

# Learning Git and GitHub By Saifullah Khan

## Setting Up Git

To set up Git, you need to download Git from web. You can directly download from their website <https://git-scm.com/downloads>, Otherwise following Git commands will not work.

### 1. Command:

```
git --version
```

This command can be used to check the git version downloaded on the system

Now, how we make it our own? We should have an account with our email with which we push or commit.

Note: we can check for some commands by `git config`. Use Command Prompt to run these commands.

### 2. Command:

```
git config --global --list
```

This will list all the existing variables we have. `--global` for global values and `--list` for values. If git is already configured it will give name and email, otherwise we will have to configure it.

To configure we use following commands:

### 3. Command:

```
git config --global user.name "(input username)" git config --global user.email "(input email address)"
```

Using `git config --global --list` will give you the user and email.

## ✓ Creating a Project

## Git Implementation on VS Code

Now, when we create a project folder, Git has no idea what we are doing. It does nothing meaning no tracking or no initializing. So what we have to do is to tell Git to start tracking our versions. There are two concepts "**Working Directory**" and "**Staging Area**", Working Directory contains all the files of our project while Staging Area is where we place our files that we want Git to track. To understand this, suppose you have 10 files in your project (working directory) and you want to track 5 files, here staging area comes in, so git tracks the files in staging area and ignores the remaining files. We just have to push the files to the staging area. All this is done on a **local repository** which is in your local (your own) machine. To share it with friends and colleagues, we create a **remote repository**.

### 4. Command:

```
git status
```

This returns the status of the repository.

## Local Repository:

First we need to create a local repository which is on the machine. To create one, we use the following command:

### 5. Command

```
git init
```

This will initialize a local Git repository in your project folder. Now we can use the the status command to view the status of our repository.

Note: Also, if not set during the installation, the branch name you get will be "Master". To get "main" branch, we can set the name while initializing the repository by using the following command: `git init -b main`

Here, we are specifying the branch name we want, which is main in this case.

## Tracking files

We can create as much files as we want but Git will not track any of it until we tell it to. So, in order to tell Git, we need to add those files to the staging area, which is in the local repository which is created by initializing the repository.

When we create a new file, it is already in the working directory. If we use `git status` at this point, it will show the files untracked and our commits. Adding files to staging area is simply easy by using the command:

### 6. Command:

```
git add (filename with extension)
```

With this command the file will be in staging area and Git will be able to track that file. Also we will be able to commit (save) any changes we make in our file. There is also a command to check the commit history which is:

### 7. Command:

```
git log
```

By this we will be able to see all the commits. Now to commit changes we use the command:

```
git commit -m "(commit message, must be logical)"
```

Quote: "*Commit early, Commit Often*" 😊

There is also a concept of `checksum`. Git also provides us Integrity meaning once you save something in a Git commit, we cannot change it without Git knowing about it. HOW? By checksum. Now imagine checksum as fingerprint for the data so whenever you change something in the data, it will change the checksum as well and Git creates a checksum for every commit. It is done with the help of `sha1` and it creates 40 number of characters and out of which we only focus on the first seven hexadecimal numbers. It has a unique checksum number for every commit ensuring Integrity.

In this way we have done our first commit. Congrats 🎉

## Making Changes and Skipping Staging Area

Now lets make some changes to the file. After making the required changes or adding something like a feature or resolving a bug, we can follow some steps to commit those changes as well.

First of all we need to understand that when we make changes, it is applied to file in the working directory, so if we try to commit changes directly it wont allow us as the modified version of the file will not be present in the staging area and we should not forget that the file is only committed when the modified version is added to the staging area and making any change will not automatically add it to the staging area. So the steps would be to make changes, add the file to staging area (`git add filename.ext`) and commit the file (`git commit -m commit message`). It means we have to add it to staging area everytime we make changes and want to commit that file. But this is not the only method. We can also just skip the adding the file to staging area step and commit directly.

To simply skip the staging part, we can modify our commit command:

## 8. Command:

```
git commit -a -m "Commit message"
```

By this command, we will be able to commit without going to the staging area. By "-a", we can understand that it means to skip the staging area or skip the adding file step.

You can also use `git add .` to add all the files present in the working directory, the ones that are modified and also the new created files

## Git diff Command

It gives us the option of finding the differences between what you have worked and what is already there in the commits. It simply means that when a user makes a change, it can be at multiple places throughout the lines of text or code and want to see what exactly has been modified, then the user can use the following command:

## 9. Command:

```
git diff
```

It shows the lines and modifications that have been made before the file has been added or committed. We can also use `git status` but it only tells about the file being untracked. Once we have added the file to the staging area, we can not see the changes, not by just using the `diff` command. We would have to modify it so that the changes are visible to us. If file has already added, we can use the following command:

```
git diff --staged
```

This command will neglect that the file has already been in the staging area and will show the changes. To terminate the `log` command, press 'q' to exit the log view and return to the terminal prompt.

## How to Remove a File From Git Repository

Suppose a scenario where you have created a file and added it to the repository and then you realize that it wasnt the file you wanted to add. For this situation you can use the following command:

## 10. Command:

```
git rm --cached 'filename.ext'
```

`rm` is for remove and `--cached` as the file data is cached and the file name with extension.

This will remove the file from the local repository and then if you want you can modify or delete the file. This is how you can save yourself from unexpected file uploads.

## GitHub (Remote Repository)

To this point, all the commits are saved on your machine. If you are working in a team and want to collaborate with them you have to share it with them so that you all can work on the same thing or different features at the same time. This is where we use a Remote Repository.

Lets suppose there are six people in a team and they have local repositories in their machine and now if they want to share, they will need a remote repository which could be private for a company or public for anyone to access. For remote repositories, we have multiple options most known is GitHub, GitLab and Bitbucket.

Now the question is how are we going to connect our local repository and our remote repository? For this, follow the following steps:

### Creating a new repository from scratch or importing an existing one

If starting from scratch, you can first create a remote repository on GitHub, then create a local repository on your machine either manually or through your terminal. You can also follow the steps for ease you get after creating a repository on GitHub.

#### Quick setup — if you've done this kind of thing before



Set up in Desktop

or

HTTPS

SSH

`git@github.com:Saifullahkhan31/test.git`

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

#### ...or create a new repository on the command line

```
echo "# test" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:Saifullahkhan31/test.git
git push -u origin main
```

#### ...or push an existing repository from the command line

```
git remote add origin git@github.com:Saifullahkhan31/test.git
git branch -M main
git push -u origin main
```

**First** create a folder, add a readme file and `.git` file by initializing (init command). Add the readme file to staging area and commit.

**Secondly** create a ssh key to link you local repo to your remote repo by this command:

## 11. Command CREATING AN SSH KEY:

`ssh-keygen -o`

It will create an ssh key which you will paste in your GitHub profile. Go to Settings -> then from menu bar on the left, Select SSH and GPG Keys -> Select new SSH Key -> Add the title and paste the key -> Click on add key. Your key will be added.

Now that the above steps are done, add the remote access to the origin ssh link:

```
git remote add origin git@github.com:Saifullahkhan31/(repo name)
```

This will make a link between the local and remote repository. Then:

```
git push -u origin main
```

This will push all the commits you made to the remote repository. '-u' flag is for the upstream, 'origin' is what we just established and 'main' is for the branch.

## Some cmd commands (for windows)

To go to the root directory: `cd\.`

To go back a directory: `cd..`

To change directory: `cd (file name)`

To create a File: `mkdir (file name)` e.g: `mkdir testfile`

To see folder files: `dir`

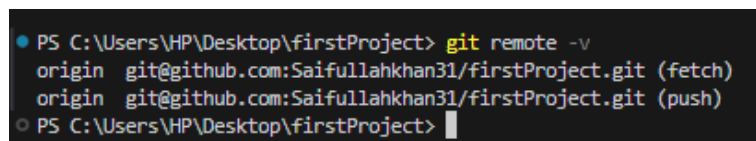
To see hidden files: `dir /a:h`

To create a readme.md file with a message: `echo "# readme file content" >> readme.md`

To see a file contents: `type file.txt` OR `more file.txt`

Now that everything is setup, you can easily create more projects make commits and push them directly to the remote repo. Even if you make more projects, you will simply initialize and commit as usual but before pushing you will need to establish the remote connection with this command `git remote add origin git@github.com:Saifullahkhan31/(repo name)` only one time and then you can push your commits as want.

Here we dont have to write 'origin' always, it depends on the repository we are working on meaning it can be different. In our case, we initially have only one remote server.



```
PS C:\Users\HP\Desktop\firstProject> git remote -v
origin  git@github.com:Saifullahkhan31/firstProject.git (fetch)
origin  git@github.com:Saifullahkhan31/firstProject.git (push)
PS C:\Users\HP\Desktop\firstProject>
```

In the above picture, we can see that we have only one origin (link) for both fetch and push because we have one repo in remote. We can rename it and also connect to other repositories and we can push and pull to any other repos if we want.

## Git Tags

Every software has a version number such as version 1, 3, or 12.02.1 something. This is the concept of versions is known as tags.

Imagine that you have made several commits and now you think that you are set to release you next version of the application or software, so you can give a tag to it.

There are two types of tagging:

1. Annotated Tagging - Is used when you want to give more information to the user like who is committing it or who owns it.

## Lightweight Tagging

First if all, if we want to check all the tags for a current project we use:

## 12. Command:

```
git tag
```

If it is empty, it means there are no tags. So let's give our first tag:

### 13. Command:

```
git tag -a v1.0 -m "1st release"
```

'-a' means annotated tag, 'v1.0' is the tag, it can also be just 1.0, and then '-m' is for the message. Your tag is done.

In order to check your tag, use `git tag` command to see your tag. To see more details or description of your tag, use this command: `git show`, this will show the tag, the tagger, time/date, tag message, commits.

Now, the thing is that the tag is only available on the local machine even when we push our commits. We also have to push our tag to the remote repo. So to push the tag use command:

## 13. Comamnd:

```
git push origin (tag)
```

This will push the tag to the remote repo with the changes made at the time of creating that tag.

## Cloning a Project

Uptil now we have learned about making our own repositories and pushing them to GitHub but what about when you see someone's repo or an open source project and you want to see the source code, use it for yourself or to contribute to it, then it is not intuitive to go on GitHub and navigate through each file. For this GitHub provides a solution of cloning the repository and download it directly into our machine.

## 14. Command:

```
git clone (https or ssh link of repo)
```

After running this command in your terminal, it will start downloading the repo with all its files to your machine. Then you would be able to open all the files in your IDE.

You will be able to see all the commits by `git log` command, with `git tag` you will be able to see all the tags and with `git show (tag no.)` you will be able to see details about a single tag. Also, you can also use `git log --pretty=oneline` command to see all the commits in one line which will show you the commit numbers and commit messages from which you will be able to see how other people write commit messages.

More on contributing:

Suppose you have contributed a feature or made any changes in the repo, how would you save those changes to the source remote repository? For this, you will have to add it staging area, then commit and then push to the origin branch. Also there is a possibility that the file you made changes to isn't added because of incorrect file path. To resolve this, you will need to type the path of the file which you can see with the `git status` command, so instead of passing the file name, you will have to write the `git add (file path)`. Then you can commit as usual. For pushing it to the remote repo, you will use push command, `git push origin main`.

Remember when we learned that is isn't necessary that the remote link name is origin, it can be renamed. In order to push, we have to know the exact name. To see the exact name we can use `git remote -v` command, then which we to push to exact remote repo.

After running this, you might see a prompt to sign in to GitHub. It is because https link may have been used to clone the repository, if ssh link is used, it won't ask for sign in.

Note: Not all repositories allow users to make changes. Some allow users while for some you have to become a contributor to make changes.

## Git Branch

Branches in Git, makes it the best version control system. So what is branching? Branching refers to the process of creating separate lines of repository development. Suppose you have a working code in your branch which is mostly named as 'main', then you want to work on some other feature, which may result in the modification of multiple files thinking it would work but it didn't work and version released to the users has some bugs, so the question arises that how will you solve those bugs now? One thing you can do is to go through all your commits but why waste the time?

What you can do is that whatever feature you or your colleagues are working on, you can work on a different branch. Working on a different branch will not effect your main branch.

## Creating a Branch: (In CLI)

There are two option for branching 1) checkout - older one 2) switch - it came in later version (switching to different branch). any one can be used. But its better to use 'switch' as it is easy to remember.

For the first time to create a branch we have to create a branch with command:

## 15. Command:

```
git checkout -b feature1
```

```
git switch -c feature1
```

Remember to use '-b' with 'checkout' and '-c' with 'switch'. If you want to see how many branches you have, you can use `git branch` (show local branches) or `git branch --all` (show local and remote branches) command, which will display a list of all branches also indicating with a star (pointer) on which branch you are. If the pointer is on the new branch in our case 'feature1', then all the changes are made in our new branch. So you will have a main branch unchanged and your new branch with all the changes and modifications. The main branch will have no idea about the changes and commits you make in another branch.

Now how do you switch to a different branch? To switch to a different branch, use:

## 16. Command:

`git switch branch main` - 'main' or the name of the branch you want to switch to. As you switch to the main branch, your code will also change to the state where you last committed. You can also write `git switch -` to go back to the previous branch. It acts as a shortcut of switching back to the last branch you came from

## Deleting a Branch:

Deleting a branch is easy in Git, just use command:

## 17. Command:

```
git branch -d (branch name) Or git branch --delete (branch name)
```

e.g: `git branch -d feature1`

You cannot delete a branch if you are currently in that particular branch as you will get an error that the branch is being used so you should switch from the branch you want to delete to main or any other branch.

You can use `git branch` to confirm the deletion.

Note that all the commits have still not made to the remote repo as we haven't pushed anything. So to push the changes of the new branch, we use a very similar command:

```
git push origin (branch name)
```

e.g: `git push origin feature`

Now your new branch and all its commits will be visible in your remote repo.

## Branching in Detail

When you initialize git in your working directory, it creates a `.git` file in staging area which tracks all your changes. So whenever you do any commit, ofcourse they are done in some branch and by default when we say `git init`, it creates a master branch, but we rename it to 'main' branch so default for us is main branch. So now whenever we commit, what git does is, it creates a snapshot of our files with a commit number which is checksum basically and there also a pointer. Now what is pointer?

```
C:\Users\HP\Desktop\firstProject>git log
commit 7387795553eb62d3d4941da470c6782bec46b635 (HEAD -> main, origin/main)
Author: [redacted]
Date: Thu Jun 20 17:26:20 2024 +0500

    Changes to readme file

commit 987dbf3c25842f197d0c9e8349c8c5506cfe5b75 (tag: v1.0)
Author: [redacted]
Date: Thu Jun 20 15:23:24 2024 +0500

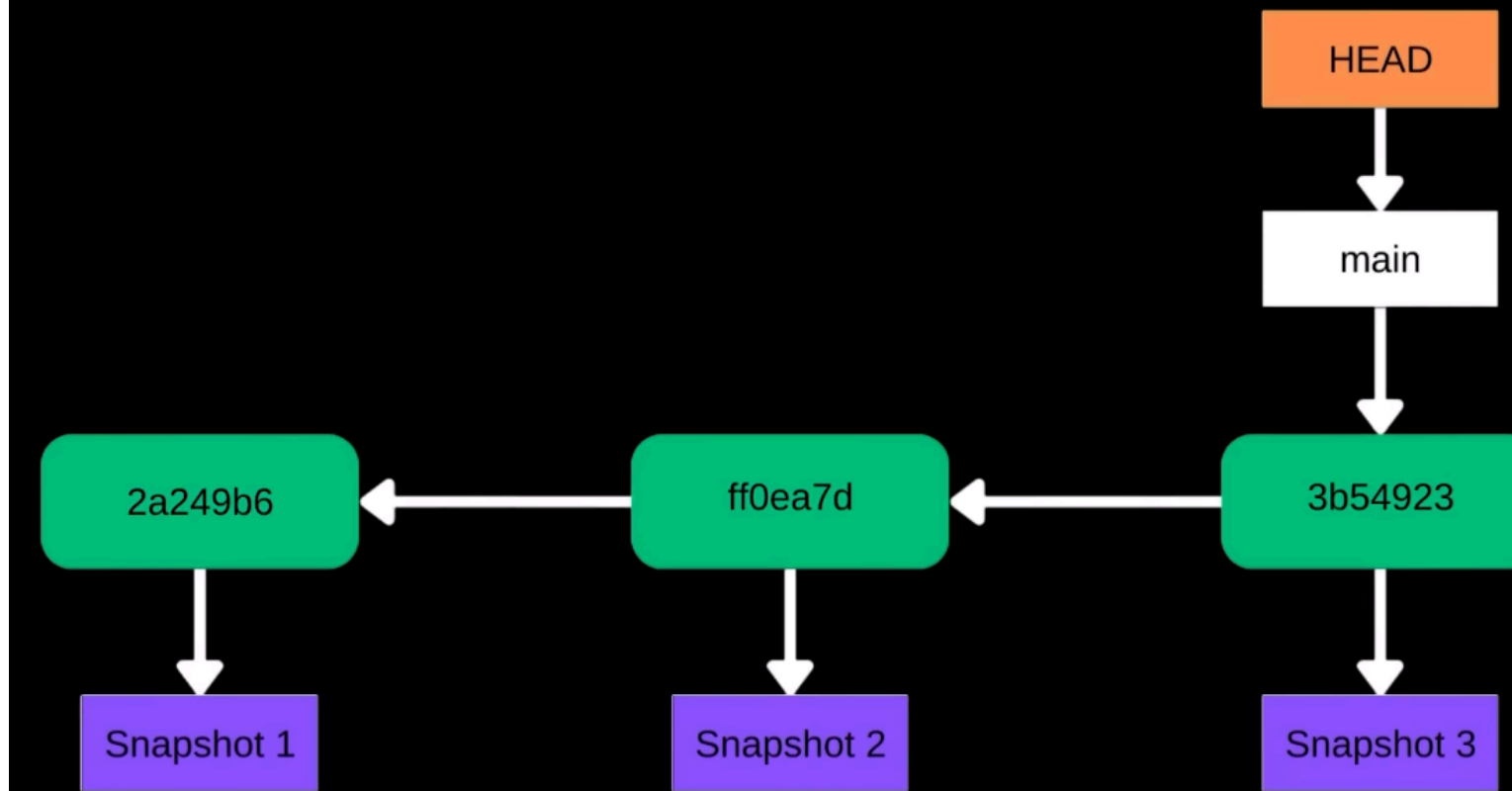
    Minor changes

commit 107e92187abc3f85f887704766ce94a151aef2ba
Author: [redacted]
Date: Thu Jun 20 15:21:57 2024 +0500

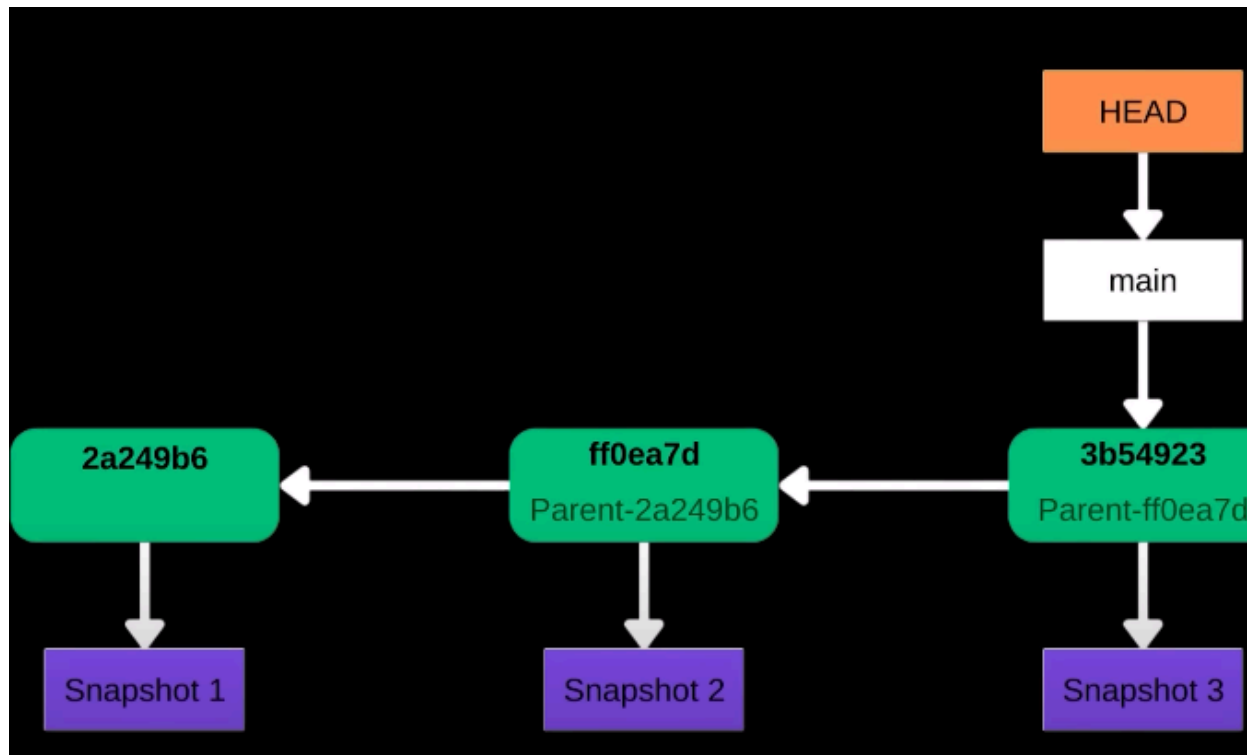
    Readme file updated
```

As you see the above picture, you will notice that there is a head which is pointing to the main branch. You can also see that that it is the last commit in our log. So the pointer points at the latest commit in a branch.



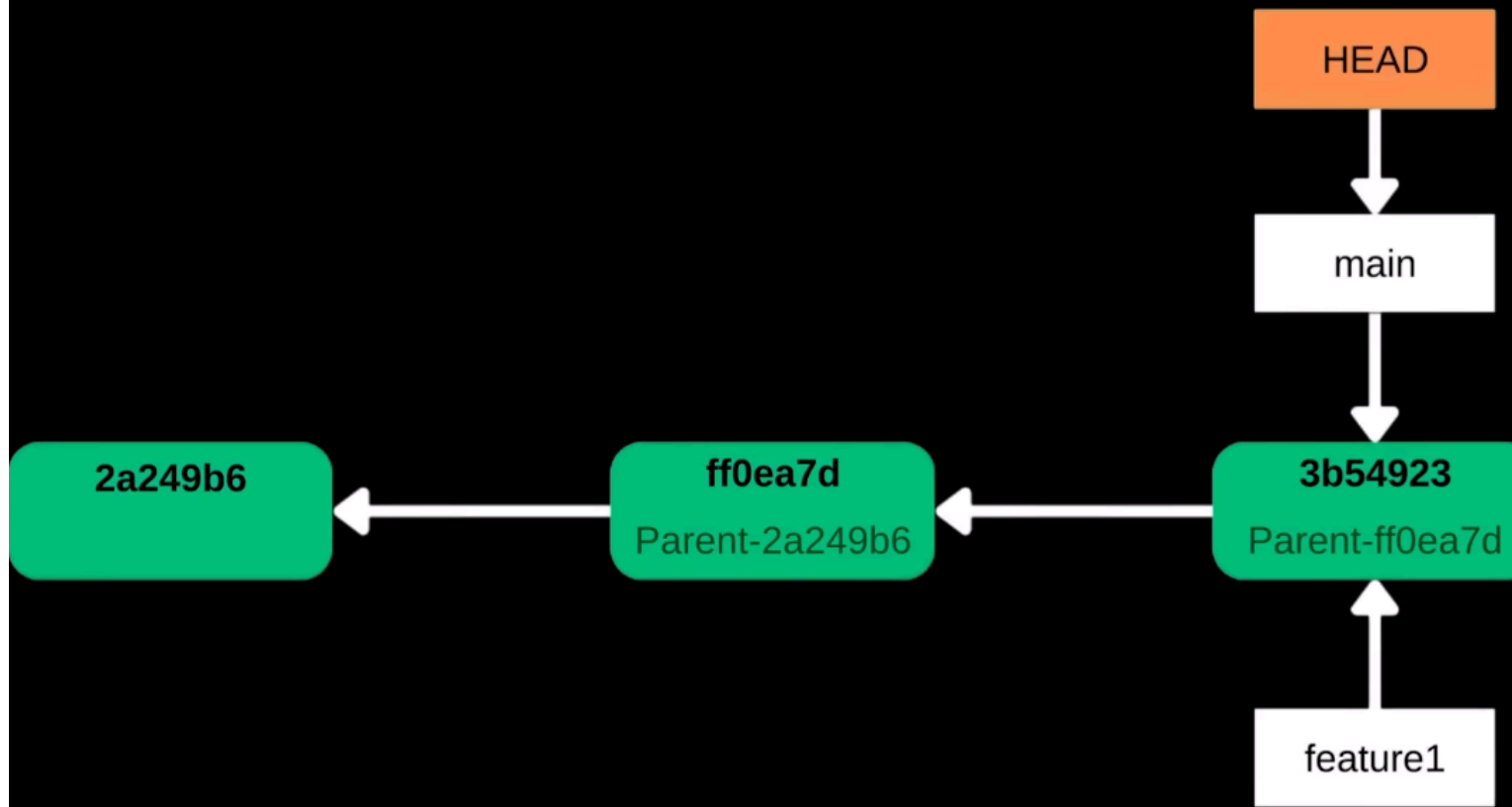


This picture shows how our commits (snapshots) are pointing to the last commit and the head with the branch at the latest commits. As a user makes another commit, the main points to it. Also in that commit, it will have those files which have been changed not the other files. Now you have the last commit and the recent commit with the new changes and it will also have the parent commit because the files that are not changed are in the previous commit so it will have the pointer to the previous one as well.

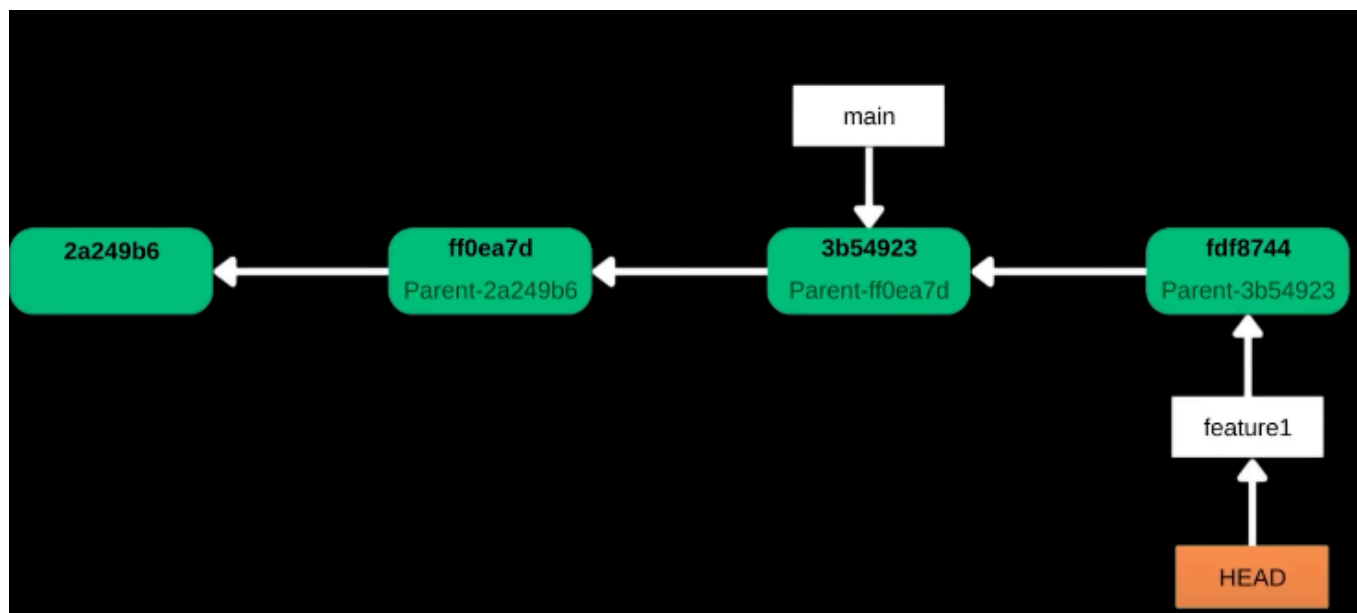


This is how it will look like as the recent commit will have the parent will have the pointer to the previous commit and so on until the very first and the head pointing to the last one.

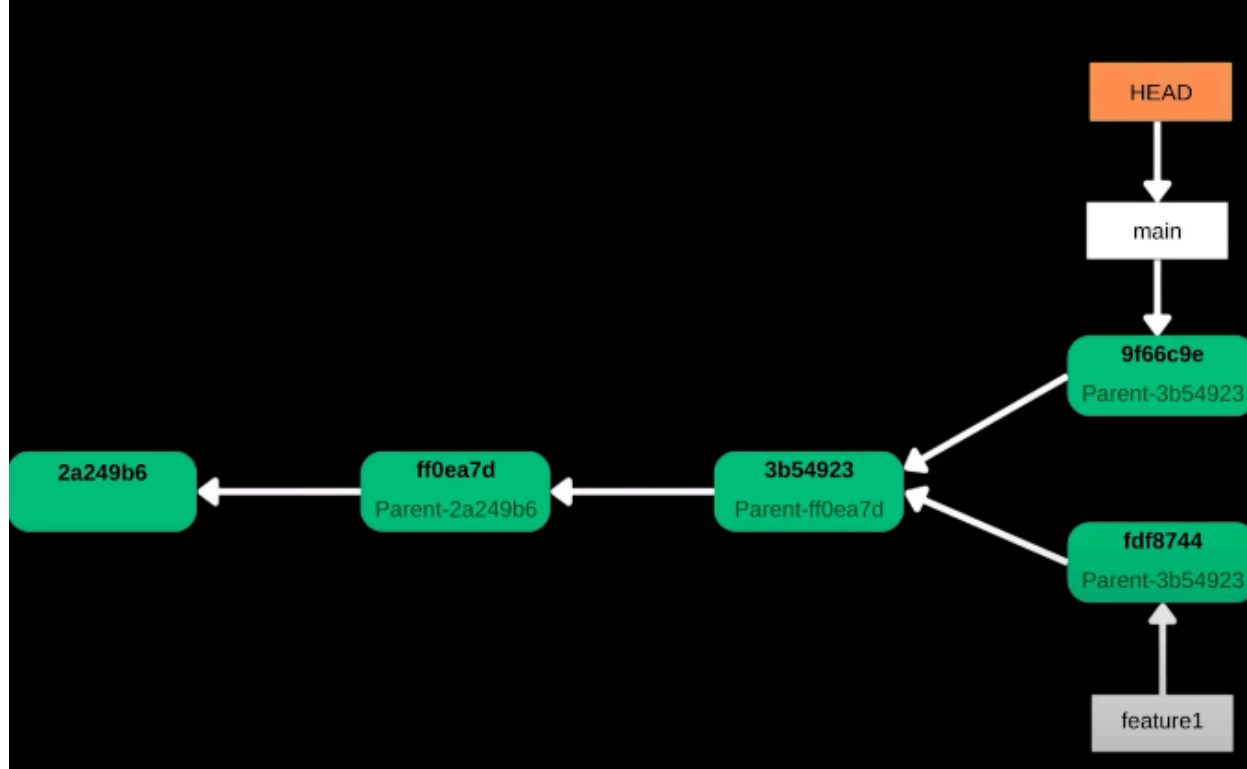
But what happens when you create a new branch? What points at what?



This is what happens when you create a new branch, both your main branch and your new branch (in this case feature1 branch) are pointing to the same commit. Then you decided to make some changes in your feature1 branch, at this point it will create a new commit and only your feature1 branch is pointing to the new commit while main is still pointing to the previous commit as the commit we made was in a different branch.



The new commit in the feature branch will be pointing to the previous commit which will be its parent commit. Also the head will move to the branch where the changes are being made. If you want to make some changes in your main branch you will have to switch to the main branch which will also make the head point to the main branch. Then when you commit and a new snapshot is taken then main and your feature1 branch will not point to the same commit. Visually it will look like this:



Here you can see that the main is pointing to the main branch with both new commits of the two branches pointing to the same parent commit.

## Ways to visualize branches:

- 1. `git log` Command:

This is the simple way we all know.

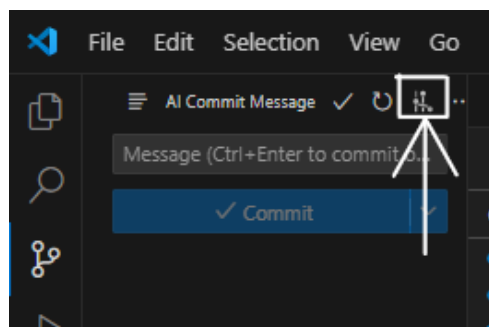
- 2. `git log --graph` Command:

You can also modify the command to see a little more visual in your terminal.

- 3. Git Graph Extension:

Git graph is a VS Code extension which will help you if you want more visual representation and understand your branches.

Install the extension and in order to use it you have to select 'Source Control' option from the side menu bar or use shortcut `Ctrl+Shift+G`. Then select this option shown in the picture below and navigate through different commits of all your branches easily.



This sums up the basic yet most important concept of Branching in GitHub

## Git Merge

So till now, we just have two branches, we can view them with `git branch` command. Suppose that your work on your feature1 branch is complete and now you are ready to release the new feature to the public in the main branch with a new tag. Now what we want to do is merge the feature branch to the main branch. How to merge it?

To merge, you need to follow these steps:

First you need to make sure that you have committed all your changes in your feature branch.

Secondly make sure you push changes of your feature branch to the remote repository. It is optional but it is a good practice to make sure your branch in remote repo is up-to-date.

Thirdly, if not, switch to main branch.

Then, pull latest changes from your remote repo to your local machine main branch. This is an important step if there are contributors or collaborators working on the repo. Pull command pulls the latest branch changes (in this case latest main branch) to your local repo to be up-to-date with the remote repo. This can be done using the following command:

## 18. Command:

```
git pull origin main (Similar to push)
```

Now, after follow above steps you can use the merge command to merge the branches:

## 19. Command:

```
git merge (branch name)
```

e.g: `git merge feature1`

This will merge the branch into one main branch, but only in your local repo. Lets not forget to push it to the remote repo.

```
git push origin main
```

Finally, your branch will be merged meaning your commits in your feature1 branch will be in your main branch and everything will be updated on your GitHub as well. Every step has importance so follow accordingly. It is safe to skip `git pull` step if you are sure that everything on your local repo is up-to-date.

---

**This concludes the guide on Git and GitHub. In this guide, we've explored the essential concepts and workflows of Git and GitHub. By now, you should have a solid understanding of:**

1. Version Controlling Basics: Understanding the importance of version control and how Git tracks changes in your projects.
2. Setting up Git: How to install and configure Git on your local machine.
3. Git Workflow:
  - Initializing a repository
  - Staging and committing changes
  - Branching and merging
4. Working with GitHub:
  - Creating and cloning repositories
  - Pushing and pulling changes

## Best Practices

To ensure a smooth and efficient workflow, keep these best practices in mind:

- **Commit Frequently:** Make small, frequent commits with clear messages to document your progress.
- **Use Branches:** Create branches for new features, bug fixes, and experiments to keep your main branch stable.
- **Review Code:** Use pull requests and code reviews to maintain code quality and facilitate collaboration.
- **Backup Regularly:** Push your changes to GitHub often to ensure you have a backup and your team is up-to-date.
- **Write Clear Messages:** Write descriptive commit messages and pull request descriptions to make it easier for others to understand your changes.

## Final Thoughts

Mastering Git and GitHub is an invaluable skill for any developer. By following the workflows and best practices outlined in this guide, you'll be well-equipped to manage your projects efficiently and collaborate effectively with your team.

**Happy coding!** 🖥️

Created By M. Saifullah Khan