Version Control with Git

Overview

This guide will take you through exercises to gain familiarity with the basics of the `git` version control system. The guide can be followed on any system which has a reasonably modern version of git is installed; no external servers are required.

Step 1: Getting started with a Git repo

Create and inspect a git repo

We will work out of our `$HOME` directory throughout this guide. None of this guide should require `root` access.

To start, we need to create a directory that we will "initialize" as a git repository.

```
$ cd $HOME
$ mkdir git-practice
$ cd git-practice
```

We should now be in the `git-practice` directory. We can turn this into a git repo with one simple command:

```
$ git init
Initialized empty Git repository in /Users/lowell/git-practice/.git/
```

If we look at the contents of the directory, we will see that we have a directory named `.git`.

```
$ ls -a
.   ..   .git
```

This `.git` directory contains the various metadata that will allow git to track changes for files in our `git-practice` directory. Let's take a look inside of it:

```
$ ls -1F .git
HEAD
config
description
hooks/
info/
objects/
refs/
```

Most of these files/directories are only used for internal tracking for git. `config` contains the local configuration for the repo. `config` can be manipulated by hand, but most of its settings can be changed with git commands. `hooks` contains a set of scripts that will automatically be run when certain actions take place, e.g. a commit. `hooks` can be a very powerful tool, but we won't be setting them up here.

`git status` is one of the most common and useful git commands. It's a good idea to run this often to see the current state of our directory.

```
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Setup global config

Before we make any changes to our repository, we'll want to set up some global configuration for our user. Specifically, we'll want to set up our name and our email address to be tied to commits and commit messages. In the following, you should substitute your own information.

```
$ git config --global user.name "J. Lowell Wofford"
$ git config --global user.email lowell@lanl.gov
```

We can view our global configuration:

```
$ git config --global --list
user.name=J. Lowell Wofford
```

```
user.email=lowell@lanl.gov
```

There is are a lot of configuration settings available. We can list all of the current settings with:

```
$ git config --list
user.name=J. Lowell Wofford
user.email=lowell@lanl.gov
core.editor=vim
merge.tool=vimdiff
Js-MacBook-Pro:git-clone lowell$ git config --list
credential.helper=osxkeychain
user.name=J. Lowell Wofford
user.email=lowell@lanl.gov
core.editor=vim
merge.tool=vimdiff
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
...
```

Add a file and make your first commit

To start version controlling files, we'll need a file to work with:

```
$ echo "Imma Textfile" > README.md
```

Let's check our status now that we've modified the directory.

```
$ git status
...
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README.md

nothing added to commit but untracked files present (use "git add" to
track)
```

It suggests we should add our `README.md` file. Adding it will move it from being an "untracked" to a "tracked" file. Let's follow the suggestion:

```
$ git add README.md
```

Check the status again:

```
$ git status
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   README.md
```

We have no added the file. This both set it to tracking, and also staged it for the next commit. Let's go ahead and make our first commit:

```
$ git commit -m "added README"
[master (root-commit) 3519411] added README
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

The `-m` option specifies a message to associate with the commit. It's a very good idea to give commit messages that are descriptive of what has changed in the directory.

Inspect and make a change

Let's see what our status is now:

```
$ git status
On branch master
nothing to commit, working tree clean
```

We no longer have any changes, staged or unstaged: "working tree clean".

We can look at information about our past commits with `git log`:

```
$ git log
commit 3519411040534adfd10c059058fe3e715da17dda (HEAD -> master)
Author: J. Lowell Wofford <lowell@lanl.gov>
Date:   Tue Mar 5 08:30:07 2019 -0700

    added README
```

Git log tells us our commit message, who made the change and when it was made. With other options, we can see more and less information. Notice the commit has a hash starting with 351941... This hash is the unique identifier for the commit.

Now, let's modify our `README.md`.

```
$ echo "...with append" >> README.md
```

Check the status:

```
$ git status
On branch master
Changes not staged for commit:...
    modified:   README.md
no changes added to commit (use "git add" and/or "git commit -a")
```

This is different than our message before. Since `README.md` is a tracked file, it reports as modified, but ***not*** staged for commit. We are also offered the `git commit -a` option now.

We can see how our current state differes from the previous commit:

```
$ git diff
...
--- a/README.md
+++ b/README.md
@@ -1 +1,2 @@
 Imma Textfile
+...with append
```

Let's commit our change to `README.md`. We can use the `-a` flag now to auto-add all modified, tracked files (note: it will ***not*** add untracked files).

```
$ git commit -a -m "update README"
[master 9ed49d5] update README
 1 file changed, 1 insertion(+)
```

`git commit -a` is handy, but can also be dangerous. You can easily accidentally commit changes you didn't intend. It's a good idea to only use this if you've carefully looked at `git status` and `git diff` to know what changes you're about to commit.

We now should have two log messages to commit. Here's a handy, compact version of the log:

```
$ git log --pretty=oneline
4c3a21ef50d10259e9accb2a1c18c1c49ab7c91a (HEAD -> master) update README
b8bbcab814f9f39e852f54911c3470b274a8084b added README
```

This `oneline` format is especially handy for locating the hash for a known commit. We are going to want to inspect the original commit, so we can grab its hash here. Fortunately, we don't need the whole hash. Git will work if we just pass enough of the hash to uniquely identify it (minimum 5 characters).

Let's `checkout` our old commit:

```
$ git checkout b8bbc
Note: checking out 'b8bbc'.
You are in 'detached HEAD' state...
```

Our branch should now be in the state of the old commit. The "detached HEAD" warning means that the current "HEAD" (current working directory) does not refer to the latest commit of any known branch, so you can't directly make commits to it (what would it be appending the commit to?).

Double check that we are in the old version:

```
$ cat README.md
Imma Textfile
```

We *do* have the expected old contents of the file. We can now revert back to the latest commit:

```
$ git checkout master
...Switched to branch 'master'
```

Then check the contents again:

```
$ cat README.md
Imma Textfile
...with append
```

We're back where we should be. We used the `diff` command before to compare a current working directory to the previous commit. We can also compare to any arbitrary commit by hash:

```
$ git diff b8bbc
...@@ -1 +1,2 @@
 Imma Textfile
+...with append
```

Step 2: Working with remotes

We have seen so far how git can work with files in a local directory. In real-world scenarios, we are interested in collaborating through git. Git offers a number of tools for collaboration, but one of the necessary tools is the

ability to synchronize similar repos across multiple locations. This is achieved with ***remotes***.

Bare repo, remote add & push

We are going to want to make a second directory to work with. We'll call this one `git-bare`. Let's create it:

```
$ cd $HOME
$ mkdir git-bare
$ cd git-bare
```

We want to `git init` this repo, but we want a location that does not necessarily have its own working copy (HEAD) but is instead an exchange point for other synchronized working copies. This is called a ***bare*** repo.

```
$ git init --bare
Initialized empty Git repository in /Users/lowell/dev/git-bare/
```

If we list the contents:

```
$ ls
HEAD  description  info  refs
config  hooks  objects
```

We see that the contents of this ***bare*** repo look like the contents of `.git` in a ***non-bare*** repo. That's because it is not expected to have a working copy, so it just tracks all of the internal data.

We'll go back to our original repo:

```
$ cd ../git-practice
```

Now, we can add our new `git-bare` repo as a `remote` for `git-practice`.

```
$ git remote add origin ../git-bare
```

This means setup the repository at `../git-bare` to be a repository I know how to synchronize with. Give it the name "origin".

Once we have a remote, we can `push` (synchronize our changes to another remote) our changes to the remote. Because we haven't done this before, our current repo is not tracking anything in "origin". We can both push and setup tracking with:

```

```

```
$ git push --set-upstream origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 234 bytes | 234.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ../git-bare
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Clone, modify, push

To further explore remotes, we will need a third repo. This time, instead of creating a blank repo and initializing it, we want to start with a new repo that is a complete clone of the existing repo. To do this:

```
$ cd $HOME
$ git clone git-bare git-clone
Cloning into 'git-clone'...
done.
```

This made a copy of `git-bare` into a new directory, `git-clone`. Let's inspect it:

```
$ cd $HOME/git-clone
$ cat README.md
Imma Textfile
...with append
```

Now, in our clone, let's create a new file.

```
$ echo "something" > new.txt
```

Let's add it and commit it:

```
$ git add new.txt
$ git commit -m "added something new.txt"
...
```

Checking status:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

Because we got this copy by a clone, we are already tracking the "origin" master. The status tells us that we have changes that "origin" doesn't know about it, and suggests we push them.

Pushing:

```
$ git push
...
To /Users/lowell/dev/git-bare
   f524f51..ac9ad59  master -> master
nothing to commit, working tree clean
```

Note that because we were already tracking, we did not need to provide any arguments to push.

Fetch & pull

Now, let's go back to the repo where we started:

```
$ cd $HOME/git-practice
```

If we look at the status here:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'...
```

We see that nothing appears to have changed. But wait! We know we made a change. What's happening?

Before we can see the change, we need to `fetch` remote changes. This tells git to download the appropriate metadata from the remote(s) to tell us about them, but not touch our working directory.

```
$ git fetch
...Unpacking objects: 100% (3/3), done.
From ../git-bare
   f524f51..ac9ad59  master     -> origin/master
```

This message lets us know that something has changed with "origin/master". Let's get more detail with status:

```
$ git status
...Your branch is behind 'origin/master' by 1 commit, and can be fast-
```

```
forwarded.
  (use "git pull" to update your local branch)...
```

Now we see that the "origin" has a commit we ***don't*** yet have. We would like to synchronize the changes from the remote to us. This is called a `pull`:

```
$ git pull
Updating f524f51..ac9ad59
Fast-forward
 new.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new.txt
```

This message tells us that one line was added to a file called `new.txt`. Let's take a look:

```
$ cat new.txt
something
```

We got the change! To recap: we made the change in `git-clone`, we then pushed it to `git-bare`, and pulled it into `git-practice`. This is a common workflow. Many collaborators can push their changes to a bare repository and others can pull them down as they want them.

Step 3: Some other useful git commands

There are a couple of commands that may prove helpful as you work with git, and they have been included here.

Reset (soft)

Let's make sure we're in our `git-practice` repo:

```
$ cd $HOME/git-practice
```

Suppose we did the following:

```
$ touch tmp
$ git add .
```

If we look at `git status`:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
```

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   tmp
```

Wait, we don't want to commit `tmp`. It's just a temporary file. But, we've now staged it for commit.

Enter `git reset`. In it's default mode it performs a "soft" reset that just moves files that have been added to the staging area out of the staging area.

```
$ git reset
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    tmp

nothing added to commit but untracked files present (use "git add" to
track)
```

We can see that `tmp` has gone back to being an unstaged, untracked file.

Stash

We're still sitting in `git-practice`, and we have this untracked `tmp` file. Suppose someone comes along and asks us to work on something else. We're not ready to commit `tmp`, but we don't want to lose what we have there.

We can temporarily set aside changes with the `stash` command:

```
$ git add tmp
$ git stash
Saved working directory and index state WIP on master: ac9ad59 added
new.txt
```

If we look in our directory now:

```
$ ls
README.md  new.txt
```

We see that `tmp` is no longer there.

We can see what stashes we have sitting around (we can have multiples) with:

```
$ git stash list
stash@{0}: WIP on master: ac9ad59 added new.txt
```

We see that we have one stash. It also tells us that it's a Work In Progress (WIP) on a particular commit, `ac9ad59`. The `{0}` tells us that it is identified as stash `0`.

We can see this particular stash with:

```
$ git stash show 0
 tmp | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
```

Ok, so this stash adds the zero-line file, `tmp`.

Let's restore our stash:

```
$ git stash pop 0
$ ls
README.md  new.txt  tmp
```

We can see that it has restored our `tmp` file. Note that the `0` here was not strictly necessary. By default, `stash pop` will pop the `0`th stash, but it is necessary for any other stash.

Conclusion

Git as a lot of features that we won't deal with in this guide. For instance, git's branch/merge/rebase workflows are perhaps one of its most distinguishing features, but they are too complicated for this short guide.

Command Reference

- `git init [--bare]` - Initialize a new git repo in a directory
- `git status` - Print the current status of the local repo
- `git config` - Get/set global and local configuration
- `git add` - Add files to the staging area
- `git commit [-a] -m <message>` - Commit staged changes
- `git log [--pretty=<fmt>]` - Print commit history
- `git diff [commit]` - Create a diff between two points
- `git checkout <commit>` - Switch to a specific commit
- `git remote add <name> <url>` - Add a remote to the local config
- `git push [--set-upstream ...]` - Push to upstream (setting it if needed)
- `git fetch [remote]` - Fetch status of a remote
- `git pull` - Pull from upstream
- `git reset` - Remove files from staging area

- `git stash [list|show|pop]` - Manipulate stashes