El zataoona

Presented by the git wizards

- 🪄 Asmaa - Mariam - George - Khalid 🪄
- 🪄 Ahmed Hamdey - Shehab - Amir

outline

.1. Intro

What is Git ?

> *Git is a open-source distributed version control system for software development.*

What is version control?

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

What is Local Version Control?

> *Stores all changes to files on your computer, making it difficult or impossible to recover them if any errors occur or files get lost.*

What is Centralized Version Control?

ℹ️ There is a single central server that contains all the versioned files, and it only keeps track them.

ℹ️ To make changes to the code, they first need to obtain a copy of the latest version from the central repository. They make their changes on their local machine and then submit them back to the central repository.

- ℹ It doesn't keep track on the versions on your devices or on the others making it very hard to merge and maintain.
- ℹ If the central repository is lost or damaged, it can be difficult or impossible to recover the code.

What is Distributed Version Control ?

A hybrid system between the previous systems. It allows you to have a copy of all the changes that have occurred throughout the project's lifetime and the codes on your computer. If anything happens to the original code, you can send your code and the history of changes that have occurred since the project started, as if nothing had happened.

Repository:

> *It's the place or folder where Git stores the code and any changes that occur in it.*

Working Tree:

> *It's the current state of the directories and files on your local machine. You can make changes to any file in it.*

What are the states of a file in git?

- Modified(untracked): You have changed the file but have not committed it to your database yet.
- Staged: You have marked a modified file in its current version to go into your next commit snapshot.
- Committed(Tracked): Here, you've told Git to take these edits and make them in the repository. Git will record that at a specific time, someone edited the file, with a message explaining why they made this commit.

.2. Hi 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.*

`< --local>`
By default, git config will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository git config gets invoked in.

`< --global>`
Global level configuration is user-specific, meaning it is applied to an operating system user.

`< --system>`
System-level configuration is applied across an entire machine. This covers all users on an operating system and all repos.

Listing configuration options

```
$ git config --list
# or
$ git config -l
```

Configuration structure

```
<section>.<key>
```

Get /set value

```
git config --global user.name
git config --global user.email
git config --global user.name "YourName"
```

Delete a configuration

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

NOTE
The first time using installing git you should set your config `user.name` `user.email`

Creating a new Repository

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 sub-directory in the current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes sub-directories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit.

```
$ mkdir myRepo
$ cd myRepo
$ 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.

```
$ echo “BATMAN”>>REDME.md
$ git add REDME.md
$ git commit -m “meassage”
```

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. For this, you need to use git log.

```
$ git status
```

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
```

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.
`--decorate` adds the names of branches or tags of the
commits that are shown.
`--oneline` shows the commit information on a single line making it easier to browse through commits at-a-glance.

```
$ git log --graph --oneline
```

.3. *Git Internals*

what does git do

what does a file system consists of?

A file system begins with a root directory (in UNIX-based systems, `/`), which usually contains other directories (for example, `/usr` or `/bin`). These directories contain other directories, and/or files (for example, `/usr/1.txt`).

---

what does git do

Git maintains snapshots of a file system which is your directory

git building blocks

- ★ Blobs: They are contents of your files. The difference between files and blobs is that blobs doesn't contain metadata.
- ★ Trees: It's the equivalent to a directory. It contains listings of blobs and other trees
- ★ Commits: commits are just snapshots of this file system. A commit object contains a pointer to the main tree (the root directory) and met-data

These objects are all identified by a SHA-1 hash

> NOTE
> Every commit holds the entire snapshot, not just diffs from the previous commit(s).

branches:

A branch is just a named reference to a commit. (JUST A SHA)

---

commands

- `git log`: Displays a history of commits, showing information like commit message, author, and date.
- `git cat-file -p` : This command allows you to view the raw content of specific Git objects (blobs, trees, commits) based on their SHA-1 hash.

.4. Branches, merge & rebase

---

branches

> What is a branch and why ?
> Branching means you diverge from the main line of development and continue to do work without messing with that main line.

create a branch :

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

> ⚠ Attention :
> After you make a new branch, git doesn't checkout(move to) the new branch automatically.

switch to a branch :

```
git checkout <targetbranch>
```

You can use the `-b` option with checkout to create a branch and switch to it.

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

Info :
Branches are virtually `free` just a SHA to a commit/tree.

---

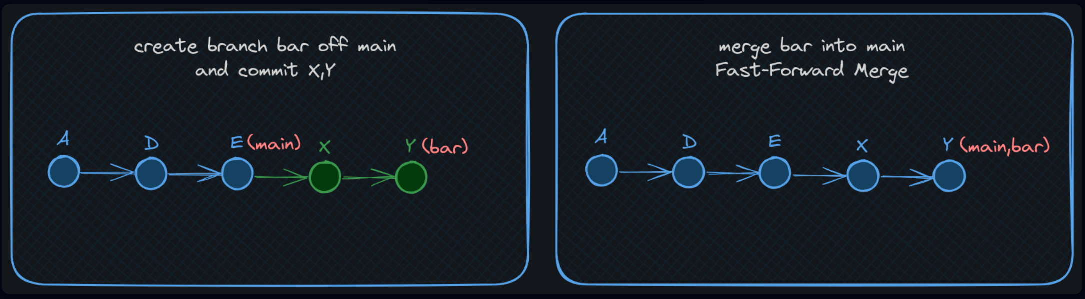
merge

what is a 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`

merge have 2 different outcomes :

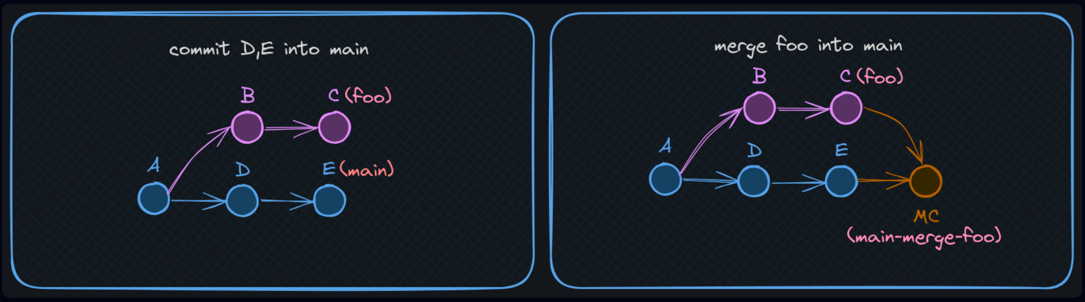
1. *Fast Forward:*

just update the pointer/reference `(no merge commits)`


create branch bar off main and commit X,Y
A
D
E (main)
X
Y (bar)
merge bar into main Fast-Forward Merge
A
D
E
X
Y (main,bar)

2. Divergence merge:

create a merge commit to combine 2 commits/histories have 2 parents



how to merge :

```
git merge <source-branch>
```

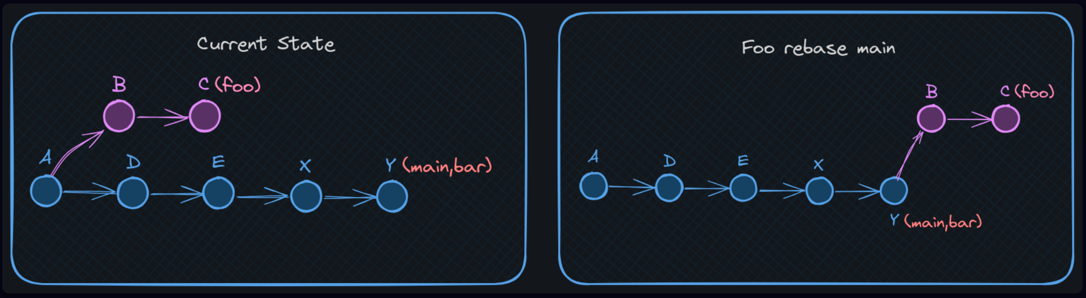
target-branch is the currently checked-out branch
conflicts are discussed on the `going remote` section.

---

rebase

what is a rebase ?

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



how to rebase :

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

NOTE :
Note the different perspective of merge's one.
rebase alters your branch to be at the tip of `<targetbranch>`

How rebase actually works :

1. checkout the latest commit at `<target-branch>`
   - think of `<target-branch>` as `main`
2. replay one commit at a time of the `<source-branch>`
   - `<srouce-branch>` is often the feature branch.
3. update source branch ref to the latest commit made.

Be careful :
Rebase alters history (notice: replay), so **don't ever rebase main or any other public shared branch**

---

merge vs. rebase

- Merge:
  - doesn't alter history
  - doesn't require `push force`
  - works with private and public branches without problems
  - makes annoying merge commits
- Rebase:
  - alters history
  - requires `push force`
  - works best with private branches only

- no annoying merge commits
- linear history which is easier to search

workflows war

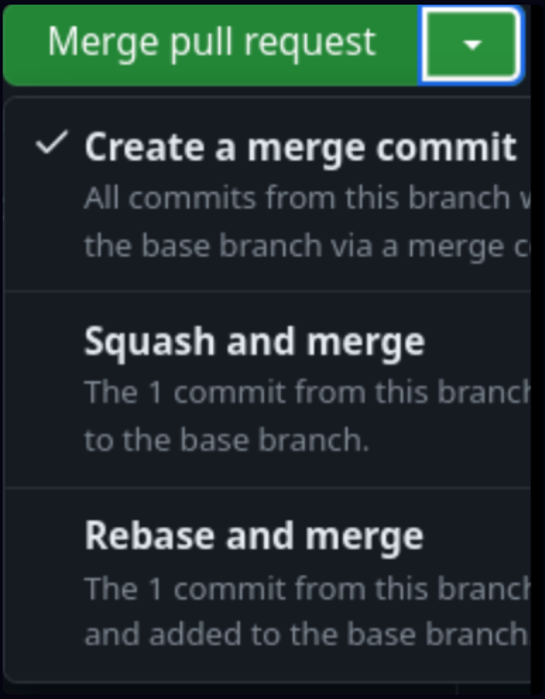
Merge flow :

- just merge always
  - merge back into main
  - ⚠ merge commits happen but we live with it

Rebase flow :

- (rebase in private branches - FF merge in public branches)
  - rebase main into your feature branch first
  - then fast-forward merge into main

Important :
In simple cases the difference between the 2 options in practice is what option you use in the following list.
but people seem to be really opinionated about this matter.
anyway `rebase flow` is the best.


Merge pull request
✓ Create a merge commit
All commits from this branch w
the base branch via a merge c
Squash and merge
The 1 commit from this branch
to the base branch.
Rebase and merge
The 1 commit from this branch
and added to the base branch

.5. *Head and reflog*

Read Carefully:

In git terminology, HEAD is a reference variable. It points to the tip of the current branch, i.e. the last commit on the current branch. It's stored inside .git/HEAD.

Git stores a full record of HEAD's movements through different commits and branches, which you can view by running *git reflog*.

This record does not get affected if a branch gets deleted. This means you can use it to recover commits that were on a deleted branch. Find the SHA of the commit you want to recover, and run *git merge* .

But, as you may know, each commit stores the entire content of the repo, and when merging a particular commit, you may not want its entire history, as there may be changes you want to discard(that were part of previous commits).

You can avoid this using *git cherry-pick*. This command allows you to merge only the changes that were new and did not exist in previous commits, thereby ignoring its previous history.

.6. Git remote

What's a remote?

remote is a COPY OF THE MAIN REPO SOMEWHERE ELSE.
That means that you can have a remote repo on your local machine.

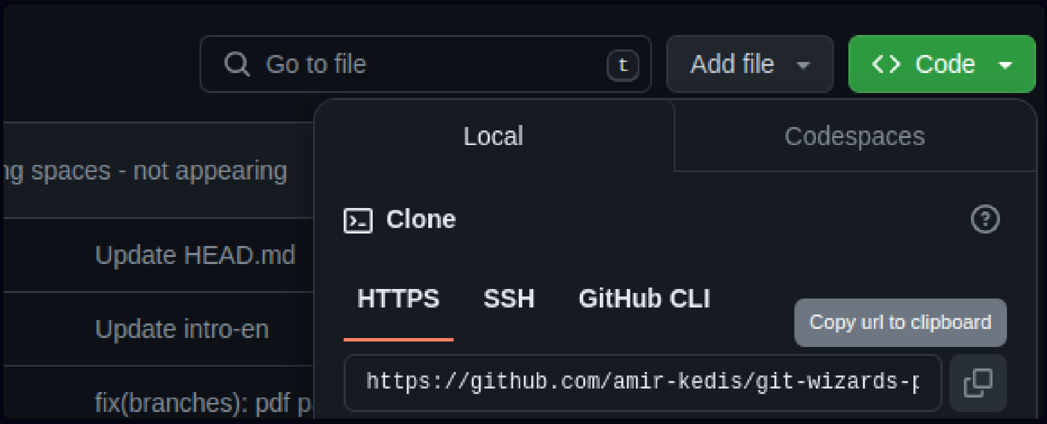
git vs github

github is used with git for online backup.

To link your local repo with a remote one:

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

or simply



```
https://github.com/amir-kedis/git-wizards-p
```

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

this will create a folder with git initialized in it

git fetch

```
git fetch
```

Here's what happens when you run `git fetch` :

1. Git contacts the remote repository.
2. It retrieves any new commits, branches, or tags that exist on the remote repository but are not present in your local repository.
3. It updates your local repository's remote-tracking branches (e.g., `origin/main`) to reflect the state of the remote repository.

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 of the remote
> tracking branches as they are not updated automatically

- You can then merge these changes using `git merge`

git pull

To merge the remote changes into your local branches:

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

💡

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

git push

It's now the case that you want to upload your changes to the remote repo.

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

merge conflicts

merge conficts happen when making different changes to the same line of the same file, or when one person edits a file and another person deletes the same file *(git can't automatically merge)*.

This is how files with conflicts look like:

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

```
Your version has these changes:
<<<<<<< HEAD
Hello from local
=======
```

```
and the remote has these changes
=======
Hello from remote.
>>>>>>>3481fb2d429347f009646b1456375e6685c1a36e
```

To resolve a merge conflict you either:

- accept your changes
- accept the remote changes
- combine both
- add completely new line

.7,8. *Fancy Stuff*

[I] Editing

1. Amend

> *Goal: edit the last commit (don't create a new one)* 🛠️

```
# first make your edits, then stage them

# Then: to edit the last commit
# without changing its message
git commit --amend --no-edit

# to edit the message use
git commit --amend -m "your updated msg"
```

2. Interactive rebase

> *Goal: rewrite history ✍🏼 🚧 (not just the last commit)*

```
# to enter this mode:
git rebase -i HEAD~3

# to edit the last 3 commits (3 is just an example)
```

Then you will be prompted with an interface where you can reorder, delete, edit, squash, and more

2.1 Reword

> *Goal: Edit the commit message* ✍🏼

- Use the keyword `reword` before the SHA
- save and continue, then you will be prompted again to change the message of desired commit

2.2 Reorder

> *Goal: change order of past commits* 📝

- Use the keyword `pick` before the SHA
- NOTE: The shown commits are ordered top to bottom from oldest to newest, just change the order of the given lines as you wish

2.3 Squash

> *Goal: Combine multiple commits into one* 📚

- Use the keyword `squash` or `fixup` before the SHA
- `fixup` by default keeps the older commit's message, while `squash` re-prompts for a message.

[II] Deleting

1. Stash

> *Goal: save my current changes away from the wokring tree (files)*

```
# make some shanges, then
git stash -m "quick draft"
# message is optional
```

> NOTE:
> Stashed changes are removed form the working tree

- `git stash pop` to pull the changes back to the current working tree (files)
- `git stash drop` to delete one entry
- `git stash clear` to clear the whole stash history

2. Revert

> *Goal: Undo the last commit.* ↩

`git revert HEAD`

> NOTE:
> automatically creates another commit.

3. Reset

> *Goal: Delete commits, restore previous version* ⏪

- `git reset --hard HEAD~1` deletes the last commit and and its changes from the current working tree

- `git reset --soft HEAD~1` deletes the last commit, but leaves the working tree as it is.
- The Working Tree in Git is a directory (and its files and subdirectories) on your file system that is associated with a repository.

[III] Tags

> *Goal: Mark a point in history* 🚩

Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1. 0.1). A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.

- To list all tags `git tag`
- To add a tag `git tag -a v1.2`

[IV] Releases

> *Goal: share a downloadable version of my work*

A Release is created from an existing tag and exposes release notes and links to download the software or source code from GitHub.

> NOTE:
> GitHub Releases must be created from existing tags
> GitHub Releases are based on tags

[V] Worktrees

> *Goal: Being able to checkout multiple branches at the same time. each in a different folder.*

Instead of using multiple clones, or unintended commits / stashes before checking out to another branch.

[VI] Workflows and GitHub Actions

> *Goal: Automate building, testing, deployment*

configure automatic actions that are executed when something happens to your repo something like a new commit / push or a new pull request

---

Thank you for getting to here