Git

Configuring Git

```
# To get the global user detail
git config --get user.name
git config --get user.email

# To set the user detail globally
git config --add --global user.name "username"
git config --add --global user.email "email@example.com"

# To set the default branch name
git config  --global init.defaultBranch master
```

In git commands are devided into high-level (Porcelain) and low-level (plumbing)

Porcelain commands are used more often

- Example of Porcelain command

git status
git add
git commit
git push
git pull
git log

- Example of plumbing command

git apply
git commit-tree
git hash-object

Repositories

A repository (repo) is just a directory where all the project files (more directory and files) are stored.

A repo contains .git file which contains all internal tracking and versioning information

To initiate new git repo

```
arbab@myPC:~/git-learn$ git init
Initialized empty Git repository in /home/arbab/git-learn/.git/
arbab@myPC:~/git-learn$
```

A file can be in many states in a Git repo few important one being

- untracked -> Not being tracked by Git
- Staged -> Marked for inclusion in next commit
- commited -> Saved to the repo history

To check the status of the files

```
arbab@myPC:~/git-learn$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   management.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        content.md
        task.md

arbab@myPC:~/git-learn$
```

To add a file in the staging area

```
git add < FILE PATH >
```

For a specific file

```
git add .
```

For every files

To commit a file

```
git commit -m "YOUR MESSAGE"
```

To change the last commit message

```
git commit --amend -m "NEW MESSAGE"
```

To check all the commits

```
arbab@myPC:~/git-learn$ git log
commit 576ef95691fd5630f1de65b64ed7a0a5452bec77 (HEAD -> master)
Author: ArbAnk007 <arbabansari.study@gmail.com>
Date:   Sun Jun 23 19:25:17 2024 +0530

    First commit
```

This is unique identifier used to identify commits which is called commit hash

For convenience we can refer to any commit within Git by using the first 7 characters of its hash

While commit hashes are derived from their content changes there's also some other stuffs that affects the end hash like:-

- The commit message
- The author's name and email
- The date and time
- Parent (previous) commit hashes

HASH = SHA

Git uses cryptographic hash function called SHA-1 to generate commit hashes

Plumbing

All the data in a Git repository is stored directly in the (hidden) .git directory. That includes all the commits, branches, tags and others.

Git is made up of objects that are stored in `.git/objects` directory. A commit is just a type of object

cat-file

Git has a built in plumbing command, `cat-file` that allows us to see the contents of a commit

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

Is p flag is for pretty print

```
arbab@myPC:~/git-learn$ git cat-file -p fb87dbac67a749c1cfba5b52299e2c8068dc5c24
tree f92cf8e5bcb9d8ef0761b87f650d39a0d68108f2
parent 576ef95691fd5630f1de65b64ed7a0a5452bec77
author ArbAnk007 <arbabansari.study@gmail.com> 1719460735 +0530
committer ArbAnk007 <arbabansari.study@gmail.com> 1719460735 +0530

Second commit
arbab@myPC:~/git-learn$ git cat-file -p f92cf8e5bcb9d8ef0761b87f650d39a0d68108f2
100644 blob 01a9f9b70bbba8bc38d6d626319ac06dfcdf0e8f    content.md
100644 blob e998bb6ef4f39a71dde3b3d79c5a7515947bfa8c    management.md
100644 blob 9f73dd3a8df9b6178c890d94b346b510ea0145c1    task.md
arbab@myPC:~/git-learn$
arbab@myPC:~/git-learn$ git cat-file -p 01a9f9b70bbba8bc38d6d626319ac06dfcdf0e8f
Hello this is content file
This is edited in second commit
Bye
```

tree :- git's way of storing a directory

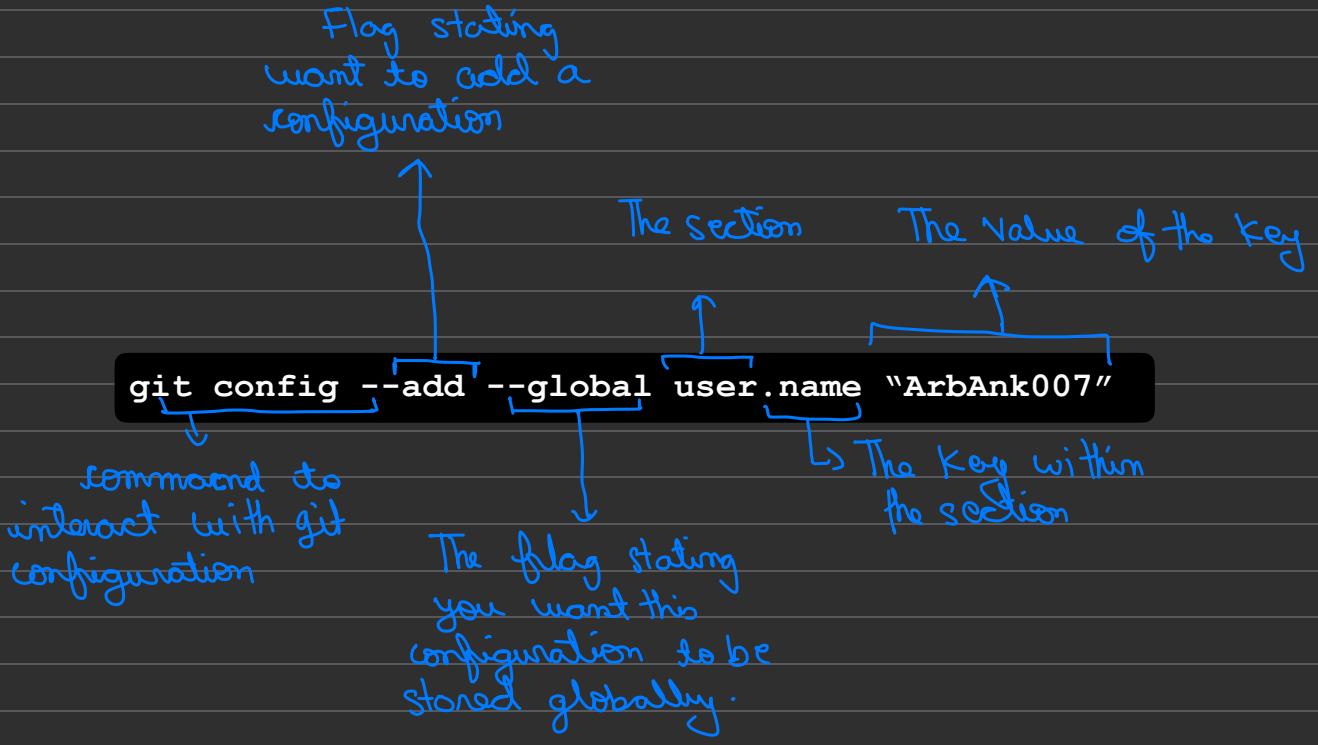
blob :- git's way of storing a file

Git stores an online snapshot of files on per commit level. It does have some performance optimizations so that .git directory doesn't get unbearably large

- Git compresses and packs files to store them more efficiently
- Git duplicates files that are the same across different commits. If a file doesn't change between commits, Git will only store it once.

Config

Git stores author information so that when making a commit it can track who made the changes



The opposite of "global" is "local" which stores the configuration in the current repository only

To get all the configuration values

```
git config --list
```

To get a single value

```
git config --get <section>.<keyname>
```

To remove a configuration value

```
git config --unset <section>.<keyname>
```

There can be duplicates in git configuration
To remove all instance of a configuration value

```
git config --unset-all <section>.<keyname>
```

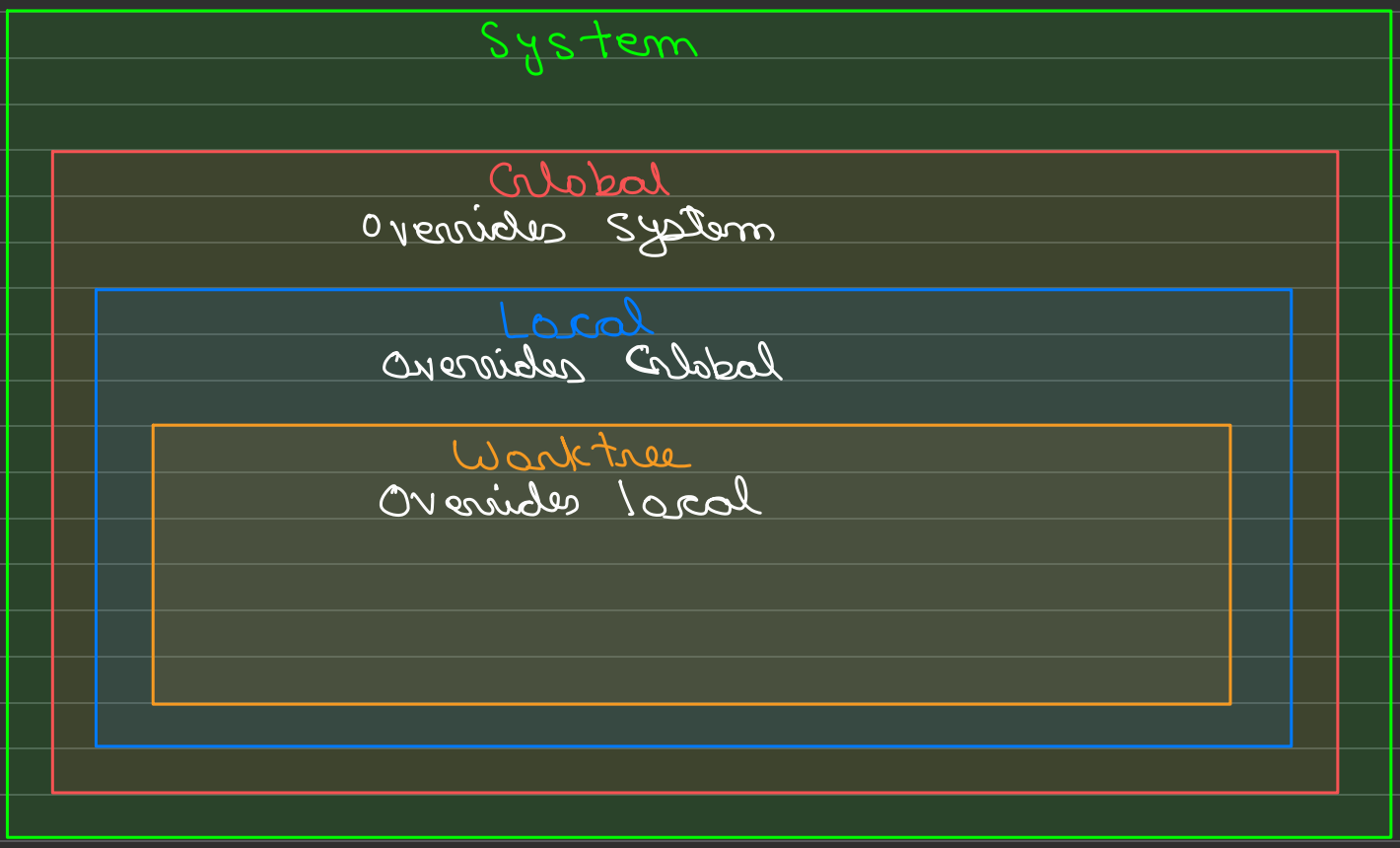
To remove an entire section from git configuration

```
git config --remove-section <section>
```

> While we can store any key/value pairs we want in git configuration, it doesn't mean we should.

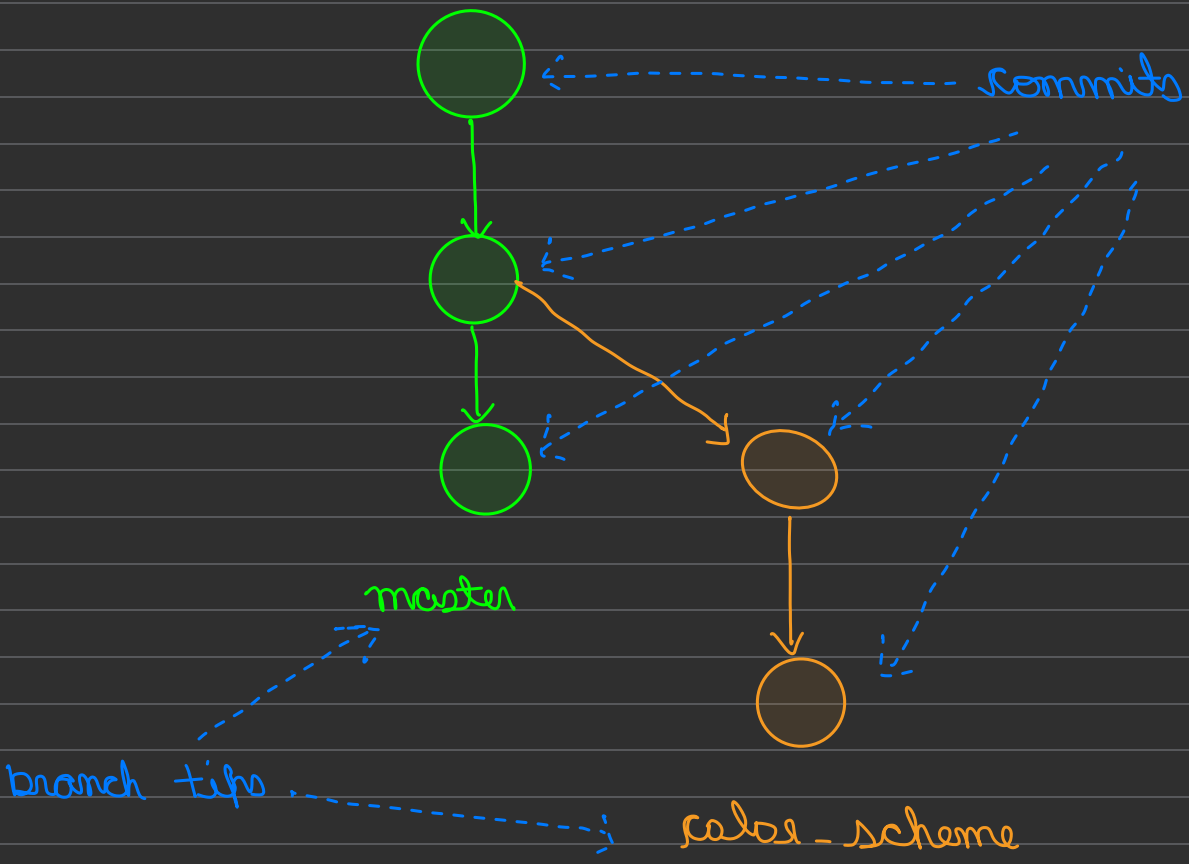
Locations of git configuration file

- system :- `/etc/gitconfig` a file that configures Git for all users on the system
- global :- `~/.gitconfig` a file that configures Git for all projects of a user
- local :- `.git/config` a file that configure git for a specific project
- worktree :- `.git/config.worktree` a file that configures Git for part of a project



Branch

A Git Branch keep track of different branches seperately



A branch is just a pointer to a commit, they're lightweight and "cheap", resource-wise to create. When we create 10 branches, we are not creating 10 copies of projects on our harddrive.

To rename a branch

```
git branch -m <oldname> <newname>
```

To delete a branch

```
git branch -d <branch_name>
```

To create a new branch

```
git switch -c <new_branch>
```

↓

This command creates and switches to the new branch

C flag tells to create a new branch if not exists

```
git branch <new_branch>
```

↓

This command just creates a new branch

To switch between branches

`git switch <branch_name>`

This is newer command which is recommended to just switch branches

`git checkout <branch_name>`

This is older command

git log Flags

- `git log --decorate=short` (default)
- `git log --decorate=no`
  No description
- `git log --decorate=full`
  shows the full ref name

```
arbab@myPC:~/git-learn$ git log --decorate=full
commit fb87dbac67a749c1cfba5b52299e2c8068dc5c24 (HEAD -> refs/heads/master)
Author: ArbAnk007 <arbabansari.study@gmail.com>
Date:   Thu Jun 27 09:28:55 2024 +0530

    Second commit

commit 576ef95691fd5630f1de65b64ed7a0a5452bec77
Author: ArbAnk007 <arbabansari.study@gmail.com>
Date:   Sun Jun 23 19:25:17 2024 +0530

    First commit
arbab@myPC:~/git-learn$ git log --decorate=no
commit fb87dbac67a749c1cfba5b52299e2c8068dc5c24
Author: ArbAnk007 <arbabansari.study@gmail.com>
Date:   Thu Jun 27 09:28:55 2024 +0530

    Second commit

commit 576ef95691fd5630f1de65b64ed7a0a5452bec77
Author: ArbAnk007 <arbabansari.study@gmail.com>
Date:   Sun Jun 23 19:25:17 2024 +0530

    First commit
arbab@myPC:~/git-learn$ git log --decorate=short
commit fb87dbac67a749c1cfba5b52299e2c8068dc5c24 (HEAD -> master)
Author: ArbAnk007 <arbabansari.study@gmail.com>
Date:   Thu Jun 27 09:28:55 2024 +0530

    Second commit

commit 576ef95691fd5630f1de65b64ed7a0a5452bec77
Author: ArbAnk007 <arbabansari.study@gmail.com>
Date:   Sun Jun 23 19:25:17 2024 +0530

    First commit
arbab@myPC:~/git-learn$
```

`git log --oneline`

Shows compact view

```
arbab@myPC:~/git-test/webflyx$ git log --oneline
8fef4a0 (HEAD -> add_classics) D: add classics.csv
8b5380d (main) C: add quotes
a132a66 B: add titles.md
1edebfd A: add contents.md
f2ae5df A: add contents.md
```

`git log --graph`

Shows tree like

```
arbab@myPC:~/git-test/webflyx$ git log --graph
* commit 8fef4a0ece39eb63e31cf80780ba584c284b634d (HEAD -> add_classics)
| Author: ArbAnk007 <arbabansari.study@gmail.com>
| Date:   Mon Jun 24 19:13:22 2024 +0530
|
|     D: add classics.csv
|
* commit 8b5380d86973d8239b7ee38ca2e2d5d096a2dd7a (main)
| Author: ArbAnk007 <arbabansari.study@gmail.com>
| Date:   Fri Jun 14 10:15:21 2024 +0530
|
|     C: add quotes
|
* commit a132a6613f86166d55e2f8fe4930c71ad3fc8964
| Author: ArbAnk007 <arbabansari.study@gmail.com>
| Date:   Fri Jun 14 10:08:13 2024 +0530
|
|     B: add titles.md
|
* commit 1edebfd4a2d119b99aedec3d0271d9357333e712
| Author: ArbAnk007 <arbabansari.study@gmail.com>
| Date:   Fri Jun 14 10:01:47 2024 +0530
|
|     A: add contents.md
|
* commit f2ae5df975f0ae698d6e380a675af077d7a35ccf
  Author: arbank07 <arbabansari.study@gmail.com>
  Date:   Fri Jun 14 09:42:01 2024 +0530

      A: add contents.md
```

`git log --parents`

Shows the hash of parents

```
arbab@myPC:~/git-test/webflyx$ git log --oneline --parents
9a65fe6 6396ad2 (HEAD -> update_dune) I: update dune.md
6396ad2 8849416 H: update dune.md
8849416 5b61d0a (main) G: update titles.md
5b61d0a 92f77a1 8fef4a0 F: Merge branch 'add_classics'
92f77a1 8b5380d E: Modified contents.md
8fef4a0 8b5380d D: add classics.csv
8b5380d a132a66 C: add quotes
a132a66 1edebfd B: add titles.md
1edebfd f2ae5df A: add contents.md
f2ae5df A: add contents.md
arbab@myPC:~/git-test/webflyx$
```

Hash of the current commit

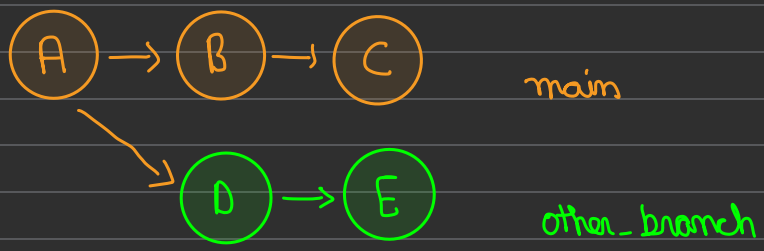
Parent hash

This has two parent hashes because this is a merge commit

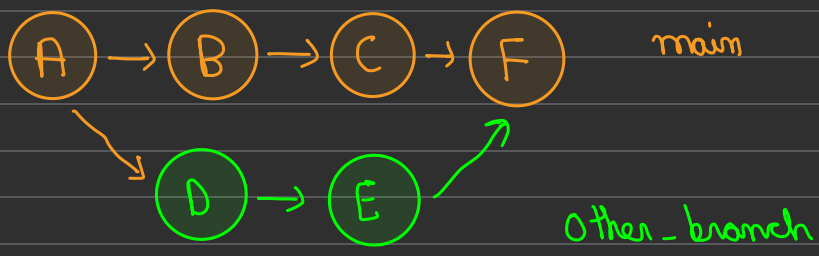
Applying all flags

```
arbab@myPC:~/git-test/webflyx$ git log --oneline --graph --parents --decorate=full
* 9a65fe6 6396ad2 (HEAD -> refs/heads/update_dune) I: update dune.md
* 6396ad2 8849416 H: update dune.md
* 8849416 5b61d0a (refs/heads/main) G: update titles.md
*   5b61d0a 92f77a1 8fef4a0 F: Merge branch 'add_classics'
|\
| * 8fef4a0 8b5380d D: add classics.csv
* | 92f77a1 8b5380d E: Modified contents.md
|/
* 8b5380d a132a66 C: add quotes
* a132a66 1edebfd B: add titles.md
* 1edebfd f2ae5df A: add contents.md
* f2ae5df A: add contents.md
arbab@myPC:~/git-test/webflyx$
```

Merge



If we merge other_branch into main. Git combines both branches by creating a new commit that has both histories as parents



F is a merge commit that has C and E as parents

F brings all the changes from D and E back into the main branch.

To merge other_branch to main we run the following command while on the main branch

```
git merge other_branch
```

then an editor prompts to enter the commit message and resolve conflicts if any

Fast forward Merge

The simplest type of merge is fast forward merge

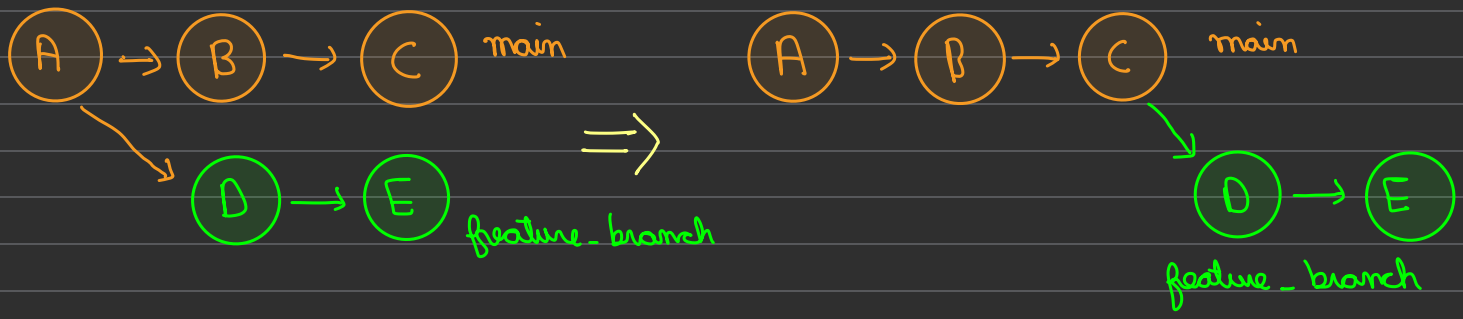
If we run `git merge other_branch` on main branch, since other_branch has all the commits that main has so it just moves the pointer of the "base" branch to the tip of the "feature" branch

Common workflow when working with git on a team of developers

- → Create a branch for a new change
- → Make the change
- → Merge the branch into main
- → Remove the branch
- → Repeat

Rebase

Visualising rebase



Eg:- We are working on feature_branch, and want to bring in the changes our team added to main so we're not working with a stale branch. We could merge main into feature_branch, but that would create an additional merge commit. Rebase avoids a merge commit by replaying the commits from feature_branch on top of main.