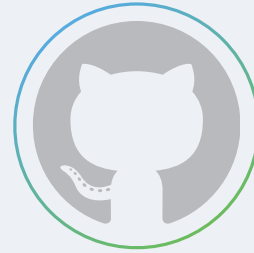


How  
uses  
to document

# GitHub



# How uses to document **GitHub**

Hi, how's it going. My name is Garen Torikian, and I work at GitHub. In case you don't know, GitHub is an online code sharing collaboration platform. A bunch of companies use it for their engineering and design teams to host and review changes they make to their code. Maybe your company uses it, if not, maybe they should.

Today I'm going to talk to you about how GitHub uses GitHub to document GitHub.

**This will be a  
talk about  
workflow.**

The title is a bit vague, so to make it more clear, this is going to be a talk about our workflow.

# **This will be a talk about our workflow.**

To be even more precise, this is going to be a talk about our workflow, using our product, which happens to be GitHub. Our whole foundation for writing and delivering documentation at GitHub has been a process that we're constantly tweaking. Our workflow is always evolving. The workflow in these slides are the result of making many mistakes. And this talk is going to be slightly opinionated, because I have opinions.

One of my main motivations for giving this talk is that I don't think there are enough blogs and talking heads out there discussing how documentation is written, and how we can improve upon our workflow. I think a lot of the process around writing documentation is taken for granted.

I want to hear about more workflows, I want to hear more about the tools people use to write their documentation. I want to know more about what people love about their processes. So in that spirit, this is our little contribution.

# **This *can* be a talk about your workflow.**

The fact that I happen to work for GitHub is irrelevant. If your company is using some other versioning tool—Perforce, a CMS, whatever—my hope is that you'll be able to leverage this talk to your advantage.

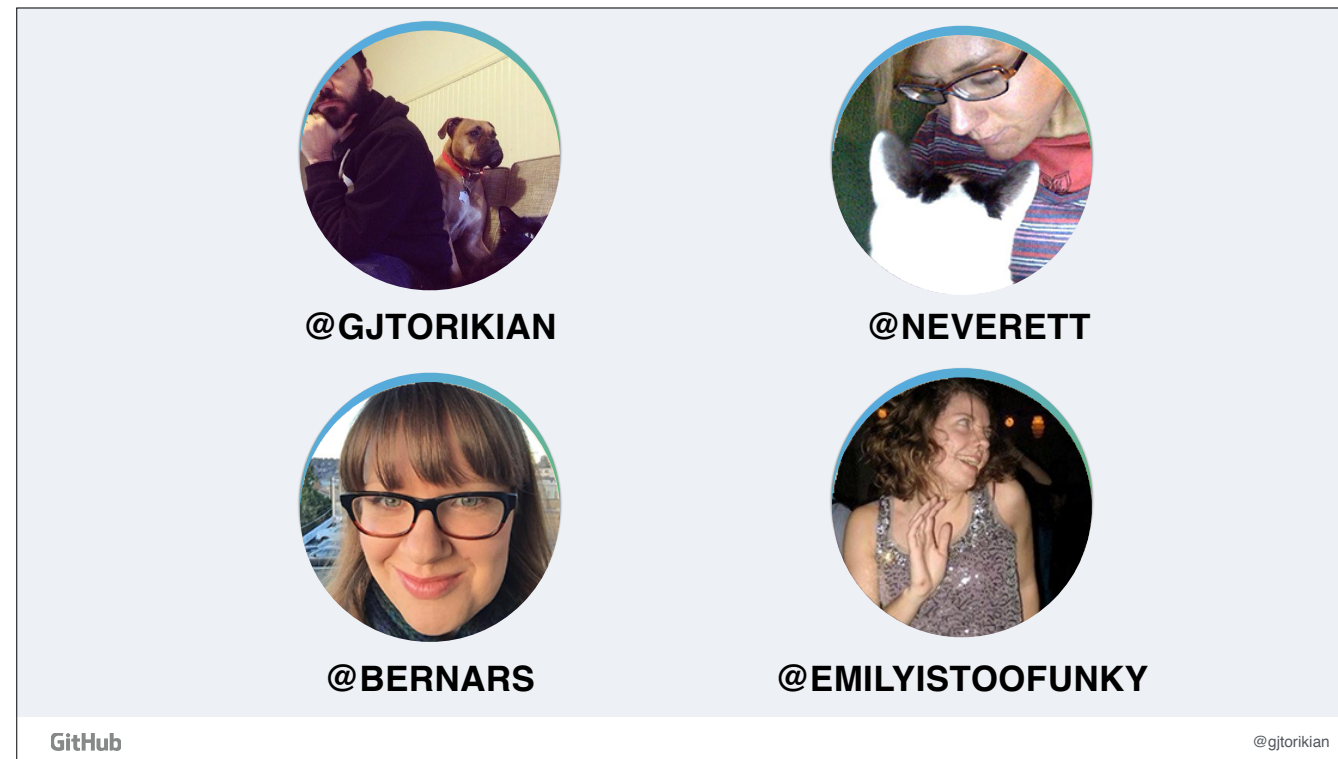
The problem is that, too often, delivering quality code is a single workflow, and delivering quality documentation is a completely separate process, and never the 'twain shall meet. GitHub was intended to be a product for engineers and open-source projects. A lot of tooling built around it still deals with shipping code, not content.

The Docs team at GitHub strove to follow patterns the company at large was using. Obviously, if your engineering team already using GitHub, then maybe you can take some of the ideas I'm presenting and pitch them to your thought leaders and decision makers.

**Fre**  
**e** **as in beer**  
**as in speech**

Above all else, GitHub writes and delivers documentation using open-source tools as well as some tooling built with our public API, so nothing I'm about to say is secret sauce.

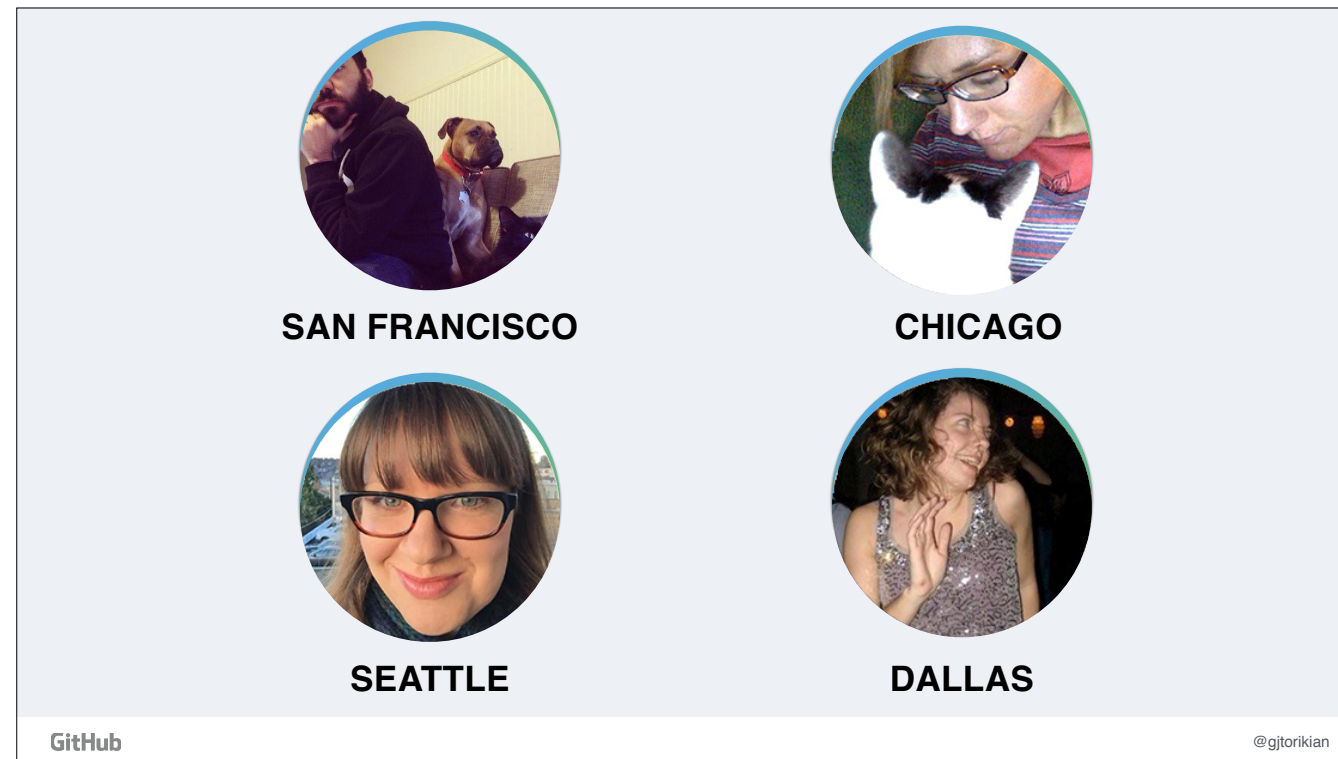
The sauce is already out there, and here's our recipe.



First, some real quick information about me, and why you should listen to anything I say.

I joined GitHub as the first tech writer. In a past life, I worked on tooling for DITA, I wrote plugins for various text editors, and I dabbled in Framemaker.

These three other disembodied heads and I represent the entirety of the documentation team. In comparison, GitHub has 115 engineers, support staff, and salespeople to assist.



It just so happens that our entire team is also remote. We meet up about two or three times a year to remind each other what our faces look like. But in addition, what this means is that all our communication is over the 'Net.



1. github.com
- 2. help.github.com**
3. gist.github.com

GitHub

@gitorikian

Another stat I usually like to toss around is that [help.github.com](https://help.github.com) is our second most visited website, right after [github.com](https://github.com), with several million page views a month.

So our writer-to-engineer ratio is about 1-to-30, and there's enormous visibility on our documentation. We basically had to come up with a workflow that was both efficient and accurate.

- 1. Write**
- 2. Review**
- 3. Build**
- 4. Publish**
- 5. Measure**

GitHub

@gitorikian

Our typical cycle probably matches how most everyone in this room operates. You write docs, you get someone to review it, you build it, you publish it.

The unique part, I think, is how we leverage bits of GitHub for all of this.

# 1. Write

GitHub

@gitorikian

We'll start at the first piece of the workflow, the writing.

# WRITE SIMPLY.

GitHub

@gitorikian

The first tenant our group emphasizes is that we adhere strongly to the idea of writing simply. At GitHub, writing simply is the idea that we prefer the five cent word to the one dollar word. We prefer short sentences over longer ones. Very rarely do we nest bullet points. We don't try to cram more than one idea onto a single page. GitHub the product is intended for a fairly technical audience, but we strive to speak like humans.

This leads directly into our next tenant, which is...

# **SIMPLY WRITE.**

GitHub

@gitorikian

...to simply write.

A few of us come from backgrounds where writing in XML is the standard for documentation. If you take a look at most documentation tools, they're either too complicated to understand or they're too simple to use.

At GitHub, we found the need to strike a balance between content reuse and a syntax that won't frustrate writers.

<pre>task id="install_emacs"&gt; &lt;title&gt;Installing GNU Emacs&lt;/title&gt;  &lt;taskbody&gt;   &lt;prereq&gt;Windows NT 4.0 or any subsequent version   disk space.&lt;/prereq&gt;    &lt;steps&gt;     &lt;step&gt;       &lt;cmd&gt;Unzip the distribution anywhere.&lt;/cmd&gt;        &lt;stepxmp&gt;&lt;screen&gt;C:\&gt; unzip emacs-21.3-bin       &lt;stepresult&gt;&lt;p&gt;Doing this will create an       &lt;filepath&gt;emacs-21.3&lt;/filepath&gt; directory.&lt;/p&gt;     &lt;/step&gt;      &lt;step&gt;       &lt;cmd&gt;Go to the bin subdirectory.&lt;/cmd&gt;        &lt;stepxmp&gt;&lt;screen&gt;C:\&gt; cd emacs-21.3\bin&lt;/s     &lt;/step&gt;</pre>	<pre>----- title: Installing GNU Emacs type: task prerequisites: Windows NT 4.0 or a  ----- 1. Unzip the distribution anywhere   \&gt;   C:\&gt; unzip emacs-21.3-bin-i386.z   \&gt;   Doing this will create an *emacs  2. Go to the bin subdirectory.   \&gt;   C:\&gt; cd emacs-21.3\bin   \&gt;</pre>
--	---

GitHub

@gitorikian

With simplicity as a goal, we use Markdown to do all of our writing. If you're not familiar with it, Markdown is a writing format designed to be written by humans.

To illustrate the point, on the left here is a traditional XML-based writing setup that probably every BigCo in the world has signed up for. On the right is the same set of instructions written in Markdown. Both of these formats produce the exact same output: a numbered list.

But only one of these is easier for a human to read, write, and review.

```
task id="install_emacs">
<title>Installing GNU Emacs</title>

<taskbody>
  <prereq>Windows NT 4.0 or any subsequent version
  disk space.</prereq>

  <steps>
    <step>
      <cmd>Unzip the distribution anywhere.</cmd>

      <stepxmp><screen>C:\> unzip emacs-21.3-bin
      <stepresult><p>Doing this will create an
      <filepath>emacs-21.3</filepath> directory.</p>
    </step>

    <step>
      <cmd>Go to the bin subdirectory.</cmd>

      <stepxmp><screen>C:\> cd emacs-21.3\bin</s
    </step>
  </steps>
</taskbody>
</task>
```

```
-----
title: Installing GNU Emacs
type: task
prerequisites: Windows NT 4.0 or a

1. Unzip the distribution anywhere
C:\> unzip emacs-21.3-bin-i386.z
Doing this will create an *emacs

2. Go to the bin subdirectory.
C:\> cd emacs-21.3\bin
```

GitHub @gitorikian

You can see how well-meaning but misguided the slide on the left is. The writer must remember to place a CMD tag within a STEP tag within a STEPS tag. The thing is, no one reading your documentation cares about these semantic details. The only thing that matters is that the reader gets a numbered list. A numbered list in Markdown is exactly how you'd expect it to be: 1, 2, 3, e.t.c.

I mean, forget all the tags! The hardest part about a writer's job should be worrying about how to produce the words, not how difficult it is to work the tooling.

Admittedly, Markdown is an incredibly loose structure—I'll talk a bit about that later. When we simplified our writing structure, the emphasis on consistency shifted onto the reviewer. Which brings us to...

# 1. Write

## 2. Review

GitHub

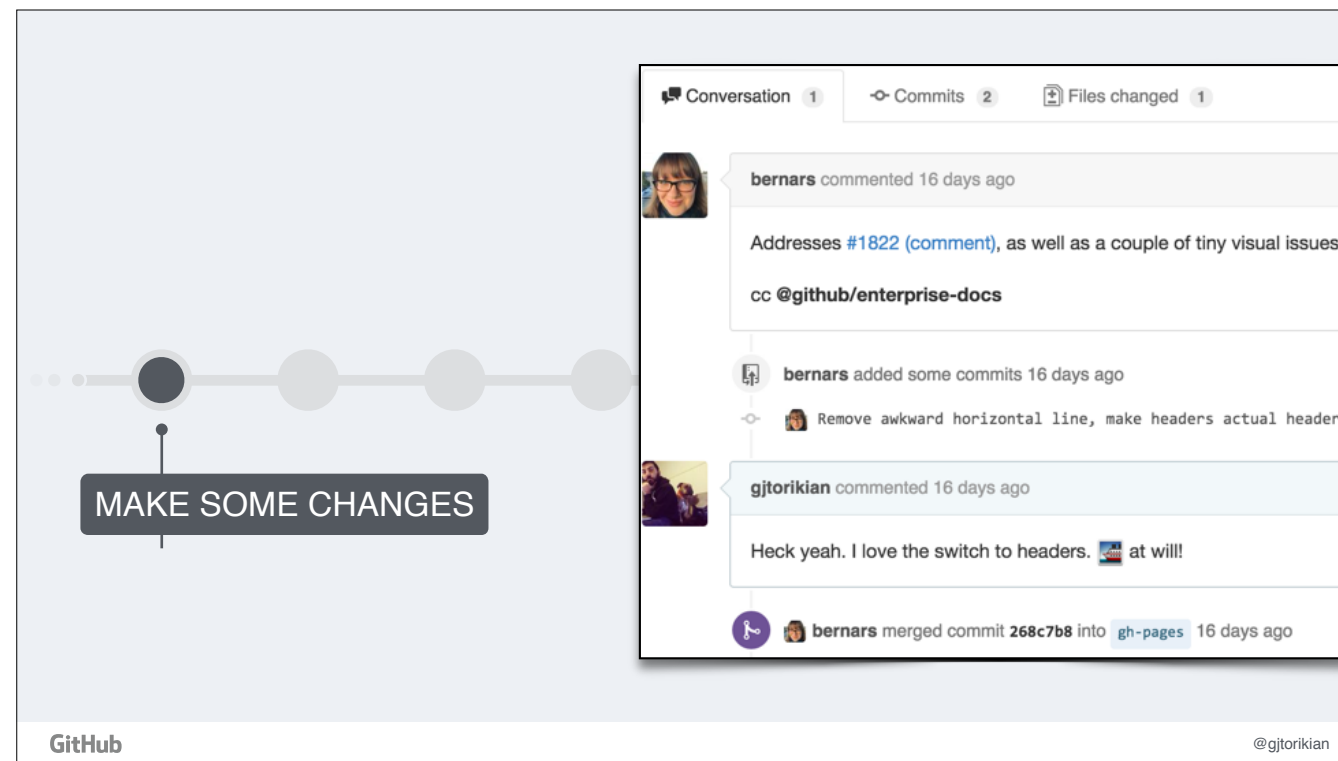
@gitorikian

The next part of the documentation workflow. Our review process takes as long as, if not more than, the writing process. It's also arguably more important. After you get the thoughts out of your head, you need someone else to sanity check them. If your coworker can't understand what you mean, your user sure won't.

In order to facilitate our content reviews, we use pull requests on GitHub exclusively.

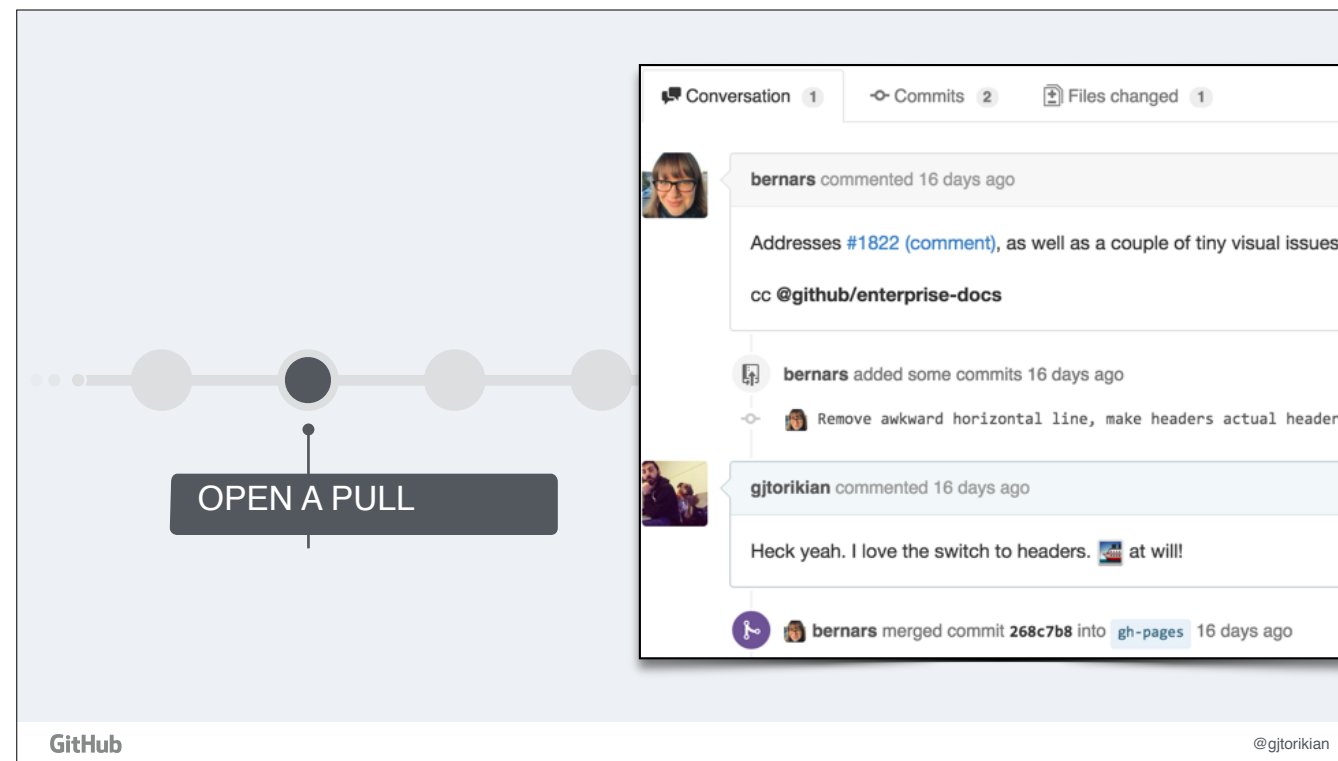
Maybe you're not using GitHub, so I'll do a very quick intro on what a pull request is:





A pull request is basically a way to introduce a change.

In this example, Sheri wants to make a couple of changes to our documentation, so she edits a few files.



She then opens a pull request on GitHub so that the whole team can see the proposed change. She'll write up a little comment describing what it is she's changing.

The image shows a GitHub pull request interface. On the left, a commit history timeline with four circles, the third of which is highlighted. A dark grey box with the text "DISCUSS IT WITH" is positioned over the timeline. On the right, a sidebar titled "Conversation" shows a list of comments and commits. The comments are from users "bernars" and "gitorikian". The commits are from "bernars" and "gh-pages". The sidebar also shows a "Files changed" section with one file changed.

Conversation 1 Commits 2 Files changed 1

bernars commented 16 days ago

Addresses [#1822 \(comment\)](#), as well as a couple of tiny visual issues.

cc @github/enterprise-docs

bernars added some commits 16 days ago

Remove awkward horizontal line, make headers actual headers

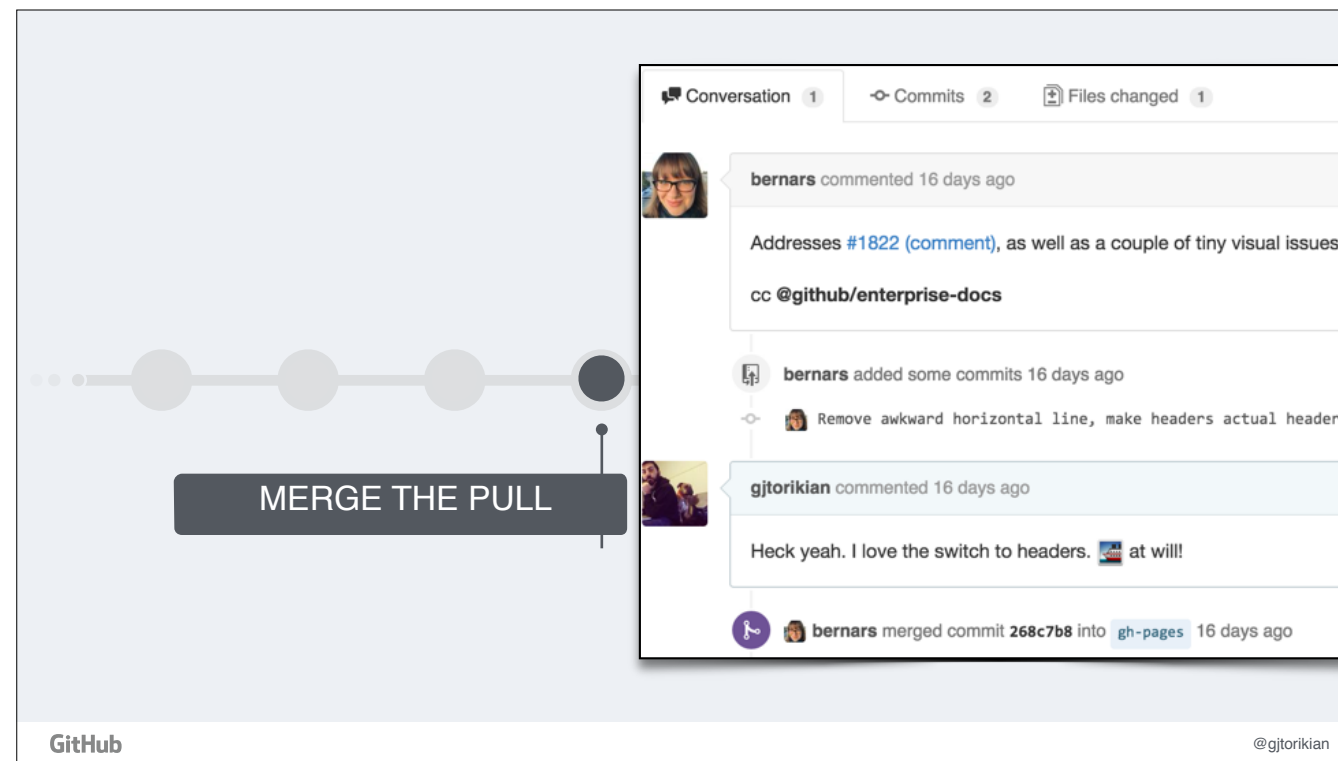
gitorikian commented 16 days ago

Heck yeah. I love the switch to headers. 🎉 at will!

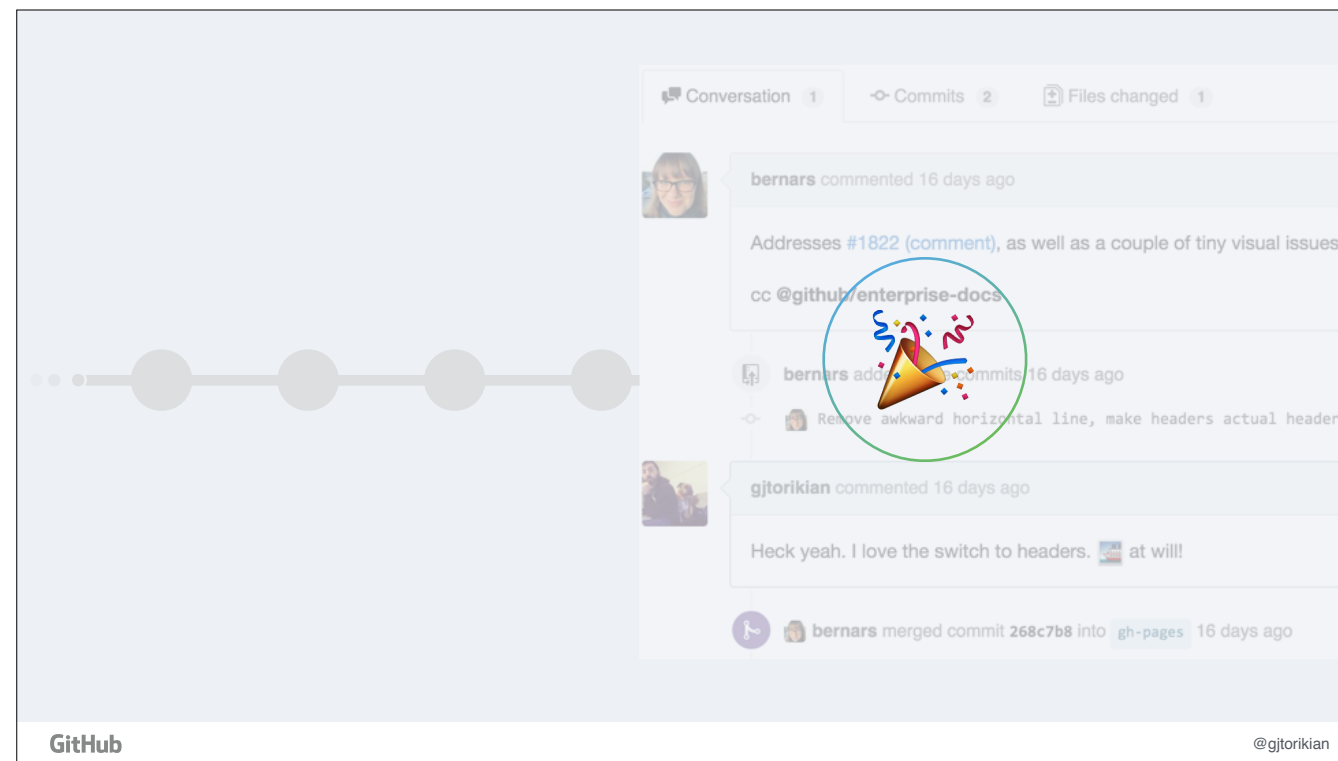
bernars merged commit 268c7b8 into gh-pages 16 days ago

GitHub @gitorikian

Others inside and outside the team can comment on the change, and offer feedback.




When she's ready, she merges the pull request.



And ta-da! Once a PR is merged, it's considered part of master and is published.

39		-7. In Terminal, [add the URL for the repository](/articles/adding-a-remote) your local repository will be pushed.
	42	+8. In Terminal, [add the URL for the repository](/articles/adding-a-remote) your local repository will be pushed.
GitHub <span>@gitorikian</span>		

If you've done a PR before, or basically committed any doc update, you might think the most important part of it is this, being able to visualize the change.

39		-7. In Terminal, [add the URL for the repository](/articles/adding-a-remote) your local repository will be pushed.
	42	+8. In Terminal, [add the URL for the repository](/articles/adding-a-remote) your local repository will be pushed.
 neverett added a note on Mar 6		
Really what we want here is "repos'tory".		

But it's actually this. Being able to comment on that change.

# PULL REQUESTS ARE DISCUSSIONS.

GitHub

@gjtorkian

Pull requests are discussions. The changes within a pull request are important, but the real power in a pull request comes from talking about those changes. If you're using Perforce or Worldsever or some CMS, making a change is one thing, but being able to discuss those changes with your team in a permanent matter is what is most powerful.

With pull requests, the discussion and the change take place at the same time. You don't need to link the two together. There's never any question as to why a change was made. This holds true for changing code and it holds just as true for changing content.



# LESS EMAIL, MORE PULL REQUESTS.

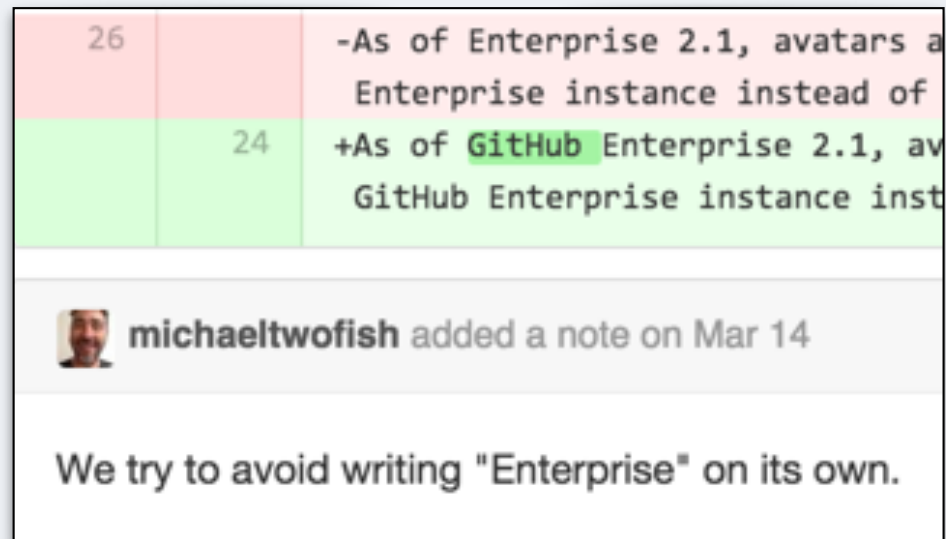
GitHub

@gitorikian

We're super keen at GitHub on being as transparent as possible. We try and discuss as much as possible out in the open, and keep our decisions public for everyone in the company.

We shy away from using email and other backchannel modes of communication and prefer our decision making to take place in a pull request. Right, so, email is a terrible pit from whence there's no escape. There are a few key ways in which discussions within pull requests are more valuable for our team.

# TEACH BY DOING.



GitHub

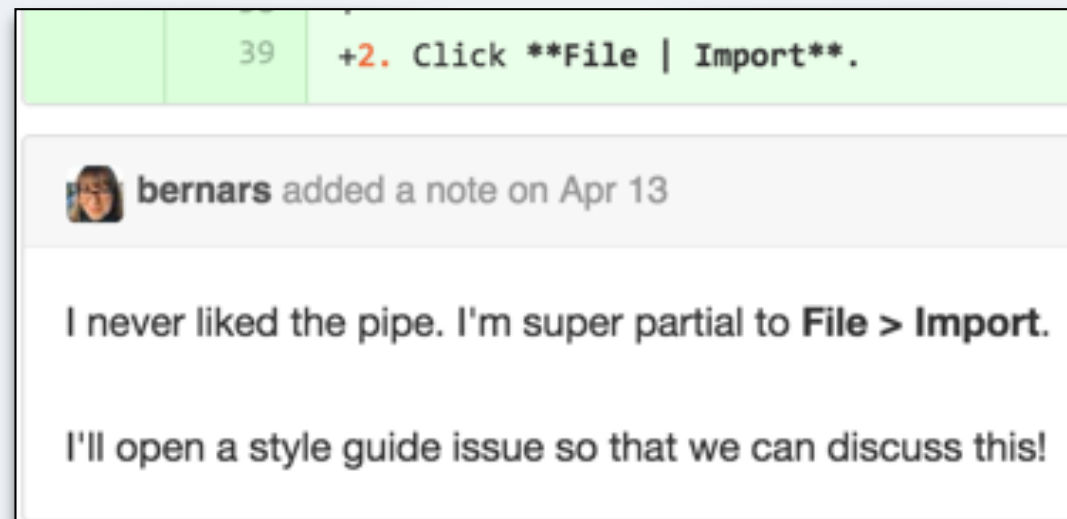
@gitorikian

First, it serves as a public forum for demonstrating to everyone inside and outside the team how we communicate, and how we arrive at our decisions.

We don't need to teach people how we write. After enough lurking and observing our ways, people outside of the team regularly jump in to correct the content. Typically these are people in support or training who are regularly perusing our documentation with customers.

We gain tiny contributions without making any attempt to do so. In this example, Michael knew that a term needed to be updated because he'd seen us discussing it before. He made the change himself, and those sorts of micro-updates free the team up to address larger structural issues with the documentation, while still enabling others in the company to participate.

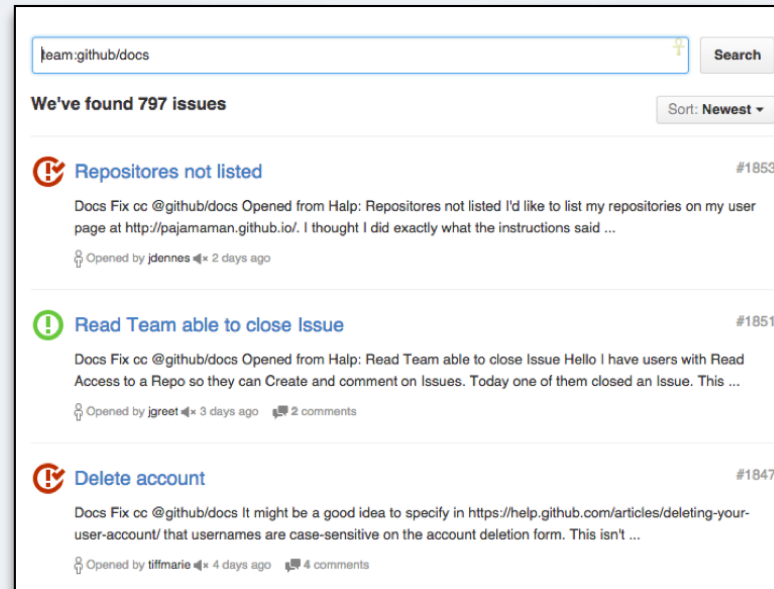
# DISCUSS AND



With a discussion that's out in the open, there's also no confusion as to why a certain feature was documented a certain way.

You can link back to any comment in a PR. It's not at all uncommon to have a discussion and link back to some previous comment that occurred months ago. Instead of pondering why a certain style was adopted, we can all see the comment where a decision was being made.

# URLS LAST LONGER.




GitHub

@gitorikian

Every pull request is a discussion with a linkable URL. A URL is something that lasts \*forever.\* The newest hire to your company probably won't have access to any old team emails, so any previous discussions will be completely lost on them. I can go back through years of doc changes and style decisions with pull requests.



You can compare this approach with something like Google Docs. Google Docs also have URLs that you can link to and share. But the content in the doc completely misses the context and reasoning behind it. Holding on to that context strengthens the team down the road. Human's memories are terrible, so it's incredibly handy to be able to cite past conversations.

# BTW: I'M DONE.





emilyistoofunky commented on Feb 26

Addresses @izuzak's [helpful reminder](#) in [#1599](#) that




Punch cards don't show merge commits either



emilyistoofunky referenced this pull request on [Feb 26](#)

**[Ready for review] Bugfix day contributi**



izuzak commented on Feb 26

@emilyistoofunky ❤️ you're my hero!

GitHub

@gitorikian

The other thing you can do in a pull request is directly ping people.

You're probably familiar with the @mention syntax from Twitter or Facebook. If you @mention a GitHub user, it'll send them a notification via email or through the web.

# GET FEEDBACK EARLY.

I'm not totally sure what capitalization scheme to use in these articles (Pages vs pages), so I just did what seemed to make sense for each use of the word "page" or phrase "User Page(s)" or "Project Page(s)".

/cc for review 🔍

**@github/docs**

**@github/enterprise-docs**

**@github/enterprise-support**

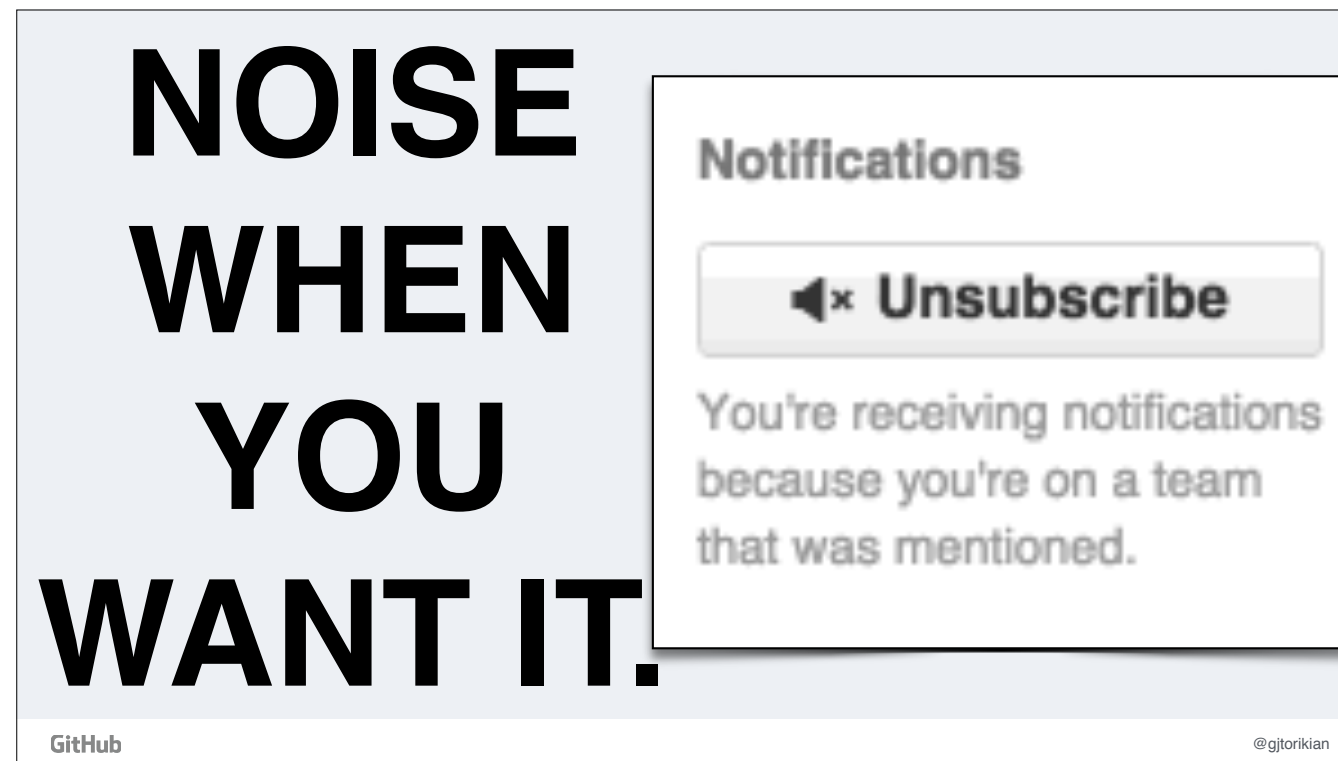
GitHub

@gitorikian

We're tending to use individual @mentions less and less. A much more powerful evolution is @mentioning teams. A team is what you'd expect it to be: a group of individuals interested in a topic. We have teams for the Docs group as a whole, as well as smaller focused teams for the Enterprise Docs and our Platform documentation.

Zooming out, we have teams revolving around various features, we have teams for security and legal, we have teams for designers...everyone is on a team of some kind.

When a feature is in the process of being written up, we tend to ping teams involved in a feature for additional technical review. This allows us to expand the scope of our review beyond the documentation team, and gets the people building the feature directly involved. It works both ways, too. When a feature is in the process of shipping, we'll often get pinged by engineers. Or at least, that's the goal, anyway.



But the absolute best part about pull requests over email is this little button.

Never underestimate the power of unsubscribing. Most of us creep around on various engineering teams in order to keep abreast of what's going on. Usually, other teams will initiate a discussion that has no bearing on the documentation. It's incredibly useful to know what's being worked on, but not incredibly helpful to remain on the thread. So we unsubscribe ourselves.

You can't really do that in an email. The problem with email is that it just continues to grow and grow, like a snowball rolling down a mountain. By the time it passes the PM, passes the engineering team, passes the designers, and reaches you, it's created a massive avalanche. You get buried in it. We're a pretty small team. We trust each other. If a feature is being picked up by someone on Docs, the rest of us usually bow out and find something else to work on.

1. Write
2. Review
- 3. Build**

All right, so it's taken me a while to get to my favorite part of the workflow. I could do a whole talk on just documentation build tooling.

Every website in the world, from the largest social network to the tiniest startup, is composed of just HTML pages. That's it. And the whole point of HTML is to take a page that's written by humans and turn it into something that's readable by a computer.



# **BUILD DOCS LIKE IT'S 1999.**

GitHub

@gjtorkian

Much like we keep our writing simple, we prefer to keep our build process simple, too.

There are a bunch of complicated techniques on the Internet that are used to assemble webpages, but I've yet to be convinced that such techniques are relevant for documentation.

- **XHTML**
- **PDF**
- **ODT**
- **Eclipse Help**
- **TocJS**
- **HTML Help**
- **Java Help**
- **Word RTF**
- **Docbook**
- **Troff**

GitHub

@gitorikian

As an example, here's a listing of every single output format that the DITA-Open Toolkit is capable of producing. That's ten different output formats, generated by one horrendous XML markup.

In almost ten years of writing technical documentation, I've only ever needed two formats.

- **XHTML**
- **PDF**
- ~~ODT~~
- ~~Eclipse Help~~
- ~~TocJS~~
- ~~HTML Help~~
- ~~Java Help~~
- ~~Word RTF~~
- ~~Docbook~~
- ~~Troff~~

GitHub

@gitorikian

What happened with DITA is what destroys every other software project: someone said, Hey, HTML is great, but what I really need is my output in Word. Or what I really need is my output in Troff. Or what I need is this new format for this one specific use case.

And I assume, people kept introducing new build formats, because people kept asking for them. It's not just DITA, though. A lot of doc tooling is made by people who want something to do everything. Sphinx, for example, can also spit out LaTeX and ePub, even if you don't need it to.

**THE MORE A TOOL TRIES TO  
DO, THE MORE COMPLICATED  
IT CAN BE TO GET IT TO DO  
WHAT YOU NEED, BECAUSE  
AFTER A WHILE IT TRIES TO  
DO TOO MUCH AND THINGS**

GitHub

@gitorikian

It's a problem when your tool supports ten different output formats, and you only really care about two of them. Nothing can do ten different things well...except maybe a swiss army knife.

Like I told you: opinionated.

But to put it another way:

**DO ONE THING,  
BUT DO IT  
WELL.**

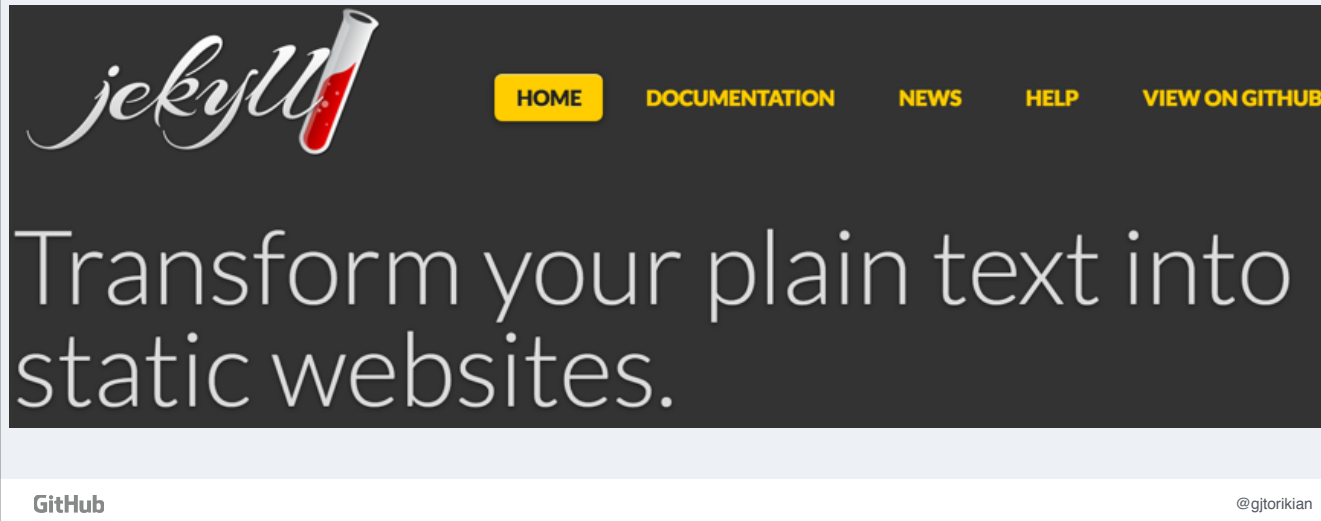
GitHub

@gitorikian

Do one thing. Do it well. That's it.

So when we talk about documentation, what's the one thing you want to do, as an author? You want to write. What's the one thing you want your build to do? You don't want to fight with ten different output formats when you only need one. You just want to build your content.

# WE USE JEKYLL.



With that in mind we use a build tool called Jekyll to create our web pages.

It takes the markdown documents that we write and it turns them into HTML. That's it. It doesn't try to do anything fancier.

Jekyll is cool in that it's an open-source project GitHub has contributed to, but doesn't maintain.

- **Support for content reuse**
- **Huge community of users**
- **Immense plugin library**
- **Live reloading**
- **Easy integration into GitHub**

GitHub

@gjtorkian

With Jekyll, we get a lot of features built for us. There's a huge library of plugins we can take advantage of.

In terms of writing, Jekyll has everything you'd expect from a mature writing process. It supports content reuse, which is essential. Our builds are incredibly quick.

And the tool just happens to integrate easily into GitHub. As a side note, Jekyll is far from the only tool that supports these points. It just happens to be the one that we use.

# SIMPLIFIED

```
{% if page.version == 'dotcom' %}
### Configuring authentication via text message

If you're unable to authenticate using a TOTP mobile app, you may use
SMS. For more information, see [Configuring two-factor authentication
via text message](/articles/configuring-two-factor-authentication-via-text-message)

{% endif %}
```

GitHub

@gitorikian

As a quick example, our documentation targets two different products: [GitHub.com](#) & GitHub Enterprise, which is sort of like an on-premise version of GitHub the website. GitHub Enterprise requires us to version our documentation. Features available in one product might not be available to another. When we're writing, we have to keep multiple content outputs in mind.

In order to support writing content for two products, we wrap our Markdown in versioning blocks, like the one shown here. This chunk of text will only show up in our 'dotcom' output. This is all pure Jekyll, we didn't do anything. To do this same technique in XML, you'd probably need to add some kind of attribute to your section tag to exclude it.

Keeping the versioning in line with the text like this really helps our team better visualize and produce content.



# EVERYONE CAN BUILD.

GitHub

@gjtorkian

Since Jekyll is so popular, it's really easy to set up on any machine. Keeping your build tools simple means that everyone at GitHub can build the documentation and contribute to it.

Because the build is so easy, it lowers the barrier for other types of contributions. Remember earlier when I mentioned the Support member who made a change to the docs? Just last week a designer saw a layout issue that bugged him. He made a visual change to the site, verified it with a build, and opened a pull request. He fixed what he wanted to without learning a brand new workflow. Quick documentation builds let people focus on doing their work, not on waiting for output.

# TEST THE BUILD.



With such a loose goosey, “anyone can make a change!” system, one other advantage that a quick build gets us is the ability to also quickly test our content. We have a pretty exhaustive test suite that we run on every change. We test for everything imaginable that can go horribly wrong. We make sure that all of our links are working. We test that images aren’t broken. We test that our content references are valid. We test that our site’s drop-down menus are still clickable. We test everything! This, again, I think ,is the advantage against DITA. We run tests on our HTML output, which ensures that the content is valid. We may as well switch the way we write to AsciiDoc or MediaWiki in the future and that’ll change nothing about our builds.

If we’re going to get outside contributions from support or designers, we want to make sure that those changes are valid, without again burdening the docs team to proofread everything. We see the status of a change on every single push to GitHub, and it helps keep us sane. More importantly, it keeps our documentation accurate for readers.

# TEST THE BUILD.

- **HTML-Proofer**
- **Capybara / Selenium**
- **GitHub Commit Status API**

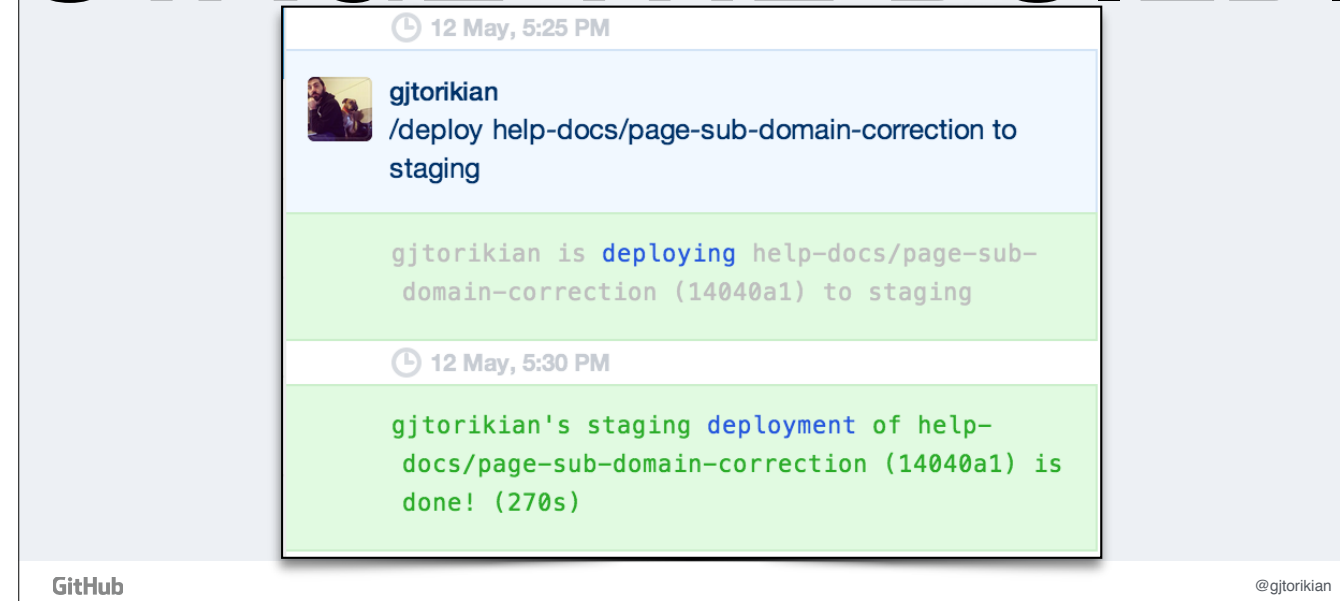
GitHub

@gitorikian

In order to run our tests, we use a tool called HTML-Proofer to run through our content. We use a library called Capybara to verify the visual elements of the site, which is sort of like Selenium, if you're familiar with that. All of these are hooked directly into the GitHub Status API.

The same GitHub status API tooling is used by our engineering teams to run tests on their code. So for us, it's a matter of mimicking tools that engineers have been using and applying them to our documentation process.

# STAGE THE BUILD.



But sometimes, even if you do test the build locally, and get a five-star rating from your reviewer, you want to be able to distribute the content you've written to a larger internal audience. In our case, we deploy our documentation to a staging server. Our staging server is a private URL that is accessible only to employees. We can deploy a branch to it, and then distribute the link internally.

Typically, we perform a staging build for really big feature ships, and we have people on our marketing and engineering teams roll through the documentation to make sure we're not completely fibbing.

# STAGE THE BUILD.

- **Jekyll-Auth**
- **GitHub Deployments API**

GitHub

@gitorikian

We use a plugin for Jekyll called Jekyll-Auth which limits access of our site to just GitHub employees.

Deployments happen using the GitHub Deployments API. Since our sites are just HTML, we can send them to any platform in the world. And again, many many engineering teams the world over have already hooked into this API and have all sorts of strategies and tooling for deploying websites.

1. Write
2. Review
3. Build
- 4. Publish**

GitHub

@gitorikian

Okay, so you've written your content, you've gotten a thumbs up on your pull request, the build looks fine, now it's time to publish.

In other documentation teams I've been a part of, publishing takes hours, with an entire team dedicated to the release process. One screw-up in the deployment could set back your evening.

# YOU MERGE, IT DEPLOYS.

A green rectangular button with rounded corners, featuring a white fork icon on the left and the text "Merge pull request" in white on the right.

Merge pull request

GitHub

@gitorikian

At GitHub, we take a simple approach: once your pull request is merged, the content is live. That's it. No one has to think about it. There's no additional automation around it. There's no dedicated team to handle the process. You click a button, the content goes live.

**JEKYLL +  
GITHUB PAGES =  
DOCUMENTATION.**

GitHub

@gitorikian

We do this using a feature called GitHub Pages. GitHub Pages is completely free, and available to every single GitHub repository.

GitHub Pages is basically a hosting platform for static sites. That's it. It does one thing and it does one thing well. It hosts websites. It hosts all the HTML that came out of your Markdown and serves them to millions of users.

Not only is our Help documentation hosted on GitHub pages, but a bunch of GitHub's marketing site is hosted there too. Every user gets a bunch of stuff for free, like support for HTTPS, and assets served by a CDN.

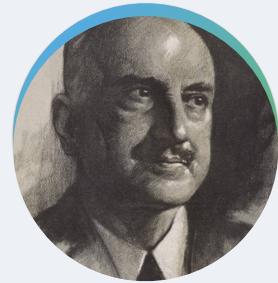


1. Write
2. Review
3. Build
4. Publish
- 5. Measure**

GitHub

@gitorikian

The last step in our documentation workflow is “Measure.” There’s a quote I like to look fondly upon whenever it’s time to review this step:



“

***THOSE WHO CANNOT  
MEASURE THE PAST ARE  
CONDEMNED TO REPEAT IT.***

- GEORGE SANTAYANA

GitHub

@gitorikian

I'm pretty sure that this quote is 100% accurate as-is.

Here's the dirty secret about documentation: sometimes there's too much of it, and sometimes it goes stale. Sometimes a feature that's been in the product forever has five different pages of documentation explaining what it does.

Is anyone actually getting value from your content? How would you know?

**MEASURE  
ALL OF IT.**

GitHub

@gitorikian

You measure your site.

# MEASURE VIEWS.

GitHub

@gitorikian

Measure the number of page views.

# MEASURE CLICKS.

GitHub

@gitorikian

Measure the number of clicks.

# MEASURE TIME.

GitHub

@gitorikian

Measure how long people are on the page.

# MEASURE SPEED.

GitHub

@gitorikian

Measure how fast it takes the page to load.

# MEASURE TICKETS.

GitHub

@gitorikian

Measure the support burden introduced by the page.



# MEASURE ALL OF IT.

GitHub

@gitorikian

Measure your website. Measure all of it. This can be as simple as hooking up Google Analytics to key parts of your site. GitHub is a data-driven company, so there are tons of people who already analyze all the data that comes in. But I guarantee that every project manager already knows how to do this.

Because what'll happen is your documentation site will grow and grow with new content, and when it comes time to trim it down, you won't know which parts you can keep, which parts you can consolidate, and which parts you can simply throw away.

# GRAPH ALL OF IT.

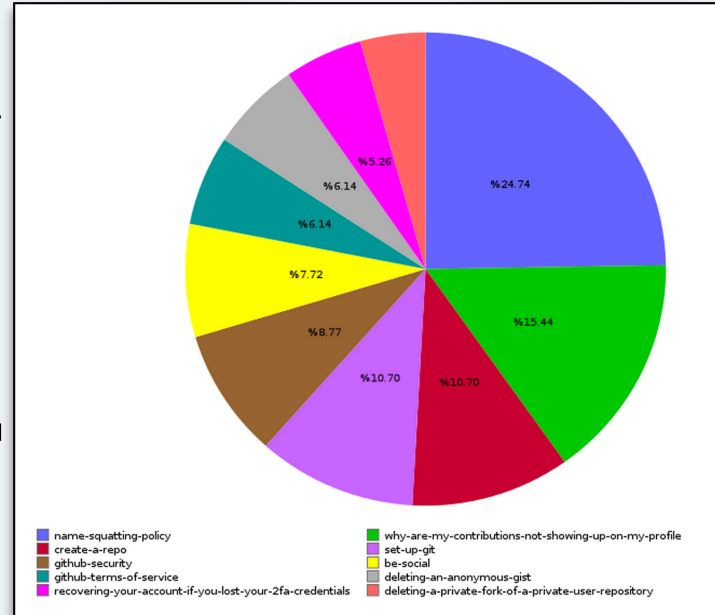
GitHub

@gitorikian

We consolidate a lot of documentation at GitHub, for the websites, for error messages on Git, for documenting Pages and Gists and our Desktop apps. In order to know which docs are important, we needed to be able to turn all the confounding numbers into graphs to help you process them.

You can hypothesize the data all that you want, but without graphing it, it's meaningless for a human to interpret.

# NUMBERS INTO PICTURES.



GitHub

@gitorikian

It's time for documentation teams to make use of the kinds of analytics available to features. Otherwise as your site grows, you end up doing a disservice to your users.

You're familiar with the sound of one hand clapping? What about the well-written documentation that no one can find?

This specific graph shows the volume of tickets generated by specific documentation articles. It tells us which articles are sending most readers to write into support. Our documentation team, in conjunction with our support team, can routinely go back to product teams and engineers and point out which features are simply broken, based on the support volume coming from the docs.

Either the doc is bad, or the feature is bad, and there's only so much the docs can do. The data can prove this.

- 1. Write**
- 2. Review**
- 3. Build**
- 4. Publish**
- 5. Measure**

GitHub

@gjtorkian

So that's it. That's how we use parts of GitHub to go through the five-step workflow process.

A lot of the ideas I went over were completely novel to GitHub even just a year or two ago. Testing your output? Measuring the views? Although the tooling for these steps existed for teams within the company, it took some discovery and effort on the documentation team to highlight the value in it for us.

- 1. Markdown**
- 2. Pull Requests**
- 3. Jekyll**
- 4. GitHub Pages**
- 5. Analytics**

GitHub

@gitorikian

Even though our specific nitty-gritty looks like this,

- 1. AsciiDoc**
- 2. Skype**
- 3. Nanoc**
- 4. Heroku**
- 5. KISSMetrics**

GitHub

@gitorikian

You know, maybe one day it'll look like this

- 1. Word**
- 2. Carrier Pigeon**
- 3. Pandoc**
- 4. Azure**
- 5. MixPanel**

Or maybe it'll look like this

# **SIMPLY WRITE.**

GitHub

@gitorikian

But the details are irrelevant. At the end of the day, it's all about being able to simply write.



*Thanks!*



*@gjtorikian*

Thanks.