Git Essential

---

1. Git stash

If you want to create an alias that stashes everything (tracked and untracked files), use this:

```
git config --global alias.stash 'stash --all'
```

---

2. Git Clean

`git clean` is a Git command used to **remove untracked files and directories** from your working directory.

Command	What it does
`git clean -n`	Show what will be deleted
`git clean -f`	Delete untracked files
`git clean -fd`	Delete untracked files and directories
`git clean -fx`	Delete all untracked and ignored files (like in `.gitignore`)

👶 Beginner Tips

- Always use `git clean -n` first to **preview**!
- If unsure, back up your files before using `git clean -f`
- Use this command when your working folder is messy and you want to "reset" untracked changes

---

3. Runs the command as a shell script (not a Git command)

```
git config --global alias.bb '!better-branch.sh'
```

- Creates a Git alias called `bb`.
- **Alias runs** a shell script named `better-branch.sh` .

The `!` is key for telling Git to run an external shell command instead of a Git subcommand.

---

4. gitconfig in macOS (global/local setting)

```
# global
nvim ~/.gitconfig

# local
nvim .git/config
```

---

5. What is `includeIf` in Git?

`includeIf` is a conditional directive in Git configuration that allows **loading specific settings based on context**, like directory path, branch, or environment.

syntax

```
[includeIf "condition"]
   setting = value
```

🔹 Use Case: Different Git Configs for Different Project Paths

You want:

- **Personal Git settings** when working in `/Users/shivamkumar/Desktop/shivam/*`
- **Company Git settings** when working in `/Users/shivamkumar/Desktop/kimbal`

🔹 Step-by-Step Setup

1. Main Global Config (`~/.gitconfig`):

```
[includeIf "gitdir:/Users/shivamkumar/Desktop/shivam/"]
  path = ~/.gitconfig-personal

[includeIf "gitdir:/Users/shivamkumar/Desktop/kimbal/"]
  path = ~/.gitconfig-company

```

> `gitdir:` matches repositories located **in or under** the specified directory.

2. Personal Config File (`~/.gitconfig-personal`):

```
[user]
  name = Shivam Personal
  email = personal@example.com

[credential]
  helper = store

[credential "https://github.com"]
  username = your-personal-username
  password = your-personal-pat

```

3. Company Config File (`~/.gitconfig-company`):

```
[user]
  name = Shivam Company
  email = company@example.com

[credential]
```

```
  helper = store

[credential "https://github.com"]
  username = your-company-username
  password = your-company-pat

```

🔹 Test It

Run this command inside a Git repo under each path:

```
git config user.name
git config credential.username

```

Ref : [How to Use .gitconfig's includeIf]

---

6 . Git Blame and Git Log

📘 Git Log – View Commit History

🔹 What it does:

Shows the **history of commits** in a Git repository.

🔹 Basic Command:

```
git log
```

🔹 What You See:

- Commit hash
- Author
- Date
- Commit message

🛠️ Useful Options:

Command	Description
`git log -p`	Shows the **diff (code changes)** with each commit
`git log --oneline`	Short, single-line summaries of each commit
`git log --stat`	Shows files changed and lines added/removed
`git log -S <string>`	Shows commits that **added or removed** a specific string
`git log --author="Name"`	Shows commits by a specific author
`git log filename`	Shows commits that changed a specific file
`git log -n 5`	Show the **last 5 commits**

✅ Example:

```
git log -p -n 1

```

> Shows the most recent commit and the exact code that was changed.

📘 Git Blame – Who Changed Each Line

🔹 What it does:

Shows **who last modified each line** of a file and in which commit.

🔹 Basic Command:

```
git blame filename
```

🔹 What You See:

- Commit hash
- Author
- Date
- Line number and content

🛠️ Useful Options:

Command	Description
`-w`	Ignores whitespace-only changes
`-L 15,26`	Show blame info for **lines 15 to 26 only**
`-C`	Track code that was **copied or moved**
`-C -C`	Track code copied from **other files and commits**
`--show-name`	Shows file names (default)
`--show-email`	Show author's email

✅ Example:

```
git blame -w -C -L 10,20 myfile.js

```

> Shows who last changed **lines 10–20** in `myfile.js`, ignoring whitespace, and tracking copied code.

🧠 When to Use Each:

Goal	Use
View commit history	`git log`
Track when/where code was added or removed	`git log -S`

Find who last modified specific lines	`git blame`
Debug a line or section of code	`git blame -L`

---

7 . 📒 Git Reflog

🔹 What is `git reflog`?

- Tracks **all changes to HEAD** (your current commit pointer).
- Records commits, checkouts, resets, rebases, merges — basically any move you make in Git.
- Helps you **recover lost commits or branches** that are no longer visible.

🔹 Why use `git reflog`?

- To **find and recover lost commits** after a reset, rebase, or accidental deletion.
- To **see the history of HEAD movements** in your repo.

🔹 Basic command:

```
git reflog

```

🔹 What you see:

- A list of recent HEAD positions with:
  - Commit hashes
  - Actions (commit, checkout, reset, etc.)
  - Commit messages or references

Example:

```
a1b2c3d (HEAD -> main) HEAD@{0}: commit: Fixed bug
4d5e6f7 HEAD@{1}: checkout: moving from feature to main

```

🔹 How to recover commits:

1. Run `git reflog` to find the lost commit hash.
2. Use:

   ```
   git checkout <commit-hash>

   ```

   or

   ```
   git reset --hard <commit-hash>

   ```

   to go back to that commit.

🔹 Tips:

- `HEAD@{n}` means “n moves ago” (e.g., `HEAD@{1}` is the previous HEAD).
- `git reflog` is local — it only tracks your repo’s HEAD history, not remote changes.

- Very handy when you mess up with `git reset` or `git checkout`.

---

8 . 📘 What is `git diff --word-diff`?

- Shows **differences between files or commits** by highlighting **word-level changes** instead of whole lines.
- Makes it easier to see **exact words added or removed**, not just entire changed lines.

command

```
git diff --word-diff
```

output

```
This is a [-bad-]{+good+} example.
```

---

9 . 📘 What is `git config --global rerere.enable=true`?

- `rerere` stands for **Reuse Recorded Resolution**.
- It helps Git **remember how you resolved merge conflicts** so that if the **same conflict happens again**, Git can automatically apply your previous resolution.

🔹 **Why use it?**

- When you merge branches, conflicts can happen.
- Resolving the same conflict multiple times is annoying.
- Enabling `rerere` saves your conflict resolutions and reuses them later, saving time.

```
git config --global rerere.enable true
```

---

10. git branch

Command	What it does	Why use it
`git branch --column`	Shows branches in multiple columns	Better branch view for many branches
`git config --global column.ui auto`	Auto-enable column output if terminal supports it	Saves typing `--column` repeatedly
`git config --global branch.sort -committerdate`	Sort branches by last commit date (desc)	Quickly find recent active branches

---

11 . Why ? 📘 `git push --force-with-lease`

🔹 **What does it do?**

- Safely **forces your local branch changes to be pushed** to the remote repository.
- Unlike `git push --force`, it **checks if the remote branch has changed** before overwriting.
- Helps **avoid accidentally overwriting someone else's work** on the remote.

🔹 **Why use it?**

- When you rewrite history locally (e.g., with `git commit --amend` or `git rebase`), the remote branch may reject a normal push.
- You need to force push to update the remote.
- `--force-with-lease` makes sure **your push only happens if no one else updated the remote branch** since your last fetch.

🔹 How it works:

- Before pushing, Git checks if the remote branch's latest commit matches what you think it is.
- If it matches, push succeeds.
- If someone else pushed changes first, your push is rejected to prevent data loss.

🔹 Command example:

```
git push --force-with-lease origin main
```

Push your local `main` branch forcefully but safely to `origin`.

🔹 Why not just `--force`?

- `--force` overwrites remote **no matter what** — risky if others have pushed changes.
- `--force-with-lease` is safer and recommended.

🔹 Tip:

If your push is rejected with `--force-with-lease`, first pull or fetch and integrate remote changes before pushing again.

---

12 . 📒 Git Commit Signing with SSH

✅ What is it?

- Signing commits proves **you authored the commit**.
- Git uses your **SSH key** to cryptographically sign commits.
- Platforms like **GitHub/GitLab** show a **"Verified" badge** on signed commits.

🧩 Step-by-Step Configuration

1. Set Git to use SSH for signing:

```
git config --global gpg.format ssh
```

➡️ Enables SSH-based signing instead of GPG.

2. Set your public SSH key for signing:

```
git config --global user.signingkey ~/.ssh/key.pub
```

➡️ Tells Git which **public key** to use for commit verification.

✅ **Note:** Add this key to your GitHub/GitLab **SSH signing keys**, not just the normal SSH keys section.

3. Set your Git email:

```
git config --global user.email "shivam@kimbsl.io"
```

➡️ The email in commits must **match your Git account** to show as "Verified."

4. (Optional) Sign all commits by default:

```
git config --global commit.gpgsign true

```

➡️ Automatically signs every commit you make.

5. Signed push (optional):

```
git push --signed
```

➡️ Signs the push itself, though most platforms focus on **commit signatures**.

🛠️ To Verify a Signed Commit:

```
git log --show-signature

```

Or view it on GitHub/GitLab — should show ✅ **Verified**.

🧠 Summary Table:

Command	Purpose
`git config --global gpg.format ssh`	Use SSH keys for signing
`git config --global user.signingkey ~/.ssh/key.pub`	Set public key for signing
`git config --global user.email`	Must match Git account email
`git config --global commit.gpgsign true`	Sign all commits automatically
`git push --signed`	Signs the push request (optional)

---

13 . 📘 Short Notes: `git maintenance`

🔹 What is it?

`git maintenance` is a feature that **automatically optimizes your Git repositories** in the background to keep them fast and clean.

🔹 Common Command:

```
git maintenance start
```

➡️ Enables **background maintenance** tasks like:

- `gc` – garbage collection
- `commit-graph` – faster `log` and `blame`
- `prefetch` – background fetching (for certain clones)

🔹 To disable:

```
git maintenance stop

```

```
gc:                    disabled
commit-graph:          hourly
prefetch:              hourly
loose-objects:         daily
incremental-repack:    daily
pack-refs:             none
```

✅ Benefits of Git Maintenance

Benefit	Why it matters
🚀 Faster Git commands	Improves speed of `log`, `status`, etc.
🧹 Automatic cleanup	Removes unnecessary or old data
🧠 Less manual work	No need to run `git gc` yourself
📈 Better performance on big repos	Keeps large repositories running smoothly
🔄 Background execution	Runs quietly without interrupting you

---

14 . 📘 Scalar in git (previously vfs)

🔹 What is Scalar?

Scalar is a Git tool developed by Microsoft to **optimize performance** when working with **very large Git repositories**.

✅ Key Features:

Feature	What it Does
🚀 Faster performance	Uses advanced Git features like commit-graph
📦 Sparse checkout	Downloads only part of the repo (on demand)
🔄 Background tasks	Runs maintenance automatically
🧠 Partial clone	Clones just metadata, fetches files when needed

🛠️ Common Commands:

- `scalar clone <repo-url>` – clone a large repo efficiently
- `scalar list` – list Scalar-managed repos
- `scalar run maintenance` – manually run Git optimizations

💡 Why Use It?

- Ideal for **huge repos** (like monorepos)
- Saves **disk space**
- Speeds up **Git operations** like `status`, `log`, `blame`

---

15 . ⏱️ Measuring Command Execution Time

- Use the `time` command before any command to measure how long it takes to run.

Syntax:

```
time <command>
```

Example:

```
time git commit-graph write
```

Notes : `>/dev/null`: Redirect output to the "null device" — basically discards the output, so nothing shows on screen.

---

16 . 📌 `git config --global fetch.writecommitgraph true`

- Enables **automatic commit-graph writing** after `git fetch`.
- Improves Git performance for commands like `git log` and `git blame`.
- Applies **globally** for your user (`--global`).
- Useful for speeding up large repositories.
- Saves you from running `git commit-graph write` manually.

---

17 . 📁 Filesystem Monitoring in Git

What is it?

- When you run commands like `git status`, Git checks your working directory for **changes** (new files, modified files, deleted files).
- Normally, Git scans the entire directory tree to find **untracked** or changed files. This can be slow in large repos.
- **Filesystem monitoring** helps Git get notified by the operating system **when files change**, instead of scanning everything all the time.
- This makes commands like `git status` faster, especially on large projects.

🔧 Important Git Configurations for Filesystem Monitoring

1. `core.untrackedCache`

```
git config core.untrackedCache true

```

- When **enabled**, Git **caches the list of untracked files**.
- This means Git doesn't have to repeatedly scan the entire directory for untracked files every time you run `git status`.
- Speeds up status checks significantly in repos with many untracked files.

2. `core.fsmonitor`

```
git config core.fsmonitor true

```

- Enables **filesystem monitoring**.
- Git listens to **filesystem events** (like file created, deleted, or modified) instead of rescanning everything.
- This is usually backed by **platform-specific file watching tools**:
  - On macOS: `FSEvents`
  - On Windows: `ReadDirectoryChangesW`
  - On Linux: `inotify`
- Greatly improves performance of Git status and other commands by reducing unnecessary scanning.

🧩 How these work together

- `core.fsmonitor` lets Git get notified about changes instantly.
- `core.untrackedCache` caches what files are untracked so Git doesn't repeatedly check them.
- Together, they speed up Git's detection of changes without full directory scans.

💡 Example Usage

Enable both globally:

```
git config --global core.untrackedCache true
git config --global core.fsmonitor true

```

---

18 . 📦 Partial Cloning in Git

What is Partial Cloning?

- Partial cloning lets you **clone only part of a repository's data** instead of the entire repo.
- This means you download **just the metadata (commits, trees)** initially, and **fetch file contents (blobs) only when needed**.
- It's useful for **huge repos** where you don't want to download everything upfront.

⚙️ Common Partial Clone Filters

- `--filter=blob:none`
  Download **all commits and trees** but **no file contents (blobs)** initially. Files download only when accessed.
- `--filter=tree:0`
  Download **no trees or blobs** initially, only commits metadata. Even more minimal, you get almost nothing upfront.

🔧 Example Commands

1. Clone without blobs (file contents):

```
git clone --filter=blob:none https://github.com/sinhaludyog/hello.git
```

- You get the repo history but no file content.
- Files download only when you check them out or access them.

2. Clone with no trees (most minimal):

```
git clone --filter=tree:0 https://github.com/sinhaludyog/hello.git
```

- You get only commit metadata initially.
- Trees and blobs are fetched on demand later.

---

19 . 📚 Git Features for Large Repos & Monorepos

1. Monorepo

- A **monorepo** is a single Git repo holding many projects or components.
- Monorepos can get **very large** with thousands of commits and files.
- Managing performance becomes important as the repo grows.

2. Multipack Indexes

- Git stores objects (commits, files) in **packfiles** to save space.
- Over time, many packfiles slow down operations.
- **Multipack indexes** are an optimization where Git creates an **index that covers multiple packfiles at once**.
- This speeds up searching for objects across packfiles, improving performance.

3. Reachability Bitmaps

- Helps Git quickly find **which commits are reachable** from refs (branches, tags).
- Instead of scanning the entire history, Git uses a **bitmap** (a compact data structure).
- Makes operations like `git log` and `git fetch` much faster in big repos.

4. Geometric Repacking

- When repacking Git objects, instead of repacking everything at once, Git does **geometric repacking**.
- This means it repacks a **small number of objects frequently**, and larger repacks less often.
- This incremental approach improves overall performance and responsiveness.

5. Sparse-Checkout

- Allows you to **check out only part of the repo's files** instead of everything.
- Useful in monorepos to work with only the projects or directories you need.
- Saves disk space and speeds up commands like `git status`.

🔧 Summary of Commands & Concepts

Feature	What It Does	Benefit
Multipack Index	Indexes many packfiles at once	Faster object lookups
Reachability Bitmap	Fast lookup of reachable commits	Speeds up log, fetch, clone
Geometric Repacking	Incremental repacking of objects	Improves repack performance
Sparse-Checkout	Check out only selected files/directories	Saves space, faster workflows

---

20 . How to use Sparse Checkout for `terraform` and `ansible` folders

Step 1: Clone your repo (without checking out all files)

- This clones the repo but **does not check out any files yet**.

Step 2: Enable sparse checkout

```
git sparse-checkout init --cone

```

- This turns on sparse checkout mode.

Step 3: Select only `terraform` and `ansible` folders

```
git sparse-checkout set terraform ansible

```

- Now Git will check out **only these two folders** into your working directory.

Step 4: Check the files in your folder

```
ls
```

- You should see only `terraform` and `ansible` directories here.
- Other files/folders will be hidden.

Summary of commands

```
git clone --no-checkout https://github.com/sinhal/devops.git
cd devops
git sparse-checkout init --cone
git sparse-checkout set terraform ansible

```
