Modern Technical Writing

< An Introduction to
Software Documentation >

Andrew Etter

Modern Technical Writing

Modern Technical Writing

Introduction

Given the pace of the software industry, I'll probably regret calling this book *Modern Technical Writing*, but *Technical Writing in 2019* or *What Currently Passes as Sane Technical Writing* weren't any better. Books like this one have a short shelf life unless a subject matter expert meticulously maintains them, which I don't intend to do. Still, I imagine the book will hold its value for the next five years—probably long enough to make writing and reading it worthwhile.

I had the idea for this book a couple years after I started working as a technical writer. I had just finished interviewing some very nice, very *experienced* people from two massive Silicon Valley companies, and we just could not find common ground. Their impression of the job was that technical writers interviewed engineers, took copious notes, wrote drafts in Adobe FrameMaker, and waited several hours for some arcane process to

build those drafts into printable books. These candidates mentioned page counts, XSLT, Perforce, and gerunds. They were proud that their page numbers were laid out recto-verso.[1] None of them seemed to give any thought to their readers' actual needs, preferring their own criteria for what constituted a job well done. They used phrases like "clear, concise, correct, and complete" and avoided words like "searchable," "scannable," "attractive," and most egregiously, "useful."

These candidates weren't stupid; they were articulate and pleasant, well-educated and inquisitive. No, I believe they were products of a dysfunctional profession, a profession of misplaced priorities, judged for too long on meaningless criteria by managers who treat the *Chicago Manual of Style* like a holy text and would rather produce minimal value from a lot of work than tremendous value from far less work. Exceptional technical writers are force multipliers within the software industry. Great documentation makes new hires productive in days instead of weeks, prevents thousands of calls to customer support, is the difference between crippling downtime and rock solid stability, and

inspires true, fervent love of development platforms. Technical writing *matters*, so I wrote this book to help people create documentation in a sensible way.

To be crystal clear, this book is about technical writing in the software industry. Most of the principles are universally applicable, though. If a particular recommendation doesn't seem relevant to your work, please take a mental step back and try to assess the high-level principles behind the recommendation. Maybe I'm completely off base, or maybe some kernel of wisdom is buried in my insufferable prose. In other words, please don't dismiss a good idea because of bad execution.

And thanks for reading.

— Andrew Etter

---

1. Recto-verso just means using slightly different layouts for the "right" and "left" pages of a book.↩

Basics

Like all writers, technical writers aim to produce content that their intended audiences will read and find useful.[1] This has to be the goal, because any other goal is absurd. Keeping it in your head at all times will help you produce more valuable work in any profession. Consider a few common misconceptions about technical writers:

- Technical writers produce *formal* documentation.

In this case, "formal" means unnecessarily stuffy and repetitive, the sort of writing that causes people to *skim* rather than *read*. When your ideal readers are tee shirt-clad software developers who mainline Hiball Energy and spend most of the year looking forward to Burning Man, it's safe to assume a more casual writing style.

Note

This does not give you license to write in an unprofessional or sloppy manner. Writing should not call attention to itself by being too formal or too casual. Occasionally reminding readers that human beings produced their documentation is fine. Making readers think that they are reading rough drafts is not.

- Technical writers produce *comprehensive* documentation.

Writing should be the minimum possible length. Oddly, most students learn this guideline in grade school and have forgotten it by grad school. If a help system must be truly comprehensive, it should be because readers will accept nothing less, which is untrue 99% of the time. Huge blocks of writing look intimidating, and excessive content waters down useful content. Identify what the audience actually needs to know, and include *only* that.

- Technical writers help protect companies from lawsuits.

No, lawyers do that, and believe me when I say that they're perfectly comfortable composing their own indecipherable text. Technical writers, at the absolute most, should *collate* legalese into their websites. You should take content verbatim from actual attorneys and spend no more time on it than it takes to press Ctrl + C and Ctrl + V. Then show them how to update the content themselves so that you can get back to your real job.

- Technical writers produce print documentation.

    Any piece of print documentation should be an absolute minimalist *Getting Started* guide (complete with requisite legal boilerplate) that directs its recipients to a web page. People are comfortable typing `www` into a web browser if they need help.

Unfortunately, producing content that people will read and find useful is, like, *really hard*. Here are the basics.

Don't Write

Don't write often, anyway. Technical writers, first and foremost, are testers and researchers. Your job is to know what people want to achieve and precisely how to achieve it. Communicating that knowledge is the last step of the process and really shouldn't comprise more than 10% of your time. When people say, "I was writing all day," they don't mean they were intermittently typing for eight straight hours. They mean they spent the entire day engaged in the writing *process*. And a big part of that process is installing, configuring, and testing software. In other words, learning.

Remember, the job of a technical writer is not to find bugs. Sure, you *will* find bugs, and if you're not aware of any, you probably don't know the product very well. You should also report these bugs to developers. But the main point of a technical writer performing testing is not to improve product quality, but to increase personal knowledge. Your testing should be centered around making you smarter.

Once you have a complete grasp of how something works, you can stop. Unlike a tester, you don't have to go any further. You don't have to reproduce tricky issues or check for unforeseen regressions in functionality. This difference makes your testing less tedious and more selfish than that of a formal quality assurance process, but because the ultimate goal is to teach others the material, you can pretend to be an altruist.

Be sure to leverage all the wonderful writing that developers and testers produce, even if it's half-complete, riddled with typos, and on a wiki page entitled "integrating with shamrock =D." Check the source repository for any files named `README.md`, and run any scripts with the `help` argument. Google any underlying open source software. Check Wikipedia for unfamiliar terms. Take notes. The learning process is time-consuming, so don't be discouraged if your measurable output is essentially nil for days or even weeks after moving onto a new project.

Contrary to popular belief, there are all kinds of stupid questions, or at least ignorant ones. The goal of all this

testing and research is to be mentally equipped to ask pointed, intelligent questions to business development, product development, quality assurance, customer support, and ideally, actual users. For business reasons, this last group might not be accessible, but try your best to get in touch with some of them.

Talking to people is an art. A tester might want to gripe about a useless or buggy feature, whereas a developer might anxiously justify architectural decisions. A product manager probably wants to discuss long-term vision and will be upset if you didn't schedule a meeting, whereas a salesperson might be perfectly happy to chat about user frustrations from adjacent bathroom stalls. A good rule of thumb is to treat everyone as busy individuals whose time is valuable, but another rule of thumb is to break this rule and be more of a jerk if an essential employee isn't giving you the information you need to do your job. Tailor your approach to the person, come prepared to meetings, batch your questions into related sets rather than bothering people every twenty minutes with individual questions, and thank everyone for their time. Your contribution to

a project isn't immediately obvious, so try not to be put out if people aren't as forthcoming as you'd like. Technical writers need to be grinders, willing to slowly but surely chip away at the block of ignorance until a beautiful sculpture emerges—or something like that, anyway.

After all that testing and research, you have to verify everything that people told you through additional testing. Tedious, but essential. At this point, though, you're well on the way to defining your audience.

Define the Audience

Writers have to make assumptions in order to achieve anything useful. Bare minimum, we assume that people can read.[2] Similarly, you should assume that your readers know how to use a trackpad and won't freeze up when the Enter key reads Return. Assuming anything less leads to writing like this:

> Place your hand on the mouse and, moving your hand (and arm, as necessary), direct the white

> cursor on your monitor. If your monitor is not on, you can turn it on by pressing and releasing the power button. The power button icon can be found in Appendix D: Icons. If your mouse has multiple buttons, press and release the leftmost one. If your mouse has one or fewer buttons, press and release either the only button, or the left section of the mouse, respectively. If you are using a trackpad...

A colleague once argued to me that you sometimes have to target your writing to just such an audience. The proper response to such an assertion is to let your eyes bulge out of their sockets and shake your head from side to side, but if you must reply, the simple counter-argument is this: you cannot help people who require that level of hand-holding. They probably won't read the documentation, and even if they do, they definitely won't understand it. There's nothing you can do for them. Better to say:

> The **Settings** menu lets you adjust mouse sensitivity and button mappings.

This statement involves a lot of assumptions. We're assuming the reader can find the menu, knows what mouse sensitivity is, and can infer that some button behaviors are customizable. We're also assuming the reader knows the basics of manipulating a computer and has the application open. I'm happy to make all of these assumptions, regardless of audience. If an application has a lot of button groups and only a few are customizable, you might need to be more specific, but the level of detail is basically fine. Don't get stuck in the quagmire of trying to help the helpless. I know it sounds cold and mean-spirited, because these are the readers most in need of assistance, but by catering to them, you are doing a real disservice to the other 90% of the population.

Now that we've defined who is *not* the audience, we can roughly divide readers into three groups: users, administrators, and developers.

Users are people who just want to achieve something with an application, whether it's resizing an image, creating a spreadsheet, or sending an email. Users are typically concerned with inputs and outputs. They type

a few things, drag a couple boxes around, and voila, a monstrosity of a PowerPoint presentation is born.

Note

A user interface is not a requirement for someone to be a user. Command line applications have users, too, people who just want to copy and paste some common commands and move on with their lives rather than mastering the syntax, chaining commands, and incorporating regular expressions.

Administrators install and configure applications. I'm not just talking about server administrators or people with administrator permissions on a computer, but anyone whose concerns are setup and maintenance. How much disk space does the application server fill over time? Where are the logs located? How do you recover data in the event of a crash? What's the simplest way to perform backups? How do you troubleshoot poor performance? In consumer applications, the user is often also the administrator. In business applications, this situation is less common.

Developers extend applications or interfaces with code. They want to improve some aspect of an existing application or build something new. More than anything, they need reference materials, short tutorials, and code samples. Conceptual introductions to a language or framework are wonderful, and if you have the time and expertise, by all means, write them. But programmers check online language and library references hundreds and hundreds of times. They only read *Programming Python* once. If you have to prioritize, go with the former. Most modern designers fall into the developer group, as well.

Knowing which group an audience falls into gives you a reasonable starting point, but you still need to home in on actual use cases in order to focus the documentation. A single inaccurate assumption about your audience can lead to a mountain of discarded work. Take the time to truly understand what your audience wants to accomplish through conversations, meetings, interviews, analytics—anything at your disposal.

Basic Functional Documentation

Now that we have knowledge and an audience, it's finally time to write. Basic functional documentation (BFD) is exactly what it sounds like: a big... fine deal. At a *minimum*, product documentation should answer the following questions:

- What is this product? Why would anyone want it?

These two questions are incredibly challenging for the average technical writer, a group of people much more focused on *how* than *what* or *why*. I've struggled with them many times. But if you can't answer these questions, you need to go back to the research phase, because you don't understand your audience at all. The answer to this question shouldn't come from a marketing department. Instead, it should be an honest, buzzword-free appraisal of capabilities and use cases.

- How does this product fit into a broader ecosystem, if at all? Does it have any dependencies?

  The answer to this question can be a single sentence, an architecture diagram, or a complex discussion of the various components necessary to create a coherent product stack. Web frameworks are perfect examples, because it's so rare to use one in isolation. They're typically paired with several complementary products, and developers need to know which ones prior to installing.

- Where can I acquire this product? If there are multiple distribution packages, which should I choose and why?

  One of the nastiest things you can do to readers is convince them that your product is useful and desirable and then direct them to a download page with fourteen different files and no guidance on which are appropriate for which users. Some people

can distinguish between `.zip` and `.tar.gz`, but many cannot.

- How do I install the product? What are the basic configuration options, if any?

  For consumer applications, the install process should be so streamlined that this section barely needs to exist. For server applications, it's often the longest, most critical part of the documentation.

- What does a simple, start to finish operation look like?

  This section can be anything from a pictorial walkthrough to a functional code sample—to be clear, not just a couple snippets—but whatever it is, it should take the user from *nothing* to *something*. It should be meaty enough to convince readers that they've learned something.

And that's it. The good news is that the principles of BFD are as applicable to vacuum cleaners as they are to databases. I created the following example for the

overpriced glass cone that I use to make my morning coffee:

> Manual drippers give you total control over how you brew your coffee. You can customize the temperature of the water, the rate at which you add water, and the sheer amount of water to arrive at your definition of a perfect cup. If you aren't picky and just want a cup of decent coffee in the morning, a conventional machine is a better option. This method does *not* produce espresso.

> Because of their unique shapes, drippers often require specialized filters. You also need a gooseneck kettle in order to more precisely control the flow of water. Some electric kettles have built-in thermometers. If yours does not, use an instant-read thermometer. A digital scale lets you measure the amount of coffee and water you use in the brewing process, which helps with day-to-day consistency. Finally, you need a burr grinder to reduce your *whole beans* to a nice, uniform grind. The total cost of this setup ranges from $150 - 500.

You can buy all of these products from Amazon. A manual burr grinder will save you a lot of money compared to an electric model, at the cost of your time and muscles. Avoid cheap blade grinders, as their inexact grinds ruin good coffee. Most other components are of comparable price and quality.

Once you have everything, adjust your grinder's settings until you arrive at a grind that is coarser than espresso, but much finer than French press. Vary this setting over time until you arrive at your favorite grind, and then *make note of it*. You can do the same thing with your water temperature. 200 °F is a good starting point.

To brew a cup of coffee...[3]

In very little space, I answered each BFD question. BFD gives you a baseline level of content for which to strive, but it also has the happy side effect of keeping you focused. Because the above example documentation is for a glass cone, I only provided the level of detail required to "operate" that cone. I didn't delve into *how* to adjust a

burr grinder or electric kettle, just that you should arrive at semi-fine coffee grounds and 200 °F water. Links to grinder and kettle documentation (if any exist) would be great additions.

> **Tip**
>
> Whenever possible, don't write from memory. Verify content as you write it. If the documentation involves shell commands or code snippets, copy and paste them after validating that they work.

Style

I hate the overbearing nature and rigidity of comprehensive style guides, but I'd be remiss if I didn't at least mention style. Don't worry, I'll keep it quick:

- Consistency is king. You sound unreliable if you use "Postgres" at the beginning of a sentence, "postgres" in the middle of a sentence, and "PostgreSQL" in one or

two random locations. Abbreviations are fine, but use them consistently, ideally after introducing them:

> PostgreSQL (Postgres) is a popular open-source database.

Likewise, if you call something a dialog in one document, don't call it a pop-up in another.

- Bias towards including headers, tables, lists, diagrams, and images. These additions make your writing more approachable and simpler to scan than paragraph after paragraph of prose. For example, almost all descriptions of how to do something should include an ordered list. In my coffee example, would an unordered list (or a table) have improved the second paragraph?

- Use inline styles to offset important text. Typically, technical writers use bold for user interface elements ("Click **Save**."); italics for emphasis ("You *must* save your work before shutting down."); and monospace

for file paths, terminal commands, and code ("Run `./bin/script.sh`.").

- For the sake of clarity, try not to verbify obscure nouns. An extreme example: "Grep it." vs. "Use grep to search the file."

- Copy edit, but don't obsess over it. A stray comma or dangling preposition isn't the end of your credibility. You can probably find a typo in this book. More than one, actually.

Catalog the Diff

One of the most important functions of a technical writer is to record changes to a product. Good change logs convince people to upgrade, inspire confidence in the direction of a product, and help developers take advantage of new features. What's new? What's different? What was removed? How do I upgrade? Any "gotchas" to be aware of? If a feature is simple, just noting that it *exists* is often better than adding new documentation on how to use it;

people can figure out how to use simple features without the help of documentation, but not if they don't know to look for them. More than anything else, change logs should be terse, minimalist, and eminently scannable.

Marketing departments often use these documents to write their own, hyped-up copy for advertisements or blog posts, but your own appraisal of differences should use a more neutral tone. In six months, the features described in the change log will no longer be so novel, but they *will* still be online. Readers shouldn't have to parse a corny sales pitch about the incredible potential of a new feature in order to learn that push notifications arrived in version 1.8.2.

Build a Website

I mentioned earlier that you should build and host a website, not distribute PDFs, but it bears repeating. Even the best documentation, like software, eventually goes out of date. PDFs get downloaded onto hard drives and then sit there like day-old bagels, growing more and more

stale until they're actively harmful. You can never update them. Even if someone downloads updated versions, every modern web browser saves the new files as `Admin_Guide (1).pdf` rather than overwriting the old files. The whole situation gives me the chills. Hosting your content on a website gives you the power to fix inaccuracies almost instantly *and* keep your content in sync with the latest software release.

Shipping an HTML help system with the software itself is barely better than PDFs. OK, the documentation doesn't go out of date, because it ships alongside its associated software, but you're still unable to fix potentially critical issues until the next software version ships. And that's *if* you can convince people to upgrade. If you must ship a help system, a minimalist user guide typically suffices.[4] If you're contractually obligated to provide more... well, I recommend following the letter of the law, not the spirit.

Help Others Write

However you decide to write and distribute your

documentation, you should do it in a way that encourages others to contribute. The reality of the profession is that even a large team of writers cannot possibly know everything worth knowing about an application, and most companies do not have a large technical writing team. The open-source software movement, mod scene in PC games, and birth of a million obscure wikis have proven that people will happily share their expertise and passion if an easy, hospitable way of doing so exists. Strangers will do it for free, and coworkers will do it even if it isn't in their job descriptions. I've had colleagues argue this point with me, but no, people really will contribute if you just *let them*.

Even if you only have 500 unique visitors to your site, a 1% contribution rate works out to five extra people helping with the documentation. I'll take that help. You likely have more readers and a higher contribution rate, so don't dismiss this idea as something that only applies to larger projects. It applies to all projects.

Ultimately, the job of a technical writer is to ensure the quality of the documentation. Hopefully you create a lot

of it yourself[5], but whether you write it or not is largely irrelevant to the end product. Quality is relevant. With quality as the objective, the contribution system should not be so open that people can just insert half-baked additions. A review process must exist.

Admittedly, the easiest way to help strangers contribute is to just create a wiki and be done with it. Wikis have some real appeal, but I don't believe they're the best option in most cases. We'll discuss why later.

Publish Frequently

Publishing should not be a special event. It should not be challenging. It shouldn't require anything more than a quick sanity check and a glance at the build log. Something about your process is broken if you need more than a minute to verify that content is ready to post to a production website.

Many ways of catching problems exist. For large, complex help systems, you can use a continuous integration system

to run a series of build tests after every commit or merge. This method is likely overkill for most help systems, where the testing should occur manually during the review process. Reviewing contributions to the documentation is more complex than just asking, "Is this content accurate? Is the writing well-organized?" It also means *building* the finished product and ensuring that the change has not introduced any new errors or warnings.

Even simple automation is effective, like a Cron job that checks out and builds the documentation every hour, writes the build log, and posts the resultant content to an internal staging server. Unlike software development, the process doesn't need to be especially elegant or sophisticated, just simple and functional.

---

1. Regarding creative writing: entertaining people is useful, too.↩

2. Not always the best assumption if your writing appears online.↩

3. I'm not actually going to cover the process, because for a morning ritual, it's embarrassingly complicated.↩

4. One that isn't detailed enough to contain any critical bugs.↩

5. Your job is in jeopardy if you don't.↩

Specifics

Now that we've discussed the basic things technical writers should do, we can delve into the nitty-gritty of how to do them.

Use Lightweight Markup

Someone once said, "XML is like violence: if it doesn't solve your problem, you aren't using enough of it." But whoever said this was definitely a) a developer and b) not writing XML by hand.

Because writing XML by hand is crazy, in the same way that writing any significant web application in JavaScript is crazy. Application developers should write TypeScript, Dart, CoffeeScript, or *practically anything else* and then compile to JavaScript. Likewise, if XML is a part of your publishing pipeline, you should write lightweight markup and then build to XML. The entire *point* of lightweight

markup is to make it easier to produce well-formed XML, and we need XML in order to build websites.

Why is lightweight markup so superior? Consider the following AsciiDoc:

```
= My Title

This introductory paragraph contains a link to
https://www.google.com[Google].

[source,ruby]
puts "Here is some code."

== My Section

* List item
* Another list item
```

Here is the corresponding DocBook, an XML-based markup language:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE  article  PUBLIC  "-//OASIS//DTD
DocBook XML V4.5//EN"
"http://www.oasis-open.org/docbook/
xml/4.5/docbookx.dtd">
<?asciidoc-toc?>
<?asciidoc-numbered?>
<article lang="en">
<articleinfo>
    <title>My Title</title>
</articleinfo>
<simpara>This    introductory    paragraph
contains a link to
<ulink url="https://www.google.com">
Google</ulink>.</simpara>
<programlisting                language="ruby"
linenumbering="unnumbered">
    puts "Here is some code."
</programlisting>
<section id="_my_section">
<title>My Section</title>
<itemizedlist>
```

```
<listitem>
<simpara>
List item
</simpara>
</listitem>
<listitem>
<simpara>
Another list item
</simpara>
</listitem>
</itemizedlist>
</section>
</article>
```

The AsciiDoc content is human-readable in raw form, straightforward to learn, and weighs in at 179 characters. The DocBook content is challenging to parse even with syntax highlighting, requires specialized knowledge of odd tags, and contains 723 characters. That's four times the number of characters for the *exact same content*.

As we discussed earlier, one of the tenets of modern technical writing is that everyone is a contributor. Storing

content directly in XML-based languages like XHTML, DocBook, and DITA dramatically reduces people's ability to contribute.

What about specialized editors? MadCap Flare and Adobe FrameMaker, two popular authoring applications, provide you with WYSIWYG editors for the rapid creation of well-formed XML. But they cost $1,448 and $999[1], respectively, and are only available for Microsoft Windows. Rather than being mere deterrents (like writing in XML), specialized applications actually *prevent* people from contributing. Amazing text editors are available on every operating system, mostly for free, and writers can use whichever they like. Popular editors include:

- Atom
- Sublime Text
- Notepad++
- TextWrangler
- gedit
- Vim
- Emacs

Note

The steep learning curve on Vim and Emacs is such that I don't recommend them for most people, but their power and flexibility are undeniable.

Microsoft Word is a wonderful choice for creating résumés and a horrible choice for creating documentation. Its lone purpose in this world—one that, again, it really does perform admirably—is to create short, attractive PDFs that can be consumed and discarded. Documentation with any sort of lifespan needs to be kept in version control, which Word's DOCX file format (a compressed collection of XML files) actively opposes. Documentation should live online, and Word's abysmal HTML export is totally unsuitable for creating websites. You have to style content in Word as you write it, rather than taking advantage of the natural separation of content and style, of HTML and CSS. Even though most companies provide licenses to their employees, Word still costs money and is only available for Windows and macOS. For documentation, lightweight markup is free and superior in every meaningful way.

Plenty of lightweight markup languages exist, but only three are really worth discussing: Markdown, reStructuredText, and AsciiDoc.

Markdown

Markdown is simultaneously incredible and infuriating, wonderful and maddening. It's the most widely used lightweight markup language in the world[2] and has the cleanest syntax, but it also has a limited set of features and no defined standard. These deficiencies have led to a couple dozen "flavors" of Markdown, including MultiMarkdown, Markdown Extra, GitHub Flavored Markdown, and a recent standardization effort called CommonMark. Each flavor adds features, some of which are implemented across multiple flavors with inconsistent syntax. The more maddening part, though, is that almost every Markdown parser (and many exist) processes whitespace *just a little bit* differently. Nested lists, for example, might have odd spacing in one flavor and not in another.

Using only "vanilla" Markdown syntax allows for broad compatibility, but you miss out on features like tables and "fenced" code blocks. Using a flavor of Markdown means you might have to rework your source files if you ever want to switch to a different flavor, but the level of effort in such a switch would likely be low. GitHub Flavored Markdown is a popular and fine choice for simple web-based help systems. Here's what it looks like:

```
# My Title

This introductory paragraph contains a
link to
[Google](https://www.google.com).

```ruby
puts "Here is some code."
```

My Section
```

```
- List item
- Another list item
```

Markdown's popularity means that many specialized text editors exist for it. The best are:

- MarkdownPad (Windows)
- iA Writer (macOS)
- ReText (Linux)

reStructuredText

reStructuredText (RST) comes from the Python community. Unlike Markdown, it has an actual, standardized implementation. It's also feature-rich, supporting tables[3], footnotes, and a wide variety of directives for code and other content blocks. With this wealth of features comes a syntax that has more than a few rough edges. Whereas you can learn Markdown in minutes, learning RST takes an hour or two. This might not seem like a big deal, but when you're trying

to empower contributors, the learning curve can be an unfortunate barrier. This book is written in RST.

The main appeal of reStructuredText is that it is the source language for one of the best documentation generators in the world: Sphinx. We'll discuss Sphinx later. Here's what RST looks like:

```
My Title
========

This introductory paragraph contains a
link to `Google
<https://www.google.com>`_.

.. code:: ruby

   puts "Here is some code."

My Section
----------
```

```
- List item
- Another list item
```

AsciiDoc

AsciiDoc is popular in the Linux community and is the language with which I have the least familiarity. Quite a few O'Reilly technology books are written in AsciiDoc.

Summary

AsciiDoc is semantically equivalent to DocBook and is thus a straightforward, obvious improvement to existing DocBook toolchains. This equivalence likely makes it the best choice for creating books, especially ones that require complex formatting. RST is a decent all-purpose language, but its lack of popularity means that it won't evolve at the same pace as Markdown. I am quite confident that the next great documentation site generator will use a flavor of Markdown as its source language, so choosing Markdown gives you a certain measure of future-proofing. At time

of writing, however, Markdown is missing many useful features compared to AsciiDoc and reStructuredText.

Online editors with live previews exist for Markdown, restructuredText, and AsciiDoc. These editors are helpful for testing out features, learning language syntax, and deciding which you prefer:

- dillinger.io (Markdown)
- rst.ninjs.org (reStructuredText)
- asciidoclive.com (AsciiDoc)

LaTeX

LaTeX is a powerful markup language with complex syntax that I purposely didn't mention until now. If you need it, you already know that you need it. If you don't need it (and few outside of academia do), its only purpose is as a stop in your publishing pipeline on the road to a PDF. More on the pipeline later.

Use Distributed Version Control

All sorts of people, most of them much smarter than me, have extolled the virtues of distributed version control systems (DVCS) like Git and Mercurial over centralized systems. The basics are that DVCS have better performance, allow for offline work, and are superior for concurrent work on the same file. For technical writers, the most important reason to use DVCS is that developers prefer them.

Because version control systems are designed for software development, each and every one of them is overkill for the typical documentation workflow. For example, you'll probably never need `git reflog`. So why worry about which system you use, when they can all do the job? Because when you select Perforce, Subversion, or CVS, you are basically confessing to potential contributors that you can't be bothered to use modern tools.

Atlassian Bitbucket and Stash are both excellent web-based interfaces for managing remote repositories. GitHub and GitHub Enterprise are also great options. While the default user interface, git-gui, is decent, I prefer Atlassian SourceTree and GitHub Desktop. The terminal is always there if you need advanced functionality. Many people even prefer it for basic, everyday operations.

If you have the opportunity to store your documentation in the same repository as its corresponding product source code, strongly consider doing so. The approach has some real appeal:

- Documentation and code branches stay in sync.
- Developers are more likely to contribute if they don't have to clone a separate repository.

Of course, this approach also has some cons:

- If a repository is large (e.g. 2 GB), checking out the entire repository just to access the 40 KB `./docs` directory can dramatically slow down documentation builds.

- Onerous commit hooks and pull request submission policies, designed to keep a code base stable, can make even simple documentation changes a chore.
- If your product is composed of code from many repositories, a single documentation repository offers simpler builds and a more cohesive writing experience.

Wherever you store your documentation, place a file named `README.md` in the root of the repository (or in `./docs`). This Markdown file should include:

- A quick summary of the product being documented
- Instructions on how to build the documentation locally
- Instructions on how to contribute

Don't Duplicate

Along with accurate driving directions and videos of frolicking puppies, links[4] are one of the most wonderful things the internet has given us. The most popular sites

on the internet are devoted, in fact, to the sharing of links between friends and strangers. Making proper use of links lets us attain a sort of Holy Grail in technical writing: single sourcing.

The core idea here is that you should break your writing into bite-sized pieces, often called *topics*, none of which overlap in content. Doing so means that if things change —and things always change—you only have to update the documentation in one place. Many topics are short, whereas others are quite long. What topics should be is relatively *complete* in their discussion of a particular subject. Unlike books, where you can assume in chapter seven that the reader remembers chapter six, topics assume very little knowledge of other topics and include plenty of links.

Unfortunately, the nature of language and learning means that some amount of duplication is always going to occur. But for the most part, linking liberally, arranging topics in a sensible way, and trying to adhere to a single source model is a good idea. reStructuredText and AsciiDoc even

allow you to embed files within other files, which further aids reusability.

The worst duplication sin of all is to store similar copies of the same documentation in version control. The root cause here is typically the desire to have one version for coworkers and another for customers, and it inevitably leads to a maintenance nightmare. Instead of multiple copies, the better approach is to use a feature sometimes called "conditional text." Conditional text lets you selectively include or exclude files (or portions of files) in the final output. AsciiDoc supports conditional text under the name "conditional inclusion macros." reStructuredText supports it through Sphinx with the `..only::` directive. Unfortunately, this feature is not present in Markdown. Attempts to add it can be charitably described as "hacky," so if you rely on conditional text, you might want to select another language.

Make Static Websites

I have perhaps an irrational bias towards static websites.

I love them. I love their speed, simplicity, portability, and security. You can host static websites practically anywhere, including Amazon S3 and GitHub Pages. They have no server-side application dependencies, no databases, and nothing to install, so migrating the entire site is as easy as moving a directory. Because all it has to do is serve pages (rather than generate them dynamically), the server needs very few hardware resources. You can test static websites on your local computer without installing anything and compress and ship them with software applications (although you shouldn't unless you have to). There's no WordPress instance for someone to hack or comments section for bots to spam. Nothing ever crashes, because there's nothing *to* crash.[5]

Sure, you could manually create a simple static website, but to do anything complex, you'll want to use a generator. The basics are that you provide a static site generator with content (lightweight markup) and a theme (templated HTML and CSS), and it processes everything into a working website. To update the site, just modify the content and process everything again.

Many static site generators exist, far too many to list. In fact, I'm sure some new, amazing static site generator is in development right now, and I already feel like a jerk for not mentioning it. But if I had to suggest a few:

- Sphinx
- Jekyll
- Hugo
- AsciiDoc
- Middleman
- Metalsmith
- MkDocs
- Hexo

Any one of these tools can create a beautiful, functional documentation website. Jekyll is the most popular. That said, I favor Sphinx for two major reasons:

- It was specifically built for documentation, rather than blogs or marketing sites, so it has a robust system for generating nested navigation menus.
- It features JavaScript-based search out of the box. Every other tool in that list (except MkDocs) does

not have out of the box search and relies on Google Custom Search, Swiftype, Algolia, Lunr, or some other search solution. Adding a search solution to a static site often makes it a whole lot less portable, so Sphinx has the right idea here.

Whichever tool you select, take the time to customize the theme. Focus on navigation and approachability. Find a designer and ask for help with colors, typefaces, font sizes, page width, and spacing. This customization is your chance to differentiate your content from the thousands of ugly, disorganized sites of the world, so don't just use the default theme. If you don't have the skills to do this customization yourself, you might need to hire someone. It's that important.

And make sure you have a search solution.

If making a static website sounds like a lot of work, well, it isn't, but an even simpler solution exists: GitHub wikis. The secret with GitHub wikis is that they're really just a collection of lightweight markup files that you can edit using your favorite text editor *or* through a web browser.

GitHub wikis support Markdown, reStructuredText, AsciiDoc, and some other languages that you shouldn't use.

You don't get a lot of flexibility in how the resulting pages are organized—just an alphabetical list—but you can make a table of contents on the home page and have a highly functional site with very little effort. You lose the ability to make your website beautiful and unique, and GitHub wikis do not have search. But those might be worthwhile trade-offs for simple sites. This way, at least you're storing lightweight markup files in a DVCS. You can always create a better site later using those same files.

Rsync

Static site generators typically regenerate the entire site for even simple changes. This behavior helps ensure that the changes didn't have any unforeseen consequences and that the site will work properly. However, it also means that copying your new site onto an older version of the site (via FTP or SCP) can result in removed or renamed

files not being overwritten and persisting on the server. The easy solution is to delete the old site from the server before copying over the new site, but a tool like Rsync can save you the hassle. Rsync transfers only the files that need to be added or updated, and it deletes stale files. Rsync is often used to keep staging and production servers in, well, sync.

If you use Windows, you probably need Cygwin to take advantage of Rsync. On macOS and Linux, installation and usage is more straightforward.

> **Note**
>
> If you use Amazon S3, the AWS Command Line Interface has a sync function that works similarly to Rsync.

A slightly more complicated alternative is to configure your production server as a remote repository and push the finished site to it, which accomplishes the same thing: new files get added, existing files get updated

(if necessary), and stale files get deleted. Unsurprisingly, GitHub Pages uses this method.

Metrics

If you aren't keeping an eye on documentation metrics, you're making a huge mistake. User research is wonderful, but knowing exactly which pages are most popular, your site's bounce rate, and common behavioral flows are all invaluable. Create a Google Analytics account, add the provided tracking code to your static site theme, and check the numbers regularly. I'm sure other tools exist, but Google Analytics is simple, free, and the industry standard. I've never used anything else.

Metrics rarely serve as an obvious pointer to the correct course of action. Rather, they serve as a valuable check against your thoughts and intuition. If you create what you think is an amazing bit of documentation, and sure enough, it's one of the most popular pages on the site, perfect! If it's not very popular, maybe no one can find it, or maybe it's not as useful as you thought. Deeper

investigation might reveal a major flaw, a minor tweak, or a reasonable explanation that requires no change at all. But not having metrics and using only your own criteria for success can be disastrous.

Note

Customers often frown upon a native application "phoning home"—especially to a third-party like Google—with usage metrics. Gathering these metrics from a native application requires costly development work, which is yet another reason why *only* hosting your documentation on a website is the better approach.

In the software industry, you often hear the questions, "What does good look like?" and "How do you measure success?" For a documentation website, any answer should include some mix of metrics, bug numbers[6], reader feedback, alignment with corporate strategy (if applicable), and finally, personal intuition. If you have access to metrics from other teams, comparative analysis often yields the most valuable insights. How does the

number of documentation bugs compare to the number of calls to customer support? Do the most-used features of the product correlate with the most-visited pages of the documentation?

I'm sure other ways of measuring success exist. I'm probably just ignorant to them. Detecting the difference between good and great documentation is an incredibly hard, unsolved problem, but that doesn't mean we can give up trying to solve it. Consider the following statements from two hypothetical technical writers:

- "In my professional opinion, the content is clear, concise, correct, and complete. The language is professional, conforms to our style guide, and projects a strong brand. Some of the tables were too wide for print, so we now enforce a two-column limit on all tables. Overall, I'm happy with the quality of the documentation."
- "The application logs show that the product only has 1,300 users, yet the documentation received 2,400 page views last month. In that same timespan, readers

> reported six inaccuracies, all of which I resolved within 72 hours. The five most popular search terms return the pages I would expect, and the design team recently helped me optimize the header margins for readability. Overall, I'm happy with the quality of the documentation."

I sincerely hope that you find the second argument more compelling. In any field, opinions become more credible when you attach quantitative metrics to them. Documentation is no different.

Script Your Complexity Away

If you go against my advice and decide to create and distribute PDFs, you might have to create a so-called publishing pipeline. Publishing pipelines are the steps through which lightweight markup files become print-ready documents. All lightweight markup languages build to HTML, but the journey to PDF often requires an extra step—hence the pipeline. You probably have to build to LaTeX first, then LaTeX to PDF. LaTeX installers are

massive 1.8 – 2.5 GB downloads, so if you use a set of machines to build your documentation, consider using just one as the PDF machine.

Pandoc is a marvelous tool for converting between markup formats. It calls itself the Swiss Army knife of markup converters and can convert to and from a huge number of formats. Unfortunately, these conversions are rarely perfect. If one day you decide that you'd rather write in AsciiDoc instead of Markdown, expect to perform some manual cleanup after running the conversion script.

For simplicity, Pandoc conceals the LaTeX step, which makes the creation of PDFs pretty straightforward:

```
pandoc your-document.md -f markdown_github
-o your-document.pdf
```

Note

You can even convert to the Microsoft Word document format by specifying `.docx` instead of `.pdf`.

Unfortunately, this method only works if all of your content is in a single file—unlikely. More likely is that you'll need to do something like this:

```
pandoc title.txt 1.md 2.md 3.md 4.md -f
markdown_github -o your-document.pdf
```

> **Note**
>
> If you want to customize the way the PDF looks, doing so requires a new LaTeX template and is extremely complex. If customization is important, consider using a tool like wkhtmltopdf or PhantomJS so that you can use CSS instead of LaTeX for styling. Whatever you do, don't compromise the look of your HTML in order to have more attractive PDFs. Everything you do should be geared towards more useful, accessible websites.

You still want HTML, though, so you should include a second build command in the same script:

```
jekyll build --source source --destination
build/html
```

You might need to check out several remote repositories before building, rename or move files, or run a supplemental script. The point is that your lightweight markup files should be as simple as possible. Don't require people to conform to silly guidelines in their writing (or worse, write in XML) just so your build script is a little easier to understand. If any part of the technical writing process should be complicated, this is it. You might need to read a book on Windows PowerShell, Bash, Rake, or Grunt.

1. Lightweight markup goes in (simple).
2. A script checks out a Git repository, applies a custom theme, compiles LESS files to CSS, converts Markdown files to AsciiDoc, and finally runs the build commands (complex).
3. A static website comes out (simple).

Commit the build script to the documentation repository and use `README.md` to explain what it does and which dependencies it has. In this case, I might list Pandoc

and Jekyll as dependencies and provide links to their installation documentation.

If you use Rsync, you probably don't want to include it in your build script. Making a separate "publish" script that calls Rsync gives you the opportunity to manually verify that the help system opens and functions properly before you send it to a remote server. This final, ten-second sanity check is worthwhile.

The Legal Problem

Unfortunately, doing your job well might actually *cause* lawsuits. I've heard unsubstantiated rumors of patent trolls using publicly-available technical documentation as evidence of infringement. In this case, the open, forward-thinking spirit of enabling users to contribute to the documentation can be financially disastrous for your company. You might need to keep your source files in an internal repository, rather than GitHub. Further, you might need to protect your documentation website behind a customer login page. You should still use lightweight

markup and distributed version control so that coworkers can contribute, but the general public might not be able to.

In short, assholes ruin everything.

Localization

Localization, the process of translating documentation to other languages, is a nightmare. If you ever *think* you need to do it, interface with management and perform a careful cost-benefit analysis, because the process is expensive, time-consuming, error-prone, and tedious. Once you've arrived at what you believe is an accurate estimate for company costs, triple it. Now you have a realistic estimate.

If you try to keep all translations of the documentation in sync at all times, you can't publish very often, which leads to lower quality documentation. To fix a bug in your native tongue, you update the content, build, and publish. To fix a bug in a translated copy of the documentation, you update the content, manually send it to a translation company,

wait days or weeks to receive the updated content, build, and publish.

Because publishing in a single language is so simple, you can work with increased velocity. You can improve large chunks of text, reorganize pages, fix awkward wordings, and generally treat everything as a working draft. Writing content for translation means meticulous review of new content, because any inaccuracy has to go through costly revisions. It means rarely refactoring old content because of the sheer expense involved in translating it into eight different languages again.[7] It means delaying software releases because it's Carnival in Brazil[8] and the Portuguese translations won't be ready for another week.

Lest you doubt me, consider this common open source localization workflow:

1. Write a script that calls gettext, a translation tool, on each of your lightweight markup files. This operation produces a collection of POT files. What POT stands for doesn't really matter. Conceptually, POT files are just line by line splits of your source files into two

strings: the original one, and an empty one for the translator to fill in. They look like this:

```
msgid "My name is Andrew."
msgstr ""

msgid "Another paragraph."
msgstr ""
```

2. Still using gettext, generate sets of PO files from the POT files, one set per language. Send the PO files to the translation company, who will insert translated content into the empty strings. PO files look nearly identical to POT files:

```
msgid "My name is Andrew."
msgstr "Je m'appelle Andrew."

msgid "Another paragraph."
msgstr "Un autre paragraphe."
```

3. When the translation company returns the PO files, commit them to version control.

4. Then you—sadly, this is not a joke—convert the PO files to MO files, again with gettext. MO files are binary compilations of PO files. Because PO files can be rather large, you compile them to MO files to improve the speed at which a machine can process them. You don't need to commit the MO files to version control.

5. Build and publish the translated help system from the MO files.

6. Repeat these steps each time you need to update the translated help system. Even though it might *feel* as if you're overwriting the translated PO files with new, blank ones, the PO files will retain their existing translations. The translation company can then fill in the gaps created by new or modified content:

```
msgid "My name is Andrew."
msgstr "Je m'appelle Andrew."

msgid "A new section."
msgstr ""
```

```
msgid "Another paragraph."
msgstr "Un autre paragraphe."
```

You don't have to follow this workflow.[2] In fact, your static site generator might not even support building from MO files. I've only provided it as an example of the complexities involved in the localization process. I'm not being melodramatic; it really is a nightmare.

For a method that uses common tools (i.e. Git), you might try:

1. Send your lightweight markup files to the translation company as-is. Tag the latest commit in your repository to mark this point in time.
2. When the translated files return, run a linter on the files, because invariably the translation company will have messed up the whitespace on your documentation. Then commit the translated files to version control.
3. Build and publish the translated help system.

4. When you want to update the translated help system, make a new branch and use git rebase to squash all work between now and your tag into a single commit. Then send the diff to the translation company, along with the latest lightweight markup files for both languages.
5. Delete the temporary branch and mark the latest commit in the repository with a new tag.

Remember, you have to repeat this process *for each language*. Some specialized software applications purport to simplify the translation process—and to be fair, I'm sure they do help a bit—but until computers can accurately translate between languages, localization will remain messy and expensive.

Contrived Workflow

This section provides an end-to-end, functional example that conforms to all of my guidelines, with a few detours to make it more realistic.

1. You get hired at a software company. Congratulations!

2. Three weeks (or months) later, you are wise and knowledgeable about the company's products and users. You also have your health insurance all sorted out.

3. You create four different files in GitHub Flavored Markdown: `about.md`, `tutorial.md`, `concepts.md`, and `process.md`.

4. You install Sphinx with `pip install sphinx`.

5. You create a directory structure and Sphinx configuration file (`conf.py`) by running `sphinx-quickstart`.

6. You try to build these Markdown files into a help system with Sphinx, but Sphinx uses reStructuredText, not Markdown. You curse several times.

7. You use Pandoc to convert the files from Markdown to reStructuredText, but after examining the resultant files, you decide you hate RST and would rather continue to write in Markdown.

8. No problem, you create a script that converts all of your Markdown files to RST and *then* builds with Sphinx. This way, you can write in Markdown, but still take advantage of Sphinx. The script looks something like this:

```
find . -name \*.md -type f -exec pandoc
-f markdown_github -t
   rst -o {}.rst {} \;
sphinx-build -b html ./source ./build
```

9. The script works, but you still need to build a TOC tree (table of contents) for Sphinx. You create a file called `index.rst` and embed the converted content from `about.md` within that file so that you have to write as little RST as possible. `index.rst` looks like this:

```
.. toctree::
   :maxdepth: 1

   tutorial
   concepts
   process

.. include:: about.rst
```

10. Everything looks pretty good, so you create a new repository on GitHub, clone the empty repository to your computer, and add your documentation to it.

11. When you commit your files, you notice that you need to edit `.gitignore` to ignore `build` (the built help system) and `*.rst` (the converted RST files) *except* `index.rst`. `.gitignore` looks like this:

```
build/
source/*.rst
!source/index.rst
```

12. You realize that anyone who views the repository on GitHub will have no idea what it is, how to build it, or how to contribute to it, so you add `README.md` to the repository root.

13. You decide that the default Sphinx theme stinks, so you download a new one and point `conf.py` at it. Then you decide that one stinks, too, and take the time to learn how to build your own.

14. Once you're happy with how everything looks, you harass your IT department until they give you the subdomain docs.yourcompany.com. When they ask what sort of server you need and what you need installed, you tell them that you built a static website and just need a directory for file transfers. They're super impressed.[10]

15. You transfer the finished help system to docs.yourcompany.com and are officially a published writer.

16. Three days later, some developers approach you about a change to the documentation. You tell them you're busy and that they should submit a pull request. They too are super impressed.[11]

17. Promotions and financial windfalls ensue.

What About Wikis

Admittedly, a wiki is simpler than the system I just described. Anyone can figure out how to edit a wiki, which is a killer feature until it isn't. Wikis have huge benefits and huge drawbacks. They certainly have a place in technical writing, but you should think carefully before selecting one.

Users don't need to install any applications, clone any repositories, or learn any strange workflows to contribute to a wiki. They can spot an issue, click **Edit**, and fix it within ten seconds. The process is remarkable. Also, every

major wiki has out of the box search, a huge improvement over most static site generators.

One major problem, however, is that wikis are web applications. If you decide to create and host one, they require maintenance over time. And if you've ever worked at a company with an internal wiki, you know how often they crash or slow down. Yes, Wikipedia has proven that wikis can scale to a huge number of users with good reliability, but that scaling is the result of a ton of hard work from some very smart people, people whose job it is to manage the site. Personally, I want to spend the smallest possible percentage of my time managing a site. I'd rather test, research, write, and curate content. Static sites are indisputably simpler to manage.

To be blunt, writing on a wiki sucks. Mercifully, some wikis use lightweight markup, but it is often a mangled form that incorporates a subset of HTML tags and barely qualifies as "lightweight." Others use a WYSIWYG editor, so the experience is like using Microsoft Word without all the convenient keyboard shortcuts. Everything in a web browser is so ephemeral, ready to be flushed at any time.

I've lost at least a few pages of work to connectivity issues and Firefox crashes. Writing lightweight markup in a text editor is faster, easier, and *safer*. GitHub wikis, which I mentioned earlier, are an exception to this rule (and several others in this section).

Wikis are fantastic for living documents that grow over time and never need to be marked as belonging to a particular software version. Many software as a service applications fit this description. Supporting multiple versions of your documentation—for example, version 3.1 and 3.2—is trivial with distributed version control. Doing the same on most wikis requires specialized plugins and some manual shuffling of content.

Version control systems like Git do an amazing job tracking changes over time. You can view the change log for individual files, easily "blame" others for problematic content, see related changes that have been lumped into commits, and merge disparate commits easily. With most wiki software, you can only view the change log for individual pages. Pull requests in distributed version control serve as a built-in review process for changes.

Because most wikis follow the "last person to edit wins" policy, you might not even notice a destructive change to the documentation. Wikipedia ameliorates this problem with a sophisticated anti-vandalism bot that uses machine learning to improve over time. Given my expected lifespan, I'm confident that I could never create such a bot. Your mileage may vary.

Ever gotten work done from a plane, train, park, or hipster coffee shop that refuses to offer wireless internet access? Not with a wiki, you haven't. With distributed version control and static site generators, you have everything you need to write, build, and review changes locally. When you have connectivity again, you can push your changes and update the website.

In short, for a wiki to make sense, your documentation should be uncontroversial and never need to be versioned. You also shouldn't mind writing in an inferior editor, only working online, and maintaining a piece of enterprise software.

---

1. As of September 2015.↩

2. Source: conjecture.↩

3. The default "grid tables" are horrifying, but CSV tables and list tables work well.↩

4. Please don't call them hyperlinks.↩

5. Aside from the server itself, anyway.↩

6. Number reported, severity of each reported bug, average time to resolve, etc.↩

7. Translation memory can only help so much.↩

8. Or Golden Week in Japan, etc.↩

9. And probably shouldn't.↩

10. To show you how impressed they are, IT builds you a Linux VM with 256 MB of RAM and 3 GB of disk space and refuses to give you superuser privileges.↩

11. But they still don't submit the pull request.

11. But they still don't submit the pull request.

The Grand Finale

Having finally finished this book, I'm a bit depressed—turns out everything I know about technical writing can be covered in just over 10,000 words.

The point, I guess, is that technical writing doesn't need to be complicated. Technical writers have just spent several decades convincing people that it should be. No part of the job is easy in execution, but all of it should sound easy in theory:

1. Learn everything about a subject.
2. Write down exactly what an audience needs to know and *no more*.
3. Make the content beautiful, discoverable, scannable, and searchable.
4. Consider everything a draft, and iterate relentlessly.
5. Make contribution simple.

I wish I had something more all-encompassing to say, something that would tie technical writing into storytelling or poetry or *life*. But no, technical writing is a job like any other, and I believe this book can help you do it a little bit better.

Acknowledgments

Even a relatively short and simple book like this one is a colossal task. Many thanks to my wife, family, and colleagues for their invaluable feedback and encouragement throughout the process.

I'd also like to thank all the readers who helped make this book such a success. If you enjoyed Modern Technical Writing, please consider leaving a review on Amazon.com.

Legal