Become a GIT pro by learning GIT architecture in 15 minutes

Max Koretskyi aka Wizard
Aug 22, 2017 · 17 min read

A step-by-step practical exercises that explore git internals

. . .

Git may seem like a complex system. Just ask google. Here are some of the titles that a quick search turns up:

> *Why the Heck is Git so Hard…*
> *Git is just too hard…*
> *Can we please stop pretending that Git is simple and easy to learn…*
> *Why is Git so complex and confusing…*

At first glance these statements may seem to hold true but once you understand the underlying concepts working with Git becomes a delightful experience. The problem with Git is that it's very flexible. Pretty much like Webpack. And the inherit characteristic of all flexible system is complexity. I strongly believe that the only way to battle this complexity is to get down to the basics underneath the provided user interface and understand the underlying mental model and architecture. Once you do that there will be no magic and unexpected results and you'll feel right at home working with these complex tools.

Both developers who have worked with Git before and who just start working with this awesome version control tool will benefit from the material in this article. If you're an experienced GIT user you will understand the lifecycle `checkout -> modify -> commit` much better. If you're just starting with Git this article will give you a great head start.

I'll be using low-level so called "plumbing" commands in this post to demonstrate how Git works under the hood. You don't need to remember them as they are rarely used during regular workflow but are indispensable when explaining the underlying Git architecture.

The article is a bit lengthy and I believe you can study it in the following way:

- skim the article top to bottom to understand the general flow of the article
- read thoroughly and follow along by implementing the exercises I use in the article

By practicing you will reinforce the knowledge you gain here. I use bash commands during exercises so you're OK if you're on Unix based OS. If you use Windows you can run `git-bash` which is installed alongside Git and then can be accessed using context menu option `Git Bash here`.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular/React/Vue data grid solution, give it a try with our guide "**Get started in 5 minutes**" guide. I'm happy to answer any questions you may have. **And follow me to stay tuned**!

. . .

Git as a folder

When you run `git init` in a folder Git creates the `.git` directory. So let's open a terminal, create a new directory where we will be working and initialize an empty git there:

```
$ mkdir git-playground && cd git-playground
$ git init
Initialized empty Git repository in path/to/git-playground/.git/

$ ls .git
HEAD config description hooks info objects refs
```

This is where Git stores all your commits and other relevant information to manipulate these commits. When you clone a repository Git copies this single directory into your folder, creates remote-tracking branches for each branch in the cloned repository and creates and checks out an initial branch that is specified by the HEAD file. We will see later what purpose the HEAD file serves in Git architecture but the takeaway here is that cloning a repository is essentially just copying `.git` directory from the other location.

Git as a database

Git is a simple key-value data store. You put a value into the repository and get a key by which this value can be accessed. The command that puts value into a database is `hash-object` and it returns a 40-character checksum hash that will be used as a key. The command creates a plain object in git repository that is called a `blob`. Let's write a simple string `f1 content` into the database:

```
$ F1CONTENT_BLOB_HASH=$( \
     echo 'f1 content' | git hash-object -w --stdin )
$ echo $F1CONTENT_BLOB_HASH
a1deaae8f9ac984a5bfd0e8eecfbafaf4a90a3d0
```

For those of you who are not very familiar with shell, the main command in the above snippet is:

```
echo 'f1 content' | git hash-object -w --stdin
```

The `echo` command outputs `f1 content` string and by using the pipe operator `|` we redirect the output to the `git hash-object` command. The param `-w` passed to the `hash-object` command tells it to store the object; otherwise, the command simply tells you what the key would be. `--stdin` tells the command to read the content from stdin; if you don't specify this, `hash-object` expects a file path at the end. As mentioned earlier `git hash-object` command returns a hash that I then save into `F1CONTENT_BLOB_HASH` variable. We could have also split the main execution and the variable assignment like this:

```
$ echo 'f1 content' | git hash-object -w --stdin
a1deaae8f9ac984a5bfd0e8eecfbafaf4a90a3d0

$ F1CONTENT_BLOB_HASH=a1deaae8f9ac984a5bfd0e8eecfbafaf4a90a3d0
```

But for convenience I'll be using a shorter version in the subsequent code snippets to assign outputs into variables. Those variables are used where a hash string is expected and to read a value the variable name is prepended with `$`.

To read a value by key we can use `cat-file` command with the `-p` option. The command expects a hash key of the object to retrieve:

```
$ git cat-file -p $F1CONTENT_BLOB_HASH
f1 content
```

As I said before `.git` is a folder and all stored values/objects are kept in this folder. So we can go into `.git/objects` and you will see the folder created by Git with the name `a1` which is the first two letters of the hash key:

```
$ ls .git/objects/ -l
a1/
info/
pack/
```

This is the way Git usually stores objects — one folder per one blob. However, Git can also merge multiple blobs into one file to generate pack files and the `pack` directory you see above is where these files are stored. Git keeps information related to these

pack objects into the `info` directory. Git generates hash for blobs based on the blob contents and so objects in Git are immutable because changing the contents would change the hash.

Let's write another string `f2 content` into the repository:

```
$ F2CONTENT_BLOB_HASH=$( \
     echo 'f2 content' | git hash-object -w --stdin )
```

As expected you can see that `\.git\objects` folder now contains two records `9b/` and `a1/`:

```
$ ls .git/objects/ -l
9b/
a1/
info/
pack/
```

Tree as an integral part

Currently we have two blobs in our repository:

```
F1CONTENT_BLOB_HASH -> 'f1 content'
F2CONTENT_BLOB_HASH -> 'f2 content'
```

We need a way to somehow group them together and also associate each blob with a filename. And this is where a tree comes into play. A tree can be created using `git mktree` command with the following syntax for each file/blob association:

```
[file-mode object-type object-hash file-name]
```

See this answer for the explanation of the file mode. We will use the mode `100644` that defines a blob as a regular file which can be read and written by the user. Those permissions are used when checking out files to a working directory to set permissions on files\directories corresponding to the trees entries.

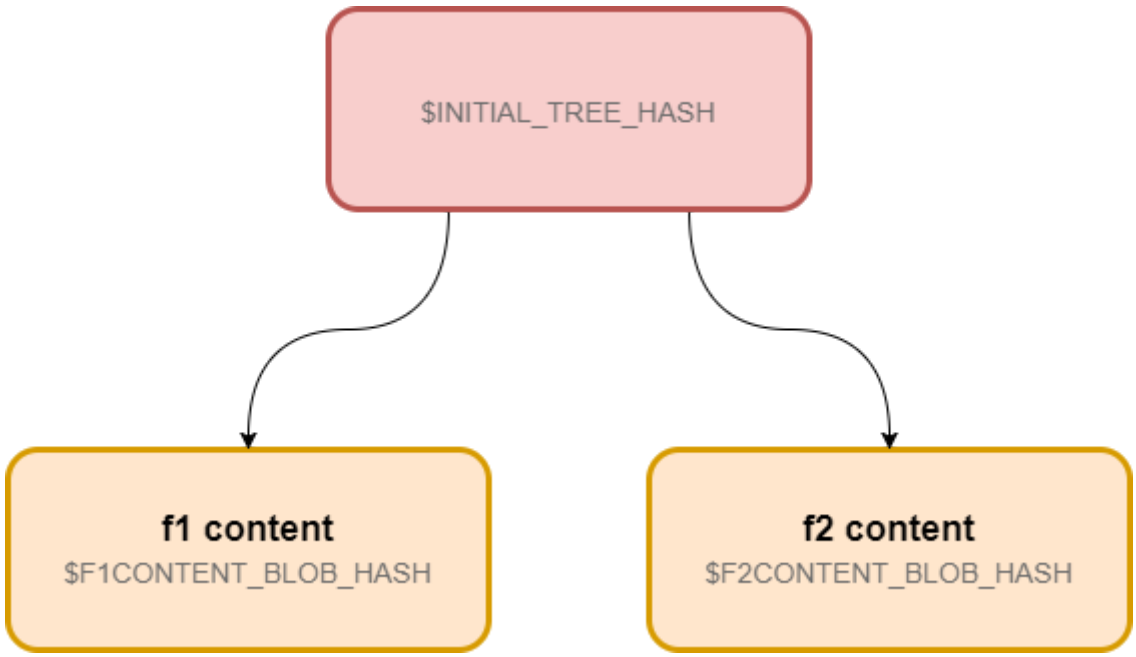
So to associate two blobs with two files we'll do the following:

```
$ INITIAL_TREE_HASH=$( \
    printf '%s %s %s\t%s\n' \
      100644 blob $F1CONTENT_BLOB_HASH f1.txt \
      100644 blob $F2CONTENT_BLOB_HASH f2.txt |
    git mktree )
```

Just as with `hash-object` the `mktree` command returns a hash key for the created tree object:

```
$ echo $INITIAL_TREE_HASH
e05d9daa03229f7a7f6456d3d091d0e685e6a9db
```

So here is what we have now:



After running the command git creates a third object in the repository of type `tree`. Let's see it:

```
$ ls .git/objects -l
e0   <--- initial tree object  (INITIAL_TREE_HASH)
9b   <--- 'f1 content' blob    (F2CONTENT_BLOB_HASH)
a1   <--- 'f2 content' blob    (F2CONTENT_BLOB_HASH)
```

When using `mktree` we can specify another tree object as a parameter instead of a blob. That tree will be associated with the directory instead of a regular file. For example, the

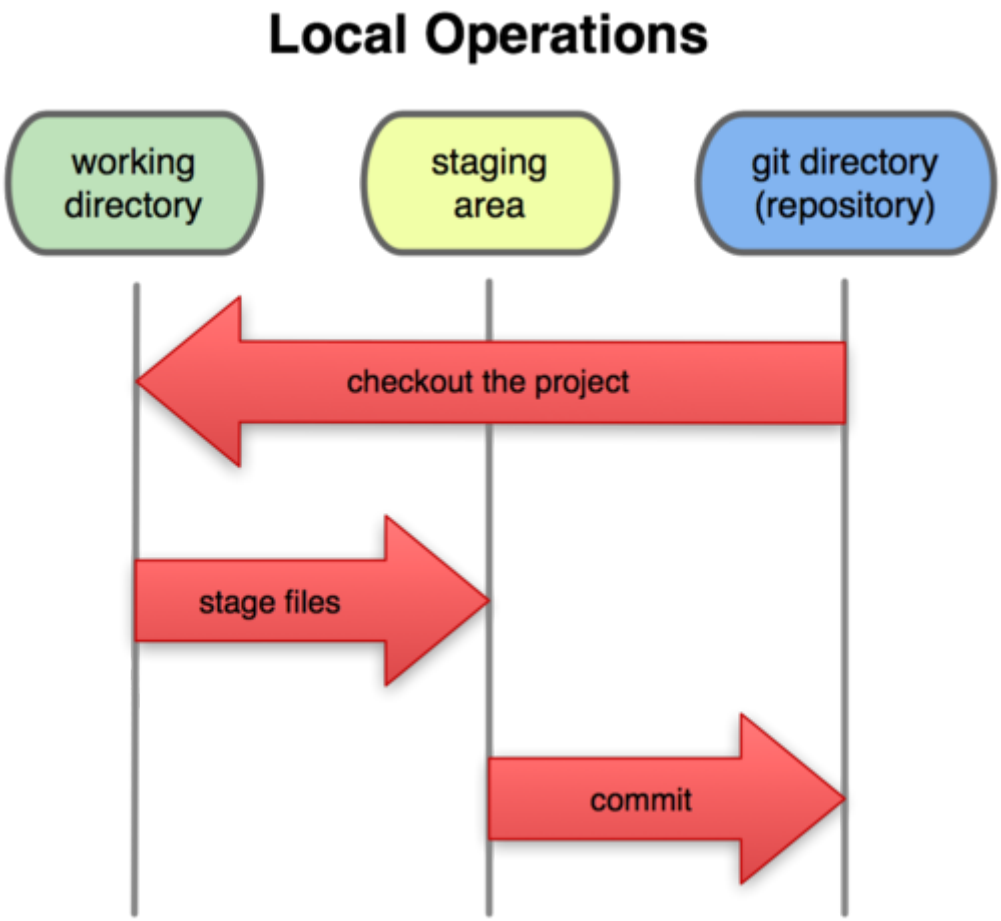
following command would create a tree with a subtree associated with the `nested-folder` directory:

```
printf '%s %s %s\t%s\n' 040000 tree e05d9da nested-folder | git mktree
```

The filemode `040000` marks a directory and we use the type `tree` instead of a `blob`. This is how git stores nested directories in the project structure.

Index is a place where trees are assembled

Everyone working with GIT should be familiar with the notion of index or staging area and have probably seen the following diagram:



On the right side you can see the git repository which stores git objects: blobs, trees, commits and tags. We used `hash-object` and `mktree` commands to directly add a blob and a tree objects to this repository. The working directory on the left is your local file system/directory where you checkout all the project files. This section explains the middle stage which we will refer to as the index file or simply the index. It is a binary

file (generally kept in `.git/index`) that resembles the structure of a tree object. It holds a sorted list of path names, each with permissions and the SHA1 of a blob/tree object.

It is the place where git prepares a tree before it:

- writes it into the repository or
- checks it out into a working directory

We now have one tree in the repository that we created in the previous chapter. We can read this tree into the index file from the repository using the `read-tree` command:

```
$ git read-tree $INITIAL_TREE_HASH
```

So now we expect the index file to have two files. We can check the structure of the current index file using `git ls-files -s` command:

```
$ git ls-files -s
100644 a1deaae8f9ac984a5bfd0e8eecfbafaf4a90a3d0 0 f1.txt
100644 9b96e21cb748285ebec53daec4afb2bdcb9a360a 0 f2.txt
```

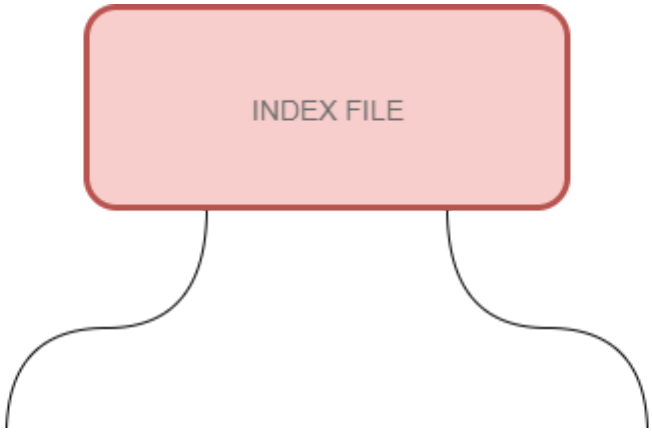
Since we haven't made any changes to the index file it is exactly the same as the tree we used to populate the index file. Once we have correct structure in the index file let's check it out into the working directory using `checkout-index` command with `-a` option:

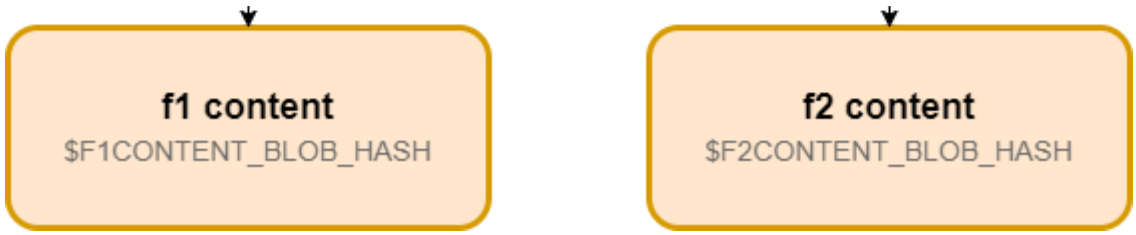
```
$ git checkout-index -a

$ ls
f1.txt f2.txt

$ cat f1.txt
f1 content

$ cat f2.txt
f2 content
```

All right! We have just checked out the files we added manually into the git repository without any commits. How cool is that?

But the index file doesn't usually stay in the state of the initial tree. You probably know that it can be modified using `git add [file path]` and `git rm --cached [file path]` commands for a single file or `git add -A` and `git reset` for the set of modified/deleted files. So let's put this knowledge into practice and create a new tree in the repository containing one file blob associated with the new `f3.txt` text file. The contents of the file will be the `f3 content` string. But instead of creating the tree manually like we did in the previous section we will use the index file for that.

Currently we have the following structure of the index file based on the `initial tree` tree we used to populate the index:





This is the base to which we will be applying changes. All the changes you make to the index file are tentative until you actually write a tree to the repository. The objects however that you add are written into the git repository immediately. If you discard the current changes to the tree they will later be picked up by the garbage collection (GC) and removed. It also means that usually if you accidentally discard changes to a file they can still be recovered until git runs GC. And it usually does so only when there are too many loose objects floating around that are not referenced anywhere.

Let's start by removing two files in the working directory:

```
$ rm f1.txt f2.txt
```

If we now run `git status` we will see the following:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   f1.txt
        new file:   f2.txt

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:    f1.txt
        deleted:    f2.txt
```

That's a lot of information. It reports two deleted files and two new files and it also says "initial commit". And here is why. When you run `git status` git makes two comparisons:

- it compares index file with current working directory — changes reported as “not staged for commit”
- it compares index file with HEAD commit — changes reported as “to be committed”

So in our case we see that git reports two deleted files for “Changes not staged for commit” and we know how it got it — it compared the current working directory with index file and found two files missing in the working directory (because we removed them).

We also see that under “Changes to be committed” git reports two new files. This is because currently we don’t yet have any commits in the repository so `HEAD` file (will be explained later) resolves to a so-called “empty tree” object with no files. So Git thinks we just started with the new fresh repository and that’s why it shows “initial commit” and treats every file in the index file as a new file.

Now if we run `git add .` it will modify the index file by removing two files and when we run `git status` again it will report no changes since we have no files neither in the working tree nor in index file:

```
$ git add .
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

We started with the task to create a new tree with one new file `f3.txt`. Let’s create this file and add it to the index:

```
$ echo 'f3 content' > f3.txt
$ git add f3.txt
```

If we run `git status` now:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   f3.txt
```

We can see that one new file is detected. Again, the changes are reported under “to be committed” so we now Git compared the index file with the “empty tree”. So we expect the index file to have this one new file blob. Let’s check it:

```
$ git ls-files -s
100644 5927d85c2470d49403f56ce27afd8f74b1a42589 0       f3.txt

# Save the hash of the f3.txt file blob
$ F3CONTENT_BLOB_HASH=5927d85c2470d49403f56ce27afd8f74b1a42589
```

Okay, the index has the correct structure now and we’re ready to create a tree from it in the repository. Let’s do that with `write-tree` command:

```
$ LATEST_TREE_HASH=$( git write-tree )
```

Great, we’ve just created the tree with the help of index. And we put the hash of the new tree into the `LATEST_TREE_HASH` variable. We could’ve done it manually by writing the `f3 content` blob to the repository and then creating a tree with `mktree` but using index is much more convenient.

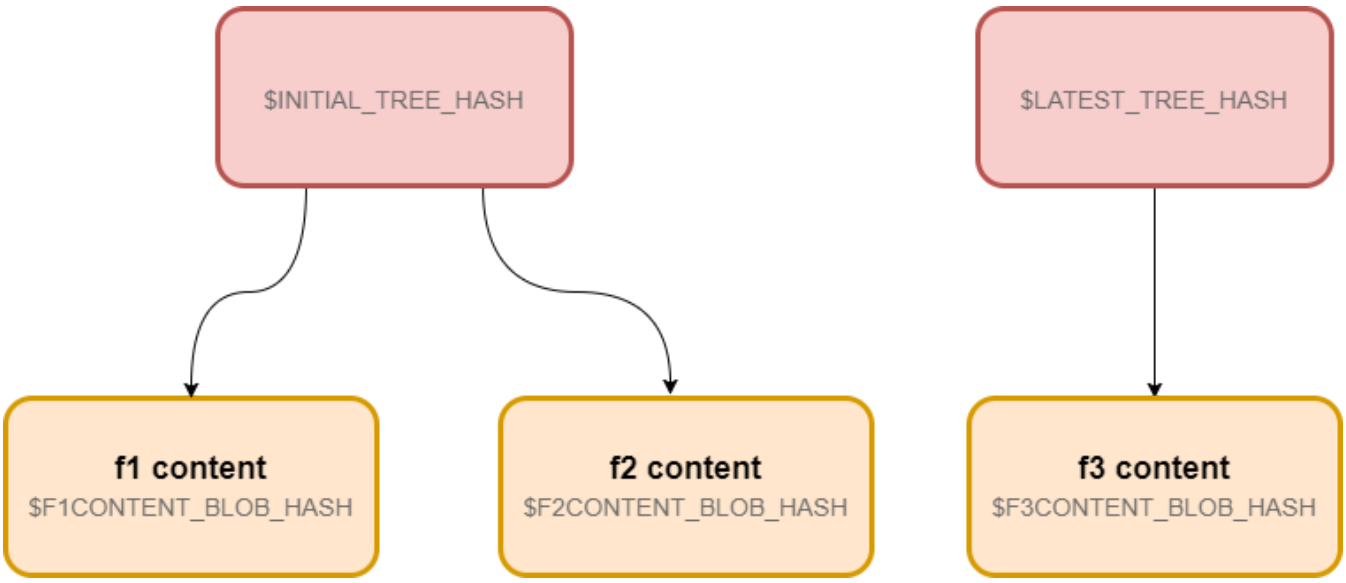
What’s interesting is that if you run `git status` now you will still see that git still thinks that there’s new file `f3.txt` :

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```

```
    new file:   f3.txt
```

That's because although we've created and saved our tree into the repository we didn't update the HEAD file that is used for comparison. Since we can only put a commit hash or a branch reference into the HEAD file but have neither now we will leave HEAD file as is for now.

So with this newly created tree we have the following objects in the repository:



A commit is a wrapper around a tree

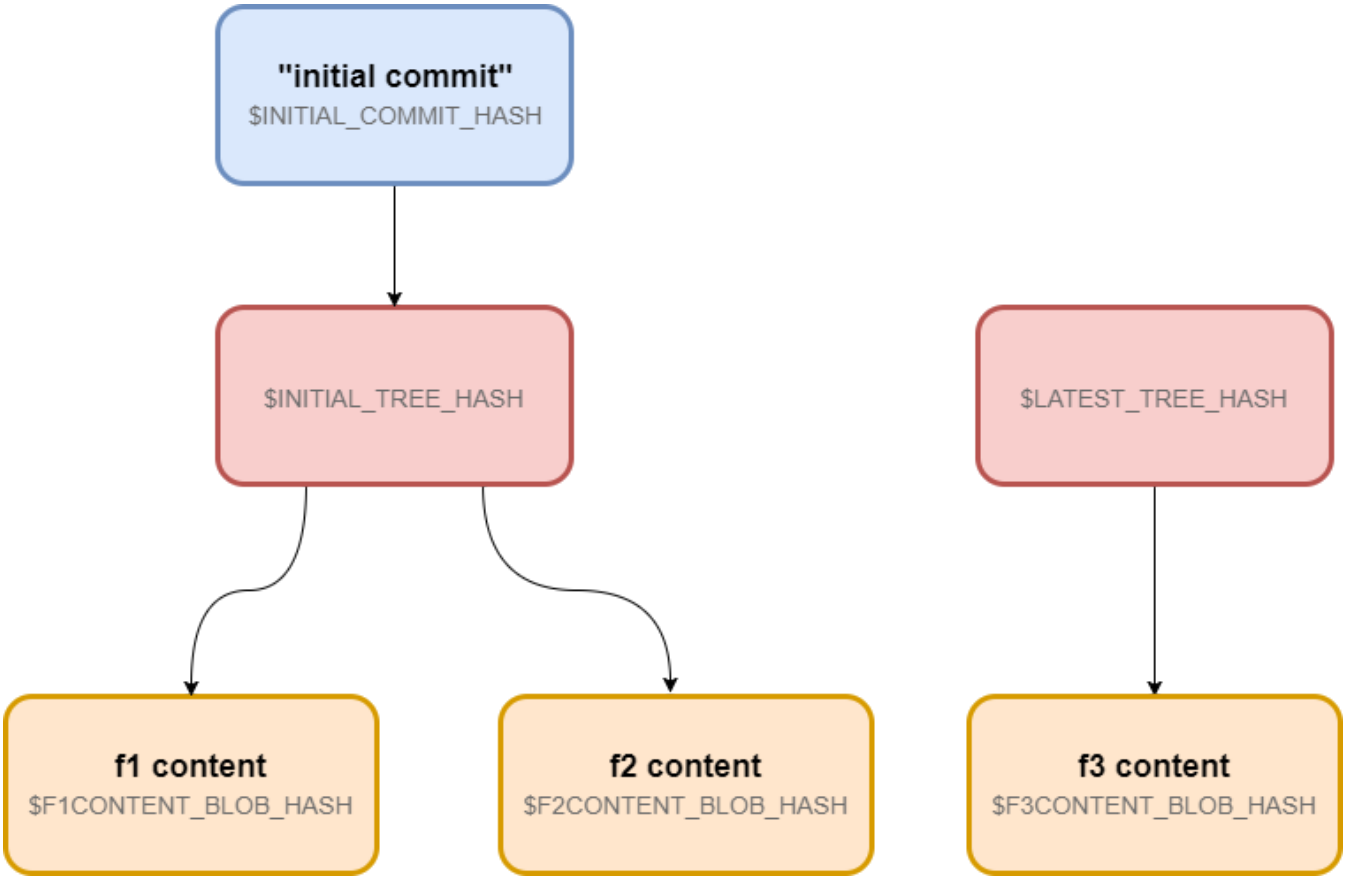
In this section it gets even more interesting. During our everyday work with Git we don't usually encounter trees or blobs. We work with commits objects. So what is a commit in git then? Actually, as simple as it gets it's just a **wrapper** around the tree object that:

- allows attaching a message to a tree (group of files)
- allows specifying parent (commit)

We now have two trees in our git repository — `initial tree` and `latest tree`. Let's wrap the first tree object in a commit using the `commit-tree` command that takes a hash of the tree to create a commit for:

```
$ INITIAL_COMMIT_HASH=$( \
    echo 'initial commit' | git commit-tree $INITIAL_TREE_HASH )
```

After you run the above command we will have the following:



And we can checkout that commit into the working directory:

```
$ git checkout $INITIAL_COMMIT_HASH
A       f3.txt
HEAD is now at a27a75a... initial commit
```

Now we can see our two files in the working directory:

```
$ ls
f1.txt f2.txt

$ cat f1.txt
f1 content

$ cat f2.txt
f2 content
```

When you run `git checkout [commit-hash]` git does the following:

- reads the tree the commit points at into the index file
- checks out index file into working directory
- updates HEAD file with the commit hash

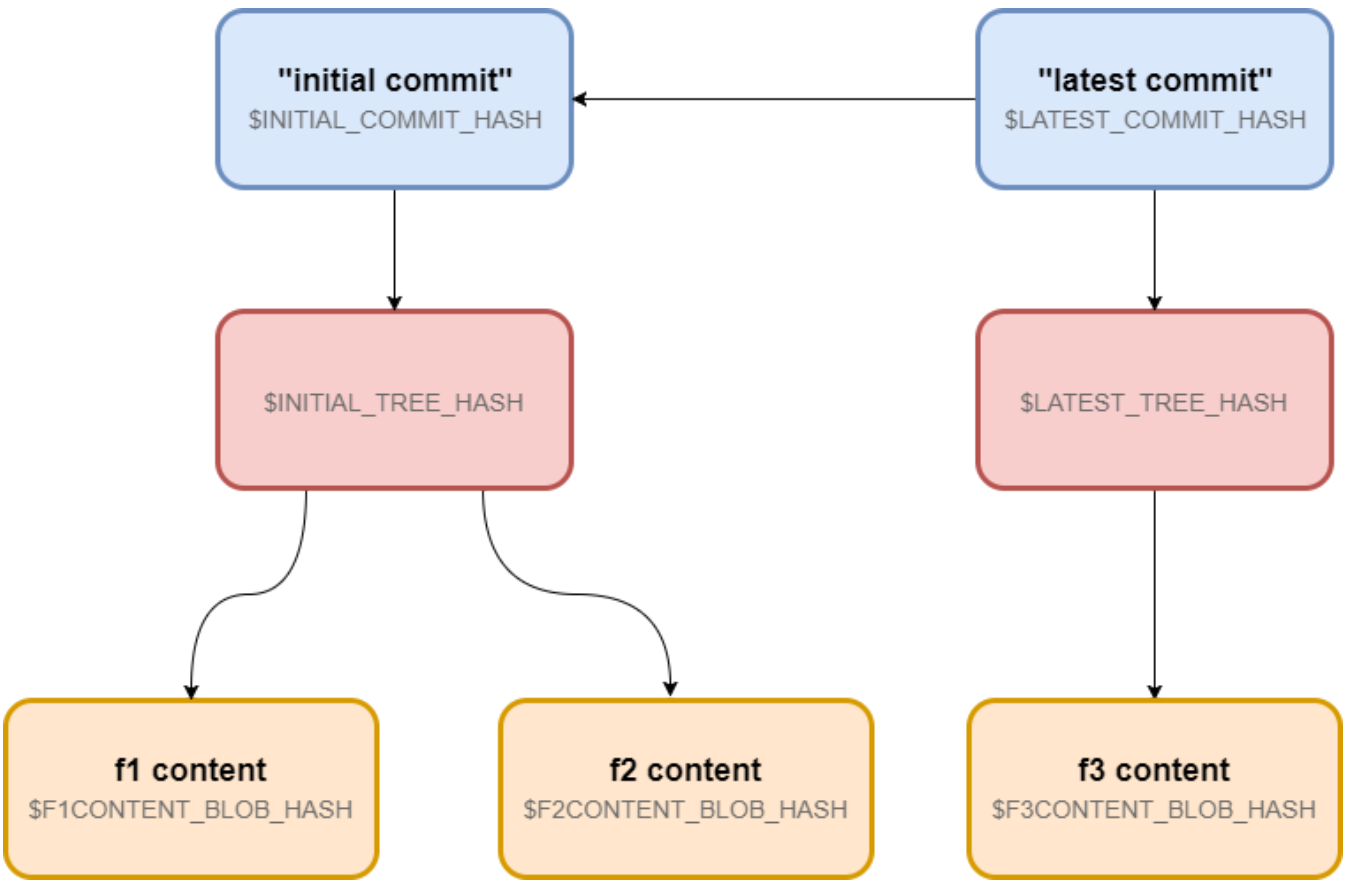
These are the operations that we did manually in the previous section.

Git history is a chain of commits

So now we know that a commit is just a wrapper around a tree. I also mentioned that it can have a parent commit. We had two trees initially and wrapped the one of them into a commit in the previous section so we still have one orphan tree left. Let's wrap it into the other new commit and make the initial commit a parent of this new commit. We will use the same `commit-tree` operation I used above but with `-p` option to specify parent:

```
$ LATEST_COMMIT_HASH=$( \
    echo 'latest commit' |
    git commit-tree $LATEST_TREE_HASH -p $INITIAL_COMMIT_HASH )
```

And so here is what we have now:



So now if you run `git log` to see a history and pass the hash of the "latest" commit you will see two commits:

```
$ git log --pretty=oneline $LATEST_COMMIT_HASH
[some hash] latest commit
[some hash] initial commit
```

And we can switch between them. Here is the initial commit:

```
$ git checkout $INITIAL_COMMIT_HASH
$ ls
f1.txt f2.txt
```

Latest commit:

```
$ git checkout $LATEST_COMMIT_HASH
$ ls
f3.txt
```

The HEAD is a reference to the checked out commit

HEAD is a simple text file located at `.git/HEAD` that references a currently checked out commit. Since we checked out the "latest" commit with the hash `$LATEST_COMMIT_HASH` commit in the previous section this is exactly what the `HEAD` file contains:

```
$ cat .git/HEAD
88d3b9901d62fc1de9219f388e700d98bdb97ba9

$ [ $LATEST_COMMIT_HASH == "88d3b9901d62..." ]; echo 'equal'
equal
```

However, usually HEAD file references the currently checked out commit through branch references. When it references the commit directly it is in a `detached state`. But even when HEAD holds a reference to a branch like this:

```
ref: refs/heads/master
```

it still resolves to the commit hash.

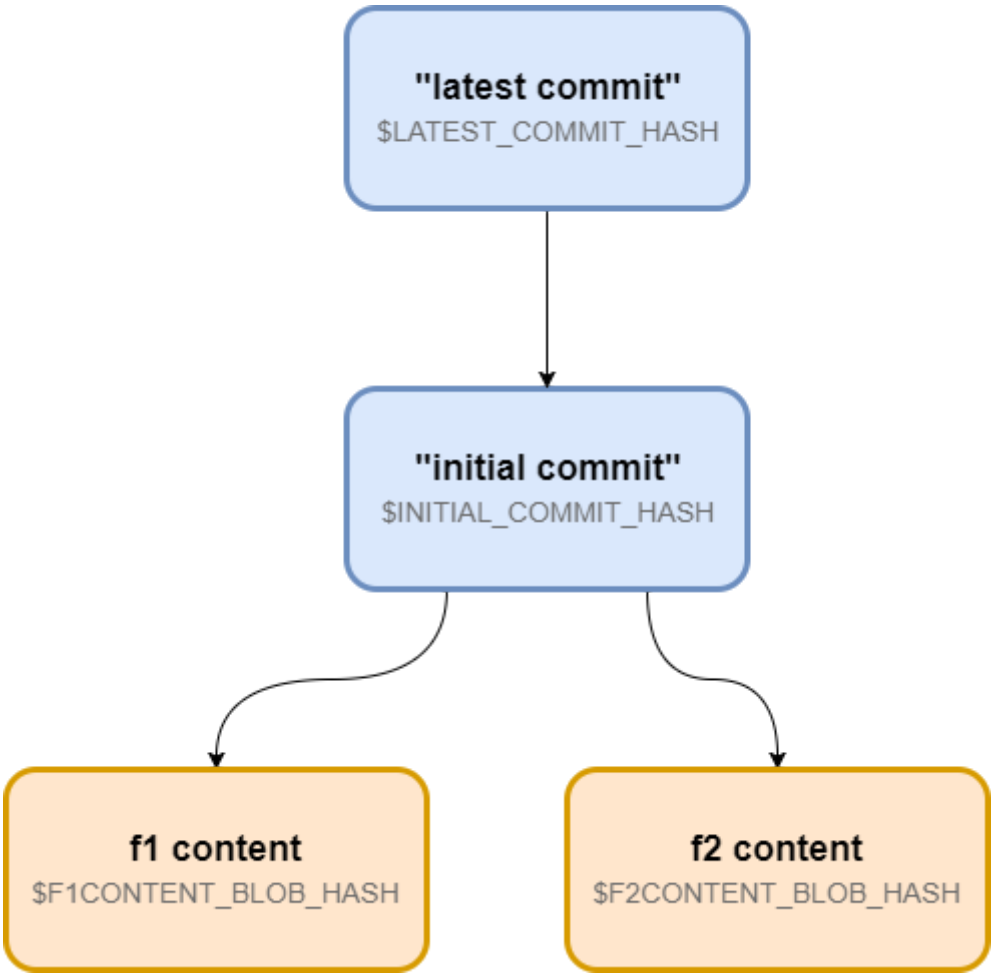
You already know that Git uses the commit referenced by the `HEAD` when executing `git status` to produce a set of changes between the index file and the currently checked out tree/commit. Another usage for the `HEAD` is to resolve a commit that will be used as a parent for the future commit.

Interestingly, HEAD file is so important for most operations that if you manually clear its contents Git thinks that it is not a git repository and reports an error:

```
fatal: Not a git repository (or any of the parent directories): .git
```

A branch is a text file pointing to a commit

So now we have two commits in our repository that constitute the following history:



```
$ git log --pretty=oneline $LATEST_COMMIT_HASH
[some hash] latest commit
[some hash] initial commit
```

Let's introduce some fork into the existing history. We will checkout the initial commit and modify the `f1.txt` file contents. After that we will make a new commit using the `git commit` command that you're accustomed to:

```
$ git checkout $INITIAL_COMMIT_HASH
$ echo 'I am modified f1 content' > f1.txt
$ git add f1.txt

$ git commit -m "forked commit"
1 file changed, 1 insertion(+), 1 deletion(-)
```

The above code snippet:

- checks out `"initial commit"` that adds `f1.txt` and `f2.txt` to the working directory
- replaces `f1.txt` contents with `I am modified f1 content` string
- updates index file with `git add`

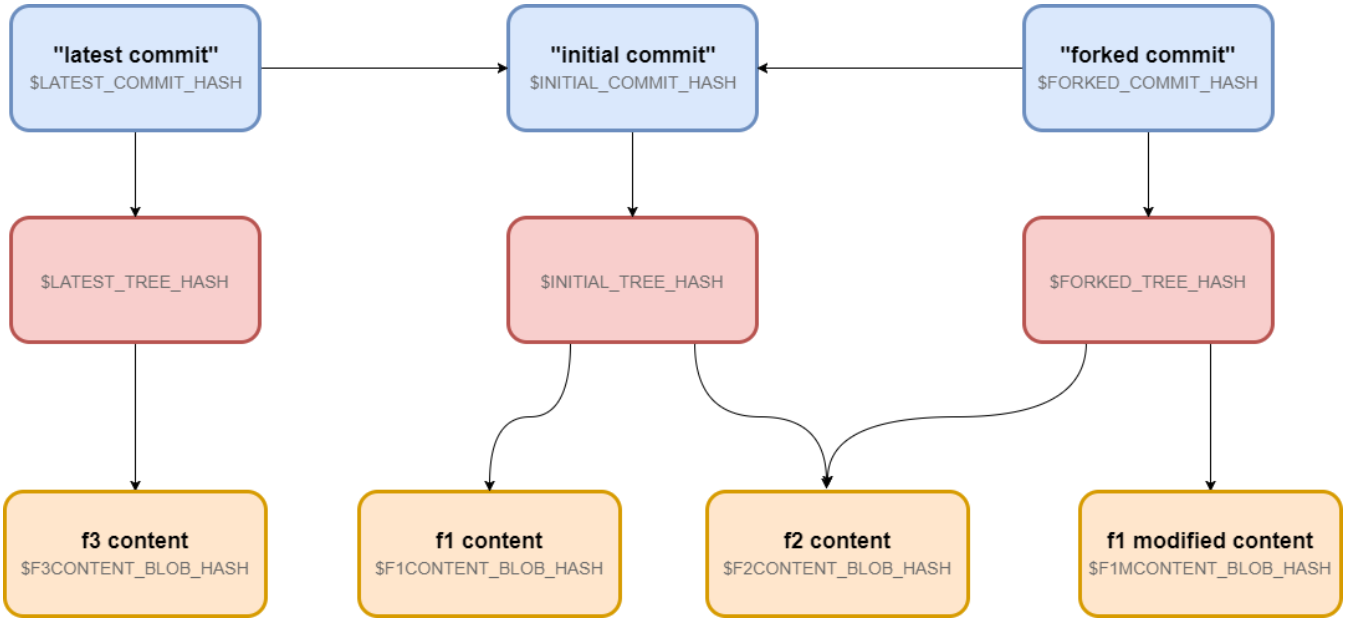
The last `git commit` command as we already know performs multiple under-the-hood operations:

- creates a tree from the index file
- writes this tree to the repository
- creates a commit object that wraps this tree
- sets the `initial commit` as a parent of the new commit since it's the commit that we currently have in the `HEAD` file

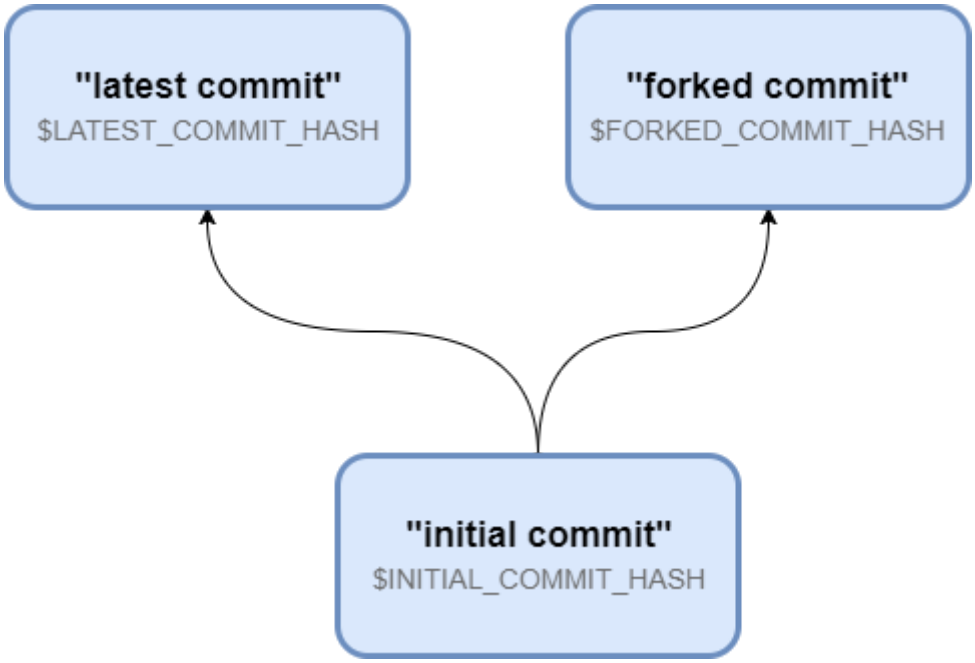
We also need to save the hash of that commit into a variable. Since Git updates HEAD with the current commit file we can read it from there:

```
FORKED_COMMIT_HASH=$( cat .git/HEAD )
```

So now we have the following objects in our git repository:



Which creates the following commit history:



Due to the presence of a fork we have two lines of work here. And it means we need to introduce two branches to track each line of work independently. Let's create a `master` branch that tracks the linear history starting from `latest commit` and the `forked` branch that tracks the history from the `forked commit`.

A branch is a text file which that contains a hash of a commit. It is part of git references — a group of objects that reference a commit. The other reference type is a lightweight

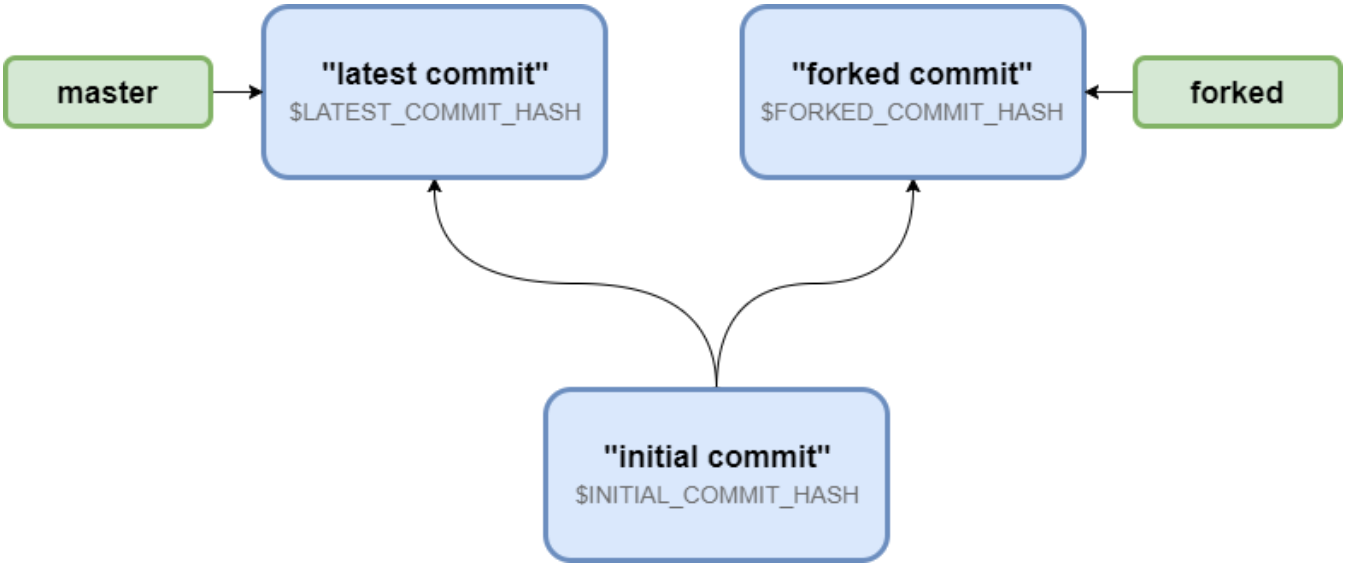
tag. Git stores all references under `.git/refs` folder and branches are stored in the directory `.git/refs/heads` . Since branch is a simple text file we can just create a file with the contents of a commit hash.

So this will point to the main branch with the “latest commit”:

```
$ echo $LATEST_COMMIT_HASH > .git/refs/heads/master
```

And this will point to the “forked” branch with the “forked commit”:

```
$ echo $FORKED_COMMIT_HASH > .git/refs/heads/forked
```



So finally we’ve come to the regular workflow you’re used to — we can now switch between branches:

```
$ git checkout master
Switched to branch 'master'

$ git log --pretty=oneline
[some hash] latest commit
[some hash] first commit

$ ls -l
f3.txt
```

And let's see another `forked` branch:

```
$ git checkout forked
Switched to branch 'forked'

$ git log --pretty=oneline
f30305a8a23312f70ba985c8c644fcdca19dab95 forked commit
f30305a8a23312f70ba985c8c644fcdca19dab95 initial commit

$ git ls
f1.txt f2.txt

$ cat f1.txt
I am modified f1 content
```

A tag is a text file pointing to a commit

You probably know that instead of a line of work that we track with branches we can track individual commits with tags. Tags are usually used to mark important development milestones like releases. Right now we have 3 commits in our repository. And we can give them names using a tag. Just as a branch a tag is a text file which contains a hash of a commit and is a part of git references group.

As you already know git stores all references under `.git/refs` folder and tags are stored in the subfolder `.git/refs/tags`. Since it's a simple text file we can create a file and put the commit hash into it.

So this will point to the latest commit:

```
$ echo $FORKED_COMMIT_HASH > .git/refs/tags/forked
```

And this will point to the initial commit:

```
$ echo $INITIAL_COMMIT_HASH > .git/refs/tags/initial
```

Once we've done that we can switch between commits using tags. Here is the initial commit:

```
$ git checkout tags/initial
HEAD is now at 285aec7... second commit

$ cat f1.txt
f1 content
```

And the forked commit:

```
$ git checkout tags/forked
$ cat f1.txt
I am modified f1 content
```

There's also "annotated tag" which is different from this light-weighted tag. It's an actual object that can contain a message just like a commit and is stored in the repository alongside other objects.

. . .

Conclusion

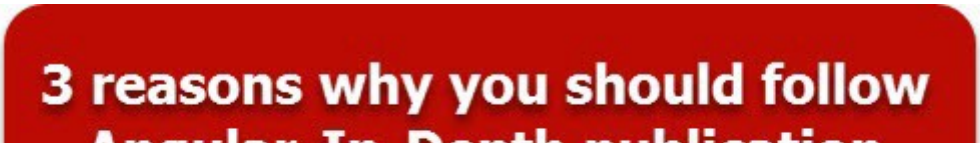
This is pretty lengthy article but I've tried to make it as transparent and clear as possible. Once you work through the article and understand all concepts I showed here you will be able to work with Git more effectively and you should never have fear of unexpected results.

If you want to learn more about Git I highly recommend reading this great book which is also free online. I'm in the process of writing the next article on Git that will explore the concepts of merge, rebase and remote repositories. Do follow me to get notified!

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.



Git Web Development Javascript Development Angular React

About Help Legal