Goal-Oriented Git

by George Brocklehurst

thoughtbot

Goal-Oriented Git

George Brocklehurst

February 20, 2015

Contents

About this book

If you want to learn how to use Git without worrying too much about what's going on under the hood, then this book is for you.

Every chapter will take something that Git can do for you—from tracking changes, to searching your files, to collaborating with others—and explain practically how to achieve that goal. We won't cover everything Git can do, but we won't just stick to the basics either: the goal of this book is to teach you a working set of Git commands that can get you through most day-to-day situations.

If you're learning Git for the first time, I'd recommend reading through in order: When we look at a particular goal, I'll assume knowledge of commands that were covered in earlier chapters.

Part I

Getting started

Goal: Get set up

What is Git?

Git is a version control system. When you work on a project—whether it's a piece of software, a Web site, a book, or anything else that goes through many revisions—using version control lets you track the changes that you make, return to previous versions, try alternative ideas, and seamlessly merge your own work with the work of others.

Git can track any kind of files, but it's best suited to plain text files. If you're not familiar with the term, these are files that just contain unformatted text as opposed to images or things like PDF.

Using the command line

Everything covered in this book will be done from the command line. While there are various applications that provide graphical interfaces for Git, the command line interface is powerful, flexible, and works on every platform. Issuing commands, one-by-one, forces us to take things step-by-step so we can really understand what's happening.

If you're not familiar with using command line tools, then there are a few terms you'll need to know. Here's a typical Git command:

```
git commit --message "My first commit"
```

We'll cover what it does later, but for now let's name its component parts:

- `git` is the command: it's the name of the program we want to run.
- `commit` is the sub-command: it tells Git what we want it to do. Not all command line interfaces use sub-commands, but since Git uses them for almost everything, in the context of this book we can just think of `git commit` as the command.
- `--message` and `"My first commit"` are both arguments: they are passed to the command to further nuance its behaviour. Arguments often come in pairs, where we tell Git what option we want to use—in this case `--message`—and then a value for that option—`"My first commit"`.

Installing and configuring Git

Before we begin, you'll need to install Git and get it set up:

1. Go to the Git downloads page, and follow the instructions for your operating system: http://git-scm.com/downloads

2. Make sure you can run Git commands from the command line. If you're on Mac OS X you can use the Terminal.app application; if you're on Windows there's an application called Git Bash that comes with Git.

   The following command should tell you what version of Git you have installed:

   ```
   $ git --version
   git version 2.2.1
   ```

   As with all of the examples in this book, the `$` at the beginning of the line indicates that this is a command; you shouldn't include it in the command you type.

   If you see a version number, as show above, then Git is installed successfully. If you see a message along the lines of `command not found`, then Git isn't correctly installed yet.

3. When Git is tracking changes to our files, it needs to know who made those changes—we'll see why this is important when we explore Git's collaborative features in Part 4.

   To identify yourself, and therefore the changes you track with Git, you need to set Git's `user.name` and `user.email` settings with the following commands, replacing my name and email address with your own:

   ```
   $ git config --global user.name "George Brocklehurst"
   $ git config --global user.email george@georgebrock.com
   ```

   The `--global` argument tells Git that these settings should apply to all of the Git projects you work on, so you'll only need to set this once. Don't worry if you don't completely follow this now; we'll talk about the `git config` command in more detail later in the book.

Goal: Track changes

If you can see how the incremental changes to your files led to their current state, then you have a better chance of understanding the project you are working on, fixing problems that arise, and collaborating successfully with others. In order to unlock the riches promised by a comprehensive history, we have to first build that history, one change at a time.

By now, you should have installed Git on your system, and so you are ready to create your first Git repository.

A repository is the place where Git keeps track of all of the changes for a given project. We need to create a repository for each project we want to track with Git.

The `git init` command

The `git init` command creates a new Git repository to keep track of changes to the files in the directory where the command is run. For example:

```
$ cd projects/git-book
$ git init
Initialized empty Git repository in /home/george/projects/git-book/.git/
```

This creates a new Git repository in the `projects/git-book` directory, telling Git that from now on we want to track the changes made to one or more of the files in that directory. Since we haven't told Git exactly which files to track yet,

the repository is empty; this is true even if the `projects/git-book` directory isn't empty.

Running the command creates a new directory, `projects/git-book/.git`, where Git will store the files that represent the repository. This is the Git directory, whereas `projects/git-book` is the working directory.

To begin with, we'll modify files in the working directory, just as we would if we weren't using Git, and then tell Git to track the changes we've made in the repository. As we get more advanced, we'll be able to conjure old or alternative versions of our project from the repository, summoning them to the working directory to do our bidding.

The `git add` and `git commit` commands

The individual changes that make up the history of our project are called commits. We commit changes to the Git repository in the same way we might commit something to memory; once it's committed we won't lose it.

Each commit consists of a set of changes—like adding a new file, changing the contents of a file, deleting a file, or changing the properties of the file in some way—and a commit message that describes the changes, and the name and email address of the commit's author.

Git provides a staging area called the index, where we build up a set of changes before committing them to the repository. We'll come back to why the index is useful later, but for now it's enough to know that creating a commit is a three stage process:

1. Change files in the working directory.
2. Add some or all of those changes to the index.
3. Create a commit from the changes staged in the index.

Let's say we've created a file in our `git-book` working directory called `chapter1.txt`, and we're happy with the contents of the file and want to commit it to the repository. That's step one, step two is to add the file to the index using the `git add` command:

```
$ git add chapter1.txt
```

And then we can create a commit using the `git commit` command:

```
$ git commit --message "Add first draft of chapter one"
[master (root-commit) 6bdb671] Add first draft of chapter one
 1 file changed, 1 insertion(+)
 create mode 100644 chapter1.txt
```

The commit message, passed using the `--message` argument, describes the changes in the commit: in this case, it adds the first draft of chapter one.

By committing, we've drawn a line in the sand. Whatever changes we make to chapter one in future, we'll always be able to get back to this revision of the file. Even though a commit only contains the changes made to a file since it was last committed, we can think of a commit as representing a particular revision of our project, since Git is able to use all of the commits up to that point to reconstruct the revision of the project that existed when that commit was made.

Now that we've included it in a commit, the file `chapter1.txt` will be tracked by Git: if we make any changes to the file in future, Git will notice that it has changed and needs to be committed again.

Summary

- Create a new Git repository with `git init`
- Stage related changes in the index with `git add`
- Commit the changes in the index with `git commit`

Goal: Understand what is being tracked

So far, we've made simple commits with just one file, but in a complex project there could be dozens or even hundreds of files. How do we know if they've been changed since they were last committed, or if they're even tracked by Git?

In the last chapter we defined three important locations:

1. The working directory, where we keep and work on the current version of our project's files.
2. The index, where we build up a set of changes ready to commit.
3. The repository, where Git stores the history of our project as a series of commits.

Git provides a number of commands for comparing the state of the repository, the index, and the working directory, so we can stay up to date with what's changed.

The `git status` command

The `git status` command will tell us the status of our files. It lists all of the changed or untracked files in the working directory, split into the following groups:

1. **Changes to be committed**: files which have already been added to the index with `git add` and will be included the next time we run `git commit`.
2. **Changes not staged for commit**: files which have been changed in the working directory, but not added to the index.
3. **Untracked files**: files which have never been committed and therefore aren't tracked by Git.

For example:

```
$ git status
HEAD detached from eec7d91
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   chapter2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapter1.txt

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

    chapter3.txt
```

In this case, `chapter2.txt` has just been added to the index for the first time with `git add`, `chapter1.txt` is tracked by Git but has been modified since it was last committed, and `chapter3.txt` isn't tracked by Git yet.

`git status` won't mention any files that haven't been changed since they were last committed. If none of your files have changed, `git status` will let you know that you have clean working directory:

```
$ git status
HEAD detached from eec7d91
nothing to commit, working directory clean
```

`git status` isn't concerned with the details of exactly what's changed in each file, just which files have been changed and therefore could be committed.

The `git diff` command

Now we know which files in our working directory have changed, it would be good to see the details of those changes. The `git diff` command provides this.

If you've used the Unix `diff` utility then the output of `git diff` might look familiar, but while `diff` shows the difference between files, `git diff` shows the difference between versions of the same file.

Without any arguments, `git diff` shows all changes that haven't been committed or added to the index.

For example, here's what `chapter1.txt` the list time it was committed:

```
CHAPTER ONE

This is the first chapter, where it all begins.
```

Since that commit, we've modified `chapter1.txt` in the working directory to look like this:

```
CHAPTER 1

This is the first chapter, where it all begins.
```

Running `git diff` shows us exactly what the differences are:

```
$ git diff
diff --git a/chapter1.txt b/chapter1.txt
index 1e41245..80a7940 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
```

```
-CHAPTER ONE
+CHAPTER 1

 This is the first chapter, where it all begins.
```

The first section gives some information about the change Git is showing: which versions are being compared, and on which lines of which files there are changes. Other than the file's name, this is more useful to machines than humans, and I usually don't read it carefully. More important are the changes themselves:

```
-CHAPTER ONE
+CHAPTER 1

 This is the first chapter, where it all begins.
```

Lines beginning with a `-` indicate a line that has been removed, and lines beginning with a `+` indicate a line that has been added. In this case a line has been changed: the old version of the line was removed, and the new version was added.

Lines beginning with a space—lines that lack both `+` and `-`—are unchanged, but are shown to give the context of the change.

The `git diff --staged` command

Remember that without any arguments `git diff` shows us the changes that haven't been staged in the index yet. `git diff --staged` shows us the changes that *have* been staged, but haven't yet been committed. In other words, `git diff --staged` shows you the changes that would be included in the commit, if you were to run `git commit` right now.

It can be useful to run `git diff --staged` before `git commit` as a final check that the right changes have been staged in the index.

Summary

- Look at the state of the working directory with `git status`
- Look at the changes since the latest commit with `git diff`
- Look at the contents of the index with `git diff --staged`

Goal: Make beautiful commits

A good commit should contain a set of related changes, with a description of why those changes were made. Much of the power of tracking history is lost if we bundle lots of unrelated changes together in the same commit, or don't take the time to describe them properly.

So far we've been assuming that all of the changes to a file belong in the same commit. Real world work is rarely this neat and tidy, though. Perhaps we were half way through changing a document, and found ourselves fixing an un-related typing error; perhaps we forgot to commit a set of changes before moving on to the next idea; perhaps we just got distracted: whatever the cause, a real project's working directory can be a messy place.

This is where the index really comes into its own: By updating the index before we commit, we can carefully select the changes that go into each commit; even going so far as to split unrelated changes to the same file over several commits.

The `git add --patch` command

In addition to the name of a file, the `git add` command can take some options to control exactly what is added to the index. Particularly useful is the `--patch` option, which shows you each change in turn and lets you decide if it should be added to the index or not.

Let's look at an example: I've made two changes to the same file but they are unrelated, so I want to add them to the repository as two separate commits. `git diff` tells me I have the following changes:

```
$ git diff
diff --git a/chapter1.txt b/chapter1.txt
index 8f6f18e..d119536 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
-The quick brown fxo jumped over the lazy dog.
+The quick brown fox jumped over the lazy dog.

-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
```

On the first line, I've corrected a typing error by changing `fxo` to `fox`, and then on the third line I've added some additional material about the dog's reaction to the fox. `git add --patch` will allow us to go through each change in turn, and decide if we want to add it to the index.

```
$ git add --patch
diff --git a/chapter1.txt b/chapter1.txt
index 8f6f18e..d119536 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
-The quick brown fxo jumped over the lazy dog.
+The quick brown fox jumped over the lazy dog.

-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

Git will show the changes—using the familiar style of `git diff`—one at a time, and asks me what I want to do with each hunk with the question `Stage this hunk [y,n,q,a,d,/,s,e,?]?`. Most of the time we can answer with `y` to add the change to the index, or `n` to ignore it for now and move on to the next change. In this case, it's a little more complicated: since my changes are very close together, Git's assumed that they're probably related and shows them as a single hunk. I want to split up this hunk, and since I need something beyond a basic "yes" or "no" I can use `?` to explain the other options:

```
Stage this hunk [y,n,q,a,d,/,s,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later 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
```

After looking through my options, I see I can use `s` to tell Git that even though these changes are close to each other they should be split up so that I can treat them differently:

```
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Split into 2 hunks.
@@ -1,2 +1,2 @@
-The quick brown fxo jumped over the lazy dog.
+The quick brown fox jumped over the lazy dog.

Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]?
```

Git splits the change into two hunks, and shows me the first of them. This change is just the correction on the first line, so I can stage it in the index using `y`. Once it's staged, Git will show me the next change. Since this change isn't related to the one I've already added to the index, I don't want to include it in the same commit, so I can skip it with `n`:

```
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? y
@@ -2,2 +2,2 @@
```

```
-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
```

I've now made a decision about each of the changes to the tracked files in my working directory, so `git add --patch` exits, and I can commit the correction that's staged, and then stage and commit the additional content:

```
$ git commit --message "Fix typing error"
[master e4188ff] Fix typing error
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git diff
diff --git a/chapter1.txt b/chapter1.txt
index 9d07dc3..d119536 100644
--- a/chapter1.txt
+++ b/chapter1.txt
@@ -1,3 +1,3 @@
 The quick brown fox jumped over the lazy dog.

-The dog was not best pleased.
+The dog was not best pleased, and barked angrily.
$ git add chapter1.txt
$ git commit --message "Add information about the dog's reaction"
[master 6e2840a] Add information about the dog's reaction
 1 file changed, 1 insertion(+), 1 deletion(-)
```

I almost always use the `--patch` option when I'm adding files to the index, even when I'm confident that all of the changes are related and will end up as a single commit: It gives me the opportunity to review my changes, and has saved me many times from committing something that wasn't quite right.

Summary

- Good commits are focused on a set of related changes
- Use `git commit --patch` to add specific changes to the index