## 

[**About version control and Git**](#_vdp1a8zspnw) **3**

[**About repositories**](#_uyri407fvea4) **3**

[**How GitHub works**](#_qi5a4p81oe98) **3**

[Snapshots, Not Differences](#_olzg5nn2bbsq) 4

[Git Has Integrity](#_eh2n2b22l5a0) 5

[The Three States](#_hcpsqutzr1te) 5

[**First-Time Git Setup**](#_knhs0u9w59oa) **6**

[**GitHub and the command line**](#_n28944dy1wz1) **7**

[Basic Git commands](#_ppqj8jewoiff) 7

[**What is Origin?**](#_e0r59g1921lh) **8**

[**Git Merge**](#_smm10up280q2) **8**

[**Git Status**](#_k83hc67p1tlt) **9**

[**Git Diff**](#_rgk3siajbywg) **9**

[**Git Rm**](#_e2kc6yz373g2) **9**

[**Ignoring Files**](#_m0ujzo6ivkpz) **10**

[**Git Pull**](#_1iv6mlbi6mzf) **11**

[fetching a remote PR (Pull Request) in to local repo](#_eie4rzvf0bee) 11

[Git vs Github](#_1y897g1mled1) **12**

[**Undoing things**](#_hjf7kikfvqsw) **13**

[Unstage file](#_xbmyrtb7snnc) 13

[Unmodifying a Modified File](#_a69p5id9skgh) 13

[**Working with Remotes**](#_ayq5ir8wtt76) **14**

[Inspecting a Remote](#_fzuyvkujbv5o) 14

[Renaming and Removing Remotes](#_l6sihmj38wvs) 14

[**Tagging**](#_1hm04idai74b) **15**

[Listing Your Tags](#_3h11x58lm7ks) 15

[Creating Tags](#_dvacfogaq2fl) 15

[Annotated Tags](#_kbp9zw2mw3f0) 15

[Lightweight Tags](#_e13551u0vxhm) 15

[Tagging Later](#_f1fkyek7hush) 15

[Sharing Tags](#_4t7wtj5rsr20) 15

[Deleting Tags](#_paqiuwh3s80q) 15

[Checking out Tags](#_xjji7f2x1ymk) 16

[**Branching**](#_i0ywgpoidhef) **17**

[Branches in a Nutshell](#_r626bfo52if5) 17

[Basic Branching and Merging](#_yp4u4349ssxp) 18

[Basic Merge Conflicts](#_8zpdmy21rig4) 18

[**Branch Management**](#_90b9mkz2v51t) **18**

[Git Branch](#_psy7xvelz14e) 18

[Changing a branch name](#_yeocvba220lr) 19

[Changing the master branch name](#_xwu637xky20g) 19

[Branching Workflows](#_tvz1mkytw2bi) 19

[Topic Branch](#_gto4ec1fxx9) 20

[Remote Branching](#_92nieinj3tnb) 20

[Pushing](#_wehhn88l8foc) 21

[Tracking Branches](#_8yd5jkt1awka) 21

[Deleting Remote Branches](#_6umdbwyt0hfo) 21

[Rebasing](#_qerhzm2e1uzx) 21

[More Interesting Rebases](#_g4dm8ftqjv51) 22

[Rebase vs. Merge](#_12mn8ehsz31) 24

[Git Checkout](#_j4lfmgobtnyn) 25

[**Github**](#_wngcsqlb22nj) **26**

[The GitHub Flow](#_8kt0z5q85n0t) 26

[**Git Advanced tools**](#_17frlcb3g30r) **26**

[Revision Selection](#_s1c0y8242gj0) 26

[RefLog Shortnames](#_vxdusb51q45j) 27

[Double Dot](#_fuz6qrtae3z1) 27

[Triple Dot](#_tgwq4ufvufto) 27

[Interactive Staging](#_fl58nu67mssc) 27

[Stashing, Cleaning](#_md80a2we5g7g) 27

[Creating a Branch from a Stash](#_qx6mj6xur3yg) 28

[Git Grep](#_ks21vkv5w5bf) 28

[Git Squash](#_6bhky4sapyj9) 29

[Splitting a Commit](#_9b1bnebfy2zu) 29

[Deleting a commit](#_w3qri07ksjy7) 29

[Aborting interactive rebase](#_ehmlksl99dsj) 29

[Git Reset and Checkout](#_3d5mqtm1eoxg) 29

## 

## About version control and Git

A version control system, or VCS, tracks the history of changes as people and teams collaborate on projects together. As developers make changes to the project, any earlier version of the project can be recovered at any time.

Developers can review project history to find out:

Which changes were made?

Who made the changes?

When were the changes made?

Why were changes needed?

In a distributed version control system, every developer has a full copy of the project and project history. Unlike once popular centralized version control systems, DVCSs don't need a constant connection to a central repository. Git is the most popular distributed version control system. Git is commonly used for both open source and commercial software development, with significant benefits for individuals, teams and businesses.

## About repositories

A repository, or Git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as snapshots in time called commits. The commits can be organized into multiple lines of development called branches. Because Git is a DVCS, repositories are self-contained units and anyone who has a copy of the repository can access the entire codebase and its history.

## How GitHub works

GitHub hosts Git repositories and provides developers with tools to ship better code through command line features, issues (threaded discussions), pull requests, code review, or the use of a collection of free and for-purchase apps in the GitHub Marketplace. With collaboration layers like the GitHub flow, a community of 15 million developers, and an ecosystem with hundreds of integrations, GitHub changes the way software is built.

### Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as **delta-based** version control).



Figure 4. Storing data as changes to a base version of each file

Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.



Figure 5. Storing data as snapshots of the project over time

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We’ll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in [Git Branching](https://git-scm.com/book/en/v2/ch00/ch03-git-branching).

### Git Has Integrity

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can’t lose information in transit or get file corruption without Git being able to detect it.

### The Three States

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: **modified**, **staged**, and **committed**:

* Modified means that you have changed the file but have not committed it to your database yet.
* Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
* Committed means that the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.



## First-Time Git Setup

Now that you have Git on your system, you’ll want to do a few things to customize your Git environment. You should have to do these things only once on any given computer; they’ll stick around between upgrades. You can also change them at any time by running through the commands again.

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. [path]/etc/gitconfig file: Contains values applied to every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically. Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.
2. ~/.gitconfig or ~/.config/git/config file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the --global option, and this affects all of the repositories you work with on your system.
3. config file in the Git directory (that is, .git/config) of whatever repository you’re currently using: Specific to that single repository. You can force Git to read from and write to this file with the --local option, but that is in fact the default. Unsurprisingly, you need to be located somewhere in a Git repository for this option to work properly.

## 

## GitHub and the command line

-h - Help Flag

### Basic Git commands

1. **git init** initializes a brand new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.
2. **git clone** creates a local copy of a project that already exists remotely. The clone includes all the project's files, history, and branches.
3. **git add** stages a change. Git tracks changes to a developer's codebase, but it's necessary to stage and take a snapshot of the changes to include them in the project's history. This command performs staging, the first part of that two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project's history. Staging and committing separately gives developers complete control over the history of their project without changing how they code and work.
   1. git add . - stage all files
4. **git commit** saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a photo. Anything that's been staged with git add will become a part of the snapshot with git commit.
5. **git status** shows the status of changes as untracked, modified, or staged.
6. **git branch** shows the branches being worked on locally.
   1. git branch -u remote/branch\_name local\_branch\_name -> cause local branch to track remote branch
   2. git branch -u remote/branch\_name -> cause current branch to track remote branch
   3. git branch -d branch\_name -> delete local branch
7. **git merge** merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the main branch for deployment.
   1. git merge branch - merge branch with current branch
   2. git merge branch1 branch2 - merge branches branch1 and branch2 on top of the current branch
   3. git merge –no-commit branch - merge branch without committing
8. **git pull** updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.
9. **git push** updates the remote repository with any commits made locally to a branch.
   1. git push -u origin branch\_name - push to a branch name
   2. git push -u origin source\_branch:target\_branch - push a (non-default) branch to a default branch

### What is Origin?

In Git, "origin" is a shorthand name for the remote repository that a project was originally cloned from. More precisely, it is used instead of that original repository's URL - and thereby makes referencing much easier. Note that origin is by no means a "magical" name, but just a standard convention.

### Git Merge

merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the main branch for deployment.

git merge branch - merge branch with current branch

git merge branch1 branch2 - merge branches branch1 and branch2 on top of the current branch

git merge –no-commit branch - merge branch without committing

Merge Strategies

**ort**

This is the default merge strategy when pulling or merging one branch. This strategy can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge.

**ours**

This option forces conflicting hunks to be auto-resolved cleanly by favoring **our** version. Changes from the other tree that do not conflict with our side are reflected in the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the **ours** merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring **our** history contains all that happened in it.

**theirs**

This is the opposite of **ours**; note that, unlike **ours**, there is no **theirs** merge strategy to confuse this merge option with.

### Git Status

The main tool you use to determine which files are in which state is the git status command.

git status -s - Short status

### Git Diff

If the git status command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the git diff command. We’ll cover git diff in more detail later, but you’ll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit?

git diff --staged - This command compares your staged changes to your last commit:

It’s important to note that git diff by itself doesn’t show all changes made since your last commit — only changes that are still unstaged. If you’ve staged all of your changes, git diff will give you no output.

### Git Rm

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:

git rm --cached README

You can pass files, directories, and file-glob patterns to the git rm command. That means you can do things such as:

git rm log/\\*.log

Note the backslash (\) in front of the \*. This is necessary because Git does its own filename expansion in addition to your shell’s filename expansion. This command removes all files that have the .log extension in the log/ directory.

## 

### Ignoring Files

Often, you’ll have a class of files that you don’t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

$ cat .gitignore

\*.[oa]

\*~

## 

### Git Pull

update the local version of a repository from a remote.

Git pull does two things:

1. Updates the remote tracking branches for all other branches (Git Fetch)
2. Updates the current local working branch (Git Merge)

# General format

git pull OPTIONS REPOSITORY REFSPEC

# Pull from specific branch

git pull REMOTE-NAME BRANCH-NAME

Note

If you have uncommitted changes, the merge part of the git pull command **will fail and your local branch will be untouched.**

Thus, you should always commit your changes in a branch before pulling new commits from a remote repository.

The local repository has a linked remote repository

* Check this by executing git remote -v
* If there are multiple remotes, git pull might not be enough information. You might need to enter git pull origin or git pull upstream.

### fetching a remote PR (Pull Request) in to local repo

git fetch origin pull/ID/head:BRANCHNAME

## Git vs Github

So, taken all together: Git vs. GitHub… what’s the difference? Simply put, Git is a version control system that lets you manage and keep track of your source code history. GitHub is a cloud-based hosting service that lets you manage Git repositories. If you have open-source projects that use Git, then GitHub is designed to help you better manage them.

After all, in the world of programming, naming conventions aren’t always intuitive. That’s why it’s worth recognizing the connections and the differences in the similarly named Git and GitHub. Both Git and GitHub give programmers valuable version-control functionality so that they can build ongoing coding projects without being afraid of messing everything up. GitHub just takes things a little bit further than Git, offering more functionality and resources, as well as a place online to store and collaborate on projects.

## 

## Undoing things

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the --amend option:

$ git commit --amend

### Unstage file

git restore –staged file

### **Unmodifying a Modified File**

git restore file

## 

## Working with Remotes

git remote -v - Shows URL that Git has stored for the shortname to be used when reading and writing to that remote

git fetch remote - The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time. If your current branch is set up to track a remote branch (see the next section and Git Branching for more information), you can use the git pull command to automatically fetch and then merge that remote branch into your current branch.

git push - When you have your project at a point that you want to share, you have to push it upstream.

### Inspecting a Remote

If you want to see more information about a particular remote, you can use the git remote show <remote> command.

git remote show origin

### Renaming and Removing Remotes

You can run git remote rename to change a remote’s shortname.

git remote rename old new

## 

## Tagging

Like most VCSs, Git has the ability to tag specific points in a repository’s history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on). In this section, you’ll learn how to list existing tags, how to create and delete tags, and what the different types of tags are.

### Listing Your Tags

git tag

### Creating Tags

Git supports two types of tags: lightweight and annotated.

#### Annotated Tags

git tag -a name -m “message”

#### Lightweight Tags

git tag name

#### Tagging Later

You can also tag commits after you’ve moved past them.

git tag name commit\_id

#### Sharing Tags

By default, the git push command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run git push origin <tagname>.

If you have a lot of tags that you want to push up at once, you can also use the --tags option to the git push command. This will transfer all of your tags to the remote server that are not already there.

#### Deleting Tags

git tag -d name

Deleting remote tag:

git push origin –delete <tagname>

#### Checking out Tags

If you want to view the versions of files a tag is pointing to, you can do a git checkout of that tag, although this puts your repository in “detached HEAD” state, which has some ill side effects:

In “detached HEAD” state, if you make changes and then create a commit, the tag will stay the same, but your new commit won’t belong to any branch and will be unreachable, except by the exact commit hash. Thus, if you need to make changes — say you’re fixing a bug on an older version, for instance — you will generally want to create a branch:

git checkout -b version2 v2.0.0

If you do this and make a commit, your version2 branch will be slightly different than your v2.0.0 tag since it will move forward with your new changes, so do be careful.

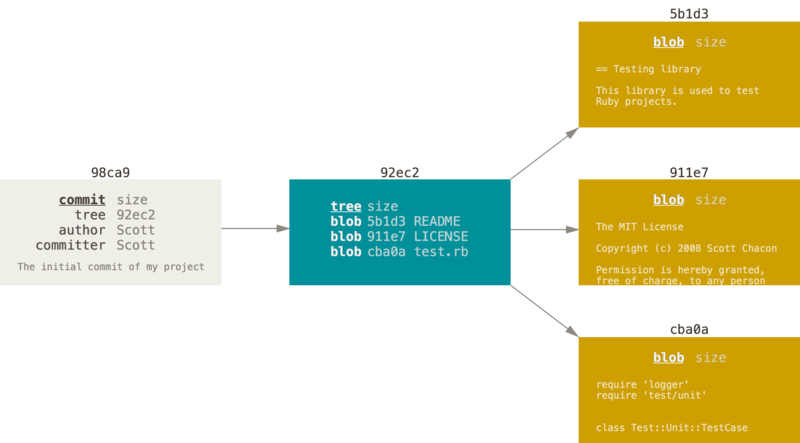
## Branching

Nearly every VCS has some form of branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

### Branches in a Nutshell

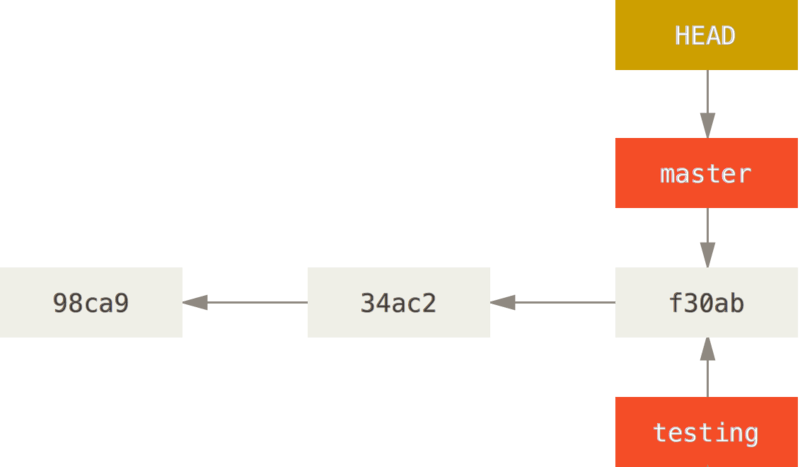
When you create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Your Git repository now contains five objects: three blobs (each representing the contents of one of the three files), one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.



A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you’re given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

How does Git know what branch you’re currently on? It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you’re currently on. In this case, you’re still on master. The git branch command only **created** a new branch — it didn’t switch to that branch.



### Basic Branching and Merging

git checkout -b <branch\_name>

git merge

git branch -d <branch\_name)

### Basic Merge Conflicts

If you changed the same part of the same file differently in the two branches you’re merging, Git won’t be able to merge them cleanly.

git mergetool

## Branch Management

### Git Branch

The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you’re currently on. To see which branches are already merged into the branch you’re on, you can run

git branch --merged:

To see all the branches that contain work you haven’t yet merged in, you can run

git branch --no-merged:

#### Changing a branch name

git branch --move bad-branch-name corrected-branch-name | Rename branch locally

git push --set-upstream origin corrected-branch-name | Push new branch

git push origin --delete bad-branch-name | Delete old branch

#### Changing the master branch name

git branch --move master main

git push --set-upstream origin main

Now you have a few more tasks in front of you to complete the transition:

* Any projects that depend on this one will need to update their code and/or configuration.
* Update any test-runner configuration files.
* Adjust build and release scripts.
* Redirect settings on your repo host for things like the repo’s default branch, merge rules, and other things that match branch names.
* Update references to the old branch in documentation.
* Close or merge any pull requests that target the old branch.

After you’ve done all these tasks, and are certain the main branch performs just as the master branch, you can delete the master branch:

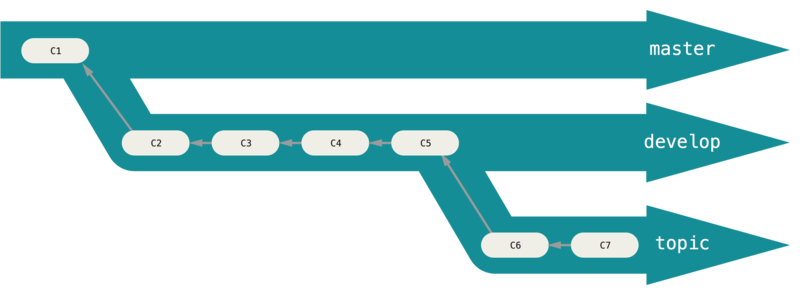
$ git push origin --delete master

### Branching Workflows

Long-Running Branches

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their master branch — possibly only code that has been or will be released. They have another parallel branch named develop or next that they work from or use to test stability — it isn’t necessarily always stable, but whenever it gets to a stable state, it can be merged into master.



#### Topic Branch

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. This is something you’ve likely never done with a VCS before because it’s generally too expensive to create and merge branches. But in Git it’s common to create, work on, merge, and delete branches several times a day.

You saw this in the last section with the iss53 and hotfix branches you created. You did a few commits on them and deleted them directly after merging them into your main branch. This technique allows you to context-switch quickly and completely — because your work is separated into silos where all the changes in that branch have to do with that topic, it’s easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they’re ready, regardless of the order in which they were created or worked on.

### Remote Branching

Remote-tracking branches are references to the state of remote branches. They’re local references that you can’t move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

Remote-tracking branch names take the form <remote>/<branch>. For instance, if you wanted to see what the master branch on your origin remote looked like as of the last time you communicated with it, you would check the origin/master branch. If you were working on an issue with a partner and they pushed up an iss53 branch, you might have your own local iss53 branch, but the branch on the server would be represented by the remote-tracking branch origin/iss53.

### Pushing

When you want to share a branch with the world, you need to push it up to a remote to which you have write access. Your local branches aren’t automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share. That way, you can use private branches for work you don’t want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named serverfix that you want to work on with others, you can push it up the same way you pushed your first branch. Run

git push <remote> <branch>:

### Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a “tracking branch” (and the branch it tracks is called an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in.

git checkout remote/branch\_name

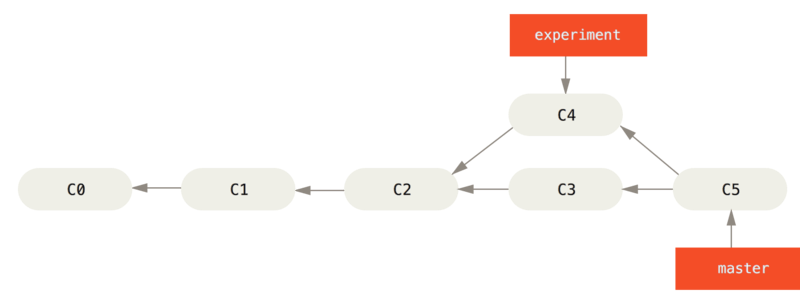
### Deleting Remote Branches

git push origin --delete serverfix

### Rebasing

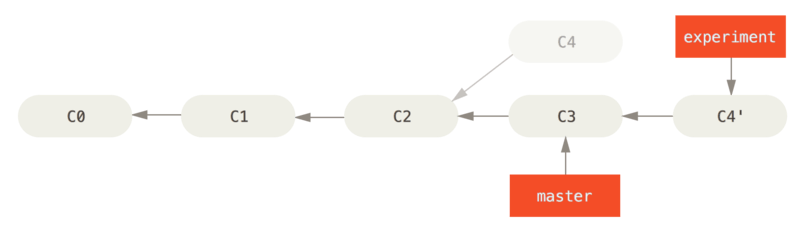
In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase. In this section you’ll learn what rebasing is, how to do it, why it’s a pretty amazing tool, and in what cases you won’t want to use it.

Initial:

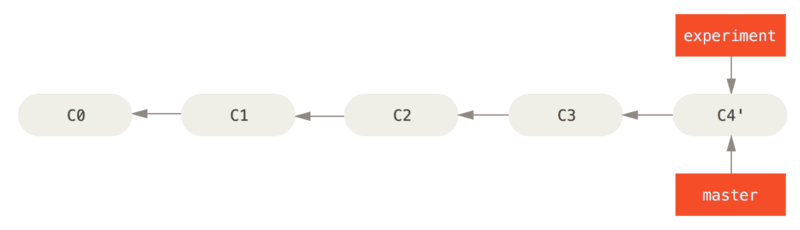


You can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called rebasing. With the rebase command, you can take all the changes that were committed on one branch and replay them on a different branch.

This operation works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.



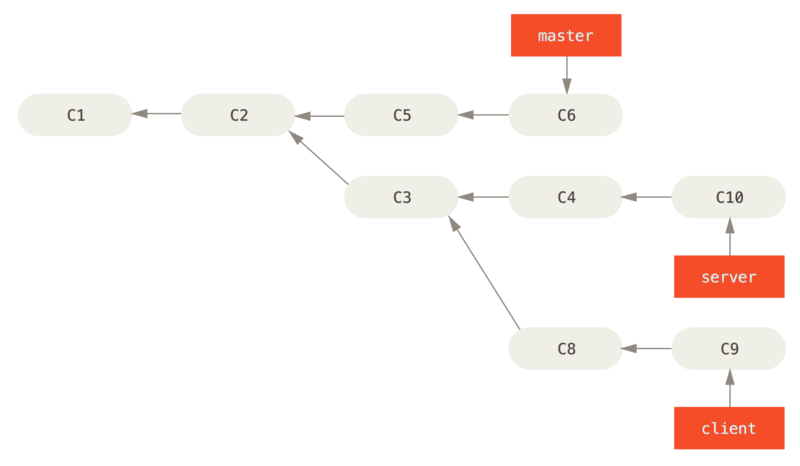
At this point, you can go back to the master branch and do a fast-forward merge.



Note that the snapshot pointed to by the final commit you end up with, whether it’s the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot — it’s only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

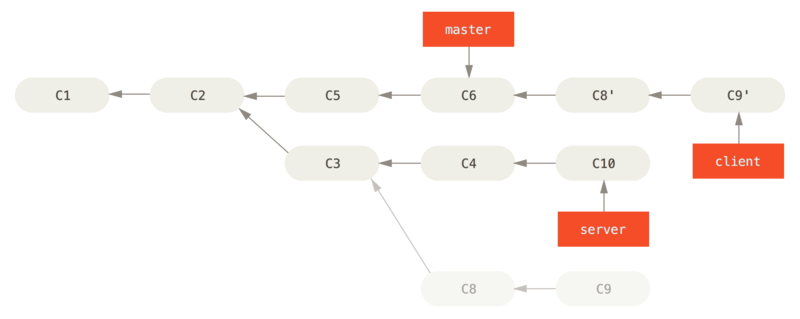
#### More Interesting Rebases

You branched a topic branch (server) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (client) and committed a few times. Finally, you went back to your server branch and did a few more commits.



Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it’s tested further. You can take the changes on client that aren’t on server (C8 and C9) and replay them on your master branch by using the --onto option of git rebase:

git rebase --onto master server client

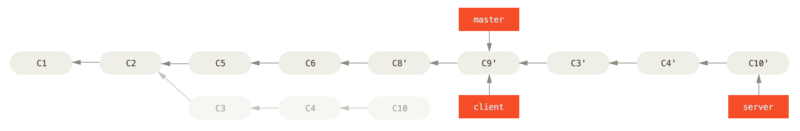


$ git checkout master

$ git merge client

Let’s say you decide to pull in your server branch as well. You can rebase the server branch onto the master branch without having to check it out first by running git rebase <basebranch> <topicbranch> — which checks out the topic branch (in this case, server) for you and replays it onto the base branch (master):

git rebase master server



$ git checkout master

$ git merge server

#### Rebase vs. Merge

Now that you’ve seen rebasing and merging in action, you may be wondering which one is better. Before we can answer this, let’s step back a bit and talk about what history means.

One point of view on this is that your repository’s commit history is a record of what actually happened. It’s a historical document, valuable in its own right, and shouldn’t be tampered with. From this angle, changing the commit history is almost blasphemous; you’re lying about what actually transpired. So what if there was a messy series of merge commits? That’s how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the story of how your project was made. You wouldn’t publish the first draft of a book, so why show your messy work? When you’re working on a project, you may need a record of all your missteps and dead-end paths, but when it’s time to show your work to the world, you may want to tell a more coherent story of how to get from A to B. People in this camp use tools like rebase and filter-branch to rewrite their commits before they’re merged into the mainline branch. They use tools like rebase and filter-branch, to tell the story in the way that’s best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you’ll see that it’s not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it’s up to you to decide which one is best for your particular situation.

You can get the best of both worlds: rebase local changes before pushing to clean up your work, but never rebase anything that you’ve pushed somewhere.

#### 

### Git Checkout

## Github

### The GitHub Flow

GitHub is designed around a particular collaboration workflow, centered on Pull Requests. This flow works whether you’re collaborating with a tightly-knit team in a single shared repository, or a globally-distributed company or network of strangers contributing to a project through dozens of forks. It is centered on the [Topic Branches](https://git-scm.com/book/en/v2/ch00/_topic_branch) workflow covered in [Git Branching](https://git-scm.com/book/en/v2/ch00/ch03-git-branching).

Here’s how it generally works:

1. Fork the project.
2. Create a topic branch from master.
3. Make some commits to improve the project.
4. Push this branch to your GitHub project.
5. Open a Pull Request on GitHub.
6. Discuss, and optionally continue committing.
7. The project owner merges or closes the Pull Request.
8. Sync the updated master back to your fork.

If you see something like Pull Request does not merge cleanly, you’ll want to fix your branch so that it turns green and the maintainer doesn’t have to do extra work.

You have two main options in order to do this. You can either rebase your branch on top of whatever the target branch is (normally the master branch of the repository you forked), or you can merge the target branch into your branch.

Most developers on GitHub will choose to do the latter, for the same reasons we just went over in the previous section. What matters is the history and the final merge, so rebasing isn’t getting you much other than a slightly cleaner history and in return is far more difficult and error prone.

If you want to merge in the target branch to make your Pull Request mergeable, you would add the original repository as a new remote, fetch from it, merge the main branch of that repository into your topic branch, fix any issues and finally push it back up to the same branch you opened the Pull Request on.

## Git Advanced tools

### Revision Selection

You can obviously refer to any single commit by its full, 40-character SHA-1 hash, but there are more human-friendly ways to refer to commits as well. This section outlines the various ways you can refer to any commit.

Short SHA-1

Git is smart enough to figure out what commit you’re referring to if you provide the first few characters of the SHA-1 hash, as long as that partial hash is at least four characters long and unambiguous; that is, no other object in the object database can have a hash that begins with the same prefix.

### RefLog Shortnames

One of the things Git does in the background while you’re working away is keep a “reflog” — a log of where your HEAD and branch references have been for the last few months.

You can see your reflog by using git reflog:

### Double Dot

Say you want to see what is in your experiment branch that hasn’t yet been merged into your master branch. You can ask Git to show you a log of just those commits with master..experiment — that means “all commits reachable from experiment that aren’t reachable from master.

git log master..experiment

### Triple Dot

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by either of two references but not by both of them.

### Interactive Staging

In this section, you’ll look at a few interactive Git commands that can help you craft your commits to include only certain combinations and parts of files. These tools are helpful if you modify a number of files extensively, then decide that you want those changes to be partitioned into several focused commits rather than one big messy commit. This way, you can make sure your commits are logically separate changesets and can be reviewed easily by the developers working with you.

### Stashing, Cleaning

TL:DR

git stash list - list stash

git stash push - push stash

git stash apply - apply stash

git stash drop - drop stash

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

To push a new stash onto your stack, run

git stash or git stash push:

You can now see that your working directory is clean:

At this point, you can switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you’ve stored, you can use git stash list:

You can reapply the one you just stashed by using the command shown in the help output of the original stash command:

git stash apply.

If you want to apply one of the older stashes, you can specify it by naming it, like this: git stash apply stash@{2}. If you don’t specify a stash, Git assumes the most recent stash and tries to apply it

The apply option only tries to apply the stashed work — you continue to have it on your stack. To remove it, you can run git stash drop with the name of the stash to remove:

#### Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you’ve since modified, you’ll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run git stash branch <new branchname>, which creates a new branch for you with your selected branch name, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

### Git Grep

Git ships with a command called grep that allows you to easily search through any committed tree, the working directory, or even the index for a string or regular expression. For the examples that follow, we’ll search through the source code for Git itself.

By default, git grep will look through the files in your working directory. As a first variation, you can use either of the -n or --line-number options to print out the line numbers where Git has found matches

### Git Squash

It’s also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message

git rebase -i

pick f7f3f6d Change my name a bit

squash 310154e Update README formatting and add blame

squash a5f4a0d Add cat-file

When you save that, you have a single commit that introduces the changes of all three previous commits.

### Splitting a Commit

git rebase -i

pick f7f3f6d Change my name a bit

edit 310154e Update README formatting and add blame

pick a5f4a0d Add cat-file

drop to the command line

$ git reset HEAD^

$ git add README

$ git commit -m 'Update README formatting'

$ git add lib/simplegit.rb

$ git commit -m 'Add blame'

$ git rebase --continue

### Deleting a commit

git rebase -i

pick 461cb2a This commit is OK

drop 5aecc10 This commit is broken

### Aborting interactive rebase

Inside: git rebase –abort

After: git reflog

### Git Reset and Checkout

The Three Trees

An easier way to think about reset and checkout is through the mental frame of Git being a content manager of three different trees.

| Tree | Role |
| --- | --- |
| HEAD | Last commit snapshot, next parent |
| Index | Proposed next commit snapshot |
| Working Directory | Sandbox |

The HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It’s generally simplest to think of HEAD as the snapshot of your last commit on that branch.

The Index

The index is your proposed next commit. We’ve also been referring to this concept as Git’s “Staging Area” as this is what Git looks at when you run git commit.

The Working Directory

Finally, you have your working directory (also commonly referred to as the “working tree”). The other two trees store their content in an efficient but inconvenient manner, inside the .git folder. The working directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the working directory as a sandbox, where you can try changes out before committing them to your staging area (index) and then to history.

The Role of Reset

Step 1: Move HEAD

The first thing reset will do is move what HEAD points to. This isn’t the same as changing HEAD itself (which is what checkout does); reset moves the branch that HEAD is pointing to.

This means if HEAD is set to the master branch (i.e. you’re currently on the master branch), running git reset 9e5e6a4 will start by making master point to 9e5e6a4.

Step 2: Updating the Index (--mixed)

The next thing reset will do is to update the index with the contents of whatever snapshot HEAD now points to.

Step 3: Updating the Working Directory (--hard)

The third thing that reset will do is to make the working directory look like the index. If you use the --hard option, it will continue to this stage.

So let’s think about what just happened. You undid your last commit, the git add and git commit commands, and all the work you did in your working directory.