FAQs Summary

Git and GitHub for Absolute Beginners

Git is a distributed version control system (VCS) that lets you save **snapshots** of your project over time. It's like a running history of your code, enabling you to track changes, revert to earlier versions, and collaborate with others. **GitHub**, on the other hand, is an online platform (a Git *hosting service*) where you can store your Git repositories and collaborate with people. It's important to understand that Git (the tool) and GitHub (the website) are different – Git works on your local machine; GitHub stores repositories in the cloud for sharing. In this tutorial, we'll introduce you to Git and GitHub from the ground up, assuming no prior knowledge of the command line or version control. You'll learn the basics of using Git across Windows, macOS, and Linux, see examples of common Git commands in action, and even go through a mini-project to practice a typical workflow (including branching, pull requests, merging, and tagging a release). We'll also cover best practices (like using **.gitignore** files and writing good commit messages), explain how teams use GitHub for collaboration, and answer frequently asked questions. By the end, you should be comfortable with the fundamentals of Git and GitHub and know how to continue practicing and learning.

What is Version Control? Why Git?

Version Control is a system that records changes to files over time so you can recall specific versions later [1]. Instead of keeping a bunch of files named "draft_v1, draft_v2, final_final, final_really," a VCS like Git keeps track of every change in a special database. This way, you can revert files or entire projects back to a previous state, compare changes line-by-line, identify who made a change that might be causing an issue, and recover lost work easily [2]. It's an essential tool for collaboration – multiple developers can work on the same project simultaneously without overwriting each other's code.

There are different types of version control systems. Older systems like Subversion (SVN) use a **centralized** model – there is one central server with the official history, and collaborators "check out" files from it. This means you need to be online and connected to commit changes, and the central server is a single point of failure [3] [4]. Git is a **distributed** VCS: every user **clones** (copies) the entire repository, including its full history, to their local machine, and can make commits offline. This offers major benefits: each clone is a full backup of the project (making data loss less likely), and you can work and experiment locally without affecting others until you're ready to share your changes [5] [6].

Centralized version control vs. distributed: In centralized systems (like SVN), collaborators commit to a single server repository, which everyone shares. In Git's distributed model, each user has their own **local repository** (copy of the project history) and only pushes or pulls updates to a shared **remote** when ready [7]. This distributed nature is the main difference between Git and older systems – Git doesn't rely on constant server access and is generally faster and more flexible for branching and merging changes.

Because of these advantages, Git has become *the* dominant VCS used in industry today. If you learn Git, you'll not only manage your own projects better, but you'll also be using the same workflow that major teams and open-source projects use. Now, let's set it up and learn some basics.

Getting Started: Installation and Setup (Windows, macOS, Linux)

Installing Git: You can install Git on Windows, macOS, and Linux:

- **Windows:** Install *Git for Windows*, which includes the Git CLI and Git Bash (a terminal emulator). Download it from the official site (git-scm.com) and run the installer. You'll then have a **Git Bash** program where you can run Git commands.
- **macOS:** Modern macOS comes with Git pre-installed. You can check by opening **Terminal** and typing `git --version`. If not present, installing Xcode Command Line Tools (`xcode-select --install`) or using Homebrew (`brew install git`) will set it up.
- **Linux:** Install from your distro's package manager (for Debian/Ubuntu: `sudo apt-get install git`, for Fedora: `sudo dnf install git`, etc.).

After installation, open your terminal (or Git Bash on Windows) and set your **name** and **email** – this information will be associated with your commits:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "you@example.com"
```

This configures Git with your identity, so that each commit you make will record who made it. You only need to do this once. You can also configure other defaults (like your default editor) but we'll stick to essentials.

Using the Command Line: We will use Git via the command line for examples. On Windows, open **Git Bash**; on Mac/Linux, use **Terminal**. The commands are the same across platforms. In the examples, lines starting with `$` indicate a command to type, and lines below (if any) show the output.

Before diving into Git, let's clarify some terminology:

- **Repository (repo):** A folder tracked by Git. When you "put a project under Git", you are creating a repository – essentially a `.git` folder inside that contains all version history.
- **Working directory:** Your project's files on disk, which you edit.
- **Staging area (index):** A temporary area where you put changes you plan to commit.
- **Commit:** A snapshot of the repository's state, with a message describing the changes. This is what forms the history.
- **Remote:** A copy of the repository on a server (like on GitHub). Typically called "origin" by default.
- **Branch:** A named series of commits (like a pointer to a commit, which moves forward as you add commits). By default, Git has a branch named `main` (historically `master`). Branches let you work on separate lines of development.
- **Clone:** A full copy of a remote repository to your local machine (including all branches and history).
- **Pull:** Fetching changes from a remote and merging them into your local branch.
- **Push:** Sending your commits from your local branch to the remote repository.

We'll explain each as we go. Now, let's create our first Git repository.

Creating a Repository and Making Your First Commit

To start tracking a project with Git, navigate into your project folder and initialize a repo:

```
$ mkdir demo-project                # create a new project directory
$ cd demo-project
$ git init
```

```
Initialized empty Git repository in /.../demo-project/.git/
```

The `git init` command creates a new `.git` directory in your project – this is where Git stores the commits and metadata (the **.git directory** is essentially the Git repository database [8]). If you ever want to "un-Git" a folder, deleting the `.git` folder will do so (but you'd lose version history, so be careful).

After `git init`, nothing is tracked yet. Let's create a file and save our first version:

```
$ echo "<h1>Hello World</h1>" > index.html   # create a sample file
$ git status
```

```
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    index.html
```

Git reports that `index.html` is *untracked* – meaning it sees a new file that isn't in version control yet [9]. To start tracking it, we add it to the **staging area** and commit:

```
$ git add index.html
$ git commit -m "Add homepage"
```

```
[main (root-commit) 1a2b3c4] Add homepage
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```

The `git add` command stages the file (tells Git we want to include it in the next commit). The `git commit` then creates a new commit with the staged changes. We used the `-m "Add homepage"` option to provide a commit message directly. A good commit message is important – here we used a short message describing the change. Now the file is under version control.

Run `git status` again and you'll see the working directory is clean (no uncommitted changes):

```
$ git status
```

```
On branch main
nothing to commit, working tree clean
```

Congratulations, you've made your first Git commit!

Each commit in Git has a unique ID (a hash like `1a2b3c4` shown above) and contains the author name, timestamp, a message, and a snapshot of changes. You can view the history of commits with `git log`.

Viewing Commit History

To see the commit history, use the `git log` command:

```
$ git log
```

```
commit 1a2b3c4d5e6f7890... (HEAD -> main)
Author: Your Name <you@example.com>
Date:   Tue Oct 14 17:00:00 2025 +0300

    Add homepage
```

By default, `git log` lists commits in reverse chronological order (most recent first) [10] . Each entry shows the commit's 40-character SHA-1 ID (abbreviated here), the author, date, and the message [10] . In our case, there's just one commit. As you add more, they will all appear here.

Reading git log: The commit ID (SHA-1) is a unique key for that snapshot. `HEAD -> main` indicates this commit is the tip of the `main` branch (and HEAD is Git's pointer to your current branch's latest commit). You'll also see branch names or tags in these logs as you create them.

For a simpler log, try `git log --oneline --graph --all`. This shows each commit as one line, and `--graph --all` draws a text-based branch/merge graph:

```
$ git log --oneline --graph --all
```

```
* 1a2b3c4 (HEAD -> main) Add homepage
```

This isn't very interesting yet (no branches or merges to show), but as your history grows, this view is super useful to visualize your project structure.

Where is this history stored? In the `.git` folder, Git stores all your commits (as objects). The `.git` directory is essentially the repository's database. It contains subfolders like `objects` (the commits and file contents), `refs` (pointers to branch heads, tags, etc.), and other metadata. You typically don't need to touch anything inside `.git` manually – but it's good to know it's just files. In fact, the entire state of the repo is contained in `.git`. If you copy that folder to someone else, they have the whole repository [11]. Conversely, if you delete it, you "unversion" your project (which is usually not what you want unless starting over).

Git vs Other Systems (SVN, etc.)

Before moving on, to answer a common question: **How is Git different from SVN or other VCS?** The main difference is Git's distributed nature. In SVN (Subversion), there's a single central repository; Git is distributed (everyone has a repository). This leads to different workflows. With Git, you commit to your **local** repo frequently and only push to a remote when ready, whereas with SVN each commit hits the central server [7]. Git also excels at branching and merging – creating branches is very fast and merging is a core part of the workflow, whereas older systems sometimes discouraged branching due to performance or complexity. This means Git encourages developers to create separate branches for features/bugfixes and merge them, enabling concurrent development streams. We'll see more on branches soon.

(Fun fact: Git was created by Linus Torvalds in 2005 to manage the Linux kernel development, after dissatisfaction with other tools. Its speed and robustness come from clever design – it stores data as snapshots (like diffs) and uses cryptographic hashes to ensure integrity.)

Making Changes: Staging and Committing

Let's continue with our `demo-project`. Suppose we edit the `index.html` file to add a paragraph, and create a new file `about.html`. Our goal is to commit these changes.

After editing or adding files, it's good practice to run `git status` to see what's changed:

```
$ echo "<p>Welcome to my site.</p>" >> index.html
$ echo "<h1>About</h1>" > about.html
$ git status
```

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  modified:   index.html
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  about.html
```

Git shows that `index.html` was modified (tracked file changed) and `about.html` is untracked (new). To prepare these for commit, we add them to the staging area:

```
$ git add index.html about.html
$ git status
```

```
Changes to be committed:
  new file:   about.html
  modified:   index.html
```

Now both changes are staged. If we realize we accidentally staged something we don't want to commit yet, we can "unstage" it with `git restore --staged <file>` (or the older `git reset HEAD <file>` command). But in this case, it's fine.

Let's commit the changes:

```
$ git commit -m "Add About page and update homepage"
```

```
[main 9d8e7f6] Add About page and update homepage
 2 files changed, 2 insertions(+)
```

Now our history has two commits. Run `git log --oneline` to verify:

```
$ git log --oneline
```

```
9d8e7f6 (HEAD -> main) Add About page and update homepage
1a2b3c4 Add homepage
```

Each commit builds on the previous. Git has stored the differences (in this case, one new file and one modified file) in the commit. If needed, you could retrieve any version of those files from these commits.

Checking Your Commit History

Aside from the command line `git log`, if your repository is on GitHub or another hosting service, you can view commit history on the website as well. On GitHub, for example, each repository's page has a **Commits** section that lists commits in reverse order. Each commit entry on GitHub shows the author, date, and message, and you can click it to see the changes (diff) introduced by that commit.

Locally, you can also use GUI tools or IDE integrations to visualize history, but mastering the basic `git log` output is very useful. You can add options to `git log` to filter or format the output. For

instance, `git log -p` will show the *patch* (diff) for each commit, and `git log --stat` gives a summary of changes (files changed and line counts). Explore these as you become more comfortable. The key idea is that Git keeps a **linked list of commits** (each commit points to a parent commit) forming the project timeline. Tools like `git log` traverse this list to display history.

For example, `git log -2 -p` would show the last two commits with their diffs:

```
$ git log -2 -p
```

(Output would display the commit messages plus the code changes in context.)

You might not need detailed logs now, but it's good to know how to dig into history when debugging ("when was this line changed, and why?").

Working with Remotes (GitHub) – Clone, Push, and Pull

Right now, our `demo-project` repository is only local. To collaborate or back it up in the cloud, we can use a **remote repository**. Let's use GitHub as the remote. There are two common ways to get a repo on GitHub: create it on GitHub and **clone** it, or create it locally (we did) and then **push** it to GitHub.

Cloning a Repository: Cloning is copying an existing repo from a server to your computer. If a project already exists on GitHub (or another Git server), you'd do:

```
$ git clone https://github.com/username/project.git
```

This makes a local copy of `project` in a folder named `project`, including all its history and branches. For example, to clone our `demo-project` (after we push it to GitHub), we'd run something like `git clone https://github.com/yourusername/demo-project.git`.

Since we started locally, we need to push to a new GitHub repo. On GitHub, create a new empty repository (via **New repository** button). Let's call it "demo-project" as well. Don't initialize it with a README (since we already have one commit locally). GitHub will show instructions to add a remote. They will look like:

```
$ git remote add origin https://github.com/yourusername/demo-project.git
$ git branch -M main
$ git push -u origin main
```

Here's what these do:

- `git remote add origin <URL>` links our local repo to a remote repository at `<URL>` and names it "origin" (the conventional default name for "the main remote server"). Now Git knows about the GitHub repository.

- `git branch -M main` ensures our local branch is named `main` (GitHub's default). If your branch was already named `main`, this is redundant, but no harm.
- `git push -u origin main` pushes the `main` branch to the `origin` remote. The `-u` (upstream) flag sets origin/main as the default upstream for our local main, which means future `git pull` or `git push` commands will know which remote and branch to use by default.

Go ahead and run those (with your repo URL):

```
$ git remote add origin https://github.com/yourusername/demo-project.git
$ git push -u origin main
```

After entering your credentials (if prompted), your commits will be uploaded. On GitHub, refresh the repo page – you should see your files and commit history.

git push: The push command sends commits from your local branch to the remote branch. If someone else had pushed different commits to origin in the meantime, Git would prevent the push until you update (to avoid overwriting others' work). Normally, you will push after doing some commits locally to share your changes.

Now that we have a remote, let's simulate how to bring others' changes into your local repo.

git pull vs git fetch: Suppose a collaborator pushes new commits to GitHub. To update your local repo, you use `git pull`. This command actually does two things: it **fetches** new commits from the remote, then **merges** them into your current branch [12]. `git fetch` by itself just downloads the new commits and updates your remote tracking branches (e.g. updates `origin/main` reference) but doesn't merge into your branch. Think of `git fetch` as "check for updates" and `git pull` as "get updates and integrate them now." [13] [14].

If you want to review changes before merging, you can do `git fetch`, then look at the commits (e.g. `git log origin/main` to see new commits on the remote). Once satisfied, you could run `git merge origin/main` to merge them. `git pull` automates these steps (fetch + merge) in one go.

In practice for beginners, `git pull` is convenient to stay up-to-date. Just be aware it might auto-merge and possibly produce a **merge commit** if your local work and remote work diverged. If there are **merge conflicts** (when the same lines were edited differently), Git will notify you and mark the conflicts in the files – you'll then manually edit to resolve, then commit the merge. Don't panic if this happens; we'll touch on conflict resolution later.

Here's how to fetch and pull:

```
$ git fetch origin         # fetch updates from remote
$ git status
```

```
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
```

Git is telling us the local main is behind the remote. We can incorporate the changes:

```
$ git pull origin main
```

```
Updating 9d8e7f6..abcd123
Fast-forward
 contact.html | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 contact.html
```

Now our main branch is up-to-date with the remote. In this case it was a *fast-forward* (no divergent work, just advanced the branch pointer). If it had to merge, you'd see a merge commit message.

In summary: - `git fetch`: Checks for changes on the remote, but doesn't modify your working code. Good for a safe look. It updates remote tracking branches. - `git pull`: Fetches and immediately merges into your current branch. Convenient but if you have local changes, it could create a merge commit or conflict. Use it when you're ready to integrate remote changes into your work [15] [16].

Most of the time, you will use `git pull`. If you want more control, use fetch+merge (or fetch + `git rebase`, an advanced technique). It's a matter of preference and workflow. For a beginner, just remember to **pull before you push** if others might have pushed changes while you were working – this prevents you from accidentally pushing to an out-of-date base.

Branching: Creating and Managing Branches

Imagine you want to add a new feature to your project, but you don't want to disturb the stable code on the main branch. You can create a **branch** to isolate your changes. Branches are one of Git's most powerful features for collaboration and parallel development. A branch in Git is essentially a pointer to a commit – a line of development that moves as you add commits [17].

By convention, the main line is the **main** branch (in older projects, it might be called **master**). You can branch off main to develop a feature or fix, then merge it back in when ready. Teams often require work to be done in separate branches and merged via pull requests (we'll cover that soon).

To list branches:

```
$ git branch
```

```
* main
```

The `*` indicates the current branch. Let's create a new branch for a feature:

```
$ git branch new-feature
$ git branch
```

```
* main
  new-feature
```

We made a branch named "new-feature", which at creation is an exact copy of main (points to same commit). We still remain on main until we **checkout** the new branch:

```
$ git checkout new-feature
```

```
Switched to branch 'new-feature'
```

Now `git branch` would show `* new-feature` as current. We could also create and switch in one command with `git checkout -b new-feature`. In newer Git versions, you can use `git switch` as well (`git switch -c new-feature` to create, or `git switch main` to switch back).

Now on "new-feature" branch, any commits we make will diverge from main. For example, let's add a file:

```
$ echo "<script>alert('Hi');</script>" > script.js
$ git add script.js
$ git commit -m "Add greeting script"
```

```
[new-feature 1122334] Add greeting script
 1 file changed, 1 insertion(+)
 create mode 100644 script.js
```

This commit is on the **new-feature** branch only. The main branch still doesn't have `script.js`. We can switch back to main and verify:

```
$ git checkout main
$ ls
```

```
about.html   index.html   (no script.js here)
```

Git automatically keeps track of the branch HEAD pointers. Our branches have diverged: main is still at "Add About page..." commit, while new-feature has an extra commit on top. We can visualize this with `git log --oneline --graph --all`:

```
$ git log --oneline --graph --all
```

```
* 1122334 (new-feature) Add greeting script
| * 9d8e7f6 (HEAD -> main, origin/main) Add About page and update homepage
|/
* 1a2b3c4 Add homepage
```

This ASCII art shows new-feature branching off after commit 9d8e7f6. Main and new-feature now have different HEADs.

Merging Branches

When your feature is done, you'll want to merge it back into main (so main includes the new changes). One way is using Git on the command line:

First, ensure you have the latest main (if collaborating, `git pull` on main). Then:

```
$ git checkout main
$ git merge new-feature
```

```
Updating 9d8e7f6..1122334
Fast-forward
 script.js | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 script.js
```

Git noticed main can be "fast-forwarded" to include new-feature's commit (because main had no new commits of its own since the branch). It simply moved main's pointer forward. Now main contains the script file as well. If main had progressed and both branches had unique commits, Git would do a real merge, creating a new merge commit that ties the histories together. In either case, after merging, it's good to test your code and then possibly delete the feature branch (to keep your list of branches tidy).

Deleting a branch: Once merged, you can delete the branch with:

```
$ git branch -d new-feature
```

This deletes the pointer, not the commits (they're now part of main, so safe). If you try to delete an unmerged branch, Git will warn or prevent it by default. Use `-D` (capital D) to force delete if you really know what you're doing (careful – you could lose commits if they weren't merged elsewhere).

In our case, new-feature was merged (fast-forward), so it deletes easily. This was a local branch. If the branch was also on GitHub (remote branch), you can delete it by `git push origin --delete new-feature`.

Example – Deleting a branch:

```
$ git branch -d new-feature
```

```
Deleted branch new-feature (was 1122334).
```

If the branch wasn't merged, `git branch -d` would refuse (you'd use `-D` to override, but that's risky). This safety helps prevent losing work.

So, the process for branching in Git is: *create branch -> work -> commit -> merge -> delete branch.* Branching is very cheap in Git – it's essentially just a name for a commit – so you should feel free to create branches for each task or experiment. In team workflows, this is standard: never work directly on main for features, instead use feature branches (often one per issue/feature), then merge via pull requests.

Git branching example: Branches are pointers to commits. Here, `master` (main) pointed to commit C originally. We created a new branch pointing to C as well. After some commits, `master` still points to C, while the feature branch advanced to D and E. When we merge back, `master` will point to E. Tags (like v1.0) are static pointers to specific commits [18] [19]. This diagram shows a simple branch (feature) diverging and then merging back into the main branch.

In a real project, you might have many branches. Common branches might include a `develop` branch (in GitFlow workflow), or release branches, but let's keep to basics.

Working on Multiple Branches and Stashing

Sometimes you might be partway through work on one branch when something urgent comes up on another branch. If you need to switch branches but have uncommitted changes that you're not ready to commit, you can use **git stash**. Stashing takes your working changes and saves them on a stack, then cleans your working directory to match the HEAD commit [20] [21]. Later, you can apply those changes back.

For example, if you're in the middle of editing files on a branch and need to quickly switch to `main` to hotfix something, you could run:

```
$ git stash            # save changes to stash
Switched to branch 'main'
... do urgent fix ...
$ git commit -m "Hotfix"
$ git push origin main
$ git checkout feature-branch
$ git stash pop        # reapply stashed changes
```

When you run `git stash`, Git will **shelve** (save) the modifications (both staged and unstaged) and revert your working directory to a clean state [21]. It will print a stash name like `stash@{0}` saved. You can list stashes with `git stash list`, which shows something like:

```
$ git stash list
```

```
stash@{0}: On feature-branch: WIP on feature-branch: 1122334 Add new feature
```

Each stash is like a little commit holding your changes. `git stash pop` applies the most recent stash to your working directory and removes it from the stash list [22] [23]. Alternatively, `git stash apply` would apply it but keep it in stash (in case you want to use it again or apply to another branch).

Use stash as a quick way to "clean your desk" and come back later. It's very handy to avoid making a messy commit just to switch context. You can even stash multiple times (each stash gets an index). Use `git stash drop stash@{n}` to discard a specific stash if you don't need it. To be safe, remember that stash is local – it doesn't sync to remotes.

For beginners, just remember: *commit or stash* before switching branches if you don't want to carry over changes.

To illustrate stash quickly:

```
$ git status
(modified files not staged)
$ git stash save "work in progress"
Saved working directory and index state WIP on feature-branch: abcdef0 Work in
progress
$ git status
working tree clean (everything stashed)
... later ...
$ git stash pop
```

This will show your previously stashed changes back in your working copy and remove that stash from the list [24] [25].

Collaborating with GitHub: Forks, Pull Requests, and Code Reviews

So far, we've worked in a single repository with one collaborator scenario. Often, especially in open-source, you won't have direct push access to someone else's repository. Instead, you **fork** the repository – which is like making your own copy on GitHub – and then contribute via a **pull request**. Let's demystify these concepts:

- **Fork:** A fork is your own copy of someone else's repo on your GitHub account [26]. For example, you want to contribute to `octocat/Spoon-Knife` repository. You click "Fork" on GitHub, and now under yourusername/Spoon-Knife you have the same content (GitHub does this efficiently without actual duplication of all data). You clone *your fork* to your local system, make a new branch, commit changes, and push to *your fork*. Then you open a **pull request** from your fork's branch to the original repository's branch (usually master/main). This workflow is common for open source: it's called the **Fork & Pull Model** [27].

- **Pull Request (PR):** A pull request is a proposal to merge commits from one branch into another (could be in the same repo or from a fork). On GitHub, you can open a PR and other developers/maintainers can review your changes, discuss them, and ultimately merge them if approved [28] [29]. It's an essential collaboration tool for code review and quality control. A PR doesn't automatically merge code; it provides a way to review and track the integration of the feature. Maintainers can request changes or run tests (often automated via CI) before accepting the PR.

Opening a pull request on GitHub is easy once your branch is pushed to GitHub. There will often be a "Compare & pull request" button if you recently pushed a new branch. In the PR, you write a description of what your change does. This is where a good **README** and **CONTRIBUTING** guide in the project will tell you what information to include (like what problem it solves, screenshots if UI, etc.). After creating the PR, it enters the review phase.

Code Review and Approval: Teams enforce code quality by requiring PRs to be reviewed and approved by other developers before merging [30] [31]. On GitHub, you can set branch protection rules that require at least one approving review and passing status checks (like tests) before the **Merge** button is enabled [30]. This ensures no code goes into main without proper oversight. Code reviews help catch bugs, improve code style, and share knowledge among the team. As a beginner, don't be intimidated – constructive feedback is there to help you and the project.

During a PR review, reviewers might leave comments on specific lines or overall suggestions. You can address these by making new commits on your PR branch (push them, and the PR updates automatically). It's common to prefix commit messages with "Fix review feedback:" or similar, or to amend commits and force-push if the project prefers a clean history (but ask first – some projects discourage force-pushing a PR). Generally, incorporate feedback, then comment that you've done so. Once reviewers are happy, they will **approve** the PR.

Merging a PR: Once approved (and all automated checks pass), a maintainer (or you, if you have permission) will merge the PR. On GitHub's PR page, they might use a **Squash Merge** (to squash all your commits into one) or a regular merge commit. The result is that your changes become part of the target

branch (e.g., main) [32]. The PR is then closed. Often, the source branch is deleted to keep things tidy (GitHub offers a button for this after merge).

If you forked a repo, your fork's main will not automatically update when the original does. You should occasionally **update your fork** by pulling from "upstream" (the original repo). This involves adding the original repo as a remote named `upstream` in your local clone (`git remote add upstream https://github.com/original/repo.git`), then doing `git fetch upstream` and merging/updating as needed [33] [34]. This way your fork doesn't fall too far behind.

GitHub Issues and Discussions

Collaboration isn't just about code. On GitHub, **Issues** are used to track tasks, bug reports, and enhancement requests. It's like a to-do or ticket system integrated with the repo. Issues can be assigned, labeled, linked to pull requests, and closed when resolved. They provide a space for discussion around a specific problem or feature. **Discussions** (a newer feature) provide an open forum for more free-form conversation (Q&A, ideas, etc.) separate from the stricter issue/PR format. Typically, if you have a question or broad topic, you might start a Discussion; if you found a bug or want a feature, you open an Issue. Maintainers might have their own guidelines on which to use when [35].

For example, on a project, you might see an Issue titled "Bug: crash when clicking X" where people discuss and eventually someone submits a PR to fix it, linking the issue (if you put "Closes #issueNumber" in a PR description, GitHub will auto-close the issue on merge). Discussions could be things like "What's the roadmap for v2.0?" or "How do I use this feature?" – more general or open-ended. They're great for community engagement without cluttering the issue tracker.

If you're a beginner contributing to a project, a good approach is: search the Issues for something to work on (many projects label easy issues with "good first issue" or "help wanted" [36]). Comment on the issue to express interest in working on it (to avoid duplication of effort). Then follow the fork/branch workflow to make your change.

GitHub Projects and Project Boards

GitHub has a feature called **Projects** (particularly the newer GitHub Projects which are more like spreadsheets for issues/PRs, and the classic Projects which are Kanban boards). These are tools for project management and planning. A GitHub Project board can track issues or PRs through stages (To Do, In Progress, Done, etc.) similar to Trello or Jira, but tied to your repo's items [37]. Teams use Projects to organize work into milestones, sprints, or feature sets. As a beginner, you might not create these yourself, but it's good to know what they are. If you see a "Projects" tab on a repository, that's what it is – a way to see progress and group tasks. Using project boards helps teams visualize the state of work and who's doing what [37].

For example, a **GitHub Project** might list all open issues and pull requests in a table or board, with fields like status, priority, assignee, etc. As issues get worked on, they move to "In Progress" and then to "Done." This doesn't affect Git directly; it's a management layer. So it's not a Git feature, but part of GitHub. Still, knowing about it helps you navigate team workflows. You might be asked to move your issue to done once merged, etc.

Enforcing Quality: Branch Protection and CI

Earlier we mentioned code reviews as a way to enforce quality. GitHub also allows **branch protection rules** on important branches (like `main`) – for instance, you can make it read-only for everyone except through approved pull requests. This means nobody can push to main directly; they *must* create a PR and have it reviewed (possibly by specific people or at least one other person) [30]. You can also require that all checks pass. Checks usually refer to Continuous Integration (CI) tests – like GitHub Actions or Travis CI running your test suite or linter on the PR. If tests fail, the PR can't be merged until fixed. These practices ensure a higher level of code quality and prevent mistakes (like "Oops, I directly pushed broken code to main"). Many teams also use **CODEOWNERS** files to automatically request specific reviewers (like, UI changes must be reviewed by the "frontend team") [38].

From a beginner perspective: be prepared that when you contribute or work on a team, you will create a branch, push it, open a PR, and then wait for CI to pass and colleagues to review. This is normal. It might feel formal, but it catches a lot of issues before they hit main. For example, imagine your PR inadvertently breaks a unit test – the CI will flag it, and you can fix it before merge, rather than breaking main and then others pulling broken code.

In short, teams use a combination of **GitHub features** to enforce quality: required reviews, required status checks, protected branches, and sometimes automated bots for linting or formatting. As a newcomer, focus on writing clear commits and PR descriptions, and be responsive to review feedback. Learn from it – code review is an excellent learning opportunity.

A Simple Project Workflow: Building a Website and Using GitHub

Let's put it all together with a mini-project. We'll create a simple website (with HTML, CSS, JS files) and walk through a complete GitHub workflow: initializing a repo, creating branches for features, making pull requests, merging them, and tagging a release.

Scenario: You're making a personal website with a homepage, an about page, etc. You want to use GitHub to manage it. Perhaps you'll even host it on GitHub Pages (optional). You'll follow a feature-branch workflow to add features.

1. **Initialize the repository:** Start a new project folder `my-website` and `git init` it (or create the repo on GitHub and clone it). Add a basic README and the initial index.html:

   ```
   $ mkdir my-website && cd my-website
   $ git init
   $ echo "# My Website" > README.md
   $ echo "<!DOCTYPE html><html><head><title>My Site</title></
   head><body><h1>Home</h1></body></html>" > index.html
   $ git add .
   $ git commit -m "Initial website with homepage"
   ```

   Now `main` has one commit (initial site).

2. **Create a GitHub repo and push:** Go to GitHub, create a repository `my-username/my-website`. Then connect your local repo as remote and push:

```
$ git remote add origin https://github.com/yourusername/my-website.git
$ git push -u origin main
```

Your code is now on GitHub. :rocket:

3. **Create a feature branch:** Let's add a new feature – a CSS stylesheet to style the site. Create and switch to a branch:

```
$ git checkout -b add-styles
Switched to a new branch 'add-styles'
```

Now create a CSS file and link it in HTML:

```
$ echo "body { font-family: sans-serif; background: #fafafa; }" > style.css
$ sed -i 's#</head>#<link rel="stylesheet" href="style.css"></head>#'
index.html   # (on Linux/macOS; edit head tag to include stylesheet)
$ git add index.html style.css
$ git commit -m "Add basic stylesheet for site style"
```

(On Windows `cmd`, the `sed -i` might not work; you could open index.html in a text editor and add `<link rel="stylesheet" href="style.css">` before `</head>`.)

Now branch `add-styles` has this commit, separate from main.

4. **Push the feature branch to GitHub:** We push this branch so we can make a PR:

```
$ git push -u origin add-styles
```

On GitHub, if you go to your repo, you'll see the branch. GitHub will likely prompt "Compare & pull request". Click that or go to the Pull Requests tab and start a new PR, choosing base = main, compare = add-styles.

5. **Open a Pull Request:** Title it "Add basic stylesheet for site" and write a description. Maybe say "This adds a stylesheet to improve background and font. No breaking changes." Submit the PR.

Now imagine you have a collaborator who reviews it. They might comment "Could you also include a screenshot?" or find a small issue. You address any feedback by making more commits on the `add-styles` branch and pushing them – the PR updates automatically. For example, if they said "Add a

`.container { max-width: 800px; margin: auto; }` for better layout", you could commit that change to style.css:

```
$ echo ".container { max-width: 800px; margin: 0 auto; }" >> style.css
$ git commit -am "Add container class for layout"
$ git push
```

(The `-am` flag stages and commits all modified tracked files in one go – use with care, but here style.css was already tracked.)

The PR now shows two commits. Once the reviewer is satisfied (and CI tests, if any, pass), they approve the PR.

6. **Merge the Pull Request:** On GitHub, click "Merge pull request". Usually you'd do "Squash and merge" or just "Merge". Let's say we squash merge to keep one commit on main. After merging, main will have a new commit combining the changes from that branch. The PR closes.

   Pull the changes back to your local main:

   ```
   $ git checkout main
   $ git pull origin main
   ```

   Now your local main has the stylesheet too. The feature branch isn't needed, you can delete it:

   ```
   $ git branch -d add-styles
   ```

   (GitHub also offers to delete the branch on merge, which you can do from the PR page.)

7. **Create another branch for a new feature:** Let's add a contact page with a simple JavaScript form validation.

   ```
   $ git checkout -b add-contact-page
   $ echo "<!DOCTYPE html><html><head><title>Contact</title></head><body>
   <h1>Contact Us</h1><form onsubmit='return checkForm()'>
     Name: <input id='name'><br>
     <button type='submit'>Send</button>
   </form>
   <script src='script.js'></script>
   </body></html>" > contact.html
   $ echo "function checkForm() {
     var name = document.getElementById('name').value;
     if(!name) { alert('Name is required'); return false; }
   ```

```
  return true;
}" > script.js
$ git add contact.html script.js
$ git commit -m "Add contact page with simple form and validation script"
```

Also don't forget to add a link to Contact from index and about pages (assuming about.html exists). If you have an about page, edit it similarly. For brevity, skip details. Commit any such changes.

Push this branch:

```
$ git push -u origin add-contact-page
```

Open a PR "Add contact page". Collaborator reviews, perhaps they suggest a small change (like adding an email field – you do so in a new commit). After revisions, merge the PR. Pull updates to main locally.

8. **Tagging a Release:** Now that we've added a couple of features, let's create a release tag. Say this is version 1.0 of the site. In Git, you create a tag for a specific commit (usually on main when you're ready to release):

   ```
   $ git checkout main
   $ git tag -a v1.0 -m "Release version 1.0 - basic website with contact
   page"
   $ git push origin v1.0
   ```

   The `-a` makes an *annotated* tag (with a message and metadata). Now on GitHub, you'll see this tag under Releases. You can go to the Releases section and even add release notes if desired (GitHub allows you to convert a tag into a formal "Release" with a title and description, which is mainly semantic sugar).

   What is a tag? A tag is like a branch that doesn't move – a label for a specific commit, often used for marking release points (v1.0, v1.1, etc.) [39]. Unlike branches, tags don't change; they permanently point to that commit. This is useful for distribution (you might attach compiled assets to a GitHub Release corresponding to a tag).

   At any time, someone can `git checkout v1.0` to get the code at that release.

9. **Creating a GitHub Release (optional):** On GitHub, if you navigate to **Releases** -> **Draft a new release**, you can choose the tag `v1.0`, give it a name "v1.0", and write release notes (e.g., "First release of the website"). This isn't required but helps communicate changes to users.

Now you've gone through a full cycle: you created a repo, made feature branches, collaborated via PRs, merged into main, and marked a release. This workflow (feature branches & PRs) is essentially how real companies and open-source projects operate on GitHub.

Some projects use variations: **GitFlow** workflow has separate `develop` and `main` (production) branches and uses release branches and hotfix branches. **Trunk-based** workflow might commit small increments directly to main (trunk) with feature flags to turn off incomplete features instead of long-lived branches [40]. Many teams do something in between – short-lived feature branches merged continuously (which is essentially trunk-based using PRs). There's also the **forking workflow** we discussed for external contributors (common in open source). Real companies choose what fits their size and release cadence [41]. For learning purposes, the feature branch + PR model we demonstrated is an excellent default.

(In case you're curious: GitFlow involves multiple long-lived branches: `main` *for releases,* `develop` *for integration of features, and supporting branches for features, releases, and hotfixes. It's useful for projects with scheduled releases. Trunk-based development encourages merging small, frequent updates into* `main` *(trunk) and deploying continuously [40]. Many modern teams favor trunk-based for rapid iteration, often with automation to deploy every commit that passes tests. But all these are built on the same Git concepts we covered – branches and merges.)*

Git Command Cheat Sheet

Here's a quick reference of common Git commands and what they do:

- `git init` – Initialize a new Git repository in the current directory (creates a `.git` folder).
- `git clone <url>` – Clone (download) a repository from `<url>` (e.g., GitHub) to your local machine [42].
- `git status` – Show the status of changes: which files are untracked, modified, or staged for commit.
- `git add <file>` – Stage a file's changes for commit. (`git add .` stages all changes in the current directory).
- `git commit -m "message"` – Commit the staged changes with a message. (Omit `-m` to open an editor for a multi-line message).
- `git log` – Show commit history. Add `--oneline`, `--graph`, etc., for condensed or graphical log [10].
- `git branch` – List branches. (`git branch <name>` creates a new branch; use `-d <name>` to delete one).
- `git checkout <branch>` – Switch to an existing branch (update working files to that branch's state). Use `-b <name>` to create and switch to a new branch immediately.
- `git switch <branch>` – (Newer alternative to checkout for switching branches; `git switch -c <name>` to create).
- `git merge <branch>` – Merge the specified branch's history into the current branch. Creates a merge commit if needed.
- `git push` – Upload commits from your local branch to the remote. Typically `git push origin <branch>`. The first push of a new branch needs `-u` to set upstream.
- `git pull` – Fetch from the remote and then merge into current branch (shortcut for `git fetch` + `git merge`) [15]. Usually `git pull origin <branch>`.
- `git fetch` – Fetch updates from the remote, but don't merge. After fetching, you can inspect or merge manually.
- `git remote -v` – List the configured remotes (like origin) and their URLs.
- `git remote add <name> <url>` – Add a new remote. E.g., adding an upstream for a fork.

- `git stash` – Stash (save) your uncommitted changes and clean your working directory [21].
- `git stash pop` – Reapply the most recent stashed changes to the working directory and remove that stash [43].
- `git stash list` – View stashes saved.
- `git restore <file>` – Discard changes in the working directory for `<file>` (dangerous if not committed anywhere else). Essentially, `git restore <file>` reverts the file to HEAD's version – use this to **safely discard unwanted changes** in a file [44].
- `git reset HEAD~1` – Undo the last commit, keeping changes in the working directory. This moves HEAD backward (use when you made a wrong commit locally and want to re-do it) [45].
- `git revert <commit>` – Create a new commit that reverses the changes of the specified commit [46]. Use this to "undo" a commit in shared history without rewriting history.
- `git tag -a <tagname> -m "msg"` – Create an annotated tag on the current commit (for marking a release) and add a message [47]. Push tags with `git push origin --tags` (or push a single tag by name).
- `git diff` – Show the differences between working directory and staging or between commits. (`git diff HEAD` shows uncommitted changes compared to last commit).
- `git blame <file>` – Show who last edited each line of `<file>` (useful for tracing history).
- `git restore --staged <file>` – Unstage a file (undo `git add` for it, but keep changes in working directory).
- `git config` – Configure settings. E.g., `git config --global core.editor nano` to set default editor, etc. (We used it for user.name and user.email).

Keep this cheat sheet handy. The more you practice, the more these become second nature.

Glossary of Key Git/GitHub Terms

- **Repository (Repo):** A collection of project files tracked by Git, along with the full history of commits. Can be local or on a remote server (like GitHub). Essentially the project's database of versions.
- **Version Control:** The management of changes to documents, code, or other collections of information over time. Allows multiple versions and collaboration with history. Git is a version control system.
- **Commit:** A snapshot of the repository at a point in time, with a message describing changes. Each commit has a unique ID (hash) [10]. Commits form a history (linked list).
- **Branch:** A named pointer to a commit, representing a line of development. The repository can have multiple branches (e.g., `main`, `dev`, feature branches). New commits normally happen on a branch, moving its pointer forward [48].
- **Main/Master:** The default primary branch of a repo. Often the stable integration branch that is deployed or released. (Note: Many projects now use "main" instead of "master" as the default name.)
- **Checkout/Switch:** The act of changing your working directory to reflect a different commit or branch. You "check out" a branch to work on it.
- **Merge:** Combining the changes from one branch into another. If branches diverged, a merge commit is created that ties them together. Fast-forward merges occur when the target branch has no new commits and just advances to the incoming commit [49].
- **Merge Conflict:** When Git can't automatically merge because the same lines were changed in both branches. Git marks the conflict in the files, and a human must resolve by editing the file to what it should be, then committing the resolution.

- **Remote:** A reference to a version of the repository on a network (usually another server). “origin” is the default name for the main remote (often pointing to GitHub). You push to and pull from remotes.
- **Pull Request (PR):** A feature of GitHub (and similar platforms) to notify others of changes you want to merge. It allows code review and discussion before merging the code [50] . Not a Git concept per se, but built on top of branches and merges.
- **Fork:** A personal copy of someone else’s repository on your GitHub account [51] . Used to propose changes when you don’t have direct access. You fork, then make a PR from your fork.
- **Clone:** A full copy of a repository (all branches and history) on your local machine [51] . You clone from a remote URL.
- **Staging Area (Index):** An intermediate area where you put changes (`git add`) before committing. It allows you to craft commits intentionally (e.g., include some changes but not others).
- **Working Directory (Working Tree):** The actual files on your filesystem that you edit. These reflect the last checked-out commit plus any changes you’re making. You `git add` from here and `git commit` to store a snapshot from here.
- **HEAD:** A reference to the current commit (typically the tip of the current branch) in your working directory. It’s essentially “what am I currently looking at?”. When you checkout a branch, HEAD points to that branch’s latest commit [52] .
- **SHA (Hash):** The 40-character identifier (often shown as short 7-character) for commits (and other Git objects). It’s generated via hashing the content and metadata. Used to uniquely identify commits.
- **Tag:** A human-friendly name given to a specific commit (often for releases). Tags are immutable markers (e.g., v1.0) [39] .
- **Fast-Forward:** A type of merge where the target branch has no new commits since it branched, so it can simply be moved up to the other branch’s commit, with no merge commit needed [49] .
- **Rebase:** (Advanced) Reapply commits on top of another base tip. It’s a way to rearrange commit history (often to make it linear) by moving a branch to start at a new position. Powerful but can be dangerous if used incorrectly on shared branches.
- **Upstream (Remote) Branch:** The remote branch that your local branch is tracking. After a `git push -u origin <branch>`, your branch knows “origin/<branch>” is its upstream. Then `git pull` and `git push` can omit specifying.
- **GitHub Actions:** The CI/CD platform integrated into GitHub. Not a Git concept, but if you see “checks” running on a PR, that’s likely GitHub Actions running tests or linters on your code.
- **.gitignore:** A file listing patterns of files for Git to ignore (not track) [53] . For example, compiled binaries, secrets, or OS-specific files. Important for keeping junk or sensitive data out of the repo.
- **Contributing Guidelines:** A document (CONTRIBUTING.md) some repos have to explain how to contribute – e.g., code style, PR process, branching strategy, etc. [54] . Always read it for a new project; it’s basically the project’s rules of engagement.
- **Issue:** On GitHub, a ticket for tracking bugs, enhancements, or tasks. It’s not part of Git, but closely integrated to reference commits/PRs. Issues help plan and discuss work.
- **Git LFS (Large File Storage):** An extension to Git for handling large files. It stores large files (binary assets, etc.) outside the normal Git objects, replacing them with pointers in your commits [55] . Useful for media or big data in repos to avoid huge repo size.

FAQ – Frequently Asked Questions

Q: Is GitHub the same as Git?
A: No – Git is the version control software that runs locally and manages your commits. GitHub is a web service (owned by Microsoft) that hosts Git repositories and adds collaboration features (pull requests, issue

tracking, etc.). You can use Git entirely without GitHub (for example, storing repos on your computer or a company server) [11]. GitHub just makes it easy to share and contribute. Think of GitHub as a social/collaborative layer over Git repos.

Q: What's the difference between *commit* and *push*?
A: A **commit** saves your changes to your *local repository*. It's a local operation – a commit doesn't automatically make it to GitHub or visible to others. A **push** sends committed changes to a **remote** repository (like the one on GitHub) [56]. For example, you might make 3 commits on your laptop, and then `git push` them all at once to GitHub. If you forget to push, your GitHub will not have your latest commits.

Q: When should I commit?
A: Commit often, whenever you reach a logical point or completed a task. Each commit should ideally contain a coherent set of changes (e.g., "Implement login API" or "Fix typo in README") with a descriptive message. Don't be afraid of making too many commits; you can always squash or rebase later if needed. Frequent commits make it easier to track progress and revert specific changes if a bug arises. Avoid committing huge chunks that mix unrelated changes – that makes it hard to review or rollback.

Q: How do I undo a mistake?
A: Git has many tools to undo mistakes: - To undo changes *before* committing (in working directory), you can checkout the file from HEAD: `git restore filename` (or older `git checkout -- filename`) to discard local changes [44]. Or if you staged something accidentally, use `git restore --staged filename` to unstage. - To undo the *last commit*, if it's not pushed yet: `git reset HEAD~1` will move HEAD back one commit, keeping the changes in your working directory so you can fix and recommit [57]. If you want to completely undo and throw away the changes, use `git reset --hard HEAD~1` (careful – this erases that commit and its changes). - To undo a commit that *was* pushed/shared, use `git revert <commit>` to make a new commit that reverses it [57]. This is safer for shared history because it doesn't rewrite history – it adds a record of undoing. - If you mess up a merge and haven't committed it yet, `git merge --abort` will stop the merge and return to pre-merge state. - In all cases, Git is pretty forgiving as long as you haven't deleted refs or used `--hard` incorrectly. The `git reflog` command can help you recover commits that were "lost" (like if you reset something, the commit still exists in reflog for a while) [58].

And there's the famous site "Oh Shit, Git!?" with solutions for common mistakes. So don't panic – most errors can be fixed.

Q: What does "fast-forward" mean when I merge or pull?
A: A *fast-forward* merge means Git simply moved the branch pointer forward because there were no divergent changes. For example, your main is at commit A, you made a branch and added commit B, and main stayed at A. Merging branch into main will just point main to B – no separate merge commit needed [49]. It's like main "fast-forwarded" in time. If both main and your branch had new commits, then a true merge commit is needed to combine them.

Q: Should I avoid committing binary or large files?
A: Yes, generally. Git works best with text files (code, markdown, etc.) that diff well. Large binary files (images, videos, datasets) can bloat the repo and Git cannot diff or merge them well. If you have to include large files, use Git LFS [55] or store them elsewhere. Also, put patterns for generated binaries in your

`.gitignore` so you don't accidentally commit them [59] [60]. For example, add lines for `*.exe`, `*.dll`, `node_modules/`, etc., as appropriate. That keeps the repo lean and clean.

Q: How can I collaborate without stepping on toes?
A: Use branches and pull requests instead of everyone hacking on main. Communicate with your team – e.g., use GitHub Issues to coordinate who's working on what. If you use feature branches, there's less risk of conflict. If two people edit the same file, Git will merge if edits are to different lines, but will conflict if same lines are edited. In case of conflict, communicate to decide the correct resolution, then fix the file and commit. Also, regularly pulling the latest main into your branch can minimize surprise conflicts later (some teams even do periodic `git merge main` into feature branches or rebase the feature branch onto main to stay updated).

Q: What are some common beginner mistakes with Git?
A: A few to watch out for: - Forgetting to **pull** before starting new work, leading to divergence or needing to merge unexpectedly. Tip: always `git pull` the latest main before creating a new feature branch. - Committing secrets or personal data (API keys, passwords) into the repo. **Avoid this!** Use environment config files (and add to `.gitignore`) for secrets. Once something is in Git history, removing it is tricky (requires rewriting history with tools like BFG). Always double-check you're not adding credentials. - Not using a .gitignore and thus committing OS files (like `.DS_Store` on macOS) or dependencies (like entire `node_modules` directories) – which clutter the repo [61]. Use templates (GitHub offers gitignore templates for many languages when creating a repo). - Making giant commits that mix many changes. It's better to break them into logical commits. This helps in code review and if you need to revert something later. - Using `git push --force` on a shared branch (like main) – this can rewrite history and screw up colleagues' clones [62]. Force-pushing your personal feature branch is okay if you know what you're doing (like after a rebase), but never force push main or any branch others use, unless absolutely necessary and coordinated. - Forgetting to switch branches – e.g., you intend to work on a new feature but stay on main and commit there. If that happens, you can recover by creating a branch at that commit (`git branch feature-x HEAD~n`) and resetting main back, etc. But try to be mindful: always check `git status` which shows your current branch. - Being afraid of merge conflicts and trying to avoid them by not collaborating – conflicts aren't that bad; resolve them calmly. Better to commit frequently and communicate than to hold back changes to "avoid conflicts". In fact, long-lived branches tend to cause more painful conflicts. Integrate often (that's why continuous integration workflows exist).

Q: How can I see what changed in a commit or pull request?
A: Use `git diff`. For a specific commit hash, `git diff <commit>^!` shows the changes in that commit (the `^!` means from its parent to itself). If you're in a PR on GitHub, the "Files changed" tab shows the diff of the PR's commits combined. Locally, if you want to see differences between branches: `git diff main..add-contact-page` would show what changes the feature branch has relative to main. Also `git log -p -2` shows the diff for the last 2 commits, etc. And GUI tools (like GitHub Desktop, or VS Code's Git panel) can show diffs easily as well.

Q: How do I update my fork with new changes from the original repo?
A: This is a common task if you fork an open source project. You should add the original repository as an `upstream` remote:

```
$ git remote add upstream https://github.com/original_owner/repo.git
$ git fetch upstream
$ git checkout main
$ git merge upstream/main   # or git rebase upstream/main
$ git push origin main
```

This fetches the latest from the original and merges it into your fork's main. Now your fork is up-to-date. Alternatively, you can use GitHub's web interface "Fetch upstream" button if available. But knowing the manual way is useful [33] [34].

Q: What is the reflog **and why is it useful?**
A: The reflog is a log of where HEAD (and refs) have been on your local repo. Even commits that are "orphaned" (no branch pointer) can be found for a while via reflog. If you accidentally reset or delete a branch, you can often recover the commit hash via `git reflog` and then recreate a branch at that hash [58]. It's like a safety net for local operations. Note that reflog is local; it doesn't exist on remote. But it's saved my skin multiple times when I thought I lost commits.

Q: Any tips for writing good commit messages?
A: Yes! A good commit message is clear and specific about *what* and *why*. Start with a short summary (50 characters or so), then if needed, a blank line and more details in the body. Use imperative mood (e.g., "Fix typo in login error message" or "Add unit tests for User model"). That's the convention – it reads like "This commit will ...". Explain the reasoning for complex changes in the body if it's not obvious. Avoid messages like "Update" or "Misc fixes" that don't tell what's inside. Future you and your teammates will thank you when looking through `git log` [63] [64].

Q: How can I practice Git effectively after this session?
A: The best way is by *using Git* in real scenarios: - Create some toy repositories and mess around – make commits, branches, merge them, introduce conflicts deliberately to resolve them. You can't break anything permanently locally, so experiment. - Use platforms like **GitHub** or **GitLab** to collaborate with friends on a small project. Even a simple recipe collection or a homework essay can be put in a repo to practice version control. - There's a great repository **firstcontributions/first-contributions** [65] which guides beginners through their first pull request on GitHub – highly recommended. It's essentially a sandbox to practice forking a repo, adding your name to a file, and making a PR. - Try out **GitHub Skills** interactive courses (available on GitHub) – they have a course for Introduction to GitHub, which uses a bot to teach you steps via issues and PRs. It's fun and informative [66] [67]. - Work on an open-source issue labeled "good first issue". Many large projects welcome beginners. This gives you real experience with forking, branching, PRs, and code reviews by others. - Lastly, read the free online **Pro Git book** (git-scm.com/book) – it covers a lot of what we did and more (like advanced branching, rebasing, cherry-picking, submodules) with examples [68]. You don't have to memorize it, but knowing what's possible is helpful.

Remember, Git has a learning curve – it's normal to find some concepts confusing at first. With regular practice, commands that initially felt arcane (`git reset HEAD~1`? `git rebase -i`?) will start to make sense. Don't be afraid to refer to documentation or use `git --help` for any command. Even experienced developers Google Git questions (and copy-paste helpful commands) – it's part of the process.

By practicing, you'll also learn to recover from mistakes confidently, which is a huge part of using Git effectively. Version control is a superpower for developers – it lets you experiment without fear because you can always revert or branch off. So embrace it in your projects, even if you're solo.

Happy coding and versioning! Git and GitHub will become trusty tools in your developer toolkit, enabling you to collaborate and manage projects with ease. As you grow more comfortable, you can explore advanced topics like interactive rebasing (to rewrite commit history), hooks (automating tasks on commit/push), continuous integration, and more. But with the foundation you've learned here, you're well on your way. Good luck and have fun creating with Git and GitHub!

7 13 10 8 69 53 70 71 46 72 20 21 73 74 35 37 75 30 31 57 48 58 44 55 76 77 40 78 79 80

---

1 2 3 4 5 6 Git - About Version Control
https://git-scm.com/book/ms/v2/Getting-Started-About-Version-Control

7 55 Git vs. SVN: What's the Difference, and Which is Better for Your Team? | Perforce Software
https://www.perforce.com/blog/vcs/git-vs-svn-what-difference

8 What is the .git folder? - Stack Overflow
https://stackoverflow.com/questions/29217859/what-is-the-git-folder

9 53 .gitignore file - ignoring files in Git | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/saving-changes/gitignore

10 Git - Viewing the Commit History
https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

11 52 Things I Wish I Knew About Git · vsupalov.com
https://vsupalov.com/git-facts/

12 13 14 15 16 56 Git pull vs. git fetch: What's the difference?
https://about.gitlab.com/blog/git-pull-vs-git-fetch-whats-the-difference/

17 47 48 Difference Between the Git Reset, Revert, and Checkout Commands | Delft Stack
https://www.delftstack.com/howto/git/git-revert-vs-reset/

18 19 39 GitHub - uoft-oss/git-workflow: A simple guide to Git
https://github.com/uoft-oss/git-workflow

20 21 22 23 24 25 43 73 git stash - Saving Changes | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/saving-changes/git-stash

26 Fork a repository - GitHub Docs
https://docs.github.com/articles/fork-a-repo

27 Forking Workflow - Kinda
https://alndaly.github.io/en/docs/others/Github/forking%20workflow

28 29 32 50 What Is a Pull Request? | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/making-a-pull-request

30 Managing a branch protection rule - GitHub Docs
https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/managing-a-branch-protection-rule

31 GitHub Code Review
https://github.com/features/code-review

33 34 Git - Contributing to a Project
https://git-scm.com/book/ms/v2/GitHub-Contributing-to-a-Project

35 GitHub Issues - OpenScPCA Documentation
https://openscpca.readthedocs.io/en/latest/communications-tools/github-issues/

36 42 51 54 68 Contributing to open source - GitHub Docs
https://docs.github.com/en/get-started/exploring-projects-on-github/contributing-to-open-source

37 Project Collaboration using GitHub | Manoov's Homepage
https://manoov.github.io/courses/instructions/

38 Managing code review settings for your team - GitHub Docs
https://docs.github.com/en/organizations/organizing-members-into-teams/managing-code-review-settings-for-your-team

40 Trunk-based Development | Atlassian
https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development

41 Trunk-Based Development Vs Git Flow: A Comparison - Assembla
https://get.assembla.com/blog/trunk-based-development-vs-git-flow/

44 How do I discard unstaged changes in Git? - Stack Overflow
https://stackoverflow.com/questions/52704/how-do-i-discard-unstaged-changes-in-git

45 49 57 Undoing Changes in Git | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/undoing-changes

46 72 Resetting, Checking Out & Reverting | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting

58 Can I recover a branch after its deletion in Git? - Stack Overflow
https://stackoverflow.com/questions/3640764/can-i-recover-a-branch-after-its-deletion-in-git

59 60 61 62 63 64 Git Mistakes a Developer Should Avoid - GeeksforGeeks
https://www.geeksforgeeks.org/git/git-mistakes-a-developer-should-avoid/

65 80 firstcontributions/first-contributions: Help beginners to ... - GitHub
https://github.com/firstcontributions/first-contributions

66 67 Hello World - GitHub Docs
https://docs.github.com/en/get-started/start-your-journey/hello-world

69 What is .env? A guide to understanding the .env file | Upsun
https://upsun.com/blog/what-is-env-file/

70 How to Write a Good README File for Your GitHub Project
https://www.freecodecamp.org/news/how-to-write-a-good-readme-file/

71 git - How to remove branch from a remote (on Atlassian Stash/Bitbucket) - Stack Overflow
https://stackoverflow.com/questions/19358162/how-to-remove-branch-from-a-remote-on-atlassian-stash-bitbucket

74 The Complete Guide to GitHub Flow: A Practical Tutorial ... - Medium
https://medium.com/@YodgorbekKomilo/the-complete-guide-to-github-flow-a-practical-tutorial-for-developers-0ea8e5ad1c1e

75 When is it acceptable to commit directly to main/master?
https://softwareengineering.stackexchange.com/questions/455944/when-is-it-acceptable-to-commit-directly-to-main-master

76 77 Git submodule | Atlassian
https://www.atlassian.com/git/tutorials/git-submodule

78 79 What Recruiters Look For in a GitHub Profile — and How to Optimize Yours - DEV Community
https://dev.to/hexshift/what-recruiters-look-for-in-a-github-profile-and-how-to-optimize-yours-j0e