Rake Routes

Rails, Ruby, and other programming.

- RSS
- Email

Search

Navigate...

- Blog
- Archives

Deliberate Git

Aug 19th, 2013

Hello Internet! Here's my talk "Deliberate Git" in blog post form.

There's also video of my presentation of Deliberate Git at Steel City Ruby 2013.

If you'd like to just read the slides they're up on Speaker Deck: Deliberate Git - Slides. Although I highly recommend just pointing people to this blog post if they want to read the talk instead of watching it online.

Special thanks to PhishMe for sending me to Steel City Ruby to give this talk!

Git

Many teams see Git as a source of frustration. A painful reminder that the rubber needs to meet the road in order to make ends meet and keep our customers happy. They see Git as just a mechanism to transport the code from development machines to production servers and keep everyone in sync.

But I want Git to do more. Being distributed means Git gives us the opportunity to do something really amazing. It allows us to make quick commits locally without breaking flow and then allows us to rewrite those commits into a cohesive story that we share with our team.

If you focus on putting more information into your repo now, you can see amazing returns when you have questions later. Let's start by looking at the most fundamental piece of a Git repository.

Git Commits

Who doesn't love git commits? Commits are amazing! I like to say that git commits are like emails to future developers, but that's not entirely accurate. They are way more awesome!

Commits last forever

Unless you remove them, git commits last forever. If you go spelunking in long running codebases like Rails you'll find commit messages that are almost ten years old.

Commits are extremely searchable

Commit messages can be easily searched in many ways. You can find commits by their message body, by their code change, by the files they touched, by author, by date, whatever.

Commits are available to everyone

Every commit that anyone on your team has ever made to the repo is instantly available to anyone that clones the repository. So make them good. The better you make your commits now, the easier your future changes can be.

When Commits Are Bad

Commits can be the answer to your question just handed to you by the previous programmer. They can also be the source of some surprisingly powerful inclinations to introduce your head to the desk.

Years ago I was working on an application. We'd been working days and nights to get this new application ready to deploy before a press release deadline. We had constantly changing business directions, constantly tweaked page layouts, the occasional complete site redesign, and many APIs that we were supposed to be talking to or providing.

In short, it was a mess.

Taking a specific example: all along we'd been using an external API to track new sales leads from an interest form on the website. Fine.

I was wrapping up a feature that used that API. Great!

Unfortunately when I was merging in my changes I found that in master the lead tracking API had been completely replaced with a new API for a new service.

Nooooo! :-(

It was late at night and I was the only one there so there no one else on the team I could talk to immediately.

I checked my emails about the project but I couldn't find anything mentioning such a big change. I checked my notes from the last several business meetings (all within the last two days) and still had nothing.

But aha! We're a brand new project using Git! I bet there's a good commit message around that code change. Of course! To the git log!

```
1 Forgot to update form action for new lead engine
```

Well ok, that's not too helpful, but we're on the right track. Maybe we just need to go further.

```
1 Switched to new lead engine.
```

Yes…yes I know that. But why? Why? And now it's time for my head to meet the desk.

Not only is that commit message not helpful, it would've been so easy for it to be helpful!

Consider this alternative:

```
1 Replace lead tracking Foo with Bar
2
3 Foo's API has dropped 5 out of the last 27 emails since we started keeping our
4 own logs. Their support is not responding.
5
6 Fred and I decided to drop in Lead Engine B after checking alternatives (see
7 wiki:LeadEngineResearch).
```

How much more useful is that? WAY more useful right?

Let's break it down.

The Subject

```
1 Replace lead tracking Foo with Bar
```

The subject line is clear, imperative, and short.

If I were looking at a list of commits that only showed the subjects then I'd still have a good idea of what this commit does.

Many Git tools and commands only show the subject of commits. So making them short and direct makes them much easier to follow.

The Body

```
1 Foo's API has dropped 5 out of the last 27 emails since we started keeping our
2 own logs. Their support is not responding.
3
4 Fred and I decided to drop in Lead Engine B after checking alternatives (see
5 wiki:LeadEngineResearch).
```

That subject line is supported by a detailed body that explains it even further.

The subject tells us what the commits does, the body tells us why.

That's what a commit should do. State the change then explain the change.

Good Commit Subjects

A good commit subject is written in the present tense and as a command.

Present Tense

Why the present tense? Because they describes what the commit **does** not what it **did**. Git commits should be written in the present tense because they happen multiple times.

They happen when you write them. They happen when you merge them into another branch. They happen when someone else pulls down those new commits to their repo. They happen when someone cherry-picks them onto their branch.

Rewriting commits that have subjects written in the past tense is nonsensical. It introduces a cognitive dissonance that makes editing your commit history feel weird. It might seem silly, but that feeling of a disconnect will subtly push you towards never editing your commits.

Command

Why should commit subjects be written as a command? Because a commit is a TODO item for the repository.

The commit says: “Hey Git! Here’s a set of changes to make.”

Git says: “Great! I’ll do that and let you know if I can’t figure something out.”

Commits aren’t a historical log. They are a description of what will happen if your commit is added to the repository. They are descriptions of what the *repo* should do.

Good Commit Bodies

Can Be Informal

The commit body does not have to be written as a present tense command using terse language. In fact, it’s usually better written informally.

Describe the “Why?”

The commit body describes the TODO item, it’s not the TODO item itself. Why are we flipping the bit or whatever we’re doing?

Are Messages to Your Team

The commit body is where you drop down and actually talk to your team. Explain things as if you were writing a message summarizing your change. Because that’s what you’re doing!

Well Written Commits

I consider a commit well written when it explains two things:

What the Code Change Does

The commit should explain what the code change does, **not** necessarily what the code does. The commit is the point at which we're pivoting. It's the only chance we have to easily capture the code before, the code after, and glue them together with an explanation.

Why the Change is Necessary

The commit should explain why the change is necessary. What are we doing differently that requires this commit to get us there? Are laying the foundation for a feature? Are we fixing a bug? Are we refactoring? These are critical things to know about the code change that we simply cannot get by reviewing the code itself.

But don't feel limited to just those two pieces of information. Push as much of your knowledge about the code as possible into the commit message. Some examples of more information are:

Alternatives Considered

How did you arrive at this solution? Maybe setting the number of connections to 500 addresses a traffic spike, or maybe that's the optimal setting you figured out after painstakingly researching and benchmarking the app. Who knows? **You do**! You're the one changing the code! So write it down!

Potential Consequences

What are the consequences? Talk about the possible effects the code change will create. Is there something we should keep an eye on? Something to watch out for? If so, write it down.

Are Git Commits like Comments?

At this point, you might be thinking: "But Stephen. That sounds an awful lot like documenting code with comments. Isn't an excessive amount of commenting a bad idea?"

Yes, lots of comments (some would say any comments) is a bad idea because it's so very easy for the comments to get out of sync with the code.

```
1 # set x to 5
2 y = 3
```

Maybe setting y to 3 *does* set x to 5. Or maybe that comment is just out of sync. We can't know! So yes, treat comments with care.

But, this kind of disparity will never happen with a git commit!

Git Commits are Always in Sync with the Code

Commits are tied to the code change that they're talking about. When the code changes, a new commit message will come in and take its place automatically.

This is great because it allows you to write documentation about the change without having to worry about the documentation getting stale. It will last exactly as long as the code it's talking about lasts.

Commit messages also don't clutter up the code, yet they're just a step away if you need them.

What About Coding Flow?

Now you might say: "But, Stephen. Well written commits sound great, but they also sound like an awful lot of work. Do you really expect me to jump back and forth between the wonderful flow of coding and the crushing annoyance of committing?"

NO! That would be awful!

WIP commits while coding

If you are working on local commits, then your commits are your business. Until they enter the public (production) record, do whatever you want! "WIP" with a shorthand reference is my goto commit when I don't want to break my flow but I know that I want to record a change.

Commit Frequently

It's important to make lots of small commits, even while you are in the flow of coding. Combining commits is very easy, making atomic commits from a big code change is harder, and breaking apart large commits into atomic commits is very hard. So err on the side of smaller commits that you can combine later.

Rewrite Your Commits

When you're at a stopping point in coding. Switch gears to writing mode and rewrite, remove, combine, and/or reorder your commits. Think of it like red-green-refactor; but code, commit, rewrite. Write the code, commit the code, then refactor the commits.

How do you rewrite commits? Git has a simple command for doing exactly that!

Git Rebase

No no, don't run! It'll be ok! (For some of you red flags are already going up.)

For those of you that don't know: rebase is a git command that allows you to modify commits that you've already made. This is very powerful but can also be a huge headache if you modify commits that are already part of someone else's history.

But don't worry! We'll be using git rebase precisely, deliberately, and safely. We'll also use it in a way that will work with any team workflow.

The magic incantation is: `git rebase -i`

The `-i` stands for interactive. And `git rebase -i` is one of git's killer features.

It's what gives you the power to hack code out quickly, and still give your team great commit messages. If you want to follow the path of good commit messages and still want to knock out code in a good flow, then this command is absolutely imperative to learn.

`git rebase -i` allows you to:

- rewrite commits
- remove commits
- combine commits
- reorder commits

(It lets you do more, but that's all we're going to worry about now.)

You write your shorthand commits, then use git rebase -i against your own local commits and rewrite them *before* pushing them anywhere. It'll be as if you only ever wrote amazing commits.

With the power of git rebase -i at our disposal we can make three dozen commits implementing a feature with most having a hastily scratched out message. Then trim the weeds with magic and end up with just 5 well written commits to share with our team.

```
$ git rev-list master..feature | wc -l
37

$ git rebase -i 1123e81
[magic happens] (/◕‿◕)/*:･ﾟ✧

$ git rev-list master..feature | wc -l
5
```

Using git rebase -i

When you're ready to start rewriting you:

- Use git log to find the commit before the first the commit you want to change
- Run `git rebase -i [that commit's hash]`

As an example, assume that 3a049ba is the stable commit before our three WIP commits and that we want to combine those WIP commits into one cohesive message.

```
# Find the stable commit hash
$ git l

* 193bn13 WIP switch fizz to bazz
* 6a0b73a WIP support fizz
* aa02d1e WIP Start to hack out foo
* 3a049ba Allow creating users via JSON

# Run git rebase -i with that commit's hash
$ git rebase -i 3a049ba
```

Git will open a screen like this in your editor

```
1  pick aa02d1e WIP Start to hack out foo
2  pick 6a0b73a WIP support fizz
3  pick 193bn13 WIP switch fizz to bazz
4
5  # Rebase 31049ba..193bn13 onto 31049ba
6  #
7  # Commands:
8  #  p, pick = use commit
9  #  r, reword = use commit, but edit the commit message
10 #  e, edit = use commit, but stop for amending
11 #  s, squash = use commit, but meld into previous commit
12 #  f, fixup = like "squash", but discard this commit's log message
13 #  x, exec = run command (the rest of the line) using shell
14 #
15 # These lines can be re-ordered; they are executed from top to bottom.
```

That seems complicated, but we only care about these parts:

```
1 pick aa02d1e WIP Start to hack out foo
2 pick 6a0b73a WIP support fizz
3 pick 193bn13 WIP switch fizz to bazz
4
5 #  p, pick = use commit
6 #  r, reword = use commit, but edit the commit message
7 #  f, fixup = like "squash", but discard this commit's log message
8 #
9 # These lines can be re-ordered; they are executed from top to bottom.
```

To rewrite the WIP commits into one message we tell Git:

- reword the first commit
- only keep the changes (and not the messages) of the last two

Like so

```
1 reword aa02d1e WIP Start to hack out foo
2 fixup 6a0b73a WIP support fizz
3 fixup 193bn13 WIP switch fizz to bazz
```

After saving and closing that file, git will then open up another editor with the first commit's message in it. After we rewrite it then git will create the new commit message that combines the three WIP commits.

```
1  Add foo feature with bazz support
2
3  The ability to foo has been requested by various
4  users. This implementation will allow foo with
5  support for the bazz protocol.
```

```
6
7  At first we thought that the fizz protocol would
8  be a good fit, but that wasn't the case when we
9  tested it against over 10,000 interactions. We
10 opened up an issue with the fizz API.
11
12 - The bazz protocol: http://example.com/bazz
13 - fizz api failure issue: http://example.com/issue/12
```

Digging into History

"Ok Stephen, now I can write awesome commits. But what good is that if I can't find them again?"

It's no good! It's no good at all. Luckily, there are a kajillion tools out there to sift through your git history. The GUI tools are easiest to get going with, but I'm going to show you a selection of command line options that you might find useful.

```
1  # list commits newest to oldest
2  git log
3
4  # list commits with their diff
5  git log -p
6
7  # only show commits whose *messages* contain the string
8  git log --grep="commit contents"
9
10 # only show commits whose *code change* contains the string
11 git log -S"diff contents"
12
13 # only show commits for the given path or file
14 git log -- path/or/file
15
16 # show all the lines of a file and their current commit
17 git blame FILENAME
```

We can use these commands to find some pretty interesting things in existing codebases on Github. Say the Rails codebase.

```
1  # Run these against rails/rails
2
3  # Find the oldest commit
4  rails/rails git log --format="%H" | tail -1 | xargs -L 1 git show
5
6  # Find commits with a magic word in them
7  rails/rails git log -S"xyzzy" -p
8
9  # Find commits that made people happy
10 rails/rails git log --grep=":heart:"
11
12 # Find commits that made people sad
13 rails/rails git log --grep="wtf" -p
14
15 # And find interesting people
```

```
16 rails/rails git log --grep="metz"
```

Let's Wrap This Up

Your code can only say what it does right now. You can't look at a method and see its history: the alternative approaches that have been considered, the algorithms that have already been outgrown, the simpler code that has been replaced, or the complicated code that's been refactored.

Not capturing this knowledge is a huge loss. You put a lot of work into that code, so push as much of it as you can into the repo so everyone can benefit.

Write your git final commit messages as if a coworker wrote you an email and asked you to please summarize the changes you made. Capture those details.

Like Adam Savage from Mythbusters says: "The only difference between science and screwing around is writing it down."

Written by Stephen Ball Aug 19th, 2013 git, talk

If you enjoyed this article, please subscribe to the Rake Routes RSS Feed or signup to receive posts via email.

Tweet 11

« Customize your IRB

Comments

1 comment

0

Leave a message...

Oldest Community Share

tjstankus · 16 days ago

Sad I missed your talk. Thanks for sharing - great stuff!

1 Reply Share ›

Comment feed Subscribe via email

Recent Posts

- Deliberate Git
- Customize your IRB
- Program like a Videogamer
- Gem spotlight: interactive_editor
- Subscribing to RubyTapas using Downcast

About Me

My name is Stephen Ball. I started programming computers back in the 80s by copying BASIC programs from 3-2-1 Contact into our Atari 800. Now I program web applications using Ruby on Rails and CoffeeScript. I love every minute of it.

Follow @StephenBallNC

Github Repos

- Status updating...

@sdball on Github

Latest Tweets

- Status updating...

Follow @stephenballnc

My Pinboard

- How A Car Engine Works (animated infographic) engine infographic car animated animation
- API Directory - ProgrammableWeb api programming web via:Heliocentrist
- PAX: Homeworld: Shipbreakers Announced - IGNA new Homeworld game! And HD remakes! Woo!

My Pinboard Bookmarks »

Google+