Git Branching

Git for Collaboration

- **Where We Are:** You know how to track files locally (`git init`, `git add`, `git commit`) on a single timeline (`main`).
- **The Missing Link:** How do we work on multiple features at once without breaking our code, and how do we inspect changes before merging them?
- **The Goal:** Master local branching workflows and command-line code review tools.

The Problem with Working on `main`

- **The Single Timeline Problem:** If you write experimental code directly on `main`, your project won't compile or run properly while you're in the middle of a thought.
- **The Solution—Branches:** Create parallel, isolated workspaces so you can experiment safely without breaking your stable code.

What is a Branch?

- **Definition:** A branch is simply a movable pointer to a specific commit in your project's history.
- **The `main` Branch:** Your stable, working source of truth.
- **Feature Branches:** A temporary workspace where you build a specific feature or fix a bug (e.g., `feature/order-processor`).

Creating Your First Feature Branch

- To create a new branch and jump onto it immediately:

  ```
  git switch -c feature/order-processor
  ```

- **The `-c` flag:** Tells Git to **c**reate the branch *and* switch to it in one step.
- *Check where you are anytime with:* `git branch` (look for the asterisk `*`).

Working on Your Feature Branch

- Once you are on your feature branch, everything you do looks identical to what you already know:

```
git add order_processor.py
git commit -m "Implement initial order processing logic
```

- **The Difference:** These commits are safely recorded on your *feature branch*, leaving `main` completely untouched.

Bringing Code Together (Local Merging)

- Once your feature code is written and tested locally, you integrate it back into your primary timeline:

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

- This merges your isolated feature branch back into your local `main` branch cleanly.

Introduction to Local Code Review

- Now that you know how to build features on branches, how do you review the code line by line—either for yourself or a peer sitting next to you?
- **The Rule:** Never merge a branch or show code without inspecting the exact changes first.

Inspecting Unstaged Changes (`git diff`)

- You modified a file on your branch, but haven't run `git add` yet. To see the exact line-by-line changes:

  ```
  git diff
  ```

- **How to Read It:**
  - Lines starting with **-** **(red)** were deleted.
  - Lines starting with **+** **(green)** were added.

Inspecting Staged Changes (`git diff --staged`)

- You ran `git add`, and want to double-check what is about to go into your next commit:

  ```
  git diff --staged
  ```

- *Why use this?* It ensures your staged package tells a clean, logical story before you type `git commit`.

Comparing Branches (`git diff`)

- Imagine your teammate (or you on a feature branch) made a separate set of changes. How do you see everything that changed compared to `main`?

  ```
  git diff main..feature/order-processor
  ```

- This command prints out every single difference between the two branches. **This is your command-line code review tool!**

Viewing History with Patches

- Standard `git log` only shows commit messages. To see the actual code changes inside past commits:

  ```
  git log -p -n 1
  ```

- **The Flag:** `-p` shows the patch (the code diff) for each commit, letting you audit history line by line.

Command Cheat Sheet

- `git branch` — List all local branches and see your current location
- `git switch -c <name>` — Create and switch to a new feature branch
- `git switch main` — Switch back to the main branch
- `git merge <name>` — Merge a branch into your current branch
- `git diff` — View unstaged changes in your working directory
- `git diff --staged` — View changes ready to be committed
- `git diff main..branch` — Compare differences between two branches
- `git log -p` — View past commit messages alongside code diffs