Code Writing

Documentation Ops for Agile Teams & Technical Writers

Brian Dominick

A Note to Readers & Contributors

There is no sense in postponing your realization: I have written a user manual for the role of tech writer. I've strung together a lot of prose in my lifetime, but I'm handling this book project with a type of concision I hope technical writers and software developers will appreciate: lots of bulleted section headings, lists, sidebars, examples, code listings, and summaries—not a lot of rants.

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

Collaborative Authoring

Code Writing 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 [appendix-contributing].

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

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 I have used Markdown and reStructuredText occasionally as well. The particular language should not matter for most examples, however.

Creative Commons License

I'm not trying to get rich of famous off this book, which is primarily a vehicle for learning. You can take this text and do what you want with it, so long as you leave a cookie trail of credit as described below.

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--*so long as you promise to freely share the results*.

See the rules in [creative-commons].

Introduction: 'DocOps' for Forward Thinking Technical Communications

This project is my attempt to bridge a gap I straddle between the worlds of writing and software development. I have spent my career (over two decades) in and out of media and technology, almost always mixing the two in some way, even when mixing them has not been my primary role.

What differentiates *Code Writing* from other technical writing books is that it is solidly rooted in how today's best engineering teams operate. The wisdom of technical writers, documentation managers, and project managers from lean/agile organizations is infused into this new approach.

DocOps parallels the field of "DevOps" (developer operations, engineers who specialize in making other engineers' work easier and more effective through tooling and automation). DocOps means hacking platforms to enable tech writers to be better at their jobs. To some extent this means better tooling, but it also means sensible collaboration and contribution policies, as well as a stable, predictable, and efficient workflow.

Objectives

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 communicating their product to its users;
- *establish systems* for collaborative documentation using bleeding-edge open source tools and platforms;
- *integrate documentation* into product development… and the product itself;
- *coordinate contributions* from agile developers in ways that complement rather than interfering with their preferred workflows;
- *convert legacy material* to a future-compatible system; and
- *'codify' your technical writing* by thinking like an engineer while sharing the product codebase.

Truth be told, I am researching and writing this book so I can be a better technical writer. I believe the exercise of writing this book will improve my skills in all of the above categories, as well. 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.

Open Source Centricity

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

TIP	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.

Open source means power

Inequitable distribution of power and inflexible hierarchies and workflows are hugely restricting factors. Fast-paced engineering teams have no room for externally imposed limitations. Like DevOps, DocOps 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. (More on this reasoning in)

Open source means speed

Open source means security

I think the ancient myth that exposing your source code makes you vulnerable has been successfully debunked by now.

The most important 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.

Platform Thinking

Platforms mean *distributed power*, and distributed power is key to comprehensive documentation, especially in agile environments. If you want to be successful in producing documentation for ever-changing software products, you'll need a platform solution. This book will help you think through the various options, including hybrid platforms that will scratch various itches coherently with a little finessing up front.

"Everything in Code"

This maxim has a dual meaning. First, all technical writing should be written in markup. This book also favors writing directly in markup, as opposed to using an abstraction tool backed by XML, such as Word, Google Docs, clients like oXygen, or your content management system's WYSIWYG UI. The case for this is developed in Writing (in) Code.

Second, put the docs in the product codebase. We'll discuss iteratively integrating your documentation source and platform into the repo and the product itself. This is addressed in

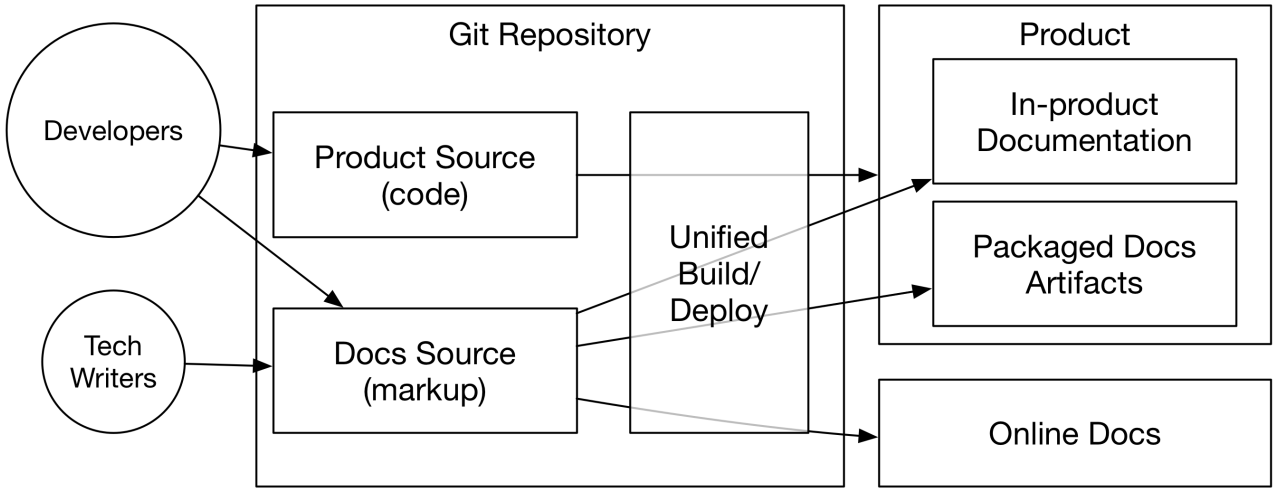

Figure 1. DocOps — General Concept

Content *Development*

You're not thinking like a developer until you look at your writing as *content development*. There's no need to claim you write software.

Lean Docs for Lean Projects

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, it is likely that you will need to iterate from a "minimum viable product" for your user manual or other documentation.

Scenario: Your New Startup—Day One

title: Scenario 1: DocuMator, Day One type: scenario tags: documator, example, exericise, scenario description: Your new job as \"Documentation Engineer\" at a SaaS startup fresh out of a much-hyped venture accelerator looks daunting. ---

You're excused for feeling intimidated your first day at DocuMator, the fledgling startup gearing up to soft-launch its eponymous product: a cloud platform for collaboratively writing and publishing product documentation. Congratulations: you're going to be documenting a product the main users of which are professional tech writers.

DocuMator is a hot company, and you're honored to be joining as a key member of the engineering organization—whether everyone on the team knows it or not. The good news is that the engineer:tech writer ratio is excellent.

Because docs are beyond critical to this product, DocuMator brought you on relatively early.

The engineering team is only five strong when you join, including the CTO/head engineer, three developers, and a DevOps specialist who will help you integrate your tools with the available infrastructure, which you are not expected to be completely familiar.

TIP	What is DocuMator?

On Day One, Shanda Abrams, the Chief Technical Officer and co-founder, reiterates the DocuMator elevator pitch she gave you during your interview. "DocuMator is a Git-backed content-management system for product documentation. It looks and feels like a CMS such as WordPress or Drupal, but it uses a flat-file backend and "

Your job, Shanda tells you, is to document their product while establishing the ideal documentation platform and workflow for the soon-to-be-growing engineering team. There is talk of eventually hiring more tech writers. "But for at least six months," Shanda says, "you should expect to be it."

Here are the milestones you have agreed to achieve.

1. Produce a 1-page HTML user guide to the minimum viable product.

   This will include:

   - account creation
   - logging in and out
   - creating documents
   - publishing content
   - managing projects
   - various other features
2. Choose a source markup language for your primary writing.

   This challenge will include your first interrogative. You will choose between:

   - Markdown
   - AsciiDoc
   - reStructuredText
   - DITA
3. Set up a simple system for sensible, collaborative internal documentation.

   This will include the second interrogative, which is arguably also the toughest. You will choose between:

   - a cloud portal/intranet/filesystem
   - a web-based content management system (CMS)
   - a wiki
   - flat files in Git

- a hybrid solution

4. Set up an integrated "changelog" for DocuMator.

   This means:

   - working with the developers to track the user-facing changes they make;
   - copy editing and formatting their notes into presentable shape;
   - integrating this source material into the documentation; and
   - making sure all release notes appear where and when they should.

5. Produce installation instructions for DocuMator Enterprise, the self-serve version of the SaaS product.

   a. Test and write up the installation procedure.

      You're old hat now, so writing up these instructions won't be a big deal. However, you'll want to reevaluate your approach given each new project's requirements. And in this case, you'll need to approach the source-info gathering process differently, since you're instructing

      - installation (not use)
      - to a different audience
      - for a different environment.

   b. Publish the DocuMator Enterprise installation guide in and HTML and PDF.

      This won't be a difficult objective, but knowing it is down the pike will inform prerequisite decisions.

6. Prepare to hire a second tech writer.

   This will mean taking some time to evaluate your documentation infrastructure. Is it ready to accommodate a newcomer? What is missing? How can you make it so they'll orientate and contribute rapidly?

The rest of this book will alternate between lessons revolving around the DocuMator-themed storyline for helping you develop into a full fledged DocOps-capable technical writer.

Part One: Writing

No matter what else you learn or forget about technical writing, remember everything you learn about *writing*. The rest changes a lot. Documentation workflows and methodologies, platforms and toolchains—all the stuff you need proficiency in to call yourself a technical writer matters not at all if you cannot write complete, accessible product explanations and instructions.

The chapters in this section 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.

Code Writing

The different "codes" we use to communicate technical concepts.

Audience and Strategy

You'll never know how to reach your audience if you don't first get to know *their challenges the way product developers do.*

Output Implications

Style and Approach

***You don't have the luxury of being able to write docs in 3 styles and let** users pick, so you pick for them.*

Documentation Anatomy

***A lot more goes into good user docs than clear instructions and illustrative** screenies.*

What documentation do users need?

Overinstruct Everything

***Finding a way to provide all the information a user may need without** overburdening them with content.*

Part Two: Coding

Writing (in) Code

***Writing docs in dynamic markup, inside the product repo, using** developers' tools.*

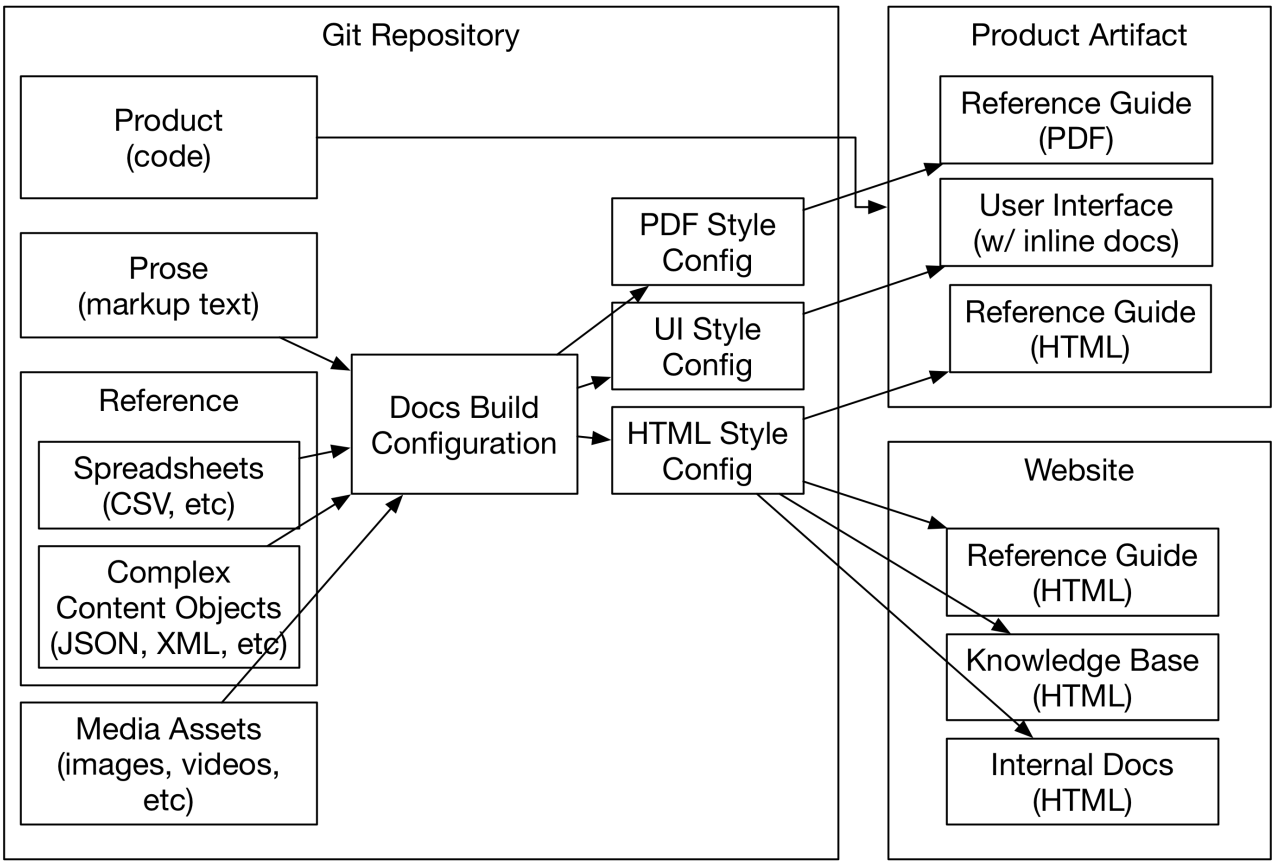

Figure 2. DocOps Overview

Writing in Lightweight Markup

You'll pretty much never see a serious software engineer *writing in* tools that heavily abstract from the code. There are exceptions to this for complex code, such as formulas and algorithms, which are perhaps better generated automatically than every character written by hand.

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 (e.g., Wordpress), contact relationship management (e.g., Salesforce), and task management (e.g., JIRA) use RDB backends. This is for good reason, as the tools require levels of abstraction that eliminate most user choice.

APIs make CMSes more interesting.

IDEs are basically reverse content management systems for source code. An IDE is a code editor that is integrated with the software environment. An IDE "knows" not just the syntax of the

language the developer is writing in (catching early generic syntax mistakes and typos); it is also configured to read variables from files outside the one you're in, as well as apply the rules of that language and environment dynamically. This helps you test and debug software as you write.

Think of well-configured IDE as spelling and grammar checking on steroids. Unlike English, computer languages are strict enough that an IDE can give far more accurate insight into code than Word is able to provide.

Docs in the Product Codebase

Writing DRY Docs

Iterate the Dynamism of Your Docs

Engineering Documentation

Think early about accommodating the engineering team's ever-growing *need for notes, specs, guides, registries, and so much more non-user-facing docs.*

Engineering departments need loads of internal documentation. The best platform for your internal docs integrates nicely with your product docs. This starts with empowering your whole team to use a language and platform that will be easy to learn and completely effective.

Style Guides & Conventions

Workflow Guides

Reference Registries

Drafting Capabilities

Knowledge Bases and Inter-team Sharing

Product Documentation

***Your team's product has a "build" process, and your docs should be an** integral part of it.*

Developer Integration

***Getting your engineers involved in thinking about DocOps, as well as** directly contributing to docs can be a big win; just before to avoid pitfalls.*

Engineers are notoriously averse to writing documentation. Except the many who are not, and do a diligent job pretty much every time. That said, If you're taking an engineering approach to the docs, engineers may be more likely to approach the docs.

Getting Engineers to Write Docs

The Balance

Too Many Cooks in the Kitchen

DocOps for non-Techwriters

Tom Sawyer the Docs Platform

Few engineers like technical writing of any kind, and fewer still want to actually document their own work. But if it looks like you're having fun putting together a documentation platform, they may grow interested in helping out.

Instructions Testing

Docs Have Bugs

You should treat error reports about your docs—from typos to downright inaccurate information—just like developers treat software bugs.

- Log them all in one place.
- Track the time it takes to resolve them.
- Trace the error back to its source.

Part Three: Publishing

Docs Hacking

***Establishing a mature, flexible toolchain that suits current and future** needs.*

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.

Platforms FTW!

Audience and Delivery

***Assess whether and how well you are reaching your audience. How** effective is your delivery? How instructive is your content?*

User Testing

Analytics

True DocOps

Making yourself useful to Product and Engineering by always being ready *to solve a documentation problem (even imperfectly).*

Solving Problems

Driving Results

Leading Content

Agility at Work

Part Four: Managing

Workflow & 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. A

Molding Processes to Work with Docs

Tech Writers Are Not Stenographers

Developers often treat tech writers like personal secretaries.

Content Management

***Grappling with version control and localization as products/features** mature, fork, internationalize, and deprecate.*

Version Entropy

Feature Status Tracking

Localization

Content Integration

The holy grail of DocOps is to enable documentation of a complex product using a single source base, without duplicate source content, while outputting in any and all necessary formats and languages. Probably none of us accomplishes this with purity, but striving for a pristinely principled DocOps environment is ideal, all else being equal.

Complex Content Objects & Extreme Reuse

Handling what I call complex content objects (CCOs) is one of the biggest challenges faced by a team dealing with lots of discrete, structured content items. Truthfully, the main kind of CCO you deal with is every page of content you write in any markup language. The markup is structuring a complex content object, even adding metadata of at least some kind (minimally dates such as created and modified).

What I mean by CCOs are all the secondary and tertiary types of content you need to maintain and reuse. Besides from pages and their subordinate content objects (such as blocks, links, and so forth), you probably need your documentation solution to handle an array of less-standard CCOs. Think of reusable content items such data you're keeping in a spreadsheet right now. If it has two or more columns, each row is a CCO, and each spreadsheet is a CCO dataset of sorts.

CCOs are typically at least one step more complex than a parameter (a key-value pair).

Example — A parameter (in YAML)

```
{
  "key-name": "parameter-value"
}
```

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

```
{
  "key-name": [
    "another-key": "that value",
    "second-child-key": 5
  ]
}
```

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 on scores of nodes at a massive datacenter. Our users can configure all of these services in myriad ways by setting key-value pairs.

Our users need reliable access to accurate documentation of our configuration settings in order to deploy and fine tune our software. Let's take a look at a configuration property's components as a YAML. 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:
      - name: "Enterprise"
        present: true
        customizations:
          override:
            deprecated: "1.6.0"
            removed:
      - name: "Developer"
        present: true
      - name: "Cloud"
        present: false
```

Even at a glance, you should be able to appreciate what makes this "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.

Considerations

Consider the following factors when solving a problem like CCO management.

authoring access

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

> In our example of a configuration parameters system, the whole point is to let engineers make necessary edits to the properties documentation source as they make changes to their impact source code.

automation

If you are integrated with a build system, it will have implications on your choice of toolchains for this function.

sorting and filtering

Will you need to re-sort CCOs by any of their parameter values? How about outputting only certain instances of a CCO type in certain places? Think again in terms of a 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 with a template). If so, you'll need to be able to effectively query the dataset, which implies more than just parsing/mapping a data dump to your desired output format.

output formats

What formats will your CCOs need to get output into?

- HTML
- Markdown
- AsciiDoc
- DITA
- Docbook
- LaTeX
- PDF
- Man
- JSON
- SQL

Your solution will need to be mappable to all the required formats. It will need to be able to export to something they can read, or external resources will need access to your CCO solution.

output languages

Internationalization support will either require another layer of development skill and effort or a budget for localization CCMS capabilities.

product versioning

If by some stroke of genius, your product does not require release version control, you may be able to get away with a conventional CMS backed by a relational database. The need to tie complex content with product version makes a flat-file approach seem necessary.

Before we settle on a flat-file approach, let's explicitly consider another option.

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.

The main problem is versioning. Source-control systems like Git don't work with relational databases, which tend to use binary files. Git has no insight into the structure of your conventional database file, so it cannot track changes made to the data. Therefore, an RDB will also require specific version tagging of version-specific data, a layer of manual complexity that is fairly prohibitive.

Another limitation of RDBs is integrating them into your codebase. Any open source project can be set up inside your repo, but truly integrating such a product can add extra challenges.

If SQL and an RDB can solve your needs, go for it. There is rarely any advantage to adding complexity or setup time to a project without an immediate or near-term need. SQL databases are highly portable.

DITA CCMSes

It is important to note that DITA's specializations are basically CCOs: XML schemas with enforceable rules.

A flat-file solution

Here is how I am solving this problem at Rocana.

Team Leadership

Coordinating and sharing responsibilities among tech writers.

Scaling Strategies

***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.*

Appendix A: Bibliography

- [modern] Andrew Etter. Modern Technical Writing: An Introduction to Software Documentation, Kindle Edition. Self-published. 2016.

Appendix B: Collaborative Authorship & Lean Publishing

This book is an experiment.

This book is being developed kind of like a software product, using the approach introduced by Peter Armstrong in *Lean Publishing*. If you wish to contribute to this book, I strongly recommend you make a donation to Peter's project and at least skim his excellent book.

Appendix C: The 'Code Writing' Style Guide

Here are the style standards employed in writing this book.

At this point, discouraging contributions with strict style/formatting requirements seems potentially self-defeating. 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 should be a tech writer, so show us your stuff.

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 via images or diagrams, of course).

AsciiDoc Conventions

Asciidoctor has the best AsciiDoc and general formatting conventions I have seen.

> **TIP** In general, use the preferred conventions of Asciidoctor Writer's Guide.

Here are some highlights/additions:

no hard line wrap

Let those lines linger. This is way better for Git.

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.

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 (`**` delimited) that reference my running scenario, so that I can specifically style just the blocks in that series later on.

Example — Scenario-classed sideblocks

```
[.scenario]
****
This is some text about the ongoing scenario example about a startup narrated
throughout this book.
****
```

This block will output with the `class="sidebarblock scenario"` attribute in its HTML5 tag, which can be selected and styled (or even acted on with JavaScript) using `sidebarblock.scenario`. This is forward-thinking, code-oriented content development.

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.

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.
```

Writing Style

The most important style is consistency, so

Appendix D: Glossary

DITA

Acronym for *Darwin Information Typing System*. An technical writing-oriented XML standard introduced by IBM in 2005. Pronounced *DII-tuh*. Wikipedia.

GUI

Acronym for *graphical user interface*. Pronounced *GOO-ee*.

UI

Initialism For *user interface*. Pronounced *U-I*.

Appendix E: Creative Commons License

Appendix F: Codebase Licensing NOTICE