# Git

## Tutorial for Beginners

# Prerequisites

- VS Code
- Git installed

Verify in any terminal with:

```
git
```

```
git --version
```

# Video format: 2 ways for each topic

1. CLI way (`git bash`)
2. Visual Interface way (VS Code)

# Again, what is `Git`?

Git is a popular version control system.

It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for: **Tracking code changes, who made changes, collaboration.**

# Configuration: `Git Config`

1. `git init` — initialize git repository (creates .git folder)
   a. **Repository**: A folder where Git tracks your project and its history.
2. `git config user.name 'Your Name'` — set your name
   a. What's the point? You need to tell git who you are (each commit is signed with a name and email) — so it's possible to track who makes changes.
3. `git config user.email 'example@email.com'` — set your email
4. `git config --list` — to check

*Note:*

- If you enter incorrect value, just write the command again with a new value. The old value is overwritten.
- Use `git config` without `--global` option to configure **locally — in current directory.**

# Configuration: The same visual way

1. Can’t do it through VS Code :)
    a. Need to `git config` Name and Email manually

Three levels of configuration scope:

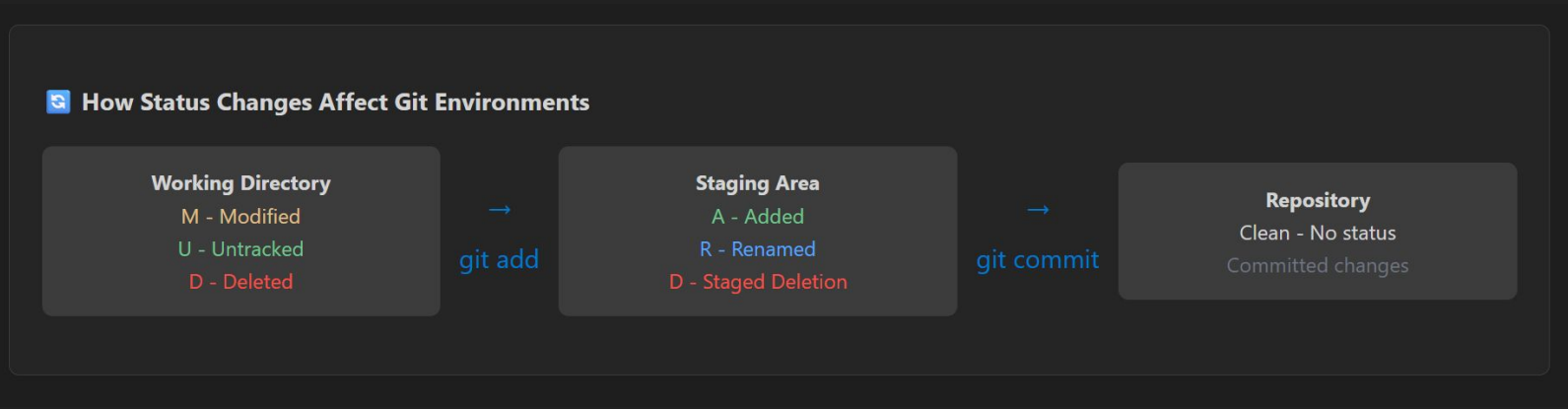
- `git config --system` — System (for all users on a device)
- `git config --global` — Global (for current user on a device)
- `git config --local` — Local (current repo. **This is default!**)

# Git Environments: Git Staging, Git Commit

1. Working Directory (Changed sth)
2. Staging Area (Saved Changes)
3. Repository (Clean - no changes)

⬇️ `git add .`
⬇️ `git commit -m 'Message'`
✅

Source of the image



# So, what is `Git Staging (Index)`?

- `Staging` — is like **dating**.
  Before we merry, we date :)

Staging is like a waiting room for your changes.

Git tracks any changes you make to your project by default. But only the chosen (staged) changes are saved. **New files are untracked unless committed.**

The main point here is that you **don’t have to include all of your changes in your next Commit**. At this stage you tell Git **what exactly you want to include in your next saved state** of the project (`Commit`) — and this is `Staging`.
⇒ This gives you control over what goes into your project history.

# `Git Staging`: Key commands

- `git add <file>` — Add a file to Staging environment
- `git add .` — Add all files **in current folder** to Staging environment
- `git add --all` OR `git add -A` — Add all files
- `git restore --staged <file>` — Unstage a file

`git log`

**See commit history**

`git status`

**See what's going on**

`git log --oneline`

**See commit history (Simplified View**

**Note: To exit the git outputs, press Q.**

# So, what is `Git Commit`?

- `Commit` — is like **marriage**.
  Now that you're sure, you can marry and settle down — s**afely save that state of your project.**

It's a save point / checkpoint / snapshot / saved state of your project.

It records **the state** of your files at a certain time, with a message describing the changes made.

- **Remember:** Git doesn't save the file themselves! Only the changes made.
- You can always go back to a previous `commit` if you need to.

The main thing here is like with functions:
1 `commit` is 1 **good change** to the project

(Functions do 1 thing
and do it well)

# `Git Commit`: Key commands

- `git commit -m 'Message'` — Commit staged changes with a message.
- `git commit` — if you don't provide `-m` option, Git opens VS Code's built-in text-editor. Add you message there, close the editor, and **commit will be made**.

**Note:** If you don't provide a message to your commit — it will be **aborted** (discarded).

- Keep first line short, under 50 characters (VS Code will prompt with red highlight)

`git status`

**See what's going on (& branch)**

`git log`

**See commit history**

# `Git Commit`: How to write Commit messages

**Rule of thumb**
When writing a commit message, ask:

**Does this sentence work after "`This commit will...`"?**
Examples:

| ❌ bad | ✅ good |
| --- | --- |
| `Added user login` | `Add user login` |
| `Fixes bug in navbar` | `Fix navbar overflow on mobile` |
| `I refactored auth logic` | `Refactor authentication logic` |

# `Git Commit`: Why imperative?

1. **Consistency with Git's own output**

Git uses imperative mood everywhere:

- `Merge branch 'main'`
- `Revert 'Fix memory leak'`
- `Update README`

Your commits blend naturally into Git history when you follow the same style.

**2. Clear, action-oriented meaning**

Imperative mood answers the question: "**What does this commit do**?"

- Add authentication; Fix navbar; Remove deprecated API etc...

# `Git Commit`: Forgot to add something? Made a typo?

- `git commit --amend -m "Corrected message"` — Fix the last commit message/ add different files/ changes
- `git reset --soft HEAD~1 \ HEAD^` — undo the last `commit` and keep your changes staged.
- `HEAD Pointer` — Git maintains a variable for referencing, called HEAD: it points to the latest `commit` and `branch`

`git status`
**See what's going on**

`git log` **(`--oneline`)**
**See commit history**

# VS Code Git Status Indicators

### M Modified

**Color:** Orange/Yellow (#e2c08d)

**Meaning:** File has been changed since last commit

**Git Status:** File is in working directory, changes not staged

**Action:** Use `git add` to stage changes

### U Untracked

**Color:** Green (#73c991)

**Meaning:** New file that Git doesn't know about

**Git Status:** File exists but not in Git repository

**Action:** Use `git add` to start tracking

### A Added (Staged)

**Color:** Green (#73c991)

**Meaning:** File is staged and ready to commit

**Git Status:** File is in staging area

**Action:** Use `git commit` to save permanently

### D Deleted

**Color:** Red (#f85149)

**Meaning:** File has been removed from working directory

**Git Status:** Deletion detected, not yet staged

**Action:** Use `git add` to stage deletion

### R Renamed

**Color:** Blue (#58a6ff)

**Meaning:** File has been renamed or moved

**Git Status:** Git detected file rename/move

**Action:** Usually auto-staged by Git

### C Conflicted

**Color:** Red/Orange (#ff7b72)

**Meaning:** Merge conflict needs resolution

**Git Status:** Conflicting changes from different branches

**Action:** Manually resolve conflicts, then stage

# `.gitignore`

Sometimes we don't want git to track certain files. This might include:

- vscode
- node_modules
- personal files like (to-do; notes)

***IMPORTANT*: Create `.gitignore` at the start of the project!** If files are ignored later, they **can still be viewed in older commits**.

Syntax:

- `#` — These are comments
- `*` — "Match any character any amount of times"

⇒ **`bash wildcards apply!`**

There are numerous .gitignore templates online.

# `Git Branches`: Different Project “Versions”

These are different versions of the project. Typically, we have:

- `main`/`master` : Where production lives (e.g., the branch linked to the website hosting).
- `dev`: The branch where all development happens (it moves a`head` of Main, and then is `merged` into `main`).

We don’t want colonial vibes, so let’s stick with ~~`master`~~ `main`. This command will rename you branch name to `main`:

- `git branch -m <name>`
- `git branch -m main`

# `Git Branches`: Why?

1. You’re not scared to break anything in production:
    a. Pushed a commit - and suddenly build fails, the site stopped responding.
2. Feeling free to experiment
3. You can keep different version of your projects, for example:
    a. `branch1`: One with HTML and CSS.
    b. `branch2`: Same but rebuilt with Bootstrap5.
4. No need to make `commits`, and `git revert` them later.

# `Git Branches`: Typical Workflow

1. The developer creates a repository with a default `main` branch
2. The developer creates a `dev` branch (as a copy of `main`) to develop new features.
3. The developer switches to `dev`/ checks out `dev`.
4. The developer creates new commits on `dev`.
5. Then the developer switches back to `main`.
6. They merge `dev` **into** `main`.
7. Now `main` = `dev` & `dev` = `main`, and the feature branch (the `dev` branch) can be deleted.

1. `git init, config...`
2. `git branch 'dev'`
3. *`git switch 'dev'` / `git checkout 'dev'`*
4. `git add, commit...`
5. *`git switch 'main'` / `git checkout 'main'`*
6. `git` **`merge`** `'dev'`
7. `git branch --delete 'dev'`

# `Git Branches`: Checkout?

When you **switch from main to dev**, **you check out dev.**

Example:

- `git checkout dev`

or (modern syntax):

- `git switch dev`

What “checkout” means in plain terms

To **check out** a branch means:

- Make that branch the current branch
- Move `HEAD` to point to that branch
- Update your working directory to match the new branch

So after:

- `git checkout dev`

You have:

`HEAD` → `dev` → latest `commit` on `dev`

# `Git Branch`: Key commands

- `git branch <name>` — Create a new branch with specified name.
- `git branch -m <name>` — Rename the branch **you're currently on** to the specified name.
- `git branch -a <name>` — List all branches.
- `git branch --delete <name>` — Delete the branch with the specified name.

- *`git switch <name>`* / *`git checkout <name>`* — Change to the branch with specified name (make that branch the current branch).
- `git merge <name>` — Merge the specified branch **into your current branch.**

`git status`

**See what's going on (& branch)**

`git log` **(`--oneline`)**

**See commit history**

# Working with Remotes: `Git push/pull`

1. Create a repository on GitHub.
2. Link local repo to remote
3. **Always `pull` first**
4. Make some changes locally...
5. Then `push` your changes to `remote`

- ***Note:*** **Don't forget to commit before pushing changes.** (can't push without **clean working directory**).
- If you delete a file from the remote repository and then pull changes to VS Code, **it will be deleted from your local computer as well.**

`Pulling`: When you (or someone else on the team) change something on the `remote` Repository, you must `pull` changes to VS Code to avoid errors.

`Pushing`: When you make changes locally, you need to `push` them to Github so **they're reflected on the `remote` Repository.**



1. Create a repository on GitHub.
2. `git remote add origin <repo url>`
3. `git pull`
4. (`git add`, `git commit`)
5. `git push -u origin main`)
   a. Here `-u` means "set upstream"

Set upstream — a tracking relationship:

=> creating a link between a local branch and a remote branch (like one on GitHub), so you can use simple `git pull` or `git push` commands without specifying the remote and branch every time.

`origin` — a default name for your main remote repo. You can rename or have multiple remotes if needed.

# Going through merge conflicts: `git merge`

- **Merge Conflict State**: when **project state** differs from your local repository and remote
  ⇒ needs to be resolved manually
- VS Code provides convenient editor for this

Workflow:

1. Merge with `git merge`
2. Accept/Discard Incoming/Outgoing changes
3. Recommit to finish merge
4. Git will automatically create message: `Merge branch 'dev'`

# `Git diff:` Compare between various commits

- `git diff HEAD` — allows to compare between current staged changes and the last commit you made.
- Extensive VS Code and Git Lens support

# Reverting changes: `git reset`

- `git reset --soft HEAD~1` — undo the last `commit` and keep your changes staged.
- with `--soft` options you can roll back your branch to a particular commit while maintaining the changes made to the working directory and index.

- ⇒ can do this natively in VS Code

1. Ctrl + Shift + P
2. Undo Last Commit

Note:
There is no native VS Code interface for `git reset`. But you can install GitLens extension which has more powerful visual interface.

Alternatively, you can reset to a particular `commit`, providing its unique ID:

```
git reset --soft
<commit_id>
```

# Sources

- W3Schools git materials helped me correctly phrase definitions.
- Also, some of my explanations and examples are based upon my knowledge from Ray Villalobos's Learning Git and GitHub course.