Codewriting

Collaborative Documentation Ops for the Agile Age

Brian Dominick

Table of Contents

A Note to Readers & Contributors

There is no sense in postponing my confession: I have written a user manual for the role of technical writer. The first draft of that sentence originally contained a reference to *irony*, but in truth it is not at all ironic that technical writers ourselves require instruction. I've strung together a lot of prose in my lifetime, but I'm casting this book project in a format I hope technical writers and software developers will appreciate: lots of bulleted section headings, lists, sidebars, examples, code listings, and summaries.

But I'm also bestowing this book with some rants, because I have things to express. You are welcome to roll your eyes right past any objectionable passages.

A tech writer is not a tech journalist

Tech writers *document and instruct* a product on behalf of its *makers* (usually *as one of its makers*). By contrast, **tech journalists** *promote and praise* products on behalf of the journalists' *sponsors*, which are often those same products.

I don't want you to just read this book; *I want you to contribute to it*.

Collaborative Authoring

Codewriting is a living document being written on a public GitHub repository. You are welcome and encouraged to contribute by forking the repo, contributing edits, and issuing a pull request (PR). For detailed instructions, see Collaborative Authorship & Lean Publishing.

Be sure to add your name to the contributors list with your PR!

Writing to Learn

Every subject in which I have achieved true proficiency, I have written about. But I do not write as an expert, conveying my accumulated wisdom; former journalist that I am, I write to motivate and structure research efforts during the intermediate part of a learning project. If I knew everything I need to know to master tech writing and DocOps, I would not be writing this book. I'm hoping you'll at least follow along, if not chip in to teach me a thing or two.

"Intermediate" may best describe my proficiency at technical writing, and then only because I join the field with nearly 20 years in software development and even longer writing professionally. I won't release editions of this book that contain known falsehoods, but it may always contain a significant amount of vagueries and shortfalls. The field of technical writing is the only subject I handle that's more complicated, uncharted territory than the multi-component enterprise platform technology I document by day.

Engaged Learning

As a technical writer, I like to use examples. I also happen to learn best from examples. Most importantly, I learn best by following along.

For these reasons, a narrative structure will be threaded throughout the book, and you are encouraged to play along. All the tools you will need are introduced as they are called for. Every tool used in these instructions will be free and open source software. Examples in this book heavily favor AsciiDoc markup language, though daring participants are welcome to follow along in a markup language of their choosing. The particular language should not matter for most examples.

If you succeed with parallel examples, please considering contributing them.

Tech Writing Can Be Fun & Funny

I'm enjoying this book, but I have to admit, the Reference Guide I maintain as the primary responsibility of my day job is not always a page-turner (for reading or writing). I get enjoyment out of many of the non-writing parts of this job, including working with great developers and getting my hands dirty with their bleeding-edge software. Still, we all know tech writing can be the opposite of pleasure writing. Fussing over every word in an instruction step, trying to shorten code listing line lengths to fit the page, or just re-conveying the same banal concepts over and over; tech writing has its downsides.

Consider this: customers pay a lot for our product license, and the closest they'll ever come to a physical artifact of our product would be if they hit print on the PDF I maintain day in and day out. It has been conveyed to me by people of good repute that joking and cursing in technical documentation for a "serious" enterprise product is a bad move.

I think that's bullshit, but I propose a compromise.

I will bite my tongue on the F-bombs, S-storms, and D-words, but let's try to inject humor into this most dry of *crafts*. It may never rise to the level of an *art form*, but I'll take *craftsperson* over *clerk* as a job genus any day.

Creative Commons License

I'm not trying to get rich or famous off this book (if that was not already obvious, you might be in the wrong place; *try tech journalism*). You can take this text and do what you want with it, so long as you leave a crumb trail of credit as described in the copyright and licensing notice.

I would be honored if a free-thinking colleague ever decided to fork this, whether to contribute substantively and propose a re-merge, or to take this idea in another direction, even in their own voice — *so long as you freely share anything you share at all.*

See the ground rules in the `LICENSE.md` file.

Acknowledgements

My first shout out has to go to Eric Sammer, CTO at Rocana, who had the foresight to set me up with such a great documentation toolchain. Eric has given me room to experiment with the docs at Rocana, just as he and the Engineering organization have kept a constant flow of new challenges streaming across my desk. I did not take this job with any semblance of a notion that I was stumbling into a nascent (niché) field that so fully fuses writing and development. Since those are the only two consistent interests on my resumé, I suspect Eric saw more than I evidently did.

I also want to express appreciation for my manager, Arthur Foelsche. When your company is so small you're the only tech writer, it helps a lot to have somebody who's well paid to feign an interest in what you do all day. Truthfully: through countless hours of coding and conversation, Arthur has pushed and helped me to solve more than a few of the problems I'm truly excellent at raising. He also encouraged me to do this book and pursue the initiatives and projects that inspired it, as did everyone in Rocana's Engineering department.

Contributors

The following people made commits, either to the book or platform.

- **Your Name (GH: yourGHusername)**
  A one-line-or-so bio of You.

See their contributions in the repo, contribute come content yourself!

Reviewers

The following people reviewed and commented on portions of this living manuscript.

- **Your Name (GH: yourGHusername)**
  Your title and/or a one-line-or-so bio of You.

Foreword

This space reserved for special invite.

Introduction to Codewriting

This project is my attempt to bridge a gap I straddle between two worlds of writing: the world of words and the world of code. I have spent my career (over two decades) in and out of media and software, almost always mixing the two in some way, even when mixing them has not been my primary role. Whether organizing and coding a 4-year experiment in online journalism or developing websites for writers and media operations, I have been dealing with text and code nearly my entire adult life.

Today, as Technical Documentation Manager at a little "Big Data" IT ops tooling startup called Rocana, gracefully combining technology work with media production remains my biggest challenge.

In technical writing, we have to learn complicated products with a level of intimacy even some engineers find perverse. We also have to maintain more and more matter from an ever-increasing array of sources, operating as both scribe and librarian in service of highly demanding engineering, product, and support teams.

And then there are the end users. Here technical writers sometimes feel quite apart from developers. We aren't the ones who deliver the part of the product users love; we make the part they resort to in frustration. Good docs strive to surprise users, if simply by being good docs.

Who is Codewriting For?

Several audience roles are front of mind for *Codewriting*, but I hope thematic lessons will somehow reach beyond people who regularly consider themselves technical communicators. So much knowledge is poorly organized, and true collaborative, open-source writing remains unpopular despite its astounding advantages. I blame accessibility, and I believe we're on the verge of a revolution in the accessibility of this approach to documentation. I further think that revolution will extend to other forms of documentation: legal, legislative, medical, culinary, pedagogical, and beyond…).

Young people are technical enough to more-intuitively make the conceptual leaps necessary, and before long distributed source control will be a basic technical skill. While only the "geeks" among Millennials and GenXers readily grasp concepts such as markup and source control systems, these technologies will likely prove trivial to the next generation.

Likewise, young people are showing greater interest in collaboration. The efficiencies of highly manageable docs libraries will be popular among the entrepreneurs and community organizers of the world alike. Which is all to say, this book is for people who want to think about the future of documentation as it evolves. That's thinking big, in ways I'll follow up on much later in Beyond Technical Communications.

For now, my focus is on several roles typically found within formal software engineering teams.

Technical Writers

Obviously, I hope lots of current or aspiring technical writers will make use of this book. But it is

definitely not limited to people who already know about or work in a docs-as-code format. I have in mind people using DITA and other conventional documentation tools. And while some background playing with code is very helpful to get started, this book should not require more than basic familiarity with how software is made.

Keep Me Honest

If any concept or instruction assumes advanced knowledge, let me know. Please either file an Issue on GitHub or write me personally. My intention is to make this book accessible to people with no code writing background.

Subject Matter Experts

In software, SMEs are mainly developers/engineers. In any case, these are people who primarily write product source code but are also integral to documenting that code.

SMEs may also include product managers, customer support engineers, and anyone else with product knowledge who might be expected to contribute to user-facing documentation in the course of their work.

DevOps Engineers

These are hackers who integrate IT/automation/infrastructure work, often including QA and testing, with the day-to-day operations of developers. While perhaps not usually making major direct contributions to docs, DevOps pros can be documentation's best friends. The broad field gives insufficient attention to docs, in my observation, but having the right DevOps specialist devoting a few cycles to docs tooling and testing can make all the difference. So I hope more DevOps people will take note of new ways to integrate docs with source code, testing, building, and packaging.

DocOps Specialists

This refers to hackers committed to building and using better docs-as-code systems, meaning workflow and process as much as tooling. I don't know that this field exists yet, but I want to make it. There are lots of people who fit the label, as I see it, but we don't have an identity or space of our own. I think a lot of us are even more socially awkward than coders — don't think for a moment I'm not hoping this book earns me some cool-cred from legit programmers.

If you're already doing DocOps, I hope you will find some sense of community and camaraderie in these pages. If you are not yet doing DocOps but have an interest in hacking platforms *and* collaboratively writing great docs, maybe this book will provide a coherent framework for approaching this new field. Even if you're unsure about *collaboration* — maybe you are working alone or are unconvinced your SMEs will contribute directly — hopefully this book will help you think about and organize your docs.

What's in a Label?

Perhaps dangerously, I use some software-industry terms in ways that might get me in trouble if I'm not careful, and I need to review this manuscript for stylistic consistency. For the most part, here is the breakdown of who's who among coders in this book.

I generally use the terms *coders*, *engineers*, *developers*, and *programmers* all to mean people who write software source code for fun or profit.

When I'm speaking in relation to a project, I use *engineer* to mean someone on the core product team and *developer* to mean someone who may extend the product. So my immediate coworkers are "engineers", and third-party developers offering to modularize our product are "developers". This term includes no bias as to the skill set of either role, just their relationship to a product.

When I say *hacker*, I typically mean someone of pretty much any skill level whose main purpose is pulling together just enough code to get something done. I include myself in this category, and I wear the label humbly as I scavenge the corners of the Web for tools shared by my betters, hopefully learning to contribute back more, as I develop the nerve to do so.

In this book, just for the fun of it, I frequently call DocOps specialists *codewriters*, as we use code to build docs platforms, and we also write *in* code *about* code.

Finally, I'm adopting the word *documentarians* for anyone directly contributing to documentation, from any role. You're a documentarian if you are into making better docs, i.e., you're reading this book.

Objectives of Codewriting

After reading this book and engaging its exercises, you should be better able to

- **describe software** to users and **instruct** them in its use;
- **support engineers** in more directly communicating their product to its users;
- **establish systems** for collaborative documentation using the latest open source tools and platforms;
- **integrate documentation** into product development and testing… and into the product itself;
- **coordinate contributions** from devs and other SMEs in ways that complement rather than interfere with their **preferred workflows**;
- **resolve complicated documentation challenges** with an open (if often hackey) approach;
- **convert legacy material** to a future-compatible system; and
- **'codify' your technical writing** by thinking like a developer while sharing the product codebase.

Even if some of these lessons do not apply in your current setting, perhaps they will inspire some creative thinking, as they have for me. There is a lot more to this approach than I will be able to capture in this book, which is part of why you're invited to capture it here with me.

Docs: Neglected and Maligned No More?

Plenty of organizations have shelled out for big-money subscription products that a whole lot of writers love, and I do not intend to knock these tools, even as I will critique what I see as their limitations. I'm more focused on portability, extensibility, and open standards. I'll touch on this more in a moment (Open Source Centricity), but I find most of the advanced technical documentation tools are lacking in some or all of these areas.

My preferred tools are lacking in several areas as well, and that's another theme of this book, because we really do have power over this tooling. We don't need to wait for companies that hide their source code to grant us sparse updates. We have so many alternatives, and they're maturing quickly at this particular moment.

In conventional software operations, it's usually the same scenario; the product gets all the technical focus, and documentation is too often considered separate from the product. If you're a technical writer, you may not even be considered a contributor to the product; your work is merely passed along *with* the product. It's highly likely that your output does not come near the product it describes until packaging — if even then.

The onset of the *Dev*Ops mindset/movement has mainly helped docs indirectly, as many of their integration tools have docs applications, even if only as a side effect or afterthought. It's high time we take advantage of all of this tooling.

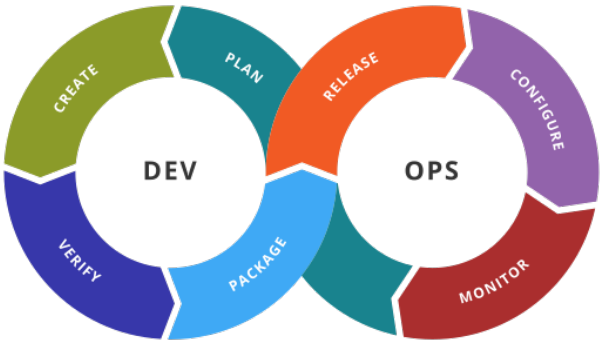

Figure 1. DevOps Toolchain by Kharnagy

The above diagram makes no mention of documentation, and I'm not complaining. This is an accurate reflection of the DevOps focus, and docs are not an integral part of that cycle. Where they are, they may be considered part of the *code* that this toolchain/process is acting on.

Documentation needs to be integrated with this process, or it will always be an afterthought, but that's not a DevOps job alone. If technical writers and documentation managers aren't going to step up and use advanced tools to integrate their work with that of engineers, how is DevOps supposed to see docs as an

integrated element? Right now, in plenty of shops, bright-eyed DevOps ninjas are pretending not to see the old-timey trainwreck that has become of the company's legacy documentation setup. They're not failing us; they're understandably averting their eyes.

Docs are as essential to product development and delivery as these other elements, or so we keep telling ourselves, product managers keep swearing by, and CTOs and engineers pay lip service to, sometimes. They're just often not as well integrated, which leads to them not being as current, complete, and accurate. Which gives product and engineering leadership actual reason to deemphasize documentation.

Documentarians can expect the tooling and attention we believe our contributions and field deserve only if we're willing to engage with it the way our SMEs do. It's time to get hacking.

Tech Workers of the World, Unite!

The good news is documentation *can* be performed using the latest/greatest methods and technologies for collaborative writing, content management, version control, product delivery, and much more. What differentiates *Codewriting* from other technical writing and communications books is that it is solidly rooted in building cohesion and collaboration among product development (Engineering), testing and delivery (DevOps, QA), and product documentation (Docs).

This advice intends to build mutual respect among the different roles for each other's work, across the members of the product development team. For tech writers, it's all about drawing lessons from our colleagues who write more code than words. As repayment for letting us tag along on their awesomesauce, we promise to provide them with consistently improving documentation experiences.

I see the wisdom of developers and project managers from lean/agile backgrounds infused into this new approach. Tech writers and docs managers in forward-thinking environments have been building this trend for a while. It's an exciting time to be gleaning insights and strategies from leaders in a nascent field — I intend to include many of their voices in this work.

My point is not that conventional tech writing techniques yield poor results. Lots of talented technical writers and documentation managers do excellent work with what I consider inferior strategies and tooling. The new methodology I expound upon in this book has meanwhile produced only a small minority of the truly remarkable technical documentation projects of our time, and it has just barely been validated at scale.

Humility Check

I wish to note that while I talk up the DocOps/docs-as-code approach, I am by no means its author, and I'm definitely not its most skilled practitioner. So I'm not bragging when I argue strenuously that this strategy is "superior". This whole book is an attempt at documenting a set of ideas I stumbled into and feel remarkably privileged to have access to, *even though they're all totally free* (as in no credit card necessary and as in open source).

My argument is simply that this approach and these tools are better for collaborative software documentation (especially for agile or agile-oriented dev teams) than the conventional industry standards and stalwarts. All else being equal, a well-established docs-as-code/DocOps system should produce "better" content — more accurate, more current, more appropriate, etc — than the conventional methods.

This is just a hypothesis, far from proven. And no doubt a conventional system with more-skilled (or simply more) contributors might well produce better docs than the best DocOps platform that is not being properly used.

The DocOps Approach

Less controversial than my view on the industry's leading tools is my claim that a DocOps mentality will make you a better *technologist*. If you see yourself as "just a tech writer", maybe it is time to think again.

- Maybe you are a full-blown **DocOps specialist** — someone who arranges optimal docs environments for herself and the PMs and engineers she works with, all using a so-called "lean startup" approach, with your own team as end users.
- Maybe immersing yourself in the tools engineers use to accomplish their work will reorient you around the development process, making you better able to communicate with devs about the product and procedures. These tools include:
    - code editors and local development environments;
    - dynamic markup language with includes, conditionals, and variables;
    - distributed source/version control repositories;
    - semi-structured data in flat files; and
    - cutting-edge infrastructure management, automation, integration, and delivery platforms.
- Maybe with tech writers working in the product codebase and participating in key engineering meetings, docs will achieve "first-class citizenship", as CTO Eric Sammer explains making docs central to the Engineering organization at Rocana, which he did even before hiring me to drive them.
- Or maybe you just need help articulating the case for a DocOps/docs-as-code approach you're already salivating over.

If none of the above bullet points rings true, or if your current work situation will not accommodate the growth necessary to head in exciting new directions, perhaps this book may still contain valuable insights. It's main goal remains integrating readers' *understanding* of the development process and environment. And *Codewriting* definitely won't stop trying to help you to rethink technical documentation holistically.

Docs as Code

My mantra, *everything in code*, has a dual meaning.

First, all technical writing should be sourced in markup and compiled to rich output like HTML and PDF, or specialized output such as Unix "man" pages, user interface elements, specialized app content, or even presentation slide decks. This is a pretty broadly accepted technical documentation principle, especially considering pretty much *any* tool you can think of saves its files in markup, whether the user ever sees that markup or not.

Codewriting further favors writing *directly* in markup, as opposed to using a visualization tool that's generating XML in the background, such as Word and Google Docs. Tech-writing tools like oXygen, Adobe Framemaker, and Madcap Flare, which use DITA, Docbook, and other XML-based markups, attempt to provide a rich-text interface to the user, mercifully suppressing the verbose, hyper-nested tags establishing the document structure. The case for this is developed in the second chapter, Writing {preposition} Code.

Second, "everything in code" means put the docs in the product codebase — not in a database, not in a separate repo. This excludes most conventional wiki and web-based CMS platforms, as they depend on relational databases that hide the source behind a tool that is wholly inadequate for source and version control. We'll discuss integrating your documentation source and platform into the repo and the product itself. This is addressed in Docs-Code Integration.

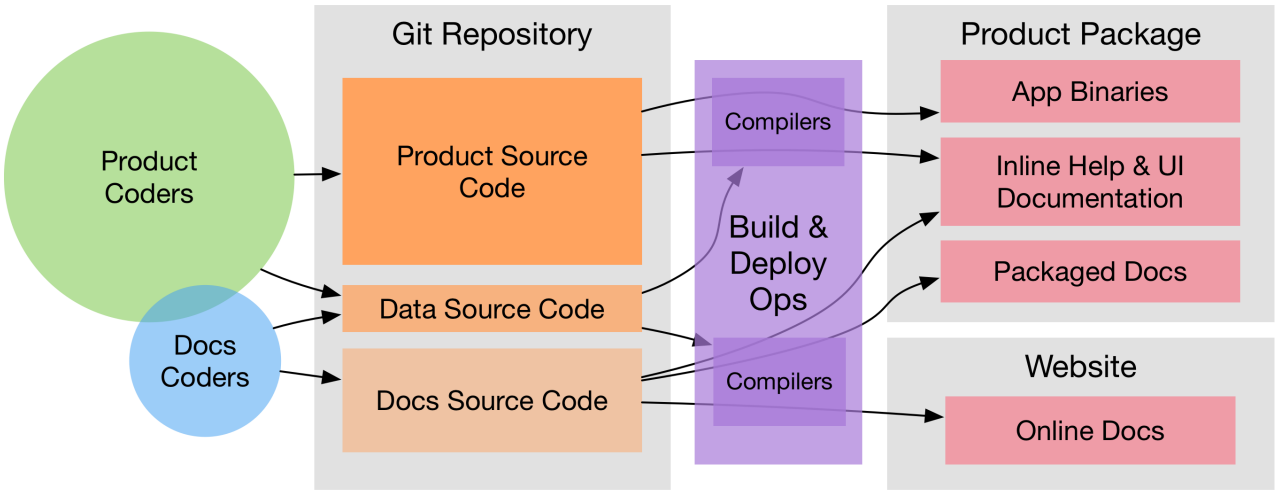

Figure 2. DocOps — General Concept

We'll detail this much further in a little while, but briefly: in the above diagram we see developers

writing code *and* docs in the same repo (or docs in a "subrepo"). The build and deploy platform compiles source code into product code, and it can do this with writing and graphics in HTML, PDF, or other output, as it can with any part of the product's user interface. Output is output — it can be packaged with the product or posted online, all with just a few commands during the release process.

There are several reasons I love the docs-as-code approach, but the key advantages are the flexibility it allows on both ends: input and output.

1. Lots of people in different roles can readily contribute to documentation efforts.
2. Content is managed as discrete components of a whole, instead of merely at the document level.
3. All content can be single-sourced, meaning tech writers and engineers maintain one canonical source of truth, no matter how many places that content needs to appear in the generated docs.
4. Besides the obvious PDF and HTML formats, content can be published as e-book formats, JSON objects, Unix man pages, even slide decks for presentations — whatever the situation calls for.
5. Conditions such as environment (where will it appear) and audience (end-users vs developers, for example) can determine what content is compiled in a given "edition".

How is any of this new?

Perhaps all this talk of flexibility leaves you scratching your head, since much of this is what every component content management system (CCMS) promises. So what's the big deal? Yes, the platform solutions I will describe in this book are technically CCMSes. But there is a big difference between the proprietary, corporatized packaged CCMS solutions on the market today and those being built from scratch by innovative codewriters, including people far more clever than I am.

You might think the biggest obstacle is budget. One of the common groans about commercial CCMSes is that they are pricey, so it would stand to reason that more low-budget or early-stage organizations would be turning to a DocOps approach. But some of the pioneers of this new wave of software documentation are at places like CA, Cisco, PayPal, Amazon, Rackspace, and Microsoft. Surely they could afford the highest priced consultants and enterprise solutions, yet prominent teams at these companies opted to hack their own toolchains using open-source ingredients.

Open Source Centricity

I love open source. I love it in principle, and I love it in practice. Free open source software (FOSS) gives us collaborative power commercial software will never permit. My bias on this matter will be on display throughout, so I thought I'd take a second to *prefend* it.

The author makes up a lot of words. He rarely explains them, instead expecting his audience to infer their meaning from context and root words. Apologies are offered in advance.

Allow me to briefly overwhelm you with reasons we should all use as much open-source software as possible.

Open source means access.

When we use and support open-source tools, we increase access to them for people with less means. Some of the biggest, greediest companies have realized the value of giving back through open source. More access means more contributors means more engagement.

Open source means power.

Inequitable distribution of power and inflexible hierarchies and workflows are hugely restricting factors in product development. Fast-paced engineering teams have no room for environmentally imposed constraints on what they can get done. Like DevOps, DocOps must make product developers (including documentarians) more productive, part of which is done by tweaking existing tools to accommodate agile content development needs.

Open source means transparency.

By definition, open source gives more people a view into our work. Transparency is good for accountability. Even if the audience that is getting a window into your work is relatively private (for instance, your engineering team), the point is to keep your technical writing copy in a repo others have access to.

Open source transparency means security and accuracy.

I think the ancient myth that exposing your source code makes you more vulnerable *per se* has been successfully debunked by now. But consider the implications of public audits of your docs, accompanied by greater capacity to incorporate users' contributions to complement or patch our work. If this sounds threatening at first, that's okay; open source collaboration has the natural effect of making you appreciate rather than fear *learning and taking effective action on* the inaccuracies our users are already seeing.

Discerning engineers prefer open source.

The most directly valuable reason you should favor and engage with open source solutions is that most of the best engineers are open source enthusiasts, if not devotees. Not only does this suggest there is something to the phenomenon, but it means you'll need to appreciate and get comfortable with open source if you want to earn the respect of the most discerning engineers you may work with. Using FOSS won't make you cool, but over-dependence on proprietary, closed-source tools will cost you respect.

Two Words: Distributed. Platforms.

These words aren't just my two favorite buzzwords, over which I'll spend much of this book poking fun at myself and the industry.

Distributed platforms mean *delegated contributing power*, which is key to comprehensive documentation, especially in growing, agile environments. If you want to be successful producing documentation for ever-changing, ever-expanding software products, the only thing I know for sure is you will need a platform solution.

Additionally, platforms are extensible. Any dynamic engineering team is constantly extending its tools. All those Slack, GitHub, and JIRA extensions, all that work in Docker, Chef, Jenkins, Gradle, and on down the list. That is platform configuration, integration, and extension.

It's Engineering pulling tools together to establish a badass development, testing, packaging, and delivery system. It doesn't work perfectly. There are complaints. The product keeps rolling out. It has bugs. They get patched. The world keeps turning, and life is a lot better than it could be.

Documentarians need to be in on that cycle.

The more complex your team and product, the more extensibility you are going to need. This means your platform will not come complete in a box, though there are promising SaaS (software as a service) solutions on the market already that meet many of the conditions I advocate in this book.

This book will help you work through the various options, including hybrid platforms that will scratch various itches coherently with a bit of iterative hacking performed along the way. In fact, we'll explore numerous tooling options as we plan and begin to build a platform solution as exercises in this book.

Lean Docs for Lean Projects

One last note before getting on with the book and exploring some topics in real depth: Documentation can be as lean and agile as any product code, even if it inherently lags behind in real time. The truth is, unless you are somehow afforded miraculous amounts of time to document your product and docs are part of your team's "definition of done" for a feature, it is likely that you will need to iterate from a "minimum viable product" for your user manual, help site, or other documentation.

Hopefully with whatever product docs you're working on, you and the rest of the product team can triage the following:

1. the essentials your docs need for their first version release;
2. what content needs updating every single release cycle;
3. what new types of content can be added during each cycle; and
4. any stretch goals that might enhance the docs in some way if we can get to it.

I started my job at Rocana over two years ago. At first, I began adding chapters to the *Rocana Ops*

Reference Guide, which I believe had three chapters when I found it. Then I went back and fleshed some of the older chapters out, or revisited old content and freshened it up.

Now I use that extra time to build systems so maintaining and freshening my docs will be more streamlined and systematic. DocOps has become a priority so that when we hit our next growth spurt, the documentation system will be able to accommodate more engineers, more product features, and more customers.

In a fantasy, I someday get to split the ol' RefGuide into several editions, all drawing from the same source codebase:

- an *Analyst's Guide* for end users who monitor data with our product;
- an *Administrator's Guide* to help configure and maintain the product;
- a *Developer's Guide* for our own team as well as third-party devs looking to extend our platform; and finally
- a *Field Guide*, which is a special edition for our engineers, including our testing and automation as well as customer success and sales engineers, where they can maintain version-controlled notes, scripts, snippets, workarounds — a knowledge base cohabitating with the product in one happy source repo.

My background in lean startups and agile development operations has given me insight into the applications of these mindsets to docs, and even with just what I can envision, the possibilities are very exciting. I want to share them with you, because there's no way I'm going to pursue them all myself, and life is too short for proprietary knowledge.

Furthermore, I suspect much of what seems like limitations of this approach may just be the boundaries of my mind and brief experience. I hope readers will prove me right about docs-as-code by showing just how little of the picture I foresee, even in my optimism.

Codewriting as Self-improvement

I am researching and writing this book so I can get better at what I do. I currently have no tech writer peers at work. This lack of direct peers has had the added benefit of forcing me to have esoteric conversations with engineers and my manager (also a developer), but I do from time to time want to talk about documentation with colleagues who live and breathe docs. My current team appreciates docs more than the average engineer, but not the way you probably do, dear reader.

I believe the exercise of writing this book will improve my skills in all of the above-listed objectives, which I may or may not already do well enough to instruct on. I am trying hard to write what I know, declare speculation as such, and generally be conservative with recommendations and judgments.

The thing is, I'm not very conservative. I have high-minded ideals, and I take a lot of risks in life. Feel free to keep my recklessness in mind.

I hope if you follow along with my experiment, you will learn with me. If you are moved to contribute and teach me directly, I will be grateful beyond words, though words are all I have to offer as compensation. We're doing this FOSS-style, remember?

Part One: Writing

No matter what else you learn or forget about the work of technical documentation, remember as much as you can about *writing* and *writing technically*. The rest changes a lot from gig to gig, but the need for clear, concise, and compelling technical content will remain. Documentation workflows and methodologies, platforms and toolchains — all the stuff you need proficiency in to call yourself a DocOps practitioner matters not at all if you cannot write complete, accessible product explanations and instructions.

The chapters in Part One address the craft of writing about software products. The rest of the noise of the field can wait; the one thing we all have in common is a love of language, so there we'll start.

Chapter 1. Docs

The cliche advice for writers of all kinds is that we should read more to get better. Mere reading does not suffice for technical writers. It is a hard pill to swallow, but there is no way around the first rule of writing, modified for our field: you have to use lots of product documentation in order to write good product documentation.

What Docs Are

As you'll see, I have a liberal view of what constitutes "technical documentation" at a software company — probably more in tune with what many call "technical communications". I also have a broad definition of what falls under the purview of technical documentation management. I prefer to be within one hop of every significant technology document the Engineering organization touches, from knowledge base articles to slide decks to drafts of blog posts and white papers. This does not mean I want to be in charge of it all; I certainly don't want to review it, let alone edit it. I just like to know what is where, or at least how to find everything. And it means I want the engineers I work with to know I care about all that stuff.

I work regularly with developers and project managers in Engineering, as well as everybody in Customer Success. Having *their* docs in order makes my life better in numerous ways which we'll explore throughout this book. For now, I want to introduce the theme that bad documentation is everybody's problem, not just users'. And documentation itself is way more than just the docs you prepare for end users or third-party developers, or even anything you might need to produce for Marketing. It's all of the meeting notes, specs, task-management issues, Git commit messages, help desk tickets, and on and on. Well kept, these are also your assets.

This book is going to dwell plenty on what tech docs *are*, but first let's clear up what docs *aren't*.

Docs Are Not for Reading

I try not to refer to *reading* docs, but rather *using* them. That is because I believe, quite simply, that *good technical writing is not even meant to be read*.

Okay, if you consider books like this one to be *technical writing*, then *some* technical writing is meant to be read like prose — although I do not expect it to be read cover-to-cover or even in order. Still, most product-related technical documentation will *not be* read, no matter how it was intended by its authors. And if your work will not be read conventionally, it makes little sense to write it as if it will be read conventionally.

If you're a technical writer more than you are a developer (and no, engineers, it doesn't count that you *feel like* you spend more time writing docs than code), you probably love to write as much as any novelist, journalist, or poet. If you're anything like me, you're working on a novel or three, and maybe

you'll get the nerve to submit that cookbook to an agent someday soon. But the only thing that's getting *published* at the moment is that damn user manual, so you'd better believe we're going to *write* that thing as if someone is *reading* it.

Usually there is even a spot here or there, in the introduction or a sidebar, where we can open up that throttle and write a few contiguous paragraphs of prose, really let the wind of the open page have its way with our free-flowing hair. We cherish the relative latitude and runway of these interludes.

But mostly we don't write; instead:

- We *keep track of* content that fits better in tables, charts, and lists than in paragraphs;
- We *follow our own instructions*, hoping to catch docs bugs before customers or developers do.
- We *chase down developers*, explaining that cave etchings are preferred to nothing, but secretly wondering why people who can code in seven programming languages can't be bothered to write a formatted list in English.
- We *pester our users and coworkers* about the docs, whether they use them, "Why not?", "Could you maybe just give them a try?", and of course the old, "Yes, they were included with the installation and are on our website".
- We *prune, reformat, and edit* scribbles, notes, and sometimes well-written content from SMEs.
- And we *create visualizations* we hope make sense to people who actually use this stuff (be that an end user or a designer who will upgrade our graphics).

When you look at a given page of a user manual or a section of a README, there might seem to be about as much of the *writer* in *product documentation* as there is the *jockey* in a *horse race*. It absolutely matters who is on the horse, but everybody knows the best horse will win as long as the jockey is competent.

But this analogy falls apart when you consider the variety of modern software products, and just how easy it is for today's software to depart from convention and require unorthodox documentation. If you are fortunate enough to document a product that is trying to reinvent its space, there may be no blueprint for how to convey that product to users. They may be more dependent on your contextualization and conceptualization of the product than they typically need to be when they encounter a "new" solution, since most product interfaces are essentially ripoffs of the most successful player in the space.

Just because we do not always get to spout off freely as I am right now does not mean decisions made by technical writers are not critical to the success of a product. Docs may not close deals very often, but bad docs can frustrate, costing degrees of a potential customer's confidence.

Reading Docs

All of that said, I do think one can appreciate good docs at face value. Sometimes when I'm looking for a way to solve a problem, I run into some incidental aspect of another product's documentation, which I

can appreciate only in the abstract. Even at a glance, I recognize excellence in presentation, and the hair stands on the back of my neck. We lucky kinksters who appreciate good docs on an aesthetic level have a distinct advantage.

Techdocuphilia Exam

Perhaps you are one of us. Which of the following scenarios applies to you? For each, answer **Not at all**, **Somewhat applies**, or **Oh, yeah, that's my jam**.

Table 1. Documentation Arousal Test

Scenario	Nope	Mabes	My Jam
You encounter an elegant quickstart landing page starting with a prominent `brew install <delightfulness>` or `apt-get install <something-dangerous>` command.			
Google led to the wrong product version, but once on the vendor's website, you were only one click away from the corresponding material for the needed version.			
Just what the doctor ordered: a particularly well-formatted table with a perfect theme and cadence to column headers, plus just the right data in a just-right arrangement.			
No matter where you look in a product's docs, a frequently used reference is available wherever it may come in handy — each version identical and accurate, no hyperlink required.			
A list of examples ends at the ideal length.			

There is no shame in loving good docs. In fact, there is no shame in loving *bad* docs.

Docs Tell a Story

Product documentation contributes to users' general impression of the product and the company that made it. Much as I'll talk about modular docs (see Topic-based Authoring) that can be parachuted into, documentation should paint a complete and accurate picture. They should make *mastering* your product seem plausible.

I originally titled this subsection "Docs *Should* Tell a Story", but the truth is, even poor docs tell a story. We'll get to that story in a second.

Reading Bad Docs

One of the first things you'll hear when you take over technical documentation for an organization is which of their competitors or upstream vendors they do *not* want their product's docs to turn out like. Take heed of this; while it's possible some bias against the third-party product is tainting their attitude

toward the docs, the opposite is far more likely. Your new coworkers are probably referring to poor documentation of a product they actually admire, if not a product they regularly use and may even adore on some level. Learn what frustrates your coworkers about those docs, then try actually using the docs with that third-party product.

We'll talk a lot about getting to know how developers operate. Earn the respect of coders you work with by walking in their shoes. If several engineers on your team are frustrated because they've been told to work with a dependency library they don't love, there's a very good chance the library is poorly documented. Go investigate and maybe play with that dependency. Even if it's above your skill level and you fail, the process will yield benefits, empathy not least among them. You will have a better understanding of what developers are talking and complaining about, and you'll have a functional analysis of what is frustrating about docs your coworkers use every day.

If you can repeat this process with users, whether they be third-party developers or end consumers, you'll glean even more insights.

Talking Docs

Ask your friends and family whether and how they use docs.

Docs are for Using, So Use Away

Use lots of documentation, and keep track of how you use it.

Skim the Docs

Also in the "let's not kid ourselves" category: let's not pretend people *read* the section of the docs they're using, even when they must use the docs. Probably the smarter they (think they) are, the less attention they actually pay to documentation.

This is certainly true for me. I must admit I tend to skip to the place I need — not just the section, but then within that section. I have a tendency to skip over introductory text and go right to a reference or list of steps. This is poor practice on my part, but good documentation is ready for it.

Good docs bold (or otherwise highlight) critical information in large text blocks.

Admonition (a.k.a. "callout") blocks are even more unmistakable. They should be used judiciously; crying wolf with lots of bright, scary boxes of `WARNING` and `NOTE` content can desensitize users.

Admonitions are incredibly valuable, so earn users' trust by keeping them concise and critical.

README, the Root Doc

The README file is the heart of any software project. It is not traditionally the domain of an end-user documentarian, but DocOps specialists will at least want to begin appreciating the value of a good README, even for proprietary (non-open-source) products.

Even if it's not your job to write the official README for your project, if you already write docs for a software product, consider creating a "shadow README" on your own time as you work through this book. Maybe by the time we're done, you can offer concrete contributions to your real-world product's README.

A good README contains all the basics needed by core engineers, third-party developers, and daring end users/beta testers alike. Even for a polished product that can be delivered through an app store or a package manager, the README holds it all together.

At risk of exaggerating its importance, I really do believe the README is the most important *file* in any software project — it's the starting point for code and docs alike. And this is where our first exercise comes in, intended to keep us thinking about that lean, iterative approach to building out a product's docs.

The best README files are accurate; everything else is gravy.

This cannot be overstated, and I'm as guilty as anyone of letting READMEs languish. If your codebase is ahead of your README, you're failing everybody on down the line. Engineers, documentarians, third-party committers of either type, and early adopters (who enable your repo to become popular) all depend on your flagship file not pissing them off.

In terms of actual content, the README provides exactly the appropriate amount of information to get started with your product, from any role. If the repo is just for developers to fork and build, then link end users to the downloads page right at the top. Don't waste their time. If you have a separate wiki or guide set up for each role, pass everybody along to the right landing page for their docs needs. But for an immature project, the README should be fairly rich. You'll want the following sections, as soon as you're able to declare *any* high-level statement about them.

overview

> What does your product do. This may seem obvious, but a shocking proportion of open source projects seem to assume we already know exactly why we've arrived at this product.
>
> A mature overview includes:
>
> - who will find this product useful
> - what dev languages are used
> - any environmental requirements

- an honest assessment of the project's status and progress

quickstart/demo

Surely your product has a bootstrap or quickstart option, or an online demo. Or at least some screenshots. I need something early on that *does* cater to the audience that knows (or has just decided) this product *should* meet their needs, so they're ready to dig in and see it action. The more hands-on, the sooner, the better.

installation/setup

If all you showed me in the last section was screenies or a demo, it's time to get me up and running. Maybe setup isn't so simple; let me know that now, but invite me to struggle along with you. Don't decide for me that I don't need your product badly enough to pull my hair out voluntarily to integrate it with my needs.

usage

Okay, now that I'm installed and configured, how does it work again? Maybe you're not ready to give me the complete lowdown, but I need at least a nudge.

contributing

Now that I'm playing with your software, how can I report bugs or share my patch proposals?

A good "Contributing" section includes:

- a warm, authentic message to reassure potential contributors that you actually want their input
- an explicit list of the kinds of contributions most needed
- clear instructions for each type of contribution
- an overview of the review/approval process for code/docs contributions

licensing/redistribution

Make it as clear as you can how you intend the product to be reused. You don't have to promise to support users in any way, and you needn't provide any kind of warrantee, but you should make it clear the conditions under which your source code may be modified and redistributed.

We will be exploring README file contents throughout the rest of this book. Not only is it the first source file your team should ever initiate, it also must be reviewed with every new merge or release. And while it's okay to begin with a nascent, skeletal README, eventually it should be a reliable of your project.

Chapter 2. Writing {preposition} Code

One of the more common attributes of "docs-as-code" is writing docs directly in markup. This brings documentation a step closer to product code, even if you're not yet keeping the docs in flat files inside the product source repository. Just as importantly for many, it liberates the writing process from proprietary tooling and overwrought interfaces.

What's with that Chapter Title?

I wasn't sure whether I wanted to title this chapter "Writing *in* Code" or "Writing Code", and I thought there was some chance I might make it "Writing in Code, about Code" or some other permutation. The uncertainty is perfect for introducing one of the most crucial technical elements of codewriting: dynamism. Allow me to demonstrate.

If I wanted to be able to publish different editions of this book in which this chapter title were variously named, I could insert a variable between "Writing" and "Code". That variable is a token that could be replaced by a value determined at the time of publishing, set to either a blank value or `in` or `in Code, about`. These would yield their respective prepositional representation depending on which audience the edition was for:

- "Writing Code"
- "Writing in Code"
- "Writing in Code, about Code"

Now, that anecdote is totally fabricated, and it's not even the most realistic example, but it provides opportunity to start making a case I'll develop throughout this chapter. My point isn't that your text has to be peppered with dynamic elements but that your documentation as a whole should be flexible and adaptive. First thing's first, however; let's take a look at that text and code.

Dynamic Writing

As we'll explore more when we look at how best to serve users' needs, the ability to serve just the right content to just the right audience in just the right place can provide great advantages. This is especially true for mature products with multiple tiers or roles of users, such as enterprise platforms.

What I've typed to make the chapter title above is the following string:

```
= Writing \{preposition} Code
```

This is dynamic lightweight markup, intended to be "parsed" and output in various rich-text formats. My dynamic lightweight markup language (DLML) of choice is called AsciiDoc.

In AsciiDoc, the `=` indicates I want the text that follows to represent a top-level heading (title), but let's set that aside for now. The real curiosity is `{preposition}`. In AsciiDoc, I can indicate a placeholder for content by wrapping the the key in `{ }` characters.

The `\` here is because in this title, I want to show you the *actual tag*, instead of parsing it with the value of a key named `preposition`. In programming, this is known as *escaping* code — using markers to indicate that you *do not* want a portion of code to be processed normally.

I can declare and then reference what AsciiDoc calls an *attribute* (known generally as a *variable* in programming as in elementary mathematics):

```
:preposition: in
= Writing {preposition} Code
```

You can see what happened there, but maybe you don't see the advantage. Why not just write the word `in` instead of `{preposition}` where I want the text substituted?

What if I were to have set up a "switch" of sorts, in order to set the value of `preposition` variably, depending on some extra input? For instance:

```
ifeval::[{audience} == "developers"]
// assign blank value
:preposition:
endif::[]
ifeval::[{audience} == "techwriters"]
:preposition: in
endif::[]
ifeval::[{audience} == "mom"]
:preposition: in Code, about
endif::[]

= Writing {preposition} Code
```

Now, we have to pass into this bit of code a variable named `audience`, which we can do when it comes time to build our document, depending which edition we're generating at the time. Once we've set `audience`, its value will determine whether `{preposition}` resolves to `in`, `in Code, about`, or remains blank. To solve our problem of having three different editions depending on who will read it, we just generate a customized doc for each audience.

This isn't likely to be an entirely new concept to any tech writer, let alone any developer. My real interest is to start showing off the dynamism made possible by DLMLs. Tag-based languages like DITA may provide a degree of dynamism, but for reasons we'll explore more later, they tend to limit collaboration.

Non-dynamic lightweight markup languages are the other contrast. Markdown is the most popular lightweight markup language, and it is highly conducive to collaboration, but it does not provide for

dynamism or semantic contextualization.

For tech writers, my promise is that there are amazing tools that will give you not just more technical power, but more buy-in from Product and Engineering. With tooling integration and SME collaboration, stakeholders will see that small investments in documentation can pay off in big ways. We're out to make the case that smart investments in DocOps can yield a bigger return than shiny solutions with big price tags.

For engineers, I'm promising you there are better ways to write great docs that center around users' needs, and you don't have to learn obscure, proprietary tools to take advantage. The truth is, engineering teams have been documenting products in code since long before this trend hit the technical communications world. Now, whether you have dedicated technical writers or you're expected to write and maintain the docs yourselves, engineering teams can document the product as well as they can document the code base or an API. And they can use tools that combine the best of the engineer's world (flat files, distributed source control, and build tooling with continuous integration) with the style, semanticism, and output flexibility of advanced tech-writing solutions.

Lightweight Markup

I feel strongly that I cannot go wrong encouraging people to try AsciiDoc. The response I've received from engineers so far has been overwhelmingly positive, and after more than two years of constant use, I still find examples of elegance and forethought in AsciiDoc all the time. As I explore its extensibility, I'm fairly confident I can solve nearly all of the hardest DocOps challenges I face with AsciiDoc-centric toolchains and frameworks.

As you evaluate my proclamations throughout this book, keep in mind that I do no internationalization or localization work on my technical docs. I am simply not yet able to incorporate translation and related challenges into my assessments.

That said, I highly respect reStructuredText (RST) and believe it is likely just a matter of one's taste in markup style and the platform for extensibility (Python for RST, Ruby or Python for AsciiDoc). I have not had the opportunity to use RST in production, and I'm just writing what I know.

We'll also look at DITA, the leading dynamic markup language for technical documentation. While not a lightweight language, DITA-based platforms offer some advantages we will explore. I will admittedly spend most of that effort deriving lessons from the highly capable and experienced DITA community in order to apply them to DLML-oriented solutions accessible to the rest of us.

Markdown will make several appearances as well, since innovators have found ways to give Markdown some quasi-dynamic superpowers.

Dynamic Lightweight Markup Language

AsciiDoc and reStructuredText differ from Markdown in several key ways. At a minimum, what makes them "dynamic" are the following characteristics:

variables

A DLML lets you set a key/value pair that can be represented by tokens throughout the document, which will be substituted with their value when processed.

conditionals

A DLML includes syntax for if/else/then logic for creating "switches" that generate different output based on their context.

includes

A DLML lets codewriters pull source and other content in from separate files in the codebase, extending content reuse potential.

extensibility

There must be at least one actively supported standard or platform for extending the DLML by adding dynamic and semantic capabilities.

A great DLML has many more features, as we will discuss later, and there are attempts underway to extend Markdown to incorporate at least some of these capabilities. A little extra tooling can enhance a Markdown-sourced docs system to accommodate dynamic features. For now, AsciiDoc and RST are a ways ahead.

In the end, I do not argue that the nascent docs-as-code movement should reject tag-based markup or non-dynamic lightweight markup in the documentation source. I can certainly think of legitimate documentation cases where variables, includes, and conditionals are simply not called for. I also know there are teams that love DITA and make widespread use of it and its tools, collaboratively and in close connection to the product, even using Git-based solutions.

Source code is source code, as we'll explore much further in Source Coding. And we have more than enough orthodoxy and invalidation in this world — there's no need to draw us/them boundaries among people trying to achieve a common goal of collaborative documentation inside the product codebase.

Semantic Structure

When I first learned to "write" HTML in 1996, I did not actually write very much HTML. The WYSIWYG (what you see is what you get) editor had already emerged, and it got all those `<i>italic</i>` and `<a href="some-long-url">link</a>` tags out of the way.

This meant I rarely had to see the code I was writing in. I only looked under the hood to fix an editor-

generated bug or to write some HTML the editor could not yet do for me with a few menu commands. When it came to writing, I worked in a tool that was essentially as elegant and practical for *writing* in HTML as the best WYSIWYG editors of today. The editor abstracted the HTML, showing me something more akin to what the world would see upon publication.

When I learned to write functional software (not just static web pages), I started to appreciate seeing all that markup, as well as all the scripts and database calls that were making my page content powerfully dynamic.

Dynamism is no small thing in digital content, but nothing is more fundamental to tech docs than *semantic structure*. Your content has to have structure that conveys *purpose and utility, not just placement and style*.

Every chunk of quotation, every admonition, every diagram, every code listing, and every instruction step — all your content has potential relevance to its digital context. More than mere clusters of characters or bytes, the assortment of files that make up your docs can have various relationships with a range of documents and media, from a print manual to the product interface to video to a presentation slide deck about the product. So it matters what you put behind your words and pictures in order to indicate *if* and *how* they should be conveyed to the audience.

A vendor referred to this as “What You See Is What You Mean” in describing how their DITA editor GUI handles semantic text.

Semantic structure can get quite heavy. Look how one popular DITA publishing tool handles semantic markup in its visual editor.

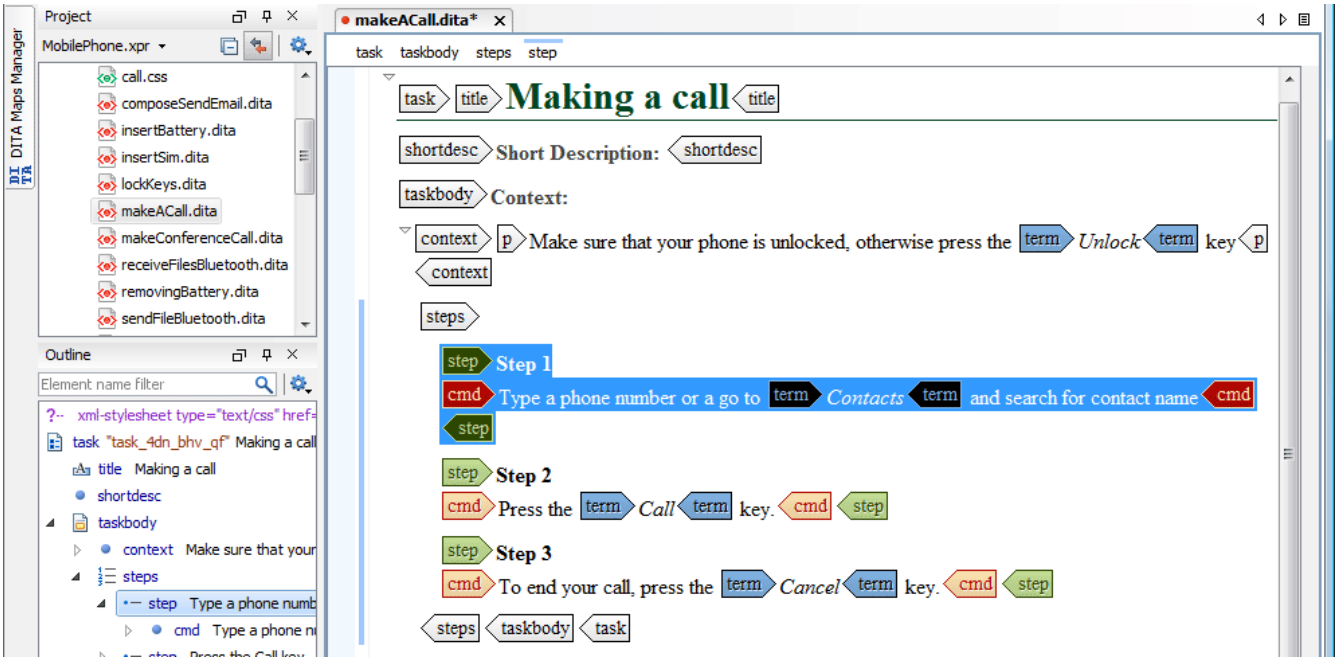

Figure 3. Semantic structure in Oxygen XML Editor (Source: oxygenxml.com (c) SyncRO Soft SR, republished under Fair Use)

This may or may not be pleasing to look at in this form, but you might be able to imagine the potential to

display this as highly visual instructions.

I intend to explore this in greater detail; contributions and suggestions are very welcome!

When you are looking at unobscured code, you are seeing not just the semantic purpose behind your content; you're seeing what exactly determines those semantics.

I felt this effect when I first let go of that WYSIWYG editor and let my HTML and other markup onto the page. It was a pain in the ass because of all those < and \ and > characters, not to mention the extra " and = marks everywhere. And the nesting; oh, the nesting.

```
<parent>
  <child>
    <grandchild>
      Some content.
    </grandchild>
  </child>
</parent>
```

But there's no denying this form of markup offers a means of directly analyzing the root cause of any markup errors, rather than hoping your tooling can identify and correct the problem automagically.

What if you could have the freedom of writing in source without the burden of all those extra characters?

Code is Powerful

If you've never written and executed your own code, it might be difficult to appreciate the power of programmatic writing. We're going to overcome that during the course of this book as you see the power of treating your documentation as an application. It's true that traditional content management systems (CMSes) offer remarkable publishing capabilities. But they also sell you short on control over your docs at the source level.

Only languages and toolchains that offer you inline dynamism — and here I mean DITA as much as AsciiDoc or RST — actually enhance the power of your writing. The ability to visualize and use programming logic as well as semantic tagging should sufficiently entice anyone inclined to integrate docs and code, as it were — a topic we'll explore soon.

De-abstracting Content

You'll pretty much never see a serious software engineer writing source code using tools that hide the code of the programming language they're coding in. There are exceptions to this for complex code, such as formulas and algorithms, which are perhaps better generated automatically than having typed every character written by hand. And some coders like to collapse portions of their code from view when

they're not touching or referencing it, which feature modern code editors typically offer out of the box.

But for the most part, developers strive to achieve a sort of Pareto efficiency with relation to their source code, to use an economistic notion. That is, good coders get as close to the source code as they can without adding undue burden. Their toolbox is such that any change would decrease the overall effectiveness of their programming. I have never personally approached this mythic relationship to code, but I have heard it spoken of under certain influences, and I believe I've caught glimpses of it in the wild.

These exemplary coders use dependency libraries carefully, but not because they're afraid to code the perfect solution themselves. They recognize the trade off each time they choose to fork a project or set out anew. Going with the current of the open source crowd has significant benefits; nobody is saying you need to be a pioneer to take advantage of the open source universe.

Technical writing and editing workflows, as well as their tooling, can follow the same principles. We can look our source code in the eye, as I am doing while typing these very words. I don't see the clusters of `1`|`0` combos behind this text, but I do see the markup notations — in fact, I type them explicitly. Our editor can help us cheat a little, with distinctive formatting inline and a WYSIWIGish view in an adjacent panel.

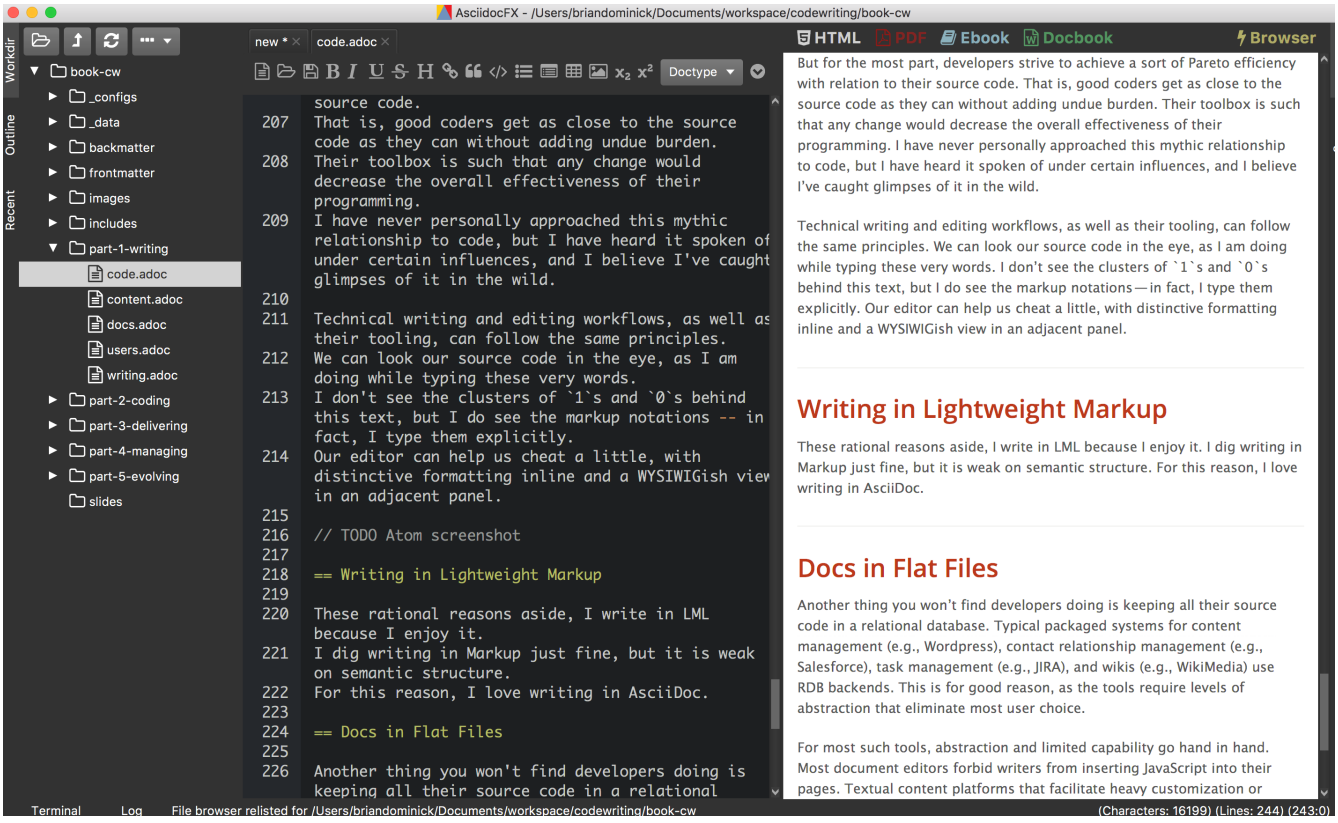

Figure 4. Since originally writing this section, AsciiDocFX has become more photogenic for AsciiDoc display than my preferred Atom editor.

In the end, your eyes and your preferred editor will help you blend your formatting as you type or scroll down your doc, just as cleanly with more direct at-a-glance insight into semantics and dynamics than you have in an advanced word processor. No kidding.

Docs in Flat Files

Another thing you won't find developers doing is keeping all their source code in a relational database. Typical packaged systems for content management (CMS, e.g., Wordpress, Drupal), contact relationship management (CRM, e.g., Salesforce), project management (PM, e.g., JIRA, Basecamp), and wikis (e.g., WikiMedia) all use RDB backends. This is for good reason, as the tools require levels of abstraction that eliminate most user choice.

For most such tools, abstraction and limited capability go hand in hand. In fact, all these management systems typically offer popular cloud-based options precisely because customers have realized that conformity is an acceptable price to pay for platform supportability and stability. Platforms remove headaches. For many choices facing a technical organization, going with a platform can be the best call.

Flat-file Anarchy

Coders are less likely to see it that way. Coding UIs are of course far more anarchic than PMs or CMSes, and not because engineers just want to be cowboys (even the ones who do). They reject systems of constraint that come in the form of form-field validations that reject improperly formatted entries; coders write their own integration tests, thank you very much.

For coders, the flat file is the Wild West. It's Ctrl+Shift+N, followed by typing a file extension that reveals the source language (`flat-file.java`, `flat-file.py`, `flat-file.rb`, `flat-file.js`, `flat-file.json` etc.).

You see the commonality.

The flat file offers not just a fresh slate, it enables source control. Many advanced CMSes offer version control, which can trace drafts and compare differences across iterations.

Source control offers all of this with the added power of *forking*. That is, independent copies can live in separate code repositories, changing and potentially reintegrating down the road. The power of this for documentation may not be immediately obvious, but as we go on, we'll see it illustrated in a few key ways.

The point is, coders love the open-ended potential of the flat file. But coders aren't berzerkers.

Reign of the Review

The virtual space of the development environment is often anything but chaotic. Great tech leads, project managers, and peers impose on engineers the kind of order you'd expect from strict editors in journalism.

The environment and workflow can have severe constraints in place at points prior to code integration. A sloppy developer like me may spend weeks thwacking freely at the keyboard before submitting the resulting source code to unit tests, regression tests, and however many rounds of peer assessments may be waiting during code review. In the end, all code must pass muster.

I'm not going to try to sell you on the complete vision just yet — I merely wish to make the case that flat files can be as simple and manageable as wiki pages, yet they leave the platform open to more flexible construction of document and data structures on the fly.

In this case, the "platform" may be more of a "framework" such as those used by developers to provide logic, convention, and usually a great deal of prepared dependency code. A development framework is a relatively open playground compared to a CMS, but don't let this notion scare you off if you're not a developer or hacker (yet).

At the same time, if you're a serious coder, I don't want to oversell the docs frameworks you'll find out there. I don't even know if this approach will prove popular, let alone get the support it needs. It's early days, and we'll discuss this more in Hacking.

The point of a CMS platform or a codewriting framework is that something is in place to help you get started producing content. It's not just a blinking cursor on a command line.

Chapter 3. Writing Content

A lot more goes into good user docs than clear instructions and illustrative screenshots, but it is not always obvious which elements users will need. The kitchen-sink approach is tempting… until you sit down to work facing a deadline. How do we decide what our users need, what would be nice but unnecessary, and what is superfluous or distracting?

The Craft

From the end-user's perspective, there are only two aspects of documentation that matter: content and delivery. Even if users could learn all about *how* you made the docs, they wouldn't be interested, and they wouldn't appreciate it. That's life. Our job is to create docs that are helpful and to get them to users in a sensible format where and when they need them.

This and other chapters will touch on *what* goes into good docs, but *how* we keep our docs in good shape remains the main focus. In maintaining good docs, no concept is more importance than the ability to establish and maintain the elusive "single source of truth" for all the content you generate, be it in the docs or in the product.

Keeping Docs DRY

Put all the information a user needs right at their fingertips without creating half a dozen sources of truth for every fact. The developer's mantra of "Don't Repeat Yourself" (DRY) applies every bit as much to technical content source as to product source code. You may display the same table two different ways in four or fourteen different places, just to be sure it is always at your user's fingertips, but keep only one copy of that table or its source data.

In the technical documentation world, you're more likely to hear this concept called *single sourcing*, which I'm also down with.

Topic-based Authoring

One of the key contributions of the DITA community is the strategic modularization of content in reusable chunks, which aids single sourcing but also helps us think about how we organize and deliver content. Known as *topic-based authoring*, this approach favors clean reuse of content source in different contexts by helping us think about content in its simplest useful form.

Wikipedia does the strategy some justice:

> A topic is a discrete piece of content that is about a specific subject, has an identifiable purpose, and can stand alone (does not need to be presented in context for the end-user to make sense of the content). Topics are also reusable. They can, when constructed properly (without reliance on other content for its meaning), be reused in any context anywhere needed.
>
> — Wikipedia, 2017-04-29

I'll get into some examples momentarily, but I don't want to introduce this topic without expressing its greater value to content strategy. As a side effect, topic-based authoring prompts us to think of content in logical blocks. I'm sure every team/style/project will define topic parameters and properties differently, but I must admit I've found this enormously helpful in my own work Even as it suggests I should make Herculean efforts to overhaul the structure of so much of my work, I know the benefits will be huge. Formally achieving what I'll call topic-based DRYness will give me all the building blocks of a truly modular docs system at Rocana. Publishing all those various guides and knowledge bases for various user audiences, all from the same canonical source, will then be a matter of just listing topics to appear in each final document.

Topical Thinking

In its simplest form, topic-based authoring has come to mean keeping my content files to the minimum size I could reasonably approve of devoting a web page to. Right out of the gate, that's a subjective judgment, but since I have spent 20 years building websites, I have a pretty concrete idea of what size and scope of information should get a dedicated page on a website, and what scopes should not.

For example, a complete series of simple installation instructions deserves a web page; each step in that simple process most likely does not. A bibliography or glossary deserves its own page on a website; a single citation or term definition does not. A person's complete dating profile deserves its own web page; the summary of their image thumbnail, age, and match rating belong in an index listing, not each to their own page.

Another way of looking at it: if a discrete content item might serve well in a "tooltip" or "popover"-style element on a web page, it probably lacks the makings of a topic. The page it is on is likely made up of one or more topics.

The supersets of topics tend to be "documents", to invoke a dangerously broad class term. Think of *web pages*, *articles*, and *chapters* as documents, even where they are also children of parent documents (*website*, *periodical*, and *book*, respectively).

In any case, a document is suited for topic-based authoring when it:

1. must reflect highly technical and exact information
2. contains various types of content sub-elements, such as
    - code listings
    - admonitions

    - instruction steps
    - diagrams
    - sidebars
    - citations
3. includes content that is best single-sourced in order to constrain divergence during reuse and redistribution

Point **3** here is crucial: we're only going to use this method for content that should not be expressed in substantially different ways such that just writing it differently each time makes more sense. This excludes concepts that should be re-expressed with nuance or fundamental differences in various places, if only to help different users learn something a different way or from a different perspective.

Single-sourcing vs inline expression

In our product Rocana Ops, the most fundamental concept we need our users to truly understand is how we treat a type of data we call an *event*. We're not the only ones in the industry to call it an event, and not everything that we call an *event* is actually what you would think of as an event. For this reason, there are several paragraphs throughout our docs that help users understand what we mean by an *event* in Rocana Ops.

In the Introduction chapter and other materials that might be read by somewhat less-technical users, an event is described in a way a Chief Information Officer at an enterprise might need to know it. This person knows at least generally how IT works and can conceptualize what our product specifically means when we describe this type of data.

As the chapters go on, more-specific roles of user necessitate different contextual understandings of the event structure and usage, so in each place, I simply write about events in a context-appropriate way.

But when it comes to the strict AVRO data schema that defines and constrains these data objects we call *events*, divergence from the exact truth is pretty dangerous. Ideally, we want to make sure the example event schema shown in the the Reference Guide is drawn from *exactly* the same file that is used by the product itself. Engineers can thus make edits to this canonical file that will automatically be reflected in all our docs talking about the event schema.

If the changes are significant, they'll probably need to be reflected elsewhere in the docs, perhaps even in some of those various descriptive paragraphs about *events*. But at least our literal expression of the event schema will stay consistent, even if I fail to note the changes, I won't have to actively copy them from the developers.

This concept of choosing *not* to treat certain discrete items as topics in and of themselves may require its own treatment. We'll start with an example that works in the other direction, where we definitely want

to use consistent language.

Let's consider a commonly reused admonition for a dangerous upgrade procedure:

Back up your data before performing this step.

Here is how I defined this chunk of content in my source, which I've just typed into the very same file this current text is typed into:

```
[WARNING]
Back up your data before performing this step.
```

In my view, this block is too small to constitute a topic in and of itself. I conclude this even though I may reuse that language in 36 places across three versions of a product being maintained, with four to nine places this exact admonition text appears in each.

With content like this, it would not even be the end of the world if we decide to update this language some day and have to do a source-wide search-and-replace operation across the docs portion of our codebase. (In fact, even most database-backed content-management systems enable this.)

I'm also not likely to want to create a new source file for each instance of such a simple snippet of content. You won't find this anywhere in my several projects of AsciiDoc files.

Example File — topics/admonitions/backup_warning.adoc

```
[WARNING]
Back up your data before performing this step.
```

No, you won't find files containing such simplistic, limited content, but you will find me craving the convenience of knowing that kind of precise knowledge or language is maintained in and drawn from a canonical source, and that I don't have to remember to search-and-replace any change to it. I find it too burdensome to give each such element its own file, but I want the advantages of single-sourcing.

Consider the elegance of being able to simply edit a single source for this discrete chunk of text. In this example, it has been brought to our attention that our users think backing their data up to their own account on the server suffices for this step, but we really want them to *download* a backup. We'll explain this elsewhere, but when we drop the warning in various places in our docs, we want them to be reminded that we mean saving the backup to their local machine.

Example File — topics/admonitions.adoc

```
// tag::backup[]
// tag::backup-title[]
.Backup Warning!
// end::backup-title[]
// tag::backup-warning[]
[WARNING]
// end::backup-warning[]
// tag::backup-text[]
Back up your data _locally_ before performing this step.
// end::backup-text[]
// end::backup[]
```

This source arrangement gives us some decent flexibility in how we output this later.

Imagine being able to keep the core statement of our admonition while enabling us to give it greater or lesser emphasis when we call that core text.

Example dynamic expression of discrete data

```
. Update the data schema.
+
include::topics/admonitions.adoc[tags="backup"]
```

This source will call up the whole block, like so:

1. Update the data schema.

Backup Warning!

Back up your data *locally* before performing this step.

Maybe we want to apply a little less emphasis on the warning later.

Example restricted expression of discrete data

```
. Update the data schema.
+
[TIP]
include::topics/admonitions.adoc[tags="backup-text"]
```

This turns our admonition into:

1. Update the data schema.

Back up your data *locally* before performing this step.

Overinstruction

Provide all the information a user may need without overburdening them with content. This is easier said than done, since you cannot predict what a particular user will need.

This often means putting more detail into our docs than we expect many users require. While you might worry this is distracting (and you might be right, depending on the user), the bigger concern is that it will instruct unnecessary effort.

Overinstruction strategy should always be clear in its aim, and advanced users should be able to quickly determine if an instruction applies to them, or if they can skip it. For me, this means using consistent patterns in examples, so users can quickly determine if there is something distinct about an example that they need to pay close attention to.

It can be annoying to encounter seemingly trivial or assumed steps spelled out in painstaking detail, but we also all know how frustrating it can be when docs assume more knowledge (or interest and ingenuity) than we actually possess.

Breadcrumbs and Circles in Docs

Documentations should never have dead ends. Every topic is related to at least one other. This does not mean we have to mandate an order to our docs; remember, reference docs are used, not read. We are curators with tremendous power over the common source and diverse expressions of our exhibits.

A way back is as important as a way through.

Optimally, the entrance to and exit from a given topic are not the same parent or sibling topic, though this can happen, especially in early iterations. Zooming in to write or edit individual topics, or to work on a family of topics, should be complemented by frequent zoom-out sessions to establish a project-wide perspective. With the macro lens affixed, map out your content, try to detect holes and missing links, then fill them.

Chapter 4. Writing for Use(rs)

You'll never know how to reach your audience if you don't first get to know their challenges the way good UI/UX developers do. If your audience consists of developers themselves, you've got your work cut out for you.

Use the Product

This is the obvious one. I have no shocking statistics on the portion of tech writers who do not actually use outside of work the product they document for work, but I bet it's a bunch of us.

This section is not called "*master* the product" for good reason: mastery is rarely the point. Master users are abnormal, and unless you've been specifically tasked with appealing to outliers, you may glean little advantage from knowing how a master user engages with your product. The more complicated a product, the more likely there is no one way to master it, and the more likely there is a broad range of user expertise you must cater to.

We see this distinction clearly in the difference between a product's official reference guide/user manual and the various aftermarket books that pop up to explore the product in different ways. From *...for Dummies* coverage to the O'Reilly treatment, unofficial documentarians rule the specialized regions, often with direct support from the product team. And this is likely as it should be, at least until you make some room to get daring with your docs.

How I Relate

My company makes enterprise IT ops software. Since I am not an IT ops agent monitoring a massive datacenter, I have little call to use our software the way our end users do. Even as we use it to monitor our own company infrastructure, that's not in my job description. But that doesn't mean I cannot seek out the perspective and pain points of users.

A major part of my actual audience is made up of Customer Success engineers and Field Sales engineers — coworkers of mine who service customers with high-touch assistance. I talk to them regularly about how they use the docs, and about how end users engage with the docs.

It also just cannot be critical that we be able to "master" the product we document. Take for example highly complex products like Photoshop or Drupal or Salesforce. Proficiency with these products merely means you know how to figure out how to achieve things with them; of course it does not imply you already know everything there is to know, every permutation of every configuration or possible use case.

This extends to consumer products as well. I'd be willing to wager some of the video game docs you've seen were not produced exclusively by tech writers who had personally reached and killed the game's

final boss. If you've ever read a video game walkthrough, however, you'll see that technical writing can in fact demonstrate a breathtaking level of mastery.

For enterprise or other business software products, we tend to see thoroughness in references, but not necessarily complete saturation in instructions. The more divergent ways a product may be used, the more official documentation needs to teach users core elements of "how to fish".

If you're documenting a scripting language or a large, complex API, forget about it — you'll still need to check your own docs at least once in a while, which is just fine. Virtually all major contemporary programming languages arguably fit the bill of *too complex to truly master*. This is mainly because languages develop methodological tribes around frameworks, integrations, conventions, and architectures. In this way a thriving software language tends to grow exponentially, led in different directions by fresh thinkers with varying needs and preferences.

Moreover, the risk of thinking fluently in any language is forgetting what it was like to find that language strange and off-putting. The best instructors readily empathize with the uncomfortable, disorienting experience of learning an unfamiliar and possibly unintuitive system can be for the student. The people who find your product's interface intuitive may never look to your docs; they're not your key audience. Masters and poseurs have a tendency to kick over the ladder once they're up, leaving newbies behind. Technical writers cannot afford to leave anyone behind.

Ski School vs Ski Patrol

Your product is a mountain. Your users are skiers. Technical writers (you) are the ski instructors. Customer support agents are ski patrol (professionals who respond to slopeside emergencies).

Your instructions can take an intermediate skier out onto an experts-only trail, but they had better get that skier back down in one piece. The ski patrol will rightly be upset if they find some mangled student of yours abandoned in the trees. Whether you left them behind or gave them the wrong idea and sent them off a cliff, it's your fault they crashed. Take care of your users in such a way that Customer Support never has to clean up after you.

Docs == Knowledge

One of the biggest mistakes a tech writer can make is to confuse oneself with the Reference Manual one maintains. What its author *knows* and what a doc actually expresses are two decidedly distinct things.

Throw in the third set of knowables (objective reality), and you get the ingredients for a graphical chuckle:

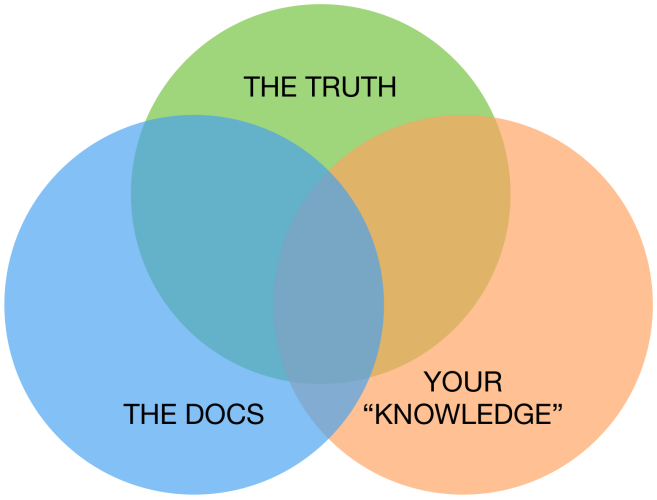

Figure 5. Venn Diagram of Knowledge–Docs–Truth Trinity

That’s *funny haha* (worth a few emoticon clicks when you post it to Slack), but the reality feels a little less comfortable, if a bit more familiar:

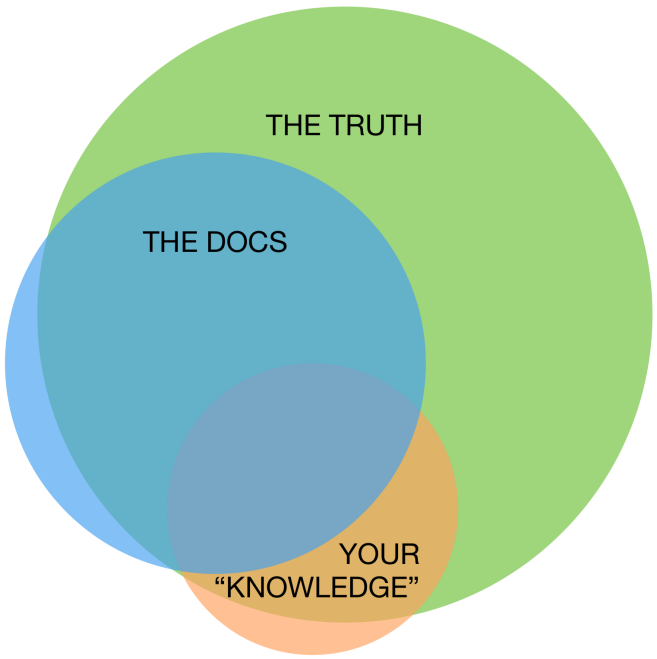

Figure 6. Venn Diagram of Real Knowledge-Docs-Truth Trinity

The proportions may vary depending on all the obvious variables. The question is, do you know the shape of your Venn diagram?

Of Course You Should Master the Product

All the above bullshit aside, hell yes master the foundation right out from under that product. Live, breathe, and love the product.

I do believe many documentation projects would suffer a net loss overall in terms of quality if they were only documented by masters of that product. However, I have also followed along with a 500-page unofficial user guide written by a master of a product and been bowled over by the sheer relevance and accuracy of that massive tome.

I am one of the few people at my company who looks under the hood at pretty much every aspect of our product. In that regard, I am a “master” of the product. That said, I don’t know any part of the product as well as *any* of the engineers who works regularly on that component.

Learn Users’ Motivations

Most users aren’t using your docs to get a clinical overview of the product. They’ve come because they’re trying to get something done. Either they’re habitual docs users or they’re looking for a specific solution. In any case, your users' objectives are critical.

Use Competitors’ Products

I used to think only product designers and Sales and Marketing needed to be familiar with the competition. Then I figured out all our customers were expecting our product to work like our dominant competitor’s offering. The good news is that I can actually explore the world customers think they’re getting when they first crack open our product. So informed, the theory is I can better ease the user’s transition from expectation to satisfaction, all without them so much as passing through the panic stage (at least so long as they open the docs on time).

By using competitors’ products, you will learn about your own users. Now you are visualizing your users’ expectations.

Does your product do something you call *parsing*, which your chief competitor has called *processing* for 10 years? You may have good reason for it, and users may not care. What they may need is to see the word *processing* somewhere in the sentence that introduces your *parsing* functionality.

Does your product call for **File menu:[>] Export** for an operation some competitors use **File menu:[>] Save** to perform? Again, you can be agnostic about which is better (I don’t even know what your product does), or you can be downright opposed to abusing “Export” the way you feel your engineers have. All that matters for your docs is that you intervene gracefully where you expect new users might reflexively look, pointing them to the new home of the functionality they desire. (This is good advice to pass on to product developers, as well, if quality assurance is a team-wide effort at your shop.)

Most importantly, use your competitors' docs so you will know what it will take to establish a clear differentiator out of documentation. What will it take to get your docs decisively in the Pros column?

The Splunk DM's Shield

In October 2015, the first time I met Rocana's then-new Director of Product in person, he threw down a gauntlet. He wanted me to see what our direct competition was capable of, and I was humbled by the artifact he handed me.

Unfolding in my hands on glossy card-stock, a splendrous 6-panel, 25"x11" reference spread made up of notations and tables, together conveying much of what is most needed by our competitor's users throughout the day. That user is also our intended user: an ITOps engineer or analyst.

Having been a hardcore nerd in my youth, I immediately notice this documentation concept is reminiscent of a Dungeon Master's shield, a global quick-reference for the referee of a Advanced Dungeons and Dragons adventure.

I know some of you are judging me right now. Laugh it up — my hunch proved right! Splunk senior technical writer Matt Ness gave a talk in which he explained the AD&D-derived origins of this ingenious document. But more pertinent to our purpose here, I beheld a form of documentation I will forever kick myself for not having done first. Since I couldn't deny the genius of the DM shield (including its marketing angle), it put me on notice that docs can make a real difference in our product's specific space.

I think I'm good at my job, and I hope Rocana edges Splunk out of our corner of the market someday, but I'm glad to know I'm up against a formidable counterpart. The temptation to copy Ness's innovation is great, especially when I consider the challenge of single-sourcing the whole thing right out of the product code repo. For now, the Rocana shield waits; I'm headed in other directions.

Lean Documentation

It's hard to believe, but there are still startups today that do not emphasize a lean UX approach. There can be virtually no doubt that early product testing and rapid iteration in response to validated and invalidated learning are critical; the days of inflicting a product blindly on a market are coming to a much-deserved end.

Docs are UX

Technical writers have to get on this train, even if that means hopping it without a ticket. Docs should get tested alongside the product, including the evaluation of multiple approaches.

Do you know the value of a diagram or illustration?

How about the efficacy of a “quickstart” version of your installation instructions?

Unlike in software development, almost every bit of a technical writer’s work is UX oriented. Many users will never see our docs, but almost every word and image in our docs is part of the product’s user experience.

user experience

developer experience

Everything is UX

While not as common as overlooking the fact that docs are pure UX, many product managers and developers do not recognize how many interfaces a complex enterprise product may have. Anywhere users interact with the product is an interface. This means command lines, installers, configuration files, plugins. Good docs affect the user’s *experience* of every UI, and there is likely always a better way to reach the user in a given context.

Docs User Testing

Test Systematically

Measure Traction

Add Value

You’ve been told your documentation should make yourself invaluable to your employers. You’ve been told that this should generally be done by heading off support requests from customers, or at least by providing Customer Support with a great reference for handling such requests from users who ignore the docs.

However, the truth is you should be providing a margin of value directly to customers. For the end user, the technical writer alone — more than the marketer or the UI devs — can provide layers of appreciation for the product that cannot be articulated during a transaction. Away from clicking and purchasing, the end user is most open to appreciate the application either (1) right before they begin using it or (2) when they realize a new level of the product’s complexity. That is where solid technical writing comes in.

Good docs are ready with a sweeping overview for the intimidated new user. For complex products or those requiring contextualization, some users head straight to the manual. This means they’re not fully distracted by the developers' and designers' crafty handiwork. Instead, *you* are more or less alone with the user. Not to take ourselves too seriously, but good docs help close transactions and calls to action.

Good docs help experienced users avert or curtail frustration. Coming to the rescue is fun. I realize you don’t even get to be there for the adventure, and it’s not like you’re pulling anyone from a burning

vehicle — I didn't say we're heroes. But your docs may save countless hours of user aggravation, and that's not nothing.

And just imagine if you can introduce the product or prevent user frustration while enriching their experience with humor or usage-enhancing context.

Part Two: Coding

DocOps is about maximizing diversity on both ends of the content continuum: an open-ended potential array of content contributors, with content source kept in one distributed repository, and delivered in myriad forms and formats, as needed, to all the right endpoints.

Here we begin a slightly more formal exploration of DocOps as a development and technical writing subspecialty. More than establishing rules, these chapters lay the groundwork for solving complex documentation challenges in forward-thinking ways.

Chapter 5. Source Coding

Codewriting refers to the different "codes" we use to communicate technical concepts, and the way we look at our overall approach to documenting software. Codewriting is writing about software in a way that involves coding alongside the product, even if most of your source code is strings and prose.

There's no getting around it: codewriting requires deep knowledge of the specific product, and a functional knowledge of software more broadly. Codewriters fully understand just about everything their product *does*, but they also have a firm grasp of how it is developed, by whom, using what assets and tools, in what environment. Great codewriters understand the product build process, agile project management concepts, and developer interfaces (such as APIs and IDEs). Even quality assurance, testing, and automation are on the codewriter's radar.

Thinking like a coder means looking at the product from the inside out. I don't recommend this as a constant mindset; the end user's perspective is far too critical to give way to the developer's vantage. But understanding the way devs look at code can be empowering, so let's take a minute to brush up on (or brush against) basic computer science. Don't worry, I have no ability to go beyond the very surface, but there is plenty of insight in the basics of programming.

Code Abstraction

Unless you can read machine code (you cannot), it does not matter how great of a software engineer you may be: humans require code abstraction in order to interface with computers. This goes for programming as well as for using programs. Even if you make a product for software developers, they want to work through an effective interface, be it a command-line interface (CLI), a graphical user interface (GUI), or an application programming interface (API). All those *I*s are effectively sets of "fields", "endpoints", "commands", "arguments", or "hooks", if not a full-blown scripting language. With such interfaces, their human brains can train other pieces of software to interact with your product.

Let's take a look at an example of code abstraction in rich text. Say we're trying to create a section title for a document, followed by some basic text.

Example 1. Code Abstraction — Final Render

Code Abstraction

What do we mean by *code abstraction*?

As HTML under the hood, that should look something like this:

Code Abstraction — Markdown converted to HTML

```
<h2>Code Abstraction</h2>
<p>What do we mean by <em>code abstraction</em>?</p>
```

And because all those extra tags are not easy to type and only serve to limit our output options, we sourced this in lightweight markup, which converted to the HTML you see above.

Code Abstraction — Markdown lightweight markup source

```
# Code Abstraction

What do we mean by *code abstraction*?
```

Behind those recognizable characters, where we'll never look, are bytes representing unicode symbols. I'm not really sure what is behind those.

For the three versions that concern us, the difference isn't massive, but I can attest it took longer to type the HTML example, and I introduced a typo the first time through. And I must say, even after twenty years of hitting < hundreds of times a day, it's still my least favorite key.

In the same way a WYSIWYG editor is an abstraction of HTML (or other rich-text XML behind it), lightweight markup languages are an abstraction of that same complexity. AsciiDoc was made to generate DocBook XML output. Tech writing guru Tom Johnson has written about writing in Markdown to generate DITA or HTML. It may seem like an arbitrary line to draw — *I want to see lightweight markup, but I don't want to see XML or bytecode* — but that's my Goldilocks zone.

Abstraction as Transformation

Most products and development environments involve much more complex forms of abstraction. Even a website typically adds more layers, such as the cascading style sheets that provide layout form, fonts, colors — pretty much everything but the content itself, down to all those boring HTML elements. A large enterprise application can have numerous interfaces opening up countless avenues of source code giving way to machine code, which in turn forms numerous portals for developer, administrator, and end-user engagement with the product.

All that is too big and frankly too abstract to grapple with, so let's get back to a simpler example. What if our product interface is bare bones? Let's say we're a new startup, and we're hoping early adopters will validate our innovative tool at an early stage. Maybe all we provide these early users is a command-line tool.

It may not feel like it when you're struggling at a command prompt, but a CLI is a layer of user-friendly abstraction, too. You think commands and options are hard to keep track of, but they're saving you from having to write and run scripts every time you want to interact with a product. This may be difficult to appreciate, but engineers understand the advantage, as will anyone who learns to explore the

Codewriting GitHub repo.

When I started writing this book, I knew I'd want to make it extremely easy to publish your preferred version of this document, as well as show off some of the powers of DocOps tools. I also wanted to reduce the number of steps it might take someone to build the book, which was kind of a pain in the butt at first.

For these reasons, I developed a tool called LiquiDoc, which provides a command-line interface. (If you built this book from source, you've already used it.) This LiquiDoc CLI abstracts a bunch of awkward logic you can learn about later, and that's the whole story of why I coded it. The script was so handy, I made a more robust version of it for work, and we released it as a Ruby gem, as well as opening up the source code.

"Code abstraction" is a way of saying *a simplified interface to underlying complication and functionality*. The *source code* developers write in is always an abstraction of underlying code. Executable code is compiled or interpreted into another form, closer to being able to instruct the microprocessor to perform specific operations.

When the developer is "done" with it (ready to test or package), source code becomes more complicated in order to interact with underlying layers — perhaps a some libraries and precompiled dependencies, a runtime environment, the operating system, and then the microprocessor. From all of this, your effort on that last layer yields a product that obfuscates all of the above, so the end user only sees what they need to — data and functionality are abstracted. Just like our Markdown above gets uglier (HTML) in order that it may become more elegant (the browser display). Sometimes specificity, performance, or functionality are sacrificed during abstraction, but a successful abstraction process creates at least a nominally new interface into a human-unfriendlier layer.

Docs as Merciful Abstraction

In this sense, user instructions and other documentation are an abstraction as well. Unless you're writing docs for hardcore engineers (and not always even then, I imagine), there will be things you leave out in order not to increase the obscurity or complexity of your product.

It can be tempting, especially on an open source product, to be fully transparent, even with internal notes, specifications, and other "work product" that often takes place in private spaces. Unfiltered data gives creative and motivated users the potential to solve their own problems. But consider all that potentially confusing if not contradictory or even hazardous information that would get out. Ranging from user frustration to misinformation to potential liability, there is definitely such a thing as TMI in product documentation, even when your product is open source and highly extensible by design.

At a certain point, the sheer volume of docs achieves sharply diminishing returns, especially considering all released docs must be maintained. A project can easily have more "documentation" files than it has originally contributed programmatic files, when you count all the meeting notes, task-management issues, support tickets, and whiteboard wars. While adding only the most marginal value for edge-case

users, too much detail can produce incalculable opportunities for typical users to get lost or discover bad information. If your whole team operates in a public repo and task-management solution, maybe it's okay that interested users can find what they need. But there's a difference between *exposing* and actually *releasing* documentation.

We'll return to the discussion of curating content and managing internal docs. For now, I wanted to demonstrate how **docs are at once user interface and abstraction** — just like the product is.

Like a Bot, But Way Better (For Now)

Consider how all of your duties at work make *you* an abstraction of sorts. If you're a technical writer, *you are a self-aware, actively learning black box* — one that can be given commands such as "Document this new feature". Once you're oriented at your job, you no longer require detailed instructions of how that gets done, if you ever even got them (fat chance). You accept input; magic happens; docs come out.

Let's take a closer look:

You are provided

1. an environment (some kind if system for documenting a product);
2. intelligent input data (knowledge from SMEs);
3. an interface to explore (the product); and
4. resources for self-directed research (source code, specs, Google).

From which you are expected to produce

1. refactored information in a format that humans can better understand (knowledge transformation);
2. a central store of metadata about the product, including version tracking, changelog, dependency and licensing requirements, installation prerequisites, etc. (information architecture);
3. iterative improvement of your output with each cycle (learning);
4. iterative improvement of the documentation platform and the development-documentation *framework/workflow* (recursive tooling);
5. iterative improvement of yourself, meaning *iterative advancement of you as a tool*, in ways that translate to other products and even contribute across departments (recursive self-improvement).

These are the very attributes of the artificial intelligence program that will eventually replace you. Put more optimistically: *You will be in demand by forward-thinking companies at least until a machine can do all of the above*.

Well, if you are a typical technical writer, machines only have to achieve competency in the first two or three procedures before they become extremely attractive. DocOps is about how not to be a typical

technical writer.

DocOps.do()

So what is the set of functions a solid codewriter carries out that make us (think we're) so special? We fancy ourselves able to wear all the necessary hats to keep the docs in ship shape, no matter the pressures and obstacles we encounter. This is partly because we're skilled, but it's mainly because we're resourceful. Either way, we exhibit more general intelligence, and at least marginally greater passion for our profession, than any computer program yet developed.

But let's pretend for a moment that we're just a fancy piece of AI software that can do a whole lot based on minimal commands.

Calling carryOut() as an abstract function

```
// Call our function
carryOut(user, act, target)
```

Let's say this code *calls* a function named `carryOut()`, to which we are supplying arguments: the *values* of variables represented by `act`, `user`, and `target`. Those arguments answer three questions the `carryOut()` function is just dying to know; we're assured `carryOut()` can take it from there.

We can't see those values in this snippet, because they have been abstracted into *variables* (`user`, `act`, `target`) so the code we see above can be reused in different environments with different results.

To keep it simple, let's say we had previously set the values of those three variables like so:

Setting inputs for carryOut()

```
// Set our variables
user   = getObject("Wendy")
act    = "integrate-the-docs"
target = getObject("product/repo")
// Call our function
carryOut(user, act, target)
```

The `carryOut()` function sees the *evaluations* of these arguments (user, act, and target); it interprets their values. As we'll see, this function expects some of these values to come in the form more complex abstractions called objects. The `getObject()` function likely performs a database query or some other means of gathering contextual data and formatting such that other functions may manipulate it as needed.

Maybe you've figured out that this function instructs Wendy to integrate the documentation into the product repo.

In our abstract digital scenario, someone prepared the function `carryOut()`, which the code above is

merely invoking. The establishment of that function (so that it can be invoked) was another abstraction.

Similarly, the procedures you carry out to get your real-life work done are far more complex than their names imply. The order to *diagram* a relationship between elements of a product suggests all kinds of specific wishes, which you must either infer, learn, or make up — often some combination of all three. What you do not require is for some manager to point at your screen to tell you which programs to use, which menu items to click or shortcut keys to press, where to save the file in what format, and so on. A framework for this might have been established at some point, but you execute all day long within those parameters, without anyone having to reinstruct you.

Let's peel back one more layer to see just what the `carryOut()` function's definition actually looks like. As in most software "eyeballing", the result is mildly enlightening but largely disappointing. That's abstraction for you.

Defining carryOut() as a function

```
define function carryOut(subject, action, target) { ①
  validateParam(value: subject, type: "subject") ②
  validateParam(value: action, type: "action")
  validateParam(value: target, type: "target")
  if find(action, subject.skillset) == false { ③
    willLearn = canLearn(subject, action) ④
    if willLearn == false { ⑤
      result.status = "fail"
      result.message = "That is too hard for me :-("
      result.target = target
    } elseif willLearn == true { ⑥
      learning = goLearn(subject, action)
      result.status = "pending"
      result.message = "I was already just learning that. ${learning.status_message}"
      result.target = target
    }
  } else if find(action, subject.skillset) == true { ⑦
    action = doAction(action, subject, target) ⑧
    result.status = action.status
    result.message = "I can already do that. Here's my status: ${action.status_message}"
    result.target = getProfile.target
  }
  return result ⑨
}
```

This example is not actual JavaScript, but rather a phony language intended to most simply illustrate the concept of abstraction in code.

① The first line establishes the function's name and expected arguments.

② A function named `validateParam()` is called against each parameter passed to the function; this abstraction represents functionality we cannot see here, which ensures the variables passed to this function exist and are of the proper type.

③ This conditional first determines what to do if the required action is not in the subject's skillset; it uses the `find()` function, an abstraction for searching an array for the value of the action variable.

④ The function `canLearn()` is an abstraction that figures out whether the Wendy object can learn the action skill. Who knows what it's doing under the hood, but we're naming its result `willLearn` because that's what its value will indicate: our subject *will learn* this skill if possible.

⑤ If the subject is deemed unteachable, the function fails and we prepare to return the same object for target that we got from our original `getObject(target)` call. (The function "breaks" here, and we skip to **9**.)

⑥ Finally some good news! We can report that the work is "pending", the `goLearn()` function is getting us knowledge, and our specific message. Here this partly includes information derived from the `goLearn()` call itself, the `learning.status_message` variable tokenized with `${}` notation.

⑦ Harkening back a couple steps to the question of our existing skillset, if this action *was* found in our skillset, we'd have skipped right to this step and be off to the races.

⑧ Unfortunately, `doAction()` is one hell of an abstraction. It's a good bet that all of the practical, fun work associated with the `carryOut()` function call is taking place under the hood of `doAction()`. Or maybe, like `carryOut()` itself, `doAction()` does some overhead work but passes on the heavy lifting to some machine-learning API halfway around the world. We simply cannot tell from here. Abstraction can be a cruel warlock.

⑨ Finally, we *return* the result of our function back to the calling code.

In other words, all the real magic takes place on an even deeper level than this code exposes. The `doAction()` function is Wendy pulling out all the stops to learn a technique to produce a result that meets a contract and passes a whole battery of unit tests.

But Wendy is resourceful, so Wendy's got this.

Docs as Abstraction

Good documentation is another layer of abstraction added to the product it covers. It gives users (and potential customers) a view of the product that uses text and images to abstract all of that complicated software into a page or a table or a bulleted list that meets users where they're able to interface.

At its core, software technical writing is creating the ideal abstraction layer between users and the product. Like a good UI, good documentation foresees the user's needs and has the right information one click or one page turn away. Even better than a well-written function, good documentation provides context inline, even as it references and links to heavy-duty references or further background.

Also like a good function or method, docs are not meant to be read, but rather *executed*, which we considered in Docs.

Writers as Contributors

If documentation is part of the product codebase, you are working like a developer every time you make a commit to the product repo.

If you're coming to codewriing from tech writing, maybe you haven't looked at your work this way in the past. Especially in the open source community, but really now throughout the broader software engineering field where collaboration is easier and more critical than ever, developers tend to think in terms of making "commits" to projects. You'll see open source participants referred to as "contributors" and "commiters", including tech writers who work in the same repository.

I suspect this stems partly from the literal sense in which contributors are usually "committing" code to the source. In perhaps most cases today, technical writers do not even use Git or another source-control tool, let alone are they working inside the product code repository.

For now, it's more important to change mindsets. Docs are part of the product, and tech writers are contributors — this is true even if they work exclusively in a Wiki or a conventional content management system well outside the product codebase. Developers should not hold these terms as their exclusive domain even when they alone make commits to the source code. Any devs doing this should consider remedying the dichotomy by bringing tech writers deeper into the Engineering fold.

Subject-Oriented Writing

You've probably at least heard of "object oriented" programming (OO). It's a general category expressing the way a development language/framework is structured around acting on data objects, as well as the approach to developing software in that environment. In OO, engineers can expect various elements of certain kinds to behave in specific, predefined ways. Whether any given developer likes coding in OO is another matter; there is a closed, consistent logic to the approach.

Technical writers have a much less exact task, which I argue can sometimes be harder to get just right. We aren't writing code to act on objects inside a closed system. Very few of our sentences will throw syntax errors that require us to rewrite if we wish to proceed with publishing our docs. Anyone who has tried to enforce grammar or style with a linter knows good writing can upset static observers; even *technical* writing is too artistic for such stringency.

We also are the primary interpreter of input data, the flawed humans that we are. We cannot program a system of explanation. And unlike an operating system or an API, if our audience of *real humans* cannot make use of our part of the product, they will not throw what developers call a "graceful error". They may just throw a fit.

Subject-oriented writing is not what it may sound like. I'm not using *subject* in the sense of *topic*; I am using it as the opposite of *object*.

If your audience is people, you cannot treat them as objects, the way a programmer can treat an API.

People are extremely complicated, meat-based computers that do not necessarily appreciate your language let alone your preferred syntax, yet you are expected to write for all of them in their various moods and modes. In truth, users often reject your medium before they even give your implementation a chance. (“I learn experientially; manuals are just for reference, at most.”)

We can blame the audience all we want for not appreciating our hard work, but the truth is, we write material pretty much nobody truly wants to read. The programmer never has to wonder this about her immediate audience, because it isn’t meat and thus has no sense of subjectivity, neither preferences nor opinions nor attitudes. If the computer can interpret a language the programmer can write, the programmer never has to be concerned that the computer will be uninterested in compiling and running her software.

The programmer need not wonder if the program may try to skip some steps depending on how close it is to lunch.

The programmer does not have to name variables, objects, or methods creatively, just consistently. She needn’t worry that her choice of a `switch` statement over an `if` statement may clash with Marketing’s preferences — those most impacted by her choices are usually close at hand. And she sure as hell shouldn’t have to restate anything in her source code in order for the computer to “get it”.

Yes, Text Can Be Harder Than Code

Let me walk back that section title a little: It’s rarely harder to *describe in text* the most complex software concept as it is to *program* anything complicated in code. Also rare is the case that someone skilled at writing software is somehow unable to string words together coherently in their native alphabet (though I have seen this). Conversely, most people who are adept at their mother tongue assume themselves unable to code their way out of a “Hello World” exercise. Clearly, most of us think code is harder than text.

Nevertheless, making a user manual *work* just as well as the software it accompanies can be at least as challenging as making the software itself, especially given the limited array of tools and dependencies at the documentarian’s disposal. Product managers are less likely to give the kind of strict requirements and resources to docs that they convey to developers. This can leave documentarians feeling rudderless.

The difference is, people tend to overlook bad documentation if the product is good; in fact, a well-designed and executed product can reduce or eliminate the need for exceptional docs. Great documentation makes using a product better — less frustrating, more rewarding. If the product itself is great, the docs have to be that much better in order to provide a added value to the end user.

A Docs Paradox

I have no fear that interfaces will suddenly, all at once cease to require auxiliary documentation, but I have to admit a product is likely better to the extent its interfaces reduce the need for documentation.

This is kind of a chilly realization for somebody staking his future on being an innovator in the technical docs space. Even at Rocana, I find myself trying to innovate new ways to enhance the product UI with DocOps tooling, and I offer heavy feedback to the front-end and back-end teams as I explore and painstakingly document their interfaces (the web GUI and some CLI tools, respectively).

I look forward to a time when our product is so mature, the interfaces so intuitive, and our reference docs so well-maintained, that I'm able to spend most of my hours providing context and a sense of purpose to the product through use cases, knowledge base articles, advanced diagrams, and any other innovative means of expression or delivery. I want to get into users' heads and have answers ready as questions reach their conscious minds.

Documentarians who produce poor docs often get away with it, even in this classic *You had one job…* scenario. If you know a tech writer who says her job is easy, you probably know someone who is simply unchallenged at work. Failing without anybody noticing is not the same as succeeding, even if it pays just as well.

Exemplary technical documentation results from mindfulness of process and constantly addressing pain points and bottlenecks along the chain. Whoever finds such a thing "easy" should have written this book in their spare time by now, saving me the trouble; I have found the job anything but easy.

Throughout this book, we highlight some of the reasons for the quality divergence between a given product's source code and its documentation, but one of the key factors is this difference in audience. If you're a developer trying to correct the way you're instructing *software* to behave, you get immediate feedback on critical errors when you run tests, if not as you write. The coder's audience is specially crafted to cry out, hopefully in clear, informative ways, when the coder screws up — long before users get their hands on the product that source code eventually becomes. In fact, a good developer takes great care to maintain an environment tailored to surfacing bugs before anyone from Quality Assurance sees them, let alone a customer or client.

If you've been wondering all this time why I yearn to do software documentation the way coders do software development, let the previous paragraph serve as illustration. We can't change our audience, those meat-based subjects who keep our lights on, but we can make our relationship to users better align with developers' relationship to users. That is, like coders, documentarians can enjoy environments that enhance and provide feedback on our output.

Chapter 6. Coding Content

I forgive you if you gathered from the previous chapter that I somehow diminish the relevance of the *developer* to their eventual human audience: the user. This is, after all, the very same user that is the tech writer's primary audience.

The truth is, coders are always adopting new ways of ensuring their output meets users' needs and expectations. Good developers know they are *not* just finessing code for a digital audience that perfectly interprets it on behalf of the end user. Those end-user meat computers we all serve are still going to interact with some kind of UI the developer's source code ultimately generates, and interactive systems are hard to get right.

Content Development

Product developers have spawned robust systems for user testing and feedback during all phases, from prototyping to post-shipping, all in order to ensure users get the most out of those developers' work. This may not be something we can precisely reproduce with early docs drafts or iterations, given the nature of our output and the potential for confusing real users. But clever observers of product managers' latest UX learning techniques have begun applying such "lean/agile" lessons to docs.

This way the DocOps discipline can eventually establish processes and tooling to significantly reduce documentation bugs — places where we get it wrong, or where our output falls short of expectations.

In the meantime, however, with even the best writing tools, there is no automated feedback mechanism that will flag a statement that disagrees with reality.

Good documentation requires:

Pre-release

- reliable **research**
- clear, organized **writing/diagramming**
- thorough, interested **review** (for accuracy)
- frank **editing** (for clarity)
- systematic **testing**

Post-release

- periodic **reassessment** (pertinence review)
- systematic **retesting** (regression)
- pruning, editing, and rewriting as needed

Code bugs suck, but well-tooled development environments may discover dozens of errors for every bug uncovered later by QA testers and end users. This has become a generally accepted responsibility of programmers.

Reducing bugs in documentation is at least as important but poorly assisted. Text editors and word processors have no idea about your context. They don't know if you're writing about a fantastical land of elves and unicorns or how to program a robot; they certainly don't grok the code your coworkers write, and they can't really interface with it.

Dev vs Docs: the Environmental Advantage

When we discuss the concept of "DocOps", we'll delve farther into this analogy, but for now consider the tools developers have made available for themselves.

Programmers use IDEs, which are code editors that live in the context of their development environments. An IDE basically enables testing the product in real time, even as the developer works her magic.

By contrast, even the best text editors only give sentence-level insight on spelling or grammar. We have to guess how it will be received by the reader.

I don't mean to exaggerate the power of the programmer's IDE, either; it definitely does not tell her if her software will do what the user expects. But in a properly configured environment, the programmer has a lot of advantages when it comes to finding bugs before they ship.

This parallel is a good illustration of the opportunities ahead for thinking in a DocOps mindset. When will tech writers have what we need to predict how our work will be used and received?

We'll explore this frontier topic in great depth in Part Five's Integrated Documentation Environment.

An unforgiving but immediately responsive silicone audience is far and away more desirable than any meat-based audience; the latter are sure to be fickle, distant, even ornery. Your organically eyeballed reader will move on with life; a REST API will wait eternally for a developer to send a properly formatted request. Don't let the coders you're surrounded by convince you otherwise. They think the reason you exist is that their time is more valuable spent writing in programming languages. Prove you're there to abstract their sparsely commented code and poor English notes into a more "human readable format"; that should shut them up.

The programmer's dirty secret is that she may spend innumerable hours setting up the IDE and broader dev environment, but eventually teams figure out the best environments and standardize their setup, automating and preconfiguring as much as possible. You can help with this. **Development environment setup** should be fully and effectively documented for new hires, as well as to establish a standard for existing developers to reference and work toward. Help your team document these standardizations collaboratively, and they'll each thank you for it next time they have to set up a new MacBook.

We will further discuss communicating with users in other sections of this book. What we need to appreciate about users for now is the similarity between *writing code without any docs to reference* on one hand, and *writing documentation without understanding users' expectations* on the other hand. Most unfortunately, human audiences don't come with APIs — documented or otherwise.

You're not thinking like a developer until you look at technical writing as *content development*. This isn't the same as writing software; it's more like coding prose. The methodology is what counts, and content development is an approach that incorporates

- source control
- version control
- iteration
- testing
- automation

Way Beyond Code

If tech writers only write and think in code, we're setting ourselves up to be automated.

We will discuss the above described threat much further in [part-5-evolution]; for now I just want to secure your attention. In all honestly, I am pretty confident that I am helping to build a technical documentation infrastructure that may make it trivial for writing bots (coded by real engineers, mind you) to do a tremendous amount of our work for us… or instead of us.

The good news is, what makes a tech writer human is actually valuable to users, and thus valuable to our organizations. Computers are probably a ways off from writing *interesting* docs that provide user-appropriate insights. (Then again, so are most human tech writers.) We'll look at the hair-raising powers of existing AI later on, but the upshot will be the ironic ways such sophisticated and thorough routines do not appreciate abstraction the way we have come to.

When the bots come for those who excel at compiling reference guides, you will have long since climbed to the higher ground of scenario-based documentation. When the bots lay off everyone who writes use

cases, you'll be safely making diagrams and constructing relatable example screenshots. And when they come for the illustrators, you will still be the one who gets why the user wants the software in the first place, and somewhere, someone who appreciates that difference may be hiring.

Every Page a UI

Our job as tech writers, in two words or less, is to *explain interfaces*. (You knew I wasn't going to go for fewer than two words.) I am tempted to say we *explain relationships* (to products), but that would be cheating. The truth is, we think we document products, but it's essentially only in terms of how they relate to their users or their context.

Even if you're dryly listing the technical attributes of a product, you do this so the user can make better use of it. Even if you're merely explaining what the product *does* to some third-party (it *sends a message* or *it spits out confetti*), you are describing the way the product interfaces with something external.

We don't write for gift-buyers (consumer products) or purchasing agents (enterprise), conveying how and why the product will make their spouse or boss happy. Marketing handles content for those audiences by adding yet another layer of abstraction that separates usage from value.

If at every turn you think of the actual *writing* part of your job as explaining interfaces or explaining users' relationship to a product, you may have an easier time looking at a blank screen.

You describe interfaces all day long; it's time to recognize your job involves designing interfaces, as well.

- Reference tables are interfaces.
- Definition lists are interfaces.
- Diagrams are interfaces.
- Indexes, glossaries, admonitions, example code, and of course screenshots — all interfaces.

These tools make up *your interface* to the user; they are how you break through and get information across a barrier, by providing it in formats users will grasp intuitively.

User Manual

Another theme woven throughout this book is the idea that the tech writer's value is in her *insights*. Anyone can write down the steps *they took* to do something. Any engineer can write down the "correct" steps for performing that same task.

The hard part of our job is not describing the technology and its use in a way we can later defend is technically accurate. The hard part of our job is bridging the gap between user's *expectations* and the product. We make the product make sense where it is unintuitive, and we expose relevance at every stage.

Tech writers answer *why*, and we supplement user interfaces with well-placed insights and well-organized references.

You're still thinking about a robot taking your job, aren't you?

If you have not already skipped ahead to Part Five, you are at least distracted by what must have felt like my unexpected forecast of your professional demise in a book of professional advice.

Perhaps you are discomforted by the irony of falling victim to some open source platform this book convinced you to contribute to. Maybe it's the *tick-tock* effect of some AI engine iterating off in the distance somewhere, whirling closer and closer. Or maybe it's the eerie expectation of one day googling a tech writing tool and finding the landing page of some startup promising three steps to integrating their tech-writing API into your company's favorite build tool.

You fully expect to curse my name when you discover the automation guys have a private Git repo called `techwriter-killer-gradle-plugin`. And that's fair.

Please try not to worry. Part Five is actually quite optimistic. If you're reading this, there is an excellent chance you will survive the robot jobocalypse.

Now, I need you to pay attention to this one last bit.

Content development as an approach to technical writing is your reverse User Manual. It is the closest thing you'll ever get to the elusive EUPI — end-user programming interface.

Content development is a *framework* for lean documentation development, including a *workflow strategy* for getting the job done. It offers a basic approach to assessing the gap between your product and its users then collaboratively bridging it with your whole team.

The journey to establishing your User Manual starts with becoming the ultimate user — not just of your own product, but of product documentation more broadly.

Lean Content

Chapter 7. Docs-Code Integration

Getting engineers involved in thinking about DocOps, as well as directly contributing to docs can be a big win; just be sure to avoid pitfalls. This chapter looks at collaborative docs platforms and workflows.

Docs + Code, Sitting in a Tree

Engineers are notoriously averse to writing documentation. Except the many who are not — those who do a diligent job pretty much every time, or who at least ensure technical writers have everything they need to produce good docs. Whatever the case, if you're taking an engineering approach to the docs, engineers may be more likely to approach the docs. This is definitely what I and some others have found so far.

Getting Engineers to Contribute

Your subject-matter experts probably do a fair bit of writing as part of their docs-related tasks. That is, while you might get lots of information during real-time meetings, I bet for the most part you and the engineers whose work you document have found yourselves sharing quite a lot — especially the details — through digital, textual means.

If you're getting everything you need via e-mail, you may be beyond help. More likely, notes come in through a project-management system or a file-management system — often both. There may instead or also be a wiki involved, or perhaps a knowledge-base CMS for internal documentation integrated with Customer Support.

Whatever the system is, if it isn't the product codebase, in or adjacent to the product repository using Git, then it's probably a solid step outside their focus. The IDE and the terminal are the coder's main tools. Engineers spend the bulk of their time between these two applications, and many wish they could spend more time there.

Meet the engineers in their preferred habitat: the development ecosystem.

Engineers and Lightweight Markup

When I talk to technical writers and engineers about lightweight markup languages, I typically get a version of one of following three responses:

1. [Engineers] don't want to learn another language.
2. [Engineers] already use Markdown on GitHub, StackOverflow, and a gazillion other places, so we should standardize around that.
3. AsciiDoc sounds amazing! How soon can we switch to it from [anything else]?

Most of the people reading this have probably worked with more engineers than I have in my career,

much of which was spent hacking away nearly alone, but my experience the last 6 years or so just doesn't bear the first two responses out. The notion that developers balk at learning new languages used *with good reason* seems absurd to me. This response was given to me off-hand by one of the engineers on my current team, but he picked up AsciiDoc gracefully; now he's a fan.

Here is the combined sentiment of 1 and 2 in the words of Hacker News commenter *zmmmmm*:

> Markdown has a lot of faults, but it solves the one single biggest problem in writing documentation that basically overrides every single other consideration: getting people to actually write it. Basically nothing else matters except this.

(This was in response to a posting of Eric Holscher's "Why You Shouldn't Use 'Markdown' for Documentation", which I recommend.)

This commenter wants to have it both ways: the engineer has to write in a format helpful to the tech writer, but engineers can't be bothered to learn a fundamentally superior language. They must stick with the one they're invested in: Markdown, the "language" they likely learned in under ten minutes total and do not consider a real language. Engineers are notoriously against learning. Right?

The truth is, Markdown is not challenging enough for engineers who care about good docs; it is too limited, covering only elements all engineers know and could hand-code in HTML.

Most of the actual elegance of Markdown translates directly or almost directly to AsciiDoc. While there are more use-cases for Markdown, AsciiDoc is growing fast, and I believe it will effectively close the integrations gap, if not gain equivalent popularity among serious developers.

A lot of technical writers think our job is to cater to the most reticent subject-matter experts — the ones we have to drag scribbled notes out of. And I do cater to such developers; I treat them as kind of a separate case, at least until they come around. I don't complain about it, and I'm not sure I even mind it. People contribute differently; it's no big deal.

It's the SMEs who want to contribute more fully or more directly that I build and curate docs solutions for. The ones who recognize we can do great work together get a different kind of attention from me. I don't merely translate their rough notes into something a user can make use of. They write and I enhance, or maybe I just review and copy edit.

These coders who are reluctant writers know I have their backs. They know I'm not going to publish bad writing; I'll punch it up or edit for grammar and clarity. They also know direct contributions are optional. But enough of them are willing to venture into the docs' corner of the codebase and poke around that they can see the value in my methods, and maybe the value in helping me improve my tooling.

Some devs are never going to love contributing directly to user-facing docs, whatever their opinion of your markup choice. Even engineers who hate directly writing docs are typically eager to get you what you need to do it. Then move on and work more with the folks who want to take docs to a new level and

leverage the power of collaborative writing and powerful publishing and delivery options.

Too Many Cooks in the Kitchen

Docs-as-code runs most of the same collaboration risks as programming source code does — namely the dreaded "merge conflict". And these can be extra frustrating when they involve blocks of text, as engineers constantly note while watching me wrestle with rough merges.

A merge conflict comes about when two or more branches make changes to the same section of a file. Resolving a merge conflict entails registering a preference — a process made much easier by GitHub's reconciliation tool.

The far bigger threat of poorly wrangled contributors is that they'll make messes of your filesystem, naming conventions, and the like. Somebody needs to referee the docs directories, and I'm afraid most of the docs-as-code systems lack great (or really any) content-management features of this kind. If you love filesystems and don't mind their limitations, you'll do great. Otherwise, you will find a significant part of your job involves manually enforcing taxonomy, organization, and naming conventions without the help of a CMS.

This is no small task, but it's far from impossible. At scale, it requires more wranglers coordinating with each other. But at a certain scale, you should be able to devote some resources to developing tools that help manage the mess. When you do, please share with the rest of us!

DocOps

Whether you're a tech writer with some interest in coding or a coder who wants to do docs right, somebody is going to have to do some hacking. DevOps rarely involves coding from scratch; devs supporting devs are more likely to be cobbling open source systems together using APIs.

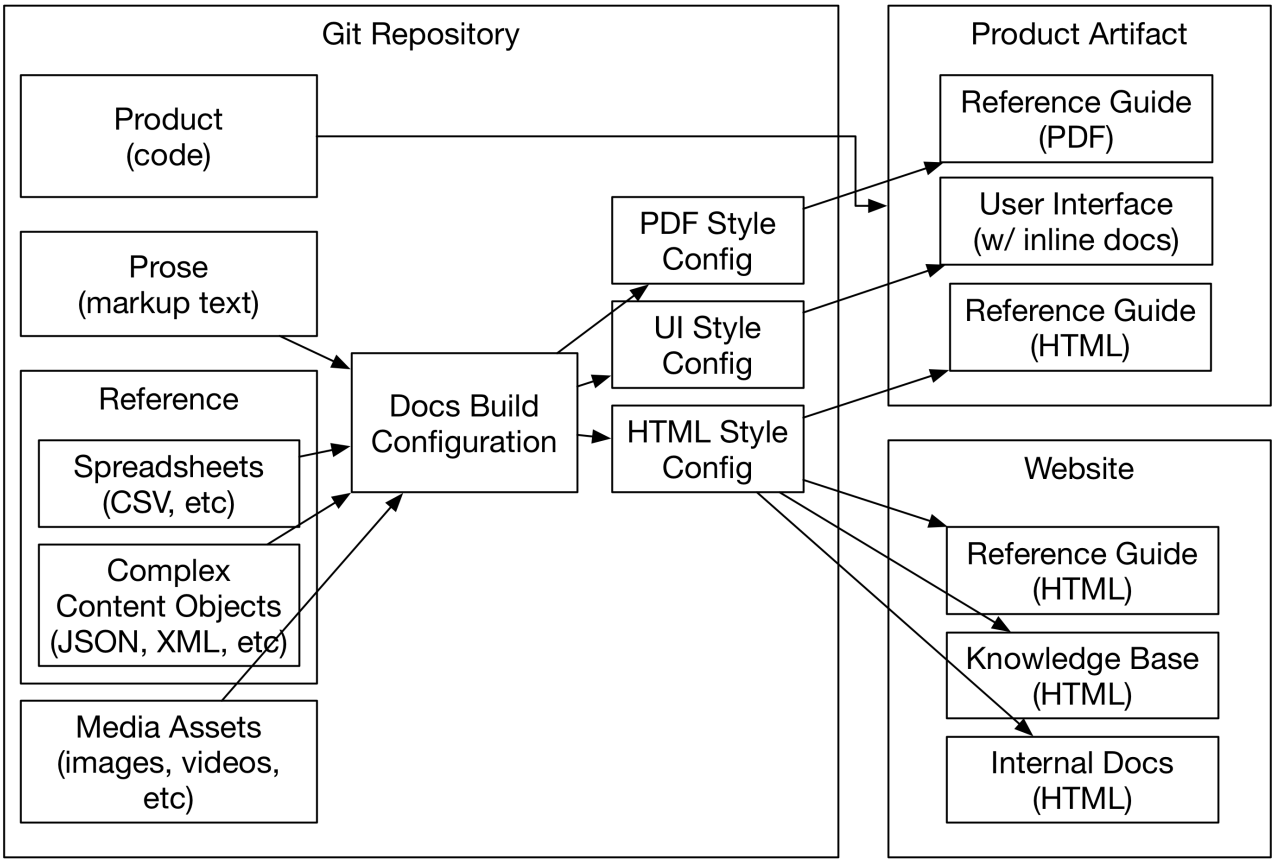

Figure 7. DocOps Overview

If your needs are at all complex, your DocOps system may similarly require a patchwork of various tools to form an appropriate platform. Demands are so various that no tool is going to accomplish complex needs out of the box. Perhaps 90 percent of software projects can be documented appropriately using out-of-the-box solutions. These can still be docs-as-code, flat-file, Git-based frameworks or systems.

But for the rest, where the product or the customers or even just the dev team are somehow peculiar, your documentation environment may be multivariate. The trick is not to sell yourself short by conforming to a simplistic solution, while avoiding the pitfalls of constantly pioneering obscure or massive bespoke projects.

Challenge Engineers with DocOps

Few engineers like technical writing of any kind, and fewer still want to actually document their own work. They may do it, but they'll never call it "fun". But if it looks like you're having a great time putting together a documentation platform using some new open source technology they've heard of but never had an excuse to try out, they may grow interested in helping out when they can spare some time. (I know, I know, but just give them a chance to miraculously discover some bandwidth if you've got something interesting going on.)

This has turned out to be the case at Rocana. Not that I would say any of my engineers is uninterested in docs, but for most, documentation is not the highlight of the product cycle. That said, most of their ears perk up when I talk about building platforms, and several have followed through on offers to help.

The Specter of Internal Docs

A lot of technical writers only work with user-facing product docs, handling none of the developers' internal specifications, resources, diagrams, and write-ups, except maybe using some of them as references after the SMEs are all done creating them.

This is a mistake. Tech writers and docs managers should play an active role in facilitating internal documentation. I argue this for two critical reasons below, but first let's explore what we mean by internal documentation.

```
registries:
  - settings
  - reserved terms
  - ports
  - protocols
  - endpoints
taxonomies:
  - glossary
environments:
  - development
  - testing
  - staging
infrastructure:
  networking:
  machines:
  services:
guides:
  - coding styles
  - conventions
  - IDE configuration
  - project management:
    - agile boards
    - task tickets
```

Perhaps each of these already has a page in your team's wiki. Internal docs-as-code platforms are essentially flat-file wikis, usually built to static sites.

The codebase may be the sexiest part of the Engineering playground, but product quality requires a collaborative, resource-rich work environment. Tools and resources should be at developers' fingertips. For example, there's nothing gorgeous about IDE configurations, but engineers working in a common language using a common codebase and build infrastructure should be sharing their IDE configs. It makes life easier.

Creating and curating an attractive, tidy space where developers can contribute to common resources is a huge value add; making it possible for them to do this without leaving their favorite tools is showing them you care. Make the tooling inclusive enough to encourage product and project managers, QA and DevOps engineers, the CTO, and the whole team to use it and use it properly.

This is also your chance to tutor Git and markup newbies. Help them get comfortable with distributed source control and writing in flat files. These are terrific professional skills; nobody has ever been harmed by having learned Git or a lightweight markup language.

I sadly cannot promise the same for JSON or XML.

Now that you know what I mean by internal docs and how they can be an additional area of collaboration, allow me to offer why this is so valuable.

Reason 1: Internal docs become user docs.

We'll explore this potentiality in [engineering_docs], but it's worth mentioning here as a key motivation for getting involved with internal docs. At the very least, knowing what docs are floating around Engineering should give you some great ideas for supplementary resources users can make use of without too much extra work on your part, even if the material is only of marginal or novel interest. The point is to consider every doc to have potential direct value to outside users, especially when at least some of those users are third-party developers looking to extend your platform.

Reason 2: Internal docs are an inroad.

I got my team interested in collaborative docs-as-code when I built a Jekyll-based website sourced in AsciiDoc flat files for our internal Engineering documentation. The site is password protected, so it includes

- official workplace policies to feature specs,
- testing plans,
- test results,
- language style guides,
- specifications and internal contracts,
- Git and JIRA usage guides, and
- seemingly some new category every month.

Once they saw the value of the system for their own needs, many of the developers started asking if they could provide their SME product content by directly contributing to the source. I haven't dragged anybody to docs-as-code, but they come along one by one, not least because I try to make it look fun.

Docs Maturation Process

Ideally, nearly every internal doc should eventually give way to a user-facing doc.

Rant ahead in 3… 2… 1…

Even with its many qualifiers and wiggle words, I realize that is a bold statement, so let's take a moment with it. I definitely do not mean that every internal document will directly become a user-facing document. (What a nightmare.) But if it was worth creating in the first place, a technical document will eventually at least inform something in the product. The accuracy of product content such as labels, instructions, icons, and other elements includes synchronicity with technical docs and even marketing and legal materials.

It's like that, and that's the way it is

A full-fledged internal-to-external process is called a *recursive unilateral document maturation cycle*, or *RUnDMC*.

This may not be an intuitive concept, but it's been valuable to me. We record institutional knowledge so

that we can perform our jobs better inside that work environment. But with software products — unlike nearly anything else one can produce — the value of ingredients often lasts beyond the current instance of this product. It's not just new designs based on existing designs; we're talking about code and docs written on top of existing product source, forking and modifying it, typically supporting multiple versions at once. New versions will be built on the same matter as the first, on down the line.

Moreover, software is made to allow its insides to be exposed. Think of a good API. Nowadays, good client-server development for web and mobile applications centers around REST APIs. APIs aren't just for third-party extensibility, either. They're best-practice architecture for proprietary/private parts of a company's own product (such as the Web interface or mobile apps) that interact with core/common parts of the backend. Facebook's website and numerous mobile apps use the same API as Tinder programmers and share-on-FB widget developers.

There may be just as much or more of Facebook's code that is kept entirely internal, but you can bet it's developed using the same techniques. Think about that: engineers are writing proprietary code as if they were building an external API, often fully documented and using highly logical user-facing endpoint paths and parameters. Good development technique means interacting with your product the way you would want to see third-party developers and to some extent end users engage with it.

This can and should lead to the slow release of hidden parts of the product, as their utility to outside users begins to catch up with that appreciated by core developers. Private tools get open sourced or otherwise released: developer recipes and hacks can be exposed to third-party developers, metrics and test results can make their way to Marketing and others who can make use of them. In fact sometimes the only thing preventing the release of internal tools to the open-source community is concern over support demands, which is usually a result of poor documentation.

This is all great news for documentarians. All else being in order, more released product documentation is almost always better. Of course, "all else" in this case includes good organization, indexing, and delivery systems, as well as respected security protocols, privacy measures, and regular content audits.

Most importantly, product code and docs code should be handled with care throughout their lifecycle, and as much of it eventually revealed as possible. The best way to identify a piece of code or documentation that has reached the end of its lifecycle is to plan its management from birth onward.

Graceful deprecation is a gift to developers and users alike. We'll discuss it more in Part Four: Managing.

This level of lifecycle anticipation means imbuing internal docs with an extra round of integrity during creation, editing, and upkeep, which may edge them that much closer to release or retirement. Imagine all those benchmark notes taken during hours of rigorous performance testing, workarounds and patches that never quite qualified for official release, sample files used in development, responses to one-off inquiries from Support, Sales, or Marketing. This is the stuff no bot is going to come up with any time soon.

You don't even have to directly monitor all the noise in Engineering, let alone actively curate and

archive the signal; the trick is to set up systems so that all knowledge has a home and a trivial route to it. Optimally, this is a system so consistent with your whole team's existing routine that complaints over any effort needed to maintain it are outweighed by the benefits of maintaining it. It cannot be frustrating or burdensome, and it has to pay off down the line.

Releasing work product can be rewarding

If it doesn't require much extra effort, selectively releasing "work product" can be a rewarding experience for many developers and support staff. Even when just as uncredited as their source code contributions, developers like to see the fruit of their work get exposed and potentially make a difference to someone.

This goes for nearly everyone at any company, all the way to the remote contractor — everybody wants to impact customers. Look to team members you don't work with as often, such as QA testers, automation and DevOps specialists, project managers. Anyone who has information that might help users succeed with or extend the product is a potential source.

If you've encouraged DocOps best practices for internal documentation, lots of internal content should be one edit and review away from releasability.

The key is to make it easy for users to find their way to the right docs as soon as they figure out they don't have the best version available.

For God's Sake, Don't Release It All!

All the above being said, for the love of all that remains unspoilt on this forsaken ball of crusted molten matter, be judicious in what you release.

Release only what you can and will maintain. Keeping track of all released documentation means *having a system in place*. If you do this right, the amount of released material could grow exponentially, with a consistent flow between internal drafts to inclusion in released artifacts. In the process, source material will undergo vetting, deprecation, pruning, distillation, and redaction as needed.

Make no mistake, however; getting this wrong can mean orphaned documents all over the Web and inside search engine indexes and users' browser bookmarks. To some extent, this is just what we deal with; often the best we can do is keep every released document updated and readily available to users.

What Developers Need

Style Guides & Conventions

Workflow Guides

Reference Registries

Drafting Capabilities

Knowledge Bases and Inter-team Sharing

Version Control & Deprecation in Internal Docs

Chapter 8. Coding in Regex

The patterning syntax known as *regular expressions* or *regex* is as notorious for its alleged steep learning curve as for its confirmed utility. Nobody questions the efficiency of this set of scripting rules. Regular expressions can be used not ony be used to find predefined patterns in arbitrary text; regex can extract contextualized data from within those matched patterns, passing it to an application for further manipulation and storage.

The concept of regular expressions is notoriously difficult to learn. I admit I didn't appreciate their potential for the first many years I begrudgingly used them to solve otherwise impossible data conversion and other IT tasks. Appreciating regex can make a world of difference to learning regex.

If you have never or barely used regex, perhaps it is about to become your first coding language. If you've used it before but never taken the time to learn its full potential, this chapter should be helpful as well.

Seasoned regex coders can keep me honest. This book is open source; help me bring the power of this intimidating but mercifully straightforward scripting syntax to struggling documentarians everywhere.

What-ex?

I pronounce the shortened term *REJ-ex*, though I believe I've heard more developers use *REGG-ex*. Or maybe it just strikes me more starkly when I hear it pronounced differently than I say it.

What is Regex?

A regular expression is a way of describing a pattern using common characters in order to search for the described pattern in a body of characters. The target can be pretty much any form of text, such as a conventional document or a flattened dataset. A regex parsing engine uses these expressions to evaluate target texts for matches.

Put differently by the regex-focused site Rexegg, "A regex is a text string that describes a pattern that a regex engine uses in order to find text (or positions) in a body of text, typically for the purposes of validating, finding, replacing, or splitting."

At its simplest application, think of regex as a much smarter version of the `*` or `%` wildcard symbols you've probably used in other contexts. Where a conventional wildcard finds *any and all text*, regex can find shockingly detailed patterns without false negatives.

The string `codewriting.org` is matched by the following regex patterns:

- `[a-zA-Z0-9][a-zA-Z0-9\-]{1,61}[a-zA-Z0-9]\.[a-zA-Z]{2,}`
- `.*@\.[a-z]+`

- `codewriting\.org`

You'll note that all of these are more complicated than the non-regex search term:

- `codewriting.org`

Probably in most cases of a find-and-replace task, I don't get to use regex.

But what if we needed to find all instances of `http://` protocol indicator for a `codewriting.org` URL in our docs and replace `http:` with `https:`. We don't want this for every instance of `http:` in the document, but we also cannot count on the domain string to always appear intact, as `http://codewriting.org`, which we could easily replace with `https://codewriting.org`. Making matters more complicated, numerous instances of a `codewriting.org` url appear with a hostname or "subdomain", such as `docs.codewriting.org`. To make sure we caught them all manually, we'd first have to create a list, and then iterate through it one by one replacing the whole string.

Here's the simple regex pattern that will save us all this trouble:

```
http\:\/\/([a-z]+\.)?codewriting\.org
```

This regex would efficiently match the following strings:

- `http://codewriting.org`
- `http://git.codewriting.org`
- `http://staging.codewriting.org`

The single set of parentheses in the regex will capture any group it matches and store it as a variable we can later reference to reinsert the captured string. The `$1` in the following replace string represents the first (and only) group captured in each of our examples.

```
https://$1codewriting.org
```

The `$1` will insert whatever was captured for each instance: either an empty string, `git.`, or `staging.`. Therefore, one find/replace procedure will handle the whole fix.

The above pattern assumes I know that all subdomains are lowercase letters (`[a-z]+`), or it would get more complicated (`[a-zA-Z0-9][a-zA-Z0-9\-]+`). This pattern indicates a single alphanumeric character followed by any combination of alphanumeric or the `-` symbol, the only allowable format for hostnames.

Let's say for some reason we also have various top-level domains in addition to different subdomains. Maybe we also bought `codewriting.net` and have different apps running on different hosts, and our docs need to reflect the distinction.

We need to upgrade our pattern to catch these new variations.

```
http\:\/\/([a-zA-Z0-9][a-zA-Z0-9\-]+\.)?codewriting\.([a-zA-Z]{2,})
```

Now we can match several new patterns that we were unable to previously, along with a second group for reflecting the top-level domain.

- http://codewriting.org
- http://git.codewriting.org
- http://staging.codewriting.org
- http://shop.codewriting.com
- http://members.codewriting.net
- http://dating.codewriting.net
- http://codewriting.ca
- http://1staging.codewriting.org

Our replace string is still pretty simple, with the second capture group token added in place of the definite `org`:

```
https://$1codewriting.$2
```

Why (and When) to Use Regex

Make no mistake about it, regular expressions are a labor-saving tool. They are for automating routines. They either save you time or they are not worth the effort of writing them.

When it comes to these everyday use cases, it's not always clear that employing regex will actually save you time. It will almost certainly be *more fun* to spend even an hour trying to write just the perfect expression that catches everything you want and nothing more. Remember, the alternative is performing a routine task with numerous slight variations, change after manual change, without introducing any errors.

> Of course errors can be introduced by improperly written regular expressions, but these mistakes tend to be consistent: they can be tested for and fixed using — you guessed it — regex. Errors introduced manually are more likely to vary. You may someday discover having mistyped `"` as `'`, but elsewhere another typo: `''`.

Still, that hour-long regex challenge will only save you time if manually finding and replacing every instance would take more than an hour to accomplish, with the added fact that you might introduce mistakes and not catch them. (You'd really want to use regular expressions to catch those mistakes, which begs the question.)

What's more, as with pretty much all open-ended task work, estimating the difficulty of writing a regular expression to solve a one-off problem can be challenging. I readily admit I have on numerous

occasions spent more time struggling with a regular expression than I would have saved if I'd nailed it outright.

Nevertheless, I think overall it has saved me hundreds of hours on day-to-day one-off tasks, including dataset migrations and HTML refactoring. But when regex is used to automate a routine task you'd otherwise have to start and repeat on a regular basis, the value can become immeasurable. We'll explore such a case in Part Four.

Suffice it to say, becoming wise about the up-front burdens imposed by regex is important, but any serious documentarian is bound to encounter cases where the right regex will save critical hours, or at least tedious hours. So let's demystify this powerful tool before you're expected to put it to use on a real-world task.

Digging Into Regex

Detailed information about new features comes in to me from engineers one of two ways. Either they refer me to the **source code** and expect me to extract all necessary details for it, or they give me a **text document** of some kind containing lots of loosely but consistently formatted clusters of data. My job is to translate some portion of those documents into user-facing information. In both cases, I turn straight to regex.

```
TODO Example needed
```

Let's assume a modest target format.

```
some text::
Another chunk of text
+
[horizontal]
Default:::
+
[horizontal]
Required::: Required.

more text::
A string similar to above
+
[horizontal]
Default::: false.
+
[horizontal]
Required::: Optional.

third line::
This is some "text" with "quotes"
+
[horizontal]
Default::: 100.
+
[horizontal]
Required::: Optional.
```

Just so you catch my aim, the unstyled Asciidoctor output of this AsciiDoc source is pretty basic, but good enough for our example.

some text

Another chunk of text

Default

Required

Required.

more text

A string similar to above

Default

false.

Required

Optional.

third line

This is some "text" with "quotes"

Default

100.

Required

Optional.

Getting back to the task at hand, we're essentially just trying to rearrange some things. But those basic changes render standard find and replace nearly useless.

Quite usefully, a regex pattern can store portions of matches as variables for insertion during a subsequent procedure. Wrapping pattern groups in parenthesis indicates that we want to store the captured content as a variable. By default, these groups are tokenized numerically and can be expressed later as `$1`, `$2`, and so forth.

Here is an appropriate regex pattern to flexibly match these entries, including discrete portions of the content, which we want to carry over in a new arrangement.

Regex matching string with capture groups

```
^\"(.*)\"\:\s(.*)\.\s(Optional\.|Required\.)(?:\sDefaults to )?(.*)\.?$
```

This probably looks both sloppy and intimidating. I still find these big patterns a bit challenging to read, let alone write, but regex isn't that hard to get comfortable with. I can't say I'm not enjoying the shock

value, though.

We are attempting to capture four groups wrapped in `( )`. Meanwhile, we're ignoring a fifth group (the one with the `Defaults to` string in it), as we have no use for that text.

Once you've captured your groups, writing the parsing template is relatively straightforward. Simply use the `$n` token, where `n` is the capture order slot for that variable. This way they can be reproduced out of order.

Regex replace template for find/replace operation

```
$1::\n$2\n+\n[horizontal]\nDefault::: $4\n+\n[horizontal]\nRequired::: $3\n
```

The `\n` token denotes a newline marker, of course you see four numbered group tokens we discussed a moment ago. The rest of the markup you'll recognize from our AsciiDoc. Let's look at what this find/replace operation produced — you'll recommend most of the other replacement template in here.

Regex find and replace result

```
some text::
Another chunk of text
+
[horizontal]
Default:::
Required::: Required.

more text::
A string similar to above
+
[horizontal]
Default::: false.
Required::: Optional.

third line::
This is some "text" with "quotes"
+
[horizontal]
Default::: 100.
Required::: Optional.
```

This is not perfect. We'll still need to do another round or two to get it how we need it to look. But you'll see now why I captured the `.` character in `Optional.` and `Required.`. Our next move will be to run a simple (non-regex) find and replace procedure on the string `Required.`, which will miss the `Required` in `Required:::`. We want to replace `Required.` with `Yes`.

Finally, we can replace `Optional.` with `No`. Now our document matches our target format precisely.

What did we save ourselves? It obviously depends on how hard it was to create the patterns. As soon as you get them right, you're pretty much done. And we did not have to manually remove quotation marks,

insert carriage returns, numerous newlines, that repetitive + symbol, and all those infernal colons.

As a bonus, we got to solve a cool little puzzle! Seriously what more could a technical documentarian ask for?

Proficient use of regex does not require memorization of all regex patterns — they can always be looked up. Besides, you'll use the same handful of character combos most of the time, adapting common patterns to your specific content or data scenario. Getting the concepts right early on is far more important than memorizing symbols, so let's start there.

Regex in .htaccess

Another powerful real-time use case for regular expressions is in the `.htaccess` file lots of us have used when we've needed to deal with website managing URLs on Apache-based webservers. This file, stored in the site's root directory, would be checked every time a request came to the site. If aspects of the request matched patterns we designated in that file, the browser would be redirected as we saw fit. We use this to force all browsers to use HTTPS protocol, or when URL paths change and we want to redirect browsers and search engines to the new address.

Example — .htaccess rewrite

```
RewriteCond %{SERVER_PORT} 80
RewriteRule ^(.*)$ https://example.com/$1
```

If the requested server port is `80`, the connection is insecure. The next line commands the resetting the request URL by capturing everything in the URL path after the home directory. So if the request is for the URL `http://example.com/some-path/index.html`, and the `.htaccess` file is in the domain root directory (`/`), the above `RewriteRule` captures all text (`.*` matches *any content*) from the beginning of the string (`^`) to the end (`$`). The parentheses indicate content to capture and store in a variable — in this case, *everything*.

In our sample URL, this will have matched `some-path/index.html`, which we now want the webserver to append to anther call to our domain, this time using HTTPS protocol. The next string is our explicit root domain followed by `$1`, the token that indicates we want to insert the our first captured string (in this case, our only captured string) right in that spot. Our URL becomes `https://example.com/some-path/index.html`.

Regex is Coding

So far we've been sticking patterns into Find fields and onto simple configuration files — not exactly delving deep into agile programming.

Part Three: Delivering

Formerly called "publishing", the act of getting your technical content in front of readers' eyes involves much more than dumping it somewhere they can find it. Our readers don’t line up outside Barnes & Noble. For technical docs, delivery means meeting the user where they are, when they need us, and frankly staying out of their way the rest of the time. Delivery also refers to the next step in our implementation journey; in Part Three, we’re delivering the beginnings of a content platform, either to ourselves or our team.

Chapter 9. Hacking

We've talked a lot about what we want to do with our docs, and whom we are writing and building them for. Now it's time to talk about how we make it all happen. Here we explore the craft of customizing delivery systems to meet our needs, simple and extraordinary alike.

Getting your docs out the right way is always a matter of cobbling together the right toolchain, with a document structure and workflow that are "right enough" for the current problem with the existing team.

Successful documentation means integrating our docs into the user's workflow in a manner that does not disrupt or distract from use of the product.

Extensible Content Platforms

Consider text-centric content platforms that facilitate heavy customization and collaboration, as well as those that offer highly tailored layouts. These systems host APIs that enable developers to extend the platform. These APIs make content management systems drastically more appealing to publishers with complex or divergent use cases, which should be pretty much everyone doing something more sophisticated than a microblog (e.g., Twitter, Tumblr) or photo feed (e.g., Instagram, Pinterest).

Continuous Integration

DevOps people look at continuous integration, or CI, as a fully immersed development, testing, and even production building environment (especially for SaaS products). The idea of CI is to establish a seamless flow between writing, testing, and integrating new code into the existing product base.

Roll Your Own Platform

Although we will continue to explore best practices for establishing a great documentation environment, this is a remarkably immature space, so almost every solution requires some hacking, cobbling, and patching.

Chapter 10. Deployment

Static Site Generators

Static Extensibility

Static Hosting

Continuous Integration

Cloud Solutions

Build and Package

Generating Packaged Artifacts

Into the UI

Chapter 11. Delivering Quality

Which practices and protocols best facilitate accuracy, clarity, and thoroughness of coverage?

Assuring Accuracy

I'm going to make a frightening suggestion to shock you a bit, only to walk it back slightly right afterward: *technical documentation managers should own product accuracy*.

Right, so I don't mean we should take on quality assurance's job of making sure the released product meets specifications. I mean documentation should consider maintaining the state of what the product requires and supports, not just what it does. Let's face it, we end up being the last ones checking whatever spreadsheets whomever is keeping to track what is going into the product. At some point, pretty much everybody has had hands on certain kinds of references: Product Management, project managers, at least one developer, maybe testing/QA, maybe even you if you came across it and got fussy.

References

Documentation of this nature is a good candidate for evaluating different systems to see what works now (and will be most likely scale) with your team. This is strictly objective material — product metadata, if you will — about which there is a discernible truth: currently supported versions, current third-party integrations, licensed dependencies, known limitations. In other words, product metadata is pretty much anything that is falsifiable about how the product fits into its market or ecosystem.

One collaborative platform will be more effective than the others in enabling your team to establish canonical documentation which aligns with existential truth. That is, documentation that is definitive and accurately reflects reality. If you have to allow Markdown, Excel, or WYSIWYG tools to ensure participation, just find a way to respect the "everything in code" mantra, and set up whatever tools will suit the task.

Diagrams

Another place where unifying engineers, product managers, and technical writers could be critical is maintaining internal and customer-facing diagrams. Any depictions of product architecture, workflow, and so forth, are prime candidates for complex collaboration.

Whether you use UML (Ascii text-based diagram markup), OmniGraffle (full-featured diagramming WYSIWYG with XML source), or something in between, the key again is "everything in code". Reviewers should be able to clearly see what has changed in a revision, and everyone needs to be able to contribute.

Keeping basic, canonical diagrams is more critical than teams realize. Visualization is one of the best ways to establish cohesion of understanding internally. This is where developers and product managers

will often surface the different ways they see a product. Standardize the way your organization does diagrams, and choose a tool with which everyone can at least tinker, if not create.

Testing Docs

It is obvious that documentation can be manually tested, much the same way users test the product itself. The practice usually involves users following along with the docs and making sure what happens on the screen matches what's in the documentation.

Manual Testing

While it's obvious what manual docs testing is, there are lots of best practices a good DocOps specialist will be mindful of.

Linters as Docs Tests

Accuracy Testing

Automating Docs Tests

Part Four: Managing

Good DocOps practice involves establishing workflows to gather and coordinate content and possibly coworkers. Management of technical content is a labor-intensive operation pretty much regardless of tooling, but technology is your only true ally on this front. Even if your memory is perfect and you can keep track of what is where and whether it's current or needs to be updated or deleted, your coworkers unfortunately lack this talent.

Chapter 12. Collaboration

Working with engineers and fellow tech writers to ship complete, accurate, useful, and engaging docs every time.

Working with Existing Development Processes

Engineers typically expect technical writers to work around them. Tech writers tend not to have much power in the situation. I cannot stress enough how important it is to minimize disruption to developers' flow, even though sometimes disruption is precisely what is called for.

If you track developers' work in a project management system (including a Kanban board or other dashboard-style objective instrument), you have probably found a systematic way to communicate with developers through it. I won't go into the boring details of our solution, but we use JIRA, configured with a field for **Release Notes**, which will be exported to source prior to compiling the docs.

Our JIRA also has an issue type called "Docs Sub-Task", which is a way for engineers to create stubs, assigned to themselves, for documenting work that is scheduled or underway. This way, I can see these tickets in the pipeline as flags for work that should be hitting my desk soon.

This is an elaborate alternative to tagging the work ticket "Needs Docs" or something. In our case, we need a way to ensure code tickets don't get closed until their associated documentation work is ready, but we also do need a place for the developer to stick their notes. The next iteration of this system that we'll test will hopefully lighten the load.

Adapting Dev Workflows to Sync with Docs

Tech Writers Are Not Stenographers

Some professional writers have horror stories about subject-matter experts treating them like assistants. Yet I have heard tech writers remark that they have no choice but to let engineers do most of the writing, since it's their code, their concept, and so forth.

Tech writers all know that sometimes, in explaining new or complicated aspects of a product to us, SMEs practically write the whole segment. I find myself copy editing installation instructions as often as writing them from experience first.

Working with Engineers

I imagine some professional tech writers have been muttering at half these pages, "As if, you lucky sonofa — " I get it that not all are lucky enough to find themselves working with teams that highly value or elevate documentation. There are different ways to approach this problem that may work better in various organizations; I would not purport such problem-solving among my skills. However, I do know

something about helping engineers see documentation differently than they've come to expect.

Every time a new engineer joins the team, I ping him or her on Slack and schedule a sync up. I present a bit about how I approach documentation, as well as what is expected from them. I want them to appreciate early on that my job is to help them do two of the parts of their job they wish could be automated: *accessing local knowledge* and *documenting their own code*. I also make it clear that I get purchase for the requisite initiatives from their support for and engagement with my procedures and technologies.

The Promise of Good Docs

Various constituencies at Rocana depend on the docs that fall under my purview. What each constituency needs out of my docs is quite different, and it might seem like I'm pulled in different directions.

But the truth is everybody wants the same thing. No matter how I get it done, I basically have to make sure the docs are always:

- **S**earchable
- **O**rganized
- **A**ccurate
- **R**eadable
- **C**anonical
- **E**ditable

  (Yes, I saw a pneumonic and I went for it. *Accurate* stayed in when *unimpeachable* would have made *SOURCE*, because *editable* implies impeachability.)

Let me introduce you to my immediate constituents.

Engineering

I need the engineering managers to see the devs are working more accurately and more efficiently under my systems. I also need them to see that, in collaboration with project managers, our documentation systems are catching the more passive documentation tasks, such as

- maintaining bill-of-materials requirements (a NOTICE file);
- announcing and following up on product feature deprecations;
- preparing content about the product for inclusion in the UI; and
- processing and copy editing the Release Notes and Changelog.

Product/CTO

The Director of Product and the CTO together determine priorities for engineering resources. I need them to see the devs are doing more of the work in Java, JavaScript, and Golang that make up the

digital assets we're selling. All of that research, coding, and testing has truly exponential value, and the docs are just there to make sure that value can be realized. That's humility I have, and it's perspective I don't mind Product having, as long as they appreciate that the docs form a critical layer, if not a profit-magnifying one.

Mind you, I have to deduct from my systems' ROI for any engineering time I depend on to establish tooling, as well as time spent reorienting the team to some new workflow routine. I have to be judicious with the engineering resources I'm afforded.

At some organizations, Product aren't focused on docs except to see content is accompanying what's released. and to learn I'm freeing up *more* engineers' time, not hogging their resources.

Support and Field Engineers

The other constituency I deal with directly is made up of engineers in our Customer Success department, as well as the Sales Engineers who have first contact with our end users. These are my primary audience, in fact; they are often literally on site with our customers while our end users engage with the product, and that means my docs need to minimize if not eliminate embarrassing frustrations. It's not unusual for a support engineer to check in with me just before a release to ask if there are any surprises, or to make sure some aspect of the product was updated.

My docs are truly for this situation, if they're for anything at all. When a highly paid product mentor from my own company is shoulder-to-shoulder with a user in a customer's IT department, both following along with my Reference Guide. I want them to find what they need, where and when they need it, and of course I want them to find only accurate information, consistent wherever and however it is presented.

These men and women are also often the first to discover bugs in my docs, which is of course invaluable for me. The field teams need a responsive system for submitting documentation bugs, and I need them to feel empowered to contribute heavily and directly to resolving docs bugs.

That may sound like a lot of pressure and a lot of varying interests to satisfy, and I admit that's often how it feels. Whenever this task feels like I'm exploring some uncharted territory of technical documentation, I just remember I work on a team that includes ReactJS coders, a machine-learning engineer, a whole bunch of Big Data platform engineers, and DevOps engineers (don't tell them I called them that — it's complicated).

Note consistency of these three interests just among my collaborators. *SOARCE* truly covers the whole array. If I maintain and facilitate good docs, everybody wins, even if almost nobody notices.

If, however, I fail to achieve those objectives, everybody notices. So there's that.

Constituents' Single Source of Truth

Collaborating with Users

All of the above is not even to have mentioned the end users themselves, or with developers extending the product for their own end users.

Most open source projects and an increasing number of contributor-driven commercial outfits are encouraging users to contribute directly to docs.

The Old Ways

Wikis

> The most recent wave of collaborative documentation may just be the second wind of the last collaborative craze. stems from very mixed, and probably mostly negative, experiences with opening wikis up to user contrubutions.

Discussion Boards

> Once upon a time, and still more often than I can bear, bulletin boards and discussion forums were used in the service of documentation. When used to solicit participation and feedback which in turn informs and improves official documentation, such as managed knowledge bases, use cases, and tutorials.
>
> Sometimes, however — and let's face it, drearier times — discussion forums are used as the docs themselves. Maybe a developer initiates threads in some of the channels, but each becomes a free-for-all of guesses and what-ifs, sometimes spiraling into flame wars. These arrangements become ne'er-pruned unofficial knowledge bases, mixing support requests with complaints, bad user advice with tardy official advice.

IRC

> IRC channels are ephemeral, flowing, and conversational formats, where one might expect to find an edge-case request mentioned, but no one would think to consider a sacred source of truth, something more valuable can be built. They are relatively anonymous, and somewhat hit or miss. Nevertheless, they always seem to have an authentic feel. A lot of times you get direct, realtime and symmetrical dialog with somene central to the project you need help on, right when you need that help. There's something to be said for that.
>
> The limitations of IRC are obvious. The more recent wave, with chat platforms like Gitter and Slack, product teams are interacting with extension developers and users in real time, with some sense of organization, flow, and a reliable archive. It seems to me these contemporary tools haven't necessarily attracted the trolls or the Google-overloading wastelands that stale Discussion Board pages become.
>
> In the end, I'm not sure I would include this category as documentation. Even if technical writers participate from time to time or derive content from such sources, I maybe it's lack of permanence or adherence to SOARCE disqualifies it.

Can User-Contributed Docs Be Done?

Chapter 13. Content Dissonance

As projects scale laterally and evolve over time, they require better handling of dimensions like versioning, forking, and localization. Only a distributed source management system will offer the flexibility needed for such growth in a docs-as-code environment, but what if you start losing track of all that content?

Version Entropy

Software and documentation have a natural tendency to diverge. Given time, even with consistent attention, more gaps between word and truth appear. In fact, if you're a *good* technical writer, like I think I am, you're too interested in adding new content to fuss over constantly improving the existing stuff. Only a *great* technical writer sets aside that universal urge and always attends to the existing landscape.

Product artifacts in which the code no longer matches the expectations set by docs has suffered what I call *version rot*. A sometimes-fatal affliction, version rot is almost always preceded by phases such as *version decay* and *version wilting*, all due to the tendency of products to undergo *version entropy* across release cycles.

The malady *version lag*, by which documentation merely trails a released version, is a fundamentally different matter. That said, version lag and version rot often mingle at the same engineering outfits.

This phenomenon can affect SaaS platforms at least as much as it affects installable artifacts, especially given how differently (and poorly) cloud-service operations sometimes handle releases.

If you are the tech writer for a SaaS shop with only one released, centralized version of one product, congratulations. Keep this to yourself at the next meetup, though.

Keeping Track of Released Material

There are probably about a million ways to release software. Consider the following data structure, intended to reflect several of the major considerations of releasing a software product. If documentation management is your career, consider the permutations of potential disarray you may someday encounter, or the range of variables you may see from job to job.

Software Release Chaos Matrix (YAML)

```
software-attributes:
  markets:
    consumer:
    business:
    enterprise:
  release:
    currently-supported-versions:
      - 1.8:
        - 1.8.0
        - 1.8.1
        - 1.8.2
      - 2.0:
        - 2.0.0
        - 2.0.1
        - 2.0.2
        - 2.0.3
    cadence:
      continuous delivery:
      manual release:
        - time-boxed
        - fixed-scope
    outlets:
      appstores:
        - apple
        - android
        - amazon
        - steam
  distro-environments:
    - webapp-responsive
    - webapp-mobile
    - ios-iphone
    - ios-ipad
    - android
    - windows
    - linux
    - macosx
  source-endpoints:
    - github.com
    - sourceforge.com
  dependencies:
    libraries:
      java:
      javascript:
    prerequisites:
      preinstall:
        codecs:
        sdks:
      compatabilities:
        operating system versions:
        hardware requirements:
```

And that's just the software you have to document. Now consider all the different ways that software

needs to be documented, with that documentation delivered across the various channels through which the product is distributed.

Documentation Coverage & Delivery Chaos Matrix (YAML)

```
documentation-attributes:
  output-formats:
    - xml
    - pdf
    - html
    - json
    - manpage
    - ebook
  delivery-endpoints:
    hosted-distro:
      - github.com
      - sourceforge.com
    api:
      - REST API
    client:
      - mobile app
      - desktop app
    in-app-content:
    print-materials:
  subject-matter:
    graphical-user-interface:
    command-line-interface:
    developer:
      - REST API
      - command-line interface
      - domain-specific language
    instructions:
      - installation
      - usage
      - troubleshooting
    references:
      - configuration-settings
      - product-attributes
      - support-channels
    legal:
      - licensing
      - warrantee
      - notices
  localization:
    languages:
      - english
      - spanish
      - french
      - mandarin
      - arabic
    interfaces:
      - north-america
      - international
```

Add several other factors that are difficult to appreciate even graphically, including

- the size of the documentation team;
- its relationship to Engineering, Product, Support, and possibly other organizations; not to mention
- the actual frequency of releases, each presumably carrying some default overhead.

The range of possibilities is truly enormous. Nobody is going to have all the answers for all the possible permutations of a given set of documentation challenges.

As much as I presented this as an attempt either to impress or intimidate, the true purpose of the above listing is to demonstrate why this book has a strategic focus, rather than a tactical one. Instead of trying to lay out a plan for every combination of factors and resources, let's examine how to use a systems approach to documentation challenges.

Feature Status Tracking

Localization

Chapter 14. Managing Complex Content

Technical content isn't only complicated from the macro perspective we took in the previous chapter. Discrete content items can themselves be highly complex, like little clusters of data stored in a routine format, all of it yearning for expression in application code and user docs. In this chapter we'll take a strategic look at handling complex content, including DRY single sourcing, validation, and collaborative workflow management.

The Challenge

Handling what I call complex content objects (CCOs) is one of the biggest challenges faced by a team dealing with lots of distinct, structured content items. I don't mean "chapters" or "topics", as in the ways we modularize tech docs. Instead, consider all the secondary and tertiary types of content you need to maintain and reuse to provide *references* for users.

Think of reusable content items such as data your team is keeping in a spreadsheet right now — the kind of content you wish you could port directly from engineers' internal documentation to your user-facing docs or UI, with just-in-time copy editing performed to boot. If that engineers' reference spreadsheet has two or more *columns*, then each *row* is a CCO. Each such spreadsheet is a *collection* of CCOs.

A glossary is a collection of CCOs, as is an API reference.

A Simple Example

CCOs are typically at least one step more complex than a parameter (a key-value pair). But let's start with a relatively "simple" form of CCO that maybe wouldn't normally induce us to seek a radical new solution to managing this content. For now, a simple parameter serves as a good warm-up example for exploring this topic.

Example — A parameter (in YAML)

```
my-key: some value for this key
```

Okay, we've got a key-value pair all right, but that's still pretty abstract. How about a more useful example?

Example — A useful parameter

```
"glossary term": The definition for the glossary term
```

Now we're getting somewhere. Let's try a real example by calling a snippet from one of the *Codewriting*

source files.

Example — Real data from _data/glossary.yml

```
  - term: complex content object
    definition: |
      Abbreviated CCO, this is an item that may be more metadata than content, stored many to a
file, as opposed to more typical text content units like chapters or topics, which may have
some metadata but are mainly text and images.
      See <<chapter-managing-complexity>>.

  - term: DITA
    definition: |
      Acronym for _Darwin Information Typing System_.
      An XML standard designed for technical writing, introduced by IBM in 2005.
      In addition to a markup standard, DITA incorporates dynamic functionality and mature
framework elements.
      https://en.wikipedia.org/wiki/Darwin_Information_Typing_Architecture[Wikipedia].

  - term: DRY
    definition: |
      For "don't repeat yourself", referring to a single-sourcing approach to code or docs.
```

What are some real-world uses for this?

The most obvious one is to generate the glossary at the back of this book, which it does.

Of course, I could have just written the Glossary in AsciiDoc markup, which is pretty much equally elegant, maybe even a little nicer:

Example — The same glossary snippet sourced as AsciiDoc

```
complex content object::
  Abbreviated CCO, this is an item that may be more metadata than content, stored many to a
file, as opposed to more typical text content units like chapters or topics, which may have
some metadata but are mainly text and images.
  See <<complex-content-objects>>.

DITA::
  Acronym for _Darwin Information Typing System_.
  An XML standard designed for technical writing, introduced by IBM in 2005.
  In addition to a markup standard, DITA incorporates dynamic functionality and mature
framework elements.
  link:https://en.wikipedia.org/wiki/Darwin_Information_Typing_Architecture[Wikipedia].

DRY::
  For "don't repeat yourself", referring to a single-sourcing approach to code or docs.
```

That markup is perfectly fine, and in fact it is the form the Glossary started in when I first began writing this book. The only real problem with storing content/data in this format is it's not portable; only AsciiDoc tools can do anything with it out of the box, and there aren't many AsciiDoc tools that focus on

manipulating complex content objects.

What if I wanted to be able to transform these glossary terms where they actually appear in the body of my book — everywhere *except* the glossary itself? Perhaps I want to highlight the terms wherever they appear in the main text, so if users hover over or click on them, they'll see the definition in a tooltip above the word.

In order to do this, I need a convenient way to handle and reformat just the term I need. AsciiDoc is not very helpful here, since it does not provide content to external tools as defined, semi-structured data like data-focused markup such as XML, JSON, and YAML. Already, we've discovered a reason we should look at storing *all* discrete, parameterized content as semi-structured or structured data.

Data Structure Terms

I'm going to use several descriptors for the format in which content can be stored "on the backend". That is, the form in which we hold our content so it can be best accessed in all the ways a product team might need it. We handle each class of content differently, especially when the content is better described as *data* than as *text*.

unstructured text

This could be prose, or it could be gobbledigük. This could be a plain text file with words or data dumped in it, but I'll more likely refer to unstructured data in terms of open form textarea fields that may permit markup but do not enforce any structure.

Unstructured Content Example

```
This is a box
  of unstructured content.

  You may have some expectation that this text
will be rendered in the shape it was
       entered.

Or
maybe not.
```

Just for the record, here is how that content looks after being processed by an AsciiDoc interpreter.

This is a box of unstructured content.

```
  You may have some expectation that this text
will be rendered in the shape it was
       entered.
```

Or maybe not.

structured text

Text-heavy documents, whether or not governed by a schema, are considered structured content if they respect standardized formatting and an orderly, hierarchical document model. Depending on the application, users may treat such documents with strict regard to a document model or schema, or they may abuse intended strictures and conventions. In either case, we'll use *structured text* to describe formats that adhere to some rhyme or reason.

Structured Content Example

```
# Structured Content

In Markdown, the `#` symbol indicates the header level.
```

semi-structured data

This refers to content in recognized formats such as JSON, YAML, XML, CSV, at least some of which needs to be handled very much like data. These items tend to appear many to a page; they often convey looped content of interest more for reference than reading.

Semi-structured Data Example

```
{
  "key": "some_property-name",
  "property": "some.property-name",
  "section": "General",
  "description": "This is some text in `AsciiDoc`, with *bold*.\n",
  "default": 15,
  "hidden": false,
  "example": "some.property-name = 22",
  "openblock": ".Example Two\n----\nThis is some text that will convert to a code
listing.\n----\n"
}
```

Imagine a collection of such property CCIs listed like a directory.

structured data

Schematified data, as in databases (including RDB and NoSQL) or XML with schema validation. This generally includes datasets that can be managed through queries.

Structured Data Example

```
CREATE TABLE IF NOT EXISTS data (
default INT,
description VARCHAR(46),
example VARCHAR(23),
hidden VARCHAR(),
key VARCHAR(18),
openblock VARCHAR(78),
property VARCHAR(18),
section VARCHAR(7)
);
INSERT INTO data VALUES
(15,'This is some text in `AsciiDoc`, with *bold*.
','some.property-name = 22',false,'some_property-name','.Example Two
```

This is some text that will convert to a code listing.

```
','some.property-name','General');
```

As you can deduct from the examples above, schemas can be advantageous. For starters, they can define the data type and constrain each value. With no such safeguards, semi-structured formats are open to invalid entries.

The disadvantage can also be seen: structured databases do not lend themselves well to representation as text/code, let alone to management in flat files. We'll look at more of the reasons I think *semi-structured* data is the technical communicator's Goldilocks data class.

Digging Deeper

We've seen how even simple key-value paired content can be output in unconventional ways, rather than always dumped as a list in whatever form it arrived. Now let's take a deeper look at a kind of content that requires even more flexibility.

Example — A Complex Content Object or CCO (in YAML)

```
my-key:
  slug: mykey
  description: The key to your project.
  required: true
  type: string
  max-length: 20
```

The best example of a CCO type that I know of is my personal Moby Dick: a means of cataloguing hundreds of configuration properties in a multi-component enterprise IT Ops product. Our product is server software — about a dozen distinct daemons that may run across scores of nodes at a massive, multi-datacenter operation. Our users can configure all of these services in myriad ways by setting key-

value pairs through various configuration interfaces.

Let's get some more views of a configuration setting and all of its properties, so we can appreciate how flexible our source needs to be.

Example 2. Example — Our product setting in the eyes of a user

My Key*:

The key to your project.

Example — And in the eyes of a front-end dev (1)

```
{
  "my-key": {
    "slug": "mykey",
    "description": "The key to your project.",
    "required": true,
    "type": "string",
    "max-length": 20
  }
}
```

Example — Again, our setting as also seen by a front-end dev (2)

```
<input
  type="string"
  name="{{ slug }}"
  id="field-{{ slug }}-input"
  value="{{ value }}"
  data-validation="maxlen: {{ max-length }}" />{{ required }}
<span class="description" id="field-{{ slug }}-description">{{ description }}</span>
```

Notice that in this example the properties do not even appear in the same order; it's more than a matter of reskinning the same old data with different markup.

Example — And our product setting in the eyes of a back-end dev

```
public createKey(String type, String slug, String description, Boolean required) {
  this.type = type;
  this.slug = slug;
  this.description = desc;
  this.required = false;
}
```

As you can see, a simple product setting parameter can be useful in diverse settings:

1. the documentation about that parameter

2. in the interface where users can edit the property and appreciate an inline description
3. in the back-end code that actually uses the setting to determine the product's functionality

Users need reliable access to accurate documentation of these settings, in the appropriate context of the component they're meant to configure.

Let's take a look at a more complete configuration setting's properties as a YAML structure. As you examine this figure, try to ignore the format and focus on the qualities of the content object.

A complex content object (CCO) represented in YAML

```
parameters:
  - key: scratch.location.path #unique identifier
    label: "scratch location"
    type: string
    attributes:
      description: |
        The path to which we should save temporary files
        used in the build process, relative to the product
        root or as an absolute path with leading `/`.
        Accepts the tokens `{{ output-type }}` and
        `{{ environment }}`.
      required: false
      required-caveat:
      default-value: "_scratch"
    example:
      type: listing
      content: |
        # where to write temporary files
        output.location = /tmp/documator/scratch
    status:
      introduced: "0.9.0"
      deprecated:
      removed:
    environments:
      - enterprise:
        present: true
        customizations:
        override:
          - deprecated: "1.6.0"
          - removed:
      - developer:
        present: true
      - cloud:
        present: false
```

Even at a glance, you should be able to appreciate what makes this content "complex". As we see in the above snippet, a multi-version, multi-environment product could quickly require a relatively complex data structure to track how its configuration parameters relate to the product's various states and forms.

Cloud Collaboration in Google Sheets

A lot of people hate Google Docs. I am sympathetic, but I am not among them. There have been countless frustrations with Google's authentication system and its quirky UI, but the truth is, I've made heavy use of each component in the Google Docs playground. Each has its legitimate uses, even if all are overused.

Consider Google Sheets as a collaborative CSV-management system. So long as you can export to the commonly-readable CSV format, you can port data managed by developers in Google sheets into just about any documentation output form imaginable.

For some content, this might be the shortest route to keeping fast-changing data as current as humanly as possible. This is doubly true when not all of your users are developers adept at working in flat files; most project managers and a huge range of SMEs will be comfortable with spreadsheets.

When content in a cloud-hosted spreadsheet changes, save the .CSV file into your source repo and use it to generate output.

Considerations

The following factors must be taken into account when strategically addressing a CCO management problem.

authoring access

Is this system going to be someone's sole domain, or is it a crowdsourced effort of two or more?

automation

If your docs are integrated with a build system, it will have implications on your choice of toolchain for this functionality.

product versioning

If by some stroke of genius or simplicity your product does not require release version control, you may be able to get away with a conventional CMS backed by a relational database (RDB). The need to tie complex content with product version argues for a flat-file approach, with settings kept in direct alignment with product code.

advanced data operations

Will you need to **sort** CCOs by any of their parameter values? How about **filtering** to output only certain instances of a CCO type in certain places? Think once more in terms of that spreadsheet; sorting and filtering are key features. Consider if you will ever need to re-sort your CCOs for publication (beyond just reformatting each CCO entry with a template).

Similarly, consider whether you'll need to combine data from different sets. This is most conveniently done with SQL, which has a concept of *unions* between data tables.

output formats

What formats will your CCOs need to be output into?

- Markdown
- AsciiDoc
- reStructuredText
- XML
- HTML
- DITA
- Docbook
- LaTeX
- PDF
- Man
- JSON

Your solution will need to be mappable to all the required formats. It will need to be exportable to something they can read, or external resources will need access to your CCO source directly and parse their needs from it.

output languages

Internationalization support will either require another layer of development skill and effort or a budget for a CCMS with these capabilities built in (and of course suiting all your other CCO needs).

quantity of objects

I don't know what the ceiling is, but there is certainly a limit to how many such objects can be sensibly managed in anything but an indexed database table. A spreadsheet application is such a tool for small data like we're talking about. It likely does not make sense to use flat files for datasources with several hundred or more entries.

relationality

No matter how many records you're keeping track of, once they need to correlate to differently structured records, your options get pretty limited. Flatter solutions work for records of self-contained data.

data dynamism

This is not the first or most-common consideration, but I think it should be among them. Since this is a major shortcoming of the RDB model, it is a place where flat-file approaches might shine. This is the inclusion of variable data inside a datasource, especially for cumulative inheritance of values set

in a linear fashion. Give me a moment to explain: In source code, parameters are often built cumulatively.

Example of cumulative parameter setting

```
```

Solution Approaches

As a codewriter, I'm always leaning for the flat-file approach, but as you can see, it's not always going to make sense. Before we settle on the flat file, let's seriously consider some other options.

Relational Databases

If your background is in website development or content management systems, the solution may seem obvious. As recently as a couple years ago, I would have solved this with a relational database. Structured Query Language is not just a way of inserting, updating, and retrieving data in a relational database; it can also express the proper schema of a data object. After all, a database table is basically a container for data objects, and the container is eminently configurable.

One major problem with the RDB solution is version control. Source-control systems like Git don't work to track the innards of relational databases, which tend to use binary files. Git has no insight into the structure of your conventional RDB file, so it cannot track changes made to the data. Therefore, an RDB will also require specific version tagging of any version-sensitive data, a layer of manual complexity that is fairly prohibitive. One of the reasons we're doing docs-as-code in the first place is so that we can associate our single-sourced product info with the version of the product to which it applies

The other major limitation of using RDBs for a constantly changing array of datasets is the interface. RDBs require query transactions to access data, including for reading and writing. This can be abstracted into form interfaces, as with a traditional CMS, but then that CMS has to be administered and maintained.

A good, open CMS platform such as Drupal provides your product build system with REST API access to your human-maintained data. So if you are willing and able to maintain such system, an external CMS can work with a docs-as-code solution. If the demands of your product will be relatively static — meaning, no new forms of complex data popping up all the time, maybe an RDB plus a web-based interface and straight JDBC/SQL on the back end can solve your needs. As an old hand with Drupal and WordPress, I always consider a dynamic, server-side solution when I encounter a new challenge.

However, the factors mitigating the RDB-backend and web-UI solution are very strong compared to the relative freedom and ease of a flat file. Sure, you'd never open a flat-file datasource to just any user, but in a professional environment with a proper workflow in place for vetting new content, flat files are superior low-volume datasources.

DITA CCMS

The DITA system's **specialization** feature can be used to define and maintain CCOs. So DITA platforms naturally come ready to handle this issue using an elegant interface with XML "under the hood". I'd call that a pretty significant advantage.

One limitation of these systems is they require special tools: editors that manage content.

Most DITA CCMSes seem to use relational database back ends, though some now handle flat-file storage and integrate nicely with Git.

Whether content is stored in an RDB or in flat files, the platform is enhanced by a database-backed content-management interface. The advantages of this are significant, including powers for reorganizing content that are tedious and clumsy to do with conventional file-management tools.

Unfortunately, those special tools really do add expense and weight. License fees range from $50 to hundreds of dollars a month *per seat*, making it prohibitive to involve all SMEs as docs contributors. Despite my idealistic description above, these tools also tend to be somewhat clunky, and they are very much Windows-oriented. Engineers tend to dislike clunkiness and Windows, not coincidentally.

Localization and translation management are another major advantage of DITA-based systems. If you need to generate documentation in multiple languages and do not wish to reinvent this (very complicated) wheel, proprietary CCMSes or major open source CMSes are probably your best options.

Flat Files

At Rocana, we strongly desired to keep our complex content in flat files. AsciiDoc has not proved robust for this situation. It falls down in several places. Complex content written in AsciiDoc is not even proper data; there is really no way to interact with discrete items outside AsciiDoc tooling. This was illustrated with the glossary example above.

We also needed to be able to output complex content in several distinct formats, including:

1. sample configuration files in what's called "INI format", a dreadfully plain blob conventional to Java platforms;
2. Reference Guide sections in the chapter "Configuring Rocana Ops", which is sourced in AsciiDoc and published as both HTML and PDF;
3. GUI forms for central configuration of remote components via our web application;
4. an internal Knowledge Base where hidden configuration settings and troubleshooting advice can be accessed by engineers and support agents, but not necessarily all end users;
5. similarly to the Glossary example, our online Reference Guide and other materials could generate a tooltip for whenever text is hovered, exposing more information about specific settings wherever those settings are referenced in the text.

As you can see, I had big plans and high hopes for all that complex content.

The challenge is to:

1. allow engineers to maintain YAML files as they manage the rest of their code
2. without sacrificing any of the ways this material was already being delivered (AsciiDoc and INI files), and meanwhile
3. add the ability to integrate data directly into the application and from a single source external to both.

After months of hemming and hawing over the best way, which included talking with consultants about how to handle the matter, I finally sat down and took a shot at coding a solution.

Introducing “LiquiDoc”

My system is simple. It’s a little command-line tool that ingests data, passes that data to templates for variable substitution, and then saves copies of the populated templates as new files in the appropriate format. A simple configuration system lets me coordinate source files (YAML) with templates (Liquid) to save the results as.

Example — LiquiDoc Coordination File

```
- action: parse
  data: data/data-sample.yml
  builds:
  - template: templates/tpl-sample.asciidoc
    output: output/sample-consumer.adoc
  - template: templates/tpl-sample.ini
    output: output/sample-consumer.properties
```

What follows is an example data source file. It contains metadata about the topic (this collection of settings) as well as a couple of dummy settings.

Example — LiquiDoc Source Data File

```
meta:
  title: Platform # Template will append "Settings"
  description: |
    This is some awesome text that describes the platform settings displayed on this page and
anything particularly special about it. Feel free to use AsciiDoc formatting here, as it will
be parsed.
  sections: # List all of the categories of properties in this file
    - name: Basic
    - name: Security
      instructions: Be careful with these!

settings:
  - key: some_setting-name
    setting: some.setting-name
    section: General
    description: This is some text in `AsciiDoc`, with *bold*.
    default: 15
    required: true
    example: some.setting-name = 22

  - key: another_setting-name
    setting: another.setting-name
    section: Security
    description: A different description for this _resource_.
    default: true
    required: false
    example: |
      # This should appear as a literal.
      another.setting-name = whatever
```

Flat-file shortcomings

Remember that concept of *relationality* in small data that I brought up earlier? Where There are ways to sensibly "join" two semi-structured data files the way an RDB joins tables, but it's clunkier than a system with a true database backing it. If you have files named `people.yaml` and `jobs.yaml`, you're going to have an uphill battle keeping people associated with their jobs, and jobs associated with the people performing them. It's not an impossible problem to address down to this layer of complexity, but only a real product-development process will nail the architecture, interface, and query language for handling such relationships for highly complex small data in flat files.

Where This is Going

Imagine the value of having canonical product data in semi-structured format, sitting side-by-side with product source code. This solution allows version control to keep information *about* the code up to date with the code itself.

Hardcore engineers will balk at the idea of storing generated source alongside its own source. This would be the case were you to store data in a format such as YAML, then generate Markdown or

AsciiDoc files from that code and commit it to the same repo as the YAML source. The source-control purists are actually right: this is poor form and an accident waiting to happen. It may also be the best option for keeping your docs in great shape, so argue for it if you need to.

The better argument, however, would be for continuous integration. With CI, the product build will just call a script, and those files will be generated during the build, then compiled along with the rest of the docs as if they'd been sitting there all along. If you are not authorized or confident enough to hack the build, maybe you can recruit a DevOps/automation engineer to champion this level of real-time integration.

Finally, the real coup is when engineers start using your semi-structured data files as source libraries for their product code. If an engineer is hacking your YAML file to document a configuration property, including its metadata (data type, constraints, required status, etc), why wouldn't that engineer just use your YAML file as the *source* of that metadata to help define their code? Any YAML file can be a library or data source for other aspects of the product. This is how we establish canonical data in DRY, single-sourced complex content systems.

Chapter 15. Managing (at) Scale

Using extensible publishing and delivery technologies to accommodate the growth of your organization, the quantity and maturity of your products, and the hopefully diversifying user base that makes up your audience.

Part Five: Evolving

In the final chapters of this book, we express and explore aspirations. As with the rise of development frameworks and agile methodologies over the past 15 years, so too can technical documentation enjoy a comparable (if humbly proportionate) renaissance of writing approaches and tooling strategies. Perhaps DocOps can ride engineering's coat tails into the new way of operating, lest we be left behind as engineers iterate their way to automating our roles entirely.

The future of DocOps may not be solely in software or product documentation more broadly. What other fields could benefit from codewriting principles?

Chapter 16. Driving TechComm Forward

Just when you were getting excited about tech writing, automation starts nipping at our heels. Tech writers must learn to stay ahead of the game, not just the field.

To use a cringe-worthy term, *thought leadership* is sparse in the technical documentation field. This has shocked me during my brief tenure practicing the craft. I can't help seeing irony in the lack of accessible writing about how to do documentation right. When coders stepped out of their closed-source silos and joined the revolution about 15 years ago, technical writers largely stayed behind. Those who have been doing it "right" so far seem to have uncharacteristically largely failed to share their successful strategies and tactics, myself included.

There are some notable exceptions I hope to have called out plentifully throughout this book. Even behind the scenes in various unsung repositories, code-comfy tech writers and word-happy coders have shared repos, snippets, and gists with lots of great gems. Please contribute your favorites to Resources.

Experiment with Me

If DocOps is to emerge (perhaps under some better name) as a concrete methodology we can more clearly instruct, it must undergo rigorous field testing and collaborative iteration. *Codewriting* is one step in the development of a mature framework for wrestling complicated software products into comprehensible clusters of coherent descriptions, illustrations, instructions, and inspirations — then delivering them to users.

Writing a book in a pubic repo is my attempt to nudge the field forward in its mindset. We're a bunch of grown ups used to corralling testy engineers and conjuring stale prose to earn a crust; of course we can collaborate with fellow pros to co-write a user manual for a next-gen docs methodology. If you've been hoping somebody would do this, join the club. Literally.

Foresee the Threat

This is where we get back to the threat posed by automation. You might think I'm joking, and it is indeed a strange topic for a tech-writing book, but in truth, very soon (surely within 10 years) it will become possible to automate a significant portion of technical writing effort without a major loss of overall quality.

If you think I am exaggerating, look at what powerful AI instances such as DeepMind and Watson are already doing.

Google's DeepMind is able to learn to play and defeat video games just by using the pixels on the screen and employing advanced trial and error to figure out the rules and "physics" of the game. With no docs to read and no insight into the product not available to an actual end user, a machine can perfectly learn

a complicated program.

DeepMind meanwhile likely has no ability to explain what it sees, let alone to determine what to share with a user vs what to leave for them to figure out.

IBM's Watson *can* read docs, and it can establish what looks like contextualized understandings of complicated matters of fact and even concept. It can also generate seemingly original summary output based on relatively complicated and highly diverse inputs.

The kind of feedback a screen-viewing, trying-and-erring bot like DeepMind will provide, converted to some reporting format, would be of incalculable value to GUI developers. QA engineers will be all over tools that can *use* software via the front-end interface. The minute some smart-ass QA engineer decides to hit btn:[print] while using DeepWatson to debug an app, stock in my trade goes way down. Once the machine's observations are expressed as descriptions of on-screen occurrences, tech writers get an amazing tool, and also possibly lose some coworkers.

This will likely reach way beyond the world of software docs, since nearly all products are digitally engineered today. Any product whose source and specs can be read is subject to at least partially automated documentation. In a just world, that would be exciting — it sounds like we get tools to alleviate the more routine aspects of our gig so we can focus on the more interesting stuff anyway.

Except the world isn't just; bosses will always hog the profits afforded by innovations. They aren't going to keep funding tech writers and docs managers out of the goodness of their hearts. Some might reallocate writer resources to get way better docs on top of such new tools, but I expect most will want cheaper adequate docs. That means big layoffs with better tooling, even if someone will always need to be around to run them.

My inclination is to call for overthrowing capitalism, but I can't explain that strategy in these pages. We're stuck with becoming invaluable to our teams, or maybe to professionals in some less-technical field who need DocOps hackers to establish and instruct collaborative docs environments.

We won't be alone among the slashed. Only the best tiers among our software developer friends — the ones who innovate chaotically and radically — will make it all the way to the Singularity as professional engineers. Technical writers' half life may already have been spent. The good news is, if you are reading this particular document, you are almost certainly in the group that can't be so easily replaced: the ones who actually *care* about excellence in technical documentation. As of yet, no machine can be programmed to desire to improve a user's life (or their day or minute) by helping them connect with a product they believe they need to use.

Are we asking for it?

Maybe it has already crossed your mind that if we programmatize the technical writing environment too much, and involve engineers too directly, might we as well just be signing our own pink slips? Dig it: the more we integrate developers into the documentation process, the less we're needed.

Wouldn't we be better off if our Robot Overlords had to slog it out with Microsoft Word or some Adobe product that still comes on CD-ROM? Maybe if we do not invent the framework for our demise, we'll stave it off. But the writing is on the wall: we can either innovate and increase our value to the engineering and support organizations we are in today, or we can find strategies to entrench ourselves and resist progress.

We need to go in fully aware that we are giving our colleagues and bosses incredible power with which to more or less replace us. Except, of course, for the fact that we do so much more for the team.

Think about it: to the extent technical writers are copying/pasting, categorizing, styling, and testing engineers' notes and comments, we're talking processes ripe for automation. Engineers and capitalists would *love* to recuperate those tech writer salaries and their CCMS license fees.

So how do you shore up your position without resisting progress?

Add Value

In a profession that has sort of prided itself on the ability to standardize and blend in, the way forward is finding ways to stand out. Become able to identify those aspects of our work that would be hardest to automate — those that do not feel like routine, with new challenges from job to job. But also be aware of the areas we might get boxed in as the need for documentation dries up.

Then innovate on the hot spots; find your way closer to the cutting edge of the field, make a difference to users *and* the rest of the product team.

We won't be the only field of John Henrys out there racing the machine, and eventually it will catch up with us. But for now the frontier is open. Let's outrun the bots for as long as we can.

Lighten the Toolset

One way to thwart this trend — to everyone's benefit, I might add — is to lighten the churn burden of documentation systems.

Sweeten the Docs with a Human Touch

Another critical strategy is to increase the value of documentation by enhancing it with a profoundly human dimension and a uniquely strategic outlook on the role of documenting software products. The last thing machines are likely to master is the ability to reflect a user's purpose and intent.

Internal Docs

Facilitate or lead the development of an internal documentation platform, as explored in Docs Maturation Process and much further in [engineering_docs]. There are truly countless ways to provide services directly to your colleagues, including Customer Support, QA, DevOps/infrastructure, IT/ITOps, and so forth. Yes, companies with large teams in all these categories typically have their own writers; but do they have a DocOps specialist working as an internal-docs manager?

Organize Resources, in the Source

Continuing on the internal-docs theme, perhaps the most beneficial act would be to centralize complex resources, storing it all in source where it's most useful to engineers. I'm always shocked at how few teams do this, until I'm reminded how significant the challenge is. I've spoken with documentation consultants who were baffled by the very notion that I would attempt to crack this nut; many of them had never even considered it.

I don't think this is commonly addressed by docs specialists, but the truth is, reference resources are a huge problem for engineers. Teams have recently flocked to cloud-based solutions like Google Docs & Drive or Zoho, which enable real-time collaboration on all kinds of docs, including spreadsheets. This way multiple members of the same team, or even multiple teams across the organization, can maintain complicated, layered listings of commonly required data.

But engineers notoriously hate most of these tools. The only thing I find they nearly all agree on is Git. Help them treat all canonical content the way they treat their code, and your value will not be in doubt.

Improve Docs Delivery

Don't settle for *writing* or even *managing* the docs; forge new ways to *deliver* docs.

Chapter 17. Integrated Documentation Environment

One key objective of DocOps must be the conceptualization and development of true Integrated Documentation Environments-- code/text editors that are "source-code aware".

What We Lack

To my knowledge, no truly integrated documentation tool is openly available today. Before you suggest your favorite IDE or CCMS, let me explain a little more of what I mean by *truly integrated*.

Imagine the following: You are typing along in your favorite lightweight markup language (let's call it AsciiDoc), and you need to express the default value of something.

Example value expression in AsciiDoc

```
The default value of *login.allowed-attempts* is `3`.
```

As a reminder, those asterisk and backtick markers are just for formatting bold and literal, respectively. That whole line is static code, but it is *about* something dynamic. It's *subject* is *dynamic*.

So what happens when this default changes in the product codebase? Typically a tech writer has to be notified, or they must discover the change during a review of some external database/spreadsheet where settings and their defaults are tracked. In any case, the tech writer has to manually make the change republish the docs or wait for the docs to publish.

It would obviously be better to use a variable here, especially if this default is going to be expressed in any second place anywhere in the documentation.

Example value expression in AsciiDoc

```
The default value of *allowed.login.attempts* is `{login_allowed-attempts__default}`.
```

Okay, that looks a little bit better, but it doesn't solve the problem of requiring someone to change the value of the AsciiDoc variable **login_allowed-attempts__default**, wherever it is defined.

Now imagine that variable is sourced from a common registry which is edited by the developer when the default for the core setting **login.allowed-attempts** was changed. Suddenly your intervention is no longer needed on the trivial matter of a changed setting value, because the value of your AsciiDoc variable (login_allowed-attempts_default) is drawn from that registry.

This is a mature integrated docs environment, possessing potentially robust awareness of the product once integrated. Such a system could incorporate unit tests and regression tests to ensure the docs

always reflect the proper value of an existing parameter compared to the version of the software each docs edition describes.

Docs Generators

The embedded documentation generators associated with various programming languages are an important cousin to my idea of an integrated docs environment, and they've been around for quite a while. Systems like JavaDoc, PyDoc, YARD (Ruby), Perldoc, JSDoc, phpDocumentor, and several others essentially transform developers' structured code comments into automated documentation output. This form of docs-as-code has been around for many years, and it is extremely useful. But it is also notoriously limited, as it is more like a code-as-docs or a docs-*of*-code solution.

Both Swagger and Doxygen are excellent black boxes that take complicated inputs and produce well-structured, predictable output. They can meet developers on their turf, deep inside the codebase. Working with such tools requires up-front planning, configuration, and testing, but it can be worth it for the developer proximity they enable.

But embedded and integrated docs generators are just not good enough to produce great user-facing product docs. These tools are more oriented toward users who need to know how to interface with or directly hack a product's inner components. So while these embedded generators may suffice for producing API docs or developer references, they don't necessarily correspond in any direct way to the end-user interface. For most products with GUIs, source code and interface do not line up 1:1.

Besides, engineers are less than keen to have tech writers poking around in their source files. Surely any engineers reluctant to edit your markup files may be even less pumped about you editing their `.rb` or `.java` files.

And through all of this, we haven't solved the matter of needing the involvement of a professional writer focused on end users' interests. Someone will always need to make up for those devs who don't love doing docs, as well as provide consistency through editing the work of those who take docs seriously.

Docs generators are cool, but they pertain to developer documentation. DocOps projects can likely take advantage of many of these tools, especially where structured data can be kept outside scripting files that express actual functionality. This way, content/data in such files can be pulled into your docs environment for hands-on management, judiciously placed and output in different formats as needed.

REST APIs

I love REST APIs. A very talented engineer I worked with once assigned me Roy Thomas Fielding's epic dissertation on *Architectural Styles and the Design of Network-based Software Architectures*. I cannot claim to have read every word, but I found in it a new level of appreciation for REST interfaces *and* for technical writing. I also found a metaphor in the REST API; the elegance of API abstraction correlates to problem solving in DocOps.

A REST API is a set of URLs that can be connected to remotely via HTTP/S protocol, either by another server or a client such as a mobile app or Javascript in your laptop browser. Every interaction between a client and a REST API is a request (precisely like every web page call or form posting by a browser — a distinct, "stateless" interaction). The server running a REST API is prepared to respond to the specifics of each interaction with a client:

- the URL ("endpoint") the client is requesting
- the request's "verb" (`POST`, `GET`, `PUT`, `DELETE`, etc.)
- any parameters contained in the request

A REST API request can be a terrifically powerful and highly complex command. Some REST APIs exchange extraordinary amounts of structured data between machines on an ongoing basis without human intervention. Others are closer to serving as the middleware of a consumer application — the way a server-side application interfaces with multiple, disparate clients such as browsers or mobile apps. Suddenly, this sounds like a problem documentation managers are familiar with: lots of requests from various corners, with the need for universalized handling.

Facebook's REST API is an obvious example. Every interface Facebook provides has to work with the same, big server-side product. The ideal way to handle this is to provide a universal interface to which all clients must conform. A REST API is client agnostic; it accepts properly addressed and formatted requests from clients, including authentication data for security, without caring whether the client was Javascript in a web browser or a server on Wall Street.

Output to Product

The trend in software product interfaces is toward structuring any strings and other data associated with UIs in JSON format. Everything from labels to inline instructions is passed to the UI as semi-structured text.

Yet very few CCMSes or general documentation processors generate JSON, whether that data starts as HTML, XML, or lightweight markup languages. There are converters that produce JSON, but DITA CCMSes, reStructuredText, Markdown, and AsciiDoc are not tooled to spit out their own content in a format UIs need.

This limitation means that in order to obey the *write-once* rule, content intended for the UI may need to start and stay in the engineers' domain, closer to where it will be output. This is very likely fine for most interfaces. The front-end team doesn't need you tinkering directly with the strings used in their UI.

However, in cases where dynamic data is used in the UI *and* in your docs, the DRY directive suggests two options:

1. Your docs build need to pull (query) those strings from the developers' arena, however and wherever they're stored.
2. You need a tool that allows proper maintenance of such content on neutral ground — somewhere they can change significant details and you can correct grammatical errors and typos, all while the content remains equally reusable by both the UI compiler and the docs compiler.

Platform Integration

Of course, it isn't just the product codebase with which I want to integrate. I get excited every time I see one of my Atom plugins has been updated, hoping it will be even a small step toward greater convenience in my daily operations. Whenever Atom gets a little more aware — such as when it learned to display locally relative includes in the AsciiDoc Preview pane — I feel that much closer to having a better window into my final product.

There are lots of complicating factors that make it non-trivial to show you perfectly how your docs will look by generating a reasonable proximity in realtime. If you write in LML, you're used to this preprocessed rich-text display version alongside your markup.

I'm not asking for pure WYSIWYG power with the ability to write *inside* my data. Our point in writing in LML is not that we wish to stare the code in the eye. Rather, we appreciate the irreducible complexity of the content models we're trying to convey. We don't wish to hide semantic elements, such as indicators that a dynamically generated object will appear in the place of this object. And in most cases, we may not even wish to see our copy mixed in with the dynamic data that will eventually populate our variables and other placeholders.

What we do need is the ability to quickly preview our content and design with the data it documents, preferably without having to rebuild every time and check it in another application altogether (such as a browser).

Admin Powers

Chapter 18. Beyond Technical Communications

There may be numerous as-yet-unproven applications of the docs-as-code approach in fields beyond software and in fact beyond engineering. Anywhere documentation can gain from sharing and collaboration, firms of all kinds could have documentarians and copywriters sharing the non-proprietary parts of their work product.

I don't have tremendous confidence that there is a particularly significant amount of money in this area, but I do see lots of need.

Legal Docs

Legal boilerplate language complicates contracts. Lawyers who manage tons of clients would rather not risk divergence in the language of similar documents between clients.

Forking & Maintaining Boilerplate Language

Open Sourcing Legal Docs

Collaborative Journalism

Protocol & Policy Codexes

Any organization that has certain ways of doing things needs these ways documented. More often than not, this does not need to be unique — a customized derivative of another group's policies and procedures would suffice. If groups shared their policy documentation, other groups could clone and modify it to suit their needs. Perhaps the copy would diverge over time, or perhaps it would stay mostly in sync even as both groups (or maybe just the original group, or maybe only the copying group) evolved the policies.

Cookbooks

Here I mean actual culinary cookbooks (which tend to be codexes in their own right) as well as technology "cookbooks" — sets of "recipes" for optimal customization of a software product.

<Your Idea Here>

This spot is begging for contributions.

Back Matter

Appendix A: Codewriting Glossary

API

For *application programming interface*, an intentional framework by which third-party developers can enable their own programs to integrate with a product. Pronounced *A-P-I*. See also REST API.

AsciiDoc

An extensible lightweight markup language with numerous powerful features such as conditional processing, file includes, and variable text substitutions. Pronounced *ASK-ee-doc*.

CCO

See **complex content object**.

CLI

Command-line interface. Any place where a command prompt expects textual input. Pronounced *C-L-I*.

complex content object

Abbreviated CCO, this is an item that may be more metadata than content, stored many to a file, as opposed to more typical text content units like chapters or topics, which may have some metadata but are mainly text and images. See Managing Complex Content.

DITA

Acronym for *Darwin Information Typing System*. An XML standard designed for technical writing, introduced by IBM in 2005. In addition to a markup standard, DITA incorporates dynamic functionality and mature framework elements. Wikipedia.

DRY

For "don't repeat yourself", referring to a single-sourcing approach to code or docs.

FOSS

Acronym for *free, open source software*. "Free" here implies both free as in do what you want with it, and free as in no money need be exchanged.

JSON

A very common format for semi-structured data, popular for data transfer, especially with JavaScript/Node.js, though quickly becoming the industry standard. Pronounced *JAY-sahn*.

Markdown

One of the lightest weight and most popular textual markup languages, popularized on GitHub, StackOverflow, various forums and comments systems, and numerous other places. Markdown, however, is not a dynamic language.

GUI

Acronym for *graphical user interface*. Pronounced *GOO-ee*.

IDE

For *integrated development environment*, a toolchain/platform that facilitates software programming in a customized context that incorporates the particularities of the source language(s) and the product build. Pronounced *I-D-E*.

REST API

An API (see API) that listens for and responds to HTTP requests to established endpoints. Pronounced *REST A-P-I*.

reStructuredText

A dynamic lightweight markup language associated with Python development.

RST

Abbreviation for reStructuredText.

SaaS

Acronym for *software as a service*. Subscription-licensed tools hosted in "the cloud", SaaS products require no user maintenance and provide thin-client remote access (e.g., browsers, mobile apps). Relevant examples include Wordpress.com, GitHub, Slack, and Office 365. Pronounced *sass*.

SME

Subject matter expert. Someone close to or highly familiar with the source/product who contributes this expertise to the documentation process. Please don't pronounce this *sme* or for god's sake *shme*.

topic

A discrete content item intended to be included (embedded or referenced) in parent document. See Topic-based Authoring.

QA

Initialism for *quality assurance*.

UI

Initialism for *user interface*. Pronounced *U-I*.

XML

Extensible markup language, a tag-based means of conveying semi-structured data.

XSL/XSLT

A highly configurable and extensible means of formatting, stylizing, and conveying XML data.

YAML

A format for semi-structured data. Usually preferred over JSON and XML when human reading and writing is called for, but lacking a standard means of schematizing templates. Pronounced *YAM-el*.

Appendix B: Resources

This is a curated page of resources helpful to exploring and implementing a docs-as-code approach to technical documentation. You are strongly encouraged to submit a pull request including your own favorites!

Source Control

At this point, I'm not going to list or explore source-control systems other than Git. It just doesn't seem worth it. That said, you should understand *why* source control and version control are important and the philosophy behind their use.

- Betterexplained's "A Visual Guide to Version Control"
  I wish I had read this series when I was first learning Git.

Git

- CodeSchool's Try Git tutorial
- Git Immersion
  Includes a great quickstart guide to help you get up and running on your own machine.

LML: Lightweight Markup Languages

This is one of the controversial defining factors of docs-as-code; you don't *have to* use a lightweight markup language to keep your docs in source and think programmatically. But as long as the language isn't too light to include semantic markup (such as most forms of Markdown and most Wiki MLs), it's hard to justify the added burden and expense of tag-based languages (HTML, DITA, Docbook, etc.).

AsciiDoc

I haven't tried to hide it; I love AsciiDoc, even though I recognize its limitations and don't find all of its syntax elegant or intuitive.

- Asciidoctor.org
  The prime resource for contemporary AsciiDoc, offered in Ruby and beyond.

My Favorite AsciiDoc Guides

- Using AsciiDoc and Asciidoctor to Write Documentation
  A great general purpose primer/quickstart.
- The Asciidoctor User Manual
  Truly the gold standard of language documentation. Asciidoctor himself Dan Allen is arguably the best practitioner of the language he resuscitated in just a handful of years.
- AsciiDoc Writer's Guide Another Dan Allen/Asciidoctor production. Also my favorite tech-writing style guide.

- Awesome Asciidoctor Notebook
  A compilation of excellent blog posts examining intermediate and advanced topics in AsciiDoc.

ReStructuredText

MarkDown

SSG: Static Site Generators

Jekyll

This is the only of the latest generation of static site generators I've tried seriously so far, owing largely to the jekyll-asciidoc plugin (Asciidoctor strikes again).

- The official Jekyll site
  This is a pretty great resource; I find myself using it as my primary reference.

A Note on Ruby

You could be forgiven if you've started to suspect I have a bias for applications coded in Ruby programming language. It's true: Git, Asciidoctor, and Jekyll are all Ruby gems.

However, quite unfortunately, I actually dislike Ruby and am a total newb at it. With my limited programming acumen, I find Ruby difficult to learn, though even I am able to tinker here and there. That said, these tools are all truly excellent, and there are a great many resources on Ruby, especially on StackOverflow, which is a pretty good place to start learning Ruby, and *the* place to go when you get stuck.

Conversion and Migration Tools

Pandoc

This tool is amazing. It is the near-perfect Rosetta Stone for markup languages and text-file formats, able to convert almost anything to almost anything else. It lacks any real framework and can require some heavy lifting to integrate, but Pandoc is the closest thing to docs in a box. Short of such programmatic applications, you can explore Pandoc simply by converting source files to alternate output formats from the command line.

- pandoc.org
- Building Publishing Workflows with Pandoc and Git (blog entry)

Swagger

Doxygen

Other Conversion/Migration Tools

For dealing with offline files, you really only need Pandoc, but below we share a few tools that will get you through special conversion challenges.

AsciiDoc Processor Google Docs add-on

This tool will quickly and effectively output properly-formatted AsciiDoc markup for Google Docs.

- Add-on Page

gdocs2md-html Google Docs script

This hacky-to-install but well-documented tool will convert your Google Docs handily to Markdown.

- GitHub repo

CCMS: Component Content Management Systems

Here's where things start to get hairy. The CCMS world seems to be dominated by DITA, IBM's XML-based markup language for technical documentation. Since the DITA ecosystem is mostly closed-source, it may be a nonstarter for serious DocOps-minded hackers. But I *do* want to see more holistic systems such as DITA CCMSes, which help *manage* your docs and files in ways most of the raw flat-file publishing "systems" don't even try.

In the spirit of promoting more concerted open source development in this space, I'll research and list a number of proprietary CCMS platforms here here.

Corilla

A techcomm CCMS that uses Markdown and a simple, friendly GUI for associating topics and files.

- Corilla signup

oXygen XML Editor

Hosted Documentation Platforms

AsciiBinder

An AsciiDoc-based publishing platform.

- AsciiBinder site
- GitHub repo

DocumentUp

A Markdown-based publishing platform.

- DocumentUp site
- GitHub repo

Read the Docs

A popular platform that enables reStructuredText- and Markdown-based formatting.

- Read the Docs site

GitBook

Perhaps this platform's coolest feature is the elegant editor they provide for free. It's simple but effective for writing in both Markdown and AsciiDoc.

- GitBook site
- GitBook editor

LeanPub

LeanPub is the productization of the "Lean Publishing" strategy we're basically following with this book, though *Codewriting* adds direct content contributions to the mix. LeanPub is a great way for authors to self-publish; it includes an e-commerce system that will pass along 90% of your book's revenues, which I think is unheard-of elsewhere in tech publishing.

- Leanpub site

Don't see your favorite platform here? Suggest it in the Issues for this GitHub project or in the

Blogs

Tech Writing and Docs Management

I'd Rather Be Writing

- I'd Rather Be Writing

Just Write Click

- Just Write Click

Every Page is Page One

- http://everypageispageone.com/

Read the Docs

- Read the Docs

hack.write()

- hack.write()

The Content Wrangler

Though not particularly docs-as-code oriented, this site occasionally features decent articles about tech writing strategy and tactics.

- The Content Wrangler

Git Tooling

Appendix C: Bibliography

- *Content Everywhere: Strategy and Structure for Future-Ready Content*, by Sarah Wachter-Boettcher. Rosenfeld Media. 2012.
- *Docs Like Code: Write, Review, Merge, Deploy, Repeat*, by Anne Gentle. Just Write Click. 2017.
- *Lean Publishing*, by Peter Armstrong. Leanpub. 2016.
- *Managing Enterprise Content: A Unified Content Strategy*, by Anne Rockley and Charles Cooper. New Riders. 2012.
- *Modern Technical Writing: An Introduction to Software Documentation*, by Andrew Etter. Self. 2016.

Appendix D: Cookbook of DocOps Recipes

This is a list of explainers, some of which directly document how parts of this book were made. Super meta, eh?

The Glossary Snippets

The example Glossary snippet from [managing-complexity] is called into the page with the following line of code sandwiched between the markup to start and stop a code listing:

Example — Including a subset of a source file with AsciiDoc tagging

```
----
include::../_data/glossary.yml[tags=cco-example]
----
```

Inside `_data/glossary.yml`, a section of code is marked with tags like so:

Example — Wrapping source code in AsciiDoc tags for inclusion

```
# tag::cco-example[]
  - term: complex content object
    definition: |
      Abbreviated CCO, this is an item that may be more metadata than content, stored many to a
file, as opposed to more typical text content units like chapters or topics, which may have
some metadata but are mainly text and images.
      See <<chapter-managing-complexity>>.

  - term: DITA
    definition: |
      Acronym for _Darwin Information Typing System_.
      An XML standard designed for technical writing, introduced by IBM in 2005.
      In addition to a markup standard, DITA incorporates dynamic functionality and mature
framework elements.
      https://en.wikipedia.org/wiki/Darwin_Information_Typing_Architecture[Wikipedia].

  - term: DRY
    definition: |
      For "don't repeat yourself", referring to a single-sourcing approach to code or docs.
# end::cco-example[]
```

That tag markup (`tag::cco-example[]` and `end::cco-example[]`) is prepended by the source language's comment marker (a single `#` for YAML), so AsciiDoc include tags will be invisible when the source file is called here or elsewhere in your build.

Cheesy, snooty tip title ahead.

You'll note those same tags are indicated in the `tags=cco-example` argument of the AsciiDoc include

macro, which will restrict the include to the content *between* those tags.

You were warned about the self-important tip title, right?

Recursive Meta Inclusion (forgive me)

I actually had to add the AsciiDoc tags manually to the previous example because Asciidoctor suppresses all such tags, even inside a literal block. This is probably a feature, not a bug, as there are use cases when I would want to hide nested AsciiDoc comments when calling their parent code.

Here's how my Example — Wrapping source code in AsciiDoc tags for inclusion example above is coded, with the tags written in around the include macro.

```
# tag::cco-example[]
include::../_data/glossary.yml[tags=cco-example]
# end::cco-example[]
```

That's right, I faked my own example, and I'd do it again.

Appendix E: Collaborative Authorship & Lean Publishing

This book is an experiment, and you are invited to partake.

Codewriting being developed kind of like a software product, using the approach introduced by Peter Armstrong in *Lean Publishing*. I can't put it better than Peter:

Some people are apparently making money off books on this subject, and if they have more confidence than I do that their work is an authoritative contribution, that's cool: by all means, check out their work. It's in the bibliography, and I really enjoyed reading it.

I think this field needs more adoption and propagation, for the benefit of us all, so I want to make this book as acceptable as possible. If a technical publsiher — or any publisher — is interested in putting out an edition, I would gladly consider any arrangement that allows me to maintain a fully open-source copy in my own repo. Whatever gets these ideas into more minds who can take it to new places, and hopefully bring it back around to this book.

From the README

All proposed edits and additions are eagerly welcomed.

What to Contribute

Here are some forms of content contributions I would love to receive:

quotations

> Quote yourself as if I interviewed you for three hours and kept some of your best advice. Write yourself right into the text, either with an outright quote or a paraphrase.

guest blocks

> Make a text block that conveys your commentary on a topic, in context.

Guest Block Syntax & Guidance

The two main types of block contributions are admonition blocks (either generic or branded) and guest sidebars, for longer prose.

admonition block

> You can either author a generic admonition, to be credited in the Acknowledgements and the Git repo, or you can brand an admonition with your name (or GH username) and mug. Admonition blocks should be kept to one short paragraph, at most.
>
> **generic admonition**

```
[TIP]
Here is my opinion about this topic.
```

branded admonition

```
[BRANDED.yourGHusername]
I'll make this do something cool by the time we "go to press".
```

In this case, also place a 150x150 pixel PNG file to use as an avatar for you. Make it your headshot or a caricature or some symbol you want to rep your mug. Name it `yourGHusername.png` and place it in `book-cw/images/avatars`.

guest sidebar

Make a sidebar for multi-paragraph contributions.

```
[guest_contribution]
.Your Sidebar's Clever Title
****
Here is the text of your sidebar.
Keep it witty, and remember to use one-sentence-per-line and other styles from the Style
Guide.

You can use paragraphing, images, tables, and so forth.
Just keep it tidy, witty, and informative.

-- Tag Yourself (link:https://twitter.com/@memememe[memememe])
****
```

To make these items most modular, it is best that you contribute them in their own `filename.adoc` file. Your pull request is welcome to also incporporate the `include::filename.adoc[]` macro in the place you think your content best fits. Otherwise, it's fine to leave it for me to suggest a placement.

How to Contribute

Here are the technical steps to contributing. If you don't know how to use Git or AsciiDoc yet, you may wish to **read the book before trying to contribute**. In fact, that's a good general recommendation, so you don't duplicate something that's already included, and so you can enhance existing content — even by contradicting it sensibly.

1. Fork the GitHub repo.

   [screenshot fork this repo] | *assets/images/screenshot_fork-this-repo.png*

2. Create a branch.

   If you clone your newly forked repo to your local machine (similarly to the procedure for cloning

this repo, above), use `git checkout -b new-branch`, where `new-branch` is a descriptive name for your contribution (e.g., `sidebar-hacking`).

3. Edit the appropriate AsciiDoc file, or create and properly include a new one.
4. Build locally to make sure your contribution builds as both PDF and HTML.
5. Issue a pull request to my repo.
   {codewriting_source_uri}
6. I'll review your contribution and respond to it as soon as I can.

If you wish to propose a contribution before you start writing/coding, create an Issue and label it `proposal`. I'll review it and let you know what I think.

Editorial Process

Only once we're both happy with the final state of a proposed change will I incorporate any of your work, and all contributors will be prominently credited, as well as remain in the git log for all eternity. One of the commits in your first PR should add yourself to the appropriate contributors' list in `book-cw/frontmatter/acknowledgements.adoc`.

I do reserve the right to include lessons from your contributions even if we cannot agree on the specific final text; any particular ideas reflected will be duly credited. As a journalist in my past life, I was fanatical about attribution, accuracy, and integrity in news media. As evidence, I submit this journalism guide) I helped write. I assure you I take proper representation and credit very seriously.

Plans for Codewriting

Words!

Lots more content coming, across several chapters

Slides!

I want to make a bulleted summary of each chapter/section as a "slide", which can be included in each section as well as compiled into a slide deck for presentations. I hope others will modify them to their liking and make use of them spreading the word about DocOps!

Exercises

I am working on a narrative about a docs-focused startup that hires the reader as Employee #3. Hijinks ensue.

Framework and Build

Here are some notes on what goes where and how it's all built.

```
_data: ①
  glossary.yml
_layouts: ②
assets: ③
  images:
  includes:
book-cw: ④
  index-book-cw.adoc ⑤
  _data: ⑥
    bibliography.yml
  includes: ⑦
  content-directories:
presentations: ⑧
scripts: ⑨
  liquidoxify.rb
  publish.rb
theme: ⑩
  fonts:
  pdf-theme.yml
Gemfile ⑪
README.adoc ⑫
```

① Global data source files, not specific to Codewriting book

② Liquid templates for mapping data to variables in precompiled files

③ Images, AsciiDoc includes, and other content used directly in output, possibly for multiple documents

④ Book content files; everything that goes *in* the book

⑤ Build configurations for the book's precompiled source files

⑥ Data source files for the book, for building complex content

⑦ Discrete content files (topics, source samples, etc)

⑧ Source for slide decks (bit of a mess at the moment)

⑨ The build scripts (`publish.rb` and `liquidoxify.rb`)

⑩ Files used to style output; hopefully these will evolve to consolidate across media

⑪ The project's Ruby dependency collection

⑫ Whoa, that's like, this file…

Legal Stuff

Appendix F: The Codewriting Style Guide

I realize it's a bit "meta" to include a style guide inside the only document it applies to, but I couldn't resist.

The risk of discouraging contributions by insisting on strict style/formatting requirements seems potentially self-defeating. It's not like dozens of people are clamoring to write this book, or else someone would have by now. I'd rather have highly flawed submissions than none. I realize it may not be worth learning a style just to commit a few paragraphs. That said, if you're contributing, you must be a tech writer for at least part of your job, *so show us your stuff*!

Here are the style standards employed in writing this book.

Writing Format

If you've already read the book or looked at its source, you'll know I'm writing in AsciiDoc. Tempting as it is to somehow allow contributors to at least write in their preferred language, contributions have to be done in AsciiDoc (unless you're contributing non-textually, such as images or diagrams, of course).

AsciiDoc Conventions

Asciidoctor has the best AsciiDoc and general formatting conventions I have seen.

In general, use the preferred conventions of Asciidoctor Writer's Guide.

Here are some highlights/additions:

1-sentence per line

To make a space after a sentence-terminating punctuation, use the return key instead of the spacebar. This gives you at-a-glance insight into sentence strength and paragraph structure, as well as making for easier diff reviewing.

- See "Ventilated Prose and Hanging Indentation" for more.

no hard line wrap

Some styles insist on chopping source lines at 80 characters or so. Liberate yourself! Let those lines linger.

- This one is also covered in "Ventilated Prose and Hanging Indentation".

insert classes liberally

Use the `[.classname]` delimiter tag markup for assigning a CSS class to significant blocks in AsciiDoc. Use this feature liberally but consistently, marking each block of that type and *specific kind of application or in a given series* consistently. For example, this feature is used to mark all of my

sidebars (**** wrapped) that reference a running theme, so that I can specifically style just the blocks in that series later on.

Example — Scenario-classed sideblocks

```
[.scenario.scenario-a]
****
This is some text about the ongoing scenario example about a startup narrated throughout
this book (or however I decide to use this).
****
```

This block will for instance output with the `class="sidebarblock scenario"` attribute in its HTML5 tag, which can be selected and styled (or even acted on with JavaScript) using `DIV.sidebarblock.scenario.scenario-a` (`<div class="sidebarblock scenario scenario-a">`). This is forward-thinking semantic markup; I'm not even sure how I intend to use it yet.

4-char block delimiters

As in the Asciidoc guide, do not be tempted to mess around with block delimiter markup. Always use the 4-character minimum (for example, `|===` or `----`), not anything longer. The sole exception to this is the `--` open (generic) block delimiter and the `<<<` triple-LT for page breaks.

Here are some ***exceptions*** to Asciidoctor style:

admonition format

Prepend admonitions with bracketed type selectors on their own lines (rather than inline, colon-delimited).

Example — Warming admonition style

```
[WARNING]
This is the text of a warning admonition block.
```

Style & Voice

The most important style is consistency. Feel free to bring your own voice to guest blocks, but see if you can approximate mine in the main body, or wherever you make edits.

Tech writers *do* have voices.

Appendix G: NOTICE of Packaged 3rd Party Software

The content of Codewriting is released under a Creative Commons ShareAlike 3.0 license. The broader Codewriting source contains code from the following free open-source projects, redistributed here under corresponding license.

Name	Author	License	Description
Hyde theme (Site) (Repo)	Mark Otto	The MIT License	“A brazen two-column Jekyll theme that pairs a prominent sidebar with uncomplicated content. It’s based on Poole, the Jekyll butler.”
FontAwesome (Site) (Repo)	Dave Gandy	Open Font License	“The iconic font and CSS toolkit.”
M+ OUTLINE FONTS (M+ TESTFLIGHT 058) (Site)	Coji Morishita	Unlimited	
Noto Serif Font (v2014-01-30) (Site) (Repo)	Google	Apache2.0	
Open Sans (Site)	Google	Apache2.0	