

[theserverside.com](https://www.theserverside.com)

A git reset hard example: An easy way to undo local commits and shift head

By Cameron McKenzie, TechTarget Published: 17 Jul 2018

4-5 minutes

[Manage](#) Learn to apply best practices and optimize your operations.

-
-

There are a number of compelling reasons as to why a developer needs a modern source code management tool like [Git](#). The ability to share code with other developers is one. The ability to isolate source code changes through the use of topic branches is another. But most of all, it's essential developers can use Git to roll back local commits and changes.

How to git reset local commits

This Git tutorial focuses on the capacity to roll back changes, undo a local commit and restore the [development environment](#) to an earlier and possibly more stable state. For traversing the commit history and rolling back to a previous state, you'll need the git reset hard command.

Git reset hard command example

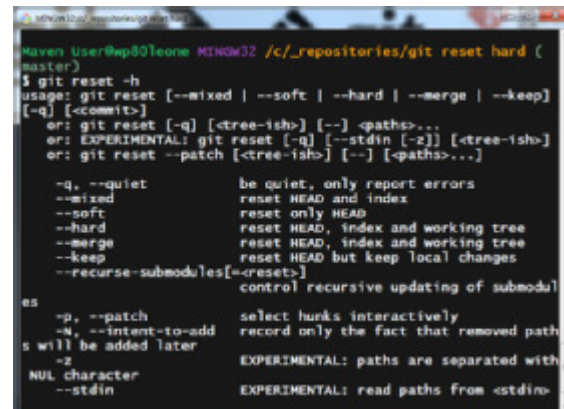
To demonstrate how the git reset hard command works, the first thing we need to do is initialize a new [Git repository](#).

```
/c/ git reset hard example
```

```
$ git init
```

Initialized empty Git repository in

```
C:/_repositories/git reset hard/.git/
```



```
Maven User@wp801eone MINGW32 /c/_repositories/git reset hard (
master)
$ git reset -h
usage: git reset [--mixed | --soft | --hard | --merge | --keep]
[-q] [<commit>]
or: git reset [-q] [<tree-ish>] [--] [<paths>...]
or: EXPERIMENTAL: git reset [-q] [--stdin [-z]] [<tree-ish>]
or: git reset --patch [<tree-ish>] [--] [<paths>...]

-q, --quiet                be quiet, only report errors
--mixed                    reset HEAD and index
--soft                     reset only HEAD
--hard                     reset HEAD, index and working tree
--merge                    reset HEAD, index and working tree
--keep                     reset HEAD but keep local changes
--recurse-submodules[=<reset>] control recursive updating of submodul
es
-p, --patch                select hunks interactively
-N, --intent-to-add         record only the fact that removed path
s will be added later
-z                          EXPERIMENTAL: paths are separated with
NUL character
--stdin                    EXPERIMENTAL: read paths from <stdin>
```

Here you'll find optional switches for the git reset command.

A rich git commit history

With the Git repository initialized, we need to create a bit of a local commit history in order to see the full power of the git reset hard command. To create that local commit history, simply create five HTML files using the touch command, and after each file is created, add the file to the Git index and issue a commit. The git bash commands for this are as follows:

```
/c/ git reset hard example
$ touch a.html
$ git add . & git commit -m "Commit #1 - 1 file"
$ touch b.html
$ git add . & git commit -m "Commit #2 - 2 files"
$ touch c.html
$ git add . & git commit -m "Commit #3 - 3 files"
$ touch d.html
$ git add . & git commit -m "Commit #4 - 4 files"
$ touch e.html
$ git add . & git commit -m "Commit #5 - 5 files"
```

After you've issued the series of touch and git commit commands, a

git reflog command will display the rich local commit history we have created.

```
/c/ git reset hard example (master)
$ git reflog
2e1dd0a (HEAD -> master) HEAD@{0}: Commit #5 - 5
files
868ca7e HEAD@{1}: commit: Commit #4 - 4 files
ebbbca3 HEAD@{2}: commit: Commit #3 - 3 files
882bf98 HEAD@{3}: commit: Commit #2 - 2 files
2f24f15 HEAD@{4}: commit (initial): Commit #1 - 1
file
```

The git reset hard command

Now let's imagine that everything after the third commit was bad, and we want to roll back the HEAD reference to the commit with an ID of *ebbbca3*. To do so, we would issue a `git reset --hard` command using that ID. By using the `--hard` switch, the git reset will not only change the HEAD reference, but it will also update all of the files in the index and the working directory as well. The full git reset hard command is as follows:

```
/c/ git reset hard example (master)
```

```
$ git reset --hard ebbbca3  
HEAD is now at ebbbca3 Commit #3 - 3 files
```

When we [inspect the working tree](#), we will see that there are now only three files, as files *d.html* and *e.html* have been removed.

```
/c/ git reset hard example (master)  
$ ls  
a.html  b.html  c.html
```

Git reset vs. revert and cherry-pick

Unlike the [git cherry-pick](#) and the [git revert](#) command, the git reset hard command does not create a new commit but actually rolls the HEAD reference right back to the ID of the local commit we used in the command. This isn't a matter of [updating the working tree](#), creating a new commit and moving the history forward. The git reset hard command actually points the HEAD right back to the old [commit](#), as though the original commits never happened.

There are other options to explore with the git reset command, including --soft, --hard and --merge. But when you're just starting with Git, the git reset --hard command is the right one to get familiar with. It's one of the first commands a developer should learn after they've mastered the [five basic git commands](#), as it represents an

easy way to roll back your local commit history, which is one of the primary reasons why developers use source code management tools in the first place.