APPENDIX C

Git Commands

Throughout the book we have introduced dozens of Git commands and have tried hard to introduce them within something of a narrative, adding more commands to the story slowly. However, this leaves us with examples of usage of the commands somewhat scattered throughout the whole book.

In this appendix, we'll go through all the Git commands we addressed throughout the book, grouped roughly by what they're used for. We'll talk about what each command very generally does and then point out where in the book you can find us having used it.

Setup and Config

There are two commands that are used quite a lot, from the first invocations of Git to common every day tweaking and referencing, the `config` and `help` commands.

git config

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories.

The `git config` command has been used in nearly every chapter of the book.

- In 'First-Time Git Setup' (Chapter 1), we used it to specify our name, email address and editor preference before we even got started using Git.
- In 'Git Aliases' (Chapter 2), we showed how you could use it to create shorthand commands that expand to long option sequences so you don't have to type them every time.
- In 'Rebasing' (Chapter 3), we used it to make `--rebase` the default when you run `git pull`.
- In 'Credential Storage' (Chapter 7), we used it to set up a default store for your HTTP passwords
- Finally, basically the entirety of 'Git Configuration' (Chapter 8) is dedicated to the command.

git help

The `git help` command is used to show you all the documentation shipped with Git about any command. While we're giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run `git help <command>`.

We introduced the `git help` command in 'Getting Help' (Chapter 1), and showed you how to use it to find more information about the `git shell` in 'Setting Up The Server' (Chapter 4).

Getting and Creating Projects

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

git init

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run `git init`.

- We talk briefly about how you can change the default branch from "master" in 'Remote Branches' (Chapter 3).
- We use this command to create an empty bare repository for a server in 'Putting the Bare Repository on the Server' (Chapter 4).
- Finally, we go through some of the details of what it actually does behind the scenes in 'Plumbing and Porcelain' (Chapter 10).

git clone

The git clone command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs git init to make it an empty Git repository, adds a remote (git remote add) to the URL that you pass it (by default named origin), runs a git fetch from that remote repository and then checks out the latest commit into your working directory with git checkout.

The git clone command is used in dozens of places throughout the book, but we'll just list a few i nteresting places.

- It's basically introduced and explained in 'Git Cloning' (Chapter 2), where we go through a few examples.
- In 'Git on the Server' (Chapter 4), we look at using the `--bare` option to create a copy of a Git repository with no working directory.
- In 'Bundling' (Chapter 7), we use it to unbundle a bundled Git repository.
- Finally, in 'Cloning a Project with Submodules' (Chapter 7) we learn the `--recursive` option to make cloning a repository with submodules a little simpler.

Though it's used in many other places through the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

Basic Snapshotting

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

git add

The `git add` command adds content from the working directory into the staging area (or "index") for the next commit. When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We'll quickly cover some of the unique uses that can be found.

- We first introduce and explain git add in detail in 'Tracking New Files' (Chapter 2).
- We mention how to use it to resolve merge conflicts in 'Basic Merge Conflicts' (Chapter 3).
- We go over using it to interactively stage only specific parts of a modified file in 'Interactive Staging' (Chapter 7).
- Finally, we emulate it at a low level in 'Tree Objects' (Chapter 10), so you can get an idea of what it's doing behind the scenes.

git status

The `git status` command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In it's normal form, it also will show you some basic hints on how to move files between these stages.

We first cover `status` in 'Checking the Status of Your Files' (Chapter 2), both in its basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the `git status` command is covered there.

git diff

The `git diff` command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (`git diff` by itself), between your staging area and your last commit (`git diff --staged`), or between two commits (`git diff master branchB`).

- We first look at the basic uses of `git diff` in 'Viewing Your Staged and Unstaged Changes' (Chapter 2), where we show how to see what changes are staged and which are not yet staged.
- We use it to look for possible whitespace issues before committing with the --check option in 'Commit Guidelines' (Chapter 5).
- We see how to check the differences between branches more effectively with the `git diff A...B` syntax in 'Determining What Is Introduced' (Chapter 5).
- We use it to filter out whitespace differences with `-w` and how to compare different stages of conflicted files with `--theirs`, `--ours` and `--base` in 'Advanced Merging' (Chapter 7).
- Finally, we use it to effectively compare submodule changes with `--submodule` in 'Starting with Submodules' (Chapter 7).

git difftool

The `git difftool` command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in git diff command.

git commit

The `git commit` command takes all the file contents that have been staged with `git add` and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

- We first cover the basics of committing in 'Committing Your Changes' (Chapter 2). There we also demonstrate how to use the `-a` flag to skip the `git add` step in daily workflows and how to use the `-m` flag to pass a commit message in on the command line instead of firing up an editor.
- In 'Undoing Things' (Chapter 2) we cover using the `--amend` option to redo the most recent commit.
- In 'Branches in a Nutshell' (Chapter 3), we go into much more detail about what `git commit` does and why it does it like that.
- We looked at how to sign commits cryptographically with the `-s` flag in 'Signing Commits' (Chapter 7).
- Finally, we take a look at what the git commit command does in the background and how it's actually implemented in 'Commit Objects' (Chapter 10).

git reset

The `git reset` command is primarily used to undo things, as you can possibly tell by the verb. It moves around the `HEAD` pointer and optionally changes the `index` or staging area and can also optionally change the working directory if you use `--hard`. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

- We first effectively cover the simplest use of `git reset` in 'Unstaging a Staged File' (Chapter 2), where we use it to unstage a file we had run `git add` on.
- We then cover it in quite some detail in 'Reset Demystified' (Chapter 7), which is entirely devoted to explaining this command.
- We use `git reset --hard` to abort a merge in 'Aborting a Merge' (Chapter 7), where we also use `git merge --abort`, which is a bit of a wrapper for the `git reset` command.

git rm

The git rm command is used to remove files from the staging area and working directory for Git. It is similar to git add in that it stages a removal of a file for the next commit.

- We cover the `git rm` command in some detail in 'Removing Files' (Chapter 2), including recursively removing files and only removing files from the staging area but leaving them in the working directory with `--cached`.
- The only other differing use of git rm in the book is in 'Removing Objects' (Chapter 10) where we briefly use and explain the `--ignore-unmatch` when running `git filter-branch`, which simply makes it not error out when the file we are trying to remove doesn't exist. This can be useful for scripting purposes.

git mv

The git mv command is a thin convenience command to move a file and then run git add on the new file and git rm on the old file. We only briefly mention this command in 'Moving Files' (Chapter 2).

git clean

The git clean command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files. We cover many of the options and scenarios in which you might used the clean command in 'Cleaning Your Working Directory' (Chapter 7).

Branching and Merging

There are just a handful of commands that implement most of the branching and merging functionality in Git.

git branch

The `git branch` command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

- Most of Chapter 3 is dedicated to the branch command and it's used throughout the entire chapter. We first introduce it in 'Creating a New Branch' and we go through most of it's other features (listing and deleting) in 'Branch Management.'
- In 'Tracking Branches' we use the git branch -u option to set up a tracking branch.
- Finally, we go through some of what it does in the background in 'Git References' (Chapter 10).

git checkout

The git checkout command is used to switch branches and check content out into your working directory.

- We first encounter the command in 'Switching Branches' (Chapter 3) along with the `git branch` command.
- We see how to use it to start tracking branches with the `--track` flag in "Tracking Branches" (Chapter 3).
- We use it to reintroduce file conflicts with `--conflict=diff3` in "Checking Out Conflicts" (Chapter 7).
- We go into closer detail on its relationship with `git reset` in "Reset Demystified" (Chapter 7).
- Finally, we go into some implementation detail in "The HEAD" (Chapter 10).

git merge

The `git merge` tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

- The `git merge` command was first introduced in 'Basic Branching' (Chapter 3). Though it is used in various places in the book, there are very few variations of the `merge` command — generally just `git merge <branch>` with the name of the single branch you want to merge in.
- We covered how to do a squashed merge (where Git merges the work but pretends like it's just a new commit without recording the history of the branch you're merging in) at the very end of 'Forked Public Project' (Chapter 5).
- We went over a lot about the merge process and command, including the `-Xignore-all-whitespace` command and the `--abort` flag to abort a problem merge in 'Advanced Merging' (Chapter 7).
- We learned how to verify signatures before merging if your project is using GPG signing in 'Signing Commits' (Chapter 7).
- Finally, we learned about Subtree merging in 'Subtree Merging' (Chapter &).

git mergetool

The `git mergetool` command simply launches an external merge helper in case you have issues with a merge in Git.

We mention it quickly in "Basic Merge Conflicts" (Chapter 3) and go into detail on how to implement your own external merge tool in "External Merge and Diff Tools" (Chapter 8).

git log

The `git log` command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you're currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

This command is used in nearly every chapter of the book to demonstrate the history of a project.

- We introduce the command and cover it in some depth in "Viewing the Commit History" (Chapter 2). There we look at the `-p` and `--stat` option to get an idea of what was introduced in each commit and the `--pretty` and `--oneline` options to view the history more concisely, along with some simple date and author filtering options.
- In "Creating a New Branch" (Chapter 3) we use it with the `--decorate` option to easily visualize where our branch pointers are located and we also use the `--graph` option to see what divergent histories look like.
- In "Private Small Team" (Chapter 5) and "Commit Ranges" (Chapter 7) we cover the `branchA..branchB` syntax to use the git log command to see what commits are unique to a branch relative to another branch. In "Commit Ranges" we go through this fairly extensively.
- In "Merge Log" and "Triple Dot" (Chapter 7) we cover using the `branchA...branchB` format and the `--left-right` syntax to see what is in one branch or the other but not in both. In "Merge Log" we also look at how to use the `--merge` option to help with merge conflict debugging as well as using the `--cc` option to look at merge commit conflicts in your history.

- In “Git Notes” we use the `--notes=` option to display notes inline in the log output, and in “RefLog Shortnames” (Chapter 7) we use the `-g` option to view the Git reflog through this tool instead of doing branch traversal.
- In “Searching” (Chapter 7) we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.
- In “Signing Commits” (Chapter 7) we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch. This is basically entirely covered in “Stashing and Cleaning” (Chapter 7).

git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

- This command is introduced and covered in detail in “Tagging” (Chapter 2) and we use it in practice in “Tagging Your Releases” (Chapter 5).
- We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in “Signing Your Work” (Chapter 7).

Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

- We first look at this command in “Fetching and Pulling from Your Remotes” (Chapter 2) and we continue to see examples of it use in “Remote Branches” (Chapter 3).
- We also use it in several of the examples in “Contributing to a Project” (Chapter 5).
- We use it to fetch a single specific reference that is outside of the default space in “Pull Request Refs” (Chapter 6) and we see how to fetch from a bundle in “Bundling” (Chapter 7).
- We set up highly custom refspecs in order to make `git fetch` do something a little different than the default in “Getting Notes” and “The Refspec” (Chapter 10).

git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you're on.

- We introduce it quickly in "Fetching and Pulling from Your Remotes" (Chapter 2) and show how to see what it will merge if you run it in "Inspecting a Remote" (Chapter 2).
- We also see how to use it to help with rebasing difficulties in "Rebase when you Rebase" (Chapter 3).
- We show how to use it with a URL to pull in changes in a one-off fashion in "Checking Out Remote Branches" (Chapter 5).
- Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in "Signing Commits" (Chapter 7).

git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

- We first look at the `git push` command in "Pushing to Your Remotes" (Chapter 2). Here we cover the basics of pushing a branch to a remote repository. In "Pushing" (Chapter 3) we go a little deeper into pushing specific branches and in "Tracking Branches" (Chapter 3) we see how to set up tracking branches to automatically push to. In "Deleting Remote Branches" (Chapter 3) we use the `--delete` flag to delete a branch on the server with `git push`.
- Throughout "Contributing to a Project" (Chapter 5) we see several examples of using git push to share work on branches through multiple remotes.
- We see how to use it to share tags that you have made with the `--tags` option in "Sharing Tags" (Chapter 2).
- In "Publishing Submodule Changes" (Chapter 7) we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.
- In "Other Client Hooks" (Chapter 8) talk briefly about the `pre-push` hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.
- Finally, in "Pushing Refspecs" (Chapter 10) we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as "origin" so you don't have to type them out all the time. You can have several of these and the git remote command is used to add, change and delete them.

- This command is covered in detail in "Working with Remotes" (Chapter 2), including listing, adding, removing and renaming them.
- It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote add <name> <url>` format.

git archive

The `git archive` command is used to create an archive file of a specific snapshot of the project. We use `git archive` to create a tarball of a project for sharing in "Preparing a Release" (Chapter 5).

git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The submodule command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources. This command is only mentioned and entirely covered in "Submodules" (Chapter 7).

Inspection and Comparison

git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

- We first use it to show annotated tag information in "Annotated Tags" (Chapter 2).
- Later we use it quite a bit in "Revision Selection" (Chapter 7) to show the commits that our various revision selections resolve to.
- One of the more interesting things we do with git show is in "Manual File Re-merging" (Chapter 7) to extract specific file contents of various stages during a merge conflict.

git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author. We showed how to use it to create a nice changelog in "The Shortlog" (Chapter 5).

git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It's a way to get a description of a commit that is as unambiguous as a commit SHA but more understandable.

We use `git describe` in "Generating a Build Number" (Chapter 5) and "Preparing a Release" (also in Chapter 5) to get a string to name our release file after.

Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from figuring out where something was introduced to figuring out who introduced it.

git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search. It is fully covered in "Binary Search" (Chapter 7) and is only mentioned in that section.

git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code. It is covered in "File Annotation" (Chapter 7) and is only mentioned in that section.

git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project. It is covered in "Git Grep" (Chapter 7) and is only mentioned in that section.

Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as thought the commit series is a series of patches. These commands help you manage your branches in this manner.

git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you're currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch that takes all the changes. Cherry picking is described and demonstrated in "Rebasing and Cherry Picking Workflows" (Chapter 5).

git rebase

The `git rebase` command is basically an automated cherry-pick. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

- Rebasing is covered in detail in "Rebasing" (Chapter 3) including covering the collaborative issues involved with rebasing branches that are already public.
- We use it in practice during an example of splitting your history into two separate repositories in "Replace" (Chapter 7), using the `--onto` flag as well.
- We go through running into a merge conflict during rebasing in "Rerere" (Chapter 7).
- We also use it in an interactive scripting mode with the `-i` option in "Changing Multiple Commit Messages" (Chapter 7).

git revert

The `git revert` command is essentially a reverse git cherry-pick. It creates a new commit that applies the exact opposite of the change introduced in the commit you're targeting, essentially undoing or reverting it. We use this in "Reverse the Commit" (Chapter 7) to undo a merge commit.

Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

git apply

The `git apply` command applies a patch created with the `git diff` or even `GNU diff` command. It is similar to what the patch command might do with a few small differences. We demonstrate using it and the circumstances in which you might do so in "Applying Patches from Email" (Chapter 5).

git am

The git am command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

- We covered usage and workflow around git am in "Applying a Patch with `am`" (Chapter 5) including using the `--resolved`, `-i` and `-3` options.
- There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in "Email Workflow Hooks" (Chapter 8).
- We also use it to apply patch formatted GitHub Pull Request changes in "Email Notifications" (Chapter 6).

git format-patch

The `git format-patch` command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted. We go through an example of contributing to a project using the `git format-patch` tool in "Public Project Over Email" (Chapter 5).

git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email. We go through an example of contributing to a project by sending patches with the `git send-email` tool in "Public Project Over Email" (Chapter 5).

git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in "Forked Public Project" (Chapter 5).

External Systems

Git comes with a few commands to integrate with other version control systems.

git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server. This command is covered in depth in "Git and Subversion" (Chapter 9).

git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record. This command is covered in depth in "A Custom Importer" (Chapter 9).

Administration

If you're administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

git gc

The `git gc` command runs "garbage collection" on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format. This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in "Maintenance" (Chapter 10).

git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies. We only quickly use this once in "Data Recovery" (Chapter 10) to search for dangling objects.

git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

- We cover this command mainly in "RefLog Shortnames" (Chapter 7), where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.
- We also go through a practical example of recovering such a lost branch in "Data Recovery" (Chapter 10).

git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

- In "Removing a File from Every Commit" (Chapter 7) we explain the command and explore several different options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.
- In "Git-p4" (Chapter 9) and "TFS" (Chapter 9) we use it to fix up imported external repositories.

Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

- The first one we encounter is `ls-remote` in "Pull Request Refs" (Chapter 6) which we use to look at the raw references on the server.
- We use `ls-files` in Chapter 7, in "Manual File Re-merging", "Rerere" and "The Index", to take a more raw look at what your staging area looks like.
- We also mention `rev-parse` in "Branch References" (Chapter 7) to take just about any string and turn it into an object SHA.
- However, most of the low level plumbing commands we cover are in Chapter 10, which is more or less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.