sitepoint

GIT
FUNDAMENTALS

BY THOM PARKIN

GIT CONTROL OF YOUR VERSION MANAGEMENT

GIT FUNDAMENTALS

BY **THOM PARKIN**

Git Fundamentals

by Thom Parkin

Peer reviewer: Ralph Mason
Peer reviewer: Nuria Zuazo
Peer reviewer: Steve Browning
Peer reviewer: Matt Parkin
Editor: Linda Jenkinson
English Editor: Paul Fitzpatrick
Cover Designer: Alex Walker

License

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com
Printed and bound in the United States of America

[2] http://www.gnu.org/licenses/#FDL

About Thom Parkin

Thom has been writing software since the days when all phones were wired. He calls himself a ParaHacker. Seduced by Rails and then enthralled with Ruby, Thom was an early adopter of RubyMotion. When he is not playing board games, you will find Thom on the SitePoint Forums[1] where he is a Team Leader.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

[1] http://www.sitepoint.com/forums/

Table of Contents

Chapter 1

An Introduction to Git

Synopsis

Git[1] is a popular, free **Version Control System**, designed for use by software developers. The essence of Version Control is the ability to capture the state of software documents, and in doing so make it possible to revert to a previous state if required. The safety net that this provides aids collaboration, and encourages freedom to experiment.

So, if you're not a software developer, why would you care about Git? Can you recall a time when, working on something important, you found yourself wishing you could undo something? With Version Control you can, in effect, rewind time. By capturing snapshots of your progress, any stage in a project can be revisited and modified quite freely. You even have the ability to cherry pick certain changed files and merge them with others from different points along your project's timeline.

One of the things that makes Git special is that it's local—in other words, based on your own computer. As such, you don't need any special servers or complex

[1] http://git-scm.com/

equipment to use it. Since Git is file-based, it works with anything you can create or manipulate on a computer—text documents, image files, audio or video—and you can easily share or transport your work to other computers, even as an email attachment. When collaborating, Git enables all members of a team to work independently, and easily blend together the best of each contributor's efforts into a final product.

Traditionally, working on a project with multiple people can quickly become a nightmare of "change management." You make changes, then your colleague does likewise, and while another colleague is updating the document you suddenly remember some important facts that need to be added. So as not to forget, you make changes to *your* copy, but now there are multiple versions of the document, all with different amendments and no easy way to consolidate them.

Software developers face precisely this dilemma all the time—not to mention designers, content writers, video artists, web developers, and so on. And the system(s) that are used to help with this are called **Source Code Management** (SCM) systems. An SCM system helps to resolve conflicts among files and conflicts within a file as a result of multiple people making changes to the same file. Git is one of the most popular SCM systems, and it's available to anyone, at no cost.

If you're a photographer or digital artist, you can save your work at any state and freely experiment. If you're a programmer, you can save a program before you make big changes to improve performance. For someone who generates documents (a teacher who creates curricula, for example) the ability to effectively rewind time is immensely valuable. Similarly, if you've ever had a very large document that represents hundreds of hours of work and the computer suddenly can't read the file, a system like Git provides integrated backup, without the tedium of making multiple copies under different names.

Regardless of the type of development or design work you do—front-end UI development or back-end coding—managing the change and evolution of your working projects is both important and challenging. With Git, you'll be able to see all your changes, and even compare the differences between versions. Projects that use a Git repository can be stored on a site like Github or, because Git is file-based, you can even host it in a Dropbox folder.

Git is not Github!

Git is not to be confused with Github[2]! Although the names are similar, Github is a very popular website that hosts version control repositories (primarily Git repositories).

Why Read This Guide?

The goal of this guide is to:

- introduce you to Git
- overcome the intimidating perception that Git is very “geeky” and confusing for non-programmers
- fill a gap in almost all other resources on the web for learning (and learning about) Git

Among those who use Git on a regular basis, there is a sort of "evangelism" movement to help lead others to understand this great, free tool. That is the purpose of this guide. It's designed as a simple tutorial; an opportunity for you to “get your feet wet” in the Git world. You are not expected to become a Git expert as a result of reading this. You will not be tested on your “geek smarts.”

After reading this you should be able to:

- decide if Git is a tool that can help your workflow
- feel much more comfortable approaching and using Git (or Github)

What to Expect

In order to understand how Git can help you, we'll take you on a tour. The tour includes the Git commands and workflow that most web developers, designers, and content specialists might use. We'll use a simple example project, "Halo Whirled," to show you how you might use Git. The "Halo Whirled" walk-through uses only the basic Git commands.

[2] https://github.com/

Like any tour, we ask that you trust the directions will not lead you into any trouble. Don't be intimidated by typing commands directly in the command line. Don't worry about response(s) from your computer that we have not explicitly described. There is nothing in this set of examples that can cause any permanent harm to your computer.

Although Git was originally designed, and is most often used, for software source control, every attempt has been made here to be very generic in the discussion. You should see how these functions and features can apply to any work you do—especially if you find yourself in an environment of constant change.

Setting up Git

The first step is to download and install Git. Git is entirely free, and can run on machines using Windows, Mac and Linux operating systems. Based on your operating system, the specific process to install Git will vary slightly. Rather than repeat the installation instructions that are already widely available on the web, we will instead point you to some some existing installation guides that we recommend:

- to install Git on a Mac, go to http://alvinalexander.com/mac-os-x/how-installgit-mac-os-x-osx
- to install Git on Windows, go to http://msysgit.github.com/
- many distributions of Linux come with Git pre-installed, but if that's not the case for you, you can download it from http://git-scm.com/download/linux

The Command Line

The most common way to use Git is via the "command line". The command line is a simple program that you find on all computers. It allows you to write instructions for the computer in a very basic way, without pretty graphics, buttons and so on. You just write text commands. This can be a bit scary for a beginner, but using the command line is definitely the best way to use Git.

There are "graphic user interface" (GUI) programs available as an alternative—programs that provide you with buttons to press and pretty interfaces—but we encour-

age you at least to learn the basics of Git using the command line. All the same, you can find a useful list of GUI options at the official Git website.[3]

The History is Important

The key concept in understand Git is the idea of **history**. Git records snapshots of the collection of files in your project. Our goal when using Git is to capture snapshots of our work. Rather than just having copies of the files in use, we can review the progression of work, reassemble it, rearrange it and repeat it as desired

It will be easier to grasp the use of Git if you keep in mind this concept of history, as most of the tasks you do in Git are related to moving through the history of your work.

[3] http://git-scm.com/downloads/guis

Chapter 2

Using Git

Let's Git Started!

In order to use Git it's first necessary to set up your project. This could be as simple as a single-file CV, or as complex as a complete website and all its subfolders.

Let's use Git to set up our Halo Whirled example project. First of all, open the command line:

- Using Windows, go to the Start Menu and choose Run. If that doesn't work for you, you can also type **CMD** in the search box. It's a little dependent upon which version of Windows you're using, but in almost all cases you can select Run and type **CMD** in the box.
- Using Linux, you can right-click the desktop or search the menus for words "Linux terminal" or "Linux console"—distros vary how exactly they label this.
- Using Mac, you can find the Terminal program in the **Utilities** folder within **Applications**. If you don't see it there, open Spotlight (the magnifying glass in the top-right corner of your screen) and search for "Terminal."

So now you have your command line open, it's time to begin using Git. However, before Git will allow you to interact with it on any project, you must set up two very simple configuration options: You will identify yourself within Git by adding your name and an email address. These can even be fabricated values if you prefer, such as "A User" and "User@email.com."

At the command line, type the following:

```
git config --global user.name "Your Name Here"

git config --global user.email "your_email@example.com"
```

Once you've entered the required commands, you'll need to press **Enter** (**Return** on a Mac) to run them.

Now that you and Git have become acquainted, let's start by creating a project directory (folder) on your system, and call it **Halo Whirled**. You can create this folder anywhere you like. Let's say, for example, that there's a folder on your computer called **Documents**, and that's where you want to store this test project. To do that, type the following in the command line:

```
cd Documents
```

You should see a clear indication that you are now in the **Documents** folder. Depending upon your operating system, you may also see other things, such as a dollar sign. For now, though, it's time to create a folder for our project. To do this, enter:

```
mkdir "Halo Whirled"
```

The quotes are only really necessary in Windows, but they don't hurt in Linux or Mac. Commands like **`mkdir`** and **`cd`** are often shorthand for English instructions. For example, **`mkdir`** means "make directory", while **`cd`** means "change directory."

Now that we have created the **Halo Whirled** folder, go into that folder by typing:

```
cd "Halo Whirled"
```

Now we are inside our new project folder. We haven't used Git at all until now. So let's tell Git to get to work in this folder. Each command to Git consist of two words, and the first is always **`git`**. To get Git going in our new project, we simply need to enter:

```
git init
```

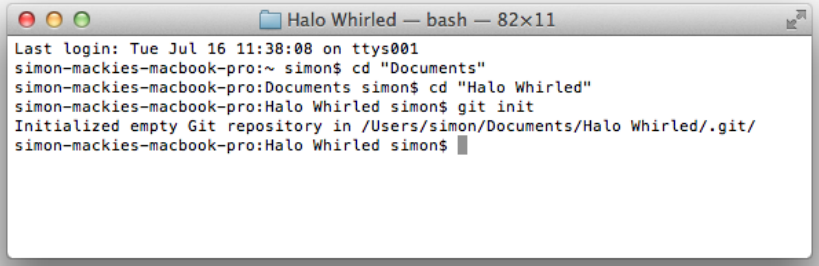

Figure 2.1. Git is initialized

You will get back a message that verifies Git is on the job and ready to start working for you with this project, similar to that shown in Figure 2.1 (your results will vary a little, depending on what operating system you're using). We've already set up a new project, and Git is ready to help us manage it! That wasn't hard, was it?

The Commit—Setting Your Work in Stone

The **commit** is fundamental to Git and source control, and is the most common action you will take using it. A commit is a snapshot in time, and represents a reproducible state of your project. The various commits you make during a project constitute the project's history.

Let's add some files to our project, and then look at how to tell Git which files you want it to keep track of. We will create a text file called **TheFirstFileUnderGitSourceControl.txt**. In Linux and Mac this is as easy as typing:

```
touch TheFirstFileUnderGitSourceControl.txt
```

If you're using Windows, you can simply create a TXT document in Notepad and then save it.

Now let's add some content to this file. For now, we'll do this in the typical way—by navigating on your computer to the **Documents** > **Halo Whirled** folder. Open the new

TheFirstFileUnderGitSourceControl.txt file in a text editor and type the year you were born. On a new line, type your favorite flavor of ice cream, for example:

```
I was born in 1842.

My favorite flavor of ice cream is vanilla.
```

Once you're done, save the file. Now let's return to the command line tool and tell Git to start tracking the history of this new file. Type the command:

```
git add TheFirstFileUnderGitSourceControl.txt
```

Don't expect any feedback at this stage. Git has happily accepted your command and executed it instantly. Git now knows that you are going to track the progress of this file.

Let's capture the current state of this file for all time by performing our first commit. Enter the following:

```
git commit -am "Initial Commit"
```

A commit has at least two parts: the file(s) to be included and a commit "message." The message is just a handy label (one or more tags, a description, or whatever) that we add to help us when searching through a long list of commits in the history. We'll be doing that shortly.

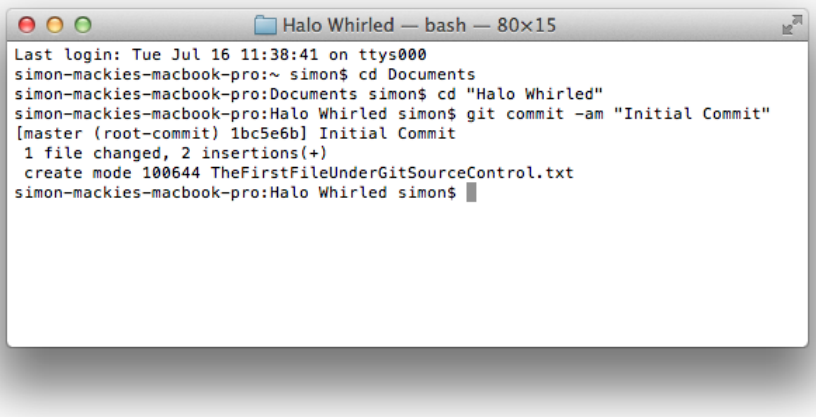

Figure 2.2. Our initial commit

The result of the commit command will be a little spurt of text, similar to that shown in Figure 2.2. If you look carefully, you'll see that it shows you what was saved in the commit. Now the fun begins. Once again, open **TheFirstFileUnderGitSourceControl.txt** file in your text editor. Delete the text you typed earlier and add some new text. It doesn't matter what it is. Save and close it, then, back in your command line tool, type this:

```
git commit -am "Changed all the text"
```

You have just begun to develop a history in Git. The command above has committed the new changes to this project's history. This example of using Git has involved only one file, but this process applies equally to any number of files and/or folders in your project.

Checkout—Stepping Back in Time

Let's say we now want to use Git to access a previous state of our project. To do so, we can use the `checkout` command. In the command line, type the following:

```
git checkout HEAD^
```

Don't be concerned by all the scary messages Git throws at you at this point. The world is not about to end!

The `checkout` command is not one you'll use like this very often, but it's handy for us at this point. To see the effect of `checkout`, open the **TheFirstFileUnderGitSourceControl.txt** file in your text editor again. You'll see that it has returned to its previous state.

It's important to note here that we haven't deleted any of our project's history by running the `checkout` command. The commits you make to your history are permanent. Indeed, there's nothing you can do in Git to make them disappear.

It's also worth noting that using `checkout` in this way isn't how we'd normally return to a project's earlier state. It's just a little cheat we're employing here to keep things simple.

When we made our commits earlier, you might recall that we added messages to them to make them easier to find. Below, you'll learn a better way to access your project's various states. Before that, though, let's make it a bit more "real world."

Go to your **Halo Whirled** project folder and place a new web page file inside, calling it **index.html**. (Use your preferred text editor to create this file.) While you're at it, create a new folder called `images` inside the **Halo Whirled** folder. Now go back to your command line tool to commit these changes by typing:

```
git add index.html

git add images/

git commit -am "A Simple Web Project"
```

Our little project is now the beginnings of a simple web site. Next we'll explore how to view the history of what we've done so far.

Log—Reviewing the History of Your Project

When you want to see the entire history of your project, and locate a particular point to check out, simply issue the command:

```
git log
```

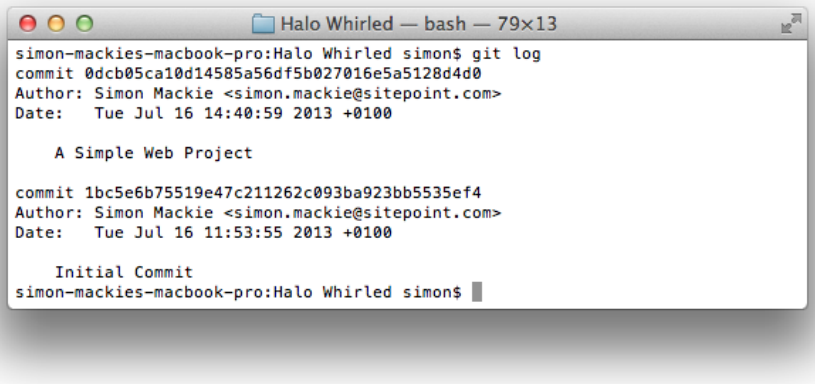

Figure 2.3. The results of running `git log`

The results of running this command should be similar to Figure 2.3. You may notice that each entry—regardless of the commit message you created for it—has a unique signature, called a **hash**, which looks something like this:

```
c178771270d4
```

Your hashes will be differant than the ones shown above. Each hash is generated automatically by Git, and each one uniquely identifies a particular commit. You can use these signatures to revert to the various states in the project's history.

Try it out using our current project. Although the actual hash you use will be different, the command will look something like this:

```
git checkout c178771270d4
```

Whenever you issue the `checkout` command, the set of files that are contained in that commit are restored to whatever state they were at the time of the commit. This means any files you currently have with the same name as those in that commit will be completely overwritten.

Using Hashes

You don't actually need to type out the full (very long!) hash. Git is smart enough to work out which commit you mean if you only type out the first few characters, as long as that short string is unambiguous (in other words, no other commit also starts with those characters). The first few characters will often suffice, and 8 characters is normally more than enough.

Status

When you're in the process of making changes, and committing them to save various states, it's easy to lose track of what has been updated, changed or committed. Git has a solution this kind of version blindness: a useful command—and arguably the second most used in Git—called `status`. It enables you to see what changes have been made and whether they've been committed to the Git repository. Try this:

```
git status
```

You should see output similar to Figure 2.4.

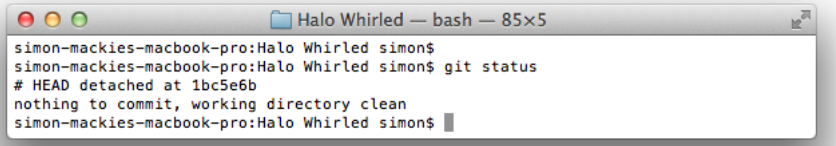

Figure 2.4. `git status` output

Now modify one of the files that's being tracked by Git. Then issue the `git status` command a second time. Make another commit, and check the status again. You should see something similar to Figure 2.5.

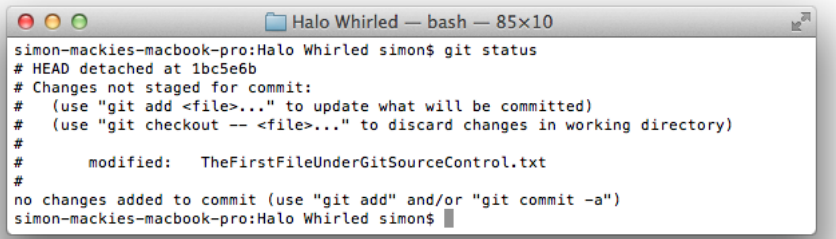

Figure 2.5. Updated output from `git status`

Finally, add another file to your project (you could create a new text file, such as **README**), and check the status one more time. Next, add the file to Git for tracking with:

```
git add <filename>
```

And, yes, you guessed it, check the status again:

```
git status
```

You should see something similar to Figure 2.6.

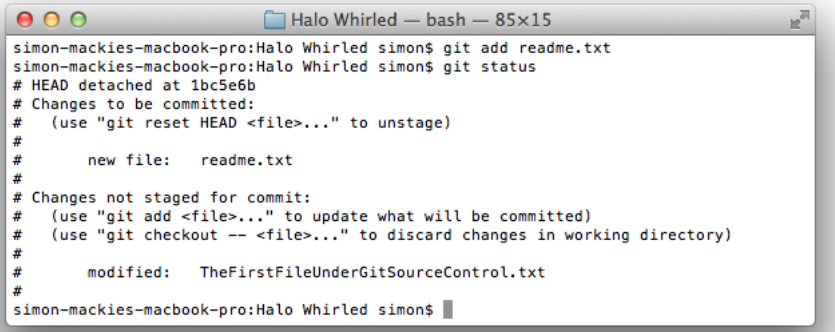

Figure 2.6. More updates from `git status`

As you can see, this command allows you to find out what changes have been made and committed and so maintain your bearings on what Git believes it's tracking.

Comparing Changes

By this stage your original file, **TheFirstFileUnderGitSourceControl.txt**, should have gone through many changes—all of which have been stored in the Git history. Make sure you have committed any recent changes. Double-check there are no outstanding changes with the `status` command:

```
git status
```

It should say "`nothing to commit, working directory clean.`" if there are no outstanding changes. Take a look at your history to locate the hash of the very first commit you made:

```
git log
```

Let's assume that the entry looks like this (yours will vary just a little):

```
commit c178771270d4ebf3f59f33201658f2a20d60eb01

Author: Thom Parkin <Thom.Parkin@Websembly.com>

Date: Sat Jan 26 17:18:23 2013 -0500
```

```
Initial Commit
```

The command to check out that “Initial Commit” would be the following (but don't do it now):

```
git checkout c178771270d
```

Instead, try this command:

```
git diff c178771270d
```

The `diff` command is intended to show you the differences between two commits. Specifically, it'll show exactly what changed in each and every file. Notice the plus (+) and minus (-) symbols. They indicate which files and which lines within each file were added and removed, respectively. This can be valuable information. Git provides the capability to retrieve those individual changes *en masse* or selectively. That's where some of the more complex commands (and the concept of merging, which we'll introduce a little later) would apply.

Branches

Often, in software development, a new feature or an attempt to solve a problem may lead the code development process off on a tangent. This concept, known as **branching,** is so common it's part of the fundamental design philosophy of Git. Assuming you have no uncommitted changes in your project (create a new commit if necessary) try this command:

```
git checkout -b trial_by_fire
```

This creates a new branch representing the current point in the history. Having a branch allows you to continue from this point without affecting the base point from which this stems.

Let’s make this clearer by taking a look at an example. Get a list of all the files Git is tracking in this project:

```
git ls-files
```

Now, issue the delete command for each file:

```
git rm <filename>
```

Then, verify you have removed everything with:

```
git status
```

And make a new commit:

```
git commit -am “Deleted everything”
```

Now we can restore our project:

```
git checkout master
```

Voila! We've restored everything back to its original state.

`master` is always the name of the main branch—the one from which you always start. Everything else branches from that. If you form an image in your mind of a railway system it might help to understand the concept of branching. Each station on the railway is like a commit in the Git history. You can travel to any station (commit) along the system (branches) to arrive at a previous location (which, in our case, is a state of the set of files).

And there's no limit to the number of branches you can create. You can always see the list of branches with:

```
git branch
```

There will be an asterisk to show you which branch is current. In order to check out a branch so you can work on it you simply enter:

```
git checkout <branchname>
```

Merging

As we discussed earlier, one of the main goals of a Source Control Management system is to allow for multiple developers to collaborate efficiently. That often means they can't afford to wait their turn to make changes to files. However, as you may recall, one of the highlighted features of Git is that the entire history is held locally—right there on your machine. So, how can multiple people collaborate, making changes to the same files, when they all have their own copies of those files? Without the special help provided by Git through merging it'd be impossible. Sadly,

exploring the detail of a merge is beyond the scope of this short introduction, but you can find more information at the Atlassian Git Tutorial[1] and the Git Reference[2].

Remotes

In order for a group to collaborate on a project they must share the Git repository. This can be done in a number of ways, but one popular tool for hosting Git repositories is Github[3]. An account on Github is free and, in order to support Open Source Software development, there is no limit to the number of publicly visible repositories you can host.

It's important to realize that Git and Github are two distinct entities. Although many times they are used together in the same sentence it's incorrect to use them interchangeably. Remember that your entire project history is in the Git repository, and *that* is local on your machine. A remote host like Github provides a centralized backup of that history.

Never Fear Change Again!

So now you know—when your projects are under Source Control with Git, the next time you find yourself cursing a seemingly irrevocable change, all is not lost.

There are even ways to undo changes that have not yet been committed! That functionality is beyond the scope of this guide, but a search on Google or Bing will yield plenty of detailed explanations.

One final important thing to note is that in the simple example we walked through here, we primarily used a single text file: **TheFirstFileUnderGitSourceControl.txt**. That was purely for illustration, but the same process can be applied to any file on your computer!

Hopefully we've covered enough here for you to have a good, basic understanding of what Git can do and why it can be so very useful when undertaking any substantial project.

[1] http://www.atlassian.com/git/tutorial/git-branches#!merge
[2] http://gitref.org/branching/
[3] https://github.com/

Return the Favor

Regardless of the type of work you're doing, it's immeasurably rewarding to contribute your talents to the world of Open Source software. Find a project on Github you use, or like, or believe could be better. Use the command:

```
git clone
```

to get a copy of the project's codebase for yourself to work on and improve. Github provides instructions for submitting a “Pull Request”[4] to have your changes reviewed and included by the project maintainer.

[4] https://help.github.com/articles/using-pull-requests

Appendix A: A Git Mini-Reference

List of Basic Git Commands

This summary of basic Git commands should be handy as a reference while you explore and learn more:

- Tell Git what files to track for changes
  - `git add` <*filename*, *folder name* or *list of filenames* (separated by spaces)>
- Store the current state so you can always return to it
  - `git commit -am` "*A message that will help you understand what this commit contains*"
- To restore all the files from a previous commit
  - `git checkout` <*an unambiguous portion of the hash*>
- To get the hashes of previous commits
  - `git log`
- To interact with "remote" git repositories (such as projects on Github)
  - `git push`
  - `git pull`