10 Git Recipes Every Developer Should Know

Bartley Anderson

First Edition

Table Of Contents

Introduction

I am writing this book as a way of bringing together most of the knowledge about Git that I rely on as a developer on a daily basis. I figured that since this knowledge is so useful to me, it might be useful to you as well, especially if it is wrapped up into one convenient package.

Since we are all at different stages in our journey with Git I included a section at the beginning to cover some of the basics about how Git fundamentally operates, and how to get started using it for your own projects.

The recipes themselves can be divided into three major groups:

1. Creating new things
2. Making changes
3. Fixing mistakes

Also, I talk about Git in terms of writing code (I'm a developer after all), but anyone who edits files can take advantage of Git. Don't think you can only use Git to write code. I know of more than a few Git projects online whose main product is a book written by multiple authors working together via Git. Don't let my perspective as a developer throw you off. All of these recipes are equally applicable to anyone who uses Git for any purpose.

What is Git

Git is a Source Code Management (SCM) tool, also known as a Version Control System (VCS).

That means it stores files for you, and keeps track of the changes you make to those files over time.

Basically, Git allows you, and others, to work on code over time, keep track of the changes that are made to it, and, occasionally, roll those changes back to a previous version, or go find some code that was changed or deleted in the past.

In this book I'm using the word "recipe" to describe routines that you, and your teammates, can use to work on code, together, day-to-day. They're recipes because you often need to run several commands in a certain sequence in order to accomplish a given task in Git, and *recipe* just seems like the most appropriate word to describe that.

Git Repositories

A Git repository is the main storage location for all of your files in a given project. There are two types of Git repositories - local and remote.

Local

A local Git repository lives in a folder on your computer. When you connect to a remote repository it will automatically create a local repository on your computer that has a full copy of the remote repository in it. The local repository on each person's machine and the remote repository that they are connected to all work together to keep all the files in the repo in sync.

You can also turn any folder on your computer into a local Git repository simply by running this command in that folder:

```
git init
```

This won't give you a connection to any remote repository, but you can easily connect a local repository that is created this way to a remote repository later on.

Remote

A remote Git repository lives in a folder on a server. Git has two main ways of connecting to remote repositories - https and ssh. You can tell which type of connection you are using by the repository address you are connecting to.

If your connection starts with `git@...` then you are using an ssh connection. For example,

```
git@example.com:user/project.git
```

The *git* in *git@...* is the ssh username. This is the typical default ssh username across many Git servers so you will see it in many places. I just wanted you to know that it's nothing more magical than that. Plus, it could be set to something other than *git*, so keep that in mind if you ever see a different name there.

If your connection starts with `https://...` then you are using an https connection. For example,

```
https://example.com/some-user/some-project.git
```

The difference between the two comes down to your login credentials.

You will need to authenticate with the repo before Git will let you commit code to it. If you are using ssh then you can register a public key with your GitHub/GitLab account and ssh will use your private key (which lives in your .ssh folder) to authenticate your connection requests. If you are using HTTPS then you will either need to type your username and password every time you commit, or you will need to set up a means of authentication in your .gitconfig file, such as using wincred on Windows, to automatically send your username and password every time you commit.

I will typically use the ssh version of the connection in the examples in this book because connecting via ssh is really the most convenient way to do it, and I highly recommend that you set up an ssh key in your git server account if you haven't done that already. The help guides for your provider will have instructions on how to do this if you need help. If you skip the process of setting up an ssh key then just replace "git@" with "https://" in the examples.

Another thing to remember is that you don't need to log in to a Git repo in order to clone it or pull changes from it - only to push changes to it. And that means that you can clone any Git repo that you find online. But you will be limited to pulling from it over https only.

We will talk about the different ways you can connect to a remote Git repository coming up in Recipe 1.

Also, the *master* branch is typically reserved for production code, and development work is done in a different remote branch - usually named *dev* or *development*, etc... Throughout this book I will assume that you are doing your development work in a remote branch named *dev*. If you are doing all of your work in the master branch, or a branch that is named something else, remember to replace *dev* with your actual branch name when you use the recipes.

Git Server Account

The first thing you need to do in order to properly learn Git is to get access to a server where you can create a Git repository. While it is possible for you to set up your own Git server, that topic is out of the scope of this book. There are several free Git servers on the internet that you can choose from, and more than a few paid ones for that matter. Plus, if you're a developer, a public source code repository makes a wonderful addition to your resume. If you don't already have a personal Git server account I suggest creating one from one of the following providers:

- github.com
- gitlab.com

I won't get into the details of comparing each of these providers other than to say that they all allow you to create personal repositories (both public and private) for free. If you're not interested in comparing them to each other, you can't go wrong with GitHub.

I will be using GitHub as my point of reference throughout this book, but the features I will talk about (such as creating a new repo, or adding ssh credentials) are available on GitLab as well.

So, if you don't already have a Git server account, go create one and come back here when once you've done that. And don't forget to setup ssh authentication while you're at it.

Merging

Git is first and foremost a tool that helps you keep your changes to any given file in sync with the changes that others make to that same file.

When you create a new file there's no chance that anyone else has made a change to it, so you don't have to worry about overwriting the work that anyone else did when you commit it to your Git repo. However, once that file is committed to the repository, someone else can pull it down, make changes to it, and push it back up to the repo.

What happens if two people are making different changes to the same file at the same time? Well, if you were using something like a shared folder to keep track of your source code, the last person to save their changes would overwrite the changes that anyone else made before them - causing their work to be lost.

Git will actually recognize that two people (or, to be more accurate, two commits) have made different changes to the same file, and, instead of overwriting the changes that someone else made, it will merge the different changes together into a new version of the file. This will preserve the work that has already been done, even if the last person to commit the file was not working on the most up-to-date version of that file.

If Git can't figure out how to merge the changes correctly - for example, if two people made different changes to the same line of code - then Git will throw a merge conflict, mark it appropriately in the file, and not allow you to commit the code until you manually merge the changes yourself.

This dance of merging changes is at the heart of everything that Git does. As you go through the recipes in this book, always keep in mind that the ultimate goal of using Git is to write code that other people can work on without overwriting the code that other people have written.

Finding Your Way Around In Git

And, before we get into the recipes themselves, here are a few ways that you can find your place when working in a git repo.

View the files that have been changed in the branch you are on.

```
git status
```

View all the file contents changes in the branch you are on. You can, optionally, specify a file name to see only the changes in that file.

```
git diff
git diff [file-name]
```

View the commit history for the branch you are on. There are also other options that you can use to view this history in more or less detail.

```
git log
git log --all --decorate --oneline --graph
```

View all of the branches for a repo. The branch you are currently on will have an asterisk (*) next to it.

```
git branch -a
```

Recipe 1 - Getting Started

Recipe 1 is broken up into three different options, depending on where you are starting from. You can connect to a remote Git repository by either creating a brand new repository, cloning an existing repository, or cloning a specific branch on an existing repository.

Recipe 1a - Create A New Repo

The easiest way to connect to a new remote repository is to create the repository and then clone it. GitHub and GitLab both have a button located at the top of the web page (when you are logged in) to create a "New repository" - or "New Project".

When you create a new repository, each of these providers will give you instructions on how to connect to your new repository. If you don't have an existing folder with files you want to commit, then it's as easy as cloning the new repository.

```
git clone git@example.com/some-user/some-project.git
```

This will create a folder named `some-project` in the directory you ran `git clone` in, and that folder will have a folder named `.git` inside of it. Git will then be able to keep track of anything you do inside the `some-project` folder.

When working on personal projects I don't usually create a new repository as soon as I start the project, though. I typically only create a new repository after I've worked on a project to some extent and decided that I actually want to keep it. That means that I usually already have a folder with files in it that I want to then connect to a new Git repo. If you create a new, empty repository on the server and then want to connect an existing folder on your computer to it you can `cd` into that folder on your computer and run

```
git init
git remote add origin user@example.com/some-user/some-project.git
```

Or, if you're connecting via https

```
git remote add origin https://example.com/some-user/some-project.git
```

This does two things.

One, `git init` turns your folder into a local Git repository.

Two, `git remote add` assigns a remote repository address to your local repository.

`origin` is a shorthand name for the remote repository address. `origin` is basically a variable name, and you can change it to whatever you want. But `origin` is the widely recognized convention so I would recommend you stick with it unless you have a good reason not to.

Once you link your local repo to your remote repo, you will need to push up the master branch in order for it to exist on your remote repo, and you can do that with this set of commands

```
git add -A
git commit -m "Initial commit"
git push -u origin master
```

Then everything that is in your local folder will now be stored in your remote repo.

Recipe 1b - Clone An Existing Repo

This is just like cloning a new repo except you can clone any existing repo and the "some-project" folder will already come with files in it.

```
git clone git@example.com/some-user/some-project.git
```

Remember, if you don't have login access to the repo then you will need to use the https connection to clone it.

```
git clone https://example.com/some-user/some-project.git
```

That's all there is to it. You can now start working in that repo.

Recipe 1c - Clone A Specific Branch From An Existing Repo

In some specific cases you might want to only clone a specific branch of a repo. For example, if a repo has one branch that is 5MB in size and another branch that is 5GB in size, and you only want to clone the 5MB branch without downloading the entire 5GB branch.

Thankfully Git allows you to work with this level of surgical precision by using the `--single-branch` option.

```
git clone -b branch-name --single-branch git@example.com/some-user/some-
project.git
```

Here the `-b branch-name` tells Git that you want to clone a specific branch, and `--single-branch` tells Git that you don't want to have references to any other branch in the repo.

Cloning a specific branch is not a common practice. Typically you will just clone the whole repo and "checkout" the branch you want to work in. But it's nice to have the option if you need it.

Recipe 2 - Committing Changes

Recipe 2 is the most common recipe you will probably use. This is the standard routine for making and committing changes to a repo.

As you make changes, you merge them into the repository. Each merge creates a new version of the files and adds another checkpoint to the history of the repository.

This is the recipe I have used for years for merging my work into the remote repository.

```
git checkout dev
git pull
git checkout -b edits
```

... do some work

```
git status
git add -A
git commit -m "commit message"
git checkout dev
git pull
git merge edits
git push
git branch -d edits
git checkout -b edits
```

... do some work

... repeat ...

Now, let's walk through each line and understand what's happening.

`git checkout dev` - I am working on the `dev` branch. `dev` is a branch in the remote repository that all development work happens in, and I want to ultimately push changes to that branch.

*Remember *dev* is the branch I'm using for my development work, you might be using *master* or some other branch.

`git pull` - I am pulling in any changes from the remote `dev` branch to my local `dev` branch so that I will be working on the latest version what's in `dev`.

`git checkout -b edits` - I am creating a new local branch, named `edits` (name it whatever you want), off of the `dev` branch to do my work in. It is a copy of the `dev` branch that I can freely modify. You should create a new local branch to do all of your work in. If you do all of your work in the `dev` branch then you will create merge noise in the history when you pull in other people's changes. You can create a new local branch with `git branch <branch-name>`, and you can checkout a branch with `git checkout <branch-name>`. Or you can do both at the same time with `git checkout -b <branch-name>`.

Once I am in my local `edits` branch I can do whatever work I need to do.

`git status` - Before I commit, I check what files I modified by running `git status`. This is an opportunity for me to do some code review on myself. In particular I'm looking for files that I should not have modified, or any files or folders that I don't want to commit. If I don't remember what changes I made to a file I can check to see what changes I made by running `git diff <path-to-file>`. If I want to undo all of my changes to a particular file I can revert it back to the state it was in when I created the `edits` branch by running `git checkout <path-to-file>`.

`git add -A` - I am staging all of my changes (new, modified, and deleted files) in my `edits` branch to be ready to commit. There are other ways to stage changes. You can stage individual files for commit with `git add <path-to-file>`. You can stage all new files and modifications in a directory with `git add .`. You can stage all modifications and deletions with `git add -u`. `git add -A` is the same as running `git add .` and `git add -u` together.

`git commit -m "commit message"` - I am committing my changes in my `edits` branch. Those changes are still only in my `edits` branch, but after this `git status` will tell me that I haven't made any changes. The `-m "commit message"` is required or else Git will show you a message about how to use `commit` correctly. If you just use `-m` without the commit message, Git will open an editor and ask you to type your commit message in there.

Note: You can see the commit history for your current branch by running `git log`.

`git checkout dev` - I'm changing back into my local copy of the `dev` branch, where I ultimately want to merge my changes.

`git pull` - I'm updating my local `dev` branch with any changes that have been pushed to the remote `dev` branch - perhaps by other people who are also working on the `dev` branch.

`git merge edits` - I'm telling Git to write my committed changes from my `edits` branch onto my local copy of the `dev` branch. Git will try to merge all of my changes automatically, but it will throw merge conflicts if it runs into anything that it can't figure out how to merge correctly. See the **Fixing Merge Conflicts** recipe for advice on how to handle this.

`git push` - Assuming that there were no merge conflicts, I push my local `dev` branch up to the remote `dev` branch, and my changes will now be available for everyone else to pull down from there. You don't want to wait for very long between the time you pull and the time you push. If someone makes changes to the `dev` branch that you don't pull down then Git will throw an error telling you that your local branch is behind your remote branch. You will just have to pull again, but that could create merge noise. For advice on how to remove that noise, see the **Rebasing: Combining Several Local Commits** recipe.

`git branch -d edits` - If you pulled down changes to the `dev` branch when you merged and committed, then those changes won't be in your current `edits` branch. You could merge your `dev` branch with your `edits` branch, or rebase your `dev` branch onto your `edits` branch, but it's guaranteed to be cleaner and easier to just delete and recreate your `edits` branch. If you didn't pull down any changes to your `dev` branch, and it said "Already up to date" when you pulled, then you don't need to delete your `edits` branch since it will be the same as your `dev` branch at this point.

`git checkout -b edits` - If you deleted your `edits` branch then simply recreate it. If you didn't delete it then run `git checkout edits` instead, and continue working.

That's it. It might seem like a lot if you've never done it before, but do this every day for a week or two and it will become muscle memory.

Recipe 3 - Fixing Merge Conflicts

As I said in the beginning of this book, Git is all about merging changes, and sometimes it just can't figure out how to correctly merge those changes. When that happens it will be up to you to merge them. The first time you do this you will probably scratch your head for a while until you figure out what Git did to your file - at least that's how I felt the first few times I had to fix merge conflicts. Hopefully this recipe will help minimize that initial confusion.

When you get a merge conflict you are usually either on a branch that's linked to your remote repo and you are trying to merge in changes from a local edits branch, or you are trying to merge changes from one local branch to another. So, for the purpose of this explanation, let's assume that you are attempting to commit changes to the remote `dev` branch, and you have done some work on your local `edits` branch.

For this scenario let's pretend that you and your co-worker, Gandalf, did some work on the same file. You both pulled the same version of the `dev` branch when you started working, and then Gandalf committed his changes before you.

For your part let's assume that you went through the following scenario and ended up with a merge conflict:

```
git checkout dev
git pull
git checkout -b edits
```

...did some work

```
git add -A
git commit -m "commit message"
git checkout dev
git pull
git merge edits
```

... merge conflict

Now you are in the dev branch and you can't push because you have unresolved merge conflicts.

For the sake of demonstration, let's say you both started with the same original copy of a file named index.js, and that file looked like this:

```
console.log('This is a demo of how to fix a merge conflict');

function originalFunction() {
  const originalNumber = 1234;
  console.log('originalNumber :', originalNumber);

  const originalName = 'this is the original text';
  console.log('originalName :', originalName);
}

originalFunction();
```

Then Gandalf made the following changes and pushed them to the dev branch:

```
console.log('This is a demo of how to fix a merge conflict');

function gandalfsFunction() {
  const gandalfsNumber = 1234;
  console.log('gandalfsNumber :', gandalfsNumber);

  const gandalfsName = 'this is the gandalfs text';
  console.log('gandalfsName :', gandalfsName);
}

gandalfsFunction();
```

And you made the following changes and tried to merge them into the dev branch:

```
console.log('This is a demo of how to fix a merge conflict');

function yourFunction() {
  const yourNumber = 1234;
  console.log('yourNumber :', yourNumber);

  const yourName = 'this is the your text';
  console.log('yourName :', yourName);
}

yourFunction();
```

But, when you tried to merge you received this error message:

```
Auto-merging src/index.js
CONFLICT (content): Merge conflict in src/index.js
Automatic merge failed; fix conflicts and then commit the result.
```

So you open up the index.js file, and you see this:

```
console.log('This is a demo of how to fix a merge conflict');

<<<<<<< HEAD
function gandalfsFunction() {
  const gandalfsNumber = 1234;
  console.log('gandalfsNumber :', gandalfsNumber);

  const gandalfsName = 'this is the gandalfs text';
  console.log('gandalfsName :', gandalfsName);
}

gandalfsFunction();
=======
function yourFunction() {
  const yourNumber = 1234;
  console.log('yourNumber :', yourNumber);

  const yourName = 'this is the your text';
  console.log('yourName :', yourName);
}

yourFunction();
>>>>>>> edits
```

Now, I specially crafted "your changes" and "Gandalf's changes" in this merge conflict to make it easy to point out what's going on, but in real life you will probably face something that looks much more cryptic.

The merge conflict is broken into two sections - Gandalf's changes, and yours. The way the code is worded, it should be easy to tell which changes were made by whom. Your code in real life will likely not contain you and your co-worker's names though. Looking at the merge conflict markers, you have:

```
<<<<<<< HEAD
=======
>>>>>>> edits
```

HEAD is simply a reference to the last commit in the branch you are currently on. In this case, it is referring to the commit that Gandalf made to the `dev` branch.

edits is the name of the branch you are trying to merge into HEAD (HEAD being the `dev` branch in this case).

So, the code between `<<<<<<< HEAD` and `=======` is the current state of the code on the `dev` branch. And the code between `=======` and `>>>>>>> edits` is your code that you are trying to merge into the `dev` branch. It is now your job to figure out how to merge your code with the code that is currently on the dev branch.

Maybe you keep all of your changes and overwrite all of Gandalf's changes, maybe you keep all of his and get rid of all of yours, or maybe you have to rewrite that particular piece of code so that you don't break the build when you push up your changes. Ultimately, the end result needs to be working code, and you need to remove those three merge conflict markers from the file before you can commit again.

The main point that you should remember from all of this, which will make deciphering merge conflicts easier for you, is that HEAD refers to the the last commit made to the branch that you are currently on. So, when you run `git checkout <branch-name>` HEAD changes to point to the last commit in `<branch-name>`.

Let's say you changed the file so it looks like this:

```
console.log('This is a demo of how to fix a merge conflict');

function mergeFunction() {
  const mergeNumber = 1234;
  console.log('mergeNumber :', mergeNumber);

  const mergeName = 'this is the merge text';
  console.log('mergeName :', mergeName);
}

mergeFunction();
```

Then you can commit your fix by committing it the same way you would commit any other change

```
git add -A
git commit -m "fixed merge conflicts"
```

Now, if you run `git log` you will see that you have two commits going in to dev - your commit from the edits branch, and your merge fix. That's ok, you can push them just like that. But, you also have the option to rebase those commits into one in order to reduce the noise in the history if you want to (see the **Rebasing: Combining Several Local Commits** recipe).

Now, depending on how long it took you to fix your conflicts, Gandalf could have made another commit to the `dev` branch. So, before you push, you should pull again.

```
git pull
```

... all good

```
git push
```

And, remember, since you pulled down changes from `dev`, your `edits` branch is no longer in line with `dev`. At this point I recommend that you delete the `edits` branch and recreate it.

```
git branch -d edits
git checkout -b edits
```

Good job! Now you're ready to start working on your next task...

Recipe 4 - Rebasing: Combining Several Local Commits

Remember at the beginning of this book when I said that Git is all about merging, and helping people avoid overwriting each others work? Well, rebasing is the tool you use when you want to actually overwrite stuff. That being said, you never want to rebase anything that lives in the remote repository, it should be treated as strictly a local repo tool unless you really know what you are doing. Just keep in mind, it won't stop you from overwriting anything and everything, so it's up to you to know when and how to use it. I suggest you create a practice repo to learn rebasing. It is not a good candidate for on-the-job-training.

This recipe comes in handy when you end up making several commits before you merge your changes into the repo and you want to combine them into a single commit.

Perhaps you keep getting pulled off of the task at hand to do other things, or you're working on a difficult problem and want to create checkpoints that you can roll back to along the way. No matter what the situation, sometimes we just end up making some noise. But there's no reason to commit that noise to the repo.

If you commit several related changes to your local branch before merging them into the repository, you can rebase them into a single commit, which will make your changes easier to follow for anyone looking through the repo history.

```
git checkout dev
git pull
git checkout -b edits
```

... do some work

```
git add -A
git commit -m "WIP - commit message 1"
```

... do some work

```
git add -A
git commit -m "WIP - commit message 2"
```

... do some work

```
git add -A
git commit -m "WIP - commit message 3"
```

At this point you have three WIP (work-in-progress) commits on your local branch that don't exist in the remote repository. They are all related to the same feature or task that you are working on, and don't actually

need to be kept in the history as three separate commits. You can use rebase to combine them all into a single commit that you can then merge into the remote repository.

You can see how many commits you've made on your current branch by running

```
git log
```

If you've been following along with my branch naming then at the top of the list you should see a commit ending with `(HEAD -> edits)`, and three commits down from that you should see a commit ending with `(origin/dev, dev)`.

DANGER! WARNING! ALERT! *Do not* include the commit with `(origin/dev, dev)` or anything below it in your rebase. That stuff all lives in the remote repository and we never rebase anything that lives in the remote repository. You will make people mad at you.

That being said, nobody else in the world knows about the top three commits in your `git log` output, so nobody will miss them if they're gone - because that's what rebasing them is going to do - overwrite them.

OK, we will start the rebasing process with this command:

```
git rebase -i HEAD~3
```

`git rebase -i` tells Git that you want to rebase in interactive mode, which will open an editor and ask you to identify the commit you want to keep, and the commits you want to squash into that commit. The `~3` tells Git that you want to rebase the previous three commits. Replace the `3` with however many commits you want to rebase - as long as none of them are in the remote repository.

Pick the commit that you want to use as the actual commit with `pick`. If you're not sure, just pick the commit at the top of the list. Then `squash` all the other commits into it.

Once you save and close the editor another editor will open and ask you to enter a name for your commit. You can choose one of the existing commit messages, or type in a new one. Make sure to comment out any of the other commit messages that you are not going to use by putting a `#` at the beginning of those lines.

Once you save and close that editor you can verify that your three commits have now been combined into one.

```
git log
```

Then you can merge your commit into the repo as usual.

```
git checkout dev
git pull
git merge edits
git push
git branch -d edits
git checkout -b edits
```

... do some work

... repeat ...

Recipe 5 - Editing A Local Commit

You can add files to your most recent commit and/or change the commit message by using

```
git commit --amend
```

To add files to the previous commit, you only need to stage them, but don't create a separate commit for them.

```
git add -A
git commit --amend
```

If you create a separate commit, then you will need to rebase and squash those commits together (see the **Rebasing: Combining Several Local Commits** recipe).

`git commit --amend` is one of those rebase-like commands that you don't want to run on commits that exist in the remote repository because it overwrites the repo history, and it will break anyone who is depending on that history.

When you run `git commit --amend` it will open an editor for you to edit the commit message. If all you want to do is change the commit message, then this is the easy way to do that. If you want to add files to the commit without changing the commit message, just add the `--no-edit` flag like this

```
git commit --amend --no-edit
```

You can also change the commit message in the command without waiting for Git to open an editor by using the `-m` option, like this

```
git commit --amend -m "your new commit message"
```

Recipe 6 - Undoing A Local Commit

Revert

If you simply want to undo a commit, but you don't want to overwrite your history, then use

```
git revert HEAD
```

Where `HEAD` points to the current commit, but can be replaced with a hash to any commit.

`git revert` will reverse any changes made in the commit you point it at, and create a new commit. The changes you reverted will still live in their respective commit in your repo history, it will simply appear as if you went through and undid them all and then made a new commit.

So, `git revert HEAD` will undo the changes in the current commit and make a new commit with those changes removed, and the commit that was reverted will still remain in the branch history.

Reset

If you want to undo a commit, and you want to remove that commit from the repo history, then you want to use `git reset`.

> Keep in mind that *reset* is another one of those history overwriting commands that you want to be very careful about using on commits that live in the remote repository.

There are three levels of undo that Git gives you with its `reset` command.

You can undo only the commit (--soft), undo the commit and staging (--mixed - default), or undo your changes completely (--hard).

This might be easier to explain if we walk through the process of making the commit, then you can see more clearly where the undoing takes place.

Let's say you have a file - call it index.html - that you need to make changes to.

When you create your edits branch with `git checkout -b edits` the index.html file is the same as the previous commit. If you run `git status` it will tell you that there's nothing to commit.

Then you make a change to index.html. Now `git status` shows you a list of files that are not staged for commit (in red).

Now you run `git add -A`, and `git status` will show you a list of changes to be committed (in green).

And then you commit your changes with `git commit -m "commit message"`. Now `git status` is back to telling you that there's nothing to commit.

Now we can walk backwards and look at undoing these changes. But I want to be clear, you don't have to run the `reset` command multiple times, you just have three versions of the command that each progressively

undoes more than the last. That is to say `--soft` does the least amount of undo, `--mixed` does what `--soft` does plus a little more, and `--hard` does what `--soft` and `--mixed` do but even more.

At this point this is what the three levels of undo will result in.

Soft Reset

```
git reset --soft HEAD^
```

This will take your changes out of the commit and put them back in staging. `git status` is back to showing you a list of changes to be committed (in green).

Mixed Reset

Mixed reset is the default, so when you run `git reset` it assumes the `--mixed` option by default.

```
git reset HEAD^
```

which is the same as

```
git reset --mixed HEAD^
```

This will take your changes out of the commit and un-stage them. `git status` is back to showing you a list of files that are not staged for commit (in red).

Hard Reset

```
git reset --hard HEAD^
```

This will completely remove your changes from index.html, and `git status` will pretend that you haven't changed a thing and tell you that there's nothing to commit.

That's all there is to it. Just like you have different stages of making changes - changed, staged, committed - you also have stages of resetting changes - `--soft` (un-committed), `--mixed` (un-staged), `--hard` (un-changed). And `git log` will appear as if your last commit never happened.

HEAD^ is a shorthand reference to the last commit you made. You can also perform these actions on other commits if you pass their hash instead of HEAD^.

Recipe 7 - Create A New Remote Branch

This is a simple recipe, but one that comes in handy.

There are many reasons why you might need to create a new branch in the remote repo. No matter what the reason, it's as simple as creating a branch, checking it out, and pushing it to the remote.

```
git checkout -b <branch-name>
git push origin <branch-name>
```

Remember that *origin* is just a shortcut name for our remote repo address.

Recipe 8 - Delete A Branch

The counterpart to creating a remote branch is to delete it. Again, this is a simple, but useful, command to know.

```
git push --delete origin <branch-name>
```

And you can delete the local version of the branch with

```
git branch -d <branch-name>
```

Remember that *origin* is just a shortcut name for our remote repo address.
Also, *-d* is the shortcut for *--delete* and they both work in either of those commands.

Once you delete a remote branch, everyone else will have to pull that deletion with the `fetch` command.

```
git fetch --prune origin
```

or

```
git fetch -p
```

Recipe 9 - Delete Stuff From The Remote Branch

Let's say you accidentally forgot to add a file or folder to .gitignore and committed and pushed that file or folder out to the remote repository. Now you want to delete it from the remote repo and also add it to your .gitignore file.

To delete a file from the repo *and* from your local hard drive, run

```
git rm <file-name>
```

To delete a file from the repo but keep it on your local hard drive, run

```
git rm --cached <file-name>
```

If you want to remove a folder you have to tell Git to delete recursively by adding the `-r` option to either of those calls, so

```
git rm -r --cached <file-name>
```

And, of course, for your changes to take effect on the remote repo, you need to commit and push

```
git commit -m "removed unwanted files"
git push
```

Then you can add the file or folder to your .gitignore file so it never gets committed again.

If you accidentally committed something that you want to completely remove from your repo history (such as a password), then you can use the `git filter-branch` command.

If you're reading this because you desperately need to use this command right now, your adrenaline might be pumping. Please, take a deep breath, and take your time.

The whole purpose of this command is to let you rewrite your Git repo history, and it is very powerful.

I won't tell you exactly what you need to do to rewrite your particular repo's history, but you can find the docs for this command at https://git-scm.com/docs/git-filter-branch

An example of how you might use this to remove a particular file from all commits in your repo is

```
git filter-branch --tree-filter 'rm --cached filename' HEAD
```

Again, this is **overwriting** your repo history and it will likely break things for other people who are using that repo. At a minimum you should at least notify others that you are going to take this action on the repo.

If you don't know what you're doing, I would recommend that you create a separate repo and practice on it first. Just keep in mind that other people are going to have to sync up with whatever changes you make.

Recipe 10 - Disaster Recovery

We all make mistakes. Sometimes those mistakes are harmless, and sometimes they threaten the existence of the company we work for. Thankfully, if you ever make one of the latter kind of mistakes and, say, erase part of your company's source code portfolio, Git can help you fix it. It's in these situations where `reflog` and "force pushing" are your friends. You don't want to use them in too many other circumstances, but they come in really handy when you're trying to undo that last `rm -rf *` commit.

The recipe that will save the day for you most of the time is this

```
git reflog
git reset HEAD@{index}
```

The `git reflog` command will show you every commit, reset, revert, rebase, etc... that it has in its cache. Basically, every time the branch's HEAD pointer moves the reflog history gets an entry for it. Each entry in the list has a HEAD@{index} entry that you can use with reset. And `git reset` will overwrite your branch with the the state of the branch at HEAD@{index}.

Also, when you reset to a given index, that command is added to the reflog history without removing anything that came before it from the reflog history, which means that you can still reset things back to the way they were if you need to. With this recipe you can make your branch time travel in a variety of ways.

If the reflog/reset recipe doesn't fix your problem, definitely search the web for answers. Git has a lot more tools and tricks than what I'm covering in this book. I'm not going to tell you to throw your hands up just yet, but I also can't predict the exact right answer for every given scenario. (I wish I could, that would be amazing!)

That being said, if you haven't pushed your mistakes up to the remote repository, and you find yourself in a tangled mess, and you're pretty sure that you can't get out of it, the recipe of last resort for you is to simply delete your project folder and `git clone` the project again.

You will lose whatever changes you made, but your mistakes will all be deleted and you can have a fresh start.

If you have pushed your mistakes up to the remote repository, and you find yourself in a tangled mess, and you've done your homework on the web, and you're pretty sure that you still can't get out of it - have the person with the latest working version of the branch on their machine force push it.

```
git push -f
```

If you were the last person to commit to the broken branch, then you need to reflog/reset to the last working commit that you pushed up and force push it. If that is not an option, then find the person with the next most recent version of the branch and have them push it.

`git push -f` will overwrite the history of your remote repository, so it should be used with great caution. If you're trying to remove a terrible mistake from the remote repository then this command is probably the right

tool for the job. But if you're just in a rush to get your code pushed out at the last minute on Friday afternoon, then I'd advise against using this - at least until Monday.

Also keep in mind that if you, or someone else, has pushed a terrible mistake up to the remote repository, the rest of the people who are using that repository need to be notified immediately so that they do not pull in the broken state of the repo. Remember, someone else on your team might hold the commit that will save the day, and you want them to preserve it and protect it.

Conclusion

I've been using Git for a long time now, and I still remember the first major mistake I made when I first started using it. I incorrectly used one of the menu options in a graphical user interface (GUI) for git in an IDE, and it did something that broke the master branch on the remote repository. And I remember sitting next to the lead engineer for what seemed like hours (probably wasn't nearly that long, but it felt like it) as he tried to fix it.

I suppose that experience could have made me afraid of Git, but it actually made me curious about it. After that I always volunteered for any tasks related to setting up Git repos, or helping people learn to use it, or helping them fix their mistakes. I also swore off using GUIs and vowed to only use git from the command line. Now, whenever someone asks me a question about how to use Git, I usually end up explaining one or more of these recipes to them, so I figured I should just write them all down.

I hope that these recipes serve you well, but there is certainly much more to know about Git. I hope that you are as curious about it as I was, and that you continue to learn more about it and teach others about it too.