Git and Git Hub

Git Day - 1

The Story of Git and GitHub — The Code Odyssey

Chapter 1: The Chaos Before Version Control

In the bustling city of **Codeville**, a group of developers—Maya, Leo, and Sam—were building an amazing new app.
Every day, they shared files over email:

> "Here's `app_final_v3_really_final_final.py`."

Soon, no one knew which file was the latest version.
Sometimes, one developer's fix overwrote another's feature.
It was chaos.

Then one day, an old engineer named **Linus** walked in and said:

> "You need *Version Control* — a way to track every change to your code, forever."

Chapter 2: What is SCM?

Linus explained:

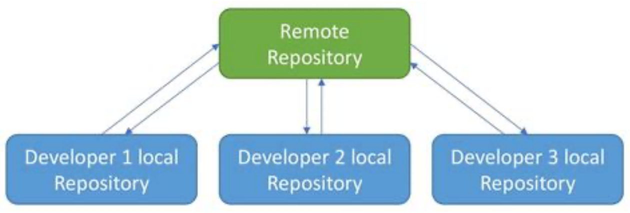
> "SCM stands for **Source Code Management** — it helps teams collaborate on code safely."

There are **three types of SCM**:

Type	Where the Code Lives	Example
Local SCM	Only on your computer	Old-school tools like RCS
Remote SCM	Central server everyone connects to	Subversion (SVN)
Distributed SCM	Each developer has the full copy of history	**Git**

> "Git," said Linus, "is *distributed* — everyone has a full history of the project. You can work even offline."

And that's how our heroes discovered **Git** — a distributed SCM that would change their coding lives forever.


Remote Repository
Developer 1 local Repository
Developer 2 local Repository
Developer 3 local Repository

Comparison of Repository Types

Aspect	Local Repositories	Remote Repositories	Distributed Repositories
Collaboration	- No collaboration — only accessible to one user or machine	- Enables team collaboration via a central server	- Full collaboration — every user has a complete copy of the repo
Community & Ecosystem Support	- Lacks ecosystem integration and third-party tools	- Integrated with platforms like GitHub, GitLab, Bitbucket	- Same ecosystem benefits as remote, plus offline flexibility
Storage Requirements	- High disk space usage (full history on one system)	- Centralized storage — less local usage	- Optimized across nodes — each user stores full but manageable copies
Dependency Management	- Hard to manage or share dependencies	- Easier with centralized dependency control	- Easier — distributed package managers and sync mechanisms
Availability & Reliability	- Reduced — data loss if local machine fails	- Dependent on remote server uptime	- Highly available — each copy acts as a backup

Scalability	- Difficult to scale for large teams or projects	- Scales but needs powerful central infrastructure	– Scales naturally — peer-to-peer replication
Maintenance Overhead	- High — manual backups, versioning, and cleanup	- Centralized maintenance, but needs monitoring	- Low — self-healing through distributed nodes
Performance	- Very fast (local operations)	- Slower — every operation involves network latency	- Fast for local operations; syncs asynchronously
Network Dependency	- None — fully offline	- Network trips required for all operations	- Works offline; syncs only when needed
Backup & Recovery	- Manual backups required	- Centralized backups needed	- Built-in redundancy (every clone is a backup)
Blocking Operations	- None (everything local)	- Possible during network or server delays	- Non-blocking — local commits independent of network
Overall Summary	Simple but isolated and hard to manage	Centralized, collaborative but network-dependent	Best of both worlds — collaboration + offline capability

---

Chapter 3: Installing Git — The Developer's Toolkit

Maya opened her terminal. To begin her Git journey, she needed the tool itself.

```
sudo su
yum update
yum install git
# or, for Ubuntu users:
apt update -y
apt install git
```

Then she verified:

```
git --version
```

Git replied with a version number — the sword was ready.

Chapter 4: Configuring Git — The Identity Scroll

Linus appeared again:

> "Before you write history, you must sign your name in every chapter."

So Maya set her identity:

```
git config --global user.name "Maya Codeheart"
git config --global user.email "maya@codeville.dev"
git config --global core.editor "vim"
```

From then on, every commit she made would bear her signature.

Chapter 5: The Gateway — SSH Setup

Now, to share code with the outside world (GitHub!), she needed a **secure connection**.

She created SSH keys (the magic keys to GitHub's castle):

```
ssh-keygen -t rsa -b 4096 -C "maya@codeville.dev"
```

Then, she copied the public key and added it to her GitHub account's settings.
Her computer was now trusted by GitHub's servers — no password needed every time she pushed code.

Chapter 6: GitHub — The Kingdom of Collaboration

GitHub was a vast land — filled with repositories (castles) and branches (towers).
It was where developers across the world stored, shared, and collaborated on code.

To enter, Maya created an account on https://github.com ↗.
Then she created a new **repository** called **"demo-project"**.

Chapter 7: Starting the Project

Maya began locally.

```
mkdir demo
cd demo
ls -a
```

She saw an empty folder — her blank canvas.

She initialized it as a Git repository:

```
cd demo
git init
```

A hidden `.git` folder appeared — her **local repository** was born.
Now Git was watching everything.

She checked:

```
ls -a
```

and saw `.git` — the secret database where Git stores every version.

Chapter 8: The Four Kingdoms of Git

Linus drew a map on the wall:

Area	Also Called	Description	Example Command
Working Tree	"Workspace"	Where you edit actual files	`git status` shows modified/untracked files
Staging Area	"Index"/ "cache"	Where you prepare files for commit	`git add file.txt` moves files here
Local Repository	".git directory"	Stores commits and history locally	`git commit` saves changes here
Remote Repository	"Origin"	Hosted copy on GitHub for sharing	`git push` uploads your commits

> "Every journey moves files through these four lands," said Linus.

Chapter 9: Creating & Tracking Files

Maya began:

```
touch Abc.txt
touch Xyz.txt
git status
```

Git replied:

> "Untracked files: Abc.txt, Xyz.txt"

They weren't yet part of the story.

She staged one:

```
git add Abc.txt
```

When she peeked inside `.git/objects/`, a new folder appeared —

> "The first hash has been born!"

Each object represented a snapshot of her file.

Then:

```
git status
```

Git said:

> "Changes to be committed: Abc.txt"
> and
> "Untracked files: Xyz.txt"

Chapter 10: The First Commit

Maya sealed her first chapter into Git history:

```
git commit -m "Learn git commit"
```

Git responded with:

> "[master (root-commit)] Learn git commit"

She peeked again inside `.git/objects/` —
More mysterious folders appeared, each representing a **commit**, **tree**, or **blob** object.

Linus smiled:

> "Each commit is a snapshot of your project at a moment in time. You can always return to it."

Chapter 11: The Great Push — Connecting to GitHub

Everything so far happened **locally** — no internet needed.
Now it was time to share her work with others.

She connected her local repo to GitHub:

```
git remote add origin git@github.com:maya-codeheart/demo-project.git
```

Then she pushed it up:

```
git push origin main
```

The repository on GitHub lit up — her code was now online.

Chapter 12: Exploring & Comparing

Later, Maya made changes and wanted to see what was different.

She used:

```
git diff
```

→ shows all changes made in the working directory compared to the last commit.

She staged those changes:

```
git add .
```

Now she wanted to see only what's staged:

```
git diff --cached
```

→ shows differences between the **staging area** and the **last commit** — i.e., what will be committed next.

Chapter 13: The Scrolls of Knowledge

Whenever Maya got stuck, she turned to the **Git manual**:

```
man git
```

But when she needed only a quick summary:

```
git add -h
```

The concise guide reminded her of command syntax without overwhelming detail.

Chapter 14: Cloning and Collaboration

Leo wanted to contribute.
He **cloned** the repo to his machine:

```
git clone git@github.com:maya-codeheart/demo-project.git
```

Now he had an **exact copy** — all commits, branches, and history.

They worked on different features, merged changes, and used Git branches to collaborate safely.

Chapter 15: The Bigger Picture — The Circle of Git Life

Here's how Maya summarized it for her team:

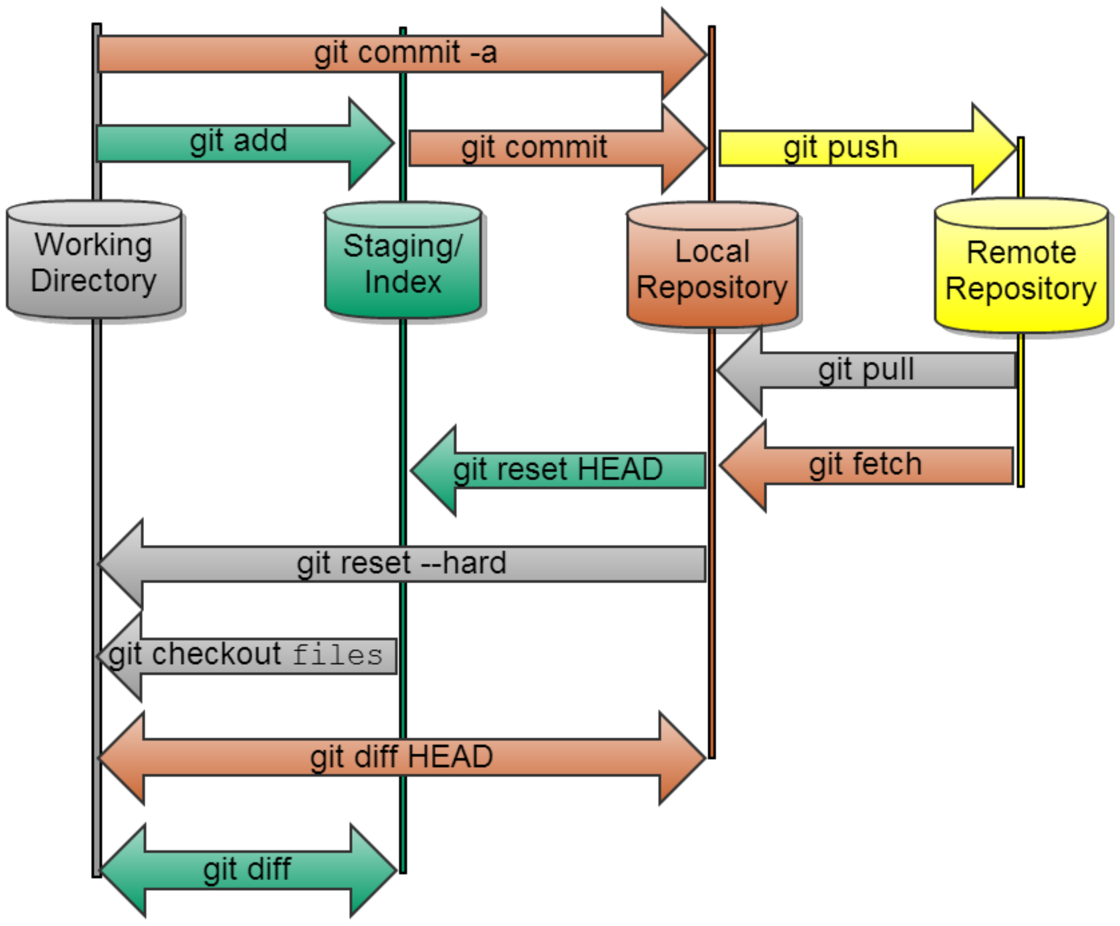
```
Working Tree → git add → Staging Area → git commit → Local Repo → git push → Remote Repo
```

And when someone else changes the remote repo, they can bring updates down:

```
git pull
```

This cycle repeats endlessly — **code evolves, commits grow, and collaboration thrives**.

Git Workflow & Commands



Epilogue — The Power of Git and GitHub

Months later, CodeCraft's app had millions of lines of code, hundreds of commits, and contributors across the globe.
Yet, everything was organized. Every bug fix, every experiment — all traceable.

Git became the silent historian of their project.
GitHub became the bridge that connected their creativity to the world.

Maya looked back at her terminal and whispered:

> "From chaos to collaboration — that's the power of Git."

Quick Recap: Git Adventure Map

Stage	Command	Purpose
Install Git	`apt-get install git`	Get Git on your system
Configure Identity	`git config --global user.name "Name"`	Set your username/email
Start Project	`git init`	Create a new local repo
Track Files	`git add file.txt`	Move files to staging area
Commit Changes	`git commit -m "msg"`	Save snapshot in local repo
Connect Remote	`git remote add origin <url>`	Link to GitHub
Push Changes	`git push origin main`	Upload commits to GitHub
Clone Repo	`git clone <url>`	Copy an existing repo
Compare Changes	`git diff`, `git diff --cached`	View differences
Help	`man git`, `git -h`	Get documentation

Git and GitHub

An exiting story to motivate people to learn git and github

SCM introduction
Types of SCM
- Local
- Remote
- Distributed
-

- Create a github account
- Create a repository

Install git
```
sudo su
 yum update
yum install git or
apt-get install git
```

Verify installation
```
git --version
```

Configuring Git
```
 git config --global user.name “name”
 git config --global user.email “email"
 git config --global core.editor "vim"
```

ssh setup

```
git clone
```

Detail help
```
man git
```

For summary/concise “help” use -h option, as in:
```
 git add -h
```

```
mkdir demo
cd demo
ls -a
git status

#Convert the folder to a git repo.
git init
ls
ls -a
```

The Four Key Areas in Git

Draw a diagram for this

Area	Also Called	Description	Example Command
Working Tree (or Working Directory)	“Workspace”	The actual files and folders you’re editing on your machine. These are not yet tracked until staged.	`git status` shows modified/untracked files
Staging Area	“Index” or “Cache”	A middle area where you prepare files for the next commit (a preview of what will be committed).	`git add file.txt`moves it here
Local Repository	“.git directory”	The database on your local machine storing commits, branches, and history.	`git commit` saves changes here

Remote Repository	"Origin" (commonly)	A server-hosted copy (e.g., GitHub, GitLab) for collaboration and backups.	`git push` / `git pull`sync changes

<u>Simple workflow</u>

```
Create
        Abc.txt
        Xyz.txt

        git status
        Adding a file to staging area
    git add Abc.txt
            Go back and check .git/objects folder
                New folder created with hash.

        git status
        Notice add files and untracked files.

    Commit
        git commit -m "Learn git commit"

            Go back and check .git/objects folder
            New folders created with hash.


    Till now we are working on local repo. We don't need internet acccess for commands till here.

        git push origin main


    Viewing only staged changes
    ---------------------------
    add a new file to the repo.
    git diff --cached
        difference between local repo directory and staged(index)
        It shows you that changes that will be in the next commit.
    git diff
        Displays all changes made to the tracked files.
    git add .
    git diff —cached
```

<u>Viewing status & differences</u>

`git status`

Show changed files and state of working tree and index/stage - need to confirm.

`git diff`

Show unstaged changes (working tree vs index/cache/stage).

`git diff --staged`

Show staged changes (index vs last commit)

`git diff <commitA> <commitB>`

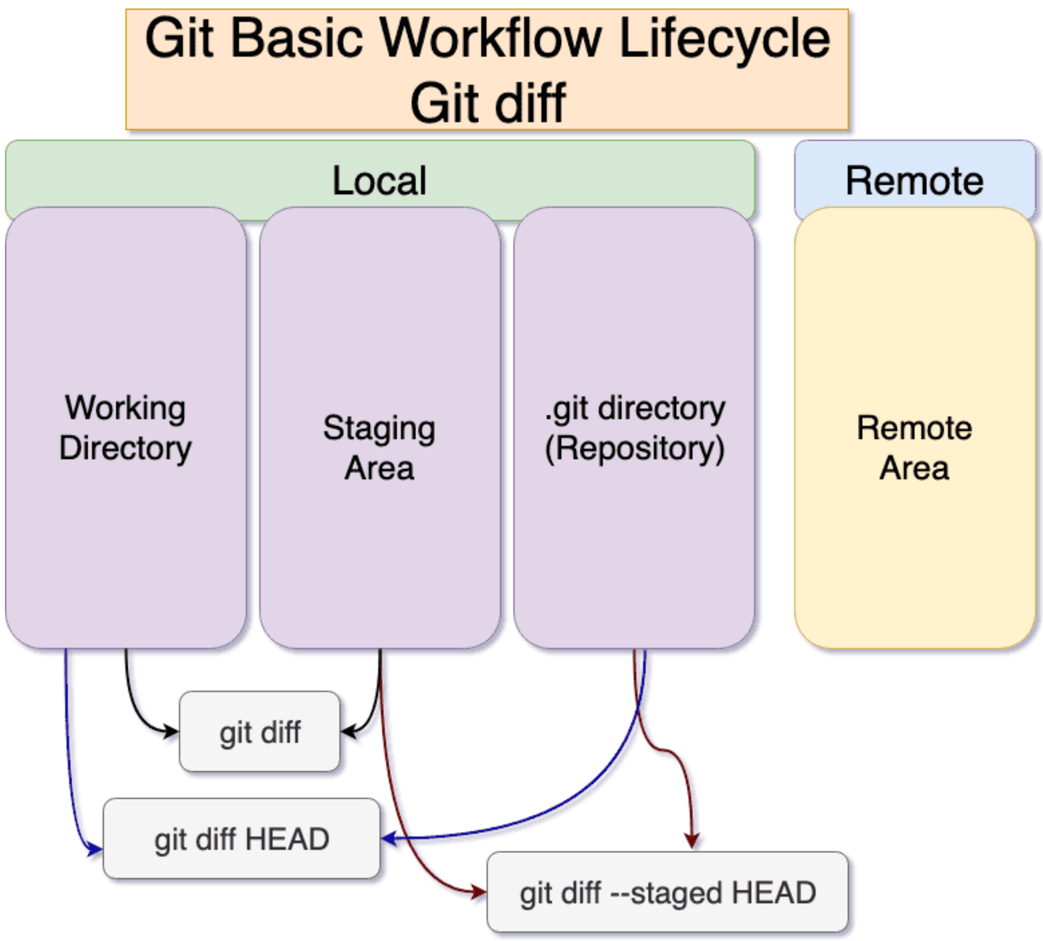
Show diffs between commits, branches, or tags.

`git log`

Show commit history (linear).

`git log --oneline --graph --decorate --all`

Compact visual commit graph.



(Confirm this diagram once)

```
git show <commit>
```

Show details and patch for a commit.

```
git blame <file>
```

Annotate file lines with commit info (who/when changed each line).

```
git shortlog -s -n
```

Summarize commits by author.

Staging & committing

```
git add file.txt
```

Stage a file (working tree → index).

```
git add -p
```

Interactively stage hunks.

```
git add .
git add -A
```

Stage all changed files (different subtle behaviors: `.` stages tracked in CWD; `-A` stages all).

```
git restore --staged file.txt
```

Unstage a file (move from index back to working tree change).

```
git commit -m "Short message"
```

Create a commit from staged changes.

```
git commit --amend
```

Amend the last commit (change message or include newly staged changes).

```
git commit -v
```

Show diff in editor when composing commit message.

```
git commit --allow-empty -m "empty commit for CI"
```

Create an empty commit (useful for triggers).

Branching & switching

https://www.freecodecamp.org/news/git-branching-commands-explained/

```
git branch
```

List local branches.

```
git branch -a
```

List all branches (local + remote).

```
git branch feature/x
```

Create a branch (does not switch).

```
git checkout feature/x
```

Switch to branch (legacy; alias available).

```
git switch feature/x
```

Switch to branch (modern recommended command).

```
git switch -c feature/y
```

Create and switch to a new branch.

```
git branch -d feature/old
```

Delete a branch (only if fully merged).

```
git branch -D feature/old
```

Force-delete branch (use carefully).

```
git merge feature/x
```

Merge named branch into current branch (creates merge commit by default).

```
git merge --no-ff feature/x
```

Force merge commit even if fast-forward possible.

```
git rebase master
```

Rebase current branch onto `master` (linearize history).

```
git rebase -i HEAD~5
```

Interactive rebase to squash, reorder, or edit last 5 commits.

```
git rebase --abort
git rebase --continue
```

Abort or continue an interrupted rebase.

Remote repositories

```
git remote -v
```

Show configured remotes and URLs.

```
git remote add origin git@github.com:user/repo.git
```

Add a remote named `origin`.

```
git fetch origin
```

Fetch updates from remote (no merge).

```
git pull
```

Fetch + merge (equivalent to `git fetch` + merge).

```
git pull --rebase
```

Fetch and rebase local commits onto fetched branch.

```
git push origin feature/x
```

Push a branch to remote.

```
git push -u origin feature/x
```

Push and set upstream (`git pull`/`git push` defaults).

```
git push --force-with-lease origin feature/x
```

Force-push safely if upstream hasn't changed unexpectedly.

```
git push --tags
```

Push tags to remote.

```
git remote set-url origin git@github.com:org/new.git
```

Change remote URL.

```
git ls-remote origin
```

Show remote refs without fetching.

Tags & releases

```
git tag v1.0.0
```

Create lightweight tag.

```
git tag -a v1.0.0 -m "release 1.0.0"
```

Create an annotated tag (recommended).

```
git show v1.0.0
```

Show tag details and commit.

```
git push origin v1.0.0
```

Push a tag to remote.

```
git push origin --tags
```

Push all tags.

Stashing

```
git stash
```

Stash uncommitted changes (working tree → stash).

```
git stash push -m "WIP: experiment"
```

Push stash with a message.

```
git stash list
```

List stashes.

```
git stash show -p stash@{0}
```

Show diff of a stash.

```
git stash apply stash@{0}
```

Reapply stash but keep it in stash list.

```
git stash pop
```

Reapply stash and remove it from stash list.

```
git stash drop stash@{0}
git stash clear
```

Remove one stash / clear all stashes.

Undoing & history rewriting (use carefully)

```
git revert <commit>
```

Create a new commit that undoes `<commit>` (safe, history-preserving).

```
git reset --soft HEAD~1
```

Move HEAD back but keep changes staged.

```
git reset --mixed HEAD~1
```

Default: move HEAD back and unstage changes (keep working tree changes).

```
git reset --hard HEAD~1
```

Move HEAD back and discard working tree and index changes (destructive).

```
git reflog
```

Show history of HEAD movement (useful to recover lost commits).

```
git cherry-pick <commit>
```

Apply the changes from `<commit>` onto current branch.

```
git filter-branch --tree-filter 'rm -rf secret' HEAD
```

Rewrite history to remove files (legacy; use `git filter-repo` if available).

Searching & selecting commits/files

```
git grep "TODO"
```

Search working tree for text.

```
git log --grep="fix memory leak"
```

Search commit messages.

```
git log -S"functionName"
```

Find commits that added/removed specific string.

```
git log --pretty=format:"%h %an %ad %s" --date=short
```

Custom commit log format.

```
git bisect start
git bisect bad HEAD
git bisect good v1.0
```

Binary search to find bad commit (tests run between steps).

Hooks (client-side & server-side)

Hooks are scripts inside `.git/hooks/` executed on events.

```
# Example: make pre-commit executable and edit it
chmod +x .git/hooks/pre-commit
```

Common hooks: `pre-commit`, `pre-push`, `commit-msg`, `post-receive` (server-side).

Use hooks for linting, tests, commit message checks, or CI integration.

Ignore files & attributes

```
echo "node_modules/" >> .gitignore
echo "*.log" >> .gitignore
git status --ignored
```

Add patterns to `.gitignore`. Ignored files won't be shown as untracked.

```
git check-ignore -v path/to/file
```

Show which `.gitignore` rule matches.

```
echo "*.bak filter=clean" > .gitattributes
```

Use `.gitattributes` for eol handling, filters, or export settings.

Large files & LFS

```
git lfs install
git lfs track "*.psd"
git add .gitattributes
```

Use Git LFS for large binary files (requires `git-lfs` installed).

Safety & best practices (commands to help)

```
git fetch --all --prune
```

Fetch all remotes and prune deleted remote branches locally.

```
git branch --merged
git branch --no-merged
```

See merged/unmerged branches (helpful before deleting old branches).

```
git remote show origin
```

Get useful info about tracking branches and push/pull behavior.

```
git config pull.rebase false
git config pull.rebase true
git config pull.ff only
```

Configure pull behavior globally or per repo.

Example workflows (commands)

Typical feature branch workflow

```
git checkout -b feature/awesome
# make changes
git add .
git commit -m "Implement awesome feature"
git push -u origin feature/awesome
# open pull request on remote
```

Rebase workflow before merging to main

```
git checkout feature/awesome
git fetch origin
git rebase origin/main
# resolve conflicts, if any, then
git push --force-with-lease origin feature/awesome
```

Merge via fast-forward or merge commit

```
git checkout main
git pull origin main
git merge --no-ff feature/awesome
git push origin main
```

Squash commits before merging

```
git checkout feature/awesome
git rebase -i origin/main
# in editor: squash commits into one
git push --force-with-lease
```

List all config

    git config --list

    core.repositoryformatversion
        https://stackoverflow.com/questions/5175208/what-are-the-different-repository-format-versions-for-the-core-repositoryforma
        https://dev.to/captainsafia/whats-in-a-git-config-3pni

View a particular config

git config —global user.name
git config <key>