Git & GitHub

An Invitation to Version Control

Jyotirmaya Shivottam

January 10, 2024

Table of Contents

Version Control Systems and why we need them?

- "Version" refers to a specific state of the codebase at a given point in time, and "Control" refers to the ability to manage changes to the codebase over time. VCS are also called **Source Control Systems**.
- VCS store **snapshots** of your codebase at different points in time. Through these snapshots, they let you **compare changes** to files.
- They also allow you to **revert files / entire projects back to any state**, thereby preventing you from losing your work, e.g., if you make a mistake in your code or accidentally delete something.
- For CS460, since you'll be working in groups, VCS will help you **collaborate** with your groupmates.

What are Git & GitHub?

- Git is a free and open-source distributed VCS, that works via the command line interface, i.e., a CLI tool. You can download and install it from https://git-scm.com/downloads.
- GitHub is a web-based hosting service for version control using Git. There are other similar services, e.g., GitLab, BitBucket, etc. There are also other VCS, e.g., Mercurial, Subversion, etc.
- **Tip**: *Git is a tool, GitHub is a service. You can use Git without GitHub, but not the other way around*.
- **Tip**: *Git / GitHub are not automated[#1] backup services. They are not a replacement for Dropbox / Google Drive / OneDrive etc*.

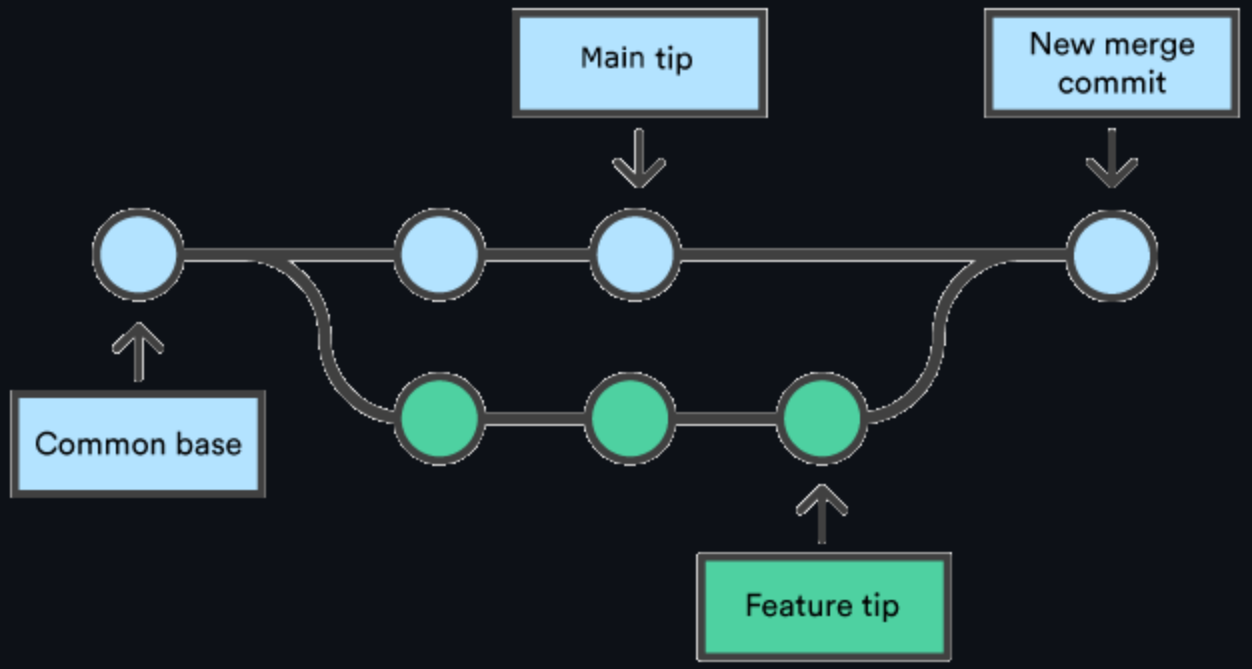
Git Primer - Some Basic Concepts

- Git is a **distributed** VCS. This means that every collaborator has a complete **working copy** of the entire codebase, including its full history.
- Git stores the codebase in a **repository**, or **repo** for short. A repo is a directory that contains all the files and folders of your project, along with a special hidden folder called `.git`.
- The `.git` folder contains all the information about the history of your project, including all the snapshots of your codebase, and the changes made to it.
- **Commits** are snapshots of your codebase at a given point in time. Each commit has a unique identifier called a **hash**, which is a long string of characters.

Git Primer - Some Keywords

- Some labels:
  - **Local** - The computer / server you are working on.
  - **Remote** - A GitHub repo that stores the codebase.
    - **Origin** - A GitHub repo, where you have pushed your codebase to / cloned your codebase from.
    - **Upstream** - A repo on GitHub that you have forked to your GitHub account.
- **Branch** - A parallel version of your codebase. You can have multiple branches, e.g., `main`, `dev`, `feature-1`, etc.
- **Forking** - Creating a copy of a repo on GitHub, under your GitHub account.
- **Pull / Merge Requests** - A request to merge a branch into another branch. This is how you collaborate with others on GitHub.

Branching & Merging Illustrated



Source: https://www.atlassian.com/git/tutorials/using-branches/git-merge

Managing the 24cs460#1 repo

1. Go to https://github.com/JeS24/24cs460dupe.
2. **Fork** the repo to your GitHub account.
3. **Clone** the repo to your computer (*next slides*).
4. Make some changes to the repo (*next slides*).
5. **Commit** the changes to your local repo (*next slides*).
6. **Push** the changes to your GitHub repo (*next slides*).
7. Create a **pull request** to merge your changes into the main repo (*next slides*).
8. Watch the pull request get **merged** (*next slides*).
9. **Sync** your local repo with the upstream repo (*next slides*).
10. Handle **merge conflicts** (*next slides*).

`git init` and `git clone`

- To clone an **existing repo**, you can use the `git clone <remote>` command. This creates a new repo in the current directory, and downloads the codebase from the remote repo.
- **OPTIONAL**: To create a **new repo**, you can use the `git init` command. This creates a new repo in the current directory.

`git remote -v`

- `git remote -v` shows you the **remote URLs** of your repo.
- `git remote show <name>` shows you information about the remote with the given name.
- **OPTIONAL**:
  - `git remote add / rm <name> <url>` adds / removes a new remote to your repo, with the given name and URL.
  - `git remote set-url <name> <url>` changes the URL of the remote with the given name.
  - `git remote prune <name>` removes all the remote-tracking branches that no longer exist on the remote with the given name.

`git branch`, `git checkout` and `git switch`

- `git branch` shows you the **branches** in your repo.
- `git branch <name>` creates a new branch with the given name.
- `git checkout <name>` / `git switch <name>` switches to the branch with the given name.
- `git checkout -b <name>` / `git switch -c <name>` creates a new branch with the given name, and switches to it.
- `git checkout <commit-hash>` / `git switch <commit-hash>` switches to the commit with the given hash.
- **Tip**: ***Branch names should be short, e.g.,*** `main`, `dev`, `feature-1`, ***etc.***

`git add`, `git rm`, `git restore`, `git status`

- *Make a change to the repo, e.g., add a new file, modify an existing file, etc.*
- `git add -A` ≡ `git add *` adds all modified files in the repo to the **staging area**.
- `git status` shows you the status of your repo, including the files in the staging area.
- `git rm --cached <file>` removes the given file from the staging area, but keeps it in the working directory.
- `git rm <file>` / `git rm -r <folder>` removes the given file / folder from the working directory (if there are no modifications).
- `git restore <path>` undoes changes to files in the working directory.

`git commit` (& `git commit --amend`)

- `git commit -m "<message>"` **commits** the changes in the staging area to the ***local repo***, with the given message. It does not affect the remote repo (till we **push**).
- `git commit -a -m "<message>"` stages ("adds") & commits all the changes to the ***local repo***, with the given message.
- **Tip**: ***Commit messages should be short, yet precise e.g., "Add README.md", "Fix typo in README.md", etc.***
- **OPTIONAL**:
  - `git commit --amend` lets you **amend** the last commit, i.e., change the commit message, add / remove files from the commit, etc.
  - `git commit --amend --no-edit` lets you amend the last commit, without changing the commit message.

`git push`

- `git push <remote> <branch>` **pushes** the changes in the local repo to the remote repo, to the given branch (here, `origin` & `main`).
- `git push -u <remote> <branch>` sets the given remote repo and branch as the **upstream** for the current branch. This lets you use `git push` (and `git pull`) without specifying the remote and branch each time.
- **OPTIONAL**:
  - `git push --force` **forces** the push to the remote repo, even if it results in a non-fast-forward merge. This is usually a bad idea, and should be avoided.
  - `git push --delete <remote> <branch>` deletes the given branch from the remote repo.

`git pull` (& `git fetch`)

- `git pull <remote> <branch>` **pulls** the changes from the remote repo to the local repo, from the given branch (here, `origin` & `main`).
- `git fetch <remote> <branch>` **fetches** the changes from the remote repo to the local repo, but ***does not merge the changes into local***.
- To **update** your local repo, you should **pull** the changes from the remote repo, and then **merge** them into your local repo.
- If there are no **merge conflicts**, `git pull` will **merge** the changes into your local repo, and **fast-forward** your branch to the latest commit on the remote repo.
- **Tip**: If you don't see remote's changes in local, try `git fetch` first, and then `git pull`.

Creating a Pull Request (PR) & Merging PRs

- Go to your GitHub repo, and click on the **Pull Requests** tab.
- Find the branch you want to merge, and click on **New pull request**.
- Select the **base** branch, i.e., the branch you want to merge into.
- Select the **compare** branch, i.e., the branch you want to merge from.
- Click on **Create pull request**.
- Add a **title** and a **description** for the PR.
- Click on **Create pull request**.
- Wait for the PR to be **merged**.

Syncing your Local Repo with the Upstream Repo

- **Tip**: ***Always pull before you push, to avoid merge conflicts.***
- Else:
  - `git fetch upstream` **fetches** the changes from the remote repo to the local repo, but ***does not merge the changes into local***.
  - `git merge upstream/<branch>` **merges** the changes from the remote repo into the local repo, from the given branch (here, `upstream` & `main`).
  - `git push origin <branch>` **pushes** the changes from the local repo to the remote repo, to the given branch (here, `origin` & `main`).

Handling Merge Conflicts

- Let us make a conflicting change to `README.md`.
- Run `git fetch`.
- `git merge upstream/main` to start a merge. It should result in a **merge conflict**.
- `git status` shows you the files with merge conflicts.
- `git diff` shows you the conflicting changes.
- Check the **staging area** in VS Code to see the conflicting changes.
- Resolve the merge conflict in VS Code. Once resolved, the file will be **staged** and you can **commit** and **push** it.
- `git merge --abort` aborts the merge.

`git reset` - The (*almost-*)Nuclear Option

- `git reset` lets you **reset** your repo to a previous state.
- `git reset --hard <commit-hash>` resets your repo to the given commit, and **deletes (!!)** all the commits after it.
- `git reset --soft <commit-hash>` resets your repo to the given commit, but **keeps (!!)** all the commits after it.
- `git reset ~<n>` resets your repo to the commit **n commits before** the current commit.
- `git reset HEAD~<n>` resets your repo to the commit **n commits before** the current commit, and **unstages** all the commits after it.
- **Tip**: ***Use*** `git reset --hard` ***with caution. It is a destructive command, and can result in data loss.***

`git revert` - The Safer Alternatives

- `git revert` lets you **undo** a commit, without deleting it.
- `git revert <commit-hash>` creates a new commit that **undoes** the changes in the given commit.
- **Tip**: *Use `git revert` instead of `git reset --hard` to undo commits.*

`git stash`

- `git stash` lets you **stash** your changes, i.e., save them for later.
- This is useful, when you want to **switch branches** or **pull remote**, but you have **uncommitted changes** in your current branch.
- `git stash pop` **pops** the last stash, i.e., it **restores** the last stash, and **deletes** it from the stash list.
- When you **stash** your changes, you can give it a **name**, e.g., `git stash push -m "<name>"`.
- `git stash list` shows you the list of stashes.
- `git stash show <stash-name>` shows you the changes in the given stash.

`.gitignore`

- `.gitignore` is a file that lets you **ignore** files and folders in your repo. The files and folders in `.gitignore` are **not tracked** by git.
- You can use **wildcards** in `.gitignore`, e.g., `*.pyc`, `*.log`, `__pycache__`, etc.
- You can also use **negation** in `.gitignore`, e.g., `!*.py`, `!*.md`, etc.
- **Tip**: *You should always add `.gitignore` to the root of your repo, and commit it.*
- **Tip**: *You can use templates for `.gitignore`, e.g., ones provided by GitHub for several languages.*

Bonus Slides

Where to get help?

- On a terminal:
  `git <subcommand> --help` ≡ `git help <subcommand>`, e.g., `git commit --help`.
- On the web:
  - Git Documentation: https://git-scm.com/docs
  - GitHub Documentation: https://docs.github.com/en
  - StackOverflow: https://stackoverflow.com/questions/tagged/git
  - ChatGPT: https://chat.openai.com/
- Some repos to practice git:
  - https://github.com/firstcontributions/first-contributions
  - Create a repo on your GitHub account and play around with it.

Some notes on GitHub usage

- GitHub has a **file size limit** of 100 MiB per file, and of 1 GiB with Git Large File Storage.
- It has a **file count limit** of 300 files per commit.
  100 MiB per file for free accounts, .
- **Tip**: ***Don't store large monolithic files, like dataset archives, in your repo.***
- **Tip**: ***Don't store too many files, like images from extracted archives, or coding artifacts, like compiled binaries, in your repo.***
- **Tip**: ***GitHub also provides access to github.dev or vscode.dev, to view and edit your repos in the browser itself.*** Try pressing the . key while viewing your repo in the browser.

GitHub Pro for Students

- GitHub offers a **free Pro account** to students, as part of the GitHub Student Developer Pack. This gives you access to a bunch of useful features, like GitHub CoPilot and GitHub Codespaces.
- You can apply for the pack at https://education.github.com/pack. They ask for some **proof of enrollment**, e.g., a student ID card, a transcript, etc., that you submit each year. I strongly recommend applying for it.
- GitHub CoPilot is an **AI pair programmer** that helps you write code. Think of it like ChatGPT but with access to your codebase. So, you can ask it to comment your code, write test cases or to explain an error / a piece of code to you, among other things.
- You can read more about it at https://copilot.github.com/.

`git config`

- `git config` lets you **configure** your git installation.
- `git config --global user.name "<name>"` sets your **name** for git commits.
- `git config --global user.email "<email>"` sets your **email** for git commits.
- `git config --global core.editor "<editor>"` sets your **editor** for git commits.
- `git config --global credential.helper "<helper>"` sets your **password / credential helper** for git.
- `git config --global --list` shows you the **global** git configuration.
- `git config --list` shows you the **local** git configuration.
- `git config --get <key>` shows you the value of the given **key** in the local git configuration, e.g., `git config --get remote.origin.url`.

`git submodule`

- `git submodule` lets you **include** another repo as a **submodule** in your repo.
- This is useful when you want to **reuse** code from another repo, e.g., a library, a framework, etc.
- `git submodule add <url>` adds the repo at the given URL as a submodule to your repo.
- `git submodule update --init --recursive` updates the submodules in your repo.
- `git submodule foreach git pull origin main` pulls the latest changes from the remote repo to the submodule.

`git <subcommand> --dry-run`

- `git <subcommand> --dry-run` is a **dry run** of a git command. It shows you what the command would do, without actually doing it.
- For example, `git add --dry-run` shows you all the files that would be added to the staging area, without actually adding them.
- Note that `git --dry-run` is not a git subcommand. It is an option that you can append to any git subcommand, e.g., `git add --dry-run`.
- Also, it is **not a universal option**. It is only available for some git subcommands, e.g., `git add`, `git rm`, `git commit`, etc.
- Other subcommands have their own dry run analogues, e.g., `git merge --no-commit --no-ff`, etc.

References

- Thorough Git tutorial: https://www.atlassian.com/git/tutorials/setting-up-a-repository
- GitHub walkthrough: https://docs.github.com/en/get-started/quickstart/hello-world

Thank you! Any questions?