"Git detached head" is a weird error message to receive...

Here’s the first thing you should know: *you haven’t done anything wrong*. Your repo isn’t broken or anything like that. The expression “Detached HEAD” might sound somewhat bizarre, but it’s a perfectly valid repository state in Git. Sure, it’s not the *normal* state, which would be—you’ve guessed it!—when HEAD is attached. The second thing you need to know is that going back to normal is super easy. If you just want to do that and get on with your day, go to the “How Do I Fix a Detached Head in Git?” section of this post to see how it’s done.

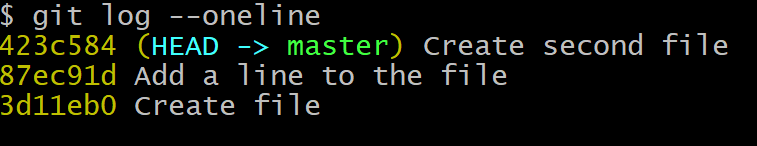
But if you want to know more—and I guess you do—stick around and we’ll help you. What does HEAD mean in Git? What does it mean for it to be attached or detached? These are the kind of questions we’ll answer in this post. By the end of it, you’ll have a better understanding of Git’s fundamentals, and detached HEADs will never trouble you again. Let’s dig in.

**Git Detached HEAD: Reproducing the “Problem”**

Let’s start with a quick demo showing how to reach the detached HEAD state. We’ll create a repository and add some commits to it:

mkdir git-head-demo  
cd git-head-demo  
git init  
touch file.txt  
git add .  
git commit -m "Create file"  
echo "Hello World!" > file.txt  
git commit -a -m "Add line to the file"  
echo "Second file" > file2.txt  
git add .  
git commit -m "Create second file"

With the commands above, we’ve created a new folder with a new repository inside it. Then we created a new empty file and committed that with the message “Create file.” Next, we added a line to that file and committed the change, with the message “Add a line to the file.” Finally, we’ve created another file with one line of text and committed that as well. If you use the **git log –oneline** command, you’ll see something like this:



Let’s say that, for testing purposes, we need to see how things were at the time of the second commit. How would we do that? As it turns out, we can check out a commit the same way we’d check out branches. Remember, branches are just names for commits. So, based on the example output above, we’d run**git checkout 87ec91d**. Keep in mind that if you’re following along by executing these commands on your own system, the hash for your commits will be different from those in the example. Use the log command to find it.

After running the command, we get the “You are in ‘detached HEAD’ state[…]” message. Before we go on, make sure you keep this in mind: you get to the detached HEAD state by checking out a commit directly.

**What Is a HEAD in Git?**

What does HEAD mean in Git? To understand that, we have to take a step back and talk fundamentals.

A Git repository is a collection of **objects** and **references**. Objects have relationships with each other, and references point to objects and to other references. The main objects in a Git repository are commits, but other objects include [blobs](https://developer.github.com/v3/git/blobs/) and [trees](https://developer.github.com/v3/git/trees/). The most important references in Git are [branches](https://rollout.io/blog/git-create-branch/), which you can think of as labels you put on a commit.

HEAD is another important type of reference. The purpose of HEAD is to keep track of the current point in a Git repo. In other words, HEAD answers the question, “Where am I right now?”

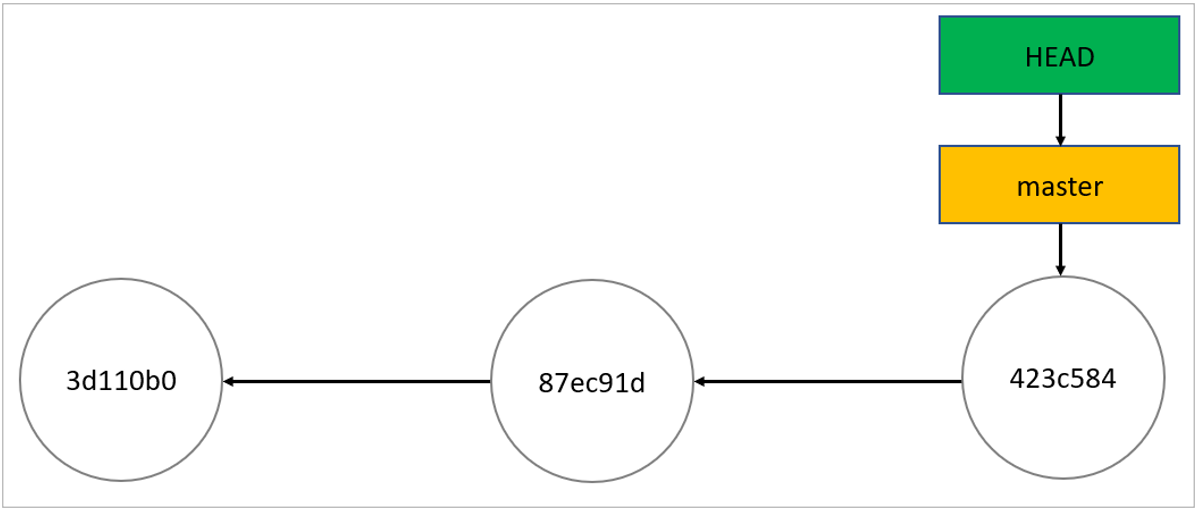
For instance, when you use the **log** command, how does Git know which commit it should start displaying results from? HEAD provides the answer. When you create a new commit, its parent is indicated by where HEAD currently points to.

Are you in ‘detached HEAD’ state?

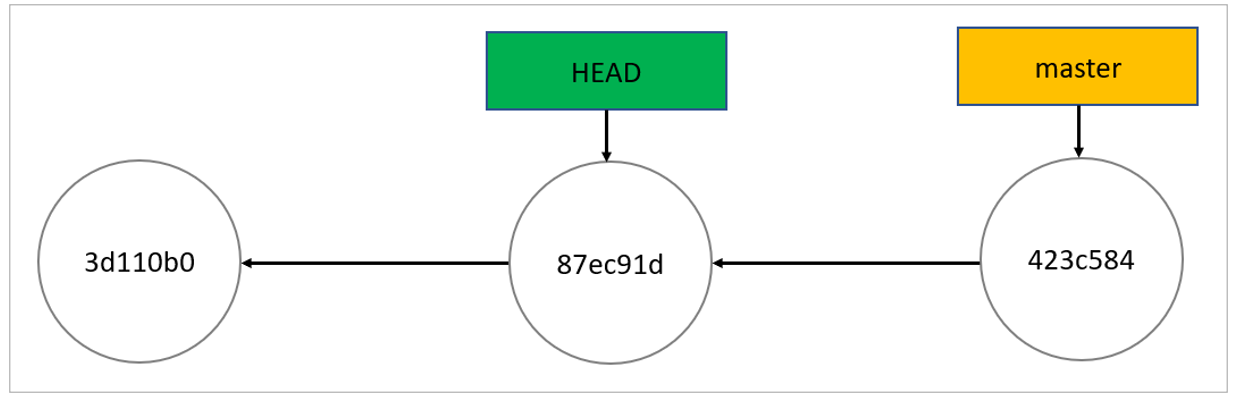
You’ve just seen that HEAD in Git is only the name of a reference that indicates the current point in a repository. So, what does it mean for it to be attached or detached?

Most of the time, HEAD points to a branch name. When you add a new commit, your branch reference is updated to point to it, but HEAD remains the same. When you change branches, HEAD is updated to point to the branch you’ve switched to. All of that means that, in these scenarios, HEAD is synonymous with “the last commit in the current branch.” This is the *normal* state, in which HEAD is *attached* to a branch.

A visual representation of our demo repository would look like this:



As you can see, HEAD points to the master branch, which points to the last commit. Everything looks perfect. After running **git checkout 87ec91d,** the repo looks like this:



This is the detached HEAD state; HEAD is pointing directly to a commit instead of a branch.

**Benefits of a Git Detached HEAD**

Are there good reasons for you to be in the detached HEAD state? You bet there are!

As you’ve seen, you detach the HEAD by checking out a commit. That’s already useful by itself since it allows you to go to a previous point in the project’s history. Let’s say you want to check if a given bug already existed last Tuesday. You can use the **log** command, filtering by date, to start the relevant commit hash. Then you can check out the commit and test the application, either by hand or by running your automated test suite.

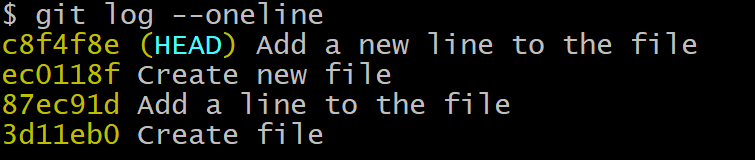
What if you could not only take a look at the past, but also change it? That’s what a detached HEAD allows you to do. Let’s review how the detached message starts:

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.

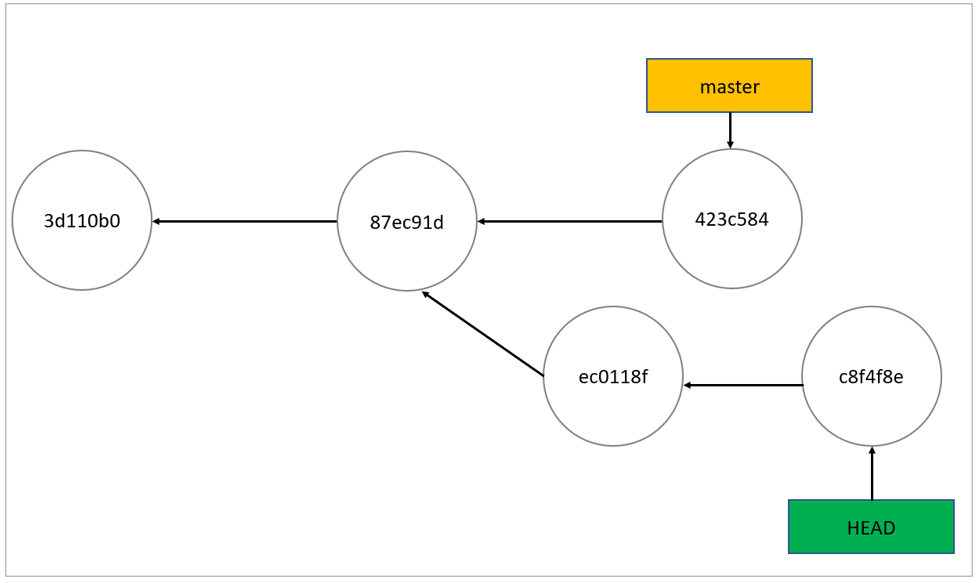
In this state, you can make experimental changes, effectively creating an alternate history. So, let’s create some additional commits in the detached HEAD state and see how our repo looks afterward:

echo "Welcome to the alternate timeline, Morty!" > new-file.txt  
git add .  
git commit -m "Create new file"  
echo "Another line" >> new-file.txt  
git commit -a -m "Add a new line to the file"

We now have two additional commits that descend from our second commit. Let’s run **git log –oneline** and see the result:



This is what the diagram looks like now:



What should you do if you want to keep those changes? And what should you do if you want to discard them? That’s what we’ll see next.

**How Do I Fix a Detached HEAD in Git?**

You can’t fix what isn’t broken. As I’ve said before, a detached HEAD is a valid state in Git. It’s not a problem. But you may still want to know how to get back to normal, and that depends on why you’re in this situation in the first place.

**Scenario #1: I’m Here by Accident**

If you’ve reached the detached HEAD state by accident—that is to say, you didn’t mean to check out a commit—going back is easy. Just check out the branch you were in before:

git checkout <branch-name>

If you’re using Git 2.23.0 or newer, you can also use **switch** instead of **checkout**:

git switch <branch-name>

**Scenario #2: I’ve Made Experimental Changes and I Want to Discard Them**

You’ve entered the detached HEAD state and made a few commits. The experiment went nowhere, and you’ll no longer work on it. What do you do? You just do the same as in the previous scenario: go back to your original branch. The changes you made while in the alternate timeline won’t have any impact on your current branch.

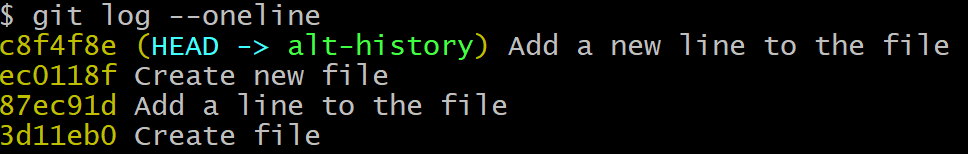
**Scenario #3: I’ve Made Experimental Changes and I Want to Keep Them**

If you want to keep changes made with a detached HEAD, just create a new branch and switch to it. You can create it right after arriving at a detached HEAD or after creating one or more commits. The result is the same. The only restriction is that you should do it before returning to your normal branch.

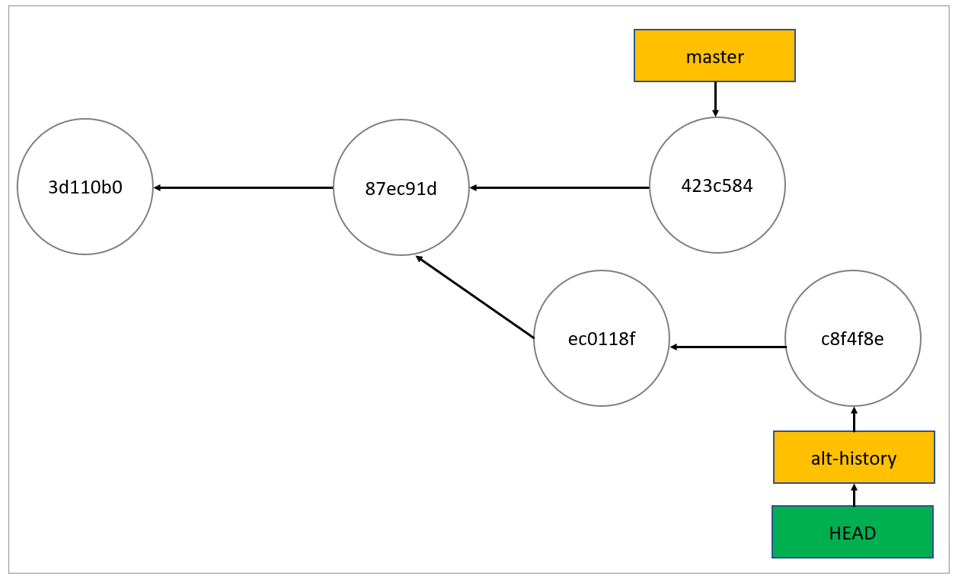
Let’s do it in our demo repo:

git branch alt-history  
git checkout alt-history

Notice how the result of **git log –oneline** is exactly the same as before (the only difference being the name of the branch indicated in the last commit):



We could just run **git branch alt-history**, and we’d be all set. That’s the new—and final—version of our diagram:



**Getting Rid of the “Git Detached HEAD” Message**

Before wrapping up, let’s share a final quick tip. Now that you understand everything about detached HEAD in Git and know that it’s not that big of a deal, seeing that message every time you check out a commit might become tiring. Fortunately, there’s a way to not see the warning anymore. Just run the following command:

git config advice.detached head false

Easy, right? You can also use the **–global**modifier if you want the change to apply to every repository and not only the current one.

**Git Detached HEAD: Less Scary Than It Sounds**

A message talking about heads not being attached doesn’t sound like your routine software error message, right? Well, it’s not an error message.

As you’ve seen in this post, a detached HEAD doesn’t mean something is wrong with your repo. **Detached HEAD** is just a less usual state your repository can be in. Aside from not being an error, it can actually be quite useful, allowing you to run experiments that you can then choose to keep or discard.

[*Carlos Schults*](http://carlosschults.net/)*is a .NET software developer with experience in both desktop and web development, and he’s now trying his hand at mobile. He has a passion for writing clean and concise code, and he’s interested in practices that help you improve app health, such as code review, automated testing, and continuous build.*