UBM Tech

Tech Digest

InformationWeek

JULY 2014

Next

DOWNLOAD PDF

How to Contribute to OPEN SOURCE

Powered by Dr.Dobb's

PLUS: Getting Started with Git >>

How to Contribute to Open Source Projects

Choosing the right project, working your way up, avoiding pitfalls, and getting results

by Andrew Binstock

Brian Behlendorf, the founder of the Apache Web Server project and a lead developer on Subversion, discusses how to get started on an OSS project — and what to expect.

DDJ: Let's say I'm a developer with some experience and I'm interested in contributing to one of the Apache projects. How do I get started?

BB: What's your motive?

DDJ: To get more experience, working with bright folks who are doing stuff in an area that interests me.

BB: That's not the most usual path. More often, it's a developer who wants help to solve a specific technical problem. After some Google searching, he's found some packages that claims to do x. And, if there's one that's free, that's the one that's going to get evaluated first. And he starts pulling it down and playing with it.

DDJ: Yes. And then what does he do?

BB: Let me talk about a sequence of events that's more likely to happen. The first step is generally to determine whether the software is any good.

When I look at a piece of open source software before I download code, I'm looking to see whether a lot of people are complaining about broken installations, or if there are questions that suggest poor programming practices. And are people getting answers quickly?

Every good open source project has a public discussion forum, email or forum-based, and has developers who have a stewardship mentality about it and care about happy customers, even if they're not paying customers. So, before even touching the code, I would evaluate the community — because there is an awful lot of code that has no community behind it, such as somebody open sourcing something they worked on at their last job, or an overnight hack, with no intention of making it usable.

Evaluate the community, look for activity, look for a release every couple of months, people who've used it and said good things or even bad things.

DDJ: So, if you determine the project fits your criteria, then what?

BB: You read some docs, you watch some people talking. Then you download the code, compile it, install it, and give it a dry run. If it's running and doing some cool things for you, you might show it to your boss. You then

This article originally appeared on Dr. Dobb's in 2011
www.drdobbs.com/231000080

deploy it to your production — and you're running. Then you discover there's a bug.

Now, what do you do with it? You probably dive in to figure out what's really going on in there, and in the course of that, you go rooting through the discussion lists and the developer lists.

Most projects have a differentiation between the users list and the developers list, and that's so that the developers can stay focused on building new stuff and those who want to help the community and the community themselves can support each other with a basic Q&A somewhere else.

DDJ: Yes.

It's amazing to me how people think of documentation as easy or an afterthought, but there's a huge difference between documentation written by someone coming up the learning curve and documentation written by someone who really knows it.

Start By Contributing Defect Data

BB: In the course of doing this, you probably looked in the issue tracker to see if anyone else has reported a "foo-bar not-found" sort of thing. In the course of narrowing it down, you've either realized that it was a mistake on your part or that you've found a bug.

The bug [might be] a pre-existing, pre-known defect and maybe you can actually add some data to it so the next person can find it more easily than you did. In which case, you want to get an account on the issue tracker and post a comment on that issue, saying, "Hey, this is happening to me, too," and try to contribute, add to the conversation that's there.

This helps because bugs take a long time to get from first being noticed to being resolved. Developers often ignore data from users, trying to recreate the conditions under which things happen. The vast majority of the work in fixing a bug is something that even users who don't understand the code can actually help with. They can actually try to replicate the bug, write a test case, etc.

And that is all extremely valuable. A programmer who is familiar with the code may be able to dive in and fix it. But that's such tedious work and it's high value because it's tedious and no one really wants to do it. So, to start, try looking for the outstanding defects and see whether they need further triage.

DDJ: OK.

BB: Karl Fogel talks about this in his book, *Producing Open Source Software* (http://is.gd/94PAdE), [a book, which without a doubt, is the best guide available for running an OSS project. — Ed.] namely, the benefit of marking certain issues as bite-sized tasks — things that developers could take on to understand the layout of the code, how different systems call each other, etc. Because there are often bugs that aren't big architectural defects but off-by-one errors or edge-case kind of things that benefit from a lot of triage.

DDJ: An excellent idea.

BB: Throughout all this, there is the conversation on the users' or developers' list. These messages are the lifeblood of the community. It's

the banter across the dinner table that drives the process. Join either the users' or developers' list. Let that simmer in the background. Don't pretend you have to understand every word, just get an ear for the music of the discussions. Eventually, you'll see comments that map to some of the situations that you see. Some of these lists have100 messages a day. You can't read all of them but you can get a feel for the gestalt of the project.

Some projects, like Apache, have more formal recognition of a developer as a committer — granted certain privileges on the repository. Even though commits can always be backed out, it is generally considered a mark of honor that other developers trust you enough to give you the keys.

Contributing to Documentation

DDJ: That's a great sequence to start with. I notice that on many projects that are trying to solicit participation, they recommend working on documentation, which always seems to be in short supply. How does that work?

BB: It's amazing to me how people think of documentation as easy or an afterthought, but there's a huge difference between documentation written by someone coming up the learning curve and documentation written by someone who really knows it. I'd say well designed and engineered documentation is more important than well designed and engineered source code. Because that's the ladder people climb up to go from casual first-time user to core user and core developer. And that has to be a solid ladder. A lot of projects try to encourage the developers, when they commit a source code change, to concurrently commit documentation changes. That's a high bar though, because many developers are not English-as-a-first-language, or are not proficient writers.

I'd say the other caveat is I think having new users come in and contribute to training materials is more appropriate. I think the format of training (especially screen capture and video, because it's a form of performance art), really forces you to learn the material: "Here's why Drupal is a kick-ass CMS, and here's how to build your first site with it." There's a saying: People remember 10 percent of what they're told and 90 percent of what they teach.

Working Your Way Up Through the Meritocracy

DDJ: Developers who are contributing out of ambition rather than because they have a specific problem to be solved may believe that the meritocracy provides a certain type of reward. Being a contributor is a feather in the cap if it's an esteemed project. So what typically moves somebody in the community's eyes from being just an occasional contributor into one of the leads, or a formal position on a project?

BB: Some projects, like Apache, have more formal recognition of a developer as a committer — granted certain privileges on the repository. Even though commits can always be backed out, it is generally

considered a mark of honor that other developers trust you enough to give you the keys. Other projects give out commit privileges like candy — apparently Gnome — and the premise behind that is it should be easy for everybody to throw their patches into the pool and we'll filter and sort through them later.

That's partly a tool question; Git is easier [for] managing a lot of users who aren't core committers. But being a committer on Apache is a big deal. The decision is made by one committer proposing to other committers on an individual project, "This person has contributed lots of valuable patches in the past, and has been helpful to new users."

I do think both Subversion and Git have made it easier for people to maintain branches and forks of code than it used to be, so there tend to be fewer fights over commit privileges. Instead, what you see is people just working.

There's always some work on a project that goes beyond self-interest. There's talk about aggregated self-interest, but it's actually enlightened self-interest, in that you've got to write code that can be understood by others, and when somebody has a newbie question, helping them find the answer to that question will pay off tremendously. There's always going to be many more users than developers, so it's incumbent on developers to give a little user support, and help new developers over the hurdles in getting their environment running and understanding the code layout. Someone who shows that level of altruism — it doesn't have to be full time — but there are a lot of people.

At Apache, it's a recognition not just for a few good patches, but for a commitment, a communication style, and an understanding of this thing called the Apache Way, which is not clearly defined but generally is do unto others as you'd have them do unto you: Have high quality code, be clear in your communication, and have a team-oriented spirit.

That's the criteria on Apache to be awarded commit privileges. And just be human, be on the mailing list, be helpful, help get the bug queue down. No active open source project has no bugs open. There's always something to do there.

DDJ: Jeff Fredrick, who headed up the CruiseControl project for a long time, told me that one of the things that happens is that the people who should become committers generally stand out by the nature of commitment and contributions. There's not a lot of discussion, it's generally pretty clear. Would you agree?

BB: Hmm, I can think of frequent examples of significant private conversations among committers over whether someone should have commit privileges, although that's less controversial because committer privileges can always be revoked.

DDJ: What about not granting privileges?

BB: I think there are some projects that err too much on the side of not granting commit privileges. It can seed various conspiracy theories as to whether it's justified or not. Sun, for example, with Open Office, really never gave a lot of commit privileges outside the Sun developers, because their working style was focused on a small cluster of developers in one physical location, having worked together for 20 years, and

they found it hard to trust other developers. So that's a case where they probably erred too much on the side of holding commit privileges too close to the vest.

DDJ: What about using branches and forks?

BB: I do think both Subversion and Git have made it easier for people to maintain branches and forks of code than it used to be, so there tend to be fewer fights over commit privileges. Instead, what you see is people just working. And they'll say, "You've got a good code base, I've got an extra patch, here's my tree, you can pull that patch from my tree, or someone else can build a derivative from that." In some ways, I think this has actually hurt the ability for communities to gel around a single code base.

Many developers are very rigorous and scientific and absolutist: "Your code sucks and you need to go back to school." It can be humiliating ... Even more subtle passive-aggressive kinds of things can cause somebody to lose face.

For instance, the Linux kernels that ship with all the different distros out there, it's pretty much a different kernel per distro. Different combinations of patches and settings. I think the Linux foundation does a good job of driving the Linux standard base, and we have much more conformance than we might otherwise have. It's still tough.

Every Apache project has a single code base. It has development branches and current branches and stable branches and all that, but the pool of developers are still focused on building one thing and building it iteratively.

Mistakes New Contributors Make

DDJ: What are the mistakes that you've seen people joining projects frequently make?

BB: Well, a little more than 10 years ago, on the Apache main HTTP developers mailing list, there was a developer who showed up from a well-regarded UNIX vendor for graphics machines, let's say, and they had recently moved to 64 bit. This was before Apache really had a portable runtime that abstracted away a lot of system calls so that it was easier to port to other platforms.

This guy showed up and hadn't really met anybody, and he emailed me and said, "I have a bunch of stuff to contribute, can I do that?" I said, "Yeah, just show up on the list and start posting patches." And I may have given him too lightweight instructions because he, ah, came on the list and said. "Great news, I ported Apache HTTPD to our 64-bit ship and have gotten permission from my company to redistribute these patches; and here is the first one out of 10." The first patch was a couple hundred individual changes. Many of them changed `#ifdef`s without allowing for the old code to continue to compile. It wasn't really an abstraction so much as a modification.

He said, "Number two will come tomorrow, and number three the next day, and you've got to apply them all in sequence because they're deltas, the first to the last, and that's how I modified and check-pointed my code." So the first patch gets posted and immediately people start saying, "This doesn't look like the right kind of change because it

breaks it on this other platform or this messy `#ifdef` just complicated the code and it would be nice to have a more elegant call that starts out so we don't have to repeat it a billion times in the code."

The next day, when he saw that response, he was flabbergasted. He said, "Wait, I can't deal with this. Guys, I ported this to 64 bit. You can't make me go back and redo all these changes. Besides, my second patch depends on everything in the first one going through, so I can't change anything. I have nine more of these." What he didn't understand is there is intense review of code that goes into an open source project and it's better to show up on the scene and say, "here's what I'd like to do, it's a substantial change and I'm wondering about the right way to go about it," rather than to say, "Good news, I've ported this to the Commodore Vic 20. And here's the changes made. Please commit these in."There's a bunch of different problems with that one; the size of the patches and the dependencies. The other is the attitude: I've written this software, I am God.

[There's] a righteous attitude that some developers get that ends up being fairly self-defeating, because it ends up accomplishing the opposite of what you intend. Instead of building confidence in your solution, it causes people to question it.

DDJ: What other errors do new contributors make?

BB: The other is just being too lazy to search through the discussion archives or to RTFM. Or, as a member of the community, to give a snarky response to somebody and it just escalates, or to give no response, and they interpret that as meaning these developers are stuck up or ignorant or hate newbies or whatever. Communication differences: it's the kinds of things that happen when two people are miscommunicating at long distances, and if they were face to face, wouldn't happen, even if they started off on the wrong foot.

DDJ: In many cases, contributors aren't just communicating across long distances, they're communicating across cultural barriers, too.

BB: Yeah. And you could add that for some projects that have a healthy user participation, a lack of understanding of the need to save face. Many developers are very

rigorous and scientific and absolutist: "Your code sucks and you need to go back to school." It can be humiliating and especially in Asian cultures, that's a death sentence — that's somebody who's never coming back to participate. Even more subtle passive-aggressive kinds of things can cause somebody to lose face.

Getting Committers To Respond

DDJ: I was looking at the Apache Poi project (http://poi.apache.org), which was looking for some point men for some of the subprojects. It gives some elaborate directions on how to participate. In a nutshell, build it, find the place you want to make your patch, submit your patch, and then bug the developer mailing list until somebody does something with your patch. It seems to me that that last bit is one where trouble lurks. If you're a newcomer and you're bugging people to respond to your patches, you're likely to rankle people if you don't do it right.

BB: If you're making genuine contributions and no one is responding? If the project is dead, there's not going to be anyone around to resuscitate it. Someone has to stand up and say, "No, this project has to come back alive, get the heart-jumpers."

DDJ: Does that ever happen? Have you seen projects that were completely dead come back to life?

BB: There are projects where it really is one core developer who people turn to when there's a question. Maybe there's an area of the code that no one understands or no one wants to touch so that the question remains unresponded to, while other conversations continue apace. There are far more underappreciated projects than living projects.

Patience, asking questions, always making sure that you're carrying a tone of appreciative inquiry, are the key. Such as saying, "I think this is the right solution, I'm really curious about what others think of it. If I'm the only one who cares, maybe the code should just be excised."

You also should rabble rouse a little bit and maybe go to the issue tracker and see who else has reported similar defects and maybe try to pull them back into the community. If you want to fire up a moribund project or portion of a project, then go out and speak on it. There are innumerable tech conferences out there these days and plenty of opportunities to speak, especially if you don't care to get paid. Telling people why a particular thing excites you is a great exercise, and a great way to make sure you really know it, too.

DDJ: Thank you. I think you've laid out a helpful roadmap, packed with useful observations and commentary that will help guide potential contributors and give them a sense of what to expect.

Andrew Binstock is editor in chief of Dr. Dobb's. *Long ago, Brian Behlendorf was the CTO at* Wired Magazine. *During his work there, he started patching the NCSA Web server. As he added more patches, a community of contributors emerged, which later forked and rewrote the server. This product became the Apache Web Server. He later founded the Apache Software Foundation. He also cofounded Collabnet, where he was a principal contributor to Subversion.*

Comment

Getting Started with Git: The Fundamentals

The distributed SCM system that's taking the world by storm has its own unique way of doing things. This tutorial explains how things work and the basic commands for getting started and checking-in changes.

By Scott Danzig

I was not particularly inspired by any SCM system until I dove into Git, created by Linus Torvalds, the founder of Linux. In this tutorial, I discuss what's unique about Git and I demonstrate how to set up a repository on GitHub, one the main free Git hosting services. Then I explain how to make a local copy of the GitHub repository, make some changes locally, and push them back to GitHub. As a side note, I learned much of what I know about Git from the book *Pro Git* (http://git-scm.com/book), which is is hosted free online. I recommend that you use the book to fill out the matter presented here and as a reference for later work with Git.

This article originally appeared on Dr. Dobb's in 2013
www.drdobbs.com/240160261

Why Git?

Git has numerous attractive benefits that, for me, make it my preferred DVCS:

- When you create a new branch, Git doesn't copy all your files over. A branch will point to the original files and only track the changes (commits) specific to that branch. This makes it blazingly fast to create branches compared to other approaches, such as Subversion (which laboriously copies the files).
- Git lets you work on your own copy of a project, merging your commits into the central repository, often on GitHub.com, when you want your commits to be available to others. Github.com, by the way, will host your project for free as long as it's open source. (And cheaply, if it's not. Another alternative is Bitbucket, which allows unlimited private Git repositories.) This means you can reliably access your code from anywhere with an Internet connection. If you lose that Internet connection, you can continue to work locally and sync up your changes when you're able to reconnect.
- When you screw up, you can usually undo your changes. You might need to call in an expert in serious cases, but there's always hope. This is the best "key benefit" a version control system can have.

- Git also lets you keep your commit history very organized. If you have lots of little changes, it lets you easily rewrite history so you see it as one big change (via something called "rebasing"). You can add/remove files in each commit, and certainly change the descriptions of each.
- It's open source, fast, and very flexible, so it's widely adopted and well-supported.
- With Git, you can create "hooks," which enable actions to occur automatically when you work on your code. A common use case is to create a hook to check the description submitted with each commit to make sure it conforms to a particular format. Perhaps you have your bugs described in a bug tracking system, and each bug has an ID #. Git can ensure each message has an entry for "Bug: SomeNumber".
- Another under-appreciated feature is how Git tracks files. It uses the SHA-1 algorithm to take the contents of files and produce a large hexadecimal number (hash code). The same file will always produce the same hash code. This way, if you move a file to a different folder, Git can detect that the file moved, and not think that you deleted one file and added another. This allows Git to avoid keeping two copies of the same file.
- While Git is not necessarily the most intuitive version control system out there, once you get used to it, you're able to browse through its internal directories and it makes complete sense. Wondering where the file with the hash code "`d482acb1302c49af36d5dabe0bccea04546496f7`" is? Check out this file: "`<your project>/.git/objects/d4/82acb1302c49af36d5dabe0bccea04546496f7`" There are also lots of lower-level commands that let you build the operations you want, in case, for instance, Git's merge command doesn't work how you'd like it to.

Tutorial

Let's jump in. In whatever programming language, you're going to start a new project, and you want to use version control? I'm going to create a silly, sample application in Scala that's very easy to understand for a demonstration. I'll assume you're familiar with your operating system's command-line interface, and that you're able to write something in the language of your choice.

Setup

Github is one of the go-to places to get your code hosted for free and it's what I'll use here. (BitBucket, Google Code, and SourceForge are some of the other free repository hosts that support Git.) All these hosts give you a home for your code that you can access from anywhere. Initial steps:

1. Go to http://GitHub.com and "Sign up for Github"
2. You'll need Git. Follow the step-by-step installation process at https://help.github.com/articles/set-up-git
3. Review how to create a new repository at https://help.github.com/articles/create-a-repo
4. Finally, you're going to want to get used to viewing files that start with a "." These files are hidden by default; so at the command line, when you're listing the contents of a directory, you need to include an "`a`" option. That's "`ls -a`" in OSX and Linux, and "`dir /a`" for Windows. In your folder options, you can turn on "Show hidden files and folders" as well.

Once you get this far, there's nothing stopping you (outside of setting aside some time to explore what Git has to offer). Let's look at some of the typical actions.

Clone a Repository

Cloning a repository lets you grab the source code from an existing project (yours or someone else's) that you have access to. Unless it's your project, you won't be able to make changes unless you "fork" the project, which means creating your own copy of it under your own account, after which you can modify it to your heart's content. I keep all of my projects locally (on my computer) in a "projects" folder in my home directory, "/Users/sdanzig/projects", so I'm going to use "projects" for this demo.

First, I fork my repository. I create a sample project, called potayto, on GitHub, as you now should know how to do. Let's get this project onto your hard drive so you can add comments to my source code for me.

First, log into your GitHub account, then go to my repository at https://GitHub.com/sdanzig/potayto and click Fork, as in Figure 1.

Then select your user account on GitHub and copy it there. When this is complete, it's as though it were your own repository and you

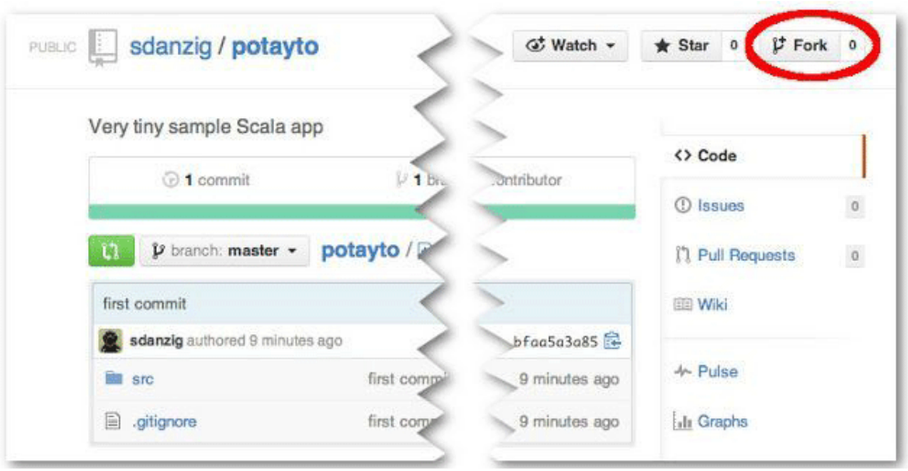

Figure 1: Forking (cloning) a repository.

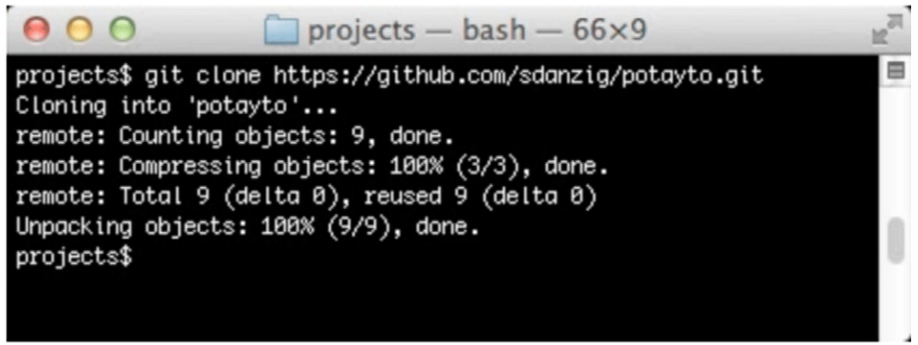

Figure 2: Copying a repository.

can actually make changes to the code on GitHub. Now, let's copy the repository onto your local hard drive, so we can both edit and compile the code there (Figure 2).

Folder Structure

There are a few key things to know about what Git is doing with your files. Type: cd potayto. There are useful things to see here when you list the contents in the potayto folder, being careful to show the hidden files and folders (with the `-a` option); see Figure 3.

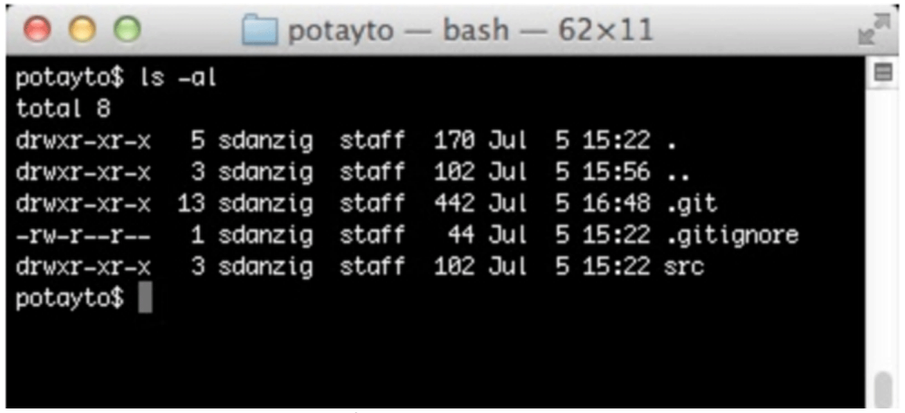

Figure 3: Examining the contents of a Git repository.

The src folder contains the source code, and its structure conforms to the Maven standard directory structure (http://is.gd/K5mJXU). You'll also see a .git folder, which contains a complete record of all the changes that were made to the potayto project, as well as a .gitignore text file. We're not going to dive into the contents of .git in this tutorial, but it's easier to understand than you think. If you're curious, please refer to the free online book at http://git-scm.com/book.

Git Log

A "commit" is a change recorded in your local repository. Type "git log" and you might have to press your space bar to scroll and type "q" at the end to quit displaying the file; see Figure 4. Git's log shows that the potayto project has 3 commits so far, from the oldest on the bottom to the most recent on top. You see the big hexadecimal numbers preceded by the word "commit"? Those are the SHA-1 codes I referred to earlier. Git also uses these SHA-1 codes to identify commits. They're big and scary, but you can just copy and paste them. Also, you need to type only enough letters and numbers for it to be uniquely identified (five is usually enough).

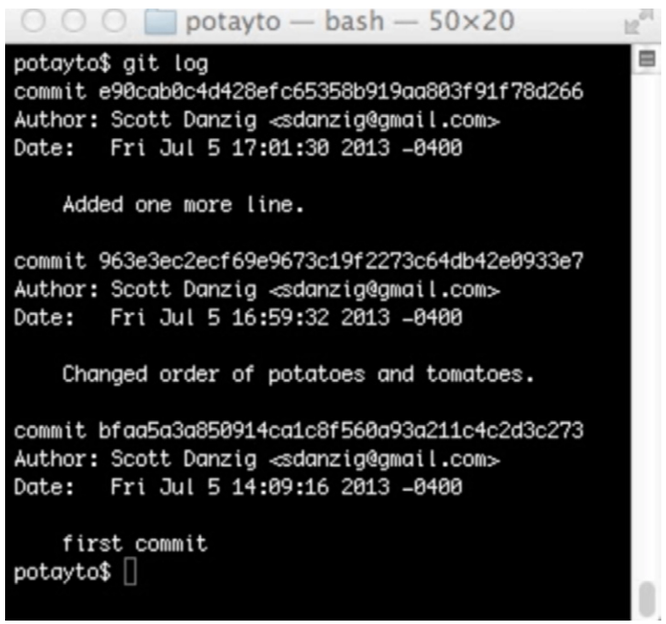

Figure 4: Output from git log.

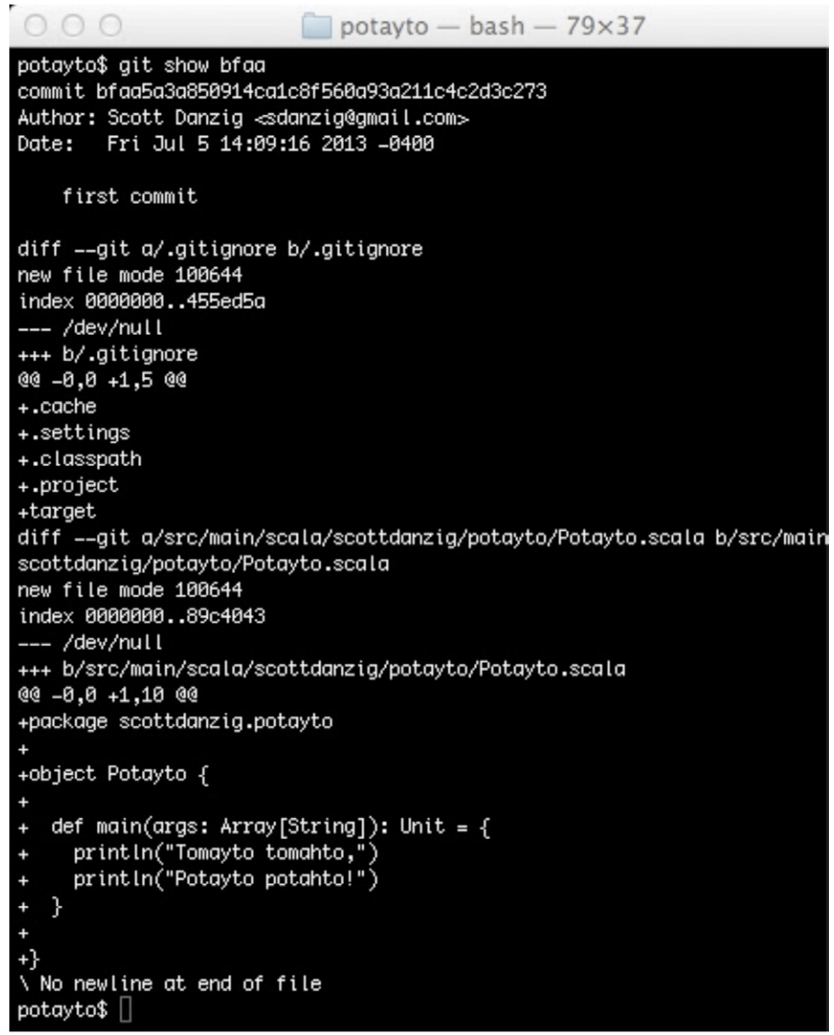

Figure 5: Contents of commit.

Let's see how my first commit started. To see the details of the first commit, type: git show bfaa. Figure 5 shows the results.

At the bottom of Figure 5, you can see that I initially checked-in my Scala application as something that merely printed out "Tomayto tomahto," "Potayto potahto!" You can see that near the bottom. The `main()` method of the Potayto is executed, and there are those two print lines.

Earlier in Figure 5, you can see the addition of the .gitignore I provided. I'm making Git ignore my Eclipse-specific dot-something files (for example, Eclipse's .project) and also the target directory, where my source code is compiled to. Git's `show` command is showing the changes in this file, not the entire files. The +'s before each line mean the lines were added. In this case, they were added because the file was previously nonexistent. That's why you see the `/dev/null` there.

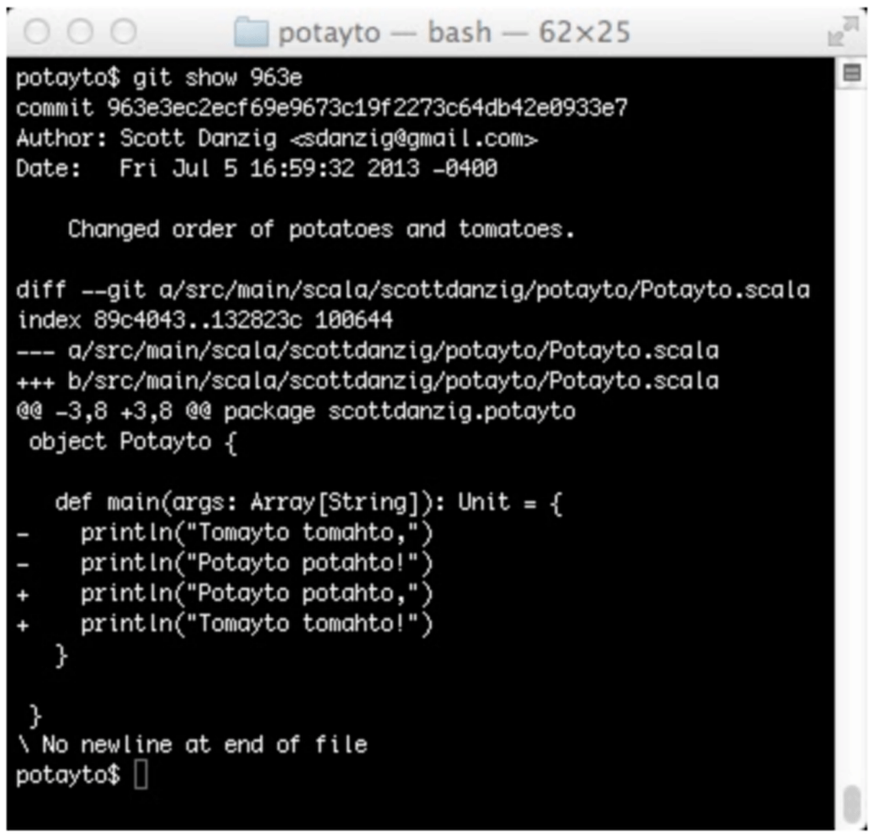

Figure 6: Commit message.

Now type `git show 963e` to get the output in Figure 6.

Here you see my informative commit message about what changed. These commit messages should be concise but comprehensive, so you're able to find the change when you need it.

After that, you see that I did exactly what the message says. I changed the order of the lyrics. You see two lines beginning with "–", preceding the lines removed; and two lines beginning with "+", preceding the lines added. You get the idea.

The .gitignore File, and Git Status

View the .gitignore file, which was dumped in Figure 5.

```
.cache
.settings
.classpath
.project
target
```

This is a manually created file in which I tell Git what to ignore. If you don't want files tracked, you add them here. I use the Eclipse IDE to write my code, and it creates hidden project files, which Git will see and want to add in to the project. Why should you be confined to using not only the same software as me to mess with my code, but also the same settings? Some teams might want to conform to the same

development environments and checking-in the project files might be a time saver, but these days, there are tools that let you easily generate such project files for popular IDEs. Therefore, I have Git ignore all the Eclipse-specific files, which all happen to start with a ".".

There's also a "target" folder in .gitignore. I've configured Eclipse to put my compiled code into that folder. We don't want Git tracking the files generated upon compilation. Let developers grabbing your source code compile it themselves after they make their modifications. You're going to want to create one for your own projects. This .gitignore file gets checked-in along with your project, so people who modify your code don't accidentally check-in their generated code as well. Other developers might be using IntelliJ IDE, which writes .idea folders and .ipr and .iws files, so they would add those to the .gitignore file.

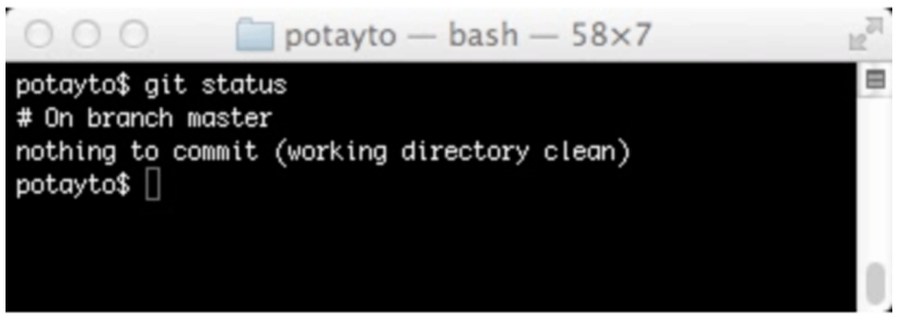

Figure 7: Status showing no new artifacts to commit.

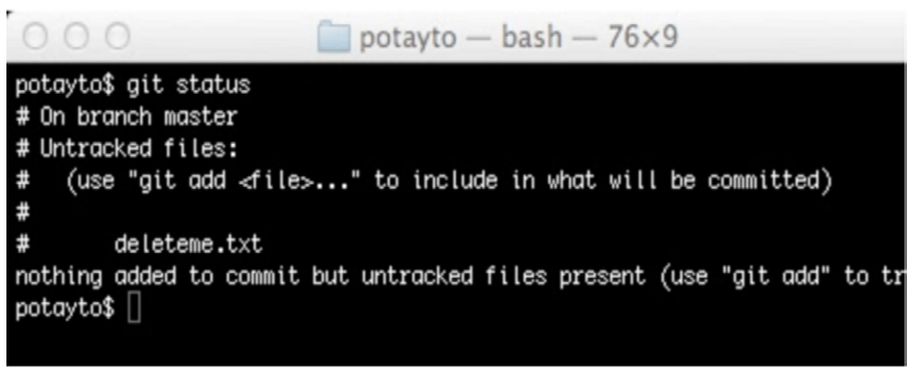

Figure 8: Status with artifacts to commit.

Getting the Status

Now, let's try this. Type `git status` (Figure 7). It shows that there is nothing new to commit to your local repository. You also see in Figure 7 that you're on the main branch of your project, "`master`." Being "on a branch" means your commits are appended to that branch. Now create a text file named "deleteme.txt" using whatever editor you want in that potayto folder and type `git status` again (Figure 8).

Use that same text editor to add "deleteme.txt" as the last line of .gitignore and check this out (Figure 9).

Other than its special treatment by Git, .gitignore is a file just like any other file in your repository, so if you want the new information saved,

Figure 9: Status with no changes to commit.

you have to commit the change just like you would commit a change to your code.

Staging Changes

One of Git's best features is that it offers a staging process. You can stage the modified files that you want to commit. Other version control systems await your one command before your files are changed in the repository — generally the remote repository for the entire team. When you commit files in Git, files are held in a staging area. You will later commit all the files from the staging area to the larger repository.

So, let's say you wanted to make a change involving files `A` and `B`. You changed file `A`. You then remembered something unrelated to do with file `Z` and you modified that. Then you went back to your initial change, modifying file `B`. Git allows you to add files `A` and `B` to staging, while leaving file `Z` "unstaged." Then you can push only the staged files to your repository. But you don't! You realize you need to make a change to file `C` as well. You "add" it. Now files `A`, `B`, and `C` are staged, and `Z` is still unstaged. You commit the staged changes only.

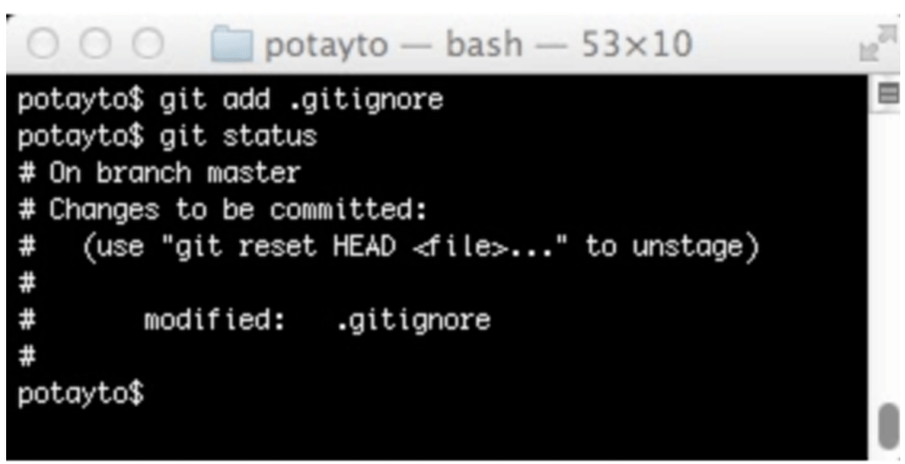

Figure 10: Promoting to staged status.

Read that last paragraph repeatedly if you didn't follow it fully. It's important. See how Git lets you prepare your commit beforehand? With a version control system such as Subversion, you'd have to remember to make your change to file `Z` later, and your "commit history" would show that you changed files `A` and `B`, then, in another entry, that you changed file `C` later.

We won't be as intricate. Let's just stage our one file for now. Look at Figure 9. Git gives you instructions for what you can do while in the repository's current state. Git is not known for having intuitive commands, but it is known for helping you out. "`git checkout -- .gitignore`" to undo your change? It's strange, but at least it tells you exactly what to do. To promote .gitignore to "staged" status, type `git add .gitignore` (Figure 10). The important thing to note here is that now your file change is listed under "Changes to be committed" and Git is spoon-feeding you what you need to type if you want to undo this staging. Don't type this: `git reset HEAD .gitignore`.

You should strive to understand what's going on (check out the *Pro Git* book I mentioned earlier to for those details), but in this situation, you simply are given means to an end when you might need it (in case you change your mind about what to stage).

By the way, it's often more convenient to just type "`git add <folder name>`" to add all modifications of files in a folder (and subfolders of that folder). It is also very common to type the shortcut "`git add .`" to stage all the modified files in your repository. This is fine as long as you're certain that you're not accidentally adding a file such as `Z` that you don't want to be grouped into this change in your commit history.

It's also useful to know how to stage the deletion of a file. Use `git rm <file>` for that.

Committing Changes to Your Repository

Time to do our first commit! To make the change in .gitignore official, type `git commit -m "Added deleteme.txt to .gitignore"`.

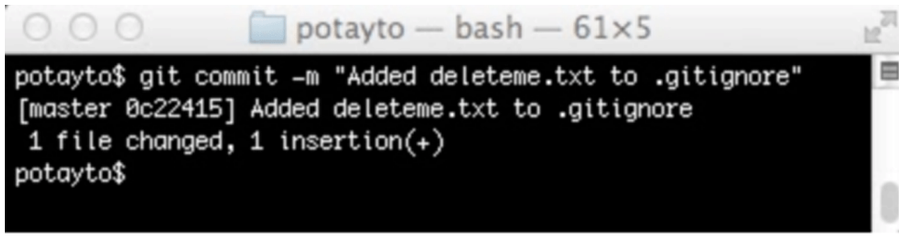

Figure 11: Committing with a commit message..

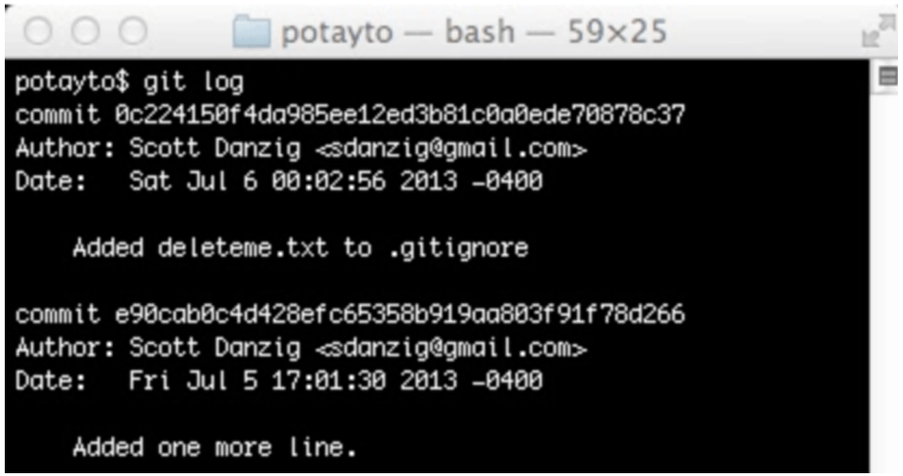

Figure 12: Log of the most recent commit.

The `-m` option is followed by the commit message. You could just type `git commit`, but then Git would load up a text editor and you'd be required to type a commit message anyway. In Mac OS X and Linux, vim is the editor that would load up; and in Windows, you'd get an error. If you prefer a full screen editor in Windows, you can type this to configure it:

```
git config --global core.editor "notepad"
```

If you end up in vim and are unfamiliar with it, note that it's a very geeky and unintuitive but powerful editor to use. In general, pressing the escape key, and typing `":x"` will save what you're writing and then exit. The same syntax will work to choose a new full screen editor in OS X and Linux, of course replacing notepad with the `/full/path/and/filename` of a different editor.

The full screen editor is necessary if you want a commit message with multiple lines, so if you hate vim, configure Git to use an editor you do like.

To see the commit you just made, type `git log`.

The change on top is yours. Oh, what the heck, let's take a look at it with diff; see Figure 13.

The `+deleteme.txt` is the change that was just committed. The way this diff works is that Git tries to show you three lines before and after each of your changes.

Here, there were no lines below your addition. The `-3,3` and `+3,4` are ranges. `-` precedes the old file's range, and `+` is for the new file. The first number in each range is a starting line number. The second number is the number of lines of the displayed sample before and after

your modification. The 4 lines displayed only totaled 3 before your change.

Note that if you want to revert changes you made, the safest way is to use "`git revert`," which automatically creates a new commit that undoes the changes in another commit. If you wanted to undo that last commit, which has the SHA-1 starting with 0c22, you would type: `git revert 0c22`. (Don't actually do this if you are following along.)

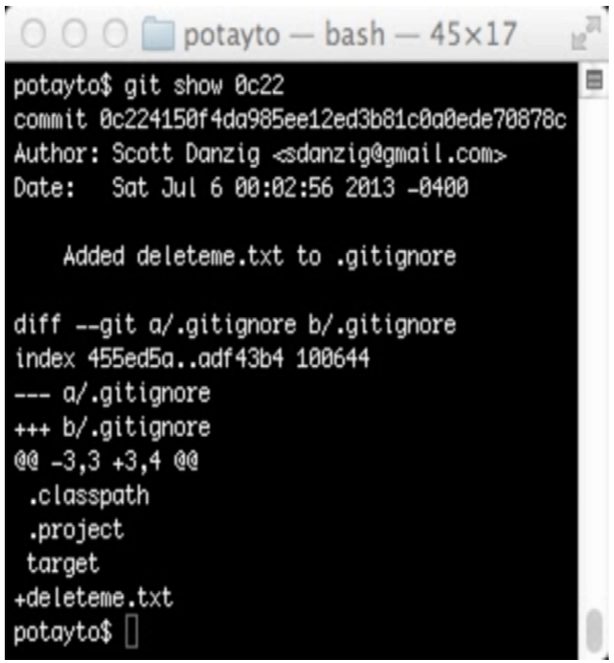

Figure 13: A diff output of the commit.

Pushing Your Changes to Remote Repository

You cloned your repository from your GitHub account. Unless something went horribly wrong, the repository on GitHub should be: `https://GitHub.com/<your GitHub username>/potayto.git`

Git automatically labels the location you cloned a repository from as "`origin`." Remember when I said the internals of a Git repository were easily accessible in that .Git folder in your project? Look at the text file `.git/config` (Figure 14). It's as simple as this.

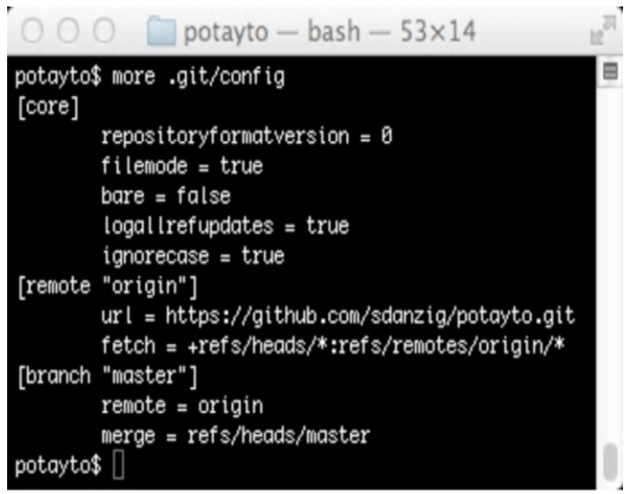

Figure 14: The contents of the .git/config file.

Before I explain how to make your changes on the version of your code stored on GitHub, I should first explain more about branches. I already noted how a branch is a separate version of your code.

A change made to one branch does not affect the version of your repository represented by another branch, unless you explicitly merge the change into it. By default, Git will put your code on a "master" branch. When you clone a project from a remote repository ("remote" in this case means hosted by GitHub), it will automatically create a local branch that "tracks" a remote branch. Tracking a branch means that Git will help you to:

- See the differences between commits made to the tracking branch (the local one) and the tracked branch (remote)
- Add your new local commits to the remote branch
- Put the new remote commits on your local branch

If you didn't have your local branch track the remote branch, you could still move changes from one to another, but it becomes more of a manual process. To do this, first, type `git status` (Figure 15).

That `deleteme.txt` change you made in your local master branch is not yet on Github! You have one commit that Github's origin) remote master branch (denoted as origin/master) does not yet have.

Let's put the change on Github. Git's `push` command, if you don't provide arguments, will just push all the changes committed in your local branches to the remote branches they track. This can be dangerous, if you have commits in another local branch and you're not quite ready to push those out also. (I once accidentally erased a week of changes in *New York Magazine*'s main repository doing this. We did manage to recover them, but don't ask.) It's better to be explicit. Type `git push origin master`.

You don't really need to concern yourself with the details of how Git does the upload.

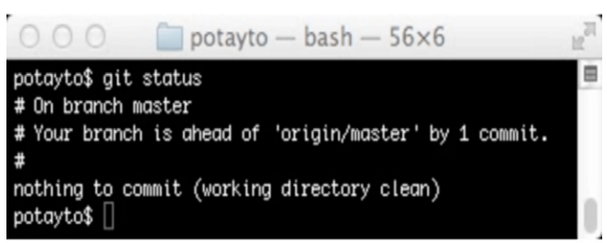

Figure 15: Showing that the local repository is ahead of the remote repository.

But as for the command you just typed, Git's `push` lets you specify the "remote" that you're pushing to, as well as the branch. By specifying the branch, you tell Git to take that particular branch ("master," in this case) and update the remote branch, on the origin (your Github potayto repository) with the same name (it will create a new remote "master" branch if it doesn't exist).

If you don't specify "master," Git will try to push the changes in all your branches to branches of the same names on the origin (if they exist there).

If you type "`git status`" again, you'll see your branch now matches the remote repository's copy of it. You can also look at the

Figure 16: Pushing commit.

changes to the origin, by typing `git log origin/master`.

This is the syntax to see a log of the commits in the master branch on your "origin" remote.

Figure 17: Log of changes to the origin.

You can see the change is there. You can also see this list of commits by logging into Github, viewing your Potayto repository, and clicking on the link in Figure 18.

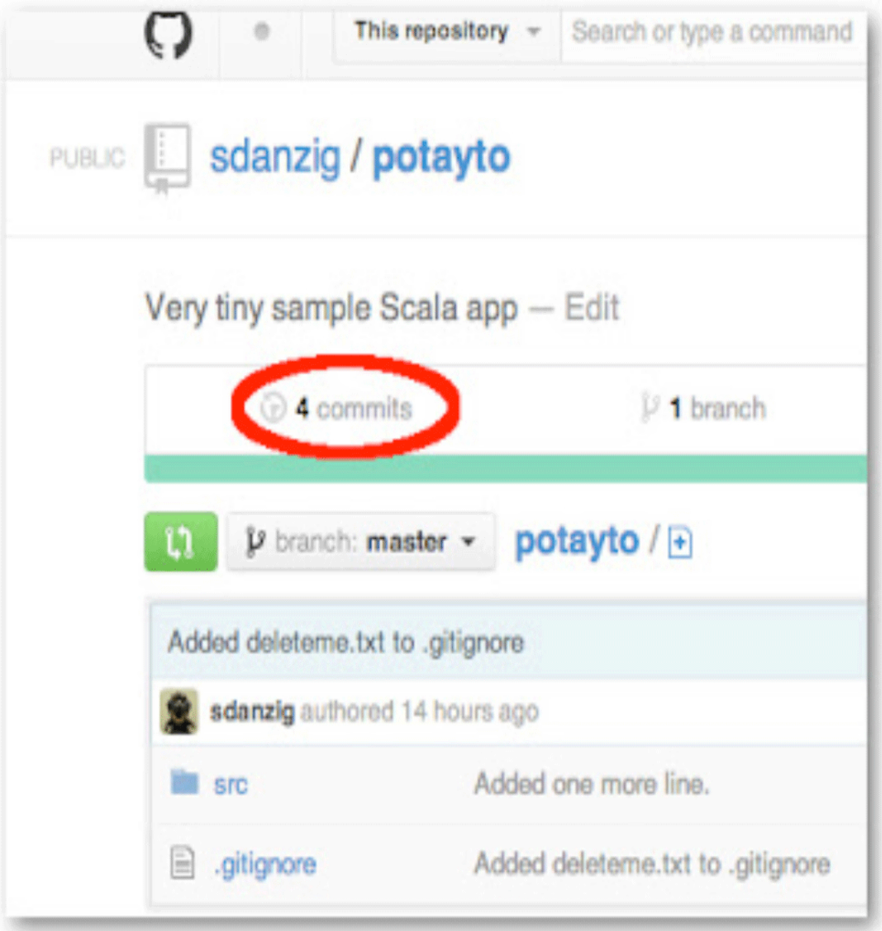

Figure 18: Seeing changes to the origin.

A continuation of this tutorial, titled "Git Tutorial: Branches and Workflow," is available at http://www.drdobbs.com/240160315. There, I examine how to pull changes from the remote repository, how to handle merges and merge conflicts, and other workflow tasks that are part of standard SCM work with Git.

After years of using many other version control systems, Git has proven to be the one that makes the most sense. It's certainly not dependent on a reliable Internet connection. It's fast. It's very flexible. After more than twenty years of professional software development, I conclude Git is an absolutely indispensable tool.

Scott Danzig has been programming for more than 20 years. His personal projects on Github can be found at https://Github.com/sdanzig.

Comment