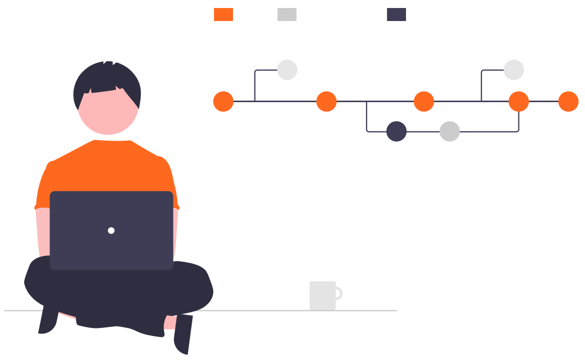
Home About

Account



Git: Merge, Cherry-Pick & Rebase

An unconventional guide

Last updated on January 26, 2022 - 11 comments

Quick Links

↑ Top

You can use this guide to get a deep understanding of how Git's merges, rebases

& cherry-picks work under the hood, so that you'll never fear them again.

(**Editor's note**: At ~5500 words, you probably don't want to try reading this on a mobile device. Bookmark it and come back later. And even on a desktop, ~~eat~~ read this elephant one bite at a time.)

Introduction

Sure, everyone and their grandmother use Git and seems to be comfortable with it.

But did you ever botch a merge and then your solution was to delete and re-clone your repository? Without quite knowing what went wrong and why?

Or did a rebase suddenly make tens of merge conflicts pop up, one after another and you didn't know what the hell was going on?

In short, do you have nagging doubts, whenever it comes to *merging*, *rebasing* and *cherry-picking*?

Fear not, you've come to the right place: The remainder of this guide will help you get rid of those fears.

(Teaser: By the end of this article, you'll understand that a `git cherry-pick` is essentially just a `git merge`. And a `git rebase` is essentially just a `git cherry-pick`? Sounds crazy? Read on!)

Git Storage Internals

Before you jump right into the nitty-gritty details of merging, let's have a look at how Git stores your files and commits.

It might seem a bit weird to start off with internal details, but take a leap of faith: Those internals are the building block for everything else in this guide, so you'll need to know them first.

Scenario: Committing Two Files

Open up your terminal and execute the following commands.

```
# create a git repo in a directory of your liking

mkdir gitinternals
```

```
cd gitinternals
git init -b main

## add two .txt files and commit them

echo "-TODO-" > LICENSE.txt
echo "a marcobehler.com guide" > README.txt

git add LICENSE.txt
git add README.txt
git commit -m "Project Setup"

## update README.txt's contents

echo "a git guide" > README.txt

git add README.txt
git commit -m "Updated README"
```

You created two .txt files in a first commit, then updated the contents of one file (`README.txt`) in a second commit.

Here's a question for you: How do you think Git will store those two commits, or rather the two versions of `README.txt`?

- Will it store full files, i.e. `a marcobehler.com guide` AND `a git guide`, somewhere?
- Will it store deltas, something like `a (-marcobehler.com)(+git) guide` (pseudo-code)?

Bonus question: How the hell would the answer to this help with merging or rebasing?

Let's find out!

Inspecting Git repos: 'git cat-file'

Let's execute a `git log` in your repository, and you'll get output similar to this:

```
# in your repository's directory
git log
```

```
# Project Setup

commit 142e5cf36d9f2047f24341883bd564b1d5170370 (HEAD -> main)
Author: Marco Behler <marco@marcobehler.com>
Date:   Tue Dec 28 09:54:44 2021 +0100

    Updated README

commit 715247c8426d3c16881539118e1eafeb38439b1c
Author: Marco Behler <marco@marcobehler.com>
Date:   Tue Dec 28 09:54:25 2021 +0100

    Project Setup
```

So far, nothing surprising - you'll see your two commits. Something that you've seen, but probably ignored plenty of times are `commit ids`. Here's the second commit's id.

```
commit 142e5cf36d9f2047f24341883bd564b1d5170370
```

More specifically, `142e5cf36d9f2047f24341883bd564b1d5170370` is not just a random id, it's a SHA-1 hash.

But, what *exactly* has been hashed here?

Instead of spoiling the answer, let's use another built-in git command: `git cat-file`. It basically allows you to have a look at *something* which git stores *somewhere* in your repository's `.git` folder, given that you happen to know its SHA1-hash. Sounds useful, right?

Execute the following command (and make sure to try this with the SHA1 hash that you are getting for your commit)

```
# make sure to change the SHA1-hash!
git cat-file -p 142e5cf36d9f2047f24341883bd564b1d5170370
```

(Note: The `-p` option makes sure to pretty-print its output.)

You'll get output similar to this:

```
# git cat-file's output
tree c4548e069652a6825894699ef7740a620ea0a6a8
parent 715247c8426d3c16881539118e1eafeb38439b1c
author Marco Behler <marco@marcobehler.com> 1641459065 +0100
committer Marco Behler <marco@marcobehler.com> 1641459065 +0100

Updated README
```

Tada! This is what a commit looks like in Git. It's a text file with...6 lines (well 5, and an empty one to delimit your commit message from the rest). Yes, really.

And if you put those lines into a `sha1sum()`, function you'll end up with your SHA1 hash : `142e5cf36d9f2047f24341883bd564b1d5170370`!

For the advanced reader, Git doesn't exactly do sha1sum(filecontent), it actually does a sha1sum(header + filecontent) - but we'll cover this in a bit.

Now, some of those lines from your commit (file) you'll be familiar with:

```
# who committed the file?
committer Marco Behler <marco@marcobehler.com> 1641459065 +0100

# what's the commit message?
Updated README
```

Whereas some other parts of the commit probably look unfamiliar:

```
tree c4548e069652a6825894699ef7740a620ea0a6a8
parent 715247c8426d3c16881539118e1eafeb38439b1c
```

Let's (rightly) assume for now that `parent(s)` simply references the commit that came before the current commit. Then, what does the `tree` line stand for? Execute another `git cat-file` to find out!

```
# make sure to change the SHA1-hash to that of your tree!
git cat-file -p c4548e069652a6825894699ef7740a620ea0a6a8
```

Look, this tree seems to be yet another text file, referencing (snapshots of) all the files in your repository at the time of the commit!

```
100644 blob ddd3b7b6335a636af9a9241096455e834f12f636    LICENSE.txt
100644 blob 773fc76fe191ceff24259d4e66efc90e86093b0c    README.txt
```

Can this be true? Well, you'll find out by doing one last *git cat-file*, this time using *README.txt's* hash.

```
git cat-file -p 773fc76fe191ceff24259d4e66efc90e86093b0c
```

Which leads to the following output:

```
"a git guide"
```

Does this look familiar? Yes, it is a snapshot of your *README.txt* file, at the time of the *second commit*, i.e. when you updated the readme. Which means that it does look like Git stores the full file contents for every commit (assuming the contents have changed)?

Well, to be sure, let's repeat the *git cat-file* game for the first commit (which serves as a great exercise, so refer back to the *git log* output and repeat the steps!). You'll end up with something like this:

```
# cat'ing README.txt snapshotted during the first commit
git cat-file -p fe066d3f7568e13ef031b495e35c94be91b6366c

"a marcobehler.com guide"
```

Take-Away: Git doesn't store deltas between commits, it always stores snapshots, i.e. the *full file*, for every commit (as long as the file changed and its SHA1-hash is not already in your repository).

This is also the reason why Git is not a great choice for projects with (mostly) many binary assets, that frequently change.

.Git Folder

Here’s another exercise. In your project folder, go to your `.git` subfolder, more specifically `.git/objects`.

```
# inside your project folder
cd .git/objects

# use 'dir' on Windows
ls -l
```

You’ll get output like this:

```
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 11
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 14
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 71
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 77
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 c4
drwxrwxrwx 1 marco marco 512 Jan  6 09:50 dd
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 fe
drwxrwxrwx 1 marco marco 512 Jan  6 09:50 info
drwxrwxrwx 1 marco marco 512 Jan  6 09:50 pack
```

Now, take note of the SHA1-hash of the `README.txt` file for either of your two commits, like `142e5cf36d9f2047f24341883bd564b1d5170370` in my case. More specifically, have a look at the first two numbers/letters, that’s `14` in my case. Look, there’s a `14` subdirectory! Let’s step inside.

```
cd 14
# use 'dir' on Windows
ls -l
```

The output being:

```
-r-xr-xr-x 1 marco marco 29 Jan  6 09:51 2e5cf36d9f2047f24341883bd564b1d5
```

One file inside the folder. Starting with `2e5`…. Hold on a second. If you take the `14` from before and join it with the filename (`2e5cf36d9f2047f24341883bd564b1d5170370`), you end up with…*exactly*, your SHA1 hash (`142e5cf36d9f2047f24341883bd564b1d5170370`)!

In fact, that file *is* the snapshot of your `README.txt` file, at the time of your commit!

Before getting too excited and trying this out on your existing legacy projects, do note that Git does compress your files, so you cannot just open that file and look at it. You can, however, take a look at this Stackoverflow answer to find out how to uncompress your file.

Phew, that was quite a ride!

Git SHA Sum

To sum up what you just learned. When you commit a new file or update a file, i.e. create a new version of a file, Git will:

- …essentially run a `sha1hash(header + filecontent)` over your file's content. Read 'How is git commit sha1 formed' for the function that is executed.
- …store the full file (not a delta) in your git repository's objects folder, with the *hash* as the filename. More specifically: the first two characters of the hash as a subdirectory, and hash.substring(2, length) as the filename.
- Note, that Git compresses your files (deflate/zlib) and also, the bigger your repository grows, will actually merge files together and store them more efficiently, so your objects folder will look slightly different in more mature, legacy projects. Still, the same basics apply.

Oh, and yes, commits and the file trees referencing all the files for a specific commit are also…just text files. And stored and hashed the same way.

It's as simple as that.

Git Refresher: Commits, Branches & Heads

A good understanding of commits & branches seems so trivial, given that you're working with these concepts on a daily basis.

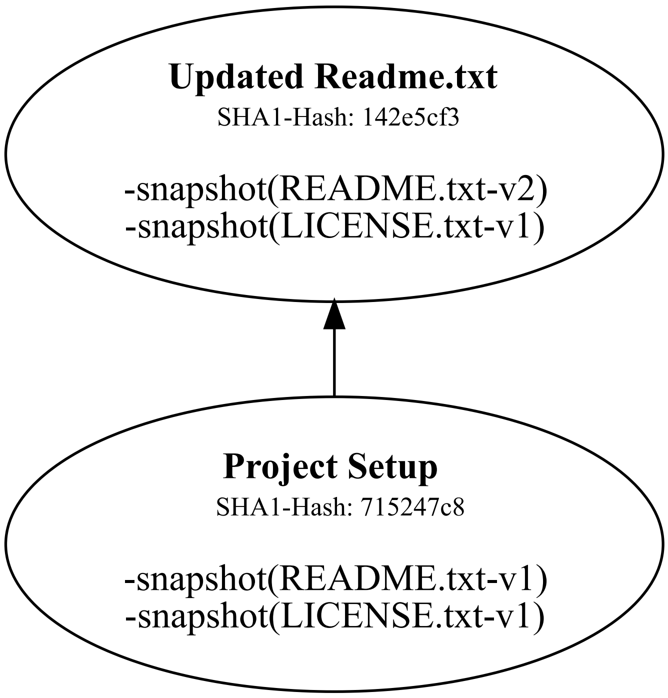
But, do you *really* know what `HEAD` refers to? Are you comfortable with `detached HEADS`?

Let's do a quick refresher.

What your repository looks like

Git repositories, or more specifically your commit history is modelled as a Directed Acyclic Graph.

Instead of getting lost in boring Computer Science theory, like edges, vertices and topological ordering, let's look at a diagram of your current commit history.



At the moment, your repository only consists of two commits. And as you learned in the previous section, every commit has references to *all* file snapshots, valid at the time of the commit.

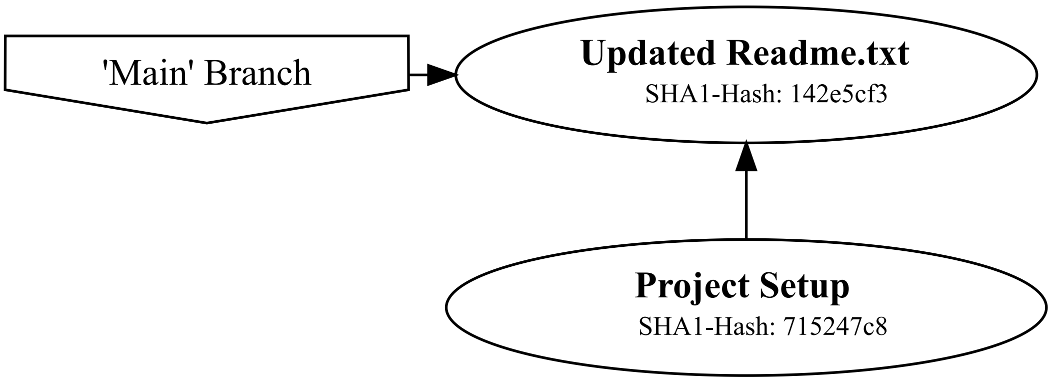
CS speak: I like to think of a Git commit history as flowing from bottom-to-top or left-to-right (directed), with each commit having one or more parent commits, but never forming a cycle (acyclic). The direction does of course not matter, it's simply how e.g. IntelliJ's Git tool window also represents the flow.

What is a branch?

A branch in Git is just a *pointer* to a commit.

Therefore, creating a branch means adding a pointer, effectively a label, to a specific commit. Conversely, deleting a branch means deleting the *pointer*.

And if you keep committing on a branch, the branch pointer will follow the *latest* commit (i.e. the so-called `tip`) like a shadow.



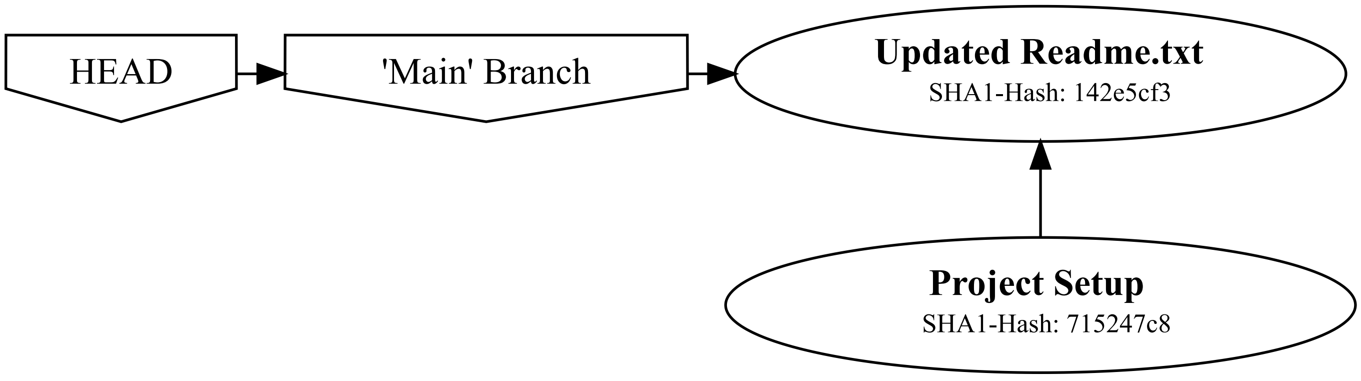
What is HEAD?

This one's almost a trick question, because even the correct answers on Stackoverflow contain subtle misconceptions.

The short answer is:

1. *HEAD* (all uppercase) is a pointer to what you have currently checked out and will *follow* along once you e.g. add new commits.
2. What you have currently checked out can be: a branch *or* a commit *or* a tag.
3. Typically, *HEAD* points to a branch.

Adding this piece of information, this is what our repository currently looks like:



If you don't trust diagrams, you can also open up your terminal and execute the following commands:

```
# inside your project folder
cat .git/HEAD # use 'type .git\HEAD' on Windows, note the backslash

refs/heads/main
```

As you can see, your *.git/HEAD* file points to your *main* branch (which in turn points to your second commit).

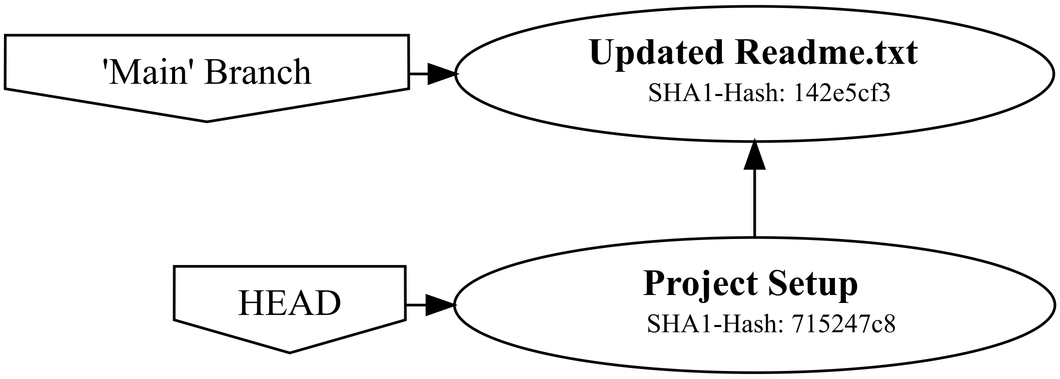
And if you keep commiting on that branch, the *HEAD* pointer will follow the *branch* pointer like a shadow.

What is a detached HEAD?

What happens, if you now checkout a revision directly?

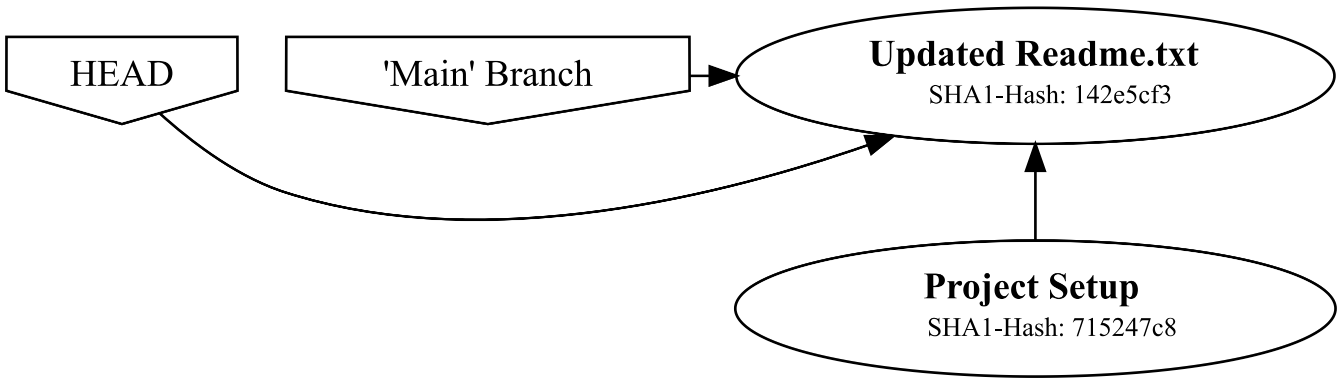
```
git checkout 715247c8426d3c16881539118e1eafeb38439b1c
cat .git/HEAD # use 'type .git\HEAD' on Windows, note the backslash

715247c8426d3c16881539118e1eafeb38439b1c
```

Exactly, now *HEAD* does not point to a branch, but it points directly to a commit!



In fact you can also check out the second commit (the one which has a branch pointer) directly. It has the same effect.

```
git checkout 142e5cf36d9f2047f24341883bd564b1d5170370
cat .git/HEAD # use 'type .git\HEAD' on Windows, note the backslash

142e5cf36d9f2047f24341883bd564b1d5170370
```



And the above two situations are exactly what a **detached HEAD** is. It doesn't mean that you, yourself are detached from HEAD, but that *HEAD* is **detached from a branch pointer**.

That's why, whenever you executed the two checkouts above, Git warned you that you are now in 'detached HEAD' state.

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
...
```

Why the warning? Because you can happily create new commits in this state, but when you switch branches, all your commits will be gone / garbage-collected. Unless, you create a new branch to retain your commits.

What are refs/heads?

This leaves us with the question of what is the difference between *HEAD* (uppercase) and *heads* (lowercase).

Well, a Git repository can have (among other things), branches and tags. The umbrella term for those is *references* or *refs*.

And whereas *HEAD* points to whatever you have checked out at the moment, a Git repository can contain many branches and each branch has a *head* (lowercase), the commit with the branch pointer. So, the full name for your 'main' branch would actually be *refs/heads/main*, for 'feature-branch-1' it would be *refs/heads/feature-branch-1* and so on.

In fact, go to your terminal and execute the following commands:

```
# inside your project folder
cd .git/refs
ls -ls # or dir on Windows

drwxrwxrwx 1 marco marco 512 Dec 28 10:56 heads
drwxrwxrwx 1 marco marco 512 Dec 28 09:54 tags
```

Ah, our references (branches, tags)! Let’s check out the heads folder.

```
cd heads
ls -ls # or dir on Windows

-rwxrwxrwx 1 marco marco 41 Dec 28 10:56 main
```

There’s our `main` branch! Go ahead and create a couple of other branches if you want, and you’ll see more files pop up in here (one file corresponding to a branch).

Git Merge: Behind the Scenes

Finally, let’s have a closer look at `merging`.

Scenario

Create and switch to a new branch called `merge_deep_dive`, in your repository. Then, assume you finally settled on a software license for your repository (GPL) and you’re going to update the `LICENSE.txt` file with it.

```
# create and checkout a new branch
git checkout -b merge_deep_dive

# update LICENSE.txt
echo "gpl-3.0" > LICENSE.txt
git add LICENSE.txt
git commit -m "Using a GNU General Public License v3.0 license"
```

```
[merge_deep_dive f05e046] Using a GNU General Public License v3.0 license
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now, a bit contrived, but switch back to *main* and update the same *LICENSE.txt* file with another license (Apache).

```
# checkout branch
git checkout main

# update LICENSE.txt
echo "apache-2.0" > LICENSE.txt
git add LICENSE.txt
git commit -m "Using a Apache license 2.0 license"

[main 02d5da3] Using a Apache license 2.0 license
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Of course, when you now attempt to merge the *merge_deep_dive* branch into your *main* branch, you'll get a conflict.

```
# merging
git merge merge_deep_dive

# conflict time!
Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Why did we get this conflict? We as humans know that we edited the same line in the same file that we are trying to merge, but how does Git know?

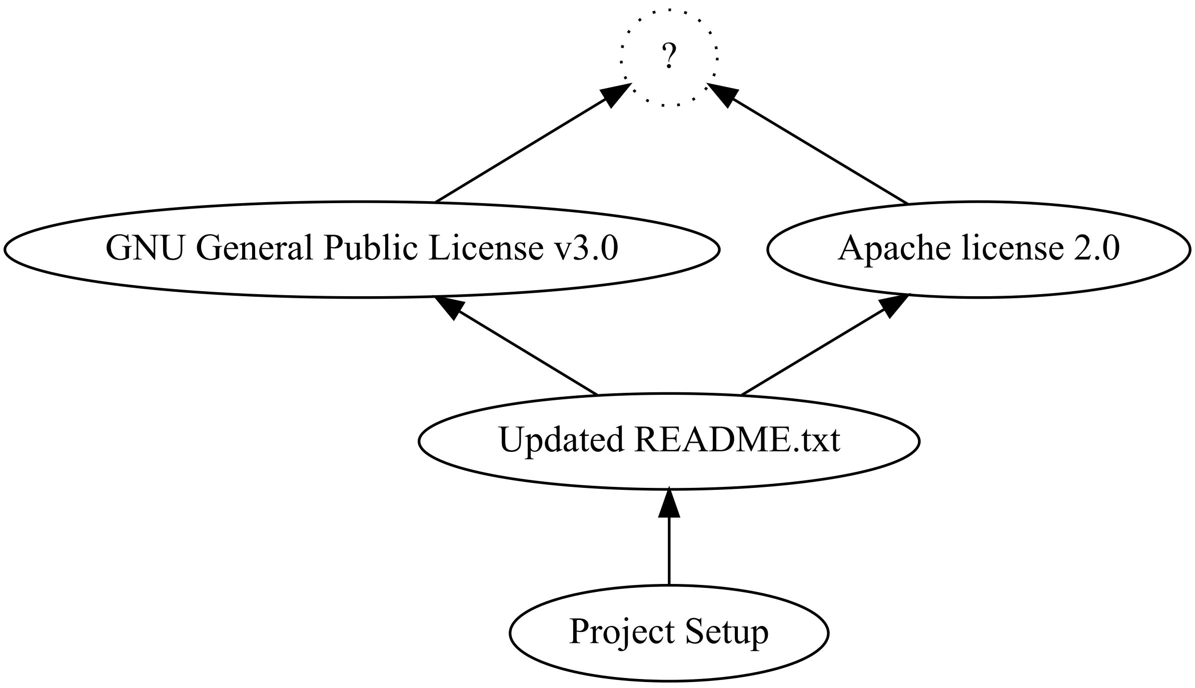
Essentially, Git will have to compare all the files, that you are trying to merge, from the two commits, line-by-line and see if there are any conflicting changes or not. (The comparison can be short-circuited, if the SHA1-hashes match).

That's a bit high-level, let's see in detail how this works.

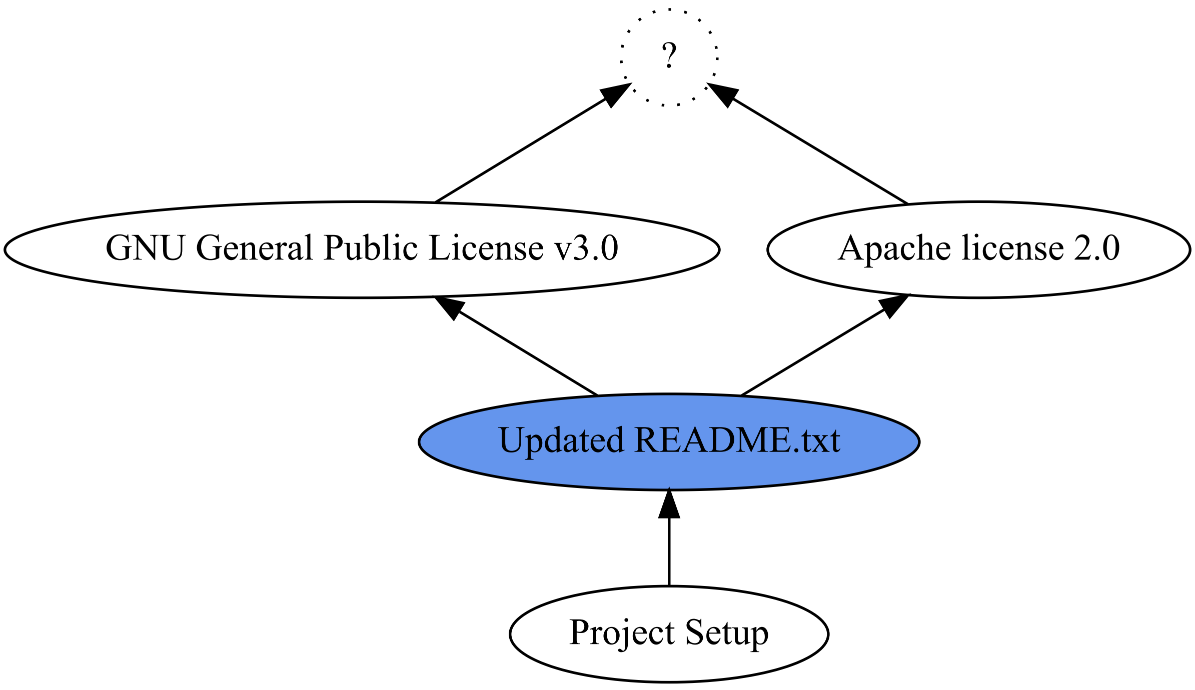
You're not restricted to just merging 2 commits, so in fact this could be n-commits, when e.g. merging n-branches.

How Git spots differences

This is your commit graph, at the time of the merge: You'll need to manually solve what the content of `License.txt` should be.



When you triggered the `git merge` command, Git will first of all try to find the so-called `merge base` - the (best) common ancestor for the two commits you're trying to merge.

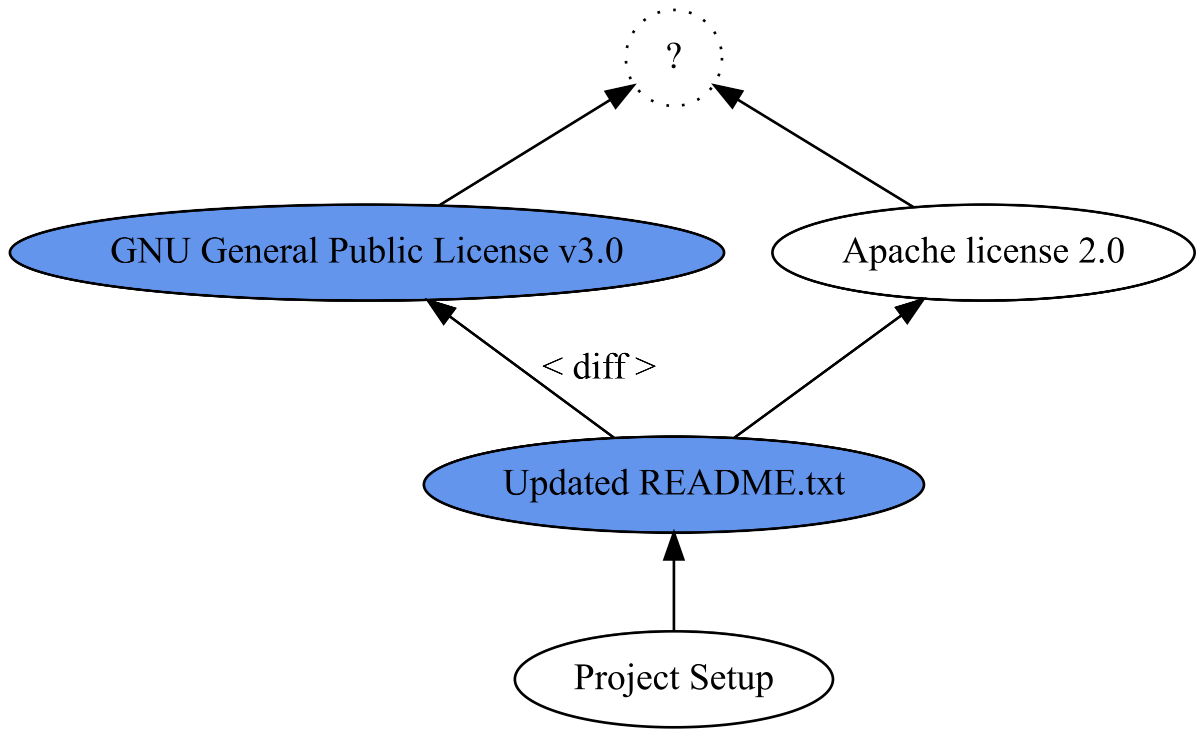


In our case, this would be the `Updated readme.txt` commit.

Then, Git will run two *diffs*, comparing all the files and lines of three commits - with potentially three different snapshots (base, branch1, branch2) of your repository's files.

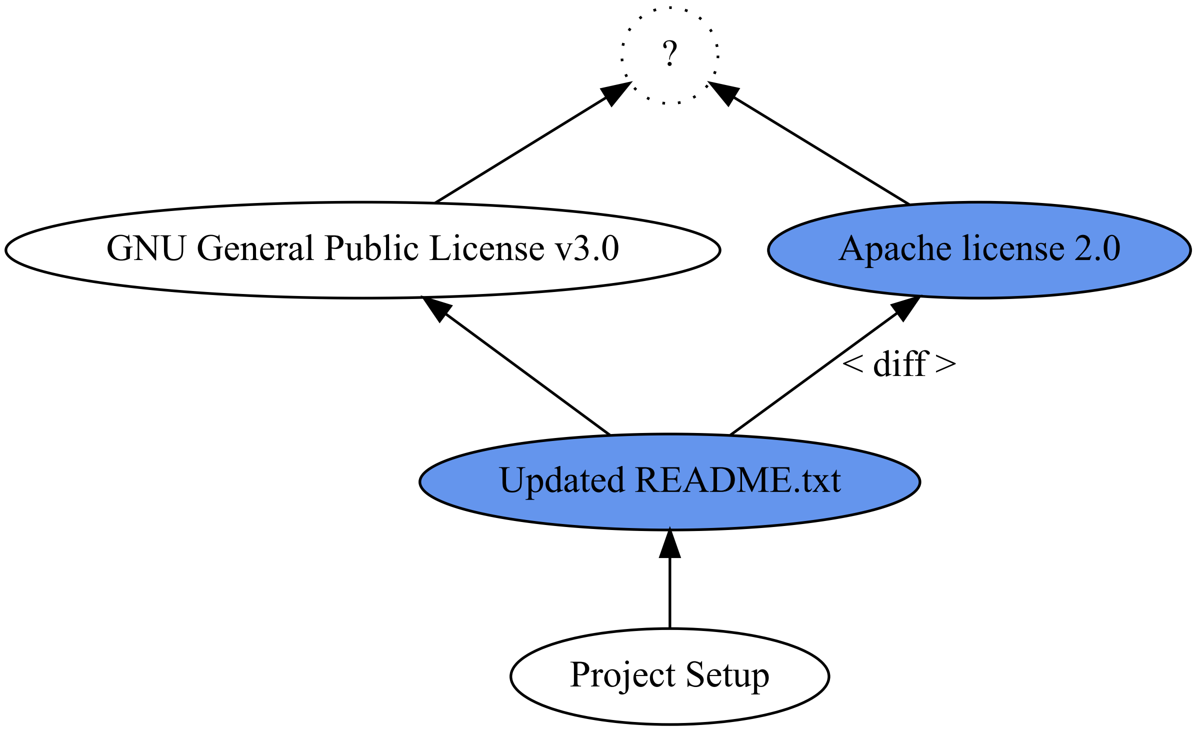
Diff 1: Merge Base vs First Merge Commit

Git tries to answer the following question: What changed between the `updated readme.txt` and `gnu general public license` commits?

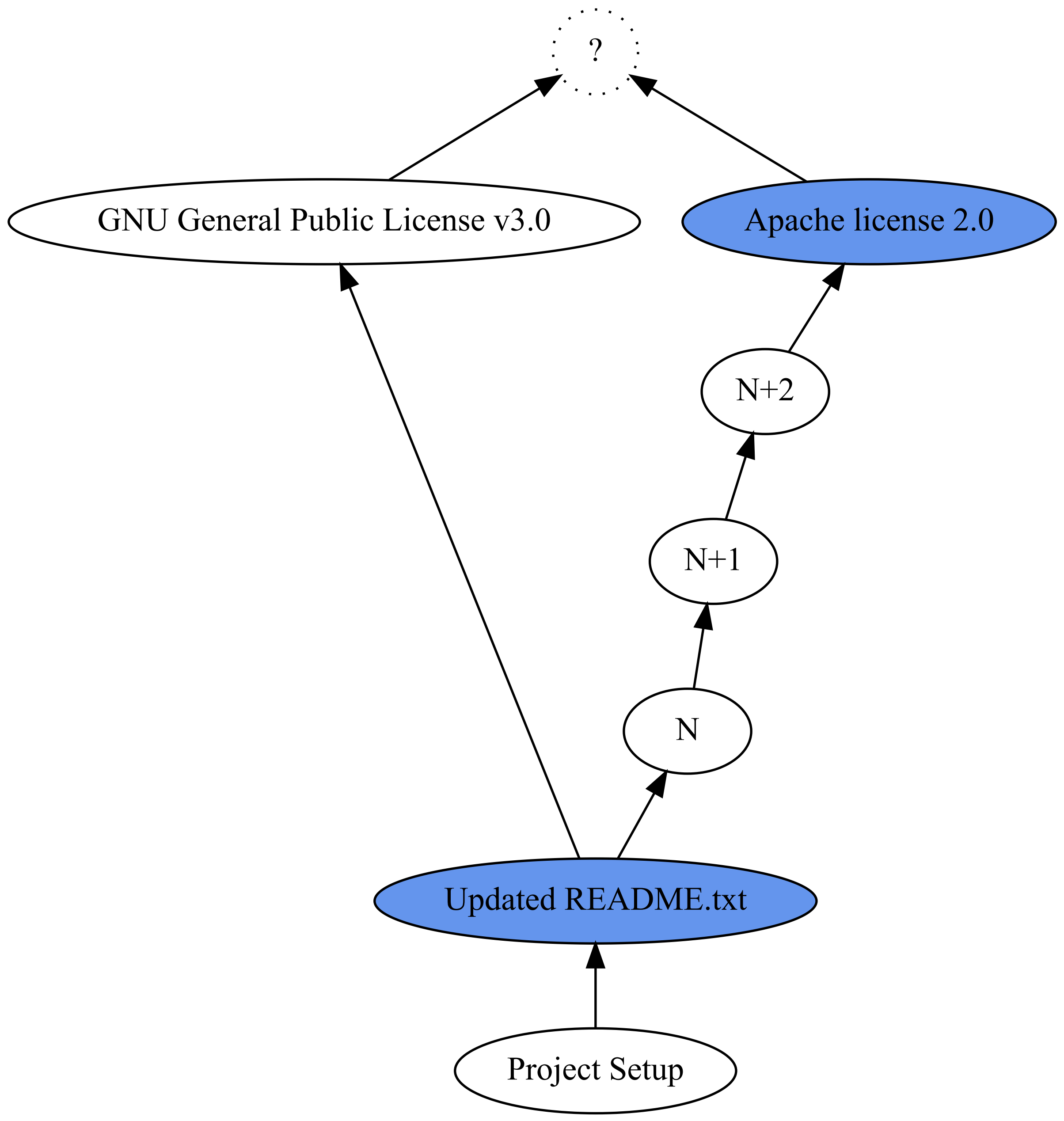


Diff 2: Merge Base vs Second Merge Commit

Then Git tries to answer the next question: What changed between the `updated readme.txt` and `apache license` commits?



Note that it doesn't matter how many commits are between the common ancestor and the commits you are trying to merge, only the files of those two will be diffed.



Git-Diff

For calculating the differences between two commits, there’s the handy `git diff` command, which will essentially compare all the files of two commits line-by-line and also handles cases like e.g. file renames.

Looking through our command line history above, quickly note down the (shorthand or long-form) commit ids and you’ll end up with something similar to this:

- **Merge Base (Updated Readme)**: 142e5cf36d9f2047f24341883bd564b1d5170370
- **Commit One (Apache License)**: 02d5da3168da45d812679eb2736b73bdc7e8b3cf

- **Commit Two (GNU public license)**: f05e046cc5e29cc76d7414d3d0a75f2483693b5d

Then, execute *git diff* twice.

```
# git diff --find-renames <hash-of-X> <hash-of-Y>

git diff 142e5cf36d9f2047f24341883bd564b1d5170370 02d5da3168da45d812679eb
```

You'll get the following output.

```
diff --git a/LICENSE.txt b/LICENSE.txt
index ddd3b7b..30d05dd 100644
--- a/LICENSE.txt
+++ b/LICENSE.txt
@@ -1 +1 @@
-"-TODO-"
+"apache-2.0"
```

And once more for the second diff.

```
# git diff --find-renames <hash-of-X> <hash-of-Y>

git diff 142e5cf36d9f2047f24341883bd564b1d5170370 f05e046cc5e29cc76d7414c
```

Which leads to the following output.

```
diff --git a/LICENSE.txt b/LICENSE.txt
index ddd3b7b..0da1e62 100644
--- a/LICENSE.txt
+++ b/LICENSE.txt
@@ -1 +1 @@
-"-TODO-"
+"gpl-3.0"
```

To fully understand the diff's output, you can read this great Stackoverflow Answer.

The gist is, that both commits changed the `LICENSE.txt` file on line number one. They removed the `-TODO-` line and each added their own, different line.

What Git will do with the diff result

Git's final job is to produce a merge commit, which means it will take your `merge base` and apply a list of changes to that base.

The problem: Git executed two diffs, so it now has two lists of changes and it needs to combine those changes into *one* final list. Here's how that works:

- If there are changes present in both diff outputs, then it needs to make sure to apply those changes only once. Removing the `-TODO-` line is such a candidate in our example, it cannot be removed twice.
- If the changes are present in only one diff and are non-conflicting, then just apply the change.
- But if there are conflicting changes present in both diffs (`+"apache-2.0"` vs `+"gpl-3.0"` in the same file on the same line, in our example), then Git cannot reasonably do anything and it needs to prompt the user (YOU!) for a manual resolve.

How does it do that?

Merge Markers

The way Git notifies you about merge conflicts in specific files is by putting *merge markers* into your conflict file(s): `<<<<<<<` and `>>>>>>>`, respectively.

Let's have another look at your `LICENSE.txt` file during this merge.

```
## or 'type LICENSE.txt' on Windows
cat LICENSE.txt
```

Which will result in output like this:

```
<<<<<<< HEAD
"apache-2.0"
=======
"gpl-3.0"
>>>>>>> merge_deep_dive
```

I have seen the odd colleague (including myself) panic when I saw the merge markers (`<<<<<<<` or `>>>>>>>`) in the past, but let's step through this version of the file line-by-line.

```
<<<<<<< HEAD
(...)
>>>>>>> merge_deep_dive
```

The whole block surrounded by the merge markers simply means that there is a merge conflict on a certain line (or surrounding lines, between different branches: `HEAD / (pointing to main)` and `merge_deep_dive`.

```
"apache-2.0"
=======
"gpl-3.0"
```

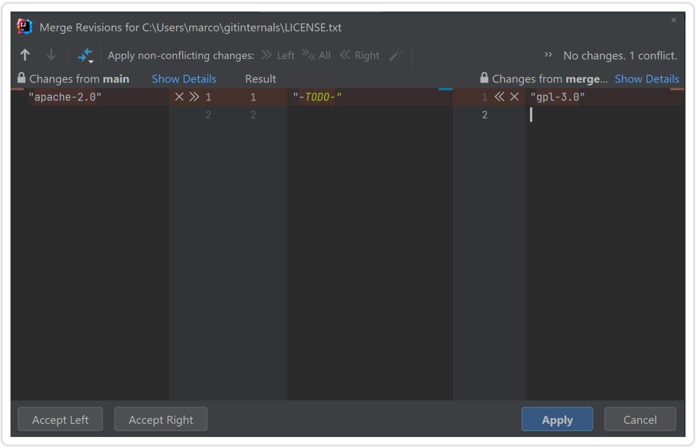
The two different versions of the line are split by `=======`.

And that's it. There's nothing special about the merge markers per-se, other than you, as a human, know that there is a conflict which you should fix and that IDEs can take those markers and convert them to something more visually appealing. What does that mean?

Well, e.g. IntelliJ's (or your favorite IDE's) real work is turning this:

```
<<<<<<< HEAD
"apache-2.0"
=======
"gpl-3.0"
>>>>>>> merge_deep_dive
```

Into a nice little dialogue window that lets you pick changes.


Merge Revisions for C:\Users\marco\gitinternals\LICENSE.txt
Apply non-conflicting changes: Left All Right
No changes. 1 conflict.
Changes from main Show Details
Result
Changes from merge... Show Details
"apache-2.0"
"-TODO-"
"gpl-3.0"
Accept Left
Accept Right
Apply
Cancel

Hence, whenever you accidentally close Intellij's merge window and see `>>>>>>>` merge markers, you shouldn't panic. *Nothing happened*.

Simply re-open the "resolve" window, let IntelliJ reparse the markers and visualize them for you, so you can click a couple buttons, instead of having to solve the merge manually, on the command line.

That's right, we haven't even talked about solving the conflict yet. Let's do that now.

If you have been wondering about what 'git merge --abort does', the answer (simplified): Remove all the merge markers and revert back your files to the state of your latest commit.

Solving a Merge

To *solve* the merge, you need to replace the marker block (`<<<<<< >>>>>>>`) with whatever you want the line to read, after the merge.

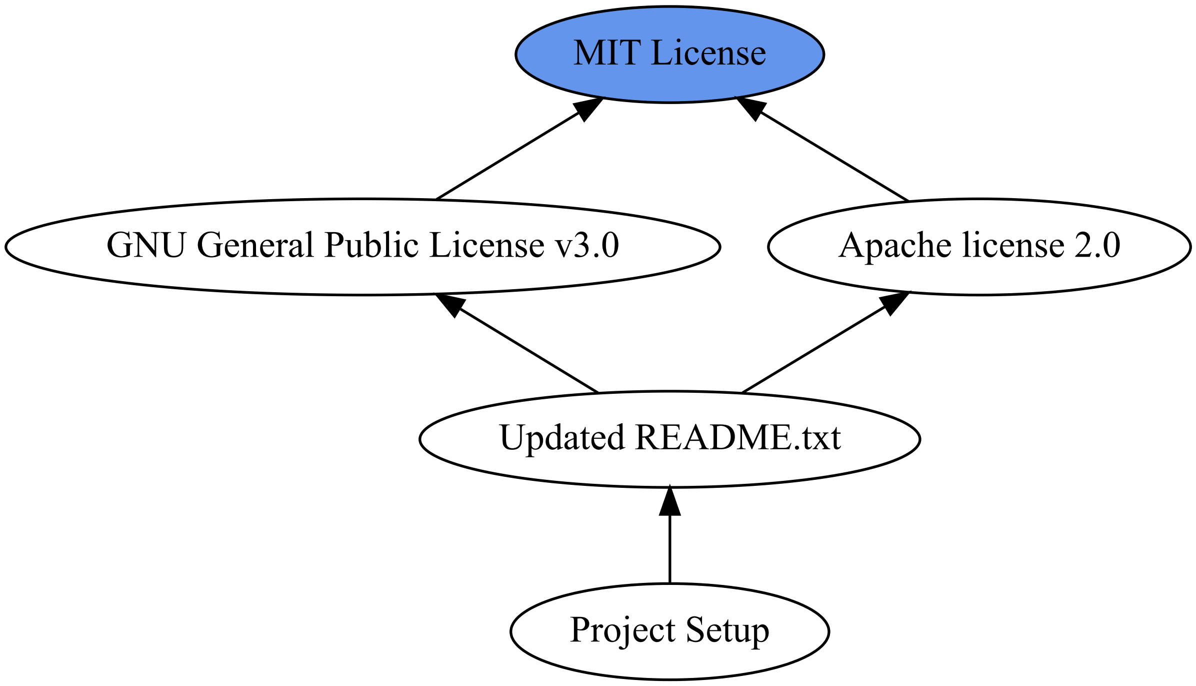
As a side-note (and I've seen that happen in the real-world™): Merge markers aren't special. They are just a couple of characters. Which means you *could* also commit your file as is, i.e. with the merge markers inside - though nonsensical, Git won't complain. (And your colleagues will love that.)

As your `LICENSE.txt` file only consists of one line, you'll simply override the complete file here with a completely new text, thus 'solving' the merge conflict.

Remember, you don't *have* to take the changes from a previous commit, you can also choose to ignore them. And in any case, Git will simply store the complete new version of the file in your repository.

```
echo "MIT No Attribution License" > LICENSE.txt
git add LICENSE.txt
git commit -m "solved the license challenge merge"

[main 2a7be29] solved the license challenge merge
```

This will change your repository’s DAG to look like this, introducing the merge commit `MIT license`.



Again, do note that the merge commit (`MIT license`) itself is not special, other than it having 2 ancestors or 'parents'. If you chose to merge 5 branches, you’d have 5 'parents' - yes, with Git you can merge as many branches as you’d like at the same time.

Summary

And that’s Git’s merge magic! Summed up:

- Find a merge base (common ancestor).
- Compare all the files of 3 (or more) commits line-by-line (if they have different SHA1 sums, that is).
- Produce a final list of changes and apply that to your merge base, to create a new commit.

Not too tricky, after all.

Here’s what’s interesting: `git merge` is used for cherry-picks and rebases as well, so we can straight away re-use our newfound knowledge in the next sections.

Git Cherry-Pick: Behind the Scenes

Cherry-Picks are interesting, because they *feel* like you can take a random commit and then somehow *clone* that on top of your current commit.

But this isn’t quite how cherry-picks work. In fact, if you *haven’t* yet read Git Merge: Behind the Scenes , do that first.

Why?

Because a `git cherry-pick` is essentially a `git merge`. Baffled? So was I.

Scenario

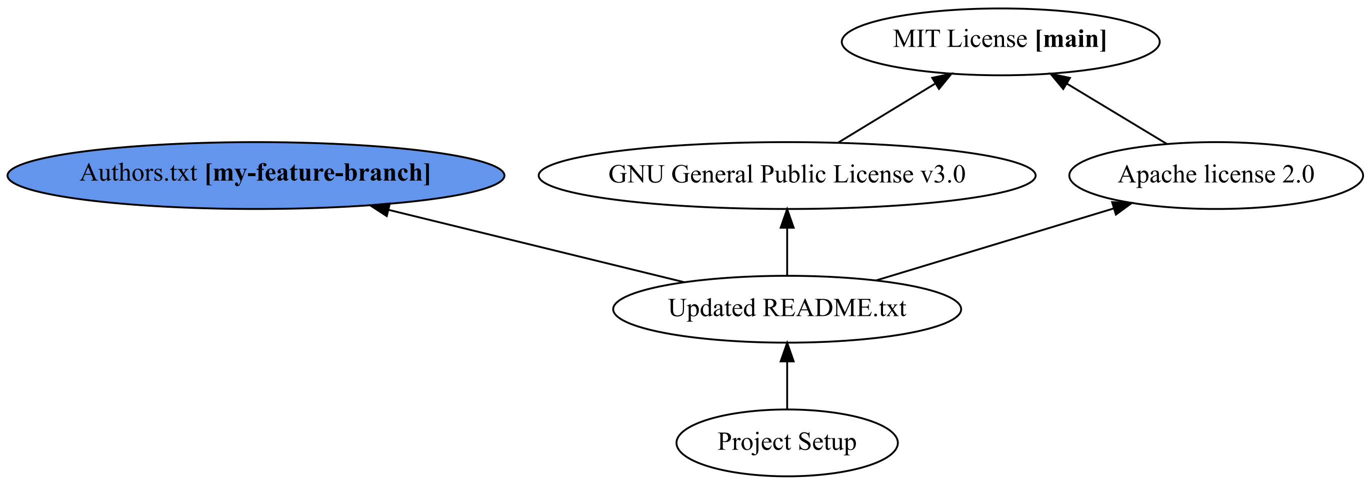
Go back to your terminal window and checkout the `updated readme.txt revision`. What will you get? Right, a **detached HEAD**.

Add a new file `AUTHORS.txt` and change `LICENSE.txt` 's content yet again.

```
# switching into detached head state
git checkout 142e5cf36d9f2047f24341883bd564b1d5170370

# adding a new commit
echo "Marco" > AUTHORS.txt
git add AUTHORS.txt

echo "proprietary" > LICENSE.txt
git add LICENSE.txt

git commit -m "added authors.txt and updated license"

[detached HEAD 6db2a07] added authors.txt and updated license
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 AUTHORS.txt

# creates a new branch to retain the commits you just created
git switch -c my-feature-branch
```

In addition to the files, you also created a new branch called `my-feature-branch`, to get out of your detached HEAD state.

Your repository now looks like this:



Switch back to your `main` branch, and cherry-pick your commit.

```
git checkout main

git cherry-pick 6db2a07

Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
error: could not apply 6db2a07... added authors.txt and updated license
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

Aha, so we're getting a merge conflict (because of our LICENSE.txt file), so we're not just blindly copying the changes from the cherry-picked commit.

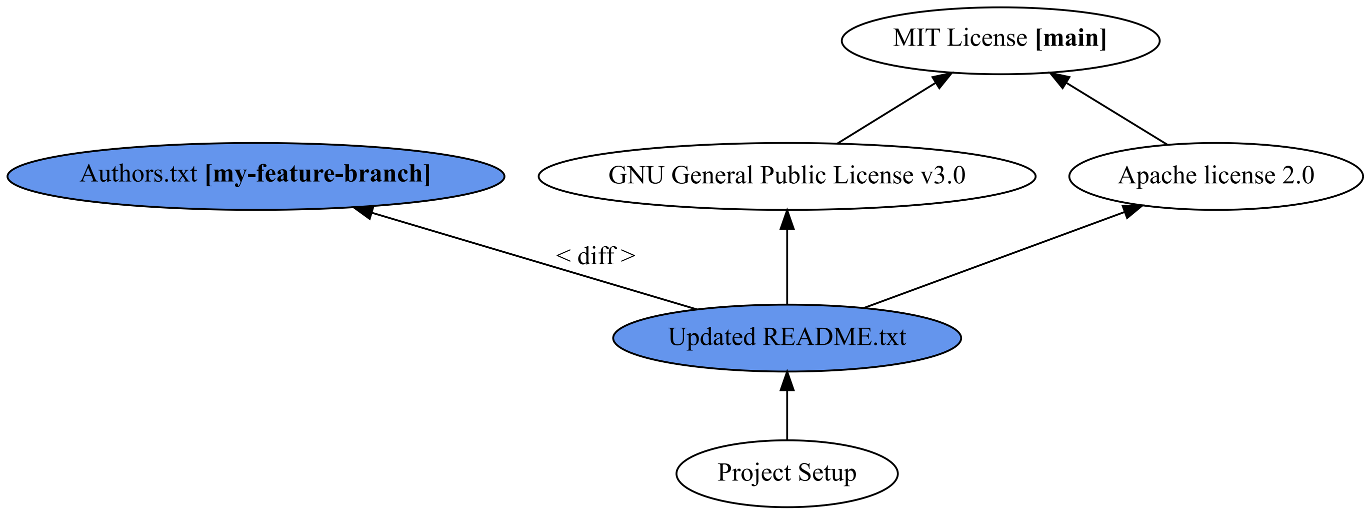
Thinking back one chapter, we know that we need (at least) two commits for a merge, but we also need a *merge base*, a common ancestor.

We've got the cherry-picked commit and our current commit, but what could be a sensible merge base?

Answer: It's the cherry-picked's parent!

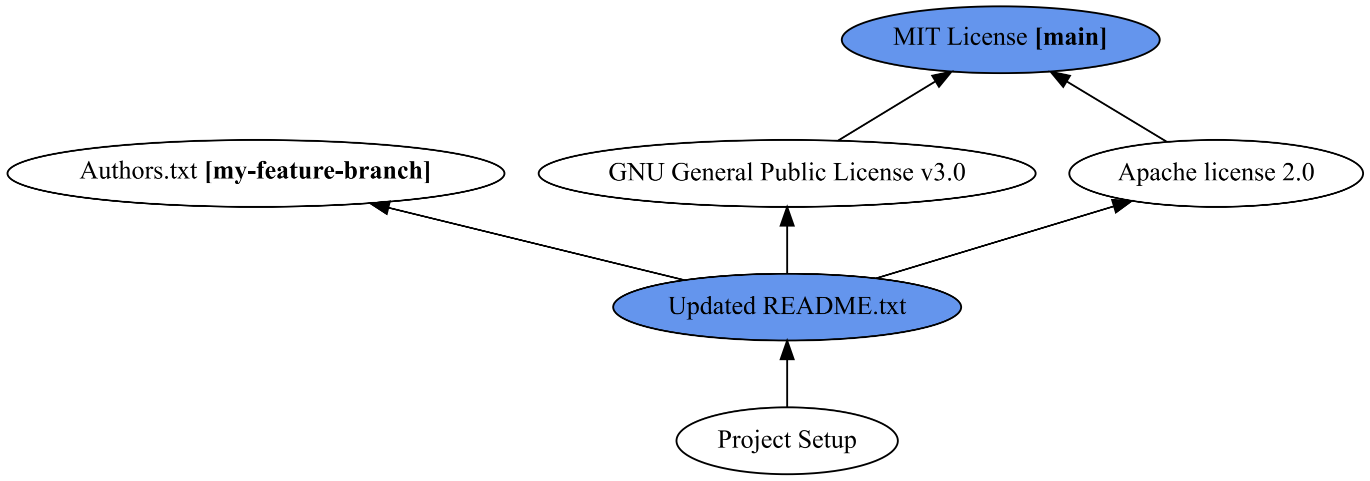
Confused? Here's the two git-diffs that will be executed.

Diff 1: `updated readme.txt` vs `Authors.txt`



This one's kind of simple. We're getting a diff list, where someone added an `authors.txt` file and also modified the `LICENSE.txt` file.

Diff 2: `updated readme.txt` vs `MIT license`



This one's a bit confusing. A diff between the merge base and our current commit, will, essentially give us all the changes to go from the merge base to... .our current commit.

Git's Cherry-Pick Algorithm

Take some time to think this through. Here's what happens:

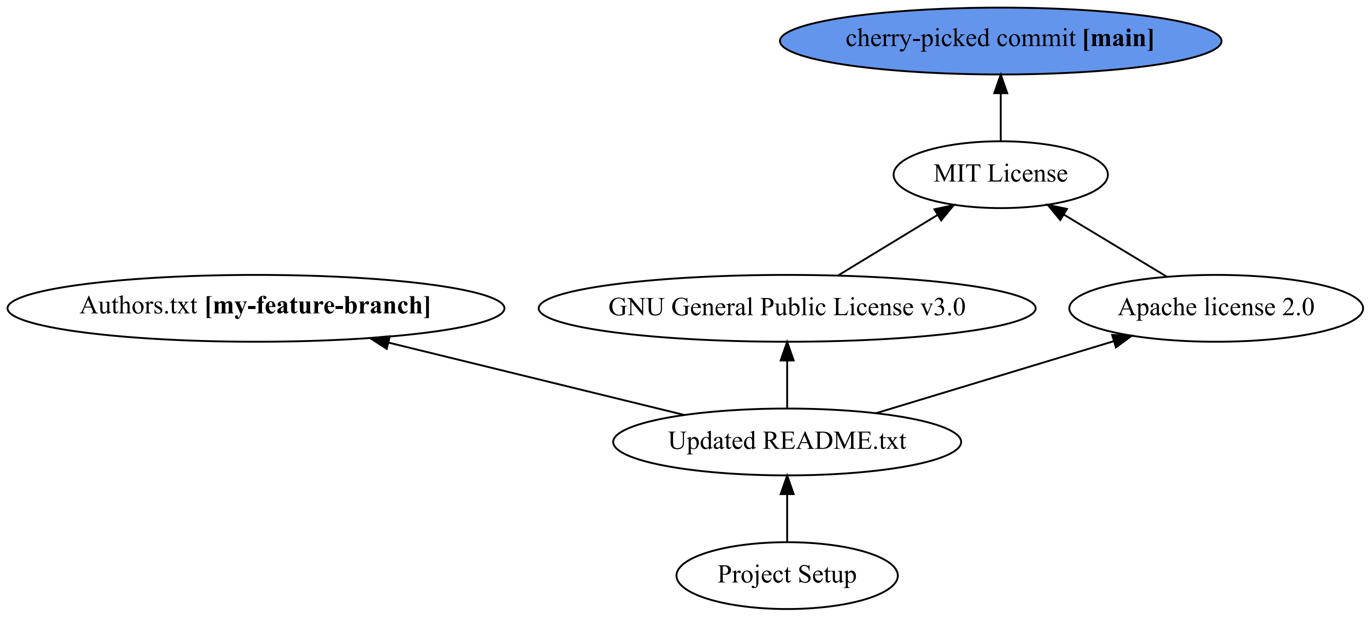
1. A diff between the cherry-picked's parent and the cherry-picked commit, will give you, well, the changes you are trying to cherry-pick.
2. A diff between the cherry-picked's parent and your current commit, will give you all the changes that Git needs to apply, to go from the parent, to your current state.

3. And if you have these two diff lists, then you can simply re-use git's merge functionality: First apply all the changes from (2). Then apply the changes you are actually interested in (1). That way you'll automatically get merge conflicts for files which are conflicting (deleted,changed,modified) in your cherry-picked commit.
4. Instead of producing a merge commit (combining multiple ancestors), Git will just create a normal commit from the combined diffs.

So, let's quickly fix the conflict (if you want, have a look at the merge markers inside `LICENSE.txt` to see it's just a normal merge process).

```
echo "Proprietary Wins" > LICENSE.txt
git add LICENSE.txt
git commit -m "cherry-picked commit"

[main 3081860] cherry-picked commit
 Date: Thu Jan 6 10:54:16 2022 +0100
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 AUTHORS.txt
```

The merge conflict is solved, and now you'll see a new commit (not a merge commit between multiple ancestors, or a clone or something else) added to your commit history.


cherry-picked commit [main]
MIT License
Authors.txt [my-feature-branch]
GNU General Public License v3.0
Apache license 2.0
Updated README.txt
Project Setup

And that, in a nutshell, is a cherry-pick:

- A merge with a forced merge base, set to the cherry-picked's parent.

- Instead of a merge commit, combining multiple ancestors, you'll create a new commit in your current branch.

> Have you ever tried to cherry-pick a merge commit? What would happen in that case? What would the merge base be? Try it out on the command line!

Git Rebase: Behind the Scenes

Imagine you have commits `L-M-N` that you want to rebase onto commit `A`.

Some time ago, I thought that it would mean taking the `L-M-N` commits as a unit and then somehow *moving* them onto `A` as they are.

But as you might have guessed by now, that's not quite how rebasing works. In fact, if you haven't yet read the Git Cherry-Pick: Behind the Scenes section, do that now.

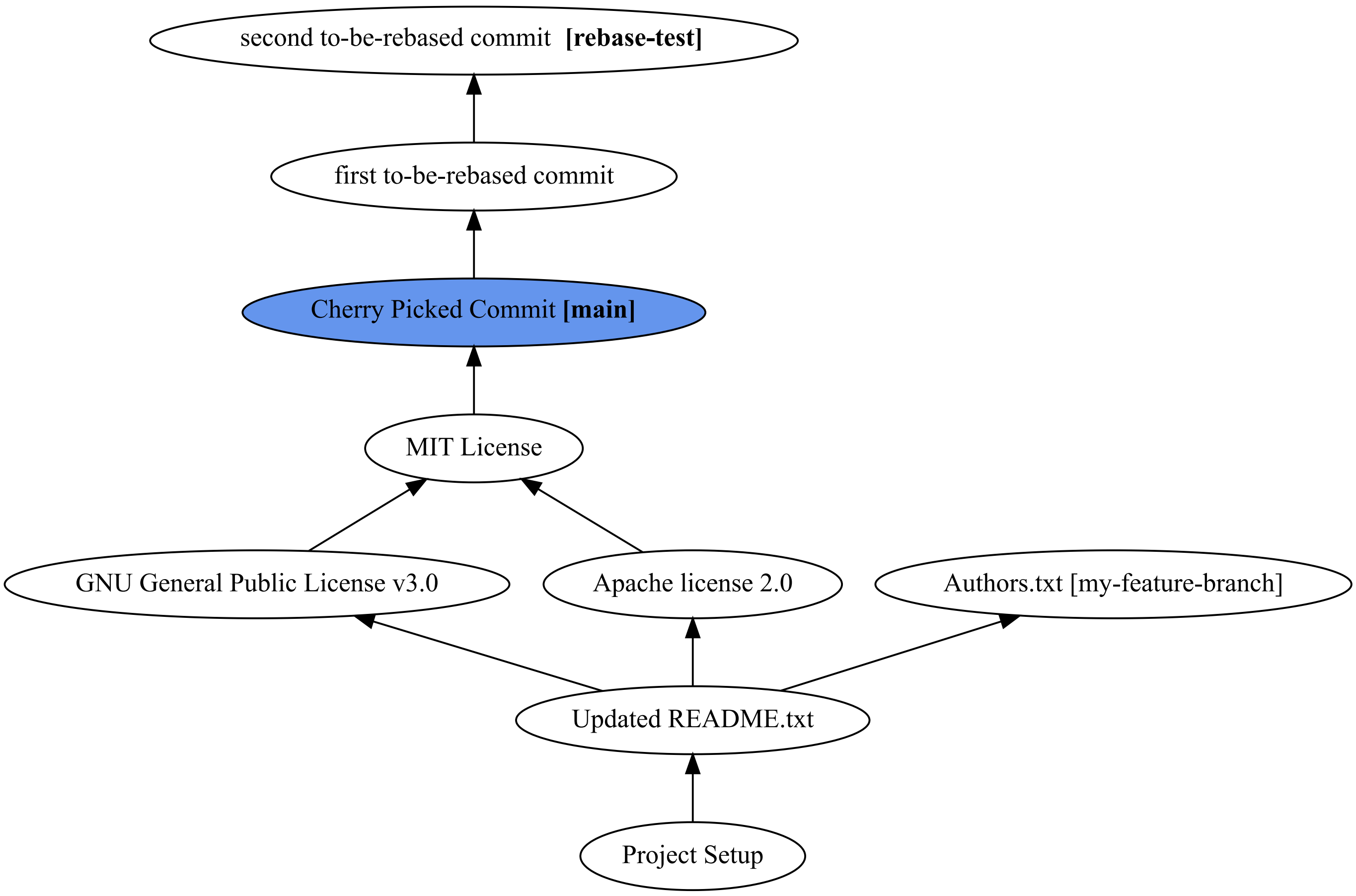
Why? Because, rebasing actually does cherry-picks under the hood! Let's find out how.

Scenario

```
# make sure you are on main branch and fork off a branch
git checkout main
git checkout -b rebase-test

## update LICENSE.txt
echo "first to-be-rebased commit" > LICENSE.txt
git add LICENSE.txt
git commit -m "first to-be-rebased commit"

[rebase-test afee899] first to-be-rebased commit
 1 file changed, 1 insertion(+), 1 deletion(-)

## update LICENSE.txt again
echo "second to-be-rebased commit" > LICENSE.txt
git add LICENSE.txt
git commit -m "second to-be-rebased commit"
```

```
[rebase-test 03da5d3] second to-be-rebased commit
 1 file changed, 1 insertion(+), 1 deletion(-)
```

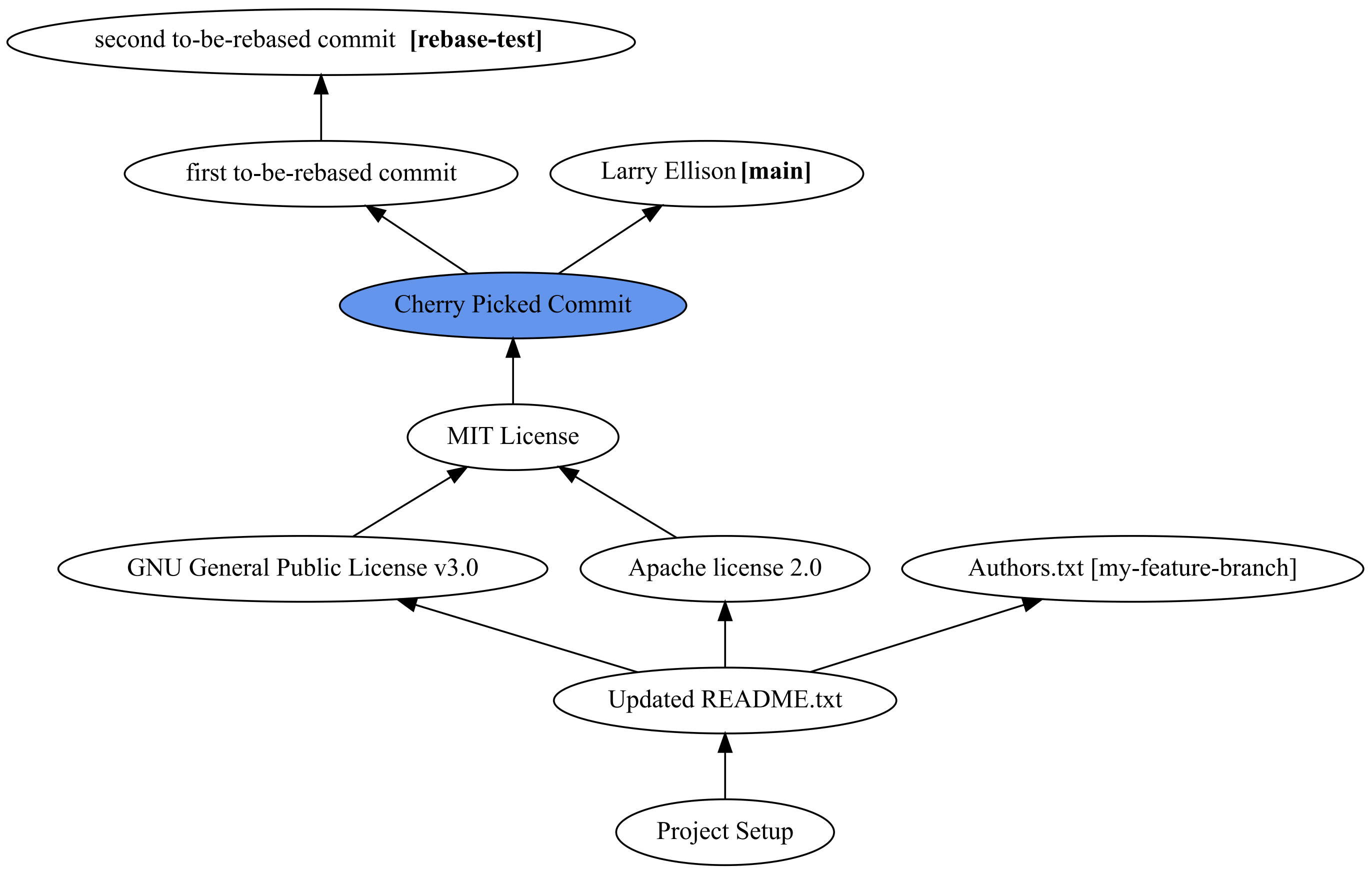
Forking from `main` branch, you created a new branch called `rebase-test`. The branch contains two commits, both of them updating the `LICENSE.txt` file (with non-sensical data).


second to-be-rebased commit **[rebase-test]**
first to-be-rebased commit
Cherry Picked Commit **[main]**
MIT License
GNU General Public License v3.0
Apache license 2.0
Authors.txt [my-feature-branch]
Updated README.txt
Project Setup

Now that we have that branch, let's go back to *main* and update our `LICENSE.txt` in that branch, yet again.

```
# make sure to checkout the main branch again
git checkout main

# update license.txt
echo "Larry Ellison license" > LICENSE.txt
git add LICENSE.txt
git commit -m "Larry Ellison license"

[rebase-test ac6ed98] Larry Ellison license
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Which will lead to this graph (Click to enlarge).


second to-be-rebased commit [rebase-test]
first to-be-rebased commit
Larry Ellison [main]
Cherry Picked Commit
MIT License
GNU General Public License v3.0
Apache license 2.0
Authors.txt [my-feature-branch]
Updated README.txt
Project Setup

Before you now try to rebase the `rebase-test` branch on top of `main`: What do you think is going to happen?

Will you get any merge conflicts? If so, how many and why?

```
# make sure you are on rebase-test
git checkout rebase-test

# rebase the branch onto main
git rebase main

error: could not apply e03d753... added rebase-test branch
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase -
Could not apply e03d753... added rebase-test branch
Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
```

Ok, so there seems to be one merge-conflict for now. Let's have a look at what it is by displaying the `LICENSE.txt` file.

```
cat LICENSE.txt # type LICENSE.txt on Windows

<<<<<<< HEAD
"Larry Ellison license"
=======
"first to-be-rebased commit"
>>>>>>> ea02446 (first to-be-rebased commit)
```

So, this is a conflict with the *first* commit that we made on the `rebase-test` branch. Let's fix the merge conflict by using both lines.

```
(echo Larry Ellison license && echo first to-be-rebased commit) > LICENSE
git add LICENSE.txt
```

What is going to happen when we continue our rebase? Let's find out.

```
git rebase --continue
```

You will first be prompted to enter a commit message for a (seemingly) new commit. Choose a message of your liking, save the commit and then you'll be brought back right to your terminal window, with yet another merge conflict waiting for you to be solved.

```
[detached HEAD fee1098] first to-be-rebased commit
 1 file changed, 2 insertions(+), 1 deletion(-)
error: could not apply 49922e8... second to-be-rebased commit
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase -
Could not apply 49922e8... second to-be-rebased commit
Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
```

What's wrong this time?

```
cat LICENSE.txt # type LICENSE.txt on Windows

<<<<<<< HEAD
Larry Ellison license
first to-be-rebased commit
=======
"second to-be-rebased commit"
>>>>>>> 49922e8 (second to-be-rebased commit)
```

Alright. It seems that Git now tried to rebase the `second-to-be-rebased commit`, leading to yet another merge conflict.

Let's fix it by putting all the three lines in our `LICENSE.txt` file and then continue with (or rather finish) the rebase.

```
(echo Larry Ellison license && echo first to-be-rebased commit && echo ec
git add LICENSE.txt

git rebase --continue
```

You'll be prompted once more to think of a commit message. Do that, save the new commit, and you'll get the following output:

```
[detached HEAD f61d31f] second to-be-rebased commit
 1 file changed, 2 insertions(+), 1 deletion(-)
Successfully rebased and updated refs/heads/rebase-test.
```

Hurray, the rebase was successful! But actually, that process was quite confusing, wasn't it?

What actually just *happened*? Let's find out!

How Rebases Work

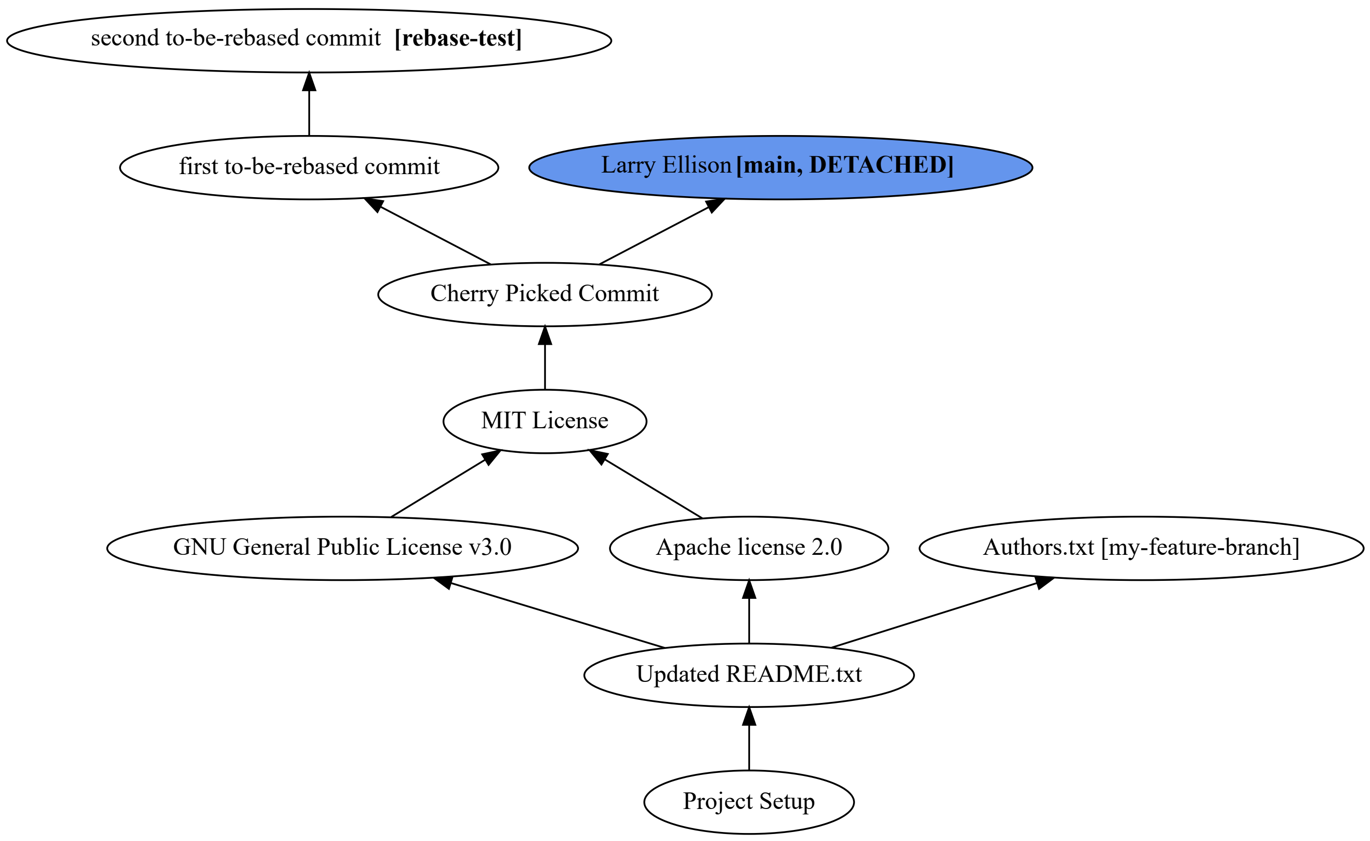
First of all, rebase collects a list of commit hashes to rebase, in our case the one from `first-commit-to-be-rebased` and `second-commit-to-be-rebased`.

In my case those ids would be:

- **First To Be Rebased Commit:** afee899...

- **Second To Be Rebased Commit:** 03da5d3d…

Then, it checks out our target, the `Larry Ellison` commit as a *detached HEAD*.

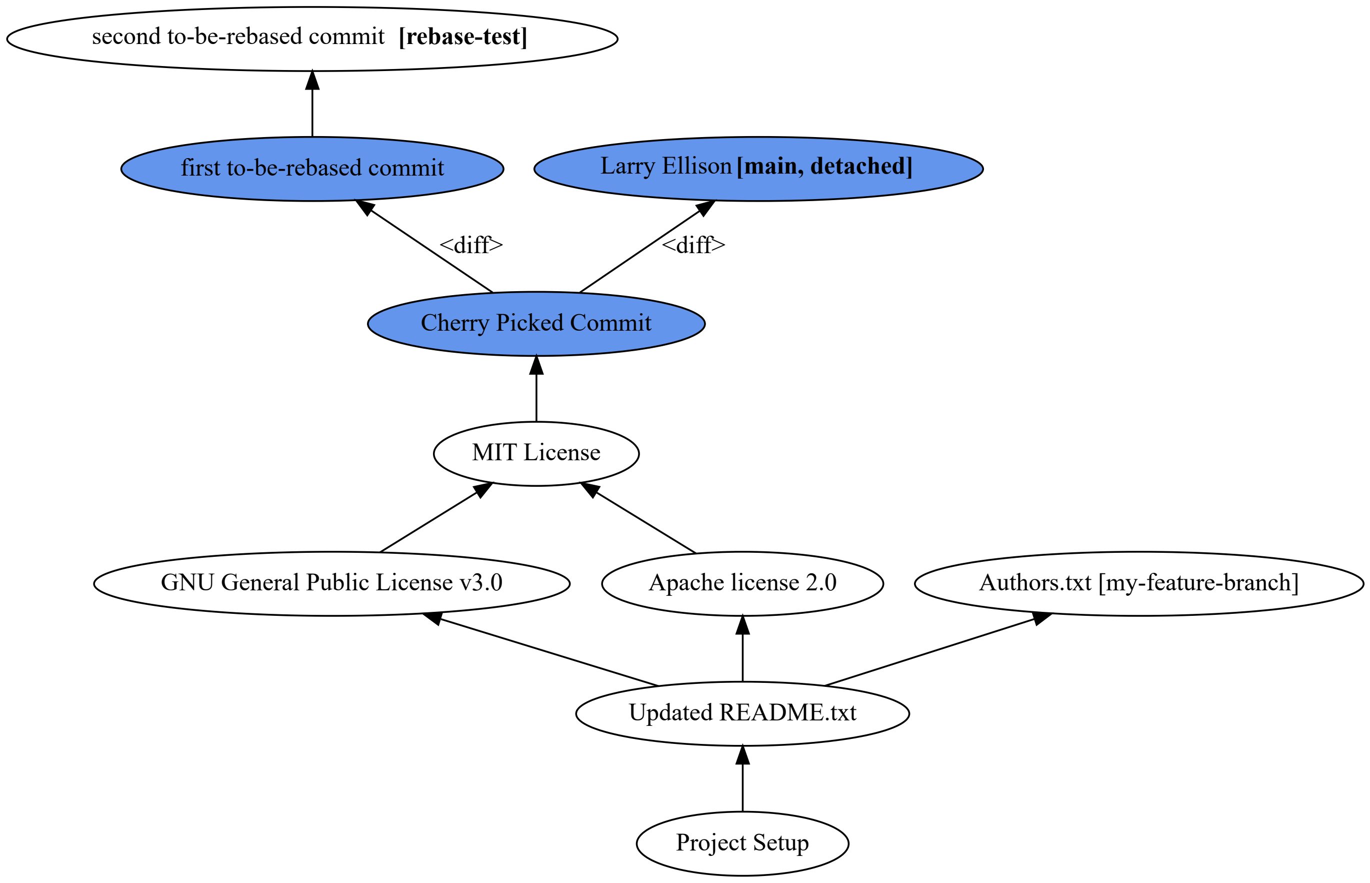

second to-be-rebased commit [rebase-test]
first to-be-rebased commit
Larry Ellison [main, DETACHED]
Cherry Picked Commit
MIT License
GNU General Public License v3.0
Apache license 2.0
Authors.txt [my-feature-branch]
Updated README.txt
Project Setup

Git will now cherry-pick each commit from that list (afee899, 03da5d3d), one at a time, in the appropriate order onto that *detached HEAD*.

First Rebase

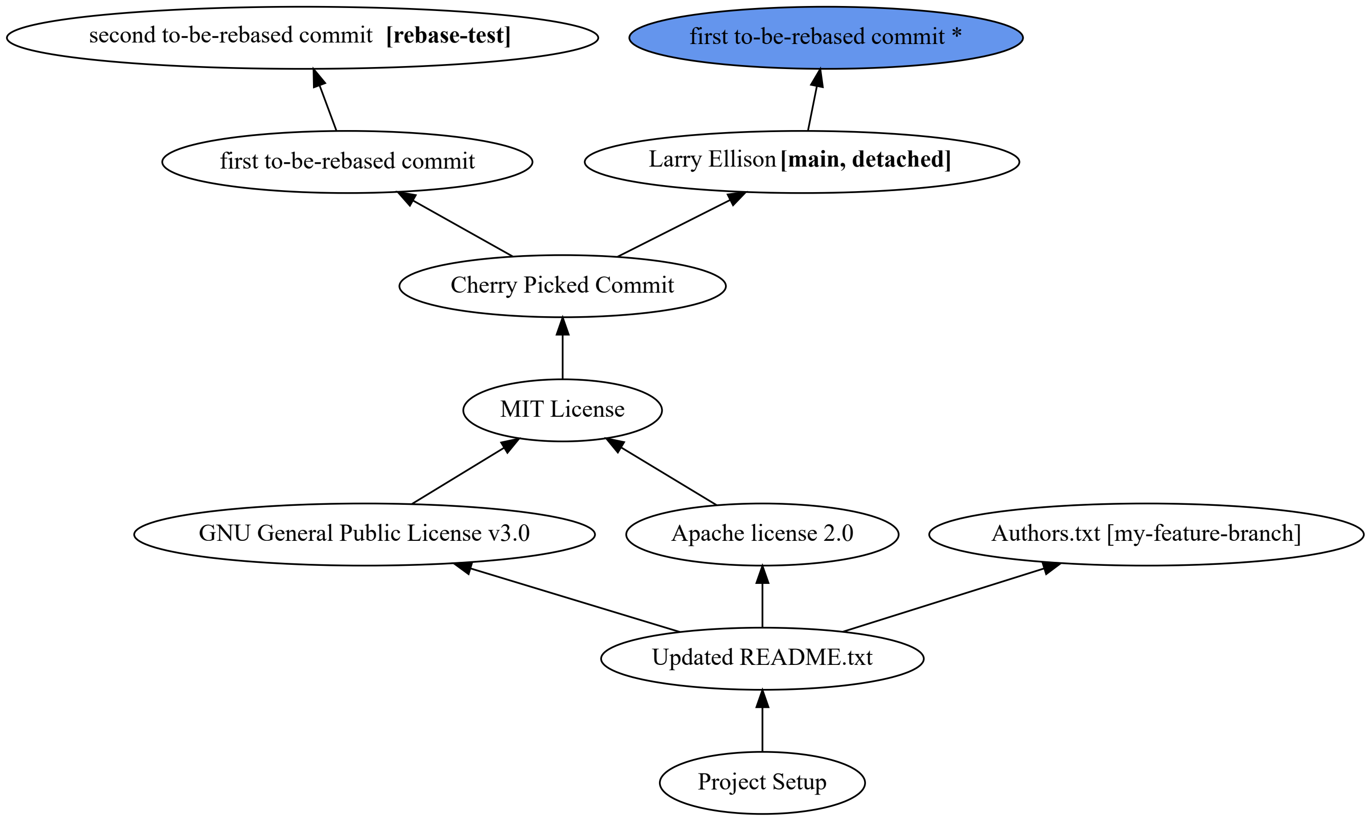
Let’s start with `first-commit-to-be-rebased`. Remember, a git cherry-pick is just a merge with a specific merge base. So, we need to find a merge base and then execute two diffs.

- **Merge Base:** `Cherry Pick Commit`. It’s the parent of the to-be-picked commit!
- **Diff 1 Candidate:** `first to-be-rebased-commit`
- **Diff 2 Candidate:** `Larry Ellison`, the detached HEAD.


second to-be-rebased commit **[rebase-test]**
first to-be-rebased commit
Larry Ellison **[main, detached]**
<diff>
<diff>
Cherry Picked Commit
MIT License
GNU General Public License v3.0
Apache license 2.0
Authors.txt [my-feature-branch]
Updated README.txt
Project Setup

And as both commits changed the same line, in the same `LICENSE.txt` file, you get a merge conflict, which you'll need to solve. But even without a merge conflict, you'll eventually always end up with a *new* commit (you're not just moving or copying the cherry-picked commit).

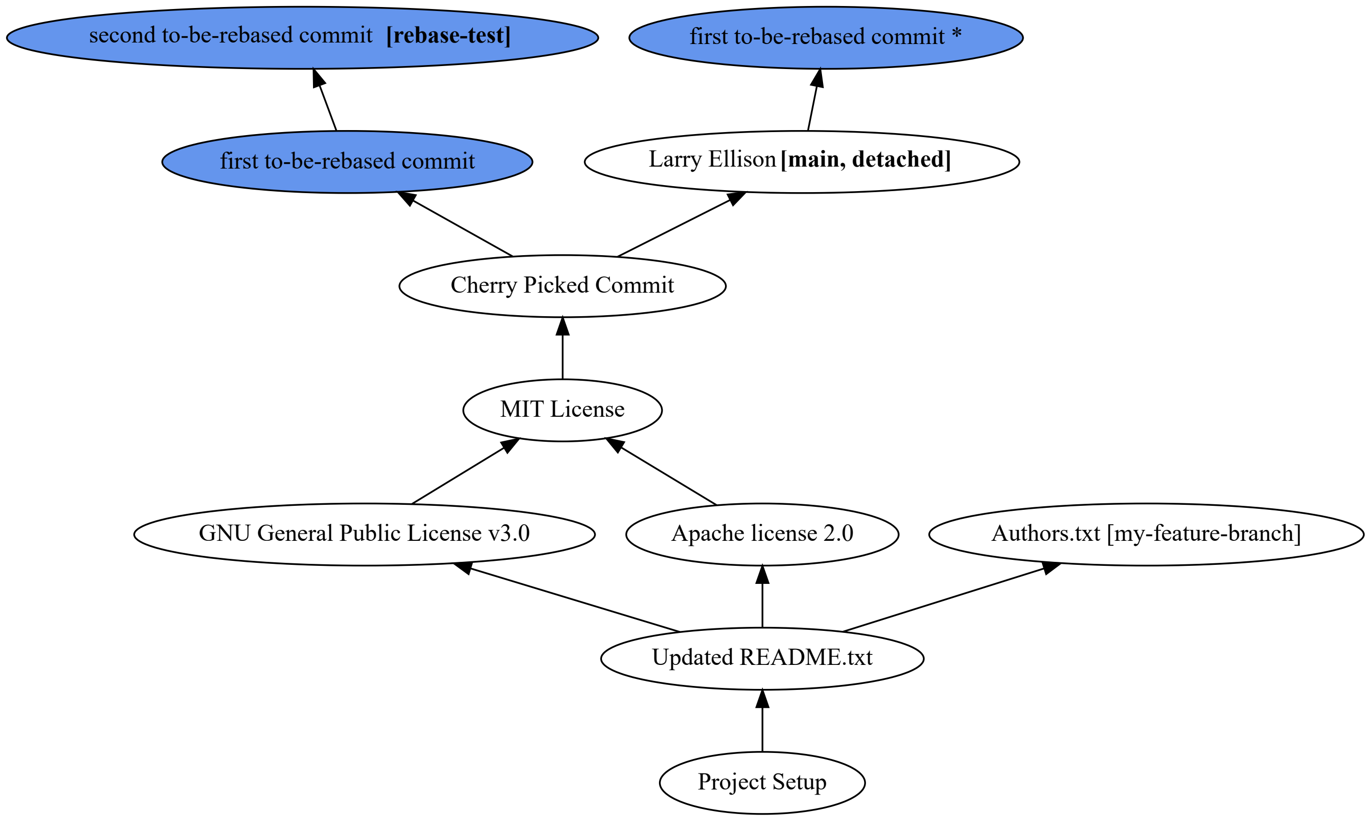
Hence, after the first cherry-pick, our graph will look like this. The '*' signals that the commit has been cherry-picked.


second to-be-rebased commit [rebase-test]
first to-be-rebased commit *
first to-be-rebased commit
Larry Ellison [main, detached]
Cherry Picked Commit
MIT License
GNU General Public License v3.0
Apache license 2.0
Authors.txt [my-feature-branch]
Updated README.txt
Project Setup

Second Rebase

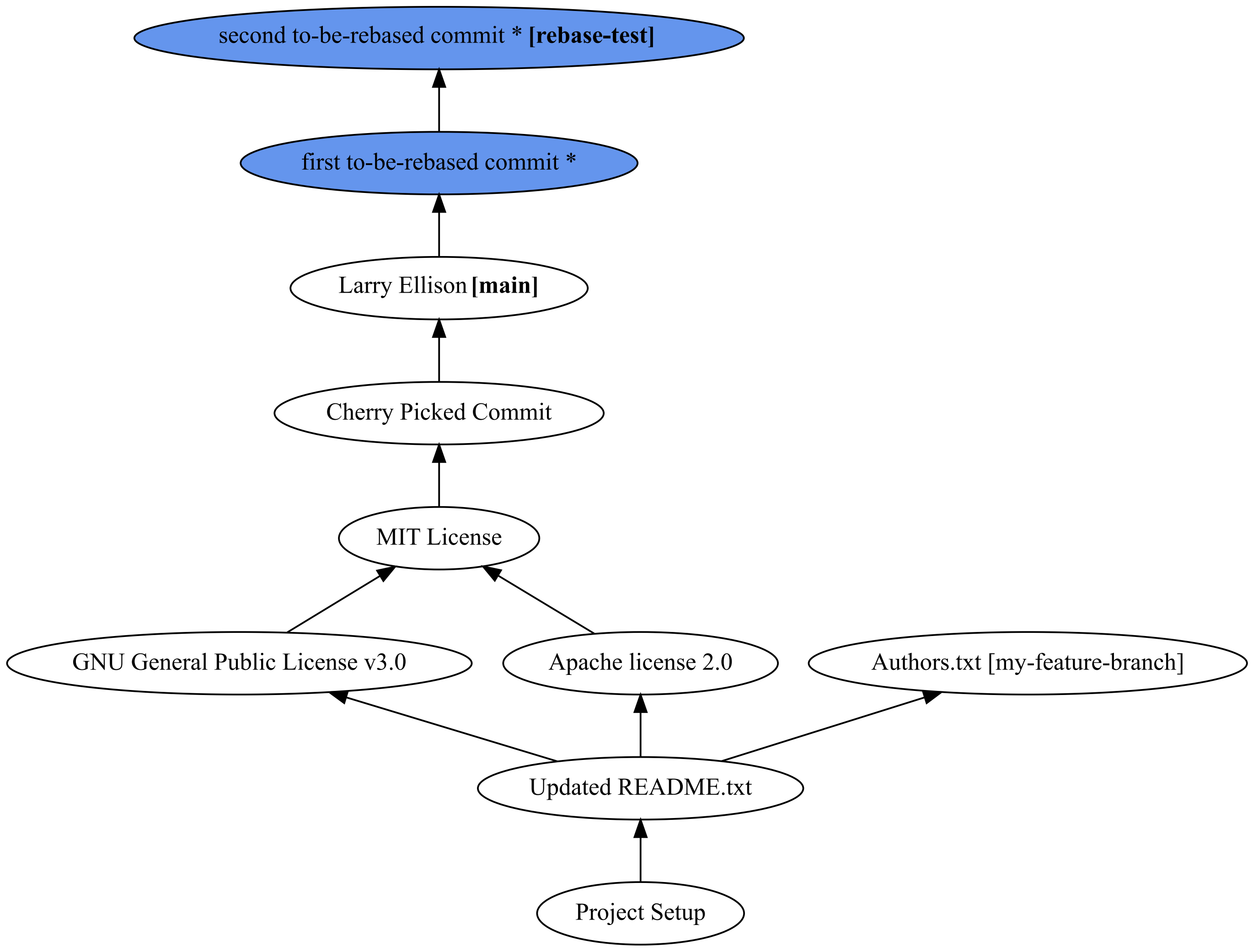
That's when the second cherry-pick happens.

- **Merge Base:** *`first-commit-to-be-rebased`* (the parent of the commit to be picked)
- **Diff 1 Candidate:** the *`second-to-be-picked-commit`*
- **Diff 2 Candidate:** the already cherry-picked *`first-commit-to-be-rebased *`*


second to-be-rebased commit **[rebase-test]**
first to-be-rebased commit *
first to-be-rebased commit
Larry Ellison **[main, detached]**
Cherry Picked Commit
MIT License
GNU General Public License v3.0
Apache license 2.0
Authors.txt [my-feature-branch]
Updated README.txt
Project Setup

This is when our second merge conflict happens, as the two *`first-to-be-rebased-commits`* already differ and the *`second-to-be-rebased-commit`* again changes the same lines in the same file.

After solving the second merge conflict, our rebase has finished and we'll end up with this graph, without a detached HEAD.


second to-be-rebased commit * [rebase-test]
first to-be-rebased commit *
Larry Ellison [main]
Cherry Picked Commit
MIT License
GNU General Public License v3.0
Apache license 2.0
Authors.txt [my-feature-branch]
Updated README.txt
Project Setup

What actually happened is that Git removed the `rebase-test` branch name off our `second-to-be-rebased commit` and pasted it into the final commit, which is the 'copy', i.e. `second-to-be-rebased commit *`.

Then it re-attached the HEAD pointer to our 'new' branch. Which means, you won't see your original commits from the `rebase-test` branch anymore and it looks like they've moved, but they are are still in Git's reflogs and the other commits you are seeing in the new `rebase-test-branch` are also completely new.

Phew, quite the story, huh?

Go through the example a couple of times yourself on the command line, to be ready for the final exam.

Final Exam

Here's a final question for you.

Imagine, in the rebase example above, you would have completely accepted the changes from your `first_to_be_rebased` commit during the first merge conflict.

Would rebasing your `second-to-be-rebased` commit then also have lead to another merge conflict? Why?

Also, would you have had to come up with new commit messages for either of those rebased commits? Why?

Let me know in the comments :)

Fin

Congratulations! That was a lot.

By now you should not only have a pretty good theoretical understanding of how Git's merges, rebases and cherry-picks work, but also a good practical understanding (you followed along on the command line, didn't you?).

What were your biggest revelations? Your biggest learnings? Most importantly: Do you now feel more comfortable with Git?

And which topics would you like to see covered in the next revision of this guide? Find a list of ideas in the next section.

Let me know on Twitter or simply send an email to marco@marcobehler.com. I'd love to hear from you!

Thanks for reading. Auf Wiedersehen.

Future Topics

- Highlighting different merge strategies (fast-word, recursive etc.)
- Using rebasing for squashes and other use-cases

Acknowledgements

A big thanks goes out to Chris Torek, who knows more about Git than I will ever know and whose Stackoverflow answers helped significantly shape this article.

Also, Joen Loeliger and Matthew McCullough, who wrote Version Control with Git.

Share

Comments

Login

Add a comment

M↓ MARKDOWN COMMENT ANONYMOUSLY ADD COMMENT

Upvotes Newest Oldest

I

Intelygenz
0 points · 10 months ago

If I accept the changes from the first-to-be-rebased commit then the second rebase step will not generate any merge conflicts as diff 2 candidate, i.e. diff between first-to-be-rebased commit and the newly cherry-picked first-to-be-rebased commit *, would be empty (the LICENSE.txt file in both are identical).

That being said, I didn't have to give different names for the commits using this strategy. Is there something that I'm missing?

M

Marco Behler
0 points · 10 months ago

Yup, correct < clap > :)

As for editing the commit message, you mean when there was a conflict or when there wasn't one?

I

Intelygenz
0 points · 10 months ago

Both

M

Marco Behler
0 points · 10 months ago

Interesting. If I remember correctly, when I hit a conflict, I definitely had to provide a commit message. If there wasn't one, then not. Will need to double check at some point.

?

Anonymous
0 points · 8 months ago

I bought this thinking it was complete. Looks like it is still in the works. Is that right? Will there be a downloadable PDF and video courses for this? Thanks!

M

Marco Behler
0 points · 8 months ago

Hi there, it actually is complete, not in the works. There's a few additional topics that might be added in a future revision, but I wanted to wait for some feedback from users first. As for PDF/video, nope, nothing concrete planned. Do you prefer video?

?

Anonymous
0 points · 10 months ago

Just curious where this commit came from?

cat'ing the first commit's file-tree/file-hash

git cat-file -p fe066d3f7568e13ef031b495e35c94be91b6366c

M

Marco Behler
0 points · 10 months ago

Ha, the wording isn't great. That's not the hash of the first commit, but the hash of the README.txt file snapshotted during the first commit. And for that, you'll have to go through log -> commit -> tree -> files. I'll fix the wording, thanks for the feedback.

Powered by **Commento**

Privacy & Terms Imprint Contact