

- » Understanding version control
- » Learning the crucial Git commands
- » Storing your project on GitHub
- » Managing different environments
- » Deploying your site

Chapter 1

Version Control and Deployment

In preparing for battle I have always found that plans are useless, but planning is indispensable.

—DWIGHT D. EISENHOWER

Okay, you've done the hard part. You've planned your site, designed it, created it, SEO-ed it, and probably consumed more coffee than any reasonable person should. The site looks good and it works. Now what?

Now you need to get it online — and keep it online, healthy, and recoverable when something inevitably goes wrong. (And something will go wrong. That's not pessimism; it's web development.)

This chapter is about the tools and habits that turn a finished project into a properly managed, deployable website. The star of the show is *version control*, a system for tracking every change you make to your project files, so you always have a record of what changed, when, and why. The main version control tool used by web developers is called Git, and once you try it, you'll wonder how you ever worked without it. I also walk you through storing your project on GitHub, managing different versions of your site across development and production environments, and getting your site live using modern deployment tools.

Understanding Version Control and Git

Imagine you're editing your website's home page. You move some elements around, rewrite the headline, and try a different color scheme. Then you realize the new version looks worse than the old one. You'd love to just go back, but you've been saving over the same file for the past hour. Oops.

That's the problem version control solves. A *version control system* keeps a history of every change you make to your project files. You can see what changed and when, compare old versions to new ones, and even roll back to any earlier state with a single command. It's like an infinite undo button for your entire project. Not just one file. Not just the current session. Every file, going all the way back to the first day you started the project (or, more realistically, the first day you implemented your version control system).

Why version control matters even if you work alone

Version control is often presented as a team tool, and it's true that it's indispensable for teams. But even if you're the only person who ever touches your website, it pays back the (relatively minor) effort required to learn it in several very practical ways:

- » **Safety net:** Break something? Roll back to the last working version in seconds. No more staring at broken code trying to remember what you changed.
- » **Audit trail:** Every change is logged with a timestamp and a message explaining what you did and why. Future-you — the one who can't remember why they changed that CSS rule three months ago — will be deeply grateful.
- » **Fearless experimentation:** Want to try a completely different layout? Rewrite the navigation from scratch? Version control means you can experiment freely, knowing you can always get back to what worked.
- » **Off-site backup:** When you store your version history on a service such as GitHub, your entire project is safely backed up in the cloud. Your laptop gets stolen? Your project survives.
- » **Professional habit:** Any time you hand a project to another developer — or come back to it yourself after six months away — version-controlled code with descriptive change messages is a thousand times easier to work with than a folder full of files named `index-final-FINAL-v3-USE-THIS-ONE.html`.

Getting to know Git

Git is the world's most widely used version control system. It was created in 2005 by Linus Torvalds (who also created the Linux kernel, in case you're keeping score). Git runs on your computer as a command-line tool, inside the folder that contains your website project. You tell Git when you want to save a snapshot of your work, and it stores that snapshot — along with all the snapshots you've ever saved — in a hidden folder called a *repository* (or *repo*, if you want to sound like you've been doing this for years).

The key thing to understand about Git is that it stores the current state of your files *and* it stores every state they've ever been in, along with the capability to determine what changed between each state. That's what makes it so powerful. You're essentially building a time machine for your project.

Yes, Git has a learning curve. No, it's not as steep as people make it sound. You need about eight commands to handle 90 percent of what a solo web developer does with Git, and I walk you through all of them in the next section.

Git Basics for Web Projects

Before you can use Git, you need it installed. Head to git-scm.com and download the installer for your operating system. If you're running macOS, you may already have it. To check, open a Terminal window, type `git --version`, and if you see a version number, you're good to go. If, instead, Terminal prompts you to install the command-line developer tools, follow the instruction to do that, and then run `git --version` again to confirm that all is well.

You also need to introduce yourself to Git before you start. Git uses your name and email to label your commits, so it knows who made each change. Open Terminal and run the following commands (remember to press Enter or Return after each command):

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

Replace the placeholders within the quotation marks with your name and email. That's a one-time setup. Git remembers these details for all your future projects.

Basic command-line survival skills

Everything in Git happens from the command line, which you get to via the Terminal app on both Windows and macOS. (Note that you might need to get Terminal from the Microsoft Store if you're running an older version of Windows.)

Once you have your command line open, here are a few essential commands that work on all platforms:

- » **cd *path* (change directory):** Moves to the directory specified by *path*. In particular:
 - `cd ~` moves to your main user account directory.
 - `cd ..` moves up to the parent directory of the current directory.
 - `cd ../sibling` moves up to the parent directory of the current directory, and then into the subdirectory specified by *sibling*.
- » **pwd (print working directory):** Displays the full path name of the current directory.
- » **ls (list) on Mac or dir (directory) on Windows:** Displays a list of the files and subdirectories in the current directory.
- » **mkdir *name* (make directory):** Creates a directory named *name*.

Don't sweat memorizing these commands now. You'll be using them a ton as you work with your Git projects, so they'll become embedded in your brain with practice.

Initializing a repository

The first step is to turn your project folder into a Git repository. First, navigate to your project folder:

```
cd ~/path/project
```

where *path* is the path to the directory where your project is stored and *project* is the directory that contains the project files.

Then initialize a new Git repository in that folder:

```
git init
```

Git responds with something like Initialized empty Git repository in `/path/project/.git/`. That `.git` folder is where Git stores its history and configuration. You never need to touch it directly; just know it's there and don't delete it.

Staging and committing your files

Git's workflow has two steps: staging and committing. *Staging* means telling Git which files you want to include in the next snapshot. *Committing* means saving that snapshot with a descriptive message. Here's the sequence:

First, check the current state of your project:

```
git status
```

Git displays the status, which at this early stage in your Git career will look like a bunch of gobbledygook. For now, there are two main things to look for:

» If your repo is up to date, you see one of the following messages:

```
On branch main
No commits yet
```

or

```
Your branch is up to date with 'origin/main'.
nothing to commit, working tree clean
```

» If, more likely, you have files that are untracked or modified but not yet staged, those files appear in red.

If you have red files, the next step is to stage everything:

```
git add .
```

The dot means “everything in this folder.” (If you want to stage just a particular file, run `git add filename`, where you replace *filename* with the name of the file you want to stage.) Run `git status` again and you see the same files now listed in green, which tells you they're staged and ready to commit. Now save the snapshot:

```
git commit -m "Initial commit: add all website files"
```

The `-m` flag is short for *message*. The message in quotes is what you'll see in the history later, so make it meaningful. After running this, Git confirms the commit with a short summary: something such as `[main (root-commit) 1c8a3d2]`

Initial commit: add all website files. That string of letters and numbers (1c8a3d2) is the commit's unique ID.

The everyday Git rhythm

From this point on, your version control workflow is a simple three-step rhythm that you repeat every time you make a meaningful set of changes:

1. Make some changes to your files (edit HTML, update CSS, add a page, fix a bug).
2. Type `git add .` to stage all the changed files.
3. Type `git commit -m "Describe what you changed and why"` to save the snapshot.

How often should you commit? As a rule, commit after each meaningful unit of work. “Updated Galaxy Glazed donut description” is a good commit. “Fixed broken link to catering page” is a good commit. “Stuff” is not a good commit. The more granular your commits, the more precisely you can pinpoint and undo problems later.



TIP

Commit messages read best in the imperative mood: “Fix broken link in footer” rather than “Fixed broken link” or “I fixed the link.” Git itself follows this convention, and a list of imperative-mood messages is much easier to scan quickly than a diary of past events. Think of each message as an instruction: If you applied this commit, what would it do to the project?

Reading your project's history

To see a list of all your commits, type:

```
git log --oneline
```

The output looks something like this:

```
a3f92c1 Update donut prices for summer menu
7d14e88 Add catering page with inquiry form
b021f47 Fix responsive layout on mobile menu page
1c8a3d2 Initial commit: add all website files
```

The most recent commit is at the top. Each line shows a short commit ID and your message. To see what changed in any given commit, type `git show` followed by the commit ID:

```
git show b021f47
```

Git displays the full differential (or *diff*, to the cognoscenti): lines that were added are marked with a plus sign, and lines that were removed are marked with a minus sign. It's a precise, line-by-line record of what changed. Once you get used to reading it, you'll find it invaluable for understanding why the code looks the way it does.

You can also compare any two commits directly to see what changed between them:

```
git diff 1c8a3d2 a3f92c1
```

Git shows everything that changed between the initial commit and the most recent one. For a project that's been running for a few weeks, this is a surprisingly satisfying way to see how much work you've done.

Telling Git what to ignore

In almost every web project, you'll have files that don't want Git to track. Hidden operating system files such as `.DS_Store` (a Mac thing), editor configuration folders such as `.vscode`, and any folders full of installed packages don't belong in your repository. Git has a simple, elegant solution: a file called `.gitignore`.

Create a plain text file named `.gitignore` in the root of your project folder. List any files or folders you want Git to ignore, one per line. Here's a sensible starting `.gitignore` for most web projects:

```
# Mac OS system files
.DS_Store

# Windows thumbnail files
Thumbs.db

# Editor settings folders
.vscode/
.idea/
```

Lines starting with # are comments. Once you've created `.gitignore`, commit it just like any other file. Git reads it automatically and skips the files and folders you've listed. You can also use wildcards: `*.log` ignores all log files, `temp/` ignores the entire `temp` folder. GitHub even maintains a library of ready-made `.gitignore` templates for dozens of project types at github.com/github/gitignore, so you're rarely starting from scratch.

Using GitHub for Website Projects

Git by itself lives entirely on your computer. That's useful, but it has one obvious weakness: if your computer dies, your version history goes with it. GitHub (github.com) is a web service that hosts Git repositories in the cloud. It's free for both public and private projects, and it's where the vast majority of the world's web projects live. Think of GitHub as a combination of off-site backup, public portfolio, and collaboration hub.



WARNING

Yes, GitHub acts as a kind of off-site backup, but only for the files you've included in your commits. You still need a robust backup strategy for *all* your files, as I explain in Book 9, Chapter 4.

GitHub and Git are different things, which trips people up at first. Git is the version control tool that runs on your computer. GitHub is a website that stores Git repositories. You use Git locally, and then save (or *push*, in the vernacular) your commits to GitHub for safekeeping.

Creating a repository on GitHub

Create a free account at github.com if you don't have one already. Then:

- 1. Click Repositories and then click the green New button on your GitHub dashboard.**

Alternatively, from any dashboard page, click + near the top right of the page and then click New Repository.

The Create New Repository page shows up.

- 2. In the Repository Name text box, type a name for your repo.**

If you already have a project name for your local repo, use that name. Otherwise, something such as `orbital-donuts` or `my-portfolio-site` works fine. Lowercase letters and hyphens are the convention.

3. (Optional) Add a description, which appears on your GitHub profile.
4. In the Choose Visibility list, select Private if you don't want the public to see your code, or leave Public as the default if you're fine with it being visible.
5. If you want GitHub to create a prepopulated `.gitignore` file for you (and you haven't already created a local `.gitignore` file), use the Add `.gitignore` list to select the template that best matches your code (such as WordPress).
6. Click Create Repository.

GitHub presents you with a page of commands for connecting your local repository to the new GitHub one. The next section explains what those commands do.

Connecting your local repo to GitHub

GitHub shows you two sets of commands, one for setting up your new GitHub repo locally and one for pushing an existing repo to GitHub. Assuming you already have an existing repo that you want to push to your new GitHub repo, here are the commands to use:

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

In the first command, replace *yourname* with your GitHub username and replace *project* with your repo name. Here's what each command does:

- » **git remote:** Tells your local Git repo where the GitHub repo lives; *origin* is a conventional nickname for the remote repository address.
- » **git branch:** Renames your default branch to *main* (GitHub's current naming convention for the primary branch).
- » **git push:** Pushes all your commits up to GitHub for the first time. The `-u` flag sets *origin main* as the default, so for every future push you can just run this:

```
git push
```

Refresh your GitHub repo page and you'll see all your project files there, each one showing the commit message from when it was last changed. Just like that, your work is safely off your hard drive and living in the cloud. Sweet!



WARNING

Never put passwords, API keys, database credentials, or any other secrets directly in your project files. Once something is committed to Git, it's in the history forever, even if you delete the file in a later commit. Store secrets as *environment variables* in your hosting service's settings panel. Netlify, Vercel, and most hosting platforms have a dedicated section for this, usually called Environment Variables or Site Variables. If some sort of secret does accidentally end up committed, revoke the credential (or whatever) immediately and create a new one.

Cloning and pulling

If you need to work on your project from a different computer — or if you've accidentally deleted your local folder (hey, it happens to the best of us) — you can *clone* the repo from GitHub to get a fresh local copy.

First use Terminal to navigate to the parent directory (that is, the directory within which you want to create the repo), and then run the following command:

```
git clone https://github.com/yourname/project.git
```

where, as usual, *yourname* is your GitHub username and *project* is your repo name. The `git clone` command creates a subdirectory named `project`, and then downloads the entire repo, including the full commit history, to that subdirectory. Everything is exactly as it was. Nothing is lost. Your sanity (or maybe your job) is saved.

If you're working on the same project from two different computers — your office desktop and your home laptop, say — you'll push from one and pull on the other to stay in sync.

After you've pushed some changes on one computer, switch to the other computer, navigate to the project directory, and then run the following command:

```
git pull
```

Git downloads any new commits from GitHub and merges them into your local copy. If you made changes in both places without syncing, Git will try to merge them automatically. Usually it succeeds. Occasionally it asks you to resolve a *merge conflict*, where two changes affect the same line and Git isn't sure which one to keep. That sounds scary, but it's straightforward because Git shows you where the conflict is and you just choose which version to keep.

Git Workflows for Solo Developers

Git was designed for large software teams, which is why its documentation can feel overwhelming. Pull requests, rebasing, cherry-picking, upstream remotes. There's a lot in there that you don't need for a solo web project. Focus on just two things: committing well and using branches for experiments.

Committing with intention

The single most valuable version control habit isn't any specific command; it's committing regularly and writing good messages. Compare two commit histories for the same project:

Less useful:

- » Worked on site.
- » More changes.
- » Stuff.
- » Untitled.

More useful:

- » Add summer menu page with seasonal donut descriptions.
- » Resize hero image to reduce load time.
- » Fix contact form layout on small screens.
- » Update footer with new business hours.

The second history tells a story. You can scan it in 30 seconds and know what happened and when. That matters enormously when something breaks or when you come back to a project after three months and can't remember how it works.

Here are a few commit message best practices that make the history even more useful:

- » Keep the first line under 50 characters. It's the title; make it scannable.
- » Use the imperative mood: *Add* not *Added*, *Fix* not *Fixed*.
- » If the commit needs more explanation, add a blank line after the first line and write a longer description. The first line is the subject; the rest is the body.

Branches: Your experimental playground

A *branch* is a parallel version of your project where you can make changes without affecting your main working copy. Branches are how you safely experiment with big changes — such as a new page layout, a revised navigation structure, a different color scheme — without risking the version that’s currently working.

Suppose you want to try a redesigned order page for the Orbital Donuts site. Here’s how to create and switch to a new branch:

```
git checkout -b order-page-redesign
```

This creates a new branch called `order-page-redesign` and switches to it immediately. Your `main` branch is untouched. Make your changes, commit them to this branch, and test everything thoroughly.

If the redesign turns out to be a disaster, you just delete the branch and your main site is completely unaffected:

```
git switch main  
git branch -D order-page-redesign
```

No harm done. The experiment is gone, the main site is intact, and you’re back where you started.

If, on the other hand, the redesign is a success, you can merge it back into `main`:

```
git checkout main  
git merge order-page-redesign
```

Everything from the `order-page-redesign` branch is now part of `main`. Nice!

Branches make trying things genuinely low-risk, and once you get comfortable with them you’ll reach for them constantly.



REMEMBER

You can see all your branches at any time with:

```
git branch
```

The current branch is marked with an asterisk. Switching between branches is always the same command: `git checkout` followed by the branch name.

Managing Environments

When web developers talk about *environments*, they're referring to the different contexts in which a website runs at different stages of the development process. Most professional web projects use three distinct environments, each with a specific purpose:

- » **Development (dev):** Your local copy, running on your own computer. This is where you write code, break things, and fix them. Nobody sees this environment but you. It's perfectly fine for it to be broken at 2 a.m. when you're in the middle of something.
- » **Staging:** A copy of the site hosted somewhere online but not publicly advertised and usually behind a password or on an obscure URL. This is where you test changes in a real hosted environment before they go live. Think of staging as the dress rehearsal before opening night.
- » **Production:** The live site that your visitors see. Changes land here only after they've been tested and approved in staging. This is the one you protect.

Why the staging step matters

The classic mistake is making changes directly in production. It's tempting because it's faster, there's no setup, and the change is just a small thing. Until the small thing breaks something you didn't expect, at a moment you can't predict, in front of real visitors.

A staging environment interrupts that pattern. You deploy the change to staging, check it in a browser, make sure nothing exploded, and then — and only then! — deploy to production. The extra overhead is small, but the protection is huge.

For example, suppose you're updating the Orbital Donuts online ordering form to add a new field for special instructions. In development, you write and test the HTML. In staging, you confirm it looks right on the hosted server, that the form still submits correctly, and that the page doesn't break on mobile devices. In production, you deploy the tested, confirmed version.

Setting up a simple staging environment

Many modern deployment platforms (which I cover in the next section) create staging environments for you automatically. Every time you push to a branch other than `main`, Netlify and Vercel generate a unique preview URL so you can see what the deployed version looks like before you merge it to production.

If you're on traditional hosting, you can set up staging manually. Create a subdomain — `staging.maplestreetbakery.com`, for example — that points to a separate folder on your server. Deploy your changes there first, test them, and then copy them to the live root folder. Add password protection to the staging subdomain through your hosting control panel so random visitors don't stumble across your half-finished work.

A staging environment doesn't need to be beautiful or fast. It just needs to work well enough for you to catch problems. The small extra effort it takes to set up will pay for itself the first time it catches a bug before a real visitor does.

Modern Deployment Options

Getting your finished site from your computer to the web used to mean using FTP (File Transfer Protocol): connecting to your server with a dedicated app and manually uploading every changed file. FTP clients such as FileZilla still exist and still work, and I cover them in a moment. But modern deployment tools are so much better that, for any new project, it's worth knowing what's available.

Netlify

Netlify (netlify.com) is one of the most popular deployment platforms for small-to-medium websites, and it's free for basic use. The core idea is simple: you connect Netlify to your GitHub repository, and from that point on, every time you push a commit to the `main` branch, Netlify automatically deploys your updated site. No FTP. No manual uploads. Just `git push` and your site is updated with your changed code within about 30 seconds.

After you create your free Netlify account, here's how to deploy a site on Netlify:

- 1. From your Netlify dashboard, click Add New Project.**
- 2. In the Import a Git Repository section, click GitHub.**
The sign in to GitHub page appears in a new window.
- 3. Enter your GitHub credentials, click Sign In, and then click Authorize to give your thumbs-up for Netlify to access your GitHub stuff.**
- 4. When prompted to install Netlify on GitHub, select the Only Select Repositories option, and then use the Select Repositories list to choose the repository you want to use with your Netlify project.**
- 5. Click Install, and then click your new project.**

6. **Leave the build settings at their defaults for a plain HTML/CSS/JS site (no build command needed).**
7. **Click Deploy *repo*, where *repo* is the name of the GitHub repo you selected in Step 4.**

Within seconds, Netlify assigns your site a temporary URL such as `sparkling-donut-42f8a1.netlify.app`. Your site is live.

You can connect your own custom domain in Netlify’s Domain Management panel. Netlify also provides free SSL certificates automatically, so your site gets HTTPS without any extra steps. As I cover in Book 1, Chapter 3, pointing your domain to Netlify is just a matter of updating two DNS records at your domain registrar.

One of Netlify’s most useful features for solo developers is *deploy previews*. When you push commits to any branch other than `main`, Netlify generates a unique preview URL for that branch automatically. This feature gives you a real hosted version of your staging environment without any extra setup. You can share the URL with a client or collaborator to get feedback before the change goes live.

Vercel

Vercel (vercel.com) is Netlify’s closest competitor and is also free for simple web apps and personal projects. (For more advanced projects, you’ll want to step up to the Pro plan, which costs \$20 per month.) The workflow is nearly identical: connect your GitHub repo, and Vercel deploys automatically on every push. Vercel is particularly popular for sites built with JavaScript frameworks but also works just as well for plain HTML/CSS/JS sites.

The main practical differences between Netlify and Vercel are subtle for a beginner, and either will serve you well. If you’re not sure which to choose, I suggest starting with Netlify since its dashboard is slightly more beginner-friendly and its documentation is excellent.

GitHub Pages

If your site is entirely static — just HTML, CSS, and JavaScript with no server-side code — GitHub Pages (pages.github.com) is a brilliantly simple free option. You don’t need a separate account or a separate service. You just enable it from the Settings tab of any GitHub repository, tell it which branch to deploy from, and GitHub automatically serves your site at `yourname.github.io/your-repo-name`.

GitHub Pages is ideal for portfolio sites, documentation, landing pages, and simple small-business sites. Its main limitation is that it doesn’t support server-side

processing (so contact forms need a third-party service, as I describe in Book 3, Chapter 2). For pure static sites, though, it's hard to beat because it's free, reliable, and connected to the same Git workflow you're already using.

Traditional FTP deployment

FTP isn't dead. If your site lives on traditional shared hosting with a cPanel control panel, you'll likely deploy via FTP, which these days is best performed over its more secure variant, SFTP (Secure File Transfer Protocol). Free clients such as FileZilla (filezilla-project.org) handle both protocols and are straightforward to use: enter your hosting credentials, connect to your server, and drag files from your computer to the server.

The downside of FTP is that it's entirely manual. You have to remember which files changed since your last upload, transfer them one by one, and cross your fingers that you didn't miss anything or accidentally overwrite a file on the server that you intended to keep. It works, but it doesn't scale gracefully, and it's easy to introduce errors.

If you're using FTP, it's especially important to use Git alongside it. Git won't automate your deploys, but it will give you the safety net and history that FTP doesn't provide. Your deploy workflow becomes the following:

1. Make changes locally.
2. Commit to Git.
3. Push to GitHub.
4. Upload changed files via FTP.

It's an extra step, but you have a complete audit trail and rollback capability, as I describe in the next section.

Rollback Strategies When Things Go Wrong

One of these days, you'll push a change to the live site and notice (I hope sooner rather than later) that the change broke something. Doh! Now what? Now you need to undo that change, and you need to do it fast. Fortunately, with a good version control setup at your disposal, this scenario doesn't have to be a crisis because it's a two-minute fix.

Reverting a commit

The cleanest way to undo a specific commit is the `git revert` command. Unlike deleting a commit (which rewrites history and can cause problems if you've already pushed), `git revert` creates a new commit that undoes the changes from a previous one. The history stays intact — you can see what happened and when — but the broken change is gone.

First, find the ID of the commit you want to undo:

```
git log --oneline
```

Say the output shows:

```
f91c3a0 Restructure CSS to use new color variables
a3f92c1 Update donut prices for summer menu
7d14e88 Add catering page with inquiry form
```

The top commit (f91c3a0) is the one that broke something. To undo it:

```
git revert f91c3a0
```

Git opens a text editor asking you to confirm the commit message for the revert (it prefills a sensible default). Save and close the editor, and Git creates a new commit that undoes the changes. Then push it to GitHub:

```
git push
```

If you're using Netlify or Vercel, that push triggers an automatic redeployment within a minute. No muss, no fuss, no crisis. Now, with the broken change undone and the history intact, you can figure out why that particular change caused the problem.



REMEMBER

Git doesn't delete history; it only adds to it. Even after a revert, the original commit is still visible in the log. This is a feature, not a bug. It means you can always go back to any state your project has ever been in, no matter how many changes have happened since. The history is permanent.

Rolling back with Netlify's deploy history

Netlify keeps a permanent record of every deploy it's ever done for your site. If you need to revert to an earlier version without touching Git, go to the Deploys tab in your Netlify dashboard. You'll see a list of every deployment, with timestamps

and commit messages. Find the last good deploy, click it, and then click Publish Deploy. Netlify rolls the live site back to that version instantly.

This feature is particularly useful when you're managing a site for a client and need to fix a broken live site in under a minute. You can roll back in Netlify in about ten seconds while you take more time to properly diagnose and fix the problem in your local environment.

Discarding uncommitted local changes

Sometimes the problem isn't something you've committed; it's something you've changed locally but haven't committed yet. You've spent an hour editing `styles.css` and it's a complete mess. You just want to throw away all those changes and go back to the last committed version. Here's how:

```
git restore styles.css
```

Git restores `styles.css` to the state it was in when you last committed. Your edits are gone, which is what you want.



WARNING

Note that there's no undo when you use `git restore`. Git doesn't keep uncommitted changes in a trash can. Make sure you genuinely want to discard them before running this command.

To discard all uncommitted changes across your entire project at once:

```
git restore .
```

Everything goes back to the last committed state. It's a nuclear option, but sometimes that's what you need.

A Git Cheat Sheet

At this point, you know enough Git to handle almost everything that comes up in a web project. Here's a compact reference for the commands you'll reach for week after week:

» **git init:** Turn a folder into a Git repository. Run this once, in the project root folder, when you start a project.

- » **git status:** See which files have changed since the last commit and which are staged. Run this constantly because it's the most informative command in Git.
- » **git add .:** Stage all changed files for the next commit. To stage a single file, replace the dot with the filename (e.g., `git add menu.html`).
- » **git commit -m "message":** Save a snapshot with a descriptive *message*.
- » **git log --oneline:** Show a compact history of commits, most recent first. Run `git log --oneline -10` if you only want the last ten.
- » **git push:** Upload your local commits to GitHub. Do this regularly because it acts as a kind of cloud backup launcher.
- » **git pull:** Download and merge commits from GitHub into your local repo. Run this when starting a session if you've worked from another machine.
- » **git checkout -b branch-name:** Create a new branch and switch to it immediately. Your main branch is untouched while you experiment in here.
- » **git switch main:** Switch back to the main branch. (Or `git switch branch-name` to switch to any branch.)
- » **git merge branch-name:** Merge another branch into the current one. Run this from main after switching back to it.
- » **git revert commit-id:** Create a commit that undoes a specific previous commit. This is the safe, history-preserving way to undo something you've already pushed.

Note that list isn't long. That's the point. Git has dozens of commands and hundreds of options, but you don't need most of them for everyday web development. Master these and you'll be in excellent shape.

