Git Cheat Sheet: Commands and Best Practices

In this post, we’re looking at the most successful source code management tool available today, Git. With every great tool, there is a CLI which compliments all the great features and options, which leads to a vast number of things you need to remember and be expected to recall within a keystrokes notice. If like me, your memory is as lethargic as an asthmatic ant carrying heavy shopping, you might just need a hand — or in this case, a Git cheat sheet!

Today we'll walk through some Git basics, including Git commands like pull, push, and fetch. At the end of the article, you can download a one-page Git cheat sheet rich with the Git commands of champions, the gems that make your SCM a pleasure to work with, the… ok, enough’s enough, let’s get down to business.

Using Git Init and Git Clone

Well, we’ll start at the beginning of course! In the beginning, Git created init and clone. Both of which give you a repository in which you can manage your source code. The difference being of course that you would use the **init** command to create a repository from scratch, whereas you’d use **clone** to literally clone, or copy an existing repository into the directory you ran the command from. Once this is done, we’re able to start our workflow! Here are the two examples in action on the ZipRebel project:

```
$ git init myProject
Initialized empty Git repository in /Users/sjmaple/myProject/.git/
```

And as a clone:

```
$ git clone https://github.com/zeroturnaround/ziprebel.git
Cloning into 'ziprebel'...
remote: Counting objects: 94, done.
remote: Total 94 (delta 0), reused 0 (delta 0), pack-reused 94
Unpacking objects: 100% (94/94), done.
Checking connectivity... done.
```

Git Branch Example

If we were to create a new feature for example, we should consider creating a new branch for all of our feature code. This is an important best practice in my opinion as keeping your new feature code separate from other changes in your master allows you to easily switch back and forth between branches without cross-polluting your code. So how do we create our branches? Well that bit is easy! Simply use the **branch** command passing in the name you'd like to call it, as follows:

```
$ git branch feature/unzip
```

In this case we've been very sensible with our naming so it's clear to read from the branch name that we're creating the unzip feature for our ZipRebel project. Although there was no feedback from this command, we can view all our branches by using the same command, but without a parameter:

```
$ git branch
  feature/unzip
* master
```

Using the Git Checkout Command

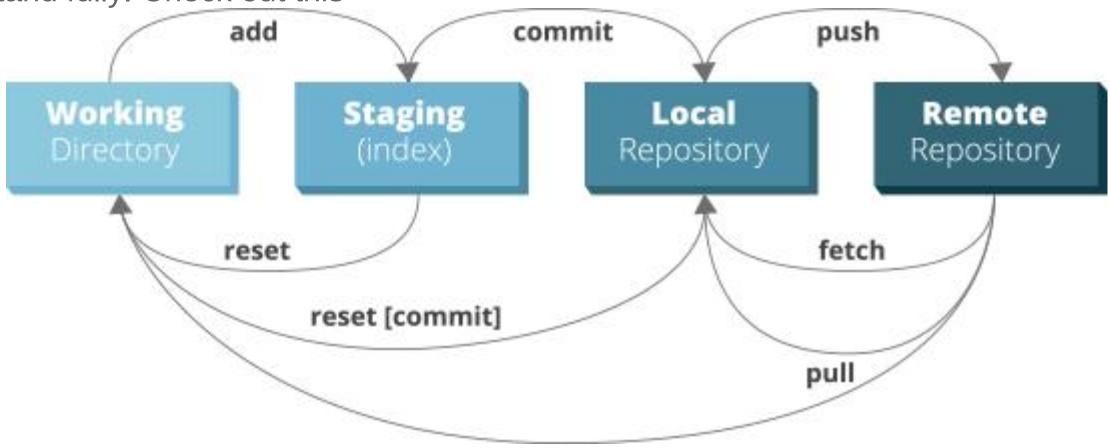
We can see a couple of things. First of all, we see our new branch has been created, and secondly, our active branch we're working on is *master*, rather than the new branch. Incidentally, you can view all branches, including remote branches by adding a couple of flags: git branch **-av**. We want to jump into the *feature/unzip* branch to begin coding our new feature, so to switch branches we'll use the checkout command:

```
$ git checkout feature/unzip
Switched to branch 'feature/unzip'
```

This will switch our branch and update our working directory should the code bases be different. As we've only just created this new branch of course, there are no updates.

Git Add Example

Let's make a change to our code -- first we'll… whooaooaaaaa there! We're going to use terminology here which we'll need to explain and point at a state diagram to understand fully. Check out this image:



Pretty, isn't it. Files in our working directory that contains all our files can be edited and changed at any time, but that does not change where they sit in the diagram above until you perform a git command on them.

Using Git Status

To see the state of all the files in your project, use the following command which will soon be ingrained in your mind:

```
$ git status
On branch feature/unzip
nothing to commit, working directory clean
```

Let's work on the flow of making a couple of changes and see how this updates our status message. Simply by editing a couple of files and not performing any other command other than a git status gives us the following:

```
$ git status
On branch feature/unzip
Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git checkout -- ..." to discard changes in working directory)

        modified:   src/main/java/org/zeroturnaround/ziprebel/ZipRebel.java
        modified:   src/test/java/org/zeroturnaround/ziprebel/ZipRebelTest.java

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

This show us we've modified a couple of files but haven't added them to staging (note that staging is also referred to as the index too) ready for our next commit.

Adding Files to Staging in Git

We can add our files to the staging, or stage our files, by using the git **add** command by referencing each file individually, groups of files or adding a wildcard which adds all modified files. We'll do the latter now:

```
$ git add .
```

...to the sound of silence! However running the git status command will show us it succeeded:

```
$ git status
On branch feature/unzip
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

        modified:   src/main/java/org/zeroturnaround/ziprebel/ZipRebel.java
        modified:   src/test/java/org/zeroturnaround/ziprebel/ZipRebelTest.java
```

Now we see that git recognizes the modified files as the changes to be committed -- a good example of Git version control hard at work. Feels good to be on the same page as your version control system, doesn't it? You'll see the **HEAD** token used in the output here. The HEAD is a pointer or reference to the current branch.

Committing Changes to Local Repository in Git

At this point we could choose to unstage our file by running git **reset** on a file which keeps changes to that file, make further changes that we can stage, or commit our changes to the local repository. Let's do the latter:

```
$ git commit -m "Added unzip capability"
[feature/unzip 05db2cc] Added unzip capability
 2 files changed, 2 insertions(+), 2 deletions(-)
```

We could have done both the add and commit in the same command if we knew we're making a small change and we know there's nothing else to commit, by running the following command while we had modified files that were not yet in staging:

```
$ git commit -am "Added unzip capability"
[feature/unzip 8c10510] Added unzip capability
 2 files changed, 2 insertions(+), 2 deletions(-)
```

The **-a** flag will add all tracked files from the working directory into staging before performing the commit. (Note that newly created files in your working directory are untracked until you manually add them). But wait, what does a commit even mean? Well that's a great question!

What is Git Commit?

> A Git commit records a snapshot of your changes, marks your name, email address and commit message giving you a commit id.

In our case above, we were given ids *05db2cc* and *8c10510* for our example commits. These can be used later to troubleshoot, blame or pinpoint moments in time of code. Talking of blame, `git blame` is a fun command that annotates each line in a file with information about who was last to change each line of code and in which commit etc. This could output a fair bit of text, so consider using a line range **-L** flag to limit the output to something manageable.

Git Diff Example

When a file can exist in multiple places with different content, it's important to understand what has changed between the places. For example, I might want to know what the differences are between my workspace file and my staging file. Well a number of options are available. In fact the diff command is an extremely useful command with many use cases. We'll cover a few here.

First of all let's consider you want to compare all changes between your working directory and your staging area. That's easy! Just call `git diff`! If you want to compare your staging area with your local repository, you can call `git diff --cached`. Equally simple. You can even compare files across a couple of commits, by using their commit ids, like this: `git diff 05db2cc 8c10510`. It's a powerful tool, so make sure you use `git diff --help` to view all possibilites to find what's right for your use case.

Merging Feature Branch Changes

Finally, once you've completed your feature development in your new branch, you'll need to merge it back into the main branch. So first of all go to the branch you want to merge your feature branch into. In this case we'll run `git checkout master`. Next we call the **merge** command passing in the branch which we want to merge into our active branch. So we run `git merge feature/unzip`.

All being well, our code will be automatically merged and we don't have to worry about a single thing! However, life being the pain in the proverbial that it is, our merge might have a conflict that we have to solve! The git status command will help you understand where the conflict exists. If you open the file, you'll notice some markers: `<<<<<<<`, `>>>>>>>`, and `=======`, that surround the lines that conflict, including the changes that each branch is trying achieve.

Reconciling Code Versions

Now we're at a fork in the road. What should we do? Well obviously you write the best code, so you could delete the opposing conflict code keeping your own followed by a new commit into the main branch. Alternatively, although rather unlikely, this other developer wrote better code than you so you might choose their code above your's before committing.

More likely however, you'll need to create a mutant hybrid of the two changes and check in Frankenstein's monster code into the main branch. In any case you will need to fix the file, and then add and commit the change to enjoy the results of your masterful conflict resolution skills.

Using Git Pull, Push, and Fetch

So far all of our changes, including our repository changes have been local. We haven't even touched our remote repository or mentioned it even, since our `git clone` all the way up back at the start of our post. At some stage we'll need to sync up our local and remote repositories and we have a few advanced Git commands that can help us achieve this: **pull**, **push** and **fetch**.

Let’s start with what happens if we have an out of date local repository. This can happen if others update the remote repository with new features, bug fixes and code. We want to grab these changes, in fact I wanted to say pull or fetch, but didn’t want to reuse the command names, and update our local repository with the latest and greatest code.

Git Fetch Example

The least intrusive way of doing this is by using the fetch command, which grabs all the latest commits from the remote repository and imports them into a remote branch. Note here that it doesn’t add them into a local branch in your workspace. Note that you can view remote branches with `git branch -av` as mentioned previously. If could merge manually from here if you chose to.

Git Pull Example

Alternatively, instead of using the fetch command, you could call git pull, which performs a git fetch followed by a git merge under the covers. Another trick you can use is `git pull --rebase`, which performs a `git fetch` followed by a `git rebase`. What’s a **rebase** I hear you cry? It’s a mechanism that allows you to apply your local commits on top of the incoming commits, rather than to have two chains in parallel that need to be merged.

Using Git Push

Lastly, the **push** command is pretty much the opposite of a git fetch. We’re simply transferring commits from our local repository to our remote repository. There’s always the possibility some files can be overwritten here, so be careful!