form compiles to which option you choose in GitHub pull request.

CH5

What is HEAD?

(HEAD -> main)



HEAD

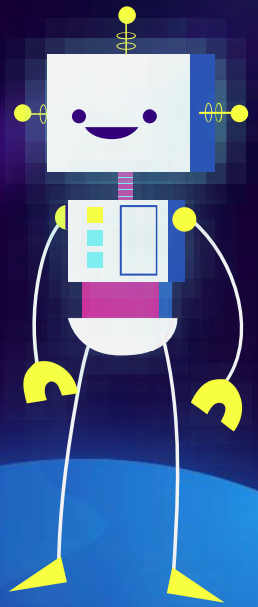
Is a reference variable in git. It points to the tip of your current branch.

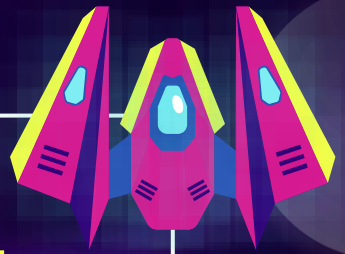
Like everything, it's stored inside *.git/*

Find it inside .git/HEAD

The content is simply a reference to the tip of your current branch (e.g. refs/heads/main).

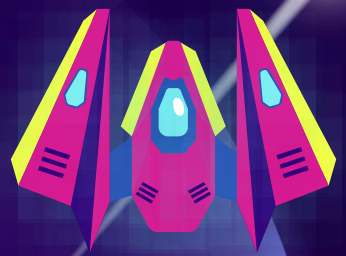
The "tip of a branch" is the latest commit made on that branch. It is stored inside .git/ as a SHA.





Great concept. What can we do with it?

Git Reflog



01

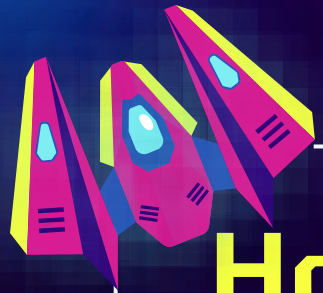
Git stores the full history of HEAD's movements.

02

When you switch from branch to branch or make new commits, it records it. Commits are stored as SHAs.

03

Everything is recorded, all the way back to *git init*!

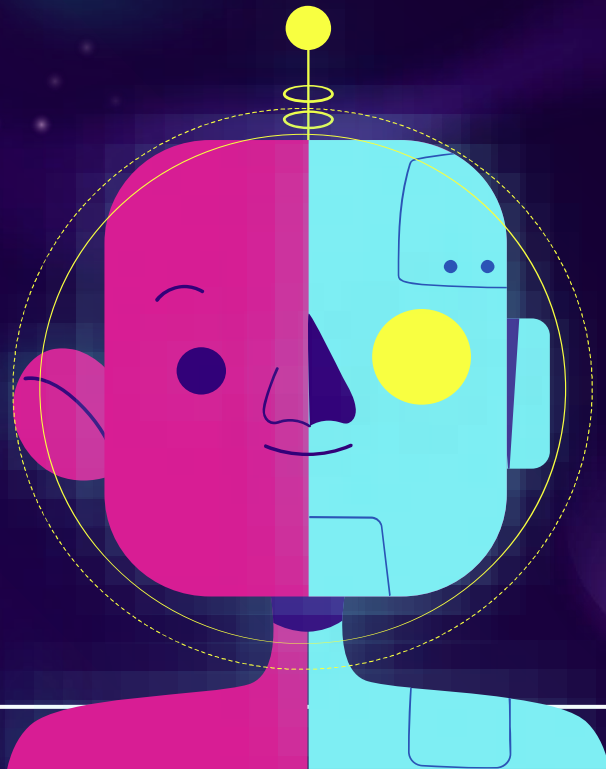


How can git reflog be a powerful feature?

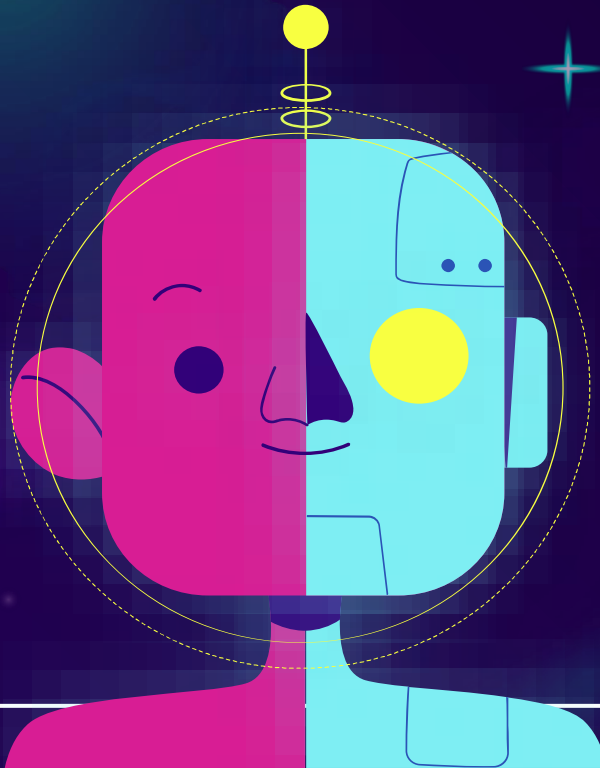
One example is restoring commits!

A. Through merging

Even if a branch gets deleted, its commits remain recorded in reflog. This means you can run ***git merge <SHA>*** and have your commits back.



One example is restoring commits!

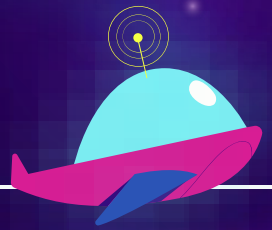


B. You could also cherry-pick!

If merging becomes too problematic, you can cherry-pick instead! Cherry-picking a commit means merging only the changes this commit introduces, and none of its previous history.

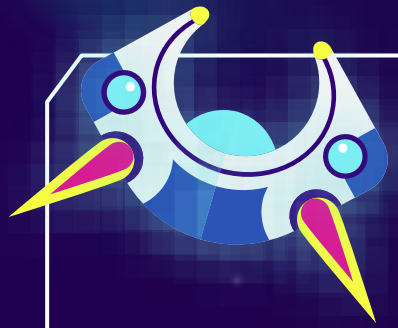
Git cherry-pick <SHA>

CH6



Remote

Are you ready to deal with the outside world?



What's remote?

Have you thought about GitHub on hearing "remote"?

Remote repositories are versions of your project somewhere else.

It is entirely possible that you can be working with a "remote" repository that is on the same host you are.

Remote does not necessarily imply that the repository is somewhere else on the network or Internet, **only that it is elsewhere.**

NOTE:

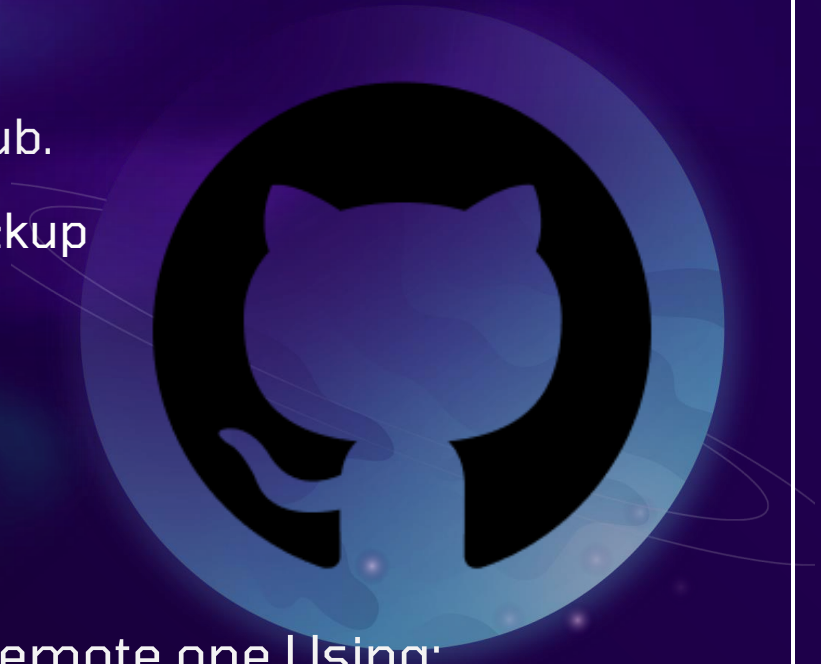
- Remote repositories can be on your local machine.

GitHub

In summary, going remote doesn't necessarily mean using GitHub.

GitHub is used with git for online backup and collaborating with others.

⚠ Throughout this chapter we'll act like we're using GitHub, but we'll actually use a remote repo in our local machine (same concepts are applied to GitHub).



We can **link** our local repo with a remote one Using:

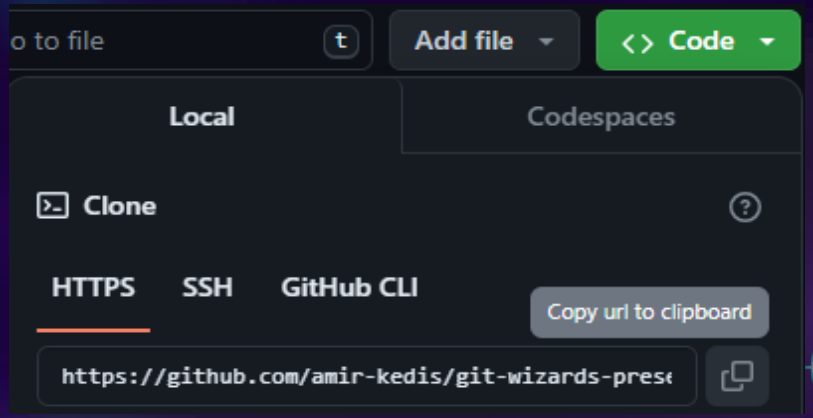
Git remote

01 Using git remote

```
git init
git remote add <remote> <url>
```

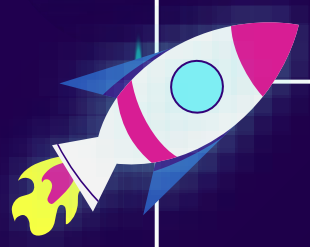
02 Using git clone

```
git clone <url> <folder-name>
```



Note:

- Using git clone creates a folder with git initialized in it. No need for git init.
- Specifying the folder name is optional.



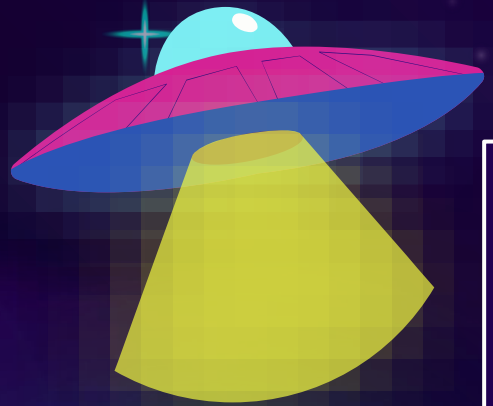
TODO: Create a new repo and add git-branches as its remote...

```
mkdir local
cd ./local
git init
git remote add origin ../git-branches
git remote -v
```

NOTE:

`git remote –v` **to list out your remotes and their locations.**

Git fetch



Git fetch

What happens when you fetch?

- Git contacts the remote repository.
- Retrieves any new commits, branches that exists in the remote repo and doesn't exist in your local one.
- Updates your local repository's remote-tracking branches to reflect the remote repo state

Git fetch

Git fetch

However, the command does not:

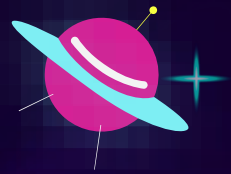
- Change the state of your current working branches.
- Merge any changes into your local branches.
- Modify any of your local files or commit history.

In summary:

fetch is used to update all the remote tracking branches as they are not updated automatically.

Note:

You can use git merge to merge the updates.



Git pull

You can use pull instead of fetching then merge to merge the remote changes into your local branch.

All you need to do is just:

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

Note:

- git pull is a combination of git fetch and git merge.

TODO: Fetch the changes from remote repo then pull them

```
git fetch
git branch -a
git pull origin main
```

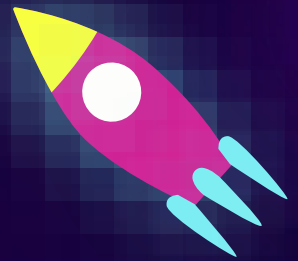
Notice how state didn't change on fetching

Git push

What if you're the one who made changes and want to make it visible in the remote repo

Just the opposite of pull

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



TODO: Create a new commit in local repo then push it to remote

```
git branch local
git checkout local
echo “local changes” >> README.md
git add .
git commit –m “local updates”
git push origin local
```

TODO: Create a new commit in both repos where both change README.md last line with:

Hello from local

local

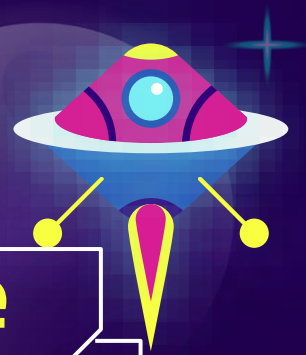
```
vim README.md
git add .
git commit –m "local"
git pull origin main
```

Hello from remote

remote

```
vim README.md
git add .
git commit –m "local"
```

Merge Conflicts



local

Hello from local

remote

Hello from remote

CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

Merge conflicts happens when git cannot automatically merge (e.g., Making different changes in the same line of the same File)

File With Conflicts

```
<<<<<<< HEAD
Hello from local
=======
Hello from remote
>>>>>>> 3481fb2d429347f009646b1456375e6685c1a36e
```

- Any `>>>>`, `=====`, `<<<<<` denote parts of the conflict.
- `<<<<<<< HEAD`
- This states that HEAD's conflicted change starts here and continues until the `=====` line.
- `====` denotes the separation of the two merges.
- The end of the merge conflict is denoted **with `>>>>` and the SHA** of the incoming conflicted change.

To Resolve Merge Conflicts:

You just need to remove the lines starting with >>>>>>
And ======
And either:

01 Accept local (current) changes

02 Accept remote changes

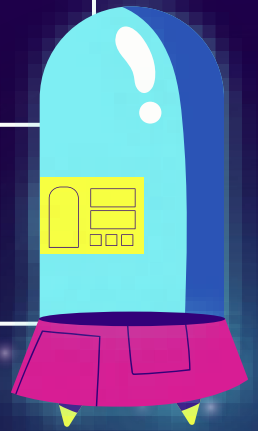
03 Combine both

04 Add completely different changes

CH7

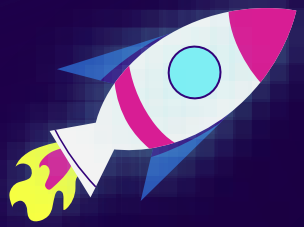
Fancy Git Commands

Re-writing History As You Wish.



Amending Commits

Amending (editing) Commits



01 Make the edit you forgot

02 Stage it

03 Now, instead of creating a new commit, you can edit (amend) the last commit directly

Try it yourself

Ex1: Make a wrong commit, then fix this same commmit

Amend

```
echo “I don’t love you” > cmp.md
git  add .
git commit -m “confession”
# realize you were wrong ..
# how to fix it ?
echo “I love you” > cmp.md
git commit --amend --no-edit
```

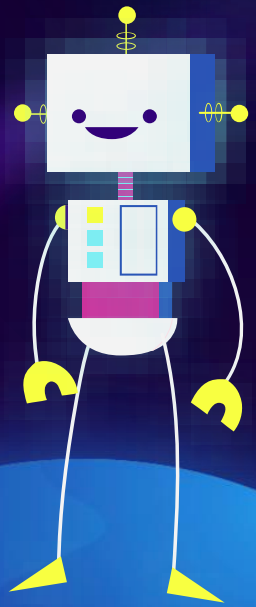
You can check before & after using git log

Interactive rebase

Edit History (not just the last commit)

You can do these and more..

- Change order of commits
- Amend (edit) older commits
- Squash (combine) commits
- Delete commits
- ....



How to start an interactive rebase?

1. View log to understand
2. Count commits back from head that you want to edit
3. Then ,...

git rebase -i HEAD~n

Where n is the number of commits back from HEAD to edit.

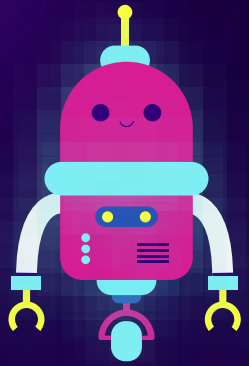
Try it yourself

Ex2: commit 4 changes ‘A’ ‘B’ ‘C’ ‘D’ to a file

rebase -i

```
echo A >> hello.md
git  add .
git commit -m “A”
# repeat for B, C, D
git log --oneline
git rebase -i HEAD~4
```

Reword, Reorder, Squash



No boring sildes here.
Let’s do it together,
Don’t be lazy

Stash

01 You made some changes

02 You don’t want to commit yet, and you don’t want to lose them

03 Stash them for now (Something like Add to watch later)

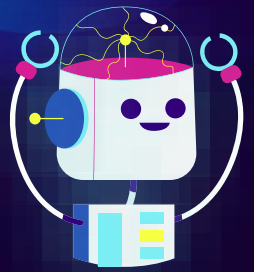
Stash

Push	git stash (OR) git stash -m “draft”	**To add an entry** ***(changes must be saged first)***
Pop	git stash pop	**To retrieve an entry**
Drop	git stash drop	**Drop one element**
Clear	git stash clear	**Clear Everything**

Stash

⚠ Attention

⚠ Adding changes to stash **removes** them from the working tree
(currently used files)



⚠ **git stash pop**
may produce conflicts!
It changes your local files and removes the selected entry from stash

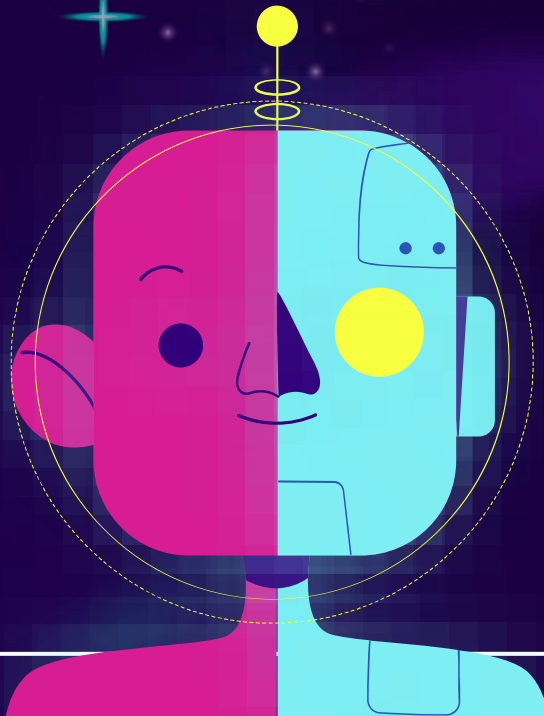
(CH7: Fancy Git)

Undo: Revert / Reset

Revert us Reset

Revert

- Creates another commit
- Typically used for public branches
- covers the mistake



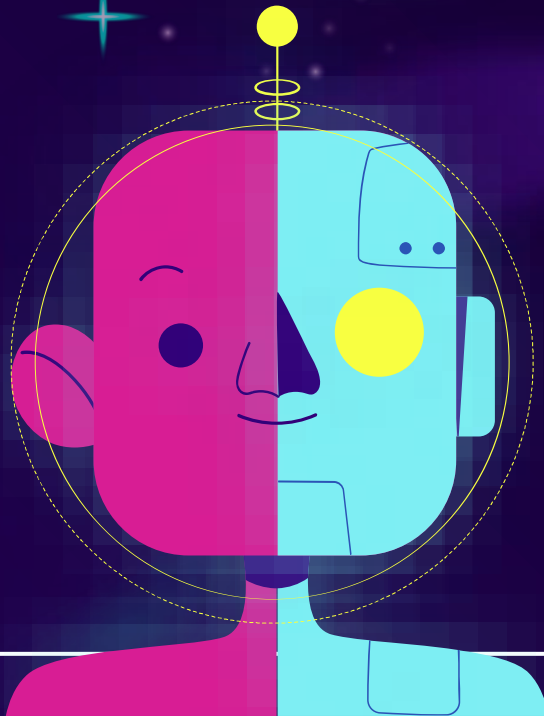
Reset

- Does not create a new commit
- Typically used for private branches
- deletes the mistake

Reset
soft vs hard

reset --soft

- commit are deleted
- their changes in the working tree are preserved



reset --hard

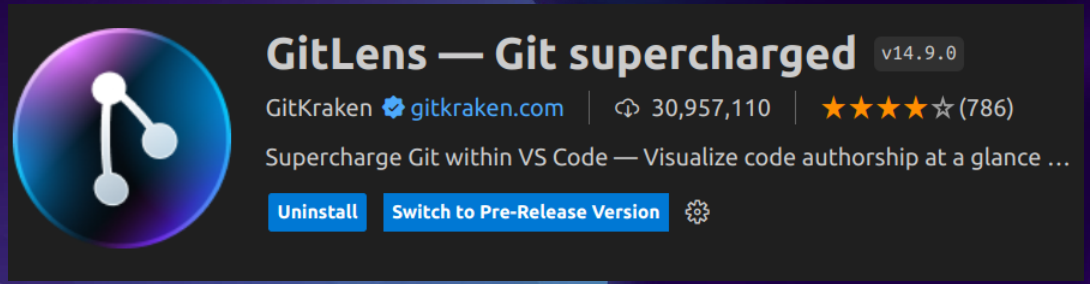
- commits are deleted
- working tree is checked out to the last commit

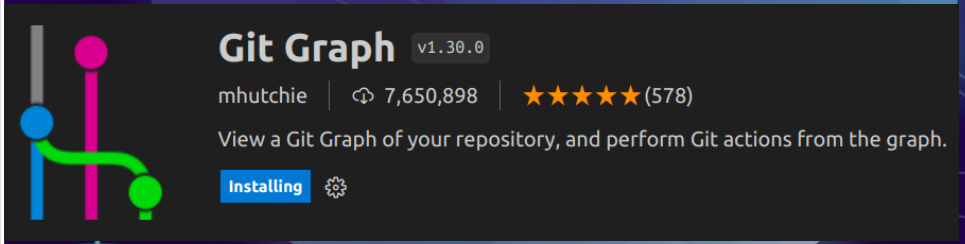
CH8

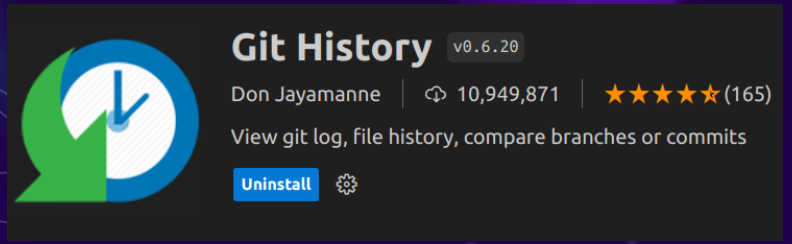
Tools & More

Faster Workflows and more to look for

Lazygit, and more alternatives







Tags & releases

Tag: Mark a specific point (commit) in history

GitHub Release: Publish a tag to the world

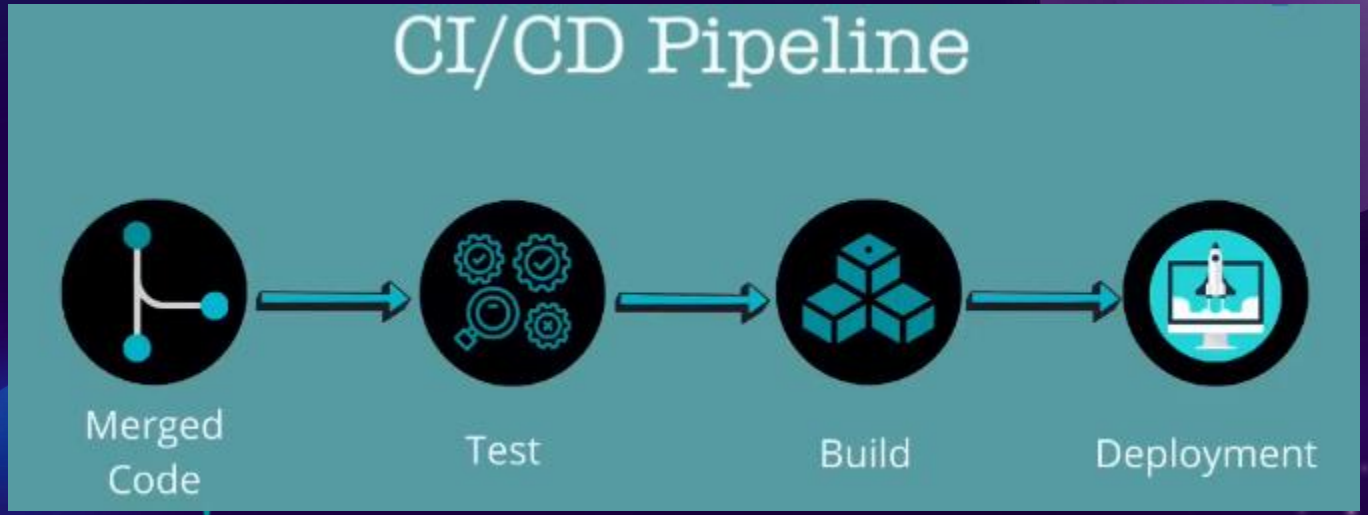
Worktrees

```
→ git-worktree.nvim git:(master) ls
branches  config  description  HEAD  hooks  info  objects  packed-refs  refs
→ git-worktree.nvim git:(master) git worktree add master
Preparing worktree (checking out 'master')
HEAD is now at 6de685a fix(indenting): Tree shitter to the rescue
→ git-worktree.nvim git:(master) git worktree add readme
Preparing worktree (checking out 'readme')
HEAD is now at 8101dae feat(readme)
→ git-worktree.nvim git:(master) git worktree add foobarbaz
Preparing worktree (new branch 'foobarbaz')
HEAD is now at 6de685a fix(indenting): Tree shitter to the rescue
→ git-worktree.nvim git:(master) ls
branches  description  HEAD  info  objects  readme  worktrees
config  foobarbaz  hooks  master  packed-refs  refs
```

before

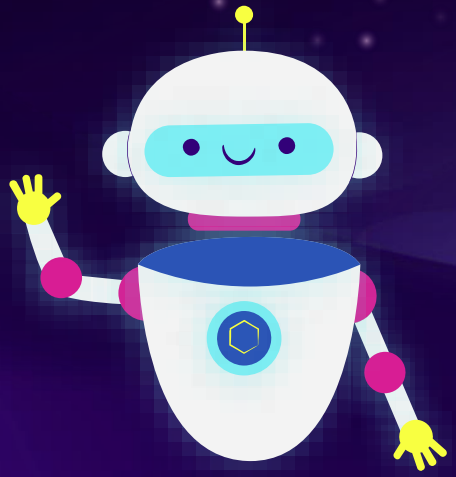
after

Workflows & GitHub Actions



(SPECIAL EFFECTS)

Thanks!



Do you have any questions?

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, infographics & images by **Freepik**

Quiz

1. Git is a ..... Version control system.

A. Distributed B. Central C. Local D. None

2. What happens when you use git add on a file that is already staged?

A. The file is unstaged
B. Nothing, the file remains staged
C. The file is deleted from the repository
D. The file is committed immediately

3. How does git internally represent a branch?

A. A separate copy of the entire repo
B. A light-weight pointer to a specific commit
C. A special file containing all changes made on that branch
D. Git doesn't support branching. This is a GitHub feature

4. Merge has ..... outcomes

A. Just 1 outcome
B. 2 outcomes (fast forward, divergence)
C. 2 outcomes (rebase, merge)
D. 3 outcomes or more

Quiz

5. ..... alters history.

A. Rebase B. Merge C. Push D. Nothing alters history in Git

6. This command shows you the full history of HEAD's movements through branches and commits.

A. Git reflog
B. Git revert
C. Git pull --rebase
D. Git config --get-regexp <branch>

7. What is the command to get all the change history of a remote repository?

A. git add remote
B. git fetch
C. git merge
D. git push

Quiz

8. What will the *git remote -v* command print in the terminal?

A. The current git version you're running
B. A list of remote repositories and their Urls
C. The last 5 git versions you've installed
D. An inline editor for modifying remote repositories

9. To edit the last commit without creating a new one, you can use:

A. git commit
B. git commit --amend
C. git amend
D. git revert

10. What is the purpose of creating tags in Git?

A. To mark specific points in the Git history for future reference
B. To create a new branch from a specific commit
C. To undo the effects of a commit
D. To temporarily save changes from the working directory