Aa

Home iOS & Swift Books Advanced Git

3
Stashes Written by Jawwad Ahmad & Chris Belanger

After you've worked with Git for some time, you might start to feel a bit constrained. Sure, the staging, committing, branching and merging bits are all well and good, and these features undoubtedly support the nonlinear here-there-and-everywhere process of modern, iterative development. But the commit is still the atomic level of Git; there's nothing below it.

It can begin to feel like Git is forcing you into a workflow wherein you can't do anything outside of a commit, and that you have to limit yourself to building a working, documented commit each and every time you want to push your work to the repository. Talk about performance anxiety!

But Git recognizes that you can't always work to the level of a commit. Development is messy and unpredictable; you may go down a few parallel rabbit holes, chasing different possibilities, before finding the right solution for a bug. Or, quite commonly, you may be building out the next great feature for your app when, suddenly, you need to switch over to a bugfix branch and get a hotfix out the door in an hour. *Codus interruptus*, for sure.

So in situations like the above, what do you do with that unfinished and uncompilable code that isn't quite ready to be committed, but that you don't want to lose?

Well, if you're a paranoid coder like me, you may have developed a past habit of duplicating code directories that you want to keep for later, like so:

```
book_repo
├──git_book_before_edits
├──git_book_old
├──git_book_old_epub_test
├──git_book_older
└──git_book_version_with_jokes_removed
```

And if you bring that same thought process to working with Git, you'll often want to do a similar thing by creating "interim commits" in the branch you're working on, just so that you don't lose your work inside the confines of the Git repository:

```
| * b10bc04 It’s the finallllllllll countdoooooooooown do-do-do-dooooooooooo
| * 5f20836 Minor tweak
| * 36ff9ca fix lldb thing from errata forums
| * 0ce8469 Moved Mach-O section to DO_NOT_ADD_THIS… will keep it around for...
| * 79c465d removed TODO
| * cebc416 ok, time to wrap this crap up, Derek
| * 42d40dd wow that sucked
| * e7e8228 omg apple changing everything internally
| * 6d162de Partial Revert "woooooooooooooot done w/ code signing"
| * 9676642 woooooooooooooot done w/ code signing
| * 986e288 ok, hopefully no more bugs on that script….
| * da5e21f Fix xattr bug
| * 52a2bc7 dsresign really looking good now
| * 6382f56 more tweaks
| * 66f0041 dsresign close to complete?
| * ba0bcf9 Added dsresign
| * 5ac42d4 meh
| * a5f08df Pre-delete a big section
| * 1d86a71 wow…. much words. Time to delete some content
.
.
.
```

(With apologies to the wonderful debugging guru Derek Selander at https://github.com/DerekSelander. Love ya, Derek.)

You can see how making interim commits to "save" your work can quickly get out of hand. It's clear that there *must* be a better way to quickly *stash* work in progress and retrieve it later, when you're done coding that duct-tape hotfix and are ready to return to refactoring the mess of code you left on a feature branch.

In fact, Git provides this feature out of the box, and it's called, unsurprisingly, `git stash`.

Introducing git stash

Let's return to your tightly knit team of Will, Xanthe, Yasmin and Zach. Things have been running smoothly for today, and you've been able to get some time to work on that all-important README file.

Note: You'll need to have completed the challenge in the previous chapter to follow along. If you haven't you can use the starter project from this chapter's materials folder.

In the **magicSquareJS** repo, open **README.md** and add a bit of text to the bottom of the file:

```
## Contributing

To contribute to this project, simply
```

But before you can complete that thought, Xanthe pings you. "Can you take a quick look at the `xUtils` branch?" she asks. And, being the responsive, agile team member you are, you agree to have a look.

But you don't want to lose that work you've done. Granted, it's not a lot, but it does capture the state of your thoughts at that moment, and you don't really want to redo that work. So save your work to **README.md** and exit out of your text editor.

Back at your command prompt, switch to the `xUtils` branch:

```
git checkout xUtils
```

Git detects that you have changes in your working tree that haven't been captured anywhere, not at least as far as Git's concerned:

```
error: Your local changes to the following files would be overwritten by checkout:
        README.md
Please commit your changes or stash them before you switch branches.
Aborting
```

Again, Git nudges you in the right direction, here: Either stash your changes or commit them before you move on. And although your reflex might be to commit them — because they'd be *safe*, you know — stashing your changes is a much better option.

How stashing works

In a manner that's quite similar to the way you created all of those "backup" directories of your code projects in the past, Git lets you take your current set of changes in your working tree and stash them in your local repository, to be retrieved later.
To see this in action, simply execute the following command:

```
git stash
```

Git will respond with a somewhat cryptic but reassuring message:

```
Saved working directory and index state WIP on main: f368651 Merge branch 'xReadmeUpdates'
```

It appears that Git has saved your current set of changes somewhere... but *where*? To see that, you'll dig into the internals of Git once again.
From the command prompt, navigate into the **.git/refs** directory:

```
cd .git/refs/
```

Inside that directory, you'll find a file named, simply, **stash**. Print the contents of that file out to the command line with the following command to inspect its contents:

```
cat stash
```

In my case, that file holds a single line, which is simply an object hash:

```
e05a7bf99cf63b3f6d63ea510af29e1d7d8e0e8e
```

Just as you did before, you're going to use this object hash to look up the metadata and associated content of this object. Execute the following command to get the details about this object, substituting the actual value of your own hash for mine:

```
git cat-file -p e05a7bf
```

Git tells me the following:

```
tree a757817e26b9efeb2db5ec03b251f945724b0e82
parent f368651693e57ad2df3615bb47a5fb1701e3b0e7
parent a7db95a364966ee2d8bd28b11bcf4c50b76964ad
author Jawwad Ahmad <jawwad.f.ahmad@gmail.com> 1629521044 -0500
committer Jawwad Ahmad <jawwad.f.ahmad@gmail.com> 1629521044 -0500

WIP on main: f368651 Merge branch 'xReadmeUpdates'
```

Well, that looks suspiciously like a commit. And, in fact, Git uses the same basic mechanism to stash your changes as it does to commit them. But instead of treating this as a commit, Git tracks this as a stash object.
I'll dive further into this stash, so feel free to try this out with your own metadata and hashes. I'll look up the `tree` hash of my stash with the following:

```
git cat-file -p a757817
```

Git shows me the snapshot of my working tree at the moment I stashed it:

```
100644 blob 7b378be30685f019b5aa49dfbdd6a1c67001d73c    .tools-version
100644 blob 28c0f4fd0553ffb10b0ead9cd9584a4d251b61c8    IGNORE_ME
100644 blob 9d47666971a2b201db4d89f0536d5766af389c7c    LICENSE
100644 blob f178fa5fb806db3aeab852dabbc8a71075162bde    README.md
100644 blob feab599b6be0efb9d20ef20e749b3b8e70e8c69f    SECRETS
040000 tree d0a7cf32f8ee481267d545000ca99dc532ef0579    css
040000 tree 29a422c19251aeaeb907175e9b3219a9bed6c616    img
100644 blob 0f28461df255e1159fc27c249b10eb924e77ffd6    index.html
040000 tree 3876f897b04b07c02dcbe321ad267b97d1db532f    js
```

And then I'll display the actual contents of the **README.md** file with the following:

```
git cat-file -p f178fa5
```

Git shows me the contents of that file, with my most recent change at the bottom (output truncated for brevity):

```
.
.
.
This project is maintained by teamWYXZ:
- Will
- Yasmin
- Xanthe
- Zack

## Contact Info

For info on this project, please contact [Xanthe](mailto:xanthe@example.com).

## Contributing

To contribute to this project, simply
```

So, just as if I'd committed this file to the repo, I have a snapshot of my working tree at a particular point in time, but held as a stash, instead of as a commit.
That's enough playing around inside the Git internals; head back to the root of your project folder with the following command:

```
cd ../..
```

Now that you understand a little about how Git stores your stash, you can move on to retrieving what you've stashed.

Retrieving stashes

You've stashed your changes so that you can check out Xanthe's `xUtils` branch, so continue to do that now:

```
git checkout xUtils
```

Git doesn't complain this time, as it sees that you've created a stash of your current working tree and there are no unstashed or uncommitted changes.
For the purposes of this exercise, you'll assume that you've looked through Xanthe's changes and given her some advice on what she should do. With that task out of the way, you can now return to your changes that you'd like to complete on the `main` branch.
Switch back to `main` with the following command. The `-` just means switch back to the branch I was previously on.

```
git checkout -
```

And take a look at the contents of **README.md** with the following:

```
cat README.md
```

Look at the end of the file, and you'll see that your changes aren't there. That makes sense, since Git switches you back to whatever the current state of `main` is on your local system. But how do you find your stashed changes and get them back into your working tree, since it's already been a long day and there's no *way* you're going to remember the hash of the stash you created earlier?

Listing stashes

Git lets you create more than one stash, and it keeps them in a stack so you can easily find the one you want to apply. But, first, you'll create another stash to illustrate this.
Imagine you don't think you need that **SECRETS** file lying around anymore, but you want to test this later to make sure you *really* don't need it. Delete that file from your working tree with the following:

```
rm SECRETS
```

And now create another stash, with the same command you used before:

```
git stash
```

Git gives you the same message as before; note that there's no mention from Git of how to identify your most recent stash.

```
Saved working directory and index state WIP on main: f368651 Merge branch 'xReadmeUpdates'
```

Note: Since you're working with a stash, it seems that Git should let you use common stack operations, and, in fact, it does. The `git stash` command is a convenience alias for `git stash push`, for quickly creating a non-named stash on the stack. You also have access to common stack operators such as `pop`, `show` and `list`, as you'll see in the following sections.
To see the stack of stashes, use the `list` option on `git stash`:

```
git stash list
```

Git shows you all of the stashes it knows about:

```
stash@{0}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
stash@{1}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
```

Since this is a stack, the stash at index 0, denoted by the `stash@{0}` label, is the most recent, with the stash at index 1, being older as it's farther down the stack.
Now, the default stash message really isn't that descriptive; if you only work with one stash at a time, that might be sufficient. But you won't always remember what the most recent stash is, if they all have the same stash message. But just as you can provide a message when you create a commit, you can specify a message when you create a stash.

Adding messages to stashes

Make another change in your project working tree, and create a temporary file with the following chained commands:

```
mkdir temp && touch temp/.keep && git add .
```

Note: The `&&` operator in bash or Zsh lets you chain commands and execute them in order, but only if the preceding command succeeded (that is, had an exit code of `0`). So, in this case, you're trying to create a directory named **temp**; if that succeeds, you then create a **.keep** file in the **temp** directory so that Git recognizes this directory. If the file creation was successful, then you call `git add .` to stage this file so that Git can track it.
Now that you have a tracked addition to your working tree, you can create a stash with an appropriate message to help you keep track of what's what. Execute the following command:

```
git stash push -m "Created temp directory"
```

In this case, you've used the `push` operator, since `git stash` alone doesn't let you supply any arguments. Now, pull up the stash stack again with **`git stash list`** to see what the stack looks like now:

```
stash@{0}: On main: Created temp directory
stash@{1}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
stash@{2}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
```

That's a little more instructive, but your memory is short. If you can't recall what exactly you did in a particular stash, you can peek at the contents of that stack entry with the following command:

```
git stash show stash@{0}
```

Git tells you briefly what the changes are in this stashed snapshot:

```
 temp/.keep | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
```

In fact, you can use most of the options from `git log` on `git stash show`, since Git is simply using its own log to show the changes contained in your snapshot. You can check out the diff, or patch, of your very first stash using the `-p` option as you would with `git log`:

```
git stash show -p stash@{2}
```

Git then shows you the entire patch of your stash. Here's mine:

```
diff --git a/README.md b/README.md
index 6c6eee6..f178fa5 100644
--- a/README.md
+++ b/README.md
@@ -19,3 +19,7 @@ This project is maintained by teamWYXZ:
 ## Contact Info

 For info on this project, please contact [Xanthe](mailto:xanthe@example.com).
+
+## Contributing
+
+To contribute to this project, simply
```

Here, you can see the final four lines added to the end of **README.md**. Speaking of which, you should probably finish those updates to the README file before you lose your creative inspiration.

Popping stashes

Now, you want to get back to the state represented by the stash at the bottom of your stack: `stash@{2}`. If you remember your stack operations from your data structures and algorithms courses, you know that you'd generally `pop` something off the top of this list. And in fact, you can do this with Git, using `git stash pop` to remove the top stash from the stack and apply that patch to your working environment.
In this case, however, the stash you want isn't at the top; rather, it's at the bottom of the stack. And you've decided that you don't really want to keep those other two stashes around, either.
Git lets you cherry-pick a particular stash out of the stack and apply it to your working tree with the `git stash apply stash@{number}` where number is zero indexed. Since you want to apply the stash at the bottom of the stack which is the third item you'd use `stash@{2}` as the argument to the `git stash apply` command.
Run the following command to apply the third item in the stack:

```
git stash apply stash@{2}
```

Git then reverts the state of your working tree to the snapshot captured in the stash. In this case, your stash doesn't have the **temp** directory, and it still has the **SECRETS** file, as you deleted that file *after* you created the stash. Pulling up a simple directory listing with `ls -la` will show you this:

```
drwxr-xr-x@ 12 jawwad  staff    384 Aug 21 00:00 .
drwxr-xr-x+ 73 jawwad  staff   2336 Aug 21 00:01 ..
drwxr-xr-x@ 18 jawwad  staff    576 Aug 21 00:00 .git
-rw-r--r--   1 jawwad  staff      5 Aug 15 20:41 .tools-version
-rw-r--r--   1 jawwad  staff     43 Aug 15 20:41 IGNORE_ME
-rw-r--r--@  1 jawwad  staff  11343 Aug 15 09:08 LICENSE
-rw-r--r--   1 jawwad  staff    539 Aug 20 23:59 README.md
-rw-r--r--   1 jawwad  staff     65 Aug 20 23:51 SECRETS
drwxr-xr-x@  5 jawwad  staff    160 Aug 20 23:50 css
drwxr-xr-x@  3 jawwad  staff     96 Aug 15 09:08 img
-rw-r--r--   1 jawwad  staff   1255 Aug 20 23:50 index.html
drwxr-xr-x@  7 jawwad  staff    224 Aug 20 23:50 js
```

Now, execute `git status` to see how Git interprets the situation, and it notes that you have one change not staged for commit:

```
On branch main
Your branch is ahead of 'origin/main' by 17 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

If you think about it, this makes sense; you didn't stage that change, so Git's snapshot of the working tree in your stash also represents Git's *tracking* status of files, whether they were staged or unstaged. And when you apply a stash, you overwrite not just the working tree, but also the staging area of your repository. Remember: You staged your addition of the **temp/.keep** file *after* you created the stash you just applied.

Applying stashes

Applying a stash doesn't remove it from the stack; you can see this if you execute `git stash list`:

```
stash@{0}: On main: Created temp directory
stash@{1}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
stash@{2}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
```

`stash@{2}` is still sitting at the bottom of the stack; Git simply applied the patch resulting from this change and left that stash on the stack. In just a bit, you'll see how to remove elements from this stack.
Open up **README.md** in an editor and finish off that excellent line of documentation that you spent all night dreaming up:

```
## Contributing

To contribute to this project, simply create a pull request on this repository and we’ll review it when we can.
```

Save your changes, and exit the editor. Now, you can stage and commit your manifesto, again using the chaining option of `&&` in bash to do it all on one line:

```
git add . && git commit -m "Updates readme"
```

Note: You can also use `git commit -am "Updates readme"` to add and commit in a single command. One minor difference is that the `git commit -am` version won't commit newly added files.
#lazygit for the win, kids! Oh, but wait. You really did want to add that **temp/.keep** file after all. Fortunately, that change was stashed at the top of the stash stack, so you can use the `pop` operator to simultaneously apply the patch of that stash and remove it from the top of the stack:

```
git stash pop
```

Git gives you a nice status message, combining the output you might expect from `git status` along with a little note at the end telling you which stash was popped from the stack:

```
On branch main
Your branch is ahead of 'origin/main' by 18 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   temp/.keep
```

```
Dropped refs/stash@{0} (a75edcdeb09e2f2c5096fca4ecb278097a310d7a)
```

You've applied that patch on top of your working tree, and because Git stashed the tracking information as well as the state of the files in your working tree, those changes have already been staged for commit.
On second thought you don't need to commit that change just yet so you'll remove it for now. Run the following commands to unstage and remove the file:

```
git reset temp/.keep
rm temp/.keep
```

Clearing your stash stack

Now, you might think you're done here, but you still have a few stashes hanging around that you don't really need. Run **`git stash list`** to see what's still hanging around in the stash stack:

```
stash@{0}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
stash@{1}: WIP on main: f368651 Merge branch 'xReadmeUpdates'
```

You can see that Git popped the top of the stack (the `On main: Created temp directory` entry), so it's gone. But you can easily get rid of all entries with the `clear` option.
In your case, you don't want any of the stash entries. So execute the following command to clear the entire stash stack:

```
git stash clear
```

Execute `git stash list` now, and you'll see that there are no more stashes hanging around.

Merge conflicts with stashes

Since applying stashes looks a *lot* like merging commits from Git's perspective, you may be wondering if you can also get conflicts when merging a stash. Well, maybe you weren't wondering this, but I'll bet you are now!
The answer is yes — you can certainly experience merge conflicts when working with stashes. You'll set up a scenario now that will show you how to resolve a merge conflict with a stash.
Recall that a merge conflict occurs when Git detects a change to a common subsequence inside a file. In your case, the best way to illustrate this is to alter the same line in the README file between a branch and a stash.
You're currently on `main`, so create a small edit to **README.md** to keep this author happy with proper use of English. Change the shorthand term "info" to "information" as shown below:

```
For information on this project, please contact [Xanthe](mailto:xanthe@example.com).
```

Save your work, exit the editor, and add this change as a stash using the following command:

```
git stash
```

Now, switch to the `xUtils` branch, where you'll create a conflicting change:

```
git checkout xUtils
```

Open **README.md** in that branch and add the following line, immediately after the `# Maintainers` section:

```
## Contact Info

For info on this project, please contact [Xanthe](mailto:xanthe@example.com) or [Will](mailto:will@example.com).
```

Here, you've added Will as a contributor, and you'll notice that this line still has the shorthand "info" in use.
Save your work and exit the editor. Now, stage and commit those changes — remember, you can't merge into a "dirty" or uncommitted branch:

```
git commit -am "Added Will as a contributor"
```

Now attempt to apply the stash with the following command:

```
git stash pop
```

Git responds with a message noting the conflict:

```
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
The stash entry is kept in case you need it again.
```

Open **README.md** in an editor to edit the conflict. You'll notice that your stash also added the Contributing section that you just added to `main`. However on this branch you only want to change the word **info** to **information** so you'll remove the Contributing section from the changes and you'll edit the conflict so that the final section looks like the following:

```
## Contact Info

For information on this project, please contact [Xanthe](mailto:xanthe@example.com) or [Will](mailto:will@example.com).
```

Save your changes and exit the editor. Now you can commit those changes to the `xUtils` branch:

```
git commit -am "Corrected language usage"
```

Challenge: Clean up the remaining stash

Is there anything left to do? Execute `git stash list` and you'll see there's still a stash there. But you popped that stash, didn't you? Of course you did.
Your challenge is twofold, and may require a little dive into the Git documentation:

. Explain why there's still a stash in the stack, even though you executed `git stash pop`.

. Remove the remaining entry from the stash without using `git stash pop` or `git stash clear`. There's one more way to remove entries from the stack with `git stash` — what is it?

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

Stashing lets you store your in-progress work and retrieve it later, without committing it to a branch.
`git stash` takes a snapshot of the current state of your local changes and saves it as a stash.
Git stores stashes as a stack; all successive stashes are pushed on the top of the stack.
Git stashes work much like commits do, and it is captured inside the .git subdirectory.
`git stash` is a convenience alias for `git stash push`.
`git stash list` shows you the stack of stashes you've made.
`git stash show <stashname>` reveals a brief summary of the changes inside a particular stash.
`git stash show -p <stashname>` shows the patch, or diff, of the changes in a particular stash.
`git stash apply <stashname>` merges the patch of a stash to your working environment.
`git stash pop` pops the top stash off of the stack and merges it with your working environment.
Merging a stash can definitely result in conflicts, if there are committed changes that touch the same piece of work. Resolve these conflicts in the same way as you would when merging two branches.

Where to go from here?

Stashes are an excellent Git mechanism that help make your life as a developer just a little bit easier. And what's nice is that, again, Git follows common workflows that come naturally to most developers, such as stashing not-quite-done-yet changes off to the side (which is an improvement over those terrible copies of directories I used to make).
The next two chapters deal with one of the more useful, yet widely misunderstood actions in Git: rebasing. There are mixed opinions out there as to whether merging or rebasing is a better strategy in Git. To help you make sense of the arguments on both sides, you'll look at rebasing in depth so you can form your own opinions about what makes the most sense for your team and your workflow.

4. Demystifying Rebasing 2. Merge Conflicts

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.

© 2022 Razeware LLC