A LIST APART

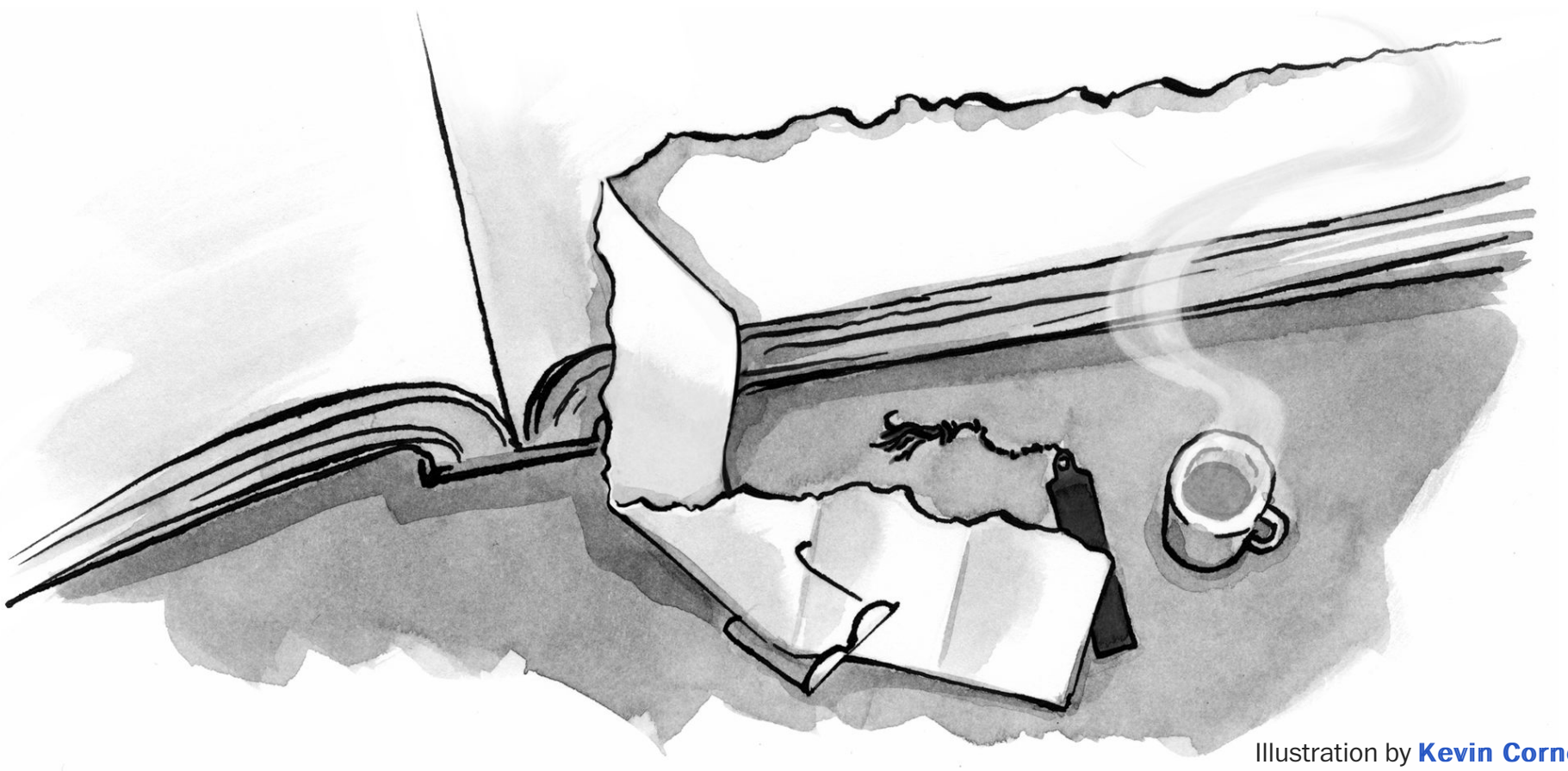

Illustration by **Kevin Cornell**

The Art of the Commit

by **David Demaree** · February 02, 2016

Published in Code, Workflow & Tools

A note from the editors: We're pleased to share an excerpt from Chapter 5 of David Demaree's new book, *Git for Humans, available now from A Book Apart (http://abookapart.com/products/git-for-humans).*

Git and tools like GitHub offer many ways to view what has changed in a commit. But a well-crafted commit message can save you from having to use those tools by neatly (and succinctly) summarizing what has changed.

The log message is arguably the most important part of a commit, because it's the only place that captures not only what was changed, but why.

What goes into a good message? First, it needs to be short, and not just because brevity is the soul of wit. Most of the time, you'll be viewing commit messages in the context of Git's commit log, where there's often not a lot of space to display text.

Think of the commit log as a newsfeed for your project, in which the log message is the headline for each commit. Have you ever skimmed the headlines in a newspaper (or, for a more current example, BuzzFeed) and come away thinking you'd gotten a summary of what was happening in the world? A good headline doesn't have to tell the whole story, but it should tell you enough to know what the story is about before you read it.

If you're working by yourself, or closely with one or two collaborators, the log may seem interesting just for historical purposes, because you would have been there for most of the commits. But in Git repositories with a lot of collaborators, the commit log can be more valuable as a way of knowing what happened when you weren't looking.

Commit messages can, strictly speaking, span multiple lines, and can be as long or as detailed as you want. Git doesn't place any hard limit on what goes into a commit message, and in fact, if a given commit does call for additional context, you can add additional paragraphs to a message, like so:

```
Updated Ruby on Rails version because security

Bumped Rails version to 3.2.11 to fix JSON security bug.
See also http://weblog.rubyonrails.org/2013/1/8/Rails-3-2-11-3-1-10-
3-0-19-and-2-3-15-have-been-released/
```

Note that although this message contains a lot more context than just one line, the first line is important because only the first line will be shown in the log:

```
commit f0c8f185e677026f0832a9c13ab72322773ad9cf
Author: David Demaree
Date:   Sat Jan 3 15:49:03 2013 -0500

Updated Ruby on Rails version because security
```

Like a good headline, the first line here summarizes the reason for the commit; the rest of the message goes into more detail.

Writing commit messages in your favorite text editor

Although the examples in this book all have you type your message inline, using the `--message` or `-m` argument to `git commit`, you may be more comfortable writing in your preferred text editor. Git integrates nicely with many popular editors, both on the command line (e.g., Vim, Emacs) or more modern, graphical apps like Atom, Sublime Text, or TextMate. With an editor configured, you can omit the `--message` flag and Git will hand off a draft commit message to that other program for authoring. When you're done, you can usually just close the window and Git will automatically pick up the message you entered.

To take advantage of this sweet integration, first you'll need to configure Git to use your editor (specifically, your editor's command-line program, if it has one). Here, I'm telling Git to hand off commit messages to Atom:

```
$: git config --global core.editor "atom --wait"
```

Every text editor has a slightly different set of arguments or options to pass in to integrate nicely with Git. (As you can see here, we had to pass the `--wait` option to Atom to get it to work.) GitHub's help documentation has a good, brief guide to setting up several popular editors *(http://bkaprt.com/gfh/05-05/)*.

Elements of commit message style

There are few hard rules for crafting effective commit messages—just lots of guidelines and good practices, which, if you were to try to follow all of them all of the time, would quickly tie your mind in knots.

To ease the way, here are a few guidelines I'd recommend always following.

BE USEFUL

The purpose of a commit message is to summarize a change. But the purpose of summarizing a change is to help you and your team understand what is going on in your project. The information you put into a message, therefore, should be valuable and useful to the people who will read it.

As fun as it is to use the commit message space for cursing—at a bug, or Git, or your own clumsiness—avoid editorializing. Avoid the temptation to write a commit message like "Aaaaahhh stupid bugs." Instead, take a deep breath, grab a coffee or some herbal tea or do

whatever you need to do to clear your head. Then write a message that describes what changed in the commit, as clearly and succinctly as you can.

In addition to a short, clear description, when a commit is relevant to some piece of information in another system—for instance, if it fixes a bug logged in your bug tracker—it's also common to include the issue or bug number, like so:

```
Replace jQuery onReady listener with plain JS; fixes #1357
```

Some bug trackers (including the one built into every GitHub project) can even be hooked into Git so that commit messages like this one will automatically mark the bug numbered `1357` as done as soon as the commit with this message is merged into `master`.

BE DETAILED (ENOUGH)

As a recovering software engineer, I understand the temptation to fill the commit message—and emails, and status reports, and stand-up meetings—with nerdy details. I *love* nerdy details. However, while some details are important for understanding a change, there's almost always a more general reason for a change that can be explained more succinctly. Besides, there's often not enough room to list every single detail about a change and still yield a commit log that's easy to scan in a Terminal window. Finding simpler ways to describe something doesn't just make the changes you've made more comprehensible to your teammates; it's also a great way to save space.

A good rule of thumb is to keep the "subject" portion of your commit messages to one line, or about 70 characters. If there are important details worth including in the message, but that don't need to be in the subject line, remember you can still include them as a separate paragraph.

BE CONSISTENT

However you and your colleagues decide to write commit messages, your commit log will be more valuable if you all try to follow a similar set of rules. Commit messages are too short to require an elaborate style guide, but having a conversation to establish some conventions, or making a short wiki page with some examples of particularly good (or bad) commit messages, will help things run more smoothly.

USE THE ACTIVE VOICE

The commit log isn't a list of static things; it's a list of changes. It's a list of actions you (or someone) have taken that have resulted in versions of your work. Although it may be tempting to use a commit message to label a version of the work—"Version 1.0," "Jan 24th deliverable"—

there are other, better ways of doing that. Besides, it's all too easy to end up in an embarrassing situation like this:

```
# Making the last homepage update before releasing the new site
$: git commit -m "Version 1.0"
```

```
# Ten minutes later, after discovering a typo in your CSS
$: git commit -m "Version 1.0 (really)"
```

```
# Forty minutes later, after discovering another typo
$: git commit -m "Version 1.0 (oh FFS)"
```

Describing changes is not only the most correct format for a commit message, but it's also one of the easiest rules to stick to. Rather than concern yourself with abstract questions like whether a given commit is the release version of a thing, you can focus on a much simpler story: *I just did a thing, and this is the thing I just did*.

Those "Version 1.0" commits, therefore, could be described much more simply and accurately:

```
$: git commit -m "Update homepage for launch"
$: git commit -m "Fix typo in screen.scss"
$: git commit -m "Fix misspelled name on about page"
```

I also recommend picking a tense and sticking with it, for consistency's sake. I tend to use the imperative present tense to describe commits: *Fix misspelled name on About page rather than fixed or fixing*. There's nothing wrong with *fixed* or *fixing*, except that they're slightly longer. If another style works better for you or your team, go for it—just try to go for it consistently.

What happens if your commit message style isn't consistent? Your Git repo will collapse into itself and all of your work will be ruined. *Kidding!* People are fallible, lapses will happen, and a little bit of nonsense in your logs is inevitable. Note, though, that following style rules like these

gets easier the more practice you get. Aim to write the best commit messages you can, and your logs will be better and more valuable for it.

About the Author

David Demaree

David Demaree has been designing and building web sites since the 1990s. He currently serves as senior product manager for Typekit at Adobe and is the author of *Git for Humans*, published by A Book Apart. You can read his thoughts about software, food, coffee, and more on Twitter and Medium.

MORE FROM THIS AUTHOR

Getting Started with Sass *(getting-started-with-sass)*

ISSN 1534-0295 ·