source: https://git-scm.com/book/en/v2

# notes on git

What is Git?

So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as CVS, Subversion or Perforce — doing so will help you avoid subtle confusion when using the tool. Even though Git’s user interface is fairly similar to these other VCSs, Git stores and thinks about information in a very different way, and understanding these differences will help you avoid becoming confused while using it.

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).

Storing data as changes to a base version of each file

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.

Git stores data as snapshots of the project over time

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.

Nearly Every Operation Is Local:

Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network. If you’re used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

For example, to browse the history of the project, Git doesn’t need to go out to the server to get the history and display it for you — it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

This also means that there is very little you can’t do if you’re offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily (to your local copy, remember?) until you get to a network connection to upload. If you go home and can’t get your VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful. In Perforce, for example, you can’t do much when you aren’t connected to the server; in Subversion and CVS, you can edit files, but you can’t commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

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 mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373

You will see these hash values all over the place in Git because it uses them so much. In fact, GIT STORES EVERYTHING IN ITS DATABASE NOT BY FILE NAME BUT BY THE HASH VALUE OF ITS CONTENTS.

Git Generally Only Adds Data:

When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As with any VCS, you can lose or mess up changes you haven’t committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see Undoing Things

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(staged means you have added it, using "git add" command).

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.

The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

The basic Git workflow goes something like this:

1:You modify files in your working tree.

2:You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.

3:You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

FILES THAT ARE ADDED OR CHANGED OR NOT STAGED DIRECTLY, WE HAVE TO ADD THEM BY USING GIT ADD <CHANGED FILES SEPERATED BY SPACE>

UNLESS THE ABOVE STEP IS DONE, THE CHANGED FILES WILL BE IN UNTRACKED STATE

If a particular version of a file is in the Git directory, it’s considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified. In Git Basics, you’ll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

The Command Line

There are a lot of different ways to use Git. There are the original command-line tools, and there are many graphical user interfaces of varying capabilities. For this book, we will be using Git on the command line. For one, the command line is the only place you can run all Git commands — most of the GUIs implement only a partial subset of Git functionality for simplicity. If you know how to run the command-line version, you can probably also figure out how to run the GUI version, while the opposite is not necessarily true. Also, while your choice of graphical client is a matter of personal taste, all users will have the command-line tools installed and available.

git --exec-path->path where git is installed in linux

git init:

create a folder where you want to perform repository related operations and then enter in to the folder and type "git init"

this will create a .git folder which will have different details such as config and other details

config file will store the values(of the user such as email,username and so on...) related to the user of the repository

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.

Each level overrides values in the previous level, so values in .git/config trump those in [path]/etc/gitconfig.

On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:\Users\$USER for most people). It also still looks for [path]/etc/gitconfig, although it’s relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are using version 2.x or later of Git for Windows, there is also a system-level config file at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer. This config file can only be changed by git config -f <file> as an admin.

You can view all of your settings and where they are coming from using:

$ git config --list --show-origin

Your Identity:

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

commands:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Again, you need to do this only once if you pass the --global option, because then Git will always use that information for anything you do on that system. If you want to override this with a different name or email address for specific projects, you can run the command without the --global option when you’re in that project.

Many of the GUI tools will help you do this when you first run them.

Your Editor:

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system’s default editor.

If you want to use a different text editor, such as Emacs, you can do the following:

$ git config --global core.editor emacs

On a Windows system, if you want to use a different text editor, you must specify the full path to its executable file. This can be different depending on how your editor is packaged.

In the case of Notepad++, a popular programming editor, you are likely to want to use the 32-bit version, since at the time of writing the 64-bit version doesn’t support all plug-ins. If you are on a 32-bit Windows system, or you have a 64-bit editor on a 64-bit system, you’ll type something like this:

$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

Note:

Vim, Emacs and Notepad++ are popular text editors often used by developers on Unix-based systems like Linux and macOS or a Windows system. If you are using another editor, or a 32-bit version, please find specific instructions for how to set up your favorite editor with Git in git config core.editor commands.

Warning:

You may find, if you don’t setup your editor like this, you get into a really confusing state when Git attempts to launch it. An example on a Windows system may include a prematurely terminated Git operation during a Git initiated edit.

to navigate to git installed location- type \\wsl$ in the path panel in file manager

Your default branch name

By default Git will create a branch called master when you create a new repository with git init. From Git version 2.28 onwards, you can set a different name for the initial branch.

To set main as the default branch name do:

$ git config --global init.defaultBranch main

Checking Your Settings

If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find at that point:

$ git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

You may see keys more than once, because Git reads the same key from different files ([path]/etc/gitconfig and ~/.gitconfig, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key’s value is by typing git config <key>:

$ git config user.name

John Doe

-Note:

Since Git might read the same configuration variable value from more than one file, it’s possible that you have an unexpected value for one of these values and you don’t know why. In cases like that, you can query Git as to the origin for that value, and it will tell you which configuration file had the final say in setting that value:

$ git config --show-origin rerere.autoUpdate

file:/home/johndoe/.gitconfig false

Getting Help

If you ever need help while using Git, there are three equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

For example, you can get the manpage help for the git config command by running this:

$ git help config

These commands are nice because you can access them anywhere, even offline. If the manpages and this book aren’t enough and you need in-person help, you can try the #git, #github, or #gitlab channels on the Libera Chat IRC server, which can be found at https://libera.chat/. These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.

In addition, if you don’t need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can ask for the more concise “help” output with the -h option

Getting a Git Repository

You typically obtain a Git repository in one of two ways:

1:You can take a local directory that is currently not under version control, and turn it into a Git repository, or

2:You can clone an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.

Initializing a Repository in an Existing Directory

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project’s directory. If you’ve never done this, it looks a little different depending on which system you’re running:

for Linux:

$ cd /home/user/my\_project

for macOS:

$ cd /Users/user/my\_project

for Windows:

$ cd C:/Users/user/my\_project

and type:

$ git init

This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet. See Git Internals for more information about exactly what files are contained in the .git directory you just created.

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

$ git add \*.c

$ git add LICENSE

$ git commit -m 'Initial project version'

We’ll go over what these commands do in just a minute. At this point, you have a Git repository with tracked files and an initial commit.

Cloning an Existing Repository

If you want to get a copy of an existing Git repository — for example, a project you’d like to contribute to — the command you need is git clone. If you’re familiar with other VCSs such as Subversion, you’ll notice that the command is "clone" and not "checkout". This is an important distinction — instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there -see Getting Git on a Server for more details).

You clone a repository with git clone <url>. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

$ git clone https://github.com/libgit2/libgit2

That creates a directory named libgit2, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2 directory that was just created, you’ll see the project files in there, ready to be worked on or used.

If you want to clone the repository into a directory named something other than libgit2, you can specify the new directory name as an additional argument:

$ git clone https://github.com/libgit2/libgit2 mylibgit

That command does the same thing as the previous one, but the target directory is called mylibgit.

Git has a number of different transfer protocols you can use. The previous example uses the https:// protocol, but you may also see git:// or user@server:path/to/repo.git, which uses the SSH transfer protocol. Getting Git on a Server will introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

Recording Changes to the Repository

At this point, you should have a bona fide Git repository on your local machine, and a checkout or working copy of all of its files in front of you. Typically, you’ll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

REMEMBER THAT EACH FILE IN YOUR WORKING DIRECTORY CAN BE IN ONE OF TWO STATES: TRACKED OR UNTRACKED. TRACKED FILES ARE FILES THAT WERE IN THE LAST SNAPSHOT, AS WELL AS ANY NEWLY STAGED FILES; THEY CAN BE UNMODIFIED, MODIFIED, OR STAGED. IN SHORT, TRACKED FILES ARE FILES THAT GIT KNOWS ABOUT.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

-To stop tracking a file you need to remove it from the index. This can be achieved with this command.

git rm --cached <file>

If you want to remove a whole folder, you need to remove all files in it recursively.

git rm -r --cached <folder>

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn’t see any untracked files, or they would be listed here. Finally, the command tells you which branch you’re on and informs you that it has not diverged from the same branch on the server. For now, that branch is always master, which is the default; you won’t worry about it here. Git Branching will go over branches and references in detail.

Let’s say you add a new file to your project, a simple README file. If the file didn’t exist before, and you run git status, you see your untracked file like so:

$ echo 'My Project' > README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

You can see that your new README file is untracked, because it’s under the “Untracked files” heading in your status output. Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit), and which hasn’t yet been staged; Git won’t start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including README, so let’s start tracking the file

Tracking New Files:

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

$ git add README

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

You can tell that it’s staged because it’s under the “Changes to be committed” heading. If you commit at this point, the version of the file at the time you ran git add is what will be in the subsequent historical snapshot. You may recall that when you ran git init earlier, you then ran git add <files> — that was to begin tracking files in your directory. The git add command takes a path name for either a file or a directory; if it’s a directory, the command adds all the files in that directory recursively.

Staging Modified Files:

Let’s change a file that was already tracked. If you change a previously tracked file called CONTRIBUTING.md and then run your git status command again, you get something that looks like this:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

The CONTRIBUTING.md file appears under a section named “Changes not staged for commit” — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the git add command. git add is a multipurpose command you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add precisely this content to the next commit” rather than “add this file to the project”. Let’s run git add now to stage the CONTRIBUTING.md file, and then run git status again:

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it. You open it again and make that change, and you’re ready to commit. However, let’s run git status one more time:

$ vim CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

What the heck? Now CONTRIBUTING.md is listed as both staged and unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the git add command. If you commit now, the version of CONTRIBUTING.md as it was when you last ran the git add command is how it will go into the commit, not the version of the file as it looks in your working directory when you run git commit. If you modify a file after you run git add, you have to run git add again to stage the latest version of the file:

$ git add CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

modified: CONTRIBUTING.md

Short Status

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

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]

\*~

The first line tells Git to ignore any files ending in “.o” or “.a” — object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a .gitignore file for your new repository before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository.

The rules for the patterns you can put in the .gitignore file are as follows:

Blank lines or lines starting with # are ignored.

Standard glob patterns work, and will be applied recursively throughout the entire working tree.

You can start patterns with a forward slash (/) to avoid recursivity.

You can end patterns with a forward slash (/) to specify a directory.

You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use. An asterisk (\*) matches zero or more characters; [abc] matches any character inside the brackets (in this case a, b, or c); a question mark (?) matches a single character; and brackets enclosing characters separated by a hyphen ([0-9]) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; a/\*\*/z would match a/z, a/b/z, a/b/c/z, and so on.

Here is another example .gitignore file:

# IGNORE ALL .A FILES

\*.a

# BUT DO TRACK LIB.A, EVEN THOUGH YOU'RE IGNORING .A FILES ABOVE

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# IGNORE ALL FILES IN ANY DIRECTORY NAMED BUILD

build/

# IGNORE DOC/NOTES.TXT, BUT NOT DOC/SERVER/ARCH.TXT(notice the way the below command is declared)

doc/\*.txt

# IGNORE ALL .PDF FILES IN THE DOC/ DIRECTORY AND ANY OF ITS SUBDIRECTORIES(notice the difference between above command and below command)

DOC/\*\*/\*.PDF

Viewing Your Staged and Unstaged Changes

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? Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed — the patch, as it were.

Let’s say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your git status command, you once again see something like this:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

To see what you’ve changed but not yet staged, type git diff with no other arguments:

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+My Project

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.

For another example, if you stage the CONTRIBUTING.md file and then edit it, you can use git diff to see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

$ git add CONTRIBUTING.md

$ echo '# test line' >> CONTRIBUTING.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: CONTRIBUTING.md

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

Now you can use git diff to see what is still unstaged:

$ git diff

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 643e24f..87f08c8 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -119,3 +119,4 @@ at the

## Starter Projects

See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).

+# test line

and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):

$ git diff --cached

diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md

index 8ebb991..643e24f 100644

--- a/CONTRIBUTING.md

+++ b/CONTRIBUTING.md

@@ -65,7 +65,8 @@ branch directly, things can get messy.

Please include a nice description of your changes when you submit your PR;

if we have to read the whole diff to figure out why you're contributing

in the first place, you're less likely to get feedback and have your change

-merged in.

+merged in. Also, split your changes into comprehensive chunks if your patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

Committing Your Changes:

Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

$ git commit -m "Story 182: fix benchmarks for speed"

[master 463dc4f] Story 182: fix benchmarks for speed

2 files changed, 2 insertions(+)

create mode 100644 README

NOW YOU’VE CREATED YOUR FIRST COMMIT! YOU CAN SEE THAT THE COMMIT HAS GIVEN YOU SOME OUTPUT ABOUT ITSELF: WHICH BRANCH YOU COMMITTED TO (MASTER), WHAT SHA-1 CHECKSUM THE COMMIT HAS (463DC4F), HOW MANY FILES WERE CHANGED, AND STATISTICS ABOUT LINES ADDED AND REMOVED IN THE COMMIT.

REMEMBER THAT THE COMMIT RECORDS THE SNAPSHOT YOU SET UP IN YOUR STAGING AREA. ANYTHING YOU DIDN’T STAGE IS STILL SITTING THERE MODIFIED; YOU CAN DO ANOTHER COMMIT TO ADD IT TO YOUR HISTORY. EVERY TIME YOU PERFORM A COMMIT, YOU’RE RECORDING A SNAPSHOT OF YOUR PROJECT THAT YOU CAN REVERT TO OR COMPARE TO LATER.

Skipping the Staging Area

Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part:

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m 'Add new benchmarks'

[master 83e38c7] Add new benchmarks

1 file changed, 5 insertions(+), 0 deletions(-)

Notice how you don’t have to run git add on the CONTRIBUTING.md file in this case before you commit. That’s because the -a flag includes all changed files. This is convenient, but be careful; sometimes this flag will cause you to include unwanted changes.

Removing Files

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.

If you simply remove the file from your working directory, it shows up under the “Changes not staged for commit” (that is, unstaged) area of your git status output:

$ rm PROJECTS.md

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")

Then, if you run git rm, it stages the file’s removal:

$ git rm PROJECTS.md

rm 'PROJECTS.md'

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: PROJECTS.md

The next time you commit, the file will be gone and no longer tracked. If you modified the file or had already added it to the staging area, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

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. Or, you can do something like this:

$ git rm \\*~

This command removes all files whose names end with a ~.

Moving Files

Unlike many other VCSs, Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we’ll deal with detecting file movement a bit later.

Thus it’s a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

$ git mv file\_from file\_to

and it works fine. In fact, if you run something like this and look at the status, you’ll see that Git considers it a renamed file:

$ git mv README.md README

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

However, this is equivalent to running something like this:

$ mv README.md README

$ git rm README.md

$ git add README

Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command. The only real difference is that git mv is one command instead of three — it’s a convenience function. More importantly, you can use any tool you like to rename a file, and address the add/rm later, before you commit.

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you’ll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author’s name and email, the date written, and the commit message.

A huge number and variety of options to the git log command are available to show you exactly what you’re looking for. Here, we’ll show you some of the most popular.

One of the more helpful options is -p or --patch, which shows the difference (the patch output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with git log. For example, if you want to see some abbreviated stats for each commit, you can use the --stat option:

As you can see, the --stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is --pretty. This option changes the log output to formats other than the default. A few prebuilt option values are available for you to use. The oneline value for this option prints each commit on a single line, which is useful if you’re looking at a lot of commits. In addition, the short, full, and fuller values show the output in roughly the same format but with less or more information, respectively:

$ git log --pretty=oneline

ca82a6dff817ec66f44342007202690a93763949 Change version number

085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test

a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit

The most interesting option value is format, which allows you to specify your own log output format. This is especially useful when you’re generating output for machine parsing — because you specify the format explicitly, you know it won’t change with updates to Git:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : Change version number

085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test

a11bef0 - Scott Chacon, 6 years ago : Initial commit

%h,%an,%ar %s are specifiers and there are variety of specifiers that are used

along with log option we also have --graph option,This option adds a nice little ASCII graph showing your branch and merge history

Undoing Things:

At any stage, you may want to undo something. Here, we’ll review a few basic tools for undoing changes that you’ve made. BE CAREFUL, BECAUSE YOU CAN’T ALWAYS UNDO SOME OF THESE UNDOS. THIS IS ONE OF THE FEW AREAS IN GIT WHERE YOU MAY LOSE SOME WORK IF YOU DO IT WRONG.

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

This command takes your staging area and uses it for the commit. If you’ve made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you’ll change is your commit message.

The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

$ git commit -m 'Initial commit'

$ git add forgotten\_file

$ git commit --amend

after entering the git commit --amend it will take you to the text editor to see the last entered message for the commit and if you change it,even in the git log command you can see the newly amended commit message, but not the message before that.You end up with a single commit — the second commit replaces the results of the first.

Note

It’s important to understand that when you’re amending your last commit, you’re not so much fixing it as replacing it entirely with a new, improved commit that pushes the old commit out of the way and puts the new commit in its place. Effectively, it’s as if the previous commit never happened, and it won’t show up in your repository history.

The obvious value to amending commits is to make minor improvements to your last commit, without cluttering your repository history with commit messages of the form, “Oops, forgot to add a file” or “Darn, fixing a typo in last commit”.

ONLY AMEND COMMITS THAT ARE STILL LOCAL AND HAVE NOT BEEN PUSHED SOMEWHERE. AMENDING PREVIOUSLY PUSHED COMMITS AND FORCE PUSHING THE BRANCH WILL CAUSE PROBLEMS FOR YOUR COLLABORATORS. FOR MORE ON WHAT HAPPENS WHEN YOU DO THIS AND HOW TO RECOVER IF YOU’RE ON THE RECEIVING END READ

Unstaging a Staged File

The next two sections demonstrate how to work with your staging area and working directory changes. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both. How can you unstage one of the two? The git status command reminds you:

$ git add \*

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

modified: CONTRIBUTING.md

Right below the “Changes to be committed” text, it says use git reset HEAD <file>…​ to unstage. So, let’s use that advice to unstage the CONTRIBUTING.md file:

$ git reset HEAD CONTRIBUTING.md

Unstaged changes after reset:

M CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

The command is a bit strange, but it works. The CONTRIBUTING.md file is modified but once again unstaged.

Note:

IT’S TRUE THAT GIT RESET CAN BE A DANGEROUS COMMAND, ESPECIALLY IF YOU PROVIDE THE --HARD FLAG. However, in the scenario described above, the file in your working directory is not touched, so it’s relatively safe.

Unmodifying a Modified File

What if you realize that you don’t want to keep your changes to the CONTRIBUTING.md file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

It tells you pretty explicitly how to discard the changes you’ve made. Let’s do what it says:

$ git checkout -- CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

You can see that the changes have been reverted.

Important

It’s important to understand that git checkout -- <file> is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the last staged or committed version. Don’t ever use this command unless you absolutely know that you don’t want those unsaved local changes.

If you would like to keep the changes you’ve made to that file but still need to get it out of the way for now, we’ll go over stashing and branching in Git Branching; these are generally better ways to go.

Remember, anything that is committed in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered (see Data Recovery for data recovery). However, anything you lose that was never committed is likely never to be seen again.

Undoing things with git restore

Git version 2.23.0 introduced a new command: git restore. It’s basically an alternative to git reset which we just covered. From Git version 2.23.0 onwards, Git will use git restore instead of git reset for many undo operations.

Let’s retrace our steps, and undo things with git restore instead of git reset.

-Unstaging a Staged File with git restore

The next two sections demonstrate how to work with your staging area and working directory changes with git restore. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both. How can you unstage one of the two? The git status command reminds you:

$ git add \*

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: CONTRIBUTING.md

renamed: README.md -> README

Right below the “Changes to be committed” text, it says use git restore --staged <file>…​ to unstage. So, let’s use that advice to unstage the CONTRIBUTING.md file:

$ git restore --staged CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

The CONTRIBUTING.md file is modified but once again unstaged.

-Unmodifying a Modified File with git restore:

What if you realize that you don’t want to keep your changes to the CONTRIBUTING.md file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

It tells you pretty explicitly how to discard the changes you’ve made. Let’s do what it says:

$ git restore CONTRIBUTING.md

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: README.md -> README

Important

It’s important to understand that git restore <file> is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the last staged or committed version. Don’t ever use this command unless you absolutely know that you don’t want those unsaved local changes.

Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we’ll cover some of these remote-management skills.

Note

Remote repositories can be on your local machine.

IT IS ENTIRELY POSSIBLE THAT YOU CAN BE WORKING WITH A “REMOTE” REPOSITORY THAT IS, IN FACT, ON THE SAME HOST YOU ARE. THE WORD “REMOTE” DOES NOT NECESSARILY IMPLY THAT THE REPOSITORY IS SOMEWHERE ELSE ON THE NETWORK OR INTERNET, ONLY THAT IT IS ELSEWHERE. WORKING WITH SUCH A REMOTE REPOSITORY WOULD STILL INVOLVE ALL THE STANDARD PUSHING, PULLING AND FETCHING OPERATIONS AS WITH ANY OTHER REMOTE.

-Showing Your Remotes

To see which remote servers you have configured, you can run the git remote command. It lists the shortnames of each remote handle you’ve specified. If you’ve cloned your repository, you should at least see origin — that is the default name Git gives to the server you cloned from:

$ git clone https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

$ cd ticgit

$ git remote

origin

You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

If you have more than one remote, the command lists them all. For example, a repository with multiple remotes for working with several collaborators might look something like this.

$ cd grit

$ git remote -v

bakkdoor https://github.com/bakkdoor/grit (fetch)

bakkdoor https://github.com/bakkdoor/grit (push)

cho45 https://github.com/cho45/grit (fetch)

cho45 https://github.com/cho45/grit (push)

defunkt https://github.com/defunkt/grit (fetch)

defunkt https://github.com/defunkt/grit (push)

koke git://github.com/koke/grit.git (fetch)

koke git://github.com/koke/grit.git (push)

origin git@github.com:mojombo/grit.git (fetch)

origin git@github.com:mojombo/grit.git (push)

This means we can pull contributions from any of these users pretty easily. We may additionally have permission to push to one or more of these, though we can’t tell that here.

Notice that these remotes use a variety of protocols; we’ll cover more about this in Getting Git on a Server.

Adding Remote Repositories

We’ve mentioned and given some demonstrations of how the git clone command implicitly adds the origin remote for you. Here’s how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run git remote add <shortname> <url>:

$ git remote

origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

Now you can use the string pb on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don’t yet have in your repository, you can run git fetch pb:

$ git fetch pb

remote: Counting objects: 43, done.

remote: Compressing objects: 100% (36/36), done.

remote: Total 43 (delta 10), reused 31 (delta 5)

Unpacking objects: 100% (43/43), done.

From https://github.com/paulboone/ticgit

\* [new branch] master -> pb/master (this is )

\* [new branch] ticgit -> pb/ticgit

Paul’s master branch is now accessible locally as pb/master — you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it. We’ll go over what branches are and how to use them in much more detail in Git Branching.

Fetching and Pulling from Your Remotes

pull=fetch+merge

As you just saw, to get data from your remote projects, you can run:

$ 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.

f you clone a repository, the command automatically adds that remote repository under the name “origin”. So, git fetch origin fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the git fetch command only downloads the data to your local repository — it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

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. This may be an easier or more comfortable workflow for you; and by default, the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: git push <remote> <branch>. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you’ve done back up to the server:

$ git push origin master

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You’ll have to fetch their work first and incorporate it into yours before you’ll be allowed to push. See Git Branching for more detailed information on how to push to remote servers.

Renaming and Removing Remotes

You can run git remote rename to change a remote’s shortname. For instance, if you want to rename pb to paul, you can do so with git remote rename:

$ git remote rename pb paul

$ git remote

origin

paul

It’s worth mentioning that this changes all your remote-tracking branch names, too. What used to be referenced at pb/master is now at paul/master.

If you want to remove a remote for some reason — you’ve moved the server or are no longer using a particular mirror, or perhaps a contributor isn’t contributing anymore — you can either use git remote remove or git remote rm:

$ git remote remove paul

$ git remote

origin

Once you delete the reference to a remote this way, all remote-tracking branches and configuration settings associated with that remote are also deleted

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

Listing the existing tags in Git is straightforward. Just type git tag (with optional -l or --list):

$ git tag

v1.0

v2.0

THIS COMMAND LISTS THE TAGS IN ALPHABETICAL ORDER; THE ORDER IN WHICH THEY ARE DISPLAYED HAS NO REAL IMPORTANCE.

You can also search for tags that match a particular pattern. The Git source repo, for instance, contains more than 500 tags. If you’re interested only in looking at the 1.8.5 series, you can run this:

$ git tag -l "v1.8.5\*"

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

v1.8.5-rc3

v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

Note

Listing tag wildcards requires -l or --list option

If you want just the entire list of tags, running the command git tag implicitly assumes you want a listing and provides one; the use of -l or --list in this case is optional.

If, however, you’re supplying a wildcard pattern to match tag names, the use of -l or --list is mandatory.

Creating Tags

Git supports two types of tags: lightweight and annotated.

A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It’s generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don’t want to keep the other information, lightweight tags are available too.

Annotated Tags

Creating an annotated tag in Git is simple. The easiest way is to specify -a when you run the tag command:

$ git tag -a v1.4 -m "my version 1.4"

$ git tag

v0.1

v1.3

v1.4

The -m specifies a tagging message, which is stored with the tag. If you don’t specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the git show command:

$ git show v1.4

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

That shows the tagger information, the date the commit was tagged, and the annotation message before showing the commit information.

Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file — no other information is kept. To create a lightweight tag, don’t supply any of the -a, -s, or -m options, just provide a tag name:

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

This time, if you run git show on the tag, you don’t see the extra tag information. The command just shows the commit:

$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

for remaining information on tags->https://git-scm.com/book/en/v2/Git-Basics-Tagging#:~:text=Change%20version%20number-,Tagging,-Later

Git Aliases

Before we move on to the next chapter, we want to introduce a feature that can make your Git experience simpler, easier, and more familiar: aliases. For clarity’s sake, we won’t be using them anywhere else in this book, but if you go on to use Git with any regularity, aliases are something you should know about.

Git doesn’t automatically infer your command if you type it in partially. If you don’t want to type the entire text of each of the Git commands, you can easily set up an alias for each command using git config. Here are a couple of examples you may want to set up:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

This means that, for example, instead of typing git commit, you just need to type git ci. As you go on using Git, you’ll probably use other commands frequently as well; don’t hesitate to create new aliases.

This technique can also be very useful in creating commands that you think should exist. For example, to correct the usability problem you encountered with unstaging a file, you can add your own unstage alias to Git:

$ git config --global alias.unstage 'reset HEAD --'

This makes the following two commands equivalent:

$ git unstage fileA

$ git reset HEAD -- fileA

This seems a bit clearer. It’s also common to add a last command, like this:

$ git config --global alias.last 'log -1 HEAD'

This way, you can see the last commit easily:

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

Test for current head

Signed-off-by: Scott Chacon <schacon@example.com>

As you can tell, Git simply replaces the new command with whatever you alias it for. However, maybe you want to run an external command, rather than a Git subcommand. In that case, you start the command with a ! character. This is useful if you write your own tools that work with a Git repository. We can demonstrate by aliasing git visual to run gitk:

$ git config --global alias.visual '!gitk'

Git Branching - Branches in a Nutshell(the link has wonderful pictorial representation, better to go through the link (https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell))

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.

Some people refer to Git’s branching model as its “killer feature,” and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

Branches in a Nutshell

To really understand the way Git does branching, we need to take a step back and examine how Git stores its data.

As you may remember from What is Git?, Git doesn’t store data as a series of changesets or differences, but instead as a series of snapshots.

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author’s name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

To visualize this, let’s assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned in What is Git?), stores that version of the file in the Git repository (Git refers to them as blobs), and adds that checksum to the staging area:

$ git add README test.rb LICENSE

$ git commit -m 'Initial commit'

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 commit and its tree

Figure 9. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

Commits and their parents

Figure 10. Commits and their parents

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.

Note

THE “MASTER” BRANCH IN GIT IS NOT A SPECIAL BRANCH. IT IS EXACTLY LIKE ANY OTHER BRANCH. THE ONLY REASON NEARLY EVERY REPOSITORY HAS ONE IS THAT THE GIT INIT COMMAND CREATES IT BY DEFAULT AND MOST PEOPLE DON’T BOTHER TO CHANGE IT.

A branch and its commit history

Figure 11. A branch and its commit history

Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let’s say you want to create a new branch called testing. You do this with the git branch command:

$ git branch testing

This creates a new pointer to the same commit you’re currently on.

Two branches pointing into the same series of commits

Figure 12. Two branches pointing into the same series of commits

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.

HEAD pointing to a branch

Figure 13. HEAD pointing to a branch

You can easily see this by running a simple git log command that shows you where the branch pointers are pointing. This option is called --decorate.

$ git log --oneline --decorate

f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the central interface

34ac2 Fix bug #1328 - stack overflow under certain conditions

98ca9 Initial commit

You can see the master and testing branches that are right there next to the f30ab commit.

Switching Branches

To switch to an existing branch, you run the git checkout command. Let’s switch to the new testing branch:

$ git checkout testing

This moves HEAD to point to the testing branch.

HEAD points to the current branch

Figure 14. HEAD points to the current branch

What is the significance of that? Well, let’s do another commit:

$ vim test.rb

$ git commit -a -m 'made a change'

The HEAD branch moves forward when a commit is made

Figure 15. The HEAD branch moves forward when a commit is made

This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran git checkout to switch branches. Let’s switch back to the master branch:

Note

git log doesn’t show all the branches all the time

IF YOU WERE TO RUN GIT LOG RIGHT NOW, YOU MIGHT WONDER WHERE THE "TESTING" BRANCH YOU JUST CREATED WENT, AS IT WOULD NOT APPEAR IN THE OUTPUT.

THE BRANCH HASN’T DISAPPEARED; GIT JUST DOESN’T KNOW THAT YOU’RE INTERESTED IN THAT BRANCH AND IT IS TRYING TO SHOW YOU WHAT IT THINKS YOU’RE INTERESTED IN. IN OTHER WORDS, BY DEFAULT, GIT LOG WILL ONLY SHOW COMMIT HISTORY BELOW THE BRANCH YOU’VE CHECKED OUT.

TO SHOW COMMIT HISTORY FOR THE DESIRED BRANCH YOU HAVE TO EXPLICITLY SPECIFY IT: GIT LOG TESTING. TO SHOW ALL OF THE BRANCHES, ADD --ALL TO YOUR GIT LOG COMMAND.

$ git checkout master

HEAD moves when you checkout

Figure 16. HEAD moves when you checkout

That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you’ve done in your testing branch so you can go in a different direction.

Note

Switching branches changes files in your working directory

It’s important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch. If Git cannot do it cleanly, it will not let you switch at all.

Let’s make a few changes and commit again:

$ vim test.rb

$ git commit -a -m 'made other changes'

Now your project history has diverged (see Divergent history). You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you’re ready. And you did all that with simple branch, checkout, and commit commands.

Divergent history

FIGURE 17. DIVERGENT HISTORY

You can also see this easily with the git log command. If you run git log --oneline --decorate --graph --all it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) Made other changes

| \* 87ab2 (testing) Made a change

|/

\* f30ab Add feature #32 - ability to add new formats to the central interface

\* 34ac2 Fix bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

BECAUSE A BRANCH IN GIT IS ACTUALLY A SIMPLE FILE THAT CONTAINS THE 40 CHARACTER SHA-1 CHECKSUM OF THE COMMIT IT POINTS TO, BRANCHES ARE CHEAP TO CREATE and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

This is in sharp contrast to the way most older VCS tools branch, which involves copying all of the project’s files into a second directory. This can take several seconds or even minutes, depending on the size of the project, whereas in Git the process is always instantaneous. Also, because we’re recording the parents when we commit, finding a proper merge base for merging is automatically done for us and is generally very easy to do. These features help encourage developers to create and use branches often.

Let’s see why you should do so.

Note

Creating a new branch and switching to it at the same time (create branch)

IT’S TYPICAL TO CREATE A NEW BRANCH AND WANT TO SWITCH TO THAT NEW BRANCH AT THE SAME TIME — THIS CAN BE DONE IN ONE OPERATION WITH git checkout -b <newbranchname>.

Note

FROM GIT VERSION 2.23 ONWARDS YOU CAN USE GIT SWITCH INSTEAD OF GIT CHECKOUT TO:

SWITCH TO AN EXISTING BRANCH: GIT SWITCH TESTING-BRANCH.

CREATE A NEW BRANCH AND SWITCH TO IT: GIT SWITCH -C NEW-BRANCH. THE -C FLAG STANDS FOR CREATE, YOU CAN ALSO USE THE FULL FLAG: --CREATE.

RETURN TO YOUR PREVIOUSLY CHECKED OUT BRANCH: GIT SWITCH -.

Basic Branching and Merging

Let’s go through a simple example of branching and merging with a workflow that you might use in the real world. You’ll follow these steps:

-Do some work on a website.

-Create a branch for a new user story you’re working on.

-Do some work in that branch.

At this stage, you’ll receive a call that another issue is critical and you need a hotfix. You’ll do the following:

-Switch to your production branch.

-Create a branch to add the hotfix.

-After it’s tested, merge the hotfix branch, and push to production.

-Switch back to your original user story and continue working.

Basic Branching

First, let’s say you’re working on your project and have a couple of commits already on the master branch.

A simple commit history

Figure 18. A simple commit history

You’ve decided that you’re going to work on issue #53 in whatever issue-tracking system your company uses. To create a new branch and switch to it at the same time, you can run the git checkout command with the -b switch:

$ git checkout -b iss53

Switched to a new branch "iss53"

This is shorthand for:

$ git branch iss53

$ git checkout iss53

Creating a new branch pointer

Figure 19. Creating a new branch pointer

You work on your website and do some commits. Doing so moves the iss53 branch forward, because you have it checked out (that is, your HEAD is pointing to it):

$ vim index.html

$ git commit -a -m 'Create new footer [issue 53]'

The `iss53` branch has moved forward with your work

Figure 20. The iss53 branch has moved forward with your work

Now you get the call that there is an issue with the website, and you need to fix it immediately. With Git, you don’t have to deploy your fix along with the iss53 changes you’ve made, and you don’t have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your master branch.

IMPORTANT POINT:

A BRANCH CONNECTION SNAPSHOT IS CHANGED TO DIFFERENT SNAPSHOT ON 2 REASONS:

1:IF A COMMIT IS MADE

2:IF THE CURRENT BRANCH IS MERGED WITH DIFFERENT BRANCH

HOWEVER, BEFORE YOU DO THAT, NOTE THAT IF YOUR WORKING DIRECTORY OR STAGING AREA HAS UNCOMMITTED CHANGES THAT CONFLICT WITH THE BRANCH YOU’RE CHECKING OUT, GIT WON’T LET YOU SWITCH BRANCHES. IT’S BEST TO HAVE A CLEAN WORKING STATE WHEN YOU SWITCH BRANCHES. THERE ARE WAYS TO GET AROUND THIS (NAMELY, STASHING AND COMMIT AMENDING) THAT WE’LL COVER LATER ON, IN STASHING AND CLEANING. For now, let’s assume you’ve committed all your changes, so you can switch back to your master branch:

$ git checkout master

Switched to branch 'master'

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix. This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let’s create a hotfix branch on which to work until it’s completed:

$ git checkout -b hotfix

Switched to a new branch 'hotfix'

$ vim index.html

$ git commit -a -m 'Fix broken email address'

[hotfix 1fb7853] Fix broken email address

1 file changed, 2 insertions(+)

Hotfix branch based on `master`

Figure 21. Hotfix branch based on master

You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your master branch to deploy to production. You do this with the git merge command:

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

YOU’LL NOTICE THE PHRASE “FAST-FORWARD” IN THAT MERGE. BECAUSE THE COMMIT C4 POINTED TO BY THE BRANCH HOTFIX YOU MERGED IN WAS DIRECTLY AHEAD OF THE COMMIT C2 YOU’RE ON, GIT SIMPLY MOVES THE POINTER FORWARD. TO PHRASE THAT ANOTHER WAY, WHEN YOU TRY TO MERGE ONE COMMIT WITH A COMMIT THAT CAN BE REACHED BY FOLLOWING THE FIRST COMMIT’S HISTORY, GIT SIMPLIFIES THINGS BY MOVING THE POINTER FORWARD BECAUSE THERE IS NO DIVERGENT WORK TO MERGE TOGETHER — THIS IS CALLED A “FAST-FORWARD.”

Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.

`master` is fast-forwarded to `hotfix`

Figure 22. master is fast-forwarded to hotfix

AFTER YOUR SUPER-IMPORTANT FIX IS DEPLOYED, YOU’RE READY TO SWITCH BACK TO THE WORK YOU WERE DOING BEFORE YOU WERE INTERRUPTED. HOWEVER, FIRST YOU’LL DELETE THE HOTFIX BRANCH, BECAUSE YOU NO LONGER NEED IT — THE MASTER BRANCH POINTS AT THE SAME PLACE. YOU CAN DELETE IT WITH THE -D OPTION TO GIT BRANCH:

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

$ git checkout iss53

Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'Finish the new footer [issue 53]'

[iss53 ad82d7a] Finish the new footer [issue 53]

1 file changed, 1 insertion(+)

Work continues on `iss53`

Figure 23. Work continues on iss53

IT’S WORTH NOTING HERE THAT THE WORK YOU DID IN YOUR HOTFIX BRANCH IS NOT CONTAINED IN THE FILES IN YOUR ISS53 BRANCH. IF YOU NEED TO PULL IT IN, YOU CAN MERGE YOUR MASTER BRANCH INTO YOUR ISS53 BRANCH BY RUNNING GIT MERGE MASTER, OR YOU CAN WAIT TO INTEGRATE THOSE CHANGES UNTIL YOU DECIDE TO PULL THE ISS53 BRANCH BACK INTO MASTER LATER.

Basic Merging

Suppose you’ve decided that your issue #53 work is complete and ready to be merged into your master branch. In order to do that, you’ll merge your iss53 branch into master, much like you merged your hotfix branch earlier. All you have to do is check out the branch you wish to merge into and then run the git merge command:

$ git checkout master

Switched to branch 'master'

$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

THIS LOOKS A BIT DIFFERENT THAN THE HOTFIX MERGE YOU DID EARLIER. IN THIS CASE, YOUR DEVELOPMENT HISTORY HAS DIVERGED FROM SOME OLDER POINT. BECAUSE THE COMMIT ON THE BRANCH YOU’RE ON ISN’T A DIRECT ANCESTOR OF THE BRANCH YOU’RE MERGING IN, GIT HAS TO DO SOME WORK. IN THIS CASE, GIT DOES A SIMPLE THREE-WAY MERGE, USING THE TWO SNAPSHOTS POINTED TO BY THE BRANCH TIPS AND THE COMMON ANCESTOR OF THE TWO.

Three snapshots used in a typical merge

Figure 24. Three snapshots used in a typical merge

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a MERGE COMMIT, and is special in that it has more than one parent.

A merge commit

Figure 25. A merge commit

Now that your work is merged in, you have no further need for the iss53 branch. You can close the issue in your issue-tracking system, and delete the branch:

$ git branch -d iss53

Basic Merge Conflicts

OCCASIONALLY, THIS PROCESS DOESN’T GO SMOOTHLY. 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. IF YOUR FIX FOR ISSUE #53 MODIFIED THE SAME PART OF A FILE AS THE HOTFIX BRANCH, YOU’LL GET A MERGE CONFLICT THAT LOOKS SOMETHING LIKE THIS:

CUSTOM NOTE:

SUPPOSE THERE ARE 2 BRANCHES, BRANCH TITANIC AND BRANCH AVATAR,EITHER IF YOU CHECKOUT TO AVATAR AND MERGE TITANIC WITH IT,OR IF YOU CHECKOUT TO TITANIC AND MERGE AVATAR WITH IT, BOTH ARE SAME AS NOW BOTH THE BRANCHES WILL BE POINTING TO SAME SNAPSHOT,BUT IT IS BETTER TO CHECKOUT TO THE BRANCH THAT IS CONSIDERED IMPORTANT AND THEN MERGE THE INSIGNIFICANT BRANCH SO THAT MAIN BRANCH WILL HAVE THE FINAL PUSH.

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

Git hasn’t automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status:

$ git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

Anything that has merge conflicts and hasn’t been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =======), while the version in your iss53 branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

<div id="footer">

please contact us at email.support@github.com

</div>

This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been completely removed. After you’ve resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you want to use a graphical tool to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts:

$ git mergetool

This message is displayed because 'merge.tool' is not configured.

See 'git mergetool --tool-help' or 'git help config' for more details.

'git mergetool' will now attempt to use one of the following tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge

Merging:

index.html

Normal merge conflict for 'index.html':

{local}: modified file

{remote}: modified file

Hit return to start merge resolution tool (opendiff):

If you want to use a merge tool other than the default (Git chose opendiff in this case because the command was run on a Mac), you can see all the supported tools listed at the top after “one of the following tools.” Just type the name of the tool you’d rather use.

Note

If you need more advanced tools for resolving tricky merge conflicts, we cover more on merging in Advanced Merging.

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you. You can run git status again to verify that all conflicts have been resolved:

$ git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: index.html

If you’re happy with that, and you verify that everything that had conflicts has been staged, you can type git commit to finalize the merge commit. The commit message by default looks something like this:

Merge branch 'iss53'

Conflicts:

index.html

#

# It looks like you may be committing a merge.

# If this is not correct, please remove the file

# .git/MERGE\_HEAD

# and try again.

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# All conflicts fixed but you are still merging.

#

# Changes to be committed:

# modified: index.html

#

If you think it would be helpful to others looking at this merge in the future, you can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.

Branch Management

Now that you’ve created, merged, and deleted some branches, let’s look at some branch-management tools that will come in handy when you begin using branches all the time.

The git branch command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

$ git branch

iss53

\* master

testing

NOTICE THE \* CHARACTER THAT PREFIXES THE MASTER BRANCH: IT INDICATES THE BRANCH THAT YOU CURRENTLY HAVE CHECKED OUT (I.E., THE BRANCH THAT HEAD POINTS TO). THIS MEANS THAT IF YOU COMMIT AT THIS POINT, THE MASTER BRANCH WILL BE MOVED FORWARD WITH YOUR NEW WORK.

TO SEE THE LAST COMMIT ON EACH BRANCH, YOU CAN RUN GIT BRANCH -V:

$ git branch -v

iss53 93b412c Fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 Add scott to the author list in the readme

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:

$ git branch --merged

iss53

\* master

Because you already merged in iss53 earlier, you see it in your list. Branches on this list without the \* in front of them are generally fine to delete with git branch -d; you’ve already incorporated their work into another branch, so you’re not going to lose anything.

TO SEE ALL THE BRANCHES THAT CONTAIN WORK YOU HAVEN’T YET MERGED IN, YOU CAN RUN GIT BRANCH --NO-MERGED:

$ GIT BRANCH --NO-MERGED

TESTING

THIS SHOWS YOUR OTHER BRANCH. BECAUSE IT CONTAINS WORK THAT ISN’T MERGED IN YET, TRYING TO DELETE IT WITH GIT BRANCH -D WILL FAIL:

$ git branch -d testing

ERROR: THE BRANCH 'TESTING' IS NOT FULLY MERGED.

If you are sure you want to delete it, run 'git branch -D testing'.

If you really do want to delete the branch and lose that work, you can force it with -D, as the helpful message points out.

Tip

The options described above, --merged and --no-merged will, if not given a commit or branch name as an argument, show you what is, respectively, merged or not merged into your current branch.

You can always provide an additional argument to ask about the merge state with respect to some other branch without checking that other branch out first, as in, what is not merged into the master branch?

$ git checkout testing

$ git branch --no-merged master

topicA

featureB

Changing a branch name

Caution

DO NOT RENAME BRANCHES THAT ARE STILL IN USE BY OTHER COLLABORATORS. DO NOT RENAME A BRANCH LIKE MASTER/MAIN/MAINLINE WITHOUT HAVING READ THE SECTION "CHANGING THE MASTER BRANCH NAME".

SUPPOSE YOU HAVE A BRANCH THAT IS CALLED BAD-BRANCH-NAME AND YOU WANT TO CHANGE IT TO CORRECTED-BRANCH-NAME, WHILE KEEPING ALL HISTORY. YOU ALSO WANT TO CHANGE THE BRANCH NAME ON THE REMOTE (GITHUB, GITLAB, OTHER SERVER). HOW DO YOU DO THIS?

Rename the branch locally with the git branch --move command:

$ git branch --move bad-branch-name corrected-branch-name

This replaces your bad-branch-name with corrected-branch-name, but this change is only local for now. To let others see the corrected branch on the remote, push it:

$ git push --set-upstream origin corrected-branch-name

Now we’ll take a brief look at where we are now:

$ git branch --all

\* corrected-branch-name

main

remotes/origin/bad-branch-name

remotes/origin/corrected-branch-name

remotes/origin/main

Notice that you’re on the branch corrected-branch-name and it’s available on the remote. However, the branch with the bad name is also still present there but you can delete it by executing the following command:

$ git push origin --delete bad-branch-name

Now the bad branch name is fully replaced with the corrected branch name.

Changing the master branch name

Warning

CHANGING THE NAME OF A BRANCH LIKE MASTER/MAIN/MAINLINE/DEFAULT WILL BREAK THE INTEGRATIONS, SERVICES, HELPER UTILITIES AND BUILD/RELEASE SCRIPTS THAT YOUR REPOSITORY USES. BEFORE YOU DO THIS, MAKE SURE YOU CONSULT WITH YOUR COLLABORATORS. ALSO, MAKE SURE YOU DO A THOROUGH SEARCH THROUGH YOUR REPO AND UPDATE ANY REFERENCES TO THE OLD BRANCH NAME IN YOUR CODE AND SCRIPTS.

Rename your local master branch into main with the following command:

$ git branch --move master main

There’s no local master branch anymore, because it’s renamed to the main branch.

To let others see the new main branch, you need to push it to the remote. This makes the renamed branch available on the remote.

$ git push --set-upstream origin main

Now we end up with the following state:

git branch --all

\* main

remotes/origin/HEAD -> origin/master

remotes/origin/main

remotes/origin/master

Your local master branch is gone, as it’s replaced with the main branch. The main branch is present on the remote. However, the old master branch is still present on the remote. Other collaborators will continue to use the master branch as the base of their work, until you make some further changes.

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(best to follow the link-https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows)

Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we’ll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate them into your own development cycle.

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. It’s used to pull in topic branches (short-lived branches, like your earlier iss53 branch) when they’re ready, to make sure they pass all the tests and don’t introduce bugs.

In reality, we’re talking about pointers moving up the line of commits you’re making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.

A linear view of progressive-stability branching

Figure 26. A linear view of progressive-stability branching

It’s generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they’re fully tested.

A “SILO” VIEW OF PROGRESSIVE-STABILITY BRANCHING

Figure 27. A “silo” view of progressive-stability branching

YOU CAN KEEP DOING THIS FOR SEVERAL LEVELS OF STABILITY. SOME LARGER PROJECTS ALSO HAVE A PROPOSED OR PU (PROPOSED UPDATES) BRANCH THAT HAS INTEGRATED BRANCHES THAT MAY NOT BE READY TO GO INTO THE NEXT OR MASTER BRANCH. THE IDEA IS THAT YOUR BRANCHES ARE AT VARIOUS LEVELS OF STABILITY; WHEN THEY REACH A MORE STABLE LEVEL, THEY’RE MERGED INTO THE BRANCH ABOVE THEM. AGAIN, HAVING MULTIPLE LONG-RUNNING BRANCHES ISN’T NECESSARY, BUT IT’S OFTEN HELPFUL, ESPECIALLY WHEN YOU’RE DEALING WITH VERY LARGE OR COMPLEX PROJECTS.

Topic Branches

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.

Consider an example of doing some work (on master), branching off for an issue (iss91), working on it for a bit, branching off the second branch to try another way of handling the same thing (iss91v2), going back to your master branch and working there for a while, and then branching off there to do some work that you’re not sure is a good idea (dumbidea branch). Your commit history will look something like this:

Multiple topic branches

Figure 28. Multiple topic branches

Now, let’s say you decide you like the second solution to your issue best (iss91v2); and you showed the dumbidea branch to your coworkers, and it turns out to be genius. You can throw away the original iss91 branch (losing commits C5 and C6) and merge in the other two. Your history then looks like this:

History after merging `dumbidea` and `iss91v2`

Figure 29. History after merging dumbidea and iss91v2

We will go into more detail about the various possible workflows for your Git project in Distributed Git, so before you decide which branching scheme your next project will use, be sure to read that chapter.

It’s important to remember when you’re doing all this that these branches are completely local. When you’re branching and merging, everything is being done only in your Git repository — there is no communication with the server.

Remote Branches

Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. You can get a full list of remote references explicitly with git ls-remote <remote>, or git remote show <remote> for remote branches as well as more information. Nevertheless, a more common way is to take advantage of remote-tracking branches.

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.

This may be a bit confusing, so let’s look at an example. Let’s say you have a Git server on your network at git.ourcompany.com. If you clone from this, Git’s clone command automatically names it origin for you, pulls down all its data, creates a pointer to where its master branch is, and names it origin/master locally. Git also gives you your own local master branch starting at the same place as origin’s master branch, so you have something to work from.

Note

“origin” is not special

Just like the branch name “master” does not have any special meaning in Git, neither does “origin”. While “master” is the default name for a starting branch when you run git init which is the only reason it’s widely used, “origin” is the default name for a remote when you run git clone. If you run git clone -o booyah instead, then you will have booyah/master as your default remote branch.

Server and local repositories after cloning.

Figure 30. Server and local repositories after cloning

If you do some work on your local master branch, and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn’t move.

Local and remote work can diverge

Figure 31. Local and remote work can diverge

To synchronize your work with a given remote, you run a git fetch <remote> command (in our case, git fetch origin). This command looks up which server “origin” is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.

`git fetch` updates your remote references

Figure 32. git fetch updates your remote-tracking branches

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let’s assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at git.team1.ourcompany.com. You can add it as a new remote reference to the project you’re currently working on by running the git remote add command as we covered in Git Basics. Name this remote teamone, which will be your shortname for that whole URL.

Adding another server as a remote

Figure 33. Adding another server as a remote

Now, you can run git fetch teamone to fetch everything the remote teamone server has that you don’t have yet. Because that server has a subset of the data your origin server has right now, Git fetches no data but sets a remote-tracking branch called teamone/master to point to the commit that teamone has as its master branch.

Remote tracking branch for `teamone/master`

Figure 34. Remote-tracking branch for teamone/master

-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>:

$ git push origin serverfix

Counting objects: 24, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.

Total 24 (delta 2), reused 0 (delta 0)

To https://github.com/schacon/simplegit

\* [new branch] serverfix -> serverfix

THIS IS A BIT OF A SHORTCUT. GIT AUTOMATICALLY EXPANDS THE SERVERFIX BRANCHNAME OUT TO REFS/HEADS/SERVERFIX:REFS/HEADS/SERVERFIX, WHICH MEANS, “TAKE MY SERVERFIX LOCAL BRANCH AND PUSH IT TO UPDATE THE REMOTE’S SERVERFIX BRANCH.” WE’LL GO OVER THE REFS/HEADS/ PART IN DETAIL IN GIT INTERNALS, BUT YOU CAN GENERALLY LEAVE IT OFF. YOU CAN ALSO DO GIT PUSH ORIGIN SERVERFIX:SERVERFIX, WHICH DOES THE SAME THING — IT SAYS, “TAKE MY SERVERFIX AND MAKE IT THE REMOTE’S SERVERFIX.” YOU CAN USE THIS FORMAT TO PUSH A LOCAL BRANCH INTO A REMOTE BRANCH THAT IS NAMED DIFFERENTLY. IF YOU DIDN’T WANT IT TO BE CALLED SERVERFIX ON THE REMOTE, YOU COULD INSTEAD RUN GIT PUSH ORIGIN SERVERFIX:AWESOMEBRANCH TO PUSH YOUR LOCAL SERVERFIX BRANCH TO THE AWESOMEBRANCH BRANCH ON THE REMOTE PROJECT.

Note

Don’t type your password every time

If you’re using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication. By default it will prompt you on the terminal for this information so the server can tell if you’re allowed to push.

If you don’t want to type it every single time you push, you can set up a “credential cache”. The simplest is just to keep it in memory for a few minutes, which you can easily set up by running git config --global credential.helper cache.

For more information on the various credential caching options available, see Credential Storage.

The next time one of your collaborators fetches from the server, they will get a reference to where the server’s version of serverfix is under the remote branch origin/serverfix:

$ git fetch origin

remote: Counting objects: 7, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From https://github.com/schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

It’s important to note that when you do a fetch that brings down new remote-tracking branches, you don’t automatically have local, editable copies of them. In other words, in this case, you don’t have a new serverfix branch — you have only an origin/serverfix pointer that you can’t modify.

To merge this work into your current working branch, you can run git merge origin/serverfix. If you want your own serverfix branch that you can work on, you can base it off your remote-tracking branch:

$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

This gives you a local branch that you can work on that starts where origin/serverfix is.

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.

When you clone a repository, it generally automatically creates a master branch that tracks origin/master. However, you can set up other tracking branches if you wish — ones that track branches on other remotes, or don’t track the master branch. The simple case is the example you just saw, running git checkout -b <branch> <remote>/<branch>. This is a common enough operation that Git provides the --track shorthand:

$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

IN FACT, THIS IS SO COMMON THAT THERE’S EVEN A SHORTCUT FOR THAT SHORTCUT. IF THE BRANCH NAME YOU’RE TRYING TO CHECKOUT (A) DOESN’T EXIST AND (B) EXACTLY MATCHES A NAME ON ONLY ONE REMOTE, GIT WILL CREATE A TRACKING BRANCH FOR YOU:

$ git checkout serverfix

BRANCH SERVERFIX SET UP TO TRACK REMOTE BRANCH SERVERFIX FROM ORIGIN.

Switched to a new branch 'serverfix'

TO SET UP A LOCAL BRANCH WITH A DIFFERENT NAME THAN THE REMOTE BRANCH, YOU CAN EASILY USE THE FIRST VERSION WITH A DIFFERENT LOCAL BRANCH NAME:

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch serverfix from origin.

Switched to a new branch 'sf'

NOW, YOUR LOCAL BRANCH SF WILL AUTOMATICALLY PULL FROM ORIGIN/SERVERFIX.

IF YOU ALREADY HAVE A LOCAL BRANCH AND WANT TO SET IT TO A REMOTE BRANCH YOU JUST PULLED DOWN, OR WANT TO CHANGE THE UPSTREAM BRANCH YOU’RE TRACKING, YOU CAN USE THE -U OR --SET-UPSTREAM-TO OPTION TO GIT BRANCH TO EXPLICITLY SET IT AT ANY TIME.

$ git branch -u origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Upstream shorthand

Note

WHEN YOU HAVE A TRACKING BRANCH SET UP, YOU CAN REFERENCE ITS UPSTREAM BRANCH WITH THE @{UPSTREAM} OR @{U} SHORTHAND. SO IF YOU’RE ON THE MASTER BRANCH AND IT’S TRACKING ORIGIN/MASTER, YOU CAN SAY SOMETHING LIKE GIT MERGE @{U} INSTEAD OF GIT MERGE ORIGIN/MASTER IF YOU WISH.

IF YOU WANT TO SEE WHAT TRACKING BRANCHES YOU HAVE SET UP, YOU CAN USE THE -VV OPTION TO GIT BRANCH. THIS WILL LIST OUT YOUR LOCAL BRANCHES WITH MORE INFORMATION INCLUDING WHAT EACH BRANCH IS TRACKING AND IF YOUR LOCAL BRANCH IS AHEAD, BEHIND OR BOTH.

$ git branch -vv

iss53 7e424c3 [origin/iss53: ahead 2] Add forgotten brackets

master 1ae2a45 [origin/master] Deploy index fix

\* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it

testing 5ea463a Try something new

SO HERE WE CAN SEE THAT OUR ISS53 BRANCH IS TRACKING ORIGIN/ISS53 AND IS “AHEAD” BY TWO, MEANING THAT WE HAVE TWO COMMITS LOCALLY THAT ARE NOT PUSHED TO THE SERVER. WE CAN ALSO SEE THAT OUR MASTER BRANCH IS TRACKING ORIGIN/MASTER AND IS UP TO DATE. NEXT WE CAN SEE THAT OUR SERVERFIX BRANCH IS TRACKING THE SERVER-FIX-GOOD BRANCH ON OUR TEAMONE SERVER AND IS AHEAD BY THREE AND BEHIND BY ONE, MEANING THAT THERE IS ONE COMMIT ON THE SERVER WE HAVEN’T MERGED IN YET AND THREE COMMITS LOCALLY THAT WE HAVEN’T PUSHED. FINALLY WE CAN SEE THAT OUR TESTING BRANCH IS NOT TRACKING ANY REMOTE BRANCH.

IT’S IMPORTANT TO NOTE THAT THESE NUMBERS ARE ONLY SINCE THE LAST TIME YOU FETCHED FROM EACH SERVER. THIS COMMAND DOES NOT REACH OUT TO THE SERVERS, IT’S TELLING YOU ABOUT WHAT IT HAS CACHED FROM THESE SERVERS LOCALLY. IF YOU WANT TOTALLY UP TO DATE AHEAD AND BEHIND NUMBERS, YOU’LL NEED TO FETCH FROM ALL YOUR REMOTES RIGHT BEFORE RUNNING THIS. YOU COULD DO THAT LIKE THIS:

$ git fetch --all; git branch -vv

Pulling

WHILE THE GIT FETCH COMMAND WILL FETCH ALL THE CHANGES ON THE SERVER THAT YOU DON’T HAVE YET, IT WILL NOT MODIFY YOUR WORKING DIRECTORY AT ALL. IT WILL SIMPLY GET THE DATA FOR YOU AND LET YOU MERGE IT YOURSELF. HOWEVER, THERE IS A COMMAND CALLED GIT PULL WHICH IS ESSENTIALLY A GIT FETCH IMMEDIATELY FOLLOWED BY A GIT MERGE IN MOST CASES. IF YOU HAVE A TRACKING BRANCH SET UP AS DEMONSTRATED IN THE LAST SECTION, EITHER BY EXPLICITLY SETTING IT OR BY HAVING IT CREATED FOR YOU BY THE CLONE OR CHECKOUT COMMANDS, GIT PULL WILL LOOK UP WHAT SERVER AND BRANCH YOUR CURRENT BRANCH IS TRACKING, FETCH FROM THAT SERVER AND THEN TRY TO MERGE IN THAT REMOTE BRANCH.

GENERALLY IT’S BETTER TO SIMPLY USE THE FETCH AND MERGE COMMANDS EXPLICITLY AS THE MAGIC OF GIT PULL CAN OFTEN BE CONFUSING.

Deleting Remote Branches

SUPPOSE YOU’RE DONE WITH A REMOTE BRANCH — SAY YOU AND YOUR COLLABORATORS ARE FINISHED WITH A FEATURE AND HAVE MERGED IT INTO YOUR REMOTE’S MASTER BRANCH (OR WHATEVER BRANCH YOUR STABLE CODELINE IS IN). YOU CAN DELETE A REMOTE BRANCH USING THE --DELETE OPTION TO GIT PUSH. IF YOU WANT TO DELETE YOUR SERVERFIX BRANCH FROM THE SERVER, YOU RUN THE FOLLOWING:

$ git push origin --delete serverfix

To https://github.com/schacon/simplegit

- [deleted] serverfix

BASICALLY ALL THIS DOES IS REMOVE THE POINTER FROM THE SERVER. THE GIT SERVER WILL GENERALLY KEEP THE DATA THERE FOR A WHILE UNTIL A GARBAGE COLLECTION RUNS, SO IF IT WAS ACCIDENTALLY DELETED, IT’S OFTEN EASY TO RECOVER.

Rebasing:(https://git-scm.com/book/en/v2/Git-Branching-Rebasing better to check images in the link and read)

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.

-The Basic Rebase

If you go back to an earlier example from Basic Merging, you can see that you diverged your work and made commits on two different branches.

-Simple divergent history

Figure 35. Simple divergent history

The easiest way to integrate the branches, as we’ve already covered, is the merge command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).

Merging to integrate diverged work history

Figure 36. Merging to integrate diverged work history

HOWEVER, THERE IS ANOTHER WAY: 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.

For this example, you would check out the experiment branch, and then rebase it onto the master branch as follows:

$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

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.

Rebasing the change introduced in `C4` onto `C3`

Figure 37. Rebasing the change introduced in C4 onto C3

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

$ git checkout master

$ git merge experiment

Fast-forwarding the `master` branch

Figure 38. Fast-forwarding the master branch

Now, the snapshot pointed to by C4' is exactly the same as the one that was pointed to by C5 in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

Often, you’ll do this to make sure your commits apply cleanly on a remote branch — perhaps in a project to which you’re trying to contribute but that you don’t maintain. In this case, you’d do your work in a branch and then rebase your work onto origin/master when you were ready to submit your patches to the main project. That way, the maintainer doesn’t have to do any integration work — just a fast-forward or a clean apply.

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 can also have your rebase replay on something other than the rebase target branch. Take a history like A history with a topic branch off another topic branch, for example. 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.

A history with a topic branch off another topic branch

Figure 39. A history with a topic branch off another topic branch

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

This basically says, “Take the client branch, figure out the patches since it diverged from the server branch, and replay these patches in the client branch as if it was based directly off the master branch instead.” It’s a bit complex, but the result is pretty cool.

Rebasing a topic branch off another topic branch

Figure 40. Rebasing a topic branch off another topic branch

Now you can fast-forward your master branch (see Fast-forwarding your master branch to include the client branch changes):

$ git checkout master

$ git merge client

Fast-forwarding your `master` branch to include the client branch changes

Figure 41. Fast-forwarding your master branch to include the client branch changes

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

This replays your server work on top of your master work, as shown in Rebasing your server branch on top of your master branch.

Rebasing your server branch on top of your `master` branch

Figure 42. Rebasing your server branch on top of your master branch

Then, you can fast-forward the base branch (master):

$ git checkout master

$ git merge server

You can remove the client and server branches because all the work is integrated and you don’t need them anymore, leaving your history for this entire process looking like Final commit history:

$ git branch -d client

$ git branch -d server

Final commit history

Figure 43. Final commit history

The Perils of Rebasing

Ahh, but the bliss of rebasing isn’t without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository and that people may have based work on.

If you follow that guideline, you’ll be fine. If you don’t, people will hate you, and you’ll be scorned by friends and family.

When you rebase stuff, you’re abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with git rebase and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let’s look at an example of how rebasing work that you’ve made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:

Clone a repository, and base some work on it.

Figure 44. Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch it and merge the new remote branch into your work, making your history look something like this:

Fetch more commits, and merge them into your work.

Figure 45. Fetch more commits, and merge them into your work

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a git push --force to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

Someone pushes rebased commits, abandoning commits you’ve based your work on.

Figure 46. Someone pushes rebased commits, abandoning commits you’ve based your work on

Now you’re both in a pickle. If you do a git pull, you’ll create a merge commit which includes both lines of history, and your repository will look like this:

You merge in the same work again into a new merge commit

Figure 47. You merge in the same work again into a new merge commit

If you run a git log when your history looks like this, you’ll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to the server, you’ll reintroduce all those rebased commits to the central server, which can further confuse people. It’s pretty safe to assume that the other developer doesn’t want C4 and C6 to be in the history; that’s why they rebased in the first place.

Rebase When You Rebase

If you do find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you’ve based work on, your challenge is to figure out what is yours and what they’ve rewritten.

It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a “patch-id”.

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we’re at Someone pushes rebased commits, abandoning commits you’ve based your work on we run git rebase teamone/master, Git will:

Determine what work is unique to our branch (C2, C3, C4, C6, C7)

Determine which are not merge commits (C2, C3, C4)

Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')

Apply those commits to the top of teamone/master

So instead of the result we see in You merge in the same work again into a new merge commit, we would end up with something more like Rebase on top of force-pushed rebase work.

Rebase on top of force-pushed rebase work

Figure 48. Rebase on top of force-pushed rebase work

This only works if C4 and C4' that your partner made are almost exactly the same patch. Otherwise the rebase won’t be able to tell that it’s a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running a git pull --rebase instead of a normal git pull. Or you could do it manually with a git fetch followed by a git rebase teamone/master in this case.

If you are using git pull and want to make --rebase the default, you can set the pull.rebase config value with something like git config --global pull.rebase true.

If you only ever rebase commits that have never left your own computer, you’ll be just fine. If you rebase commits that have been pushed, but that no one else has based commits from, you’ll also be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run git pull --rebase to try to make the pain after it happens a little bit simpler.

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.

my notes on git:

-git is the most popular version control system(VCS).

-a version control system helps us to track history of a data such as code and let more people work together easily

-a version control system is of 2 types:

1: centralized:

a VCS in which all members connect to a single server to get latest copy of the code, or to share the changes with others

ex:subversion,microsoft team foundation,git

for subversion,microsoft team foundation-server should be online to collaborate

2:distributed:

every member has a copy of the project with it's history on their machine(available offline), if the server is offline, all the members of this distributed VCS can share data with each other.

ex: git, mercurial

-why git?

open source

super fast

free

scalable

cheap branching/merging

-ways of using git

command line(fastest and easiest sometimes)

code editors and IDE's(ex:VScode)

-repositories-project itself or ,folder/place where the project is kept

-github- a website where you host your repositories online

-clone- bring a copy of your repository that is hosted somewhere else in to your local

-git add . (this will add all the untracked and modified files and let them staged so that git will know about them and when committed they are included in the generated snapshot)

-git commit -m '<enter commit message>' -m '<extra description for the current commit>'

-to push using token

\*need to find this process online and write it here

-you can login to your githuba ccount and create a repo and add some files and save them under the repository that you have created, to get the code from github to your local, you can use the git clone command and copy all the code from github to local machine using git.after getting that code in to local, you might change the code based on requirement and then yo might want to push the code to github.to do that, you might want to prove to github that you are the owner of the github account. in order to connect your local machine eith github can be through ssh keys and the process is as below

-before you push code to git from your local, you need to generate public and private keys and the process is as follows(if you have already generated keys refer the link - https://docs.github.com/en/authentication/connecting-to-github-with-ssh/about-ssh)

SYNTAX TO GENERATE KEY:ssh-keygen -t rsa -b 4096 -C <email that is linked to github>

saivinil\_pratap@TIGER02143:~$ ssh-keygen -t rsa -b 4096 -C saivinil.pratap@tigeranalytics.com

Generating public/private rsa key pair.

Enter file in which to save the key (/home/saivinil\_pratap/.ssh/id\_rsa)(THIS IS THE PLACE WHERE THE DEFAULT SSH KEY IS STORED-/home/saivinil\_pratap/.ssh/id\_rsa): first\_key(HERE YOU SHOULD GIVE THE NAME FOR THE KEY)

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in first\_key

Your public key has been saved in first\_key.pub

The key fingerprint is:

SHA256:hI3GXMnK401r7YrfxAmKusyPDYdSxqtrcZVMt3SiSLQ saivinil.pratap@tigeranalytics.com

The key's randomart image is:

+---[RSA 4096]----+

| .. ... |

| ..+ Oo. |

| .E+.@.\* |

| . . \*+o. |

| + .. +So |

| + + ...+o.. |

|. \* o .. .+ |

| \* \* . o. |

|+.\*oo ..o.. |

+----[SHA256]-----+

saivinil\_pratap@TIGER02143:~$ ls |grep first\_key (THIS COMMAND WILL SEARCH FOR THE KEYS THAT ARE GENERATED)

first\_key (THIS IS THE PRIVATE KEY THAT IS USED TO CONNECT TO GIHUB AND PUSH YOUR CODE TO GITHUB OR USE YOUR ACCOUNT VIA LOCAL MACHINE, YOU CAN USE PRIVATE KEY TO SHOW GITHUB THAT GENEREATED THIS PRIVATE KEY, PUBLIC KEY IS GENERATED FROM PRIVATE KEY, AS IT IS GENERATED FROM PRIVATE KEY, IT IS A MATHAMETICAL PROOF THAT IT IS YOU WHO CREATED THE PROJECT, SO IT IS IMPORTANT FOR YOU TO )

first\_key.pub (THIS IS THE KEY THAT YOU ARE GOING TO UPLOAD IN TO THE GITHUB INTERFACE, .pub STANDS FOR PUBLIC, IT MEANS THE KEY IS PUBLIC )

saivinil\_pratap@TIGER02143:~$ cd .ssh

saivinil\_pratap@TIGER02143:~/.ssh$ ls

known\_hosts known\_hosts.old

saivinil\_pratap@TIGER02143:~/.ssh$ cd ../

saivinil\_pratap@TIGER02143:~$ cat first\_key.pub

ssh-rsa  saivinil.pratap@tigeranalytics.com(YOU CAN SEE YOUR EMAIL THAT YOU USE TO CONNECT TO GITHUB)

(NOW GO TO YOUR GITHUB ACCOUNT AND GO TO SETTINGS AND THEN CLICK ON "SSH AND CPG KEYS" CLICK ON "NEW SSH KEY" ADD DESCRIPTION AND PASTE THE KEY AND REMOVE THE EMPTY LINES (IF ANY))

-now if did not give a name to the key it will be stored in ~/.ssh.is\_rsa:

saivinil\_pratap@TIGER02143:~/demorepo/lasttry$ cd /home/saivinil\_pratap/.ssh/id\_rsa(after executing ls you can see the id\_rsa)

saivinil\_pratap@TIGER02143:~/.ssh$ eval "$(ssh-agent -s)" (START THE SSH-AGENT IN THE BACKGROUND.)

Agent pid 640

saivinil\_pratap@TIGER02143:~/.ssh$ ssh-add ~/.ssh/id\_rsa (Add your SSH private key to the ssh-agent. If you created your key with a different name, or if you are adding an existing key that has a different name, replace id\_ed25519 in the command with the name of your private key file.)

Identity added: /home/saivinil\_pratap/.ssh/id\_rsa (saivinil.pratap@tigeranalytics.com)

saivinil\_pratap@TIGER02143:~/.ssh$ ssh -T git@github.com (TESTING YOUR SSH CONNECTION)

Hi saivinilpratap-ta! You've successfully authenticated, but GitHub does not provide shell access.

saivinil\_pratap@TIGER02143:~/.ssh$

-STASHING AND CLEANING-link:https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning

you are working on something important and it is in midway(changes which can not be commited yet) and you got a critical bug and you need to start working on it on priority basis.then you can save the current work using stash command and then switch

-stashes saved on one branch are available on other branches as well

let’s assume the current git status is as follows:

$ git status

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: index.html

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: lib/simplegit.rb

Now you want to switch branches, but you don’t want to commit what you’ve been working on yet, so you’ll stash the changes. To push a new stash onto your stack, run git stash or git stash push:

$ git stash

Saved working directory and index state \

"WIP on master: 049d078 Create index file"

HEAD is now at 049d078 Create index file

(To restore them type "git stash apply")

You can now see that your working directory is clean:

$ git status

# On branch master

nothing to commit, working directory 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:

$ git stash list

stash@{0}: WIP on master: 049d078 Create index file

stash@{1}: WIP on master: c264051 Revert "Add file\_size"

stash@{2}: WIP on master: 21d80a5 Add number to log

In this case, two stashes were saved previously, so you have access to three different stashed works. 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:

$ git stash apply

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: index.html

modified: lib/simplegit.rb

no changes added to commit (use "git add" and/or "git commit -a")

You can see that Git re-modifies the files you reverted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from. HAVING A CLEAN WORKING DIRECTORY AND APPLYING IT ON THE SAME BRANCH AREN’T NECESSARY TO SUCCESSFULLY APPLY A STASH. YOU CAN SAVE A STASH ON ONE BRANCH, SWITCH TO ANOTHER BRANCH LATER, AND TRY TO REAPPLY THE CHANGES. YOU CAN ALSO HAVE MODIFIED AND UNCOMMITTED FILES IN YOUR WORKING DIRECTORY WHEN YOU APPLY A STASH — GIT GIVES YOU MERGE CONFLICTS IF ANYTHING NO LONGER APPLIES CLEANLY.

THE CHANGES TO YOUR FILES WERE REAPPLIED, BUT THE FILE YOU STAGED BEFORE WASN’T RESTAGED. TO DO THAT, YOU MUST RUN THE GIT STASH APPLY COMMAND WITH A --INDEX OPTION TO TELL THE COMMAND TO TRY TO REAPPLY THE STAGED CHANGES. IF YOU HAD RUN THAT INSTEAD, YOU’D HAVE GOTTEN BACK TO YOUR ORIGINAL POSITION:

$ git stash apply --index

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: index.html

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: lib/simplegit.rb

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:

$ git stash list

stash@{0}: WIP on master: 049d078 Create index file

stash@{1}: WIP on master: c264051 Revert "Add file\_size"

stash@{2}: WIP on master: 21d80a5 Add number to log

$ git stash drop stash@{0}

Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)

You can also run git stash pop to apply the stash and then immediately drop it from your stack.

Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the --keep-index option to the git stash command. This tells Git to not only include all staged content in the stash being created, but simultaneously leave it in the index.

$ git status -s

M index.html

M lib/simplegit.rb

$ git stash --keep-index

Saved working directory and index state WIP on master: 1b65b17 added the index file

HEAD is now at 1b65b17 added the index file

$ git status -s

M index.html

ANOTHER COMMON THING YOU MAY WANT TO DO WITH STASH IS TO STASH THE UNTRACKED FILES AS WELL AS THE TRACKED ONES. BY DEFAULT, GIT STASH WILL STASH ONLY MODIFIED AND STAGED TRACKED FILES. IF YOU SPECIFY --INCLUDE-UNTRACKED OR -U, GIT WILL INCLUDE UNTRACKED FILES IN THE STASH BEING CREATED. HOWEVER, INCLUDING UNTRACKED FILES IN THE STASH WILL STILL NOT INCLUDE EXPLICITLY IGNORED FILES; TO ADDITIONALLY INCLUDE IGNORED FILES, USE --ALL (OR JUST -A).

$ git status -s

M index.html

M lib/simplegit.rb

?? new-file.txt

$ git stash -u

Saved working directory and index state WIP on master: 1b65b17 added the index file

HEAD is now at 1b65b17 added the index file

$ git status -s

$

Finally, if you specify the --patch flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

$ git stash --patch

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index 66d332e..8bb5674 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -16,6 +16,10 @@ class SimpleGit

return `#{git\_cmd} 2>&1`.chomp

end

end

+

+ def show(treeish = 'master')

+ command("git show #{treeish}")

+ end

end

test

Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file

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 stash branch testchanges

M index.html

M lib/simplegit.rb

Switched to a new branch 'testchanges'

On branch testchanges

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: index.html

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

Rebasing (more interactive with images in it-https://git-scm.com/book/en/v2/Git-Branching-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.

The Basic Rebase

IF YOU GO BACK TO AN EARLIER EXAMPLE FROM BASIC MERGING, YOU CAN SEE THAT YOU DIVERGED YOUR WORK AND MADE COMMITS ON TWO DIFFERENT BRANCHES.

SIMPLE DIVERGENT HISTORY

FIGURE 35. SIMPLE DIVERGENT HISTORY

THE EASIEST WAY TO INTEGRATE THE BRANCHES, AS WE’VE ALREADY COVERED, IS THE MERGE COMMAND. IT PERFORMS A THREE-WAY MERGE BETWEEN THE TWO LATEST BRANCH SNAPSHOTS (C3 AND C4) AND THE MOST RECENT COMMON ANCESTOR OF THE TWO (C2), CREATING A NEW SNAPSHOT (AND COMMIT).

Merging to integrate diverged work history

Figure 36. Merging to integrate diverged work history

However, there is another way: 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.

For this example, you would check out the experiment branch, and then rebase it onto the master branch as follows:

$ git checkout experiment

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

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.

Rebasing the change introduced in `C4` onto `C3`

Figure 37. Rebasing the change introduced in C4 onto C3

AT THIS POINT, YOU CAN GO BACK TO THE MASTER BRANCH AND DO A FAST-FORWARD MERGE.

$ git checkout master

$ git merge experiment

Fast-forwarding the `master` branch

Figure 38. Fast-forwarding the master branch

NOW, THE SNAPSHOT POINTED TO BY C4' IS EXACTLY THE SAME AS THE ONE THAT WAS POINTED TO BY C5 IN THE MERGE EXAMPLE. THERE IS NO DIFFERENCE IN THE END PRODUCT OF THE INTEGRATION, BUT REBASING MAKES FOR A CLEANER HISTORY. IF YOU EXAMINE THE LOG OF A REBASED BRANCH, IT LOOKS LIKE A LINEAR HISTORY: IT APPEARS THAT ALL THE WORK HAPPENED IN SERIES, EVEN WHEN IT ORIGINALLY HAPPENED IN PARALLEL.

OFTEN, YOU’LL DO THIS TO MAKE SURE YOUR COMMITS APPLY CLEANLY ON A REMOTE BRANCH — perhaps in a project to which you’re trying to contribute but that you don’t maintain. In this case, you’d do your work in a branch and then rebase your work onto origin/master when you were ready to submit your patches to the main project. That way, the maintainer doesn’t have to do any integration work — just a fast-forward or a clean apply.

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 can also have your rebase replay on something other than the rebase target branch. Take a history like A history with a topic branch off another topic branch, for example. 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.

A history with a topic branch off another topic branch

Figure 39. A history with a topic branch off another topic branch

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

This basically says, “TAKE THE CLIENT BRANCH, FIGURE OUT THE PATCHES SINCE IT DIVERGED FROM THE SERVER BRANCH, AND REPLAY THESE PATCHES IN THE CLIENT BRANCH AS IF IT WAS BASED DIRECTLY OFF THE MASTER BRANCH INSTEAD.” IT’S A BIT COMPLEX, BUT THE RESULT IS PRETTY COOL.

Rebasing a topic branch off another topic branch

Figure 40. Rebasing a topic branch off another topic branch

Now you can fast-forward your master branch (see Fast-forwarding your master branch to include the client branch changes):

$ git checkout master

$ git merge client

Fast-forwarding your `master` branch to include the client branch changes

Figure 41. Fast-forwarding your master branch to include the client branch changes

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

This replays your server work on top of your master work, as shown in Rebasing your server branch on top of your master branch.

Rebasing your server branch on top of your `master` branch

Figure 42. Rebasing your server branch on top of your master branch

Then, you can fast-forward the base branch (master):

$ git checkout master

$ git merge server

You can remove the client and server branches because all the work is integrated and you don’t need them anymore, leaving your history for this entire process looking like Final commit history:

$ git branch -d client

$ git branch -d server

Final commit history

Figure 43. Final commit history

The Perils of Rebasing

Ahh, but the bliss of rebasing isn’t without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository and that people may have based work on.

If you follow that guideline, you’ll be fine. If you don’t, people will hate you, and you’ll be scorned by friends and family.

When you rebase stuff, you’re abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with git rebase and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let’s look at an example of how rebasing work that you’ve made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:

Clone a repository, and base some work on it.

Figure 44. Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch it and merge the new remote branch into your work, making your history look something like this:

Fetch more commits, and merge them into your work.

Figure 45. Fetch more commits, and merge them into your work

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a git push --force to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

Someone pushes rebased commits, abandoning commits you’ve based your work on.

Figure 46. Someone pushes rebased commits, abandoning commits you’ve based your work on

Now you’re both in a pickle. If you do a git pull, you’ll create a merge commit which includes both lines of history, and your repository will look like this:

You merge in the same work again into a new merge commit

Figure 47. You merge in the same work again into a new merge commit

If you run a git log when your history looks like this, you’ll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to the server, you’ll reintroduce all those rebased commits to the central server, which can further confuse people. It’s pretty safe to assume that the other developer doesn’t want C4 and C6 to be in the history; that’s why they rebased in the first place.

Rebase When You Rebase

If you do find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you’ve based work on, your challenge is to figure out what is yours and what they’ve rewritten.

It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a “patch-id”.

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we’re at Someone pushes rebased commits, abandoning commits you’ve based your work on we run git rebase teamone/master, Git will:

Determine what work is unique to our branch (C2, C3, C4, C6, C7)

Determine which are not merge commits (C2, C3, C4)

Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')

Apply those commits to the top of teamone/master

So instead of the result we see in You merge in the same work again into a new merge commit, we would end up with something more like Rebase on top of force-pushed rebase work.

Rebase on top of force-pushed rebase work

Figure 48. Rebase on top of force-pushed rebase work

This only works if C4 and C4' that your partner made are almost exactly the same patch. Otherwise the rebase won’t be able to tell that it’s a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running a git pull --rebase instead of a normal git pull. Or you could do it manually with a git fetch followed by a git rebase teamone/master in this case.

If you are using git pull and want to make --rebase the default, you can set the pull.rebase config value with something like git config --global pull.rebase true.

If you only ever rebase commits that have never left your own computer, you’ll be just fine. If you rebase commits that have been pushed, but that no one else has based commits from, you’ll also be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run git pull --rebase to try to make the pain after it happens a little bit simpler.

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.

6.1 GitHub - Account Setup and Configuration

GitHub is the single largest host for Git repositories, and is the central point of collaboration for millions of developers and projects. A large percentage of all Git repositories are hosted on GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things. So while it’s not a direct part of the Git open source project, there’s a good chance that you’ll want or need to interact with GitHub at some point while using Git professionally.

This chapter is about using GitHub effectively. We’ll cover signing up for and managing an account, creating and using Git repositories, common workflows to contribute to projects and to accept contributions to yours, GitHub’s programmatic interface and lots of little tips to make your life easier in general.

If you are not interested in using GitHub to host your own projects or to collaborate with other projects that are hosted on GitHub, you can safely skip to Git Tools.

Account Setup and Configuration

The first thing you need to do is set up a free user account. Simply visit https://github.com, choose a user name that isn’t already taken, provide an email address and a password, and click the big green “Sign up for GitHub” button.

The GitHub sign-up form

Figure 81. The GitHub sign-up form

The next thing you’ll see is the pricing page for upgraded plans, but it’s safe to ignore this for now. GitHub will send you an email to verify the address you provided. Go ahead and do this; it’s pretty important (as we’ll see later).

Note

GitHub provides almost all of its functionality with free accounts, except some advanced features.

GitHub’s paid plans include advanced tools and features as well as increased limits for free services, but we won’t be covering those in this book. To get more information about available plans and their comparison, visit https://github.com/pricing.

CLICKING THE OCTOCAT LOGO AT THE TOP-LEFT OF THE SCREEN WILL TAKE YOU TO YOUR DASHBOARD PAGE. YOU’RE NOW READY TO USE GITHUB.

SSH Access

As of right now, you’re fully able to connect with Git repositories using the https:// protocol, authenticating with the username and password you just set up. However, to simply clone public projects, you don’t even need to sign up - the account we just created comes into play when we fork projects and push to our forks a bit later.

IF YOU’D LIKE TO USE SSH(a network protocol that connects users to a remote computer over a secure connection) REMOTES, YOU’LL NEED TO CONFIGURE A PUBLIC KEY. IF YOU DON’T ALREADY HAVE ONE, SEE GENERATING YOUR SSH PUBLIC KEY. Open up your account settings using the link at the top-right of the window:

The “Account settings” link

Figure 82. The “Account settings” link

Then select the “SSH keys” section along the left-hand side.

The “SSH keys” link

Figure 83. The “SSH keys” link.

From there, click the “Add an SSH key” button, give your key a name, paste the contents of your ~/.ssh/id\_rsa.pub (or whatever you named it) public-key file into the text area, and click “Add key”.

Note

Be sure to name your SSH key something you can remember. You can name each of your keys (e.g. "My Laptop" or "Work Account") so that if you need to revoke a key later, you can easily tell which one you’re looking for.

Your Avatar

Next, if you wish, you can replace the avatar that is generated for you with an image of your choosing. First go to the “Profile” tab (above the SSH Keys tab) and click “Upload new picture”.

The “Profile” link

Figure 84. The “Profile” link

We’ll choose a copy of the Git logo that is on our hard drive and then we get a chance to crop it.

Crop your uploaded avatar

Figure 85. Crop your avatar

Now anywhere you interact on the site, people will see your avatar next to your username.

If you happen to have uploaded an avatar to the popular Gravatar service (often used for Wordpress accounts), that avatar will be used by default and you don’t need to do this step.

Your Email Addresses

THE WAY THAT GITHUB MAPS YOUR GIT COMMITS TO YOUR USER IS BY EMAIL ADDRESS. IF YOU USE MULTIPLE EMAIL ADDRESSES IN YOUR COMMITS AND YOU WANT GITHUB TO LINK THEM UP PROPERLY, YOU NEED TO ADD ALL THE EMAIL ADDRESSES YOU HAVE USED TO THE EMAILS SECTION OF THE ADMIN SECTION.

Add all your email addresses

Figure 86. Add email addresses

IN ADD EMAIL ADDRESSES WE CAN SEE SOME OF THE DIFFERENT STATES THAT ARE POSSIBLE. THE TOP ADDRESS IS VERIFIED AND SET AS THE PRIMARY ADDRESS, MEANING THAT IS WHERE YOU’LL GET ANY NOTIFICATIONS AND RECEIPTS. THE SECOND ADDRESS IS VERIFIED AND SO CAN BE SET AS THE PRIMARY IF YOU WISH TO SWITCH THEM. THE FINAL ADDRESS IS UNVERIFIED, MEANING THAT YOU CAN’T MAKE IT YOUR PRIMARY ADDRESS. IF GITHUB SEES ANY OF THESE IN COMMIT MESSAGES IN ANY REPOSITORY ON THE SITE, IT WILL BE LINKED TO YOUR USER NOW.

Two Factor Authentication

Finally, for extra security, you should definitely set up Two-factor Authentication or “2FA”. Two-factor Authentication is an authentication mechanism that is becoming more and more popular recently to mitigate the risk of your account being compromised if your password is stolen somehow. Turning it on will make GitHub ask you for two different methods of authentication, so that if one of them is compromised, an attacker will not be able to access your account.

You can find the Two-factor Authentication setup under the Security tab of your Account settings.

2FA in the Security Tab

Figure 87. 2FA in the Security Tab

If you click on the “Set up two-factor authentication” button, it will take you to a configuration page where you can choose to use a phone app to generate your secondary code (a “time based one-time password”), or you can have GitHub send you a code via SMS each time you need to log in.

After you choose which method you prefer and follow the instructions for setting up 2FA, your account will then be a little more secure and you will have to provide a code in addition to your password whenever you log into GitHub.

Contributing to a Project

Now that our account is set up, let’s walk through some details that could be useful in helping you contribute to an existing project.

Forking Projects

If you want to contribute to an existing project to which you don’t have push access, you can “fork” the project. When you “fork” a project, GitHub will make a copy of the project that is entirely yours; it lives in your namespace, and you can push to it.

Note

HISTORICALLY, THE TERM “FORK” HAS BEEN SOMEWHAT NEGATIVE IN CONTEXT, MEANING THAT SOMEONE TOOK AN OPEN SOURCE PROJECT IN A DIFFERENT DIRECTION, SOMETIMES CREATING A COMPETING PROJECT AND SPLITTING THE CONTRIBUTORS. IN GITHUB, A “FORK” IS SIMPLY THE SAME PROJECT IN YOUR OWN NAMESPACE, ALLOWING YOU TO MAKE CHANGES TO A PROJECT PUBLICLY AS A WAY TO CONTRIBUTE IN A MORE OPEN MANNER.

This way, projects don’t have to worry about adding users as collaborators to give them push access. People can fork a project, push to it, and contribute their changes back to the original repository by creating what’s called a Pull Request, which we’ll cover next. This opens up a discussion thread with code review, and the owner and the contributor can then communicate about the change until the owner is happy with it, at which point the owner can merge it in.

To fork a project, visit the project page(any project that you want in your namespace) and click the “Fork” button at the top-right of the page.

The “Fork” button

Figure 88. The “Fork” button

After a few seconds, you’ll be taken to your new project page, with your own writeable copy of the code.

The GitHub Flow

GitHub is designed around a particular collaboration workflow, centered on PULL REQUESTS(LET YOU TELL OTHERS ABOUT CHANGES YOU'VE PUSHED TO A BRANCH IN A REPOSITORY ON GITHUB. ONCE A PULL REQUEST IS OPENED, YOU CAN DISCUSS AND REVIEW THE POTENTIAL CHANGES WITH COLLABORATORS AND ADD FOLLOW-UP COMMITS BEFORE YOUR CHANGES ARE MERGED INTO THE BASE BRANCH). 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 workflow covered in Git Branching.

Here’s how it generally works:

Fork the project.

Create a topic branch from master.

Make some commits to improve the project.

Push this branch to your GitHub project.

Open a Pull Request on GitHub.

Discuss, and optionally continue committing.

The project owner merges or closes the Pull Request.

Sync the updated master back to your fork.

This is basically the Integration Manager workflow covered in Integration-Manager Workflow, but instead of using email to communicate and review changes, teams use GitHub’s web based tools.

Let’s walk through an example of proposing a change to an open source project hosted on GitHub using this flow.

Tip

You can use the official GitHub CLI tool instead of the GitHub web interface for most things. The tool can be used on Windows, MacOS, and Linux systems. Go to the GitHub CLI homepage for installation instructions and the manual.

Creating a Pull Request

Tony is looking for code to run on his Arduino programmable microcontroller and has found a great program file on GitHub at https://github.com/schacon/blink.

The project we want to contribute to

Figure 89. The project we want to contribute to

The only problem is that the blinking rate is too fast. We think it’s much nicer to wait 3 seconds instead of 1 in between each state change. So let’s improve the program and submit it back to the project as a proposed change.

First, we click the 'Fork' button as mentioned earlier to get our own copy of the project. Our user name here is “tonychacon” so our copy of this project is at https://github.com/tonychacon/blink and that’s where we can edit it. We will clone it locally, create a topic branch, make the code change and finally push that change back up to GitHub.

$ git clone https://github.com/tonychacon/blink (1)

Cloning into 'blink'...

$ cd blink

$ git checkout -b slow-blink (2)

Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (macOS) (3)

# If you're on a Linux system, do this instead:

# $ sed -i 's/1000/3000/' blink.ino (3)

$ git diff --word-diff (4)

diff --git a/blink.ino b/blink.ino

index 15b9911..a6cc5a5 100644

--- a/blink.ino

+++ b/blink.ino

@@ -18,7 +18,7 @@ void setup() {

// the loop routine runs over and over again forever:

void loop() {

digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)

[-delay(1000);-]{+delay(3000);+} // wait for a second

digitalWrite(led, LOW); // turn the LED off by making the voltage LOW

[-delay(1000);-]{+delay(3000);+} // wait for a second

}

$ git commit -a -m 'Change delay to 3 seconds' (5)

[slow-blink 5ca509d] Change delay to 3 seconds

1 file changed, 2 insertions(+), 2 deletions(-)

(BEFORE YOU PUSH YOU HAVE TO ESTABLISH CONNECTION WITH GITHUB REMOTE BRANCH AND YOU HAVE TO USE THE COMMAND -git remote set-url origin https://USERNAME:TOKEN@github.com/USERNAME/REPONAME.git ON FURNISHING THE DETAILS WE GET THE BELOW COMMAND, TOKEN IS TO BE GENERATED BY FOLLOWING A SET OF STEPS IN GITHUB ACCOUNT)

saivinil\_pratap@TIGER02143:~/blink/blink$ git remote set-url origin https://saivinilpratap-ta:ghp\_lmX2ES2m3kP9HyeJsQDCH9FBXVzGwl07ygHM@github.com/saivinilpratap-ta/blink.git

$ git push origin slow-blink (6)

Username for 'https://github.com': tonychacon

Password for 'https://tonychacon@github.com':

Counting objects: 5, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

To https://github.com/tonychacon/blink

\* [new branch] slow-blink -> slow-blink

Clone our fork of the project locally.

Create a descriptive topic branch.

Make our change to the code.

Check that the change is good.

Commit our change to the topic branch.

Push our new topic branch back up to our GitHub fork.

Now if we go back to our fork on GitHub, we can see that GitHub noticed that we pushed a new topic branch up and presents us with a big green button to check out our changes and open a Pull Request to the original project.

You can alternatively go to the “Branches” page at https://github.com/<user>/<project>/branches to locate your branch and open a new Pull Request from there.

Pull Request button

Figure 90. Pull Request button

If we click that green button, we’ll see a screen that asks us to give our Pull Request a title and description. It is almost always worthwhile to put some effort into this, since a good description helps the owner of the original project determine what you were trying to do, whether your proposed changes are correct, and whether accepting the changes would improve the original project.

We also see a list of the commits in our topic branch that are “ahead” of the master branch (in this case, just the one) and a unified diff of all the changes that will be made should this branch get merged by the project owner.

Pull Request creation

Figure 91. Pull Request creation page

When you hit the 'Create pull request' button on this screen, the owner of the project you forked will get a notification that someone is suggesting a change and will link to a page that has all of this information on it.

Note

Though Pull Requests are used commonly for public projects like this when the contributor has a complete change ready to be made, it’s also often used in internal projects at the beginning of the development cycle. Since you can keep pushing to the topic branch even after the Pull Request is opened, it’s often opened early and used as a way to iterate on work as a team within a context, rather than opened at the very end of the process.

Iterating on a Pull Request

At this point, the project owner can look at the suggested change and merge it, reject it or comment on it. Let’s say that he likes the idea, but would prefer a slightly longer time for the light to be off than on.

Where this conversation may take place over email in the workflows presented in Distributed Git, on GitHub this happens online. The project owner can review the unified diff and leave a comment by clicking on any of the lines.

PR line comment

Figure 92. Comment on a specific line of code in a Pull Request

Once the maintainer makes this comment, the person who opened the Pull Request (and indeed, anyone else watching the repository) will get a notification. We’ll go over customizing this later, but if he had email notifications turned on, Tony would get an email like this:

Email notification

Figure 93. Comments sent as email notifications

Anyone can also leave general comments on the Pull Request. In Pull Request discussion page we can see an example of the project owner both commenting on a line of code and then leaving a general comment in the discussion section. You can see that the code comments are brought into the conversation as well.

PR discussion page

Figure 94. Pull Request discussion page

Now the contributor can see what they need to do in order to get their change accepted. Luckily this is very straightforward. Where over email you may have to re-roll your series and resubmit it to the mailing list, with GitHub you simply commit to the topic branch again and push, which will automatically update the Pull Request. In Pull Request final you can also see that the old code comment has been collapsed in the updated Pull Request, since it was made on a line that has since been changed.

Adding commits to an existing Pull Request doesn’t trigger a notification, so once Tony has pushed his corrections he decides to leave a comment to inform the project owner that he made the requested change.

PR final

Figure 95. Pull Request final

An interesting thing to notice is that if you click on the “Files Changed” tab on this Pull Request, you’ll get the “unified” diff — that is, the total aggregate difference that would be introduced to your main branch if this topic branch was merged in. In git diff terms, it basically automatically shows you git diff master…​<branch> for the branch this Pull Request is based on. See Determining What Is Introduced for more about this type of diff.

The other thing you’ll notice is that GitHub checks to see if the Pull Request merges cleanly and provides a button to do the merge for you on the server. This button only shows up if you have write access to the repository and a trivial merge is possible. If you click it GitHub will perform a “non-fast-forward” merge, meaning that even if the merge could be a fast-forward, it will still create a merge commit.

If you would prefer, you can simply pull the branch down and merge it locally. If you merge this branch into the master branch and push it to GitHub, the Pull Request will automatically be closed.

This is the basic workflow that most GitHub projects use. Topic branches are created, Pull Requests are opened on them, a discussion ensues, possibly more work is done on the branch and eventually the request is either closed or merged.

Note

Not Only Forks

It’s important to note that you can also open a Pull Request between two branches in the same repository. If you’re working on a feature with someone and you both have write access to the project, you can push a topic branch to the repository and open a Pull Request on it to the master branch of that same project to initiate the code review and discussion process. No forking necessary.

Advanced Pull Requests

Now that we’ve covered the basics of contributing to a project on GitHub, let’s cover a few interesting tips and tricks about Pull Requests so you can be more effective in using them.

Pull Requests as Patches

It’s important to understand that many projects don’t really think of Pull Requests as queues of perfect patches that should apply cleanly in order, as most mailing list-based projects think of patch series contributions. Most GitHub projects think about Pull Request branches as iterative conversations around a proposed change, culminating in a unified diff that is applied by merging.

This is an important distinction, because generally the change is suggested before the code is thought to be perfect, which is far more rare with mailing list based patch series contributions. This enables an earlier conversation with the maintainers so that arriving at the proper solution is more of a community effort. When code is proposed with a Pull Request and the maintainers or community suggest a change, the patch series is generally not re-rolled, but instead the difference is pushed as a new commit to the branch, moving the conversation forward with the context of the previous work intact.

For instance, if you go back and look again at Pull Request final, you’ll notice that the contributor did not rebase his commit and send another Pull Request. Instead they added new commits and pushed them to the existing branch. This way if you go back and look at this Pull Request in the future, you can easily find all of the context of why decisions were made. Pushing the “Merge” button on the site purposefully creates a merge commit that references the Pull Request so that it’s easy to go back and research the original conversation if necessary.

Keeping up with Upstream

If your Pull Request becomes out of date or otherwise doesn’t merge cleanly, you will want to fix it so the maintainer can easily merge it. GitHub will test this for you and let you know at the bottom of every Pull Request if the merge is trivial or not.

PR merge failure

Figure 96. Pull Request does not merge cleanly

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.

For example, let’s say that in the “tonychacon” example we were using before, the original author made a change that would create a conflict in the Pull Request. Let’s go through those steps.

$ git remote add upstream https://github.com/schacon/blink (1)

$ git fetch upstream (2)

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (3/3), done.

Unpacking objects: 100% (3/3), done.

remote: Total 3 (delta 0), reused 0 (delta 0)

From https://github.com/schacon/blink

\* [new branch] master -> upstream/master

$ git merge upstream/master (3)

Auto-merging blink.ino

CONFLICT (content): Merge conflict in blink.ino

Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino (4)

$ git add blink.ino

$ git commit

[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \

into slower-blink

$ git push origin slow-blink (5)

Counting objects: 6, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (6/6), done.

Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.

Total 6 (delta 2), reused 0 (delta 0)

To https://github.com/tonychacon/blink

ef4725c..3c8d735 slower-blink -> slow-blink

Add the original repository as a remote named upstream.

Fetch the newest work from that remote.

Merge the main branch of that repository into your topic branch.

Fix the conflict that occurred.

Push back up to the same topic branch.

Once you do that, the Pull Request will be automatically updated and re-checked to see if it merges cleanly.

PR fixed

Figure 97. Pull Request now merges cleanly

One of the great things about Git is that you can do that continuously. If you have a very long-running project, you can easily merge from the target branch over and over again and only have to deal with conflicts that have arisen since the last time that you merged, making the process very manageable.

If you absolutely wish to rebase the branch to clean it up, you can certainly do so, but it is highly encouraged to not force push over the branch that the Pull Request is already opened on. If other people have pulled it down and done more work on it, you run into all of the issues outlined in The Perils of Rebasing. Instead, push the rebased branch to a new branch on GitHub and open a brand new Pull Request referencing the old one, then close the original.

References

Your next question may be “How do I reference the old Pull Request?”. It turns out there are many, many ways to reference other things almost anywhere you can write in GitHub.

Let’s start with how to cross-reference another Pull Request or an Issue. All Pull Requests and Issues are assigned numbers and they are unique within the project. For example, you can’t have Pull Request #3 and Issue #3. If you want to reference any Pull Request or Issue from any other one, you can simply put #<num> in any comment or description. You can also be more specific if the Issue or Pull request lives somewhere else; write username#<num> if you’re referring to an Issue or Pull Request in a fork of the repository you’re in, or username/repo#<num> to reference something in another repository.

Let’s look at an example. Say we rebased the branch in the previous example, created a new pull request for it, and now we want to reference the old pull request from the new one. We also want to reference an issue in the fork of the repository and an issue in a completely different project. We can fill out the description just like Cross references in a Pull Request.

PR references

Figure 98. Cross references in a Pull Request

When we submit this pull request, we’ll see all of that rendered like Cross references rendered in a Pull Request.

PR references rendered

Figure 99. Cross references rendered in a Pull Request

Notice that the full GitHub URL we put in there was shortened to just the information needed.

Now if Tony goes back and closes out the original Pull Request, we can see that by mentioning it in the new one, GitHub has automatically created a trackback event in the Pull Request timeline. This means that anyone who visits this Pull Request and sees that it is closed can easily link back to the one that superseded it. The link will look something like Link back to the new Pull Request in the closed Pull Request timeline.

PR closed

Figure 100. Link back to the new Pull Request in the closed Pull Request timeline

In addition to issue numbers, you can also reference a specific commit by SHA-1. You have to specify a full 40 character SHA-1, but if GitHub sees that in a comment, it will link directly to the commit. Again, you can reference commits in forks or other repositories in the same way you did with issues.

GitHub Flavored Markdown

Linking to other Issues is just the beginning of interesting things you can do with almost any text box on GitHub. In Issue and Pull Request descriptions, comments, code comments and more, you can use what is called “GitHub Flavored Markdown”. Markdown is like writing in plain text but which is rendered richly.

See An example of GitHub Flavored Markdown as written and as rendered for an example of how comments or text can be written and then rendered using Markdown.

Example Markdown

Figure 101. An example of GitHub Flavored Markdown as written and as rendered

The GitHub flavor of Markdown adds more things you can do beyond the basic Markdown syntax. These can all be really useful when creating useful Pull Request or Issue comments or descriptions.

Task Lists

The first really useful GitHub specific Markdown feature, especially for use in Pull Requests, is the Task List. A task list is a list of checkboxes of things you want to get done. Putting them into an Issue or Pull Request normally indicates things that you want to get done before you consider the item complete.

You can create a task list like this:

- [X] Write the code

- [ ] Write all the tests

- [ ] Document the code

If we include this in the description of our Pull Request or Issue, we’ll see it rendered like Task lists rendered in a Markdown comment.

Example Task List

Figure 102. Task lists rendered in a Markdown comment

This is often used in Pull Requests to indicate what all you would like to get done on the branch before the Pull Request will be ready to merge. The really cool part is that you can simply click the checkboxes to update the comment — you don’t have to edit the Markdown directly to check tasks off.

What’s more, GitHub will look for task lists in your Issues and Pull Requests and show them as metadata on the pages that list them out. For example, if you have a Pull Request with tasks and you look at the overview page of all Pull Requests, you can see how far done it is. This helps people break down Pull Requests into subtasks and helps other people track the progress of the branch. You can see an example of this in Task list summary in the Pull Request list.

Example Task List

Figure 103. Task list summary in the Pull Request list

These are incredibly useful when you open a Pull Request early and use it to track your progress through the implementation of the feature.

Code Snippets

You can also add code snippets to comments. This is especially useful if you want to present something that you could try to do before actually implementing it as a commit on your branch. This is also often used to add example code of what is not working or what this Pull Request could implement.

To add a snippet of code you have to “fence” it in backticks.

```java

for(int i=0 ; i < 5 ; i++)

{

System.out.println("i is : " + i);

}

```

If you add a language name like we did there with 'java', GitHub will also try to syntax highlight the snippet. In the case of the above example, it would end up rendering like Rendered fenced code example.

Rendered fenced code

Figure 104. Rendered fenced code example

Quoting

If you’re responding to a small part of a long comment, you can selectively quote out of the other comment by preceding the lines with the > character. In fact, this is so common and so useful that there is a keyboard shortcut for it. If you highlight text in a comment that you want to directly reply to and hit the r key, it will quote that text in the comment box for you.

The quotes look something like this:

> Whether 'tis Nobler in the mind to suffer

> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?

Once rendered, the comment will look like Rendered quoting example.

Rendered quoting

Figure 105. Rendered quoting example

Emoji

Finally, you can also use emoji in your comments. This is actually used quite extensively in comments you see on many GitHub Issues and Pull Requests. There is even an emoji helper in GitHub. If you are typing a comment and you start with a : character, an autocompleter will help you find what you’re looking for.

Emoji autocompleter

Figure 106. Emoji autocompleter in action

Emojis take the form of :<name>: anywhere in the comment. For instance, you could write something like this:

I :eyes: that :bug: and I :cold\_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda\_face:

When rendered, it would look something like Heavy emoji commenting.

Emoji

Figure 107. Heavy emoji commenting

Not that this is incredibly useful, but it does add an element of fun and emotion to a medium that is otherwise hard to convey emotion in.

Note

There are actually quite a number of web services that make use of emoji characters these days. A great cheat sheet to reference to find emoji that expresses what you want to say can be found at:

https://www.webfx.com/tools/emoji-cheat-sheet/

Images

This isn’t technically GitHub Flavored Markdown, but it is incredibly useful. In addition to adding Markdown image links to comments, which can be difficult to find and embed URLs for, GitHub allows you to drag and drop images into text areas to embed them.

Drag and drop images

Figure 108. Drag and drop images to upload them and auto-embed them

If you look at Drag and drop images to upload them and auto-embed them, you can see a small “Parsed as Markdown” hint above the text area. Clicking on that will give you a full cheat sheet of everything you can do with Markdown on GitHub.

Keep your GitHub public repository up-to-date

Once you’ve forked a GitHub repository, your repository (your "fork") exists independently from the original. In particular, when the original repository has new commits, GitHub informs you by a message like:

This branch is 5 commits behind progit:master.

But your GitHub repository will never be automatically updated by GitHub; this is something that you must do yourself. Fortunately, this is very easy to do.

One possibility to do this requires no configuration. For example, if you forked from https://github.com/progit/progit2.git, you can keep your master branch up-to-date like this:

$ git checkout master (1)

$ git pull https://github.com/progit/progit2.git (2)

$ git push origin master (3)

If you were on another branch, return to master.

Fetch changes from https://github.com/progit/progit2.git and merge them into master.

Push your master branch to origin.

This works, but it is a little tedious having to spell out the fetch URL every time. You can automate this work with a bit of configuration:

$ git remote add progit https://github.com/progit/progit2.git (1)

$ git fetch progit (2)

$ git branch --set-upstream-to=progit/master master (3)

$ git config --local remote.pushDefault origin (4)

Add the source repository and give it a name. Here, I have chosen to call it progit.

Get a reference on progit’s branches, in particular master.

Set your master branch to fetch from the progit remote.

Define the default push repository to origin.

Once this is done, the workflow becomes much simpler:

$ git checkout master (1)

$ git pull (2)

$ git push (3)

If you were on another branch, return to master.

Fetch changes from progit and merge changes into master.

Push your master branch to origin.

This approach can be useful, but it’s not without downsides. Git will happily do this work for you silently, but it won’t warn you if you make a commit to master, pull from progit, then push to origin — all of those operations are valid with this setup. So you’ll have to take care never to commit directly to master, since that branch effectively belongs to the upstream repository.

tagging: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).

Listing Your Tags:

Listing the existing tags in Git is straightforward. Just type git tag (with optional -l or --list):

$ git tag

v1.0

v2.0

This command lists the tags in alphabetical order; the order in which they are displayed has no real importance.

You can also search for tags that match a particular pattern. The Git source repo, for instance, contains more than 500 tags. If you’re interested only in looking at the 1.8.5 series, you can run this:

$ git tag -l "v1.8.5\*"

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

Creating Tags:

Git supports two types of tags: lightweight and annotated.

A LIGHTWEIGHT TAG is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

ANNOTATED TAGS, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It’s generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don’t want to keep the other information, lightweight tags are available too.

Annotated Tags

CREATING AN ANNOTATED TAG IN GIT IS SIMPLE. THE EASIEST WAY IS TO SPECIFY -A WHEN YOU RUN THE TAG COMMAND:

$ git tag -a v1.4 -m "my version 1.4"

$ git tag

v0.1

v1.3

v1.4

THE -M SPECIFIES A TAGGING MESSAGE, which is stored with the tag. If you don’t specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the git show command:

$ git show v1.4

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4 (this is the message)

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

That shows the tagger information, the date the commit was tagged, and the annotation message before showing the commit information.

Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file — no other information is kept. To create a lightweight tag, don’t supply any of the -a, -s, or -m options, just provide a tag name:

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

This time, if you run git show on the tag, you don’t see the extra tag information. The command just shows the commit:

$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

Tagging Later

You can also tag commits after you’ve moved past them. Suppose your commit history looks like this:

$ git tag -a v1.2 9fceb02

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.

$ git push origin --tags

$######################################################$######################################################

--notes:

- let's suppose that we have created a local project using git and we want to push it to github. before we do that, we have to establish a connection with github, then we have to create a repository in github then, we have to use the following command:

git remote add origin <repo link(ssh one is preferred)>

- not everytime we will go copy the link and perform git push, to push to a default repo (which is already connected, use the below command):

git push --set-upstream <branch name> or git push -u <branch name>

-to create a new branch and checkout to it in one command:

-there are 2 branches-master,feature.suppose you are in a branch named master and you want to compare the changes of feature branch,use the following command:

git diff <branch name>

ex: git diff feature

if you want to check the changes in the current branch that are made after the recent commit, then just checkout to the branch and then just give

git diff

-Pull Request or PR:

basic request to pull the code from current branch to target branch. suppose you are in feature branch and want to send that code to master branch, then you have to raise a PullRequest from feature branch to master branch,now anyone can review the code, or can comment on them, or ask you to make further changes, after the PR is approved, you should delete the sub branch from where you requested PR.

-basic workflow in git:

1:

1.1:if you are starting new you might want to create a folder for project and then navigate to that directory in cli(command line interface) and then run git init

1.2:if you are cloning a project(from github) you have to create folder for project and then use "git clone" and copy the project in to yous

1.3:if you want to work on existing popular repo's then you might want to fork their work and save it as a repo in your igthub and then perform git clone (covering the same steps that are included in 1.2)

2:after having project on your local, then start making changes, all these changes stay untracked as long as you dont add them using git command, they will be in unstaged state.

3:after you add them, they will be moved to staged state.

4:these changes that are in staged state have to be committed, so that they can be pushed to your github repo.

5:now after pushing, let's assume that someone has cloned your project and made some changes and pushed code, so now you have to take a clone and then start working again and continue the cycle from step 2 above

git checkout -b <branch name>

configuration of git:

1:set up username and email

git config --global user.name " "

git config --global user.email " "

2:after giving details check wether they are populated or not using the following command:

git config --global -l

-generally branches that are merged in to main branch have very less chances to be reused so it is better to delete them if we are sure of the developments

-to delete a branch

git branch -d <branch name>

-suppose you had commited and you want to uncommit the changes that are made, then you can use the following command:

git reset HEAD~1 (HEAD is used here because the head is pointed to the snapshot where last commit is performed)

if you want to reset it to a particular version which is 3-4 commits previous to the current one, you can use the sha hash that you get in git log (check the alphanumeric that you get after each commit when you use git log)

command: git reset <hash>

ex: git reset d2351b966893e7722d8d7ad1d86e29b04b583657(this can be dwelled in detail, but for now just remember that it is possible for you to revert back and get previous versions)

let us say that you are working on a branch and you created a new branch and switched to it and then,you modified a file, the changes that are performed here will not be confined to this branch unless you commit them in that branch,

-head is a pointer to last commit on the current branch

-remote is simply a reference or pointer to a repository , that repository might be somehere not local but on current network or on an enterprise or on internet

ex:github,gitlab,bitbucket and others

-default branch in a git repository is master

-to get an information on a particular command in git

use:git help <command name>

-suppose you have a project in you rlocal and you want to create a repository, use the below command:

git init <PROJECT NAME> (PROJECT NAME IS NEEDED WHEN YOU WANT TO CREATE A PROJECT FRESH, AS IT WILL CREATE A NEW PROJECT FOLDER IN CWD WITH THE NAME MENTIONED)

-removing or moving can be done with and without git's knowledge(

with git's knowledge-(it is not possible in some of the versions )

git mv <file> target directory

git rm <file\_path>

-notice the git word in above command, it means git will track them for our ease

without git knowledge:

mv <file> target directory

rm <file\_path>

-there are some files which you dont want the git to track and inform you of, for that you can create a .gitignore in the project folder and

let's suppose you don't want to track all the log files, then just add \*.log in .gitignore file

if you want to ignore only error.log, then just add error.log in .gitignore so that git will track all other log files except the error.log file

-Git's staging area is sometimes referred to as the index

-The easiest way to add all files to your Git repository is to use the “git add” command followed by the “-A” option for “all”. In this case, the new (or untracked), deleted and modified files will be added to your Git staging area.

-git commit -a

Using the -a flag allows you to skip over running “git add” on all changed files. This is because the -a flag instructs git to stage all modified files. HOWEVER, THE -A FLAG WILL NOT STAGE ANY NEW FILES.

-The .gitignore file in your repository must be version controlled like any other file in Git

-suppose you have a project in your local git and you have to establish a connection with a remote repo on github, us the below command

-when you want to get some one's project, first you want to fork it to your github account and then clone it in to your local using git clone command.

-git remote add origin git@github.com:username/reponame.git

-ORIGIN is the primary remote location name used by convention in Git

-it is a good habit to take a pull before you push to repository for the following reasons:

It allow you to integrate and test other's changes with your changes

It's a good habit to develop

It's not required when you are the sole user of the remote repository

Your push might fail if others have made changes to the remote repository since your last pull

-github doesnot allow you to create an empty folder, to have a folder with no contents Standard Operating Procedure(SOP) is to create an empty .gitkeep file

-git status and its tracking will be based on .git, that is why if you change the name of the project, it becomes a different folder, but it will have the same track of the changes that happened

-when you want to raise a pull request(PR) (PR or issue is an concept in github and it does not come anywhere in your local git), before you actually do that, you should create a branch in local, make changes and try to commit them and the push them, after you push the code, then you will have the option to raise PR, if you already have been working on an issue in the current branch and if that is going to solve the issue, when raising a PR, you can mention closes #<issue number>, so that when a PR is merged then the issue will automatically get closed

-shell command to download the data:

curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv

$######################################################$######################################################

Here are some recommendation for a GIT based development workflow for your day-to-day work:

# Naming convention for a branch to be created

1:Create an Issue in the Issue tracker before starting any work that requires changing code in the repository. The issue should capture the requirements of a task. Say, you need to add a new feature or fix a bug in the code. You start by creating a new Issue if one doesn’t already exist and describe the feature or the bug in some detail.

2:Once you create an issue, create a branch to work on the issue. It is often a good idea to use the issue id as part of the branch name. A good convention to follow is to name your branches as <fix-type>/<issue id>/<short moniker>. Here, <fix-type> indicates the type of code change:

enh - enhancement

feat - new feature

fix - a bug fix

doc - a documentation update

test - update tests

Here are some example branch names, see if you can guess the intent of the code changes in the branch

enh/issue#100/support-s3

fix/issue#200/missing-config

doc/issue#32/update-dev-guide

As you work on your project, you will accumulate a lot of branches. You don’t want to delete your local branches as GIT keeps them out of your way and its good to have them in case you need them later. The above scheme helps you organize and search your branches effectively.

3:As you start working on your feature or bug, it often helps to start with a test case. For bugs, the test should fail and then you start working on a fix to get the test to pass. For a new feature, you can write tests to mock out its behavior. This is called Test Driven Development (TDD) and is a good way to streamline your development process.

4:As you work, make small logical commits and push to the remote repository frequently. This ensures that you don’t lose your changes if you lose your local copy by accident. Use short useful commit messages as these can be helpful when you look at the log/record of all the changes done. Consider using the <fix-type> labels to express the intent of a commit. Note that the commit messages have an active voice. This is a typical convention followed by many projects and is a good practice to adopt. Here is an example short commit messages:

ENH: Support S3 backend for file storage.

BUG: Fix test failure due to incorrect path manipulation.

DOC: Fix typo in developer guide

TEST: Add tests for S3 backend support.

CLN: Format code

MAINT: Update scikit package dependency to latest version.

5:Once you are ready to share your changes with the team, verify that you have followed coding standards and style guidelines as followed by your team.

6:Before creating a PR, sync your local repository with remote repository using a git pull. If the master has changed since you created your branch, you have one of two options:

Do a git rebase on top of the latest master. This will apply all your changes one-by-one using the updated master and give you an opportunity to fix any conflicts.

Merge the latest master onto your branch and fix any conflicts. Once fixed, push your branch again.

7:Create a PR with a meaningful title and a good description. The title should clearly convey the intent of the PR. The description should atleast have details on

The contributions from the PR

Any implementation/design choices made

Reference(s) to the issues and/or other PRs that might be related to this.

8:Before you assign a reviewer, take a look at the diff view showing the changes from the PR and review it once and fix any glaring issues. Also, verify that the CI tests pass. Now, request a review from your team members.

9:Note that you don’t have to wait until you have completed everything to open a PR. Sometimes, it is very useful to get early feedback on your approach before you go too far down the wrong road. In such cases the convention is to add a prefix [WIP] (Work In Progress) to indicate that the PR is not yet complete. You can ask a team member for a review and be specific about what you are looking for. Once you are done and ready for a final review, ensure all the requirements in the previous step are taken care of and remove the WIP prefix and request a review.

commonly used commands in git:

# commonly used commands/scenarios in git

to stop adding particular files -> git add --all -- ':!path/to/file1.txt' ':!path/to/file2.txt' ':!path/to/folder1/\*' or git add -- . -- ':!path/to/file1.txt' ':!path/to/file2.txt' ':!path/to/folder1/\*'

to ignore a git directory from being pushed, you create a .gitignore file in the same directory which you want to ignore and add \* in the .gitignore file inside the directory, which will ignore all the files

when you do git pull, it pulls newly remotely created branches as well, but for you to see them in local, you need to give git branch -av

to push to a remote branch that does not exist in your local (simply means you are pushing code from current branch to a remote branch that is not in your local)

syntax: git push <remote> <local\_branch>:<remote\_name>

example: git push origin main:preprod (preprod branch is not in your system but you are pushing code from your local main branch to a remote branch)

-if you are not able to see the remote branches in your local, and you want them in your local, just give the command "git checkout <name of remote branch that you want in your local>

-whenever you have created an issue branch and then made some changes to code, you can merge it with main in 2 ways,1: by checking out to issue branch and then merging with main, 2: checking out to main branch and merge issue branch with it, the second way is better because this way main branch will have the final push

-if you want to copy changes from a branch X to branch Y and then

1: git checkout Y #checking out to branch Y

2:git pull origin X #to make sure that we are taking latest code of branch X

3:git pull origin Y #to make sure that we are taking latest code of branch Y

4:git push origin Y # to push all the combined changes to branch Y

## To check whether user is logged in or not

git config -l

## to list all available branches in current project

1. To see local branches, run this command: git branch.
2. To see remote branches, run this command: git branch -r.
3. To see all local and remote branches, run this command: git branch -a.

## cloning an existing repository

you can clone an existing git repository to your local after you are logged in and provided that you have enough permissions to that repo from your logged in account

you will have that repo as a folder in your local, you might see only one (main) branch (because only one branch is downloaded), but all other branches are in remote state (not downloaded, but exist), if you want to switch to that remote branch

## if you have changes that you don’t want to discard and at the same time to pull from remote branch or to switch branch

you can stash the changes, take pull from the branch and then unstash (apply the stashed code) on the required branch,

link: <https://youtu.be/H5BLEPhqxe8>

## when you want to push your changes to repository

1. Be in the branch where you want to push the code
2. Do a git pull
3. Merge changes (if any)
4. Do git push

## whenever you are starting new changes in a particular branch

1. Check out to that branch after committing the recent changes
2. Take pull from remote branch using **git pull**
3. Start working on new changes

## While creating a new branch

1. Before you create a new branch, be on the branch where you want to create a new branch, take fresh pull from target branch using **git pull**
2. Then create a new branch (easy in vs code, just need to click on branch and type the new branch name and press enter)
3. Publish the branch to remote github repository

## Deleting a local branch

1. You should not be on the branch you want to delete, check out to other branch
2. The branch is local, if it is not published to remote github
3. Run the command git branch --delete <branch that you want to delete>

## Undoing recent commit in local so that you get code in remote branch

There might be scenarios where you want to discard your local changes and just want your local to mimic the code that is already there in remote

git reset --hard origin/<branch\_name> to discard local commit

## renaming local branch

checkout to that branch, then use the below command

git branch -m <new name you want to give the branch>

## running **git fetch** periodically.

The main difference between git fetch and git pull is that git fetch only updates your local copy of the remote repository, while git pull updates your local copy of the remote repository and then merges the changes into your current branch.

Here is a table that summarizes the differences between git fetch and git pull:

|  |  |
| --- | --- |
| Command | Description |
| git fetch | Updates your local copy of the remote repository by downloading the  latest changes from the remote. |
| git pull | Updates your local copy of the remote repository and then merges  the changes into your current branch. |

Here are some additional details about the two commands:

* git fetch does not change your local working copy.
* git pull may change your local working copy if there are any conflicts between the changes that you have made and the changes that have been made to the remote repository.
* git fetch is a safe command to run. It will not overwrite any of your local changes.
* git pull can be a dangerous command to run if you have uncommitted changes in your local working copy. If there are any conflicts between the changes that you have made and the changes that have been made to the remote repository, you may lose your local changes.

In general, it is a good idea to run git fetch periodically, especially if you are working on a project with other people. This will help to ensure that your local copy of the repository is up-to-date and that you are working with the most recent version of the code. You can then use git pull to merge the changes into your current branch when you are ready.

## To unstage a committed (staged) file

git restore --staged <file relative path>

# Pending hands on tasks

Publish local branch to remote repo using git commands