Workshop: Build Your Developer Identity with Git & GitHub

Welcome to your first week as a developer! Today, we aren't just learning abstract commands; we are going to build the foundation of your professional online presence: your **Personal Portfolio Website**.

To do this, we need to use the industry-standard tools for saving work and collaborating: **Git** and **GitHub**.

The Mission Scenario

Imagine your computer is your **private workshop**. You are building a product (your website portfolio).

1. **Git** is your magical camera and clipboard in that workshop. It records every change you make and lets you save specific versions in case you mess up later.
2. **GitHub** is a **public showroom** in the cloud. It's where you put your finished work so the world (and future employers) can see it.

Today, we build in the workshop, ship it to the showroom, and make it live on the internet.

Prerequisites

- A terminal (Git Bash on Windows, Terminal on Mac/Linux).
- Git installed (`git --version` returns a number).
- A GitHub.com account.
- A text editor (VS Code recommended).

Part 1: The Private Workshop (Getting Started Locally)

Step 1: Create your workspace

We need a folder to hold your project.

1. Open your terminal.
2. Navigate to where you want to keep your projects (e.g., `cd ~/Desktop` or `cd ~/Documents`).
3. Create a new folder and enter it:

Bash

```
mkdir my-portfolio
cd my-portfolio
```

Step 2: Initialize Git (Turning on the lights)

Right now, this is just a normal folder. Git has no idea it exists. Let's tell Git to start watching this folder.

Bash

```bash
git init
```

- **Concept:** This command creates a hidden sub-folder (`.git`) that tracks every change you make from now on. Your folder is now a **Git Repository**.

Step 3: Create content (The Working Tree)

Let's create the homepage of your portfolio.

1. Open VS Code in this folder (`code .`).
2. Create a new file named `index.html`.
3. Add this basic HTML code and save the file:

HTML

```html
<!DOCTYPE html>
<html>
<head>
   <title>My Portfolio</title>
</head>
<body>
   <h1>Hello, I'm [Your Name]!</h1>
   <p>I am a software developer currently learning Git.</p>
</body>
</html>
```

Step 4: Check the status

Go back to your terminal. Let's ask Git what it sees.

Bash

```bash
git status
```

- **What you see:** Git shows `index.html` in red under "Untracked files".
- **Concept:** Git sees a new file in your **Working Tree** (your current workspace), but it's not currently tracking it.

Step 5: Staging (Packing the box)

We want to save this file. In Git, saving happens in two steps. First, we have to choose *what* we want to save. This is called **Staging**.

Think of it like packing a box to move houses. You put items into the box before you tape it shut.

Bash

```
git add index.html
```

Run `git status` again.

- **What you see:** The file is now green and listed under "Changes to be committed". It's in the "Staging Area" (the box is packed, but open).

Step 6: Committing (Sealing the box)

Now we take a snapshot of the staged files. This is a permanent "save point" in history.

Bash

```
git commit -m "Initial commit: Added homepage structure"
```

- **Concept:** The `-m` flag allows you to attach a message. *Always* write clear messages describing what you did. This is the "tape and label" on the moving box.

Part 2: Parallel Universes (Branching)

Imagine you want to try adding a crazy neon style to your website, but you're scared you'll break the nice, clean version you just saved.

In Git, we don't need to duplicate folders. We use **Branches**.

Step 7: Create a new feature branch

Right now, you are on the default main timeline (usually called `main` or `master`). Let's create a parallel universe to experiment in.

Bash

```
git checkout -b feature-style
```

- **Concept:** `checkout -b` means "Create a new branch named 'feature-style' AND switch to it immediately."

Step 8: Make changes safely

You are now in the `feature-style` timeline. Let's make some changes.

1. In VS Code, create a new file named `style.css`.
2. Add some simple CSS:

CSS

```css
body {
    background-color: #333;
    color: #00ff00; /* Hacker green */
    font-family: sans-serif;
}
```

3. Link this CSS file in your `index.html` head section:

HTML

```html
<head>
    <title>My Portfolio</title>
    <link rel="stylesheet" href="style.css">
</head>
```

4. Save both files.

Step 9: Save the experiment

Let's save these changes *only* to this experimental branch.

Bash

```bash
git status
# You should see modified index.html and new file style.css

git add .
# (The dot means "add everything in the current folder")
```

```
git commit -m "Added experimental dark theme css"
```

Step 10: The Magic Trick (Switching branches)

Now, watch the magic of Git. Let's go back to our original timeline.

Look at your VS Code editor. Then run this command in the terminal:

Bash

```
git switch main
# Note: Older git versions use 'git checkout main'
```

Look back at VS Code. The style.css file vanished! The index.html reverted to the plain white version!

Don't panic. Your work is safe in the other universe.

Part 3: The Public Showroom (GitHub)

We have a local repo. Now we need a remote one on GitHub so we can share it.

Step 11: Create a GitHub Repository

1. Log into GitHub.com.
2. Click the "+" icon in the top right -> "New Repository".
3. **Repository Name:** my-portfolio-demo (or similar).
4. **Important:** Select "Public", but DO NOT check any boxes under "Initialize this repository with..." (No README, No .gitignore yet). We want an empty shell.
5. Click "Create repository".

Step 12: Link local to remote

GitHub will show you a page of instructions. Look for the section **"...or push an existing repository from the command line"**.

Copy the lines that look like this (use your actual URL):

Bash

```
git remote add origin https://github.com/YOUR_USERNAME/my-portfolio-demo.git
git branch -M main
git push -u origin main
```

- **Concept - Remote:** You just told your local Git that "origin" is a nickname for that specific URL on GitHub.
- **Concept - Push:** You just sent your local `main` branch commits up to the GitHub cloud.

Refresh your GitHub page. You should see your `index.html` there!

Part 4: The Payoff (Going Live)

Let's put this on the actual internet using **GitHub Pages**.

1. On your GitHub repository page, click **Settings** (top right tab).
2. On left sidebar, scroll down and click **Pages**.
3. Under "Source", click the dropdown that says "None" and select `main`.
4. Click "Save".

GitHub may take 1-5 minutes to build your site. Refresh the page until you see a green box that says: "Your site is published at..."

Click the link. You are live on the internet! Share that URL in the class chat.

Part 5: Professional collaboration (Pull Requests)

Remember that cool green "style" branch we made locally? It's still hidden on your laptop. Let's merge it into the main site using the professional workflow.

Step 13: Push the branch

We need to send that parallel universe up to GitHub.

Bash

```bash
# Make sure you are on the feature branch first
git switch feature-style

# Push this specific branch up to origin
git push origin feature-style
```

Step 14: The Pull Request (PR)

1. Go back to your GitHub repository page.
2. You should see a yellow banner saying `feature-style had recent pushes`. Click the green **"Compare & pull request"** button.
3. **Title:** "Adding Dark Mode Theme".
4. **Description:** "I created a CSS file to give the site a dark mode look. Ready for review."

5. Click **"Create Pull Request"**.

- **Concept:** A Pull Request is exactly that—you are *requesting* that the maintainer of the `main` branch *pulls* your changes in. In a job setting, your team would review your code here before accepting it.

Step 15: Merging

Since you are the owner, you can approve your own PR.

1. Scroll down on the PR page.
2. Click the green **"Merge pull request"** button.
3. Click **"Confirm merge"**.

Wait a minute or two, refresh your live GitHub Pages URL. Your live site should now have the dark mode applied!

Step 16: Closing the loop (IMPORTANT!)

Your *remote* `main` branch on GitHub is now updated with the styles. But your *local* `main` branch on your laptop is still behind!

```
Bash

# Switch back to main locally
git switch main

# Pull the new changes down from GitHub
git pull origin main
```

Your local `main` is now synchronized with the cloud.

Part 6 : The Conflict Lab

To get a **real merge conflict**, two branches must "diverge" from the same starting point and then try to change the **exact same line** before they are joined back together.

To create a conflict, we must make two branches "fight" over the same line of code at the same time.

Here is the corrected, foolproof way to trigger a conflict for your demo.

Step 17: Establish a Common Starting Point

Ensure both you and Git are on the same page.

```bash
git switch main
# Make sure your index.html has a specific line, e.g., <h1>My Portfolio</h1>
# Add and commit it if you haven't.
```

Step 18: Branch A - The "Professional" Change

We will create a branch to make the headline look professional.

1. `git checkout -b professional-version`
2. Open `index.html` and change `<h1>My Portfolio</h1>` to:
3. HTML

```html
<h1>[Your Name] | Lead Developer</h1>
```

4.
5.
6. **Save and Commit:**
7. Bash

```bash
git add index.html
git commit -m "Updated to professional headline"
```

8.
9.

Step 19: Branch B - The "Creative" Change (The Setup)

Crucial Step: You must switch back to `main` **before** creating the second branch so that both branches start from the same "old" version of the file.

1. git switch main
   (Notice the headline went back to "My Portfolio")
2. `git checkout -b creative-version`
3. Open `index.html`. The line still says `<h1>My Portfolio</h1>`. Change it to:
4. HTML

```html
<h1>Welcome to the Creative Lab of [Your Name]</h1>
```

5.
6.
7. **Save and Commit:**
8. Bash

```
git add index.html
git commit -m "Updated to creative headline"
```

9.
10.

Step 20: The Collision

Now we have two branches that both changed the same line from the same original state.

1. Switch back to your "base" branch:
2. Bash

```
git switch main
```

3.
4.
5. Merge the first branch (This will be a **Fast-Forward**, no conflict yet):
6. Bash

```
git merge professional-version
```

7.
8.
9. Now, try to merge the second branch. **This will trigger the conflict:**
10. Bash

```
git merge creative-version
```

11.
12.

The Terminal will now say:

CONFLICT (content): Merge conflict in index.html. Automatic merge failed; fix conflicts and then commit the result.

Part 7: Resolving the Conflict

Git is now "stuck" between two universes. It has modified your `index.html` file to show you both options.

Step 21: The Resolution

Open `index.html` in VS Code. You will see:

```html
<<<<<<< HEAD
<h1>[Your Name] | Lead Developer</h1>
=======
<h1>Welcome to the Creative Lab of [Your Name]</h1>
>>>>>>> creative-version
```

1. **The Choice:** You are the judge.
   - To keep the "Professional" one, delete everything except that line.
   - To keep the "Creative" one, delete everything except that line.
   - To keep **both**, just remove the `<<<<`, `====`, and `>>>>` markers.
2. **The Save:** Once the file looks exactly how you want (no markers left!), save it.
3. **The Finalization:** You must tell Git you are done "cleaning up":
4. Bash

```bash
git add index.html
git commit -m "Resolved conflict: Decided to go with professional title"
```

5.
6.

Why did this work this time?

By switching back to `main` before creating the second branch, you created **divergent history**.

1. `main` moved forward with the "Professional" change.
2. `creative-version` moved forward from an *older* point in time.
3. When you tried to merge `creative-version` into the *new* `main`, Git realized they both tried to claim the same space in history.

Part 9: Visualizing the "Train Tracks"

Note for Trainers:Visualizing the "Train Tracks" of Git history is the "Aha!" moment for most students. It turns abstract commands into a physical map of their work.

Now that you've handled a merge conflict, let's look at the "map" of what actually happened to your project's history. Git keeps a record of every branch and merge, which we can visualize directly in the terminal.

Step 22: The "All-Seeing" Log Command

Run this command in your terminal:

```bash
git log --graph --oneline --all
```

Breaking down the command:

- `--graph`: Draws the "train tracks" using text characters.
- `--oneline`: Squeezes each commit into one line so it's easier to read.
- `--all`: Shows **all** branches, not just the one you are currently standing on.

Step 23: Reading the Map

When you run that command, you should see something that looks like this:

```plaintext
* 7a2b3c4 (HEAD -> main) Resolved conflict: Decided to go with professional title
|\
| * 5e6f7g8 (creative-version) Updated to creative headline
* | 1a2b3c4 (professional-version) Updated to professional headline
|/
* 9z8y7x6 Initial commit: Added homepage structure
```

How to interpret this:

1. **The Bottom (`9z8y7x6`):** This is where you started. Both branches share this history.
2. **The Split (`|\`):** This is the moment you ran `git checkout -b` from `main` while another branch already existed. The history **diverged**.
3. **The Parallel Nodes:** Notice how the "Creative" and "Professional" commits sit side-by-side? These happened in parallel universes.
4. **The Merge (`7a2b3c4`):** The two lines of the "track" come back together. This is your **Merge Commit**—the bridge you built by resolving the conflict.

Pro-Tip: Create an Alias

The `git log --graph --oneline --all` command is long and hard to remember. Most professional developers create a "shortcut" (alias) for it.

Tell your students to run this **once** on their machines:

```bash
git config --global alias.adog "log --all --decorate --oneline --graph"
```

Now, they can just type `git adog` (think: "A Dog") to see their entire project structure beautifully visualized anytime!

Challenge for Students

To wrap up the session, ask the students to:

1. Look at their `git adog` output.
2. Identify which commit is their "Merge Commit."
3. Delete their experimental branches (`git branch -d creative-version`) and run `git adog` again to see how the labels (but not the history) change.

Part 10: The Time Traveler's Safety Net

In this lab, you will intentionally make a "mistake" on your portfolio and learn how to fix it using two different methods.

Task 1: The "Safe" Undo (`git revert`)

Use this when you have already pushed to GitHub and want to undo a specific change without confusing your teammates.

1. **Make a mistake: Open `index.html` and add something silly, like `<h1>I AM A POTATO</h1>`.**
2. **Commit it: `git add .` then `git commit -m "Added potato title"`.**
3. **Find the ID: Run `git log --oneline`. Copy the 7-character ID (SHA) of that "potato" commit.**
4. **Undo it: Run `git revert <your-commit-id>`.**

5. **Result: Git will open a text editor asking for a message. Save and close. Notice that the potato is gone, but if you run `git log`, the "potato" commit is still in your history—now followed by a "Revert" commit.**

Task 2: The "Eraser" Undo (`git reset`)

Use this only for local mistakes that you haven't shared with anyone else yet.

1. **Make another mistake: Change your background color in `style.css` to `hotpink`.**
2. **Commit it: `git add .` then `git commit -m "Pink background"`.**
3. **The "Oh No" moment: You decide that commit should never have existed.**
4. **Reset it: Run `git reset --hard HEAD~1`.**
   - **`HEAD~1` means "Go back one step from where I am now."**
   - **`--hard` means "Discard the changes entirely from my files too."**
5. **Result: Run `git log`. The "Pink background" commit has completely vanished from history. It's like it never happened.**

⚠️ Warning: Never use `git reset --hard` on commits that you have already pushed to GitHub! It will cause massive headaches for anyone else working on the project. Use `revert` for public history and `reset` for private local cleanups.

Git Cheat Sheet

Git Cheat Sheet is tailored specifically to the workflow we used to build your portfolio. It's organized by the "Life Cycle" of a feature so you can use it as a reference for your future projects.

🛠️ Setup & Initialization

Command	**What it does**
git init	Starts a brand new local repository in your current folder.

`git config --global user.name "Your Name"`	Sets your name for your commits (do this once).
`git config --global user.email "email@example.com"`	Sets your email for your commits (do this once).

📝 The Daily Workflow (Save Points)

Command	**What it does**
`git status`	**The most important command.** Tells you what files are changed or staged.
`git add <file>`	Stages a specific file (puts it in the "box").
`git add .`	Stages **all** changed files in the current directory.
`git commit -m "message"`	Permanently saves your staged snapshot with a descriptive note.

🌿 Branching (Parallel Universes)

Command	**What it does**
`git branch`	Lists all your local branches.
`git checkout -b <name>`	Creates a new branch and switches to it immediately.

git switch <name>	Switches to an existing branch.
git merge <name>	Merges the specified branch into your **current** branch.
git branch -d <name>	Deletes a branch (use this after a successful merge).

☁️ GitHub & Remotes (The Showroom)

Command	**What it does**
git remote add origin <url>	Links your local folder to a repository on GitHub.
git push -u origin main	Sends your local commits to GitHub (the -u remembers the link for next time).
git push	Sends updates to GitHub after the first link is established.
git pull	Grabs the latest changes from GitHub and merges them into your local code.

🔍 Visualizing & Debugging

Command	**What it does**
git log --oneline	Shows a condensed list of your commit history.

`git adog`	(Our custom shortcut) Shows the "train tracks" of all branches and merges.
`git diff`	Shows the exact line-by-line changes you've made but haven't staged yet.
`git merge --abort`	**Emergency Button:** Cancels a messy merge conflict and resets your files.

💡 Pro-Tips for Students

- **Commit Small, Commit Often:** It’s better to have 10 small commits that describe specific changes than one giant commit called "fixed everything."
- **Read the Terminal:** Git is very talkative. If something goes wrong, it usually tells you exactly how to fix it in the error message.
- **Always `git status`:** Before you type any command, run `git status`. It prevents you from committing things you didn't mean to.

🏆 Git Challenge Lab: The Portfolio Expansion

Objective: Add a "Contact Me" section and a "Projects" section using separate branches, then handle the integration.

Task 1: The "Contact" Feature

1. **Create a new branch called `feature-contact`.**
2. **Open your `index.html` and add a new section at the bottom:**

```html
<section id="contact">
```

```
  <h2>Contact Me</h2>

  <p>Email: student@example.com</p>

</section>
```

3. Stage and Commit this change with the message `"Add contact section"`.
4. Push this branch to GitHub.

Task 2: The "Project" Feature

1. Switch back to the `main` branch (Important: Don't branch off `feature-contact`!).
2. Create another new branch called `feature-projects`.
3. Open `index.html` and add a list of projects *above* where the contact section would be:

```
<section id="projects">

  <h2>My Projects</h2>

  <ul>

    <li>Project 1: Personal Portfolio</li>

  </ul>

</section>
```

4. Stage and Commit this change with the message `"Add projects list"`.

Task 3: The Integration (The Merge)

1. Switch back to `main`.
2. Merge the `feature-contact` branch into `main`. (This should be a clean merge).

3. Merge the `feature-projects` branch into `main`.
4. Check for Conflicts: Since you edited the same file in different places, Git might handle this automatically (a "Three-way merge"), OR it might ask for help if you placed the code on the same lines.
   - If a conflict occurs: Resolve it by keeping both sections.
   - If no conflict occurs: Check your file to make sure both the "Projects" and "Contact" sections are there.

Task 4: The "Final Polish" & Sync

1. Run your favorite visualization command: `git adog` (or `git log --graph --oneline --all`).
2. Observe how the "train tracks" look now with two features merged in.
3. Push your updated `main` branch to GitHub:
4.

   Bash

```
git push origin main
```

5.
6.
7. Check your live GitHub Pages URL to ensure your new sections are visible to the world!

💡 Bonus (For Fast Finishers)

Try to "Break" your CSS. Create a branch called `broken-styles`, change your background color to something ugly (like `magenta`), commit it, and then use `git revert` or `git reset` to "undo" that mistake and get back to your beautiful portfolio.

Quiz

1. In the Git workflow, what is the primary purpose of the 'Staging Area'?

    A. To host your code on a public server so others can see it.

    B. To permanently delete files that you no longer want in your project history.

    C. To serve as a temporary 'loading zone' where you choose which changes to include in the next commit.

    D. To automatically save every keystroke you make in your text editor.

2. Which command allows you to create a new branch called 'feature-fix' and switch to it at the same time?

    A. git merge feature-fix

    B. git branch feature-fix

    C. git commit -m 'feature-fix'

    D. git checkout -b feature-fix

3. What is the most common cause of a 'Merge Conflict' in Git?

    A. When two different branches have modified the exact same line of the same file.

    B. When you lose your internet connection while running a git push command.

    C. When you try to commit a file that hasn't been added to the staging area yet.

    D. When you have more than five branches in a single local repository.

4. Inside a file with a merge conflict, what does the marker <<<<<<< HEAD indicate?

    A. The end of the file where the conflict was successfully resolved.

    B. A line of code that Git has determined is a syntax error.

    C. The changes coming from the external branch you are trying to merge in.

    D. The beginning of the changes present on your current, active branch.

Correct answer:D

In Git, 'HEAD' refers to the branch you are currently standing on during the merge attempt.

5. When visualizing history with git log --graph --oneline, what does a 'Fast-Forward' merge look like?

A. Two lines that split apart and then curve back together at a 'Merge Commit' node.

B. A straight vertical line where the pointer simply moves up to the latest commit.

C. A series of red 'X' marks indicating that the merge was aborted.

D. A dotted line indicating that the changes were pushed to a remote server.

Right answer B
Fast-forward merges occur when there is a direct linear path between the two branches, so no divergent 'tracks' are needed.

6. Which command is used to link your local repository to a specific 'showroom' or repository on GitHub?

A. git clone origin

B. git push origin main

C. git remote add origin

D. git init

7. In a professional setting, what is the primary goal of a 'Pull Request' (PR)?

A. To notify teammates of your changes and request a code review before merging into the main branch.

B. To delete all the branches in your local repository to clean up your workspace.

C. To force your local code to overwrite whatever is currently on the GitHub server.

D. To change your GitHub username or profile settings.

8. You have made changes to your code. What is the correct sequence of commands to save these changes and send them to GitHub?

A. git add . → git commit -m 'message' → git push

B. git add . → git push → git commit -m 'message'

C. git push → git add . → git commit -m 'message'

D. git commit -m 'message' → git push → git add .

9. What happens when you run 'git pull'?

A. It creates a new branch on GitHub that matches your current local branch.

B. It sends your local commits to the remote server for others to see.

C. It deletes your local code and replaces it with a fresh copy from GitHub.

D. It fetches changes from the remote repository and immediately tries to merge them into your local branch.

10. What is the benefit of using an 'Alias' like the 'git adog' command we created?

A. It allows you to use Git without having to install it on your computer.

B. It creates a short, easy-to-remember shortcut for a long and complex Git command.

C. It improves the performance of Git so that commits happen 50% faster.

D. It encrypts your commit messages so that only you can read them.

TLDR

Title: Git & GitHub Mastery: Week 1 Essentials **Subject:** Version Control & Collaborative Development **Topics:**

- The Git Workflow (Workspace, Staging, Repository)
- Branching Strategies & Management
- Merging Mechanics (Fast-Forward vs. Three-Way)
- Conflict Resolution & Markers
- Remote Collaboration with GitHub

Summary: This study guide covers the foundational knowledge required to manage a software project's history using Git and collaborate via GitHub. It emphasizes the lifecycle of a change—from local editing to remote sharing—and highlights the critical skill of merging divergent codebases.

Key Concepts:

- **The Three Pillars of Git:** Understanding the separation between your **Working Directory** (current edits), the **Staging Area** (the "packing" phase), and the **Local Repository** (permanent snapshots).
- **Divergent History:** Learning how to create "parallel universes" using branches so that experimentation does not break the stable version of a project.
- **Merging Logic:**
  - **Fast-Forward Merge:** Happens when the destination branch hasn't moved. Git just moves the "label" forward. It looks like a straight line in your log.
  - **Three-Way Merge:** Happens when both branches have unique commits. Git creates a new "Merge Commit" to join them. This looks like "train tracks" splitting and joining in your log.
- **Conflict Markers:** Identifying and interpreting `<<<<<<< HEAD`, `=======`, and `>>>>>>>` to manually resolve disagreements between branches.
- **The PR Workflow:** Using Pull Requests as a social and professional layer for code review before integrating changes into a main codebase.

Vocabulary List:

- **Repository (Repo):** A central location where all the files and the history of a project are stored.
- **Commit:** A snapshot of your project at a specific point in time.
- **HEAD:** A pointer to the current branch/commit you are working on.
- **Origin:** The default nickname for your remote repository on GitHub.
- **Clone:** Creating a local copy of a remote repository.

- **Pull:** A command that fetches changes from a remote and merges them into your local branch.
- **Alias:** A custom shortcut for long Git commands (e.g., `git adog`).

Key Questions:

- Why is it important to use `git status` before and after staging files?
- How does a 'Fast-Forward' merge differ visually from a 'Three-Way' merge in a git log?
- What are the three steps required to resolve a merge conflict after you've edited the files?
- What is the difference between a `git branch` and a `git checkout -b`?
- How does a Pull Request facilitate better code quality in a team environment?