Git and GitHub

What is Git ?

Git is a popular version control system that manages and track changes to files over time, allowing multiple people to collaborate on a project while maintaining a history of all modifications.

It is used for:

1. Tracking changes to file, such as:
    1. code
    2. documents
    3. configuration files
2. Tracking who made changes
3. Coding collaboration
4. Stores the file in a repository

Overall, version control ensures that changes are organized, recoverable and easily managed, making it a critical tool in software development

Why use Git ?

1. It allows multiple developers to work on the same codebase simultaneously without overwriting each other's work, providing a clear record of who made changes and why.
2. Facilitate rollback to previous versions if issues arise
3. Support branching and merging, which are crucial for experimenting with new features and managing different stages of development
4. Ensures code quality, accountability and efficient collaboration in projects

Installing Git locally:

To use Git on your local machine, you need to install it first. However, on most Linux distributions and macOS, Git comes preinstalled.

You can verify the Git Versions by running:

```
git --version
```

If Git is installed, you will see output similar to:

```
git version 2.42.0
```

if it is not installed, you can install it using your system's package manager:

For macOS:

```
brew install git
```

For Ubuntu/Debian-based Linux:

```
sudo apt-get update
sudo apt-get install git
```

For windows:

Git does not come preinstalled on Windows. You can download the binary from the official Git release page and follow the installation instructions.

What is a Repository ?

A repository is a storage location for your project's code, documentation and other files. It serves as a central hub for collaboration, version control and code management

git init:

The git init commands 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 will run in a new project.

All you have to do is cd into your project subdirectory and in your terminal run:

```
git init
```

Typical Output:

```
Initialized empty Git repository in /home/user/project/.git/
```

Git creates a hidden folder called .git, which stores all the information needed to track your project's history

git config:

The git command is used to set up your Git configuration-things like your name, email and preferences. It tell Git who you are and how it should behave on your computer.

Example (basic setup):

```
git config --global user.name "Your name"
git config --global user.email "your.email@example.com"
```

This set your name and email for all your git projects. Git will use this information when you make commits

Levels of configuration:

1. --local: only for the current project (saved in .git/config)
2. --global: for all your Git projects (saved in ~/.gitconfig)
3. --system: for all users on the system

Most of the time, you will use –global

Understanding the Git Workflow:

When working with Git, your files move through 3 main areas before your changes are permanently saved. These areas help you control what gets recorded in your project's history.

1. Working directory:

- your project folder – the place where you actually work on your files
- can create,edit or delete files here as you normally would
- At this stage Git simply notices that something has changes, but does not record it yet

Example: You edit a file called index.html in your project folder

2. Staging Area (Index):

- Before saving changes to the repository you first stage them using 'git add' command
- This tells Git which changes you want to include in your next snapshot

Examples:

a. Add a single file

```
git add index.html
```

b. Add multiple files

```
git add file1.txt file2.txt file3.txt
```

c. Add all changes in the folder.

```
git add .
```

Think of the staging area as a shopping cart-you review and choose what to include in your next commit.

3. Commiting changes

Once your changes are staged, you save them permanently in Git's history by commiting:

```
git commit -m "Add a homepage content"
```

A commit is like taking a snapshot of your project at that exact moment in time. Git stores that snapshot inside the hidden .git folder folder so you can always go back, compare or undo changes later

Git ignore:

Git sees every file in your working directory as one of 3 things:

1. Tracked – a file which has been previously staged or commited
2. Untracked – a file which has not been staged or commited
3. ignored – a file which git has been explicitly told to ignore

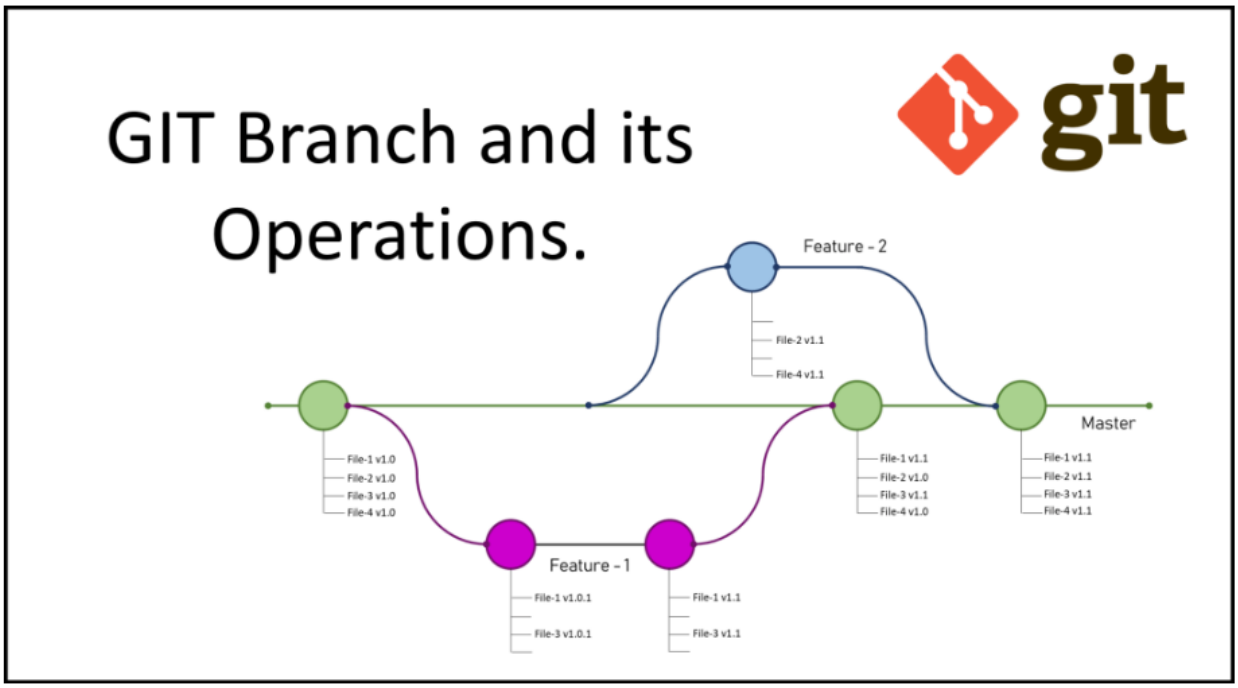
Ignored files are usually build artifact and machine generate files that can be derived from your repository source or should otherwise not be commited. For example:

- dependecy caches, such as the contents of /node_modules or /packages
- compile code such as .o , .pyc and .class files
- build output directorise such as /bin, /out or /target
- files generated at runtime such as .log, .lock, or .tmp
- hidden system files such as .DS_Store or Thumbs.db
- personal IDE config files, such as .idea/workspace.xml
- environment variable such as .env

Ignored files are tracked in a special file name .gitignore that is checked at the root of your repository. There is no explicit git ignore command: instead the .gitignore file must be edited and commited by hand when you have new files that you wish to .ignore.

Branching Basics:

A branch in Git is like a separate workspace or timeline for your projects. Creating a branch in git, allows you to work on different features or fixes without affecting the main codebase.



1. Viewing Branches:

To see all the branches in your repository:

```
git branch
```

Output Example:

```
* main
 feature-branch
```

- * indicates the current branch
- other branches (if any) would be listed without *

2. Creating a new branch

```
git branch feature-branch
```

This command does not show output to verify run the first command: git branch

3. Creating and switching to a branch

```
git checkout -b feature-branch
```

Output example:

```
Switched to a new branch ‘feature-branch’
```

4. Switching between branches

```
git checkout main
```

Output example:

```
Switched to branch ‘main’
```

5. Renaming a Branch

To rename the current branch

```
git branch -m new-name
```

To rename a different branch

```
git branch -m old-name new-name
```

Output example:

```
Renamed branch ‘feature-brach’ to ‘new-feature’
```

6. Deleting a branch:

```
git branch -d feature-branch
```

Output Example:

```
Deleted branch feature-branch (was d4e5f6g).
```

Merging Basics:

A merge in Git is the process of combining changes from one branch into another. When you want to integrate updates from one branch (the source) into another branch (the target) you need to perform a merge, This involves resolving conflicts between the 2 branches, if any exists. The goal of merging is to create a new commit that represents the combined changes from both branches, resulting in a single history for your project

Basic Merge Workflow:

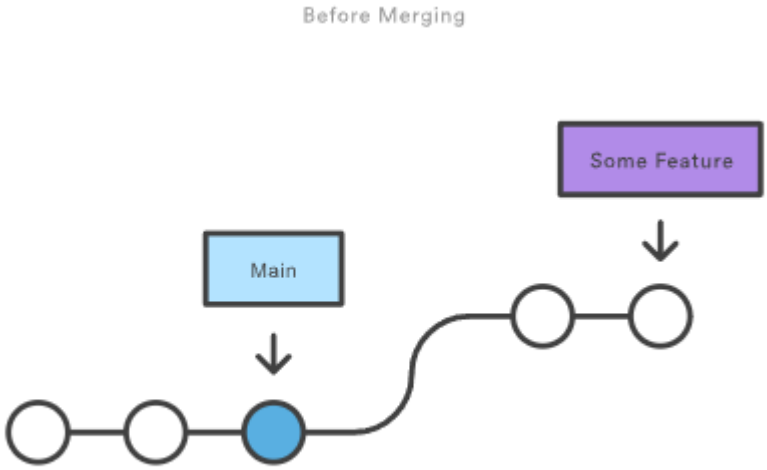
Suppose you are on main an want to merge feature-branch:

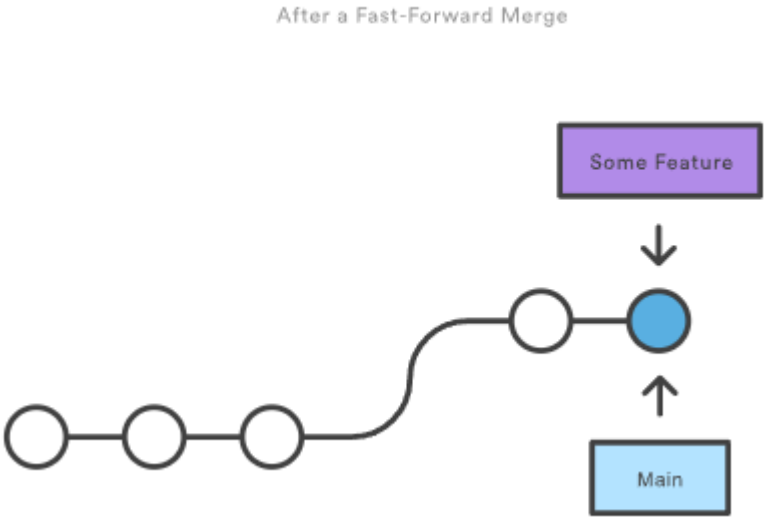
```
git checkout main
git merge feature-branch
```

Types of Merges:

a. Fast-Forward Merge

- Happens when the branch you are merging directly follows the branch you are on
- Git just moves the pointer forward, no new commit is created

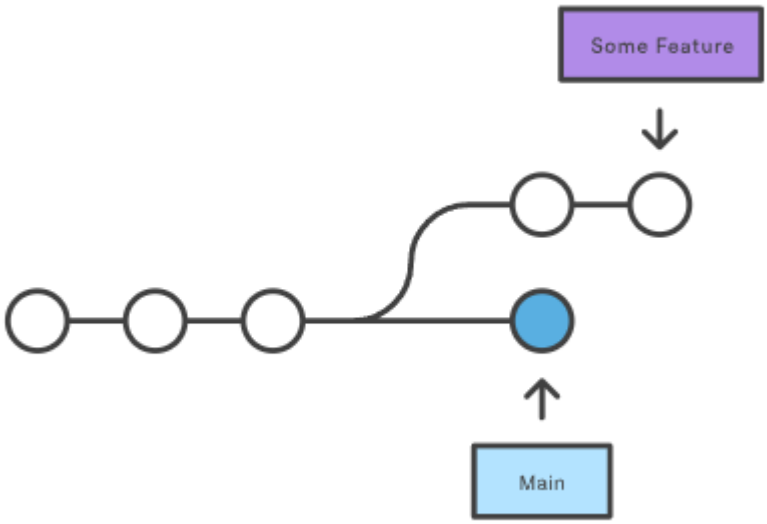




Output Example:

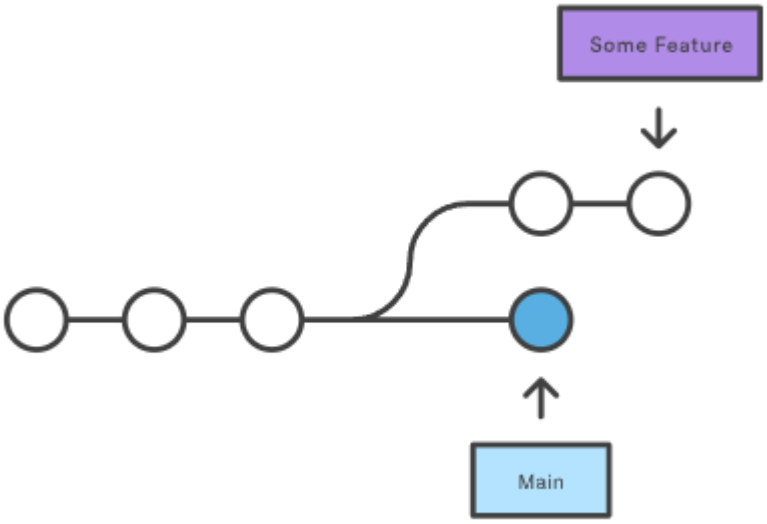
```
Updating a1b2c3d..d4e5f6g
Fast-forward
 file.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

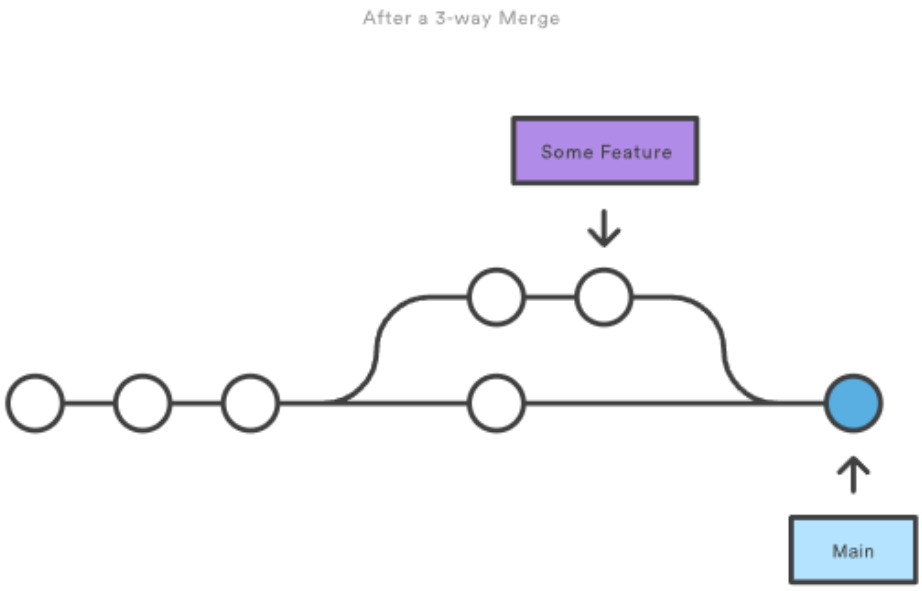
However a fast-forward mege is not possible if the branches have diverged like in this use case:



b. 3-Way Merge

- Happens when both branches have independent commits since they diverged
- Git creates a new merge commit that combines the changes from both branches





Output Example:

```
Merge made by the 'recursive' strategy.
 file.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

c. Merge Conflicts

Sometimes Git can not automatically combine changes. This is a called merge conflict

- Git marks conflicts in the file with special markers (<<<<<< , ======,>>>>>>)
- You need to edit the file manually to resolve the conflict
- After resolving, mark it as resolved and commit

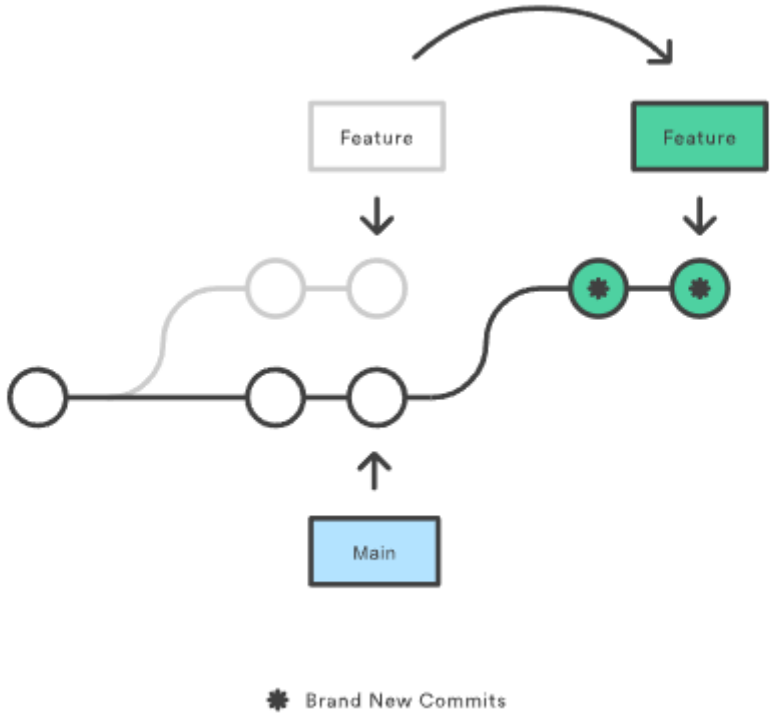
```
git add conflicted-file.txt
git commit
```

d. Tips:

- Use “git status” to see which files are staged or have conflicts
- Use ‘git log –graph –online –all’ to visualize the branches and merges
- Always merge into the branch you want to update, not the branch you are merging from

Git Rebase:

Rebasing is a Git command that moves or reapplies your commit on top of another branch. It is a way to keep a clean linear commit history instead of having a branch with multiple merge commits



From a content perspective, rebase is changing the base of your branch from one commit to another making it appear as if you would created your branch from a different commit

Why do we want to maintain a “clean history” ?

1. A bug is identified in the main branch. A feature that was working succesfully is now brokne
2. A developer examines the history of the main branch using `git log` and because of the ‘clean history’ the developer can now identify when the bug was introduced

What is GitHub Essentials ?

GitHub is a cloud based platform for hosting Git repositories, collaborating on projects and sharing code. It builds on Gits but add features like:

1. Repositories for storing and managing code
2. Branches for parallel development
3. Pull requests for code review and merging
4. Issues for tracking tasks and bugs
5. Collaborative tools like project boards and wikis

Understanding and mastering these fundamentals components allows developers to effectively manage their projects, collaborate with team members and contribute to open-source initiatives

Creating Account:

To get started with GitHub, you will need to create a free personal account on GitHub.com and verify your email address. Every person who uses GitHub sign in to a personal account. Your personal account is your identity on GitHub.com and has a username and profile

Adding a new SSH key to your GitHub account:

You can access and write data in repositories on GitHub using SSH (Secure Shell protocol). When you connect via SSH, you authenticate using a private key file on your local machine

Generating a new SSH Key:

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

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

3. At the prompt type a secure passphrase or empty for no passphrase

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

Adding your SSH key to the ssh-agent:

1. Start the ssh-agent in the background

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

2. Add your SSH private key to the ssh-agent

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

Adding a new SSH key to your account:

1. Copy the SSH public key to your clipboard

```
$ cat ~/.ssh/id_ed25519.pub
# Then select and copy the contents of the id_ed25519.pub file
# displayed in the terminal to your clipboard
```

2. In the upper-right corner of any page on GitHub, click your profile picture, then click Settings.
3. In the "Access" section of the sidebar, click SSH and GPG keys.
4. Click New SSH key or Add SSH key.
5. In the "Title" field, add a descriptive label for the new key. For example, if you're using a personal laptop, you might call this key "Personal laptop".
6. Select the type of key, either authentication or signing.
7. In the "Key" field, paste your public key.
8. Click Add SSH key.
9. If prompted, confirm access to your account on GitHub.

Cloning a Repository and Understanding Git remotes:

A remote in Git is a version of your repository that is hosted on another server, usually on platform like GitHub. Remotes allow you to synchronize your local repository with a central copy

1. Cloning a Repository:

What happens when you clone:

- Git creates a local copy of the repository
- The default branch (usually main or master) is checkout locally
- Git automatically sets the upstream branch of your local main to trach the remote origin/main

```
git clone https://github.com/username/repo.git
```

Output Examples:

```
Cloning into 'repo'...
remote: Enumerating objects: 42, done.
remote: Counting objects: 100% (42/42), done.
remote: Compressing objects: 100% (35/35), done.
Receiving objects: 100% (42/42), 5.23 KiB | 5.23 MiB/s, done.
Resolving deltas: 100% (10/10), done.
```

After cloning

```
git status
```

Output Example:

```
On branch main
Your branch is up to date with 'origin/main'.
```

- The local main branch is tracking the remote origin/main
-

This means you can run

```
git pull
git push
```

without specifying the remote branch, because the upstream is already set

2. Checking remotes:

After cloning, you can see the remote(s) linked to your repository

```
git remote -v
```

Output Example:

```
origin  https://github.com/username/repo.git (fetch)
origin  https://github.com/username/repo.git (push)
```

- origin is the default remote name
- fetsh - used to download changes from the remote
- push – used to upload changes to the remote

3. Working with Remotes

Fetching changes – Download commits from the remote without merging:

```
git fetch origin
```

Pulling Changes – Download and merge remotes changes into your current branch

```
git pull origin main
```

Output Example:

```
Updating a1b2c3d..d4e5f6g
Fast-forward
 file.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Pushing Changes – Upload your local commits to the remote branch

```
git push origin main
```

Output Example:

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 500 bytes | 500.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/username/repo.git
   a1b2c3d..d4e5f6g  main -> main
```

4. Adding a new Remote

If you want to link another remote repository:

```
git remote add upstream https://github.com/otheruser/repo.git
```

- upstream is a custom name for the new remote
- useful when working with forks

5. Creating a new Branch and pushing to remote:

After creating a local branch, you can push it to the remote and set it to track the remote branch at the same time:

```
git checkout -b feature-branch
git push -u origin feature-branch
```

Output Example:

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 500 bytes | 500.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/username/repo.git
 * [new branch]      feature-branch -> feature-branch
Branch 'feature-branch' set up to track remote branch 'feature-branch' from
'origin'.
```

- -u or (--set-upstream) sets the remote branch as upstream so future git push or git pull commands works without specifying the branch or remote again. You only need to switch the branch in your local repository

6. Pushing Changes to remote

a. Move changes to the staging area using git add command:

```
git add file.txt
```

b. Save your changes in a commit:

```
git commit -m “Add feature x”
```

c. Pushing changes to remote:

```
git push
```

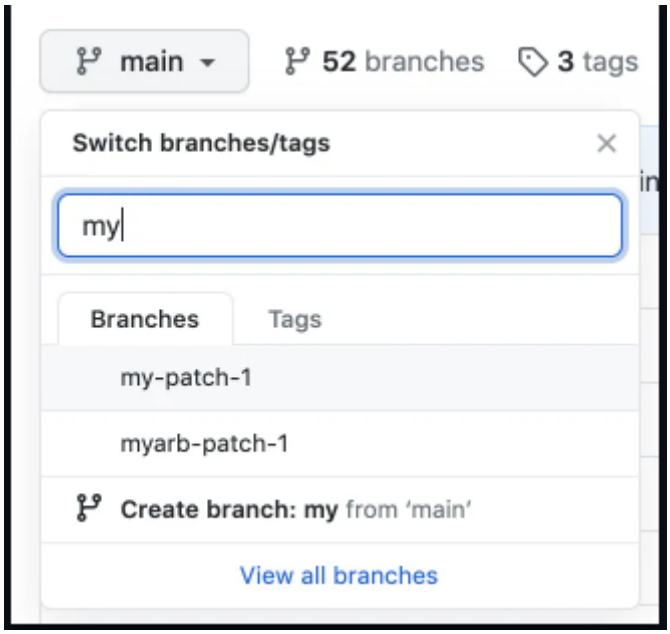
Collaboration on GitHub (Prs):

A Pull Request (PR) is the main way to propose changes to a project. It allows collaborator to:

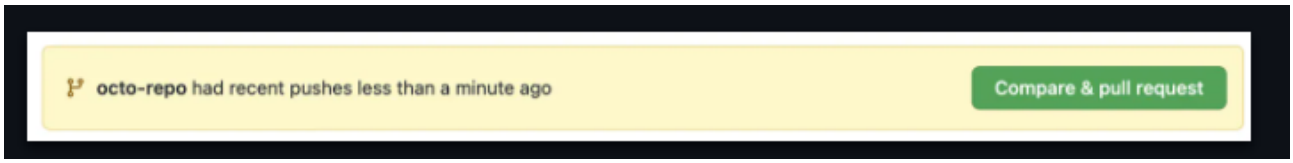
1. Review your changes
2. Discuss improvements or issues
3. Suggest edits before merging

Creating the pull request:

1. On GitHub, navigate to the main page of the repository
2. In the “Branch” menu choose the branch that contains your commits.



3. Above the list of files, in the yellow banner, click Compare & pull request to create a pull request for the associated branch.



4. Use the base branch dropdown menu to select the branch you would like to merge your changes into, then use the compare branch drop-down menu to choose the topic branch you made your changes in
5. Type a title and description for your PR
6. To create a PR that is ready for review, click Create Pull Requests

Updating your PR:

If you need to make more changes after opening a PR:

```
git add .
git commit -m "Update feature based on review"
git push
```

GitHub automatically updates the PR with your new commits