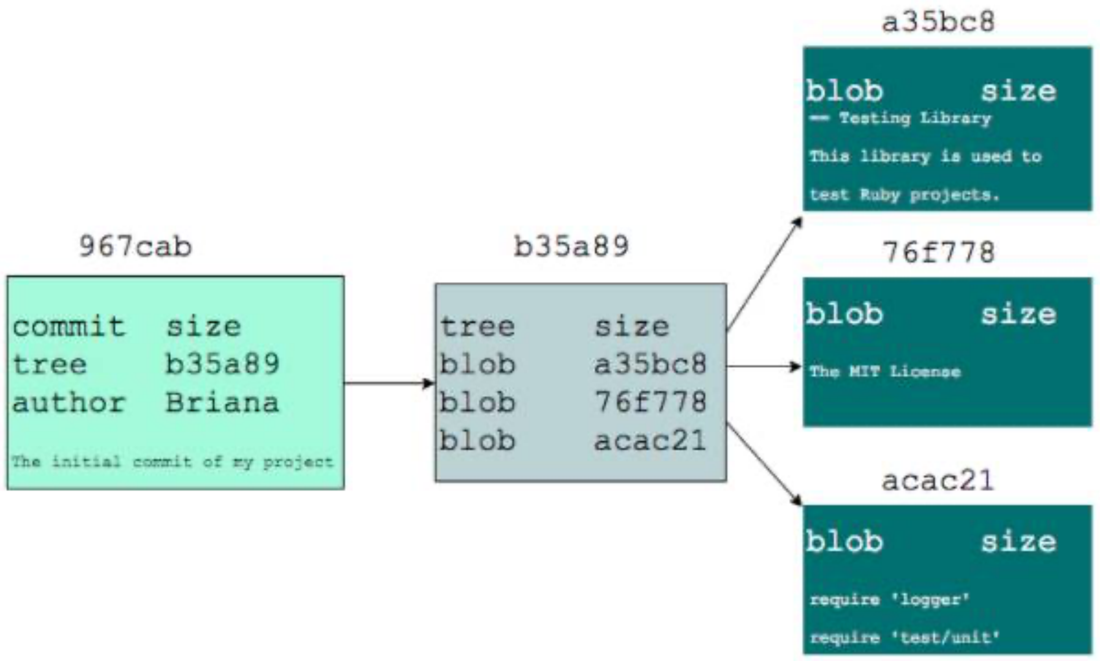
Work with Your History in Git

How Git Stores Data

When we previously discussed commits, we identified them as snapshots of your project. Each snapshot contains a lot of information. The files compressed in the snapshot are each given a unique SHA-1 hash—referred to as blobs. Those blobs are referenced by a tree, and that tree is referenced by the commit.



As you make changes to your files and create new commits, Git identifies the changed files and applies a new SHA-1 hash to the file, while unchanged files retain their existing SHA-1 hash. As the SHA-1 hash for a file changes, it references the previous SHA-1 hash as the parent. In GitHub, you can see the first seven characters of the SHA-1 hash (**A**) on any commit.

Note

Although SHA-1 hashes are being used today, it's important to note that the Git project has decided to pick a new hashing algorithm moving forward. Git has chosen SHA-256 as the successor to SHA-1, and work is currently in progress towards this transition.

This is very similar to how your commit history operates. As you create new commits, they reference the previous commit as its parent. This reference point is very important. This linear, parent/child relationship creates a consistent history and is what enables Git to merge branches together.

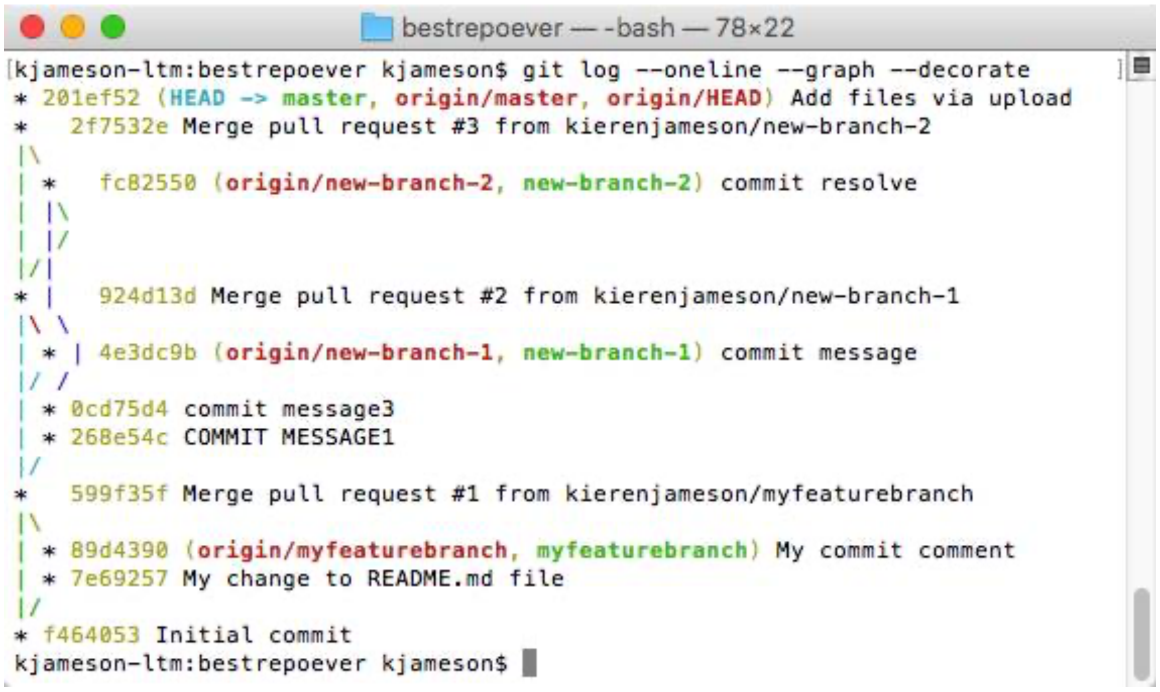
Explore Your History with Git

While working on your project it can be helpful to review your commit history. On GitHub.com, you can access your project history by selecting the commit button from the code tab on your project. Locally, you can use `git log`.

The `git log` command enables you to display a list of all of the commits on your current branch. By default, the `git log` command presents a lot of information all at once.

Use some of the `git log` modifiers to cultivate an easy-to-read list that provides some valuable information.

- `git log -10` will only show the 10 most recent commits.
- `git log --oneline` is a great way to view commit history by displaying the first seven characters of the SHA-1 hash and commit message of the commits on the current branch.
- `git log --oneline --graph` presents commit history in a ASCII graph displaying the different branches in the repository and their commits.
- `git log --oneline --graph --decorate` displays the same ASCII graph that is displayed using the --graph modifier, but also includes the branch name(s) for the different commits being displayed.

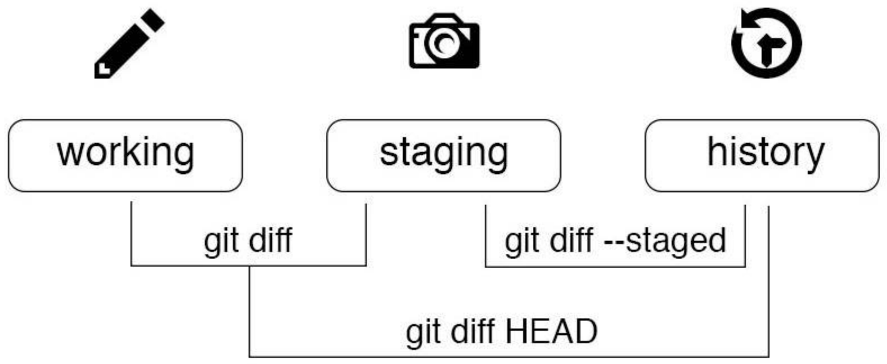


```
kjameson-ltm:bestrepoever kjameson$ git log --oneline --graph --decorate
* 201ef52 (HEAD -> master, origin/master, origin/HEAD) Add files via upload
*   2f7532e Merge pull request #3 from kierenjameson/new-branch-2
|\
| *   fc82550 (origin/new-branch-2, new-branch-2) commit resolve
| |\
| |/
|/|
* |   924d13d Merge pull request #2 from kierenjameson/new-branch-1
|\ \
| * | 4e3dc9b (origin/new-branch-1, new-branch-1) commit message
|/ /
| * 0cd75d4 commit message3
| * 268e54c COMMIT MESSAGE1
|/
*   599f35f Merge pull request #1 from kierenjameson/myfeaturebranch
|\
| * 89d4390 (origin/myfeaturebranch, myfeaturebranch) My commit comment
| * 7e69257 My change to README.md file
|/
* f464053 Initial commit
kjameson-ltm:bestrepoever kjameson$
```

Compare Versions of Files

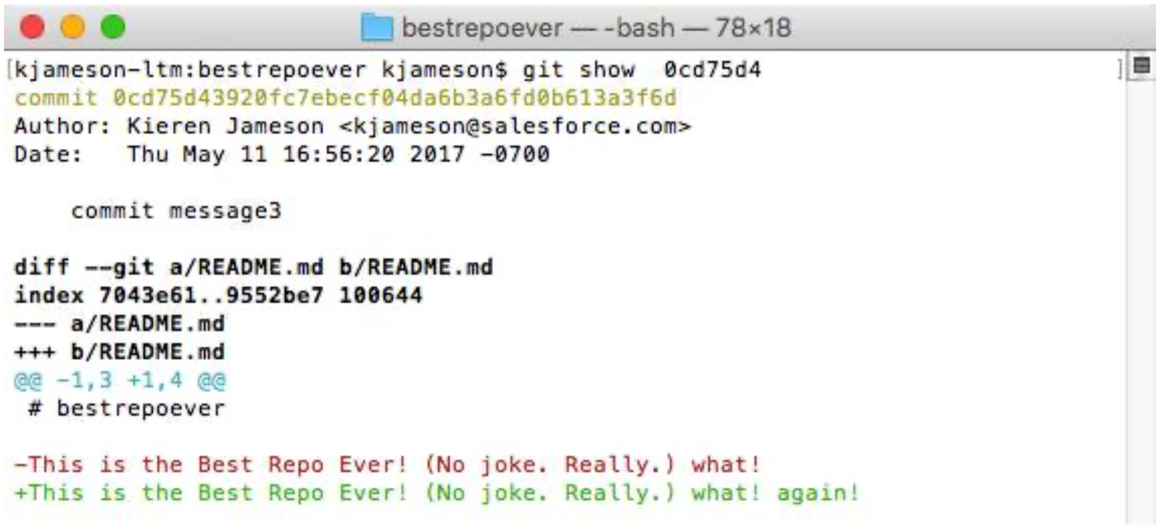
As you prepare to craft that perfect commit, viewing the differences between what is currently in your working directory and staging area helps you `git add` the right files to your commit.

By default, the `git diff` command helps you review the changes between the last commit of your project and the various states of your files (for example, those in the working directory or your staging area).



You can also use `git diff` to compare between any two commits, branches, or tags in the repository. For example, to compare two commits with SHA-1 hash references `4e3dc9b` and `0cd75d4`, enter command: `git diff 4e3dc9b 0cd75d4`

Finally, if you would like to view the changes that were made in a previous commit, you can use the `git show <SHA-1>` command to display the details of that specific commit. It includes things like commit author, time and date of the commit, and a list of the changes that were made to the various assets within the repository.



```
kjameson-ltm:bestrepoever kjameson$ git show  0cd75d4
commit 0cd75d43920fc7ebecf04da6b3a6fd0b613a3f6d
Author: Kieren Jameson <kjameson@salesforce.com>
Date:   Thu May 11 16:56:20 2017 -0700

    commit message3

diff --git a/README.md b/README.md
index 7043e61..9552be7 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,4 @@
 # bestrepoever

-This is the Best Repo Ever! (No joke. Really.) what!
+This is the Best Repo Ever! (No joke. Really.) what! again!
```

Undo a Previous Change

Sometimes we make small mistakes (or big ones!). Thankfully, Git comes equipped with commands that allow us to fix our mistakes. But beware, some commands that help us fix mistakes destructively modify the commit ID. Since these commit IDs are immutable, you could potentially cause issues for other collaborators. As a general rule, you should only use `git revert` if the commit has been pushed to the remote.

Undoing Changes

`git revert` creates a *new* commit with changes that are the opposite of the commit that is functionally being 'undone'.

For example, imagine within the commit history of a repository, there was a commit that changed all of your Heading 3s to Heading 5s. You could use `git revert` to change all of your headings back to Heading 3 instead of going back through every Heading 5 and changing them back to Heading 3s.

`git revert` can be used on commits at any point in the repository's history without affecting other work. Note, you can't use `git revert` to resolve conflicts—if there are more recent conflicting changes, Git will ask you to resolve as if it's a merge conflict.

Using `git revert` is a safe way to undo a specific change, preventing any of the typical complications that occur when altering the commit history of a project.

Amending Commits

Earlier, we discussed `git commit` and identified that commit messages are an important aspect of crafting a detailed project history.

Sometimes in the excitement of creating a commit, you might make a typo within the commit message. You can use `git commit --amend` to make a modification to the last commit you made. This will alter the commit history of your project, so it is recommended that you don't use `git commit --amend` if you have already pushed your commits to the remote.

You can also use `git commit --amend` to rewrite the most recent commit to include files in your staging area.

Rewind to an Earlier Point in History

We know programmers love to experiment and try new things. However, sometimes during the course of that experimentation we realize we went down the wrong path and the commits we were making might not be as useful as we originally thought.

Git has a `git reset` command that can help rewind the history of our project, but, it alters the commit history, which as mentioned before, might cause issues for other collaborators. It is highly recommended that you use `git reset` *only when you **have not** pushed your commits to your* *remote* branch. `git reset` comes in three distinct flavors, `--soft`, `--mixed`, and `--hard`.

- `git reset --soft` takes the identified commit(s) and places all of the changes in the staging area. This is helpful if you want to take a group of commits and squash them into a single larger commit.
- `git reset --mixed`, the default mode for `git reset`, takes the identified commit(s) and places all of the changes in the working directory. Like `--soft`, this is helpful if you want to take a group of small commits and combine some of the changes to make larger commits. But you can also use it to make additional changes to the files and then re-create the commit history.
- `git reset --hard` will take the identified commit(s) and destroy them. Be careful with this, because they don't go in your trash or recycle bin—the files essentially don't exist and are completely removed from your repository. Any uncommitted changes to files that are currently in the working directory or staging area will also be deleted. You can lose work with git reset `--hard`.

An example of using reset could look something like this:

`git reset --soft HEAD~2` would rewind the branch you are on by two commits (remember HEAD is a pointer to the tip of your branch). The changes that had been made in those last two commits would be reflected in the staging area.

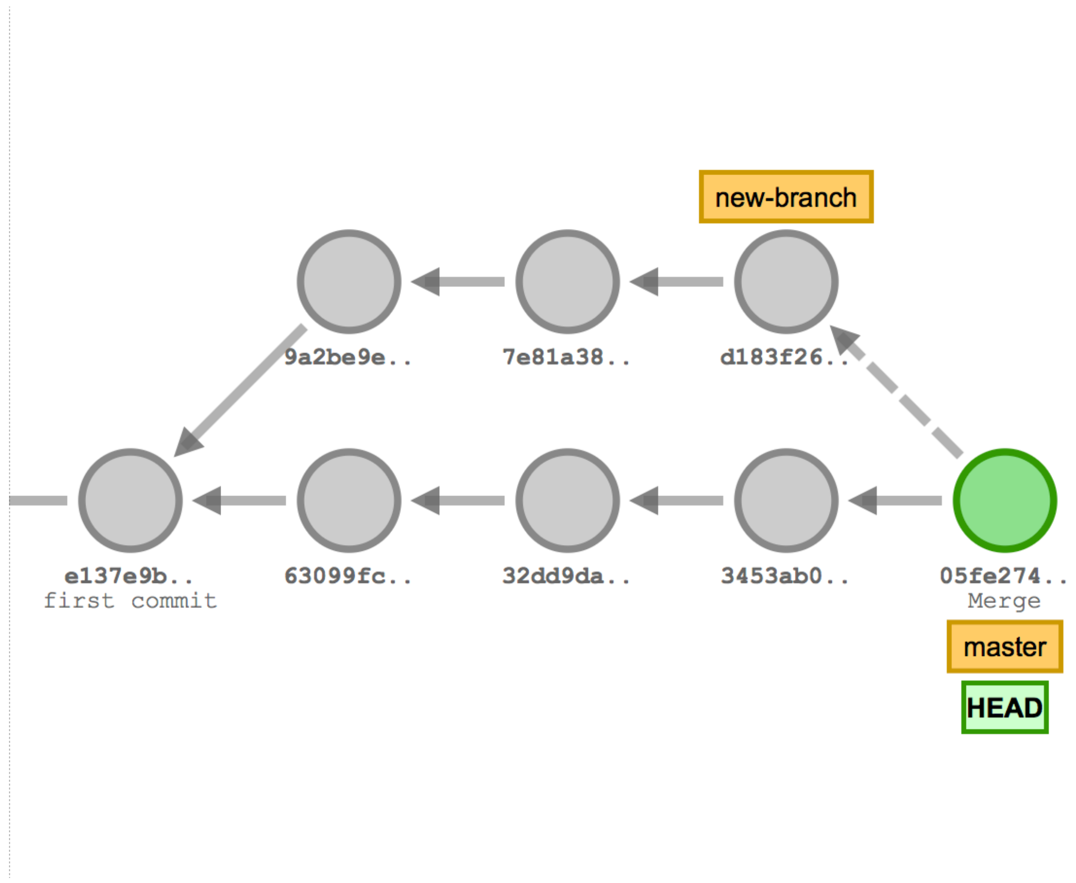
Git Merge Strategies

When it comes to merging, there are two primary methods that Git uses to apply your changes to the master branch (or whatever branch you are merging into): Recursive Merge and Fast-Forward Merge.

There is nothing right or wrong about either merge strategy, but it is important to know how each affects how history will look. At first glance these might seem less like merge strategies, and more like this is just how git handles merges—but we talk through that after we discuss the two merges.

Recursive Merge

A recursive merge occurs when your feature branch doesn't have the latest version of code in the branch you're trying to merge into (sometimes, but not always, master).

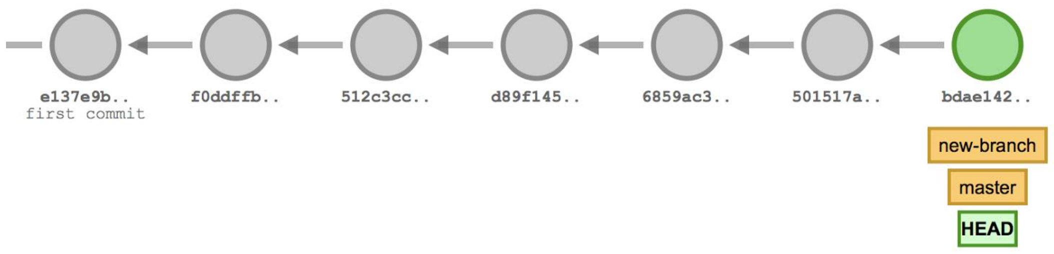


When you create a feature branch, you're basing it off the original branch at its current state. As you make changes to your branch, other collaborators might be merging their own changes into the original branch.

When you create a pull request and merge your changes, a merge commit is created. This takes the changes you made to your branch and the current state of the branch you're merging into and creates a new commit that combines those changes.

Fast-Forward Merge

A fast-forward merge occurs when there have been no new commits, other than the ones you're trying to merge, on the original branch since you created your feature branch from it.



Since the original branch doesn't have any changes, the tip of the branch is simply fast forwarded to include the changes on your branch. With a fast-forward merge, Git does not create a new merge commit.

Turn a Recursive Merge into a Fast-Forward Merge

The `git rebase` command is powerful and can do a lot of cool things in your repository (all of which rewrite your history, so use it with care). One of the most popular uses of `git rebase` is to create a fast-forward merge, when Git would have defaulted to a recursive merge.

Remember, a recursive merge happens when the branch you're merging into has new changes since you created your branch. Rebase picks up the commit you made on your branch and reapplies them after the last commit on the branch you select. You can also use `git rebase` to modify the way your commit history looks. Using the interactive modifier, `-i` with rebase allows you to edit previous commit messages, squash commits into a larger commit or commits, and rearrange your commit history.