Git Remotes

Git: Remotes, Rebasing & Clean History

- **Where We Are:** You know how to track files locally, build features on separate branches, and review code using `git diff` and local merges.
- **The Missing Link:** How do we share code with teammates across the internet, and how do we clean up messy, experimental commit histories before they land on `main`?
- **The Goal:** Master remote repositories (`push`/`pull`) and advanced history management (rebasing and squashing).

What is a Remote?

- **Definition:** A remote is a version of your repository hosted on a server (like GitHub, GitLab, or Bitbucket) accessible to your entire team.
- **The `origin` Name:** Git's default nickname for the primary remote URL where you cloned or pushed your project.
- **The Workflow Shift:** Your local machine is your private workspace; the remote is the central hub where code reviews and integrations happen.

Connecting and Pushing

- To link a local repository to a remote URL and upload your branch for the first time:

```
git remote add origin https://github.com/org/repo.git
git push -u origin feature/order-processor
```

- **The `-u` (upstream) flag:** Links your local branch to the remote branch. After running this once, future uploads only require typing `git push`.

Pulling Changes from Teammates

- While you are working, teammates are pushing their own updates to the remote repository. To download and integrate their work into your local branch:

  ```
  git pull origin main
  ```

- **What `pull` actually does:** It performs two actions behind the scenes: a `git fetch` (downloads remote changes) followed by a `git merge` (integrates them into your current branch).

The Messy Commit Problem

- During feature development, your local history often looks like this:
  - *"WIP: start order processor"*
  - *"fix typo"*
  - *"oops forgot semicolon"*
  - *"actually fix test"*
- **The Goal:** Before submitting a Pull Request for code review, professional engineers clean up this history so it reads like a professional, logical story.

Introduction to Interactive Rebasing

- Interactive rebase allows you to rewrite, reorder, or combine your recent local commit history before sharing it.
- To review and clean up your last 4 commits:

  ```
  git rebase -i HEAD~4
  ```

- **What opens:** A text editor showing your recent commits with commands you can apply to them (`pick`, `squash`, `reword`, etc.).

Squashing Commits (Interactive)

- **Squashing:** Merging multiple messy commits into a single, cohesive, atomic commit.
- **How it works in the rebase editor:**
  - Keep the top commit as `pick`.
  - Change `pick` to `squash` (or `s`) for the commits below it that you want to absorb.
- **Result:** Git combines all their code changes and prompts you to write *one* clean, professional commit message.

Manual Squashing Without Interactive Mode

- **The Problem:** `git rebase -i` opens a command-line text editor (like Nano or Vim), which can be intimidating for beginners.
- **The Alternative:** You can manually squash your last few commits in two simple steps:

  1. **Reset softly:** Go back in time 3 commits, but keep all your changes safely staged:

     ```
     git reset --soft HEAD~3
     ```

  2. **Commit fresh:** All your changes are now bundled together in the staging area. Just make one clean commit:

     ```
     git commit -m "Implement complete order processing featu
     ```

Standard Non-Interactive Rebasing

- **The Scenario:** You've been working on `feature/order-processor` for a few days. Meanwhile, new code was merged into `main`.
- **The Command:**

```
git switch feature/order-processor
git rebase main
```

- **What is actually happening under the hood?**
  1. **Pause:** Git temporarily "unplugs" your feature branch commits.
  2. **Update:** It fast-forwards your starting point to the absolute latest version of `main`.
  3. **Replay:** Git takes your feature commits and *replays* them one by one on top of the new `main`.
- **The Benefit:** Your history stays in a clean, straight line, proving your code works smoothly with the latest project updates before review.

Rebase vs. Merge

- **`git merge`:** Preserves exact historical context. It creates a "merge commit" showing when two branches came together.
- **`git rebase`:** Replays your feature branch commits on top of the latest `main`, creating a perfectly **straight, linear history**.
- *Golden Rule:* Never rebase commits that have already been pushed to a shared remote branch that others are working on!

Handling Rebase Conflicts

- When you rebase or pull, Git might pause if you and a teammate edited the exact same line of code.
- **The 3-Step Resolution Process:**
  1. Open the conflicted file and look for conflict markers (<<<<<<<, =======, >>>>>>>).
  2. Edit the code to keep the correct version, then save.
  3. Stage and continue:

```
git add <file>
git rebase --continue
```

Command Cheat Sheet

- `git push -u origin <branch>` — Upload branch and set upstream tracking
- `git pull` — Download and merge remote updates into your current branch
- `git rebase main` — Replay your current branch on top of `main`
- `git rebase -i HEAD~n` — Interactively edit/squash your last `n` commits
- `git reset --soft HEAD~n` — Manually undo `n` commits while keeping changes staged
- `git rebase --continue` — Resume a rebase after fixing merge conflicts