ROCKABLE

GETTING GOOD WITH GIT

Includes Screen Casts!

Andrew Burgess

ROCKABLE

Rockablepress.com
Envato.com

Introduction to Git

So, you want to learn about Git? Then you've come to the right place. In this ebook, I'll be guiding you through the sometimes-confusing waters of using Git to manage your development projects. But before we get into that, let's take a step back and discover what exactly Git is, and why you would want to use it.

What is Git?

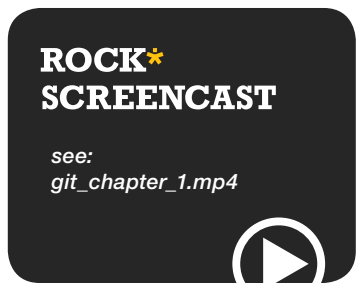
Git is a source code manager (SCM). More accurately, it's a distributed version control system. But what exactly does that mean? Let's break it down.

Git is Development Software

First off, Git is not a language, concept, or best practice. It's a program, a piece of software that you should use in your development, just like your text editor or FTP client. So what is it for? Git manages your source code… but what does that mean?

Git is for Versioning

The idea behind Git (and the other source code managers, which we'll talk about soon) is that it's a smart idea to keep snapshots of your coding projects. This means that, as you code, at intervals you declare a point in the history of the project. Think of a



timeline, where all the markers represent milestones in your coding. Where you put these markers is up to you, but it's wise to put them in when you've completed something. For example, you might make a mark after each feature you've implemented. That way your project timeline might look something like this:

- mark 1 — started the project, added the CSS and JS files
- mark 2 — built the main structure of the website
- mark 3 — added the navigation

I hope you're warming up to the idea of code snapshots. To steal an example from Tom Preston-Werner (a prominent member of the Git community) it's like taking a photo of your child every year, to track her growth. So if you're grokking the concept of timeline-markers for code, I think you'll see where Git fits in. If you wanted to make these snapshots manually, you'd have to copy your project directory and rename it every time. Git does this, and much more, for you.

Git is Distributed

That's why Git is called a version control system. But it's also distributed. Distributed, in version control systems, means that the code repositories (that's what your project folders are officially called) don't need to have a single home. With some systems, the main repository is kept on a server, and you as the programmer just get a copy of the most recent version of the project to work from. Then, when you want to create a new snapshot, you have to send that snapshot back to the server.

Let's back up a second: version control systems are made so that more than one person can work on the same project at once. Each programmer gets the code, works on it, and takes snapshots

(which are properly called *commits*). Then, they can share their commits with each other.

So, you make a commit and send it back to the server. This is NOT a distributed system. It required access to the server to make a commit or get updates from others (who have sent their changes to the server). Git is just the opposite. Almost everything you do with Git is done on your own machine. Of course, you can still share your repository with others, but not in the non-distributed way: you can send commits directly to other computers, and get commits back from them. Also, every copy of the repository includes the full commit history with Git. That's not true of some non-distributed (or centralized) systems, where you get only the latest commit from the server. The distributed way is safer, in that every copy of the repository is a full copy, so there should be no great loss if one dies. With centralized systems, if the server goes down, the commit history goes with it.

Why should I use a Version Control System?

We already saw that a system like Git makes it easy to create a code timeline, but why would you want to do that? And are there other benefits? Let's discuss some reasons why you should use a source code manager.

Freedom to Play

When you're using a version control system (VCS) and making commits regularly, you don't have to worry about breaking anything. If you really mess something up, just roll back to the latest commit and keep going.

Freedom to Branch

We'll talk more about branching in Chapter 3, but it basically lets you take your project in two or more directions within a single repository (trust me, it will feel magical). This is useful in situations where you're building version two of a project, but need to supply bug fixes for version one. Or, maybe you want to implement a rather challenging feature. It would definitely be a good move to give it a branch of its own. Without branching, you'd have to duplicate your project directory each time you wanted to try something new: not fun.

Freedom to Share

Using a VCS makes it really easy to share your project with others, and let them help you develop it. Without a VCS, you'd have to copy, compare, and integrate their changes all by hand. Again, not fun.

Where did Git come from?

Time for a history lesson: Git was created by Linus Torvalds, the creator of the Linux kernel. Actually, Linus built Git for managing the code for the Linux kernel. He looked at the available SCMs but came to the conclusion that none of them were fast enough. So, he built his own. You can have the peace of mind that Git will be able to handle your projects with blazing speed and — unless you're building an operating system — super-high efficiency.

Why not use another Source Code Manager?

Hopefully by now you're convinced that it's a good idea to use a Version Control System. But why Git? There are several other options out there, the main ones being:

- Subversion
- Mercurial
- Perforce
- Bazaar

What's wrong with these? Really, there's nothing inherently wrong with them, but here are a few reasons why you might prefer Git:

- Git is fast
- Git is easy to learn
- Git offers a staging area
- GitHub is available for sharing

There are other reasons, but they won't make much sense if you're not familiar with Git. You can read about these reasons and others in detail at WhyGitIsBetterThanX.com. Once you've gotten the hang of Git, go back and review that site. When I did, I was surprised to see what other SCMs were missing!

Summary

In this chapter, you've learned what Git is, where it came from, and why you should use it. Now, let's dive into the terminal and get our feet wet with a few commands!

2

Commands

By default, Git is a completely command line-based program, so if you want to use it well, you should be familiar with the command line. If your command line-fu isn't quite so polished, this chapter will help you get up to speed.

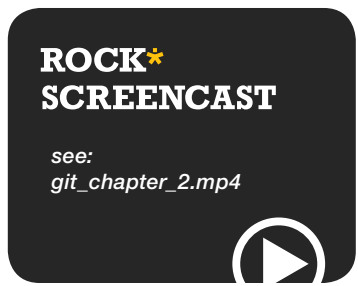
A Warning

Before we go anywhere near the command line, you need to know something: 99% of everything you do on the command line is irreversible. This is especially true of deleting things. When you remove files or folders from the command line, they don't go to the trash, the recycle bin, or any other kind of halfway house. They're gone forever (barring some kind of hardware level recovery… and even that won't always work). Therefore, it's smart to double-check your commands as you're getting used to the command line.

That said, working on the command line can be much more efficient and (if you're geeky like me) very fun. Don't get scared off by the cryptic nature of it. Take things slowly, and it'll be worth your time.

Also, don't worry about hurting anything today. All but one of the commands we're about to discuss are perfectly harmless.

There's one more thing to note: if a command was executed successfully, it usually won't give you any feedback. It may seem a bit confusing, but get used to the fact that if the command



line is talking back to you, you'd better pay attention: something is probably wrong! (Of course, there are several commands — and Git has many of them — that are specifically made to output information.)

Another Warning

Git will work just as well on one operating system as it will on another. However, the other command line commands will differ from OS to OS. Everything we'll discuss here will work fine on Linux distributions and Mac OS X. If you're on Windows, don't worry: most of it will work for you, too. If there are any differences, I'll be sure to point them out.

Oh, and if you are on Linux, please skip the rest of this chapter; you will learn absolutely nothing.

Opening the Command Line

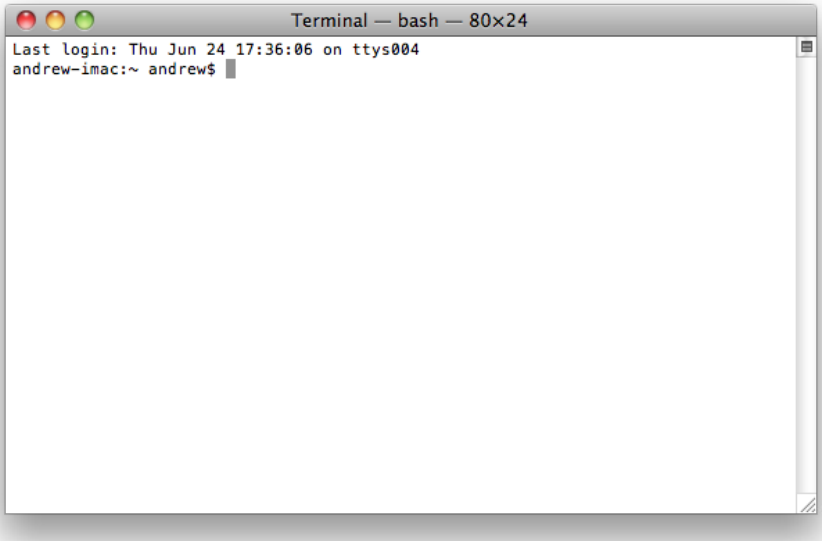
Although we will all know what you're talking about when you say "command line," neither OS X nor Windows call it that. On the Mac, it's called Terminal. To open it, open */Applications/Utilities/Terminal.app* (or just do Spotlight/Quicksilver search for "Terminal").

The Windows manifestation of this beast is called Command Prompt. To open it on Windows XP, choose *Start* ➤ *Programs* ➤ *Accessories* ➤ *Command Prompt*. If you've got Windows Vista or 7, simply search "Command Prompt" or "cmd.exe" from the Start Menu ("cmd.exe" from the Run box should work in any version).

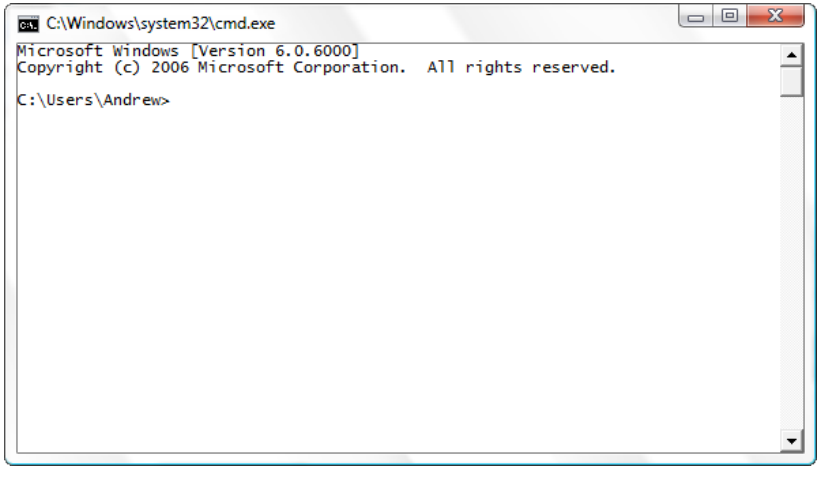
Now that you're looking at a command line, let's begin!

What You're Looking At

So you've got your command prompt open; don't be intimidated. It probably looks like this:



Or, like this:



So, what does this mean? Let's look at the Mac Terminal first. The line starts with the name of the computer: in this case, it's **andrew-imac**. Then, after a colon, we have the name of the current folder (we'll talk about this next). After a space, we have the name of the current user account (**andrew**), and then a dollar sign. Take note: this setup can be changed, so you may see terminals set up differently.

On the Windows prompt, you've simply got the path to the current folder and a greater-than sign (>).

It's after these lines that you'll enter one or more commands. When you hit 'enter,' they will execute, and you'll get another prompt, letting you know the previous command has been completed.

Commands

Let's move on to the commands, already!

whoami

We'll start with an easy command. Type the following on your command line (note: the dollar sign at the beginning is my representation of your command line; don't type it):

```
$ whoami
```

Yes, that's "Who am I?". On OS X, the Terminal will print out your user name; on Windows, you'll get both the computer's name and your user name.

You may be wondering why we got feedback from this command. Remember, there are two command camps: there are commands that tell you something (like **whoami**), and there are commands that

do something (these are the ones that usually don't respond when successful).

pwd

Next, if you're on a Mac, try this command:

```
$ pwd
```

Yes, this means something: you're asking the command line to Print the Working Directory ("directory" is an alternate name for a folder, which you'd better get used to if you plan to work on the command line much). This brings up another command line concept: when working in a terminal, you are always working from a certain folder. The command **pwd** is one way to find out what folder you are in. On a Mac, you may have gotten a response like:

```
/Users/andrew
```

Above, I said that the name of your current folder (or working directory) shows in your prompt. Your home folder is represented by the tilde (~), so that's what shows in your command line: the **pwd** command returns the full path.

The **pwd** command doesn't exist on the Windows Command Prompt… but you won't really need it: the complete path of the folder you are currently in is displayed in your prompt, so you can always see it.

ls / dir

So if you're always in a folder when working on the command line, you're probably working with the contents of that folder. Therefore, you'll want to know what that content is! That's what our next command is for: on the Mac, use

```
$ ls
```

If you're on Windows, do this:

```
$ dir
```

The **ls** command stands for "list files"; **dir** means "directory." Both of these commands do basically the same thing: show you what files are in the current directory. On a Mac, this probably looks something like this:

```
aFolder   another.txt   nothingHere.txt   something.txt
```

And on Windows, you'll see this:

```
06/25/2010  06:47 AM    <DIR>          .
06/25/2010  06:47 AM    <DIR>          ..
06/25/2010  06:47 AM    <DIR>          aFolder
06/25/2010  06:46 AM                 0 another.txt
06/25/2010  06:46 AM                 0 nothingHere.txt
06/25/2010  06:46 AM                 0 something.txt
               3 File(s)              0 bytes
               3 Dir(s)  43,051,921,408 bytes free
```

There's a lot more information shown by default on Windows, but **ls** can show all that as well, with a few options that we won't get into right now.

While we're here, look at the first two entries in the directory list from Windows: they are **.** (period) and **..** (double period). On both Windows and Mac, a single dot represents the folder you are currently in, and a double dot represents the folder above, or the "parent" of the folder you're in. These will come in handy.

cd

This is an easy one: **cd** stands for Change Directory. We can use this command to switch from folder to folder. Let's say the directory you're in has a directory named "images" inside it. To move into that folder, just type this:

```
$ cd images
```

This is the first command we've looked at in which we use a parameter, but parameters are very common on the command line. You can also use this command to jump more than one level, with something like

```
$ cd path/to/deeper/folder
```

But how do we get back up the file structure? Remember those double dots? They'll bring you up a folder:

```
$ cd ..
```

Of course, you can jump up multiple folders at once in the same way you went down into them: **../..**

One more thing to note here: what happens if you don't give **cd** a directory parameter? On a Mac, it will take you to your home directory (just like **cd ~**). On Windows, it will act the way **pwd** does on the Mac: print the path of the folder you're in.

mkdir

So, now we know how to move between folders and see what's in them; so how do we make folders? On both Windows and Mac, it's pretty simple:

```
$ mkdir newFolderName
```

Really, it's that easy.

rm / del

This is the one command we'll look at that has potential danger, so be cautious when you're using it. The **rm** command is for removing files and folders.

```
$ rm fileName.ext
$ rm folderName
```

If you're trying to remove a folder that isn't empty, you'll get an error. There's an easy way to delete a folder with content, but I'll leave that for you to discover.

On Windows, you'll have to use the **del** command (think "delete"). Just like on Mac, this works for files and folders:

```
$ del theFile.ext
$ del theFolder
```

cp and mv / copy and move

The next step is moving files around. Again, it's really not that hard. Here's how to copy files: the first parameter is the file you want to copy; the second one is the destination.

```
$ cp file.ext ~/other/path/file.ext
```

If you want to move a file, you can use the same syntax, just substitute **cp** with **mv**.

```
$ mv thisFile.ext /to/this/folder
```

Notice how I didn't put a filename at the end of the path; if you don't want to change the file name when you move it, you don't have to put the filename on the destination path. This works for `cp` as well.

ROCK* TIP

The `mv` command can also be used as a file renaming utility: just use `mv oldFile.name newFile.name`

On Windows, these commands are the full words **`copy`** and **`move`**. They work pretty much the same as on the Mac, although you should note that paths on Windows require back-slashes (\ ); on the Mac, it's all forward-slashes (/).

Opening Files

Let's now talk about opening files for editing. When you're on the command line, there are two ways to edit the file:

1. From an external editor
2. From within the terminal window

For all editors, the standard way to open a file is to use the editor's command followed by the file to open. For example, on the Mac, TextMate installs a command to open files from the command line. Try this:

```
$ mate index.html
```

This command would open the **`index.html`** file in TextMate. There are so many editors — on both Windows and Mac — that I can't go over them all. Google around and you'll probably find instructions for your editor of choice.

For editing files right on the command-line, try *nano* on the Mac and *edit* on the PC:

```
$ nano onTheMac.txt
```

```
$ edit onThePC.txt
```

This will open an editor right within your command line window.

ROCK* TIP

If you're on Windows, there's a good chance you just have to use the path to your text editor's executable file, handing it the file to edit. For example:

```
$ "C:\Program Files\
 Notepad++\notepad++
.exe" default.css
```

exit

When you're done on the command line, you can exit simply by using the `exit` command:

```
$ exit
```

If you're on Windows, this will close the command prompt window. If you're on Mac, the exit command won't close the window, but it will stop all the running processes. If you want Terminal to close the window on executing `exit`, go to the *Terminal Preferences*, under *Settings*, then *Shell*, choose "Close if the shell exited cleanly" under "When the shell exits."

The PATH

Let's conclude with a bit of theory: what exactly is going on when you type a command on the command line? Well, all these commands are simply little single-task programs. They're stored in one or (usually) more folders on your computer. How does the command line know which folder to look in to find the program you're trying to execute? It uses a variable called `$PATH` (just `PATH` on Windows). This is just a list of paths to the folders that hold

these commands. When it wants to find a program or script, it just searches through each folder in the **$PATH**.

So what's in your PATH? Well, here's how to find out, as well as the contents on my PATHs (I've added line breaks for clarity).

Mac

```
$ echo $PATH
/usr/local/bin:/usr/local/sbin:
/usr/local/mysql/bin:~/bin:/opt/local/sbin:
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:
/usr/local/git/bin:/usr/X11/bin:
/opt/local/bin
```

PC

```
$ PATH
PATH=C:\Perl\site\bin;C:\Perl\bin;C:\Windows\system32;
C:\Windows;
C:\Windows\System32\Wbem;
C:\Program Files\Java\jdk1.6.0_18\bin;
C:\tools\;C:\Python26\;C:\Ruby186\bin
```

Notice that paths on Windows are separated by semicolons, while on Mac they are separated by colons. While most of what you see here is default, I've added a few extra paths on both my Mac and PC. This is more advanced than we'll get here, but know that it can be done.

Advanced Command Line Skills

If you've tested out each of the commands we've looked at, you're well on your way to mastering the command line. If you're interested in learning more, there are plenty of resources. Sometimes, the best resource is the built-in documentation on the

commands. On Mac, you can access the manual for a command with one of the following commands:

```
$ man <command-name>
$ <command-name> -h
$ <command-name> --help
$ <command-name> help
```

On Windows, this should work:

```
$ <command-name> /?
```

If you're on Mac or Linux, and you want to learn more about the command line, I heartily recommend PeepCode's Meeting the Command Line and Advanced Command Line screencasts. On Windows, check out Windows PowerShell, Microsoft's much more powerful version of the command line.

Summary

In this chapter we've looked at the basics of working on the command line. Now we're ready to dive in to Git!

3

Configuration

Well, now you know why should use Git and you're comfortable on the command line. Now we're ready to start actually using Git itself. Of course, we'll start by installing it!

Installing Git

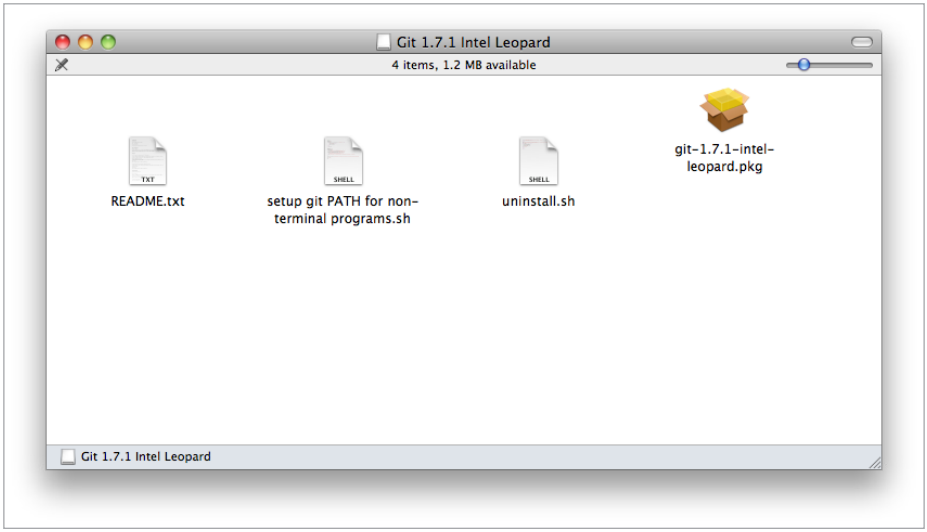
Installing Git is very OS-dependent, so I'll show you how to do it first on Mac, then on Windows.

ROCK* SCREENCAST

see: git_chapter_3.mp4

On the Mac

There are several ways to install Git on the Mac: the easiest way is to simply use the Git OS X Installer. Head over to the git-osx-installer page on Google Code, and click "Download the packages here." Then, download the latest image; in my case, that's Git 1.7.1. Mount the image and install the package inside.



If you plan to use Git from programs other than the terminal, run the shell script in the Git image.

If you use a package manager such as Homebrew or MacPorts, there's an easier way to get Git.

Homebrew

```
$ sudo brew install git
```

MacPorts

```
$ sudo port selfupdate
$ sudo port install git-core
```

(The **sudo** part just means, run the command as the root user. You'll usually have to do that when installing software. When using **sudo**, the command line will ask you for your user account password.)

If you really want to get adventurous, you can install Git from source. First, though, you'll have to install the Apple Developer Tools. You can get them with a free Apple Developer registration at the Mac Dev Center (if you're not on Snow Leopard, install them from the CDs/DVDs for your version of Mac OS X). Once you've got that under your belt, you can continue.

We're going to be installing Git to the **/usr/local** directory, so you want to make sure that Terminal will be able to find it in the **$PATH**. Using the command we talked about above, check for the **/usr/local/bin** path (**bin** stands for binary, and is the sub-directory that your compiled Git programs will be stored in). If it's not there, open **.profile** and add this line to the end:

```
export PATH="/usr/local/bin:$PATH"
```

ROCK* TIP

What's `.profile`*? The Mac Terminal uses a file called* `.profile` *to keep track of command line customizations. As you get more accustomed to the command line, you'll find yourself going to it and other files to make edits. the* `.profile` *file is in your home directory, but you won't be able to see it by default. The easiest way to get to it is via the Terminal command* `mate ~/.profile`*; replace "mate" with your editor of choice.*

After you save and close **`.profile`**, restart Terminal. Running **`echo $PATH`** should now show you that **`/usr/local/bin`** is in your **`$PATH`**.

Now we're ready to install Git. We'll get into some commands we haven't discussed, so if you aren't comfortable on the command line, you should probably use one of the previous options. If you're good to go, just be careful and let these commands be a launching point for your own command line research!

We'll start by downloading and extracting the Git source code:

```
$ curl -O http://kernel.org/pub/software/scm/git/
    git-1.7.1.tar.bz2
$ tar xzvf git-1.7.1
```

Now, we'll move into that folder and build Git:

```
$ cd git-1.7.1
$ ./configure --prefix=/usr/local
$ make
$ make install
```

Great that's it! Now, you can delete the folder and archive:

```
$ cd ..
$ rm git-1.7.1
$ rm git-1.7.1.tar.bz2
```

Now, you're ready to go. You'll be able to see that Git was installed successfully (and where it was installed) by running this command:

```
$ which git
```

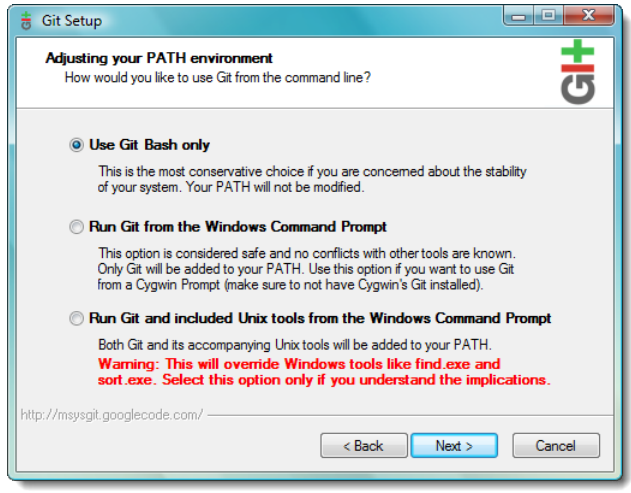
You should get **`/usr/local/bin/git`**. If you do, you've successfully installed Git from source! (Thanks to Dan Benjamin for these instructions on installing Git from source.)

On Windows

Because of its Linux roots, Git doesn't run quite as "natively" on Windows as it does on Mac. However, PC users can still take advantage of the Git goodness. Here's how:

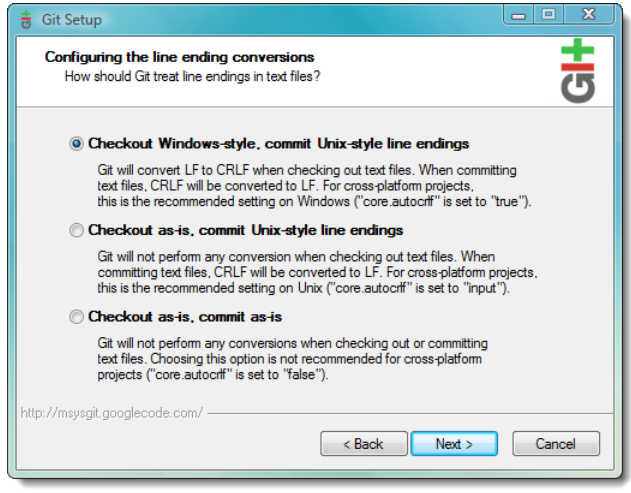
We'll install Git on Windows using the Msysgit installer, which you can get from its Google Code project page. Download the latest full installer from the downloads page, which appears to be version 1.7.0.2 as of this writing (it's labeled as the featured beta). Launch the installer and start moving through it. It will look like your regular app installation until you come to this panel (*top image on p. 30*).

This is asking how much you want to integrate Git with your system. If you read through the options, you'll see that, by default, Msysgit will create a separate command line that you can run Git commands on. Of course, you'll be able to run all the regular commands as well. The second command will include the path to the Git tools in your PATH environment variable, allowing you to run Git commands from the regular **`cmd.exe`**. The last command, well padded with a warning, will include Git and its additional Unix tools in your PATH; this will override the default Windows command line



tools that have the same name as the Unix ones, so make sure you want this if you choose this option. Since I prefer the Unix tools, I'll choose option three. If you're not sure, option two will be more convenient than option one.

The next dialog may cause some confusion as well:



This is all about the invisible characters that mark line endings. Basically, Unix systems require a line feed character, while Windows wants a carriage return character and a line feed character. According to the dialog, “Checkout Windows-style, commit Unix-style” is best for Windows when doing cross-platform projects. If you’re only working with Windows computers, you could choose “Checkout as-is, commit as-is,” but I wouldn’t recommend it; you never know when your project might need to be edited on another OS.

After these decisions, Git will complete the installation. You’ll probably get a “Git Bash” shortcut on your desktop; although you may have chosen to include Git in your PATH so that you could use the commands from the regular command prompt; this shortcut provides you a few extras, such as text highlighting and a bit of Git info in the prompt. Once you get the hang of Git, you’re free to choose whichever you please.

So, now you’ve got Git setup on your system; it’s time to do some configuration before we start using it.

Configuring Git

I know you’re chomping at the bit, but there’s one more step to take before we can start. We have to configure Git so it knows who you are.

Open up your command line (really, you shouldn’t close it for the rest of the book) and execute this:

```
$ git config --global user.name "Andrew Burgess"
$ git config --global user.email "andrew8088@gmail.com"
```

Of course, replace my name and email address with your own. This information is important, and you’ll see why later. While there

are other configuration settings you can use, I'll just set one more for now: all the programming languages I use are whitespace independent, so I'll set:

```
$ git config --global apply.whitespace nowarn
```

This way, Git will ignore whitespace changes. There are many other configuration settings, but you shouldn't need them until you get more advanced. If you want to check them out, you can look over the official Git documentation.

Using Git

Well, we're finally ready to start using Git! The first step is to set up our Git repository. Open your command line, create a new directory in the location of your choice, move into it, and let's get going.

Git Init

It's incredibly simple to start using Git on a project; just run this command from within the project folder:

```
$ git init
Initialized empty Git repository in /Users/andrew/
Documents/Projects/newApp/.git/
```

That's it; that's all you have to do. Whether you're just starting your project or you've already worked on it for a while, just run this command in the project directory and you're ready to use Git to manage your project.

For the technical, notice the path in the response to the **`git init`** command: it's your project directory with a **`.git`** directory inside. That **`.git`** directory is the home for all Git's work.

Git Status

It's smart to use the **`git status`** command all the time when you're learning Git. Of course, it will be handy once you're proficient as well. Here's how it works. Let's say you have three files in project:

```
README
index.html
default.css
```

So, you run **`git init`** on this directory; then, run this:

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   README
#   default.css
#   index.html
nothing added to commit but untracked files present
(use "git add" to track)
```

The first line tells you that you're on the master branch; we'll discuss branching later on in this chapter. The next line — initial commit — is a reminder that we have not committed anything to our new git repository (we're preparing the initial commit). Next we have a list of untracked files. Here's what goes on: Git doesn't automatically track the progress of all the files in your project repository; you have to let it know what to track. We'll look at this soon (the status message gives you a hint), but for now, notice that

all three of the files we have in our directory are correctly listed as untracked.

Git Add

So, Git has seen the files in our project; however, we haven't yet told Git to track the files, so let's do that now. We do this with the **`git add`** command.

We'll look at two ways to do this; first, we can tell Git to track all the files in the directory (and all sub-directories) by using the `.` (dot) parameter:

```
$ git add .
```

If you don't want to add all the files, you can do them individually by giving their names as parameters.

```
$ git add index.html default.css
```

So, what happens when we use the **`git add`** command? Git is "moving" the files into what's called the staging area. You haven't yet told Git to take a snapshot of the project as it is; this is the preparation step. Think of the staging area as a loading dock: the files are ready to go, but they haven't yet been loaded on a truck. It's up to you to tell Git what to do with these staged files. Let's say you ran the second command above, putting **`index.html`** and **`default.css`** to the staging area. What would you get in response to the **`git status`** command?

```
$ git status
# On branch master
#
# Initial commit
#
```

```
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
#   new file: default.css
#   new file: index.html
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   README
```

As you can see, Git has added another section to the output: "Changes to be committed." This means that **`default.css`** and **`index.html`** are on the loading dock, waiting further instruction. Let's give that instruction now!

Git Commit

As we've discussed, the primary purpose of source code management systems is to keep a snapshot record of your project as it grows up. Before we do that, run **`git add .`** to make sure we have everything in the staging area (loading dock). Now, we're ready to take the snapshot.

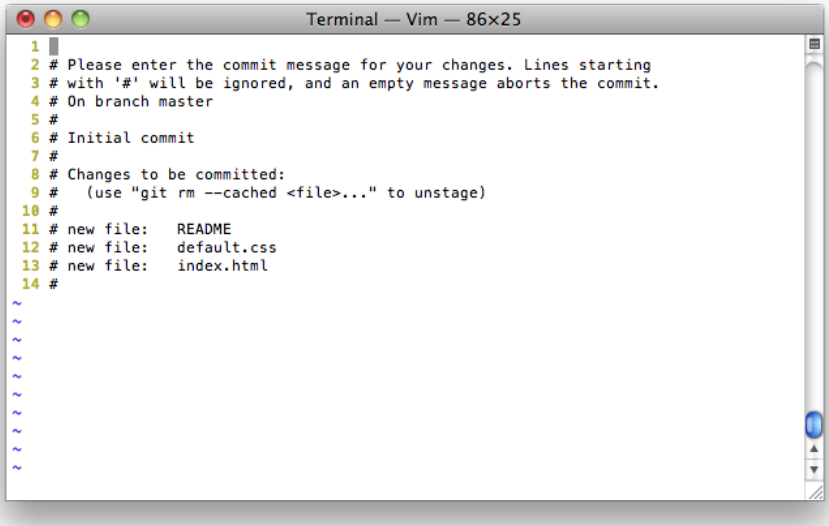
As we've discussed, the correct name for a project snapshot is a commit. The simplest way to make a commit is to execute this command:

ROCK* TIP

It's important to understand that Git doesn't track files: it tracks content. This means that when you change the content of a file by adding or removing lines, you'll need to re-stage those files; this might seem like a nuisance, but it's a very powerful concept. You'll learn a shortcut for staging and commiting edited files in the next chapter.


```
$ git commit
```

This will whisk you away to your command line's default editor; here's what I get:



For every commit you make, you'll supply a short message to remind you what's in the commit. It's a little change-log, if you will, recording the changes between this and the previous commit. Since this is the first commit in our project, I'll just type in "initial commit" and close the editor. (Although there are many other lines — really, a git status message — in the editor, only lines that do not start with # will be used in the commit message.)

Once you close the editor, you'll get this output:

```
[master (root-commit) 604ff85] initial commit
 3 files changed, 10 insertions(+), 0 deletions(-)
 create mode 100644 README
 create mode 100644 default.css
 create mode 100644 index.html
```

While you don't have to understand all of this, notice these things:

- The top line includes three important pieces: the name of the branch we're on (master; again, more on this soon), the SHA-1 hash for the commit (see below), and our commit message.
- The next line tells you what's been done in this commit; you can see that three files were changed in this commit, with 10 lines added and none removed.
- Finally, we see that Git created our three files; of course, we made the files, but Git created a record of them in that `.git` folder.

Let's go back to that SHA-1 hash for a minute. SHA-1 is a cryptographic hash function; basically, Git takes the content of the commit and creates a 40-character hash from it. Without getting too technical, I'll note that the chances of creating two identical hashes are close enough to impossible. These are used to identify commits (and a number of other things, too) in Git. What you see above is just the first portion of the hash. Often, just the first seven characters are enough for identification.

We can see more information about our commit by running the `git log` command:

```
$ git log
commit 80f84da53585307947546188cd0540a6ff63f64f
Author: Andrew Burgess <andrew8088@gmail.com>
Date:   Wed Jun 30 14:51:43 2010 -0400
    initial commit
```

Now you can see the complete 40-character hash. You can also see the author (Git is using those config settings), the date and time I made the commit, and the commit message. As you make more commits, this command will output this information for all

your commits, descending from newest to oldest. The name says it: it's a log of your commits.

There's much more to discuss about the **`git commit`** command; we'll come back to it in chapter 4.

Ignoring Files

Let's take a break from Git commands to look at another feature: the **`.gitignore`** file. As we've seen, Git automatically notices other files in our directory, but doesn't automatically add them to the staging area or repository. Let's add a "config.php" file to our project; assume that this has some private information — say, the usernames and passwords to our server and database — that we don't want to get out. However, as we'll see, Git repositories are made to pass around; and really, anyone else using our project should create their own config file. But if we run **`git status`**, we get this:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   config.php
nothing added to commit but untracked files present (use
"git add" to track)
```

It will be very annoying to have to remember to leave **`config.php`** out whenever we stage a commit. Therefore, we'll use a neat trick of Git's. In your project directory, create a file called **`.gitignore`**. Then, simply put the name of the file we want Git to ignore in that file.

```
# file named .gitignore

config.php
```

It's that easy. Save and close the file. Now if we run **`git status`**:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   .gitignore
nothing added to commit but untracked files present (use
"git add" to track)
```

You can see that Git has found the **`.gitignore`** file and is following the rules. I'm going to add a few other lines to the file:

```
config.php
#Mac OS X files
.DS_Store

#Vim leave-behinds
*.swp
```

Notice the structure: every item goes on its own line. I can leave comments by beginning the line with a hash. I can match individual files, or use a wildcard to match a group of files.

Branching

One of the strongest features of Git is its ability to branch. What's branching? Imagine this: you're working on a project and you have an idea for a new feature. You don't want to risk messing up the

project; so, you create a new branch on your Git repository to work on the feature. When it's done and ready, you can merge your changes into the main line. Basically, branching creates a safe sandbox (identical to the main project when you start) for you to play in, where you don't have to worry about hurting other things.

So far, our project has only had one branch: it's the default branch, and it's usually called the master branch. You can see the branches you have by using the command git branch.

```
$ git branch
* master
```

ROCK* TIP

If you want to exclude all but one of a certain type of file, the bang (`!`) is your friend. Just do something like this in your `.gitignore` file:

```
*.log
!errors.log
```

Starting the line with a bang tells Git to be sure to include whatever comes next; so in this case, all the log files except for `errors.log` will be ignored.

An asterisk will precede the name of the branch we're on. So, let's create another branch. We do that with the **`git branch`** command:

```
$ git branch authorization
```

We've just created a branch named "authorization." Now if we run **`git branch`**, we'll see it.

```
$ git branch
  authorization
* master
```

But we're still on the master branch. So, let's switch to our new branch. We can do that with **`git checkout`**:

```
$ git checkout authorization
Switched to branch 'authorization'
```

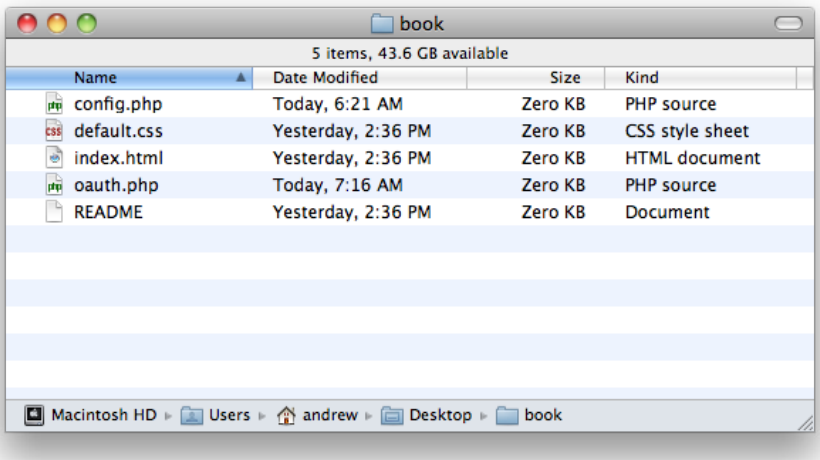
Now, let's create a new file, say "oauth.php". I'll stage it and commit it.

```
$ git status
# On branch authorization
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   oauth.php
nothing added to commit but untracked files present (use
"git add" to track)

$ git add oauth.php

$ git commit
[authorization 8cdf227] added authentication
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 oauth.php
```

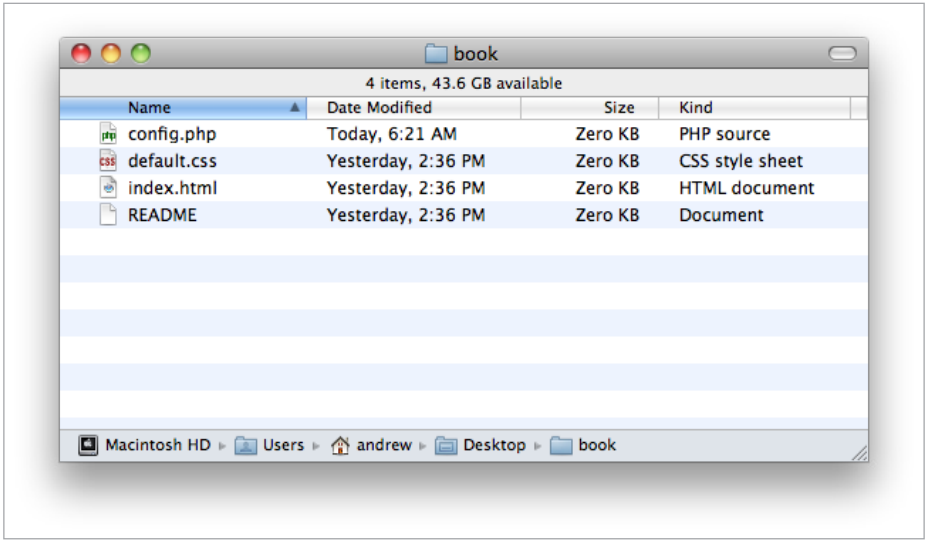
Now watch this: here's proof the "oauth.php" file really exists:



Now, I'll switch back to the master branch:

```
$ git checkout master
```

And now, look at our project:



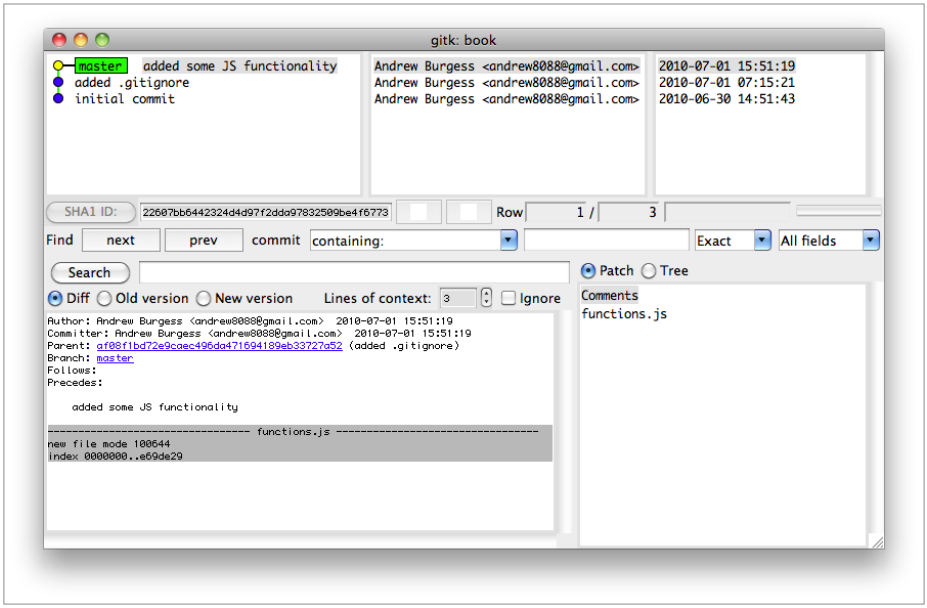
Yes, that file is completely gone; that's part of branches: whatever changes were made on one branch will not be visible at all from the other branches. Since Git tracks contents, this works for files and whole directories down to lines and characters.

There are a few ways to visualize branching; to give us something to see, I'm going to make two more commits, one on each of the branches we've created.

The first visualization tool Git comes with is a GUI program called gitk, and we'll look at that now. To open it, just run the command:

```
$ gitk
```

This will open another window; this is what it looks like:



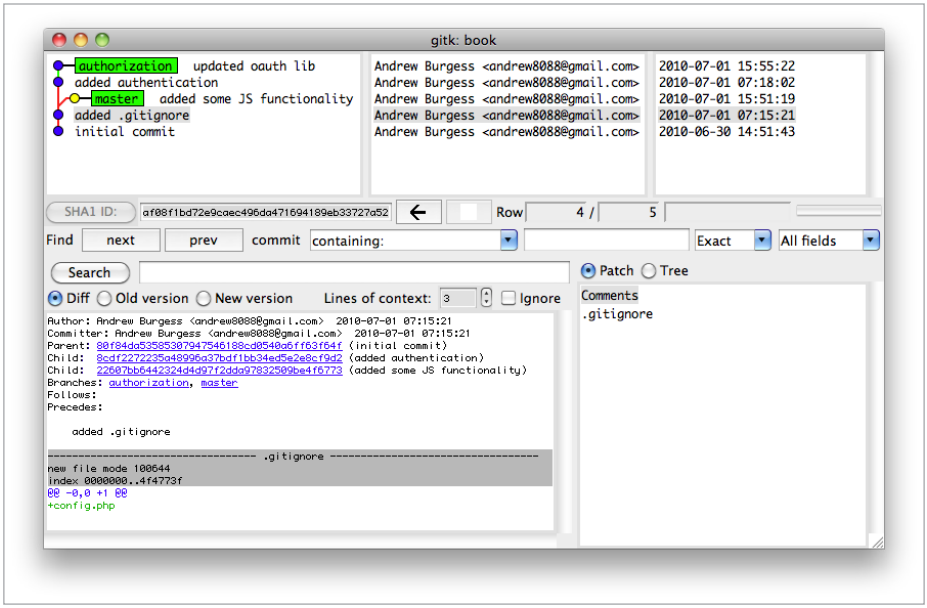
In the top left corner, you can see a timeline, with each commit represented by a dot. You can see the commit messages, too. In the top middle, you can see who made each commit and to the right, the date and time. Right now, it's just me committing, but if this was a real project, you might have several people committing at once. There are some tools in the middle. Moving down, there are some tools we won't discuss. At the bottom right, you can see more detailed commit information of the commit selected above. It tells you what commit came before (the parent commit) and what commit came after (the child commit). To the right, you can see what files were adjusted or added, either in the form of what's new (patch view) or what's there at the time of a commit (tree view).

But why can't we see our authorization branch? It's because I executed the **`gitk`** command from the master branch. If we switch to the authorization branch and do it, we'd just get that branch.

Instead, I'll execute the command again, but I'll give it the `--all` flag.

```
$ gitk --all
```

Here's what this gives us:



Now, you can see both the branches at once; you can see the relationship between them. It's pretty clear that we've made one commit on the master branch since we branched off with the authorization branch. Also, the authorization branch has two commits since creation.

There's another way to view your Git repository tree, this time without leaving the command line. We briefly used the **`git log`** command before; let's run it now to see what we get. Just like we used the `--all` flag with **`gitk`** to see all branches, we'll do the same with **`log`**:

ROCK* TIP

Command extensions like `--all` *for* `gitk` *are called flags. Think of them as the settings or options for the command line programs. On the Mac, single letter flags are usually preceded by a single dash (like* `ls -l`*) and word-length flags are usually preceded by a double dash (like* `gitk --all`*). On Windows, all flags are preceded by a backslash (like* `dir /B`*). Most of the time, single-letter commands are an abbreviation, which helps you remember them. For example, the lowercase* `l` *flag for the* `ls` *command stands for "long form." We'll look at more flags for Git commands in Chapter 4.*


```
$ git log --all
commit 39a1b50c88f7bf2696b95b08ace27711b652a5c9
Author: Andrew Burgess <andrew8088@gmail.com>
Date:   Thu Jul 1 15:55:22 2010 -0400

    updated oauth lib

commit 22607bb6442324d4d97f2dda97832509be4f6773
Author: Andrew Burgess <andrew8088@gmail.com>
Date:   Thu Jul 1 15:51:19 2010 -0400

    added some JS functionality

commit 8cdf2272235a48996a37bdf1bb34ed5e2e8cf9d2
Author: Andrew Burgess <andrew8088@gmail.com>
Date:   Thu Jul 1 07:18:02 2010 -0400

    added authentication

commit af08f1bd72e9caec496da471694189eb33727a52
Author: Andrew Burgess <andrew8088@gmail.com>
Date:   Thu Jul 1 07:15:21 2010 -0400
```

```
    added .gitignore

commit 80f84da53585307947546188cd0540a6ff63f64f
Author: Andrew Burgess <andrew8088@gmail.com>
Date:   Wed Jun 30 14:51:43 2010 -0400

    initial commit
```

So we've got information from all our commits, but where's the tree view? Try this:

```
$ git log --graph --all --oneline
* 39a1b50 updated oauth lib
* 8cdf227 added authentication
| * 22607bb added some JS functionality
|/
* af08f1b added .gitignore
* 80f84da initial commit
```

I've added two flags here: **`--graph`** is the one that gives us the tree view; as you can see, it's very similar to **`gitk`**. What about the **`--oneline`** flag? That just reduces the amount of information about each commit that we see. It just shows the first seven characters of the hash and the commit message. You don't have to use **`--graph`** and **`--oneline`** together. I've just done so to take up less space.

So we've talked about branching your code to work on separate ideas. Let's say we now want to bring those two branches together. In Git, this is called merging. Let's see how it's done.

This is the simplest way. Note this: I'm on the master branch.

```
$ git merge authorization
Merge made by recursive.
 oauth.php |     7 +++++++
```

```
 1 files changed, 7 insertions(+), 0 deletions(-)
 create mode 100644 oauth.php
```

Git spits out a bit of information about the merge. Let's look at our commit log again:

```
$git log --graph --all --oneline
* 08eaf7c Merge branch 'authorization'
|\
| * 39a1b50 updated oauth lib
| * 8cdf227 added authentication
* | be96dbe edited index.html
* | 22607bb added some JS functionality
|/
* af08f1b added .gitignore
* 80f84da initial commit
```

As you can see, Git has merged the two branches and committed the result. It's clear from the commit message what's happened: we've merged in the authorization branch. In most cases, merging your branches will be that easy.

Tagging

Sometimes, you'll want to point to a certain commit in your projects history. The usual example is versions: tagging is a great way to mark different versions of, say, a library or framework. Here's how you do it.

As you might guess, the command we're looking at is **`git tag`**. There are two types of tags: lightweight and annotated. Lightweight tags are like a branch that can't be updated: really, they're just a pointer. An annotated tag is more useful: it contains a lot of the information that a regular commit has.

When you're at a point in your project that you want to tag, here's what to do. For lightweight tags, it's this simple:

```
$ git tag v0.5.6
```

Notice that the tag name is a parameter. However, it's often smarter — but no harder — to use annotated tags:

```
$ git tag -a stable-1
```

If you want to add a message to the tag, use the **`-m`** flag.

```
$ git tag -a RC1 -m 'the first release candidate'
```

So, now that you've tagged your project, what can those tags tell you? First, running **`git tag`** without any flags or parameters lists your tags. Here's our little sample project:

```
$ git tag
v0.5
```

You can see information about a tag by running **`git show`** and passing it the name of the tag:

```
$ git show v0.5
tag v0.5
Tagger: Andrew Burgess <andrew8088@gmail.com>
Date:   Sat Jul 3 10:06:16 2010 -0400

half-way to release 1

commit 08eaf7c6b31157d0d5473b84abbbf265bc12cee5
Merge: be96dbe 39a1b50
Author: Andrew Burgess <andrew8088@gmail.com>
Date:   Sat Jul 3 07:28:16 2010 -0400

            Merge branch 'authorization'
```

You can see who made the tag, when they did, and what message they left with it. You can also see what commit it points to.

I've gone ahead and made two more commits on this project. Now, I'll run the `git describe` command:

```
$ git describe
v0.5-2-g8ee0f4a
```

This tells us that, since the tag "v0.5", we've made two commits; it also give us the hash of the most recent commit. Note that the "`-g`" is not part of the commit hash; it's a suffix that stands for "Git". According to the Git documentation, it's "useful in an environment where people may use different SCMs."

Referencing Commits: Git Treeishes

Quite often in Git, you'll need to give a reference to a commit as a parameter to a command; this commit pointer is called a *Treeish*. Here are some of the ways you can do this:

- **The SHA-1 hash:** using the hash is the surest way to point to the commit you want. You don't have to use all 40 characters; usually the first 5-7 characters are enough to distinguish it.
- **A Branch, Tag, or Remote Name:** Using a branch name will point to the latest commit on that branch. Using a tag name will point to the commit that the tag points to. We'll learn about remotes in the next chapter, but you can use them, too.
- **HEAD:** in Git, `HEAD` is an alias for the latest commit on the currently checked out branch.

- **A Date Spec:** you can append a date spec to the end of a branch name to get the commit that was last at a given time. For example, to see where the feature branch was four days ago, you would do this: `feature@{4 days ago}`.
- **An Ordinal Spec:** an ordinal spec works just like a date spec, except that it returns the nth previous commit on a branch. To get the third-last commit on feature, do this: `feature@{3}`.
- **Carrot Parent:** the carrot parent will modify a reference to a merge commit to point to one of its parents. So, if `HEAD` is a merge commit, `HEAD^` will refer to the parent of that commit on the same branch. `HEAD^2` will refer to the commit's other parent (the commit on the other branch).
- **Tilde Parent:** this works just like the carrot parent, except that it points to the grandparent of a commit. So `HEAD~` refers to the parent of the parent of a commit; `HEAD~2` refers to the great-grandparent commit.

Summary

We've covered a lot in this chapter: everything from setting up Git on your computer to branching and merging your projects. You might be overwhelmed, but don't give up yet! In the next chapter, we're going to learn about the more advanced commands of Git, as well as look at some of the deeper functionality of the commands you've already met!

4

Beyond the Basics

So you're ready to move on? Good! Several of the commands in this chapter you've already used, but I'll show you how to make them more efficient through some command flags. We'll also look at working with repositories stored on remote computers, and so much more.

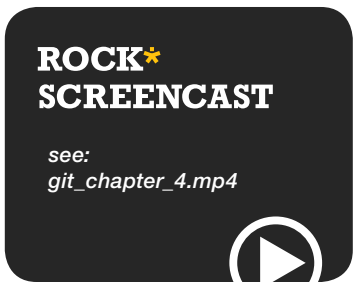
Git Add Revisited

We'll start by coming back to the command `git add`. It may seem like a relatively simple command — how complicated can it get when adding files to your staging area? — but there's some hidden functionality lurking behind a couple of flags.

Interactive Add

Often, you'll want to get just the right code into a commit. Of course, this starts with putting just the right code into the staging area to commit! One nice way to do this is with Interactive Add.

I've made changes to two of the files in our project, as well as added another file. Now, we'll look at crafting them into a commit with Interactive Add. To get into the Interactive Add console, use the `-i` flag.



```
$ git add -i
           staged    unstaged path
  1:    unchanged       +3/-0 default.css
  2:    unchanged       +1/-2 index.html

*** Commands ***
 1: [s]tatus  2: [u]pdate  3: [r]evert  4: [a]dd untracked
 5: [p]atch   6: [d]iff    7: [q]uit    8: [h]elp
What now>
```

So, what do we see here? At the top, you see what's changed since the last commit. You can see the two files I've changed. Interactive Add doesn't automatically see currently untracked files. This is the status view of our working directory.

To execute commands within this view, you type either the number of the command, the letter of the command (enclosed in square brackets) or the whole command word. Let's look at each of the commands.

Status

The `status` command shows you the current status of your staging area and working directory. It is what's first printed out when you start Interactive Add. You've got four columns. The first gives each file a number for easy reference. The second shows you what has been staged, the third shows you want has not been staged (what is still in the working directory). The last column gives you the path of the file.

ROCK* TIP

What are those numbers in the "unstaged" column? You'll see them in multiple areas of the Interactive Add console. They represent the number of lines changed in the given file. The number following a plus sign represents the number of lines added to the file. The number following a minus sign represents the number of lines deleted.

Update

The second command is **update**. This does the same thing as a regular **git add** with a file: it simply adds the file to the staging area. When you hit **2** (or **u**) and press 'Enter,' you'll get a prompt like this:

```
             staged    unstaged path
  1:    unchanged       +3/-0 [d]efault.css
  2:    unchanged       +1/-2 [i]ndex.html
Update>>
```

Now you're in a "sub-prompt" for adding files to the staging area. Currently, neither of these files are in the staging area (as you can see from the "staged" column). Just type the number or letter of the file and hit 'Enter' to stage it. You can also use ranges, such as **1-4** to add files 1, 2, 3, and 4. If you precede a file or range of files with a minus sign, it will remove them from the staging area. Files that have been added to the staging area will have an asterisk before them. Here's how I would add **default.css** to the staging area:

```
             staged    unstaged path
  1:    unchanged       +3/-0 [d]efault.css
  2:    unchanged       +1/-2 [i]ndex.html
Update>> 1
             staged    unstaged path
* 1:    unchanged       +3/-0 [d]efault.css
  2:    unchanged       +1/-2 [i]ndex.html
Update>>
```

Now I'll hit enter again (at the blank prompt) to exit the Update prompt. If I run status again, you'll see that **default.css** has been added to the staging area:

```
              staged    unstaged path
     1:        +3/-0     nothing default.css
     2:    unchanged       +1/-2 index.html
```

Revert

Command three is **revert**. It does exactly the reverse of what **update** does: it removes whole files from the staging area and returns them to the working directory. Really, it works exactly the way **update** does, so there's nothing more to say.

Add Untracked

I mentioned that I'd added another file to the project, but, by default, Interactive Add only shows files that are being tracked. The **add untracked** command will fix just that. So press **4** to begin.

Once you're in the add untracked sub-prompt, you'll see a list of all the currently untracked files.

```
What now> 4
  1: [c]ontact.html
Add untracked>>
```

I've only got one, in this case. Type either the letter or number of the file you want to add. As in the other commands, you'll see an asterisk beside the added files. To reverse the effect, precede the number with a minus sign.

```
Add untracked>> 1
* 1: [c]ontact.html
Add untracked>> -1
  1: [c]ontact.html
Add untracked>> 1
* 1: [c]ontact.html
```

```
Add untracked>>
added one path
```

Now running the status command shows us that **contact.html** has been added:

```
What now> s
           staged    unstaged path
  1:        +5/-0     nothing contact.html
  2:        +3/-0     nothing default.css
  3:    unchanged       +1/-2 index.html
```

Patch

Patch is an interesting command. Until now, we've only staged complete files at once. But let's say we've done a lot of work at once, but we want to separate it into different commits. That's where this command comes in. Hit **p** or **5** to enter the **patch** sub-prompt.

Patch will start by listing all the files with content in the working directory (that's content that hasn't been staged). For me, that's only one file.

```
What now> p
staged unstaged path
1: unchanged +1/-2 [i]ndex.html
Patch update>>
```

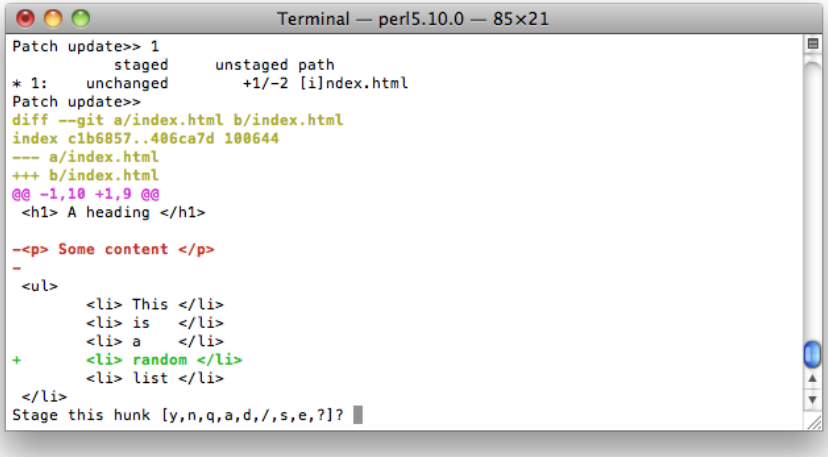
I'll start the patching by entering the number of the file I want to split up. Once you've chosen the files you want (remember, they'll have an asterisk at the front of the line) hit 'Enter' at the empty prompt. You'll see something like this:

```
Patch update>> 1
           staged    unstaged path
* 1:    unchanged       +1/-2 [i]ndex.html
Patch update>>
diff --git a/index.html b/index.html
index c1b6857..406ca7d 100644
--- a/index.html
+++ b/index.html
@@ -1,10 +1,9 @@
 <h1> A heading </h1>

-<p> Some content </p>
-
 <ul>
    <li> This </li>
    <li> is   </li>
    <li> a    </li>
+   <li> random </li>
    <li> list </li>
 </li>
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

This might be a bit overwhelming at first, but it will make more sense in your console, because Git colour-codes parts of the prompt (*see top image on page 59*).

At the top, you're told that Git is doing a diff of the current **`index.html`** and the last **`index.html`** that was committed (don't worry about what a diff is, we'll discuss it soon). After some hashes, you're told that the lines that begin with a minus sign (`-`) are from the old **`index.html`** and the lines that begin with plus sign (`+`) are from the new **`index.html`** (lines with neither are unchanged between versions). The main middle chunk is the piece of the file in question. Hopefully you can see how, since the last commit of this file, I removed two lines and added another one. At the very bottom, were asked if we want to stage this hunk of code. Those



characters in the square brackets are our options, the answers to the question. You can hit **?** to understand each one in more detail:

```
y - stage this hunk
n - do not stage this hunk
q - quit, do not stage this hunk nor any of the remaining
ones
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks
in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

The **s** command is for splitting hunks into smaller ones. Let's look at how that works. I'll tell Git to split out the hunk above into smaller pieces. It should be obvious where it will split:

```
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Split into 2 hunks.
@@ -1,8 +1,6 @@
 <h1> A heading </h1>

-<p> Some content </p>
-
 <ul>
    <li> This </li>
    <li> is   </li>
    <li> a    </li>
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]?
```

Git split our hunk into two hunks and is presenting us with the first one. This includes the two lines I removed. I want those lines removed for this commit, so I'll choose **y**. I'll choose **n** for the last patch.

Now I'm out of patch mode; I'll run the **`status`** command to see what we've got:

```
What now> s
           staged     unstaged path
  1:        +5/-0      nothing contact.html
  2:        +3/-0      nothing default.css
  3:        +0/-2        +1/-0 index.html
```

As you can see on **`index.html`**, I've staged the removal of two lines; the addition of one line has been left unstaged, on the working directory. Now I'll leave Interactive Add (with command **7** or **q**) and run a regular **`git status`**:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:  contact.html
#   modified:  default.css
#   modified:  index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be
committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#   modified:  index.html
#
```

As you can see, the **index.html** file is both in the "Changes to be committed" and "Changed but not updated" sections.

Diff

There's one more command inside Interactive Add that we want to cover: **diff**. Let's jump back into the Interactive Add and check it out. There's not much to do in this command. It just shows you the differences between your latest commit and the files in the staging area. When I enter the **diff** sub-prompt, I'm prompted to choose a file from the staging area. Git automatically shows you the changes, in the same way the **patch** command does. Then, it returns you to the Interactive Add console. Here's the diff for my **default.css** file:

```
What now> d
           staged    unstaged path
  1:        +5/-0     nothing [c]ontact.html
  2:        +3/-0     nothing [d]efault.css
  3:        +0/-2       +1/-0 [i]ndex.html
Review diff>> 2
diff --git a/default.css b/default.css
index cafd94f..2add0a5 100644
--- a/default.css
+++ b/default.css
@@ -1,3 +1,6 @@
 body {
       font-size:100%;
 }
+h1 {
+      font-weight:bold;
+}
```

Quit and Help

I think you can figure out what commands **7** and **8** do on your own!

So, that's the Interactive Add console. It's very powerful, and gives you fine-grained control over your commits.

Add with Patch

We can use the patch behavior that we saw in the Interactive Add console directly from the normal command line with **`git add`**'s flag **`-p`**. Let's try it:

```
$ git add -p
diff --git a/index.html b/index.html
index 5d24ca6..406ca7d 100644
--- a/index.html
```

```
+++ b/index.html
@@ -4,5 +4,6 @@
     <li> This </li>
     <li> is   </li>
     <li> a    </li>
+    <li> random </li>
     <li> list </li>
  </li>
Stage this hunk [y,n,q,a,d,/,e,?]?
```

This is the other piece of **index.html**, the hunk that we didn't stage in the Interactive Add console. This has all the same commands that **patch** in Interactive Add has (although the single-letter options aren't all shown). I'll stage this hunk with **y**, and then commit all our work.

Speaking of commits, did you know **git commit** includes some interesting options yet undiscussed?

Git Commit Revisited

There are a few more tricks up **git commit**'s sleeve. Let's check them out now.

Amending a Commit

Let's say you've made a commit… and, right after you do, you realize that you forgot to stage a few changes in another file. Or, you forgot to add a new file to the staging area. Now is the time to use **git commit**'s **--amend** flag. Here's how it works: after the erroneous commit, move whatever changes you forgot to make to the staging area. Then, run this:

```
$ git commit --amend
```

What this does is add the content of the staging area to the latest commit. You'll then be asked for a commit message. This commit and its message will override the erroneous commit, so all that you see is the edited commit. (Actually, it's not editing the commit — it's removing the old one and creating a new one. You'll see they have difference hashes.) Now, there's only one, perfect commit.

ROCK* TIP

If you want the commit message and data to be the same as the erroneous commit (or another commit), add the `-C` *flag*

```
$ git commit --amend
-C HEAD
```

The `-C` *flag takes a pointer to a commit as a parameter.* `HEAD` *refers to the latest commit that was made on the current branch. You could also use the hash of the commit you want to reference.*

Making Quicker Commits

There are two flags that will help you just git in and git out — I mean, get in and get out — faster when you're making commits. The first one is the `-a` flag. 'A' stands for 'all'. In fact, you could use `--all` instead if you wanted to. This flag simply automatically commits all the currently tracked files. There's no need to move them all to the staging area. Just throw in the `-a` flag and you're done. Of course, this is most efficient when you haven't created any new files: Git will auto-stage all the files in the working directory and wrap them into a commit.

The second shortcut flag is `-m`. This is followed by a quoted string, which is — wait for it — the commit message. You can also use the format `--message='message here'` if you want.

Putting these two flags together will really save you some time. Instead of staging all your files, running `git commit`, and writing your commit message in an editor, you could simply run this:

```
$ git commit -am 'this is the commit message'
```

Git Reset

Occasionally, you might find yourself wanting to undo something you did in Git. Usually, **`git reset`** is the way to go. Right off the top, I'll tell you that **`git reset`** can be very complicated, depending on what you want to rewind, and to where. If I don't cover something you want to do, check out the official documentation for git reset.

So let's say first that you're working on a project and you want to clean out the changes you've made and go back to most recent commit. Here's what to do:

```
$ git reset --hard HEAD
```

The **`--hard`** flag puts the content of whatever is in the targeted commit (in this case, **`HEAD`**) into both the working directory and the staging area. Since **`HEAD`** is the most recent commit, it's as though we've made no changes since last committing. Running **`git status`** will tell us there's nothing to commit.

There are other flags to use with **`git reset`**. Let's say you want to rewind your project to several commits back, but you want to keep the changes you haven't committed yet. Try this:

```
$ git reset --soft 3ce072c72d948abfa
```

Of course, replace my hash with your own. The **`--soft`** flag will leave whatever changes you've made in your working directory and staging area, but will restore **`HEAD`** to the commit you've chosen. In other words, the most recent commit will now be the commit you chose.

There are many other situations in which you can use **`git reset`**. If you think you might need some of the other functionality of this

command, check out the documentation for a comprehensive table.

Git Checkout Revisited

We already looked at using the **`git checkout`** command: it's the tool for switching between branches. But there's more than meets the eye. You can add the **`-b`** flag to create a branch and switch to it all at the same time:

```
$ git checkout -b newBranchName
Switched to a new branch 'newBranchName'
```

Also, **`git checkout`** can also do some resetting work. While **`git reset`** rolls back full commits, you can use **`git checkout`** to roll back individual files. To roll back to the version of the file in the staging area, use this:

```
$ git checkout -- file.ext
```

If you want to rollback to the version of a file in a commit, replace those two dashes with a pointer to the commit:

```
$ git checkout HEAD file.ext
$ git checkout master~2 file.exe
```

Git Diff

Another really useful Git command is **`git diff`**. You'll use this command to find out what has changed between two files, commits, or branches. Let's say you want to see what changes have been made to your working directory since it was last committed. You would do something like this:

```
$ git diff
```

If you wanted to see the changes in just one file, add that filename as a parameter:

```
$ git diff index.html
```

If you want to compare what's in the staging area with the latest commit, add the flag `--cached`.

```
$ git diff --cached
$ git diff --cached index.html
```

To compare your working directory with a commit, use a pointer to that commit instead of the `--cached` flag.

```
$ git diff be96dbeab1
$ git diff HEAD^2 index.html
```

You can also use `git diff` to see the differences between branches. Just put the two branch names as parameters, with two dots between them:

```
$ git diff master..otherBranch
```

This will give you a diff of the latest commits on each branch. To get the diff of **otherBranch** and it's closest ancestor on master put in three dots. Of course, fill in your own branch names.

```
$ git diff master...otherBranch
```

Here's the tree view of our sample project:

So, **git diff master..authorization** will return us the differences between the commits "edited oauth function" and "other commit." Doing **git diff master...authorization** (notice the third dot) will return the differences between "edited oauth function" and "updated oauth lib."

Git Stash

Here's an interesting scenario: let's say you're working on a feature and you find a bug in your code. Since the bug is unrelated to your feature, you'd like to fix it before going on with your feature. But what do you do with the changes you've made for your feature? It's **git stash** to the rescue!

git stash is almost like a temporary commit. When you want to stash away some changes, first add those changes to the staging area. Then, do this:

```
$ git stash
Saved working directory and index state WIP on master:
3d0b0a4 other commit
HEAD is now at 3d0b0a4 other commit
```

You'll get a message similar to this one, telling you that your work in progress on the particular branch was stored.

After you commit a bug-fix, you'll want to pull your previous changes out of the stash. To do that, run this:

```
$ git stash apply
```

This will put the contents of the stash back in the staging area. You'll get a message very similar to a **`git stash`** message when doing this.

You can stash more than one stash, if you'd like to. To see all the stashes, use the **`list`** command:

```
$ git stash list
stash@{0}: On master: started form on contact list
stash@{1}: On master: README changes
```

In this example, I have stashed two items. But notice how I've got nice stash messages? If you're stashing multiple stashes, this is a smart move. To add a stash message, use the save command when stashing:

```
$ git stash save "message here"
```

If you want to bring a certain stash into the staging area, use the stash ordinal spec, at the beginning of the entry. To bring back my README changes in the above example, I'd do this:

```
$ git stash apply stash@{1}
```

Working with Remote Repositories

So far, all the work we've done with Git is completely on our own machine. That's great. Actually, that's one of Git's strong points: several of the other version control systems require that you have network access. Git doesn't require this, but it becomes even more

powerful when you do. Let's look at some of that functionality right now.

The first thing we need is another computer to share our project with. I happen to have just that: a system running Ubuntu Lucid Lynx, with Git installed. If you don't have another computer to use, don't worry: this is exactly what GitHub will be for you. Just reading along for now will put you in good stead when we get to GitHub in the next chapter.

A Bit of Setup

Before we begin, you'll want to be sure you've set up SSH properly on both your main machine (the client) and the machine that will hold the remove repository (the server). There are many great tutorials out there for setting that up: I'd recommend you read this one on Lifehacker for getting set up on your local network. Also, the ones on GitHub will be necessary for setting up your SSH keys; they've got ones for Mac, for Windows, and for Linux. You can ignore the part about adding the keys to your GitHub account; we'll do that in the next chapter.

Git Clone

Now that you're set up, let's start. Although I mentioned that one computer will be our client and one our server, there's actually no difference. Git views all computers that have a copy of a given repository as equal. We'll begin by copying the project directory to the other computer — you can use Dropbox, zip it up and email it, or even just use a thumb drive.

I've copied my sample repository to `/home/andrew/project` on my Ubuntu machine. Now, on my main computer, I'll open a command prompt and move to the folder I want the repository to live in.

We're finally back to Git. Check out the command, and then we'll go over it, piece by piece.

```
$ git clone andrew@192.168.2.18:project theProject
```

We're using the **`git clone`** command here. This command will make a local copy of the remote project you're cloning. The next section is the remote repository we want to clone. In this case, it's on a local network. We're signing into the remote computer as the user 'andrew' and getting the repository in the folder 'project' in the user's home directory. When you're working with GitHub, they'll give you a URL to clone with: sometimes it will look similar to this one, sometimes it will be a regular http URL. Actually, this parameter is just the path to another repository. It could even be on the same computer.

The last piece of this command ('theProject') is completely optional; **`git clone`** is going to create a directory for the repository we're cloning. This parameter is the name I'd like it to give the folder. If I left this off, Git would use the remote directory's name as the name for our local repository's folder.

Git Remote

It's really that easy to get a copy of a repository from a remote computer. Now that you've cloned the repo, try running this:

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/authorization
  remotes/origin/master
```

You know the **`git branch`** command. We use it to list all the branches in our repository. This time, I've added the **`-a`** flag, which shows all local branches and remote-tracking branches. So now

the question is, what's a remote tracking branch? First, you have to learn what a remote is: a remote is simply a pointer to that remote repository that we cloned. Remote tracking branches point to the specific branches on the remote repository. This allows us to keep track of the changes made in the remote repository and integrate them if we'd like. As you can see above, I've got three remote-tracking branches: one tracks the master branch of the remote repo, one tracks its authorization branch, and one tracks its HEAD.

It's not hard to add other tracking branches (this would be useful if you shared this project with others and they made public copies of their repositories available). Just do this:

```
$ git remote add remoteName remoteURL
```

You can actually see your remotes in your Git config file. Inside your project directory, open **`.git/config`** in Notepad.exe on Windows. If you're on a Mac, run this command from your project: **`cat .git/config`**. You should see a piece that looks like this:

```
[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
    url = andrew@192.168.2.18:project
```

You can see the URL of the remote repository, as well as what remote branches I'm tracking.

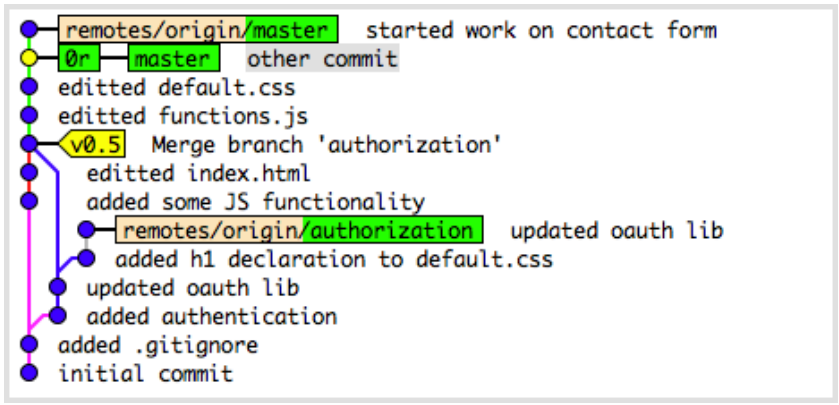
Git Fetch / Git Pull

I've made a few changes in our remote repository, and now I want to get those changes in my local repo. Here's how to do that: the first step is to fetch those changes from the remotes:

```
$ git fetch origin master
andrew@192.168.2.18's password:
remote: Counting objects: 5, done.
```

```
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From 192.168.2.18:project
   3d0b0a4..2d418e6  master     -> origin/master
```

I've called the **`git fetch`** command and passed it the name of the remote I'd like to get ("origin") and its branch ("master"). Let's run **`gitk --all`** to see a tree view of our commits:

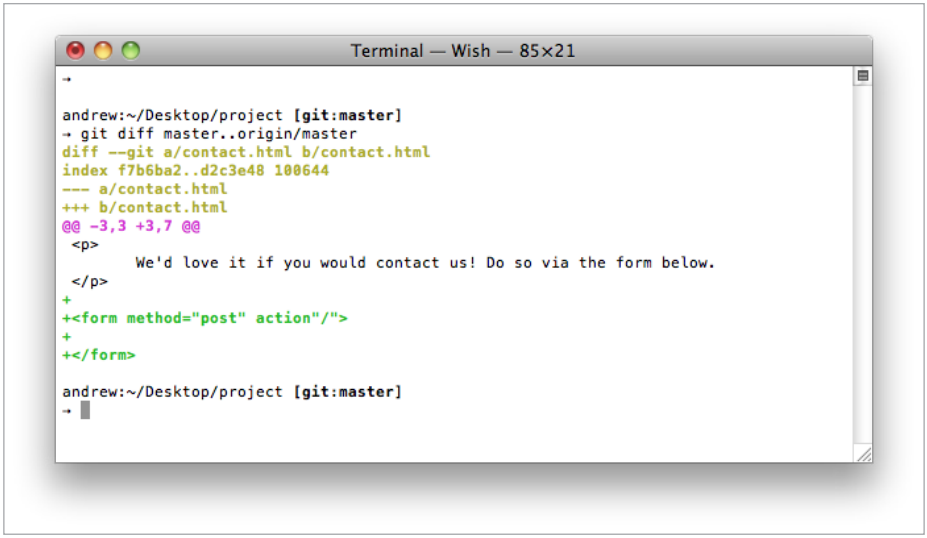


As you can see, our remote-tracking branch "remotes/origin/master" (which points to the master branch on our remote repo) has moved forward one commit. We can look at the changes in that branch by running a diff on our local master branch and the remote master branch:

```
$ git diff master..origin/master
```

The result is shown on the top of page 74.

So, let's say we like the changes that were made on the remote. How do we merge them into our master branch? It's pretty simple;



first, make sure you're on the branch you want to merge the changes into:

```
$ git checkout master
Already on 'master'
Your branch is behind 'origin/master' by 1 commit, and can
be fast-forwarded.
$git merge origin/master
Updating 3d0b0a4..2d418e6
Fast-forward
 contact.html | 4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Notice that when we checkout the master branch, Git notices that we've fetched changes that we aren't using yet, and lets us know that, in this case, we're one commit behind "origin/master." Then, it's just a simple merge operation to bring in those changes.

There is a shortcut command for this operation instead of running **`git fetch remoteName branchName`** and **`git merge remoteName/branchName`**, I could have just run this instead:

```
$ git pull origin master
```

This will fetch the changes on the remote branch that you specify and automatically merge them into whatever branch you are on. If you know that's what you want, then this works fine; but use it with caution, especially when pulling onto your master branch. When using pull, it's definitely better to pull to another branch for testing.

ROCK* TIP

Although in this example we've merged the remote changes directly with the master branch, that might not always be wise. Conventionally, your master branch should be production quality code, so you don't want to bring in untested changes. It'd be better to checkout a new branch off master (so it would be at the same place master is) and then merge the changes into that branch. Then, after testing the code, you could merge that branch into master.

Git Push

If you're using GitHub as your remote, you won't be making changes directly on your remote repository as I have in this example. You'll make changes in your local repository and then push them up to GitHub.

So let's push some changes to our remote repo. I've made a commit in my local repository. Now let's push it to the remote:

```
$ git push origin master
andrew@192.168.2.18's password:
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 389 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To andrew@192.168.2.18:project
   80bf94a..02afb67  master -> master
```

This will push whatever branch I'm on to the remote and branch I specify. In this case, I'm specifying the master branch on the remote 'origin.'

ROCK* TIP

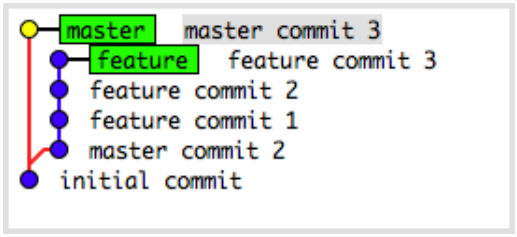
What we've just done here is really a very bad idea. For this to work properly, you have to make sure the working directory of the branch you're pushing to is clean, AND you have to be checked out on a different branch on the remote repository (or take the harder route!). So when would you use the `git push` *command? As a best practice, you should only push to bare repositories. A bare repository is one that has been cloned with the* `--bare` *flag. Basically, it only includes the information in the* `.git` *directory, so there's no working directory and you can't make changes directly to it (only through push). A bare repository is what you'll get at GitHub.*

And that's a wrap for working with remote repositories; we'll look at more of this in practice when we introduce ourselves to GitHub.

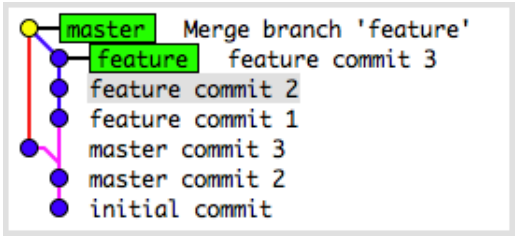
Git Rebase

We've looked at how to use **`git merge`** when bringing changes from one branch to another. There's another way to do this, which we'll look at now: it's called **`git rebase`**.

`git rebase` can be somewhat complicated, but we'll go over its standard functionality. Imagine this scenario: you've made a branch for a feature and applied several commits to it (this could also be a tracking branch you've fetched). You've also made commits on the master branch after branching, so you're history looks like this:



You're ready to bring the changes from the feature branch onto the master branch. So far, we've done this using **`git merge`**. If we merge the feature branch with the master branch (**`git merge feature`**), this is what we'll get:



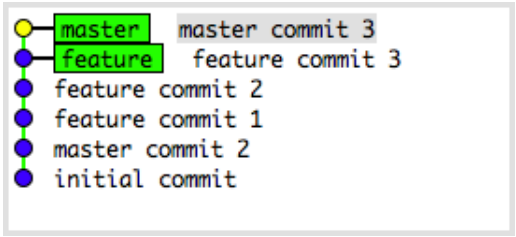
You might think this looks rather ugly, or you might want the changes you made on the feature branch to be applied before the master branch commits made since the feature was branched. In this case, you'd do a rebase.

```
$ git rebase feature
```

If any conflicts arise, open the files with the conflicts and make the desired changes (Git tells you which files have conflicts and puts the two conflicting code hunks one below the other in their file, surrounded by separating lines, so you'll know where they are.). Then, run this:

```
$ git rebase --continue
```

That will finish off the rebase. After that, your commit history will look like this:



As you can see, rebasing doesn't introduce another commit as merging does.

Summary

Now you know some of the more advanced Git commands and how to use them. Really, there's so much more that you can do with Git. For more resources, check out Appendix A. A note of encouragement: some of the commands and concepts behind Git can be rather hard to grasp. Don't be disappointed if you don't completely understand them the first time around. Read the hard chunks a few times, and check the sites and ebooks in Appendix A for other articles and screencasts about particular commands or concepts. I found that reading/watching a few different perspectives usually helps clear up the confusion. And of course, make sure you try out all the Git commands; there's no better way to learn them than to take 'em out for a spin yourself!

5

GitHub

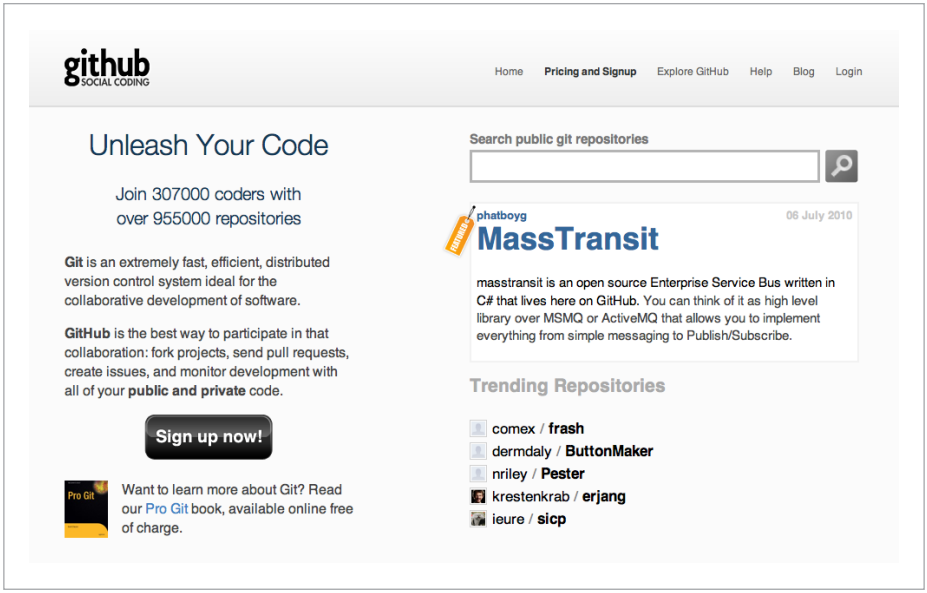
You've seen how well you can keep track of the history and current position of a project by using Git. This power is amplified when you can share the project with others. You've seen how to connect with a remote repository and use it as a central place for your projects. Of course, you could set up a public server for sharing your projects, but why would you? GitHub has already done all the hard work for you. In this chapter, we'll look at how to use the tools GitHub offers.

What is GitHub?

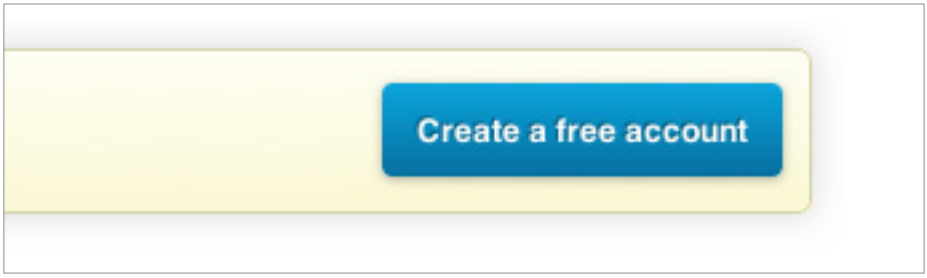
GitHub is a social network for developers; it's a place to share your project and help others with theirs. Technically, your GitHub repositories will be bare repositories that you can push to and pull from. Let's see how it all works!

Signing Up

Of course, you'll first need a GitHub account. Let's head over to GitHub.com. Here's what you'll see:



Feel free to tour around the site. Once you get an idea of what's there, hit the *Pricing and Signing* button at the top of the home page. As you'll see, GitHub has quite a few paid plans. For now, the free plan will be more than enough for us. That's pretty near the top: just click "Create a free account."

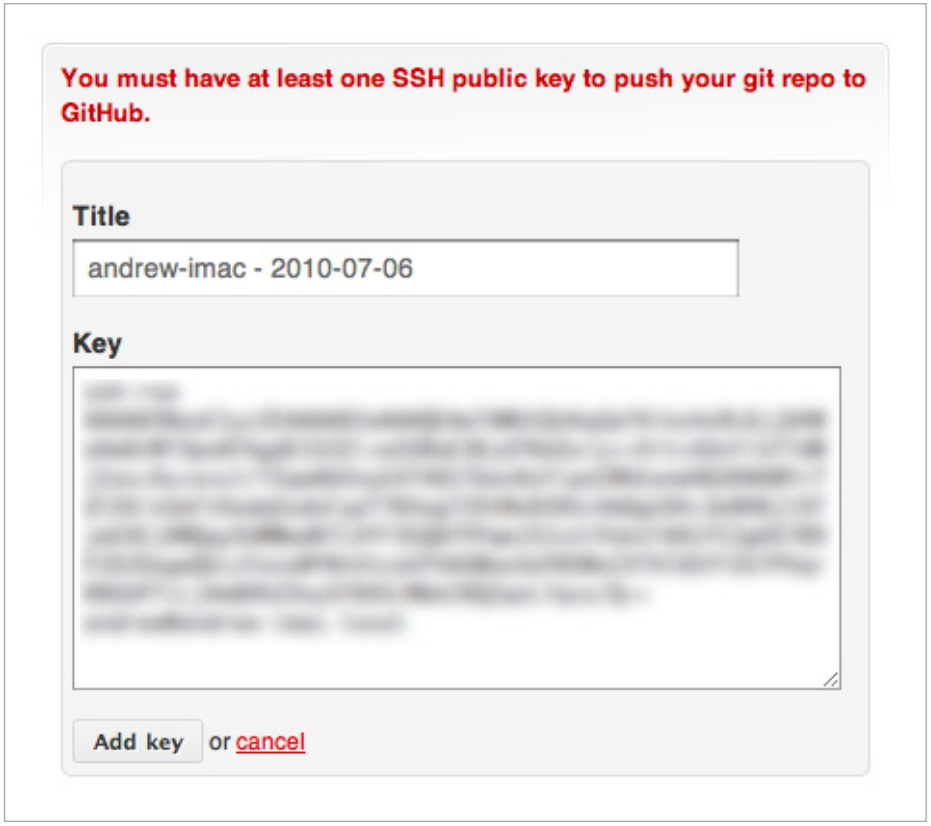


Fill in your desired user name, email, and password. Then, you can click "Create my account."

You'll be taken to your GitHub dashboard. It looks rather bare right now, but we'll fix that soon! Right now, let's head over to your

account settings (the link is near the top right corner). You can poke around in here, but right now we want to connect GitHub to our local computer. If we want to push to our forthcoming GitHub repositories, we'll have to give GitHub our SSH key. If you followed the instructions I linked to in Chapter 4 (here they are again: Mac, Windows, Linux), you should have created an SSH key. Let's give that to GitHub.

Your SSH keys folder is named `.ssh`, which is in your home (on Windows, user) folder. Inside that folder, you should have a file called `id_rsa.pub`. You want to copy the contents of this file. To do this from the command line, run the command `cat ~/.id_rsa.pub` (if you're on Windows, be sure to do this from



the Git Bash). It will print it out to the command line for you to copy. Then, go back to the GitHub account settings and click “SSH Public Keys” on the left. Click “Add Another Public Key”, then type in a title (this is to help you distinguish your keys, since you can have more than one) and paste in your SSH key.

Don’t forget to click “Add Key”! You can assure that your key will work by running this:

```
$ ssh git@github.com
Identity added: /Users/andrew/.ssh/id_rsa (/Users/andrew/.
ssh/id_rsa)
PTY allocation request failed on channel 0
ERROR: Hi andrew8088! You've successfully authenticated,
but GitHub does not provide shell access
                Connection to github.com closed.
```

You can’t actually use a secure shell on GitHub, so this won’t work, but we can see enough to know that we’ve successfully authenticated. Now, let’s start working with some repositories!

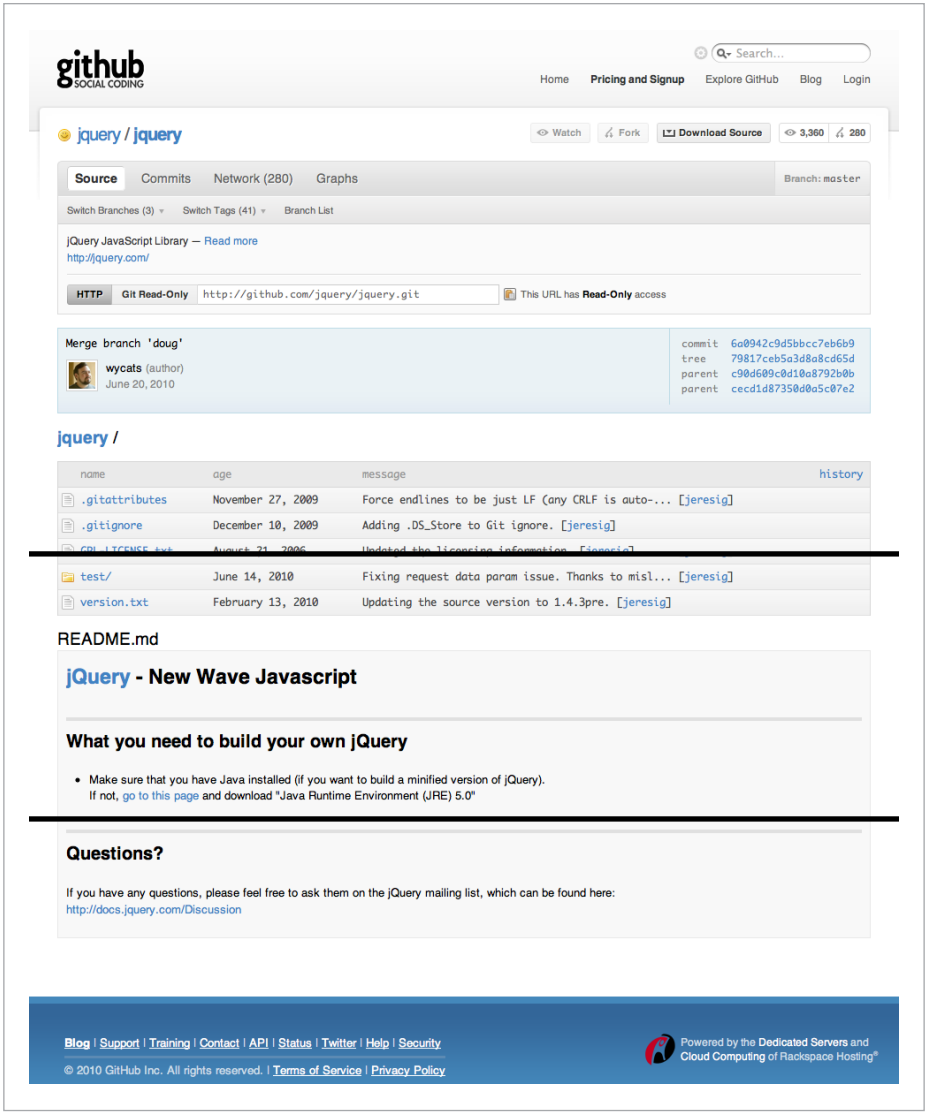
ROCK* TIP

It might get cumbersome to type in your SSH passphrase every time you want to connect to GitHub. This article (http://help.github.com/working-with-key-passphrases/) on SSH key passphrases should help!

Tour of the Tools

Let’s look at the tools GitHub offers us for working with repositories. First, we’ll look at the jQuery repository.

This is your typical repository page.
We’ll come back to the tools at the top in a minute. Notice the project description, right under the toolbars. Underneath that, you can see the public clone URL. Under that, you can see the information about the latest commit: the committer, the date, and the commit message. To the right, you can see the relevant hashes.



Still going down, we've got the file listing next (I've had to cut some out to save space). You can see the name of the file or folder, the date it was last committed, and the coinciding commit message. Underneath that, GitHub displays the project's README file, if it has one.

Now let's look at those tools. Here's a better view of the toolbar.

In the top right corner, you'll see a few command buttons that anyone can use: *Watch*, *Fork*, and *Download Source*. GitHub allows you to keeps tabs on any repository by watching it. Any commits that are made to this repo will show up in your timeline, on your homepage. If you want to get even more involved with a project, you can fork it. GitHub will create a copy of this repository in your account, so you can push to it.

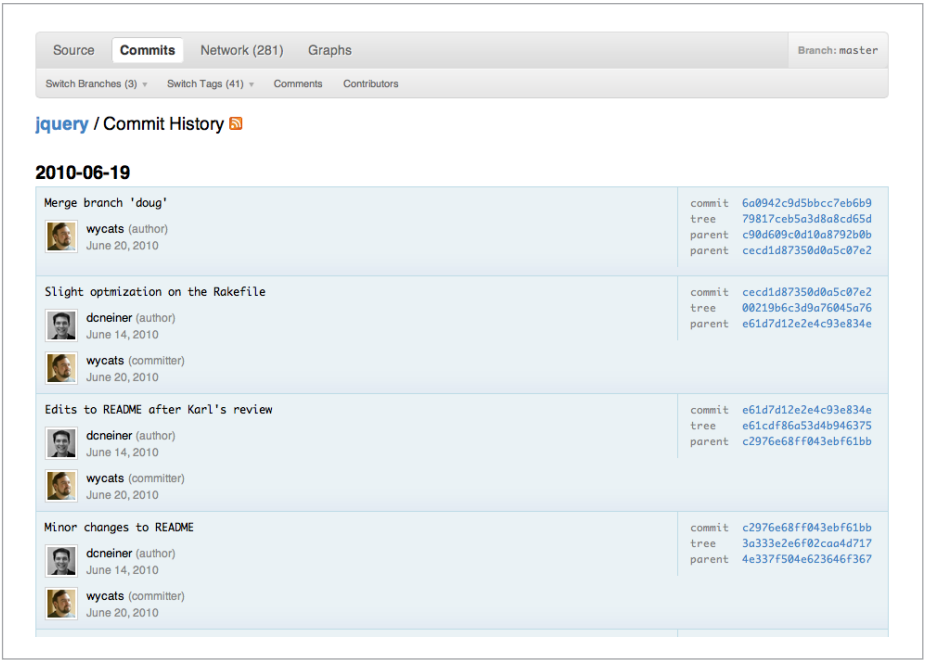
ROCK* TIP

To understand forking well, you need to understand the usual GitHub workflow. If you want to help with a project, you can't commit directly to someone else's repository, because you can't authenticate with their account. Therefore, you should fork the project on GitHub, clone your repository down, make the changes, commit them, and push them back to your public copy on GitHub. Then, you can send the original project owner a pull request (we'll see these later). If they've made changes since you forked their repository, they'll probably want you to pull in their changes before accepting yours.

You can also just download the source as a ZIP or TAR archive. You'll be able to download the project at its latest point, or at any tag. This is a great reason to tag your project at each new version.

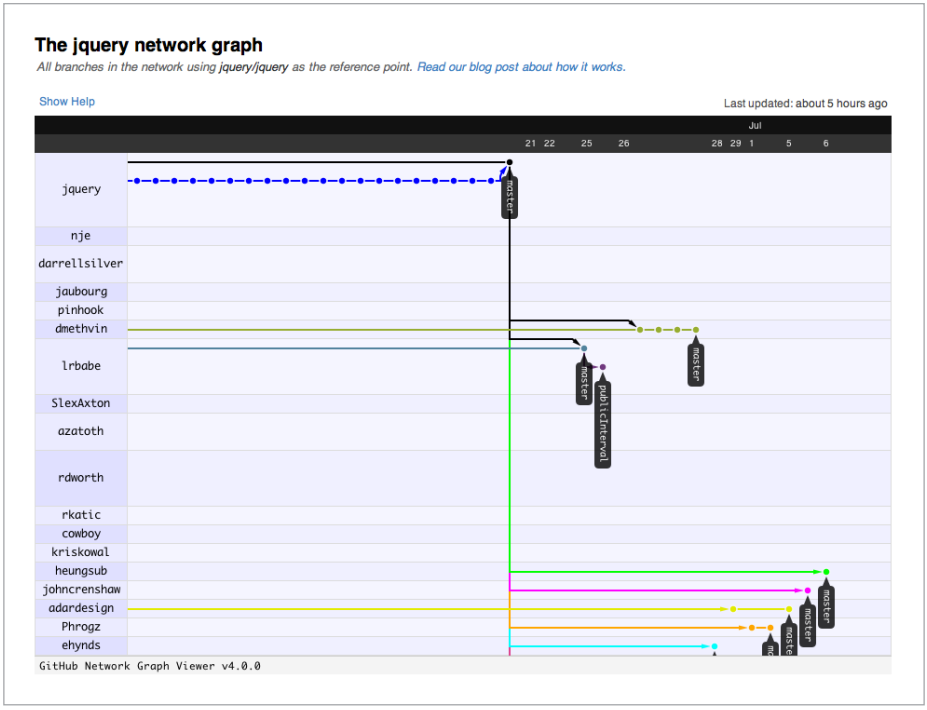
To the very right of those buttons, you can see how many people are watching this repository, and how many people have forked it.

Now we'll move to the main toolbar: as you can see, we're on the *Source* tab, where we can (obviously) view the project files and folders. The next tab is *Commits*, which will show up a commit history, vaguely similar to `git log`.

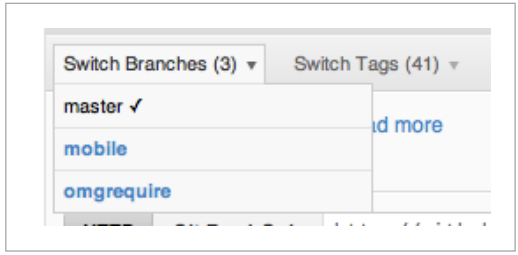


The next tab is the *Network* tab. This tab houses a graph that allows you to see what others who have forked your repository have committed. Each person will be listed on the left, then all the commits that are specific to them will be listed in line with their name. The graph is drawn from the perspective of the owner of the repository you're looking at, so that user will be the top one in the graph. The next user down will only show the commits that they have but the owner does not. The next user will only show the commits the two above him don't have… and so on. It's a bit hard to explain, but when you need it, you'll see how useful it is!

The next tab is *Graphs*, which reveals other interesting information about the repository in the form of graphs. You can find out here who's having the most impact on a project, what languages the project uses, and when most commits are made, among other things.



There's a secondary toolbar, right beneath the main one we've been discussing, but you won't use these tools too often. They're tab-specific, so move back to the *Source* tab and we'll check out just two. These are *Switch Branch* and *Switch Tags*. You can probably guess what they're for: you can use these to look at different branches or tags in a repository.



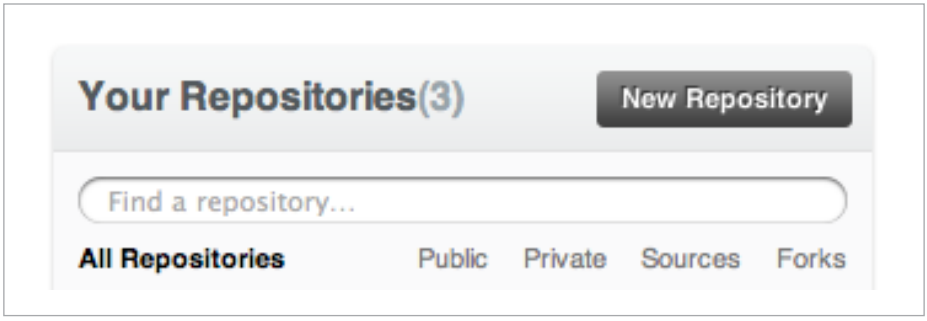
There are three other tabs on the main toolbar that are optional; they aren't really Git-related, but they're occasionally helpful. They are *Downloads*, *Wiki*, and *Issues*. These are all enabled or disabled from the repository admin panel (which we'll see when we create

our own repo). The *Download* tab will automatically create downloadable copies of your project based on your tags. The *Wiki* Tab allows you to create information pages or documentation for your project. Finally, the *Issues* tab is a lightweight issues tracker for your project.

Creating a GitHub Repository

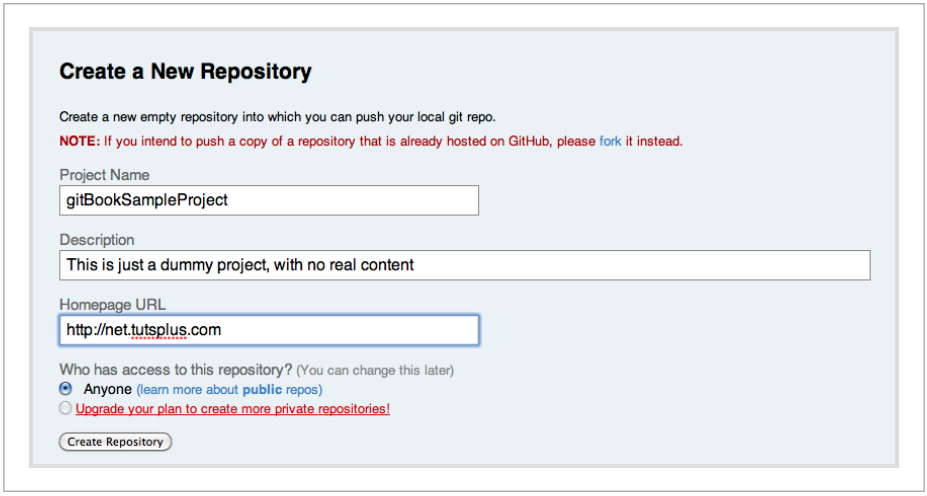
Now that we have an idea of how GitHub works, let's create a repository. We'll take the sample project that we've already got and push it up to GitHub.

The first step is to set up the repository on GitHub. I'll start by logging into my account. On the right side of your homepage, you'll see a button to create a new repository:



When you click that, you'll be taken to a form. All you really need to fill in is the project name, but it's smart to give it a description and URL, if it has one (*see top image, page 88*).

Then, GitHub will display next-step instruction; they briefly explain how to config Git and how to create a repository if you don't already have one. Since we do, we'll follow the "Existing Git Repo?" instructions.



Since you're already familiar with Git remotes and how to use them, you should understand what GitHub is telling us to do. Here are its instructions:

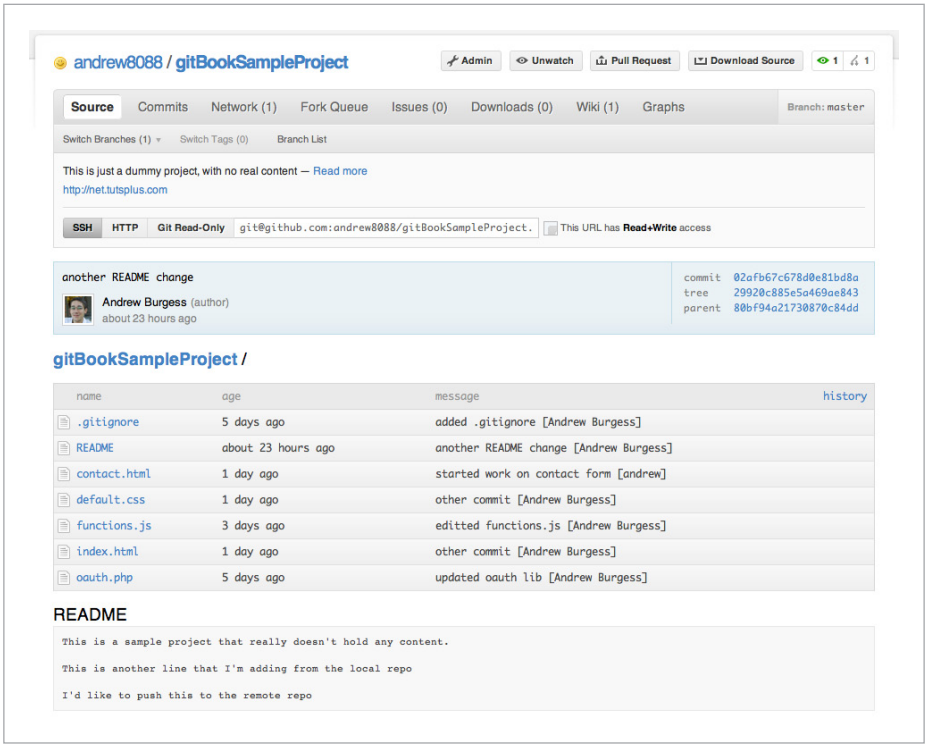
```
$ cd existing_git_repo
$ git remote add origin git@github.com:andrew8088/
  gitBookSampleProject.git
$ git push origin master
```

We need to add a remote to our project, and in this case, they're telling us to call it "origin". However, since we cloned our project from our other remote, Git automatically gave that remote the name "origin." We could change that one, but it will be easier just to change the GitHub remote. I'll run this:

```
$ git remote add public git@github.com:andrew8088/
  gitBookSampleProject.git
$ git push public master
Counting objects: 38, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (25/25), done.
Writing objects: 100% (38/38), 3.39 KiB, done.
```

```
Total 38 (delta 14), reused 24 (delta 8)
To git@github.com:andrew8088/gitBookSampleProject.git
 * [new branch]      master -> master
```

Since our GitHub repository is a bare repository, we don't have to worry about messing anything up when pushing to it. It looks like everything was successful. Let's go back to GitHub and press *Continue*. Now GitHub takes you to your repository's page, which will look something like this:

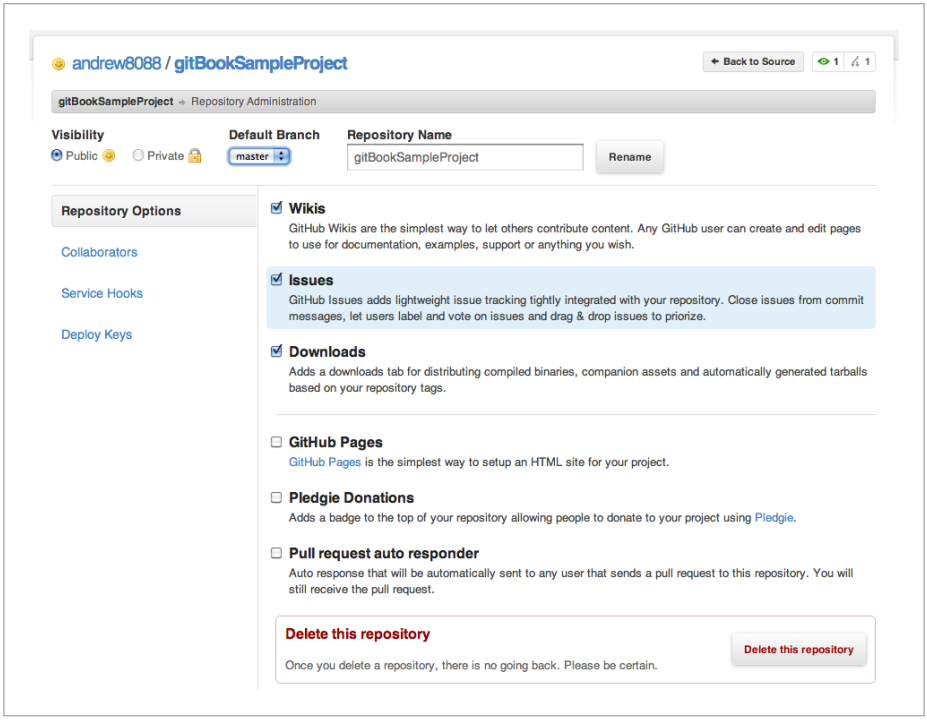


There aren't very many differences to the repository page from an owner's perspective. The main differences are:

- the URL is now readable and writable

- you can access the repository admin panel
- you can send and receive a pull request

Let's check out the repo admin panel. The button is up near the *Watch* button.



Feel free to look around here. You can enable/disable the extra tabs, as well as a few other features, such as donations and pull-request auto-responders. You can choose whether your repository should be public or private (only paid accounts can have private repos), choose your default branch, and reset your repository's name. You can also enable Service Hooks and add Deploy keys, but that gets beyond the scope of this book. If you're interested, check the GitHub help pages for information.

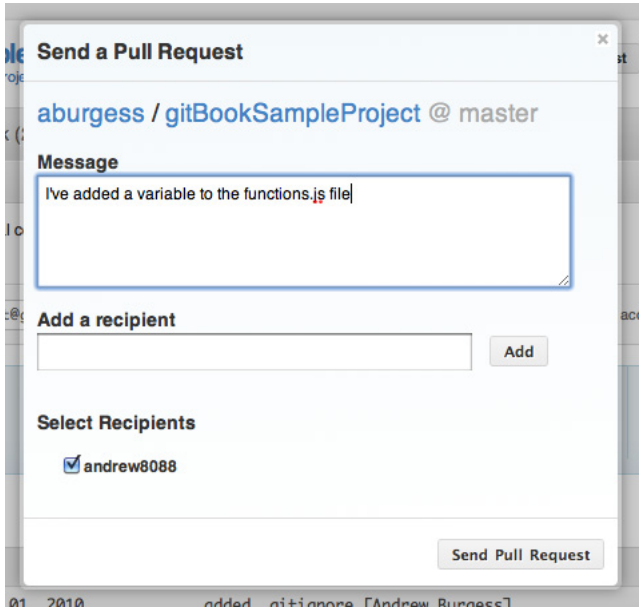
Forking a Repository

Now let's fork the project we just created; I'm going to sign in as a second user and visit the page. After forking the repository via the *Fork* button, you can see from the project title that it was forked:

I'll clone this to my computer and make a change. Now that I've committed, I'll push it back up to GitHub.

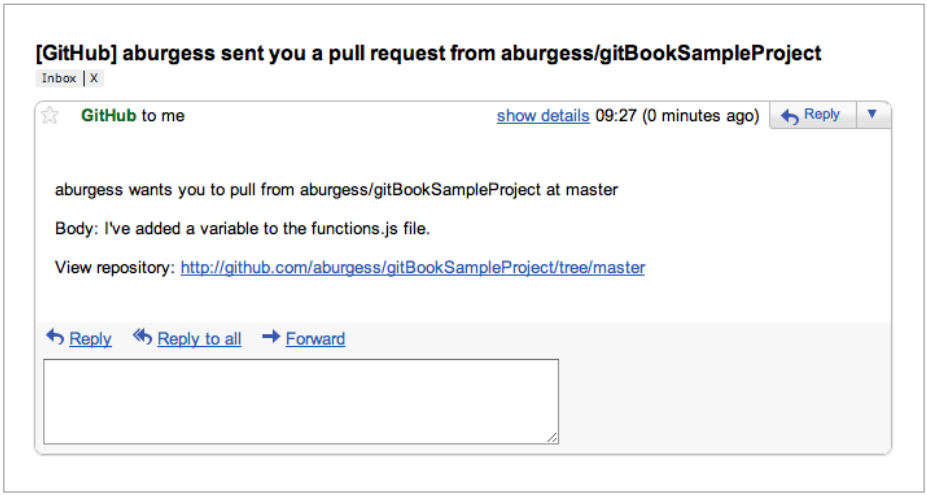
```
$ git push origin master
```

Now my changes are public in my Git repository. If I want to share the changes with my main account (the owner of the original repository), I'll have to send a pull request. The pull request button is up between the *Watch* and *Download Source* buttons.

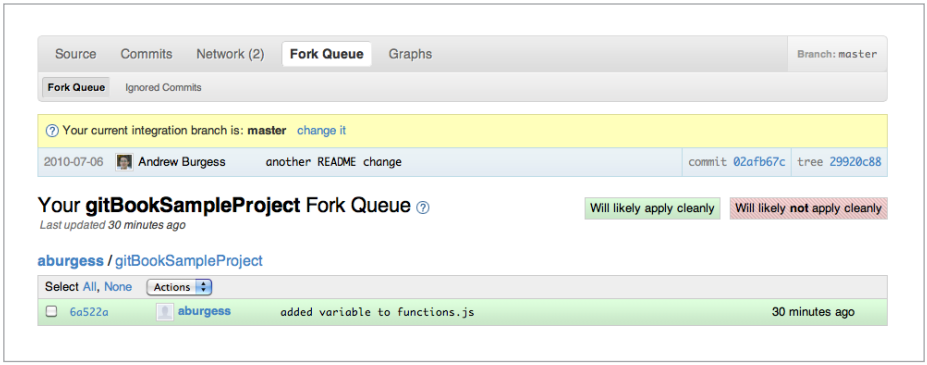


Fill in a message and select the recipients you'd like. By default, the person you forked the project from will be there, but if you want to send this request to other people who forked the same repo, you can add their usernames. Then, click "Send Pull Request".

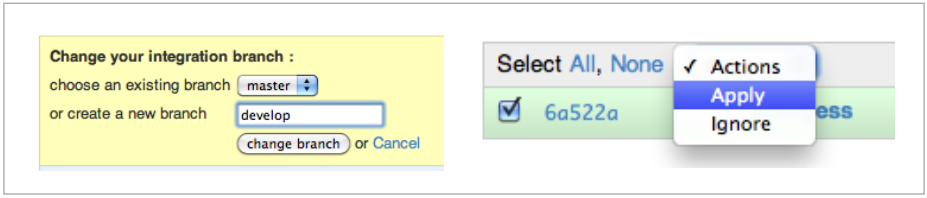
Now I'll move back to my main account. I now have an email from GitHub, telling me that I've received a pull request.



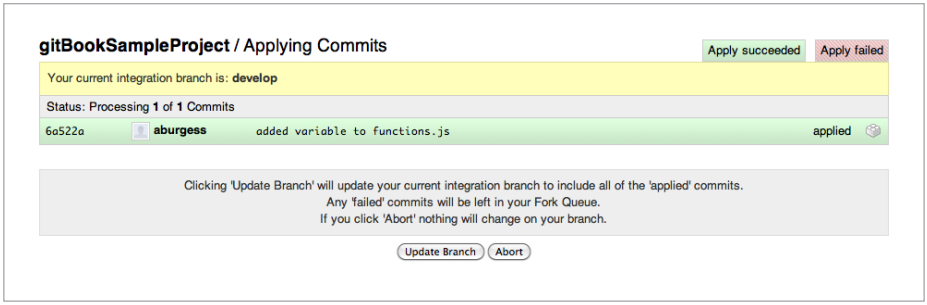
If I open my repository page, I'll see that there's a new tab at the top: *Fork Queue*. This lists all the pull requests I've received but haven't integrated.



As you can see, the pull request is highlighted green. This means that GitHub thinks the commit (or patch) will apply cleanly (with no conflicts). However, I want to be sure everything is okay before I pull it into the master branch, so I'll create a new branch for that, just under the toolbar.



If you want to look at the changes applying the patch will make, you can click on its hash or commit message. I'm going to select it via the checkbox at the front of the entry (you can integrate as many patches as you want at a time) and choose *Apply* from the *Actions* menu.



GitHub will integrate the patches and let you know which ones applied successfully and which ones did not. You then have the option to update your integration branch, or abort the process.

I'll updated the branch. Now, if I look at my develop branch, I'll see that the changes have been made:

Summary

That's our tour of GitHub… and that's our book on Git! By now, you should be able to successfully use Git to organize your projects and use GitHub to collaborate with other developers. If you'd like to take your knowledge of Git to the next level, check out Appendix A for some excellent follow-up resources.

APPENDICES

Appendix A: More Git Resources

While I hope you've learned a lot from this book, it would be impossible for me to teach you everything there is to know about Git. But don't worry — there are plenty of resources that you can check out next. Here are some that I have found to be helpful:

- **YUI Theater: Git, GitHub, and Social Coding:** this is a talk Tom Preston-Werner, Chris Wanstrath, and Scott Chacon gave at Yahoo! last year. Acually, this is where I first learned about Git. http://developer.yahoo.com/yui/theater/video.php?v=prestonwerner-github
- **GitCasts.com:** a great series of short screencasts on Git by Scott Chacon. Also on this site is his RailsConf 2008 talk (http://www.gitcasts.com/posts/railsconf-git-talk), which is a great way to start learning what Git is doing behind the scenes. http://gitcasts.com/
- **Git Community Book:** available as HTML or PDF. http://book.git-scm.com/
- **Pro Git:** An Apress book written by Scott Chacon (~$35). You can read the book online (http://progit.org/book/) as well. http://progit.org/
- **Git Magic:** An ebook by Ben Lynn, available in HTML and PDF. http://www-cs-students.stanford.edu/~blynn/gitmagic/
- **Git Peepcode Screencast:** An hour-long screencast that will teach you how to use Git ($12). http://peepcode.com/products/git

- **Git Internals PDF for Peepcode:** This PDF by Scott Chacon goes into detail about the inner workings of Git ($12). https://peepcode.com/products/git-internals-pdf
- **Git Cheat Sheet:** Created by Zack Rusin. Available in SVG and PNG (check the comments for a PDF version). http://zrusin.blogspot.com/2007/09/git-cheat-sheet.html

Appendix B: Git GUIs

While I prefer to use the command line tools for Git, maybe you want something with a user interface. If that's so, check out the brief list of Git GUIs below.

Mac

- **GitBox:** http://gitbox.pierlis.com/
- **GitX:** http://gitx.frim.nl/

Windows

- **TortoiseGit:** http://code.google.com/p/tortoisegit/
- **Git Extensions:** http://sourceforge.net/projects/gitextensions/

Linux

- **Giggle:** http://live.gnome.org/giggle
- **Gitg:** http://trac.novowork.com/gitg/

About The Author

Andrew Burgess is a Canadian web developer and is the Associate Editor at Nettuts+, where he has published numerous popular tutorials and screencasts. He's also a web development reviewer on Envato's Tuts+ Marketplace, and as a web developer, he specializes in JavaScript and Ruby. Andrew lives with his family in Oshawa, Canada.

Check out Andrew's personal site at: http://andrewburgess.ca/

Or follow him on Twitter: @Andrew8088

Your Screencasts

Use the links below to download your series of screencasts.

http://distro.rockablepress.com/extras/git1-164sl8v

http://distro.rockablepress.com/extras/git2-164sl8v

http://distro.rockablepress.com/extras/git3-164sl8v

http://distro.rockablepress.com/extras/git4-164sl8v

http://distro.rockablepress.com/extras/git5-164sl8v

In Getting Good with Git, Nettuts+ Associate Editor Andrew Burgess will guide you through the sometimes-scary waters of source code management with Git, the fast version control system.

Git's speed, efficiency, and ease-of-use have made it the popular choice in the world of source code managers. And with a service like GitHub available for sharing your code, there's no question about whether learning Git is worth your time!

In this book, Andrew Burgess will take you from knowing nothing about source code management to being able to use Git proficiently. You'll look at why you should use a version control system, why Git is better than the other options, and how to set up and use Git. This book covers some of the advanced features of Git, and includes an appendix of other resources that will take your Git knowledge to the next level. We'll even get to know GitHub!

This book also includes a screencast series that walks you though most of the concepts that the book teaches. With over two and a half hours of video, you'll get a solid grounding in Git without much effort.

ROCKABLE*

Get Good